```
71        buffer++;
72     }
73     return bytes_write;
74 }
```

先看下代码第二个函数 sys_pipe，它接受 1 个参数，存储管道文件描述符的数组 pipefd，功能是创建管道，成功后描述符 pipefd[0] 可用于读取管道，pipefd[1] 可用于写入管道，然后返回值为 0，否则返回-1。

函数先调用 get_free_slot_in_global 从 file_table 中获得可用的文件结构空位下标，记为 global_fd，然后第 19 行为该文件结构中的 fd_inode 分配一页内核内存做管道的环形缓冲区。接着第 22 行调用 ioqueue_init 初始化环形缓冲区。

第 28 行将该文件结构的 fd_flag 置为宏 PIPE_FLAG，宏 PIPE_FLAG 定义在 pipe.h 中，代码是“#define PIPE_FLAG 0xFFFF”，正如我们在设计阶段所说的，复用了文件结构中的 fd_flag 成员，把该值置为 0xFFFF 来表示此文件结构对应的是管道。

接着第 31 行把 fd_pos 置为 2，表示有两个文件描述符对应这个管道，这两个文件描述符是第 32～33 行通过 pcb_fd_install 来安装的，返回的描述符分别存储到 pipefd[0]和 pipefd[1]中，我们分别用它们来读取和写入管道。最后通过 return 返回 0，管道创建成功。

现在返回去说第一个函数 is_pipe，它接受 1 个参数，文件描述符 local_fd，也就是 pcb 中数组 fd_table 的下标，功能是判断文件描述符 local_fd 是否是管道。判断的原理是先找出 local_fd 对应的 file_table 中的下标 global_fd，然后判断文件表 file_talbe[global_fd]的 fd_flag 的值是否为 PIPE_FLAG。

函数 pipe_read 接受 3 个参数，文件描述符 fd、存储数据的缓冲区 buf、读取数据的数量 count，功能是从文件描述符 fd 中读取 count 字节到 buf。

第 41 行获得了 fd 对应的 file_table 中的下标 global_fd，第 44 行获得了相应文件结构中的环形缓冲区。

第 47～48 行根据缓冲区中数据量 ioq_len 和待读取的数据量 count 的大小，选择两者中较小的值作为读取的实际数据量 size，size 就是咱们在上节中所说的“适量”。第 49～53 行通过 while 循环调用 ioq_getchar 逐字节完成读取。

函数 pipe_write 功能是把缓冲区 buf 中的 count 个字节写入管道对应的文件描述符 fd。其实现同 pipe_read 雷同，不说了。

管道的操作也是通过文件系统，因此要修改相关文件系统的代码，如代码 15-41 所示。

代码 15-41　（project/c15/j/fs/fs.c）

```
...略
375 /* 关闭文件描述符 fd 指向的文件，成功返回 0，否则返回-1 */
376 int32_t sys_close(int32_t fd) {
377     int32_t ret = -1;   // 返回值默认为-1，即失败
378     if (fd > 2) {
379          uint32_t global_fd = fd_local2global(fd);
380          if (is_pipe(fd)) {
381               /* 如果此管道上的描述符都被关闭，释放管道的环形缓冲区 */
382               if (--file_table[global_fd].fd_pos == 0) {
383                    mfree_page(PF_KERNEL, file_table[global_fd].fd_inode, 1);
384                    file_table[global_fd].fd_inode = NULL;
385               }
386               ret = 0;
387          } else {
388               ret = file_close(&file_table[global_fd]);
389          }
390          running_thread()->fd_table[fd] = -1; // 使该文件描述符位可用
391     }
392     return ret;
393 }
394
395 /* 将 buf 中连续 count 个字节写入文件描述符 fd,
    成功则返回写入的字节数，失败返回-1 */
396 int32_t sys_write(int32_t fd, const void* buf, uint32_t count) {
397     if (fd < 0) {
398          printk("sys_write: fd error\n");
399          return -1;
```

```
400     }
401     if (fd == stdout_no) {
402          /* 标准输出有可能被重定向为管道缓冲区，因此要判断 */
403          if (is_pipe(fd)) {
404               return pipe_write(fd, buf, count);
405          } else {
406               char tmp_buf[1024] = {0};
407               memcpy(tmp_buf, buf, count);
408               console_put_str(tmp_buf);
409               return count;
410          }
411      } else if (is_pipe(fd)){        /* 若是管道就调用管道的方法 */
412          return pipe_write(fd, buf, count);
413      } else {
414          uint32_t _fd = fd_local2global(fd);
415          struct file* wr_file = &file_table[_fd];
416          if (wr_file->fd_flag & O_WRONLY || wr_file->fd_flag & O_RDWR) {
417               uint32_t bytes_written  = file_write(wr_file, buf, count);
418               return bytes_written;
419          } else {
420               console_put_str("sys_write: not allowed to write file
                  without flag O_RDWR or O_WRONLY\n");
421               return -1;
422          }
423     }
424 }
425
426     /* 从文件描述符 fd 指向的文件中读取 count 个字节到 buf,
        若成功则返回读出的字节数，到文件尾则返回-1 */
427     int32_t sys_read(int32_t fd, void* buf, uint32_t count) {
428      ASSERT(buf != NULL);
429      int32_t ret = -1;
430      uint32_t global_fd = 0;
431      if (fd < 0 || fd == stdout_no || fd == stderr_no) {
432          printk("sys_read: fd error\n");
433      } else if (fd == stdin_no) {
434          /* 标准输入有可能被重定向为管道缓冲区，因此要判断 */
435          if (is_pipe(fd)) {
436               ret = pipe_read(fd, buf, count);
437          } else {
438              char* buffer = buf;
439              uint32_t bytes_read = 0;
440              while (bytes_read < count) {
441                 *buffer = ioq_getchar(&kbd_buf);
442                 bytes_read++;
443                 buffer++;
444              }
445              ret = (bytes_read == 0 ? -1 : (int32_t)bytes_read);
446          }
447      } else if (is_pipe(fd)) {     /* 若是管道就调用管道的方法 */
448          ret = pipe_read(fd, buf, count);
449      } else {
450          global_fd = fd_local2global(fd);
451          ret = file_read(&file_table[global_fd], buf, count);
452      }
453      return ret;
454 }
…略
```

