Lab 1 Xv6 and Unix utilities

目录

[1. Boot xv6(easy) 1](#_Toc174490752)

[1.1 实验目的 1](#_Toc174490753)

[1.2 实验步骤 1](#_Toc174490754)

[1.3 实验中遇到的问题和解决方法 3](#_Toc174490755)

[1.4 实验心得 4](#_Toc174490756)

[2. Sleep (easy) 4](#_Toc174490757)

[2.1 实验目的 4](#_Toc174490758)

[2.2 实验步骤 5](#_Toc174490759)

[2.3 实验中遇到的问题和解决方法 7](#_Toc174490760)

[2.4 实验心得 7](#_Toc174490761)

[3. Pingpong (easy) 8](#_Toc174490762)

[3.1 实验目的 8](#_Toc174490763)

[3.2 实验步骤 8](#_Toc174490764)

[3.3 实验结果 9](#_Toc174490765)

[3.4 遇到的问题 10](#_Toc174490766)

[3.5 实验心得 12](#_Toc174490767)

[4. Primes(moderate) 12](#_Toc174490768)

[4.1 实验目的 12](#_Toc174490769)

[4.2 实验步骤 12](#_Toc174490770)

[4.3 实验结果 15](#_Toc174490771)

[4.4 实验中遇到的问题和解决方法 16](#_Toc174490772)

[4.5 实验心得 16](#_Toc174490773)

[5. Find(moderate) 17](#_Toc174490774)

[5.1 实验目的 17](#_Toc174490775)

[5.2 实验步骤 17](#_Toc174490776)

[5.3 实验结果 20](#_Toc174490777)

[5.4 实验中遇到的问题和解决办法 21](#_Toc174490778)

[5.5 实验心得 22](#_Toc174490779)

[6. Xargs（moderate） 22](#_Toc174490780)

[6.1 实验目的 22](#_Toc174490781)

[6.2 实验步骤 22](#_Toc174490782)

[6.3 实验结果 25](#_Toc174490783)

[6.4 实验中遇到的问题和解决办法 25](#_Toc174490784)

[6.5 实验心得 26](#_Toc174490785)

[7. 测试结果 27](#_Toc174490786)

Boot xv6(easy)

* 1. 实验目的
* 学习如何使用基于Unix的简单操作系统xv6及其系统调用。
* 了解Git的一些基本指令和用法。
* 学会xv6的基本使用方法。
* 熟悉xv6的相关内容，掌握如何获取实验资源以及如何在Linux终端启动xv6。
  1. 实验步骤
* 更新软件包列表

$ sudo apt-get update

* 安装qemu、gcc、GDB、git等依赖项

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

* 克隆xv6实验的Git仓库到本地，并进入该目录

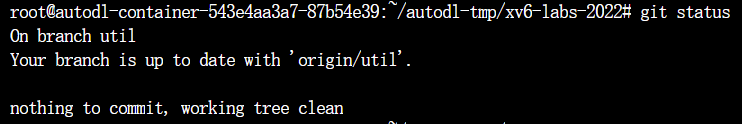
$ git clone git://g.csail.mit.edu/xv6-labs-2022

