第45回 | 解析并执行 shell 命令

Original 闪客 低并发编程 2022-07-31 17:30 Posted on 北京

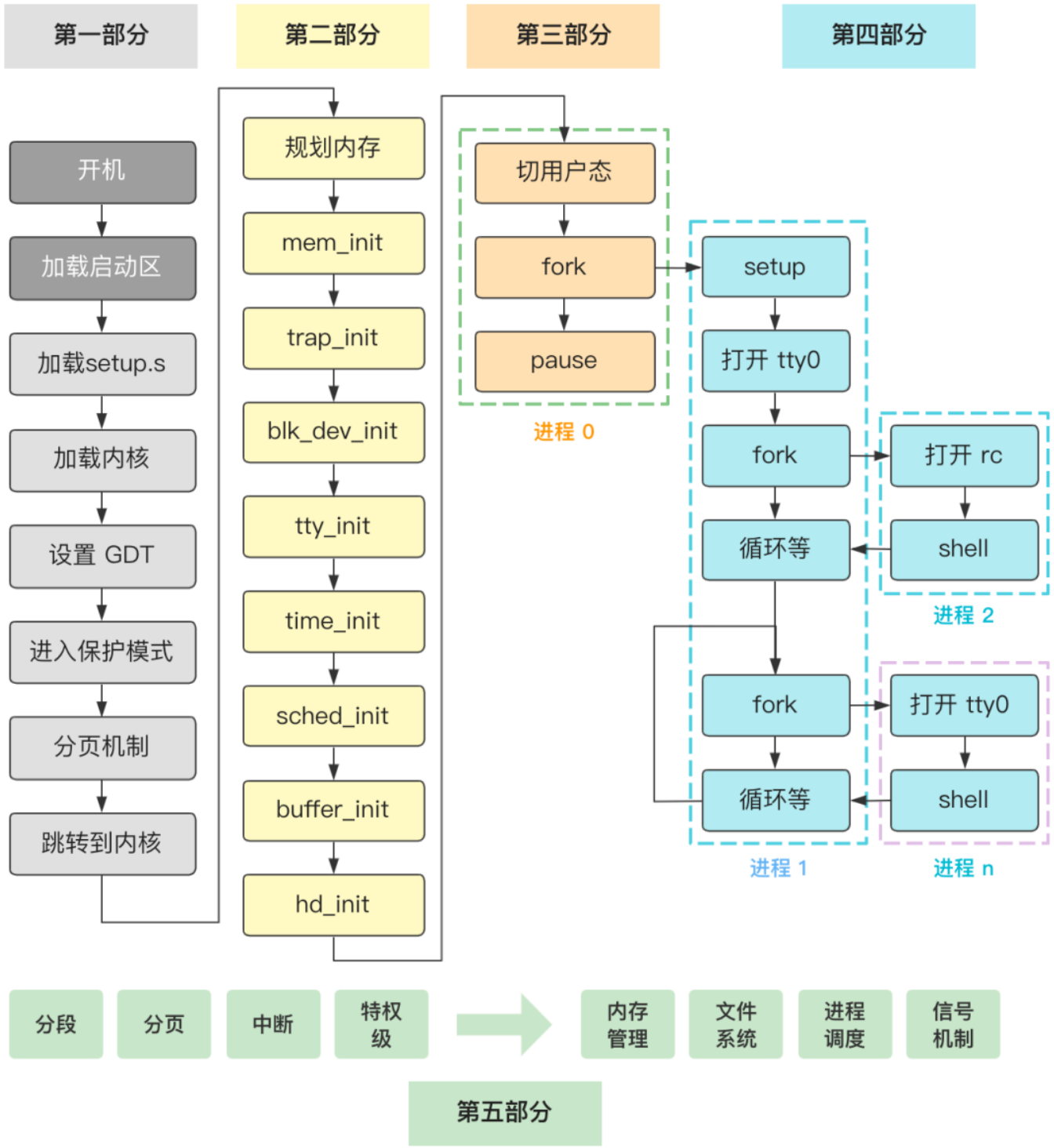
收录于合集
#操作系统源码 52 #一条shell命令的执行 8

新读者看这里，老读者直接跳过。

本系列会以一个读小说的心态，从开机启动后的代码执行顺序，带着大家阅读和赏析 Linux 0.11 全部核心代码，了解操作系统的技术细节和设计思想。

本系列的 GitHub 地址如下，希望给个 star 以示鼓励（文末**阅读原文**可直接跳转，也可以将下面的链接复制到浏览器里打开）
https://github.com/sunym1993/flash-linux0.11-talk