关闭文件时，描述符 fd 对应的可能是管道，因此在函数 sys_close 中，我们在第 380～386 行加入了管道的处理，第 380 行通过函数 is_pipe(fd)判断关闭的文件描述符是否是管道，如果是就在第 382 行将相应文件结构的 fd_pos 减 1，如果减 1 后的值为 0，这说明没有文件描述符打开它了，所以在第 383 行调用 mfree_page 将管道环形缓冲区占用的 1 页内核页框释放。随后在第 384 行将相应文件结构中的 fd_inode 置为 NULL。

写入文件时，有可能写入的是管道，因此函数 sys_write 也做出了改动，在第 401 行处理标准输出的代码块中，第 403 行判断，如果标准输出是管道，这说明标准输出被重定向了（以后我们实现 shell 中管道操作就会涉及到重定向），就调用 pipe_write 方法写管道。第 411 行，如果 fd 不是标准描述符（标准输入、标准输出等），依然要通过 is_pipe 判断其是否是管道，如果是，就调用 pipe_write 方法写管道。

读入文件时，有可能读入的是管道，因此函数 sys_read 加入了对管道的处理。标准输入有可能被重定向，因此第 435 行调用 is_pipe 对此情况判断，如果确实是重定向了，就调用 pipe_read 读取管道。第 447～452 行是处理非标准描述符的代码，第 447 行判断如果是管道，就在第 448 行调用 pipe_read 完成。

管道是由父子进程共享的，因此在 fork 时也要增加管道的打开数，见代码 15-42。

代码 15-42　(project/c15/j/ userprog/fork.c)

```
...略
114 /* 更新 inode 打开数 */
115 static void update_inode_open_cnts(struct task_struct* thread) {
116      int32_t local_fd = 3, global_fd = 0;
117      while (local_fd < MAX_FILES_OPEN_PER_PROC) {
118         global_fd = thread->fd_table[local_fd];
119         ASSERT(global_fd < MAX_FILE_OPEN);
120         if (global_fd != -1) {
121              if (is_pipe(local_fd)) {
122                   file_table[global_fd].fd_pos++;
123              } else {
124                   file_table[global_fd].fd_inode->i_open_cnts++;
125              }
126         }
127         local_fd++;
128      }
129 }
...略
```

在函数 update_inode_open_cnts 的第 121 行，调用 is_pipe 判断是否为管道，如果是，就在第 122 行将对应文件结构的 fd_pos 加 1。

由于有了管道，程序退出时也要考虑相应的处理，见代码 15-43。

代码 15-43　(project/c15/j/ userprog/wait_exit.c)

```
...略
17 static void release_prog_resource(struct task_struct* release_thread) {
...略
59       /* 关闭进程打开的文件 */
60       uint8_t local_fd = 3;
61       while(local_fd < MAX_FILES_OPEN_PER_PROC) {
62          if (release_thread->fd_table[local_fd] != -1) {
63               if (is_pipe(local_fd)) {
64                    uint32_t global_fd = fd_local2global(local_fd);
65                    if (--file_table[global_fd].fd_pos == 0) {
66                         mfree_page(PF_KERNEL, file_table[global_fd].fd_inode, 1);
67                         file_table[global_fd].fd_inode = NULL;
68                    }
69               } else {
70                    sys_close(local_fd);
71               }
72          }
73          local_fd++;
74      }
75 }
```

