



同济大学

Tongji University

《操作系统课程设计》 项目报告

报告名称: xv6及Labs课程项目

班 级: 42028702

学 号: 2351493

姓 名: 胡浩杰

指导老师: 王冬青

目录

Lab0:Environment Setup
Lab1:Utilities
Boot xv6 (easy)
sleep (easy)
pingpong (easy)
primes (moderate)/(hard)
find (moderate)
xargs (moderate)
Lab2:System Calls
System call tracing (moderate)
Sysinfo (moderate)
Lab3:Page Tables
Print a page table (easy)
A kernel page table per process (hard)
Simplify copyin/copyinstr (hard)
Lab4:Traps
RISC-V assembly (easy)
Backtrace (moderate)
Alarm (hard)
Lab5:Lazy allocation
Eliminate allocation from sbrk() (easy)

Lazy allocation (moderate).....	
Lazytests and Usertests (moderate).....	
Lab6:Copy-on-Write Fork for xv6	
Implement copy-on write(hard).....	
Lab7:Multithreading	
Uthread: switching between threads (moderate).....	
Using threads (moderate).....	
Barrier(moderate).....	
Lab8:Locks	
Memory allocator (moderate).....	
Buffer cache (hard).....	
Lab9:File system	
Large files (moderate).....	
Symbolic links (moderate).....	
Lab10:Mmap	
mmap (hard).....	
Lab11:Networking	
Your Job (hard).....	

Lab0:Environment Setup

[实验目的]

Windows 子系统适用于 Linux(Windows Subsystem for Linux,简称 WSL)是 Microsoft 提供的一项功能，允许用户在 Windows 操作系统上运行 Linux 环境，而无需安装虚拟机或双重启动。这对于开发人员和系统管理员非常有用，因为它结合了 Windows 的便利性和 Linux 的强大工具集。

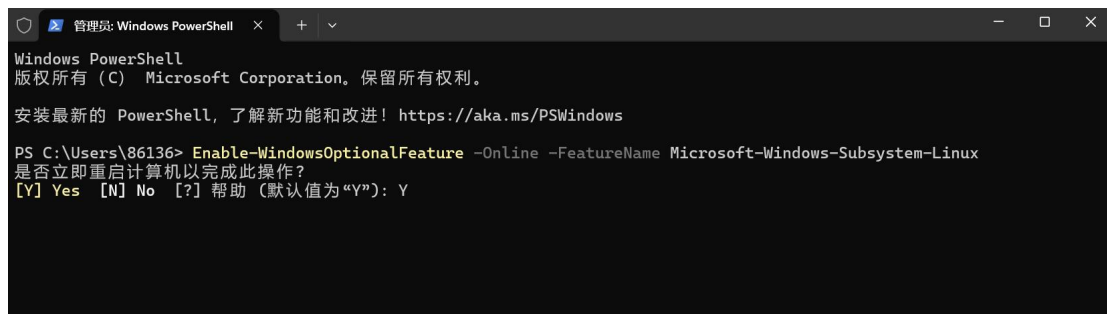
这个实验的目的主要是完成安装 Ubuntu20.04、配置 XV6 系统和 github 远端仓库的建立。

[实验步骤]

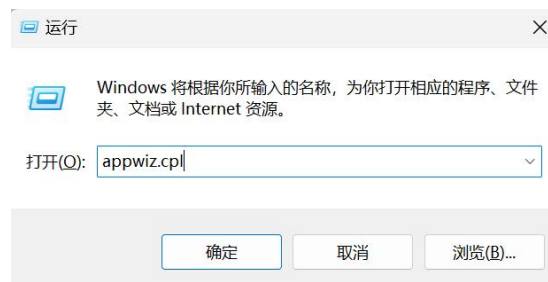
1.安装 Ubuntu20.04

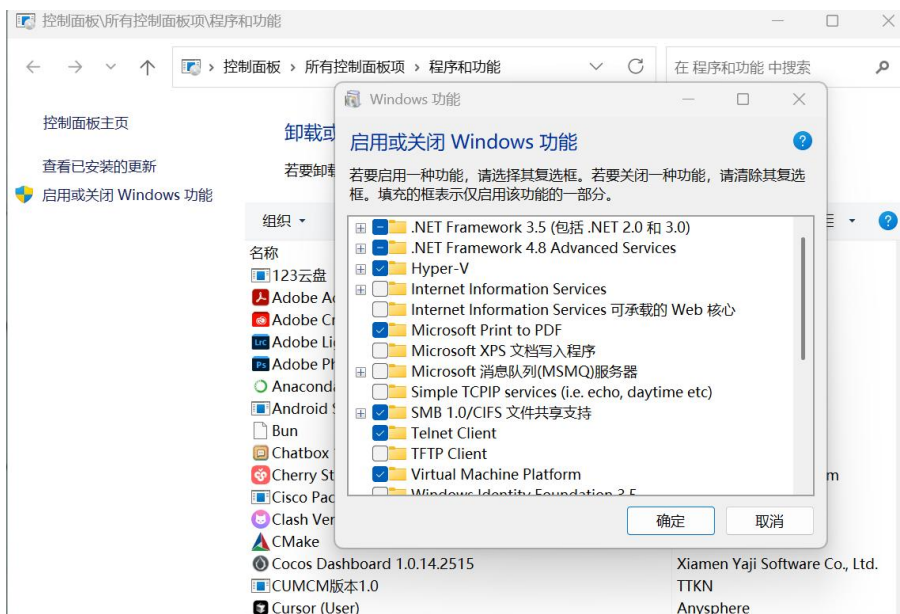
首先我们需要开启个人电脑上的 WSL 支持。

打开使用管理员权限的 Shell,输入命令 `Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux`,以上命令会激活 WSL 服务，然后需要重启系统。

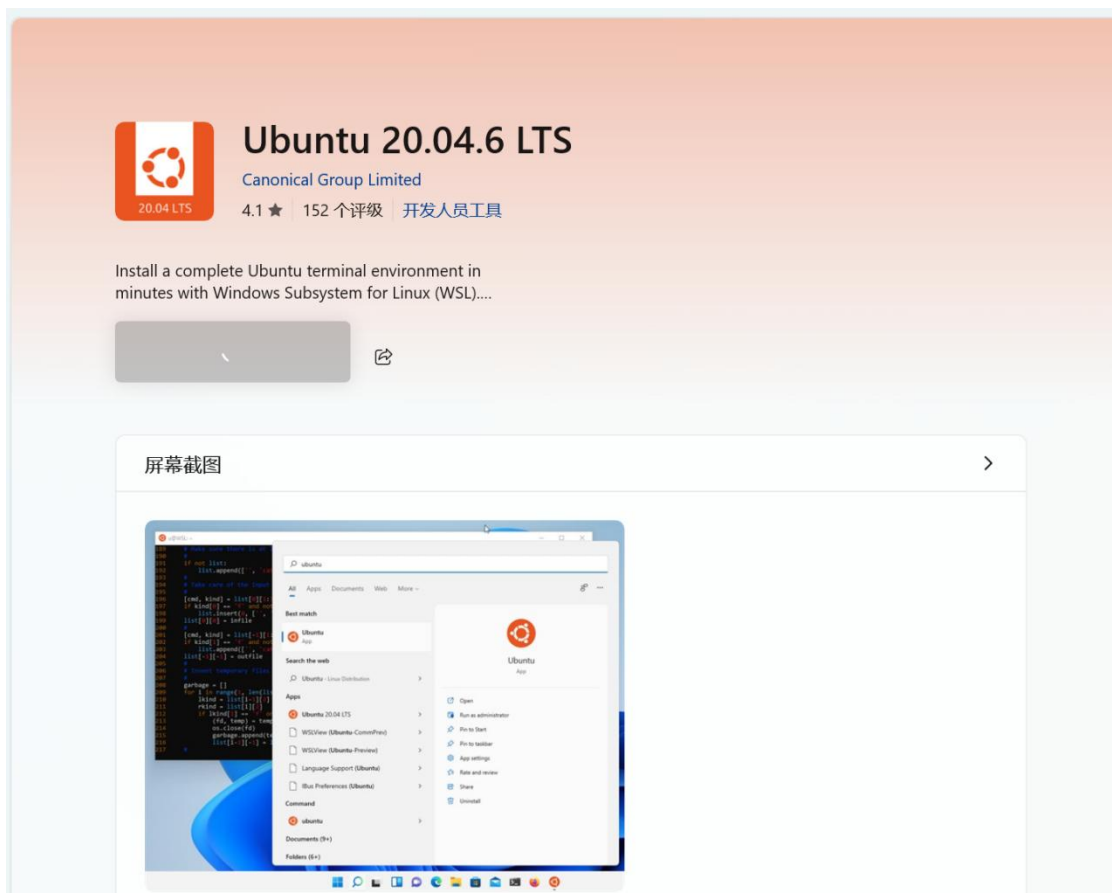


电脑重启之后，Win + R，输入 `appwiz.cpl`，在左上角找到“启动或关闭 Windows 功能”，此时我们就会看到这个选项处于选中状态。





在微软应用商店里搜索 ubuntu ， 实验选择 Ubuntu 20.04 LTS 安装。



下载完后，打开终端，根据指导完成新用户的注册和密码输入即可。

```
jack@DESKTOP-EMMCGRI: ~  
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 6.6.87.2-microsoft-standard-WSL2 x86_64)  
  
* Documentation: https://help.ubuntu.com  
* Management:   https://landscape.canonical.com  
* Support:      https://ubuntu.com/advantage  
  
System information as of Tue Jul 8 21:29:32 CST 2025  
  
System load: 0.41          Processes: 96  
Usage of /: 0.1% of 1006.85GB    Users logged in: 0  
Memory usage: 8%             IPv4 address for eth0: 172.20.156.181  
Swap usage: 0%  
  
Expanded Security Maintenance for Applications is not enabled.  
  
0 updates can be applied immediately.  
  
Enable ESM Apps to receive additional future security updates.  
See https://ubuntu.com/esm or run: sudo pro status  
  
The list of available updates is more than a week old.  
To check for new updates run: sudo apt update  
  
This message is shown once a day. To disable it please create the  
/home/jack/.hushlogin file.  
jack@DESKTOP-EMMCGRI: $
```

2.配置 XV6 系统

根据 <https://pdos.csail.mit.edu/6.828/2020/tools.html> 网站 tool 上的教学指南，在 Ubuntu 终端命令行输入命令 `sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu`，一键安装安装 qemu 等所需依赖。

```
jack@DESKTOP-EMMCGRI: $ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-  
riscv64-linux-gnu binutils-riscv64-linux-gnu  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
acl adwaita-icon-theme at-spi2-core binutils binutils-common binutils-x86-64-linux-gnu cpp cpp-9  
cpp-9-riscv64-linux-gnu cpp-riscv64-linux-gnu dpkg-dev fakeroot fontconfig g++ g++-9 gcc  
gcc-10-base gcc-10-cross-base gcc-9 gcc-9-cross-base gcc-9-cross-base-ports  
gcc-9-riscv64-linux-gnu gcc-9-riscv64-linux-gnu-base gdb gdbserver gstreamer1.0-plugins-base  
gstreamer1.0-plugins-good gstreamer1.0-x gtk-update-icon-cache hicolor-icon-theme  
humanity-icon-theme ibverbs-providers ipxe-qemu libaal libalgorithm-diff-perl  
libalgorithm-diff-xs-perl libalgorithm-merge-perl libasan5 libatk-bridge2.0-0 libatk1.0-0  
libatk1.0-data libatomic1 libatomic1-riscv64-cross libatspi2.0-0 libavahi-client3  
libavahi-common-data libavahi-common3 libbavc1394-0 libbabeltrace1 libbinutils  
libboost-iostreams1.71.0 libboost-thread1.71.0 libbrotli1.0.7 libc-dev-bin libc6 libc6-dbg  
libc6-dev libc6-dev-riscv64-cross libc6-riscv64-cross libcaca0 libcacard0 libcairo-gobject2  
libcairo2 libcc1-0 libcdparanoia0 libcolor2 libcrypt-dev libctf-nobfd0 libctf0 libcupsw2  
libdatatriel libdtkpg-perl libdvd4 libdwl libelf1 libepoxy0 libfakeroot libfdt1  
libfile-fcntllock-perl libgbl libgcc-9-dev libgcc-9-dev-riscv64-cross libgcc-s1  
libgcc-s1-riscv64-cross libgdk-pixbuf2.0-0 libgdk-pixbuf2.0-bin libgdk-pixbuf2.0-common libgomp1  
libgomp1-riscv64-cross libgraphite2-3 libgstreamer-plugins-base1.0-0  
libgstreamer-plugins-good1.0-0 libgtk-3-0 libgtk-3-bin libgtk-3-common libharfbuzz0b libibverbs1  
libiec1883-0 libiscsi7 libisl22 libitm1 libjack-jackd2-0 libjbig0 libjpeg-turbo8 libjpeg8  
liblcms2-2 liblsm0 libmp3lame0 libmpc3 libmpg123-0 libnl-3-200 libnl-route-3-200 libopus0  
liborc-0.4-0 libpango-1.0-0 libpangocairo-1.0-0 libpangoft2-1.0-0 libpcsc-lite libpixmap-1-0  
libpnm0 libquadmath0 librados2 libraw1394-11 librbdl librdmacm1 librest-0.7-0 librsync2-2  
librsync2-common librsamplerate0 libshout3 libslirp0 libsoup-gnome2.4-1 libspeex1 libspice-server1  
libstdc++-9-dev libstdc++6 libtag1v5 libtag1v5-vanilla libthai-data libthai0 libtheora0 libtiff5  
libtsan0 libtwolame0 libubsan1 libusbredirparser1 libv4l-0 libv4lconvert0 libvirglrenderer1  
libvisual-0.4-0 libvpx6 libvte-2.91-0 libvte-2.91-common libwavpack1 libwayland-cursor0  
libwayland-egl1 libwayland-server0 libwebp6 libxcb-render0 libxcursor1 libxdamage1 libxkbcommon0  
linux-libc-dev linux-libc-dev-riscv64-cross make manpages-dev qemu-block-extra qemu-system-common  
qemu-system-data qemu-system-gui qemu-utils seabios sharutils ubuntu-mono  
Suggested packages:  
binutils-doc cpp-doc gcc-9-locales debconf-keyring g++-multilib g++-9-multilib gcc-9-doc  
gcc-multilib autoconf automake libtool flex bison gcc-doc gcc-9-multilib gdb-riscv64-linux-gnu  
gdb-doc git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs  
git-mediawiki git-svn gvfs glibc-doc colord cups-common bzip2 libdv-bin oss-compat  
libvisual-0.4-plugins jackd2 liblms2-utils opus-tools pscd libraw1394-doc librsync2-bin speex  
gstreamer1.0-plugins-ugly libstdc++-9-doc make-doc samba vde2 debotstrap sharutils-doc bsd-mailx  
| mailx  
The following NEW packages will be installed:  
acl adwaita-icon-theme at-spi2-core binutils binutils-common binutils-riscv64-linux-gnu  
binutils-x86-64-linux-gnu build-essential cpp cpp-9 cpp-9-riscv64-linux-gnu cpp-riscv64-linux-gnu  
dpkg-dev fakeroot fontconfig g++ g++-9 gcc gcc-10-cross-base gcc-9 gcc-9-base  
gcc-9-cross-base gcc-9-cross-base-ports gcc-9-riscv64-linux-gnu gcc-9-riscv64-linux-gnu-base gcc-riscv64-linux-gnu
```

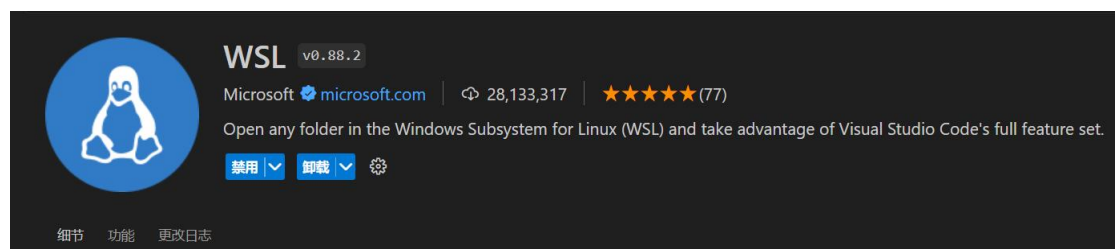
测试安装，在终端输入命令 `riscv64-unknown-elf-gcc --version` 和命令 `qemu-system-riscv64 --version`。

```
jack@DESKTOP-EMMCGRI:~$ riscv64-unknown-elf-gcc --version
riscv64-unknown-elf-gcc () 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

jack@DESKTOP-EMMCGRI:~$ qemu-system-riscv64 --version
QEMU emulator version 4.2.1 (Debian 1:4.2-3ubuntu6.30)
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
```

克隆实验所需代码 `git clone git://g.csail.mit.edu/xv6-labs-2020`。

在命令行中输入 `cd xv6-labs-2020/`，切换到刚刚克隆到本地的文件夹下。这时，需要在 VSCode 中安装插件 Remote-WSL。



切换到 WSL shell 中，进入目标文件夹，输入 `code .` 命令，即可在当前目录打开 Windows 下的 VSCode。

```
jack@DESKTOP-EMMCGRI:~$ cd xv6-labs-2020/
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ code
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$
```

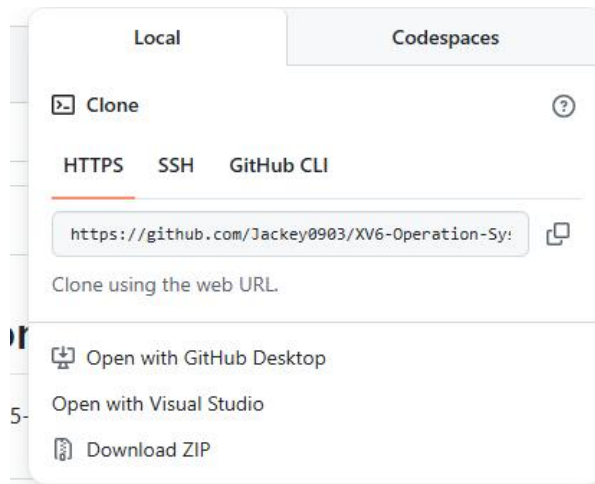
完成以上环境的配置，便可以在我们的 VSCode 中修改代码，在相对应的终端进行调试，观察实验结果。

3.github 仓库的建立

我的 github 仓库建立和版本控制参考以下教程完成：

https://xv6.dgs.zone/labs/use_git/git1.html

首先需要在 github 上建立一个远端仓库，同时获取改远端仓库的 HTTPS。



接着在 WSL 终端打开我们代码所在的子目录。

```
cd xv6-labs-2020/
```

```
cat .git/config
```

添加 git 仓库地址使用以下命令：

```
git remote add github
```

```
https://github.com/Jackey0903/XV6-Operation-System.git
```

```
cat .git/config
```

之后输入你的用户名和密码即可完成，最后再将实验所用到的分支推送到 github 即可，比如以下推送实验 1 所使用的 util 分支：

```
git checkout util
```

```
git push github util:util
```

之后其他实验的版本控制类似，按照指导要求切换到相应的实验目录下，即可对不同的实验管理不同的分支，之后在传送到远程仓库中。

[实验中遇到的问题和解决方法]

环境的搭建是比较麻烦的步骤，在本次实验初始环境的搭建中，我遇到了问题，通过查找资料和搜索相关教程均得到了较好的解决。

- Ubuntu 20.04.6 LTS repo int 提示/usr/bin/env: “python”: 权限不够。这个问题的原因可能是这是由于 ubuntu20.04 默认安装的 python3，将 python 命令配置 i

为了 python3 为软连接，此时只需要通过命令添加配置为 python 软连接即可。

输入代码 `sudo ln -s /usr/bin/python3.8 /usr/bin/python` 即可解决。

- 在 WSL 终端输入命令 `make qemu` 时，出现报错：

Command 'make' not found, but can be installed with:

```
sudo apt install make          # version 4.2.1-1.2, or
```

```
sudo apt install make-guile    # version 4.2.1-1.2
```

通过查找资料，发现原因是我的系统上没有安装 `make` 工具，执行以下两条命令 `sudo apt-get update` 和 `sudo apt-get install make` 即可解决问题。

[实验心得]

搭建 xv6 实验环境，无疑是深入学习操作系统基础知识的绝佳契机。这一过程首先要求我们安装一系列必要工具，其中 `make` 工具与 RISC-V 工具链是核心组件。在实际操作中，我深刻体会到，透彻理解并熟练运用开发工具链绝非可有可无——它直接关系到后续编译工作的顺畅度与调试环节的效率，是保障实验推进的基础前提。

然而，xv6 环境的搭建并非坦途，过程中往往会遭遇各种挑战。比如工具链版本与系统环境的兼容性冲突，或是因权限配置不当导致的操作受阻等问题。值得注意的是，每一次排查并解决问题的经历，都是深化对系统运行机制与工具链工作原理理解的过程。这些积累不仅帮助我们攻克了当下的障碍，更为后续实验的开展筑牢了根基。

需要强调的是，环境搭建与配置的过程具有一定的复杂性和不确定性，不同的硬件环境、系统版本都可能引发独特的问题。面对这些未知挑战，耐心排查、逐步纠错的态度与能力至关重要，这本身也是操作系统学习中不可或缺的实践素养。

Lab1:Utilities

Boot xv6 (easy)

[实验目的]

利用 WSL 终端切换到 `xv6-labs-2020` 代码的 `util` 分支，同时实验 `qemu` 模拟器启动并运行 `xv6` 系统，观察实验现象和结果。

[实验步骤]

