What is initrd image in Linux

The initial RAM disk (initrd) is an initial root file system that is mounted prior to when the real root file system is available. The initrd is bound to the kernel and loaded as part of the kernel boot procedure. The kernel then mounts this initrd as part of the two-stage boot process to load the modules to make the real file systems available and get at the real root file system.

initrd provides the capability to load a RAM disk by the boot loader. This RAM disk can then be mounted as the root filesystem and programs can be run from it. Afterwards, a new root file system can be mounted from a different device. The previous root (from initrd) is then moved to a directory and can be subsequently unmounted.

How initrd works

initrd provides the capability to load a RAM disk by the bootloader. This RAM disk can then be mounted as the root fileystem and programs can be run from it. Afterwards, a new root file system can be mounted from a different device. Theprevious root (from initrd) is then moved to a directory andcan be subsequently unmounted. initrd is mainly designed to allow system startup to occur in two phases, where the kernel comes up with a minimum set of compiled-in drivers, and where additional modules are loaded from initrd.

When using initrd, the system typically boots as follows:

1. The boot loader loads the kernel and the initial RAM disk
2. The kernel converts initrd into a "normal" RAM disk and frees the memory used by initrd
3. initrd is mounted read-write as root
4. /linuxrc is executed (this can be any valid executable, including shell scripts; it is run with uid 0 and can do basically everything init can do)
5. linuxrc mounts the "real" root file system
6. linuxrc places the root file system at the root directory using the pivot_root system call
7. The usual boot sequence (e.g. invocation of /sbin/init) is performed on the root file system
8) The initrd file system is removed

Note

That changing the root directory does not involve unmounting it. It is therefore possible to leave processes running on initrd during that procedure. Also note that file systems mounted under initrd continue to be accessible.

Boot Loaders

What happens when you turn on a computer system? At the most basic level, we know that the computer's components power on, and the operating system "boots up" so users have a way of interacting with the systems – whether that is to play games, run a web server, set up in-depth applications or otherwise. But how does a computer know *what* to boot? If a processor pulls data from the system's memory, how can it work with a freshly booted computer that does not have any processes within its memory stores? For this, the computer uses a *boot loader*, which is a small amount of code designed to prepare the system and then pass it to the more complex kernel, which in turn manages the operating system itself.

Boot loaders live at a predefined location within a hard drive (persistent memory); it is from this location that code is pulled into random-access memory for the processor to run. From here, the boot loader prepares the system environment, then either hands off control to the kernel or a *boot sector*. From the kernel, the operating system runs; from the boot sector, the new boot loader prepares the system for the next stage of the boot process – whether this is another boot sector or the final process to run in the chain. Having the boot loader taken over by another booting process one or more times is called *chain loading*.

Certain boot loaders, particularly those used by Linux distributions, are often capable of *dual* or *multi-booting*. This is used when one computer has multiple operating systems from which to boot. These boot launchers often feature simple terminal-style interfaces from which to select an operating system.

Master Boot Record

Initially, the *master boot partition* organizational schema was the only option for booting into Linux (and other) systems. This is the boot section at the beginning of a computer partition that contains the executable code for booting into the operating system. The master boot record can work with up to four partitions with boot disks up to two terabytes in size*. These limitations are removed when using the alternative *GUID partition table*.

GUID Partition Table

The GUID partition table is a UEFI partition organizational schema and can coexist with MBR partitions. Unlike MBR, it does not contain partition amount or size limitations. Because of this, some BIOS systems use GPT, and the greater technological community tends to be moving away from MBR.

Many Linux-compatible boot loaders work with both MBR and GPT, including *GNU GRUB*, an oft-used boot loader.

GNU GRUB

A common boot loader among Linux users and distributions is GRUB, created by the GNU Project, with *GRUB 2* being the most-recent version of GRUB, and GRUB 1 considered the legacy version. GRUB 1 is no longer under active development and is being phased out of usage.

GRUB uses chain loading with its two-step loading process. Step 1, located in the MBR, loads the first section of *core.img* and contains what is needed to boot into GRUB. Stage 1.5 finishes loading the image. The second stage, finally, boots GRUB from the */boot* menu and prompts the user in a terminal interface to select an operating system. Once selected, GRUB hands off the system to the kernel.

As implied, GRUB features multi-boot, in line with the Free Software Foundation's Multiboot Specification. This allows the user to boot from any number of compliant operating systems and makes switching between systems fairly easy.

