sleep(easy)

> Implement the UNIX program `sleep` for xv6;
>
> your `sleep` should pause for a user-specified number of ticks.
>
> A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip.
>
> Your solution should be in the file `user/sleep.c` .

第一个程序比较简单，就是简单的包装一下系统调用 `sys_sleep()` ,这个程序主要是为了熟悉一下做lab的流程:

1. 创建 `user/sleep.c`
2. 在 `Makefile` 中 `UPROGS` 下添加 `$U/_sleep\`
3. 编写 `sleep.c`
4. 运行测试程序 `grade-lab-util` 并且指明要测试的函数
   如 `grade-lab-util sleep`

成功则会显示如下信息

```
== Test sleep, no arguments == fatal: bad config line 1 in file /home/ubuntu/.gitconfig
sleep, no arguments: OK (8.5s)
== Test sleep, returns == sleep, returns: OK (3.4s)
== Test sleep, makes syscall == sleep, makes syscall: OK (3.1s)
```

sleep.c：

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(2, "Usage: %s <num> \n",argv[0]);
    exit(1);
  }
  if (sleep(atoi(argv[1])) < 0) {
    fprintf(2, "Sleep error\n");
  }
  exit(0);
}
```

pipe(esay)

> Write a program that uses UNIX system calls to "ping-pong" a byte between two processes over a pair of pipes, one for each direction.
>
> The parent should send a byte to the child; the child should print ": received ping", where is its process ID, write the byte on the pipe to the parent, and exit; the parent should read the byte from the child, print ": received pong", and exit.
>
> Your solution should be in the file `user/pingpong.c` .

值得一提的是，从原则上讲，对于执行任何系统调用都要去检查是否成功，但是我偷懒了没做。

对于 `pipe(fd[2])` 来说,可能实际是这样的:

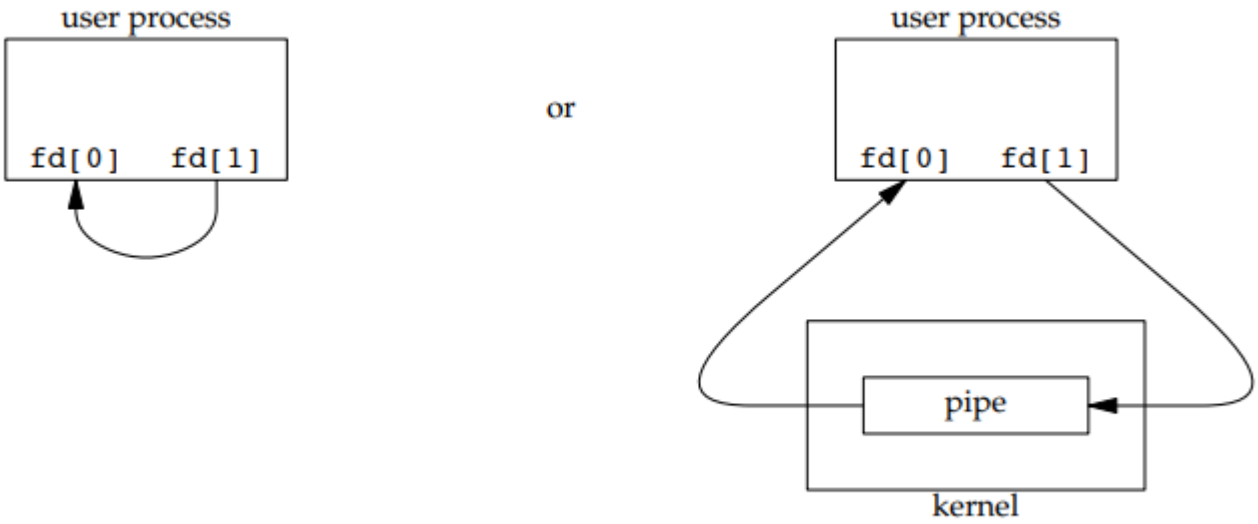


Figure 15.2 Two ways to view a half-duplex pipe

很明显这没什么用。但是如果我们使用了 `fork()` ,那么就会变得有意思起来。

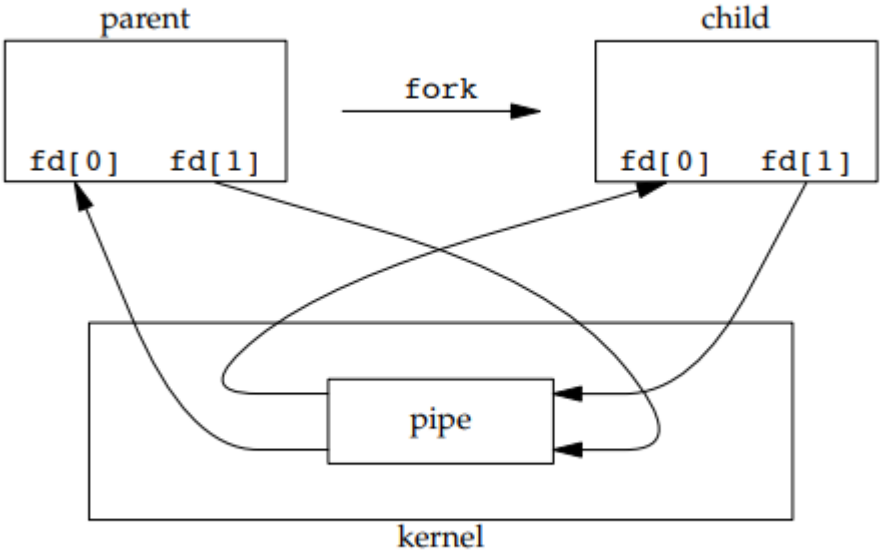


Figure 15.3 Half-duplex pipe after a `fork`

在此之后，我们只需要将我们不需要的那部分 `close()` ,就可以获得一个我们所需要的 `pipe` 。下面程序的

23,24与32,33行做的就是这样的工作。

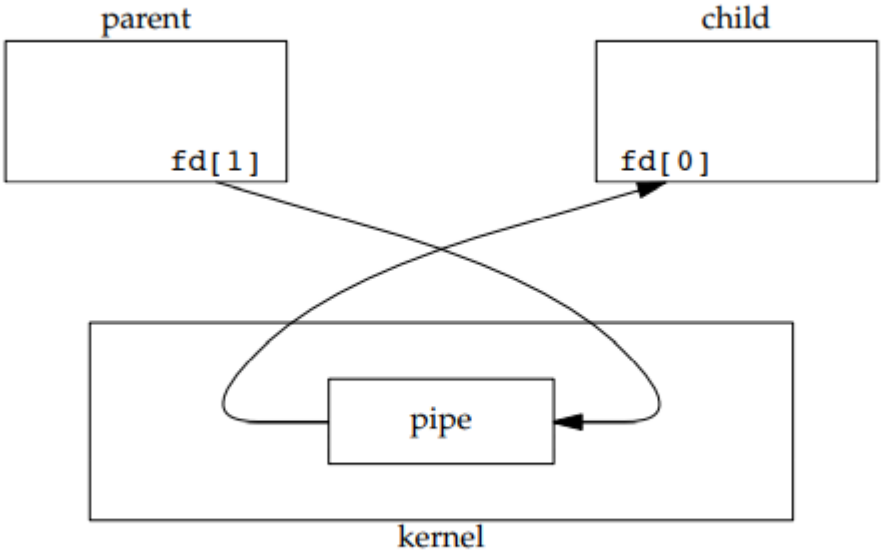


Figure 15.4 Pipe from parent to child

