Lab6 实验报告

零、实验目的

1. 掌握管道的原理与底层细节
2. 实现管道的读写
3. 复述管道竞争情景
4. 实现基本 shell
5. 实现 shell 中涉及管道的部分

一、思考题

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？

```
#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\n", 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

> 上面这种不同步修改 `pp_ref` 而导致的进程竞争问题在 `user/lib/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` 函数将一个文件描述块复制到另一个文件描述块，对于一个匿名管道，就是 `pipe` 所在的数据块。在过程中，父进程和子进程产生数据冒险， `dup` 函数可能在运行中间改变文件描述符的引用次数导致出现不可预料的问题。

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说明。

在系统调用前关闭了中断，不会在 `syscall` 中再次出现系统调用，因此 MOS 中系统调用一定是原子操作。

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 `pipe_close` 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，在 `fd.c` 中有一个 `dup` 函数，用于复制文件描述符。试想，如果要复制的文件描述符指向一个管道，那么是否会出现与 `close` 类似的问题？请模仿上述材料写写你的理解。

可以解决进程竞争问题。

这里考虑的完全是 `pageref` 。显然有 `pageref(pipe) >= pageref(fd)` ，而先取消 `pipe` 的映射，可能导致应该是 `pageref(pipe)>pageref(fd)` 变成 `pageref(pipe)==pageref(fd)` ，但是先取消 `fd` 的映射就不可能出现这个情况，因此不会误判另一端已经关闭。

可能出现上述问题，在6.3已经提到了原因。

`dup` 函数也会出现与 `close` 类似的问题，也就是可能因为先进行 `fd` 的 `pageref` 添加导致出现 `pageref(pipe)==pageref(fd)` ，因此先增加 `pageref(pipe)` 就不会出现这种情况，这里事实上是认为过程中任何一个时刻都不能出现满足那个情况的时候，因为很有可能会出现在那个时刻的前一时刻进行进程切换导致出现问题的情况。

Thinking 6.5

> 思考以下三个问题。
>
> - 认真回看 Lab5 文件系统相关代码，弄清打开文件的过程。
> - 回顾 Lab1 与 Lab3，思考如何读取并加载 ELF 文件。
> - 在 Lab1 中我们介绍了 `data text bss` 段及它们的含义， `data` 段存放初始化过的全局变量， `bss` 段存放未初始化的全局变量。关于 `memsize` 和 `filesize` ，我们在 Note 1.3.4中也解释了它们的含义与特点。关于 Note 1.3.4，注意其中关于“ `bss` 段并不在文件中占数据”表述的含义。回顾 Lab3 并思考： `elf_load_seg()` 和 `load_icode_mapper()` 函数是如何确

