# 2023/09/02

## XV6第一章

了解了各种syscall的基本功能。

### fork（）

我认为fork（）调用非常有趣，fork（）会同时返回两个值，并且从父进程分裂出一个内存一模一样的子进程。经过我的观察，我认为fork产生子进程并拷贝内存的时机是在系统刚刚执行到fork的时候，而产生返回值是在fork的中间（也许是结尾）；因为这样才有可能在子进程与父进程中都获得fork的返回值。也就是说，fork执行产生新进程并拷贝内存的过程中，会一并将fork还未执行完的部分（也就是产生返回值的部分）一并拷贝到子进程中。

在我想通这一点之前，我非常疑惑于文档中给出的形如

if(fork() == 0) {…}

else {…}

的调用。难以理解父进程中的fork为何以及何时产生返回值。

### fork（）与exec（）的配合

说来惭愧，当我最初阅读到*“你可能会疑惑为什么 fork 和 exec 为什么没有被合并成一个调用，我们之后将会发现，将创建进程——加载程序分为两个过程是一个非常机智的设计。*”的时候，并没有发现自己有类似的疑惑。

但是当我阅读到样例代码段:

char \*argv[2];

argv[0] = "cat";

argv[1] = 0;

if(fork() == 0) {

close(0);

open("input.txt", O\_RDONLY);

exec("cat", argv);

}

的时候便恍然大悟。fork之后不在子程序中立即exec而是留下了一点点“供我们修改子进程的空间”。新分配的文件描述符总是当前进程未被使用的最小描述符，这样一来，当我close掉原本的标准输出描述符“0”之后，open所自动分配的文件描述符将永远是0！这样一来，我再也不用在open中显式地给出文件描述符以供子进程调用，子进程所调用的程序也不必增添一个额外的参数处理代码段来接收文件描述符，默认为0即可。

另外，exec的机制也非常简单粗暴，完全抛弃掉现有内存中装载的程序，重新装入新的程序。

### 文件描述符

文档描述的非常详细，并没有令我疑惑或者引发我思考的地方。

阅读完这一块之后，我猜想文件描述符共享偏移的机制也许有助于实现多个进程对同一文件的读写。

### 管道

实在是第0章中最抽象，最难以理解的部分了！考察如下样例：

int p[2];

char \*argv[2];

argv[0] = "wc";

argv[1] = 0;

pipe(p);

if(fork() == 0) {

close(0);

dup(p[0]);

close(p[0]);

close(p[1]);

exec("/bin/wc", argv);

} else {

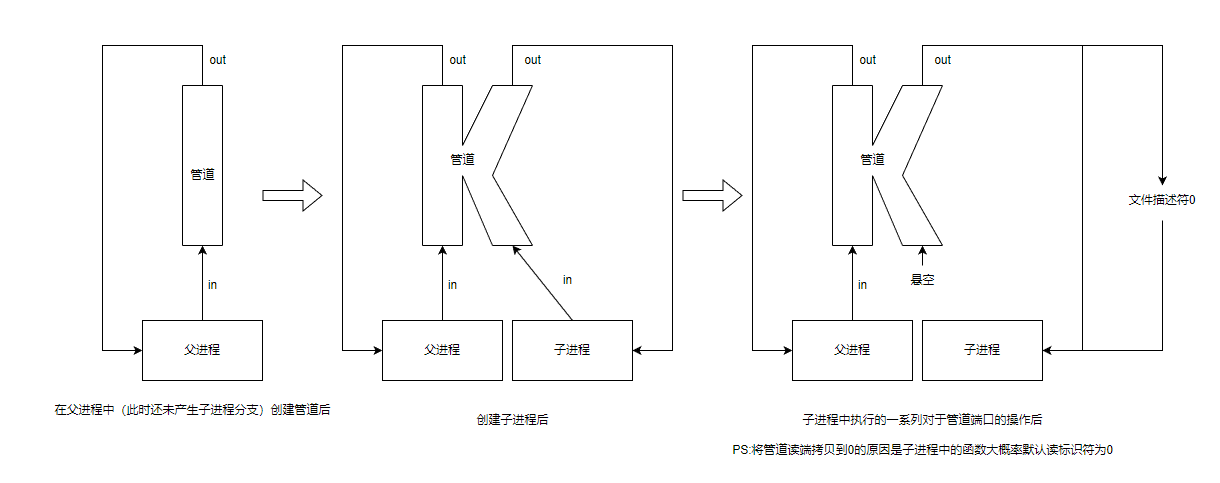
write(p[1], "hello world\n", 12);

close(p[0]);

close(p[1]);