$ cd xv6-labs-2022

* 检查状态

$ git status

输出应为：



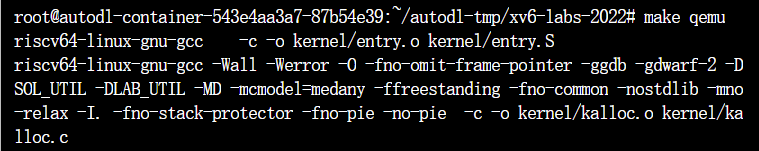
此外，可以通过查看日志来发现一些与旧版本的差异：

$ git log

* 在RISC-V模拟器qemu上构建并运行xv6

$ make qemu

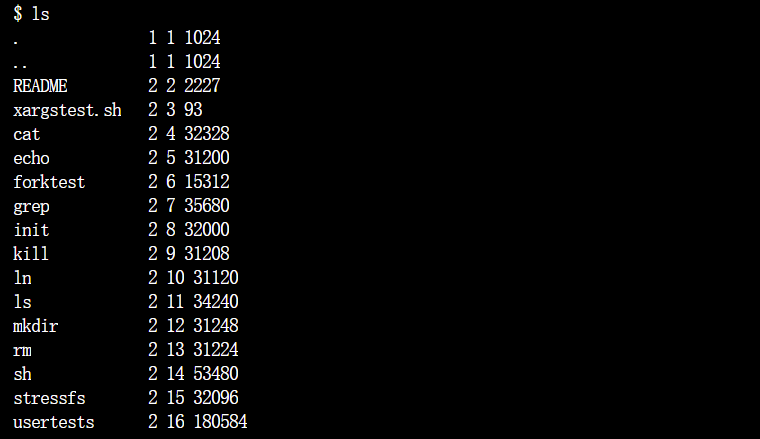
* 运行make qemu 命令后，将进行编译和链接操作，最终启动qemu模拟器。运行成功应该看到类似如下的显示：



······



* 在xv6的shell钟输入命令，如ls、echo和Ctrl-p，来探索xv6的文件系统和shell。使用ls命令查看示例：



* 按下Ctrl+a后再按下x键以退出QEMU模拟器。
  1. 实验中遇到的问题和解决方法
* 问题：

在尝试通过 apt-get 命令安装软件包时，出现了一些问题。具体错误信息如下：

E: Unable to locate package gdb-multiarch

E: Unable to locate package qemu-system-misc

E: Unable to locate package gcc-riscv64-linux-gnu

E: Unable to locate package binutils-riscv64-linux-gnu

* 问题分析：

从错误信息中可以看出，系统无法找到 gdb-multiarch、qemu-system-misc、gcc-riscv64-linux-gnu 和 binutils-riscv64-linux-gnu 这几个软件包。可能的原因包括：软件包名错误或拼写错误。软件源配置不正确或不完整。需要的软件包不在默认的软件源中。

* 解决方法：

为了解决上述问题，我采取了以下步骤：

* 更新软件包列表:

$ sudo apt-get update

* 安装必要的软件源：

$ sudo add-apt-repository universe

* 确认没有拼写错误后再次尝试安装软件包：

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

* 1. 实验心得

在此次实验中，我的主要目标是搭建 RISC-V 架构下的操作系统开发环境。通过执行make qemu命令，我成功地启动了xv6，能够再qemu窗口中看到操作系统的输出。这让我对xv6的运行环境有了基本的了解。

Sleep (easy)

* 1. 实验目的
* 在xv6系统中实现UNIX程序sleep。
* 查看学习系统中已有的源代码，了解xv6中如何传递命令行参数的。
* 熟悉项目的文件结构和逻辑关系。
  1. 实验步骤
* 阅读xv6文档书的第1章。
* 查看user/目录中的其他一些程序（例如user/echo.c和user/grep.c），以了解如何获取传递给程序的命令行参数。
* 在kernel文件夹下找到sysproc.c源文件，找到其中的sys\_sleep函数并阅读，本实验将调用该函数来实现。编写一个用户程序sleep，sleep应该暂停用户指定的滴答数。滴答是xv6内核定义的时间概念，即来自计时器芯片的两次中断之间的时间。
* 编写sleep程序的代码，创建user/sleep.c并在其中实现，代码如下：

#include "kernel/types.h" // 包含内核类型定义

#include "user.h" // 包含用户库函数

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

// 检查传递给程序的参数个数是否为 2（包括程序名称本身）

if (argc != 2) {

// 如果不是，则打印一条错误信息和一条使用信息

fprintf(2, "Error: sleep requires 1 argument\nRight Usage: sleep ticks\n");

// 然后以非零状态码退出

exit(1);

}

// 使用 atoi 函数将传递给程序的第一个参数（argv[1]）转换为整数

int ticks = atoi(argv[1]);

// 调用 sleep 系统调用，使程序休眠 ticks 个时钟周期

sleep(ticks);

// 调用 exit 函数以零状态码退出程序

exit(0);

