Linux Kernel and Booting

Ishtiyaque Ahmad
Lecturer, CSE, BUET

Introduction

- Linux is an open source UNIX like operating system.
- Though initially it was developed only for x86 architecture, now it has compatibility to run on almost all architectures.
- Some important features of the kernel:
  - Monolithic kernel
  - Six primary subsystems:
    - Process management
    - Interprocess communication
    - Memory management
    - File system management
    - I/O management
    - Networking
- Mostly written in C
- Some Parts are written in assembly
- Default compiler is GCC
- It is generally assumed that the community of Linux kernel developers is composed by 5000 or 6000 members.
- As of 2013, the 3.10 release of the Linux kernel had 15,803,499 lines of code.

Downloading the kernel

- The current Linux source code is always available in both a complete tarball and an incremental patch from the official home of the Linux kernel, http://www.kernel.org.
- Uncompress the kernel:

  $ tar xvzf linux-x.y.z.tar.gz

  This uncompresses and untars the source to the directory linux-x.y.z.

The kernel source is typically installed in /usr/src/linux. Note that you should not use this source tree for development. The kernel version that your C library is compiled against is often linked to this tree. Besides, you do not want to have to be root to make changes to the kernel. Instead, work out of your home directory and use root only to install new kernels. Even when installing a new kernel, /usr/src/linux should remain untouched.

Building the Kernel (Tools Required)

- Only three packages are needed in order to successfully build a kernel: a compiler, a linker, and a make utility.
  - Compiler: To build the kernel, the gcc C compiler must be used. The required version of gcc for compiling a particular kernel can be found at /Documentation/Changes. The available version of gcc in the system can be retrieved by the command $ gcc --version
  - Linker: The C compiler, gcc, does not do all of the compiling on its own. It needs an additional set of tools known as binutils to do the linking and assembling of source files. The binutils package also contains useful utilities that can manipulate object files in lots of useful ways, such as to view the contents of a library. The following command can be used to know the current version of binutils in the system: $ ld -v
  - Make: make is a tool that walks the kernel source tree to determine which files need to be compiled, and then calls the compiler and other build tools to do the work in building the kernel. The kernel requires the GNU version of make, which can usually be found in a package called make for a particular distribution. The current version of make can be known by: $ make --version
  - Other requirements can be found in *linux/Documentation/Changes*

Building the Kernel (Configuration)

- The kernel configuration is kept in a file called .config in the top directory of the Kernel source tree. If you have just expanded the kernel source code, there will be no .config file, so it needs to be created.
- The most basic method of configuring a kernel is to use the make config method:

```
$ make config
```

- The kernel configuration program will step through every configuration option and ask whether to enable this option or not.
- Every kernel version comes with a “default” kernel configuration. To create this default configuration, the following command should be used:

```
$ make  defconfig
```

- The menuconfig way of configuring a kernel is a console-based program that offers a way to move around the kernel configuration using the arrow keys on the keyboard. To start up this configuration mode, enter:

```
$ make menuconfig
```

Kernel Modules

- If you want to add code to a Linux Kernel, the most basic way to do that is to add some source files to the kernel source tree and recompile the kernel. In fact, the kernel configuration process consists mainly of choosing which files to include in the kernel to be compiled.
- But you can also add code to the Linux kernel while it is running. A chunk of code that you add in this way is called a Kernel Module.
- Modules are pieces of code that can be loaded and unloaded into the kernel upon demand.
- They extend the functionality of the kernel without the need to reboot the system.
- These modules can do lots of things, but they typically are one of three things: 1) device drivers; 2) filesystem drivers; 3) system calls.
- The kernel isolates certain functions, including these, especially well so they don't have to be intricately wired into the rest of the kernel.
- Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.
- Another advantage is that Kernel Modules help you diagnose system problems. A bug in a device driver which is bound into the kernel can stop your system from booting at all. And it can be really hard to tell which part of the base kernel caused the trouble. If the same device driver is a Kernel Module, the base kernel may be up and running before the device driver even gets loaded. If your system dies after the base kernel is up and running, it's an easy matter to track the problem down to the trouble-making device driver and just not load that device driver until you fix the problem.

