同濟大學
TONGJI UNIVERSITY

操作系统

课程设计

姓名：韩明洋

学号：2253206

专业：软件学院 软件工程

指导老师：王冬青

序言

计算机科学作为一门不断进步和发展的领域，推动了现代社会的方方面面。操作系统作为计算机系统的核心组成部分，在保障计算机硬件和软件协同运作方面发挥着重要作用。本报告旨在介绍我在 MIT 6.S081 课程中进行的实验，这些实验深入探讨了操作系统的内部工作原理，旨在为我打下坚实的计算机科学基础。

MIT 6.S081 课程为我们提供了深入理解操作系统的机会，通过实际动手操作，我们能够更好地理解计算机系统的底层原理。本实验报告将详细介绍我在课程中完成的实验内容，以及在这个过程中遇到的挑战和收获。

在本实验报告中，我将分别介绍课程中涵盖的各个实验项目，并针对实验环境的搭建、实验心得等进行详细的讲解。在接下来的报告中，我将详细描述每个实验的过程、原理和结果。我希望通过本报告能够传达出我在 MIT 6.S081 课程和实验中的学习心得和成果，也希望能够激发更多人对操作系统和计算机科学的兴趣。

目录

一、xv6 实验介绍

1. xv6 实验简介

xv6 是一个简单而高效的操作系统，由麻省理工学院（MIT）开发，旨在用于教学和研究目的。xv6 基于 Unix 操作系统的早期版本，特别是 V6 和 V7，以及少量的 Plan9 操作系统的思想。它提供了一个良好的平台，供学生深入了解操作系统的内部工作原理，同时也为研究人员提供一个轻量级的系统用于实验和探索。xv62021 实验是 MIT6.S081 课程中的一部分，每个实验都涉及编写代码、进行调试和测试，以及撰写相应的实验报告。通过这些实验，学生将深入了解操作系统的内部工作原理，培养解决问题和系统设计的能力。

2. 实验项目

笔者在本次操作系统课程设计中选择的 xv6 及 Labs 课程项目，版本为 2021 年版本，大致分为以下十个实验：

1. 实用工具实验：Xv6 and Unix utilities
2. 系统调用实验：Lab: system calls
3. 页表实验：Lab: page tables
4. 中断实验：Lab: traps
5. 写时复制实验：Lab: copy on-write
6. 多线程实验：Lab: multithreading
7. 网卡驱动实验：Lab: network driver
8. 锁实验：Lab: lock
9. 文件系统实验：Lab: file system
10. 内存映射实验：Lab: mmap

3.实验环境

笔者选择在虚拟机(VMware Workstation pro16)运行 Ubuntu 20.04，使用 qemu 来模拟 RISC-V 进行实验。

4.实验目的

在十次实验中，学习如何利用操作系统提供的自带功能与工具、学习操作系统的各种操作、改进操作系统中的各种功能，同时在学习并理解操作系统的过程中，了解操作系统的机制，对操作系统有更深刻的理解与认识。

5.实验要求

完成十个实验内容，包括引导程序、核心代码，文件系统，控制台等，可以借鉴其他操作系统的设计思想，至少有一半代码量由项目组完成。

6. 代码仓库链接

项目已经开源到：
https://github.com/summerapril5/os-xv6.git

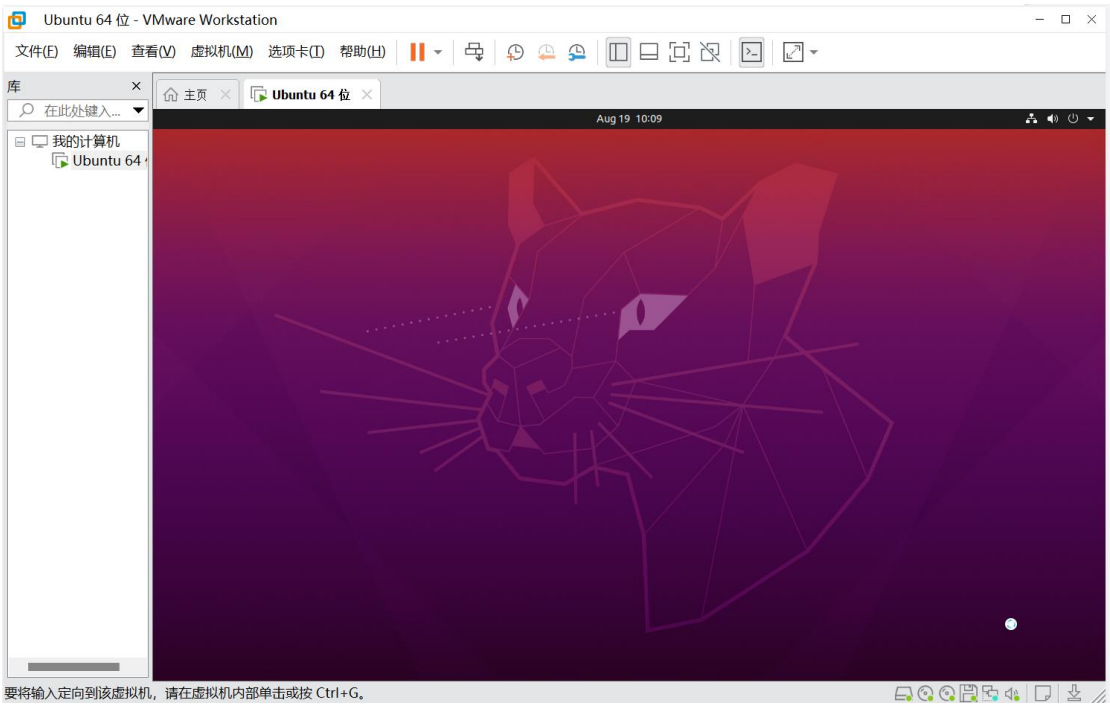
二、实验前期准备

1.虚拟机的安装

笔者选择 VMware Workstation pro16 作为虚拟机平台，使用默认配置进行安装，重启计算机。

重启后在 VMWare 中新建虚拟机，选择下载的 Ubuntu 对应的 iso，完成虚拟机的创建。进行用户名、密码的输入：(用户名：os001，密码：123456)。

成功安装后结果如下图所示：



2. Ubuntu 环境配置

3.1 Ubuntu 软件源的配置

Ubuntu 的安装完成后，需要对其进行一系列的配置。首先，先对软件源进行配置，以加快后续工具链的安装速度。操作如下：

Ctrl+alt+t 打开终端

1. 备份原来的源：

```
sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak
```

2.进行 vim 的安装和配置：

```
sudo apt-get install vim vim /etc/vim/vimrc
```

在 vim 配置文件中添加以下代码：

```
set ai                    set aw                    set flash                 set ic                    set nu                    set number                set showmatch             set showmode              set showcmd               set warn                  set ws                    set wrap                  colorscheme evening       filetype plugin on        set autoindent            set cindent               set noignorecase          set ruler                 set scrolloff=5           set
```

3 打开文件后修改：

```
sudo vim /etc/apt/sources.list
```

在此文件后添加内容：

```
deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal main restricted universe multiverse
deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-updates main restricted universe multiverse
deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-backports main restricted universe multiverse
deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-security main restricted universe multiverse
```

4.更新源和软件

```
sudo apt-get update
sudo apt-get upgrade
```

3.2 RISC-V 工具链的配置

在 Ubuntu 的终端中安装相应的 RISC-V 工具链：

```
sudo apt-get update && sudo apt-get upgrade -y
sudo apt-get install -y git build-essential gdb-multiarch qemu-system-misc gccriscv64-linux-gnu binutils-riscv64-linux-gnu
```

3.3 克隆 xv6 源码

Xv6 完整源码：

```
git clone git://github.com/mit-pdos/xv6-riscv.git
```

实验所用的源码：

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

3.4 本地编译

切换到刚刚克隆到本地的文件夹下:

```
cd xv6-labs-2021
```

切换到实验分支:

```
git checkout util
```

编译，进入操作系统:

```
make qemu
```

3.5 SSH 远程连接

为提高完成实验和进行操作的效率，笔者使用了安装在宿主机(Windows 10 系统)的 VScode 与虚拟机进行 SSH 远程连接，从而可以使用 VScode 对 Ubuntu 系统下的文件进行修改。

3.5.1 Ubuntu 系统的配置

Ubuntu 下 安装 OpenSSH-Server 并配置

```
// 先卸载
 sudo apt-get remove openssh-server
// 安装
sudo apt-get install openssh-server
 // 重启 ssh 服务
sudo service ssh --full-restart
// 自动启动
sudo systemctl enable ssh
```

编辑配置文件

```
sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.backup # 备份 sudo vim /etc/ssh/sshd_config # 编辑
```

配置文件中添加如下配置:

```
Port 22
UsePrivilegeSeparation no
PasswordAuthentication yes
PermitRootLogin yes
AllowUsers os001# 这里的 "os001" 改成登陆用户名
RSAAuthentication yes
PubKeyAUthentication yes
```

端口可选，默认为 22，对应前面配置文件中的端口号即可。
保存后，重启 ssh 服务器，重启服务：

```
sudo service ssh --full-restart
```

3.5.2 VScode 的配置

在 VScode 的插件商店中搜索并下载 Remote - SSH 插件，下载完成后，

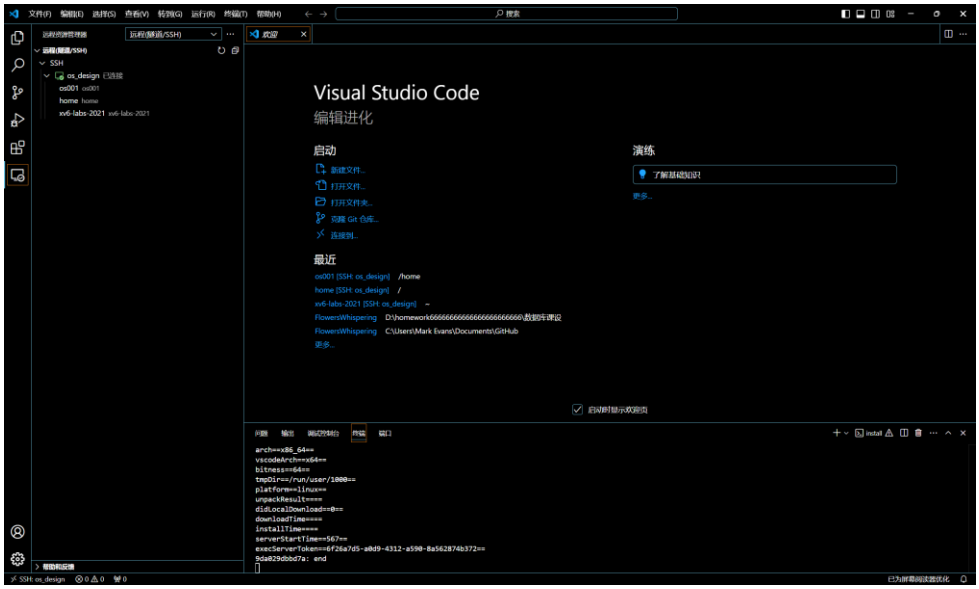
1. 进入 设置 ，搜索 ssh ，找到并选中拓展中的 Remote-SSH 中的 Show Login Terminal 选项， 因为在连接的时候，终端会让你输入 yes 或者密码等。
2. 接着，需要配置你的 Linux 服务器地址信息，按 CTRL+SHIFT+P ，搜索 ssh ，找到 Open Configuration File 选项

（如何获取自己虚拟机的 IP 地址？ 在终端中输入 ifconfig 即可）

3. 输入如下：

```
Host os_design
  HostName IP 地址
  User os001
```

然后即可进行连接，成功连接后如下图所示：



三、lab1：Xv6 and Unix utilities

1. Boot xv6

1.1 实验要求

理解操作系统的引导启动过程，编译代码库中的源码，并通过 qemu 编译运行 xv6 系统。

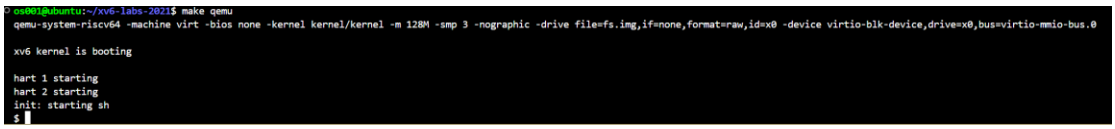
1.2 实验内容

在 xv6-labs-2021 文件夹中将实验的代码库切换到 Lab Utilities ，而后即可使用 make qemu 运 行 xv6 即可。

```
cd xv6-labs-2021-util
git checkout util
make qemu
```

1.3 实验结果

输入 make qemu 后， make 将自动执行编译和链接操作， qemu 也将启动 xv6 系统，得到结果如下：



1.4 实验注意事项

在 qemu 中，想要强制终止程序运行，不能使用 ctrl + C 强制结束程序，需要先按 ctrl + A 再按 X 即可退出。

要注意 git 分支的切换和之前环境的配置。

2. sleep

2.1 实验要求

在该实验中，我们需要实现 Unix 中经典的实用工具 sleep 。sleep 工具的作用是等待给定的 tick 数 并退出（tick 是指两次次时钟中断的间隔时间）。

2.2 实验内容

根据要求，我们需要将实现的源码放置在 user/sleep.c 。在开始实验之前，我们可以先观察 user/echo.c , `user/grep.c , user/rm.c 等文件中的书写风格和习惯。

而后我们在 user/sleep.c 进行源码的编写，实现代码如下

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

int main(int argc,char* argv[])
{
    int second = atoi(argv[1]);
    if(argc <= 1){
        printf("usage: sleep [seconds]\n");
        exit(0);
    }
    sleep(second);
    exit(0);
}
```

思路如下：

由于 sleep 只接受一个参数，因此先通过 argc 对于输入的参数数量进行判断，如果小于等于 1，则提示用户 sleep 的使用方法；否则通过 atoi 函数将用户输入的秒数转换为 int 变量，最后使用 exit(0) 退出进程。

2.3 实验结果

结束代码的编写后，使用 make qemu 进行编译运行，输入

```
sleep 100
```

可以使进程睡眠，而后在终端中输入

```
./grade-lab-util sleep
```

评测结果如下图所示，可见测试全部通过。

```
os001@ubuntu:~/xv6-labs-2021$ ./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 (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.9s)
```

2.4 实验注意事项

在完成代码的编写后，需要将该源文件加入到 xv6 的 Makefile 里，这样工具链才会编译我们 刚才编写的用户程序。打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面 加入一行：

```
$U/_sleep\
```

我们应注意到在该实验中编写 c 语言程序和往常不同， include 部分引入的头文件不是标准的 c 语言库的头文件，而且程序结束时使用的是 exit(0) 而非一般的 return 0 。

3. pingpong

3.1 实验要求

在本实验中，需要实现一个名为 pingpong 的实用工具，用以验证 xv6 中的进程之间相互通信的一 些机制。

3.2 实验内容

在该实验中，程序需要创建一个子进程，并使用管道与子进程通信：父进程首先发送一字节的数据 给子进程，子进程接收到该数据后，在 shell 中打印 ：received ping ，其中 pid 为子进程的进 程号；子进程接收到数据后，向父进程通过管道发送一字节数据，父进程收到数据后在 shell 中打印 ：received pong ， 其中 pid 为父进程的进程号，实现代码如下：

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
char buf[128];
int main(int argc, char* argv[])
{
    int p1[2], p2[2], pid;
    pipe(p1); // 创建管道 p1，用于父进程向子进程发送数据
    pipe(p2); // 创建管道 p2，用于子进程向父进程发送数据
    if (fork() == 0) // 创建子进程
   {
        close(p2[1]); // 关闭子进程中管道 p2 的写入端
        close(p1[0]); // 关闭子进程中管道 p1 的读取端
        pid = getpid(); // 获取子进程的进程 ID
        int num = read(p2[0], buf, 1); // 从管道 p2 的读取端读取 1 个字节的数据到 buf 中
        if (num == 1) // 如果成功读取了一个字节的数据
     {
       printf("%d: received ping\n", pid); // 打印接收到 ping 的消息和子进程的进程 ID

       write(p1[1], buf, 1); // 将读取到的数据写入管道 p1 的写入端，即向父进程发送 pong 消息

       }
    }
    else // 父进程
    {
        close(p1[1]); // 关闭父进程中管道 p1 的写入端
        close(p2[0]); // 关闭父进程中管道 p2 的读取端
        pid = getpid(); // 获取父进程的进程 ID
        write(p2[1], buf, 1); // 向管道 p2 的写入端写入 1 个字节的数据，即向子进程发送 ping 消息

        int num = read(p1[0], buf, 1); // 从管道 p1 的读取端读取 1 个字节的数据到 buf 中
        if (num == 1) // 如果成功读取了一个字节的数据
        {
            printf("%d: received pong\n", pid); // 打印接收到 pong 的消息和父进程的进程 ID

    }
    }
     exit(0); // 退出程序
}
```

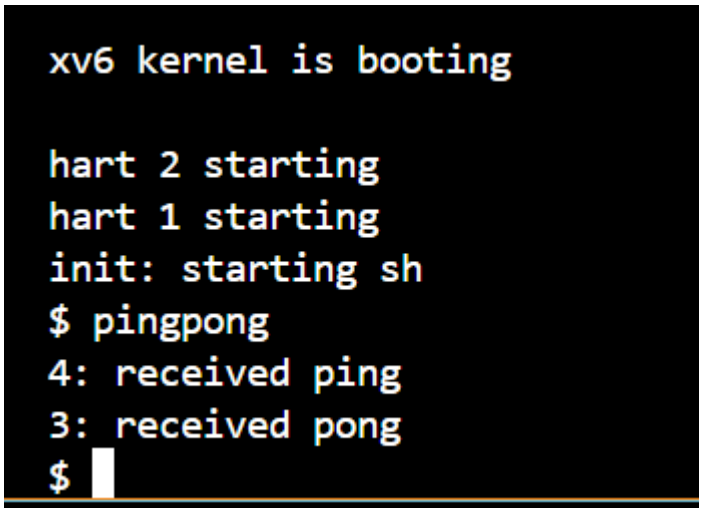
这里涉及到 fork 的本身机制： fork 会把父进程的寄存器（又称为上下文）、内存空间和进程控制 块复制一份，用来生成子进程，并且将子进程的进程控制块中的父进程指针指向其父进程；此时父子进 程几乎完全一致，为了方便程序判断自己是父还是子， fork 会给两个进程不同的返回值，父进程得到 的是子进程的 pid ， 而子进程的返回值则为 0。

