

实验报告

操作系统小学期

——xv6-labs-2021

姓 名: 杨景翔
学 号: 2351576
指 导 教 师: 王冬青
学院、专业: 软件学院 软件工程

同济大学

Tongji University

目录

1.	环境准备	6
1.1.	安装 Ubuntu 24.....	6
1.2.	安装所需软件	7
1.3.	测试安装环境	7
1.4.	获取实验室的 xv6 源代码	7
2.	Lab1 Xv6 and Unix utilities	7
2.1.	Boot xv6.....	7
2.1.1.	实验目的	7
2.1.2.	实验步骤	8
2.1.3.	实验中遇到的问题与解决办法	9
2.1.4.	实验心得	10
2.2.	Sleep	10
2.2.1.	实验目的	10
2.2.2.	实验步骤	10
2.2.3.	实验中遇到的问题与解决办法	12
2.2.4.	实验心得	12
2.3.	Pingpong.....	13
2.3.1.	实验目的	13
2.3.2.	实验步骤	13
2.3.3.	实验中遇到的问题和解决办法	16
2.3.4.	实验心得	16
2.4.	Primes.....	16
2.4.1.	实验目的	16
2.4.2.	实验步骤	17
2.4.3.	实验中遇到的问题和解决办法	18
2.4.4.	实验心得	19
2.5.	Find	19
2.5.1.	实验目的	19
2.5.2.	实验步骤	19
2.5.3.	实验中遇到的问题和解决办法	24

2.5.4.	实验心得	24
2.6.	Xargs.....	24
2.6.1.	实验目的	24
2.6.2.	实验步骤	25
2.6.3.	实验中遇到的问题和解决办法	27
2.6.4.	实验心得	28
2.7.	Lab1 成绩.....	28
3.	Lab2 system calls.....	28
3.1.	System call tracing	28
3.1.1.	实验目的	28
3.1.2.	实验步骤	29
3.1.3.	实验中遇到的问题和解决办法	31
3.1.4.	实验心得	32
3.2.	Sysinfo	32
3.2.1.	实验目的	32
3.2.2.	实验步骤	32
3.2.3.	实验中遇到的问题和解决办法	35
3.2.4.	实验心得	35
3.3.	Lab2 成绩.....	36
4.	page tables.....	36
4.1.	Speed up system calls.....	36
4.1.1.	实验目的	36
4.1.2.	实验步骤	36
4.1.3.	实验中遇到的问题和解决办法	38
4.1.4.	实验心得	38
4.2.	Print a page table	38
4.2.1.	实验目的	38
4.2.2.	实验步骤	38
4.2.3.	实验中遇到的问题和解决办法	41
4.2.4.	实验心得	41
4.3.	Detecting which pages have been accessed.....	41
4.3.1.	实验目的	41

4.3.2.	实验步骤	41
4.3.3.	实验中遇到的问题和解决办法	44
4.3.4.	实验心得	44
4.4.	Lab3 成绩	44
5.	Traps	45
5.1.	RISC-V assembly	45
5.1.1.	实验目的	45
5.1.2.	实验步骤	45
5.1.3.	实验心得	48
5.2.	Backtrace	48
5.2.1.	实验目的	48
5.2.2.	实验步骤	48
5.2.3.	实验中遇到的问题和解决办法	51
5.2.4.	实验心得	51
5.3.	Alarm	51
5.3.1.	实验目的	51
5.3.2.	实验步骤	52
5.3.3.	实验中遇到的问题和解决办法	55
5.3.4.	实验心得	55
5.4.	Lab4 成绩	56
6.	Copy-on-Write Fork for xv6	56
6.1.	Implement copy-on write	56
6.1.1.	实验目的	56
6.1.2.	实验步骤	56
6.1.3.	实验中遇到的问题和解决办法	61
6.1.4.	实验心得	62
6.2.	Lab5 成绩	62
7.	Multithreading	63
7.1.	Uthread: switching between threads	63
7.1.1.	实验目的	63
7.1.2.	实验步骤	63
7.1.3.	实验中遇到的问题和解决办法	67

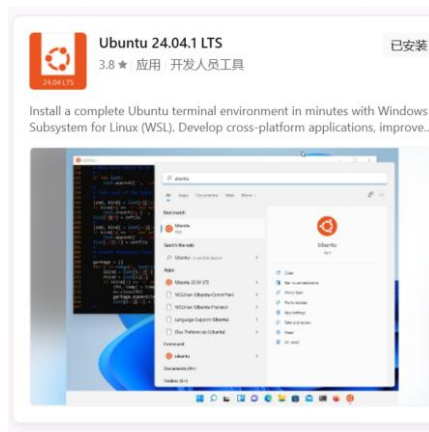
7.1.4.	实验心得	67
7.2.	Using threads.....	68
7.2.1.	实验目的	68
7.2.2.	实验步骤	68
7.2.3.	实验中遇到的问题和解决办法	71
7.2.4.	实验心得	71
7.3.	Barrier.....	71
7.3.1.	实验目的	71
7.3.2.	实验步骤	72
7.3.3.	实验中遇到的问题和解决办法	73
7.3.4.	实验心得	74
7.4.	Lab6 成绩.....	74
8.	Networking.....	74
8.1.	Your Job	75
8.1.1.	实验目的	75
8.1.2.	实验步骤	75
8.1.3.	实验中遇到的问题和解决办法	80
8.1.4.	实验心得	81
8.2.	Lab7 成绩.....	81
9.	Locks.....	81
9.1.	Memory allocator	81
9.1.1.	实验目的	81
9.1.2.	实验步骤	82
9.1.3.	实验中遇到的问题和解决办法	86
9.1.4.	实验心得	87
9.2.	Buffer cache.....	87
9.2.1.	实验目的	87
9.2.2.	实验步骤	88
9.2.3.	实验中遇到的问题和解决办法	93
9.2.4.	实验心得	93
9.3.	Lab8 成绩.....	94
10.	file system	94

10.1.	Large files.....	94
10.1.1.	实验目的	94
10.1.2.	实验步骤	95
10.1.3.	实验中遇到的问题和解决办法	102
10.1.4.	实验心得	102
10.2.	Symbolic links	102
10.2.1.	实验目的	102
10.2.2.	实验步骤	103
10.2.3.	实验中遇到的问题和解决办法	107
10.2.4.	实验心得	107
10.3.	Lab9 成绩.....	108
11.	mmap	108
11.1.2.	实验步骤	108
11.1.3.	实验中遇到的问题和解决办法	109
11.1.4.	实验心得	109
11.2.	Lab10 成绩.....	109

1. 环境准备

1.1. 安装 Ubuntu 24

本项目使用 WSL2（Windows 子系统 for Linux 2）进行安装。确保已启用 Windows 子系统 for Linux 后，从 Microsoft Store 安装 Ubuntu 24.04。



1.2. 安装所需软件

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

1.3. 测试安装环境

检查 QEMU 版本: `qemu-system-riscv64 --version`

检查 RISC-V GCC 版本（至少安装一个）：

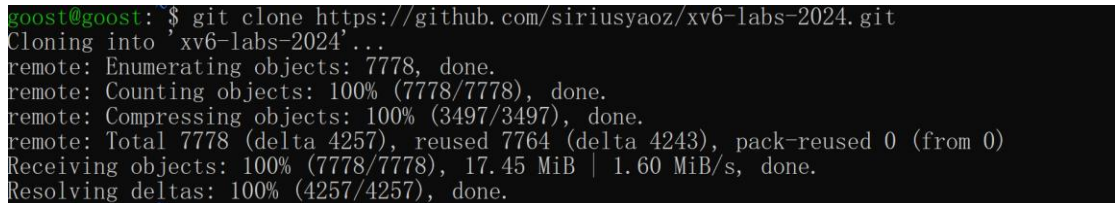
`riscv64-linux-gnu-gcc --version` # Debian 版

`riscv64-unknown-elf-gcc --version` # 裸机版

`riscv64-unknown-linux-gnu-gcc --version` # Linux 版

1.4. 获取实验室的 xv6 源代码

```
git clone git://g.csail.mit.edu/xv6-labs-2021
```



```
goost@goost:~$ git clone https://github.com/siriusyaoz/xv6-labs-2024.git  
Cloning into 'xv6-labs-2024'...  
remote: Enumerating objects: 7778, done.  
remote: Counting objects: 100% (7778/7778), done.  
remote: Compressing objects: 100% (3497/3497), done.  
remote: Total 7778 (delta 4257), reused 7764 (delta 4243), pack-reused 0 (from 0)  
Receiving objects: 100% (7778/7778), 17.45 MiB | 1.60 MiB/s, done.  
Resolving deltas: 100% (4257/4257), done.
```

1.5. Github 仓库地址

<https://github.com/oolong050309/Xv6-Labs-OS-2025.git>

2. Lab1 Xv6 and Unix utilities

本实验将熟悉 xv6 操作系统及其系统调用。

2.1. Boot xv6

2.1.1. 实验目的

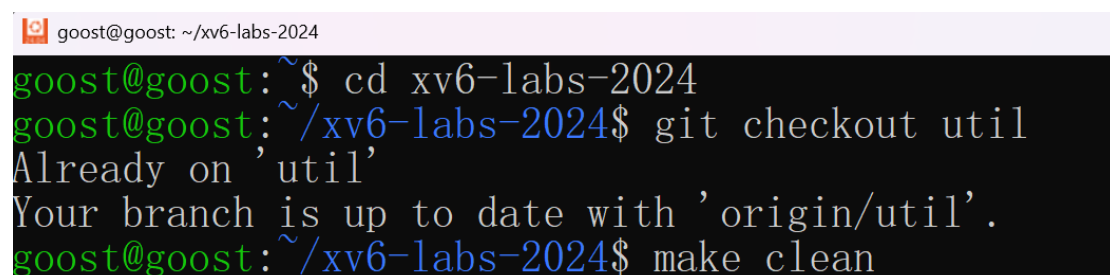
熟悉如何获取 xv6 实验代码并在本地构建运行一个最小 UNIX-like 内核镜像。理解启动过程中的关键输出（内核启动信息、init 启动 shell 等），并能用基本命令检验文件系统和进程情况。掌握在开发环境中常用的调试与故障定位方法（编译失败、缺少工具链、QEMU 配置等）。

2.1.2. 实验步骤

A. 切到实验要求的分支并清理

git checkout util

make clean

A terminal window with a light purple title bar showing the user 'goost' at host 'goost' in the directory '~/xv6-labs-2024'. The terminal output shows the user running 'cd xv6-labs-2024', then 'git checkout util', which returns 'Already on \'util\'' and 'Your branch is up to date with \'origin/util\''. Finally, the user runs 'make clean'.

B. 构建

在仓库根目录执行构建命令：

make qemu

A terminal window showing the command 'make qemu' being executed in the directory '~/xv6-labs-2024'.

观察编译与链接阶段的输出：编译内核、生成用户程序、创建文件系统镜像（fs.img）等。等待 qemu-system-riscv64 启动并看到内核自检信息（如 xv6 kernel is booting）。

C. 在 xv6 Shell 中验证

出现 init: starting sh 后，在 shell 提示符下运行：ls


```
goost@goost: ~/xv6-labs-2024
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2452
xargstest.sh 2 3 99
cat        2 4 35512
echo       2 5 34360
forktest   2 6 16232
grep        2 7 38952
init        2 8 34816
kill        2 9 34288
ln          2 10 34096
ls          2 11 37624
mkdir       2 12 34344
rm          2 13 34336
sh          2 14 57080
stressfs    2 15 35224
usertests   2 16 181128
grind        2 17 50792
wc          2 18 36496
zombie      2 19 33688
console     3 20 0
$

选择 goost@goost: ~/xv6-labs-2024
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$
```

使用 Ctrl-p（内核快捷键）查看当前进程信息，确认有 init 和 sh 两个进程正在运行。

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

D. 退出

退出 QEMU：按 Ctrl-a 然后 x 以返回宿主机终端。

```
goost@goost: ~/xv6-labs-2024
$ xQEMU: Terminated
agoost@goost:~/xv6-labs-2024$
```

2.1.3. 实验中遇到的问题与解决办法

- A. 找不到或无法运行 riscv64-unknown-elf-gcc 等交叉编译工具链。make 报错提示命令不存在或链接失败。解决：安装 RISC-V 工具链（可以使用发行版包、源码编译或下载预编译二进制）。把工具链的 bin 目录加入 PATH。

- B. 没有权限在 Windows 系统中直接修改 WSL2 中的文件。解决：输入 `sudo chown -R "$(id -u -n):$(id -g -n)"` . 把整个仓库的所有权改为当前用户。

2.1.4. 实验心得

通过亲手构建并运行 xv6，我对操作系统的启动流程、用户空间与内核之间的界面（系统调用）以及最小文件系统的构成有了更直观的理解。实践中看到的编译、链接与镜像制作流程补齐了课堂上较抽象的概念。提前准备好 RISC-V 交叉编译链与 QEMU 环境，可以大大节省调试时间；对 make 输出的每一步保持关注，遇错先从最明显的错误信息入手定位。

2.2. Sleep

2.2.1. 实验目的

实现一个用户态的 sleep 程序（放在 user/sleep.c），能够让 xv6 进程根据用户输入的 tick 数暂停执行，从而熟悉用户态到内核的系统调用接口。理解 xv6 中“tick”的概念（由定时器中断驱动），并掌握如何在用户程序中调用内核提供的系统调用（sleep）。学会在 Makefile 中注册新程序、构建镜像并在 xv6 shell 中运行与测试用户程序。

2.2.2. 实验步骤

A. 新建源文件

在仓库的 user/ 目录下创建文件 sleep.c

B. 实现程序主体

在 main 中解析命令行参数（检查 argc），如果参数缺失或非法，打印使用提示并返回非 0 值。使用 atoi() 将字符串参数转换为整数（ticks）。对负数做保护（例如当成 0）。调用用户态可用的 sleep(ticks) 系统调用。调用结束后调用 exit(0) 退出。

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

```

#include "kernel/stat.h"

#include "user/user.h"

int

main(int argc, char **argv)

{

    if(argc != 2){

        fprintf(2, "usage: sleep seconds\n");

        exit(1);

    }

    int time = atoi(argv[1]); //atoi() turn string into int

    if(time < 0){

        printf("Invaild number of ticks.");

        exit(1);

    }

    sleep(time);

    exit(0);

}

```

C. 修改 Makefile

在 Makefile 的 UPROGS 列表中添加你的程序（通常以 `_sleep` 的形式加到列表里，项目约定可能为 `user/_sleep`）。例如在 UPROGS 中加入 `_sleep`。

D. 构建并运行

在项目根目录执行：`make qemu`。运行`$ sleep 10`

```
goost@goost: ~/xv6-labs-2021-1
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ sleep 10
$ _
```

E. 测试

输入`./grade-lab-util sleep`进行测试。

```
goost@goost: ~/xv6-labs-2024
== Test sleep, no arguments == sleep, no arguments: OK (2.3s)
== Test sleep, returns == sleep, returns: OK (0.4s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.1s)
```

2.2.3. 实验中遇到的问题与解决办法

- A. 参数处理错误无参数时程序崩溃或行为不确定；传入非数字字符串时程序立刻返回（`atoi` 返回 0）。解决：在 `main` 中先判断 `argc`，如果不等于 2 则打印 `usage: sleep ticks` 并返回；对 `atoi` 的返回值做合理检查（例如提示“参数应为非负整数”）。如果需要更严格的数字验证，可手工检查每个字符是否为数字。
- B. 以秒为单位误解（把 `ticks` 当成秒）。解决：在实验报告中明确 `ticks` 与真实时间的换算关系（若需要以秒为单位使用，程序可提供额外参数或换算：`sleep_seconds = ticks * tick_duration_seconds`，或者增加 `-s` 选项把数字乘以 `ticks_per_second`）。

2.2.4. 实验心得

通过实现 `sleep`，更清楚用户态调用内核服务的流程（从 `user` 目录的 C 程序经 `usys.S` 调用内核实现 `sys_sleep`），并对 `xv6` 中时间的基本概念（`tick`、中断）有更直观的认知。遇到由参数或编译/链接引起的问题时，先在宿主机用 `make` 输出定位错误；在 `xv6` 中可用 `printf` 做简单验证，或查看 `qemu` 的串口日志。

2.3. Pingpong

2.3.1. 实验目的

实现一个用户态程序 pingpong，通过 xv6 的 pipe 与 fork 系统调用在父子进程间传递一个字节并打印相应消息。目标是加深对进程间通信（IPC）、文件描述符管理和进程同步（父子通信、等待子进程退出）的理解。同时练习将新程序加入 Makefile、构建并在 xv6 shell 中运行与调试用户程序的完整流程。

2.3.2. 实验步骤

A. 新建源文件

在项目 user/ 目录下创建 pingpong.c

B. 实现程序主体

创建两条管道 p1（父 → 子）和 p2（子 → 父）。父进程先向 p1 写入一个字节（表示 ping），子进程从 p1 读完后打印 <pid>: received ping，再把字节写到 p2，子进程退出；父进程从 p2 读到字节后打印 <pid>: received pong 并退出。父子双方都要关闭自己不需要的管道端口（避免死锁）；读写时检查返回值；用 getpid() 打印进程 id；父进程用 wait() 等待子进程退出。

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

```
#include "kernel/stat.h"
```

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

```
//pingpong.c
```

```
#define READED 0
```

```
#define WRITEEND 1
```

```
int
```

```
main(int argc, char **argv)

{

    int pid;

    int p1[2],p2[2];

    char buf[1];

    pipe(p1);//parent -> child

    pipe(p2);//child -> parent

    pid = fork();

    if(pid<0) { exit(1); }

    else if(pid==0)

    { //child

        close(p1[WRITEEND]);

        close(p2[READED]);

        read(p1[READED],buf,1);//read a btye from p1

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

        write(p2[WRITEEND]," ",1);//wrire a byte to p2

        close(p2[WRITEEND]);

        close(p1[READED]);

        exit(0);

    }
```

```

else{

//parent

    close(p1[READED]);

    close(p2[WRITEEND]);

    write(p1[WRITEEND], "x", 1);

    read(p2[READED],buf,1);

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

    close(p2[READED]);

    close(p1[WRITEEND]);

}

exit(0);

}

```

C. 修改 Makefile

在 Makefile 的 UPROGS 列表里加入 `_pingpong`。保存并返回项目根目录。

D. 构建并运行

在宿主机运行：`make qemu`。输入 `pingpong`，观察到以下运行结果：



```

选择 goost@goost: ~/xv6-labs-2024
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$