Kernel Modules

- Kernel Modules can save you memory, because you have to have them loaded only when you're actually using them. All parts of the base kernel stay loaded all the time. And in real storage, not just virtual storage.
- Sometimes you *have* to build something into the base kernel instead of making it a module. Anything that is necessary to get the system up far enough to load modules must obviously be built into the base kernel. For example, the driver for the disk drive that contains the root filesystem must be built into the base kernel.
- There is a tendency to think of Kernel Modules like user space programs. They do share a lot of their properties, but modules are definitely not user space programs. They are part of the kernel. As such, they have free run of the system and can easily crash it.

Build and Install

- The kernel can be built using the following command:

  $ make

- To install the modules, we have to enter:

  $make modules_install

This will install all the modules that have been built and place them in the proper location in the filesystem for the new kernel to properly find. Modules are placed in the /lib/modules/kernel_version directory, where kernel_version is the kernel version of the new kernel that has just been built.

- After the modules have been successfully installed, the main kernel image must be installed:

  $make install

This will kick off the following process:

1. The kernel build system will verify that the kernel has been successfully built.

2. The build system will install the static kernel portion into the/boot directory and name this executable file based on the kernel version of the built kernel.

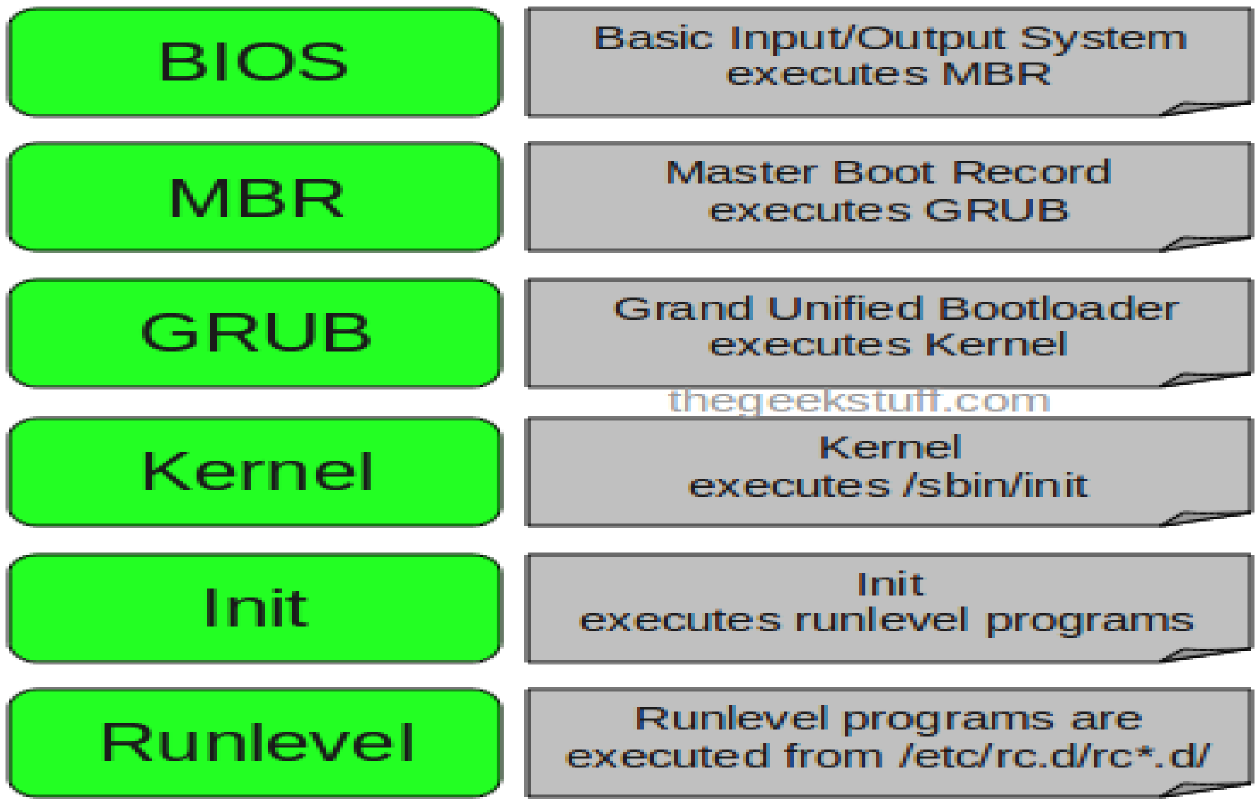
3. Any needed initial ramdisk images will be automatically created, using the modules that have just been installed during the modules_install phase.