}

我将用一系列图来详细描述发生了什么：



pipe具有阻塞的功能，也就是说，当输入端有文件标识符绑定时，输出端的程序将被挂起直到从输出端接收到数据。其余时间输出端的程序将等待输入端程序执行完毕并产生结果。这得以实现进程的同步。

注意！管道仅有在所有写端关闭（脱离标识符）并且管道内不再有任何内容的时候，会返回0（文件结束标识符）。如果子程序不打算给自己写任何内容，最好在管道建立之初就关闭子程序的写端，否则子程序的读端将永远接收不到0！

同时，当我在进行简单的pipe实验时，发现$wc wc|echo中的echo并不会生效。查看echo源码后得知：echo并不会接收任何标识符输入，echo的输入是以argv数组的形式传入的。

### 文件系统

得以了解到cd的真面目，也就是chdir（change directory）。

其他内容都是偏向于了解性质，感觉在后来的学习中可能会深入了解。

## Lab1 Utilities

### sleep

#include "kernel/types.h"

#include "user.h"

int main(int argc,char\*\* argv)

{

  int time;

  if(argc<2){

    exit(0);

  }

  time=atoi(argv[1]);

  if(time<0){

    exit(1);

  }

  sleep(time);

  exit(0);

}

为了和xv6自带的程序风格相对应，在没有参数的时候将不做任何事（而不是返回一条提示信息）。

### pipe

#include "kernel/types.h"

#include "user.h"

int main(int argc,char\*\* argv)

{

    char buf[64];

    int p1[2],p2[2];

    pipe(p1);

    pipe(p2);

    int pid=fork();

    if(pid==0){

        close(p1[1]);

        close(p2[0]);

        if(read(p1[0],buf,sizeof(buf)==-1)){

        exit(1);

        }

        close(p1[0]);

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

        write(p2[1],"pong",strlen("pong"));

        close(p2[1]);

        exit(0);

    }

    else{

        close(p2[1]);

        close(p1[0]);

        write(p1[1],"ping",strlen("pong"));

        close(p1[1]);

        if(read(p2[0],buf,sizeof(buf))==-1){

            exit(1);

        }

        close(p2[0]);

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

        wait(0);

        exit(0);

    }

}

注意！pipe的读端为0写端为1，不可调换。

起初我一直在考虑如何同步子进程与父进程的读/写。突然间我意识到pipe本身就具有同步性，当一方进程进行read操作时，如果另一方进程还没有write，读方进程就会单纯挂起，什么也不做。

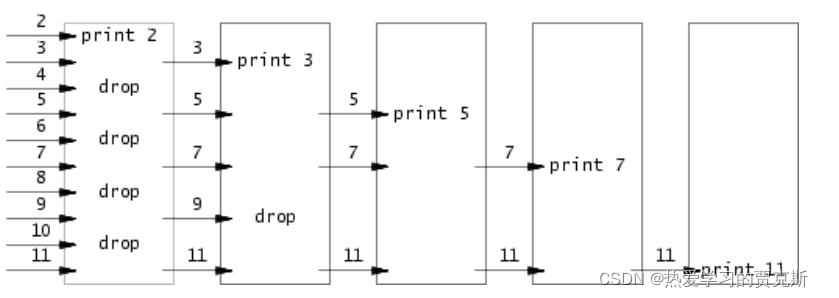
———————————————————————————————————————

还是很不习惯离开了成熟ide后的编码….两个简单exercise花了两小时。

# 2023/09/03

## Lab1 Utilities

### primes



最开始并没有什么头绪，直到我发现lab中的链接提供了这一张示意图，它的内容非常易懂，即：每一轮以这列数中最小者为因子，剔除被该因子整除的所有数，将余下数传入下一轮。这样一来，每一轮的第一个数必定是素数。

