# Lab: Xv6 and Unix utilities

## Boot XV6

### 实验目的：

利用qemu运行xv6

### 实验步骤：

1. 安装相应配置包

在ubuntu的终端下输入以下命令：

sudo apt update

sudo apt install build-essential gcc make perl dkms git gcc-riscv64-unknown-elf gdb-multiarch qemu-system-misc

1. 利用git获得实验环境

先克隆源码

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

进入实验目录

cd xv6-labs-2020

切换到第一个实验的分支

git checkout util

1. 启动xv6

在xv6-labs-2020目录下的终端中输入make qemu启动xv6

得到如下结果表示已经成功编译并启动xv6系统

xv6 kernel is booting

hart 2 starting

hart 1 starting

init: starting sh

### 实验中遇到的困难和解决办法：

第一步下载安装包的时候显示无法连接到对应网站，后来检查发现是没有联网，简介成功后就好了。

### 实验心得：

很简单的一个小实验，这些基本的步骤在后面的每一个实验中都要运用到。接下来就可以好好沉浸在xv6系统下的学习中了！

## sleep

### 实验目的：

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.

为 xv6 系统实现 UNIX 的 sleep 程序。sleep 程序使得当前进程暂停一定的时钟周期数，时钟周期数由用户指定。例如 sleep 100 ，则当前进程暂停，等待 100 个时钟周期后才继续执行。

### 实验步骤：

#include "kernel/types.h"

#include "user/user.h"

//argc表示命令行参数的数量，argv是一个字符串数组，其中存储了命令行参数的值

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

    if(argc < 2){//检查命令行参数的数量是否小于2

        fprintf(2, "err\n");//将字符串"err\n"输出到标准错误流（stderr）

        exit(1);

    }

    sleep(atoi(argv[1]));//将第1个参数转成整数

    exit(0);

}

熟悉一个名为sleep的系统调用，当用户输入错误时，要打印错误信息，并且使用atoi将字符串转化为整数

### 实验中遇到的困难和解决办法：

这个实验也比较简单，按照实验提示来完成没有遇到什么困难。

### 实验心得：

之前写main函数基本都是int main()，不带参数，虽然知道有带参的用法，但一下子转变过来还是有点不适应，忘了这两个参数表示什么了，搜一搜明白了一个是参数的数量一个是参数的值。

## pingpong

### 实验目的：

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 "<pid>: received ping", where <pid> 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 "<pid>: received pong", and exit. Your solution should be in the file user/pingpong.c.

使用 UNIX 系统调用编写一个程序 pingpong ，在一对管道上实现两个进程之间的通信。父进程应该通过第一个管道给子进程发送一个信息 “ping”，子进程接收父进程的信息后打印 "<pid>: received ping" ，其中是其进程 ID 。然后子进程通过另一个管道发送一个信息 “pong” 给父进程，父进程接收子进程的信息然后打印 "<pid>: received pong" ，然后退出。

### 实验步骤：

#include "kernel/types.h"

#include "user/user.h"

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

    char bite = 'x';//用来传递的字节

    int ch\_to\_fa[2], fa\_to\_ch[2];//声明了两个整型数组,用来存储管道的文件描述符

    //利用pipe系统调用，创建两个管道

    //读端是0，写端是1

    pipe(ch\_to\_fa);

    pipe(fa\_to\_ch);

    // fork系统调用

    int pid = fork();

    int exitNum = 0;

    if (pid == 0) { //子进程，要写给父

        close(fa\_to\_ch[1]);//关闭父管道写端

        close(ch\_to\_fa[0]);//关闭子管道读端

        if (read(fa\_to\_ch[0], &bite, sizeof(char)) != sizeof(char)) {//子读有错误

            printf("child read err\n");

            exitNum = 1;

        }

        else{//子进程成功读到，输出信息

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

        }

        if (write(ch\_to\_fa[1], &bite, sizeof(char)) != sizeof(char)) {//父写有错误

            printf("child write err\n");

            exitNum = 1;