}

* 将sleep程序添加到Makefile的UPROGS列表中，这样运行make qemu命令时将编译新增的sleep程序，修改后如下：

UPROGS=\

$U/\_cat\

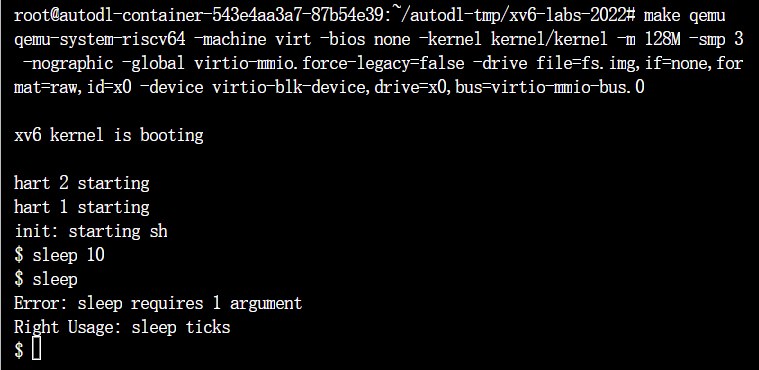
$U/\_echo\

······

$U/\_zombie\

$U/\_sleep\ # 新增的sleep函数

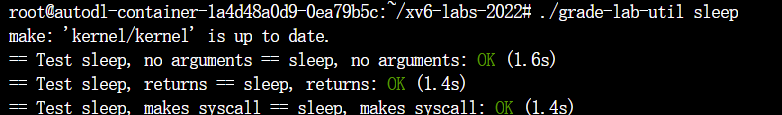
* 在xv6 shell中运行程序进行测试，结果如下：



* 运行以下命令以进行grade:

./grade-lab-util sleep

结果如下：



* 1. 实验中遇到的问题和解决方法
* 问题：

在使用 QEMU 模拟器运行 xv6 操作系统时，尝试执行 sleep 命令时出现错误，提示 exec sleep failed。这个问题导致无法正常使用 sleep 程序。

* 解决方法：

要解决这个问题，需要修改 xv6 的 Makefile 文件，确保 sleep 程序被正确编译和包含在可执行文件列表中。具体步骤如下：

1. 打开 xv6 源代码目录中的 Makefile 文件。
2. 找到 UPROGS 变量定义的部分。这个变量包含了所有需要编译并包含在 xv6 中的用户程序列表。
3. 将 sleep 程序的名称添加到 UPROGS 列表中。
4. 在该列表中添加 \_sleep\。
5. 保存并关闭 Makefile 文件
6. 重新编译 xv6 系统以应用更改

完成上述步骤后， sleep 程序应当被正确编译并包含在 xv6 系统中，再次运行应不会出现 exec sleep failed 的错误提示。

* 1. 实验心得

这次实验让我更加深入地理解了系统调用的工作原理。系统调用是用户程序与内核交互的重要接口，通过系统调用，用户程序可以请求内核提供各种服务。在实现这些服务时，内核中有相应的处理函数，如 sys\_sleep，但是用户程序不能直接调用这些内核函数，而是要通过封装在库函数中的系统调用接口来间接调用。

这个问题提醒我，在编写操作系统相关的程序时，要特别注意系统调用的使用方式，理解系统调用的机制和流程。在未来的学习和实验中，我会更加注重查阅和理解相关资料，避免类似的错误，提高编程能力和解决问题的能力。

Pingpong (easy)

* 1. 实验目的
* 在xv6系统中实现UNIX程序pingpong
* 了解父子进程的关系，学会从父进程向子进程传递数据
* 了解管道的作用，并学会运用
* 学习如何在两个进程之间建立通信通道
* 了解如何使用fork系统调用来创建一个子进程
* 学习如何在程序中获取当前进程的进程ID（PID）
  1. 实验步骤

在该程序中，需要创建两条管道，一个是父进程到子进程的，一个是子进程到父进程的。并实现父子进程之间的通信。

* 创建user/pingpong.c
* 在Makefile中的UPROGS下添加 $U/\_pingpong\
* 编写pingpong.c，代码如下：

#include "kernel/types.h" // 引入内核定义的类型，如 int, char 等

#include "kernel/stat.h" // 引入内核定义的状态码，如 T\_DIR, T\_FILE 等

#include "user/user.h" // 引入用户程序可用的函数，如 printf, fork, pipe 等

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

{

int p2c[2], c2p[2]; // 定义两个整数数组，分别表示父进程到子进程和子进程到父进程的管道

char buf; // 定义一个字符变量，用于存储从管道中读取或写入的数据

// 检查命令行参数的数量

if (argc != 1) {

fprintf(2, "Error: pingpong requires 0 argument\nRight Usage: %s\n", argv[0]);

exit(1);