GRUB also provides users with a BASH-like interface to interact with the subsystem, allowing users to have more control over GRUB usage and configuration.

Boot loaders, overall, can be as simple as immediately passing control to a kernel or almost as complex as certain operating systems it is made to launch; GRUB, for example, is particularly complex as a boot loader, whereas others, such as gummiboot, focus on being configuration-less and "just working." And boot loaders for Linux do not stop with GRUB, as many systems still use LILO (the Linux Loader) or others. With a selection of boot loaders available – and even the possibility of making your own – there are options available for those looking to configure their system head-to-toe, or even just explore the possibilities.

** Note that this does not prevent the system from accessing hard drives larger than two terabytes in size. It is only the boot partition that cannot be excess of 2TB.*

GRUB vs LILO

For a computer to run after being turned on, it needs several software programs to help it. One of these software programs is the bootstrap loader or boot loader. It loads the main operating system of a computer.

Operating system programs are stored in hard drives; CD, DVD, USD flash drive, floppy disk, and flash memory card and cannot be accessed by the computer's central processor because it can only execute programs found in ROM.
Those found in RAM need to be accessed with the help of the boot loader such as BIOS, EFI, SLOF, OpenBoot, OpenBIOS, BOOTMGR, Syslinux, NTLDR, GRUB, and LILO. These programs allow the computer to communicate with its user.

GRUB is Grand Unified Bootloader which can boot operating systems developed by Linux, Mach4, vSTA, DOS, and many other operating systems. It can load kernels in different binary formats which are entered in a known state making it easy for new users.

There are several options in the config file, and it allows users to multiboot and use the operating systems that are installed in the computer and choose which kernel config to use in an operating system's partition.
It supports multiple executable formats and does not need geometry translation. It has a bash-like command prompt which enables users to boot up an installed operating system from a floppy disk, CD-ROM, or USD device.
LILO, on the other hand, is a generic boot loader for Linux. It is the code which BIOS loads into the computer memory at start up. Like GRUB, it can boot an operating system from an external source like a floppy disk or hard disk.
It can write a Master Boot Record (MBR) on a device and can locate kernels, load them in memory, and start them up. It allows users to start DOS, Windows, OS/2, and other configs from Linux. It was the default boot loader of Linux until it was replaced by GRUB.
Unlike GRUB, LILO does not allow booting from a network and needs to be reinstalled to the MBR after the configuration file is altered while GRUB automatically defaults to its command line interface. LILO is easier to use than GRUB, though, because it is simpler.

Summary:

1.GRUB is a boot loader which can be used for Linux, vSTA, DOS, and other operating systems while LILO is a generic boot loader for Linux.
2.Both GRUB and LILO can boot operating systems from external devices such as floppy disks and hard drives, but 3.GRUB allows booting from a network while LILO does not.
4.When the configuration file is altered, LILO needs to be reinstalled to the MBR while GRUB defaults to its command line interface.
5.GRUB is more complicated to use while LILO is simpler and easier to use.
6.LILO is the old default boot loader for Linux while GRUB is the new default boot loader.
7.GRUB can be used for various other operating systems unlike LILO which is used only for Linux operating systems.

What is meant by Compilation?

