《Linux 内核修炼之道》

作者：华清远见

专业始于专注 卓识源于远见

第 5 章 系统调用

本章简介

大部分介绍 Linux 内核的书籍都没有仔细说明系统调用，这应该算是一个失误。内核发展到现在，我们实际需要的系统调用现在已经十分完美，从这个意义上来说，再耗费宝贵的时间去研究系统调用的实现是毫无意义的事情。

然而，对于希望能够对内核有更深理解的我们来说，仔细研究少量系统调用仍是十分值得的。这样就有机会初步了解一些概念，并可以趁机详细了解一下内核编程的特点，就像系统调用本身在应用程序和内核间的桥梁作用一样，学习并理解它也是我们走向内核的一个很好的过渡。

5.1 系统调用概述

一个稳定运行的 Linux 操作系统需要内核和用户应用程序之间的完美配合，内核提供各种各样的服务，然后用户应用程序通过某种途径使用这些服务，进而契合用户的不同需求。

用户应用程序访问并使用内核所提供的各种服务的途径即是系统调用。在内核和用户应用程序相交界的地方，内核提供了一组系统调用接口，通过这组接口，应用程序可以访问系统硬件和各种操作系统资源。比如用户可以通过文件系统相关的系统调用，请求系统打开文件、关闭文件或读写文件；可以通过时钟相关的系统调用，获得系统时间或设置定时器等。

内核提供的这组系统调用通常也被称之为系统调用接口层。系统调用接口层作为内核和用户应用程序之间的中间层，扮演了一个桥梁，或者说中间人的角色。系统调用把应用程序的请求传达给内核，待内核处理完请求后再将处理结果返回给应用程序。

5.1.1 系统调用、POSIX、C 库、系统命令和内核函数

（1）系统调用和 POSIX。

系统调用虽然是内核和用户应用程序之间的沟通桥梁，是用户应用程序访问内核的入口点，但通常情况下，应用程序是通过操作系统提供的应用编程接口（API）而不是直接通过系统调用来编程。

操作系统 API 的主要作用是把操作系统的功能完全展示出来，提供给应用程序，基于该操作系统，与文件、内存、时钟、网络、图形、各种外设等互操作的能力。此外，操作系统 API 通常还提供许多工具类的功能，比如操纵字符串、各种数据类型、时间日期等。

在 UNIX 世界里，最通用的操作系统 API 基于 POSIX（Portable Operating System Interface of UNIX，可移植操作系统接口）标准。POSIX 的诞生和 UNIX 的发展密不可分，UNIX 于 20 世纪 70 年代诞生于 Bell lab，并于 20 世纪 80 年代向美各大高校分发 V7 版的源码以做研究。UC Berkeley 在 V7 的基础上开发了 BSD UNIX。

后来很多商业厂家意识到 UNIX 的价值也纷纷以 Bell Lab 的 System V 或 BSD 为基础来开发自己的 UNIX，较著名的有 Sun OS、AIX、VMS 等。虽然这带来了 UNIX 的繁荣，但由于各厂家对 UNIX 的开发各自为政，UNIX 的版本相当混乱，给软件的可移植性带来很大困难，对 UNIX 的发展极为不利。

为结束这种局面，IEEE 制订了 POSIX 标准，目标是提供一套大体上基于 UNIX 的可移植操作系统标准，提高 UNIX 环境下应用程序的可移植性。然而，POSIX 并不局限于 UNIX。许多其他的操作系统，例如 DEC OpenVMS 和 Microsoft Windows NT，都支持 POSIX 标准

POSIX 标准定义了“POSIX 兼容”的操作系统所必须提供的服务。Linux 兼容于 POSIX 标准，提供了根据 POSIX 而定义的 API 函数。这些 API 函数和系统调用之间有着直接的关系，一个 API 函数可以由一个系统调用实现，也可以通过调用多个系统调用来实现，还可以完全不使用任何系统调用。

（2）系统调用和 C 库。

操作系统 API 通常都以 C 库的方式提供，Linux 也是如此。C 库提供了 POSIX 的绝大部分 API，同时，内核提供的每个系统调用在 C 库中都具有相应的封装函数。系统调用与其 C 库封装函数的名称常常相同，比如，read 系统调用在 C 库中的封装函数即为 read 函数。

C 库中的系统调用封装函数在最终调用到相应系统调用之前，往往不做多少额外的工作。不过，某些情况下会有些例外，比如对于两个相关的系统调用 truncate 和 truncate64，C 库中的封装函数 truncate 函数即需要决定它们中的哪个应该最终被调用。

当然，如图 5.1 所示，系统调用和 C 库函数之间并不是一一对应的关系。可能几个不同的函数会调用到同一个系统调用，比如 malloc 函数和 free 函数都是通过 brk 系统调用来扩大或缩小进程的堆栈，execl、execlp、execle、execv、execvp 和 execve 函数都是通过 execve 系统调用来执行一个可执行文件。

