lab 6 实验报告

班级：222115

学号：22373386

姓名：高铭

一、思考题 (Thinking)

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？

```
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

> 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` 的作用是将 `oldfdnum` 指向的数据复制给 `newfdnum`，一共包含两次 `map` 过程：
  - 将 `newfd` 所在的虚拟页映射到 `oldfd` 所在的物理页
  - 将 `newfd` 的数据所在的虚拟页映射到 `oldfd` 的数据所在的物理页。
- 考虑如下代码：

```
// 子进程
    read(p[0], buf, sizeof(buf));
// 父进程
    dup(p[0], newfd);
    write(p[1], "Hello", 5);
```

- `fork` 结束后，子进程先进行。但是在 `read` 之前发生了时钟中断，此时父进程开始进行进行。
- 父进程在 `dup(p[0])` 中，已经完成了对 `p[0]` 的映射，这个时候发生了中断，还没有来得及完成对 `pipe` 的映射。
- 此时回到子进程，进入 `read` 函数，但是 `ref(p[0])` 与 `ref(pipe)` 的值均为2，认为此时写进程关闭。

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 entry.S 中，有如下语句：

```
exc_gen_entry:
        SAVE_ALL
        mfc0    t0, CP0_STATUS
        # 保持处理器处于内核态(UM==0)，关闭中断(STATUS_IE)，允许嵌套异常
        and     t0, t0, ~(STATUS_UM | STATUS_EXL | STATUS_IE)
        mtc0    t0, CP0_STATUS
```

`syscall` 跳转到内核态时，我们禁用全局中断，因此系统调用不会被中断，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。

- 可以解决上面场景所述的进程竞争问题。指导书中说，`ref(p[0]) < ref(pipe)` 恒成立，因此如果先解除 `p[0]` 的映射，则 `ref(p[0])` 更要小于 `ref(pipe)`，永远不会出现 `ref(p[0]) == ref(pipe)`。
- `dup` 函数也会出现与 `close` 类似的问题。因为 `pipe` 的引用次数总比 `fd` 要高，当管道的 `dup` 进行到一半时， 若先映射 `fd`，再映射 `pipe`，就会使得 `fd` 的引用次数的+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。这个问题也可以通过调换两个map的顺序来解决。

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?

- 文件的打开过程： `user/lib/files.c` 文件中的 `open` 函数调用同文件夹下的 `fsipc_open` 函数, `fsipc_open` 通过调用 `fsipc` 函数向服务进程进行进程间通信，并接收返回的消息。相应文件系统服务进程 `serve_open` 函数调用 `file_open` 对进行文件打开操作，最终通过IPC机制实现与用户进程对文件描述符的共享。
- 我们通过 `kern/env.c` 文件中的 `load_icode` 函数来读取并加载ELF文件。
- **`elf_load_seg` 函数**：在该函数处理程序的循环中，当处理到.bss段时，该函数会调用 `map_page` 把相应的虚拟地址映射到物理页上，但不会从文件中加载数据。而在 `map_page` 的内部会调用 `load_icode_mapper` 函数将页面进行映射并根据参数将内容置为0；
- **`load_icode_mapper` 函数**：当处理到.bss段时，不从源数据中复制任何内容。最终调用 `page_insert` 函数将其置入页表并指定权限。该函数会根据传入的参数在页表项中建立映射关系，并初始化页面为0.

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中有如下代码段：

```
1   // stdin should be 0, because no file descriptors are open yet
2   if ((r = opencons()) != 0) {
3           user_panic("opencons: %d", r);
4   }
5   // stdout
6   if ((r = dup(0, 1)) < 0) {
7           user_panic("dup: %d", r);
8   }
```

在进程初始化的时候，0和1被安排为标准输入和标准输出。

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？

```
for (;;) {
        if (interactive) {
                printf("\n$ ");
        }
        readline(buf, sizeof buf);

        if (buf[0] == '#') {
                continue;
        }
        if (echocmds) {
                printf("# %s\n", buf);
        }
        if ((r = fork()) < 0) {
                user_panic("fork: %d", r);
        }
        if (r == 0) {
                runcmd(buf);
                exit();
        } else {
                wait(r);
        }
}
```

上述代码是user/sh.c中 `main()` 函数的部分，可见MOS中需要 `fork` 一个子shell来处理输入的命令，因此是外置命令。

我认为是因为Linux的 `cd` 指令对使用频率和速度的要求高，若多次使用 `cd` 指令则会多次调用 `fork` 生成子shell，此种方法过于低效，故选择外部命令。

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？

输出如下：

```
[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 ...
```

- 可以观察到两次 `spawn`，分别打开了 ls.b 和 cat.b 进程。
- 有4次进程销毁，分别是左指令的执行进程、右指令的执行进程和 `spawn` 打开的两个执行进程

二、难点分析

对我而言，本次实验的难点在于理解shell的全局运行流程。在编写代码的时候，常常要结合前几次lab，特别是lab1、lab3、lab4这几次实验，需要回顾 `fork` 和加载等过程，才能把知识点做延伸。

此外，我第一次提交时最后一个点无法通过，经反复检查、查看讨论区，发现有同学和我一样在编写 user/sh.c 中的 `parsecmd` 函数时，在执行 `fd = open(t, ...)` 时没有处理 fd<0 的情况。这也再次提醒了我要时刻注意细节的处理。

三、实验体会

总体而言，我认为lab6的难度并不高，但是由于结合了整个学期前面所学的知识，需要花费一点时间去复习以前写的代码。

理解shell的整个过程以后，OS实验算是正式结束了。从lab0到lab6的一次次实验，从底层到顶层，从微观到宏观，我对操作系统的理解进一步加深了。如果说OS理论课是高屋建瓴地纵览操作系统的各部分，实验课则是深入其中，自己动手去实现理论课所学的东西。二者互补，让我对于知识点的掌握更为牢固。虽然这学期每个隔周周三都要经历强度颇高的上机考试，每次课下的任务量也不可谓不大，但是收获颇丰，很值得！