An overview of Linux Boot Process for Embedded Systems

Kunal Singh ● December 25, 2008

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.

....

References:
http://www.ibm.com/developerworks/linux/library/l-linuxboot/
http://duartes.org/gustavo/blog/post/how-computers-boot-up
http://duartes.org/gustavo/blog/post/kernel-boot-process

Previous post by Kunal Singh:
Building Linux Kernel for Desktops