



第 7 章 系统调用

操作系统作为计算机系统资源的管理者，需要一种手段向用户进程提供服务使得用户进程可以访问这些系统资源，这一手段就是系统调用。通过系统调用，应用程序可以安全、有效地访问和使用计算机系统中的各种资源。

本章内容简介如下。

7.1 节介绍了系统调用与应用编程接口的区别。

7.2 节讲解了访问系统调用的几种不同方法，及其各自的特点和适用场合。

7.3 节讲解了系统调用的工作原理、系统调用门的初始化过程以及系统调用的处理过程。

7.4 节讲解了系统调用和普通函数调用的区别，以及系统调用过程中如何在用户态、内核态传递所需的参数。

7.5 节介绍了如何向系统中添加一个新的系统调用，以及依次的处理步骤。

7.6 节讨论了 Linux 内核对 Intel 在 Pentium II 处理器中引入的快速系统调用指令（sysenter/sysexit）的支持与实现方法。



7.1

系统服务接口的种类

本节从编程者的角度，扼要介绍一下应用编程接口（Application Program Interface，简称为 API）和系统调用（System Call）之间的区别和联系。

7.1.1 系统调用接口

系统调用是内核向用户进程提供服务的唯一手段，用户进程只能直接或间接地通过系统调用来访问系统中的资源。系统调用涉及系统状态的转变，在用户进程使用系统调用访问系统资源时，系统会从用户态切换到内核态，执行相应的内核态函数，完成相应的任务后返回到用户态。

7.1.2 应用编程接口

应用编程接口着重强调如何通过一个接口来获得相应的服务。通常该接口已经稳定下来，成为一种事实的规范或由某些国际标准组织维护的标准，如 IEEE 制定的 POSIX 标准就是一个 API 规范。该规范规定了符合 POSIX 标准的操作系统必须提供的应用编程接口，它并不明确要求接口是通过系统调用还是通过用库函数实现的。实际上，C 库（glibc）除了实现标准 C 规范所要求的库函数外，一个很重要的任务就是提供一套封装例程（wrapper routine）将系统调用封装后供用户编程时使用。

系统调用只是一种应用编程接口、一种 API，而应用编程接口 API 包括其他各种各样的编程接口，如标准 C 语言中提供的库函数接口、BIOS 提供的中断调用接口、图形界面编程中使用的 OpenGL 编程接口等。从应用程序编程者的观点来看，系统调用接口和其他编程接口之间是没有差别的，用户惟一关心的事情就是函数名、参数类型及返回代码的含义。然而，从系统编程者的观点来看，它们是有差别的：系统调用接口与内核态密切相关，引起系统运行级别的转换；而其他编程接口不一定涉及内核态，不一定导致运行级别的转换。

7.2

系统调用的访问手段

本节里介绍可以通过哪些手段访问系统调用接口，依次为使用 libc 中为每个系统调用编写的封装函数、使用 libc 特殊的封装函数 syscall() 以及直接使用内嵌汇编来访问系统调用接口。

7.2.1 使用封装函数

下面介绍如何通过 C 库（glibc）为每个系统调用提供的对应封装函数（wrapper routine）来访问系统调用。这里以返回当前进程 ID 的系统调用 sys_getpid() 为例，看一

下通过调用 `glibc` 中对应的接口，向屏幕打印当前进程的进程标识符 `pid`。请看下面的示例程序 7.1。

```

1 #include "stdio.h"
2 #include "unistd.h"
3
4 int main(int argc, char ** argv)
5 {
6     printf("use libc ,pid is:%d\n",getpid());
7     return(0);
8 }

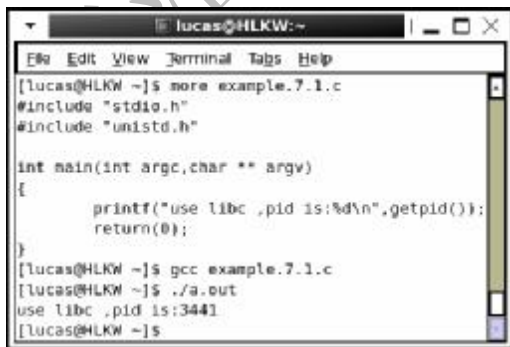
```

示例程序 7.1：使用封装函数访问系统调用

该函数中使用的函数 `getpid()` 是 GNU C 库函数 `glibc` 中为系统调用 `sys_getpid()` 提供的一个封装函数，用于返回当前进程的标识符。它在 GNU C 库提供的头文件 `/usr/include/unistd.h` 中的第 571 行声明如下：

```
extern __pid_t getpid (void) __THROW;
```

上面编写的函数在编译、运行后会返回当前进程的标识符，运行结果如图 7.1 所示。注意每次运行时，进程标识符通常情况下是不一样的。



```

lucas@HLKW:~$ more example.7.1.c
#include "stdio.h"
#include "unistd.h"

int main(int argc, char ** argv)
{
    printf("use libc ,pid is:%d\n",getpid());
    return(0);
}
lucas@HLKW ~]$ gcc example.7.1.c
lucas@HLKW ~]$ ./a.out
use libc ,pid is:3441
lucas@HLKW ~]$

```

图 7.1 使用封装函数访问系统调用

7.2.2 使用通用接口

下面介绍另一种访问系统调用接口的方法，该方法基于 `glibc` 库中提供的通用接口函数 `syscall()` 来访问系统调用。该接口函数在文件 `/usr/include/unistd.h` 中的第 1009 行定义如下。（对于不同的发行版，这里给出的行号可能不一样，此处以系统 Red Hat Enterprise Linux Version 5 Server 为例。）

```
extern long int syscall (long int __sysno, ...) __THROW;
```

既然内核中为不同的系统设置了相应的封装函数，那么为什么还要有该接口函数呢？首先介绍一下该接口函数 `syscall()` 的目的。假设你升级了内核版本且想使用新内

核版本中新增加的系统调用，而系统的 libc 库并没有你想使用的系统调用对应的封装函数，那么此时就是接口函数 `syscall()` 大显身手的时候了。这也是称该函数为通用接口的原因。

通过接口函数 `syscall()` 虽然可以访问系统中所有的系统调用，但是使用通用接口函数 `syscall()` 通常都会造成可移植性和代码可读性都比较差。所以最好还是使用系统调用对应的封装函数。下面的示例程序显示了如何通用接口函数 `syscall()` 访问系统调用 `sys_getpid()`，并将当前进程的进程号打印到屏幕上。

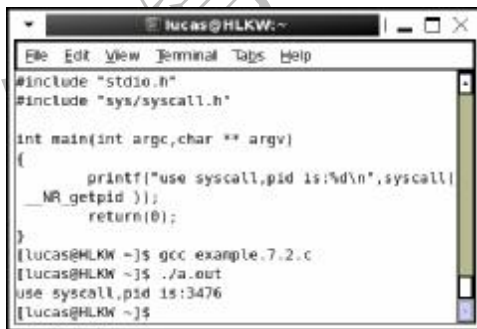
```

1      #include "stdio.h"
2      #include "sys/syscall.h"
3
4      int main(int argc, char ** argv)
5      {
6          printf("use syscall, pid is:%d\n", syscall( __NR_getpid ));
7          return(0);
8      }

```

示例程序 7.2：使用通用接口访问系统调用

该示例程序中的 `__NR_getpid` 是一个宏定义，定义于头文件 `/usr/include/asm/unistd.h` 中的第 28 行，其值为 20，是系统调用 `sys_getpid` 的系统调用号。该头文件包含于头文件 `sys/syscall.h` 中。上述示例程序编译、运行后的结果如图 7.2 所示。



```

lucas@HLKW:~
File Edit View Terminal Tabs Help
#include "stdio.h"
#include "sys/syscall.h"

int main(int argc, char ** argv)
{
    printf("use syscall, pid is:%d\n", syscall(
__NR_getpid ));
    return(0);
}

[lucas@HLKW ~]$ gcc example.7.2.c
[lucas@HLKW ~]$ ./a.out
use syscall, pid is:3476
[lucas@HLKW ~]$

```

图 7.2 使用通用接口访问系统调用

7.2.3 使用内嵌汇编

虽然直接使用内嵌汇编访问系统调用是可行的，但是一般不直接使用内嵌汇编来访问系统提供的系统调用接口，原因如下：

- 1 可移植性差；
- 1 使用方法复杂、难度大。

幸好内核提供了 7 个非常有用的宏定义，依次为：`__syscall0`、`__syscall1`、`__syscall2`、`__syscall3`、`__syscall4`、`__syscall5`、`__syscall6`。它们分别用来封装不同数目参数的系统调用，最后的数字表示该宏定义用于访问需要几个参数的系统调用，如 `__syscall0` 用于访

问不需要参数的系统调用，`_syscall3` 用于访问需要三个参数的系统调用。其中，宏定义 `_syscall0`（其他宏定义的实现类似）在当前内核头文件 `src/include/asm/unistd` 中的第 319 行开始定义，代码如下：

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name)); \
__syscall_return(type,__res); \
}
```

从代码可以看出，每个宏定义都最终使用内嵌汇编来访问 0X80 号软中断。该中断门在系统初始化过程中被设置为系统调用的入口，具体初始化过程，请参见 7.3.2 小节。示例程序 7.3 展示了如何利用宏定义 `_syscall0()` 来访问系统调用 `sys_getpid`，并返回当前进程的进程号。