本回的内容属于第五部分。



你会跟着我一起，看着一个操作系统从啥都没有开始，一步一步最终实现它复杂又精巧的设计，读完这个系列后希望你能发出感叹，原来操作系统源码就是这破玩意。

以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

第一部分 进入内核前的苦力活

------- 正文开始 -------

新建一个非常简单的 info.txt 文件。

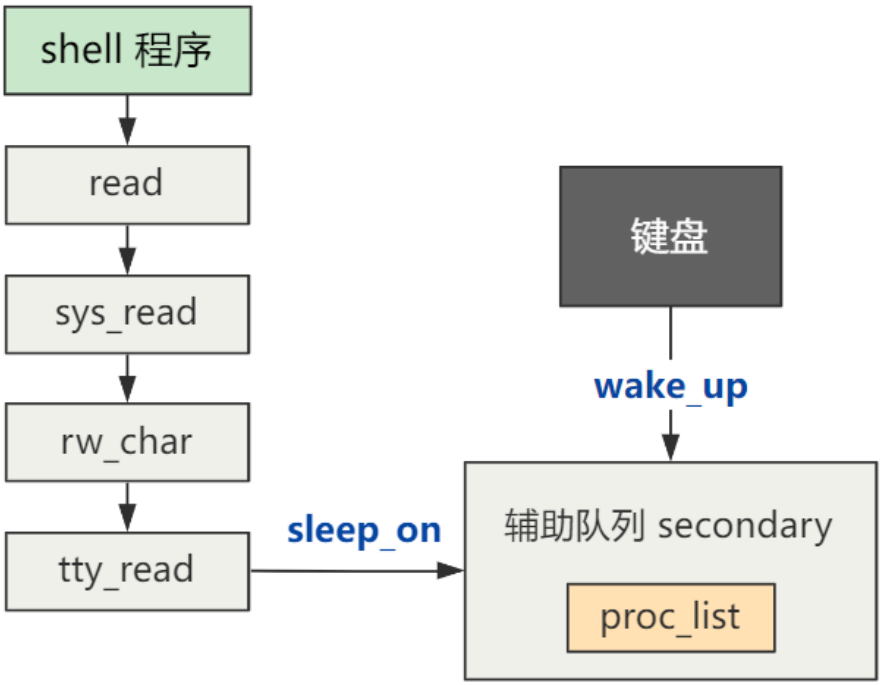
```
name:flash
age:28
language:java
```

在命令行输入一条十分简单的命令。

```
[root@linux0.11] cat info.txt | wc -l
3
```

这条命令的意思是读取刚刚的 info.txt 文件，输出它的行数。

在上一回中，我们讲述了进程在读取你的命令字符串时，可能经历的进程的阻塞与唤醒，也即 Linux 0.11 中的 sleep_on 与 wake_up 函数。



接下来，shell 程序就该解析并执行这条命令了。

```
// xv6-public sh.c
int main(void) {
    static char buf[100];
    // 读取命令
    while(getcmd(buf, sizeof(buf)) >= 0){
        // 创建新进程
        if(fork() == 0)
            // 执行命令
            runcmd(parsecmd(buf));
        // 等待进程退出
        wait();
    }
}
```

也就是上述函数中的 runcmd 命令。

首先 parsecmd 函数会将读取到 buf 的字符串命令做解析，生成一个 cmd 结构的变量，传入 runcmd 函数中。

```
// xv6-public sh.c
void runcmd(struct cmd *cmd) {
    ...
    switch(cmd->type) {
        ...
        case EXEC:
        ecmd = (struct execcmd*)cmd;
        ...
        exec(ecmd->argv[0], ecmd->argv);
        ...
        break;

        case REDIR: ...
        case LIST: ...
        case PIPE: ...
        case BACK: ...
    }
}
```

然后就如上述代码所示，根据 cmd 的 type 字段，来判断应该如何执行这个命令。

比如最简单的，就是直接执行，也即 **EXEC**。

如果命令中有分号；说明是多条命令的组合，那么就当作 **LIST** 拆分成多条命令依次执行。

如果命令中有竖线 | 说明是管道命令，那么就当作 **PIPE** 拆分成两个并发的命令，同时通过管道串联起输入端和输出端，来执行。

我们这个命令，很显然就是个管道命令。

```
[root@linux0.11] cat info.txt | wc -l
```

管道理解起来非常简单，但是实现细节却是略微复杂。

所谓管道，也就是上述命令中的 |，实现的就是将 | **左边的程序的输出**（stdout）作为 | **右边的程序的输入**（stdin），就这么简单。

那我们看看，它是如何实现的，我们走到 runcmd 方法中的 PIPE 这个分支里，也就是当解析出输入的命令是一个管道命令时，所应该做的处理。

```
// xv6-public sh.c
void runcmd(struct cmd *cmd) {
    ...
    int p[2];
    ...
    case PIPE:
        pcmd = (struct pipecmd*)cmd;
        pipe(p);
        if(fork() == 0) {
            close(1);
            dup(p[1]);
            close(p[0]);
            close(p[1]);
            runcmd(pcmd->left);
        }
        if(fork() == 0) {
            close(0);
            dup(p[0]);
            close(p[0]);
            close(p[1]);
            runcmd(pcmd->right);
        }
        close(p[0]);
        close(p[1]);
        wait(0);
        wait(0);
        break;
    ...
}
```

