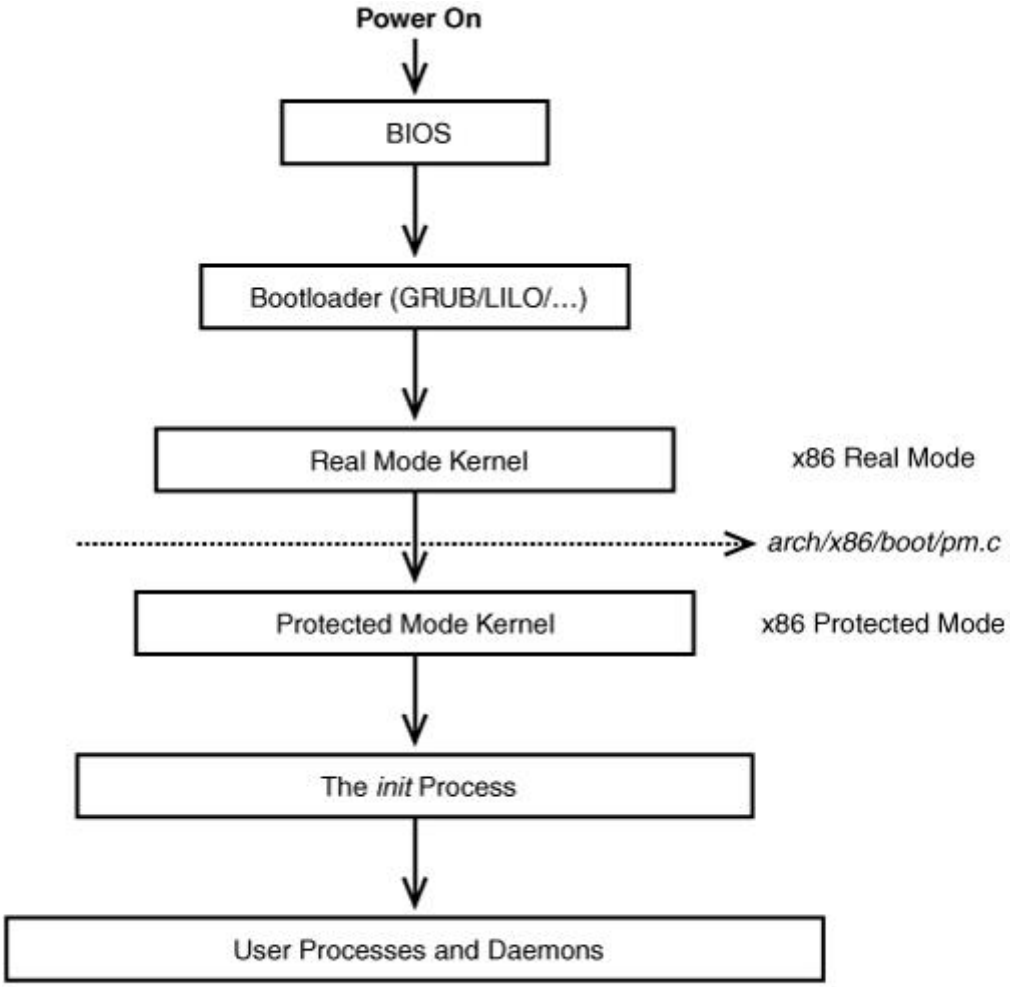
# 第 2 章 内核一瞥

在我们开始步入 Linux 设备驱动的神秘世界之前，让我们先熟悉一些从驱动开发人员应该理解的基本的内核概念。我们将学习到内核定时器、同步机制以及内存分配方法，但是，先让我们从顶层视角开始探索，扫描一下内核发出的启动信息，并在感兴趣的地方设置停下来看一看。

## 启动过程

图 2.1 显示了基于 x86 计算机 Linux 系统的启动顺序。第一步是 BIOS 从启动设备中导入主引导记录（ MBR ），接下来 MBR 中的代码查看分区表并从活动分区读取 GRUB 、 LILO 或 SYSLINUX 等 bootloader ，之后 bootloader 会加载压缩后的内核映像并将控制权传递给它。内核取得控制权后，会将自身解压缩并投入运转。

**图 2.1 基于 x86 的硬件上 Linux 的启动过程**




基于 x86 的处理器有两种操作模式：实模式和保护模式。在实模式下，用户仅可以使用 1MB 内存，并且没有任何保护。保护模式则更加复杂，用户可以使用更多的高级功能（如分页）。 CPU 提供了一条由实模式通向保护模式的道路，但是，这条路只允许单向行驶，用户不能从保护模式再切换回实模式。

内核初始化的第一步是执行实模式下的汇编代码，之后执行保护模式下 init/main.c 文件（上一章我们修改了这个文件）中的 start_kernel() 函数。 start_kernel() 函数首先会初始化 CPU 子系统，之后让内存管理和进程管理系统就位，接下来启动外部总线和 I/O 设备，最后的一步是激活所有 Linux 进程的父亲 init 。 init 执行用户空间的脚本以启动必要的内核服务，它最终派生控制台终端程序并显示登录（ login ）提示。

接下来，每一小节的标题都是图 2.2 中的一条打印信息，这些信息来源于基于 x86 的笔记本电脑的 Linux 启动过程。如果你在启动体系结构上启动 Linux ，消息以及语义可能会有所改变。如果本节中的一些内容读起来非常晦涩，请不要担心。目前的目的仅是从 100 英尺 的高度给你一个视图，让你初次品尝内核甜点的味道。接下来要提到的许多概念都会在以后的章节中进行更深的论述。

**图 2.2 内核启动信息**

```
Linux version 2.6.23.1y (root@localhost.localdomain) (gcc version 4.1.1
20061011 (Red


Hat 4.1.1-30)) #7 SMP PREEMPT Thu Nov 1 11:39:30 IST 2007


BIOS-provided physical RAM map:


 BIOS-e820: 0000000000000000 - 000000000009f000 (usable)

 BIOS-e820: 000000000009f000 - 00000000000a0000 (reserved)

 ...

758MB LOWMEM available.


...


Kernel command line: ro root=/dev/hda1
```

```
...

Console: colour VGA+ 80x25

...

Calibrating delay using timer specific routine.. 1197.46 BogoMIPS (lpj=2394935)

...

CPU: L1 I cache: 32K, L1 D cache: 32K

CPU: L2 cache: 1024K

...

Checking 'hlt' instruction... OK.

...

Setting up standard PCI resources
```

```
...

NET: Registered protocol family 2

IP route cache hash table entries: 32768 (order: 5, 131072 bytes)

TCP established hash table entries: 131072 (order: 9, 2097152 bytes)

...

checking if image is initramfs... it is

Freeing initrd memory: 387k freed

...

io scheduler noop registered
io scheduler anticipatory registered (default)

...
```

```
00:0a: ttyS0 at I/O 0x3f8 (irq = 4) is a NS16550A

...

Uniform Multi-Platform E-IDE driver Revision: 7.00alpha2

ide: Assuming 33MHz system bus speed for PIO modes; override with idebus=xx

ICH4: IDE controller at PCI slot 0000:00:1f.1

Probing IDE interface ide0...

hda: HTS541010G9AT00, ATA DISK drive

hdc: HL-DT-STCD-RW/DVD DRIVE GCC-4241N, ATAPI CD/DVD-ROM drive

...

serio: i8042 KBD port at 0x60,0x64 irq 1

mice: PS/2 mouse device common for all mice
```

```
...

Synaptics Touchpad, model: 1, fw: 5.9, id: 0x2c6ab1, caps: 0x884793/0x0

...

agpgart: Detected an Intel 855GM Chipset.

...

Intel(R) PRO/1000 Network Driver - version 7.3.20-k2

...

ehci_hcd 0000:00:1d.7: EHCI Host Controller

...

Yenta: CardBus bridge found at 0000:02:00.0 [1014:0560]
```

```
...

Non-volatile memory driver v1.2

...

kjournald starting. Commit interval 5 seconds

EXT3 FS on hda2, internal journal

EXT3-fs: mounted filesystem with ordered data mode.
...
INIT: version 2.85 booting
...
```

BIOS-provided physical RAM map

内核解析从 BIOS 中读取到的系统内存映射，并率先将这些信息打印出来：

```
BIOS-provided physical RAM map:
BIOS-e820: 0000000000000000 - 000000000009f000 (usable)
...
BIOS-e820: 00000000ff800000 - 0000000100000000 (reserved)
```

实模式下的初始化代码通过使用 BIOS 的 int 0x15 服务并执行 0xe820 号函数来获得系统的内存映射信息。内存映射信息中包含了预留的和可用的内存，内核将使用这些信息创建其可用的内存池。在附录 B 《 Linux 和 BIOS 》的《实模式调用》一节，我们会对 BIOS 提供的内存映射问题进行更深入的讲解。

758MB LOWMEM Available

896MB 以内的常规的可被寻址的内存区域被称作低端内存。内存分配函数 kmalloc() 就是从该区域分配内存的。高于 896MB 被称为高端内存，只有在采用特殊的方式进行映射后才能被访问。在启动过程中，内核会计算并显示这些内存 zone 内总的页数，在本章的稍后，会对这些内存 zone 进行更深入的分析。