```
-----
#include "stdio.h"
/*unistd.h must be the curent kernel`s header file*/
#include "/usr/src/kernels/2.6.18-8.el5-i686\
/include/asm/unistd.h"

static int errno;
_syscall0(long,getpid);

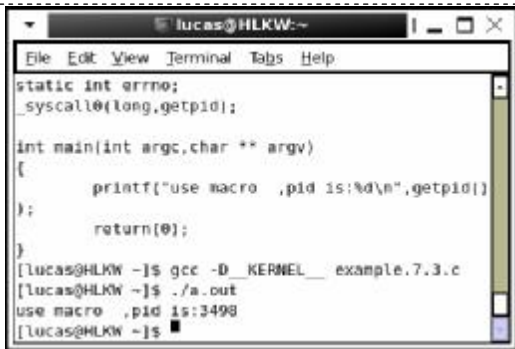
int main(int argc,char ** argv)
{
printf("use macro ,pid is:%d\n",getpid());
return(0);
}
-----
```

示例程序 7.3：使用内嵌汇编访问系统调用

细心的读者可能已经注意到了，到目前为止，我们使用了来自 3 个不同地方的头文件 `unistd.h`，它们分别来自：

```
/usr/include/unistd.h,
/usr/include/asm/unistd.h,
/usr/src/kernels/2.6.18-8.el5-i686/include/asm/unistd.h.
```

下面简要说明一下它们的区别：第 1 个头文件 `unistd.h` 包含了 `glibc` 中对系统调用的封装函数的声明；第 2 个头文件 `unistd.h` 包含当前发行版系统中的系统调用号；第 3 个头文件 `unistd.h` 除了包含当前内核版本的系统调用号还包含了上面所讲的 7 个宏定义。上面的代码段编译、运行后的结果如图 7.3 所示。



```

lucas@HLKW:~$ gcc -D_KERNEL__ example.7.3.c
lucas@HLKW:~$ ./a.out
use macro ,pid is:3498
lucas@HLKW:~$

```

图 7.3 使用内嵌汇编访问系统调用

7.3

系统调用的工作机制

本节讨论 Linux 内核中系统调用的工作机制，同时介绍系统调用的初始化过程以及系统调用的处理过程。下面分小节对这些问题进行讨论和分析。

7.3.1 系统调用的基本要素

用户进程通常情况下运行于用户态；在用户进程请求系统调用后，系统切换到内核态，执行相关的内核态函数为该用户进程服务，用户进程此时运行于内核态；在完成用户进程的系统调用请求后，系统重新返回到用户态，用户进程继续在用户态执行。

1. 系统调用门

从上面的论述可知，一种体系结构的处理器要能运行 Linux 系统，需要提供两种运行级别：供内核使用特权级别和供用户进程使用的受限级别。实际上，当前所有主流的 32 位处理器的架构都有两种以上的运行级别，这是计算机体系结构发展的结果，也是有效实现多用户多进程操作系统的基础。

例如，Intel IA32 体系结构就有 0~3 共 4 个运行级别，0 号运行级别具有最高的权限，3 号运行级别具有最低的权限。Linux 内核在最高权限的 0 号级别运行，称之为内核态；其他软件都在最低权限的 3 号级别运行，称之为用户态。采取不同权限的运行级别主要是为了保护，由于用户进程在最低权限的级别上运行，所以它们不会有意或无意破坏其他进程或内核数据。

不同的运行级别保护了内核数据的安全和其他进程不受影响。那么用户进程如何才能取得内核的服务呢？用户进程如何完成由用户态到内核态的转换呢？和其他多用户多任务操作系统一样，Linux 使用软中断机制来完成用户进程不同运行级别的切换。当用户进程需要完成特权模式下才能完成的某些功能时，必须严格按照系统调用提供接口才能进入特权模式，由系统调用处理函数完成特定的功能。

系统调用接口基于软中断来实现。在内核的初始化过程中为系统调用设置了 0X80

号中断门，系统调用时必须通过该中断门的安全验证，然后切换到进程的内核态栈，内核为该用户进程服务过程中的函数调用使用该内核态栈。详细请参见 7.3.2 小节。

2. 内核态栈

为了保证系统数据不被非法访问，必须保证不同运行级别的数据互相隔绝，即用户进程的每种运行态都有相应的独立栈空间。在 Linux 系统中，一个用户进程包括一个用户态栈和一个内核态栈。用户态栈存储了用户进程运行于用户态时函数调用的参数、局部变量和其他辅助数据，内核态栈相应地包括了用户进程运行于内核态时函数调用的参数、局部变量和其他辅助数据；从而保证了不同运行级别的数据互相隔绝。关于内核态栈的详细论述请参见 3.2.4 小节。

3. 系统调用号

在了解了内核保证用户进程和内核的隔离机制后，接下来分析内核如何判断用户请求了哪一个系统调用。所有的系统调用入口函数均为 `system_call()`，那么该函数必定需要一个参数来表明用户进程请求了哪一个系统调用。实际上，内核为每个系统调用分配了一个惟一的编号，我们称之为系统调用编号。这些编号在文件 `src/include/asm/unistd.h` 中定义，该文件中定义系统调用编号的片段如下所示。

华清远见

代码清单 7.1——系统调用号片段

功能简介：该代码片断保存了系统调用的编号，即系统调用号。

文件：src/include/asm/unistd.h

```

8  #define __NR_restart_syscall    0
9  #define __NR_exit                1
10 #define __NR_fork                2
11 #define __NR_read                3
12 #define __NR_write               4
13 #define __NR_open                5
14 #define __NR_close               6
15 #define __NR_waitpid             7
16 #define __NR_creat               8
17 #define __NR_link                9
18 #define __NR_unlink              10
19 #define __NR_execve              11
20 #define __NR_chdir               12
21 #define __NR_time                13
22 #define __NR_mknod               14
23 #define __NR_chmod               15
24 #define __NR_lchown              16
25 #define __NR_break               17
26 #define __NR_oldstat             18
27 #define __NR_lseek               19
28 #define __NR_getpid              20

```

用户在发出一个系统调用请求时，系统调用号作为参数被保存在处理器的%eax寄存器中，系统调用入口函数 system_call()根据寄存器%eax 中的系统调用编号调用相应的系统调用处理程序，完成用户的系统调用请求。

4. 系统调用表

系统调用号从何而来？编号的依据是什么呢？答案是系统调用表 sys_call_table。系统调用表在文件 src/arch/i386/kernel/syscall_table.S 中定义，其代码片段如下。

代码清单 7.2——系统调用表片段

功能简介：该代码片断保存了系统调用表，系统调用时，内核根据该系统调用表找到相应的系统调用处理函数。

文件：src/arch/i386/kernel/syscall_table.S

```

2  ENTRY(sys_call_table)
3      .long sys_restart_syscall /*0-old "setup()" system call,used
for restarting */

```

```

4      .long sys_exit
5      .long sys_fork
6      .long sys_read
7      .long sys_write
8      .long sys_open      /* 5 */
9      .long sys_close
10     .long sys_waitpid
11     .long sys_creat
12     .long sys_link
13     .long sys_unlink    /* 10 */
14     .long sys_execve
15     .long sys_chdir
16     .long sys_time
17     .long sys_mknod
18     .long sys_chmod     /* 15 */
19     .long sys_lchown16
20     .long sys_ni_syscall /* old break syscall holder */
21     .long sys_stat
22     .long sys_lseek
23     .long sys_getpid    /* 20 */

```

从上面的代码片段可知，系统调用表中的每一个表项均为长整型，在 32 位机器中为 4 个字节。这 4 个字节用于存储对应系统调用处理函数的入口地址。例如系统调用表的第 20 号表项用于保存系统调用处理函数 `sys_getpid()` 的入口地址，该函数入口地址的值在构建内核镜像 `vmlinux` 的过程中被确定，在生成 `vmlinux` 的过程中链接器进行符号表重定位时，链接器 `ld` 负责计算出符号表 `sys_getpid` 的地址、将该地址填充到了系统调用表 `sys_call_table` 的第 20 号表项中。

可以将系统调用表 `sys_call_table` 看作 C 语言中的数组，数组中的每一个元素保存了一个系统调用处理函数的入口地址，而系统调用号为该数组的下标。通过系统调用号可以获得相应系统调用处理函数的地址。系统调用号和系统调用表的表项具有一一对应的关系。

7.3.2 系统调用门的初始化

通过上一小节的分析，可知系统调用通过软中断来实现。在系统初始化过程中的异常初始化阶段，负责异常初始化的函数 `trap_init()` 通过下面的函数调用设置了系统调用入口函数为 `system_call()`。

```
set_system_gate(SYSCALL_VECTOR, &system_call);
```

其中，`SYSCALL_VECTOR` 在文件 `src/include/asm-i386/mach-default/irq_vectors.h` 中的第 31 行定义如下：

```
#define SYSCALL_VECTOR    0x80
```

由上可知系统调用占用了 `0x80` 号中断向量号。这里我们回顾一下函数 `set_system_gate()` 的功能，其代码如下：