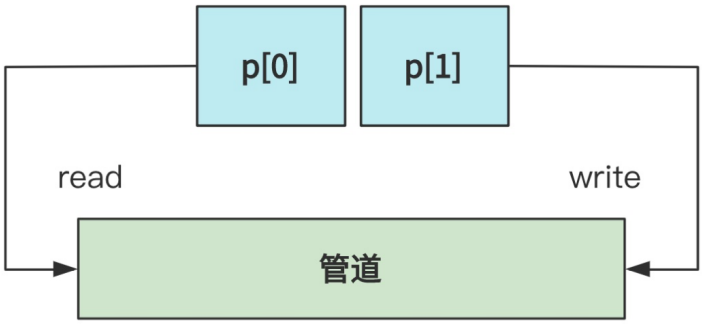
首先，我们构造了一个大小为 2 的数组 **p**，然后作为 pipe 的参数传了进去。

这个 pipe 函数，最终会调用到系统调用的 **sys_pipe**，我们先不看代码，通过 man page 查看 pipe 的用法与说明。

DESCRIPTION top

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see pipe(7).

可以看到，pipe 就是创建一个**管道**，将传入数组 p 的 p[0] 指向这个管道的读口，p[1] 指向这个管道的写口，画图就是这样子的。



当然，这个管道的本质是一个**文件**，但是是属于**管道类型的文件**，所以它的本质的本质实际上是一块**内存**。

这块内存被当作管道文件对上层提供了像访问文件一样的读写接口，只不过其中一个进程只能读，另一个进程只能写，所以再次抽象一下就像一个管道一样，数据从一端流向了另一段。

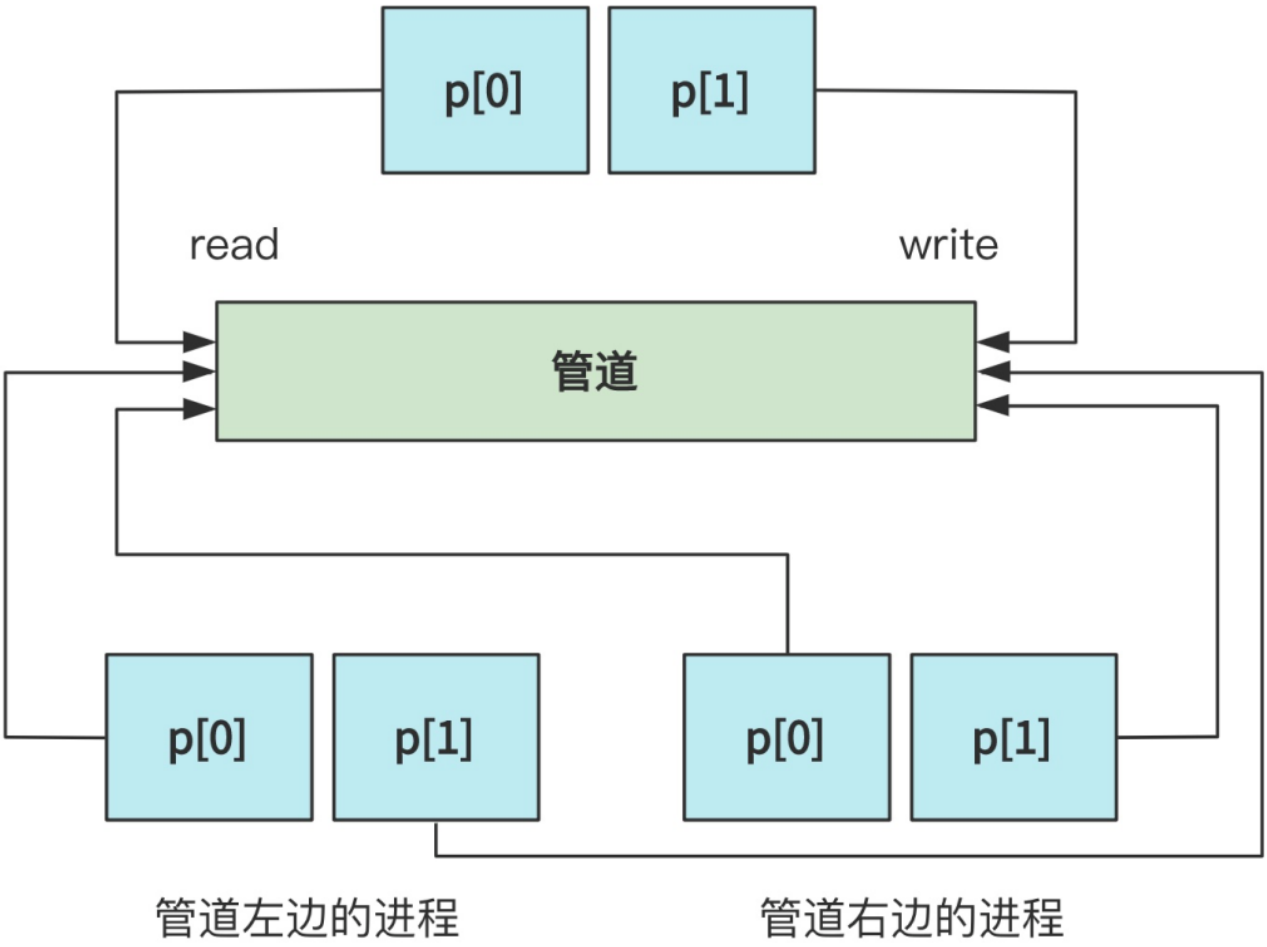
你说它是内存也行，说它是文件也行，说它是管道也行，看你抽象到那一层了，这个之后再展开细讲，先让你迷糊迷糊。