Kernel Command Line: ro root=/dev/hda1

Linux 的 bootloader 通常会给内核传递一个命令行。命令行中的参数类似于传递给 C 程序中 main() 函数的 argv[] 列表，唯一的不同是它们是传递给内核的。你可以在 bootloader 的配置文件中增加命令行参数，当然，也可以在运行过程中对 bootloader 的提示行进行修改 [1] 。如果你正在使用 GRUB 这个 bootloader ，归因于发行版的不同，其配置文件可能是 /boot/grub/grub.conf 或者是 /boot/grub/menu.lst 。如果你正在使用 LILO ，配置文件为 /etc/lilo.conf 。下面给出了一个 grub.conf 文件的例子（增加了一些注释），阅读了紧接着“ title kernel 2.6.23 ”后的一行之后，你会发现前述打印信息的由来。 \

[1] 嵌入式设备上的 bootloader 通常经过了“瘦身”，并不支持配置文件或类似机制。归因于此，许多非 x86 体系结构提供了 CONFIG_CMDLINE 这个内核配置选项，通过它，用户可以在编译内核时提供内核命令行。

```
default 0  #Boot the 2.6.23 kernel by default
timeout 5  #5 second to alter boot order or parameters


title kernel 2.6.23     #Boot Option 1
 #The boot image resides in the first partition of the first disk
 #under the /boot/ directory and is named vmlinuz-2.6.23. 'ro'
 #indicates that the root partition should be mounted read-only.
 kernel (hd0,0)/boot/vmlinuz-2.6.23 ro root=/dev/hda1
```

```
  #Look under section "Freeing initrd memory:387k freed"
  initrd (hd0,0)/boot/initrd

#...
```

命令行参数将影响启动过程中的代码执行路径。举一个例子，假设某命令行参数为 bootmode ，如果该参数被设置为 1 ，意味着你希望在启动过程中打印一些调试信息并在启动结束时切换到 runlevel 的第 3 级（到我们分析 init 进程的打印信息时，会学习到 runlevel 的含义）；如果 bootmode 参数被设置为 0 ，意味着你希望启动过程相对简洁，并且设置 runlevel 为 2 。因为你已经熟悉了 init/main.c 文件，让我们在该文件中增加如下修改：

```
static unsigned int bootmode = 1;
static int __init
is_bootmode_setup(char *str)
{
  get_option(&str, &bootmode);
  return 1;
}

/* Handle parameter "bootmode=" */
__setup("bootmode=", is_bootmode_setup);

if (bootmode) {
  /* Print verbose output */
  /* ... */
}
```

```
/* ... */


/* If bootmode is 1, choose an init runlevel of 3, else
   switch to a run level of 2 */
if (bootmode) {
  argv_init[++args] = "3";
} else {
  argv_init[++args] = "2";
}


/* ... */
```

请重新编译内核并尝试新的修改。另外，本书第 18 章《嵌入式 Linux 》的《内存分布》一节也将对命令行参数进行更多的讲解。

Calibrating Delay...1197.46 BogoMIPS (lpj=2394935)

在启动过程中，内核会计算处理器在一个 jiffy 时间内运行一个内部的 delay 循环的次数。 jiffy 的含义是系统定时器 2 个连续的节拍之间的间隔。如果你所期待的那样，该计算必须被校准到你的 CPU 的处理速度。校准的结果被存储在称为 loops_per_jiffy 的内核变量中。使用 loops_per_jiffy 的一个场合是某设备驱动希望进行小的微妙级别的延迟的时候。

为了理解 delay 循环校准代码，让我们看一下定义于 init/calibrate.c 文件中的 calibrate_delay() 函数。该函数机智地使用整型运算得到了浮点的精度。如下的代码片段（增加了一些注释）显示了该函数的开始部分，这部分用于得到一个粗略的 loops_per_jiffy ：

```
loops_per_jiffy = (1 << 12); /* Initial approximation = 4096 */
printk(KERN_DEBUG "Calibrating delay loop... ");
while ((loops_per_jiffy <<= 1) != 0) {
ticks = jiffies;  /* As you will find out in the section, "Kernel
```

```
                Timers," the jiffies variable contains the
                number of timer ticks since the kernel
                started, and is incremented in the timer
                interrupt handler */

  while (ticks == jiffies); /* Wait until the start
                    of the next jiffy */
  ticks = jiffies;
  /* Delay */
  __delay(loops_per_jiffy);

  /* Did the wait outlast the current jiffy? Continue if
     it didn't */
  ticks = jiffies - ticks;
  if (ticks) break;
}

loops_per_jiffy >>= 1; /* This fixes the most significant bit and is
                  the lower-bound of loops_per_jiffy */
```

上述代码首先假定 loops_per_jiffy 高于 4096 ，这可以转化为处理器速度大约为每秒 100 万条指令，即 1MIPS 。接下来，它等待 jiffy 被刷新（ 1 个新的节拍的开始），并开始运行 delay 循环 __delay(loops_per_jiffy) 。如果这个 delay 循环持续了 1 个 jiffy 以上，将使用以前的 loops_per_jiffy 值（将当前值右移 1 位）修复当前 loops_per_jiffy 的最高位；否则，该函数继续通过左移 loops_per_jiffy 值来探测出其最高位。在内核计算出最高位后，它开始计算低位并微调其精度：

```
loopbit = loops_per_jiffy;
```

```
/* Gradually work on the lower-order bits */
while (lps_precision-- && (loopbit >>= 1)) {
  loops_per_jiffy |= loopbit;
  ticks = jiffies;
  while (ticks == jiffies); /* Wait until the start
                               of the next jiffy */
ticks = jiffies;

  /* Delay */
  __delay(loops_per_jiffy);

  if (jiffies != ticks)       /* longer than 1 tick */
    loops_per_jiffy &= ~loopbit;
}
```

上述代码计算出了 delay 循环跨越 jiffy 边界时 loops_per_jiffy 的低位值。这个被校准的值可被用于获取 BogoMIPS （其实它是一个并非科学的处理器速度指标）。你可以使用 BogoMIPS 作为衡量处理器运行速度的相对尺度。在 1.6Ghz 基于 Pentium M 的笔记本电脑上，根据前述启动过程的打印信息， delay 循环校准的结果趋向于 loops_per_jiffy 的值为 2394935 。获得 BogoMIPS 的方式如下：

```
BogoMIPS = loops_per_jiffy * 1 秒内的 jiffy 数 * delay 循环消耗的指令数（以百万为单位）

= (2394935 * HZ * 2) / (1 million)

= (2394935 * 250 * 2) / (1000000)

= 1197.46 ( 与启动过程打印信息中的值一致 )
```

在本章《内核定时器》一节，将有对 jiffy 、 HZ 和 loops_per_jiffy 更深入的阐述。

Checking HLT Instruction

由于 Linux 内核支持多种硬件平台，启动代码会检查体系结构相关的 bug 。其中一项工作就是验证停机（ HLT ）指令。

x86 处理器的 HLT 指令会将 CPU 置入一种低功耗睡眠模式，直到下一次硬件中断发生之前维持不变。当内核想让 CPU 进入空闲状态时（查看 arch/x86/kernel/process_32.c 文件中定义的 cpu_idle () 函数 ），它会使用 HLT 指令。对于有问题的 CPU 而言，命令行参数 no-hlt 可以禁止 HLT 指令。如果 no-hlt 被设置，在空闲的时候，内核会进行忙等待而不是通过 HLT 给 CPU 降 温。

当 init/main.c 中的启动代码调用 include/asm-your-arch/bugs.h 中定义的 check_bugs() 时，会打印上述信息。

NET: Registered Protocol Family 2

Linux 套接字（ socket ）层是用户空间应用程序访问各种网络协议的统一接口。每个协议通过 include/linux/socket.h 文件中定义的被分配给它的独一无二的家族（ family ）号注册自身。上述打印信息中的 Family 2 代表 AF_INET （ Internet 协议）。启动过程中另一个常见的被打印的信息是 AF_NETLINK （ Family 16 ）。 Netlink socket 提供了用户进程和内核通信的方法。通过 netlink socket 可完成的功能还包括存取路由表和地址解析协议（ ARP ）表（ include/linux/netlink.h 文件给出了完整的用法列表 ）。对于此类任务而言， netlink socket 比系统调用更合适，因为前者具有采用异步机制、更易于实现和可动态连接的优点。

内核中经常使能的另一个协议家族是 AF_UNIX 或 UNIX-domain 套接字。 X Windows 等程序使用它们在同一个系统在进行进程间通信。

Freeing Initrd Memory: 387k Freed

Initrd 是一种由 bootloader 加载的常住内存的虚拟磁盘映像。在内核启动后，会将其挂载为初始根文件系统，这个初始根文件系统中存放着挂载实际根文件系统磁盘分区时所依赖的可动态连接的模块。由于内核可运行于各种各样的存储控制器硬件平台上，把所有可能的磁盘驱动都直接放进基本的内核映像中并非一种灵活的方式。你所使用的系

统的存储设备的驱动被打包放入了 initrd 中，在内核启动后、实际的根文件系统被挂载之前，这些驱动才被加载。使用 mkinitrd 命令可以创建一个 initrd 映像。

