

39 | 管道：项目组A完成了，如何交接给项目组B？

2019-06-26 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超


时长 12:43 大小 11.66M



在这一章的第一节里，我们大概讲了管道的使用方式以及相应的命令行。这一节，我们就具体来看一下管道是如何实现的。

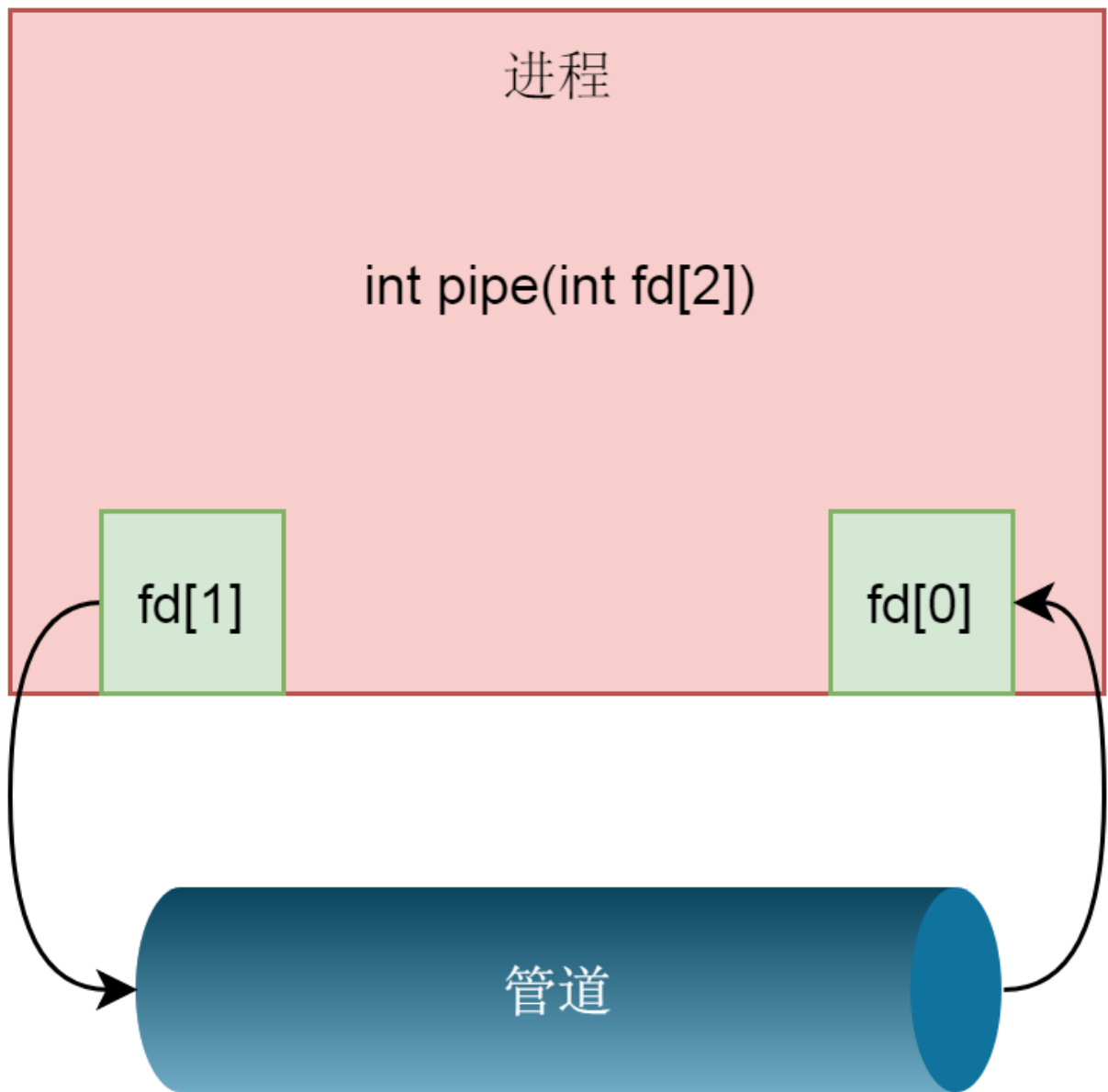
我们先来看，我们常用的**匿名管道**（Anonymous Pipes），也即将多个命令串起来的竖线，背后的原理到底是什么。

上次我们说，它是基于管道的，那管道如何创建呢？管道的创建，需要通过下面这个系统调用。

 复制代码

```
1 int pipe(int fd[2])
```

在这里，我们创建了一个管道 `pipe`，返回了两个文件描述符，这表示管道的两端，一个是管道的读取端描述符 `fd[0]`，另一个是管道的写入端描述符 `fd[1]`。



我们来看在内核里面是如何实现的。

[复制代码](#)

```
1 SYSCALL_DEFINE1(pipe, int __user *, fildes)
2 {
3     return sys_pipe2(fildes, 0);
4 }
5
6 SYSCALL_DEFINE2(pipe2, int __user *, fildes, int, flags)
7 {
```


```

8      struct file *files[2];
9      int fd[2];
10     int error;
11
12     error = __do_pipe_flags(fd, files, flags);
13     if (!error) {
14         if (unlikely(copy_to_user(filides, fd, sizeof(fd)))) {
15         .....
16             error = -EFAULT;
17         } else {
18             fd_install(fd[0], files[0]);
19             fd_install(fd[1], files[1]);
20         }
21     }
22     return error;
23 }

```

在内核中，主要的逻辑在 pipe2 系统调用中。这里面要创建一个数组 files，用来存放管道的两端的打开文件，另一个数组 fd 存放管道的两端的文件描述符。如果调用 __do_pipe_flags 没有错误，那就调用 fd_install，将两个 fd 和两个 struct file 关联起来。这一点和打开一个文件的过程很像了。

我们来看 __do_pipe_flags。这里面调用了 create_pipe_files，然后生成了两个 fd。从这里可以看出，fd[0] 是用于读的，fd[1] 是用于写的。


 复制代码

```

1 static int __do_pipe_flags(int *fd, struct file **files, int flags)
2 {
3     int error;
4     int fdw, fdr;
5     .....
6     error = create_pipe_files(files, flags);
7     .....
8     error = get_unused_fd_flags(flags);
9     .....
10    fdr = error;
11
12    error = get_unused_fd_flags(flags);
13    .....
14    fdw = error;
15
16    fd[0] = fdr;
17    fd[1] = fdw;
18    return 0;
19    .....