```

E. 测试

输入 `./grade-lab-util pingpong` 进行测试。

```
goost@goost: ~/xv6-labs-2024
goost@goost: ~/xv6-labs-2024$ python3 ./grade-lab-util pingpong
\make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (1.2s)
(Old xv6.out.pingpong failure log removed)
```

2.3.3. 实验中遇到的问题和解决办法

程序运行后死等（阻塞），没有输出。原因是没有正确关闭不需要的一端管道，导致 `read` 一直等待写端保持打开（因为写端未全部关闭时 `read` 不返回 0）。解决办法：严格按照父子职责关闭端口：父：关闭 `p1[0]`（读端）和 `p2[1]`（写端），只保留 `p1[1]`（写）和 `p2[0]`（读）。子：关闭 `p1[1]`（写端）和 `p2[0]`（读端），只保留 `p1[0]`（读）和 `p2[1]`（写）。这样 `read` 在读不到数据且写端被关闭时能返回 0，避免死锁。

2.3.4. 实验心得

通过实现 `pingpong`，我加深了对 `pipe` 的工作机制、文件描述符语义（读端/写端的打开/关闭）和 `fork` 后父子进程资源继承的理解。实际写代码时，许多理论细节（例如关闭不必要的端口以避免阻塞）在真实运行中立即显现，这是最有价值的部分。我对进程间同步有了更直观的感受：即使只传一个字节，也要考虑什么时候读、什么时候写、何时关闭文件描述符以及如何回收子进程，这些都关系到程序能否健壮运行。

2.4. Primes

2.4.1. 实验目的

实现基于管道（`pipe`）与进程（`fork`）的并发素数筛（Sieve of Eratosthenes 风格），练习用进程和管道搭建数据流式计算模型。熟悉在 `xv6` 中动态创建多级进程管道、文件描述符管理、进程间同步（等待/回收）以及如何避免耗尽系统资源（文件描述符/进程数）。理解如何把整数以二进制形式通过管道传输以提高效率，并学会在有限资源的嵌入式教学内核上调试并发程序。

2.4.2. 实验步骤

A. 新建源文件

在 `user/` 目录下创建源文件：`user/primes.c`

B. 实现程序主体

主进程生成 `2..280` 的整数流写入第一个管道。每遇到一个新的素数 `p`，就创建一个新的进程（筛子进程），该进程从左侧管道读入数字，跳过能被 `p` 整除的数，把其他数写入其右侧管道；每个筛子进程负责过滤它自己的素数倍数并打印 `prime p`。这样逐级创建筛子直至数流结束或达到 `280`。实现要点：

按需创建进程：当筛子进程读到第一个能被它处理的数（即新的素数）后，创建下一个筛子进程并把后续数写入下游管道；避免一次性创建大量进程。

二进制传输：使用 `int` (32-bit) 二进制写入/读取管道，避免字符串格式化带来的额外开销。

关闭不需要的文件描述符：每个进程在 `fork` 后立即关闭不需要的管道端（读端或写端），防止管道写端未全部关闭导致下游 `read` 永远阻塞。

读到 EOF 的处理：当 `read` 返回 `0`（写端关闭）时，进程正常退出并 `wait/exit`。

资源限制防护：因为 `xv6` 的进程和文件描述符有限，主进程只生成到 `280`，且确保及时回收子进程（`wait`），避免用尽内核资源。

C. 修改 Makefile

在 `Makefile` 的 `UPROGS` 列表中加入 `_primes`。保存并返回项目根目录。

D. 构建并运行

在宿主机运行：`make qemu`。输入 `primes`，观察到以下运行结果：

```
goost@goost: ~/xv6-labs-2024
xv6 kernel is booting
hart 2 starting
hart 1 starting
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
$
```

运行 `primes` 时，主进程开始向第一个管道写入 2..280 的整数。第一个筛子进程读到 2 就把 `prime 2` 打印出来，并为之后的过滤创建下游筛子；同理接着会产生 `prime 3, 5, 7, ...`。每个 `prime N` 的出现对应了一个新筛子进程的“第一次读到 `N` 并确认 `N` 为素数”的事件。

E. 测试

输入 `./grade-lab-util pingpong` 进行测试。

```
goost@goost: ~/xv6-labs-2024
goost@goost:~/xv6-labs-2024$ python3 ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.6s)
(Old xv6.out.primes failure log removed)
```

2.4.3. 实验中遇到的问题和解决办法

程序在运行中很快耗尽文件描述符或进程，程序崩溃或 QEMU 内核报错。在某处没有关闭不需要的 `pipe` 端，或一次性创建了太多筛子进程。解决方法：确保每个进程在 `fork` 后立刻关闭自己不需要的读/写端（父与子职责明确）。延迟创建下游进程 —— 只有在读取到新的素数时才创建对应的筛子

（按需创建）。主进程限制到 280（课程要求），并在主进程结束前 wait 所有子进程以回收进程表项。

2.4.4. 实验心得

这个实验把“管道 + 进程”这样的抽象思想用实际代码很好地串联起来。实现过程中我真正体会到“每个进程都继承了管道端，但必须主动关闭不需要的端”的重要性；这是很多并发死锁/阻塞问题的根源。实现按需创建筛子进程的思路让我更理解了惰性计算与资源节约的必要性——不是越并发越好，而是要在受限资源下恰当地设计并发策略。

2.5. Find

2.5.1. 实验目的

实现一个简化版的 find，在 xv6 的目录树中查找并打印所有与指定名字匹配的文件路径。通过本实验熟悉目录读取（open/read）、stat/fstat 判别文件类型、递归遍历目录树、字符串比较（strcmp）以及路径拼接的正确做法。锻炼在受限环境下写稳健递归代码并防止资源泄露

2.5.2. 实验步骤

A. 新建源文件

在项目的 user/ 目录下创建源文件。

B. 实现程序主体

main 检查 argc，期望格式 find <path> <name>，若参数不对则打印用法并 exit(1)。使用 open(path, O_RDONLY) 打开起始目录（或直接在递归函数中以 . 为起点）。使用与 user/ls.c 相同的方法读取目录：用 read(fd, &de, sizeof(de)) 读取 struct dirent（或项目对应的目录项结构），并跳过空条目。对每个目录项，构造完整路径字符串 path/entry->name。注意缓冲区大小（避免越界），使用手工拼接并保证以 \0 结尾。如果 entry 名为 “.” 或 “..”，必须跳过以免进入无限递归。调用 stat(childpath, &st) 判断该条目是普通文件还是目录。若是目录则递归调用 find(childpath, target)；若

是普通文件并且 `strcmp(entry->name, target) == 0` 则打印 `childpath`。每次 `open` 后及时 `close(fd)`；递归返回前不要忘了关闭临时打开的 `fd`。使用 `strcmp()` 来判断名称是否匹配（不要用 `==`）。所有递归结束后在 `main` 中 `exit(0)`。

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

#include "kernel/stat.h"

#include "kernel/fs.h"

void find(char *path, char *filename)

{

    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_DIR)

    {

        fprintf(2, "find: %s is not a directory\n", path);

        close(fd);

        return;

    }

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

    {

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

        close(fd);

        return;

    }

    strcpy(buf, path);

    p = buf + strlen(buf);

    *p++ = '/';

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

    {

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

```

        continue;

    memmove(p, de.name, DIRSIZ);

    p[DIRSIZ] = 0;

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

    {

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

        continue;

    }

    if (st.type == T_DIR)

    {

        find(buf, filename);

    }

    else if (st.type == T_FILE && strcmp(de.name, filename) == 0)

    {

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

    }

}

close(fd);

}

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

```

```

{

    if (argc != 3)

    {

        fprintf(2, "usage: find path filename...\n");

        exit(1);

    }

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

    exit(0);

}

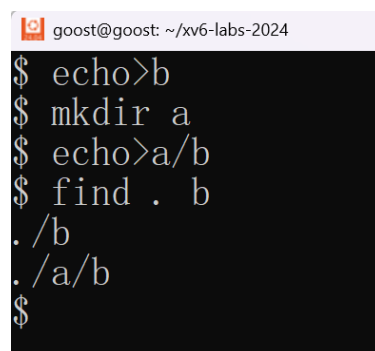
```

C. 修改 Makefile

在 Makefile 的 UPROGS 列表中加入 `_find`

D. 构建并运行

输入 `make qemu`，在 `xv6 shell` 中执行以下操作来构造测试目录结构并运行 `find`：



```

goost@goost: ~/xv6-labs-2024
$ echo>b
$ mkdir a
$ echo>a/b
$ find . b
./b
./a/b
$

```

`echo>b`：在当前目录下创建并写入一个空文件 `b`（等价于 `echo > b`，在 `xv6 shell` 中会创建一个大小为 0 或含换行的文件）。`mkdir a`：创建目录 `a`。`echo>a/b`：在 `a` 目录下创建文件 `b`（相当于 `a/b`）。`find . b`：从当前目录 `.` 开始递归查找名为 `b` 的所有文件，并把匹配的相对路径打印出来。

输出结果：./b：表示当前目录下找到一个名为 b 的文件（就是第一步创建的那个）。./a/b：表示在子目录 a 下也找到了名为 b 的文件（就是第三步创建的）。find . b 按照递归遍历把所有匹配名为 b 的路径打印出来。

E. 测试

输入./grade-lab-util find 进行测试

```
goost@goost: ~/xv6-labs-2024
goost@goost:~/xv6-labs-2024$ python3 ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.0s)
== Test find, in sub-directory == find, in sub-directory: OK (1.2s)
== Test find, recursive == find, recursive: OK (0.8s)
```

2.5.3. 实验中遇到的问题和解决办法

- A. 使用 == 比较字符串导致匹配失败，因为== 比较指针地址而不是字符串内容。解决：使用 strcmp(de.name, target) == 0。
- B. 路径缓冲区越界或路径拼接错误，没正确检查剩余缓冲区长度或在拼接时忘记加 /。解决：为路径分配足够空间（例如 char buf[512]），在拼接前用 strlen 检查剩余空间是否大于 strlen(name)+2（包含 / 与 \0），或者使用安全的拼接函数并在超长时跳过该条目并打印警告。

2.5.4. 实验心得

通过实现 find，我更熟悉了目录项结构、如何安全地构建路径以及递归遍历目录树要处理的边界条件（./...、路径长度）。这些看似小的细节在嵌入式/教学内核环境下会立刻暴露为运行时错误。实践中最容易出错的是路径拼接与文件描述符的管理（open/close）。每次递归调用都要小心资源释放，避免 fd 泄露。

2.6. Xargs

2.6.1. 实验目的

实现一个简化版的 xargs：读取标准输入的每一行，把该行作为额外参数追加到给定命令的参数列表末尾，并用 fork/exec 执行该命令。通过该实验巩

固对 fork、exec、wait、标准输入读取、argv 构造与边界处理（行长、参数数量）的理解。熟悉在 xv6 受限环境下的错误检查与父子进程资源管理（等待子进程、避免参数越界）。

2.6.2. 实验步骤

A. 新建源文件

在仓库 user/ 目录下创建 user/xargs.c 并打开编辑。

B. 实现程序主体

在 main 中解析命令行参数，原始命令为 argv[1..]（必须至少有一个命令）。从标准输入逐字符读取，累积成一行（直到遇到 \n 或 EOF）。每读到一行（去掉末尾换行），构造新的 argv2：先拷贝原始命令参数，再把该行作为最后一个参数，最后以 NULL 结尾。fork() 子进程：子进程调用 exec(argv2[0], argv2)；父进程在 fork 后 wait() 子进程结束（本实现每行一个 exec）。对 read/fork/exec 的返回值检查；对超出 MAXARG 或行过长做截断或报错处理。

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

#include "kernel/param.h"

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

{

    int argc2 = argc - 1;

    char *argv2[MAXARG];

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

    {

        argv2[i - 1] = argv[i];
```

```
}

char s[64];

int c = 0, flag = 0;

while (read(0, &s[c++], sizeof(char)))

{

    if (s[c - 1] != ' ' && (flag = s[c - 1] != '\n'))

        continue;

    s[c - 1] = '\0';

    argv2[argc2] = (char *)malloc(c);

    strcpy(argv2[argc2++], s);

    c = 0;

    if (flag)

        continue;

    argv2[argc2] = 0;

    if (fork() == 0)

    {

        exec(argv2[0], argv2);

        exit(0);

    }

    else
```

```

        wait(0);

        argc2 = argc - 1;

    }

    exit(0);

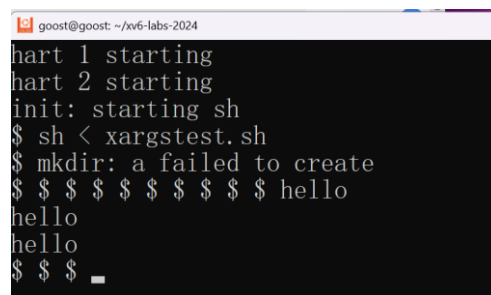
}

```

C. 修改 Makefile

在 Makefile 的 UPROGS 列表中加入 `_xargs`

D. 构建并运行



```

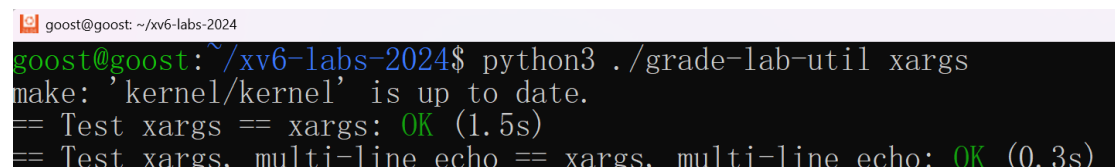
goost@goost: ~/xv6-labs-2024
hart 1 starting
hart 2 starting
init: starting sh
$ sh < xargstest.sh
$ mkdir: a failed to create
$ $ $ $ $ $ $ $ $ hello
hello
hello
$ $ $ _

```

运行后产生两个 hello 符合预期结果

E. 测试

输入 `./grade-lab-util xargs` 进行测试。



```

goost@goost: ~/xv6-labs-2024
goost@goost: ~/xv6-labs-2024$ python3 ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (1.5s)
== Test xargs, multi-line echo == xargs, multi-line echo: OK (0.3s)

```

2.6.3. 实验中遇到的问题和解决办法

`argv` 数目超过 `MAXARG` 导致 `exec` 失败或越界写入，构造 `argv2` 时超出 `MAXARG`，发生数组越界或 `exec` 参数不正确。解决：在拼装 `argv2` 前检查原始参数数量并保证追加后不超 `MAXARG-1`；若超出则提示错误并跳过或截断额外参数。

2.6.4. 实验心得

实现 xargs 加深了我对“把文本行转换为 argv 并用 exec 启动命令”这一流程的理解。最容易出错的是边界检查（行长、参数数量、argv 的 NULL 终止），所以在实现中务必对每个系统调用的返回值进行检查并在边界处给出明确处理策略。调试时在 fork 前打印将要 exec 的 argv2 非常有帮助。

2.7. Lab1 成绩

增加 time.txt 文件输入完成实验的时长。输入 ./grade-lab-util 查看实验一的得分。

```
goost@goost: ~/xv6-labs-2024
goost@goost:~/xv6-labs-2024$ python3 ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.9s)
== Test sleep, returns == sleep, returns: OK (1.1s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.9s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.6s)
== Test find, in current directory == find, in current directory: OK (0.8s)
== Test find, in sub-directory == find, in sub-directory: OK (0.9s)
== Test find, recursive == find, recursive: OK (1.3s)
== Test xargs == xargs: OK (0.9s)
== Test xargs, multi-line echo == xargs, multi-line echo: OK (0.5s)
== Test time ==
time: OK
Score: 110/110
```

3. Lab2 system calls

3.1. System call tracing

3.1.1. 实验目的

实现一个新的内核系统调用 trace(mask)，通过位掩码控制对某些系统调用的跟踪（在系统调用返回时打印进程 id、系统调用名和返回值），从而加深对 xv6 系统调用路径（用户态存根 → ecall → 内核处理 → 返回）的理解。学习如何在 xv6 内核中新增系统调用：添加用户态声明与存根、分配 syscall 号、在内核实现 sys_* 函数、并在 syscall() 分发函数处添加额外行为（打印 trace）。掌握进程结构（proc）中保存进程级状态、在 fork() 中继承父状态、以及在并发/多核环境下正确访问进程字段的方法。

3.1.2. 实验步骤

A. 新建源文件

在 `user/` 目录已存在 `user/trace.c`。

B. 实现程序主体

在 `user/user.h` 中添加原型：`int trace(int mask);`

在 `user/usys.pl` 中添加一行：`entry(trace)`。这样 `usys.pl` 生成的 `user/usys.S` 会包含 `trace` 的用户态存根。

在 `kernel/syscall.h` 中为 `SYS_trace` 分配一个新的枚举项：`#define SYS_trace <next_available_number>`

在 `kernel/sysproc.c` 中添加 `sys_trace()`，实现步骤：从用户参数获取 `mask`（用已有的 `argint()` 函数）；将 `mask` 保存到当前进程 `proc` 结构中新加的字段（例如 `int tracemask;`）。返回 0 表示成功。

```
uint64

sys_trace(void)//add for lab2

{

    int mask;

    if(argint(0, &mask)<0)

        return -1;

    myproc()->syscall_trace = mask;

    return 0;

}
```

在 kernel/proc.h 的 struct proc 中新增字段: int tracemask; // 位掩码: 哪些 syscall 要被跟踪.

在进程创建/初始化 (allocproc() / userinit() 等) 处将 tracemask 初始化为 0 (默认不跟踪)。

在 kernel/proc.c 的 fork() 中, 复制父进程的 tracemask 到子进程 (如 np->tracemask = p->tracemask;), 以便子进程继承跟踪设置。

在 kernel/syscall.c 的 syscall() (或相应的分发返回点) 处, 在系统调用返回前判断当前进程的 tracemask 是否对应该系统调用号置位。若置位, 则打印一行类似: <pid>: syscall <name> -> <ret>

其中 <name> 从一个内核内的数组 syscallnames[] 根据 syscall 号索引取得 (需在 syscall.c 或合适文件中添加该数组并按 syscall 号顺序填入名字字符串)。保证打印在设置返回值 (tf->a0) 之后或能正确读取到返回值的位置。

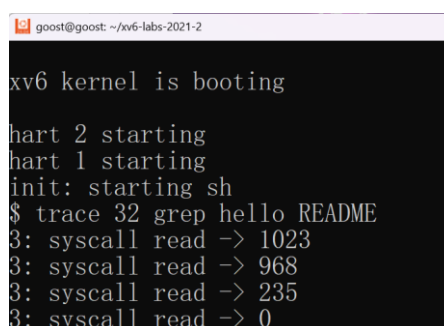
访问 & 修改 proc->tracemask 的位置 (sys_trace、fork、syscall) 应遵循 xv6 对 proc 的访问规则: 在需要时获取进程锁或在仅对当前进程且处于临界区的情况下安全直接访问

C. 修改 Makefile

在 Makefile 的 UPROGS 增加 \$U/_trace

D. 构建并运行

输入 make qemu, 并输入 trace 32 grep hello README, 运行结果如下



