Lab1 Xv6 and Unix utilities

目录

1 Boot xv6

1.1 实验目的

配置后续实验所需的环境，并认识 xv6 的初步使用，了解主要命令行操作

1.2 实验步骤

1. 下载 github 上的“xv6-labs-2020”到本地，本机存放于 \\wsl.localhost\Ubuntu-20.04\root 的路径下

方法：在命令行输入 **$git clone git://g.csail.mit.edu/xv6-labs-2020**

2. 进入目录

方法：在命令行输入 **$cd xv6-labs-2020**

3. 切换到新分支“util”

方法：在命令行输入 **$_git checkout util**

4. 构建并运行 xv6

方法：在命令行输入 **$make qemu**

```
Markdown
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
```

运行结果如上所示，表示成功运行 xv6

5. 测试 xv6，输入 ls

方法：在命令行输入 **$ls**

```
Markdown
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2059
xargstest.sh   2 3 93
cat            2 4 23888
echo           2 5 22720
forktest       2 6 13080
```

```
grep            2 7 27248
init            2 8 23824
kill            2 9 22696
ln              2 10 22648
ls              2 11 26120
mkdir           2 12 22792
rm              2 13 22784
sh              2 14 41656
stressfs        2 15 23792
usertests       2 16 147432
grind           2 17 37912
wc              2 18 25032
zombie          2 19 22184
console         3 20 0
```

运行结果如上所示，表示成功查看了各个文件中的内容

6. 查看各进程信息

方法：在命令行输入 **Ctrl + p**

```
Markdown
1 sleep  init
2 sleep  sh
```

运行结果如上所示，表示成功查看各个进程的状态

7. 退出 qemu

方式：在命令行输入 **Ctrl-a x**

```
Markdown
QEMU: Terminated
```

显示以上结果，说明已经退出 qemu

1.3 实验中遇到的问题和解决方法

本实验严格按照教程即可，无错误和问题

1.4 实验心得

通过完成启动 xv6 的实验，我不仅对操作系统的启动过程有了更深入的了解，还学到了许多与编程和调试相关的技能，初步认识到了 xv6 的启动与使用。这个实验提供了

一个很好的机会，让我实践操作系统的基本原理，并加深了我对软件工程的兴趣。我期待在接下来的实验中继续探索和学习。

2 sleep

1.1 实验目的

初步了解 xv6 系统中暂停功能的实现，使用函数模拟操作系统的具体功能。通过 sleep 程序，使系统暂停一定时钟周期，熟悉 linux 下函数需要存放的位置，认识 kernel 和 user 文件夹下的主要库函数，掌握 linux 编译运行的基本操作。

- 在开始编码之前，请阅读 xv6 书的第 1 章。
- 查看 user/中的一些其他程序 （例如 user/echo.c、user/grep.c 和 user/rm.c），了解如何获取传递给程序的命令行参数。
- 如果用户忘记传递参数，sleep 应该打印一条错误消息。
- 命令行参数作为字符串传递；您可以使用 atoi 将其转换为整数（请参阅 user/ulib.c）。
- 使用系统调用 sleep。
- 请参阅 kernel/sysproc.c 以了解实现 sleep 系统调用的 xv6 内核代码（查找 sys_sleep），请参阅 user/user.h 以了解可从用户程序调用 sleep 的 C 定义，以及 user/usys.S 以了解汇编代码从用户代码跳转到内核进行睡眠。
- 确保 main 调用 exit()以退出程序。
- 将你的睡眠程序添加到 Makefile 中的 UPROGS 中；完成此操作后，make qemu 将编译您的程序，您将能够从 xv6 shell 运行它。
- 查看 Kernighan 和 Ritchie 的书 *《C 编程语言（第二版）》*（K&R）来了解 C。

1.2 实验步骤

1. sleep.c 函数编写

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

int main(int argc, char* argv[]) {
  if (argc != 2) {
    fprintf(2, "Usage: sleep <ticks>\n");
    exit(1);
```

```
  }

  int ticks = atoi(argv[1]);
  sleep(ticks);
  fprintf(1,"Process has slept for %d ticks\n",ticks);
  exit(0);
}
```

思路分析：

在这个示例代码中，我们首先检查命令行参数的数量是否为 2。如果不是 2 个参数，则输出用法提示并退出程序。接下来，我们使用 atoi 函数将第二个命令行参数转换为整数值，并将其存储在 ticks 变量中。然后，我们调用 sleep 函数，将 ticks 作为参数传递给它，以使当前进程暂停指定的时钟周期数。最后，我们使用 fprintf 函数打印一条消息，并退出。

2. 编译准备

将 sleep.c 文件置于 user 文件夹下

并在文件名为 Makefile 的文件的中 UPROGS 一项的最后一行加上$U/_sleep\ (注意：没有这行命令，整个函数将无法编译!!)

3. 执行函数

先在命令行输入 make qemu 进入 xv6

然后输入类似”sleep X“ 的命令，效果是可以使程序暂停 x 个时钟周期，这里我们以 sleep 5 为例子