2.6 内核提供了一种称为 initramfs 的新功能，它在几个方面较 initrd 更为优秀。后者模拟了一个磁盘（因而被称为 initramdisk 或 initrd ），会带来 Linux 块 I/O 子系统的开销（如缓冲），然后前者基本上如同一个被挂载的文件系统一样，由自身获取缓冲（因此被称作 initramfs ）。

不同于 initrd ，基于页缓冲建立的 initramfs 如同页缓冲一样会动态地变大和缩小，从而减少了其内存消耗。另外， initrd 要求你的内核映像包含了 initrd 所使用的文件系统（例如，如果你的 initrd 为 EXT2 文件系统，内核必须包含 EXT2 驱动），然而 initramfs 不需要文件系统支持。再者，由于 initramfs 只是页缓冲之上的一小层，因此它的代码量很小。

用户可以将初始根文件系统打包为一个 cpio 压缩包 [2] ，并通过 initrd= 命令行参数传递给内核。当然，也可以在内核配置过程中通过 INITRAMFS_SOURCE 选项直接编译进内核。对于后一种方式而言，用户可以提供 cpio 压缩包的文件名或者包含 initramfs 的目录树。在启动过程中，内核会将文件解压缩为一个 initramfs 根文件系统，如果它找到了 /init ，它就会执行该顶层的程序。这种获取初始根文件系统的方法对于嵌入式系统而言特别有用，因为在嵌入式系统中系统资源非常宝贵。使用 mkinitramfs 可以创建一个 initramfs 映像，查看文档 Documentation/filesystems/ramfs-rootfs-initramfs.txt 可获得更多信息。

[2] cpio 是一种 UNIX 压缩文件格式，从 www.gnu.org/software/cpio 可以下载到它。

在本例中，我们使用的是通过 initrd= 命令行参数向内核传递初始根文件系统cpio 压缩包的方式。在将压缩包中的内容解压为根文件系统后，内核将释放该压缩包所占据的内存（本例中为 387K ）并打印上述信息。释放后的页面会被分发给内核中的其他部分以便被申请。

在第 18 章中我们会发现，在嵌入式系统开发过程中， initrd 和 initramfs 有时候也可被用作嵌入式设备上实际的根文件系统。

IO Scheduler Anticipatory Registered (Default)

I/O 调度器的主要目标是通过减少磁盘的定位次数以增加系统的吞吐率。在磁盘定位过程中，磁头需要从当前的位置移动到感兴趣的目标位置，这会带来一定的延迟。 2.6 内核提供了 4 种不同的 I/O 调度器： Deadline 、 Anticipatory 、 Complete Fair Queuing 以及 NOOP 。 从上述内核打印信息可以看出，本例将 Anticipatory 设置为了缺省的 I/O 调度器。在第 14 章《块设备驱动》中，我们将学习 I/O 调度的知识。

Setting Up Standard PCI Resources

启动过程的下一阶段会初始化 I/O 总线和外围控制器。内核会通过遍历 PCI 总线来探测 PCI 硬件，接下来再初始化其他的 I/O 子系统。从图 2.3 中中我们会看到 SCSI 子系统、 USB 控制器、视频芯片（ 855 北桥芯片组信息中的一部分）、串口（本例中为 8250 UART ）、 PS/2 键盘和鼠标、软驱、 ramdisk 、 loopback 设备、 IDE 控制器（本例中为 ICH4 南桥芯片集中的一部分）、触控板、以太网控制器（本例中为 e1000 ）以及 PCMCIA 控制器初始化的启动信息。图 2.3 中—— > 符号指向的为 I/O 设备的标识（ ID ）。

**Figure 2.3. Initializing buses and peripheral controllers during boot.**

Code View:

```
SCSI subsystem initialized                    → SCSI
usbcore: registered new driver hub            → USB
agpgart: Detected an Intel 855 Chipset.       → Video
[drm] Initialized drm 1.0.0 20040925
PS/2 Controller [PNP0303:KBD,PNP0f13:MOU]
at 0x60,0x64 irq 1,12 serio: i8042 KBD port   → Keyboard
serial8250: ttyS0 at I/O 0x3f8 (irq = 4)
is a NS16550A                                 → Serial Port
Floppy drive(s): fd0 is 1.44M                 → Floppy
RAMDISK driver initialized: 16 RAM disks
of 4096K size 1024 blocksize                  → Ramdisk
loop: loaded (max 8 devices)                  → Loop back
ICH4: IDE controller at PCI slot
0000:00:1f.1                                  → Hard Disk
...
input: SynPS/2 Synaptics TouchPad as
/class/input/input1                           → Touchpad
e1000: eth0: e1000_probe: Intel® PRO/1000
Network Connection                            → Ethernet
Yenta: CardBus bridge found at
0000:02:00.0 [1014:0560]                      → PCMCIA/CardBus
...
```

本书会以单独的章节讨论了许多个上述的驱动子系统，请注意如果驱动以模块的形式被动态连接到内核，其中的一些消息也许只有在内核启动后才会被显示。

EXT3-fs: Mounted Filesystem

EXT3 文件系统已经成为 Linux 事实上的文件系统。 EXT3 在退役的 EXT2 文件系统基础上增添了日志层，该层可用于崩溃后文件系统的快速恢复。它的目标是不经由耗时的文件系统检查（ fsck ）操作即可获得一个一致的文件系统。 EXT2 仍然是新文件系统的工作引擎，但是 EXT3 层会在进行实际的磁盘改变之前记录文件交互的日志。 EXT3 向后兼容于 EXT2 ，因此，你可以在你现存的 EXT2 文件系统上批上 EXT3 的大衣或者脱去 EXT3 的大衣以回归到 EXT2 文件系统。

**EXT4**

EXT 文件系统的最新版本是 EXT4 ，自 2.6.19 内核以来， EXT4 已经被增加到了主线 Linux 内核中，但是被注明为“ experimental ”，名称为 ext4dev 。 EXT4 很大程度上向后兼容于 EXT3 ，其主页为 www.bullopensource.org/ext4 。

EXT3 会启动一个称为 kjournald 的内核辅助线程（在接下来的一章中将深入讨论内核线程）来完成日志功能。在 EXT3 投入运转以后，内核挂载根文件系统并做好“业务”上的 准备：

```
EXT3-fs: mounted filesystem with ordered data mode
kjournald starting. Commit interval 5 seconds
VFS: Mounted root (ext3 filesystem).
```

INIT: Version 2.85 Booting

所有 Linux 进程的父进程 init 是内核完成启动序列后运行的第 1 个程序。在 init/main.c 的最后几行，内核会搜索一个不同的位置以定位到 init ：

```
if (ramdisk_execute_command) { /* Look for /init in initramfs */
 run_init_process(ramdisk_execute_command);
}
```

```
if (execute_command) { /* You may override init and ask the kernel
                        to execute a custom program using the
                        "init=" kernel command-line argument. If
                         you do that, execute_command points to the
                        specified program */
  run_init_process(execute_command);
}

/* Else search for init or sh in the usual places .. */
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");
panic("No init found. Try passing init= option to kernel.");
```

init 会接受 /etc/inittab 的指引。它首先执行 /etc/rc.sysinit 中的系统初始化脚本，该脚本的一项最重要的职责就是激活交换（ swap ）分区，这会导致如下启动信息被打印：

```
Adding 1552384k swap on /dev/hda6
```

让我们来仔细看看上述这段话的意思。 Linux 用户进程拥有 3GB 的虚拟地址空间（见《内存分配》一节），构成“工作集”的页被保存在 RAM 中。但是，如果有太多程序需要内存资源，内核会释放一些被使用了的 RAM 页面并将其存储到称为交换空间（ swap space ）的磁盘分区中。根据经验法则，交换分区的大小应该是 RAM 的 2 倍。在本例中，交换空间位于 /dev/hda6 这个磁盘分区，其大小为 1552384K 字节。

接下来， init 开始运行 /etc/rc.d/rcX.d/ 目录中的脚本， X 是 inittab 中定义的运行级别。 Runlevel 是根据预期的工作模式所进入的执行状态。例如，多用户文本模式意味着 runlevel 为 3 ， X Windows 则意味着 runlevel 为 5 。因此，当你看到“ INIT: Entering runlevel 3” 这条信息的时候， init 就已经开始执行 /etc/rc.

d/rc3.d/ 目录中的脚本了。这些脚本会启动动态设备命名子系统（第 4 章《打下基础》中将讨论 udev ），并加载网络、音频、存储设备等驱动所对应的内核模块：

```
Starting udev: [ OK ]

Initializing hardware... network audio storage [Done]

...
```

最后， init 发起虚拟控制台 终端 ，你现在就可以登录了。

**内核模式和用户模式**

MS-DOS 等操作系统在单一的 CPU 模式下执行，但是一些类 UNIX 的操作系统则使用了 2 种模式。在 Linux 机器上， CPU 将或者处于受信任的内核模式，或者处于受限制的用户模式。除了内核本身处于内核模式以外，所有的用户进程都运行在用户模式之上。

内核模式的代码可以无限制地使用完整的 CPU 指令集并访问所有的内核和 I/O 空间。但是，如果用户模式的进程要享有此特权，并必须通过系统调用向设备驱动或其他内核模式的代码请求服务。另外一个不同是，用户模式的代码允许发生缺页，而内核模式的代码则不允许。