```

static void __init set_system_gate(unsigned int n, void *addr)
{
    _set_gate(idt_table+n,15,3,addr,__KERNEL_CS);
}

```

对照“5.2.11A32 架构下的处理机制”中有关门描述符的内容，可知该函数设置第 n 号门描述符为一个陷阱门，并且可以从运行级别 3 来访问该陷阱门。该陷阱门处理函数所在的段选择子为 `__KERNEL_CS`、段内偏移量为 `addr`。关于更详细的中断、异常机制，请参见第 5 章。

7.3.3 系统调用的处理过程

这里以系统调用 `sys_getpid` 为例分析系统调用的处理过程。首先我们给出请求该系统调用的用户程序代码，看用户程序请求系统调用时需要做哪些准备工作；接着看看系统响应系统调用请求后具体做了哪些工作来为用户进程服务。

1. 请求系统调用

为了方便讨论，这里采用了内嵌汇编代码来请求系统调用。下面的代码用于访问系统调用号为 20 的系统调用 `sys_getpid()`，该系统调用用于返回当前进程的进程号。下面对这一段程序进行详细的分析。

```

-----
1  #include "stdio.h"
2  #define getpid_syscall_num 20
3
4  int main(int argc, char ** argv)
5  {
6      int pid;
7
8      asm ("int $0x80" \
9          : "=a"(pid) \
10         : "0"(getpid_syscall_num)\
11         );
12     printf("pid is %d\n",pid);
13     printf("pid is %d\n",getpid());
14     return(0);
15 }
-----

```

示例程序 7.4：使用内嵌汇编请求系统调用 `getpid`

第 1~2 行中，第 1 行代码包含的头文件 `stdio.h` 对随后使用的函数 `printf()`、`getpid()`

进行了类型声明，函数 `printf()` 在第 12、13 行用于向屏幕输出当前进程的进程号；函数 `getpid()` 是 `glibc` 对系统调用 `sys_getpid` 的封装，用于获取当前进程的进程号。第 2 行代码中使用的宏定义 `getpid_syscall_num` 的值为 20，即定义 `getpid_syscall_num` 的值为系统调用 `sys_getpid` 的系统调用号。关于系统调用号请参见 7.3.1 小节。

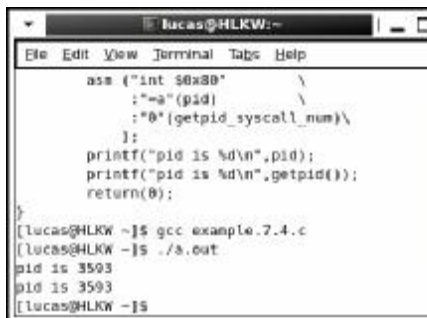
第 6、12 和 13 行中，第 6 行代码声明了用于保存进程号的整型局部变量 `pid`，由第 8~11 行的内嵌汇编代码向该变量中填充当前进程的进程号；第 12 行用于向屏幕打印当前进程的进程号。第 13 行调用 `glibc` 对系统调用 `sys_getpid` 的封装函数 `getpid()`，验证第 8~11 行内嵌汇编代码的正确性。

第 8~11 行，这段内嵌汇编使用软中断汇编指令 `int 0x80` 号中断，进入为系统调用设置的中断门。注意，内嵌汇编首先根据输出、输入列表设置了相关的上下文环境后，才执行语句模板中的汇编指令。

- I 输出列表。其中第 9 行代码说明了这段汇编的输出列表，即将运算结果保存到变量 `pid` 中，修饰字符串 “`=a`” 要求将变量 `pid` 与寄存器 `%eax` 关联起来，即运算过程中的临时结果会存放在寄存器 `%eax` 中，运算的最终结果才保存到变量 `pid` 中。这样减少了运算过程中的访存次数，提高了系统的效率；
- I 输入列表。其中第 10 行代码用于说明这段代码的输入列表，这里为 `getpid_syscall_num`、修饰字符 “`0`” 表明使用该变量的值初始化第 0 号占位符对应的变量，在这里就是使用变量 `getpid_syscall_num` 的值初始化寄存器 `%eax`。

也就是说，执行汇编指令 `int 0x80` 之前，寄存器 `%eax` 中已经保存了变量 `getpid_syscall_num` 的值，即系统调用 `sys_getpid` 的系统调用号。关于 GCC 内嵌汇编，请参见附录 C。

上面的程序编译、运行的结果如图 7.4 所示。由图中结果可知，上面使用的内嵌汇编代码成功地请求了系统调用 `sys_getpid`。那么执行软中断汇编指令 `int 0x80` 后，系统到底做了哪些工作？如何找到系统调用的处理程序？如何将系统调用的结果保存到变量 `pid` 中？请参见下面小节的详细分析。



```

lucas@HLKW:~$ gcc example_7.4.c
lucas@HLKW:~$ ./a.out
pid is 3593
pid is 3593
lucas@HLKW:~$

```

图 7.4 系统调用 `sys_getpid`

2. 处理系统调用

接下来分析系统调用的处理过程。执行 “`int 0x80`” 软中断指令后，系统进入中断处理周期，此时处理器的控制单元要完成一系列的权限检查，以及在权限检查通过的情况下完成最基本的现场保护；然后由系统调用处理程序负责进一步的处理。下面对系统调用处理过程的这两个阶段进行分析和讨论。

(1) 硬件处理阶段。

在这一阶段，控制单元负责进行中断门权限的检查。由于系统调用门被设置用来让用户进程访问，所以在权限检查时不会出现问题。同时由于当前运行级别 (3) 的权限低于系统调用处理函数运行级别 (0) 的权限，即发生了运行级别的切换，此时控制单元依次完成下面的操作。

- ① 处理器的控制单元首先将栈段选择子寄存器 `%ss`、栈指针寄存器 `%esp` 保存

的值保存到对外不可见的寄存器中（这些寄存器只供处理器自己使用，任何编程手段都无法访问这类寄存器）；然后从任务状态寄存器%tss 中获取当前进程的内核态栈信息来设置系统的栈选择子寄存器%ss 和栈指针寄存器%esp；再将保存到不可见寄存器中的%ss、%esp 保存到内核态栈中。

② 接下来处理器的控制单元将系统标志寄存器%eflags 的值保存到当前内核态栈的栈顶。

③ 最后，控制单元将用户态的指令段选择子寄存器%cs、指令指针寄存器%eip 保存到当前进程的内核态栈中，然后依据系统调用门描述符中处理函数所在的段选择子、段内偏移地址的值来设置系统的指令段选择子寄存器%cs、指令指针寄存器%eip。关于硬件处理的详情请参见 5.4.1 小节。

完成上面这些基本的保护之后，指令段选择子寄存器%cs、指令指针寄存器%eip 指向系统调用入口函数 `system_call()` 的地址。接下来由系统调用处理入口函数进行处理。

(2) 软件处理阶段。

在处理器的控制单元完成最基本的现场保护后，由系统软件做进一步的处理。这里我们从系统调用入口函数 `system_call` 入手开始分析。

代码清单 7.3——函数 `system_call`

功能简介：该函数是系统调用软中断的入口函数，用于根据系统调用号分发系统调用、调用相应的系统调用处理函数。

文件：`src/arch/i386/kernel/entry.S`

```

225 ENTRY(system_call)
226     pushl %eax          # save orig_eax
227     SAVE_ALL
228     GET_THREAD_INFO(%ebp)
229                                     # system call tracing in operation / emulation
230     /* Note, _TIF_SECCOMP is bit number 8, and so it needs testw
and not testb */
231     testw $(_TIF_SYSCALL_EMU|_TIF_SYSCALL_TRACE \
                |_TIF_SECCOMP|_TIF_SYSCALL_AUDIT),TI_flags(%ebp)
232     jnz syscall_trace_entry
233     cmpl $(nr_syscalls), %eax
234     jae syscall_badsys
235 syscall_call:
236     call *sys_call_table(,%eax,4)
237     movl %eax,EAX(%esp)      # store the return value
238 syscall_exit:
239     cli                    # make sure we don't miss an interrupt
240                                     # setting need_resched or sigpending
241                                     # between sampling and the iret

```

```

242     movl TI_flags(%ebp), %ecx
243     testw $_TIF_ALLWORK_MASK, %cx # current->work
244     jne syscall_exit_work
245
246 restore_all:
247     movl EFLAGS(%esp), %eax      # mix EFLAGS, SS and CS
248     # Warning: OLDSS(%esp) contains the wrong/random values if we
249     # are returning to the kernel.
250     # See comments in process.c:copy_thread() for details.
251     movb OLDSS(%esp), %ah
252     movb CS(%esp), %al
253     andl $(VM_MASK | (4 << 8) | 3), %eax
254     cmpl $((4 << 8) | 3), %eax
255     je  ldt_ss      # returning to user-space with LDT SS
256 restore_nocheck:
257     RESTORE_REGS
258     addl $4, %esp
259 1:   iret

```

第 226 行代码将寄存器 `%eax` 的值保存到当前进程内核态栈的栈顶。由上面的讨论可知寄存器 `%eax` 保存了被请求系统调用的系统调用号。

第 227 行代码调用宏定义 `SAVE_ALL` 来保存现场。关于该宏定义的详细分析请参见 5.4.2 小节。使用该宏定义保存完现场后，当前进程的内核态栈的内容如图 7.5 所示。

第 228 行代码调用宏定义 `GET_THREAD_INFO` 获取当前进程的进程描述符中的成员变量 `thread_info` 的地址，并将该地址保存到寄存器 `%ebp` 中。该宏定义在文件 `src/include/asm-i386/thread_info.h` 中的第 120 行开始定义，代码如下。关于该宏定义的工作原理，请参见 4.2.4 小节。