首先需要安装相应的配置包，在 wsl 终端输入以下命令：

`sudo apt update`

```
Need to get 12.5 MB of archives.  
After this operation, 87.3 MB of additional disk space will be used.  
Do you want to continue? [Y/n] y  
0% [Working]  
Get:1 http://archive.ubuntu.com/ubuntu focal/main amd64 dctrl-tools amd64 2.24-3 [61.5 kB]  
Get:2 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 dkms all 2.8.1-5ubuntu2 [66.8 kB]  
Get:3 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 linux-headers-5.4.0-216 all 5.4.0-216.236 [11.0 MB]  
Get:4 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 linux-headers-5.4.0-216-generic amd64 5.4.0-216.236 [1362 kB]  
Get:5 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 linux-headers-generic amd64 5.4.0-216.208 [2340 B]  
Fetched 12.5 MB in 3s (3782 kB/s)  
Selecting previously unselected package dctrl-tools.  
(Reading database ... 56532 files and directories currently installed.)  
Preparing to unpack .../dctrl-tools_2.24-3_amd64.deb ...  
Unpacking dctrl-tools (2.24-3) ...  
Selecting previously unselected package dkms.  
Preparing to unpack .../dkms_2.8.1-5ubuntu2_all.deb ...  
Unpacking dkms (2.8.1-5ubuntu2) ...  
Selecting previously unselected package linux-headers-5.4.0-216.  
Preparing to unpack .../linux-headers-5.4.0-216_5.4.0-216.236_all.deb ...  
Unpacking linux-headers-5.4.0-216 (5.4.0-216.236) ...  
Selecting previously unselected package linux-headers-5.4.0-216-generic.  
Preparing to unpack .../linux-headers-5.4.0-216-generic_5.4.0-216.236_amd64.deb ...  
Unpacking linux-headers-5.4.0-216-generic (5.4.0-216.236) ...  
Selecting previously unselected package linux-headers-generic.  
Preparing to unpack .../linux-headers-generic_5.4.0-216.208_amd64.deb ...  
Unpacking linux-headers-generic (5.4.0-216.208) ...  
Setting up linux-headers-5.4.0-216 (5.4.0-216.236) ...  
Setting up linux-headers-5.4.0-216-generic (5.4.0-216.236) ...  
Setting up linux-headers-generic (5.4.0-216.208) ...  
Setting up dctrl-tools (2.24-3) ...  
Setting up dkms (2.8.1-5ubuntu2) ...  
Processing triggers for man-db (2.9.1-1) ...  
jack@DESKTOP-EMMCGRI: ~/xv6-labs-2020$
```

`sudo apt install build-essential gcc make perl dkms git gcc-riscv64-unknown-elf
gdb-multiarch qemu-system-misc`

在 WSL 终端进入实验目录并且切换到第一个实验分支 util。

`cd xv6-labs-2020`

`git checkout util`

```
jack@DESKTOP-EMMCGRI: ~/xv6-labs-2020$ cd xv6-labs-2020  
-bash: cd: xv6-labs-2020: No such file or directory  
jack@DESKTOP-EMMCGRI: ~/xv6-labs-2020$ git checkout util  
Switched to branch 'util'  
Your branch is up to date with 'origin/util'.
```

我们这时可以在 xv6-labs-2020 目录下的终端中输入 `make qemu` 来启动 xv6；
会得到如下结果表示，已经成功编译并启动 xv6 系统：

```

mmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
ballocc: first 591 blocks have been allocated
ballocc: 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 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 23968
echo            2 5 22800
forktest        2 6 13168
grep            2 7 27320
init            2 8 23896
kill            2 9 22768
ln              2 10 22720
ls              2 11 26208
mkdir           2 12 22872
rm              2 13 22856
sh              2 14 41752
stressfs        2 15 23872
usertests       2 16 147512
grind           2 17 37984
wc              2 18 25112
zombie          2 19 22272
console         3 20 0

```

此时我们可以在终端中输入 `ls`, 显示出来的是 `mkfs` 包含在初始文件系统; 大多数是可以运行的程序。刚刚运行了其中之一: `ls`;

如果要退出 `qemu`, 请键入: `Ctrl-a x`, 即先按下 `Ctrl-a`, 再按下 `x`, 即可退出 `qemu`。

[实验中遇到的问题和解决方法]

第一个实验的内容相对浅显易懂, 操作流程也较为顺畅, 整个过程中并未遇到明显的阻碍或需要特别解决的问题。

[实验心得]

第一个实验的内容相对浅显易懂, 操作流程也较为顺畅, 整个过程中并未遇到明显的阻碍或需要特别解决的问题。

Sleep (easy)

[实验目的]

Implement the UNIX program sleep for xv6; your sleep should pause for a user-specified number of ticks. A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip. Your solution should be in the file user/sleep.c.

实现 xv6 的 UNIX 程序休眠；您的休眠应暂停用户指定的提示数。滴答是由 xv6 内核定义的时间概念，即来自计时器芯片的两次中断之间的时间。您的解决方案应该在文件 user/sleep.c 中。

[实验步骤]

实验代码：

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

int main(int argc, char *argv[]) // argc:表示传递给程序的命令行参数的数量;argv:包含传递给程序的命令行参数
{
    if (argc < 2) // 检查命令行参数的数量是否小于 2
    {
        fprintf(2, "Error...\n");
        exit(1);
    }
    sleep(atoi(argv[1])); // 将第 1 个参数转成整数,并且 sleep
    exit(0);
}
```

在 C 语言中，int argc, char *argv[] 是标准的主函数 main 的参数，用于接收命令行输入参数。argc 是一个整数，表示传递给程序的命令行参数的数量。这个数量包括程序本身的名字，所以它至少为 1。argv 是一个字符指针数组，包含传递给程序的命令行参数。每一个元素都是一个字符串(即字符指针)，表示一个命令行参数。argv[0]是程序的名字，argv[1]是第一个命令行参数，以此类推。

执行 sleep 10 命令：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sleep 10
$
```

在这种情况下：argc 的值为 2，因为有两个参数传递给程序 sleep 和 10。

argv 是一个数组，内容如下：argv[0]是字符串 "sleep"，即程序的名字；argv[1]是字符串 "10"，即第一个命令行参数。

查看正确得分：

```
ajack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.5s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.1s)
```

[实验中遇到的问题和解决方法]

这个实验的内容同样相对简单，在严格遵循指导说明进行操作的过程中，一切都进展顺利，没有出现任何需要额外处理的问题。

[实验心得]

首先，借助这个实验，我掌握了为 xv6 操作系统新增程序的具体方法；其次，在 VSCode 终端运行程序的过程中，通过深入理解 argc 与 argv 这两个命令行参数的作用机制，我能够编写出更具灵活性和动态性的命令程序，进而利用系统调用来对进程行为实现精准控制。

Pingpong (easy)

[实验目的]

Write a program that uses UNIX system calls to "ping-pong" a byte between two processes over a pair of pipes, one for each direction. The parent should send a byte to the child; the child should print "<pid>: received ping", where <pid> is its process ID, write the byte on the pipe to the parent, and exit; the parent should read the byte from the child, print "<pid>: received pong", and exit. Your solution should be in the file user/pingpong.c.

编写一个使用 UNIX 系统调用的程序来在两个进程之间“ping-pong”一个字节，请使用两个管道，每个方向一个。父进程应该向子进程发送一个字节；子进程应该打印“<pid>: received ping”，其中<pid>是进程 ID，并在管道中写入字节

发送给父进程，然后退出;父级应该从读取从子进程而来的字节，打印 “<pid>: received pong”，然后退出。您的解决方案应该在文件 user/pingpong.c 中。

[实验步骤]

实验代码:

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

int main(int argc, char *argv[])
{
    if (argc != 1) // 首先检查命令行参数个数，若不为 1 直接输出
error
    {
        fprintf(2, "Error...\n");
        exit(1);
    }
    int p[2]; // 定义一个整型数组 p，用于存放管道的两个文件描述符
    pipe(p); // 创建管道，p[0] 为读端，p[1] 为写端
    if (fork() == 0) // 如果 fork() == 0，则为子进程
    {
        close(p[0]); // 子进程是需要写的，使用关闭读端
        char temp = 'x';
        if (write(p[1], &temp, 1))
            fprintf(0, "%d: received ping\n", getpid()); // 向
标准输出打印消息，包含子进程的 PID
        close(p[1]); // 子进程写完了关闭写端
    }
    else // 此时为父进程
    {
        wait((int *)0); // 父进程需要等待子进程结束
        close(p[1]); // 父进程读，关闭写端
        char temp;
        if (read(p[0], &temp, 1))
            fprintf(0, "%d: received pong\n", getpid()); // 向
标准输出打印消息，包含父进程的 PID
        close(p[0]); // 父进程读完，关闭读端
    }
    exit(0);
}
```

这个程序清晰呈现了父子进程间的简易通信模型:

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


子进程通过管道写入数据，父进程则从管道读取数据，由此模拟出类似“ping-pong”的交互通信过程。具体而言，由于子进程负责写入操作，因此在写入时需关闭管道的读端，待数据写入完毕后，再关闭自身的写端；而父进程专注于读取操作，它需要等待子进程完成写入，在读取前先关闭自身的写端，读取完成后再关闭读端。

值得一提的是，实验中涉及的 `fork` 是 Unix 系统中的重要系统调用，其核心功能是创建新进程——新进程是调用进程（父进程）的副本。从返回值来看，`fork()` 在子进程中返回 0，在父进程中则返回新创建子进程的进程 ID（PID），这一特性为区分和控制父子进程的行为提供了关键依据。

在 `make qemu` 后执行 `pingpong` 命令：

查看正确得分：

```
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (1.0s)
```

[实验中遇到的问题和解决方法]

在本次实验的实现过程中，子进程数据写入与父进程数据读取部分的代码逻辑相对清晰，但关键在于合理区分父子进程的执行路径，而这一目标的实现依赖于对 `fork()` 系统调用机制的准确理解。具体来说，`fork()` 函数通过不同的返回值来标识当前执行上下文：在子进程中返回 0，在父进程中返回子进程的 PID（非零值）。基于这一特性，我们可以在代码中采用条件分支结构（如 `if-else` 语句）对父子进程的行为进行差异化处理。通过这种方式，能够简洁且有效地实现父子进程在管道通信中的不同职责，确保数据的正确传输与处理流程。

[实验心得]

本次实验的核心目标，是借助 `fork` 与 `pipe` 这两个系统调用，搭建起父子进程间的简易通信机制。在程序实现的过程中，我首先深入探究了 `fork` 创建新进程的内在逻辑——它不仅能生成一个全新的进程，更重要的是，这个新进程会与父进程共享相同的代码段和数据空间，这种特性为进程间的协作奠定了基础。而通过对 `pipe` 的实践，我掌握了在进程间构建通信通道的方法：利用文件描述符来完成数据的读取与写入操作，让原本相互独立的进程得以传递信息。在具体实现中，我完成了子进程向父进程发送消息的功能：先是由父进程创建管道，接着

调用 `fork` 生成子进程；之后，子进程通过管道发送一个字符，父进程则读取该字符并输出对应的信息，整个过程清晰地展现了进程间通信的基本流程。

Primes (moderate)/(hard)

[实验目的]

Write a concurrent version of prime sieve using pipes. This idea is due to Doug McIlroy, inventor of Unix pipes. The picture halfway down this page and the surrounding text explain how to do it. Your solution should be in the file `user/primes.c`.

使用管道编写 `prime sieve`(筛选素数)的并发版本。这个想法是由 Unix 管道的发明者 Doug McIlroy 提出的。请查看这个网站，该网页中间的图片 and 周围的文字解释了如何做到这一点。您的解决方案应该在 `user/primes.c` 文件中。

[实验步骤]

实验代码：

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

int subProcess(int *oldFd)
{
    close(oldFd[1]); // 关闭原管道写端
    int fd[2];
    int prime;
    int num;
    if (read(oldFd[0], &prime, 4)) // 若能从原管道读到数据
    {
        printf("prime %d\n", prime); // 第一个数据质数,进行输出
        pipe(fd);                    // 创建管道和子进程
        if (fork() == 0)              // 子进程
            subProcess(fd);           // 递归调用
        else                          // 父进程
        {
            close(fd[0]);              // 关闭新管道读端
            while (read(oldFd[0], &num, 4)) // 从原管道进行读
            {
```

```

        if (num % prime != 0) // 不能被记录的质数整除则
写入新管道
            write(fd[1], &num, 4);
    }
    close(oldFd[0]); // 此时父进程的原管道关闭，则关闭
原管道的读端
    close(fd[1]); // 关闭新管道的写端
    wait((int *)0); // 等待子进程结束
}
}
else
    close(oldFd[0]); // 此时说明原管道已关闭,第一个数字都读
    不出，不创建子进程直接关闭原管道读端
    exit(0);
}
int main()
{
    int fd[2];
    pipe(fd);
    if (fork() == 0) // 子进程
        subProcess(fd);
    else // 父进程
    {
        close(fd[0]);
        for (int i = 2; i <= 35; ++i) //遍历 2~35 写入管道写端
            write(fd[1], &i, 4);
        close(fd[1]); // 写完关闭管道写端并等待子进程结束
        wait((int *)0);
    }
    exit(0);
}

```

这段代码的实现存在一定复杂度。在 `main` 函数里，我们首先要创建一个管道 `fd`，接着使用 `fork` 创建子进程。父进程会遍历 2 到 35 的数字，并将这些数字写入管道的写端。子进程则会调用 `subProcess (fd)` 函数，由这个函数来处理管道里的数据。

在 `subProcess` 函数中，会先把传入管道的写端 `oldFd [1]` 关闭，只保留读端来读取数据。之后尝试从管道读取一个整数，把这个整数当作初始的质数 `prime`。要是成功读取到数据，就会将其输出到标准输出，然后创建一个新的管道 `fd` 和一个子进程。子进程会再次递归调用 `subProcess (fd)` 函数，对新管道中的数据进行处理。

在父进程方面，会先关闭新管道的读端 `fd[0]`，然后从旧管道读取剩下的数据。对于读取到的每个数字 `num`，会检查它是否能被当前的质数 `prime` 整除，若不能整除，就把该数字写入新管道的写端 `fd[1]`。最后，父进程会关闭原始管道的读端 `oldFd[0]` 和新管道的写端 `fd[1]`，并等待子进程结束。

在 `make qemu` 后执行 `primes` 命令：

```
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$
```

查看正确得分：

```
jack@DESKTOP-EMMCGRI:/mnt/d/xv6-labs-2020$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (2.1s)
```

[实验中遇到的问题和解决方法]

在实现并发素数筛选算法时，我遇到了几个关键问题。首先是管道文件描述符的管理不当，导致进程无法正常结束。例如，在递归调用 `subProcess` 函数时，没有正确关闭所有不需要的文件描述符，使得子进程一直等待数据，最终形成僵尸进程。通过仔细检查代码，我明确了每个进程在不同阶段应关闭的文件描述符，确保数据传输结束后立即关闭相应通道。其次，数据传输格式的不一致性也引发了问题。由于没有统一指定数据大小，在读取整数时出现了错误。我通过固定使用 4 字节（即 `int` 类型大小）进行读写操作，解决了这个问题。另外，递归终止条件不明确导致程序陷入无限循环。我在 `subProcess` 函数中添加了对 `read` 返回值的检查，当返回 0 时表示数据处理完毕，及时终止递归。最后，进程等待机制不完善使得父进程提前退出，留下僵尸进程。通过在父进程中调用 `wait` 函数，确保子进程结束后再继续执行，解决了这个问题。这些问题的解决让我深刻认识

到多进程编程中资源管理和流程控制的重要性。

[实验心得]

本次实验通过实现并发素数筛选算法，让我对 Unix 管道和进程间通信有了更深入的理解。使用管道实现进程间数据传输的方式简洁高效，避免了共享内存带来的同步问题，体现了操作系统设计的精妙之处。递归算法在素数筛选中的应用非常巧妙，每个子进程专注于处理一个质数，形成了类似筛子的结构，逐步过滤出所有素数，这种设计思想值得学习。在调试过程中，我体会到多进程程序的复杂性，由于执行顺序不确定，定位问题更加困难。我学会了使用 `printf` 语句输出关键信息，跟踪程序执行流程，这对调试多进程程序非常有帮助。文件描述符的正确管理是多进程编程的关键，一个未关闭的文件描述符可能导致整个程序无法正常结束。通过这次实验，我对操作系统的进程管理和通信机制有了更直观的认识，也提升了自己的编程和调试能力。未来我将继续探索操作系统的奥秘，努力提高专业水平。

Find (moderate)

[实验目的]

Write a simple version of the UNIX find program: find all the files in a directory tree with a specific name. Your solution should be in the file `user/find.c`.

写一个简化版本的 UNIX 的 `find` 程序：查找目录树中具有特定名称的所有文件，你的解决方案应该放在 `user/find.c`。

[实验步骤]

实验代码：

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

char *pfilename(char *path)
{
    char *p;
    for (p = path + strlen(path); p >= path && *p != '/'; p--)
// 找到最后一个斜杠后的第一个字符。
    ;
}
```

```

        p++;
        return p;
    }
    int find(char *path, char *filename)
    {
        int fd;           // 存储打开的目录的文件描述符
        char buf[512], *p; // buf 用于存储当前正在检查的目录或文件的
        // 路径,p 用作遍历和操作 buf 数组
        struct stat st;    // 用于存储文件或目录的信息
        struct dirent de;  // 表示一个目录条目,用于遍历目录的内容
        if ((fd = open(path, 0)) < 0) // 如果 open 系统调用失败 (返回负值), 则会打印错误消息并退出
        {
            fprintf(2, "open fail%s\n", path);
            exit(1);
        }
        if (fstat(fd, &st) < 0) // 获取有关打开的文件或目录的信息
        {
            fprintf(2, "fstat fail%s\n", path); // 失败就打印错误消息, 关闭文件描述符, 然后退出
            close(fd);
            exit(1);
        }
        switch (st.type)
        {
            case T_FILE:           // 常规文件
                if (0 == strcmp(pfilename(path), filename)) // 比较文件名与指定的文件名,如果匹配就打印
                {
                    fprintf(1, "%s\n", path);           // 标准输出
                    break;
                }
            case T_DIR:           // 目录
                strcpy(buf, path); // 将给定路径复制到 buf 中
                p = buf + strlen(buf); // 将指针移动到 buf 的末尾
                *p++ = '/';
                while (read(fd, &de, sizeof(de)) == sizeof(de))
                {
                    // 读取目录的内容,将结果存储在 de
                    // 结构体中
                    if (de.inum == 0) // 如果读取的条目的 inum 字段为 0, 表示无效条目, 代码将继续下一次循环
                    {
                        continue;
                    }
                    // 如果读取的条目的名称是当前目录 (.) 或上级目录 (..), 代码也将继续下一次循环
                    if (0 == strcmp(".", de.name) || 0 == strcmp("..", de.name))

```

```

        continue;
        // 拼接路径, 将条目的名称复制到 buf 中。并通过调用 stat
函数获取该条目的信息存储在 st 结构体中。
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0; // 在 buf 数组中的目录项名称的末尾添加
一个空字符 (\0), 以确保字符串以空字符结尾, 从而使其成为一个有效的 C
字符串

        if (stat(buf, &st) < 0)
        {
            printf("find: cannot stat %s\n", buf);
            continue;
        }
        find(buf, filename); // 递归
    }
    break;
}
close(fd); // 关闭文件描述符, 并返回 0 表示函数执行成功结束
return 0;
}
int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        fprintf(2, "not enough arguments\n");
        exit(1);
    }
    find(argv[1], argv[2]);
    exit(0);
}

```

这段代码定义了一个递归查找文件的函数`find`，它接收两个参数：`path`代表搜索路径，`filename`代表要查找的目标文件名。其中，`pfilename`函数用于从路径中提取文件名。首先，程序会尝试打开指定路径，如果打开失败，会输出错误信息并终止。接着，调用`fstat`函数获取所打开文件或目录的状态信息，若获取失败，也会输出错误信息并终止。随后，程序通过`switch`语句判断文件类型，并执行相应操作：若为普通文件，则将其文件名与目标文件名进行比较，若匹配则输出该文件的完整路径。

若为目录，则遍历目录中的所有条目，跳过无效条目，对于有效条目，构建新路径并递归调用`find`函数继续搜索。

最后，关闭文件描述符并返回 0，表示函数成功执行完毕。在 `make qemu` 后执行相应命令：

```
jack@DESKTOP-EMMCGRI:/mnt/d/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,i
d=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
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
```

查看正确得分：

```
jack@DESKTOP-EMMCGRI:/mnt/d/xv6-labs-2020$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory:
OK (0.9s)
== Test find, recursive == find, recursive: OK (1.0s)
jack@DESKTOP-EMMCGRI:/mnt/d/xv6-labs-2020$
```

[实验中遇到的问题 and 解决方法]

在实现简化版 `find` 程序时，主要遇到三个关键问题。一是路径拼接错误，最初在处理目录条目时，直接使用 `strcat` 拼接路径导致缓冲区溢出，尤其是当原路径较长时容易超出 `buf` 数组范围。通过固定 `buf` 大小为 512 字节，用指针定位到路径末尾手动添加斜杠和目录名，并严格控制字符串结束符，解决了路径格式混乱的问题。二是递归遍历陷入死循环，因未正确过滤当前目录（`.`）和上级目录（`..`），导致程序反复访问同一目录。在读取目录条目后，通过 `strcmp` 判断并跳过这两个特殊目录，避免了无限递归。三是文件类型判断失误，最初误将设备文件等非常规文件纳入处理，导致输出异常。通过严格检查 `stat` 结构体的 `type` 字段，只处理 `T_FILE` 和 `T_DIR` 类型，确保程序只对文件和目录进行操作。这些问题的解决让我意识到，文件系统操作中细节处理的重要性，尤其是路径管理和特殊目录的过滤。

[实验心得]

本次实现 `find` 程序的实验，让我对文件系统的层次结构和目录遍历机制有了直观认识。递归算法在此处的应用尤为巧妙——通过不断深入子目录并回溯，实现了对整个目录树的完整扫描，这种思想与树的深度优先遍历异曲同工。在处理路径时，手动管理指针和缓冲区的经历，让我理解了字符串操作在系统编程中的严谨性，任何一个缺少结束符或路径分隔符的细节都可能导致程序崩溃。同时，对文件描述符的开闭管理也加深了我对系统资源的理解：每次打开目录后必须及时关闭，否则会造成资源泄漏。此外，通过对比目标文件名与路径中的文件名，我掌握了从路径中提取文件名的技巧，这对后续处理文件相关操作很有帮助。这次实验不仅锻炼了代码实现能力，更让我体会到系统工具设计中“分层处理、递归深入”的核心思路，为理解更复杂的文件系统工具奠定了基础。

Xargs (moderate)

[实验目的]

Write a simple version of the UNIX `xargs` program: read lines from the standard input and run a command for each line, supplying the line as arguments to the command. Your solution should be in the file `user/xargs.c`.

编写一个简化版 UNIX 的 `xargs` 程序：它从标准输入中按行读取，并且为每一行执行一个命令，将行作为参数提供给命令。你的解决方案应该在 `user/xargs.c`

[实验步骤]

实验代码：

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

int main(int argc, char *argv[])
{
    char *args[MAXARG]; // 保存执行的参数
    int p;
    for (p = 0; p < argc; p++) // 先把 xargs 自带的参数读进去
        args[p] = argv[p];
    char buf[256];
```

```

while (1) // 进入循环，每次读一行内容
{
    int i = 0;
    while ((read(0, buf + i, sizeof(char)) != 0) &&
buf[i] != '\n') // 读取标准输入一行的内容
        i++;
    if (i == 0) // 读完所有行
        break;
    buf[i] = 0; // 字符串结尾，exec 要求的
    args[p] = buf; // 把标准输入传进的一行参数附加到 xargs
这个函数后面
    args[p + 1] = 0; // exec 读到 0 就表示读完了
    if (fork() == 0)
    {
        // 子进程
        exec(args[1], args + 1); // 前者是 xargs，后者是参
数，执行成功会自动退出
        printf("exec err\n"); // 如果执行失败，会打印以
下信息
    }
    else
        wait((void *)0);
}
exit(0);
}

