同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 1

2151767 薛树建

日期：2023 年 7 月 20 日

目录

Lab1: Xv6 and Unix utilities

1. Boot xv6(easy)

1) 实验目的

本实验目的在于使用户熟悉 xv6 的相关操作与运行。

2) 实验步骤

在进行本实验之前，确定你已经配置好了相关环境以及工具，同时使用该指令
git clone git://g.csail.mit.edu/xv6-labs-2021
克隆了本实验的源码。

本实验步骤如下：

a) 首先进入虚拟机，输入 cd 指令将目录设置为本实验源码目录下

b) 输入 git checkout util 指令来切换到本实验 lab1 所使用的分支

c) 接着输入 make qemu 来构建并运行 xv6，本实验 xv6 操作系统的运行依赖于 qeum 来实现.

d) 输入 ls 即可查看 xv6 的文件系统，结果如下：

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2226
xargstest.sh   2 3 93
cat            2 4 23896
echo           2 5 22728
forktest       2 6 13080
grep           2 7 27248
init           2 8 23824
kill           2 9 22696
ln             2 10 22648
ls             2 11 26128
mkdir          2 12 22800
rm             2 13 22792
sh             2 14 41664
stressfs       2 15 23800
usertests      2 16 156008
grind          2 17 37968
wc             2 18 25040
sleep          2 19 22512
pingpong       2 20 24088
primes         2 21 24552
find           2 22 26448
xargs          2 23 24280
console        3 24 0
```

e) 实验结束

3) 实验中遇到的问题和解决方法

由于本实验的部分较为简单，在实际过程中并未遇到过问题。

4) 实验心得

本实验主要都是一些关于 xv6 的基本操作，较为简单，但是与后面实验的关系联系也是较为紧密。

2. sleep (easy)

1) 实验目的

本实验目的是为 xv6 系统实现一个睡眠的程序。该程序需要用户指定相应的睡眠时间 tick（即两次时钟中断的间隔时间），同时执行完毕之后退出。同时我们需要将自己实现的源码放置在 user/sleep.c 文件中。

2) 实验步骤

a) 首先第一步是阅读 xv6 参考书的第一章节内容，同时查看代码中的一些其他程序来了解程序是如何获取命令行参数的

b) 其次我们需要参阅实现睡眠的系统调用的 xv6 内核代码 kernel/sysproc.c（查找 sys_sleep 函数），user/user.h 以获取可从用户程序调用的睡眠的 c 定义，user/usys.S 以获取从用户代码跳转到内核以进行睡眠的汇编程序代码。
sys_sleep 函数在内核中的实现如下：

```
uint64
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(myproc()->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

usys.s 中关于 sleep 的部分如下

```
.global sleep
sleep:
 li a7, SYS_sleep
 ecall
 ret
```

那么在实现 sleep 的用户程序的大致思路便是如何进行 sleep 的系统调用了。

c) 我们首先在 Makefile 中添加对 sleep 程序，'$U/_sleep\ '，从而使得 make qemu 之后，我们可以在命令行中去运行。

d) 在 sleep 函数中，mian 函数的参数 argc 与 argv 分别为参数的个数与参数列表，其中 argv 的第一个参数为函数名称，我们在编写 sleep 函数时应该首先去判断参数个数是否合法，接着在调用 uesr.h 中声明的 sleep 函数进行系统调用，最后运行 exit(0)来释放进程。

函数如下：

```
int main(int argc, char* argv[]){
    //只接受一个参数
    if(argc != 2){
        fprintf(2, "sleep: wrong numbers of arguments.");
    }
    sleep(atoi(argv[1]));
    exit(0);//释放内存
}
```

3) 实验中遇到的问题和解决方法

本实验为第一个正式来写代码的实验，难度不大，在实验中遇到的问题包括为如何将字符数组的参数转化为数字，如何将错误信息打印在控制台上。

对于第一个问题，在 xv6 官方网站的提示中点名了可以使用相关的 atoi 函数来进行数字的转换；对于第二个问题，在官方的实验参考书中介绍了文件描述符等内容，0 为标准输入设备，1 为标准输出设备，2 为错误输出设备，根据这些内容，可以单纯使用 fprintf 函数来将错误的内容（错误内容主要为函数的参数数量错误）进行输出。