也有可能一个函数调用多个系统调用。更有些函数并不依赖于任何系统调用，比如 strcpy 函数（复制字符串）和 atoi 函数（转换 ASCII 为整数），因为它们并不需要向内核请求任何服务。

华清远见

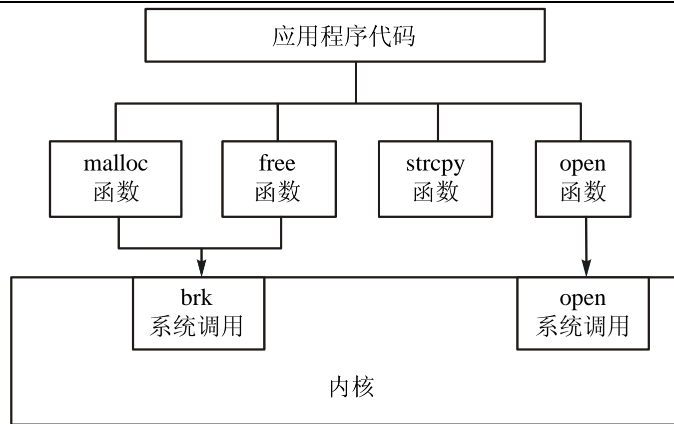

图 5.1 C 库函数与系统调用

实际上，从用户的角度看，系统调用和 C 库之间的区别并不重要，他们只需通过 C 库函数完成所需功能。相反，从内核的角度看，需要考虑的则是提供哪些针对确定目的的系统调用，并不需要关注它们如何被使用。

（3）系统调用与系统命令。

系统命令位于 C 库的更上层，是利用 C 库实现的可执行程序，比如最为常用的 ls、cd 等命令。

strace 工具可以跟踪命令的执行，使用希望跟踪的命令为参数，并显示出该命令执行过程中所使用到的所有系统调用。比如，如果希望了解在执行 pwd 命令时都调用了哪些系统调用，可以使用下面的命令：

```
$strace pwd
```

结果会产生大量的信息，显示出 pwd 命令执行过程中所调用到的各个系统调用：

```
……
write(1, "/usr/src/linux-2.6.23\n", 22/usr/src/linux-2.6.23) = 22
close(1)                                  = 0
munmap(0xb7f5a000, 4096)                  = 0
exit_group(0)
```

（4）系统调用和内核函数。

内核函数与 C 库函数的区别仅仅是内核函数在内核实现，因此必须遵守内核编程的规则。

系统调用最终必须具有明确的操作。用户应用程序通过系统调用进入内核后，会执行各个系统调用对应的内核函数，即系统调用服务例程，比如系统调用 getpid 的服务例程是内核函数 sys_getpid。

系统调用服务例程之外，内核中存在着大量的内核函数。有些局限于某个内核文件自己使用，有些则是 export 出来供内核其他部分共同使用。对于 export 出来的内核函数，可以使用 ksyms 命令或通过 /proc/ksyms 文件查看。

5.1.2 系统调用表

系统调用表 sys_call_table 存储了所有系统调用对应的服务例程的函数地址，在 arch/i386/kernel/syscall_table.S 文件中被定义：

```
001 ENTRY(sys_call_table)
002     .long sys_restart_syscall    /* 0 - old "setup()" system call, used for restarting */
003     .long sys_exit
004     .long sys_fork
005     .long sys_read
006     .long sys_write
007     .long sys_open        /* 5 */
    ……
320     .long sys_getcpu
321     .long sys_epoll_pwait
322     .long sys_utimensat      /* 320 */
```

```
323     .long sys_signalfd
324     .long sys_timerfd
325     .long sys_eventfd
326     .long sys_fallocate
```

从中可发现两个特别之处。首先，所有系统调用服务例程的命名均遵守一定的规则，即在系统调用名称之前增加“sys_”前缀，比如 open 系统调用对应 sys_open 函数。

其次，内核提供的系统调用数目非常有限，到 2.6.23 版本的内核也不过才达到仅仅 325 个，使用“man 2 syscalls”命令即可以浏览到所有系统调用的添加历史。这也是系统调用与 C 库函数的区别之一：系统调用通常只提供最小的接口，C 库函数则在此基础之上提供更多复杂的功能。

5.1.3 系统调用号

既然系统调用表集中存放了所有系统调用服务例程的地址，那么系统调用在内核中的执行就可以转化为从该表获取对应的服务例程并执行的过程。

这个过程中一个很重要的环节就是系统调用号。每个系统调用都拥有一个独一无二的系统调用号，用户应用通过它，而不是系统调用的名称，来指明要执行哪个系统调用。

系统调用号的定义在 include/asm-i386/unistd.h 文件。

```
008 #define __NR_restart_syscall     0
007 #define __NR_exit           1
009 #define __NR_fork           2
010 #define __NR_read           3
011 #define __NR_write          4
012 #define __NR_open           5
……
326 #define __NR_getcpu      318
327 #define __NR_epoll_pwait 319
328 #define __NR_utimensat       320
329 #define __NR_signalfd        321
330 #define __NR_timerfd     322
331 #define __NR_eventfd     323
332 #define __NR_fallocate       324
```