```

`xargs` 命令的作用是把标准输入转换为命令行参数。在我们当前使用的用例里，`|` 作为管道命令，能将左侧命令的标准输出转变为标准输入，供右侧命令当作参数使用。这段代码先是读取程序自身的参数（也就是从命令行输入的参数），并把它们存到 `args` 数组之中。接着，程序进入一个无限循环。在每次循环时，它会从标准输入读取一行内容（以换行符 `\n` 作为结束标志），然后将这行内容存入 `buf` 字符数组。之后，把这行内容添加到 `args` 数组里，同时将 `args` 数组的最后一个元素设为 0，这样做是为了给 `exec` 函数的参数充当结束标记。在子进程中，会调用 `exec` 函数去执行 `args[1]` 所指定的命令，并且把 `args + 1` 作为参数传递给该命令。要是 `exec` 函数执行成功，子进程就会自动退出；要是执行失败，子进程则会打印出 "exec err" 的错误信息。而在父进程中，会使用 `wait` 函数来等待子进程执行完毕。

在 `make qemu` 后执行相应命令：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ echo nice to meet u | xargs echo hi
hi nice to meet u
$
```

查看正确得分：

```
jack@DESKTOP-EMMCGRI:/mnt/d/xv6-labs-2020$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (1.4s)
```

[实验中遇到的问题和解决方法]

在我看来，本次实验的关键之处主要有两点。其一，我们得弄清楚 `xargs` 命令的具体功能。为此，我在网上查阅了相关资料，从而了解到这个指令的作用。其二，要掌握代码里 `read` 函数的用法。就拿 `read(0, buf + i, sizeof(char))` 这个语句来说：其中的 `0` 代表标准输入的文件描述符；`buf + i` 意味着从 `buf` 数组的第 `i` 个位置开始读取数据；`sizeof(char)` 表示每次读取的字节数，一般为 `1` 字节。

[实验心得]

本次实验让我深入理解了管道与命令行参数传递的机制。`xargs` 作为连接管道两端的工具，其核心在于将标准输入转化为命令参数，这一过程让我清晰看到进程间数据流转的路径：左侧命令的输出通过管道变为右侧 `xargs` 的输入，再被拆解为参数传递给目标命令。实现中，参数数组的构建尤为关键，既要保留原始命令参数，又要动态拼接输入内容，还要严格遵循 `exec` 函数的格式要求，这让我体会到系统调用对参数格式的严谨性。通过 `fork` 与 `exec` 的配合，子进程负责执行命令而父进程等待其完成，这种进程协作模式是 Unix 系统设计的精髓。实验也让我认识到细节的重要性，比如字符串结束符的设置、输入读取的边界判断，这些看似微小的细节直接影响程序的稳定性，为理解更复杂的命令行工具奠定了基础。

Lab1 的实验结果截图: ./grade-lab-util

```
jack@DESKTOP-EMMCGRI:/mnt/d/xv6-labs-2020$ ./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 (2.2s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.1s)
== Test primes == primes: OK (2.0s)
== Test find, in current directory == find, in current directory:
OK (1.0s)
== Test find, recursive == find, recursive: OK (1.0s)
== Test xargs == xargs: OK (0.9s)
== Test time ==
time: OK
Score: 100/100
```

Lab2: System Calls

首先, 在开始 Lab2 时, 需要通过切换实验分支, 获取实验资源:

git fetch

git checkout syscall

make clean

System call tracing (moderate)

[实验目的]

In this assignment you will add a system call tracing feature that may help you when debugging later labs. You'll create a new trace system call that will control tracing. It should take one argument, an integer "mask", whose bits specify which system calls to trace. For example, to trace the fork system call, a program calls `trace(1 << SYS_fork)`, where `SYS_fork` is a syscall number from `kernel/syscall.h`. You have to modify the xv6 kernel to print out a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; you don't need to print the system call arguments. The trace system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.

在本作业中, 您将添加一个系统调用跟踪功能, 该功能可能会在以后调试实

验时对您有所帮助。您将创建一个新的 `trace` 系统调用来控制跟踪。它应该有一个参数，这个参数是一个整数“掩码” (mask)，它的比特位指定要跟踪的系统调用。例如，要跟踪 `fork` 系统调用，程序调用 `trace(1 << SYS_fork)`，其中 `SYS_fork` 是 `kernel/syscall.h` 中的系统调用编号。如果在掩码中设置了系统调用的编号，则必须修改 `xv6` 内核，以便在每个系统调用即将返回时打印出一行。该行应该包含进程 `id`、系统调用的名称和返回值；您不需要打印系统调用参数。`trace` 系统调用应启用对调用它的进程及其随后派生的任何子进程跟踪，但不应影响其他进程。

[实验步骤]

根据提示，我们首先要在 `Makefile` 中添加代码：

```
$U/_trace\
```

执行 `make qemu` 时，我们发现编译器在编译 `user/trace.c` 文件时出错，原因是系统调用的用户空间存根尚未创建。为了解决这个问题，我们需要进行以下操作：

1. 在 `user/user.h` 文件中添加系统调用的原型定义。
2. 将存根信息添加到 `user/usys.pl` 文件中。
3. 把系统调用编号添加到 `kernel/syscall.h` 文件里。

这里的 `Makefile` 会调用 `user/usys.pl` 这个 `perl` 脚本，该脚本会生成实际的系统调用存根文件 `user/usys.S`。在这个文件中，汇编代码会借助 `RISC-V` 的 `ecall` 指令实现从用户态到内核态的转换。

所以我们需要加上系统调用的声明。分别在 `user.h`、`usys.pl` 和 `syscall.h` 加上以下三行代码：

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

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

```
#define SYS_trace 22
```

这个时候我们会发现 `trace` 函数是可以运行的了，但是实际上的 `trace` 函数还没有被实现，所以我们需要编写 `trace` 函数。包括在 `proc.h` 中，为 `proc` 增加一个 `mask` 变量、获取系统调用的参数和修改 `fork`，使其在 `copy` 时将 `mask` 传递。

```
uint64 sys_trace(void)
{
    argint(0, &(myproc()->mask));
    return 0;
}
```

```
extern uint64 sys_trace(void);

[SYS_trace] "trace",

np->mask = p->mask;
```

最后我们需要修改 syscall 函数，使其打印 trace 的输出。

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

    num = p->trapframe->a7;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        p->trapframe->a0 = syscalls[num]();
        if (p->mask & (1 << num))
        {
            printf("%d: syscall %s -> %d\n", p->pid, syscallname[num],
p->trapframe->a0);
        }
    }
    else
    {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}
```

即在 Makefile 的 UPROGS 中添加 \$U/_trace;

将系统调用的原型添加到 user/user.h，存根添加到 user/usys.pl，以及将系统调用编号添加到 kernel/syscall.h，Makefile 调用 perl 脚本 user/usys.pl，它生成实际的系统调用存根 user/usys.S，这个文件中的汇编代码使用 RISC-V 的 ecall 指令转换到内核；

在 kernel/sysproc.c 中添加一个 sys_trace() 函数，它通过将参数保存到 proc 结构体里的一个新变量中来实现新的系统调用。从用户空间检索系统调用参数的函数在 kernel/syscall.c 中；


```

hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
Usage: trace mask command
$ trace 32 grep hello README
4: syscall read -> 1023
4: syscall read -> 986
4: syscall read -> 95
4: syscall read -> 0
$ trace 2147483647 grep hello README
grep: cannot open README2147483647
$ trace 2147483647 grep hello README
6: syscall trace -> 0
6: syscall exec -> 3
6: syscall open -> 3
6: syscall read -> 1023
6: syscall read -> 986
6: syscall read -> 95
6: syscall read -> 0
6: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
8: syscall fork -> 9
test forkforkfork: 8: syscall fork -> 10
10: syscall fork -> 11
11: syscall fork -> 12
11: syscall fork -> 13
12: syscall fork -> 14
11: syscall fork -> 15
11: syscall fork -> 16
12: syscall fork -> 17
11: syscall fork -> 18
11: syscall fork -> 19
12: syscall fork -> 20
11: syscall fork -> 21
12: syscall fork -> 22
11: syscall fork -> 23
12: syscall fork -> 24
13: syscall fork -> 25

```

修改 fork() 将跟踪掩码从父进程复制到子进程；

修改 kernel/syscall.c 中的 syscall() 函数以打印跟踪输出。

在 make qemu 后执行相应命令如右图：

全部执行后：

```

12: syscall fork -> 69
12: syscall fork -> 70
11: syscall fork -> -1
12: syscall fork -> -1
13: syscall fork -> -1
OK
8: syscall fork -> 71
ALL TESTS PASSED

```

[实验中遇到的问题解决方法]

本次实验是我首次完整实现系统调用，整体难度适中。操作时需严格遵循指导步骤，关键在于仔细核对每一个添加项，确保没有遗漏任何必要的代码或配置。

通过按部就班地执行流程，最终顺利完成了实验任务。

[实验心得]

通过本次实验，我深入理解了系统调用的实现机制，尤其是系统调用追踪技术的原理和应用。从代码层面剖析系统调用的执行流程，让我对操作系统内核与用户程序之间的交互有了更直观的认识。系统调用追踪技术不仅帮助我监控程序运行时的系统调用行为，还为后续调试和优化工作提供了有力的工具支持。

Sysinfo (moderate)

[实验目的]

In this assignment you will add a system call, `sysinfo`, that collects information about the running system. The system call takes one argument: a pointer to a struct `sysinfo` (see `kernel/sysinfo.h`). The kernel should fill out the fields of this struct: the `freemem` field should be set to the number of bytes of free memory, and the `nproc` field should be set to the number of processes whose state is not `UNUSED`. We provide a test program `sysinfotest`; you pass this assignment if it prints "sysinfotest: OK".

在这个作业中，您将添加一个系统调用 `sysinfo`，它收集有关正在运行的系统的信息。系统调用采用一个参数：一个指向 `struct sysinfo` 的指针（参见 `kernel/sysinfo.h`）。内核应该填写这个结构的字段：`freemem` 字段应该设置为空闲内存的字节数，`nproc` 字段应该设置为 `state` 字段不为 `UNUSED` 的进程数。我们提供了一个测试程序 `sysinfotest`；如果输出“`sysinfotest: OK`”则通过。

[实验步骤]

根据提示，我们首先要在 `Makefile` 中添加代码：

```
$U/_sysinfotest
```

同样的，运行 `make qemu`，我们看到编译器无法编译，因为系统调用的用户空间存根还不存在：所以我们需要在 `kernel/user.h` 中提前声明 `struct sysinfo`，在 `usys.pl` 中加上 `entry`，在 `syscall.h` 中加上 `trace` 的编号。

```

struct sysinfo;
int sysinfo(struct sysinfo *);

entry("sysinfo");

#define SYS_sysinfo 23

```

接下来，我们要实现 sys_info 函数的具体功能。具体步骤如下：

1. 首先获取用户传递的地址。
2. 定义并填充 sysinfo 结构，获取系统的空闲内存和当前运行的进程数。
3. 将这些信息从内核空间复制到用户空间。

若上述任何步骤失败，返回-1 表示错误；若全部成功则返回 0。

```

uint64 sys_sysinfo(void)
{
    // 获取用户传递的地址
    uint64 addr;
    if (argaddr(0, &addr) < 0)
        return -1;

    // 定义并填充 sysinfo 结构
    struct sysinfo info;
    freememory(&info.freemem);
    procnum(&info.nproc);
    // 将信息从内核空间复制到用户空间
    if (copyout(myproc()->pagetable, addr, (char *)&info, sizeof
info) < 0)
        return -1;
    return 0;
}

```

也就是要在 Makefile 的 UPROGS 里加上 \$U/_sysinfotest。运行 make qemu 时，user/sysinfotest.c 会编译失败，这时需按照和上一个作业相同的步骤添加 sysinfo 系统调用：要在 user/user.h 中声明 sysinfo() 的原型，且需预先声明 struct sysinfo 的存在；sysinfo 需要把一个 struct sysinfo 复制回用户空间；为了获取空闲内存量，要在 kernel/kalloc.c 中添加一个函数；为了获取进程数，要在 kernel/proc.c 中添加一个函数。

在 make qemu 后执行相应命令：

```
xv6 kernel is booting

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

[实验中遇到的问题和解决方法]

基于上一个实验的经验，本次实验的整体流程对我而言并不陌生。核心任务在于实现 `sys_info` 函数的具体功能，这主要涉及三个关键函数的编写：

获取空闲内存：调用 `freememory(&info.freemem)` 获取系统剩余内存，并将结果存储在 `info.freemem` 中。

统计进程数量：通过 `procnum(&info.nproc)` 获取当前运行的进程数，结果存入 `info.nproc`。

数据跨空间复制：使用 `copyout(myproc()->pagetable, addr, (char *)&info, sizeof info)` 将内核空间的信息复制到用户空间。其中：

`myproc()->pagetable`：获取当前进程的页表

`addr`：用户空间目标地址

`(char *)&info`：待复制数据的起始地址

`sizeof info`：数据大小

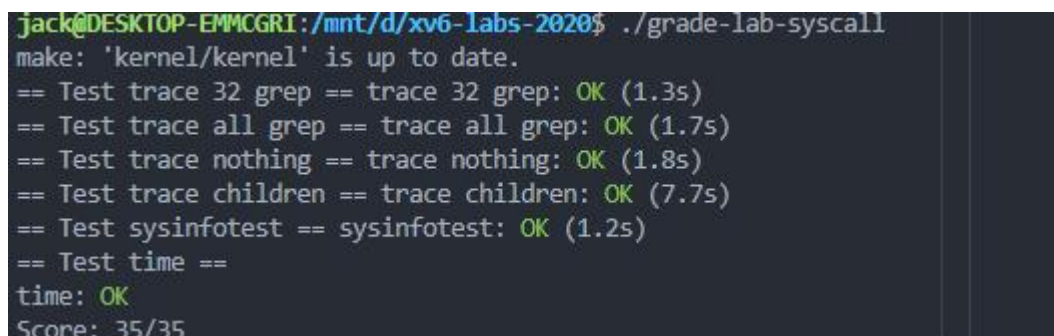
在实现过程中，我参考了 `sysfile.c` 和 `file.c` 的代码模式，通过分析这些系统组件的实现方式，逐步理解了跨空间数据传输的机制。当遇到复制失败的情况时，通过返回 -1 进行错误处理，确保系统调用的健壮性。

[实验心得]

在本次实验中，我掌握了在 `xv6` 操作系统里新增系统调用的方法。整个实现过程分为四个关键步骤：首先，在 `'user/user.h'` 头文件中定义系统调用在用户态的函数原型，为用户程序提供调用接口；其次，在同级目录的 `'usys.pl'` 脚本里添加汇编代码，负责处理从用户态到内核态的转换逻辑；接着，在 `'kernel/syscall.h'` 中为新系统调用分配唯一编号，确保内核能够识别该调用；最后，在内核代码层面实现系统调用的具体功能，涵盖设计相关数据结构以及实现依赖函数等工作。通过完成这次实验，我对 `xv6` 内核的架构和工作机制进行了

深入研究，特别是对系统调用的实现原理和用户态/内核态交互过程有了更清晰、透彻的认识。

Lab2 的实验结果截图：./grade-lab-syscall



```
jack@DESKTOP-EMMCGRI:/mnt/d/xv6-labs-2020$ ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.3s)
== Test trace all grep == trace all grep: OK (1.7s)
== Test trace nothing == trace nothing: OK (1.8s)
== Test trace children == trace children: OK (7.7s)
== Test sysinfotest == sysinfotest: OK (1.2s)
== Test time ==
time: OK
Score: 35/35
```

Lab3:Page Tables

首先，开始 Lab3 时，需要通过以下命令切换实验分支，获取实验资源：

git fetch

git checkout pgtbl

make clean

Print a page table (easy)

[实验目的]

Define a function called `vmprint()`. It should take a `pagetable_t` argument, and print that pagetable in the format described below. Insert `if(p->pid==1)` `vmprint(p->pagetable)` in `exec.c` just before the `return argc`, to print the first process's page table. You receive full credit for this assignment if you pass the `pte printout` test of `make grade`.

定义一个名为 `vmprint()` 的函数。它应当接收一个 `pagetable_t` 作为参数，并以下面描述的格式打印该页表。在 `exec.c` 中的 `return argc` 之前插入 `if(p->pid==1)` `vmprint(p->pagetable)`，以打印第一个进程的页表。如果你通过了 `pte printout` 测试的 `make grade`，你将获得此作业的满分。

[实验步骤]

根据提示，我们首先需要在 `def.h` 中声明 `vmprint()` 函数，这样就可以在文件 `exec.c` 中调用。

```
void vmprint(pagetable_t pagetable);
```

之后我们就可以在 kernel/vm.c 中编写 vmprint 函数了，代码如下：

```
// 递归打印页表的内容
void printwalk(pagetable_t pagetable, int depth)
{
    // 一个页表中有 2^9 = 512 个页表项
    for (int i = 0; i < 512; i++)
    {
        pte_t pte = pagetable[i];
        if (pte & PTE_V)
        {
            for (int j = 0; j < depth; j++) // 打印深度缩进
            {
                printf("..");
                if (j != depth - 1)
                    printf(" ");
            }
            // 这页表项指向一个更低级的页表
            uint64 child = PTE2PA(pte);
            printf("%d: pte %p pa %p\n", i, pte, child);

            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0)
            {
                printwalk((pagetable_t)child, depth + 1);
            }
        }
    }
}

// 打印整个页表的内容
void vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable);
    printwalk(pagetable, 1);
}
```

这段代码包含两个核心函数，用于实现页表的递归打印功能：

vmprint 函数：作为页表打印的入口，该函数首先输出页表的起始地址，随后调用 printwalk 函数，启动对整个页表结构的递归遍历过程。

printwalk 函数：采用递归策略遍历并输出多级页表的内容，核心逻辑如下：

遍历当前页表的每个表项（PTE）

若表项有效（通过 pte & PTE_V 判断）：

按当前递归深度输出缩进格式

将表项转换为物理地址并打印

若表项不指向物理页（即无 R/W/X 权限）：

递归调用自身处理下一级页表

这种实现方式体现了多级页表的层次化遍历思想，通过递归方式自动处理任意深度的页表结构，确保完整展示整个虚拟地址空间到物理地址的映射关系。

运行 `make qemu` 指令便会得到以下结果：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f63000
..0: pte 0x0000000021fd7c01 pa 0x0000000087f5f000
.. ..0: pte 0x0000000021fd7801 pa 0x0000000087f5e000
.. .. ..0: pte 0x0000000021fd801f pa 0x0000000087f60000
.. .. ..1: pte 0x0000000021fd740f pa 0x0000000087f5d000
.. .. ..2: pte 0x0000000021fd701f pa 0x0000000087f5c000
..255: pte 0x0000000021fd8801 pa 0x0000000087f62000
.. ..511: pte 0x0000000021fd8401 pa 0x0000000087f61000
.. .. ..510: pte 0x0000000021fed807 pa 0x0000000087fb6000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$
```

[实验中遇到的问题解决方法]

这个小实验在算是比较简单的。凭借已学的操作系统理论知识，我们很容易想到用递归的方式来实现打印功能。所以在代码编写和逻辑理解上，并没有遇到太大的困难。

[实验心得]

本次实验让我对多级页表的结构和地址映射机制有了直观认识。通过递归遍历页表的实现，我深刻体会到虚拟地址空间的层次化设计：页表项通过权限位区分功能（中间节点或叶节点），各级页表逐级索引，最终完成虚拟地址到物理地址的映射。这种设计不仅高效利用了内存（无需为整个地址空间分配连续页表），也为地址转换提供了清晰的逻辑路径。在代码实现中，递归思想的应用尤为巧妙——通过函数自调用自然适配页表的多级结构，无需手动处理固定级数的嵌套，

体现了程序设计对数据结构的灵活适配。同时，对页表项标志位的解析让我认识到，这些看似微小的位运算（如 PTE_V 判断有效性、PTE_R/W/X 区分类型）是操作系统控制内存访问的核心手段。这次实验不仅锻炼了对复杂数据结构的遍历能力，更加深了对内存管理底层机制的理解，为后续学习虚拟内存相关知识奠定了基础。

通过参考 xv6 books，我可以更好的理解 xv6 三级页表的逻辑如下：

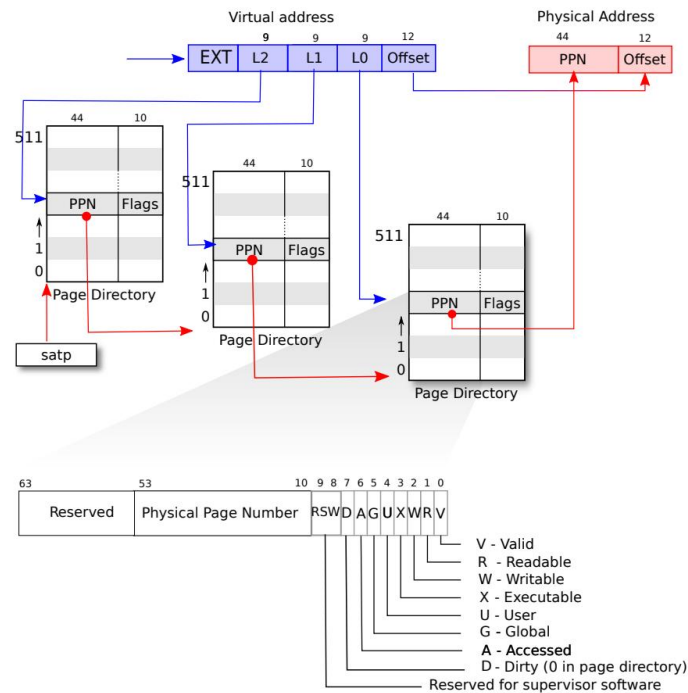


Figure 3.2: RISC-V address translation details.