```

创建一个管道，大部分的逻辑其实都是在 `create_pipe_files` 函数里面实现的。这一章第一节的时候，我们说过，命名管道是创建在文件系统上的。从这里我们可以看出，匿名管道，也是创建在文件系统上的，只不过是一种特殊的文件系统，创建一个特殊的文件，对应一个特殊的 inode，就是这里面的 `get_pipe_inode`。

 复制代码

```

1 int create_pipe_files(struct file **res, int flags)
2 {
3     int err;
4     struct inode *inode = get_pipe_inode();
5     struct file *f;
6     struct path path;
7     .....
8     path.dentry = d_alloc_pseudo(pipe_mnt->mnt_sb, &empty_name);
9     .....
10    path.mnt = mntget(pipe_mnt);
11
12    d_instantiate(path.dentry, inode);
13
14    f = alloc_file(&path, FMODE_WRITE, &pipefifo_fops);
15    .....
16    f->f_flags = O_WRONLY | (flags & (O_NONBLOCK | O_DIRECT));
17    f->private_data = inode->i_pipe;
18
19    res[0] = alloc_file(&path, FMODE_READ, &pipefifo_fops);
20    .....
21    path_get(&path);
22    res[0]->private_data = inode->i_pipe;
23    res[0]->f_flags = O_RDONLY | (flags & O_NONBLOCK);
24    res[1] = f;
25    return 0;
26    .....
27 }
```

从 `get_pipe_inode` 的实现，我们可以看出，匿名管道来自一个特殊的文件系统 `pipefs`。这个文件系统被挂载后，我们就得到了 `struct vfsmount *pipe_mnt`。然后挂载的文件系统的 `superblock` 就变成了：`pipe_mnt->mnt_sb`。如果你对文件系统的操作还不熟悉，要返回去复习一下文件系统那一章啊。

```

1 static struct file_system_type pipe_fs_type = {
2     .name          = "pipefs",
3     .mount          = pipefs_mount,
4     .kill_sb        = kill_anon_super,
5 };
6
7 static int __init init_pipe_fs(void)
8 {
9     int err = register_filesystem(&pipe_fs_type);
10
11     if (!err) {
12         pipe_mnt = kern_mount(&pipe_fs_type);
13     }
14     .....
15 }
16
17 static struct inode * get_pipe_inode(void)
18 {
19     struct inode *inode = new_inode_pseudo(pipe_mnt->mnt_sb);
20     struct pipe_inode_info *pipe;
21     .....
22     inode->i_ino = get_next_ino();
23
24     pipe = alloc_pipe_info();
25     .....
26     inode->i_pipe = pipe;
27     pipe->files = 2;
28     pipe->readers = pipe->writers = 1;
29     inode->i_fop = &pipefifo_fops;
30     inode->i_state = I_DIRTY;
31     inode->i_mode = S_IFIFO | S_IRUSR | S_IWUSR;
32     inode->i_uid = current_fsuid();
33     inode->i_gid = current_fsgid();
34     inode->i_atime = inode->i_mtime = inode->i_ctime = current_time(inode);
35
36     return inode;
37     .....
38 }

```

我们从 `new_inode_pseudo` 函数创建一个 `inode`。这里面开始填写 `Inode` 的成员，这里和文件系统的很像。这里值得注意的是 `struct pipe_inode_info`，这个结构里面有个成员是 `struct pipe_buffer *bufs`。我们可以知道，**所谓的匿名管道，其实就是内核里面的一串缓存。**

