# Lab 6

## 思考题

### Thinking 6.1

父子部分倒过来即可。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int fildes[2];

char buf[100];
int status;

int main() {

    status = pipe(fildes);

    if (status == -1) {
        printf("error\n");
    }

    switch (fork()) {
    case -1:
        break;

    case 0:                                     /* 子进程 - 作为管道的写者 */
        close(fildes[0]);                       /* 关闭不用的读端 */
        write(fildes[1], "Hello world\n", 12); /* 向管道中写数据 */
        close(fildes[1]);                       /* 写入结束，关闭写端 */
        exit(EXIT_SUCCESS);

    default:                                    /* 父进程 - 作为管道的读者 */
        close(fildes[1]);                       /* 关闭不用的写端 */
        read(fildes[0], buf, 100);              /* 从管道中读数据 */
        printf("father-process read:%s", buf); /* 打印读到的数据 */
        close(fildes[0]);                       /* 读取结束，关闭读端 */
        exit(EXIT_SUCCESS);
    }
}
```

### Thinking 6.2

1. `dup` 中需要先调用 `close` 来关闭 `newfd`，故 `close` 存在的问题 `dup` 同样存在。
2. 考虑情境如下：

```
pipe(p);
if (fork() == 0) {
    close(p[1]);
    read(p[0], buf, sizeof buf);
} else {
    dup(p[0], newfd);
    write(p[1], "Hello", 5);
}
```

- 假设 `fork` 结束后，子进程先执行。时钟中断产生在 `close(p[1])` 与 `read` 之间，父进程开始执行。
- 父进程在 `dup(p[0], newfd)` 过程中，已经完成了 `newfd` 对 `p[0]` 的映射，还没有完成 `newfd` 对 `pipe` 的映射。假设这时时钟中断产生，进程调度后子进程接着执行。
- 注意此时各个页的引用情况：`pageref(p[0])` = 3，而 `pageref(p[1])` = 1。但注意，此时 `pipe` 的 `pageref` 是 3，子进程中 `p[0]` 引用了 `pipe`，同时父进程中未完成 `newfd` 对 `pipe` 的映射，所以在 父进程中只有 `p[0]`、`p[1]` 引用了 `pipe`。
- 子进程执行 `read`，`read` 中首先判断写者是否关闭。比较 `pageref(pipe)` 与 `pageref(p[0])` 之后发现它们都是 3，说明写端已经关闭，于是子进程退出。

## Thinking 6.3

在 exception 发生时硬件关闭了对中断的响应，因此系统调用期间不会发生中断，故为原子操作。

详见 MIPS R3000 文档：

KUc,
IEc The two basic CPU protection bits.
