第 4 章　系统初始化

当你想要运行程序时，你需要把程序的文件名敲入 shell——或者更为流行的，在如 GNOME 或者 KDE 等之类桌面环境中点击相应的图标——这样就能将其装载进内核并运行。但是，首先必须有其它的软件来装载并运行内核；这通常是诸如 LOADLIN 或者 LILO 之类的内核引导程序。更进一步，我们还需要其它的软件来装载运行内核引导程序——称之“内核引导程序的引导程序”——而且看起来似乎运行内核引导程序的引导程序也需要内核引导程序的引导程序的引导程序，等等，这个过程是无限的。

这个无限循环的过程必然最终在某个地方终止，这就是硬件。因此，在最低的层次上，启动系统的第一步是从硬件中获得帮助。该硬件总是运行一些短小的内置程序——软件，但是这些软件是被固化在只读存储器中，存储在已知地址中。因此，在这种情况下就不需要软件引导程序了——它能够运行更大更复杂的程序，直到内核自身装载成功为止。按照这种方式，系统自己的引导过程（bootstrap）会引发系统的启动，当然这只是术语“系统引导（booting）”的一个比喻。虽然不同体系结构的引导过程的具体细节差异很大，但是它们的原则都基本相同。

前面的工作都完成以后，内核就已经成功装载了。随后内核可以初始化自身以及系统的其它部分。

本章首先将简单介绍基于 x86 PC 机的典型自启动方式，接着回顾一下每一步工作在什么时机发生，最后我们还要介绍的是内核的相应部分。

引导 PC 机

本节简要介绍 x86 PC 是如何引导的。本节的目的不是让你精通 PC 是怎样引导的——这超出了本书的范围——而是向你展示特定体系结构一般的引导方式，为下文中的内核初始化进行铺垫。

首先，机器中的每个 CPU 都要自行初始化，接着可能要用几分之一秒的时间来执行自测试。在多处理器的系统中，这个过程会更复杂些——但是实际上也并不多。在双处理器的 Pentium 系统中，一个 CPU 总是作为主 CPU 存在，另外一个 CPU 则是辅 CPU。主 CPU 执行启动过程中的剩余工作，随后内核才会激活辅 CPU。在多处理器的 Pentium Pro 系统中，CPU 必须根据 Intel 定义的算法“抢夺标志”——来动态决定由哪个 CPU 启动系统。取得标志的 CPU 启动系统，随后内核激活其它的 CPU。无论是哪种情况，启动程序的剩余部分只与一个 CPU 有关。这样，在随后的一段时间内，我们可以认为该系统中只有一个 CPU 是可用的，而不考虑其它的 CPU，或者说这些 CPU 被暂时隐藏了。另一方面，内核还需要明确的激活所有其它的 CPU——这一点你可以在本章后续部分看到。

接下来，CPU 从 0xfffffff0 单元中取得指令并执行，这个地址非常接近于 32 位 CPU 的最后可用的地址。因为大多数 PC 都没有 4GB 的 RAM，所以通常在这个地址上并没有实际内存的。内存硬件可以虚拟使用它。对那些确实有 4GB 内存的机器来说，它们也只是仅仅损失了供 BIOS 使用的顶端地址空间末尾的少量内存（实际上 BIOS 在这里只保留了 64K 的空间——这种损失在 4GB 的机器中是可以忽略的）。

该地址单元中存储的指令是一条跳转指令，这条指令跳转到基本输入输出（BIOS）代码的首部。BIOS 内置在主板中，它主要负责控制系统的启动。请注意 CPU 实际上并不真正关心 BIOS 是否存在，这样就使得在诸如用户定制的嵌入系统之类的非 PC 体系结构的计算

机中使用 Intel 的 CPU 成为可能。CPU 执行在目标地址中发现的任何指令，在这里使用跳转指令转移到 BIOS 只是 PC 体系结构的一部分。（实际上，跳转指令自己是 BIOS 的一部分，但是这不是考虑这个问题的最方便的方法。）

BIOS 使用内置的规则来选择启动设备。通常情况下，这些规则是可以改变的，方法是在启动过程开始时按下一个键（例如，在我的系统中是 Delete 键）并通过一些菜单选项浏览选择。但是，通常的过程是 BIOS 首先试图从软盘启动，如果失败了，就再试图从主硬盘上启动。如果又失败了，就再试图从 CD-ROM 上启动。为了使问题更具体，这里讨论的情况假定是最普通的，也就是启动设备是硬盘。

