**Lab 1 Xv6 and Unix utilities**

1. **Boot xv6 (**[**easy**](https://pdos.csail.mit.edu/6.828/2020/labs/guidance.html)**)//前置实验**
   1. **实验目的**

在Ubantu虚拟机上模拟xv6系统：切换到xv6-labs-2020 代码的 util 分支，并利用 QEMU 模拟器启动 xv6 系统。

* 1. **实验步骤**

配置环境：下载vmplayer，配置Linux Ubantu虚拟机，根据实验官网Tools界面引导，以管理员身份使用apt指令安装所需软件包(git、qemu、gcc、gdb、riscv编译链工具等)

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

由于qemu-system-misc的某个更新破坏了实验所用内核兼容性，故卸载新版并安装旧版qemu

$ sudo apt-get remove qemu-system-misc

$ sudo apt-get install qemu-system-misc=1:4.2-3ubuntu6

将实验所需xv6-labs-2020代码库克隆到本地，进入目录下后进入util分支下，并使用make qemu指令通过qemu模拟器启动xv6系统。

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

Cloning into 'xv6-labs-2020'...

...

$ cd xv6-labs-2020

$ git checkout util

Branch 'util' set up to track remote branch 'util' from 'origin'.

Switched to a new branch 'util'

$ make qemu

……

xv6 kernel is booting

hart 2 starting

hart 1 starting

init: starting sh

$

输入 ls 命令(用于显示指定工作目录下之内容)，即创建了一个子进程 ls ，可以看到 xv6 目录下的文件、目录及其所占字节大小。

$ ls

. 1 1 1024

.. 1 1 1024

README 2 2 2059

xargstest.sh 2 3 93

cat 2 4 24256

echo 2 5 23080

forktest 2 6 13272

grep 2 7 27560

init 2 8 23816

kill 2 9 23024

ln 2 10 22880

ls 2 11 26448

mkdir 2 12 23176

rm 2 13 23160

sh 2 14 41976

stressfs 2 15 24016

usertests 2 16 148456

grind 2 17 38144

wc 2 18 25344

zombie 2 19 22408

console 3 20 0

这些输出是mkfs包含在初始文件系统中的文件，大多数是可以运行的程序。

xv6没有ps命令(用于显示当前进程状态，类似windows的任务管理器)。按下Ctrl + p，内核将打印有关每个进程的信息，例如在刚完成make qemu后尝试结果如下。一行用于初始化，一个代表SH。

1 sleep init

2 sleep sh

退出xv6系统：在xv6中按Ctrl + a，然后按 x 即可退出 xv6 系统。

* 1. **实验中遇到的问题和解决方法**

问题：配置环境时出现安装失败及qemu无法正常识别软件库。

解决方式：更新软件包后再次下载apt。

# 列出所有可更新的软件清单命令

sudo apt update

# 升级软件包

sudo apt upgrade

* 1. **实验心得**

在成功配制出了xv6-labs-2020的实验环境，为后续实验打下基础。

1. **sleep** **(**[**easy**](https://pdos.csail.mit.edu/6.828/2020/labs/guidance.html)**)**
   1. **实验目的**

在 xv6 中实现 UNIX 程序 sleep，使其暂停执行一段用户指定的时钟滴答数。一个 tick 是 xv6 内核定义的时间概念，即来自定时器芯片的两次中断之间的时间间隔。

命令行格式: sleep <ticks>

解决方案应该在文件 user/sleep.c 中完成。

* 1. **实验步骤**
     1. 添加编译声明

Makefile(UPROGS): 编译声明

UPROGS=\

......

$U/\_sleep\

* + 1. 新增用户态sleep程序，接受参数并调用系统sleep函数

user/sleep.c

先检查传入的参数数量是否正确，若正确则调用系统sleep函数进行休眠。

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

int

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

{

//检查参数数量，小于2则报错，多于2则忽略后面的

if (argc < 2) {

fprintf(2, "Usage: sleep <ticks>\n");

exit(1);

}

//如果参数小于零则不做任何操作

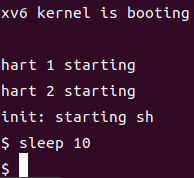
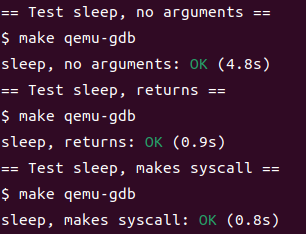
int ticks = atoi(argv[1]);

sleep(ticks);

exit(0);

