

Xv6 and Utilities 实验报告



Tongji University

专 业：软 件 工 程

指 导 老 师：王 冬 青

年 级：大 学 二 年 级

学 号：2153061

姓 名：谢 嘉 麒

1. Boot xv6 (easy)

1.1.实验目的

1.2.实验步骤

1.2.1.实验环境选择

1.2.2.安装RISC-V工具链

1.2.3.抓取Xv6实验代码

1.2.4.启动Xv6系统

1.3.实验中遇到的问题和解决方法

1.3.1.git clone命令报错域名解析暂时失败

1.3.2.ssh远程连接报错

1.4.实验心得

2. sleep (easy)

2.1.实验目的

2.2.实验步骤

2.2.1.熟悉Xv6编码风格

2.2.2.查找sleep工具的系统调用

2.2.3.编辑sleep.c

2.2.4.编译运行Xv6

2.3.实验中遇到的问题和解决方法

2.4.实验心得

3. pingpong (easy)

3.1.实验目的

3.2.实验步骤

3.2.1.资料搜集与学习

3.2.2.编辑pingpong.c

3.2.3.编译与检测

3.3.实验中遇到的问题和解决方法

3.3.1.对于系统调用的使用

3.3.2.设置管道的方法问题

3.4.实验心得

4. primes (moderate)/(hard)

4.1.实验目的

4.2.实验步骤

4.2.1.确定最初的父进程并写入2-35

4.2.2.在子进程中的部分

4.2.3.如果fork未进入子进程

4.2.4.编译与检测

4.3.实验中遇到的问题和解决方法

4.3.1.基于管道的多进程质数筛的理解

4.3.2.递归实现质数筛

4.4.实验心得

5. find (moderate)

5.1.实验目的

5.2.实验步骤

5.2.1.浏览user/lis.c学习读目录方法

5.2.2.移植lis.c代码并实现find.c

5.2.3.编译与检测

5.3.实验中遇到的问题和解决方法

5.3.1.文件的理解与Unix视角下的文件结构

5.4.实验心得

6. xargs (moderate)

6.1.实验目的

6.2.实验步骤

- 6.2.1.理解xargs的使用
 - 6.2.2.xargs.c的具体实现
 - 6.2.3.编译与检测
- 6.3.实验中遇到的问题和解决方法
- 6.4.实验心得
- 7.Lab Xv6 and Utilities总结
 - 7.1.编译与检测
 - 7.2.实验心得与体会

1. Boot xv6 (easy)

1.1.实验目的

配置环境支持MIT-Lab提供的xv6系统，并为后续试验提供实验环境和本地编译。

1.2.实验步骤

1.2.1.实验环境选择

实验环境选用Linux系统，笔者在Windows本机上安装Vmware Workstation作为虚拟机，在虚拟机中配置Ubuntu20.04.6（清华镜像）系统作为实验环境。

1.2.2.安装RISC-V工具链

Ubuntu系统配置成功后，需要运行以下命令安装RISC-V工具链：

```
sudo apt-get install -y git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

1.2.3.抓取Xv6实验代码

工具与环境配置完成后，在Ubuntu系统中配置git，并在终端输入以下代码抓取Xv6实验代码库：

```
git clone git://g.csail.mit.edu/xv6-labs-2021
```

1.2.4.启动Xv6系统

在1.2.3.中将2021版Xv6代码抓取到Ubuntu系统下的home/xv6-labs-2021文件夹中，需要对代码进行编译并启动系统，具体操作方法如下：

a.在home目录下打开终端，利用cd指令跳转到xv6-labs-2021目录中（或者直接在xv6-labs-2021中打开终端）：