4) 实验心得

本实验难度不大，主要保证在做实验之前需要先查看一下参考书，把一些较为基础的内容给掌握了之后便好办了。我们在后续做每一章的实验之前都应该先阅读相关的参考书，只有这样在做实验的学习过程中才能更得心应手。

本实验评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.2s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.9s)
```

3. pingpong(easy)

1) 实验目的

本实验目的主要目的是编写一个程序，该程序中的父子进程通过管道来互相传递字节，父进程首先通过通过 pipe 向子进程发送一段字节，子进程收到后在控制台上上打印相应字符串，接着子进程再将向应的内容通过管道发送给父进程，父进程接收到之后也输出一段响应内容

2) 实验步骤

a) 本实验大致步骤为，首先在 uesr 文件夹下建立名为 pingpong.c 的文件用来编写自己的程序。

b) 通过使用 fork 函数来建立子进程，使用 pipe 函数来建立管道，由于父子进程之间的匿名管道是单向的，因而我们需要建立两个管道，分别用来处理父进程向子进程传递消息与子进程像父进程传递消息，具体传递的消息的方式为调用 write 函数与 read 函数。需要注意的一点是，建立 pipe 的操作必须在 fork 函数之前完成，不然的话父子进程之间用的就不是同一 pipe，传递不了信息。

c) 通过 getpid 函数来获取当前进程的进程号。同时在父子进程输出提示信息时需要同步操作，可以在父进程中添加 wait 函数来父进程等待子进程结束后再运行。

d) 在 makefile 文件中加入‘$U/_pingpong\‘，保证可以运行 pingpong 程序。

实验结果如下：

```
$ pingpong
4: received ping
3: received pong
```

3) 实验中遇到的问题和解决方法

实验中遇到的最主要的问题是对 fork 函数与 pipe 函数不是很了解，通过查阅 xv6 参考书来掌握其相关的内容。

fork 函数与 sleep 类似，也是一个系统调用，程序在运行 fork 函数时，在创建一个子进程，该子进程复制了父亲进程的内容，同时从 fork 函数之后的部分开始运行。可以通过 fork 函数的返回值来判断此时为父进程还是子进程，如果 fork 函数返回值大于 0 表明此时是父进程，如果等于 0，表明此时是孩子进程，小于 0 则表明创建进程失败。

对于 pipe 函数，它的参数为一个二维数组，实际上是通过将该二维数组第一个元素与第二个元素创建为文件描述符，分别对应管道的写端与读端；在进行传递数据时，写端的进程需要先调用 close 函数来关闭读端，读端的进程类似。

4) 实验心得

通过本实验，我大致了解了管道在进程之间传递消息的机制，同时也是对 fork 函数有了更加进一步的理解。在之前上操作系统课的时候，对这二者仅仅停留在一个概念的层面，在本实验中通过创建一个小型的程序让我对这两个函数的运行方式有了更加深入的了解，接下来在后续的实验中可能还需要去读内核部分的源码来查看它们的实现方式。

本实验评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (1.1s)
```

4. primes (moderate)/(hard)

1) 实验目的

本实验目的让我们编写一段程序来实现一个基于管道的质数筛，该方法的基本思想在 xv6 的网站上已经给出了，大致思路将 2 到 35 这些数字（当然可以不仅仅只到 35）通过管

道传递给子进程，子进程每次去其第一个数字作为质数输出，同时判断剩下的数字中是否能够整除该数字，不能整除的数字在通过管道传递给其子进程，知道某个管道中不在拥有内容，说明质数已经被全部输出，此时释放内存，程序结束。

2) 实验步骤

使用该思想来解决这个问题的话，无可避免的需要使用递归的思路，本人的大致实验步骤如下：

a) 首先在 user 文件夹下建立一个名为 primes.c 的文件，在该文件中我们将来实现该程序；同时将 primes 程序添加到 makefile 中，以保证可以在 xv6 中运行

