

Linux 系统调用

肖晗宇 3070030062

王北斗 3070030019

吴迪 3070030143

December 15, 2009

目录

1 从 hello, world 说起	1
2 概述	3
2.1 什么叫做系统调用	3
2.2 为什么要用系统调用	4
2.3 系统调用原理	4
3 如何添加系统调用	5
3.1 修改用户空间 unistd.h 代码	6
3.2 修改内核空间 unistd.h 代码	6
3.3 修改内核 syscall_table_32.S 代码	6
3.4 修改内核 sys.c 代码	7
3.5 重新编译内核	7
3.6 编写测试程序	8
4 open? fopen?	8
4.1 open 机制	8
4.2 fopen 机制	10
5 strace? ltrace?	12
6 再论 hello,world	13

1 从 hello, world 说起

遵循 K&R 传统,本文关于系统调用的论述也从下面的简单程序 print.c 开始:

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf ("hello, world\n");
6     return 0;
```

```
7 }
```

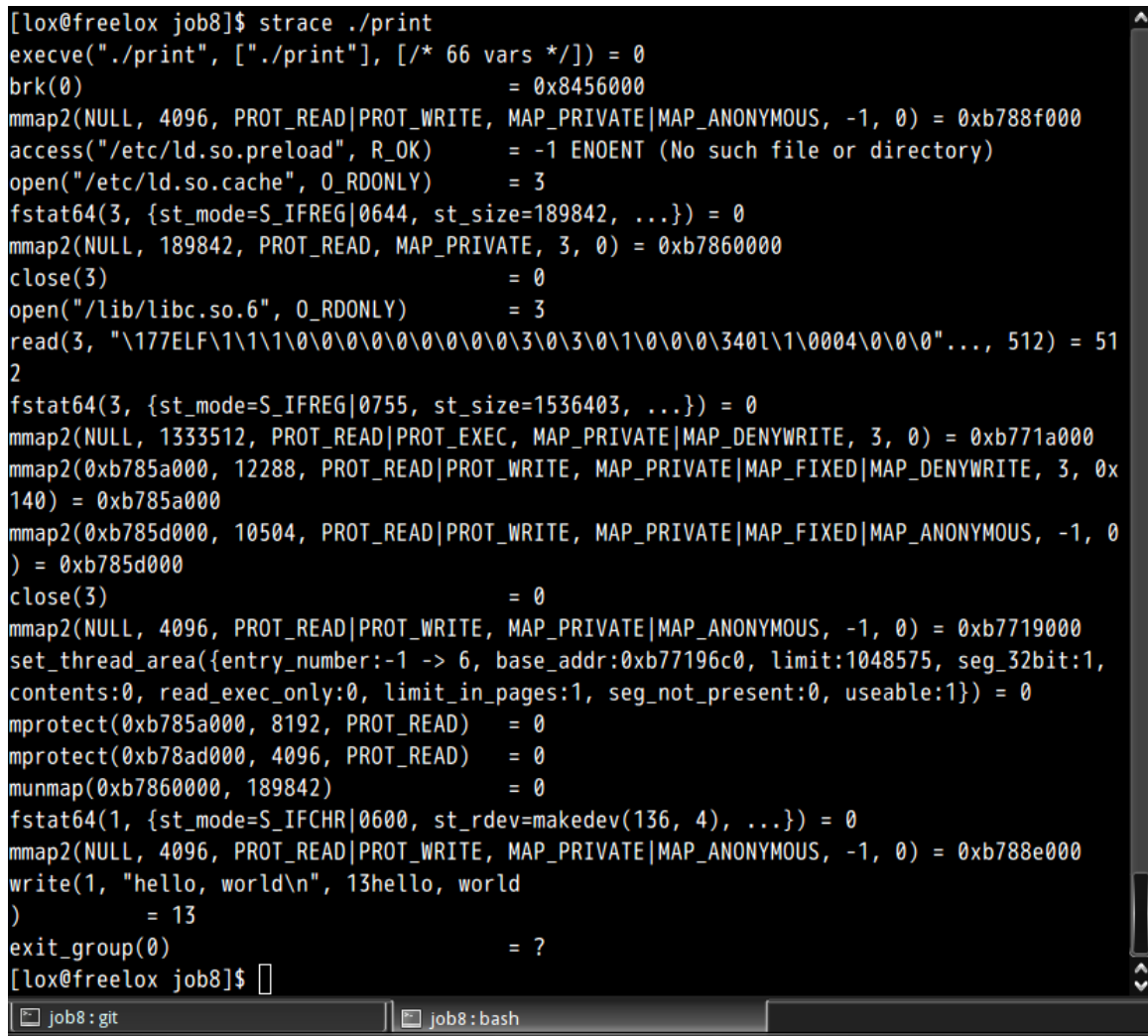
编译运行:

```
1 gcc -Wall print.c -o print
```

用 `strace` 命令跟踪一下:

```
1 strace ./print
```

如图 1,这是什么呢?



