



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: 42028701

学 号: 2253744

姓 名: 林觉凯

指导老师: 王冬青

# 目录

<b>Lab0:Environment Setup</b>	4
<b>Lab1:Utilities</b>	9
Boot xv6 (easy)	9
sleep (easy)	11
pingpong (easy)	12
primes (moderate)/(hard)	15
find (moderate)	17
xargs (moderate)	21
<b>Lab2:System Calls</b>	23
System call tracing (moderate)	23
Sysinfo (moderate)	26
<b>Lab3:Page Tables</b>	28
Print a page table (easy)	29
A kernel page table per process (hard)	31
Simplify copyin/copyinstr (hard)	36
<b>Lab4:Traps</b>	39
RISC-V assembly (easy)	39
Backtrace (moderate)	42
Alarm (hard)	44
<b>Lab5:Lazy allocation</b>	48
Eliminate allocation from sbrk() (easy)	48

Lazy allocation (moderate).....	49
Lazytests and Usertests (moderate).....	52
<b>Lab6:Copy-on-Write Fork for xv6</b> .....	53
Implement copy-on write(hard).....	53
<b>Lab7:Multithreading</b> .....	59
Uthread: switching between threads (moderate).....	59
Using threads (moderate).....	63
Barrier(moderate).....	64
<b>Lab8:Locks</b> .....	66
Memory allocator (moderate).....	67
Buffer cache (hard).....	70
<b>Lab9:File system</b> .....	73
Large files (moderate).....	73
Symbolic links (moderate).....	79
<b>Lab10:Mmap</b> .....	82
mmap (hard).....	82
<b>Lab11:Networking</b> .....	88
Your Job (hard).....	89

# 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。

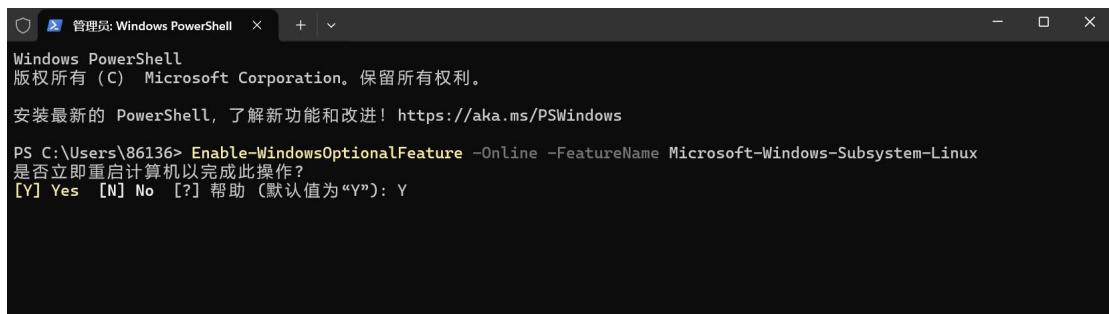
这个实验的目的主要是完成安装 Ubuntu20.04、配置 XV6 系统和 github 远端仓库的建立。

## [实验步骤]

### 1. 安装 Ubuntu20.04

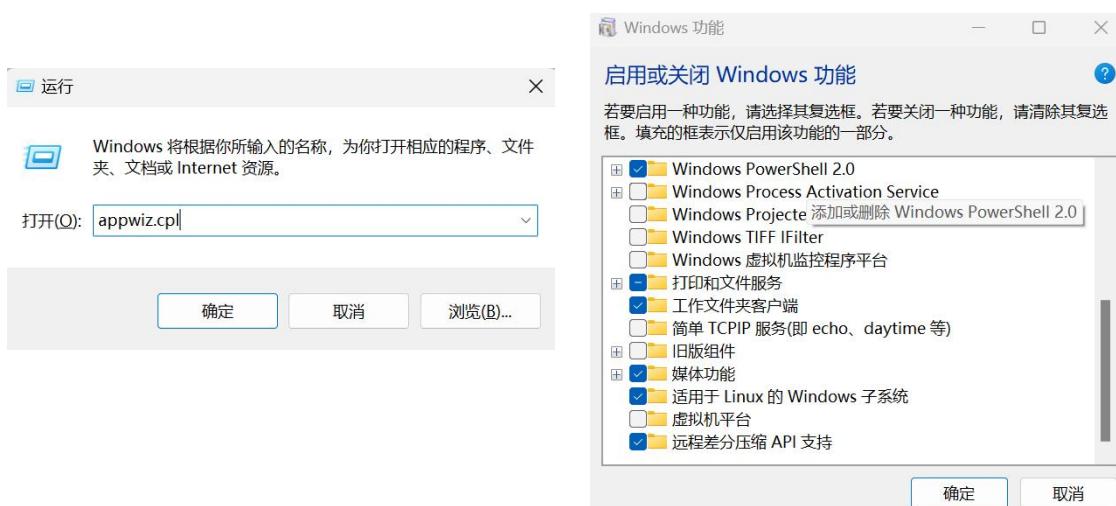
首先我们需要开启个人电脑上的 WSL 支持。

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。

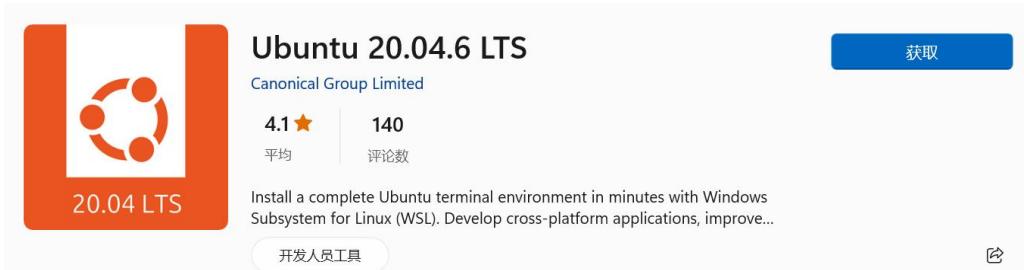


```
管理壳: Windows PowerShell
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。
安装最新的 PowerShell, 了解新功能和改进! https://aka.ms/PSWindows
PS C:\Users\86136> Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
是否立即重启计算机以完成此操作?
[Y] Yes [N] No [?] 帮助 (默认值为“Y”): Y
```

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。



之后在微软应用商店里搜索 ubuntu，选择自己喜欢的版本的 Ubuntu 安装即可。本次实验选择的是 Ubuntu 20.04 LTS。



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。

```
ljk@LJK'sPC: ~
Installing, this may take a few minutes...
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: Ljk
adduser: Please enter a username matching the regular expression configured
via the NAME_REGEX[_SYSTEM] configuration variable. Use the --force-badname'
option to relax this check or reconfigure NAME_REGEX.
Enter new UNIX username: ljk
New password:
Retype new password:
passwd: password updated successfully
Installation successful!
适用于 Linux 的 Windows 子系统现已在 Microsoft Store!
你可以通过运行 "wsl.exe --update" 进行升级 或通过访问 https://aka.ms/wslstorepage
从 Microsoft Store 安装 WSL 将提供最新的 WSL 更新, faster.
有关详细信息, 请访问 https://aka.ms/wslstoreinfo<
>

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.10.16.3-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sun Jul 14 23:13:59 CST 2024

System load: 0.09      Processes:          8
Usage of /: 0.5% of 250.98GB  Users logged in:   0
Memory usage: 2%           IPv4 address for eth0: 172.26.141.208
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/ljk/.hushlogin file.
```

## 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。

```
ljk@LJK'sPC: ~
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.10.16.3-microsoft-standard-WSL2 x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

System information as of Mon Jul 15 00:21:27 CST 2024

System load: 0.0 Processes: 8
Usage of /: 1.2% of 250.98GB Users logged in: 0
Memory usage: 4% IPv4 address for eth0: 172.26.141.208
Swap usage: 0%

Expanded Security Maintenance for Applications is not enabled.

136 updates can be applied immediately.
93 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

*** System restart required ***

This message is shown once a day. To disable it please create the
/home/ljk/.hushlogin file.
ljk@LJK 'sPC: $ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
[sudo] password for ljk:
Reading package lists... Done
Building dependency tree
Reading state information... Done
binutils-riscv64-linux-gnu is already the newest version (2.34-6ubuntul.9).
binutils-riscv64-linux-gnu set to manually installed.
build-essential is already the newest version (12.ubuntul.1).
git is already the newest version (1:2.25.1-ubuntu3.13).
git set to manually installed.
qemu-system-misc is already the newest version (1:4.2-3ubuntu6.29).
gdb-multiarch is already the newest version (9.2-0ubuntul'20.04.2).
The following packages were automatically installed and are no longer required:
  gnuile-2.0-libs libgcc1
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  cpp-9-riscv64-linux-gnu cpp-riscv64-linux-gnu gcc-9-cross-base-ports gcc-9-riscv64-linux-gnu gcc-9-riscv64-linux-gnu-base libgcc-9-dev-riscv64-cross
Suggested packages:
  gcc-9-locales cpp-doc gcc-9-doc autoconf automake libtool flex bison gdb-riscv64-linux-gnu gcc-doc
The following NEW packages will be installed:
  cpp-9-riscv64-linux-gnu cpp-riscv64-linux-gnu gcc-9-cross-base-ports gcc-9-riscv64-linux-gnu gcc-9-riscv64-linux-gnu-base gcc-riscv64-linux-gnu libgcc
```

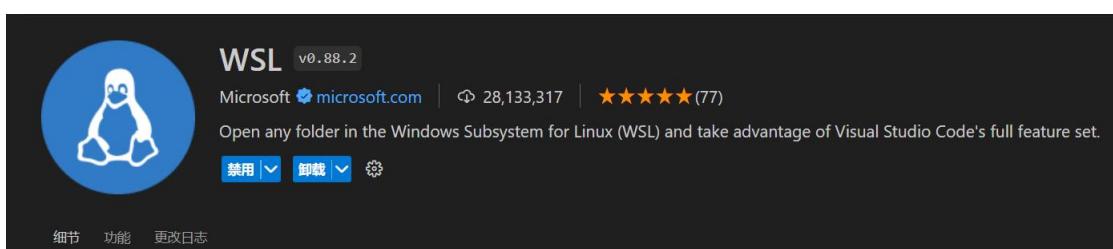
测试安装，在终端输入命令 riscv64-unknown-elf-gcc --version 和命令 qemu-system-riscv64 --version。

```
ljk@LJK 'sPC: ~/xv6-labs-2020$ 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.

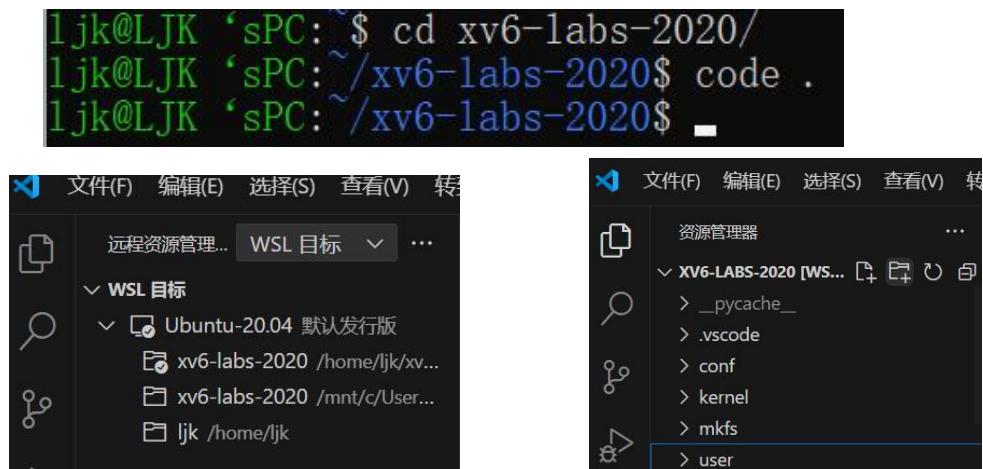
ljk@LJK 'sPC: ~/xv6-labs-2020$ qemu-system-riscv64 --version
QEMU emulator version 4.2.1 (Debian 1:4.2-3ubuntu6.29)
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
ljk@LJK 'sPC: ~/xv6-labs-2020$
```