如果程序退出时忘记关闭打开的文件或管道，在函数 release_prog_resource 中要关闭它们。第 63 行判断关闭的若是管道，就在第 65 行将对应文件结构的 fd_pod 减 1，如果减 1 后的值为 0，这说明没有进程再打开此管道了，此管道没用了，在第 66 行调用 mfree_page 回收管道环形缓冲区占用的一页内核页框。

另外，pipe 的系统调用我就悄悄添加了。

好啦，涉及的相关代码就改完了，本节到这结束。

15.7.4　利用管道实现进程间通信

本节咱们编写用户进程，在用户程序中创建管道来验证父子进程间的通信功能，见代码 15-44。

代码 15-44　(project/c15/j/command/prog_pipe.c)

```
1 #include "stdio.h"
2 #include "syscall.h"
3 #include "string.h"
4 int main(int argc, char** argv) {
```

```
    int32_t fd[2] = {-1};
    pipe(fd);
    int32_t pid = fork();
    if(pid) {      // 父进程
        close(fd[0]);  // 关闭输入
        write(fd[1], "Hi, my son, I love you!", 24);
        printf("\nI`m father, my pid is %d\n", getpid());
        return 8;
    } else {
        close(fd[1]);  // 关闭输出
        char buf[32] = {0};
        read(fd[0], buf, 24);
        printf("\nI`m child, my pid is %d\n", getpid());
        printf("I`m child, my father said to me: \"%s\"\n", buf);
        return 9;
    }
}
```

prog_pipe.c 是咱们的测试用例，主要用来演示管道的功能，另外说明一下，主函数中的参数 argc 和 argv 并未用上。

函数开头先定义了数组 fd[2]，它用来存储管道返回的两个文件描述符。接着调用“pipe(fd)”创建管道，此时数组 fd 中已经是管道的两个描述符，我们用 fd[0]读管道，fd[1]写管道。接着调用 fork 派生子进程。父进程负责写管道，子进程读管道，因此在父进程代码中，第 9 行通过 close 关闭 fd[0]，然后调用 write 系统调用写入字符串“Hi, my son, I love you!”，父爱如山，满满正能量。然后调用 printf 输出“\nI`m father, my pid is…”，最后返回 8，父进程结束。

子进程通过 close 关闭 fd[1]，接着定义 32 字节的缓冲区 buf，然后调用 read 从 fd[0]中读取管道数据。然后第 17 行输出“\nI`m child, my pid is…”，接着第 18 行输出父进程对自己说的话，最后返回 9 子进程结束。

用户进程很简单，介绍完了，编译脚本 compile.c 同之前类似，不单独贴出了，编译后生成二进制文件是 prog_pipe。下面是把 prog_pipe 写入根目录的代码，用过后要注释掉，见代码 15-45。

代码 15-45 （project/c15/j/kernel/main.c）

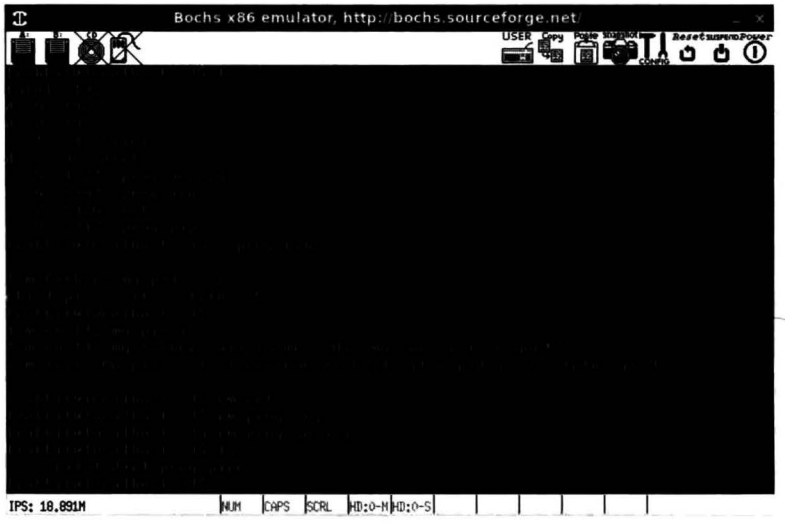
```
…略
int main(void) {
    put_str("I am kernel\n");
    init_all();

/*************    写入应用程序    *************/
    uint32_t file_size = 5343;
    uint32_t sec_cnt = DIV_ROUND_UP(file_size, 512);
    struct disk* sda = &channels[0].devices[0];
    void* prog_buf = sys_malloc(file_size);
    ide_read(sda, 300, prog_buf, sec_cnt);
    int32_t fd = sys_open("/prog_pipe", O_CREAT|O_RDWR);
    if (fd != -1) {
        if(sys_write(fd, prog_buf, file_size) == -1) {
            printk("file write error!\n");
            while(1);
        }
    }
/*************    写入应用程序结束    *************/
    cls_screen();
    console_put_str("[rabbit@localhost /]$ ");
    thread_exit(running_thread(), true);
    return 0;
}
…略
```

