# Linux系统：整体

## 1 综合

### 1.1 ++

——

**linux下文件的目录**

/bin 存放系统可执行程序（大部分系统命令）
/sbin 存放 root 用户的系统可执行程序
/boot 存放内核 和启动程序的相关文件
/lib 库目录，存放系统最基本的动态库
/media 默认挂载设备媒体的目录，例如 U 盘、光驱
/mut 推荐挂载设备媒体的目录
/usr 用于存放庞大而复杂的目录(unix system resource，用于安装软件的目录)
/proc 系统内存的映射（随着系统的运行，时长变化的）
/ect 系统软件的启动和配置目录
/dev 用于存放设备文件 /home 家目录，所用用户的根目录（当前用户的根目录是/home/user）

——

**linux下文件的类型**

b 块设备文件 c 字符设备文件 d 目录文件 –普通文件 l (软)连接文件
p 管道文件 s 本地套接字

——

**linux下的权限分为哪3个组**

用户权限、同组用户权限、其他用户权限

——

**操作系统和内核简介**

操作系统是指整个系统负责完成最基本功能和系统管理的那些部分。这些部分包括内核，设备驱动程序，启动引导程序，命令行Shell或者其他种类用户界面，基本的文件管理工具和系统工具。系统这个词其实包含了操作系统和所有运行在它上面的应用程序。
用户界面是操作系统的外在表象，内核才是操作系统的内在核心。系统的其他部分必须依靠内核的部分软件提供服务，像管理硬件设备，分配系统资源等。内核有时候被称作是管理者或者是操作系统核心。通常一个内核由负责相应中断的中断服务程序，负责管理多个进程从而分享处理器时间的调度程序，负责管理进程地址空间的内存管理程序和网络，进程间通信等系统服务程序共同组成。对于提供保护机制的现代系统来说，内核独立与普通应用程序，它一般处于系统态，拥有受保护的内存空间和访问设备的权限。这种系统和被保护起来的内存空间，系统成为内核空间。相对的，应用程序在用户空间执行。它只能看到允许它们使用的部分系统资源，并且只是要某系些特定的系统功能，不能直接访问硬件，也不能访问内核划给别人的内存范围，还有其他一些使用限制。当内核运行的时候，系统以内核态，进入内核空间执行。而执行一个普通用户程序时，系统将以用户态进入用户空间执行。

在系统中运行的应用程序通过系统调用来与内核通信（见图1–1）。应用程序通常调用库函（比如C库函数）再由库函数通过系统调用界面，让内核代其完成各种不同任务。一些函数库调用提供了系统调用不具备的许多功能，在那些较为复杂的函数中，调用内核的操作通常只是整体工作的一个步骤而已。举个例子，拿printf函数来说，它提供了数据的缓存和格式化等操作，调用write函数将数据写到控制台上只不过是其中的一个动作罢了。不过，也有些库函数系统调用就是一一对应的关系，比如，open库函数除了调用open系统调用之外，几乎什么不做。还有一些C库函数，像strcpy,根本就不需要直接调用系统级的操作。当一个应用程执行一条系统调用，我们说内核正在代其执行。如果进一步解释，在这种情况下，应用程序被为通过系统调用在内空间运行，而内核被称为运行于进程上下文中，这种交互关系一应用序通过系统调用界面陷入内核一是应用程序完成其工作的基本行为方式。

内核还要负责管理系统的硬件设备。现有的几乎所有的体系结构，包括全部Linux支持的体结构，都提供了中断机制。当硬件设备想和系统通信的时候，它首先要发出一个异步的中断号去打断处理器的执行，继而打断内核的执行。中断通常对应着一个中断号，内核通过这个断号查找相应的中断服务程序并调用这个程序响应和处理中断。举个例子，当你敲击键盘时候，键盘控制器发送一个中断信号告知系统，键盘缓冲区有数据到来。内核注意到这个中对应的中断号，调用相应的中断服务程序。该服务程序处理键盘数据然后通知键盘控制器可继续输入数据了。为了保证同步，内核可以停用中止一既可以停止所有的中断也可以有选地停止某个中断号对应的中断。许多操作系统的中断服务程序，包括Linux的，都不在进程下文中执行。它们在一个与所有进程都无关的、专门的中断上下文中运行。之所以存在这样专门的执行环境，就是为了保证中断服务程序能够在第一时间响应和处理中断请求，然后地退出。这些上下文代表着内核活动的范围。实际上我们可以将每个处理器在任何指定时间点上的活动必然概括为下列三者之一：