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。



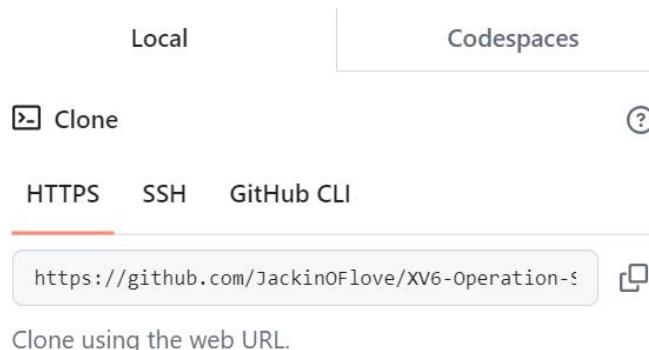
完成以上环境的配置，便可以在我们的 VSCode 中修改代码，在相对应的终端进行调试，观察实验结果。

### 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](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/JackinOflove/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，我遇到了许多问题，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。

- 在 github 仓库 push 的时候，老是给我报错：

```
fatal: unable to access 'https://github.com/JackinOflove/XV6-Operation-System.git':  
Could not resolve host: github.com,
```

在查找资料和询问 Chatgpt 之后，才发现 git 在拉取或者提交项目时，中间会有 git 的 http 和 https 代理，但是我们本地环境本身就有 SSL 协议了，所以取消 git 的 https 代理即可，不行再取消 http 的代理。

解决方法为输入以下两行命令：

```
//取消 http 代理
```

```
git config --global --unset http.proxy
```

```
//取消 https 代理  
git config --global --unset https.proxy  
之后继续 git 提交即可完成相应操作。
```

## [实验心得]

搭建 xv6 实验环境是一个学习操作系统基础的好机会，搭建 xv6 实验环境需要安装一些必要的工具，如 make 和 RISC-V 工具链。这一步让我意识到了解并熟悉开发工具链的重要性，这对后续的编译和调试工作至关重要。

同时，搭建 xv6 环境并非一帆风顺，期间遇到了一些问题。例如，工具链的兼容性问题以及权限问题等。每一次解决问题的过程都让我对系统环境和工具链有了更深的理解和掌握，为我之后的实验打好基础。

最后，以上环境的搭建与配置过程中很可能遇到其他不一样的问题，都需要我们一步一步查错纠错。

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

```
ljk@LJK 'sPC:/mnt/c/Users/86136$ sudo apt update  
[sudo] password for ljk:  
Hit:1 http://security.ubuntu.com/ubuntu focal-security InRelease  
Hit:2 http://archive.ubuntu.com/ubuntu focal InRelease  
Get:3 http://archive.ubuntu.com/ubuntu focal-updates InRelease [128 kB]  
Hit:4 http://archive.ubuntu.com/ubuntu focal-backports InRelease  
Fetched 128 kB in 3s (43.1 kB/s)  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
140 packages can be upgraded. Run 'apt list --upgradable' to see them.  
ljk@LJK 'sPC:/mnt/c/Users/86136$ -
```

```

1jk@LJK 'sPC:/mnt/c/Users/86136$ sudo apt install build-essential gcc make perl dkms git gcc-riscv64-unknown-elf gdb-m
Reading package lists... Done
Building dependency tree
Reading state information... Done
gcc is already the newest version (4:9.3.0-1ubuntu2).
build-essential is already the newest version (12.8ubuntu1.1).
git is already the newest version (1:2.25.1-1ubuntu3.13).
git set to manually installed.
qemu-system-misc is already the newest version (1:4.2-3ubuntu6.29).
gdb-multiarch is already the newest version (9.2-0ubuntu1~20.04.2).
The following packages were automatically installed and are no longer required:
  guile-2.0-libs libgc1c2
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  binutils-riscv64-unknown-elf devtools libperl5.30 linux-headers-5.4.0-189 linux-headers-5.4.0-189-generic
    linux-headers-generic perl-base perl-modules-5.30
Suggested packages:
  debtags menu make-doc perl-doc libterm-readline-gnu-perl | libterm-readline-perl-perl libb-debug-perl
  liblocale-codes-perl
The following packages will be REMOVED:
  make-guile
The following NEW packages will be installed:
  binutils-riscv64-unknown-elf devtools dkms gcc-riscv64-unknown-elf linux-headers-5.4.0-189
    linux-headers-5.4.0-189-generic linux-headers-generic make
The following packages will be upgraded:
  libperl5.30 perl perl-base perl-modules-5.30
4 upgraded, 8 newly installed, 1 to remove and 136 not upgraded.

```

在 WSL 终端进入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
```

```

1jk@LJK 'sPC:~/xv6-labs-2020$ cd xv6-labs-2020
1jk@LJK 'sPC:~/xv6-labs-2020$ git checkout util
Already on 'util'
Your branch is up to date with 'origin/util'.

```

我们这时变可以在 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：

```

1jk@LJK 'sPC: ~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -k
io-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 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
sleep      2 15 22656
find       2 16 25632
pingpong   2 17 23592
xargs      2 18 24400
primes     2 19 26040
stressfs   2 20 23872
usertests  2 21 147512
grind      2 22 37984
wc         2 23 25104
zombie     2 24 22272
console    3 25 0
QDMXNjy5   2 26 0
b0iib70J   1 27 64
HscKCUQh   1 31 48
a          1 33 48
c          1 35 48
b          2 37 6
IuHGYYLn  2 38 0
DagBBnE8   1 39 64
HpjnKbIk   1 43 48
$ QEMU: Terminated
1jk@LJK 'sPC: ~/xv6-labs-2020$
```

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]

第一个实验的内容较为简单，没有遇到什么问题。

## [实验心得]

这是一个简单的小实验，但是也是非常基础的。这些基本的步骤在后面的每一个实验中都要运用到，这次实验为后续的实验打下了基础。

## 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：

```
● 1jk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.9s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
```

## [实验中遇到的问题和解决方法]

该实验的内容也较为简单，依照指导的要求，没有遇到什么问题。

## [实验心得]

首先，通过这个实验，我学习了如何为 xv6 操作系统添加新的程序；其次，我在 vscode 终端运行程序，通过理解 argc 和 argv 命令行参数，我可以编写更灵活和动态的命令行程序来使用系统调用来控制进程的行为。

## 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);
    }
```

这个程序展示了父子进程之间的简单通信模型。子进程向管道写入数据，然后父进程从管道读取数据，模拟了一个“ping-pong”的通信过程。子进程负责写，所以在写的时候需要关闭读端，写完之后在关闭自己的写端；父进程负责读，所以在读的时候需要等待子进程写完，读之前关闭自己的写端，读完后再关闭自己的读端。`fork` 是 Unix 系统调用，它用于创建一个新的进程。`fork()`: 创建一个新的进程。新的进程是调用进程的副本。子进程: `fork()` 返回 0。父进程: `fork()` 返回新创建的子进程的进程 ID (PID)。

在 make qemu 后执行 pingpong 命令：

查看正确得分：

```
● ljk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (0.4s)
(Old xv6.out.pingpong failure log removed)
```

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

## [实验中遇到的问题和解决方法]

本次实验的过程中子进程写和父进程读的代码较好编写，主要是要区分子进程和父进程，即读懂 `fork()` 函数是关键。`fork()` 函数通过返回值 0 或 1 来区别子进程和父进程，即我们可以在代码中只用 `if-else` 语句来区别子进程和父进程的具体行为，通过这个来编写子进程和父进程相应的代码。

## [实验心得]

这个实验的主要目标是利用 `fork` 和 `pipe` 系统调用来实现父子进程之间的简单通信。在实现这个程序的过程中，我首先理解了 `fork` 如何创建一个新进程，以及新进程如何与父进程共享相同的代码和数据空间。

通过 `pipe`，我学会了如何在进程间创建一个通信通道，使用文件描述符进行读写操作。实验中，我实现了一个子进程向父进程发送消息的功能。具体步骤是，父进程创建管道并调用 `fork` 生成子进程，然后子进程通过管道发送一个字符，而父进程读取这个字符并打印相应的信息。

## 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(翻译在下面), 该网页中间的图片和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
                    write(fd[1], &num, 4);
            }
        }
    }
}
```

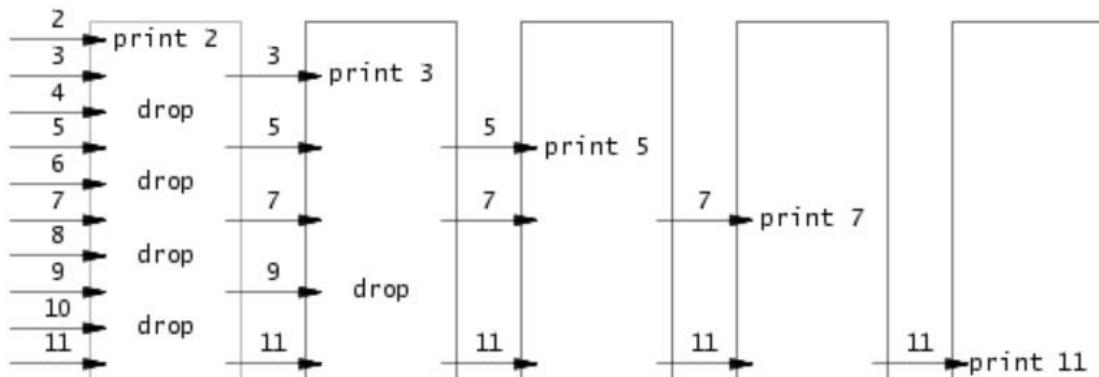
写入新管道

