Lab：Xv6

学号：2353924
姓名：冯俊财
课号：42028704
教师：王冬青
链接：https://github.com/qianlinnian/Os_design.git

目录

Lab: Xv6 ..... **1**
目录 **2**
Lab: Xv6 环境搭建 **1**
1. 环境搭建 1
1）实验目的 1
2）实验步骤 1
3）实验中遇到的问题和解决方法 4
4）实验心得 5
Lab: Xv6 and Unix utilities **6**
1. Boot xv6(easy) 6
1）实验目的 6
2）实验步骤 6
2. Sleep(easy) 7
1）实验目的 7
2）实验步骤 7
3）实验中遇到的问题和解决方法 9
4）实验心得 9
3. pingpong(easy) 10
1）实验目的 10
2）实验步骤 10
3）实验中遇到的问题和解决方法 11
4）实验心得 11
4. Primes(moderate)/(hard) 12
1）实验目的 12
2）实验步骤 12
3）实验中遇到的问题和解决方法 13
4）实验心得 14
5. Find (moderate) 15
1）实验目的 15
2）实验步骤 15
3）实验中遇到的问题和解决方法 16
4）实验心得 17
6. Xargs(moderate) 17
1）实验目的 17
2）实验步骤 17
3）实验中遇到的问题和解决方法 19
4）实验心得 20

Lab: system calls **21**
1. System call tracing (moderate) 21
1）实验目的 21
2）实验步骤 21
3）实验中遇到的问题和解决方法 25
4）实验心得 25
2. Sysinfo (moderate) 27
1）实验目的 27
2）实验步骤 27
3）实验中遇到的问题和解决方法 31
4）实验心得 31
Lab: page tables **32**
1. Speed up system calls (easy) 32
1）实验目的 32
2）实验步骤 32
3）实验中遇到的问题和解决方法 34
4）实验心得 35
2. Print a page table (easy) 36
1）实验目的 36
2）实验步骤 36
3）实验中遇到的问题和解决方法 38
4）实验心得 38
3. Detecting which pages have been accessed (hard) 39
1）实验目的 39
2）实验步骤 39
3）实验中遇到的问题和解决方法 42
4）实验心得 43
Lab: traps **44**
1. RISC-V assembly (easy) 44
1）实验目的 44
2）实验步骤 44
3）实验中遇到的问题和解决方法 47
4）实验心得 47
2. Backtrace (moderate) 48
1）实验目的 48
2）实验步骤 48
3）实验中遇到的问题和解决方法 50
4）实验心得 50

3. Alarm (hard) 51
1）实验目的 51
2）实验步骤 51
3）实验中遇到的问题和解决方法 54
4）实验心得 54
Lab: Copy-on-Write Fork for xv6 55
1. Implement copy-on write(hard) 55
1）实验目标 55
2）实验步骤 55
3）实验中遇到的问题和解决方法 62
4）实验心得 62
Lab: Multithreading 63
1. Uthread: switching between threads (moderate) 63
1）实验目标 63
2）实验步骤 63
3）实验中遇到的问题和解决方法 68
4）实验心得 68
2. Using threads (moderate) 69
1）实验目标 69
2）实验步骤 69
3）实验中遇到的问题和解决方法 72
4）实验心得 73
3. Barrier(moderate) 74
1）实验目标 74
2）实验步骤 74
3）实验中遇到的问题和解决方法 75
4）实验心得 76
Lab: networking 77
1. Job (hard) 77
1）实验目标 77
2）实验步骤 77
3）实验中遇到的问题和解决方法 81
4）实验心得 81
Lab: locks 82
1. Memory allocator (moderate) 82
1）实验目标 82
2）实验步骤 82
3）实验中遇到的问题和解决方法 84

4）实验心得 84
2. Buffer cache (hard) 85
1）实验目标 85
2）实验步骤 85
3）实验中遇到的问题和解决方法 93
4）实验心得 94
Lab: file system 95
1. Large files (moderate) 95
1）实验目标 95
2）实验步骤 95
3）实验中遇到的问题和解决方法 99
4）实验心得 99
2. Symbolic links (moderate) 100
1）实验目标 100
2）实验步骤 100
3）实验中遇到的问题和解决方法 104
4）实验心得 105
Lab: mmap 106
1. JOP 106
1）实验目标 106
2）实验步骤 106
3）实验中遇到的问题和解决方法 117
4）实验心得 117

Lab：Xv6 环境搭建

1. 环境搭建

1）实验目的

搭建环境以便于后续实验进行。

2）实验步骤

由于试验过程中出现的种种问题，整体实验环境有不同的改变，起初仅仅使用 VMware Workstation 虚拟机软件进行实验，后为了提高操作便捷性，采用了 Vm 与 Vscode 结合，在之后，采用了 wsl 和 Vscode 进行实验，更加轻量化和快捷。

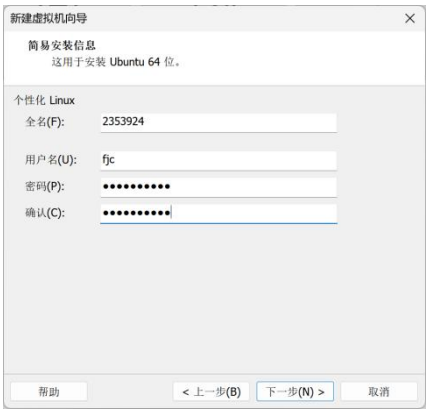
(1) VMware Workstation

① 下载 VMware Workstation

② 下载 Ubuntu 最新的长期支持的 iso 镜像

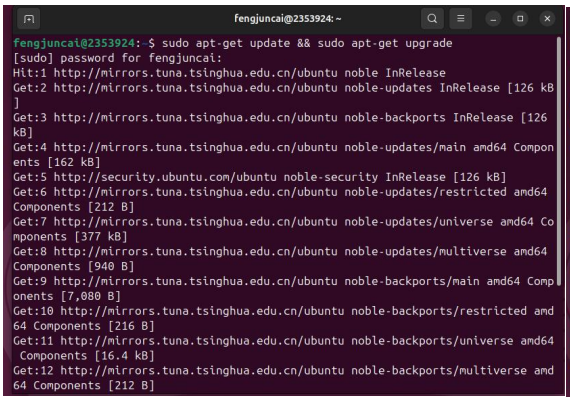
ubuntu-24.04.2-desktop-amd64.iso	2025/7/3 16:49	HaoZip.iso	6,194,550 KB

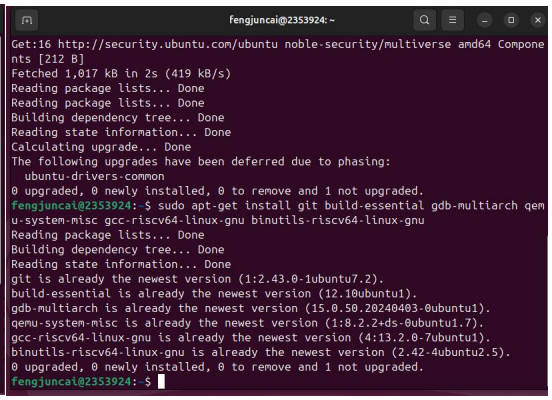
③ 配置基础设置



④ 配置所需环境

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

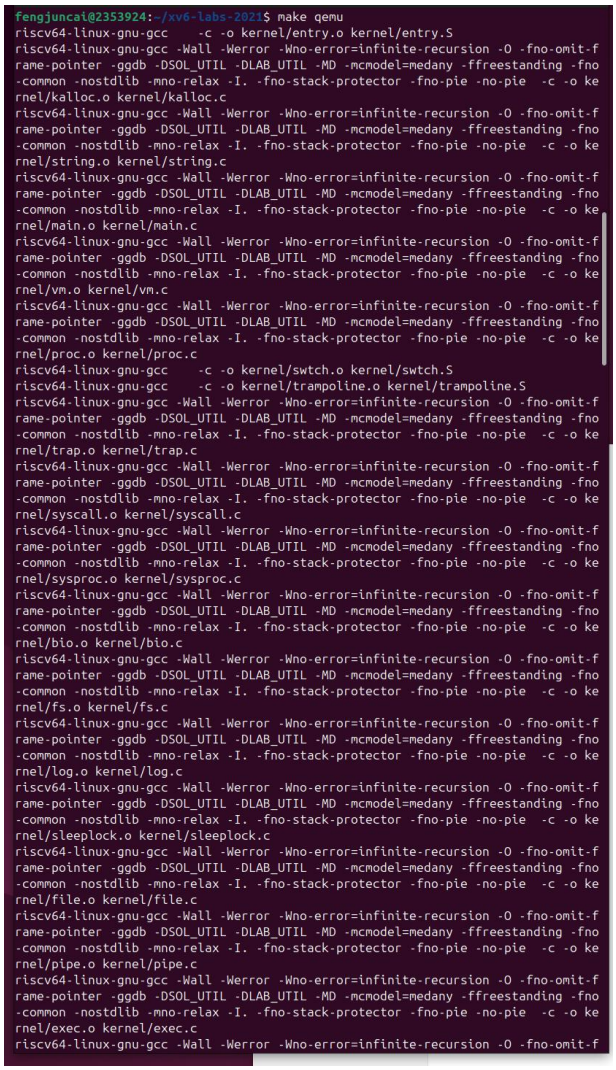





```
$ sudo apt install vim #下载 vim，便于编写代码
```

⑤ 测试安装

```
# in the xv6 directory
$ make qemu
# ... lots of output ...
init: starting sh
```



(2) Vscode 配置（基于 vm）

① Vscode 下载 ssh 插件

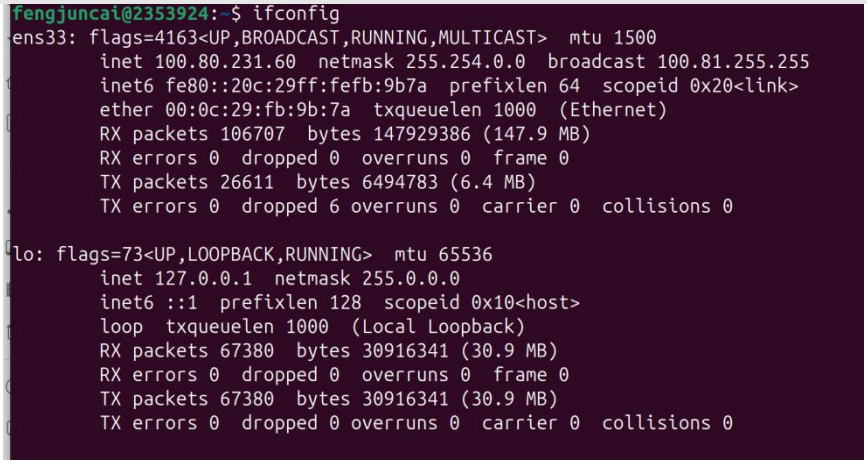
② VMware 虚拟机相关设置

将虚拟机网络设置改为桥接，并下载安装相关配置。

```
sudo apt update
sudo apt install openssh-server
sudo apt install net-tools
```

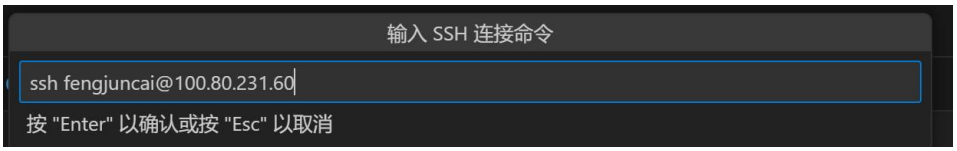
③ Vscode 和虚拟机进行连接

```
Ifconfig #得到 ip 地址
```

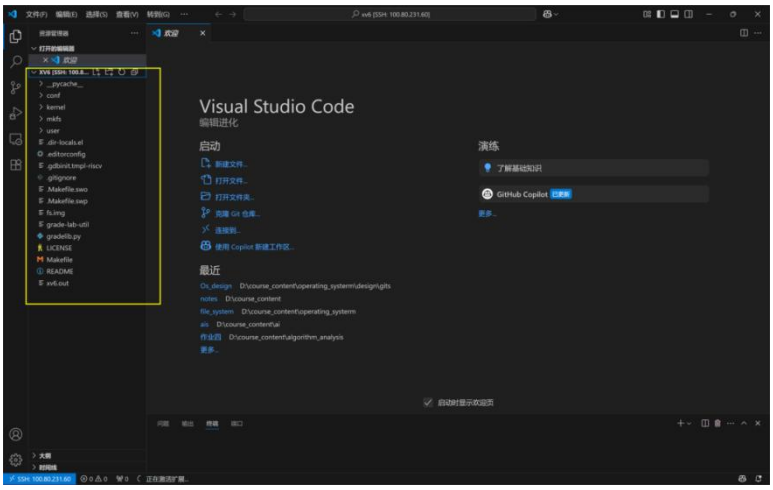


```
fengjuncai@2353924:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 100.80.231.60  netmask 255.254.0.0  broadcast 100.81.255.255
        inet6 fe80::20c:29ff:fefb:9b7a  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:fb:9b:7a  txqueuelen 1000  (Ethernet)
        RX packets 106707  bytes 147929386 (147.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 26611  bytes 6494783 (6.4 MB)
        TX errors 0  dropped 6 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 67380  bytes 30916341 (30.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 67380  bytes 30916341 (30.9 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

输入 ssh 连接命令进行连接，之后输入密码登录。



连接成功之后选择相应的文件夹打开，即可进行相应操作。



(3) Wsl 配置

① 打开 PowerShell（以管理员身份）

② 输入以下命令，启用 WSL 和虚拟机平台：

```
dism.exe /online /enable-feature
/featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

dism.exe /online /enable-feature
/featurename:VirtualMachinePlatform /all /norestart
```

③ 重启电脑

④ 安装 WSL 以及 Linux 发行版

在 PowerShell（管理员）中输入：

```
wsl --install
```

这会自动安装最新版的 WSL 和 Ubuntu。

⑤ 打开 ubuntu 注册账号。

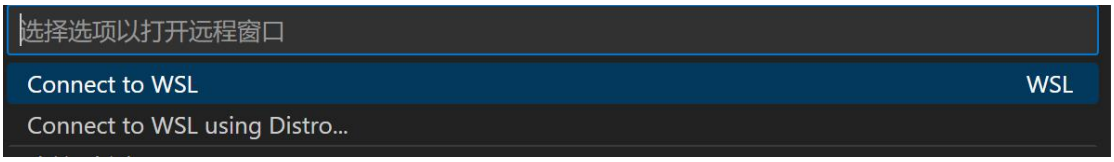
⑥ 将 xv6 的相应文件下载到 wsl 中的用户文件夹中。

⑦ 配置代理，以便访问 GitHub

(4) Vscode 配置（基于 wsl）

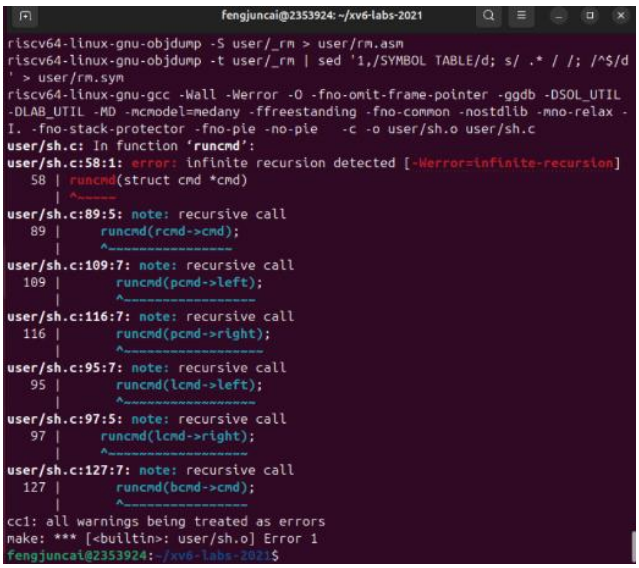
① 下载 wsl 插件

② 连接到 wsl，并选择打开相应的文件夹



③ 配置相应的 proxy

3）实验中遇到的问题和解决方法



GCC 编译器检测到了 user/sh.c 文件中的无限递归问题。

编辑 Makefile 文件，找到编译选项并添加忽略无限递归警告的标志：

```
$ nano Makefile
将 CFLAGS = -Wall -Werror -O -fno-omit-frame-pointer -ggdb
修改为：CFLAGS = -Wall -Werror -Wno-error=infinite-recursion -O
-fno-omit-frame-pointer -ggdb  并保存。
```

4）实验心得

通过本次 xv6 环境搭建实验，我学会了在 Ubuntu 系统中配置操作系统开发环境，包括安装交叉编译器、模拟器等必要工具。实验过程中遇到了 GCC 编译器的无限递归警告问题，通过修改 Makefile 中的编译选项成功解决，这让我认识到仔细分析错误信息和灵活使用编译选项的重要性。

整个环境搭建过程加深了我对 Linux 系统操作和开发工具使用的理解，也让我体会到良好的实验环境是完成后续复杂实验的重要基础。通过这次实践，我对即将开始的操作系统核心概念实验更有信心，期待通过后续的动手实践进一步理解操作系统的工作原理。

Lab: Xv6 and Unix utilities

1. Boot xv6(easy)

1）实验目的

启动 xv6 环境。

2）实验步骤

(1) Clone 相应代码

```
$ git clone git://g.csail.mit.edu/xv6-labs-2021
$git clone git://g.csail.mit.edu/xv6-labs-2021
克隆到"xv6-labs-2021"……
...
$ cd xv6-labs-2021
$ git checkout util
已设置为从"origin"跟踪远程分支"util"。
已切换到新分支"util"
```

(2) 运行代码

```
构建并运行 xv6：
$ make qemu
```

(3) 测试

可以用 make grade 评分程序测试解决方案。

2. Sleep(easy)

1）实验目的

本实验旨在通过实现 sleep 工具，熟悉 Xv6 操作系统的基本系统调用使用方法，理解进程的休眠与唤醒机制，加深对操作系统时钟和进程调度的理解。

2）实验步骤

(1) 编写代码

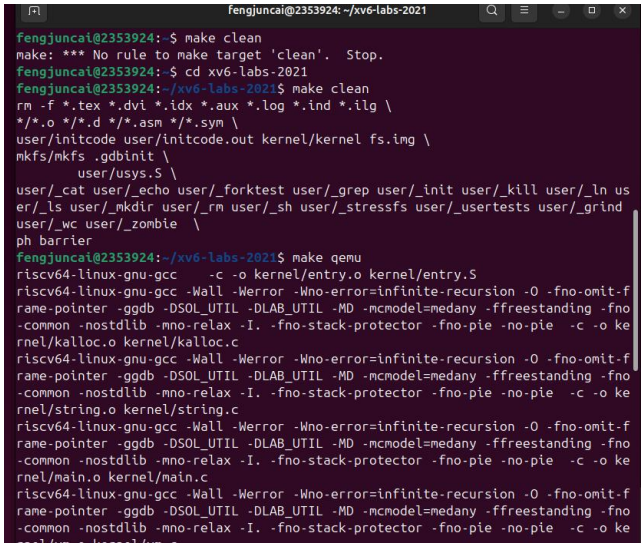
位置/user/sleep.c

```
fengjuncai@2353924: ~/xv6-labs-2021/user
#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);
  exit(0);
}
~
~
~
~
~
~
~
~
~
"sleep.c" [dos] 14L, 235B                    1,1           All
```

(2) 更新变量

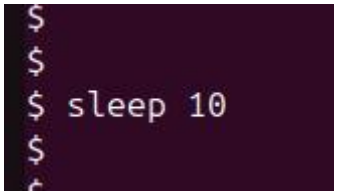
```
UPROGS=\
        $U/_cat\
        $U/_echo\
        $U/_forktest\S
        $U/_grep\
        $U/_init\
        $U/_kill\
        $U/_ln\
        $U/_ls\
        $U/_mkdir\
        $U/_rm\
        $U/_sh\
        $U/_stressfs\
        $U/_usertests\
        $U/_grind\
        $U/_wc\
        $U/_zombie\
        $U/_sleep\
```

(3) 重新编译

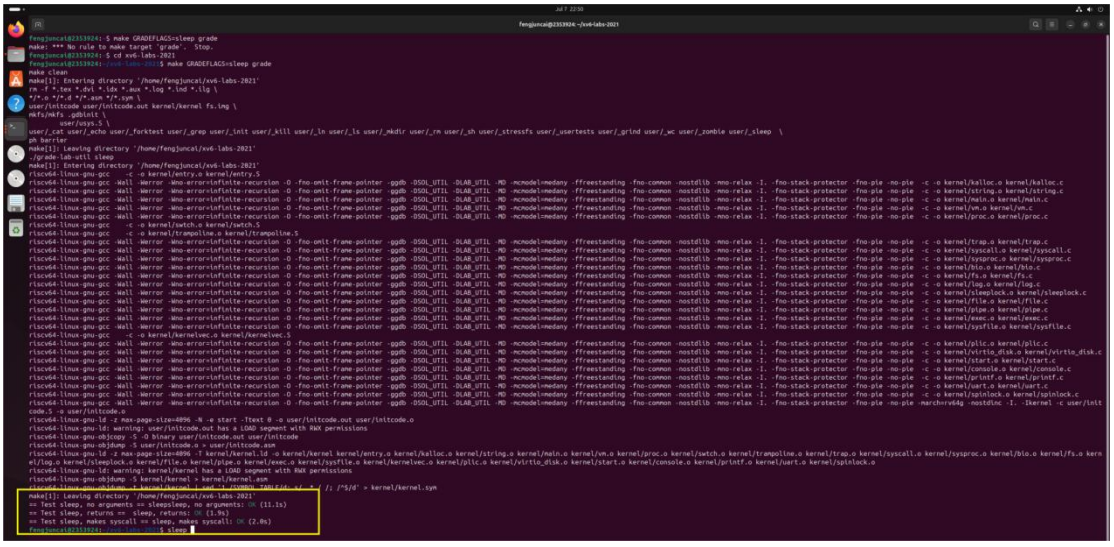


(1) 运行代码

```
$ cd xv6-labs-2021 # 进入相应的目录
$ make clean #清空
$ make qemu #重新编译
$ sleep 10 # 运行代码
```



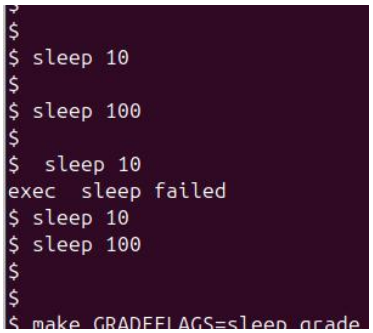
在 xv6 中，sleep 的单位是 “tick”，而不是秒。这里的 ticks 是时钟中断的次数。



运行官方测试程序后，返回 OK，通过测试。

3）实验中遇到的问题和解决方法

在运行 sleep 程序过程中，会突然报错。



可能原因分析：

(1) 可执行文件被删除或覆盖：xv6 的文件系统空间有限，用户目录下的文件如果被其他程序或测试覆盖，可能会导致 sleep 文件暂时不存在或损坏。

(2) 文件描述符或进程资源耗尽：如果系统中打开的文件太多，或者进程数达到上限，exec 加载新程序时会失败。但你的情况貌似不是持续性错误，后面又能运行，说明资源不是一直被占满。

(3) 并发或测试脚本干扰：如果你在运行自动测试（如 make grade）时手动操作 shell，可能会有并发，测试脚本可能会临时删除或替换 sleep，导致你手动执行时找不到。

(4) 文件系统不稳定或有 bug：xv6 的文件系统是教学用的，偶尔会有未同步的情况，极端情况下可能会有短暂的文件“丢失”。

解决建议：

(1) 避免在自动测试运行时手动运行 sleep：测试脚本有可能会清理或重建用户程序，导致你手动运行时偶尔找不到。

(2) 重启 xv6 或重新 make clean && make：如果频繁遇到此问题，可以完全重启 xv6 或重新编译、格式化文件系统，保证环境干净。

(3) 确认 sleep 是否真的存在：在 shell 下运行 ls 看看 sleep 是否在 / 或 /user 目录下。

(4) 减少并发操作：尽量不要在测试脚本运行时频繁手动操作 shell。

4）实验心得

xv6 的 sleep 实现是基于“tick 计数”，即每隔一段固定时间（比如 10ms 或 100ms）产生一次时钟中断，sleep(n) 就是让进程挂起 n 个 tick。

由于 tick 是离散的，而不是连续的，所以 sleep 的实际时间会有上下浮动。如果在 tick 快结束时调用 sleep，实际上会等到下一个 tick 才开始计数，所以会“多等”将近一个 tick。

如果在 tick 刚开始时调用 sleep，则误差最小。因此，sleep 的实际耗时通常是 “请求的时间” ± 一个 tick 的误差。

通过本次实验，我理解了 xv6 中 sleep 的时间机制基于 tick 计数而非精确的秒数，由于 tick 是离散的时钟中断，实际休眠时间会有±1 个 tick 的误差。这让我认识到操作系统时钟管理的复杂性，以及理论与实际实现之间的差异。通过这次实验，我对操作系统的进程调度和时钟中断机制有了更直观地认识。

3. pingpong(easy)

1）实验目的

本实验旨在通过实现 pingpong 程序，掌握 xv6 操作系统中的进程间通信机制，熟悉管道（pipe）、fork()系统调用的使用方法，理解父子进程之间的数据传递和同步机制，加深对进程创建和进程间通信的理解。

2）实验步骤

(1) 编写代码

位置 /user/pingpong.c

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

int
main()
{
  int p1[2], p2[2];
  pipe(p1); // p1[1]父写，p1[0]子读
  pipe(p2); // p2[1]子写，p2[0]父读

  int pid = fork();
  if(pid < 0){
    fprintf(2, "fork failed\n");
    exit(1);
  }

  if(pid == 0){
    // 子进程
    char buf;
    close(p1[1]); // 关闭写端
    close(p2[0]); // 关闭读端

    read(p1[0], &buf, 1);
    printf("%d: received ping\n", getpid());
    write(p2[1], "a", 1);

    close(p1[0]);
    close(p2[1]);
    exit(0);
  } else {
    // 父进程
    char buf;
    close(p1[0]); // 关闭读端
    close(p2[1]); // 关闭写端

    write(p1[1], "a", 1);
    read(p2[0], &buf, 1);
    printf("%d: received pong\n", getpid());

    close(p1[1]);
    close(p2[0]);
    wait(0);
    exit(0);
  }
}
~
~
```