        }

        exit(exitNum);

    }

    else { //父进程

        close(fa\_to\_ch[0]);//关闭父读端

        close(ch\_to\_fa[1]);//关闭子写端

        if (write(fa\_to\_ch[1], &bite, sizeof(char)) != sizeof(char)) {//父写有错误

            printf("parent write err\n");

            exitNum = 1;

        }

        if (read(ch\_to\_fa[0], &bite, sizeof(char)) != sizeof(char)) {//子读有错误

            printf("parent read err\n");

            exitNum = 1;

        }

        else {

            printf("%d: received pong\n", getpid());//父成功读，打印信息

        }

        exit(exitNum);

    }

}

`fork`是一个系统调用，用于创建一个新的进程。它是操作系统中进程管理的重要机制之一。`fork`系统调用的机制如下：

1. 当程序执行到`fork`系统调用时，操作系统会创建一个新的进程，称为子进程。子进程是原始进程的一个复制品，包括代码、数据、堆栈以及其他相关的进程控制块。

2. 新创建的子进程会拥有自己独立的进程标识符（PID），可以通过系统调用`getpid()`获取自己的PID，而通过`getppid()`获取父进程的PID。

3. 子进程会继承父进程的一些属性，如打开的文件描述符、信号处理器等。这是因为在创建子进程时，操作系统会复制父进程的内存空间给子进程，包括这些属性。

4. 子进程会从`fork`调用的返回处继续执行，但是在父进程和子进程中，`fork`的返回值是不同的。**对于父进程，`fork`会返回子进程的PID，而对于子进程，`fork`会返回0**。

5. 父进程和子进程是**并发执行**的，它们各自独立地执行代码。可以根据`fork`的返回值来判断当前进程是父进程还是子进程，从而采取不同的逻辑分支。

1. 子进程和父进程之间的主要区别是它们的PID不同，因此可以通过这个特性来区分它们的执行路径。通常在编写代码时，会在`fork`之后的代码块中使用条件语句来执行不同的逻辑。

需要注意的是，`fork`系统调用是一种比较昂贵的操作，因为它涉及到对进程的复制和资源的分配。因此，在实际编程中，应当谨慎使用`fork`，并避免在频繁调用的地方使用，以免产生过多的进程导致系统负荷过大。

创建一个父进程和一个子进程，通过创建这两个管道，父进程和子进程可以在它们之间进行双向通信。其中一个进程可以将数据写入一个管道，而另一个进程则可以从另一个管道中读取这些数据。父进程将ping发送给子进程，子进程打印自己的pid，然后子进程再把pong发给父进程，父进程打印自己的pid和相应提示信息

### 实验中遇到的问题和解决办法：

主要的问题还是fork函数生成的父子进程，具体在代码上要怎么处理。因为在课上只学习了概念，知道会分为父子两个进程，但不知道在代码上一个函数里怎么体现。后来查资料知道了其实只用把一个函数想象成两个同时在进行的函数就好了，用if-else来区分不同进程的应处理的情况。

### 实验心得：

第一次从实际工程上接触了fork()和管道的概念，真正意义上从课本的概念跳出，转而到具有实际意义的实践上。

## primes

### 实验目的：

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](http://swtch.com/~rsc/thread/) and the surrounding text explain how to do it. Your solution should be in the file user/primes.c.

