实验目的

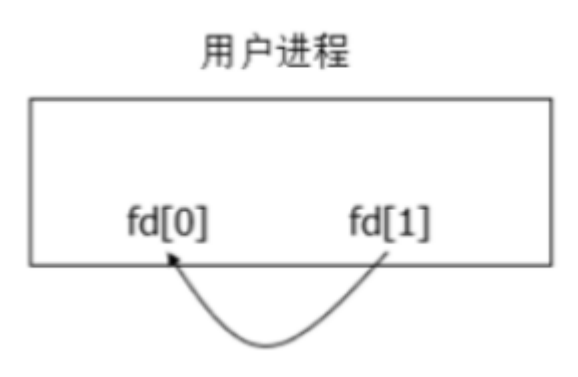
- 掌握管道的原理与底层细节
- 实现管道的读写
- 复述管道竞争情景
- 实现基本 shell
- 实现 shell 中涉及管道的部分

实验难点

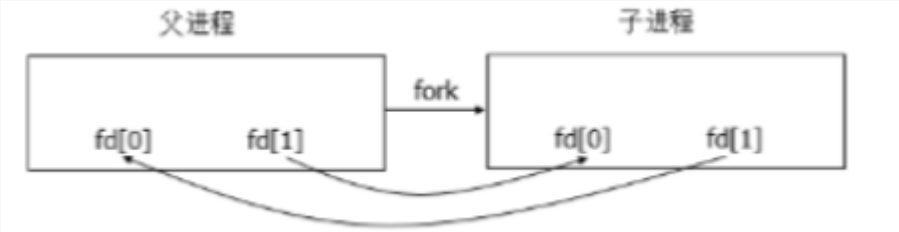
管道

其实这部分的理解难点是关于文件描述符的部分和构造管道时的一系列映射过程，但由于笔者选修了 unix 一般专业课，在大作业中已经自己实现过了一遍管道机制，因此再回到操作系统实验中理解、处理起来就比较轻松了。在这里笔者将父子进程间建立管道的机制描述如下：

- `int pipe(int filedes[2])` - 使用 pipe 函数来创建一个管道
  - 创建一个管道，若成功返回 0，不成功返回 -1
  - 由参数 filedes 返回两个文件描述符：filedes[0] 和 filedes[1]，前者为读而打开，后者为写而打开
    - 创建管道之后的情形如下图所示，将数据写入 fd[1]，从 fd[0] 中读出：