回过头看程序。

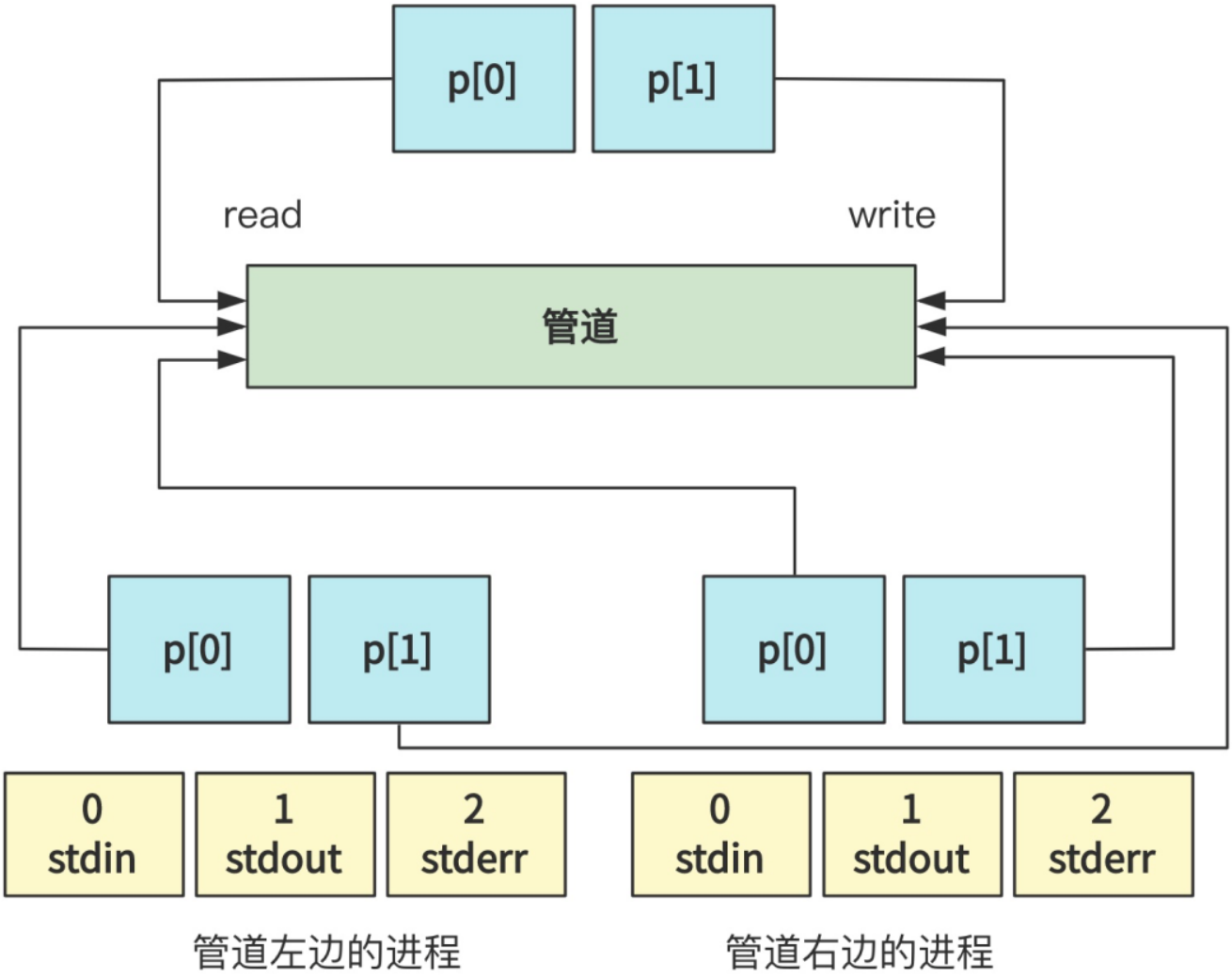
```
// xv6-public sh.c
void runcmd(struct cmd *cmd) {
    ...
    int p[2];
    ...
    case PIPE:
        pcmd = (struct pipecmd*)cmd;
        pipe(p);
        if(fork() == 0) {
            close(1);
            dup(p[1]);
            close(p[0]);
            close(p[1]);
            runcmd(pcmd->left);
        }
        if(fork() == 0) {
            close(0);
            dup(p[0]);
            close(p[0]);
            close(p[1]);
            runcmd(pcmd->right);
        }
        close(p[0]);
        close(p[1]);
        wait(0);
        wait(0);
        break;
    ...
}
```

在调用完 pipe 搞出了这样一个管道并绑定了 p[0] 和 p[1] 之后，又分别通过 fork 创建了两个进程，其中第一个进程执行了**管道左边的程序**，第二个进程执行了**管道右边的程序。**

