2024 操作系统 Lab6 串讲

2024 操作系统助教组

管道

管道的使用

- `pipe` 函数创建管道
  - 分配两个文件描述符 `fd`
  - 将两个文件描述符的数据对应的虚拟地址空间**映射到同一个物理页**
  - 设置两个文件描述符的属性
  - 返回 `fd[0]` 和 `fd[1]`

这一过程将页面作为一个环形队列缓冲区。向 `pipe` 函数返回的文件描述符的写入等同于在缓冲区的一端添加数据；而读取则等同于在缓冲区的一端取出数据。

虽然 `pipe` 返回了**文件描述符**，并以文件的方式进行读取，但管道并不涉及磁盘操作。这体现了“一切皆文件”的思想。

管道

Pipe 结构体

`Pipe` 结构体划分了分配给管道的页面空间。其中记录了当前读指针 `p_rpos` 和写指针 `p_wpos` 的位置。并划分 `PIPE_SIZE` 大小的区域作为缓冲区。

```
#define PIPE_SIZE 32 // small to provoke races

struct Pipe {
    u_int p_rpos;        // read position
    u_int p_wpos;        // write position
    u_char p_buf[PIPE_SIZE]; // data buffer
};
```

`p_rpos` 给出了下一个将要从管道读的数据的位置，而 `p_wpos` 给出了下一个将要向管道写的数据的位置。只有读者可以更新 `p_rpos` ，同样，只有写者可以更新 `p_wpos` 。

管道将 `PIPE_SIZE` 大小的缓冲区 `p_buf` 作为环形队列使用，因此下一个要读或写的位置 `i` 实际上是 `i % PIPE_SIZE` 。

管道

管道的使用

管道的读写（文件操作）：

- 读取（`read` 函数）
  - 当读指针落后于写指针时，从循环缓冲区中读取 1 字节，将读指针后移
  - 当读指针追上写指针时
    - 若写者已经关闭或已经读取了部分内容，返回当前读取的字节数
    - 否则，阻塞等待（写者没有关闭且未读取任何内容）
- 写入（`write` 函数）
  - 当写指针没有绕到读指针之后并追上读指针时，将 1 字节写入循环缓冲区，将写指针后移
  - 当写指针绕到读指针之后并追上读指针时
    - 若读者已经关闭，成功返回
    - 否则，阻塞等待（读者没有关闭且未写完所有内容）

管道

C 语言下的多态

不论是磁盘文件、管道抑或是标准输入输出，都能够通过统一的文件描述符接口进行控制，这背后正是多态的思想。

多态的本质是相同接口，不同行为。在 C 语言下实现多态，关键是函数指针的使用。对于一个函数指针变量，指针本身是统一的调用接口，而指针地址的不同则反映了不同的行为。

因而，我们使用如下 `Dev` 结构体定义了不同设备的统一接口。

```
struct Dev {
    int dev_id;
    char *dev_name;
    int (*dev_read)(struct Fd *, void *, u_int, u_int);
    int (*dev_write)(struct Fd *, const void *, u_int, u_int);
    int (*dev_close)(struct Fd *);
    int (*dev_stat)(struct Fd *, struct Stat *);
    int (*dev_seek)(struct Fd *, u_int);
};
```

类似的数据结构也是 C++、Java 等面向对象语言中多态的底层实现原理。

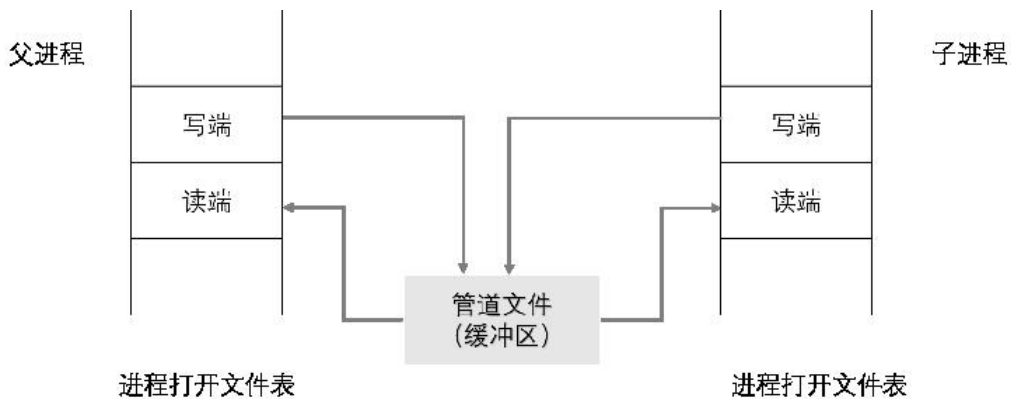
管道

管道的创建

`pipe` 函数传入一维数组 `pfd` 作为参数，通过设定 `pfd[0]` 和 `pfd[1]` 的取值返回读端和写端的文件描述符。

```
int pipe(int pfd[2]);
```