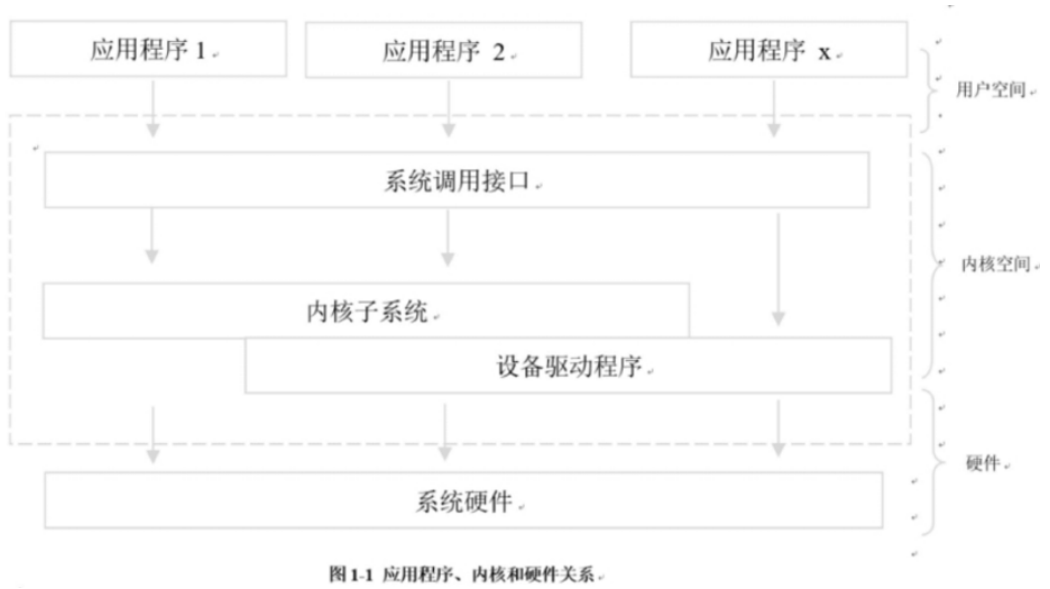


图 1-1 应用程序、内核和硬件关系

- 运行于用户空间，执行用户进程
- 运行与内核空间，处于进程上下文，代表某个特定的进程执行
- 运行与内核空间，处于中断上下文，与任何进程无关，处理某个特定的中断

以上所列几乎包括所有情况，即使边边角角的情况也不例外，例如CPU空闲时，内核就运行一个空进程，处于进程上下文，但运行于内核空间。

--

**Linux内核版本**

Lux内核有两种：稳定的和处于开发中的。稳定的内核具有工业级的强度，可以广泛地应用和部署。新推出的稳定内核大部分都只是修正了一些Bug或是加入了一些新的设备驱动程序。另一方面处于开发中的内核中许多东西变化得都很快。而且由于开发者不断试验新的解决方案，内核常常发生剧烈的变化。

Lix通过一个简单的命名机制来区分稳定的和处于开发中的内核（见图1-2）。这种机制使用三个或者四个用“.”分隔的数字来代表不同内核版本。第一个数字是主版本号，第二个数字是从版本号，第三个数字是修订版本号，第四个可选的数字为稳定版本号(stable version)。从副版本号可以反映出该内核是一个稳定版本还是一个处于开发中的版本：该数字如果是偶数，那么此内核就是稳定版；如果是奇数，那么它就是开发版。举例来说，版本号为2630.1的内核，它就是一个稳定版。这个内核的主版本号是2，从版本号是6，修订版本号是30，稳定版本号是1。头两个数字在一起描述了“内核系列”一在这个例子中，就是2.6版内核系列。


处于开发中的内核一般要经历几个阶段。最开始，内核开发者们开始试验新的特性，这时候出现错误和混乱是在所难免的。经过一段时间，系统渐渐成熟，最终会有一个特性审定的声明。这时候，Lius就不再接受新的特性了，而对己有特性所进行的后续工作会继续进行。当Lius认为这个新内核确实是趋于稳定后，就开始审定代码。这以后，就只允许再向其中加入修改bug的代码了。在经过一个短暂（希望如此）的准备期之后，Lius会将这个内核作为一个新的稳定版推出。例如，1.3系列的开发版稳定在2.0，而2.5稳定在2.6。

--

**Linux内核开发者社区**

当你开始开发内核代码时，你就成为全球内核开发社区的一分子了。这个社区最重要的论坛是linux kernel mailing list(常缩写为lkml)。你可以在http:/vegr kernel.org上订阅邮件。要注意的是这个邮件列表流量很大，每天有超过几百条的消息，所以其他的订阅者（包括所有的核心开发人员，甚至包括Lius本人)可没有心思听人说废话。这个邮件列表可以给从事内核开发的人提供价值无穷的帮助，在这里，你可以寻找测试人员，接受评论(peer review),向人求助。