将其与 sys_call_table 的定义相比较可以发现，每个系统调用号都依次对应了 sys_call_table 中的某一项。内核正是将系统调用号作为下标去获取 sys_call_table 中的服务例程函数地址。

系统调用号与系统调用为相依相生的关系，一旦分配就不能再有任何变更，即使该系统调用被删除，它所拥有的系统调用号也不能被回收利用。

5.1.4 系统调用服务例程

系统调用最终由系统调用服务例程完成明确的操作。所有的系统调用服务例程集中声明在 include/linux/syscalls.h 文件，但分散定义在很多不同的文件。比如 getpid 系统调用用于获取当前进程的 PID，它的服务例程 sys_getpid 在 kernel/timer.c 文件中定义为：

```
954 asmlinkage long sys_getpid(void)
955 {
956     return current->tgid;
957 }
```

除了都具有“sys_”前缀之外，所有的系统调用服务例程命名与定义还必须遵守其他的一些规则。首先，函数定义中必须添加 asmlinkage 标记，通知编译器仅从堆栈中获取该函数的参数。

其次，必须返回一个 long 类型的返回值表示成功或错误，通常返回 0 表示成功，返回负值表示错误。当然，getpid 系统调用非常简单，不可能会失败，通过命令“man 2 getpid”可以查看它的手册，里面也明确指出了这一点。

每个系统调用的系统调用号、命名以及操作目的都是固定的，但内核如何去实现并没有明确规定，不同版本、不同架构的内核实现都有可能会有所变化。

5.1.5 如何使用系统调用

如图 5.2 所示，用户应用可以通过两种方式使用系统调用。第一种方式是通过 C 库函数，包括系统调用在 C 库中的封装函数和其他普通函数。

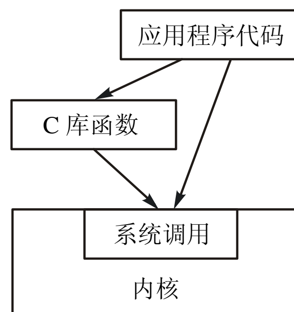

图 5.2 使用系统调用的两种方式

第二种方式是使用_syscall 宏。2.6.18 版本之前的内核，在 include/asm-i386/unistd.h 文件中定义有 7 个_syscall 宏，分别是：

```
_syscall0(type,name)
_syscall1(type,name,type1,arg1)
_syscall2(type,name,type1,arg1,type2,arg2)
_syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)
_syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)
_syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)
_syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6)
```

其中，type 表示所生成系统调用的返回值类型，name 表示该系统调用的名称，type*N*、arg*N* 分别表示第 *N* 个参数的类型和名称，它们的数目和_syscall 后面的数字一样大。这些宏的作用是创建名为 name 的函数，_syscall 后面跟的数字指明了该函数的参数的个数。

比如 sysinfo 系统调用用于获取系统总体统计信息，使用_syscall 宏定义为：

```
_syscall1(int, sysinfo, struct sysinfo *, info);
```

展开后的形式为：

```
int sysinfo(struct sysinfo * info)
{
    long __res;
    __asm__ volatile("int $0x80" : "=a" (__res) : "0" (116),"b" ((long)(info)));
    do {
        if ((unsigned long)(__res) >= (unsigned long)(-(128 + 1))) {
            errno = -(__res);
            __res  = -1;
        }
        return (int) (__res);
    } while (0);
}
```

可以看出，_syscall1(int, sysinfo, struct sysinfo *, info)展开成一个名为 sysinfo 的函数，原参数 int 就是函数的返回类型，原参数 struct sysinfo *和 info 分别构成新函数的参数。

在程序文件里使用_syscall 宏定义需要的系统调用，就可以在接下来的代码中通过系统调用名称直接调用该系统调用。下面是一个使用 sysinfo 系统调用的实例。

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

代码清单 5.1 sysinfo 系统调用使用实例

```
00 #include <stdio.h>
01 #include <stdlib.h>
02 #include <errno.h>
03 #include <linux/unistd.h>
04 #include <linux/kernel.h>      /* for struct sysinfo */
05
06 _syscall1(int, sysinfo, struct sysinfo *, info);
07
08 int main(void)
09 {
10    struct sysinfo s_info;
11    int error;
12
13    error = sysinfo(&s_info);
14    printf("code error = %d\n", error);
15    printf("Uptime = %lds\nLoad: 1 min %lu / 5 min %lu / 15 min %lu\n"
16        "RAM: total %lu / free %lu / shared %lu\n"
17        "Memory in buffers = %lu\nSwap: total %lu / free %lu\n"
18        "Number of processes = %d\n",
19        s_info.uptime, s_info.loads[0],
20        s_info.loads[1], s_info.loads[2],
21        s_info.totalram, s_info.freeram,
22        s_info.sharedram, s_info.bufferram,
23        s_info.totalswap, s_info.freeswap,
24        s_info.procs);
25        exit(EXIT_SUCCESS);
26 }
```