由于 fork 出的子进程会原封不动复制父进程打开的文件描述符，所以目前的状况如下图所示。



当然，由于每个进程，一开始都打开了 0 号标准输入文件描述符，1 号标准输出文件描述符和 2 号标准错误输出文件描述符，所以目前把文件描述符都展开就是这个样子。（父进程的我就省略了）

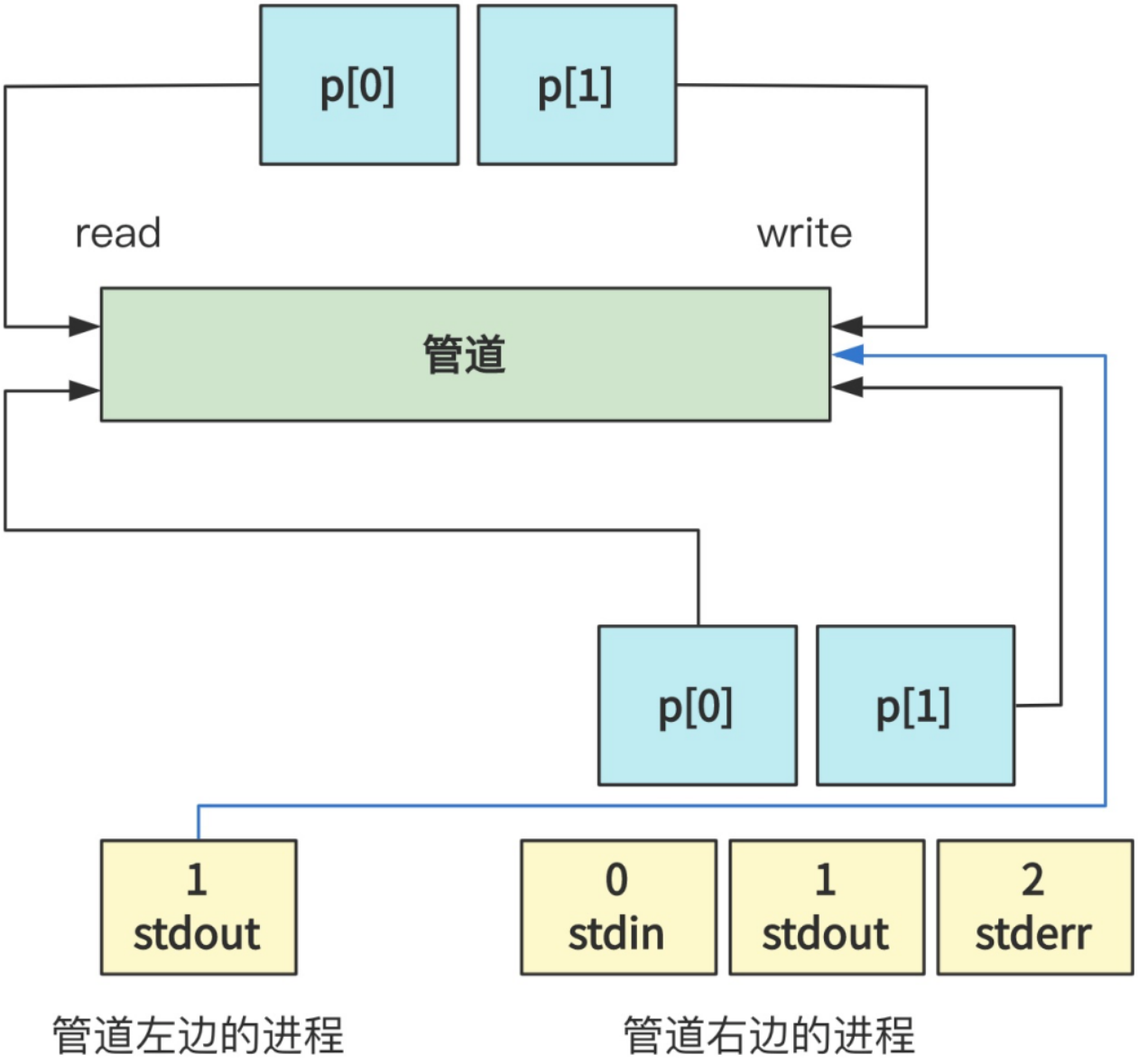


现在这个线条很乱，不过没关系，看代码。左边进程随后进行了如下操作。

```
// fs/pipe.c
...
if(fork() == 0) {
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->left);
}
...
```

即**关闭（close）**了 1 号标准输出文件描述符，**复制（dup）**了 p[1] 并填充在了 1 号文件描述符上（因为刚刚关闭后空缺出来了），然后又把 p[0] 和 p[1] 都**关闭（close）**了。