另外一个需要注意的是 `pipefifo_fops`，将来我们对于文件描述符的操作，在内核里面都是对应这里面的操作。

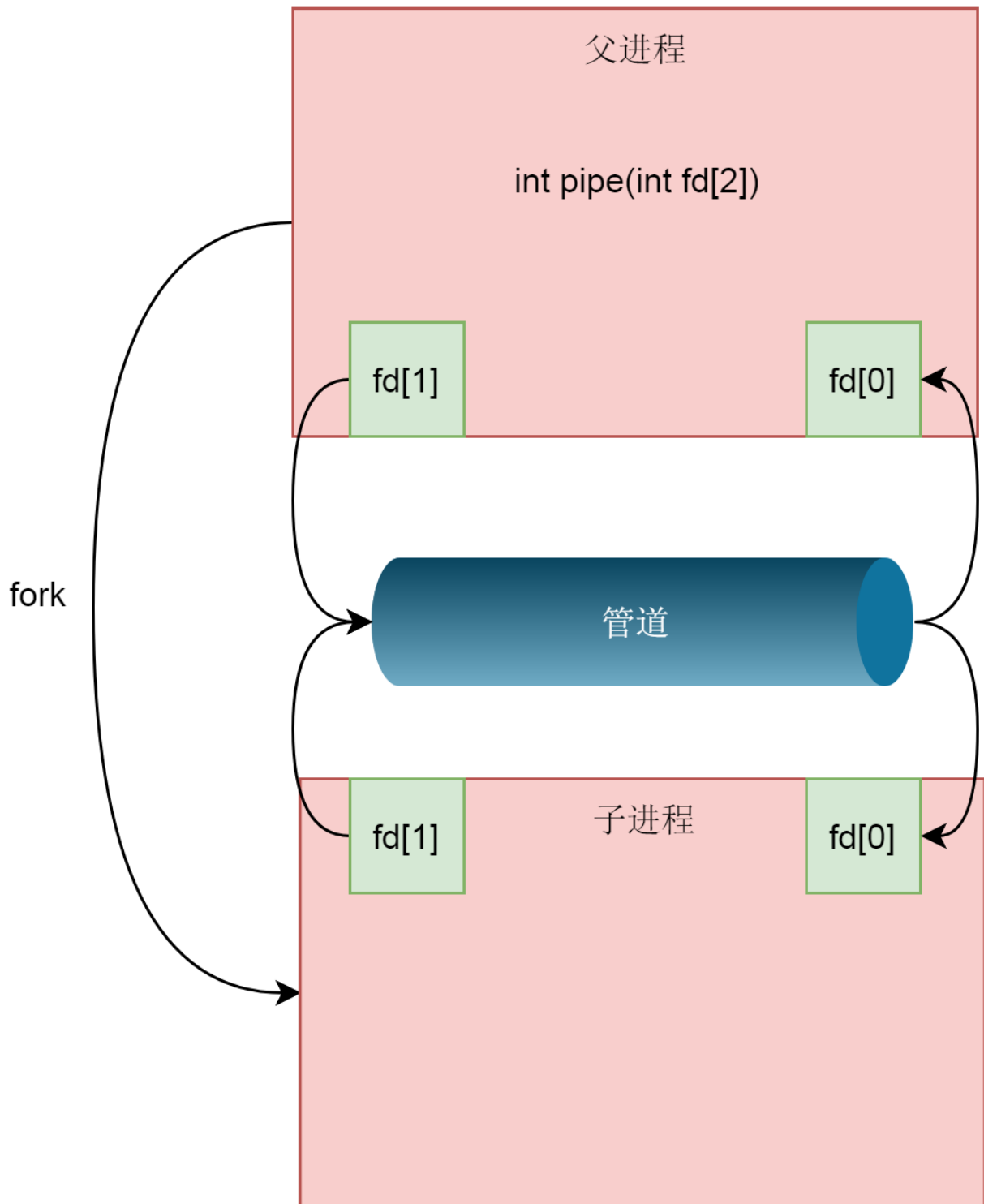
 复制代码

```
1 const struct file_operations pipefifo_fops = {
2     .open          = fifo_open,
3     .llseek        = no_llseek,
4     .read_iter     = pipe_read,
5     .write_iter    = pipe_write,
6     .poll          = pipe_poll,
7     .unlocked_ioctl = pipe_ioctl,
8     .release       = pipe_release,
9     .fsync         = pipe_fsync,
10 };
```

我们回到 `create_pipe_files` 函数，创建完了 `inode`，还需创建一个 `dentry` 和他对应。`dentry` 和 `inode` 对应好了，我们就要开始创建 `struct file` 对象了。先创建用于写入的，对应的操作为 `pipefifo_fops`；再创建读取的，对应的操作也为 `pipefifo_fops`。然后把 `private_data` 设置为 `pipe_inode_info`。这样从 `struct file` 这个层级上，就能直接操作底层的读写操作。

至此，一个匿名管道就创建成功了。如果对于 `fd[1]` 写入，调用的是 `pipe_write`，向 `pipe_buffer` 里面写入数据；如果对于 `fd[0]` 的读入，调用的是 `pipe_read`，也就是从 `pipe_buffer` 里面读取数据。

但是这个时候，两个文件描述符都是在一个进程里面的，并没有起到进程间通信的作用，怎么样才能使得管道是跨两个进程的呢？还记得创建进程调用的 `fork` 吗？在这里面，创建的子进程会复制父进程的 `struct files_struct`，在这里面 `fd` 的数组会复制一份，但是 `fd` 指向的 `struct file` 对于同一个文件还是只有一份，这样就做到了，两个进程各有两个 `fd` 指向同一个 `struct file` 的模式，两个进程就可以通过各自的 `fd` 写入和读取同一个管道文件实现跨进程通信了。

