管道和Shell

由于临近期末时间紧张，本次笔记从简。

管道

0 为读端，1 为写端。

相关定义

```
struct Dev devpipe = {
    .dev_id = 'p',
    .dev_name = "pipe",
    .dev_read = pipe_read,
    .dev_write = pipe_write,
    .dev_close = pipe_close,
    .dev_stat = pipe_stat,
};

#define BY2PIPE 32 // small to provoke races

struct Pipe {
    u_int p_rpos;          // read position，仅读者可更新
    u_int p_wpos;          // write position，仅写者可更新
    u_char p_buf[BY2PIPE]; // data buffer，循环队列，根据缓冲区情况需要控制读者写者阻塞
};
```

除此之外，我们还需要能够感知另一端是否关闭，避免一直等待已经不存在的对方。

相关函数

相关函数具体的说明在指导书中有。

```
// user/lib/pipe.c

// 首先为读端和写端分别申请文件描述符，并申请文件描述符相应的共享页，而后 syscall_mem_alloc
为读端申请页面，之后 syscall_mem_map 将其共享给写端，返回执行状态，文件描述符号存放在数组中
int pipe(int pfd[2]);

// 判断管道另一端是否关闭，通过 pageref(rfd) = pageref(pipe) 类似的方式，这个函数使用
while (runs != env->env_runs) 判断是否发生进程切换，如果没有，则认为读取的值是准确的
static int _pipe_is_closed(struct Fd *fd, struct Pipe *p);
```

Shell

相关的函数同样在指导书中进行了说明，此处不再赘述。

大致描述

对于管道而言，其文件描述符主要的作用就是判断管道另一端是否关闭，并不对读写进行控制。而对读写的控制似乎更多地只是靠约定俗成。对于 Shell，我们采取的方式是在 Shell 中不断地 fork，对于每一个新的命令行都 fork 出一个新的进程执行，如果遇到管道，就再接着 fork，而这些 fork 出来的运行器则会 spawn 出新的进程具体执行程序。