

操作系统课程设计：xv6 及 Labs 课程项目

组号：39 学号：2253893 苗君文

目录

Tools Used in 6.S081	3
Lab1: Xv6 and Unix utilities	4
Boot xv6 (easy)	4
sleep (easy)	5
pingpong (easy)	8
primes (moderate)/(hard)	10
find (moderate)	13
xargs (moderate)	15
Lab2: system calls	17
System call tracing (moderate)	18
Sysinfo (moderate)	21
Lab3: page tables	24
Speed up system calls (easy)	24
Print a page table (easy)	27
Detecting which pages have been accessed (hard)	30
Lab4: traps	35
RISC-V assembly (easy)	35
Backtrace (moderate)	38
Alarm (hard)	42
Lab5: Copy-on-Write Fork for xv6	46
Implement copy-on write(hard)	46
Lab6: Multithreading	52
Uthread: switching between threads (moderate)	52
Using threads (moderate)	57
Barrier(moderate)	59
Lab7: networking	62

Lab8: locks	66
Memory allocator (moderate)	66
Buffer cache (hard)	71
Lab9: file system	79
Large files (moderate)	79
Symbolic links (moderate)	83
Lab10: mmap	87

仓库链接: <https://github.com/MEKSAAA/OS-xv6-labs-2021>

Tools Used in 6.S081

一、实验目的

在 Windows 系统上安装本课程需要的所有工具。

二、实验步骤

1) 安装 WSL

以管理员身份运行 cmd

输入 `wsl - install` 命令

2) 安装 Ubuntu

```
meksa@MEKSA: ~  
Installing, this may take a few minutes...  
Please create a default UNIX user account. The username does not need to match your Windows username.  
For more information visit: https://aka.ms/wslusers  
Enter new UNIX username: meksa  
New password:  
Retype new password:  
passwd: password updated successfully  
Installation successful!  
To run a command as administrator (user "root"), use "sudo <command>".  
See "man sudo_root" for details.  
  
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.153.1-microsoft-standard-WSL2 x86_64)  
  
* Documentation:  https://help.ubuntu.com  
* Management:    https://landscape.canonical.com  
* Support:        https://ubuntu.com/advantage  
  
This message is shown once a day. To disable it please create the  
/home/meksa/.hushlogin file.  
meksa@MEKSA: $
```

3) 运行以下代码以安装本课程所需的所有软件：

```
$ 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
```

4) 测试是否安装完成

`riscv64-unknown-elf-gcc --version`

`qemu-system-riscv64 - version`

`riscv64-linux-gnu-gcc - version`

`riscv64-linux-gnu-ld --version`

三、实验中遇到的问题和解决方法

在首次运行 Ubuntu 时，出现如下报错：

```
Installing, this may take a few minutes...  
WslRegisterDistribution failed with error: 0x8004032d  
Error: 0x8004032d (null)  
Press any key to continue...
```

通过询问 ChatGPT 得知是因为没有启用相应的 Windows 功能，解决方法如下：

- 打开“控制面板” > “程序” > “启用或关闭 Windows 功能”。
- 勾选“适用于 Linux 的 Windows 子系统”和“虚拟机平台”，然后点击“确定”。
- 重启计算机。

四、实验心得

根据 tools 的指示，使我初次接触 Linux 系统，并学会安装 Linux 子系统，安装 Ubuntu，也对 Linux 的命令行有了一定的了解和实操。

Lab1: Xv6 and Unix utilities

Boot xv6 (easy)

一、实验目的

本实验的目的是在本地编译并运行 xv6，实验主要覆盖环境设置、获取 xv6 源码、编译运行 xv6 及简单的系统操作。

二、实验步骤

将 xv6-labs-2021 代码克隆到本地，进入到相应的目录，切换到 util 分支。

```
meksa@MEKSA:~$ git clone git://g.csail.mit.edu/xv6-labs-2021
Cloning into 'xv6-labs-2021'...
remote: Enumerating objects: 7051, done.
remote: Counting objects: 100% (7051/7051), done.
remote: Compressing objects: 100% (3423/3423), done.
remote: Total 7051 (delta 3702), reused 6830 (delta 3600), pack-reused 0
Receiving objects: 100% (7051/7051), 17.20 MiB | 2.82 MiB/s, done.
Resolving deltas: 100% (3702/3702), done.
warning: remote HEAD refers to nonexistent ref, unable to checkout.

meksa@MEKSA:~$ cd xv6-labs-2021
meksa@MEKSA:~/xv6-labs-2021$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

在 xv6 目录中执行了 make qemu 命令，编译生成了 xv6 的可执行文件并启动 QEMU 虚拟机。

```
meksa@MEKSA:~/xv6-labs-2021$ make qemu
riscv64-unknown-elf-gcc -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -fno-stack-protector -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -fno-stack-protector -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -fno-stack-protector -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -fno-stack-protector -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -fno-stack-protector -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -c
```

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

运行 ls 命令，验证文件系统的正确性。

```
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
xargstest.sh 2 3 93
cat        2 4 24184
echo       2 5 23008
forktest   2 6 13232
grep       2 7 27488
init       2 8 23744
kill       2 9 22952
ln         2 10 22792
ls         2 11 26376
mkdir      2 12 23096
rm         2 13 23080
sh         2 14 41904
stressfs   2 15 23952
usertests  2 16 157000
grind      2 17 38120
wc         2 18 25280
zombie     2 19 22336
console    3 20 0
```

通过 Ctrl-a x 退出 QEMU 虚拟机

```
$ QEMU: Terminated
meksa@MEKSA: ~/xv6-labs-2021$
```

三、实验中遇到的问题和解决方法

无

四、实验心得

通过本次实验，我对操作系统的启动过程有了更深入的理解。特别是在设置编译环境和使用 QEMU 虚拟机方面，掌握了一些实用技巧。在获取和管理源码的过程中，进一步熟悉了 Git 的使用，包括分支管理和代码提交。

sleep (easy)

一、实验目的

本实验旨在为 xv6 操作系统实现一个基本的 UNIX 程序 sleep，该程序可以根据用户指定的时间间隔（以 ticks 为单位）暂停运行。通过该实验，将了解 xv6 系统调用的工作原理，掌握如何编写简单的用户程序，并学会使用 Makefile 编译和运行程序。

二、实验步骤

1. 在 user 目录下创建 sleep.c 文件，并实现 sleep 功能。

```

user > C sleep.c
1  #include "kernel/types.h"
2  #include "user/user.h"
3
4  int main(int argc, char *argv[]) {
5      if (argc != 2) {
6          fprintf(2, "Usage: sleep <ticks>\n");
7          exit(1);
8      }
9
10     int ticks = atoi(argv[1]);
11     sleep(ticks);
12
13     exit(0);
14 }

```

2. 添加 sleep 程序到 Makefile

```

UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $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. 编译并启动 xv6

```

meksa@MEKSA:~/xv6-labs-2021$ make qemu
riscv64-unknown-elf-gcc -Wall -Werror -O -fcommon -fno-common -nostdlib -mno-relax -march=rv64imac -mabi=lp64 -L/usr/riscv64-unknown-elf-ld -z max-page-size=4096 -o xv6.out.o user/_malloc.o

```

4. 测试 sleep 程序

```

$ sleep 10

```

5. 运行测试成功

```

meksa@MEKSA:~/xv6-labs-2021$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.2s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
meksa@MEKSA:~/xv6-labs-2021$

```

三、实验中遇到的问题和解决方法

1. 在尝试 git push 时不成功

```
meksa@MEKSA:~/xv6-labs-2021$ git push my-xv6 util:util
Username for 'https://github.com': meksamiao
Password for 'https://meksamiao@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com/MEKSAAA/OS-xv6-labs-2021.git/'
```

查阅资料后，发现需要使用 SSH 密钥进行认证，添加 SSH 密钥后，将远程仓库 my-xv6 的 URL 更新为 SSH 形式：

```
git remote set-url my-xv6 git@github.com:MEKSAAA/OS-xv6-labs-2021.git
```

尝试再次推送：

```
git push my-xv6 util:util
```

但仍然失败

通过命令 `ssh -T git@github.com` 检查 SSH 配置是否正确

```
mjw@MEKSA MINGW64 ~
$ ssh -T git@github.com

The authenticity of host 'github.com (20.205.243.166)' can't be established.
ED25519 key fingerprint is SHA256:+DiY3wvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? Host key verification failed.
```

通过询问 ChatGPT，得知这是因为首次连接 github 时未验证主机密钥引起的，需要确认主机密钥的真实性。手动添加主机密钥：

```
mjw@MEKSA MINGW64 ~
$ ssh-keyscan -t ed25519 github.com >> ~/.ssh/known_hosts
# github.com:22 SSH-2.0-babeld-64443082

mjw@MEKSA MINGW64 ~
$ ssh -T git@github.com
Hi MEKSAAA! You've successfully authenticated, but GitHub does not provide shell access.
```

成功使用 SSH 连接。

但是这仅仅是在 Windows 系统下可以成功连接，而在 Linux 虚拟机中仍然没有成功连接 SSH。于是我尝试将 Windows 的 .ssh 文件内容复制到 Linux 虚拟机中。尝试重新添加 ssh：

```
meksa@MEKSA:~/xv6-labs-2021$ ssh-add ~/.ssh/id_rsa
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                WARNING: UNPROTECTED PRIVATE KEY FILE!                @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0644 for '/home/meksa/.ssh/id_rsa' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
```

而出现当前私钥文件权限设置过于宽松的警告，按以下命令修改私钥文件权限并重新添加私钥，尝试连接 github，验证成功。

```

meksa@MEKSA:~/xv6-labs-2021$ chmod 600 ~/.ssh/id_rsa
meksa@MEKSA:~/xv6-labs-2021$ ssh-add ~/.ssh/id_rsa
Identity added: /home/meksa/.ssh/id_rsa (2994735278@qq.com)
meksa@MEKSA:~/xv6-labs-2021$ ssh -T git@github.com
Hi MEKSAAA! You've successfully authenticated, but GitHub does not provide shell access.
meksa@MEKSA:~/xv6-labs-2021$

```

2. make grade 失败

```

$ ./grade-lab-util sleep
exec ./grade-lab-util failed
$ make GRADEFLAGS=sleep grade
exec make failed
$

```

查阅网上信息后，得知应退出 qemu，直接在 branch 中运行命令即可。

四、实验心得

通过本次实验，我深入理解了 xv6 操作系统的系统调用机制，并学会了如何编写和测试简单的用户程序。同时，通过解决遇到的问题，我提高了调试和解决问题的能力。实验的各个步骤帮助我掌握了在 xv6 环境下开发的基本流程，为后续的实验和学习打下了坚实的基础。

pingpong (easy)

一、实验目的

本实验的目的是通过实现一个简单的“ping-pong”程序来理解和使用 UNIX 系统调用进行进程间通信。具体来说，该程序将使用管道和进程创建系统调用，将一个字节在父子进程之间传递，并输出接收到消息的进程 ID。

二、实验步骤

1. 创建管道

使用 pipe 系统调用创建两个管道，一个用于父进程向子进程发送字节，另一个用于子进程向父进程发送字节。

2. 创建子进程

使用 fork 系统调用创建子进程。

3. 父进程操作

关闭父进程不需要的管道端（子进程写的管道写端和父进程读的管道读端）。

向子进程写入一个字节。

从子进程读取一个字节。

打印收到的消息。

4. 子进程操作

关闭子进程不需要的管道端（父进程写的管道写端和子进程读的管道读端）。

从父进程读取一个字节。

打印收到的消息。

向父进程写入一个字节。

```
int main() {
    int p1[2], p2[2];
    char buf;

    // 创建两个管道
    pipe(p1);
    pipe(p2);

    if (fork() == 0) {
        // 子进程
        close(p1[1]); // 关闭父进程写端
        close(p2[0]); // 关闭子进程读端

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

        close(p1[0]);
        close(p2[1]);
    } else {
        // 父进程
        close(p1[0]); // 关闭子进程读端
        close(p2[1]); // 关闭父进程写端

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

        close(p1[1]);
        close(p2[0]);
    }

    exit(0);
}
```

5. 在 Makefile 中添加 pingpong

6. 编译并运行

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

7. make grade 测试

```
meksa@MEKSA:~/xv6-labs-2021$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (0.7s)
meksa@MEKSA:~/xv6-labs-2021$
```

三、实验中遇到的问题和解决方法

实验过程中对管道的理解不够，查阅资料后加深了理解。

管道式一种用于进程间通信的机制。管道允许一个进程将数据写入一个文件描述符，另一个进程从这个文件描述符中读取数据，实现在进程之间传递数据。实验中通过 `pipe` 函数创建一个管道，并将两个文件描述符保存在 `pipefd` 数组中，`pipefd[0]` 是管道的读端，`pipefd[1]` 是管道的写端。一个进程可以写入数据到 `pipefd[1]`，另一个进程可以从 `pipefd[0]` 读取数据。

四、实验心得

通过本次实验，我加深了对 UNIX 系统调用的理解，特别是管道和进程创建方面。此外，我还学会了如何在 `xv6` 环境下编写和运行用户程序。

primes (moderate)/(hard)

一、实验目的

本实验的目的是使用 UNIX 管道和进程创建一个并发版的素数筛选程序。这种方法是由 UNIX 管道的发明者 Doug McIlroy 提出的。通过使用管道和 `fork` 系统调用，程序可以将一系列数通过管道传输，并利用多个进程筛选出素数。

二、实验步骤

1. 创建主进程

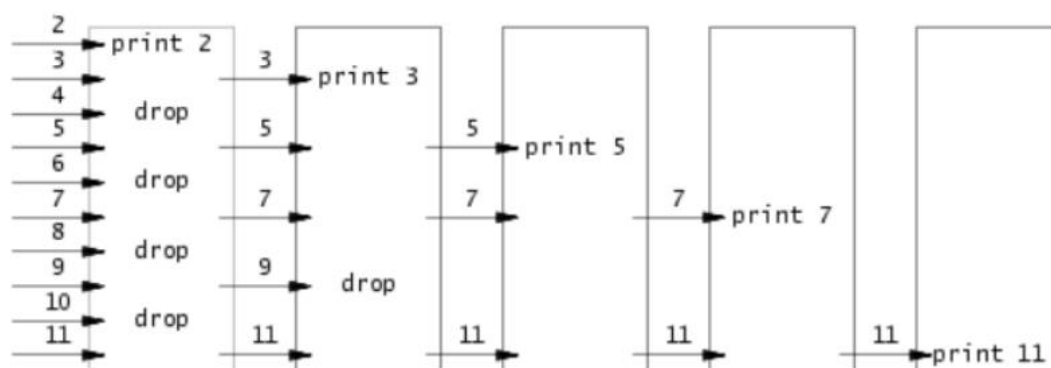
主进程生成数字 2 到 35，并将它们写入第一个管道。

2. 创建筛选进程

每个筛选进程从前一个进程的管道读取数字，筛选出第一个素数，并将其打印。

创建一个新的管道，用于向下一个进程传递剩下的数字。

筛选过程如下：



筛选通过 `filter` 函数实现，其步骤如下：

- (1) 关闭当前管道的写入段 p[1]，疑问当前进程只需要从管道读取数据。
- (2) 从管道的读取端 p[0] 中读取第一个数字，并将其打印为素数。如果读取失败（返回 0），则退出进程。
- (3) 使用 fork 创建一个新的子进程。如果是子进程，则关闭当前管道的读取端 p[0]，并递归调用 filter 函数处理新的管道 newp。
- (4) 如果是父进程，则关闭新管道的读取端 newp[0]。然后，从当前管道的读取端 p[0] 读取数字，如果数字不能被当前素数整除，则写入新管道的写入端 newp[1]。处理完所有数字后，关闭当前管道的读取端 p[0] 和新管道的写入端 newp[1]。等待子进程结束（wait(0)），并退出（exit(0)）。

```
void
filter(int p[2])
{
    int prime;
    int n;
    int newp[2];

    // 读取并打印第一个数字，即素数
    close(p[1]);
    if(read(p[0], &prime, sizeof(prime)) == 0)
        exit(0);
    printf("prime %d\n", prime);

    // 创建新的管道并派生新进程
    pipe(newp);
    if(fork() == 0){
        close(p[0]);
        filter(newp);
    } else {
        close(newp[0]);
        while(read(p[0], &n, sizeof(n))) {
            if(n % prime != 0)
                write(newp[1], &n, sizeof(n));
        }
        close(p[0]);
        close(newp[1]);
        wait(0);
        exit(0);
    }
}
```

3. 关闭不必要的文件描述符

每个进程只保留自己需要的文件描述符，关闭其余的文件描述符以节省资源。

4. 进程终止

当主进程完成数字写入后，等待所有子进程终止。

5. 编译运行

```
xv6 kernel is booting
hart 1 starting
hart 2 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
$
```

6. make grade 以测试

```
meksa@MEKSA:~/xv6-labs-2021$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.1s)
meksa@MEKSA:~/xv6-labs-2021$
```

三、实验中遇到的问题和解决方法

1. 文件描述符资源限制

在 xv6 中，文件描述符和进程数量是有限的。如果不关闭不必要的文件描述符，程序很快就会耗尽这些资源。

解决方法：确保每个进程只保留自己需要的文件描述符，并关闭其余的文件描述符。

2. 进程同步和终止

主进程需要等待所有子进程终止，确保所有输出都已完成。

解决方法：使用 wait 系统调用来等待子进程终止。

3. 在 make qemu 时出现报错

```
meksa@MEKSA:~/xv6-labs-2021$ make qemu
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -ffree-
standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/pingpong.o user/pingpong
.c
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_pingpong user/pingpong.o user/ulib.o user/usys
.o user/printf.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_pingpong > user/pingpong.asm
riscv64-unknown-elf-objdump -t user/_pingpong | sed '1,/SYMBOL TABLE/d; s/ .* //; /^$/d' > user/pingpong.sym
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -ffree-
standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/primes.o user/primes.c
user/primes.c: In function 'main':
user/primes.c:54:1: error: control reaches end of non-void function [-Werror=return-type]
 54 | }
    |
cc1: all warnings being treated as errors
make: *** [builtin: user/primes.o] Error 1
meksa@MEKSA:~/xv6-labs-2021$
```

在 main 函数的最后添加 return 0，确保函数总是返回一个整数。

四、实验心得

通过本次实验，我深入理解了 UNIX 管道和进程之间的通信机制。利用管道和 fork 系统调用可以实现复杂的并发任务，如素数筛选。同时，我也意识到在有限资源的环境中，合理管理资源（如文件描述符和进程）是至关重要的。通过解决这些问题，我学会了如何编写高效且健壮的并发程序。

find (moderate)

一、实验目的

实现一个简易版本的 UNIX find 程序，能够在目录树中查找特定名称的所有文件。

二、实验步骤

1. 创建并编辑 user/find.c 文件。

2. 查看 user/ls.c 文件以理解如何读取目录内容。学习使用递归以便在子目录中查找文件。

3. 编写代码

使用系统调用 open、read、close 来读取目录内容。

使用递归函数遍历目录树。

使用 strcmp() 比较字符串，查找特定文件名。

代码分为两个主要部分：

find 函数：负责递归遍历目录树，查找指定文件。

main 函数：程序入口，处理命令行参数并调用 find 函数。

其中，main 函数先检查命令行参数数量是否正确。如果参数不足，打印用法提示并退出。然后调用 find 函数，传入命令行参数中的路径和文件名，开始查找文件。最后调用 exit(0) 正常退出程序。

Find 函数的实现步骤如下：

(1) 初始化并打开目录：定义一个缓冲区 buf 用于存储路径，定义文件描述符 fd。尝试打开路径 path，如果失败则打印错误信息并返回。

(2) 获取文件状态：使用 fstat 获取文件或目录的状态信息。如果失败则打印错误信息并返回。

(3) 根据文件类型处理：如果是文件类型 T_FILE，检查文件名是否与目标文件名匹配。如果匹配，则打印路径。如果是目录类型 T_DIR，则要进行以下操作：

路径长度检查：检查路径长度是否超过缓冲区大小。

复制路径：将当前路径复制到缓冲区 buf，并将指针 p 指向缓冲区末尾。

读取目录内容：循环读取目录项，跳过无效目录项和 `..` 目录。

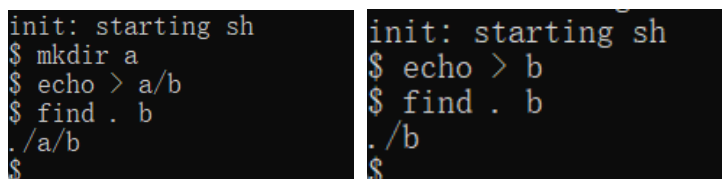
递归调用：对于每个有效目录项，构造新路径并递归调用 `find` 函数。

(4) 关闭文件描述符：处理完毕后，关闭打开的文件或目录。

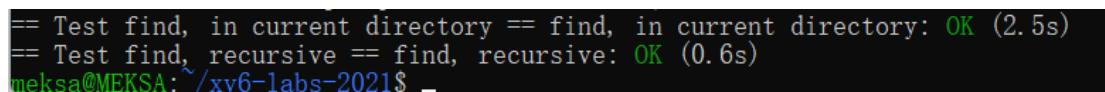
4. 测试代码

将新程序添加到 Makefile 中的 UPROGS 列表中。

编译并运行 xv6，测试 `find` 程序的功能。



5. make grade 测试



三、实验中遇到的问题和解决方法

1. 目录读取问题

最初对于如何正确读取目录内容和处理目录路径长度有限制感到困惑。由于 xv6 的文件系统和目录结构与现代文件系统不同，读取目录和处理路径的操作需要特别注意。

因此，我查看 `user/ls.c` 中的示例代码，理解如何使用 `read` 函数读取目录项，并使用 `fstat` 函数获取文件类型信息。此外，使用 `memmove` 函数将目录项名称复制到缓冲区，确保路径字符串的正确性。