}

// 创建两个管道，每个管道有两个文件描述符，分别表示读端和写端

pipe(p2c); // p2c[0] 是读端，p2c[1] 是写端

pipe(c2p); // c2p[0] 是读端，c2p[1] 是写端

if (fork() == 0) { // 创建一个子进程，并判断是否是子进程

// 子进程

read(p2c[0], &buf, 1); // 从父进程到子进程的管道的读端读取一个字节的数据，并存储到 buf 中

printf("%d: received ping\n", getpid()); // 打印当前进程的 ID 和收到的消息

write(c2p[1], &buf, 1); // 将 buf 中的数据写入到子进程到父进程的管道的写端

exit(0); // 结束子进程，并返回 0 表示正常退出

}

else {

// 父进程

write(p2c[1], "p", 1); // 将字符 'p' 写入到父进程到子进程的管道的写端

read(c2p[0], &buf, 1); // 从子进程到父进程的管道的读端读取一个字节的数据，并存储到 buf 中

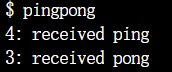
printf("%d: received pong\n", getpid()); // 打印当前进程的 ID 和收到的消息

exit(0); // 结束父进程，并返回 0 表示正常退出

}

}

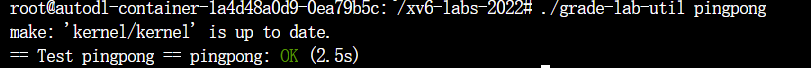
* 1. 实验结果
* 在xv6 shell中运行程序进行测试，结果如下：



* 运行以下命令以进行grade:

./grade-lab-util pingpong

结果如下：



* 1. 遇到的问题
* 问题：

新接触的到一些函数不会使用，如：fork函数、read函数、pipe函数。

* 解决方法：

查阅相关资料，了解函数功能、参数含义

* read( )

read( )是一个用于从文件、管道或其他输入流中读取数据的系统调用。它的函数定义如下：ssize\_t read(int fd, void \*buf, size\_t count);

其中， fd是要读取数据的文件描述符， buf是一个指向缓冲区的指针，用于存储读取到的数据，count是要读取的字节数。

read( )函数会尝试从fd指定的文件或输入流中读取最多count个字节的数据，并将其存储到 buf 指向的缓冲区中。它返回实际读取到的字节数。如果到达文件末尾或输入流结束，read( )会返回0。如果发生错误， read( )会返回-1，并设置全局变量errno来指示错误原因。

read( )函数会从文件描述符当前指向的位置开始读取数据，读取完毕后，文件描述符会自动向后移动相应的字节数。

* fork( )

fork( )是一个用于创建新进程的系统调用。它通过复制当前进程来创建一个新的子进程。子进程几乎与父进程完全相同，但它们是两个独立的进程，可以独立运行和执行不同的任务。

fork( )函数的返回值用于区分父进程和子进程。在父进程中，fork( )返回新创建的子进程的进程ID；在子进程中，fork( )返回0。

* pipe( )

pipe( )函数用于在两个进程之间创建一个单向的管道。它接受一个大小为2的整型数组作为参数，用于存储管道的两个文件描述符。pipe( )函数会在数组中返回两个文件描述符，其中fd[0]用于读取数据，fd[1]用于写入数据。

管道：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。

* 问题：

不知道如何从父进程向子进程传递数据

* 解决方法：

使用管道进行进程间通信。通过创建两个管道，一个用于父进程向子进程传递数据，另一个用于子进程向父进程传递数据。父进程使用管道的写端写入数据，子进程使用管道的读端读取数据。

