**实验一 Xv6 and Unix utilities**

**1.1 实验目的**

1. 熟悉xv6以及其系统调用

2. 了解git的基本用法

3. 通过阅读xv6-book了解进程和内存

4. 掌握I/O、文件描述符、管道和文件系统的相关知识

**1.2 实验内容**

1.2.1 Boot xv6（easy）

掌握如何运行程序并且判断其是否正确，了解xv6的运行指令。

1.2.2 sleep（easy）

为xv6实现UNIX程序sleep；您的睡眠应该暂停用户指定的节拍数。tick是xv6内核定义的时间概念，即计时器芯片两次中断之间的时间。您的解决方案应该在user/sleep.c文件中。

1.2.3 pingpong (easy)

编写一个程序，使用UNIX系统调用在一对管道上的两个进程之间“乒乓”一个字节，每个方向一个。父级应向子级发送一个字节；子进程应打印“<pid>：received ping”，其中<pid>是其进程ID，将管道上的字节写入父进程，然后退出；父级应从子级读取字节，打印“<pid>：received pong”，然后退出。您的解决方案应该在user/pingpong.c文件中。

1.2.4 primes (moderate)/(hard)

使用管道编写prime sieve的并发版本。您的解决方案应该在user/primes.c文件中。您的目标是使用管道和叉来设置管道。第一个进程将数字2到35输入到管道中。对于每个素数，您将安排创建一个进程，该进程通过一个管道从其左邻居读取数据，并通过另一个管道向其右邻居写入数据。由于xv6的文件描述符和进程数量有限，因此第一个进程可以在35处停止。