- You may have to update the bootloader (Typically grub) so that the newly installed kernel is shown in the list of bootable kernels.

Booting a Linux Kernel

Boot Process

- When the computer is switched on, it's of no use because the data stored in the memory(RAM) is garbage and there is no Operating System running. The first thing motherboard does is to initialize its own firmware, the chipset and tries to get the CPU running
- The CPU then starts executing BIOS code, and follows some steps to complete the boot process.

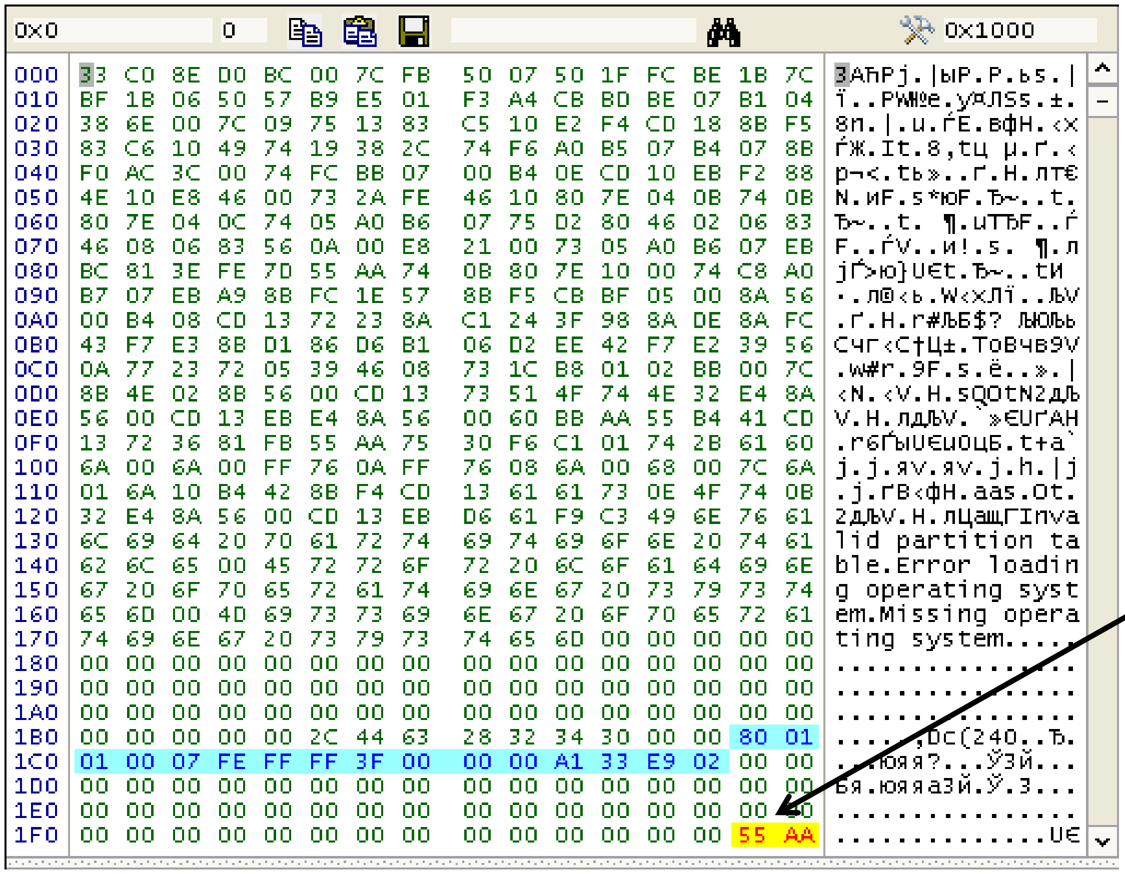

BIOS
Basic Input/Output System executes MBR
MBR
Master Boot Record executes GRUB
GRUB
Grand Unified Bootloader executes Kernel
thegeekstuff.com
Kernel
Kernel executes /sbin/init
Init
Init executes runlevel programs
Runlevel
Runlevel programs are executed from /etc/rc.d/rc*.d/