使用管道将 2 至 35 中的素数筛选出来，这个想法归功于 Unix 管道的发明者 Doug McIlroy 。链接中间的图片和周围的文字解释了如何操作。最后的解决方案应该放在 user/primes.c 文件中。

目标是使用 pipe 和 fork 来创建管道。第一个进程将数字 2 到 35 送入管道中。对于每个质数，你要安排创建一个进程，从其左邻通过管道读取，并在另一条管道上写给右邻。由于 xv6 的文件描述符和进程数量有限，第一个进程可以停止在 35 。

### 实验步骤：

主进程生成2-35的数字，从左边的进程中依次拿一个数，对于每一个取到的数字，如果不能被该进程中的第一个数整除，那么就传入下一个进程。每一个进程中的第一个数就是一个质数。在实现过程中，需要维护读端和写端的管道，依次读取上一个进程写入管道的内容，并将不能被第一个数整除的数写到下一个进程。

伪代码

p = get a number from left neighbor

print p

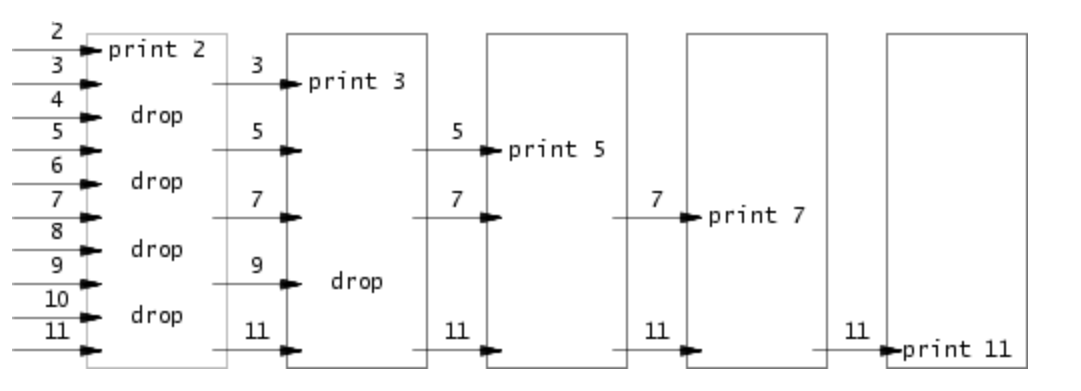
loop:

n = get a number from left neighbor

if (p does not divide n)

send n to right neighbor

图示：



#include "kernel/types.h"

#include "user/user.h"

//递归寻找素数

void primes(int lpipe[2])

{

    int firstNum, data;

    close(lpipe[1]);//关闭左管写端

    if (read(lpipe[0], &firstNum, sizeof(int)) != sizeof(int)) {//读取有误

        printf("read err\n");

        exit(1);

    }

    printf("prime %d\n", firstNum);

    //当前的管道

    int rpipe[2];

    pipe(rpipe);

    //依次读取前一个进程的数据，把不能被第一个数（质数）整除的数加入下一级进程

    while (read(lpipe[0], &data, sizeof(int)) == sizeof(int)) {// 从左管道读取数据

        if (data % firstNum != 0)// 将无法整除的数据传递入右管道

            write(rpipe[1], &data, sizeof(int));

    }

    close(lpipe[0]);

    close(rpipe[1]);

    //继续递归找质数

    if (fork() == 0) {

        primes(rpipe);

    }

    else {

        close(rpipe[0]);

        wait(0);

    }

    exit(0);

}

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

{

    int p[2];

    pipe(p);//建立管道用于进程间通信

    // 写入初始数据

    for (int i = 2; i <= 35; i++)

        write(p[1], &i, sizeof(int));

    // 针对父子进程不同的操作

    if (fork() == 0) {//对于下一级子进程，递归调用函数输出质数

        primes(p);

    }

    else {//父进程要等待子进程结束

        close(p[1]);

        close(p[0]);

        wait(0);

    }

    exit(0);