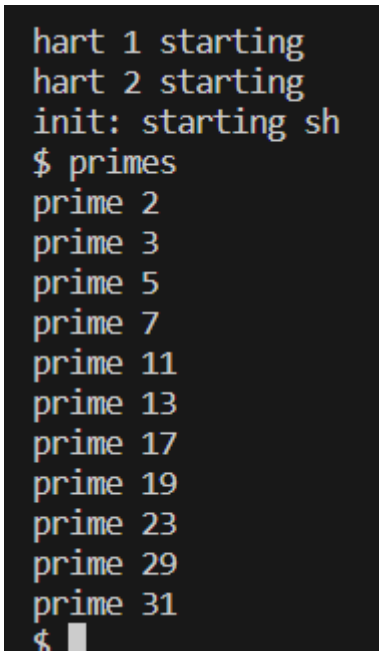
b) 接着我们需要定义一个递归函数，其传入参数为管道的文件描述符，将其中的第一个数字读取出来，如果没有读取出内容则函数递归结束；如果能读出数字，用 fprintf 将其输出到控制台上。在创建子进程之前要注意先 pipe 创建管道。
如下：

```
close(fd[1]);//关闭写端
int pipes[2];

int num=0;
int mid=0;
read(fd[0],&mid,sizeof(mid));

if(mid==0)
    return;
if(pipe(pipes)<0){
    fprintf(2,"error: failed pipe");
    exit(1);
}
fprintf(1,"prime %d\n",mid);//每次打印第一个读到的数字
```

c) 在上述过程之后便需要对管道中剩下的数字进行筛选，选出其中不能被该数字整除的数字，利用该进程中创建的管道将其传入给 fork 的子进程中，同时父进程需要等待子进程结束后再运行。最后可以将 2 到 maxvalue 中的所有质数打印出来。
该部分如下：

```
int p=fork();
if(p==0){
    primes(pipes);
}
else if(p>0){
    close(pipes[0]);//关闭读端
    while(read(fd[0],&num,sizeof(num))){
        if(num%mid!=0)
            write(pipes[1],&num,sizeof(num));
    }
    close(pipes[1]);
    close(fd[0]);
    wait((int*)0);
}
```

d) 最后便是将主函数部分补齐，将 2 到 35 传入管道中并创建子进程来运行 primes 函数。最终输出结果如下：



3) 实验中遇到的问题及解决方法

在做本实验时，遇到的问题主要是父进程通过管道向子进程传递信息这一部分，因为在我做第二个实验时，是将 pipe 放在 fork 前面实现了，当时做的时候并没有注意。但是我在做第三个实验时，是将 pipe 的建立放在了 fork 函数之后，在实际运行过程中，父进程与子进程之间根本传递不了信息。

我遇到的该问题其实是粗心大意所导致的，当时是将父进程穿的字符一个一个的进行了打印，但是子进程那边依然接收不到。最后折腾了好长时间才发现是将 pipe 放到了 fork 函数后面，等于说父进程和子进程之间各自创建了互不联系的 pipe，因而才会出现上述的进程之间传递不了信息的情况。

还有一个问题就是对该方法的执行原理不太明白，后来通过浏览 xv6 网站提供的文章才大致理解。

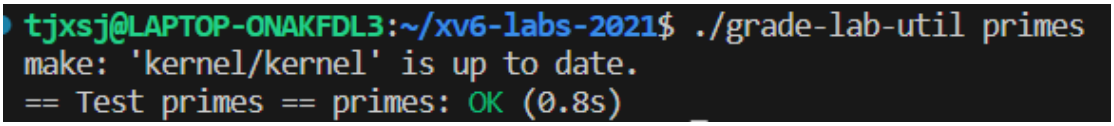
4) 实验心得

本实验其实难度不大，但是我却是花费了较长时间才完成，主要原因还是粗心大意，不小心将 pipe 放在了 fork 之前而导致的。

还有一点就是使用 fprintf 进行打印来调试程序效率挺高的。

简而言之，通过本实验，我对于 pipe，fork，write 与文件描述符的理解更进了一步。并且也学习到了这种筛选素数的方法。

以下为本次实验的评分结果：



5. find (moderate)

1) 实验目的

本实验主要是实现了基于 xv6 文件的系统的 find 程序，其作用为在某指定文件夹中查找某文件，其接受两个参数，其中第一个为所寻找的文件夹，第二个参数为需要寻找到文件名称，同时该实验提供了 ls 的实现可供参考（如何进行文件夹的遍历）。