3.3 实验结果

实验完成后，输入 make qemu 编译，输入

```
pingpong
```

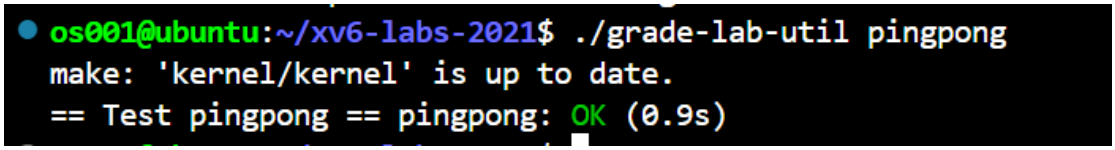
得到结果如下：



再输入

```
./grade-lab-util pingpong
```

得到结果如下：



3.4 实验注意事项

在完成代码的编写后，需要将该源文件加入到 xv6 的 Makefile 里，这样工具链才会编译我们 刚才编写的用户程序。打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面 加入一行：

```
$U/_pingpong\
```

要注意子进程和父进程的需要端口部分，需要通过在 fork 前打开两个双向管道，在父子进程中 关闭对应管道的读端和写端的方式进行解决。

4. primes

4.1 实验要求

本实验需要使用管道（pipe）和进程复制（fork）来建立一个管道。第一个进程将数字 2 到 35 输入 管道中。对于每个质数，我需要创建一个进程，它通过一个管道从左边的进程读取数据，并通过另一个 管道向右边的进程写入数据。由于 xv6 系统有限的文件描述符和进程数量，第一个进程可以在数字 35 时 停止。

4.2 实验内容

一个经典的案例是 Doug McIlroy 提出的基于管道的多进程质数筛，该质数筛的基本思路是每一个 进程负责检验一个质数是否为输入的因子，在每次发现一个新质数后，就新建一个进程并令其负责检验 该质因子，进程间通过管道传递数据，具体的实现代码如下:

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
const int maxN = 35;
int isfirst = 1;
int mypipe[2], frd, fwt;
long num;//每次被筛选的数
long prime;
int main(int argc, char* argv[])
{ //第一个进程进行特殊处理
    if (isfirst)
    {
        pipe(mypipe);
        frd = mypipe[0];//读取端
        fwt = mypipe[1];//写入端
        isfirst = 0;
        for (num = 2; num <= maxN; num++)
       {
        write(fwt, (void *)&num, 8);
       }
        close(fwt);
    }
    if (fork() == 0)
    {
        if (read(frd, (void *)&prime, 8))
       {
           printf("prime %d\n", prime);
           pipe(mypipe);
           fwt = mypipe[1];
        }
        while (read(frd, (void *)&num, 8))
        {
        if (num % prime != 0)
           write(fwt, (void *)&num, 8);
        }
        frd = mypipe[0];
        close(fwt);
        main(argc, argv);
    }
    else
    {
        wait((int*)0);
        close(frd);
    }
    exit(0);
}
```

最初的父进程负责产生 2 到 maxN 的整数，然后产生一个子进程用于检验 2。第一个不能被 2 整除的数用于产生下一个子进程，并输出该数，且质数筛的性质保证了该数为质数。

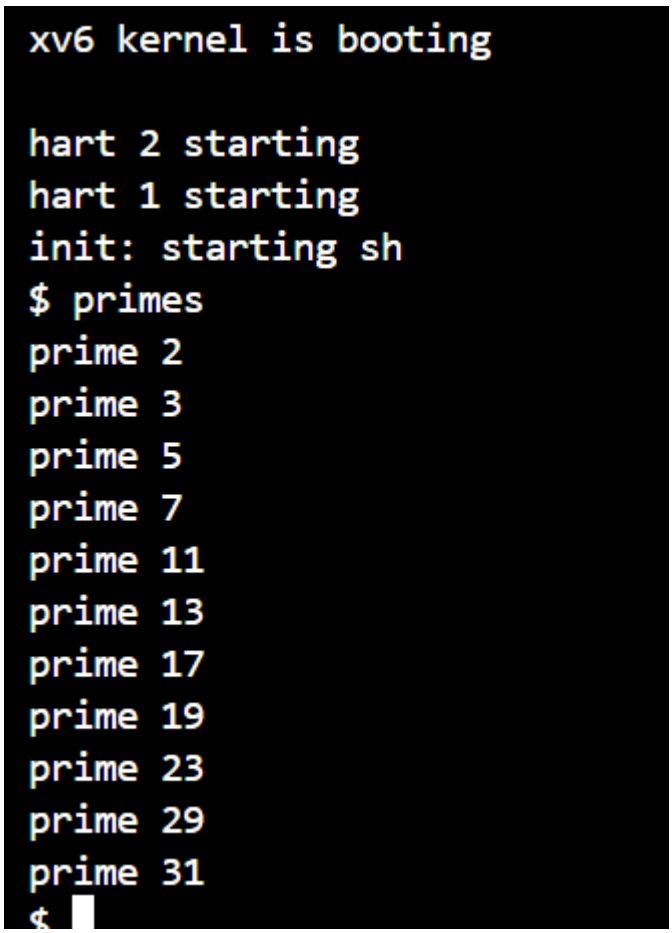
实现过程中，需要维护读端和写端的管道，不断读取上一个进程写入管道的内容，并在合适的条件下生成子进程并将其它数字写入管道。笔者的实现为了简洁，使用了主函数递归来避免多次的 fork 调用。

4.3 实验结果

实验完成后，输入 make qemu 编译，输入

```
primes
```

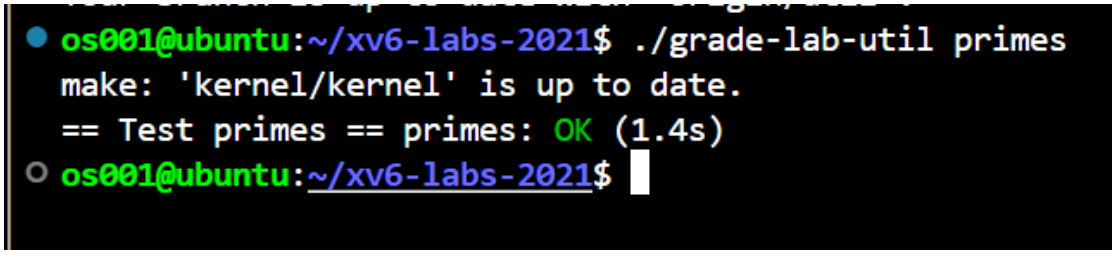
得到结果如下：



再输入

```
./grade-lab-util primes
```

得到结果如下：



4.4 实验注意事项

在完成代码的编写后，需要将该源文件加入到 xv6 的 Makefile 里，这样工具链才会编译我们 刚才编写的用户程序。打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面 加入一行：

```
$U/_primes\
```

需要对管道逻辑的具体原理进行掌握

5. find

5.1 实验要求

在本实验中需要编写一个简化版本的 UNIX find 程序：在一个目录树中查找所有具有特定名称的文 件。

5.2 实验内容

实验的主要思路是通过递归地遍历目录树，对每个文件和目录执行所需的操作。在处理文件时，比 较文件名是否匹配；在处理目录时，读取目录项并递归调用 find 函数以深入子目录。这样，就能够找 到满足条件的文件，并输出它们的完整路径，实现代码如下：

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fs.h"
// 从完整路径中提取文件名
char *fmtname(char *path)
{
    char *p;// Find first character after last slash.
    for (p = path + strlen(path); p >= path && *p != '/'; p--)
    ;
    p++;
    return p;
}
// 递归查找目录中具有指定名称的文件
void find(char *path, char *filename)
{
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;
    if ((fd = open(path, 0)) < 0)
    {
     fprintf(2, "find: cannot open %s\n", path);
    return;
     }
    if (fstat(fd, &st) < 0)
        {
        fprintf(2, "find: cannot stat %s\n", path);
        close(fd);
        return;
    }
    switch (st.type)
    {
    case T_FILE:
        if (strcmp(fmtname(path), filename) == 0)
        {
        printf("%s\n", path);
        }
        break;
    case T_DIR:
        if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf)
        {
        printf("find: path too long\n");
        break;
        }
        strcpy(buf, path);
        p = buf + strlen(buf);
        *p++ = '/';// 循环读取下一个目录项
0)
```

```c
 while (read(fd, &de,
sizeof(de)) ==
sizeof(de))
        {
            if (de.inum ==
0)
                continue;
            memmove(p,
de.name, DIRSIZ);
            p[DIRSIZ] = 0;
            if (stat(buf,
&st) < 0)
            {
            printf("find:
cannot stat %s\n", buf);
            continue;
            }// 如果是.或
者..则说明是父文件夹忽略
            if
(strcmp(fmtname(buf),
".") != 0 &&
strcmp(fmtname(buf),
"..") != 0)
            {
            find(buf,
filename);
             }
        }
        break;
    }
    close(fd);
}
int main(int argc, char
*argv[])
{
if (argc != 3)
   {
        printf("usage:
find [path]
[filename]\n");
        exit(0);
    }
    find(argv[1],
argv[2]);
    exit(0);
}
```

对于 find 函数：是递归查找函数的主体。它会打开指定的路径，并检查文件类型以及目录中的所 有项。根据文件类型，采取不同的操作。

如果是文件（T_FILE 类型），则将文件名与所需的文件名进行比较，如果相同，则打印出文件 的完整路径。

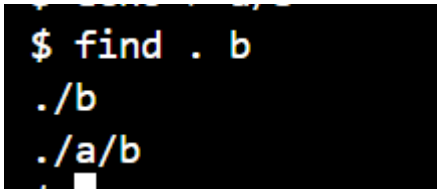
如果是目录（T_DIR 类型），则在目录中循环读取目录项，忽略特殊目录"."和"..."，然后对每个 子项递归调用 find 函数。

5.3 实验结果

实验完成后，输入 make qemu 编译，输入

```
echo > b
mkdir a
echo > a/b
find . b
```

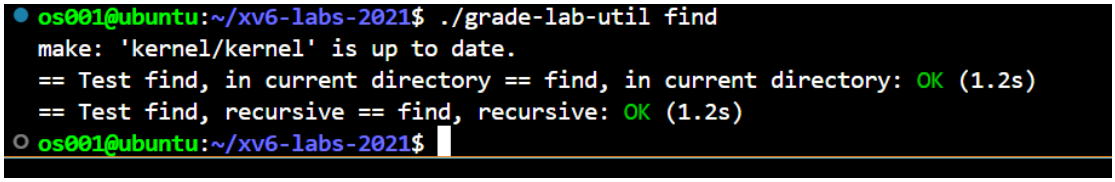
得到结果如下：



再输入

```
./grade-lab-util find
```

得到结果如下：



5.4 实验注意事项

在完成代码的编写后，需要将该源文件加入到 xv6 的 Makefile 里，这样工具链才会编译我们 刚才编写的用户程序。打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面 加入一行：

```
$U/_find\
```

注意文件和目录的不同处理方式：文件直接和指定的文件名进行比较，目录则要遍历目录中的 每个目录项。

6. xargs

6.1 实验要求

xargs 是一个简化版的命令行工具，用于将标准输入的内容作为命令的参数传递，并执行该命令。

6.2 实验内容

实现一个简化版本的 xargs 命令，通过从标准输入读取每行参数，将这些参数作为命令的参数，并通 过 fork 和 exec 执行这些命令。在主函数中，通过循环调用 readline 函数，逐行读取标准输入的参数。 对于每一行参数，使用 fork 创建一个子进程。在子进程中，设置参数数组 pass_args ，然后使用 exec 函数执行指定的命令，传递参数数组。父进程等待子进程执行结束，然后继续读取下一行参数并重复上 述过程。

实现代码如下：

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

#include "kernel/stat.h"

#include "kernel/param.h"

#include "user/user.h"

char args[MAXARG][512]; // 存储命令行参数的二维字符数组

char *pass_args[MAXARG]; // 用于传递给 exec 函数的参数数组

int preargnum, argnum; // preargnum：命令行参数的数量；argnum：每行输入的参数数量

char ch; // 存储读取的字符

char arg_buf[512]; // 临时存储每行输入的参数

int n; // 存储 read 函数返回的字节数

// 读取一行输入

int readline()

{

    argnum = preargnum;// 将命令行参数数量赋值给 argnum

    memset(arg_buf, 0, sizeof(arg_buf));// 清空参数缓冲区

    for (;;)

     {

      n = read(0, &ch, sizeof(ch));// 从标准输入读取一个字符

      if (n == 0)

     {

       return 0;// 如果读取结束，返回 0

      }

       else if (n < 0)

      {

          fprintf(2, "read error\n");// 读取错误，输出错误信息

          exit(1);

       }

       else

      {

       if (ch == '\n')

       {

         memcpy(args[argnum], arg_buf, strlen(arg_buf) + 1);// 将参数缓冲区的内容复制到 args 数组中

         argnum++;// 参数数量加一

        return 1;// 返回读取成功的状态

       }

       else if (ch == ' ')

       {

        memcpy(args[argnum], arg_buf, strlen(arg_buf) + 1);// 将参数缓冲区的内容复制到 args 数组中
```

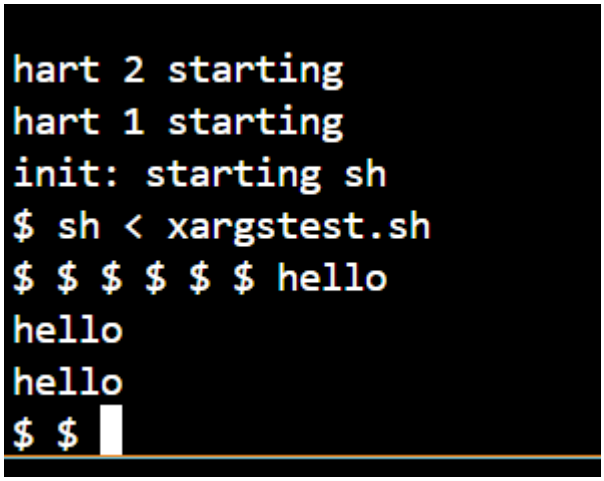
```c
int main(int argc, char *argv[])

{

    if (argc < 2)

    {

        printf("usage: xargs [command] [arg1] [arg2] ... [argn]\n");

        exit(0);

   }

     preargnum = argc - 1;// 获取命令行参数的数量

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

     memcpy(args[i], argv[i + 1], strlen(argv[i + 1]));// 将命令行参数复制到 args 数组中

    while (readline())// 循环读取输入的每一行

   {

   if (fork() == 0)

    {

       *args[argnum] = 0;// 参数数组末尾置空

        int i = 0;

       while (*args[i])

       {

       pass_args[i] = (char *)&args[i];// 将参数数组中的每个参数地址赋值给 pass_args 数组

       i++;

       }

        *pass_args[argnum] = 0;// pass_args 数组末尾置空

       exec(pass_args[0], pass_args);// 执行指定的命令

       printf("exec error\n");// 如果 exec 函数执行失败，输出错误信息

       exit(0);

    }

    else

    {

     wait((int *)0);// 父进程等待子进程结束

    }

   }

    exit(0);

}
```

6.3 实验结果

输入：

```
sh < xargstest.sh
```

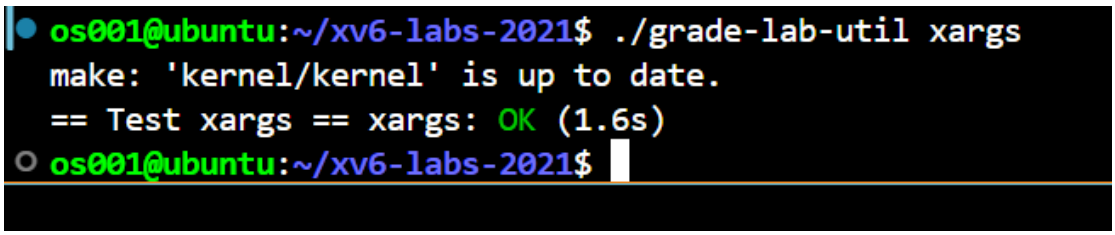
得到结果如下：



再输入

```
./grade-lab-util xargs
```

得到结果如下：



6.4 实验注意事项

在完成代码的编写后，需要将该源文件加入到 xv6 的 Makefile 里，这样工具链才会编译我们 刚才编写的用户程序。打开 Makefile ，找到 UPROGS 环境变量，然后按照其格式，在其后面 加入一行：

```
$U/_xargs\
```

7.完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-util
git checkout util
./grade-lab-util
```

结果如下，可见测试全部通过：

```
os001@ubuntu:~/xv6-labs-2021-util$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.5s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.1s)
== Test find, in current directory == find, in current directory: OK (1.1s)
== Test find, recursive == find, recursive: OK (1.3s)
== Test xargs == xargs: OK (1.1s)
== Test time ==
time: OK
Score: 100/100
os001@ubuntu:~/xv6-labs-2021-util$
```

8.遇到的问题及解决方法

在进行本次实验中时，因为是初次接触到操作系统的内核和第一次写不引入标准 c 语言库的 c 程 序，难免会有不习惯的地方，同时在编写程序时会遇到过内存错误、死锁、错误的文件操作等 问题，需要参考实验指导书以及网上的资料对代码进行调试修改。 总是会忘记在 Makefile 中添加新编写的程序的入口，导致编译失败。

9. 实验收获

在完成"Lab: Xv6 and Unix utilities"实验后，我获得了许多宝贵的实验收获。我对操作系统的底层 机制有了更深刻的理解，通过实际编码和调试，我深入探索了进程管理、文件操作和系统调用等重要概 念。这使我能够更好地理解操作系统是如何运作的。