父子进程间通信，父进程写入信息，子进程读到管道信息后，输出 ping，并向管道内写入信息；父进程读到信息后输出 pong。并且及时关闭通道以及结束进程，以避免程序持续运行，占用资源。

(2) 更新变量

```
$U/_sleep\
$U/_pingpong\
```

(3) 重新编译

```
fengjuncai@2353924: ~/xv6-labs-2021
fengjuncai@2353924:~$ cd xv6-labs-2021
fengjuncai@2353924:~/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
        user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln us
er/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie user/_sleep user/_pingpong\        \
ph barrier
fengjuncai@2353924:~/xv6-labs-2021$ make qemu
riscv64-linux-gnu-gcc    -c -o kernel/entry.o kernel/entry.S
riscv64-linux-gnu-gcc -Wall -Werror -Wno-error=infinite-recursion -O -fno-omit-f
```

(4) 运行代码

```
$ pingpong
6: received ping
5: received pong
$
```

```
$ make GRADEFLAGS=pingpong grade #测试程序
```

```
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/fengjuncai/xv6-labs-2021'
== Test pingpong ==
pingpong: OK (10.4s)
```

3）实验中遇到的问题和解决方法

```
fengjuncai@2353924:~/xv6-labs-2021$ make qemu
make: *** No rule to make target 'user/_pingpong\', needed by 'fs.img'.  Stop.
fengjuncai@2353924:~/xv6-labs-2021$ make qemu
make: *** No rule to make target 'user/_pingpong\', needed by 'fs.img'.  Stop.
fengjuncai@2353924:~/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
```

问题：makefile 里有一步需要生成 user/_pingpong 文件，但找不到相关的生成规则。但是我已经在 makefile 中添加了_pingpong\ 且文件命名正确。

解决方法：经检查，在 makefile 中_pingpong\ 后添加了没有意义的空格，删除后可正常运行。

4）实验心得

通过本次 pingpong 实验，我深入理解了 xv6 中进程间通信的基本原理。实验过程中最大的收获是掌握了管道机制，理解了父子进程如何通过 pipe 进行双向数据传递。遇到的 Makefile 配置问题让我认识到在系统编程中细节的重要性，一个多余的空格就能导致编译失败。

4. Primes(moderate)/(hard)

1）实验目的

本实验旨在通过实现并发版本的埃拉托斯特尼筛法（Sieve of Eratosthenes）来寻找素数，掌握 xv6 操作系统中的多进程编程技术，熟练运用管道（pipe）进行进程间通信，理解并发算法的设计思想，加深对进程创建、管道通信和进程同步的理解。

2）实验步骤

(1) 编写代码

位置：/user/primes.c

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

void primes(int fd) {
    int prime;
    int n;
    int p[2];

    // 读取第一个数，作为本轮素数
    if (read(fd, &prime, sizeof(prime)) != sizeof(prime)) {
        // 没有数字了，递归出口
        close(fd);
        exit(0);
    }
    printf("prime %d\n", prime);

    // 创建新管道
    pipe(p);

    if (fork() == 0) {
        // 子进程：递归处理新管道
        close(p[1]);        // 只读
        primes(p[0]);
        // primes会exit
    } else {
        // 父进程：筛选并写入新管道
        close(p[0]);        // 只写
        while (read(fd, &n, sizeof(n)) == sizeof(n)) {
            if (n % prime != 0) {
                write(p[1], &n, sizeof(n));
            }
        }
        // 关闭所有用过的fd
        close(fd);
        close(p[1]);
        wait(0); // 等待子进程
        exit(0);
    }
}

int main(void) {
    int p[2];
    pipe(p);

    if (fork() == 0) {
        // 子进程：负责递归筛选
        close(p[1]);
        primes(p[0]);
        // primes会exit
    } else {
        // 父进程：写2~35到管道
        close(p[0]);
        for (int i = 2; i <= 35; i++) {
            write(p[1], &i, sizeof(i));
        }
        close(p[1]);
        wait(0); // 等待子进程
    }
    exit(0);
}
                                                    1,1           All
```

① 创建初始进程，生成 2 到 35 的数字序列。

② 每个进程从管道读取数字，第一个数字作为素数，筛选掉该数字的倍数。

③ 创建子进程和新管道，将不能被该素数整除的数字传递给下一级。

④ 递归创建筛选进程，直到没有数字需要处理。

(2) 修改变量

```
$U/_pingpong\
$U/_primes\
```

(3) 重新编译

```
fengjuncai@2353924:~/xv6$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
        user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln us
er/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie user/_sleep user/_pingpong user/_primes  \
ph barrier
fengjuncai@2353924:~/xv6$ make qemu
riscv64-linux-gnu-gcc    -c -o kernel/entry.o kernel/entry.S
riscv64-linux-gnu-gcc -Wall -Werror -Wno-error=infinite-recursion -O -fno-omit-f
rame-pointer -ggdb -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -ffreestanding -fno
```

(4) 运行程序

```
init: starting sh
$
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
```

```
$ make GRADEFLAGS=primes grade #测试程序
```

```
make[1]: Leaving directory '/home/fengjuncai/xv6'
== Test primes == primes: OK (4.9s)
```

3）实验中遇到的问题和解决方法

父进程创建子进程后，如果不等待子进程结束，会产生僵尸进程。需要使用 wait() 来等待子进程结束，系统调用确保子进程资源被正确回收。

无限递归警告：

```
riscv64-linux-gnu-gcc -Wall -Werror -Wno-error=infinite-recursion -O -fno-omit-frame-poi
 -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-prote
ser/primes.o user/primes.c
user/primes.c: In function ‘primes’:
user/primes.c:4:6: warning: infinite recursion detected [-Winfinite-recursion]
    4 | void primes(int fd) {
      |      ^~~~~~
user/primes.c:23:9: note: recursive call
   23 |         primes(p[0]);
      |         ^~~~~~~~~~~~
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_primes user/prime
```

编译器只是静态分析，无法判断程序是否有递归出口，所以警告无限递归。实际运行不会有问题。

4）实验心得

通过本次 primes 实验，我深刻体会到了并发编程的复杂性和精妙之处。埃拉托斯特尼筛法的并发实现让我理解了如何将经典算法转化为多进程协作的形式。

实验中最大的挑战是管理多级管道和进程间的同步，每个进程既要作为数据的消费者，又要作为下一级进程的数据提供者。这种流水线式的处理方式展现了 UNIX 哲学中"做好一件事"的设计思想。

5. Find (moderate)

1）实验目的

本实验旨在通过实现简化版的 find 命令来掌握 xv6 操作系统中的文件系统操作，熟练运用目录遍历和文件状态查询等系统调用，理解递归算法在目录树遍历中的应用，加深对文件系统结构、目录操作和文件类型判断的理解。通过这个实验，我们能够体验递归算法解决树形结构问题的思想。

该 find 命令能在指定目录及其子目录中查找指定名称的文件，递归遍历目录结构，并且输出匹配文件的完整路径。

2）实验步骤

(1) 编写代码

位置：/user/find.c

```c
void find(char *path, char *target) {
    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;
    }

    if(st.type == T_FILE){
        // 只比较最后一段
        p = path + strlen(path);
        while(p >= path && *p != '/')
            p--;
        p++;
        if(strcmp(p, target) == 0)
            printf("%s\n", path);
    } else if(st.type == T_DIR){
        while(read(fd, &de, sizeof(de)) == sizeof(de)){
            if(de.inum == 0)
                continue;
            if(strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
                continue;
            int len = strlen(path);
            memmove(buf, path, len);
            if(path[len-1] != '/'){
                buf[len] = '/';
                len++;
            }
            memmove(buf + len, de.name, DIRSIZ);
            buf[len + DIRSIZ] = 0;
            if(stat(buf, &st) < 0)
                continue;
            find(buf, target);
        }
    }
    close(fd);
}
```

① 初始化：打开路径，获取文件状态

② 类型判断：区分文件和目录两种情况

③ 文件处理：提取文件名并比较

④ 目录处理：遍历目录项，递归搜索子目录

⑤ 资源管理：正确关闭文件描述符

⑥ 错误处理：各个环节都有相应的错误检查和处理

(2) 修改变量

```
$U/_primes\
$U/_find\
```

(3) 重新编译

```
fengjuncai@2353924:~/xv6$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
        user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/
user/_stressfs user/_usertests user/_grind user/_wc user/_zombi
\
ph barrier
fengjuncai@2353924:~/xv6$ make qemu
riscv64-linux-gnu-gcc    -c -o kernel/entry.o kernel/entry.S
riscv64-linux-gnu-gcc -Wall -Werror -Wno-error=infinite-recursi
```

(4) 执行代码

```
$ find / find
/find
$ find / sleep
/sleep
$ find / primes
$ find / primes
/primes
$ find / pingpong
/pingpong
$
```

```
$ make GRADEFLAGS=find grade #测试程序
```

```
make[1]: Leaving directory '/home/fengjuncai/xv6'
== Test find, in current directory == find, in current directory: OK (5.7s)
== Test find, recursive == find, recursive: OK (1.6s)
```

3）实验中遇到的问题和解决方法

问题：无法访问 user 等文件夹。

```
$ find / find
/find
$ find /user find
find: cannot open /user
$ find / sleep
/sleep
$ find / primes
/primes
$ find / pingpong
/pingpong
```

```
$ /ls
.              1 1 1024
..             1 1 1024
README         2 2 2226
xargstest.sh   2 3 93
cat            2 4 21088
echo           2 5 20032
forktest       2 6 12080
grep           2 7 23584
init           2 8 20896
kill           2 9 19920
ln             2 10 19832
ls             2 11 23296
mkdir          2 12 20008
rm             2 13 20000
sh             2 14 36008
stressfs       2 15 21016
usertests      2 16 136384
grind          2 17 33984
wc             2 18 21768
zombie         2 19 19440
sleep          2 20 19808
pingpong       2 21 20896
primes         2 22 21168
find           2 23 22496
console        3 24 0
```

在 xv6 操作系统的实验过程中，尝试在 shell 中访问 /user、/kernel 等目录时，系统提示找不到该目录或文件。起初以为这些目录应该和源码目录结构一致，通过查阅资料和实验，了解到 xv6 虚拟机中的文件系统是独立的，仅包含少量的用户程序和测试文件，并不包含源码目录中的 user/、kernel/ 等文件夹。因此，在虚拟机 shell 中访问这些目录会失败。

4）实验心得

通过这个实验，我对 xv6 文件系统有了更深入地理解。xv6 的文件系统结构相对简单，根目录下只包含基本的用户程序和系统文件，让我认识到了虚拟机环境与开发环境的区别。

实验中最大的收获是对递归算法的理解。目录树遍历本质上是深度优先搜索，递归是最自然的实现方式，体现了分而治之的算法思想。

文件系统编程比想象中复杂，看似简单的文件查找功能，实际需要考虑文件类型判断、路径拼接、错误处理等诸多细节，需要全面的系统编程知识。

6. Xargs(moderate)

1）实验目的

本实验旨在通过实现 xargs 命令来深入理解 Unix 管道机制和进程间通信。xargs 是一个强大的命令行工具，它从标准输入读取数据，将读取的内容作为参数传递给指定的命令执行。通过这个实验，我们将学会如何处理标准输入流、解析命令行参数、动态构建参数列表，以及使用 fork 和 exec 系统调用创建子进程执行命令。实验还将帮助我们理解 Unix"一切皆文件"的设计理念，掌握进程创建、参数传递和进程同步等核心概念，加深对操作系统进程管理机制的理解。

2）实验步骤

(1) 编写代码

位置：/user/xargs.c

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

// 去除参数两侧成对的引号（单引号或双引号）
void strip_quotes(char *s) {
    int len = strlen(s);
    if(len >= 2) {
        if((s[0] == '"' && s[len-1] == '") || (s[0] == '\'' && s[len-1] == '\'')) {
            // 向前拷贝
            memmove(s, s+1, len-2);
            s[len-2] = 0;
        }
    }
}
// 将"\n"转译为换行符
void unescape_newline(char *s) {
    char *src = s, *dst = s;
    while (*src) {
        if (src[0] == '\\' && src[1] == 'n') {
            *dst++ = ' ';  // 写入' '
            src += 2;       // 跳过 \n
        } else {
            *dst++ = *src++; // 正常拷贝
        }
    }
    *dst = '\0'; // 结尾加上结束符
}
```

```c
int main(int argc, char *argv[])
{
    char buf[512];
    char *args[MAXARG];
    int i;
    int n_flag = 0; // -n选项的值，0表示没有-n选项
    int cmd_start = 1; // 命令开始的索引


    // 解析命令行参数
    if(argc < 2) {
        fprintf(2, "Usage: xargs [-n num] command [args...]\n");
        exit(1);
    }

    // 检查是否有-n选项
    if(argc >= 3 && strcmp(argv[1], "-n") == 0) {
        n_flag = atoi(argv[2]);
        cmd_start = 3;
        if(argc < 4) {
            fprintf(2, "Usage: xargs [-n num] command [args...]\n");
            exit(1);
        }
    }

    // 复制命令行参数到args数组
    int base_arg_count = 0;
    for(i = cmd_start; i < argc; i++) {
        args[base_arg_count++] = argv[i];
    }

    // 逐行处理标准输入
    int buf_pos = 0;
    char c;
```

```c
while(read(0, &c, 1) == 1) {
    if(c == '\n') {
        // 处理一行输入
        buf[buf_pos] = 0; // 字符串结束符

        // 跳过空行
        if(buf_pos == 0) {
            buf_pos = 0;
            continue;
        }

         // 对每个输入参数做 unescape_newline
        for(int x = 0; x < buf_pos; x++) {
            //printf("处理前 %s\n ", buf);
            strip_quotes(buf);
            unescape_newline(buf);
            //printf("处理后 %s\n ", buf);
        }

        // 将输入行按空格分割
        char *input_args[MAXARG];
        int input_count = 0;

        char *p = buf;
        while(*p && input_count < MAXARG - base_arg_count - 1) {
            // 跳过前导空格
            while(*p == ' ' || *p == '\t') p++;
            if(*p == 0) break;

            input_args[input_count++] = p;

            // 找到下一个空格或字符串结尾
            while(*p && *p != ' ' && *p != '\t') p++;
            if(*p) {
                *p = 0;
                p++;
            }
        }

        // 根据-n选项决定如何执行
        if(n_flag > 0) {
            // 有-n选项，按指定数量分组执行
            for(int j = 0; j < input_count; j += n_flag) {
                // 构建参数列表
                int arg_count = base_arg_count;
                for(int k = j; k < j + n_flag && k < input_count; k++) {
                    args[arg_count++] = input_args[k];
                }
                args[arg_count] = 0;

                // fork并执行
                if(fork() == 0) {
                    exec(args[0], args);
                    fprintf(2, "xargs: exec %s failed\n", args[0]);
                    exit(1);
                } else {
                    wait(0);
                }
            }
        } else {
            // 没有-n选项，将所有输入参数一次性传递
            int arg_count = base_arg_count;
            for(int j = 0; j < input_count; j++) {
                args[arg_count++] = input_args[j];
            }
            args[arg_count] = 0;

            // fork并执行
            if(fork() == 0) {
                exec(args[0], args);
                fprintf(2, "xargs: exec %s failed\n", args[0]);
                exit(1);
            } else {
                wait(0);
            }
        }

        buf_pos = 0; // 重置缓冲区
    } else {
        // 将字符添加到缓冲区
        if(buf_pos < sizeof(buf) - 1) {
            buf[buf_pos++] = c;
        }
    }
}

exit(0);
```

① 参数准备：将命令行参数作为基础命令和参数

② 输入处理：从标准输入读取数据行

③ 参数扩展：将每行数据分割成参数，对引号和换行符进行额外处理，并且追加到基础参数后面

④ 命令执行：使用完整的参数列表执行命令

⑤ 循环处理：对每一行输入重复上述过程

(2) 修改变量

```
$U/_find\
$U/_xargs\
```

(3) 重新编译

```
fengjuncai@2353924:~/xv6$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img
mkfs/mkfs .gdbinit \
        user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_
/_sh user/_stressfs user/_usertests user/_grind user/
er/_find  \
ph barrier
fengjuncai@2353924:~/xv6$ make qemu
```

(4) 运行程序

运行结果：

```
hart 1 starting
hart 2 starting
init: starting sh
$ echo "1\n2" | xargs -n 1 echo line
line 1
line 2
$ echo hello too | xargs echo bye
bye hello too
$
```

```
$ sh < xargstest.sh
$ $ $ $ $ $ hello
hello
hello
$ $
```

```
$ make GRADEFLAGS=xargs grade #测试程序
```

```
ll.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/
exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/start.o kernel/console.o kern
el/printf.o kernel/uart.o kernel/spinlock.o
riscv64-linux-gnu-ld: warning: kernel/kernel has a LOAD segment with RWX permissions
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/fengjuncai/xv6'
== Test xargs == xargs: OK (13.9s)
fengjuncai@2353924:~/xv6$
```

```
$ make grade #整体测试程序
```

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (5.1s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.9s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (0.8s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (0.9s)
== Test primes ==
$ make qemu-gdb
primes: OK
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (0.9s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.2s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.3s)
== Test time ==
time: OK
Score: 100/100
```

3）实验中遇到的问题和解决方法

问题：实验中发现，xargs 无法正确处理输入参数中的引号和 \n 字符串。具体表现为：当输入参数包含引号时，未能去除引号；当输入参数中包含 \n 字符串时，未能将其正确转换为换行符，导致最终输出不符合预期。

实验中，通过运行 echo 发现，问题出在 echo 的设置比较简单，没有去除掉引号，以及没有将"\n"字符串转换为换行符，实现的功能比较简单，从而影响了 xargs 的实现。

```
$ echo "hello time "
"hello time "
$ echo hellowold
hellowold
$ echo "1\n2"
"1\n2"
$ echo -e "1\n2"
-e "1\n2"
$
```

解决方法：在 xargs 程序中，增加了去除引号（strip_quotes）和将 \n 字符串转换为空格或换行的函数（unescape_newline），从而实现了对特殊字符的正确处理，保证了输出的准确性。

4）实验心得

通过 xargs 实验，我对 Unix 管道机制有了更深入地理解。xargs 将标准输入转换为命令参数，体现了 Unix"小工具组合"的设计思想。实验中最大的收获是学会了字符串处理的复杂性，处理引号和转义字符看似简单，实际需要考虑很多细节。通过实现 strip_quotes 和 unescape_newline 函数，解决了特殊字符处理问题。

在进程管理方面，通过 fork 和 exec 的组合使用，我理解了 Unix 进程创建的基本模式。父进程创建子进程执行命令，然后等待子进程完成，这种方式简洁而有效。调试过程让我认识到问题定位的重要性，通过分析 echo 命令的限制，找到了问题根源并针对性地解决。

Lab: system calls

1. System call tracing (moderate)

1）实验目的

在本实验中，添加一个系统调用跟踪功能，该功能有助于调试后续的实验。通过实现 trace 系统调用，我们能够监控和记录指定进程及其子进程执行的系统调用，包括系统调用的名称、参数和返回值。这个功能对于理解程序运行机制、调试系统问题和分析程序行为具有重要意义。

实验任务的重点是“让 trace 这个系统调用能被用户程序调用”。

2）实验步骤

1. 系统调用基础设施

分配系统调用号：给 trace 分配唯一编号 22

用户态接口声明：在 user.h 中声明 int trace(int mask)

自动生成包装代码：通过 usys.pl 生成用户态到内核态的调用桥梁

2. 进程状态扩展

添加追踪掩码字段：在进程控制块中添加 trace_mask 字段

初始化和清理：在进程分配/释放时正确初始化 trace_mask

继承机制：fork 时子进程继承父进程的追踪设置

3. 内核实现

sys_trace 实现：接收用户传入的 mask 参数，设置当前进程的追踪掩码

系统调用名称映射：建立系统调用号到名称的映射表

追踪逻辑：在 syscall() 分发函数中添加判断，当系统调用匹配追踪掩码时打印信息

4. 编译集成

添加到构建系统：将 trace 用户程序加入 Makefile，使其能被编译和使用

具体实现：

(1) 切换分支：

```
$  git fetch
$ git checkout syscall
```

(2) 增加系统调用号，在 kernel/syscall.h 加一行：

```
#define SYS_trace 22   # 22 根据最大号为 21，所以可以选择 22
```

为 trace 系统调用分配一个唯一的编号（22），让内核能够识别“trace”这个新加的系统调用。

```
#define SYS_trace  22
#define SYS_sysinfo 23
```

(3) 用户头文件 user/user.h 文件中补充：

```
int trace(int mask);
```

声明 trace 系统调用的用户态接口，方便应用程序直接调用 trace()。

```
user.h
int trace(int mask);
```

(4) 用户态系统调用 stub，在 user/usys.pl 加一行：

```
entry("trace");
```

自动生成 trace 的系统调用包装代码(stub)，让用户程序能像普通函数一样调用 trace()，实际由内核实现。

```
usys.pl
entry("trace");
```

(5) 在 kernel/proc.h 里，给 struct proc 添加 trace_mask 字段：

```
uint64 trace_mask;
```

为每个进程保存一个 trace_mask，表示当前进程要追踪哪些系统调用（用掩码 bit 位表示）。

```
proc.h
struct proc {
  uint64 sz;
  uint64 trace_mask;
```

(6) 在 kernel/sysproc.c 里实现 sys_trace：

```
uint64 sys_trace(void)
{
  int mask;
  if (argint(0, &mask) < 0)
    return -1;
  myproc()->trace_mask = mask;
  return 0;
}
```

实现 trace 系统调用的内核端逻辑，把用户传来的 mask 保存在当前进程的 trace_mask 字段里。

```
sysproc.c
uint64
sys_trace(void)
{
  int mask;
  if (argint(0, &mask) < 0)
    return -1;
  myproc()->trace_mask = mask;
  return 0;
}
```

(7) fork 时继承 trace_mask：

```
np->trace_mask = p->trace_mask;
```

让新 fork 出来的子进程继承父进程的 trace_mask，保持追踪设置一致。

```
safestrcpy(np->name, p->name, sizeof(p->name));
np->trace_mask = p->trace_mask;
pid = np->pid;
```

(8) kernel/syscall.c 中增加 syscall 名字数组：

```
static char *syscall_names[] = {
  [SYS_fork]    "fork",
  [SYS_exit]    "exit",
  [SYS_wait]    "wait",
  [SYS_pipe]    "pipe",
  [SYS_read]    "read",
  [SYS_kill]    "kill",
  [SYS_exec]    "exec",
  [SYS_fstat]   "fstat",
  [SYS_chdir]   "chdir",
  [SYS_dup]    "dup",
  [SYS_getpid]  "getpid",
  [SYS_sbrk]    "sbrk",
  [SYS_sleep]   "sleep",
  [SYS_uptime]  "uptime",
  [SYS_open]    "open",
  [SYS_write]   "write",
  [SYS_mknod]   "mknod",
  [SYS_unlink]  "unlink",
  [SYS_link]    "link",
  [SYS_mkdir]   "mkdir",
  [SYS_close]   "close",
  [SYS_trace]   "trace",
};
```

```
static char *syscall_names[] = {
  [SYS_fork]    "fork",
  [SYS_exit]    "exit",
  [SYS_wait]    "wait",
  [SYS_pipe]    "pipe",
  [SYS_read]    "read",
  [SYS_kill]    "kill",
  [SYS_exec]    "exec",
  [SYS_fstat]   "fstat",
  [SYS_chdir]   "chdir",
  [SYS_dup]    "dup",
  [SYS_getpid]  "getpid",
  [SYS_sbrk]    "sbrk",
  [SYS_sleep]   "sleep",
  [SYS_uptime]  "uptime",
  [SYS_open]    "open",
  [SYS_write]   "write",
  [SYS_mknod]   "mknod",
  [SYS_unlink]  "unlink",
  [SYS_link]    "link",
  [SYS_mkdir]   "mkdir",
  [SYS_close]   "close",
  [SYS_trace]   "trace",
  [SYS_sysinfo]  "sysinfo",
};
```

用数组保存每个系统调用的名字，方便后面打印追踪信息时显示人类可读的系统调用名。并且声明 sys_trace 以及在 syscall 分发表即 syscalls[]中添加 trace 相应的信息。