```
Markdown
$ sleep 5
Process has slept for 5 ticks
```

可以看到，在输入命令后，等待了 5 个时钟周期才显示下一行结果，提示本次执行暂停的时间

4. 进行测试

在测试前需要先退出，在命令行输入 Ctrl-a x 即可

然后输入./grade-lab-util sleep 或者 make GRADEFLAGS=sleep grade 命令来检测本次实验的得分

```
Markdown
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (2.6s)
```

```
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
```

得到的结果如上所示

1.3 实验中遇到的问题和解决方法

1. 找不到函数放置位置

由于是第一次需要使用函数来模拟操作的作业，不知道如何在 linux 下实现。在查阅了 xv6 课本第一章的相关知识后，对整个流程有了一个初步的了解，才能最终实现此次作业。

2. 在测试的时候，出现 /usr/bin/env: ‘python’: No such file or directory 的报错

在查阅资料后，意识到是 python3 配置的路径有误，解决方案如下

a. 查看 python3 的版本

`python3 --version`

b. 查找 python3 的安装位置：

`whereis python3`

c. 为其创建符号连接：

`sudo ln -s /usr/bin/python3 /usr/bin/python`

d. 尝试再次启动

1.4 实验心得

本次实验完整模拟了一个简单功能的实现总流程，需要我们写函数，编辑，运行，测试。通过完成使用 xv6 实现 sleep 函数的实验，我不仅深入了解了操作系统的调度和时间管理，还学到了许多与系统编程和调试相关的技能。这个实验不仅加深了我对操作系统的理解，还提升了我的编程能力和解决问题的能力。我期待在接下来的实验中继续学习和成长。

3 pingpong

3.1 实验目的

编写一个程序，使用 UNIX 系统调用通过一对管道（每个方向一个）在两个进程之间“pingpong”一个字节。父进程应向子进程发送一个字节；子进程应打印“<pid>：已收到 ping”，其中 <pid> 是其进程 ID，将管道上的字节写入到父进程，然后退出；

父进程应该从子进程读取字节，打印“<pid>：收到 pong”，然后退出。解决方案应该位于文件 user/pingpong.c 中。

3.2 实验要求

- 使用管道创建管道。
- 使用 fork 创建一个孩子。
- 使用 read 从管道读取数据，使用 write 向管道写入数据。
- 使用 getpid 查找调用进程的进程 ID。
- 将程序添加到 Makefile 中的 UPROGS 中。
- xv6 上的用户程序具有一组有限的可用库函数。您可以在 user/user.h 中看到该列表 ；源代码（系统调用除外）位于 user/ulib.c、user/printf.c 和 user/umalloc.c 中。

3.3 实验步骤

1. pingpong.c 函数编写

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

int main(int argc, char* argv[])
{
  int parent[2], child[2];
  pipe(parent);//创建父进程管道
  pipe(child);//创建子进程管道
  char msg[32];
  if (fork()) {
    write(parent[1], "ping", 4);
    read(child[0], msg, 4);
    wait(0);// 等待子进程结束
    fprintf(1, "%d: received %s\n", getpid(), msg);
  }// 父进程
  else {
    write(child[1], "pong", 4);
    read(parent[0], msg, 4);
    fprintf(1, "%d: received %s\n", getpid(), msg);
```

```
  }//子进程
  exit(0);
}
```

思路分析：

该代码创建了两个管道：parent 用于父进程和子进程之间的通信，child 用于子进程和父进程之间的通信。在 fork()之后，父进程通过 write(parent_fd[1], "ping", 4)将字符串 "ping" 写入父进程的管道。然后，父进程通过 read(child_fd[0], buf, 4)从子进程的管道中读取消息。父进程使用 wait(0)等待子进程结束，然后打印接收到的消息。子进程部分，子进程通过 write(child_fd[1], "pong", 4)将字符串 "pong" 写入子进程的管道。然后，子进程通过 read(parent_fd[0], buf, 4)从父进程的管道中读取消息，并打印接收到的消息。最后，父进程和子进程通过调用 exit(0)正常退出程序。

2. 编译准备

将 pingpong.c 文件置于 user 文件夹下

并在文件名为 Makefile 的文件的中 UPROGS 一项的最后一行加上$U/_pingpong\ (注意：没有这行命令，整个函数将无法编译!!)