通过实现类似 Unix 实用工具的功能，我加深了对进程间通信和同步的认识。处理并发操作时，我学会了如何使用锁和信号量来避免竞态条件和死锁问题，以及如何通过进程间通信机制实现数据共享和通信。实验也锻炼了我的编程技能和调试能力。我在编写代码、测试功能、定位错误和修复 bug 的过程 中积累了宝贵的经验。通过与同学的讨论和资料查阅，也收获到了获得新知识的快乐。

四、lab2：system calls

1. System call tracing

1.1 实验要求

在这个实验中需要添加一个系统调用跟踪功能，这将在后续实验中进行调试时可能会有所帮助。需 要创建一个新的跟踪系统调用，用于控制跟踪。该系统调用应该接受一个整数参数 mask ，其中的位指 定要跟踪的系统调用。例如，要跟踪 fork 系统调用，程序会调用 trace(1 << SYS_fork) ，其中 SYS_fork 是来自 kernel/syscall.h 的系统调用号。你需要修改 xv6 内核，在每个系统调用即将返回 时，如果该系统调用的号码在 mask 中被设置，就会打印出一行信息。该行信息应该包含进程 ID、系统调用的名称和返回值。

1.2 实验内容

在 xv6-labs-2021 文件夹中将实验的代码库切换到 Lab Utilities ， 而后即可使用 make qemu 运 行 xv6 即可。

```
cd xv6-labs-2021-syscall
git fetch
git checkout syscall
make clean
```

按照下面的步骤将 trace 系统调用加入到户态库和内核中：

1. 在 user/user.h 中的系统调用部分加入一行

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

作为 trace 系统调用在用户态的入口。

2. 在 user/usys.pl 中的系统调用部分加入一行

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

用以生成调用 trace(int) 入口时在用户态执行的汇编代码（这段代码被称为 stubs ）。

3. 在 kernel/syscall.h 中

```
#define SYS_trace 22
```

为 trace 分配一个系统调用的编号。

4.为了保存每个进程的 trace 的参数，我们需要在进程控制块中加入一个新的变量。查看 kernel/proc.h ，找到进程的数据结 构 struct proc ，在其中加入一行 int tracemask; ，如下所示：

```
int tracemask;
```

5. 为了将 trace(int) 的参数保存到 struct proc 中，我们需要在 kernel/sysproc.c 中实现 sys_trace(void) 函数，代码如下：

```
uint64
sys_trace(void)
{
  // printf("sys_trace: called.\n");
  int mask;
  argint(0, &mask);
   // printf("trace: the mask is %d \n", mask);
  myproc()->tracemask = mask;
  return 0;
}
```

6. 其中， argint(0, &mask); 用于获取用户传入系统调用的第一个参数，而 myproc() 则返回当前进程控制块的指针，将其 tracemask 赋值为获取到的参数即可。然后我们需要修改 kernel/syscall.c 中的 syscall(void) ，使其能够根据 tracemask 打印所需要的信息。首 先定义一个字符串常量数组，保存各系统调用的名称：

```
extern uint64 sys_trace(void);
...
static uint64 (*syscalls[])(void) = { ...
[SYS_trace] sys_trace, };
```

7. 然后修改 syscall(void) ，在调用系统调用后，增加如下 if 语句的内容：

```
void syscall(void)
{
  int num;
  struct proc *p = myproc();
  num = p->trapframe->a7;
  // if (num == 22)
  //   printf("%d: called system call %d.\n", p->pid, num);
  if (num > 0 && num < NELEM(syscalls) && syscalls[num])
  {

    p->trapframe->a0 = syscalls[num]();

    if (p->tracemask >> num & 1)
    {
      printf("%d: syscall %s -> %d\n",
             p->pid, syscallnames[num], p->trapframe->a0);
    }
```

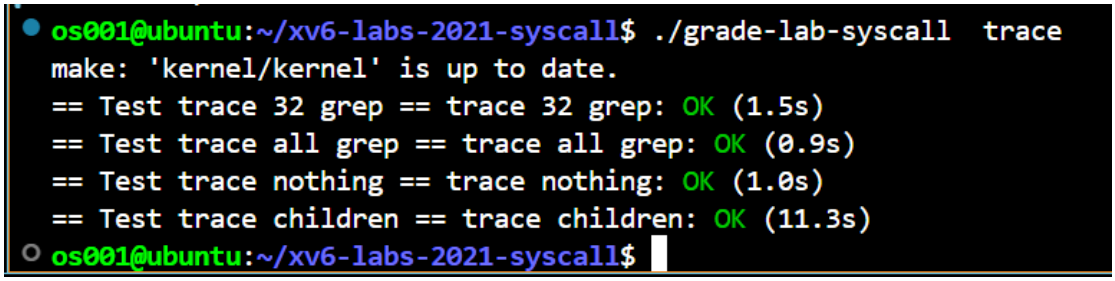
8. 最后，为了让 fork 后的子进程能够继承 trace 的 tracemask，需要在 kernel/proc.c 中的 fork(void) 中加入复制 tracemask 的语句

```
// copy trace mask
  np->tracemask = p->tracemask;
```

1.3 实验结果

实验完成后，输入

```
cd xv6-labs-2021-syscall
git fetch
git checkout syscall
./grade-lab-syscall    trace
```



1.4 实验注意事项

首先要理解命令行各个参数的具体含义，这是最基本的要求。

要注意在 user/user.h 、 user/usys.pl 、 kernel/syscall.h 、 kernel/syscall.c 等文 件里添加程序调用的入口，后面很多实验程序这几点也至关重要。

2. Sysinfo

2.1 实验要求

在这个实验中，需要将添加一个名为 sysinfo 的系统调用，用于收集有关正在运行的系统的信息。 这个系统调用接受一个参数： 一个指向 struct sysinfo 的指针（参见 kernel/sysinfo.h ）。内核应 该填充这个结构的字段： freemem 字段应该设置为空闲内存的字节数， nproc 字段应该设置为状态不 是 UNUSED 的进程数量。

2.2 实验内容

首先效仿上一个 trace 实验，我们需要在 user/user.h 、 user/usys.pl 、 kernel/syscall.h 、 kernel/syscall.c 等文件中添加程序调用的入口，这里不再赘述。

然后为了获得空闲内存大小，我们需要在 kernel/kalloc.c 实现一个函数。由于 xv6 管

理内存空闲空 间使用的是空闲链表，参照链表的结构后，只需遍历链表并计算数量，然后乘以页面大小即可。实现如 下：

```
// Return the amount of free memory.
int getfreemem(void)
{
  int count = 0;
  struct run *r;
  acquire(&kmem.lock);
   r = kmem.freelist;
  while (r)
 {
    count++;
    r = r->next;
 }
  release(&kmem.lock);
  return count * PGSIZE;
}
```

统计空闲进程控制块的数量的函数用静态数组管理，需要用一个循环遍历该数组：

```
// Return the number of non-UNUSED procs in the process table.
int getnproc(void)
{
  struct proc *p;
  int count = 0;
  for (p = proc; p < &proc[NPROC]; p++)
 {
   acquire(&p->lock);
   if (p->state != UNUSED)
  {
   count++;
    release(&p->lock);
  }
   else
  {
    release(&p->lock);
  }
 }
  return count;
```

由于虚拟存储机制的存在，内核态的页表与用户态的页表不同，而用户传入的地址是用

户态的地 址，故而剩下的问题就在于如何将内核态的数据复制到用户态的数据结构中。这里用的方法将 struct sysinfo 拷贝到用户态进程中，在 kernel/sysproc.c 中的实现如下：

```
extern int getnproc(void);
extern int getfreemem(void);
uint64
sys_sysinfo(void)
{
  struct proc *p = myproc();
  struct sysinfo st;
  uint64 addr; // user pointer to struct stat
  st.freemem = getfreemem();
  st.nproc = getnproc();
  if (argaddr(0, &addr) < 0)
    return -1;
  if (copyout(p->pagetable, addr, (char *)&st,
sizeof(st)) < 0)
    return -1;
  return 0;
}
```

2.3 实验结果

实验完成后，输入

```
./grade-lab-syscall sysinfo
```

得到结果如下，可见测试通过：

```
os001@ubuntu:~/xv6-labs-2021-syscall$ ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (2.1s)
os001@ubuntu:~/xv6-labs-2021-syscall$
```

2.4 实验注意事项

理解 copyout(p->pagetable, addr, (char *)&st, sizeof(st))：接受进程的页表，并将内核态中 的一段数据拷贝到进程内存空间中的地址处。因此可以利用这段代码将结构体拷贝到用户态进 程中；

访问内核态中的数据结构时，最好进行获取锁和释放锁的操作，避免数据变“脏”：

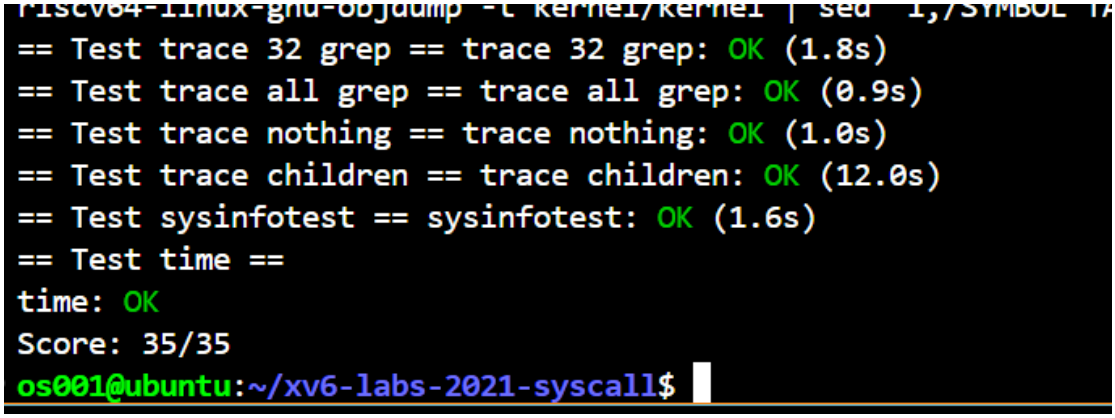
```
acquire(&kmem.lock);
release(&kmem.lock);
```

3. 完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-syscall
git fetch
git checkout syscall
./grade-lab-syscall
```

结果如下，可见测试全部通过：



4. 遇到的问题及解决方法

在最开始做本章节的实验时，由于经验尚且加上自己的粗心大意，导致很多时候因为忘记添加必要 的入口和调用方法而编译不通过，在花费了很大的时间代价查出错误后，吸取了教训，对于实验指导书 应该认真阅读，逐步操作。

5. 实验收获

通过对于整个实验的回望可以得出系统调用的简要执行过程：

1. 在 user/user.h 中的系统调用部分加入系统调用在用户态的入口。
2. 在 user/usys.pl 中的系统调用部分加入在用户态执行的汇编代码（ stubs ）。
3. 在 kernel/syscall.h 中为系统调用分配一个系统调用的编号。
4. 在内核代码中实现系统调用，包括设计对应的数据结构和函数等。

五、lab3： page tables

1. Speed up system calls

1.1 实验要求

一些操作系统（如 Linux ）通过在用户空间和内核之间的只读区域共享数据，加速特定的系统调 用。这消除了执行这些系统调用时需要进行内核交叉的需要。在本实验中，我们需要在 xv6 中为 getpid 系统调用实现这种优化。

1.2 实验内容

```
cd xv6-labs-2021-pgtbl
git fetch
git checkout pgtbl
make clean
```

我们需要在进程创建时，将一个只读权限的内存页面映射到 USYSCALL 位置。这个映射的页面开头 会存储一个内核数据结构 struct usyscall ，在进程创建时进行初始化，用于存储进程的 pid 信息。 在这个实验中，我们可以使用用户空间函数 ugetpid() 来替代需要进行内核态到用户态拷贝的 getpid 系统调用。 ugetpid() 函数已在用户空间实现，我们只需完成映射页面的工作。

为实现这个目标，我们需要修改 kernel/proc.c 中的 proc_pagetable(struct proc *p) 函数， 该函数用于为新创建的进程分配页面。在 proc_pagetable() 函数中，完成新页面的分配后，可以使用 mappages 函数进行页面映射，页面映射时，需要设定其权限为只读，同时在进程结束后应释放内存， 故实现如下：

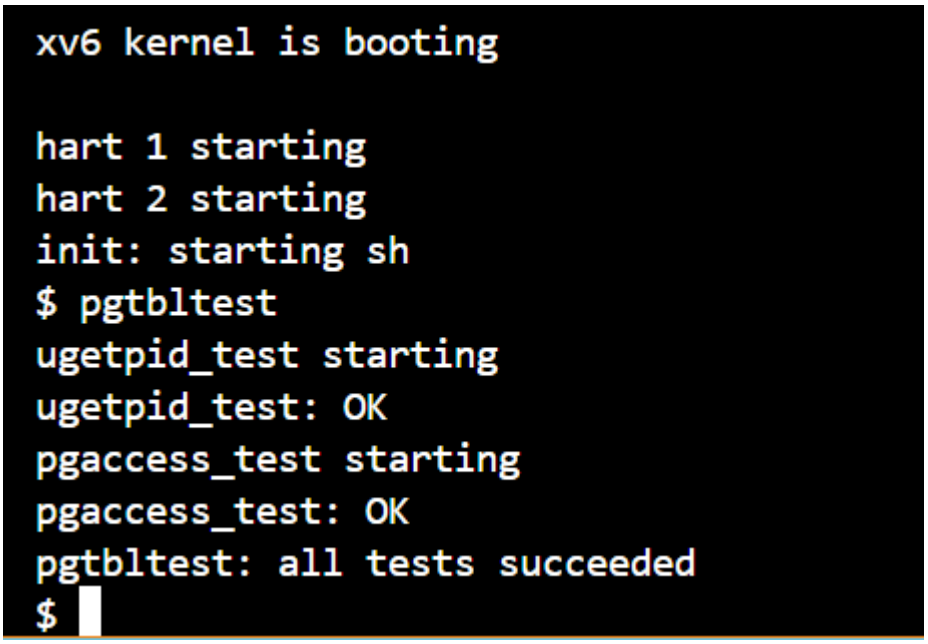
```
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;
  // 创建一个空页表。
  pagetable = uvmcreate();
  if(pagetable == 0)
    return 0;
  // 将一个只读的用户页映射到 USYSCALL，以进行优化，用于 getpid()。
  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }
  // 将跳板代码（用于系统调用返回）映射到最高的用户虚拟地址。
  // 只有内核使用它，在进入/退出用户空间时使用，所以不包含 PTE_U。
  if(mappages(pagetable, TRAMPOLINE, PGSIZE,
              (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
```

1.3 实验结果

实验完成后，输入

```
pgtbltest
```

得到结果如下，可见测试通过：



1.4 实验注意事项

考虑并发情况：由于在内核态进行相关操作，可能会有并发读写的情况。因此，在执行共享页 的分配和数据结构初始化时，必须确保使用适当的锁机制来避免并发问题。

设置权限位：为确保页面在正确的权限下进行映射，需要注意设置适当的权限位。例如，在创 建只读的共享页时，应当设置权限位为 PTE_R | PTE_U ，以确保该页面的内容对用户空间可 见且只读。

2. Print a page table

2.1 实验要求

为了可视化 RISC-V 页表，并可能有助于以后的调试，该实验需要编写一个函数，用于打印页表的内 容。

2.2 实验内容

为了方便使用与页表相关的函数，我们将在 kernel/vm.c 中实现一个名为 vmprint() 的函数。通 过借鉴递归的方式，我们可以稍作修改来构建一个用于遍历页表并按层数格式打印内容的函数 vmprintwalk(pagetable_t pagetable, int depth) 。然后，在实现 vmprint() 函数时，只需要将 初始深度设置为 1，调用 vmprintwalk(pagetable_t,int) 。最后，按照要求，在 kernel/exec.c 中 的 exec() 函数的 return argc 之前，添加条件语句 if(p->pid==1)vmprint(p->pagetable) ，以实现 在特定情况下对页表内容的打印。

vmprint() 的实现如下：

```
void
vmprint(pagetable_t pagetable)
{
  printf("page
table %p\n",pagetable);
  vmprintwalk(pagetable,1);
}
```

vmprintwalk(pagetable_t pagetable, int depth) 的实现如下：

```
void
vmprintwalk(pagetable_t pagetable, int depth)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // this PTE points to a lower-level page table.
      for (int n = 0; n < depth; n++)
        printf(" ..");
      printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
      uint64 child = PTE2PA(pte);
      vmprintwalk((pagetable_t)child, depth+1);
    } else if(pte & PTE_V){
```

exec() 的实现如下：

```
if(p->pid==1) vmprint(p->pagetable);
return argc; // this ends up in a0, the first argument to main(argc, argv)
```

2.3 实验结果

编译并启动 xv6 后，即可看到预期的输出：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6e000
 ..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
 .. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
 .. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
 .. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
 .. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
 ..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
 .. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
 .. .. ..509: pte 0x0000000021fddc13 pa 0x0000000087f77000
 .. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

2.4 实验注意事项

根据实验手册的提示，我可以参考 freewalk 函数的实现，发现它也是通过递归的方式逐层释 放页表。基于这一思路，我决定采用递归的方式来实现对三级页表的 PTE 打印。在 exec 函数中添加这段代码的目的是：让用户进程在初始化后就能够调用 vmprint 函数。这 样，当用户进程开始运行时，可以在需要的时候输出页表的内容，从而更好地了解进程的内存 映射情况。

3.Detecting which pages have been accessed

3.1 实验要求

现代编程语言往往具备内存垃圾回收（GC）功能，而实现垃圾回收器需要了解自上次检查以来页面 的访问情况。虽然可以通过纯软件方式实现，但效率较低。在 xv6 中，通过与操作系统结合，利用页表 的硬件机制（访问位），我们可以添加一个名为 pgaccess() 的系统调用，用于读取页表的访问位，并将信息传递给用户态程序，以提供页面自上次检查以来的访问情况。

pgaccess() 系统调用的实现需要接收三个参数：第一个参数表示需要检查页面的起始地址，第二个参数表示从起始地址向后检查多少个页面，第三个参数是一个指针，指向存储结果的位向量的起始地 址。这种方式可以有效地向用户程序提供页面的访问信息。

3.2 实验内容

首先，按照前文提及的方法添加系统调用以及其在 kernel/sysproc.c 中的实现 sys_pgaccess() ，并使用 argint() 和 argaddr() 获取传入的参数。接下来，我们将对获取的参数进行预处理。对于每 个需要检查的页面，我们将使用 kernel/vm.c 中提供的 walk(pagetable_t, uint64, int) 函数，来 获取它的页表项。我们会重置该页面的 PTE_A 访问位，并使用位运算将该页面的访问状态置入一个临时 位向量中。最后，我们将这个临时位向量的内容拷贝到用户内存空间中指定的地址。 这样做的目的是，通过对页面的访问位进行操作，我们可以获得页面自上次检查以来的访问情况， 并将这些信息传递给用户程序，从而实现获取页面访问状态的功能。具体实现如下：