```

        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);
}

```

这段代码还是有一定难度的。首先，我们需要在 main 函数中，创建一个管道 fd，然后通过 fork 创建了一个子进程。父进程遍历 2~35 写入管道写端，在子进程中调用 subProcess(fd) 函数，该函数将负责处理管道中的数据。



在 subProcess 函数中，首先关闭了传入的管道的写端 oldFd[1]，保留读端用于读取数据。接着尝试从管道中读取一个整数，作为初始的质数 prime。如果成

功读取到数据，将其输出到标准输出，并创建一个新的管道 fd 和一个子进程。子进程继续递归调用 subProcess(fd) 函数，处理新管道中的数据。

在父进程中，关闭新管道的读端 fd[0]，然后从旧管道中读取剩余的数据。对于每个读取到的数字 num，检查是否能被当前的质数 prime 整除，如果不能则将其写入新管道 fd[1] 中。最后，关闭原始管道的读端 oldFd[0]，关闭新管道的写端 fd[1]，并等待子进程结束。

在 make qemu 后执行 primes 命令：

查看正确得分：

```
● ljk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (0.6s)
```

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
```

## [实验中遇到的问题和解决方法]

本次实验的内容是比较复杂的，一开始没有想到利用递归的方法去解决问题，如何将父进程和子进程的关系用代码的关系表示出来也是有点困难的。之后在教学视频资源的指导下，一步步理解了管道和递归关系的写法，完成了编写。还有 wait((int \*)0) 这行代码，父进程需要等待子进程结束，在本次代码的两处地方都有运用到，不要漏写。

## [实验心得]

在这次较难代码编写的过程中，我学到了如何有效地利用进程间通信和递归调用来解决复杂的计算问题。通过使用管道传递数据，并在子进程中递归处理剩余数据，我深入理解了并发编程的基本概念和技术。

## 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);
}

```

这段代码实现了一个递归的文件搜索函数 `find`，它接受两个参数：`path` 表示要搜索的路径，`filename` 表示我们需要查找的文件名。其中，`pfilename` 函数提取路径中的文件名部分。首先，代码会尝试打开指定的路径，如果打开失败，则会输出错误消息并退出程序。接下来，代码调用 `fstat` 函数获取打开文件或目录的信息，如果获取失败，则会输出错误消息并退出程序。我们这里使用 `switch` 语句，根据文件或目录的类型执行不同的操作：

如果是一个常规文件，将比较文件名与指定的文件名，如果匹配则输出该文件的路径。

如果是一个目录，代码会遍历目录中的每个目录项，跳过无效的条目，对于有效的条目，将构建新的路径并递归调用 `find` 函数进行继续搜索。

最后，关闭文件描述符，并返回 0 表示函数执行成功结束。

在 `make qemu` 后执行相应命令：

查看正确得分：

```
● ljk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (0.7s)
== Test find, recursive == find, recursive: OK (1.0s)
```

```
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
```

## [实验中遇到的问题和解决方法]

这道题还是比较复杂的，一开始没有头绪，之后看了指导的提示“Look at `user/ls.c` to see how to read directories.”发现可以在 `user/ls.c` 上寻找相应方法。而且该指导让我们注意 `Don't recurse into "." and ".."`，这里实验我们需要使用代码 `if(0 == strcmp(".", de.name) || 0 == strcmp.."., de.name)) continue;` 来正确地处理这两个特殊的目录名。

## [实验心得]

在这次实验中，在教学资料的帮助下，我实现了一个递归的文件搜索函数，旨在遍历目录树并查找特定文件。在实现递归文件搜索的过程中，我深入理解了递归的概念；同时通过使用 `open`、`fstat`、`read` 和 `stat` 等系统调用，我对这些调用的作用和使用方法有了更深入的了解。

## 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
    }
    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
    }
}
```

这个函数后面

```

        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，是将标准输入转为命令行参数。而我们这里使用的使用用例中的|的作用是管道命令，将左侧命令的标准输出转换为标准输入，提供给右侧命令作为参数。这段代码首先将程序本身的参数(即从命令行传入的参数)读取并存储在 args 数组中。接下来，进入一个无限循环，每次循环都会读取标准输入中的一行内容(以换行符\n 为结束符)，并将其保存到 buf 字符数组中。然后，将该行内容追加 args 数组中，并将 args 数组的最后一个元素设为 0，以便作为 exec 函数的参数结束标记。在子进程中，调用 exec 函数执行 args[1] 指定的命令，并将 args + 1 作为参数传递给该命令。如果 exec 执行成功，子进程会自动退出。如果 exec 执行失败，子进程会打印出"exec err"的错误信息。在父进程中，使用 wait 函数等待子进程执行完毕。

在 make qemu 后执行相应命令：

查看正确得分：

```
● ljk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (0.8s)
```

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ echo hello too | xargs echo bye
bye hello too
```

## [实验中遇到的问题和解决方法]

我觉得这次实验的重点首先是要了解 xargs 命令要让我们干什么，我上网查找了资料，了解到该指令的作用；其次，我觉得在代码中的 read 函数的要会用，read(0, buf + i, sizeof(char)): 0 是标准输入的文件描述符;buf + i 表示从 buf 数组的第 i 个位置开始读取数据;sizeof(char)是每次读取的字节数，通常是 1 字节。

## [实验心得]

该实验相对于前面两个实验简单一点，我主要收获到了如何处理标准输入并将其作为参数传递给后续的命令(管道)，同时更加了解了一些库函数的调用，让我更加熟悉了标准输入输出的处理方法和进程的创建以及执行。

## Lab1 的实验结果截图：./grade-lab-util

```
● ljk@LJK 'sPC:~/xv6-labs-2020$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.2s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.1s)
== Test primes == primes: OK (1.0s)
== Test find, in current directory == find, in current directory: OK (1.0s)
== Test find, recursive == find, recursive: OK (1.1s)
== Test xargs == xargs: OK (1.0s)
== Test time ==
time: OK
Score: 100/100
```

## Lab2:System Calls

首先，在开始 Lab2 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 syscall

make clean

```
ljk@LJK 'sPC:~/xv6-labs-2020$ cd xv6-labs-2020/
ljk@LJK 'sPC:~/xv6-labs-2020$ git fetch
ljk@LJK 'sPC:~/xv6-labs-2020$ git checkout syscall
Switched to branch 'syscall'
Your branch is up to date with 'origin/syscall'.
ljk@LJK 'sPC:~/xv6-labs-2020$ code .
ljk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
/*/*.o /*/*.d /*/*.asm /*/*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_ine
ljk@LJK 'sPC:~/xv6-labs-2020$
```

### 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( $1 << \text{SYS\_fork}$ ), 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( $1 \ll \text{SYS\_fork}$ )，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 中添加右侧代码： `$U/_trace\`

运行 make qemu，我们看到编译器无法编译 user/trace.c，因为系统调用的用户空间存根还不存在：所以我们需要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。

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；

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：

全部执行后会有以下语句：

ALL TESTS PASSED

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
9: syscall fork -> 11
9: syscall fork -> 12
10: syscall fork -> 13
9: syscall fork -> 14
10: syscall fork -> 15
```

## [实验中遇到的问题和解决方法]

这次实验是第一次完整地实现一次系统调用，主要是在指导的参考下做好每一步即可，注意不要少加了写上面都行就可以，总体的难度来说不大。

## [实验心得]

在这个实验中，我从代码上接触了系统调用的实现流程，深入学习了系统调用追踪的概念和实现方法。通过使用系统调用追踪技术，我能够监控程序与操作系统内核之间的交互，获取详细的系统调用信息。

## 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 中添加右侧代码：`$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 sysinf，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|
```

之后我们便需要实现 sys\_info 函数的具体功能。需要调用首先获取用户传递的地址。然后，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。最后，将这些信息从内核空间复制到用户空间。如果任何步骤失败，返回 -1 表示错误；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;
}
```

即 Makefile 的 UPROGS 中添加 \$U/\_sysinfotest 当运行 make qemu 时， user/sysinfotest.c 将会编译失败，遵循和上一个作业一样的步骤添加 sysinfo 系统调用。要在 user/user.h 中声明 sysinfo() 的原型，需要预先声明 struct sysinfo 的存在 ‘sysinfo 需要将一个 struct sysinfo 复制回用户空间；要获取空闲内存量，在 kernel/kalloc.c 中添加一个函数要获取进程数，在 kernel/proc.c 中添加一个函数。

在 make qemu 后执行相应命令如图：

```
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]

有了前一个实验的基础铺垫，这个实验的流程我们大致熟悉了。主要还是 sys\_info 函数具体功能的实现。在编写代码的过程中，主要就是三个函数的实现。

freememory(&info.freemem): 调用 freememory 函数获取系统中剩余的空闲内存，并将其存储在 info.freemem 中； procnum(&info.nproc): 调用 procnum 函数获取系统中当前正在运行的进程数，并将其存储在 info.nproc 中； copyout(myproc()->pagetable, addr, (char \*)&info, sizeof info): myproc()->pagetable: 获取当前进程的页表。addr: 用户空间的目的地址。 (char \*)&info: 要复制的数据，强制类型转换为 char\* 类型。 sizeof info: 要复制的数据大小。 copyout 函数将数据从内核空间复制到用户空间。如果复制失败，返回 -1。通过查阅资料和参考 sysfile.c 和 file.c 的代码，逐步读懂这三个重要函数的实现就问题不大。

## [实验心得]

在这次实验中，我学习了如何在 xv6 操作系统中添加一个新的系统调用。首先我们需要在 user/user.h 中加入系统调用在用户态的入口，其次在同级目录下的 usys.pl 中加入在用户态执行的汇编代码，然后在 kernel/syscall.h 中为系统调用分配一个系统调用的编号，最后深入内核代码，实现系统调用具体的逻辑，包括设计数据结构及其依赖的函数等。通过这个实验我对 xv6 的内核进行了深入的剖析，对于系统调用的理解也更加透彻。

## Lab2 的实验结果截图: ./grade-lab-syscall

```
● 1jk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.0s)
== Test trace all grep == trace all grep: OK (0.9s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (7.9s)
== Test sysinfotest == sysinfotest: OK (1.0s)
== Test time ==
time: OK
Score: 35/35
○ 1jk@LJK'sPC:~/xv6-labs-2020$ 
```

## 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

```
1jk@LJK 'sPC:~/xv6-labs-2020$ git fetch
1jk@LJK 'sPC:~/xv6-labs-2020$ git checkout pgtbl
Branch 'pgtbl' set up to track remote branch 'pgtbl' from 'origin'.
Switched to a new branch 'pgtbl'
1jk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill \
user/_stats
1jk@LJK 'sPC:~/xv6-labs-2020$ 
```

\*在这里，我遇到了一个卡了很久的问题。我想花一定篇幅记录一下。我在切换完 pgtbl 分支后，打开相应的 vscode 代码，发现无论我怎么再次切换，目录项中的测试成绩的文件始终是 grade-lab-util，很奇怪，其他的实验打开都是每个实验相应测试程序，唯有这个我试了好机会几回都是这样。大概花了一个小时时间来找问题，最后我在官网的代码上重新下载放入文件才解决问题。在解决这个问题的过程中，我学到了很多关于 git 的代码，收获了挺多实用的命令。