}

### 实验中遇到的问题和解决办法：

主要还是父子进程的概念到代码实现的一个脱钩。感觉自己对这个点很吃力，明明概念理解的比较清晰但是一到实践就不会了。后来开看了教学视频、舍友互助下终于写完了。

实验心得：

这个实验也不算难，主要还是把所想的概念转化为代码实现就好。

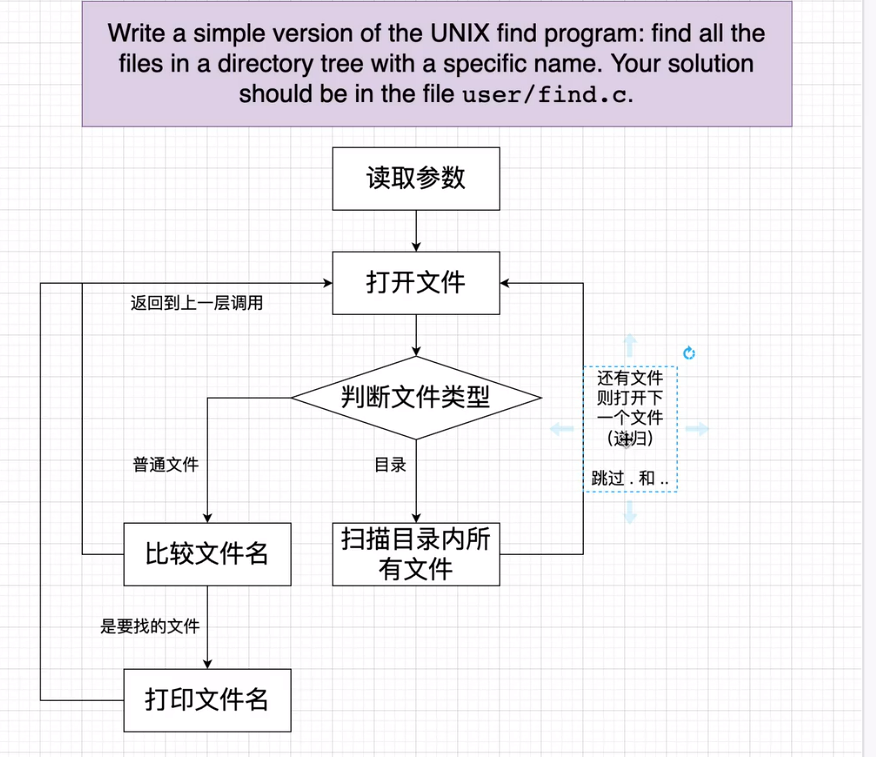
## find

### 实验目的：

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.

编写一个简单的 UNIX `find` 程序，在目录树中查找包含特定名称的所有文件。解决方案应放在 `user/find.c`

### 实验步骤：



#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/fs.h"

char\* pfilename(char \*path)

{

  char \*p;

  //找到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;

}

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

    int fd;//存储打开的目录的文件描述符

    char buf[512], \*p;//buf用于存储当前正在检查的目录或文件的路径,p用作遍历和操作buf数组

    struct stat st;//用于存储文件或目录的信息

    struct dirent de;//表示一个目录条目，用于遍历目录的内容

    if ((fd = open(path, 0)) < 0) {//如果open系统调用失败（返回负值），则会打印错误消息并退出

        fprintf(2, "open fail%s\n", path);

        exit(1);

    }

    //获取有关打开的文件或目录的信息

    if(fstat(fd, &st) < 0) {//失败就打印错误消息，关闭文件描述符，然后退出

        fprintf(2, "fstat fail%s\n", path);

        close(fd);

        exit(1);

    }

    switch(st.type) {

        case T\_FILE: //常规文件

            if(0 == strcmp(pfilename(path), filename)) {//比较文件名与指定的文件名,如果匹配就打印

                fprintf(1, "%s\n", path);//标准输出

            }

            break;

        case T\_DIR: //目录

            strcpy(buf, path);//将给定路径复制到buf中

            p = buf + strlen(buf);//将指针移动到buf的末尾

            \*p++='/';

            while(read(fd, &de, sizeof(de)) == sizeof(de)) {//读取目录的内容,将结果存储在de结构体中

                if(de.inum == 0) {//如果读取的条目的inum字段为0，表示无效条目，代码将继续下一次循环

                    continue;

                }

                //如果读取的条目的名称是当前目录（.）或上级目录（..），代码也将继续下一次循环

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

                    continue;

