实　　验　　报　　告

学　　号	21030031009	姓　　名	惠欣宇	专业班级	计算机 4 班
课程名称	**操作系统**			学期	**2023 年秋季学期**
任课教师	窦金凤	完成日期	2024.1.5	上机课时间	周五 3-6
实 验 名 称	**管道与 Shell**				

Exercise 部分：

Exercise 6.1

> **Exercise 6.1** 仔细观察 pipe 中新出现的权限位PTE_LIBRARY，根据上述提示修改 fork 系统调用，使得**管道缓冲区是父子进程共享的**，不设置为写时复制的模式。 ■

这里在 lab4 已经做过修改，因此这里不需要修改。

Exercise 6.2

> **Exercise 6.2** 根据上述提示与代码中的注释，填写 user/pipe.c 中的 piperead、pipewrite、_pipeisclosed 函数并通过 testpipe 的测试。 ■

管道的读取操作，逐字节传输数据。循环中的逻辑如下：

检查管道是否被关闭（使用 _pipeisclosed 函数），如果是，则返回已经读取的字节数 i。

如果读取位置 p_rpos 等于写入位置 p_wpos，说明管道为空。如果此时已经复制了一些字节（i > 0），则直接返回已经读取的字节数，否则调用 syscall_yield 进行让步。

在 p_rpos 小于 p_wpos 的情况下，将管道缓冲区的字节逐一复制到目标缓冲区 rbuf 中，并递增 i 和 p_rpos。

循环继续执行，直到达到读取字节数 n 或者管道为空且关闭。

代码如图 1：

```
115 static int piperead(struct Fd *fd, void *vbuf, u_int n, u_int offset)
116 {
117     // Your code here.  See the lab text for a description of
118     // what piperead needs to do.  Write a loop that
119     // transfers one byte at a time.  If you decide you need
120     // to yield (because the pipe is empty), only yield if
121     // you have not yet copied any bytes.  (If you have copied
122     // some bytes, return what you have instead of yielding.)
123     // If the pipe is empty and closed and you didn't copy any data out, return 0.
124     // Use _pipeisclosed to check whether the pipe is closed.
125     int i =0;
126     struct Pipe *p;
127     char *rbuf = vbuf ;
128     p = (struct Pipe*)fd2data(fd);
129     while(1){
130         if(_pipeisclosed(fd,p)){
131             return i;
132         }
133         if(p->p_rpos == p->p_wpos){
134             if(i==0){
135                 syscall_yield();
136             } else {
137                 return i;
138             }
139         }
140         while (p->p_rpos < p->p_wpos){
141             if(i >= n){
142                 return i;
143             }
144             rbuf[i] = p->p_buf[p->p_rpos % BY2PIPE];
145             i++;
146             p->p_rpos++;
147         }
148     }
149     user_panic("piperead not implemented");
150 }
```

图 1　piperead 函数实现

管道的写入操作，逐字节传输数据。循环中的逻辑如下：

循环对输入缓冲区 vbuf 中的每个字节进行处理。

当写入位置 p_wpos 等于读取位置 p_rpos 加上 BY2PIPE 时，说明管道已满。在这种情况下，如果管道被关闭（使用 _pipeisclosed 函数检查），则返回 0；否则，通过 syscall_yield 进行让步，等待管道中有空闲位置。

在管道未满的情况下，将 vbuf 中的字节逐一复制到管道缓冲区 p_buf 中，并递增 p_wpos。循环继续执行，直到将输入缓冲区中的所有字节都写入管道。

代码如图 2：

```
152 static int pipewrite(struct Fd *fd, const void *vbuf, u_int n, u_int offset)
153 {
154     // Your code here.  See the lab text for a description of what
155     // pipewrite needs to do.  Write a loop that transfers one byte
156     // at a time.  Unlike in read, it is not okay to write only some
157     // of the data.  If the pipe fills and you've only copied some of
158     // the data, wait for the pipe to empty and then keep copying.
159     // If the pipe is full and closed, return 0.
160     // Use _pipeisclosed to check whether the pipe is closed.
161     int i;
162     struct Pipe *p;
163     char *wbuf = vbuf;
164     p = (struct Pipe *) fd2data(fd);
165     for(i = 0; i < n; i++){
166         while(p->p_wpos == p->p_rpos + BY2PIPE){
167             if(_pipeisclosed(fd, p)){
168                 writef("writing on closed pipe\n");
169                 return 0;
170             }
171             syscall_yield();
172         }
173         p->p_buf[p->p_wpos % BY2PIPE] = wbuf[i];
174         p->p_wpos++;
175 
176     }
177     return n;
178     user_panic("pipewrite not implemented");
179 }
```