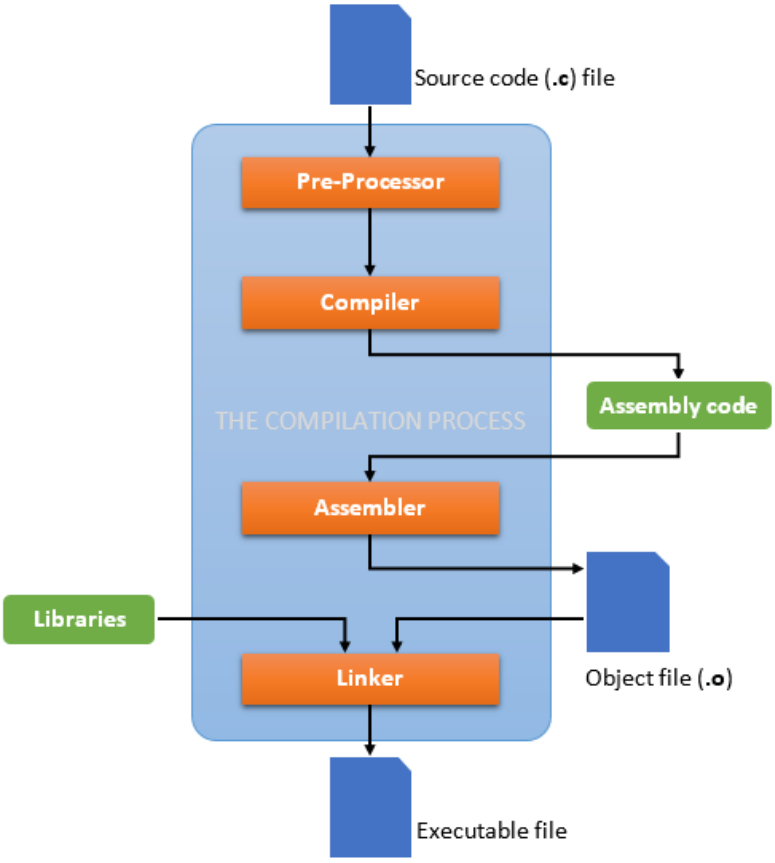
The process of translating source code written in high level to low level machine code is called as Compilation. The compilation is done by a special software known as compiler. The compiler checks source code for any syntactical or structural errors and generates object code with extension **.obj** (in Windows) or **.o** (in Linux) if source code is error free.

The C compilation

The entire C compilation is broken to four stages.

1. Pre-processing
2. Compilation
3. Assembling and
4. Linking

The below image describes the entire C compilation process.



Knowing how compilation works can be very helpful both when writing code and debugging. Compiling a C program is a multi-stage process. At an overview level, the process can be split into four separate stages: Preprocessing, compilation, assembly, and linking. Traditional C compilers orchestrate this process by invoking other programs to handle each stage.

In this post, I'll walk through each of the four stages of compiling the following C program:

```
/*
 * "Hello, World!": A classic.
 */

#include <stdio.h>

int
main(void)
{
        puts("Hello, World!");
        return 0;
}
```

Preprocessing

The first stage of compilation is called preprocessing. In this stage, lines starting with a # character are interpreted by the *preprocessor* as *preprocessor commands*. These commands form a simple macro language with its own syntax and semantics. This language is used to reduce repetition in source code by providing functionality to inline files, define macros and to conditionally omit code.

Before interpreting commands, the preprocessor does some initial processing. This includes joining continued lines (lines ending with a `\`) and stripping comments.

To print the result of the preprocessing stage, pass the `-E` option to `cc`:

```
cc -E hello_world.c
```

Given the “Hello, World!” example above, the preprocessor will produce the contents of the `stdio.h` header file joined with the contents of the `hello_world.c` file, stripped free from its leading comment:

```
[lines omitted for brevity]

extern int __vsnprintf_chk (char * restrict, size_t,
       int, size_t, const char * restrict, va_list);
# 493 "/usr/include/stdio.h" 2 3 4
# 2 "hello_world.c" 2

int
main(void) {
 puts("Hello, World!");
 return 0;
}
```

Compilation

The second stage of compilation is confusingly enough called compilation. In this stage, the preprocessed code is translated to *assembly instructions* specific to the target processor architecture. These form an intermediate human readable language.

The existence of this step allows for C code to contain inline assembly instructions and for different *assemblers* to be used.

Some compilers also supports the use of an integrated assembler, in which the compilation stage generates *machine code* directly, avoiding the overhead of generating the intermediate assembly instructions and invoking the assembler.

To save the result of the compilation stage, pass the `-S` option to `cc`:

```
cc -S hello_world.c
```

This will create a file named `hello_world.s`, containing the generated assembly instructions. On Mac OS 10.10.4, where `cc` is an alias for `clang`, the following output is generated:

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 10
    .globl  _main
```

```
    .align  4, 0x90
_main:                                  ## @main
    .cfi_startproc
## BB#0:
    pushq   %rbp
Ltmp0:
    .cfi_def_cfa_offset 16
Ltmp1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
Ltmp2:
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    leaq    L_.str(%rip), %rdi
    movl    $0, -4(%rbp)
    callq   _puts
    xorl    %ecx, %ecx
    movl    %eax, -8(%rbp)          ## 4-byte Spill
    movl    %ecx, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc

    .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
    .asciz  "Hello, World!"