```
goost@goost: ~/xv6-labs-2021-2
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
```

trace 程序通过先调用内核 trace(mask) 把 tracemask 写入当前进程, 然后 exec 目标程序 (grep)。在 grep 运行过程中, 内核每次某个系统调用执行完

返回时都会检查该进程的 `tracemask`：如果对应系统调用号的位被置位，内核就打印该 `syscall` 的名字与返回值，从而得到上面的行。

```
goost@goost: ~/xv6-labs-2021-2
$ trace 2 usertests forkforkfork
usertests starting
4: syscall fork -> 5
test forkforkfork: 4: syscall fork -> 6
6: syscall fork -> 7
7: syscall fork -> 8
7: syscall fork -> 9
8: syscall fork -> 10
7: syscall fork -> 11
8: syscall fork -> 12
9: syscall fork -> 13
10: syscall fork -> 14
8: syscall fork -> 15
9: syscall fork -> 16
7: syscall fork -> 17
8: syscall fork -> 18
10: syscall fork -> 19
7: syscall fork -> 20
9: syscall fork -> 21
8: syscall fork -> 22
7: syscall fork -> 23
10: syscall fork -> 24
9: syscall fork -> 25
7: syscall fork -> 26
```

`usertests forkforkfork` 这个测试会做大量的 `fork()` 调用（递归/嵌套式地创建进程树），所以会迅速产生很多新进程。`xv6` 给新进程分配的 PID 是按顺序增长的（通常是逐一增加的整数），因此在大量 `fork` 时你会看到许多连续的、增大的返回值（父进程看到的返回值是新子进程的 PID）。

E. 测试

输入 `./grade-lab-syscall trace` 进行测试。

```
goost@goost: ~/xv6-labs-2021-2
goost@goost: ~/xv6-labs-2021-2$ python3 ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.4s)
== Test trace all grep == trace all grep: OK (1.1s)
== Test trace nothing == trace nothing: OK (0.9s)
== Test trace children == trace children: OK (10.0s)
```

3.1.3. 实验中遇到的问题和解决办法

- A. 忘记在 `kernel/syscall.h` 增加 `SYS_trace` 导致链接或分发错误，编译或链接时 `syscall` 表与名字数组索引不对，或运行时 `syscall()` 无法识别新的号。解决：在 `kernel/syscall.h` 按顺序给 `SYS_trace` 分配正确的编号，并在 `syscall.c` 的名称数组中对应位置加入 “`trace`” 字符串。确保 `syscall` 表（`syscalls[]`）与名字数组一致。

- B. 没有把 trace mask 复制到子进程 (fork)，父进程调用 trace() 后父可见但子进程没有继承跟踪，导致子进程没有跟踪输出。解决：在 fork() 成功创建子进程时设置 `np->tracemask = p->tracemask;` (p 为父进程，np 为新进程)，并在 `allocproc()` / `userinit()` 处初始化 tracemask。

3.1.4. 实验心得

通过实现 trace 我更清楚地理解了系统调用从用户态到内核态再返回的完整路径：用户端 usys 存根通过 `ecall` 进入内核，内核在 `syscall()` 中分发到对应 `sys_*` 实现，然后把返回值写回寄存器并返回用户态。把跟踪功能放在 `syscall()` 的返回点让我直观体会到“什么时候可见返回值”。在实现过程中，最容易犯的错误是“接口不同步”——也就是忘记在 `syscall.h`、`usys.pl`、`user.h`、`sysproc.c` 等多处同步添加对应条目；每处少一项都会导致编译或运行出错。把需要修改的文件列成清单并按顺序逐项修改、重构建，能极大减少反复调试时间。继承父进程设置（在 `fork()` 中复制 `tracemask`）的要求提醒我：很多“进程级属性”必须在创建时显式传递，否则行为会不可预期。

3.2. Sysinfo

3.2.1. 实验目的

在 xv6 中新增 `sysinfo(struct sysinfo *)` 系统调用，内核填充并把系统信息结构体复制回用户空间，字段包括空闲内存字节数 `freemem` 和当前非 UNUSED 进程数 `nproc`。通过实现该系统调用，理解内核如何从用户态接收指针、如何在内核收集信息（遍历 `free list` / 遍历 `proc` 表）、如何用 `copyout()` 把数据写回用户空间，以及如何把新 `syscall` 的用户存根加入构建流程。学会在内核中安全地访问共享数据（保护 `free list` 与 `proc` 表的并发性），并掌握 `syscall` 增加所需修改的全部位置（`user.h`、`usys.pl`、`kernel/syscall.h`、`kernel/sysproc.c`、`kernel/kalloc.c`、`kernel/proc.c` 等）。

3.2.2. 实验步骤

- A. 新建源文件

课程提供了 user/sysinfotest.c

B. 实现程序主体

在 user/user.h 中预声明 struct sysinfo; 并加入原型:

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

在 user/usys.pl 中添加一行: entry(sysinfo)

为 sysinfo 分配 syscall 号: 在 kernel/syscall.h 中添加 SYS_sysinfo (按现有编号顺序插入)。在 kernel/syscall.c 中的 syscalls[] (或分发表) 里添加 sys_sysinfo 的映射, 并在 syscall 名称数组中加入 "sysinfo"

在 kernel/sysproc.c 中新增实现 sys_sysinfo(void):

使用 argptr() / argint() / argaddr() 等现有辅助函数从用户参数中取到用户空间指针 addr (或 argptr(0, (char**)&addr, sizeof(struct sysinfo)))。

在内核中构造一个 struct sysinfo kinfo, 填充字段:

kinfo.freemem = kfreemem_in_bytes(); (调用在 kalloc.c 新增的函数)

kinfo.nproc = count_procs(); (调用在 proc.c 新增的函数)

使用 copyout(myproc()->pagetable, addr, (char*)&kinfo, sizeof(kinfo)) 把 kinfo 复制回用户空间; 返回 0 表示成功, 失败返回 -1。

```
uint64
```

```
sys_sysinfo(void)//add for lab2
```

```
{
```

```
    // 从用户态读入一个指针, 作为存放 sysinfo 结构的缓冲区
```

```

uint64 addr;

if(argaddr(0, &addr) < 0)

    return -1;


struct sysinfo sinfo;

sinfo.freemem = count_free_mem(); // kalloc.c

sinfo.nproc = count_process(); // proc.c

if(copyout(myproc()->pagetable, addr, (char *)&sinfo, sizeof(sinfo)) < 0)

    return -1;

return 0;

}

```

在 `kalloc.c` 添加统计空闲内存的辅助函数：安全地遍历 `kalloc` 的 `free list` 统计空闲页数并乘以 `PGSIZE` 返回字节数。

在 `proc.c` 添加统计进程数的辅助函数：遍历 `ptable`（使用 `proc` 表的锁 `ptable.lock`）统计 `state != UNUSED` 的进程数并返回。

C. 修改 Makefile

在 `Makefile` 的 `UPROGS` 中加入 `$U/_sysinfotest`

D. 构建并运行

输入 `make qemu`，输入 `sysinfotest` 运行结果如下：

```
goost@goost: ~/xv6-labs-2021-2
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$
```

user/sysinfotest 程序在用户态分配或声明了一个 struct sysinfo 的实例并调用 sysinfo(&info)。内核 sys_sysinfo() 在内核态填充 freemem 与 nproc 并通过 copyout() 将整个结构体复制回该用户缓冲区。测试程序检验 freemem 与 nproc 是否在合理范围内并输出 OK。因此能够看到 OK 表示：用户端存根存在、内核实现成功执行、copyout() 正确将数据返回给用户，且统计结果被认为有效。

E. 测试

输入 ./grade-lab-syscall sysinfo 进行测试

```
goost@goost: ~/xv6-labs-2021-2
goost@goost:~/xv6-labs-2021-2$ python3 ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (2.1s)
```

3.2.3. 实验中遇到的问题和解决办法

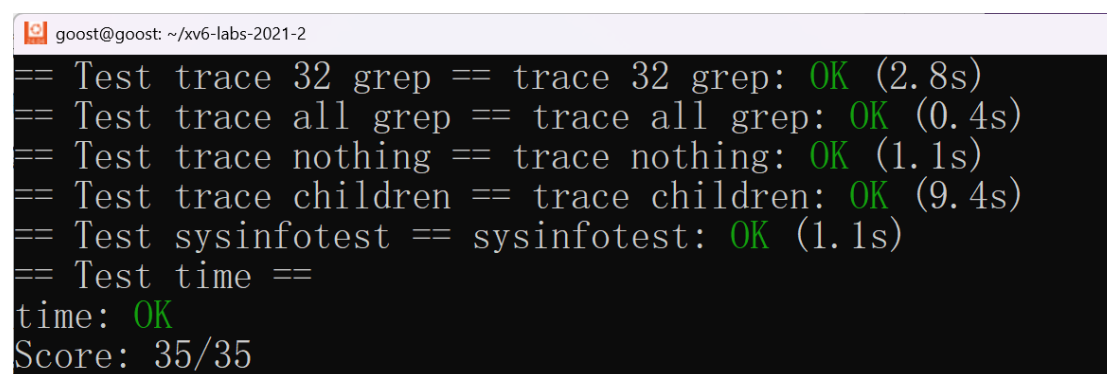
- A. 用户端编译失败 —— 找不到 sysinfo 原型或 struct sysinfo，user/sysinfotest.c 在编译时提示 implicit declaration 或未定义 struct sysinfo。解决：在 user/user.h 里预声明 struct sysinfo; 并添加 int sysinfo(struct sysinfo *); 的原型；同时在 user/usys.pl 添加 entry(sysinfo)，使 usys.pl 生成 user/usys.S 存根。
- B. copyout() 失败（copyout 返回 -1），sysinfo() 返回错误或测试失败，copyout() 报错表示目标用户地址不可写或地址不合法。解决：确保使用正确的用户地址（argaddr(0, &addr) 或 argptr() 获取），且 sysinfo 的调用方提供了已分配且大小足够的缓冲（测试程序通常保证）。在内核实现前检查 argptr() / argaddr() 的返回值并在失败时返回 -1。

3.2.4. 实验心得

通过实现 `sysinfo`，我对“如何把内核内部状态暴露给用户态”这一流程有了更清晰的认识：从在用户态声明结构体和系统调用原型，到在内核计数空闲页与进程并安全地把结果 `copyout` 回去，整个链路上每一步都要小心处理边界与并发。实践中最容易忽略的是并发保护（访问 `free list` 与 `proc table` 必须按内核的锁策略进行），以及用户/内核接口的一致性（`user.h`、`usys.pl`、`syscall.h`、`sysproc.c` 必须配套修改）。实现并调通 `sysinfotest` 给了我很大的信心：不仅 `syscall` 的接口链路通了，而且我也学会了在内核中安全地读取和汇总状态信息。

3.3. Lab2 成绩

增加 `time.txt` 文件输入完成实验的时长。输入 `./grade-lab-syscall` 查看实验二的得分。



```
goost@goost: ~/xv6-labs-2021-2
== Test trace 32 grep == trace 32 grep: OK (2.8s)
== Test trace all grep == trace all grep: OK (0.4s)
== Test trace nothing == trace nothing: OK (1.1s)
== Test trace children == trace children: OK (9.4s)
== Test sysinfotest == sysinfotest: OK (1.1s)
== Test time ==
time: OK
Score: 35/35
```

4. page tables

4.1. Speed up system calls

4.1.1. 实验目的

通过在用户态和内核态之间共享只读页面，优化 `getpid()` 系统调用的效率，减少系统调用陷入内核带来的开销。掌握 `xv6` 页表操作方法，理解内核如何通过虚拟内存机制为用户进程提供高效的系统服务。

4.1.2. 实验步骤

A. 新建源文件

本实验无需新建额外源文件

B. 实现程序主体

在 `memlayout.h` 中新增用户态可见的共享页地址

在 `proc.c` 的 `allocproc()` 中：调用 `kalloc()` 分配一页物理内存作为共享页。在该页首地址存放 `struct usyscall`，并赋值 `p->pid`。

在 `proc.c` 的 `proc_pagetable()` 中：使用 `mappages()` 将共享页映射到 `USYSCALL`，权限设置为 只读 (`PTE_R | PTE_U`)。

在 `proc.c` 的 `freeproc()` 中：调用 `kfree()` 释放分配的共享页，防止内存泄漏。

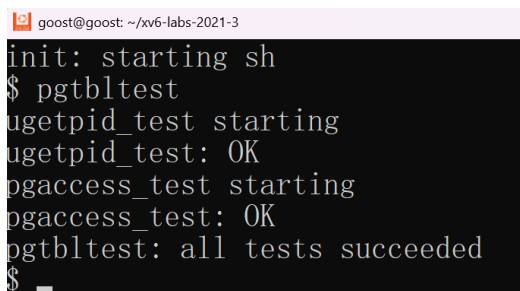
在用户态新增 `ugetpid()` 函数：

```
int ugetpid(void) {  
  
    struct usyscall *u = (struct usyscall*) USYSCALL;  
  
    return u->pid;  
  
}
```

C. 修改 Makefile

确认 `UPROGS` 中包含了 `pgtbltest`

D. 构建并运行



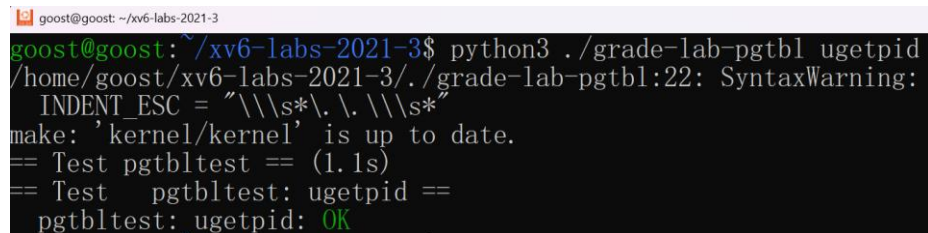
```
goost@goost: ~/xv6-labs-2021-3  
init: starting sh  
$ pgtbltest  
ugetpid_test starting  
ugetpid_test: OK  
pgaccess_test starting  
pgaccess_test: OK  
pgtbltest: all tests succeeded  
$
```

运行后，若 `ugetpid` 正确通过，说明优化成功。

测试时可以看到 `ugetpid` 与 `getpid` 返回结果一致，但前者无需陷入内核，速度更快。

E. 测试

输入 `./grade-lab-pgtbl ugetpid` 进行测试



```
goost@goost: ~/xv6-labs-2021-3
goost@goost: ~/xv6-labs-2021-3$ python3 ./grade-lab-pgtbl ugetpid
/home/goost/xv6-labs-2021-3/./grade-lab-pgtbl:22: SyntaxWarning:
  INDENT_ESC = "\\s*\\.\\.\\.\\s*"
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (1.1s)
== Test   pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
```

4.1.3. 实验中遇到的问题和解决办法

忘记释放共享页，进程退出时未释放共享页，导致内存泄漏。解决：在 `freeproc()` 中调用 `kfree()` 回收内存。

4.1.4. 实验心得

通过本实验，我认识到系统调用的优化思路：对于频繁调用的简单接口，可以通过用户空间共享页减少内核切换开销。类似的机制在 Linux 中也有体现，比如 VDSO。

同时我也体会到页表操作的严谨性，权限控制和内存回收是系统安全与稳定的关键。经过这次实验，我对操作系统的内存管理和系统调用实现有了更深入的理解。

4.2. Print a page table

4.2.1. 实验目的

实现一个函数 `vmprint()`，用于输出 xv6 中某个进程的页表结构，以便更直观地理解 RISC-V 多级页表的层级关系和内核的虚拟内存映射过程。

4.2.2. 实验步骤

A. 新建源文件

不需要新建文件，只需修改 xv6 内核中的 vm.c、defs.h、exec.c 文件。

B. 实现程序主体

在 defs.h 中声明函数：void vmprint(pagetable_t, int);

在 exec.c 中调用 vmprint(), 在 exec() 函数中，return argc; 之前插入：

```
if(p->pid == 1) {  
  
    vmprint(p->pagetable, 0);  
  
}
```

在 vm.c 中实现 vmprint()：

```
void  
  
printwalk(pagetable_t pagetable, uint level) {  
  
    char* prefix;  
  
    if (level == 2) prefix = "..";  
  
    else if (level == 1) prefix = ".. ..";  
  
    else prefix = ".. ...";  
  
    for(int i = 0; i < 512; i++){ // 每个页表有 512 项  
  
        pte_t pte = pagetable[i];  
  
        if(pte & PTE_V){ // 该页表项有效  
  
            uint64 pa = PTE2PA(pte); // 将虚拟地址转换为物理地址  
  
            printf("%s%d: pte %p pa %p\n", prefix, i, pte, pa);  
  
            if((pte & (PTE_R|PTE_W|PTE_X)) == 0){ // 有下一级页表  
  
                printwalk((pagetable_t)pa, level - 1);  
            }  
        }  
    }  
}
```

```

    }

}

}

}

//遍历页表并打印

void

vmprint(pagetable_t pagetable) {

    printf("page table %p\n", pagetable);

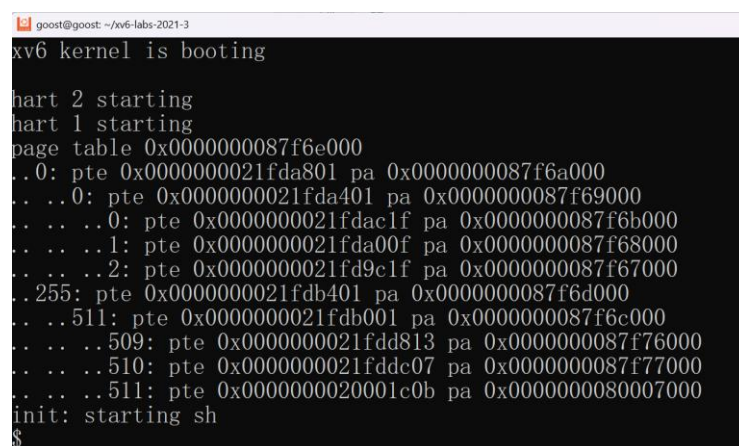
    printwalk(pagetable, 2);

