

Xv6(2020) Labs Report

姓名：林琪
学号：2352609

仓库链接：<https://github.com/lq1911/OS.git>

- Lab0 : Environment Setup
- ▼ Lab1 : Xv6 and Unix utilities

1. boot xv6

- 1.1. 实验目的
- 1.2. 实验步骤

2. sleep

- 2.1. 实验目的
- 2.2. 实验步骤
- 2.3. 实验小结

3. pingpong

- 3.1. 实验目的
- 3.2. 实验步骤
- 3.3. 实验小结

4. primes

- 4.1. 实验目的
- 4.2. 实验步骤
- 4.3. 实验小结

5. find

- 5.1. 实验目的
- 5.2. 实验步骤
- 5.3. 实验小结

6. xargs

- 6.1. 实验目的
- 6.2. 实验步骤
- 6.3. 实验小结

- ▼ Lab2 : system calls

1. System call tracing

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

2. Sysinfo

- 2.1. 实验目的
- 2.2. 实验步骤
- 2.3. 实验小结

- ▼ Lab3 : page tables

▼ 1. Print a page table

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

▼ 2. A kernel page table per process

- 2.1. 实验目的
- 2.2. 实验步骤
- 2.3. 实验小结

▼ 3. Simplify

- 3.1. 实验目的
- 3.2. 实验步骤
- 3.3. 实验小结

▼ Lab4 : traps

▼ 1. RISC-V assembly

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

▼ 2. Backtrace

- 2.1. 实验目的
- 2.2. 实验步骤
- 2.3. 实验小结

▼ 3. Alarm

- 3.1. 实验目的
- 3.2. 实验步骤
- 3.3. 实验小结

▼ Lab5 : lazy page allocation

▼ 1. Eliminate allocation from sbrk

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

▼ 2. Lazy allocation

- 2.1. 实验目的
- 2.2. 实验步骤
- 2.3. 实验小结

▼ 3. Lazytests and Usertests

- 3.1. 实验目的
- 3.2. 实验步骤
- 3.3. 实验小结

▼ Lab6 : Copy-on-Write Fork for xv6

▼ 1. Implement copy-on write

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

▼ Lab7 : Multithreading

1. Uthread: switching between threads

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

2. Using threads

- 2.1. 实验目的
- 2.2. 实验步骤
- 2.3. 实验小结

3. Barrier

- 3.1. 实验目的
- 3.2. 实验步骤
- 3.3. 实验小结

▼ Lab 8 : locks

1. Memory allocator

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

2. Buffer cache

- 2.1. 实验目的
- 2.2. 实验步骤
- 2.3. 实验小结

▼ Lab 9 : file system

1. Large files

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

2. Symbolic links

- 2.1. 实验目的
- 2.2. 实验步骤
- 2.3. 实验小结

▼ Lab 10 : mmap

1. mmap

- 1.1. 实验目的
- 1.2. 实验步骤
- 1.3. 实验小结

▼ Lab 11 : networking

1. networking

- 1.1. 实验目的

- 1.2. 实验步骤
- 1.3. 实验小结

Lab0 : Environment Setup

- 使用 VMware Workstation 创建 Ubuntu 虚拟机

- 安装本项目所需依赖

```
$ 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
```

- 测试安装

```
$ qemu-system-riscv64 --version
```

```
$ riscv64-linux-gnu-gcc --version
```

- 编译内核

下载xv6内核源码: \$ git clone git://github.com/mit-1 pdos/xv6-riscv.git

更新镜像源:

```
$ sudo nano /etc/apt/sources.list
```

```
$ sudo apt-get update
```

Lab1 : Xv6 and Unix utilities

1. boot xv6

1.1. 实验目的

构建、熟悉 xv6 及其系统调用。

1.2. 实验步骤

- 获取 xv6 源代码并切换到 util 分支

```
$ git clone git://g.csail.mit.edu/xv6-labs-2020
```

```
$ cd xv6-labs-2020
```

```
$ git checkout util
```

- 构建并运行 xv6

```
$ make qemu
```

```
lq@vm:~/xv6-labs-2020$ 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 -ggdb -DSOL_UTIL -MD -mcmmodel=medany -f
freestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/s
tart.o kernel/start.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmmodel=medany -f
freestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/c
onsole.o kernel/console.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmmodel=medany -f
freestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/p
rintf.o kernel/printf.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmmodel=medany -f
freestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/u
art.o kernel/uart.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmmodel=medany -f
freestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/k
alloc.o kernel/kalloc.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmmodel=medany -f
freestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/s
pinlock.o kernel/spinlock.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmmodel=medany -f
freestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/s
tring.o kernel/string.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmmodel=medany -f
freestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/m
```

```
$ ls
```

这些是 mkfs 在初始文件系统中包含的文件，大多数是可以运行的程序，刚刚运行的 ls 就是其中一个程序。

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ ls
. 1 1 1024
.. 1 1 1024
README 2 2 2059
xargstest.sh 2 3 93
cat 2 4 24424
echo 2 5 23240
forktest 2 6 13416
grep 2 7 27728
init 2 8 23984
kill 2 9 23192
ln 2 10 23040
ls 2 11 26616
mkdir 2 12 23336
rm 2 13 23328
sh 2 14 42144
stressfs 2 15 24184
usertests 2 16 148608
grind 2 17 38304
wc 2 18 25512
zombie 2 19 22576
console 3 20 0
$
```

\$ Ctrl-a x

退出qemu

2. sleep

2.1. 实验目的

本实验旨在实现 xv6 的 sleep 命令，加深对系统调用、时间管理和命令行参数处理的理解。

2.2. 实验步骤

- 编写程序

在 user 目录下，创建一个名为 sleep.c 的文件，代码如下：

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

int main(int argc, char const *argv[])
{
    if (argc != 2) { //参数错误
        fprintf(2, "usage: sleep <time>\n");
        exit(1);
    }
    sleep(atoi(argv[1]));
    exit(0);
}
```

- 编辑 Makefile

将编写好的 sleep 程序添加到 Makefile 的 UPROGS 中。

打开 Makefile，在 Makefile 中找到名为 UPROGS 的行，添加 sleep 程序的目标名称：\$U/_sleep\。

- 编译并测试程序

```
lq@vn:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file
=fs.1mg,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ sleep 10
$ QEMU: Terminated
```

- 单元测试

```
lq@vn:~/xv6-labs-2020$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.9s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.2s)
```

2.3. 实验小结

本实验成功实现了 xv6 下的 sleep 程序，关键步骤包括解析命令行参数、调用 xv6 的 sleep 系统调用及处理异常输入。结果表明，程序能正确暂停指定时间片后返回，验证了其功能性和正确性，加深了对 xv6 系统调用机制的理解。

通过该实验，掌握了在 xv6 中开发用户程序的方法，理解了时间片调度和系统调用的实现原理，这对研究操作系统内核机制具有重要意义。同时，提升了错误处理、用户输入验证及系统调用使用的能力，为开发更复杂的系统应用奠定了基础。后续可进一步研究 xv6 的其他系统调用，深化对内核的理解。

3. pingpong

3.1. 实验目的

本实验旨在通过编写一个使用 UNIX 系统调用的 pingpong 程序，加深对进程间通信和管道机制的理解。具体目标包括：

1. 掌握管道的基本使用：使用 pipe() 创建两个管道，分别用于父子进程之间的双向通信。
2. 熟悉进程创建：通过 fork() 创建子进程，并区分父子进程的执行逻辑。
3. 实践读写操作：使用 read() 和 write() 在管道中传递数据，确保正确同步。
4. 理解进程间同步：父进程先发送字节，子进程接收后返回响应，验证管道的正确使用。
5. 学习进程管理：使用 getpid() 获取进程 ID，并在输出中区分父子进程。

3.2. 实验步骤

- 编写程序

在 user/ 目录下创建一个名为 pingpong.c 的文件，代码如下：

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

#define RD 0 //pipe的read端
#define WR 1 //pipe的write端

int main(int argc, char const *argv[]) {
    char buf = 'P'; //用于传送的字节

    int fd_c2p[2]; //子进程->父进程
    int fd_p2c[2]; //父进程->子进程
    pipe(fd_c2p);
    pipe(fd_p2c);

    int pid = fork();
    int exit_status = 0;

    if (pid < 0) {
        fprintf(2, "fork() error!\n");
        close(fd_c2p[RD]);
        close(fd_c2p[WR]);
        close(fd_p2c[RD]);
        close(fd_p2c[WR]);
        exit(1);
    } else if (pid == 0) { //子进程
        close(fd_p2c[WR]);
        close(fd_c2p[RD]);

        if (read(fd_p2c[RD], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "child read() error!\n");
            exit_status = 1; //标记出错
        } else {
            fprintf(1, "%d: received ping\n", getpid());
        }

        if (write(fd_c2p[WR], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "child write() error!\n");
            exit_status = 1;
        }

        close(fd_p2c[RD]);
        close(fd_c2p[WR]);

        exit(exit_status);
    } else { //父进程
        close(fd_p2c[RD]);
        close(fd_c2p[WR]);

        if (write(fd_p2c[WR], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "parent write() error!\n");
            exit_status = 1;
        }

        if (read(fd_c2p[RD], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "parent read() error!\n");
            exit_status = 1; //标记出错
        } else {
            fprintf(1, "%d: received pong\n", getpid());
        }

        close(fd_p2c[WR]);
        close(fd_c2p[RD]);

        exit(exit_status);
    }
}
```

```
}
```

- 编辑 Makefile

打开 Makefile，在 Makefile 中找到名为 UPROGS 的行，添加 pingpong 程序的目标名称：\$U/_pingpong\

- 编译并测试程序

```
lq@vm:~/xv6-labs-2020$ make qemu
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mmodel=medany -f
freestanding -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/uvs.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
mkfs/mkfs fs.img README user/xargstest.sh user/_cat user/_echo user/_forktest user/_grep user/_init user/
_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind user/_wc user/_zombie user/_sleep user/_pingpong
nmeta 46 (boot, super) blocks 30 (node blocks 13, bitmap blocks 1) blocks 954 total 1000
alloc: first 646 blocks have been allocated
alloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file
=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

子进程接收到父进程发送的字节并打印其进程 ID 和消息 "received ping"

子进程将字节发送回父进程

父进程接收到来自子进程的字节并打印其进程 ID 和消息 "received pong"

- 单元测试

```
lq@vm:~/xv6-labs-2020$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (0.8s)
```

3.3. 实验小结

1. 进程间通信：在这个实验中，父进程和子进程通过两个管道实现了双向通信。管道是一种 UNIX 提供的 IPC（进程间通信）机制，允许数据在进程之间流动。通过 pipe 系统调用，实验创建了两个管道 p 和 q，分别用于父进程向子进程发送数据以及子进程向父进程返回数据。管道的使用使得进程间的数据传输变得简单有效，同时也展示了 UNIX 系统下进程通信的基础。
2. 在这个实验中，父进程首先写入数据到管道 p，然后等待子进程读取该数据。子进程读取后，打印消息并写回数据到管道 q，父进程再从 q 读取数据，完成整个通信流程。这种严格的执行顺序保证了进程同步，确保了数据的正确传递。
3. 进程同步：进程同步是确保多个进程按照预期顺序执行的关键。在这个实验中，进程同步主要通过管道和 wait 系统调用实现。父进程通过 write 将数据写入管道 p，然后通过 read 从管道 q 读取数据。由于 read 是阻塞操作，父进程会等待直到子进程写入数据到管道 q。这种机制保证了父进程在子进程完成任务之前不会继续执行。此外，父进程通过 wait 系统调用等待子进程结束，进一步确保了进程的同步性。

4. primes

4.1. 实验目的

编写一个使用管道实现的并发版本的素数筛选算法。

使用管道和 fork 来建立管道。第一个进程将数字 2 到 35 送入管道。对于每一个质数，应将安排创建一个进程，通过管道从左邻右舍读取数据，并通过另一个管道向右邻右舍写入数据。由于 xv6 的文件描述符和进程数量有限，第一个进程可以在 35 处停止。

4.2. 实验步骤

- 编写程序：

在 user/ 目录下创建一个名为 primes.c 的文件，代码如下：

```

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

#define RD 0
#define WR 1

const uint INT_LEN = sizeof(int);

/***
 * @brief 读取左邻居的第一个数据
 * @param lpipe 左邻居的管道符
 * @param pfist 用于存储第一个数据的地址
 * @return 如果没有数据返回-1,有数据返回0
 */
int lpipe_first_data(int lpipe[2], int *dst)
{
    if (read(lpipe[RD], dst, sizeof(int)) == sizeof(int)) {
        printf("prime %d\n", *dst);
        return 0;
    }
    return -1;
}

/***
 * @brief 读取左邻居的数据，将不能被first整除的写入右邻居
 * @param lpipe 左邻居的管道符
 * @param rpipe 右邻居的管道符
 * @param first 左邻居的第一个数据
 */
void transmit_data(int lpipe[2], int rpipe[2], int first)
{
    int data;
    // 从左管道读取数据
    while (read(lpipe[RD], &data, sizeof(int)) == sizeof(int)) {
        // 将无法整除的数据传递入右管道
        if (data % first)
            write(rpipe[WR], &data, sizeof(int));
    }
    close(lpipe[RD]);
    close(rpipe[WR]);
}

/***
 * @brief 寻找素数
 * @param lpipe 左邻居管道
 */
void primes(int lpipe[2])
{
    close(lpipe[WR]);
    int first;
    if (lpipe_first_data(lpipe, &first) == 0) {
        int p[2];
        pipe(p); // 当前的管道
        transmit_data(lpipe, p, first);

        if (fork() == 0) {
            primes(p); // 递归的思想，在一个新的进程中调用
        } else {
            close(p[RD]);
            wait(0);
        }
    }
    exit(0);
}

```

```

int main(int argc, char const *argv[])
{
    int p[2];
    pipe(p);

    for (int i = 2; i <= 35; ++i) //写入初始数据
        write(p[WR], &i, INT_LEN);

    if (fork() == 0) {
        primes(p);
    } else {
        close(p[WR]);
        close(p[RD]);
        wait(0);
    }

    exit(0);
}

```

- 编辑 Makefile

打开 Makefile，在 Makefile 中找到名为 UPROGS 的行，添加 primes 程序的目标名称：\$U/_primes\

- 编译并测试程序

```

lq@vm:~/xv6-labs-2020$ make qemu
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mmodel=medany -f
freestanding -fno-common -fno-stdlib -fno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/primes.o user/primes.c
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_primes user/primes.o user/ulib.o user/sys.o user/printfo.o user/malloc.o
riscv64-unknown-elf-objdump -S user/_primes > user/primes.asm
riscv64-unknown-elf-objdump -t user/_primes | sed '1,/SYMBOL TABLE/d; s/.* / /; /$/d' > user/primes.s
mkfs/mkfs fs.img README user/xargstest.sh user/_cat user/_echo user/_forktest user/_grep user/_init user
/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind user/_wc user/_zombie user/_sleep user/_pingpong user/_primes
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
alloc: first 672 blocks have been allocated
alloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file
=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$ QEMU: Terminated

```

- 单元测试

```

lq@vm:~/xv6-labs-2020$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.0s)

```

4.3. 实验小结

本实验采用了基于管道的并发素数筛选算法，通过进程间通信实现数据传递与处理。

具体实现方法如下：

- 管道与进程创建：程序通过调用 pipe() 创建管道，并使用 fork() 创建子进程。父进程负责将数字 2 到 35 写入管道，子进程则负责从管道中读取数据并进行筛选。
- 数据传递与筛选：子进程通过递归调用 filter 函数实现素数的筛选。每个子进程从管道中读取一个数并判断其是否为素数。若为素数，则打印并创建新的管道和子进程，继续筛选剩余数据。这种方法通过进程间的递归调用实现了并发处理，每个进程仅处理一个素数及其倍数的过滤工作，极大地提高了程序的可扩展性和效率。
- 资源管理：在程序中，父进程在完成数据写入后关闭管道的写端，并通过 wait() 等待子进程结束。子进程在读取数据完成后，关闭相应的管道端口，避免了资源泄漏。

优点如下：

- 并发处理：通过使用多个进程和管道实现并发处理，提高了程序的执行效率。
- 代码结构清晰：采用递归调用的方式使得代码结构简洁明了，每个子进程只负责处理一个素数及其倍数的过滤工作。

不足点如下：

- 资源消耗大：每个素数的筛选都需要创建新的进程和管道，随着数据量的增加，系统资源的消耗也会显著增加。
- 进程管理复杂：由于每个素数的筛选都依赖于递归调用，需要管理多个子进程的创建和结束，增加了程序的复杂性和调试难度。

实验中遇到的问题与解决方法如下：

- 问题：父进程向管道中写入数据后，子进程可能会因为管道中的数据尚未完全写入而无法正确读取，导致素数筛选过程中的数据丢失或错误。
- 解决方法：父进程在将数字 2 到 35 写入管道后，关闭写端 p[1]，表示写入完成。子进程在读取数据时，正确处理读端口关闭的情况，通过检测读取返回值是否为0来判断管道是否已经关闭。

5. find

5.1. 实验目的

1. 编写一个简单版本的 UNIX 查找程序，查找目录树中带有特定名称的所有文件。
2. 理解文件系统中目录和文件的基本概念和组织结构。
3. 熟悉在 xv6 操作系统中使用系统调用和文件系统接口进行文件查找操作。
4. 应用递归算法实现在目录树中查找特定文件。

5.2. 实验步骤

- 编写程序
在 user/ 目录下创建一个名为 find.c 的文件，代码如下：

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

#include "kernel/fs.h"
#include "kernel/stat.h"
#include "user/user.h"

void find(char *path, const char *filename)
{
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;

    if ((fd = open(path, 0)) < 0) {
        fprintf(2, "find: cannot open %s\n", path);
        return;
    }

    if (fstat(fd, &st) < 0) {
        fprintf(2, "find: cannot fstat %s\n", path);
        close(fd);
        return;
    }

    //参数错误, find的第一个参数必须是目录
    if (st.type != T_DIR) {
        fprintf(2, "usage: find <DIRECTORY> <filename>\n");
        return;
    }

    if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {
        fprintf(2, "find: path too long\n");
        return;
    }

    strcpy(buf, path);
    p = buf + strlen(buf);
    *p++ = '/'; //p指向最后一个'/'之后
    while (read(fd, &de, sizeof de) == sizeof de) {
        if (de.inum == 0)
            continue;
        memmove(p, de.name, DIRSIZ); //添加路径名称
        p[DIRSIZ] = 0; //字符串结束标志
        if (stat(buf, &st) < 0) {
            fprintf(2, "find: cannot stat %s\n", buf);
            continue;
        }
        //不要在“.”和“..”目录中递归
        if (st.type == T_DIR && strcmp(p, ".") != 0 && strcmp(p, "..") != 0) {
            find(buf, filename);
        } else if (strcmp(filename, p) == 0)
            printf("%s\n", buf);
    }

    close(fd);
}

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(2, "usage: find <directory> <filename>\n");
        exit(1);
    }
    find(argv[1], argv[2]);
}
```

```
    exit(0);
}
```

- 编辑 Makefile

打开 Makefile，在 Makefile 中找到名为 UPROGS 的行，添加 find 程序的目标名称：\$U/_find\

- 编译并测试程序

