Lab 实验报告 I

Lab: Xv6 and Unix utilities

丁睿

dromniscience@gmail.com

更新：2020 年 10 月 27 日

目录

1 任务完成清单

Subtask	Done?	Time
`sleep.c`	Y	10min
`pingpong.c`	Y	20min
`primes.c`	Y	2h
`find.c`	Y	1h
`xargs.c`	Y	1.5h

Grade: 100/100

2 详细情况 & 困难和收获

我的作业已开放在这个网页上。

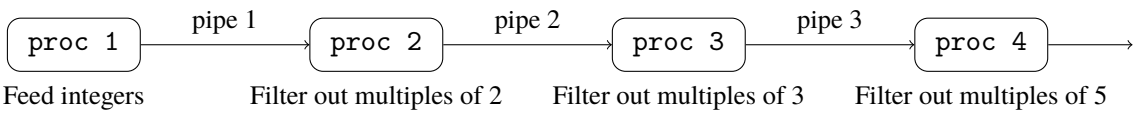
由于 Sleep 和 Pingpong 过于基础，我们跳过它们。其中涉及到的内容会被后面的几个练习覆盖。我们的重点是 Primes。Find 的核心是内核数据结构`struct dirent`的使用和一个简单的 DFS；Xargs 的核心则是书写一个鲁棒的将标准输入处理成`argv`参数的函数过程。

在实现中，我检查了所有关键的系统调用并在出错时设置用户友好的提醒。当你阅读代码时可以不必在意这些检查。

2.1 Primes

该 lab 作业最核心的问题包括如下几个方面。

- 每一个进程 (除了第一个进程) 都要模拟一个筛子，它必须存储自己使用的素数;
- 进程仅能从管道获取数据或者向管道写入数据 (不包括命令行的打印)。它们需要被动态地创建，而且彼此间要组成链式的父子关系，如下。其中箭头 $a \to b$ 表示 a 是 b 的父亲。



- 每个父进程必须正确地回收子进程,因此需要通过在`fork()`时正确地关闭管道,实现`proc 1`输入完所有的整数后，一旦关闭管道，后续的子进程就从阻塞的`read()`中返会 0，然后等待回收自己的子进程的层层传递的效果。
- 不能使用`exec()`函数。这要求每个子进程必须有类似的逻辑流，并且要在合适的地方被`fork()` 出来。

基于以上分析，我们提出如下伪码框加。接下来将解释该框架如何满足了以上需求。

```
int p[2] = {-1, -1}, prime;
int mark = 0;

int
main()
{
    close STDIN_FILENO, STDERR_FILENO to save space;
    pipe(p);

    /* Parent */
    if(fork() > 0){
        close p[0]; // Close unneeded fd
        /* We do not feed sieve number itself to the sieve */
        Feed 3 to 35 to pipe;

        close(p[1]); // It induces chain reaction
        wait for the child to end;
        exit(0);
    }

    /* Child */
    close p[1]; // A prerequisite to start chain reaction
    prime = 2; // sieve number of the current process
    print 2 to STDOUT_FILENO;
    while(read(p[0], num) != 0){
        if(num % prime){
            if(!created)
                new_sieve(num);
            else /* Already has a child sieve */
                write(p[1], num);
        }
    }

    close(p[0]);
    if(p[1] > 0){ // It indicates whether a child exists
        close(p[1]); // Transmit chain reaction
        wait for the child to end;
    }
    exit(0);
}
```

而new_sieve的伪码如下。

```
void
new_sieve(base)
{
    print base to console;
    int tmp[2];

    pipe(tmp);

    /* Parent */
    if(fork() > 0){
        close tmp[0]; // Close unneeded fd
        p[1] = tmp[1]; // Use new pipe to write
        mark = 1; // Already fork a child sieve
        return;
    }

    /* Child */
    close p[0]; // Close unneeded fd
    close tmp[1]; // A prerequisite to start chain reaction
    p[0] = tmp[0];
    p[1] = -1; // Must be -1. We write this to emphasize it
    mark = 0; // Must be 0. We write this to emphasize it
    prime = base; // sieve number
    return;
}
```

在这段程序中我们区别对待了`proc 1`和其后所有的子进程，因为它的逻辑不太一样。`proc 1`只是简单地提供子进程运行的动力——依次填入数字，最后它通过关闭管道来中断动力。于是子进程将依次识别 EOF 来跳出循环去关闭自己到子进程的管道[1]，把这个连锁反应传递下去。然后再从链的最深处每个节点被父节点回收，这个过程一直到达链头，所有进程都干净而完美地结束。

因此，必须在子进程中关闭每一个`fork()`时的管道输入，以保证每个管道输入的引用计数是 1。

为了使子进程有类似的结构，我们将子进程必要的数据，例如`p[0]`(表示管道输入的 fd)、`prime`(表示自己筛的是哪个素数的倍数) 都在最开始设置好。因此`main()`和`next_sieve()`里 Child 最开始的一段是一样的。一旦设置好后，子进程就是简单地从管道里取数据，然后把没有筛掉的扔给自己的子进程 (如果还没有就调用`next_sieve()`。**这很像递归，但请注意每个进程只调用这个过程至多一次！**)。因此只要子进程从该函数返回到原来循环中的位置就能正确展开自己的工作。