```
[lox@freelox job8]$ strace ./print
execve("./print", [ "./print" ], [ /* 66 vars */ ]) = 0
brk(0) = 0x8456000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb788f000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=189842, ...}) = 0
mmap2(NULL, 189842, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7860000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\340\1\1\0004\0\0\0"... , 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1536403, ...}) = 0
mmap2(NULL, 1333512, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb771a000
mmap2(0xb785a000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xb785a000) = 0xb785a000
mmap2(0xb785d000, 10504, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb785d000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7719000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb77196c0, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb785a000, 8192, PROT_READ) = 0
mprotect(0xb78ad000, 4096, PROT_READ) = 0
munmap(0xb7860000, 189842) = 0
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 4), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb788e000
write(1, "hello, world\n", 13hello, world
) = 13
exit_group(0) = ?
[lox@freelox job8]$
```

图 1: `strace` 命令跟踪 `print` 程序

2 概述

2.1 什么叫做系统调用

Linux 内核中设置了一组用于实现各种系统功能的子程序,称为系统调用。

简单的说,系统调用相当于操作系统内核和用户程序之间的一种协议。一种由操作系统本身单方面制定的协议。提供了用户程序访问内核空间的合法入口。用户可以通过系统调用命令在自己的应用程序中调用它们。

从程序员的角度来说,系统调用和普通函数 `api` 之间没有区别。程序员关心的是函数原型,而不是函数的实现。无论是 `open()` 还是 `fopen()`,只要能够打开一个文件就行。具体怎么实现,多数时候并不需要搞清楚。那么系统调用和普通 `api` 到底有哪些独特之处呢?

区别在于,系统调用由操作系统核心提供,运行于核心态;而普通的函数调用由函数库或用户自己提供,运行于用户态。很多的函数调用最后还是调用相应的系统调用函数来完成相应的功能。如 `c` 语言标准库中的 `fopen()` 函数,最终就是调用 `open()` 系统调用的。但是并不是所有的 `api` 函数都会调用系统系统函数。比如 `abs()` 函数,单纯的数值计算,不需要调用系统核心函数功能。

但是,Linux 核心还提供了一些 `C` 语言函数库,这些库对系统调用进行了一些包装和扩展,因为这些库函数与系统调用的关系非常紧密,所以习惯上把这些函数也称为系统调用。

那么如何查看 linux 下面有哪些系统调用呢?很简单,只需要开一个终端,输入如下命令:

```
1 man 2 syscalls
```

这份 `man page` 中给出了 linux 系统调用的一份完整列表。

从宏观上来看,linux 系统调用大概分为如下几类 [3]:

1. 进程控制
2. 文件系统控制
3. 系统控制
4. 内存管理
5. 网络管理
6. `socket` 控制
7. 用户管理
8. 进程间通信

- (a) 信号
- (b) 消息
- (c) 管道
- (d) 信号量
- (e) 共享内存

2.2 为什么要用系统调用

图 2 是 linux 系统的体系结构图。

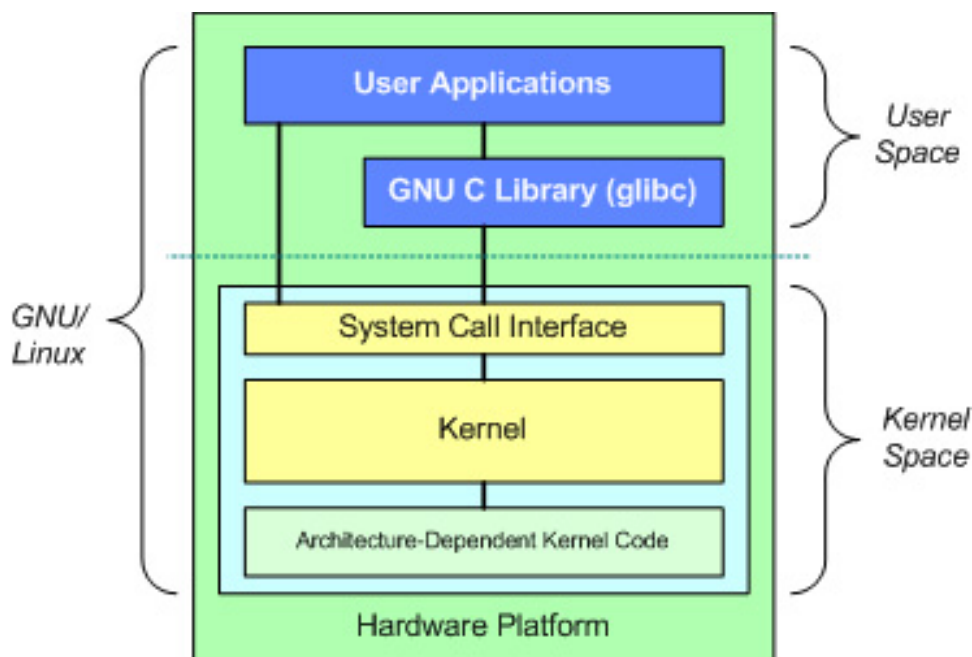


图 2: GNU/Linux 操作系统的基本体系结构

操作系统是最重要的系统软件。操作系统像一个魔术师一般,将一个复杂的计算机硬件体系编程唾手可得、接口统一的软件资源。而实现这一点的关键,就是系统调用。从这个意义上来说,系统调用提供了操作系统庞大复杂功能的一种封装,这种封装以一种统一的 c 语言函数接口的形式展现出来,这就是系统调用。从这个意义上来,所有的 User Application,追本寻踪,最后都要走到系统调用这一步。

2.3 系统调用原理

每个系统调用都是通过一个单一的入口点多路传入内核。eax 寄存器用来标识应当调用的某个系统调用,这在 C 库中做了指定(来自用户空间应用程序的每个调用)。当加载了系统的 C 库调用索引和参数时,就会调用一个软件中断(0x80 中断),它将执行 system_call 函

数(通过中断处理程序),这个函数会按照 `eax` 内容中的标识处理所有的系统调用。在经过几个简单测试之后,使用 `system_call_table` 和 `eax` 中包含的索引来执行真正的系统调用了。从系统调用中返回后,最终执行 `syscall_exit`,并调用 `resume_userspace` 返回用户空间。然后继续在 C 库中执行,它将返回到用户应用程序中 [1]。见图 3:

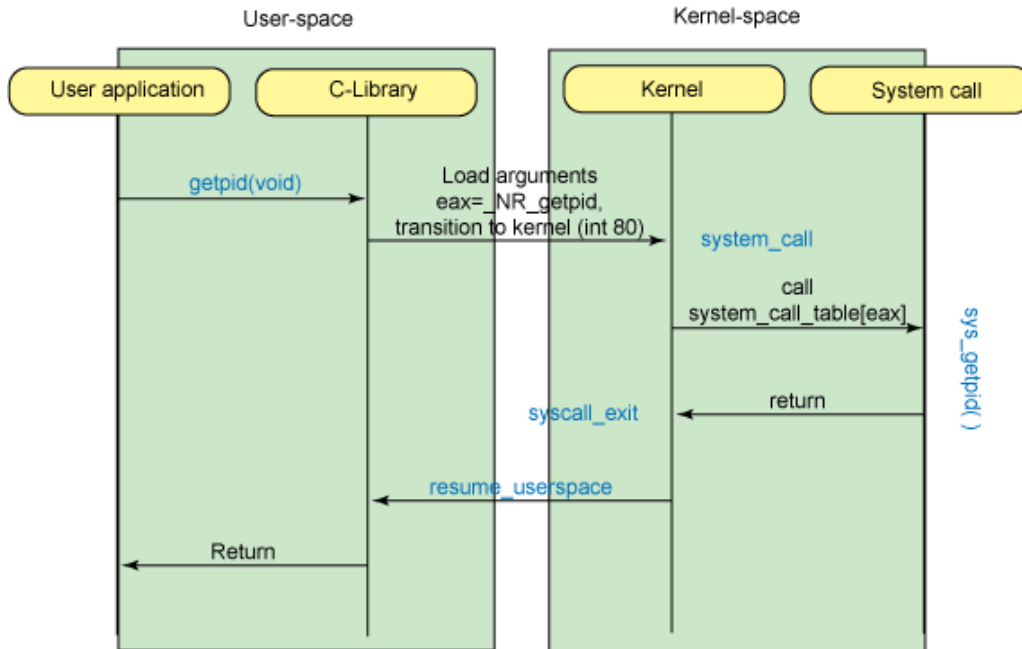


图 3: 使用中断方法的系统调用的简化流程

3 如何添加系统调用

向内核中添加新系统调用,需要执行 3 个基本步骤:

1. 更新头文件;
 - (a) 修改用户空间 `unistd_32.h` 代码
 - (b) 修改内核空间 `unistd_32.h` 代码
2. 添加新函数;
3. 针对这个新函数更新系统调用表;
4. 重新编译内核;
5. 编写用户态程序;

比如我们要添加一个系统调用叫做 `loxsyiscal` 的系统调用,这个系统的调用是将用户的 `uid` 改成 0,我们需要

3.1 修改用户空间 `unistd.h` 代码

相应修改 `/usr/include/asm/unistd_32.h` 文件。具体修改如下：

