**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？**

1 **switch** (fork()) {

2 **case** -1: */\* Handle error \*/*

3 **break**;

4

5 **case** 0: */\* Child - writes to pipe \*/*

6 close(fildes[0]); */\* Read end is unused \*/*

7 write(fildes[1], "Hello world**\n**", 12); */\* Write data on pipe \*/*

8 close(fildes[1]); */\* Child will see EOF \*/*

9 exit(EXIT\_SUCCESS);

10

11 **default**: */\* Parent - writes to pipe \*/*

12 close(fildes[1]); */\* Write end is unused \*/*

13 read(fildes[0], buf, 100); */\* Get data from pipe \*/*

14 printf("parent-process read:%s",buf); */\* Print the data \*/*

15 close(fildes[0]); */\* Finished with pipe \*/*

16 exit(EXIT\_SUCCESS);

17 }

18

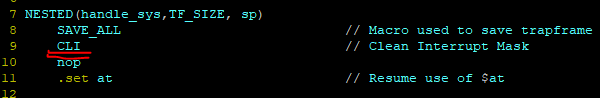
将代码修改为父进程操作读端，而子进程操作写端。

**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/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？**

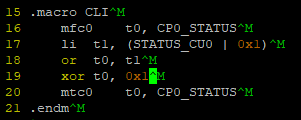
dup函数首先复制了fd描述符所在的页，接着才在for循环中逐页复制了fd所对应的4MB的数据区。这两步并不是原子操作，若fd所在页刚被复制之后还未复制数据区就发生了一个时钟中断，切换到其他进程运行。这时fd描述符所对应的文件数据复制尚未完成，对fd描述符对应文件的操作就会出错。

**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。**

系统调用一定是原子操作。系统调用是操作系统内核为各种用户进程提供服务的接口，从这个角度来说，系统调用不应该被来自用户进程的时钟中断打断。从代码实现角度来看，在我们进行系统调用时，首先会进入handle\_sys函数，在这个函数的开头，就使用CLI宏来关闭了中断位。



CLI宏定义在include/stackframe.h中，作用就是暂时关闭中断位。



**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 pipeclose 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。**

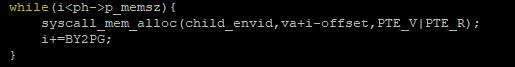
**• 我们只分析了close 时的情形，那么对于dup 中出现的情况又该如何解决？请模仿上述材料写写你的理解。**

可以解决上述场景的进程竞争问题。fork结束后，子进程先执行，时钟中断产生在了close(p[1])与read之间，也就是p[1]已经解除了对于pipe的映射，同时也解除了对于p[1]的映射。接着父进程在close(p[0])中产生时钟中断，此时已经解除了对p[0]的映射，但是还没有来得及解除p[0]对于pipe的映射。此时分析每个页的实际引用情况，pageref(p[0])=1，pageref(p[1])=1,pageref(pipe)=1,此时子进程执行read函数，就不会判断写端已经关闭而直接退出。

对于dup中出现的情况，只需要交换fd与fd数据区的复制顺序。因为我们需要保证fd文件描述符完成复制时它所对应的内容也完成复制，故只需要将复制数据区的操作放在复制fd描述符对映页的操作之前，就可以解决fd复制而对应文件内容未复制的问题。

**Thinking 6.5 bss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。请回答你设计的函数是如何实现上面这点的？**

在usr\_load\_elf函数中，将ELF的text端加载全部加载进内存后，对于未到p\_memsz大小的部分，使用syscalll\_mem\_alloc函数在子进程的对应地址处分配一页的大小，并设置标志位为可读可写，这个函数所调用的page\_alloc函数会主动调用bzero将这一页地址数据清0，既分配了空间，又实现了bss端初始值设置为0的操作。