A kernel page table per process (hard)

[实验目的]

Your first job is to modify the kernel so that every process uses its own copy of the kernel page table when executing in the kernel. Modify struct proc to maintain a kernel page table for each process, and modify the scheduler to switch kernel page tables when switching processes. For this step, each per-process kernel page table should be identical to the existing global kernel page table. You pass this part of the lab if usertests runs correctly.

你的第一项工作是修改内核来让每一个进程在内核中执行时使用它自己的

内核页表的副本。修改 `struct proc` 来为每一个进程维护一个内核页表，修改调度程序使得切换进程时也切换内核页表。对于这个步骤，每个进程的内核页表都应当与现有的全局内核页表完全一致。如果你的 `usertests` 程序正确运行了，那么你就通过了这个实验。

[实验步骤]

我们的任务是修改内核来让每一个进程在内核中执行时使用它自己的内核页表的副本，所以首先需要在 `kernel/proc.h` 中，给 `proc` 增加内核页表。

```
pagetable_t kernelpgtbl;    // add kernel page table
```

接下来，我们对 `kvminit` 函数进行重构。原函数的逻辑是直接初始化并映射全局变量 `kernel_pagetable`。现在我们将这部分功能抽取出来，单独创建一个 `kvm_init_one` 函数，专门用于初始化单个内核页表。然后在 `kvminit` 函数中调用这个新函数，并将返回的页表赋值给 `kernel_pagetable`。

重构后的代码结构更加清晰，将页表初始化的核心逻辑封装在独立的函数中，提高了代码的复用性和可维护性。这样一来，`kvminit` 函数只需关注整体流程的控制，而具体的页表初始化工作则由 `kvm_init_one` 函数负责实现。

```
void      kvmmap_with_certain_page(pagetable_t pg, uint64 va, uint64 pa, uint64 sz, int perm);
pagetable_t  kvm_init_one();

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

在 `kernel/vm.c` 下实现以下函数：

```
pagetable_t kvm_init_one()
{
    pagetable_t newpg = uvmcreate();

    // copy but forget to modify the first argument. so sad
    // uart registers
    kvmmap_with_certain_page(newpg, UART0, UART0, PGSIZE, PTE_R
| PTE_W);
    // virtio mmio disk interface
    kvmmap_with_certain_page(newpg, VIRTIO0, VIRTIO0, PGSIZE,
PTE_R | PTE_W);
    // CLINT
    kvmmap_with_certain_page(newpg, CLINT, CLINT, 0x10000,
PTE_R | PTE_W);
```

```

    // PLIC
    kvmmap_with_certain_page(newpg, PLIC, PLIC, 0x400000, PTE_R
| PTE_W);
    // map kernel text executable and read-only.
    kvmmap_with_certain_page(newpg, KERNBASE, KERNBASE,
(uint64)etext - KERNBASE, PTE_R | PTE_X);
    // map kernel data and the physical RAM we'll make use of.
    kvmmap_with_certain_page(newpg, (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_with_certain_page(newpg, TRAMPOLINE,
(uint64)trampoline, PGSIZE, PTE_R | PTE_X);
    return newpg;
}
/*
 * create a direct-map page table for the kernel.
 */
void kvminit()
{
    kernel_pagetable = kvm_init_one();
}

```

记得在 vm.c 中添加头文件。

```

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

```

在原先 xv6 中，所有内核栈均设置在`procinit`函数中初始化，为实现本实验功能，所以我们需将初始化移动到`allocproc`中，并调用刚才写的函数。

在 proc.c 中将 procinit 中的以下代码注释：

```

// 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;

```

之后在 allocproc 中添加以下代码：

```

// add kernel page table
p->kernelpgtbl = kvm_init_one();
if (p->kernelpgtbl == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
}

```

```

    }
    // init in allocproc
    // Allocate a page for the process's kernel stack.
    // Map it high in memory, followed by an invalid
    // guard page.
    char *pa = kalloc();
    if(pa == 0)
        panic("kalloc");
    uint64 va = KSTACK((int)(p - proc));
    kvmmap_with_certain_page(p->kernelpgtbl, va, (uint64)pa,
PGSIZE, PTE_R | PTE_W);
    p->kstack = va;

```

修改 freeproc 函数，使其能正确释放内核页表和内核栈：

```

if (p->kstack){
    pte_t* pte = walk(p->kernelpgtbl, 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;
if (p->kernelpgtbl){
    kvm_free_pgtbl(p->kernelpgtbl);
}
p->kernelpgtbl = 0;
// free pg recursively
void kvm_free_pgtbl(pagetable_t pg){
    for (int i = 0; i < 512; i++){
        pte_t pte = pg[i];
        // copy wrong!!
        // if((pte & PTE_V) && (PTE_R|PTE_W|PTE_X) == 0){
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
            uint64 child = PTE2PA(pte);
            kvm_free_pgtbl((pagetable_t)child);
            pg[i] = 0;
        }
    }
}
kfree((void*)pg);
}

```

最后修改 kvmpa，将默转换的内核页表替换为正在运行的进程，用于查找给

定虚拟地址(va)在内核页表中的对应页表项(pte)。

```
pte_t *pte;
uint64 pa;

pte = walk(myproc()->kernelpgtbl, va, 0);
if(pte == 0)
    panic("kvmpa");
```

我们可以在终端运行 usertests，可以得到以下正确结果：

```
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=
6265
          sepc=0x00000000000003e7a stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=
=6269
          sepc=0x00000000000002188 stval=0x000000000000fbc0
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test 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
```

显示 ALL TESTS PASSED，我们完成了内核页表的修改流程。

[实验中遇到的问题解决方法]

在实现进程专属内核页表的过程中，主要遇到四个关键问题。一是内核页表映射不完整，最初 kvm_init_one 函数未正确复制全局内核页表的所有映射项，导致外设（如 UART、PLIC）或内核代码区映射缺失，系统启动后出现设备访问错误。通过逐一核对全局页表的映射范围（从外设地址到内核代码段、栈区），

确保 `kvm_init_one` 中每个映射项与原全局页表一致，解决了映射不全的问题。二是内核栈虚拟地址冲突，由于将内核栈初始化移至 `allocproc` 后，未正确计算每个进程的 `KSTACK` 虚拟地址，导致多个进程的内核栈映射到同一虚拟地址，引发内存访问冲突。通过严格按进程索引计算 `KSTACK` 值，并在 `allocproc` 中为每个进程单独映射内核栈到其私有内核页表，确保了栈地址的唯一性。三是进程切换时页表未同步，调度器最初未切换内核页表，导致进程运行在内核态时仍使用前一个进程的页表，引发地址翻译错误。在 `swtch` 上下文切换时，通过修改 `satp` 寄存器加载当前进程的 `kernelpgtbl`，实现了内核页表的动态切换。四是页表资源泄漏，`freeproc` 函数初期未递归释放内核页表的多级页表项，仅释放顶级页表导致内存泄漏。通过实现 `kvm_free_pgtbl` 递归遍历并释放所有级别的页表页，同时释放内核栈的物理内存，解决了资源泄漏问题。

[实验心得]

本次实验让我深刻理解了内核页表私有化对进程隔离的意义。通过为每个进程创建独立的内核页表，不仅强化了进程间的内存隔离（即使在内核态也无法访问其他进程的内核栈），也为后续实现更精细的内存权限控制奠定了基础。实验中，内核页表的“复制 - 私有映射”思路尤为关键：通过 `kvm_init_one` 复制全局映射确保系统基础功能正常，同时为每个进程单独映射内核栈实现隔离，这种“共享基础映射 + 私有资源隔离”的设计平衡了效率与安全性。在调试过程中，对页表切换时机的把握让我意识到：进程切换不仅涉及用户页表的切换，内核页表的同步同样重要，两者共同构成了进程上下文的完整切换。此外，资源释放的递归处理（如 `kvm_free_pgtbl`）体现了多级页表的树形结构特性，也让我更深刻地理解了内存管理中“分配 - 释放”对称设计的重要性。这次实验不仅锻炼了内核代码的修改能力，更让我对操作系统的进程隔离机制有了从抽象到具体的认知。

Simplify copyin/copyinstr (hard)

[实验目的]

Replace the body of `copyin` in `kernel/vm.c` with a call to `copyin_new` (defined in `kernel/vmcopyin.c`); do the same for `copyinstr` and `copyinstr_new`. Add mappings for user addresses to each process's kernel page table so that `copyin_new` and

copyinstr_new work. You pass this assignment if usertests runs correctly and all the make grade tests pass.

将定义在 kernel/vm.c 中的 copyin 的主题内容替换为对 copyin_new 的调用(在 kernel/vmcopyin.c 中定义); 对 copyinstr 和 copyinstr_new 执行相同的操作。为每个进程的内核页表添加用户地址映射, 以便 copyin_new 和 copyinstr_new 工作。如果 usertests 正确运行并且所有 make grade 测试都通过, 那么就完成了此项作业。

[实验步骤]

在 xv6 系统中, 为了让已进入内核的进程能更便捷地获取用户态的地址空间, 专门设计了一套机制: 将用户态的页表整合到内核页表当中。这样一来, 当处于内核态的进程需要查询用户态的某个地址空间时, 就不必再切换回用户态, 直接在内核环境里就能完成地址的查询与翻译工作。这一设计有效减少了用户态与内核态之间频繁切换所产生的开销。

首先在 kernel/defs.h 中声明函数 uvm2kvm。

既然声明了 uvm2kvm 函数, 我们就需要在 kernel/vm.c 中实现`uvm2kvm`函数, 将用户页表转换到内核页表。注意 PLIC 限制, 同时将 PTE_U 设为 0。

```
void uvm2kvm(pagetable_t userpgtbl, pagetable_t kernelpgtbl,
uint64 from, uint64 to)
{
    if (from > PLIC) // PLIC limit
        panic("uvm2kvm: from larger than PLIC");
    from = PGROUNDDOWN(from); // align
    for (uint64 i = from; i < to; i += PGSIZE)
    {
        pte_t *pte_user = walk(userpgtbl, i, 0);
        pte_t *pte_kernel = walk(kernelpgtbl, i, 1);
        if (pte_kernel == 0)
            panic("uvm2kvm: allocating kernel pagetable fails");
        *pte_kernel = *pte_user;
        *pte_kernel &= ~PTE_U;
    }
}
```

根据实验主页的提示, 我们需要把 kernel/vm.c 中的 copyin()和 copyinstr()替换成 kernel/vmcopyin.c 的 copyin_new()和 copyinstr_new()。

```

int      copyin_new(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len);
int      copyinstr_new(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max);
int copyin(pagetable_t pagetable, char *dst, uint64 srcva,
uint64 len){
    return copyin_new(pagetable, dst, srcva, len);
}
int copyinstr(pagetable_t pagetable, char *dst, uint64 srcva,
uint64 max){
    return copyinstr_new(pagetable, dst, srcva, max);
}

```

在 xv6 操作系统中, `fork()`, `exec()`, `sbrk()` 等系统调用会改变用户态的内存映射, 因此需要对这些系统调用进行相应的更改, 以确保它们在每个进程独立的内核页表实现中正确工作。

`fork()`:

```

np->sz = p->sz;
uvm2kvm(np->pagetable, np->kernelpgtbl, 0, np->sz);
np->parent = p;

```

`exec()`:

```

// add vmprint
if(p->pid==1) vmprint(p->pagetable);
uvm2kvm(p->pagetable, p->kernelpgtbl, 0, p->sz);
return argc; // this ends up in a0, the first argument to
main(argc, argv)

```

`userinit()`:

```

uvminit(p->pagetable, initcode, sizeof(initcode));
p->sz = PGSIZE;
uvm2kvm(p->pagetable, p->kernelpgtbl, 0, p->sz); // copy from
user to kernel
// prepare for the very first "return" from kernel to user.
p->trapframe->epc = 0;      // user program counter

```

`growproc()`:

```

} else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
}
uvm2kvm(p->pagetable, p->kernelpgtbl, sz - n, sz);
p->sz = sz;
return 0;

```

[实验中遇到的问题 and 解决方法]

本次实验颇具挑战性, 主要工作集中在替换两个关键函数以及修改相关依赖。我通过参考指导视频逐步完成了实验, 但仍需在课后深入理解和记忆相关知识点。

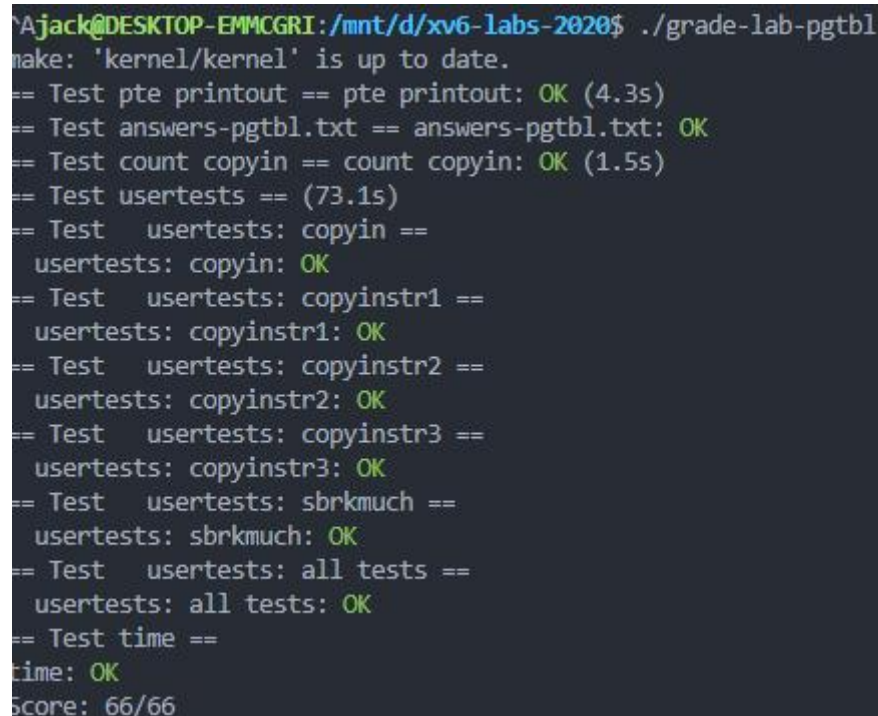
在测试阶段，使用 `./grade-lab-pgtbl` 命令时遇到了 `Test answers-pgtbl.txt FAIL: Cannot read answers-pgtbl.txt` 的错误。通过查阅资料发现，需要手动创建 `answers-pgtbl.txt` 文件并添加相应的文本内容，才能顺利通过测试。这一过程让我意识到实验文档阅读和环境配置细节的重要性。

[实验心得]

本次实验的流程复杂，核心任务是实现两个用于用户空间与内核空间数据交互的函数。其中，`copyin` 函数负责将用户空间的数据复制到内核空间，而 `copyinstr` 函数则专门处理以空字符结尾的字符串复制。

在实现过程中，我对这两个函数进行了简化和优化，这不仅加深了我对操作系统内核内存管理机制和页表工作原理的理解，还积累了宝贵的系统级编程经验。通过处理用户空间到内核空间的数据传输问题，我更加清晰地认识到内核开发中内存访问安全的重要性。

Lab3 的实验结果截图： `./grade-lab-pgtbl`



```
Ajack@DESKTOP-EMMCGRI:/mnt/d/xv6-labs-2020$ ./grade-lab-pgtbl
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (4.3s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin == count copyin: OK (1.5s)
== Test usertests == (73.1s)
== 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
```

Lab4:Traps

首先，开始 Lab4 时，需要通过以下命令切换实验分支，获取实验资源：

```
git fetch
```

```
git checkout traps
```

```
make clean
```

RISC-V assembly (easy)

[实验目的]

Explore how to implement system calls with traps. You will first warm up the stack and then implement an example of user-level trap processing; understand the basic concepts of the RISC-V assembly by reading the user/call.c file in the xv6 warehouse.

探讨如何使用陷阱实现系统调用。您将首先对堆栈进行热身练习，然后实现用户级陷阱处理的示例；通过阅读 xv6 仓库中的 user/call.c 文件，理解 RISC-V 汇编的基本概念。

[实验步骤]

我们首先使用 `make fs.img` 指令编译文件 `user/call.c`，生成可读的汇编程序文件 `user/call.asm`，如下图：

```
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ make fs.img
make: 'fs.img' is up to date.
```

在 `call.asm` 中可以找到函数 `g`、`f` 和 `main` 的代码：

```
int g(int x) {
    0: 1141          addi sp,sp,-16
    2: e422          sd s0,8(sp)
    4: 0800          addi s0,sp,16
    return x+3;
}
    6: 250d          addiw a0,a0,3
    8: 6422          ld s0,8(sp)
    a: 0141          addi sp,sp,16
    c: 8082          ret
```

```

int f(int x) {
    e: 1141                addi  sp,sp,-16
    10: e422                sd   s0,8(sp)
    12: 0800                addi  s0,sp,16
    return g(x);
}
    14: 250d                addiw a0,a0,3
    16: 6422                ld   s0,8(sp)
    18: 0141                addi  sp,sp,16
    1a: 8082                ret

```

```

void main(void) {
    1c: 1141                addi  sp,sp,-16
    1e: e406                sd   ra,8(sp)
    20: e022                sd   s0,0(sp)
    22: 0800                addi  s0,sp,16
    printf("%d %d\n", f(8)+1, 13);
    24: 4635                li   a2,13
    26: 45b1                li   a1,12
    28: 00000517            auipc a0,0x0
    2c: 7b850513            addi  a0,a0,1976 # 7e0 <malloc+0x102>
    30: 00000097            auipc ra,0x0
    34: 5f6080e7            jalr  1526(ra) # 626 <printf>
    exit(0);
    38: 4501                li   a0,0
    3a: 00000097            auipc ra,0x0
    3e: 274080e7            jalr  628(ra) # 2ae <exit>
}

```

回答以下问题:

(1) Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf? 哪些寄存器保存函数的参数? 例如, 在 main 对 printf 的调用中, 哪个寄存器保存 13?

参数通过寄存器 a0 至 a7 来传递, 依次对应第 1 到第 8 个参数。24: “4635 li a2,13” 的含义是: 把立即数 13 载入到寄存器 a2 中, 此时寄存器 a2 中存储的就是 13。

(2) Where is the call to function f in the assembly code for main? Where is the call to g? (Hint: the compiler may inline functions.) 在 main 的汇编代码中, 对函数 f 的调用在哪里? 对 g 的调用在哪里? (提示: 编译器可以内联函数)

没有这样的汇编代码，因为函数 h 被内联到函数 f 中，函数 f 又进一步被内联到 main 函数中。

(3)At what address is the function printf located?函数 printf 位于哪个地址？

以下这段代码：

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

跳转到 ra+1536 的位置，此时 ra 的值为 pc 的值，就是说 printf 的地址为 0x30+1536=0x630。

(4)What value is in the register ra just after the jalr to printf in main?在 main 中 jalr 到 printf 后，ra 寄存器的值多少？

根据 jalr 指令功能，在跳转后 ra 的值为 pc+4=0x34+4=0x38。

(5)Run the following code.

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

What is the output?If the RISC-V were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

运行下面的代码,回答问题

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

输出是什么？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？

57616 转为十六进制是 e110。而&i 所指向的字符串是 "rld"。输出是 He110 World；那就将 i 设置为 0x726c6400；但是 57616 不需要修改。

%x 表示以十六进制数形式输出整数，57616 的 16 进制表示是 e110，与大小端序无关。%s 是输出字符串，意思是以整数 i 所在的开始地址，按照字符的格式读取字符，直到读取到 ‘\0’ 为止。用小端序表示的时候，内存中存放的数是：72 6c 64 00，刚好对应 rld。当是大端序的时候就反过来了，因此需要将 i 以 16 进制数的方式逆转一下。

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

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

在下面的代码中，之后会打印什么 'y='?(注：答案不是具体值。)为什么会出现这种情况？

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

y 输出的是一个受调用前代码影响的“随机”的值，因为 `printf` 尝试读的参数数量比提供的参数数量多，第二个参数 '3' 通过 `a3` 传递，而第三个参数对应的寄存器 `a2` 在调用前不会被设置为任何具体的值，而是会包含调用发生的任何已经在里面的值。

[实验中遇到的问题和解决方法]

本次实验过程中，一个主要的挑战在于对汇编代码的阅读与理解。汇编语言作为一种低级语言，其语法结构、指令含义以及寄存器操作方式都与高级编程语言存在显著差异，这使得初学者在面对汇编代码时往往会感到晦涩难懂，难以快速把握代码的逻辑流程和执行意图。不过，这一难点并非无法攻克。在实际解决问题的过程中，将查阅到的资料与具体的实验代码相结合，逐行分析指令的作用，追踪数据在寄存器和内存中的流转过程，逐步梳理出代码的执行逻辑。通过这样的方式，实验中遇到的汇编代码理解难题便可以得到有效的解决，从而顺利推进实验的进行。

[实验心得]

本次实验让我深入理解了 RISC-V 架构下函数调用的底层机制。通过分析汇编代码，直观看到函数参数通过 `a0-a7` 寄存器传递、返回地址由 `ra` 寄存器保存、跳转指令 `jlr` 的工作原理，这些细节让“函数调用”从抽象概念变为具体的指令序列。内联函数的处理方式也给我留下深刻印象 —— 编译器通过直接嵌入函数体减少调用开销，体现了代码优化与底层实现的关联。大小端序的对比实验则揭示了数据存储方式对程序行为的影响，尤其是在字符串处理中，字节序直接决定了内存数据到字符的映射关系。此外，`printf` 函数参数数量不匹配时的“随机值”现象，让我认识到 C 语言的弱类型检查特性，以及寄存器状态对程序输出的潜在影响。这次实验不仅巩固了汇编语言基础，更建立了高级语言与底层硬件执行的联系，为理解后续陷阱与系统调用机制奠定了基础。

Backtrace (moderate)

[实验目的]

For debugging it is often useful to have a backtrace: a list of the function calls on the stack above the point at which the error occurred. This experiment is designed to implement the `backtrace()` function in `kernel/printf.c`. Insert the call to this function in `sys_sleep()` and then run `bttest`, which calls `sys_sleep`. Write the function `backtrace()`, read the stack frame (frame pointer) and output the function to return the address.

对于调试，有一个回溯通常是有用的：在错误发生点上方的堆栈上调用的函数列表。本实验要在`kernel/printf.c`中实现`backtrace()`函数。在`sys_sleep()`中插入对这个函数的调用，然后运行`bttest`，它调用`sys_sleep`。编写函数`backtrace()`，遍历读取栈帧(frame pointer)并输出函数返回地址。

[实验步骤]

在 `kernel/defs.h` 中添加 `backtrace` 的原型，那样我们就能够在 `sys_sleep` 中引用 `backtrace`，在 `kernel/defs.h` 添加如下代码：

