《操作系统》实验

6.2 I/O设备管理实验 - 代码分析

祝嘉栋 2012211196 @304班

1. 实验内容

阅读 Linux/Minux 中以下模块的调用主线

1. `print()` 函数内部实现模块调用主线
2. `scanf()` 函数内部实现模块调用主线

写出分析报告

2. 分析报告

以下报告采用 `glibc 2.17` 进行分析

`printf()` 函数

用户空间

首先，在 `/libio/stdio.h` 中找到 `printf()` 的函数声明

```
0358 /* Write formatted output to stdout.
0359
0360    This function is a possible cancellation point and therefore not
0361    marked with __THROW.  */
0362 extern int printf (const char *__restrict __format, ...);
```

但是实际上没有名为 `printf()` 的函数定义，在 `/stdio-common/printf.c` 中找到强别名替换

```
0040 ldbl_strong_alias (__printf, printf);
```

`print` 被替换成了 `__printf`，在相同文件内找到 `__printf()` 的定义

```
0024 /* Write formatted output to stdout from the format string FORMAT.
 */
0025 /* VARARGS1 */
0026 int
0027 __printf (const char *format, ...)
0028 {
0029   va_list arg;
0030   int done;
0031
0032   va_start (arg, format);
0033   done = vfprintf (stdout, format, arg);
0034   va_end (arg);
0035
0036   return done;
0037 }
```

观察到调用了 `vfprintf()` 函数

在 `/stdio-common/vfprintf.c` 中找到 `vfprintf()` 的定义，在定义使用 `outstring()` 来输出字符串。

```
0220 int
0221 vfprintf (FILE *s, const CHAR_T *format, va_list ap)
0222 {
...
0756       outstring (string, workend - string);                    \
...
2057 }
```

在同文件中找到 `outstring()` 宏的定义

```
0160 #define outstring(String, Len)
...
0164       if ((size_t) PUT (s, (String), (Len)) != (size_t) (Len))
       \
...
```

发现调用了 `PUT()`，其定义在117行

```
0117 # define PUT(F, S, N)   _IO_sputn ((F), (S), (N))
```

继续跟踪 `_IO_sputn()` ，找到 `/libio/libioP.h`

```
0381 #define _IO_sputn(__fp, __s, __n) _IO_XSPUTN (__fp, __s, __n)
```

而 `_IO_XSPUTN` 实际上调用了 `stdout` 变量的 `__xsputn()` 方法

```
// /libio/libioP.h

0173 #define _IO_XSPUTN(FP, DATA, N) JUMP2 (__xsputn, FP, DATA, N)

0122 #define JUMP2(FUNC, THIS, X1, X2) (_IO_JUMPS_FUNC(THIS)->FUNC) (THI
S, X1, X2)

0115 # define _IO_JUMPS_FUNC(THIS) _IO_JUMPS ((struct _IO_FILE_plus *) (
THIS))

0105 #define _IO_JUMPS(THIS) (THIS)->vtable

...

0325 struct _IO_FILE_plus
0326 {
0327   _IO_FILE file;
0328   const struct _IO_jump_t *vtable;
0329 };

0290 struct _IO_jump_t
0291 {
0292     JUMP_FIELD(size_t, __dummy);
0293     JUMP_FIELD(size_t, __dummy2);
0294     JUMP_FIELD(_IO_finish_t, __finish);
0295     JUMP_FIELD(_IO_overflow_t, __overflow);
0296     JUMP_FIELD(_IO_underflow_t, __underflow);
0297     JUMP_FIELD(_IO_underflow_t, __uflow);
0298     JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
0299     /* showmany */
0300     JUMP_FIELD(_IO_xsputn_t, __xsputn);
0301     JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
```

```
0302     JUMP_FIELD(_IO_seekoff_t, __seekoff);
0303     JUMP_FIELD(_IO_seekpos_t, __seekpos);
0304     JUMP_FIELD(_IO_setbuf_t, __setbuf);
0305     JUMP_FIELD(_IO_sync_t, __sync);
0306     JUMP_FIELD(_IO_doallocate_t, __doallocate);
0307     JUMP_FIELD(_IO_read_t, __read);
0308     JUMP_FIELD(_IO_write_t, __write);
0309     JUMP_FIELD(_IO_seek_t, __seek);
0310     JUMP_FIELD(_IO_close_t, __close);
0311     JUMP_FIELD(_IO_stat_t, __stat);
0312     JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
0313     JUMP_FIELD(_IO_imbue_t, __imbue);
0314 #if 0
0315     get_column;
0316     set_column;
0317 #endif
0318 };
```

