北航操作系统实验报告 Lab 6

学号：71066001

姓名：陈伟杰

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？ ■

答：

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.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("child-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** 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？ ■

答：在我们 dup 函数里，我们可以知道主要的目的是将一个文件描述符 oldfdnum 对应的文件映射到另一个文件描述符 newfdnum 上，出现预想之外的情况有比如我们 dup 函数会先将旧文件描述符的引用次数加 1，然后将新文件描述符的引用次数变为旧文件描述符的引用次数。也就是说，调用 dup 函数会先将 fd[0] 的引用次数加 1，再将 pipe 的引用次数加 1。如果这两个操作之间发生了时钟中断，可能会导致引用计数不正确并且可能导致管道读端的错误关闭。为了避免这种情况，您可以在调用 dup 函数之前增加对 fd[0] 和 pipe 引用计数的显式更新，并确保这些更新发生在单个操作的上下文中，以避免时钟中断的问题

**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。 ■

答：系统调用的原子性来源于操作系统的内核态执行机制即不会被其他进程或中断干扰。然而，并非所有的系统调用都是原子操作。fork() 系统调用不是原子操作。

下面是一个简单的 fork() 调用的例子，可以看到在 fork() 调用之后，父进程和子进程的执行是并行的，不是原子操作：

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

int main() {
    pid_t pid = fork();
    if (pid == -1) {
        printf("fork failed\n");
        return -1;
    }
    if (pid == 0) {
        // Child process
        printf("Child process\n");
        sleep(1);
    } else {
        // Parent process
        printf("Parent process\n");
        sleep(1);
    }
    return 0;
}
```

**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，用于复制文件内容。试想，如果要复制的是一个管道，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。

■

答：先关闭 fd[0]，再取消映射 pipe，这样可以解决上述场景的竞争问题。因为我们在关闭 fd[0] 的时候更新了 pipe 的计数，所以在取消映射 pipe 的时候，不会出现引用计数错误的情况，所以如果在复制管道之前，有其他进程正在使用相同的管道描述符，则可能会出现数据丢失或消息传递中断的不可预测行为。

**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，各位已经在 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。

■

答：第一个问题答案：过程就是当用户进程试图打开一个文件时，文件系统服务进程需要一个文件描述符来存储文件的基本信息和用户进程中关于文件的状态，若成功打开文件，则该函数返回文件描述符的编号，总的来说调用 open 函数，访问权限，解析所获得的 inode。

第二个问题答案：加载 ELF 头，加载程序头，比如 section 的位置和大小，段的权限，加载各个段到内存中的不同区域，例如代码段、数据段、堆和栈等最后根据程序头中提供的信息进行配置

第三个问题答案：在 load_icode_mapper 和 elf_load_seg()这个函数里在加载 ELF 文件时，不需要特别处理 bss 段的加载。因为会在 setup_vm 函数中确保正确地初始化 bss 段虚拟内存，在 elf_load_seg（）函数中，如果段类型是 PT_LOAD，它将调用传递给它的回调函数 load_icode_mapper 来处理该段的每个页面，现在返回在 load_icode_mapper 函数中，如果 src 为 NULL，则不会复制任何数据到该页中，因此不会从文件中加载 bss 段的数据。相反，当 setup_vm 函数调用 memset 函数将整个 bss 段的虚拟内存初始化为 0。

**Thinking 6.6** 通过阅读代码空白段的注释我们知道，将文件复制给标准输入或输出，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。 ■

答：在 user/init.c 中，有如下代码，在此处将 0 和 1 安排为了标准输入输出

```
// 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？ ■

答：在 MOS 中我们用到的 Shell 命令是是外部命令，Linux 的 cd 命令是内部命令是因为 cd 命令需要改变当前工作目录，而工作目录是一个 shell 属性，cd 命令它不可能改变父进程的工作目录改变的是自己的工作目录状态。

**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？

■

答：
在我的 shell 中观察到两次 spawn 分别是 3805 和 4006 进程
在我的 shell 中观察到 4 次进程销毁分别是 3805，4006，3004，2803 四个进程

实验困难
本次 lab 消耗的时间比较长，在毫无进展的时候都是参考往届同学的代码，更多的困扰我是在填写有关管道的函数比如_pipe_is_closed,pipe_read,pipe_write 这几个函数琢磨，因为这几个函数如果不解决和熟透的话对后面很难搞，总的来说还需要更细致的学习。

心得体会
这次 lab 相对来说虽然内容不多，但也是花了自己很长的时间，主要理解代码花了比较多时间，一直在脑海模拟，由于没有上机测试，所以自己在做这个 lab 的时候是一个“走马观花”“急”的想法，望考试结束后，自己能继续深入了解这个 lab 的乐趣，最后就是整个 OS 的实验就结束了，在整个实验中自己学到了很多，课程组的推进也是非常合理，理论搭配实验的方法能让我更加了解操作系统的原理，学到了很多，望自己越来越好！学到东西才是最好！！