```
$ find . b
./b
./a/b
$
```

- 单元测试

```
[1@vm:~/xv6-labs-2020]$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.3s)
== Test find, recursive == find, recursive: OK (1.0s)
```

5.3. 实验小结

本实验实现了一个简单的 find 命令，该命令能递归遍历目录树并查找目标文件。

通过本次实验，我学会了：

1. 目录读取：通过参考 user/ls.c，我了解了如何读取目录内容以及提取文件名。
2. 递归算法：递归方法让我们能够深入到子目录中进行查找，同时避免了无限递归的问题。
3. 文件系统接口的使用：在 xv6 操作系统中，我们使用系统调用和文件系统接口来实现文件和目录的操作。
4. 字符串处理：C 语言中的字符串处理需要特别注意，使用 strcmp 进行字符串比较，避免了直接使用==进行比较的错误。
5. 错误处理：在文件操作中，我们增加了错误检查，提高了程序的健壮性。

在实验中，我遇到了无限递归的问题。在遍历目录时，如果递归进入"."和".."目录，会导致无限递归，最终导致栈溢出。解决方案是在递归调用 find 函数之前，检查当前目录项是否为"."或".."，如果是则跳过。这可以通过在 find 函数中增加如下代码实现：

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

6. xargs

6.1. 实验目的

1. 编写一个简单版的 UNIX xargs 程序：从标准输入读取每一行，并将行作为参数传递给命令执行。
2. 熟悉命令行参数获取和处理：实验需要解析命令行参数并进行适当处理，包括选项解析和参数拆分。
3. 学习执行外部命令：实验中需要调用 exec 函数来执行外部命令，理解执行外部程序的基本原理。

6.2. 实验步骤

- 编写程序

在 user/ 目录下创建一个名为 xargs.c 的文件，代码如下：

```

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

int main(int argc, char *argv[]) {
    char inputBuf[32]; // 记录上一个命令的输入
    char charBuf[320]; // 存储所有标记字符的缓冲区
    char* charBufPointer = charBuf;
    int charBufSize = 0;

    char *commandToken[32]; // 使用空格(' ')分隔输入后记录的标记
    int tokenSize = argc - 1; // 记录标记数量(初始值为argc - 1, 因为xargs不会被执行)
    int inputSize = -1;

    // 首先将初始argv参数复制到commandToken
    for(int tokenIdx=0; tokenIdx<tokenSize; tokenIdx++)
        commandToken[tokenIdx] = argv[tokenIdx+1];

    while((inputSize = read(0, inputBuf, sizeof(inputBuf))) > 0) {
        for(int i = 0; i < inputSize; i++) {
            char curChar = inputBuf[i];
            if(curChar == '\n') { // 如果读取到'\n', 执行命令
                charBuf[charBufSize] = 0; // 在标记的末尾设置'\0'
                commandToken[tokenSize++] = charBufPointer;
                commandToken[tokenSize] = 0; // 在数组末尾设置空指针

                if(fork() == 0) { // 创建子进程执行命令
                    exec(argv[1], commandToken);
                }
                wait(0);
                tokenSize = argc - 1; // 初始化
                charBufSize = 0;
                charBufPointer = charBuf;
            }
            else if(curChar == ' ') {
                charBuf[charBufSize++] = 0; // 标记字符串的结尾
                commandToken[tokenSize++] = charBufPointer;
                charBufPointer = charBuf + charBufSize; // 切换到新字符串的起始位置
            }
            else {
                charBuf[charBufSize++] = curChar;
            }
        }
        exit(0);
    }
}

```

- 编辑 Makefile

打开 Makefile，在 Makefile 中找到名为 UPROGS 的行，添加 xargs 程序的目标名称：\$U/_xargs\

- 编译并测试程序

```

lq@vm:/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file
=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ sleep 10
$ QEMU: Terminated

```

- 单元测试

```

lq@vm:~/xv6-labs-2020$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (2.4s)

```

6.3. 实验小结

通过本次实验，我学会了：

- 命令行参数处理：学会了如何解析和处理命令行参数，理解了参数传递和选项解析的基本原理。

- 进程管理：了解了如何在 xv6 环境中创建和管理进程，通过 fork 创建子进程，并通过 exec 执行外部命令。
- 标准输入输出处理：学会了如何从标准输入读取数据，并将其作为命令参数传递执行。

Lab2 : system calls

切换分支

```
$ git fetch
$ git checkout syscall
$ make clean
```

1. System call tracing

1.1. 实验目的

本实验旨在添加一个系统调用追踪功能，以便在后续实验中进行调试。具体要求如下：

- 创建一个新的 trace 系统调用，用于控制追踪功能。
- trace 系统调用应接受一个整数参数 "mask"，该参数的每个位表示要追踪的特定系统调用。例如，为了追踪 fork 系统调用，程序可以调用 trace(1 << SYS_fork)，其中 SYS_fork 是来自 kernel/syscall.h 的系统调用号。
- 修改 xv6 内核，使其在每个系统调用即将返回时打印出一行信息（如果该系统调用的号码在 mask 中设置了）。打印的信息应包括进程 ID、系统调用名称和返回值，但不需要打印系统调用的参数。
- trace 系统调用应仅为调用它的进程及其后续 fork 的子进程启用追踪，而不影响其他进程。

1.2. 实验步骤

- 在 kernel/syscall.h 添加宏定义，模仿已经存在的系统调用序号的宏定义： `#define SYS_trace 22`

```
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_trace 22 // new-lab2
```

- 官方已提供了用户态的 trace 函数（user/trace.c），因此我们只需在 user/user.h 文件中声明用户态可以调用 trace 系统调用。接着，需要查看 trace.c 文件，明确该系统调用的参数和返回值的类型。代码 `trace(atoi(argv[1])) < 0` 显示 trace 函数传入的是一个数字，并且与 0 进行比较。结合实验提示，可以确定传入参数的类型为 int，并且推测返回值类型也是 int。

基于以上分析，我们可以在内核中声明 trace 系统调用： `int trace(int);`

```
char* sbrk(int);
int sleep(int);
int uptime(void);
int trace(int); // new-lab2

// ulib.c
int stat(const char*, struct stat*);
char* strcpy(char*, const char*);
void *remove(void*, const void*, int);
```

- 查看 user/usys.pl 文件，该文件中使用 Perl 语言自动生成用户态系统调用接口的汇编语言文件 usys.S。因此，我们需要在 user/usys.pl 文件中加入以下语句： `entry("trace");`

```
entry("sbrk");
entry("sleep");
entry("uptime");
entry("trace"); # new-lab2
-- INSERT --
```

- 执行 ecall 指令后会跳转至 kernel/syscall.c 文件中的 syscall 函数处，并执行该函数。

接下来，`p->trapframe->a0 = syscalls[num]();` 语句通过调用 `syscalls[num]()` 函数，并将返回值保存在 a0 寄存器中。`syscalls[num]()` 函数在当前文件中定义，调用具体的系统调用命令。把新增的 trace 系统调用添加到函数指针数组 `*syscalls[]` 上：

```
[SYS_mkdir] sys_mkdir,
[SYS_close] sys_close,
[SYS_trace] sys_trace, // new-lab2
};
```

- 在文件开头给内核态的系统调用 trace 加上声明，即在 kernel/syscall.c 加上：`extern uint64 sys_trace(void);`
- 根据提示，trace 系统调用应该有一个参数，一个整数“mask(掩码)”，其指定要跟踪的系统调用。所以，在 kernel/proc.h 文件的 proc 结构体中，新添加一个变量 mask，使得每一个进程都有自己的 mask，即要跟踪的系统调用。

```

// these are private to the process, so p->lock need not be held.
uint64 kstack;           // Virtual address of kernel stack
uint64 sz;               // Size of process memory (bytes)
pagetable_t pagetable;   // User page table
struct trapframe *trapframe; // data page for trampoline.S
struct context context;  // swtch() here to run process
struct file *file[NFILE]; // Open files
struct inode *cwd;        // Current directory
char name[16];           // Process name (debugging)
int tracemask;
};

```

- 在 kernel/sysproc.c 给出 sys_trace 函数的具体实现，把传进来的参数给到现有进程的 mask 即可。

```

uint64
sys_trace(void)
{
    int mask;
    if(argint(0,&mask) < 0)
        return -1;

    myproc()->tracemask = mask;
    return 0;
}

```

- 由于 RISCV 的 C 规范是将返回值放在 a0 寄存器中，所以在调用系统调用时，我们只需判断是否为 mask 规定的输出函数，如果是，就进行输出操作。首先，在 kernel/proc.h 文件中，proc 结构体中的 name 字段是线程的名字，不是函数调用的函数名称。因此，我们需要在 kernel/syscall.c 中定义一个数组来存储系统调用的名字。这里需要注意，系统调用的名字必须按顺序排列，第一个为空字符串。

```

static char *syscall_names[]={
    "", "fork", "exit", "wait", "pipe",
    "read", "kill", "exec", "fstat", "chdir",
    "dup", "getpid", "sbrk", "sleep", "uptime",
    "open", "write", "mknod", "unlink", "link",
    "mkdir", "close", "trace"};

```

- 在 kernel/syscall.c 文件的 syscall 函数中添加打印系统调用情况的语句。mask 是按位判断的，因此需要使用按位运算。进程序号可以通过 p->pid 获取，函数名称可以从我们刚刚定义的数组 syscall_names[num] 获取，返回值则是 p->trapframe->a0。以下是更新后的 syscall 函数代码：

```

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

```

- 在 kernel/proc.c 中 fork 函数调用时，添加子进程复制父进程的 mask 的代码： np->tracemask = p->tracemask;
- 编辑 Makefile
在 Makefile 的 UPROGS 中添加：\$U/_trace\
• 编译并测试程序

```

xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
 3: syscall read -> 1023
 3: syscall read -> 966
 3: syscall read -> 70
 3: syscall read -> 0
$ trace 2147483647 grep hello README
 4: syscall trace -> 0
 4: syscall exec -> 3
 4: syscall open -> 3
 4: syscall read -> 1023
 4: syscall read -> 966
 4: syscall read -> 70
 4: syscall read -> 0
 4: syscall close -> 0
$ grep hello README
$
$ trace 2 usertests forkforkfork
usertests starting
 7: syscall fork -> 8
test forkforkfork: 7: syscall fork -> 9

```

- 单元测试

```

lq@vm:/xv6-labs-2020$ ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (0.9s)
== Test trace all grep == trace all grep: OK (1.1s)
== Test trace nothing == trace nothing: OK (0.9s)
== Test trace children == trace children: OK (17.7s)
lq@vm:/xv6-labs-2020$ 

```

1.3. 实验小结

本次实验成功实现了系统调用追踪功能，通过添加新的 trace 系统调用，能够按需控制追踪特定的系统调用。在实现过程中，通过解决变量命名冲突、确保系统调用编号的正确定义、生成用户态系统调用接口、正确继承 tracemask 以及完善输出信息，最终达到了预期的实验目标。此功能在后续实验和开发中将大大提升调试效率和系统理解深度。

2. Sysinfo

2.1. 实验目的

在本实验中，我们将添加一个名为 sysinfo 的系统调用，用于收集系统运行信息。该系统调用需要一个参数：指向 struct sysinfo 的指针（参见 kernel/sysinfo.h）。内核需要填写该结构体的以下字段：freemem 字段应设置为可用内存的字节数，nproc 字段应设置为状态不是 UNUSED 的进程数。

2.2. 实验步骤

- 定义一个系统调用的序号。系统调用序号的宏定义在 kernel/syscall.h 文件中。在 kernel/syscall.h 添加宏定义 SYS_sysinfo 如下：`#define SYS_sysinfo 23`
- 在 user/usys.pl 文件加入下面的语句：`entry("sysinfo");`
- 在 user/user.h 中添加 sysinfo 结构体以及 sysinfo 函数的声明：

```

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

- 在 kernel/syscall.c 中新增 sys_sysinfo 函数的定义：`extern uint64 sys_sysinfo(void);`
- 在 kernel/syscall.c 中函数指针数组新增 sysinfo，并把新增的 sysinfo 系统调用添加到函数指针数组 *syscalls[] 上：

```

static char *syscall_names[] = {
  "", "fork", "exit", "wait", "pipe",
  "read", "kill", "exec", "fstat", "chdir",
  "dup", "getpid", "sbrk", "sleep", "uptime",
  "open", "write", "mknod", "unlink", "link",
  "mkdir", "close", "trace", "sysinfo"};
```

- 在进程中已经保存了当前进程的状态，因此我们可以直接遍历所有进程，判断它们的状态是否为 UNUSED 并进行计数。根据 proc 结构体的定义，访问进程状态时必须加锁。我们在 kernel/proc.c 中新增了一个名为 nproc 的函数，用于获取可用进程的数量，代码如下：

```

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

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

```

- 在 kernel/kalloc.c 中添加一个 free_mem 函数，用于收集可用内存的数量。参考 kernel/kalloc.c 文件中的 kalloc() 和 kfree() 函数可以看出，内核通过 kmem.freelist 链表维护未使用的内存。链表的每个节点对应一个页表大小 (PGSIZE)。分配内存时，从链表头部取走一个页表大小的内存；释放内存时，使用头插法将其插入到该链表。因此，计算未使用内存的字节数 freemem 只需遍历该链表，得到链表节点数，并与页表大小 (4KB) 相乘即可。

```

uint64
free_mem(void)
{
    struct run *r;
    uint64 num=0;
    acquire(&kmem.lock);
    r=kmem.freelist;
    while(r){
        num++;
        r=r->next;
    }
    release(&kmem.lock);
    return num * PGSIZE;
}

```

- 在 kernel/defs.h 中添加上述两个新增函数的声明：

```

// kalloc.c
int64 free_mem(void);
// proc.c
uint64 nproc(void);

```

- 在 sys_sysinfo 函数的实现中，首先使用 argaddr 函数读取用户态数据 sysinfo 的指针地址。然后，将内核中获取的 sysinfo 数据，按 sizeof(info) 大小复制到该指针指向的内存位置。

以下是 kernel/sysproc.c 文件中添加的 sys_sysinfo 函数的具体实现：

```

// add header
#include "sysinfo.h"

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

    if(argaddr(0,&addr)<0)
        return -1;

    info.freemem=free_mem();
    info.nproc=nproc();
    if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
        return -1;
    return 0;
}

```

```

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

    if(argaddr(0,&addr)<0)
        return -1;

    info.freemem=free_mem();
    info.nproc=nproc();
    if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
        return -1;
    return 0;
}

```

- 最后在 user 目录下添加一个 sysinfo.c 用户程序

```

#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/sysinfo.h"
#include "user/user.h"
int main(int argc, char *argv[])
{
    if (argc != 1){
        fprintf(2, "Usage: %s need not param\n", argv[0]);
        exit(1);
    }
    struct sysinfo info;
    sysinfo(&info);
    printf("free space: %d\nused process: %d\n", info.freemem, info.nproc);
    exit(0);
}

```

- 编辑 Makefile

在 Makefile 的 UPROGS 中添加: \$U/_sysinfotest

- 编译并测试程序

```

xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ sysinfo
free space: 133386240
used process: 3
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$ 

```

- 单元测试

```

lq@vm:~/xv6-labs-2020$ ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (3.8s)

```

2.3. 实验小结

通过本次实验，我成功实现了 sysinfo 系统调用，深入理解了系统调用的实现过程、锁机制在内核编程中的重要性，以及如何调试和解决实际编程中的问题。

推送至仓库

```
lq@vm:~/xv6-labs-2020$ git add .
lq@vm:~/xv6-labs-2020$ git commit -m "2025-08-03:lab2"
[syscall ceff35d] 2025-08-03:lab2
 11 files changed, 106 insertions(+), 1 deletion(-)
 create mode 100644 user/sysinfo.c
lq@vm:~/xv6-labs-2020$ git push github syscall:syscall
Enumerating objects: 108, done.
Counting objects: 100% (77/77), done.
Delta compression using up to 4 threads
Compressing objects: 100% (35/35), done.
Writing objects: 100% (52/52), 7.18 KiB | 3.59 MiB/s, done.
Total 52 (delta 34), reused 31 (delta 15), pack-reused 0
remote: Resolving deltas: 100% (34/34), completed with 14 local objects.
remote:
remote: Create a pull request for 'syscall' on GitHub by visiting:
remote:   https://github.com/lq1911/OS/pull/new/syscall
remote:
To https://github.com/lq1911/OS.git
 * [new branch]      syscall -> syscall
```

Lab3 : page tables

切换分支

```
$ git fetch
$ git checkout pgtbl
$ make clean
```

1. Print a page table

1.1. 实验目的

为了帮助可视化 RISC-V 页表，并且可能有助于未来的调试，该项目将需要编写一个函数来打印页表的内容。定义一个名为 vmprint() 的函数。它应该接收一个 pagetable_t 参数，并按照下面描述的格式打印该页表。在 exec.c 中，在 return argc 之前插入 if(p->pid==1) vmprint(p->pagetable) 语句，以打印第一个进程的页表，不输出无效的 PTE。

1.2. 实验步骤

- 在 exec.c 中的 return argc 之前插入代码： `if(p->pid==1) vmprint(p->pagetable)`
- 查看 kernel/vm.c 里面的 freewalk 方法，根据 freewalk 写下递归函数。对于每一个有效的页表项都打印其和其子项的内容。如果不是最后一层的页表就继续递归。通过 level 来控制前缀“..”的数量。

```

/**
 * @param pagetable 所要打印的页表
 * @param level 页表的层级
 */
void
_vmpprint(pagetable_t pagetable, int level){
    // there are 2^9 = 512 PTEs in a page table.
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        // PTE_V is a flag for whether the page table is valid
        if(pte & PTE_V){
            for (int j = 0; j < level; j++){
                if (j) printf(" ");
                printf("..");
            }
            uint64 child = PTE2PA(pte);
            printf("%d: pte %p pa %p\n", i, pte, child);
            if((pte & (PTE_R|PTE_W|PTE_X)) == 0){
                // this PTE points to a lower-level page table.
                _vmpprint((pagetable_t)child, level + 1);
            }
        }
    }
}

/**
 * @brief vmpprint 打印页表
 * @param pagetable 所要打印的页表
 */
void
vmpprint(pagetable_t pagetable){
    printf("page table %p\n", pagetable);
    _vmpprint(pagetable, 1);
}

```

- 将函数声明加入 kernel/defs.h

```

int copyin(pagetable_t, char *, uint64, uint64);
int copyinstr(pagetable_t, char *, uint64, uint64);
void vmpprint(pagetable_t);

```

- 编译并测试程序

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6e000
..0: pte 0x0000000021fd801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fd801 pa 0x0000000087f69000
... ... ..0: pte 0x0000000021fd801 pa 0x0000000087f6b000
... ... ..1: pte 0x0000000021fd800f pa 0x0000000087f68000
... ... ..2: pte 0x0000000021fd801f pa 0x0000000087f67000
..255: pte 0x0000000021fdb001 pa 0x0000000087f6d000
... ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
... ...510: pte 0x0000000021fdd807 pa 0x0000000087f76000
... ...511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh

```

- 单元测试