但是自 2.6.19 版本开始，_syscall 宏被废除，我们需要使用 syscall 函数，通过指定系统调用号和一组参数来调用系统调用。

syscall 函数原型为：

```
int syscall(int number, ...);
```

其中 number 是系统调用号，number 后面应顺序接上该系统调用的所有参数。下面是 gettid 系统调用的调用实例。

代码清单 5.2 gettid 系统调用使用实例

```
00 #include <unistd.h>
01 #include <sys/syscall.h>
02 #include <sys/types.h>
03
04 #define __NR_gettid       224
05
06 int main(int argc, char *argv[])
07 {
08    pid_t tid;
09
10    tid = syscall(__NR_gettid);
11 }
```

大部分系统调用都包括了一个 SYS_符号常量来指定自己到系统调用号的映射，因此上面第 10 行可重写为：

```
tid = syscall(SYS_gettid);
```

5.1.6 为什么需要系统调用

为什么需要系统调用？主要有以下两方面原因。

（1）系统调用可以为用户空间提供访问硬件资源的统一接口，以至于应用程序不必去关注具体的硬件访问操作。比如，读写文件时，应用程序不用去管磁盘类型，甚至于不用关心是哪种文件系统。

（2）系统调用可以对系统进行保护，保证系统的稳定和安全。系统调用的存在规定了用户进程进入内核的具体方式，换句话说，用户访问内核的路径是事先规定好的，只能从规定位置进入内核，而不准许肆意跳入内核。有了这样的进入内核的统一访问路径限制才能保证内核的安全。

我们可以形象地描述这种机制：作为一个游客，你可以买票要求进入野生动物园，但你必须老老实实地坐在观光车上，按照规定的路线观光游览。当然，不准下车，因为那样太危险，不是让你丢掉小命，就是让你吓坏了野生动物。

5.2 系统调用执行过程

系统调用的执行过程主要包括如图 5.3 与图 5.4 所示的两个阶段：用户空间到内核空间的转换阶段，以及系统调用处理程序 system_call 函数到系统调用服务例程的阶段。

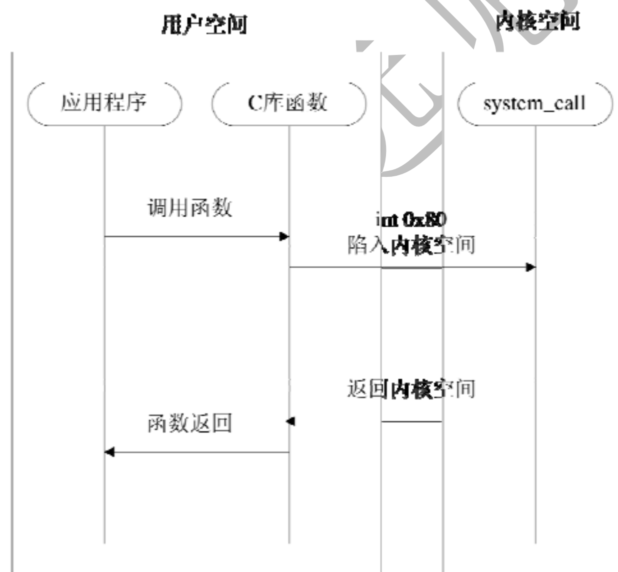

图 5.3 用户空间到内核空间

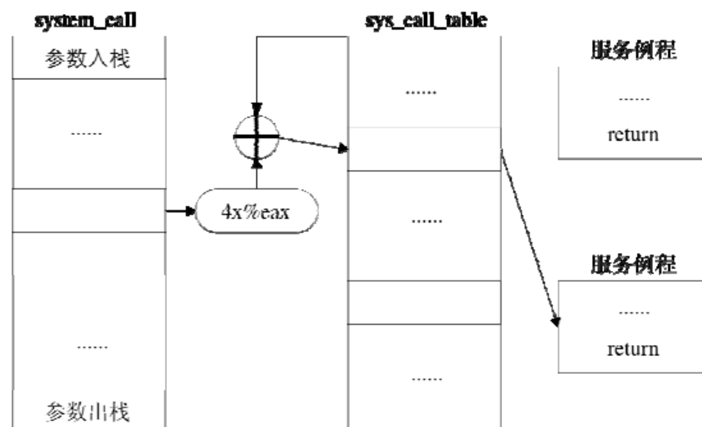

图 5.4 system_call 函数到系统调用服务例程

（1）用户空间到内核空间。

如图 5.3 所示，系统调用的执行需要一个用户空间到内核空间的状态转换，不同的平台具有不同的指令可以完成这种转换，这种指令也被称作操作系统陷入（operating system trap）指令。