图 2　pipewrite 函数实现

检查管道是否已关闭，具体逻辑如下：

使用 pageref 函数获取文件描述符 fd 和管道 p 的引用次数，并保存到变量 pfd 和 pfp 中。

通过循环等待 env->env_runs（这是一个用于标记进程运行状态的计数器）的变化，确保在获取引用次数期间没有其他进程对引用次数进行修改。

检查 pfd 和 pfp 是否相等，如果相等，则说明文件描述符 fd 和管道 p 的引用次数相同，即所有对应的文件描述符都关闭了。因此，返回 1 表示管道已关闭；如果不相等，则返回 0 表示管道未关闭。

如果循环检查期间 env->env_runs 发生了变化，可能有其他进程进行了引用次数的修改，因此需要重新执行循环检查。

代码如图 3：

```
70  static int _pipeisclosed(struct Fd *fd, struct Pipe *p)
71  {
72      // Your code here.
73      //
74      // Check pageref(fd) and pageref(p),
75      // returning 1 if they're the same, 0 otherwise.
76      //
77      // The logic here is that pageref(p) is the total
78      // number of readers *and* writers, whereas pageref(fd)
79      // is the number of file descriptors like fd (readers if fd is
80      // a reader, writers if fd is a writer).
81      //
82      // If the number of file descriptors like fd is equal
83      // to the total number of readers and writers, then
84      // everybody left is what fd is.  So the other end of
85      // the pipe is closed.
86      int pfd,pfp,runs;
87      do{
88          runs = env->env_runs;
89          pfd = pageref(fd);
90          pfp = pageref(p);
91      }while(runs != env->env_runs);
92      if(pfd == pfp){
93          return 1;
94      }else{
95          return 0;
96      }
97      user_panic("_pipeisclosed not implemented");
98  }
```

图 3　_pipeisclosed 函数实现

Exercise 6.3

Exercise 6.3 修改 user/pipe.c 中的 pipeclose 与 user/fd.c 中的 dup 函数以避免上述情景中的进程竞争情况。 ■

pipeclose 函数用于关闭管道文件描述符：

将传入的文件描述符 fd 映射的 struct Fd 结构体进行解引用，并将其地址保存在 tmp 中。

通过 syscall_mem_unmap 分别释放 fd 和 fd 对应的数据地址的映射关系。

返回 0 表示成功关闭。

代码如图 4 所示。

```
187 static int pipeclose(struct Fd *fd)
188 {
189     struct Fd *tmp = fd;
190     syscall_mem_unmap(0,fd);
191     syscall_mem_unmap(0, fd2data(tmp));
192     return 0;
193 }
```

图 4　pipeclose 函数实现

dup 主要功能是复制文件描述符:

通过 fd_lookup 函数获取旧文件描述符 oldfdnum 对应的 struct Fd 结构体，并检查是否获取成功。

调用 close 函数关闭新文件描述符 newfdnum，释放新文件描述符对应的资源。

通过 INDEX2FD 宏将新文件描述符编号 newfdnum 转换为对应的 struct Fd 结构体，并获取旧文件

描述符 oldfd 对应的数据地址 ova 和新文件描述符 newfd 对应的数据地址 nva。

遍历旧文件描述符对应的虚拟地址范围，通过 syscall_mem_map 将旧文件描述符的相应页面映射到新文件描述符对应的地址上。这样，新文件描述符就复制了旧文件描述符的映射关系。

最后，通过 syscall_mem_map 将旧文件描述符的 struct Fd 结构体映射到新文件描述符对应的地址上，完成文件描述符的复制。

如果过程中出现错误，通过 syscall_mem_unmap 释放已映射的资源，并返回错误码。

代码如图 5 所示。

```
131 int dup(int oldfdnum, int newfdnum)
132 {
133     int i, r;
134     u_int ova, nva, pte;
135     struct Fd *oldfd, *newfd;
136 
137     if ((r = fd_lookup(oldfdnum, &oldfd)) < 0) {
138         return r;
139     }
140     close(newfdnum);
141     newfd = (struct Fd *)INDEX2FD(newfdnum);
142     ova = fd2data(oldfd);
143     nva = fd2data(newfd);
144     if ((* vpd)[PDX(ova)]) {
145         for (i = 0; i < PDMAP; i += BY2PG) {
146             pte = (* vpt)[VPN(ova + i)];
147 
148             if (pte & PTE_V) {
149                 if ((r = syscall_mem_map(0, ova + i, 0, nva + i,
150                                          pte & (PTE_V | PTE_R | PTE_LIBRARY))) < 0) {
151                     goto err;
152                 }
153             }
154         }
155     }
156     if ((r = syscall_mem_map(0, (u_int)oldfd, 0, (u_int)newfd,
157                              ((*vpt)[VPN(oldfd)]) & (PTE_V | PTE_R | PTE_LIBRARY))) < 0) {
158         goto err;
159     }
160     return newfdnum;
161 err:
162     syscall_mem_unmap(0, (u_int)newfd);
163 
164     for (i = 0; i < PDMAP; i += BY2PG) {
165         syscall_mem_unmap(0, nva + i);
166     }
167     return r;
168 }
```