## 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);
}

```

这段代码主要由两个函数组成，`vmprint` 函数是页表打印的入口点。它首先打印页表的起始地址，然后调用 `printwalk` 函数开始递归打印页表内容；`printwalk` 函数递归地遍历并打印给定的页表及其所有下级页表的内容，最主要的是遍历的思想。函数先遍历页表中的每一个页表项，如果页表项有效(`pte & PTE_V`)，则打印当前深度的缩进，将页表项转换为物理地址并打印；如果该页表项不是指向一个物理页(即没有读、写、执行权限)，则递归打印下一级页表。

运行 `make qemu` 指令便会得到以下结果：

```

xv6 kernel is booting

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

```

## [实验中遇到的问题和解决方法]

这个小实验是第三个 Lab 中算简单的一个了，在学习操作系统理论知识的基础上，我们便会挺容易地想到使用递归的方法来实现打印。这个小实验在编写和理解的方面没有太大的问题。(但是下面两个实验就开始难度剧增了)

## [实验心得]

通过编写和调试页表遍历代码，加深了对页表层次结构的理解。这有助于理解操作系统如何通过页表管理内存。同时深入了解页表项中的各种标志位(如 PTE\_V, PTE\_R, PTE\_W, PTE\_X)及其用途。在编写代码方面，我掌握了如何设计递归函数来遍历复杂的数据结构。特别是在处理多级嵌套的结构时，递归是一种非常自然且有效的方式。

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

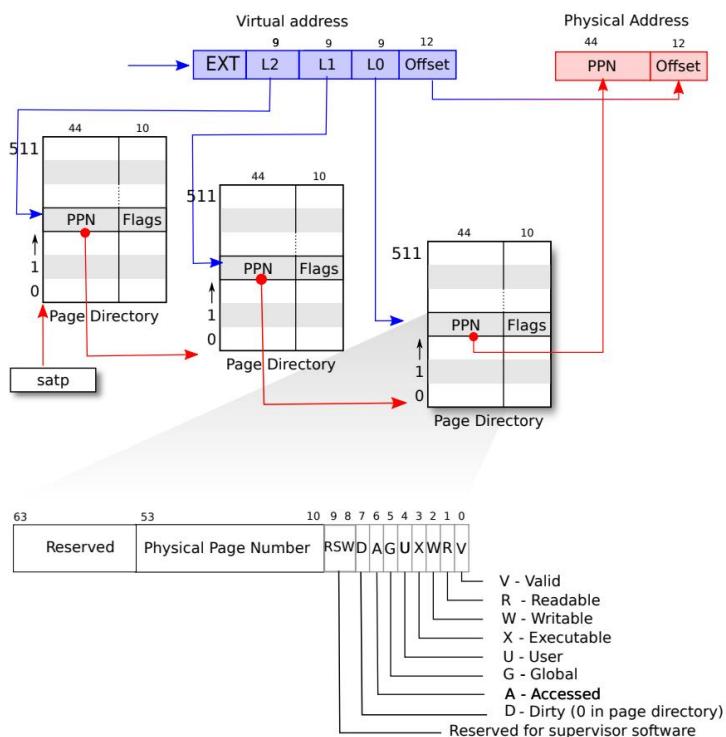


Figure 3.2: RISC-V address translation details.

## 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;
```

之后修改 kvminit 函数。原先的函数逻辑是将全局变量 kernel\_pagetable 初始化并 map。这里将这个功能抽出来，先写一个 kvm\_init\_one，表示初始化一个内核页表，再在 kvminit 里调用此函数，赋值给 kernel\_pagetable。

```
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));
kvmmmap_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：

```

        sepc=0x00000000000201a stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x00000000000000d pid=6265
        sepc=0x000000000003e7a stval=0x0000000000012000
OK
test sbrkarg: OK
test validate: OK
test stacktest: usertrap(): unexpected scause 0x00000000000000d pid=6269
        sepc=0x000000000002188 stval=0x000000000000fb0c0
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test input: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdat: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED

```

显示 ALL TESTS PASSED 即可完成内核页表的修改流程。

## [实验中遇到的问题和解决方法]

此次实验的难度极大，主要是许多函数的修改和增加。本来想要自己看完理解一下学习编写的，但是发现许多要点的添加都没有写进去。编写函数体的过程是比较简单的，主要是将 kvm\_init\_one() 函数和其余函数联系起来，而且函数位于的位置也需要我们理解 xv6 工作的大致流程。最后，我在指导视频的帮助下完成了此次作业的实现，还需要课后更加深入地理解和记忆。

## [实验心得]

通过这次艰难的实验，我初步了解了操作系统中的页表管理机制，并掌握了实现和调试复杂内核代码的技能。这个实验不仅增强了对操作系统理论的理解，

还提升了实际编写和调试内核代码的能力。这些经验和技能将在未来的系统编程和操作系统开发中起到重要作用。这次实验我解决问题的方式主要是参考和不断实验，模仿一些简单的系统自带函数编写较难的函数。

## 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]

这个实验也挺难的，我们首先来了解实验是要让我们干什么：

xv6 为了方便让已进入内核的进程获取用户态的地址空间，设计了一种将用户态页表塞到内核页表中的机制。如此一来，当已进入内核的进程想要查询进程用户态的某一地址空间时，无需再切回用户态，直接在内核中就能实现查询和地址翻译工作，这样大大节省了 user 和 kernel 之间来来回回切换的代价。

首先在 kernel/defs.h 中声明函数。

```
void uvm2kvm(pagetable_t u, pagetable_t k, uint64 from, uint64 to);
```

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) vmpprint(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;
}

```

这样本实验就成功地完成了！

## [实验中遇到的问题和解决方法]

此次实验的难度也挺大，主要是两个函数的替换和其他依赖的修改。本次实验我依旧在学习中完成，我在指导视频的帮助下完成了此次作业的实现，还需要课后更加深入地理解和记忆。还有一个问题就是使用./grade-lab-pgtbl 测试分数的时候，一直报错 Test answers-pgtbl.txt FAIL: Cannot read answers-pgtbl.txt，在查询相关资料和参考文献，需要补充 answers-pgtbl.txt 文件并且加上文字文本，这样就可以成功通过测试了。

## [实验心得]

这个实验发过程也较为复杂，主要是两个函数(用于从用户空间复制数据到内核空间)的实现。copyin 函数用于从用户空间复制数据到内核空间；copyinstr 函数用于从用户空间复制一个以 null 结尾的字符串到内核空间。我通过简化和优化 copyin 和 copyinstr 函数，加深了对内核内存管理和页表机制的理解，同时一定程度上地积累了编程经验。

## Lab3 的实验结果截图：./grade-lab-pgtbl

```

● 1jk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-pgtbl
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (1.3s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin == count copyin: OK (0.9s)
== Test usertests ==
(83.9s)
== 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

```
ljk@LJK 'sPC:~/xv6-labs-2020$ git fetch
ljk@LJK 'sPC:~/xv6-labs-2020$ git checkout traps
Branch 'traps' set up to track remote branch 'traps' from 'origin'.
Switched to a new branch 'traps'
ljk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill \
e user/_call user/_bittest
ljk@LJK 'sPC:~/xv6-labs-2020$
```

## 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 xv 6 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：

```
● ljk@LJK 'sPC:~/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：

<pre>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</pre>	<pre>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</pre>
---	--

```

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: 7b050513      addi  a0,a0,1968 # 7d8 <malloc+0xea>
    30: 00000097      auipc ra,0x0
    34: 600080e7      jalr  1536(ra) # 630 <printf>
    exit(0);
    38: 4501          li    a0,0
    3a: 00000097      auipc ra,0x0
    3e: 27e080e7      jalr  638(ra) # 2b8 <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？

参数使用寄存器 a0 到 a7 传递，分别对应第 1 到第 8 个参数。24: “4635 li a2,13”：表示将立即数 13 加载到寄存器 a2，寄存器 a2 保存 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，而函数 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>
```

即跳转到 ra+1536 的位置，此时 ra 的值即为 pc 的值，也即 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 指令的功能，在刚跳转后 ra 的值为  $pc + 4 = 0x34 + 4 = 0x38$ 。

(5) Run the following code.

```

unsigned int i = 0x00646c72;
printf("H%0x Wo%0s", 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%lx 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 转换为十六进制是 e110。&i 所指向的字符串是 "rld"。输出是 He110 World; 需要将 i 设置为 0x726c6400; 57616 不需要修改。

%x 表示以十六进制数形式输出整数, 57616 的 16 进制表示就是 e110, 与大小端序无关。%s 是输出字符串, 以整数 i 所在的开始地址, 按照字符的格式读取字符, 直到读取到 '\0' 为止。当是小端序表示的时候, 内存中存放的数是: 72 6c 64 00, 刚好对应 rld。当是大端序的时候, 则反过来了, 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]

本次实验的难点是汇编代码的阅读和理解, 通过查找资料和学习基础语法基础, 便可以解决这个实验中遇到的问题。

## [实验心得]

通过这次实验，我对 RISC-V 的汇编语言有了一定的理解，特别是函数调用和参数传递的过程。我了解到函数参数是通过寄存器传递的，这是因为寄存器的访问速度比内存快得多，所以在函数调用中使用寄存器来传递参数可以提高程序的运行速度。

## 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，那样我们就能在 sys\_sleep 中引用 backtrace，在 kernel/defs.h 添加如下代码：

```
void backtrace(void);
```

我们是在 printf.c 文件中实现该函数的，所以我们添加的原型应该放在 //printf.c 这段注释下实现。同时根据提示，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;
}
```

我们需要在 kernel/sysproc.c 中的 sys\_sleep() 函数中使用 backtrace() 函数，所以我们要在 kernel/sysproc.c 中加入使用这段函数的代码：

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

    backtrace();
    if (argint(0, &n) < 0)
        return -1;
    acquire(&tickslock);
```

接下来我们就可以在在 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 (PGROUNDOWN(fp) < PGROUNDDUP(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
    }
}
```

我们要实现的 backtrace 函数的作用是遍历并打印当前程序的调用栈(即函数调用的历史记录)，帮助调试程序。首先我们需要获取当前的帧指针 fp，循环条件是确保 fp 在同一页面内，以避免跨页面访问引发的错误，然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，即可成功输出实验结果；按照指导提示完成命令的检测，输出的图片如图，测试成功：

```
xv6 kernel is booting

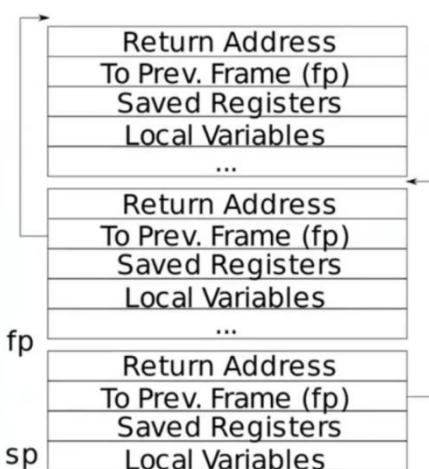
hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002ce4
0x0000000080002bbe
0x00000000800028a8
$ QEMU: Terminated
● ljk@LJK'sPC:~/xv6-labs-2020$ addr2line -e kernel/kernel
0x0000000080002ce4
/home/ljk/xv6-labs-2020/kernel/sysproc.c:62
0x0000000080002bbe
/home/ljk/xv6-labs-2020/kernel/syscall.c:140
0x00000000800028a8
/home/ljk/xv6-labs-2020/kernel/trap.c:76
```