}

```

C. 构建并运行

输入 `make qemu`



```

goost@goost: ~/xv6-labs-2021-3
xv6 kernel is booting

hart 2 starting
hart 1 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
$

```

第一行 `page table 0x...` 是传入 `vmprint()` 的顶层页表的虚拟地址（即 `page table` 的页框地址）。后续每一行描述了一个有效的 PTE（页表项）。每行前面的缩进（`..`）表示该 PTE 在页表树中的深度（顶层索引深度 0，下一层深度 1，依此类推）。

4.2.3. 实验中遇到的问题和解决办法

打印了无效 PTE，忘记判断 PTE_V，打印出很多无效项。解决：增加 `if(pte & PTE_V)` 判断，只打印有效页表项。

4.2.4. 实验心得

通过本实验，我更加直观地理解了 RISC-V 的三级页表结构。`vmprint()` 的递归输出让我能看到从顶层页表到最终物理页的映射关系。同时我也意识到调试工具在操作系统学习中的重要性，`vmprint()` 不仅用于本实验，也能在后续实验帮助定位内存映射问题。

4.3. Detecting which pages have been accessed

4.3.1. 实验目的

实现一个系统调用 `pgaccess(start_va, n, user_mask)`，让内核扫描从 `start_va` 开始的 `n` 个用户页，检测哪些页被访问过（由硬件在 PTE 的 A 位记录），并把结果以位掩码的形式写回用户缓冲区。实验目的有三点：

学会在内核中读取和修改页表 PTE（使用 `walk()`）；理解并使用 RISC-V PTE 的访问（A）位，并在内核中清除该位以便下一次调用可以检测到“自上次查询后”的访问；掌握用户/内核地址数据传输（`argaddr / copyout`）和系统调用链路的完整实现流程（用户声明、`usys` 存根、`syscall.h` 注册、内核实现）。

4.3.2. 实验步骤

A. 实现程序主体

在内核中限制 `n`（页数）上限，例如 `MAX_PGACCESS = 64`（可改），避免用户传入过大值导致资源或时间耗尽。用 `argaddr(0, &start) / argint(1, &n) / argaddr(2, &maskaddr)` 获取参数。在内核里准备一个临时缓冲区 `uint64_t bits[(n+63)/64]`（全部置零）。对于 `i = 0 .. n-1`：计算 `va = start + i * PGSIZE`（确保 `va` 是页对齐的并在用户地址空间范围内）。`pte = walk(pagetable, va, 0)` 获取指向 PTE 的指针（若 `walk` 返回 0，说明该虚

拟页没有 PTE，跳过）。如果 `pte` 有效且 `(*pte & PTE_A)` 为真，则在 `bits[i/64]` 中设置第 `(i%64)` 位为 1，并清除该 PTE 的 A 位 `(*pte = *pte & ~PTE_A)`。在修改 PTE 后（为确保 TLB/缓存一致），最好调用 `sfence_vma()`（或 `sfence_vma()` 等平台适配函数），使处理器看到更新；单核实验环境可能不强制要求，但更稳妥。`copyout(myproc()->pagetable, maskaddr, (char*)bits, bytes)` 把内核临时缓冲区复制回用户空间。

返回 0 表示成功（或返回错误码）。

```
int
sys_pgaccess(void)
{
    // lab pgtbl: your code here.

    uint64 start_va;

    if(argaddr(0, &start_va) < 0)

        return -1;

    int page_num;

    if(argint(1, &page_num) < 0)

        return -1;

    uint64 result_va;

    if(argaddr(2, &result_va) < 0)

        return -1;
```

```

struct proc *p = myproc();

if(pgaccess(p->pagetable,start_va,page_num,result_va) < 0)

    return -1;

return 0;

}

```

B. 修改 Makefile

在 user/usys.pl 增加: entry(pgaccess)

在 user/user.h 增加原型, 例如:

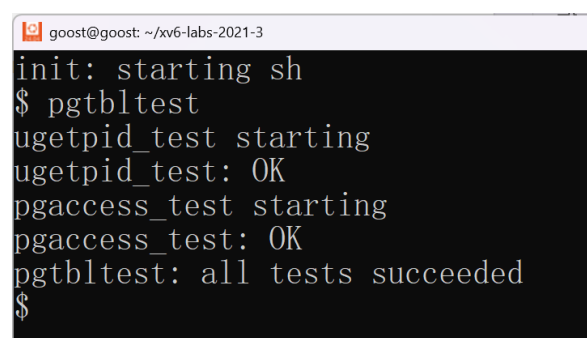
在 kernel/syscall.h 添加宏 SYS_pgaccess (按顺序分配编号)。

在 kernel/syscall.c 的 syscalls[] 表中加入 sys_pgaccess。

确认 UPROGS 包含 pgtbltest 以便自动测试。

C. 构建并运行

输入 make qemu 后, 输入 pgtbltest 运行



```

goost@goost: ~/xv6-labs-2021-3
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$

```

pgtbltest 中的 pgaccess 测试会访问一组页, 然后调用 pgaccess() 检查哪些页被硬件标记为已访问; 如果内核正确检测并清除了 A 位、并正确把位掩码写回用户空间, 测试会输出通过信息 (例如显示 OK)。

4.3.3. 实验中遇到的问题和解决办法

没有 clear A 位（忘了清），第一次 `pgaccess()` 后位被置 1，但第二次 `pgaccess()` 仍然看到这些位被置 1（无法检测“自上次调用后”的访问）。解决：在检测到 PTE_A 时清除该位（`*ptep &= ~PTE_A`），并考虑使用 `sfence_vma()` 保证 TLB 一致性。

4.3.4. 实验心得

通过实现 `pgaccess`，我把“虚拟地址 → 页表 → PTE 标志位”这条链路理解得更清楚了：硬件会在页表位上做标记（A/D），内核可以读取这些位并据此实现高级功能（如写时复制、垃圾回收辅助、页面访问统计等）。实验强调了“小位意义大”：PTE 的单个位（A/D/U 等）对程序权限、可见性和性能策略都至关重要，所以读写 PTE 时必须非常谨慎、并注意并发与 TLB 同步。最有效的调试方式是：先用 `vmprint()` 输出页表结构，确认特定 VA 是否有 PTE；在内核中打印每个检查到的 PTE 值（仅用于调试）；再用 `pgaccess` 的用户端测试用例逐步验证。

4.4. Lab3 成绩

增加 `time.txt` 文件输入完成实验的时长。输入 `./grade-lab-pgtbl` 查看实验三的得分。

```

goost@goost:~/xv6-labs-2021-3$ python3 ./grade-lab-pgtbl
/home/goost/xv6-labs-2021-3/./grade-lab-pgtbl:22: SyntaxWarning:
  INDENT_ESC = "\\s*\\.\\.\\.\\s*"
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (0.8s)
== Test   pgtbltest: ugetpid ==
   pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
   pgtbltest: pgaccess: OK
== Test pte printout == pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests == (141.1s)
== Test   usertests: all tests ==
   usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46

```

5. Traps

5.1. RISC-V assembly

5.1.1. 实验目的

熟悉并观察 RISC-V 的函数调用约定与生成的汇编代码（了解寄存器如何传参、返回地址的保存、帧指针与栈帧布局等）。学会把 C 程序编译成汇编并阅读 call.asm，理解编译器如何实现函数调用、内联与参数传递。回答一组与调用约定、字节序、printf 可变参数相关的问题并把答案写入 answers-traps.txt，为后续的陷阱与回溯实验做准备。

5.1.2. 实验步骤

- A. 生成并查看汇编，输入 make fs.img。
- B. 相关问题

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

答：寄存器 a0 到 a7（即 x10 到 x17）是用于存放函数调用的参数，由以下代码可以得到在调用 printf 的时候，寄存器 a2 保存的是 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.)

答：函数 `f` 调用了函数 `g`；函数 `g` 将传入的参数加上 3 后返回。编译器在生成代码时进行了内联优化，虽然在代码中看起来是通过 `printf("%d %d\n", f(8)+1, 13)` 调用了 `f` 函数，但实际上在汇编代码中，编译器直接将 `f(8)+1` 计算并替换为 12。这表明编译器在编译阶段已经对函数调用进行了优化，因此在 `main` 函数的汇编代码中，没有直接调用 `f` 和 `g`，而是编译器在运行前已经计算并替换了这些函数的结果。

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

答：这个 `auipc` 指令是将 `0x0` 左移 12 位，然后加上当前的 PC 值 `0x30`，并存储到 `ra` 寄存器中。接下来的 `jalr` 指令会根据 `ra` 计算出跳转地址，并将当前 `PC + 4` 存入 `ra`，作为稍后 `printf` 的返回地址。所以，`printf` 函数的位置是 `0x30+1536=0x630`。查阅代码后确认其地址确实在 `0x630`。

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

答：执行完 `jalr` 跳转到 `printf` 之后，`ra` 寄存器的值为 `0x38`。具体过程如下：在程序的 `0x30` 地址处，`auipc ra, 0x0` 将当前程序计数器 `pc` 的值存入 `ra` 中。接着，在 `0x34` 处，`jalr 1536(ra)` 跳转到偏移后的地址，即 `printf` 函数所在的 `0x630` 位置。根据参考资料的信息，执行这条 `jalr` 指令后，`ra` 的值会设置为当前 `pc+4`，也就是返回地址 `0x38`。

(5) Run the following code.

```
unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);
```

What is the output? Here's an ASCII table that maps bytes to characters. The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set `i` to


in order to yield the same output? Would you need to change 57616 to a different value?

答：执行 `make qemu` 并输入 `call` 命令，得到了输出：打印 HE110 World。在 RISC-V 体系结构中，由于其采用小端存储格式，当 `i` 的值为 0x00646c72 时，字节顺序在内存中存储为 72 6c 64 00，对应的 ASCII 字符为“rld”。而数字 57616 转换为 16 进制为 0xe110，因此 `%x` 格式化符号打印输出 “HE110 World”。在大端存储模式下，为了保证输出的字符顺序与小端一致，`i` 的值需要调整为 0x726c6400，这样字节顺序在内存中为 00 64 6c 72，对应的 ASCII 字符依然为 “rld”。无论是大端还是小端，57616 的十六进制值始终为 0xe110，因此打印结果相同。因此，无需改变 57616 的值，但需调整 `i` 的值以适应大端存储格式下的字符输出顺序。

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

```
printf("x=%d y=%d", 3);
```

答：执行 `make qemu` 并输入 `call` 命令，得到了输出：x=3 y=5221



```
goost@goost: ~/xv6-labs-2021-4
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ call
dlr
HE110 World
x=3 y=1
$
```

在这个例子中，`printf` 函数因少传递了一个参数而导致输出不确定的结果。根据函数传参规则，`y=` 后的值应该来自寄存器 `a2`。由于 `a2` 寄存器未被显式赋值，它保留了调用之前的值，这可能是随机的或受之前代码的影响。因此，`y=` 后面的值为一个无法预测的不可靠值。简单来说，`printf` 函数试图读取的参数数量超过了实际提供的参数，因此从寄存器中获取到的未定义值导致输出不可预测。

5.1.3. 实验心得

通过实际查看编译器生成的汇编（call.asm），我把 RISC-V 的调用约定从抽象概念变成了可观测的、具体的指令序列：函数参数靠 a0..a7，返回值放在 a0/a1，返回地址保存在 ra，s0/fp 用作帧指针，sp 管理栈。理解这些对后面实现 backtrace、处理 trapframe 与实现用户级 alarm 非常关键。遇到编译器内联是一个常见“陷阱”：为了观察“真实的调用”，要在编译时关闭优化（-O0 -fno-inline）。对字节序（endianess）与 printf 的可变参数机制要亲手验证一遍（把 i 的十六进制拆成字节，再看 %s 如何打印），这样在阅读别人的汇编或调试时不会被迷惑。

5.2. Backtrace

5.2.1. 实验目的

实现内核级 backtrace()，能够沿着内核栈帧链打印调用链上保存的返回地址，帮助定位内核中出错的位置。熟悉 RISC-V 的帧指针约定（s0 / fp）和栈帧布局：如何从当前帧顺着保存的帧指针走上去并读出每个栈帧保存的返回地址。掌握在内核中安全遍历内核栈（利用 PGROUNDDOWN / PGROUNDUP 判断栈边界）以及把地址转换为源码位置（addr2line）的调试流程。

5.2.2. 实验步骤

A. 实现程序主体

GCC 将当前函数的帧指针放在寄存器 s0（即 fp）。我们通过内联汇编读出 s0 的当前值作为当前帧指针。每个栈帧约定如下（xv6 / RISC-V GCC 常见约定）：fp（当前帧指针）指向该帧的基址。*(fp - 8) 存放本帧的返回地址（saved ra）。*(fp - 16) 存放上一个帧的 fp（saved fp）。

从当前 fp 开始，循环读取 ra = *(fp - 8) 并打印，然后把 fp = *(fp - 16) 继续向上，直到 fp 越过当前内核栈页（用 PGROUNDDOWN(fp) / PGROUNDUP(fp) 判断）或 fp 为 0。

在 kernel/riscv.h 中添加读取 fp 的内联函数：


```
// kernel/riscv.h
```

```
static inline uint64
```

```
r_fp(void)
```

```
{
```

```
    uint64 x;
```

```
    asm volatile("mv %0, s0" : "=r" (x));
```

```
    return x;
```

```
}
```

在 kernel/defs.h 中声明 backtrace() 原型: void backtrace(void);

在 kernel/printf.c 中实现 backtrace(): #include "param.h"

```
#include "types.h"
```

```
#include "riscv.h"
```

```
#include "defs.h"
```

```
#include "memlayout.h"
```

```
void
```

```
backtrace(void)
```

```
{
```

```
    printf("backtrace:\n");
```

```
    uint64 fp = r_fp();           // 当前帧指针 (s0)
```

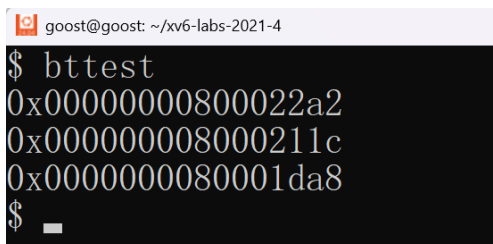
// 每个内核线程分配一页内核栈，栈顶/底可由
PGROUNDDOWN/PGROUNDUP 计算

```
while (fp != 0) {  
  
    // 检查 fp 是否在当前内核栈范围内  
  
    uint64 stackbase = PGROUNDDOWN(fp);  
  
    uint64 stacktop  = stackbase + PGSIZE;  
  
    if (fp < stackbase || fp >= stacktop) break;  
  
  
    // saved return address 位于 fp - 8  
  
    uint64 ra = *((uint64*)(fp - 8));  
  
    if (ra == 0) break;  
  
    printf("0x%p\n", ra);  
  
  
    // saved previous frame pointer 位于 fp - 16  
  
    uint64 next_fp = *((uint64*)(fp - 16));  
  
    if (next_fp == fp) break; // 防止死循环（守门）  
  
    fp = next_fp;  
  
}  
  
}
```

在 kernel/sysproc.c 中的 sys_sleep() 或对应位置插入 backtrace() 调用

B. 构建并运行

输入 `make qemu`, 输入 `bttest` 结果如下:



```
goost@goost: ~/xv6-labs-2021-4
$ bttest
0x00000000800022a2
0x000000008000211c
0x0000000080001da8
$
```

每一行的十六进制地址是对应栈帧保存的返回地址（即被调用函数在返回时会跳回的指令地址）。这些地址是内核文本段（`kernel/kernel`）中的指令地址。

5.2.3. 实验中遇到的问题和解决办法

`fp` 指针越界或陷入无限循环, `backtrace` 在遍历时访问非法地址导致 `panic`, 或循环不终止。栈帧布局不符合假定（不同编译器/版本保存偏移不同）, 或读取的 `fp` 被损坏。解决: 在每次迭代前用 `PGROUNDDOWN(fp) / PGROUNDUP(fp)` 校验 `fp` 仍在当前内核栈页范围内; 当 `next_fp == 0` 或 `next_fp == fp` 时立即停止; 增加守门检查（例如限制最多迭代深度为 32）。

5.2.4. 实验心得

实现 `backtrace` 的过程把“编译器产生的栈帧约定”从纸上概念变为可操作的调试工具。能在 `kernel panic` 或在特定 `syscall` 时直接看到调用链, 对内核调试帮助极大。实现时的关键点是: 要相信约定但又要做防守式编程——假定 `fp - 8 / fp - 16` 存放返回地址和上一个 `fp`, 但在读取之前必须检查 `fp` 是否在合理的内核栈范围内, 防止内存访问异常。把 `addr2line` 与 `objdump` 的使用结合起来, 能把抽象地址快速映射到源码行, 这在定位 `bug` 时非常高效。

5.3. Alarm

5.3.1. 实验目的

在 `xv6` 中实现用户级定时回调机制: 添加系统调用 `sigalarm(int ticks, void (*handler)())` 与 `sigreturn()`。使用户进程在自身消耗了指定 CPU

tick 数量后由内核暂时跳转执行用户提供的 handler，handler 返回后进程能无缝恢复到被打断处继续执行。学习并实践 trap（陷阱）处理、trapframe 的保存/恢复、以及如何在内核安全地修改返回到用户态的上下文（epc、寄存器等）。理解并避免 handler 重入、并学会如何用最小改动完成可靠的恢复逻辑。

5.3.2. 实验步骤

A. 实现程序主体

在 user/user.h 中添加原型：

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

```
int sigreturn(void);
```

在 user/usys.pl 添加：

```
entry(sigalarm)
```

```
entry(sigreturn)
```

在 kernel/syscall.h 中添加枚举

在 proc 结构中新增字段（kernel/proc.h）

```
int alarm_interval;
```

```
int alarm_ticks_left;
```

```
uint64 alarm_handler;
```

```
int alarming;
```

```
struct trapframe alarm_tf;
```

```
int alarm_saved;
```

实现 sys_sigalarm() 与 sys_sigreturn()。

uint64

sys_sigalarm(void)

{

int ticks;

uint64 handler;

struct proc *p = myproc();

if(argint(0, &ticks) < 0) return -1;

if(argaddr(1, &handler) < 0) return -1; // or argaddr for pointer

acquire(&p->lock);

p->alarm_interval = ticks;

p->alarm_handler = handler;

p->alarm_ticks_left = ticks; // 重新装载计数器

p->alarming = 0;

p->alarm_saved = 0;

release(&p->lock);

return 0;

}

uint64

```

sys_sigreturn(void)

{

    struct proc *p = myproc();

    acquire(&p->lock);

    if (!p->alarm_saved) {

        release(&p->lock);

        return -1; // 没有 backup, 非法调用

    }

    // 恢复 trapframe (拷贝回当前 tf)