在 `pipe` 函数中，我们使用 `syscall_mem_alloc` 分配两个文件描述符以及 `Pipe` 结构体使用的内存空间，并设置 `PTE_LIBRARY` 权限位。该权限位使得 fork 之后父子进程能够共享页面，可以实现对同一缓冲区的读写，如下图所示。

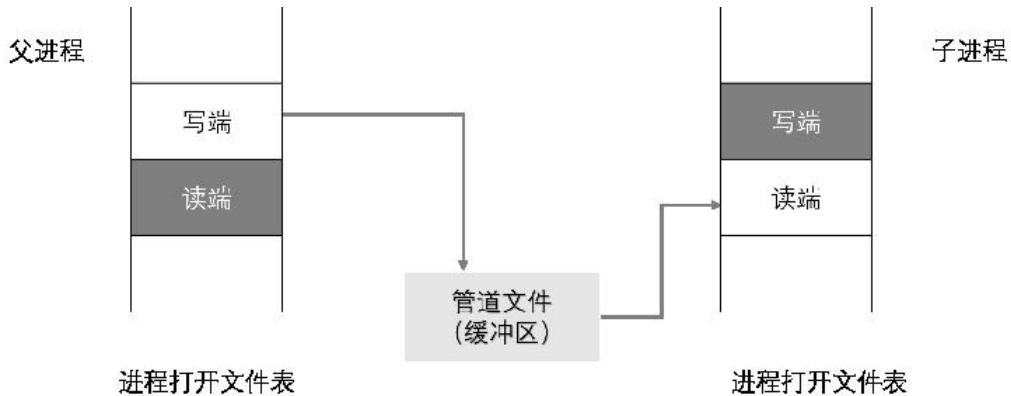


管道

管道的创建

管道的创建在 fork 前完成，在 fork 之后父子进程均持有两个文件描述符。然而队列是单向的，因而我们需要持有文件描述符的双方各关闭读写的一端。

如图所示，父进程关闭了读端文件描述符、子进程关闭了写端文件描述符，这样便构造了父进程写入、子进程读取的单向管道。



管道

管道的读取

`pipe_read` 实现了管道的读取操作（`.dev_read = pipe_read`）。

```
static int pipe_read(struct Fd *fd, void *vbuf, u_int n, u_int offset);
```

主要流程如下：

- 使用 `fd2data` 获取 `Pipe` 结构体
- 以字节为单位不断从缓冲区中读取数据
- 若**读指针**追上**写指针**（`p_rpos == p_wpos`），说明**缓冲区为空**。如果此时管道已经关闭，则直接返回；否则使用 `syscall_yield` 让出控制权。（此处为忙等待）
- 若缓冲区不为空，则读取一个字节的数据，并移动读指针 `p_rpos`
- 不断循环，直到读取所有 `n` 字节数据

管道

管道的写入

`pipe_write` 实现了管道的写入操作（`.dev_write = pipe_write`）。

```
static int pipe_write(struct Fd *fd, const void *vbuf, u_int n, u_int offset);
```

代码逻辑和 `pipe_read` 类似，主要流程如下：

- 使用 `fd2data` 获取 `Pipe` 结构体
- 以字节为单位不断向缓冲区中写入数据
- 若**写指针**追上**读指针**（`p_wpos - p_rpos == PIPE_SIZE`），说明**缓冲区已满**。如果此时管道已经关闭，则直接返回；否则使用 `syscall_yield` 让出控制权。（此处为忙等待）
- 若缓冲区未满，则写入一个字节的数据，并移动写指针 `p_wpos`
- 不断循环，直到写入所有 `n` 字节数据

管道

管道关闭的判断

在管道的读写操作中使用了 `_pipe_is_closed` 判断管道是否关闭。这是因为若读取时缓冲区为空或写入时缓冲区已满，则读/写进程应当等待另一进程的写/读操作。此时如果管道已经关闭，则永远不会进行写/读操作，这会导致当前进程永远等待下去。

判断管道是否关闭的关键在于如下恒等式：

```
pageref(rfd) + pageref(wfd) == pageref(pipe)
```

在正常情况下，pipe 页面被两个进程分别引用，而 `rfd` 和 `wfd` 则分别被一个进程引用，因此有 `1 + 1 == 2`。而当某一进程退出时，其持有的文件描述符（这里以 `wfd` 为例）和 pipe 页面的引用均减 1，这时有 `1 + 0 == 1`。因此我们只需要判断是否有 `pageref(rfd) == pageref(pipe)` 即可得知持有 `wfd` 的进程是否已经退出。