下面是运行的结果，如图 15-22 所示。

如图 15-22 所示，先执行 ls –l 查看文件写入的结果，prog_pipe 已经写入成功了，执行该命令后，prog_pipe 父进程先执行，第一行输出了“I`m father, my pid is 2”，然后此时父进程就退出了，需要其父进程 my_shell 为其善后。接着第二行输出的“child_pid 2，it`s status:8”是由 my_shell 获取其子进程 prog_pipe 后输出的，该子进程就是 prog_pipe 父进程。第三行输出的命令提示符“[rabbit@localhost /]”是 my_shell 捕获 prog_pipe 父进

程后的下一轮循环输出的。第四行是 prog_pipe 子进程运行，输出“I'm child，my pid is 5”，接着第五行输出父进程对自己所表达的内容。prog_pipe 父进程提前退出了，它在退出时，已经将其子进程过继给 init，因此 prog_pipe 子进程执行完后，为其做善后工作的是 init，此时 init 输出“I'm init，My pid is 1，I receive…”。



▲图 15-22　父子进程间通过管道通信

最后执行了三个 rm 命令，把 cat、prog_arg 和 prog_no_arg 这三个测试程序都删除了。

目测运行结果是正确的，因此本节到这就结束了，咱们还有最后一节。

15.7.5　在 shell 中支持管道

今天我们让 shell 支持管道操作。

管道操作大伙儿都了解吧，很多命令行界面都支持此类操作，比如 Windows 命令行窗口和 Linux 的 shell，管道符是‘|’，在命令行中可以有多个管道符，在管道符的左右两端各有一条命令，因此命令行中若包含管道符，至少要有两条命令。在命令行中支持管道通常是为了数据的二次加工、过滤出感兴趣的部分，比如“ps -ef|grep php-cgi”，这样会把 php-cgi 的信息从进程列表中过滤出来，但这样输出的信息中又包括 grep 命令本身，因此一般用双层管道：“ps -ef|grep php-cgi|grep –v grep”，其中“grep –v grep”是过滤出不包含 grep 的文本行，这样输出的信息就全是 php-cgi 的信息。

管道之所以可以这样用，原因是利用了输入输出重定向。通常情况下键盘是程序的输入，屏幕是程序的输出，它们都是标准的输入输出，即之前所说的 stdin 和 stdout。既然有“标准的”输入输出，就一定存在非标准的情况，这就是输入输出重定向。如果命令的输入并不来自于键盘，而是来自于文件，这就称为输入重定向，如果命令的输出并不是屏幕，而是想写入到文件，这就称为输出重定向。利用输入输出重定向的原理，可以将一个命令的输出作为另一个命令的输入。因此命令行中若包括管道符，则将管道符左边命令的输出作为管道符右边命令的输入。

管道操作的原理就是这样，以上所说的似乎和平时了解的差不多，如果觉得依然只是在表面上陈述，并没有说到骨子里，除了我个人表述的原因外，估计就是缺乏实践经验造成的，任何知识在缺乏实际操作经验的情况下都显得“虚无缥缈、飘忽不定”，因此只能在实际代码中理解了。

管道的核心就是输入输出重定向，称为核心其实实现起来并不难，再加上咱们本身的定位就是入门……不啰嗦了，见代码 15-46。

代码 15-46　(project/c15/k/shell/pipe.c)

```
…略
37 /* 将文件描述符 old_local_fd 重定向为 new_local_fd */
38 void sys_fd_redirect(uint32_t old_local_fd, uint32_t new_local_fd) {
39     struct task_struct* cur = running_thread();
```

```
40      /* 针对恢复标准描述符 */
41      if (new_local_fd < 3) {
42          cur->fd_table[old_local_fd] = new_local_fd;
43      } else {
44          uint32_t new_global_fd = cur->fd_table[new_local_fd];
45          cur->fd_table[old_local_fd] = new_global_fd;
46      }
47  }
…略
```

