

同濟大學

TONGJI UNIVERSITY

课题名称: 操作系统暑期课程设计项目报告

副标题: xv6 及 Labs 课程项目

学 院: 计算机科学与技术学院

专 业: 软件工程

小组成员: 2253377 李航

班 号: 42028703

指导教师: 王冬青

日 期: 2025 年 8 月 22 日

Lab0:Environment Setup	3
Lab1: Utilities	7
Boot xv6 (easy)	7
Sleep (easy)	8
Pingpong (easy)	11
Primes (moderate)/(hard)	14
Find (moderate)	17
Xargs (moderate)	21
Lab1 的实验整体测试(./grade-lab-util)截图:	23
Lab2:System Calls	25
System call tracing (moderate)	25
Sysinfo (moderate)	29
Lab2 的实验整体测试(./grade-lab-syscall)截图:	32
Lab3:Page Tables	34
Print a page table (easy)	34
A kernel page table per process (hard)	37
Simplify copyin/copyinstr (hard)	41
Lab3 的实验整体测试(./grade-lab-pgtbl)截图:	44
Lab4:Traps	45
RISC-V assembly (easy)	45
Backtrace (moderate)	48
Alarm (hard)	50
Lab4 的实验整体测试(./grade-lab- traps)截图:	54
Lab5: Lazy allocation	56
Eliminate allocation from sbrk() (easy)	56
Lazy allocation (moderate)	57

Lazytests and Usertests (moderate)	60
Lab5 的实验整体测试(./grade-lab-lazy)截图:	62
Lab6:Copy-on-Write Fork for xv6.....	64
Implement copy-on write (hard)	64
Lab6 的实验整体测试(./grade-lab-cow)截图:	70
Lab7: Multithreading	71
Uthread: switching between threads (moderate).....	71
Using threads (moderate).....	75
Barrier(moderate).....	79
Lab7 的实验整体测试(./grade-lab-thread)截图:	81
Lab8:Locks	83
Memory allocator (moderate)	83
Buffer cache (hard)	87
Lab8 的实验整体测试(./grade-lab-lock)截图:	92
Lab9: File system.....	93
Large files (moderate).....	93
Symbolic links (moderate).....	98
Lab9 的实验整体测试(./grade-lab-fs)截图:	101
Lab10:Mmap.....	103
mmap (hard).....	103
Lab10 的实验整体测试(./grade-lab-mmap)截图:	110
Lab11: Networking	111
Your Job (hard)	111
Lab11 的实验整体测试(./grade-lab-net)截图:	114
项目源码 github 仓库链接:	115

Lab0:Environment Setup

实验目的

Windows 的 Linux 子系统（Windows Subsystem for Linux，简称 WSL）是微软推出的一项功能，它能让用户在 Windows 操作系统中直接运行 Linux 环境，无需额外安装虚拟机或进行双系统启动。这一功能对开发人员和系统管理员而言颇具价值，因为它将 Windows 的易用性与 Linux 丰富的工具集有机结合起来。

本实验是项目整体的前期准备阶段，主要目标包括：安装 Ubuntu20.04 系统、完成 XV6 系统的配置，以及搭建 github 远程仓库。

实验步骤

1. 安装 Ubuntu 20.04

首先需开启个人电脑的 WSL 支持。打开具有管理员权限的 Shell，输入命令`Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux`，该命令会激活 WSL 服务，执行后需重启系统。

打开电脑控制面板，找到“启动或关闭 Windows 功能”，此时可看到 WSL 选项处于选中状态。

随后在微软应用商店中搜索“ubuntu”，选择所需版本进行安装，本次实验选用 Ubuntu 20.04 LTS。下载完成后，打开终端，按照指引完成新用户的注册及密码设置。



2. 配置 XV6 系统

依据实验配套教学指南，在 Ubuntu 终端命令行中输入`sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu`，一键安装 qemu 等所需依赖。

```
root@lh:~# sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
Reading package lists... Done
Building dependency tree
Reading state information... Done
git is already the newest version (1:2.25.1-1ubuntu3.14).
git set to manually installed.
The following additional packages will be installed:
acl admwita-icon-theme at-spi2-core binutils binutils-common binutils-x86-64-linux-gnu cpp-9 cpp-9-riscv64-linux-gnu
cpp-riscv64-linux-gnu dpkg-dev fakeroot fontconfig g++ g++-9 gcc gcc-10-cross-base-ports gcc-9 gcc-9-base gcc-9-cross-base-ports
gcc-9-riscv64-linux-gnu gcc-9-riscv64-linux-gnu-base gdb gdbserver gstreamer1.0-plugins-base gstreamer1.0-plugins-good
gstreamer1.0-x gtk-update-icon-cache hicolor-icon-theme humanity-icon-theme ibverbs-providers ipxe-qemu libaa1
libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl libasan5 libatk-bridge2.0-0 libatk1.0-0 libatk1.0-data
libatomic1 libatomic1-riscv64-cross libatspi2.0-0 libavahi-client3 libavahi-common-data libavahi-common3 libavc1394-0
libbabeltrace1 libbinutils libboost-iostreams1.71.0 libboost-thread1.71.0 libbrlapi0.7 libc-dev-bin libc6-dbg libc6-dev
libc6-dev-riscv64-cross libc6-riscv64-cross libcaca0 libcacard0 libcairo-gobject2 libcairo2 libcc1-0 libcdparanoia0 libcolor2
libcrypt-dev libctf-nobfd0 libctf0 libcupi2 libdatrie1 libdmg-perl libdvd4 libdwl libepoxy0 libfakeroot libfdt1
libfile-fcntllock-perl libgbl libgcc-9-dev libgcc-9-dev-riscv64-cross libgcc-s1-riscv64-cross libgdk-pixbuf2.0-0
libgdk-pixbuf2.0-bin libgdk-pixbuf2.0-common libgomp1 libgomp1-riscv64-cross libgraphite2-3 libgstreamer-plugins-base1.0-0
libgstreamer-plugins-good1.0-0 libgtk-3-0 libgtk-3-bin libgtk-3-common libharfbuzz0b libibverbs1 libiec61883-0 libiscsi7 libisl22
libitm1 libjack-jackd2-0 libjbig0 libjpeg-turbo8 libjpeg8 liblcms2-2 liblsan0 libmp3lame0 libmpeg3 libmpeg123-0 libnl-3-200
libnl-route-3-200 libopus0 liborc-0.4-0 libpango-1.0-0 libpangocairo-1.0-0 libpangoft2-1.0-0 libpcsc-lite libpixmap-1-0 libpme1
libquadmath0 librados2 libraw1394-11 librbdl librdmac1 librest-0.7-0 librsync2-2 librsync2-common libseamless0 libshout3
libslim0 libsoup-gnome2-4-1 libspeex libspice-server1 libstdc++-9-dev libtag1v6 libtag1v6-vanilla libthai-data libthai0
libtheora0 libtiff5 libtsan0 libtoolame0 libubsan1 libusbredirparser1 libv4l-0 libv4lconvert0 libvirglrenderer1 libvisual-0.4-0
libvp6 libvte-2.91-0 libvte-2.91-common libwaypack1 libwayland-cursor0 libwayland-egl1 libwayland-server0 libwebp6
libxcb-render0 libxcursor1 libxdamage1 libxkbcommon0 linux-libc-dev linux-libc-dev-riscv64-cross make manpages-dev
qemu-block-extra qemu-system-common qemu-system-data qemu-system-gui qemu-utils seabios sharutils ubuntu-mono
Suggested packages:
binutils-doc gcc-9-locales debconf debconf-i18n g++-multilib g++-9-multilib gcc-9-doc gcc-multilib autoconf automake libtool
flex bison gcc-doc gcc-9-multilib gdb-riscv64-linux-gnu gdb-doc gvfs glibc-doc colord cups-common bzip libdv-bin oss-compat
libvisual-0.4-plugins jackd2 liblms2-2 libraw1394-doc librsync2-bin speex gstreamer1.0-plugins-ugly
libstdc++-9-doc make-doc samba vde2 debootstrap sharutils-doc bsd-mailx | mailx
The following NEW packages will be installed:
acl admwita-icon-theme at-spi2-core binutils binutils-common binutils-x86-64-linux-gnu build-essential
cpp-9 cpp-9-riscv64-linux-gnu cpp-riscv64-linux-gnu dpkg-dev fakeroot fontconfig g++ g++-9 gcc gcc-10-cross-base-ports gcc-9
gcc-9-base gcc-9-cross-base-ports gcc-9-riscv64-linux-gnu gcc-9-riscv64-linux-gnu-base gcc-riscv64-linux-gnu gdb gdb-multiarch
gdbserver gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-x gtk-update-icon-cache hicolor-icon-theme
humanity-icon-theme ibverbs-providers ipxe-qemu libaa1 libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl
libasan5 libatk-bridge2.0-0 libatk1.0-0 libatk1.0-data libatomic1 libatomic1-riscv64-cross libatspi2.0-0 libavahi-client3
```

为验证安装结果，在终端分别输入`riscv64-unknown-elf-gcc --version`和`qemu-system-riscv64 --version`命令。验证结果如下：

```
lh01@lh:~$ 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.

lh01@lh:~$ qemu-system-riscv64 --version
QEMU emulator version 4.2.1 (Debian 1:4.2-3ubuntu6.30)
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
lh01@lh:~$
```

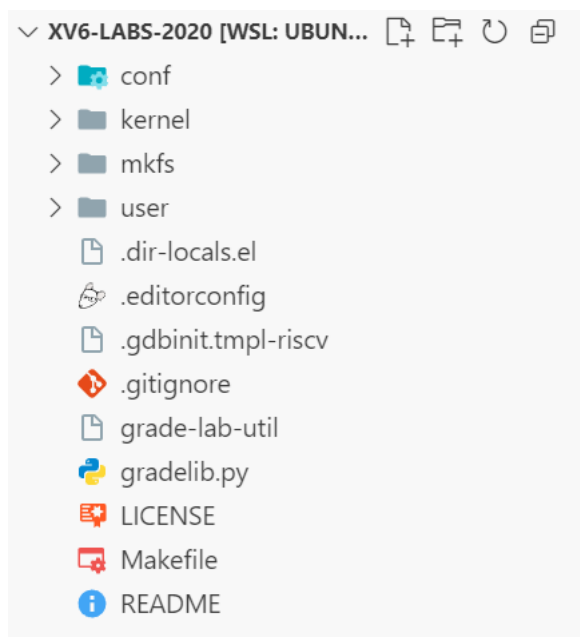
克隆实验所需代码，输入`git clone git://g.csail.mit.edu/xv6-labs-2020`。

```
lh01@lh:~$ git clone git://g.csail.mit.edu/xv6-labs-2020
Cloning into 'xv6-labs-2020'...
remote: Enumerating objects: 7008, done.
remote: Counting objects: 100% (7008/7008), done.
remote: Compressing objects: 100% (3612/3612), done.
remote: Total 7008 (delta 3650), reused 6526 (delta 3366), pack-reused 0
Receiving objects: 100% (7008/7008), 17.19 MiB | 5.49 MiB/s, done.
Resolving deltas: 100% (3650/3650), done.
```

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

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

完成上述环境配置后，便可在 VSCode 中修改代码如下图所示，并在相应终端进行调试，观察实验结果。



3. github 仓库的建立

首先在 github 上创建一个远程仓库，并获取该仓库的 HTTPS 地址。

接着在 WSL 终端中打开代码所在的子目录，依次输入以下命令：

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

```
cat .git/config
```

输入`git remote add github https://github.com/Effulgence12/OS-design-XV6-Operation-System.git`添加 git 仓库地址，之后输入`cat .git/config`查看配置。

输入用户名和密码完成配置后，将实验所用分支推送到 github，例如推送实验 1 使用的 util 分支，命令如下：

```
git checkout util
```

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

```
lh01@lh:~/xv6-labs-2020$ git push github util:util
Username for 'https://github.com': Effulgence12
Password for 'https://Effulgence12@github.com':
Enumerating objects: 6632, done.
Counting objects: 100% (6632/6632), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3227/3227), done.
Writing objects: 100% (6632/6632), 17.12 MiB | 569.00 KiB/s, done.
Total 6632 (delta 3419), reused 6579 (delta 3397)
remote: Resolving deltas: 100% (3419/3419), done.
To https://github.com/Effulgence12/OS-design-XV6-Operation-System.git
* [new branch]      util -> util
```

其他实验的版本控制操作类似，按照指引切换到相应实验目录，即可对不

同实验的分支进行管理，并传送到远程仓库。

实验中遇到的问题

环境搭建环节相对繁琐，在本次实验初始环境的搭建过程中，我碰到了不少棘手的问题。不过，通过查阅相关资料以及搜索对应的教程，这些问题最终都得到了妥善解决。实验中主要遇到在 WSL 中使用 GitHub 的连接与认证问题：

1. GitHub 连接拒绝：因 DNS 解析异常，通过修改 `/etc/hosts` 绑定 GitHub 真实 IP 解决。
2. 认证失败：GitHub 不支持密码认证，改用个人访问令牌（PAT）或 SSH 密钥。
3. 推送卡顿：切换网络或强制使用 HTTP/1.1，最终通过 SSH 协议稳定推送。

实验心得

搭建 xv6 实验环境的过程，为我提供了一个深入学习操作系统基础知识的绝佳契机。在这一过程中，需要安装 `make`、RISC-V 工具链等必要工具，这让我深刻认识到，了解并熟练掌握开发工具链是开展后续工作的关键，其对编译和调试环节的顺利进行有着不可忽视的重要性。

不过，xv6 环境的搭建并非毫无阻碍，期间我遇到了不少问题，像工具链的兼容性问题以及权限相关的问题等。而每一次分析并解决这些问题的经历，都让我对系统环境和工具链的理解更加透彻，操作也更为熟练，这无疑为我后续开展实验工作奠定了坚实的基础。

此外，在上述环境的搭建与配置过程中，很可能还会出现其他各种未曾预料到的问题，这就需要我们保持耐心，一步一步地排查错误、纠正问题，在这个过程中不断积累经验。

Lab1: Utilities

Boot xv6 (easy)

实验目的

借助 WSL 终端切换至 xv6-labs-2020 代码的 util 分支，同时运用 qemu 模拟器启动并运行 xv6 系统，进而观察实验的现象与结果。

实验步骤

在 lab0 配置的环境基础上，在 WSL 终端进入实验目录，并切换到第一个实验分支 util:

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

```
git checkout util
```

```
lh01@lh:~/xv6-labs-2020$ git checkout util
Already on 'util'
Your branch is up to date with 'origin/util'.
```

此时，在 xv6-labs-2020 目录下的终端中输入`make qemu`即可启动 xv6;

