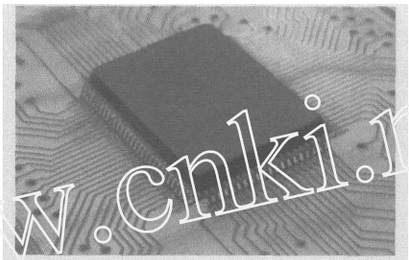
Linux 内核解读入门

□ 喻锋荣

针对好多 Linux 爱好者对内核很有兴趣却无从下手，本文旨在介绍一种解读 Linux 内核源码的入门方法，而不是解说 Linux 复杂的内核机制。



1. 核心源程序的文件组织

（1）Linux 核心源程序通常都安装在/usr/src/Linux 下，而且它有一个非常简单的编号约定：任何偶数的核心（例如 2.0.30）都是一个稳定的发行的核心，而任何奇数的核心（例如 2.1.42）都是一个开发中的核心。

本文基于稳定的 2.2.5 源代码，第二部分的实现平台为 Redhat Linux 6.0。

（2）核心源程序的文件按树形结构进行组织，在源程序树的最上层你会看到这样一些目录：

● Arch：arch 子目录包括了所有和体系结构相关的核心代码。它的每一个子目录都代表一种支持的体系结构，例如 i386 就是关于 intel cpu 及与之相兼容体系结构的子目录。PC 机一般都基于此目录；

● Include：include 子目录包括编译核心所需要的大部分头文件。与平台无关的头文件在 include/Linux 子目录下，与 intel cpu 相关的头文件在 include/asm－i386 子目录下，而 include/scsi 目录则是有关 scsi 设备的头文件目录；

● Init：这个目录包含核心的初始化代码（注：不是系统的引导代码），包含两个文件 main.c 和 Version.c，这是研究核心如何工作的一个非常好的起点；

● Mm：这个目录包括所有独立于 cpu 体系结构的内存管理代码，如页式存储管理内存的分配和释放等，而和体系结构相关的内存管理代码则位于 arch/*/mm/，例如 arch/i386/mm/Fault.c；

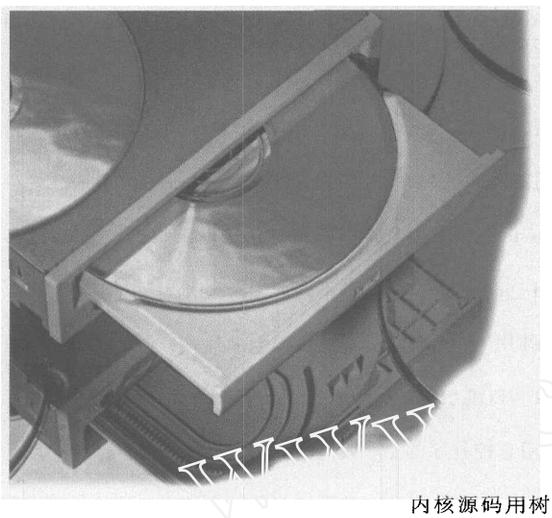
● Kernel：主要的核心代码，此目录下的文件实现了大多数 Linux 系统的内核函数，其中最重要的文件当属 sched.c，同样，和体系结构相关的代码在 arch/*/kernel 中；

● Drivers：放置系统所有的设备驱动程序；每种驱动程序又各占用一个子目录，如/block 下为块设备驱动

管理锦囊

5. 资金到位要有时间概念　如果有两家创投，一家的钱马上就可以到位，另外一家的钱还可望而不可及，你该怎么办？抓住现在就能用的钱。创业，讲到底，靠的是资金的扶持，实实在在的几百万，好过虚无飘渺的数千万。

6. 先找人，再找钱　有的创业者认为最要紧的是投资，只要有了钱，怎么都好办，其实不然，创业的过程有一个顺序先后的问题，不是找到钱之后再去找人，而应该是先找人：CEO、CTO、CFO，建立管理团队，然后再找资金。人，是最难的一环。一旦资金到位，整个公司就快速地运作起来，时机不等人。