2. 递归调用问题

在递归调用过程中，需要避免进入 `.` 和 `..` 目录，以防止无限递归。同时，需要确保在递归过程中正确处理路径字符串。

因此，我在递归前检查目录名，跳过 `.` 和 `..`。在处理路径字符串时，使用缓冲区拼接路径和文件名，并在每次递归调用结束后关闭相应的文件描述符，避免资源泄漏。

3. 资源管理问题

由于 xv6 系统对文件描述符数量有限制，如果不正确关闭文件描述符，可能会导致资源耗尽，程序崩溃。

解决方法是在每次打开文件和目录后，及时关闭不需要的文件描述符。特别是在递归调用时，确保每个子目录处理完毕后关闭相应的文件描述符，释放资源。

四、实验心得

通过这次实验，我深入理解了 UNIX 系统调用在文件和目录操作中的应用，并掌握了递归算法在目录遍历中的使用。

本实验让我更加熟悉了 `open`、`read`、`fstat` 和 `close` 等系统调用的使用。这些调用是文件和目录操作的基础，通过本次实验，我对它们的用法和注意事项有了更深入的理解。递归算法在处理目录树结构时非常有效，同时也让我理解了在递归过程中进行资源管理的重要性，例如正确关闭文件描述符。

xargs (moderate)

一、实验目的

实现一个简易版本的 UNIX `xargs` 程序，从标准输入读取每一行，并为每一行运行一次指定的命令，将该行作为命令的参数。

二、实验步骤

1. 创建并编辑 `user/xargs.c` 文件。

2. 总体思路

从标准输入读取每一行。

使用 `fork` 和 `exec` 系统调用来执行指定的命令。

在父进程中使用 `wait` 等待子进程完成命令执行。

3. 代码实现

创建了 `run_command` 函数用来运行命令，此函数用于创建一个子进程，在子进程中执行指定的命令，并等待子进程完成。其中：

`fork()` 用于创建一个新进程。返回值在父进程中是子进程的 PID，在子进程中是 0。

`exec(cmd, argv)` 用于在子进程中执行命令 `cmd`，并传递参数 `argv`。

`exit(0)` 在子进程中调用 `exit` 以确保子进程结束。

`wait(0)` 在父进程中等待子进程结束。

Main 函数中首先定义缓冲区 `buffer` 用于存储从标准输入读取的数据；变量 `n` 存储读取的字节数；`args` 数组用于存储命令的参数。而后检查参数个数是否至少为 2。将命令和初始参数复制到 `args` 数组，并计算初始参数的个数。读取标准输入，并逐行处理：使用 `read` 逐块读取输入，并在读取到换行符时，将其替换为字符串结束符 `'\0'`；将每一行作为参数传递给命令；调用 `run_command` 运行命令。

4. 编译并运行

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

三、实验中遇到的问题和解决方法

1. 对 echo 的理解不明

查阅后，得知 echo 是一个命令行工具，用于在标准输出上显示文本行。它通常用于在终端或脚本中输出一些消息或变量的值。

在命令行中，echo 的基本语法是：echo [option(s)] [string(s)]，其中 option(s) 是可选的选项，用于控制输出格式或特定行为。string(s) 是要显示在标准输出上的文本或变量。有一些常见选项，如：

- n: 不输出末尾的换行符。默认情况下，echo 会在输出完毕后加上一个换行符。
- e: 启用特殊字符的解析。例如，\n 表示换行符，\t 表示制表符等。

2. 读取输入的问题

在读取标准输入时，需要逐字符读取直到换行符。这要求处理换行符和缓冲区的末尾字符。

可以使用 read 函数逐块读取输入，并在读取到换行符时，将其替换为字符串结束符 '\0'，以便后续处理。

3. 参数传递的问题

需要将读取到的行作为参数传递给指定的命令，同时还需要处理初始参数。

解决方法是将初始参数复制到 args 数组，并在每次读取新行时，将行内容添加到 args 数组的末尾。确保在每次调用 exec 前，args 数组以 NULL 结尾。

4. 子进程管理的问题

需要使用 fork 创建子进程执行命令，并在父进程中等待子进程完成。

通过在子进程中调用 exec 执行命令，并在父进程中使用 wait 等待子进程完成，以确保命令执行的顺序和正确性。

四、实验心得

通过这次实验，我深入理解了 UNIX 系统调用在进程管理和输入输出处理中的应用，并掌握了 fork 和 exec 系统调用的使用。

本实验让我更加熟悉了 fork、exec 和 wait 系统调用的使用。这些调用是进程管理的基础，通过本次实验，我对它们的用法和注意事项有了更深入的理解。

通过实现逐字符读取输入并处理换行符，我进一步巩固了在 C 语言中处理字符串和缓冲区的知识。

我也学会了如何使用 `exec` 调用执行命令，并将参数传递给命令。这对于理解命令行程序的工作机制非常有帮助。

本章 lab 的 `make grade` 结果：

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.8s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.4s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (0.7s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.1s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.0s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.2s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.1s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (0.9s)
== Test time ==
time: OK
Score: 100/100
meksa@MEKSA:~/xv6-labs-2021$
```

Lab2: system calls

本实验将要求添加一些新的系统调用，以帮助了解它们是如何工作的，并对 `xv6` 内核的内部结构做一定了解。

开始本章实验前，需要将代码切换到 `syscall` 分支：

```
meksa@MEKSA:~$ cd xv6-labs-2021
meksa@MEKSA:~/xv6-labs-2021$ git checkout syscall
Already on 'syscall'
Your branch is up to date with 'origin/syscall'.
meksa@MEKSA:~/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/_in
r/_stressfs user/_usertests user/_grind user/_wc user/_
ph barrier
meksa@MEKSA:~/xv6-labs-2021$
```

System call tracing (moderate)

一、实验目的

本实验的目的是在 xv6 操作系统中添加一个系统调用跟踪功能，用于在调试时提供帮助。具体要求如下：

创建一个新的 `trace` 系统调用，该调用接受一个整数参数 `"mask"`，用于指定要跟踪的系统调用。

修改 xv6 内核，使其在每个系统调用返回时输出相应的信息（进程 ID、系统调用名称和返回值），但仅当该系统调用的编号在 `mask` 中设置时。

`trace` 系统调用应仅影响调用它的进程及其后续 `fork` 的子进程，而不影响其他进程。

二、实验步骤

1. 添加系统调用 `trace` 的用户态接口

在 `user/user.h` 中添加 `trace` 系统调用的原型：`int trace(int mask);`

在 `user/usys.pl` 中添加 `trace` 系统调用的 stub：`entry("trace");`

2. 分配系统调用号

在 `kernel/syscall.h` 中为 `trace` 分配一个新的系统调用号：`#define SYS_trace 22`

3. 实现 `trace` 系统调用的内核态函数

在 `kernel/sysproc.c` 中实现 `sys_trace` 函数：

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

4. 在进程结构体中添加 `trace_mask` 变量

在 `kernel/proc.h` 中的 `struct proc` 中添加一个 `trace_mask` 变量，用于存储 `trace` 掩码。

5. 修改 `fork` 函数以继承 `trace_mask`

在 `kernel/proc.c` 中修改 `fork` 函数，使子进程继承父进程的 `trace_mask`：

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

6. 修改 `syscall` 函数以打印跟踪信息

在 kernel/syscall.c 中修改 syscall 函数，添加对 trace_mask 的检查和打印功能：

```
extern char *syscallnames[];

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]();
        if (p->trace_mask & (1 << num)) {
            printf("%d: syscall %s -> %d\n",
                p->pid, syscallnames[num], p->trapframe->a0);
        } else {
            printf("%d %s: unknown sys call %d\n",
                p->pid, p->name, num);
            p->trapframe->a0 = -1;
        }
    }
}
```

7. 添加系统调用名称数组

在 kernel/syscall.c 中添加系统调用名称数组：

```
char *syscallnames[] = {
    [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",
};
```

8. 在 Makefile 中添加 trace 程序

添加 \$U/_trace\ 到 UPROGS。

9. make qemu 编译并测试

```

$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
$
$ trace 2147483647 grep hello README
5: syscall trace -> 0
5: syscall exec -> 3
5: syscall open -> 3
5: syscall read -> 1023
5: syscall read -> 968
5: syscall read -> 235
5: syscall read -> 0
5: syscall close -> 0
$
$ grep hello README
$
$ trace 2 usertests forkforkfork
usertests starting

```

10. make grade 测试

```

make[1]: Leaving directory '/home/meksa/xv6-labs-2021'
== Test trace 32 grep == trace 32 grep: OK (3.1s)
== Test trace all grep == trace all grep: OK (1.1s)
== Test trace nothing == trace nothing: OK (0.8s)
== Test trace children == trace children: OK (13.5s)
meksa@MEKSA:~/xv6-labs-2021$

```

三、实验中遇到的问题和解决方法

1. 打印系统调用名称

为了在 syscall 函数中获取系统调用的名称，我采用创建一个包含所有系统调用名称的数组，通过系统调用号索引获取相应的名称。

2. 父子进程间掩码继承问题

在实现 fork 时，需要确保子进程能够继承父进程的 trace_mask。

因此，选择在 fork 函数中，明确复制父进程的 trace_mask 到子进程。

3. 在测试时每次 trace 都出现了 trace failed 的现象，检查相关代码后，发现是在 syscall.c 文件中的 syscalls 中没有添加 [SYS_trace] sys_trace，添加后即可正常运行。测试成功。

四、实验心得

通过本次实验，对 xv6 内核中的系统调用机制有了更深入的理解。学会了如何添加新的系统调用，并将其集成到内核和用户态程序中。掌握了进程间通过 fork 继承属性的方法。了解了如何在内核中使用掩码来控制特定功能的启用与否。体会到了系统调用跟踪在调试和性能分析中的重要作用。

Sysinfo (moderate)

一、实验目的

本实验的目的是在 xv6 操作系统中添加一个系统调用 sysinfo，用于收集系统的运行信息。具体要求如下：

添加一个新的系统调用 sysinfo，它接受一个指向 struct sysinfo 的指针作为参数。

内核应填充该结构体的字段：freemem 字段应设置为空闲内存的字节数，nproc 字段应设置为状态不是 UNUSED 的进程数。

通过提供的测试程序 sysinfotest 进行测试，当程序打印 "sysinfotest: OK" 时，表示实验通过。

二、实验步骤

1. 在 Makefile 的 UPROGS 中添加 \$U/_sysinfotest。

2. 添加系统调用 sysinfo 的用户态接口

在 user/user.h 中添加 sysinfo 系统调用的原型：int sysinfo(struct sysinfo *); 并在此之前需要申明 struct sysinfo; 此结构体已经在 kernel/sysinfo.h 定义。

user/usys.pl 中添加 sysinfo 系统调用的 stub：entry("sysinfo");

3. 分配系统调用号

在 kernel/syscall.h 中为 sysinfo 分配一个新的系统调用号：

```
#define SYS_sysinfo 23
```

4. 实现 sysinfo 系统调用的内核态函数

在 kernel/sysproc.c 中实现 sys_sysinfo 函数：

```
uint64
sys_sysinfo(void){
    struct sysinfo info;
    struct sysinfo *p;

    if(argaddr(0, (uint64*)&p) < 0)
        return -1;

    // 获取空闲内存
    info.freemem = freemem();

    // 获取非 UNUSED 进程数
    info.nproc = nproc();

    if(copyout(myproc()->pagetable, (uint64)p, (char *)&info, sizeof(info)) < 0)
        return -1;

    return 0;
}
```

5. 收集空闲内存信息

在 kernel/kalloc.c 中添加 freemem 函数:

```
//Calculate the free memory
uint64
freemem(void) {
    struct run *r;
    uint64 free_memory = 0;

    acquire(&kmem.lock);
    for(r = kmem.freelist; r; r = r->next)
        free_memory += PGSIZE;
    release(&kmem.lock);

    return free_memory;
}
```

6. 收集进程信息

在 kernel/proc.c 中添加 nproc 函数:

```
// Calculate used process
uint64
nproc(void) {
    struct proc *p;
    int count = 0;

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

    return count;
}
```

7. 在 kernel/syscall.c 中, 增添 extern uint64 sys_trace(void); 与 [SYS_sysinfo] sys_sysinfo。

8. make qemu 编译并测试

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$
```

9. make grade 测试

```
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (1.8s)
```

三、实验中遇到的问题和解决方法

1. make qemu 时出现下图报错

```
kernel/sysproc.c: In function 'sys_sysinfo':
kernel/sysproc.c:119:20: error: implicit declaration of function 'freemem' [-Werror=implicit-
function-declaration]
  119 |         info.freemem = freemem();
      |                        ~~~~~~
kernel/sysproc.c:122:18: error: implicit declaration of function 'nproc'; did you mean 'mypro
c'? [-Werror=implicit-function-declaration]
  122 |         info.nproc = nproc();
      |                        ~~~~~
                           myproc
cc1: all warnings being treated as errors
make: *** [Makefile:135: kernel/sysproc.o] Error 1
```

说明 freemem 和 nproc 函数在 kernel/sysproc.c 中没有声明，因此可以在 kernel/sysproc.c 中声明这两个函数。

2. 拷贝数据到用户空间

在思考如何将内核中的数据拷贝到用户空间时，我参考了 sys_fstat 函数，使用 copyout 函数将数据从内核空间拷贝到用户空间。

sys_fstat 是一个系统调用，用于获取文件的状态信息，并将这些信息复制到用户空间的一个结构体中。其主要步骤如下：

- 1) 获取参数：从系统调用参数中获取文件描述符和指向 stat 结构体的指针。
- 2) 找到文件对应的内核结构体：通过文件描述符找到对应的文件。
- 3) 获取文件状态：调用内部函数 filestat 获取文件的状态信息。
- 4) 复制到用户空间：使用 copyout 函数将状态信息从内核空间复制到用户空间。

sys_sysinfo 函数的实现类似于 sys_fstat，其

3. 获取内存空闲空间

为了计算空闲内存的字节数，我的方法是在 kalloc.c 中遍历空闲链表，并累加所有空闲页的大小。目的是将系统信息复制到用户空间的结构体中。其主要步骤如下：

- 1) 获取参数：从系统调用参数中获取指向 sysinfo 结构体的指针。
- 2) 填充系统信息：获取当前系统的空闲内存和进程数，并填充到 sysinfo 结构体中。
- 3) 复制到用户空间：使用 copyout 函数将信息从内核空间复制到用户空间。

4. 获取非 UNUSED 进程数

为了统计状态不是 UNUSED 的进程数，通过在 proc.c 中遍历进程表，统计相应的进程数。

四、实验心得

通过本次实验，对 xv6 内核中的系统调用机制有了更深入的理解。具体而言：我学会

了如何添加新的系统调用，并将其集成到内核和用户态程序中。掌握了如何在内核中收集系统信息，并将其安全地传递给用户空间。了解了内存管理和进程管理的基本原理，通过实践加深了对这些概念的理解。体会到了系统信息收集在性能监控和调试中的重要作用。

总之，本实验不仅提升了对操作系统内核的理解，还锻炼了调试和问题解决的能力，为后续的实验打下了良好的基础。

本章 lab 的 make grade 结果：

```
make[1]: Leaving directory '/home/meksa/xv6-labs-2021'
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (3.6s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.8s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (14.0s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (1.8s)
== Test time ==
time: OK
Score: 35/35
meksa@MEKSA:~/xv6-labs-2021$
```

Lab3: page tables

本章实验将探索页表并对其进行修改，以简化将数据从用户空间复制到内核空间的函数。

启动实验前，先切换到 pgtbl 分支：

```
meksa@MEKSA:~/xv6-labs-2021$ git fetch
gitmeksa@MEKSA:~/xv6-labs-2021$
meksa@MEKSA:~/xv6-labs-2021$ git checkout pgtbl
Branch 'pgtbl' set up to track remote branch 'pgtbl' from 'origin'
Switched to a new branch 'pgtbl'
```

Speed up system calls (easy)

一、实验目的

本实验的目的是通过在用户空间和内核之间共享只读数据区域，优化某些系统调用的性能，从而减少进入内核的开销。具体来说，本实验要求我们为 `getpid()` 系统调用实现

这一优化。在每个进程创建时，在 USYSCALL 虚拟地址处映射一个只读页面，该页面包含一个 struct usyscall，用于存储当前进程的 PID。用户空间的 ugetpid() 函数将自动使用这个 USYSCALL 映射，从而加速 getpid() 系统调用。

二、实验步骤

1. 查看 memlayout.h 文件

该头文件描述了在 QEMU 模拟的 RISC-V 机器上运行的 xv6 内核的物理和虚拟内存布局。它定义了各种硬件组件和内存区域的地址。它为内核和用户空间设置了虚拟内存布局，包括跳板页和系统调用共享页等特殊页面。它引入了 struct usyscall 并映射它以优化系统调用。

关于内核内存布局，KERNBASE 是内核的基地址，PHYSTOP 是内核物理内存的结束位置。关于内核和用户空间的虚拟内存布局，TRAMPOLINE 是在用户和内核空间中将跳板页面映射到最高地址，KSTACK(p) 用来计算进程 p 的内核堆栈地址，并有保护页进行保护。用户内存布局分为：文本（代码段）、原始数据和 BSS（数据段）、堆栈、堆、USYSCALL（用户和内核共享的页面，用于系统调用）、TRAMPFRAME（存储上下文切换的陷入帧信息，位于跳板页下方）、TRAMPOLINE（用于处理系统调用和中断的共享页面）。

2. 在 proc.h 中添加 usyscall 字段

在 struct proc 结构中添加一个 struct usyscall * 类型的字段，用于指向用户空间的 usyscall 结构。

3. 在 kernel/proc.c 中的 proc_pagetable() 中添加虚拟页的映射关系

此处仿照将 trapframe 映射到 TRAMPOLINE 的方式：

在 proc.h 中添加 USYSCALL 的结构体：struct usyscall *usyscall; 使用 mappages() 函数，将 usyscall 结构（其物理地址由 (uint64) (p->usyscall) 给出）映射到用户空间的 USYSCALL 地址处，映射的页面大小为 PGSIZE，允许读和访问；执行失败时，将 USYSCALL、TRAMPOLINE 从页表中删去并释放。

4. 在 proc.c 中修改 allocproc()

在 allocproc() 函数中分配一个页面，并将其映射到 USYSCALL 地址。同时，初始化 usyscall 结构以存储当前进程的 PID。

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

5. 在 proc.c 中修改 freeproc()

在 freeproc() 函数中释放分配给 usyscall 结构的页面。

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

6. 在 proc.c 中修改 fork()

在 fork() 函数中确保子进程继承父进程的 usyscall 映射。

```
// Allocate a page for the usyscall structure and map it into the user space
void *page = kalloc();
if (page == 0) {
    freeproc(np);
    release(&np->lock);
    return -1;
}

// Initialize the usyscall structure with the parent's PID
((struct usyscall *)page)->pid = p->pid;

// Map the page into the child's user space at USYSCALL
if (mappages(np->pagetable, USYSCALL, PGSIZE, (uint64)page, PTE_R | PTE_U) != 0) {
    kfree(page);
    freeproc(np);
    release(&np->lock);
    return -1;
}
```

7. make qemu 编译并运行

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgtbltest: pgaccess_test failed: incorrect access bits set, pid=3
```

三、实验中遇到的问题和解决方法

1. 在 fork 函数中添加共享只读页以优化 getpid 系统调用遇到困难

最终通过以下步骤进行修改：

(1) 在分配和初始化子进程 (allocproc) 之后、内存复制 (uvmcopy) 之前，分配一个新页并将其映射到子进程的页表中。

(2) 分配页后，将父进程的 PID 写入页中的 usyscall 结构中。

(3) 使用 mappages 函数将该页映射到子进程的用户空间，并确保设置为只读。

2. make qemu 时出现 kerneltrap

这可能是由内存访问错误或页面映射错误或指针使用错误导致的。

要确保所有操作都正确无误：

确保在 allocproc 和 freeproc 函数中正确分配和释放资源。

确保 `kalloc` 成功分配了内存，且 `kfree` 在出错时正确释放了内存。

确保 `mappages` 函数正确地将页映射到用户空间。

确保 `mappages` 中设置的页表项权限是正确的。

可以通过添加调试信息来定位问题，在每个关键步骤之后添加 `printf` 调用来打印相关信息。

但各种调试后都没有变化。

经仔细检查后，发现 `proc_pagetable` 中没有添加虚拟页的映射关系。

添加相应代码：