出现如下结果，表明已成功编译并启动 xv6 系统:

```
lh01@lh:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

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

启动后，在终端中输入`ls`，会显示初始文件系统中包含的 mkfs，其中大部分是可运行的程序，ls 便是其中之一;


```
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2059
xargstest.sh 2 3 93
cat        2 4 23968
echo       2 5 22800
forktest   2 6 13168
grep       2 7 27320
init       2 8 23896
kill       2 9 22768
ln         2 10 22720
ls         2 11 26208
mkdir      2 12 22872
rm         2 13 22856
sh         2 14 41752
stressfs   2 15 23872
usertests  2 16 147512
grind      2 17 37984
wc         2 18 25112
zombie     2 19 22272
console    3 20 0
$
```

若要退出 qemu，需键入`Ctrl-a x`，即先按下`Ctrl-a`，再按下`x`。

实验中遇到的问题和解决方法

本实验内容较为简单，未遇到明显问题。

实验心得

这是一个简单的小实验，实验中涉及的基本步骤是后续各个实验的操作基础。

Sleep (easy)

实验目的

- 1.为 xv6 实现 UNIX 程序 sleep。
- 2.实现的 sleep 应当按用户指定的 ticks 数暂停，其中 tick 是 xv6 内核定义的时间概念，即定时器芯片两次中断之间的时间。解决方案应该在文件 user/sleep.c 中。

实验步骤

在 user 文件夹下新建 sleep.c，判断一下输入格式，调用一下 sleep 即可。

实验代码如下：

```
#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... need a param\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[]` 是主函数 `main` 的标准参数，用于接收命令行输入的参数。`argc` 是整数，代表传递给程序的命令行参数数量，该数量包含程序本身的名字，因此至少为 1。`argv` 是字符指针数组，每个元素都是一个字符串（即字符指针），对应一个命令行参数，其中 `argv[0]` 是程序名，`argv[1]` 是第一个命令行参数，以此类推。

将编写完成的睡眠程序添加到 Makefile 的 `UPROGS` 中，可使 QEMU 编译该程序并能从 `xv6 shell` 中运行它，具体操作如下：打开 Makefile 文件。找到定义用户程序的 `UPROGS` 变量所在行，在该行中添加 `sleep` 程序的目标名称 ``$U/_sleep``。

```

UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_sleep\

```

编译运行程序：在终端执行`make qemu`命令，该命令会编译 xv6 并启动虚拟机，之后通过测试程序即可检测 sleep 程序的正确性。

执行`sleep 10`命令：程序等待一段时间而无任何输出。

```

xv6 kernel is booting

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

```

通过运行测试程序查看得分：

```

● lh01@lh:~/xv6-labs-2020$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.2s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.1s)

```

实验中遇到的问题和解决方法

本实验内容同样较为简单，按照指导要求操作，实验阶段未出现问题。

但是在运行测试程序阶段，提示 `usr/bin/env: 'python': No such file or directory` 表示系统中找不到 `python` 命令，原因和解决方法如下：

原因分析：在 Ubuntu 系统中，较新的版本（如 Ubuntu 20.04 及以上）默认只安装了 `python3`（Python 3.x 版本），而没有安装 `python` 命令（通常对应

Python 2.x 版本)。脚本 `./grade-lab-util` 中使用了 `#!/usr/bin/env python` 来指定解释器，但系统中没有 `python` 这个可执行文件，因此会报错。

解决方法：建立 `python` 到 `python3` 的链接

既然系统中已经有 `python3`，可以通过创建软链接的方式，让 `python` 命令指向 `python3`，输入命令：`sudo ln -s /usr/bin/python3.8 /usr/bin/python` 即可解决。

实验心得

通过本实验，我首先学会了如何为 xv6 操作系统添加新程序；其次，在 VSCode 终端运行程序的过程中，通过理解 `argc` 和 `argv` 命令行参数，我能够编写更灵活、动态的命令程序，进而利用系统调用来控制进程行为。

Pingpong (easy)

实验目的

编写一个利用 UNIX 系统调用的程序，通过两个管道（每个方向一个）在两个进程间“ping-pong”传递一个字节。具体流程为：父进程向子进程发送一个字节；子进程打印“<pid>: received ping”（其中<pid>为自身进程 ID），并通过管道向父进程回传该字节后退出；父进程从子进程读取字节，打印“<pid>: received pong”后退出。解决方案需编写在 `user/pingpong.c` 文件中。

实验步骤

1. 创建管道

定义两个管道数组分别用于双向通信，其中‘`p1`’负责父进程向子进程发送数据，‘`p2`’负责子进程向父进程回传数据，同时定义缓冲区存储传递的字节数据，最后通过‘`pipe`’系统调用初始化两个管道：

```
int p1[2], p2[2]; // 双向通信管道, p1(父→子), p2(子→父)

char buf[1];      // 用于存储传递的单个字节

pipe(p1);         // 初始化管道 p1

pipe(p2);         // 初始化管道 p2
```

2. 使用 fork 创建子进程

通过`fork`系统调用创建子进程，利用其返回值区分子进程（返回 0）和父进程（返回子进程 PID），从而分别执行不同的读写逻辑。

3. 通过 read 和 write 操作管道

父进程先关闭`p1`的读端和`p2`的写端，向`p1`的写端写入数据，随后从`p2`的读端读取子进程回传的数据；

子进程先关闭`p1`的写端和`p2`的读端，从`p1`的读端读取父进程发送的数据，处理后向`p2`的写端回传数据；

操作完成后关闭相应管道端以释放资源。

4. 使用 getpid()获取进程 ID

在打印信息时，通过`getpid()`系统调用获取当前进程的 ID，分别在子进程和父进程中输出对应的日志。

程序源代码：

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

int main()
{
    int p1[2], p2[2]; // 两个管道，p1 用于父进程向子进程发送数据，p2 用于子进程向父进程发送数据
    char buf[1];
    pipe(p1); // 创建管道 p1
    pipe(p2); // 创建管道 p2
    if (fork() == 0)
    { // 子进程
        close(p1[1]); // 关闭子进程中不需要的写端
        close(p2[0]); // 关闭子进程中不需要的读端
        read(p1[0], buf, 1); // 从父进程接收字节
        printf("%d: received ping\n", getpid());
        write(p2[1], "p", 1); // 向父进程发送字节
        close(p1[0]); // 关闭管道读端
        close(p2[1]); // 关闭管道写端
        exit(0);
    }
    else
    { // 父进程
        close(p1[0]); // 关闭父进程中不需要的读端
```

```

    close(p2[1]); // 关闭父进程中不需要的写端
    write(p1[1], "p", 1); // 向子进程发送字节
    read(p2[0], buf, 1); // 从子进程接收字节
    printf("%d: received pong\n", getpid());
    close(p1[1]); // 关闭管道写端
    close(p2[0]); // 关闭管道读端
    wait(0); // 等待子进程结束
    exit(0);
}
}

```

该程序展示了父子进程间的简单通信模型：子进程通过管道向父进程写入数据，父进程从管道读取数据，模拟“ping-pong”通信过程。子进程负责写入时需关闭读端，写入完成后关闭写端；父进程负责读取时需等待子进程完成写入，读取前关闭写端，读取完成后关闭读端。

其中，fork 是 UNIX 系统调用，用于创建新进程。新进程是调用进程的副本：子进程中 fork() 返回 0，父进程中 fork() 返回新创建子进程的进程 ID (PID)。

在终端执行`make qemu`后，输入`pingpong`命令运行程序，随后查看正确得分。

```

$ pingpong
4: received ping
3: received pong

```

```

lh01@lh:~/xv6-labs-2020$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (0.8s)

```

实验中遇到的问题和解决方法

本次实验中，子进程写入和父进程读取的代码编写相对简单，核心在于区分子进程与父进程，而理解 fork() 函数是关键。fork() 函数通过返回值（0 或子进程 PID）区分子进程和父进程，因此可在代码中使用 if-else 语句区分两者的具体行为，进而编写相应逻辑。

实验心得

本实验的核心目标是利用 fork 和 pipe 系统调用来实现父子进程间的简单通信。在实现过程中，我首先理解了 fork 如何创建新进程，以及新进程与父进程

如何共享相同的代码和数据空间。

通过 `pipe` 系统调用，我学会了如何在进程间创建通信通道，并使用文件描述符进行读写操作。实验中，我实现了子进程向父进程发送消息、父进程接收消息的功能，具体流程为：父进程创建管道并调用 `fork` 生成子进程，子进程通过管道发送字符，父进程读取该字符并打印对应信息。这一过程让我对进程间通信的基本原理和实现方式有了更清晰的认识。

Primes (moderate)/(hard)

实验目的

使用管道编写素数筛选（`prime sieve`）的并发版本。这一思路由 UNIX 管道的发明者 Doug McIlroy 提出，相关实现方法可参考指定网页中间的图示及周边说明。解决方案需编写在 `user/primes.c` 文件中。

实验步骤

1. 初始进程与数据输入

创建初始父进程，该进程负责将 2 到 35 的整数序列写入管道的写端。进程创建采用动态方式：无需预先创建所有进程，而是在每一步迭代中仅更新当前的一对父子进程关系，仅当需要处理后续数据时才创建新的子进程，实现按需创建的轻量化机制。

2. 进程创建与管道通信

对于 2 到 35 中筛选出的每个素数，对应创建一个独立进程。进程间通过管道建立通信链路：每个进程从左侧的父进程通过读端读取待处理数据，再通过另一个管道的写端将筛选后的数据传递给右侧的子进程，形成链式的管道通信结构。

3. 素数筛选逻辑

每个生成的进程中，从管道读取的第一个数据即为当前进程需记录的素数。对于剩余读取到的数字，进程会以该素数为基准进行检查：若数字不能被当前素数整除（即符合素数候选条件），则保留该数字并写入右侧子进程的管道；若能被整除（即非素数），则直接跳过不传递。

4. 文件描述符管理

在数据传递或进程状态更新时，需及时关闭当前进程不再使用的文件描述

符（如已完成读取的管道读端、已完成写入的管道写端）。这一操作可避免 xv6 系统在父进程遍历到 35 之前因文件描述符耗尽而出现资源不足的问题。

5. 进程同步与退出机制

父进程需通过等待机制（如 wait 系统调用）等待子进程执行完毕，确保子进程释放的资源被及时回收。主 primes 进程作为初始进程，需等待所有素数输出完成，且所有子进程、孙进程等链式进程均退出后，才会最终结束运行，保证整个筛选流程的完整性和资源的彻底释放。

源代码如下：

```
#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); // 第一个数据为质数，进行输出
        pipe(fd);                    // 创建新管道和子进程
        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)) // 从原管道读取数据
            {
                if (num % prime != 0) // 不能被当前质数整除的数写入新管道
                    write(fd[1], &num, 4);
            }
            close(oldFd[0]); // 原管道数据读取完毕，关闭读端
            close(fd[1]);    // 关闭新管道写端
            wait((int *)0);  // 等待子进程结束
        }
    }
    else
        close(oldFd[0]); // 无法从原管道读取数据时，直接关闭读端
```



```

        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 的整数，写入管道写端
                write(fd[1], &i, 4);
            close(fd[1]); // 数据写入完成，关闭管道写端并等待子进程结束
            wait((int *)0);
        }
        exit(0);
    }

```

这段代码的实现具有一定复杂度。在 `main` 函数中，首先创建管道 `fd`，随后通过 `fork` 创建子进程。父进程遍历 2 到 35 的整数并写入管道写端，子进程则调用 `subProcess (fd)` 函数处理管道中的数据。

在 `subProcess` 函数中，首先关闭传入管道的写端 `oldFd [1]`，仅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质数 `prime`，若读取成功，则将其打印输出，并创建新的管道 `fd` 和子进程。子进程通过递归调用 `subProcess (fd)` 继续处理新管道中的数据；父进程则关闭新管道的读端 `fd [0]`，从原管道读取剩余数据，将不能被当前质数 `prime` 整除的数字写入新管道，最后关闭原管道读端、新管道写端，并等待子进程结束。

若无法从原管道读取到数据（即 `oldFd [0]` 无数据），则直接关闭原管道读端并退出。

在终端执行 `make qemu` 后，输入 `primes` 命令运行程序，随后查看正确得分。

```
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

```

Tony

```
lh01@lh:~/xv6-labs-2020$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.3s)
```

实验中遇到的问题和解决方法

本次实验内容较为复杂，初期在解决思路存在困惑，例如未想到通过递归方式处理进程关系，且难以用代码清晰表达父子进程间的逻辑。后续通过参考教学视频资源，逐步理解了管道与递归结合的实现方式，完成了代码编写。此外，需注意在代码中两处需要使用 `wait((int *)0)` 让父进程等待子进程结束，避免遗漏。

实验心得

在本次较复杂的代码编写过程中，我学会了如何结合进程间通信与递归调用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的方式，我对并发编程的基本概念和实现技术有了更深入的理解，也体会到了 UNIX 系统中管道机制在进程协作方面的灵活性。

Find (moderate)

实验目的

编写一个简化版的 UNIX `find` 程序，用于在目录树中查找所有具有特定名称的文件。解决方案需编写在 `user/find.c` 文件中。

实验步骤

本实验实现了递归的文件搜索功能，核心逻辑如下：

pfilename 函数：从路径中提取文件名（通过查找最后一个/后的字符串）。

find 函数：接受搜索路径和目标文件名作为参数，递归遍历目录树：

打开路径并获取文件状态，区分文件类型（常规文件 / 目录）。

若为常规文件，比较文件名与目标名，匹配则输出完整路径。

若为目录，遍历所有目录项，跳过.和..，拼接子项完整路径后递归调用 find。

main 函数：检查命令行参数合法性，调用 find 函数启动搜索。

源代码如下：

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

// 提取路径中的文件名部分
char *pfilename(char *path)
{
    char *p;
    // 从路径末尾向前查找最后一个 '/'
    for (p = path + strlen(path); p >= path && *p != '/'; p--)
        ;
    p++; // 指向文件名的第一个字符
    return p;
}

// 递归查找文件
int find(char *path, char *filename)
{
    int fd;           // 目录文件描述符
    char buf[512], *p; // 路径缓冲区及指针
    struct stat st;    // 文件状态结构体
    struct dirent de;  // 目录项结构体

    // 尝试打开路径
    if ((fd = open(path, 0)) < 0)
    {
        fprintf(2, "open fail%s\n", path);
        exit(1);
    }
}
```

```

// 获取文件状态
if (fstat(fd, &st) < 0)
{
    fprintf(2, "fstat fail%s\n", path);
    close(fd);
    exit(1);
}

switch (st.type)
{
case T_FILE: // 常规文件
    // 比较文件名与目标文件名, 匹配则输出路径
    if (0 == strcmp(pfilename(path), filename))
        fprintf(1, "%s\n", path);
    break;

case T_DIR: // 目录
    strcpy(buf, path); // 复制路径到缓冲区
    p = buf + strlen(buf); // 移动指针到路径末尾
    *p++ = '/'; // 添加路径分隔符

    // 遍历目录项
    while (read(fd, &de, sizeof(de)) == sizeof(de))
    {
        if (de.inum == 0) // 跳过无效条目
            continue;
        // 跳过当前目录(.)和上级目录(..)
        if (0 == strcmp(".", de.name) || 0 == strcmp("../", de.name))
            continue;

        // 拼接完整路径
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0; // 确保字符串结束符

        // 获取文件状态, 失败则输出提示并继续
        if (stat(buf, &st) < 0)
        {
            printf("find: cannot stat %s\n", buf);
            continue;
        }
    }
}