2) 实验步骤

本实验提供了 user/ls.c 的实现，可以用来参考

a) 首先我们需要现在 user 文件夹下建立一个 find.c 文件用来存放我们的程序，同时将 find 程序加入到 makefile 文件中，使得可以在 xv6 中运行。

b) 我们可以先来查看 ls.c 来查看对应的文件夹遍历方式，在该程序中，函数 fmtname 是用来提取路径中的最内位置的文件名称。在 ls.c 中，它首先是通过 open 函数来打开对应的路径，并返回一个文件描述符 fd。接着调用 fstat 来获取该文件的 stat 详细信息。根据 stat 中的 type 属性来判断是文件还文件夹，若是文件，则直接输出，若为文件夹，则依次 read 其中的目录项，并构造为新的路径递归传入 ls 函数中，进而遍历该路径下的所有文件。

c) 根据上述思路，我们可以直接借用其代码，只需要略微修改。

d) 首先我们的目的是寻找该名称的文件，因而我们的递归函数的参数为两个，一个是当前目录，另一个则是文件名称，思路大致一致，先调用 open 函数获取该路径的文件描述符，接着获取该路径的详细属性。如果是文件，则直接比对名字，符合则输出该路径；如果是目录，则依次 read 出其中的各个目录项，合成新的路径用来调用递归函数。

e) 注意不要将'..'与'.'考虑在内，如果 read 出的目录项是这两个，则重新 read。

```
if(strcmp(de.name,".")==0||strcmp(de.name,"..")==0)
  continue;
if(de.inum == 0)
  continue;
```

f) 实验结果如下：

```
init: starting sh
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
$
```

3) 实验中遇到的问题与解决方法

本次实验中遇到的问题主要还是在阅读 ls 代码时对于 xv6 的文件的组织形式不太了解进而导致在阅读代码时效率低下。

通过在网上查阅相关的 stat 与 dirent 结构体之后，大致对二者有了一定地了解。

stat 结构体用于获取文件的详细信息，包括文件类型、文件大小、访问权限、修改时间等。dirent 结构体用于读取目录中的文件信息，包括文件名和文件的 inode 号。在 xv6 系统中，前者主要储存了文件的磁盘设备，结点数，文件类型等信息；后者则主要用来读取目录中的目录项信息（文件名称），用来合成新路径并加以调用。

4) 实验心得

在本次实验时，通过让我们实现一个小的 find 程序的同时也让我们了解到了 xv6 底层的文件组织与 open, fstat 等操作函数。通过这个实验，我学到了很多关于文件系统和查找算法的知识，并且提升了我的编程技能。

本实验中我首先了解了文件系统的基本概念和组织结构，学习了文件和目录的层次结构，完成实验后，我对操作系统的文件系统和文件查找有了更深入的了解。我意识到文件系统的设计和实现是如此复杂而又重要，它直接影响到我们日常使用计算机的体验。

当然通过这个实验，我也更加熟悉了 C 语言的使用，提升了我的编程能力和调试技巧。

实验评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.3s)
== Test find, recursive == find, recursive: OK (1.1s)
```

6. xargs (moderate)

1) 实验目的

本次实验让我们实现一个较为简单的 xargs 程序，其功能大致为将一行指令的输出结果通过管道运算符重定向到另一行指令中。大体实现步骤是从标准输入中读取对应的参数，将其与 argv 中的参数组合为新的参数表列，接着需要调用 fork 函数来常见子进程，父进程等待，子进程通过调用 exec 函数来执行相应指令。

2) 实验步骤

本实验的实验步骤大致如下：

a) 首先与上面几个代码一致，我们需要在 user 文件夹下建立一个 xargs.c 文件来储存我们的代码，并且在 makefile 文件中的 UPROGS 中加入 xargs 文件来方便 xv6 可以执行调用。

b) 对于程序，其需要从标准输入中读取所需的参数并将这些参数加在自身的参数表列之后,xargs 中的 argv 的第二个参数为要执行代码的函数名，从第三个参数往后则为要执行的指令的参数，我们需要先从标准输入中读取需要追加的参数表列。