在 2.4 和更早的内核中，仅仅用户模式的进程可以被其他进程抢占，除非发生以下情况，否则内核模式代码可以一直独占 CPU ：

（ 1 ） 它自愿放弃 CPU

（ 2 ） 发生中断或异常

2.6 内核引入了内核抢占，大多数内核模式的代码也可以被抢占。

**进程上下文和中断上下文**

内核可以处于两种上下文：进程上下文和中断上下文。在系统调用之后，用户应用程序进入内核空间，此后内核空间针对用户空间相应进程的代表就运行于进程上下文。异步发生的中断会引发中断处理程序被调用，中断处理程序就运行于中断上下文。中断上下文和进程上下文不可能同时发生。

运行于进程上下文的内核代码是可抢占的，而进程上下文则不会被抢占。因此，内核会限制中断上下文的工作，不允许其执行如下操作：

（ 1 ）进入睡眠状态或主动放弃 CPU

（ 2 ）占用 mutex

（ 3 ）执行耗时的任务

（ 4 ）访问用户空间虚拟内存

本书第 4 章《中断处理》一节会对中断上下文进行更深入的讨论。

**内核定时器**

内核中许多部分的工作都高度依赖于时间的推移。 Linux 内核使用了硬件提供的不同的定时器以支持忙等待或睡眠等待等依赖于时间的服务。 忙等待时， CPU 会不断运转，但是睡眠等待时，进程将放弃 CPU 。因此，只有在后者不可取的情况下，才可以考虑使用前者。内核也提供了这样的便利：在特定的时间之后调度某函数运行。

我们首先来讨论一些重要的内核定时器变量（ jiffies 、 HZ 和 xtime ）的含义，接下来，我们会使用 Pentium 时间戳计数器（ TSC ）测量基于 Pentium 的系统的运行次数，之后，我们也分析一下 Linux 怎么使用实时钟（ RTC ）。

**HZ 和 Jiffies**

系统定时器能以可编程的频率中断 CPU 。此频率即为每秒的定时器节拍数，对应着内核变量 HZ 。选择合适的 HZ 值需要权衡。较大的 HZ 值将带来更小的定时器间隔时间，因此进程调度的准确性会更高。但是，更大的 HZ 值也会带来更大的开销和更大的电源消耗，因为更多的 CPU 周期将被耗费在定时器中断上下文中。

HZ 的值依赖于体系结构。在 x86 系统上，在 2.4 内核上，该值缺省设置为 100 ，在 2.6 内核中，该值变为 1000 ，而在 2.6.13 中，它又被降低到了 250. 在基于 ARM 的平台上， 2.6 内核将 HZ 设置为 100 。在目前的内核中，你可以在编译内核时通过配置菜单选择一个 HZ 值。该选项的缺省值依赖于你的发行版。

2.6.21 内核开始支持无节拍的内核（ CONFIG_NO_HZ ），它会根据系统的负载动态触发定时器中断。无节拍系统的实现超出了本章的范围。

j iffies 变量记录了自系统启动依赖，系统定时器已经触发的次数。内核每秒钟将 jiffies 变量增加 HZ 次。因此，对于 HZ 值为 100 的系统， 1 个 jiffy 等于 10 毫秒，而对于 HZ 为 1000 的系统， 1 个 jiffy 仅为 1 毫秒。

为了更好地理解 HZ 和 jiffies 变量，请看下面的取自 IDE 驱动 (drivers/ide/ide.c) 的代码片段，该段代码会一直轮训磁盘驱动器的忙状态：

```
unsigned long timeout = jiffies + (3*HZ);
while (hwgroup->busy) {
 /* ... */
 if (time_after(jiffies, timeout)) {
  return -EBUSY;
 }
 /* ... */
}
return SUCCESS;
```

如果忙条件在 3 秒内被清除，上述代码将返回 SUCCESS ，否则，返回 -EBUSY 。 3*HZ 是 3 秒内的 jiffies 数量。计算出来的超时 jiffies + 3*HZ ，将是 3 秒超时发生后新的 jiffies 值。 time_after() 的功能是将目前的 jiffies 值与请求的超时时间对比，检测溢出。类似函数还包括 time_before() 、 time_before_eq() 和 time_after_eq() 。

jiffies 被定义为 volatile 类型，它会告诉编译器不要优化该变量的存取代码。这样就确保了每个节拍发生的定时器中断处理程序都能更新 jiffies 值，并且循环中的每一步都会重新读取 jiffies 值。

对于 jiffies 向秒转换，可以查看 USB 主机控制器驱动 drivers/usb/host/ehci-sched.c 中的如下代码片段：

```
if (stream->rescheduled) {
 ehci_info(ehci, "ep%ds-iso rescheduled " "%lu times in %lu
        seconds\n", stream->bEndpointAddress, is_in? "in":
        "out", stream->rescheduled,
         ((jiffies – stream->start)/HZ));
}
```

上述调试语句计算出 USB 端点流（见第 11 章《 USB 设备驱动》）被重新调度 stream->rescheduled 次所耗费的秒数。 jiffies-stream->start 是从开始到现在消耗的 jiffies 数量，将其除以 HZ 就得到了秒数值。

假定 jiffies 位 1000 ， 32 位的 jiffies 大约会 50 天的时间越界。由于系统的运行时间可以比该时间长许多倍，因此，内核提供了另一个变量 jiffies_64 以存放 64 位的 jiffies 。连接器讲 jiffies_64 的低 32 位与 32 位的 jiffies 指向同一个地址。在 32 位的机器上，为了将一个 u64 变量赋值给另一个，编译器需要需要 2 条指令，因此，读 jiffies_64 的操作不具备原子性。可以将 drivers/cpufreq/cpufreq_stats.c 文件中定义的 cpufreq_stats_update() 作为实例来学习。

**长延时**

在内核中，以 jiffies 为单位进行的延迟通常被认为是长延时。一种可能但非最佳的实现长延时的方法是忙等待。实现忙等待的函数有“占着茅坑不拉屎”之嫌，它本身不利用 CPU 进行有用的工作，同时还不让其他程序使用 CPU 。如下代码将占用 CPU 1 秒：

```
unsigned long timeout = jiffies + HZ;

while (time_before(jiffies, timeout)) continue;
```

时间长延时的更好方法是睡眠等待而不是忙等待，在这种方式中，本进程会在等待时将 CPU 出让给其他进程， schedule_timeout() 完成此功能：

```
unsigned long timeout = jiffies + HZ;

schedule_timeout(timeout); /* Allow other parts of the

                        kernel to run */
```

这种延时仅仅确保超时时较低的精度，由于只有在时钟节拍引发的内核调度才会更新 jiffies ，所以超时的最大精度是 HZ 。另外，即使你的进程已经超时并可被调度，但是调度器仍然可能基于优先级策略选择运行队列的其他进程 [5] 。

[5] 在 2.6.23 内核中，睡着 CFS 调度器的出现，调度性质发生了改变。第 19 章《用户空间的设备驱动》会对进程调度进行讨论。

用于睡眠等待的另 2 个函数是 wait_event_timeout() 和 msleep() ，此 2 者的实现都基于 schedule_timeout() 。 wait_event_timeout() 的使用场合是：在一个特

定的条件满足或者超时发生后，代码期待继续运行。 msleep() 则用于睡眠指定的毫秒数。

这种长延时技术仅仅实用于进程上下文。睡眠等待不能用于中断上下文，因为中断上下文不允许执行 schedule() 或睡眠（第 4 章的《中断处理》一节给出了中断上下文可以做和不能做的事情）。 在中断中进行短时间的忙等待是可行的，但是进行长时间的忙等则被认为不可赦免的罪行。在中断禁止时，进行长时间的忙等待也被看作禁忌。

为了支持在将来的某时刻进行某项工作，内核也提供了定时器 API 。你可以通过 init_timer() 动态地定义一个定时器，也可以通过 DEFINE_TIMER() 静态创建。此后，将处理函数的地址和参数绑定给一个 timer_list ，并使用 add_timer() 注册它即可：

```
#include <linux/timer.h>


struct timer_list my_timer;


init_timer(&my_timer);          /* Also see setup_timer() */
my_timer.expire = jiffies + n*HZ; /* n is the timeout in number
                            of seconds */
my_timer.function = timer_func;  /* Function to execute
                            after n seconds */
my_timer.data = func_parameter;  /* Parameter to be passed
                            to timer_func */
add_timer(&my_timer);           /* Start the timer */
```

上述代码只会让定时器昙花一现。如果你想 timer_func() 函数被周期性地执行，你需要在 timer_func() 加上相关地代码指定其在下次超时后调度自身：

```
static void timer_func(unsigned long func_parameter)
{
 /* Do work to be done periodically */
 /* ... */
```

```
    init_timer(&my_timer);
    my_timer.expire   = jiffies + n*HZ;
    my_timer.data     = func_parameter;
    my_timer.function = timer_func;
    add_timer(&my_timer);
}
```

你可以使用 mod_timer() 修改 my_timer 的到期时间，使用 del_timer() 取消定时器，或使用 timer_pending() 以查看 my_timer 当前是否处于 pending 状态。 查看 kernel/timer.c 源代码，你会发现 schedule_timeout() 内部就使用了这些 API 。