从这种启动设备上启动，BIOS 读取第一个扇区的信息——首 512 个字节——称之为主引导记录（MBR）。接下来发生的内容有赖于 Linux 是怎样在系统上安装的。为使讨论形象具体，我们假定 LILO 是内核的载入程序。在典型的设置中，BIOS 检测 MBR 中的关键数字（为了确认该数据段的确是 MBR）并在 MBR 中检测引导扇区的位置。这一扇区包含了 LILO 的开始部分，然后 BIOS 将其装入内存，开始执行。

注意我们现在已经实现了从硬件和内置软件的范围到实际软件范围的转变，从有形范围到无形范围，也就是说从你可以接触的部分到不可接触的部分。

下面就是 LILO 的责任了。它把自己其余的部分装载进来，在磁盘上找到配置数据，这些数据指明从什么地方可以得到内核，启动时要通过什么选项。LILO 接着装载内核到内存并跳转到内核。

通常，内核以压缩形式存储，只有少量指令足以完成解压缩的任务，也就是自解压可执行文件，是以非压缩形式存储的。因此，内核的下一步工作是自解压缩内核镜像。到这里，内核就已经完成了装载的过程。

下面是到现在进行的步骤地简要描述：

1. CPU 初始化自身，接着在固定位置执行一条指令。
2. 这条指令跳转到 BIOS 中。
3. BIOS 找到启动设备并获取 MBR，该 MBR 指向 LILO。
4. BIOS 装载并把控制权转交给 LILO。
5. LILO 装载压缩内核。
6. 压缩内核自解压并把控制权转交给解压的内核

正如你所见到的，引导过程每一步都将你带入更大量更复杂的代码块中，一直到最后成功地运行了内核为止。

依赖于你计算层次的方式，CPU 成为内核引导程序的引导程序的引导程序的引导程序（CPU 装载 BIOS，BIOS 装载 LILO，LILO 装载压缩内核，压缩内核装载解压内核；但是你可以合理的考虑是否这些步骤都满足引导程序的定义）。

初始化 Linux 内核

在内核成功装入内存（如果需要就解压缩）以及一些关键硬件，例如已经在低层设置过的内存管理器（MMU，请参看第 8 章）之后，内核将跳转到 **start_kernel**（19802 行）。这个函数完成其余的系统初始化工作——实际上，几乎所有的初始化工作都是由这个函数实现的。因此，**start_kernel** 就是本节的核心。

start_kernel

19802：**__init** 标示符在 gcc 编译器中指定将该函数置于内核的特定区域。在内核完成自身初始化之后，就试图释放这个特定区域。实际上，内核中存在两个这样的区

域，.text.init 和.data.init——第一个是代码初始化使用的，另外一个是数据初始化使用的。（诸如可以在进程间共享的代码和字符串常量之类的“文本（Text）”是在可执行程序中的“纯区域”中使用的一个术语。）另外你也可以看到**__initfunc** 和**__initdata** 标志，前者和**__init** 类似，标志初始化专用代码，后者则标志初始化专用数据。

19807：如前所述，即使在多处理器系统中，在启动时也只使用一个 CPU。Intel 称之为引导程序处理器（Bootstrap Processor，简称为 BSP），它在内核代码的某些地方有时也称之为 BP。BSP 首次运行这一行时，跳过后面的 **if** 语句，并减小 **boot_cpu** 标志，从而当其它 CPU 运行到此处时，都要运行 if 语句。等到其它 CPU 被激活执行到这里时，BSP 已经在 idle 循环中了（本章稍后会更详细的讨论这个问题），**initialize_secondary**（4355 行）负责把其它 CPU 加入到 BSP 中。这样，其它 CPU 就不用执行 **start_kernel** 的剩余部分了——这也是一件好事，因为这意味着不用再进行对许多硬件进行冗余初始化等工作了。

顺便说一下，这种奇异的小小的改动只有对于 x86 是必需的；对于其它平台，调用 **smp_init** 完全可以处理 SMP 设置的其它部分，这一点马上就会讨论。因此，其它平台的 **initialize_secondary** 的定义都是空的。

19816：打印内核标题信息（20099 行），这里显示了有关内核如何编译的信息，包括在什么机器上编译，什么时间编译，使用什么版本的编译器，等等。如果中间任何一步发生了错误，在寻找机器不能启动的原因时查明内核的来源是一个有用的线索。