函数 sys_fd_redirect 接受 2 个参数，旧文件描述符 old_local_fd、新文件描述符 new_local_fd，功能是将文件描述符 old_local_fd 重定向为 new_local_fd。

函数原理很简单，我们知道文件描述符是 pcb 中数组 fd_table 的下标，数组元素的值是全局文件表 file_table 的下标，因此很容易想到，文件描述符重定向的原理就是：将数组 fd_table 中下标为 old_local_fd 的元素的值用下标为 new_local_fd 的元素的值替换。

另外，pcb 中文件描述符表 fd_table 和全局文件表 file_table 中的前 3 个元素都是预留的，它们分别作为标准输入、标准输出和标准错误（未实现，但依然预留），因此，如果 new_local_fd 小于 3 的话，不需要从 fd_table 中获取元素值，可以直接把 new_local_fd 赋值给 fd_table[old_local_fd]，而这通常用于将输入输出恢复为标准的输入输出，下面看实现。

第 39 行获取了当前线程 cur，第 41～42 行对标准输入输出做了特殊处理，如果 new_local_fd 小于 3，直接将 new_local_fd 给 cur->fd_table[old_local_fd] 赋值，否则，第 44～45 行，先获得 new_local_fd 对应的 file_table 下标 new_global_fd，然后将 new_global_fd 赋值给 cur->fd_table[old_local_fd]，至此完成了重定向。

下面还要在 shell.c 中增加代码，见代码 15-47。

代码 15-47 （project/c15/k/shell/shell.c）

```
…略
117 /* 执行命令 */
118 static void cmd_execute(uint32_t argc, char** argv) {
119     if (!strcmp("ls", argv[0])) {
120          buildin_ls(argc, argv);
121     } else if (!strcmp("cd", argv[0])) {
122         if (buildin_cd(argc, argv) != NULL) {
123             memset(cwd_cache, 0, MAX_PATH_LEN);
124             strcpy(cwd_cache, final_path);
125         }
126     } else if (!strcmp("pwd", argv[0])) {
…略
166 char* argv[MAX_ARG_NR] = {NULL};
167 int32_t argc = -1;
168 /* 简单的 shell */
169 void my_shell(void) {
170     cwd_cache[0] = '/';
171     while (1) {
172        print_prompt();
173        memset(final_path, 0, MAX_PATH_LEN);
174        memset(cmd_line, 0, MAX_PATH_LEN);
175        readline(cmd_line, MAX_PATH_LEN);
176        if (cmd_line[0] == 0) {    // 若只键入了一个回车
177            continue;
178        }
179
180        /* 针对管道的处理 */
181        char* pipe_symbol = strchr(cmd_line, '|');
182        if (pipe_symbol) {
183     /* 支持多重管道操作，如 cmd1|cmd2|..|cmdn,
184      * cmd1 的标准输出和 cmdn 的标准输入需要单独处理 */
185
186     /*1 生成管道*/
187          int32_t fd[2] = {-1};      // fd[0]用于输入，fd[1]用于输出
188           pipe(fd);
189          /* 将标准输出重定向到 fd[1],
            使后面的输出信息重定向到内核环形缓冲区 */
190          fd_redirect(1,fd[1]);
191
```

```
192     /*2 第一个命令 */
193          char* each_cmd = cmd_line;
194          pipe_symbol = strchr(each_cmd, '|');
195          *pipe_symbol = 0;
196
197          /* 执行第一个命令，命令的输出会写入环形缓冲区 */
198          argc = -1;
199          argc = cmd_parse(each_cmd, argv, ' ');
200          cmd_execute(argc, argv);
201
202          /* 跨过'|', 处理下一个命令 */
203          each_cmd = pipe_symbol + 1;
204
205          /* 将标准输入重定向到 fd[0]，使之指向内核环形缓冲区*/
206          fd_redirect(0, fd[0]);
207     /*3 中间的命令,命令的输入和输出都是指向环形缓冲区 */
208          while ((pipe_symbol = strchr(each_cmd, '|'))) {
209             *pipe_symbol = 0;
210             argc = -1;
211             argc = cmd_parse(each_cmd, argv, ' ');
212             cmd_execute(argc, argv);
213             each_cmd = pipe_symbol + 1;
214           }
215
216     /*4 处理管道中最后一个命令 */
217           /* 将标准输出恢复屏幕 */
218           fd_redirect(1,1);
219
220          /* 执行最后一个命令 */
221           argc = -1;
222           argc = cmd_parse(each_cmd, argv, ' ');
223           cmd_execute(argc, argv);
224
225     /*5  将标准输入恢复为键盘 */
226           fd_redirect(0,0);
227
228     /*6 关闭管道 */
229           close(fd[0]);
230           close(fd[1]);
231       } else {          // 一般无管道操作的命令
232           argc = -1;
233           argc = cmd_parse(cmd_line, argv, ' ');
234           if (argc == -1) {
235                printf("num of arguments exceed %d\n", MAX_ARG_NR);
236                continue;
237           }
238           cmd_execute(argc, argv);
239        }
240     }
241     panic("my_shell: should not be here");
242 }
```