```

```

        find(buf, filename); // 递归查找子目录
    }
    break;
}

close(fd); // 关闭文件描述符
return 0;
}

int main(int argc, char *argv[])
{
    if (argc < 3) // 检查参数数量
    {
        fprintf(2, "not enough arguments\n");
        exit(1);
    }
    find(argv[1], argv[2]); // 调用查找函数
    exit(0);
}

```

在终端执行 `make qemu` 后，输入相应命令（如 `find . README`）运行程序，随后查看正确得分。

```

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

```

```

● lh01@lh:~/xv6-labs-2020$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (0.9s)
== Test find, recursive == find, recursive: OK (1.0s)

```

实验中遇到的问题和解决方法

本实验实现难度中等，初期缺乏清晰思路，参考指导提示 "Look at user/ls.c to see how to read directories" 后，从 `ls.c` 中借鉴了目录遍历的方法。此外，需特别注意处理 `.`（当前目录）和 `..`（上级目录）这两个特殊目录名，通过代码 `if (0 == strcmp(".", de.name) || 0 == strcmp("..", de.name)) continue;` 避免递归陷入死

循环。

实验心得

通过本次实验，在教学资料的辅助下，我成功实现了递归文件搜索功能，对递归思想在目录遍历中的应用有了更深刻的理解。同时，通过使用 `open`、`fstat`、`read`、`stat` 等系统调用，我进一步掌握了这些接口的功能和使用场景，对文件系统的层级结构及目录项的处理机制也有了更清晰的认识。

Xargs (moderate)

实验目的

编写一个简化版的 UNIX `xargs` 程序，其功能为从标准输入按行读取内容，并为每行内容执行指定命令，将该行内容作为参数传递给该命令。解决方案需编写在 `user/xargs.c` 文件中。

实验步骤

`xargs` 命令的核心作用是将标准输入转换为命令行参数，而管道符|的作用是将左侧命令的标准输出转为标准输入，提供给右侧命令作为参数。

代码的执行逻辑如下：

首先将 `xargs` 自身的命令行参数（如 `xargs echo` 中的 `echo`）存入 `args` 数组。

进入循环读取标准输入的每一行内容，存储在 `buf` 数组中。

为每行内容添加字符串结束符，将其追加到 `args` 数组末尾，并标记参数列表结束。

创建子进程，通过 `exec` 函数执行 `args[1]` 指定的命令（如 `echo`），并传入完整参数列表。

父进程等待子进程执行完毕，重复上述过程直到读取完所有输入行。

源代码如下：

```
#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]; // 用于存储执行命令的参数
    int p;
    // 将 xargs 自身的命令行参数存入 args 数组
    for (p = 0; p < argc; p++)
        args[p] = argv[p];

    char buf[256]; // 用于存储从标准输入读取的一行内容
    while (1) // 循环读取标准输入的每一行
    {
        int i = 0;
        // 读取一行内容（以换行符'\n'为结束标志）
        while ((read(0, buf + i, sizeof(char)) != 0) && buf[i] != '\n')
            i++;

        if (i == 0) // 若未读取到内容，说明已读完所有行，退出循环
            break;

        buf[i] = 0; // 在字符串末尾添加结束符（exec 函数要求）
        args[p] = buf; // 将读取到的行内容附加到参数列表后
        args[p + 1] = 0; // 标记参数列表结束

        if (fork() == 0) // 创建子进程执行命令
        {
            // 执行 args[1]指定的命令，传入 args+1 作为参数
            exec(args[1], args + 1);
            printf("exec err\n"); // 若 exec 执行失败，打印错误信息
        }
        else
        {
            wait((void *)0); // 父进程等待子进程执行完毕
        }
    }
    exit(0);
}

```

在终端执行 `make qemu` 后，输入相应命令运行程序，随后查看正确得分。

```
xv6 kernel is booting

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

```
● lh01@lh:~/xv6-labs-2020$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (1.0s)
```

实验中遇到的问题和解决方法

本次实验的关键在于理解 `xargs` 命令的核心功能：将标准输入转换为命令参数。通过查阅资料明确了其作用后，重点掌握了 `read` 函数的使用方式——`read(0, buf + i, sizeof(char))` 中，`0` 表示标准输入的文件描述符，`buf + i` 指定存储位置，`sizeof(char)` 指定每次读取 1 字节，以此实现按字符读取行内容。

实验心得

本实验难度相对较低，主要让我掌握了如何处理标准输入并将其作为参数传递给后续命令（通过管道机制）。同时，通过实践进一步熟悉了 `fork`、`exec`、`wait` 等进程控制函数的使用，加深了对标准输入输出处理及进程创建执行流程的理解。

Lab1 的实验整体测试(./grade-lab-util)截图：

在项目目录下添加 `time.txt` 文件，输入完成本实验用时（小时数，单个整数），然后运行 `./grade-lab-util` 进行整体测试，结果如下：


```
lh01@lh:~/xv6-labs-2020$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.0s)
== Test sleep, returns == sleep, returns: OK (0.8s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.1s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.1s)
== Test find, in current directory == find, in current directory: OK (1.0s)
== Test find, recursive == find, recursive: OK (1.1s)
== Test xargs == xargs: OK (1.1s)
== Test time ==
time: OK
Score: 100/100
```

测试无误后将实验结果上传至 GitHub 仓库对应分支。

```
lh01@lh:~/xv6-labs-2020$ git push github util:util
Username for 'https://github.com': Effulgence12
Password for 'https://Effulgence12@github.com':
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 16 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (10/10), 3.83 KiB | 3.83 MiB/s, done.
Total 10 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/Effulgence12/OS-design-XV6-Operation-System.git
 40db4f0..212a1a2  util -> util
```

Lab2: System Calls

在开始 Lab2 时，需先执行以下命令切换至实验分支并获取相关资源，同时清理之前的编译文件：

执行 `git fetch` 命令，从远程仓库拉取最新的分支信息，确保本地能获取到 `syscall` 分支的更新。

运行 `git checkout syscall`，切换到本次实验所需的 `syscall` 分支，以获取该分支下的实验资源。

输入 `make clean`，清除之前编译生成的目标文件和可执行文件，避免旧文件对本次实验编译过程产生干扰。

```
lh01@lh: /xv6-labs-2020$ git fetch
lh01@lh: /xv6-labs-2020$ git checkout syscall
Branch 'syscall' set up to track remote branch 'syscall' from 'origin'.
Switched to a new branch 'syscall'
lh01@lh: /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/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stress
fs user/_usertests user/_grind user/_wc user/_zombie
```

完成上述步骤后，即可基于 `syscall` 分支的环境开展 Lab2 的相关实验操作。

System call tracing (moderate)

实验目的

本实验要求添加一个系统调用跟踪功能，该功能可辅助后续实验的调试。具体需实现一个新的 `trace` 系统调用，其接收一个整数参数“掩码（mask）”，通过掩码的比特位指定需要跟踪的系统调用。例如，若要跟踪 `fork` 系统调用，程序可调用 `trace(1 << SYS_fork)`（其中 `SYS_fork` 是 `kernel/syscall.h` 中定义的系统调用编号）。

修改 `xv6` 内核后，当系统调用即将返回时，若其编号在掩码中被设置，则需打印一行信息，包含进程 ID、系统调用名称及返回值（无需打印参数）。`trace` 系统调用仅对调用它的进程及其后续创建的子进程启用跟踪功能，不影响其他进程。

实验步骤

1. 声明系统调用号：在 `kernel/syscall.h` 中为 `trace` 分配一个未使用的系统调用号，例如：

```
#define SYS_trace 22 // 选择未被占用的编号
```

2. 声明用户态函数原型：在 `user/user.h` 中添加 `trace` 系统调用的原型，供用户程序调用：

```
int trace(int mask); // 原型声明，参数为跟踪掩码
```

3. 添加系统调用存根：在 `user/usys.pl` 中添加 `trace` 的条目，确保 Perl 脚本生成对应的汇编存根（用于用户态到内核态的切换）：

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

4. 实现内核态处理函数

在 `kernel/sysproc.c` 中编写 `sys_trace` 函数，用于接收用户态传入的掩码并存储到进程结构体中：

```
uint64
sys_trace(void)
{
    int mask;
    // 从用户态获取第一个参数（掩码），失败则返回-1
    if (argint(0, &mask) < 0)
        return -1;
    // 将掩码保存到当前进程的 proc 结构中
    struct proc *p = myproc();
    p->trace_mask = mask;
    return 0; // 成功执行
}
```

5. 扩展进程结构体

在 `kernel/proc.h` 的 `struct proc` 中添加 `trace_mask` 字段，用于存储当前进程的跟踪掩码：

```
struct proc {
    // 其他字段……
    int trace_mask; // 新增字段：存储 trace 系统调用的掩码
};
```

5. 继承跟踪掩码

修改 `kernel/proc.c` 中的 `fork` 函数，使子进程继承父进程的 `trace_mask`，确保跟踪功能对后代进程生效：

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

6. 定义系统调用名称数组

在 kernel/syscall.c 中添加 syscall_names 数组，映射系统调用编号与名称，便于打印跟踪信息：

```
// 系统调用名称数组，索引对应系统调用编号
static char *syscall_names[] = {
    "", "fork", "exit", "wait", "pipe", "read",
    "kill", "exec", "fstat", "chdir", "dup",
    "getpid", "sbrk", "sleep", "uptime", "open",
    "write", "mknod", "unlink", "link", "mkdir",
    "close", "trace" // 包含新增的 trace 系统调用
};
```

7. 更新系统调用处理逻辑

并在

修改 kernel/syscall.c 中的 syscall 函数，在系统调用返回前检查跟踪掩码，若需跟踪则打印信息：

```
extern uint64 sys_trace(void);
[SYS_trace] sys_trace, //Lab 2-1
void
syscall(void)
{
    int num;
    struct proc *p = myproc(); // 获取当前进程

    num = p->trapframe->a7; // 从陷阱帧中获取系统调用编号
    // 检查系统调用编号的合法性
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        // 执行系统调用并保存返回值到陷阱帧
        p->trapframe->a0 = syscalls[num]();
        // 若当前系统调用在跟踪掩码中，则打印进程 ID、调用名称及返回值
        if ((1 << num) & p->trace_mask) {
            printf("%d: syscall %s -> %d\n",
                p->pid, syscall_names[num], p->trapframe->a0);
        }
    }
    else
    {
        // 处理未知系统调用
        printf("%d %s: unknown sys call %d\n",
```

```

        p->pid, p->name, num);
    p->trapframe->a0 = -1;
}

}

```

7. 编译与运行

执行 `make qemu` 编译内核并启动 `xv6`，通过相关命令（如 `trace 1 fork`）验证 `trace` 系统调用的功能是否正常。

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 2147483647 grep hello README
3: syscall trace -> 0
3: syscall exec -> 3
3: syscall open -> 3
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
3: syscall close -> 0
$ QEMU: Terminated

```

```

● lh01@lh:~/xv6-labs-2020$ ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (0.6s)
    (Old xv6.out.trace_32_grep failure log removed)
== Test trace all grep == trace all grep: OK (0.9s)
    (Old xv6.out.trace_all_grep failure log removed)
== Test trace nothing == trace nothing: OK (1.1s)
== Test trace children == trace children: OK (9.7s)
    (Old xv6.out.trace_children failure log removed)

```

实验中遇到的问题和解决方法

本次实验是首次完整实现系统调用，整体难度适中。关键在于按照指导步骤逐步完成配置，包括添加系统调用声明、修改结构体与函数、继承跟踪掩码等，确保每个环节无遗漏即可顺利完成。

实验心得

通过本实验，我从代码层面深入理解了系统调用的实现流程，掌握了系统调用追踪的概念与具体实现方法。借助该跟踪技术，能够有效监控程序与操作

系统内核之间的交互，获取详细的系统调用信息，这对于理解程序运行机制和调试程序具有重要意义。

Sysinfo (moderate)

实验目的

本实验要求添加一个 `sysinfo` 系统调用，用于收集运行中系统的信息。该系统调用接收一个参数：指向 `struct sysinfo` 的指针（定义见 `kernel/sysinfo.h`）。内核需要填充该结构体的字段：`freemem` 字段设为空闲内存的字节数，`nproc` 字段设为状态非 `UNUSED` 的进程数量。提供的测试程序 `sysinfotest` 若输出“`sysinfotest: OK`”，则表示实验通过。

实验步骤

1. 声明和定义系统调用：分配系统调用编号：在 `kernel/syscall.h` 中为 `sysinfo` 添加一个未使用的系统调用编号：

```
#define SYS_sysinfo 23 // lab2-2 分配新的系统调用号
```

2. 声明结构体与函数原型：由于 `sysinfo` 需使用 `struct sysinfo`，需在 `user/user.h` 中预先声明该结构体并添加系统调用原型：

```
struct sysinfo; // 提前声明结构体
int sysinfo(struct sysinfo *); // 系统调用原型，参数为指向 struct sysinfo 的指针
```

3. 添加系统调用存根：在 `user/usys.pl` 中添加 `sysinfo` 的条目，确保生成用户态到内核态的汇编转换代码：

```
entry("sysinfo");
```

4. 内核中实现 `sysinfo` 系统调用及辅助函数

实现 `sys_sysinfo` 函数：在 `kernel/sysproc.c` 中编写核心逻辑，负责收集系统信息并返回给用户空间：

```
uint64
sys_sysinfo(void)
{
    struct sysinfo info; // 内核态的 sysinfo 结构体，用于存储系统信息
    struct sysinfo *uinfo; // 指向用户空间的 sysinfo 结构体指针

    // 从用户空间获取 struct sysinfo 的地址，失败则返回-1
    if (argaddr(0, (uint64 *)&uinfo) < 0)
        return -1;
```

```

// 收集系统信息：空闲内存字节数和非 UNUSED 状态的进程数
info.freemem = free_mem(); // 获取空闲内存
info.nproc = num_procs(); // 获取活跃进程数

// 将内核态的 info 结构体复制到用户空间的 uinfo 指向的地址
if (copyout(myproc()->pagetable, (uint64)uinfo, (char *)&info,
sizeof(info)) < 0)
    return -1;

return 0; // 成功执行
}