```
// map the the usyscall just below TRAPFRAME
if(mappages(pagetable, USYSCALL, PGSIZE,
            (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
    //执行失败则将USYSCALL、TRAMPOLINE从页表中删去并释放
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
}
```

重新运行后，可以成功通过 `ugetpid` 测试：

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

四、实验心得

通过本次实验，我深入理解了如何在 `xv6` 操作系统中进行页面映射和共享数据。通过在用户空间和内核之间共享只读数据区域，可以显著减少某些系统调用的开销，例如 `getpid()`。这种优化方法不仅适用于 `getpid()`，还可以推广到其他频繁调用且无需修改内核数据的系统调用，如 `getppid()` 和 `uptime()`。总之，本实验提高了我对操作系统内存管理和系统调用机制的理解和掌握。

除了 `getpid()`，以下系统调用也可以通过类似的方式加速：

`getppid()`：获取父进程 ID。可以在 `usyscall` 结构中添加一个字段存储父进程 ID。

`uptime()`：获取系统运行时间。可以在 `usyscall` 结构中添加一个字段存储系统启动时间或当前时间。

通过在 `usyscall` 结构中添加这些额外的信息，并在合适的时机更新这些信息，可以减少进入内核的频率，从而提高系统调用的性能。

Print a page table (easy)

一、实验目的

本实验的目的是实现一个函数 `vmprint()`，用于打印 RISC-V 页表的内容。通过这个函数，能够直观地看到页表的结构和各级页表项的映射关系，从而帮助调试和理解 xv6 操作系统中的虚拟内存管理。实验的目标是让 `vmprint()` 函数按照指定格式输出页表内容，并在 xv6 内核中插入合适的调用点来验证其工作效果。

二、实验步骤

1. 实现 `vmprint()` 函数

在 `kernel/vm.c` 中定义 `vmprint()` 函数，函数接受一个 `pagetable_t` 类型的参数。

使用递归的方法遍历页表树，从顶层页表开始，逐级打印有效的页表项。

打印格式要求每个页表项包含索引、PTE 的值、以及由 PTE 中提取的物理地址。

利用宏 `PTE2PA` 和 `PTE_FLAGS` 从页表项中提取物理地址和标志位，并用 `%p` 格式化输出。

2. 在 `kernel/defs.h` 中声明 `vmprint()` 函数

将 `vmprint()` 函数的原型添加到 `kernel/defs.h` 中，以便在其他文件中调用。

3. 在 `exec.c` 中插入 `vmprint()` 调用

在 `exec.c` 中 `exec()` 函数的末尾，插入 `vmprint()` 的调用，以便在第一个进程 (`pid=1`) 的页表创建后打印其内容。具体来说，可以在 `return argc;` 之前加入以下代码：

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

4. 编译并运行 xv6

使用 `make qemu` 编译并启动 xv6，观察输出是否符合预期格式。

验证 `vmprint()` 函数的输出是否正确反映了第一个进程的页表结构。

```
xv6 kernel is booting
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
$
```

在 RISC-V 的三级页表结构中，每个页表条目（PTE）指向下一级页表或实际的物理页面。vmprint 输出显示了从顶层页表（Level 1）到最终的物理页面的映射过程。

- Level 1: 是顶级页表，它包含 512 个条目中的一个或多个。输出的第一个条目是 0，它指向一个下一级页表，物理地址为 0x0000000087f6a000。
- Level 2: 对应条目 0，指向一个下一级页表，物理地址为 0x0000000087f69000。
- Level 3: 这个页表包含实际映射的物理页面。例如，0x0000000087f6b000 映射到第一个物理页面，0x0000000087f68000 映射到第二个物理页面，0x0000000087f67000 映射到第三个物理页面。

关于页表的内容，Page 0 对应第一个映射的虚拟地址。0x0000000087f6b000 是物理页面的实际内容，PTE2PA(pte) 的结果指向这个物理地址。Page 2 映射到物理地址 0x0000000087f67000，它是第三个物理页面。

对于每个 PTE，要关注 PTE 中的权限位（PTE_R、PTE_W、PTE_X 等）来确定用户模式下是否可以读/写/执行这些内存区域。从 vmprint 的输出中，Page1 第一个物理页面（即 0x0000000087f6b000）可能包含可执行代码或只读数据。可以通过 pte 值中的 PTE_R、PTE_W、PTE_X 位来判断。如果 PTE_W 位未设置，意味着在用户模式下，这个页表条目映射的页面是只读的；如果 PTE_X 位设置了，意味着这个页面包含可执行代码。

第三个物理页面，这里映射的物理地址是 0x0000000087f67000，它可能包含数据或堆栈。它的具体内容取决于权限位设置以及程序对内存的使用。

三、实验中遇到的问题和解决方法

1. 输出格式问题

在实现 vmprint() 的过程中，确保输出格式与实验要求完全一致，包括缩进、PTE 的表示方式、以及物理地址的打印格式。

通过调整 printf 的参数和递归调用，最终实现了符合要求的格式。

2. 页表遍历中的错误处理

在递归遍历页表时，需要确保只打印有效的页表项（即 PTE_V 位被设置的项）。最初的实现中错误地打印了无效项，通过增加条件检查解决了这个问题。

3. 测试时的不同物理地址

由于 xv6 的运行环境不同，实际输出的物理地址可能与实验示例不一致。确认页表结构和索引无误后，不必纠结于物理地址的差异。

四、实验心得

本次实验通过实现 vmprint() 函数，加深了对 RISC-V 页表结构的理解。通过打印页表，可以直观地看到虚拟地址如何映射到物理地址，这对于理解虚拟内存的工作机制非常有帮助。实验中的关键是确保页表的遍历和打印格式符合要求，同时理解页表各级的含义及其作用。在未来的调试工作中，vmprint() 函数也将成为一个重要的工具，用于快速定位和分析内存管理相关的问题。

Detecting which pages have been accessed (hard)

一、实验目的

本实验的目标是为 xv6 操作系统添加一个新的系统调用 `pgaccess()`，用于检测并报告哪些页面被访问过。这个功能对垃圾回收器（自动内存管理的一种形式）非常有用，因为它可以提供哪些页面被读写过的信息。

二、实验步骤

1. 理解 PTE_A 访问位的作用

在 RISC-V 架构中，每个页面表项（PTE）中包含一个访问位（PTE_A）。当一个页面被 CPU 访问（读取或写入）时，硬件会自动将该页面对应的 PTE 中的访问位设置为 1。

本实验需要定义 PTE_A 常量，这样在检测访问状态时可以用它来判断页面是否被访问过。

2. 在 kernel/riscv.h 中定义 PTE_A

打开 `kernel/riscv.h` 文件，找到页面表项相关的定义。

添加以下定义以表示访问位：

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_A (1L << 6)
```

这个定义表示 PTE_A 位位于页面表项的第 6 位。

3. 实现 `sys_pgaccess()` 系统调用

打开 `kernel/sysproc.c`，定义 `sys_pgaccess()` 函数，它是用户空间调用 `pgaccess()` 的实际内核实现。

这个函数需要解析传入的三个参数，并检查指定范围内的页面访问情况。

```

uint64
sys_pgaccess(void) {
    uint64 start_va;
    int num_pages;
    uint64 user_mask;

    // 解析用户传入的参数
    if (argaddr(0, &start_va) < 0)
        return -1;
    if (argint(1, &num_pages) < 0)
        return -1;
    if (argaddr(2, &user_mask) < 0)
        return -1;

    // 设置一个内核缓冲区来存储访问结果
    uint64 mask = 0;
    pagetable_t pagetable = myproc()->pagetable;

    for (int i = 0; i < num_pages; i++) {
        pte_t *pte = walk(pagetable, start_va + i * PGSIZE, 0);
        if (pte && (*pte & PTE_V) && (*pte & PTE_A)) {
            mask |= (1L << i); // 设置对应位
            *pte &= ~PTE_A;    // 清除访问位
        }
    }

    // 将内核缓冲区内容拷贝到用户空间
    if (copyout(pagetable, user_mask, (char *)&mask, sizeof(mask)) < 0)
        return -1;

    return 0;
}

```

4. 在 kernel/syscall.c 中添加对 sys_pgaccess 的支持

在 kernel/syscall.c 中的 syscall() 函数里，添加对 sys_pgaccess 的处理。

```

#ifdef LAB_PGTBL
[SYS_pgaccess] sys_pgaccess,
#endif

```

5. 在 defs.h 中声明 sys_pgaccess

6. 在用户空间调用 pgaccess() 系统调用

编写用户程序 user/pgaccess_test.c。

```

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

int main() {
    char *addr = sbrk(4096 * 2); // 分配两个页的内存
    addr[0] = 'x'; // 访问第一个页

    unsigned int accessed = 0;
    if (pgaccess(addr, 2, &accessed) < 0) {
        printf("pgaccess failed\n");
        exit(1);
    }

    if (accessed & 1)
        printf("Page 0 accessed\n");
    if (accessed & 2)
        printf("Page 1 accessed\n");

    exit(0);
}

```

将 `pgaccess_test.c` 添加到 `Makefile` 中，编译并运行该程序，验证 `pgaccess` 系统调用是否正常工作。

7. 调试与测试

使用 `vmprint()` 函数来打印页面表的结构，帮助验证页面表项是否正确设置和清除。

运行 `pgtbltest` 测试程序，确保 `pgaccess()` 系统调用能正确检测页面访问情况，并通过所有测试。

三、实验中遇到的问题和解决方法

1. make qemu 时出现报错

```
meksa@MEKSA:~/xv6-labs-2021$ make qemu
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_P
standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -
c.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_P
standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_P
standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_P
standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_P
standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_P
standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -
all.c
kernel/syscall.c:111:15: error: conflicting types for 'sys_pgaccess'
 111 | extern uint64 sys_pgaccess(void);
      |
In file included from kernel/syscall.c:8:
kernel/defs.h:107:17: note: previous declaration of 'sys_pgaccess' was here
 107 | int sys_pgaccess(void);
      |
make: *** [Makefile:130: kernel/syscall.o] Error 1
meksa@MEKSA:~/xv6-labs-2021$
```

这个错误是由于 `sys_pgaccess` 函数的声明在两个文件中不一致造成的。具体来说，在 `kernel/defs.h` 中，声明了 `sys_pgaccess` 返回一个 `int` 类型，而在 `kernel/syscall.c` 中声明了它返回一个 `uint64` 类型。这种类型不一致会导致编译器报错。

解决方法：在两个文件中统一 `sys_pgaccess` 函数的返回类型为 `uint64`。但修改后发现还有报错，仔细检查后发现 `sysproc.c` 中已有 `sys_pgaccess` 函数，重复定义了。

此外，还有错误在 `kernel/sysproc.c` 中调用 `walk` 函数时，该函数的声明或定义没有被正确地包含在文件中，导致编译器无法识别 `walk` 函数。

解决方法：在 `kernel/defs.h` 中添加 `walk` 函数的声明。

```
pte_t *walk(pagetable_t pagetable, uint64 va, int alloc);
```

将以上问题修正后，可以正常编译运行。


```
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 0x0000000021fdaclf pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9clf 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
$
```

2. 访问位定义的问题

问题：最初在定义 PTE_A 访问位时，无法确定其确切位置和含义。

解决方法：通过查阅 RISC-V 的架构文档，明确了访问位的位置和作用，最终正确地定义了 PTE_A 常量。

3. 清除 PTE 访问位

问题：在检测页面访问状态后，如何清除 PTE 中的 PTE_A 位成为一个难点。误操作可能导致页面状态不能正确更新。

解决方法：仔细阅读了 walk() 函数的实现细节，确认了如何安全地修改页面表项，并成功地清除了 PTE_A 位。

4. 结果位掩码的生成与拷贝

问题：生成结果位掩码并将其拷贝到用户缓冲区中，确保位顺序正确，是一个需要细心处理的过程。

解决方法：通过在内核中先生成位掩码，再使用 copyout() 函数将其拷贝到用户缓冲区，确保了结果的正确性。

5. make grade 时报错

```
== Test pgtbltest ==
$ make qemu-gdb
(3.0s)
== Test   pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.4s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: FAIL
Cannot read answers-pgtbl.txt
== Test usertests ==
$ make qemu-gdb
```

查阅论坛以及阅读操作文档，发现是少在根目录下添加 answers-pgtbl.txt 文本，添加完成之后，需要向其中加入较多文字，之后提交便可以通过测试，获取满分。

四、实验心得

在这个实验中，我实现了一个新的系统调用 `pgaccess()`，其目的是检测哪些页面被访问过（读取或写入）。该系统调用通过检查 RISC-V 页表中的访问位（Accessed bit），将这些信息反馈给用户态。通过这个实验，我对操作系统如何管理虚拟内存，以及如何与硬件交互以跟踪页面的使用情况有了更深入的理解。

在操作系统的内存管理中，追踪哪些页面被访问过是很有用的，例如在实现垃圾回收机制时。通过这个实验，我学习了如何利用硬件提供的访问位来检测页面的访问情况。访问位是由硬件在每次页面被访问时自动设置的，这样我们可以通过读取这个位来知道页面是否被访问过。

为了实现 `pgaccess()`，我们需要使用 `walk` 函数来遍历页表，从而找到对应的页表项（PTE）。这让我更加深入理解了页表的结构，以及如何通过多级页表定位特定的物理页面。尽管这是一个较为复杂的操作，但通过实践，我对页表的工作机制有了更清晰的认识。

在实现过程中，我遇到了编译错误，例如函数声明不匹配、头文件包含错误等。这些问题提醒我，精确管理代码的依赖关系和函数声明是编写系统级代码的基础。在修复这些问题的过程中，我学会了如何更加系统地调试和检查代码。

在实验的最后，通过阅读访问位的状态，我们成功实现了对页面访问的检测。然而，这也让我意识到，在实际应用中如何合理地使用这些信息是一个更大的挑战。例如，在某些情况下，频繁地检查和清除访问位可能会影响系统性能。这也让我更好地理解操作系统设计中需要权衡的各个方面。

通过这个实验，我不仅学会了如何检测页面访问，还加深了对虚拟内存管理和页表的理解。这些知识对以后理解和设计更复杂的内存管理机制会有很大帮助。同时，实验中遇到的各种问题也让我意识到细节处理的重要性，特别是在系统编程中，任何细小的错误都可能导致系统崩溃或不可预料的行为。这次实验的收获将为我未来的操作系统学习和开发奠定坚实的基础。

本 Lab 的 `make grade` 结果如下图所示：

```
== Test pgtbltest ==
$ make qemu-gdb
(3.1s)
== Test  pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test  pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (1.0s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(205.7s)
== Test  usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
meksa@MEKSA:~/xv6-labs-2021$
```

Lab4: traps

本实验探索如何使用陷阱实现系统调用。首先使用栈做一个热身练习，然后实现一个用户级陷阱处理的示例。

RISC-V assembly (easy)

一、实验目的

本实验旨在理解 RISC-V 汇编代码，掌握基本的寄存器操作、函数调用和内存布局。同时，通过分析汇编代码，加深对 RISC-V 指令集架构和编译器行为的理解。

二、实验步骤

1. 采用以下操作，切换到 Lab4 实验环境中。

```
meksa@MEKSA:~/xv6-labs-2021$ git fetch
meksa@MEKSA:~/xv6-labs-2021$ git checkout traps
git: 'checkout' is not a git command. See 'git --help'.

The most similar command is
    checkout
meksa@MEKSA:~/xv6-labs-2021$ git checkout traps
Branch 'traps' set up to track remote branch 'traps' from 'origin'
Switched to a new branch 'traps'
meksa@MEKSA:~/xv6-labs-2021$ make clean
```

2. 在 xv6 项目目录下，使用 make fs.img 命令编译代码，生成 user/call.asm 文件，其中包含 user/call.c 文件的汇编代码。

```
meksa@MEKSA:~/xv6-labs-2021$ make fs.img
gcc -DSOL_TRAPS -DLAB_TRAPS -Werror -Wall -I. -o mkfs/mkfs mkfs/mkfs.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb
-standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fn
perl user/usys.pl > user/usys.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb
-standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fn
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb
-standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fn
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb
-standing -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fn
.c
```

```
user/_call:      file format elf64-littleriscv
```

```
Disassembly of section .text:
```

```
0000000000000000 <g>:
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int g(int x) {
| 0: 1141                addi  sp,sp,-16
```

3. 分析汇编代码:

打开 user/call.asm 文件, 找到函数 g、f 和 main 的汇编代码。

根据 RISC-V 指令集手册, 分析寄存器、函数调用及返回地址等信息。

```
int g(int x) {  
    0: 1141          addi sp,sp,-16  
    2: e422          sd s0,8(sp)  
    4: 0800          addi s0,sp,16  
    return x+3;  
}
```

```
int f(int x) {  
    e: 1141          addi sp,sp,-16  
    10: e422         sd s0,8(sp)  
    12: 0800         addi s0,sp,16  
    return g(x);  
}
```

```
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+0xea>  
    30: 00000097      auipc ra,0x0  
    34: 5f8080e7      jalr 1528(ra) # 628 <printf>  
    exit(0);  
    38: 4501          li a0,0  
    3a: 00000097      auipc ra,0x0  
    3e: 276080e7      jalr 630(ra) # 2b0 <exit>  
}
```

3. 回答问题

根据汇编代码回答以下问题, 并将答案记录在文件 answers-traps.txt 中:

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

函数参数存储在寄存器 a0~a7 中, 例如 main 函数的 `printf` 中的 13 寄存在 a2 寄存器中。

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.)

没有这样的代码。 `g` 被内联 inline 到 `f(x)` 中, 然后 `f` 又被进一步内联到 `main`

中。

At what address is the function `printf` located?

`printf` 函数的地址可以通过在 `main` 函数中查找 `jalr` 指令找到。在汇编代码中，`jalr` 指令会跳转到一个寄存器保存的地址，而这个地址就是 `printf` 函数的入口地址。

0x0000000000000628, `main` 中使用 `pc` 相对寻址来计算得到这个地址。

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

在执行 `jalr` 指令后，寄存器 `ra`（返回地址寄存器）会保存调用者函数的下一条指令的地址。也就是说，在调用 `printf` 后，`ra` 会保存 `main` 函数中 `printf` 调用之后的那条指令的地址。

0x0000000000000038, `jalr` 指令的下一条汇编指令的地址。

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?

[Here's a description of little- and big-endian](#) and [a more whimsical description](#).

输出是 HE110 World

在大端模式下，要得到相同的输出，变量 `i` 应该设为 0x726c6400，使内存布局为 00 64 6c 72，这样与小端模式下的结果一致。

57616 的值可以保持不变，因为它只影响 `%x` 的输出，与内存布局无关。

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);
```

在这段代码中，`printf` 的格式字符串指定了两个整数 `%d`，但只传递了一个参数 3。由于 `printf` 在栈上读取参数，因此 `y` 后打印的值是不确定的，它可能是栈中先前的内容或者随机值。因为少传了一个参数，`printf` 会继续从栈中读取一个未指定的值。这种行为通常被称为“未定义行为”（Undefined Behavior）。实际输出内容取决于当时栈上的内容，因此无法确定一个具体值。

三、实验中遇到的问题和解决方法

1. 理解寄存器的使用

通过查阅 RISC-V 手册，明确 a0-a7 寄存器用于传递函数参数，ra 寄存器保存返回地址。

在分析汇编代码时，发现 main 函数中调用 printf 时，寄存器 a0 保存了第一个参数，其他参数依次存储在 a1-a2 寄存器中。

2. 函数内联优化

发现编译器对小函数进行了内联优化，导致汇编代码中没有显式的 call 指令。这需要通过分析指令序列来确认函数调用。

解决方法是手动分析代码，结合寄存器操作和指令流，确认函数的实际调用位置。

3. 字节序问题

理解小端和大端的概念，分析代码中的内存布局，确定如何在不同字节序下输出相同的结果。

通过修改变量赋值的方式，实现大端模式下的正确输出。

4. printf 参数匹配问题

发现 printf 函数中参数个数不匹配时，可能导致未定义行为。需要仔细分析栈和寄存器的状态，理解 printf 的工作机制。

四、实验心得

通过本次实验，我深入理解了 RISC-V 的汇编语言以及底层计算机体系结构。分析汇编代码的过程帮助我加深了对编译器行为的理解，特别是内联优化和寄存器分配策略。此外，通过解决字节序问题，我进一步巩固了对小端和大端存储方式的理解。

实验还让我意识到，掌握底层代码对于提高编程能力和调试复杂问题至关重要。熟悉汇编代码和硬件体系结构，可以帮助开发者更好地理解高级编程语言的执行过程，并有效地优化程序性能。

Backtrace (moderate)

一、实验目的

在调试操作系统内核时，函数调用栈的回溯（Backtrace）对于定位和分析错误非常有帮助。通过本实验，将学习如何在 RISC-V 架构的 xv6 操作系统中实现 backtrace 功能，以便在程序发生错误时输出函数调用栈的信息。这个功能将在调试过程中极大地帮助开发者理解错误发生的路径。

二、实验步骤

1. 初始化 Backtrace 功能

首先，在 kernel/printf.c 中定义 backtrace() 函数。这个函数将遍历当前的调用栈，输出各个函数的返回地址。

在 kernel/riscv.h 中添加如下内联函数 r_fp()，用于获取当前函数的帧指针（frame pointer）。RISC-V 架构中，帧指针存储在 s0 寄存器中。

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

在 kernel/defs.h 中声明 backtrace() 函数的原型，使得其他文件可以调用它。

2. 实现 Backtrace 函数

在 kernel/printf.c 中实现 backtrace() 函数，遍历调用栈并打印每一层的返回地址。帧指针 fp 用于在栈中找到上一个栈帧的位置，返回地址（ra）存储在当前栈帧的 fp-8 处，而上一个栈帧的帧指针存储在 fp-16 处。

使用 PGROUNDUP(fp) 和 PGROUNDUP(fp) 来计算当前栈的上下边界，确保遍历过程不会超出当前栈的范围。

```
void  
backtrace(void) {  
    uint64 fp = r_fp(); // 获取当前帧指针  
  
    printf("backtrace:\n");  
  
    while (fp != 0) {  
        uint64 ra = *(uint64 *)(fp - 8); // 返回地址  
        printf("%p\n", ra);  
  
        fp = *(uint64 *)(fp - 16); // 上一个栈帧的帧指针  
        if (fp < PGROUNDUP(fp) || fp >= PGROUNDUP(fp)) {  
            break; // 遇到栈顶或栈底，停止遍历  
        }  
    }  
}
```

3. 调用 Backtrace 函数

在 sys_sleep() 函数中插入对 backtrace() 的调用，使得在调用 sleep 时可以输出当前的调用栈信息。