    *p->trapframe = p->alarm_tf; // 假设 p->trapframe 为指针到用户 trapframe

    p->alarming = 0;

    p->alarm_saved = 0;

    // 重新装载计时器

    p->alarm_ticks_left = p->alarm_interval;

    release(&p->lock);

    return 0;

}

```

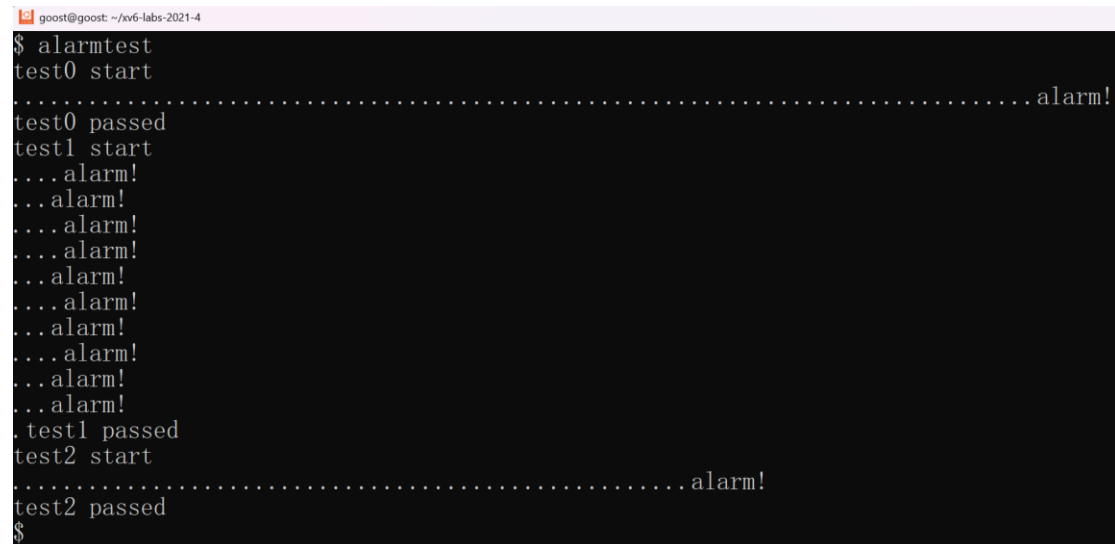
最后在 usertrap / timer interrupt 处加入计数与触发。

B. 修改 Makefile

在 Makefile 的 UPROGS 把 alarmtest 加入

C. 构建并运行

输入 `make qemu`，输入 `alarmtest` 运行，结果如下：



```
goost@goost: ~/xv6-labs-2021-4
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
...alarm!
...alarm!
...alarm!
...alarm!
...alarm!
...alarm!
...alarm!
...alarm!
...alarm!
test1 passed
test2 start
.....alarm!
test2 passed
$
```

5.3.3. 实验中遇到的问题解决办法

`alarmtest` 触发了 `alarm!` 但程序随即崩溃，只把 `epc` 改到 `handler` 地址但没有把整个 `trapframe` 备份并在 `sigreturn` 恢复，导致 `handler` 返回后没有正确恢复寄存器与程序计数器。解决：在触发前把 `*p->trapframe` 完整拷贝到 `p->alarm_tf`（备份），并在 `sys_sigreturn` 中把备份拷回 `*p->trapframe`。这会恢复所有通用寄存器、返回地址和 `epc`。

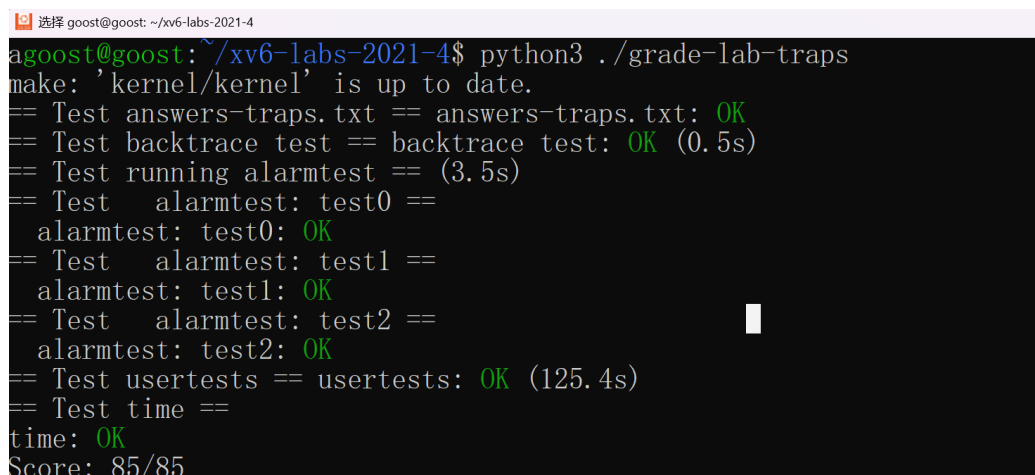
5.3.4. 实验心得

实验把“如何把内核态的中断处理转成用户级的回调”这条链路讲清楚了：关键是保存/修改/恢复 `trapframe`。把 `epc` 指向用户 `handler`，是实现“在返回用户态前先执行 `handler`”的简洁方式；而 `sigreturn()` 恢复备份的 `trapframe` 则确保能无缝回到被打断处继续执行。最容易出错的点是没有完整保存上下文或没有防止 `handler` 重入。因此实践中采用“拷贝整个 `struct trapframe`”和“设置 `inhandler` 标志”的模式既直观又健壮。调试建议：先实现 `test0`（只需能看到 `alarm!`），再实现上下文保存/恢复（通过 `sys_sigreturn` 恢复）。使用 `make CPUS=1 qemu-gdb` 单核并配合断点观察 `trapframe` 与 `epc` 的变化能大大缩短调试时间。实验还让我理解到操作系统在

实现用户级信号/回调时的复杂性：看似“只跳到一个函数执行”但背后涉及保全寄存器、处理递归、保证线程安全以及和 scheduler/syscall 的交互。

5.4. Lab4 成绩

增加 time.txt 文件输入完成实验的时长。输入 ./grade-lab-traps 查看实验四的得分。



```
agoost@agoost:~/xv6-labs-2021-4$ python3 ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (0.5s)
== Test running alarmtest == (3.5s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests == usertests: OK (125.4s)
== Test time ==
time: OK
Score: 85/85
```

6. Copy-on-Write Fork for xv6

6.1. Implement copy-on write

6.1.1. 实验目的

实现 Copy-On-Write (COW) 版本的 fork(): 父子进程在 fork() 后共享用户物理页（不立即复制），只在进程尝试写某页时才复制该页。通过此实验理解页表 PTE 的位（读/写/用户/保留位）、缺页（page fault）处理路径，以及如何在内核里安全地分配/释放/共享物理页（引用计数）。减少 fork() 的内存与时间开销，同时保证语义正确：写时复制后每个进程最终看到它自己的可写页。

6.1.2. 实验步骤

A. 实现程序主体

在 kernel/riscv.h（或合适的头文件）定义 COW 标记宏（使用 RSW 软件位）：`#define PTE_COW 0x200`

在 kalloc.c 中添加一个整型数组 `static int refcount[N]`，其中索引可以按 `pa / PGSIZE` 或 `(pa >> PGSHIFT)` 计算。数组大小设为能覆盖最大物理页数

提供封装函数：

```
void
```

```
page_ref_init(void) { /* 在 kinit 时初始化数组 */ }
```

```
void
```

```
page_ref_inc(uint64 pa) {
```

```
    int idx = pa >> PGSHIFT;
```

```
    acquire(&ref_lock);
```

```
    refcount[idx]++;
```

```
    release(&ref_lock);
```

```
}
```

```
int
```

```
page_ref_dec(uint64 pa) {
```

```
    int idx = pa >> PGSHIFT;
```

```
    acquire(&ref_lock);
```

```
    refcount[idx]--;
```

```

    int c = refcount[idx];

    release(&ref_lock);

    return c;

}

int

page_ref_get(uint64 pa) { /* 返回 refcount[idx] (带锁) */ }

```

在 `kalloc()` 返回新页时把对应 `refcount[idx] = 1`。（或在分配后调用 `page_ref_inc`）

修改 `kfree()`：当要“释放”物理页时，先 `page_ref_dec(pa)`，只有当计数为 0 时才把页放回 `free list`；否则不放回。

修改 `uvmcopy()`，把 `uvmcopy(pagetable, newpagetable, sz)` 改为：对于父每个有效用户页：找出父 PTE，得到物理页 `pa`。在子页表上建立一个映射到同一 `pa` 的 PTE（跟父一样的权限），但清除写权限（`PTE_W`），并设置 `PTE_COW`（软件位）以标识它是 COW 映射。同时在父的 PTE 上清除写权限并设置 `PTE_COW`（父与子都要变为不可写 + COW）。对该 `pa` 调用 `page_ref_inc(pa)`。

在处理 `page fault`（或 `load/store fault`）时，要识别是否为 COW 写故障：

```

uint64 va = r_scause()? (或从 trapframe->... 获取 faulting va using r_stval())

pte_t *p = walk(p->pagetable, va, 0);

if (p == 0 || !(*p & PTE_V)) {

    // 原本是未映射 -> 按原行为处理（可能 kill）

} else if ((*p & PTE_COW) && !(*p & PTE_W)) {

```

```

// COW 情形

uint64 pa = PTE2PA(*p);

if (page_ref_get(pa) > 1) {

    // 需要复制

    char *mem = kalloc();

    if (mem == 0) {

        // 内存不足：按要求 kill 进程

        myproc()->killed = 1;

        return;

    }

    memmove(mem, (char*)pa_to_kernel_va(pa), PGSIZE);

    uint64 newpa = kernel_va_to_pa(mem); // 平台相关方式获得 pa

    // 安装新映射： update PTE to point to newpa with PTE_W set, clear PTE_COW

    *p = PA2PTE(newpa) | (original_flags & ~PTE_COW) | PTE_W;

    // decrement original page refcount because this process no longer references it

    page_ref_dec(pa);

} else {

    // 只有当前进程引用该页（refcount == 1），可以直接 make writable:

    *p = *p | PTE_W;    // 允许写

    *p = *p & ~PTE_COW; // 清除 COW 标志

```

```

    }

    sfence_vma(); // 刷新 TLB (如需要)

    return; // 返回用户态, 重试写操作

}

```

修改 `uvmunmap()` / 释放页表时的逻辑, 当进程释放一段用户虚拟内存 (例如 `uvmunmap()` 或 `uvmfree()` 在 `exit` 时), 在移除每个 PTE 前需要: 取出 `pa`, 并对该 `pa` 调用 `page_ref_dec(pa)`; 如果返回值为 0, 则真正调用 `kfree(pa)` 把页放回空闲链表; 否则不要释放。清理页表项 (`*pte = 0`)。

```

void uvmunmap(...) {

    for each va to unmap:

        pte = walk(pagetable, va, 0);

        if (pte && (*pte & PTE_V)) {

            uint64 pa = PTE2PA(*pte);

            if (page_ref_dec(pa) == 0) {

                kfree((char*)pa);

            }

            *pte = 0;

        }

    }

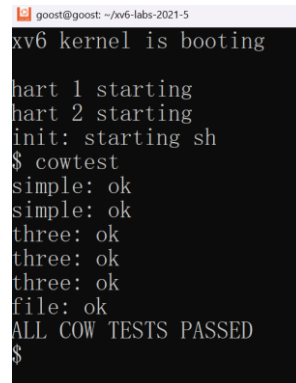
}

```

修改 `copyout()`, `copyout()` 在写入用户地址时也可能碰到 COW 页 (用户地址对应 PTE 为 COW / not writable), 所以在写入之前复用上面的 COW 处理逻辑: 如果目标 PTE 有 `PTE_COW` 或缺写权限, 则先触发一次本地的 COW 处理 (与 `page fault` 相同的分支), 使该进程获得可写页, 然后再复制数据。

B. 构建并运行

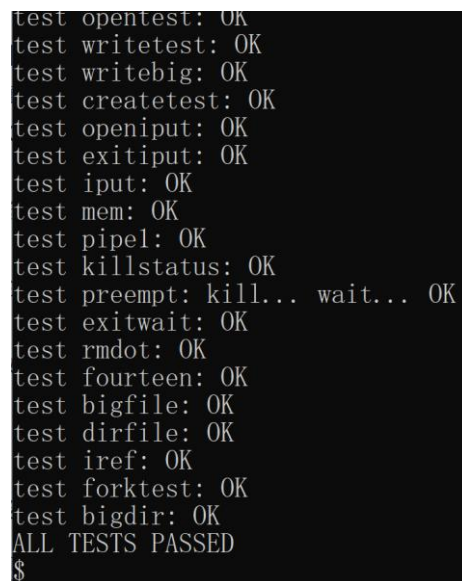
输入 `make qemu`, 然后输入 `cowtest` 运行测试。



```
goost@goost: ~/xv6-labs-2021-5
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
$
```

在未实现 COW 的 xv6 中 `fork()` 为子复制整块内存, 若父分配了很多页则没有足够物理内存导致 `fork` 失败。实现 COW 后 `fork()` 仅复制页表、增加引用计数、清写位, 因此不会消耗那么多即时物理内存, `simple` 测试得以通过。

在命令行中输入 `usertests`, 产生结果如下:



```
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipel: OK
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

6.1.3. 实验中遇到的问题和解决办法

写入共享页时没有触发正确的分离（写操作崩溃或父子都被影响），没有在父 PTE 上也清除写权限（只在子上去掉写），导致写在父上直接修改了物理页（破坏语义）。解决：保证 `uvmcopy()` 对父和子都去掉写权限并设置

PTE_COW。在 page fault 的处理路径中要精确判断：只有当 PTE_COW 被设置且是写访问才走 COW 分离逻辑；对真正未映射地址或对没有 COW 标志的只读页按原策略处理（可能 kill）。调试时打印出 faulting VA、PTE 值与 refcount，帮助定位为何没走分离分支。

6.1.4. 实验心得

通过实现 COW，我把页表、PTE 各个位的含义、缺页处理、以及内核如何在用户态和内核态之间控制内存访问等知识串成了一条链条。COW 是一个将性能与正确性折中的经典技术——它能显著减少 fork() 时无谓的物理页复制，但要求在内核里精确地追踪共享与写入语义。实现中最容易出错的不是“分配一页/复制一页”的代码本身，而是引用计数的一致性和边界情况（出错回滚、进程退出、uvmunmap）。因此把引用计数操作封装、并在所有会修改映射的路径中对称调用 inc/dec 非常重要。

6.2. Lab5 成绩

增加 time.txt 文件输入完成实验的时长。输入 ./grade-lab-cow 查看实验五的得分。

```
goost@goost:~/xv6-labs-2021-5$ python3 ./grade-lab-cow
make: 'kernel/kernel' is up to date.
== Test running cowtest == (4.5s)
== Test simple ==
    simple: OK
== Test three ==
    three: OK
== Test file ==
    file: OK
== Test usertests == (129.3s)
== 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
```

7. Multithreading

7.1. Uthread: switching between threads

7.1.1. 实验目的

本次实验目的是设计并实现一个**用户级线程（uthread）**的上下文切换机制：实现 `thread_create()`、`thread_schedule()` 和 `thread_switch`，使若干线程能在同一进程内各自拥有栈并轮流执行。通过本实验我想理解并掌握：如何为新线程建立独立栈与初始上下文（使其第一次被调度时能从指定函数开始执行）；在用户空间保存/恢复上下文（寄存器与栈指针），实现线程间切换；如何调试并验证线程调度和上下文保存的正确性。

7.1.2. 实验步骤

A. 实现程序主体

给 `struct thread` 增加用于保存上下文的字段，比如保存 callee-saved 寄存器与栈指针：

```
#define N_SAVEREG 12 // s0..s11

struct thread {

    int id;

    int state;           // RUNNABLE, RUNNING, EXITED ...

    void *stack;         // 指向分配的栈底（malloc 返回）

    uint64 regs[N_SAVEREG]; // 保存 s0..s11

    uint64 sp;           // 保存 sp，以便恢复栈

    void (*start_fn)(void*); // thread entry

    void *arg;
```

```

    struct thread *next;

};

```

`thread_create()`: 分配 `struct thread`, 用 `malloc` 分配线程栈 (例如 `STACK_SZ = 4096 * 4`)。初始化 `thread->stack` 与 `thread->sp`: 把 `sp` 设为栈顶 (栈生长向下, 记得对齐到 16 字节)。在栈上为首次运行构造一个“假栈帧”: 把返回地址 `ra` 设为一个 `thread_exit` 的地址或 `wrapper`, 使得当新线程首次被调度并 `ret` 时会调用 `thread_exit()`。具体策略是把 `thread->sp` 降低若干并把入口函数地址放到栈上, 或者把 `regs` 中的保存位 (如 `s0..s11`) 设置为合适的初值并设置一个“标记”表明这是首次运行。把 `start_fn`、`arg` 存入 `thread`, 并把 `state = RUNNABLE`, 将该线程插入 `runqueue`。

```

struct thread *thread_create(void (*fn)(void*), void *arg) {

    t = malloc(sizeof(*t));

    t->stack = malloc(STACK_SZ);

    sp = (uint64)t->stack + STACK_SZ;

    sp = sp - sp%16; // 16-byte 对齐

    // 为首次运行设置好 stack frame, 使得恢复 sp 后执行 ret 会跳到 thread_stub

    // thread_stub: calls fn(arg); thread_exit();

    // 把初始 s0..s11 置 0

    for (i=0; i<N_SAVEREG; i++) t->regs[i]=0;

    t->sp = sp;

    t->start_fn = fn;

    t->arg = arg;

    t->state = RUNNABLE;
}

```



```

    enqueue(runqueue, t);

    return t;

}

```

`thread_schedule()`: 调度函数要选出 `next` (RUNNABLE) 线程并把当前线程 `t` 与 `next` 进行切换。核心是调用 `thread_switch(&t->context, &next->context)`。

```

void thread_schedule() {

    struct thread *cur = current_thread;

    struct thread *next = pick_runnable();

    if (!next) {

        printf("thread_schedule: no runnable threads\n");

        return;

    }

    next->state = RUNNING;

    cur->state = (cur->exited? EXITED : RUNNABLE);

    current_thread = next;

    thread_switch(&cur->regs[0], &next->regs[0]); // 传 old 和 new 上下文指针

}

```

`thread_switch`: `thread_switch` 的职责是: 把 `callee-save` 寄存器和 `sp` 从当前线程保存到 `old` 指向的内存位置, 然后把 `sp` 和 `callee-save` 寄存器从 `new` 指向的内存加载回来, 并返回 (从而在新线程的上下文继续执行)。

```

// a0 = old_ctx, a1 = new_ctx