## [实验中遇到的问题和解决方法]

我觉得这次实验的重点是理解代码\*(uint64 \*)(fp - 8)和代码\*(uint64 \*)(fp - 16)的意思，通过查阅资料和相关参考文献，我才知道 fp - 8 用于获取当前栈帧的返回地址，fp - 16 用于获取上一个栈帧的帧指针。即该地址-8 即是返回地址(打印内容)的地址，-16 就是上一个 fp 的地址。

## [实验心得]

本次实验让我深入理解了函数调用栈和栈帧结构的原理。在实现 backtrace 函数的过程中，我学会了如何通过帧指针(fp)回溯函数调用的历史。特别是在解析 fp - 8 和 fp - 16 的意义时，我明白了返回地址和前一个栈帧指针在内存中的存储位置。如右图，返回地址位于栈帧帧指针的固定偏移(-8)位置，并且保存的帧指针位于帧指针的固定偏移(-16)位置。这次实验不仅强化了我对计算机系统底层工作的认识还提高了我在调试复杂程序时的能力。



## 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，这些操作类似于 Lab2。

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

extern uint64 sys_sigalarm(void);      #[SYS_sigalarm] sys_sigalarm,
extern uint64 sys_sigreturn(void);     #[SYS_sigreturn] sys_sigreturn,
```

在 kernel/proc.h 中增加一些有关 alarm 的属性，同时在在 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
        return -1;
    myproc()->alarm_interval = interval; // 设置当前进程的闹钟信
    myproc()->alarm_handler = handler;   // 设置当前进程的闹钟信
    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();
}

```

当指定设备中断发生时(which\_dev == 2), 如果当前进程设置了闹钟信号时间间隔(p->alarm\_interval 非零), 则会递增计数器 p->alarm\_ticks。当计数器值达到设定的时间间隔(p->alarm\_ticks == p->alarm\_interval)时, 系统保存当前进程的寄存器上下文(trapframe)到 alarm\_trapframe 中, 并将程序计数器(epc)设置为闹钟信号处理程序的地址(p->alarm\_handler), 以便在返回用户态时执行闹钟信号处理程序。最后, 调用 yield() 函数将当前进程放回调度队列, 让操作系统调度其他进程执行。

成功运行 alarmtest:

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
.....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 rmdat: 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]

这次实验在总体上是不算太难的，因为实现代码的框架和 Lab2 中的那几个实验类似，向 XV6 添加一个特性。但是在具体一些函数代码的编写是比较困难的，需要运用到好多库函数来实现，这对于之前没有接触过 XV6 操作系统代码的我们来说有些困难。这次实验我完成的主要方法是学习和参考，在理解每一行代码实现的意义的同时解决实验问题。

## [实验心得]

这次实验让我深入理解了操作系统中闹钟信号机制的实现和运作原理。通过编写和调试 sys\_sigalarm 和 sys\_sigreturn 系统调用，以及相关的中断处理代码，我学会了如何通过设置时间间隔和信号处理程序来管理进程的定时任务。整个过程不仅提升了我对系统调用和中断处理的理解，还加强了我在编写底层代码时的调试能力。

## Lab4 的实验结果截图：./grade-lab-traps

```
● ljk@LJK'sPC:~/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 == (3.9s)
== Test alarmtest: test0 ==
    alarmtest: test0: OK
== Test alarmtest: test1 ==
    alarmtest: test1: OK
== Test alarmtest: test2 ==
    alarmtest: test2: OK
== Test usertests == usertests: OK (125.3s)
== 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
```

```
ljk@LJK ~$ git fetch
ljk@LJK ~$ git checkout lazy
Branch 'lazy' set up to track remote branch 'lazy' from 'origin'.
Switched to a new branch 'lazy'
ljk@LJK ~$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
/*/*.o /*/*.d /*/*.asm /*/*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_ki
e user/_lazytests
```

## 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;
}
```

如上图, 删去原本调用的 `growproc()` 函数, 增加 `myproc()->sz`; 使用 addr 暂存 myproc()->sz 旧的值, 同时 myproc()->sz 增加 n, 然后返回旧的大小(即 addr 的值)。在 make qemu 后执行 echo hi 命令, 得到结果符合预期:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
        sepc=0x0000000000012ac stval=0x000000000004008
panic: uvmunmap: not mapped
```

## [实验中遇到的问题和解决方法]

这个实验算是很简单的一个了, 按照实验指导的提示, 找到相应的函数进行修改即可。整个实验过程没有遇到什么问题。

## [实验心得]

在本次实验中, 我修改了 xv6 的 sys\_sbrk() 系统调用, 使其不再分配实际内存页面, 仅增加进程的大小。这一修改帮助我深入理解了操作系统内存管理的原理, 特别是内存分配与进程大小调整的关系。

## **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 hints, r\_scause() == 13 or 15 表示一个 page fault, 同时 r\_stval 返回导致 page fault 的虚拟地址。系统调用的中断码是 8, page fault 的中断码是 13 和 15。因此，这里我们增加对 r\_scause() 中断原因进行判断，如果是 13 或是 15，则说明没有找到地址。错误的虚拟地址被保存在了 STVAL 寄存器中，我们中断判断时，如果出错(虚拟地址不合法或者没有成功映射到物理地址)，也要终止进程。

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，因为`uvmunmap`是用来释放内存调用的，释放内存时，页表内有些地址并没有实际分配内存，因此没有进行映射。如果在`uvmunmap`中发现了没有映射的地址，不需要`panic`。因此我们需要修改`kernal/vm.c`的`uvmunmap()`函数，注释掉那两行`panic`，并`continue`。

```

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，测试成功。

The screenshot shows the xv6 kernel booting process. It starts with "xv6 kernel is booting", followed by "hart 2 starting" and "hart 1 starting". Then it reaches the "init" stage, with "starting sh". Finally, the command "\$ echo hi" is entered at the prompt, and "hi" is displayed as the output.

## [实验中遇到的问题和解决方法]

`kalloc()` 函数用于分配新的物理页面，如果它返回 0，说明没有足够的物理内存。解决方法：确保在实际应用中物理内存没有被过度使用，并且在开发和测试环境中留出足够的内存用于页面分配。如果遇到 `kalloc()` 失败，代码会将进程标记为 `killed`，这意味着该进程会在后续处理过程中被终止。

## [实验心得]

本次实验让我深入了解了操作系统中的页面错误处理机制，并实际实现了页面分配和地址映射的基本逻辑。在实验过程中，我遇到了多种常见问题，例如物理内存分配失败和地址映射失败。通过仔细检查函数的返回值和实施适当的错误

处理，我成功解决了这些问题，确保了系统的稳定性和可靠性。此外，通过在不同条件下测试代码的行为，我加深了对内存管理的理解，特别是在处理非法内存访问和中断处理方面。

## 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.

我们为您提供了一个名为 lazytests 的 xv6 用户程序，它测试一些可能会给您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]

不需要修改代码，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 即可。

```
xv6 kernel is booting

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

```
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdir: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
kalloc == 0
ALL TESTS PASSED
```

### [实验中遇到的问题和解决方法]

这个实验的完成是在前一个实验的基础上，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。

## Lab5 的实验结果截图: ./grade-lab-lazy

```
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

```
ljk@LJK 'sPC:~/xv6-labs-2020$ git fetch
ljk@LJK 'sPC:~/xv6-labs-2020$ git checkout cow
Switched to branch 'cow'
Your branch is up to date with 'origin/cow'.
ljk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_in
e user/_cowtest
```

### 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]

根据实验指导提示，我们首先需要修改 uvmcopy()将父进程的物理页映射到子进程，而不是分配新页。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;
}

```

copy-on-write (COW) fork() 的目标是推迟到子进程实际需要物理内存拷贝时再进行分配和复制物理内存页面。这段代码的目的就是将父进程的物理页映射到子进程，而不是分配新页。

之后修改 usertrap() 以识别页面错误。当 COW 页面出现页面错误时，使用 kalloc() 分配一个新页面，并将旧页面复制到新页面，然后将新页面添加到 PTE 中并设置 PTE\_W。相应的代码如下：

```

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
}

```

以上代码实现了在写页错误发生时的异常处理机制，包括页表项的操作、物理内存的分配和释放，以及异常情况下的错误处理和设备中断处理。

接下来就要修改 kalloc.c 文件了，当 kalloc() 分配页时，将页的引用计数设置为 1。当 fork 导致子进程共享页面时，增加页的引用计数；每当任何进程从其页表中删除页面时，减少页的引用计数。kfree() 只应在引用计数为零时将页面放回空闲列表。可以将这些计数保存在一个固定大小的整型数组中。你必须制定一个如何索引数组以及如何选择数组大小的方案。例如，您可以用页的物理地址除以 4096 对数组进行索引，并为数组提供等同于 kalloc.c 中 kinit() 在空闲列表中放置的所有页面的最高物理地址的元素数。

```
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，包括取消写保护位并重新映射页面，以及处理页表项和物理页面的操作。如果页表项 pte 的写保护位未设置且保留位设置(\*pte & PTE\_RSW)，则执行以下操作：获取物理页面引用锁。如果物理页面的引用计数为 1，表示只有当前进程使用该页面，则将页表项取消保留位 PTE\_RSW 并设置写权限 PTE\_W。如果物理页面有多个进程共享，则分配新的物理内存 mem

并将原物理页面数据复制到新内存中。更新新页表项的标志并将新内存映射到页表，如果操作失败则释放新内存和引用锁，返回-1。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 1 starting
hart 2 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
```

```
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 rmdat: 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]

此次实验的难度大，代码量也大，需要我们对 Copy-on-Write 有着良好的理解。所以我首先上网查询资料了解了 Copy-on-Write 的主要原理，明白了这种技术是在解决什么问题。我完成本次实验的基本方法是参考和学习，最后在指导视频和参考文献的帮助下完成了实验的任务。

## [实验心得]

copy-on-write (COW) fork() 的目标是推迟到子进程实际需要物理内存拷贝时再进行分配和复制物理内存页面。当父进程调用 fork() 时，操作系统并不立即复制父进程的地址空间。相反，它将子进程的地址空间设置为与父进程完全相同的映像。如果子进程尝试修改共享页面的内容，则操作系统才会执行实际的页面复制。这样避免了不必要的内存复制，有效地提升了操作系统的性能和资源利用率。通过这次实验，我更加深入地理解了操作系统如何处理进程间内存共享和复制，加深了我对操作系统原理的理解。

## Lab6 的实验结果截图：

### /grade-lab-cow

```
● 1jk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-cow
make: 'kernel/kernel' is up to date.
== Test running cowtest == (3.7s)
== Test simple ==
  simple: OK
== Test three ==
  three: OK
== Test file ==
  file: OK
== Test usertests == (72.6s)
== 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
```