```

lq@vn: /xv6-labs-2020$ ./grade-lab-pttbl pte
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (1.6s)

```

1.3. 实验小结

- 在本实验中，我们编写了一个函数 vmpprint()，以便打印 RISC-V 页表的内容。通过在 exec.c 文件中的适当位置调用该函数，可以有效地输出第一个用户进程（控制台进程）的页表信息，从而帮助我们可视化页表的结构和内容。
 - 函数实现：在 vmpprint() 函数中，我们通过三重循环遍历三级页目录，每一层页目录都使用 PTE_V 位判断页表条目是否有效。当发现有效条目时，我们会递归进入下一层页目录，直到到达最低级页目录。对于最低级页目录中的每个有效条目，我们会输出对应的物理地址。

- 代码插入位置：我们选择在 exec 函数的 return argc 之前插入 if(p->pid==1) vmprint(p->pagetable)，目的是打印第一个用户进程（即控制台进程）的页表。通过这一修改，可以确保我们在 xv6 启动时获得页表的输出信息。
 - 结果验证：通过编译并运行修改后的 xv6 系统，我们成功输出了控制台进程的页表信息，从而验证了 vmprint() 函数的正确性和有效性。
2. 输出结果的可读性：通过输出的页表信息，可以直观地看到页表条目的层次结构和对应的物理地址。这对于理解和调试分页机制有很大帮助。然而，输出的格式可能需要进一步美化，以便在更复杂的场景下提供更好的可读性。
3. 递归与循环的选择：在本实验中，我们选择了循环的方法来实现页表遍历。尽管递归方法在逻辑上更简洁，但循环方法在控制复杂性和性能上可能具有一定优势。在实际应用中，两种方法各有优劣，可以根据具体需求进行选择。
4. 对无效 PTE 的处理：为了避免输出无效的页表条目，我们在遍历过程中添加了 if (top_pte & PTE_V) 的检查。这确保了输出结果仅包含有效的页表条目，从而提高了输出信息的有效性和准确性。

2. A kernel page table per process

2.1. 实验目的

本实验的目的是让每个进程都有自己的内核页表，这样在内核中执行时使用它自己的内核页表的副本。

2.2. 实验步骤

- 编写代码

在 kernel/proc.h 文件中给 struct proc 结构体添加内核页表数据成员

```
struct proc {
    // 添加内核页表 lab3-2
    pagetable_t kpagetable;
};
```

- 在 kernel/vm.c 文件中实现对每个进程的内核页表的初始化函数，参考 kvminit() 和 mappages() 函数的功能同时记得添加头文件，并且要注意头文件顺序

```
// lab3-2
#include "spinlock.h"
#include "proc.h"
...
// 初始化kernel页表 lab3-2
pagetable_t _kvminit() {
    pagetable_t pgtbl = uvmcreate();
    _kvmmmap(pgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);
    _kvmmmap(pgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
    _kvmmmap(pgtbl, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
    _kvmmmap(pgtbl, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
    _kvmmmap(pgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
    _kvmmmap(pgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);
    _kvmmmap(pgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
    return pgtbl;
}

void _kvmmmap(pagetable_t pagetable, uint64 va, uint64 pa, uint64 sz, int perm) {
    if(mappages(pagetable, va, sz, pa, perm) != 0)
        panic("_kvmmmap");
}
```

- 修改 kernel/vm.c 原来的 kvminit() 函数，调用 _kvminit() 完成对全局内核页表的初始化

```

void
_kvminit()
{
    // kernel_pagetable = (pagetable_t) kalloc();
    // memset(kernel_pagetable, 0, PGSIZE);

    // // uart registers
    // kvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W);

    // // virtio mmio disk interface
    // kvmmap(VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

    // // CLINT
    // kvmmap(CLINT, CLINT, 0x10000, PTE_R | PTE_W);

    // // PLIC
    // kvmmap(PLIC, PLIC, 0x400000, PTE_R | PTE_W);

    // // map kernel text executable and read-only.
    // kvmmap(KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

    // // map kernel data and the physical RAM we'll make use of.
    // kvmmap((uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);

    // // map the trampoline for trap entry/exit to
    // // the highest virtual address in the kernel.
    // kvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

    kernel_pagetable = _kvminit();
}

```

- 在 kernel/defs.h 中添加相应函数声明

```

// lab3-1
void vmprint(pagetable_t);
// 初始化kernel页表 lab3-2
pagetable_t _kvminit();
// 映射
void _kvmmap(pagetable_t, uint64, uint64, uint64, int);
// vm.c的walk函数
pte_t* walk(pagetable_t, uint64, int);

```

- 在 kernel/proc.c 文件中修改 procinit() 函数和 allocproc() 函数，将其中对于内核栈 kstack 的初始化移动至 allocproc() 函数中

```

void
procinit(void)
{
    struct proc *p;

    initlock(&pid_lock, "nextpid");
    for(p = proc; p < &proc[NPROC]; p++) {
        initlock(&p->lock, "proc");

        // Allocate a page for the process's kernel stack.
        // Map it high in memory, followed by an invalid
        // guard page.
        // 将该处处理移动到allocproc() lab3-2
        // char *pa = kalloc();
        // if(pa == 0)
        //     panic("kalloc");
        // uint64 va = KSTACK((int)(p - proc));
        // kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
        // p->kstack = va;
    }
    kvminithart();
}

...

static struct proc*
allocproc(void)
{
    ...
found:
    ...

    // An empty user page table.
    p->pagetable = proc_pagetable(p);
    if(p->pagetable == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }

    // 增加内核页表 lab3-2
    p->kpagetable = _kvminit();
    if (p->kpagetable == 0) {
        freeproc(p);
        release(&p->lock);
        return 0;
    }

    // 在此处初始化内核页表 lab3-2
    char* pa = kalloc();
    if (pa == 0)
        panic("kalloc");
    uint64 va = KSTACK((int)(p - proc));
    _kvmmap(p->kpagetable, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
    p->kstack = va;

    // Set up new context to start executing at forkret,
    // which returns to user space.
    memset(&p->context, 0, sizeof(p->context));
    p->context.ra = (uint64)forkret;
    p->context.sp = p->kstack + PGSIZE;
}

```

```
    return p;
}
```

- 在 kernel/proc.c 文件中修改 scheduler()，切换进程的同时也要切换进程各自的内核页表，同时需要刷新快表

```
void
scheduler(void)
{
    ...
    for(p = proc; p < &proc[NPROC]; p++) {
        ...
        p->state = RUNNING;
        c->proc = p;

        // 同时也要切换每个进程的内核页表 lab3-2
        w_satp(MAKE_SATP(p->kpagetable));
        // 刷新快表 lab3-2
        sfence_vma();

        swtch(&c->context, &p->context);

        // 切换回全局的内核页表 lab3-2
        kvm_inithart();

        // Process is done running for now.
        ...
    }
    release(&p->lock);
    ...
}
```

- 在 kernel/proc.c 文件中修改 freeproc() 函数，释放相应的内核栈和内核页表参考第一个实验遍历页表的方式释放内核页表实现 proc_freetkpagetable() 函数

```

// 释放内核页表辅助递归函数 lab3-2
void proc_freekpagetable(pagetable_t kpagetable) {
    for (int i = 0; i < 512; ++i) {
        pte_t pte = kpagetable[i];
        if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
            uint64 child = PTE2PA(pte);
            proc_freekpagetable((pagetable_t)child);
            kpagetable[i] = 0;
        }
    }
    kfree((void*)kpagetable);
}

// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
    if(p->trapframe)
        kfree((void*)p->trapframe);
    p->trapframe = 0;

    // 释放内核栈 lab3-2
    if (p->kstack) {
        pte_t* pte = walk(p->kpagetable, p->kstack, 0);
        if (pte == 0)
            panic("freeproc: kstack");
        kfree((void*)PTE2PA(*pte));
    }
    p->kstack = 0;

    if(p->pagetable)
        proc_freepagetable(p->pagetable, p->sz);
    p->pagetable = 0;

    // 释放内核页表 lab3-2
    if (p->kpagetable)
        proc_freekpagetable(p->kpagetable);
    p->kpagetable = 0;

    p->sz = 0;
    ...
}

```

- 在 kernel/vm.c 文件中修改 kvmpa(), 将全局内核页表转换成当前进程对应的内核页表

```

uint64
kvmva(uint64 va)
{
    ...
    uint64 pa;
    // 使用进程自己的内核页表
    pte = walk(myproc()->kpagetable, va, 0);
    ...
}

```

- 编译并测试程序

```

xv6 kernel is booting
hart 1 starting
hart 2 starting
page table 0x000000000087f64000
... 0: pte 0x0000000021fd8001 pa 0x000000000087f60000
... ..0: pte 0x0000000021fd7c01 pa 0x000000000087f5f000
... ...0: pte 0x0000000021fd841f pa 0x000000000087f61000
... ... .1: pte 0x0000000021fd780f pa 0x000000000087f5e000
... ... ..2: pte 0x0000000021fd741f pa 0x000000000087f5d000
... 255: pte 0x000000000021fd8c01 pa 0x000000000087f63000
... ..511: pte 0x0000000021fd8801 pa 0x000000000087f62000
... ...510: pte 0x0000000021fed807 pa 0x000000000087fb6000
... ... ..511: pte 0x0000000020001c0b pa 0x00000000008007000
init: starting sh
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x0000000000000000c pid=3234
    sepc=0x00000000000053fe stval=0x00000000000053fe
usertrap(): unexpected scause 0x000000000000000d pid=6253
    sepc=0x000000000000201a stval=0x00000000001dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6265
    sepc=0x0000000000003e76 stval=0x00000000000012000
OK
test sbrkarg: OK
test validatestest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6269
    sepc=0x0000000000002188 stval=0x000000000000fbcd
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitinput: OK
test input: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdo: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$ 

```

2.3. 实验小结

本实验通过为每个进程分配独立的内核页表，优化了 Xv6 内核的内存管理机制。主要工作包括：

1. 内核页表初始化

在 proc 结构中新增 kernelpt 字段，用于存储进程的内核页表。

实现 proc_kpt_init 函数，复制全局内核页表的固定映射（如设备地址、内核代码等），并调用 uvmmap 为进程的内核栈建立专属映射。

2. 调度与页表切换

修改 scheduler，在进程切换时通过 proc_inithart 将进程的内核页表加载到 SATP 寄存器，并在切换回内核时恢复全局内核页表。

更新 kvmppa 函数，使其使用当前进程的内核页表转换虚拟地址。

3. 资源释放

在 freeproc 中释放进程的内核栈和内核页表，递归清理页表项 (proc_freetokenpt)，避免内存泄漏。

3. Simplify

3.1. 实验目的

本实验是实现将用户空间的映射添加到每个进程的内核页表，将进程的页表复制一份到进程的内核页表。

3.2. 实验步骤

- 在 kernel/vm.c 文件中实现 uvm2kvm() 函数，将进程中用户页表复制到内核页表

```

void uvm2kvm(pagetable_t upagetable, pagetable_t kpagetable, uint64 src, uint64 dst) {
    if (src > PLIC)
        panic("uvm2kvm: src larger than PLIC");
    src = PGROUNDDOWN(src);
    for (uint64 i = src; i < dst; i += PGSIZE) {
        pte_t* pte_k = walk(kpagetable, i, 1);
        if (pte_k == 0)
            panic("uvm2kvm: kernel pagetable fails");
        *pte_k = *pte_u;
        *pte_k &= ~PTE_U;
    }
}

```

- 在 kernel/vm.c 文件中修改 copyin() 和 copyinstr()

```

int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
    // uint64 n, va0, pa0;
    ...
    // return 0;
    return copyin_new(pagetable, dst, srcva, len);
}

int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
    // uint64 n, va0, pa0;
    ...
    // }
    return copyinstr_new(pagetable, dst, srcva, max);
}

```

- 修改系统调用 fork(), 使页表正确映射

```

int
fork(void)
{
    ...
    np->sz = p->sz;

    // lab3-3
    uvm2kvm(np->pagetable, np->kpagetable, 0, np->sz);

    np->parent = p;

    ...
}

```

- 在 exec.c 中调用这个函数

```

// 在返回argc之前调用vmprint lab3-1
if (p->pid == 1)
    vmprint(p->pagetable);

// lab3-3
uvm2kvm(p->pagetable, p->kpagetable, 0, p->sz);

return argc; // this ends up in a0, the first argument to main(argc, argv)

```

- 在 kernel/proc.c 文件中修改 userinit()、growproc()

```

void
userinit(void)
{
    ...
    p->sz = PGSIZE;

    // lab3-3
    uvm2kvm(p->pagetable, p->kpagetable, 0, p->sz);

    // prepare for the very first "return" from kernel to user.
    ...
}

int
growproc(int n)
{
    ...
    if(n > 0){
        if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
            return -1;
        }
    } else if(n < 0){
        sz = uvmdealloc(p->pagetable, sz, sz + n);
    }

    // lab3-3
    uvm2kvm(p->pagetable, p->kpagetable, sz - n, sz);

    p->sz = sz;

    ...
}

```

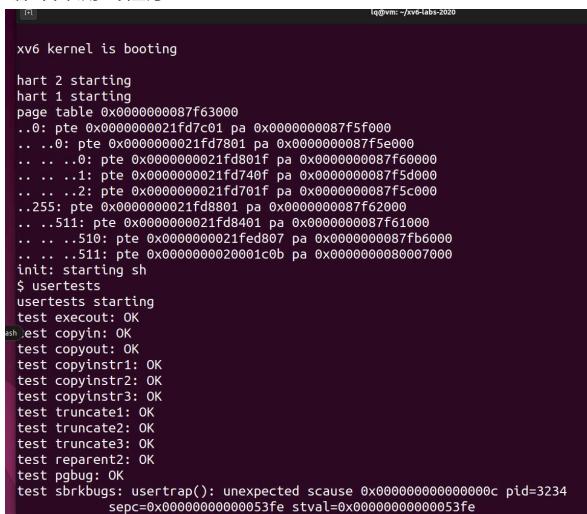
- 在 kernel/defs.h 文件中添加相关函数声明

```

// lab3-3
int copyin_new(pagetable_t, char*, uint64, uint64);
int copyinstr_new(pagetable_t, char*, uint64, uint64);
void uvm2kvm(pagetable_t, pagetable_t, uint64, uint64);

```

- 编译并测试程序



The screenshot shows a terminal window titled 'xv6 kernel is booting'. It displays the boot process of the xv6 kernel, including Hart initialization and page table setup. Following the boot, it shows the execution of user tests. The test results are as follows:

```

$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3234
              sepc=0x0000000000053fe stval=0x000000000000053fe

```

```

Terminal 8月 4 17:45
lq@vm: ~/xv6-labs-2020
usertrap(): unexpected scause 0x000000000000000d pid=6253
    sepc=0x000000000000201a stval=0x00000000001dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6265
    sepc=0x0000000000003e76 stval=0x000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6269
    sepc=0x0000000000002188 stval=0x00000000000fbcb0
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitput: OK
test iput: OK
test mem: OK
test pipe: 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
5

```

- 单元测试

```

lq@vm: ~/xv6-labs-2020 $ ./grade-lab-pgtbl
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (1.3s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin == count copyin: OK (1.1s)
== Test usertests == (230.1s)
== Test usertests: copyin ==
    usertests: copyin: OK
== Test usertests: copyinstr1 ==
    usertests: copyinstr1: OK
== Test usertests: copyinstr2 ==
    usertests: copyinstr2: OK
== Test usertests: copyinstr3 ==
    usertests: copyinstr3: OK
== Test usertests: sbrkmuch ==
    usertests: sbrkmuch: OK
== Test usertests: all tests ==
    usertests: all tests: OK
== Test time ==
time: OK
Score: 66/66

```

3.3. 实验小结

在 kernel/vm.c 文件中实现 uvm2kvm() 函数，将进程中用户页表复制到内核页表。这个函数只是复制了 p->pagetable 的物理地址，并没有申请新的空间，因此，复制的结果是 p->pagetable 和 p->kpagetable 共享同一个物理地址。同时，对于去掉标志位 PTE_U、PTE_W、PTE_X 是因为内核需要对该页表进行读，但 PTE_U 规定只能由用户访问，因此需要去掉；因为没必要修改，所以也去掉 PTE_W、PTE_X。

推送至仓库

```

lq@vm: ~/xv6-labs-2020 $ git add .
lq@vm: ~/xv6-labs-2020 $ git commit -m "2025-08-04:lab3"
[pgtbl f4bf522] 2025-08-04:lab3
8 files changed, 250 insertions(+), 71 deletions(-)
create mode 100644 answers-pgtbl.txt
create mode 100644 time.txt
lq@vm: ~/xv6-labs-2020 $ git push github pgtbl:pgtbl
Enumerating objects: 59, done.
Counting objects: 100% (59/59), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (35/35), done.
Writing objects: 100% (45/45), 17.08 KiB | 4.27 MiB/s, done.
Total 45 (delta 27), reused 20 (delta 6), pack-reused 0
remote: Resolving deltas: 100% (27/27), completed with 14 local objects.
remote:
remote: Create a pull request for 'pgtbl' on GitHub by visiting:
remote:     https://github.com/lq1911/OS/pull/new/pgtbl
remote:
To https://github.com/lq1911/OS.git
 * [new branch]      pgtbl -> pgtbl

```

Lab4 : traps

切换分支

```

$ git fetch
$ git checkout traps
$ make clean

```

1. RISC-V assembly

1.1. 实验目的

了解一些 RISC-V 汇编很重要。在 xv6 repo 中有一个文件 user/call.c。make fs.img 会对其进行编译。并生成 user/call.asm 中程序的可读汇编版本。

1.2. 实验步骤

- 在 Xv6 的命令行中输入运行 make fs.img，编译 user/call.c 程序，得到可读性比较强的 user/call.asm 文件。

```
balloc: first 643 blocks have been allocated
balloc: write bitmap block at sector 45
lq@vm:~/xv6-labs-2020$ cat user/call.asm

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
    2: e422            sd    s0,8(sp)
    4: 0800            addi   s0,sp,16
    sh return x+3;
}
    6: 250d            addiw a0,a0,3
    8: 6422            ld    s0,8(sp)
    a: 0141            addi   sp,sp,16
    c: 8082            ret

000000000000000e <f>:
int f(int x) {
    e: 1141            addi   sp,sp,-16
    10: e422            sd    s0,8(sp)
```

- Q: 哪些寄存器包含函数的参数？例如，哪个寄存器在 main 对 printf 的调用中保留13？

A: a0 a1 a2 ... a7等寄存器存储函数参数。通过实验目的中的提示进行操作，查看 user/call.asm 文件，可以看出a2寄存器保留了13。

```
void main(void) {
    ...
    22: 0800            addi   s0,sp,16
    printf("%d\n", f(8)+1, 13);
    24: 4635            li    a2,13
    ...
}
```

- Q: 在main的汇编代码中对函数f的调用在哪里？对函数g的调用在哪里？（提示：编译器可以内联函数）。

A: 并没有对 f 和 g 函数的调用，g 被内联 f 函数，f 函数被内联到 main 函数。

- Q: 函数 printf 位于哪个地址？

A: 查看 user/call.asm 文件，看到 printf 的入口为 0x630。

```
34: 600080e7        jalr   1536(ra) # 630 <printf>
```

- Q: main 中，jalr 跳转到 printf 之后，ra 的值是多少？

A: ra = pc + 4 = 0x34 + 4 = 0x38 即返回地址。

- Q: 运行下面的代码,回答问题