```

#define GET_THREAD_INFO(reg)      \
    movl $-THREAD_SIZE, reg;      \
    andl %esp, reg

```

第 231~232 行代码判断当前进程描述符中成员变量 `thread_info` 中的当前进程状态 `flags` 是否设置了这些标记位。这些标记位的含义请参见 4.2.4 小节。如果设置了这些标记位，则跳转到标号 `syscall_trace_entry` 指示的代码段执行；否则顺序执行。这里我们假设没有设置这些标记位。

第 233~234 行代码首先判断传递进来的系统调用号是否合法，即是否超出了内核实现的系统调用的最大系统调用号。如果大于，说明系统调用号不合法，跳转到标号 `syscall_badsys` 指示的代码段进行出错处理；否则顺序执行。这里我们假设传递进来的

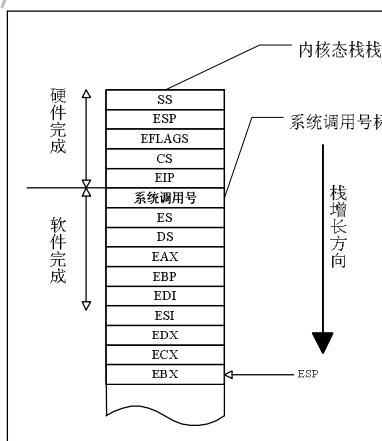


图 7.5 系统调用处理时的内核态栈

系统调用号合法，即顺序执行。

第 235~236 行代码根据系统调用号（保存在寄存器%eax 中）调用具体的系统调用处理函数，该函数的地址保存在存储单元"sys_call_table+4*%eax"中。注意，这里 %eax 的值为 20，也就是调用系统调用处理函数 sys_getpid()。关于该函数的具体功能，请参见对该函数的详细分析。

第 237 行代码负责将系统调用处理函数保存在寄存器%eax 中的返回值移动到内核态栈相应的位置中。系统调用处理完毕、进行系统调用返回时，负责进行现场恢复的宏定义 RESTORE_REGS 会将保存在内核态栈中的该返回值填充到寄存器%eax 中，这就是 7.3.3 中的“请求系统调用”所给的内嵌汇编代码中通过将变量 pid 和寄存器 %eax 相关联，即可获得系统调用返回值的原因。

第 238 行代码开始进行系统调用的退出处理工作。其中 239 行使用汇编指令 cli 关闭中断，因为系统调用门是一个陷阱门，与中断门不一样，在通过陷阱门进入内核态时处理器并不会自动关中断，在进入中断处理到此中间系统可以响应外设的中断；同时也说明，从这里开始直到返回到用户态之前，系统不会再响应外设中断。

第 242~244 行代码用于判断在系统调用处理过程中，当前进程是否被设置了宏定义 _TIF_ ALLWORK_MASK 所代表的事件标记。该宏定义在文件 src/include/asm-i386/thread_info.h 中的第 164 行定义如下：

```
#define _TIF_ALLWORK_MASK (0x0000FFFF & ~_TIF_SECCOMP)
```

其中，_TIF_SECCOMP 对应于标记 TIF_SECCOMP，其值在同一文件中的第 156 行定义如下：

```
#define _TIF_SECCOMP (1<<TIF_SECCOMP)
```

由上可知，该宏定义包含了除 TIF_SECCOMP 以外的进程状态标记信息：在有这些标记时，在返回用户态之前需要作进一步的处理，因为当前进程还有一些需要在内核态处理的事务。关于进程状态标记信息，请参见 4.2.4 小节。

第 246~255 行代码判断请求此次系统调用进程的用户态的栈选择子寄存器%ss 是否指向了一个局部段描述符表 LDT 中的段描述符，且系统不处于虚拟 8086 工作模式下，并请求此次系统调用进程的运行级别为 3，即为用户进程（内核态也可以请求系统调用）。在这种情况下，需要执行代码段对局部描述符表作特殊的处理，这里不做详细的分析。

第 256~259 行代码进行现场的恢复工作，第 257 行中宏定义 RESTORE_REGS 的功能与宏定义 SAVE_ALL 恰好相反，用于将保存在内核态的现场信息还原到相应的寄存器中，完成现场的恢复工作。第 258 行用于将栈定中的元素移出，该元素即系统调用号。第 259 行使用指令 iret 进行中断的返回，该指令以当前栈中的元素按出栈顺序设置系统的 %eip、%cs、%eflags、%esp、%ss 寄存器。到此为止，系统完成了一次系统调用处理，返回用户态继续运行。

代码清单 7.4——函数 sys_getpid

功能简介：该系统调用处理函数用于返回当前进程的进程号。

文件：src/kernel/timer.c

```
942 asmlinkage long sys_getpid(void)
```

```

943 {
944     return current->tgid;
945 }

```

该系统调用处理函数非常简单，直接返回当前进程的 `tgid`，即当前进程的线程组号。Linux 在 IA32 体系结构下的应用二进制接口（Application Binary Interface，简称 ABI）规范规定函数的返回值保存到寄存器 `eax` 中，即当前进程的线程组号被保存到了寄存器 `%eax` 中。读者可能会问，为什么不是返回当前进程的 `pid`？答案请参考 4.2.1 小节。

7.4

系统调用的参数传递

在普通的函数调用过程中（包括内核态内部、用户态内部的函数调用），调用函数首先将实际参数保存到当前运行级别对应的栈中，然后使用指令 `call` 来调用被调用函数。被调用函数首先从当前的栈中取得相应的参数值，再进行相应的处理。与普通的函数调用不同，系统调用跨越了两个不同的运行级别，这两个运行级别分别有自己独立的栈空间。通过上一节的学习可知：内核设计时为了保护系统核心不遭受用户进程有意或无意的破坏，进行了大量的隔离处理，使得用户空间不能随意访问内核空间，也就无法使用内核栈向运行于内核态的系统调用处理函数传递参数；但系统调用处理函数和普通的被调用函数一样，需要参数传递和返回值保存。本节就这些问题进行讨论，分析在系统调用中用户态进程和内核态如何进行参数传递、数据交互。

7.4.1 少量参数的情况

在系统调用需要传递少量参数的情况下（在 IA32 体系结构下，不多于 6 个 32 位的整型数据），通常采用处理器中的寄存器完成参数的传递工作。系统调用过程中所需要的参数由寄存器 `%eax`、`%ebx`、`%ecx`、`%edx`、`%esi` 和 `%edi` 来传递。事实上，系统调用过程中系统调用号就是通过处理器中的寄存器 `%eax` 进行传递的，系统调用入口函数 `system_call()` 根据保存在该寄存器中的不同系统调用号，调用相应的系统调用处理函数来完成具体的系统调用。

具体的系统调用处理函数所需参数依次由除了 `%eax` 外剩余的 5 个寄存器：`%ebx`、`%ecx`、`%edx`、`%esi` 和 `%edi` 来传递。在系统调用处理过程中，这些保存在处理器寄存器中的参数被系统调用入口函数 `system_call()` 压入到内核态栈中，然后该入口函数通过指令 `call` 调用相应的系统调用处理函数进行处理。具体的系统调用处理函数从内核态栈中取得相应的参数值，完成具体的处理工作。

7.4.2 大量参数的情况

当系统调用需要在用户态和内核态传递大量数据时，处理器中内置的寄存器显然不够使用。那么此时如何进行处理呢？由于大量的数据通常存储在一个数据块中，此时采用一个指向该数据块的指针和该数据块长度的值来描述这一参数，再和传递少量数据的情况一样将这两个通过处理器中的参数进行传递。

到此为止好像问题都已经解决了，但是与上一小节中传递的每一参数都代表一个数值的情况不同，此时传递的某些参数值代表了一个地址，且在系统调用处理过程中通常要从该地址读数据或者向该地址写数据。这时就需要做一些参数验证，因为如果用户进程在系统调用时给了一个非法地址，而内核态的处理函数不做任何验证、直接向该地址中写数据会造成系统漏洞，危害系统的安全。

内核中，使用宏定义 `access_ok()` 判断当前的用户进程是否有权限对一个内存区域进行操作。内核态中经常使用的用于访问进程用户态空间的函数 `copy_from_user()`、`copy_to_user()` 中就使用该宏定义来做参数的合法性验证。该宏定义在文件 `src/include/asm-i386/uaccess.h` 中的第 84 行定义如下：

```
#define access_ok(type,addr,size) (likely(__range_ok(addr,size)
== 0))
```

该宏定义用于判断当前进程是否拥有访问地址空间 `[addr,addr+size]` 的权限，具体判断过程由宏定义 `__range_ok()` 进行处理。下面请看对宏定义 `__range_ok()` 的详细分析。

代码清单 7.5——宏定义 `__range_ok`

功能简介：该宏定义用于判断当前进程是否对某一个地址空间有访问权限。

文件：`src/include/asm-i386/uaccess.h`