BIOS

- BIOS performs a Power-On Self-Test (POST) for all of the different hardware components in the system to make sure everything is working properly
- Retrieves information from CMOS (Complementary Metal-Oxide Semiconductor) a battery operated memory chip on the motherboard that stores time, date, and critical system information.
- It looks for a valid boot loader in the first sector of floppy, CD-ROM, or hard drive.
- Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.

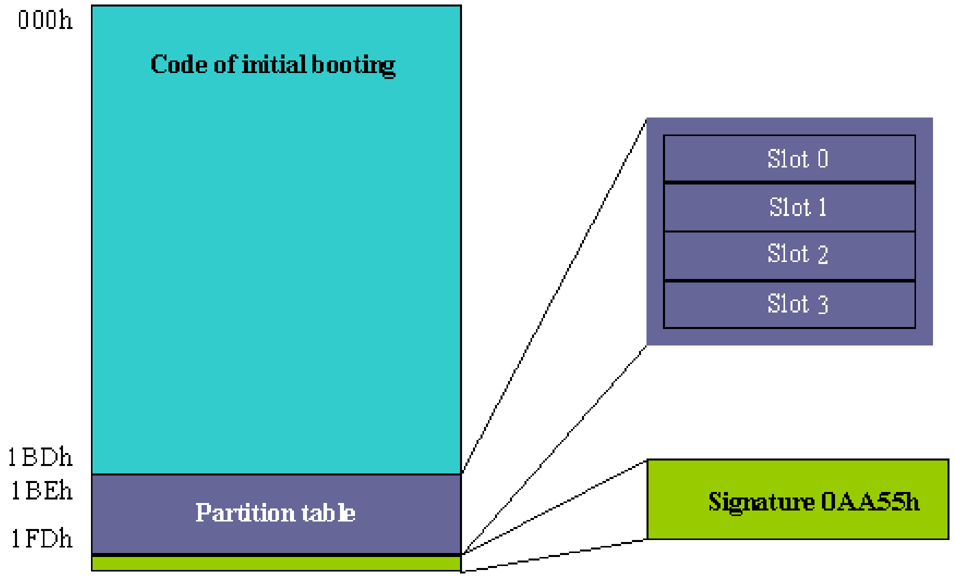
BIOS

How to detect “valid” Bootloader?



```
000  33 C0 8E D0 BC 00 7C FB  50 07 50 1F FC BE 1B 7C  3AЋPj.|ыP.P.ьs.|
010  BF 1B 06 50 57 B9 E5 01  F3 A4 CB BD BE 07 B1 04  ї..PW№e.у¤ЛЅѕ.±.
020  38 6E 00 7C 09 75 13 83  C5 10 E2 F4 CD 18 8B F5  8n.|.u.ѓЕ.вфН.‹х
030  83 C6 10 49 74 19 38 2C  74 F6 A0 B5 07 B4 07 8B  ѓЖ.It.8,tц µ.ґ.‹
040  F0 AC 3C 00 74 FC BB 07  00 B4 0E CD 10 EB F2 88  р¬<.tь»..ґ.Н.лт€
050  4E 10 E8 46 00 73 2A FE  46 10 80 7E 04 0B 74 0B  N.иF.s*юF.Ђ~..t.
060  80 7E 04 0C 74 05 A0 B6  07 75 D2 80 46 02 06 83  Ђ~..t. ¶.uТЂF..ѓ
070  46 08 06 83 56 0A 00 E8  21 00 73 05 A0 B6 07 EB  F..ѓV..и!.s. ¶.л
080  BC 81 3E FE 7D 55 AA 74  0B 80 7E 10 00 74 C8 A0  јЃ>ю}UЄt.Ђ~..tИ 
090  B7 07 EB A9 8B FC 1E 57  8B F5 CB BF 05 00 8A 56  ·.л©‹ь.W‹хЛї..ЉV
0A0  00 B4 08 CD 13 72 23 8A  C1 24 3F 98 8A DE 8A FC  .ґ.Н.r#ЉБ$?˜ЉЮЉь