Linux 通过软中断来实现这种陷入，具体对于 X86 架构来说，是软中断 0x80，也即 int $0x80 汇编指令。软中断和我们常说的中断（硬件中断）不同之处在于——它由软件指令触发而并非由硬件外设引发。

int 0x80 指令被封装在 C 库中，对于用户应用来说，基于可移植性的考虑，不应该直接调用 int $0x80 指令。陷入指令的平台依赖性，也正是系统调用需要在 C 库进行封装的原因之一。

通过软中断 0x80，系统会跳转到一个预设的内核空间地址，它指向了系统调用处理程序（不要和系统调用服务例程相混淆），即在 arch/i386/kernel/entry.S 文件中使用汇编语言编写的 system_call 函数。

（2）system_call 函数到系统调用服务例程。

很显然，所有的系统调用都会统一跳转到这个地址进而执行 system_call 函数，但正如前面所述，到 2.6.23 版为止，内核提供的系统调用已经达到了 325 个，那么 system_call 函数又该如何派发它们到各自的服务例程呢？

软中断指令 int 0x80 执行时，系统调用号会被放入 eax 寄存器，同时，sys_call_table 每一项占用 4 个字节。这样，如图 5.5 所示，system_call 函数可以读取 eax 寄存器获得当前系统调用的系统调用号，将其乘以 4 生成偏移地址，然后以 sys_call_table 为基址，基址加上偏移地址所指向的内容即是应该执行的系统调用服务例程的地址。

另外，除了传递系统调用号到 eax 寄存器，如果需要，还会传递一些参数到内核，比如 write 系统调用的服务例程原型为：

```
sys_write(unsigned int fd, const char * buf, size_t count);
```

调用 write 系统调用时就需要传递文件描述符 fd、要写入的内容 buf 以及写入字节数 count 等几个内容到内核。ebx、ecx、edx、esi 以及 edi 寄存器可以用于传递这些额外的参数。

正如之前所述，系统调用服务例程定义中的 asmlinkage 标记表示，编译器仅从堆栈中获取该函数的参数，而不需要从寄存器中获得任何参数。进入 system_call 函数前，用户应用将参数存放到对应寄存器中，system_call 函数执行时会首先将这些寄存器压入堆栈。

对于系统调用服务例程，可以直接从 system_call 函数压入的堆栈中获得参数，对参数的修改也可以一直在堆栈中进行。在 system_call 函数退出后，用户应用可以直接从寄存器中获得被修改过的参数。

并不是所有的系统调用服务例程都有实际的内容，有一个服务例程 sys_ni_syscall 除了返回-ENOSYS 外不做任何其他工作，在 kernel/sys_ni.c 文件中定义。

```
10 asmlinkage long sys_ni_syscall(void)
11 {
12        return -ENOSYS;
13 }
```

sys_ni_syscall 的确是最简单的系统调用服务例程，表面上看，它可能并没有什么用处，但是，它在 sys_call_table 中占据了很多位置。多数位置上的 sys_ni_syscal 都代表了那些已经被内核中淘汰的系统调用，比如：

```
.long sys_ni_syscall /* old stty syscall holder */
.long sys_ni_syscall /* old gtty syscall holder */
```

就分别代替了已经废弃的 stty 和 gtty 系统调用。如果一个系统调用被淘汰，它所对应的服务例程就要被指定为 sys_ni_syscall。

我们并不能将它们的位置分配给其他的系统调用，因为一些老的代码可能还会使用到它们。否则，如果某个用户应用试图调用这些已经被淘汰的系统调用，所得到的结果，比如打开了一个文件，就会与预期完全不同，这将令人感到非常奇怪。

其实，sys_ni_syscall 中的“ni”即表示“not implemented（没有实现）”。

系统调用通过软中断 0x80 陷入内核，跳转到系统调用处理程序 system_call 函数，并执行相应的服务例程，但由于是代表用户进程，所以这个执行过程并不属于中断上下文，而是处于进程上下文。

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京|上海|深圳|成都|南京|武汉|西安|广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

因此，系统调用执行过程中，可以访问用户进程的许多信息，可以被其他进程抢占（因为新的进程可能使用相同的系统调用，所以必须保证系统调用可重入），可以休眠（比如在系统调用阻塞时或显式调用 schedule 函数时）。

这些特点涉及进程调度的问题，在此不做深究，读者只需要理解当系统调用完成后，把控制权交回到发起调用的用户进程前，内核会有一次调度。如果发现有优先级更高的进程或当前进程的时间片用完，那么就会选择高优先级的进程或重新选择进程运行。

5.3 系统调用示例

本节通过对几个系统调用的剖析来讲解它们的工作方式。

5.3.1 sys_dup

dup 系统调用的服务例程为 sys_dup 函数，在 fs/fcntl.c 文件中定义如下。

代码清单 5.3 dup 系统调用的服务例程

```
192 asmlinkage long sys_dup(unsigned int fildes)
193 {
194     int ret = -EBADF;
195     struct file * file = fget(fildes);
196 
197     if (file)
198         ret = dupfd(file, 0);
199     return ret;
200 }
```