```
extern uint64 sys_trace(void);
```

```
[SYS_close]   sys_close,
[SYS_trace]   sys_trace,
```

(9) 在 syscall() 函数里加判断和输出：

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();

    // 只有在有效的系统调用执行后才检查 trace
    if((p->trace_mask & (1 << num)) != 0) {
      printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], p->trapframe->a0);
    }
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

每当有系统调用发生，就判断 trace_mask 是否要追踪这个调用，如果需要就打印出调用的名字、返回值等信息，实现“系统调用追踪”。

(10) Kernel/proc.c 的 allocproc(void)和 freeproc(struct proc *p)中补充：

```
p->trace_mask = 0;
```

```
p->trace_mask = 0;
```

(11) Makefile 中加 _trace\ 到 UPROGS 列表：

让 user/trace.c 能被自动编译并包含进 xv6 镜像，方便在 xv6 shell 里直接运行 trace 测试程序。

```
Makefile
UPROGS=\
$U/_wc\
$U/_zombie\
$U/_trace\
```

(12) 运行程序：

单独运行 trace

```
$ trace 2 ls
.              1 1 1024
..             1 1 1024
README         2 2 2226
cat            2 3 27640
echo           2 4 26296
forktest       2 5 15624
grep           2 6 31440
init           2 7 27048
kill           2 8 26192
ln             2 9 26032
ls             2 10 30216
mkdir          2 11 26296
rm             2 12 26288
sh             2 13 47840
stressfs       2 14 27304
usertests      2 15 168240
grind          2 16 42304
wc             2 17 28728
zombie         2 18 25512
trace          2 19 26600
console        3 20 0
```

```
$ make GRADEFLAGS=trace grade #测试程序
trace 32 grep hello README
trace 2147483647 grep hello README
trace 2 usertests forkforkfork
```

```
$ trace 32 grep hello README
18: syscall read -> 1023
18: syscall read -> 968
18: syscall read -> 235
18: syscall read -> 0
$ trace 2147483647 grep hello README
19: syscall trace -> 0
19: syscall exec -> 3
19: syscall open -> 3
19: syscall read -> 1023
19: syscall read -> 968
19: syscall read -> 235
19: syscall read -> 0
19: syscall close -> 0
$ trace 2 usertests forkforkfork
usertests starting
20: syscall fork -> 21
test forkforkfork: 20: syscall fork -> 22
22: syscall fork -> 23
OK
20: syscall fork -> 24
ALL TESTS PASSED
```

```
make[1]: Leaving directory '/home/fengjuncai/xv6'
== Test trace 32 grep == trace 32 grep: OK (5.2s)
== Test trace all grep == trace all grep: OK (1.1s)
== Test trace nothing == trace nothing: OK (0.9s)
== Test trace children == trace children: OK (13.4s)
    (Old xv6.out.trace_children failure log removed)
```

3）实验中遇到的问题和解决方法

```
make[1]: Leaving directory '/home/fengjuncai/xv6'
== Test trace 32 grep == trace 32 grep: OK (11.4s)
== Test trace all grep == trace all grep: OK (2.1s)
== Test trace nothing == trace nothing: OK (2.2s)
== Test trace children == Timeout! trace children: FAIL (30.6s)
    ...
        14: syscall fork -> -1
        7: syscall fork -> -1
        OK
        3: syscall fork -> 18
        qemu-system-riscv64: terminating on signal 15 from pid 52110 (make)
    MISSING '^ALL TESTS PASSED'
    QEMU output saved to xv6.out.trace_children
make: *** [Makefile:338: grade] Error 1
fengjuncai@2353924:~/xv6$
```

问题：实验测试程序总是缺少”ALL TESTS PASSED”的输出，但是单独运行的时候往往没有问题。

解决方法：查找原因可能是运行时间过长，来不及输出。同时 wsl 更加轻量化，将环境切换为 wsl 后，能够通过测试。

4）实验心得

通过这次系统调用跟踪实验，我对操作系统内核的工作机制有了更深入地理解。实验让我亲手体验了从用户态到内核态的完整系统调用流程，从在用户头文件中声明接口，到在内核中实现具体功能，每一步都加深了我对操作系统架构的认识。

最有收获的是理解了系统调用的分层设计。用户程序通过简单的函数调用接口，经过 usys.pl 生成的包装代码，最终到达内核中的具体实现函数。这种分层抽象既保证了用户程序

的简洁性，又维护了内核的安全性。特别是通过修改 syscall.c 中的分发机制，我明白了内核如何通过系统调用号来路由到对应的处理函数。

进程管理方面的学习也很重要。通过在 proc 结构体中添加 trace_mask 字段，并在 fork 时实现继承机制，我理解了进程控制块的作用和进程间状态传递的方式。这种设计让 trace 功能能够自然地传递给子进程，体现了 Unix 系统设计的一致性。

实验中遇到的测试超时问题让我认识到系统编程中性能的重要性。通过切换到更轻量的 WSL 环境解决问题，也让我学会了在不同环境下调试系统级程序的技巧。整个实验不仅让我掌握了系统调用的实现方法，更重要的是培养了系统级编程的思维方式，为后续更复杂的内核实验奠定了坚实基础。

2. Sysinfo (moderate)

1）实验目的

学会如何实现一个返回复杂数据结构的系统调用，掌握内核中内存管理和进程管理的相关函数调用，理解如何在内核态和用户态之间安全地传递数据结构，以及学习如何遍历系统资源并统计相关信息。

2）实验步骤

1. 系统调用基础设施

分配系统调用号：给 sysinfo 分配唯一编号 23

注册系统调用：在内核的系统调用分发表中添加 sys_sysinfo 入口

用户态接口：声明用户态调用接口和数据结构

自动生成包装代码：通过 usys.pl 生成调用桥梁

2. 核心功能实现

空闲内存统计（freemem）

进程数量统计（nproc）

3. 系统调用实现

4. 数据结构

需要定义 struct sysinfo 结构体（通常在 kernel/sysinfo.h 中）：

具体实现：

(1) 添加系统调用号

文件：kernel/syscall.h

```
#define SYS_sysinfo 23
```

```
#define SYS_sysinfo 23
```

(2) 添加系统调用入口

文件：kernel/syscall.c

```
// 在 extern 声明部分添加
extern uint64 sys_sysinfo(void);
// 在 syscalls 数组中添加
static uint64 (*syscalls[])(void) = {
[SYS_sysinfo] sys_sysinfo,  // 添加这一行
};
Static uint64 *syscall_names[] = {
[SYS_sysinfo] "sys_sysinfo",  // 添加这一行
}
```

```
extern uint64 sys_sysinfo(void);
```

```
[SYS_sysinfo]   sys_sysinfo,
```

```
[SYS_sysinfo]  "sysinfo",
```

(3) 添加用户空间接口

文件：user/user.h

```
// 在文件中添加
struct sysinfo;
int sysinfo(struct sysinfo*);
```

```
struct sysinfo;
```

```
int sysinfo(struct sysinfo* info);
```

文件：user/usys.pl

```
// 在文件中末尾添
Entry("sysinfo");
```

```
entry("sysinfo");
```

(4) 实现获取空闲内存的函数

文件：kernel/kalloc.c

```
// 在文件末尾添加
uint64
freemem(void)
{
  struct run *r;
  uint64 count = 0;

  acquire(&kmem.lock);
  r = kmem.freelist;
  while(r) {
    count++;
    r = r->next;
  }
  release(&kmem.lock);
  return count * PGSIZE;
}
```

```
uint64
freemem(void)
{
  struct run *r;
  uint64 count = 0;

  acquire(&kmem.lock);
  r = kmem.freelist;
  while(r) {
    count++;
    r = r->next;
  }
  release(&kmem.lock);
  return count * PGSIZE;
}
```

(5) 实现获取进程数量的函数

文件：kernel/proc.c

```
// 在文件末尾添加
uint64 nproc(void)
{
  struct proc *p;
  uint64 count = 0; // 初始化 count 为 0，用于统计非未使用状态的进程数量
  // 遍历进程表中的所有进程槽位
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);   // 获取当前进程 p 的锁
    if(p->state != UNUSED) {  // 检查进程状态是否不等于 UNUSED（未使用）
      count++;          // 如果进程正在使用中，计数器加 1
    }
    release(&p->lock);   // 释放当前进程 p 的锁
  }
  return count;          // 返回统计到的活跃进程总数
}
```

```
uint64
nproc(void)
{
  struct proc *p;
  uint64 count = 0;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state != UNUSED) {
      count++;
    }
    release(&p->lock);
  }
  return count;
}
```

(6) 添加函数声明

文件：kernel/defs.h

```
// kalloc.c
uint64          freemem(void);  // 添加这一行
// proc.c
uint64          nproc(void);    // 添加这一行
```

```
uint64          freemem(void);
```

```
uint64          nproc(void);
```

(7) 实现系统调用函数

文件：kernel/sysproc.c

```
#include "sysinfo.h"  // 在文件顶部添加
// 在文件末尾添加 sys_sysinfo 函数
uint64
sys_sysinfo(void)
{
  uint64 addr;
  struct sysinfo info;
  struct proc *p = myproc();
  // 获取用户传入的地址参数
  if(argaddr(0, &addr) < 0)
    return -1;
    // 收集系统信息
  info.freemem = freemem();
  info.nproc = nproc();
    // 将信息复制到用户空间
  if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
    return -1;
  return 0;
}
```

```
#include "sysinfo.h"
```

```
uint64
sys_sysinfo(void)
{
  uint64 addr;
  struct sysinfo info;
  struct proc *p = myproc();

  // 获取用户传入的地址参数
  if(argaddr(0, &addr) < 0)
    return -1;

  // 收集系统信息
  info.freemem = freemem();
  info.nproc = nproc();

  // 将信息复制到用户空间
  if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
    return -1;

  return 0;
}
```

(8) Makefile 中补充信息

```
$U/_sysinfotest\
```

```
$U/_sysinfotest\
```

(9) 编译测试

```
$ make GRADEFLAGS=sysinfo grade
```

```
make[1]: Leaving directory '/home/fengjuncai/xv6'
== Test sysinfotest == sysinfotest: OK (8.1s)
    (Old xv6.out.sysinfotest failure log removed)
fengjuncai@Qln:~/xv6$ make GRADEFLAGS=sysinfo grade
```

```
$ make grade
```

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (4.5s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.1s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (18.3s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (2.8s)
== Test time ==
time: OK
Score: 35/35
fengjuncai@Qln:~/xv6$
```

3）实验中遇到的问题和解决方法

```
riscv64-linux-gnu-ld: warning: user/_trace has a LOAD segment with RWX permissions
make[1]: *** No rule to make target 'user/_sysinfo', needed by 'fs.img'.  Stop.
```

表明 Makefile 试图构建一个名为 user/_sysinfo 的目标，但找不到相应的规则。

问题原因：

在 Makefile 中错误地添加了：$U/_sysinfo\。

sysinfo 是一个系统调用，不是一个独立的用户程序。它的作用类似于 fork()、exec() 这些系统调用。Makefile 中需要添加的是测试程序 sysinfotest。

解决方法：

Makefile 实际上添加：$U/_sysinfotest\

4）实验心得

通过 sysinfo 系统调用的实现，我学习了系统资源监控机制。这次实验使我理解了内核中系统状态信息的收集方法，以及如何将这些信息安全地传递到用户程序。

关键收获在于理解了内核态与用户态间的数据传输原理。利用 copyout 函数实现 sysinfo 结构体从内核到用户空间的安全复制，让我认识到操作系统维护内存访问安全的重要性，有效防止了用户程序对内核内存的直接访问。

实现过程中，系统资源的遍历统计方法给我很大启发。空闲内存统计通过 freemem 函数遍历内存链表完成，活跃进程统计则由 nproc 函数遍历进程表实现，这些操作让我对 xv6 的资源管理机制有了更深层次的认识。

Makefile 配置过程中出现的问题帮助我区分了系统调用与用户程序的概念，明确了 sysinfo 作为系统调用接口与 sysinfotest 作为测试程序的不同作用。这次实验显著提升了我的内核编程能力。

Lab: page tables

1. Speed up system calls (easy)

1）实验目的

一些操作系统（例如 Linux）通过在用户空间和内核之间的只读区域中共享数据来加速某些系统调用。

实验的主要目的是实现一个共享页面机制，让某些系统调用能够通过用户态和内核态共享的内存页面来快速获取信息，从而避免频繁的上下文切换开销。具体来说，我们需要在用户进程的虚拟地址空间中映射一个只读的共享页面，内核可以在这个页面中放置一些系统信息（如进程 ID 等），用户程序可以直接读取而无需进行系统调用。

2）实验步骤

(1) 实现共享页面机制

在 kernel/proc.c 中修改进程创建代码：

```
// 在 allocproc()函数中添加
static struct proc*
allocproc(void)
{
  struct proc *p;
  // ... 现有代码 ...
  // 分配用户信息页面
  if((p->usyscall = (struct usyscall*)kalloc()) == 0){
    freeproc(p);//如果分配失败 释放资源和锁
    release(&p->lock);
    return 0;
  }
  p->usyscall->pid = p->pid;//存储 pid
  return p;
}
```

```
if((p->usyscall = (struct usyscall*)kalloc()) == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
p->usyscall->pid = p->pid;
```

(2) 映射共享页面到用户空间

在 kernel/proc.c 的 proc_pagetable() 函数中：

```
pagetable_t
proc_pagetable(struct proc *p)//为进程 p 创建页表
{
  pagetable_t pagetable;  // ... 现有代码 ...
  // 映射 usyscall 页面
  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmunmap(pagetable, TRAPFRAME, 1, 0);//取消映射
    uvmfree(pagetable, 0);//释放资源
    return 0;
  }
  return pagetable;
}
```

```
if(mappages(pagetable, USYSCALL, PGSIZE,
            (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmfree(pagetable, 0);
  return 0;
}
```

(3) 清理资源

在 kernel/proc.c 中修改清理函数：

```
static void
freeproc(struct proc *p){
  if(p->usyscall)
    kfree((void*)p->usyscall);
  p->usyscall = 0;
  // ... 其他清理代码 ...
}
Void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
//释放进程的用户页表
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmfree(pagetable, sz);
}
```

```
if(p->usyscall)
  kfree((void*)p->usyscall);
p->usyscall = 0;
```

```
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmfree(pagetable, sz);
}
```

(4) 定义数据结构

在 kernel/memlayout.h 中定义地址：

```
#define USYSCALL (TRAPFRAME - PGSIZE)
```

在 kernel/proc.h 的 struct proc 中添加 usyscall 字段：

```
struct usyscall *usyscall;
```

```
#define USYSCALL (TRAPFRAME - PGSIZE)
```

```
struct usyscall *usyscall;
```

(5) 测试

```
Way 1:
$ make qemu
$ pgtbltest
Way 2:
$ make GRADEFLAGS=pgtbltest grade
```

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
```

3）实验中遇到的问题和解决方法

问题，一个内核恐慌，导致用户程序访问未映射的内存地址。

```
hart 1 starting
hart 2 starting
panic: uvmunmap: not mapped
```

原因：缺少了映射的步骤

解决方法：在 proc_pagetable() 中添加映射代码，确保 USYSCALL 页面正确映射到用户地址空间

4）实验心得

通过实现系统调用加速机制，我理解了操作系统性能优化的核心思想。传统系统调用需要用户态与内核态频繁切换，开销较大，而共享内存机制可以让用户程序直接访问系统信息，避免系统调用开销。

实验中掌握了虚拟内存管理的关键技术，包括页面分配、地址映射和权限控制。特别是页表项标志位的设置，既保证了用户程序的读取权限，又维护了系统安全。

同时认识到内存资源管理的重要性，必须在进程创建和销毁时正确分配和释放资源，避免内存泄漏，确保系统稳定性。

2. Print a page table (easy)

1）实验目的

为了帮助可视化 RISC-V 页表，本实验的目的是编写一个打印页表内容的函数，用于调试和理解 xv6 操作系统的虚拟内存管理机制。通过打印页表的层次结构，可以直观地观察虚拟地址到物理地址的映射关系，帮助理解三级页表的工作原理和内存布局。

2）实验步骤

(1) 在 kernel/vm.c 添加函数实现

```
void
vmprint_recursive(pagetable_t pagetable, int level)
{
  // 遍历页表的 512 个条目
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if(pte & PTE_V){  // 如果页表项有效
      // 打印缩进
      for(int j = 0; j <= level; j++){
        printf(".. ");
      }
      // 打印页表项信息
      uint64 pa = PTE2PA(pte);
      printf("%d: pte %p pa %p\n", i, pte, pa);

      // 如果不是叶子节点，继续递归
      if((pte & (PTE_R|PTE_W|PTE_X)) == 0){
        uint64 child = PTE2PA(pte);
        vmprint_recursive((pagetable_t)child, level + 1);
      }
    }
  }
}
void
vmprint(pagetable_t pagetable)
{
  printf("page table %p\n", pagetable);
  vmprint_recursive(pagetable, 0);
}
```

```c
void
vmprint_recursive(pagetable_t pagetable, int level)
{
  // 遍历页表的512个条目
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if(pte & PTE_V){  // 如果页表项有效
      // 打印缩进
      for(int j = 0; j <= level; j++){
        printf(".. ");
      }
      // 打印页表项信息
      uint64 pa = PTE2PA(pte);
      printf("%d: pte %p pa %p\n", i, pte, pa);

      // 如果不是叶子节点，继续递归
      if((pte & (PTE_R|PTE_W|PTE_X)) == 0){
        uint64 child = PTE2PA(pte);
        vmprint_recursive((pagetable_t)child, level + 1);
      }
    }
  }
}
```

```c
void
vmprint(pagetable_t pagetable)
{
  printf("page table %p\n", pagetable);
  vmprint_recursive(pagetable, 0);
}
```

(2) 在 kernel/defs.h 添加函数声明

```
// vm.c
void            vmprint(pagetable_t);  // 添加
```

```c
void            vmprint(pagetable_t);
```

(3) 在 kernel/exec.c 添加调用

```
// 在 exec() 函数的最后，return argc; 之前添加
if(p->pid == 1)
  vmprint(p->pagetable);
```

```c
if(p->pid == 1)
  vmprint(p->pagetable);

return argc; // this ends up in
```

(4) 编译测试

```
Make qemu
```

```
hart 1 starting
hart 2 starting
page table 0x0000000087f6e000
.. 0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. .. 0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. .. 0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. .. 1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. .. 2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
.. 255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. .. 511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. .. 509: pte 0x0000000021fdd813 pa 0x0000000087f76000
.. .. .. 510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. .. 511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$
$
```

3）实验中遇到的问题和解决方法

问题：vmprint_recursive 函数是否需要 defs.h 中声明。

原因：函数作用域：vmprint_recursive 是一个内部辅助函数，只在 vm.c 文件内部被 vmprint 函数调用，不需要被其他文件访问。

解决：不需要在 defs.h 中声明。但是代码中要注意代码定义的顺序，因为 vmprint 函数需要调用 vmprint_recursive 函数，所以需要先定义递归函数，或者向前声明。

4）实验心得

通过完成这个页表打印实验，我对 RISC-V 的三级页表结构有了更直观的理解。当看到程序输出的层次化页表内容时，原本抽象的虚拟地址到物理地址转换过程变得清晰可见。递归遍历页表的实现让我深刻体会到了树状数据结构在系统编程中的应用。

在编程过程中遇到的函数声明顺序问题让我认识到 C 语言编译的特点，学会了合理使用 static 关键字和前向声明。通过位操作判断页表项类型的实现，让我感受到系统编程中每个 bit 都至关重要的精确性。

将 vmprint 调用添加到 exec 函数中，让我能够观察到进程创建时的真实内存布局，这种可视化调试方法比单纯看代码更加有效。这个实验不仅帮助我掌握了具体的技术实现，更重要的是培养了系统性思维和调试能力。

3. Detecting which pages have been accessed (hard)

1）实验目的

实验目标是实现 pgaccess()系统调用，该系统调用用于报告哪些页面已被访问过。这个系统调用接受三个参数：首先，它接受要检查的第一个用户页面的起始虚拟地址；其次，它接受要检查的页面数量；最后，它接受一个用户地址，指向用于存储结果的缓冲区，结果以位掩码的形式存储（位掩码是一种数据结构，每个页面使用一个位来表示，其中第一个页面对应最低有效位）旨在深入理解操作系统虚拟内存管理的核心机制。

2）实验步骤

(1) 在 kernel/riscv.h 中添加 PTE_A 定义

```
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_A (1L << 6) // 添加这一行 - Access bit
```

```
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_A (1L << 6)
```

(2) 在 kernel/sysproc.c 中实现系统调用

```
#ifdef LAB_PGTBL
int
sys_pgaccess(void)
{
  uint64 base;
  int len;
  uint64 mask;
  if(argaddr(0, &base) < 0)//获取起始地址
    return -1;
  if(argint(1, &len) < 0)//获取长度
    return -1;
  if(argaddr(2, &mask) < 0)//获取缓冲区地址
    return -1;
return pgaccess(base, len, mask);// 调用内核实现函数执行实际的页面访问检测
}
#endif
```

```c
#ifdef LAB_PGTBL
int
sys_pgaccess(void)
{
  // 获取参数
  uint64 base;
  int len;
  uint64 mask;
  if(argaddr(0, &base) < 0)
    return -1;
  if(argint(1, &len) < 0)
    return -1;
  if(argaddr(2, &mask) < 0)
    return -1;

  return pgaccess(base, len, mask);
}
#endif
```

(3) 在 kernel/vm.c 中实现 pgaccess() 函数

```c
#ifdef LAB_PGTBL
Int pgaccess(uint64 base, int len, uint64 mask)
{
  struct proc *p = myproc();
  uint64 result = 0;

  // 检查参数有效性
  if(len > 64) return -1;  // 最多 64 页

  for(int i = 0; i < len; i++){
    uint64 va = base + i * PGSIZE;
    pte_t *pte = walk(p->pagetable, va, 0);

    if(pte == 0) continue;  // 页面不存在
    if((*pte & PTE_V) == 0) continue;  // 页面无效

    // 检查访问位
    if(*pte & PTE_A){
      result |= (1L << i);  // 设置对应位
      *pte &= ~PTE_A;       // 清除访问位
    }
  }

  // 将结果复制到用户空间
  if(copyout(p->pagetable, mask, (char*)&result, sizeof(result)) < 0)
    return -1;
```

```
  return 0;
}
#endif
```

```
#ifdef LAB_PGTBL
int
pgaccess(uint64 base, int len, uint64 mask)
{
  struct proc *p = myproc();
  uint64 result = 0;

  // 检查参数有效性
  if(len > 64) return -1;  // 最多64页

  for(int i = 0; i < len; i++){
    uint64 va = base + i * PGSIZE;
    pte_t *pte = walk(p->pagetable, va, 0);

    if(pte == 0) continue;  // 页面不存在
    if((*pte & PTE_V) == 0) continue;  // 页面无效

    // 检查访问位
    if(*pte & PTE_A){
      result |= (1L << i);  // 设置对应位
      *pte &= ~PTE_A;       // 清除访问位
    }
  }

  // 将结果复制到用户空间
  if(copyout(p->pagetable, mask, (char*)&result, sizeof(result)) <
    return -1;

  return 0;
}
#endif
```

(4) 在 kernel/defs.h 中添加函数声明

```
//vm.c
#ifdef LAB_PGTBL
int            pgaccess(uint64, int, uint64);
#endif
```

```
#ifdef LAB_PGTBL
int             pgaccess(uint64, int, uint64);
#endif
```

(5) 在 kernel/syscall.h 中添加系统调用号

```
#define SYS_pgaccess 30  // 添加这一行
```

(6) 在 kernel/syscall.c 中注册系统调用

```
extern uint64 sys_pgaccess(void);// 在文件开头的外部函数声明部分添加
static uint64 (*syscalls[])(void) = {
// ... 其他系统调用 ...
[SYS_close]   sys_close,
[SYS_pgaccess] sys_pgaccess,  // 添加这一行
};
```

```
#ifdef LAB_PGTBL
extern uint64 sys_pgaccess(void);
#endif
```

```
[SYS_close]   sys_close,
#ifdef LAB_NET
[SYS_connect] sys_connect,
#endif
#ifdef LAB_PGTBL
[SYS_pgaccess] sys_pgaccess,
#endif
```

(7) 在 user/user.h 中添加用户空间声明

```
// system calls
int pgaccess(void *base, int len, void *mask);  // 添加这一行
```

```
#ifdef LAB_PGTBL
int pgaccess(void *base, int len, void *mask);
```

(8) 在 user/usys.pl 中添加系统调用入口

```
Entry("pgaccess");
```

```
entry("pgaccess");
```

(9) 编译测试

```
Make grade
```

```
#define SYS_pgaccess  30
```

```
== Test pgtbltest ==
$ make qemu-gdb
(4.1s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(206.2s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

3）实验中遇到的问题和解决方法

问题：编译错误

```
kernel/vm.c:480:24: error: invalid use of undefined type ‘struct proc’
  480 |     pte_t *pte = walk(p->pagetable, va, 0);
      |                        ^~
kernel/vm.c:493:15: error: invalid use of undefined type ‘struct proc’
  493 |   if(copyout(p->pagetable, mask, (char*)&result, sizeof(result)) < 0)
      |              ^~
make[1]: *** [Makefile:130: kernel/vm.o] Error 1
make[1]: Leaving directory '/home/fengjuncai/xv6'
make: *** [Makefile:336: grade] Error 1
```

```
kernel/proc.h:88:19: error: field ‘lock’ has incomplete type
   88 |   struct spinlock lock;
      |                   ^~~~
make[1]: *** [Makefile:130: kernel/vm.o] Error 1
make[1]: Leaving directory '/home/fengjuncai/xv6'
```

原因：缺少结构体的声明，同时，proc.h 中的结构体 spinlock 也需要 spinlock.h 中的声明，所以需要按照顺序添加两个头文件。

```
#include "spinlock.h"
#include "proc.h"
```

4）实验心得

通过本次实验，我深入理解了操作系统虚拟内存管理机制，特别是页表访问位（Access bit）的工作原理。实验让我学习了在 xv6 系统中完整实现新系统调用的全过程。

在实现 pgaccess()系统调用过程中，我学会了 RISC-V 架构下页表项的结构，理解了硬件 MMU 如何自动设置 PTE_A 访问位。通过使用 walk()函数遍历页表来定位页表项，加深了对三级页表结构的认识。

实验中最具挑战性的是确保内核态和用户态间数据传递的安全性。通过 copyout()函数将位掩码结果传递到用户空间，学会了跨地址空间数据传输的正确方法。处理页表项时需要检查页面有效性，这让我认识到系统编程中错误处理的重要性。

编译过程中的头文件依赖问题也让我更重视代码结构组织和模块间依赖关系的管理。

Lab: traps

1. RISC-V assembly (easy)

1）实验目的

了解一些 RISC-V 汇编。

2）实验步骤

```
make fs.img
```

查看 call.asm 中生成程序的可读汇编版本。

(1) Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?