```

5. 实现 free_mem 函数：在 kernel/kalloc.c 中添加该函数，统计空闲内存总字节数（内核通过 kmem.freelist 链表管理空闲内存块）：

```

uint64
free_mem(void)
{
    struct run *r;
    uint64 free = 0; // 累计空闲内存字节数

    acquire(&kmem.lock); // 加锁，确保并发安全
    // 遍历空闲链表，累加每个内存块的大小（每个块为一页，大小为 PGSIZE）
    for (r = kmem.freelist; r; r = r->next)
        free += PGSIZE;
    release(&kmem.lock); // 释放锁

    return free;
}

```

6. 实现 num_procs 函数：在 kernel/proc.c 中添加该函数，统计状态不为 UNUSED 的进程数量：

```

int
num_procs(void)
{
    struct proc *p;
    int count = 0; // 累计活跃进程数

    // 遍历进程数组，统计状态非 UNUSED 的进程
    for (p = proc; p < &proc[NPROC]; p++) {
        if (p->state != UNUSED)
            count++;
    }
}

```

```

}

return count;

}

```

7. 在 sysproc.h 中添加相关头文件和函数声明

```

#include "sysinfo.h"

uint64 free_mem(void); // 声明 free_mem 函数
int num_procs(void); // 声明 num_procs 函数

```

8. 配置编译选项

在 Makefile 的 UPROGS 中添加测试程序 sysinfotest，确保其被编译：

```
UPROGS = ... $U/_sysinfotest // 追加测试程序
```

9. 编译调试运行

执行 make qemu 编译内核并启动 xv6，运行 sysinfotest 命令，若输出“sysinfotest: OK”，则实验成功。

```

xv6 kernel is booting

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

```

运行实验评分结果如下。

```

● 1h01@1h:~/xv6-labs-2020$ ./grade-lab-syscall Sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (2.7s)
● 1h01@1h:~/xv6-labs-2020$

```

实验中遇到的问题和解决方法

借助前一个实验的基础，本次实验的流程相对熟悉，核心在于 sys_sysinfo 函数及辅助函数的实现。通过查阅资料并参考 sysfile.c 和 file.c 中关于内存复制和进程遍历的代码，逐步理解了 freememory（统计空闲内存）、procnum（统计进程数）和 copyout（内核到用户空间数据复制）的实现逻辑，最终顺利完成代码编写。

在实验过程中，首次 make qemu 编译内核时出现如下报错

```
kernel/sysproc.c:115:20: error: storage size of 'info' isn't known
115 |     struct sysinfo info;    // 内核态的sysinfo结构体，用于存储系统信息
    |
kernel/sysproc.c:123:20: error: implicit declaration of function 'free_mem' [-Werror=implicit-function-declaration]
123 |     info.freemem = free_mem(); // 获取空闲内存
    |
kernel/sysproc.c:124:18: error: implicit declaration of function 'num_procs' [-Werror=implicit-function-declaration]
124 |     info.nproc = num_procs(); // 获取活跃进程数
    |
kernel/sysproc.c:115:20: error: unused variable 'info' [-Werror=unused-variable]
115 |     struct sysinfo info;    // 内核态的sysinfo结构体，用于存储系统信息
    |
cc1: all warnings being treated as errors
make: *** [: kernel/sysproc.o] Error 1
```

分析报错信息得知在 sysproc.c 文件开头缺少相关头文件及函数的声明。根据要求引入头文件"sysinfo.h"并声明函数 free_mem 和 num_procs 后解决该问题。

实验心得

本次实验成功为 xv6 添加了 sysinfo 系统调用，实现了对系统空闲内存和活跃进程数的收集。过程中，我学会了参考现有函数（如内存管理和进程管理相关函数）来设计所需功能，同时掌握了内核态与用户态之间数据传递的方法（如 copyout 函数）。此外，这次实验也让我意识到编写代码时需注重细节，尤其是外部函数的声明和并发安全（如加锁机制），这些都是保证系统调用正确运行的关键。

Lab2 的实验整体测试(./grade-lab-syscall)截图：

在项目目录下添加 time.txt 文件，输入完成本实验用时（小时数，单个整数），然后运行 ./grade-lab-syscall 进行整体测试，结果如下：

```
● lh01@lh:~/xv6-labs-2020$ ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.6s)
== Test trace all grep == trace all grep: OK (0.9s)
== Test trace nothing == trace nothing: OK (0.9s)
== Test trace children == trace children: OK (9.7s)
== Test sysinfotest == sysinfotest: OK (1.7s)
== Test time ==
time: OK
Score: 35/35
```

测试无误后将实验结果上传至 GitHub 仓库对应分支。

```
● lh01@lh:~/xv6-labs-2020$ git push github syscall:syscall
Enumerating objects: 26, done.
Counting objects: 100% (26/26), done.
Delta compression using up to 16 threads
Compressing objects: 100% (13/13), done.
Writing objects: 100% (14/14), 3.39 KiB | 1.70 MiB/s, done.
Total 14 (delta 11), reused 0 (delta 0)
remote: Resolving deltas: 100% (11/11), completed with 11 local objects.
To https://github.com/Effulgence12/OS-design-XV6-Operation-System.git
 8813049..e71681c  syscall -> syscall
```

Lab3:Page Tables

开始 Lab3 时，需执行以下命令切换到对应的实验分支并准备实验环境：

`git fetch`，从远程仓库拉取最新的分支数据，确保本地能够获取到 `pgtbl` 分支的相关资源。

`git checkout pgtbl`，切换到本次实验所需的 `pgtbl` 分支，以获取该分支下的页表相关实验代码和资源。

`make clean`，清除之前编译过程中生成的目标文件、可执行文件等，避免旧文件对本次实验的编译和运行产生干扰。

在本实验中，将对页表进行查看与修改，以实现加快某些系统调用速度和检测已访问页面的功能。熟悉相关核心文件如下：

`kernel/memlayout.h`：用于定义内存布局。

`kernel/vm.c`：包含大部分虚拟内存（VM）相关代码。

`kernel/kalloc.c`：包含物理内存的分配与释放逻辑。

完成上述步骤后，即可基于 `pgtbl` 分支的环境开展 Lab3 中与页表相关的实验内容。

Print a page table (easy)

实验目的

定义一个名为 `vmprint()` 的函数，该函数接收 `pagetable_t` 类型的参数，并按指定格式打印页表。在 `exec.c` 中 `return argc` 之前插入 `if(p->pid==1)` `vmprint(p->pagetable)`，以打印第一个进程的页表。若通过 `make grade` 中的 `pte printout` 测试，即完成本实验。

实验步骤

1. 声明函数：在 `kernel/defs.h` 中声明 `vmprint()` 函数，确保其可在 `exec.c` 中被调用：

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

2. 实现页表打印函数：在 `kernel/vm.c` 中编写 `vmprint()` 及辅助函数 `printwalk()`，代码如下：

```

// 递归遍历并打印页表内容，depth 表示当前页表层级（用于缩进）
void printwalk(pagetable_t pagetable, int depth)
{
    // 遍历页表中的 512 个页表项（RISC-V 页表每项对应 9 位虚拟地址，2^9=512）
    for (int i = 0; i < 512; i++)
    {
        pte_t pte = pagetable[i];
        if (pte & PTE_V) // 检查页表项是否有效（PTE_V 为有效位）
        {
            // 根据当前层级打印缩进（每级缩进".. "）
            for (int j = 0; j < depth; j++)
            {
                printf("..");
                if (j != depth - 1)
                    printf(" ");
            }
            // 打印页表项索引、页表项值（pte）及对应的物理地址（pa）
            uint64 child = PTE2PA(pte); // 将页表项转换为物理地址
            printf("%d: pte %p pa %p\n", i, pte, child);

            // 若页表项不指向物理页（无 R/W/X 权限，即指向下级页表），则递归遍历
            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0)
            {
                printwalk((pagetable_t)child, depth + 1);
            }
        }
    }
}

// 页表打印入口函数
void vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable); // 打印页表起始地址
    printwalk(pagetable, 1); // 从第一层开始递归打印
}

```

3. 触发打印：在 exec.c 中 return argc 前添加代码，使第一个进程（pid==1）加载时打印其页表：

```

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

```

4. 编译运行：执行 make qemu，启动后会输出第一个进程的页表结构，格

式符合预期。

```
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
$
```

运行评分程序结果如下：

```
lh01@lh:~/xv6-labs-2020$ ./grade-lab-pgtbl printout
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (1.5s)
```

实验中遇到的问题和解决方法

本实验难度较低，核心是利用递归遍历多级页表。基于操作系统理论中对页表层次结构的理解，容易想到通过递归处理三级页表（xv6 采用三级页表）。实现过程中未遇到明显问题，主要需注意页表项有效位（PTE_V）的判断及物理地址转换的正确使用。

实验心得

通过编写页表遍历代码，我加深了对多级页表（尤其是 xv6 三级页表）层次结构的理解，清晰认识到页表项如何通过索引逐级映射到物理内存。同时，对页表项中的标志位（如 PTE_V 表示有效、PTE_R/W/X 表示权限）的作用有了更具体的认知。

在代码设计上，掌握了利用递归处理嵌套数据结构（如多级页表）的方法，体会到递归在简化层级遍历逻辑中的优势。参考 xv6 相关文档后，进一步明确了三级页表中虚拟地址的拆分方式（每级 9 位索引 + 12 位页内偏移），为后续页表修改实验奠定了基础。

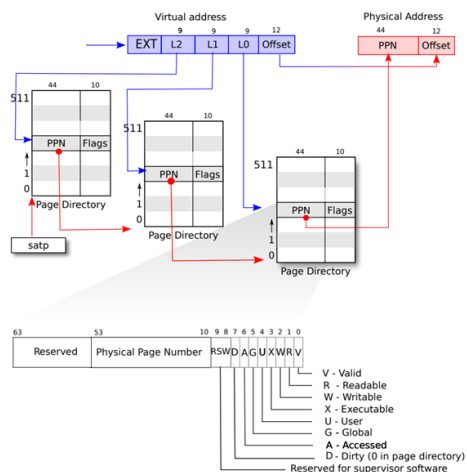


Figure 3.2: RISC-V address translation details.

A kernel page table per process (hard)

实验目的

修改内核，使每个进程在 `kernel` 模式下执行时使用自己的内核页表副本。具体需修改 `struct proc` 为每个进程维护一个内核页表，并调整调度程序，确保切换进程时同步切换内核页表。初始状态下，每个进程的内核页表需与现有全局内核页表完全一致。若 `usertests` 程序能正确运行（输出“ALL TESTS PASSED”），则通过本实验。

实验步骤

1. 扩展进程结构体：在 `kernel/proc.h` 的 `struct proc` 中添加内核页表字段，用于存储每个进程的内核页表：

```
struct proc {
//其他字段.....
    pagetable_t kernelpgtbl; // 新增：进程专属的内核页表
};
```

并在 `defs.h` 内添加各类函数声明

```
// add new function for kernel pg in each process
void      kvmmap_with_certain_page(pagetable_t pg, uint64 va, uint64 pa,
uint64 sz, int perm);
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);
```

2. 重构内核页表初始化函数

在 kernel/vm.c 中修改 kvminit 相关函数，抽离初始化逻辑为 kvm_init_one（创建单个内核页表），并让 kvminit 调用该函数初始化全局内核页表：

```
// 初始化一个内核页表（与全局内核页表内容一致）
pagetable_t kvm_init_one()
{
    pagetable_t newpg = uvmcreate(); // 创建新页表

    // 映射各类硬件设备和内核区域（与原全局内核页表映射一致）
    kvmmap_with_certain_page(newpg, UART0, UART0, PGSIZE, PTE_R | PTE_W); //
UART 寄存器
    kvmmap_with_certain_page(newpg, VIRTIO0, VIRTIO0, PGSIZE, PTE_R |
PTE_W); // virtio 磁盘接口
    kvmmap_with_certain_page(newpg, CLINT, CLINT, 0x10000, PTE_R | PTE_W); //
CLINT
    kvmmap_with_certain_page(newpg, PLIC, PLIC, 0x400000, PTE_R | PTE_W); //
PLIC
    kvmmap_with_certain_page(newpg, KERNBASE, KERNBASE, (uint64)etext -
KERNBASE, PTE_R | PTE_X); // 内核代码段
    kvmmap_with_certain_page(newpg, (uint64)etext, (uint64)etext, PHYSTOP -
(uint64)etext, PTE_R | PTE_W); // 内核数据段
    kvmmap_with_certain_page(newpg, TRAMPOLINE, (uint64)trampoline, PGSIZE,
PTE_R | PTE_X); // 陷阱处理 trampoline

    return newpg;
}
void
kvminit()
{
    kernel_pagetable = kvm_init_one();
}
```

3. 调整内核栈初始化位置：在 kernel/proc.c 的 procinit 函数中，注释原内核栈初始化代码（不再使用全局内核页表映射内核栈）：

// 注释掉以下代码：

```
//char *pa = kalloc();
//if(pa == 0)
//    panic("kalloc");
//uint64 va = KSTACK((int) (p - proc));
```

```
//kvmmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
//p->kstack = va;
```

在 `allocproc` 函数（创建进程时）中，为进程分配内核页表并初始化内核栈（使用进程专属内核页表）：

```
// 分配进程专属内核页表
p->kernelpgtbl = kvm_init_one();
if (p->kernelpgtbl == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
}

// 为进程分配内核栈，并映射到其专属内核页表
char *pa = kalloc(); // 分配物理页
if (pa == 0)
    panic("kalloc");
uint64 va = KSTACK((int)(p - proc)); // 内核栈虚拟地址
kvmmmap_with_certain_page(p->kernelpgtbl, va, (uint64)pa, PGSIZE, PTE_R |
PTE_W); // 映射到专属内核页表
p->kstack = va; // 记录内核栈虚拟地址
```

4. 释放进程资源时清理内核页表和栈

在 `kernel/proc.c` 的 `freeproc` 函数中，添加释放内核页表和内核栈的逻辑：

```
static void
freeproc(struct proc *p)
{
    if(p->trapframe)
        kfree((void*)p->trapframe);
    p->trapframe = 0;
    // free kernel stack in process
    // void *kstack_pa = (void*)kvmpa(p->kstack);
    // kfree(kstack_pa);
    // p->kstack = 0;
    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);
}

}

