Lab6实验报告

19376273 陈厚伦

思考题

Thinking6.1

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?

除了将`case 0:`与`default:`分支的内容交换一下，还要注意父进程开始要先调度子进程完成写，自己才能读。

```
switch (fork()) {
    case -1:
        break;
    case 0: /* 子进程 */
        close(fildes[0]);
        write(fildes[1], "Hello world\n", 12);
        close(fildes[1]);
        exit(EXIT_SUCCESS);
    default: /* 父进程 */
        syscall_yield(); /* 调度 让子进程先写 */
        close(fildes[1]);
        read(fildes[0], buf, 100);
        printf("child-process read:%s", buf);
        close(fildes[0]);
}
```

Thinking6.2

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`的作用是把一个文件描述符的内容映射到另一个文件描述符，假如映射之前：

p[0]	p[1]	pipe
1	1	2

此时如果要映射读端，会

①先将`p[0]`的引用次数+1

②将`pipe`的引用次数+1

如果①②之间发生了时钟中断，次数`p[0]==pipe==2`，切换后另一个进程通过`_pipeisclosed()`判断时，根据`pageref(p[0]) = pageref(pipe) = 2`，会认为写端关闭，从而出现错误。

Thinking6.3

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。

在我们的MOS中，调用系统调用函数**陷入内核态时会屏蔽中断，系统调用结束后才会解除中断屏蔽。**执行系统调用的过程中，不会因为时钟中断而打断，所以我们的MOS中系统调用是原子的。

```
// set the interruption bit
.macro STI
        mfc0    t0,     CP0_STATUS
        li      t1, (STATUS_CU0 | 0x1)
        or      t0, t1
        mtc0    t0, CP0_STATUS

.endm
// clear the interruption bit
.macro CLI
        mfc0    t0, CP0_STATUS
        li      t1, (STATUS_CU0 | 0x1)
        or      t0, t1
        xor     t0, 0x1
        mtc0    t0, CP0_STATUS
.endm
```

`Linux`中并不是所有系统调用都是原子操作，例如write系统调用就可以被打断。单核情况下屏蔽中断位可以保证原子性，但是多核情况下不适用，屏蔽了一个CPU的中断位，另一个CPU依然可以执行内存读写操作，**无法保证一段代码是单独且完整执行。**

Thinking6.4

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，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。

可以解决这个问题，`pageref(fd) < pageref(pipe)`，`unmap`的时候先解除`fd`的`ref`，仍然可以保证`pageref(fd) < pageref(pipe)`，从而避免了大的先减1中间过程可能出现两者相等的情况。

`dup`同理，要先将`pipe`的`ref`+1，再将`fd`的`ref`+1，从而保证`pageref(pipe) > pageref(fd)`。

Thinking6.5

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?

处理ELF加载的是lab3的`load_icode_mapper`，当`binsize < sgsize`的时候，`sgsize-binsize`的部分将会用0补齐。

Thinking6.6

为什么我们的 *.b 的 text 段偏移值都是一样的，为固定值?

`user`的`link script`文件对`.text`段的地址进行了规定。

Thinking6.7

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。

```
if ((rightpipe = fork()) == 0) {
    // 子进程 读端 从管道标准输入读
    dup(p[0],0);
    close(p[0]);
    close(p[1]);
    goto again;
}
else {
    // 父进程 写端 标准输出写到管道
    dup(p[1],1);
    close(p[1]);
    close(p[0]);
    goto runit;
}
break;
```

实验难点图示

> 管道Pipe

管道的机制大概可以如图所示，管道是一种存在于内存中的文件，通过文件系统进行管理，是进程通信的一种方式。

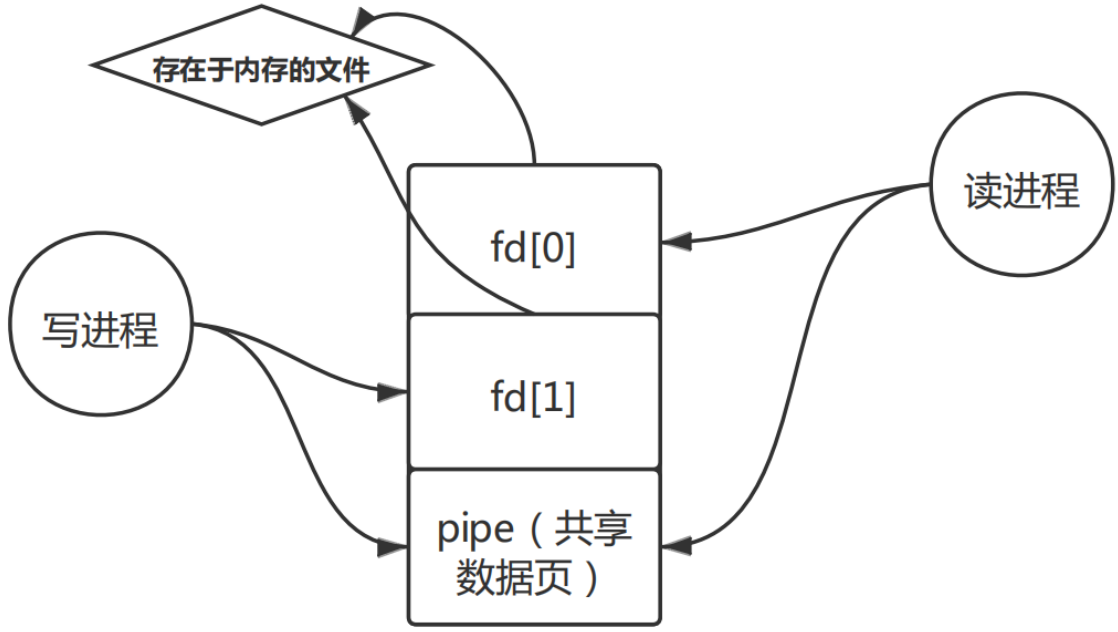
读进程和写进程每次都会打开管道的两个端，使用的时候要先关闭不用的那一端，读端和写端其实被映射到了同一片内存的区域，两个进程的页面的映射的相关关系通过以下的图加以表示。

