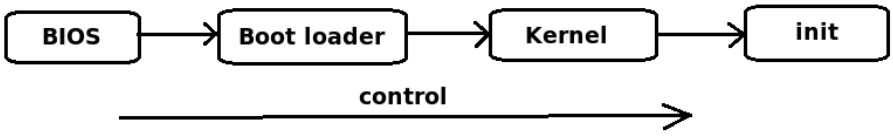
Linux 系统启动过程分析

by 王斌斌
binbinwang118@gmail.com

Linux 系统启动过程分析
操作系统的启动过程，实际上是控制权移交的过程。Linux 系统启动包含四个主要的阶段：BIOS initialization, boot loader, kernel initialization, and init startup.见下图：



阶段一、**BIOS initialization**，主要功能如下：

1. Peripherals detected
2. Boot device selected
3. First sector of boot device read and executed

系统上电开机后，主板 BIOS（Basic Input / Output System）运行 POST（Power on self test）代码，检测系统外围关键设备（如：CPU、内存、显卡、I/O、键盘鼠标等）。硬件配置信息及一些用户配置参数存储在主板的 CMOS(Complementary Metal Oxide Semiconductor)上（一般 64 字节），实际上就是主板上一块可读写的 RAM 芯片，由主板上的电池供电，系统掉电后，信息不会丢失。
执行 POST 代码对系统外围关键设备检测通过后，系统启动自举程序， 根据我们在 BIOS 中设置的启动顺序搜索启动驱动器（比如的硬盘、光驱、网络服务器等）。选择合适的启动器，比如通常情况下的硬盘设备，BIOS 会读取硬盘设备的第一个扇区（MBR，512 字节），并执行其中的代码。实际上这里 BIOS 并不关心启动设备第一个扇区中是什么内容，它只是负责读取该扇区内容、并执行，BIOS 的任务就完成了。此后将系统启动的控制权移交到 MBR 部分的代码。
注： 在我们的现行系统中，大多关键设备都是连在主板上的。因此主板 BIOS 提供了一个操作系统（软件）和系统外围关键设备（硬件）最底级别的接口，在这个阶段，检测系统外围关键设备是否“准备好”，以供操作系统使用。

阶段二、**Boot Loader**
关于 Boot Loader，简单的说就是启动操作系统的程序，如 grub，lilo，也可以将 boot loader 本身看成一个小系统。
The BIOS invokes the boot loader in one of two ways:

1. It pass control to an initial program loader (IPL) installed within a driver's Master Boot Record (MBR)
2. It passes control to another boot loader, which passes control to an IPL installed within a partition's boot sector.

In either case, the IPL must exist within a very small space, no larger than 446 bytes. Therefore, the IPL for GRUB is merely a first stage, whose sole task is to locate and load a second stage boot loader, which does most of the work to boot the system.

There are two possible ways to configure boot loaders:
Primary boot loader: Install the first stage of your Linux boot loader into the MBR. The boot loader must be configure to pass control to any other desired operating systems.
Secondary boot loader: Install the first stage of your Linux boot loader into the boot sector of some partition. Another boot loader must be installed into the MBR, and configured to pass control to your Linux boot loader.

假设 Boot Loader 为 **grub (grub-0.97)**，其引导系统的过程如下：
grub 分为 **stage1 (stage1_5) stage2** 两个阶段。stage1 可以看成是 initial program loaderl（IPL），而 stage2 则实现了 grub 的主要功能，包括对特定文件系统的支持（如 ext2,ext3，reiserfs 等），grub 自己的 shell，以及内部程序（如： kernrl， initrd， root ）等。

stage 1：MBR（512 字节，0 头 0 道 1 扇区），前 **446** 字节存放的是 stage1，后面存放硬盘分区表信息，BIOS 将 stag1 载入内存中 0x7c00 处并跳转执行。 stage1（/stage1/stage.S）的任务非常但存，仅仅是将硬盘 0 头 0 道 2 扇区读入内存。0 头 0 道 2 扇区内容是源代码中的 /stage2/start.S，编译后 512 字节，它是 stage2 或者 stage1_5 的入口。
注：此时 stage1 是没有能力识别文件系统的，其定位硬盘 0 头 0 道 2 扇区过程如下：
BIOS 将 stage1 载入内存 0x7c00 处并执行，然后调用 BIOS INIT13 中断，将硬盘 0 头 0 道 2 扇区内容载入内存 0x7000 处，然后调用 copy_buffer 将其转移到内存 0x8000 处。定位 0 头 0 道 2 扇区有两种寻址方式：LBA、CHS，代码如下：

```
// LBA: 调用 BIOS INIT 13 中断 0x42 号功能
/*
 * BIOS call "INT 0x13 Function 0x42" to read sectors from disk
into memory
 * Call with %ah = 0x42
 * %dl = drive number
 * %ds:%si = segment:offset of disk address packet
 * Return:
 * %al = 0x0 on success; err code on failure
 */

        movb $0x42, %ah
        int $0x13

        /* LBA read is not supported, so fallback to CHS. */
        jc chs_mode

        movw $STAGE1_BUFFERSEG, %bx // STAGE1_BUFFERSEG 在 stag1.h 中
定义，为 0x7000
        jmp copy_buffer


//CHS：调用 BIOS INIT 13 中断 0x2 号功能
/*
 * BIOS call "INT 0x13 Function 0x2" to read sectors from disk into
memory
 * Call with %ah = 0x2
 * %al = number of sectors
```