--

**内核源码树**

| 目录 | 描述 |
| --- | --- |
| Crypto | 加密 API |
| Documentation | 内核源码文档 |
| Drivers | 设备驱动程序 |
| Firmware | 使用某些驱动程序而需要的设备固件 |
| Fs | VFS 和各种文件系统 |
| Include | 内核头文件 |
| Init | 内核引导和初始化 |
| Ipc | 进程间的通信代码 |
| Kernel | 核心子系统 |
| Lib | 通用内核函数 |
| Mm | 内存管理子系统和 VM |
| Net | 网络子系统 |
| Samples | 示例，示范代码 |
| Scripts | 编译内核所用的脚本 |
| Security | Linux 安全模块 |
| Sound | 语音子系统 |
| Usr | 早期用户空间代码（所谓的 initramfs） |
| Tools | 在 Linux 开发中用的工具 |
| Virt | 虚拟化基础结构 |

在内核树根目录中很多文件是值得提及。COPYING文件是内核许可证，(GNU GPL VI2)。

## 1.2 内核开发特点

1--

相对于用户空间内应用程序的开发，内核开发有一些独特之处。尽管这些差异并不会使开发内核代码的难度超过开发用户代码，但它们依然有很大不同。 这些特点使内核成了一只性格迥异的猛兽。一些常用的准则被颠覆了，而又必须建立许多全新的准则。尽管有许多差异一目了然（人人都知道内核可以做它想做的任何事），但还是有一些差异晦暗不明。最重要的差异包括以下几种：

- 内核编程时既不能访问C库也不能访问标准的C头文件。
- 内核编程时必须使用GNU C。
- 内核编程时缺乏像用户空间那样的内存保护机制。
- 内核编程时难以执行浮点运算。
- 内核给每个进程只有一个很小的定长堆栈。
- 由于内核支持异步中断、抢占和SMP,因此必须时刻注意同步和并发。
- 要考虑可移植性的重要性。

让我们仔细考察一下这些要点，所有内核开发者必须牢记以上要点。

2--

**无libc库或无标准头文件**

与用户空间的应用程序不同，内核不能链接使用标准C函数库一或者其他的那些库也不行。不过最主要的原因还是速度和大小。对内核来说，完整的C库一哪怕是它的一个子集，都太大且太低效了。别着急，大部分常用的C库函数在内核中都已经得到了实现。比如操作字符串的函数组就位于lib/stg.c文件中。只要包含<linux/string.h>头文件，就可以使用它们。

头文件都指的是组成内核源代码树的内核头文件。内核源代码文件不能包含外部头文件，就像它们不能用外部库一样。基本的头文件位于内核源代码树顶级目录下的include目录中。例如，头文件<linux/inotify.h>对应内核源代码树的include/linux/inotify.h.体系结构相关的头文件集位于内核源代码树的arch//include/asm目录下。例如，如果编译的是x86体系结构，则体系结构相

关的头文件就是arch/x86/include/asm.内核代码通过以asm/为前缀的方式句含这些头文件，例如<asm/ioctl h>.
所有现的的函数中，最名的就数printf函数了。内核代码虽然无法调用printf,但它提供的primk函数几乎与printf相同，printk函数负责把格式化好的字符串拷贝到内核日志缓冲区上，这样，syslog程序就可以通过读取该冲区来获取内核信息。primk的用法像printf:

```
printk("Hello world!A string:'%s' and an integer:'%d'\n", str, i);
```

printk和printf之间的一个显著区别在于，printk允许你通过指定一个标志来设置优先级。syslogd会根据这个优先级标志来决定在什么地方显示这条系统消息。 例子： 定义一个内联函数的时候，需要使用static作为关键字，并且用inline限定它。比如：

```
static inline void wolf(unsigrfed long tail size)
```

内联函数必须在使用之前就定义好，否则编译器就没法把这个函数展开。实践中一般在头文件中定义内联函数。由于使用了static作为关键字进行限制，所以编译时不会为内联函数单独建立一个函数体。如果一个内联函数仅仅在某个源文件中使用，那么也可以把它定义在该文件开始的地方。
在内核中，为了类型安全和易读性，优先使用内联函数而不是复杂的宏。

3--
**内联汇编**
gcc编译器支持在C函数中嵌入汇编指令。当然，在内核编程的时候，只有知道对应的体系 结构，才能使用这个功能。 我们通常使用asm()指令嵌入汇编代码。例如，下面这条内联汇编指令用于执行x86处理器 的rdtsc指令，返回时间戳(tsc)寄存器的值：

```
unsigned int low,high;
asm volatile("rdtsc":"=a"(low),"=d"(high));
/*low和high分别包含64位时间戳的低32位和高32位*/
```