                }

                //拼接路径，将条目的名称复制到buf中。并通过调用stat函数获取该条目的信息存储在 st 结构体中。

                memmove(p, de.name, DIRSIZ);

                p[DIRSIZ] = 0;//在buf数组中的目录项名称的末尾添加一个空字符（\0），以确保字符串以空字符结尾，从而使其成为一个有效的C字符串

                if(stat(buf, &st) < 0) {

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

                    continue;

                }

                //递归

                find(buf, filename);

            }

            break;

    }

    //关闭文件描述符，并返回0表示函数执行成功结束

    close(fd);

    return 0;

}

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

{

    if (argc < 3) {

        fprintf(2, "not enough arguments\n");

        exit(1);

    }

    find(argv[1], argv[2]);

    exit(0);

}

这段代码实现了一个递归的文件搜索函数 find，它接受两个参数：path 表示要搜索的路径，filename 表示要查找的文件名。

主要思路如下：

首先，代码会尝试打开指定的路径，如果打开失败，则会输出错误消息并退出程序。

接下来，代码调用 fstat 函数获取打开文件或目录的信息，如果获取失败，则会输出错误消息并退出程序。

使用 switch 语句，根据文件或目录的类型执行不同的操作：

如果是一个常规文件，将比较文件名与指定的文件名，如果匹配则输出该文件的路径。

如果是一个目录，代码会遍历目录中的每个目录项，跳过无效的条目（如当前目录 . 和上级目录 ..），对于有效的条目，将构建新的路径并递归调用 find 函数进行继续搜索。

最后，关闭文件描述符，并返回0表示函数执行成功结束。

在 main 函数中，首先检查传递给程序的参数数量是否足够，如果不足，则输出错误消息并退出程序。然后，调用 find 函数开始搜索文件，并传递给它要搜索的路径和文件名。

整体上，这段代码实现了一个递归的文件搜索功能，它会遍历指定路径下的所有文件和子目录，并输出匹配的文件路径。

实验中遇到的问题和解决办法：

好久不写c语言代码了，一下子写比较复杂的函数有些语法还不适应，又回过头复习了一遍语法。

### 实验心得：

这个实验本身概念和代码上都属于中规中矩的难度，主要还是在实验之前先把概念理解清楚。

## xargs

### 实验目的：

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.

编写一个简单的 UNIX `xargs` 程序，从标准输入中读取行并为每一行运行一个命令，将该行作为命令的参数提供。解决方案应该放在 `user/xargs.c` 中。

实验步骤：

在uer目录下新建`xargs.c`，编写代码。

这段代码是一个简单的实现了类似于 Linux 命令行工具 xargs 的功能。xargs 命令可以从标准输入读取一系列参数，并将这些参数作为命令的参数执行。

代码的主要逻辑如下：

首先，声明了一个 char\* 数组 args，用于保存要执行的命令及其参数。

然后，通过 for 循环将程序的命令行参数复制到 args 数组中。

接下来，进入一个无限循环，每次循环都会读取标准输入中的一行内容（以换行符 \n 为结束符），并将其保存到 buf 字符数组中。

然后，将该行内容（作为参数）追加到 args 数组中，并将 args 数组的最后一个元素设为 0，以便作为 exec 函数的参数结束标记。

在子进程中，调用 exec 函数执行 args[1] 指定的命令，并将 args + 1 作为参数传递给该命令。如果 exec 执行成功，子进程会自动退出。

如果 exec 执行失败，子进程会打印出 "exec err" 的错误信息。

在父进程中，使用 wait 函数等待子进程执行完毕。

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/param.h"

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

{

    char\* args[MAXARG];//保存执行的参数

    //先把xargs自带的参数读进去

    int p;

    for (p = 0; p < argc; p++)

        args[p] = argv[p];

    char buf[256];

    while(1){//进入循环，每次读一行内容

        int i = 0;