本节中把 shell.c 中原本判断内建、外部命令的一堆 if else 封装到第 118 行的函数 cmd_execute 中，不多说了，本次对管道的处理是函数 my_shell 中第 181～230 行。

第 181 行通过 strchr 函数在 cmd_line 中寻找管道字符'|'，如果找到，pipe_symbol 的值则为字符'|'的地址，下面讨论下处理管道命令的思路。

在命令行中可以出现多个管道符接连过滤数据的情况，比如“cmd1|cmd2|..|cmdn”，这其中包括了 n 个命令的接力配合，我们称之为多重管道操作。我们讨论过了，管道操作中前一个命令的输出作为后一个命令的输入，cmd1 是第 1 个命令，没人为它提供输入，因此其输入不变，仍为标准输入，但其输出是要传给命令 cmd2，因此 cmd1 的标准输出不能指向屏幕了，必须要重定向到管道的环形缓冲区中，命令 cmd2 的标准输入必须也重定向到管道的环形缓冲区才能够获得 cmd1 的输出，cmd2 的输出为了传给 cmd3，必须也要将标准输出重定向到管道环形缓冲区，cmd4 为了获得 cmd3 的输出结果，必须将 cmd4 的标准输入重定向到管道环形缓冲区……依次类推，当执行到命令 cmdn 时，cmdn 的标准输入必须要指向管道环形缓冲区才能获得命令 cmdn-1 提供的输出，但 cmdn 是最后一个命令，它要将结果打印到屏幕，因此其标

准输出不用改变，依然为屏幕。也就是说，除 cmd1 的标准输入和 cmdn 的标准输出不变外，其他命令的标准输入和输出都要重定向到管道。下面分六步来完成管道操作。

第 186～191 行完成第一步，生成管道，这是调用 pipe 系统调用完成的。第 190 行调用 fd_redirect(1,fd[1]) 将标准输出重定向到用于写管道的文件描述符 fd[1]，至此程序的输出都写到管道中。

第 193～206 行开始第二步，解析第 1 个命令并执行。命令行中的各个命令是用指针 each_cmd 记录的，它指向各命令在 cmd_line 中的地址。解析出命令后调用 cmd_execute 执行，然后在第 203 行使 pipe_symbol 加 1，跨过 cmd_line 中的相应的'|'。在执行第 2 个命令之前，在第 206 行执行“fd_redirect(0,fd[0])”将标准输入重定向到管道，这样第 2 个命令才能获得第 1 个命令的输出。

第 208～214 行完成第三步，循环处理 cmd2～cmdn-1，此时它们的标准输入和输出都已指向管道，继续解析命令并执行就可以了，不多说了。

执行完 while 循环后，第 218～223 行完成第四步，调用“fd_redirect(1,1)”将标准输出恢复为屏幕，然后第 223 行执行最后一个命令，此时命令的输出信息会在屏幕上显示。

第 226 行是第五步，调用“fd_redirect(0,0)”将标准输入恢复为键盘。

第 229～230 行是第六步，将管道关闭。至此管道的处理就完成了。第 231～239 行是一般无管道符的处理。

按理说该是测试的时候了，可我们还没有从标准输入获取数据的用户程序呢，立即把之前的 cat.c 改改，使 cat 无参数时，默认从键盘获取数据，见代码 15-48。