```
extern pte_t *walk(pagetable_t, uint64, int);
int sys_pgaccess(void)
{
  // lab pgtbl: your code here.
  uint64 srcva, st;
  int len;
```

```
uint64 buf = 0;
  struct proc *p = myproc();
  acquire(&p->lock);
  argaddr(0, &srcva);
  argint(1, &len);
  argaddr(2, &st);
  if ((len > 64) || (len < 1))
    return -1;
  pte_t *pte;
  for (int i = 0; i < len; i++)
 {
    pte = walk(p->pagetable, srcva + i * PGSIZE, 0);
    if(pte == 0){
      return -1;
   }
    if((*pte & PTE_V) == 0){
      return -1;
   }
    if((*pte & PTE_U) == 0){
      return -1;
   }
    if(*pte & PTE_A){
      *pte = *pte & ~PTE_A;
      buf |= (1 << i);
   }
 }
  release(&p->lock);
  copyout(p->pagetable, st, (char *)&buf, ((len -1) / 8) + 1);
  return 0;
```

3.3 实验结果

实验完成后，输入

```
pgtbltest
```

```
 .. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
 ..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
 .. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
 .. .. ..509: pte 0x0000000021fddc13 pa 0x0000000087f77000
 .. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$
```

3.4 实验注意事项

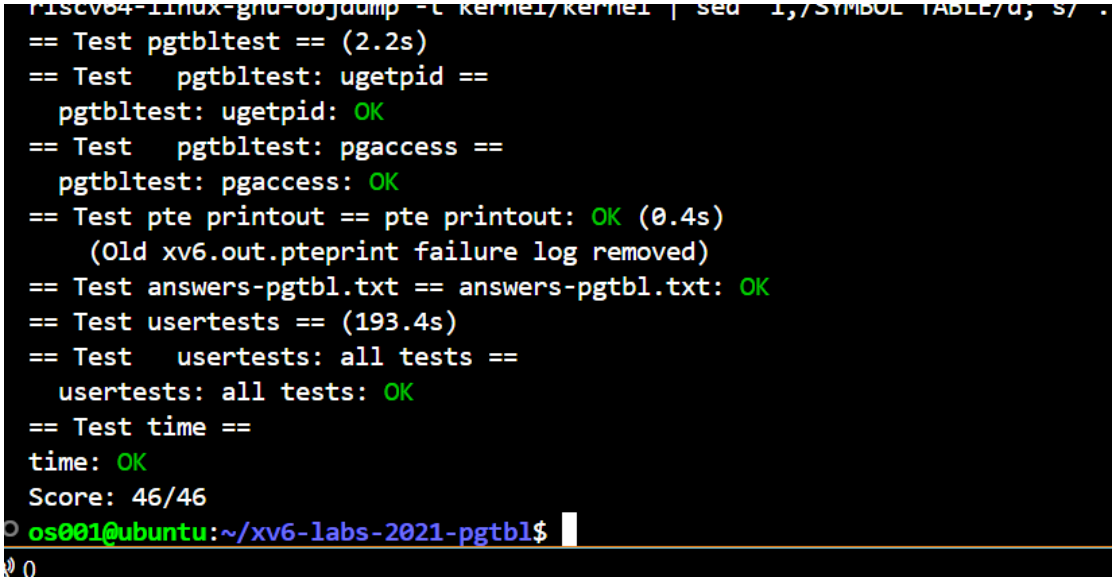
实现 pgaccess() 系统调用时，确保正确设计参数传递的方式，以及在内核态和用户态之间进 行参数传递的正确性。参数的传递和解析需要准确无误，以确保实验的正确性。 在获取页面访问状态时，要确保正确使用 walk() 函数和适当的位操作来处理页表项中的访问位。确保在获取和重置访问位时不会影响其他相关数据。

4. 完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-pgtbl
git fetch
git checkout pgtbl
make clean
./grade-lab-pgtbl
```

结果如下，可见测试全部通过：



5. 遇到的问题及解决方法

在操作页表项时，偶尔会出现错误，导致访问位、权限位等设置不正确，从而影响系统的正常运 行。仔细查阅文档和相关资料后，正确理解页表项的结构和属性。同时在代码中使用正确的位操作来处 理页表项。

6. 实验收获

通过本次实验，我了解到了 RISC-V 的页表机制是用于实现虚拟内存管理的重要部分，它允许操作系 统将物理内存映射到进程的虚拟地址空间，实现了地址隔离、内存保护和资源共享等功能。RISC-V 的页 表机制包括以下几个关键要点：

1. 多级页表结构：RISC-V 使用了多级页表结构，通常是三级。每个级别都有一个页表，用于将虚 拟地址映射到物理地址。不同级别的页表通过指针链接在一起，形成一个层次结构。

2. 虚拟地址到物理地址的映射： 每个页表项（Page Table Entry，PTE）包含了虚拟地址的一部 分和物理地址的一部分。通过对页表的遍历，可以将虚拟地址映射到相应的物理地址。

3. 访问权限和属性位： 页表项除了包含虚拟地址和物理地址，还包含了一些标志位，用于指示页 面的属性和访问权限，如读、写、执行等。这些位能够实现内存保护和访问控制。

4. 分页和分段： RISC-V 的页表机制可以支持分页和分段两种方式的内存管理。分页是将虚拟地址 空间分割成大小相等的页，而分段则是将虚拟地址空间分成多个段，每个段具有不同的属性。

5. TLB 缓存： 为了加速地址翻译过程，RISC-V 使用了快速缓存，称为 TLB（Translation Lookaside Buffer），它存储了最近的虚拟地址到物理地址的映射关系，以避免每次都要遍历整个页表。

6. TLB 失效和异常处理： 如果 TLB 中没有找到所需的虚拟地址到物理地址的映射，就会发生 TLB 失效。在这种情况下，操作系统将会处理页表遍历，将正确的映射关系添加到 TLB 中。如果出 现页表项不存在或权限错误等异常情况，会触发相应的异常处理程序。

六、lab4：traps

1. RISC-V assembly

1.1 实验要求

由于本次实验及之后的实验中，会涉及到机器指令级别的操作和调试，故而了解一些关于 RISC-V 汇 编的知识将有助于实验的进行。这个实验几乎不需要编写代码，主要内容是观察一些汇编代码和它们的 行为。

1.2 实验内容

本实验的主要内容是阅读代码并回答 xv6 手册中提出的问题：

1. 问题： 1. 哪些寄存器包含函数的参数？
   2. 例如，在 main 函数调用 printf 时，哪个寄存器保存了数字 13？

答案：

1. 'ax' 寄存器包含函数的参数，其中 'a' 表示 'argument'（参数）。
2. 寄存器 'a2' 在 main 函数调用 printf 时保存了数字 13。

2. 问题： 1. 在 main 函数的汇编代码中，函数 f 的调用在哪里？
   2. 函数 g 的调用在哪里？（提示：编译器可能会内联函数。）

答案：

1. 52: fce080e7 jalr -50(ra) # 1c
2. 34: fd0080e7 jalr -48(ra) # 0

3. 问题：函数 printf 位于哪个地址？

答案：

6c: 9cc080e7 jalr -1588(ra) # a34

4. 问题：在 main 函数中的 jalr 跳转到 printf 后，寄存器 ra 中的值是多少？

   答案： 0x1

5. 问题：运行以下代码。

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

输出是什么？这是一个 ASCII 表(http://web.cs.mun.ca/~michael/c/ascii-table.html)，将字节 映射到字符。 输出取决于 RISC-V 的小端序。如果 RISC-V 是大端序，为了得到相同的输出，你会将 i 设置为什么 值？你需要将 57616 更改为不同的值吗？ 答案： 输出是："HE110 World" 如果 RISC-V 是大端序，你需要将 i 设置为 0x726c6400。你不需要将 57616

更改为不同的值。

6. 问题：在以下代码中，'y='后面会打印什么？（注意：答案不是一个特定的值。）为什么会出 现这种情况？

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

答案：将会打印出 y= 后的地址的 4 个字节内容。这会发生因为未定义的内存内容。

2. Backtrace

2.1 实验要求

为了调试，通常会有一个回溯（backtrace）：在错误发生点上方堆栈中的函数调用列表。编译器在 每个堆栈帧中放置了一个帧指针，该指针保存了调用者的帧指针地址。你的回溯（backtrace）应该使用 这些帧指针来沿着堆栈向上走，并在每个堆栈帧中打印保存的返回地址。

2.2 实验内容

首先，在 kernel/defs.h 文件中添加声明： void backtrace(void); ，这是为了在后续步骤中能 够引入回溯函数。

按照实验指导书的提示，我们需要实现一个能够读取 s0 寄存器值的函数 r_fp() 。为此，在 kernel/riscv.h 文件中按照指导书的写法，添加这个函数的定义。

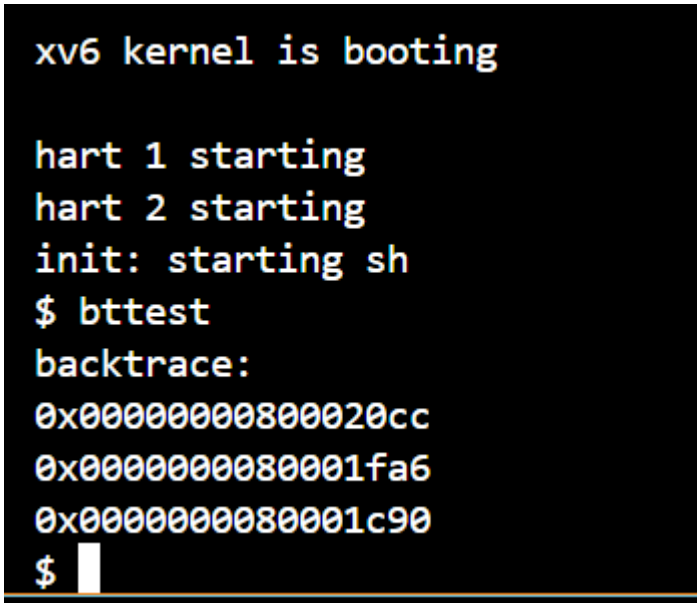
然后，在 kernel/printf.c 文件中，我们可以实现 backtrace 函数。这个函数会利用已获取的当 前函数栈帧 fp 的值，以及位于栈帧中的返回地址，来打印出回溯信息。具体代码如下：

```
void
backtrace(void)
{
  printf("backtrace:\n");
  // 从当前函数栈帧的 fp 值开始，依次向上遍历栈帧
  for (uint64 *fp = (uint64 *)r_fp();
(uint64)fp                          <
PGROUNDUP((uint64)fp); fp =
(uint64 *)(*(fp-2)))
 {
    printf("%p\n",*(fp-1)); // 打印栈帧中的返回地址
 }
}
```

最后，我们可以在 sys_sleep 和 panic 函数中添加对 backtrace 函数的调用，这样就能在这两个 关键位置触发回溯，从而完成实验的要求。

2.3 实验结果

编译运行 xv6 ， 然后执行 bttest ， 将其输出粘贴到 addr2line -e kernel/kernel ， 得到结果如 下所示:



2.4 实验注意事项

backtrace 函数使用一种迭代的方法，通过不断循环来输出当前函数的返回地址。在每次迭代中，它获取当前函数栈帧的返回地址，并将其输出。迭代过程会一直持续，直到到达了当前 页表的起始地址，此时迭代结束。这样，通过一系列迭代操作，函数可以逐步遍历栈帧，输出 每个栈帧中的返回地址。

3. Alarm

3.1 实验要求

为了实现在一个进程消耗一定数量的 CPU 时间后发出提醒，我们需要编写 sigalarm(n, fn) 系统 调用。当用户程序调用 sigalarm(n, fn) 系统调用后，在该进程消耗了 n 个时钟周期后，系统会中断 当前进程的执行并运行提供的函数 fn 。

3.2 实验内容

在实验开始之前，我们先照例在 user/user.h 加入两个系统调用的入口，分别用于设置定时器和从 定时器中断处理过程中返回:

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

首先解决第一个问题，联想到 Lab System calls 中 trace 的实现，我们需要在每个进程的控制块的数据结构中加入存储 sigalarm(n, fn) 中参数的项，即在 kernel/proc.h 的 struct proc 中加入如下的项：

```
int  alarminterval;                  // sys_sigalarm()
alarm interval in ticks
  int  alarmticks;                   // sys_sigalarm()
alarm interval in ticks
  void  (*alarmhandler)();           // sys_sigalarm()
pointer to the alarm handler
function
  struct trapframe alarmtrapframe;// for saving
registers
  int sigreturned;
```

然后我们需要在 kernel/proc.c 的 allocproc(void) 中对这些变量进行初始化，在调用 sigalarm(n, fn) 系统调用时，执行的 kernel/sysproc.c 中的 sys_sigalarm() 需根据传入的参数 设置 struct proc 的对应项，实现如下：

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

接下来，我们需要在定时器中断发生时更新具有 sigalarm 的进程已消耗的时钟周期数。为此，打 开 kernel/trap.c 文件，找到 usertrap() 函数，在判断是否为定时器中断的 if 语句块中添加相应的 代码实现。换句话说，我们需要在定时器中断发生时，检查是否有进程使用了 sigalarm 系统调用，并 根据其设定的周期数更新已消耗的时钟周期数。这样，我们能够实现定时提醒功能并让相关进程在消耗 一定 CPU 时间后执行指定函数，实现如下：

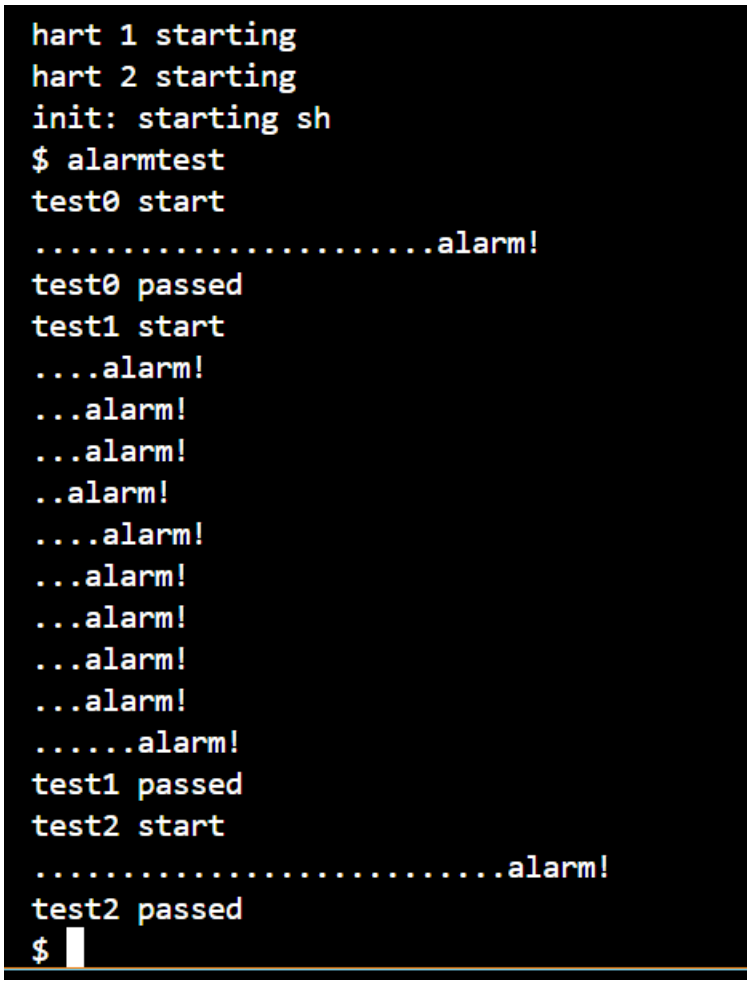
```
if(which_dev == 2)
{
  p->alarmticks += 1;
  if ((p->alarmticks >= p->alarminterval) &&
(p->alarminterval > 0))
 {
    p->alarmticks = 0;
    if (p->sigreturned == 1)
   {
      p->alarmtrapframe = *(p->trapframe);
      p->trapframe->epc = (uint64)p->alarmhandler;
      p->sigreturned = 0;
      usertrapret();
   }
 }
  yield();
```

完成了 sigalarm 触发过程后，还有一个问题需要解决：在传入的 fn 执行完毕后，如何回到进程 的正常执行过程中。由于之前我们保存了进程的执行上下文，所以在 fn 调用 sigreturn() 时，我们 需要在内核态对应的 sys_sigreturn() 函数中恢复备份的上下文，并返回到用户态。这一实现需要放 在 kernel/sysproc.c 文件中。换句话说，我们需要在 sys_sigreturn() 函数中完成上下文的恢 复，并确保进程能够回到正常的执行状态,实现如下：

```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  p->sigreturned = 1;
  *(p->trapframe) = p->alarmtrapframe;
  usertrapret();
  return 0;
}
```

3.3 实验结果

编译运行 xv6 ， 然后执行 alarmtest ， 得到结果如下，可知测试通过：



3.4 实验注意事项

在进程的数据结构中保存 sigalarm 的参数，以便在定时中断中进行更新。
在定时中断处理中，对每个进程的已消耗的 tick 数进行更新，然后检查是否满足触发条件。
当触发条件满足时，调用中断处理过程 fn ，需要确保正确的上下文切换和保存。
在中断处理过程 fn 执行完毕后，需要恢复进程的上下文并返回到正常执行状态。