```
unsigned int i = 0x00646c72;
printf("H%x Wo%s", 57616, &i);
```

- Q1: 输出是什么？

A1: 输出是 He110 World 字符串

- Q2: 如果 RISC-V 是大端序的，要实现同样的效果，需要将 i 设置为什么？需要将 57616 修改为别的值吗？

A2: 需要将 i 设置为 0x726c6400；57616 不修改。%x 表示以十六进制数形式输出，57616 的 16 进制表示就是 e110，与大小端序无关，%s 是输出字符串，以整数i所在的开始地址读取，直到读取到 '0' 为止。当是小端序表示的时候，内存中存放的数是：72 6c 64 00，对应 r1d。

- Q3: 在下面的代码中，“y=”之后将打印什么？（注意：答案不是一个特定的值）为什么会发生这种情况？

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

A3：输出为：x=3 y=1。y 输出的值取决于执行这条语句前寄存器 a2 的值。

1.3. 实验小结

本实验中，ra 寄存器出现频次较高，对于该寄存器，我进一步了解到：在 RISC-V 架构中，ra 寄存器是一个特殊的寄存器，全称为返回地址寄存器（Return Address Register）。它用于保存函数调用的返回地址，即函数执行完毕后继续执行的下一条指令的地址。当程序执行函数调用时，当前指令的地址会被保存到 ra 寄存器中。然后，函数执行完成后，通过从 ra 寄存器中恢复保存的返回地址，程序可以顺利返回到函数调用的位置，继续执行后续的指令。在汇编代码中，通常使用jal和jalr指令进行函数调用和返回。在函数调用时，jal 或 jalr 指令将返回地址保存到 ra 寄存器中。而在函数返回时，使用 jr、ret 或 jalr 指令将保存在ra寄存器中的返回地址加载到程序计数器（PC）中，以实现跳转到函数调用的下一条指令。

2. Backtrace

2.1. 实验目的

实现一个回溯（backtrace）功能，用于在操作系统内核发生错误时，输出调用堆栈上的函数调用列表。这有助于调试和定位错误发生的位置。

2.2. 实验步骤

- 在 kernel/defs.h 声明 backtrace()，使得 sys_sleep 可以调用这个函数：void backtrace()。在 kernel/riscv.h 中添加内联函数 r_fp()，从而能够读取帧指针的值。GCC 编译器存储当前执行函数的帧指针在寄存器 S0，因此我们从 S0 中读取：

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

- 在 kernel/printf.c 中编写 backtrace() 的实现，使其输出所有的栈帧：

```
void backtrace() {
    // 当前栈帧
    uint64 fp = r_fp();
    // 用户栈最高地址
    uint64 top = PGROUNDDUP(fp);
    // 用户栈最低地址
    uint64 bottom = PGROUNDDOWN(fp);
    // 输出当前栈中返回地址
    for (; fp >= bottom && fp < top; fp = *((uint64 *) (fp - 16))) {
        printf("%p\n", *((uint64 *) (fp - 8)));
    }
}
```

该函数通过调用上述的 r_fp() 函数读取寄存器 s0 中的当前函数栈帧 fp 参考RISC-V的栈结构, 我们可以知道 fp-8 存放返回地址, fp-16 存放原栈帧。从而可以通过原栈帧得到上一级栈结构, 直到获取到最初的栈结构。

- 接着在 kernel/sysproc.c 的 sys_sleep() 函数中调用 backtrace()

```
uint64
sys_sleep(void)
{
    ...
    release(&tickslock);
    // lab4-2
    backtrace();
    return 0;
}
```

- 然后在 kernel/printf.c 的函数 panic() 中调用 backtrace()

```

void
panic(char *s)
{
    pr.locking = 0;
    printf("panic: ");
    printf(s);
    printf("\\\\n");
    // lab4-2
    backtrace();
    panicked = 1; // freeze uart output from other CPUs
    for(;;)
        ;
}

```

- 编辑 Makefile

打开 Makefile，在 Makefile 中找到名为 UPROGS 的行，添加 primes 程序的目标名称：\$U/_primes\

- 编译并测试程序

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bttest
0x00000000080002d34
0x00000000080002b98
0x00000000080002882

```

- 单元测试

```

lq@vn:~/xv6-labs-2020$ ./grade-lab-traps backtrace
make: 'kernel/kernel' is up to date.
== Test backtrace test == backtrace test: OK (2.1s)
  (Old xv6.out.backtracetest failure log removed)

```

2.3. 实验小结

- 实验中的问题及解决方法

- 获取上一级栈帧的终止条件：在实现过程中，需要考虑如何正确识别和处理栈帧的终止条件。通过使用 PGROUNDDOWN() 和 PGROUPUP() 函数，计算栈帧所在页面的边界地址，确保循环在合理的边界内运行，避免越界访问。
- 栈帧指针的有效性检查：在遍历栈帧时，需要确保栈帧指针 fp 的有效性。通过检查 fp 是否在用户栈空间页面的范围内，确保访问的地址是合法的，避免出现异常访问和崩溃。
- 多级调用栈的处理：在输出栈帧信息时，需要正确处理多级调用栈。通过依次访问每一级栈帧并输出相应的返回地址，保证调用栈信息的完整性和准确性。

- 实验心得

在本次实验中，通过编写 backtrace() 函数并成功实现栈帧信息的输出，我深刻体会到了对底层栈结构的理解和对系统调用栈的掌握的重要性。特别是在解决获取上一级栈帧的终止条件和栈帧指针有效性检查的问题时，进一步加深了我对 RISC-V 架构和操作系统内部机制的认识。

3. Alarm

3.1. 实验目的

本次实验将向 xv6 内核添加一个新的功能，即周期性地为进程设置定时提醒。这个功能类似于用户级的中断/异常处理程序，能够让进程在消耗一定的 CPU 时间后执行指定的函数，然后恢复执行。通过实现这个功能，我们可以为计算密集型进程限制 CPU 时间，或者为需要周期性执行某些操作的进程提供支持。

3.2. 实验步骤

test0: invoke handler

- 在 user/user.h 文件中声明需要添加的系统调用 sigalarm 和 sigreturn

```

// lab4-3
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);

```

- 在 user/usys.pl 文件中添加上述两个系统调用的入口

```
# lab4-3
entry("sigalarm");
entry("sigreturn");
```

- 在 kernel/syscall.h 添加定义

```
// lab4-3
#define SYS_sigalarm 22
#define SYS_sigreturn 23
```

- 在 kernel/syscall.c 添加函数声明

```
// lab4-3
extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);
...
static uint64 (*syscalls[])(void) = {
...
// lab4-3
[SYS_sigalarm] sys_sigalarm,
[SYS_sigreturn] sys_sigreturn,
};
```

- 在 kernel/proc.h 文件中给 proc 结构体增加数据成员

```
struct proc {
...
// lab4-3
// 间隔
int interval;
// 调用函数地址
uint64 handler;
// 经过的时钟周期数
int passtick;
...
};
```

- 在 kernel/proc.c 文件的 allocproc() 函数中对新的数据成员初始化赋值

```
static struct proc*
allocproc(void)
{
...
found:
...
// lab4-3
p->interval = 0;
p->handler = 0;
p->passtick = 0;

return p;
}
```

- 回到 kernel/sysproc.c 并中添加 sys_sigalarm() 函数的实现，为 proc 中的相关数据成员赋值

```

uint64 sys_sigalarm(void) {
    int interval;
    uint64 handler;
    struct proc* p = myproc();
    // 错误检查
    if (argint(0, &interval) < 0 || argaddr(1, &handler) < 0 || interval < 0)
        return -1;
    // 赋值
    p->interval = interval;
    p->handler = handler;
    p->passticker = 0;
    return 0;
}

```

- 针对 test0，在 kernel/sysproc.c 中添加 sys_sigreturn() 函数的实现

```

// lab4-3-test0
uint64 sys_sigreturn(void) {
    return 0;
}

```

- 在 kernel/trap.c 文件的 usertrap() 函数中添加时钟中断添加相应的处理代码

```

void
usertrap(void)
{
    ...
    if(p->killed)
        exit(-1);
    // lab4-3
    // 时钟中断
    if (which_dev == 2) {
        // 经过了规定的时间间隔重置 执行handler
        if (p->interval != 0 && ++p->passticker == p->interval) {
            p->passticker = 0;
            p->trapframe->epc = p->handler;
        }
    }
    ...
    usertrapret();
}

```

- 在 Makefile 文件中添加相关指令，使得可以对 alarmtest.c 编译： \$U/_alarmtest\\编译并测试程序

```

$ make qemu
$ alarmtest
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ alarmtest
test0 start
.....alarm!
test0 passed

```

test1/test2: resume interrupted code

- 继续在 kernel/proc.h 文件中给 proc 结构体增加数据成员，增加 struct trapframe* 类型的 sigframe 字段，以及 int 类型的 sigflag 字段

```

struct proc {
    ...
    struct trapframe* sigframe;
    int sigflag;
    ...
};

```

- 在 kernel/trap.c 文件中的 usertrap() 函数中执行 handler 之前对 sigframe 进行赋值

```

void
usertrap(void)
{
    ...
    if(p->killed)
        exit(-1);
    // lab4-3
    // 时钟中断
    if (which_dev == 2) {
        // 经过了规定的时间间隔重置 执行handler
        if (p->interval != 0 && ++p->passtick == p->interval && p->sigflag == 0) {

            // 使用trapframe后的一部分内存，trapframe大小为288B
            // 因此只要在trapframe地址后288以上地址都可
            // 此处512只是为了取整数幂
            p->sigframe = p->trapframe + 512;
            memmove(p->sigframe, p->trapframe, sizeof(struct trapframe));

            p->passtick = 0;
            p->trapframe->epc = p->handler;
            p->sigflag = 1;
        }
    }
    ...
    usertrapret();
}

```

- 在 kernel/sysproc.c 文件中实现 sys_sigreturn() 函数，它将 frame 复制回来

```

// lab4-3
uint64 sys_sigreturn(void) {
    struct proc* p = myproc();
    // 判断地址是否正确
    if (p->sigframe != p->trapframe + 512)
        return -1;
    memmove(p->trapframe, p->sigframe, sizeof(struct trapframe));
    // 重置
    p->passtick = 0;
    p->sigframe = 0;
    p->sigflag = 0;
    return 0;
}

```

- 在 kernel/proc.c 文件中的 allocproc() 中，初始化 p->sigframe

```

static struct proc*
allocproc(void)
{
    ...
found:
    ...
    // lab4-3
    p->interval = 0;
    p->handler = 0;
    p->passtick = 0;
    p->sigframe = 0;
    p->sigflag = 0;

    return p;
}

```

- 编译并测试程序

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
..alarm!
.alarm!
.alarm!
.alarm!
.alarm!
.alarm!
.alarm!
.alarm!
test1 passed
test2 start
.....alarm!
test2 passed
```

```
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pbug: OK
test srbkbugs: usertrap(): unexpected scause 0x0000000000000000 pid=3236
           sepc=0x00000000000053fe stval=0x00000000000053fe
usertrap(): unexpected scause 0x000000000000000c pid=3237
           sepc=0x00000000000053fe stval=0x00000000000053fe
OK
test badarg: OK
test reparent: OK
test twochildren: OK
```

```
test exitinput: OK
test input: OK
test mem: OK
test pipe1: 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
$
```

- 单元测试

```
lq@vn: /xv6-labs-2020$ ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (1.3s)
== Test running alarmtest == (4.6s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests == usertests: OK (283.9s)
(Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 85/85
```

3.3. 实验小结

这次实验使我深入理解了操作系统的信号处理机制，通过实现定时提醒功能，我学会了如何在内核中添加系统调用、管理进程状态和处理中断，提升了系统编程和调试能力。

此外，这次实验也涉及到用户态和管理态的转换，我再次巩固了如何设置声明和入口使得二者连接。实验中遇到的挑战，如正确保存和恢复 trapframe 以及防止函数重入，使我认识到细致的状态管理和全面的测试对于系统开发的重要性。

通过测试程序，我明白在修改内核操作时，应确保不影响系统稳定性，即在实现定时中断处理功能时，要确保不会影响系统的正常运行，确保中断处理程序能够及时

返回，避免影响其他中断和系统调度。进行充分的测试，我们才能确保定时中断处理不会导致系统崩溃或异常。

这次实践不仅增强了我的理论知识，也提高了独立解决问题的能力。

推送至仓库

```
lq@vn: /xv6-labs-2020$ git push github traps:traps
Counting objects: 100% (55/55), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (29/29), done.
Writing objects: 100% (39/39), 14.16 KiB | 7.08 MiB/s, done.
Total 39 (delta 24), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (24/24), completed with 15 local objects.
remote:
remote: Create a pull request for 'traps' on GitHub by visiting:
remote:   https://github.com/lq1911/OS/pull/new/traps
remote:
To https://github.com/lq1911/OS.git
 * [new branch]      traps -> traps
```

Lab5 : lazy page allocation

切换分支

```
$ git fetch  
$ git checkout lazy  
$ make clean
```

1. Eliminate allocation from sbrk

1.1. 实验目的

这个实验的目的是在 xv6 操作系统中实现用户堆内存的延迟分配功能。具体来说，就是修改 sbrk() 系统调用，让它不立即分配物理内存，而是只增加进程的虚拟地址空间大小；当进程首次访问这些地址时，通过页错误触发内核分配物理页并映射，从而减少不必要的内存分配，提高效率。

1.2. 实验步骤

- 在 kernel/sysproc.c 文件中修改 sys_sbrk() 函数，删去对 growproc() 函数调用，并为 myproc()->sz 增加 n。

```
uint64  
sys_sbrk(void)  
{  
    int addr;  
    int n;  
  
    if(argint(0, &n) < 0)  
        return -1;  
  
    // lab5-1  
    struct proc* p = myproc();  
    addr = p->sz;  
    p->sz += n;  
    // 错误处理  
    if (n < 0) {  
        p->sz = uvmdealloc(p->pagetable, addr, addr + n);  
    }  
    // if(growproc(n) < 0)  
    //     return -1;  
    return addr;  
}
```

- 编译并测试程序

启动 xv6 系统 QEMU 模拟器，键入指令 echo hi 进行测试：

```
xv6 kernel is booting  
hart 2 starting  
hart 1 starting  
init: starting sh  
$ echo hi  
usertrap(): unexpected scause 0x000000000000000f pid=3  
    sepc=0x00000000000012a6 stval=0x0000000000004008  
, panic: uvmunmap: not mapped
```

1.3. 实验小结

实验较简单。这里出现 usertrap() 是因为还没有进行实际的内存分配，知识虚晃一枪给 p->sz 增加了而已，growproc(n) 也没有调用。

2. Lazy allocation

2.1. 实验目的

修改trap.c中的代码以响应来自用户空间的页面错误，方法是在错误地址映射新分配的物理内存页面，然后返回到用户空间，让进程继续执行。您应该在生成“usertrap ()：...”消息的 printf 调用之前添加代码。为了让 echo hi 正常工作，您需要修改任何其他xv6内核代码。

2.2. 实验步骤

- 处理page fault：根据提示，当 r_scause() == 13 or 15 即表示一个 page fault。同时，r_stval() 返回 在 kernel/trap.c 文件中的 usertrap() 函数进行 page fault 的处理。

```
void
usertrap(void)
{
    ...
    if(r_scause() == 8){
        ...
    }
    // lab5-2 page fault的处理
    else if (r_scause() == 13 || r_scause() == 15) {
        char* pa;
        // 分配物理页面
        if ((pa = kalloc()) != 0) {
            // 用0填充
            memset(pa, 0, PGSIZE);
            // 引发page fault的虚拟地址向下取整
            uint64 va = PGROUNDDOWN(r_stval());
            // 页表映射
            if (mappages(p->pagetable, va, PGSIZE, (uint64)pa, PTE_W | PTE_R | PTE_U) != 0) {
                // 映射失败
                kfree(pa);
                printf("usertrap(): mappages() failed\n");
                // 杀死进程
                p->killed = 1;
            }
        }
        // 分配物理页面失败
        else {
            printf("usertrap(): kalloc() failed\n");
            // 杀死进程
            p->killed = 1;
        }
    }
    ...
}
```

- 处理 uvmunmap 错误：uvmunmap() 函数的功能是释放内存映射，但是页表中有些地址并没有分配实际的内存，没有进行映射，对于这些地址，直接跳过即可

修改 kernel/vm.c 文件中的的 uvmunmap() 函数，注释掉两行对 panic 的调用，并为所在分支添加 continue：

```

void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
    ...
    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
        if((pte = walk(pagetable, a, 0)) == 0)
            // lab5-2
            // panic("uvmunmap: walk");
        continue;
        if(*pte & PTE_V == 0)
            // lab5-2
            // panic("uvmunmap: not mapped");
        continue;
        if(PTE_FLAGS(*pte) == PTE_V)
            panic("uvmunmap: not a leaf");
        if(do_free){
            uint64 pa = PTE2PA(*pte);
            kfree((void*)pa);
        }
        *pte = 0;
    }
}

```

- 编译并测试程序

启动 xv6 系统 QEMU 模拟器，键入指令 `echo hi` 进行测试：

```

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

```

2.3. 实验小结

这里用到了 RISC-V 中的 stval 寄存器，该寄存器包含导致页面错误的虚拟地址，专门用来存放于 trap 有关的信息，帮助 OS 或其他软件更快确定和完成 trap 的处理。其中的存放非零值有两种情况：

- 内存访问非法：这种情况包括硬件断点（Hardware Breakpoints）、地址未对齐（address misaligned）、访问故障（access-fault，可能是没有权限等）、缺页故障（page-fault）等情况。当这种情况发生时，stval 会存储出错位置的虚拟地址。比如缺页故障发生时，stval 就会记录到底是对哪个虚拟地址的访问导致了本次缺页故障，内核就可以根据此信息去加载页面进入内存。
- 指令非法访问（illegal instruction）：执行的指令非法时，stval 会将这条指令的一部分位记录下来。

这里 page fault 就属于第一种情况 通过 `r_scause()` 返回值判断，确认是 page fault 然后为当前合理的虚拟地址申请对应的物理地址，判断逻辑为：如果申请物理地址没成功或者虚拟地址超出范围了，那么杀掉进程。如果申请内存成功了，但如果虚拟地址不合法，需要再释放掉这块内存。中断判断时，如果出错（虚拟地址不合法或者没有成功映射到物理地址），就杀死进程。同时，有些没有实际申请物理空间的，因此没有映射，在接触映射函数进行的时候需要跳过。

3. Lazytests and Usertests

3.1. 实验目的

修改内核代码，以便所有 lazytests 和 usertests 都能通过：

- 处理负的 `sbrk()` 参数
- 如果某个页面的虚拟内存地址高于使用 `sbrk()` 分配的虚拟内存地址，则杀死该进程
- 正确处理 `fork()` 中的父级到子级内存副本
- 处理进程将有效地址从 `sbrk()` 传递给系统调用（例如读取或写入），但尚未分配该地址的内存的情况
- 正确处理内存不足：如果页面故障处理程序中的 `kalloc()` 失败，请终止当前进程
- 处理用户堆栈下方无效页面上的错误

3.2. 实验步骤

- sys_sbrk()参数为负处理：这里参考原来 kernel/proc.c 文件中 growproc() 函数的处理方法 在 kernel/sysproc.c 文件中的 sys_sbrk() 函数对 n 的大小进行判断，同时还需要对 addr + n 是否溢出进行判断。