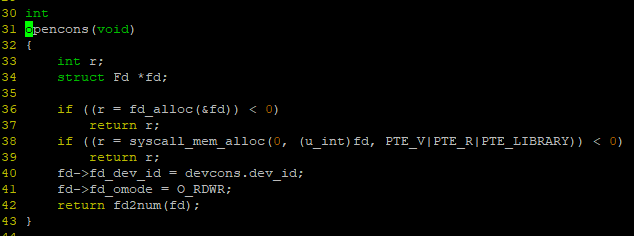


**Thinking 6.6 为什么我们的\*.b 的text 段偏移值都是一样的，为固定值？**

参考小操作系统的内存分布图，代码段为UTEXT以下的区域，故任意elf文件的偏移量都应该是一样的，这样方便设置指针时统一地将其设置为UTEXT处。

**Thinking 6.7 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。**

标准输入与标准输出的安排是通过user/init.c中调用函数opencons来实现的。此时是直接从内核运行代码，还没有完成对文件描述符的相关设置，故通过opencons函数获取的就是0号和1号文件描述符。



**二、实验难点**

这次实验分为两个部分。

第一部分主要是跟文件系统有关——管道是进程间通信的一种方式，而对管道读端、写端的操作与对文件系统的操作很相似。所不同的是读端与写端两个文件操作符映射到了同一片内存区域，并在fork的配合下，在父子进程间建立起进程间通信管道。管道实现起来较为简单，原理也比较好理解。这一部分的难点主要在于管道竞争的处理以及内存管理中page reference加一的处理。page\_reference的问题通过分析先前写过的代码，对物理页面引用次数的增加在page\_insert里面就进行了，写这次代码时检查了一遍内存管理与进程部分与分配页面有关的操作，删去了对引用次数的不正确增加。

对于管道竞争与同步问题，指导书给了我们一个非常重要的研究进程互斥与竞争的思路。在解决管道进程问题的过程中，我跟着指导书的讲解，对管道部分的代码进行逐行分析，找到了竞争可能发生的位置，并尝试着自己思考解决管道竞争方法的可行性，如调换一些代码的顺序，加入一些中间检测变量等操作。

第二部分，也就是shell部分可以说是对先前所有写过代码的一次大测试。shell部分最重要的一点就是spawn函数的填写。更新之后的指导书对这一部分的讲解很详细。spawn函数的作用是帮助我们调用文件系统中的可执行文件并执行。其实我们在lab3就已经进行过类似的操作——加载可执行文件到内存。区别是lab3中加载进程的可执行文件过程是在内核态下进行的，而这次需要要通过系统调用，从文件系统中读出可执行文件并加载到子进程的地址空间中，是在用户态下进行的。在填spawn函数时，我主要仿照lab3的写法，填写了usr\_load\_elf函数，将原本内核态下的页面分配与复制换成了用户态的系统调用以及usr\_bcopy，实现将文件中的文件拷贝到内存对应地址。

**三、体会**

完成所有实验后，编译运行自己的代码，看到那个命令行的$符号，心里还是很开心的。要通过课下测试所有的测试点，需要保证填写过的甚至没填过的函数都准确无误。感觉这次实验类似一个对我们搭起来的摇摇欲坠的OS大厦的一个总结。从内存布局、elf文件的组成、内存管理到进程管理、文件系统……shell看似内容很少，实际上是由先前的所有部分环环相扣而成。在完成操作系统实验的过程中，除了代码的填写，对于代码的调试也十分重要。在之前的实验中，我都是通过printf以及writef来定位bug，有过在学长的指导下在vimlinux.linux中找函数名设置断点调试的经历，也在lab4经历过由于pull版本错误发生的不知道在哪的bug而导致的重构。一路走来，除了对操作系统理解得更加深刻，完成综合代码并测试的工程能力得到了锻炼，一周一次课下一次课上的课程设计也锻炼了我们的抗压能力。

**四、指导书建议**

spawn为何要使用fork函数来创建子进程，以及system\_env\_alloc函数在这里的可行性，希望指导书能予以说明。