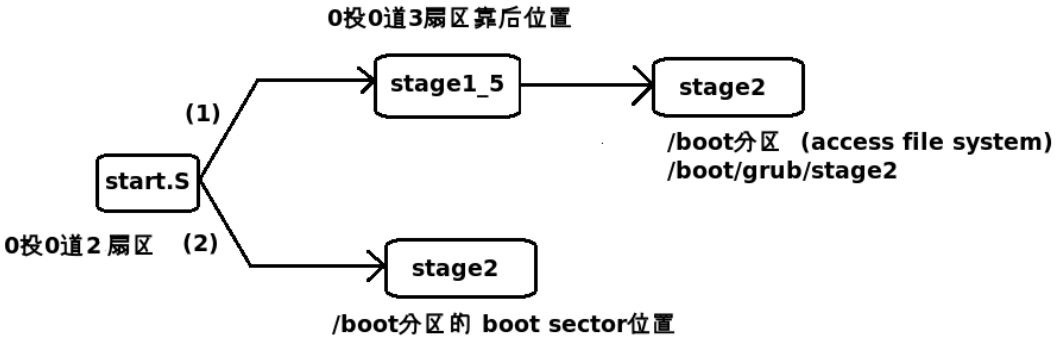
```
 * %ch = cylinder
 * %cl = sector (bits 6-7 are high bits of "cylinder")
 * %dh = head
 * %dl = drive (0x80 for hard disk, 0x0 for floppy disk)
 * %es:%bx = segment:offset of buffer
 * Return:
 * %al = 0x0 on success; err code on failure
 */

        movw $STAGE1_BUFFERSEG, %bx // STAGE1_BUFFERSEG在stag1.h中定义，为0x7000
        movw %bx, %es /* load %es segment with disk buffer */

        xorw %bx, %bx /* %bx = 0, put it at 0 in the segment */
        movw $0x0201, %ax /* function 2 */
        int $0x13

        jc read_error

        movw %es, %bx
// COPY_BUFFER
copy_buffer:
        movw ABS(stage2_segment), %es

        /*
         * We need to save %cx and %si because the startup code in
         * stage2 uses them without initializing them.
         */
        pusha
        pushw %ds

        movw $0x100, %cx
```

```
        movw %bx, %ds
        xorw %si, %si
        xorw %di, %di

        cld

        rep
        movsw

        popw %ds
        popa

        /* boot stage2 */
        jmp *(stage2_address) //stage2_address 为 0x8000
```

start.S： start.S 的主要功能是将 stage2 或 stage1_5 从硬盘载入内存，如果是 stage2，则载入 0x8200 处；如果是 stage1_5，则载入 0x2200 处。参见如下代码：

```
bootit:
        /* print a newline */
        MSG(notification_done)
        popw %dx /* this makes sure %dl is our "boot" drive */
#ifdef STAGE1_5
        ljmp $0, $0x2200 // 跳转到 0x2200 执行
#else /* ! STAGE1_5 */
        ljmp $0, $0x8200 // 跳转到 0x8200 处执行
#endif /* ! STAGE1_5 */
```