```
cd xv6-labs-2021
```

b.利用make qemu指令编译运行xv6，正常运行下会输出多行log，并启动xv6系统：

```
make qemu
```

c.系统第一次运行后会显示很多log，之后再次使用make qemu指令运行后会直接显示结果，运行结果如下：

```
cinderlord@ubuntu:~/xv6-labs-2021$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M
-smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device vir
tio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
```

d.退出系统时，先按下Ctrl+A键，再按下X键终止系统运行。

1.3.实验中遇到的问题和解决方法

1.3.1.git clone命令报错域名解析暂时失败

实验问题主要在于Ubuntu系统的安装和Linux系统的环境配置。最初安装Ubuntu24.04版本后，在抓取git代码仓库时会报错无法连接或域名解析暂时失败。将版本下调至20.04后抓取成功。

1.3.2.ssh远程连接报错

笔者最初希望利用ssh远程连接本机VSCode和Ubuntu系统，但连接时不断报错并拒绝连接。考虑到Ubuntu系统已经提供较为完善的GUI界面，且目前虚拟机已经能够在本机上较为稳定地运行，故选择在Ubuntu系统中安装VSCode for Linux来对代码进行编辑。

1.4.实验心得

通过安装和调试Linux虚拟机，进一步熟悉了Linux系统安装与运行的过程，同时也学习了Linux的基本指令和使用方法，对配置环境有了更进一步的理解。

2. sleep (easy)

2.1.实验目的

实验要求实现Unix工具sleep，使其等待给定的参数时间段（两次时钟中断的时间间隔）并在等待后退出。要求在user/sleep.c中书写实现代码。

2.2.实验步骤

2.2.1.熟悉Xv6编码风格

通过浏览其他Xv6代码来熟悉系统的代码编写风格，从头文件，基本主函数main和程序返回/退出几个方面来进行把握。

2.2.2.查找sleep工具的系统调用

在user/user.h中找到其为sleep定义了如下的系统调用：

```
int sleep(int);
```

可以看出sleep接受一个参数，并且参数类型为int。

2.2.3.编辑sleep.c

在sleep.c中引入头文件"kernel/types.h", "kernel/stat.h"和"user/user.h"。

因为工具比较简单, 故而可以在单个main函数中进行实现。通过步骤2.2.1.可以得到Xv6系统代码中main函数一般为带参main函数, 其中参数argc代表参数个数, *argv[]用来存放具体参数。

a.首先检查参数个数是否合法, 并对结果进行报告;

b.如果得到合法参数, 即参数以char类型存放在argv[1]中, 则利用atoi函数将其转换为int类型, 并且作为输入参数传递给sleep(int)函数进行函数调用。

```
//sleep according to the given parameter
sleep(atoi(argv[1]));
```

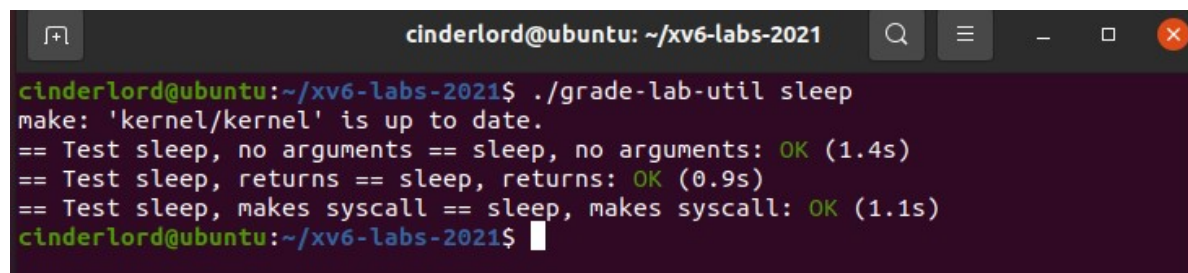
c.main函数实现后利用exit(0)退出函数。

d.注意, 完成代码编辑后需要将源文件加入makefile来保重工具链编译自定义工具。方法如下: 在makefile文件中找到UPROGS变量, 并在其后加入"\$U/_sleep", 保存文件。

2.2.4.编译运行Xv6

在终端中运行make qemu指令, 重新编译Xv6代码。利用实验提供的测评工具进行检查, 在终端中输入./grade-lab-util sleep, 观察检测结果。

检测结果如下, 即工具能够完成预期功能:



```
cinderlord@ubuntu: ~/xv6-labs-2021
cinderlord@ubuntu:~/xv6-labs-2021$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.4s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.1s)
cinderlord@ubuntu:~/xv6-labs-2021$
```

2.3.实验中遇到的问题和解决方法

笔者最初并不熟悉Xv6代码风格及带参main函数的规范及含义, 通过搜集资料和学习解决了这一问题。

2.4.实验心得

通过本次实验, 笔者熟悉了实验代码的编写与检验方法, 更进一步理解了sleep工具的实现与作用。

3. pingpong (easy)

3.1.实验目的

实现pingpong工具, 进而验证Xv6的部分进程通讯机制。具体要求为:

a.程序创建子进程, 并通过管道与子进程进行通信;

b.父进程发送一字节数据给子进程, 子进程接收数据后在shell中打印: receieved ping。

c.子进程接收数据后向父进程发送一字节数据, 父进程收到数据后在shell中打印: receieved pong。

d. 要求创建user/pingpong.c文件存放工具代码

3.2.实验步骤

3.2.1.资料搜集与学习

为了更好地实现pingpong工具，则需要对实现pingpong所需要的系统调用进行了解和学习。因为这一工具涉及到管道，进程，管道读写和进程号几个部分，因此需要学习以下几种系统调用：

- a. fork系统调用：用于创建子进程；
- b. pipe系统调用：用于创建管道；
- c. read系统调用：用于读取管道；
- d. write系统调用：用于写管道；
- e. getpid系统调用：用于查询进程号；

此外实验要求在shell中进行格式化输出，因此查询user/user.h可以找到实验提供了printf函数，可以用于输出。

3.2.2.编辑pingpong.c

pingpong工具设计进程之间的管道通讯，由于其功能和实现并不复杂，故可以直接在main函数中完成代码。

在创建子进程前，为进程设置两个双向管道，通过以下代码进行实现：

```
//declare two pipe
int p1[2],p2[2];
pipe(p1);
pipe(p2);
```

创建管道后，调用fork()系统调用来创建子进程，通过资料可以得知，fork()会返回两种不同的值，父进程得到子进程的pid，而子进程则得到返回值0，因此，可以通过系统调用返回值来区分父进程和子进程：

```
//create a child progress
int pid=fork();
```

在子进程中，分别关闭两个管道的读端口和写端口（使用close()实现），此时子进程通过p2管道的读端口p2[0]来获取一比特信息(read())，如果得到，则打印：received ping。

在父进程中，分别关闭p1，p2两个管道的写端口和读端口（使用close()实现），通过pid = getpid()语句获得pid，并将缓冲区buf中的内容写入p2的写端口(write())。此时若能够从p1的读端口读取一比特数据(read())，则打印：received pong。

```
if(pid==0)
{
    //close the reading of p1 and writing of p2
    close(p1[0]);
    close(p2[1]);
    pid = getpid();
    //if successfully read byte from p2
    if(read(p2[0],buf,1)==1)
```

```

    {
        printf("%d: received ping\n", pid);
        //write buf into reading of p1
        write(p1[1], buf, 1);
    }
}
else
{
    //close the writting of p1 and reading of p2
    close(p1[1]);
    close(p2[0]);
    pid = getpid();
    //write buf into p2
    write(p2[1], buf, 1);
    //if successfully read byte
    if(read(p1[0], buf, 1) == 1)
    {
        //pong successfully
        printf("%d: received pong\n", pid);
    }
}
}

```

3.2.3. 编译与检测

需要修改makefile文件，将“\$U/_pingpong\”加入环境变量UPROGS。

在xv6文件目录下运行指令make qemu进行编译，输入pingpong，打印结果符合要求：

```

$ pingpong
4: received ping
3: received pong
$ QEMU: Terminated

```

退出系统，在文件目录下输入 ./grade-lab-util pingpong，使用实验提供的测评工具进行检测，得到如下结果，则工具合格：