19817：初始化内核自身的部分组件——内存，硬件中断，调度程序，等等。尤其是 **setup_arch** 函数（19765 行）完成体系结构相关的设置，此后在 **command_line**（传递到内核的参数，在下面讨论）、**memory_start** 和 **memory_end**（内核可用物理地址范围）中返回结果。下面这些函数都希望驻留在内存低端的；它们使用 **memory_start** 和 **memory_end** 来传递该信息。在函数获得所希望的值后，返回值指明了新的 **memory_start** 的值。

19823：分析传给内核的各种选项。**parse_options** 函数（19707 行，在随后的分析内核选项一节中讨论）也设置了 **argv** 和 **envp** 的初值。

19833：内核运行过程中也可以自行对所进行的工作进行记录，周期性地对所执行的指令进行抽样，并使用所获得的结果更新表格。这在定时器中断过程中通过调用 **x86_do_profile**（1896 行）来实现，该部分将在第 6 章中介绍。

如图 4.1 中说明的那样，这个表格把内核划分为几个大小相同的范围，并简单跟踪在一次中断的时间内每个范围中运行多少条指令。这种记录当然是非常粗糙的——甚至不是依据函数和行号进行划分的，而只是使用近似的地址——但是这样代价很低、快速、短小，而且有助于专家判断最关键的问题要点。每个表格条目所涉及到地址的多少——还有问题发生地点的不确定性——可以通过简单修改 **prof_shift**（26142 行）来调节。**profile_setup**（19076 行，在本章中后面讨论）可以让你在启动的时候设置 **prof_shift** 的值，这样比为修改这个数字而重新编译内核要清晰方便得多。

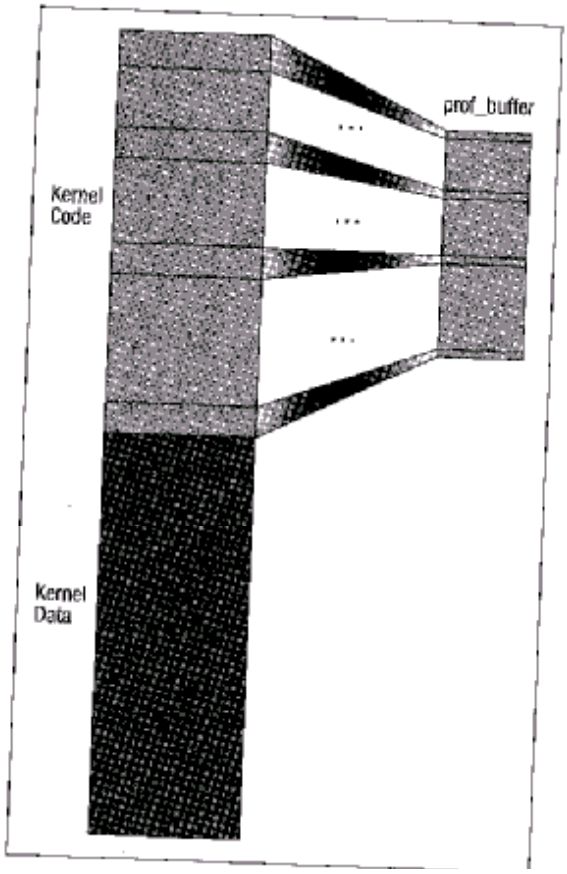

图 4.1 描述用缓存（profiling buffer）

这个 **if** 程序块为记录表格分配内存，并把所有项都清零。注意到如果 **prof_shift** 是 0（缺省值），那么记录功能就被关掉了，**if** 程序段不再被执行，也不为表格分配空间。

19846：内核通过调用 **sti**（13104 行是 UP 版本的——注意该主题在第 6 章中有更详细的介绍）开始接收硬件中断。首先需要激活定时器中断，以便后来对 **calibrate_delay**（19654 行）的调用可以计算机器的 BogoMIPS 的值（在下一节“ BogoMIPS ”中介绍）。因为一些设备驱动程序需要 BogoMIPS 的值，所以内核必需在大部分硬件、文件系统等等初始化之前计算出这个值来。

19876：测试该 CPU 的各种缺陷，比如 Pentium F00F 缺陷（请参看第 8 章），记录检测到的缺陷，以便于内核的其它部分以后可以使用它们的工作。（为了节省空间起见，我们省略掉了 **check_bugs** 函数。）

19882：调用 **smp_init**（19787 行），它又调用了其它的函数来激活 SMP 系统中其它 CPU：在 x86 的平台上，**smp_boot_cpus**（4614 行）初始化一些内核数据结构，这些数据结构跟踪检测另外的 CPU 并简单的将其改为保持模式；最后 **smp_commence**（4195 行）使这些 CPU 继续执行。