```
57 #define __range_ok(addr,size) ({
\
58     unsigned long flag,sum;
\
59     __chk_user_ptr(addr);
\
60     asm("addl %3,%1 ; sbb1 %0,%0; cmpl %1,%4; sbb1 $0,%0"
\
61         : "=&r" (flag), "=r" (sum)
\
62         : "1" (addr), "g"((int)(size)), "g"(current_thread_info()
->addr_limit.seg)); \
63     flag; })
```

第 58 行代码声明该宏定义使用的局部变量。其中变量 `flag` 用来存储该宏定义的最终结果，变量 `sum` 保存了参数 `addr` 和 `size` 的和以及这段地址空间的最大值。

第 59 行中的宏定义 `__chk_user_ptr()` 用于判断当前进程是否有访问地址 `addr` 的权限。该宏定义在没有选中选项 `__CHECKER__` 的情况下为空，不做处理。

第 60~62 行的内嵌汇编用于计算出 `addr+size` 的值并将该值与当前进程中的 `thread_info` 数据结构保存的进程地址上限作比较，如果小于该上限，设置 `flag` 的值为 0，表示有对该地址空间的访问权限；否则设置 `flag` 的值为一个不等于 0 的常数，表示没有对该地址空间的访问权限。

第 63 行的用法比较特殊，是 GUN C 对标准 C 的一种扩展：通过用圆括号 “()” 将整个复合语句括起来时，可以将复合语句中最后一条表达式的值当做整个复合语句的值。这行代码作为复合语句中的最后一条语句，修改复合语句的值为 `flag`，即将 `flag` 的值作为整个宏定义的结果。

除了上面提及的 `copy_from_user()`、`copy_to_user()` 两个函数，内核中它提供了其他几个函数用于访问用户态空间，这些函数都采用了相应的验证机制，确保参数的合

法性。限于篇幅，这里不做详细介绍。

7.5

如何添加新系统调用

本节介绍如何向系统中添加一个新的系统调用，新添加的系统调用名称为 **HLKW** (how Linux kernel works 的简称)，用来返回系统当前 **jiffies** 的值。下面对添加系统调用的每一步骤进行说明，最后给出测试代码，对新添加系统调用进行测试和验证。

7.5.1 前期准备工作

在为一个新的系统调用编写具体的处理函数之前，需要做一些前期的准备工作，包括为新添加的系统调用分配系统调用号、在系统调用表中为其设置系统调用表表项等。

1. 修改系统调用表

每一个系统调用都在系统调用表中占用一个表项，系统调用表位于文件 `src/arch/i386/kernel/syscall_table.S` 中。为了添加一个新的系统调用，需要为将要添加的系统调用在系统调用表中分配一个表项，这里我们将新的系统调用添加到该表的最后。修改后的系统调用表内容如下。

代码清单 7.6——设置系统调用表表项

功能简介：添加新系统调用后的系统调用表。

文件：`src/arch/i386/kernel/syscall_table.S`

```

1  .data
2  ENTRY(sys_call_table)
3      .long sys_restart_syscall /* 0 - old "setup()" system call, used for
restarting */
4      .long sys_exit
5      .long sys_fork
6      .long sys_read
7      .long sys_write
8      .long sys_open      /* 5 */
    略去部分不相关代码。
290     .long sys_request_key
291     .long sys_keyctl
292     .long sys_ioprio_set
293     .long sys_ioprio_get      /* 290 */
294     .long sys_inotify_init
295     .long sys_inotify_add_watch

```

```
296     .long sys_inotify_rm_watch
297     .long sys_HLKW
```

由上可知，新添加的系统调用在系统调用表中占用了第 294 号的位置，即为新添加的系统调用分配的系统调用号为 294。在编写测试代码时，我们将通过该系统调用号来访问新添加的系统调用。

2. 添加系统调用号

在系统调用表中为新的系统调用设置了表项之后，需要将与该表项对应的系统调用号添加到系统调用的头文件 `src/include/asm/unistd.h` 中，这样其他模块才能使用该系统调用。参见下面修改过的代码段。

代码清单 7.7——设置系统调用号

功能简介：添加新系统调用号后的头文件 `unistd.h`。

文件：`src/include/asm-i386/unistd.h`

```
297 #define __NR_ioprio_set          289
298 #define __NR_ioprio_get          290
299 #define __NR_inotify_init        291
300 #define __NR_inotify_add_watch   292
301 #define __NR_inotify_rm_watch    293
302 #define __NR_HLKW                294
303
304 #define NR_syscalls 295
```

7.5.2 添加处理函数

在完成前面的前期准备之后，接下来需要找一个合适地方编写新添加系统调用的处理函数。原则上该函数可以在任何一个现有的内核源代码中实现，也可以新建一个内核源文件来实现新添加的系统调用处理函数。考虑到该系统调用与时间相关，我们就把该系统调用添加到文件 `src/kernel/time.c` 的末尾，参见下面的代码段。

代码清单 7.8——新添加系统调用的处理函数

功能简介：系统调用处理函数 `sys_HLKW` 的源代码。

文件：`src/kernel/time.c`

```
594     ret = jiffies_64;
595     } while (read_seqretry(&time_lock, seq));
596     return ret;
597 }
598
599 EXPORT_SYMBOL(get_jiffies_64);
600 #endif
601
602 EXPORT_SYMBOL(jiffies);
603
604 asmlinkage long sys_HLKW(void)
605 {
```

```
606     return jiffies;  
607 }
```

到此为止，就添加好了一个系统调用。接下来我们需要编译、安装新的内核，并使用新的内核引导系统，然后就可以测试新添加的系统调用了。

华清远见

7.5.3 测试新系统调用

这里我们编写一个测试程序来测试新添加的系统调用。通过对“7.2 系统调用的访问手段”的学习，我们知道访问一个系统调用有 3 种方法，但这里可以使用的只有两种，因为 C 库中没有新添加系统调用的相应封装函数，只能采用其他两种方法。下面使用这两种方法分别进行测试。

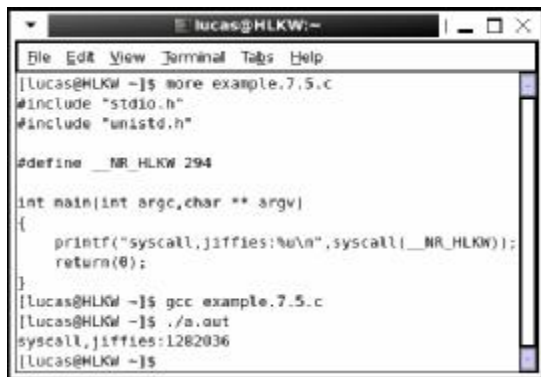
1. 使用通用接口

使用通用接口函数 `syscall()` 的测试代码如示例程序 7.5 所示。注意：这里包含的头文件 `unistd.h` 是 `libc` 提供的标准头文件，该头文件包含了通用接口函数 `syscall()` 的声明。

```
-----  
-----  
#include "stdio.h"  
#include "unistd.h"  
  
#define __NR_HLKW 294  
  
int main(int argc, char ** argv)  
{  
    printf("syscall, jiffies: %u\n", syscall(__NR_HLKW));  
    return(0);  
}  
-----  
-----
```

示例程序 7.5：使用通用接口测试新系统调用

编译、运行的结果如图 7.6 所示。



```
lucas@HLKW:~  
File Edit View Terminal Tabs Help  
[lucas@HLKW ~]$ more example.7.5.c  
#include "stdio.h"  
#include "unistd.h"  
  
#define __NR_HLKW 294  
  
int main(int argc, char ** argv)  
{  
    printf("syscall, jiffies: %u\n", syscall(__NR_HLKW));  
    return(0);  
}  
[lucas@HLKW ~]$ gcc example.7.5.c  
[lucas@HLKW ~]$ ./a.out  
syscall, jiffies: 1282036  
[lucas@HLKW ~]$
```

图 7.6 使用通用接口测试新添加的系统调用

2. 使用内嵌汇编

使用内嵌汇编的测试代码如示例程序 7.6 所示。这里使用的头文件是内核源代码种的头文件，此头文件包含了宏定义 `_syscall0()` 的声明。

```

1      #include "stdio.h"
2      /*unistd.h must be the curent kernel`s header file*/
3      #include "/home/lucas/Linux-2.6.15.5/include/asm/unistd.h"
4
5      static int errno;
6      _syscall0(long,HLKW);
7
8      int main(int argc,char ** argv)
9      {
10         printf("macro ,jiffies:%u\n",HLKW());
11         return(0);
12     }

```

示例程序 7.6: 使用内嵌汇编测试新系统调用

编译、运行的结果如图 7.7 所示。