* 1. 实验心得

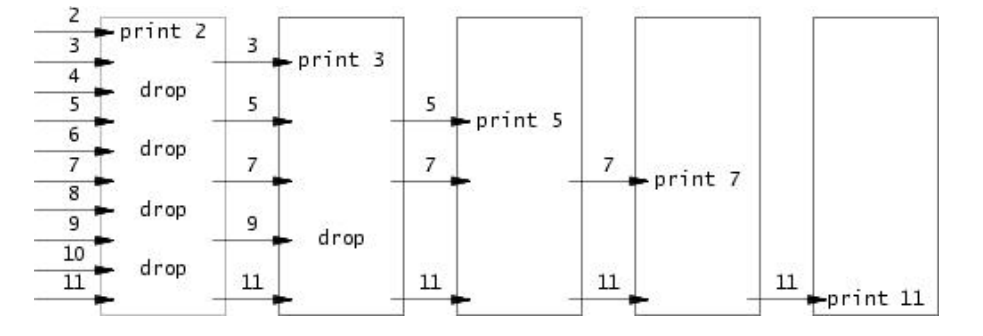
在本次实验中，通过编写一个简单的“乒乓”通信程序，深入理解了进程间通信的基本原理和使用管道（pipe）进行数据传输的方法。实验中，我们成功地创建了父进程和子进程，并通过管道实现了父子进程之间的双向通信。

在实验中需要注意： 使用fork( )函数创建子进程时，需注意fork( )返回值的判断，以便区分父进程和子进程的操作。在程序过程中，需要特别注意管道的开关管理。如果一个管道的读写两端都被关闭，那么该管道就会被关闭。若再次需要使用通道时，需要重新建立管道。

Primes(moderate)

* 1. 实验目的
* 学习如何在两个进程之间建立通信通道
* 通过管道来写出一个可以筛出2-35中的素数的程序
  1. 实验步骤

如下图所示，首先将所有的数字输入到最左边的管道，然后打印出输入管道的第一个数2，并将管道中所有2的倍数剔除。接着将剔除后的所有数字输入到右边的管道，打印第一个数字3并剔除3的倍数，以此类推最终打印出所有素数。



* 创建user/primes.c
* 在Makefile中的UPROGS下添加 $U/\_primes\
* 编写primes.c，代码如下：

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#define READEND 0

#define WRITEEND 1

// 子进程函数

void child(int\* fd) {

int p, n;

// 关闭写端

close(fd[WRITEEND]);

// 从管道中读取一个素数

if (read(fd[READEND], &p, sizeof(p)) == 0) {

// 如果读取失败，关闭读端并退出

close(fd[READEND]);

exit(0);

}

// 打印素数

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

// 创建新的管道

int newfd[2];

pipe(newfd);

// 创建新的子进程

if (fork() == 0) {

child(newfd);

}

// 关闭新管道的读端

close(newfd[READEND]);

// 循环读取数字

while (read(fd[READEND], &n, sizeof(n)) != 0) {

// 如果数字不是素数的倍数，写入新管道

if (n % p != 0) {

write(newfd[WRITEEND], &n, sizeof(n));

}

}

// 关闭旧管道的读端和新管道的写端

close(fd[READEND]);

close(newfd[WRITEEND]);

// 等待子进程退出

wait(0);

exit(0);

}

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

int fd[2];

// 创建管道

pipe(fd);

// 创建子进程

if (fork() == 0) {

child(fd);

}

// 关闭读端

close(fd[READEND]);

// 向管道中写入2到35的数字

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

write(fd[WRITEEND], &i, sizeof(i));

}

// 关闭写端

close(fd[WRITEEND]);

// 等待子进程退出

wait(0);

exit(0);

}

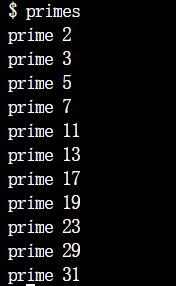
* 说明
* 在main( )函数中，fd变量是一个整型数组，用于存储管道的两个文件描述符。当main( )函数调用fork( )函数创建子进程时，子进程会继承父进程的所有变量和文件描述符。因此，在这种情况下，fd变量是父进程和子进程共享的。

即：在父进程和子进程中，fd变量都指向同一个管道。父进程可以通过fd[1]向管道中写入数据，而子进程可以通过fd[0]从管道中读取数据。

* 在子进程中，main( )函数并不会重新执行。也就是说，子进程不会再次执行main( )函数中的代码，而是直接从child(fd)函数开始执行。

当父进程调用fork( )函数创建子进程时，子进程会从fork( )函数的返回处开始执行。也就是说，子进程会继承父进程的程序计数器、堆栈指针和其他寄存器的值，并从fork( )函数返回的地方开始执行。