1.2.5 find（moderate）

编写一个简单版本的UNIX查找程序：查找目录树中具有特定名称的所有文件。您的解决方案应该在user/find.c文件中。

1.2.6 xargs（moderate）

编写UNIX xargs程序的简单版本：从标准输入中读取行，并为每行运行一个命令，将该行作为参数提供给该命令。您的解决方案应该在user/xargs.c文件中。

1.2.7 submit

提交实验。

**1.3 实验过程**

1.3.1 启动xv6

（1）获取xv6资源并且切换到util分支

通过运行如下指令进行clone和切换分支

$ **git clone git://g.csail.mit.edu/xv6-labs-2020**

$ **cd xv6-labs-2020**

$ **git checkout util**

（2）查看git日志

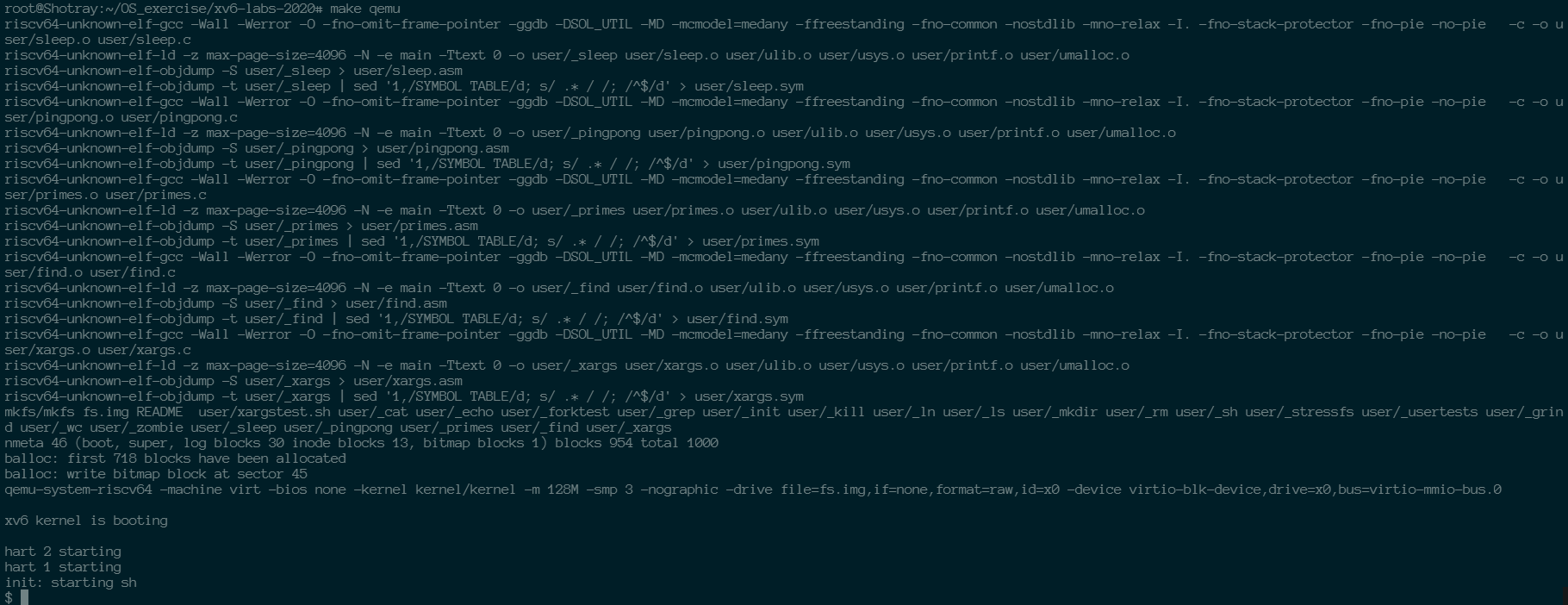
$ **git log**

（3）尝试第一次提交

$ **git commit -am 'my solution for util lab exercise 1'**

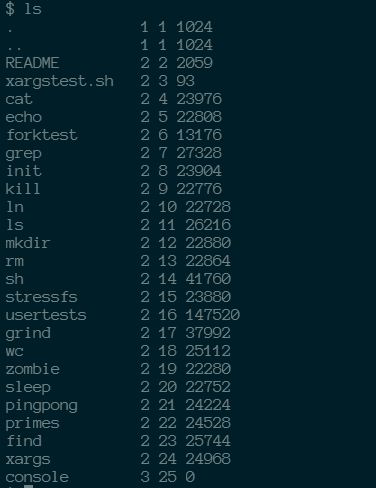
（4）运行xv6

$ **make qemu**



（5）查看目录

$ **ls**



（6）退出xv6

**Ctrl-a x**

1.3.2 sleep

（1）在kernel/sysproc.c中查看sleep这一系统调用

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;

}

（2）在所写的sleep.c中调用系统函数，当缺少参数或参数过多时报错，并且调用系统函数sleep，并在程序结尾添加exit()函数

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

int

main(int argc, char \*argv[])

{

    if(argc!=2){

        printf("Argument error!\n");

        exit(1);

    }

    int number=atoi(argv[1]);

    sleep(number);

    exit(0);

}

（3）将sleep程序添加到Makefile中的UPROGS中；完成后，make qemu将编译程序，并能够从xv6 shell运行它



（4）运行sleep





1.3.3 pingpong

（1）阅读xv6-book了解文件描述符所代表的含义。其中，0表示标准输入（standard input），1表示标准输出（standard output），2表示标准错误（standard error）。

（2）调用read(fd,buf,n)从文件描述符fd中读取最多n个字节，将它们复制到buf中，并返回读取的字节数；调用write(fd,buf,n)将n个字节从buf写入文件描述符fd，并返回写入的字节数,只有在发生错误时，才会写入少于n个字节。

（3）管道是一个小型内核缓冲区，作为一对文件描述符公开给进程，一个用于读取，一个用于写入。将数据写入管道的一端，可以从管道的另一端读取该数据。管道为进程提供了一种通信方式。

（4）fork()函数的返回值为0时代表当前处于子进程中。

（5）当处于父进程时，将（任意）数据写入管道中并关闭p[1]，在数据写入后子进程从管道中读入并关闭p[0]，输出<pid>:received ping并将数据写入管道，作为子进程运行完的判断并关闭p[1]，当父进程等待写入完成后进行数据的读取，读取到后证明子进程已经运行完成，关闭p[0]并输出<pid>:received pong。

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

int

main(int argc, char \*argv[])

{

    if(argc!=1){

        printf("Argument error!\n");

        exit(1);