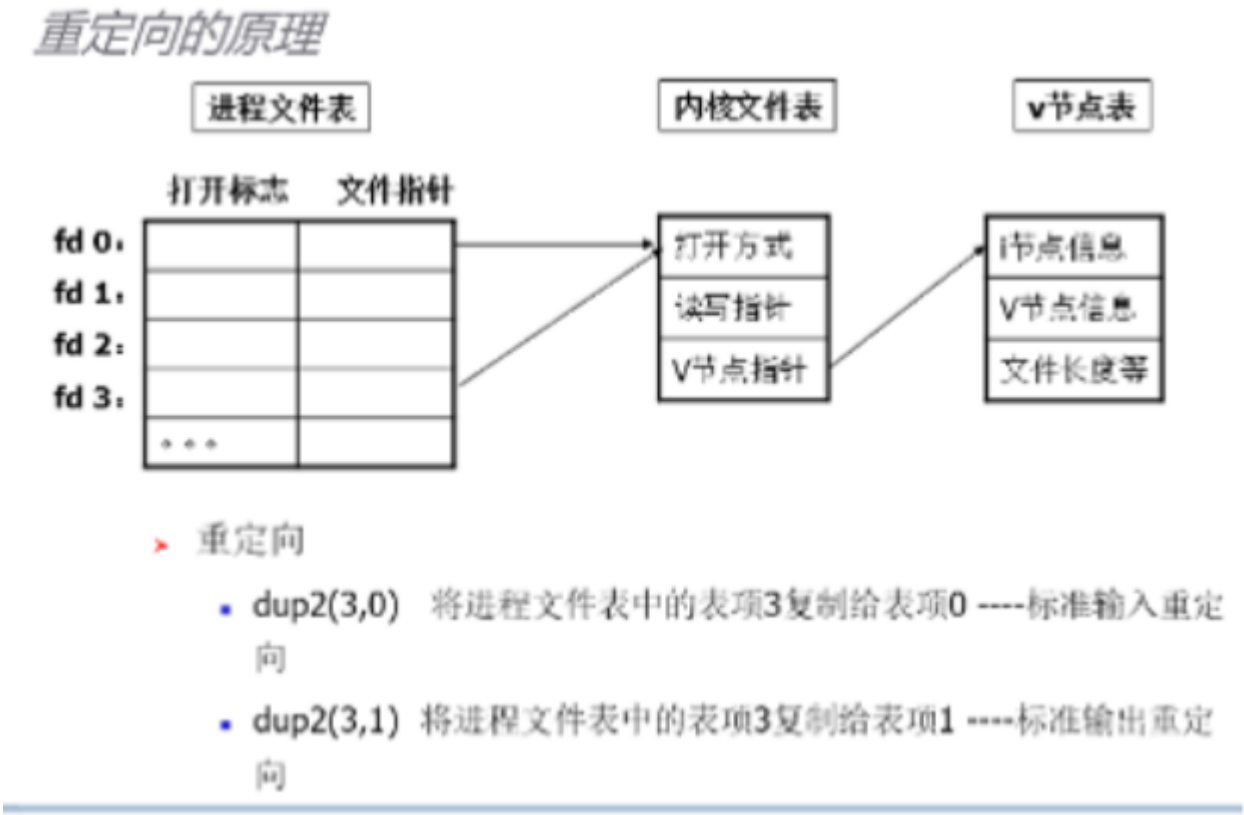
- 管道在单个进程中没有意义，通常在 pipe 函数之后，会立即调用 fork，产生一个子进程，这也是本实验中的应用场景：



- fork 之后，选择通信方向：如果父进程写、子进程读，则父进程关闭读端，子进程关闭写端，分别通过调用 close(fd[0]) 和 close(fd[1]) 函数实现；反之，子进程关闭 fd[0]，父进程关闭 fd[1]。
  - 规则
    - 写端关闭时，read 函数返回 0，表示文件结束
    - 读端关闭时，write 函数返回 -1，并出现 SIGPIPE 异常（信号）
  - 常量 PIPE_BUF 规定了内核管道缓冲区的大小，每次 write 的字节数需要小于它

捋清楚了这个过程后，管道的实现就变得比较容易了。很大程度上还是吃了 unix 大作业的老本。

创建管道的基础是 pipe 函数，该函数执行成功时会修改文件描述符数组的内容。按照约定，shell 启动新应用程序时会将 0、1、2 这 3 个数字的描述符打开为 stdin、stdout、stderr，在此处 pipe 函数将 fd[0] 映射到 read 端，fd[1] 映射到 write 端，因此向管道文件读写数据的过程其实是在读写内核的缓冲区。至于映射的细节，则要追究到 dup 函数。



当然还有一个有意思的问题，就是指导书中花了大篇幅来口头模拟的**管道竞争**问题，这本质上就是一个安全问题——在 java 等语言中，我们当然可以通过加锁等操作来简单地实现安全机制，但在 C 里这个过程就比较繁琐了。

由于时钟中断可能会在函数执行的中间发生（比如我们模拟的 dup 还有 close），因此我们在 _pipeisclosed() 中使用了 env_runs 变量来对进程执行次数进行计数，通过判断是否在同一时间片内，以确认我们对 pageref 的读取过程中没有发生进程切换，进而保证修改的同步性，当然也可以调整映射顺序，以使得管道机制中 pageref(fd) 和 pageref(pipe) 的大小关系严格恒成立，防止发生对读端或写端开关情况的误判。