3. 执行函数

先在命令行输入 make qemu 进入 xv6

然后输入 pingpong，查看结果

```
Markdown
$ pingpong
4: received ping
3: received pong
```

4. 进行测试

在测试前需要先退出，在命令行输入 Ctrl-a x 即可

然后输入./grade-lab-util pingpong 或者 make GRADEFLAGS=pingpong grade 命令来检测本次实验的得分

```
Markdown
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (1.0s)
```

得到的结果如上所示

3.4 实验中遇到的问题和解决方法

本次实验的主要难点在于理解题目的要求，本程序的要求是父进程应向子进程发送一个字节；子进程应打印“<pid>：已收到 ping”，其中 <pid> 是其进程 ID，将管道上的字节写入到父进程，然后退出；父进程应该从子进程读取字节，打印“<pid>：收到 pong”，然后退出。在起初由于审题不仔细，始终不能理解答案中“4”和“3”的出现，对后续的函数编写产生了误解。在查阅相关资料后，意识到自己的错误，事实上输出行的“4”和“3”是由 getpid()函数产生的，其代表的是进程 ID，其目的是取得进程识别码，许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。最后，在纠正了原本错误的理解之后，顺利完成此次实验。

3.5 实验心得

xv6 上的用户程序具有一组有限的可用库函数，可以在 user/user.h 中看到该列表 ；源代码（系统调用除外）位于 user/ulib.c、user/printf.c 和 user/umalloc.c 中。本次实验使用了许多库中的函数，加深了我对于 xv6 的理解。在进行 xv6 实验时，我有机会尝试实现一个基于管道和 fork 的 pingpong 函数，用于在父子进程之间进行简单的消息传递。通过完成这个实验，我对进程间通信和并发编程有了更深入的了解，并学到了一些关键的编程技巧。这个实验提供了一个很好的机会，让我在实践中探索操作系统的并发性和进程间通信的实现。我期待在接下来的实验中继续学习和发展。

4 primes

4.1 实验目的

使用管道编写素数筛的并发版本。这个想法源自 Unix 管道的发明者 Doug McIlroy。解决方案应该位于文件 user/primes.c 中。目标是使用 pipe 和 fork 来设置管道,第一个进程将数字 2 到 35 输入管道。对于每个质数，您将安排创建一个进程，通过管道从其左邻居读取数据，并通过另一个管道向其右邻居写入数据。由于 xv6 的文件描述符和进程数量有限，第一个进程可以停在 35 个。

4.2 实验要求

- 请小心关闭进程不需要的文件描述符，否则您的程序将在第一个进程达到 35 之前耗尽资源来运行 xv6。
- 一旦第一个进程达到 35，它应该等待整个管道终止，包括所有子进程、孙进程等。因此，主 primes 进程仅应在所有输出打印完毕以及所有其他 primes 进程退出后退出。
- 提示：当管道的写入端关闭时， 读取返回零。
- 最简单的方法是直接将 32 位（4 字节）int 写入管道，而不是使用格式化的