用户空间的 clock_settime() 和 clock_gettime() 函数可用于获得内核定时器服务。用户应用程序可以使用 setitimer() 和 getitimer() 来控制一个 alarm 信号在特定的超时后发生。

**短延时**

在内核中，小于 jiffy 的延时被认为是短延时。这种延时在进程或中断上下文都可能发生。由于不可能使用基于 jiffy 的方法实现短延时，之前讨论的睡眠等待将不再能用于小的超时。这种情况下，唯一的解决途径就是忙等待。

实现短延时的内核 API 包括 mdelay() 、 udelay() 和 ndelay() ，分别支持毫秒、微妙和纳秒级的延时。这些函数的实际实现依赖于体系结构，而且也并非在所有平台上都被完整实现。忙等待的实现方法是测量 CPU 执行一条指令的时间，为了延时，执行一定数量的指令。从前文可知，内核会在启动过程中进行测量出一个 loops_per_jiffy 值。短延时 API 就使用了 loops_per_jiffy 值来决定它们需要进行循环的数量。 为了实现握手进程中 1 微妙的延时， USB 主机控制器驱动（ drivers/usb/host/ehci-hcd.c ）会调用 udelay() ，而 udelay() 的内部会调用 loops_per_jiffy ：

```
do {
    result = ehci_readl(ehci, ptr);
    /* ... */
    if (result == done) return 0;
```

```
    udelay(1);    /* Internally uses loops_per_jiffy */
    usec--;
  } while (usec > 0);
```

**Pentium 时间戳计数器**

时间戳计数器（ TSC ）是 Pentium 兼容处理器中的一个计数器，它记录自启动以来 CPU 消耗的时钟周期数。由于 TSC 睡着处理器周期速率的比例正常，它提供了非常高的精确度。 TSC 通常被用于剖析 和监测 代码。使用 rdtsc 指令可测量某段代码的执行时间，其精度达到微妙级。 TSC 的节拍可以被转化为秒，方法是将其除以 CPU 时钟速率（包含在内核变量 cpu_khz 中）。

在如下的代码片段中。 low_tsc_ticks 和 high_tsc_ticks 分别包含了 TSC 的低 32 位和高 32 位。低 32 位可能在数秒内溢出（具体时间依赖于处理器速度），但是这已经用于许多代码的 剖析 了：

```
unsigned long low_tsc_ticks0, high_tsc_ticks0;
unsigned long low_tsc_ticks1, high_tsc_ticks1;
unsigned long exec_time;
rdtsc(low_tsc_ticks0, high_tsc_ticks0); /* Timestamp
                                before */
printk("Hello World\n");            /* Code to be
                                 profiled */
rdtsc(low_tsc_ticks1, high_tsc_ticks1); /* Timestamp after */
exec_time = low_tsc_ticks1 - low_tsc_ticks0;
```

在 1.8GHz Pentium 处理器上， exec_time 的结果为 871 （或半微妙）。

在 2.6.21 内核中，针对高精度定时器的支持 (CONFIG_HIGH_RES_TIMERS) 已经被融入了内核。它使用了硬件特定的高速定时器来提供对 nanosleep() 等 API 高精度的支持。 在基于 Pentium 的机器上，内核借助的是 TSC 。

**实时钟**

RTC 在非易失性存储器上记录绝对时间。在 x86 PC 上， RTC 位于由电池供电[4]的 互补金属氧化物半导体 （ CMOS ）存储器的顶部。从第 5 章《字符设备驱动》的图 5.1 可以看出传统 PC 体系结构中 CMOS 的位置。在嵌入式系统中， RTC 可能被集成到处理器中，也可能通过 I2C 或 SPI 总线在外部连接，见第 8 章。

[4]RTC 的电池能够持续使用很多年，通过会超过电脑的使用寿命，因此，你从来都不需要替换它。

使用 RTC ，你可以完成如下工作：

（ 1 ）读取、设置绝对时间，在时钟更新时产生中断；

（ 2 ）产生频率从 2HZ 到 8192HZ 之间的周期性中断；

（ 3 ）设置 alarm

许多应用程序需要使用绝对时间或称墙上时间（ wall time ）。 jiffies 是相对于系统启动后的时间，它不包含墙上时间。内核将墙上时间记录在 xtime 变量中，在启动过程中，会根据从 RTC 读取到的目前的墙上时间初始化 xtime ，在系统停机后，墙上时间会被写回 RTC 。你可以使用 do_gettimeofday() 读取墙上时间，其最高精度 由硬件 决定：

```
#include <linux/time.h>

static struct timeval curr_time;

do_gettimeofday(&curr_time);

my_timestamp = cpu_to_le32(curr_time.tv_sec); /* Record timestamp */
```

用户空间也包含一系列可以访问墙上时间的函数，包括：

（ 1 ） time() ，该函数返回日历时间，或从 Epoch （ 1970 年 1 月 1 日 00:00:00 ）以来经历的秒数；

（ 2 ） localtime() ，以分散的形式返回日历时间；

（ 3 ） mktime() ， 进行 localtime() 的反向工作；

（ 4 ） gettimeofday() ，如果你的平台支持的话，该函数将以微妙精度返回日历时间。

用户空间使用 RTC 的另一种途径是通过字符设备 /dev/rtc 来进行，同一时刻只有一个进程允许返回该字符设备。

在第 5 章和第 8 章，本书将对 RTC 驱动进行更深入的讨论。另外，在第 19 章给出了一个使用 /dev/rtc 以微妙级精度执行周期性工作的应用程序例子。

### 内核中的并发

随着多核笔记本电脑时代的到来，对称多处理器（ SMP ）的使用不再被限于高科技用户。 SMP 和内核抢占是多线程执行的 2 种场景。多个线程能够同时操作共享的内核数据结构，因此，对这些数据结构的访问必须被串行化。

接下来，我们会讨论并发访问情况下保护共享内核资源的基本概念。我们以一个简单的例子开始，并逐步引入中断、内核抢占和 SMP 等复杂概念。

### 自旋锁和互斥体

访问共享资源的代码区域称作临界区。自选锁（ spinlock ）和互斥体（ mutex ， mutual exclusion 的缩写）是保护内核临界区的 2 种基本机制。我们一个一个分析。

自选锁可以确保在同时只有一个线程进入临界区。其他想进入临界区的线程必须不停地原地打转，知道第 1 个线程释放自选锁。

注意：这里所说的线程不仅限于内核线程，还包含用户线程进入内核后的代表。

下面的例子演示了自选锁的基本用法：

```
#include <linux/spinlock.h>

spinlock_t mylock = SPIN_LOCK_UNLOCKED; /* Initialize */


/* Acquire the spinlock. This is inexpensive if there
  * is no one inside the critical section. In the face of
  * contention, spinlock() has to busy-wait.
  */
spin_lock(&mylock);
```

```
/* ... Critical Section code ... */


spin_unlock(&mylock); /* Release the lock */
```

与自选锁不同的是，互斥体在进入一个被占用的临界区之前，不会原地打转而是使当前线程进入睡眠状态。如果要等待的时间较长，互斥体比自选锁会更合适，因为自选锁会消耗 CPU 资源。在使用互斥体的场合，多于 2 次进程切换时间都可被认为是长时间，因此一个互斥体会引起本线程睡眠，而当其被唤醒时，它需要被切换回来。

因此，在很多情况下，决定使用自选锁还是互斥体相对来说很容易：

（1 ）如果临界区需要睡眠，只能使用互斥体，因为在获得自选锁后进行调度、抢占以及在等待队列上睡眠都是非法的；

（2 ）由于互斥体会在面临竞争的情况下将当前线程置于睡眠状态，因此，在中断处理函数中，只能使用自选锁。（在第 4 章中，你将学习到更多的关于中断上下文的限制。）

下面的例子演示了互斥体使用的基本方法：

```
#include <linux/mutex.h>


/* Statically declare a mutex. To dynamically
   create a mutex, use mutex_init() */
static DEFINE_MUTEX(mymutex);


/* Acquire the mutex. This is inexpensive if there
  * is no one inside the critical section. In the face of
  * contention, mutex_lock() puts the calling thread to sleep.
  */
mutex_lock(&mymutex);
```

```
/* ... Critical Section code ... */


mutex_unlock(&mymutex);      /* Release the mutex */
```

为了论证并发保护的用法，我们首先以一个仅存在于进程上下文的临界区开始，并以下面的顺序逐步引入复杂性：

（1）非抢占内核，单 CPU 情况下存在于进程上下文的临界区；

（2）非抢占内核，单 CPU 情况下存在于进程和中断上下文的临界区；

（3）可抢占内核，单 CPU 情况下存在于进程和中断上下文的临界区；

（4）可抢占内核，SMP 情况下存在于进程和中断上下文的临界区。

### 老的信号量接口

互斥体接口代替了老的信号量接口（semaphore），它互斥体诞生于-rt 树，在 2.6.16 内核中被融入主线内核。

尽管如此，但是老的信号量仍然在内核和驱动中被广泛使用。信号量接口的基本用法如下：

```
#include <asm/semaphore.h>  /* Architecture dependent
                            header */


/* Statically declare a semaphore. To dynamically
   create a semaphore, use init_MUTEX() */
static DECLARE_MUTEX(mysem);


down(&mysem);     /* Acquire the semaphore */


/* ... Critical Section code ... */
```

```
up(&mysem);      /* Release the semaphore */
```