哪些寄存器包含函数的参数？比如，在 main 调用 printf 时，哪个寄存器保存了 13？

a0	第 1 个参数
a1	第 2 个参数
a2	第 3 个参数
a3	第 4 个参数
a4	第 5 个参数
a5	第 6 个参数
a6	第 7 个参数
a7	第 8 个参数

13 保存在 a2 中。

```
void main(void) {
  1c: 1141                addi  sp,sp,-16
  1e: e406                sd ra,8(sp)
  20: e022                sd s0,0(sp)
  22: 0800                addi  s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                li a2,13
```

(2) Where is the call to function f in the assembly code for main? Where is the call to g? (Hint: the compiler may inline functions.)

在 main 的汇编代码中，调用函数 f 的位置在哪里？调用 g 的位置在哪里？（提示：编译器可能会内联函数。）

Main 中没有调用 f 和 g 的位置。

因为编译器进行了完全内联优化：编译器在编译时就计算出了 f(8)+1 的结果，计算时候直接加载了常数。

(3) At what address is the function printf located?

函数 printf 位于哪个地址？

```
void main(void) {
  1c: 1141                addi  sp,sp,-16
  1e: e406                sd ra,8(sp)
  20: e022                sd s0,0(sp)
  22: 0800                addi  s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                li a2,13
  26: 45b1                li a1,12
  28: 00000517            auipc a0,0x0
  2c: 7a850513            addi  a0,a0,1960 # 7d0 <malloc+0x102>
  30: 00000097            auipc ra,0x0
  34: 5e6080e7            jalr  1510(ra) # 616 <printf>
  exit(0);
  38: 4501                li a0,0
  3a: 00000097            auipc ra,0x0
  3e: 274080e7            jalr  628(ra) # 2ae <exit>
```

auipc ra,0x0 (地址 0x30)：将当前 PC 值加载到 ra 寄存器：PC = 0x30，所以 ra = 0x30

jalr 1510(ra) (地址 0x34)：1510 是十进制，转换为十六进制：$1510_{10} = 0x5e6_{16}$

跳转地址 = ra + 0x5e6 = 0x30 + 0x5e6 = 0x616

所以 printf 位于地址 0x616

如图：

```
void
printf(const char *fmt, ...)
{
  616: 711d                addi  sp,sp,-96
  618: ec06                sd ra,24(sp)
  61a: e822                sd s0,16(sp)
  61c: 1000                addi  s0,sp,32
  61e: e40c                sd a1,8(s0)
  620: e810                sd a2,16(s0)
  622: ec14                sd a3,24(s0)
  624: f018                sd a4,32(s0)
  626: f41c                sd a5,40(s0)
  628: 03043823            sd a6,48(s0)
  62c: 03143c23            sd a7,56(s0)
   va_list ap;
```

(4) What value is in the register ra just after the jalr to printf in main?

在 main 中执行 jalr 到 printf 之后，寄存器 ra 中的值是多少？

jalr 指令执行时会：

保存返回地址：将下一条指令的地址保存到 ra 寄存器

跳转：跳转到目标地址（printf）

jalr 指令位于地址：0x34

jalr 指令执行之后的下一条指令地址：0x34 + 4 = 0x38

这个地址 0x38 就是 li a0,0 指令的位置

ra 寄存器中的值是 0x38。

(5) 执行代码

代码：

```
void main(void) {
  printf("%d %d\n", f(8)+1, 13);
  unsigned int i = 0x00646c72;
  printf("H%x Wo%s", 57616, &i);

  exit(0);
}
```

输出：

```
$ call
12 13
HE110 World$
```

H%x 部分：

57616 转换为十六进制：$57616_{10} = E110_{16}$ 输出：HE110

Wo%s 部分：

&i 指向内存中的字节序列

在 Little-Endian 系统中，0x00646c72 在内存中存储为：

低地址 → 高地址 72 6c 64 00

作为字符串读取：

0x72 = 'r' 0x6c = 'l' 0x64 = 'd' 0x00 = '\0' (字符串结束符)

输出：World (实际上是 "rld")

如果系统是 Big-Endian，要得到相同的输出：

1. 修改变量 i：

unsigned int i = 0x726c6400; // 需要改变

原因：Big-Endian 按高位字节在前存储

0x726c6400 在 Big-Endian 内存中存储为：低地址 → 高地址 72 6c 64 00

这样 %s 仍然能读到 "rld\0"

2. 数值 57616 不需要改变：

57616 的十六进制表示 E110 不受字节序影响

%x 格式化输出的是数值本身，不是内存字节序

(6) n the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

在下面的代码中，'y=' 之后将打印什么？ （注意：答案不是具体值。 为什么会这样？

```
$ call
x=3 y=7285$
```

答案：y= 后面打印的是未初始化变量的垃圾值。

原因：变量 y 没有被正确初始化，所以它包含了栈内存中的随机数据。这是一个典型的编程错误，应该在使用变量前先初始化它。

3）实验中遇到的问题和解决方法

问题 1：理解 RISC-V 汇编指令格式

问题描述：初次接触 RISC-V 汇编时，对指令格式和寄存器命名不熟悉，特别是 auipc 和 jalr 指令的组合使用。

解决方法：

查阅 RISC-V 指令集手册，了解各指令的具体功能，通过实际计算地址偏移来理解指令执行过程。

问题 2：编译器优化导致的困惑

问题描述：在汇编代码中找不到函数 f 和 g 的调用，与预期的函数调用不符。

解决方法：

理解编译器优化机制，特别是内联优化，通过分析常数加载指令，推断编译器已经在编译时计算出结果。

4）实验心得

通过本次实验，我深入了解了 RISC-V 汇编语言的特点。RISC-V 指令集设计简洁，寄存器使用规律，如 a0-a7 用于参数传递，相比 x86 汇编更容易理解和分析。

实验中最令我印象深刻的是编译器的优化能力。编译器能够在编译时进行常量折叠和函数内联，这种优化提高了程序执行效率，但也增加了调试难度。理解编译器优化对于系统编程很重要。

通过分析字节序和内存布局，我认识到底层细节的重要性。不同架构的字节序差异会影响程序行为，未初始化变量可能导致不可预测的程序行为。理解这些底层细节有助于定位和解决复杂问题。

实验中关于未初始化变量的问题让我反思编程习惯，要养成声明变量时立即初始化的好习惯，在代码审查时特别注意变量的初始化问题，编写代码时考虑各种边界情况。

2. Backtrace (moderate)

1）实验目的

backtrace 主要任务是：实现一个能打印出当前调用栈信息的函数，帮助我们理解函数调用的嵌套关系。这在调试和理解内核执行流程时非常有用。

本实验的目标是：理解 x86 调用约定（函数调用如何保存/恢复返回地址、参数、ebp 等）。实现 mon_backtrace，递归/循环打印出当前调用栈，以及学会如何通过栈帧（frame pointer，ebp）追踪函数调用链。

2）实验步骤

(1) 在 kernel/printf.c 中实现 backtrace() 函数

1.获取当前帧指针：通过 r_fp() 得到当前函数的帧指针（s0）。

2.遍历栈帧：每个栈帧结构如下（RISC-V calling convention）：

[fp - 8] 是保存的返回地址（ra） [fp - 16] 是上一个帧指针（fp）

3.打印返回地址：循环输出每一层的返回地址，直到到达栈底（即 fp 超出当前栈页）。

4.终止条件：fp 为 0 或超出当前栈页范围。

```
void
backtrace(void)
{
  uint64 fp = r_fp(); // 当前帧指针
  uint64 stack_top = PGROUNDUP(fp);      // 当前栈页顶部
  uint64 stack_bottom = PGROUNDDOWN(fp); // 当前栈页底部

  printf("backtrace:\n");
  // 每个栈帧结构:
  // [fp - 8]  -> 返回地址  [fp - 16] -> 上一个帧指针

  while (fp != 0 && fp >= stack_bottom && fp + 16 <= stack_top)
    uint64 ra = *((uint64 *)(fp - 8));    // 取出返回地址
    printf("%p\n", (void*)ra);

    fp = *((uint64 *)(fp - 16));          // 取出上一个帧指针
  }
}
```

(2) 在 kernel/defs.h 中声明 backtrace

```
void backtrace(void);
```

```
void            backtrace(void);
```

(3) 在 kernel/riscv.h 中添加读取帧指针的函数

```
static inline uint64
r_fp()
{
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x) );
  return x;
}
```

```c
static inline uint64
r_fp()
{
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x) );
  return x;
}
```

(4) 在 kernel/sysproc.c 的 sys_sleep 函数中调用 backtrace()

```
backtrace();
```

```c
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  if(n < 0)
  backtrace();
```

(5) 编译并运行

```
make clean
make qemu
运行 bttest
```

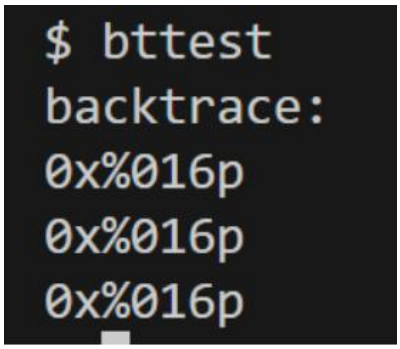
```
$ bttest
backtrace:
0x0000000080002126
0x0000000080001fee
0x0000000080001cd8
```

(6) 使用 addre2line 工具解析地址

```
addr2line -e kernel/kernel //先退出 qemu
//然后黏贴运行 bttest 得到的地址，最后 Ctrl-D 结束
```

```
0x0000000080002126
/home/fengjuncai/xv6/kernel/sysproc.c:66
0x0000000080001fee
/home/fengjuncai/xv6/kernel/syscall.c:140 (discriminator 1)
0x0000000080001cd8
/home/fengjuncai/xv6/kernel/trap.c:76
```

3）实验中遇到的问题和解决方法



问题：backtrace()函数打印格式不对。

解决方法：xv6 的 printf 实现不支持 %016p，只支持简单的 %p。

将输出函数格式改为：printf("%p\n", (void*)ra);

4）实验心得

通过本次实验，我深入理解了 RISC-V 架构下函数调用的栈帧结构和调用约定。实验让我掌握了如何通过帧指针追踪函数调用链，这对理解程序执行流程和调试非常重要。

在实现 backtrace()函数过程中，我学会了 RISC-V 的栈帧布局：返回地址保存在[fp-8]，上一个帧指针保存在[fp-16]。通过 r_fp()获取当前帧指针并循环遍历，能够完整追踪函数调用的嵌套关系。

实验中遇到的 printf 格式问题让我认识到不同系统对格式化输出支持的差异。通过将%016p 改为简单的%p 格式，解决了输出格式不匹配的问题。

使用 addr2line 工具将地址转换为具体的函数名和行号，让我学会了如何将底层的内存地址映射到源代码，这是系统调试的重要技能。

3. Alarm (hard)

1）实验目的

本实验的目标是让用户程序可以通过 sigalarm 系统调用，在每隔 N 个时钟中断后自动执行一个用户自定义的函数（handler），并且在 handler 执行完后可以回到原来的执行点继续运行（通过 sigreturn）。

2）实验步骤

(1) 在进程结构体中添加字段

```
在 proc.h 的 struct proc 结构体中添加如下字段：
int alarmticks; // 用户设置的周期
int tickcount; // 计数器
void *handler; // handler 函数指针
int handling_alarm; // 标记是否正在 handler 里
struct trapframe alarm_tf; // 保存 handler 前的 trapframe
```

```
struct proc {

  int alarmticks; // 用户设置的周期
  int tickcount; // 计数器
  void *handler; // handler 函数指针
  int handling_alarm; // 标记是否正在 handler 里
  struct trapframe alarm_tf; // 保存 handler 前的 trapframe
```

(2) 实现 sigalarm 系统调用

位置：sysproc.c

```
 uint64
sys_sigalarm(void)
{
  int ticks;
  uint64 handler;
  if (argint(0, &ticks) < 0 || argaddr(1, &handler) < 0)
    return -1;
  struct proc *p = myproc();//获取进程
  p->alarmticks = ticks;//时钟间隔
  p->handler = (void*)handler;//处理函数
  p->tickcount = 0;//重置计数器
  return 0;
}
并注册 sys_sigalarm
```

```c
uint64
sys_sigalarm(void)
{
  int ticks;
  uint64 handler;
  if (argint(0, &ticks) < 0 || argaddr(1, &handler) < 0)
    return -1;
  struct proc *p = myproc();
  p->alarmticks = ticks;
  p->handler = (void*)handler;
  p->tickcount = 0;
  return 0;
}
```

```c
#define SYS_sigalarm 22
```

```c
extern uint64 sys_sigalarm(void);
```

```c
[SYS_sigalarm] sys_sigalarm,
```

(3) 实现 sigreturn 系统调用

在 sysproc.c 添加 sys_sigreturn:

```c
uint64
sys_sigreturn(void)//从 alarm 信息处理函数中安全返回被中断的源程序位置
{
  struct proc *p = myproc();//获取当前进程
  //回复被 alarm 中断时的寄存器状态
  memmove(p->trapframe, &p->alarm_tf, sizeof(struct trapframe));
  p->handling_alarm = 0;//清除标志 准备下一次触发
  return 0;
}
```

在 syscall.c 注册 sys_sigreturn

```c
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  memmove(p->trapframe, &p->alarm_tf, sizeof(struct trapframe));
  p->handling_alarm = 0;
  return 0;
}
```

```c
#define SYS_sigreturn 23
```

```c
extern uint64 sys_sigreturn(void);
```

```c
[SYS_sigreturn] sys_sigreturn,
```

(4) 在 usertrap 处理时钟中断

位置：trap.c 的 usertrap()

```
 if (which_dev == 2) { // 时钟中断检测
  if (p->alarmticks > 0 && !p->handling_alarm) {
//进程设置了 alarm 且当前没有再处理 alarm
    p->tickcount++;//计数器++
    if (p->tickcount >= p->alarmticks) {//判断是否到达设定的时间
      p->tickcount = 0;//重置计数器
      p->handling_alarm = 1;//设置处理标志
      memmove(&p->alarm_tf, p->trapframe, sizeof(struct trapframe));
      p->trapframe->epc = (uint64)p->handler;//跳转到 handler
    }
  }
}
```

```
if (which_dev == 2) { // timer interrupt
  if (p->alarmticks > 0 && !p->handling_alarm) {
    p->tickcount++;
    if (p->tickcount >= p->alarmticks) {
      p->tickcount = 0;
      p->handling_alarm = 1;
      memmove(&p->alarm_tf, p->trapframe, sizeof(struct trapframe))
      p->trapframe->epc = (uint64)p->handler;
    }
  }
}
```

(5) 用户态声明与实现

User.h

```
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

```
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

Usys.pl

```
entry("sigalarm");
entry("sigreturn");
```

```
entry("sigalarm");
entry("sigreturn");
```

(6) 测试

```
Make grade
```

```
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (4.9s)
== Test running alarmtest ==
$ make qemu-gdb
(3.7s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb


usertests: OK (186.8s)
== Test time ==
time: OK
Score: 85/85
```

3）实验中遇到的问题和解决方法

```
== Test   alarmtest: test0 ==
  alarmtest: test0: FAIL
    ...
         hart 1 starting
         init: starting sh
         $ alarmtest
         exec alarmtest failed
         $ qemu-system-riscv64: terminating on signal 15 from pid 20111 (make)
    MISSING '^test0 passed$'
== Test   alarmtest: test1 ==
  alarmtest: test1: FAIL
    ...
         hart 1 starting
         init: starting sh
         $ alarmtest
         exec alarmtest failed
         $ qemu-system-riscv64: terminating on signal 15 from pid 20111 (make)
    MISSING '^\.?test1 passed$'
== Test   alarmtest: test2 ==
  alarmtest: test2: FAIL
    ...
         hart 1 starting
         init: starting sh
         $ alarmtest
         exec alarmtest failed
         $ qemu-system-riscv64: terminating on signal 15 from pid 20111 (make)
    MISSING '^\.?test2 passed$'
== Test usertests ==
$ make qemu-gdb
```

问题：用户程序启动失败。

解决方法：检查过内核程序没有问题后，检查一下测试程序是否加入到了 makefile。如果没有，则补充。

4）实验心得

通过本次实验，我深入理解了操作系统信号机制的实现原理。实验让我掌握了如何在内核中实现定时器中断处理和用户自定义信号处理函数。

在实现过程中，最关键的是学会了上下文保存与恢复。通过 alarm_tf 字段保存 trapframe，确保信号处理函数执行后能正确返回原执行点。在 usertrap()中处理时钟中断时，需要精确控制计数器并避免重入问题。

实验中遇到的用户程序启动失败问题提醒我构建系统配置的重要性，检查 Makefile 确保测试程序正确编译是基本技能。

总体而言，实验加深了我对中断处理和信号系统的理解，提升了系统编程能力。

Lab: Copy-on-Write Fork for xv6

1. Implement copy-on write(hard)

1）实验目标

本实验旨在为 xv6 操作系统实现写时复制（Copy-on-Write, COW）机制。通过修改 fork() 系统调用，使父子进程初始时共享物理内存页面，只有在写操作时才进行页面复制，从而节省内存并提升性能。实验要求深入理解虚拟内存管理、页表权限和异常处理，并通过测试验证实现的正确性。

2）实验步骤

定义一个新的页表项标志 PTE_COW，表示“此页为写时复制”。

fork 时，对所有可写的用户页：

去掉写权限，加上 COW 标志。

父子进程页表都映射同一物理页。

增加物理页的引用计数。

当父或子进程写入 COW 页时，触发 page fault：

分配一个新物理页，把原页内容复制过去。

新页只属于当前进程，并恢复写权限。

旧页引用计数减一，必要时释放。

(1) 添加物理页面引用计数机制

将物理内存页的引用计数实现安放在 kernel/kalloc.c

```
static int reference_count[(PHYSTOP - 0x80000000) / PGSIZE];
 static int idx_rc(uint64 pa){ //物理地址转换为数组索引
    return (pa - 0x80000000) / PGSIZE;
 }
 void add_rc(uint64 pa){//增加指定物理页面的引用计数
    reference_count[idx_rc(pa)]++;
 }
 void sub_rc(uint64 pa){//减少引用计数
    reference_count[idx_rc(pa)]--;
 }
 void
 kinit()//初始化
 {
   initlock(&kmem.lock, "kmem");//初始化自旋锁
  memset(reference_count,0,sizeof(reference_count));//初始化计数
   freerange(end, (void*)PHYSTOP);//将物理内存加入空闲页面链表
 }
```

```
static int reference_count[(PHYSTOP - 0x80000000) / PGSIZE];
static int idx_rc(uint64 pa){
  return (pa - 0x80000000) / PGSIZE;
}
void add_rc(uint64 pa){
  reference_count[idx_rc(pa)]++;
}
void sub_rc(uint64 pa){
  reference_count[idx_rc(pa)]--;
}
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  memset(reference_count,0,sizeof(reference_count));
  freerange(end, (void*)PHYSTOP);
}
```

(2) 修改 kfree 函数

在 kfree 函数中对所要释放的物理内存页的引用计数进行判断，如其引用计数大于 1，则仅将其减一；如引用次数小于 1（当为 1 时仅有一个虚拟内存页映射至该页，当为 0 时为 kinit 初始化调用 kfree），则释放该页并将其添加至空闲列表：

```
void
kfree(void *pa)
{
  struct run *r;
  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)//参数有效性检查
    panic("kfree");
  if(reference_count[idx_rc((uint64)pa)] > 1){//检查引用计数
      sub_rc((uint64)pa);//减少引用计数
      return;//提前返回不实际释放
  }

  memset(pa, 1, PGSIZE);   //用垃圾数据填充页面 帮助检测悬挂指针错误
  r = (struct run*)pa;//页面转换为空闲链表节点

  acquire(&kmem.lock);
  r->next = kmem.freelist;//当前页面连接到空闲链表头部
  kmem.freelist = r;//更新空闲链表头指针
  release(&kmem.lock);

  reference_count[idx_rc((uint64)pa)] = 0;//引用计数重置为 0
}
```

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)p
    panic("kfree");
  if(reference_count[idx_rc((uint64)pa)] > 1){
      sub_rc((uint64)pa);
      return;
  }
  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
  reference_count[idx_rc((uint64)pa)] = 0;
}
```

(3) 修改 kalloc

在 kalloc 中，如空闲列表不为空，则将被分配的物理内存页的引用计数初始化为 1：

```
void *
kalloc(void)//分配页面
{
  struct run *r;
  acquire(&kmem.lock);
  r = kmem.freelist;//获取空闲链表头部
  if(r)
    kmem.freelist = r->next;//更新链表到下一个页面
  release(&kmem.lock);
  if(r)//如果成功分配页面
    memset((char*)r, 5, PGSIZE); //垃圾数据填充
  if(r)
    reference_count[idx_rc((uint64)r)] = 1;//引用计数设置为 1
   return (void*)r;
  }
```

```
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk

  if(r)
    reference_count[idx_rc((uint64)r)] = 1;
  return (void*)r;
}
```

(4) 修改 vm.c

更改 uvmcopy 以使得在 fork 时父子进程共享同一片物理内存。在这里，定义利用了页表项的用户保留 flag 项，规定 #define PTE_C (1L << 5) ，以标记虚拟内存页是否为 COW 页（即被写时复刻的页）：

```
#define PTE_C (1L << 5)
```

```
Int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz){//复制页表
    pte_t *pte;
    uint64 pa, i;
    uint flags;
  //遍历所有需要复制的虚拟页面
    for(i = 0; i < sz; i += PGSIZE){
      if((pte = walk(old, i, 0)) == 0)//在父进程中查找
        panic("uvmcopy: pte should exist");
      if((*pte & PTE_V) == 0)//检查是否存在
        panic("uvmcopy: page not present");
      pa = PTE2PA(*pte);//提取物理页面
      *pte = (*pte & ~PTE_W) | PTE_C;//权限
      flags = PTE_FLAGS(*pte);//获取修改后的标志位
      if(mappages(new, i, PGSIZE, pa, flags) != 0)//子进程中建立映射，指向同一物理页面      goto err;
      add_rc(pa);//增加物理页面的引用计数
    }
    return 0;
   err://错误处理
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
  }
```

```
for(i = 0; i < sz; i += PGSIZE){
  if((pte = walk(old, i, 0)) == 0)
    panic("uvmcopy: pte should exist");
  if((*pte & PTE_V) == 0)
    panic("uvmcopy: page not present");
  pa = PTE2PA(*pte);
  *pte = (*pte & ~PTE_V) | PTE_C;
  flags = PTE_FLAGS(*pte);

  if(mappages(new, i, PGSIZE, pa, flags) != 0)
    goto err;
  add_rc(pa);
}
return 0;
```

(5) 修改 trap.c

在 usertrap 中捕捉 COW 页的被写入情况。