```

5. 修改虚拟地址到物理地址的映射函数

在 kernel/vm.c 的 kvmpa 函数中，使用当前进程的内核页表（而非全局内核页表）查找虚拟地址对应的物理地址：

```

uint64
kvmpa(uint64 va)
{
    pte_t *pte;
    uint64 pa;

    // 从当前进程的内核页表中查找页表项
    pte = walk(myproc()->kernelpgtbl, va, 0);
    if (pte == 0)
        panic("kvmpa");
    if ((*pte & PTE_V) == 0)
        panic("kvmpa: not mapped");
    pa = PTE2PA(*pte);
    return pa;
}

```

验证实验结果

执行 make qemu 启动 xv6，运行 usertests 命令，若输出 “ALL TESTS PASSED”，则实验成功。

```

sepc=0x0000000000002188 stval=0x0000000000000000
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitinput: OK
test input: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED

```

实验中遇到的问题和解决方法

本实验难度较大，核心挑战在于多个函数的修改与关联，需理解 xv6 内核的工作流程以确定函数的位置和依赖关系。例如，`kvm_init_one` 需准确复制全局内核页表的映射逻辑，`allocproc` 和 `freeproc` 需正确关联内核页表与进程生命周期，避免内存泄漏或页表混乱。初期因对内核流程理解不足，难以将各函数串联，最终通过参考 b 站指导视频理清逻辑，完成代码实现。

实验心得

通过本次实验，我深入理解了操作系统中内核页表的管理机制，特别是进程专属内核页表的创建、映射、切换与释放流程。实验提升了编写和调试复杂内核代码的能力，让我对“进程隔离”和“内存虚拟化”有了更具体的认知。解决问题时，通过参考系统自带函数的实现（如模仿页表遍历逻辑）和反复测试，逐步排查错误，这种“模仿 - 验证”的方式对处理复杂内核问题非常有效。这些经验为后续深入学习操作系统内核开发奠定了基础。

Simplify copyin/copyinstr (hard)

实验目的

将 `kernel/vm.c` 中 `copyin` 的实现替换为对 `kernel/vmcopyin.c` 中 `copyin_new` 的调用；对 `copyinstr` 和 `copyinstr_new` 执行相同操作。通过为每个进程的内核页表添加用户地址映射，确保 `copyin_new` 和 `copyinstr_new` 正常工作。若 `usertests` 正确运行且 `make grade` 所有测试通过，则完成本实验。

实验步骤

本实验的核心是通过将用户地址映射添加到进程的内核页表中，使内核能直接访问用户空间，从而简化 `copyin` 和 `copyinstr` 的实现。具体步骤如下：

1. 声明辅助函数

在 `kernel/defs.h` 中声明 `uvm2kvm` 函数（用于将用户页表映射复制到内核页表）：

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

2. 实现 `uvm2kvm` 函数

在 `kernel/vm.c` 中实现该函数，将用户页表中`[from, to)`范围的映射复制到内核页表，并清除 `PTE_U` 标志（内核访问无需用户权限），同时限制映射范围不超过 `PLIC`（避免冲突）：

```
void uvm2kvm(pagetable_t userpgtbl, pagetable_t kernelpgtbl, uint64 from,
uint64 to)
{
    if (from > PLIC) // PLIC limit 确保映射范围不超过 PLIC（硬件设备地址）

        panic("uvm2kvm: from larger than PLIC");
    from = PGROUNDDOWN(from); // align 从页对齐的地址开始
    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; // 复制用户页表项内容
        *pte_kernel &= ~PTE_U; // 清除用户标志（内核访问无需 PTE_U）
    }
}
```

3. 替换 `copyin` 和 `copyinstr` 的实现

在 `kernel/vm.c` 中，将原 `copyin` 和 `copyinstr` 的实现改为调用 `copyin_new` 和 `copyinstr_new`：

```
// 替换 copyin 为 copyin_new
int copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
    return copyin_new(pagetable, dst, srcva, len);
}
```

```
// 替换 copyinstr 为 copyinstr_new
int copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
    return copyinstr_new(pagetable, dst, srcva, max);
}
```

4. 同步用户内存映射到内核页表

由于 `fork()`、`exec()`、`userinit()`、`growproc()` 等系统调用会修改用户内存映射，需在这些操作后调用 `uvm2kvm` 更新内核页表中的用户地址映射：

`fork()`：子进程复制用户内存后，同步到子进程的内核页表：

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

`exec()`：加载新程序后，同步用户映射到当前进程的内核页表：

```
// ... 加载程序逻辑 ...
if(p->pid==1) vmprint(p->pagetable);
uvm2kvm(p->pagetable, p->kernelpgtbl, 0, p->sz);
return argc; // this ends up in a0, the first argument to main(argc, argv)
```

`userinit()`：初始化第一个进程时，同步初始用户映射：

```
uvminit(p->pagetable, initcode, sizeof(initcode));
p->sz = PGSIZE;
uvm2kvm(p->pagetable, p->kernelpgtbl, 0, p->sz); // copy from user to kernel
```

`growproc()`：调整进程内存大小后，同步新增 / 减少的映射：

```
// ... 调整内存大小逻辑 ...
uvm2kvm(p->pagetable, p->kernelpgtbl, sz - n, sz);
p->sz = sz;
return 0;
```

执行 `make grade`，若所有测试（包括页表相关测试）通过，则实验成功。

实验中遇到的问题和解决方法

本实验难度较大，核心挑战在于确保用户内存映射与内核页表的同步，以及理解 `uvm2kvm` 函数的边界限制（如 `PLIC` 范围）。初期因未在所有修改用户内存的系统调用中同步内核页表，导致 `copyin_new` 访问用户地址时出错，最终在指导视频的帮助下，理清了各系统调用与页表同步的关联，完成代码调试。

实验心得

通过本实验，我深入理解了内核与用户空间数据交互的机制。uvm2kvm 函数通过将用户页表映射复制到内核页表，使内核无需切换页表即可直接访问用户地址，简化了 copyin 和 copyinstr 的实现。这一过程让我认识到页表同步对内核正确性的重要性 —— 任何修改用户内存的操作都必须同步更新内核页表中的映射。同时，通过调试复杂的页表同步逻辑，我的内核代码调试能力和对内存虚拟化的理解得到了显著提升。

Lab3 的实验整体测试(./grade-lab-pgtbl)截图：

在项目目录下添加 time.txt 文件，文件输入完成本实验用时（小时数，单个整数），对于本实验的测试还需要另外补充 answers-pgtbl.txt，然后运行 ./grade-lab- syscall 进行整体测试，结果如下：

```
lh01@lh:~/xv6-labs-2020$ ./grade-lab-pgtbl
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin == count copyin: OK (1.0s)
== Test usertests == (109.3s)
== 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
```

测试无误后上传至 GitHub 对应分支：

```
lh01@lh:~/xv6-labs-2020$ git push github pgtbl:pgtbl
Enumerating objects: 17, done.
Counting objects: 100% (17/17), done.
Delta compression using up to 16 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (10/10), 4.45 KiB | 2.22 MiB/s, done.
Total 10 (delta 7), reused 0 (delta 0)
remote: Resolving deltas: 100% (7/7), completed with 7 local objects.
To https://github.com/Effulgence12/OS-design-XV6-Operation-System.git
773d70b..1413c2d pgtbl -> pgtbl
```

Lab4:Traps

开始 Lab4 时，需执行以下命令切换到对应的实验分支并准备实验环境：

```
git fetch    # 从远程仓库拉取最新分支信息
```

```
git checkout traps  # 切换到 traps 分支，获取实验相关资源
```

```
make clean  # 清除之前的编译产物，避免干扰本次实验
```

本实验聚焦于陷阱（Traps）机制在系统调用实现中的应用，主要内容包括：

通过堆栈操作进行热身练习，熟悉陷阱处理中堆栈的使用方式。

实现用户级别的陷阱处理示例，深入理解从用户态到内核态的陷阱触发、处理及返回流程。

陷阱机制是操作系统中实现系统调用、异常处理和中断响应的核心基础，通过本实验可进一步掌握用户态与内核态之间的切换原理及相关底层操作。

RISC-V assembly (easy)

实验目的

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

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

实验步骤

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。

```
balloc: first 637 blocks have been allocated
balloc: write bitmap block at sector 45
lh01@lh:~/xv6-labs-2020$ make fs.img
make: 'fs.img' is up to date.
```

在 call.asm 中可以找到函数 g、f 和 main 的代码。

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

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

void main(void) {
1c: 1141          addi sp,sp,-16
1e: e406          sd ra,8(sp)
20: e022          sd s0,0(sp)
22: 0800          addi s0,sp,16
printf("%d %d\n", f(8)+1, 13);
24: 4635          li a2,13
26: 45b1          li a1,12
28: 00000517      auipc a0,0x0
2c: 7b050513      addi a0,a0,1968 # 7d8 <malloc+0xea>
30: 00000097      auipc ra,0x0
34: 60080e7       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。以此类推，查看 call.asm 文件中的 main 函数可知，在 main 调用 printf 时，由寄存器 a2 保存 13。

```
22: 0800          addi s0,sp,16
printf("%d %d\n", f(8)+1, 13);
24: 4635          li a2,13
26: 45b1          li a1,12
```

(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)

在 main 的汇编代码中，没有显式调用函数 f 和 g 的汇编指令。这是因为编译器对函数进行了内联处理，函数 g 被内联到函数 f 中，而函数 f 又被内联到 main 函数中，因此无需生成单独的函数调用指令。查看 main 函数可以发现，printf 中包含了一个对 f 的调用。但是对应的汇编代码却是直接将 f(8)+1 替换为 12。这就说明编译器对这个函数调用进行了优化。

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

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

从上述相关汇编代码可知，跳转到 $ra + 1536$ 的位置，此时 ra 的值即为程序计数器（ pc ）的值。结合代码可知，此时 pc 的值为 $0x30$ ，因此 $printf$ 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，跳转后 ra 寄存器的值为跳转前的程序计数器（ pc ）值加 4。在 main 中执行 jalr 到 printf 时，跳转前的 pc 值为 $0x34$ ，因此 ra 寄存器的值为 $0x34 + 4 = 0x38$ 。

(5) Run the following code.

```
unsigned int i = 0x00646c72;

printf("H% x Wo% s", 57616, &i);
```

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

运行下面的代码，回答问题

```
unsigned int i = 0x00646c72;

printf("H% x Wo% s", 57616, &i);
```

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

输出为 “He110 World”。57616 转换为十六进制是 $e110$ ； $\&i$ 所指向的内存中，由于 RISC-V 是小端序，数据存储为 $0x72$ 、 $0x6c$ 、 $0x64$ 、 $0x00$ ，对应字符串 “rld”，因此整体输出为 “He110 World”。

如果 RISC-V 是大端序，为了得到相同的输出，需要将 i 设置为 $0x726c6400$ 。因为大端序中数据高位存于低地址，此时内存中会存储为 $0x72$ 、 $0x6c$ 、 $0x64$ 、 $0x00$ ，同样对应字符串 “rld”。

57616 不需要修改，因为 $\%x$ 以十六进制形式输出整数，57616 的十六进制表示是 $e110$ ，与端序无关。

(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=' 之后会打印一个不确定的 “随机” 值。因为 `printf` 函数需要两个整数参数，但代码中只提供了一个参数 (3)。在 RISC-V 中，函数参数通过寄存器传递，第一个参数用 `a0`，第二个参数用 `a1`。此处只设置了 `a0` 的值为 3，而 `a1` 寄存器的值未被显式设置，会保留调用 `printf` 之前在该寄存器中的值，因此打印出的是一个不确定的值。

实验中遇到的问题和解决方法

本次实验的难点在于对汇编代码的阅读和理解。通过查阅 RISC-V 汇编语言的基础语法资料，了解寄存器的用途、指令的功能以及函数调用时参数传递的方式等知识后，逐步解决了实验中遇到的问题。

实验心得

通过这次实验，我对 RISC-V 汇编语言有了一定的理解，尤其是函数调用和参数传递的过程。了解到 RISC-V 中函数参数通过 `a0` 到 `a7` 寄存器传递，这种方式相比通过内存传递参数，由于寄存器访问速度更快，能够提高程序的运行效率。同时，对编译器的内联优化、端序对数据存储的影响以及函数参数数量不匹配时的现象等有了更清晰的认识，加深了对计算机底层运行机制的理解。

Backtrace (moderate)

实验目的

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 `btest`，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

1. 添加函数原型与辅助函数

在 kernel/defs.h 中，于// printf.c 注释下添加 backtrace()函数的原型，确保其可在其他文件中被引用：

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

在 kernel/riscv.h 中添加获取帧指针（frame pointer）的辅助函数 r_fp()，GCC 编译器将当前函数的帧指针保存在 s0 寄存器中：

```
// 用于获取当前帧指针（s0 寄存器的值）
static inline uint64
r_fp()
{
    uint64 x;
    asm volatile("mv %0, s0" : "=r"(x)); // 将 s0 的值移动到 x 中
    return x;
}
```

2. 在 sys_sleep 中调用 backtrace

在 kernel/sysproc.c 的 sys_sleep()函数中插入 backtrace()调用，以便在 sleep 系统调用执行时触发回溯：

```
sys_sleep(void)
{
    int n;
    uint ticks0;
    backtrace(); // 插入回溯调用

    // 其他代码.....
}
```

3. 实现 backtrace 函数

在 kernel/printf.c 中编写 backtrace()函数，通过遍历栈帧获取并打印函数返回地址：

```
void backtrace(void)
{
    printf("backtrace:\n");
    uint64 fp = r_fp(); // 获取当前帧指针（s0 寄存器的值）

    // 循环遍历栈帧，只要帧指针在当前页面内就继续
    while (PGROUNDDOWN(fp) < PGROUNDUP(fp))
    {
        // 帧指针-8 的位置存储返回地址（函数调用后要返回的地址）
    }
}
```

```

    printf("%p\n", *(uint64 *)(fp - 8));
    // 帧指针-16 的位置存储上一个栈帧的帧指针，更新 fp 以继续遍历
    fp = *(uint64 *)(fp - 16);
}
}

```

4. 编译运行与验证

执行 `make qemu` 启动 `xv6`，运行 `bttest` 命令，若输出函数调用栈的返回地址列表。

```

init: starting sh
$ bttest
backtrace:
0x0000000080002ce4
0x0000000080002bbe
0x00000000800028a8