```
// printf.c
void printf(char *, ...);
void panic(char *) __attribute__((noreturn));
void printfinit(void);
void backtrace(void);
```

在 `printf.c` 文件中实现该函数的，所以添加的原型应该放在`//printf.c` 这段注释下实现。同时 GCC 编译器将当前正在执行的函数的帧指针保存在 `s0` 寄存器，将下面的函数添加到 `kernel/riscv.h`：

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

我们要在 kernel/sysproc.c 中的 sys_sleep()函数中使用 backtrace()函数，所以要在 kernel/sysproc.c 中加入使用这段函数的代码，如图所示：

```
uint64
sys_sleep(void)
{
    int n;
    uint ticks0;
    backtrace();
    if (argint(0, &n) < 0)
        return -1;
    acquire(&tickslock);
```

接下来我们在 kernel/printf.c 中实现 backtrace 的具体逻辑的编写了，具体的实现代码如下：

```
void backtrace(void)
{
    printf("backtrace:\n");
    uint64 fp = r_fp(); // 返回当前的帧指针
    while (PGROUNDDOWN(fp) < PGROUNDUP(fp)) // 只要当前帧指针 fp
    在同一个页面内，就继续循环。
    {
        printf("%p\n", *(uint64 *) (fp - 8));
        fp = *(uint64 *) (fp - 16); // 更新 fp 为前一个栈帧的帧指针
    }
}
```

我们需要实现的 backtrace 函数，其功能是遍历并打印程序当前的调用栈（也就是函数调用的历史记录），以此辅助程序调试。具体实现步骤如下：首先，获取当前的帧指针 fp；接着，设置循环条件以确保 fp 处于同一页面内，这是为了防止因跨页面访问而导致错误；最后，打印当前栈帧的返回地址。*(uint64 *) (fp - 8)通过解引用帧指针前 8 个字节的位置获取返回地址；更新帧指针为前一个栈帧的帧指针。*(uint64 *) (fp - 16)通过解引用帧指针前 16 个字节的位置获取上一个栈帧的帧指针。

make qemu 之后输入 bttest 命令，成功输出实验结果；按照指导提示完成命令的检测，输出的图片如图，测试成功：


```

hart 2 starting
hart 1 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002d24
0x0000000080002bfe
0x00000000800028b4
axQEMU: Terminated
a jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ addr2line -e kernel/kernel
^Z
[2]+  Stopped                  addr2line -e kernel/kernel
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ addr2line -e kernel/kernel 0x0000000080002d
24
/home/jack/xv6-labs-2020/kernel/sysproc.c:62
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ addr2line -e kernel/kernel 0x0000000080002d
24 0x0000000080002bfe 0x00000000800028b4
/home/jack/xv6-labs-2020/kernel/sysproc.c:62
/home/jack/xv6-labs-2020/kernel/syscall.c:140
/home/jack/xv6-labs-2020/kernel/trap.c:78
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$

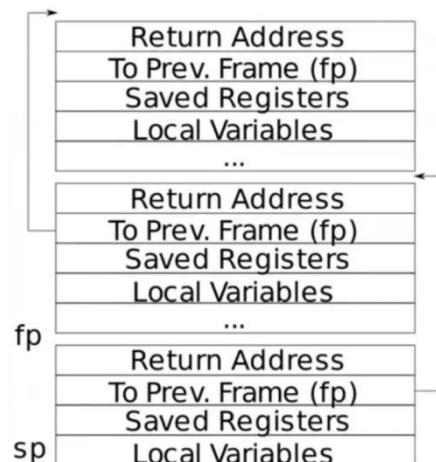
```

[实验中遇到的问题解决方法]

我觉得这次实验的问题主要是是返回地址与前帧指针的偏移计算错误，初期混淆了栈帧中返回地址和前帧指针的存储位置，导致打印的地址无效。通过查阅 RISC-V 栈帧结构，明确返回地址位于 `fp-8` 处，前帧指针位于 `fp-16` 处，修正了地址偏移量，最后成功完成实验。

[实验心得]

这次实验使我对函数调用栈及栈帧结构的原理有了更为深刻的理解。在实现 `backtrace` 函数的过程中，我掌握了借助帧指针(`fp`)来回溯函数调用历史的方法。尤其是在解析 `fp-8` 和 `fp-16` 的含义时，我搞清楚了返回地址和前一个栈帧指针在内存中的具体存储位置。就像右图所展示的那样，返回地址处于栈帧帧指针固定偏移(-8)的位置，而保存的帧指针则位于帧指针固定偏移(-16)的位置。此次实验不仅加深了我对计算机系统底层运行机制的认知，也提升了我调试复杂程序的能力。



Alarm (hard)

[实验目的]

In this exercise you'll add a feature to xv6 that periodically alerts a process as it uses CPU time. This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want to take some periodic action. More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers; you could use something similar to handle page faults in the application, for example. Your solution is correct if it passes alarmtest and usertests.

在本练习中，您将向 xv6 添加一个特性，该特性会在进程使用 CPU 时间时定期提醒进程。这可能对于想要限制它们占用多少 CPU 时间的计算绑定进程很有用，或者对于想要计算但也想要采取一些周期性操作的进程很有用。更一般地说，您将实现一种用户级中断/故障处理程序的原始形式；例如，您可以使用类似的方法来处理应用程序中的页面错误。如果您的解决方案通过了警报的测试和用户测试，您的解决方案是正确的。

[实验步骤]

首先，我们需要在 user/user.h 中添加两个系统调用的函数原型：

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

然后在 user/usys.pl 脚本中添加两个系统调用的相应 entry，在 kernel/syscall.h 和 kernel/syscall.c 添加相应声明，类似于 Lab2。

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

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

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

```
[SYS_sigalarm] sys_sigalarm,  
[SYS_sigreturn] sys_sigreturn,
```

在 kernel/proc.h 中增加一些有关 alarm 的属性，同时在 kernel/proc.c 的 allocproc 函数中初始化这些属性：

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

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

接下来就是在 kernel/sysproc.c 中编写 sys_sigalarm 和 sys_sigreturn 函数了：

```
uint64 sys_sigalarm(void) // 设置一个闹钟信号处理程序和时间间隔。
{
    int interval;
    uint64 handler;

    if (argint(0, &interval) < 0) // 获取第一个参数（时间间隔）
        return -1;
    if (argaddr(1, &handler) < 0) // 获取第二个参数（信号处理程序
的址）
        return -1;
    myproc()->alarm_interval = interval; // 设置当前进程的闹钟信
号时间间隔
    myproc()->alarm_handler = handler; // 设置当前进程的闹钟信
号处理程序地址
    return 0;
}
uint64 sys_sigreturn(void) // 用于从信号处理程序中返回，恢复进
程在信号处理程序被调用之前的状态。
{
    memmove(myproc()->trapframe, &(myproc()->alarm_trapframe),
sizeof(struct trapframe)); // 恢复进程的 trapframe（寄存器上下文）
为闹钟信号处理之前的状态
    myproc()->alarm_ticks =
0; //
重置闹钟信号计数器
    return 0;
}
```

最后在 kernel/trap.c 中处理 interrupt 即可。

```
if (which_dev == 2) // 检查是否是指定设备的中断
{
    if (p->alarm_interval)
    {
        if (++p->alarm_ticks == p->alarm_interval) // 递增
alarm_ticks 计数器
        {
            memmove(&(p->alarm_trapframe), p->trapframe,
sizeof(*(p->trapframe))); // 在闹钟信号处理程序执行完毕后，恢复到中
断之前的状态。
            p->trapframe->epc = p->alarm_handler;
        }
    }
    yield();
}
```

当指定设备（编号为 2）触发中断时，系统会先检查当前进程是否设置了闹钟信号的时间间隔——即判断进程的`alarm_interval`是否非零。若满足这一条件，便会将进程的计数器`alarm_ticks`递增。

当计数器的值与设定的时间间隔相等（`alarm_ticks == alarm_interval`时），系统会执行一系列关键操作：首先将当前进程的寄存器上下文（存储在`trapframe`中）保存到`alarm_trapframe`，随后将程序计数器（`epc`）设置为闹钟信号处理程序的地址（`alarm_handler`）。这一设置确保进程从内核态返回用户态时，会优先执行闹钟信号处理程序。

完成上述操作后，系统会调用`yield()`函数，将当前进程重新放入调度队列，由操作系统调度其他进程获得执行机会。

成功运行 alarmtest:

```
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
test1 passed
test2 start
.....alarm!
test2 passed
$
```

成功运行 usertests，最后返回 ALL TESTS PASSED:

```
test exitiput: 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
```

[实验中遇到的问题和解决方法]

这次实验总体而言不算太难，因为其代码实现框架与 Lab2 中的几个实验较为相似，都是向 XV6 系统添加一个新特性。不过，在具体编写某些函数代码时却颇具难度，这需要运用到不少库函数来实现功能。对于此前从未接触过 XV6 操作系统代码的我们来说，这无疑带来了一定的挑战。在本次实验中，我主要通过学习和参考的方式来完成的任务，在理解每一行代码实现意义的基础上，逐步解决实验中遇到的问题。

[实验心得]

本次实验让我深入理解了用户态信号处理与内核中断机制的协作原理。闹钟信号通过“内核时钟中断触发 + 用户态处理程序执行”的流程，实现了进程对周期性事件的响应，这种设计体现了操作系统在用户态与内核态间架起通信桥梁的思路。实验中，对中断上下文的精准把控尤为关键——内核在触发信号时需保存完整的寄存器状态，而用户态处理程序结束后需通过 `sys_sigreturn` 完整恢复，这种“保存 - 替换 - 恢复”的机制确保了信号处理不干扰进程原有执行逻辑。此外，时钟中断的周期性与计数器的配合，展示了时间片管理在操作系统中的基础作用，进程通过设置 `alarm_interval` 自主控制信号频率，体现了用户态对系统资源的灵活调度能力。这次实验不仅掌握了信号机制的实现方法，更让我体会到操作系统通过中断、系统调用和上下文管理的协同，为用户程序提供高效事件响应的底层逻辑。

Lab4 的实验结果截图: ./grade-lab-traps

```
jack@DESKTOP-EMMCGRI:~/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 (1.5s)
== Test running alarmtest == (3.4s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests == usertests: OK (52.6s)
== Test time ==
time: OK
Score: 85/85
```

Lab5: Lazy allocation

首先, 开始 Lab5 时, 需要通过以下命令切换实验分支, 获取实验资源:

git fetch

git checkout lazy

make clean

Eliminate allocation from sbrk() (easy)

[实验目的]

Your first task is to delete page allocation from the sbrk(n) system call implementation, which is the function sys_sbrk() in sysproc.c. The sbrk(n) system call grows the process's memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new sbrk(n) should just increment the process's size (myproc()->sz) by n and return the old size. It should not allocate memory -- so you should delete the call to growproc() (but you still need to increase the process's size!).

你的首项任务是删除 sbrk(n) 系统调用中的页面分配代码(位于 sysproc.c 中的函数 sys_sbrk())。sbrk(n) 系统调用将进程的内存大小增加 n 个字节, 然后返回新分配区域的开始部分(即旧的大小)。新的 sbrk(n) 应该只将进程的大小 (myproc()->sz) 增加 n, 然后返回旧的大小。它不应该分配内存——因此您应该删除对 growproc() 的调用(但是您仍然需要增加进程的大小!)。

[实验步骤]

按照实验指导,在`kernel/sysproc.c`修改`sys_sbrk()`函数,删去原本调用的`growproc()`函数,增加`myproc()->sz`;

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if (argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    // if(growproc(n) < 0)
    //     return -1;
    myproc()->sz += n;
    return addr;
}
```

删去本调用的`growproc()`函数,添加`myproc()->sz`;用`addr`暂存`myproc()->sz`旧的值,同时`myproc()->sz`增加`n`,返回旧的大小(即`addr`的值)。`make qemu`后执行`echo hi`命令,得到结果如下:

```
hart 1 starting
hart 2 starting
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
          sepc=0x00000000000012ac stval=0x0000000000004008
panic: uvmunmap: not mapped
```

[实验中遇到的问题和解决方法]

这个实验比较简单,整个实验过程没有遇到什么问题。

[实验心得]

在本次实验里,我对`xv6`的`sys_sbrk()`系统调用进行了修改,让它不再分配实际的内存页面,而只是单纯增加进程的大小。通过这一修改,我得以深入理解操作系统内存管理的原理,尤其是内存分配与进程大小调整之间的关联。

Lazy allocation (moderate)

[实验目的]

Modify the code in `trap.c` to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. You should add your code just before the `printf` call that produced the "`usertrap(): ...`" message. Modify whatever other xv6 kernel code you need to in order to get `echo hi` to work.

修改 `trap.c` 中的代码以响应来自用户空间的页面错误，方法是新分配一个物理页面并映射到发生错误的地址，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。你可以修改任何其他 xv6 内核代码，以使 `echo hi` 正常工作。

[实验步骤]

根据提示信息，当 `r_scause()` 的返回值为 13 或 15 时，表明发生了页面错误（page fault），同时 `r_stval()` 会返回导致该页面错误的虚拟地址。其中，系统调用的中断码是 8，而页面错误的中断码为 13 和 15。因此，在这里我们需要增加对 `r_scause()` 返回的中断原因的判断：若其值为 13 或 15，就说明未找到对应的地址。此时，发生错误的虚拟地址已存储在 `STVAL` 寄存器中。在进行中断判断时，一旦出现错误（无论是虚拟地址不合法，还是未能成功映射到物理地址），都需要终止相应的进程。所以我们需要在 `trap.c` 中对 `usertrap` 函数进行修改：

```
intr_on();

    syscall();
}
else if (r_scause() == 13 || r_scause() == 15) // 对缺页异常的处理
{
    uint64 fault_va = r_stval();
    if (PGROUNDDOWN(p->trapframe->sp) >= fault_va ||
    fault_va >= p->sz) // 检查 fault_va 是否在合法的地址范围内
    {
        p->killed = 1;
    }
    else
    {
        char *pa = kalloc();
        if (pa != 0) // 检查页面分配结果
        {
```

```

        memset(pa, 0, PGSIZE);
        if (mappages(p->pagetable, PGROUNDDOWN(fault_va),
PGSIZE, (uint64)pa, PTE_R | PTE_W | PTE_U) != 0)
        {
            printf("haha\n");
            kfree(pa);
            p->killed = 1;
        }
    }
    else
    {
        printf("kalloc == 0\n");
        p->killed = 1;
    }
}
else if ((which_dev = devintr()) != 0)
{
    // ok
}
else
{

```

接着，由于 `uvmunmap` 用于释放内存，而释放内存时，页表中部分地址并未实际分配内存，因此未进行映射。若在 `uvmunmap` 中遇到未映射的地址，无需执行 `panic`。所以我们要修改 `kernl/vm.c` 中的 `uvmunmap()` 函数，将那两行 `panic` 代码注释掉，替换为 `continue` 语句。

```

if ((pte = walk(pagetable, a, 0)) == 0)
    // panic("uvmunmap: walk");
    continue;
if ((*pte & PTE_V) == 0)
    // panic("uvmunmap: not mapped");
    continue;

```

最后，测试实验结果，在 `make qemu` 之后输入 `echo hi`,测试结果如图所示：

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ echo hi
hi
$ 

```

[实验中遇到的问题解决方法]

主要是内存释放时的恐慌错误，`uvmunmap` 函数在遇到未映射的页表项时会触发 `panic`，而延迟分配场景下未映射的页表项是正常现象。通过将 `panic` 替换为 `continue`，允许释放未实际分配的页面，解决了程序退出时的崩溃问题。此外，还遇到物理页分配失败的处理缺失，通过在 `kalloc` 返回空时标记进程为 `killed`，确保内存不足时能正确终止进程，提升了鲁棒性。

[实验心得]

本次实验让我深刻理解了延迟分配策略对内存利用效率的提升作用。传统分配方式在进程调用 `sbrk` 时立即分配物理页，而延迟分配仅在进程实际访问地址时才分配，避免了对未使用内存的浪费，这种“按需分配”的思路是现代操作系统内存管理的核心优化手段。实现中，缺页异常的处理是关键：通过捕获用户态页错误，动态分配物理页并建立映射，让进程无需感知底层分配细节即可继续执行，体现了操作系统“透明性”的设计原则。同时，地址合法性校验与内存释放逻辑的调整，展示了内核对内存安全与资源管理的平衡——既要满足进程动态扩存需求，又要严格限制访问范围防止越界。这次实验不仅掌握了缺页异常处理的实现方法，更让我体会到操作系统通过硬件异常（页错误）与软件机制（动态映射）结合，实现高效内存管理的精妙设计。

Lazytests and Usertests (moderate)

[实验目的]

We've supplied you with lazytests, an xv6 user program that tests some specific situations that may stress your lazy memory allocator. Modify your kernel code so that all of both lazytests and usertests pass.

我们为您提供了 lazytests，这是一个 xv6 用户程序，它测试一些可能会给您的惰性内存分配器带来压力的特定情况。修改内核代码，使所有 lazytests 和 usertests 都通过。

[实验步骤]

我们不修改代码，在前一个实验的基础上，`make qemu`，输入 lazytests 和 usertests 即可，都显示出 ALL TESTS PASSED 就可以了。

<pre>\$ lazytests lazytests starting running test lazy alloc test lazy alloc: OK running test lazy unmap test lazy unmap: OK running test out of memory test out of memory: OK ALL TESTS PASSED</pre>	<pre>test mem: kalloc == 0 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 kalloc == 0 ALL TESTS PASSED</pre>
---	---

[实验中遇到的问题和解决方法]

这个实验只要进行 lazytests 和 usertests 测试即可，因此没有遇到什么问题。

[实验心得]

和上一个实验相同，本实验只是进行两个测试，没有什么大问题。

Lab5 的实验结果截图： ./grade-lab-lazy

```
== 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
```

Lab6:Copy-on-Write Fork for xv6

首先，开始 Lab6 时，需要通过以下命令切换实验分支，获取实验资源：

```
git fetch
```

```
git cow
```

```
make clean
```

Implement copy-on write(hard)

[实验目的]

Your task is to implement copy-on-write fork in the xv6 kernel. You are done if your modified kernel executes both the cowtest and usertests programs successfully.

您的任务是在 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。

[实验步骤]

我们首先需要修改 uvmcopy()将父进程的物理页映射到子进程，不是分配新页。我们在子进程和父进程的 PTE 中清除 PTE_W 标志。

```
int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;

    for (i = 0; i < sz; i += PGSIZE) // 循环遍历页表
    {
        if ((pte = walk(old, i, 0)) == 0) // 获取页表项
            panic("uvmcopy: pte should exist");
        if ((*pte & PTE_V) == 0)
            panic("uvmcopy: page not present");
        // set unwrite and set rsw
        *pte = ((*pte) & (~PTE_W)) | PTE_RSW;
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte);
        // stop copy
        // if((mem = kalloc()) == 0)
        //     goto err;
        // memmove(mem, (char*)pa, PGSIZE);
    }
}
```

```

        if (mappages(new, i, PGSIZE, pa, flags) != 0) // 将新的页
表映射到物理地址 pa
        {
            // kfree(mem);
            goto err;
        }
        add_kmem_ref((void *)pa);
    }
    return 0;
err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}

```

写时复制（COW）机制下的 fork() 函数，其设计目标是推迟物理内存的分配与复制操作，直到子进程确实需要用到这些物理内存页面时才执行。这段代码的作用，就是将父进程的物理页映射到子进程中，而非为子进程分配新的物理页。之后，需要对 usertrap() 函数进行修改，使其能够识别页面错误。当写时复制页面发生页面错误时，通过 kalloc() 分配一个新页面，把旧页面的内容复制到新页面中，随后将新页面添加到页表项（PTE）中，并设置 PTE_W 标志位。相关代码如下：

```

    intr_on();
    syscall();
}
else if (r_scause() == 15) // 写页错误处理
{
    // write page fault
    uint64 va = PGROUNDDOWN(r_stval());
    pte_t *pte;
    if (va > p->sz || (pte = walk(p->pagetable, va, 0)) == 0)
    {
        p->killed = 1;
        goto end;
    }
    if (((*pte) & PTE_RSW) == 0 || ((*pte) & PTE_V) == 0 || ((*pte)
& PTE_U) == 0)
    {
        p->killed = 1;
        goto end;
    }
    uint64 pa = PTE2PA(*pte);
    acquire_ref_lock();
    uint ref = get_kmem_ref((void *)pa);

```

```

    if (ref == 1){
        *pte = ((*pte) & (~PTE_RSW)) | PTE_W;
    }
    else{
        char *mem = kalloc();
        if (mem == 0){
            p->killed = 1;
            release_ref_lock();
            goto end;
        }
        memmove(mem, (char *)pa, PGSIZE);
        uint flag = (PTE_FLAGS(*pte) | PTE_W) & (~PTE_RSW);
        if (mappages(p->pagetable, va, PGSIZE, (uint64)mem,
flag) != 0)
        {
            kfree(mem);
            p->killed = 1;
            release_ref_lock();
            goto end;
        }
        kfree((void *)pa);
    }
    release_ref_lock();
}
else if ((which_dev = devintr()) != 0){
    // ok
}

```

上述代码实现了写页错误发生时的异常处理机制，涵盖页表项操作、物理内存的分配与释放，以及异常场景下的错误处理和设备中断处理。

接下来需要修改 `kalloc.c` 文件：当 `kalloc()` 分配页面时，要将该页面的引用计数设为 1；当 `fork` 操作使子进程共享页面时，需增加该页面的引用计数；每当有进程从其页表中删除页面，就减少该页面的引用计数。`kfree()` 仅在引用计数为零时，才将页面放回空闲列表。

这些计数可存储在一个固定大小的整型数组中。你需要设计一套数组索引方案和数组大小选择方案。例如，可以用页面的物理地址除以 4096 作为数组的索引，并根据 `kalloc.c` 中 `kinit()` 在空闲列表中放置的所有页面的最高物理地址，来确定数组应包含的元素数量。