19883：把 init 函数作为内核线程终止，这比较复杂；请参看本章后面有关 **init** 的讨论。

19885：增加 idle 进程的 **need_resched** 标志，这样做的原因在此时可能还比较模糊。直到读完了第 5、6、7 章以后，才能有个清楚的概念；但是，在下一个定时器中断结束之前（在第 6 章中讨论），**system_call**（171 行，在第 5 章中讨论）函数中会注意到 idle 进程的 **need_fesched** 标志增加了，并且调用 **schedule**（26686 行，第 7 章）释放 CPU，并将其赋给更应该获取 CPU 的进程。

19886：已经完成了内核初始化的工作——或者不管怎样，已经把需要完成的少量责任传递给了 **init**——所剩余的工作不过是进入 idle 循环以消耗空闲的 CPU 时间片。因此，本行调用 **cpu_idle**（2014 行）——idle 循环。正如你可以从 **cpu_idle** 本身可以发现的一样，该函数从不返回。然而，当有实际工作要处理时，该函数就会被抢占。

注意到 **cpu_idle** 只是反复调用 **idle** 系统调用（下一章将讨论系统调用），它通过 **sys_idle**（2064 行）实现真正的 idle 循环——2014 行对应 UP 版本，2044 行针对 SMP 版本。它们通过执行 **hlt**（对应“ halt ”）指令把 CPU 转入低功耗的“ 睡眠 ”状态。只要没有实际的工作处理，CPU 都将转入这种状态。

BogoMIPS

BogoMIPS 的数字由内核计算并在系统初始化的时候打印。它近似的给出了每秒钟 CPU 可以执行一个短延迟循环的次数。在内核中，这个结果主要用于需要等待非常短周期的设备驱动程序——例如，等待几微秒并查看设备的某些信息是否已经可用。

由于没有正确理解 BogoMIPS 的含义，BogoMIPS 在各处都被滥用，就仿佛它可以满足人类最原始、最深层次的需求：把所有计算机性能的信息简化为一个数字。“ BogoMIPS ”中的“ Bogo ”部分来源于“ 伪（bogus）”，就正是为了防止这种用法：虽然这个数字比大多数性能比较有效很多，但是它仍然是不准确的、容易引起误解的、无用的和不真实的，根本不适合将它用于机器间差别的对比。但是这个数字仍然非常吸引人，这也正是我们在这里讨论这个问题的原因。（顺便说一下，BogoMIPS 中“ MIPS ”部分是“ millions of instructions per second（百万条指令每秒）”的缩写，这是计算机性能对比中的一个常用单位。）

calibrate_delay

19654：**calibrate_delay** 是近似计算 BogoMIPS 数字的内核函数。

19622：作为第一次估算，**calibrate_delay** 计算出在每一秒内执行多少次 **__delay** 循环（6866 行），也就是每个定时器滴答（timer tick）——百分之一秒——内延时循环可以执行多少次。

19664：计算一个定时器滴答内可以执行多少次循环需要在滴答开始时就开始计数，或者应该尽可能与它接近。全局变量 **jiffies**（16588 行）中存储了从内核开始保持跟踪时间开始到现在已经经过的定时器滴答数；第 6 章中将介绍它的实现方式。**jiffies** 保持异步更新，在一个中断内——每秒一百次，内核暂时挂起正在处理的内容，更新变量，然后继续刚才的工作。如果不这样处理，下一行的循环就永远不可能退出。从而，如果 **jiffies** 不声明为 **volatile**——简单的说，这个值变化的原因对于编译器是透明的——gcc 仍然可能对该循环进行优化，并引起该循环进入不能退出的状态。虽然目前的 gcc 还没有如此高的智能，然而它的维护者应该完全能够为它实现这种智能。

19669：定时器又前移了一个滴答，因此又产生一个新的滴答。下一步是要等待 **loops_per_sec** 延时循环调用定时器循环，接着检测是否最少有一个完整的滴答已经完成。如果是这样，就退出首次近似估算循环；如果没有，就把 **loops_per_sec** 的值加倍，然后重新启动这个过程。

这个循环的正确性依赖于如下的事实：现有的机器在任何地方都不能每秒执行 2^{32} 次延时循环——对于 64 位机来说则远低于每秒 2^{64} 次——虽然这只是一个微不足道的问题。