图 5　dup 函数实现

Exercise 6.4

> **Exercise 6.4** 根据上面的表述，修改_pipeisclosed函数，使得它满足”同步读”的要求。注意 env_runs 变量是需要维护的。 ■

在完成 exercise6.2 时已经将“同步读”考虑进去，因此这里不需要做出修改。

Exercise 6.5

> **Exercise 6.5** 根据以上描述，补充完成 user/sh.c 中的 `void runcmd(char *s)`。 ■

Runcmd 其实是实现了一个简单的 shell 解释器，它可以解析和执行用户输入的命令：

使用 gettoken 函数从字符串 s 中获取命令参数，并存储在 argv 数组中，同时记录参数个数 argc。

根据命令的不同部分执行相应的操作：

< 操作符：使用 open 函数打开文件，并通过 dup 函数将文件描述符复制到标准输入（文件描述符 0）。

> 操作符：使用 open 函数打开文件，并通过 dup 函数将文件描述符复制到标准输出（文件描述符 1）。

| 操作符：创建管道，并通过 fork 函数创建子进程。在子进程中通过 dup 将管道读端复制到标准输入，然后递归执行命令。在父进程中通过 dup 将管道写端复制到标准输出。

& 操作符：将 run_back 标志设为 1，表示后台运行。

; 操作符：通过 fork 创建子进程，子进程递归执行命令。

执行命令：

如果命令是 declare 或 unset，则调用 run_incmd 函数处理。

否则，通过 spawn 函数创建新进程执行命令。如果 spawn 返回值大于等于 0，表示成功创建进程，则等待该进程的执行完成。

在父进程中处理后续操作：

如果不是后台运行，等待子进程的执行完成（调用 wait 函数）。

如果是后台运行，创建一个新的子进程（back_id），在这个子进程中输出相关信息，并等待原子进程的执行完成。

补充完成的 runcmd 代码如下：

```
void runcmd(char *s, u_int env_id)
{
    char *argv[MAXARGS], *t;
    int argc, c, i, r, p[2], fd, rightpipe;
    int fdnum;
    int run_back = 0, back_id;
    int isright;
    rightpipe = 0;
    gettoken(s, 0);
again:
    argc = 0;
    for(;;){
        c = gettoken(0, &t);
        switch(c){
        case 0:
```

```
        goto runit;
    case 'w':
        if(argc == MAXARGS){
            writef("too many arguments\n");
            exit();
        }
        argv[argc++] = t;
        break;
    case '<':
        if(gettoken(0, &t) != 'w'){
            writef("syntax error: < not followed by word\n");
            exit();
        }
        // Your code here -- open t for reading,
        r=open(t, O_RDONLY);
        if(r < 0){
            user_panic("< open file failed");
        }
        fd = r;
        dup(fd, 0);
        close(fd);
        break;
    case '>':
        // Your code here -- open t for writing,
        if(gettoken(0, &t) != 'w'){
            writef("syntax error: < not followed by word\n");
            exit();
        }
        r = open(t, O_WRONLY);
```

```
        if(r < 0){
            writef("\033[0m\033[1;31m> open file failed!\033[0m");
            exit();
        }
        fd = r;
        dup(fd, 1);
        close(fd);
        break;
    case '|':
        pipe(p);
        if((rightpipe = fork()) == 0){
            dup(p[0], 0);
            close(p[0]);
            close(p[1]);
            goto again;
        }
        else{
            dup(p[1], 1);
            close(p[1]);
            close(p[0]);
            goto runit;
        }
        break;
    case '&':
        run_back = 1;
        break;
    case ';':
        if(isright = fork() == 0 ){
            goto again;
```

```
                }else{
                        goto runit;
                }
                break;
        }
    }

runit:
    if(argc == 0) {
        if (debug_) writef("EMPTY COMMAND\n");
        return;
    }
    argv[argc] = 0;
    if( strcmp(argv[0], "declare") == 0 ||
        strcmp(argv[0], "unset") == 0 ){
        r = -1;
        run_incmd(argc, argv, env_id);
    }
    else if ((r = spawn(argv[0], argv)) < 0){
    }
    if(strcmp(argv[0], "declare") != 0 && strcmp(argv[0], "unset") != 0 )
    close_all();
    if (r >= 0) {
        if(!run_back){
        if (debug_) writef("[%08x] WAIT %s %08x\n", env->env_id, argv[0], r);
            wait(r);
        }else{
            back_id = fork();
            if(back_id == 0){
```

```
                writef(LIGHT_GREEN(\n[%08x] running\t), r);
                for(i=0; i<argc; i++){
                    writef(LIGHT_GREEN(%s)" ", argv[i]);
                }
                wait(r);
                writef(LIGHT_GREEN(\n[%08x] done\t), r);
                for(i = 0; i<argc;i++){
                    writef("%s ", argv[i]);
                }
                writef("\n");
                exit();
            }
        }
    }
    if (rightpipe) {
        if (debug_) writef("[%08x] WAIT right-pipe %08x\n", env->env_id, rightpipe);
        wait(rightpipe);
    }
    if(isright){
        if(debug_) writef("[%08x] WAIT right arg %08x\n", env->env_id, isright);
        wait(isright);
    }

    if(strcmp(argv[0], "declare") != 0 && strcmp(argv[0], "unset") != 0 )
        exit();
}
```