```

lucas@HLKW:~$ gcc -D _KERNEL__ example.7.6.c
lucas@HLKW ~]$ ./a.out
macro ,jiffies:1304451
lucas@HLKW ~]$

```

图 7.7 通过内嵌汇编测试新添加的系统调用

7.6

什么是快速系统调用

通过“第 5 章中断和异常”的学习可知，在产生一个中断（广义的中断，包括硬

中断、软中断、异常)后,处理器中控制器会进行一系列权限检查,只有得到核实后,控制单元才设置中断处理所需要的执行环境。基于软中断 `int 0x80` 的系统调用同样也需要作一系列的检查,才能进入内核态进行系统调用的处理工作。

由于系统调用的特点,这一系列权限检查变得多余。为此 Intel 在 Pentium II 处理器中引入了一对新的指令 `SYSENTER/SYSEXIT`,用于实现快速系统调用;其中指令 `SYSENTER` 用于进入内核态,而指令 `SYSEXIT` 用于从内核态返回。这两条指令避免了权限检查,直接将处理器置为预定义的运行级别(基于软中断的系统调用需要从门描述符中获得运行级别信息)。同时,通过将系统调用所需的执行环境信息保存在一组型号相关寄存器(Model Specific Register,简称 MSR)中,避免了访问内存,进一步提高进入内核态的速度。下面分小节对快速系统调用进行全面的介绍和分析。

7.6.1 工作机制

与指令对 `INT/IRET` 不同,快速系统调用指令对 `SYSENTER/SYSEXIT` 不具有调用、返回关系,因为指令 `SYSENTER` 并不会为指令 `SYSEXIT` 保存任何返回信息,不指示 `SYSEXIT` 返回到何处继续执行,也就是说指令 `SYSEXIT` 并不一定返回到指令 `SYSENTER` 后的下一个条指令继续执行。这两条指令所需的相关信息由处理器内部的一组相关的寄存器(MSR)提供,这些寄存器的名称及用途如下。

- I **MSR_IA32_SYSENTER_CS**: 保存了系统调用处理过程所使用的内核态代码段的段选择子,用于在通过指令 `SYSENTER` 进入内核态时,设置代码段选择子寄存器%cs。同时,紧随在该寄存器所指示段描述符后面的三个段描述符被依次认为是内核数据段、用户代码段、用户数据段的段描述符(这些段描述符保存在全局描述符表 GDT 中,有先后次序)。快速系统调用指令 `SYSENTER/SYSEXIT` 依赖这些次序来完成内核态、用户态执行环境的设置工作。
- I **MSR_IA32_SYSENTER_EIP**: 保存了系统调用处理过程的入口地址,用于在通过指令 `SYSENTER` 进入内核态时,设置指令指针寄存器%eip。
- I **MSR_IA32_SYSENTER_ESP**: 保存了系统调用处理过程所使用内核态的栈指针信息,用于在通过指令 `SYSENTER` 进入内核态时,设置内核态栈的栈指针寄存器%esp。

1. 请求快速系统调用处理过程

在用户态的代码执行了 `SYSENTER` 指令之后,处理器中的控制单元将会完成以下操作,然后进入内核态进行系统调用的处理。

- (1) 将寄存器 `MSR_IA32_SYSENTER_CS` 所指示的段描述符装载到代码段选择子寄存器%cs 中。
- (2) 将寄存器 `MSR_IA32_SYSENTER_EIP` 的值装载到指令指针寄存器%eip 中。
- (3) 将寄存器 `MSR_IA32_SYSENTER_CS` 的值加 8 作为一个段选择子,然后将该段选择子对应的段描述符装载到栈基址寄存器%ss 中。
- (4) 将寄存器 `MSR_IA32_SYSENTER_ESP` 的值装载到栈指针寄存器%esp 中。

2. 快速系统调用返回处理过程

在内核态对系统调用服务完毕之后，执行指令 `SYSEXIT` 完成系统调用的返回过程。在该过程中处理器的控制单元完成以下处理，设置用户态的执行环境。

(1) 将寄存器 `MSR_IA32_SYSENTER_CS` 的值加 16 作为一个段选择子，然后将该段选择子对应的段描述符装载到代码段段选择子寄存器 `%cs` 中。

(2) 将寄存器 `%edx` 的值装载到指令指针寄存器 `%eip` 中。

(3) 将寄存器 `MSR_IA32_SYSENTER_CS` 的值加 16 作为一个段选择子，然后将该段选择子对应的段描述符装载到栈段段选择子寄存器 `%ss` 中。

(4) 将寄存器 `%ecx` 的值装载到栈指针寄存器 `%esp` 中。

由上面的分析可知，在使用指令 `SYSENTER` 请求快速系统调用之前，需要初始化好相关的型号相关寄存器；在使用指令 `SYSEXIT` 进行快速系统调用返回之前，还要保证寄存器 `%ecx`、`%edx` 的正确性，以便能正确地返回到用户态继续运行。在内核的初始化过程中，函数 `enable_sep_cpu()` 负责完成这组寄存器的初始化，该函数的调用关系如图 7.8 所示。在系统初始化或系统休眠恢复过程中，该函数会被系统中的每一个处理调用，完成本地处理器上快速系统调用的初始化。具体处理过程，请参见下面对该函数的详细分析。

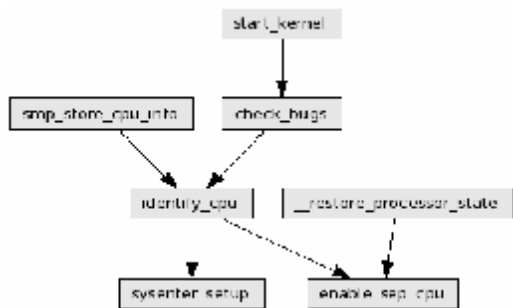


图 7.8 快速系统调用初始化调用关系框图

代码清单 7.9——函数 enable_sep_cpu

功能简介：该函数用于设置快速系统调用相关的几个寄存器，用于指定快速系统调用时指令段、栈指针以及指令指针的值。

文件：src/arch/i386/kernel/sysenter.c

```

24 void enable_sep_cpu(void)
25 {
26     int cpu = get_cpu();
27     struct tss_struct *tss = &per_cpu(init_tss, cpu);
28
29     if (!boot_cpu_has(X86_FEATURE_SEP)) {
30         put_cpu();
31         return;
32     }
33
34     tss->ss1 = __KERNEL_CS;
35     tss->esp1 = sizeof(struct tss_struct) + (unsigned long) tss;
36     wrmsr(MSR_IA32_SYSENTER_CS, __KERNEL_CS, 0);
37     wrmsr(MSR_IA32_SYSENTER_ESP, tss->esp1, 0);
38     wrmsr(MSR_IA32_SYSENTER_EIP, (unsigned long) sysenter_entry,
0);
39     put_cpu();
40 }

```

第 26、27 和 39 行中，前两行代码用于获得当前处理器对应的任务状态段的地址，并将该地址保存在局部变量 tss 中。其中，第 26 行调用的宏定义 get_cpu() 首先禁用内核态抢占，然后返回当前处理器在系统中的编号；第 27 行中的宏定义 per_cpu() 用于获得每处理器变量的本地拷贝。第 39 行中宏定义 put_cpu() 与第 26 行的操作相对，用于使能内核态抢占。关于内核态抢占，请参见 8.1.3 小节。

第 29~32 行代码判断负责系统引导的处理（BootSstrap Processor）是否支持快速系统调用指令 SYSENTER/SYSEXIT，如果不支持，则调用宏定义 put() 使能内核态抢

占，然后返回。也就是说，只有在系统引导处理器支持快速系统调用的情况下，才为系统中的每个处理器设置快速系统调用所需要的参数；否则，系统中不支持快速系统调用。

第 34~35 行代码设置当前任务状态段中的成员变量 `ssl`、`esp1` 分别为快速系统调用所使用的代码段段选择子（内核代码段 `__KERNEL_CS`）和快速系统调用使用的临时（紧急）内核态栈的栈底。这一内核态栈保存在任务状态段数据结构中，共有 64×4 字节。详情请参见 4.2.3 小节。

第 36~38 行代码调用宏定义 `wrmsr()` 设置快速系统调用所需的 3 个 MSR 寄存器。分别设置 `MSR_IA32_SYSENTER_CS` 为内核代码段段选择子 `__KERNEL_CS`；设置 `MSR_IA32_SYSENTER_ESP` 为任务状态段中预留的临时（紧急）内核态栈的栈底地址；设置 `MSR_IA32_SYSENTER_EIP` 为快速系统调用处理函数的入口地址 `sysenter_entry`。

7.6.2 实现策略

为了保证向上兼容，内核首先保留了原有的基于软中断的 `int 0x80` 系统调用处理方式，这样在 `glibc` 库函数没有实现快速系统调用的封装函数时，系统仍可正常工作。同时，在快速系统调用的实现上内核做了大量的工作，实现了一个统一的调用接口。在系统支持快速系统调用的情况下，接口后面的处理函数基于快速系统调用指令 `SYSENTER/SYSEXIT` 来实现；在系统不支持快速系统调用的情况下，接口后面的处理函数使用软中断 `init0x80` 来实现。这样使得 `glibc` 可以使用统一的接口来访问两种不同的系统调用而觉察不到任何差别，避免了 `glibc` 维护两套不同的系统调用封装函数。

对比 Windows 的实现策略，虽然 XP 实现了基于指令 `SYSENTER/SYSEXIT` 的快速系统调用，但同时也完全废除了原有的基于软中断的系统调用机制，这也是 windows XP 最低要求处理器为 Pentium II 的原因，即 Windows XP 不再支持老型号的处理器。而 2.6 的 Linux 内核则没有此限制，同时支持所有型号的 IA32 体系结构的处理器。下面对 Linux 内核的实现策略进行讨论，分析其如何实现上述特性。

为讨论方便，这里首先简单介绍一下程序库的概念。根据使用方法和时机不同，通常可以将库分为以下三种。

- 1 静态库 (Static Library)：该类型的库本质上是目标代码 (Object Code) 的集合，在程序编译过程中，用于向最终生成的可执行程序提供所需的目标代码（这些目标代码段被链接到可执行程序之中）。
- 1 共享库 (Shared Library)：也称为共享目标代码 (Share Object, 简称 so)，因此英文中称共享库名称为 `soname` (Share Object NAME)。共享库名称 `soname` 并不是共享库的文件名，请读者予以区分。该类型的库并不编译到可执行程序之中，而是在程序被装入时，装载程序负责将其所需的共享库映射到进程的地址空间中，然后程序才可以运行。该类型的库可被系统中多个进程同时共享（在系统内存只有一份拷贝）使用，其在不同进程地址空间中的位置不一样，所以共享库是位置无关代码 (Position Independent Code, 简称 PIC)。
- 1 动态装载库 (Dynamically Loaded Library)：该类型的库本质上是共享库，区别在于其使用方法和时机。只有在一个进程运行过程中需要时，才显式地使用函数 `dlopen()` 打开一个动态链接库，然后使用该库函数中提供的例程。动

态装载库到进程地址空间映射关系不是在程序转入运行时建立的，而是在需要时才建立起来的。在使用完动态装载库后，需要显式地使用函数 `dlclose()` 关闭该动态库。

1. 内核方面

为了在统一的接口中实现两种不同的系统调用处理方式，内核采用了共享库的方法，为这两种系统调用处理方式分别提供了一个共享库。这两个共享库分别由目录 `src/arch/i386/kernel/` 中的文件 `vsyscall-sysenter.S`、`vsyscall-int80.S` 生成；它们的入口函数名称都是 `__kernel_vsyscall`，在编译的过程中都使用了编译标志 `vsyscall-flags`，该标志在文件 `src/arch/i386/kernel/Makfile` 中的第 56 行定义如下：

