列可以在主内核代码、中断处理函数或其他异步处理请求代码中进行并发操作，自旋锁是非常必要的。

事实上，内核代码在很多地方都含有同步变量。早期的Linux内核只有一个大内核锁（Big Kernel Lock，BKL）。由于它阻止了不同的处理器并发运行内核代码，这使得内核的效率非常低下，特别是在多处理器平台上。所以，很多新的同步点被更加细粒度地引入了。

10.3.5 启动Linux系统

每个平台的细节都有不同，但是整体来说，下面的步骤代表了启动的过程。当计算机启动时，BIOS加电自检（POST），并对硬件进行检测和初始化，这是因为操作系统的启动过程可能会依赖于磁盘访问、屏幕、键盘等。接下来，启动磁盘的第一个扇区，即主引导记录（MBR），被读入到一个固定的内存区域并且执行。这个分区中含有一个很小的程序（只有512字节），这个程序从启动设备中，通常是IDE或SCSI磁盘，调入一个名为boot的独立程序。boot程序将自身复制到高地址的内存当中从而为操作系统释放低地址的内存。

复制完成后，boot程序读取启动设备的根目录。为了达到这个目的，boot程序必须能够理解文件系统和目录格式，这个工作通常由引导程序，如GRUB（多系统启动管理器），来完成。其他流行的引导程序，如Intel的LILO，不依赖于任何特定的文件系统。相反，他们需要一个块映射图和低层地址，他们描述了物理扇区、磁头和磁道，可以帮助找到相应的需要被加载的扇区。

然后，boot程序读入操作系统内核，并把控制交给内核。从这里开始，boot程序完成了它的任务，系统内核开始运行。

内核的开始代码是用汇编语言写成的，具有较高的机器依赖性。主要的工作包括创建内核堆栈、识别CPU类型、计算可用内存、禁用中断、启用内存管理单元，最后调用C语言写成的main函数开始执行操作系统的主要部分。

C语言代码也有相当多的初始化工作要做，但是这些工作更逻辑化（而不是物理化）。C语言代码开始的时候会分配一个消息缓冲区来帮助调试启动出现的问题。随着初始化工作的进行，信息被写入消息缓冲区，这些信息与当前正在发生的事件相关，所以，如果出现启动失败的情况，这些信息可以通过一个特殊的诊断程序调出来。我们可以把它当作是操作系统的“飞行信息记录器”（即空难发生后，侦查员寻找的黑盒子）。

接下来，内核数据结构得到分配。大部分内核数据结构的大小是固定的，但是一少部分，如页面缓存和特殊的页表结构，依赖于可用内存的大小。

从这里开始，系统进行自动配置。使用描述何种设备可能存在配置文件，系统开始探测哪些设备是确实存在的。如果一个被探测的设备给出了响应，这个设备就会被加入到已连接设备表中。如果它没有响应，就假设它未连接或直接忽略掉它。不同于传统的UNIX版本，Linux系统的设备驱动程序不需要静态链接，它们可以被动态加载（就像所有的MS-DOS和Windows版本一样）。

关于支持和反对动态加载驱动程序的争论非常有趣，值得简要地阐述一下。动态加载的主要论点是同样的二进制文件可以分发给具有不同系统配置的用户，这个二进制文件可以自动加载它所需要的驱动程序，甚至可以通过网络加载。反对动态加载的主要论点是安全。如果你正在一个安全的环境中运行计算机，比如说银行的数据库系统或者公司的网络服务器，你肯定不希望其他人向内核中插入随机代码。系统管理员可以在一个安全的机器上保存系统的源文件和目标文件，在这台机器上完成系统的编译链接，然后通过局域网把内核的二进制文件分发给其他的机器。如果驱动程序不能被动态加载，这就阻止了那些知道超级用户密码的计算机使用者或其他人向系统内核注入恶意或漏洞代码。而且，在大的站点中，系统编译链接的时候硬件配置都是已知的。需要重新链接系统的变化非常罕见，即使是在系统中添加一个硬件设备也不是问题。

一旦所有的硬件都配置好了，接下来要做的事情就是细心地手动运行进程0，建立它的堆栈，运行它。进程0继续进行初始化，做如下的工作：配置实时时钟，挂载根文件系统，创建init进程（进程1）和页面守护进程（进程2）。

init进程检测它的标志以确定它应该为单用户还是多用户服务。前一种情况，它调用fork函数创建一个shell进程，并且等待这个进程结束。后一种情况，它调用fork函数创建一个运行系统初始化shell脚本