Thinking 部分：

Thinking 6.1

Thinking 6.1 示例代码中，父进程操作管道的写端，子进程操作管道的读端。如果现在想让父进程作为“读者”，代码应当如何修改？ ■

只需要让父进程先关闭写通道。

Thinking 6.2

Thinking 6.2 上面这种不同步修改 pp_ref 而导致的进程竞争问题在 user/fd.c 中的 dup 函数中也存在。请结合代码模仿上述情景，分析一下我们的 dup 函数中为什么会出现预想之外的情况？ ■

dup 函数的作用是将一个文件描述符（例如 fd0）所对应的内容映射到另一个文件描述符（例如 fd1）中。在执行 dup 后，fd0 和 pipe 的引用次数都会增加 1，而 fd1 的引用次数会变为 fd0 的引用次数。然而，如果在复制文件描述符页面的过程中发生了时钟中断，可能导致 pipe 的引用次数没有来得及增加。这种情况下，如果另一进程调用了 pipeisclosed，它可能会发现 pageref(fd[0]) 等于 pageref(pipe)，从而错误地认为读/写端已经关闭。

Thinking 6.3

Thinking 6.3 阅读上述材料并思考：为什么系统调用一定是原子操作呢？如果你觉得不是所有的系统调用都是原子操作，请给出反例。希望能结合相关代码进行分析。 ■

在进行系统调用时，系统进入内核，此时会关闭时钟中断。

Thinking 6.4

Thinking 6.4 仔细阅读上面这段话，并思考下列问题

- 按照上述说法控制 **pipeclose** 中 fd 和 pipe unmap 的顺序，是否可以解决上述场景的进程竞争问题？给出你的分析过程。
- 我们只分析了 close 时的情形，那么对于 dup 中出现的情况又该如何解决？请模仿上述材料写写你的理解。

这个问题可以通过确保 pageref(pipe) 大于 pageref(fd) 来解决。在实际实现中，可以先对 pipe 进行映射，然后对 fd 进行映射，这样就可以确保 pipe 的引用次数先于 fd 的引用次数增加。此时，如果

读缓冲区为空且写缓冲区满，系统会继续循环，直到两者的引用次数都被减少到零，进而保证正确的处理。

Thinking 6.5

Thinking 6.5 bss 在 ELF 中并不占空间，但 ELF 加载进内存后，bss 段的数据占据了空间，并且初始值都是 0。请回答你设计的函数是如何实现上面这点的？ ■

当程序加载到 bin_size 到 sgsize 之间的数据时，说明已经进入了 BSS 段。在这个阶段，可以使用 bzero 函数将该段的数据赋值为 0，而无需再读取 ELF 文件的数据。

Thinking 6.6

Thinking 6.6 为什么我们的 *.b 的 text 段偏移值都是一样的，为固定值？

在 user/user.lds 文件中设定 text 段地址为固定值 0x00400000，因此偏移值都一样。

Thinking 6.7

Thinking 6.7 在哪步，0 和 1 被”安排”为标准输入和标准输出？请分析代码执行流程，给出答案。 ■

如图 6，init 中的 unmain 函数中将 0 映射在 1 上，相当于就是把控制台的输入输出缓冲区当做管道。

```
61      if (r != 0)
62          user_panic("first opencons used fd %d", r);
63      if ((r = dup(0, 1)) < 0)
64          user_panic("dup: %d", r);
65
```

图 6　unmain 函数部分内容

本次实验耗时 6 小时