ASCII I/O。

- 您应该仅在需要时在管道中创建流程。
- 将程序添加到 Makefile 中的 UPROGS 中。

4.3 实验步骤

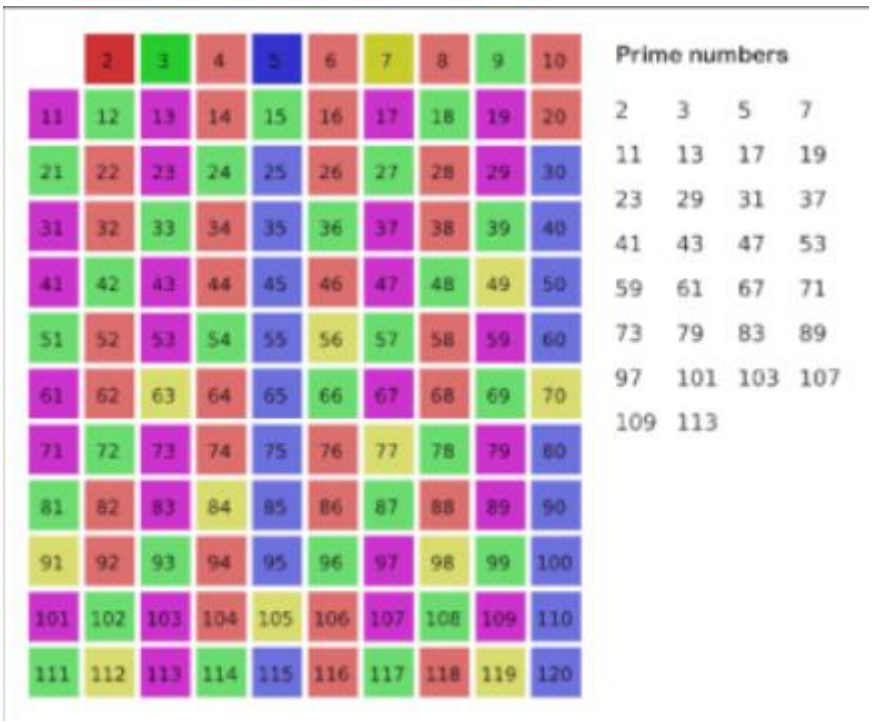
1. 质数筛

在进行函数编写前，我们需要先了解用质数筛来筛选质数的基本思想，其中主流的质数筛有高斯筛和埃氏筛，本实验采用更适合计算机编程的埃氏筛来实现。

埃氏筛是一种用于筛选出一定范围内所有质数的算法。它是古希腊数学家埃拉托色尼斯（Eratosthenes）在约公元前 240 年提出的。埃氏筛的基本思想是从小到大遍历所有自然数，并标记其倍数为合数。具体步骤如下：

- 创建一个长度为 n+1 的布尔数组（或标记数组），初始值为 true。其中 n 是我们要筛选的范围上限。
- 从 2 开始，遍历到 sqrt(n)为止。对于每个遍历到的数，如果其标记为 true，表示它是质数，然后将其倍数（排除本身）的标记设置为 false。
- 遍历完成后，所有标记为 true 的数都是质数。

埃氏筛的核心思想是利用倍数的性质，将合数的倍数排除掉，从而筛选出质数。在遍历的过程中，我们只需要考虑小于等于 sqrt(n)的数，因为大于 sqrt(n)的数的倍数在之前的遍历中已经被标记过了。在实现埃氏筛算法时，我们可以结合数组操作和循环来实现标记和排除合数的过程。算法的性能可以通过适当的优化和空间复杂度的改进来提高。

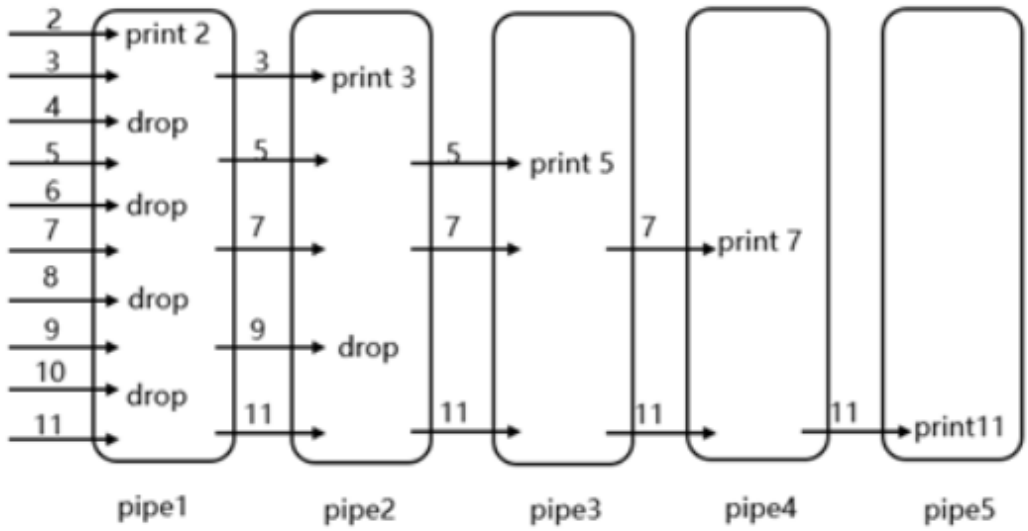


传统的埃氏筛算法

基于管道实现的质数筛算法是一种利用管道和多进程的方法来并行地筛选质数。它通过将不同进程之间的数据传递和处理分离，以提高算法的效率。下面是简要的描述：

- 首先，创建一个管道，用于父进程和子进程之间的通信。
- 父进程负责生成一系列自然数，并将它们逐个写入管道中。
- 父进程创建第一个子进程，该子进程负责读取管道中的自然数，并判断是否为质数。
- 如果子进程确定某个数是质数，它将该数写入另一个管道中。
- 父进程继续生成下一个自然数，并将其写入管道中。
- 父进程创建下一个子进程，负责读取管道中的自然数，并判断是否为质数。子进程重复上述步骤，将质数写入下一个管道中。
- 这个过程可以持续进行，直到所有自然数都被遍历完。
- 最后，父进程读取最后一个管道中的质数，这些质数即为筛选出的结果。