19677：现在内核已经清楚 **loops_per_sec** 循环调用延时循环在这台机器上要花费超过百分之

一秒的时间才能完成，因此，内核将重新开始进行估算。为了提高效率，内核使用折半查找算法计算 **loops_per_sec** 的实际值，我们假定开始的时候，实际值在现在计算结果和其一半之间——实际值不可能比现在计算值还大，但是可以（而且可能）稍微小一点。

19681：和前面使用的方式一样，**calibrate_delay** 查看是否这个 **loops_per_sec** 已经减小了的值还是比较大，需要耗费一个完整的定时器间隔。如果还是相当大，实际值应该小于当前计算值或者就是当前值，因此，使用更小的值继续查询；如果不够大，就使用一个更大的值继续查询。

19691：内核有一种很好的方法来计算一个定时器滴答中执行延时循环的次数。这个数字乘以一秒内滴答的数量就得到了每秒内可以执行的延时循环的次数。这种计算只是一种估算，乘法也累积了误差，因此结果并不能精确到纳秒。但是这个数字供内核使用已经足够精确了。

19693：为了让用户感到激动，内核打印出这个数字。注意这里明显省略了%f 的格式限定——内核尽量避免浮点数运算。这个计算过程中最有用的常量是 500,000；它是用一百万除以 2 得来，理由是每秒钟执行一百万条指令，而每个 delay 循环的核心是 2 条指令（**decl** 和一条跳转指令）。

分析内核选项

parse_options 函数分析由内核引导程序发送给内核的启动选项，在初始化过程中按照某些选项运行，并将剩余部分传送给 init 进程（在本章后面部分提到）。这些选项可能已经存储在配置文件中了，也可能是由用户在系统启动时敲入的——内核并不关心这些。类似的细节全部是内核引导程序应该关注的内容。

parse_options

19707：参数已经收集在一条长的命令行中，内核被赋给指向该命令行头部的一个指针；内核引导程序在前面已经将该行存储在一个指定地址中。

19718：中断下一个参数，保持指向下一个参数的指针以供下一次循环使用。注意系统使用空格而不是通常的空白来分隔内核参数；制表符并不能把当前参数和下一个参数分隔开。如果发现了分隔字符空格，下一行就使用字节 0 覆盖，这样 **line** 可以作为包含有唯一一个内核选项的标准 C 字符串来使用了。如果没有发现空格，就该函数关心的内容而言，其余的部分都具有相同的属性——这只有在处理 **line** 中最后一个选项的情况下才会发生，循环就会在下次开始时结束。

注意该代码不会跳过多个空格。假设 **line** 值如下所述（两个空格）：

rw debug

这会被当作三个选项：“ rw ”，“ ”（空字符串）和 “ debug ”。因为空字符串不是有效的内核选项，它将会被传递到初始化的过程（这一点随后就可以看到）——这当然不是用户所希望的。因此，内核引导程序应该负责对多个空格进行压缩。LILO 通过忽略用户多敲的空格，完美的解决了这个问题。

19721：现在开始解释这些选项。最前面的两个选项——**ro** 和 **rw**——指明内核要装载根文件系统，也就是根目录（ / 目录）所在的位置，而分别处于只读和读/写模式。

19729：第三种可能性，**debug**，增加了调试信息的数量；这些调试信息要通过调用 **do_syslog** 打印出来（25724 行）。

19733：开始几个选项是简单的独立标志，它们并不使用参数。内核也可以辨认形为

option=value 的选项。本行就是一个例子，这里内核引导程序定义了一个命令来代替 init 运行；它使用 **init=*/some/other/program*** 的形式。这里的代码舍弃了 **init=** 部分，为随后 **init** 的使用而把剩余部分在 **execute_command** 中保存起来（20044 行，后面会讨论到）。和其它大部分参数的处理方法不同，本处功能不能在 **checksetup**（19612 行，马上就讨论到）中实现，这是因为它改变了该函数的局部变量。很快，你就可以看到前面三个选项之所以也在这里处理而不是在 **checksetup** 中处理的原因。

19745：大部分内核选项都是由 **checksetup** 函数分析的。如果 **checksetup** 处理了某个选项，就返回真值，循环继续进行。

19750：否则，**line** 中没有已经被辨认的内核选项。在这种情况下，它被作为一个供 init 进程使用的选项或者环境变量来处理——如果其形式为 **envar=*value***，就作为环境变量处理；否则，就作为选项处理。如果 **argv_init** 和 **envp_init**（分别见 19057 和 19059 行）数组中有足够的空间，选项和环境变量就存储在里面供以后 **init** 函数使用。