.subsections_via_symbols
```

Assembly

During the assembly stage, an assembler is used to translate the assembly instructions to machine code, or *object code*. The output consists of actual instructions to be run by the target processor.

To save the result of the assembly stage, pass the `-c` option to `cc`:

```
cc -c hello_world.c
```

Running the above command will create a file named `hello_world.o`, containing the object code of the program. The contents of this file is in a binary format and can be inspected using `hexdump` or `od` by running either one of the following commands:

```
hexdump hello_world.o
od -c hello_world.o
```

Linking

The object code generated in the assembly stage is composed of machine instructions that the processor understands but some pieces of the program are out of order or missing. To produce an executable program, the existing pieces have to be rearranged and the missing ones filled in. This process is called linking.

The *linker* will arrange the pieces of object code so that functions in some pieces can successfully call functions in other pieces. It will also add pieces containing the instructions for library functions

used by the program. In the case of the “Hello, World!” program, the linker will add the object code for the `puts` function.

The result of this stage is the final executable program. When run without options, `cc` will name this file `a.out`. To name the file something else, pass the `-o` option to `cc`:

```
cc -o hello_world hello_world.c
```

https://opensourceforu.com/2011/01/understanding-a-kernel-oops/

Definition - What does Direct Memory Access (DMA) mean?

Direct memory access (DMA) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations. The process is managed by a chip known as a DMA controller (DMAC).

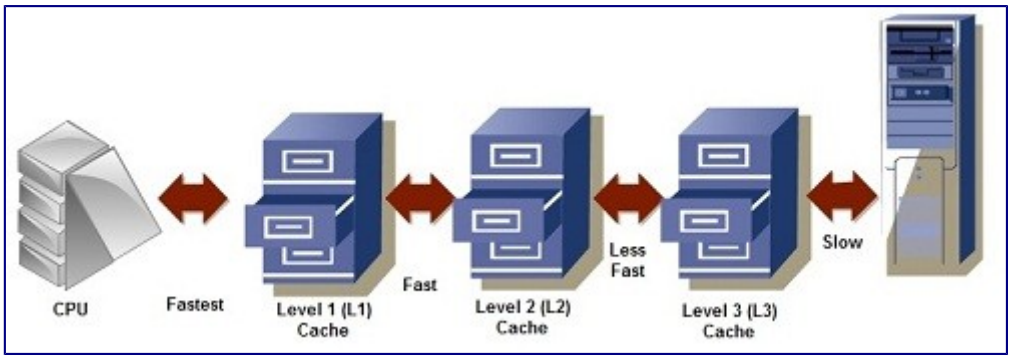
Cache definition : The *Cache Memory* (Pronounced as "cash") is the volatile computer memory which is very nearest to the CPU so also called **CPU memory**, all the Recent Instructions are Stored into the Cache Memory. It is the fastest memory that provides high-speed data access to a computer microprocessor. Cache meaning is that it is used for storing the input which is given by the user and which is necessary for the computer microprocessor to Perform a Task. But the Capacity of the Cache Memory is too low in compare to Memory (random access memory (RAM)) and Hard Disk.

Importance of Cache memory

The cache memory lies in the path between the processor and the memory. The cache memory therefore, has lesser access time than memory and is faster than the main memory. A cache memory have an access time of 100ns, while the main memory may have an access time of 700ns.

The cache memory is very expensive and hence is limited in capacity. Earlier cache memories were available separately but the microprocessors contain the cache memory on the chip itself.

The need for the cache memory is due to the mismatch between the speeds of the main memory and the CPU. The CPU clock is very fast, whereas the main memory access time is comparatively slower. Hence, no matter how fast the processor is, the processing speed depends more on the speed of the main memory (the strength of a chain is the strength of its weakest link). **It is because of this reason that a cache memory having access time closer to the processor speed is introduced.**



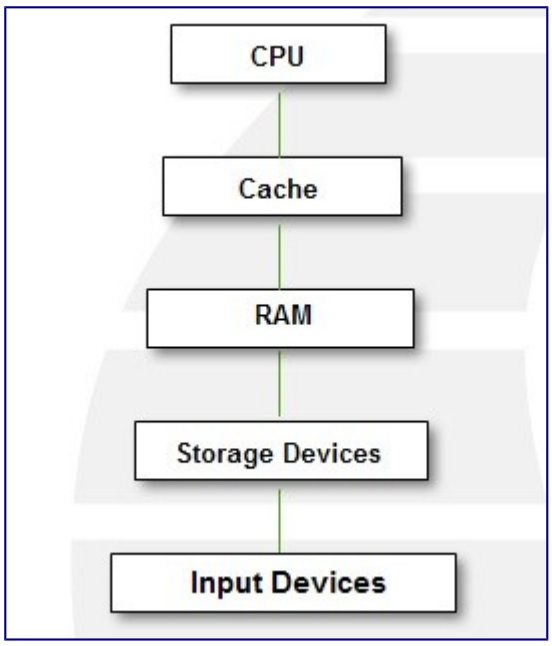
The cache memory stores the program (or its part) currently being executed or which may be executed within a short period of time. The cache memory also stores temporary data that the CPU may frequently require for manipulation.

The cache memory works according to various algorithms, which decide what information it has to store. These algorithms work out the probability to decide which data would be most frequently needed. This probability is worked out on the basis of past observations.

It acts as a high speed buffer between CPU and main memory and is used to temporary store very active data and action during processing since the cache memory is faster then main memory, the processing speed is increased by making the data and instructions needed in current processing available in cache. The cache memory is very expensive and hence is limited in capacity.