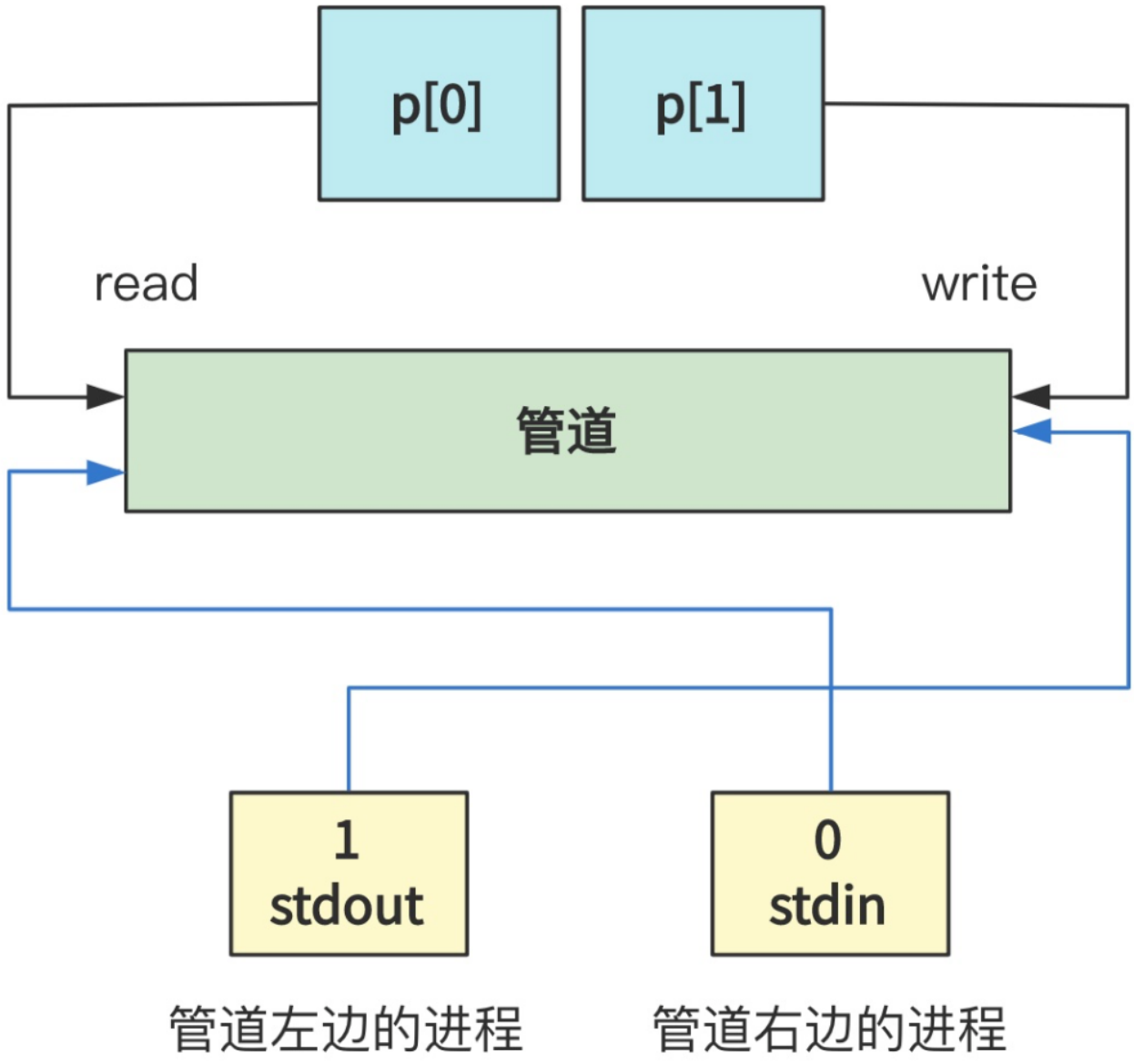
你再读读这段话，最终的效果就是，将 1 号文件描述符，也就是标准输出，指向了 p[1] 管道的写口，也就是 p[1] 原来所指向的地方。



同理，右边进程也进行了类似的操作。

```
// fs/pipe.c
...
if(fork() == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->right);
}
...
```

只不过，最终是将 0 号标准输入指向了管道的读口。

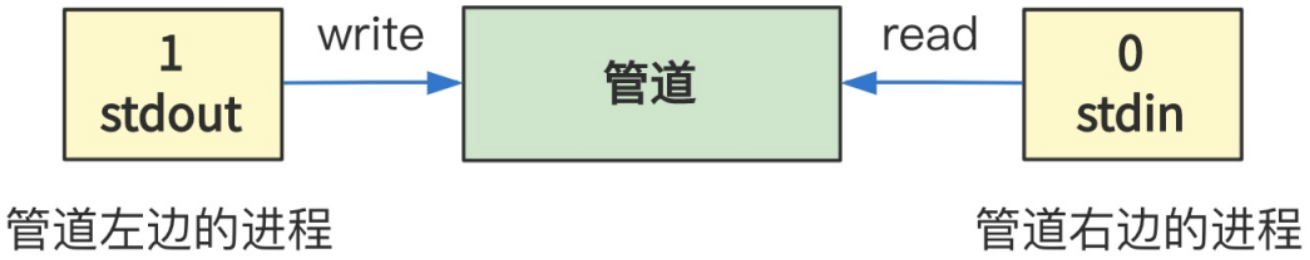


这是两个子进程的操作，再看父进程。

```
// xv6-public sh.c
void runcmd(struct cmd *cmd) {
    ...
    pipe(p);
    if(fork() == 0) {...}
    if(fork() == 0) {...}
    // 父进程
    close(p[0]);
    close(p[1]);
    ...
}
```