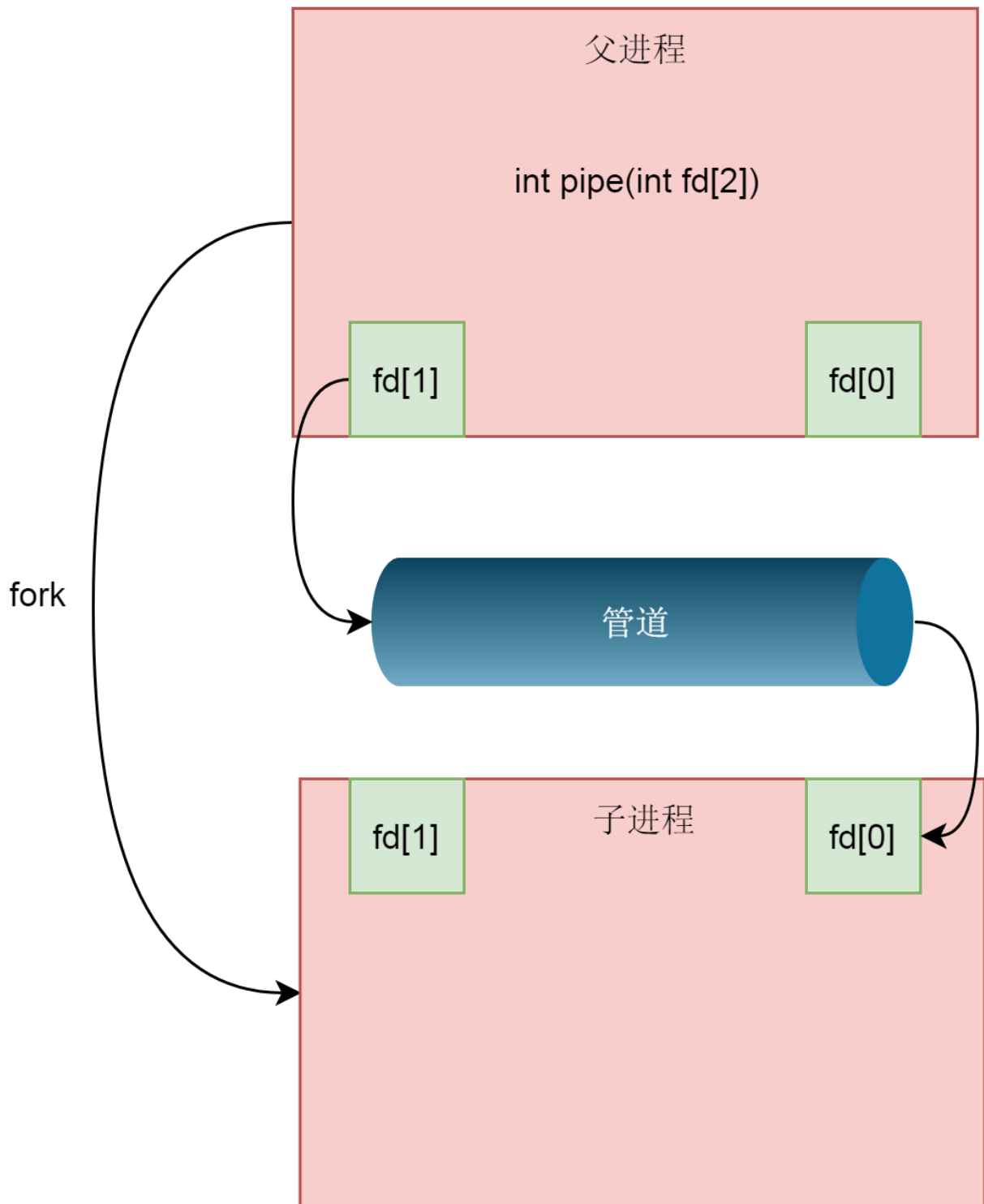


由于管道只能一端写入，另一端读出，所以上面的这种模式会造成混乱，因为父进程和子进程都可以写入，也都可以读出，通常的方法是父进程关闭读取的 fd，只保留写入的 fd，而子进程关闭写入的 fd，只保留读取的 fd，如果需要双向通行，则应该创建两个管道。

一个典型的使用管道在父子进程之间的通信代码如下：

```
1 #include <unistd.h>
2 #include <fcntl.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <errno.h>
6 #include <string.h>
7
8 int main(int argc, char *argv[])
9 {
10     int fds[2];
11     if (pipe(fds) == -1)
12         perror("pipe error");
13
14     pid_t pid;
15     pid = fork();
16     if (pid == -1)
17         perror("fork error");
18
19     if (pid == 0){
20         close(fds[0]);
21         char msg[] = "hello world";
22         write(fds[1], msg, strlen(msg) + 1);
23         close(fds[1]);
24         exit(0);
25     } else {
26         close(fds[1]);
27         char msg[128];
28         read(fds[0], msg, 128);
29         close(fds[0]);
30         printf("message : %s\n", msg);
31         return 0;
32     }
33 }
```

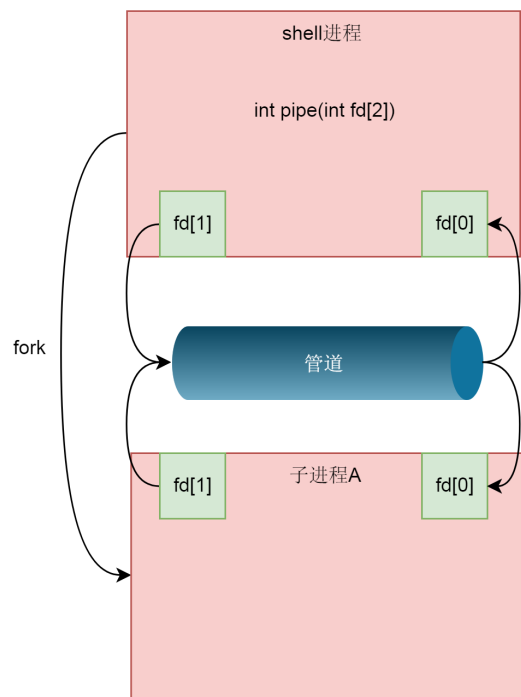




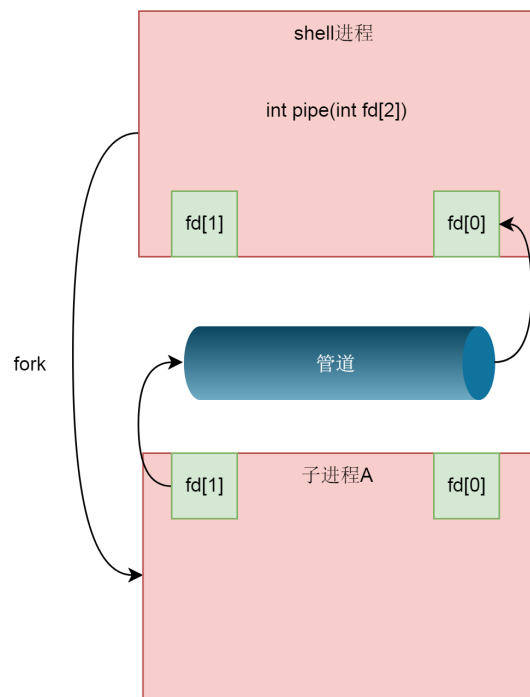
到这里，我们仅仅解析了使用管道进行父子进程之间的通信，但是我们在 shell 里面的不是这样的。在 shell 里面运行 `A|B` 的时候，A 进程和 B 进程都是 shell 创建出来的子进程，A 和 B 之间不存在父子关系。