```
ljk@LJK 'sPC:~/xv6-labs-2020$ git fetch
ljk@LJK 'sPC:~/xv6-labs-2020$ git checkout thread
Branch 'thread' set up to track remote branch 'thread' from 'origin'.
Switched to a new branch 'thread'
ljk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
   /*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
    user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill \
e user/_uthread
```

## 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]

首先我们需要首先在 user/uthread.c 定义一个结构体，保存寄存器内容：，这样每个线程都有自己的上下文，可以在切换线程时保存和恢复。

```
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;
};
```

并且注意将这个结构体并加入到下文的线结构体中。

在进程切换时，只需要交换调用方保存的进程上下文。`ra` 寄存器代表的是返回地址(return address),`sp` 寄存器代表的是栈底指针(stack pointer)，这两个寄存器并不需要在切换进程时交换，而是由系统更新。修改 `user/uthread_switch.S`，实现 `thread_switch`:

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

修改 `thread_create()` 函数，使其能在线程数组中遍历，发现一个未初始化的线程就执行初始化。

```
// YOUR CODE HERE  
t->context.ra = (uint64)func;
```

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

修改 `thread_schedule()` 函数，它负责负责用户多线程间的调度。它从当前线程的位置开始，在线程数组中寻找一个 ‘RUNNABLE’ 状态的线程进行运行，在找到线程后调用函数 `‘thread_switch()’` 进行线程的切换。

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

## [实验中遇到的问题和解决方法]

这次实验主要是要让我们理解多线程编程，为用户级线程系统设计上下文切换机制。主要是要掌握进程切换时的操作系统要做的一些具体的事件函数。我在课外找到了一篇详细解释进程切换的视频，在其指导下完成了此次实验。

## [实验心得]

在进行这个实验的过程中，我对多线程编程以及用户级线程系统的设计有了深入的理解，特别是如何设计线程的上下文切换机制。

`kernel/trap.c` 中 `usertrap` 函数是用于将用户空间切换到内核空间，会保存用户空间 32 个寄存器然后切换至内核空间中，在其中有如下代码段：用于查看是否是由于时钟中断导致的，如果是则调用 `yield()` 函数处理，将 CPU 资源让出来。

```
// give up the CPU if this is a timer interrupt.  
if(which_dev == 2)  
    yield();
```

而 `yield()` 函数具体做的操作如下：

```
// Give up the CPU for one scheduling round.  
void  
yield(void)
```

```

{
    struct proc *p = myproc();
    acquire(&p->lock);
    p->state = RUNNABLE;
    sched();
    release(&p->lock);
}

```

kernel/proc.c 中有 sched 函数的实现过程，在判断是否符合上下文切换的条件后处理中断，后调用 swtch 函数进行上下文切换。

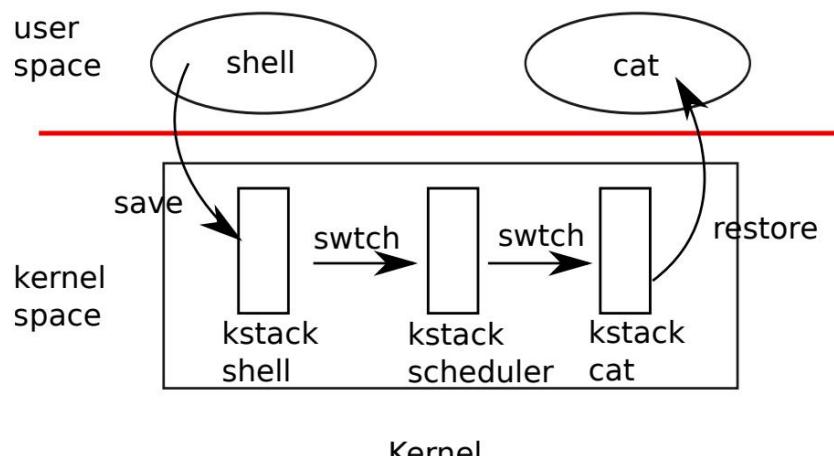
```

sched(void)
{
    int intena;
    struct proc *p = myproc();

    if(!holding(&p->lock))
        panic("sched p->lock");
    if(mycpu()->noff != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(intr_get())
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, &mycpu()->context);
    mycpu()->intena = intena;
}

```

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：



Kernel

## 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]

根据实验指导的提示，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);  
}
```

线程的安全问题是桶中的链表进行操作而导致的，需要在链表操作的前后加锁，所以我们需要在`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：

```
● ljk@LJK'sPC:~/xv6-labs-2020$ make ph  
    gcc -o ph -g -O2 notxv6/ph.c -pthread  
● ljk@LJK'sPC:~/xv6-labs-2020$ ./ph 1  
100000 puts, 6.085 seconds, 16435 puts/second  
0: 0 keys missing  
100000 gets, 6.079 seconds, 16450 gets/second  
● ljk@LJK'sPC:~/xv6-labs-2020$ ./ph 2  
100000 puts, 2.690 seconds, 37171 puts/second  
0: 0 keys missing  
1: 0 keys missing  
200000 gets, 5.792 seconds, 34532 gets/second
```

## [实验中遇到的问题和解决方法]

当不同线程对同一个 bucket 进行 put 操作时，可能会覆盖前一个 put 的结果，造成错误。当整个哈希表被加锁后整体的性能甚至不如之前，原因是每一时刻只能有一个线程进行操作，相当于单线程了，优化思路是降低锁的粒度，从对于整个哈希表加锁到对于每个 bucket 加锁。

## [实验心得]

这个实验还是比较简单的，有了本学期理论课进程调度电梯代码的编写，这个实验还是比较好理解一点的。这个实验提高了我在实际项目中应用并行编程的能力，理解了线程的创建、同步、数据传递和互斥锁的使用。

## 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]

我们在这个实验中需要实现阻塞线程，直到所有线程都到达屏障点，然后才允许所有线程继续执行。代码如下：

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

首先我们需要锁住互斥锁，使得唯一一个进程进入临界区，这时加已到达屏障的线程数增加一。之后检查是否所有线程都到达屏障点，若所有线程都到达屏障点，重置线程数，唤醒所有等待的线程；若不是，则继续等待。最后解除互斥锁，离开临界区。

输入如图所示代码测试  
实验结果成功通过：

```
● ljk@LJK'sPC:~/xv6-labs-2020$ make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
● ljk@LJK'sPC:~/xv6-labs-2020$ ./barrier 2
OK; passed
```

## [实验中遇到的问题和解决方法]

这段代码的实现还是比较简单的，有点向我们操作系统理论课程上要求书写的进程代码。这次实验只要是要理解它需要我们实现的功能是什么，理解好阻塞线程概念之后，按照正常的逻辑顺序和指导提示的逻辑顺序书写就可以顺利完成此次实验。

## [实验心得]

在本次关于线程屏障的实验中，我深入理解并实践了线程同步的基本原理。通过实现一个简单的屏障函数，掌握了如何使用互斥锁和条件变量来协调多个线程的执行。这一过程让我认识到，在多线程编程中，确保数据一致性和线程安全性的重要性。特别是在共享资源的访问和修改时，通过互斥锁来保护临界区是非常必要的。此外，条件变量的使用让我学会了如何让线程在合适的时机等待和唤醒，从而提高程序的执行效率和可靠性。

## Lab7 的实验结果截图：./grade-lab-thread

```
● ljk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.3s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (8.3s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (20.9s)
== 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
ljk@LJK 'sPC:~/xv6-labs-2020$ git fetch
ljk@LJK 'sPC:~/xv6-labs-2020$ git checkout lock
Branch 'lock' set up to track remote branch 'lock' from 'origin'.
Switched to a new branch 'lock'
ljk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
/*/*.o /*/*.d /*/*.asm /*/*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_k
e user/_stats user/_kalloc test user/_bcachetest
```

# 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 kalloc test 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 kalloc tests 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 kalloc test 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() 只有一个空闲列表，由一个锁进行保护，当多个 cpu 需要分配内存时，抢占该锁的次数将会大大增加。所以我们需要为每个 cpu 都创建一个空闲列表，并且分配一个锁。

首先在 kernel/kalloc.c 中将 kmem 修改为数组，为每个 CPU 分配一个 kmem。

```
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，为当前 CPU 的'freelist'分配所有的空闲内存空间。

```
for (int i = 0; i < NCPU; i++)
{
    sprintf(lockname, sizeof(lockname), "kmem%d", i);
    initlock(&kmem[i].lock, lockname);
}
```

修改 kfree 对这些锁进行释放, 适应新的数据结构, 使用 push\_off() 和 pop\_off() 来关闭和打开中断, 以便 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
```

最后修改 kalloc 函数, 用于查找当前 CPU 中是否有空闲块, 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、usertests sbrkmuch 和 usertests:

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ kalloc test
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache: #fetch-and-add 0 #acquire() 11
lock: bcache: #fetch-and-add 0 #acquire() 2
lock: bcache: #fetch-and-add 0 #acquire() 11
lock: bcache: #fetch-and-add 0 #acquire() 11
lock: bcache: #fetch-and-add 0 #acquire() 14
lock: bcache: #fetch-and-add 0 #acquire() 12
lock: bcache: #fetch-and-add 0 #acquire() 19
lock: bcache: #fetch-and-add 0 #acquire() 20
lock: bcache: #fetch-and-add 0 #acquire() 284
lock: bcache: #fetch-and-add 0 #acquire() 11
--- top 5 contended locks:
lock: proc: #fetch-and-add 23123 #acquire() 120209
lock: proc: #fetch-and-add 4073 #acquire() 120236
lock: virtio_disk: #fetch-and-add 4012 #acquire() 75
lock: proc: #fetch-and-add 3732 #acquire() 120214
lock: proc: #fetch-and-add 3272 #acquire() 120194
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK

```

ALL TESTS PASSED:

```

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

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 rmdat: 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]

这次实验主要是为每一个 CPU 都创建一个空闲列表，并且分配一个锁，这个比较好实现，只需要将原先 kmem 的属性变为一个数组存储即可。本次实验还有一个需要思考的地方就是分配可以内存的方法。经过查阅相关资料，允许不同 CPU 之间共享内存池的方式可以很好解决这个问题，首先，获取当前 CPU 的 ID。尝试从当前 CPU 的空闲列表中获取空闲块，如果成功则直接返回。如果当前 CPU 的空闲列表为空，则遍历其他所有 CPU 的空闲列表，尝试借用其中的空闲块。如果找到了空闲块，则将其从其他 CPU 的空闲列表中移除，并加入到当前 CPU 的空闲列表中，然后返回。

## [实验心得]

一开始我对使用 `push_off()` 和 `pop_off()` 两个函数的作用不太理解，经过查阅资料，发现这两个函数用来临时关闭和恢复中断。中断可以处理系统运行时发生的异常和错误。`push_off()` 函数的作用是将中断禁用，即关闭中断。在多任务操作系统或者多线程环境中，关闭中断是为了保护关键代码段或者确保某些操作的原子性而必要的操作。`pop_off()` 函数的作用是恢复之前被禁用的中断状态，即重新启用中断。函数 `cpuid` 返回当前的核心编号，但只有在中断关闭时调用它并使用其结果才是安全的。所以我们在获取 CPU 的 ID 乾需要调用 `push_off()` 函数并且加锁，使用完毕后调用 `pop_off()` 函数并解锁。