0B0  43 F7 E3 8B D1 86 D6 B1  06 D2 EE 42 F7 E2 39 56  Cчг‹С†Ц±.ТоBчв9V
0C0  0A 77 23 72 05 39 46 08  73 1C B8 01 02 BB 00 7C  .w#r.9F.s.ё..».|
0D0  8B 4E 02 8B 56 00 CD 13  73 51 4F 74 4E 32 E4 8A  ‹N.‹V.Н.sQOtN2дЉ
0E0  56 00 CD 13 EB E4 8A 56  00 60 BB AA 55 B4 41 CD  V.Н.лдЉV.`»ЄUґAН
0F0  13 72 36 81 FB 55 AA 75  30 F6 C1 01 74 2B 61 60  .r6ЃыUЄu0цБ.t+a`
100  6A 00 6A 00 FF 76 0A FF  76 08 6A 00 68 00 7C 6A  j.j.яv.яv.j.h.|j
110  01 6A 10 B4 42 8B F4 CD  13 61 61 73 0E 4F 74 0B  .j.ґB‹фН.aas.Ot.
120  32 E4 8A 56 00 CD 13 EB  D6 61 F9 C3 49 6E 76 61  2дЉV.Н.лЦaщГInva
130  6C 69 64 20 70 61 72 74  69 74 69 6F 6E 20 74 61  lid partition ta
140  62 6C 65 00 45 72 72 6F  72 20 6C 6F 61 64 69 6E  ble.Error loadin
150  67 20 6F 70 65 72 61 74  69 6E 67 20 73 79 73 74  g operating syst
160  65 6D 00 4D 69 73 73 69  6E 67 20 6F 70 65 72 61  em.Missing opera
170  74 69 6E 67 20 73 79 73  74 65 6D 00 00 00 00 00  ting system.....
180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1B0  00 00 00 00 00 2C 44 63  28 32 34 30 00 00 80 01  .....,Dc(240..Ђ.
1C0  01 00 07 FE FF FF 3F 00  00 00 A1 33 E9 02 00 00  ...юяя?...Ў3й...
1D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  Бя.юяяa3й.Ў.3...
1E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 AA  ..............UЄ
```

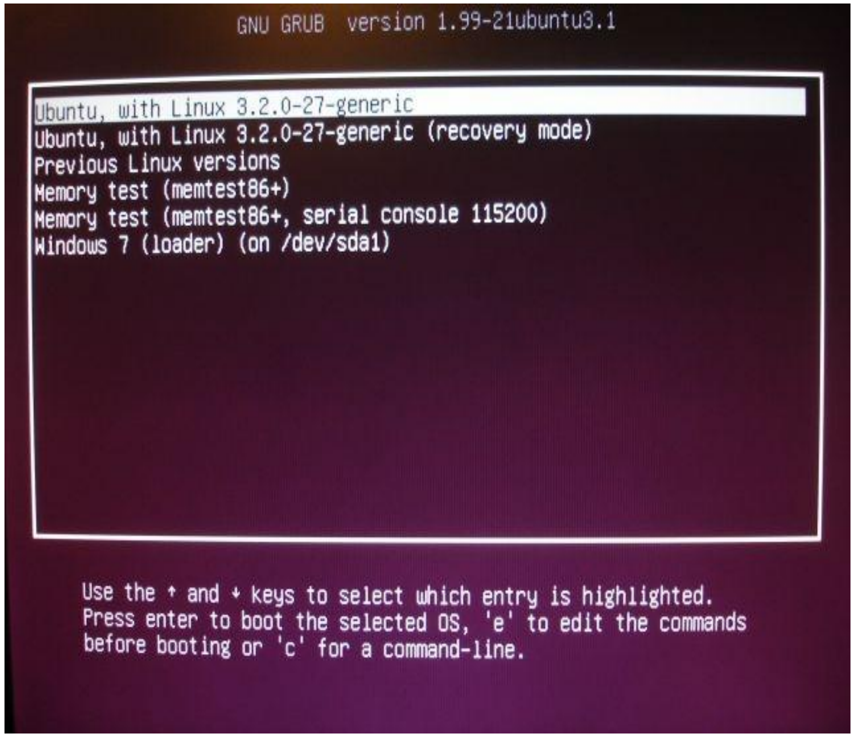
MBR (Master Boot Record)