除了 sys_ni_call()以外，sys_dup()称得上是最简单的服务例程之一，但是它却是 Linux 输入/输出重定向的基础。

在 Linux 中，执行一个 shell 命令时通常会自动打开 3 个标准文件：标准输入文件（stdin），通常对应终端的键盘；标准输出文件（stdout）和 标准错误输出文件（stderr），通常对应终端的屏幕。shell 命令从标准输入文件中得到输入数据，将输出数据输出到标准输出文件，而将错误信息输出到标准错误文件中。

比如下面的命令：

```
$cat /proc/cpuinfo
```

将把 cpuinfo 文件的内容显示到屏幕上，但是如果 cat 命令不带参数，则会从 stdin 中读取数据，并将其输出到 stdout，比如：

```
$cat
Hello!
Hello!
```

用户输入的每一行都将立刻被输出到屏幕上。

输入重定向是指把命令的标准输入重定向到指定的文件中，即输入可以不来自键盘，而来自一个指定的文件。所以说，输入重定向主要用于改变一个命令的输入源。

输出重定向是指把命令的标准输出或标准错误输出重新定向到指定文件中。这样，该命令的输出就不显示在屏幕上，而是写入到指定文件中。我们经常会利用输出重定向将程序或命令的 log 保存到指定的文件中。

那么 sys_dup()又是如何完成输入/输出的重定向呢？下面通过一个例子进行说明。

当我们在 shell 终端下输入“echo hello”命令时，将会要求 shell 进程执行一个可执行文件 echo，参数为“hello”。当 shell 进程接收到命令之后，先在/bin 目录下找到 echo 文件（我们可以使用 which 命令获得命令所在的位置），然后创建一个子进程去执行/bin/echo，并将参数传递给它，而这个子进程从 shell 进程

继承了 3 个标准输入/输出文件，即 stdin、stdout 和 stderr，文件号分别为 0、1、2。它的工作很简单，就是将参数“hello”写到 stdout 文件中，通常都是我们的屏幕上。

但是如果我们将命令改成“echo hello > txt”，则在执行时输出将会被重定向到磁盘文件 txt 中。假定之前该 shell 进程只有上述 3 个标准文件打开，则该命令将按如下序列执行。

（1）打开或创建文件 txt，如果 txt 中原来有内容，则清除原来的内容，其文件号为 3。

（2）通过 dup 系统调用复制文件 stdout 的相关数据结构到文件号 4。

（3）关闭 stdout，但是由于 4 号文件也同时引用 stdout，所以 stdout 文件并未真正关闭，只是腾出 1 号文件号位置。

（4）通过 dup 系统调用，复制 3 号文件（即文件 txt），由于 1 号文件关闭，其位置空缺，故 3 号文件被复制到 1 号，即进程中原来指向 stdout 的指针指向了 txt。

（5）通过系统调用 fork 和 exec 创建子进程并执行 echo，子进程在执行 cat 前关闭 3 号和 4 号文件，只留下 0、1、2 三个文件，请注意，这时的 1 号文件已经不是 stdout 而是文件 txt 了。当 cat 想向 stdout 文件写入“hello”时自然就写入到了 txt 中。

（6）回到 shell 进程后，关闭指向 txt 的 1 号与 3 号文件文件，再用 dup 和 close 系统调用将 2 号恢复至 stdout，这样 shell 就恢复了 0、1、2 三个标准输入/输出文件。

5.3.2 sys_reboot

Linux 下有关关机与重启的命令主要有 shutdown、reboot、halt、poweroff、telinit 和 init。它们都可以达到关机或重启的目的，但是每个命令的工作流程并不一样。