```c
#include "kernel/types.h"
#include "user/user.h"
#define PIPE_RD 0
#define PIPE_WR 1

int main(int argc,char *argv[]){
        if (argc > 1) {
                fprintf(2, "Usage: %s\n",argv[0]);
                exit(1);
        }
        int pfd[2]; /*  perant process -> child process*/
        int cfd[2]; /*  perant process <- child process*/

        char buf[4];

        pipe(pfd);
        pipe(cfd);

        int pid;
        pid = fork();

        if (pid == 0){
                close(pfd[PIPE_WR]);
                close(cfd[PIPE_RD]);

                if (read(pfd[PIPE_RD],buf,1) == 1)
                        printf("%d: received ping\n",getpid());
                write(cfd[PIPE_WR],"p",1);
                exit(0);
        }
        else{
                close(pfd[PIPE_RD]);
                close(cfd[PIPE_WR]);

                write(pfd[PIPE_WR],"p",1);
                if (read(cfd[PIPE_RD],buf,1) == 1)
                        printf("%d: received pong\n",getpid(),buf);
        }
        exit(0);
}
```

find(moderate)

> Write a simple version of the UNIX find program: find all the files in a directory tree with a specific name. Your solution should be in the file `user/find.c` .
>
> - Don't recurse into "." and "..".

find的基本结构与ls差不多，可以仿照的先制作一个，下方是两个主要的不同点。

对于

```
// Return blank-padded name.*
if(strlen(p) >= DIRSIZ)
  return p;
 memmove(buf, p, strlen(p));
 memset(buf+strlen(p), ' ', DIRSIZ-strlen(p));
 return buf;
// 如果p的长度大于DIRSIZ,则直接返回p,p的长度小于DIRSIZ则在p的文件名后面补齐空格
// 在find里面，补齐空格会导致与findname的比较很难写，所以直接去掉就好了
```

对于第二块差别

```
case T_FILE:
      strcpy(buf, path);        //将path复制到buf里
      p = buf + strlen(buf);//p为一个指针，指向buf(path)的末尾
      *p++ = '/';       //在末尾添加/ 比如 path为 a/b/c 经过这步后变为 a/b/c/<-p
      while (read(fd, &de, sizeof(de)) == sizeof(de)) {
          //读取一个folder的字节
          //下列if的条件详见题目的要求
        if (de.inum == 0 || (strcmp(de.name, ".") == 0) || (strcmp(de.name, "..") == 0))
          continue;
          //拼接出形如 a/b/c/de.name 的新路径(buf)
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0;
          //递归查找
        find(buf, findname);
      }
      break;
```

以下是完整的 find.c

```c
#include "../kernel/types.h"
#include "../kernel/stat.h"
#include "../user/user.h"
#include "../kernel/fs.h"



char *fmtname(char *path) {
  char *p;
  for (p = path + strlen(path); p >= path && *p != '/'; p--)
    ;
  p++;
  return p;
}
void equal_print(char *path, char *findname) {
  if (strcmp(fmtname(path), findname) == 0)
    printf("%s\n", path);
}
void find(char *path, char *findname) {
  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);
    return;
  }

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

  switch (st.type) {
    case T_FILE:
      equal_print(path, findname);
      break;

    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;
        find(buf, findname);
      }
      break;
    }
  close(fd);
}

int main(int argc, char *argv[]) {
  if (argc != 3)
    fprintf(2, "Usage: %s findpath findname\n",argv[0]);
  find(argv[1], argv[2]);
  exit(0);
}
```

xargs(moderate)

> Write a simple version of the UNIX xargs program: read lines from the standard input and run a command for each line, supplying the line as arguments to the command. Your solution should be in the file `user/xargs.c` .

主要是了解xargs这个指令，编写的过程还算简单，按照提示一步一步的来就可以了。

按照在unix中的xargs，如果参数为1的情况下是默认命令为echo，但是我在这里并没有这么做，也可以通过的xv6的测试。