```
uint64
sys_sleep(void)
{
    int n;
    uint ticks0;

    if(argint(0, &n) < 0)
        return -1;
    backtrace(); // 添加调用栈回溯

    acquire(&tickslock);
    ticks0 = ticks;
    while(ticks - ticks0 < n){
        if(myproc()->killed){
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock);
    }
    release(&tickslock);
    return 0;
}
```

在 panic 函数中也调用 backtrace(), 以便在内核发生 panic 时能够自动输出函数调用栈。

```
void
panic(char *s)
{
    backtrace(); // 添加调用栈回溯
    pr.locking = 0;
    printf("panic: ");
    printf(s);
    printf("\n");
    panicked = 1; // freeze uart output from other CPUs
    for(;;)
        ;
}
```

4. 测试 Backtrace 功能

使用 bttest 测试 sys_sleep 中的 backtrace 功能。bttest 会触发 sleep 调用, 从而执行插入的 backtrace(), 打印调用栈信息。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x00000000800020dc
0x0000000080001fa4
0x0000000080001c8e
$
```

使用 addr2line -e kernel/kernel 命令, 将输出的地址转换为具体的代码行, 以便进一步分析。


```
meksa@MEKSA:~/xv6-labs-2021$ addr2line -e kernel/kernel
0x00000000800020dc
/home/meksa/xv6-labs-2021/kernel/sysproc.c:66
0x0000000080001fa4
/home/meksa/xv6-labs-2021/kernel/syscall.c:140
0x0000000080001c8e
/home/meksa/xv6-labs-2021/kernel/trap.c:76
```

5. 将 `backtrace()` 加到 `kernel/printf.c` 中的 `panic()` 函数中，应用程序崩溃了就可以把他的调用关系打印出来。

三、实验中遇到的问题和解决方法

1. 理解帧指针与栈帧结构

初次编写 `backtrace` 时，对 RISC-V 栈帧结构的理解不足，导致帧指针计算错误。解决方法是参考 RISC-V 的架构文档和 `xv6` 的相关代码，详细分析栈帧的布局，最终确定了正确的 `fp-8` 和 `fp-16` 偏移量。

了解相关材料后对函数调用栈和栈帧有所理解。

函数调用栈 (Stack)：由高地址往低地址增长；在 `xv6` 里，有一页大小 (4KB)；栈指针 (`stack pointer`) 保存在 `sp` 寄存器里。

栈帧 (Stack Frame)：当前栈帧的地址保存在 `s0/fp` 寄存器里；当前栈帧的地址也叫栈帧的指针 (`frame pointer`, `fp`)，指向该栈帧的最高处；栈帧指针往下偏移 8 个字节是函数返回地址 `return address`，往下偏移 16 个字节是上一个；栈帧的栈帧指针

(`previous frame pointer`)；`sp` 是 `stack pointer`，用于指向栈顶 (低地址)，保存在寄存器中；`fp` 是 `frame pointer`，用于指向当前帧底部 (高地址)，保存在寄存器中，同时每个函数栈帧中保存了调用当前函数的函数 (父函数) 的 `fp` (保存在 `to prev frame` 那一栏中)；这些栈帧都是由编译器 编译生成的汇编文件生成的。

2. 边界条件处理

在遍历调用栈时，如果不正确处理栈的边界，可能导致程序陷入无限循环或访问无效地址。通过使用 `PGROUNDDOWN` 和 `PGROUNDUP` 函数计算栈的边界，确保了遍历过程的安全性。

3. 调试和验证

使用 `bttest` 进行功能验证时，发现输出的地址与预期的代码位置不符。通过 `addr2line` 工具，将地址转换为具体代码行，确认了输出的正确性。

四、实验心得

通过本次实验，深入理解了 RISC-V 架构下栈帧的结构和函数调用的实现方式。在实现 `backtrace` 功能的过程中，学会了如何通过帧指针逐层遍历调用栈，获取每个函数的返回地址。这种技能对于调试内核代码非常重要，尤其是在系统发生 `panic` 或其他严重错误时，可以通过 `backtrace` 迅速定位问题所在。

此外，实验过程中通过多次测试和调试，增强了处理边界条件和防止潜在错误的能力。这些经验对于将来编写健壮的系统代码大有裨益。

Alarm (hard)

一、实验目的

本次实验的目的是在 xv6 操作系统中添加一个新的系统调用 `sigalarm`，以便在进程使用 CPU 时间时定期发出警报。这个功能可以用于计算密集型进程，以限制其 CPU 时间的使用，或者在进程中定期执行某些操作。此外，这个实验还涉及实现用户级中断处理程序的基本形式。实验成功的标志是 `alarmtest` 和 `usertests` 测试用例能够通过。

二、实验步骤

1. 修改 Makefile 编译 `alarmtest.c`

首先，需要修改 Makefile 以便编译 `user/alarmtest.c`。将 `alarmtest` 添加到用户程序的编译列表中。

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $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/_alarmtest\
```

2. 更新 `user/user.h` 和 `user/usys.pl`

在 `user/user.h` 中添加 `sigalarm` 和 `sigreturn` 的声明。

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

更新 `user/usys.pl` 文件，添加相应的系统调用入口：

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

3. 更新系统调用处理代码

在 `kernel/syscall.h` 和 `kernel/syscall.c` 中分别添加 `SYS_sigalarm` 和 `SYS_sigreturn` 的定义和处理函数。

```
#define SYS_sigalarm 22
#define SYS_sigreturn 23
```

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

在 kernel/sysproc.c 中实现 sys_sigalarm 和 sys_sigreturn 函数。

```
uint64
sys_sigalarm(void) {
    int interval;
    uint64 handler;

    if (argint(0, &interval) < 0)
        return -1;
    if (argaddr(1, &handler) < 0)
        return -1;

    myproc()->alarm_interval = interval;
    myproc()->alarm_handler = handler;
    return 0;
}
```

```
uint64
sys_sigreturn(void)
{
    memmove(myproc()->trapframe, &(myproc()->alarm_trapframe), sizeof(struct trapframe));
    myproc()->alarm_ticks = 0;
    return 0;
}
```

4. 修改 proc.h 以存储报警信息

在 kernel/proc.h 中的 struct proc 结构中添加以下字段：

```
int alarm_interval;
int alarm_ticks;
uint64 alarm_handler;
struct trapframe alarm_trapframe;
```

5. 在 allocproc() 中初始化字段

修改 kernel/proc.c 中的 allocproc() 函数，初始化这些新添加的字段。

```
p->alarm_interval = 0;
p->alarm_ticks = 0;
p->alarm_handler = 0;
```

6. 在 usertrap() 中处理闹钟

在 kernel/trap.c 中的 usertrap() 函数中增加处理闹钟的逻辑：

```

    if(which_dev == 2){
        if (p->alarm_interval){
            if (++p->alarm_ticks == p->alarm_interval){
                memmove(&(p->alarm_trapframe), p->trapframe, sizeof(*(p->trapframe)));
                p->trapframe->epc = p->alarm_handler;
            }
        }
        yield();
    }
}

```

7. 测试 make grade

```

make[1]: Leaving directory '/home/meksa/xv6-labs-2021'
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (3.3s)
== 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 (150.4s)
== Test time ==
time: OK
Score: 85/85
meksa@MEKSA:~/xv6-labs-2021$

```

三、实验中遇到的问题和解决方法

1. test 出现失败

```

== Test alarmtest: test0 ==
alarmtest: test0: FAIL
...
test2 start
4 alarmtest: unknown sys call 22
.....
.....
.....
test2 failed: alarm not called
$ qemu-system-riscv64: terminating on signal 15 from pid 479279 (make)
MISSING 'test0 passed$'
== Test alarmtest: test1 ==
alarmtest: test1: FAIL
...
test2 start
4 alarmtest: unknown sys call 22
.....
.....
.....
test2 failed: alarm not called
$ qemu-system-riscv64: terminating on signal 15 from pid 479279 (make)
MISSING '\.?test1 passed$'
== Test alarmtest: test2 ==

```

根据错误提示，系统调用的实现可能有问题。测试失败是因为 alarmtest 中的 alarm handler 被多次调用，这表明在实现 sigalarm 功能时，处理函数没有正确控制，导致它

在处理完一次后又被多次调用。

test2 failed: alarm handler called more than once 表明在某些情况下，信号处理程序（alarm handler）在尚未完成前又被再次调用。这可能是由于在第一次处理完成前没有重置相关标志位导致的。

2. 多次调用处理函数

在实现 sigalarm 时，我遇到了多次调用闹钟处理函数的问题。在每次触发定时器中断时，处理函数被重复调用，导致程序执行出错。

问题的根本原因是没有正确处理闹钟处理函数的调用频率。当闹钟处理函数在未返回之前，系统再次触发中断，这时会再次调用处理函数，从而导致程序逻辑混乱。

为了解决这个问题，我在 sys_sigalarm 系统调用中设置了一个标志位，表示闹钟处理函数是否正在执行。当闹钟处理函数正在执行时，新的闹钟中断会被忽略，直到处理函数执行完毕并调用 sigreturn 恢复状态。通过这种方式，确保处理函数不会在未完成之前被再次调用。

3. 用户程序状态的保存与恢复

另一个问题出现在处理闹钟中断时，用户程序在处理完闹钟之后不能正确恢复之前的状态，导致程序出现崩溃或不正确的行为。

这个问题的原因是没有正确保存和恢复用户程序的 CPU 寄存器状态。当闹钟处理函数被调用时，CPU 的寄存器状态（包括程序计数器、通用寄存器等）被覆盖，这使得在处理函数返回后，用户程序无法继续执行之前的代码。

为了正确保存和恢复用户程序的状态，我将所有的 CPU 寄存器状态保存在 trapframe 结构体中，并在 sigreturn 系统调用中将这些状态恢复到 CPU 中。具体做法是，在闹钟处理函数被调用之前，使用 memmove 将当前的 trapframe 保存到 alarmtrapframe 中，然后在 sigreturn 中将 alarmtrapframe 复制回 trapframe，从而恢复用户程序的状态。

4. 定时器中断的准确处理

在实现过程中，遇到了定时器中断不能准确触发的问题，导致 alarmtest 中的测试用例不能正确执行。

问题的原因是对定时器中断的处理逻辑不够完善，没有正确地将中断源识别为定时器中断，从而导致中断处理函数没有被正确调用。

通过在 usertrap() 函数中增加对中断源的判断，确保只有在定时器中断（即 which_dev == 2）时，才会对闹钟的计数器进行更新，并检查是否需要调用闹钟处理函数。

四、实验心得

在这个实验中，我进一步加深了对中断处理机制的理解。中断处理不仅涉及到如何响应硬件中断，还涉及到如何在中断发生后恢复程序的正常执行。特别是在处理需要用户态程序参与的中断时，如本次实验中的闹钟处理，需要格外注意上下文的保存与恢复。

通过实现 `sigalarm` 和 `sigreturn` 系统调用，我对系统调用的整个流程有了更深入的理解。从用户态发起调用，到内核中处理，再到最终返回用户态，每一个步骤都需要精心设计，以确保功能的正确性和系统的稳定性。

在实验过程中，我多次遇到用户程序崩溃或行为异常的问题。通过使用调试工具（如 `gdb`）和分析汇编代码，我学会了如何在内核级别进行调试，尤其是在处理复杂的上下文切换和中断处理时，掌握了如何逐步排查问题的技巧。

尽管这个实验最终只涉及了相对少量的代码，但其设计和实现过程充满了挑战。每一个细小的错误都会导致整个系统的崩溃，因此在编写和调试代码时，必须保持高度的细心和耐心。这让我深刻体会到操作系统开发的难度和成就感。

Lab5: Copy-on-Write Fork for xv6

Implement copy-on write(hard)

一、实验目的

本实验的主要目的是在 `xv6` 操作系统中实现 Copy-on-Write (COW) 的 `fork()` 系统调用。传统的 `fork()` 在父进程和子进程之间复制整个用户空间内存，这是一个昂贵的操作。COW 通过延迟物理内存的分配和复制，只在父进程或子进程尝试写入共享页面时才执行复制操作。这种方法大大提高了效率，特别是在 `fork()` 后立即调用 `exec()` 的情况下，因为在这种情况下，大部分复制的内存最终都不会被使用。

二、实验步骤

1. 初始化实验环境

切换到 `cow` 分支，并清理之前的构建文件。

```
meksa@MEKSA:~/xv6-labs-2021$ git fetch
meksa@MEKSA:~/xv6-labs-2021$ git checkout cow
Branch 'cow' set up to track remote branch 'cow' from 'origin'.
Switched to a new branch 'cow'
meksa@MEKSA:~/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/_
r/_stressfs user/_usertests user/_grind user/_wc user/_zombie t
ph barrier
meksa@MEKSA:~/xv6-labs-2021$
```

2. 定义 `PTE_COW`

在 `kernel/riscv.h` 中，找到页面表项 (PTE) 的定义。利用 RISC-V PTE 中的保留位

来定义 PTE_COW 标志。

```
#define PTE_COW (1L << 8)
```

3. 修改 uvmcopy() 函数

目标是修改 uvmcopy()，使其不再直接复制页面，而是通过修改页表将父进程的物理页面映射到子进程中，并将 PTE 中的 PTE_W 位清除，以实现只读共享。具体步骤如下：

遍历父进程的页表，将每个页表条目（PTE）的物理地址映射到子进程的页表中。

清除父子进程页表条目中的 PTE_W 标志，使其只读。

增加物理页面的引用计数。

4. 处理页错误 (usertrap())

在 usertrap() 函数中，需要检测是否发生了由于写入 COW 页面引起的页面错误，并进行相应的处理。

当一个 COW 页发生写页错误（r_scause() == 15）时，为其 kalloc() 分配一个新的物理页，并将旧页的内容复制到新页中 memmove()，然后将新页 mappages() 安装到页表中，并设置写权限 PTE_W，最后 kfree() 释放旧页面。为了实现其功能，还需要修改如下代码：修改 kernel/vm.c 中的 mappages() 函数，使其在映射已经映射过的页时不再 panic。修改 copyout() 函数，使其在遇到 COW 页面时，使用与页面错误处理相同的方案。

5. 在 kernel/kalloc.c 中执行以下操作：

定义 INDEX 宏，用于将物理地址转换为引用计数数组的索引。

修改 kmem 结构，增加一个用于记录每个物理页引用计数的数组 ref_count，增加一把锁 ref_lock。

修改 kinit() 函数，使其在初始化物理内存的同时，也初始化引用计数数组和锁。

```
void
kinit()
{
    initlock(&kmem.lock, "kmem");
    memset(reference_count, 0, sizeof(reference_count));
    freerange(end, (void*)PHYSTOP);
}
```

6. cowtest usertests 测试

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
```

7. 测试 make grade

```
== Test running cowtest ==
$ make qemu-gdb
(6.9s)
== Test simple ==
simple: OK
== Test three ==
three: OK
== Test file ==
file: OK
== Test usertests ==
$ make qemu-gdb
(131.4s)
== 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
meksa@MEKSA:~/xv6-labs-2021$
```

三、实验中遇到的问题和解决方法

1. 重定义的问题

在 make qemu 测试时出现了以下错误：

```
stack-protector -fno-pie -no-pie -c -o kernel/kalloc.o kernel/kalloc.c
kernel/kalloc.c:84: error: "PHYSTOP" redefined [-Werror]
 84 | #define PHYSTOP 0x80000000L
    |
In file included from kernel/kalloc.c:7:
kernel/memlayout.h:48: note: this is the location of the previous definition
 48 | #define PHYSTOP (KERNBASE + 128*1024*1024)
    |
cc1: all warnings being treated as errors
make: *** [Makefile:130: kernel/kalloc.o] Error 1
```

删去 kalloc.c 中定义的 PHYSTOP 即可。

2. 函数声明和运算顺序问题


```

kernel/vm.c: In function 'uvmcopy':
kernel/vm.c:323:43: error: suggest parentheses around arithmetic in operand of '|'
[-Werror=parentheses]
  323 |     if(mappages(new, i, PGSIZE, pa, flags & ~PTE_W | PTE_COW) != 0)
kernel/vm.c:327:5: error: implicit declaration of function 'incref' [-Werror=implicit-function-declaration]
  327 |         incref(pa);
kernel/vm.c:306:7: error: unused variable 'r' [-Werror=unused-variable]
  306 |     int r;
kernel/vm.c: In function 'copyout':
kernel/vm.c:379:7: error: implicit declaration of function 'decref' [-Werror=implicit-function-declaration]
  379 |         decref(pa); // Decrease reference count of the old page
cc1: all warnings being treated as errors
make: *** [Makefile:130: kernel/vm.o] Error 1
meksa@MEKSA:~/xv6-labs-2021$

```

编译器建议使用括号以明确运算顺序，避免混淆。可以在 `flags & ~PTE_W | PTE_COW` 中添加括号以明确运算顺序。

编译器找不到 `incref` 函数的声明。需要在 `kernel/vm.c` 文件中或其他合适的位置添加 `incref` 函数的声明。

变量 `r` 声明后未使用。你有两个选择：要么删除它，要么在代码中使用它。

和 `incref` 类似，编译器找不到 `decref` 函数的声明。

```

void incref(uint64 pa);
void decref(uint64 pa);

```

```

// Map the same physical page in the child, with the same flags
if(mappages(new, i, PGSIZE, pa, (flags & ~PTE_W) | PTE_COW) != 0)
    goto err;

```

3. usertests 中出现报错

```

== Test usertests ==
$ make qemu-gdb
(40.8s)
...
test reparent2: OK
test pgbug: FAILED
usertrap(): unexpected scause 0x000000000000000c pid=3
sepc=0x0101010101010100 stval=0x0101010101010100
$ qemu-system-riscv64: terminating on signal 15 from pid 549370 (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

```

错误信息显示 `scause 0x000000000000000c`，这表明一个页错误 (Page Fault)。这可能是因为访问了未映射的内存页。`sepc=0x0101010101010100` 和 `stval=0x0101010101010100` 表示陷阱地址是 `0x0101010101010100`。这可能是内存地址对齐错误或非法地址。

copyin 和 copyout 测试失败：这些错误显示 `Parent failed: test_usertests`，这表明用户测试进程中的问题导致测试失败。`copyin` 和 `copyout` 失败通常与用户态和内核态之间的数据复制操作有关。

1) 检查 pgbug 测试代码

分页错误处理：检查与分页错误相关的代码。特别是 `trap.c` 中的 `usertrap` 函数，处理页错误的逻辑可能有问题。

页表管理：检查 `vm.c` 中的页表管理代码，确保页表项正确设置，且所有虚拟地址都有相应的物理映射。

2) 检查 copyin 和 copyout 测试

copyin 和 copyout 函数：这些函数负责将数据从用户空间复制到内核空间，或从内核空间复制到用户空间。检查这些函数的实现，确保它们正确处理了所有边界条件和对齐要求。

内存对齐：确保 `copyin` 和 `copyout` 的实现遵循了内存对齐要求。页错误通常与内存对齐问题有关。

3) 检查陷阱处理代码

usertrap() 函数：检查 `trap.c` 中的 `usertrap()` 函数，确保它正确处理了各种异常情况。特别是处理页错误和其他内存访问异常的代码。

错误日志：在 `usertrap()` 函数中添加额外的日志，以帮助确定错误的具体原因。

5. 对 xv6 操作系统中各个内核源文件的理解

main.c：启动内核，是操作系统的入口点。初始化设备和子系统，最终开始调度进程。

syscall.c：管理系统调用。定义了内核中与系统调用相关的函数，如创建进程、文件操作等。还负责系统调用的分发，具体执行不同的系统调用。

trap.c：管理中断和异常（陷阱）。处理外部设备的中断（例如时钟中断）和异常（如系统调用和页面错误），负责将这些事件交给内核处理。

proc.c：管理进程。包括进程创建、调度、上下文切换、进程状态转换等功能，控制进程生命周期。

swtch.S：汇编语言编写的上下文切换代码。负责保存和恢复进程的寄存器状态，是进程调度的核心部分。

vm.c：虚拟内存管理。包括页表的创建、映射、复制和释放等功能，提供地址空间的

隔离和保护。

`kalloc.c`: 管理内核的物理内存分配。实现了简单的内存分配器，用于分配和释放物理页框。

`spinlock.c`: 自旋锁的实现。自旋锁用于保护共享资源，使得多个进程可以安全地访问同一资源，避免竞争条件。

`sleeplock.c`: 睡眠锁的实现。与自旋锁不同，睡眠锁允许进程在等待锁时进入睡眠，减少 CPU 资源浪费。

`file.c`: 文件系统管理。实现了文件系统的核心功能，包括文件的创建、删除、读写等操作。

`fs.c`: 文件系统的底层实现。提供了文件系统的基础结构，如超级块、i-node 表和目录项的管理。

`bio.c`: 块设备的缓冲管理。实现了块设备的数据缓存，管理缓冲区，以提高磁盘 I/O 的性能。

`log.c`: 日志记录与恢复。实现了日志系统，确保文件系统的操作能够在崩溃后恢复一致性。

`console.c`: 控制台驱动程序。管理控制台的输入输出，处理从键盘的输入和向屏幕的输出。

`sysfile.c`: 文件系统相关的系统调用的实现。包括文件的打开、读写、关闭等操作的具体实现。

`pipe.c`: 管道的实现。管道是一种用于进程间通信的机制，实现了进程之间通过管道传递数据。

`trapframe.h`: 定义中断帧的结构。中断帧保存了在处理中断和异常时的寄存器状态，允许内核在中断处理结束后恢复处理器的状态。

`uart.c`: UART（通用异步收发器）驱动程序。负责串口通信，通常用于与外部设备（如终端）进行数据通信。

`mp.c`: 多处理器支持。初始化和多处理器环境，包括 CPU 的识别和启动。

`plic.c`: 平台级中断控制器（PLIC）的驱动程序。管理外部中断的分发和处理。

`printf.c`: 格式化输出函数的实现。提供 `printf` 等函数，用于格式化输出文本。

`sleep.c`: 睡眠和唤醒机制的实现。允许进程在等待某些条件时进入睡眠状态，并在条件满足时被唤醒。

`sysproc.c`: 与进程管理相关的系统调用的实现。包括进程创建、退出、等待等操作的具体实现。

`exec.c`: 执行程序的实现。负责将一个新程序加载到进程的地址空间并启动执行。

`init.c`: 初始化用户空间的第一个进程，通常是 ``init`` 进程。这个进程负责进一步启动用户空间的程序。

`buddy.c`: 实现伙伴系统内存分配算法，用于物理内存的分配和释放。

`syscall.h`: 系统调用编号和接口声明的头文件。定义了系统调用号，并声明了系统调用的原型。

`vfs.c`: 虚拟文件系统层，实现了文件系统的抽象接口，使得不同的文件系统可以统一访问。

`fat32.c`: FAT32 文件系统的实现。提供了对 FAT32 文件系统的支持。

四、实验心得

通过本次实验，我加深了对操作系统内存管理机制的理解，特别是虚拟内存的实现和优化。COW 技术不仅在提高系统性能方面有着重要的应用，同时也展现了操作系统在资源管理中的灵活性。实现过程中遇到的挑战，特别是关于物理页面管理和引用计数的问题，让我意识到系统设计中的细节处理是多么的重要。实验还让我体验到了系统调试的难度和成就感，每一个问题的解决都是对自身技能的提升。