这些命令并不都是互相独立的，比如，poweroff、reboot 即是 halt 的符号链接，但是它们最终都是通过 reboot 系统调用来完成关机或重启操作。

reboot 系统调用的服务例程为 sys_reboot 函数，在 kernel/sys.c 文件中定义如下。

代码清单 5.4 reboot 系统调用的服务例程

```
896 asmlinkage long sys_reboot(int magic1, int magic2, unsigned int cmd, void __user * arg)
897 {
898     char buffer[256];
899 
900     /* We only trust the superuser with rebooting the system. */
901     if (!capable(CAP_SYS_BOOT))
902         return -EPERM;
903 
904     /* For safety, we require "magic" arguments. */
905     if (magic1 != LINUX_REBOOT_MAGIC1 ||
906        (magic2 != LINUX_REBOOT_MAGIC2 &&
907                 magic2 != LINUX_REBOOT_MAGIC2A &&
908             magic2 != LINUX_REBOOT_MAGIC2B &&
909                 magic2 != LINUX_REBOOT_MAGIC2C))
910         return -EINVAL;
911 
912     /* Instead of trying to make the power_off code look like
913      * halt when pm_power_off is not set do it the easy way.
914      */
915     if ((cmd == LINUX_REBOOT_CMD_POWER_OFF) && !pm_power_off)
916         cmd = LINUX_REBOOT_CMD_HALT;
917 
918     lock_kernel();
919     switch (cmd) {
920     case LINUX_REBOOT_CMD_RESTART:
```

```
921             kernel_restart(NULL);
922             break;
923
924         case LINUX_REBOOT_CMD_CAD_ON:
925             C_A_D = 1;
926             break;
927
928         case LINUX_REBOOT_CMD_CAD_OFF:
929             C_A_D = 0;
930             break;
931
932         case LINUX_REBOOT_CMD_HALT:
933             kernel_halt();
934             unlock_kernel();
935             do_exit(0);
936             break;
937
938         case LINUX_REBOOT_CMD_POWER_OFF:
939             kernel_power_off();
940             unlock_kernel();
941             do_exit(0);
942             break;
943
944         case LINUX_REBOOT_CMD_RESTART2:
945             if (strncpy_from_user(&buffer[0], arg, sizeof(buffer) - 1) < 0) {
946                 unlock_kernel();
947                 return -EFAULT;
948             }
949             buffer[sizeof(buffer) - 1] = '\0';
950
951             kernel_restart(buffer);
952             break;
953
954         case LINUX_REBOOT_CMD_KEXEC:
955             kernel_kexec();
956             unlock_kernel();
957             return -EINVAL;
958
959 #ifdef CONFIG_HIBERNATION
960         case LINUX_REBOOT_CMD_SW_SUSPEND:
961             {
962                 int ret = hibernate();
963                 unlock_kernel();
964                 return ret;
965             }
966 #endif
967
968         default:
969             unlock_kernel();
970             return -EINVAL;
971         }
972         unlock_kernel();
```

```
973        return 0;
974 }
```

顾名思义，reboot 系统调用可以用于重新启动系统，但根据所提供的参数不同，它还能够完成关机、挂起系统、允许或禁止使用 Ctrl+Alt+Del 组合键重启等不同的操作。我们还要特别注意内核里对 sys_reboot() 的注释，在使用它之前首先要使用 sync 命令同步磁盘，否则磁盘上的文件系统可能会有所损坏。

第 901 行检查调用者是否有合法权限。capable 函数用于检查是否有操作指定资源的权限，如果它返回非零值，则调用者有权进行操作，否则无权操作。比如，这一行的 capable(CAP_SYS_BOOT)即检查调用者是否有权限使用 reboot 系统调用。

第 905 行～第 910 行通过对两个参数 magic1 和 magic2 的检测，判断 reboot 系统调用是不是被偶然调用到的。如果 reboot 系统调用是被偶然调用的，那么参数 magic1 和 magic2 几乎不可能同时满足预定义的这几个数字的集合。

从第 919 行开始，sys_reboot() 对调用者的各种使用情况进行区分。为 LINUX_REBOOT_CMD_RESTART 时，kernel_restart()将打印出“Restarting system.”消息，然后调用 machine_restart 函数重新启动系统。

为 LINUX_REBOOT_CMD_CAD_ON 或 LINUX_REBOOT_CMD_CAD_OFF 时，分别允许或禁止 Ctrl+Alt+Del 组合键。我们还可以在/etc/inittab 文件指定是否可以使用 Ctrl+Alt+Del 组合键来关闭并重启系统。如果希望完全禁止这个功能，需要将/etc/inittab 文件中的下面一行注释掉。

```
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
```

为 LINUX_REBOOT_CMD_HALT 时，打印出“System halted.”消息，和 LINUX_REBOOT_CMD_RESTART 情况下类似，但只是暂停系统而不是将其重新启动。

为 LINUX_REBOOT_CMD_POWER_OFF 时，打印出“Power down.”消息，然后关闭机器电源。

为 LINUX_REBOOT_CMD_RESTART2 时，接收命令字符串，该字符串说明了系统应该如何关闭。

最后，LINUX_REBOOT_CMD_SW_SUSPEND 用于使系统休眠。

5.4 系统调用的实现

一个系统调用的实现并不需要去关心如何从用户空间转换到内核空间，以及系统调用处理程序如何去执行，你需要做的只是遵循几个固定的步骤。

5.4.1 如何实现一个新的系统调用

为 Linux 添加新的系统调用是件相对容易的事情，主要包括有 4 个步骤：编写系统调用服务例程；添加系统调用号；修改系统调用表；重新编译内核并测试新添加的系统调用。

下面以一个并无实际用处的 hello 系统调用为例，来演示上述几个步骤。

（1）编写系统调用服务例程。

遵循前面所述的几个原则，hello 系统调用的服务例程实现为：

```
01 asmlinkage long sys_hello(void)
02 {
03        printk(“Hello!\n”);
04        return 0;
05 }
```