```
vsyscall-flags = -shared -s -Wl,-soname=Linux-gate.so.1
```

也就是说，这两个共享库的名称 `soname` 都为 `Linux-gate.so.1`。随后我们将从用户空间读出这一共享库，获得共享库名称，以对我们的思路加以验证。验证的结果，请参见图 7.9。

```
lucas@HLKW ~]$ ldd /usr/bin/who
        linux-gate.so.1 => (0x00c84000)
        libc.so.6 => /lib/libc.so.6 (0x0083a000)
        /lib/ld-linux.so.2 (0x00818000)
[lucas@HLKW ~]$ ldd /bin/cat
        linux-gate.so.1 => (0x00183000)
        libc.so.6 => /lib/libc.so.6 (0x0083a000)
        /lib/ld-linux.so.2 (0x00818000)
[lucas@HLKW ~]$ ldd /bin/cat
        linux-gate.so.1 => (0x008f8000)
        libc.so.6 => /lib/libc.so.6 (0x00118000)
        /lib/ld-linux.so.2 (0x00818000)
[lucas@HLKW ~]$
```

图 7.9 虚拟动态共享库 `Linux-gate.so.1`

在系统初始化过程中，内核负责判断系统引导处理器（Bootstrap Processor）是否支持快速系统调用指令，如果支持，将共享库 `vsyscall-sysenter.so` 装载到固定映射页面中；否则，将共享库 `vsyscall-int80.so` 装载到固定映射页面中，该固定映射页面由枚举元素 `FIX_VSYSCALL` 指定。关于该固定映射页面，请参见 3.6.2 小节。该过程的具体初始化由函数 `sysenter_setup()` 完成，其在系统初始化过程中的调用路径请参见图 7.8。具体初始化步骤，请看下面对该函数的详细分析。

代码清单 7.10——函数 `sysenter_setup`

功能简介：该函数根据处理器是否支持快速系统调用，分别向快速系统调用虚拟页面空间设置所需的共享库。

文件：`src/arch/i386/kernel/sysenter.c`

```
49 int __init sysenter_setup(void)
50 {
51     void *page = (void *)get_zeroed_page(GFP_ATOMIC);
52
```

```

53     __set_fixmap(FIX_VSYSCALL, __pa(page), PAGE_READONLY_EXEC);
54
55     if (!boot_cpu_has(X86_FEATURE_SEP)) {
56         memcpy(page,
57             &vsyscall_int80_start,
58             &vsyscall_int80_end - &vsyscall_int80_start);
59         return 0;
60     }
61
62     memcpy(page,
63         &vsyscall_sysenter_start,
64         &vsyscall_sysenter_end - &vsyscall_sysenter_start);
65
66     return 0;
67 }

```

第 51~53 行中，第 51 行调用接口函数 `get_zeroed_page()` 获得了一个被填充为 0 的物理页框，返回了该页框在内核地址空间的线性地址并将其保存在变量 `page` 中。第 53 行将申请到的该物理页框映射到了固定映射地址中 `FIX_VSYSCALL` 对应的虚拟页面中，并设置对应的页表项属性为宏定义 `PAGE_READONLY_EXEC` 的值。该宏定义在文件 `src/include/asm-i386/pgtable.h` 中的第 152 行开始定义，代码如下：

```

#define PAGE_READONLY_EXEC \
    __pgprot(_PAGE_PRESENT | _PAGE_USER | _PAGE_ACCESSED)

```

由上可知，用户进程具有访问该虚拟页面的权限，具有只读和可执行权限（因为没有设置 `_PAGE_RW` 标记）。关于固定映射地址空间，请参见 3.6.2 小节；关于宏定义 `__pa()`，请参见 3.6.1 小节。

第 55~66 行中，第 55~60 行代码在处理器不支持快速系统调用指令 `SYSENTER/SYSEXIT` 的情况下，将标记 `vsyscall_int80_start`、`vsyscall_int80_end` 之间的内容拷贝到上面申请的物理页框中；第 62~66 行代码在处理器支持快速系统调用指令的情况下，将标记 `vsyscall_sysenter_start`、`vsyscall_sysenter_end` 之间的内容拷贝到上面申请的物理页框中。这 4 个标记在文件 `src/arch/i386/kernel/vsyscall.S` 中的第 5 行开始声明，代码如下：

```

.globl vsyscall_int80_start, vsyscall_int80_end
vsyscall_int80_start:
    .incbin "arch/i386/kernel/vsyscall-int80.so"
vsyscall_int80_end:

.globl vsyscall_sysenter_start, vsyscall_sysenter_end
vsyscall_sysenter_start:
    .incbin "arch/i386/kernel/vsyscall-sysenter.so"
vsyscall_sysenter_end:

```

其中，伪指令 `incbin` 用于将一个文件中的内容逐字地读取（包含）到当前位置，可知该文件容纳了这两个不同的共享库，这 4 个全局可见标记分别记录了两个共享库的开头和结尾位置信息。

通过该函数的初始化，4GB 的线性地址空间中的倒数第二个虚拟页面被初始化为用户进程可访问的页面（倒数第一个虚拟页面保留，没有使用），该页面中包含了一个共享库。通常称该页面中包含的共享库为虚拟动态共享库（Virtual Dynamic Share Object，简称 VDSO），因为该共享库不是以一个文件的格式存在于文件系统中，而是

由内核提供的，对用户进程而言是虚拟的。

该页面中包含共享库的内容依赖于处理器是否支持快速系统调用指令，在处理器支持快速系统调用指令的情况下，共享库为 `vsyscall-sysenter.so`；否则，共享库为 `vsyscall-int80.so`。这样内核为用户空间提供了一个统一接口的共享库，虽然库函数的实现依赖于处理器，但提供的接口是一样的。用户空间可以通过统一的接口来访问系统调用，而不必关心系统调用是用软中断还是用快速系统调用指令实现。

2. 用户方面

谈到系统调用的完成，不能孤立地只看内核代码。我们知道，系统调用通常被封装成库函数提供给用户程序调用，用户程序调用库函数后，由 `glibc` 库负责调用内核提供的系统调用接口函数。由于内核方面做了大量的工作，使得 `glibc` 无需考虑系统调用方式，减小了 `glibc` 代码的复杂性，只需将原有的“`int $0x80`”软中断指令替换成“`call 入口地址`”形式指令即可。

当一个可执行程序被装入、运行时（通过系统调用 `exec`），内核首先判断该可执行文件的格式；然后调用对应的装载程序对该可执行文件进行分析，映射代码段、数据段到合适的地址空间，同时分析该可执行程序依赖的共享库，将所需的共享库也映射到进程的地址空间；最后设置程序的入口点以及进程的堆栈信息。经过这一番的初始化之后，可执行程序才可以运行。

在支持快速系统调用的系统中，所有可执行程序（除了那些静态链接的可执行程序）都依赖一个共享库名称为 `Linux-gate.so.1` 的库，图显示了这种依赖关系。从图中可以看出，并没有实际的文件与该共享库相对应，该共享库即为内核中向用户进程提供的虚拟动态共享库 `VDSO`，该共享库的名称来源于共享代码编译过程指定的参数 `vsyscall-flags`。细心的读者可以看出，该虚拟动态共享库映射到进程的地址空间是不一样的，这是因为内核采用了随机地址空间映射办法，该办法可以将进程重要的地址空间隐蔽起来，增加攻击者攻击的难度。

那么 `glibc` 封装函数如何知道虚拟动态共享库 `Linux-gate.so.1` 在进程地址空间中的地址呢？如何才能跳转到该地址、执行相应的指令进入内核态进行系统调用处理呢？答案是在可执行程序的装载过程中，内核将向装载程序提供一个辅助向量（`Auxiliary Vectors`）来指示装载程序如何对可执行程序的运行环境进行初始化。

该辅助向量中的元素 `AT_SYSINFO` 指定了如何映射快速系统调用页面，以及将该页面映射到进程地址空间的什么位置。示例程序 7.7 根据 `AT_SYSINFO` 信息获取快速系统调用页面的内容，将其输入到标准出错 `stderr` 上。同样 `glibc` 的封装函数可以根据该信息获得快速系统调用页面的地址（处理细节有较大的差别），在用户请求系统调用时，调转到相应的地址，执行对应的指令（快速终端指令 `sysenter` 或软中断指令 `int 0x80`），然后进入内核态、进行系统调用的处理。