KUc is set 0 when running with kernel privileges, 1 for user mode. In kernel mode, software can get at the whole program address space, and use privileged ("co-processor 0") instructions. User mode restricts software to program addresses between 0x0000 0000 and 0x7FFF FFFF, and can be denied permission to run privileged instructions; attempts to break the rules result in an exception.
IEc is set 0 to prevent the CPU taking any interrupt, 1 to enable.
KUp, IEp"KU previous, IE previous":
on an exception, the hardware takes the values of KUc and IEc and saves them here; at the same time as changing the values of KUc, IEc to [0, 0] (kernel mode, interrupts disabled). The instruction *rfe* can be used to copy KUp, IEp back into KUc, IEc.

## Thinking 6.4

在 `pipe_close` 中，先解除对 `fd` 的映射，再解除对 `pipe` 的映射，可以解决上述场景的进程竞争问题。`fd` 引用次数的 -1 先于 `pipe`。在两个 `unmap` 的间隙，`pageref(pipe) > pageref(fd)` 仍成立，即使此时发生中 断，也不会影响判断管道是否关闭的正确性。

会出现。在 `dup` 函数中，会先进行 `newfd` 对 `p[0]` 的映射，再进行 `newfd` 对 `pipe` 的映射，使得 `fd` 引用次数的 +1 先于 `pipe`，这就导致在两个 `map` 的间隙，会出现 `pageref(pipe) == pageref(fd)` 的情况。

## Thinking 6.5

前两个问题在题目中已经得到解答。

bss 段在 ELF 中并不占空间，但 ELF 的程序段头中存有该段需映射到的虚拟地址与占用空间。`elf_load_seg()` 函数读取后调用 `map_page()` 回调函数映射到内存，对于超出 `bin_size` 的部分，即 bss 部分，回调函数的 `src` 参数为 `NULL`。而在此处加载的回调函数 `load_icode_mapper()` 中，在 `src == NULL` 时会直接分配并映射一页物理内存。由于物理内存在分配时会初始化为 0，故加载后的 bss 段初始值为 0。

## Thinking 6.6

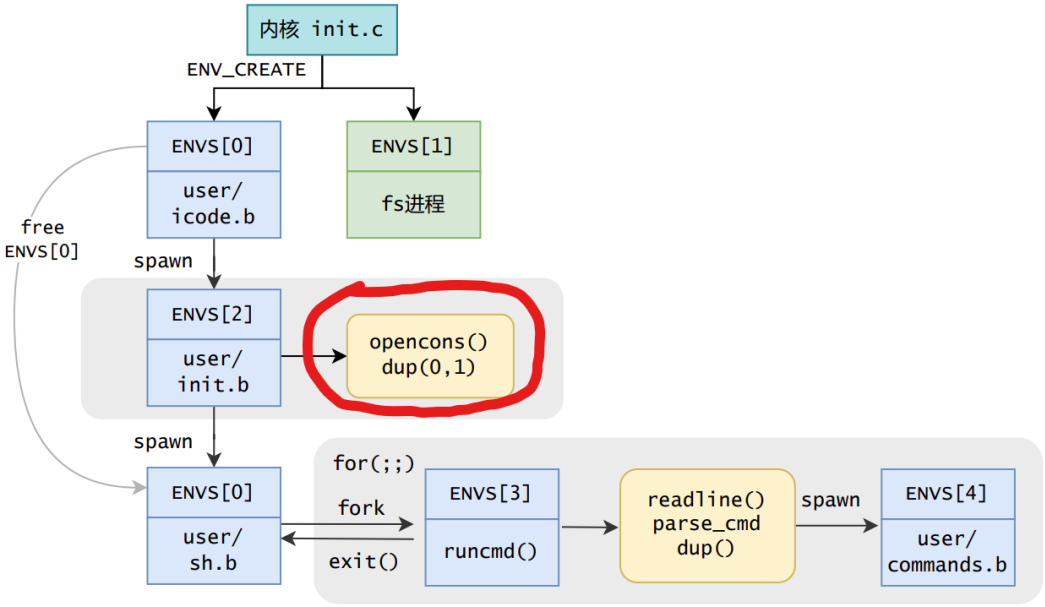


图 6.4: shell 启动执行过程

`user/init.c` 中。

```
    // stdin should be 0, because no file descriptors are open yet
    if ((r = opencons()) != 0) {
        user_panic("opencons: %d", r);
    }
    // stdout, dup to 1
    if ((r = dup(0, 1)) < 0) {
        user_panic("dup: %d", r);
    }
```

## Thinking 6.7

外部命令。因为 `cd` 会改变 shell 的状态。

## Thinking 6.8

两次 `spawn`，分别为生成 `ls.b` 和 `cat.b`。

四次进程销毁，分别为 `ls.b` 进程 `cat.b` 进程、生成 `cat.b` 的 shell 子进程和生成 `ls.b` 的 shell 子进程。

# 难点分析

shell 部分较为简单，主要难点在于对管道的理解和管道关闭的判断。

# 实验感想

Lab6 的实验带领我们最终完成了我们的操作系统，在看到 shell 成功运行的时候，相信无论是谁都会成就感满满 。尽管一学期的实验并不轻松，但也有许多收获。*It was all worth it, wasn't it?*