```

cinderlord@ubuntu:~/xv6-labs-2021$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (1.5s)

```

3.3.实验中遇到的问题和解决方法

3.3.1.对于系统调用的使用

实验要求自主了解并学习实现工具所需要的系统调用，因此首先需要判断这一工具涉及到哪些系统调用，并对这些内容逐个学习，理解。

3.3.2.设置管道的方法问题

笔者在尝试过程中发现，将管道的设置安置在进程创建之后会产生预期之外的结果。经过对系统调用fork()的了解得知，fork会将父进程的内容完整复制到子进程中，因此可以在父进程中打开双向管道，再创建子进程，这样能够简化困难并避免错误。

3.4.实验心得

通过本次实验，笔者进一步学习了管道的相关内容，通过读写管道时使用read和write不难看出，管道本质上是桥接进程通信的特殊文件。同时管道也能够灵活使用，通过读写端口的打开与关闭，来实现不同的功能。

在进程创建部分也颇有收获，首先了解了fork系统调用的基本内涵和使用方法，理解了进程创建时父进程和子进程之间的交互与联系。本次实验将进程创建和管道联系起来，也为这两个知识模块架起了桥梁。

4. primes (moderate)/(hard)

4.1.实验目的

本次实验基于Doug McIlroy（Unix管道的发明者）提出的多进程质数筛案例。实验要求使用pipe和fork来建立起管道线，最初的父进程产生从2到35的整数，并将其送入管道线。之后产生子进程检验一个质数是否是输入的因子。第一个不能被质因子整除的数字用于产生下一个子进程，并且输出这个数字，且其质数筛的本质会保证这一数字是质数。实验需要创建一个进程通过管道从左邻居读取数据，并在合适情况下通过另一个管道写给右邻居。

4.2.实验步骤

4.2.1.确定最初的父进程并写入2-35

由于子进程的检测与管道通信是一个相对重复的过程，因此可以使用递归来编写代码。当代码第一次进入main函数时，设置最初的父进程，这一功能通过标识符isFirst实现。

在最初的父进程中置isFirst=0，表示以后的递归不再与父进程有关（即不再进入这一代码块）。创建管道p1，并将其读端口赋予fdr，将写端口赋予fdw，利用for循环将整数2-35写入fdw即管道的写端口中。为了安全起见，在写入完成后将p1的写端口关闭。

4.2.2.在子进程中的部分

通过fork()==0进入子进程，利用read系统调用从fdr读端口读取一个整数，重新声明p1管道，并将其写端口继续赋予fdw。利用while循环不断执行 read(fdr,(void*)&n,8) 语句，即不断从fdr读端口读取数字，判断如果这个数字符合质数筛条件，则利用 write(fdw,(void*)&n,8) 语句将其写入fdw写端口。在循环读取结束后更新fdr，关闭fdw，并且递归调用main函数。核心代码如下：

```
//write a number into fdw if it can't be divided by p
while(read(fdr,(void*)&n,8))
{
    if(n%p!=0)
    {
        write(fdw,(void*)&n,8);
    }
}
//update fdr
fdr=p1[0];
close(fdw);
main(argc,argv);
```


4.2.3.如果fork未进入子进程

如果fork()!=0未进入子进程，则利用wait系统调用来要求父进程等待，并关闭fdr端口来保证工具的安全性。

4.2.4.编译与检测

依旧是需要将\$U/_primes\加入makefile文件的环境变量UPROGS中，方便Linux系统进行编译。在Xv6文件目录下启动终端，并且在其中运行make qemu进行编译运行。系统运行成功后在终端中输入./grade-lab-util primes进行自动评测，结果如下：