通过使用管道和多进程，这种方法将质数的筛选过程并行化，提高了效率。每个子进程负责处理一部分数值，同时可以利用已经筛选出的质数来加速判断过程。整个算法的时间复杂度取决于质数的个数和范围，相对于串行的质数筛算法有一定的性能提升。需要注意的是，管道实现的质数筛算法在具体实现时需要考虑进程之间的同步和通信问题，以确保正确性和避免竞态条件。此外，对于较大范围的质数筛选，可能需要对算法进行进一步的优化和改进，以提高效率和降低资源消耗。



基于管道的质数筛

2. primes.c 函数编写

```
Markdown
#include "kernel/types.h"
#include "user/user.h"
#define MAX 35

int main() {
  int parent[2], child[2];
  pipe(parent); // 创建父进程的管道
  pipe(child); // 创建子进程的管道
  if (fork()) {
    close(parent[0]);
    close(child[1]);
    int num;
    for (num = 2; num <= MAX; ++num) {
      write(parent[1], &num, sizeof(num)); // 向管道写入自然数
    }
    close(parent[1]); // 写入结束
    int prime;
    while (read(child[0], &prime, sizeof(prime)) > 0) {
      fprintf(1, "prime %d\n", prime);
    }
    close(child[0]); // 读取结束
    wait(0);
  }//父进程
  else {
    close(parent[1]);
    close(child[0]);
    int num;
    while (read(parent[0], &num, sizeof(num)) > 0) {
      int is_prime = 1;
      for (int i = 2; i * i <= num; ++i) {
        if (num % i == 0) {
          is_prime = 0;
          break;
        }
      }
      if (is_prime) {
        write(child[1], &num, sizeof(num)); // 向管道写入质数
```

```
      }
    }
    close(parent[0]); // 读取结束
    close(child[1]); // 写入结束
    exit(0);
  }//子进程
  exit(0);
}
```

思路分析：

在上述示例代码中，创建了两个管道：parent 用于父进程和子进程之间的通信，child 用于子进程和父进程之间的通信。父进程负责生成从 2 到 35 的自然数，并将它们逐个写入 parent 管道。子进程负责从 parent 管道读取自然数，并判断是否为质数。如果是质数，子进程将其写入 child 管道。父进程在循环结束后关闭写端，然后从 child 管道读取质数，并打印出来。子进程在循环结束后关闭读端和写端，然后退出。编译并运行这个程序，你将看到输出结果中列出了从 2 到 35 范围内的所有质数。

2. 编译准备

将 primes.c 文件置于 user 文件夹下

并在文件名为 Makefile 的文件的中 UPROGS 一项的最后一行加上$U/_primes\ (注意：没有这行命令，整个函数将无法编译!!)

3. 执行函数

先在命令行输入 make qemu 进入 xv6

然后输入 primes，查看结果

```
Markdown
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
```

4. 进行测试

在测试前需要先退出，在命令行输入 Ctrl-a x 即可

然后输入./grade-lab-util primes 或者 make GRADEFLAGS=primes grade 命令来检测本次实验的得分

```
Markdown
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (0.7s)
```

4.4 实验中遇到的问题和解决方法

本次实验的难点在于理解质数筛的原理，以及相应的计算机算法的实现。

参考资料：https://oi-wiki.org/math/number-theory/sieve/

在努力理解算法的原理、思想和步骤后，通过对算法背后的数学和计算原理的理解，基本让我充分了解了关于质数筛的知识，以便下一步代码的编写。

4.5 实验心得

在完成 xv6 中的 primes 实验后，我对操作系统的并发编程、进程间通信和质数筛算法有了更深入的了解。primes 实验需要实现进程间通信，通过管道实现父子进程之间的消息传递。这使我意识到进程间通信在并发编程中的重要性，它是实现多个进程之间协作和数据传递的关键。primes 实验要求编写一个质数筛算法，通过并发的方式筛选出一定范围内的质数。这让我对质数筛算法的实现和优化有了更深入的了解，包括如何设计算法、选择数据结构和考虑性能优化。这个实验为我今后在操作系统和并发编程领域的学习和研究打下了坚实的基础。