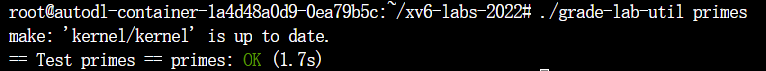
* 1. 实验结果
* 在xv6 shell中运行程序进行测试，结果如下：



* 运行以下命令以进行grade:

./grade-lab-util primes

结果如下：



* 1. 实验中遇到的问题和解决方法
* 问题：close 方法的使用没有掌握
* 解决方法：通过查阅资料了解close函数的具体使用方法。

close( )函数用于关闭一个文件描述符。它接受一个整数参数，表示要关闭的文件描述符。、

close( )函数的原理是通过操作系统内核来关闭一个文件描述符。当你调用 close( )函数时，它会通知内核关闭指定的文件描述符。内核会释放与该文件描述符相关联的所有资源，包括内存缓冲区、锁等。

* 1. 实验心得

在这次实验中，我实现了一个基于进程间通信的素数筛选算法，通过管道和子进程递归地筛选素数。实验的核心是使用管道和多进程通信。通过管道在父子进程之间传递数据，使得每个进程都可以接收上一个进程传递的数字并进行处理。每个子进程负责读取一个素数p，并过滤掉所有p的倍数。剩余的数字通过管道传递给下一个子进程，继续筛选。这种递归调用的方式确保了每个进程都只处理当前范围内的数字。

管道在进程间通信中起到了关键作用。每个子进程创建一个新的管道，用于传递筛选后的数字。这种方式不仅简化了数据传递过程，也保证了数据的有序处理。每个子进程在完成自己的任务后，需要关闭管道并等待子进程结束。这种方式确保了资源的正确释放和进程的有序退出。

Find(moderate)

* 1. 实验目的
* 学习如何递归地遍历目录，并在每个目录中查找特定的文件
* 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
  1. 实验步骤
* 查看user/ls.c文件，了解如何读取目录。其中有两个函数：

void ls(char \*path)：用于显示指定路径下的所有文件（包含文件夹）

char \*fmtname(char \*path)：提取出给定路径的文件名并返回规范化字符串

* 创建user/find.c
* 在Makefile中的UPROGS下添加 $U/\_find\
* 编写find.c，代码如下：

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/fs.h"

// 定义 find 函数，用于在指定目录及其子目录中查找文件

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

int fd;

struct stat st;

struct dirent de;

char buf[512], \* p;

// 打开目录

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

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

return;

}

// 获取文件状态

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

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

close(fd);

return;

}

// 判断是否为目录

switch (st.type) {

case T\_FILE: // 如果文件类型是普通文件，不做任何处理

break;

case T\_DIR: // 如果文件类型是目录，继续处理

// 如果路径名太长，超过了缓冲区的大小，打印错误信息并跳出

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

printf("find: path too long\n");

break;

}

// 把路径名复制到缓冲区中

strcpy(buf, path);

// 把指针 p 指向缓冲区的末尾

p = buf + strlen(buf);

// 在末尾添加一个斜杠，作为目录的分隔符

\*p++ = '/';

// 循环读取目录中的每个条目，直到读完或出错

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

// 如果条目是空的或者是当前目录或上级目录，跳过不处理

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

continue;

// 把条目的文件名复制到 p 指向的位置，形成完整的路径名

memmove(p, de.name, DIRSIZ);

// 在文件名后面添加一个空字符，作为字符串的结束符

p[DIRSIZ] = 0;

// 获取条目的文件状态，如果失败，打印错误信息并继续下一个条目

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

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

continue;

}

// 如果找到了匹配的文件名，打印路径

if (!strcmp(de.name, filename))

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

// 如果是目录，递归查找

if (st.type == T\_DIR)

find(buf, filename);

}

break;

}

close(fd);

}

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

// 检查命令行参数个数

if (argc != 3) {

fprintf(2, "Error: find requires 2 arguments\nRight Usage: find [path] [filename]\n");

exit(1);

}

// 调用 find 函数进行查找

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

exit(0);