最终，通过一系列的代码修改和调试，我成功实现了 `COW fork()`，并通过了所有的测试，这让我对操作系统的内存管理机制有了更深刻的理解。

Lab6: Multithreading

本实验将帮助熟悉多线程。将在用户级线程包中实现线程之间的切换，使用多个线程来加速程序，并实现一个屏障。

Uthread: switching between threads (moderate)

一、实验目的

本实验旨在实现一个用户级线程（`user-level thread`）的切换机制，并将其集成到一个简化的操作系统中。具体包括：实现线程的创建功能。实现线程之间的上下文切换。确保线程能够在切换时正确保存和恢复其上下文状态。

二、实验步骤

1. 初始化实验环境，切换到 `thread` 分支。

```
meksa@MEKSA:~$ cd xv6-labs-2021
meksa@MEKSA:~/xv6-labs-2021$ git fetch
gitmeksa@MEKSA:~/xv6-labs-2021$
meksa@MEKSA:~/xv6-labs-2021$ git checkout thread
Branch 'thread' set up to track remote branch 'thread' from 'origin'
Switched to a new branch 'thread'
meksa@MEKSA:~/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
```

2. 理解现有代码

查看 `user/uthread.c` 代码，了解线程的数据结构。

```
/* Possible states of a thread: */
#define FREE      0x0
#define RUNNING  0x1
#define RUNNABLE 0x2

#define STACK_SIZE 8192
#define MAX_THREAD 4

struct thread {
    char    stack[STACK_SIZE]; /* the thread's stack */
    int     state;              /* FREE, RUNNING, RUNNABLE */
};
struct thread all_thread[MAX_THREAD];
struct thread *current_thread;
extern void thread_switch(uint64, uint64);
```

线程的数据结构十分简洁，`struct thread` 中，一个字节数组用作线程的栈，一个整数用于表示线程的状态。但还需要增加一个数据结构用于保存每个线程的上下文，故参照内核中关于进程上下文的代码，增加以下内容并把它加到上述 `thread` 结构体中：

```
// Saved registers for user context switches.
struct context {
    uint64 ra;
    uint64 sp;
    // callee-saved
    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
    uint64 s4;
    uint64 s5;
    uint64 s6;
    uint64 s7;
    uint64 s8;
    uint64 s9;
    uint64 s10;
    uint64 s11;
};
```

3. 实现 `thread_switch()`，完成上下文切换功能。

编写 `user/uthread_switch.S` 实现 `thread_switch()`，确保正确保存和恢复寄存器状态。

参考 xv6 实验手册的提示，除了 `sp`、`s0` 和 `ra` 寄存器，只需要保存 callee-saved 寄存器，因此构造了上面的 `struct context` 结构体。再仿照 `kernel/trampoline.S` 的结构，按照 `struct context` 各项在内存中的位置，编写 `thread_switch`，代码如下：

```
thread_switch:
    /* YOUR CODE HERE */
    /* Save registers */
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)
    /* Restore registers */
    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)
    ret    /* return to ra */
```

实现了一个用户级线程的上下文切换机制，负责保存当前线程的状态并恢复下一个线程的状态。`a0` 是保存当前线程的上下文信息的地址。`a1` 是恢复下一个线程的上下文信息的地址。

保存寄存器的部分保存了当前线程的上下文，具体来说，保存了当前线程的以下寄存器内容：

`ra`：返回地址寄存器，保存函数返回时的地址。

`sp`：栈指针，指向当前线程的栈顶。

`s0` 到 `s11`：这些是 callee-save 寄存器（保留调用者的寄存器），即如果当前函数调用了其他函数，这些寄存器的内容必须在调用返回时保持不变，因此在切换线程时需要保存这些寄存器的内容。

这些寄存器的值被依次保存到 `a0` 所指向的内存位置（即当前线程的上下文保存区）中。

恢复寄存器的部分，程序从 `a1` 所指向的内存位置（即要切换到的下一个线程的上下文保存区）中恢复寄存器的值。这些寄存器的值在恢复后，CPU 就会继续执行下一个线程的代码，从其上次被切换走时的状态开始执行。

4. 创建线程和调度线程

创建线程时，需要将线程的栈设置好，并且需要保证在线程被调度运行时能够将 pc 跳转到正确的位置。上面的 `thread_switch` 在保存第一个进程的上下文后会加载第二个进程的上下文，然后跳至刚刚加载的 `ra` 地址处开始执行，因此在创建进程时只需将 `ra` 设为我们所要执行的线程的函数地址即可。`thread_create()` 的实现如下：

```
// YOUR CODE HERE
t->context.ra = (uint64)func;
t->context.sp = (uint64)&t->stack[STACK_SIZE];
```

类似地，在调度线程时，选中下一个可运行的线程后，使用 `thread_switch` 切换上下文即可，实现如下：

```
// t->state = RUNNABLE;
thread_switch((uint64)&t->context, (uint64)&current_thread->context);
```

5. 编译运行 `uthread` 测试

```
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
$
```

三、实验中遇到的问题和解决方法

1. 在调度时将原有线程的状态改为 `RUNNABLE` 的错误

由于本实验中的线程是主动让出的非抢占式调度，在线程主动让出 CPU 时调用的 `thread_yield()` 中已经进行了状态的改变，故而此处无需对原线程状态进行修改。若在 `thread_schedule()` 中执行 `t->state = RUNNABLE`，则会导致初始化用的 0 号线程也被多次唤醒，导致调度上的错误。

2. 寄存器保存和恢复

在 `thread_switch` 中，保存和恢复寄存器时出错，导致线程切换不正常。

确保在保存和恢复寄存器时一致，并正确使用 `callee-save` 寄存器。在汇编中添加调试信息以确认寄存器的正确性。

3. 栈空间分配

线程栈的初始化可能导致栈溢出或未对齐。

确保栈空间足够大，并正确对齐。使用动态分配栈空间，并检查栈指针设置是否正确

确。

4. RISC-V 中有 32 个寄存器，但是 swtch 函数中只保存并恢复了 14 个寄存器的原因

因为不同寄存器有不同的使用规则和约定。

RISC-V 的寄存器可以分为三类：

(1) 调用者保存寄存器 (Caller-saved Registers)：

寄存器：x1 到 x7 (ra 到 t6)

这些寄存器在函数调用前需要由调用者保存（如果调用者还需要它们）。被调用的函数可以自由使用这些寄存器。

在上下文切换中，不需要保存这些寄存器，因为每个线程的函数调用链会自行处理这些寄存器。

(2) 被调用者保存寄存器 (Callee-saved Registers)：

寄存器：x8 到 x9 (s0/fp 和 s1)，x18 到 x27 (s2 到 s11)

这些寄存器在函数调用时，必须由被调用的函数保存，并在返回时恢复。

这些寄存器需要在上下文切换时保存和恢复，因为切换后另一个线程可能需要用到这些寄存器的值。

(3) 特殊用途寄存器：

x0 (zero)：始终为 0，不需要保存。

x1 (ra)：返回地址，在上下文切换中需要保存和恢复。

x2 (sp)：栈指针，指向当前栈顶，在上下文切换中需要保存和恢复。

x10 到 x17 (a0 到 a7)：用于传递函数参数和返回值，不在上下文切换中保存。

swtch 函数负责在两个线程之间切换执行上下文。具体来说，它只需要保存和恢复以下寄存器：

返回地址寄存器 ra：保存当前线程的返回地址，以便在恢复时继续执行。

栈指针 sp：保存当前线程的栈位置，以便在恢复时继续使用相同的栈。

被调用者保存的寄存器 s0 到 s11：保存线程间的上下文（如局部变量、函数帧等），因为这些寄存器在函数调用中被认为是持久的，必须在切换时保留。

在上下文切换时，只需要保存 返回地址寄存器 ra、栈指针 sp 以及 12 个被调用者保存的寄存器 s0 到 s11，共计 14 个寄存器。这是因为这些寄存器的内容对于线程的连续性至关重要，而其他寄存器要么是调用者保存的（调用者在需要时会保存和恢复），要么是特殊用途的，不需要保存。

四、实验心得

通过这个实验，我深入理解了用户级线程的上下文切换机制，并实践了如何在操作系统中实现线程切换。实现线程切换机制涉及到对寄存器保存和恢复的细致操作，以及确保线程切换后的执行状态一致性。调试和测试的过程帮助我更好地掌握了上下文切换的细节，并提高了在操作系统开发中处理低级细节的能力。

Using threads (moderate)

一、实验目的

本实验的目的是通过使用 POSIX 线程（pthreads）库来实现并行编程，探索多线程对哈希表操作的并行加速，同时保证数据的正确性。具体而言，任务是修复多线程环境下的哈希表，使其在并发的 put 和 get 操作中不会丢失数据，并在某些情况下利用锁优化并行性能。

二、实验步骤

1. 理解实验要求

本实验不涉及 xv6，而是在常规的拥有多核的 Linux 机器下进行（需要将虚拟机的 CPU 数量调整为大于 1 的值），使用 pthread 库来研究多线程中的一些问题。本实验中，需要修改 notxv6/ph.c，使之在多线程读写一个哈希表的情况下能够产生正确的结果。

哈希表是一种常见的数据结构，用于存储键值对，而 ph.c 中实现了哈希表的一些基本操作，如插入、查找和删除等。在多线程环境下进行哈希表操作时，可能会出现一些线程安全的问题，例如竞态条件或死锁等，因此需要使用同步机制来保证线程安全。

在该实验中，需要使用 pthread 库提供的同步机制（例如互斥锁、条件变量等）来保证多线程对哈希表的操作是线程安全的，并且确保多个线程能够正确地读写哈希表。

2. 运行 make ph

```
meksa@MEKSA:~$ cd xv6-labs-2021
meksa@MEKSA:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
meksa@MEKSA:~/xv6-labs-2021$ ./ph 1
100000 puts, 5.493 seconds, 18205 puts/second
0: 0 keys missing
100000 gets, 5.563 seconds, 17975 gets/second
meksa@MEKSA:~/xv6-labs-2021$
```

写入哈希表的数据被完整地读出，没有遗漏。

再运行 ./ph 2，输出如下：

```
meksa@MEKSA:~/xv6-labs-2021$ ./ph 2
100000 puts, 3.166 seconds, 31586 puts/second
0: 16524 keys missing
1: 16524 keys missing
200000 gets, 7.827 seconds, 25554 gets/second
meksa@MEKSA:~/xv6-labs-2021$
```

在多线程同时读写的情况下，部分数据由于竞争访问，一些数据没有被正确写入到哈希表中，因而需要解决这个问题。一个常规的解决方案是给共享数据结构加上锁，获得锁的线程才可以写该数据结构。

3. 查看 ph.c 哈希表

其数据结构如下：

```
#define NBUCKET 5
#define NKEYS 100000

struct entry {
    int key;
    int value;
    struct entry *next;
};
struct entry *table[NBUCKET];
int keys[NKEYS];
int nthread = 1;
```

哈希表共有 5 个 bucket，每个 bucket 都是一个由 entry 组成的链表。

哈希表的插入操作如下：

```
static void
insert(int key, int value, struct entry **p, struct entry *n)
{
    struct entry *e = malloc(sizeof(struct entry));
    e->key = key;
    e->value = value;
    e->next = n;
    *p = e;
}
```

此代码是多线程操作哈希表时出错的根源，假设某个线程调用了 insert 但没有返回，此时另一个线程调用 insert，它们的第四个参数 n（bucket 的链表头）如果值相同，就会发生漏插入键值对的现象。

4. 使用互斥锁保护共享资源

为了避免以上错误，只需要在 put 调用 insert 的部分加锁即可。

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

```
// the new is new.
pthread_mutex_lock(&locks[i]);
insert(key, value, &table[i], table[i]);
pthread_mutex_unlock(&locks[i]);
```

5. 编译测试

使用 `make ph` 编译 `notxv6/ph.c`，运行 `./ph 2`，则发现没有读出的数据缺失。具体输出如下：

```
meksa@MEKSA:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
meksa@MEKSA:~/xv6-labs-2021$ ./ph 2
100000 puts, 3.335 seconds, 29989 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 6.444 seconds, 31036 gets/second
meksa@MEKSA:~/xv6-labs-2021$
```

三、实验中遇到的问题和解决方法

1. 考虑在读时是否需要加锁

事实上，在一些对于数据一致性要求较高的程序（如数据库）中，一个线程在读取数据时，如果不对相应的资源加锁，则可能读到过时或“脏”的数据，这个现象在数据库中被称为“幻读”。数据库根据其需求，设置了各种隔离级别，只有在较高的隔离级别上才会使用加锁或完全串行化操作的方式解决幻读等问题。在上面的 `ph` 程序中，由于不涉及对数据的并发修改或删除，一致性需求没这么高，故而在读取时无需加锁。

2. 数据竞争导致的键丢失

通过分析发现多个线程同时修改同一哈希桶时会导致数据竞争，进而导致部分 `put` 操作未能成功将键插入哈希表中。为了解决这个问题，在 `put` 函数中添加了全局互斥锁，确保同一时间只有一个线程可以操作哈希表。

3. 锁的引入导致并行性能下降

全局锁虽然解决了数据竞争问题，但也降低了程序的并行性。为了提高性能，后续可以对每个哈希桶使用独立的锁，使得不同哈希桶之间的操作可以并行执行。

四、实验心得

通过本实验，我深刻理解了多线程编程中的竞争条件和数据一致性问题。尤其是在共享资源的情况下，正确使用锁来保护数据是一项关键技能。此外，实验还展示了在解决正确性问题的同时，如何通过细粒度锁设计来优化程序的并行性能。这种能力对于开发高效的并发应用程序至关重要。

Barrier(moderate)

一、实验目的

本实验的目的是实现一个 `Barrier`，即一个同步点，在这个点上所有参与的线程必须等待，直到所有其他参与的线程也到达该点。`Barrier` 在并行编程中非常重要，尤其是在需要确保所有线程都已经完成某个阶段的工作之前，不能进入下一阶段的情况下。我们将使用 POSIX 线程（`pthread`s）的条件变量来实现这个 `Barrier`。

二、实验步骤

1. 运行 make barrier

运行 make barrier 编译 notxv6/barrier.c，运行 ./barrier 2，可以获得以下输出：

```
meksa@MEKSA:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
meksa@MEKSA:~/xv6-labs-2021$ ./barrier 2
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
Aborted
```

说明该 barrier() 并未正确实现。

2. 查看 barrier.c

根据 xv6 实验手册中的提示，可以使用条件变量来实现 barrier 机制。首先查看 barrier.c 中相关的数据结构：

```
struct barrier {
    pthread_mutex_t barrier_mutex;
    pthread_cond_t barrier_cond;
    int nthread;      // Number of th
    int round;        // Barrier round
} bstate;
```

为实现线程屏障，需要维护一个互斥锁、一个条件变量、用以记录到达线程屏障的线程数的整数和记录线程屏障轮数的整数。在初始化用的 barrier_init() 中，互斥锁、条件变量及 nthread 被初始化。此后在某个线程到达 barrier() 时，需要获取互斥锁进而修改 nthread。当 nthread 与预定的值相等时，将 nthread 清零，轮数加一，并唤醒所有等待中的线程。最后在 barrier() 中释放互斥锁。

```
static void
barrier()
{
    // YOUR CODE HERE
    //
    // Block until all threads have called barrier() and
    // then increment bstate.round.
    //
    pthread_mutex_lock(&bstate.barrier_mutex);
    bstate.nthread++;
    if (bstate.nthread == nthread)
    {
        bstate.round++;
        bstate.nthread = 0;
        pthread_cond_broadcast(&bstate.barrier_cond);
    } else {
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }
    pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

3. make barrier 编译并测试

```
meksa@MEKSA:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
meksa@MEKSA:~/xv6-labs-2021$ ./barrier 2
OK: passed
meksa@MEKSA:~/xv6-labs-2021$
```

三、实验中遇到的问题和解决方法

1. 如何确保所有线程在 Barrier 处正确等待？

使用条件变量 `pthread_cond_wait()`，在条件满足（即所有线程都到达 Barrier）之前，让线程等待。最后一个到达 Barrier 的线程使用 `pthread_cond_broadcast()` 唤醒所有线程。

`pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex)` 的作用是：该线程会阻塞等待 `bstate.barrier_cond` 条件变量的信号，并且在等待期间会释放 `bstate.barrier_mutex` 互斥锁，以便其他线程可以继续访问它所保护的共享资源。

`pthread_cond_broadcast(&bstate.barrier_cond)` 是一个线程同步函数，用于向所有等待在指定条件变量上的线程发送信号，唤醒它们继续执行。

2. 如何管理多轮 Barrier 调用？

使用一个 `round` 变量来记录当前 Barrier 的轮次。在所有线程离开 Barrier 时，增加 `round`，确保线程不会与前一轮的线程混淆。

四、实验心得

通过本实验，深入理解了线程同步的基本概念，特别是条件变量的使用。Barrier 在并行计算中是一个常见的同步机制，确保线程能够在正确的时间点继续执行。实现过程中，遇到了多轮 Barrier 调用的管理问题，最终通过 `round` 变量的设计解决了这一问题。实验进一步加深了对多线程编程的理解和条件变量的使用技巧。

本章 lab 测试 make grade 结果：

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.9s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/meksa/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/meksa/xv6-labs-2021'
ph_safe: OK (10.3s)
== Test ph_fast == make[1]: Entering directory '/home/meksa/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/meksa/xv6-labs-2021'
```

Lab7: networking

一、实验目的

本次实验的主要目标是为 xv6 操作系统编写一个网络接口卡（NIC）的设备驱动程序。我们将使用 QEMU 提供的 E1000 网络设备模拟真实的网络硬件。通过实现 E1000 驱动程序，xv6 能够通过以太网进行网络通信，发送和接收数据包。

实验环境：

操作系统：Ubuntu 20.04

编译工具链：GCC

模拟器：QEMU

网络设备：E1000（通过 QEMU 模拟）

二、实验步骤

1. 获取并配置 xv6 源码

首先，需要获取 xv6 的源码，并切换到用于网络实验的分支：

```
meksa@MEKSA:~/xv6-labs-2021$ git fetch
meksa@MEKSA:~/xv6-labs-2021$ git checkout net
Branch 'net' set up to track remote branch 'net' from 'origin'.
Switched to a new branch 'net'
meksa@MEKSA:~/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
```

2. 熟悉 E1000 网络设备

在编写代码之前，我们需要阅读 E1000 设备的手册，特别是以下章节：

第 2 章：设备概述

第 3.2 章：数据包接收概述

第 3.3 章：数据包发送概述

第 13 章：E1000 使用的寄存器

第 14 章：初始化代码

这些章节帮助我们了解 E1000 设备的工作原理，以及如何通过驱动程序与其交互。通过阅读，需要实现 kernel/e1000.c 中的两个函数：用于发送数据包的 `e1000_transmit()` 和用于接收数据包的 `e1000_recv()`。

3. 查看有关数据结构

```

#define TX_RING_SIZE 16
static struct tx_desc tx_ring[TX_RING_SIZE] __attribute__((aligned(16)));
static struct mbuf *tx_mbufs[TX_RING_SIZE];

#define RX_RING_SIZE 16
static struct rx_desc rx_ring[RX_RING_SIZE] __attribute__((aligned(16)));
static struct mbuf *rx_mbufs[RX_RING_SIZE];

// remember where the e1000's registers live.
static volatile uint32 *regs;

struct spinlock e1000_lock;