通常，应该为新的系统调用服务例程创建一个新的文件进行存放，但也可以将其定义在其他文件之中并加上注释做必要说明。同时，还要在 include/linux/syscalls.h 文件中添加原型声明：

```
asmlinkage long sys_hello(void);
```

sys_hello 函数非常简单，仅仅打印一条语句，并没有使用任何参数。如果我们希望 hello 系统调用不仅能打印“hello!”欢迎信息，还能够打印出我们传递过去的名称，或者其他的一些描述信息，则 sys_hello 函数可以实现为：

```
01 asmlinkage long sys_hello(const char __user *_name)
02 {
03       char *name;
04       long ret;
05
06       name = strndup_user(_name, PAGE_SIZE);
07       if (IS_ERR(name)) {
08           ret = PTR_ERR(name);
09           goto error;
10       }
11
12       printk("Hello, %s!\n", name);
13       return 0;
14  error:
15       return ret;
16 }
```

第二个 sys_hello 函数使用了一个参数，在这种有参数传递发生的情况下，编写系统调用服务例程时必须仔细检查所有的参数是否合法有效。因为系统调用在内核空间执行，如果不加限制任由用户应用传递输入进入内核，则系统的安全与稳定将受到影响。

参数检查中最重要的一项就是检查用户应用提供的用户空间指针是否有效。比如上述 sys_hello 函数参数为 char 类型指针，并且使用了__user 标记进行修饰。__user 标记表示所修饰的指针为用户空间指针，不能在内核空间直接引用，原因主要如下。

用户空间指针在内核空间可能是无效的。

用户空间的内存是分页的，可能引起页错误。

如果直接引用能够成功，就相当于用户空间可以直接访问内核空间，产生安全问题。

因此，为了能够完成必须的检查，以及在用户空间和内核空间之间安全地传送数据，就需要使用内核提供的函数。比如在 sys_hello 函数的第 6 行，就使用了内核提供的 strndup_user 函数（在 mm/util.c 文件中定义）从用户空间复制字符串 name 的内容。

（2）添加系统调用号。

每个系统调用都会拥有一个独一无二的系统调用号，所以接下来需要更新 include/asm-i386/unistd.h 文件，为 hello 系统调用添加一个系统调用号。

```
328 #define __NR_utimensat       320
329 #define __NR_signalfd        321
330 #define __NR_timerfd     322
331 #define __NR_eventfd     323
332 #define __NR_fallocate       324
333 #define __NR_hello           325  /*分配 hello 系统调用号为 325*/
334
335 #ifdef __KERNEL__
336
337 #define NR_syscalls 326  /*将系统调用数目加 1 修改为 326*/
```

（3）修改系统调用表。

为了让系统调用处理程序 system_call 函数能够找到 hello 系统调用，我们还需要修改系统调用表 sys_call_table，放入服务例程 sys_hello 函数的地址。

```
322 .long sys_utimensat      /* 320 */
323 .long sys_signalfd
```

```
324 .long sys_timerfd
325 .long sys_eventfd
326 .long sys_fallocate
327 .long sys_hello /*hello 系统调用服务例程*/
```

新的系统调用 hello 的服务例程被添加到了 sys_call_table 的末尾。我们可以注意到，sys_call_table 每隔 5 个表项就会有一个注释，表明该项的系统调用号，这个好习惯可以在查找系统调用对应的系统调用号时提供方便。

（4）重新编译内核并测试。

为了能够使用新添加的系统调用，需要重新编译内核，并使用新内核重新引导系统。然后，我们还需要编写测试程序对新的系统调用进行测试。针对 hello 系统调用的测试程序如下：

```
00 #include <unistd.h>
01 #include <sys/syscall.h>
02 #include <sys/types.h>
03
04 #define __NR_hello        325
05
06 int main(int argc, char *argv[])
07 {
08       syscall(__NR_hello);
09       return 0;
10 }
```

然后使用 gcc 编译并执行：

```
$gcc -o hello hello.c
$./hello
Hello!
```

由执行结果可见，系统调用添加成功。

5.4.2 什么时候需要添加新的系统调用

虽说添加一个新的系统调用非常简单，但这并不意味着用户有必要这么做。添加系统调用需要修改内核源代码、重新编译内核，如果更进一步希望自己添加的系统调用能够得到广泛的应用，就需要得到官方的认可并分配一个固定的系统调用号，还需要将该系统调用在每个需要支持的体系结构上实现。因此我们最好使用其他替代方法和内核交换信息，如下所示。

使用设备驱动程序。创建一个设备节点，通过 read 和 write 函数进行读写访问，使用 ioctl 函数进行设置操作和获取特定信息。这种方法最大的好处在于可以模块式加载卸载，避免了编译内核等过程，而且调用接口固定，容易操作。

使用 proc 虚拟文件系统。利用 proc 接口获取系统运行信息和修订系统状态是一种很常见的手段，比如读取/proc/cpuinfo 可以获得当前系统的 CPU 信息，通过设备驱动提供的 proc 接口还可以设置硬件寄存器。

sysfs 文件系统。sysfs 文件系统在 2.6 内核被引入，是一个类似于 proc 文件系统的特殊文件系统，用于对系统的设备进行管理，它把实际连接到系统上的设备和总线组织成层次结构，并向用户提供详细的内核数据结构信息，用户可以利用这些信息以实现和内核的交互。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

华清远见