```
uint64
sys_sbrk(void)
{
    ...
    // lab5-3
    if (n >= 0 && addr + n >= addr) {
        p->sz += n;
    }
    else if (n < 0 && addr + n >= PGROUNDUP(p->trapframe->sp)) {
        p->sz = uvmdealloc(p->pagetable, addr, addr + n);
    }
    else {
        return -1;
    }
    ...
}
```

- 创建子进程：fork() 是通过 uvmcopy() 来进行父进程向子进程的复制，这里修改 kernel/vm.c 文件中 uvmcopy()，将 PTE 不存在和无效两种情况引发的 panic 改为 continue。

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    ...
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walk(old, i, 0)) == 0)
            // panic("uvmcopy: pte should exist");
            continue;
        if((*pte & PTE_V) == 0)
            // panic("uvmcopy: page not present");
            continue;
        ...
    }
    return 0;
}
```

- page fault 虚拟地址超出范围，在 kernel/traps.c 文件的 usertrap() 函数，在该情况时杀死进程：

```

void
usertrap(void)
{
    ...
    if(r_scause() == 8){
        ...
    }
    // lab5-2 page fault的处理
    else if (r_scause() == 13 || r_scause() == 15) {
        char* pa;
        // 获得虚拟地址lab5-3
        uint64 va = r_stval();

        // 判断虚拟地址是否超出范围 lab5-3
        if (va >= p->sz) {
            printf("usertrap(): invalid va=%p higher than p->sz=%p\n", va, p->sz);
            p->killed = 1;
            goto end;
        }
        if (va < PGROUNDDUP(p->trapframe->sp)) {
            printf("usertrap(): invalid va=%p below the user stack sp=%p\n", va, p->trapframe->sp);
            p->killed = 1;
            goto end;
        }

        // 分配物理页面
        if ((pa = kalloc()) != 0) {
            // 用0填充
            memset(pa, 0, PGSIZE);
            // 引发page fault的虚拟地址向下取整
            // uint64 va = PGROUNDDOWN(r_stval());
            va = PGROUNDDOWN(va);
            ...
        }
        ...
    }
    // lab5-3
end:
    if (p->killed)
        exit(-1);
    ...
}

```

- 读写使用未分配的物理内存：read()/write() 两个函数会调用 kernel/vm.c文件中的copyin()/copyout()来完成用户态到核心态的读写，这两个函数都会调用kernel/vm.c文件中的walkaddr()函数完成物理地址到虚拟地址的转换 原本PTE无效、不存在，PTE_U标志位缺失都是异常，但在lazy allocation的情况下，是被允许的 这里需要注意，我们需要先判断虚拟地址是否在用户堆空间范围内，因为 lazy allocation是针对于此，然后才进行相应处理 对kernel/vm.c文件中的的walkaddr()函数处理。

```

uint64
walkaddr(pagetable_t pagetable, uint64 va)
{
    pte_t *pte;
    uint64 pa;
    // lab5-3
    struct proc* p = myproc();

    if(va >= MAXVA)
        return 0;

    pte = walk(pagetable, va, 0);
    // lazy allocation lab5-3
    if (pte == 0 || (*pte & PTE_V) == 0) {
        // 在用户堆空间内
        if (va >= PGROUNDDOWN(p->trapframe->sp) && va < p->sz) {
            char* pa;
            if ((pa = kalloc()) == 0)
                return 0;
            memset(pa, 0, PGSIZE);
            if (mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)pa, PTE_W | PTE_R | PTE_U) != 0) {
                kfree(pa);
                return 0;
            }
        }
    }
    else
        return 0;
}

// if(pte == 0)
//     return 0;
// if((*pte & PTE_V) == 0)
//     return 0;
if((*pte & PTE_U) == 0)
    return 0;
pa = PTE2PA(*pte);
return pa;
}

```

- 编译并测试程序

```

xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
usertrap(): invalid va=0x00000000000004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000001004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000002004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000003004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000004004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000005004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000006004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000007004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000008004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x00000000009004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x0000000000a004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x0000000000b004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x0000000000c004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x0000000000d004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0x0000000000e004000 higher than p->sz=0x00000000000003000
usertrap(): invalid va=0xfffffff80003808 higher than p->sz=0x0000000081003810
test out of memory: OK
ALL TESTS PASSED

```

- 单元测试

```
lq@vm:~/xv6-labs-2020$ ./grade-lab-laz
make: 'kernel/kernel' is up to date.
== Test running lazytests == (6.1s)
== Test lazy: map ==
lazy: map: OK
== Test lazy: unmap ==
lazy: unmap: OK
== Test usertests == (250.3s)
== Test usertests: pbug ==
usertests: pbug: OK
== Test usertests: sbrkbugs ==
usertests: sbrkbugs: OK
== Test usertests: argptest ==
usertests: argptest: OK
== Test usertests: sbrkmuch ==
usertests: sbrkmuch: OK
== Test usertests: sbrkfail ==
usertests: sbrkfail: OK
== Test usertests: sbrkarg ==
usertests: sbrkarg: OK
== Test usertests: stacktest ==
usertests: stacktest: OK
== Test usertests: execout ==
usertests: execout: OK
== Test usertests: copyin ==
usertests: copyin: OK
```

```
== Test usertests: iput ==
usertests: iput: OK
== Test usertests: mem ==
usertests: mem: OK
== Test usertests: pipe1 ==
usertests: pipe1: OK
== Test usertests: preempt ==
usertests: preempt: OK
== Test usertests: exitwait ==
usertests: exitwait: OK
== Test usertests: rmdot ==
usertests: rmdot: OK
== Test usertests: fourteen ==
usertests: fourteen: OK
== Test usertests: bigfile ==
usertests: bigfile: OK
== Test usertests: dirfile ==
usertests: dirfile: OK
== Test usertests: iref ==
usertests: iref: OK
== Test usertests: forktest ==
usertests: forktest: OK
== Test time ==
time: OK
Score: 119/119
lq@vm:~/xv6-labs-2020$
```

3.3. 实验小结

- 问题与解决方法

遇到以下两个错误

```
kernel/vm.c:109:26: error: dereferencing pointer to incomplete type 'struct proc'
```

```
In file included from kernel/vm.c:8:
kernel/proc.h:87:19: error: field 'lock' has incomplete type
  87 |   struct spinlock lock;
      |           ^~~~
```

根据提示，应该按顺序添加两个头文件

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

- 实验心得

通过本次实验，我更加深刻地了解到什么是 lazy allocation，利用退出资源的分配来节省内存和提高性能，在实现的过程中，一直遵循着“真正需要的时候才分配物理内存”的原则。具体来说，我们在 xv6 系统中利用 page fault 来实现，用户态通过 sbrk() 进行对 heap 上内存的增加或减少，如果申请的空间很大，将会花费很多时间。这时候引入了 lazy allocation，等到实际用到的时候，会引发 page fault，在 usertrap 中处理，为其申请物理内存空间。通过这种方法，将一次进行大量申请空间的开销分散到读写内存中，从而一定程度上提高交互性。我们还需要清楚地了解几个寄存器的定义。

- scause 存储中断类型
- stval 存储发生中断时访问的虚拟地址

同时对于中断类型，本实验用到的有

- scause 为13时对应 Load page fault
- scause 为15时对应 Store/AMO page fault

Lab6 : Copy-on-Write Fork for xv6

切换分支

```
$ git fetch  
$ git checkout cow  
$ make clean
```

1. Implement copy-on write

1.1. 实验目的

在 xv6 内核中实现 copy-on-write fork。如果修改后的内核同时成功执行 cowtest 和 usertests 程序就完成了。

1.2. 实验步骤

- 设置 COW 标记位，从而可以区分一般的 page fault 和 COW 引发的页面错误 利用 PTE 中保留某个比特位设置即可。在 kernel/riscv.h 文件中添加：

```
// lab6  
#define PTE_COW (1L << 8)
```

- 我们增加一个数据结构帮助处理页面使用的计数 并且添加计数增加和减少的方法 注意其中需要利用自旋锁，从而实现进程间的互斥 在 kernel/defs.h 增添这两个函数的声明

```
// lab6  
void add_cnt(uint64);  
uint8 sub_cnt(uint64);
```

- 在 Makefile 中添加 \$K/cow.o

```
...  
$K/virtio_disk.o \  
$K/cow.o  
...
```

- 新建 kernel/cow.c 文件，实现上述操作：

```

#include "types.h"
#include "memlayout.h"
#include "spinlock.h"
#include "riscv.h"
#include "defs.h"

// 计数结构
struct {
    uint8 cnt;
    struct spinlock lock;
    //12 就是页面大小
} cow[(PHYSTOP - KERNBASE) >> 12];

// 增加
void add_cnt(uint64 pa) {
    if (pa < KERNBASE)
        return;

    pa = (pa - KERNBASE) >> 12;
    acquire(&cow[pa].lock);
    ++cow[pa].cnt;
    release(&cow[pa].lock);
}

// 减少 需要返回判断是否为零
uint8 sub_cnt(uint64 pa) {
    uint8 ans;
    if (pa < KERNBASE)
        return 0;
    pa = (pa - KERNBASE) >> 12;
    acquire(&cow[pa].lock);
    ans = --cow[pa].cnt;
    release(&cow[pa].lock);
    return ans;
}

```

- 使用 fork() 函数创建子进程的时候，其中调用 uvmcopy() 函数将父进程的用户页表复制到子进程中。在 COW 中，不会复制，而是将子进程虚拟页同样映射在与父进程相同的物理页上 因此，我们需要将写标志位 PTE_W移除，添加 COW 标志位 PTE_COW 修改 kernel/vm.c 文件中的uvmcopy()函数：

```

int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;
    // 不分配实际物理内存lab6
    // char *mem;

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

        // 清除PTE_W标志位 增加COW标志位lab6
        flags = PTE_FLAGS(*pte & (~PTE_W)) | PTE_COW;
        *pte = PA2PTE(pa) | flags;

        // 不分配实际物理内存lab6
        // if((mem = kalloc()) == 0)
        //     goto err;
        // memmove(mem, (char*)pa, PGSIZE);
        if(mappages(new, i, PGSIZE, pa, flags) != 0){
            // kfree(mem);
            goto err;
        }
        // 计数增加
        add_cnt(pa);
    }

    return 0;
}

err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}

```

- 修改 usertrap() 和 copyout() 函数，两处的写入的页面错误错误均需要进行相同的操作，于是我们参考 walkaddr() 在 kernel/vm.c 文件中添加一个函数 walkcowaddr():

```

uint64 walkcowaddr(pagetable_t pagetable, uint64 va) {
    uint64 pa;
    char* mem;
    pte_t* pte;
    uint flag;

    // 判断范围
    if (va >= MAXVA)
        return 0;
    // 找到虚拟地址对应的pte
    pte = walk(pagetable, va, 0);
    if (pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0)
        return 0;
    pa = PTE2PA(*pte);
    if ((*pte & PTE_W) == 0) {
        if ((*pte & PTE_COW) == 0)
            return 0;
    }
    // 分配新的物理内存
    if ((mem = kalloc()) == 0)
        return 0;
    // 复制页表内容
    memmove(mem, (void*)pa, PGSIZE);
    // 取消COW标志位变为写标志
    flag = (PTE_FLAGS(*pte) & (~PTE_COW)) | PTE_W;

    // 取消原来的映射 映射到新分配的物理内存
    uvmunmap(pagetable, PGROUNDDOWN(va), 1, 1);
    if (!mappages(pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem, flag) != 0) {
        kfree(mem);
        return 0;
    }
    return (uint64)mem;
}
return pa;
}

```

- 在 kernel/proc.c 文件中 usertrap() 函数中调用上述 walkcowaddr():

```

void
usertrap(void)
{
    ...
    ...
    if(r_scause() == 8){
        ...
    }
    // 写页面错误 lab6
    else if (r_scause() == 15) {
        uint64 va = r_stval();
        if (walkcowaddr(p->pagetable, va) == 0) {
            goto end;
        }
    }
    else if((which_dev = devintr()) != 0){
        // ok
    } else {
        // lab6
    }
end:
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("          sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
}

if(p->killed)
    exit(-1);
...
}

```

- 在 kernel/vm.c 文件中 copyout() 函数中调用上述 walkcowaddr():

```

int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;

    while(len > 0){
        va0 = PGROUNDDOWN(dstva);
        // pa0 = walkaddr(pagetable, va0);
        pa0 = walkcowaddr(pagetable, va0);
        if(pa0 == 0)
            return -1;
        ...
    }
    return 0;
}

```

- 修改 kernel/kalloc.c 中的 kalloc() 函数，这里将一个物理页面分配给一个进程，计数增加

```

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

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
        kmem.freelist = r->next;
    release(&kmem.lock);
    // 计数增加 lab6
    add_cnt((uint64)r);
    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}

```

- 修改 kernel/kalloc.c 中的 kalloc() 函数，这里负责释放一个页面，计数减少，并且需要判断是否减少为零，计数为零才真正释放物理页面。

```

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

    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");
    // 计数减少lab6 不为零直接退出不用释放物理页面
    if (sub_cnt((uint64)pa))
        return;
    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);
    ...
}

```

- 修改 kernel/kalloc.c 文件中的 freerange() 函数 这里负责把空闲物理页面送给 kfree() 调用，这里需要先计数增加，保证第一次引用减少后不会越界，同时也刚好可以正常计数

```

void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE) {
        // 先增加计数lab6
        add_cnt((uint64)p);
        kfree(p);
    }
}

```

- 修改 trap.c、defs.h：

```

void
usertrap(void)
{
    ...
    ...
    if(r_scause() == 8){
        ...
    }
    // 写页面错误 lab6
    else if (r_scause() == 15) {
        uint64 va = r_stval();
        if (walkcowaddr(p->pagetable, va) == 0) {
            goto end;
        }
    }
    else if((which_dev = devintr()) != 0){
        // ok
    } else {
        // lab6
    }
end:
    ...
}

```

- 编译并测试程序

```

$ cowtest
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
$ usertests
== COW TESTS PASSED
$ usertests
usertests starting
test execut: usertrap(): unexpected scause 0x000000000000000f pid=19
    sepc=0x000000000002abe stval=0x0000000000010b88
OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsrbk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3248
    sepc=0x000000000005556 stval=0x000000000005556
usertrap(): unexpected scause 0x000000000000000c pid=3249
    sepc=0x000000000005556 stval=0x000000000005556
OK
test badarg: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdat: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED

```

- 单元测试

```

lq@vm:~/xv6-labs-2020$ ./grade-lab-cow
make: 'kernel/kernel' is up to date.
== Test running cowtest == (12.2s)
[INFO] Test simple ==
simple: OK
== Test three ==
three: OK
== Test file ==
file: OK
== Test usertests == (225.7s)
== 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

```

1.3. 实验小结

- 实验理解

本实验的基本思想和上一个实验的思想我认为有相似之处，都是对于大内存空间申请分配的时候做出的优化。不同点在于，上一个的虚拟地址是完全没有对应的物理内存的，并且只要访问到发生page fault 就会申请物理内存；但本实验的重点在于 fork() 创建子进程的时候，会让父子进程共享所有的物理页，但是这些都标记为只读，两个进程之间只要有一个要写入，就会引发缺页错误 Store page fault，这时候才申请物理空间，复制页表并重新映射。

- spinlock

这个自旋锁，在对于引用页表计数的时候起到了关键作用，结合在操作系统理论课上学到的互斥概念，我们对多个进程都有可能更改的变量访问的时候需要加锁，保证同一时间只有一个进程对其访问 其中自旋锁有一些特性，例如忙等待的锁机制：如果一个内核代码试图获取一个被持有的自旋锁，那么这段内核代码需要一直忙等待，直到自旋锁释放；同时要求自旋锁持有者尽快完成临界区的执行任务。如果临界区中的执行时间过长，在锁外面忙等待的CPU比较浪费，特别是自旋锁临界区里不能睡眠；自旋锁可以在中断上下文中使用。

- 编译链接

为什么在 Makefile 中添加 `$k/cow.o` 呢？为不是像之前的实验一样。原因在于这次实验我们是在 kernel 目录下新建文件，因此链接的时候要加进来。

推送至仓库

```
lq@lq:~/xv6-labs-2020$ git add .
lq@lq:~/xv6-labs-2020$ git commit -m "2025-08-09:lab6"
[cow d09f911] 2025-08-09:lab6
 8 files changed, 127 insertions(+), 12 deletions(-)
create mode 100644 kernel/cow.c
create mode 100644 kernel/cow.h
lq@lq:~/xv6-labs-2020$ git push github cow:cov
fatal: unable to access 'https://github.com/lq1911/OS.git/': Could not resolve host: github.com
lq@lq:~/xv6-labs-2020$ export https_proxy=http://100.80.116.35:7890;export http_proxy=http://100.80.116.35:7890
lq@lq:~/xv6-labs-2020$ git push github cow:cov
Enumerating objects: 69, done.
Counting objects: 100% (54/54), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (27/27), done.
Writing objects: 100% (36/36), 14.00 KiB | 2.00 MiB/s, done.
Total 36 (delta 21), reused 14 (delta 6), pack-reused 0
remote: Resolving deltas: 100% (21/21), completed with 13 local objects.
remote:
remote: Create a pull request for 'cov' on GitHub by visiting:
remote:   https://github.com/lq1911/OS/pull/new/cov
remote:
To https://github.com/lq1911/OS.git
 * [new branch]      cov > cov
```

Lab7 : Multithreading

切换分支

```
$ git fetch
$ git checkout thread
$ make clean
```

1. Uthread: switching between threads

1.1. 实验目的

为用户级线程系统设计上下文切换机制，然后实现它。

1.2. 实验步骤

- 参考 kernel/proc.h 文件中的 struct context 进程上下文结构体，在 user/uthread.c 文件较为靠前的位置中声明线程上下文 struct thread_context 结构体：

```

// 线程上下文lab7
struct thread_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;
};

};


```

- 在 struct thread 线程结构体中添加数据成员记录上下文：

```

struct thread {
    char      stack[STACK_SIZE]; /* the thread's stack */
    int       state;           /* FREE, RUNNING, RUNNABLE */
    // 上下文lab7
    struct thread_context thrdctx;
};


```

- 在 thread_create 线程创建函数初始化上下文数据成员 该函数遍历所有线程，将未初始化的线程进行初始化：

```

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

    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == FREE) break;
    }
    t->state = RUNNABLE;
    // YOUR CODE HERE
    // 初始化上下数据成员 lab7
    t->thrdctx.ra = (uint64)func;
    t->thrdctx.sp = (uint64)t->stack + STACK_SIZE;
}

```

- 修改 thread_schedule() 函数，在其中调用 thread_switch() 线程切换函数：

```

void
thread_schedule(void)
{
    ...
    if (current_thread != next_thread) {          /* switch threads? */
        next_thread->state = RUNNING;
        t = current_thread;
        current_thread = next_thread;
        /* YOUR CODE HERE
         * Invoke thread_switch to switch from t to next_thread:
         * thread_switch(??, ??);
         */
        thread_switch((uint64)&t->thrdctx, (uint64)&current_thread->thrdctx);
    } else
        next_thread = 0;
}

```

- 参照 kernel/swtch.S 中的 swtch() 函数，在 user/uthread_switch.S 文件中编写线程的上下文切换函数 thread_switch：

```
.text
/*
 * save the old thread's registers,
 * restore the new thread's registers.
 */
.globl thread_switch
thread_switch:
    /* YOUR CODE HERE */
    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)

    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 */
```

- 编译并测试程序

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 2
thread_c 3
thread_a 3
thread_b 3
thread_c 4
thread_a 4
thread_b 4
thread_c 5
thread_a 5
```

```
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
$ QEMU: Terminated
```

- 单元测试