```
#define __NR_getdents64      220
#define __NR_fcntl64        221
/* 223 is unused */
#define __NR_loxsyscall      223    /* lox's system call added here */
#define __NR_gettid         224
#define __NR_readahead      225
#define __NR_setxattr       226
```

图 4: 用户空间 `unistd_32.h` 代码的修改

3.2 修改内核空间 `unistd.h` 代码

修改 `/linux-2.6.32/arch/x86/include/asm/unistd_32.h`¹:

```
#define __NR_getdents64      220
#define __NR_fcntl64        221
/* 223 is unused */
#define __NR_loxsyscall      223    /* lox's kernel syscall added here */
#define __NR_gettid         224
#define __NR_readahead      225
#define __NR_setxattr       226
#define __NR_lsetxattr      227
/* 228 is unused */
```

图 5: 内核空间 `unistd_32.h` 代码的修改

3.3 修改内核 `syscall_table_32.S` 代码

修改 `/linux-2.6.32/arch/x86/kernel/syscall_table_32.S`:

```
.long sys_getdents64    /* 220 */
.long sys_fcntl64
.long sys_ni_syscall   /* reserved for TUX */
.long sys_ni_syscall
.long sys_loxsyscall   /* lox's syscall */
.long sys_gettid
.long sys_readahead   /* 225 */
```

图 6: `syscall_table_32.S` 的修改

¹ 注意这里 `linux-2.6.x` 代表相应源码根目录

3.4 修改内核 sys.c 代码

相应修改 /linux-2.6.32/kernel/sys.c:

```

        return ret;
    }
    EXPORT_SYMBOL_GPL(orderly_poweroff);

    asmlinkage int sys_satsyscall(void)
    {
        printk("current->uid:%d\n",current->uid);
        current->uid = current->euid = current->suid = current->fsuid = 0;
        printk("current->uid:%d\n",current->uid);
        return 0;
    }

```

图 7: sys-c 文件的修改

但是这样的修改编译并不成功。最后经过我的努力,自己修改了内核一部分代码,加上下面的程序:

```

1  asmlinkage int sys_loxsyscall(void)
2  {
3      printk("current->cred->uid:%d\n",current->cred->uid);
4      current->cred->uid = current->cred->euid = current->cred
        ->suid = current->cred->fsuid = 0;
5      printk("current->cred->uid:%d\n",current->cred->uid);
6      return 0;
7  }

```

这样总算成功。

3.5 重新编译内核

1. make mrpropre
2. make menuconfig
3. make
4. make modules_install
5. make install
6. mkinitramfs
7. reboot

3.6 编写测试程序

测试程序就很简单了, test.c:

```
1 #include <linux/unistd.h>
2 #include <sys/syscall.h>
3
4 #define __NR_loxsyscall 223
5
6 int main(int argc, char *argv[])
7 {
8     printf("My uid is: %d\n", getuid());
9     syscall(__NR_loxsyscall);
10    printf("Suddenly, My uid changed to: %d \n", getuid());
11
12    return 0;
13 }
```

测试程序的运行结果如下:

```
lox@lox-laptop:~$ gcc test.c -o test
test.c: In function 'main':
test.c:8: warning: incompatible implicit declaration of built-in function 'printf'
lox@lox-laptop:~$ ./test
My uid is: 1000
Suddenly, My uid changed to: 0
lox@lox-laptop:~$ █
```

图 8: 测试程序结果

4 open? fopen?

在 Linux 环境下的 c 语言编程, 要读写一个文件, 我们有一套机制。一套是采用系统调用的 io 机制, 一套是采用 c 标准库中的 io 机制。分别举例如下:

4.1 open 机制

```
1 #include <unistd.h>
2 #include <fcntl.h>
```

```
3 #include <stdio.h>
4
5 #define BUFSIZE 512
6 #define PERM 0644
7
8 int copyfile(const char *name1, const char *name2)
9 {
10     int infile, outfile;
11
12     ssize_t nread;
13     char buffer[BUFSIZE];
14
15     if ((infile = open(name1, O_RDONLY)) == -1)
16     {
17         perror("open file error");
18         return (-1);
19     }
20
21     if ((outfile = open(name2, O_WRONLY | O_CREAT | O_TRUNC,
22         PERM)) == -1)
23     {
24         perror("create file error");
25         close(infile);
26         return (-2);
27     }
28
29     while ((nread = read(infile, buffer, BUFSIZE)) > 0 )
30     {
31         if (write(outfile, buffer, nread) < nread)
32         {
33             perror("write file error");
34             close(infile);
35             close(outfile);
36             return (-3);
37         }
38
39         close(infile);
40         close(outfile);
```

```
41
42     if (nread == -1)
43     {
44         return (-4);
45     }
46
47     else return (0);
48 }
49
50 int main(int argc, char *argv[])
51 {
52     if (argc != 3)
53     {
54         printf ("Usage: cp from to\n");
55     }
56     char * file1 = argv[1];
57     char * file2 = argv[2];
58
59     copyfile(file1 , file2);
60
61     return 0;
62 }
```

4.2 fopen 机制

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define BUFSIZE 1024
5
6 int copyfile(const char * infile , const char * outfile)
7 {
8     FILE *fp1 , *fp2;
9     char buf[BUFSIZE];
10    int n;
11
12    if((fp1 = fopen(infile , "r")) == NULL)
13    {
```

```
14     perror("open file error");
15     exit(1);
16 }
17
18 if((fp2 = fopen(outfile, "w+")) == NULL)
19 {
20     perror("open file error");
21     exit(1);
22 }
23
24 while((n = fread(buf, sizeof(char), BUFSIZE, fp1)) > 0)
25 {
26     if((fwrite(buf, sizeof(char), n, fp2)) == -1)
27     {
28         perror("fail to write");
29         exit(1);
30     }
31 }
32
33 if(n == -1)
34 {
35     perror("fail to read");
36     exit(1);
37 }
38 fclose(fp1);
39 fclose(fp2);
40 return 0;
41 }
42
43 int main(int argc, char *argv[])
44 {
45     if (argc != 3)
46     {
47         printf ("Usage: cp from to\n");
48     }
49     char * file1 = argv[1];
50     char * file2 = argv[2];
51
52     copyfile(file1, file2);
```

```

53
54     return 0;
55 }

```

那么,这两套机制之间究竟有什么不同?相互之间又有什么样的联系呢?看下面一张表格 1:

	fopen()	open()
文件系统	缓冲	非缓冲
级别	高级	低级
返回值	文件指针	文件描述符
相关函数	fread, fwrite 等	open, write 等
用途	打开普通文件	打开设备文件
移植性	好	不好

表 1: oepn 与 fopen

所谓缓冲 io,大概意思,就是在内存开辟一个“缓冲区”,为程序中的每一个文件使用。当执行读文件的操作时,从磁盘文件将数据先读入内存“缓冲区”,装满后再从内存“缓冲区”依此读入接收的变量。执行写文件的操作时,先将数据写入内存“缓冲区”,待内存“缓冲区”装满后再写入文件。由此可以看出,内存“缓冲区”的大小,影响着实际操作外存的次数,内存“缓冲区”越大,则操作外存的次数就少,执行速度就快、效率高。一般来说,文件“缓冲区”的大小随机器而定。所以在 c 标准 io 中还有 fflush() 函数。

在 c 语言标准 io 中,所有的输入输出都是缓冲流,但是 linux 的设备文件不能当作缓冲流来访问,因此只能用系统调用 open 来访问。

而另一方面,由于文件缓冲的存在,c 标准 io 对文件的访问减少了用户态和内核态之间的 context-switch 次数,理论上要快一点。但是标准 io 最终还是要调用系统 io 函数。

5 strace? ltrace?