```

```

thread_switch:

    // save callee-saves to *a0

    sd s0, 0(a0)

    sd s1, 8(a0)

    ...

    sd s11, 88(a0)

    sd sp, 96(a0)    // 保存原线程的栈指针

    // load callee-saves from *a1

    ld s0, 0(a1)

    ld s1, 8(a1)

    ...

    ld s11, 88(a1)

    ld sp, 96(a1)    // 恢复新线程的栈指针

    ret

```

B. 构建并运行

在终端中执行 `make qemu` 编译并运行 `xv6`。在命令行中输入 `uthread`,结果如下:

```

goost@goost: ~/xv6-labs-2021-6
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
$

```

C. 测试

输入 `./grade-lab-thread uthread` 测试

```

goost@goost: ~/xv6-labs-2021-6
goost@goost:~/xv6-labs-2021-6$ python3 ./grade-lab-thread uthread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.7s)
goost@goost:~/xv6-labs-2021-6$

```

7.1.3. 实验中遇到的问题解决办法

切换后程序崩溃，切换到某线程后 crash。新线程的栈未正确对齐或栈顶/初始 frame 安排错误，导致 ret 跳到垃圾地址；或在 `thread_switch` 保存/恢复寄存器时偏移计算错误（读写了错误的内存地址）。解决：确保分配栈时 `sp` 对齐到 16 字节；在栈上为首次执行放置一个已知的返回地址（指向 `thread_start wrapper`），`wrapper` 调用 `start_fn(arg)` 后再调用 `thread_exit()`；在汇编中校验 `sd/ld` 的偏移与 C 结构布局严格一致；在 `gdb` 中打印内存确认保存位置正确。

7.1.4. 实验心得

通过实现用户级线程切换，我更深刻地理解了调用约定（`caller-save` vs `callee-save`）对上下文切换实现的影响。只保存 `callee-save` 寄存器可以大幅简化切换逻辑，这正是操作系统/库设计上利用 ABI 的巧妙之处。切换上下文看似“保存几个寄存器，恢复几个寄存器”，但细节非常容易出错：栈对齐、首次进入线程的“bootstrap” frame、以及在汇编中严格对齐 C 结构偏移，都是常见坑。调试这类问题时，使用 `gdb` 单步汇编与在关键点打印寄存器/内存状态非常有帮助。

7.2. Using threads

7.2.1. 实验目的

理解并发访问共享数据结构（哈希表）时会产生的竞态与数据破坏问题。学会用 POSIX 线程（pthread）和互斥锁保护共享结构，保证并发正确性。在保证正确性的同时，尝试用更细粒度的锁（例如“每个桶一个锁”）提高并行性能，观察并行加速效果。熟悉在真实 Linux/macOS 主机上进行多线程性能实验及调试方法（该部分不能在 xv6/qemu 上完成）。

7.2.2. 实验步骤

A. 实现程序主体

理解缺失键的发生原因：在两个线程并发执行 `put(key)` 时，可能发生如下竞态导致 key 丢失：线程 T1 调用 `put(k)`：计算 hash，找到 bucket B；读取 bucket B 的头指针（NULL）。在 T1 准备插入新节点前，线程切换到 T2。线程 T2 也调用 `put(k2)`，也往同一个 bucket B 插入（把 head 指向新节点 N2）。切回 T1，T1 继续执行，基于旧的 head（NULL）把自己的新节点 N1 链到 head（即 `N1->next = NULL`），然后把 bucket head 指向 N1。结果 N2 被覆盖（丢失），此前由 T2 插入的节点不在链表中 —— 导致 “keys missing”。结论：并发修改同一 bucket 的链表必须用互斥保护。

在 `notxv6/ph.c` 顶部声明并初始化互斥锁：`pthread_mutex_t ph_lock;`

在 `main()` 或初始化函数中：`pthread_mutex_init(&ph_lock, NULL);`

在 `put()` 中（对哈希表修改的临界段）加锁/解锁：`void put(...) {`

```
    pthread_mutex_lock(&ph_lock);
```

```
    // 原有 insert 逻辑（计算 bucket，插入节点）
```

```
    pthread_mutex_unlock(&ph_lock);
```

```
}
```

在 `get()` 中（若 `get` 的实现也会读取共享结构，且并发读写可能不安全）：`int get(...)` {

```
    pthread_mutex_lock(&ph_lock);

    // 读取逻辑

    pthread_mutex_unlock(&ph_lock);

}
```

每桶锁（提高并行性，目标 `ph_fast`），替换全局锁为“每个 bucket 一个锁”。在文件顶部：`#define NHASH 4096`

```
pthread_mutex_t bucket_lock[NHASH];
```

初始化（在 `main`）：`for (i = 0; i < NHASH; i++)`
`pthread_mutex_init(&bucket_lock[i], NULL);`

在 `put()`：`int h = hash(key);`

```
pthread_mutex_lock(&bucket_lock[h]);
```

```
insert_into_bucket(h, key, value);
```

```
pthread_mutex_unlock(&bucket_lock[h]);
```

在 `get()`：`int h = hash(key);`

```
pthread_mutex_lock(&bucket_lock[h]);
```

```
val = lookup_in_bucket(h, key);
```

```
pthread_mutex_unlock(&bucket_lock[h]);
```

// 简化示例：`notxv6/ph.c` 中修改

```
pthread_mutex_t bucket_lock[NHASH];
```

```

void init_hash() {

    for (int i = 0; i < NHASH; i++)

        pthread_mutex_init(&bucket_lock[i], NULL);

}


void put(int key, int value) {

    int h = key % NHASH;

    pthread_mutex_lock(&bucket_lock[h]);

    // 原有插入逻辑：遍历 bucket[h] 链表，插入新节点

    pthread_mutex_unlock(&bucket_lock[h]);

}


int get(int key) {

    int h = key % NHASH;

    pthread_mutex_lock(&bucket_lock[h]);

    // 查找并返回

    pthread_mutex_unlock(&bucket_lock[h]);

}

```

B. 构建并运行

在终端中执行 `make ph` 编译并运行 `xv6`。在命令行中输入 `./ph 1` 和 `./ph 2`，产生结果如下，`keys missing` 在单线程和多线程下都为 0，符合预期。

```
goost@goost: ~/xv6-labs-2021-6
goost@goost:~/xv6-labs-2021-6$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
goost@goost:~/xv6-labs-2021-6$ ./ph1
-bash: ./ph1: No such file or directory
goost@goost:~/xv6-labs-2021-6$ ./ph 1
100000 puts, 5.125 seconds, 19510 puts/second
0: 0 keys missing
100000 gets, 4.933 seconds, 20270 gets/second
```

```
goost@goost: ~/xv6-labs-2021-6
goost@goost:~/xv6-labs-2021-6$ ./ph 2
100000 puts, 1.798 seconds, 55618 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 5.454 seconds, 36674 gets/second
goost@goost:~/xv6-labs-2021-6$ _
```

7.2.3. 实验中遇到的问题和解决办法

粗粒度锁正确但性能差，用单个全局锁后 `./ph 2` 的 `puts/sec` 甚至不如 `./ph 1`，或者提升不足。全局锁把所有并行工作串行化，造成并行浪费。实现更细粒度锁（每桶锁或分段锁）。注意避免在单次操作中同时获取多个桶锁而导致死锁（若必须获取多个锁，规定固定的获取顺序）。还可以减少临界区内工作量（只在真正需要读/写共享数据时才持锁）。

7.2.4. 实验心得

初次并发编程时容易只关注性能（想要最大并行度），结果往往错过了关键的一点：先用正确的（可能粗粒度）同步保证数据一致，再逐步放宽粒度提升性能。实践中我先实现全局锁保证 `ph_safe`，然后在保证正确的前提下逐步优化到每桶锁以满足 `ph_fast`。每桶锁在多数情况下是一个很好的折中：它在常见负载下能显著提升并行度，但同时会增加死锁、竞态和实现复杂性（比如重哈希、跨桶操作）。调试细粒度同步时要耐心，借助日志和单步排查非常有帮助。

7.3. Barrier

7.3.1. 实验目的

本实验实现一个线程屏障（barrier）：让 n 个线程在某一点都到达后再同时放行。通过本实验可以：熟悉 pthread 条件变量（pthread_cond_t）与互斥锁的配合用法；理解如何在多轮（round）同步场景下避免竞态与“绕圈”竞争问题；能写出健壮的屏障实现，能经受住线程快速循环进入/离开屏障的竞争场景

7.3.2. 实验步骤

A. 实现程序主体

使用 pthread_mutex_t 保护共享状态，用 pthread_cond_t 使线程等待并被唤醒。维护三个状态变量：nthread：屏障参与线程总数（常量，由外部传入，不在每轮中修改）；count：当前轮次已经到达屏障的线程计数；round：当前轮次编号（整数，最后到达的线程把它加 1 并广播唤醒）。每个线程进入 barrier() 时记录自己看到的 round（myround = round），将 count++。若 count == nthread，说明本轮最后一个到达：把 count 清 0，round++，pthread_cond_broadcast(&cond) 唤醒其它等待者。否则调用 pthread_cond_wait(&cond, &mutex) 在 cond 上等待；用 while（myround == round）pthread_cond_wait(...) 防止虚假唤醒（spurious wakeups）。这样就能保证：只有当 round 变化后（即本轮全部到齐）正在等待的线程才会被唤醒并继续；新到达的线程看到的 myround 已不同，不会误解早被唤醒的条件。

```
void barrier(void) {

    pthread_mutex_lock(&bstate.mutex);

    int myround = bstate.round;      // 记住当前轮次

    bstate.count++;                  // 表示又一个线程到达

    if (bstate.count == bstate.nthread) {

        // 本轮最后一个到达
```



```

        bststate.count = 0;                // 为下一轮重置计数

        bststate.round++;                  // 开启下一轮

        pthread_cond_broadcast(&bststate.cond); // 唤醒本轮所有等待者

    } else {

        // 等待，注意使用 while 循环以防虚假唤醒

        while (myround == bststate.round) {

            pthread_cond_wait(&bststate.cond, &bststate.mutex);

        }

    }

    pthread_mutex_unlock(&bststate.mutex);

}

```

B. 构建并运行

在终端中执行 `make barrier` 编译并运行 `notxv6/barrier.c`。在命令行中输入 `./barrier 2`，产生结果如下：

```

gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
goost@goost:~/xv6-labs-2021-6$ ./barrier 2
OK; passed

```

7.3.3. 实验中遇到的问题和解决办法

- A. 使用 `if` 而不是 `while` 包裹 `pthread_cond_wait` 导致虚假唤醒出错，在某次运行中，有线程在 `pthread_cond_wait` 返回后直接继续，但 `round` 实际未改变，结果提前离开导致断言失败或不正确行为。解决：把 `if` 改为 `while (myround == bststate.round) pthread_cond_wait(...)`，确保在 `cond_wait` 返回后重新检查判断条件。

- B. 错误地修改了 `nthread` 或复用 `nthread` 导致轮次竞争，在某些实现中把 `nthread` 用作“剩余到达线程数”并在退出时修改，结果被快速重入的线程干扰下一轮的计数。解决：把 `nthread` 视为常量（参与屏障的线程总数），不要在每轮修改。为每轮计数使用单独的 `count` 变量（如上实现）。每轮结束由最后一个线程把 `count` 复位为 0，并把 `round` 加一。

7.3.4. 实验心得

这个题目强化了条件变量与互斥锁的配合使用：`cond_wait` 总与某个 predicate（这里是 `myround != round`）配套出现，并且需要在循环中判断以处理虚假唤醒。设计上关键是把“每轮状态”与“跨轮竞争”分离：`round` 用来区分不同轮次，`count` 只负责当前轮次到达计数，`nthread` 始终为常量。用 `round` 防止“快速线程绕圈”影响仍在等待的线程，是一个通用且可靠的技巧。调试这类同步问题时，打印当前 `round/count` 与线程 `id` 的日志非常有帮助（不过在高并发下日志本身也会干扰调度，所以仅在调试阶段使用）。

7.4. Lab6 成绩

增加 `time.txt` 文件输入完成实验的时长。输入 `./grade-lab-thread` 查看实验六的得分。

```
goost@goost: ~/xv6-labs-2021-6
goost@goost:~/xv6-labs-2021-6$ python3 ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.1s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (9.0s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (17.3s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (2.2s)
== Test time ==
time: OK
Score: 60/60
goost@goost:~/xv6-labs-2021-6$ _
```

8. Networking

8.1. Your Job

8.1.1. 实验目的

为 xv6 实现对仿真 E1000 网卡的驱动，完成发送 (`e1000_transmit()`) 与接收 (`e1000_recv()`) 两部分。理解并实践基于描述符环 (TX/RX ring) 的 DMA 驱动设计：如何填描述符、如何处理硬件状态位、如何回收/分配数据缓冲 (mbuf)。掌握与设备交互的寄存器使用 (特别是 E1000_TDT / E1000_RDT 等)，并学会在中断/系统调用并发场景下用锁保护驱动内部状态。能用 `nettests` / `make server` / `tcpdump` 对驱动进行端到端验证，并通过 `make grade` 的网络测试。

8.1.2. 实验步骤

A. 实现程序主体

全局 `regs` 指向网卡控制寄存器映射，读写 `regs[E1000_TDT]` / `regs[E1000_RDT]` 与驱动通信。TX 描述符数组 `tx_ring[TX_RING_SIZE]`，每个 `struct tx_desc` 包含 `addr/len/cmd/status` 等字段；驱动通常在内核中还保存并行的 `struct mbuf *tx_mbuf[TX_RING_SIZE]` 数组以保存对应 mbuf 指针。RX 描述符数组 `rx_ring[RX_RING_SIZE]`，初始化时每个描述符里放置一个 mbuf 的 `m->head` (缓冲地址)。收到包时硬件把长度与状态写进对应描述符。必须在合适位置使用锁 (`e1000_lock`) 保护对环、mbuf 数组和寄存器的并发访问 (因为既可被系统调用路径访问，也可被中断处理访问)。

```
e1000_transmit():
```

```
int e1000_transmit(struct e1000 *dev, struct mbuf *m) {
```

```
    acquire(&dev->tx_lock);
```

```
    // 1. 读硬件 TDT (tail index)
```

```
    int tdt = regs[E1000_TDT];    // 或者 readl(&regs[E1000_TDT])
```

```

int i = tdt;                                // 描述符索引

// 2. 检查 descriptor 是否可用: status 要有 DD (descriptor done)

if (!(dev->tx_ring[i].status & E1000_TXD_STAT_DD)) {

    release(&dev->tx_lock);

    return -1; // 无可用描述符, caller 自行 free mbuf
}

// 3. 如果该描述符上之前保存了旧 mbuf, 先释放它

if (dev->tx_mbuf[i]) {

    mbuffree(dev->tx_mbuf[i]);

    dev->tx_mbuf[i] = 0;

}

// 4. 填写描述符: addr, length, cmd 标志 (EOP, RS 等)

dev->tx_ring[i].addr = (uint64) m->head;

dev->tx_ring[i].length = m->len;

dev->tx_ring[i].cmd = <必要命令位>; // 参见手册: EOP|RS 等

dev->tx_ring[i].status = 0; // 清状态 (硬件会在发送后写入 DD)

```

```

// 保存 mbuf 指针以便以后释放

dev->tx_mbuf[i] = m;


// 5. 更新 TDT 告知硬件

regs[E1000_TDT] = (i + 1) % TX_RING_SIZE;


release(&dev->tx_lock);

return 0;

}

e1000_recv() : void e1000_recv(struct e1000 *dev) {

    acquire(&dev->rx_lock);


    // 1. 从硬件读 RDT; 下一个要检查的 index = (regs[E1000_RDT] + 1) %
RX_RING_SIZE

    int rdt = regs[E1000_RDT];

    int i = (rdt + 1) % RX_RING_SIZE;


    // 2. 循环处理所有已完成的描述符 (status & DD)

    while (dev->rx_ring[i].status & E1000_RXD_STAT_DD) {

        // 获取 mbuf 指针已在初始化时保存在 dev->rx_mbuf[i]

        struct mbuf *m = dev->rx_mbuf[i];

```

```

m->len = dev->rx_ring[i].length;    // 描述符里报告的长度


// 交给上层网络栈处理

net_rx(m);


// 为该 descriptor 分配新的 mbuf 替代刚刚交出的那个

struct mbuf *m2 = mbufalloc();

if (m2 == 0) {

    // 无内存可用：丢弃（或做其他处理），并清 descriptor 指针以防硬件写入未初始化内存

    dev->rx_ring[i].addr = 0;

} else {

    dev->rx_ring[i].addr = (uint64) m2->head;

    dev->rx_mbuf[i] = m2;

}


// 清状态位以便硬件再次写入（status=0）

dev->rx_ring[i].status = 0;


// 更新索引 i、并写回 RDT（最后处理的索引）

rdt = i;

```

```

regs[E1000_RDT] = rdt;

i = (i + 1) % RX_RING_SIZE;

}

release(&dev->rx_lock);

}

```

`e1000_init()` 提供了环及描述符的初始化参考（会为 RX 描述符分配 mbuf 并写入 `addr`）。实现时参考它的初始化风格保持一致。驱动应该注册中断处理函数（已有框架），在中断处理里调用 `e1000_recv()` 并做必要的调度/唤醒操作。发送路径在成功填描述符后通常不需要等待中断立即回调；但 TX 描述符的释放（mbuf 的回收）会出现在后续某次中断或在发送路径再次检查时（取决于硬件行为），因此在 `e1000_transmit()` 中我们也主动释放之前完成的 mbuf。

驱动可能在以下两种上下文并发访问相同数据结构：系统调用上下文（用户进程调用发送）和中断上下文（接收中断）；因此必须用内核锁（spinlock）保护 `tx_ring`、`tx_mbuf`、`rx_ring`、`rx_mbuf`、以及对 `regs` 的某些并发写（如果存在 race）。在中断处理里要尽量做少量工作（例如只把 mbuf 交给网络栈或把任务放到队列里并在软中断/工作线程里完成更多处理）。

B. 构建并运行

在一个窗口的终端中执行 `make server`，打开另一个窗口利用 `make qemu` 指令运行 `xv6`，运行 `nettests` 以测试数据包的发送和接收功能。

```
goost@goost: ~/xv6-labs-2021-7
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ nettests
nettests running on port 26099
testing ping: OK
testing single-process pings: OK
testing multi-process pings: OK
testing DNS
DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
DNS OK
all tests passed.
$
```

```
goost@goost: ~/xv6-labs-2021-7
goost@goost:~/xv6-labs-2021-7$ make server
python3 server.py 26099
listening on localhost port 26099
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
```