4. 完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-traps
git fetch
git checkout traps
make clean
make grade
```

结果如下，可见测试全部通过：

```
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (2.0s)
== Test running alarmtest ==
$ make qemu-gdb
(3.2s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (181.4s)
== Test time ==
time: OK
Score: 85/85
os001@ubuntu:~/xv6-labs-2021-traps$
```

5. 遇到的问题及解决方法

在处理中断时，偶尔会涉及内存分配和释放，可能会出现内存管理问题，导致内存泄漏或访问越 界。这时候则需要注意在处理内存分配和释放时，确保遵循正确的内存管理原则，及时释放不再使用的 内存，避免访问越界。使用工具检查内存分配和释放的正确性。

6. 实验收获

通过本次实验，我对 RISC-V 的中断机制有了一定的了解，其是用于处理硬件中断和异常的系统级特 性。它提供了一种机制，允许处理器在执行程序时，根据外部或内部事件的发生，中断正常的执行流 程，以处理这些事件。以下是 RISC-V 的中断机制的简要说明:

1. 中断源: 中断可以由多种源触发，包括硬件设备（如时钟、外部 IO 设备）、异常情况（如缺页 异常、非法指令异常）以及软件触发（通过软件中断指令）。

2. 中断向量表: RISC-V 使用中断向量表来存储不同中断的处理程序入口地址。这个表通常是在内 存的一个特定位置，当发生中断时，处理器根据中断号从中断向量表中查找相应的处理程序入 口地址。

3. 中断处理过程: 当中断发生时，处理器会暂停当前指令的执行，并且保存当前的上下文信息（包括寄 存器状态等）。 处理器根据中断号从中断向量表中获取中断处理程序的入口地址。 执行中断处理程序，这可能涉及到一系列的操作，如保存当前上下文、处理中断事 件、进行状态切换等。 中断处理程序完成后，恢复之前保存的上下文信息，并继续执行中断发生前的程序指 令。

4. 中断优先级: RISC-V 支持多种中断优先级，处理器会根据中断优先级来确定哪个中断应该被处 理。更高优先级的中断会打断正在处理的较低优先级中断。

5. 中断控制寄存器: 处理器通常有一些专门的控制寄存器用于控制中断的使能、屏蔽和优先级等 设置。这些寄存器的设置可以影响中断的触发和处理。

七、lab5：Copy-on-Write Fork for xv6

1. Implement copy-on write

1.1 实验要求

在这个实验中，我的任务是在 xv6 内核中实现写时复制（Copy-On-Write）的进程创建机制，即确保 实现写时复制的机制，以确保在进行进程创建时，父子进程之间共享相同的物理内存页，只有在某个进 程尝试修改这些页时，才会进行实际的复制操作。

1.2 实验内容、

```
cd xv6-labs-2021-cow
git fetch
git checkout cow
make clean
```

为了实现写时复制(Copy-On-Write)机制，我们需要对物理内存分配器进行修改。在 COW 中，一 个物理页面可能被多个页表引用，因此我们需要引入一个数据结构来跟踪页面的引用计数。为了做到这 一点，在 kalloc.c 中我们添加了以下变量:

```
struct spinlock reflock;
uint8 referencecount[PHYSTOP/PGSIZE];
```

然后在初始化内存管理器的 kinit 中添加初始化引用计数锁的语句，并更改释放物理页面的代码，使 得每次释放仅将引用计数减一，直到引用计数为 0 后才真正释放页面，实现如下:

```
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  initlock(&reflock, "ref");
  freerange(end, (void*)PHYSTOP);
}

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
  {
    acquire(&reflock);
    referencecount[(uint64)p / PGSIZE] = 0;
    release(&reflock);

    kfree(p);
  }
}
```

接下来要将父进程的页面映射给子进程，并给父进程和子进程的页表设为只读。

首先修改 vm.c 中的 uvmcopy 使用 mappages 只映射页面、增加引用计数并修改权限为只读，而不 分配页面，这里使用了分页机制中为软件使用保留的第 8 位作为标志位，若其为 1 ，则说明其为 COW 页:

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  // 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");
    *pte &= ~(PTE_W);
    *pte |= PTE_COW;
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    // if((mem = kalloc()) == 0)
    //   goto err;
    // memmove(mem, (char*)pa, PGSIZE);
    // if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
    //   kfree(mem);
    //   goto err;
    // }
    if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
      goto err;
    }
    // printf("uvmcopy: lazy copying va=%p pa=%p\n",i,pa);
    acquire(&reflock);
    referencecount[PGROUNDUP((uint64)pa)/PGSIZE]++;
    release(&reflock);
  }
  return 0;

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

通过引入 uvmcopy 操作，我们引入了对页面的引用计数。在删除页面映射的过程 uvmunmap 中，我 们也需要添加相应的操作来保持引用计数的正确性。这样，我们就完成了写时复制（COW）机制的前半 部分。如果父进程和子进程在 fork 后都没有尝试写入内存，那么一切都会平稳进行。然而，如果其中 一个进程尝试写入内存，就会触发页面权限错误，从而引发中断并最终调用 usertrap() 函数。这个机 制可以确保写入操作不会对其他进程产生影响，从而保证进程之间的数据隔离性。

```
extern struct spinlock reflock;
extern uint8 referencecount[PHYSTOP/PGSIZE];
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");
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    acquire(&reflock);
    referencecount[PGROUNDUP((PTE2PA(*pte)))/PGSIZE]--;
    if(do_free && referencecount[PGROUNDUP((PTE2PA(*pte)))/PGSIZE] < 1){
    // if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    release(&reflock);
    *pte = 0;
  }
}
```

当发生写页面权限引起的错误时，会导致 SCAUSE 寄存器被设置为 12 或 15，而 STVAL 寄存器中将存 储导致违反权限的虚拟内存地址。通过使用 riscv.h 中提供的读取寄存器函数，我们可以获取这两个寄存 器的内容。然后，我们可以进行适当的判断，如果发现违例访问

是由 COW 机制引起的，我们可以进行以 下后续工作：分

```
extern uint8 referencecount[PHYSTOP/PGSIZE];

void

usertrap(void)

{

  int which_dev = 0;



  if((r_sstatus() & SSTATUS_SPP) != 0)

    panic("usertrap: not from user mode");



  // send interrupts and exceptions to kerneltrap(),

  // since we're now in the kernel.

  w_stvec((uint64)kernelvec);



  struct proc *p = myproc();



  // save user program counter.

  p->trapframe->epc = r_sepc();



  if(r_scause() == 8){

    // system call



    if(p->killed)

      exit(-1);



    // sepc points to the ecall instruction,

    // but we want to return to the next instruction.

    p->trapframe->epc += 4;



    // an interrupt will change sstatus &c registers,

    // so don't enable until done with those

registers.

    intr_on();

syscall();

  } else if((which_dev = devintr()) != 0){

    // ok

  } else if (r_scause() == 12 || r_scause() == 15){

    // deal with cow pages

    pte_t *pte;

    uint64 pa, va;

    // uint64 va;

    uint flags;

    char *mem;

    va = r_stval();

    if(va >= MAXVA)

    {

      p->killed = 1;

      exit(-1);

    }



    if((pte = walk(p->pagetable, va, 0)) == 0)

    {

      // panic("cowhandler: pte should exist");

      p->killed = 1;

      exit(-1);

    }

    if((*pte & PTE_V) == 0)

    {

      // panic("cowhandler: page not present");

      p->killed = 1;

      exit(-1);

    }

    if((*pte & PTE_COW) == 0)

    {

      // panic("cowhandler: page not cow");

      p->killed = 1;

      exit(-1);

    }



    pa = PTE2PA(*pte);

    flags = PTE_FLAGS(*pte) | PTE_W;

    flags &= ~(PTE_COW);

    // printf("cowhandler: scause=%d va=%p

pa=%p\n",r_scause(),va,pa);

    if((mem = kalloc()) == 0)

    {

      // printf("%d: cowhandler: kalloc failed, killed the

process\n",p->pid);

      p->killed = 1;

      exit(-1);

    }

    // printf("%d: cowhandler: kalloc succeeded with
```

面、修改页面权限和标志等。如果判断不是 COW 引起的 违例访问，我们需要终止进程并进行相应的清理工作。在 trap.c 文件中，我实现了类似的逻辑来处理 这种情况，以确保系统的稳定性和数据完整性。

需要在修改 copyout() 函数以适应 COW 机制。copyout() 函数在内核态修改用户态的页面内存， 不会引发从用户态来的权限违例。因此，在处理 COW 页面时需要进行特殊处理。具体查看 vm.c 中的 copyout() 函数实现，以下是相关代码：

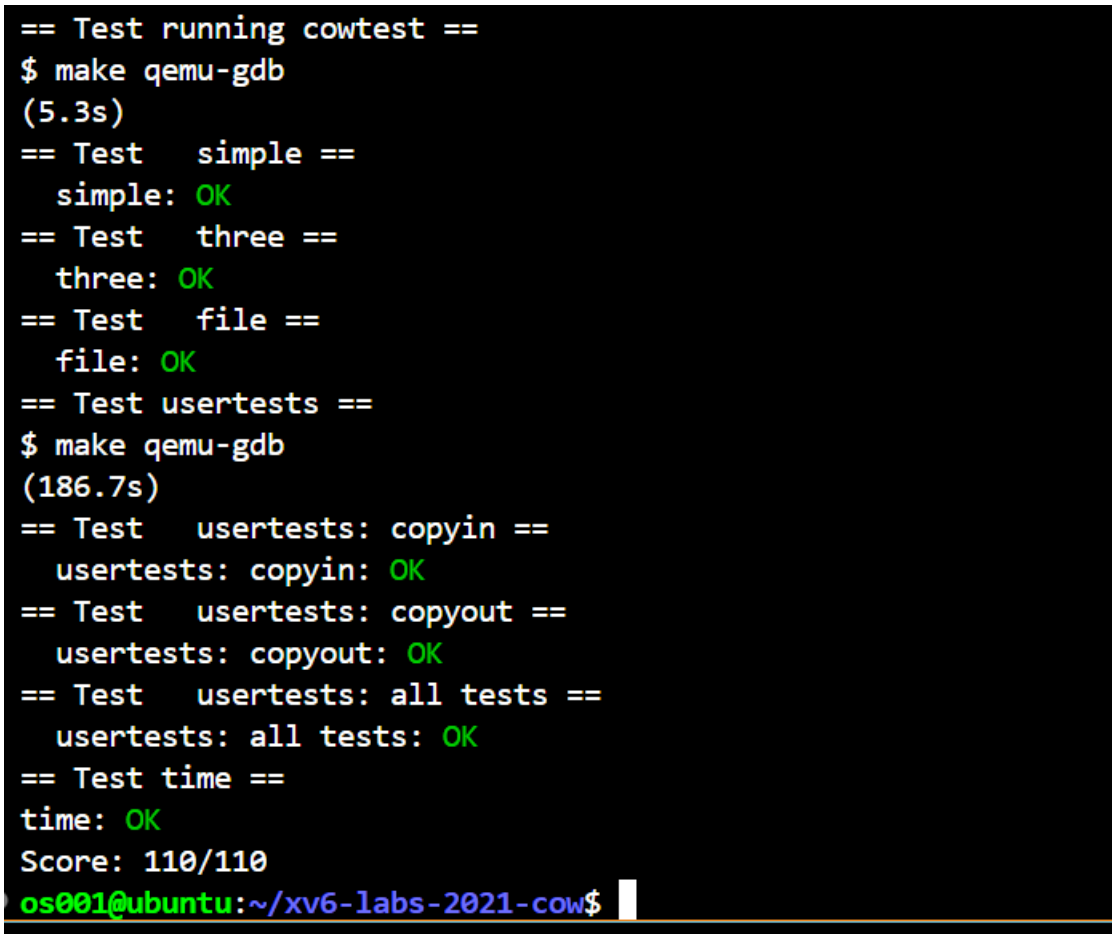
```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;
  pte_t *pte;
  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if(dstva >= MAXVA)
      return -1;
    pte = walk(pagetable, va0, 0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    if (*pte & PTE_COW)
    {
      uint flags;
      char *mem;
      flags = PTE_FLAGS(*pte) | PTE_W;
      flags &= ~(PTE_COW);
      if((mem = kalloc()) == 0)
      {
        // printf("copyout: kalloc failed, killed the process\n");
        return -1;
      }
      memmove(mem, (char*)pa0, PGSIZE);
      uvmunmap(pagetable, va0, 1, 1);
      if(mappages(pagetable, va0, PGSIZE, (uint64)mem, flags) != 0)
      {
        kfree(mem);
        panic("copyout: mappages failed");
      }
```

2.完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-cow
git fetch
git checkout cow
make grade
```

结果如下，可见测试全部通过：



3.遇到的问题及解决方法

1.如何标记页面为 COW 页面，使得父子进程可以共享页面但保证数据不被破坏？
通过在页表项中设置特定的标志位，例如在 PTE_FLAGS 中添加一个标志位表示页面是 COW 页面。这样父子 进程可以共享页面，但在写入时会触发中断，从而进行相应处理。

2.如何在中断处理程序中实现 COW 机制，以及如何处理错误情况？
解决方法：在中断处理程序 中，我需要判断中断原因，通过读取 SSTATUS 和 STVAL 寄存器的值来判断是否是 COW 页面引起 的错误。如果是 COW 页面引起的错误，我会进行页面

拷贝、权限修改等操作，然后返回用户态继续执行。如果在处理期间出现错误，我会将进程标记为被杀死状态，然后在中断处理程序 返回后结束进程。

4.实验收获

在我们的 COW fork 实验中，采用了惰性分配页面的方式，而不是直接分配页面，这带来了以下几个优势：

1. 提升速度：初始的 fork 操作无需真正复制内存，因此 fork 的执行速度更快。
2. 资源利用：对于计算密集型任务，子进程可能在无需写内存的情况下执行，因此可以创建更多 的进程，而且不受物理内存的限制。
3. 内存局部性：即使子进程需要写入某些页面，代码段的页面仍然可以是父子进程共享的，这符 合程序的局部性原则。
4. 提高缓存效率：共享的内存页面可以显著提高处理器缓存的效率，从而加快速度。

实际上，COW 机制除了在内存管理中使用外，在文件系统、操作系统以及其他软件中也有广泛的应 用。惰性机制的核心思想是根据需求进行分配，从而避免资源的浪费。惰性机制不仅在内存管理中使 用，还可以在文件系统中的延迟加载、编程语言中的惰性求值等方面看到。

然而，在一些性能要求较高的场景中，仅仅避免浪费还不够，还需要系统能够提前准备所需资源。 因此，现代软件系统中不仅包括惰性机制，还引入了一些基于启发式算法的预测机制，以便在预测成功 时获得性能提升，但预测失败时可能会导致资源浪费和性能损失。

在操作系统层面，由于预测失败的代价较大，更多的优化仍然是从保守的避免资源浪费的角度出 发，确保系统稳定和高效运行。

八、 lab6： Multithreading

1. Uthread: switching between threads

1.1 实验要求

在这个实验中，我们的任务是完成一个用户态线程库的功能。xv6 提供了一些基础的代码文件，分别 是 user/uthread.c 和 user/uthread_switch.S 。我们需要在 user/uthread.c 中实现 thread_create() 和 thread_schedule() 函数， 同时在 user/uthread_switch.S 中实现 thread_switch() 函数，用于实现线程的上下文切换机制。这些函数的完成将允许我们在用户态创建和 管理线程，实现多线程的功能。

1.3 实验内容

```
cd xv6-labs-2021-thread
git fetch
git checkout thread
make clean
```

在线程的数据结构中，使用了一个字节数组作为线程的栈，并且使用一个整数来表示线程的状态。 然而，为了有效地保存和恢复线程的执行状态，我们还需要添加一个数据结构来存储每个线程的上下文 信息。这类似于在操作系统内核中关于进程上下文的处理方式，通过保存寄存器的值、程序计数器等信 息，以便在切换线程时能够正确地恢复线程的执行状态。因此，我们需要参考操作系统内核中关于处理 进程上下文的代码，在线程数据结构中增加相应的上下文信息来实现这一功能。

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

参考 xv6 实验手册的提示，除了 sp 、 s0 和 ra 寄存器，我们只需要保存 callee-saved 寄存 器，因此构造了上面的 struct context 结构体。有了该结构体，我们仿照 kernel/trampoline.S 的 结构， 按照 struct context 各项在内存中的位置， 在 user/uthread_switch.S 中加入如下的代码：

```
        .text

        /*
         * save the old thread's registers,
         * restore the new thread's registers.
         */
        .globl thread_switch
thread_switch:
        /* YOUR CODE HERE */
        /* Save registers */
        # addi ra, ra, 8; # t1 = t1 + imm


    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)
```

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

在进行线程的上下文切换后，接下来需要完成线程的创建和调度部分。在线程创建时，我们需要设 置线程的栈空间，并确保在线程被调度运行时，程序计数器（pc）能正确跳转到相应的位置。在前述的 thread_switch 函数中，当保存了第一个线程的上下文后，会加载第二个线程的上下文，并从加载的返 回地址（ra）开始执行。因此，在创建线程时，我们只需要将返回地址（ra）设置为要执行线程的函数 地址即可。因此， thread_create() 的实现如下：

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


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

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

```
void
thread_schedule(void)
{
  struct thread *t, *next_thread;


  /* Find another runnable thread. */
  next_thread = 0;
  t = current_thread + 1;
  for(int i = 0; i < MAX_THREAD; i++){
    if(t >= all_thread + MAX_THREAD)
      t = all_thread;
    if(t->state == RUNNABLE) {
      next_thread = t;
      break;
    }
    t = t + 1;
  }
  if (next_thread == 0) {
    printf("thread_schedule: no runnable threads\n");
    exit(-1);
  }
```

```
if (current_thread != next_thread) {         /* switch
threads?  */
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    // printf("ra: %p\n",t->context.ra);
    // printf("sp: %p\n",t->context.sp);
    // printf("ra after: %p\n",current_thread->context.ra);
    // printf("sp after: %p\n",current_thread->context.sp);
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    // t->state = RUNNABLE; 这里不用把线程状态改成 RUNNABLE，否则就
会让 0 线程也能跑，导致出错；yield()已经改过了
    thread_switch((uint64)&t->context,
(uint64)&current_thread->context);


  } else
    next_thread = 0;
}
```