这解释了从 19736 行开始的注释。字符串 **auto** 并不是任何内核选项的前缀，因此它应该被作为 init 的一个参数存储在 **argv_init** 数组中——这在大多数情况下都是可行的，因为 **auto** 是 init 可以识别的选项。但是，当使用 **init=**的形式给出内核选项时，通常是执行 shell 而不是 init，**auto** 会使 shell 混淆；因此，安全一点的方法是，**parse_options** 在此处忽略所有与此有关的 init 参数。

奇怪的是，当 **argv_init** 或者 **envp_init** 空间用完时，整个循环就结束了。仅仅因为 **argv_init** 的空间用完了并不意味着 line 中就不再含有 init 使用的环境变量，反之亦然。此外，可能还剩下许多内核选项没有处理。当你考虑到 **MAX_INIT_ARGX**（19029 行）和 **MAX_INIT_ENVS**（19030 行）都通过使用**#define** 被预定义为 8——这是一个很容易超过的下限——这种行为就更奇怪了。如果在 19752 行和 19756 行的 **break** 改成 **continue**，那么循环可以继续处理内核选项，而不会写入超过 **argv_init** 和 **envp_init** 数组界限的空间。如果 **command_line** 中仍然包含有并不是为 init 而定义的内核选项，那么这一点就是非常重要的。

19760：所有的内核选项都处理完成了。最后一步是要使用 **NULL** 填充 **argv_init** 和 **envp_init** 数组的末尾，从而使得 init 可以知道在哪里终止。

checksetup

19612：**checksetup** 函数负责进行大部分内核选项的处理过程。它把这些内核选项分为三类：一类使用内核普通参数来分析=sign 之后的部分；另一类自行分析=sign 之后的部分；还有一类自行分析整个行，包括= sign 前面的部分和= sign 后面的部分。第一类被认为是使用“现成”的参数，这与为第二类提供的“原始”参数相对应。最后一类只由一个 IDE 驱动程序组成；内核首先在 19619 行检查并处理这种情况，以使其不会在随后的处理中造成麻烦。

19625：接下来，**checksetup** 扫描整个 **raw_params** 数组（19552 行）并判断是否该内核选项应该不加处理的保留。**raw_params** 中的元素是 **struct kernel_param** 类型（19223 行）的，它把内核选项前缀和装载选项时调用的函数联系起来。如果数组中的某些项的 **str** 成员以 **line** 为前缀，就会调用 **line** 后面的相应函数（也就是前缀之后的部分），随后 **checksetup** 会返回一个非零值以表明它已经对该内核选项进行了处理。**raw_params** 数组以两个 **NULL** 结束，因此在检测到 **str** 成员是 **NULL** 时，循环就可以结束了。在这种情况下，显然循环已经到达了 **raw_params** 数组的结尾，但是仍然没有找到匹配的情况。当然，测试 **setup_func** 成员也可以取得同样好的效果。

这个循环说明了一点：与大多数内核非常不同的是，这里的初始化并不需要尽可能

的快。如果内核比从前多用几微秒来启动，这并没有什么实际的损失——毕竟用户应用程序还没有开始运行，所以他们并没有损失什么东西。

最终结果是代码效率很低，而且存在很多优化的可能。例如，**raw_params** 数组中字符串的长度可以在 **raw_params** 中暂存，而不用在 19626 行多次重复计算。更好的解决方法是，可以把 raw_params 数组中的项按照字符顺序排序，这样 **checksetup** 就可以进行折半查找。

在 **raw_params** 的情况中实现排序并没有什么障碍，但是这样也可能并不能获得很大的优势，因为折半查找的优点只有在比较大的数组中才能充分表现出来（所谓比较大的确切值在不同的环境中也有所不同）。**raw_params** 的姊妹数组 **cooked_params**（19228 行）当然是足够大的，可以显示出折半查找的优势；但是这样就引发了一个新的问题：对 **cooked_params** 进行排序比较难用，因为这可能需要分隔一些**#ifdef** 程序段——请参看从 19268 行到 19272 行的例子。进一步说，因为算法只是查找前缀，而不使用完全匹配，在遍历数组中的各个项时对遍历次序比较敏感，所以这种特性在使用不同的查找次序时就很难再保持了。然而，这些问题并不是不可克服的（程序员可以预先静态地为引导程序建立一颗前缀树），如果性能在这里是主要因素，那么这种努力也是值得的。但是，由于性能在这里并不是主要问题，所以简单性才被作为最重要的因素体现出来。