不过，有了上面父子进程之间的管道这个基础，实现 A 和 B 之间的管道就方便多了。

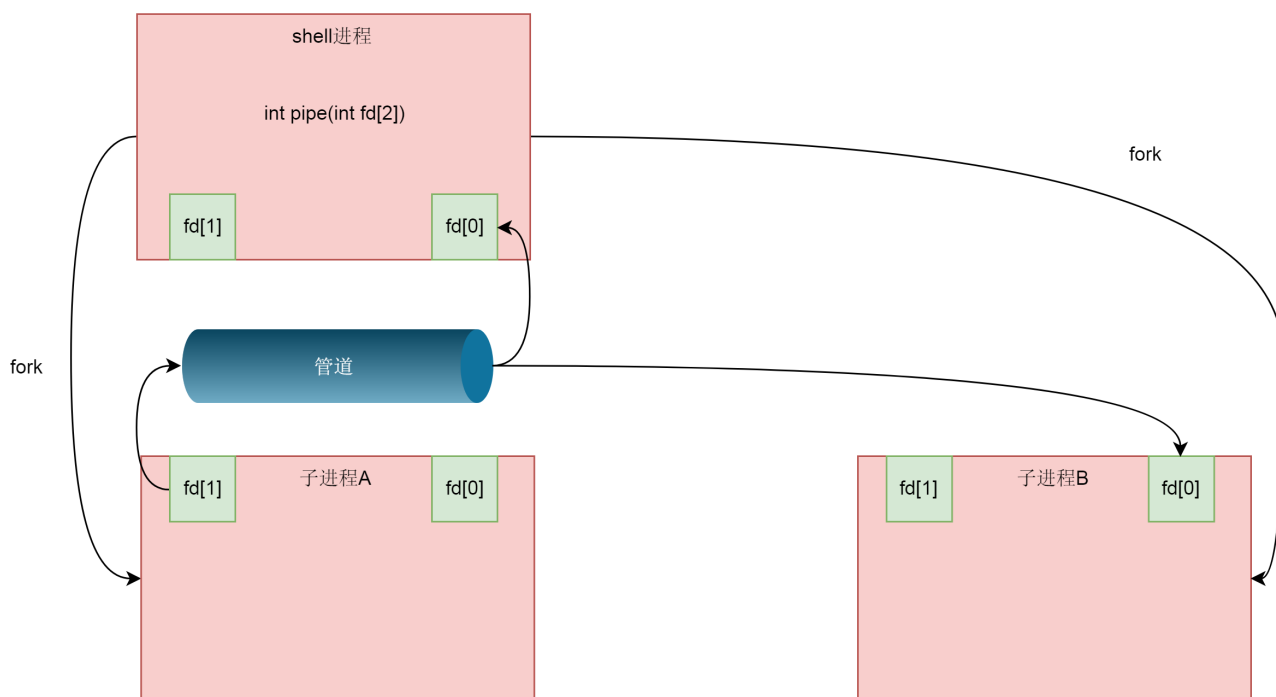
我们首先从 shell 创建子进程 A，然后在 shell 和 A 之间建立一个管道，其中 shell 保留读取端，A 进程保留写入端，然后 shell 再创建子进程 B。这又是一次 fork，所以，shell 里面保留的读取端的 fd 也被复制到了子进程 B 里面。这个时候，相当于 shell 和 B 都保留读取端，只要 shell 主动关闭读取端，就变成了一管道，写入端在 A 进程，读取端在 B 进程。



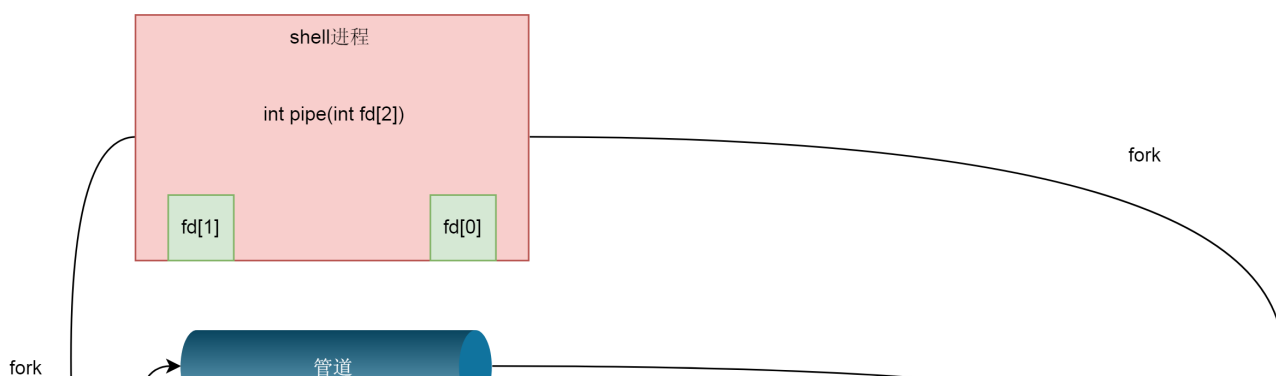
(1)

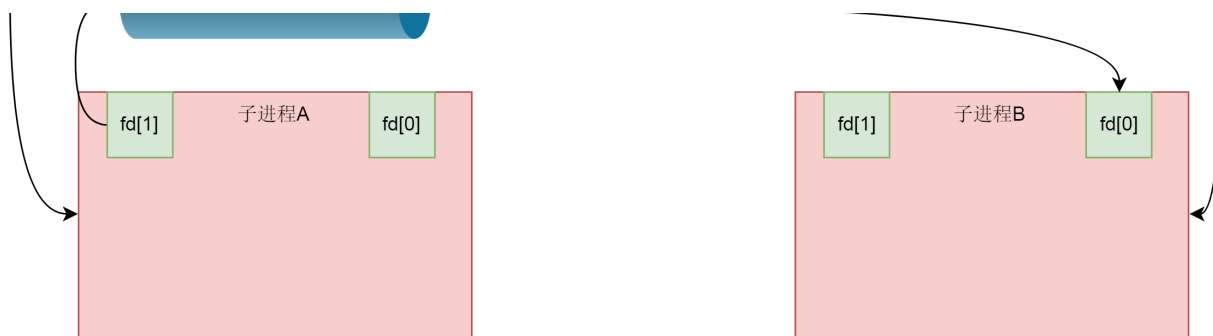


(2)