```
while(read(0, arguments[argnum]+index, 1) != 0){
    if(arguments[argnum][index] == ' '){
```

循环每次读取一个字节，直到遇到空格，认为读入了一个参数。

c) 接着我们需要对 argv 做处理，取出它的第一个参数（xargs 函数名）。

```
for(i = 1;i < argc; i++){
    arguments[i-1] = argv[i];//复制原有的参数
    if(argv[i]==0)
        break;
}
```

d) 在上述循环读取部分中，如果将标准输入设备的参数全部读取之后，便可以将其与第三步生成的 arguments 参数列表进行合并。这时大体的思路，我这边采取的方案是每读取一个便将其赋值给 arguments 数组。不过需要注意的是，读取完毕之后再参数列表的最后一个元素一定需要以0结尾，否则将不能正常运行exec函数。

e) 最后便是主进程调用 fork 创建子进程，子进程继承了父进程的数据与代码段。在子进程执行 exec 函数，参数为我们上面生成的参数表列及其函数地址，在父进程中则需要等待子进程结束之后释放资源。

```
if(fork() == 0){
    exec(arguments[0], arguments);
    exit(1);
}else {
    wait((int*)0);
}
```

3) 实验中遇到的问题与解决方法

对于本实验，其实在写代码的过程中，初见时不能理解 xargs 函数到底是，通过在网上查找资料，才理解了它的作用。xargs 是一个使用标准数据流构建执行管道的命令，大体就是可以通过标准输入来向其他指令中传递参数。不过其功能较为强大，我们本次实现了只是一个简单版本的。

在中间环节写代码的时候，我创建子进程并运行 exec 函数之后没有反应，查找了较长时间错误，后来通过查看 exec 函数的部分内核代码才发现问题所在。

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp -= strlen(argv[argc]) + 1;
  sp -= sp % 16; // riscv sp must be 16-byte aligned
  if(sp < stackbase)
    goto bad;
  if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[argc] = sp;
}
ustack[argc] = 0;
```

实际上就是在这里，它在传递参数时，将最后一个元素设置为 0，作为结束的标志位，我们在创建参数列表时，实际上也要将最后一个元素设置为 0。这样才能正确运行 exec，实际测试之后确实如此。

在实验网站的 hints 中提到让我们使用 fork 与 exec 函数来读取输入并执行，我个人其实不太理解为什么不能直接就在父进程中 exec 相关的函数。实际上在阅读实验指导书的过程中，也大概明白了其中原因。

Xv6 中 exec()系统调用用于将当前进程替换为新的可执行文件，并从那个新的可执行文件的开始处开始执行代码。换句话说，exec()会完全替换进程的代码和数据，并开始执行新的程序。因此，一旦执行了 exec()，原来进程的代码就不再执行，完全被替换。而如果直接在父进程中 exec 函数的话，对于多行输入的参数（以'\n'分隔）则只会运行将第一个参数传入的情况。

因而在我们的代码中，需要使用 fork 创建子进程，用子进程去执行对应的函数，父进程得以继续读取参数，直到 read 函数返回 0 表示读取完毕。

4) 实验心得

本次实验的总体来讲还是较为容易，在完成本次实验的过程中，我也了解到了许多知识。

首先，我了解了使用 fork 和 exec 系统调用来创建子进程并执行外部命令。这让我更深入地理解了进程的概念和操作系统对进程的管理方式。这在实验指导书中其实有所提及。

我还学到了如何处理管道和文件重定向。在 xargs 程序中，我需要将外部命令的输出通过管道传递给父进程，并将其重定向到文件中。这让我更加熟悉了 Linux 系统中的 I/O 重定向和管道机制。

本实验评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (0.9s)
```

总结

本次实验整体难度不大，通过本次实验我也学习到了许多关系 xv6 操作系统的知识。

首先我学会了如何搭建和运行 xv6 操作系统。通过网上一些资料的步骤，我成功地将 xv6 操作系统编译并运行在我的虚拟机上。其次本实验通过让我实际动手去实现一些常用的系统调用，同时也熟悉了一些 linux 下 xv6 操作系统的操作。

通过完成 6.S081 实验第一个 lab1，我对平时操作系统理论课上的原理和知识有了更深入的了解。

本次实验总体评分如下：