Luix的内核混合使用了C语言和汇编语言。在偏近体系结构的底层或对执行时间要求严格的地方，一般使用的是汇编语言。而内核其他部分的大部分代码是用C语言编写的。
分支声明
对于条件选择语句，gcc内建了一条指令用于优化，在一个条件经常出现，或者该条件很少 出现的时候，编译器可以根据这条指令对条件分支选择进行优化。内核把这条指令封装成了宏， 比如likely()和unlikely(),这样使用起来比较方便。 例如，下面是一个条件选择语句：

```
if (error)
/../
如果想要把这个选择标记成绝少发生的分支：
/我们认为error绝大多数时间都会为0
if (unlikely(error)) {
/../
相反，如果我们想把一个分支标记为通常为真的选择：
我们认为success通常都不会为0
if (likely(success)) {
/.../
```

在你相要对其个冬件选择语句进行优化之前、一定要搞清楚是不是在在这么一个条件。 在绝大多数情况下都会成立。这点十分重要：如果你的判断正确，确实是这条件占压倒性的地位，那么性能会得到提升；如果你搞错了，性能反而会下降。正如上面这些例子所示， 一些错误条件进行判断的时候会用到unlikely()和likely().你可以猜到，unlikely()在内核中会得 到更广泛的使用，因为if语句往往判断一种特殊情况。

4--

**没有内存机制保护**

如果一个用户程序试图进行一次非法的内存访问，内核就会发现这个错误，发送SIGSEGV信号，并结束整个进程。然而，如果是内核自己非法访问了内存，那后果就很难控制了。内核中发生的内存错误会导致oops,这是内核中出现的最常见的一类错误。在内核中，不应该去做访问非法的内存地址， 引用空指针之类的事情，否则它可能会死掉，却根本不告诉你一声一在内核里，风险常常会比外面大一些。 此外，内核中的内存都不分页。也就是说，你每用掉一个字节，物理内存就减少一个字节。所以，在你想往内核里加入什么新功能的时候，要记住这点。变量类型定义以及静态变量的使用合理的使用内存。

5--

**不要轻易在内核中使用浮点数**

在用户空间的进程内进行浮点操作的时候，内核会完成从整数操作到浮点数操作的模式转换。在执行浮点指令时到底会做些什么，因体系结构不同，内核的选择也不同，但是，内核桶常捕获陷阱并着手于整数到浮点方式的转变。

与用户空间进程不同，内核并不能完美地支持浮点操作，因为它本身不形陷入。在内核中使用浮点数时，除了要人工保存和恢复浮点寄存器，还有其也一些琐碎的事情要做。但是现在的CPU core内都植入了浮点运算器来加速浮点运算。

6--

**容积小而固定的栈**

用户空间的程序可以从栈上分配大量的空间来存放变量，甚至巨大的结构体或者是包含数以千计的数据项的数组都没有问题。之所以可以这么做，是因为用户空间的栈本身比较大，而且还能动态地增长（在DOS那个年代，这种低级的操作系统即使在用户空间也只有固定大小的栈）。 内核栈的准确大小随体系结构而变。在x86上，栈的大小在编译时配置，可以是4KB也可以是8KB.从历史上说，内核栈的大小是两页，这就意味着，32位机的内核栈是8KB,而64位机是16KB,这是固定不变的。每个处理器都有自己的栈。

7--

**同步与并发**

内核很容易产生竞争条件。和单线程的用户空间程序不同，内核的许多特性都要求能够并发地访问共享数据，这就要求有同步机制以保证不出现竞争条件，特别是： Liux是抢占多任务操作系统。内核的进程调度程序即兴对进程进行调度和重新调度，内核必须和这些任务同步。 Liux内核支持对称多处理器系统(SMP)。所以，如果没有适当的保护，同时在两个或两个以上处理器上执行的内核代码很可能会同时访问共享的同一资源。

中断时异步到来的，完全不顾及当前正在执行的代码。也就是说，如果不加以适当的保护，中断完全有可能在代码访问资源的时候到来，这样，中断处理程序就有可能访问同一资源。 Linux内核可以抢占。所以，如果不加以适当的保护，内核中一段正在执行的代码可能会被另外一段代码抢占，从而有可能导致几段代码同时访问相同资源。

常用的解决竞争的办法就是自旋锁和信号量。

8--

**可移植性**

尽管用户空间的应用程序不太注意可移植的问题，然而Linux却是个可移植的操作系统，并且要一直保持这种特点。也就是说，大部分的C代码应该与体系结构无关，在许多不同体系结构的计算机上都可以编译执行，因此，必须把与体系结构相关的代码从内核代码树到特定目录中适当的分离出来。 比如保持32位、64为对齐，不假定字长和页面长度等一系列准则都有助于移植性。