下面要找到 `stdout` 的 `__xsputn()` 的实现

```
> /libio/stdio.c

// 发现 stdout 即 _IO_2_1_stdout_
0034 _IO_FILE *stdout = (FILE *) &_IO_2_1_stdout_;

> /libio/stdfiles.c

// 使用 DEF_STDFILE 宏初始化 _IO_2_1_stdout_
0069 DEF_STDFILE(_IO_2_1_stdout_, 1, &_IO_2_1_stdin_, _IO_NO_READS);

// 发现函数列表为 _IO_file_jumps
0062   struct _IO_FILE_plus NAME \
0063     = {FILEBUF_LITERAL(CHAIN, FLAGS, FD, NULL), \
0064        &_IO_file_jumps};

> /libio/fileops.c

// 找到 _IO_file_jumps
1541 const struct _IO_jump_t _IO_file_jumps =
1542 {
1543   JUMP_INIT_DUMMY,
```

```
1544   JUMP_INIT(finish, _IO_file_finish),
1545   JUMP_INIT(overflow, _IO_file_overflow),
1546   JUMP_INIT(underflow, _IO_file_underflow),
1547   JUMP_INIT(uflow, _IO_default_uflow),
1548   JUMP_INIT(pbackfail, _IO_default_pbackfail),
1549   JUMP_INIT(xsputn, _IO_file_xsputn),
1550   JUMP_INIT(xsgetn, _IO_file_xsgetn),
1551   JUMP_INIT(seekoff, _IO_new_file_seekoff),
1552   JUMP_INIT(seekpos, _IO_default_seekpos),
1553   JUMP_INIT(setbuf, _IO_new_file_setbuf),
1554   JUMP_INIT(sync, _IO_new_file_sync),
1555   JUMP_INIT(doallocate, _IO_file_doallocate),
1556   JUMP_INIT(read, _IO_file_read),
1557   JUMP_INIT(write, _IO_new_file_write),
1558   JUMP_INIT(seek, _IO_file_seek),
1559   JUMP_INIT(close, _IO_file_close),
1560   JUMP_INIT(stat, _IO_file_stat),
1561   JUMP_INIT(showmanyc, _IO_default_showmanyc),
1562   JUMP_INIT(imbue, _IO_default_imbue)
1563 };
```

看到 `stdout` 的 `xsputn` 实现为 `IO_file_xsputn` ，而 `IO_file_xsputn` 是一个带版本的符号，在2.1版本的glibc中符号被替换为 `_IO_new_file_xsputn`

```
1538 versioned_symbol (libc, _IO_new_file_xsputn, _IO_file_xsputn, GLIBC
_2_1);

// _IO_new_file_xsputn() 函数定义
1268 _IO_size_t
1269 _IO_new_file_xsputn (f, data, n)
1270      _IO_FILE *f;
1271      const void *data;
1272      _IO_size_t n;
...
1335       count = new_do_write (f, s, do_write); // 调用了new_do_write
//
1348 }

// new_do_write() 函数定义
0507 static
0508 _IO_size_t
0509 new_do_write (fp, data, to_do)
0510      _IO_FILE *fp;
0511      const char *data;
0512      _IO_size_t to_do;
0513 {
...
0530   count = _IO_SYSWRITE (fp, data, to_do); // 调用了 _IO_SYSWRITE 宏
...
0539 }
```

调用 `_IO_SYSWRITE` 宏即为调用 `stdout` 的 `write` 方法

```
> /libio/libioP.h

0246 #define _IO_SYSWRITE(FP, DATA, LEN) JUMP2 (__write, FP, DATA, LEN)
```

在上面提及的 `_IO_file_jumps` 中找到 `__write()` 方法的实现为 `_IO_new_file_write()`

```
> /libio/fileops.c

1241 _IO_ssize_t
1242 _IO_new_file_write (f, data, n)
1243      _IO_FILE *f;
1244      const void *data;
1245      _IO_ssize_t n;
1246 {
...
1250       _IO_ssize_t count = (__builtin_expect (f->_flags2
1251                            & _IO_FLAGS2_NOTCANCEL, 0)
1252                ? write_not_cancel (f->_fileno, data, to_do)
1253                : write (f->_fileno, data, to_do));
...
1266 }
```