```

---
1 #include<stdio.h>
2 #include<elf.h>
3
4 int main(int argc, char** argv, char** envp)

```

```

5 {
6   void * VDSO_start;
7   Elf32_auxv_t *auxv;
8
9   while(*envp != NULL){
10      envp++;
11   };
12
13   auxv = (Elf32_auxv_t *) (--envp);
14   for( ; auxv->a_type != AT_NULL; auxv++)
15   {
16      if( auxv->a_type == AT_SYSINFO ){
17         printf("AT_SYSINFO:0x%x\n", auxv->a_un.a_val);
18         VDSO_start=(void *)auxv->a_un.a_val;
19         break;
20      }
21   }
22
23   VDSO_start =(void *)((int)VDSO_start & 0xFFFFF000);
24   fwrite( VDSO_start, 4096, 1, stderr);
25 }

```

示例程序 7.7: 获取快速系统调用页面内容

第 1~13 行代码用于找到辅助向量在环境变量数组 `envp` 中的位置, 该辅助向量是环境变量数组 `envp` 中的最后一个元素。找到后, 将环境变量的地址保存在变量 `auxv` 中。

第 14~21 行代码查询辅助向量, 找到辅助向量的 `AT_SYSINFO` 项, 该项保存了虚拟动态共享库映射到进程地址空间的信息。其中 `AT_SYSINFO` 是一个宏定义, 该宏定义在头文件 `/usr/include/elf.h` 中的第 967 行定义, 代码如下:

```

/* Pointer to the global system page used for
   system calls and other nice things. */
#define AT_SYSINFO 32
#define AT_SYSINFO_EHDR 33

```

第 23~24 行中, 第 23 行代码首先获取虚拟动态共享库所在页面的起始地址并将其保存在指针变量 `VDSO_start` 中。计算过程思路为: 按照 4KB 对齐。因为页面是最小的映射单位。第 24 行代码调用函数 `fwrite()` 将该页面中的内容写到标准出错 `stderr` 上。

示例程序 7.5 进程编译、运行的结果如图 7.10 所示。这里我们通过将标准出错重定向到文件 `dump` 上, 将读取的虚拟动态共享库保存到文件 `dump` 中。然后对该文件进行分析, 可知该页面中的内容为共享库, 该库的名称为 `Linux-gate.so.1`, 对其进行反汇编可知, 当前系统采用普通软中断系统调

```

lucas@HLKW:~$ ./example.7.7
AT_SYSINFO:0x324400
dump:   file format elf32-i386

Disassembly of section .text:

00004000 <_kernel_vsycall>:
400:  c0 80                int    $0x80
402:  c3                  ret
403:  90                  nop
404:  90                  nop
lucas@HLKW:~$

```

图 7.10 获取虚拟动态共享库 VDSO 并对其进行分析

用，即系统不支持快速系统调用。

7.6.3 处理过程

本小节讨论快速系统调用的处理过程。这里假设系统处理器支持快速系统调用指令 `SYSENTER`。在用户进程请求系统调用后，`glibc` 的封装函数会将系统调用号保存到寄存器 `%eax` 中，再跳转到虚拟动态共享代码 `VDSO` 的入口，这段代码负责将处理器的当前状态信息保存到用户态栈中；然后执行快速系统调用指令 `sysenter`，进入内核态完成系统调用的处理过程。在处理完毕后，该虚拟动态共享代码 `VDSO` 的后半部负责将保存到用户态栈的状态信息恢复到处理的寄存器中。

通过对“7.6.1 工作机制”的学习可知，在执行快速系统调用指令 `SYSENTER` 之后，系统将进入内核态，从标记 `sysenter_entry` 指示的地址开始运行。该标记是快速系统调用的入口地址。关于该标记指示代码段的具体功能和处理过程，请参见下面对该段代码的详细分析。

代码清单 7.11——函数 `sysenter_entry`

功能简介：快速系统调用的入口函数。负责设置系统调用使用的内核态栈和系统调用处理函数所需的参数。

文件：`src/arch/i386/kernel/entry.S`

```

179 ENTRY(sysenter_entry)
180     movl TSS_sysenter_esp0(%esp),%esp
181     sysenter_past_esp:
182         sti
183         pushl $(__USER_DS)
184         pushl %ebp
185         pushfl
186         pushl $(__USER_CS)
187         pushl $SYSENTER_RETURN
188
189 /*
190  * Load the potential sixth argument from user stack.
191  * Careful about security.
192  */
193     cmpl $__PAGE_OFFSET-3,%ebp
194     jae syscall_fault
195 1:   movl (%ebp),%ebp
196     .section __ex_table,"a"
197     .align 4
198     .long 1b,syscall_fault
199     .previous
200
201     pushl %eax
202     SAVE_ALL
203     GET_THREAD_INFO(%ebp)

```

第 180 行代码用于设置系统调用处理使用的内核态栈，通过对“7.6.1 工作机制”的学习可知，使用快速系统调用指令进入内核态后，栈指针 `%esp` 指向了当前处理器任务状态段中预留的 64×4 字节的空间。此时需要设置栈指针寄存器 `%esp` 指向当前进程的内核态栈，当前进程内核态栈的信息保存在任务状态段的成员变量 `esp0` 中，其距任务状态段结束地址的偏移量保存在宏定义 `TSS_sysenter_esp0` 中。该宏定义在文件

src/arch/i386/kernel/asm-offsets.c 中的第 67 行开始定义，代码如下。通过该行的操作，栈指针寄存器 %esp 指向了当前进程内核态栈的栈顶。

```
DEFINE(TSS_sysenter_esp0, offsetof(struct tss_struct, esp0) -
sizeof(struct tss_struct) )
```

第 182 行代码设置系统标记寄存器 EFLAGS 中断使能位 IF，允许系统响应外设中断。如果此时系统响应了外部中断，由于当前运行级别为 0，控制单元将借用当前内核态栈，不需要向内核态栈压入中断前系统的栈基址寄存器 %ss、栈指针寄存器 %esp。具体原因，请参见 5.4.1 小节。

在中断处理保护完现场之后，内核态栈中的内容如图 7.11 所示。但在中断处理完毕、进行中断返回时，仍将访问不存在的 %ss、%esp 信息，此时会超出当前进程内核态栈的页框，修改了系统其他模块的数据，会造成系统的崩溃。内核代码中的解决方法是在进程内核态栈的栈底保留 8 字节的空间。具体的解决方法，请参见 4.4.2 小节。

第 183 至 202 行代码以“纯手工方式”向内核态栈中压入系统调用所需的参数。对比基于软中断的系统调用处理过程，处理器的控制单元将自动向内核态栈中压入部分参数，自动压入的参数个数及类型，请参见 5.4.1 小节。这些压入的参数用于构成系统调用处理函数使用的参数 struct pt_regs。关于该数据结构中的成员变量及其相对布局，请参见 5.4.2 小节。

第 203 行代码用于计算出当前进程的 thread_info 的地址，并将其保存在寄存器 %ebp 中，关于该宏定义，请参见 4.2.4 小节。在此之后，快速系统调用后面的处理过程和普通系统调用的处理过程类似，此处不再赘述，请读者参考本章前面几个小节的论述。

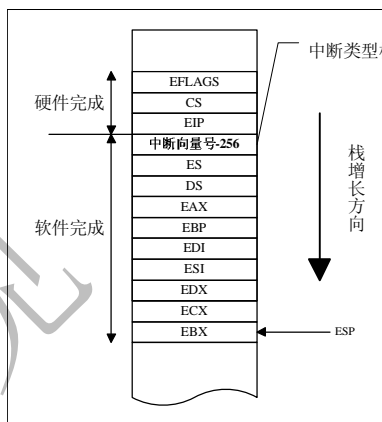


图 7.11 残缺的内核态栈

推荐课程：嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章: <http://www.embedu.org/courses/index.htm>
- 课程内容: <http://www.embedu.org/courses/course1.htm>
- 项目实战: <http://www.embedu.org/courses/project.htm>
- 出版教材: <http://www.embedu.org/courses/course3.htm>
- 实验设备: <http://www.embedu.org/courses/course5.htm>

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：

<http://www.farsight.com.cn/courses/TS-LinuxApp.htm>

- 嵌入式 Linux 系统开发班：

<http://www.farsight.com.cn/courses/TS-LinuxEMB.htm>

- 嵌入式 Linux 驱动开发班：

<http://www.farsight.com.cn/courses/TS-LinuxDriver.htm>

华清远见