你没有看错，**父进程仅仅是将 p[0] 和 p[1] 都关闭掉了**，也就是说，父进程执行的 pipe，仅仅是为两个子进程申请的文件描述符，对于自己来说并没有用处。

那么我们忽略父进程，最终，**其实就是创建了两个进程，左边的进程的标准输出指向了管道（写），右边的进程的标准输入指向了同一个管道（读）**，看起来就是下面的样子。



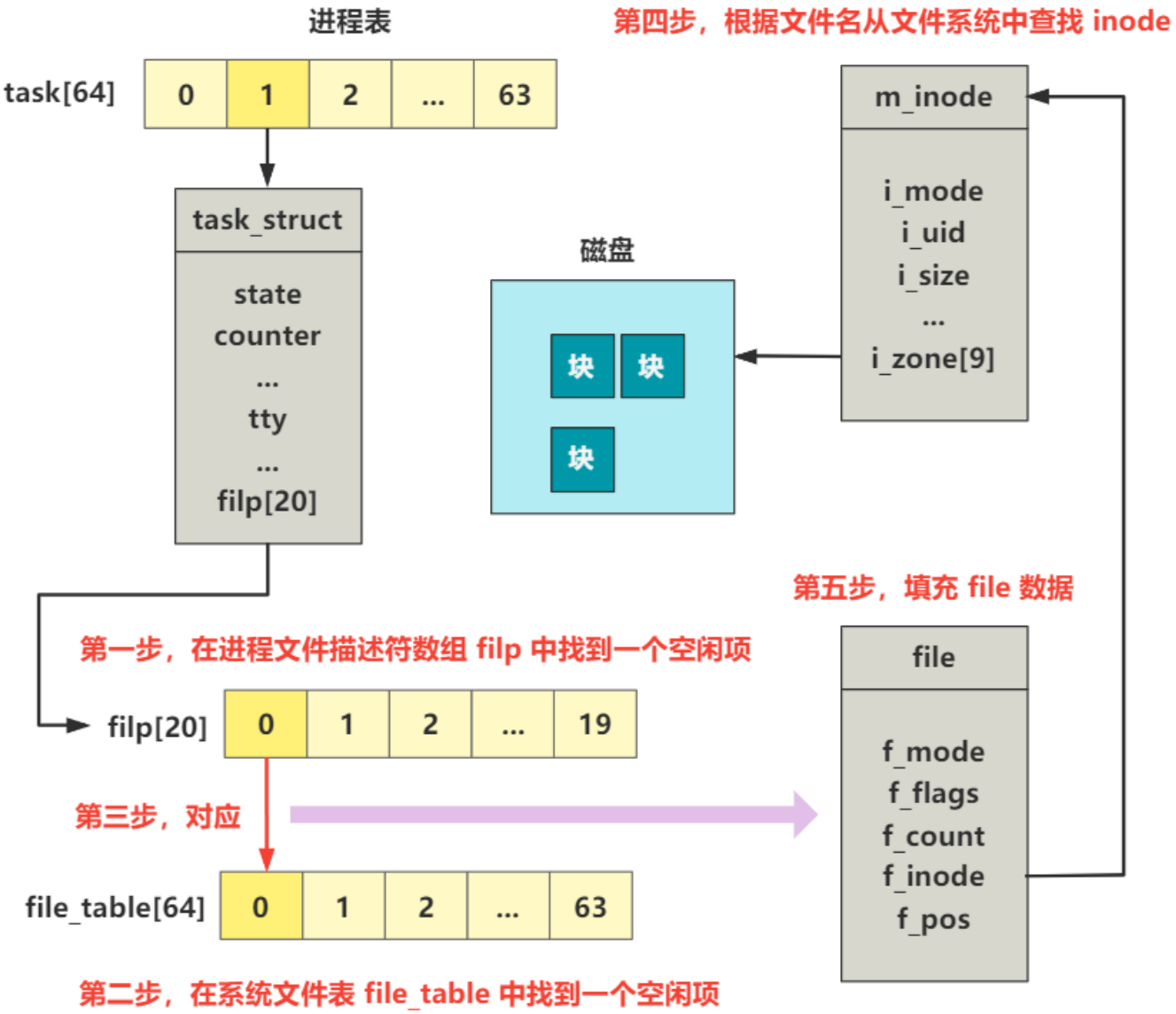
而管道的本质就是一个文件，只不过是管道类型的文件，再本质就是一块内存。所以这一顿操作，其实就是把两个进程的文件描述符，指向了一个文件罢了，就这么点事情。