Type of Cache memory

Cache memory improves the speed of the CPU, but it is expensive.Type of Cache Memory is divided into different level that are L1,L2,L3:



Level 1 (L1) cache or Primary Cache

L1 is the primary type cache memory. The Size of the L1 cache very small comparison to others that is between 2KB to 64KB, it depent on computer processor. It is a embedded register in the computer microprocessor(CPU).The Instructions that are required by the CPU that are firstly searched in L1 Cache. Example of registers are accumulator, address register,, Program counter etc.

Level 2 (L2) cache or Secondary Cache

L2 is seconday type cache memory. The Size of the L2 cache is more capacious than L1 that is between 256KB to 512KB.L2 cache is Located on computer microprocessor.After searching the Instructions in L1 Cache,if not found then it searched into L2 cache by computer microprocessor. The high-speed system bus interconnecting the cache to the microprocessor.

Level 3 (L3) cache or Main Memory

The L3 cache is larger in size but also slower in speed than L1 and L2,it's size is between 1MB to 8MB.In Multicore processors, each core may have seperate L1 and L2,but all core share a common L3 cache. L3 cache double speed than the RAM.

https://study.com/academy/lesson/cache-memory-definition-lesson-quiz.html

An overview of Linux Boot Process for Embedded Systems

This Text provides an insight in to the Embedded Linux Boot Process. Reader should have a basic Knowledge of Boot Process in general and should be familiar with Embedded Linux Boot Process.

.....
PART-A
.....

(1) Software components Involved in Embedded Linux Boot Process
  (a) Bootloader
  (b) kernel Image
  (c) root file system - either an initrd image or a NFS location

(2) Steps during Booting process of a conventional PC
  (a) System Startup - PC-BIOS/BootMonitor
  (b) Stage1 bootloader - MBR
  (c) stage2 bootloader - LILO,GRUB etc
  (d) kernel - Linux
  (e) init - The User Space

(3) Booting process for an Emebedded Systems
  (a) Instead of BIOS you will run program from a fixed location in Flash
  (b) The components involved in the first steps of PC boot process are combined in to a single "boot strap firmware", called "boot loader".
  (c) Bootloader also provides additional features useful for development & debugging.

(4) What is System Startup?
  [Exact process depends on the Target Hardware]
  (a) CPU starts exectuing BIOS at address 0xFFFF0
  (b) POST (Power On Self Test) - is the first step of BIOS.
  (c) run time services - involve local device enumeration and initialization
  (d) After the POST is complete, POST related code is flushed out of memory. But BIOS runtime services remain in memory and are available to the target OS.
  (e) The runtime searches for devices that are both active and bootable in order of preference defined in CMOS settings
  (f) The primary boot loader is loaded and BIOS returns control to it

(5) The Primary boot loader

(a) Performs few optional initializations

(b) Its main job is to Load the secondary boot loader

(6) Secondary boot loader

(a) The Second Stage boot loader loads the Linux & an optional initial RAM disk in to the memory

(b) on PC, the initrd is used as a temporary root files system, before final root file system gets mounted. However, on embedded systems, the initrd is generally the final root file system.

(c) The secondary loader passes control to the kernel image - kernel is decompressed & initialized

(d) So, the secondary boot loaderis the kernel Loader, can also load optional initial RAM disk (initrd), and then invokes the kernel image

(7) Kernel Invocation

(a) As the kernel is invoked, it performs the checks on system hardware, enumerates the attached hardware devices, mounts the root device

(b) Next it loads the necessary kernel modules

(c) First user-space program (init) now starts and high-level system initialization is performed

(d) The Kernel Invocation Process is similar on Embedded Linux Systems as well as on PC. We will discuss this in detail in following text.

(8) Kernel Image

(a) Is typically a compressed image [zlib compression]

(b) Typical named a zImage (<512 KB) or bzImage (> 512 KB)