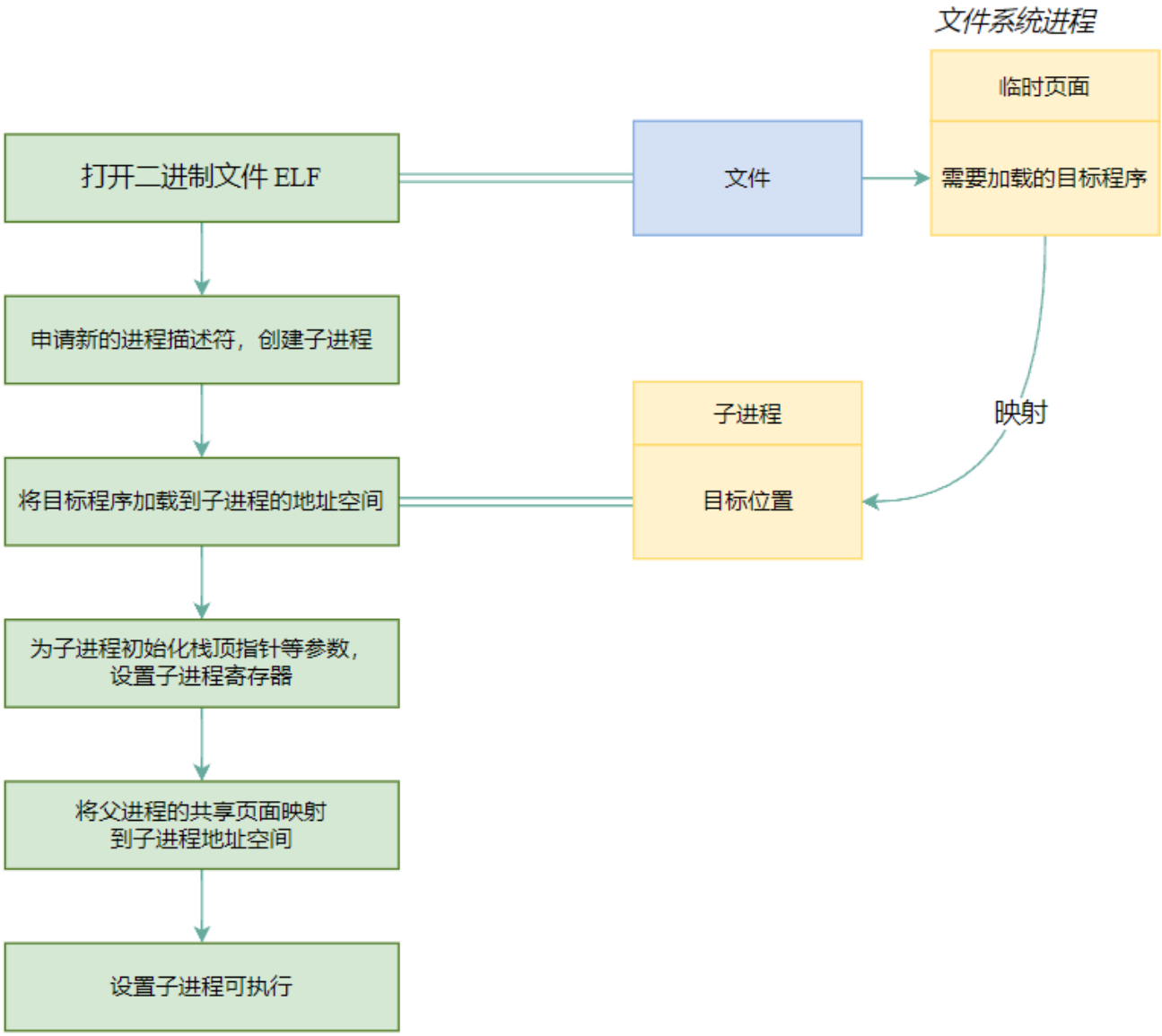
spawn 函数

个人认为 spawn 函数是整个 OS 课设中笔者遇到的最大困难，修改了无数次才通过评测……虽然指导书中粗略地分解了它的流程，但实现起来还是遇到了各种问题。事实上主要的问题是在用户态下实现 load_icode_mapper 函数以获取二进制文件中的 .text 段，剩下的过程跟着流程机械执行就好了。只不过其中考察的内容几乎横跨了之前做过的所有 lab，所以书写的过程还是要细心。

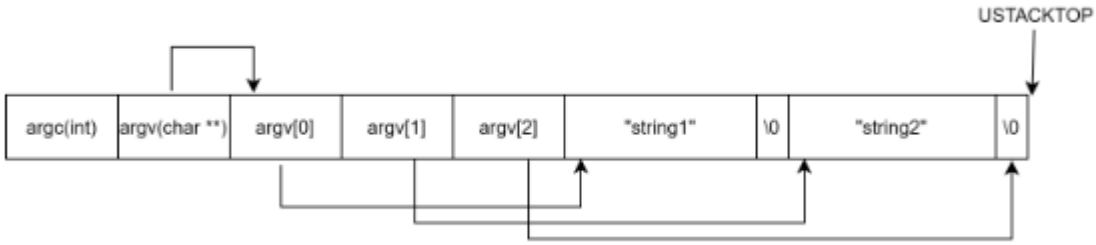
- **目标程序的加载**

  由于此时在用户态下，因此传递两个进程之间的内容时要通过系统调用。我们需要支持一个临时页面来实现在用户态加载二进制文件，即：将文件系统进程的一个临时页面映射到子进程所需要的页面，然后文件系统在该页面的具体位置上复制内容，使得子进程的相应位置出现对应的信息，最后文件系统进程再解除这个暂时页面的映射，开始下一个位置的映射。

  这个临时页面其实就相当于一个**工具页**，不得不说，“工具页”的思维其实是这个函数的实现核心。



- **为子进程初始化栈空间**



用户态没法直接操作子进程的栈空间，因此还是使用“工具页”的思路，将准备的参数填充到 TMPPAGE 处，然后再把 TMPPAGE 映射到子进程的栈空间里。

shell

在这里笔者依然很大程度上吃了 unix 大作业的老本，自己实现过一遍 shell 程序后就会觉得比较好理解。

shell 起到的事实上是一个**字符串解析**的作用，我们需要补全的也就是一些机械的解析过程，让程序根据解析到的内容获取参数等，然后执行特定功能的函数。这一部分难度比较小，也没什么需要特别说明的。

思考题

Thinking 6.1

如果需要让父进程为“读者”，则需要关闭父进程的写通道。由于 fork() 函数执行完毕并返回后会先执行父进程再执行子进程，因此在进行正式的读写操作之前应该先调度一下，让子进程先执行写操作，才能使得父进程为读者。

具体代码修改如下：

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

int fildes[2];
/* buf size is 100 */
char buf[100];
int status;

int main()
{
    status = pipe(fildes);
    if (status == -1) {
        /* an error occurred */
        printf("error\n");
    }

    switch (fork()) {
        case -1:    /* Handle error */
            break;
        case 0:     /* Child - writes to pipe */
            close(fildes[0]);                       /* Read end is unused */
            write(fildes[1], "Hello world\n", 12);  /* Write data on pipe */
            close(fildes[1]);                       /* Child will see EOF */
            exit(EXIT_SUCCESS);
        default:    /* Parent - reads from pipe */
            yield();                                /* Switch to child process firstly. */
            close(fildes[1]);                       /* Write end is unused */
            read(fildes[0], buf, 100);              /* Get data from pipe */
            printf("child-process read:%s", buf);   /* Print the data */
            close(fildes[0]);
    }
}

```

Thinking 6.2

可以给出如下所示的场景：