```
lq@vm:~/xv6-labs-2020$ ./grade-lab-thread uthread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (3.4s)
```

1.3. 实验小结

本实验让我了解到了用户线程切换上下文的基本步骤 基本上大部分都可以参照 xv6 系统内部进程调度的代码进行编写，比较简单。以下两点是我在写代码的时候比较疑惑的地方，这里展开分析。

其中的 ra 寄存器的作用之前的实验已经提到过也使用过。它的全称为 Return Address，用来保存线程切换的返回地址。当一个线程被抢占或者暂停执行的时候，当前线程的执行状态需要保存，在切换回该线程的时候，程序才可以通过 ra 寄存器中保存的返回地址恢复到之前被恢复的执行点。

而对于 sp 寄存器，全称为 Stack Pointer，指示了当前线程的栈顶位置。在线程切换之前，操作系统会保存当前线程的上下文信息，包括程序计数器、通用寄存器等，以便稍后恢复该线程的执行状态。其中也包括保存当前线程的栈指针值。操作系统决定切换到另一个线程时，它会加载下一个线程的上下文信息，包括栈指针值。通过将 sp 寄存器设置为下一个线程的栈指针值，操作系统可以确保下一个线程从正确的栈位置开始执行。操作系统决定切换到另一个线程时，它会加载下一个线程的上下文信息，包括栈指针值。通过将 sp 寄存器设置为下一个线程的栈指针值，操作系统可以确保下一个线程从正确的栈位置开始执行。

2. Using threads

2.1. 实验目的

修改代码，使某些 put 操作在保持正确性的同时并行运行。当 make grade 表示您的代码通过了 ph_safe 和 ph_fast 测试时，您就完成了。

2.2. 实验步骤

• 未改进测试

- 构建包含不安全线程的哈希表的 ph 程序 执行 make ph 命令：

```
lq@vm:~/xv6-labs-2020$ make ph
gcc -o ph -g -O2 notxv6/ph.c -pthread
```

- 运行ph 1程序，即使用单线程运行该哈希表，查看输出执行 ./ph 1 命令：

```
lq@vm:~/xv6-labs-2020$ ./ph 1
100000 puts, 17.549 seconds, 5698 puts/second
0: 0 keys missing
100000 gets, 17.700 seconds, 5650 gets/second
```

没有键丢失

- 运行ph 2程序，即使用两个线程运行该哈希表，查看输出执行 ./ph 2 命令

```
lq@vm:~/xv6-labs-2020$ ./ph 2
100000 puts, 6.626 seconds, 15092 puts/second
1: 16836 keys missing
0: 16836 keys missing
200000 gets, 15.889 seconds, 12588 gets/second
```

时间加快但存在键丢失的问题

• 回答问题

多个线程同时调用 put() 时，有一定几率对同一个桶的数据进行操作，前一个操作还未完成，后一个就将前一个操作的数据覆盖。我们可以设置素数个数的桶，在取模的同时降低映射到同一个桶的可能性。

我们为每一个桶加锁，虽然保证了同一个时间一个桶的数据仅能被一个线程访问并修改，但可能会降低并发性。

• 编写代码

- 我们在此利用互斥锁解决线程不安全问题。对于哈希表，当多个线程同时对一个桶操作的时候，有可能造成数据丢失，因此我们给每个桶配置一个互斥锁。

在 notxv6/ph.c 文件中声明互斥锁数组：

```
// 一个桶一个的互斥锁数组 lab7-2
pthread_mutex_t lock[NBUCKET];
```

- 在 notxv6/ph.c 文件中的 main() 函数初始化互斥锁：

```

int
main(int argc, char *argv[])
{
    ...
    // 初始化互斥锁 lab7
    for (int i = 0; i < NBUCKET; ++i) {
        pthread_mutex_init(&lock[i], NULL);
    }
}

```

- 在 notxv6/ph.c 文件中的 put() 函数中的 insert() 操作的前后加上互斥锁的相关操作，保证操作的互斥性：

```

static
void put(int key, int value)
{
    ...
    // update the existing key.
    e->value = value;
} else {
    // 加锁
    pthread_mutex_lock(&lock[i]);
    // the new is new.
    insert(key, value, &table[i], table[i]);
    // 解锁
    pthread_mutex_unlock(&lock[i]);
}
}

```

• 测试程序

执行 make ph 编译文件，执行 ./ph 2 进行测试：

```

lq@vm:~/xv6-labs-2020$ ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (1.3s)
== Test running alarmtest == (4.6s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests == usertests: OK (283.9s)
(Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 85/85

```

• 单元测试

```

2000000 getcs, 14.924 seconds, 15402 getcs/second
lq@vm:~/xv6-labs-2020$ ./grade-lab-thread ph_fast
make: 'kernel/kernel' is up to date.
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (54.5s)
lq@vm:~/xv6-labs-2020$ ./grade-lab-thread ph_safe
make: 'kernel/kernel' is up to date.
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (22.8s)

```

2.3. 实验小结

本次实验相当于是对锁的应用，之前的实验中也有使用过，因此没什么问题，比较基础，使用互斥锁保证数据不会同时被多个线程访问和修改造成丢失的情况。

当多个线程同时访问共享资源时，可能会导致数据不一致或者其他问题。为了避免这种情况，可以使用线程互斥锁来实现线程间的互斥访问。

线程互斥锁是一种同步机制，用于保护共享资源，确保在任意时刻只有一个线程可以访问该资源。

使用线程互斥锁可以有效地避免多个线程同时访问共享资源而引发的竞态条件和数据不一致问题。然而，过度使用互斥锁可能会导致性能下降，因为它会引入线程间的竞争和等待。因此，在设计多线程程序时，需要权衡使用互斥锁的粒度和性能之间的关系。

3. Barrier

3.1. 实验目的

实现一个屏障 Barrier：应用程序中的一个点，所有参与的线程在此点上必须等待，直到所有其他参与线程也达到该点。

3.2. 实验步骤

- 未改进测试

编译 barrier 程序，它可以使得多个线程执行至同一位置后在继续执行，执行 `make barrier`，执行 `./barrier 2` 命令，使两个线程运行：

```
pi@raspberrypi:~/xv6-labs-2020$ make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
pi@raspberrypi:~/xv6-labs-2020$ ./barrier 2
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
Aborted (core dumped)
```

发现程序报错

- 编写代码

完成 barrier() 函数的实现，通过记录到达屏障点的线程数量，进行线程的唤醒或等待，从而实现该函数的功能：

```
static void
barrier()
{
    // YOUR CODE HERE
    //

    // Block until all threads have called barrier() and
    // then increment bstate.round.
    //

    // 加锁
    pthread_mutex_lock(&bstate.barrier_mutex);
    // 计数增加
    ++bstate.nthread;
    // 到达数量
    if (bstate.nthread == nthread) {
        ++bstate.round;
        // 重置
        bstate.nthread = 0;
        // 广播唤醒
        pthread_cond_broadcast(&bstate.barrier_cond);
    }
    else {
        // 没到达数量继续等待
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }
    // 解锁
    pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

- 测试程序

执行 `make barrier` 重新编译，依次执行 `./barrier 1`，`./barrier 2`，`./barrier 5` 测试：

```
pi@raspberrypi:~/xv6-labs-2020$ make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
pi@raspberrypi:~/xv6-labs-2020$ ./barrier 1
OK; passed
pi@raspberrypi:~/xv6-labs-2020$ ./barrier 2
OK; passed
pi@raspberrypi:~/xv6-labs-2020$ ./barrier 5
OK; passed
```

- 单元测试

```
pi@raspberrypi:~/xv6-labs-2020$ ./grade-lab-thread barrier
make: 'kernel/kernel' is up to date.
== Test barrier == make: 'barrier' is up to date.
barrier: OK (249.6s)
```

```
pi@raspberrypi:~/xv6-labs-2020$ ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (3.4s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (22.6s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (57.8s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (249.7s)
== Test time ==
time: OK
Score: 60/60
```

3.3. 实验小结

本实验是对锁的另一个使用，从而实现线程同步功能。让我深刻地理解到，原来同步不只有之前操作系统理论课上提到的信号量帮助实现。

下面介绍实现的两个比较重要的函数 `pthread_cond_wait()` 函数用于在多线程编程中实现条件变量的等待操作。条件变量是一种线程间同步的机制，它允许一个或多个线程等待某个特定条件的发生 `pthread_cond_broadcast()` 函数用于将当前线程阻塞，并等待条件变量的信号。当其他线程调用 `pthread_cond_signal()` 或 `pthread_cond_broadcast()` 发送信号时，被阻塞的线程将被唤醒并继续执行。

本实验中就是等待到了一定数量的线程到达某点后就执行 `pthread_cond_broadcast()` 发送信号唤醒线程 `pthread_cond_wait()` 函数需要与互斥锁一起使用，以确保线程在等待条件变量时不会出现竞态条件。

在调用 `pthread_cond_wait()` 之前，通常需要先获取互斥锁，然后在等待期间释放互斥锁，以允许其他线程修改共享数据。当线程被唤醒后，它会重新获取互斥锁，并检查条件是否满足 `pthread_cond_broadcast()` 函数用于向所有等待在特定条件变量上的线程发送信号，它会唤醒所有等待在该条件变量上的线程，这些线程之后可以竞争获取锁并继续执行。这个函数广播一个信号给所有等待线程，相当于通知它们某个条件已经满足，可以继续执行了。

同时，他也需要配合互斥锁一起使用。一般的使用方式是，在修改共享数据之前，先获取互斥锁，然后检查条件是否满足，如果不满足，则调用 `pthread_cond_wait()` 等待条件满足。当其他线程修改了共享数据，并调用 `pthread_cond_broadcast()` 发送信号后，等待的线程会被唤醒，再次检查条件是否满足，如果满足则继续执行，否则继续等待。

推送至仓库

```
Score: 60/60
lq@vn:/xv6-labs-2028$ git add .
lq@vn:/xv6-labs-2028$ git commit -m "2025-08-09:lab7"
[thred 884b24e] 2025-08-09:lab7
 6 files changed, 96 insertions(+), 3 deletions(-)
  create mode 100644 answers-thread.txt
  create mode 100644 time.txt
lq@vn:/xv6-labs-2028$ git push github thread:thread
Enumerating objects: 142, done.
Counting objects: 100% (128/128), done.
Delta compression using up to 4 threads
Compressing objects: 100% (66/66), done.
Writing objects: 100% (110/110), 25.49 KiB | 3.19 MiB/s, done.
Total 110 (delta 68), reused 72 (delta 41), pack-reused 0
remote: Resolving deltas: 100% (68/68), completed with 16 local objects.
remote:
remote: Create a pull request for 'thread' on GitHub by visiting:
remote:   https://github.com/lq1911/OS/pull/new/thread
remote:
To https://github.com/lq1911/OS.git
 * [new branch]      thread -> thread
```

Lab 8 : locks

切换分支

```
$ git fetch
$ git checkout lock
$ make clean
```

1. Memory allocator

1.1. 实验目的

实现每个 CPU 的空闲列表，并在 CPU 的空闲列表为空时进行窃取。必须提供所有以 `kmem` 开头的锁名称。也就是说，您应该为每个锁调用 `initlock`，并传递以 `kmem` 开头的名称。

1.2. 实验步骤

- 在 `kernel/kalloc.c` 文件中修改 `kmem` 的定义，变为一个 `kmem` 数组，使得每个 CPU 都有对应的 `freelist` 和 `lck`：

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

- 在 `kernel/kalloc.c` 文件中修改 `kmem` 的初始化函数 `kinit()`，为 `kmem` 数组中的每个元素进行初始化：

```

void
kinit()
{
    // 给每个kmem初始化 lab8-1
    for (int i = 0; i < NCPU; ++i)
        initlock(&kmem[i].lock, "kmem");
    freerange(end, (void*)PHYSTOP);
}

```

- 在 kernel/kalloc.c 文件中修改 kfree() 函数，获取当前 CPUID 时先关闭中断，然后再对该 CPU 执行相应的锁操作：

```

void
kfree(void *pa)
{
    ...
    r = (struct run*)pa;
    // 关闭中断 获取那个CPU lab8-1
    push_off();
    int index = cpuid();
    pop_off();

    acquire(&kmem[index].lock);
    r->next = kmem[index].freelist;
    kmem[index].freelist = r;
    release(&kmem[index].lock);
}

```

- 在 kernel/kalloc.c 文件中修改 kalloc() 函数，获取当前 CPUID 时先关闭中断，查找当前 CPU 有无空闲块，有就返回，没有就找 CPU 中的空闲块返回：

```

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

    // 关闭中断 获取那个CPU
    push_off();
    int index = cpuid();
    pop_off();

    acquire(&kmem[index].lock);
    r = kmem[index].freelist;
    // 有空闲块
    if(r)
        kmem[index].freelist = r->next;
    else {
        for (int i = 0; i < NCPU; ++i) {
            if (i == index)
                continue;
            acquire(&kmem[i].lock);
            r = kmem[i].freelist;
            if (r)
                kmem[i].freelist = r->next;
            release(&kmem[i].lock);
            if (r)
                break;
        }
    }
    release(&kmem[index].lock);
    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}

```

- 编译并测试程序

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ kalloc test
start test1
test1 results:
... lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 195794
lock: kmem: #fetch-and-add 0 #acquire() 41363
lock: kmem: #fetch-and-add 0 #acquire() 195904
lock: bcache: #fetch-and-add 0 #acquire() 1248
... top 5 contended locks:
lock: virtio_disk: #fetch-and-add 309714 #acquire() 114
lock: proc: #fetch-and-add 28393 #acquire() 118451
lock: proc: #fetch-and-add 17610 #acquire() 118466
lock: proc: #fetch-and-add 8137 #acquire() 118629
lock: proc: #fetch-and-add 6308 #acquire() 118619
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

```
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
```

1.3. 实验小结

根据实验提示，本实验的逻辑比较简单，基本上都是对 kmem 数组根据当前使用的 CPU 进行定位，找到CPU的内存块。

实验中，这段提示看得不是很明白：cpuid 函数返回当前内核号，但是只有在中断关闭时调用它并使用其结果才是安全的。您应该使用 push_off() 和 pop_off() 来打开和关闭中断。

于是我们上网搜索相关资料，得到：

在操作系统中，中断是一种机制，允许外部事件（如硬件设备的输入/输出请求）打断正在执行的程序，并立即处理这些事件。当中断发生时，处理器会暂停当前正在执行的指令，保存当前的上下文，并跳转到相应的中断处理程序。

在多任务操作系统中，同时可能有多个任务在运行，这些任务共享计算机的资源。为了确保任务之间的正确协作和资源的正确使用，操作系统需要对中断进行管理和控制。

当调用 cpuid 函数时，它会读取当前内核号并返回结果。然而，在多任务环境中，其他任务可能会在任何时候被调度并运行，包括在 cpuid 函数执行期间。如果在执行 cpuid 函数期间发生中断，可能会导致内核号的值变化，从而导致不正确的结果。

为了避免这种情况，可以使用 push_off() 和 pop_off() 函数来关闭和打开中断。通过调用 push_off() 函数，可以禁用中断，确保在执行 cpuid 函数期间不会发生中断。然后，在获取到 cpuid 函数的结果后，可以调用 pop_off() 函数来恢复中断状态，使得其他任务能够继续正常运行。

通过使用 push_off() 和 pop_off() 函数来关闭和打开中断，可以确保在执行 cpuid 函数期间不会发生中断，从而保证获取到的内核号是准确和安全的。这样可以避免由于中断引起的竞态条件和不一致性问题。

2. Buffer cache

2.1. 实验目的

修改块缓存，使 bcache 中所有锁的 acquire 循环迭代次数在运行 bcachetest 时接近于零。理想情况下，块缓存中涉及的所有锁的计数之和应该为零，但如果总和小于 500 也可以。修改 bget 和 brelse，以便 bcache 中不同块的并发查找和释放不太可能在锁上发生冲突（例如，不必全部等待 bcache.lock）。

2.2. 实验步骤

- 修改 kernel/buf.h 文件中的 buf 结构体定义，添加数据成员 tick 记录上次使用时间，用于 LRU 方案：

```
struct buf {
    ...
    uchar data[BSIZE];
    // 用于LRU记录上次使用时间 lab8-2
    uint tick;
};
```

- 修改 kernel/bio.c 文件中 bcache 的定义 定义素数个数的桶，从而尽可能地减少桶冲突 为每个桶 bcache 中的每个桶都加上锁：

```

struct {
    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.

    // lab8-2
    struct spinlock big_lock;
    struct spinlock lock[NBUCKET];
    struct buf head[NBUCKET];
} bcache;

```

- 在 kernel/bio.c 文件中添加哈希函数，从而能通过块号映射到对应的桶：

```

// 哈希函数
int hash(int blockno) {
    return blockno % NBUCKET;
}

```

- 修改 kernel/bio.c 文件中 bcache 的初始化函数 binit()：

```

void
binit(void)
{
    struct buf *b;

    initlock(&bcache.big_lock, "bcache_big_lock");
    // Create linked list of buffers
    // bcache.head.prev = &bcache.head;
    // bcache.head.next = &bcache.head;
    // 初始化桶链表 lab8-2
    for (int i = 0; i < NBUCKET; ++i) {
        initlock(&bcache.lock[i], "bcache_bucket");
        bcache.head[i].prev = &bcache.head[i];
        bcache.head[i].next = &bcache.head[i];
    }
    // 初始化buf lab8-2
    for(b = bcache.buf; b < bcache.buf+NBUF; b++){
        b->next = bcache.head[0].next;
        b->prev = &bcache.head[0];
        initsleeplock(&b->lock, "buffer");
        bcache.head[0].next->prev = b;
        bcache.head[0].next = b;
    }
}

```

- 在 kernel/bio.c 文件中修改 bget() 函数，判断是否命中，若命中就返回；若不命中，按从大到小的顺序解锁，此时可能有新缓存，仍要判断是否命中；若仍不命中，就按 LRU 的方式查找获取空闲块；若没有就去别的桶获取空闲块：

```

static struct buf*
bget(uint dev, uint blockno)
{
    struct buf *b;
    // lab8-2
    int index = hash(blockno);
    int min_tick = __UINT32_MAX__;
    struct buf* target_buf = 0;

    acquire(&bcache.lock[index]);

    // Is the block already cached?
    // 命中
    for(b = bcache.head[index].next; b != &bcache.head[index]; b = b->next){
        if(b->dev == dev && b->blockno == blockno){
            b->refcnt++;
            // 记录使用时间lab8-2
            // b->tick = ticks;
            release(&bcache.lock[index]);
            acquiresleep(&b->lock);
            return b;
        }
    }
    release(&bcache.lock[index]);

    // 不命中
    // Not cached.
    // Recycle the least recently used (LRU) unused buffer.
    acquire(&bcache.big_lock);
    acquire(&bcache.lock[index]);
    // 当前桶中找空闲
    for (b = bcache.head[index].next; b != &bcache.head[index]; b = b->next) {
        if (b->dev == dev && b->blockno == blockno) {
            b->refcnt++;
            release(&bcache.lock[index]);
            release(&bcache.big_lock);
            acquiresleep(&b->lock);
            return b;
        }
    }
    for (b = bcache.head[index].next; b != &bcache.head[index]; b = b->next) {
        if (b->refcnt == 0 && (target_buf == 0 || b->tick < min_tick)) {
            min_tick = b->tick;
            target_buf = b;
        }
    }

    if (target_buf) {
        target_buf->dev = dev;
        target_buf->blockno = blockno;
        target_buf->refcnt++;
        target_buf->valid = 0;
        release(&bcache.lock[index]);
        release(&bcache.big_lock);
        acquiresleep(&target_buf->lock);
        return target_buf;
    }

    // 从其他桶找空闲块
    for (int i = hash(index + 1); i != index; i = hash(i + 1)) {
        acquire(&bcache.lock[i]);
        for (b = bcache.head[i].next; b != &bcache.head[i]; b = b->next) {
            if (b->refcnt == 0 && (target_buf == 0 || b->tick < min_tick)) {
                min_tick = b->tick;
            }
        }
    }
}