```
    } else if((which_dev = devintr()) != 0){ // ok //设备中断处理
   } else {//处理页面错误异常 cow
      uint64 va = PGROUNDDOWN(r_stval());//获取触发异常的虚拟地址
      if(va >= MAXVA){//检查是否在有效范围内
         p->killed = 1;
         exit(-1);
      }
      pte_t* pte = walk(p->pagetable,va,0);//查找对应的页表项
      if(pte == 0){
         p->killed = 1;
         exit(-1);
      }
      uint64 pa = PTE2PA(*pte);//提取物理地址和标志位
      uint64 flag = PTE_FLAGS(*pte);
      if((r_scause() == 13 || r_scause() == 15) && (flag & PTE_C) ){
         char* mem = kalloc();//分配新的物理地址
         if(mem == 0){
            printf("here\n");
            p->killed = 1;
            exit(-1);
         }
         memmove(mem,(char*)pa,PGSIZE);//复制原页面内容到新页面
         kfree((char*)pa);//释放对原页面的引用
         *pte = PA2PTE(mem) | (flag & ~PTE_C) | PTE_W;//恢复权限
      }
      else{
         printf("usertrap(): unexpected scause %p pid=%d\n",
r_scause(), p->pid);
         printf("          sepc=%p stval=%p\n", r_sepc(),
r_stval());
         p->killed = 1;
      }
```

```
} else {
  uint64 va = PGROUNDDOWN(r_stval());
  if(va >= MAXVA){
      p->killed = 1;
      exit(-1);
  }
  pte_t* pte = walk(p->pagetable,va,0);
  if(pte == 0){
      p->killed = 1;
      exit(-1);
  }
  uint64 pa = PTE2PA(*pte);
  uint64 flag = PTE_FLAGS(*pte);
  if((r_scause() == 13 || r_scause() == 15) && (flag & PTE_C) ){
      char* mem = kalloc();
      if(mem == 0){
          printf("here\n");
          p->killed = 1;
          exit(-1);
      }
      memmove(mem,(char*)pa,PGSIZE);
      kfree((char*)pa);
      *pte = PA2PTE(mem) | (flag & ~PTE_C) | PTE_W;
  } else {
      printf("usertrap(): unexpected scause %p pid=%d\n", r_scaus
      printf("            sepc=%p stval=%p\n", r_sepc(), r_stval(
      p->killed = 1;
  }
```

在 copyout 中增加与 usertrap 的类似的判断代码，防止对 COW 页的直接写入

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64
len)
    {
      uint64 n, va0, pa0, flag;
      pte_t* pte;
      while(len > 0){//循环处理
        va0 = PGROUNDDOWN(dstva);//目标地址对齐
        if(va0 >= MAXVA)//检查是否有效
            return -1;
        pte = walk(pagetable,va0,0);//查找页表项
        if(pte == 0)
            return -1;
        pa0 = PTE2PA(*pte);//提取物理地址和标志位
        flag = PTE_FLAGS(*pte);
        if(flag & PTE_C){//COW 页面处理
            char* mem = kalloc();//分配新的物理页面
            if(mem == 0)
                return -1;
            memmove(mem,(char*)pa0,PGSIZE);//页面复制
            kfree((char*)pa0);//释放引用
            *pte = PA2PTE((uint64)mem) | (flag & ~PTE_C) | PTE_W;
            //更新页表项 指向新的页面 清除 COW 标志 添加写权限
```

```
        pa0 = (uint64)mem;//使用新的地址
      }
      if(pa0 == 0)
        return -1;
      n = PGSIZE - (dstva - va0);//当前页面剩余字节数
      if(n > len)//不能超过总的待复制字节数
        n = len;
      memmove((void *)(pa0 + (dstva - va0)), src, n);//执行复制
      len -= n;//减少剩余字节数
      src += n;//移动指针
      dstva = va0 + PGSIZE;//移动到下一页的起始地址
    }
    return 0;
  }
```

```
  pte_t *pte;
  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    if(va0 >= MAXVA)
      return -1;
    pte = walk(pagetable, va0, 0);
    if(pte == 0)
      return -1;
    pa0 = PTE2PA(*pte);
    flag = PTE_FLAGS(*pte);
    if(flag & PTE_C){
      char *mem = kalloc();
      if(mem == 0)
        return -1;
      memmove(mem, (char*)pa0, PGSIZE);
      kfree((char*)pa0);
      *pte = PA2PTE(mem) | (flag & ~PTE_C) | PTE_W;
      pa0 = (uint64)mem;
    }
```

(6) 清理和测试

```
Make grade
```

```
== Test running cowtest ==
$ make qemu-gdb
(11.5s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(176.8s)
    (Old xv6.out.usertests failure log removed)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
```

3）实验中遇到的问题和解决方法

```
Timeout! (300.3s)
    ...
        OK
        test opentest: OK
        test writetest: OK
        test writebig: panic: balloc: out of blocks
        qemu-system-riscv64: terminating on signal 15 from pid 22657 (make)
    MISSING '^ALL TESTS PASSED$'
    QEMU output saved to xv6.out.usertests
== Test   usertests: copyin ==
  usertests: copyin: FAIL
    Parent failed: test_usertests
== Test   usertests: copyout ==
  usertests: copyout: FAIL
    Parent failed: test_usertests
== Test   usertests: all tests ==
  usertests: all tests: FAIL
    Parent failed: test_usertests
== Test time ==
time: OK
Score: 81/110
make: *** [Makefile:340: grade] Error 1
```

报错：块不足

```
#define FSSIZE       10000  // size of file system in blocks
```

解决方式：增加块的数量。

4）实验心得

通过本次实验，我深入理解了写时复制（COW）机制的实现原理和虚拟内存管理的核心概念。实验让我掌握了如何通过延迟复制来优化内存使用和提升系统性能。

在实现过程中，最关键的是理解引用计数机制。通过为每个物理页面维护引用计数，确保只有在引用计数为 1 时才真正释放页面，这避免了过早释放共享页面的问题。同时，在 kfree 和 kalloc 中正确管理引用计数是整个 COW 机制的基础。

修改 uvmcopy 函数让我理解了 fork 时的内存共享策略：移除写权限并添加 COW 标志，使父子进程共享同一物理页面。在 usertrap 和 copyout 中处理页面错误时，需要分配新页面、复制内容并恢复写权限，这个过程体现了 COW 的核心思想。

实验中遇到的"块不足"问题提醒我系统资源管理的重要性，通过增加可用块数量解决了内存分配失败的问题。

总体而言，这个实验加深了我对虚拟内存管理和优化策略的理解，提升了系统编程和调试能力。

Lab: Multithreading

1. Uthread: switching between threads (moderate)

1）实验目标

实现一个用户级线程系统的上下文切换机制。

2）实验步骤

(1) 添加上下文结构

在 user/uthread.c 文件开头添加上下文结构定义：用于保存当前进程状态，把当前线程的寄存器值保存起来，以便于再回复目标线程的状态时候，可以把目标线程直线保存的寄存器值加载回 CPU。

```
 // 在文件开头添加这个结构
struct context {
  uint64 ra;    // 返回地址
  uint64 sp;    // 栈指针

  // callee-saved registers (被调用者保存的寄存器)
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

```
struct context {
  uint64 ra;    // 返回地
  uint64 sp;    // 栈指针

  // callee-saved registe
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

然后修改 struct thread：

```
 struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct context context;       /* 保存寄存器上下文 */
};
```

```
struct thread {
  char       stack[STACK_SIZE];
  int        state;
  struct context context;
};
```

(2) 添加函数声明

在全局变量声明附近添加：

```
extern void thread_switch(struct context *old, struct context
*new);
```

(3) 实现 thread_create()

找到 thread_create() 函数，完成实现：

```
 void thread_create(void (*func)())//创建线程
{
  struct thread *t;
```

```
  // 找到一个空闲的线程槽
  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;//线程状态设置为运行
  t->context.sp = (uint64)t->stack + STACK_SIZE;/设置栈指针到栈顶
  // 返回地址设为线程函数，这样第一次切换到这个线程时会执行该函数
  t->context.ra = (uint64)func;
}
```

```
void
thread_create(void (*func)())
{
  struct thread *t;

  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->context.sp = (uint64)t->stack + STACK_SIZE;

  // 返回地址设为线程函数，这样第一次切换到这个线程时会执行该函数
  t->context.ra = (uint64)func;
}
```

(4) 修改 thread_schedule()

找到 thread_schedule() 函数中的注释部分，添加切换调用：

```
 void
thread_schedule(void)
{  //前面的代码
  if (current_thread != next_thread) {    //判断是否需要切换线程
    next_thread->state = RUNNING;//状态设置为 running
    t = current_thread;//保留原本的线程
    current_thread = next_thread;//更新当前的线程指针
    // 关键：调用汇编函数进行上下文切换
    thread_switch(&t->context, &next_thread->context);
//上下文切换 保存当前状态 当前线程被暂停,直到将来被重新调度时才会从这里继续
  } else
    next_thread = 0;
}
```

```
if (current_thread != next_thread) {         /* switch threads?
  next_thread->state = RUNNING;
  t = current_thread;
  current_thread = next_thread;
  t->state = RUNNABLE;                       /* set old thread t
  thread_switch(&t->context, &next_thread->context);
} else
  next_thread = 0;
```

(5) 实现汇编函数 thread_switch

编辑 user/uthread_switch.S 文件：

```
    .text

    /*
        * save the old thread's registers,
        * restore the new thread's registers.
        */

    .globl thread_switch
thread_switch:
    # a0 = &old->context
    # a1 = &new->context

    # 保存旧线程的寄存器到 old->context
    sd ra, 0(a0)      # 保存返回地址
    sd sp, 8(a0)      # 保存栈指针
    sd s0, 16(a0)     # 保存 s0
    sd s1, 24(a0)     # 保存 s1
    sd s2, 32(a0)     # 保存 s2
    sd s3, 40(a0)     # 保存 s3
    sd s4, 48(a0)     # 保存 s4
    sd s5, 56(a0)     # 保存 s5
    sd s6, 64(a0)     # 保存 s6
    sd s7, 72(a0)     # 保存 s7
    sd s8, 80(a0)     # 保存 s8
    sd s9, 88(a0)     # 保存 s9
    sd s10, 96(a0)    # 保存 s10
    sd s11, 104(a0)   # 保存 s11

    # 从 new->context 恢复新线程的寄存器
    ld ra, 0(a1)      # 恢复返回地址
```

```
    ld sp, 8(a1)      # 恢复栈指针
    ld s0, 16(a1)     # 恢复 s0
    ld s1, 24(a1)     # 恢复 s1
    ld s2, 32(a1)     # 恢复 s2
    ld s3, 40(a1)     # 恢复 s3
    ld s4, 48(a1)     # 恢复 s4
    ld s5, 56(a1)     # 恢复 s5
    ld s6, 64(a1)     # 恢复 s6
    ld s7, 72(a1)     # 恢复 s7
    ld s8, 80(a1)     # 恢复 s8
    ld s9, 88(a1)     # 恢复 s9
    ld s10, 96(a1)    # 恢复 s10
    ld s11, 104(a1)   # 恢复 s11

    ret               # 跳转到 ra 寄存器指向的地址
```

```
    .globl thread_switch
thread_switch:
    # a0 = &old->context
    # a1 = &new->context

    # 保存旧线程的寄存器到 old->context
    sd ra, 0(a0)      # 保存返回地址
    sd sp, 8(a0)      # 保存栈指针
    sd s0, 16(a0)     # 保存 s0
    sd s1, 24(a0)     # 保存 s1
    sd s2, 32(a0)     # 保存 s2
    sd s3, 40(a0)     # 保存 s3
    sd s4, 48(a0)     # 保存 s4
    sd s5, 56(a0)     # 保存 s5
    sd s6, 64(a0)     # 保存 s6
    sd s7, 72(a0)     # 保存 s7
    sd s8, 80(a0)     # 保存 s8
    sd s9, 88(a0)     # 保存 s9
    sd s10, 96(a0)    # 保存 s10
    sd s11, 104(a0)   # 保存 s11

    # 从 new->context 恢复新线程的寄存器
    ld ra, 0(a1)      # 恢复返回地址
    ld sp, 8(a1)      # 恢复栈指针
    ld s0, 16(a1)     # 恢复 s0
```

输出　调试控制台　终端　端口

(6)　编译测试

输出与教程吻合。

```
问题    输出    调试控制台    终端    端口

thread_a 94
thread_b 94
thread_c 95
thread_a 95
thread_b 95
thread_c 96
thread_a 96
thread_b 96
thread_c 97
thread_a 97
thread_b 97
thread_c 98
thread_a 98
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
```

3）实验中遇到的问题和解决方法

1. 为什么 thread 只需要保存/恢复被调用者保存的寄存器。

在函数调用约定中，寄存器分为两类：

调用者保存寄存器（caller-save）：调用函数前由调用者负责保存被调用者保存寄存器（callee-save）：被调用函数负责保存和恢复

thread_switch 只需要保存被调用者保存的寄存器，因为：

1 调用者保存的寄存器在调用 thread_switch 之前已经被调用者处理了

2 线程切换时，我们需要保存的是那些跨函数调用仍需保持的寄存器状态

4）实验心得

通过本次用户级线程实验，我深入理解了线程上下文切换的基本原理和实现方法。

上下文切换的本质就是保存当前线程的寄存器状态，然后恢复目标线程的寄存器状态。通过定义 struct context 结构体来保存关键寄存器，包括返回地址、栈指针和被调用者保存寄存器。在实验过程中我明白了为什么只需要保存被调用者保存寄存器，因为调用者保存寄存器在函数调用时已经被调用者处理了。

线程调度机制的实现让我印象深刻。在 thread_create()函数中，通过设置 context.ra = (uint64)func，使得新线程在首次被调度时能够跳转到指定的线程函数执行。在 thread_schedule()函数中调用 thread_switch()实现真正的上下文切换，完成线程状态的转换。

实验过程中遇到了一些问题，比如初期对寄存器偏移量计算不熟悉，通过仔细分析结构体布局得到解决。对于栈指针为什么要设置为 stack + STACK_SIZE，后来理解了栈是向下增长的特性。最困惑的是返回地址的设置作用，理解后发现这是实现线程首次执行的关键机制。

这个实验让我从底层角度理解了线程切换的工作原理，体会到了系统编程中汇编语言的重要性，也加深了对操作系统线程调度机制的认识，为后续深入学习操作系统原理打下了坚实基础。

2. Using threads (moderate)

1）实验目标

本次的实验目标是，探索使用哈希表的线程和锁的并行编程。

2）实验步骤

(1) 步骤 1：编译并测试原始程序

```
$ cd notxv6
$ make ph
$ ./ph 1    # 单线程测试
$ ./ph 2    # 双线程测试 - 会发现数据丢失
```

```
fengjuncai@Qln:~$ cd
fengjuncai@Qln:~$ cd xv6
fengjuncai@Qln:~/xv6$ ls
LICENSE    __pycache__          fs.img            kernel  time.txt
Makefile   answers-thread.txt   grade-lab-thread  mkfs    user
README     conf                 gradelib.py       notxv6  xv6.out
fengjuncai@Qln:~/xv6$ cd notxv6
fengjuncai@Qln:~/xv6/notxv6$ make pg
make: *** No rule to make target 'pg'.  Stop.
fengjuncai@Qln:~/xv6/notxv6$ make ph
cc     ph.c   -o ph
```

```
fengjuncai@Qln:~/xv6/notxv6$ ./ph 1
100000 puts, 12.283 seconds, 8141 puts/second
0: 0 keys missing
100000 gets, 14.577 seconds, 6860 gets/second
fengjuncai@Qln:~/xv6/notxv6$ ./ph 2
100000 puts, 6.601 seconds, 15148 puts/second
0: 1171 keys missing
1: 1171 keys missing
200000 gets, 9.850 seconds, 20304 gets/second
```

(2) 步骤 2：分析问题

Ph 1 测试一个线程的时候，能够正常运行。

但是 Ph2 测试两个线程的时候，无法正常进行，遗失了大量数据，存在并发问题。

(3) 步骤 3：添加锁保护

在 ph.c 中添加互斥锁：

```
pthread_mutex_t lock;           // 声明互斥锁

// 在 main 函数中初始化锁
pthread_mutex_init(&lock, NULL);

// 在 put 函数中添加锁保护
static
```

```
void put(int key, int value)
{
  int i = key % NBUCKET;

  // 加锁保护临界区
  pthread_mutex_lock(&lock);
  • • •
  // 解锁
  pthread_mutex_unlock(&lock);
}
```

```
pthread_mutex_t lock;
```

```
main(int argc, char *argv[])
{
  pthread_t *tha;
  void *value;
  double t1, t0;
  pthread_mutex_init(&lock, NULL);
```

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;
  // 加锁保护临界区
  pthread_mutex_lock(&lock);

  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    insert(key, value, &table[i], table[i]);
  }
  pthread_mutex_unlock(&lock);

}
```

结果：没有数据损失。

```
fengjuncai@Qln:~/xv6/notxv6$ ./ph 1
100000 puts, 10.773 seconds, 9283 puts/second
0: 0 keys missing
100000 gets, 14.226 seconds, 7029 gets/second
fengjuncai@Qln:~/xv6/notxv6$ ./ph 2
100000 puts, 19.126 seconds, 5229 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 11.051 seconds, 18098 gets/second
```

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (5.8s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/fengjuncai/xv6'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/fengjuncai/xv6'
ph_safe: OK (29.2s)
== Test ph_fast == make[1]: Entering directory '/home/fengjuncai/xv6'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/fengjuncai/xv6'
ph_fast: FAIL (50.4s)
    Parallel put() speedup is less than 1.25x
```

通过 make grade 测试，我们可以看到 ph_safe 测试通过，但是由于锁不足，程序运行较慢，ph_fast 没有通过。

(4) 步骤 4：优化锁的粒度

为了提高性能，可以为每个哈希桶使用单独的锁：

```
pthread_mutex_t locks[NBUCKET];  // 每个桶一个锁
```

```
pthread_mutex_t locks[NBUCKET];
```

```
// 初始化所有锁
for(int i = 0; i < NBUCKET; i++) {
    pthread_mutex_init(&locks[i], NULL);
}
// 在 put 函数中使用对应桶的锁
static
void put(int key, int value)
{
  int i = key % NBUCKET;
  pthread_mutex_lock(&locks[i]);   // 只锁定对应的桶

  • • •
  pthread_mutex_unlock(&locks[i]);
}
```

```
for(int i = 0; i < NBUCKET; i++) {
  pthread_mutex_init(&locks[i], NULL);
}
```

```
// 加锁保护临界区
pthread_mutex_lock(&locks[i]);

struct entry *e = 0;
for (e = table[i]; e != 0; e = e->next) {
  if (e->key == key)
    break;
}
if(e){
  // update the existing key.
  e->value = value;
} else {
  // the new is new.
  insert(key, value, &table[i], table[i]);
}
pthread_mutex_unlock(&locks[i]);
```

运行结果：没有数据损失。

```
fengjuncai@Qln:~/xv6/notxv6$ ./ph 1
100000 puts, 9.233 seconds, 10831 puts/second
0: 0 keys missing
100000 gets, 12.371 seconds, 8084 gets/second
fengjuncai@Qln:~/xv6/notxv6$ ./ph 2
100000 puts, 4.145 seconds, 24127 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 10.400 seconds, 19230 gets/second
```

Make grade 结果：ph_saffe 和 ph_fast 都通过测试。

```
== Test ph_safe == make[1]: Entering directory '/home/fengjuncai/xv6'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/fengjuncai/xv6'
ph_safe: OK (22.0s)
== Test ph_fast == make[1]: Entering directory '/home/fengjuncai/xv6'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/fengjuncai/xv6'
ph_fast: OK (44.4s)
== Test barrier == make[1]: Entering directory '/home/fengjuncai/xv6'
```

3）实验中遇到的问题和解决方法

问题：未能通过 pg_fast。

原因：应用全局锁的时候，所有线程都是串行执行；应用桶级锁的话，不同的桶可以并行操作，并且可以减少锁的竞争，提高并行度，加快程序执行速度。

4）实验心得

通过这个哈希表并发编程实验，我深刻体会到了并发编程的复杂性。当看到单线程程序正常运行，而双线程却出现数据丢失时，我意识到多线程编程远比想象中复杂，竞态条件是一个真实存在的威胁。

添加全局锁虽然解决了数据安全问题，但性能测试的失败让我明白仅仅保证正确性是不够的。全局锁将所有线程串行化，完全抵消了多线程的优势。这让我认识到好的并发程序需要在正确性和性能之间找到平衡。

细粒度锁的引入是关键突破。将单一全局锁替换为每个桶一个锁后，程序性能显著提升并通过了所有测试。这让我理解了锁粒度设计的重要性，哈希表桶之间的数据独立性为并行操作提供了可能。

3. Barrier(moderate)

1）实验目标

屏障（Barrier） 是一个同步原语，用于确保所有线程都到达某个点后才能继续执行。实验目标是实现一个线程屏障（Thread Barrier），确保所有线程在继续执行之前都到达同一个同步点。

2）实验步骤

(1) 初始条件

```
$ cd notxv6
$ make barrier
$ ./barrier 2
```

```
fengjuncai@Qln:~/xv6/notxv6$ ./barrier 2
barrier: barrier.c:45: thread: Assertion `i == t' failed.
barrier: barrier.c:45: thread: Assertion `i == t' failed.
Aborted (core dumped)
```

(2) 实现函数 barrier()

```
static void barrier(){ //实现线程屏障机制
 pthread_mutex_lock(&bstate.barrier_mutex);///加锁
 bstate.nthread++; //记录到达 barrier 的线程数
 if (bstate.nthread == nthread) {//最后一个线程到达
   bstate.nthread = 0;
   bstate.round++;
   pthread_cond_broadcast(&bstate.barrier_cond);
 } else {//不是最后一个线程
   pthread_cond_wait(&bstate.barrier_cond,
&bstate.barrier_mutex);
 }
 pthread_mutex_unlock(&bstate.barrier_mutex);//释放锁
}
```

```
static void
barrier()
{
  pthread_mutex_lock(&bstate.barrier_mutex);

  bstate.nthread++;
  if (bstate.nthread == nthread) {
    bstate.nthread = 0;
    bstate.round++;
    pthread_cond_broadcast(&bstate.barrier_cond);
  } else {
    pthread_cond_wait(&bstate.barrier_cond, &bstate.b
  }

  pthread_mutex_unlock(&bstate.barrier_mutex);

}
```

(3) 测试

```
fengjuncai@Qln:~/xv6/notxv6$ ./barrier 1
OK; passed
fengjuncai@Qln:~/xv6/notxv6$ ./barrier 2
OK; passed
fengjuncai@Qln:~/xv6/notxv6$ ./barrier 4
OK; passed
fengjuncai@Qln:~/xv6/notxv6$ ./barrier 8
OK; passed
fengjuncai@Qln:~/xv6/notxv6$ ./barrier 16
OK; passed
```

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (6.5s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/fengjuncai/xv6'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/fengjuncai/xv6'
ph_safe: OK (18.3s)
== Test ph_fast == make[1]: Entering directory '/home/fengjuncai/xv6'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/fengjuncai/xv6'
ph_fast: OK (42.2s)
== Test barrier == make[1]: Entering directory '/home/fengjuncai/xv6'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthrea
d
make[1]: Leaving directory '/home/fengjuncai/xv6'
barrier: OK (4.0s)
== Test time ==
time: OK
Score: 60/60
```

3）实验中遇到的问题和解决方法

问题：完成代码后发现程序运行时侯卡住。

```
cc     barrier.c    -o barrier
fengjuncai@Qln:~/xv6/notxv6$ ./barrier 2
Thread 133563234121408 reached barrier, round 0, bstate 1
Total threads: 0
Thread 133563234121408 waiting at barrier, round 0, bstate 1
Thread 133563225728704 reached barrier, round 0, bstate 2
Total threads: 0
Thread 133563225728704 waiting at barrier, round 0, bstate 2
L: Ubuntu  thread*  0↓ 2↑  ⊗ 0 ⚠ 0  0  行 33，列 49  空格: 2  UTF-8  LF  {}
```

输出中间变量发现，问题在于对变量了解有误，导致 bstate.nthread，以及 bstate.round 变量没有更新，且条件判断失败。另外通过观察代码，全局变量完全没有用处，可以删除。

解决方法：

删除全局变量 round，并正确更新变量和判断条件。

4）实验心得

通过这个线程屏障实验，我理解了同步原语的基本原理。屏障机制虽然概念简单，但实现时需要仔细协调多个线程。

实验中遇到程序卡死问题，调试后发现是对全局变量理解有误，计数器没有正确更新。这让我认识到并发编程中变量状态管理的重要性，必须清楚每个变量的作用和生命周期。

解决问题的过程中，我学会了通过输出中间变量来调试并发程序，也体会到代码简洁性的价值。删除无用变量后程序逻辑更清晰。

这个实验让我深刻理解了条件变量和互斥锁的配合使用：最后到达的线程唤醒所有等待线程，其他线程在条件变量上等待。这种设计既保证同步正确性，又避免了资源浪费。

Lab: networking

1. Job (hard)

1）实验目标

在 kernel/e1000.c 中完成 e1000_transmit（） 和 e1000_recv（），以便驱动程序可以发送和接收数据包，为 xv6 操作系统实现一个完整的网络设备驱动程序，使 xv6 具备网络通信能力。

2）实验步骤

(1) 发送功能：

① 实现发送队列检查