> 保加载 ELF 文件时，`bss` 段数据被正确加载进虚拟内存空间。`bss` 段 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。请回顾 `elf_load_seg()` 和 `load_icode_mapper()` 的实现，思考这一点是如何实现的？下面给出一些对于上述问题的提示，以便大家更好地把握加载内核进程和加载用户进程的 区别与联系，类比完成 `spawn` 函数。

关于第一个问题，在 Lab3 中我们创建进程，并且通过 `ENV_CREATE(...)` 在内核态加载了初始进程，而我们的 `spawn` 函数则是通过和文件系统交互，取得文件描述块，进而找到 ELF 在“硬盘”中的位置，进而读取。
关于第二个问题，已经在 Lab3 中填写了 `load_icode` 函数，实现了 ELF 可执行文件中读取数据并加载到内存空间，其中通过调用 `elf_load_seg` 函数来加载各个程序段。 在 Lab3 中我们要填写 `load_icode_mapper` 回调函数，在内核态下加载 ELF 数据到内存空间；相应地，在 Lab6 中 `spawn` 函数也需要在用户态下使用系统调用为 ELF 数据分配空间。

Thinking 6.6

> 通过阅读代码空白段的注释我们知道，将标准输入或输出定向到文件，需要我们将其 `dup` 到 0 或 1 号文件描述符（`fd`）。那么问题来了：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。

在 `user/init.c`：

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

Thinking 6.7

> 在 `shell` 中执行的命令分为内置命令和外部命令。在执行内置命令时 `shell` 不需要 `fork` 一个子 `shell`，如 Linux 系统中的 `cd` 命令。在执行外部命令时 `shell` 需要 `fork` 一个子 `shell`，然后子 `shell` 去执行这条命令。 据此判断，在 MOS 中我们用到的 `shell` 命令是内置命令还是外部命令？请思考为什么 Linux 的 `cd` 命令是内部命令而不是外部命令？

内置命令是 shell 程序自己的功能，直接执行，不需要创建新的进程；外部命令是磁盘上的程序，需要创建新的进程。

MOS 中用到的 shell 命令是外部命令。

`cd` 用于改变当前的工作目录。如果是外部命令，会在子 shell 中执行改变目录，父 shell 的工作目录没有被改变。因此必须是内置命令。

Thinking 6.8

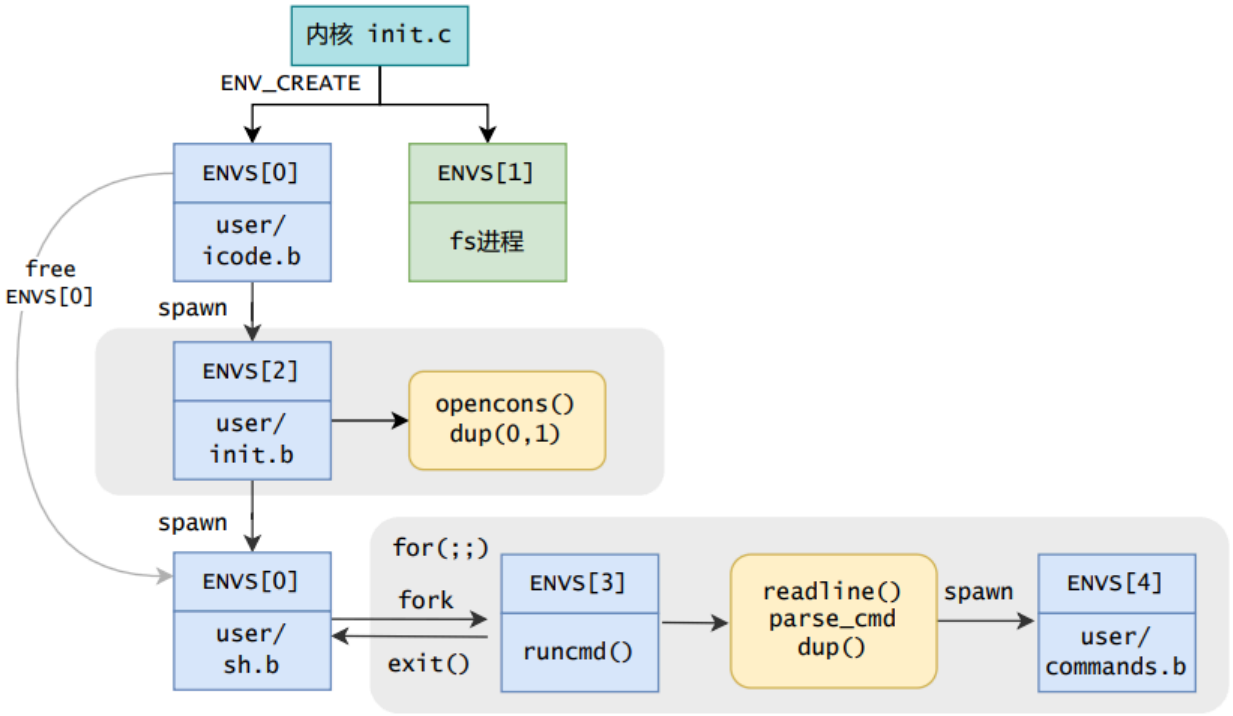
> 在你的 `shell` 中输入命令 `ls.b | cat.b > motd`。

> - 请问你可以在你的 `shell` 中观察到几次 `spawn` ？分别对应哪个进程?
> - 请问你可以在你的 `shell` 中观察到几次进程销毁？分别对应哪个进程?