}

* 说明
* find函数思路：

1. 打开目录：

使用open函数打开指定路径，如果失败，打印错误信息并返回。

1. 获取文件状态：

使用fstat函数获取文件状态信息，如果失败，打印错误信息并关闭文件描述符后返回。

1. 判断文件类型：

使用switch语句判断文件类型：

普通文件 (T\_FILE)：直接跳过，不做处理。

目录 (T\_DIR)：进行进一步处理。

1. 处理目录：

检查路径长度是否超过缓冲区大小，如果超过，打印错误信息并跳过。

将路径名复制到缓冲区 buf 中，并在末尾添加一个斜杠 /。

1. 读取目录内容：

使用read函数循环读取目录中的每个条目：

如果条目是空的，或者是当前目录 . 或上级目录 ..，则跳过。

将条目的文件名复制到缓冲区中，形成完整的路径名，并添加一个空字符作为结束符。

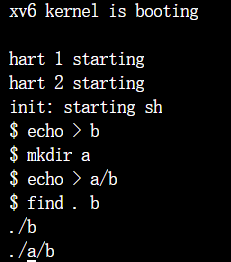
使用stat函数获取条目的文件状态，如果失败，打印错误信息并继续下一个条目。如果找到匹配的文件名，打印完整路径。

如果条目是目录，递归调用 find 函数。

1. 关闭文件描述符：

关闭当前目录的文件描述符。

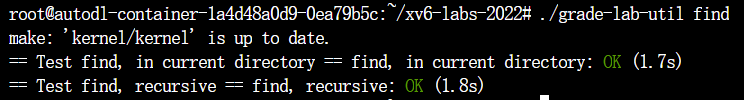
* 1. 实验结果
* 在xv6 shell中运行程序进行测试，结果如下：



* 运行以下命令以进行grade:

./grade-lab-util find

结果如下：



* 1. 实验中遇到的问题和解决办法
* 问题：memmove方法的使用没有掌握
* 解决方法：通过查阅资料了解memmove函数的具体使用方法。

memmove( )

memmove函数用于将一块内存区域的内容复制到另一块内存区域。它的原型为：void \*memmove(void \*dest, const void \*src, size\_t n);

其中，dest是目标内存区域的起始地址，src是源内存区域的起始地址，n是要复制的字节数。

memmove(p, de.name, DIRSIZ) 这行代码的作用是把条目的文件名复制到p指向的位置，形成完整的路径名。其中，p是一个指针，它指向缓冲区中路径名的末尾，de.name是条目的文件名，DIRSIZ是文件名的最大长度。

具体来说，memmove( )函数会从源内存地址（即 de.name）复制DIRSIZ个字节到目标内存地址（即p指向的位置）。这样，条目的文件名就被复制到了缓冲区中路径名的末尾，形成了一个完整的路径名。

如果文件名小于DIRSIZ，那么memmove( )函数不会复制DIRSIZ个字节，但只会复制真实文件名长度。

* 1. 实验心得

在这次实验中，我实现了一个类似Unix中find命令的功能，用于在指定目录及其子目录中查找特定文件。通过这次实验，我深入理解了系统调用、文件和目录操作以及递归处理的相关知识。通过这次实验，我深入理解了memmove函数的用法和作用。memmove可以在源和目的地内存区域有重叠的情况下安全地移动数据，而不引发数据损坏。这对于处理字符串和内存块非常有用。通过递归调用find函数处理目录及其子目录，我学习了递归处理在实际编程中的应用。递归方法使得复杂的目录结构处理变得简单高效。

Xargs（moderate）

* 1. 实验目的

1. 学习如何使用系统调用来执行外部程序。
2. 学习如何使用进程等待函数。
3. 了解xargs的相关作用以及其原理。
   1. 实验步骤

* 创建user/xargs.c
* 在Makefile中的UPROGS下添加 $U/\_xargs\
* 编写xargs.c，代码如下：

#include "kernel/types.h"

#include "kernel/stat.h"

#include "user/user.h"

#include "kernel/fs.h"

#include "kernel/fcntl.h"

#include "kernel/param.h"

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

char buf[512]; // 缓冲区

char\* p = buf; // 指向缓冲区的指针

char\* args[MAXARG]; // 参数数组

int i, n;

// 如果参数数量小于2，报错，打印正确用法并退出

if (argc < 2) {

fprintf(2, "Error: xargs requires 1 or more arguments\nRight Usage: xargs command ...\n");

exit(1);