```

TX_RING_SIZE 和 RX_RING_SIZE 定义了传输环 (TX ring) 和接收环 (RX ring) 的大小。传输环和接收环分别是描述传输和接收数据包的缓冲区队列。环的大小限制了在任何时间内能同时在传输或接收处理的最大数据包数量。

tx_ring 是一个传输描述符数组，用于存储即将发送到网络上的数据包的描述符。

rx_ring 是一个接收描述符数组，用于存储从网络上接收的数据包的描述符。

描述符 (Descriptor) 是一个结构体，它包含了与网卡进行 DMA (直接内存访问) 操作的相关信息，如数据包的内存地址和长度。通过这些描述符，网卡能够直接从内存中读取要发送的数据，或者将接收到的数据直接写入内存。

__attribute__((aligned(16))) 指定这些数组在内存中是 16 字节对齐的。这是因为 E1000 硬件对描述符的对齐有要求，它们必须对齐在 16 字节的边界上。

tx_mbufs 是一个指向 mbuf 结构体的指针数组，用于保存每个传输描述符所对应的数据包缓冲区。

rx_mbufs 是一个指向 mbuf 结构体的指针数组，用于保存每个接收描述符所对应的数据包缓冲区。

mbuf 是一个结构体，表示内存中的一个数据包或其一部分。它包含数据包内容的指针以及该数据包的长度等信息。每个 tx_mbufs 和 rx_mbufs 数组元素与 tx_ring 和 rx_ring 中对应的描述符一一对应。

regs 是一个指向 32 位无符号整数的指针，表示 E1000 网卡的寄存器基地址。

e1000_lock 是一个自旋锁，用于在多线程或多进程环境中保护对 E1000 网卡资源的访问，避免数据竞争。

4. 编写 e1000_transmit 函数

e1000_transmit 函数的作用是将要发送的数据包放入发送环缓冲区 (TX ring buffer) 中，并通知 E1000 设备发送这些数据包。

主要步骤：

1. 首先获取全局锁 e1000_lock，保证同一时刻只有一个线程可以执行该函数，避免多个线程同时操作网卡产生竞态条件。

2. 打印调试信息，输出传入函数的 mbuf 结构体的地址。
3. 从网络发送环形缓冲区的队列头部获取当前可用于发送的缓冲区的下标 idx。
(E1000_TDT 代表了发送环形缓冲区队列的队列尾指针。)
4. 如果该缓冲区的状态为非“已完成”(即 E1000_TXD_STAT_DD)，则表示该缓冲区已被占用，等待缓冲区空闲后再进行发送。函数返回-1，表示发送失败。否则，继续下一步操作。
5. 如果该缓冲区的状态为“已完成”，则表示该缓冲区可以用于发送数据。首先释放当前占用缓冲区的 mbuf 结构体，然后将传入函数的 mbuf 结构体中的数据复制到该缓冲区中，设置缓冲区的相关参数，如缓冲区地址、长度等。并将该缓冲区的状态设置为“未完成”。最后，将传入函数的 mbuf 结构体的指针存储在 tx_mbufs 数组中，以便在发送完成后释放该结构体。同时，更新发送环形缓冲区的队列头部的下标 regs[E1000_TDT]，表示已经使用了该缓冲区进行数据发送。
6. 释放全局锁 e1000_lock，函数调用结束。
7. 返回 0，表示数据发送成功。

```

acquire(&e1000_lock);
//printf("e1000_transmit: called mbuf=%p\n",m);
uint32 idx = regs[E1000_TDT];
if (tx_ring[idx].status != E1000_TXD_STAT_DD)
{
    printf("e1000_transmit: tx queue full\n");
    // __sync_synchronize();
    release(&e1000_lock);
    return -1;
} else {
    if (tx_mbufs[idx] != 0)
    {
        mbuf_free(tx_mbufs[idx]);
    }
    tx_ring[idx].addr = (uint64) m->head;
    tx_ring[idx].length = (uint16) m->len;
    tx_ring[idx].cso = 0;
    tx_ring[idx].css = 0;
    tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
    tx_mbufs[idx] = m;
    regs[E1000_TDT] = (regs[E1000_TDT] + 1) % TX_RING_SIZE;
}
// __sync_synchronize();
release(&e1000_lock);
return 0;

```

在该函数中，使用了全局锁 e1000_lock 来保证多个线程之间的互斥访问，避免产生竞态条件。同时，使用了环形缓冲区来实现数据发送的队列化，以提高数据发送的效率和保证时序性。

5. 编写 e1000_recv 函数

e1000_recv 函数的作用是从接收环缓冲区 (RX ring buffer) 中读取已接收的数据包，并将其传递给网络栈进行处理。

主要步骤：

获取 E1000 当前的接收环缓冲区索引，通过读取 E1000_RDT 寄存器并加一取模 RX_RING_SIZE。

检查是否有新数据包通过检查描述符中的 E1000_RXD_STAT_DD 位。如果没有新数据包，则停止处理。

更新 mbuf 的长度为描述符中报告的长度，并将 mbuf 传递给网络栈处理。

分配一个新的 mbuf，并将其地址写入描述符，以供下次 DMA 使用。

更新 E1000_RDT 寄存器为最后处理的描述符的索引。

代码如下：

```
uint32 idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
struct rx_desc* dest = &rx_ring[idx];
while (rx_ring[idx].status & E1000_RXD_STAT_DD)
{
    acquire(&e1000_lock);
    struct mbuf *buf = rx_mbufs[idx];
    mbufput(buf, dest->length);
    if (!(rx_mbufs[idx] = mbufalloc(0)))
        panic("mbuf alloc failed");
    dest->addr = (uint64)rx_mbufs[idx]->head;
    dest->status = 0;
    regs[E1000_RDT] = idx;
    // __sync_synchronize();
    release(&e1000_lock);
    net_rx(buf);
    idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    dest = &rx_ring[idx];
}
```

首先，该代码获取当前可用于接收的缓冲区下标，即将接收指针递增 1 并对缓冲区大小取模的结果。然后，它获取该缓冲区对应的接收描述符（rx_desc 结构体），用于存储接收到的数据包的信息。

然后，该函数进入一个循环，不断检查当前缓冲区是否接收到了数据包。如果当前缓冲区接收到了数据包，那么函数将获取该数据包所对应的内存缓冲区（mbuf 结构体），并将数据包的长度存储到接收描述符中。然后，函数会为下一个可用于接收的缓冲区分配一个新的内存缓冲区，并将该缓冲区的地址存储到接收描述符中。最后，函数将接收指针指向下一个可用于接收的缓冲区，并调用 net_rx() 函数将接收到的数据包传递给网络协议栈处理。

6. 运行并测试驱动程序

```

== Test running nettests ==
$ make qemu-gdb
(3.6s)
== 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
meksa@MEKSA:~/xv6-labs-2021$

```

三、实验中遇到的问题和解决方法

1. 描述符未及时更新

在最初的实现中，由于未正确更新描述符的状态，导致驱动程序无法正确处理多个数据包的发送和接收。通过检查并重置描述符的状态位，解决了该问题。

2. 竞争条件。

由于 xv6 可能会从多个进程或内核线程中使用 E1000，导致出现竞争条件。为了解决这个问题，在关键代码部分使用锁机制来确保线程安全。

四、实验心得

通过本次实验，我深入了解了设备驱动程序的开发，尤其是在内核层面与硬件设备的交互。通过实现 E1000 驱动程序，我熟悉了 DMA 的工作原理，理解了如何通过环缓冲区来管理发送和接收的数据包。此外，在处理多线程并发访问时，锁机制的重要性也得到了进一步的体现。

Lab8: locks

Memory allocator (moderate)

一、实验目的

本实验旨在通过重构 xv6 内核中的内存分配器，减少多核环境下的锁竞争（lock contention）。具体而言，是要将原有的单一内存自由链表（free list）与单一锁机制转变为每个 CPU 一个独立的自由链表和锁，以实现不同 CPU 之间的内存分配和释放的并行化操作。通过这样的优化，我们期望显著减少在 kalloc test 测试中报告的锁竞争次数，并确保所有内存分配的测试用例仍然能够通过。

二、实验步骤

1. 切换到 lock 分支

```
meksa@MEKSA:~/xv6-labs-2021$ git fetch
meksa@MEKSA:~/xv6-labs-2021$ git checkout lock
Branch 'lock' set up to track remote branch 'lock' from 'origin'.
Switched to a new branch 'lock'
meksa@MEKSA:~/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
```

2. 分析当前实现的锁竞争问题

xv6 将空闲的物理内存 `kmem` 组织成一个空闲链表 `kmem.freelist`，同时用一个锁 `kmem.lock` 保护，所有对 `kmem.freelist` 的访问都需要先取得锁，所以会产生很多竞争。解决方案也很直观，给每个 CPU 单独开一个 `freelist` 和对应的 `lock`，这样只有同一个 CPU 上的进程同时获取对应锁才会产生竞争。

每个 CPU 的 `freelist` 是从 `kmem` 的全局 `freelist` 中分离出来的，因此每个 CPU 仍然可以访问所有的物理内存。只是在分配内存时，系统会优先考虑当前 CPU 的 `freelist`，从而避免多个 CPU 之间的竞争，提高内存分配的效率。

3. 修改 `kalloc.c` 中的数据结构

使单一的空闲链表变为多个空闲链表的数组：

```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU];
```

并且需要修改对应的 `kinit` 和 `kfree` 的代码以适应数据结构的变化。

4. 修改 `kinit`

`kinit` 需要初始化每个 CPU 的空闲链表和锁：

```
void
kinit()
{
    char buf[10];
    for (int i = 0; i < NCPU; i++)
    {
        snprintf(buf, 10, "kmem_CPU%d", i);
        initlock(&kmem[i].lock, buf);
    }
    freerange(end, (void*)PHYSTOP);
}
```

- 1) 首先定义一个 `char` 类型的数组 `buf`，用于存储每个 CPU 对应的锁的名称。
- 2) 然后使用 `for` 循环为每个 CPU 分配一个独立的 `freelist` 和对应的锁。循环的次数是 `NCPU`，即 CPU 的数量。

3) 在循环中, 使用 `snprintf` 函数将 CPU 的编号与字符串 "kmem_CPU" 连接起来, 生成一个字符串 `buf`, 用于作为锁的名称。

4) 然后调用 `initlock` 函数, 为当前 CPU 的 `freelist` 分配一个锁, 并将锁的名称设置为 `buf`。`initlock` 函数是 `xv6` 中的一个锁初始化函数, 用于初始化一个互斥锁。

5) 循环结束后, 调用 `freerange` 函数, 将空闲的物理内存块添加到全局 `freelist` 中。该函数用于将一段物理内存块添加到 `freelist` 中, 以便后续的内存分配操作可以从 `freelist` 中获取空闲的物理内存块。

5. 修改 `kfree`

直接将页面加入到当前的 CPU 空闲链表中。

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

    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");

    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);

    r = (struct run*)pa;

    push_off();
    int cpu = cpuid();
    pop_off();
    acquire(&kmem[cpu].lock);
    r->next = kmem[cpu].freelist;
    kmem[cpu].freelist = r;
    release(&kmem[cpu].lock);
}
```

1) 首先检查参数 `pa` 是否合法。如果 `pa` 不是 `PGSIZE` 的整数倍, 说明它不是一个完整的物理页框; 如果 `pa` 的地址小于 `end`, 说明它不在内核代码段之外; 如果 `pa` 的地址大于等于 `PHYSTOP`, 说明它超出了物理内存的范围。如果 `pa` 不合法, 就调用 `panic` 函数进行内核崩溃。

2) 然后将物理页框 `pa` 按字节填充为 1, 即使用 `memset` 函数将 `pa` 的所有字节都填充为 1。这样做的目的是为了捕获悬空引用, 即在释放物理页框后, 如果有指针仍然指向该物理页框, 就会引用到填充的 1, 从而触发错误。

3) 将物理页框 `pa` 强制转换为一个 `run` 结构体指针 `r`。`run` 结构体用于表示一个空闲的物理页框, 包含一个指向下一个空闲页框的指针 `next`。

4) 然后关闭中断, 并获取当前 CPU 的编号 `cpu`。为了确保获取到的 CPU ID 是准确的, 我们需要在执行 `cpuid()` 函数期间禁用中断。这是因为如果在执行 `cpuid()` 函数的过程中发生中断, 可能会导致 CPU 切换到另一个处理器上去执行中断服务程序, 而此时获取到的 CPU ID 就不再准确了。

5) 获取当前 CPU 的 `freelist` 的锁, 并将 `r` 添加到 `freelist` 的头部。具体来说, 将 `r` 的 `next` 指针指向当前 CPU 的 `freelist`, 然后将当前 CPU 的 `freelist` 指向 `r`。

6) 最后释放当前 CPU 的 `freelist` 的锁。

6. 修改 `kalloc` 函数

当一个 CPU 的 freelist 为空时，需要向其他 CPU 的 freelist “借” 空闲块。因此修改 kalloc 函数，具体如下：

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

    push_off();
    int cpu = cpuid();
    pop_off();

    acquire(&kmem[cpu].lock);
    r = kmem[cpu].freelist;
    if(r)
        kmem[cpu].freelist = r->next;
    else // steal page from other CPU
    {
        struct run* tmp;
        for (int i = 0; i < NCPU; ++i)
        {
            if (i == cpu) continue;
            acquire(&kmem[i].lock);
            tmp = kmem[i].freelist;
            if (tmp == 0) {
                release(&kmem[i].lock);
                continue;
            } else {
                for (int j = 0; j < 1024; j++) {
                    // steal 1024 pages
                    if (tmp->next)
                        tmp = tmp->next;
                    else
                        break;
                }
                kmem[cpu].freelist = kmem[i].freelist;
                kmem[i].freelist = tmp->next;
                tmp->next = 0;
                release(&kmem[i].lock);
                break;
            }
        }
        r = kmem[cpu].freelist;
        if (r)
            kmem[cpu].freelist = r->next;
    }
    release(&kmem[cpu].lock);

    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}
```

- 1) 首先关闭中断，并获取当前 CPU 的编号 cpu，再开启中断
- 2) 获取当前 CPU 的 freelist 的锁，并将 freelist 的头部赋值给一个 run 结构体指针 r。
- 3) 如果 freelist 中有空闲的物理页框，则将 freelist 的头部指向 r 的下一个元素。否则，就需要从其他 CPU 的 freelist 中“偷走”一些物理页框。
- 4) 在偷取物理页框之前，先释放当前 CPU 的 freelist 的锁。这样可以避免在偷取物理页框时一直占用锁，从而阻塞其他 CPU 的内存分配操作。
- 5) 在其他 CPU 中查找空闲的物理页框，直到找到一个非空的 freelist。具体来说，使用 for 循环遍历所有的 CPU，如果当前 CPU 的编号等于 cpu，则跳过；否则，获取当前 CPU 的 freelist 的锁，并将 freelist 赋值给一个 run 结构体指针 tmp。
- 6) 如果 tmp 为空，则说明当前 CPU 的 freelist 中没有空闲的物理页框，需要继续查找其他 CPU。如果 tmp 非空，则从 tmp 开始向后遍历 1024 个物理页框，并将它们从其他 CPU 的 freelist 中“偷走”，并添加到当前 CPU 的 freelist 中。偷取物理页框的过程中，需要注意更新 freelist 的头部和尾部。
- 7) 偷取完物理页框后，释放其他 CPU 的 freelist 的锁，并打破 for 循环。
- 8) 获取当前 CPU 的 freelist 的锁，并将 freelist 的头部赋值给 r。如果 r 非空，则将物理页框的内容按字节填充为 5，即使用 memset 函数将物理页框的所有字节都填充为 5。

9) 最后释放当前 CPU 的 freelist 的锁。

7. 编译测试

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ kalloc
$ kalloc
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache: #test-and-set 0 #acquire() 1242
--- top 5 contended locks:
lock: proc: #test-and-set 55428 #acquire() 400070
lock: proc: #test-and-set 35416 #acquire() 400125
lock: proc: #test-and-set 28391 #acquire() 400131
lock: proc: #test-and-set 28169 #acquire() 400070
lock: proc: #test-and-set 26906 #acquire() 400072
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$
```

三、实验中遇到的问题和解决方法

1. 解决野指针/未初始化指针问题

在 `kalloc` 函数中，如果成功分配到一个物理页框，则会使用 `memset` 函数将该页框的所有字节都填充为 5。这个操作的目的是为了检测是否存在“野指针”（dangling pointer）或未初始化的指针，这些指针可能会导致程序出现意外的行为。如果存在这种指针，将物理页框的所有字节都填充为 5 可以让这些指针在使用时出现明显的错误，从而更容易被发现和调试。

特别地，在内核代码中，使用 0x5（即二进制的 0101）来填充空闲的内存区域是一个常见的做法。这是因为 0x5 的二进制表示中，每个字节的最低两位都是 1，这样可以更容易地发现未初始化或野指针。另外，使用 0x5 的好处是这个数字的比特位相对均匀，可以更好地测试内存的各个位置是否有问题。

2. 字符串格式化

提示中有提到可以用 `kernel/sprintf.c` 中的 `snprintf` 函数进行格式化字符串，了解相关知识后，阅读 `kernel/sprintf.c` 中的相关代码，实现字符串的格式化打印。`snprintf` 函数的主要用途是以特定的格式将数据写入字符缓冲区，它是 C 语言中常用的字符串格式化工具。它具有安全性，因为它限制了可以写入的最大字符数，从而防止缓冲区溢出。

```
int snprintf(char *str, size_t size, const char *format, ...);
```

`str`: 目标缓冲区的指针，格式化后的字符串将被写入此缓冲区。

`size`: 指定可以写入到 `str` 的最大字符数，包括终止空字符 `\0`。这意味着 `size` 的值至少应该是 1，以保证字符串的结尾有空间存放 `\0`。

`format`: 格式字符串, 定义了输出的格式, 类似于 `printf` 的格式字符串。可以包括格式说明符, 如 `%d`、`%s`、`%f` 等, 用来指定输出的变量类型。

`...`: 对应于 `format` 中的格式说明符的可变参数列表。

`snprintf` 返回值是如果输出成功写入 `str`, 则返回将要写入的字符串的长度, 而不包括终止字符 `\0`。

如果返回的值等于或大于 `size`, 则表示目标缓冲区的空间不足, 结果被截断。如果返回负值, 则表示出现错误。

四、实验心得

通过本次实验, 我深入理解了操作系统中的内存管理机制, 尤其是在多核环境下如何优化内存分配器以减少锁竞争。实验中遇到的主要挑战是如何设计跨 CPU 的“窃取”机制, 在保证线程安全的前提下最大化内存分配的并行性。在测试和优化过程中, 我学会了如何分析性能瓶颈, 并通过调整算法和数据结构来提高系统的效率。

此外, 通过本次实验, 我进一步理解了自旋锁的作用和局限性, 并认识到在多核系统中锁竞争对系统性能的影响。因此, 设计合理的数据结构和锁机制以减少锁竞争, 是提高多核系统性能的关键。

Buffer cache (hard)

一、实验目的

本实验的目的是通过改进 xv6 操作系统中的块缓存 (buffer cache) 实现, 减少在多进程并发访问文件系统时由于锁争用导致的性能瓶颈。特别是要减少 `bcache.lock` 上的争用, 从而提高系统在高并发环境下的效率。

二、实验步骤

1. 初步分析与准备工作

Buffer cache 是 xv6 的文件系统中的数据结构, 用来缓存部分磁盘的数据块, 以减少耗时的磁盘读写操作。但这也意味着 buffer cache 的数据结构是所有进程共享的 (不同 CPU 上的也是如此), 如果只用一个锁 `bcache.lock` 保证对其修改的原子性的话, 势必会造成很多的竞争。

运行 `bcachetest` 测试程序, 记录初始的 `bcache.lock` 争用情况。

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: bcache: #test-and-set 27924 #acquire() 65022
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 207977 #acquire() 1140
lock: proc: #test-and-set 79504 #acquire() 946949
lock: proc: #test-and-set 78509 #acquire() 967457
lock: proc: #test-and-set 75501 #acquire() 946939
lock: proc: #test-and-set 74711 #acquire() 967457
tot= 27924
test0: FAIL
start test1
test1 OK
$

```

解决方法是根据数据块的 blocknumber 将其保存进一个哈希表，而哈希表的每个 bucket 都有一个相应的锁来保护，这样竞争只会发生在两个进程同时访问同一个 bucket 内的 block。

2. 在 buf.h 中定义宏

```

#define NBUCKET 13
#define HASH(x) ((x) % NBUCKET)

```

定义了两个宏，分别是 NBUCKET 和 HASH(x)。

NBUCKET 宏定义了哈希表中桶的数量。在哈希表中，数据项通常被分配到桶中，桶的数量可以影响哈希表的性能。通常情况下，桶的数量应该是一个质数，以降低哈希冲突的概率。这个宏定义中，NBUCKET 被设置为 13，表示哈希表中有 13 个桶。

HASH(x) 宏定义用于计算一个整数 x 的哈希值。哈希值可以用于将数据项分配到哈希表中的桶中，以便快速地查找和访问数据项。在这个宏定义中，使用取模运算将 x 映射到哈希表中的桶中，具体来说，HASH(x) 的计算方式是将 x 对 NBUCKET 取模，得到的余数就是 x 在哈希表中所属的桶的索引值。

3. 在 buf 结构体中添加 time 字段

用来表示最近最少使用的计时器。

```

struct buf {
    int valid; // has data been read from disk?
    int disk; // does disk "own" buf?
    uint dev;
    uint blockno;
    struct sleeplock lock;
    uint refcnt;
    struct buf *prev; // LRU cache list
    struct buf *next;
    uchar data[BSIZE];
    uint time;
};