```

void *
kalloc(void)

```



```

{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if (r){
        kmem.freelist = r->next;
        kmem.ref_count[INDEX((void *)r)] = 1;
    }
    release(&kmem.lock);
    if (r)
        memset((char *)r, 5, PGSIZE); // fill with junk
    return (void *)r;
}
int get_kmem_ref(void *pa){
    return kmem.ref_count[INDEX(pa)];
}
void add_kmem_ref(void *pa){
    kmem.ref_count[INDEX(pa)]++;
}
void acquire_ref_lock(){
    acquire(&kmem.ref_lock);
}
void release_ref_lock(){
    release(&kmem.ref_lock);
}
}

```

最后需要修改 copyout() 在遇到 COW 页面时使用与页面错误相同的方案。在文件 kernel/vm.c 中修改代码：

```

    if (va0 >= MAXVA) // 地址检查和页表项获取
        return -1;
    if ((pte = walk(pagetable, va0, 0)) == 0)
        return -1;
    if (((*pte & PTE_V) == 0) || ((*pte & PTE_U) == 0)) // 页
    表项有效性检查
        return -1;

    pa0 = PTE2PA(*pte);
    if (((*pte & PTE_W) == 0) && (*pte & PTE_RSW)) // 写保护
    处理
    {
        acquire_ref_lock();
        if (get_kmem_ref((void *)pa0) == 1){
            *pte = (*pte | PTE_W) & (~PTE_RSW);

```

```

    }
    else
    {
        char *mem = kalloc();
        if (mem == 0){
            release_ref_lock();
            return -1;
        }
        memmove(mem, (char *)pa0, PGSIZE);
        uint new_flags = (PTE_FLAGS(*pte) | PTE_RSW) &
(~PTE_W);
        if (mappages(pagetable, va0, PGSIZE, (uint64)mem,
new_flags) != 0)
        {
            kfree(mem);
            release_ref_lock();
            return -1;
        }
        kfree((void *)pa0);
    }
    release_ref_lock();
}

```

这段代码主要用于处理写保护异常，具体操作包括取消页面的写保护位、重新映射页面，以及对页表项和物理页面进行相关处理。当页表项 `pte` 未设置写保护位，且保留位已设置（即 `*pte & PTE_RSW` 的条件成立）时，会执行以下流程：

1. 获取物理页面的引用锁；
2. 若该物理页面的引用计数为 1（意味着只有当前进程在使用此页面），则取消页表项的保留位 `PTE_RSW`，并设置其写权限 `PTE_W`；
3. 若该物理页面被多个进程共享，则通过分配新的物理内存 `mem`，将原物理页面的数据复制到新内存中；
4. 更新新页表项的标志，将新内存映射到页表；若此操作失败，需释放新内存和引用锁，并返回 -1；
5. 释放原物理页面 `pa0` 的内存。

最后进行实验结果测试，在 `make qemu` 之后进行 `cowtest` 和 `usertests`，最后显示的结果都为 `ALL TESTS PASSED` 即表示成功完成实验。

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED

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
```

[实验中遇到的问题解决方法]

在实现写时复制（COW）机制时，主要遇到两个关键问题。一是引用计数管理混乱，初期未通过锁机制保护引用计数的增减操作，导致多进程并发修改时出现计数错误，引发页面提前释放或内存泄漏。通过在 `add_kmem_ref` 和 `get_kmem_ref` 中加入 `ref_lock` 自旋锁，确保计数操作的原子性，解决了并发一致性问题。二是页表项标志位处理错误，未正确设置 `PTE_RSW` 作为 COW 标记，或在页面复制后未清除该标志，导致写操作时重复触发复制逻辑。通过严格在 `uvmcopy` 中设置 `PTE_RSW` 并清除 `PTE_W`，在页面复制后恢复 `PTE_W` 并清除 `PTE_RSW`，确保标志位状态与页面状态一致。

[实验心得]

本次实验让我深刻理解了写时复制（COW）机制对内存利用效率的显著提升。传统 `fork` 会立即复制父进程所有物理页，造成大量冗余操作，而 COW 通过共享物理页并延迟复制，仅在实际写入时分配新页，大幅减少了内存消耗和复制开销，尤其适合 `fork` 后立即执行 `exec` 的场景。实现中，引用计数的设计是核心——它精准追踪页面的共享次数，决定页面是否需要复制或释放，这种轻量级的资源追踪方式体现了操作系统对效率与正确性的平衡。页表项标志位（如 `PTE_RSW` 和 `PTE_W`）的巧妙运用，则实现了对共享页面的访问控制，让硬件

（MMU）与软件（异常处理）协同工作：MMU 检测写操作并触发异常，内核捕获异常后执行复制逻辑，这种“硬件触发 + 软件处理”的模式是 COW 实现的关键。此外，copyout 等内核函数对 COW 的适配，展示了内核机制的整体性——任何涉及内存写入的操作都需考虑特殊页面状态，确保系统行为的一致性。这次实验不仅掌握了 COW 的实现细节，更让我体会到操作系统通过延迟操作、资源共享和异常处理的协同，实现高效内存管理的深层逻辑。

Lab6 的实验结果截图：

./grade-lab-cow

```
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./grade-lab-cow
make: 'kernel/kernel' is up to date.
== Test running cowtest == (2.8s)
== Test  simple ==
    simple: OK
== Test  three ==
    three: OK
== Test  file ==
    file: OK
== Test usertests == (53.3s)
== Test  usertests: copyin ==
    usertests: copyin: OK
== Test  usertests: copyout ==
    usertests: copyout: OK
== Test  usertests: all tests ==
    usertests: all tests: OK
== Test time ==
    time: OK
Score: 110/110
```

Lab7:Multithreading

首先，开始 Lab7 时，需要通过以下命令切换实验分支，获取实验资源：

git fetch

git thread

make clean

Uthread: switching between threads (moderate)

[实验目的]

Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan. When you're done, make grade should say that your solution passes the uthread test.

您的工作是提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划。完成后，make grade 应该表明您的解决方案通过了 uthread 测试。

[实验步骤]

我们首先要在 user/uthread.c 中定义一个结构体，用于存储寄存器内容。通过这种方式，每个线程都能拥有独立的上下文信息，从而在进行线程切换时，可以对这些上下文进行保存与恢复操作。

```
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;
};
```

同时需要注意，要将这个结构体添加到下文的线程结构体中。在进行进程切换时，只需交换调用方保存的进程上下文即可。其中，ra 寄存器表示返回地址（return address），sp 寄存器表示栈指针（stack pointer），这两个寄存器无需在进程切换时进行交换，而是由系统负责更新。请修改 user/uthread_switch.S 文件，以实现 thread_switch 函数：

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 */

```

修改 `thread_create()` 函数，使其能在线程数组中遍历。

```

t->context.ra = (uint64)func;
t->context.sp = (uint64)&t->stack[STACK_SIZE - 1];

```

需要修改 `thread_schedule()` 函数，该函数承担着用户多线程之间的调度工作。它会从当前线程所在的位置开始，在线程数组中查找处于 `RUNNABLE` 状态的线程来运行，找到目标线程后，便调用 `thread_switch()` 函数完成线程切换。

```

thread_switch((uint64)&t->context,
(uint64)&current_thread->context);

```

进入 `xv6`, `make qemu` 之后，输入 `uthread` 后得到预期结果，表示测试成功：

```
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
$
```

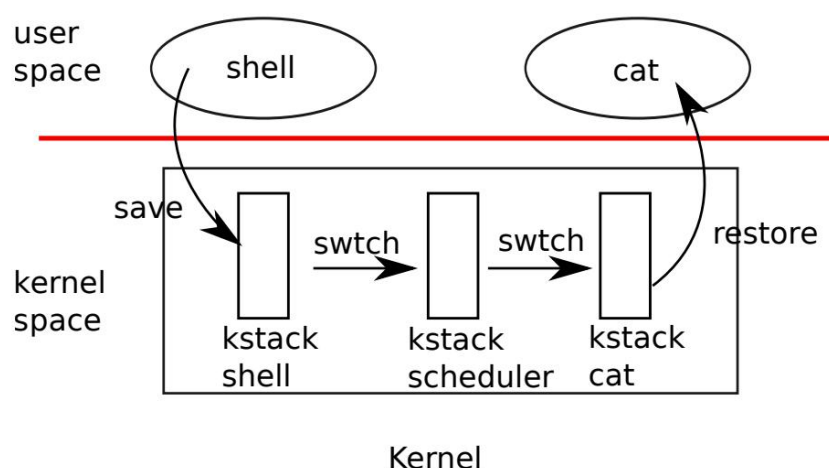
[实验中遇到的问题解决方法]

这次实验主要问题是调度逻辑遗漏当前线程状态，`thread_schedule` 函数在切换线程后未更新原线程的状态，导致线程重复被调度。通过在切换前将当前线程标记为 `RUNNABLE`（若未终止），确保调度器仅选择就绪状态的线程，避免了无效调度。此外，还遇到线程切换时参数传递错误，因 `thread_switch` 的参数顺序颠倒，导致上下文读写错位，通过调整参数顺序使源上下文和目标上下文正确对应，解决了切换逻辑错误。

[实验心得]

本次实验让我深入理解了用户级线程切换的核心机制。线程切换的本质是上下文的保存与恢复 —— 通过保存当前线程的寄存器状态，加载目标线程的寄存器状态，实现 CPU 执行权的无缝转移，这种轻量级的切换方式比进程切换更高效，因为无需陷入内核态。实验中，被调用者保存寄存器的处理尤为关键：根据 RISC-V 调用规范，这些寄存器由被调用者负责保存，线程切换时必须完整保存和恢复，否则会导致函数调用链中的数据丢失，这体现了硬件规范与软件实现的紧密关联。线程调度逻辑则展示了多线程协作的核心：`thread_schedule` 通过遍历线程数组选择就绪线程，`thread_switch` 负责实际切换，二者配合实现了简单的轮询调度，为理解操作系统进程调度提供了微观视角。此外，栈指针的正确初始化让我认识到线程栈的重要性 —— 它不仅是函数调用的载体，更是线程独立执行的基础。这次实验不仅掌握了线程切换的实现方法，更让我体会到用户级线程通过纯软件机制实现并发的巧妙，为理解更高层次的并发模型奠定了基础。从一个

用户进程切换到另一个用户进程。在本例中，xv6 使用一个 CPU 运行。如下图：



Using threads (moderate)

[实验目的]

In this assignment you will explore parallel programming with threads and locks using a hash table. You should do this assignment on a real Linux or MacOS computer (not xv6, not qemu) that has multiple cores. Most recent laptops have multicore processors.

在本作业中，您将探索使用哈希表的线程和锁的并行编程。您应该在具有多个内核的真实 Linux 或 MacOS 计算机(不是 xv6，不是 qemu)上执行此任务。最新的笔记本电脑都有多核处理器。

[实验步骤]

我们首先需要在 ph.c 中声明多线程的锁，并且在 main 函数里初始化：

```
pthread_mutex_t lock[NBUCKET];

// 初始化锁
for (int i = 0; i < NBUCKET; i++){
    pthread_mutex_init(&lock[i], NULL);
}
```

线程的安全问题是对桶中的链表进行操作而导致的，我们要在链表操作的前后加锁，在`put`函数读写 bucket 之前加锁，在函数结束时释放锁。

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

```

完成修改后运行如下指令，测试结果成功：

```

$ QEMU: terminated
● jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ make ph
gcc -o ph -g -O2 notxv6/ph.c -pthread
● jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./ph 1
100000 puts, 4.184 seconds, 23899 puts/second
0: 0 keys missing
100000 gets, 4.292 seconds, 23299 gets/second
● jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./ph 2
100000 puts, 2.186 seconds, 45747 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 4.265 seconds, 46888 gets/second
○ jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$

```

[实验中遇到的问题和解决方法]

当不同线程对同一个 bucket 执行 put 操作时，可能会覆盖前一次 put 的结果，进而导致错误。若对整个哈希表加锁，其整体性能甚至会比之前更差，这是因为同一时刻仅能有一个线程进行操作，相当于同于单线程运行。优化的思路是降低锁的粒度，将对整个哈希表加锁改为对每个 bucket 单独加锁。

[实验心得]

本次实验让我深刻理解了多线程并发控制中锁粒度与性能的权衡关系。哈希表的桶结构天然适合细粒度锁设计 —— 不同桶的操作相互独立, 通过为每个桶分配锁, 既能保证同一桶内操作的原子性 (避免数据竞争), 又能让不同桶的操作在多核处理器上并行执行, 充分利用硬件资源。这种 “分而治之” 的思想是并行编程的核心优化手段。实验中, 对 RISC-V 多线程机制的实践也让我体会到: 线程安全并非简单依赖全局锁, 而是需要结合数据结构的特性设计针对性的同步策略。例如, 哈希表的桶隔离性使其无需全局锁, 而链表等无结构关联的数据则可能需要更复杂的同步机制。此外, 通过对比全局锁与桶级锁的性能差异, 我直观感受到锁粒度对并行效率的影响 —— 过粗的锁会扼杀并行性, 过细的锁则可能增加开销, 找到合适的平衡点是关键。这次实验不仅掌握了多线程同步的基本方法, 更让我认识到并行编程中 “数据独立性” 对性能优化的重要性。

Barrier(moderate)

[实验目的]

In this assignment you'll implement a barrier: a point in an application at which all participating threads must wait until all other participating threads reach that point too. You'll use pthread condition variables, which are a sequence coordination technique similar to xv6's sleep and wakeup.

在本作业中, 您将实现一个屏障(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。您将使用 pthread 条件变量, 这是一种序列协调技术, 类似于 xv6 的 sleep 和 wakeup。

[实验步骤]

实验中需要实现阻塞线程, 直至所有线程都到达屏障点, 然后才允许所有线程继续执行。实验代码如下:

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

```

        pthread_mutex_lock(&bstate.barrier_mutex); // 锁住互斥锁，进入临界区
        bstate.nthread++; // 增加已到达屏障的线程数
        if (bstate.nthread == nthread) // 检查是否所有线程都到达屏障点
        {
            bstate.round++; // 增加轮次计数器
            bstate.nthread = 0; // 重置已到达屏障的线程数
            pthread_cond_broadcast(&bstate.barrier_cond); // 唤醒所有等待的线程
        }
        else{
            pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex); // 当前线程等待，直到被唤醒
        }
        pthread_mutex_unlock(&bstate.barrier_mutex); // 解锁互斥锁，离开临界区
    }

```

首先要锁定互斥锁，保证只有一个进程能进入临界区，此时把已到达屏障的线程数加一。之后检查是否所有线程都抵达了屏障点：若是，就重置线程数，并唤醒所有处于等待状态的线程；若不是，当前当前线程继续等待。最后释放互斥锁，退出临界区。测试实验结果成功通过：

```

jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./barrier 2
OK; passed
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$

```

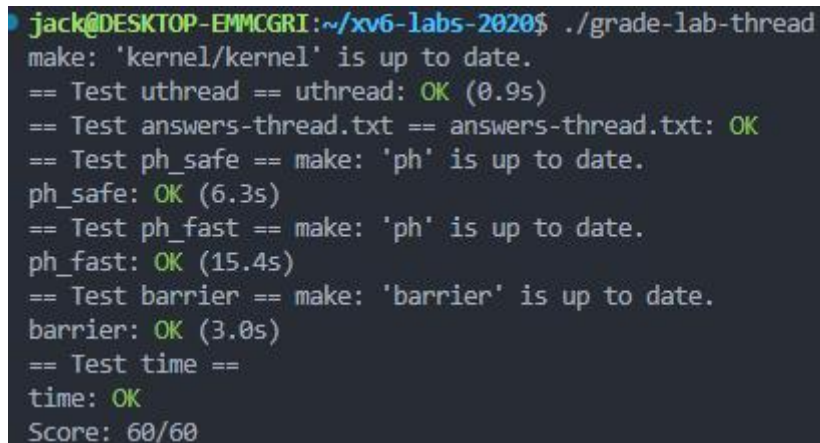
[实验中遇到的问题和解决方法]

这段代码的实现相对简单。本次实验的关键在于理解需要实现的功能是什么，在吃透阻塞线程的概念后，按照正常的逻辑顺序以及指导提示中的逻辑步骤来编写，就能顺利完成此次实验。

[实验心得]

本次实验让我深入理解了线程同步中屏障机制的核心逻辑。屏障通过“等待 - 唤醒”模式实现多线程协同，确保所有线程到达指定点后再共同推进，这种同步方式在并行计算中尤为重要（如分阶段任务的阶段切换）。实验中，互斥锁与条件变量的配合是关键：互斥锁保护共享状态（`nthread` 和 `round`）的修改，条件变量则实现线程的高效等待与唤醒，二者结合既保证了数据一致性，又避免了忙等带来的资源浪费。通过对比 `pthread_cond_signal` 和 `pthread_cond_broadcast` 的差异，我认识到：屏障需要唤醒所有等待线程，而单一唤醒仅适用于生产者 - 消费者等一对一唤醒场景。此外，轮次计数器（`round`）的设计体现了屏障的可复用性，使同一组线程能在多轮任务中重复使用屏障同步，无需重新初始化。这次实验不仅掌握了条件变量的使用方法，更让我体会到并行编程中“协同推进”的设计思想，为处理复杂多线程同步问题奠定了基础。

Lab7 的实验结果截图：./grade-lab-thread



```
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (0.9s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (6.3s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (15.4s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (3.0s)
== Test time ==
time: OK
Score: 60/60
```

Lab8:Locks

首先，开始 Lab8 时，需要通过以下命令切换实验分支，获取实验资源：

git fetch

git lock

make clean

Memory allocator (moderate)

[实验目的]

Your job is to implement per-CPU freelists, and stealing when a CPU's free list is empty. You must give all of your locks names that start with "kmem". That is, you should call `initlock` for each of your locks, and pass a name that starts with "kmem". Run `kalloctest` to see if your implementation has reduced lock contention. To check that it can still allocate all of memory, run `usertests sbrkmuch`. Your output will look similar to that shown below, with much-reduced contention in total on kmem locks, although the specific numbers will differ. Make sure all tests in `usertests` pass. `make grade` should say that the `kalloctests` pass.

您的工作是实现每个 CPU 的空闲列表，并在 CPU 的空闲列表为空时进行窃取。所有锁的命名必须以“kmem”开头。也就是说，您应该为每个锁调用 `initlock`，并传递一个以“kmem”开头的名称。运行 `kalloctest` 以查看您的实现是否减少了锁争用。要检查它是否仍然可以分配所有内存，请运行 `usertests sbrkmuch`。您的输出将与下面所示的类似，在 `kmem` 锁上的争用总数将大大减少，尽管具体的数字会有所不同。确保 `usertests` 中的所有测试都通过。评分应该表明考试通过。

[实验步骤]

原先的 `kalloc()` 函数仅使用一个空闲列表，由单一锁进行保护。这会导致当多个 CPU 需要分配内存时，对该锁的抢占次数大幅增加，影响效率。

为解决这一问题，我们需要为每个 CPU 分别创建一个空闲列表，并为每个列表分配独立的锁。具体操作是在 `kernel/kalloc.c` 中，将 `kmem` 修改为数组形式，以便为每个 CPU 分配对应的 `kmem` 实例。

```
struct
{
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU]; // 每个 CPU 都有一个独立的内存管理单元
```

修改 `kinit` 函数，给目前的 CPU 的 `'freelist'` 分配所有的空闲内存空间。

```
for (int i = 0; i < NCPU; i++)
{
    snprintf(lockname, sizeof(lockname), "kmem%d", i);
```

```
    initlock(&kmem[i].lock, lockname);
}
```

修改 kfree 对释放锁，适应新的数据结构，使用 push_off() 和 pop_off() 来关闭和打开中断，让 cpuid() 能够正确获得当前 CPU 编号：

```
push_off(); // 关闭中断，以确保在获取 CPU ID 之前不会发生中断
int id = cpuid();
acquire(&kmem[id].lock); // 获取当前 CPU 的自旋锁
r->next = kmem[id].freelist;
kmem[id].freelist = r;
release(&kmem[id].lock); // 释放自旋锁
pop_off(); // 恢复中断
```

修改 kalloc 函数，查找当前 CPU 中是否有空闲块，如果没有则需要从其他 CPU 中获取。

```
push_off(); // 关闭中断，确保以下操作不被打断
int id = cpuid(); // 获取当前 CPU 的 ID
acquire(&kmem[id].lock); // 获取当前 CPU 的自旋锁
r = kmem[id].freelist; // 尝试从当前 CPU 的空闲列表中获取内存块
if (r)
    kmem[id].freelist = r->next;
Else
{
    int new_id;
    for (new_id = 0; new_id < NCPU; ++new_id)
    {
        if (new_id == id)
            continue;
        acquire(&kmem[new_id].lock);
        r = kmem[new_id].freelist;
        if (r)
        {
            kmem[new_id].freelist = r->next;
            release(&kmem[new_id].lock);
            break;
        }
        release(&kmem[new_id].lock);
    }
}
release(&kmem[id].lock); // 释放当前 CPU 的自旋锁
pop_off(); // 恢复中断
```

输入如图所示代码测试实验结果成功通过：

分别为 kalloc test、usertest sbrkmuch 和 usertest:

```
$ kalloc test
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache: #fetch-and-add 32 #acquire() 7095
lock: bcache: #fetch-and-add 0 #acquire() 5605
lock: bcache: #fetch-and-add 0 #acquire() 16036
lock: bcache: #fetch-and-add 0 #acquire() 14976
lock: bcache: #fetch-and-add 0 #acquire() 301416
lock: bcache: #fetch-and-add 19 #acquire() 175568
lock: bcache: #fetch-and-add 243 #acquire() 187082
lock: bcache: #fetch-and-add 19 #acquire() 140196
lock: bcache: #fetch-and-add 1823 #acquire() 672484
lock: bcache: #fetch-and-add 27 #acquire() 76201
lock: bcache: #fetch-and-add 3 #acquire() 71334
lock: bcache: #fetch-and-add 0 #acquire() 23712
lock: bcache: #fetch-and-add 0 #acquire() 10044
--- top 5 contended locks:
lock: proc: #fetch-and-add 76826842 #acquire() 5128305
lock: proc: #fetch-and-add 48191002 #acquire() 5112577
lock: proc: #fetch-and-add 19622995 #acquire() 5112480
lock: proc: #fetch-and-add 16106567 #acquire() 5115829
lock: proc: #fetch-and-add 15509021 #acquire() 5107184
tot= 2166
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
```