(3)





(4)

接下来我们要做的事情就是，将这个管道的两端和输入输出关联起来。这就要用到 `dup2` 系统调用了。

复制代码

```
1 int dup2(int oldfd, int newfd);
```

这个系统调用，将老的文件描述符赋值给新的文件描述符，让 `newfd` 的值和 `oldfd` 一样。

我们还是回忆一下，在 `files_struct` 里面，有这样一个表，下标是 `fd`，内容指向一个打开的文件 `struct file`。

复制代码

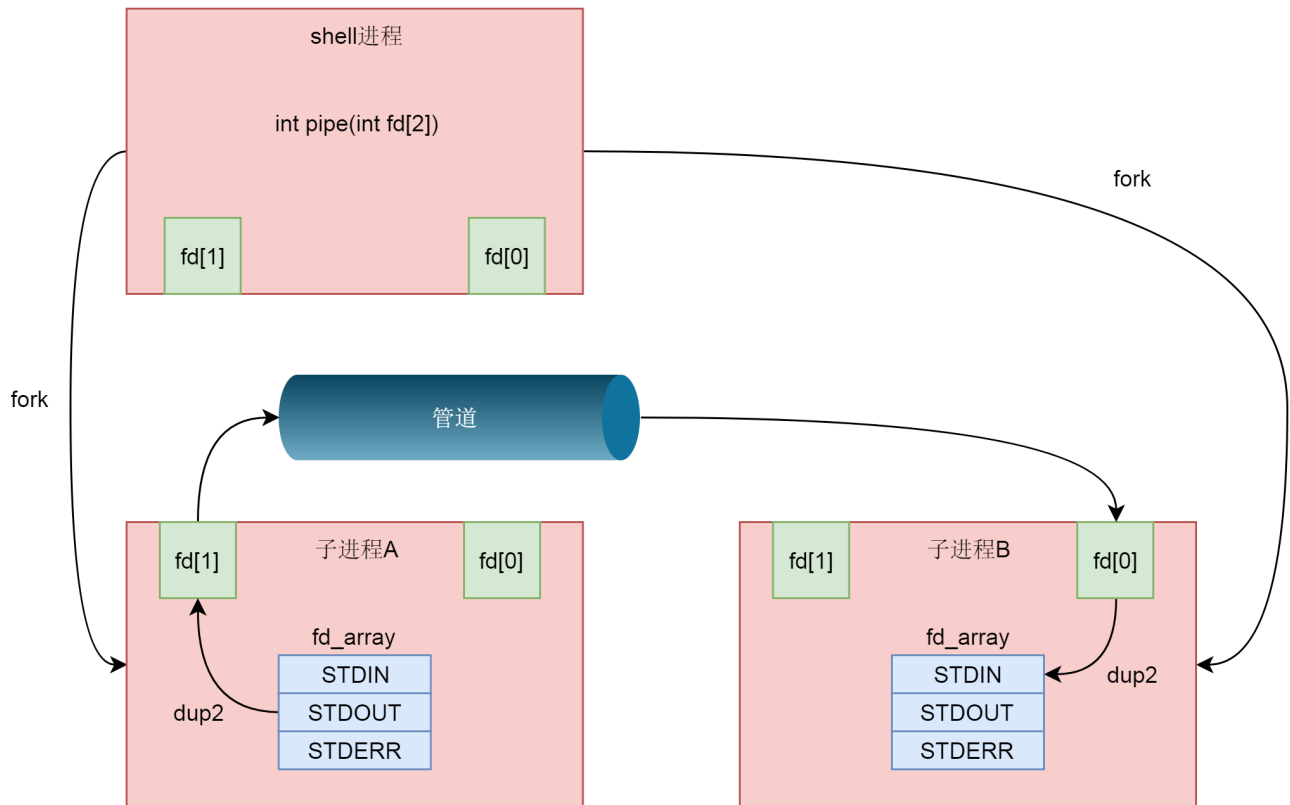
```
1 struct files_struct {
2     struct file __rcu * fd_array[NR_OPEN_DEFAULT];
3 }
```

在这个表里面，前三项是定下来的，其中第零项 `STDIN_FILENO` 表示标准输入，第一项 `STDOUT_FILENO` 表示标准输出，第三项 `STDERR_FILENO` 表示错误输出。

在 A 进程中，写入端可以做这样的操作：`dup2(fd[1],STDOUT_FILENO)`，将 `STDOUT_FILENO`（也即第一项）不再指向标准输出，而是指向创建的管道文件，那么以后往标准输出写入的任何东西，都会写入管道文件。

在 B 进程中，读取端可以做这样的操作，`dup2(fd[0],STDIN_FILENO)`，将 `STDIN_FILENO` 也即第零项不再指向标准输入，而是指向创建的管道文件，那么以后从标准输入读取的任何东西，都来自于管道文件。

至此，我们才将 `A|B` 的功能完成。




为了模拟 `A|B` 的情况，我们可以将前面的那一段代码，进一步修改成为下面这样：

复制代码

```
1 #include <unistd.h>
2 #include <fcntl.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <errno.h>
6 #include <string.h>
7
8 int main(int argc, char *argv[])
9 {
10     int fds[2];
11     if (pipe(fds) == -1)
12         perror("pipe error");
13
14     pid_t pid;
15     pid = fork();
```


