Lab6 实验报告

游子诺

2019 年 7 月 1 日

1 实验思考题

Thinking 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？

如果向颠倒父子的读写身份，需要修改代码内容主要是管道口和对于管道的行为，具体来说，父进程在读取前应该关闭管道口的写端而保留管道口的读端（关闭 1 号），而子进程在写前应该关闭管道口的读端而保留管道口的写端（关闭 0 号）。

在具体实现时，可采用将 switch-case 语句中的条件发生变化，使父进程读而子进程写。

Thinking 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？

在一个进程在检测另一个进程是否关闭接口时，采用比对 p[i] 和 pipe 的页面应用次数的方式，但在多进程状况下由于关闭和打开过程可能被打断，因此会出现错误的判断。

dup 函数是用于浅拷贝“抽象文件”用的，当被拷贝的是一个管道时，涉及到 pipe 和 p[i] 的两个的拷贝，在原代码中，若先行拷贝 p[i]，则可能在 pipe 拷贝前被时间片所打断，造成 pipe 没有及时被增加，这时候如果有其他进程进行 close 判断，则可能出现 p[i] 引用和 pipe 引用相等的情况，导致错误的关闭。

解决方法：避免此类问题的一个通用思路是，在不能 close 状态时，引用量大的在增加引用时优先增加，引用量小的在减少引用时优先减少。

Thinking 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。

lib/syscall.S 用户进程调用 syscall 进入内核态后，由内核态识别到是系统调用中断后来到的系统调用通用入口，**是所有系统调用必须经过的一段汇编代码**，代码中有如下内容：

```
NESTED(handle_sys,TF_SIZE, sp)
SAVE_ALL                                // Macro used to save trapframe
CLI                                      // Clean Interrupt Mask nop
.set at                                  // Resume use of $at
......
```

由此段内容可知，在进入内核并具体处理系统调用时，均会统一地用 CLI 宏定义关闭时钟，进而使得系统的操作原子化。

Thinking 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。

根据分析可以得出，通过更换映射与解除映射的顺序，可以避免页面竞争的问题：

- 以 fd[0] 和 pipe 两者为例作为讨论对象，在正常情况下两者的页面引用满足

  $$pageref(fd[0]) <= pageref(pipe)$$

  并且当且仅当“=”时表示管道的另一端已被关闭，原本代码中在 unmap 时先操作了 pipe 导致**出现了与 fd[0] 引用相等的短暂时刻**，而这个短暂时刻可能因为时间片轮转而被“放大”，影响其他进程。

- 如果在 unmap 时先操作 fd[0]，那么就避免了这个短暂时刻的发生，这样一来此处的竞争问题就消灭了。进一步提炼，无论是 map 还是 unmap，**只要避免在变换的中间状态产生不稳定的状态转移条件即可**。（在此题中转移条件就是“=”）。

- 对于 dup 函数，根据上段所提炼的，就是避免产生**不稳定的等于状态**，因此应该先映射 pipe，再映射 fd。

Thinking 5

请解释 spawn 函数中注释标记为 Share memory 一段的作用，并说明为什么该段代码是正确的。你可以尝试对该段代码进行改动以探究其对运行结果的影响.

以 PTE_LIBRARY 标记的页面用以表示共享的页面，spawn 用于创建一个新的进程，而新进程起步时并不是什么都不需要，往往需要之前运行的一些环境，LIBRARY 就提供了这样的功能。在 lab6 中，library 主要用于为新进程提供**已经被打开文件的控制信息**。

在尝试时，以 cat.c 回显进程为例，当在 shell 中不带参数的输入 cat.b 后产生了回显功能，但 cat.c 代码中若没有参数，其**直接使用的 read 和 write 对 fd[0] 和 fd[1] 进行读写**，可见创建此进程时用 library 授权了自 shell 建立时就初始化好的标准输入输出文件的使用权限。若注释掉此部分，cat.b 无法执行正常的标准输入输出回显功能。

Thinking 6

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？

单个文件最大 4mb，bss 段可能远超 4mb 大小，在实际加载时，我不会为 bss 段创建任何一张新页，只会保证在其他创建的新页中若有 bss 段内容则一定要为 0。bss 段的内容在使用时会自然产生 pgfault，由操作系统分配新页，由 mm/pmap.c 中的 page_alloc 可知分配新页时会自动赋为 0，恰好满足要求。