调用了 `write_not_cancel` 和 `write` 函数，在 `unistd.h` 中定义

`write` 函数为Linux内核编译时根据系统调用表自动生成的函数，即进行了系统调用，将输出文件的文件描述符、字符串首地址、字符串长度传给内核空间。

内核空间

对内核空间的分析采用了 `linux 3.16` 版本内核

write系统调用的函数定义位于Linux内核源码 `/fs/read_write.c` 中：

```
> /fs/read_write.c

477 SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
478                 size_t, count)
479 {
480         struct file *file;
481         ssize_t ret = -EBADF;
482         int fput_needed;
483
484         file = fget_light(fd, &fput_needed);
485         if (file) {
486                 loff_t pos = file_pos_read(file);
487                 ret = vfs_write(file, buf, count, &pos);
488                 file_pos_write(file, pos);
489                 fput_light(file, fput_needed);
490         }
491
492         return ret;
493 }
```

观察到调用了vfs_write()函数

```
> /fs/read_write.c

420 ssize_t vfs_write(struct file *file, const char __user *buf, size_t
count, loff_t *pos)
421 {
...
435                         ret = file->f_op->write(file, buf, count, po
s);
...
446 }
```

注意到一开始传递给 `write` 系统调用的文件描述符为1，编写一个简单程序使用 `printf()` 输出一段字符串，观察 `/proc/[pid]/fd` 中打开文件列表。发现文件描述符1对应设备 `/dev/pts/9` 即伪终端。

```
$ ll /proc/9303/fd
总用量 0
dr-x------ 2 edward edward  0 12月 30 19:36 ./
dr-xr-xr-x 9 edward edward  0 12月 30 19:36 ../
lrwx------ 1 edward edward 64 12月 30 19:36 0 -> /dev/pts/9
lrwx------ 1 edward edward 64 12月 30 19:36 1 -> /dev/pts/9
lrwx------ 1 edward edward 64 12月 30 19:36 2 -> /dev/pts/9
```

则 `fget_light()` 函数会获取到 `/dev/pts/9` 的文件结构，使用该文件结构的 `write` 方法写入终端。

`pts` 设备驱动位于 `/drivers/tty/pty.c`

```
> /drivers/tty/pty.c


114 static int pty_write(struct tty_struct *tty, const unsigned char *bu
f, int c)
115 {
116         struct tty_struct *to = tty->link;
117
118         if (tty->stopped)
119                 return 0;
120
121         if (c > 0) {
122                 /* Stuff the data into the input queue of the other
end */
123                 c = tty_insert_flip_string(to->port, buf, c);
124                 /* And shovel */
125                 if (c)
126                         tty_flip_buffer_push(to->port);
127         }
128         return c;
129 }

> /drivers/tty/tty_flip.h

 32 static inline int tty_insert_flip_string(struct tty_port *port,
 33                 const unsigned char *chars, size_t size)
 34 {
```

```
 35         return tty_insert_flip_string_fixed_flag(port, chars, TTY_NO
RMAL, size);
 36 }

> /drivers/tty/tty_buffer.c

294 int tty_insert_flip_string_fixed_flag(struct tty_port *port,
295                 const unsigned char *chars, char flag, size_t size)
296 {
297         int copied = 0;
298         do {
299                 int goal = min_t(size_t, size - copied, TTY_BUFFER_P
AGE);
300                 int flags = (flag == TTY_NORMAL) ? TTYB_NORMAL : 0;
301                 int space = __tty_buffer_request_room(port, goal, fl
ags);
302                 struct tty_buffer *tb = port->buf.tail;
303                 if (unlikely(space == 0))
304                         break;
305                 memcpy(char_buf_ptr(tb, tb->used), chars, space);
306                 if (~tb->flags & TTYB_NORMAL)
307                         memset(flag_buf_ptr(tb, tb->used), flag, spa
ce);
308                 tb->used += space;
309                 copied += space;
310                 chars += space;
311                 /* There is a small chance that we need to split the
 data over
312                    several buffers. If this is the case we must loop
 */
313         } while (unlikely(size > copied));
314         return copied;
315 }
```

可以看到最终将要输出的字符串拷贝到了输出缓冲区的末尾，之后再通过`tty_flip_buffer_push()`将缓冲区文件打到屏幕上。

`scanf()`函数

scanf函数的调用过程与printf大致相同，在此不再赘述。

参考资料

- "Where the printf() Rubber Meets the Road" - hostilefork.com
- "glibc Cross Reference" - Cross reference for open source projects
- "Linux Cross Reference" - Free Electrons
- "Printf背后的故事" - Florian