信号量可以被配置为允许多个预定数量的线程同时进入临界区，但是，这种用法非常罕见。

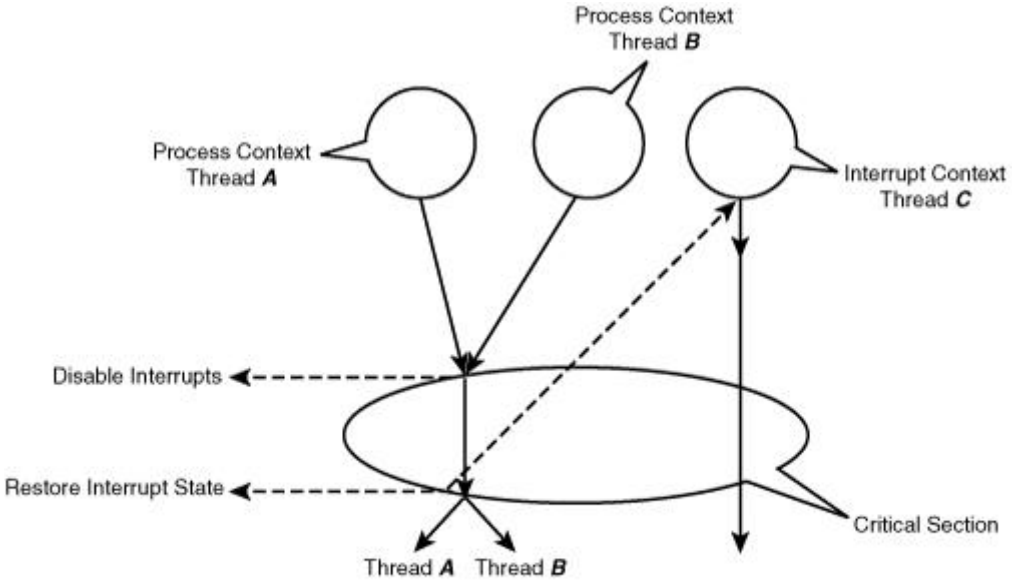
### 案例 1 ：进程上下文，单 CPU ，非抢占内核

这种情况最为简单，不需要加锁，因此不再赘述。

### 案例 2 ：进程和上下文，单 CPU ，非抢占内核

在这种情况下，为了保护临界区，仅仅需要禁止中断。如图 2.4 ，假定进程上下文的执行单元 A 、B 以及中断上下文的执行单元 C 都企图进入相同的临界区。

**图 2.4 进程和中断上下文进入临界区**




由于执行单元 C 总是在中断上下文执行，它会优先于执行单元 A 和 B ，因此，它不要担心保护的问题。执行单元 A 和 B 也不必关心彼此会被互相打断，因为内核是非抢占的。因此，执行单元 A 和 B 仅仅需要担心 C 会在它们进入临界区的时候横行进入。为了实现此目的，它们会在进入临界区之前禁止中断：

```
Point A ：
```

```
local_irq_disable(); /* Disable Interrupts in local CPU */

/* ... Critical Section ... */

local_irq_enable();  /* Enable Interrupts in local CPU */
```

但是，如果当执行到 Point A 的时候已经被禁止，local_irq_enable() 将产生副作用，它会重新使能中断，而不是恢复之前的中断状态。可以这样修复它：

```
unsigned long flags;


Point A:

local_irq_save(flags);    /* Disable Interrupts */

/* ... Critical Section ... */

local_irq_restore(flags); /* Restore state to what

                          it was at Point A */
```

不论 Point A 的中断处于什么状态，上述工作都将正确执行。

### 案例 3 ：进程和中断上下文，单 CPU ，抢占内核

如果内核使能了抢占，仅仅禁止中断将不再能确保对临界区的保护，因为另一个处于进程上下文的执行单元可能会进入临界区。重新回到图 2.4 ，现在，除了 C 以外，执行单元 A 和 B 必须提防彼此。显而易见，解决该问题的方法是在进入临界区之前禁止内核抢占、中断，并在退出临界区的时候恢复内核抢占和中断。因此，执行单元 A 和 B 使用了自选锁 API 附带 irq 的变体：

```
unsigned long flags;


Point A:

/* Save interrupt state.

 * Disable interrupts - this implicitly disables preemption */

spin_lock_irqsave(&mylock, flags);
```

```
/* ... Critical Section ... */


/* Restore interrupt state to what it was at Point A */
spin_unlock_irqrestore(&mylock, flags);
```

我们不需要在最后显示地恢复 Point A 的抢占状态，因为内核自身会通过一个名叫抢占计数器的变量维护它。在抢占被禁止时（通过调用 preempt_disable() ），计数器会增加；在抢占被使能时（通过调用 preempt_enable() ），计数器被减少。只有在计数器为 0 的时候，抢占才发挥作用。

### 案例 4 ：进程和中断上下文， SMP 机器，抢占内核

现在临界区执行于 SMP 机器上，而且你的内核配置了 CONFIG_SMP 和 CONFIG_PREEMPT 。到目前为止讨论的场景中，自旋锁原语发挥的作用仅限于使能和禁止抢占和中断，时间的锁功能并未被完全编译进来。在 SMP 机器内，锁逻辑被编译进来，而且自旋锁原语确保了 SMP 安全性。SMP 使能的含义如下：

```
unsigned long flags;


Point A:
 /*
   - Save interrupt state on the local CPU
   - Disable interrupts on the local CPU. This implicitly disables
     preemption.
   - Lock the section to regulate access by other CPUs
  */
 spin_lock_irqsave(&mylock, flags);


 /* ... Critical Section ... */


 /*
```

```
     - Restore interrupt state and preemption to what it
       was at Point A for the local CPU
     - Release the lock
    */
   spin_unlock_irqrestore(&mylock, flags);
```

在 SMP 系统上，获取自旋锁时，仅仅本 CPU 上的中断被禁止。因此，一个进程上下文的执行单元（图 2.4 中的执行单元 A ）在一个 CPU 上运行的同时，一个中断处理函数（图 2.4 中的执行单元 C ）可能运行在另一个 CPU 上。非本 CPU 上的中断处理函数必须自旋等待本 CPU 上的进程上下文代码退出临界区。中断上下文需要调用 spin_lock()/spin_unlock() ：

```
spin_lock(&mylock);

/* ... Critical Section ... */

spin_unlock(&mylock);
```

除了有 irq 变体以外，自旋锁也有底半部（BH ）变体。在锁被获取的时候，spin_lock_bh() 会禁止底半部，而 spin_unlock_bh() 则会在锁被释放时重新使能底半部。我们将在第 4 章讨论底半部。

### -rt 树

实时（-rt ）树，也被称作CONFIG_PREEMPT_RT 补丁集，实现了内核中一些针对低延时的修改。该补丁集可以从www.kernel.org/pub/linux/kernel/projects/rt 下载，它允许内核的大部分位置可被抢占，但是用自旋锁代替了一些互斥体。它也合并了一些高精度的定时器。数个-rt 功能已经被融入了主线内核。在工程的wiki 页上，能找到详细的文档，地址为http://rt.wiki.kernel.org/ 。

为了提供性能，内核也定义了一些针对特定环境的特定的锁原语。使能实用于你的代码执行场景的互斥机制将使你的代码更高效。下面我们来看一下这些特定的互斥机制。

## 原子操作

原子操作原子执行轻量级的、仅执行一次的操作，例如修改计数器、有条件的增 1 、设置位等。原子操作可以确保操作的串行化，不再需要锁进行并发访问保护。原子操作的具体实现依赖于体系结构。

为了在释放内核网络缓冲区（称为 skbuff ）之前检查是否还有余留的数据引用，定义于 net/core/skbuff.c 的 skb_release_data() 函数将进行如下操作：

```
if (!skb->cloned ||
  /* Atomically decrement and check if the returned value is zero */
    !atomic_sub_return(skb->nohdr ? (1 << SKB_DATAREF_SHIFT) + 1 :
                  1,&skb_shinfo(skb)->dataref)) {
  /* ... */
  kfree(skb->head);
}
```

当 skb_release_data() 执行的时候，另一个调用 skbuff_clone() （也定义于 net/core/skbuff.c ）的执行单元也许在同步地增加数据引用计数:

```
/* ... */
/* Atomically bump up the data reference count */
atomic_inc(&(skb_shinfo(skb)->dataref));
/* ... */
```

原子操作的使用将确保数据引用计数不会被这 2 个执行单元“蹂躏”。它也消除了使用锁去保护单一整型变量的争论。

内核也支持 set_bit() 、clear_bit() 和 test_and_set_bit() 操作，它们可用于原子地进行位修改。查看 include/asm-your-arch/atomic.h 文件可以看出你所在体系结构所支持的原子操作。

### 读者—写者锁

另一个特定的并发保护机制是自旋锁的变体读者—写者锁。如果每个执行单元在访问临界区的时候要么是读，要么是写共享的数据结构，但是它们都不会同时进行读和写操

作，这种锁是最好的选择。多则读执行单元被允许同时进入临界区。读者自旋锁可以这样定义：

```
rwlock_t myrwlock = RW_LOCK_UNLOCKED;


read_lock(&myrwlock);     /* Acquire reader lock */
/* ... Critical Region ... */
read_unlock(&myrwlock);   /* Release lock */
```