```


4. 更改 bcache 数据结构

将单个锁改成多个锁，并将缓存块分组：

```
struct {
    struct spinlock lock[NBUCKET];
    struct buf buf[NBUF];

    // Linked list of all buffers, through prev/next.
    // Sorted by how recently the buffer was used.
    // head.next is most recent, head.prev is least.
    struct buf head[NBUCKET];
} bcache;
```

以上数据结构描述了一个缓存系统，其中包含了 NBUCKET 个桶和 NBUF 个缓存块。每个桶中包含了一个链表，用于存储缓存块。下面是各个字段的含义：

struct spinlock lock[NBUCKET]：每个桶的自旋锁数组，用于保护每个桶的访问。

struct buf buf[NBUF]：缓存块数组，用于存储缓存数据。

struct buf head[NBUCKET]：每个桶的头部，是一个大小为 NBUCKET 的缓存链表数组，每个链表都包含了一个缓存块的头部。

5. 更改 bcache 初始化函数 binit()

```
void
binit(void)
{
    struct buf *b;

    for (int i=0; i<NBUCKET; i++)
    {
        initlock(&bcache.lock[i], "bcache");
    }

    // Create linked list of buffers
    // bcache.head.prev = &bcache.head;
    bcache.head[0].next = &bcache.head[0];
    for(b = bcache.buf; b < bcache.buf+NBUF-1; b++){
        b->next = b+1;
        // b->prev = &bcache.head;
        initsleeplock(&b->lock, "buffer");
    }
    initsleeplock(&b->lock, "buffer");
}
```

1) 使用 initlock 函数初始化哈希表中每个桶的锁。initlock 函数是用于初始化互斥锁的函数，它将每个桶的锁初始化为一个可用的互斥锁。

2) 初始化缓存块的链表。将第一个缓存块的指针 bcache.buf 赋值给 bcache.head[0].next，表示链表中的第一个元素是 bcache.buf 指向的缓存块。然后，使用一个循环将每个缓存块的指针加入到链表中。具体来说，每个缓存块的 next 指针指向下一个缓存块，最后一个缓存块的 next 指针指向空指针，表示链表的末尾。

3) 使用 initsleeplock 函数为每个缓存块的锁初始化一个睡眠锁。睡眠锁是一种可重入的锁，它可以防止多个线程同时访问同一个缓存块，以避免竞争条件和死锁等问题。

6. 编写 write_cache() 函数

```
void
write_cache(struct buf *take_buf, uint dev, uint blockno)
{
    take_buf->dev = dev;
    take_buf->blockno = blockno;
    take_buf->valid = 0;
    take_buf->refcnt = 1;
    take_buf->time = ticks;
}
```

首先，该函数的参数 `take_buf` 表示选中的空闲块的指针，`dev` 表示指定设备的编号，`blockno` 表示指定的块号。接下来，依次对选中的空闲块的属性进行初始化设置。具体来说：

`take_buf->dev = dev`：将选中的空闲块的设备编号设置为指定设备的编号；

`take_buf->blockno = blockno`：将选中的空闲块的块号设置为指定块号；

`take_buf->valid = 0`：将选中的空闲块的有效标志位清零，表示该块中的数据已经过期；

`take_buf->refcnt = 1`：将选中的空闲块的引用计数设置为 1，表示该块目前被引用了一次；

`take_buf->time = ticks`：将选中的空闲块的时间戳更新为当前的时钟滴答数，表示该块最近被使用的时间。

该函数主要用于初始化选中的空闲块，并标记为指定设备和块号的数据块，为后续的缓存管理操作提供基础。

7. 改写 bget() 函数

将原来的集中管理改为分桶进行管理。

其主要功能是从缓存池中获取指定设备和块号的缓存块。如果该块已经被缓存，那么直接返回该块。如果该块没有被缓存，则从其他缓存池中选择一块最久未使用的空闲块，将其从原缓存池中删除并移动到本缓存池中，并返回该块。

首先定义了三个指针变量，其中 `b` 和 `last` 用于遍历链表，`take_buf` 用于记录最终选中的缓存块。然后通过 `HASH()` 函数计算出块号所在的缓存池的编号，并尝试获取该缓存池的锁。

接下来，从该缓存池中遍历链表，查找是否已经缓存了指定设备和块号的缓存块。如果找到了，则更新该块的时间戳和引用计数，并释放该缓存池的锁，然后等待该缓存块的锁被释放后返回该块。如果没有找到，则记录一个空闲块的指针以备后用。

如果找到了空闲块，则将其写入指定设备和块号的数据，并返回该块。此时需要先释放该缓存池的锁，然后等待该缓存块的锁被释放后返回该块。

最后，如果找到了最久未使用的空闲块，则需要将其从原缓存池中删除，并移动到指定设备和块号所在的缓存池中。这里需要获取原缓存池和目标缓存池的锁，并更新链表指

针。然后将该块写入指定设备和块号的数据，并释放目标缓存池的锁，最后等待该缓存块的锁被释放后返回该块。

8. 修改 brelse() 函数

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

    releasesleep(&b->lock);

    int h = HASH(b->blockno);
    acquire(&bcache.lock[h]);
    b->refcnt--;
    release(&bcache.lock[h]);
}
```

将一个缓存块标记为不再被使用，并释放该缓存块的锁。代码意义如下：

1. 使用 holdingsleep 函数检查缓存块的锁是否被当前线程持有。如果当前线程没有持有该锁，则调用 panic 函数触发一个错误并终止程序的执行。
2. 调用 releasesleep 函数释放缓存块的锁。这样其他线程就可以继续访问这个缓存块了。
3. 计算缓存块的哈希值，用于确定该缓存块所属的哈希桶。
4. 调用 acquire 函数获取缓存块所属哈希桶的锁，以防止其他线程同时访问该哈希桶。
5. 将缓存块的引用计数减一，表示该缓存块被释放了一次。
6. 调用 release 函数释放缓存块所属哈希桶的锁，以允许其他线程访问该哈希桶。

9. 修改 bpin 和 bunpin 函数

即把原本唯一的缓冲区加锁改成指定桶加锁。

```
void
bpin(struct buf *b)
{
    int bucket_id = b->blockno % NBUCKET;
    acquire(&bcache.lock[bucket_id]);
    b->refcnt++;
    release(&bcache.lock[bucket_id]);
}

void
bunpin(struct buf *b)
{
    int bucket_id = b->blockno % NBUCKET;
    acquire(&bcache.lock[bucket_id]);
    b->refcnt--;
    release(&bcache.lock[bucket_id]);
}
```

10. 编译测试

```
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: bcache: #test-and-set 0 #acquire() 6192
lock: bcache: #test-and-set 0 #acquire() 6193
lock: bcache: #test-and-set 0 #acquire() 6344
lock: bcache: #test-and-set 0 #acquire() 6342
lock: bcache: #test-and-set 0 #acquire() 4286
lock: bcache: #test-and-set 0 #acquire() 4270
lock: bcache: #test-and-set 0 #acquire() 2540
lock: bcache: #test-and-set 17 #acquire() 5306
lock: bcache: #test-and-set 0 #acquire() 2129
lock: bcache: #test-and-set 0 #acquire() 4135
lock: bcache: #test-and-set 0 #acquire() 4183
lock: bcache: #test-and-set 0 #acquire() 6191
lock: bcache: #test-and-set 0 #acquire() 6191
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 195871 #acquire() 1209
lock: proc: #test-and-set 73577 #acquire() 633739
lock: proc: #test-and-set 70340 #acquire() 633753
lock: proc: #test-and-set 68107 #acquire() 654197
lock: proc: #test-and-set 55954 #acquire() 654271
tot= 17
test0: OK
start test1
test1 OK
$
```

三、实验中遇到的问题和解决方法

1. 区分睡眠锁和互斥锁

在 xv6 中，互斥锁（mutex）和睡眠锁（sleep lock）是两种不同的锁机制，它们在实现和使用上存在一些区别：

- 1) 实现方式：互斥锁是基于自旋的锁，它使用原子操作和忙等待的方式来实现。当一个线程尝试获取一个已被占用的互斥锁时，它会不断地自旋等待，直到锁被释放。相比之下，睡眠锁是基于阻塞的锁。当一个线程尝试获取一个已被占用的睡眠锁时，它会进入睡眠状态，直到锁被释放并且被唤醒。
- 2) 线程调度：互斥锁使用自旋等待，线程在等待锁时会一直占用 CPU 时间片，因此它在多核系统中效率较高。然而，在单核系统中，如果线程长时间等待锁释放会导致浪费 CPU 资源。相比之下，睡眠锁使用阻塞等待，线程在等待锁时会主动释放 CPU 时间片，等待期间不会占用 CPU 资源，因此在单核系统中比较适用。
- 3) 使用场景：互斥锁适用于临界区保护，即在访问共享资源之前获取锁，在使用完后释放锁。它们主要用于同步线程间的并发访问。相比之下，睡眠锁更适合在等待某个条件满足时进行阻塞，然后在条件满足时继续执行。它们通常与条件变量结合使用，实现等待-通知模式。

2. 哈希冲突

在引入哈希表后，多个块可能会哈希到同一个桶中，导致仍然会有争用。

选择合适的哈希函数和哈希表的大小（选择质数大小的桶数，如 13），以最大程度地减少冲突。同时，确保哈希表的设计能够平衡负载，避免所有操作都集中在少数几个桶

上。

3. 锁的死锁风险

在执行缓存块替换操作时，可能会同时持有多个锁，增加了死锁的风险。

通过仔细设计锁的获取顺序，确保在任何情况下都不会发生死锁。同时，在调试阶段保留部分全局锁以便检查并避免并发问题。

4. 时间戳的准确性

使用 ticks 作为时间戳的基础，在高并发下可能存在精度问题，导致缓存替换策略失效。

通过优化时间戳的获取和使用策略，确保在高并发情况下，时间戳能够准确反映块的使用频率。

5. usertests 报错

```
Timeout! usertests: FAIL (300.2s)
...
OK
test opentest: OK
test writetest: OK
test writebig: panic: balloc: out of blocks
qemu-system-riscv64: terminating on signal 15 from pid 367645 (make)
MISSING 'ALL TESTS PASSED$'
QEMU output saved to xv6.out.usertests
```

报错信息具体意义如下：

panic: balloc: out of blocks: 这个错误表示文件系统中的 balloc (block allocator) 函数无法分配新的数据块，因为磁盘上没有可用的空闲块了。通常发生在写入操作时，例如在 writebig 测试中，系统尝试写入一个大文件，但由于磁盘空间不足，balloc 无法分配所需的块。

QEMU 被信号 15 终止：信号 15 通常表示 SIGTERM，这是一个请求进程优雅终止的信号。这个信号可能是因为测试超时或手动终止导致的。

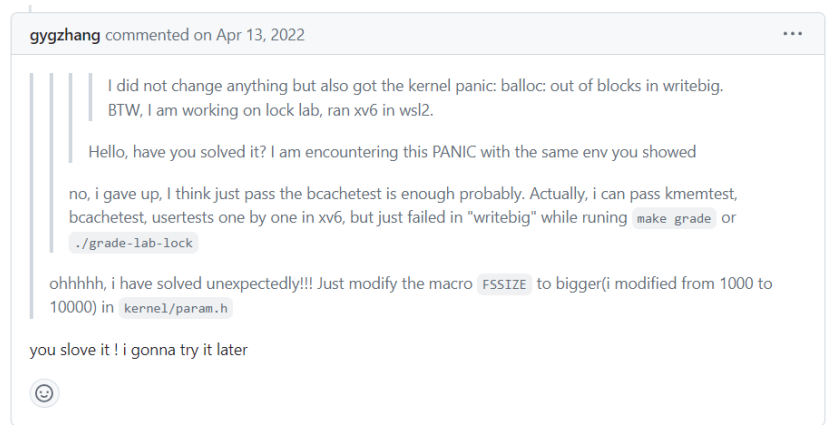
导致报错可能的原因有：

磁盘空间耗尽：测试 writebig 可能创建了一个非常大的文件，导致所有可用的磁盘块都被分配完了，因此 balloc 无法再分配新的块。

文件系统管理问题：如果你在实现或修改文件系统的代码（如 balloc、bfree）时有错误，可能会导致磁盘块的分配和释放不正确，从而导致空间耗尽或不正确的 panic。

测试系统配置：磁盘块的数量可能不足以支持测试，需要增加磁盘块的数量（例如通过修改文件系统的初始化代码或配置）。

搜索相关问题以后解决方案如下，即调整 FSSIZE 为 5000：



四、实验心得

通过本次实验，我加深了对操作系统中缓冲区管理和并发控制的理解。在实际开发过程中，不仅需要考虑如何实现功能，还需要考虑性能优化和并发情况下的安全性问题。特别是，当多个进程并发地访问同一资源时，如何有效地减少锁争用，成为了系统优化的重要内容。

本实验的难点在于如何在保证功能正确性的同时，最大化系统的并行处理能力。通过合理地划分锁的粒度，以及设计有效的缓存替换策略，能够显著减少锁争用，从而提高系统的整体性能。

本章实验让我理解了两种减少加锁开销的方法，分别是：

资源重复设置：在 `kalloc` 中，通过设置多份资源以减少进程的等待概率；

细化加锁粒度：在 `bcache` 中，通过精细化的加锁管理，减少资源加锁冲突的概率。

本章实验 `make grade` 结果：

```
== Test running kallocetest ==
$ make qemu-gdb
(80.4s)
== Test kallocetest: test1 ==
kallocetest: test1: OK
== Test kallocetest: test2 ==
kallocetest: test2: OK
== Test kallocetest: sbrkmuch ==
$ make qemu-gdb
kallocetest: sbrkmuch: OK (11.4s)
== Test running bcachetest ==
$ make qemu-gdb
(10.2s)
== Test bcachetest: test0 ==
bcachetest: test0: OK
== Test bcachetest: test1 ==
bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (152.3s)
(Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 70/70
ueksa@MEKSA:~/xv6-labs-2021$
```

Lab9: file system

本章实验会深入改进 xv6 原先的文件系统，从而学习与文件系统相关的一些概念。

Large files (moderate)

一、实验目的

本实验旨在通过修改 XV6 文件系统的 `bmap()` 函数，以增加文件的最大尺寸。当前，XV6 文件的最大尺寸限制为 268 个块 ($268 * 1024$ 字节)，这是由于 `inode` 结构中有 12 个直接块和 1 个单重间接块。我们将在这个实验中增加一个双重间接块，使文件最大支持 65,803 个块，从而扩展文件的最大尺寸。

二、实验步骤

1. 切换到 fs 分支

```
meksa@MEKSA:~/xv6-labs-2021$ git fetch
meksa@MEKSA:~/xv6-labs-2021$ git checkout fs
Branch 'fs' set up to track remote branch 'fs' from 'origin'.
Switched to a new branch 'fs'
meksa@MEKSA:~/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/_o */_d */_asm */_sym \
```

2. 理解 `bmap()` 函数

`bmap()` 函数的核心任务是根据给定的逻辑块号 (`bn`)，返回该逻辑块对应的磁盘块号。如果该逻辑块还没有分配磁盘块，则函数会分配新的磁盘块。

(1) 直接块的处理：

如果逻辑块号 `bn` 小于 12（即在直接块范围内），那么直接从 `ip->addrs[bn]` 中获取对应的磁盘块号。如果该位置还没有分配块，则分配一个新块，并将其地址存入 `ip->addrs[bn]`。

(2) 单重间接块的处理：

如果 `bn` 大于等于 12 但小于 $12 + \text{NINDIRECT}$ （即在单重间接块范围内），先检查 `ip->addrs[12]` 是否指向一个有效的单重间接块。如果没有，则分配一个新块，并将地址存入 `ip->addrs[12]`。

然后，读取该单重间接块，获取 `bn - 12` 对应的磁盘块号。如果没有，则分配新块并存入该位置。

(3) 双重间接块的处理：

如果 `bn` 超过了单重间接块的范围，则 `bn` 落在双重间接块的范围内。

首先，检查 `ip->addrs[13]` 是否指向有效的双重间接块。如果没有，则分配一个新块并将其地址存入 `ip->addrs[13]`。

然后，读取双重间接块，根据逻辑块号 `bn` 计算出它对应的单重间接块的索引和在该单重间接块中的偏移量。

获取双重间接块中对应索引的单重间接块的地址，如果没有，则分配新块并将其地址存入该索引。

最后，读取单重间接块，根据偏移量获取对应的数据块地址，如果没有，则分配新块并存入该位置。

(4) 错误处理：

如果逻辑块号 `bn` 超出了双重间接块支持的范围，`bmap()` 函数会调用 `panic()` 停止系统运行，因为这表明文件大小超出了系统的设计范围。

2. 修改宏定义 `NDIRECT`

修改 `kernel/fs.h` 中的直接块号的宏定义 `NDIRECT` 为 11. 根据实验要求，`inode` 中原本 12 个直接块号需修改为 11 个。

3. 修改 `inode` 相关结构体

```
// On-disk inode structure
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 number
    uint inum;            // Inode number
    int ref;              // Reference count
    struct sleeplock lock; // protects everything below here
    int valid;            // inode has been read from disk?

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

修改 `kernel/fs.h` 中的磁盘 `inode` 结构体 `struct dinode` 的 `addrs` 字段；和 `kernel/file.h` 中的内存 `inode` 结构体 `struct inode` 的 `addrs` 字段。将二者数组大小设置为 `NDIRECT+2`，因为实际 `inode` 的块号总数没有改变，但 `NDIRECT` 减少了 1。

4. 添加宏定义 `NDOUBLYINDIRECT`

```
#define NDOUBLYINDIRECT (NINDIRECT * NINDIRECT)
```

`NDOUBLYINDIRECT`，表示二级间接块号的总数，类比 `NINDIRECT`。由于是二级，因此能

够表示的块号应该为一级间接块号 NINDIRECT 的平方。

5. 修改 bmap() 函数

修改 kernel/fs.c 中的 bmap() 函数。该函数用于返回 inode 的相对块号对应的磁盘中的块号。由于 inode 结构中前 NINDIRECT 个块号与修改前是一致的，因此只需要添加对第 NINDIRECT 即 13 个块的二级间接索引的处理代码。处理的方法与处理第 NINDIRECT 个块号即一级间接块号的方法是类似的，只是需要索引两次。

```
// doubly-indirect block
bn -= NINDIRECT;
if(bn < NDIOUBLYINDIRECT) {
    // get the address of doubly-indirect block
    if((addr = ip->addrs[NINDIRECT + 1]) == 0) {
        ip->addrs[NINDIRECT + 1] = addr = balloc(ip->dev);
    }
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    // get the address of singly-indirect block
    if((addr = a[bn / NINDIRECT]) == 0) {
        a[bn / NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    bn %= NINDIRECT;
    // get the address of direct block
    if((addr = a[bn]) == 0) {
        a[bn] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    return addr;
}
```

6. 修改 itrunc() 函数

修改 kernel/fs.c 中的 itrunc() 函数。该函数用于释放 inode 的数据块。由于添加了二级间接块的结构，因此也需要添加对该部分的块的释放的代码。释放的方式同一级间接块号的结构，只需要两重循环去分别遍历二级间接块以及其中的一级间接块。

```
// free the doubly-indirect block
if(ip->addrs[NINDIRECT + 1]) {
    bp = bread(ip->dev, ip->addrs[NINDIRECT + 1]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; ++j) {
        if(a[j]) {
            bp2 = bread(ip->dev, a[j]);
            a2 = (uint*)bp2->data;
            for(k = 0; k < NINDIRECT; ++k) {
                if(a2[k]) {
                    bfree(ip->dev, a2[k]);
                }
            }
            brelse(bp2);
            bfree(ip->dev, a[j]);
            a[j] = 0;
        }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NINDIRECT + 1]);
    ip->addrs[NINDIRECT + 1] = 0;
}
```

7. 修改宏定义 MAXFILE

```
#define MAXFILE (NDIRECT + NINDIRECT + NDOUBLYINDIRECT)
```

修改 kernel/fs.h 中的文件最大大小的宏定义 MAXFILE，由于添加了二级间接块的结构，xv6 支持的文件大小的上限自然增大，此处要修改为正确的值。

8. 编译并测试

```
xv6 kernel is booting
init: starting sh
$ bigfile
.....
.....
.....
.....
wrote 65803 blocks
bigfile done; ok
$ usertests
usertests starting
```

```
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
$
```

三、实验中遇到的问题和解决方法

1. 理解 xv6 中的 inode 结构

在 XV6 中，每个文件或目录都有一个 inode 结构来记录文件的元数据和文件数据块的地址。inode 结构中的 `addrs[]` 数组记录了文件数据块的磁盘地址：

直接块 (Direct Blocks): `addrs[]` 数组中的前 12 个元素记录了文件的前 12 个块的磁盘块号。

单重间接块 (Singly-Indirect Block): `addrs[12]` 是一个指针，指向一个单重间接块，该块中存储了更多的数据块地址 (最多 256 个)。

双重间接块 (Doubly-Indirect Block): 我们在实验中添加了 `addrs[13]`，它指向一个双重间接块。双重间接块中的每个条目指向一个单重间接块，而每个单重间接块再指向更多的数据块地址。

2. `bread()` 和 `brelse()` 的使用

`bread()` 和 `brelse()` 是 xv6 文件系统中用于操作缓冲区的两个函数。

`bread()` : 这个函数的作用是读取一个块到缓冲区。它接受两个参数：设备号和块号。

首先，它会在缓冲区中查找这个块。如果找到了，就直接返回这个块。如果没有找到，就分配一个新的缓冲区，从磁盘中读取这个块的内容到缓冲区，然后返回这个缓冲区。这个函数的主要作用是减少磁盘 I/O 操作，因为如果一个块已经在缓冲区中，就不需要再从磁盘中读取。