```c
#include "../kernel/types.h"
#include "../user/user.h"
#include "../kernel/fs.h"
#include "../kernel/param.h"
#include "../kernel/stat.h"

int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(2, "Usage: xargs <command>\n");
    exit(1);
  }
  if (argc + 1 > MAXARG) {
    fprintf(2, "Too many arguments\n");
    exit(1);
  }

  char *cmd = argv[1];
  char *params[MAXARG], buf[512];
  int i;
  /* for Child  execcmd */
  for (i = 1; i < argc; i++) {
    /* 去除argv[]中的xargs */
    params[i - 1] = argv[i];
  }
  params[argc] = 0;
  /* for Child  execcmd */

        for(;;) {
      i = 0;
      for(;;) {
      /*  从命令行读取一条指令  */
        int n = read(0, &buf[i], 1);
        if (n == 0 || buf[i] == '\n')
          break;
        i++;
      }

      if (i == 0)
        break;
      buf[i] = 0;
      params[argc - 1] = buf;
      /*将命令行输入拼接到指令末尾，之后使用子进程exec执行cmd params 直到再也无法从命令行读取输入*/
      if (fork() == 0) {
```

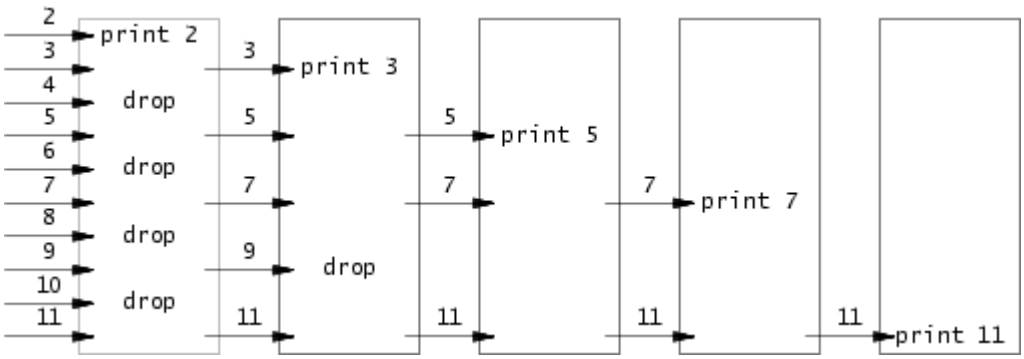
```
        exec(cmd, params);
        exit(0);
      }
      else {
        wait((int *)0);
      }
    }
  exit(0);
}
```

prime(moderate)/(hard)

> Write a concurrent version of prime sieve using pipes. This idea is due to Doug McIlroy, inventor of Unix pipes. The picture halfway down this page and the surrounding text explain how to do it. Your solution should be in the file `user/primes.c` .

最有意思的一个实验。思路可以参考下方程序

```
p = get a number from left neighbor
print p
loop:
    n = get a number from left neighbor
    if (p does not divide n)
        send n to right neighbor
```



```c
#include "../kernel/types.h"
#include "../kernel/stat.h"
#include "../user/user.h"


#define PIPE_RD 0
#define PIPE_WR 1


void
child(int pp[])
{
  close(pp[PIPE_WR]);
  int i;
  if (read(pp[PIPE_RD], &i, sizeof(i)) == 0) {
    close(pp[PIPE_RD]);
    exit(0);
  }
  printf("prime %d\n", i);
/*从左邻居中读取第一个数，并且打印*/
  int num, cp[2];
  pipe(cp);
/*将剩下的数字发送给他的右邻居*/
  if (fork() == 0) {
    close(pp[PIPE_RD]);
    close(cp[PIPE_WR]);
    child(cp);
  } else {
    close(cp[PIPE_RD]);
    while (read(pp[PIPE_RD], &num, sizeof(num)) != 0) {
      if (num % i != 0) {
        write(cp[PIPE_WR], &num, sizeof(num));
      }
    }
    close(pp[PIPE_RD]);
    close(cp[PIPE_WR]);
    wait((int *) 0);
  }
  exit(0);
}

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

```
{
  if (argc > 1) {
    fprintf(2, "Usage: primes\n");
    exit(1);
  }

  int p[2];
  pipe(p);

  if (fork() == 0) {
    child(p);
  } else {
    close(p[PIPE_RD]);
    // 构建单向管道，并且将2~35写入管道中，即最初的左邻居。
    for (int i = 2; i <= 35; i++) {
      write(p[PIPE_WR], &i, sizeof(i));
    }
    close(p[PIPE_WR]);
    wait((int *) 0);
  }

  exit(0);
}
```