Glibc 的 `mkfifo` 函数会调用 `mknodat` 系统调用，还记得咱们学字符设备的时候，创建一个字符设备的时候，也是调用的 `mknod`。这里命名管道也是一个设备，因而我们也用 `mknod`。

 复制代码

```

1  SYSCALL_DEFINE4(mknodat, int, dfd, const char __user *, filename, umode_t, mode, unsigned
2  {
3      struct dentry *dentry;
4      struct path path;
5      unsigned int lookup_flags = 0;
6      .....
7  retry:
8      dentry = user_path_create(dfd, filename, &path, lookup_flags);
9      .....
10     switch (mode & S_IFMT) {
11     .....
12         case S_IFIFO: case S_IFSOCK:
13             error = vfs_mknod(path.dentry->d_inode, dentry, mode, 0);
14             break;
15     }
16     .....
17 }

```

对于 `mknod` 的解析，我们在字符设备那一节已经解析过了，先是通过 `user_path_create` 对于这个管道文件创建一个 `dentry`，然后因为是 `S_IFIFO`，所以调用 `vfs_mknod`。由于这个管道文件是创建在一个普通文件系统上的，假设是在 `ext4` 文件上，于是 `vfs_mknod` 会调用 `ext4_dir_inode_operations` 的 `mknod`，也即会调用 `ext4_mknod`。

 复制代码

[illegible]

```


15     handle = ext4_journal_current_handle();
16     if (!IS_ERR(inode)) {
17         init_special_inode(inode, inode->i_mode, rdev);
18         inode->i_op = &ext4_special_inode_operations;
19         err = ext4_add_nondir(handle, dentry, inode);
20         if (!err && IS_DIRSYNC(dir))
21             ext4_handle_sync(handle);
22     }
23     if (handle)
24         ext4_journal_stop(handle);
25     .....
26 }
27
28 #define ext4_new_inode_start_handle(dir, mode, qstr, goal, owner, \
29                                     type, nblocks) \
30     __ext4_new_inode(NULL, (dir), (mode), (qstr), (goal), (owner), \
31                       0, (type), __LINE__, (nblocks))
32
33 void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
34 {
35     inode->i_mode = mode;
36     if (S_ISCHR(mode)) {
37         inode->i_fop = &def_chr_fops;
38         inode->i_rdev = rdev;
39     } else if (S_ISBLK(mode)) {
40         inode->i_fop = &def_blk_fops;
41         inode->i_rdev = rdev;
42     } else if (S_ISFIFO(mode))
43         inode->i_fop = &pipefifo_fops;
44     else if (S_ISSOCK(mode))
45         ; /* leave it no_open_fops */
46     else
47         .....
48 }

```

在 `ext4_mknod` 中，`ext4_new_inode_start_handle` 会调用 `__ext4_new_inode`，在 `ext4` 文件系统上真的创建一个文件，但是会调用 `init_special_inode`，创建一个内存中特殊的 `inode`，这个函数我们在字符设备文件中也遇到过，只不过当时 `inode` 的 `i_fop` 指向的是 `def_chr_fops`，这次换成管道文件了，`inode` 的 `i_fop` 变成指向 `pipefifo_fops`，这一点和匿名管道是一样的。

这样，管道文件就创建完毕了。

接下来，要打开这个管道文件，我们还是会调用文件系统的 open 函数。还是沿着文件系统的调用方式，一路调用到 pipefifo_fops 的 open 函数，也就是 fifo_open。

 复制代码

```
1 static int fifo_open(struct inode *inode, struct file *filp)
2 {
3     struct pipe_inode_info *pipe;
4     bool is_pipe = inode->i_sb->s_magic == PIPEFS_MAGIC;
5     int ret;
6     filp->f_version = 0;
7
8     if (inode->i_pipe) {
9         pipe = inode->i_pipe;
10        pipe->files++;
11    } else {
12        pipe = alloc_pipe_info();
13        pipe->files = 1;
14        inode->i_pipe = pipe;
15        spin_unlock(&inode->i_lock);
16    }
17    filp->private_data = pipe;
18    filp->f_mode &= (FMODE_READ | FMODE_WRITE);
19
20    switch (filp->f_mode) {
21    case FMODE_READ:
22        pipe->r_counter++;
23        if (pipe->readers++ == 0)
24            wake_up_partner(pipe);
25        if (!is_pipe && !pipe->writers) {
26            if ((filp->f_flags & O_NONBLOCK)) {
27                filp->f_version = pipe->w_counter;
28            } else {
29                if (wait_for_partner(pipe, &pipe->w_counter))
30                    goto err_rd;
31            }
32        }
33        break;
34    case FMODE_WRITE:
35        pipe->w_counter++;
36        if (!pipe->writers++)
37            wake_up_partner(pipe);
38        if (!is_pipe && !pipe->readers) {
39            if (wait_for_partner(pipe, &pipe->r_counter))
40                goto err_wr;
41        }
42        break;
43    case FMODE_READ | FMODE_WRITE:
44        pipe->readers++;
45        pipe->writers++;
46        pipe->r_counter++;
```

```
47         pipe->w_counter++;
48         if (pipe->readers == 1 || pipe->writers == 1)
49             wake_up_partner(pipe);
50         break;
51     .....
52     }
53     .....
54 }
```

在 `fifo_open` 里面，创建 `pipe_inode_info`，这一点和匿名管道也是一样的。这个结构里面有个成员是 `struct pipe_buffer *bufs`。我们可以知道，**所谓的命名管道，其实是也是内核里面的一串缓存。**

接下来，对于命名管道的写入，我们还是会调用 `pipefifo_fops` 的 `pipe_write` 函数，向 `pipe_buffer` 里面写入数据。对于命名管道的读入，我们还是会调用 `pipefifo_fops` 的 `pipe_read`，也就是从 `pipe_buffer` 里面读取数据。