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

```
sepc=0x00000000000041fc stval=0x0000000000012000
OK
test sbrkarg: OK
test validate test: OK
test stacktest: usertrap(): unexpected scause 0x00000000000000d pid=6276
sepc=0x00000000000022cc stval=0x000000000000fb90
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 rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

[实验中遇到的问题解决方法]

本次实验的核心是为每个 CPU 创建独立的空闲列表并配备专属锁，这部分实现相对简单，只需将原有的 `kmem` 属性改为数组存储即可。实验中需要重点思考的是内存分配方法。通过查阅资料发现，允许不同 CPU 共享内存池是个有效的解决方案：首先获取当前 CPU 的 ID，尝试从其空闲列表中获取空闲块，成功则直接返回；若当前 CPU 的空闲列表为空，则遍历其他所有 CPU 的空闲列表，尝试从中借用空闲块。一旦找到，就将该空闲块从其他 CPU 的列表中移除，加入当前 CPU 的列表后再返回。

[实验心得]

本次实验让我深刻理解了多核环境下内存分配器的优化思路。通过为每个 CPU 分配独立的空闲列表和锁，显著减少了锁争用——原本全局锁导致的所有 CPU 串行等待，变为仅当本地列表为空时才需访问其他 CPU 的列表，大幅提升了并行性。这种“本地优先，远程窃取”的策略是多核系统资源管理的典型模式，平衡了效率与公平性。实验中，中断控制的重要性尤为突出：`push_off()` 和 `pop_off()` 确保了 CPU ID 在操作期间的稳定性，避免了因中断切换 CPU 导致的锁错乱，这体现了内核代码对执行环境原子性的严格要求。此外，锁命名规范的遵守和多 CPU 遍历逻辑的优化，让我认识到系统级编程中细节（如锁粒度、命名规范）对正确性和可维护性的关键影响。这次实验不仅掌握了多核内存分配的实现方法，更让我体会到操作系统在硬件并行性与软件同步间寻求平衡的深层设计逻辑。

Buffer cache (hard)

[实验目的]

Modify the block cache so that the number of acquire loop iterations for all locks in the bcache is close to zero when running `bcachetest`. Ideally the sum of the counts for all locks involved in the block cache should be zero, but it's OK if the sum is less than 500. Modify `bget` and `brelse` so that concurrent lookups and releases for different blocks that are in the bcache are unlikely to conflict on locks (e.g., don't all have to wait for `bcache.lock`). You must maintain the invariant that at most one copy of each

block is cached. When you are done, your output should be similar to that shown below (though not identical). Make sure `usertests` still passes. `make grade` should pass all tests when you are done.

修改块缓存，以便在运行 `bcachetest` 时，`bcache`(buffer cache 的缩写)中所有锁的 `acquire` 循环迭代次数接近于零。理想情况下，块缓存中涉及的所有锁的计数总和应为零，但只要总和小于 500 就可以。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`)。你必须保护每个块最多缓存一个副本的不变量。完成后，您的输出应该与下面显示的类似(尽管不完全相同)。确保 `usertests` 仍然通过。完成后，`make grade` 应该通过所有测试。

[实验步骤]

Buffer cache 是 `xv6` 文件系统的组成部分，其功能是缓存部分磁盘数据，以此减少磁盘操作的时间开销。不过，这也使得所有进程（包括运行在不同 CPU 上的进程）都会共享这一数据结构。若仅依靠一个锁 `bcache.lock` 来确保对它修改的原子性，会引发大量的竞争问题，进而可能导致性能下滑。因此，我们首先要定义 `bucket` 的数量并将其添加到 `bcache` 中，同时实现一个简单的哈希函数：

```
#define NBUCKETS 13
int hash(uint blockno){
    return blockno % NBUCKETS;
}
```

将数据块均匀地分配到十三个桶中。函数 `hash` 需要在 `def.h` 中声明。

修改 `kernel/bio.c` 中的 `bcache` 结构体，添加每个桶的锁。

```
struct{
    struct spinlock lock[NBUCKETS]; // 每个桶的锁
    struct buf buf[NBUF];
    // Linked list of all buffers, through prev/next.
    // Sorted by how recently the buffer was used.
    // head.next is most recent, head.prev is least.
    struct buf head[NBUCKETS];
} bcache;
```

这些数据需要被准确地初始化：

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

```

    for (int i = 0; i < NBUCKETS; i++) // 初始化每个桶的锁{
        initlock(&bcache.lock[i], "bcache");
    }
    // Create linked list of buffers
    for (int i = 0; i < NBUCKETS; i++) // 创建缓冲区的双向链表{
        bcache.head[i].prev = &bcache.head[i];
        bcache.head[i].next = &bcache.head[i];
    }
    for (b = bcache.buf; b < bcache.buf + NBUF; b++) // 将所有
缓冲区放入第一个桶的链表中{
        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;
    }
}

```

接下来，我们需要对 `bget` 和 `brelse` 这两个函数进行修改：将所有原本使用的 `bcache.lock` 替换为 `bcache.hashlock[hashcode]`，同时把 `bcache.head` 改为 `bcache.head[hashcode]`。这样的改动意味着每个桶都拥有了独立的锁，以及各自对应的缓冲区双向链表。最后，要对 `bget()` 函数做进一步调整，使其能够在缓存系统中找到可用的缓冲区。具体来说，当找到一个引用计数为 0 的缓冲区时，需将其从原位置移至新位置，然后返回该缓冲区。

```

while (1){
    i = (i + 1) % NBUCKETS;
    if (i == id) // 防止死循环
        continue;
    acquire(&bcache.lock[i]);
    for (b = bcache.head[i].prev; b != &bcache.head[i]; b =
b->prev){
        if (b->refcnt == 0){
            b->dev = dev;
            b->blockno = blockno;
            b->valid = 0;
            b->refcnt = 1;

            b->prev->next = b->next; // 断开当前缓冲区的链表连接
            b->next->prev = b->prev;
            release(&bcache.lock[i]);
            b->prev = &bcache.head[id]; // 将缓冲区插入到新的位置
            b->next = bcache.head[id].next;
        }
    }
}

```

```

        b->next->prev = b;
        b->prev->next = b;
        release(&bcache.lock[id]);
        acquiresleep(&b->lock);
        return b;
    }
}
release(&bcache.lock[i]);
}
panic("bget: no buffers");

```

测试实验结果成功通过 bcachetest 和 usertests:

```

$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: bcache: #fetch-and-add 32 #acquire() 13289
lock: bcache: #fetch-and-add 0 #acquire() 11800
lock: bcache: #fetch-and-add 0 #acquire() 22376
lock: bcache: #fetch-and-add 0 #acquire() 21311
lock: bcache: #fetch-and-add 0 #acquire() 308687
lock: bcache: #fetch-and-add 19 #acquire() 182658
lock: bcache: #fetch-and-add 243 #acquire() 192524
lock: bcache: #fetch-and-add 19 #acquire() 145313
lock: bcache: #fetch-and-add 1823 #acquire() 677663
lock: bcache: #fetch-and-add 27 #acquire() 81302
lock: bcache: #fetch-and-add 3 #acquire() 74683
lock: bcache: #fetch-and-add 0 #acquire() 28561
lock: bcache: #fetch-and-add 0 #acquire() 14231
--- top 5 contended locks:
lock: proc: #fetch-and-add 76922153 #acquire() 8298794
lock: proc: #fetch-and-add 48270138 #acquire() 8283013
lock: proc: #fetch-and-add 19681978 #acquire() 8282921
lock: proc: #fetch-and-add 18603515 #acquire() 8279337
lock: proc: #fetch-and-add 16168988 #acquire() 8286275
tot= 2166
test0: OK
start test1
test1 OK
$

```

usertests 结果为:

ALL TESTS PASSED

ALL TESTS PASSED

[实验中遇到的问题和解决方法]

本次实验同样颇具挑战性。首先要考虑到，若仅用一个锁来保证修改操作的原子性，会引发大量冲突，因此需要根据数据块的 `blocknumber` 将其存入哈希表，且哈希表的每个 `bucket` 都配备相应的锁进行保护；其次，双向链表的代码编写较为复杂，特别是在遍历链表时，一旦条件判断或迭代更新出现错误，就可能导致无限循环或访问越界，这就要求我们扎实掌握数据结构相关知识。在本次实验中，我主要通过参考和学习，在教学指导视频的辅助下，顺利完成了实验任务。

[实验心得]

本次实验让我深刻理解了分桶策略对并发缓存系统性能的提升作用。缓冲区缓存作为磁盘与内存间的关键中间层，其锁设计直接影响文件系统的并发效率。通过将全局锁拆分为多个桶级锁，不同块的操作可并行执行，这是“数据分片”思想在系统设计中的典型应用——将共享资源分解为独立子集，降低同步粒度。实验中，哈希函数的选择尤为重要：均匀的哈希分布能避免某些桶成为热点，确保锁争用在各桶间均衡分布。缓冲区迁移机制（跨桶窃取空闲块）则体现了资源动态调度的灵活性，既保证了各桶的独立性，又在局部资源不足时实现全局平衡。此外，双向链表的操作细节（断链与插链）让我认识到，并发数据结构的正确性依赖于对边界条件的精准处理，任何一步链表操作的疏漏都可能导致系统崩溃。这次实验不仅掌握了缓存系统的并发优化方法，更让我体会到操作系统通过“分而治之”与“动态调度”结合，实现高效资源管理的深层逻辑。

Lab8 的实验结果截图：./grade-lab-lock

```
● ajack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./grade-lab-lock
make: 'kernel/kernel' is up to date.
== Test running kalloc test == (35.0s)
== Test   kalloc test: test1 ==
    kalloc test: test1: OK
== Test   kalloc test: test2 ==
    kalloc test: test2: OK
== Test kalloc test: sbrkmuch == kalloc test: sbrkmuch: OK (3.8s)
== Test running bcachetest == (4.1s)
== Test   bcachetest: test0 ==
    bcachetest: test0: OK
== Test   bcachetest: test1 ==
    bcachetest: test1: OK
== Test usertests == usertests: OK (71.8s)
== Test time ==
time: OK
Score: 70/70
```

Lab9:File system

首先，开始 Lab9 时，需要通过以下命令切换实验分支，获取实验资源：

```
git fetch
```

```
git fs
```

```
make clean
```

Large files (moderate)

[实验目的]

Modify `bmap()` so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block. You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode. The first 11 elements of `ip->addrs[]` should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block. You are done with this exercise when `bigfile` writes 65803 blocks and `usertests` runs successfully.

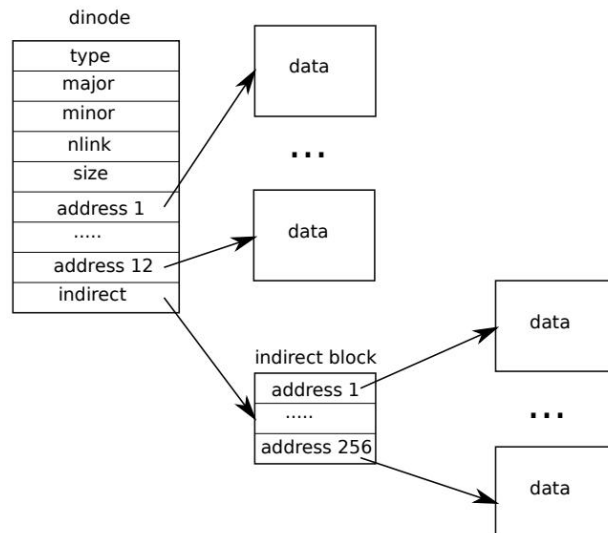
修改 `bmap()`，以便除了直接块和一级间接块之外，它还实现二级间接块。你只需要有 11 个直接块，而不是 12 个，为你的新的二级间接块腾出空间；不允许更改磁盘 inode 的大小。`ip->addrs[]` 的前 11 个元素应该是直接块；第 12 个应该是一个一级间接块(与当前的一样)；13 号应该是你的新二级间接块。当 `bigfile` 写入 65803 个块并成功运行 `usertests` 时，此练习完成。

[实验步骤]

xv6 的文件系统是层级结构的，总共有 7 个层级：

File descriptor
Pathname
Directory
Inode
Logging
Buffer cache
Disk

每一个 Inode 的数据结构如图所示：



如图所示，Inode 当前支持一级间接块数据结构，其本身存储 12 个直接块地址和 1 个间接索引地址，因此可索引 268（12+256）个文件块。本次实验旨在将这一结构改造为二级间接索引，使其能索引 256×256 个文件块，从而让单个文件的最大块数达到 65803（ $256 \times 256 + 256 + 11$ ）。具体改造步骤如下：首先需在 fs.h 中把 NDIRECT 的值从 12 修改为 11，接着定义二级页表相关参数以调整最大文件容量。其中 NDINDIRECT 将表示二级索引块号的总数，其可表示的块号数量为一级索引的平方（ 256×256 ）。

```

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

```

NDIRECT 变了，所以修改 denode 和 inode，将 addrs 的大小加 1：

```

uint addrs[NDIRECT + 2]; // Data block addresses (add one)

```

添加一个二级索引，模仿一级索引的实现，在一级索引中再进行一次一级索引的操作。

```

static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a;
    struct buf *bp;
    if (bn < NDIRECT) // 处理直接块{
        if ((addr = ip->addrs[bn]) == 0) // 如果块地址为 0，则分配
            一个新的块并记录到 inode。
            ip->addrs[bn] = addr = balloc(ip->dev);
        return addr;
    }
}

```

```

    bn -= NDIRECT;
    if (bn < NINDIRECT) // 处理一级间接块{
        // Load indirect block, allocating if necessary.
        if ((addr = ip->addrs[NDIRECT]) == 0) // 如果间接块地址为 0,
        则分配一个新的间接块。
            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 < NDINDIRECT) // 处理二级间接块{
        if ((addr = ip->addrs[NDIRECT + 1]) == 0) // 如果需要分配
        二级内存
            ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
        bp = bread(ip->dev, addr);
        a = (uint *)bp->data;
        if ((addr = a[bn / NINDIRECT]) == 0) // 读取二级间接块
        {
            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");
}

```

首先检查块号是否处于直接块范围内：若对应块地址为 0，就分配新块并将其记录到 inode 中，然后返回该块地址。接着检查块号是否在一级间接块范围内：如果间接块地址为 0，先分配新的间接块，再将间接块内容读取到缓冲区；

当块地址为 0 时，分配新块并记录到间接块中，随后返回该块地址。再检查块号是否处于二级间接块范围内：若二级间接块地址为 0，需分配新的二级间接块，并将其内容读取到缓冲区；当二级间接块中对应的地址为 0 时，分配新的一级间接块并记录到二级间接块中；之后读取一级间接块到缓冲区，若其中的块地址为 0，分配新块并记录到一级间接块中，最后返回该块地址。最后修改 itrunc 函数，为其增添释放二级页表数据块的功能：

```
void itrunc(struct inode *ip)
{
    int i, j;
    struct buf *bp;
    uint *a;
    for (i = 0; i < NDIRECT; i++) // 处理直接块{
        if (ip->addrs[i]){
            bfree(ip->dev, ip->addrs[i]);
            ip->addrs[i] = 0;
        }
    }
    if (ip->addrs[NDIRECT]) // 处理一级间接块{
        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;
    }
    struct buf *bp2;
    uint *a2;
    if (ip->addrs[NDIRECT + 1]) // 处理二级间接块{
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint *)bp->data;
        for (j = 0; j < NINDIRECT; j++){
            if (a[j]){
                bp2 = bread(ip->dev, a[j]);
                a2 = (uint *)bp2->data;
                for (i = 0; i < NINDIRECT; i++){
                    if (a2[i])
                        bfree(ip->dev, a2[i]);
                }
            }
        }
    }
```

```

        brelse(bp2);
        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);
}

```

具体操作如下：通过遍历 **inode** 的直接块、一级间接块和二级间接块，释放所有关联的磁盘块并重置文件大小。

首先处理直接块：遍历 inode 的直接块地址数组 `addrs`，若地址不为 0，则释放该块并将地址置为 0。

接着处理一级间接块：检查一级一级间接块地址是否存在，若存在，将间接块内容读取到缓冲区；遍历间接块中的地址数组，释放所有非零地址对应的块，之后释放该间接块并将其地址置为 0。

最后处理二级间接块：检查二级间接块地址是否存在，若存在，将二级间接块内容读取到缓冲区；遍历二级间接块中的地址数组，对每个非零地址，读取对应的一级间接块到缓冲区；遍历该一级间接块中的地址数组，释放所有非零地址对应的块，随后释放该一级间接块并将其地址置为 0；全部处理完毕后，释放二级间接块并将其地址置为 0。输入如图代码测试结果成功通过 `bigfile` 和 `usertests`(ALL TESTS PASSED):

[illegible]

```
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
```

[实验中遇到的问题和解决方法]

本次实验的问题是 `itrunc` 函数未释放二级间接块，导致文件截断后二级间接块及其指向的一级块未被释放，造成磁盘空间泄漏。通过在 `itrunc` 中递归释放二级块指向的所有一级块，再释放二级块本身，确保所有关联块被正确回收。此外，还遇到缓冲区同步错误，因未在修改间接块后调用 `log_write`，导致索引变更未写入磁盘，通过在分配新间接块后添加 `log_write`，确保了索引的持久性。

[实验心得]

本次实验让我深入理解了文件系统中多级索引对文件大小的扩展作用。`xv6` 原有的直接块 + 一级间接块设计仅支持 268 个块，而通过引入二级间接块，将最大块数提升至 65803 ($11 + 256 + 256 \times 256$)，这种层级索引结构是突破单 `inode` 存储限制的经典方案。实验中，块号的分层映射逻辑尤为关键：不同范围的块号对应不同的索引层级，直接块适用于小文件快速访问，间接块则用于扩展大容量文件，这种“局部直接访问 + 全局间接索引”的设计平衡了效率与扩展性。`bmap` 函数中对二级间接块的处理，体现了递归索引的思想——通过多层间接块的嵌套，将有限的 `inode` 地址空间转化为近乎无限的文件容量（受限于磁盘大小）。此外，`itrunc` 函数的完善让我认识到，文件系统不仅要正确分配资源，还要在释放时进行完整的递归清理，避免资源泄漏。这次实验不仅掌握了多级索引的实现方法，更让我体会到操作系统通过层级结构突破硬件限制，实现

灵活存储管理的精妙思路。

Symbolic links (moderate)

[实验目的]

You will implement the `symlink(char *target, char *path)` system call, which creates a new symbolic link at `path` that refers to file named by `target`. For further information, see the man page `symlink`. To test, add `symlinktest` to the Makefile and run it. Your solution is complete when the tests produce the following output (including `usertests` succeeding).

您将实现 `symlink(char *target, char *path)` 系统调用，该调用在引用由 `target` 命名的文件的路径处创建一个新的符号链接。有关更多信息，请参阅 `symlink` 手册页(注：执行 `man symlink`)。要进行测试，请将 `symlinktest` 添加到 Makefile 并运行它。当测试产生以下输出(包括 `usertests` 运行成功)时，您就完成本作业了。

[实验步骤]

我们要完成一个系统调用，大致的流程框架依旧在 Lab2 中熟知了，按照指导参考的提示，添加一个 `sysylink` 系统调用：

在 `syscall.h` 中添加：

```
#define SYS_symlink 22
```

在 `syscall.c` 中添加：

```
extern uint64 sys_symlink(void);  
[SYS_symlink] sys_symlink,
```

在 `usys.pl` 中添加：

```
entry("symlink");
```

在 `user.h` 中添加：

```
int symlink(char *target, char *path);
```

在 `stat.h` 中新增文件类型 `T_SYMLINK`：

```
#define T_SYMLINK 4
```

在 `fcntl.h` 中新增文件标志位 `O_NOFOLLOW`：

```
#define O_NOFOLLOW 0x600
```

我们就需要真正实现 `sys_symlink()` 函数了：

```
uint64 sys_symlink(void)  
{  
    char path[MAXPATH], target[MAXPATH]; // 存储符号链接的路径和  
    // 存储符号链接指向的目标路径  
    struct inode *ip;
```

```

        if (argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH)
< 0)
            return -1;
        begin_op(); // 开始一个文
件系统操作
        if ((ip = create(path, T_SYMLINK, 0, 0)) == 0) // 创建一个
新的符号链接文件{
            end_op();
            return -1;
        }
        if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) //
将目标路径 target 写入到符号链接文件中{
            iunlockput(ip);
            end_op();
            return -1;
        }
        iunlockput(ip);
        end_op(); // 结束文件系统操作
        return 0;
    }
}

```

首先通过 `argstr` 函数获取系统调用参数：调用 `argstr(0, target, MAXPATH)` 获取第一个参数（目标路径 `target`）并存储到 `target` 数组中；调用 `argstr(1, path, MAXPATH)` 获取第二个参数（符号链接路径 `path`）并存储到 `path` 数组中。接着调用 `begin_op` 函数启动文件系统操作（通常用于保证文件系统一致性），随后调用 `create` 函数创建新的符号链接文件，再通过 `writei` 函数将目标路径 `target` 写入该符号链接文件，最后调用 `iunlockput(ip)` 完成 `inode` 的解锁与释放，并调用 `end_op` 函数结束文件系统操作。最后需修改 `sys_open` 函数：当创建符号链接时，若目标路径本身是符号链接，需递归获取其指向的目标路径，直至找到非符号链接路径。若搜索次数达到阈值，则判定文件打开失败。

```

        else{
            int max_depth = 20, depth = 0; // 设置符号链接解析的最大递
归深度为 20，防止无限循环
            while (1){
                if ((ip = namei(path)) == 0) // 调用 namei 函数解析路径，
返回对应的 inode 指针 ip{
                    end_op();
                    return -1;
                }
                ilock(ip); // 调用 ilock 函数锁定 inode，以便进行安全的读
取和操作
            }
        }
    }
}

```



```

if (ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0)
{
    if (++depth > max_depth) // 递增递归深度 depth{
        iunlockput(ip);
        end_op();
        return -1;
    }
    if (readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH)
    {
        iunlockput(ip); // 调用 readi 函数读取符号链接的目标
        end_op();
        return -1;
    }
    iunlockput(ip);
}
else
    break;

```

路径

输入如图代码测试结果成功通过 symlinktest 和 usertests(ALL TESTS PASSED):