}

* + 1. make测试

* 1. **实验中遇到的问题和解决方法**

问题：实验较简单，未遇到问题。

* 1. **实验心得**

成功实现了用户态sleep程序对系统sleep函数的调用，理解了从控制台获取参数的方式。

1. **pingpong** **(**[**easy**](https://pdos.csail.mit.edu/6.828/2020/labs/guidance.html)**)**
   1. **实验目的**

管道（Pipe）是一种在操作系统中用于进程间通信的机制。它允许一个进程的输出直接成为另一个进程的输入，从而实现这两个进程之间的数据传输。编写pingpong.c中的函数以实现父子进程通信，学习管道相关内容。

命令行格式: pingpong

解决方案应该在文件 user/pingpong.c 中完成。

* 1. **实验步骤**
     1. 添加编译声明

Makefile(UPROGS): 编译声明

UPROGS=\

......

$U/\_pingpong\

* + 1. 新增用户态pingpong程序，创建进程与子进程，实现其间的通信并打印出相关信息

user/pingpong.c

首先创建父进程到子进程、子进程到父进程的管道(大小为2，对应读/写)

在进入父/子进程时，关闭自己进程读入端与子/父进程写出端管道，进行读入或写出并在命令行打印相关消息。

//xv6-kabs-2020 Labs Utility

//pingpong

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

int

main(void) {

char buffer[16]; // 缓冲区

int parent\_to\_child[2]; // 父进程到子进程的管道

int child\_to\_parent[2]; // 子进程到父进程的管道

//检查管道创建是否出错

if (pipe(parent\_to\_child) < 0 || pipe(child\_to\_parent) < 0) {

fprintf(2, "Error in pingpong: Failed to create pipe.\n");

exit(1);

}

int pid = fork();

//检查进程创建是否出错

if (pid < 0) {

fprintf(2, "Error in pingpong: Fork failed.\n");

exit(1);

}

if (pid == 0) {// 子进程

close(parent\_to\_child[1]); // 关闭父进程写端

close(child\_to\_parent[0]); // 关闭子进程读端

read(parent\_to\_child[0], buffer, strlen("ping"));

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

write(child\_to\_parent[1], "pong", strlen("pong"));

close(parent\_to\_child[0]); // 关闭父进程读端

close(child\_to\_parent[1]); // 关闭子进程写端

} else {

// 父进程

close(parent\_to\_child[0]); // 关闭父进程读端

close(child\_to\_parent[1]); // 关闭子进程写端

write(parent\_to\_child[1], "ping", strlen("ping"));

wait(0);

//等到子进程退出

read(child\_to\_parent[0], buffer, strlen("pong"));

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

close(parent\_to\_child[1]); // 关闭父进程写端

close(child\_to\_parent[0]); // 关闭子进程读端

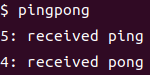
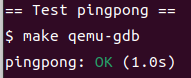
wait(0); // 等待子进程退出

}

exit(0);

}

* + 1. make测试

* 1. **实验中遇到的问题和解决方法**

问题：不清楚管道的定义与作用

解决方式：根据xv6 book与视频学习，回顾课本知识。得知管道通常用于将一个进程的输出数据传递给另一个进程进行处理，而无需将数据写入临时文件或者使用其他更复杂的通信方式。管道的主要优点是简单性和效率，适用于需要快速传递数据的情况，如通过一个进程过滤和处理另一个进程的输出。同时，管道通信是一种单向、半双工的通信方式，有些情况下可能不够灵活。在更复杂的进程间通信场景中，可能需要使用其他 IPC 机制，如消息队列、共享内存等。

* 1. **实验心得**

实现了父子进程的简单通信，更生动地理解了在课本中所学的管道方式进程间通信方式。

1. **primes (moderate)/(hard)**
   1. **实验目的**

基于埃氏筛算法，筛选出35以内的质数。与一般的算法不同的是，需要充分使用pipe和fork方法来编写primes程序。

解决方案应该在文件 user/primes.c 中完成。

* 1. **实验步骤**
     1. 添加编译声明

Makefile(UPROGS): 编译声明

UPROGS=\

......