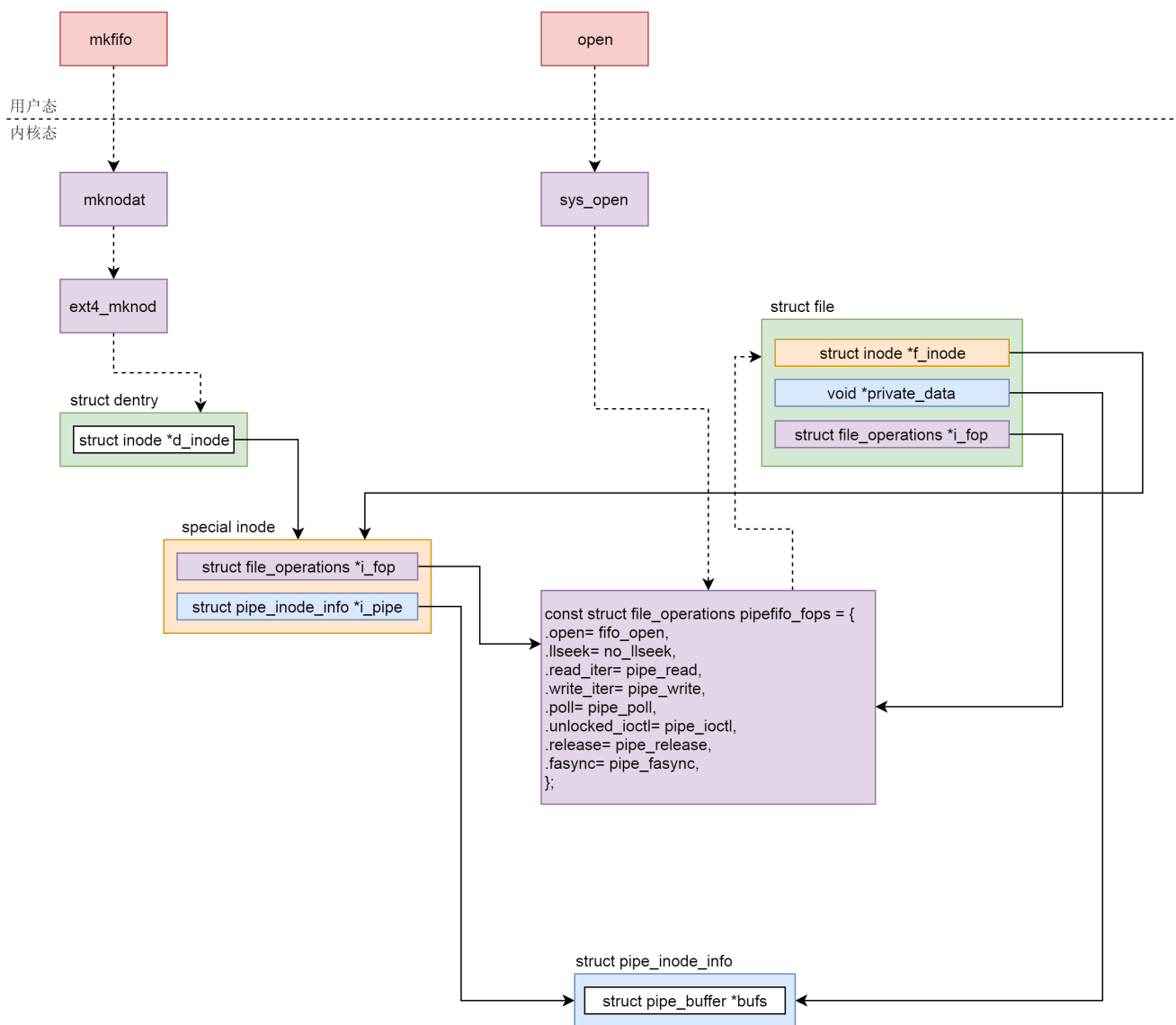
总结时刻

无论是匿名管道，还是命名管道，在内核都是一个文件。只要是文件就要有一个 `inode`。这里我们又用到了特殊 `inode`、字符设备、块设备，其实都是这种特殊的 `inode`。

在这种特殊的 `inode` 里面，`file_operations` 指向管道特殊的 `pipefifo_fops`，这个 `inode` 对应内存里面的缓存。

当我们用文件的 `open` 函数打开这个管道设备文件的时候，会调用 `pipefifo_fops` 里面的方法创建 `struct file` 结构，他的 `inode` 指向特殊的 `inode`，也对应内存里面的缓存，`file_operations` 也指向管道特殊的 `pipefifo_fops`。

写入一个 `pipe` 就是从 `struct file` 结构找到缓存写入，读取一个 `pipe` 就是从 `struct file` 结构找到缓存读出。



课堂练习

上面创建匿名管道的程序，你一定要运行一下，然后试着通过 `strace` 查看自己写的程序的系统调用，以及直接在命令行使用匿名管道的系统调用，做一个比较。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

趣谈 Linux 操作系统


像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 38 | 信号（下）：项目组A完成了，如何及时通知项目组B？

下一篇 40 | IPC（上）：不同项目组之间抢资源，如何协调？

精选留言 (3)

 写留言



Sharry

2019-06-27

- 匿名管道: 只能在管道创建进程及其后代之间通信
 - 通过 pipe 系统调用创建
 - ****inode 由特殊的文件系统 pipefs 创建****
 - ****inode 关联的 fos 为 pipefifo_fops****
- 命名管道: 通过管道文件名, 可以在任意进程之间通信...



石维康

2019-06-26

在ext4_mknod函数里调用init_special_inode时传入的是上一步

ext4_new_inode_start_handle返回的inode。为什么文中还会说"但是会调用init_special_inode，创建一个内存中特殊的 inode"？
在init_special_inode中也没有看到创建虚拟inode的地方？



有铭

2019-06-26

管道更像是流处理，还是批处理？