```
if (fork() == 0) {
    dup(p[1]);
    read(p[0]);
} else {
    dup(p[0]);
    write(p[1]);
}
```

由于 dup 函数的功能是将一个文件描述符所对应的内容映射到另外一个描述符中，因此：

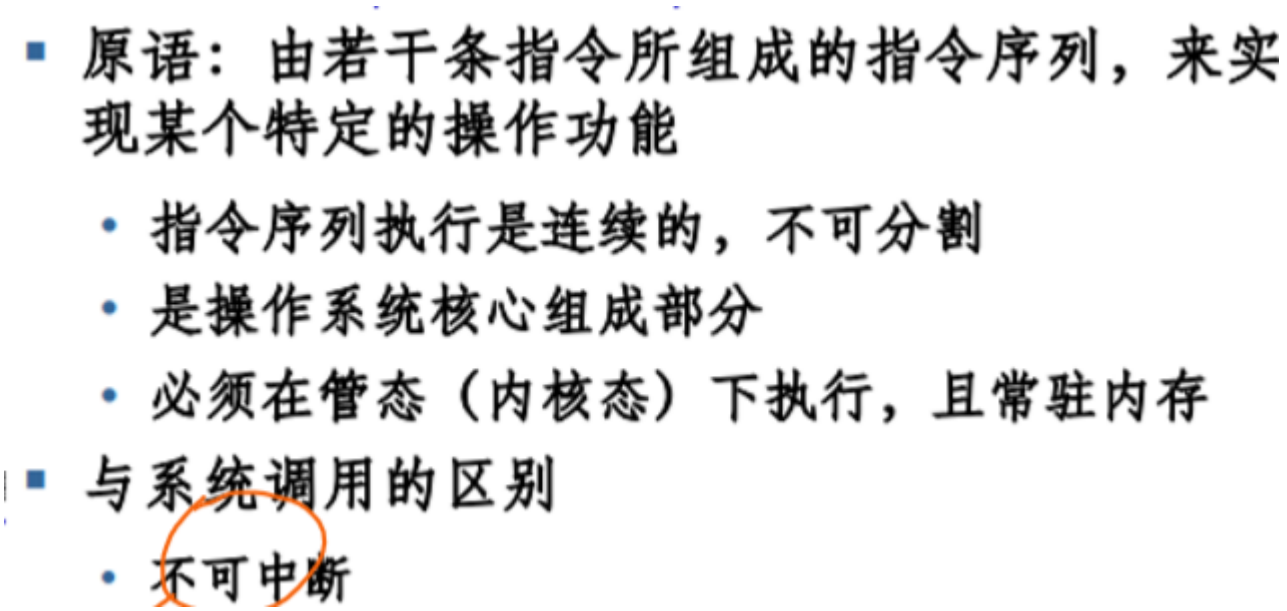
- fork 之后，子进程先开始执行，时钟中断产生在 dup(p[1]) 和 read 之间，父进程开始执行。父进程在 dup(p[0]) 中，p[0] 已经解除了对 pipe 的映射（unmap），还没有来得及解除对 p[0] 的映射，时钟中断产生。
- 执行 dup 映射之前，pageref(p[1]) == pageref(p[0]) == 1, pageref(pipe) == 2；然后，由于在 dup 函数中需要先对 fd 执行一次 syscall_mem_map，再对 data 执行 syscall_mem_map（在这里对 pipe 进行操作），因此会先将 p[0] 的引用次数 +1，此时 pageref(p[0]) == 2。
- 而如果在这个时刻恰好发生了一次时钟中断，进程切换后，子进程首先就会调用 _pipeisclosed 函数判断写者是否关闭，而此刻比较 pageref(p[0]) 与 pageref(pipe) 之后发现它们都是 2，说明写端已经关闭，于是子进程退出。这样就造成了错误。

Thinking 6.3

进行系统调用时，可以发现如下代码：