```
cinderlord@ubuntu:~/xv6-labs-2021$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.7s)
```

4.3.实验中遇到的问题和解决方法

4.3.1.基于管道的多进程质数筛的理解

实验背景是Doug McIlroy提出的基于管道的多进程质数筛。因此在进行实验前需要对这一工具进行一定的理解和学习。实验只需要用管道编写一个并发版本的初筛。

4.3.2.递归实现质数筛

算法直接进行实现的话会较为复杂，且产生很多的代码冗余。考虑到工具涉及到子进程的不断读取和写入，可以使用递归来解决问题。然而使用递归会消耗较多的空间和时间，因此笔者猜测这一方法并不是实现多进程质数筛的最优办法。

4.4.实验心得

通过本次实验，笔者进一步学习和理解了多进程质数筛这一Unix系统中的重要工具，通过将进程和管道进一步结合，巩固了进程调用与管道通信的知识。实验通过具体的工具来引导笔者将这两部分具象化地结合起来，使知识结构更加整体化，并对各种系统调用的代码使用方法与注意事项有了更进一步的认识。

5. find (moderate)

5.1.实验目的

实验要求编写一个简单的Unix查找程序，即查找目录树中具有特定名称的所有文件，将代码实现在user/find.c中。

5.2.实验步骤

5.2.1.浏览user/lis.c学习读目录方法

根据提示先浏览学习user/lis.c中读目录的方法，不难看出其函数lis()使功能的主要实现，而主体功能性代码是以下部分：

```
char buf[512], *p;
int fd;
struct dirent de;
struct stat st;
```

```

if((fd = open(path, 0)) < 0){
    fprintf(2, "ls: cannot open %s\n", path);
    return;
}

if(fstat(fd, &st) < 0){
    fprintf(2, "ls: cannot stat %s\n", path);
    close(fd);
    return;
}

```

这部分代码首先根据路径path以读的方式打开文件（Unix中一切都可以看做文件，包括普通文件，目录和设备），将“文件名”返回给fd。接着使用fstat()把这个file的信息填充到st结构体中。

在kernel/stat.h中可以查到这个结构体的定义。

接着在switch-case语句中判断file的类型。如果是文件，就输出文件名称，种类，inode number和文件大小；如果是目录，就利用dirent结构体不断读取sizeof(struct dirent)大小的数据，得到下一个文件的名称。不断用stat将文件信息填充到st结构体中，并输出信息。

5.2.2.移植ls.c代码并实现find.c

根据5.2.1的分析结果将ls.c中可以重复使用的代码移植到find.c中，包括ls函数的结构部分和fmtname()函数。

ls()函数实现与ls.c中基本相同，根据文件路径打开目录并填充结构体后进入switch-case结构，在T_DIR中，与ls.c中不同，不需要输出所有文件的所有信息，而是进行筛选，筛选到符合查找条件的名字再输出buf缓冲区中存储的相对路径即可。如果file类型是目录且并不是.和..，则进入目录进行递归查找，具体代码修改如下：

```

case T_DIR:
    if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
        printf("ls: path too long\n");
        break;
    }
    strcpy(buf, path);
    p = buf+strlen(buf);
    *p++ = '/';
    while(read(fd, &de, sizeof(de)) == sizeof(de)){
        if(de.inum == 0)
            continue;
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0;
        if(stat(buf, &st) < 0){
            printf("ls: cannot stat %s\n", buf);
            continue;
        }
        printf("%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
    }
    break;
}

```

5.2.3.编译与检测

同样将其加入makefile的环境变量UPROGS中进行编译，进入Xv6目录下打开终端执行make qemu指令进行编译运行。在目录终端下输入./grade-lab-util find进行自动评测，评测结果如下：