```
int
e1000_transmit(struct mbuf *m) //m 要发送的数据包
//通过 E1000 网卡发送网络数据包
// 工作原理：
// 1. 使用环形发送队列(TX Ring)管理待发送的数据包
//  2. 通过 DMA 方式让网卡直接访问内存中的数据
// 3. 使用描述符(Descriptor)告诉网卡数据的位置和属性
{
  acquire(&e1000_lock);

  // 获取当前发送队列尾指针
  uint32 tdt = regs[E1000_TDT];

  // 检查描述符是否可用 (DD bit = 1 表示可用)
  if (!(tx_ring[tdt].status & E1000_TXD_STAT_DD)) {
    // 队列满，无法发送
    release(&e1000_lock);
    return -1;
  }

  // 释放之前可能存在的 mbuf
  if (tx_mbufs[tdt]) {
    mbuffree(tx_mbufs[tdt]);
  }

  // 填充发送描述符
```

```
  tx_ring[tdt].addr = (uint64)m->head;          // 数据缓冲区地址
  tx_ring[tdt].length = m->len;                 // 数据长度
  tx_ring[tdt].cso = 0;                         // 校验和偏移
  tx_ring[tdt].cmd = E1000_TXD_CMD_EOP |        // End of Packet
                     E1000_TXD_CMD_RS;          // Report Status
  tx_ring[tdt].status = 0;                      // 清除状态位
  tx_ring[tdt].css = 0;                         // 校验和起始

  // 保存 mbuf 指针，用于后续释放
  tx_mbufs[tdt] = m;

  // 更新队列尾指针，通知硬件有新的数据包要发送
  regs[E1000_TDT] = (tdt + 1) % TX_RING_SIZE;

  release(&e1000_lock);
  return 0;  // 发送成功
}
```

```
acquire(&e1000_lock);
printf("e1000_transmit: begin\n");
// 获取当前发送队列尾指针
uint32 tdt = regs[E1000_TDT];

// 检查描述符是否可用 (DD bit = 1 表示可用)
if (!(tx_ring[tdt].status & E1000_TXD_STAT_DD)) {
  // 队列满，无法发送
  release(&e1000_lock);
  printf("e1000_transmit: TX ring full\n");
  return -1;
}

// 释放之前可能存在的mbuf
if (tx_mbufs[tdt]) {
  mbuffree(tx_mbufs[tdt]);
}

// 填充发送描述符
tx_ring[tdt].addr = (uint64)m->head;          // 数据缓冲区地址
tx_ring[tdt].length = m->len;                 // 数据长度
tx_ring[tdt].cso = 0;                         // 校验和偏移
tx_ring[tdt].cmd = E1000_TXD_CMD_EOP |        // End of Packet
                   E1000_TXD_CMD_RS;          // Report Status
tx_ring[tdt].status = 0;                      // 清除状态位
tx_ring[tdt].css = 0;                         // 校验和起始

// 保存mbuf指针，用于后续释放
tx_mbufs[tdt] = m;

// 更新队列尾指针，通知硬件有新的数据包要发送
regs[E1000_TDT] = (tdt + 1) % TX_RING_SIZE;

release(&e1000_lock);
printf("e1000_transmit: packet sent, TDT updated to %d\n", regs[
return 0;  // 发送成功
```

(2) 接收功能：

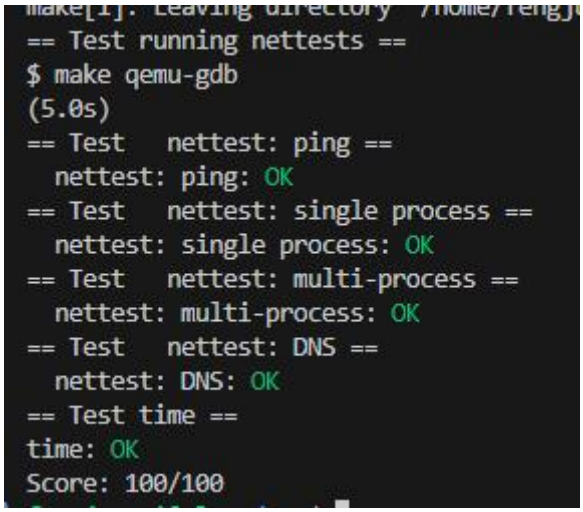
① 实现接收

```
static void
e1000_recv(void)
{
  acquire(&e1000_lock);

  // 处理所有可用的接收数据包
  while (1) {
    // 计算下一个要处理的描述符索引
    uint32 rdt = regs[E1000_RDT];
    uint32 next = (rdt + 1) % RX_RING_SIZE;

    // 检查描述符是否有新数据
    if (!(rx_ring[next].status & E1000_RXD_STAT_DD)) {
      break;  // 没有更多数据包
    }

    // 获取接收到的 mbuf
    struct mbuf *m = rx_mbufs[next];
    if (!m) {
      panic("e1000_recv: no mbuf");
    }

    // 设置接收数据的长度
    m->len = rx_ring[next].length;

    // 将数据包传递给网络栈
    net_rx(m);

    // 分配新的 mbuf 用于下次接收
    rx_mbufs[next] = mbufalloc(0);
    if (!rx_mbufs[next]) {
      panic("e1000_recv: mbufalloc failed");
    }

    // 重新设置接收描述符
    rx_ring[next].addr = (uint64)rx_mbufs[next]->head;
```

```
    rx_ring[next].status = 0;  // 清除状态位

    // 更新队列尾指针，告诉硬件有新的缓冲区可用
    regs[E1000_RDT] = next;
  }

  release(&e1000_lock);
}
void
e1000_intr(void)
{
  // 读取中断原因寄存器
  uint32 icr = regs[E1000_ICR];

  // 检查是否是接收中断
  if (icr & E1000_ICR_RXT0) {
    e1000_recv();
  }

  // 检查是否有其他中断类型
  if (icr & E1000_ICR_TXDW) {
    // 发送完成中断 - 通常不需要特殊处理
  }
}
```

(3) 测试

```
Make grade
```



3）实验中遇到的问题和解决方法

```
Timeout! (34.5s)
== Test   nettest: ping ==
  nettest: ping: FAIL
    ...
        init: starting sh
        $ nettests
        nettests running on port 26099
        testing ping: panic: acquire
        qemu-system-riscv64: terminating on signal 15 from pid 28941 (make)
    MISSING '^testing ping: OK$'
== Test   nettest: single process ==
  nettest: single process: FAIL
    ...
        init: starting sh
        $ nettests
        nettests running on port 26099
        testing ping: panic: acquire
        qemu-system-riscv64: terminating on signal 15 from pid 28941 (make)
    MISSING '^testing single-process pings: OK$'
== Test   nettest: multi-process ==
  nettest: multi-process: FAIL
    ...
        init: starting sh
        $ nettests
        nettests running on port 26099
        testing ping: panic: acquire
        qemu-system-riscv64: terminating on signal 15 from pid 28941 (make)
    MISSING '^testing multi-process pings: OK$'
== Test   nettest: DNS ==
  nettest: DNS: FAIL
    ...
        init: starting sh
        $ nettests
        nettests running on port 26099
        testing ping: panic: acquire
        qemu-system-riscv64: terminating on signal 15 from pid 28941 (make)
    MISSING '^DNS OK$'
% 0
```

问题：即使代码检查过逻辑没有问题，仍然报错。

检查过程中，发现是由于 printf 的输出。中断处理函数运行在中断上下文中，printf 可能会调用系统调用或锁，在中断上下文中是危险的，可能导致了死锁或系统崩溃。

解决方式：删除 printf 语句。

4）实验心得

通过这个网络驱动实验，我深入理解了操作系统中网络设备驱动的工作原理和实现细节。网络驱动看似只是发送和接收数据包，但实际实现需要仔细处理硬件寄存器、环形缓冲区、内存管理等多个方面，每个细节都关系到系统稳定性。

实验中遇到的 printf 问题让我深刻认识到中断处理的特殊性。中断处理函数运行在特殊的执行环境中，不能随意调用可能阻塞的函数，这个错误提醒我在系统编程中必须时刻注意执行上下文。网络驱动中大量使用锁机制来保护共享资源，通过实现这些锁的使用，我更好地理解了在多线程环境下如何保证数据一致性。

发送和接收都使用环形缓冲区的设计让我印象深刻，这种设计既提高了效率又简化了内存管理。通过 tail 指针的循环更新，实现了高效的数据包处理流程。整个实验过程让我体会到系统级编程的复杂性和严谨性要求，也为深入理解操作系统底层机制打下了坚实基础。

Lab: locks

1. Memory allocator (moderate)

1）实验目标

为每个 CPU 维护独立的空闲页面链表，并且每个 CPU 都有自己的锁，当某个 CPU 的空闲列表为空时候，可以从其他 CPU 窃取页面。

2）实验步骤

(1) 在 kalloc.c 中修改数据结构

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU];  // kmem 结构修改为每个 CPU 一个，减少锁的竞争
//初始化部分 一次初始化每个 CPU 的锁 修改为如下代码
```

```
// 原来的单一 kmem结构体被修改为每个CPU一个kmem结构体
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU];
```

(2) 修改 kinit()函数初始化所有锁

```
void
kinit()
{
  for(int i = 0; i < NCPU; i++) {
    initlock(&kmem[i].lock, "kmem");
  }
  freerange(end, (void*)PHYSTOP);
}
```

```
void
kinit()
{
  for(int i = 0; i < NCPU; i++) {
    initlock(&kmem[i].lock, "kmem");
  }
  freerange(end, (void*)PHYSTOP);
}
```

(3) 修改 kfree()函数使用对应 CPU 的锁和链表

```
  // 获取当前 CPU ID ，并使用对应的锁 和 freelist
  push_off();//关中断防止 CPU 切换
  int cpu_id = cpuid();
  acquire(&kmem[cpu_id].lock);
  r->next = kmem[cpu_id].freelist;
```

```
kmem[cpu_id].freelist = r;
release(&kmem[cpu_id].lock);
pop_off(); // 恢复中断
```

```
// 获取当前CPU ID ，并使用对应的锁 和 freelist
push_off();//关中断防止CPU切换
int cpu_id = cpuid();
acquire(&kmem[cpu_id].lock);
r->next = kmem[cpu_id].freelist;
kmem[cpu_id].freelist = r;
release(&kmem[cpu_id].lock);
pop_off(); // 恢复中断
```

(4) 修改 kalloc()函数使用对应 CPU 的锁和链表

```
push_off(); // 关中断防止 CPU 切换
 int cpu_id = cpuid();
 acquire(&kmem[cpu_id].lock);
 r = kmem[cpu_id].freelist;
 if(r)
   kmem[cpu_id].freelist = r->next;
 release(&kmem[cpu_id].lock);

 if(!r){
   for(int i=0;i<NCPU;i++){
     if(i==cpu_id)  continue;// 跳过当前 CPU

     acquire(&kmem[i].lock);
     r = kmem[i].freelist;
     if(r){
       kmem[i].freelist = r->next;
       release(&kmem[i].lock);
       break; // 找到一个空闲页，跳出循环
     }
     release(&kmem[i].lock);
   }
 }
 pop_off(); // 恢复中断
```

```
push_off(); // 关中断防止CPU切换
int cpu_id = cpuid();
acquire(&kmem[cpu_id].lock);
r = kmem[cpu_id].freelist;
if(r)
  kmem[cpu_id].freelist = r->next;
release(&kmem[cpu_id].lock);

if(!r){
  for(int i=0;i<NCPU;i++){
    if(i==cpu_id)  continue;// 跳过当前CPU

    acquire(&kmem[i].lock);
    r = kmem[i].freelist;
    if(r){
      kmem[i].freelist = r->next;
      release(&kmem[i].lock);
      break; // 找到一个空闲页，跳出循环
    }
    release(&kmem[i].lock);
  }
}
```

(5) 测试

```
$ make qemu-gdb
(115.5s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (18.0s)
```

3）实验中遇到的问题和解决方法

在 kalloc()函数中实现页面窃取时，如果不小心处理锁的获取和释放顺序，可能会造成死锁。特别是在多个 CPU 同时尝试从对方窃取页面时。

解决方法：采用简单的锁获取策略，每次只持有一个 CPU 的锁，并且按照固定的顺序遍历其他 CPU。在获取其他 CPU 的锁之前，先释放当前 CPU 的锁，避免同时持有多个锁。

4）实验心得

通过这个 per-CPU 内存分配器的实验，我理解了将全局的单一内存链表改为每个 CPU 独立的链表，可以大大减少了不同 CPU 之间的锁竞争，这种设计思想在高并发系统中非常重要。

实验中最大的收获是理解了中断处理和 CPU 调度的复杂性。在内核编程中，必须时刻考虑进程可能在任何时候被调度到不同的 CPU 上执行，因此需要通过关闭中断来保证关键代码段的原子性。这让我认识到系统编程中对执行环境控制的重要性。

页面窃取机制的实现让我体会到了负载均衡的思想。当某个 CPU 的内存不足时，可以从其他 CPU 借用资源，这种设计既保证了性能又保证了资源的充分利用。同时，在实现窃取机制时需要仔细处理锁的顺序，避免死锁问题，这培养了我在并发编程中的谨慎思维。

2. Buffer cache (hard)

1）实验目标

在 xv6 的原始实现中，buffer cache 使用单一的全局锁 bcache.lock 来保护整个缓存结构，这在多核系统中会成为严重的性能瓶颈。当多个 CPU 核心同时进行磁盘 I/O 操作时，它们必须串行地获取这个全局锁，导致大量的锁竞争和等待时间。

本实验的目标是修改 buffer cache (kernel/bio.c) 以减少锁竞争，在多核系统上提高性能。

2）实验步骤

(1) 修改数据结构，从单锁到多锁

```
#define NBUCKET 13
struct bucket {
  struct spinlock lock;  //每个桶独立的锁
  struct buf head;       //每个桶独立的 LRU 链表
};

struct bucket bcache[NBUCKET];//13 个桶，
struct buf buf[NBUF];  // 独立存储所有 buffer
```

```
#define NBUCKET 13 //13个桶
struct bucket {
  struct spinlock lock;  //每个桶独立的锁
  struct buf head;//每个桶独立的LRU链表
};
struct bucket bcache[NBUCKET];//13个桶，
struct buf buf[NBUF];  // 独立存储所有buffer
```

(2) 修改初始化函数 - 分布式初始化

```
//循环初始化哈希桶
for (int i = 0; i < NBUCKET; i++) {
  // 每个桶都有独立的 LRU 链表
  initlock(&bcache[i].lock, "bcache");
  bcache[i].head.prev = &bcache[i].head;//头结点的前驱指向自己
  bcache[i].head.next = &bcache[i].head;//头结点的后继指向自己
}
//将所有缓冲区分配到各个桶中
for (struct buf *b = buf; b < buf + NBUF; b++) {
  initsleeplock(&b->lock,"buffer");//初始化每个缓冲区的睡眠锁
  int bucket = (b - buf) % NBUCKET; // 分配到对应的桶
  //设置新节点 b 的后继为原本 head 的后继
  b->next = bcache[bucket].head.next;
  //设置新节点 b 的前驱是 head
  b->prev = &bcache[bucket].head;
  //更新原来 head 后继结点的前驱指针为 b
```

```
        bcache[bucket].head.next->prev = b;
        //更新 head 的后继指针指向新的节点
        bcache[bucket].head.next = b;
      }
```

```
// 初始化桶
for (int i = 0; i < NBUCKET; i++) {
  initlock(&bcache[i].lock, "bcache");
  // 每个桶都有独立的LRU链表
  bcache[i].head.prev = &bcache[i].head;
  bcache[i].head.next = &bcache[i].head;
}
for (struct buf *b = buf; b < buf + NBUF; b++) {
  initsleeplock(&b->lock,"buffer");
  int bucket = (b - buf) % NBUCKET; // 分配到对应的桶
  b->next = bcache[bucket].head.next;
  b->prev = &bcache[bucket].head;
  bcache[bucket].head.next->prev = b;
  bcache[bucket].head.next = b;
}
```

(3) 修改 bget 函数 - 分桶查找和跨桶偷取

```
    static struct buf*
    bget(uint dev, uint blockno)//获取指定设备和块号的缓冲区
    {
      int bucket =blockno % NBUCKET; // 根据 blockno 选择桶
      acquire(&bcache[bucket].lock); // 获取对应桶的锁
      struct buf *b;
      //第一步：在目标桶中查找已经存在的缓冲区
      for(b = bcache[bucket].head.next; b != &bcache[bucket].head;
b = b->next){
        //检查设备号和块号是否匹配
        if(b->dev == dev && b->blockno == blockno){
          b->refcnt++;//引用计数增加
           release(&bcache[bucket].lock); // 释放桶的锁
           acquiresleep(&b->lock); // 获取 buffer 的 sleep lock
           return b;//返回找到的缓冲区
        }
      }

      // 第二步：在目标桶中查找空闲 buffer
      for(b = bcache[bucket].head.prev; b != &bcache[bucket].head;
b = b->prev){
        if(b->refcnt == 0) {//检查计数是否为 0
          b->dev = dev;//找到新的设备号和块号
          b->blockno = blockno;
```

```
          b->valid = 0;//标记数据无效，需要从磁盘读取
          b->refcnt = 1;//设置引用计数为1
          release(&bcache[bucket].lock);//释放锁
          acquiresleep(&b->lock); // 获取buffer的sleep lock
          return b; // 返回找到的buffer
        }
      }
      //第三步：目标桶满了，需要从其他桶偷取
      release(&bcache[bucket].lock); // 释放目标桶的锁

      //遍历其他桶查找空闲的buffer
      for (int i = 0; i < NBUCKET; i++) {
        if (i == bucket) continue; // 跳过当前桶
        acquire(&bcache[i].lock); // 获取其他桶的锁
        for (b = bcache[i].head.prev; b != &bcache[i].head; b =
b->prev) {
          if (b->refcnt == 0) { // 找到空闲的buffer
            //从当前的桶中移除
            b->next->prev = b->prev;
            b->prev->next = b->next;
            release(&bcache[i].lock); // 释放其他桶的锁
          //将缓冲区插入到目标桶
            acquire(&bcache[bucket].lock); // 重新获取目标桶的锁
            b->next = bcache[bucket].head.next;
            b->prev = &bcache[bucket].head;
            bcache[bucket].head.next->prev = b;
            bcache[bucket].head.next = b;
            //配置buffer
            b->dev = dev;//配置设备号和块号
            b->blockno = blockno;
            b->valid = 0;
            b->refcnt = 1;
            release(&bcache[bucket].lock); // 释放目标桶的锁
            acquiresleep(&b->lock); // 获取buffer的sleep lock
            return b;
          }
        }
        release(&bcache[i].lock); // 释放其他桶的锁
```

```
    }
    panic("bget: no buffers");
  }
```

```c
static struct buf*
bget(uint dev, uint blockno)
{
  int bucket =blockno % NBUCKET; // 根据blockno选择桶
  acquire(&bcache[bucket].lock); // 获取对应桶的锁
  struct buf *b;
  //第一步：在目标桶中查找
  for(b = bcache[bucket].head.next; b != &bcache[bucket].head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache[bucket].lock); // 释放桶的锁
      acquiresleep(&b->lock); // 获取buffer的sleep lock
      return b;
    }
  }

  // 第二步：在目标桶中查找空闲buffer
  for(b = bcache[bucket].head.prev; b != &bcache[bucket].head; b = b->prev){
    if(b->refcnt == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;
      release(&bcache[bucket].lock);
      acquiresleep(&b->lock); // 获取buffer的sleep lock
      return b; // 在目标桶中找到空闲的buffer
    }
  }
  //第三步：目标桶满了，需要从其他桶偷取
  release(&bcache[bucket].lock); // 释放桶的锁

  //遍历其他桶查找空闲的buffer
  for (int i = 0; i < NBUCKET; i++) {
    if (i == bucket) continue; // 跳过当前桶
    acquire(&bcache[i].lock); // 获取其他桶的锁
    for (b = bcache[i].head.prev; b != &bcache[i].head; b = b->prev) {
      if (b->refcnt == 0) {
        // 找到空闲的buffer
        //从当前的桶中移除
        b->next->prev = b->prev;
        b->prev->next = b->next;
        release(&bcache[i].lock); // 释放其他桶的锁

        acquire(&bcache[bucket].lock); // 重新获取目标桶的锁
        b->next = bcache[bucket].head.next;
        b->prev = &bcache[bucket].head;
        bcache[bucket].head.next->prev = b;
        bcache[bucket].head.next = b;
        //配置buffer
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
        release(&bcache[bucket].lock); // 释放目标桶的锁
        acquiresleep(&b->lock); // 获取buffer的sleep lock
        return b;
      }
    }
    release(&bcache[i].lock); // 释放其他桶的锁
  }

  panic("bget: no buffers");
}
```

(4) brelse 修改

```
Void brelse(struct buf *b) //释放缓冲区
{
 if(!holdingsleep(&b->lock))//安全检查 只有锁持有者才能释放缓冲区
    panic("brelse");
  releasesleep(&b->lock);//释放睡眠锁
  // 先尝试根据 blockno 找到对应的桶
  int bucket = b->blockno % NBUCKET;//在预期桶中查找缓冲区
  acquire(&bcache[bucket].lock);
```

```
      // 检查buffer是否在预期的桶中
      struct buf *p;
      for (p = bcache[bucket].head.next; p != &bcache[bucket].head;
p = p->next) {
        if (p == b) {      // 找到了buffer
          b->refcnt--;      // 减少引用计数
          if (b->refcnt == 0) { //如果没有其他使用者了
            b->next->prev = b->prev;// 前驱节点直接指向后继节点
            b->prev->next = b->next;// 后继节点指向前驱节点
            b->next = bcache[bucket].head.next;
            b->prev = &bcache[bucket].head;
            bcache[bucket].head.next->prev = b;
            bcache[bucket].head.next = b;
          }
          release(&bcache[bucket].lock);
          return;
        }
      }
      release(&bcache[bucket].lock);

      // 如果在预期桶中没找到，说明buffer被移到了其他桶，需要遍历查找
      for (int i = 0; i < NBUCKET; i++) {
        if (i == bucket) continue; // 已经检查过了
        acquire(&bcache[i].lock);
        for (p = bcache[i].head.next; p != &bcache[i].head; p =
p->next) {
          if (p == b) {
            b->refcnt--;
            if (b->refcnt == 0) {
              // no one is waiting for it.
              b->next->prev = b->prev;
              b->prev->next = b->next;
              b->next = bcache[i].head.next;
              b->prev = &bcache[i].head;
              bcache[i].head.next->prev = b;
              bcache[i].head.next = b;
            }
            release(&bcache[i].lock);
```

```
        return;
      }
    }
    release(&bcache[i].lock);
  }
  panic("brelse: buffer not found in any bucket");
}
```

```
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  // 先尝试根据blockno找到对应的桶
  int bucket = b->blockno % NBUCKET;
  acquire(&bcache[bucket].lock);

  // 检查buffer是否在预期的桶中
  struct buf *p;
  for (p = bcache[bucket].head.next; p != &bcache[bucket].head; p = p->next) {
    if (p == b) {
      // 找到了buffer
      b->refcnt--;
      if (b->refcnt == 0) {
        // no one is waiting for it.
        b->next->prev = b->prev;
        b->prev->next = b->next;
        b->next = bcache[bucket].head.next;
        b->prev = &bcache[bucket].head;
        bcache[bucket].head.next->prev = b;
        bcache[bucket].head.next = b;
      }
      release(&bcache[bucket].lock);
      return;
    }
  }
  release(&bcache[bucket].lock);

  // 如果在预期桶中没找到，说明buffer被移到了其他桶，需要遍历查找
  for (int i = 0; i < NBUCKET; i++) {
    if (i == bucket) continue; // 已经检查过了
    acquire(&bcache[i].lock);
    for (p = bcache[i].head.next; p != &bcache[i].head; p = p->next) {
      if (p == b) {
        b->refcnt--;
        if (b->refcnt == 0) {
          // no one is waiting for it.
          b->next->prev = b->prev;
          b->prev->next = b->next;
          b->next = bcache[i].head.next;
          b->prev = &bcache[i].head;
          bcache[i].head.next->prev = b;
          bcache[i].head.next = b;
        }
        release(&bcache[i].lock);
        return;
      }
    }
    release(&bcache[i].lock);
  }
  panic("brelse: buffer not found in any bucket");
}
```