即使这样，在类似的 **root_dev_names** 数组（19085 行）中——这个数组把硬件设备名的前缀映射到它们的主 ID 号上——开发者仍然可以简单地通过把比较常用的项（IDE 和 SCSI 磁盘）放在不太常用的项（串口 IDE CDs）的前面以节省出一点性能。但是我在 **raw_params** 或 **cooked_params** 中并没有发现与之类似的模式。

另外一件需要注意的事是：现在你可以猜想一下为什么 **ro**，**rw** 和 **debug** 选项在 **parse_options** 中测试而不在这里测试——**parse_options** 要检测精确的匹配，但是 **checksetup** 只检测前缀。作为一个特殊的情况，**ro** 选项碰巧正好是 **root=**（19553 行）的前缀，这样如果这三个选项彼此合并，就需要仔细处理了。这似乎仍然是一个相当无力的原因。考虑一下 **noinitrd** 选项（19251 行）。这是 **cooked_params** 的一个项，因而只需要匹配前缀，而且与之相关联的设置函数（**no_initrd**，19902 行）将忽略所有可能已经传递给它们的参数——这正像 **ro**，**rw** 和 **debug** 被包含在 **cooked_params** 中时所可能进行的工作一样。

19632：这个循环为 **cooked_params** 数组的处理工作和前面一个循环为 **raw_params** 数组的处理工作相同。这两个循环（当然不包括循环使用的数组）间的唯一区别是本循环在调用设置函数之前，使用 **get_options**（19062 行）处理 **line** 中=sign 后面的部分。简单的说，**get_options** 使用 10 个负整数填充 **ints[1]**到 **ints[10]**。**ints[0]**中是 **ints** 中使用元素的个数——也就是，它记录了存储在 **ints** 中的 **intsget_options** 数量。接着这个数组将被传递给设置函数，该设置函数则会按照自己喜欢的方式对该数组内容进行解释。

19640：返回 0，说明 **line** 中所包含的内核选项不能被函数理解。

profile_setup

19076：**profile_setup** 是 **checksetup** 调用的设置函数的一个完美的例子：这个函数十分短小，使用 **ints** 参数处理了部分内容。而且到目前为止你也应该对它的目的有了一定了解。正如前面提到的一样，用户可以在启动的时候设置 **prof_shift** 的值——好，这里正是它的实现方式。当内核启动过程提供 **profile=**选项时，就调用 **profile_setup** 函数。前缀字符串和函数在 19235 行被联系在一起。注意这是在 **cooked_params** 中，因此 **profile_setup** 取得的是处理过的参数。

19079：如果参数中存在 **profile**=的形式，就使用 **profile**=后面的第一个数字作为 **prof_shift** 的新值。选项给出的其它参数都被简单的忽略了。

19081：如果给出了 **profile**=选项，但是没有为它提供参数，**prof_shif**t 的缺省值就是 2。这个缺省值有些奇怪，因为我们已经知道，这意味着使用四分之一的内核可用内存来配置其余部分——这是一个很大的开销。但是另一方面，使用这些内存有助于更精确的定位问题热点——只有很少的几条指令存在不确定性，这样应该比较容易地把问题限制在一两行源程序代码内。那张图也并不是像我所画的那样简单：因为图中只描述了内核代码，这种开销还不到内核所有内存空间的 25%，但是对于所覆盖的代码量来说却并不止 25%。

init

init 从许多方面看都是一个非常特殊的进程。这是内核运行的第一个用户进程，它要负责触发其它必需的进程以使系统作为一个整体进入可用的状态。这些工作由/etc/inittab 文件控制，通常包括设置 getty 进程以接受用户登录；建立网络服务，例如 FTP 和 HTTP 守护进程；等等。如果没有这些进程，用户就不可能完成多少工作，这样成功启动内核就显得没有多大意义了。

这种设计的另外一个重要的副作用是 init 是系统中所有进程的祖先。init 产生 getty 进程，getty 进程产生 login 进程，login 进程产生你自己的 shell，使用自己的 shell，可以产生每一个你运行的进程。在所有的结果中，这有助于确保内核进程表中的所有项最终都能够得到处理。进程结束以后将其清除（回收）的工作首先应由其父进程完成；如果父进程已经退出，那么祖父进程就要担负起这种责任；如果祖父进程已经退出，那么曾祖父进程就要担负起这种责任，周而复始。通过这种方式，从不退出的 init 进程就可能要负责回收其它进程。