(c) At the head of this image (in file head.S) is a routine that does some minimal amount of hardware set up and then decompresses the kernel contained in the kernel image and places in to high memory (high memory & low memory)

.....

PART-B

.....

(1) Kernel Invocation Process - A Summary

(a) zImage Entry Point

(b) PERFORM BASIC HARDWARE SET UP

(c) PERFORM BASIC ENVIRONMENT SET UP (stack etc)

(d) CLEAR BSS

[Now We have set up the run time environment for the code to be executed next]

(e) DECOMPRESS THE KERNEL IMAGE
(f) Execute the decompressed Kernel Image
- INITIALIZE PAGE TABLES
- ENABLE MMU
- DETECT CPU (& optoinal FPU) TYPE & SAVE THIS INFO

[With above set up, we are now ready to execute a general C Code. Till now we only executed asm routines.]

(g) The First Kernel C function
- DO FURTHER INITIALIZATIONS
- LOAD INITRD

[The above code is being executed by swapper process, the one with pid 0]

(h) The Init Process
- FORK INIT PROCESS
- Init process is with pid 1
- Invoke Scheduler
- RELINQUISH CONTROL TO SCHEDULER

(2) zImage Entry point
(a) This is a call to the absolute physical address by boot loader

- Refer to file arch/***/boot/compressed/head.S: start() in kernel source.
- For the ARM process this is "arch/arm/boot/compressed/head.S: start()"

(b) start() performs
- basic hardware set up
- basic environment set up
- clears bss
- calls the decompress_kernel()

(3) Decompressing Kernel Image
(a) This is a call to arch/***/boot/compressed/misc.c: decompress_kernel()
- This function decompresses the kernel image, stores it in to the RAM & returns the address of decompressed image in RAM.
(b) on ARM processor this maps to "arch/arm/boot/compressed/misc.c: decompress_kernel()" routine.

(4) Execute the decompressed Kernel Image
(a) After we have got the (uncompressed) kernel image in RAM, we execute it.

(b) Execution starts with call_kernel() function call [from start()].

(c) call_kernel() will start executing the kernel code, from Kernel entry point.

(d) arch/***/kernel/head.S contains the kernel entry point.

- separate entry points for Master CPU and Secondary CPUs (for SMP systems).
- This code is in asm
- Page Tables are Initialized & MMU is enabled.
- type of CPU alongwith optional FPU is detected and stored
- For Master CPU; start_kernel(), which is the first C function to be executed in kernel, is called.
- For secondary CPUs (on an SMP system); secondary_start_kernel() is the first C function to be called.

(e) On ARM process it maps to "arch/arm/kernel/head.S

- Contains kernel ENTRY points for master and secondary CPU.
- For Master CPU "mmap_switched()" is called as soon as mmu gets enabled. The mmap_switched() saves the CPU info makes a call to start_kernel()
- For Secondary CPU "secondary_start_kernel()" is called as soon as MMU gets enabled.

(5) The first kernel C function

(a) The start_kernel() function is being executed by the swapper process.

(b) Refer to init/main.c: start_kernel() in the kernel source.

(c) start_kernel():

- a long list of initialization functions are called: this sets up interrupts, performs further memory configuration & loads the initrd.
- calls rest_init() in the End.

(d) econdary_start_kernel() for secondary CPUs (on SMP systems).

- arch/***/kernel/smp.c: secondary_start_kernel()
- for ARM, arch/arm/kernel/smp.c
- there is not rest_init() call for secondary CPUs.

(6) Init process

(a) Refer to init/main.c: rest_init() in kernel source.

(b) Executed only on the Master CPU

(c) rest_init() forks new process by calling kernel_thread() function

(d) kernel_thread(kern_init,*,*); kern_init has PID-1

(e) kern_init() will call the initialization scripts.

(f) kernel_thread() defined in "arch/***/kernel/process.c: kernel_thread()".

(g) on ARM, arch/arm/kernel/process.c

(7) Invoke scheduler

(a) The rest_init() calls cpu_idle() in end [after it is done creating the init process]

(b) For the Secondary CPUs (on SMP systems), cpu_idle is directly

called from secondary_start_kernel [no step-5 & hence no init process].
(c) cpu_idle() defined in "arch/***/kernel/process.c: cpu_idle()".
(d) on ARM, arch/arm/kernel/process.c

(8) initrd image
(a) The initrd serves as a temporary root file system in RAM & allows the kernel to fully boot without having to mount and physical disks. Since the necessary modules needed to interface with peripherals
can be part of initrd the kernel can be very small.