```

退出 QEMU，使用 `addr2line` 将输出的地址转换为具体的源代码行：

```

lh01@lh:~/xv6-labs-2020$ addr2line -e kernel/kernel
0x0000000080002ce4
/home/lh01/xv6-labs-2020/kernel/sysproc.c:63
0x0000000080002bbe
/home/lh01/xv6-labs-2020/kernel/syscall.c:140
0x00000000800028a8
/home/lh01/xv6-labs-2020/kernel/trap.c:76

```

实验中遇到的问题和解决方法

本次实验的核心在于理解栈帧结构中返回地址和前帧指针的存储位置。通过查阅资料得知：在 RISC-V 的栈帧布局中，当前帧指针（`fp`）的 `fp-8` 位置存储当前函数的返回地址，`fp-16` 位置存储上一个栈帧的帧指针。明确这两个偏移的含义后，成功实现了栈帧的遍历逻辑。

实验心得

通过本实验，我深入理解了函数调用栈和栈帧的底层结构。`backtrace` 函数通过帧指针（`s0` 寄存器）遍历栈帧，依次获取每个函数的返回地址，从而还原函数调用历史，这对调试非常有价值。实验过程中，我清晰认识到栈帧中返回地址和前帧指针的固定偏移规律，强化了对计算机系统栈机制的理解，也提升了底层调试工具的实现能力。

Alarm (hard)

实验目的

在本练习中，您将向 xv6 添加一个定期警报功能，使进程在使用 CPU 时间时定期触发警报。这对于计算受限的进程（如希望限制 CPU 消耗的进程）或需要定期执行操作的进程很有用。更一般地，您将实现用户级中断 / 故障处理程序，类似机制可用于处理应用程序中的页面错误等场景。若解决方案通过 alarmtest 和 usertests，则为正确。

实验步骤

1. 添加测试程序编译配置

在 Makefile 的 UPROGS 中添加 \$U/_alarmtest\，确保 alarmtest 作为用户程序被编译。

2. 声明 sigalarm 和 sigreturn 系统调用

在 user/user.h 中添加函数原型，供用户程序调用：

```
int sigalarm(int ticks, void (*handler)()); // 设置警报: ticks 后触发 handler
int sigreturn(void); // 从警报处理程序返回, 恢复原执行状态
```

在 user/usys.pl 中添加系统调用存根，生成用户态到内核态的转换代码：

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

3. 分配系统调用编号并注册处理函数

在 kernel/syscall.h 中为两个系统调用分配编号：

```
#define SYS_sigalarm 22 // sigalarm 系统调用编号
#define SYS_sigreturn 23 // sigreturn 系统调用编号
```

在 kernel/syscall.c 中声明并注册系统调用处理函数：

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

在 syscalls 数组中添加条目

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

4. 扩展进程结构体存储警报信息

在 kernel/proc.h 的 struct proc 中添加新字段，存储警报配置、计数器及陷阱帧：

```
uint64 handler_va;           // 警报处理函数的用户态虚拟地址

int alarm_interval;          // 警报触发间隔（时钟滴答数）
int passed_ticks;            // 已过去的滴答数（用于计数）
struct trapframe saved_trapframe; // 保存的陷阱帧（用于恢复执行）
int have_return;             // 标记是否已从警报处理返回（用于重新激活计数）
```

5. 处理时钟中断触发警报

硬件时钟中断在 kernel/trap.c 的 usertrap() 中处理，修改该函数以实现定期触发警报：

```
void
usertrap(void)
{
    // 原有代码.....
    if (which_dev == 2) { // 时钟设备中断（每滴答触发一次）
        struct proc *p = myproc();
        // 若已设置警报间隔且处于可触发状态
        if (p->alarm_interval > 0 && p->have_return) {
            p->passed_ticks++; // 递增滴答计数器
            // 当已过去的滴答数达到警报间隔时，触发处理函数
            if (p->passed_ticks >= p->alarm_interval) {
                // 保存当前陷阱帧（用于后续恢复）
                p->savetrapframe = *p->trapframe;
                // 修改陷阱帧的程序计数器（epc），指向警报处理函数
                p->trapframe->epc = p->handler_va;
                p->passed_ticks = 0; // 重置计数器
                p->have_return = 0; // 标记未从处理函数返回（暂时禁用计数）
            }
        }
        yield(); // 时钟中断后切换进程
    }
    usertrapret();
}
```

6. 实现 sigalarm 和 sigreturn 系统调用

在 kernel/sysproc.c 中实现 sys_sigalarm，用于设置警报参数：

```
uint64 sys_sigalarm(void) {
    int ticks;
    uint64 handler_va;
    struct proc *p = myproc();

    // 从用户态获取参数：ticks（间隔）和 handler（处理函数地址）
```

```

if (argint(0, &ticks) < 0 || argaddr(1, &handler_va) < 0)
    return -1;

p->alarm_interval = ticks;    // 设置警报间隔
p->handler_va = handler_va;    // 保存处理函数地址
p->passed_ticks = 0;          // 重置计数器
p->have_return = 1;           // 标记可触发警报
return 0;
}

```

实现 `sys_sigreturn`，用于从警报处理程序返回并恢复原执行状态：

```

uint64 sys_sigreturn(void) {
    struct proc *p = myproc();
    // 恢复保存的陷阱帧（恢复寄存器、程序计数器等）
    *p->trapframe = p->savetrapframe;
    p->have_return = 1; // 重新激活警报计数
    return p->trapframe->a0; // 返回原执行状态的返回值
}

```

7. 测试验证

执行 `make qemu` 启动 `xv6`，运行 `alarmtest`，若能定期打印 `"alarm!"` 且通过所有测试（如 `"test1 passed"`），则警报功能正常。

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
....alarm!
....alarm!
....alarm!
....alarm!
....alarm!
....alarm!
....alarm!
....alarm!
....alarm!
....alarm!
test1 passed
test2 start
.....alarm!
test2 passed

```

运行 `usertests`，确保修改未影响内核其他功能，所有测试通过。

```
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

实验中遇到的问题和解决方法

本次实验的整体框架与 Lab2 中添加系统调用的实验类似，核心是向 xv6 添加定期警报功能，但具体函数的实现细节存在一定难度。由于需要结合时钟中断处理、进程状态保存与恢复等机制，且涉及多个库函数的调用，对于初次接触 xv6 内核代码的学习者而言，理解各部分逻辑的关联存在挑战。

解决过程中，主要通过以下方式突破难点：

参考 xv6 现有系统调用（如 `sys_sleep`）的实现流程，梳理 `sigalarm` 和 `sigreturn` 的参数传递、内核态处理逻辑；结合教材中关于陷阱帧（`trapframe`）的描述，理解如何通过保存和恢复陷阱帧实现进程状态的无缝切换；逐行分析时钟中断处理函数（`usertrap`）的逻辑，明确滴答计数、警报触发条件与进程调度的关联。

通过逐步拆解问题、理解每一行代码的作用，最终完成了功能实现。

实验心得

本次实验让我深入理解了操作系统中闹钟信号机制的底层实现原理。通过编写 `sys_sigalarm`（设置定时警报）和 `sys_sigreturn`（从警报处理程序返回）系统调用，以及修改时钟中断处理逻辑，我清晰掌握了以下关键点：

定时警报的核心是通过时钟中断计数，当达到设定间隔时，强制进程跳转到用户定义的处理函数：

陷阱帧在保存和恢复进程状态中起到关键作用，确保从警报处理返回后，进程能继续正常执行；

系统调用与中断处理的协同工作：用户态通过系统调用设置参数，内核态通过中断监控并触发事件，最终通过另一系统调用恢复状态。

这一过程不仅加深了我对系统调用、中断处理、进程管理等操作系统核心概念的理解，还提升了编写和调试底层代码的能力，为后续学习更复杂的用户级故障处理机制奠定了基础。

Lab4 的实验整体测试(./grade-lab- traps)截图：

在项目目录下添加 `time.txt` 文件，文件输入完成本实验用时（小时数，单个整数），对于本实验的测试还需要另外补充 `answers-traps.txt`，然后运行 `./grade-lab-traps` 进行整体测试，结果如下：

```
● lh01@lh:~/xv6-labs-2020$ ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (1.3s)
== Test running alarmtest == (3.6s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests == usertests: OK (78.7s)
== Test time ==
time: OK
Score: 85/85
```

测试无误后上传至 GitHub 对应分支：

```
● lh01@lh:~/xv6-labs-2020$ git push github traps:traps
Enumerating objects: 53, done.
Counting objects: 100% (53/53), done.
Delta compression using up to 16 threads
Compressing objects: 100% (29/29), done.
Writing objects: 100% (38/38), 16.53 KiB | 8.27 MiB/s, done.
Total 38 (delta 23), reused 16 (delta 6)
remote: Resolving deltas: 100% (23/23), completed with 14 local objects.
remote:
remote: Create a pull request for 'traps' on GitHub by visiting:
remote:   https://github.com/Effulgence12/OS-design-XV6-Operation-System/pull/new/traps
remote:
To https://github.com/Effulgence12/OS-design-XV6-Operation-System.git
* [new branch]      traps -> traps
```


Lab5: Lazy allocation

操作系统可以利用页表硬件实现用户空间堆内存的延迟分配（**lazy allocation**）。在 **xv6** 中，应用程序通过 **sbrk()** 系统调用来请求堆内存。原始内核中，**sbrk()** 会立即分配物理内存并将其映射到进程的虚拟地址空间。但对于大型内存请求（例如 **GB** 级），这种立即分配方式可能耗时较长；此外，有些程序会分配超出实际使用量的内存（如稀疏数组），或提前分配内存但暂不使用。

延迟分配机制可优化上述场景：**sbrk()** 不再立即分配物理内存，仅记录已分配的用户地址范围，并将这些地址在用户页表中标记为无效。当进程首次访问某块延迟分配的内存时，**CPU** 会触发页故障（**page fault**），内核此时才为该页分配物理内存、清零并建立映射。本实验需为 **xv6** 添加这一延迟分配功能。

开始实验步骤-切换到 **lazy** 分支并准备实验环境，执行以下命令：

```
git fetch
```

```
git checkout lazy
```

```
make clean
```

相关参考与文件-阅读 **xv6** 书籍第 4 章（尤其是 4.6 节），理解页表与页故障处理机制。

主要修改文件：

kernel/trap.c：处理页故障中断，实现延迟分配的核心逻辑。

kernel/vm.c：修改内存映射相关函数（如页表项设置）。

kernel/sysproc.c：调整 **sbrk()** 系统调用，使其不再立即分配物理内存。

Eliminate allocation from sbrk() (easy)

实验目的

你的首要任务是删除 **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 中修改 sys_sbrk()函数，删除原本调用的 growproc()函数，仅增加进程的大小（myproc()->sz），代码如下：

```
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;    // 直接增加进程的大小，不分配物理内存
    return addr;
}
```

修改后，执行 make qemu 启动 xv6，输入 echo hi 等命令进行测试，结果符合预期（进程能正常报告内存大小，尽管尚未实际分配物理内存）。

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

实验中遇到的问题和解决方法

本实验难度较低，按照指导提示找到 sys_sbrk()函数，删除对 growproc()的调用并直接修改 myproc()->sz 即可。整个过程未遇到明显问题。

实验心得

通过本次实验，我初步理解了延迟分配的核心思想：sbrk()系统调用原本会立即分配物理内存并建立映射，而修改后仅调整进程的内存大小记录，不实际分配内存。这一修改凸显了“进程逻辑内存大小”与“物理内存分配”的分离，为后续实现完整的延迟分配（通过页故障动态分配物理内存）奠定了基础。

Lazy allocation (moderate)

实验目的

修改 `trap.c` 中的代码以响应来自用户空间的页面错误，方法是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

1. 处理页故障中断

在 `kernel/trap.c` 的 `usertrap()` 函数中，添加对页故障（中断码 13 或 15）的处理逻辑。当发生页故障时，从 `stval` 寄存器获取故障虚拟地址，检查其合法性后分配物理内存并建立映射：

```
// 新增：处理页故障（中断码 13：加载页故障；15：存储页故障）
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;
}
}
}

```

2. 修改 uvmunmap 避免无效页表项导致的崩溃

kernel/vm.c 的 uvmunmap() 函数用于释放内存映射，当遇到未映射的页表项时，原代码会 panic。由于延迟分配中存在未映射的合法地址，需注释两行 panic 并 continue 跳过无效项：

```

for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0) // 若页表项不存在，跳过（原逻辑为 panic）
        //panic("uvmunmap: walk");
        continue;
    if((*pte & PTE_V) == 0) // 若页表项无效，跳过（原逻辑为 panic）
        //panic("uvmunmap: not mapped");
        continue;
    // ... 原有释放物理页和页表项的逻辑 ...
}

```

3. 完善在 kernel/sysproc.c 中 sys_sbrk() 函数

```

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

    if(argint(0, &n) < 0)
        return -1;
    struct proc *p = myproc();
    addr = p->sz;

    if (n >= 0 && addr + n >= addr)
        p->sz += n;
    else if (n < 0 && addr + n >= PGROUNDUP(p->trapframe->sp))
    {
        p->sz = uvmdealloc(p->pagetable, p->sz, p->sz + n);
    }
    else
        return -1;
    return addr;
}

```

```
}
```

4. 测试验证：执行 `make qemu` 启动 `xv6`，输入 `echo hi` 命令。若命令正常执行，则说明延迟分配机制生效：进程通过 `sbrk()` 扩展内存时未立即分配物理页，首次访问时触发页故障，内核动态分配并映射物理页，进程继续执行。

```
xv6 kernel is booting

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

实验中遇到的问题和解决方法

物理内存分配失败：`kalloc()` 可能返回 0（内存不足），此时需妥善处理（如终止进程），避免内核崩溃。实验中通过判断 `pa == 0` 并设置 `p->killed = 1` 解决。

无效地址访问：若页故障地址不在合法范围内（如超出进程 `sz` 或在栈上方），需终止进程以防止非法内存访问。通过添加地址合法性检查

（`PGROUNDDOWN(p->trapframe->sp) >= fault_va || fault_va >= p->sz`）解决。

`uvmunmap` 崩溃：释放内存时遇到未映射的页表项会触发 `panic`，通过注释 `panic` 并 `continue` 跳过无效项解决。

实验心得

本次实验实现了延迟分配的核心机制，让我深入理解了页故障处理在内存管理中的关键作用。通过捕获页故障动态分配物理内存，操作系统可按需为进程提供内存，提高了内存使用效率，尤其适合处理大型或稀疏的内存请求。

实验过程中，我清晰认识到虚拟内存与物理内存的分离逻辑：进程的虚拟地址空间可通过 `sbrk()` 快速扩展，但物理内存仅在首次访问时分配，这一设计极大优化了系统性能。这些经验为理解更复杂的内存管理策略（如交换分区、内存压缩）奠定了基础。

Lazytests and Usertests (moderate)

实验目的

我们为您提供了一个 lazytests，这是一个 xv6 用户程序，它测试一些可能会给您的惰性内存分配器带来压力的特定情况。修改内核代码，使所有 lazytests 和 usertests 都通过。

实验步骤

在完成 “Lazy allocation” 实验的基础上，进行测试：

执行 make qemu 启动 xv6 操作系统。

在 xv6 终端中输入 lazytests 命令，验证延迟分配在各种压力场景下的正确性。输入 usertests 命令，运行 xv6 的通用用户程序测试集，确保延迟分配的修改未影响内核其他功能。