因此，为了确保这些重要的工作都能正确执行，内核初始化进程所需要做的最后一步工作就是创建 init 进程，接下来就加以描述。

init

20044：**unused** 参数来源于该函数的非常规调用。**init** 函数——不要和 *init 进程*搞混了，后者是它随后要创建的——作为内核线程开始生命周期，一个作为内核的一部分运行的进程。（如果你编写过多线程的程序，这里的内核线程可能会同你所已经知道的线程意义有所不同——在那种意义下，它不是一个内核线程。）实际上，**init** 函数就像是新进程使用的剥离出来了的 **main** 函数，**unused** 参数是一个独立的指针，其值指向为给定进程所提供的信息——这比通常使用 **argc**，**argv** 和 **envp** 参数传递的信息要少得多。**init** 函数碰巧不需要额外的信息，因此这个参数命名为 **unused**，就是要强调这一点。

为了确保在这一点上你不会产生困惑，我们在这里再对整个机制进行扼要重复：**init** 函数是内核的一部分；它在内核中作为内核的一个独立的执行部分运行；也就是说，无论从哪个方面看它都是内核代码。但是，init 进程就不是这样了。在某些方面，init 进程是一个特殊的进程，但是不属于内核本身；其代码存储在磁盘上单独的可执行映像中，这和其它程序一样。因为 **init** 函数后来产生 init 进程，而它自己又恰好作为进程运行，这样就很容易产生混淆。

因为 idle 进程已经占据了进程 ID 号（PID）0，**init**（当然是 init）就被赋值为下一个可用的 PID，也就是 1。（进程 ID 在第 7 章中讨论。）内核重复假定 PID 为 1 的进程是 init，因此这种特性在没有充分地相互作用，也就是没有同步地进行修改的情况

下是不能改变的。

20046：调用 **lock_kernel**（17492 行对应 UP 版本；10174 行对应 SMP 版本）执行后续几行，而不会受到其它会受到随后工作的影响的内核模块的干扰。内核锁随后在 20053 行被释放。

20047：调用 **do_basic_setup**（19916 行）初始化总线并随同其它工作产生一些其它内核线程。

20052：内核已完全完成初始化了，因此 **free_initmen**（7620 行）可以舍弃内核的.text.init 节的函数和.data.init 节的数据。所有使用**__initfunc** 标记过的函数和使用**__initdata** 标记过的数据现在都不能使用了，它们曾经获得的内存现在也可能重新用于其它目的了。

20055：如果可能，打开控制台设备，这样 init 进程就拥有一个控制台，可以向其中写入信息，也可以从其中读取输入信息。实际上 init 进程除了打印错误信息以外，并不使用控制台，但是如果调用的是 shell 或者其它需要交互的进程，而不是 init，那么就需要一个可以交互的输入源。如果成功执行 **open**，/dev/console 就成为 init 的标准输入源（文件描述符 0）。

20059：调用 **dup** 打开/dev/console 文件描述符两次，这样，init 就也使用它供标准输出和标准错误使用（文件描述符 1 和 2）。假设 20055 行的 **open** 成功执行（正常情况），init 现在就有三个文件描述符——标准输入、标准输出以及标准错误——全都加载在系统控制台之上。

20067：如果内核命令行中给出了到 init 的直接路径（或者别的可替代的程序），现在就试图执行 **init**。

因为当 **execve** 成功执行目标程序时并不返回，只有当前面的所有处理过程都失败时，才能执行相关的表达式。接下来的几行在几个地方查找 init，按照可能性由高到低的顺序依次是：首先是/sbin/init，这是 init 标准的位置；接下来是两个可能的位置，/etc/init 和/bin/init。

20072：这些是 init 可能出现的所有地方。如果现在还没有出现，**init** 就无法找到它的这个同名者了，机器可能就崩溃了。因此，它就会试图建立一个交互的 shell（/bin/sh）来代替。现在 **init** 最后的希望就是 root 用户可以修复这种错误并重新启动机器。（可以肯定，root 也正是希望如此。）

20073：**init** 甚至不能创建 shell——一定是发生了什么问题！好，按照它们所说的，当所有其它情况都失败时，调用 **panic**（25563 行）。这样内核就会试图同步磁盘，确保其状态一致，然后暂停进程的执行。如果超过了内核选项中定义的时间，它也可能会重新启动机器。