管道的管理使用`Pipe`结构体管理，包括当前的读位置，当前的写位置，共享缓冲区，两个进程共享缓冲区时，修改多个映射的时候不是原子的操作，进程的切换导致出现错误，所以要注意对映射顺序的安排，这个可以通过如下原则加以考虑：

`pageref(bigger) > pageref(smaller)`

即时刻保持pageref高的物理页要始终高于pageref低的物理页。

读取pageref的时候因为进程的切换也要注意更新。

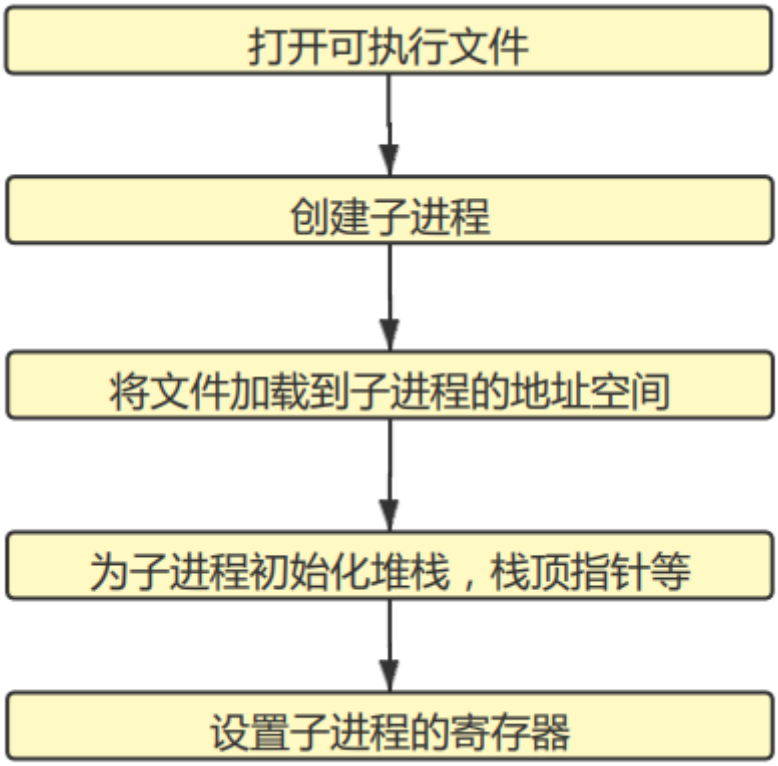


C语言贴近底层的语言没有现成的锁的机制，这个lab中许多的地方是通过调整顺序实现安全，比较繁琐。

> Shell

Shell本质是一个字符串解析工具，根据解析到的内容，执行特定的功能函数。`runrmd()`函数完成了这个功能，即根据读到的字符串内容决定接下来的操作，例如是打开一个读文件，还是打开一个写文件，还是创建一个管道。

其中最难的是`spawn`函数，这个函数的执行步骤非常明确，难点在于它贯穿了lab1-lab5的许多内容，非常考验对前面知识的理解程度和熟练程度。`spawn`函数的功能是获取一个可执行文件，然后创建一个进程来执行它，用到了加载ELF文件的内容，和进程的各种设置，`spawn`的流程大体如下：



其中将文件加载的子进程的地址空间和lab3的任务是类似的，在lab4中我们是将可执行文件加载的系统的初始进程中，而这里是根据shell中输入的命令加载一个可执行文件并为其创建一个可以执行它的进程。

体会与感想

`lab6` 管道与shell部分理论课没有涉及太多，所以入手起来感觉非常困难，读了许久才慢慢找到感觉。整体上花费的时间略低于lab5。总体来说，过程不是很复杂，`pipe` 和 `shell` 的内容指导书给出了很明确的步骤，但是牵涉到前几个lab的内容太多，加上对之前的内容有所遗忘，所以在调用各种函数的时候很困难。

残留难点

对Shell的许多函数细节还没有进一步仔细看，对内部实现不了解。

spawn函数理解还不够充分。

ELF文件还需要复习。