```

```

        target_buf = b;
    }
}

if (target_buf) {
    target_buf->dev = dev;
    target_buf->refcnt++;
    target_buf->valid = 0;
    target_buf->blockno = blockno;
    // 从原桶中删除
    target_buf->next->prev = target_buf->prev;
    target_buf->prev->next = target_buf->next;
    release(&bcache.lock[i]);
    //加锁
    target_buf->next = bcache.head[index].next;
    target_buf->prev = &bcache.head[index];
    bcache.head[index].next->prev = target_buf;
    bcache.head[index].next = target_buf;
    release(&bcache.lock[index]);
    release(&bcache.big_lock);
    acquireSleep(&target_buf->lock);
    return target_buf;
}
release(&bcache.lock[i]);
}

release(&bcache.lock[index]);
release(&bcache.big_lock);
panic("bget: no buffers");
}

```

- 在 kernel/bio.c 文件中 brelse() 中调用哈希函数，决定对应那个桶：

```

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

    releasesleep(&b->lock);

    // lab8-2
    int index = hash(b->blockno);

    acquire(&bcache.lock[index]);
    b->refcnt--;
    if (b->refcnt == 0) {
        // LRU记录辅助
        b->tick = ticks;
    }

    release(&bcache.lock[index]);
}

```

- 在 kernel/bio.c 文件中修改 bpin() 函数和 bunpin() 函数：

```

void
bpin(struct buf *b) {
    // lab8-2
    int index = hash(b->blockno);
    acquire(&bcache.lock[index]);
    b->refcnt++;
    release(&bcache.lock[index]);
}

void
bunpin(struct buf *b) {
    // lab8-2
    int index = hash(b->blockno);
    acquire(&bcache.lock[index]);
    b->refcnt--;
    release(&bcache.lock[index]);
}

```

- 编译并测试程序

```

test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdat: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED

```

```

lock: bcache_big_lock: #fetch-and-add 0 #acquire() 115
lock: bcache_bucket: #fetch-and-add 0 #acquire() 2132
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4128
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4326
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6330
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6333
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6324
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6699
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6706
lock: bcache_bucket: #fetch-and-add 0 #acquire() 7744
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4138
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4148
lock: bcache_bucket: #fetch-and-add 0 #acquire() 2131
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4140
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 155108 #acquire() 1229
lock: proc: #fetch-and-add 52892 #acquire() 81748
lock: proc: #fetch-and-add 25256 #acquire() 81368
lock: proc: #fetch-and-add 20633 #acquire() 81368
lock: proc: #fetch-and-add 19571 #acquire() 81390
tot= 0
test0: OK
start test1
test1 OK

```

- 单元测试

```

lq@vm:~/xv6-labs-2020$ ./grade-lab-lock
make: 'kernel/kernel' is up to date.
== Test running kalloc test == (172.6s)
== Test kalloc test: test1 ==
    kalloc test: test1: OK
== Test kalloc test: test2 ==
    kalloc test: test2: OK
== Test kalloc test: sbrkmuch == kalloc test: sbrkmuch: OK (18.5s)
== Test running bcache test == (20.1s)
== Test bcache test: test0 ==
    bcache test: test0: OK
== Test bcache test: test1 ==
    bcache test: test1: OK
== Test user tests == user tests: OK (241.1s)
== Test time ==
time: OK
Score: 70/70

```

2.3. 实验小结

在本次实验中，我深感数据结构的重要性，对于链表的操作一不小心就会造成程序的错误，最后还是得在纸张上画出示意图才能更加清晰地理解每一条语句在干什么。本次实验的任务是利用哈希映射来完成不同不同数据块到不同的桶的映射，这里为了简单，直接只用了取余构造哈希函数，同时利用了素数个桶保证尽可能地减少哈希冲突；最后这里实际上使用了开散列的方式，也就是说每一个桶可以存储多个元素，以链表相互连接，这样的方法简单易实现，并且确实可以有效处理大量的哈希冲突；但它有可能导致较长的搜索时间，特别是当链表长度变得很长的时候。并且，这里我们根据程序的局部性原理，使用 LRU 算法，选择最近最少访问的块进行替换。这里很好地为我展现了在计算机组成原理课程上的理论的实现，更加加深了理解。

推送至仓库

```

score: 10/10
lq@vm:~/xv6-labs-2021$ git add .
lq@vm:~/xv6-labs-2021$ git commit -m "2025-08-10:lab8"
[lock 46b69cd] 2025-08-10:lab8
 4 files changed, 161 insertions(+), 46 deletions(-)
 create mode 100644 time.txt
lq@vm:~/xv6-labs-2021$ git push github lock:lock
Enumerating objects: 72, done.
Counting objects: 100% (72/72), done.
Delta compression using up to 4 threads
Compressing objects: 100% (35/35), done.
Writing objects: 100% (55/55), 18.55 KiB | 4.64 MiB/s, done.
Total 55 (delta 36), reused 32 (delta 17), pack-reused 0
remote: Resolving deltas: 100% (36/36), completed with 16 local objects.
remote:
remote: Create a pull request for 'lock' on GitHub by visiting:
remote:     https://github.com/lq1911/OS/pull/new/lock
remote:
To https://github.com/lq1911/OS.git
 * [new branch]      lock -> lock

```

Lab 9 : file system

切换分支

```

$ git fetch

$ git checkout fs

$ make clean

```

1. Large files

1.1. 实验目的

本次实验的目的是扩展 xv6 文件系统，使其支持更大的文件大小。当前，xv6 的文件大小限制为 268 个块，或 $268 * \text{BSIZE}$ 字节（在 xv6 中，`BSIZE` 为 1024）。这一限制源于 xv6 中的 `inode` 结构，其包含 12 个“直接”块号和一个“单间接”块号，后者引用一个可以容纳多达 256 个块号的数据块，因此总共可以引用的块数为 $12+256=268$ 个。为了增加 xv6 文件的最大大小，本实验将通过将其中一个直接数据块号替换为一个两层间接数据块号，该数据块号指向一个包含间接数据块号的数据块，从而扩展文件系统的存储能力。

1.2. 实验步骤

- xv6 系统的 `inode` 原先有 12 个直接索引，1 个一级索引，我们将一个直接做因修改为二级索引，即 11 个直接索引、1 个一级索引和 1 个二级索引。总共可以指向 $11 + 256 + 256^2 = 65803$ 个物理块。修改 `kernel/fs.h` 文件中的直接索引块数为 11：

```

// #define NDIRECT 12
// lab9-1
#define NDIRECT 11

```

- `NDIRECT` 变少了但其实总量不变，仍是 13，因此需要修改 `kernel/fs.h` 文件中的 `struct dinode` 的 `addrs` 数据成员：

```

struct dinode {
    ...
    uint addrs[NDIRECT+2]; // Data block addresses
};

```

- 同时也要修改 `kernel/file.h` 文件中的 `struct inode` 的 `addrs` 数据成员：

```

struct inode {
    ...
    uint size;
    // lab9-1
    uint addrs[NDIRECT+2];
};

```

- 在 `kernel/fs.h` 文件中定义二级索引总数，并修改文件最大体积：

```

// 二级索引能找到的物理块数量lab9-1
#define NSECDIRECT (NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NSECDIRECT)

```

- 参考 `kernel/fs.c` 的 `bmap()` 函数进一步添加对于二级索引的处理：

```

static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a;
    struct buf *bp;

    if(bn < NDIRECT){
        if((addr = ip->addrs[bn]) == 0)
            ip->addrs[bn] = addr = balloc(ip->dev);
        return addr;
    }
    bn -= NDIRECT;

    if(bn < NINDIRECT){
        // Load indirect block, allocating if necessary.
        if((addr = ip->addrs[NDIRECT]) == 0)
            ip->addrs[NDIRECT] = addr = balloc(ip->dev);
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
        if((addr = a[bn]) == 0){
            a[bn] = addr = balloc(ip->dev);
            log_write(bp);
        }
        brelse(bp);
        return addr;
    }

    // 到二级索引 lab9-1
    bn -= NINDIRECT;
    if (bn < NSECDIRECT) {
        if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
            ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
        }
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;

        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;
        if ((addr = a[bn]) == 0) {
            a[bn] = addr = balloc(ip->dev);
            log_write(bp);
        }
        brelse(bp);
        return addr;
    }

    panic("bmap: out of range");
}

```

- 参考 kernel/fs.c 的 itrunc() 函数进一步添加对于二级索引的处理：

```

void
itrunc(struct inode *ip)
{
    // k, bp1, a1用于二级索引 lab9-1
    int i, j, k;
    struct buf *bp, *bp1;
    uint *a, *a1;

    for(i = 0; i < NDIRECT; i++){
        if(ip->addrs[i]){
            bfree(ip->dev, ip->addrs[i]);
            ip->addrs[i] = 0;
        }
    }

    if(ip->addrs[NDIRECT]){
        bp = bread(ip->dev, ip->addrs[NDIRECT]);
        a = (uint*)bp->data;
        for(j = 0; j < NINDIRECT; j++){
            if(a[j])
                bfree(ip->dev, a[j]);
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT]);
        ip->addrs[NDIRECT] = 0;
    }

    // 释放二级索引的物理块
    if (ip->addrs[NDIRECT + 1]) {
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint*)bp->data;
        // 遍历二级索引
        for (j = 0; j < NINDIRECT; ++j) {
            if (a[j]) {
                bp1 = bread(ip->dev, a[j]);
                a1 = (uint*)bp1->data;
                for (k = 0; k < NINDIRECT; ++k) {
                    if (a1[k]) {
                        bfree(ip->dev, a1[k]);
                    }
                }
                brelse(bp1);
                bfree(ip->dev, a[j]);
                a[j] = 0;
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 1]);
        ip->addrs[NDIRECT + 1] = 0;;
    }

    ip->size = 0;
    iupdate(ip);
}

```

- 编译并测试程序

```

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

```

```
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3241
    sepc=0x00000000000569e stval=0x0000000000000569e
usertrap(): unexpected scause 0x000000000000000c pid=3242
    sepc=0x00000000000569e stval=0x0000000000000569e
OK
```

```
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitput: OK
test iput: OK
test mem: OK
test pipe1: 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
```

1.3. 实验小结

- 问题及解决方案

- 在修改 bmap() 函数时，由于需要处理二级间接块，索引的复杂性显著增加。一级间接块的处理相对简单，因为只需要一次索引即可获取目标数据块的地址。然而，二级间接块涉及两次索引：第一次索引获取一级间接块的地址，第二次索引获取实际数据块的地址。这要求在代码实现中准确管理这两个层次的索引关系，否则可能导致数据块访问错误或系统崩溃。
通过仔细设计 bmap() 函数的逻辑，确保在处理二级间接块时正确地获取和使用每个索引值。同时，在调试阶段，增加了对索引值的检查和验证，确保函数在不同情况下都能正确返回数据块的地址。
- 在 itrunc() 函数的实现中，由于需要释放的块数增加，释放过程变得更加复杂。除了释放直接块和一级间接块外，还需要遍历并释放二级间接块及其引用的一级间接块。这一过程中，任何遗漏或错误都可能导致内存泄漏或其他潜在问题。
通过增加多层次的循环来遍历和释放所有相关块，并严格管理每一层次的释放操作，确保所有分配的块在不再需要时都能被正确释放。同时，加入了更多的错误检查机制，以捕捉并处理可能出现的异常情况。

- 心得

通过此次实验，深入理解了文件系统在处理大文件时的结构扩展问题，特别是在有限的 inode 空间内如何高效地管理大量数据块。此次实验的核心在于如何利用二级间接块来突破文件系统原有的限制，从而支持更大的文件存储。

2. Symbolic links

2.1. 实验目的

本次实验的主要目标是在 xv6 操作系统中实现符号链接（软链接）功能。符号链接是一种通过路径名引用另一个文件的方式，与硬链接不同，它可以跨越不同的磁盘设备。通过实现这一系统调用，将深入理解路径名查找的工作原理。

2.2. 实验步骤

- 添加 symlink 系统调用，在 user/user.h 文件中声明需要添加的系统调用 symlink：

```
// lab9-2
int symlink(char* target, char* path);
```

- 在 user/usys.pl 文件中添加上述系统调用的入口：

```
# lab9-2
...
entry("uptime");
entry("symlink");
```

- 在 kernel/syscall.h 添加定义：

```
// lab9-2
#define SYS_close 21
#define SYS_symlink 22
```

- 在 kernel/syscall.c 添加函数声明:

```
// lab9-2
extern uint64 sys_uptime(void);
extern uint64 sys_symlink(void);
...
static uint64 (*syscalls[])(void) = {
...
// lab9-2
[SYS_close] sys_close,
[SYS_symlink] sys_symlink,
};
```

- 为了实现软链接, 我们需要声明新的一种文件类型 在 kernel/stat.h 文件中添加:

```
...
#define T_DEVICE 3 // Device
// 软链接文件类型 lab9-2
#define T_SYMLINK 4
...
```

- 在 kernel/fcntl.h 文件中添加新的文件标志位 O_NOFOLLOW:

```
#define O_NOFOLLOW 0x004
```

- 实现 sys_symlink() 函数, 用于生成符号链接, 新建一个 inode 写入相关数据。在 kernel/sysfile.c 中实现 sys_symlink() 函数:

```
uint64 sys_symlink(void) {
    char target[MAXPATH], path[MAXPATH];
    struct inode* ip;

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

    begin_op();
    // 创建符号链接
    if ((ip = create(path, T_SYMLINK, 0, 0) == 0)) {
        end_op();
        return -1;
    }
    // 写入目标路径
    if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
        iunlockput(ip);
        end_op();
        return -1;
    }
    iunlockput(ip);
    end_op();
    return 0;
}
```

- 修改 kernel/sysfile.c 文件中的 sys_open() 函数:

```

uint64
sys_open(void)
{
    ...
    if(omode & O_CREATE){
        ip = create(path, T_FILE, 0, 0);
        if(ip == 0){
            end_op();
            return -1;
        }
    } else {
        // lab9-2
        const int max_depth = 20;
        int depth = 0;
        while (1) {
            if((ip = namei(path)) == 0){
                end_op();
                return -1;
            }
            ilock(ip);
            if (ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
                // 超出最深层数
                if (++depth > max_depth) {
                    iunlockput(ip);
                    end_op();
                    return -1;
                }
                if (readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH) {
                    iunlockput(ip);
                    end_op();
                    return -1;
                }
                iunlockput(ip);
            }
            else {
                break;
            }
        }
        if(ip->type == T_DIR && omode != O_RDONLY){
            iunlockput(ip);
            end_op();
            return -1;
        }

        if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
            iunlockput(ip);
            end_op();
            return -1;
        }
        ...
    }
}

```

- 编译并测试程序

```

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

```

```
test concurrent synthesis: OK
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3242
    sepc=0x00000000000569e stval=0x00000000000569e
usertrap(): unexpected scause 0x000000000000000c pid=3243
    sepc=0x00000000000569e stval=0x00000000000569e
ex
```

```
        sepc=0x0000000000041fa stval=0x000000000012000
OK
test sbrkarg: OK
test validateitest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6220
    sepc=0x0000000000022ce stval=0x00000000000fb90
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitinput: OK
test input: OK
test mem: OK
test pipel: OK
test preempt: kill... wait... OK
test exitwait: OK
test rndot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

2.3. 实验小结

- 问题与解决方法

修改 `sys_open()` 函数错误导致一直无法启动 xv6 系统，因此我们多找几个网上的参考 `sys_open()` 函数中原来的 `ilock(ip)` 忘记删除，导致多次 `ilock()` 函数的调用，删掉就好了。

- 实验心得

这里软链接的实现利用了 xv6 系统中的 `begin_op()` 和 `end_op()` 函数，他们可以实现文件系统的原子操作，主要用于确保在对文件系统进行修改的时候，不会发生竞争条件或数据不一致的情况，从而保证了原子性和一致性。有点类似数据库系统中的事务的处理方式而对于 `sys_open()` 函数，其中对于软链接的处理方法如下 该函数主要用于根据给定的路径字符串 `path` 找到对应的文件节点。它通过循环和条件判断来处理可能存在的符号链接，并限制了最大打开的深度。

首先，我们定义了最大深度 `max_depth`，用于打开的层数，从而避免过度消耗系统的资源。然后，进入一个无限循环，直到找到目标文件节点或发生错误才会退出循环。在循环中，首先调用 `namei(path)` 函数来查找路径对应的文件节点。如果未找到文件节点，则执行清理操作并返回 -1 表示失败。如果找到了文件节点，代码会对该节点进行加锁，以确保在访问期间不会被其他进程修改。然后，检查该节点的类型是否为符号链接，并检查打开模式是否允许跟随符号链接。如果需要处理符号链接，代码会检查当前深度是否超过了最大深度 `max_depth`。如果超过最大深度，则执行清理操作并返回 -1 表示失败。如果深度未超过最大深度，代码会读取符号链接的内容，并将结果存储在 `path` 变量中。如果读取的内容长度小于 `MAXPATH`，表示读取失败，同样执行清理操作并返回 -1 表示失败。最后，代码会解锁并释放文件节点。如果文件节点不是符号链接或者不需要处理符号链接，则跳出循环。

推送至仓库

```
lq@vm:~/xv6-labs-2020$ git add .
[lq@vm:~/xv6-labs-2020] 2025-08-10:lab9
[15 2ce3f87] 2025-08-10:lab9
12 files changed, 128 insertions(+), 18 deletions(-)
..._eate mode 106644 time.txt
mode change 106755 => 106644 user/sys.pl
lq@vm:~/xv6-labs-2020$ git push github fs:st
Enumerating objects: 112, done.
Counting objects: 100% (92/92), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (46/46), done.
Writing objects: 100% (66/66), 17.51 KiB | 4.38 MiB/s, done.
Total 66 (delta 47), reused 32 (delta 17), pack-reused 0
remote: Resolving deltas: 100% (47/47), completed with 23 local objects.
remote:
remote: Create a pull request for 'fs' on GitHub by visiting:
remote:   https://github.com/lq1911/OS/pull/new/fs
remote:
To https://github.com/lq1911/OS.git
 [new branch]   fs -> fs
```

Lab 10 : mmap

切换分支

```
$ git fetch
$ git checkout mmap
$ make clean
```

1. mmap

1.1. 实验目的

本次实验的目标是为 xv6 操作系统添加 mmap 和 munmap 系统调用，以实现对进程地址空间的精细控制。通过这两个系统调用，可以实现内存映射文件的功能，包括共享内存和将文件映射到进程的地址空间等。这对于理解虚拟内存管理和页面错误处理机制具有重要意义。

1.2. 实验步骤

- 添加 mmap 和 munmap 两个系统调用，在 user/user.h 文件中声明需要添加的系统调用 mmap 和 munmap：

```
int uptime(void);
// lab10
void* mmap(void *, int, int, int, uint);
int munmap(void *, int);
```

- 在 user/usys.pl 文件中添加上述系统调用的入口：