(5) bpin 修改

```
Void bpin(struct buf *b) {//固定缓冲区 防止被回收
  // 先尝试根据 blockno 找到对应的桶
  int bucket = b->blockno % NBUCKET;
  acquire(&bcache[bucket].lock);

  // 检查 buffer 是否在预期的桶中
  struct buf *p;
```

```
    for (p = bcache[bucket].head.next; p != &bcache[bucket].head;
p = p->next) {
      if (p == b) {
        b->refcnt++;      release(&bcache[bucket].lock);
        return;
      }
    }
   release(&bcache[bucket].lock);
  // 如果在预期桶中没找到，遍历其他桶
    for (int i = 0; i < NBUCKET; i++) {
      if (i == bucket) continue; // 已经检查过了
      acquire(&bcache[i].lock);
      for (p = bcache[i].head.next; p != &bcache[i].head; p =
p->next) {
        if (p == b) {
          b->refcnt++;        release(&bcache[i].lock);
          return;
        }
      }
      release(&bcache[i].lock);
    }
    panic("bpin: buffer not found in any bucket");
  }
```

```
void
bpin(struct buf *b) {
  // 先尝试根据blockno找到对应的桶
  int bucket = b->blockno % NBUCKET;
  acquire(&bcache[bucket].lock);

  // 检查buffer是否在预期的桶中
  struct buf *p;
  for (p = bcache[bucket].head.next; p != &bcache[bucket].head; p = p->next) {
    if (p == b) {
      b->refcnt++;
      release(&bcache[bucket].lock);
      return;
    }
  }
  release(&bcache[bucket].lock);

  // 如果在预期桶中没找到，遍历其他桶
  for (int i = 0; i < NBUCKET; i++) {
    if (i == bucket) continue; // 已经检查过了
    acquire(&bcache[i].lock);
    for (p = bcache[i].head.next; p != &bcache[i].head; p = p->next) {
      if (p == b) {
        b->refcnt++;
        release(&bcache[i].lock);
        return;
      }
    }
    release(&bcache[i].lock);
  }
  panic("bpin: buffer not found in any bucket");
}
```

(6) bunpin 修改

```
void
bunpin(struct buf *b) { //取消固定缓冲区 允许被回收
  // 先尝试根据blockno找到对应的桶
  int bucket = b->blockno % NBUCKET;
  acquire(&bcache[bucket].lock);
  // 检查buffer是否在预期的桶中
  struct buf *p;
//在预期桶中搜索目标缓冲区
  for (p = bcache[bucket].head.next; p != &bcache[bucket].head;
p = p->next) {
    if (p == b) {
      b->refcnt--;
      release(&bcache[bucket].lock);
      return;
    }
  }
  release(&bcache[bucket].lock);
  // 如果在预期桶中没找到，遍历其他桶
  for (int i = 0; i < NBUCKET; i++) {
    if (i == bucket) continue; // 已经检查过了
    acquire(&bcache[i].lock);
    for (p = bcache[i].head.next; p != &bcache[i].head; p =
p->next) {
      if (p == b) {
        b->refcnt--;
        release(&bcache[i].lock);
        return;
      }
    }
    release(&bcache[i].lock);
  }
  panic("bunpin: buffer not found in any bucket");
}
```

```c
void
bunpin(struct buf *b) {
  // 先尝试根据blockno找到对应的桶
  int bucket = b->blockno % NBUCKET;
  acquire(&bcache[bucket].lock);

  // 检查buffer是否在预期的桶中
  struct buf *p;
  for (p = bcache[bucket].head.next; p != &bcache[bucket].head; p = p->next) {
    if (p == b) {
      b->refcnt--;
      release(&bcache[bucket].lock);
      return;
    }
  }
  release(&bcache[bucket].lock);

  // 如果在预期桶中没找到，遍历其他桶
  for (int i = 0; i < NBUCKET; i++) {
    if (i == bucket) continue; // 已经检查过了
    acquire(&bcache[i].lock);
    for (p = bcache[i].head.next; p != &bcache[i].head; p = p->next) {
      if (p == b) {
        b->refcnt--;
        release(&bcache[i].lock);
        return;
      }
    }
    release(&bcache[i].lock);
  }
  panic("bunpin: buffer not found in any bucket");
}
```

(7) 测试

```
== Test running kalloctest ==
$ make qemu-gdb
(116.0s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (18.6s)
== Test running bcachetest ==
$ make qemu-gdb
(14.5s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (213.7s)
== Test time ==
time: OK
Score: 70/70
```

3）实验中遇到的问题和解决方法

问题：bget 函数中缺少 sleep lock 获取，在找到匹配的 buffer 后，没有获取 sleep lock 就直接返回：这样子违反了接口约定，调用者可能在没有锁保护的情况下访问 buffer 数据，可能导致数据竞争和不一致性。

解决方法：添加锁的获取：acquiresleep(&b->lock);

问题： brelse、bpin、bunpin 函数的低效查找策略，原始实现是直接遍历所有桶，或只在预期桶中查找。第一种情况：当 buffer 被跨桶偷取后，无法找到它，导致 panic；第二种情况：每次操作都要获取所有桶的锁，导致严重的锁争用。

解决方法：先快速查找，再全面查找：第一阶段：检查 buffer 最可能在的桶（预期桶），第二阶段：如果没找到，再遍历其他桶。

4）实验心得

通过这个 Buffer Cache 优化实验，我深刻体会到了分布式锁设计在高并发系统中的重要性和复杂性。将单一的全局锁拆分为多个桶锁的设计思想让我认识到，在多核系统中，锁竞争往往是性能瓶颈的根源。通过哈希分桶的方式，不同的 block 可以并行访问，大大提高了系统的并发性能，这种"分而治之"的思想在系统设计中具有广泛的应用价值。

实验中最大的挑战是在保证数据一致性的同时优化性能。跨桶偷取机制的实现让我深刻理解了这种平衡的复杂性，既要保证 buffer 能够在桶间灵活移动以提高资源利用率，又要确保在移动过程中不会出现数据竞争和死锁问题。这需要精心设计锁的获取和释放顺序，任何疏忽都可能导致系统崩溃或数据不一致。

在实现 brelse、bpin、bunpin 函数时，我意识到在分布式数据结构中，错误处理变得更加复杂。由于 buffer 可能被移动到任意桶中，简单的查找策略可能导致系统崩溃。这让我学会了设计 robust 的查找策略，先在最可能的位置查找，再进行全面搜索，确保系统的健壮性。同时，通过修复 sleep lock 的获取问题，我深刻理解了系统编程中接口约定的重要性，每个函数都有明确的前置条件和后置条件，违反这些约定会导致难以调试的并发错误。

虽然新的查找策略在最坏情况下需要遍历所有桶，但在实际应用中，由于局部性原理，大多数 buffer 都会在预期的桶中找到，因此平均性能得到了显著提升。这让我理解了在系统优化中，平均情况往往比最坏情况更重要，算法复杂度分析需要结合实际的使用场景。

Lab: file system

1. Large files (moderate)

1）实验目标

更改 xv6 文件系统代码，以支持每个 inode 中的“双间接”块，其中包含 256 个单间接块地址，每个地址最多可以包含 256 个数据块地址。结果将是，一个文件将能够包含多达 65803 个块，或 256*256+256+11 个块（11 个而不是 12 个，因为我们将牺牲一个直接块号来换取双间接块）。

2）实验步骤

(1) 修改常量定义

在 kernel/fs.h 中修改：

```
#define NDIRECT 11          // 从 12 改为 11（减少 1 个直接块）
#define NINDIRECT (BSIZE / sizeof(uint))  // 256 (1024/4)
#define NDINDIRECT (NINDIRECT * NINDIRECT) // 256*256 = 65536
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
// inode 结构保持不变，因为 addrs 数组大小仍然是 NDIRECT+2
// addrs[0-10]: 直接块  addrs[11]: 间接块   addrs[12]: 双重间接块

Uint addrs[NDIRECT + 2]
```

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDINDIRECT (NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
```

(2) 修改 addres ，保持 addrs 数组大小不变。

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses
};
```

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device num
  uint inum;          // Inode numb
  int ref;            // Reference
  struct sleeplock lock; // protect
  int valid;          // inode has

  short type;         // copy of di
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+2];
};
```

(3) 修改 bmap() 函数

在 kernel/fs.c 中，找到 bmap() 函数并修改：

```
static uint
bmap(struct inode *ip, uint bn)//双重间接块处理部分
{
```

```
    ••前面的代码••--
    // 处理双重间接块
    bn -= NINDIRECT;//调整快号
    if(bn < NDINDIRECT){
      // 预计算索引，避免重复计算
      uint index1 = bn / NINDIRECT;
      uint index2 = bn % NINDIRECT;
      // 分配双间接块（如果需要）
      if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
        ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
      }
      // 读取双间接块
      bp = bread(ip->dev, addr);
      a = (uint*)bp->data;
      // 为双间接块分配单间接块（如果需要）
      if ((addr = a[index1]) == 0) {
        a[index1] = addr = balloc(ip->dev);
        log_write(bp);
      }
      brelse(bp);
      // 读取单间接块
      bp = bread(ip->dev, addr);
      a = (uint *)bp->data;
      // 为单间接块分配数据块（如果需要）
      if ((addr = a[index2]) == 0) {
        //分配新的单简介块
        a[index2] = addr = balloc(ip->dev);
        log_write(bp);
      }
      brelse(bp);
      return addr;
    }
    panic("bmap: out of range");
  }
```

```
if(bn < NDINDIRECT){
  // 如果文件的双间接块不存在，则分配一个
  if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
    addr = balloc(ip->dev);
    if (addr == 0)
      return 0;
    ip->addrs[NDIRECT + 1] = addr;
  }

  // 读取双间接块
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;

  // 计算在单间接块数组中的索引，即第几个单间接块
  uint index1 = bn / NINDIRECT;

  // 如果这个单间接块不存在，则分配一个
  if ((addr = a[index1]) == 0) {
    addr = balloc(ip->dev);
    if (addr == 0)
      return 0;
    a[bn / NINDIRECT] = addr;
    log_write(bp);  // Record changes in the log
  }
  brelse(bp);

  // 读取相应的单间接块
  bp = bread(ip->dev, addr);
  a = (uint *)bp->data;

  // 计算在单间接块中的索引，即单间接块中的第几个数据
  uint index2 = bn % NINDIRECT;

  // 如果这个数据块不存在，则分配一个
  if ((addr = a[index2]) == 0) {
    addr = balloc(ip->dev);
    if (addr == 0)
      return 0;
    a[bn % NINDIRECT] = addr;
    log_write(bp);  // Record changes in the log
  }
  brelse(bp);
  return addr;  // Returns the actual data block
}
```

(4) 修改 itrunc() 函数

```
void
itrunc(struct inode *ip)//itrunc 函数的双重间接块释放部分
{
  • • •
  // 释放直接块
  // 释放间接块
  • • •
  if(ip->addrs[NDIRECT+1]){
     // 读取双间接块
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;
      //遍历所有的单简洁块
    for (i = 0; i < NINDIRECT; ++i) {
      if (a[i] == 0) continue;//跳过未分配的单简介块
      // 读取单间接块
      struct buf* bp2 = bread(ip->dev, a[i]);
      uint* b = (uint*)bp2->data;
```

```
      //释放所有的数据块
        for (j = 0; j < NINDIRECT; ++j) {
          if (b[j])
            bfree(ip->dev, b[j]); // 释放数据块
        }
        brelse(bp2);//释放单间接块的缓冲区
        bfree(ip->dev, a[i]); // 释放单间接块
        a[i] = 0;
      }
      brelse(bp);
      bfree(ip->dev, ip->addrs[NDIRECT + 1]); // 释放双间接块
      ip->addrs[NDIRECT + 1] = 0;
    }
    ip->size = 0;
    iupdate(ip);
  }
```

```
  if (ip->addrs[NDIRECT + 1]) {
    // 读取双间接块
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;

    for (i = 0; i < NINDIRECT; ++i) {
      if (a[i] == 0) continue;

      // 读取单间接块
      struct buf* bp2 = bread(ip->dev, a[i]);
      uint* b = (uint*)bp2->data;
      for (j = 0; j < NINDIRECT; ++j) {
        if (b[j])
          bfree(ip->dev, b[j]); // 释放数据块
      }
      brelse(bp2);

      bfree(ip->dev, a[i]); // 释放单间接块
      a[i] = 0;
    }
    brelse(bp);

    bfree(ip->dev, ip->addrs[NDIRECT + 1]); // 释放双间接块
    ip->addrs[NDIRECT + 1] = 0;
  }
  ip->size = 0;
  iupdate(ip);
}
```

(5) 编译测试

```
make qemu
bigfile
```

```
$ bigfile
.........................................................................................
.........................................................................................
.........................................................................................
.........................................................................................
.........................................................................................
.........................................................................
wrote 65803 blocks
bigfile done; ok
$
```

3）实验中遇到的问题和解决方法

从截图可以看到，编译过程中出现了一个断言失败的错误：

```
mkfs: mkfs/mkfs.c:85: main: Assertion `(BSIZE % sizeof(struct dinode)) == 0' failed.
make: *** [Makefile:263: fs.img] Aborted (core dumped)
```

这个错误是因为 BSIZE（块大小）不能被 `struct dinode` 的大小整除。

经过仔细分析后发现，问题出现在我最初只修改了 NDIRECT 常量，但忘记了相应调整 addrs 数组的大小。由于文件系统的设计要求 inode 结构在磁盘上的布局必须保持一致性，当我将 NDIRECT 从 12 改为 11 时，如果不同步修改 addrs 数组大小，就会导致结构体大小不匹配。

解决措施是：修改地址数目，在 inode 和 dinode 结构体中修改 addrs 数组为 addrs[NDIRECT + 2]。

4）实验心得

通过这个大文件支持的实验，我深入理解了文件系统中多级索引结构的设计原理和实现细节。双重间接块的引入虽然只是在原有基础上增加了一层间接性，但其复杂度却呈指数级增长，这让我深刻体会到系统设计中的权衡思想。为了支持更大的文件，我们牺牲了一个直接块的访问效率，但换来了文件大小从 12 个块扩展到 65803 个块的巨大提升，这种以空间换时间或以时间换空间的设计思路在系统编程中随处可见。

实验过程中，我对文件系统的层次化设计有了更深的认识。从直接块到间接块再到双重间接块，每一层都有其特定的作用和适用场景。小文件主要使用直接块以获得最快的访问速度，中等大小的文件使用间接块平衡了访问效率和存储容量，而大文件则需要双重间接块来突破容量限制。这种分层设计不仅体现了系统设计的优雅，也展现了在有限资源约束下的最优化策略。

在调试过程中，我深刻体会到了系统编程中错误处理的重要性。文件系统作为操作系统的核心组件，任何小的错误都可能导致数据丢失或系统崩溃。特别是在处理多级指针和复杂的内存管理时，必须确保每一个分配都有对应的释放，每一个引用都是有效的。这种严谨性不仅体现在代码实现上，更体现在设计思维上，需要在编写代码之前就考虑到各种边界条件和异常情况。

2. Symbolic links (moderate)

1）实验目标

本实验的目标是向 xv6 添加符号链接。符号链接（或软链接）通过路径名引用链接文件;当打开符号链接时，内核会遵循指向引用文件的链接。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。

2）实验步骤

(1) 首先，为符号链接创建一个新的系统调用号并注册，向 user/usys.pl、user/user.h 添加一个条目，并在 kernel/sysfile.c 中实现一个空 sys_symlink。

```
#define SYS_symlink 22  // 系统调用号
user/usys.pl
Entry("symlink");
user/user.h
Int symlink(char*,char*);
Syscall.c //进行注册系统调用
extern uint64 sys_symlink(void);
[SYS_symlink] sys_symlink,
```

```
#define SYS_symlink 22      entry("symlink");
```

```
int symlink(const char*, const char*);
```

```
[SYS_symlink] sys_symlink,
```

```
#define T_SYMLINK 4
```

```
extern uint64 sys_symlink(void);
```

(2) 将新的文件类型 （T_SYMLINK） 添加到 kernel/stat.h 以表示符号链接。

```
#define T_SYMLINK 4    // Symbolic link
```

(3) 向 kernel/fcntl.h （O_NOFOLLOW） 添加一个新标志，该标志可用于 open 系统调用。请注意，传递给 open 的标志使用按位 OR 运算符组合，因此新标志不应与任何现有标志重叠。

```
#define O_NOFOLLOW 0x004
```

(4) 实现 symlink 系统调用，以在引用目标的路径处创建一个新的符号链接。请注意，系统调用不需要存在目标即可成功。需要选择某个位置来存储符号链接的目标路径。

```
uint64
sys_symlink(void) {
  char target[MAXPATH], path[MAXPATH];//目标文件路径
  struct inode *ip;//符号链接文件的路径
```

```
    int n;//target 字符串长度
    if ((n = argstr(0, target, MAXPATH)) < 0
      || argstr(1, path, MAXPATH) < 0) {
      return -1;
    }
    begin_op();//开始
    // 创建 inode
    if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {//创建 inode
      end_op();
      return -1;
    }
    // 写入目标路径
    if(writei(ip, 0, (uint64)target, 0, n) != n) {
      iunlockput(ip);
      end_op();
      return -1;
    }
    iunlockput(ip);//释放 inode 引用并解锁
    end_op();//结束操作
    return 0;
  }
```

```
// Generating symbolic links
uint64
sys_symlink(void) {
  char target[MAXPATH], path[MAXPATH];
  struct inode *ip;
  int n;

  if ((n = argstr(0, target, MAXPATH)) < 0
    || argstr(1, path, MAXPATH) < 0) {
    return -1;
  }

  begin_op();
  // create the symlink's inode
  if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
    end_op();
    return -1;
  }
  // write the target path to the inode
  if(writei(ip, 0, (uint64)target, 0, n) != n) {
    iunlockput(ip);
    end_op();
    return -1;
  }

  iunlockput(ip);
  end_op();
  return 0;
}
```

(5) 修改 open 修改 sys_open 处理符号链接.

在 kernel/sysfile.c 中修改 sys_open 函数：

```
uint64
sys_open(void)//处理符号连接的核心逻辑
{
  ···已有代码···
  // handle the symlink
  if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
    if((ip = follow_symlink(ip)) == 0) {
      // 此处不用调用 iunlockput()释放锁
      // follow_symlinktest()返回失败时,锁在函数内已经被释放
      end_op();
      return -1;
    }
  }
 ···已有代码···
}
```

```
// handle the symlink - lab 9.2
if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
  if((ip = follow_symlink(ip)) == 0) {
    end_op();
    return -1;
  }
}
```

(6) 实现符号链接跟随函数

在 kernel/sysfile.c 中添加：

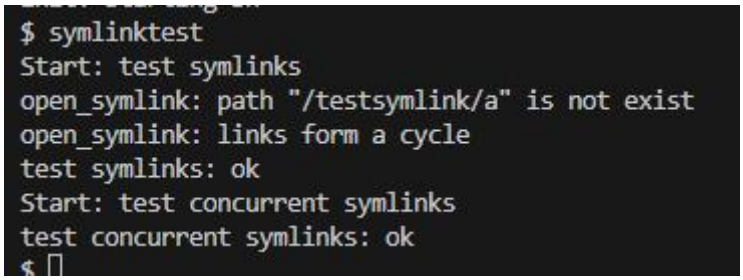
```
static struct inode*
follow_symlink(struct inode* ip) {
//递归解析符号链接 直到找到最终的目标文件
  uint inums[NSYMLINK];//存储已经访问的 inode 编号
  int i, j;
  char target[MAXPATH];//目标路径
  for(i = 0; i < NSYMLINK; ++i) {//逐层解析符号链接
    inums[i] = ip->inum;
    // read the target path from symlink file
    if(readi(ip, 0, (uint64)target, 0, MAXPATH) <= 0) {
      iunlockput(ip);
      printf("open_symlink: open symlink failed\n");
      return 0;
    }
```

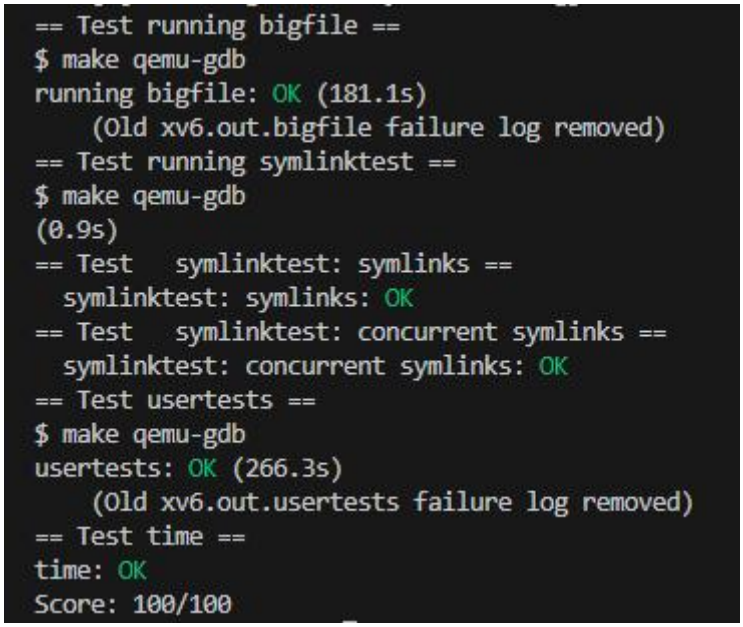
```
      iunlockput(ip);//释放当前符号链接的 inode
      if((ip = namei(target)) == 0) {//解析路径 获得目标 inode
    printf("open_symlink: path \"%s\" is not exist\n", target);
        return 0;
      }
      for(j = 0; j <= i; ++j) {//循环检测
        if(ip->inum == inums[j]) {//检测到循环引用
          printf("open_symlink: links form a cycle\n");
          return 0;
        }
      }
      ilock(ip);//锁定目标 inode 并检查类型
      if(ip->type != T_SYMLINK) {
        return ip;
      }
    }
    iunlockput(ip);
    printf("open_symlink: the depth of links reaches the
limit\n");
    return 0;
  }
```

```
tatic struct inode*
ollow_symlink(struct inode* ip) {
  uint inums[NSYMLINK];
  int i, j;
  char target[MAXPATH];

  for(i = 0; i < NSYMLINK; ++i) {
    inums[i] = ip->inum;
    // read the target path from symlink file
    if(readi(ip, 0, (uint64)target, 0, MAXPATH) <= 0) {
      iunlockput(ip);
      printf("open_symlink: open symlink failed\n");
      return 0;
    }
    iunlockput(ip);

    // get the inode of target path
    if((ip = namei(target)) == 0) {
      printf("open_symlink: path \"%s\" is not exist\n", target);
      return 0;
    }
    for(j = 0; j <= i; ++j) {
      if(ip->inum == inums[j]) {
        printf("open_symlink: links form a cycle\n");
        return 0;
      }
    }
    ilock(ip);
    if(ip->type != T_SYMLINK) {
      return ip;
    }
  }

  iunlockput(ip);
  printf("open_symlink: the depth of links reaches the limit\n");
  return 0;
```

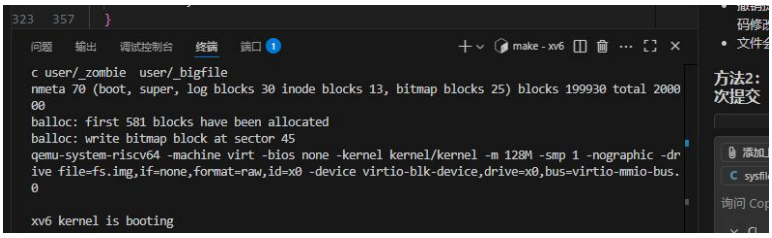
(7) 测试

```
Symlinktest
make grade
```





3）实验中遇到的问题和解决方法



问题：系统卡在了“xv6 kernel is booting”这一步。

分析原因：

sys_open 的逻辑死锁，代码修改不完善，导致了任何打开不存在文件的操作都会失败。当 sys_open 逻辑错误时：无法打开任何不存在的文件，无法创建新文件（因为逻辑永远走不到创建分支），系统进程被阻塞，等待文件操作完成，最终导致整个系统挂起。

Follow_syslink 的内存管理问题，字符串未终止 target 数组可能没有 \0 结尾；错误处理不完整，某些错误路径可能导致资源泄漏，潜在的无限循环：如果符号链接形成环，可能导致系统资源耗尽。

解决方法：

Sys_open 修复代码，正确处理找到文件和没有找到文件的情况。Follow_symlink 内存管

理修复，添加字符串终止符，确保字符串正确终止，在可能存在的资源泄露方面，需要正确的释放资源。