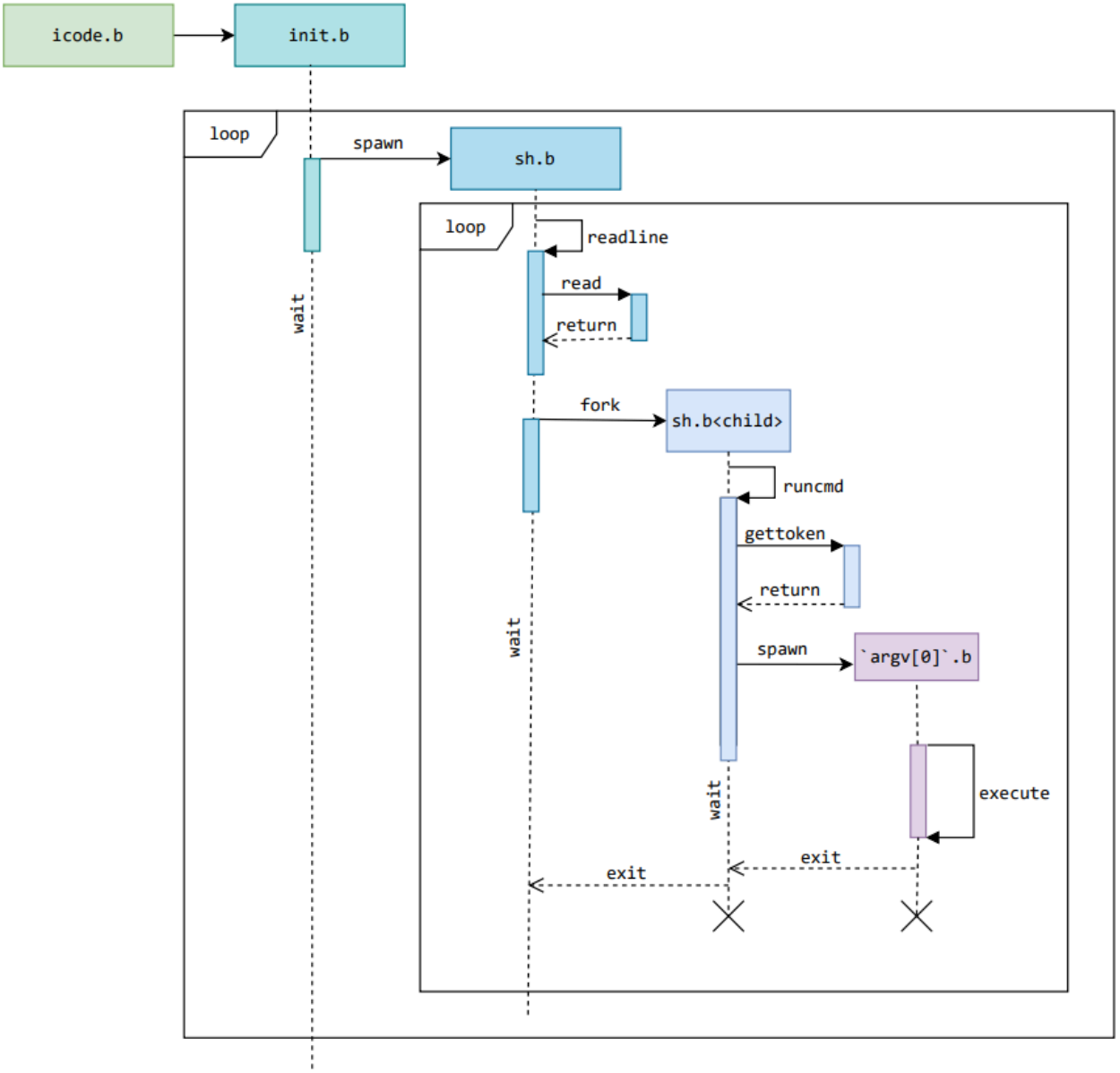
```
[00002803] pipecreate
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00004006] destroying 00004006
[00004006] free env 00004006
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...
```

二、实验难点

shell 启动执行过程



shell 相关函数调用关系



三、实验体会

居然是最后一个 lab 了！OS 实验接近尾声！

本学期线上测试都采用了对照标答学习的方法，自己画图总结重点，效果其实还不错！

限时测试每一次 exam 都 100 分了，但 extra 只写出一次。也算没有遗憾了！