管道

管道的关闭

`pipe_close` 函数实现了管道的关闭。

```
static int pipe_close(struct Fd *fd);
```

需要注意，进程通过 `pipe_close` 关闭管道，本质是通过两次调用 `syscall_mem_unmap` 解除到文件描述符和缓冲区的映射。这意味着管道的关闭不是原子操作，在两次系统调用之间可能破坏恒等式。导致提前出现 `pageref(fd) == pageref(page)` 的情况。

导致问题出现的根本原因是 `pipe` 的引用次数总比 `fd` 高。当管道的 `close` 进行到一半时，若先解除 `pipe` 的映射，再解除 `fd` 的映射，就会使得 `pipe` 引用次数的 -1 先于 `fd` ，因此出现 `pageref(fd) == pageref(page)` 。所以，我们只需要让引用次数更小的 `fd` 先 -1，便可解决这一问题。

管道

管道的关闭

`_pipe_is_closed` 函数传入文件描述符 `fd` 和管道结构体 `p` 的地址，并返回条件 `pageref(fd) == pageref(p)` 的判断结果。

```
static int _pipe_is_closed(struct Fd *fd, struct Pipe *p);
```

这里我们还需解决一个问题，在 `_pipe_is_closed` 中需要两次调用 `pageref` 获取 `fd` 和 `p` 的引用次数，此过程也非原子操作。因此我们在两次调用 `pageref` 时，应当保证这两次调用间没有发生进程切换。为了实现这一点，我们需要使用到之前一直都没用到的变量 `env_runs`。

`env_runs` 记录了一个进程被调用 `env_run` 恢复执行的次数，我们可以在两次调用 `page_ref` 的前后分别查看 `env_runs` 的取值，如果两次取值相同，则没有发生进程切换，否则发生了进程切换。若发生了进程切换，则重复这一过程。

Shell

Shell 的启动

Shell 是一类为使用者提供操作界面的软件。分为 GUI Shell 和 CLI Shell，通常来说 Shell 指 CLI Shell。在 Lab6 中我们提供的 MOS Shell 就是一个简单的 CLI Shell。Shell 是用户与计算机交互的重要接口，了解其在 MOS 中的启动过程能让我们对操作系统有更深入的了解。

在 `mips_init` 的最后，我们使用 `ENV_CREATE` 创建了一个硬编码到内核当中的进程 `user_icode`。

```
void mips_init(u_int argc, char **argv, char **penv, u_int ram_low_size) {
    // ...
    ENV_CREATE(user_icode);
    ENV_CREATE(fs_serv);
    schedule(0);
    halt();
}
```

`user_icode` 的源码为 `user/icode.c`。该进程的主要作用是调用 `spawnl` 函数运行 `init.b` 程序。

```
spawnl("init.b", "init", "initarg1", "initarg2", NULL)
```

Shell

Shell 的启动

`init.b` 的源码为 `user/init.c`。其中我们调用 `opencons` 打开终端，返回文件描述符 `0` 作为标准输入；又使用 `dup` 函数复制出文件描述符 `1` 作为标准输出。