但是，如果一个写执行单元进入了临界区，其他的读和写都不被允许进入。写者锁的用法如下：

```
rwlock_t myrwlock = RW_LOCK_UNLOCKED;


write_lock(&myrwlock);    /* Acquire writer lock */
/* ... Critical Region ... */
write_unlock(&myrwlock);  /* Release lock */
```

net/ipx/ipx_route.c 中的 IPX 路由代码是使用读者—写者锁的真实例子。一个称作 ipx_routes_lock 的读者—写者锁将保护 IPX 路由表的并发访问。要通过查找路由表实现包转发的执行单元需要请求读者锁。需要添加和删除路由表中入口的执行单元必须获取写者锁。由于通过读路由表的情况比更新路由表的情况多地多，使用读者—写者锁重大地提供了性能。

和传统的自旋锁一样，读者—写者锁也有相应的 irq 变体：read_lock_irqsave() 、read_lock_irqrestore() 、write_lock_irqsave() 和 write_lock_irqrestore() 。这些函数的含义与传统自旋锁相应的变体相似。

2.6 内核引入的顺序锁（seqlock ）是一种支持写着多于读者的读者—写者锁。在在一个变量的写操作比读操作多地多的情况下，这种锁非常有用。前文讨论的 jiffies_64 变量就是使用顺序锁的一个例子。写执行单元不必等待一个已经进入临界区的读者，因此，读执行单元也许会它们进入临界区的操作会失败因此需要重试：

```
u64 get_jiffies_64(void) /* Defined in kernel/time.c */
```

```
{
  unsigned long seq;
  u64 ret;
  do {
    seq = read_seqbegin(&xtime_lock);
    ret = jiffies_64;
  } while (read_seqretry(&xtime_lock, seq));
  return ret;
}
```

写者会使用 write_seqlock() 和 write_sequnlock() 保护临界区。

2.6 内核还引入了另一种称为读—拷贝—更新（RCU ）的机制，该锁用于提高读操作远多于写操作时的性能。其基本理念是读执行单元不需要加锁，但是写执行单元会变得更加复杂，它们会在数据结构的一份拷贝上执行更新操作，并代替读者看到的指针。为了确保所有正在进行的读操作的完成，原子拷贝会一直被保持到所有 CPU 上的下一次上下文切换。使用 RCU 的情况很复杂，因此，只有在确保你确实需要使用它而不是前文的其他原语的时候，才适宜选择它。include/linux/rcupdate.h 文件中定义了 RCU 的数据结构和接口函数，Documentation/RCU/* 包含丰富的文档。

fs/dcache.c 文件中包含一个 RCU 的使用例子。在 Linux 中，每个文件都与一个目录入口信息（dentry 结构体）、元数据信息（存放在 inode 中）和实际的数据（存放在数据块中）关联。每次你操作一个文件的时候，文件路径中的组件会被解析，相应的 dentry 会被获取。为了加速未来的操作，dentry 结构体被缓存在称为 dcache 的数据结构中。任何时候，对 dcache 进行查找的数量都远多于 dcache 的更新操作，因此，对 dcache 的访问适宜用 RCU 原语进行保护。

### 调试

由于难于重现，并发相关的问题通常非常难调试。在编译和测试你的代码的时候使能 SMP （CONFIG_SMP ）和抢占（CONFIG_PREEMPT ）是一种很好的理念，即便你的产品将运行在当CPU 、禁止抢占的情况下。在Kernel hacking 下有一个称为Spinlock and rw-lock debugging 的配置选项CONFIG_DEBUG_SPINLOCK ，它能帮助你找到

一些常见的自旋锁错误。Lockmeter （http://oss.sgi.com/projects/lockmeter/ ）等工具可用于锁相关的统计信息。

一个常见的并发问题在访问共享资源之前忘记加锁。这会导致一些不同的执行单元杂乱地竞争。这种问题，被称作“竞态”，可能会导致一些其他的行为。

另一个可能的问题是在某些代码路径里你忘记了释放锁，这会导致死锁。为了理解这个问题，让我们分析如下代码：

```
spin_lock(&mylock);     /* Acquire lock */


/* ... Critical Section ... */


if (error) {            /* This error condition occurs rarely */
  return -EIO; /* Forgot to release the lock! */
}


spin_unlock(&mylock);   /* Release lock */
```

在 if (error) 被满足后，任何要获取 mylock 的执行单元都会死锁，内核也可能因此而冻结。

如果在你写完代码的数月或数年以后首次出现了问题，回过头来调试它将变地更为辣手。（在第 21 章《设备驱动的调试》一章的《Kdump 》一节有一个相关的调试例子。）因此，为了避免遭遇这种不快，在你设计软件的结构的时候，就应该考虑并发逻辑。

### p roc 文件系统

proc 文件系统（procfs ）是一种虚拟的文件系统，它创建内核内部的视窗。你浏览 procfs 看到的数据是在内核运行过程中产生的。procfs 中文件可被用于配置内核参数、查看内核结构体、从设备驱动中收集统计信息或者获取通用的系统信息。

Procfs 是一种虚假的文件系统，这意味着驻留于 procfs 中的文件并不与物理存储设备如硬盘等关联。相反地，这些文件中的数据由内核中相应的入口点按需动态创建。因

此，procfs 中的文件大小都显示为 0 。Procfs 通常被在启动过程中挂载在/proc 目录，通过运行 mount 命令你可以看出这一点。

为了取得 procfs 能力的第一感觉，请查看/proc/cpuinfo 、/proc/meminfo 、 /proc/interrupts 、/proc/tty/driver/serial 、/proc/bus/usb/devices 和/proc/stat 的内容。通过写/proc/sys/ 目录中的文件可以在运行时修改某些内核参数。例如，通过向/proc/sys/kernel/printk 文件 echo 一个新的值，你可以改变内核 printk 日志的级别。许多实用程序（如 ps ）和系统性能监视工具（如 sysstat ）就是通过驻留于/proc 中的文件来获取信息的。

2.6 内核引入的 seq 文件简化了大的 procfs 操作。附录 C 《Seq 文件》对此进行了描述。

### 内存分配

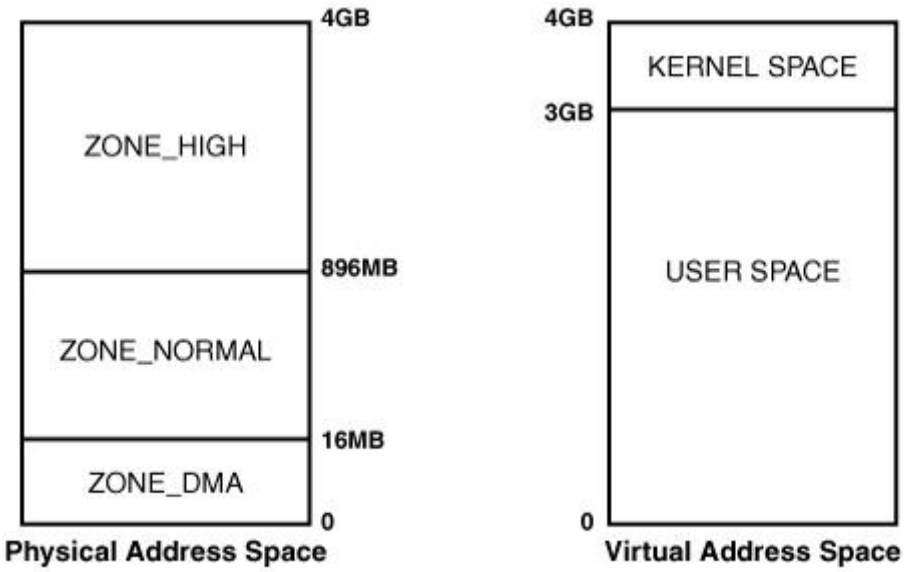
一些设备驱动必须意识到内存 zone 的存在，另外，许多驱动需要内存分配函数的服务。本节我们将简要地对此二者进行讨论。

内核会以分页形式组织物理内存，而页大小则依赖于具体的体系结构。在基于 x86 的机器上，其大小为 4096 字节。物理内存中的每一页都有一个与之对应的 page 结构体（定义在 include/linux/mm_types.h 文件中）：

```
struct page {
  unsigned long flags; /* Page status */
  atomic_t _count;     /* Reference count */
  /* ... */
  void * virtual;      /* Explained later on */
};
```

在 32 位 x86 系统上，缺省的内核配置会将 4GB 的地址空间分成给用户空间的 3GB 的虚拟内存空间和给内核空间的 1GB 的空间（如图 2.5 ）。这导致内核能处理的处理内存有 1GB 的限制。现实情况是，限制为 896MB ，因为地址空间的 128MB 已经被内核数据结构占据。通过改变 3GB/1GB 的分割线你可以增加这个限制，但是由于减少了用户进程虚拟地址空间的大小，对于内存密集型的应用程序，可能会导致一些问题。