代码 15-48 （project/c15/k/command/cat.c）

```
1  #include "syscall.h"
2  #include "stdio.h"
3  #include "string.h"
4  int main(int argc, char** argv) {
5      if (argc > 2) {
6          printf("cat: argument error\n");
7          exit(-2);
8      }
9
10     if (argc == 1) {
11         char buf[512] = {0};
12         read(0, buf, 512);
13         printf("%s",buf);
14         exit(0);
15     }
16
17     int buf_size = 1024;
18     char abs_path[512] = {0};
19     void* buf = malloc(buf_size);
20     if (buf == NULL) {
21         printf("cat: malloc memory failed\n");
22         return -1;
23     }
24     if (argv[1][0] != '/') {
25         getcwd(abs_path, 512);
26         strcat(abs_path, "/");
27         strcat(abs_path, argv[1]);
28     } else {
29         strcpy(abs_path, argv[1]);
30     }
31     int fd = open(abs_path, O_RDONLY);
32     if (fd == -1) {
33          printf("cat: open: open %s failed\n", argv[1]);
34          return -1;
35     }
36     int read_bytes= 0;
37     while (1) {
38        read_bytes = read(fd, buf, buf_size);
39        if (read_bytes == -1) {
40            break;
41        }
42        write(1, buf, read_bytes);
```

```
43      }
44      free(buf);
45      close(fd);
46      return 66;
47  }
```

这个版本的 cat.c 就是在上一版的基础上，加了第 10～15 行，当无参数时，直接调用 read 系统调用从键盘获取数据。编译还是用 compile.sh 就行了，同之前类似，不贴代码了。

下面是在 main.c 中将 cat 写入。另外说一下，在上节的图 15-22 中，我们已经将根目录下曾经的测试用例删除了，目录根目录中只有普通文件 file1，目录 dir1，程序 prog_pipe 以及“.”和“..”。所以下面代码中可以在根目录中写入新的 cat 程序，见代码 15-49。

代码 15-49　（project/c15/k/kernel/main.c）

```
…略
21 int main(void) {
22     put_str("I am kernel\n");
23     init_all();
24
25 /*************    写入应用程序    *************/
26     uint32_t file_size = 5698;
27     uint32_t sec_cnt = DIV_ROUND_UP(file_size, 512);
28     struct disk* sda = &channels[0].devices[0];
29     void* prog_buf = sys_malloc(file_size);
30     ide_read(sda, 300, prog_buf, sec_cnt);
31     int32_t fd = sys_open("/cat", O_CREAT|O_RDWR);
32     if (fd != -1) {
33          if(sys_write(fd, prog_buf, file_size) == -1) {
34              printk("file write error!\n");
35              while(1);
36          }
37     }
38 /*************    写入应用程序结束    *************/
39     cls_screen();
40     console_put_str("[rabbit@localhost /]$ ");
41     thread_exit(running_thread(), true);
42     return 0;
43 }
…略
```

另外，为了显示系统支持的命令，我加了个内建命令 help，当在 shell 中输入 help 时，系统会打印支持的内建命令及快捷键。原理是实现了 help 系统调用，下面是 help 对应的 sys_help 代码，它定义到了 fs.c 中，见代码 15-50。

代码 15-50　（project/c15/k/fs/fs.c）

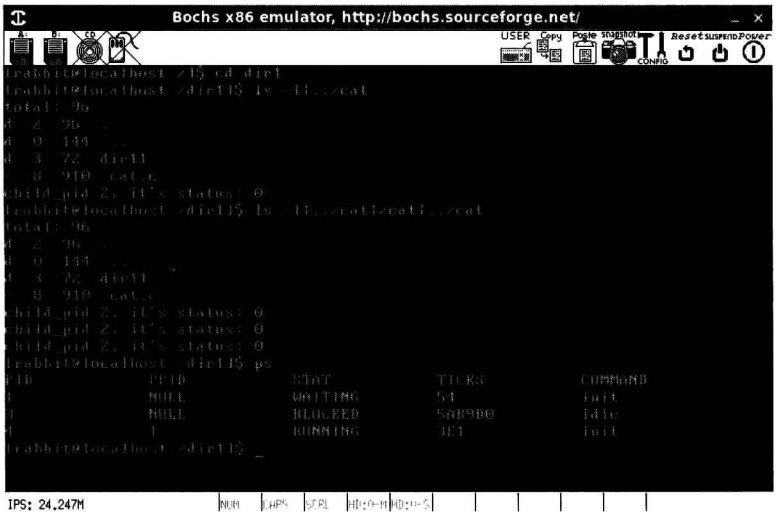
```
…略
891 /* 显示系统支持的内部命令 */
892 void sys_help(void) {
893     printk("\
894  buildin commands:\n\
895         ls: show directory or file information\n\
896         cd: change current work directory\n\
897         mkdir: create a directory\n\
898         rmdir: remove a empty directory\n\
899         rm: remove a regular file\n\
900         pwd: show current work directory\n\
901         ps: show process information\n\
902         clear: clear screen\n\
903  shortcut key:\n\
904         ctrl+l: clear screen\n\
905         ctrl+u: clear input\n\n");
906 }
…略
```

