Lab6 实验报告

19373135 田旗舰

一、实验思考题

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？

交换 case0 和 default 的内容。

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 函数将一个文件描述符对应的内容映射到另一个文件描述符，由于对文件描述符的引用次数和 pipe 的引用次数修改存在先后顺序，因此如果在其之间发生时钟中断，跳转到另一进程调用_pipeisclosed 函数，则可能会出现判断错误。

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

在实验操作系统中，系统调用陷入内核态时会屏蔽中断，因此系统调用一定是原子操作，不会被打断。而在 Unix 中系统调用不一定是原子操作，如 write 系统调用可以被打断。

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，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，由于 pipe 的引用次数总比 fd 要高，所以先解除 fd 的映射，再

解除 pipe 的映射，使得 fd 的引用次数减 1 先于 pipe 的引用次数减 1，所以不会出现 fd 与 pipe 引用次数相同的误判。

会出现，所以，同理，由于 pipe 的引用次数总比 fd 要高，应当先映射 pipe，使 pipe 的引用次数加 1 先于 fd。

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？

MemSize 大于 FileSize 的部分即为 bss 段，分配页面时将其初始化为 0，使得 bss 的数据初值为 0。

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？

链接器设置为固定值。

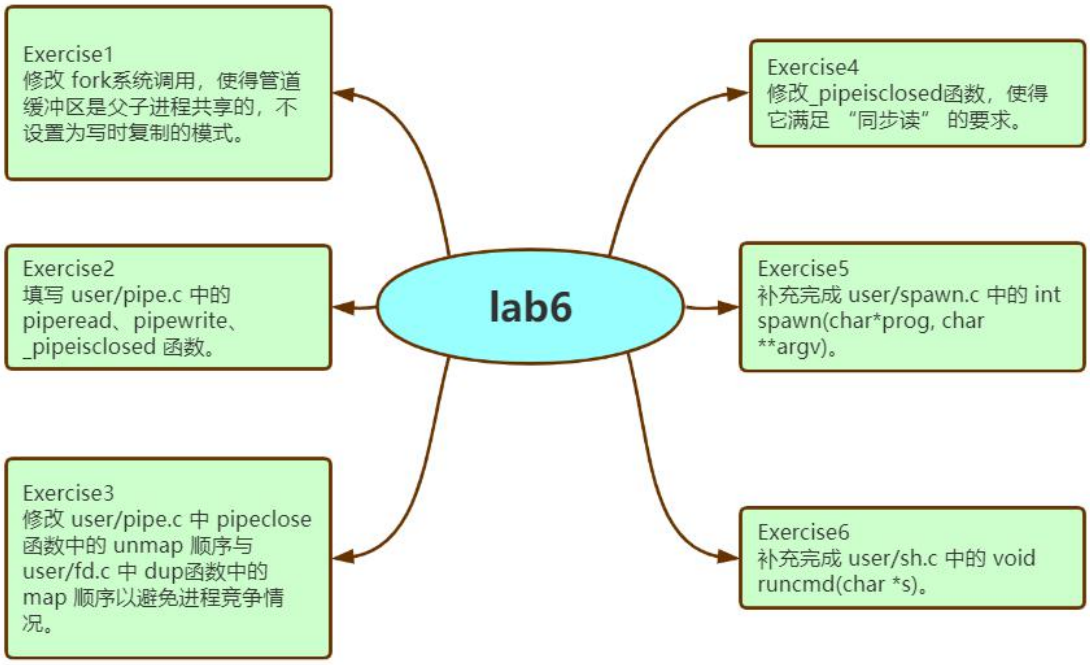
Thinking 6.7

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。

在创建 pipe 时安排的，在 pipe.c 的 pipe 函数中

```
int
pipe(int pfd[2])
{
	int r, va;
	struct Fd *fd0, *fd1;

	// allocate the file descriptor table entries
	if ((r = fd_alloc(&fd0)) < 0
	||  (r = syscall_mem_alloc(0, (u_int)fd0, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
		goto err;

	if ((r = fd_alloc(&fd1)) < 0
	||  (r = syscall_mem_alloc(0, (u_int)fd1, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
		goto err1;

	// allocate the pipe structure as first data page in both
	va = fd2data(fd0);
	if ((r = syscall_mem_alloc(0, va, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
		goto err2;
	if ((r = syscall_mem_map(0, va, 0, fd2data(fd1), PTE_V|PTE_R|PTE_LIBRARY)) < 0)
		goto err3;

	// set up fd structures
	fd0->fd_dev_id = devpipe.dev_id;
	fd0->fd_omode = O_RDONLY;

	fd1->fd_dev_id = devpipe.dev_id;
	fd1->fd_omode = O_WRONLY;
```

二、实验难点图示



本次实验主要分为两个部分：管道和 shell。其中前 4 个 exercise 与管道相关，后 2 个 exercise 与 shell 相关，相比较而言管道部分较为简单，只需要注意进程切换时产生的错误判断管道关闭以及同步读的的问题，而 shell 部分难度较高，不仅阅读的代码量较大，需要填写的部分提示也较少，需要建立在理解的基础上进行。

具体而言，本次实验的难度如下：

1. 管道中进程竞争问题的处理，包括调整 unmap 的顺序以及实现 _pipeisclosed 中同步的要求。

2. shell 部分读取并加载 elf 文件，由于时间久远有些遗忘，需要重新熟悉 elf 文件格式，而加载 elf 文件的部分与 load_icode_mapper 函数十分相似，注意处理 MemSize 大于 FileSize 的那一部分。

三、体会与感想

本次实验难度尚可，大致与 lab5 相同，总共花费了大概 2 整天的时间，可能是熟能生巧，debug 的时间有所减少。

随着 lab6 的结束，OS 正式的实验部分也告一段落。这 6 次实验帮助我建立

起了整个操作系统的概念，了解了其中部分重要机制的实现，也算是有了较大的收获。但回顾整个小操作系统，还有相当一部分的代码对我来说较为陌生。提供相应的硬件接口，从 0 开始编写一个操作系统对我来说还不太现实，但或许也正是操作系统的复杂性，才有了操作系统这门课程，也才需要我们更加深入地去学习操作系统。“路漫漫其修远兮,吾将上下而求索”，接下来还有挑战性任务，希望能够顺利完成。

四、指导书反馈

似乎指导书中关于实验正确结果的图片与实际有些不一致的地方，建议更新。

五、残留难点

虽然顺利通过了测评，但是会出现莫名奇妙的 bug，比如刚刚退出仿真后再立即运行，有时 shell 无法启动，但也不是每次都能复现，不知是何种原因所致。