图 2.5 32 位 PC 系统上缺省的地址空间分布




内核中用于映射低于 896MB 物理内存的地址与物理地址之间存在线性便宜，被称作逻辑地址。在支持“高端内存”的情况下，在通过特定的方式映射这些区域产生对应的虚拟地址后，内核将能访问超过 896MB 的内存。所有的逻辑地址都是内核虚拟地址，而所有的虚拟地址并非一定是逻辑地址。

因此，存在如下的内存 zone:

（1 ）ZONE_DMA (<16MB) ，该 zone 用于直接内存访问（DMA ）。由于传统的 ISA 设备有 24 条地址线，只能访问开始的 16MB ，因此，内核将该区域献给了这些设备；

（2 ）ZONE_NORMAL (16MB to 896MB)，常规地址区域，也被称作低端内存。低端内存页的 page 结构体的 virtual 字段包含了对应的逻辑地址；

（3 ）ZONE_HIGH (>896MB)，仅仅在通过 kmap() 映射页为虚拟地址后才能访问。（通过 kunmap() 可去除映射）。相应的内核地址为虚拟地址而非逻辑地址。如果相应的页未被映射的话，高端内存页的 page 结构体的 virtual 字段将指向 NULL 。

kmalloc() 是一个用于从 ZONE_NORMAL 区域返回连续内存的内存分配函数，其原型如下：

```
void *kmalloc(int count, int flags);
```

count 是要分配的字节数，flags 是一个模式说明符。支持的所有标志列在 include/linux./gfp.h 文件中（gfp 的意思是“get free page ”），如下的标志最常用：

（1 ）GFP_KERNEL ，被进程上下文用来分配内存。如果指定了该标志，kmalloc() 将被允许睡眠以等待其他页被释放；

（2 ）GFP_ATOMIC ，用于在中断上下问获取内存。在这种模式下，kmalloc() 不允许进行睡眠等待以获得空闲页，因此 GFP_ATOMIC 分配成功的可能性比用 GFP_KERNEL 低。

由于 kmalloc() 返回的内存保留了“前世”的内容，因此，如果将它暴露给用户空间，可到会导致安全问题，因此我们可以 kzalloc() 获得被填充为 0 的内存。

如果你需要分配大的内存缓冲区，而且也不要求内存在物理上联系，可以用 vmalloc() 代替 kmalloc() ：

```
void *vmalloc(unsigned long count);
```

count 是要请求分配的内存大小，该函数返回内核虚拟地址。

Vmalloc() 允许比 kmalloc() 更大的分配尺寸，但是它更慢，而且不能从中断上下文调用。另外，你不能用 vmalloc() 返回的物理上不连续的内存执行 DMA 。在设备打开时，高性能的网络驱动通常会使用 vmalloc() 来分配较大的描述符环行缓冲区。

内核还提供了一些更复杂的内存分配技术，包括后备缓冲区（look aside buffer ）、slab 和 mempool ，它们超出了本章的范围。

## 查看源代码

内存启动始于执行 arch/x86/boot/ 目录中的实模式汇编代码。查看 arch/x86/kernel/setup_32.c 文件可以看出保护模式的内核怎样获取实模式内核收集的信息。

第一条信息来自于 init/main.c 中的代码，深入挖掘 init/calibrate.c 可以对 BogoMIPS 校准理解地更清楚，而 include/asm-your-arch/bugs.h 则包含体系结构相关的检查。

内核中的时间服务由驻留于 arch/your-arch/kernel/ 中的体系结构相关的部分和实现于 kernel/timer.c 中的通用部分组成。从 include/linux/time*.h 可以获取相关的定义。

jiffies 定义于 linux/jiffies.h 文件。HZ 的值与处理器相关，可以从 include/asm-your-arch/param.h 找到。

内存管理源代码存放在顶级 mm/ 目录中。

表 2.1 给出了本章中主要的数据结构以及其在源代码树中定义的位置。表 2.2 则列出了本章中主要内核编程接口及其定义的位置。

**表 2.1 数据结构总结**

| 数据结构 | 位置 | 描述 |
|---|---|---|
| HZ | include/asm-your-arch/param.h | Number of times the system timer ticks in 1 second |
| loops_per_jiffy | init/main.c | Number of times the processor executes an internal delay-loop in 1 jiffy |
| timer_list | include/linux/timer.h | Used to hold the address of a routine that you want to execute at some point in the future |
| timeval | include/linux/time.h | Timestamp |
| spinlock_t | include/linux/spinlock_types.h | A busy-locking mechanism to ensure that only a single thread enters a critical section |
| semaphore | include/asm-your-arch/semaphore.h | A sleep-locking mechanism that allows a predetermined number of users to enter a critical section |

| 数据结构 | 位置 | 描述 |
| --- | --- | --- |
| mutex | include/linux/mutex.h | The new interface that replaces semaphore |
| rwlock_t | include/linux/spinlock_types.h | Reader-writer spinlock |
| page | include/linux/mm_types.h | Kernel's representation of a physical memory page |

表 2.2 内核编程接口总结

| 内核接口 | 位置 | 描述 |
| --- | --- | --- |
| time_after()<br>time_after_eq()<br>time_before()<br>ime_before_eq() | include/linux/jiffies.h | Compares the current value of jiffies with a specified future value |
| schedule_timeout() | kernel/timer.c | Schedules a process to run after a specified timeout has elapsed |
| wait_event_timeout() | include/linux/wait.h | Resumes execution if a specified condition becomes true or if a timeout occurs |

| 内核接口 | 位置 | 描述 |
| --- | --- | --- |
| DEFINE_TIMER() | include/linux/timer.h | Statically defines a timer |
| init_timer() | kernel/timer.c | Dynamically defines a timer |
| add_timer() | include/linux/timer.h | Schedules the timer for execution after the timeout has elapsed |
| mod_timer() | kernel/timer.c | Changes timer expiration |
| timer_pending() | include/linux/timer.h | Checks if a timer is pending at the moment |
| udelay() | include/asm-your-arch/delay.h arch/your-arch/lib/delay.c | Busy-waits for the specified number of microseconds |
| rdtsc() | include/asm-x86/msr.h | Gets the value of the TSC on Pentium-compatible processors |
| do_gettimeofday() | kernel/time.c | Obtains wall time |
| local_irq_disable() | include/asm-your-arch/system.h | Disables interrupts on the local CPU |
| local_irq_enable() | include/asm-your-arch/system.h | Enables interrupts on the local CPU |

| 内核接口 | 位置 | 描述 |
| --- | --- | --- |
| local_irq_save() | include/asm-your-arch/system.h | Saves interrupt state and disables interrupts |
| local_irq_restore() | include/asm-your-arch/system.h | Restores interrupt state to what it was when the matching local_irq_save() was called |
| spin_lock() | include/linux/spinlock.h kernel/spinlock.c | Acquires a spinlock. |
| spin_unlock() | include/linux/spinlock.h | Releases a spinlock |
| spin_lock_irqsave() | include/linux/spinlock.h kernel/spinlock.c | Saves interrupt state, disables interrupts and preemption on local CPU, and locks their critical section to regulate access by other CPUs |
| spin_unlock_irqrestore() | include/linux/spinlock.h kernel/spinlock.c | Restores interrupt state and preemption and releases the lock |
| DEFINE_MUTEX() | include/linux/mutex.h | Statically declares a mutex |
| mutex_init() | include/linux/mutex.h | Dynamically declares a mutex |

| 内核接口 | 位置 | 描述 |
| --- | --- | --- |
| mutex_lock() | kernel/mutex.c | Acquires a mutex |
| mutex_unlock() | kernel/mutex.c | Releases a mutex |
| DECLARE_MUTEX() | include/asm-your-arch/semaphore.h | Statically declares a semaphore |
| init_MUTEX() | include/asm-your-arch/semaphore.h | Dynamically declares a semaphore |
| up() | arch/your-arch/kernel/semaphore.c | Acquires a semaphore |
| down() | arch/your-arch/kernel/semaphore.c | Releases a semaphore |
| atomic_inc()<br>atomic_inc_and_test()<br>atomic_dec()<br>atomic_dec_and_test()<br>clear_bit()<br>set_bit()<br>test_bit()<br>test_and_set_bit() | include/asm-your-arch/atomic.h | Atomic operators to perform lightweight operations |

| 内核接口 | 位置 | 描述 |
| --- | --- | --- |
| read_lock()<br>read_unlock()<br>read_lock_irqsave()<br>read_lock_irqrestore()<br>write_lock()<br>write_unlock()<br>write_lock_irqsave()<br>write_lock_irqrestore() | include/linux/spinlock.h<br>kernel/spinlock.c | Reader-writer variant of spinlocks |
| down_read()<br>up_read()<br>down_write()<br>up_write() | kernel/rwsem.c | Reader-writer variant of semaphores |
| read_seqbegin()<br>read_seqretry()<br>write_seqlock()<br>write_sequnlock() | include/linux/seqlock.h | Seqlock operations |

| 内核接口 | 位置 | 描述 |
| --- | --- | --- |
| kmalloc() | include/linux/slab.h<br>mm/slab.c | Allocates physically contiguous memory from ZONE_NORMAL |
| kzalloc() | include/linux/slab.h<br>mm/util.c | Obtains zeroed kmalloced memory |
| kfree() | mm/slab.c | Releases kmalloced memory |
| vmalloc() | mm/vmalloc.c | Allocates virtually contiguous memory that is not guaranteed to be physically contiguous. |