8.1.3. 实验中遇到的问题和解决办法

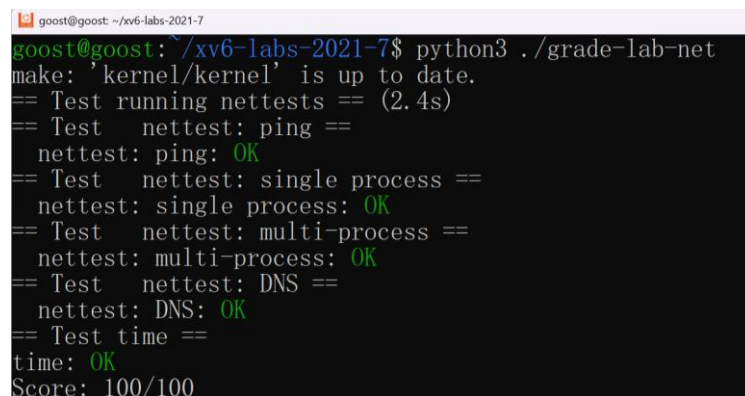
- A. TX 描述符被覆盖 / 发送的包乱序或丢失，向 host 发包但 host 没收到或收到了错误的报文；packets.pcap 显示数据异常。没有检查 descriptor 的 DD 标志就复用描述符，或者在写描述符之前就更新了 TDT。解决：在使用某个 TX 描述符前，先检查 status & E1000_TXD_STAT_DD。只有确认硬件完成之前的发送才能复用该描述符；在填完所有描述符字段并保存 mbuf 指针后再写 regs[E1000_TDT]。
- B. 接收环（RX）描述符回收错误，导致后续 DMA 写入非法地址 / 内核 panic，收到中断但驱动在访问 rx 描述符或 mbuf 时出现 crash；tcpdump 显示接收后环状行为异常。交付给 net_rx() 后没有为该描述符分配并写回一个新的有效 mbuf 地址，或没有清除 status 位；硬件再次到达该 descriptor 时会写入被释放或 NULL 的地址。解决：在交付 mbuf 后立即为该 descriptor 分配新的 mbuf (mbufalloc())，把 m2->head 写到 descriptor 的 addr 字段，并把 descriptor status 清零；最后更新 E1000_RDT。在内存分配失败时，选择丢弃该接收并保持 descriptor 无效以避免写入未映射内存。

8.1.4. 实验心得

掌握发送与接收环的索引管理、状态位含义、以及与硬件寄存器的交互是实现驱动的关键。务必始终保证“写描述符 → 保存 mbuf 指针 → 更新寄存器（TDT/RDT）”这条顺序不被破坏。驱动既可能在系统调用上下文被访问，也在中断上下文运行，缺少锁保护会导致难以复现的崩溃或数据损坏。选用合适粒度的锁（tx/rx 分离）能兼顾性能与正确性。

8.2. Lab7 成绩

增加 time.txt 文件输入完成实验的时长。输入 ./grade-lab-thread 查看实验七的得分。

A terminal window with a black background and green text. The prompt is 'goost@goost: ~/xv6-labs-2021-7'. The user has entered 'python3 ./grade-lab-net'. The output shows the kernel is up to date, followed by a series of tests: 'Test running nettests == (2.4s)', 'Test nettest: ping ==', 'nettest: ping: OK', 'Test nettest: single process ==', 'nettest: single process: OK', 'Test nettest: multi-process ==', 'nettest: multi-process: OK', 'Test nettest: DNS ==', 'nettest: DNS: OK', 'Test time ==', 'time: OK', and finally 'Score: 100/100'.

```
goost@goost: ~/xv6-labs-2021-7$ python3 ./grade-lab-net
make: 'kernel/kernel' is up to date.
== Test running nettests == (2.4s)
== Test nettest: ping ==
nettest: ping: OK
== Test nettest: single process ==
nettest: single process: OK
== Test nettest: multi-process ==
nettest: multi-process: OK
== Test nettest: DNS ==
nettest: DNS: OK
== Test time ==
time: OK
Score: 100/100
```

9. Locks

9.1. Memory allocator

9.1.1. 实验目的

分析并解决 xv6 kalloc()/kfree() 的锁争用问题（kmem.lock 成为多核瓶颈），提高多核并行性。通过把单一全局空闲链表改造为 per-CPU freelist（每核一个空闲链表 + 一个锁）并实现**窃取（steal）**机制，使不同 CPU 的分配/释放尽量并行执行，从而大幅降低 kalloc_test 报告的锁自旋失败次数（#fetch-and-add）。掌握在内核中正确使用 cpuid()、push_off()/pop_off()、以及自旋锁的设计与命名要求（新锁名称须以 “kmem” 开头以通过测试统计识别）。

9.1.2. 实验步骤

A. 实现程序主体

使用数组 `struct kmem cpu_kmem[NCPU];`，每个元素包含一个自旋锁和一个指向该 CPU 空闲链表头的指针 (`struct run *freelist`)。`kalloc()` 首先在当前 CPU 的 `freelist` 上尝试分配；若为空则尝试从其它 CPU 的 `freelist` “窃取”一部分页（把对方 `list` 切分一半），最后若仍空则返回 0（或根据实现尝试其他回退策略）。`kfree()` 总是把页放回当前 CPU 的 `freelist`（这样释放发生在使用者所在 CPU 本地，减少跨核争用）。所有新加锁在 `initlock()` 时使用以 “`kmem`” 起头的名字，例如 “`kmem0`”，“`kmem1`” 或简单 “`kmem`” 加索引后缀，满足测试对锁名的检查要求。必须在使用 `cpuid()` 前调用 `push_off()`（关闭中断）并在使用后调用 `pop_off()`，防止被中断打断导致 `cpuid` 值不准确。

```
// kernel/param.h 定义 NCPU

struct kmem {

    struct spinlock lock;    // 名称要以 "kmem" 起头: initlock(&kmem[i].lock,
                             "kmem%d");

    struct run *freelist;    // 本 CPU 的空闲页链表

};

static struct kmem cpu_kmem[NCPU];
```

在系统启动初始化内存时 (`freerange()` 被某个 CPU 调用)，把该段内存全部加入调用者 CPU 的 `freelist`。对每个 `cpu_kmem[i]` 调用 `initlock(&cpu_kmem[i].lock, name)`，`name` 形式如 “`kmem%d`”。

```
void push_off();

int cid = cpuid(); // safe because interrupts off
```

```

// try local list

acquire(&cpu_kmem[cid].lock);

r = cpu_kmem[cid].freelist;

if (r) {

    cpu_kmem[cid].freelist = r->next;

    release(&cpu_kmem[cid].lock);

    pop_off();

    return (char*)r;

}

release(&cpu_kmem[cid].lock);


// 本核空，尝试 steal

for (int i = 1; i < NCPU; ++i) {

    int other = (cid + i) % NCPU;

    acquire(&cpu_kmem[other].lock);

    if (cpu_kmem[other].freelist) {

        // 从 other 的链表切出一半到本核

        struct run *head = cpu_kmem[other].freelist;

        struct run *mid = split_list_in_half(head);

        cpu_kmem[other].freelist = mid_next; // 剩下部分
    }
}

```

```

// 将 head..mid 添加入本核

release(&cpu_kmem[other].lock);

acquire(&cpu_kmem[cid].lock);

add_list_to_front(&cpu_kmem[cid].freelist, head);

// now take one

r = cpu_kmem[cid].freelist;

cpu_kmem[cid].freelist = r->next;

release(&cpu_kmem[cid].lock);

pop_off();

return (char*)r;

}

release(&cpu_kmem[other].lock);

}

```

// 如果所有核都空，返回 0

```
pop_off();
```

```
return 0;
```

split_list_in_half 的实现：遍历链表用快慢指针找到中点，将链表分为两半，返回中点后的头（给 other 保留），前半给 thief。

```
void kfree(char *v) {
```

```
    if (v == 0 || badaddr) panic();
```

```
struct run *r = (struct run*)v;

// clear memory for debug

memset(v, 1, PGSIZE);


push_off();

int cid = cpuid();

acquire(&cpu_kmem[cid].lock);

r->next = cpu_kmem[cid].freelist;

cpu_kmem[cid].freelist = r;

release(&cpu_kmem[cid].lock);

pop_off();

}
```

B. 构建并运行

在终端中执行 `make qemu` 编译，运行 `kalloctest` 测试，观察到 `acquire` 的循环迭代次数明显减少，表明每个 CPU 现在有了自己的 `freelist`，减少了 CPU 之间在访问内存分配器时的竞争。修改后的锁竞争情况有所改善。

```
goost@goost: ~/xv6-labs-2021-8
$ kalloc_test
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 65704
lock: kmem: #test-and-set 0 #acquire() 186649
lock: kmem: #test-and-set 0 #acquire() 180692
lock: bcache: #test-and-set 0 #acquire() 21
lock: bcache.bucket: #test-and-set 0 #acquire() 21
lock: bcache.bucket: #test-and-set 0 #acquire() 2
lock: bcache.bucket: #test-and-set 0 #acquire() 7
lock: bcache.bucket: #test-and-set 0 #acquire() 5
lock: bcache.bucket: #test-and-set 0 #acquire() 10
lock: bcache.bucket: #test-and-set 0 #acquire() 7
lock: bcache.bucket: #test-and-set 0 #acquire() 51
lock: bcache.bucket: #test-and-set 0 #acquire() 36
lock: bcache.bucket: #test-and-set 0 #acquire() 525
lock: bcache.bucket: #test-and-set 0 #acquire() 4
lock: bcache.bucket: #test-and-set 0 #acquire() 2
lock: bcache.bucket: #test-and-set 0 #acquire() 2
--- top 5 contended locks:
lock: proc: #test-and-set 256779 #acquire() 1208608
lock: proc: #test-and-set 250083 #acquire() 1208569
lock: proc: #test-and-set 246012 #acquire() 1208603
lock: proc: #test-and-set 241350 #acquire() 1208568
lock: proc: #test-and-set 236541 #acquire() 1208569
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.
```

运行 `usertests sbrkmuch` 测试，确保内存分配器的正确性：

```
goost@goost: ~/xv6-labs-2021-8
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$
```

9.1.3. 实验中遇到的问题和解决办法

- A. `cpuid()` 的结果不稳（导致把页放到了错误的 `cpu freelist`），分配/释放记录落在错误的 `freelist`，上下文切换时出现严重的不平衡，最终 `kalloc_test` 仍然高争用或内存耗尽。在调用 `cpuid()` 时没有关闭中断，导致当前线程在获取 `cpuid()` 后被抢占到另一个核继续执行，从而 `cpuid()` 的返回与随后操作不一致。解决：在 `kalloc()/kfree()` 中在读

取 `cpuid()` 前调用 `push_off()`（关闭中断），并在完成后调用 `pop_off()`（恢复中断）。这样保证 `cpuid()` 与随后操作都在同一核上执行。

- B. 窃取实现不当导致死锁或高争用，窃取（steal）代码中同时持有两个 CPU 的锁（按不同顺序）会在并发下死锁；或者窃取过于频繁/粒度太小导致反而增加争用开销。不统一的锁获取顺序或在持锁期间做太多工作导致长时间占用锁。另一个常见问题是把“把整条链表直接移走”作为窃取策略，会让被窃取 CPU 彻底没有空闲页，反复窃取操作频繁发生。解决：统一窃取步骤：只在 other 的锁下切分一半链表（O(1) 修改 pointer），然后立刻释放 other 锁，再去持 local 锁把切下来的那半链表链接到 local。这样避免在持有两个锁时进行长操作，并且通过“切一半”降低了窃取频率。避免获取多个锁的复杂交叉：仅在 other 上做切分操作（持其锁），再短时间持本核锁把切分来的块并入本核，两个锁的持有时间都很短，减少死锁可能。确保窃取只在本核空闲时触发（并且按环形顺序尝试其它核），降低竞争。

9.1.4. 实验心得

单一全局锁在多核环境下是典型的性能瓶颈。把全局数据结构拆分为 per-CPU 或分段（sharding）数据结构并配合窃取/平衡机制，是常见且有效的缩减争用策略。本实验把 free pages 按 CPU 本地化，绝大多数 `kalloc()/kfree()` 操作都能在本核就完成，从而把自旋失败次数显著降低。在内核中一定要小心使用 `cpuid()`：必须在 `push_off()/pop_off()` 范围内使用，避免被中断或迁移到别的核。窃取策略要权衡，切分大小（切一半 vs 切固定数量）影响争用频率与负载均衡质量；切一半通常是个简单而稳妥的折中。锁的命名/可观测性也很重要：自动化测试依赖锁名做统计，遵循命名约定可避免不必要的尴尬失败。

9.2. Buffer cache

9.2.1. 实验目的

降低 xv6 块缓存（block cache，位于 `kernel/bio.c`）在多核并发读写时的锁争用，使 `bcachetest` 报告的锁自旋失败次数显著下降（理想情况 sum 接近 0，允许小量误差）。在保证语义正确（每个磁盘块在缓存中最多只有一

份)的前提下,重设计数据结构与锁策略:将全局串行的 `bcache.lock` 拆分为多个桶(bucket)锁或其它细粒度并发结构,从而实现不同块并发访问时尽量不互相阻塞。学会处理缓存替换(eviction)与桶之间移动时可能出现的并发问题与死锁,并在开发过程中逐步验证与回退策略(先用全局锁线性化验证正确性,再移除全局锁以获得并发性)。

9.2.2. 实验步骤

A. 实现程序主体

用哈希桶把缓冲区按 block number 分散到多个链表上。每个 bucket 维护自己的链表与一把锁(`struct spinlock bucket_lock[]`)。查找某个 block 的 buf 时只需抓取对应 bucket 的锁,从而不同 block(哈希到不同桶)不会争用同一锁。`bget()` 的查找路径:在 bucket 锁下查找;命中则增加 `b->refcnt` 并返回(释放锁或在锁内返回,视实现决定)。当缓存未命中时,需要选择一个可回收(`unused`、`refcnt==0`)的 buf 来替换,被替换的 buf 可能位于另一个 bucket(旧 blockno 对应的桶);替换过程要小心锁顺序以避免死锁。为释放(`brelse()`)减少争用:不再持全局 `bcache.lock` 去更新时间顺序或链表头尾;而是更新 buf 的 `lastuse`(使用 ticks)并在对应 bucket 下做最小修改,这样 `brelse` 对不同块也能并行化。把淘汰(eviction)序列化:设计一个单独的 `evict_lock`(名字以“`bcache`”起头,如“`bcache.evict`”),在 eviction 阶段持此锁以简化替换逻辑——即只有一个线程在做寻找并回收 victim 的全局动作,从而避免在复杂移动时需要同时持有很多 bucket 锁而引起死锁。

```
#define NBUCKET 13 // 质数,减少冲突(可根据 buf 数量调整)
```

```
struct bucket {  
  
    struct spinlock lock;        // 名称要以 "bcache" 起头  
  
    struct buf *head;           // 此 bucket 的链表  
  
};
```



```
static struct bucket bcache_buckets[NBUCKET];
```

```
static struct spinlock evict_lock; // 名称 "bcache.evict"
```

在 struct buf 中加入：

```
struct buf {
```

```
    uint lastuse;    // 用 ticks 记录最后一次使用时间
```

```
    struct buf *bnext; // 在 bucket 链上的指针（替代旧的全局链表指针）
```

```
    int bucketno;    // 当前在第几个 bucket（便于移动时知道原桶）
```

```
};
```

```
bget() :
```

```
struct buf* bget(dev, blockno) {
```

```
    int bno = bucket_for(blockno);
```

```
    acquire(&bcache_buckets[bno].lock);
```

```
    // 1. 在 bucket 链表中查找
```

```
    for (b = bucket.head; b; b = b->bnext) {
```

```
        if (b->dev==dev && b->blockno==blockno) {
```

```
            b->refcnt++;
```

```
            b->lastuse = ticks; // 更新使用时间
```

```
            release(&bcache_buckets[bno].lock);
```

```
            return b;
```

```
        }
```

```

}

// miss: need to allocate a buf for this block

// To avoid complicated cross-bucket juggling, use eviction lock

release(&bcache_buckets[bno].lock);

acquire(&evict_lock);

// find victim: search buckets for a buf with refcnt==0 and smallest lastuse

victim = choose_victim(); // 持有 evict_lock 时进行（内部可能要短暂持有
bucket 锁来测试 refcnt/移除）

if (victim == NULL) { release(&evict_lock); return 0; }

// remove victim from its bucket

int oldb = victim->bucketno;

acquire(&bcache_buckets[oldb].lock);

remove_from_bucket(victim, &bcache_buckets[oldb]);

release(&bcache_buckets[oldb].lock);

// initialize victim for new block

victim->dev = dev; victim->blockno = blockno;

victim->flags = B_BUSY; victim->refcnt = 1;

victim->lastuse = ticks;

```

```

// insert victim into target bucket

acquire(&bcache_buckets[bno].lock);

add_to_bucket_head(victim, &bcache_buckets[bno]);

victim->bucketno = bno;

release(&bcache_buckets[bno].lock);


release(&evict_lock);

return victim;

}

```

brelse() :

```

void brelse(struct buf *b) {

    int bno = b->bucketno;

    acquire(&bcache_buckets[bno].lock);

    b->refcnt--;

    b->lastuse = ticks;

    // 如果 refcnt == 0, 不需要做更多（不会移动），只是留下供其它人重用

    release(&bcache_buckets[bno].lock);

}

```

在 binit() 中为每个 bucket 调用 initlock(&bucket.lock, "bcache.bucket")（或 "bcache.bucket%d"），并初始化 evict_lock 名称为 "bcache.evict"。