$U/\_primes\

* + 1. 新增用户态primes程序，通过进程的迭代实现质数筛

user/primes.c

迭代：每次fork一个进程，父进程通过管道逐个传输自己筛选后的元素。而fork出的子进程从父进程中读入父进程的所有元素(即目前还被认为是质数的序列)，读入的第一个元素即当前子进程中埃氏筛将要筛去的因子。在筛选完毕后，子进程重复以上过程。

特别注意：文件描述符是循环分配的。在该程序中，每个进程中使用管道完毕后都需要及时关闭对应文件描述符，避免导致资源不足分配失败。

//xv6-kabs-2020 Labs Utility

//primes

//埃氏筛的管道方式实现

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

void

primes(int read\_fd) {//read\_fd为上一个进程的读管道文件描述符

int top;//当前输出的质数(从管道读入的第一个数)

int cur;//后续读入的数

int p[2];//管道

if (read(read\_fd, &top, sizeof(int)) > 0) {//如果还有读入

printf("prime %d\n", top);

pipe(p);

int pid = fork();

if (pid < 0) {//进程创建失败

fprintf(2, "Error in primes: Fork failed.\n");

exit(0);

} else if (pid == 0) {//子进程

close(p[0]);//子进程不需要读管道

while (read(read\_fd, &cur, sizeof(int)) > 0) {

if (cur % top != 0) {//筛选并写入管道

write(p[1], &cur, sizeof(int));

}

}

close(p[1]);

} else {//父进程

close(p[1]);//父进程不需要用到写管道

wait(0);

primes(p[0]);//将管道读端作为参数传入函数

close(p[0]);//关闭父进程读管道

}

} else {

close(read\_fd);

exit(0);

}

}

int

main(void) {

int p[2];

pipe(p);

for (int i = 2; i <= 35; ++i) {

write(p[1], &i, sizeof(int));

}

close(p[1]);

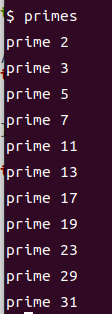
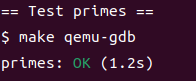
primes(p[0]);

close(p[0]);

exit(0);

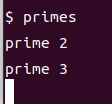
}

* + 1. make测试

* 1. **实验中遇到的问题和解决方法**

问题：进程卡死在primes 3



解决方式：复习xv6 book后，得知文件描述符是循环使用的，打开文件或创建新文件描述符时，操作系统会分配一个最小的文件描述符数值。关闭描述符时，该数值才被释放。而xv6中可用值只有0~15，共16个。

文件描述符 0: 标准输入（stdin）

文件描述符 1: 标准输出（stdout）

文件描述符 2: 标准错误输出（stderr）

文件描述符 3-15: 用户打开的文件。

在该程序中，每个进程中使用管道完毕后都没有关闭对应文件描述符，导致资源不足卡死。

else if (pid == 0) {//子进程

close(p[0]);//子进程不需要读管道

while (read(read\_fd, &cur, sizeof(int)) > 0) {

if (cur % top != 0) {//筛选并写入管道

write(p[1], &cur, sizeof(int));

}

}

close(p[1]);

} else {//父进程

close(p[1]);//父进程不需要用到写管道

wait(0);

primes(p[0]);//将管道读端作为参数传入函数

close(p[0]);//关闭父进程读管道

* 1. **实验心得**

在pingpong的基础上更加深入了解了管道与文件描述符的概念与作用，并理解到了及时释放无用资源是操作系统优化中很重要的一件事。

1. **find (moderate)**
   1. **实验目的**

参考ls.c，完成查找(指定目录下，默认为.)指定文件并打印所有文件路径的程序find。

命令行格式: find <filename> / find <dirname> <filename>

解决方案应该在文件 user/find.c 中完成。

* 1. **实验步骤**
     1. 添加编译声明

Makefile(UPROGS): 编译声明

UPROGS=\

......

$U/\_find\

* + 1. 新增用户态find程序，复制部分ls.c内容并进行修改以实现功能。

user/find.c

//xv6=labs-2020 Labs Utilities

//find

//参考ls

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/fs.h"//文件系统相关头文件,使用相关结构体

fmtname() 函数接受一个路径字符串，并从中提取文件名。它从路径末尾向前查找直到遇到 / 字符，然后返回指向文件名的指针。

char\*

fmtname(char \*path)//ls中的函数(更改过),将文件转化为文件名,(例如将"./echo"转化为"echo",find的path就是前者

{

char \*p;

// Find first character after last slash.

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

;

p++;

return p;

}

find() 函数接受两个参数：path 表示要搜索的目录路径，fileName 表示要查找的文件名。

先打开 path 对应的文件（目录），使用 fstat() 函数获取文件的状态信息，存储在 st 结构体中。（包括基本的错误判断处理）

void

find(char \*path, char \*fileName) {//path为当前目录,fileName为文件名

int fd;

struct dirent de;//目录项结构体,包括文件名(大小为DIRSIZ=14的char数组)与inum(inode号,类型为ushort)

struct stat st;//文件信息结构体(包括文件类型/大小等)

char buf[512];

char \*p;

if ((fd = open(path, 0)) < 0) {

fprintf(2, "Error in find: Failed to open %s.\n", path);

return;

}

if (fstat(fd, &st) < 0) {

fprintf(2, "Error in find: Failed to stat %s.\n", path);

close(fd);

return;

}//读取文件及其信息

根据文件类型 st.type 进行不同的操作：

如果是文件（T\_FILE），比较文件名是否与指定的 fileName 相同，如果相同则输出文件路径。

如果是目录（T\_DIR），则进入目录，递归地搜索每个目录项。

若目录项无效（de.inum=0）或为特殊目录名(.和..)