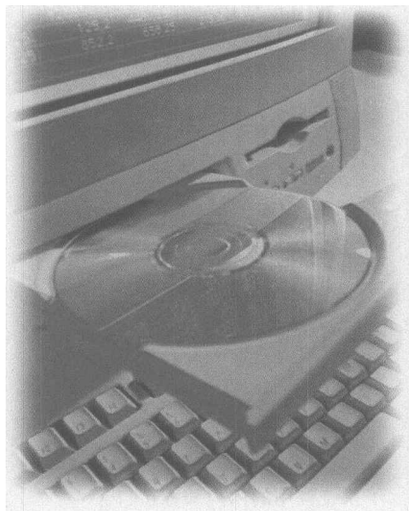


文件，它是对该目录下的文件的一些说明，同样有利于我们对内核源码的理解。

2. 解读实战：为你的内核增加一个系统调用

虽然 Linux 的内核源码用树形结构组织得非常合理、科学，把功能相关联的文件都放在同一个子目录下，这样使得程序更具可读性。然而，Linux 的内核源码实在是太大而且非常复杂，即便采用了很合理的文件组织方法，在不同目录下的文件之间还是有很多的关联，分析核心的一部分代码通常要查看其他的几个相关的文件，而且可能这些文件还不在同一个子目录下。

体系的庞大复杂和文件之间关联的错综复杂，可能就是很多人对其望而生畏的主要原因。当然，这种令人生畏的劳动所带来的回报也是非常令人着迷的：你不仅可以从中学到很多的计算机的底层的知识（如下面将讲到的系统的引导），体会到整个操作系统体系结构的精妙和在解决某个具体细节问题时算法的巧妙，而且更重要的是在源码的分析过程中，你就会被一点一点地、潜移默化地专业化；甚至，只要分析 1/10 的代码后，你就会深刻地体会到，什么样的代码才是一个专业的程序员写的，什么样的代码是一个业余爱好者写的。为了使读者能更好的体会到这一特点，下面举了一个具体的内核分析实例，希望能通过这个实例，使读者对 Linux 的内核的组织有些具体的认识，读者从中也可以学到一些对内核的分析方法。



以下即为分析实例：

(1)操作平台

硬件：cpu intel Pentium II；

软件：Redhat Linux 6.0；内核版本 2.2.5

(2)相关内核源代码分析

① 系统的引导和初始化：Linux 系统的引导有好几种方式，常见的有 Lilo、Loadin 引导和 Linux 的自举引导(bootsect - loader)，而后者所对应源程序为 arch/i386/boot/bootsect.S，它为实模式的汇编程序，限于篇幅在

程序，比如 ide(ide.c)。如果你希望查看所有可能包含文件系统的设备是如何初始化的，你可以看 drivers/block/genhd.c 中的 device_setup()。它不仅初始化硬盘，也初始化网络，因为安装 nfs 文件系统的时候需要网络。

其他如 Lib 放置核心的库代码；Net，核心与网络相关的代码；Ipc，这个目录包含核心的进程间通信的代码；Fs，所有的文件系统代码和各种类型的文件操作代码，它的每一个子目录支持一个文件系统，例如 fat 和 ext2；Scripts，此目录包含用于配置核心的脚本文件等。

一般在每个目录下都有一个 .depend 文件和一个 Makefile 文件，这两个文件都是编译时使用的辅助文件，仔细阅读这两个文件对弄清各个文件之间的联系和依托关系很有帮助，而且在有的目录下还有 Readme

此不做分析。无论是哪种引导方式，最后都要跳转到 arch/i386/Kernel/setup. S。 setup. S 主要是进行时模式下的初始化，为系统进入保护模式做准备。此后，系统执行 arch/i386/kernel/head. S（对经压缩后存放的内核要先执行 arch/i386/boot/compressed/head. S)；head. S 中定义的一段汇编程序 setup_idt，它负责建立一张 256 项的 idt 表(Interrupt Descriptor Table)，此表保存着所有自陷和中断的入口地址，其中包括系统调用总控程序 system_call 的入口地址。当然，除此之外，head. S 还要做一些其他的初始化工作。