        //读取标准输入一行的内容

        while((read(0,buf+i,sizeof(char)) != 0) && buf[i] != '\n')

            i++;

        if(i==0)//读完所有行

            break;

        buf[i]=0;//字符串结尾，exec要求的

        args[p]=buf;//把标准输入传进的一行参数附加到xargs这个函数后面

        args[p+1]=0;//exec读到0就表示读完了

        if(fork()==0){//子进程

            exec(args[1],args+1);//前者是xargs，后者是参数，执行成功会自动退出

            //如果执行失败，会打印以下信息

            printf("exec err\n");

        }

        else{

            wait((void\*)0);

        }

    }

    exit(0);

}

### 实验中遇到的问题和解决办法：

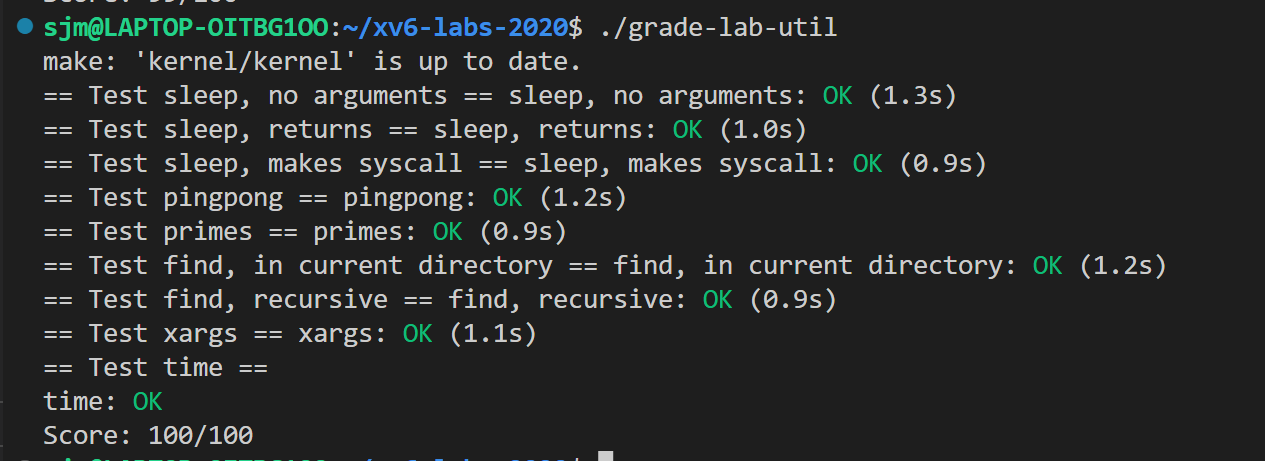
1.while(true)会报错true未声明。

在编译C语言时有时会遇到这样的错误提示：

`true' undeclared (first use in this function) or `false' undeclared (first use in this function)

就是说  bool, true, false 都是undeclared，原因很简单，因为真正的C中没有这些关键字，c和早期的c++里没有关键字bool，使用BOOL可以，但BOOL不是内置类型了，都是通过typedef或者宏来定义的，通常都会被定义成int类型。后来的c++出现了内置类型bool，值只能为true（1）和false（0）。

2.报错：control reaches end of non-void function，在最后没有加上exit(0)，编程者角度来看有些情况不可能存在，是不需要在每个控制流都要有退出返回值的，但是在实际编程中还是要在非void函数的每个出口都设置返回值。



### 实验心得：

这个实验让我进一步理解了操作系统的基本原理和内部机制，并让我对计算机系统的整体运作有了更深入的了解。我意识到操作系统是计算机系统中至关重要的组成部分。它不仅负责管理硬件资源和提供接口供应用程序使用，还承担着多任务调度、内存管理、文件系统等关键任务。学习和实践操作系统的构建过程，让我更加珍视这个底层软件系统对计算机的作用。操作系统是一个庞大而复杂的主题，通过这个实验只是迈出了学习的第一步。