(b) pivot_root() routine: the root files system is pivoted where the initrd root file system is unmounted & the real root file system is mounted.
(c) In any embedded system, the initrd could be the root file system.

ARM

Boot Sequence for an ARM based embedded system

This post is going to explore the boot sequence for a Boot ROM based embedded system. It is based on my experiences with an ARM processor based embedded system, but the concept of an integrated Boot ROM is used by other modern CPUs and microcontrollers as well.

In an ARM embedded system, at the time of power on, CPU is uninitialized and a basic clock setup, system specifics' setup is required before proceeding to the bigger and complex tasks. A piece of code is required at power on which does the basic system setup before handing over the control to the bootloader present in flash(already programmed) or to support the download tool for programming the flash and then handing over the control to the bootloader present in flash.

For this purpose, a hardware bootloader generally called as Boot Rom is provided by vendor (pre-loaded into the processors' internal ROM).This is hardwired at the manufacturing time. After a power on reset, that causes the processor core to jump to the reset vector, Boot Rom is the first code to execute in the processor.

Responsibilities of Bootrom

- Bootrom performs the essential initialization including programming the clocks, stacks, interrupt set up etc.
- Bootrom will detect the boot media using a system register. This is to determine where to find the software bootloader. A particular sequence of probing for boot media is followed as defined by the manufacturer. This includes the order of looking for bootloader in external NOR/NAND flash, or probing for some specific characters on UART /USB for establishing connection with downloader to download the binary in flash. If no bootloader is found in any external memory, bootrom listens for a download request on UART/USB port to start the download process.
- Thus during the probing process, if the flash has already been programmed, software bootloader will be detected to be available in flash , if not so –it will be downloaded to the

flash by bootrom.(This probing sequence will generally assign higher priority to external memory , so if bootloader is detected to be present in memory, it won't proceed to downloading the image again)

- For platforms using NAND flash , the bootrom will load this boot loader to internal RAM and set the program counter at the load address of the SW bootloader in RAM.
- For platforms using NOR flash, control is transferred to the external flash (NOR flash is XiP- Execute in Place).

There is another point to be noted here regarding bad block management support in Boot ROM. If sw bootloader is residing only in block 0 of flash, no Bad block Management is required in Boot ROM as block 0 is guaranteed to be good by the manufacturer but if sw bootloader resides in block 1 onwards or is large enough to span multiple blocks, then ,Boot ROM needs to include bad block handling. Bad block management will include relocating the contents of the block in flash that goes bad in the process of accessing the block to a new block and updating the bad block/relocation table of the platform.

Software Bootloader

The main task of S/W bootloader is to load the OS and pass over the execution to it after setting up necessary environment for its setup. For this, the bootloader must first initialize the DDR memory (this includes setting up the controller, refresh rate etc).It must also perform bad block management while accessing the flash memory.

After the system setup, bootloader’s responsibility would be to look for an OS to boot. Again, like Boot Rom, if OS is not already loaded to flash, it will load this from the boot media in flash and execute-in-place in case of NOR flash, or place it in RAM in case of NAND flash. It will also place some boot parameters in memory for the OS to read when it starts up if required.

After all the necessary system setup, bootloader will pass over the execution to OS and go out of scope.

In my upcoming posts, I plan to move on to the details that constitute a bootloader.An ARM bootloader specifically.Till then , happy booting!