使用 memmove() 将目录项的文件名复制到 buf 中，并在末尾添加 /。

递归调用 find() 函数，传递更新后的路径 buf。

switch (st.type) {//判断文件类型

case T\_FILE://如果是文件,则比较

if (!strcmp(fileName, fmtname(path))) {

printf("%s\n", path);

}

break;

case T\_DIR://如果是目录,则进入下级目录搜索

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

fprintf(2, "Error in find: path too long\n");

break;

}

strcpy(buf, path);

p = buf + strlen(buf);

\*p++ = '/';//把当前目录复制到buf中并且末尾添加'/'

// 读取目录中的每一项

while (read(fd, &de, sizeof(de)) == sizeof(de)) {

if (!de.inum || !strcmp(de.name, ".") || !strcmp(de.name, "..")) {//不递归"."和".."(inum=0是无效目录项)

continue;

}

memmove(p, de.name, DIRSIZ);

p[DIRSIZ] = 0;

find(buf, fileName);

}

break;

}

关闭打开的文件描述符。

close(fd);

}

main() 函数解析命令行参数：

如果只提供了一个参数，则从当前目录开始搜索文件名为参数值的文件。

如果提供了两个参数，则使用第一个参数作为起始目录，搜索文件名为第二个参数值的文件。

在搜索完成后，程序通过 exit() 终止运行。

int

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

if (argc < 2 || argc > 3) {

fprintf(2, "Usage: find <path> <fileName>\n");

exit(1);

} else if(argc == 2){

find(".", argv[1]);

} else{

find(argv[1], argv[2]);

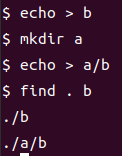
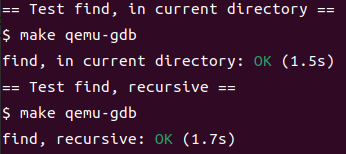
}

exit(0);

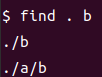
}

* + 1. make测试

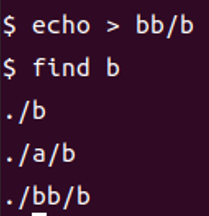
实验手册指定测试及make grade结果：

追加测试：

1、

2、

3、 

* 1. **实验中遇到的问题和解决方法**

问题：初次编写find函数时递归混乱，缺少错误判断机制，无法实现功能。

解决方式：画流程图，理清思路，必要时参考示例代码如ls.c和他人代码（例如调用不熟悉的系统函数时）学习思路、惯用套路及代码规范。

* 1. **实验心得**

由于之前操作系统实现文件系统的大作业，我在完成这个实验的时候能够充分联想之前的关于目录与文件系统的知识，能够较好地完成代码，完成程序后还能进行自己追加测试，发现网络上部分现存实现代码的错误之处。（例如某篇CSDN高浏览量文章中的代码能够完成实验指定的测试并且满分，但在追加测试中出现错误。详细见xargs实验遇到的问题部分）。

1. **xargs (moderate)**
   1. **实验目的**

读取标准输出中的信息，然后将其作为标准输入，拼接到下一个命令后面来执行。(可以将一个程序的输出拼接到另一个程序的输入)

命令行格式: xargs <arg>...//设置最高为32个参数

解决方案应该在文件 user/xargs.c 中完成。

* 1. **实验步骤**
     1. 添加编译声明

Makefile(UPROGS): 编译声明

UPROGS=\

......

$U/\_xargs\

* + 1. 新增用户态find程序，复制部分ls.c内容并进行修改以实现功能。

user/find.c

//xv6=labs-2020 Labs Utilities

//find

//参考ls

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/fs.h"//文件系统相关头文件,使用相关结构体

fmtname() 函数接受一个路径字符串，并从中提取文件名。它从路径末尾向前查找直到遇到 / 字符，然后返回指向文件名的指针。

char\*

fmtname(char \*path)//ls中的函数(更改过),将文件转化为文件名,(例如将"./echo"转化为"echo",find的path就是前者

{

char \*p;

// Find first character after last slash.