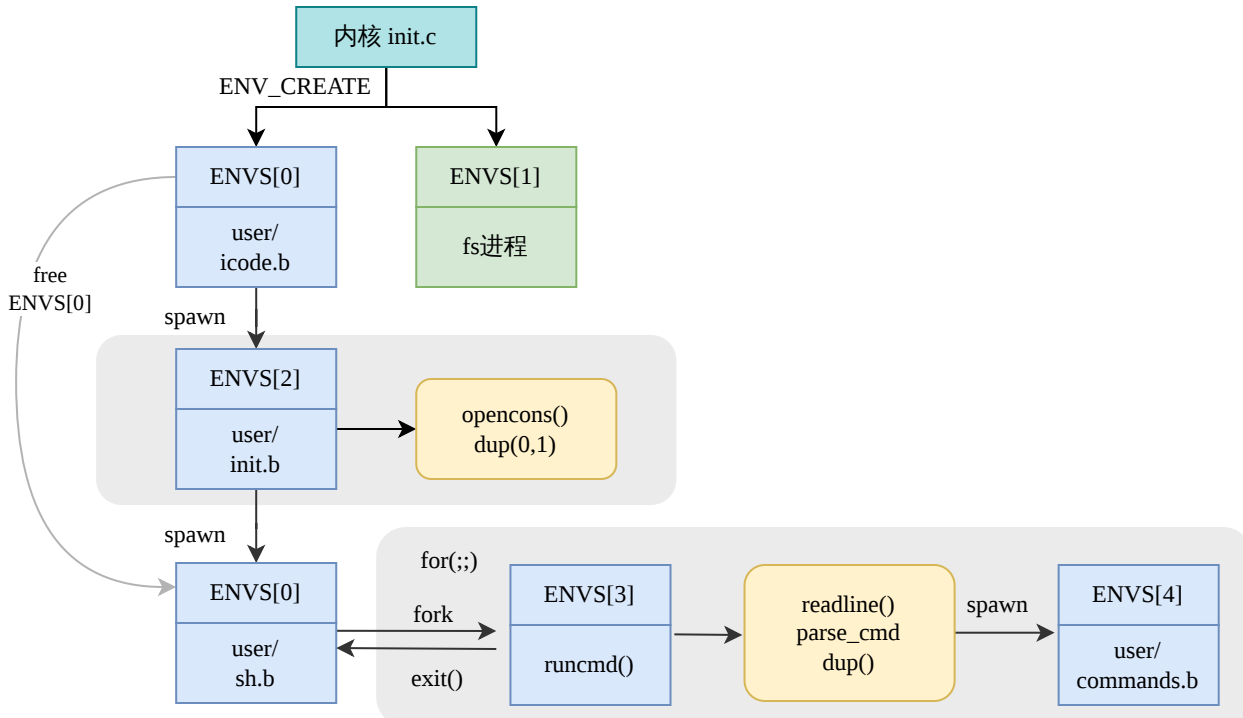
```
    if ((r = opencons()) != 0) {
        user_panic("opencons: %d", r);
    }
    if ((r = dup(0, 1)) < 0) {
        user_panic("dup: %d", r);
    }
```

最后，`init` 进程调用 `spawnl` 函数，运行 Shell 进程。

```
    while (1) {
        r = spawnl("sh.b", "sh", NULL);
        if (r < 0) {
            return r;
        }
        wait(r);
    }
```

Shell

Shell 的启动



Shell

spawn 函数

在前面，我们调用了 `spawnl` 函数。而其中实现将磁盘中的可执行文件加载到内存中并运行的功能的实际为 `spawn` 函数。该函数的主要流程如下

```
int spawn(char *prog, char **argv);
```

- 从文件系统读取指定的文件（二进制 ELF 文件，在 MOS 中为 `*.b`）
- 调用 `syscall_exofork` 申请新的进程控制块
- 调用 `init_stack` 为子进程初始化地址空间。对于栈空间，由于 `spawn` 需要将命令行参数传递给用户程序，所以要将参数也写入用户栈中
- 将目标程序加载到子进程的地址空间中，这里复用了 Lab3 中使用过的 `elf_load_seg` 函数
- 调用 `syscall_set_trapframe` 设置子进程的寄存器（栈指针 sp 和用户程序入口 EPC）
- 调用 `syscall_mem_map` 将父进程的共享页面映射到子进程的地址空间中
- 最后，调用 `syscall_set_env_status` 设置子进程为可执行状态

Shell

命令的解析

CLI Shell 的主要功能是解析用户输入的命令，根据命令执行程序并采取一系列附加操作。命令的解析与下学期的编译技术课程更加相关，因此在实验中并不需要各位同学实现。各位只需实现 Shell 的输入输出重定向以及管道功能。

- 对于输入输出重定向，本质是以读/写模式使用 `open` 函数打开对应文件，并使用 `dup` 函数将文件的描述符复制到标准输入（0）或标准输出（1）。
- 对于管道，本质是使用 `pipe` 函数创建管道，使用 `dup` 函数将写描述符复制到前一进程的标准输出（1）；将读描述符复制到后一进程的标准输入（0）。

课下习题提示

- Exercise 6.1

  1. 使用 `while` 而非 `if` 判断缓冲区是否为空或已满
  2. 使用 `_pipe_is_closed` 判断管道是否已关闭

- Exercise 6.2-3

  交换顺序即可

- Exercise 6.4

  1. 使用 `readn` 读取文件数据
  2. 在加载 ELF 段的循环中，需要首先使用 `seek` 定位段在文件中位置 `seek(fd, ph_off)`
  3. elf_load_seg 中的回调函数是 `spawn_mapper` ，`data` 为 `&child`

课下习题提示

- Exercise 6.5

  1. 使用 `open` 或 `pipe` 创建文件描述符
  2. 使用 `dup` 函数复制文件描述符后需要使用 `close` 关闭原文件描述符