注：这里的 stage2 或者 stage1_5 不是 /boot 分区 /boot/grub 目录下的文件，这个时候 grub 还没有能力识别任何文件系统。分以下两种情况：
（1）假如 start.S 读取的是 stage1_5，它存放在硬盘 0 头 0 道 3 扇区向后的位置，stage1_5 作为 stage1 和 stage2 中间的桥梁，stage1_5 有识别文件系统的能力，此后，grub 才有能力去访问 /boot 分区 /boot/grub 目录下的 stage2 文件，将 stage2 载入内存并执行。
（2）假如 start.S 读取的是 stage2，同样，这个 stage2 也不是 /boot 分区 /boot/grub 目录下的 stage2，这个时候 start.S 读取的是存放在 /boot 分区 Boot Sector 的 stage2。这种情况下就有一个限制：因为 **start.S** 通过 **BIOS** 中断方式直接对硬盘寻址（而非通过访问具体的文件系统），其寻址范围有限，限制在 **8GB** 以内。因此这种情况需要将 /boot 分区分在硬盘 8GB 寻址空间之前。
如下图：



trouble shootting：很明显，假如是情形（2），我们将/boot/grub 目录下的内容清空，依然能成功启动 grub；假如是情形（1），将/boot/grub 目录下 stage2 删除后，则系统启动过程中 grub 会启动失败。

stage2，如上所说，start.S 作为 stage2 或者 stage1_5 的入口，最终都会把 stage2 载入内存并执行。stage2 作为 grub 的主要功能实现，其存放于具体的文件系统下，如 /boot/grub/stage2，也可存放于/boot 分区的 boot sector。下面分析 stage2 过程。
stage2 的入口是 **asm.S(/stage2/asm.S)**，asm.S 文件对一些变量做初始化，如 config_file，参见如下代码：

```
VARIABLE(config_file)
#ifndef STAGE1_5
        .string "/boot/grub/menu.lst"
#else /* STAGE1_5 */
        .long 0xffffffff
        .string "/boot/grub/stage2"
#endif /* STAGE1_5 */
```

假设定义了 STAGE1_5，则 config_file 是 /boot/grub/stage2；如果定义的是 stage2，则 config_file 是 /boot/grub/menu.lst。
在 asm.S 之前都是汇编语言，这部分汇编代码很重要的工作是完成了 **C** 语言环境的初始还，从汇编进化到 C。asm.S 中会调用 **init_bios_info (void)**，这是整个 C 语言代码的入口，在/stage2/common.c 中定义。C 语言运行环境初始化好后，转入 **stage2.c** 文件的 void cmain (void)函数。

```
/* This is the starting function in C. */
void
cmain (void)
{
… …
}
```

之后调用 grub_open()函数打开 config_file

```
is_opened = grub_open (config_file);
```

不论是图形菜单选选择相应的启动项，还是在 grub> shell 下执行相应的指令。 都由 find_command() 函数在 struct builtin *builtin_table[] 变量中找相应的指令， 然后执行对应指令的 XXX_func() 函数，都是 **static** 类型的函数。其主要代码如下：

```
// stage2.c
/* This is the starting function in C. */
void
cmain (void)
{
... ...
    /* Get the pointer to the builtin structure. */
    builtin = find_command (cmdline);
... ...
}

// builtins.c
/* Find the builtin whose command name is COMMAND and return the
pointer. If not found, return 0. */
struct builtin *
find_command (char *command)
{
    struct builtin **builtin;
    ... ...
     for (builtin = builtin_table; *builtin != 0; builtin++)
     {
          ... ...
       }
}
// builtins.c
/* The table of builtin commands. Sorted in dictionary order. */
struct builtin *builtin_table[] =
{
    ... ...
}

static int
XXX_func()
```

```
{
... ...
}
```

关于 grub 常用的几个指令对应的函数：

```
grub>root (hd0,0)
grub>kernel /xen.gz-2.6.18-37.el5
grub>module /vmlinuz-2.6.18-37.el5xen ro root=/dev/sda2
grub>module /initrd-2.6.18-37.el5xen.img
grub>boot
```

如上显示，是 RHLE5 系统启动 grub 用的几个指令：
(1) **root** 指令为 grub 指定了一个根分区，其对应的函数是：

```
static int
root_func (char *arg, int flags)
{
  return real_root_func (arg, 1);
}

static int
real_root_func (char *arg, int attempt_mount)
{
... ...
}
```

(2) **kernel** 指令将操作系统内核载入内存，其对应的函数是：

```
/* kernel */
static int
kernel_func (char *arg, int flags)
{
... ...
}
```

(3) **module** 指令加载指定的模块，其对应的函数是：

```
/* module */
static int
```

```
module_func (char *arg, int flags)
{
... ...
}
```

(4) **boot** 指令调用相应的启动函数启动 OS 内核，其对应的函数是：

```
/* boot */
static int
boot_func (char *arg, int flags)
{
... ...
}
```