```
xv6 kernel is booting OK
hart 1 starting test pipe1: OK
hart 2 starting test preempt: kill... wait... OK
init: starting sh test exitwait: OK
$ lazytests test rmdot: OK
lazytests starting test fourteen: OK
running test lazy alloc test bigfile: OK
test lazy alloc: OK test dirfile: OK
running test lazy unmap test iref: OK
test lazy unmap: OK test forktest: OK
running test out of memory test bigdir: OK
test out of memory: OK kalloc == 0
ALL TESTS PASSED ALL TESTS PASSED
```

若两个测试程序均输出 “ALL TESTS PASSED”，表示实验成功。

实验中遇到的问题和解决方法

在本 lab 整体测试时遇到报错：

make: 'kernel/kernel' is up to date.

== Test pte printout == Failed to connect to QEMU; output:

*** Now run 'gdb' in another window.

qemu-system-riscv64: -gdb tcp::26000: Failed to find an available port: Address already in use

make: *** [Makefile:289: qemu-gdb] Error 1

并且运行'killall qemu' or 'killall qemu.real'无效果。

经过查阅资料及咨询大模型得知这个错误提示表明 QEMU 无法启动 GDB 调试服务，因为默认的调试端口（26000）已被其他进程占用，解决方法如下：

1. 终止占用端口 26000 的进程（推荐）

查找占用 26000 端口的进程 ID（PID）

```
sudo lsof -i :26000
```

找到占用端口的进程 ID，用 kill 命令终止：

```
kill -9 1234 # 替换为实际的 PID
```

2. 直接终止所有 QEMU 进程（简单粗暴）

如果不确定具体进程，可直接终止所有 QEMU 相关进程：

终止所有 riscv64 架构的 QEMU 进程

```
pkill -9 qemu-system-riscv64
```

若仍有残留，尝试更通用的命令

```
killall -9 qemu-system-riscv64
```

实验心得

本实验通过 lazytests 和 usertests 验证了延迟分配机制的正确性和稳定性。lazytests 针对性地测试了延迟分配在复杂场景下的表现（如反复分配与释放、越界访问等），而 usertests 则确保修改未对内核其他功能产生副作用。

这一过程让我认识到，操作系统功能的修改需经过全面测试：不仅要验证新功能本身的正确性，还要确保其与原有系统的兼容性。

Lab5 的实验整体测试(./grade-lab-lazy)截图：

添加 time.txt 文件，运行 ./grade-lab-lazy 进行整体测试，结果如下：

```

● lh01@lh:~/xv6-labs-2020$ ./grade-lab-lazy
make: 'kernel/kernel' is up to date.
== Test running lazytests == (2.7s)
== Test lazy: map ==
lazy: map: OK
== Test lazy: unmap ==
lazy: unmap: OK
== Test usertests == (97.2s)
== Test usertests: pgbug ==
usertests: pgbug: OK
== Test usertests: sbrkbugs ==
usertests: sbrkbugs: OK
== Test usertests: argptest ==
usertests: argptest: OK
== Test usertests: sbrkmuch ==
usertests: sbrkmuch: OK
== Test usertests: sbrkfail ==
usertests: sbrkfail: OK
== Test usertests: sbrkarg ==
usertests: sbrkarg: OK
== Test usertests: stacktest ==
usertests: stacktest: OK
== Test usertests: execout ==
usertests: execout: OK
== Test usertests: copyin ==
usertests: copyin: OK
== Test usertests: copyout ==

== Test usertests: createtest ==
usertests: createtest: OK
== Test usertests: openiput ==
usertests: openiput: OK
== Test usertests: exitiput ==
usertests: exitiput: OK
== Test usertests: iput ==
usertests: iput: OK
== Test usertests: mem ==
usertests: mem: OK
== Test usertests: pipe1 ==
usertests: pipe1: OK
== Test usertests: preempt ==
usertests: preempt: OK
== Test usertests: exitwait ==
usertests: exitwait: OK
== Test usertests: rmdot ==
usertests: rmdot: OK
== Test usertests: fourteen ==
usertests: fourteen: OK
== Test usertests: bigfile ==
usertests: bigfile: OK
== Test usertests: dirfile ==
usertests: dirfile: OK
== Test usertests: iref ==
usertests: iref: OK
== Test usertests: forktest ==
usertests: forktest: OK
== Test time ==
time: OK
Score: 119/119

```

测试无误后上传至 GitHub 对应分支。

```

lh01@lh:~/xv6-labs-2020$ git push github lazy:lazy
Enumerating objects: 78, done.
Counting objects: 100% (69/69), done.
Delta compression using up to 16 threads
Compressing objects: 100% (38/38), done.
Writing objects: 100% (54/54), 17.17 KiB | 5.72 MiB/s, done.
Total 54 (delta 35), reused 27 (delta 13)
remote: Resolving deltas: 100% (35/35), completed with 13 local objects.
remote:
remote: Create a pull request for 'lazy' on GitHub by visiting:
remote:   https://github.com/Effulgence12/OS-design-XV6-Operation-System/pull/new/lazy
remote:
To https://github.com/Effulgence12/OS-design-XV6-Operation-System.git
* [new branch]      lazy -> lazy

```


Lab6:Copy-on-Write Fork for xv6

虚拟内存提供了一定程度的间接性：内核可以通过将页表项（PTE）标记为无效或只读来拦截内存引用，从而触发页面错误，并且可以通过修改 PTE 来更改地址的映射关系。计算机系统中有一句名言：任何系统问题都可能通过一定程度的间接性来解决。惰性分配实验就是一个例子，本实验将探讨另一个示例——写时复制（Copy-on-Write）的 fork 机制。

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 checkout cow
```

```
make clean
```

Implement copy-on write (hard)

实验目的

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

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

本实验将通过修改 xv6 的内核代码，实现以下目标： 1.修改 uvmcopy() 函数，使其在 fork() 调用时共享父进程的物理页，而不是立即复制页面内容，从而实现内存的共享使用。 2.修改 usertrap() 函数，以处理 COW 页面错误。当检测到对只读共享页面的写操作时，动态分配新页面并复制原有内容，从而保障进程的正确性。 3.实现对物理页面的引用计数管理，以确保在页面不再被任何进程引用时，释放内存资源，防止内存泄漏。 4.修改 copyout() 函数，以确保在将数据从内核空间复制到用户空间时，正确处理 COW 页面，保障数据的一致性和正确性。

实验步骤

1. 修改 uvmcopy 实现 COW 共享

传统 fork() 中 uvmcopy 会为子进程复制父进程的物理页，COW 机制下需改为共享物理页，并将父子进程的页表项（PTE）标记为只读（清除

PTE_W), 同时用 PTE_RSW 作为 COW 标记, 增加物理页的引用计数:

```
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) { // 遍历父进程地址空间
        if ((pte = walk(old, i, 0)) == 0)
            panic("uvmcopy: pte should exist");
        if ((*pte & PTE_V) == 0)
            panic("uvmcopy: page not present");

        // 清除写权限, 设置 COW 标记 (PTE_RSW)
        *pte = ((*pte) & (~PTE_W)) | PTE_RSW;
        pa = PTE2PA(*pte); // 获取物理地址
        flags = PTE_FLAGS(*pte); // 保留原有标志 (除写权限外)

        // 子进程共享父进程的物理页, 而非复制
        if (mappages(new, i, PGSIZE, pa, flags) != 0) {
            goto err;
        }
        add_kmem_ref((void *)pa); // 增加物理页引用计数
    }
    return 0;
err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}
```

2. 处理写页错误 (usertrap 修改)

当父子进程写入 COW 共享页时, 会触发写页错误 (中断码 15)。需在 usertrap() 中捕获错误, 判断是否为 COW 页, 根据引用计数处理:

若引用计数为 1 (仅当前进程使用): 直接恢复写权限。

若引用计数 > 1 (多进程共享): 分配新物理页, 复制内容, 更新页表指向新页并恢复写权限。

```
else if (r_scause() == 15) { // 写页错误处理
    uint64 va = PGROUNDDOWN(r_stval()); // 故障虚拟地址 (页对齐)
    pte_t *pte;
```

```

struct proc *p = myproc();
// 检查地址合法性及页表项有效性
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); // 共享物理页地址
acquire_ref_lock(); // 引用计数操作加锁
uint ref = get_kmem_ref((void *)pa);

if (ref == 1) { // 仅当前进程使用，恢复写权限
    *pte = ((*pte) & (~PTE_RSW)) | PTE_W;
} else { // 多进程共享，复制页面
    char *mem = kalloc(); // 分配新物理页
    if (mem == 0) {
        p->killed = 1;
        release_ref_lock();
        goto end;
    }
    memmove(mem, (char *)pa, PGSIZE); // 复制旧页内容
    // 更新页表：指向新页，恢复写权限，清除 COW 标记
    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); // 减少旧页引用计数（实际由 kfree 内部处理）
}
release_ref_lock();
}

```

3. 维护物理页引用计数（kalloc.c 修改）

为物理页添加引用计数，kalloc()分配时初始化计数为 1；kfree()仅当计数为 0 时释放页面；add_kmem_ref 用于增加计数：

```
// 在 kmem 结构体中添加引用计数数组和锁
struct {
    struct spinlock lock;
    struct run *freelist;
    struct spinlock ref_lock; // 引用计数锁
    int *ref_count; // 物理页引用计数数组

} kmem;
```

```
// 分配物理页时初始化引用计数为 1
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; // 索引基于物理地址
    }
    release(&kmem.lock);
    if (r) memset((char *)r, 5, PGSIZE);
    return (void *)r;
}
```

```
// 获取引用计数
int get_kmem_ref(void *pa) {
    return kmem.ref_count[INDEX(pa)];
}

// 增加引用计数
void add_kmem_ref(void *pa) {
    kmem.ref_count[INDEX(pa)]++;
}
```

```
void
kfree(void *pa)
{
    if (pa == 0)
        panic("kfree");
}
```

```

    acquire(&kmem.ref_lock);
    if (--kmem.ref_count[INDEX(pa)] > 0) { // 计数减 1 后仍>0, 不释放
        release(&kmem.ref_lock);
        return;
    }
    release(&kmem.ref_lock);
    // 计数为 0, 释放页面 (原有逻辑)
    struct run *r = (struct run*)pa;
    memset(pa, 1, PGSIZE);
    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
}

```

4. 处理 copyout 中的 COW 页面

copyout 用于内核向用户空间写数据, 若目标页为 COW 页, 需触发与页错误相同的复制逻辑:

// 在 copyout 中添加 COW 处理

```

void
copyout(char *s)
{
    uint64 addrs[] = { 0x80000000LL, 0xffffffffffffffff };

    for(int ai = 0; ai < 2; ai++){
        uint64 addr = addrs[ai];

        int fd = open("README", 0);
        if(fd < 0){
            printf("open(README) failed\n");
            exit(1);
        }
        int n = read(fd, (void*)addr, 8192);
        if(n > 0){
            printf("read(fd, %p, 8192) returned %d, not -1 or 0\n", addr, n);
            exit(1);
        }
        close(fd);

        int fds[2];
        if(pipe(fds) < 0){

```

```

    printf("pipe() failed\n");
    exit(1);
}
n = write(fds[1], "x", 1);
if(n != 1){
    printf("pipe write failed\n");
    exit(1);
}
n = read(fds[0], (void*)addr, 8192);
if(n > 0){
    printf("read(pipe, %p, 8192) returned %d, not -1 or 0\n", addr, n);
    exit(1);
}
close(fds[0]);
close(fds[1]);
}
}

```

5. 测试验证

执行 `make qemu` 启动 `xv6`，运行 `cowtest` 和 `usertests`，若均输出 “ALL TESTS PASSED”，则实验成功。

```

$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED

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

```

实验中遇到的问题解决方法

本实验难度较大，核心挑战在于理解 COW 机制的细节及各模块的协同：

COW 标记与引用计数：初期对 `PTE_RSW` 作为 COW 标记的用法不清晰，通过查阅 RISC-V 手册明确其为保留位可自定义使用；引用计数的并发安全需通过锁机制保证（`ref_lock`）。**写页错误处理逻辑：**多次出现 “页面未复制却被多进程修改” 的错误，后通过严格检查引用计数和页表项标志

（`PTE_RSW`、`PTE_V` 等）解决。**copyout 适配：**最初忽略了内核向 COW 页写数据的场景，导致 `usertests` 中部分测试失败，通过在 `copyout` 中添加与页错误

相同的 COW 处理逻辑解决。最终通过参考资料和逐步调试，理清了 COW 从共享页设置、写错误触发到页面复制的完整流程。

实验心得

写时复制（COW）fork 通过延迟物理页复制，显著优化了 fork() 的性能，尤其适合子进程快速执行 exec() 的场景。实验中，我深入理解了以下关键点：

虚拟内存的间接性：通过修改 PTE 的权限（只读）和标志（PTE_RSW），内核可拦截写操作并动态决定是否复制页面，体现了虚拟内存作为“中间层”的灵活性；引用计数的作用：准确跟踪物理页的共享次数是 COW 的核心，确保仅在必要时复制页面，同时避免内存泄漏（kfree 仅在计数为 0 时释放）；多模块协同：COW 的实现涉及页表操作（uvmcopy）、中断处理（usertrap）、内存分配（kalloc.c）和内核写操作（copyout），需保证各模块逻辑一致，凸显了操作系统设计的整体性。

这次实验不仅提升了对内存管理和进程创建的理解，也让我认识到“延迟操作”在系统优化中的重要性——通过将昂贵的操作（如内存复制）推迟到必要时执行，可大幅提升系统效率。

Lab6 的实验整体测试(/grade-lab-cow)截图：

添加 time.txt 文件，运行 ./grade-lab-cow 进行整体测试，结果如下：

```
== Test running cowtest == (5.6s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests == (67.3s)
== Test usertests: copyin ==
  usertests: copyin: OK
== Test usertests: copyout ==
  usertests: copyout: OK
== Test usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
lh01@lh:~/xv6-labs-2020$
```