5 find

5.1 实验目的

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。您的解决方案应该位于文件 user/find.c 中。

5.2 实验要求

- 查看 user/ls.c 以了解如何读取目录。
- 使用递归允许 find 深入到子目录。
- 不要递归成“.” 和 ”..”。

- 文件系统的更改在 qemu 运行期间持续存在；运行一个干净的文件系统，make clean 然后 make qemu.
- 您需要使用 C 字符串。看一下 K&R（C 书），例如第 5.5 节。
- 请注意，== 并不像 Python 中那样比较字符串。请改用 strcmp()。
- 将程序添加到 Makefile 中的 UPROGS 中。

5.3 实验步骤

1. 思路分析

文件搜索的核心在于用递归的思想实现对子文件的搜索。而文件搜索主要分为两种情况，一种是当前位置搜索到文件，则直接进行比较如果满足要求直接输出；另一种情况是比较到目录类型，所以在这种情况下我们需要需要系统调用读取目录中的每个目录项并存储，然后遍历目录中的每个文件和子目录，最后用递归的方法继续比较。

2. find.c 函数编写

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

char* fmtname(char* path)
{
  static char buf[DIRSIZ + 1];
  char* p;
  for (p = path + strlen(path); p >= path && *p != '/'; p--);
  p++;
  // 查找最后一个斜杠后的第一个字符
  if (strlen(p) >= DIRSIZ)
    return p;
  memmove(buf, p, strlen(p)); // 从 p 复制 strlen(p)个字符到 buf
  memset(buf + strlen(p), '\0', DIRSIZ - strlen(p));// 复制 '\0'到
buf+strlen(p)的 DIRSIZ - strlen(p)位置
  return buf;// 返回空格填充的文件名
}// 从路径中提取文件名

void find(char* path, char* fileName) {
  int fd = open(path, 0);
  char buf[512], * p;
```

```
  struct dirent de;
  if (fd < 0) {
    return;
  }
  struct stat st;
  if (stat(path, &st) < 0) {
    close(fd);
    return;
  } // 通过 stat 系统调用获取指定路径的文件或目录的状态，并将结果存储在 struct stat 结构体变量 st 中。如果获取状态失败，则关闭文件描述符并返回。
  switch (st.type)
  {
  case T_FILE: //文件的情况
    if (strcmp(fmtname(path), fileName) == 0) {
      fprintf(1, "%s\n", path); //直接打印路径
    }
    break;
  case T_DIR: // 目录的情况
    if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {
      break;
    }
    strcpy(buf, path);
    p = buf + strlen(buf);
    *p++ = '/';
    while (read(fd, &de, sizeof(de)) == sizeof(de)) {
    // 通过 read 系统调用读取目录中的每个目录项，并将目录项存储在 struct dirent 结构体变量 de 中。
      if (de.inum == 0)
        continue;
      if (strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
        continue;
      memmove(p, de.name, DIRSIZ);
      p[DIRSIZ] = 0;
      // 构建子目录路径：将当前目录路径与目录项的名称拼接起来，构建子目录的路径。
      if (stat(buf, &st) < 0) {
        continue;
      }
      find(buf, fileName);// // 递归调用 find 函数，查找子目录下的文件
    }
```

```
    break;
  }
  close(fd);
}// 递归查找指定目录下的文件，并打印匹配的文件路径

int main(int argc, char* argv[]) {
  if (argc < 3) {
    fprintf(2, "Usage: find [dir] [filename]\n");
    exit(1);
  }// 判断输入是否合法
  find(argv[1], argv[2]);
  exit(0);
}
```

其中

fmtname 函数：该函数用于从给定的路径中提取文件名。它查找最后一个斜杠后的第一个字符，并返回一个空格填充的文件名。

find 函数：该函数是递归地查找指定目录下的文件，并打印与指定文件名匹配的文件路径。它接受两个参数，path 表示当前要查找的路径，fileName 表示要匹配的文件名。

main 函数：主函数接收命令行参数，调用 find 函数来查找指定目录下与指定文件名匹配的文件。

3. 编译准备

将 find.c 文件置于 user 文件夹下

并在文件名为 Makefile 的文件的中 UPROGS 一项的最后一行加上$U/_find\ (注意：没有这行命令，整个函数将无法编译!!)

4. 执行函数

先在命令行输入 make qemu 进入 xv6

然后输入诸如 “find . a”，之类的语句，查看结果

此处我们以“find . find”为例，输入结果如下