② 系统初始化后运行的第一个内核程序 asmlinkage void __init start_kernel(void) 定义在 /usr/src/linux/init/main. c 中，它通过调用 usr/src/linux/arch/i386/kernel/traps. c 中的一个函数 void __init trap_init(void) 把各自陷和中断服务程序的入口地址设置到 idt 表中，其中系统调用总控程序 system_cal 就是中断服务程序之一；void __init trap_init (void) 函数则通过调用一个宏 set_system_gate(SYSCALL_VECTOR, &system_call)；把系统调用总控程序的入口挂在中断 0x80 上。

其中 SYSCALL_VECTOR 是定义在 /usr/src/linux/arch/i386/kernel/irq. h 中的一个常量 0x80，而 system_call 即为中断总控程序的入口地址，中断总控程序用汇编语言定义在/usr/src/linux/arch/i386/kernel/entry. S 中。

③ 中断总控程序主要负责保存处理机执行系统调用前的状态，检验当前调用是否合法，并根据系统调用向量，使处理机跳转到保存在 sys_call_table 表中的相应系统服务例程的入口，从系统服务例程返回后恢复处理机状态退回用户程序。

而系统调用向量则定义在/usr/src/linux/include/asm－386/unistd. h 中，sys_call_table 表定义在 /usr/src/linux/arch/i386/kernel/entry. S 中，同时在 /usr/src/linux/include/asm－386/unistd. h 中也定义了系统调用的用户编程接口。

④ 由此可见，Linux 的系统调用也像 dos 系统的 int 21h 中断服务，它把 0x80 中断作为总的入口，然后转到保存在 sys_call_table 表中的各种中断服务例程的入口地址，形成各种不同的中断服务。

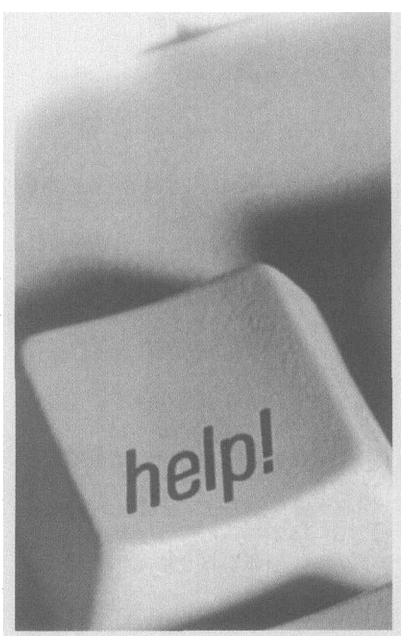
由以上源代码分析可知，要增加一个系统调用就必须在 sys_call_table 表中增加一项，并在其中保存好自己的系统服务例程的入口地址，然后重新编译内核，当然，系统服务例程是必不可少的。

由此可知，在此版 Linux 内核源程序 <2. 2. 5> 中，与系统调用相关的源程序文件就包括以下这些:

* arch/i386/boot/bootsect. S
* rch/i386/Kernel/setup. S
* rch/i386/boot/compressed/head. S
* rch/i386/kernel/head. S
* nit/main. c
* rch/i386/kernel/traps. c
* rch/i386/kernel/entry. S
* rch/i386/kernel/irq. h
* nclude/asm－386/unistd. h

当然，这只是涉及到的几个主要文件。而事实上，增加系统调用真正要修改的文件只有 include/asm－386/unistd. h 和 arch/i386/kernel/entry. S 两个。

(3)源码的修改



① kernel/sys. c 中增加系统服务例程如下：

```
asmlinkage int sys_addtotal(int numdata)
{
int i = 0, enddata = 0;
while(i <= numdata)
enddata + = i + +;
return enddata;
}
```

该函数有一个 int 型入口参数 numdata，并返回从 0 到 numdata 的累加值，然而也可以把系统服务例程放在一个自己定义的文件或其他文件中，只是要在相应文件中作必要的说明。