那就简单多了，附代码如下：

#include "kernel/types.h"

#include "user.h"

void childproc(int\* p1){

    close(p1[1]);

    int first;

    int p2[2];

    if(read(p1[0],&first,sizeof(int))<=0){

        close(p1[0]);

        exit(0);

    }

    printf("prime:%d\n",first);

    pipe(p2);

    if(fork()==0){

        childproc(p2);

    }else{

        close(p2[0]);

        int num;

        while(read(p1[0],&num,sizeof(int))>0){

            if(num%first!=0)write(p2[1],&num,sizeof(int));

        }

        close(p1[0]);

        close(p2[1]);

        wait(0);

    }

    exit(0);

}

int main(int argc,char\*\* argv){

    int p[2];

    pipe(p);

    if(fork()==0){

        childproc(p);

    }else{

        close(p[0]);

        for(int i=2;i<=35;i++){

        write(p[1],&i,sizeof(int));

        }

        close(p[1]);

        wait(0);

    }

    exit(0);

}

把分裂子进程的代码段封装为函数，以递归调用。

### xargs

#include "kernel/types.h"

#include "user.h"

#include "kernel/param.h"

int main(int argc,char\*\* argv){

    char buf[512];

    char\* xargv[MAXARG];

    if(argc<2)exit(0);

    for(int i=1;i<argc;i++){

        xargv[i-1]=argv[i];

    }

    while(1){

        int read\_idx=0;

        for(int i=0;i<512;i++){

            int length=read(0,&buf[i],sizeof(char));

            if(buf[i]=='\n'){

                buf[i]='\0';

                break;

            }

            else if(length<1)break;

            read\_idx++;

        }

        if(read\_idx==0)break;

        xargv[argc-1]=buf;

        if(fork()==0){

            exec(xargv[0],xargv);

            exit(0);

        }else{

            wait(0);

        }

    }

    exit(0);

}

要点在于从标准输入中读取内容，并且和xargs所执行的命令的参数进行拼接。

### find

#include "kernel/types.h"

#include "user.h"

#include "kernel/stat.h"

#include "kernel/fs.h"

char\*

fmtname(char \*path)

{

  char \*p;

  //把p移动到path最后一个斜杠后的第一个字符

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

    ;

  p++;

  return p;

}

void

find(char \*path,char\* filename)

{

  char buf[512], \*p;

  int fd;

  struct dirent de;

  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\_DEVICE:

  case T\_FILE:

    if(strcmp(fmtname(path),filename)==0){

        printf("%s %s\n", path);

    }

    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||strcmp(de.name,".")==0||strcmp(de.name,"..")==0)

        continue;

      memmove(p, de.name, DIRSIZ);

      p[DIRSIZ] = 0;

      find(buf,filename);

    }

    break;

  }

  close(fd);

}

int

main(int argc, char \*argv[])

{

  if(argc < 3){

    exit(0);

  }

  find(argv[1],argv[2]);

  exit(0);

}

find和ls.c非常相似。在copy了ls.c的代码后，对main进行了一些修改以适应功能；去掉了输出文件信息的代码；去掉了fmtname中规格化输出的代码；当当前文件是文件夹时，逐行扫描文件夹内容(正如ls.c所做的那样)，但不同的是，对于每一行内容，都将其拼接到已有路径末尾，再一次进行find，这样就可以做到搜索所有”可疑”文件了！

另外，在设计文件系统的程序中，包含非常多的指针操作，做这一块的程序的时候应该勤画图。

-------------------------------------------------------------------------------

### 总结：

对ls.c的阅读让我得以一窥xv6的文件系统，对于文件夹而言，文件夹更像是一个文本，每一行都对应着文件夹所包含的一个内容。

筛素数向我展示了一种全新的寻找素数的方法，我认为管道是一个非常具有特征，非常强大的功能。