添加系统调用的过程就不多说了，图 15-23 所示是目前所有的系统调用号。

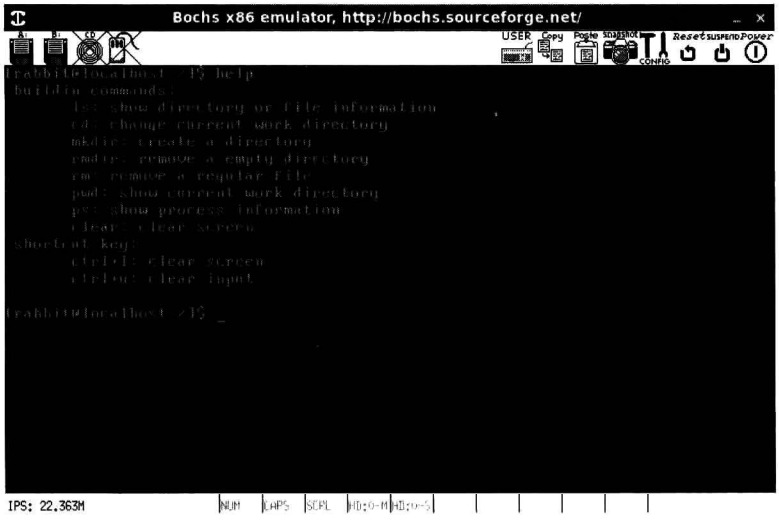
编译运行之后，下面的两张图是运行结果，如图 15-24 和图 15-25 所示。

```
    SYSCALL_NR {
    SYS_GETPID,
    SYS_WRITE,
    SYS_MALLOC,
    SYS_FREE,
    SYS_FORK,
    SYS_READ,
    SYS_PUTCHAR,
    SYS_CLEAR,
    SYS_GETCWD,
    SYS_OPEN,
    SYS_CLOSE,
    SYS_LSEEK,
    SYS_UNLINK,
    SYS_MKDIR,
    SYS_OPENDIR,
    SYS_CLOSEDIR,
    SYS_CHDIR,
    SYS_RMDIR,
    SYS_READDIR,
    SYS_REWINDDIR,
    SYS_STAT,
    SYS_PS,
    SYS_EXECV,
    SYS_EXIT,
    SYS_WAIT,
    SYS_PIPE,
    SYS_FD_REDIRECT,
    SYS_HELP
};
```

▲图 15-23 系统调用号



▲图 15-24 管道处理



▲图 15-25 系统帮助

目测符合预期，因此本节就到这了。

参考文献

1.《x86/x64 体系探索及编程》，作者邓志。

2.《深入理解 Linux 内核（第 3 版）（涵盖 2.6 版）》，作者博韦等。

3.《Linux 内核设计与实现（原书第 3 版）》，作者拉芙（Robert Love），译者陈莉君、康华。

4.《深入 Linux 内核架构》，作者莫尔勒（Wolfgang Mauerer），译者郭旭。

5.《Linux 内核完全剖析》，作者赵炯。

6.《Linux 内核源代码情景分析（上、下册）》，作者毛德操、胡希明。

7.《一个操作系统的实现》，作者于渊。

8.《自己动手写嵌入式操作系统》，作者蓝枫叶。

9.《操作系统设计与实现（第三版）（上、下册）》，作者安德鲁（Andrew S.Tanenbaum）、塔嫩鲍姆（Albert S.Woodhull）。

10.《现代操作系统（原书第 3 版）》，作者塔嫩鲍姆（Tanenbaum.A.S），译者陈向群、马洪兵。

11.《计算机的心智操作系统之哲学原理（第 2 版）》，作者邹恒明。

12.《C 语言入门经典（第 5 版）》，作者霍尔顿（Ivor Horton）。

13.《C 语言程序设计：现代方法（第 2 版）》，作者金（K.N.King）。

14.《Linux 程序设计（第 4 版）》，作者马修（Neil Matthew）、斯通斯（Richard Stones）。

15. 斯坦福大学教学操作系统 Pintos。

16.《x86 汇编语言：从实模式到保护模式》，作者李忠、王晓波。

17.《汇编语言（第 3 版）》，作者王爽。

18.《基于 Linux 系统的汇编语言程序设计》，作者程楠。

19.《Linux 内核完全注释》，作者赵炯。

20.《微机原理与接口技术教程》，作者王克义、鲁守智。

21.《微机原理与接口技术（第 2 版）》，作者周明德、蒋本珊。

22.《INTEL 开发手册三卷》。

23.《跟我一起写 makefile》。

24.《GNU gcc 嵌入式系统开发》，作者董文军。

以上列举不详尽，如有遗漏请见谅。