```
# lab10
...
entry("uptime");
entry("mmap");
entry("munmap");
```

- 在 kernel/syscall.h 添加定义：

```
#define SYS_close 21
// lab10
#define SYS_mmap 22
#define SYS_munmap 23
```

- 在 kernel/syscall.c 添加函数声明：

```
// lab10
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
...
static uint64 (*syscalls[])(void) = {
...
// lab10
[SYS_close] sys_close,
[SYS_mmap] sys_mmap,
[SYS_munmap] sys_munmap,
};
```

- 在 Makefile 文件中添加对 mmaptest 的编译：

```
...
$U/_zombie\
$U/_mmaptest\
```

- 在 kernel/proc.h 文件中结构体 struct proc 之前定义虚拟内存区域结构体 vma，一个进程具有多少 vma，并且在 struct proc 结构体中添加数据成员 vma 数组：

```
// 定义虚拟内存区域结构体lab10
#define VMASIZE 16

struct vma {
    struct file *file; // 文件结构体指针
    uint64 addr;      // mmap映射起始地址
    int len;          // mmap映射内存大小
    int prot;         // 用户权限
    int flags;        // mmap标志位
    int offset;       // 文件偏移量
    int fd;           // 文件描述符
    int used;         // 是否使用过
};

// Per-process state
struct proc {
    ...
    char name[16];           // Process name (debugging)
    // 一个进程对应一个vma数组 lab10
    struct vma vma[VMASIZE];
};

```

- 在kernel/sysfile.c文件夹中实现上述两个系统调用：

```

uint64 sys_mmap(void) {
    uint64 addr;
    int len, prot, flags, fd, offset;
    struct proc *p = myproc();
    struct file *file;
    if(argaddr(0, &addr) || argint(1, &len) || argint(2, &prot) ||
       argint(3, &flags) || argfd(4, &fd, &file) || argint(5, &offset))
        return -1;

    if(!file->writable && (prot & PROT_WRITE) && flags == MAP_SHARED)
        return -1;

    len = PGROUNDUP(len);
    if(p->sz > MAXVA - len)
        return -1;

    for(int i = 0; i < VMASIZE; i++) {
        if(p->vma[i].used == 0) {
            p->vma[i].used = 1;
            p->vma[i].addr = p->sz;
            p->vma[i].len = len;
            p->vma[i].prot = prot;
            p->vma[i].flags = flags;
            p->vma[i].fd = fd;
            p->vma[i].file = file;
            p->vma[i].offset = offset;
            filedup(file);
            p->sz += len;
            return p->vma[i].addr;
        }
    }
    return -1;
}

uint64 sys_munmap(void) {
    uint64 addr;
    int len;
    struct proc *p = myproc();
    struct vma *vma = 0;
    if(argaddr(0, &addr) || argint(1, &len))
        return -1;
    addr = PGROUNDDOWN(addr);
    len = PGROUNDUP(len);
    for(int i = 0; i < VMASIZE; i++) {
        if (addr >= p->vma[i].addr || addr < p->vma[i].addr + p->vma[i].len) {
            vma = &p->vma[i];
            break;
        }
    }
    if(vma == 0)
        return 0;
    if(vma->addr == addr) {
        vma->addr += len;
        vma->len -= len;
        if(vma->flags & MAP_SHARED)
            filewrite(vma->file, addr, len);
        uvmunmap(p->pagetable, addr, len/PGSIZE, 1);
        if(vma->len == 0) {
            fclose(vma->file);
            vma->used = 0;
        }
    }
}

```

```
    return 0;
```

```
}
```

- 修改 kernel/trap.c 文件中的 usertrap() 函数，增加对 page fault 的判断：

```
void
usertrap(void)
{
    ...
    if(r_scause() == 8){
        ...
    }
    // 处理page fault lab10
    else if (r_scause() == 13 || r_scause() == 15) {
        uint64 va = r_stval();
        if(va >= p->sz || va > MAXVA || PGROUNDDOWN(va) == PGROUNDDOWN(p->trapframe->sp))
            p->killed = 1;
        else {
            struct vma *vma = 0;
            for (int i = 0; i < VMASIZE; i++) {
                if (p->vma[i].used == 1 && va >= p->vma[i].addr &&
                    va < p->vma[i].addr + p->vma[i].len) {
                    vma = &p->vma[i];
                    break;
                }
            }
            if(vma) {
                va = PGROUNDDOWN(va);
                uint64 offset = va - vma->addr;
                uint64 mem = (uint64)kalloc();
                if(mem == 0) {
                    p->killed = 1;
                }
                else {
                    memset((void*)mem, 0, PGSIZE);
                    ilock(vma->file->ip);
                    readi(vma->file->ip, 0, mem, offset, PGSIZE);
                    iunlock(vma->file->ip);
                    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;
                    if(mappages(p->pagetable, va, PGSIZE, mem, flag) != 0) {
                        kfree((void*)mem);
                        p->killed = 1;
                    }
                }
            }
        }
    }
    else if((which_dev = devintr()) != 0){
        // ok
    }
    ...
}
```

- 在 kernel/vm.c 文件中修改 uvmunmap() 函数和 uvmcopy() 函数，取消不有效就 panic 的操作：

```

void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
    uint64 a;
    pte_t *pte;

    if((va % PGSIZE) != 0)
        panic("uvmunmap: not aligned");

    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
        if((pte = walk(pagetable, a, 0)) == 0)
            panic("uvmunmap: walk");
        if((*pte & PTE_V) == 0)
            // panic("uvmunmap: not mapped");
        continue;
        if(PTE_FLAGS(*pte) == PTE_V)
            // panic("uvmunmap: not a leaf");
        continue;
        if(do_free){
            uint64 pa = PTE2PA(*pte);
            kfree((void*)pa);
        }
        *pte = 0;
    }
}

int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    ...
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walk(old, i, 0)) == 0)
            panic("uvmcopy: pte should exist");
        if((*pte & PTE_V) == 0)
            // panic("uvmcopy: page not present");
        continue;
        pa = PTE2PA(*pte);
        ...
    }
    ...
}

```

- 修改 kernel/proc.c 文件中的 exit() 函数和 fork() 函数：

```

int
fork(void)
{
    ...
    np->state = RUNNABLE;

    // lab10
    for(int i = 0; i < VMASIZE; i++) {
        if(p->vma[i].used){
            memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));
            filedup(p->vma[i].file);
        }
    }

    release(&np->lock);

    return pid;
}

void
exit(int status)
{
    ...
    // Close all open files.
    for(int fd = 0; fd < NOFILE; fd++){
        if(p->ofile[fd]){
            struct file *f = p->ofile[fd];
            fileclose(f);
            p->ofile[fd] = 0;
        }
    }

    for(int i = 0; i < VMASIZE; i++) {
        if(p->vma[i].used) {
            if(p->vma[i].flags & MAP_SHARED)
                filewrite(p->vma[i].file, p->vma[i].addr, p->vma[i].len);
            fileclose(p->vma[i].file);
            uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].len / PGSIZE, 1);
            p->vma[i].used = 0;
        }
    }

    begin_op();
    iput(p->cwd);
    end_op();
    p->cwd = 0;

    ...
}

```

- 编译并测试程

```

xv6 kernel is booting
hart 1 starting
hart 2 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

```

```

mmaptest: all tests succeeded
$ usertest
usertest starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3242
           sepcc=0x000000000000569c stval=0x000000000000569c
usertrap(): unexpected scause 0x000000000000000c pid=3243
           sepcc=0x000000000000569c stval=0x000000000000569c
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: 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

```

1.3. 实验小结

- 问题及解决
 - 虚拟内存区域管理：在实现 mmap 系统调用时，需要确保在进程的虚拟地址空间中合理分配一个连续的虚拟内存区域，并记录该区域的信息。为了管理这些虚拟内存区域，我们引入了 VMA 结构体，并在 proc 结构体中添加了 vma_pool 字段。最初在设计 VMA 池的分配和释放机制时，如何高效地管理这些 VMA 条目成为一个难点。解决方案是在 vma_pool 中维护一个固定大小的数组，并通过标记条目的使用状态来分配和释放 VMA，这使得管理逻辑更加简单高效。
 - 页面错误处理与物理内存分配：实现 mmap 系统调用时，我们采用了延迟分配（Lazy Allocation）的策略，这意味着在调用 mmap 时只分配虚拟地址空间，并不立即分配物理内存。物理内存的分配是在发生页面错误时进行的。因此，在 usertrap 中添加页面错误处理逻辑变得至关重要。在实现过程中，最初的页面错误处理逻辑没有正确分配物理页面，导致进程访问 mmap 内存时出现段错误。通过调试和分析，我们确定了问题出在页面表的更新和物理内存分配的时机上。最终，通过修改 vm.c 中的 uvmcopy 和 uvmunmap 函数，确保在发生页面错误时正确分配物理内存并更新页面表，问题得以解决。
- 实验心得：通过本次实验，我们深入理解了操作系统中的虚拟内存管理机制，特别是 mmap 和 munmap 系统调用在实际应用中的重要性。实验中，我们不仅实现了基本的内存映射功能，还处理了页面错误、内存延迟分配、进程退出时的资源管理等复杂场景。这个过程加深了我们对虚拟内存、页面表、进程地址空间等概念的理解。

推送至仓库

```

lq@vm:~/xuc-labs-2023$ git add .
lq@vm:~/xuc-labs-2023$ git commit -m "2025-08-10:lab10"
[mmap bcf722a] 2025-08-10:lab10
11 files changed, 176 insertions(+), 6 deletions(-)
create mode 100644 time.txt
mode change 100755 => 100644 user/usys.pl
lq@vm:~/xuc-labs-2023$ git push github mmap:mmap
Enumerating objects: 42, done.
Counting objects: 100% (42/42), done.
Delta compression using up to 4 threads
Compressing objects: 100% (24/24), done.
Writing objects: 100% (27/27), 14.19 KiB | 3.55 MiB/s, done.
Total 27 (delta 17), reused 6 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (17/17), completed with 13 local objects.
remote:
remote: Create a pull request for 'mmap' on GitHub by visiting:
remote:   https://github.com/lq1911/OS/pull/new/mmap
remote:
To https://github.com/lq1911/OS.git
 * [new branch]      mmap -> mmap

```

Lab 11 : networking

切换分支

```

$ git fetch
$ git checkout net
$ make clean

```

1. networking

1.1. 实验目的

在 kernel/e1000.c 中完成 e1000_transmit 和 e1000_recv()，以便驱动程序可以发送和接收数据包。

1.2. 实验步骤

- 在 kernel/e1000.c 实现 e1000_transmit() 和 e1000_recv() 两个函数

```
int
e1000_transmit(struct mbuf *m)
{
    //
    // Your code here.
    //

    // the mbuf contains an ethernet frame; program it into
    // the TX descriptor ring so that the e1000 sends it. Stash
    // a pointer so that it can be freed after sending.
    //

    acquire(&e1000_lock);
    uint32 next_index = regs[E1000_TDT];
    if((tx_ring[next_index].status & E1000_TXD_STAT_DD) == 0){
        release(&e1000_lock);
        return -1;
    }
    if(tx_mbufs[next_index])
        mbuffree(tx_mbufs[next_index]);
    tx_ring[next_index].addr = (uint64)m->head;
    tx_ring[next_index].length = (uint16)m->len;
    tx_ring[next_index].cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;
    tx_mbufs[next_index] = m;
    regs[E1000_TDT] = (next_index+1)%TX_RING_SIZE;
    release(&e1000_lock);
    return 0;
}

static void
e1000_recv(void)
{
    //
    // Your code here.
    //

    // Check for packets that have arrived from the e1000
    // Create and deliver an mbuf for each packet (using net_rx()).
    //

    uint32 next_index = (regs[E1000_RDT]+1)%RX_RING_SIZE;
    while(rx_ring[next_index].status & E1000_RXD_STAT_DD){
        if(rx_ring[next_index].length>MBUF_SIZE){
            panic("MBUF_SIZE OVERFLOW!");
        }
        rx_mbufs[next_index]->len = rx_ring[next_index].length;
        net_rx(rx_mbufs[next_index]);
        rx_mbufs[next_index] = mbufalloc(0);
        rx_ring[next_index].addr = (uint64)rx_mbufs[next_index]->head;
        rx_ring[next_index].status = 0;
        next_index = (next_index+1)%RX_RING_SIZE;
    }
    regs[E1000_RDT] = (next_index-1)%RX_RING_SIZE;
}
```

- 测试程序

在一个窗口中运行 `make server`，在另一个窗口中运行 `make qemu`，然后在 xv6 中运行 `nettests`

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ nettests
nettests running on port 26099
testing ping: OK
testing single-process pings: OK
testing multi-process pings: OK
testing DNS
DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
DNS OK
all tests passed.
$
```

```
lq@vm: ./xv6-labs-2020$ make server
python3 server.py 26099
listening on localhost port 26099
a message from xv6!
```

- 键入 **Ctrl+c** 服务端终止运行，键入 **tcpdump -XXnr packets.pcap** 查看传输数据内容传输大量数据

```
0x0030: 7320 7468 6520 686f 7374 21 s.the.host!
09:36:17.007193 IP 10.0.2.2,26099 > 10.0.2.15,2003: UDP, length 17
0x0000: 5254 0012 3456 5255 0a00 0202 0800 4500 RT..4VRU.....E.
0x0010: 0020 006d 0000 4011 6243 0a00 0202 0a00 ..n..@bc.....
0x0020: 020f 65f3 07d3 0019 3213 7468 6973 2069 ..e....2.this.i
0x0030: 7320 7468 6520 686f 7374 21 s.the.host!
09:36:17.007199 IP 10.0.2.2,26099 > 10.0.2.15,2001: UDP, length 17
0x0000: 5254 0012 3456 5255 0a00 0202 0800 4500 RT..4VRU.....E.
0x0010: 0020 006e 0000 4011 6242 0a00 0202 0a00 ..n..@b6.....
0x0020: 020f 65f3 07d1 0019 3215 7468 6973 2069 ..e....2.this.i
0x0030: 7320 7468 6520 686f 7374 21 s.the.host!
09:36:17.039242 IP 10.0.2.15,10000 > 8.8.8.53: 6828+ A? pdos.csail.mit.edu. (36)
0x0000: ffff ffff ffff 5254 0012 3456 0800 4500 .....RT..4V.E.
0x0010: 0040 0000 0000 6411 3a8f 0a00 020f 0888 .0....d:.....
0x0020: 0888 2710 0035 002c 0000 1aac 0100 0001 ..'.5.....'.
0x0030: 0008 0000 0000 0470 646f 7305 6373 6169 .....pdos.csai
0x0040: 6c03 6d69 7403 6564 7500 0001 0001 l.mit.edu....
09:36:17.346542 IP 8.8.8.53 > 10.0.2.15,10000: 6828 1/0/0 A 128.52.129.126 (52)
0x0000: 5254 0012 3456 5255 0a00 0202 0800 4500 RT..4VRU.....E.
0x0010: 0050 006f 0000 4011 5e10 0808 0808 0a00 .P.o..@.^.....
0x0020: 020f 0035 2710 003c 8e7d 1aac 8180 0001 ..5'..s.).....
0x0030: 0001 0000 0000 0479 646f 7305 6373 6169 .....pdos.csai
0x0040: 6c03 6d69 7403 6564 7500 0001 c00c l.mit.edu....
0x0050: 0001 0001 0000 0708 0004 8034 817e .....4.-.
```

- 单元测试

回到开启 xv6 系统的终端，退出 xv6 进行单元测试：

```
lq@vm: ./xv6-labs-2020$ ./grade-lab-net
make: 'kernel/kernel' is up to date.
== Test running nettests == (3.8s)
== 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
```

1.3. 实验小结

遇到不了解的方面，我们要先去看看相关的概念才好入手，否则就算实验通过了也不知道自己所写的每一行代码是用来做什么的，在其中发挥什么样的功能。实际上本实验就是实现一个简陋的网卡驱动

- 发送函数

我们首先加锁，保护发送数据的时候不会有其他进程来干扰，然后从寄存器中获取下一个可用的传输描述符的索引，通过读取 E1000_TDT，然后检查 ring 是否溢出。如果 E1000_TXD_STAT_DD 没有在 E1000_TDT 索引的描述符中被设置，则 E1000 还没有结束之前的传输请求，返回 error。

若已经设置，使用 mbuffree() 函数释放最后的（索引到的描述符指向的）已经被发送（但还没有释放）的 mbuf（如果使用过的话）然后设置 mbuf 的传输数据的地址以及数据大小，已经传输描述符的命令字段。

这里需要查询以下：

- E1000_TXD_CMD_EOP 表示该数据包是帧的结束
- E1000_TXD_CMD_RS 表示发送后释放该描述符

然后将 m 存储到 tx_mbuf 数组中，以便在发送后可以释放。

最后更新寄存器数组中下一个可用描述符的索引。

- 接收函数

首先从寄存器中获取下一个可用的接收描述符的索引。

利用 while 循环和 E1000_RXD_STAT_DD 判断是否有心得数据包准备好接收。

然后将接收到的数据包的长度存储在对应的 mbuf 中。

将接收到的数据包传输给网络层处理，用来实现解析数据包的内容等操作。

接着给下一个接受描述符分配一个新的 mbuf，以便接收下一个数据包。

将新的mbuf的头部地址存储到接收描述符中，以便驱动程序可以将数据写入该地址。

将接收描述符的状态字段清零，表示已经处理该数据包。

更新下一个可用接收描述符的索引。

更新寄存器中的接收描述符表的尾部索引。

- 以太网

实验中提到的以太网是一种常见的局域网技术，用于在计算机和设备之间传输数据。它是最常用的有线局域网技术之一，广泛应用于家庭、办公室和企业网络中。

以太网使用双绞线或光纤等物理媒介来传输数据。它基于一组标准化的通信协议和规范，其中最常见的是IEEE 802.3系列标准。这些标准定义了数据传输的格式、速率、错误检测和纠正等方面的规定，确保不同设备之间可以相互通信和交换数据。

以太网的工作原理是通过发送和接收数据帧来实现。数据帧是数据传输的基本单位，包括源和目的地址、数据内容以及校验和等信息。计算机或设备通过网络交换机或集线器等网络设备连接到以太网，它们可以通过发送数据帧来与其他设备进行通信。

- 数据包

数据包是在计算机网络中传输的基本单位。它是将数据按照特定的格式进行封装和分割后的一段信息。数据包通常包含了源地址、目的地址、控制信息、有效载荷等部分。

在计算机网络中，数据通常被分割成较小的数据包进行传输。这样做的好处是可以提高网络的效率和可靠性。较大的数据可以被分割成多个数据包，每个数据包独立传输，到达目的地后再重新组装成完整的数据。

数据包的封装和分割过程是在网络协议栈中的传输层和网络层完成的。在传输层，数据被封装成传输层协议的数据段或用户数据报。在网络层，数据段或用户数据报再被封装成网络层协议的数据包。

数据包的封装和解封装过程中，会在数据包的头部添加一些控制信息，如源地址和目的地址，以便网络设备能够正确地将数据包传输到目的地。数据包还可以包含一些错误检测和纠正的校验和，以确保数据的完整性和准确性。

在网络中，数据包通过网络设备（如路由器、交换机）在不同的网络节点之间进行转发和交换，最终到达目的地。在目的地，数据包被解封装，提取出有效载荷中的数据，并进行相应的处理和应用。