```
Markdown
$ find . find
./find
```

5. 进行测试

在测试前需要先退出，在命令行输入 Ctrl-a x 即可

然后输入./grade-lab-util find 或者 make GRADEFLAGS=find grade 命令来检测本次

实验的得分

```
Markdown
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK
(1.4s)
== Test find, recursive == find, recursive: OK (0.9s)
```

得到的结果如上所示

5.4 实验中遇到的问题和解决方法

对我而言，本实验的一大难点在于如何处理比较到目录类型，当查找到目录类型时，需要系统调用读取目录中的每个目录项并存储，然后遍历目录中的每个文件和子目录。通过比较目录项的名称是否为"."或".."，来判断当前目录项是否为当前目录或父目录。如果是，则继续下一个目录项。最后，再将目录项合并，输出结果。

5.5 心得体会

本次实验是我在 xv6 实验中首次使用递归的思想来解决问题，在实现上参考了 ls 命令的输出。find 实验涉及到对 xv6 文件系统的操作，通过打开、读取和关闭文件描述符，以及使用 stat 函数获取文件的元数据信息。通过这个实验，我更好地理解了文件系统的基本操作和文件结构。然后我们需要实现递归查找指定目录下的文件，通过遍历目录和递归调用自身来实现。这让我深刻理解了递归算法在解决问题中的应用，尤其是在树形结构的问题中。本次实验也是我们操作系统课程中文件系统大作业的一次延伸，又一次对文件搜索进行处理，加深了我对于文件管理的理解。

6 xargs

6.1 实验目的

编写一个简单版本的 UNIX xargs 程序：从标准输入读取行并为每行运行一个命令，将该行作为参数提供给命令。您的解决方案应该位于文件 user/xargs.c 中。

6.2 实验要求

- 使用 fork 和 exec 在每行输入上调用命令。在父级中使用 wait 来等待子级完成命令。
- 要读取单行输入，请一次读取一个字符，直到出现换行符（'\n'）。
- kernel/param.h 声明了 MAXARG，如果您需要声明 argv 数组，这可能很有用。

- 将程序添加到 Makefile 中的 UPROGS 中。
- 文件系统的更改在 qemu 运行期间持续存在；获得干净的文件系统运行使干净进而制作 qemu。

6.3 实验步骤

1. xargs 基本概念

xargs 是给命令传递参数的一个过滤器，也是组合多个命令的一个工具。xargs 可以将管道或标准输入（stdin）数据转换成命令行参数，也能够从文件的输出中读取数据。也可以将单行或多行文本输入转换为其他格式，例如多行变单行，单行变多行。xargs 默认的命令是 echo，这意味着通过管道传递给 xargs 的输入将会包含换行和空白，不过通过 xargs 的处理，换行和空白将被空格取代。xargs 是一个强有力的命令，它能够捕获一个命令的输出，然后传递给另外一个命令。

参考资料：https://www.runoob.com/linux/linux-comm-xargs.html

2. 思路分析

shell 中可以使用管道将之前的命令的标准输出作为之后的命令的标准输入。而 xargs 的作用是将 xargs 之前的命令的标准输出作为之后的命令的附加的命令行参数。标准输入和命令行参数，有很大的不同。所以我们需要做的就是每次将前面的命令放到后面，然后依次执行每一个子进程即可。

3. xargs.c 代码编写

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

int main(int argc, char* argv[]) {
  char buf[512], buf2[32][32];
  char* pass[32];
  for (int i = 0; i < 32; ++i) {
    pass[i] = buf2[i];
  }// 初始化 pass 数组，用于存储参数
  for (int i = 1; i < argc; ++i) {
    strcpy(buf2[i - 1], argv[i]);
  } // 将命令行参数拷贝到 buf2 数组中
  int m;
  char* p = buf;
  int pos = argc - 1;
  char* c = pass[pos];
```

```
  pass[pos + 1] = 0;// 设置 pass 数组的末尾为 NULL，作为 exec 函数的参数列表结束标志
  m = read(0, buf, 512); // 从标准输入读取数据，并解析参数
  do {
    p = buf;
    if (m < 0) {
      exit(1);
    }
    // 遍历读取到的数据
    while (p < buf + m) {
      if (*p == ' ' || *p == '\n') {
        *c = '\0';
        if (fork()) {
          wait(0);
        } // 创建子进程，执行参数对应的命令
        else {
          exec(pass[0], pass);
        }
        ++p;
        c = pass[pos];
      }
      else {
        *c++ = *p++;
      }
    }
  }
  while ((m = read(0, buf, 512)) > 0);
  if (m < 0) {
    fprintf(2, "xargs: read error\n");
    exit(1);
  }
  exit(0);
}
```