Thinking 7

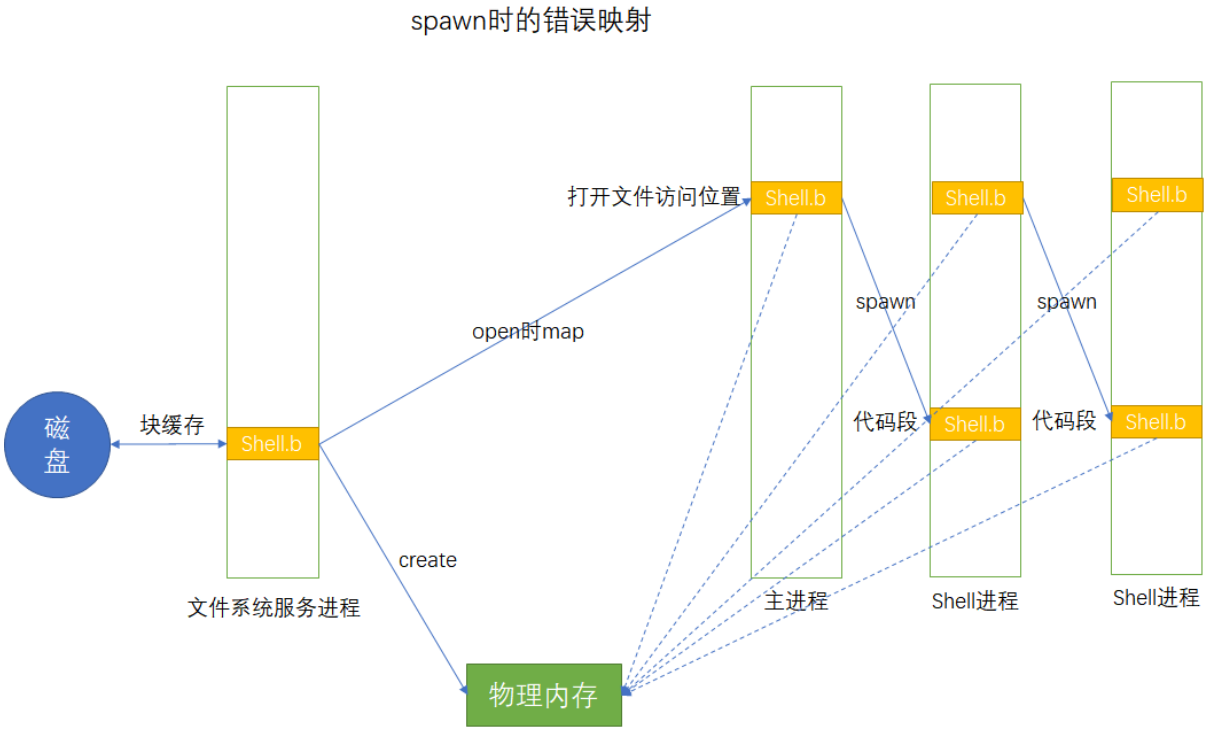
Thinking 6.6 为什么我们的 *.b 的 text 段偏移值都是一样的，为固定值？

user 文件夹下有.lds 链接脚本文件，其中规定了编译器该如何链接目标文件，text 段的偏移值就是规定的内容。

2 实验难点图示

2.1 “错误的无脑 map”

在 lab6 的 shell 编程时，最有挑战性的部分当属 spawn.c 文件的填写了，此部分指导书所给的内容十分有限，因此需要在较好地掌握了 shell 系统运行原理的基础上编写。其中我踩过的一个大坑就是在 spawn 时将运行代码从文件导入到新进程代码段，以下图进行总结：