- It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
- After the MBR is loaded into the memory, CPU starts executing it from the beginning.
- MBR is 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) MBR validation check in last 2 bytes.
- Usually an Operating System doesn’t fit in such a little space. So MBR usually contains information about the jump address of another more powerful bootloader like GRUB (or LILO in old systems).
- So, in simple terms MBR loads and executes the GRUB boot loader.

GRUB (GRand Unified Bootloader)

- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- GRUB displays a splash screen, waits for few seconds, if you don’t enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem (the older Linux loader LILO didn’t understand filesystem).
- Loads the kernel and initramfs into memory



```
GNU GRUB  version 1.99-21ubuntu3.1

Ubuntu, with Linux 3.2.0-27-generic
Ubuntu, with Linux 3.2.0-27-generic (recovery mode)
Previous Linux versions
Memory test (memtest86+)
Memory test (memtest86+, serial console 115200)
Windows 7 (loader) (on /dev/sda1)

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, 'e' to edit the commands
before booting or 'c' for a command-line.
```

Kernel

- Initializes devices and loads initrd module
- Mounts the root file system as specified in the “root=” in grub.conf
- Kernel executes the /sbin/init program
- Since init is the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1
- initrd stands for Initial RAM Disk.
- initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted.
- It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

init

- The kernel, once it is loaded, finds *init* in sbin(*/sbin/init*) and executes it.
- This init process reads */etc/inittab* file and sets the path, starts swapping, checks the file systems, and so on.
- It runs all the boot scripts(/etc/rc.d/*,/etc/rc.boot/*)
- starts the system on specified run level in the file /etc/inittab
- Following are the available run levels
  - 0 – halt
  - 1 – Single user mode
  - 2 – Multiuser, without NFS
  - 3 – Full multiuser mode
  - 4 – unused
  - 5 – X11
  - 6 – reboot
- *init* identifies the default initlevel from /etc/inittab and uses that to load all appropriate programmes

N.B. Path may vary according to distributions

Runlevel

- When the Linux system is booting up, various services get started. For example, it might say "starting sendmail …. OK". Those are the runlevel programs, executed from the run level directory as defined by the kernel run level.
- Depending on default init level setting, the system will execute the programs from the directory /etc/rcX.d, where X is the runlevel .
- Under the /etc/rcX.d/ directories, there are programs that start with S and K.
- Programs starts with S are used during startup. S for startup.
- Programs starts with K are used during shutdown. K for kill.
- There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
- For example, S12syslog is to start the syslog daemon, which has the sequence number of 12. S80sendmail is to start the sendmail daemon, which has the sequence number of 80. So, syslog program will be started before sendmail.
- **N.B. Path may vary according to distributions**

Exploring the Kernel

Directories in the Kernel Source Code

- arch
  The arch subdirectory contains all of the architecture specific kernel code. It has further subdirectories, one per supported architecture, for example i386 and alpha.
- crypto
  Codes related to cryptography reside here.
- drivers
  All of the system's device drivers live in this directory. They are further sub-divided into classes of device driver, for example block.
- fs
  Code related to specific file systems reside here.
- include
  The include subdirectory contains most of the include files needed to build the kernel code. It too has further subdirectories including one for every architecture supported. The include/asm subdirectory is a soft link to the real include directory needed for this architecture, for example include/asm-i386. To change architectures you need to edit the kernel makefile and rerun the Linux kernel configuration program.
- init
  This directory contains the initialization code for the kernel and it is a very good place to start looking at how the kernel works.
- ipc
  This directory contains the kernels inter-process communications code.