这就完成了我们的分析。其中最精巧的地方在于每个子进程的相似的程序逻辑设计。我想这也是该 lab 被标记为 moderate/hard 的原因。

[1]如果有的话，这个靠`p[1]`是不是-1 可以标识出来。

2.2 Find

注意到目录被内核组织为具有如下结构的数组。

```
/* kernel/fs.h */
// Directory is a file containing a sequence of dirent
   structures.
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

因此我们编写一个函数`find_dir()`递归地查找当前目录下的同名文件。对其中的每个文件，利用`fstat()`函数检查文件的类型，如果是目录就递归调用自己，否则匹配文件名。DFS 保证了任何时刻文件描述符池的打开文件数尽可能的小——正比于路径上的节点数量。

注意`fstat()`的参数是一个 fd，这意味着必须先打开文件。那么我们在何时关上呢？正确答案是普通文件应该立刻关上，而目录应该在递归调用`find_dir()`之后关上。

2.3 Xargs

程序的逻辑很简单。不管是`xargs`前面是个管道还是 console，此刻都被重定向到了标准输入 1。我们首先复制`argv[1:argc-1]`。然后拼接上每行的参数即可。

重点就是从标准输入解析命令行参数。我们容易写出以下正则表达式 (Regular Expression)，L 代表一行：

$$\Sigma = \text{ASCII}$$
$$L = SP^* \circ (\Sigma/\{LF, SP, 0\})^+ \circ (SP^+ \circ (\Sigma/\{LF, SP, 0\})^+)^* \circ SP^* \circ (LF \cup 0) \ \cup \ SP^* \circ LF$$

根据该表达式，容易构造出相应的程序。每次我们都选择读取一个空行或者字母组成的数据段。如果是字母段，那么因为其后要么是空格 (SP)，要么是换行符 (LF) 或者 0，我们总能放心地将末尾追加一个零。最后，记得将构造出来的参数最后再加一个 NULL 指针表示参数的结束。

以下列出了实现代码的关键部分。其中`char **p`是指向当前的标准输入内容的指针 (我一次性把所有的标准输入读了下来放在`char line[MAXN]`里，而不是一行一行地读取)，它总是指向一个行的开始。因为我希望修改行指针的值，所以不得不传入了一个指针的指针。`char **_argv`保存新的参数字符串，其中最低的`argc - 1`指针是固定的，就是`xargs`后面紧接的那些。解析出来的新参数字符串的指针紧跟着放在后面，最后固定添加一个 0 表示结束。这里，所有新的参数字符串仍然放在`char line[MAXN]`原来的位置上，只是把每个参数的后一位 (根据 L，它总是 LF、SP、0 中的一个) 都改成 0 表示字符串的结束。

因此这个方法在空间利用上是比较高效的，同时时间复杂度也是最好的 (只顺序扫描标准输入一遍)。

```
/* kernel/xargs.c */
int
parse_line(char **p, char **myptr, char **_argv){
    while(1){
        switch(**p){
            case ' ':
                **p = 0;
                (*p)++;
                while(**p == ' ') (*p)++;
                break;
            case '\n':
                **p = 0;
                (*p)++;
                *myptr = 0;
                if(*(*p + 1) == 0) return E_INPUT;
                return E_CMD;
            case 0:
                *myptr = 0;
                return E_INPUT;
            default:
                *myptr = *p;
                myptr++;
                if(myptr - _argv >= MAXARG){
                    fprintf(2, "xargs: too many arguments!\n");
                    exit(1);
                }
                while(**p != 0 && **p != ' ' && **p != '\n')
                    (*p)++;
        }
    }
}
```

3 对课程和 lab 的建议

1. 提供一个像 ICS 课程csapp.c和csapp.h的文件，其中包含 Stevens 格式的错误处理包装函数 (wrapper function)。它检查系统调用的返回值，并在出错 (通常是一个负值) 时打印到标准错误设备然后退出。
   这一点上甚至 xv6 的源码做得都不够好。
2. 如果可以，我希望能够减少一些 slides 上布置的课后题，虽然它们确实也很有意义。我个人感觉这门课的作业量远超 ICS，尤其是书面作业。如果可以，我希望更多地增加源码阅读和 lab 报告的比重。这两项作业我认为在意义上比课后题重要得多，而且更具有挑战性，同时它们的质量也更容易量化。我认为一个没有仔细读源码的同学的报告会有明显的空洞

与不足之处，但是相比之下课后题的考量就比较模糊，而且很多时候不得不拼凑各种网站博客的内容。在写 lab 或者源码阅读报告时，我们通常也需要了解大量系统结构的细节知识，因此同样很训练查找资料以及学习理解的能力。

感谢您的阅读与考虑！

4 参考资料

1. *xv6: a simple, Unix-like teaching operating system*
   Russ Cox, Frans Kaashoek, Robert Morris *August 31, 2020*
2. 现代操作系统 [M]
   A.S.Tanenbaum, H. Bos 著, 陈向群 马洪兵 译, 北京: 机械工业出版社, 2011: 47-95
3. Git 教程 by 廖雪峰