- Bug 的直接现象：在 shell 中创建子 shell 后，子 shell 再运行一些常规的进程会出现”0x0 TOO LOW 的问题”
- Bug 的直接原因：子 shell 创建常规进程时会 fork，但是其并没有成功地将子 shell 进程的进程块中的用户态 pgfault_handler 入口赋予有效的值，导致 fork 后的 COW 缺页中断时内核态向用户态处理函数跳转时发生错误。
- Bug 的间接原因：在 set_pgfault_handler 函数中，程序识别到 ___pgfault_handler 非零，进而跳过了对子 shell 进程块中的入口设置。
- Bug 的根本原因：**子 shell 在第一次使用 fork 时 pgfault_handler 就有非零值的原因在于“在非可重入代码上滥用页面映射“**，根据上图我们可以看到，shell.b 文件自文件服务从磁盘调出后，由文件服务进程创建了物理页用于缓存文件中的内容，而后这个物理页被多次映射到”主进程“,” shell 进程“,” 子 shell 进程“三个进程的虚拟空间中。而**两个 shell 所跑的代码都在同一个物理页上**，全局变量 ___pgfault_handler 就是代码段中的一部分内容，因此子 shell 中的全局变量早已

被母 shell 使用过了，后续使用等同于当成共享内容用，**对于非可重入代码而言当然是会出问题的。**

- Bug 解决方法：**深拷贝**，运用各类手段真正地把代码复制一份。

2.2 补充：文件中的 bss 段

通过之前 lab3 的 elf_loader 可知，bss 段并没有存储在文件中，而是以 bin_size 和 sgsize 之间的偏差体现。目前实现的操作系统中，支持的文件最大大小为 4MB，但这其实是对 text 和 data 段的限制，bss 段所定义的大小可能超过 4MB。因此，在 spawn.c 中的具体处理时仅考虑加载 text 和 data 段，bss 段的使用随着 pageout 插入空白新页自动实现（不过需要注意当 text+data 段并没有对齐 BY2PG 时，最后一页剩余的未对齐部分需要及时清零，否则不满足 bss 的要求）。

3 体会与感想

3.1 操作系统的 map 与 oo 中的 clone

在 lab6 中填写 spawn 函数时遇到的 bug 令我想起了 oo 课上有关 clone 函数的问题。clone 函数在自定义类中实际是重写的，因为系统默认情况下的 clone 是“浅拷贝”，对于自定义类中的许多对象只会拷贝其引用而不拷贝对象中的数据；在 os 中，这种浅拷贝其实就是页式内存管理中的映射-map：**浅拷贝和映射都容易导致多个对象或多个进程对同一对象作出不期望的影响。**

在具体实践时，我们可以采取以下解决方案：

- 深拷贝：在 oo 中就是重写 clone 方法以明确地生成对象的副本，在 os 中就是采用 read 方法去访问文件中的信息（虽然文件信息确实可用 fd2data 去找到），read 方法参数需要提供目的地字符串地址，并在 read 时把内容一字节一字节地搬过去，这样的行为其实就是在做深拷贝。
- 不可变对象：oo 中即为不提供修改方法；os 中即为可重入式代码。

3.2 抽象文件的概念

在 lab5 中我们实现了文件系统，那时我对代码文件的组织有一个问题：”为什么 file.c 和 fd.c 要分开定义，这两者有什么区别？“。这个问题在 lab6 中得到了很好的解答，在 lab6 中共有磁盘文件、管道和标准输入输出三种文件，但用户却都使用 fd 实现对应的功能，**文件与用户交互的直接媒介就是 fd.c 中所定义的通用函数**，这些通用函数会具体调用 file.c、pipe.c、console.c 中的实现来完成功能。

这样的文件系统就做到了对用户的透明，用户无需关心内部实现，在新增文件系统时方便许多。

4 指导书残留难点与反馈

4.1 指导书错误反馈

- 6.3.1 节：spawn 函数实现时，显然不可以用 fork() 方法去创建新进程，一方面这样创建出来的新进程数据肯定是有问题的；另一方面，fork() 函数结束时会立即运行子进程，这种行为肯定与 spawn 函数目的相违背。

4.2 指导书难点

- 管道进程问题的表现方式：在指导书中用了几段话模拟管道竞争的过程，我在这部分阅读起来比较麻烦，建议通过**连环画**的形式展现这个不稳定状态下的竞争问题。

- 由于直接映射文件内容到代码段造成的问题：作业中的代码不是可重入码，因此不能用映射而是要实际拷贝的方式把文件内容加载到运行段上，建议**加深对文件从磁盘-文件服务进程-open 文件进程物理数据存在性的图示分析**。