我们首先定义了两个缓冲区 buf 和 buf2，用于接收上一个程序的标准输出和传递给 exec 的参数。在这个实现中，我们假设传入的参数不会超过 32 个，并且每个参数的长度不超过 32 个字符。请注意，我没有对输入的参数进行安全性检查，这意味着一个过长的参数列表可能会导致缓冲区溢出，破坏掉堆栈数据，导致程序出现错误中断。在实际实现中，可以进行安全性检查或使用动态长度的数组来避免此类问题。接下来，我们使用一个 do...while()循环来读取上一个程序的输出。如果读取到的内容长度为

0 或读取失败，立即停止 xargs 的执行。如果读取到空格或换行符，根据题目要求，我们的`-n`参数固定为 1，将其作为最小单元来执行指令，创建一个新的进程并调用 exec 函数来执行相应的命令。

4. 编译准备

将 xargs.c 文件置于 user 文件夹下

并在文件名为 Makefile 的文件的中 UPROGS 一项的最后一行加上$U/_xargs\（注意：没有这行命令，整个函数将无法编译!!）

5. 执行函数

先在命令行输入 make qemu 进入 xv6

此处我们以 “echo hello too | xargs echo bye” 为例，输入结果如下

```
Markdown
bye hello
bye too
```

6. 进行测试

在测试前需要先退出，在命令行输入 Ctrl-a x 即可

然后输入./grade-lab-util xargs 或者 make GRADEFLAGS=xargs grade 命令来检测本次实验的得分

```
Markdown
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (1.7s)
```

得到的结果如上所示

6.4 实验中遇到的问题和解决方法

本次实验较大的问题在于如何理解 xargs 函数的功能，如何理解为什么要使用 xargs。

事实上 xargs 是一个常用的命令行工具，用于将标准输入的内容作为参数传递给其他命令并执行。它的主要功能如下：

- 参数传递：xargs 将标准输入的内容分割成多个参数，并将这些参数传递给指定的命令进行执行。
- 参数数量控制：xargs 可以控制每次执行的命令参数数量。默认情况下，xargs 将尽可能多地将参数传递给每个命令。如果参数数量超过命令行的最大长度限制，xargs 会将参数分批传递给命令。
- 参数定界符：xargs 可以根据指定的定界符（默认是空格字符）来分隔标准输入的

内容，并将分隔后的字符串作为参数传递给命令。

- 替换字符串：xargs 可以在命令中使用特定的替换字符串（默认是{}），表示将参数插入到命令中的位置。
- 多行输入处理：xargs 可以处理多行输入，不仅仅限于单行输入。它可以根据指定的定界符来确定行与行之间的分隔。
- 使用管道：xargs 常常与其他命令结合使用，通过管道将前一个命令的输出作为 xargs 的输入。

通过这些功能，xargs 能够提高命令行的灵活性和处理能力。它允许用户将标准输入的内容以参数的形式传递给其他命令，并根据需要进行参数控制和替换。这使得在命令行中处理大量参数变得更加高效和方便。

6.5 心得体会

本次 xargs 实验让我更深入地理解了命令行工具的实现原理和使用方式。通过手动实现 xargs 功能，我更好地掌握了参数传递、参数控制和字符串替换等命令行工具的核心概念和功能。我不仅加深了对操作系统和命令行工具的理解，还提高了编程能力和问题解决能力。这个实验让我更深入地了解了 xv6 操作系统的内部工作原理，并提供了一个实践的机会，让我在实际编程中应用所学的知识和技能。我相信这些经验和心得将对我今后的学习有所裨益。

7 Submit the lab

1. 创建一个新文件 time.txt，并在其中放入一个整数，即您在实验室上花费的小时数。不要忘记 git add 和 git commit 该文件。
2. 退出 xv6，实验目录下输入 make grade，即可得到本次实验的总得分，结果如下

```
Markdown
make[1]: Leaving directory '/root/xv6-labs-2020/lab1'
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.2s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.9s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.0s)
== Test pingpong ==
$ make qemu-gdb
```

```
pingpong: OK (1.0s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.1s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.0s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.0s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.1s)
== Test time ==
time: OK
Score: 100/100
```