1.3 实验结果

实验完成后，编译 xv6 系统，输入 uthread，可以看到结果符合预期：

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

1.4 实验注意事项

需要注意区分在本实验中的不同概念，即进程、线程和协程。在 xv6 中，一个进程只包含一个线程，因此在本实验中进程和线程的区分并不显著。而在本实验中，用户线程的切换并不涉及进入内核态，而是通过让出 CPU 来实现的，这与协程的概念相似。换句话说，本实验中的用户线程切换更类似于协程的行为。

2. Using threads

2.1 实验要求

在本次实验中，需要使用线程和锁来探索并行编程，以实现一个哈希表。

2.2 实验内容

在本次实验中，我们需要修改名为 ph.c 的文件，以便在多线程同时读写哈希表时确保正确的结 果。在并发情况下，由于多个线程竞争访问共享的数据，可能会导致数据被不正确地写入到哈希表中， 因此我们需要找到方法解决这个问题。一种常见的解决方案是引入互斥锁，这样只有获得锁的线程才能 修改共享数据。在查阅了有关 pthread 线程锁的文档和资料后，我们采取了以下步骤来实现：

1. 我们首先在代码中定义了一个互斥锁，用于保护哈希表。

```
pthread_mutex_t lock;
```

2.在启动线程之前，我们对互斥锁进行初始化，以确保线程可以正确地使用它。

```
pthread_mutex_init(&lock, NULL);
```

3. 在执行写入操作之前，我们获取互斥锁，这将确保只有一个线程能够同时修改哈希表。在写入 操作完成后，我们释放互斥锁，以便其他线程可以继续访问哈希表。

```
static
void put(int key, int value)
{

  int i = key % NBUCKET;

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

}
```

2.3 实验结果

实验完成后，运行 make ph 编译 notxv6/ph.c ，运行 ./ph 2 ，则发现没有读出的数据缺失，如下 图所示：

```
os001@ubuntu:~/xv6-labs-2021-thread$ ./ph 2
100000 puts, 2.495 seconds, 40075 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 4.979 seconds, 40173 gets/second
os001@ubuntu:~/xv6-labs-2021-thread$
```

2.4 实验注意事项

在处理哈希表中的读取（get）操作时，并不涉及对数据的并发修改或删除操作，因此在理论 上可以不加锁。然而，为了确保整个系统的正确性，我们还是在读取操作中引入了互斥锁，以 避免可能的竞争条件。通过在读取操作中加锁，我们能够保证在任何情况下，对于正在被多个 线程访问的共享数据，始终只有一个线程能够进行读取操作。这样可以避

免不必要的数据不一 致性和错误。

3. Barrier

3.1 实验要求

在本实验中，需要实现一个屏障（Barrier）：即在应用程序中的一个特定点，所有参与的线程必须 等待，直到所有其他参与的线程也达到了该点为止。我们将使用 pthread 条件变量，这是一种类似于 xv6 中的睡眠（sleep）和唤醒（wakeup）机制的序列协调技术。

3.2 实验内容

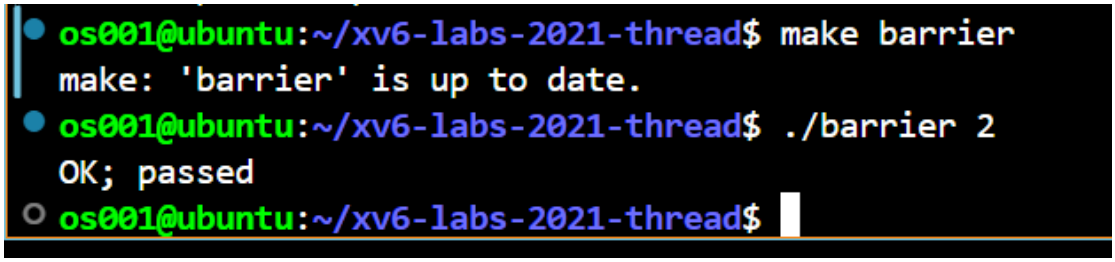
为了实现线程屏障，我们需要使用一些关键的同步机制和变量。首先，我们需要一个互斥锁来确保 多个线程之间对关键数据的访问是互斥的，从而避免竞争条件。其次，我们使用条件变量来在线程之间 进行等待和唤醒的操作，以便在达到线程屏障时进行同步。还有两个整数变量，一个用于记录到达线程 屏障的线程数量，另一个用于记录线程屏障的轮数。

为了实现线程屏障，我们需要使用一些关键的同步机制和变量。首先，我们需要一个互斥锁来确保 多个线程之间对关键数据的访问是互斥的，从而避免竞争条件。其次，我们使用条件变量来在线程之间 进行等待和唤醒的操作，以便在达到线程屏障时进行同步。还有两个整数变量，一个用于记录到达线程 屏障的线程数量，另一个用于记录线程屏障的轮数。在初始化操作 barrier_init() 中，我们初始化互斥锁、条件变量以及记录线程数量的变量 nthread 。然后，在某个线程到达线程屏障点时，首先需要获取互斥锁，以确保对共享变量的访问是互 斥的。在获取锁后，我们修改 nthread 的值，表示已经到达屏障的线程数量增加了。接着，我们检查 是否所有线程都已到达屏障，即 nthread 是否等于预定的数量。如果相等，就说明所有线程已到达屏 障，此时我们将 nthread 清零，增加轮数，并唤醒所有等待中的线程，使它们继续执行。最后，不要 忘记在 barrier() 函数中释放互斥锁，以便其他线程可以继续执行。具体实现如下：

```
static void

barrier()

{

  pthread_mutex_lock(&bstate.barrier_mutex);

  bstate.nthread ++;

  if (bstate.nthread == nthread)

  {

    bstate.round++;

    bstate.nthread = 0;

    pthread_cond_broadcast(&bstate.barrier_cond);

  } else {

    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);

  }

  pthread_mutex_unlock(&bstate.barrier_mutex);

}
```

3.3 实验结果

实验完成后，运行 make barrier 编译 notxv6/barrier.c ，运行 ./barrier 2 ，如下图所示，可 见通过：



3.4 实验注意事项

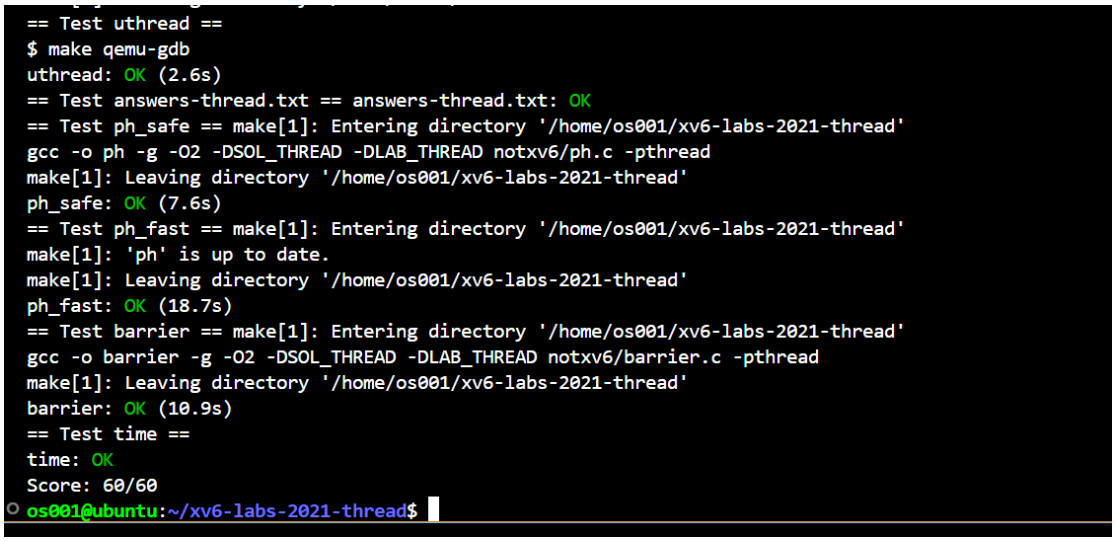
条件变量是一种用于多线程同步的机制，它依赖于共享的全局变量来协调线程间的操作。条件 变量通常涉及两个主要动作：一个线程等待条件变量满足特定条件时才继续执行，而另一个线 程则负责使得这个条件成立。为了避免并发竞争的问题，条件变量的使用通常与互斥锁相结合。

4. 完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-thread
git fetch
git checkout thread
make clean
make grade
```

结果如下，可见测试全部通过：



九、 lab7： networking

1. Your Job

1.1 实验要求

驱动程序是操作系统的重要组成部分，负责沟通操作系统和实际工作的硬件。在本次实验中，我们会在 xv6 中为一个现实中广泛使用的硬件：Intel E1000 网卡，编写驱动程序，从而体会编写驱动程序的一般步骤。

1.2 实验内容

```
cd xv6-labs-2021-net
git fetch
git checkout net
make clean
```

根据 Intel 提供的关于 E1000 网卡的驱动的开发者文档 Intel E1000 Software Developer's Manual 和 xv6 的实验手册，我们主要需要实现 kernel/e1000.c 中的两个函数：用于发送数据包的 e1000_transmit()和用于接收数据包的 e1000_recv()。

这里我们关注两个环形缓冲区： tx_ring 和 rx_ring 。根据 Intel E1000 Software Developer's Manual 上的描述，我们只需要将需要发送的数据包放入环形缓冲区中，设置好对应的参 数并更新管理缓冲区的寄存器，即可视为完成了数据包的发送。此后网卡的硬件会自动在合适的时间将 我们放入的数据包按照配置发送出去，在 kernel/e1000.c 中的实现如下：

```
int
e1000_transmit(struct mbuf *m)
{
  acquire(&e1000_lock);
  // printf("e1000_transmit: called mbuf=%p\n",m);
  uint32 idx = regs[E1000_TDT];
  // ask the E1000 for the TX ring index
  // at which it's expecting the next packet,
  // by reading the E1000_TDT control register.
  if (tx_ring[idx].status != E1000_TXD_STAT_DD)
  {
    // check if the the ring is overflowing
    // If E1000_TXD_STAT_DD is not set in the descriptor indexed by E1000_TDT,
    // the E1000 hasn't finished the corresponding previous transmission request,
    // so return an error.
    printf("e1000_transmit: tx queue full\n");
    __sync_synchronize();
    // https://www.cnblogs.com/weijunji/p/xv6-study-16.html 的作者这里有
__sync_synchronize(); 但本猫还没搞明白其作用。
    release(&e1000_lock);

    return -1;
  } else {
    // Otherwise, use mbuffree() to free the last mbuf
    // that was transmitted from that descriptor (if there was one).
    if (tx_mbufs[idx] != 0)
    {
      // printf("e1000_transmit: freeing old mbuf tx_mbufs[%d]=%p\n",idx,tx_mbufs[idx]);
      mbuffree(tx_mbufs[idx]);
    }
    tx_ring[idx].addr = (uint64) m->head;
    tx_ring[idx].length = (uint16) m->len;
    tx_ring[idx].cso = 0;
    tx_ring[idx].css = 0;
    // Set the necessary cmd flags
    // (look at Section 3.3 in the E1000 manual)
    tx_ring[idx].cmd = E1000_TXD_CMD_RS |
E1000_TXD_CMD_EOP;
    // and stash away a pointer to the mbuf for later
freeing.
    tx_mbufs[idx] = m;
    // Finally, update the ring position
    // by adding one to E1000_TDT modulo TX_RING_SIZE.
    regs[E1000_TDT] = (regs[E1000_TDT] + 1) %
TX_RING_SIZE;
    // printf("e1000_transmit: package added to tx
queue %d\n",idx);
  }
  __sync_synchronize();
  // https://www.cnblogs.com/weijunji/p/xv6-study-
16.html 的作者这里有__sync_synchronize(); 但本猫还没搞明
白其作用。
  release(&e1000_lock);

  return 0;
}
```

对于 e1000_recv() 函数，可以按照类似的方式进行处理，但在将数据包从 rx_ring 拷贝出来 时，需要使用合适的函数。根据 xv6 实验手册的提示，我们可以调用 net.c 中的 net_rx() 函数来完 成这个拷贝过程，而无需自己实现。需要注意的是，当网卡硬件产生一次中断后，我们的中断处理函数 e1000_intr 会执行 regs[E1000_ICR] = 0xffffffff ，以通知网卡我们已经完成了这次中断中所有 数据包的处理。因此，我们的 e1000_recv() 函数需要将 rx_ring 中的所有内容都拷贝出来，具体的 实现如下：

```c
extern void net_rx(struct mbuf *);

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 idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;

  struct rx_desc* dest = &rx_ring[idx];

  // check if a new packet is available

  // by checking for the E1000_RXD_STAT_DD bit in the status portion of the descriptor

  // 查了 https://www.cnblogs.com/weijunji/p/xv6-study-16.html

  // e1000_recv 函数，这里注意一次中断应该把所有到达的数据都处理掉，剩下的按 hints 里面的来就行了

  while (rx_ring[idx].status & E1000_RXD_STAT_DD)

  {

    acquire(&e1000_lock);



    struct mbuf *buf = rx_mbufs[idx];

    mbufput(buf, dest->length);

    if (!(rx_mbufs[idx] = mbufalloc(0)))

      panic("mbuf alloc failed");

    dest->addr = (uint64)rx_mbufs[idx]->head;

    dest->status = 0;

    regs[E1000_RDT] = idx;

    __sync_synchronize();

    // https://www.cnblogs.com/weijunji/p/xv6-study-16.html 的作者这里有__sync_synchronize(); 但本猫还没搞明白其作用。

    // 好像会使得发包收包的顺序有变化

    release(&e1000_lock);



    net_rx(buf);

    idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;

    dest = &rx_ring[idx];

  }



}
```

这里同样需要加锁保证互斥，并使用 __sync_synchronize 保证内存访问顺序，否则会使接收数据 包的顺序产生变化。

1.3 实验结果

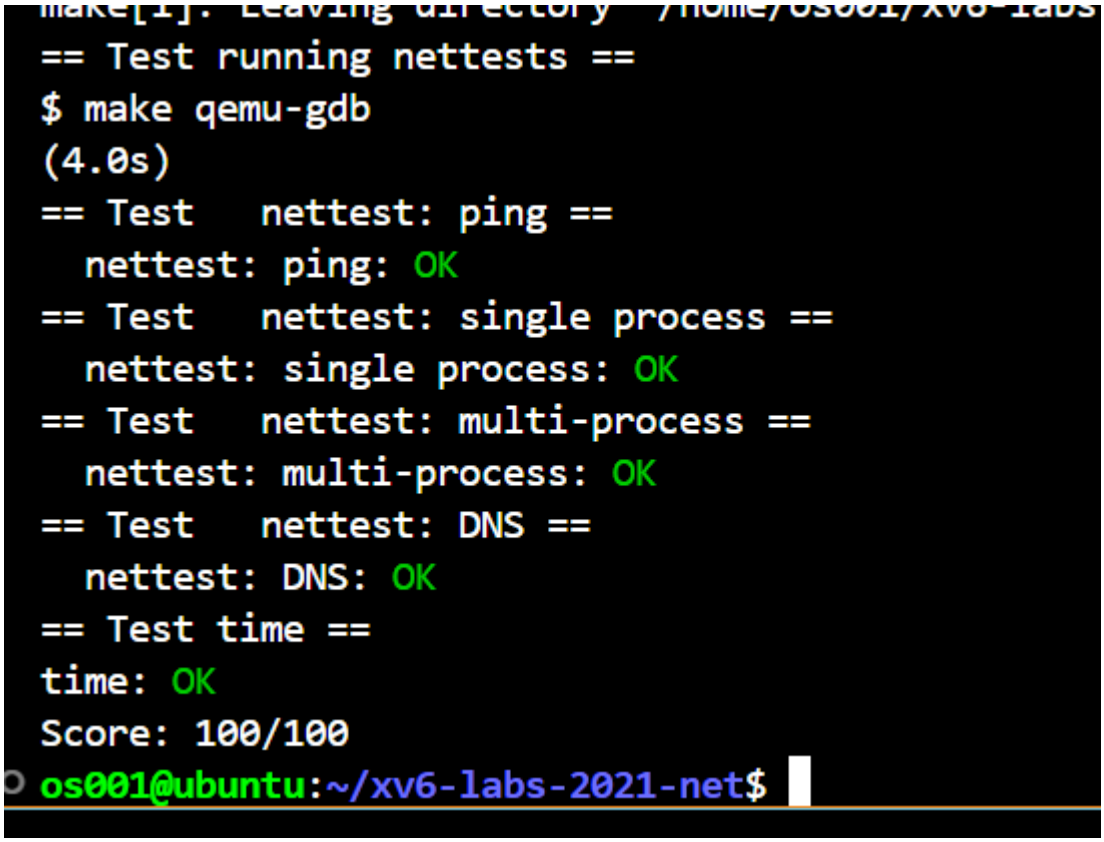
使用 make server 运行 qemu 宿主机的服务器，然后用 make qemu 编译并启动 xv6 ，在 xv6 中运 行 nettests

2.完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-net
git fetch
git checkout net
make clean
make grade
```

结果如下，可见测试全部通过：



3. 遇到的问题及解决方法

实现网络驱动可能涉及多个线程之间的并发访问，包括发送和接收数据的线程。正确处理这些并发 操作以及合适的同步机制是关键，否则可能会出现竞态条件、数据丢失等问题。解决方法： 使用互斥锁、条件变量等同步机制来保护共享资源的访问，确保多个线程之间

的操作是 安全的。遵循良好的并发编程实践，确保对共享资源的访问是原子的，避免出现竞态条件。

4. 实验收获

编写外设驱动程序需要从获取设备文档开始，逐步深入了解设备工作原理，设计合适的数据结构， 并在内核模块中实现初始化和功能代码。最后，通过测试和验证确保驱动程序在真实环境中的可靠性和 稳定性。这种思路也适用于许多其他外设的驱动开发，步骤如下：