阶段三、**Kernel Initialization**

如阶段 2 所述，grub>boot 指令后，系统启动的控制权移交给 kernel。 Kernel 会立即初始化系统中各设备并做相关配置工作，其中包括 CPU、I/O、存储设备等 （关于 Kernel 这部分工作，具体怎么处理不清楚）。

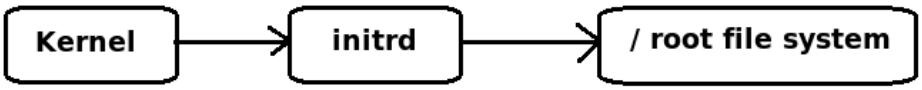
关于设备驱动加载，有两部分：一部分设备驱动编入 **Linux Kernel** 中，Kernel 会调用这部分驱动初始化相关设备，同时将日志输出到 kernel message buffer，系统启动后，dmesg 可以查看到这部分输出信息。另外有一部分设备驱动并没有编入 Kernel，而是作为模块形式放在 **initrd(ramdisk)** 中。下面详述一下 initrd。

在 2.6 内核中，支持两种格式的 initrd，一种是 2.4 内核的文件系统镜像 image-initrd，一种是 cpio 格式。以 cpio 格式为例，内核判断 initrd 为 cpio 的文件格式后，会将 initrd 中的内容释放到 rootfs 中。 initrd 一种基于内存的文件系统，启动过程中，系统在访问真正的根文件系统 / 时，会先访问 initrd 文件系统。将 initrd 中的内容打开来看，会发现有 bin、devetc、lib、procsys、sysroot、 init 等文件（包含目录）。其中包含了一些设备的驱模拟块，比如 scsi ata 等设备驱动模块，同时还有几个基本的可执行程序 insmod, modprobe, lvm，nash。主要目的是加载一些存储介质的驱动模块，如上面所说的 scsi ideusb 等设备驱动模块，初始化 LVM，把 / 根文件系统以只读方式挂载。

initrd 中的内容释放到 rootfs 中后，Kernel 会执行其中的 init 文件，这里的 init 是一个脚本，由 nash 解释器执行。这个时候 内核的控制权移交给 init 文件处理，我们查看 init 文件的内容，主要也是加载各种存储介质相关的设备驱动。

驱动加载后，会创建一个根设备，然后将根文件系统 / 以只读的方式挂载。这步结束后，执行 switchroot，转换到真正的根 /上面去，同时运行/sbin/init 程序，开启系统的 1 号进程，此后，系统启动的控制权移交给 init 进程。关于 switchroot，这是在 nash 中定义的程序。

initrd 处理流程图



思考：为什么会存在一个 initrd 文件系统？Linux Kernel 需要适应多种不同的硬件架构，但是将所有的硬件驱动编入 Kernel 又是不实际的，而且 Kernel 也不可能每新出一种硬件结构，就将该硬件 的设备驱动写入内核。实际上 Linux Kernel 仅是包含了基本的硬件驱动，在系统安装过程中会检测系统硬件信息，根据安装信息和系统硬件信息将一部分设备驱动写入 initrd 。这样在以后启动系统时，一部分设备驱动就放在 initrd 中来加载。

阶段四、**init Initialization**

init 进程起来后，系统启动的控制权移交给 init 进程。
/sbin/init 进程是所有进程的父进程，当 init 起来之后，它首先会读取配置文件/etc/inittab，进行以下工作：
1）执行系统初始化脚本(/etc/rc.d/rc.sysinit),对系统进行基本的配置，以读写方式挂载根文件系统及其它文件系统，具体作用见后面说明。到这一步系统基本算运行起来了，后面需要进行运行级别的确定及相应服务的启动。
2）确定启动后进入的运行级别
3) 执行/etc/rc.d/rc, 该文件定义了服务启动的顺序是先 K 后 S,而具体的每个运行级别的服务状态是放在/etc/rc.d/rcn.d(n=0~6)目录下，所有的文件均链接至/etc/init.d 下的相应文件。
4）有关 key sequence 的设置
5) 有关 UPS 的脚本定义
6）启动虚拟终端/sbin/mingetty
7）在运行级别 5 上运行 X
这时呈现给用户的就是最终的登录界面。
至此，系统启动过程完毕：）

说明：
1） /etc/rc.d/rc.sysint -- System Initialization Tasks
它的主要工作有：
配置 selinux，
系统时钟，
内核参数（/etc/sysctl.conf），
hostname，
使能 swap 分区，
根文件系统的检查和二次挂载（读写），
激活 RAID 和 LVM 设备
使能磁盘 quota
检查并挂载其它文件系统
等等。