②把 smlinkage int sys_addtotal(int) 的入口地址加到 sys_call_table 表中。

arch/i386/kernel/entry. S 中的最后几行源代码修改前为：

```
... ...
. long SYMBOL_NAME(sys_sendfile)
. long SYMBOL_NAME(sys_ni_syscall) /* streams1 */
. long SYMBOL_NAME(sys_ni_syscall) /* streams2 */
. long SYMBOL_NAME(sys_vfork) /* 190 */
reptNR_syscalls - 190
. long SYMBOL_NAME(sys_ni_syscall)
. endr 修改后为: ... ...
. long SYMBOL_NAME(sys_sendfile)
. long SYMBOL_NAME(sys_ni_syscall)/* streams1 */
. long SYMBOL_NAME(sys_ni_syscall)/* streams2 */
. long SYMBOL_NAME(sys_vfork) /* 190 */
/* add by I */
. long SYMBOL_NAME(sys_addtotal)
. rept NR_syscalls - 191
. long SYMBOL_NAME(sys_ni_syscall)
. endr
```

③把增加的 sys_call_table 表项所对应的向量，在 include/asm - 386/unistd. h 中进行必要申明，以供用户进程和其他系统进程查询或调用。

增加后的部分 /usr/src/linux/include/asm - 386/unistd. h 文件如下：

```
... ...
#define __NR_sendfile 187
#define __NR_getpmsg 188
#define __NR_putpmsg 189
#define __NR_vfork 190
/* add by I */
#define __NR_addtotal 191
```

④ 测试程序(test. c)如下：

```
#include <linux/unistd. h>
#include <stdio. h>
_syscall1(int, addtotal, int, num)
main()
{
int i, j;
do
printf( " Please input a number) \");
while(scanf("%d", &i) = = EOF);
if((j = addtotal(i)) = = -1)
printf( " Error occurred in syscall - addtotal(); \n");
printf("Total from 0 to %d is %d \)n", i, j);
}
```

对修改后的新的内核进行编译，

并引导它作为新的操作系统，运行几个程序后可以发现一切正常；在新的系统下对测试程序进行编译（注：由于原内核并未提供此系统调用，所以只有在编译后的新内核下，此测试程序才可能被编译通过），运行情况如下：

```
$gcc -o test test.c
$./test
Please input a number
36
Total from 0 to 36 is 666
```

可见，修改成功，而且对相关源码的进一步分析可知，在此版本的内核中，从/usr/src/linux/arch/i386/kernel/entry.S 文件中对 sys_call_table 表的设置可以看出，有好几个系统调用的服务例程都是定义在 /usr/src/linux/kernel/sys.c 中的同一个函数：

```
asmlinkage int sys_ni_syscall
(void)
{
return -ENOSYS;
}
```

例如第 188 项和第 189 项就是如此：

```
... ...
.long SYMBOL_NAME
(sys_sendfile)
.long SYMBOL_NAME
(sys_ni_syscall) /*
streams1 */
.long SYMBOL_NAME
(sys_ni_syscall) /*
streams2 */
.long SYMBOL_NAME(sys_vfork)
/* 190 */
... ...
```

而这两项在文件 /usr/src/linux/include/asm-386/unistd.h 中却申明如下：

```
... ...
#define __NR_sendfile 187
#define __NR_getpmsg 188 /*
some people actually want streams */
#define __NR_putpmsg 189 /*
some people actually want streams */
#define __NR_vfork 190
```

由此可见，在此版本的内核源代码中，由于 asmlinkage int sys_ni_syscall (void) 函数并不进行任何操作，所以包括 getpmsg，putpmsg 在内的好几个系统调用都是不进行任何操作的，即有待扩充的空调用；但它们却仍然占用着 sys_call_table 表项，估计这是设计者们为了方便扩充系统调用而安排的，所以只需增加相应服务例程（如增加服务例程 getmsg 或 putpmsg），就可以达到增加系统调用的作用。

3. 结束语

当然对于庞大复杂的 Linux 而言，一篇文章远远不够，而且与系统调用相关的代码也只是内核中极其微小的一部分，重要的是方法，掌握好的分析方法，所以上述分析只是起个引导作用，而真正的分析还有待读者自己的努力。