```
cinderlord@ubuntu:~/xv6-labs-2021$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.8s)
(Old xv6.out.find_curdir failure log removed)
== Test find, recursive == find, recursive: OK (1.4s)
(Old xv6.out.find_recursive failure log removed)
```

5.3.实验中遇到的问题和解决方法

5.3.1.文件的理解与Unix视角下的文件结构

笔者最初并不了解Unix视角下的文件结构，但结合操作系统文件结构和资料的学习，理解了Unix系统中一切都可以看作file，包括所有的普通文件，目录和设备。理解了这一点后再思考ls.c中的代码就会方便很多，也能够理解switch-case中将目录和文件进行并列判断的用意。

5.4.实验心得

通过本次实验，笔者进一步了解到Unix文件系统的基本实现方法，同时也通过比较Unix和传统Windows文件系统，加深了自己对这两种操作系统的文件系统的理解和认识。通过自主实现递归查找文件名，与操作系统作业-文件系统中实现的递归查找Windows文件名进行对比思考，不难看出两种系统的相同点与不同点。

6. xargs (moderate)

6.1.实验目的

实验要求编写一个简单的Unix xargs程序，实现从标准输入读取行，将每一行作为参数提供命令，并运行命令。具体代码实现在文件user/xargs.c中。

6.2.实验步骤

6.2.1.理解xargs的使用

通过之前的实验已经了解到shell中可以使用管道|将其之前的命令的标准输出作为之后命令的标准输入。而xargs的功能是将其之前命令的标准输出作为之后命令的附加命令行参数。

6.2.2.xargs.c的具体实现

首先将参数总数preargnum赋予argnum，并且设置一个全为0的缓冲区。

利用for语句进入无限循环，不断read，并根据其结果进行下一步行动：

- 当read返回0，表示这是file的结尾，则没有更多的输出，直接结束程序；
- 当read返回小于0，则表示发生错误，打印error happen，并且exit(1)；
- 其余情况下，特殊判断ch == '\n'和ch == ' '的情况，前者将参数存入args中并返回，此时正确读取；后者将参数写入args中，并为下一个参数置0；
- 其余情况就将ch添加到arg的buf缓冲区即可。

主要功能代码如下：

```

if (ch == '\n')
{
    //store the parameter into args
    memcpy(args[argnum], arg_buf, strlen(arg_buf) + 1);
    argnum++;
    //read successfully
    return 1;
}
else if (ch == ' ')
{
    //store the parameter into args
    memcpy(args[argnum], arg_buf, strlen(arg_buf) + 1);
    argnum++;
    //set 0 for next parameter
    memset(arg_buf, 0, sizeof(arg_buf));
}
else
{
    //add ch to arg_buf
    arg_buf[strlen(arg_buf)] = ch;
}

```

6.2.3.编译与检测

加入环境变量后，在Xv6目录下执行make qemu命令。编译成功后退出系统，并执行自动检测程序./grade-lab-util xargs，得到检测结果如下，试验成功：

```

cinderlord@ubuntu:~/xv6-labs-2021$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (1.9s)

```

6.3.实验中遇到的问题和解决方法

实验主要涉及到命令的读取。值得注意的一点事，要区分管道|和xargs的|xargs之间的区别，同时理解标准输入和命令行参数这两个不同的概念。

6.4.实验心得

通过本次实验，笔者进一步理解了标准输入和命令行参数这两个基本概念，同时通过对管道和xargs的比较学习，建立了更加全面的知识体系。

7.Lab Xv6 and Utilities总结

7.1.编译与检测

实验要求在完成所有实验后，在Xv6目录下创建time.txt文件，写入完成实验的小时数，在终端中执行./grade-lab-util命令，对整个实验进行评分，结果如下：

```
cinderlord@ubuntu:~/xv6-labs-2021$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.9s)
== Test sleep, returns == sleep, returns: OK (1.2s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.8s)
minal)st pingpong == pingpong: OK (1.0s)
)st primes == primes: OK (1.2s)
== Test find, in current directory == find, in current directory: OK (1.1s)
== Test find, recursive == find, recursive: OK (1.4s)
== Test xargs == xargs: OK (1.2s)
== Test time ==
time: OK
Score: 100/100
```

可见测试全部通过，得到满分。

7.2.实验心得与体会

通过本次实验，笔者主要收获如下几点：

- a. 在Windows主机下完整配置虚拟机和Ubuntu系统。
- b. 对于进程，管道，多进程质数筛，Unix视角下的文件结构和文件操作，xargs等知识体系有了进一步的理解。
- c. 通过阅读Xv6代码了解其系统调用，通过查阅官方文档来学习具体的系统实现和调用规则。