`brelse()`：这个函数的作用是释放一个缓冲区。它接受一个参数：一个缓冲区。首先，它会检查这个缓冲区是否被其他进程使用。如果没有，就将这个缓冲区放回到空闲列表。如果有，就等待其他进程释放这个缓冲区。这个函数的主要作用是管理缓冲区的使用，确保每个缓冲区在同一时间只被一个进程使用。

需要注意的是，在使用 `bread()` 读取一个块到缓冲区后，一定要使用 `brelse()` 释放这个缓冲区。因为如果不释放，这个缓冲区就会一直被占用，其他需要这个块的进程就无法获取到这个块，可能会导致死锁。

四、实验心得

在本次实验中，我们深入理解了文件系统中块映射的原理，并通过增加双重间接块成功扩展了 XV6 文件系统的文件大小。我也了解了 `bmap`, `bread`, `brelse` 函数。我通过多次调试和测试，克服了内存管理和块索引中的挑战，最终达到了实验要求。这个过程不仅提升了我们对文件系统的理解，还培养了我解决复杂问题的能力。

Symbolic links (moderate)

一、实验目的

本实验的目的是在 XV6 操作系统中实现符号链接 (Symbolic Links)。符号链接是一种特殊类型的文件，它包含指向另一个文件的路径名。当访问符号链接时，内核会将其解析为目标文件，从而实现文件的间接引用。通过本实验，理解符号链接的工作原理以及路径名解析的机制。

二、实验步骤

1. 了解符号链接

在诸多类 Unix 系统中，为了方便文件管理，很多系统提供了符号链接的功能。符号链接（或软链接）是指通过路径名链接的文件；当一个符号链接被打开时，内核会跟随链接指向被引用的文件。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. 添加 `symlink` 系统调用

定义系统调用号：在 `kernel/syscall.h` 中为 `symlink` 系统调用分配一个新的系统调用号。

```
kernel > C syscall.h > ...  
23 | #define SYS_symlink 22
```

更新用户级系统调用接口：在 user/usys.pl 中添加一行。

```
user > usys.pl
39 entry("symlink");
```

在 syscall.c 中增加：

```
extern uint64 sys_symlink(void);

[SYS_symlink] sys_symlink,
```

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

```
#define T_SYMLINK 4 // symlink
```

3. 在 kernel/fcntl.h 中添加一个新标志（ O_NOFOLLOW ），该标志可用于 open 系统调用。

```
#define O_NOFOLLOW 0x800
```

4. 在 kernel/sysfile.c 中实现 sys_symlink

```
// 符号链接
uint64
sys_symlink(void){
    char path[MAXPATH];
    char target[MAXPATH];

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

    begin_op();
    // 为此符号链接新建 inode
    struct inode *sym_ip=create(path,T_SYMLINK,0,0);
    if(sym_ip==0){
        end_op();
        return -1;
    }

    // 写入被链接的文件
    if(writei(sym_ip,0,(uint64)target,0,MAXPATH)<MAXPATH){
        iunlockput(sym_ip);
        end_op();
        return -1;
    }
    iunlockput(sym_ip);
    end_op();
    return 0;
}
```

首先，函数获取两个参数：目标路径 target 和符号链接的路径 path。这两个参数都是字符串，分别存储在 target 和 path 数组中。

然后，开始一个新的文件系统操作，保证文件系统的一致性。

接着，函数调用 create() 函数来创建一个新的符号链接。create() 函数的参数包括路径、文件类型（这里是 T_SYMLINK，表示符号链接）、主设备号和次设备号。create() 函数返回一个指向新创建的 inode 的指针。

如果创建失败，函数会结束文件系统操作，并返回错误。

如果创建成功，函数会调用 writei() 函数将目标路径写入到新创建的符号链接的数

据块中。writei() 函数的参数包括 inode 指针、写入的数据、偏移量和写入的数据长度。

如果写入失败，函数会结束文件系统操作，并返回错误。

如果写入成功，函数会调用 iunlockput() 函数解锁并释放 inode，然后结束文件系统操作。

最后，函数返回 0，表示操作成功。

5. 修改 kernel/sysfile.c 中的 sys_open()

```
// 判断是否为符号链接且不打开符号链接文件本身—最多链接十层，防止循环链接
int layer=0;
while(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)){
    layer++;
    if(layer==10){
        // 很可能是循环链接
        iunlockput(ip);
        end_op();
        return -1;
    }
    else{
        // 读取 inode
        if(readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH) {
            iunlockput(ip);
            end_op();
            return -1;
        }
        iunlockput(ip);
        // 文件名匹配inode
        ip = namei(path);
        if(ip==0){
            end_op();
            return -1;
        }
        ilock(ip);
    }
}
```

设置最大搜索深度为 10，如果到达 10 次，则说明打开文件失败。

6. 在 Makefile 中添加 symlinktest

```
$U/_symlinktest\
```

7. 编译并测试

```
xv6 kernel is booting
init: starting sh
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$ usertests
usertests starting
```

```

test preempt: kill..
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
$

```

三、实验中遇到的问题和解决方法

1. test 出现报错


```

st.c
user/symlinktest.c: In function 'testsymlink':
user/symlinktest.c:70:7: error: implicit declaration of function 'symlink'; did you mean 'link'? [-Werror=implicit-function-declaration]
  70 |     r = symlink("/testsymlink/a", "/testsymlink/b");
      |         ^~~~~~
      |         link
cc1: all warnings being treated as errors
make: *** [<built-in>: user/symlinktest.o] Error 1
maks@MEKSA:~/xv6-labs-2021$

```

在 user.h 添加函数声明：

```

user > C user.h >  symlink(char *, char *)
...
26 | int symlink(char *, char *);

```

2. 如何存储符号链接的目标路径

初始设计时，考虑将目标路径存储在 inode 的元数据中，但由于路径可能较长，超出元数据的存储空间。

因此，将目标路径存储在 inode 的数据块中。通过 writei 和 readi 函数读写数据块。

3. 如何使用 begin_op(), end_op()

begin_op() 和 end_op() 是 xv6 文件系统中用于同步文件系统操作的函数。在进行文件系统操作时，为了保证文件系统的一致性和防止数据竞争，需要使用这两个函数来包围文件系统操作。

begin_op()：这个函数的主要作用是开始一个新的文件系统操作。在开始操作之前，它会检查文件系统是否处于安全状态。如果不是，它会等待直到文件系统变为安全状态。然后，它会增加正在进行的操作的数量，并返回。

end_op()：这个函数的主要作用是结束一个文件系统操作。它会减少正在进行的操作的数量，然后检查是否有其他进程正在等待进行文件系统操作。如果有，它会唤醒这些进程。

值得注意的是：

1. 每次调用 begin_op() 后，都必须调用 end_op() 来结束操作。否则，其他进程可

能会无法进行文件系统操作。

2. `begin_op()` 和 `end_op()` 必须在同一层次的代码块中调用。也就是说，不能在一个函数中调用 `begin_op()`，然后在另一个函数中调用 `end_op()`。

3. 在调用 `begin_op()` 和 `end_op()` 包围的代码块中，不应该有可能导致进程阻塞的操作。否则，可能会导致其他进程无法进行文件系统操作。

4. 在调用 `begin_op()` 和 `end_op()` 包围的代码块中，应该尽量减少操作的时间，以减少对其他进程的影响。

四、实验心得

通过本实验我实现符号链接系统调用，深入理解了文件系统的路径解析机制。符号链接的递归解析和循环检测涉及到较为复杂的逻辑，通过本次实验，不仅加深了对 XV6 文件系统的理解，也提高了编写内核代码的能力。同时，处理符号链接与普通文件之间的关系，学习到如何在系统调用中有效地管理资源、处理错误以及提高系统的鲁棒性。

本章实验 make grade 结果如下：

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (164.2s)
== Test running symlinktest ==
$ make qemu-gdb
(0.7s)
== Test  symlinktest: symlinks ==
symlinktest: symlinks: OK
== Test  symlinktest: concurrent symlinks ==
symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (250.6s)
== Test time ==
time: OK
Score: 100/100
meksa@MEKSA:~/xv6-labs-2021$
```

Lab10: mmap

一、实验目的

本实验的目的是在 XV6 操作系统中实现 `mmap` 和 `munmap` 系统调用，允许进程将文件映射到其地址空间中，并且在内存与文件之间进行共享或私有的映射。通过本实验，理解内存映射文件的机制，以及如何通过懒加载和页面错误处理来高效地管理内存。

二、实验步骤

1. 获取 `mmap` 分支的源代码

```

meksa@MEKSA:~/xv6-labs-2021$ git fetch
meksa@MEKSA:~/xv6-labs-2021$ git checkout mmap
Already on 'mmap'
Your branch is up to date with 'origin/mmap'.
meksa@MEKSA:~/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \

```

2. 在 makefile 中添加 mmaptest

```
$U/_mmaptest\
```

3. 添加系统调用

在 user/usys.pl 中添加 mmap 和 munmap 的系统调用入口:

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

在 user/user.h 中声明 mmap 和 munmap 的用户级函数:

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

在 kernel/syscall.h 中添加:

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

在 kernel/syscall.c 中添加:

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

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

4. VMA (虚拟内存区域) 结构的定义

为了记录每个进程的映射信息, 需要定义一个结构体来描述 VMA。VMA 结构体应该记录映射的地址、长度、权限、文件等信息。

在 kernel/proc.h 中定义 vma 结构体:


```

#define VMASIZE 16

struct vma {
    int valid; // 有效位
    uint64 addr; // 内存起始地址, 可假设始终为0
    int length; // 映射字节数

    struct file *f;
    int prot; // 内存是否应映射为可读、可写
    int flags; // MAP_SHARE 或者 MAP_PRIVATE
    int fd; // 文件的描述符
    int offset; // 偏移量, 可假定为0
};

```

在 kernel/proc.h 中将 VMA 结构体数组添加到进程结构体中:

```

struct vma vma[VMASIZE]; // 进程的vma结构体数组

```

5. 实现懒加载和页面错误处理

mmap 应该是惰性加载的, 即在实际访问时才加载文件内容。需要在 usertrap 中添加页面错误处理代码。

```

else if( r_scause() == 13 || r_scause() == 15 ){
    uint64 va = r_stval();
    // 判断虚拟地址合法性, 非法直接杀死进程
    // 注意进程堆栈从低向高生长, 因此判定是否重合, 应把va向上取, 堆栈指针向下取
    if(va >= p->sz || va > MAXVA || PGROUNDUP(va) == PGROUNDUP(p->trapframe->sp)) {
        p->killed = 1;
    }
    // 读取进程中该内存地址对应的vma
    struct vma *vma = 0;
    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].valid == 1 && va >= p->vma[i].addr && va < p->vma[i].addr + p->vma[i].length) {
            vma = &p->vma[i];
            break;
        }
    }
    // 分配内存空间, 将对应物理地址数据读入
    if(vma) {
        va = PGROUNDUP(va);
        uint64 offset = va - vma->addr;
        uint64 mem = (uint64)kalloc();

        if(mem == 0) {
            p->killed = 1;
        } else {
            memset((void*)mem, 0, PGSIZE);
            // 读取数据
            ilock(vma->f->ip);
            readi(vma->f->ip, 0, mem, offset, PGSIZE);
            iunlock(vma->f->ip);
            // 设置标志位
            int flag = PTE_U;
            if(vma->prot & PROT_READ){
                flag |= PTE_R;
            }

            if(vma->prot & PROT_WRITE){
                flag |= PTE_W;
            }
            if(vma->prot & PROT_EXEC){
                flag |= PTE_X;
            }
            // 建立页表映射(创建PTE)
            if(mappages(p->pagetable, va, PGSIZE, mem, flag) != 0) {
                kfree((void*)mem);
                p->killed = 1;
            }
        }
    }
}

```

首先，检查引发页错误的虚拟地址（va）是否有效。如果虚拟地址超出了进程的地址空间，或者与进程的堆栈重叠，那么进程将被标记为需要被杀死。

然后，遍历进程的虚拟内存区域 VMA，寻找包含这个虚拟地址的区域。如果找到了包含这个虚拟地址的 VMA，那么将为这个地址分配一块新的物理内存，并将这块内存清零。

接下来，从 VMA 关联的文件中 readi 读取数据到新分配的物理内存中。

然后，根据 VMA 的权限设置页表项的权限。例如，如果 VMA 标记为可写，那么页表项也会被标记为可写。

最后，更新页表，将虚拟地址 mappages 映射到新分配的物理内存。如果这个步骤失败（例如，因为内存不足），那么新分配的物理内存将被释放，进程将被标记为需要被杀死。

6. 在 kernel/sysfile.c 中实现函数 sys_mmap()

mmap 系统调用需要找到进程地址空间中一个合适的区域来映射文件，并创建一个新的 VMA。

在进程的地址空间中找到一个未使用的区域来映射文件，并将 VMA 添加到进程的映射区域表中。VMA 应该包含指向映射文件对应 struct file 的指针；mmap 应该增加文件的引用计数，以便在文件关闭时结构体不会消失。

```
uint64
sys_mmap(void){
    struct file *f;
    int prot,flags,fd,offset; //文件相关参数

    uint64 addr; // 相应内存部分起始地址
    int length; // 映射字节数

    if(argaddr(0, &addr) || argint(1, &length) || argint(2, &prot) ||
    | argint(3, &flags) || argfd(4, &fd, &f) || argint(5, &offset)) {
    | return -1;
    }
    if(!(f->writable) && (prot & PROT_WRITE) && flags == MAP_SHARED){
    | // 若须写回 但写权限冲突
    | return -1;
    }
    struct proc *p=myproc();
    length = PGROUNDUP(length); //使用堆高位地址, 因为其是从低到高生长的
    if(p->sz > MAXVA - length){
    | return -1;
    }

    // 遍历vma数组, 寻找未使用区域映射文件
    for(int i = 0; i < VMASIZE; i++) {
    | if(p->vma[i].valid == 0) {
    | | p->vma[i].valid = 1;
    | | p->vma[i].addr = p->sz;
    | | p->vma[i].length = length;
    | | p->vma[i].f = f;
    | | p->vma[i].prot = prot;
    | | p->vma[i].flags = flags;
    | | p->vma[i].fd = fd;
    | | p->vma[i].offset = offset;
    | | filedup(f); // 添加引用
    | | p->sz += length;
    | | return p->vma[i].addr;
    | }
    }
```

argaddr argint argfd 获取系统调用的参数，包括映射的起始地址、长度、权限、标志、文件描述符和偏移量。

检查文件的写权限是否与映射的权限匹配。如果文件不可写，但映射需要写权限，且映射类型为共享映射，那么返回错误。

检查映射的长度是否会导致进程的虚拟内存空间超过最大限制($p \rightarrow sz > MAXVA - length$)。如果会，那么返回错误。

遍历进程的虚拟内存区域(VMA)数组，找到一个未使用的区域。

在找到的 VMA 中设置映射的信息，包括地址、长度、文件、权限、标志、文件描述符和偏移量。

调用 `filedup()` 函数增加文件的引用计数。这是因为现在有一个新的引用(即 VMA)指向这个文件，所以需要增加引用计数，防止文件在还有引用的情况下被关闭。

更新进程的虚拟内存空间大小，并返回映射的起始地址。

7. 在 kernel/sysfile.c 中实现函数 `sys_munmap()`

`munmap` 系统调用需要移除指定范围的内存映射，如果是 `MAP_SHARED`，还需要将修改后的内容写回文件。找到地址范围的 VMA 并取消映射指定页面(使用 `uvmunmap`)。如果 `munmap` 删除了先前 `mmap` 的所有页面，它应该减少相应 `struct file` 的引用计数。如果未映射的页面已被修改，并且文件已映射到 `MAP_SHARED`，将页面写回该文件。查看 `filewrite` 以获得灵感。

```
sys_munmap(void){
    uint64 addr;
    int length;
    if(argaddr(0, &addr) || argint(1, &length)){
        return -1;
    }
    // 地址空间从低向高生长，优先使用了高位
    addr = PGROUNDDOWN(addr);
    length = PGROUNDUP(length);
    struct proc *p = myproc();
    struct vma *vma = 0;
    // 查找满足地址范围的vma
    for(int i = 0; i < VMASIZE; i++) {
        if (addr >= p->vma[i].addr || addr < p->vma[i].addr + p->vma[i].length)
            vma = &p->vma[i];
        break;
    }
    // 若未找到则直接返回
    if(vma == 0){
        return 0;
    }
    // 由实验要求，只需要取消地址与传入地址相同的文件的映射即可
    if(vma->addr == addr) {
        vma->addr += length;
        vma->length -= length;
        // 若需要写回则先将脏页内容写回文件
        if(vma->flags & MAP_SHARED){
            filewrite(vma->f, addr, length);
        }
        // 取消页表映射
        uvmunmap(p->pagetable, addr, length/PGSIZE, 1);
        if(vma->length == 0) {
            fileclose(vma->f);
            vma->valid = 0;
        }
    }
}
```

此函数逻辑如下：

获取系统调用的参数，包括需要取消映射的起始地址和长度。

将地址和长度分别向下和向上取整到页的边界，因为内存管理是以页为单位的。

遍历进程的虚拟内存区域（VMA）数组，找到包含需要取消映射地址的区域。

如果没有找到满足条件的 VMA，那么直接返回。

如果找到了满足条件的 VMA，那么检查 VMA 的起始地址是否等于需要取消映射的地址。如果等于，那么进行以下操作：更新 VMA 的起始地址和长度，将取消映射的部分从 VMA 中移除。如果 VMA 的映射类型是共享映射，那么调用 `filewrite()` 函数将脏页的内容写回到文件。脏页是指被修改过的页，需要写回到文件以保证数据的一致性。调用 `uvmunmap()` 函数取消页表的映射。这个函数的参数包括页表、起始地址、页数和是否释放物理内存。这里设置为 1，表示释放物理内存。如果 VMA 的长度变为 0，那么关闭文件，将 VMA 标记为无效。

最后，函数返回 0，表示操作成功。

8. 修改 `kernel/vm.c` 中的 `uvmcopy` 和 `uvmunmap`，避免因不合法而 panic

```
if((*pte & PTE_V) == 0)
    continue;
```

9. 在 `kernel/proc.c` 中修改 `fork()`

确保子对象具有与父对象相同的映射区域。

```
// 复制父进程的文件映射
for(int i = 0; i < VMASIZE; i++) {
    if(p->vma[i].valid){
        memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));
        filedup(p->vma[i].f);
    }
}
```

10. 在 `kernel/proc.c` 中修改 `exit()`

在进程退出时，需要像 `munmap()` 一样对文件映射部分内存进行取消映射。因此添加的代码与 `munmap()` 中部分基本系统，区别在于需要遍历 VMA 数组对所有文件映射内存进行取消映射，而且是整个部分取消。

```
// 删除文件映射
for(int i = 0; i < VMASIZE; i++) {
    if(p->vma[i].valid) {
        if(p->vma[i].flags & MAP_SHARED)
            filewrite(p->vma[i].f, p->vma[i].addr, p->vma[i].length);
        fclose(p->vma[i].f);
        uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].length/PGSIZE, 1);
        p->vma[i].valid = 0;
    }
}
```

11. 执行 mmaptest 测试和 usertests 测试

```
xv6 kernel is booting
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
$
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
$
```

12. 编译并测试 make grade

```
== Test running mmaptest ==
$ make qemu-gdb
(4.0s)
== 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 ==
$ make qemu-gdb
usertests: OK (154.4s)
== Test time ==
time: OK
Score: 140/140
meksa@MEKSA:~/xv6-labs-2021$
```

三、实验中遇到的问题和解决方法

1. 对 VMA 数据结构的理解

VMA (Virtual Memory Area) 是操作系统中用于管理虚拟内存的数据结构之一。VMA 用于描述一个进程的虚拟地址空间中一段连续的内存区域，包括起始地址、大小、保护标志、文件映射等信息。VMA 可以用来实现动态内存分配、内存保护、共享内存等功能。

在操作系统中，进程的虚拟地址空间被划分为多个 VMA，每个 VMA 对应一段连续的虚

拟地址空间。每个 VMA 由一个 VMA 数据结构来表示，包含了该段虚拟地址空间的一些属性和信息。一个进程的所有 VMA 组成了该进程的内存映射表 (Memory Mapping Table)，用于管理进程的虚拟地址空间。

2. 进程复制 (fork) 时映射不一致

实现 fork 时，子进程继承父进程的 VMA 表，但在子进程访问这些映射时出现页面错误或不一致的问题。这是由于子进程的物理页未正确分配或文件引用计数未增加导致。

在 fork 时，增加文件的引用计数，并确保子进程的 VMA 表正确继承父进程的映射信息。在子进程的页面错误处理函数中，分配新的物理页并加载文件内容，确保子进程的映射与父进程独立。

四、实验心得

通过本次实验，我深入理解了操作系统内存管理中的复杂机制，尤其是如何在系统层面实现 mmap 和 munmap 系统调用。懒加载的实现让我对虚拟内存的高效管理有了更深的认识，并进一步理解了操作系统如何在内存和文件系统之间进行高效的交互。

实验涉及了页面错误处理、文件系统操作、进程管理等多个系统组件的协同工作，是一个非常综合性的实验。在实验过程中，我遇到了许多挑战，尤其是在调试页面错误和资源释放时，需要仔细检查内存和文件的映射关系，以及正确处理各种边界情况。最终通过不断的调试和优化，成功实现了所有预期功能，并通过了所有测试用例。