## 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 文件系统的一部分，它的作用是保存磁盘的一部分数据，减少磁盘操作的时间消耗。但这也导致所有进程(包括在不同 CPU 上的进程)都会共享这个数据结构。如果我们仅使用一个锁 bcache.lock 来保证对它修改的原子性，将会产生大量的竞争，这可能导致性能下降。

所以我们首先定义 buckets 数量并添加到 bcache 中。同时实现一个简单的 hash:

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

这里我们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。

现在我们需要修改 kernel/bio.c 中的 bcache 结构体，加上每个桶的锁。

```

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;

```

接下来，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;
    }
}

```

下一步我们需要在 bget 和 brelse 这两个函数中，将所有 bcache.lock 改为 bcache.hashlock[hashcode]，把 bcache.head 改为 bcache.head[hashcode]，表示每个桶的锁和每个缓冲区的双向链表。

最后在 bget() 函数中修改，实现一个用于在缓存系统中找到一个可用的缓冲区的功能。如果找到了一个引用计数为 0 的缓冲区，则将其从原来的位置移到新的位置并返回该缓冲区。

```

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");
}

```

输入如图所示代码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 0 #acquire() 2135
lock: bcache: #fetch-and-add 0 #acquire() 4120
lock: bcache: #fetch-and-add 0 #acquire() 2266
lock: bcache: #fetch-and-add 0 #acquire() 4274
lock: bcache: #fetch-and-add 0 #acquire() 4323
lock: bcache: #fetch-and-add 0 #acquire() 6322
lock: bcache: #fetch-and-add 0 #acquire() 6606
lock: bcache: #fetch-and-add 0 #acquire() 6621
lock: bcache: #fetch-and-add 0 #acquire() 6938
lock: bcache: #fetch-and-add 0 #acquire() 6202
lock: bcache: #fetch-and-add 0 #acquire() 6199
lock: bcache: #fetch-and-add 0 #acquire() 4143
lock: bcache: #fetch-and-add 0 #acquire() 4144
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 53958 #acquire() 1155
lock: proc: #fetch-and-add 49152 #acquire() 76463
lock: proc: #fetch-and-add 11797 #acquire() 76138
lock: proc: #fetch-and-add 11454 #acquire() 76116
lock: proc: #fetch-and-add 10982 #acquire() 76139
tot= 0
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]

这次实验也是比较有挑战性的。首先，我们要想到如果只用一个锁保证对其修改的原子性的话，会造成很多冲突，所以我们需要根据数据块的 blocknumber 将其保存进一个哈希表，哈希表的每个 bucket 都有一个相应的锁来保护；其次，

双向链表代码的编写也是比较复杂的，尤其是当我们在遍历链表时，条件判断和迭代更新错误，可能导致无限循环或访问越界，这就需要我们好好复习数据结构的内容。本次实验我的主要完成方法是参考和学习，在教学指导视频的帮助下，我顺利地完成了这次实验的任务。

## [实验心得]

通过本次实验，我深入了解了缓存缓冲区管理的实现原理和细节。在实现过程中，我学会了如何使用锁机制保护共享资源，避免并发访问带来的数据一致性问题。同时，我掌握了双向链表的操作，包括插入和删除节点的具体步骤。这些技能对于理解操作系统中缓冲区管理和文件系统的设计非常重要。

## Lab8 的实验结果截图：./grade-lab-lock

```
● ljk@LJK 'sPC:~/xv6-labs-2020$ ./grade-lab-lock
make: 'kernel/kernel' is up to date.
== Test running kalloc test == (52.4s)
== Test kalloc test: test1 ==
    kalloc test: test1: OK
== Test kalloc test: test2 ==
    kalloc test: test2: OK
== Test kalloc test: sbrkmuch == kalloc test: sbrkmuch: OK (6.6s)
== Test running bcachetest == (7.6s)
== Test bcachetest: test0 ==
    bcachetest: test0: OK
== Test bcachetest: test1 ==
    bcachetest: test1: OK
== Test usertests == usertests: OK (113.2s)
== 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

```
ljk@LJK 'sPC:~/xv6-labs-2020$ git fetch
ljk@LJK 'sPC:~/xv6-labs-2020$ git checkout fs
Branch 'fs' set up to track remote branch 'fs' from 'origin'.
Switched to a new branch 'fs'
ljk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
 */*.o */*.d */*.asm */*.sym \
 user/initcode user/initcode.out kernel/kernel fs.img \
 mkfs/mkfs .gdbinit \
 user/usys.S \
 user/_cat user/_echo user/_forktest user/_grep user/_init user/_bigfile
```

## Large files (moderate)

### [实验目的]

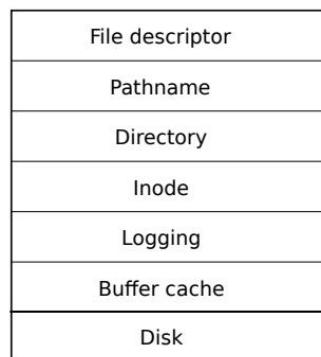
Modify bmap() so that it implements a doubly-indirect block, in addition to direct

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.

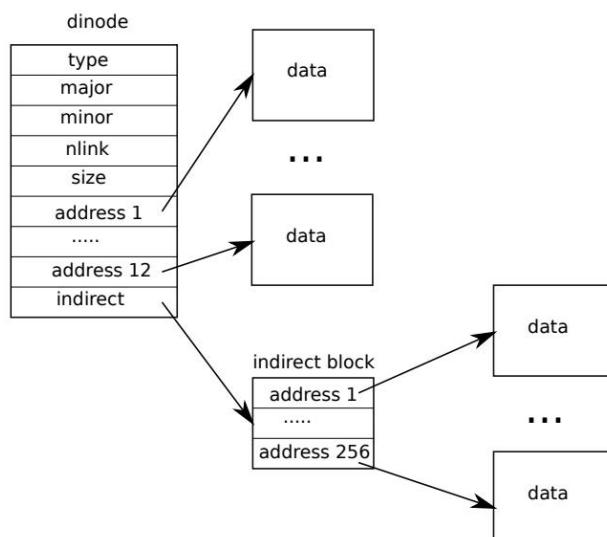
修改 bmap(), 以便除了直接块和一级间接块之外，它还实现二级间接块。你只需要有 11 个直接块，而不是 12 个，为你的新的二级间接块腾出空间；不允许更改磁盘 inode 的大小。ip->addrs[]的前 11 个元素应该是直接块；第 12 个应该是一个一级间接块(与当前的一样)；13 号应该是你的新二级间接块。当 bigfile 写入 65803 个块并成功运行 usertests 时，此练习完成。

## [实验步骤]

在 xv6 中，文件系统是层级结构的，总共有 7 个层级：



在这其中，每一个 Inode 的数据结构如图所示：



如图所示，Inode 支持了一级间接的块数据，Inode 本身储存 12 个块地址和一个间接索引地址，因此可以索引  $268(=12+256)$  个文件块。本实验的目的是将一个直接索引节点改为二级间接索引(它可以索引的文件块数为  $256*256$  个)，使一个文件的总大小可以达到  $65803(=256*256+256+11)$  个。

所以我们首先需要将 fs.h 中 NDIRECT 从 12 修改为 11，再定义二级页表，修改最大文件。NDINDIRECT 表示二级索引块号的总数，能够表示的块号个数是一级索引的平方  $256*256$ 。

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

由于 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)
```

接着根据指导，现在要添加一个二级索引，那么可以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 = malloc(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 = malloc(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 = malloc(ip->dev);
                    log_write(bp);
                }
                brelse(bp);
                return addr;
            }
            panic("bmap: out of range");
        }

```

检查块号是否在直接块范围内，如果块地址为 0，则分配一个新的块并记录到 inode，返回块地址；检查块号是否在一级间接块范围内，如果间接块地址为 0，则分配一个新的间接块，读取间接块到缓冲区，如果块地址为 0，则分配一个新的块并记录到间接块中，返回块地址；检查块号是否在二级间接块范围内，如果二级间接块地址为 0，则分配一个新的二级间接块，读取二级间接块到缓冲区，如果二级间接块的地址为 0，则分配一个新的一级间接块并记录到二级间接块中。读取一级间接块到缓冲区，如果块地址为 0，则分配一个新的块并记录到一级间接块中，返回块地址。

最后修改 itrunc，增加用于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);
}

```

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。遍历 inode 的直接块地址数组 addrs，如果地址不为 0，则释放该块并将地址置为 0；检查一级间接块地址是否存在，如果存在，读取间接块到缓冲区。遍历间接块中的地址数组，释放所有非零地址的块。释放间接块并将地址置为 0；检查二级间接块地址是否存在。如果存在，读取二级间接块到缓冲区。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。遍历一级间接块中的地址数组，释放所有非零地址的块，释放一级间接块并将地址置为 0，释放二级间接块并将地址置为 0。

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

The screenshot shows the xv6 kernel booting process and the results of a series of tests. On the left, the kernel prints "xv6 kernel is booting" and "init: starting sh". It then runs the command "\$ bigfile" which writes 65803 blocks. Finally, it prints "bigfile done; ok". On the right, a list of tests is shown, each followed by "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 rmdat: OK, test fourteen: OK, test bigfile: OK, test dirfile: OK, test iref: OK, test forktest: OK, test bigdir: OK. The final message is "ALL TESTS PASSED".