}

// 将命令行参数复制到参数数组中

for (i = 1; i < argc; i++)

args[i - 1] = argv[i];

// 读取标准输入

while ((n = read(0, p, 1)) > 0) {

// 如果读取到换行符

if (\*p == '\n') {

\*p = 0; // 将换行符替换为字符串结束符

args[argc - 1] = buf; // 将缓冲区中的字符串作为参数添加到参数数组中

args[argc] = 0; // 设置参数数组的结束标志

// 创建子进程执行命令

if (fork() == 0) {

exec(args[0], args);

exit(0);

}

else {

wait(0); // 父进程等待子进程完成

}

p = buf; // 重置指针指向缓冲区开头

}

else {

p++; // 指针后移

}

}

exit(0); // 退出程序

}

* 说明
* sh

$ sh < xargstest.sh 是一个 shell 命令，它的作用是在 xv6 系统中运行 xargstest.sh 脚本来测试 xargs 程序。

sh 是 xv6 系统中的 shell 程序，它用于执行命令。 < 符号表示输入重定向，它将文件中的内容作为标准输入传递给命令。在这个例子中， < xargstest.sh 表示将 xargstest.sh 文件中的内容作为标准输入传递给 sh 命令。

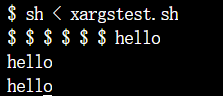
因此，整个命令的意思是：使用 sh 命令执行 xargstest.sh 文件中的脚本。这将运行脚本中的测试命令来测试 xargs 程序是否正确实现了预期的功能。

* exec( )

exec 是一个系统调用，它用于在当前进程中执行一个新的程序。它接受两个参数：要执行的程序的文件名和一个指向参数数组的指针。在这里， exec(args[0], args) 表示执行 args[0] 指定的程序，并将 args 数组中的参数传递给该程序。

当 exec 被调用时，它会用指定的程序替换当前进程的代码、数据和堆栈段。然后，新程序从其入口点开始执行。这意味着 exec 调用之后的代码将不会被执行，除非 exec 调用失败。 exec(args[0], args) 被用于在子进程中执行命令。当 fork 函数创建一个新的子进程时，它会返回两次：一次在父进程中返回子进程的进程 ID，一次在子进程中返回 0 。因此， if (fork() == 0) 判断语句用于检查当前代码是否在子进程中执行。如果是，则调用 exec(args[0], args) 来执行命令，并在执行完成后调用 exit(0) 退出子进程。

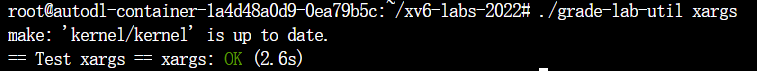
* 1. 实验结果
* 在xv6 shell中运行程序进行测试，结果如下：



* 运行以下命令以进行grade:

./grade-lab-util xargs

结果如下：



* 1. 实验中遇到的问题和解决办法
* 问题：不知道如何在子进程中执行命令
* 解决方法：在子进程中使用 fork() 创建新的进程，并在新进程中使用 exec 系列函数执行命令。
* 问题：对实验内容的理解有疑惑
* 解决方法：

我一开始对于这个函数的功能感到很疑惑，看完任务要求后也不知道它到底要我做什么。后来我去查了相关的资料，了解了标准输入等一系列相关概念后才弄明白这个实验在做什么。

例子中的find . b | xargs grep hello 指令，其实就是用xargs将find命令查找到的文件名作为参数传递给grep命令，然后grep命令将该文件名作为参数，最终会在这些文件中查找包含"hello"字符串的行。理解了xargs指令的意义之后实验就变得清晰起来。

* 1. 实验心得

这次实验实现了一个简化版的 xargs 命令，通过读取标准输入并将其作为参数传递给指定的命令进行执行。通过使用 fork 创建子进程，exec 执行命令，以及 wait 等待子进程完成，我对进程控制有了更深的理解。这些系统调用在实际开发中非常重要，尤其是在需要并行处理或执行外部命令时。在实验中，我学会了如何构建参数数组，并将标准输入数据添加到数组中以便传递给 exec 函数。这对处理动态输入参数和构建命令行工具非常有帮助。

测试结果