1. 获取设备文档和资料： 首先需要获取关于目标外设的技术文档、规范和手册。这些文档将提供 有关设备寄存器、通信协议、操作流程等重要信息。

2. 了解设备工作原理： 深入了解外设的工作原理、通信方式和数据交换过程。理解设备的基本功 能和操作流程对于正确编写驱动至关重要。

3. 设计数据结构： 根据设备的数据交换需求，设计适当的数据结构用于在内核和设备之间传递信 息。这可能包括缓冲区、描述符、控制结构等。

4. 创建内核模块： 在操作系统内核中创建一个模块，该模块将负责与外设交互。在 Linux 中，可 以使用内核模块来实现驱动。

5. 设备初始化： 编写设备初始化代码，设置设备寄存器、配置参数以及初始化数据结构。确保设 备在使用前处于正确的状态。

6. 编写具体功能： 根据设备的功能，实现对应的驱动代码。这可能包括读写操作、中断处理、 DMA 控制等。

7. 测试和验证： 在实际硬件上或仿真环境中测试驱动程序的正确性和稳定性。验证驱动是否按预 期与设备进行交互，处理数据交换和错误情况。

十、 lab8： locks

1. Memory allocator

1.1 实验要求

本实验旨在测试 xv6 操作系统的内存分配器的性能，特别是在多进程并发访问下的情况。在实验 中，你需要完成一个名为 kalloctest 的程序，该程序会创建三个进程，这些进程会不断地扩大和缩小 它们的地址空间，从而导致大量的 kalloc 和 kfree 调用。在这个过程中， kalloc 和 kfree 都会获 取 kmem.lock 锁，这会引发并发的锁竞争。

```
cd xv6-labs-2021-lock
git fetch
git checkout lock
make clean
```

1.2 实验内容

首先，我们修改 kalloc.c 中的数据结构，使单一的空闲链表变为多个空闲链表的数组：

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

然后，我们需要根据上面的分配与释放的逻辑，对 kalloc() 进行修改。首先需要利用 cpuid() 读取特 殊寄存器以获得当前运行的 CPU 的编号，该编号对应着一个空闲链表，在获取该编号时需要关闭中断， 以免发生不测。成功获取 CPU 编号后，我们需要获取对应的空闲链表的锁，然后试图分配页面，若该 CPU 的空闲链表中没有空闲页面，则到其它 CPU 中借用一个最后若分配成功，则返回获取的页面，笔 者完整的实现如下：

```
void *
kalloc(void)
{
  struct run *r;
  push_off();
  int c = cpuid();
  pop_off();

  acquire(&kmem[c].lock);
  r = kmem[c].freelist;
```

```
acquire(&kmem[i].lock);
    r = kmem[i].freelist;
    if(r)
    {
      kmem[i].freelist = r->next;
      release(&kmem[i].lock);
      break;
    } else {
      release(&kmem[i].lock);
    }
```

对分配内存的 kalloc() 修改完成后，我们还需要修改对应的用于释放内存的 kfree()，直接将页 面加入到当前的 CPU 空闲链表中，实现如下：

```
void
kfree(void *pa)
{
  struct run *r;
  push_off();
  int c = cpuid();
  pop_off();

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

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

  r = (struct run*)pa;

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

最后不要忘记初始化内存分配器的 kinit()，其需要初始化每个 CPU 的空闲链表和锁。最后不要忘记 初始化内存分配器的 kinit()，其需要初始化每个 CPU 的空闲链表和锁：

```
void
kinit()
{
  for (int i = 0; i<NCPU; i++)
  {
    // char buf[9];
    // snprintf(buf, 6, "kmem_%d", i);
    // initlock(&kmem[i].lock, buf);
    initlock(&kmem[i].lock, "kmem");
  }

  freerange(end, (void*)PHYSTOP);
}
```

1.3 实验结果

编译运行 xv6 ， 然后执行 kalloctest ， 得到结果如下， 可知测试通过:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 173511
lock: kmem: #test-and-set 0 #acquire() 100119
lock: kmem: #test-and-set 0 #acquire() 159422
lock: bcache: #test-and-set 0 #acquire() 1248
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 1993118 #acquire() 114
lock: proc: #test-and-set 1741575 #acquire() 581005
lock: proc: #test-and-set 1514220 #acquire() 581005
lock: proc: #test-and-set 1408306 #acquire() 581005
lock: proc: #test-and-set 1302373 #acquire() 581005
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$
```

1.4 实验注意事项

根据修改后的数据结构, 当出现报错时, 我们需要定位到相关的函数并进行适当的修改。特别是在 对内存进行释放时， 需要确保将页面加入到当前 CPU 的空闲链表中， 从而实现借取后不归还的操作。这 可以通过在 kfree() 函数中进行相应的调整来实现。

2. Buffer cache

2.1 实验要求

需要处理并减少对 bcache.lock 的争用。你可以通过修改相关的代码，使用适当的同步机制来减 少锁的争用， 从而提高文件系统的性能。通过解决 bcache.lock 的争用问题， 需要优化多进程读取文 件的效率， 并改善文件系统的性能表现。

2.2 实验内容

依照上面的思想， 我们首先改造 bcache 的数据结构， 将单个锁改成多个锁， 并将缓存块分组:

```
struct {
  struct spinlock lock[NBUCKET];
  struct buf buf[NBUF];
  struct buf head[NBUCKET];
} bcache;
```

然后，修改对应的一些头文件中的定义：

```
param.h #define NBUF

defs.h int can_lock(int, int);
buf.h
struct buf {
 int valid;
 int disk;
uint dev;
uint blockno;
struct sleeplock lock;
uint refcnt;
struct buf *prev;
struct buf *next;
uchar data[BSIZE];
uint time; };
```

然后修改对应的 bcache 初始化函数 binit()，使之与修改后的数据结构相适应：

```
void

binit(void)

{

  struct buf *b;



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

  {

    initlock(&bcache.lock[i], "bcache");

  }

  // Create linked list of buffers

  // bcache.head[0].prev = &bcache.head[0];

  bcache.head[0].next = &bcache.buf[0];

  for(b = bcache.buf; b < bcache.buf+NBUF-1; b++){

    b->next = b+1;

    initsleeplock(&b->lock, "buffer");

  }

  initsleeplock(&b->lock, "buffer");

}
```

2.3 实验结果

编译运行 xv6，然后执行 bcachetest，得到结果如下，可知测试通过：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 32955
lock: kmem: #test-and-set 0 #acquire() 55
lock: kmem: #test-and-set 0 #acquire() 90
lock: bcache: #test-and-set 0 #acquire() 2128
lock: bcache: #test-and-set 0 #acquire() 4145
lock: bcache: #test-and-set 0 #acquire() 2292
lock: bcache: #test-and-set 0 #acquire() 4290
lock: bcache: #test-and-set 0 #acquire() 4340
lock: bcache: #test-and-set 0 #acquire() 6336
lock: bcache: #test-and-set 0 #acquire() 6761
lock: bcache: #test-and-set 0 #acquire() 9021
lock: bcache: #test-and-set 0 #acquire() 6193
lock: bcache: #test-and-set 0 #acquire() 6201
lock: bcache: #test-and-set 0 #acquire() 6195
lock: bcache: #test-and-set 0 #acquire() 4137
lock: bcache: #test-and-set 0 #acquire() 4137
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 14244326 #acquire() 1251
lock: proc: #test-and-set 1861059 #acquire() 294497
lock: proc: #test-and-set 1243688 #acquire() 294498
lock: proc: #test-and-set 1190282 #acquire() 294498
lock: proc: #test-and-set 1175652 #acquire() 294498
tot= 0
test0: OK
start test1
test1 OK
$
```

2.4 实验注意事项

在修改 param.h 文件中的 \#define FSSIZE 时，将其从默认的 1000 修改为 10000。这是 因为在运行测试 writebig 时，可能会出现报错信息 "panic: balloc: out of blocks"，这是因 为测试要求分配更多的磁盘块空间。通过将 FSSIZE 设置为更大的值，我们可以确保系统有足 够的磁盘块来满足测试的需求，避免出现该报错。这个修改将允许测试 writebig 成功运 行，确保文件系统正常工作。

3. 完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-lock
git fetch
git checkout lock
make clean
make grade
```

结果如下，可见测试全部通过：

```
== Test running kalloctest ==
$ make qemu-gdb
(57.7s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (10.2s)
== Test running bcachetest ==
$ make qemu-gdb
(45.8s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (203.2s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 70/70
os001@ubuntu:~/xv6-labs-2021-lock$
```

4. 遇到的问题及解决方法

死锁是指多个线程因为相互等待对方释放资源而陷入无限等待的状态，导致程序无法继续执行。

解决方法：避免循环等待 确保线程按照相同的顺序请求锁，避免出现循环等待的情况，从而预防死锁。

5. 实验收获

降低锁开销的方法在上述两个实验中得到了清晰的展示，以下是对这些方法的总结:

1. 重复资源分配: 在 kalloc 中，通过多次分配资源来减少进程等待的概率。 在多处理器的环境下，为了降低争用，可以将资源分为多份，每个处理器使用不同的 资源池。这减少了等待锁的概率，从而提高并发性能。

2. 细粒度加锁: 在 bcache 中，通过使用细粒度的锁管理，减少资源加锁冲突的概率。对于多个资源或数据结构，可以采用不同的锁来保护它们，使得并发访问时只有部分 资源被锁定，减少了阻塞其他线程的可能性。 使用更小的锁粒度，使得线程只需要竞争更少的锁，从而减少了竞争和等待。

这两种方法都旨在提高多线程环境下的程序性能和效率，避免由于加锁引起的性能瓶颈。通过资源 的重复设置和细粒度的加锁管理，可以最大程度地减少线程之间的竞争和等待，从而提升并发程序的执 行效率。

十一、lab9：file system

1. Large files

1.1 实验要求

在这个实验中，将增加 xv6 文件的最大大小限制。目前 xv6 文件的最大限制为 268 个数据块，或者 268 * BSIZE 字节（xv6 中 BSIZE 为 1024）。这个限制是因为 xv6 的 inode 包含了 12 个“直接”块号和 一个“间接单级”块号，而间接单级块又指向一个块，其中包含了最多 256 个额外的块号，总共可以达到 12+256=268 个块。在这个实验中，我们需要修改 xv6 的文件系统以支持更大的文件大小。

1.2 实验内容

```
cd xv6-labs-2021-fs
git fetch
git checkout fs
make clean
```

xv6 的实验手册中推荐的方案是：使用三级索引，共有 11 个直接索引，1 个间接索引块和 1 个二 级间接索引块，故总共支持文件大小为 11 + 256 + 256 × 256 = 65803 块。

首先我们修改 fs.h 中的一些定义，使之符合三级索引的要求：

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

由于文件的读写需要使用 bmap() 来找到需要操作的文件块，故而我们修改 fs.c 中用于找到文件块 的 bmap() ，使之能够支持三级索引。一个简单的思路是通过访问文件的位置来判断该位置是位于几级 索引中，这样可以复用原先的大部分代码，而只需要实现二级间接索引的部分。

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

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

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

```
if(bn < NDOUBLEINDIRECT){
    // load doubly-indirect block, allocating if
necessary.
    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;
    if((addr = a[bn%NINDIRECT]) == 0){
      a[bn%NINDIRECT] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

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

1.3 实验结果

编译运行 xv6 ， 然后执行 bigfile ， 得到结果如下，可知测试通过：

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

1.4 实验注意事项

在处理资源释放时需要特别注意，确保在释放资源后不会引起内存泄漏或者其他问题。

在实现二级间接索引时，实际上需要进行两次索引操作，一次用于找到一级索引块，然后再在 一级索引块中找到二级索引块，最终在二级索引块中找到数据块。在进行这些索引

操作时，需 要正确处理索引的下标，确保数据的正确获取。

2. Symbolic links

2.1 实验要求

在这个实验中，我们需要在 xv6 中添加符号链接（Symbolic Links）。符号链接是通过路径名引用 链接的文件；当打开符号链接时，内核会根据链接跟踪到所指向的文件。符号链接类似于硬链接（Hard Links），但硬链接只能指向同一磁盘上的文件，而符号链接可以跨越不同的磁盘设备。

2.2 实验内容

首先，按照通常的方法在 user/usys.pl 和 user/user.h 中加入该系统调用的入口，然后在 kernel/sysfile.c 中加入空的 sys_symlink 。修改头文件 kernel/stat.h ， 增加一个文件类型：

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

然后在头文件 kernel/fcntl.h 加入一个标志位，供 open 系统调用使用：

```
#define O_NOFOLLOW 0x010
```

然后，在 Makefile 中添加用户程序 symlinktest ，并确保它能够通过编译。经过这些准备工作， 现在我们可以开始真正着手实现 symlink(target, path) 系统调用以及相应的 open 系统调用。在 sys_symlink 中，你可以仿照 sys_mknod 的结构，创建一个节点并将符号链接的数据写入其中。通过 这样的步骤，你可以为 xv6 添加对符号链接的支持，并实现对应的系统调用，使其能够正确地处理符号 链接文件。

```
uint64
sys_link(void)
{
  char name[DIRSIZ], new[MAXPATH], old[MAXPATH];
  struct inode *dp, *ip;

  if(argstr(0, old, MAXPATH) < 0 || argstr(1, new, MAXPATH) < 0)
    return -1;

  begin_op();
  if((ip = namei(old)) == 0){
    end_op();
    return -1;
  }

  ilock(ip);
  if(ip->type == T_DIR){
    iunlockput(ip);
    end_op();
    return -1;
  }

  ip->nlink++;
  iupdate(ip);
  iunlock(ip);
```

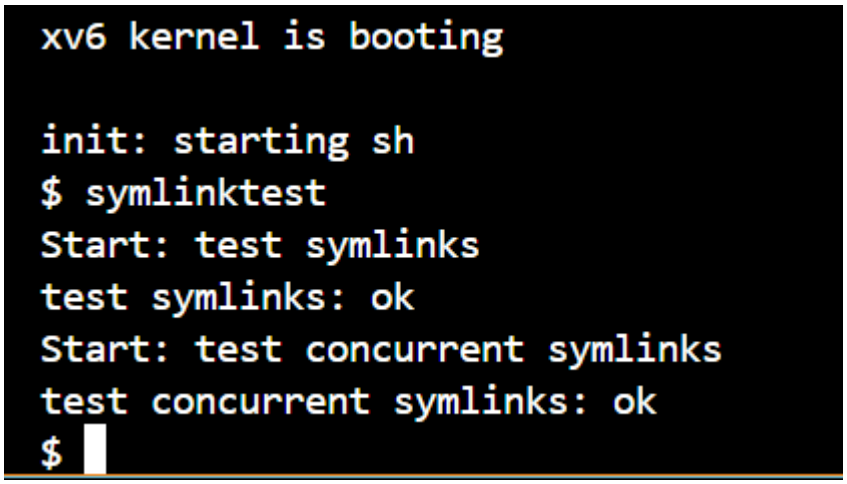
```
 if((dp =
nameiparent(new, name))
== 0)
    goto bad;
  ilock(dp);
  if(dp->dev != ip->dev
|| dirlink(dp, name,
ip->inum) < 0){
    iunlockput(dp);
    goto bad;
  }
  iunlockput(dp);
  iput(ip);

  end_op();

  return 0;

bad:
  ilock(ip);
  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  end_op();
  return -1;
}
```

注意到其中调用了创建节点的函数 create ，故我们也要对其进行修改，然后修改 sys_open ， 使 之能够跟随软链接并读取其指向的内容:

```
if(ip->type == T_SYMLINK) {
      if((omode & O_NOFOLLOW) == 0){
        char target[MAXPATH];
        int recursive_depth = 0;
        while(1){
          if(recursive_depth >= 10){
      iunlockput(ip);
      end_op();
      return -1;
    }
          if(readi(ip, 0, (uint64)target, ip->size-MAXPATH,
MAXPATH) != MAXPATH){
      return -1;
    }
```

2.3 实验结果

编译运行 xv6 ， 然后执行 symlinktest ， 得到结果如下， 可知测试通过:



2.4 实验注意事项

需要对硬链接和软链接有清楚的认知:
硬链接和软链接是文件系统中两种不同类型的链接方式， 区别主要在于:

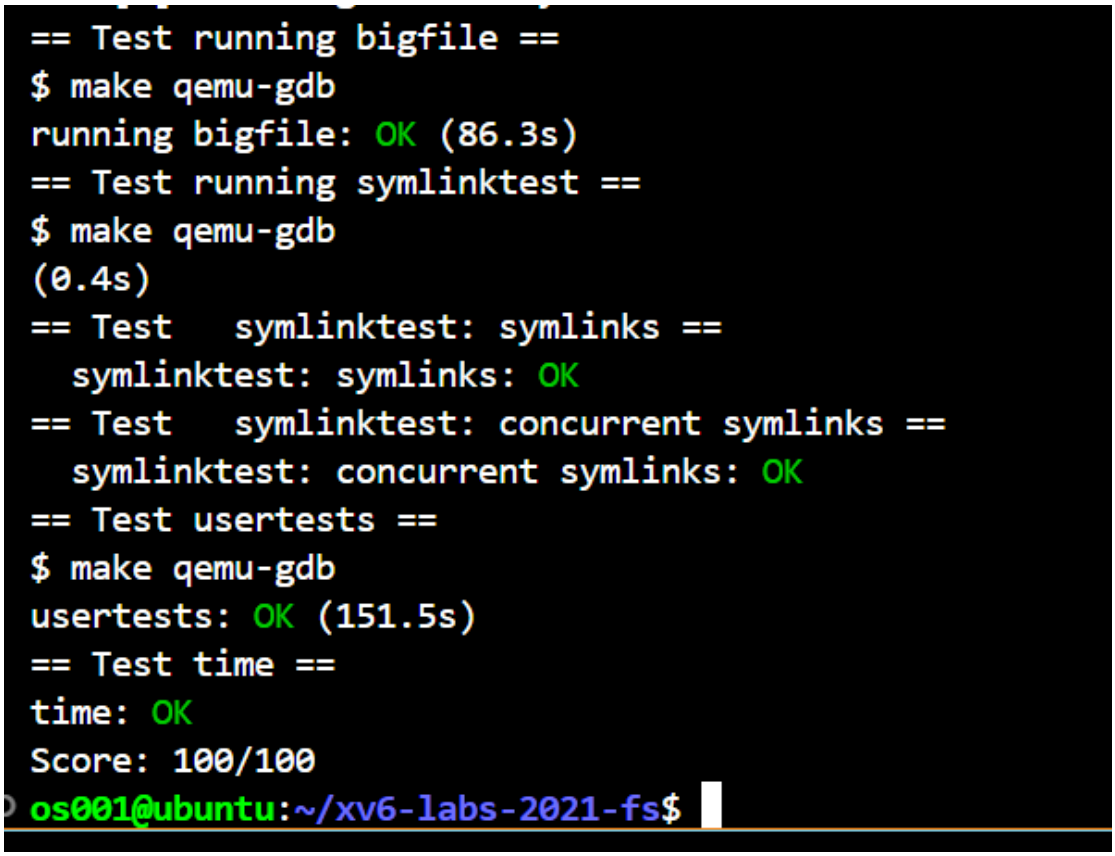
1. 物理位置:
   硬链接: 硬链接是指多个文件名指向同一个文件数据块， 它们在磁盘上实际上共享相 同的数据块， 因此对一个硬链接的更改会影响其他所有硬链接。硬链接只能指向同一 文件系统中的文件， 无法跨文件系统。
   软链接: 软链接是一个特殊的文件， 其中存储着目标文件的路径名。软链接实际上是一个指向目标文件的快捷方式， 类似于 Windows 中的快捷方式。它可以跨越文件系
2. 文件大小:
   硬链接: 硬链接没有独立的文件大小， 它们共享相同的数据块。删除一个硬链接并不 会影响实际的数据块， 只有当所有硬链接都被删除后， 数据块才会被释放。
   软链接: 软链接是一个独立的文件， 它包含了指向目标文件的路径名。软链接本身的 大小通常很小， 但是它需要额外的存储空间来存储路径名信息。
3. 更新和删除:
   硬链接: 由于硬链接共享数据块, 对任何一个硬链接的更改会影响其他所有硬链接。
   软链接: 软链接只是一个指向目标文件的路径名， 因此对软链接的更改不会影响目标 文件， 删除软链接也不会影响目标文件。
4. 跨文件系统:
   硬链接: 只能在同一文件系统内创建硬链接。
   软链接: 可以跨越文件系统， 指向不同的位置。
5. 目标类型:
   硬链接: 只能链接普通文件， 无法链接目录或特殊文件。
   软链接: 可以链接目录、文件和特殊文件。

3. 完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-fs
git fetch
git checkout fs
make clean
make grade
```

结果如下，可见测试全部通过：



4. 遇到的问题及解决方法

忘记修改 file.h 中的 inode ，导致 ip->addr 最后一个地址有值，报错 panic: virtio_disk_intr status ，通过上网查阅资料和与同学讨论解决该问题。

5.实验收获

在这个实验中，我们已经涉及了 Unix 文件系统的许多关键特性，并对 xv6 文件系统进行了改进， 使其与一些早期的现代文件系统（如 ext2 ）相当。这包括了以下特性：

1. 基于 Inode 的多级索引文件组织：使用 Inode 数据结构来管理文件元数据和数据块的指针，支持多级索引以有效地管理大文件。

2. 日志文件系统：引入日志机制，保护文件系统免受崩溃和意外断电的影响，确保数据的一致性。

3. 硬链接和软链接：实现了硬链接和软链接，允许文件间的共享和引用，提供更灵活的文件管理方式。

4. Inode 缓存与数据块缓存：采用缓存机制提高文件系统的访问性能，包括 Inode 缓存和数据 块缓存，减少对磁盘的频繁读写。

5. 基于缓存的写入优化：通过将写入操作缓存并批量处理，减少频繁的磁盘写入，提高写入性能。

虽然 xv6 在上述特性上已经有了不错的实现，但仍然有一些现代文件系统的特性没有在本实验中涉 及。这些特性包括：

1. 基于 Extent 的空间管理方式：一种更有效的空间管理方式，用连续的范围（Extent）来代替 传统的分散的数据块。
2. 成组的空间管理：将数据块划分成组，以更高效地管理和分配文件数据块。
3. 基于 B 树的索引：采用 B 树数据结构来优化文件系统索引，提高索引查询的性能。
4. 原子性读写操作：实现原子性读写操作，确保多个进程同时读写时数据的一致性。
5. 跨物理卷管理：支持在不同物理卷上分布文件数据，提高存储空间的利用率和容量。
6. 软件 RAID 机制：实现软件级别的 RAID（冗余磁盘阵列）来提高数据的冗余性和可用性。
7. 快照机制：支持创建文件系统的快照，以便在需要时还原文件系统到特定时间点的状态。
8. 只读文件系统优化：针对只读文件系统进行优化，提高只读访问的性能和效率。

虽然在本实验中没有直接涉及上述特性，但通过学习一些代表性的现代文件系统（如 ext4 、 xfs 、 btrfs 、 erofs ），我们可以更深入地了解这些特性的原理和实现方式。这有助于我们把握操作 系统中文件系统的发展方向，为未来的文件系统设计和实现提供参考。

十二、 lab10：mmap

1. mmap

1.1 实验要求

在这个实验中，你将向 xv6 中添加 mmap 和 munmap 系统调用，这两个系统调用允许 UNIX 程序对 其地址空间进行详细控制。它们可以用于在进程之间共享内存，将文件映射到进程的地址空间中，并作 为用户级页面故障方案的一部分，比如在讲座中讨论的垃圾回收算法。在本实验中，你将专注于实现内 存映射文件的功能。

1.2 实验过程

```
cd xv6-labs-2021-mmap
git fetch
git checkout mmap
make clean
```

首先定义 vma 结构体用于保存内存映射信息，并在 proc 结构体中加入 struct vma *vma 指针：

```
#define NVMA 16
#define VMA_START (MAXVA / 2)
struct vma{
  uint64 start;
  uint64 end;
  uint64 length; // 0 means vma not used
  uint64 off;
  int permission;
  int flags;
  struct file *file;
  struct vma *next;
  struct spinlock lock;
};
struct proc {
...
  struct vma *vma;
...
};
```

之后实现对 vma 分配的代码：

```
struct vma vma_list[NVMA];
struct vma* vma_alloc(){
  for(int i = 0; i < NVMA; i++){
    acquire(&vma_list[i].lock);
    if(vma_list[i].length == 0){
      return &vma_list[i];
    }else{
      release(&vma_list[i].lock);
    }
  }
  panic("no enough vma");
}
```

实现 mmap 系统调用，这个函数主要就是申请一个 vma ，之后查找一块空闲内存，填入相关信息， 将 vma 插入到进程的 vma 链表中去：

```
uint64
sys_mmap(void)
{
  uint64 addr;
  int length, prot, flags, fd, offset;
  if(argaddr(0, &addr) < 0 || argint(1, &length) < 0 || argint(2, &prot) < 0 ||
argint(3, &flags) < 0 || argint(4, &fd) < 0 || argint(5, &offset) < 0){
    return -1;
 }
  if(addr != 0)
    panic("mmap: addr not 0");
  if(offset != 0)
    panic("mmap: offset not 0");
  struct proc *p = myproc();
  struct file* f = p->ofile[fd];
  int pte_flag = PTE_U;
  if (prot & PROT_WRITE) {
    if(!f->writable && !(flags & MAP_PRIVATE)) return -1; // map to a unwritable
file with PROT_WRITE
    pte_flag |= PTE_W;
 }
  if (prot & PROT_READ) {
    if(!f->readable) return -1; // map to a unreadable file with PROT_READ
    pte_flag |= PTE_R;
 }
  struct vma* v = vma_alloc();
  v->permission = pte_flag;
  v->length = length;
  v->off = offset;
  v->file = myproc()->ofile[fd];
  v->flags = flags;
  filedup(f);
  struct vma* pv = p->vma;
  if(pv == 0){
    v->start = VMA_START;
    v->end = v->start + length;
    p->vma = v;
 }else{
    while(pv->next) pv = pv->next;
    v->start = PGROUNDUP(pv->end);
    v->end = v->start + length;
    pv->next = v;
    v->next = 0;
```

接下来就可以在 usertrap 中对缺页中断进行处理：查找进程的 vma 链表，判断该地址是否为映射 地址，如果不是就说明出错，直接返回；如果在 vma 链表中，就可以申请并映射一个页面，之后根据 vma 从对应的文件中读取数据：

```
int
mmap_handler(uint64 va, int scause)
{
  struct proc *p = myproc();
  struct vma* v = p->vma;
  while(v != 0){
    if(va >= v->start && va < v->end){
      break;
   }
    //printf("%p\n", v);
    v = v->next;
 }
  if(v == 0) return -1; // not mmap addr
  if(scause == 13 && !(v->permission & PTE_R)) return -2; // unreadable vma
  if(scause == 15 && !(v->permission & PTE_W)) return -3; // unwritable vma
  // load page from file
  va = PGROUNDDOWN(va);
  char* mem = kalloc();
  if (mem == 0) return -4; // kalloc failed

  memset(mem, 0, PGSIZE);
  if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, v->permission) != 0){
    kfree(mem);
    return -5; // map page failed
 }
  struct file *f = v->file;
  ilock(f->ip);
  readi(f->ip, 0, (uint64)mem, v->off + va - v->start, PGSIZE);
  iunlock(f->ip);
  return 0;
}
```

写回函数先判断是否需要写回，当需要写回时就仿照 filewrite 的实现，将数据写回到对应的文件 当中去，这里的实现是直接写回所有页面，但实际可以根据 PTE_D 来判断内存是否被写入，如果没有写 入就不用写回：

```
if(!(v->permission & PTE_W) || (v->flags & MAP_PRIVATE)) // no need to
writeback
    return;
  if((addr % PGSIZE) != 0)
    panic("unmap: not aligned");
  printf("starting writeback: %p %d\n", addr, n);
  struct file* f = v->file;
  int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;
  int i = 0;
  while(i < n){
    int n1 = n - i;
    if(n1 > max)
      n1 = max;
    begin_op();
    ilock(f->ip);
    printf("%p %d %d\n",addr + i, v->off + v->start - addr, n1);
    int r = writei(f->ip, 1, addr + i, v->off + v->start - addr + i, n1);
    iunlock(f->ip);
    end_op();
    i += r;
```

1.2 实验结果

输入 mmaptest

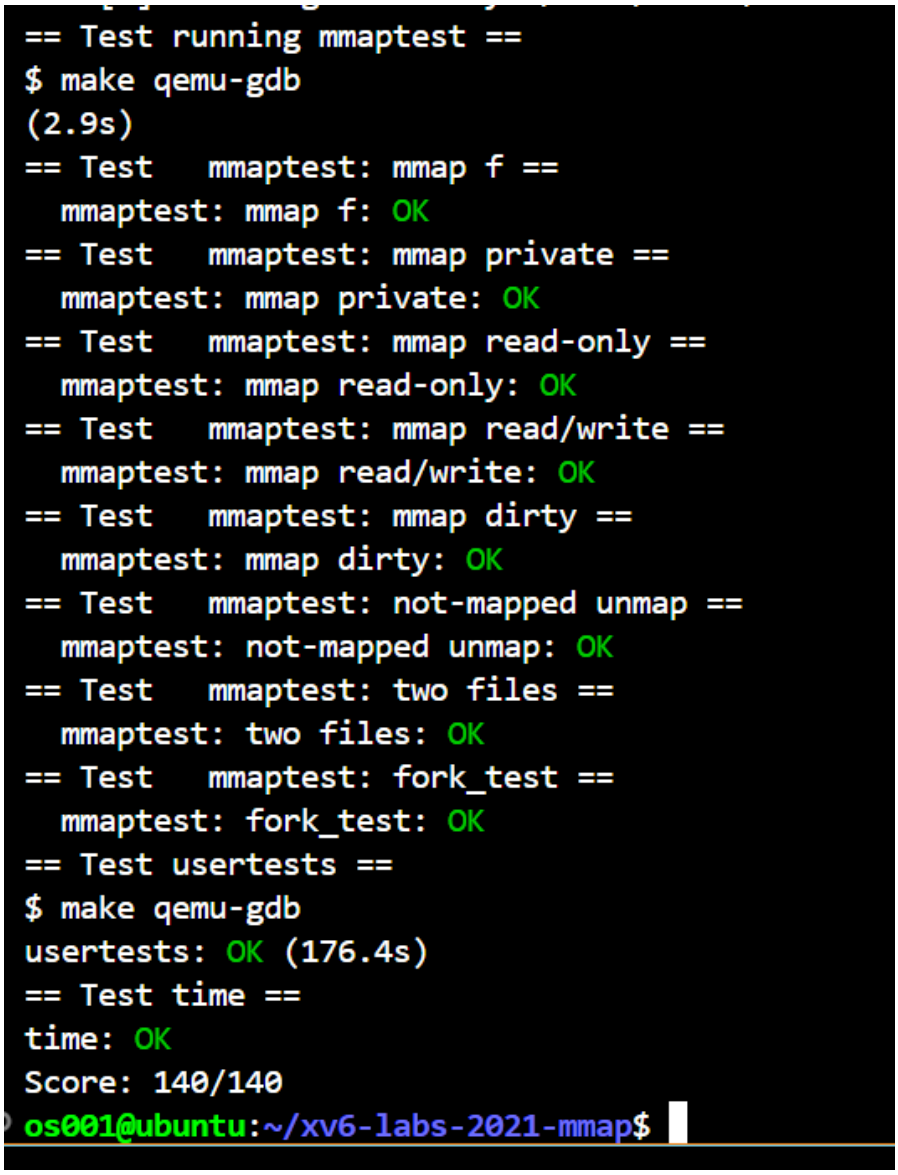
```
xv6 kernel is booting

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
$
```

2. 完全测试

在目录下创建一个 time.txt，内容为完成实验的小时数。在终端执行

```
cd xv6-labs-2021-mmap
git fetch
git checkout mmap
make clean
make grade
```

结果如下，可见测试全部通过：



3. 遇到的问题及解决方法

理映射到进程地址空间的虚拟内存页可能会涉及到地址冲突、页表更新等问题。

解决方法： 在 mmap 和 munmap 的实现中，需要仔细考虑如何管理进程的页表，确

保映射的虚拟 页与已有的页不冲突。同时，对于取消映射的情况，需要正确地处理页表的更新和释放。

4. 实验收获

在 mmap 实验中，我获得了以下实验收获：

深入理解内存管理： 通过实现 mmap 和 munmap 系统调用，我更深入地理解了虚拟内存的 概念以及内存映射的工作原理。我学会了如何将文件内容映射到进程的虚拟地址空间，以及如 何管理和操作这些映射。

掌握文件系统集成： 实现 mmap 和 munmap 也涉及到文件系统的操作。我理解了如何处理 文件的读取、写入和取消映射等操作，同时也考虑了文件在虚拟地址空间中的布局。

并发和同步机制： 在实现 mmap 的过程中，我考虑了多个进程同时访问映射文件的情况，从 而深入了解了如何使用同步机制，如锁，来保护共享资源免受并发访问的影响。

理解系统调用和用户态编程： 实现系统调用是操作系统的核心功能之一。通过编写 mmap 和 munmap 系统调用，我更深刻地理解了系统调用接口和用户态编程的过程。

优化内存管理： 在实现 mmap 和 munmap 过程中，我思考了如何有效地管理内存页的分配 和释放，以及如何优化页面的读写和回写策略，从而提高性能和资源利用率。

系统设计和调试能力： 实验中我需要将不同模块（内存管理、文件系统、系统调用等）进行集成，这锻炼了我的系统设计和调试能力，使我更深入地理解整个操作系统的协作运作。

尾声

xv6 是一个经典的操作系统教学项目，它提供了一个深入理解操作系统原理和概念的学习平台。在本次实验中，我对 xv6`进行了全面的探索，从内存管理、多线程编程、文件系统到进程同步等方面，深入理解了操作系统的核心机制和功能。

在实验的过程中，我首先深入研究了内存管理的原理，学习了虚拟内存、页面置换和内存保护等概念。通过实现页面置换算法，我深刻认识到了内存管理在操作系统中的重要性，以及如何合理利用有限的物理内存资源。

接着，我探索了多线程编程和同步机制，学习了如何使用锁和条件变量来确保多线程的安全访问共享资源。在实现线程库和线程同步的过程中，我深入理解了并发编程的挑战和解决方法，体会到了多线程环境下数据竞争和死锁等问题的影响。

在文件系统的部分，我学习了磁盘块的组织方式、文件元数据的管理和文件访问控制等知识。通过实现符号链接和内存映射文件等功能，我更深刻地理解了文件系统在操作系统中的作用，以及如何通过索引、目录结构等方式实现文件的高效管理。

在实验的过程中，我还遇到了各种问题，比如线程同步引发的死锁、文件系统的性能优化等。通过调试和查阅资料，我不断优化代码，解决了这些问题，并获得了宝贵的经验。

通过完成这些实验，我不仅加深了对操作系统原理的理解，还提升了编程技能和问题解决能力。我认识到操作系统是计算机系统中不可或缺的一部分，它为应用程序提供了资源管理和抽象层，使得计算机能够高效运行。

最后，我要衷心感谢我的老师，在课堂上传授知识，他们的深刻见解和专业知识使我受益匪浅，让我在实验中不断进步，不断克服困难，取得了更好的成果；其次，我要感谢学校为我们提供了如此优秀的教育资源。从课程设置到实验环境，都为我们提供了良好的学习平台。我在这里学到了不仅是知识，还有创新、思考和团队协作的重要性。学校为我们提供了一个培养综合素质的环境，让我们能够更好地面对未来的挑战；最后，我还要感谢 xv6 提供了一个优秀的学习平台，让我有机会亲身体验操作系统的工作原理，从而更好地理解计算机科学的核心概念。这些实验不仅增强了我的学术知识，还培养了我分析问题和解决问题的能力，为我今后的学习和职业发展打下了坚实的基础。