```
== Test running bigfile ==
$ make qemu-gdb
Timeout! running bigfile: FAIL (184.3s)
    ...
        xv6 kernel is booting

        init: starting sh
        $ bigfile
        ........................................................................
........................................................................
........................................................................
 (make)
    MISSING '^wrote 65803 blocks$'
    QEMU output saved to xv6.out.bigfile
== Test running symlinktest ==
$ make qemu-gdb
(1.3s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
Timeout! usertests: FAIL (360.3s)
    ...
        test createtest: OK
        test openiput: OK
        test exitiput: OK
        test iput: OK
        test mem: qemu-system-riscv64: terminating on signal 15 from pid 2149 (make)
    MISSING '^ALL TESTS PASSED$'
    QEMU output saved to xv6.out.usertests
== Test time ==
time: OK
Score: 41/100
make: *** [Makefile:336: grade] Error 1
```

错误：timeout，原因是电脑开了节能模式，性能较低，运行的慢。

解决方法：关闭节能模式。

4）实验心得

通过实现符号链接这个实验，我深刻理解了文件系统中不同类型链接的本质区别和设计哲学。符号链接与硬链接的最大区别在于，硬链接直接指向 inode，而符号链接存储的是目标文件的路径字符串，这种间接性带来了更大的灵活性，允许跨文件系统链接，但也引入了更多的复杂性。当我在实现过程中需要处理符号链接的循环检测、深度限制、以及各种错误情况时，我意识到这种看似简单的功能背后隐藏着大量的边界条件和异常处理逻辑。

实验中最具挑战性的部分是实现 follow_symlink 函数，这个函数需要递归地解析符号链接直到找到真正的目标文件。在这个过程中，我深刻体会到了系统编程中错误处理的复杂性。每一层的符号链接解析都可能失败，可能是因为目标文件不存在，可能是因为权限不足，也可能是因为形成了循环引用。为了防止无限递归和资源耗尽，我必须实现循环检测机制和深度限制，这让我认识到在系统设计中防御性编程的重要性。

在调试过程中遇到的系统启动卡死问题让我深刻理解了操作系统内核开发的脆弱性。一个看似微小的逻辑错误就可能导致整个系统无法启动，这是因为文件系统是操作系统的基础设施，任何核心功能的异常都会影响到系统的正常运行。当 sys_open 函数的逻辑出现问题时，系统甚至无法创建和打开基本的文件，导致启动过程中的各种初始化操作失败。这次经历让我明白，在内核开发中，每一行代码都必须经过仔细考虑，任何疏忽都可能带来灾难性的后果。

整个实验让我对文件系统的抽象层次有了更深的认识。从底层的 inode 管理到上层的路径解析，每一层都有其特定的职责和约束。符号链接作为文件系统抽象的一部分，它通过增加一层间接性来获得更大的灵活性。这种思想在计算机科学的各个领域都有广泛应用，让我对系统设计的本质有了更深的理解。

Lab: mmap

1. JOP

1）实验目标

实现 mmap() 和 munmap() 系统调用，允许程序将文件映射到内存中。

2）实验步骤

添加系统调用声明

(1) 在 makefile 中添加 mmaptest

```
$U/_mmaptest\
```

(2) 修改 kernel/syscall.h

```
// 在文件末尾添加
#define SYS_mmap   22
#define SYS_munmap 23
```

```
#define SYS_mmap   22
#define SYS_munmap 23
```

(3) 修改 kernel/syscall.c

```
// 添加函数声明
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
// 在 syscalls 数组中添加
static uint64 (*syscalls[])(void) = {
// ... 其他系统调用 ...
[SYS_mmap]    sys_mmap,
[SYS_munmap]  sys_munmap,
};
```

```
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
```

```
[SYS_mmap]    sys_mmap,
[SYS_munmap]  sys_munmap,
```

(4) 修改 user/usys.pl

```
# 在文件末尾添加
entry("mmap");
entry("munmap");
```

```
entry("mmap");
entry("munmap");
```

(5) 修改 user/user.h

```
// 在文件末尾添加
void *mmap(void*, int, int, int, int, int);
int munmap(void*, int);
```

```
void *mmap(void *,int,int,int,int,int);
int munmap(void *,int);
```

第二步：定义 VMA 结构体

(6) 修改 kernel/proc.h

```
// 添加 VMA 结构体定义
#define NVMA 16
struct vm_area {
  uint64 addr;    // mmap address
  int len;        // mmap memory length
  int prot;       // permission
  int flags;      // the mmap flags
  int offset;     // the file offset
  struct file* f; // pointer to the mapped file
};
struct proc {
  // ... 其他字段 ...
  struct vma_area vmas[NVMA];  // 虚拟内存区域数组
};
```

```
#define NVMA 16    struct vm_area vma[NVMA];
```

```
struct vm_area {
  uint64 addr;    /
  int len;        /
  int prot;       /
  int flags;      /
  int offset;     /
  struct file* f; /
};
```

第三步：添加 mmap 相关常量

(7) 在 Memlayout.h 中定义 mmap 的最小地址

```
// the minimum address the mmap can ues - lab 10
#define MMAPMINADDR (TRAPFRAME - 10 * PGSIZE)
```

```
// the minimum address the mmap can ues - lab 10
#define MMAPMINADDR (TRAPFRAME - 10 * PGSIZE)
```

(8) 定义权限和标志位：

在 kernel/fcntl.h 中，需要 mmap 系统调用实现 MAP_ANONYMOUS ,MAP_FIXED 标志位。

```
#define PROT_NONE       0x0
#define PROT_READ       0x1
#define PROT_WRITE      0x2
#define PROT_EXEC       0x4

#define MAP_SHARED      0x01
#define MAP_PRIVATE     0x02
#define MAP_ANONYMOUS   0x04
#define MAP_FIXED       0x08
```

第四步：实现核心函数

(9) 修改 kernel/sysfile.c

实现 sys_mmap 和 sys_munmao 函数 并适当修改相关函数；

```
//补充 头文件和定义
#include "memlayout.h"
#define max(a, b) ((a) > (b) ? (a) : (b))
#define min(a, b) ((a) < (b) ? (a) : (b))
 // 用于将文件映射到进程的地址空间
uint64
sys_mmap(void) {
  uint64 addr;
  int len, prot, flags, offset;
  struct file *f;
  struct vm_area *vma = 0;
  struct proc *p = myproc();
  int i;
//参数解析和验证
  if (argaddr(0, &addr) < 0||argint(1, &len) < 0 || argint(2, &prot) < 0
|| argint(3, &flags) < 0|| argfd(4, 0, &f) < 0 || argint(5, &offset) < 0) {
    return -1;
  }
//验证映射标志
  if (flags != MAP_SHARED && flags != MAP_PRIVATE) {
    return -1;
  }
//共享映射且需要写权限
  if (flags == MAP_SHARED && f->writable == 0 && (prot & PROT_WRITE)) {
    return -1;
  }
  //验证长度和偏移
  if (len < 0 || offset < 0 || offset % PGSIZE) {
    return -1;
  }
  //分配 vma
  for (i = 0; i < NVMA; ++i) {
    if (!p->vma[i].addr) {
      vma = &p->vma[i];
```

```
        break;
      }
    }
    if (!vma) {
      return -1;
    }
    addr = MMAPMINADDR;//计算映射地址
   //找到所有的现有映射的最高地址
    for (i = 0; i < NVMA; ++i) {
      if (p->vma[i].addr) {
        // get the max address of the mapped memory
        addr = max(addr, p->vma[i].addr + p->vma[i].len);
      }
    }
    addr = PGROUNDUP(addr);
    if (addr + len > TRAPFRAME) {
      return -1;
    }
  //初始化 vma
    vma->addr = addr;
    vma->len = len;
    vma->prot = prot;
    vma->flags = flags;
    vma->offset = offset;
    vma->f = f;
    filedup(f);     // increase the file's reference count
    return addr;
  }

  uint64
  sys_munmap(void) {
    uint64 addr, va;
    int len;
    struct proc *p = myproc();
    struct vm_area *vma = 0;
    uint maxsz, n, n1;
    int i;
    if (argaddr(0, &addr) < 0 || argint(1, &len) < 0) {  return -1; }
    if (addr % PGSIZE || len < 0) {  return -1; }
    // 查找 vma
    for (i = 0; i < NVMA; ++i) {
      if (p->vma[i].addr && addr >= p->vma[i].addr
          && addr + len <= p->vma[i].addr + p->vma[i].len) {
        vma = &p->vma[i];
```

```
        break;
      }
    }
    if (!vma) {    return -1;  }
    if (len == 0) {  return 0;  }
  //处理共享映射的脏页写回
    if ((vma->flags & MAP_SHARED)) {
      // 计算单次写入的最大大小（受事务限制）
      maxsz = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
      for (va = addr; va < addr + len; va += PGSIZE) {
        if (uvmgetdirty(p->pagetable, va) == 0) {  continue;}
        // 计算需要协会的字节数
        n = min(PGSIZE, addr + len - va);
        for (i = 0; i < n; i += n1) {//分批写回
          n1 = min(maxsz, n - i);
          begin_op();
          ilock(vma->f->ip);
          if (writei(vma->f->ip, 1, va + i, va - vma->addr + vma->offset +
i, n1) != n1) {
            iunlock(vma->f->ip);
            end_op();
            return -1;
          }
          iunlock(vma->f->ip);
          end_op();
        }
      }
    }
  //解除页表映射
    uvmunmap(p->pagetable, addr, (len - 1) / PGSIZE + 1, 1);
    // 更新 vma
    if (addr == vma->addr && len == vma->len) {//完全解除映射
      vma->addr = 0;
      vma->len = 0;
      vma->offset = 0;
      vma->flags = 0;
      vma->prot = 0;
      fileclose(vma->f);
      vma->f = 0;
    } else if (addr == vma->addr) {//从头部解除映射
      vma->addr += len;
      vma->offset += len;
      vma->len -= len;
    } else if (addr + len == vma->addr + vma->len) {//从尾部
```

```
        vma->len -= len;
      } else {//从中间
        panic("unexpected munmap");
      }
      return 0;
    }
```

```c
uint64
sys_mmap(void) {
  uint64 addr;
  int len, prot, flags, offset;
  struct file *f;
  struct vm_area *vma = 0;
  struct proc *p = myproc();
  int i;

  if (argaddr(0, &addr) < 0 || argint(1, &len) < 0
      || argint(2, &prot) < 0 || argint(3, &flags) < 0
      || argfd(4, 0, &f) < 0 || argint(5, &offset) < 0) {
    return -1;
  }
  if (flags != MAP_SHARED && flags != MAP_PRIVATE) {
    return -1;
  }
  // the file must be written when flag is MAP_SHARED
  if (flags == MAP_SHARED && f->writable == 0 && (prot & PROT_WRITE)) {
    return -1;
  }
  // offset must be a multiple of the page size
  if (len < 0 || offset < 0 || offset % PGSIZE) {
    return -1;
  }

  // allocate a VMA for the mapped memory
  for (i = 0; i < NVMA; ++i) {
    if (!p->vma[i].addr) {
      vma = &p->vma[i];
      break;
    }
  }
  if (!vma) {
    return -1;
  }

  // assume that addr will always be 0, the kernel
  //choose the page-aligned address at which to create
  //the mapping
  addr = MMAPMINADDR;
  for (i = 0; i < NVMA; ++i) {
    if (p->vma[i].addr) {
      // get the max address of the mapped memory
      addr = max(addr, p->vma[i].addr + p->vma[i].len);
    }
  }
  addr = PGROUNDUP(addr);
  if (addr + len > TRAPFRAME) {
    return -1;
  }
  vma->addr = addr;
  vma->len = len;
  vma->prot = prot;
  vma->flags = flags;
  vma->offset = offset;
  vma->f = f;
  filedup(f);     // increase the file's reference count

  return addr;
}
```

```c
// lab 10 added
uint64
sys_munmap(void) {
  uint64 addr, va;
  int len;
  struct proc *p = myproc();
  struct vm_area *vma = 0;
  uint maxsz, n, n1;
  int i;

  if (argaddr(0, &addr) < 0 || argint(1, &len) < 0) {
    return -1;
  }
  if (addr % PGSIZE || len < 0) {
    return -1;
  }

  // find the VMA
  for (i = 0; i < NVMA; ++i) {
    if (p->vma[i].addr && addr >= p->vma[i].addr
        && addr + len <= p->vma[i].addr + p->vma[i].len) {
      vma = &p->vma[i];
      break;
    }
  }
  if (!vma) {
    return -1;
  }

  if (len == 0) {
    return 0;
  }

  if ((vma->flags & MAP_SHARED)) {
    // the max size once can write to the disk
    maxsz = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
    for (va = addr; va < addr + len; va += PGSIZE) {
      if (uvmgetdirty(p->pagetable, va) == 0) {
        continue;
      }
      // only write the dirty page back to the mapped file
      n = min(PGSIZE, addr + len - va);
      for (i = 0; i < n; i += n1) {
        n1 = min(maxsz, n - i);
        begin_op();
        ilock(vma->f->ip);
        if (writei(vma->f->ip, 1, va + i, va - vma->addr + vma->offset + i, n1) != n1) {
          iunlock(vma->f->ip);
          end_op();
          return -1;
        }
        iunlock(vma->f->ip);
        end_op();
      }
    }
  }
```

```c
  }
  uvmunmap(p->pagetable, addr, (len - 1) / PGSIZE + 1, 1);
  // update the vma
  if (addr == vma->addr && len == vma->len) {
    vma->addr = 0;
    vma->len = 0;
    vma->offset = 0;
    vma->flags = 0;
    vma->prot = 0;
    fileclose(vma->f);
    vma->f = 0;
  } else if (addr == vma->addr) {
    vma->addr += len;
    vma->offset += len;
    vma->len -= len;
  } else if (addr + len == vma->addr + vma->len) {
    vma->len -= len;
  } else {
    panic("unexpected munmap");
  }
  return 0;
}
```

第五步：实现页面错误处理

(10) 修改 kernel/trap.c

```
  } else if (r_scause() == 12 || r_scause() == 13 || r_scause() == 15) { // mmap 页面错误
      char *pa;
      uint64 va = PGROUNDDOWN(r_stval());
      struct vm_area *vma = 0;
```

```
    int flags = PTE_U;
    int i;
    // 查找对应的 VMA
    for (i = 0; i < NVMA; ++i) {
      // 类似 Linux mmap，允许修改映射页面末尾的剩余字节
      if (p->vma[i].addr && va >= p->vma[i].addr
          && va < p->vma[i].addr + p->vma[i].len) {
        vma = &p->vma[i];
        break;
      }
    }
    if (!vma) {
      goto err;
    }
    // 为映射页面的 PTE 设置写标志和脏标志
    if (r_scause() == 15 && (vma->prot & PROT_WRITE)
        && walkaddr(p->pagetable, va)) {
      if (uvmsetdirtywrite(p->pagetable, va)) {
        goto err;
      }
    } else {
      if ((pa = kalloc()) == 0) {
        goto err;
      }
      memset(pa, 0, PGSIZE);
      ilock(vma->f->ip);
      if (readi(vma->f->ip, 0, (uint64) pa, va - vma->addr +
vma->offset, PGSIZE) < 0) {
        iunlock(vma->f->ip);
        goto err;
      }
      iunlock(vma->f->ip);
      if ((vma->prot & PROT_READ)) {
        flags |= PTE_R;
      }
      // 仅在存储页面错误且映射页面可写时
      // 设置 PTE 写标志和脏标志，否则不设置这两个标志
      if (r_scause() == 15 && (vma->prot & PROT_WRITE)) {
        flags |= PTE_W | PTE_D;
      }
      if ((vma->prot & PROT_EXEC)) {
        flags |= PTE_X;
      }
      if (mappages(p->pagetable, va, PGSIZE, (uint64) pa,
```

```
flags) != 0) {
        kfree(pa);
        goto err;
      }
    }
```

```
} else if (r_scause() == 12 || r_scause() == 13 || r_scause() == 15) { // mmap page fault - lab10
  char *pa;
  uint64 va = PGROUNDDOWN(r_stval());
  struct vm_area *vma = 0;
  int flags = PTE_U;
  int i;
  // find the VMA
  for (i = 0; i < NVMA; ++i) {
    // like the Linux mmap, it can modify the remaining bytes in
    //the end of mapped page
    if (p->vma[i].addr && va >= p->vma[i].addr
        && va < p->vma[i].addr + p->vma[i].len) {
      vma = &p->vma[i];
      break;
    }
  }
  if (!vma) {
    goto err;
  }
  // set write flag and dirty flag to the mapped page's PTE
  if (r_scause() == 15 && (vma->prot & PROT_WRITE)
      && walkaddr(p->pagetable, va)) {
    if (uvmsetdirtywrite(p->pagetable, va)) {
      goto err;
    }
  } else {
    if ((pa = kalloc()) == 0) {
      goto err;
    }
    memset(pa, 0, PGSIZE);
    ilock(vma->f->ip);
    if (readi(vma->f->ip, 0, (uint64) pa, va - vma->addr + vma->offset, PGSIZE) < 0) {
      iunlock(vma->f->ip);
      goto err;
    }
    iunlock(vma->f->ip);
    if ((vma->prot & PROT_READ)) {
      flags |= PTE_R;
    }
    // only store page fault and the mapped page can be written
    //set the PTE write flag and dirty flag otherwise don't set
    //these two flag until next store page falut
    if (r_scause() == 15 && (vma->prot & PROT_WRITE)) {
      flags |= PTE_W | PTE_D;
    }
    if ((vma->prot & PROT_EXEC)) {
      flags |= PTE_X;
    }
    if (mappages(p->pagetable, va, PGSIZE, (uint64) pa, flags) != 0) {
      kfree(pa);
      goto err;
    }
  }
```

(11) Vm.c 设置脏页标志位

```
 #define PTE_D (1L << 7) // 脏标志位

// 获取虚拟地址对应 PTE 的脏标志位
int uvmgetdirty(pagetable_t pagetable, uint64 va) {
  pte_t *pte = walk(pagetable, va, 0);
  if(pte == 0) { return 0;}
  return (*pte & PTE_D);
}

// 设置虚拟地址对应 PTE 的脏标志位和写标志位
int uvmsetdirtywrite(pagetable_t pagetable, uint64 va) {
  pte_t *pte = walk(pagetable, va, 0);
  if(pte == 0) { return -1;}
  *pte |= PTE_D | PTE_W;
  return 0;
}
```

```c
// get the dirty flag of the va's PTE - lab10
int uvmgetdirty(pagetable_t pagetable, uint64 va) {
  pte_t *pte = walk(pagetable, va, 0);
  if(pte == 0) {
    return 0;
  }
  return (*pte & PTE_D);
}

// set the dirty flag and write flag of the va's PTE - lab10
int uvmsetdirtywrite(pagetable_t pagetable, uint64 va) {
  pte_t *pte = walk(pagetable, va, 0);
  if(pte == 0) {
    return -1;
  }
  *pte |= PTE_D | PTE_W;
  return 0;
}
```

(12) Proc.c 修改 exit 和 fork 的系统调用

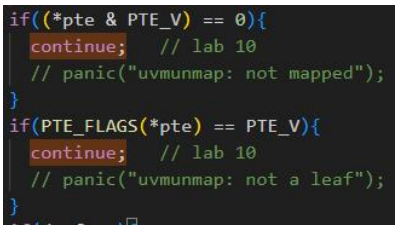
```c
/* Fork() 中的 VMA 复制处理*/
// 遍历父进程的所有 VMA 槽位
for (i = 0; i < NVMA; ++i) {
    if (p->vma[i].addr) {  // 检查 VMA 是否有效（地址非零表示已使用）
    // 将父进程的 VMA 结构完整复制给子进程
    // 包括：地址范围、权限、标志、文件偏移等所有字段
       np->vma[i] = p->vma[i];
    // 增加映射文件的引用计数 防止父进程退出时文件被过早关闭，确保子进程仍能访问
        filedup(np->vma[i].f);
    }
}
/* Exit() 中的 VMA 清理处理*/
exit(int status){
    struct proc *p = myproc();
    int i;
    struct vm_area* vma;

    // 计算单次文件写入的最大字节数
    // 基于日志系统的块数限制，避免事务过大
    uint maxsz = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
    uint64 va;      // 当前处理的虚拟地址
    uint n, n1, r;  // 写入相关的大小和偏移变量

    if(p == initproc)  panic("init exiting");

    // === VMA 清理：遍历进程的所有内存映射区域 ===
    for (i = 0; i < NVMA; ++i) {
        // 跳过未使用的 VMA 槽位
        if (p->vma[i].addr == 0) { continue; }
```

```
        vma = &p->vma[i];  // 获取当前 VMA 的指针
        // === 共享映射的脏页写回处理 ===
        if ((vma->flags & MAP_SHARED)) {
        // 遍历映射区域的每一页
        for (va = vma->addr; va < vma->addr + vma->len; va +=
PGSIZE) {
         // 检查当前页是否为脏页（是否被写入过）
           if (uvmgetdirty(p->pagetable, va) == 0) {
           continue;  // 未被修改的页面跳过，无需写回}
           // 计算当前页需要写回的字节数
           // 最后一页可能不足 PGSIZE
           n = min(PGSIZE, vma->addr + vma->len - va);
           // === 分块写入循环：处理大于 maxsz 的页面 ===
           for (r = 0; r < n; r += n1) {
           // 计算本次写入的字节数，不超过 maxsz 限制
             n1 = min(maxsz, n - r);
            // 注意：原代码中的 n-i 应该是 n-r
             // 开始文件系统事务
             begin_op();
             // 锁定文件的 inode，确保写入操作的原子性
             ilock(vma->f->ip);
             if (writei(vma->f->ip, 1, va + r, va - vma->addr +
vma->offset + r, n1) != n1) {
             // 写入失败时的错误处理
                iunlock(vma->f->ip);
                end_op();
                panic("exit: writei failed");}
             // 释放 inode 锁
             iunlock(vma->f->ip);
             end_op();
            }
           }
         }
    // === 解除页表映射 ===
    // 计算需要解除映射的页面数量（向上取整）， 释放物理页面并清除页表项
    uvmunmap(p->pagetable, vma->addr, (vma->len - 1) / PGSIZE +
1, 1);
```

```
      // === 清理 VMA 结构 ===
      vma->addr = 0;       // 清除映射地址
      vma->len = 0;        // 清除映射长度
      vma->offset = 0;     // 清除文件偏移
      vma->flags = 0;      // 清除标志位
      vma->offset = 0;     // 重复清除（可能是代码错误）
      // 关闭文件引用，减少引用计数
      fileclose(vma->f);
      vma->f = 0;          // 清除文件指针
  }
  // ... 继续 exit()的其他清理工作
}
```

(13) 修改 kernel/vm.c

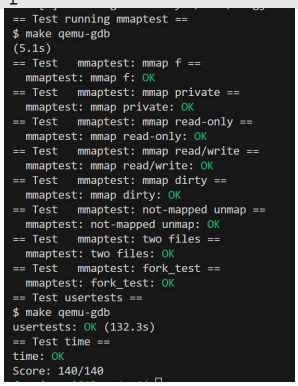
```
    修改 uvmcopy 和 uvmunmap，避免因不合法而 panic ；
   if ((*pte & PTE_V) == 0)
      // panic("uvmunmap: not mapped");
      continue;
   if ((*pte & PTE_V) == 0)
      // panic("uvmcopy: page not present");
      continue;
```



测试

(14) 编译并运行测试：

```
    $ make qemu    $ mmaptest
```



3）实验中遇到的问题和解决方法

```
fork_test starting
panic: uvmunmap: walk
```

错误：因为 fork() 没有正确处理 mmap 区域 导致的。

当进程调用 fork 创建子进程时，子进程需要继承父进程的完整地址空间，包括所有的 mmap 映射区域。如果不修改 fork 函数，子进程就无法访问父进程已经建立的文件映射，因为子进程的 VMA 数组是空的。更重要的是，每个 VMA 都关联着一个文件结构体，当子进程继承这些映射时，必须增加相应文件的引用计数，否则当父进程退出时文件可能被过早关闭，导致子进程的映射失效。通过在 fork 中复制 VMA 信息并调用 filedup 增加文件引用计数，确保了父子进程都能正常访问共享的文件映射。

修改 exit 函数同样关键，因为进程退出时必须正确清理所有的 mmap 资源。对于 MAP_SHARED 类型的映射，进程在映射区域中的任何修改都应该写回到原文件中，这就需要检查每个映射页面的脏位标志，将被修改过的页面内容同步到磁盘。如果不进行这种写回操作，进程对文件的修改就会丢失，违背了 MAP_SHARED 的语义。同时，进程退出时还需要取消所有映射的页表项，释放占用的物理内存，并调用 fileclose 减少文件的引用计数。这种清理工作确保了系统资源不会泄漏，也保证了文件系统的一致性。

4）实验心得

通过这个 mmap/munmap 实验，深刻体会到系统调用实现的复杂性远超想象。一个看似简单的内存映射功能，实际需要协调内存管理、文件系统、进程管理等多个子系统，体现了操作系统作为复杂生态系统的特点。

最有价值的收获是理解了 Lazy Loading 的设计哲学。初始时不分配物理内存，而是通过页面错误机制按需加载，这种延迟分配的思想在现代系统中随处可见。配合脏页机制只写回修改过的页面，展现了操作系统追求性能优化的精妙设计。

从调试 panic: uvmunmap: walk 错误的过程中学到，单元测试通过不代表系统集成没有问题。这个错误揭示了 fork 过程中 VMA 信息复制的重要性，以及需要将 panic 改为 continue 来提高系统健壮性。这种系统性思考和错误分析能力对工程实践很有价值。

VMA 结构体的设计是整个实现的核心，它抽象了内存映射区域的所有关键信息。通过这个数据结构，让我理解了操作系统如何管理复杂的虚拟地址空间，以及资源生命周期管理的重要性。

这个实验不仅实现了一个重要功能，更重要的是培养了系统性设计思维，让我对数据库系统、共享内存等实际应用场景的底层原理有了更深刻的理解。