```
/* /include/stackframe.h */
.macro CLI
    mfc0 t0, CP0_STATUS
    li t1, (STATUS_CU0 | 0x1)
    or t0, t1
    xor t0, 0x1
    mtc0 t0, CP0_STATUS
.endm
```

可见进行系统调用时系统陷入内核，会将时钟中断关闭，以保证系统调用的过程不会被打断（事实上似乎只在咱们实验的 MOS 系统中是这种情况？Linux 中并非所有的系统调用都是原子操作——否则操作系统理论课的内容就不好解释了。），因此在这里系统调用都是原子操作。



Thinking 6.4

- 可以解决。因为程序正常运行时应该满足 pageref(pipe) > pageref(fd) 的情景（即写端不关闭），而执行 unmap 操作时如果优先解除 fd 的映射，就可以保持这种大于关系恒成立，使得父进程开始运行时不满足写端关闭的条件，即使发生了时钟中断，也不会出现竞争问题导致的运行错误。
- 在程序执行时，如果在 dup 中先对 fd 进行映射，由 Thinking 6.2 中的过程模拟其实就可以知道，此时 pageref(fd) 先增加，由于程序执行时有 pageref(pipe) >= pageref(fd) 的情景，可能会在时钟中断后造成读端已满然后关闭（或者写端已经关闭）的虚假情形，出现与 close 类似的问题。

Thinking 6.5

```
/* /lib/env.c/load_icode_mapper */
while (i < sgsize) {
        size = MIN(BY2PG, sgsize - i);
        r = page_alloc(&p);
        if (r != 0) {
            return r;
        }
        page_insert(env->env_pgdir, p, va + i, PTE_R);
        bzero((void*)page2kva(p), size);
        i += size;
    }
```

使用 load_icode_mapper() 处理 bss 段的数据时：

因为 bss 在 ELF 中不占空间，因此在处理时，当加载到 bin_size ~ sgsize 之间的数据时就使用 bzero 给空位赋 0，不向数据段复制内容，这样就能完成对 bss 段数据的初始化（初始值为 0 ）.

- .bss（better save space）：未初始化的全局变量和静态变量
  - 被初始化为0的全局变量和静态变量也在这里
  - 并不占据实际空间，它只是一个占位符（4B），区分已初始化和未初始化的变量是为了节省空间，当程序运行时，会在内存中分配这些变量，并把初始值设为0.

Thinking 6.6

因为在 user/user.lds 文件中将 .text 段的地址统一约定为了 0x00400000，因此所有文件在链接时起始地址都相同，*.b 的 text 段偏移值都相同。

```
. = 0x00400000;

  _text = .;                /* Text and read-only data */
  .text : {
       *(.text)
       *(.fixup)
       *(.gnu.warning)
       }
```

Thinking 6.7

- 我们在 MOS 中使用的 shell 命令是外部命令，因为执行时需要额外地 fork 子进程（子 shell）去执行具体的命令，且具体命令的执行过程都被包装为了代码文件，比如 user/cat.c, user/ls.c 。
- 内部命令实际上是 shell 程序的一部分，其中包含的是一些比较简单的 linux 系统命令，这些命令由 shell 程序识别并在 shell 程序内部完成运行。我们知道 cd 命令需要改变当前目录，因此会对当前 shell 的环境做出改变，如果将其设计为外部命令的话，外部命令将只会改变子 shell 的 PWD 而无法将其值返回给父进程以更改父 shell 的 PWD，进而无法完成 cd 命令的功能。
  - 由此也可以知道，需要改变当前 shell 环境的命令都应当设计为内部命令，比如 exit。

Thinking 6.8

在 user/init.c 中，如下的代码将 0 和 1 安排为了标准输入和标准输出：