    }

    int p[2];

    pipe(p);

    char buf[2];

    if(fork()==0){

        int pid=getpid();

        if(read(p[0],buf,1)!=1){

            printf("Read error!\n");

            exit(1);

        }

        close(p[0]);

        printf("%d: received ping\n",pid);

        if(write(p[1],buf,1)!=1){

            printf("Write error!\n");

            exit(1);

        }

        close(p[1]);

    }else{

        int pid=getpid();

        char info[2]="a";

        if(write(p[1],info,1)!=1){

            printf("Write error!\n");

            exit(1);

        }

        close(p[1]);

        wait(0);

        if(read(p[0],buf,1)!=1){

            printf("Read error!\n");

            exit(1);

        }

        close(p[0]);

        printf("%d: received pong\n",pid);

    }

    exit(0);

}

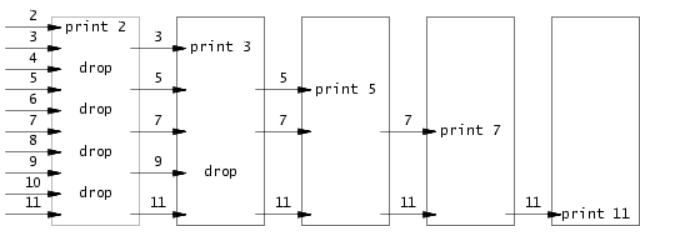
（6）运行pingpong





1.3.4 primes

（1）本次编写的素数筛原理如下图所示，从2开始进行遍历，并消去所有当前数字的倍数，直到完成筛选。



（2）通过递归实现素数筛，并及时关闭不需要的文件描述符，防止资源耗尽。

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

void newPro(int p[2]){

    int prime;

    int flag;

    int n;

    close(p[1]);

    if(read(p[0],&prime,4)!=4){

        printf("Read error");

        exit(1);

    }

    printf("prime %d\n",prime);

    flag=read(p[0],&n,4);

    if(flag){

        int newp[2];

        pipe(newp);

        if(fork()==0){

            newPro(newp);

        }else{

            close(newp[0]);

            if(n%prime)

                write(newp[1],&n,4);

            while(read(p[0],&n,4))

                if(n%prime)

                    write(newp[1],&n,4);

            close(p[0]);

            close(newp[1]);

            wait(0);

        }

    }

    exit(0);

}

int

main(int argc, char \*argv[])

{

    int p[2];

    pipe(p);

    if(fork()==0){

        newPro(p);

    }else{

        close(p[0]);

        for(int i=2;i<=35;i++){

            if(write(p[1],&i,4)!=4){

                printf("Write error!\n");

                exit(1);

            }

        }

        close(p[1]);

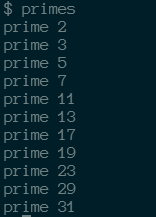
        wait(0);

    }

    exit(0);

}

（3）运行primes





1.3.5 find

（1）查看ls.c阅读如何进行程序的编写。

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/fs.h"

char\*

fmtname(char \*path)

{

  static char buf[DIRSIZ+1];

  char \*p;

  // Find first character after last slash.

  for(p=path+strlen(path); p >= path && \*p != '/'; p--)

    ;

  p++;

  // Return blank-padded name.

  if(strlen(p) >= DIRSIZ)

    return p;

  memmove(buf, p, strlen(p));

  memset(buf+strlen(p), ' ', DIRSIZ-strlen(p));

  return buf;

}

void

ls(char \*path)

{

  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;

  }

  switch(st.type){

  case T\_FILE:

    printf("%s %d %d %l\n", fmtname(path), st.type, st.ino, st.size);

    break;

  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;

  }

  close(fd);

}

int

main(int argc, char \*argv[])

{

  int i;

  if(argc < 2){

    ls(".");

    exit(0);

  }

  for(i=1; i<argc; i++)

    ls(argv[i]);

  exit(0);

}

（2）参考ls.c的编写方法对find.c进行编写，不用对.和..进行判断，并将所有包含查询文件的路径输出

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/fs.h"

void

find(char const \*path,char const \*target)