```
xv6 kernel is booting
init: starting sh
$ bigfile
wrote 65803 blocks
bigfile done; 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 rmdat: 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]

本次实验的理论知识点和我们操作系统理论课上教的差不多，所以感觉比较熟悉，主要是代码的编写。在调试的时候，我遇到过这个问题：

```
kernel/fs.c: In function 'bmap':
kernel/fs.c:412:12: error: 'NDINDIRECT' undeclared (first use in this function); did you mean 'NINDIRECT'?
  412 |     if (bn < NDINDIRECT)
                  ^~~~~~
                  NINDIRECT
```

显示的是'NDINDIRECT' undeclared (first use in this function)，表示这个变量还未定义。之后我才发现我还需要在 kernel/fs.h 中声明一下这些变量，这是一个简单的但是由很容易犯错的问题。

## [实验心得]

在本次实验中，我深入理解了文件系统的块管理机制，尤其是直接块、一级间接块和二级间接块的处理过程。通过编写和调试 bmap 和 itrunc 函数，我学会了如何在文件系统中实现从逻辑块号到物理地址的映射，并掌握了如何正确释放文件占用的磁盘块。这不仅让我巩固了操作系统理论课程上的知识，还增强了我对数据结构和操作系统底层原理的理解。

## 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]

在本次实验中我们需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
```

以上基本的属性添加后，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;
}

```

首先使用 `argstr` 函数获取系统调用参数。`argstr(0, target, MAXPATH)` 获取第一个参数，即目标路径 `target`，并将其存储在 `target` 数组中。`argstr(1, path, MAXPATH)` 获取第二个参数，即符号链接的路径 `path`，并将其存储在 `path` 数组中。调用 `begin_op` 函数开始一个文件系统操作。这通常用于确保文件系统的一致性，再调用 `create` 函数创建一个新的符号链接文件，调用 `writei` 函数将目标路径 `target` 写入到符号链接文件中，最后调用 `iunlockput(ip)` 解锁并释放 `inode`。调用 `end_op` 结束文件系统操作。

最后修改 `sys_open` 函数在创建符号链接时，如果目标路径是一个符号链接本身，我们需要递归地获取符号链接的目标路径，直到找到一个非符号链接的路径为止。当搜索次数达到一定次数后表示文件打开失败。

```

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
路径
            end_op();
            return -1;
        }
        iunlockput(ip);
    }
    else
        break;
}

```

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

```

xv6 kernel is booting

init: starting sh
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok

```

```

test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: 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

```

## [实验中遇到的问题和解决方法]

本次实现一个系统调用的具体流程已经比较地熟悉，最主要的树实现相关连接函数。设定合理的最大递归深度(max\_depth)来防止无限递归循环。如果符号链接形成环或过于复杂的链条，可能会导致递归过深；同时需要确保多个线程或进程访问相同文件或符号链接时不会产生竞态条件，这通过对 inode 的加锁(ilock)和解锁(iunlockput)来保证。代码的编写还是有一定难度的，最后在指导视频和参考文献的帮助下我完成了本次实验。

## [实验心得]

通过本次实验，我深入理解了操作系统中文件系统的工作机制，尤其是符号

链接的解析与处理。通过编写 `sys_symlink` 函数，我学会了如何在文件系统中创建符号链接，并且通过递归解析路径中的符号链接，掌握了如何处理文件系统中的间接引用。同时，我还认识到在编写系统级代码时，资源管理和错误处理的重要性，确保每个操作都有对应的错误处理分支，避免资源泄漏和死锁。

## Lab9 的实验结果截图: ./grade-lab-fs

```
● ljk@LJK 'sPC:~/xv6-labs-2020$ ./grade-lab-fs
make: 'kernel/kernel' is up to date.
== Test running bigfile == running bigfile: OK (113.3s)
== Test running symlinktest == (0.6s)
== Test symlinktest: symlinks ==
    symlinktest: symlinks: OK
== Test symlinktest: concurrent symlinks ==
    symlinktest: concurrent symlinks: OK
== Test usertests == usertests: OK (188.8s)
== 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

```
ljk@LJK 'sPC:~/xv6-labs-2020$ git fetch
ljk@LJK 'sPC:~/xv6-labs-2020$ git checkout mmap
Switched to branch 'mmap'
Your branch is up to date with 'origin/mmap'.
ljk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
/*/*.o /*/*.d /*/*.asm /*/*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_in
e
```

### 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]

实现 `sys_mmap` 和 `sys_munmap` 系统调用的具体方法在前文的 Lab2 已经详细

流程过了，我们这也需要在源代码中加入如下代码：

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

根据提示 3，跟踪 mmap 为每个进程映射的内容。定义与第 15 课中描述的 VMA(虚拟内存区域)对应的结构体，记录 mmap 创建的虚拟内存范围的地址、长度、权限、文件等。vma，即虚拟内存区域(Virtual Memory Area)，是操作系统中用于管理进程虚拟内存的一个关键概念。它表示进程地址空间中的一个连续区域，并且具有相同的属性(如读、写、执行权限)。

所以我们需要定义 VMA 结构体，并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 中添加 VMA：

```
struct vma vma[VMASIZE];
```

接下来我们需要 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;
            }
        }
    }
}

```

页面错误时需要进行处理，处理由缺页中断引起的情况，并根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。在进程的虚拟地址空间中查找对应的虚拟内存区域，并将缺失的页面从文件加载到内存中，最终将该页面映射到进程的地址空间。如果任何一步失败，都会将进程标记为已被杀死。

接下来，我们就需要在 kernel/sysfile.c 中具体地实现两个函数：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 = PROUNDUP(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` 函数通过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，我们需要进行参数提取与验证、权限检查、长度调整和范围调整等。最后再进行遍历当前进程的虚拟内存区域数组 `vma`，寻找一个未使用的条目进行虚拟内存区域分配。

`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` 用于取消这种映射。接下来需要编写 `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 = PGROUNDOWN(addr);
    length = PROUNDUP(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)
            filewrite(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){
        fileclose(vma->file);
        vma->used = 0;
    }
}
return 0;
}

```

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。如果映射区域是共享的，还会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，确保内存的合理使用和资源的有效释放。

最后修改 uvmcopy 和 uvmunmap，避免因不合法而 panic 即可：

```

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 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

```

## [实验中遇到的问题和解决方法]

本实验的内容还是比较难的。理解什么是 mmap 技术和 munmap 技术时做好实验的前提。mmap 和 munmap 是一对用于管理进程虚拟内存映射的系统调用。它们的主要关系是互补的：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 用于取消这种映射。还有一个关键点就是 VMA 虚拟内存区域结构体的设计，一开始对这个结构体的设计没有什么头绪，但是在理解好 mmap

技术和 munmap 技术的关键点后，提示在参考的指导下完成了设计。

## [实验心得]

在本次实验中，我深入理解并实现了内存映射(mmap)和取消映射(munmap)的系统调用。通过实现 sys\_mmap 和 sys\_munmap 函数，我学会了如何在操作系统中管理进程的虚拟内存区域(VMA)。在 sys\_mmap 的实现过程中，我理解了如何从用户空间获取参数，并将文件或匿名内存映射到进程的虚拟地址空间。同时，通过实现 sys\_munmap，我掌握了如何取消映射并释放资源，确保系统内存的有效利用。通过这次实验，我不仅提升了编码能力，还增强了对操作系统底层机制的理解，为后续深入学习操作系统内核打下了坚实的基础。

## Lab10 的实验结果截图：./grade-lab-mmap

```
● ljk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-mmap
make: 'kernel/kernel' is up to date.
== Test running mmapptest == (1.6s)
== Test mmapptest: mmap f ==
    mmapptest: mmap f: OK
== Test mmapptest: mmap private ==
    mmapptest: mmap private: OK
== Test mmapptest: mmap read-only ==
    mmapptest: mmap read-only: OK
== Test mmapptest: mmap read/write ==
    mmapptest: mmap read/write: OK
== Test mmapptest: mmap dirty ==
    mmapptest: mmap dirty: OK
== Test mmapptest: not-mapped unmap ==
    mmapptest: not-mapped unmap: OK
== Test mmapptest: two files ==
    mmapptest: two files: OK
== Test mmapptest: fork_test ==
    mmapptest: fork_test: OK
== Test usertests == usertests: OK (83.2s)
== 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

```
ljk@LJK 'sPC:~/xv6-labs-2020$ git fetch
ljk@LJK 'sPC:~/xv6-labs-2020$ git checkout net
Branch 'net' set up to track remote branch 'net' from 'origin'.
Switched to a new branch 'net'
ljk@LJK 'sPC:~/xv6-labs-2020$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
/*/*.o /*/*.d /*/*.asm /*/*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
        user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/ \
e user/_nettests \
ph barrier
```

## 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]

按照实验指导要求所述，我们需要在 `kernel/e1000.c` 中完成两个函数的编写：`e1000_transmit` 和 `e1000_recv` 函数，前者函数用于将以太网帧传递给 `e1000` 网络设备的发送描述符环，并启动发送过程；后者函数用于接收来自 `e1000` 网络设备的数据包，并将它们传递给网络栈进行处理。

```
int e1000_transmit(struct mbuf *m)
{
    //
    // Your code here.
    //
    // the mbuf contains an ethernet frame; program it into
    // the TX descriptor ring so that the e1000 sends it. Stash
    // a pointer so that it can be freed after sending.
    //
    acquire(&e1000_lock);
    // 获得 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])
        mbuffree(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。
}

```

该函数接收一个 struct mbuf \*m 为参数，首先获取 e1000\_lock 锁，用于同步对 e1000 网络设备的访问。接着从寄存器中读取当前可用的描述符环的下一个索引，并检查描述符的状态，若状态不可用，说明发送队列已满，无法继续发送新的数据包，需要释放锁并返回 -1，表示发送失败。之后将 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，并且设置该描述符的命令字段。更新寄存器中的 TDT 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。最后别忘了释放锁。

```

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

```

该函数首先从寄存器中读取当前可用的 RX 描述符环的下一个索引。通过将 RDT 寄存器的值加一，并取模接收描述符环的大小，得到下一个可用的描述符索引。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。检查接收到的数据包长度是否超过 mbuf 的最大大小。

如果超过，触发 panic，表明数据包过大无法处理。然后将接收到的数据包长度存储到对应的 mbuf 中。调用 net\_rx() 函数，将 mbuf 传递给网络栈进行处理。网络栈将处理和解析接收到的数据包。分配一个新的 mbuf，并将其存储到接收描述符的缓冲区中，准备接收下一个数据包，将新的 mbuf 的头部地址存储到接收描述符中，以便接收数据包时能够正确写入数据。最后清零描述符状态字段，更新接收描述符索引和 RDT 寄存器。

该实验需要两个终端来验证结果，首先在第一个终端内输入 make server:

```
问题 输出 调试控制台 终端 端口 1

● ljk@LJK'sPC:~/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):

#### [实验中遇到的问题和解决方法]

本次实验的内容和计算机网络密切相关，本学期学习了计算机网络课程，因此对这方面比较熟悉。主要是写代码编程是遇到了两个问题。第一就是在 make qemu 是出现了一下报错：

```
kernel/e1000.c:163:5: error: redefinition of 'e1000_transmit'  
163 | int e1000_transmit(struct mbuf *m)  
     | ^~~~~~  
kernel/e1000.c:96:5: note: previous definition of 'e1000_transmit' was here  
 96 | int e1000_transmit(struct mbuf *m)  
     | ^~~~~~
```

经过检查后发现我在同一个文件 kernel/e1000.c 中定义了两次目标函数，这是一个比较低级的错误，删除了就没有报错了。

第二个问题就是在编写 e1000\_transmit 函数时，在一开始需要获取 e1000\_lock 锁，在最后记得要释放该锁。

还有就是本次实验的验证需要开启两个不同的终端进行输入。

## [实验心得]

在本次实验中，我们深入探讨并实现了 e1000 网络设备的数据包接收和发送功能。通过实现 e1000\_transmit 函数，我们掌握了如何编程 e1000 的 TX 描述符环来发送数据包，并确保发送后的 mbuf 可以被正确释放。通过实现 e1000\_recv 函数，我们学会了如何检查 RX 描述符的状态、处理接收到的数据包，并将它们传递给网络栈进行进一步处理。通过这些编程实践，我们加深了对网络设备驱动程序内部机制的理解，尤其是描述符环和中断处理机制。

## Lab11 的实验结果截图：./grade-lab-net

```
● ljk@LJK'sPC:~/xv6-labs-2020$ ./grade-lab-net
make: 'kernel/kernel' is up to date.
== Test running nettests == (2.5s)
== 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/JackinOflove/XV6-Operation-System.git>