那么此时，再让我们看看 sys_pipe 函数的细节。

```
// fs/pipe.c
int sys_pipe(unsigned long * fildes) {
    struct m_inode * inode;
    struct file * f[2];
    int fd[2];

    for(int i=0,j=0; j<2 && i<NR_FILE; i++)
        if (!file_table[i].f_count)
            (f[j++]=i+file_table)->f_count++;
    ...
    for(int i=0,j=0; j<2 && i<NR_OPEN; i++)
        if (!current->filp[i]) {
            current->filp[ fd[j]=i ] = f[j];
            j++;
        }
    ...
    if (!(inode=get_pipe_inode())) {
        current->filp[fd[0]] = current->filp[fd[1]] = NULL;
        f[0]->f_count = f[1]->f_count = 0;
        return -1;
    }
    f[0]->f_inode = f[1]->f_inode = inode;
    f[0]->f_pos = f[1]->f_pos = 0;
    f[0]->f_mode = 1;       /* read */
    f[1]->f_mode = 2;       /* write */
    put_fs_long(fd[0],0+fildes);
    put_fs_long(fd[1],1+fildes);
    return 0;
}
```

不出我们所料，和**进程打开一个文件**的步骤是差不多的，可以回顾下 第33回 | 打开终端设备文件 这一回，下图是进程打开一个文件时的步骤。



而 pipe 方法与之相同的是，都是从进程中的**文件描述符表 filp** 数组和系统的**文件系统表 file_table** 数组中寻找空闲项并绑定。

不同的是，打开一个文件的前提是文件已经存在了，根据文件名找到这个文件，并提取出它的 inode 信息，填充好 file 数据。

而 pipe 方法中并不是打开一个已存在的文件，而是**创建一个新的管道类型的文件**，具体是通过 **get_pipe_inode** 方法，返回一个 inode 结构。然后，填充了两个 file 结构的数据，都指向了这个 inode，其中一个的 f_mode 为 1 也就是写，另一个是 2 也就是读。（f_mode 为文件的操作模式属性，也就是 RW 位的值）

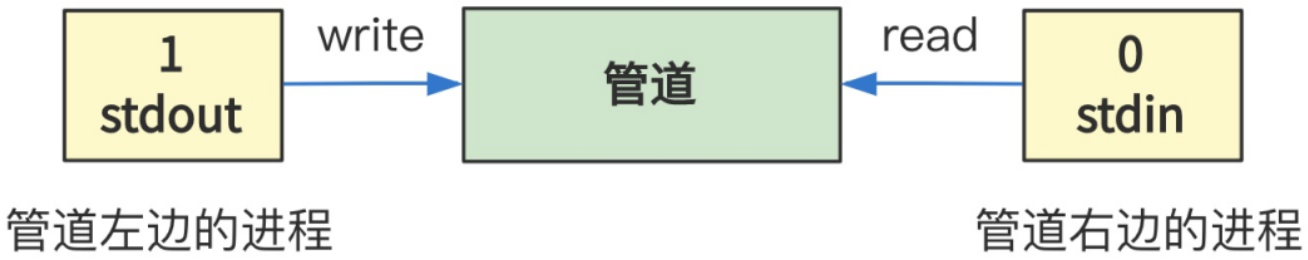
创建管道的方法 get_pipe_inode 方法如下。

```
// fs.h
#define PIPE_HEAD(inode) ((inode).i_zone[0])
#define PIPE_TAIL(inode) ((inode).i_zone[1])

// inode.c
struct m_inode * get_pipe_inode(void) {
    struct m_inode *inode = get_empty_inode();
    inode->i_size=get_free_page();
    inode->i_count = 2; /* sum of readers/writers */
    PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0;
    inode->i_pipe = 1;
    return inode;
}
```

可以看出，正常文件的 inode 中的 i_size 表示文件大小，而管道类型文件的 i_size 表示供管道使用的这一页内存的起始地址。

OK，管道的原理在这里就说完了，最终我们就是实现了一个进程的输出指向了另一个进程的输入。



回到最开始的 runcmd 方法。

```
// xv6-public sh.c
void runcmd(struct cmd *cmd) {
    ...
    switch(cmd->type) {
        ...
        case EXEC:
        ecmd = (struct execcmd*)cmd;
        ...
        exec(ecmd->argv[0], ecmd->argv);
        ...
        break;

        case REDIR: ...
        case LIST: ...
        case PIPE: ...
        case BACK: ...
    }
}
```

如果展开每个 switch 分支你会发现，不论是更换当前目录的 **REDIR** 也就是 cd 命令，还是用分号分隔开的 **LIST** 命令，还是我们上面讲到的 **PIPE** 命令，最终都会被拆解成一个个可以被解析为 **EXEC** 类型的命令。

EXEC 类型会执行到 **exec** 这个方法，在 Linux 0.11 中，最终会通过系统调用执行到 **sys_execve** 方法。

这个方法就是最终加载并执行具体程序的过程，在 第35回 | execve 加载并执行 shell 程序 和 第36回 | 缺页中断，我们已经讲过如何通过 execve 加载并执行 shell 程序了，并且在加载 shell 程序时，并不会立即将磁盘中的数据加载到内存，而是会在真正执行 shell 程序时，引发缺页中断，从而按需将磁盘中的数据加载到内存。

这个流程在本回我们就不再赘述了，不过当初在讲这块流程以及其它需要将数据从硬盘加载到内存的逻辑时，总是跳过这一步的细节。

那么我们下一回，就彻底把这个硬盘到内存的流程拆开了讲解！

欲知后事如何，且听下回分解。