{

  char buf[512], \*p;

  int fd;

  struct dirent de;

  struct stat st;

  if((fd = open(path, 0)) < 0){

    fprintf(2, "find: cannot open %s\n", path);

    exit(1);

  }

  if(fstat(fd, &st) < 0){

    fprintf(2, "find: cannot stat %s\n", path);

    exit(1);

  }

  switch(st.type){

  case T\_FILE:

    fprintf(2,"Wrong");

    exit(1);

  case T\_DIR:

    if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){

      printf("find: 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||strcmp(de.name,".")==0||strcmp(de.name,"..")==0)

        continue;

      memmove(p, de.name, DIRSIZ);

      p[DIRSIZ] = 0;

      if(stat(buf, &st) < 0){

        printf("find: cannot stat %s\n", buf);

        continue;

      }

      if(st.type==T\_DIR){

          find(buf,target);

      }else if(st.type==T\_FILE){

          if(strcmp(de.name,target)==0){

              printf("%s\n",buf);

          }

      }

    }

    break;

  }

  close(fd);

}

int

main(int argc, char \*argv[])

{

  if(argc !=3 ){

    fprintf(2,"Argument error!\n");

    exit(1);

  }

  char const \*path=argv[1];

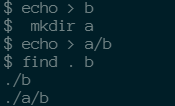
  char const \*target=argv[2];

  find(path,target);

  exit(0);

}

（3）运行find





1.3.6 xargs

（1）为实现xargs指令需要对字符串进行处理，从缓冲流中读入字符并进行分割处理

#include "kernel/types.h"

#include "kernel/stat.h"

#include "kernel/param.h"

#include "user/user.h"

int getArgv(char \*new\_argv[],int curr\_argc){

   char buf[1024];

   char temp;

   int n;

   char \*p=buf;

   while((n=read(0,&temp,1))&&temp!='\n'){

      \*p=temp;

      p++;

   }

   \*p='\0';

   if(n==0) return 0;

   int offset=0;

   while(offset<n){

      new\_argv[curr\_argc++]=buf+offset;

      while(buf[offset]!=' '&&offset<n){

         offset++;

      }

      while(buf[offset]==' '&&offset<n){

         buf[offset++]='\0';

      }

   }

   return curr\_argc;

}

int main(int argc, char \*argv[])

{

   if(argc<=1){

      fprintf(2,"Argument error!\n");

      exit(1);

   }

   char \*new\_argv[MAXARG];

   for(int i=1;i<argc;i++){

      new\_argv[i-1]=malloc(strlen(argv[i]+1));

      strcpy(new\_argv[i-1],argv[i]);

   }

   int curr\_argc;

   int status;

   while((curr\_argc=getArgv(new\_argv,argc-1))!=0){

      new\_argv[curr\_argc]=0;

      if(fork()==0){

         exec(argv[1],new\_argv);

         fprintf(2,"exec failed!\n");

         exit(1);

      }

      wait(&status);

   }

   exit(0);

}

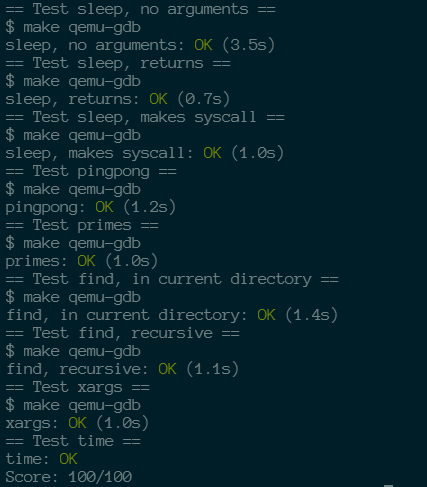
（2）运行xargs





**1.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。



**1.5 实验小结**

本次实验对xv6系统进行了简单的介绍，并且通过各个实验对read、write等系统调用更加熟悉，了解了管道的基本用法等，逐渐从最基本的操作系统接口对操作系统进行了解。

在完成实验的过程中也遇到了一些困难，比如怎么进行素数筛的编写、如何通过read和write进行管道接收消息的判断等，我通过阅读xv6-book以及询问同学等方法对遇到的困难进行解决，最终完成了实验。