建议桶数为质数（如 13 或 17），且不要过小以避免冲突，也不要过大以免内存碎片或管理开销。

B. 构建并运行

在终端中执行 `make qemu` 编译, 运行 `bcachetest` 测试, 观察到 `test-and-set` 的操作和锁获取次数明显减少, 表明并发访问缓冲区池时存在的竞争减少, 改进方案有效提升了系统的并发性能。

```
goost@goost: ~/xv6-labs-2021-8
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 86736
lock: kmem: #test-and-set 0 #acquire() 97597
lock: kmem: #test-and-set 0 #acquire() 96692
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: bcache: #test-and-set 0 #acquire() 439
lock: bcache.bucket: #test-and-set 0 #acquire() 2588
lock: bcache.bucket: #test-and-set 0 #acquire() 3487
lock: bcache.bucket: #test-and-set 0 #acquire() 3616
lock: bcache.bucket: #test-and-set 0 #acquire() 3528
lock: bcache.bucket: #test-and-set 9 #acquire() 3453
lock: bcache.bucket: #test-and-set 0 #acquire() 3353
lock: bcache.bucket: #test-and-set 0 #acquire() 3446
lock: bcache.bucket: #test-and-set 0 #acquire() 2375
lock: bcache.bucket: #test-and-set 0 #acquire() 2866
lock: bcache.bucket: #test-and-set 0 #acquire() 1074
lock: bcache.bucket: #test-and-set 0 #acquire() 2068
lock: bcache.bucket: #test-and-set 0 #acquire() 1065
lock: bcache.bucket: #test-and-set 0 #acquire() 2068
--- top 5 contended locks:
lock: proc: #test-and-set 16963248 #acquire() 3600555
lock: proc: #test-and-set 815740 #acquire() 3618071
lock: virtio_disk: #test-and-set 714331 #acquire() 2197
lock: proc: #test-and-set 688209 #acquire() 3685880
lock: proc: #test-and-set 657801 #acquire() 3685934
tot= 9
test0: OK
start test1
test1 OK
```

运行 `usertests` 测试, 测试结果表明, 系统其他功能一切正常。

```
goost@goost: ~/xv6-labs-2021-8
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitinput: OK
test input: OK
test mem: OK
test pipel: OK
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

9.2.3. 实验中遇到的问题和解决办法

- A. 直接为每个 buf 加锁（粒度太细）导致复杂的死锁情形，在试图在替换时同时取得 victim 的原 bucket 锁与目标 bucket 锁时出现死锁（两个线程互相等待对方释放对方持有的桶锁）。解决：采用单独的 eviction 锁（bcache.evict）序列化 victim 的选择与移动；choose_victim() 为每个桶短时获取该桶锁去查看/refcnt/移除（不长时间持锁），而不是一次性同时持有多个桶锁。另一种可选做法是在需要同时持有多个锁时强制统一获取顺序（例如先锁编号小的桶），但序列化 eviction 更简单可靠。
- B. brelse 更新 lastuse 时的竞态导致 LRU 信息不一致，在没有合适同步的情况下，多个 CPU 并发更新同一个 buf 的 lastuse（或读取时读取到旧值），导致 eviction 选择错误（回收仍在使用的块）。解决：把 lastuse 的更新放在 bucket 锁保护下（acquire(bucket.lock) -> update lastuse -> release），或者使用原子写（但仍需小范围锁保护 refcnt 与 lastuse 的复合关系）。这样 brelse 仅在对应桶上短时加锁，保证 lastuse 与 refcnt 的一致性。

9.2.4. 实验心得

把单一全局锁分成桶锁通常能显著降低并发争用，但实现会引入移动、替换时的复杂并发问题。把复杂操作（如 eviction）序列化是一个务实的折中：既能保证正确性，又能把常见路径的并行性能最大化。开发流程里先保持全局

锁以验证功能正确性，再逐步移除全局锁并引入细粒度锁与序列化 eviction。这样可以快速定位并发缺陷。锁命名与可观察性很重要：自动化测试依赖锁名统计争用（本实验要求锁名以“bcache”开头），遵循命名约定能避免“实现正确但被判定为失败”的尴尬。在单核（make CPUS=1 qemu）下先保证功能正确性，再切换到多核进行并发测试；为关键位置加 cprintf 日志（并小心不要在中断上下文打印太多），用逐步解除全局锁的方法缩小问题。bcachetest 给出的锁统计是最直接有价值的反馈。这次实验让我体会到工程实践中常见的权衡：在多数路径（查找/释放）使用细粒度锁以获取并行吞吐；而把少见且复杂的路径（miss+evict）序列化以保正确性和实现简单性。这样的权衡在实际生产系统也很常见。

9.3. Lab8 成绩

增加 time.txt 文件输入完成实验的时长。输入 ./grade-lab-lock 查看实验八的得分。

```
goost@goost: ~/xv6-labs-2021-8
goost@goost:~/xv6-labs-2021-8$ python3 ./grade-lab-lock
make: 'kernel/kernel' is up to date.
== Test running kallocetest == (45.1s)
== Test kallocetest: test1 ==
kallocetest: test1: OK
== Test kallocetest: test2 ==
kallocetest: test2: OK
== Test kallocetest: sbrkmuch == kallocetest: sbrkmuch: OK (7.1s)
== Test running bcachetest == (10.7s)
== Test bcachetest: test0 ==
bcachetest: test0: OK
== Test bcachetest: test1 ==
bcachetest: test1: OK
== Test usertests == usertests: OK (144.3s)
== Test time ==
time: OK
Score: 70/70
goost@goost:~/xv6-labs-2021-8$
```

10. file system

10.1. Large files

10.1.1. 实验目的

本实验的目标是扩展 xv6 文件系统，使其能够支持更大的文件。默认情况下，xv6 文件最大仅能容纳 268 个数据块（12 个直接块 + 1 个间接块），约 268 KB。实验要求在 xv6 文件系统中为 inode 增加一个 双重间接块，从而将最大文件大小扩展到约 65,803 个数据块。通过修改 `bmap()` 实现逻辑块号到磁盘块号的映射，使得大文件能够被正常创建、写入与读取。

10.1.2. 实验步骤

A. 实现程序主体

修改 inode 结构，打开 `kernel/fs.h` 文件，查找 `struct dinode` 结构的定义。该结构描述了 inode 在磁盘上的格式。查找并修改 `NDIRECT` 和 `NINDIRECT` 的定义。

修改 `struct dinode` 和 `struct inode`。更新 `addrs` 数组的大小，以支持新的块结构。

```
struct dinode {  
  
    ... uint addrs[NDIRECT+2];  
  
};  
  
struct inode {  
  
    ... uint addrs[NDIRECT+2];  
  
};
```

更新 `bmap` 函数，打开 `kernel/fs.c` 文件中的 `bmap()` 函数并修改。计算逻辑块号：从逻辑块号中去除已经由直接块和单间接块映射的块数。分配并定位双间接块：根据剩余的逻辑块号，分配双间接块、单间接块，并定位实际的数据块。处理块号映射：对于逻辑块号大于等于 `NINDIRECT` 的情况，使用双间接块进行映射。具体过程如下：首先映射到双间接块。然后映射到单间接块。最后映射到实际的数据块。

```
static uint bmap(struct inode *ip, uint bn)
```

```

{

uint addr, *a;

struct buf *bp;

if(bn < NDIRECT){

if((addr = ip->addrs[bn]) == 0)

ip->addrs[bn] = addr = balloc(ip->dev);

return addr;

}

bn -= NDIRECT;

if(bn < NINDIRECT){

// Load indirect block, allocating if necessary. if((addr = ip->addrs[NDIRECT]) == 0)

ip->addrs[NDIRECT] = addr = balloc(ip->dev);

bp = bread(ip->dev, addr);

a = (uint*)bp->data;

if((addr = a[bn]) == 0){

a[bn] = addr = balloc(ip->dev);

log_write(bp);

}

brelse(bp);

return addr;

```



```

}

bn -= NINDIRECT;

// 去除已经由直接块和单间接块映射的块数，以得到在双间接块中的相对
块号

if (bn < NDBL_INDIRECT) {

// 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[index1] = 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块

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[index2] = addr;

log_write(bp); // Record changes in the log

}

brelse(bp);
```

```

return addr; // Returns the actual data block

}

panic("bmap: out of range");

}

```

确保块的释放，在 kernel/fs.c 的 itrunc 函数中，添加对双层间接映射的清除逻辑，确保释放双层映射的数据块释放。双间接块：读取双间接块，遍历其中的每个单间接块。释放单间接块：对于每个单间接块，读取并释放其中的数据块。释放直接块：在完成双间接块和单间接块的释放后，释放直接块。

```

void

itrunc(struct inode *ip)

{

int i, j;

struct buf *bp;

uint *a;

// 遍历并释放直接块

for(i = 0; i < NDIRECT; i++){

if(ip->addrs[i]){

bfree(ip->dev, ip->addrs[i]); // 释放数据块

ip->addrs[i] = 0; // 将指针置为 0

}

}

// 处理单间接块

```

```

if(ip->addr[NDIRECT]){

bp = bread(ip->dev, ip->addr[NDIRECT]); // 读取单间接块

a = (uint*)bp->data;

for(j = 0; j < NINDIRECT; j++){

if(a[j])

bfree(ip->dev, a[j]); // 释放间接块中的数据块

}

brelse(bp); // 释放缓冲区

bfree(ip->dev, ip->addr[NDIRECT]); // 释放单间接块

ip->addr[NDIRECT] = 0; // 将指针置为 0

}

// 处理双间接块

if (ip->addr[NDIRECT + 1]) {

bp = bread(ip->dev, ip->addr[NDIRECT + 1]); // 读取双间接块

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); // 释放缓冲区

bfree(ip->dev, a[i]); // 释放单间接块

a[i] = 0; // 将指针置为 0

}

brelse(bp); // 释放缓冲区

bfree(ip->dev, ip->addrs[NDIRECT + 1]); // 释放双间接块

ip->addrs[NDIRECT + 1] = 0; // 将指针置为 0

}

// 将文件大小设为 0

ip->size = 0;

iupdate(ip); // 更新 inode 信息

}

```

B. 构建并运行

在终端中执行 `make qemu` 编译并运行 `xv6`。在命令行中输入 `bigfile`，产生结果如下：

```
goost@goost: ~/xv6-labs-2021-9
xv6 kernel is booting

init: starting sh
$ bigfile
.....
.....
.....
.....
.....
wrote 65803 blocks
bigfile done; ok
$
```

10.1.3. 实验中遇到的问题和解决办法

- A. 修改 NDIRECT 后编译失败, 因为 struct inode 和 struct dinode 中的 `addrs[]` 数组长度必须保持一致, 否则会导致数据结构不匹配。解决: 同步修改 inode 和 dinode, 保持 `addrs[]` 的一致性。
- B. 大文件写入出错, `bigfile` 失败, 因为在双重间接块处理时, 没有正确分配和释放二级间接块。解决: 在 `bmap()` 中增加严格的分配逻辑, 确保 `brelease()` 每次都被正确调用, 避免磁盘块泄漏。

10.1.4. 实验心得

通过本实验, 我深入理解了 文件系统 中的块映射机制。原本 xv6 仅支持 直接块 + 单重间接块 的映射, 限制了文件大小; 而通过增加 双重间接块, 系统可以扩展支持到更大规模的文件。在实现过程中, 最大的收获是掌握了 逻辑块号到磁盘块号的层级映射 思路, 同时理解了 `bmap()` 如何在读写过程中动态分配新块。实验让我更加清晰地认识到文件系统 设计中的 扩展性与复杂度 权衡。

10.2. Symbolic links

10.2.1. 实验目的

本实验的目标是为 xv6 文件系统实现 符号链接 (symbolic link) 功能。符号链接是一种特殊的文件, 它包含了另一个文件的路径名。在访问符号链接时, 内核会解析该路径并重定向到真正的目标文件。通过实现符号链接, 可以更好地支持类 UNIX 文件系统中的灵活文件访问机制。

10.2.2. 实验步骤

A. 实现程序主体

在 kernel/syscall.h 中定义新的系统调用号#define SYS_symlink 22

在 kernel/syscall.c 中声明系统调用 extern uint64
SYS_symlink(void);

更新系统调用表以包含新的 symlink 系统调用。在 user/usys.pl 中添加系统调用条目 entry("symlink");

在 user/user.h 中声明 symlink 函数 int symlink(char*, char*);

在 kernel/fcntl.h 中添加新的打开标志 O_NOFOLLOW ,

```
#define O_NOFOLLOW 0x004
```

在 Makefile 中添加 symlinktest.c 的编译

在 kernel/sysfile.c 中实现 sys_symlink 函数，用于创建符号链接

```
int sys_symlink(void) {  
  
    char *path, *target;  
  
    struct inode *ip;  
  
    int n;  
  
    // 获取系统调用参数，路径和目标路径  
  
    if (argstr(0, &path) < 0 || argstr(1, &target) < 0)  
  
    return -1; // 获取参数失败，返回错误码 -1  
  
    begin_op(); // 开始一个文件系统操作  
  
    // 创建一个符号链接类型的 inode
```

```

if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {

end_op(); // 结束文件系统操作

return -1; // 创建失败，返回错误码 -1

}

// 计算目标路径的长度，并准备写入符号链接的内容

n = strlen(target) + 1;

// 将目标路径写入符号链接的 inode 中

if(writei(ip, 0, (uint64)target, 0, n) != n) {

iunlockput(ip); // 解锁并释放 inode

end_op(); // 结束文件系统操作

return -1; // 写入失败，返回错误码 -1

}

iunlockput(ip); // 解锁并释放 inode

end_op(); // 结束文件系统操作

return 0; // 成功创建符号链接，返回 0

}

```

在 kernel/sysfile.c 中，修改 sys_open 函数以处理符号链接：

```

int sys_open(void) {

char *path;

int fd, flags;

```



```

// 获取系统调用参数，路径和标志

if (argstr(0, &path) < 0 || argint(1, &flags) < 0)

return -1; // 获取参数失败，返回错误码 -1

struct file *f = 0;

struct inode *ip;

begin_op(); // 开始一个文件系统操作

// 根据路径名查找对应的 inode

ip = namei(path);

if (ip == 0) {

end_op(); // 结束文件系统操作

return -1; // 查找失败，返回错误码 -1

}

// 处理符号链接

if (ip->type == T_SYMLINK) {

// 如果标志中包含 O_NOFOLLOW，说明不希望跟随符号链接

if (flags & O_NOFOLLOW) {

iunlockput(ip); // 解锁并释放 inode

end_op(); // 结束文件系统操作

return -1; // 不允许跟随符号链接，返回错误码 -1

}

```

```

// 解析符号链接，获取符号链接指向的实际 inode

ip = follow_symlink(ip);

}

// 现有的文件打开逻辑

// 为新文件分配一个 file 结构体

f = filealloc();

if (f == 0 || (fd = fdalloc(f)) < 0) {

if (f)

fileclose(f); // 分配失败，关闭并释放 file 结构体

iunlockput(ip); // 解锁并释放 inode

end_op(); // 结束文件系统操作

return -1; // 文件分配或描述符分配失败，返回错误码 -1

}

f->type = FD_INODE; // 设置 file 类型

f->ip = ip; // 关联 inode

f->off = 0; // 初始偏移量为 0

f->readable = !(flags & O_WRONLY); // 设置可读性

f->writable = !(flags & O_RDONLY); // 设置可写性

iunlockput(ip); // 解锁并释放 inode

end_op(); // 结束文件系统操作

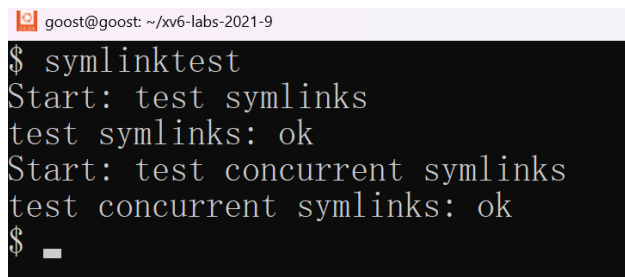
```

```
return fd; // 返回文件描述符号
```

B. 构建并运行

在终端中执行 `make qemu` 编译并运行 `xv6`。在命令行中输入 `symlinktest`,

产生结果如下:



```
goost@goost: ~/xv6-labs-2021-9
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$
```

10.2.3. 实验中遇到的问题和解决办法

- A. symlink inode 无法保存路径字符串, 没有在 inode 中分配数据块写入路径字符串。解决: 调用 `writei()` 将 `target` 字符串写入符号链接 inode 的数据区, 并在 `readi()` 时取出。
- B. 递归解析导致死循环, 因为符号链接指向自身或多个符号链接互相引用。解决: 在 `sys_open()` 解析符号链接时限制最大递归层数 (如 10), 超过则报错返回。

10.2.4. 实验心得

通过本实验, 我进一步理解了 文件系统抽象与路径解析机制。符号链接作为一种间接引用方式, 增强了文件系统的灵活性, 但同时也引入了 循环解析、权限控制 等潜在问题。

实验中最核心的体会是: 文件系统的 inode 不仅仅可以存储数据块映射, 还可以存储“元数据”, 通过不同的 inode 类型, 内核可以支持更多种类的文件抽象。

此外, 调试符号链接时必须特别小心路径解析中的递归逻辑, 否则极易出现死循环。

10.3. Lab9 成绩

增加 time.txt 文件输入完成实验的时长。输入 `./grade-lab-fs` 查看实验九的得分。

```
goost@goost: ~/xv6-labs-2021-9
goost@goost: ~/xv6-labs-2021-9$ python3 ./grade-lab-fs
make: 'kernel/kernel' is up to date.
== Test running bigfile == running bigfile: OK (92.7s)
== Test running symlinktest == (0.6s)
== Test  symlinktest: symlinks ==
    symlinktest: symlinks: OK
== Test  symlinktest: concurrent symlinks ==
    symlinktest: concurrent symlinks: OK
== Test usertests == usertests: OK (216.7s)
== Test time ==
time: OK
Score: 100/100
```

11. mmap

11.1. mmap

11.1.1. 实验目的

11.1.2.

11.1.3. 实验步骤

```
goost@goost: ~/xv6-labs-2021-10
hart 2 starting
hart 1 starting
init: starting sh
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded
$
```

11.1.4. 实验中遇到的问题和解决办法

11.1.5. 实验心得

11.2. Lab10 成绩

增加 time.txt 文件输入完成实验的时长。输入 `./grade-lab-mmap` 查看实验十的得分。

```
goost@goost: ~/xv6-labs-2021-10
Agoost@goost:~/xv6-labs-2021-10$ python3 ./grade-lab-mmap
make: 'kernel/kernel' is up to date.
== Test running mmaptest == (2.2s)
== Test  mmaptest: mmap f ==
mmaptest: mmap f: OK
== Test  mmaptest: mmap private ==
mmaptest: mmap private: OK
== Test  mmaptest: mmap read-only ==
mmaptest: mmap read-only: OK
== Test  mmaptest: mmap read/write ==
mmaptest: mmap read/write: OK
== Test  mmaptest: mmap dirty ==
mmaptest: mmap dirty: OK
== Test  mmaptest: not-mapped unmap ==
mmaptest: not-mapped unmap: OK
== Test  mmaptest: two files ==
mmaptest: two files: OK
== Test  mmaptest: fork test ==
mmaptest: fork_test: OK
== Test usertests == usertests: OK (99.2s)
== Test time ==
time: OK
Score: 140/140
```