```

$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$

```

```

OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected
                sepc=0x00000000000022ce st
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test 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

```

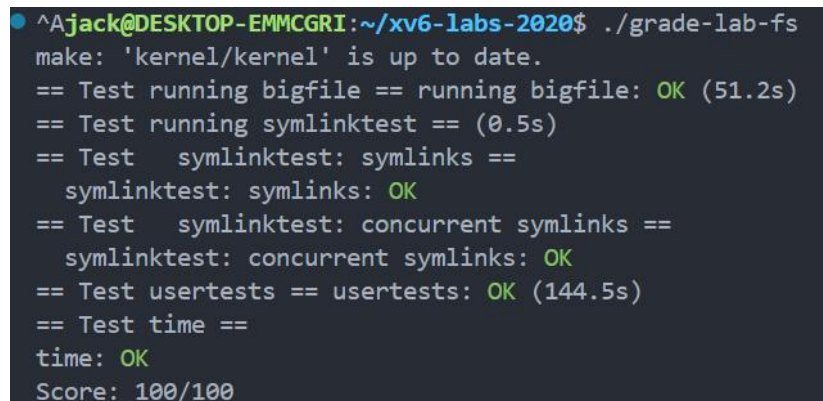
[实验中遇到的问题和解决方法]

经过之前的实践，我对系统调用的实现流程已较为熟悉，本次实验的核心在于实现符号链接相关函数。为避免无限递归循环，需设定合理的最大递归深度（`max_depth`）——若符号链接形成环路或链条过于复杂，可能导致递归过深的问题。同时，为确保多线程或多进程访问相同文件或符号链接时不出现竞态条件，需通过对 `inode` 执行加锁（`ilock`）和解锁（`iunlockput`）操作来保证同步。本次代码编写有一定难度，最终在指导视频和参考文献的辅助下，我顺利完成了实验任务。

[实验心得]

本次实验让我深入理解了符号链接在文件系统中的实现机制。符号链接作为一种特殊的文件类型，其核心是通过存储目标路径字符串，实现对其他文件的间接引用，这种设计为文件系统提供了灵活的路径映射能力。实验中，递归解析逻辑是关键：当打开一个符号链接时，系统需要多次解析路径直至找到最终目标文件，而深度限制则是防止恶意循环链接导致系统异常的必要保护。符号链接与硬链接的本质区别也尤为明显——硬链接依赖 `inode` 的引用计数，而符号链接是独立文件，存储路径信息，这使得符号链接可以跨文件系统，且删除目标文件会导致链接失效。此外，`O_NOFOLLOW` 标志的支持体现了操作系统对访问控制的细致考量，允许程序选择是否跟随链接，增强了文件操作的安全性。这次实验不仅掌握了系统调用的扩展方法，更让我体会到文件系统通过抽象数据类型（如特殊文件类型）实现复杂功能的设计思想。

Lab9 的实验结果截图：./grade-lab-fs



```
^Ajack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./grade-lab-fs
make: 'kernel/kernel' is up to date.
== Test running bigfile == running bigfile: OK (51.2s)
== Test running symlinktest == (0.5s)
== Test  symlinktest: symlinks ==
    symlinktest: symlinks: OK
== Test  symlinktest: concurrent symlinks ==
    symlinktest: concurrent symlinks: OK
== Test usertests == usertests: OK (144.5s)
== Test time ==
    time: OK
Score: 100/100
```

Lab10:Mmap

首先，开始 Lab10 时，需要通过以下命令切换实验分支，获取实验资源：

```
git fetch
```

```
git mmap
```

```
make clean
```

mmap (hard)

[实验目的]

You should implement enough mmap and munmap functionality to make the mmaptest test program work. If mmaptest doesn't use a mmap feature, you don't need to implement that feature.

您应该实现足够的 mmap 和 munmap 功能，以使 mmaptest 测试程序正常工作。如果 mmaptest 不会用到某个 mmap 的特性，则不需要实现该特性。

[实验步骤]

在 makefile 文件中添加 mmaptest

```
$U/_mmaptest\
```

在 syscall.h 中添加：

```
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
[SYS_mmap]    sys_mmap,
[SYS_munmap]  sys_munmap,
```

在 usys.pl 中添加：

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

在 user.h 中添加：

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

按照提示 3 的要求，我们需要跟踪 mmap 为每个进程映射的内容。这需要定义一个与第 15 课中描述的 VMA（虚拟内存区域）相对应的结构体，用于记录 mmap 创建的虚拟内存范围的地址、长度、权限、关联文件等信息。VMA（虚拟内存区域）是操作系统管理进程虚拟内存的关键概念，它代表进程地址空间中一个连续的区域，且该区域具有统一的属性（如读、写、执行权限）。因此，我们需要定义 VMA 结构体，并将其添加到进程结构体中：

```
#define VMASIZE 16
```

```

struct vma
{
    struct file *file; // 指向映射文件的指针
    int fd;            // 文件描述符
    int used;          // 标志该 VMA 是否被使用
    uint64 addr;       // 虚拟内存区域的起始地址
    int length;        // 虚拟内存区域的长度
    int prot;          // 内存保护标志
    int flags;         // 映射标志
    int offset;        // 文件偏移量
};

```

在结构体 struct proc 中加 VMA:

```

struct vma vma[VMASIZE];

```

需要 usertrap, 处理 page fault:

```

else // 如果地址在合法范围内{
    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].length) // 找到包
含该地址的 VMA {
            vma = &p->vma[i]; // 将该 VMA 记录在 vma 变量中
            break;
        }
    }

    if (vma) // 如果找到有效的 VMA, 将虚拟地址对齐到页边界
    {
        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;
        }
    }
}
}
}

```

当发生页面错误时，需要对缺页中断进行处理，依据进程的虚拟内存区域（VMA）信息，将缺失的页面加载到内存中。具体而言，要在进程的虚拟地址空间中找到对应的虚拟内存区域，从文件中加载缺失的页面至内存，并将该页面映射到进程的地址空间。若上述过程中任何一步出现失败，都需将进程标记为已被杀死。接下来，我们需要在`kernel/sysfile.c`中具体实现两个函数：`sys_mmap`和`sys_munmap`：

```

// 用于将文件映射到进程的地址空间
uint64 sys_mmap(void)
{
    uint64 addr;
    int length, prot, flags, fd, offset;
    struct proc *p = myproc();
    struct file *file;
    //// 获取系统调用参数，如果任何一个参数提取失败，返回 -1
    if (argaddr(0, &addr) || argint(1, &length) || argint(2, &prot)
||
        argint(3, &flags) || argfd(4, &fd, &file) || argint(5,
&offset))
        return -1;
    // 检查文件是否可写，保护标志是否需要写权限，以及映射标志是否为
共享映射
    // 如果条件不满足，返回 -1
    if (!file->writable && (prot & PROT_WRITE) && flags ==
MAP_SHARED)
        return -1;
    // 将映射长度向上舍入到最近的页边界
    length = PGROUNDUP(length);
    // 检查是否映射的长度超过了进程的最大虚拟地址空间
    if (p->sz > MAXVA - length)
        return -1;
    // 遍历进程的虚拟内存区域（VMA）数组，找到一个未使用的 VMA 条目

```

```

for (int i = 0; i < VMASIZE; i++)
{
    if (p->vma[i].used == 0){
        // 设置该 VMA 条目的参数
        p->vma[i].used = 1;
        p->vma[i].addr = p->sz;
        p->vma[i].length = length;
        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 += length;
        return p->vma[i].addr;
    }
}
return -1;
}

```

‘sys_mmap’函数的作用是将文件或内存区域映射到进程的虚拟地址空间。在参考指导下，我们需要完成参数提取与验证、权限检查、长度调整及范围调整等操作。最后，遍历当前进程的虚拟内存区域数组‘vma’，寻找未使用的条目来分配虚拟内存区域。‘mmap’用于将文件或匿名内存映射到进程的虚拟地址空间，而‘munmap’则用于取消这种映射。接下来，我们需要编写‘sys_munmap’函数：

```

// 通过取消进程地址空间中指定区域的映射，释放相关资源
uint64
sys_munmap(void)
{
    uint64 addr;
    int length;
    struct proc *p = myproc();
    struct vma *vma = 0;
    // 从用户空间获取参数 addr 和 length
    if (argaddr(0, &addr) || argint(1, &length))
        return -1;
    addr = PGROUNDDOWN(addr);
    length = PGROUNDUP(length);
    // 遍历当前进程的虚拟内存区域（VMA）数组
    for (int i = 0; i < VMASIZE; i++){
        if (addr >= p->vma[i].addr || addr < p->vma[i].addr +
p->vma[i].length){
            // 找到对应的 VMA

```

```

        vma = &p->vma[i];
        break;
    }
}
// 如果没有找到对应的 VMA，则返回 0
if (vma == 0)
    return 0;
// 如果找到的 VMA 的起始地址等于要取消映射的地址
if (vma->addr == addr){
    vma->addr += length;
    vma->length -= length;
    // 如果 VMA 是共享映射，则将内存内容写回文件
    if (vma->flags & MAP_SHARED)
        fwrite(vma->file, addr, length);
    // 取消进程页表中的映射
    uvmunmap(p->pagetable, addr, length / PGSIZE, 1);
    // 如果 VMA 的长度变为 0，则关闭文件并标记 VMA 为未使用
    if (vma->length == 0){
        fclose(vma->file);
        vma->used = 0;
    }
}
return 0;
}

```

`sys_munmap`函数的功能是取消进程地址空间中指定区域的映射并释放相关资源。若映射区域为共享类型，还需将修改的内容写回文件。通过更新和管理进程的虚拟内存区域数组，确保内存得到合理利用及资源被有效释放。最后需要修改`uvmcopy`和`uvmunmap`函数，避免因操作不合法而导致系统崩溃(panic):

```

if ((*pte & PTE_V) == 0)
    // panic("uvmunmap: not mapped");
    continue;
if ((*pte & PTE_V) == 0)
    // panic("uvmcopy: page not present");
    continue;

```

输入如图代码测试结果成功通过 mmaptest 和 usertests(ALL TESTS PASSED):


```
xv6 kernel is booting

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

```
test kernmem: OK
test sbrkfail: OK
test sbrkarg: OK
test validatetest: OK
test stacktest: OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test 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
$ □
```

[实验中遇到的问题和解决方法]

本次实验的难度依然较大。理解 `mmap` 和 `munmap` 技术是完成实验的前提——二者是一对用于管理进程虚拟内存映射的系统调用，呈现出互补关系：`mmap`

负责将文件或匿名内存映射到进程的虚拟地址空间，而 `munmap` 则用于取消这种映射。实验中的另一个关键点是 `VMA`（虚拟内存区域）结构体的设计。起初我对此毫无头绪，但在深入理解 `mmap` 和 `munmap` 技术的核心逻辑后，结合参考资料的指导，最终完成了结构体的设计。整个过程既需要扎实掌握虚拟内存管理的理论知识，也需要结合实际场景梳理数据结构与系统调用的关联逻辑。

[实验心得]

本次实验让我深刻理解了内存映射（`mmap`）机制将文件与进程虚拟内存关联的核心原理。`mmap` 通过将文件内容直接映射到进程地址空间，使文件操作转化为内存操作，简化了 `IO` 流程并提高了效率。实验中，`VMA` 的设计是关键——它作为进程与文件映射的元数据，记录了映射的范围、权限等信息，为缺页中断处理提供了依据，这种“按需加载”（懒加载）的策略与之前的 `Lazy allocation` 异曲同工，体现了操作系统“延迟操作”以优化资源利用的思想。

共享映射与私有映射的区别也尤为重要：共享映射（`MAP_SHARED`）的修改会同步到文件，而私有映射（未实现）的修改仅在进程内有效，这种区分满足了不同的协作需求。`munmap` 时的资源清理（解除映射、同步文件、释放 `VMA`）则展示了操作系统对资源生命周期的完整管理，确保映射取消后不残留无效页表项或文件引用。

这次实验不仅掌握了 `mmap` 的实现细节，更让我体会到操作系统通过虚拟内存抽象，将文件 `IO` 与内存操作统一起来的精妙设计，为理解现代操作系统的内存与文件交互机制提供了重要视角。

Lab10 的实验结果截图：./grade-lab-mmap

```
a1ack@DESKTOP-EMMGR1:~/xv6-labs-2020$ ./grade-lab-mmap
make: 'kernel/kernel' is up to date.
== Test running mmaptest == (1.6s)
== Test  mmaptest: mmap f ==
mmaptest: mmap f: OK
== Test  mmaptest: mmap private ==
mmaptest: mmap private: OK
== Test  mmaptest: mmap read-only ==
mmaptest: mmap read-only: OK
== Test  mmaptest: mmap read/write ==
mmaptest: mmap read/write: OK
== Test  mmaptest: mmap dirty ==
mmaptest: mmap dirty: OK
== Test  mmaptest: not-mapped unmap ==
mmaptest: not-mapped unmap: OK
== Test  mmaptest: two files ==
mmaptest: two files: OK
== Test  mmaptest: fork test ==
mmaptest: fork test: OK
== Test usertests == usertests: OK (61.7s)
== Test time ==
time: OK
Score: 140/140
```

Lab11:Networking

首先，开始 Lab11 时，需要通过以下命令切换实验分支，获取实验资源：

```
git fetch
```

```
git net
```

```
make clean
```

Your Job (hard)

[实验目的]

Your job is to complete `e1000_transmit()` and `e1000_recv()`, both in `kernel/e1000.c`, so that the driver can transmit and receive packets. You are done when `make grade` says your solution passes all the tests.

您的工作是在 `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。当 `make grade` 表示您的解决方案通过了所有测试时，您就完成了。

[实验步骤]

根据实验指导要求，我们需要在 `'kernel/e1000.c'` 中实现两个函数：`'e1000_transmit'` 和 `'e1000_recv'`。其中，`'e1000_transmit'` 函数的功能是将以太网帧传递到 `e1000` 网络设备的发送描述符环，并启动发送流程；`'e1000_recv'` 函数则用于接收来自 `e1000` 网络设备的数据包，再将其传递给网络栈进行后续处理。

```
int e1000_transmit(struct mbuf *m)
{
    // 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);
    // 获取 e1000_lock 锁，这个锁用于同步对 e1000 网络设备的访问，
    // 以防止并发访问带来的问题。

    uint32 next_index = regs[E1000_TDT];
    // 从寄存器中读取当前可用的 TX（发送）描述符环的下一个索引。
    if ((tx_ring[next_index].status & E1000_TXD_STAT_DD) == 0)
    {
```

```

    // 检查当前下一个描述符的状态是否为 "E1000_TXD_STAT_DD"（表示描述符是否可用）。
    // 如果不可用，则说明发送队列已满，无法继续发送新的数据包，所以需要释放锁并返回-1，表示发送失败。
    release(&e1000_lock);
    return -1;
}
if (tx_mbufs[next_index])
    mbuf_free(tx_mbufs[next_index]);
// 检查下一个描述符的 mbuf 指针是否非空，如果非空，则释放之前可能存储在该描述符中的 mbuf。
tx_ring[next_index].addr = (uint64)m->head;
tx_ring[next_index].length = (uint16)m->len;
// 将 mbuf 中的数据包内容的头部地址和长度存储到下一个描述符中，以便将数据包发送。
tx_ring[next_index].cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;
// 设置下一个描述符的命令字段。其中 EOP 表示该描述符为一个完整数据包的结束描述符，
// RS 表示发送时将报告状态（Report Status），即在数据包发送完成后触发中断。
tx_mbufs[next_index] = m;
// 将当前 mbuf 存储到下一个描述符对应的缓冲区中，以便在发送完成后能够释放 mbuf。
regs[E1000_TDT] = (next_index + 1) % TX_RING_SIZE;
// 更新寄存器中的 TDT（Transmit Descriptor Tail）指针，使其指向下一个可用的描述符。
// 这样做后，该描述符就准备好发送数据包了。
release(&e1000_lock);
// 释放 e1000_lock 锁，允许其他线程再次访问 e1000 网络设备。
return 0;
// 返回 0 表示数据包发送成功。
}

```

该函数接收`struct mbuf *m`作为参数，执行流程如下：首先获取`e1000_lock`锁，以实现对 e1000 网络设备访问的同步；接着从寄存器中读取发送描述符环的下一个可用索引，并检查描述符状态——若状态显示不可用，表明发送队列已满，无法继续发送新数据包，此时需释放锁并返回-1，表示发送失败。之后，将`mbuf`中数据包内容的头部地址和长度存入下一个描述符，并设置该描述符的命令字段；随后更新寄存器中的 TDT 指针，使其指向新的下一个可用描述符，完成这一步后，当前描述符即准备好发送数据包。最后，释放之前获取的锁。

```
static void
```

```

e1000_recv(void)
{
    uint32 next_index = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    // 从寄存器中读取当前可用的 RX（接收）描述符环的下一个索引。

    while (rx_ring[next_index].status & E1000_RXD_STAT_DD)
    {
        // 循环检查下一个描述符的状态是否为 "E1000_RXD_STAT_DD"（表示描述符中有新的数据包到达）。
        if (rx_ring[next_index].length > MBUF_SIZE)
        {
            panic("MBUF_SIZE OVERFLOW!");
        }
        // 检查数据包的长度是否超过了 mbuf 的最大大小（MBUF_SIZE）。如果超过了，就触发 panic（内核恐慌）。
        rx_mbufs[next_index]->len = rx_ring[next_index].length;
        // 将接收到的数据包长度存储到对应的 mbuf 中。
        net_rx(rx_mbufs[next_index]);
        // 调用 net_rx() 函数，将 mbuf 传递给网络栈处理，以进行后续的数据包处理和解析。
        rx_mbufs[next_index] = mbufalloc(0);
        // 分配一个新的 mbuf，并将其存储到接收描述符的缓冲区中，以准备接收下一个数据包。
        rx_ring[next_index].addr =
        (uint64)rx_mbufs[next_index]->head;
        // 将新的 mbuf 的头部地址存储到接收描述符中，以便接收数据包时能够正确写入数据。
        rx_ring[next_index].status = 0;
        // 将接收描述符的状态字段清零，表示该描述符已被处理完毕，可以用于接收新的数据包。
        next_index = (next_index + 1) % RX_RING_SIZE;
        // 更新下一个可用接收描述符的索引，准备处理下一个接收到的数据包。
    }
    regs[E1000_RDT] = (next_index - 1) % RX_RING_SIZE;
    // 更新寄存器中的 RDT（Receive Descriptor Tail）指针，使其指向最后一个处理过的接收描述符。
}

```

该函数的执行流程如下：首先从寄存器中读取接收描述符环（RX 描述符环）的下一个可用索引——通过将 RDT 寄存器的值加 1 后对接收描述符环的大小取模获得。随后循环检查该索引对应的描述符状态是否为'E1000_RXD_STAT_DD'，以此判断是否有新数据包到达。接着检查接收到的数据包长度：若超过 mbuf

的最大容量，则触发 panic（因数据包过大无法处理）；否则将数据包长度存入对应的 mbuf 中，并调用`net_rx()`函数将 mbuf 传递给网络栈，由网络栈完成数据包的处理与解析。之后需分配新的 mbuf，将其存入接收描述符的缓冲区以准备接收下一个数据包，同时将新 mbuf 的头部地址写入接收描述符，确保后续数据包能被正确写入。最后清零描述符的状态字段，并更新接收描述符索引与 RDT 寄存器。验证该实验结果需使用两个终端：首先在第一个终端中输入`make server`：

```
jack@DESKTOP-EMCGRI:~/xv6-labs-2020$ make server
python3 server.py 26099
listening on localhost port 26099
[]
```

再在第二个终端中输入 make qemu，然后进行 nettests 测试(all tests passed):

```
$ 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.
```

```
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
a message from xv6!
```


[实验中遇到的问题和解决方法]

本次实验内容与计算机网络知识关联紧密，由于本学期修习过计算机网络课程，我对这部分知识较为熟悉。不过在代码编写过程中仍遇到了一些问题：

在实现 `e1000_transmit` 函数时，需要注意在函数开始处获取 `e1000_lock` 锁，并在函数结束前释放该锁，以保证对网络设备访问的同步性。

我再测试的时候发现挂着代理就会错误，原来代理影响网络实验的原因是代理服务器改变了 DNS 解析结果，使 `pdos.csail.mit.edu` 被解析为 `198.18.0.57`，而非测试程序预期的 IP 地址。代理还改变了网络路由路径，导致测试失败。解决方法是临时关闭代理，或修改测试程序中的预期 IP 地址。

此外，本次实验的结果验证需要开启两个不同的终端分别进行操作输入。

[实验心得]

本次实验让我深入理解了网络设备驱动与硬件交互的底层机制。`e1000` 网卡通过描述符环（TX/RX ring）实现高效的数据包传输，驱动程序需通过读写寄存器和描述符与硬件协同，这种“内存映射 IO”模式是设备驱动的典型设计。实验中，发送流程的核心是维护描述符环的状态：仅当硬件完成上一个数据包发送（DD 标志置位）时，才能填充新的描述符并更新尾指针，这种“生产者 - 消费者”模型确保了数据传输的有序性。接收流程则通过循环检查 DD 标志发现新数据包，处理后及时更新接收描述符，为下一次接收做准备，体现了硬件中断与软件轮询结合的高效处理方式。

描述符环的环形结构设计尤为巧妙：通过取模运算实现索引的循环复用，避免了线性数组的地址溢出问题，同时让硬件和软件可以独立推进指针，减少了同步开销。此外，`mbuf` 作为数据包的内存载体，其管理（分配、释放、传递给网络栈）是驱动与上层协议栈交互的关键，确保了数据在 `kernel` 中的顺畅流转。

这次实验不仅掌握了网络驱动的基本实现方法，更让我体会到操作系统中硬件抽象层的重要性 —— 驱动程序通过封装硬件细节，为上层提供统一的接口（如 `net_rx`），使复杂的网络交互变得可控可扩展。

。

Lab11 的实验结果截图：./grade-lab-net


```
jack@DESKTOP-EMMCGRI:~/xv6-labs-2020$ ./grade-lab-net
make: 'kernel/kernel' is up to date.
== Test running nettests == (1.5s)
== 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
```

本项目的 github 源码托管链接:

<https://github.com/Jackey0903/XV6-Operation-System.git>