```
if ((r = opencons()) < 0)
    user_panic("opencons: %e", r);
if (r != 0)
    user_panic("first opencons used fd %d", r);
if ((r = dup(0, 1)) < 0)
    user_panic("dup: %d", r);      /* 在这里，可以看到 0 和 1 被映射为标准输入和标准输出，它将 0 映射到了 1 上，可以认为是针对标准输入输出的缓冲区建立一个管道。 */
```

Thinking 6.9

- 观察到 2 次 spawn，分别对应进程 [00001c03] 和 [00002404]：

```
$ ls.b | cat.b > motd

[00001c03] pipecreate

[00001c03] SPAWN: ls.b

serve_open 00001c03 ffff000 0x0

serve_open 00002404 ffff000 0x1

[00002404] SPAWN: cat.b

serve_open 00002404 ffff000 0x0
```

- 观察到 4 次进程销毁，分别对应进程 [00003406], [00002c05], [00002404] 和 [00001c03]：

```
serve_open 00003406 ffff000 0x0

serve_open 00003406 ffff000 0x0

[00003406] destroying 00003406

[00003406] free env 00003406

i am killed ...

[00002c05] destroying 00002c05

[00002c05] free env 00002c05

i am killed ...

[00002404] destroying 00002404

[00002404] free env 00002404

i am killed ...

[00001c03] destroying 00001c03

[00001c03] free env 00001c03

i am killed ...
```

实验体会

正如在 实验难点 部分屡次提到的，笔者在这单元很大程度上吃了 unix 大作业手写 shell 的老本，而且本单元需要仔细阅读的代码量也比较小（不禁回想起了 lab5 的时光……但不理解 lab5 部分的内容的话完成这一单元也是比较困难的），因此将作业实现下来难度并没有那么大，当然，综合性很强又难以下手的 spawn 函数还是给我好好上了一课，让笔者在不断翻看前几个单元的代码 / 指导书中度过了几个充实的夜晚，关于 ELF 文件的内容更是让我好好复习了一番理论课和 CSAPP 的相关章节，其附加作用是远大于实现代码本身的用途的。

同时，本单元另一个有趣的地方就是管道部分对管道竞争问题的模拟，手动模拟了几遍后给人一种 OO 第二单元多线程问题的既视感。离开了方便的锁机制，我们需要从比较底层的角度思考进程切换给程序造成的不安全影响，并且使用映射顺序管理等方法来维护程序的安全，这个过程还是令笔者收获颇丰——未来即使使用上了更为高级方便的语言，我想我也不会忘记考虑这些底层的安全问题。

实现了 shell 的过程，比起“完成 OS 的大厦”，更像是给 OS 缝好了一件漂亮的衣服，让我们简陋的实验能够出来“见见世面”，和其缔造者握握手——仿佛被蓝仙女的魔杖拂过的木偶匹诺曹，这个过程给人的成就感是非常强烈的。因此我也想为 unix 这门稍微擦边的课打个小小的广告，最后手动实现一个自己的（花里胡哨的）shell 的经历还是十分有意思。不知道未来的 OS 实验中能否有条件实现类似的内容？

最后，lab6 实验报告写到尾声的一刻，似乎也就标志着 OS 实验课即将落下帷幕了（当然，此刻笔者的挑战性任务还未完成，与 OS 的缘分尚能再续几天）。虽然中间遇到了不少困难~~（回想起上半学期被隔离时，上机上到一半隔离点网断了，直接送走本次实验，真的会谢……）~~，但它和本学期选修的 unix 课程依旧共同构成了笔者探索计算机世界另一角落的美妙旅程，有关内核、底层的一切在我看来都那样新鲜与有趣，操作系统世界是无比广阔的，因此我相信我与它们的缘分不会随着 OS 课程的结束而消散。将来在学习与工作生涯中，我也会继续学着从这些可以说是藏在计算机世界暗处的角度来思考问题（比如安全，比如性能），感谢老师和助教们在课程与实验中给我带来的许多帮助。

祝 OS 课越办越好！