Directories in the Kernel Source Code

- kernel

  The main kernel code. Again, the architecture specific kernel code is in arch/*/kernel.
- lib

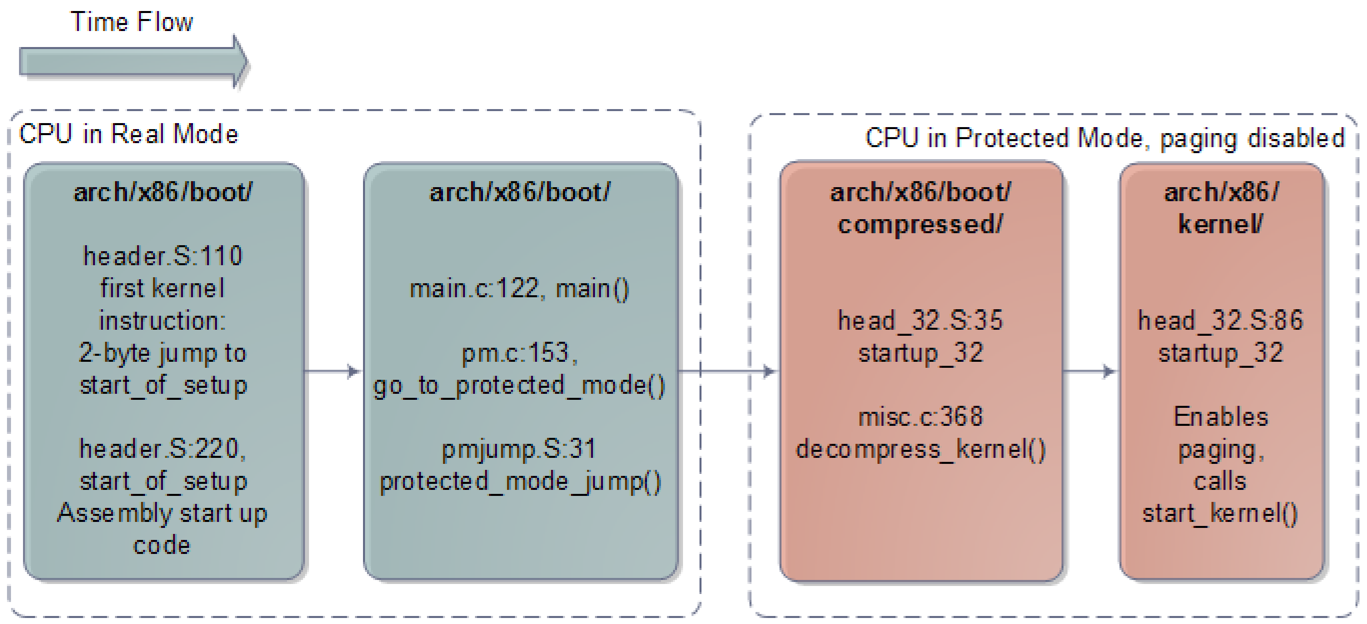
  This directory contains the kernel's library code. The architecture specific library code can be found in arch/*/lib/.
- mm

  This directory contains all of the memory management code. The architecture specific memory management code lives down in arch/*/mm/, for example arch/i386/mm/fault.c.
- net

  The kernel's networking code.
- scripts

  This directory contains the scripts (for example awk and tk scripts) that are used when the kernel is configured.

Flow of the Kernel code


Time Flow
CPU in Real Mode
arch/x86/boot/
header.S:110 first kernel instruction: 2-byte jump to start_of_setup
header.S:220, start_of_setup Assembly start up code
arch/x86/boot/
main.c:122, main()
pm.c:153, go_to_protected_mode()
pmjump.S:31 protected_mode_jump()
CPU in Protected Mode, paging disabled
arch/x86/boot/ compressed/
head_32.S:35 startup_32
misc.c:368 decompress_kernel()
arch/x86/ kernel/
head_32.S:86 startup_32
Enables paging, calls start_kernel()

Look at the kernel source code according to the above flow diagram.