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

;

p++;

return p;

}

find() 函数接受两个参数：path 表示要搜索的目录路径，fileName 表示要查找的文件名。

先打开 path 对应的文件（目录），使用 fstat() 函数获取文件的状态信息，存储在 st 结构体中。（包括基本的错误判断处理）

void

find(char \*path, char \*fileName) {//path为当前目录,fileName为文件名

int fd;

struct dirent de;//目录项结构体,包括文件名(大小为DIRSIZ=14的char数组)与inum(inode号,类型为ushort)

struct stat st;//文件信息结构体(包括文件类型/大小等)

char buf[512];

char \*p;

if ((fd = open(path, 0)) < 0) {

fprintf(2, "Error in find: Failed to open %s.\n", path);

return;

}

if (fstat(fd, &st) < 0) {

fprintf(2, "Error in find: Failed to stat %s.\n", path);

close(fd);

return;

}//读取文件及其信息

根据文件类型 st.type 进行不同的操作：

如果是文件（T\_FILE），比较文件名是否与指定的 fileName 相同，如果相同则输出文件路径。

如果是目录（T\_DIR），则进入目录，递归地搜索每个目录项。

若目录项无效（de.inum=0）或为特殊目录名(.和..)

使用 memmove() 将目录项的文件名复制到 buf 中，并在末尾添加 /。

递归调用 find() 函数，传递更新后的路径 buf。

switch (st.type) {//判断文件类型

case T\_FILE://如果是文件,则比较

if (!strcmp(fileName, fmtname(path))) {

printf("%s\n", path);

}

break;

case T\_DIR://如果是目录,则进入下级目录搜索

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

fprintf(2, "Error in find: path too long\n");

break;

}

strcpy(buf, path);

p = buf + strlen(buf);

\*p++ = '/';//把当前目录复制到buf中并且末尾添加'/'

// 读取目录中的每一项

while (read(fd, &de, sizeof(de)) == sizeof(de)) {

if (!de.inum || !strcmp(de.name, ".") || !strcmp(de.name, "..")) {//不递归"."和".."(inum=0是无效目录项)

continue;

}

memmove(p, de.name, DIRSIZ);

p[DIRSIZ] = 0;

find(buf, fileName);

}

break;

}

关闭打开的文件描述符。

close(fd);

}

main() 函数解析命令行参数：

如果只提供了一个参数，则从当前目录开始搜索文件名为参数值的文件。

如果提供了两个参数，则使用第一个参数作为起始目录，搜索文件名为第二个参数值的文件。

在搜索完成后，程序通过 exit() 终止运行。

int

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

if (argc < 2 || argc > 3) {

fprintf(2, "Usage: find <path> <fileName>\n");

exit(1);

} else if(argc == 2){

find(".", argv[1]);

} else{

find(argv[1], argv[2]);

}

exit(0);

}

* + 1. make测试

先自定义部分运行代码测试运行实验手册的示例：

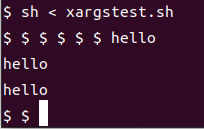
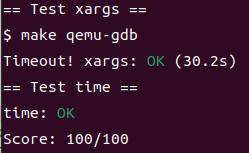
首先在b中写入hello(包括尾零)，运行ls命令，确认b的大小已经变为6字节，即成功写入hello。



然后运行示例，能够找到对应hello



make clean后重新make qemu，进行实验手册指定测试，以及make grade结果：

追加：修改echo内容/修改xargstest.sh追加不同目录、多级目录及文件、echo入不同文本等。例如将echo改为写入worldhello，

ps：这部分就是查出网上个别代码出错的地方，若grep的参数为hello，全部输出为hello或全部为worldhello，分析后发现是其程序的文件路径与文件名转化应用出错，导致grep只会重复三次搜寻根目录下的b文件中的文本而没有找a/b与c/b。如果采用原有test中几个文件全部写入hello的方式则无法查出此错误。

xargstest.sh

mkdir a

echo hello > a/b

mkdir c

echo hello > c/b

echo hello > b

find . b | xargs grep hello





* 1. **实验中遇到的问题和解决方法**

问题：出于好奇，完成实验后为他人代码debug时困扰

解决方式：画流程图，理清思路，必要时参考示例代码如ls.c和他人代码（例如调用不熟悉的系统函数时）学习思路、惯用套路及代码规范。

* 1. **实验心得**

在该实验中，我不仅完成了实验内容的测试，同上一个实验一样，我还追加了一些测试以保证在实验外的内容中我的代码也能够正常运行而不是因为巧合没有体现出错误。

此外，还检查出了能够成功完成实验的、看似正确的现存代码的错误，有一点成就感，但不多。