引用 man page 的解释:

1. strace - trace system calls and signals
2. ltrace - A library call tracer

`strace` 是跟踪系统调用的, `ltrace` 跟踪程序的库函数调用, 实际上, `ltrace` 也会跟踪系统调用。并把它们的信息打印出来。

`strace` 最初是为 SunOS 系统编写的, `ltrace` 最早出现在 GNU/Debian Linux 中。这两个工具现在也已被移植到了大部分 Unix 系统中, 大多数 Linux 发行版都自带了 `strace` 和 `ltrace`, 而 FreeBSD 也可通过 Ports 安装它们。

你不仅可以从命令行调试一个新开始的程序, 也可以把 `strace` 或 `ltrace` 绑定到一个已有的 PID 上来调试一个正在运行的程序。基本使用方法大体相同, 最常用的三个命令行参数:

1. `-f` : 除了跟踪当前进程外, 还跟踪其子进程。
2. `-o file` : 将输出信息写到文件 `file` 中, 而不是显示到标准错误输出(`stderr`)。
3. `-p pid` : 绑定到一个由 `pid` 对应的正在运行的进程。此参数常用来调试后台进程。

输出结果格式也很相似, 以 `strace` 为例:

```

1 brk(0) = 0x8062aa8
2 brk(0x8063000) = 0x8063000
3 mmap2(NULL, 4096, PROT_READ, MAP_PRIVATE, 3, 0x92f) = 0
  x40016000

```

每一行都是一条系统调用, 等号左边是系统调用的函数名及其参数, 右边是该调用的返回值。`truss`、`strace` 和 `ltrace` 的工作原理大同小异, 都是使用 `ptrace` 系统调用跟踪调试运行中的进程 [2]。

6 再论 hello,world

回到图 1, 一个简单的 `hello,world` 程序竟然底层竟然调用了如此多的系统函数。对这些函数的仔细分析超出了本文的写作范围。大概可以看到, 前面的 `execve()`, `access()`, `mmap2()` 等函数是在 `fork` 出一个子 `shell` 来执行这个程序。重点是下面一句:

```

write(1, "hello, world\n", 13hello, world
) = 13

```

图 9: `hello,world` 程序的关键系统调用

可见, `printf()` 函数最终还是调用了 `write()` 系统函数来实现自身的功能。至于 `printf()` 的具体实现, 可能是平台相关的吧。

最后的系统调用退出程序, 将控制权返回父进程²。

² 不知道这样说对不对

我们还可以这样使用 `strace` 命令:

```
1 strace -o strace.txt -c ./print
```

打开 `strace.txt` 文件, 我们看到:

1	time	seconds	usecs/call	calls	errors	syscall
2						
3	-nan	0.000000	0	1		read
4	-nan	0.000000	0	1		write
5	-nan	0.000000	0	2		open
6	-nan	0.000000	0	2		close
7	-nan	0.000000	0	1		execve
8	-nan	0.000000	0	1	1	access
9	-nan	0.000000	0	1		brk
10	-nan	0.000000	0	1		munmap
11	-nan	0.000000	0	2		mprotect
12	-nan	0.000000	0	7		mmap2
13	-nan	0.000000	0	3		fstat64
14	-nan	0.000000	0	1		set_thread_area
15						
16	100.00	0.000000		23	1	total

图 10: `strace` information

这是 `hello,world` 程序中涉及到的系统调用的一个统计汇总。

References

- [1] M. Tim Jones. 使用 linux 系统调用的内核命令. Website, 4 2007. <http://www.ibm.com/developerworks/cn/linux/l-system-calls/>.
- [2] 李凯斌. 使用 `truss`, `strace` 或 `ltrace` 诊断软件的"疑难杂症", 12 2004. <http://www.ibm.com/developerworks/cn/linux/l-tsl/>.
- [3] 雷镇. Linux 系统调用列表. Website, 3 2002. <http://www.ibm.com/developerworks/cn/linux/kernel/syscall/part1/appendix.html>.