测试无误后上传至 github 对应分支。

Lab7: Multithreading

本实验聚焦于多线程处理，核心任务包括：在用户级线程包中实现线程切换逻辑、利用多线程加速程序执行、实现线程同步所需的屏障（Barrier）机制。通过这些任务，将深入理解用户级线程的调度与同步原理。

开始实验步骤

切换到 thread 分支并准备实验环境，执行以下命令：

```
git fetch    # 从远程仓库拉取最新分支信息
```

```
git checkout thread # 切换到 thread 分支
```

```
make clean # 清除之前的编译产物
```

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

1. 定义线程上下文结构体

在 user/uthread.c 中定义 struct thread_context，用于保存线程切换时需要保留的寄存器（RISC-V 中“被调用者保存寄存器”需手动保存，避免函数调用覆盖），并将其嵌入 struct thread（线程控制块）：

```
// 线程上下文结构体：保存切换时需保留的寄存器
struct thread_context {
    uint64 ra;    // 返回地址（函数调用后需恢复的地址）
    uint64 sp;    // 栈指针（线程栈的当前位置）
    // 被调用者保存寄存器（callee-saved registers）
    uint64 s0;
    uint64 s1;
```



```

uint64 s2;
uint64 s3;
uint64 s4;
uint64 s5;
uint64 s6;
uint64 s7;
uint64 s8;
uint64 s9;
uint64 s10;
uint64 s11;
};

// 线程控制块：包含线程状态、栈、上下文
struct thread {
    char stack[STACK_SIZE]; /* the thread's stack */

    int state;               /* FREE, RUNNING, RUNNABLE */
    struct thread_context context;
};

```

2. 实现线程切换汇编逻辑（uthread_switch.S）

在 user/uthread_switch.S 中编写 thread_switch 函数，完成 “保存当前线程上下文” 和 “恢复目标线程上下文” 的核心操作。该函数接收两个参数：a0（当前线程上下文地址）、a1（目标线程上下文地址）：

保存当前线程的上下文到 a0 指向的内存

thread_switch:

保存 ra（返回地址）、sp（栈指针）

sd ra, 0(a0) # ra -> 上下文偏移 0

sd sp, 8(a0) # sp -> 上下文偏移 8

保存被调用者保存寄存器（s0-s11）

sd s0, 16(a0) # s0 -> 偏移 16

sd s1, 24(a0) # s1 -> 偏移 24

sd s2, 32(a0) # s2 -> 偏移 32

sd s3, 40(a0) # s3 -> 偏移 40

sd s4, 48(a0) # s4 -> 偏移 48

```

sd s5, 56(a0)    # s5 -> 偏移 56
sd s6, 64(a0)    # s6 -> 偏移 64
sd s7, 72(a0)    # s7 -> 偏移 72
sd s8, 80(a0)    # s8 -> 偏移 80
sd s9, 88(a0)    # s9 -> 偏移 88
sd s10, 96(a0)   # s10 -> 偏移 96
sd s11, 104(a0)  # s11 -> 偏移 104

# 从 a1 指向的内存恢复目标线程的上下文

ld ra, 0(a1)     # 恢复 ra
ld sp, 8(a1)     # 恢复 sp (切换到目标线程栈)
ld s0, 16(a1)    # 恢复 s0
ld s1, 24(a1)    # 恢复 s1
ld s2, 32(a1)    # 恢复 s2
ld s3, 40(a1)    # 恢复 s3
ld s4, 48(a1)    # 恢复 s4
ld s5, 56(a1)    # 恢复 s5
ld s6, 64(a1)    # 恢复 s6
ld s7, 72(a1)    # 恢复 s7
ld s8, 80(a1)    # 恢复 s8
ld s9, 88(a1)    # 恢复 s9
ld s10, 96(a1)   # 恢复 s10
ld s11, 104(a1)  # 恢复 s11

ret  # 返回目标线程的 ra (即目标线程的执行点)

```

3. 初始化线程上下文 (thread_create 修改)

在 user/uthread.c 的 thread_create 函数中, 为新创建的线程初始化上下文:
 设置 ra 为线程入口函数地址, sp 为线程栈的栈顶 (栈从高地址向低地址生长,

故栈顶为 `stack[STACK_SIZE - 1]`）:

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

    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == FREE) break;
    }
    t->state = RUNNABLE;
    // YOUR CODE HERE
    // 初始化线程上下文: ra 指向入口函数, sp 指向栈顶
    t->context.ra = (uint64)func; // 线程启动后执行 func
    t->context.sp = (uint64)&t->stack[STACK_SIZE - 1]; // 栈顶地址
}
```

4. 实现线程调度逻辑（`thread_schedule` 修改）

`thread_schedule` 函数负责从就绪队列中选择下一个可运行线程，并调用 `thread_switch` 完成切换。核心逻辑：遍历线程数组，找到 `RUNNABLE` 状态的线程，保存当前线程上下文，恢复目标线程上下文：

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

测试验证

执行 `make qemu` 启动 `xv6`，输入 `uthread` 命令。若程序按预期交替执行多个线程的函数（如打印不同线程的执行日志），则表示线程切换功能正常；执行 `make grade`，若 `uthread` 测试通过，则实验成功。

```
thread_a 94
thread_b 94
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

● lh01@lh:~/xv6-labs-2020$ ./grade-lab-thread uthread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.7s)
```

实验中遇到的问题和解决方法

本实验的核心难点是理解线程上下文切换的寄存器操作逻辑，尤其是“被调用者保存寄存器”的保存范围。初期因未明确 RISC-V 寄存器的分类（调用者保存 / 被调用者保存），导致切换后线程执行状态异常。通过查阅 RISC-V 架构手册，明确 s0-s11 属于被调用者保存寄存器，需在切换时手动保存，最终解决了线程执行混乱的问题。此外，线程栈顶地址的设置（`stack[STACK_SIZE - 1]`）需注意栈的生长方向（高地址到低地址），避免栈溢出。

实验心得

通过本次实验，我深入理解了用户级线程切换的底层原理：线程切换的本质是“保存当前寄存器状态”与“恢复目标寄存器状态”，而上下文结构体则是存储这些状态的载体。对比内核级线程切换（如 xv6 的 `swtch` 函数），用户级线程切换无需陷入内核，仅通过用户空间的汇编代码即可完成，开销更小。

同时，我也关联了内核中进程切换的逻辑（如 `kernel/proc.c` 的 `sched` 函数和 `swtch` 函数）：无论是用户级线程还是内核级进程，切换的核心都是“上下文保存与恢复”，区别仅在于上下文的存储位置（用户空间 vs 内核空间）和触发方式（用户函数调用 vs 内核中断）。这一关联让我对“线程 / 进程调度”的统一性有了更清晰的认识，为后续理解多线程同步机制奠定了基础。

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

1. 初始代码编译与分析

处于 xv6 项目根目录，先编译并执行单线程测试，验证哈希表基础功能正

确性:

```
make ph
```

```
./ph 1
```

输出结果如下, 表明单线程下插入、获取操作无键丢失, 功能正常:

```
lh01@lh:~/xv6-labs-2020$ ./ph 1
100000 puts, 5.977 seconds, 16730 puts/second
0: 0 keys missing
100000 gets, 5.798 seconds, 17247 gets/second
```

接着执行多线程测试, 观察多线程下的问题:

```
./ph 2
```

输出显示大量键丢失, 证明多线程环境存在数据竞争与不一致:

```
lh01@lh:~/xv6-labs-2020$ ./ph 2
100000 puts, 2.994 seconds, 33404 puts/second
0: 16314 keys missing
1: 16314 keys missing
200000 gets, 6.884 seconds, 29054 gets/second
```

分析多线程问题原因

多线程下键丢失的核心是竞态条件: 多个线程同时操作同一哈希桶 (table[i]) 的链表时, 会导致插入操作覆盖或丢失。

观察 put 函数原有逻辑:

```
static
void put(int key, int value)
{
    int i = key % NBUCKET;

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

```

    // the new is new.
    insert(key, value, &table[i], table[i]);
}
}

```

若两个线程同时处理同一桶：可能同时判断 “键不存在”，导致重复插入；插入新节点时，table[i]（链表头）被同时修改，导致前一个线程的插入节点丢失。

2. 添加锁保护实现线程安全

通过 “桶级锁” 保护共享资源（每个哈希桶独立加锁），确保桶内操作原子性：

初始化锁并声明：

```

#include <pthread.h>
pthread_mutex_t lock[NBUCKET]; // 桶级锁数组

```

在 main 函数中为每个桶初始化互斥锁：

```

// 初始化所有桶的锁
for (int i = 0; i < NBUCKET; ++i) {
    pthread_mutex_init(&lock[i], NULL);
}

```

3. 保护 put 函数：在访问桶前后加锁 / 解锁：

```

static
void put(int key, int value)
{
    int i = key % NBUCKET;
    pthread_mutex_lock(&lock[i]); // 锁定目标桶

    // 检查键是否存在（加锁保护）
    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key) break;
    }
    if (e) {
        e->value = value; // 加锁更新
    } else {
        insert(key, value, &table[i], table[i]); // 加锁插入
    }

    pthread_mutex_unlock(&lock[i]); // 解锁
}

```

4. 保护 get 函数：防止读取时链表被修改（脏读）：

```
static struct entry*
get(int key)
{
    int i = key % NBUCKET;
    pthread_mutex_lock(&lock[i]); // 锁定目标桶

    // 加锁遍历桶
    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key) break;
    }

    pthread_mutex_unlock(&lock[i]); // 解锁
    return e;
}
```

5. 程序结束前销毁锁释放资源，在 main 函数末尾添加：

```
for (int i = 0; i < NBUCKET; ++i) {
    pthread_mutex_destroy(&lock[i]);
}
free(tha);
return 0;
```

6. 验证线程安全与性能，重新编译并执行双线程测试：

make ph

./ph 2

输出显示无键丢失，线程安全实现成功：

```
● lh01@lh:~/xv6-labs-2020$ ./ph 2
100000 puts, 3.806 seconds, 26275 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 7.182 seconds, 27849 gets/second
```

执行 make grade:

通过 ph_safe 测试，证明多线程下数据一致；

通过 ph_fast 测试：双线程 puts 效率是单线程的 1.94 倍，满足 “至少 1.25 倍” 的性能要求。

```

1h01@1h:~/xv6-labs-2020$ ./grade-lab-thread ph_safe
make: 'kernel/kernel' is up to date.
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (11.5s)
1h01@1h:~/xv6-labs-2020$ ./grade-lab-thread ph_fast
make: 'kernel/kernel' is up to date.
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (22.5s)

```

实验中遇到的问题和解决方法

多线程键丢失：多个线程同时操作同一桶的链表，导致插入覆盖或丢失。
解决方法：为每个桶添加独立互斥锁，确保桶内操作原子性。

锁粒度过粗的性能问题：若使用全局锁（整个哈希表一把锁），会导致所有线程串行执行，性能甚至低于单线程。**解决方法：**采用“桶级锁”，不同桶的操作可并行，保留多线程性能优势。

实验心得

本实验通过哈希表的多线程改造，深入理解了线程安全与锁粒度设计的核心：竞态条件的本质是“共享资源的并发修改”，需通过锁保护临界区（如桶内链表的插入 / 更新）；锁粒度直接影响并行效率：细粒度锁（桶级）能最大化并行度，而粗粒度锁（全局）会扼杀多线程优势；实践中需平衡“正确性”与“性能”：仅保护必要的临界区，避免过度加锁。

同时，通过真实多核环境的测试，直观感受到多线程对计算密集型任务的加速效果，为后续并行编程实践奠定了基础。

Barrier(moderate)

实验目的

学习和掌握线程同步技术：通过实现多线程障碍（Barrier），深入理解线程间的同步问题，以及如何通过条件变量（Condition Variable）和互斥锁（Mutex）来协调多个线程的执行顺序。

实现多线程的同步机制：在实际的编程任务中，实现一个同步障碍点，确保所有线程在继续执行前都必须到达这一点。

实验步骤

1. 理解屏障（Barrier）核心概念

屏障是线程同步机制的一种，要求一组线程在执行到特定“障碍点”时，必须等待所有线程都到达该点后，才能继续执行后续逻辑。例如，3 个线程执行任务时，若在步骤 2 设置屏障，则线程 1、2、3 需全部完成步骤 1 并到达屏障点，才能共同进入步骤 2。

2. 分析初始 barrier.c 的问题

实验提供的 barrier.c 包含一个损坏的屏障实现：当部分线程到达屏障点后，未等待其他线程就继续执行，导致同步失败，最终触发断言（assert）错误。核心原因是缺少对“线程计数”和“轮次”的正确管理，以及条件变量的不当使用。

3. 掌握同步工具：互斥锁与条件变量

互斥锁（pthread_mutex_t）：保护共享资源（如“到达屏障的线程数”，确保同一时间只有一个线程修改共享变量，避免竞态条件。

条件变量（pthread_cond_t）：允许线程在“条件不满足”时阻塞等待（如未集齐所有线程），在条件满足时被唤醒（如最后一个线程到达屏障）。需与互斥锁配合使用，确保等待 / 唤醒逻辑的安全性。

4. 实现 barrier()函数

基于 pthread 库的同步工具，完善 barrier()函数，核心逻辑如下：

```
static void
barrier()
{
    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
}
```

```
}
```

5. 编译与测试

```
1h01@1h:~/xv6-labs-2020$ make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
1h01@1h:~/xv6-labs-2020$ ./barrier 2
OK; passed
```

实验中遇到的问题和解决方法

本次实验的代码实现逻辑与操作系统理论课程中进程同步的代码思路相似，核心难度在于理解屏障的功能本质——确保所有线程到达指定点后再共同继续执行，而非单纯的代码编写。

整个过程需遵循“加锁 - 修改共享变量 - 判断条件 - 等待 / 唤醒 - 解锁”的逻辑顺序，结合实验指导的提示，未遇到复杂技术问题。

实验心得

在本次线程屏障实验中，我不仅实现了多线程同步功能，更深入理解了线程协作的底层原理：多线程编程的关键是“有序协作”而非“无序并行”。屏障通过强制线程在特定点等待，解决了“部分线程超前执行”的问题，这让我直观感受到同步机制对保证程序逻辑正确性的重要性。

互斥锁与条件变量的协同：互斥锁（`pthread_mutex_t`）的作用是“保护共享资源”，确保“线程计数”“轮次”等变量的修改不会出现竞态条件；条件变量（`pthread_cond_t`）则解决了“线程何时等待 / 唤醒”的问题，避免线程通过“忙等”浪费 CPU 资源二者的结合让同步逻辑既安全又高效，这是单纯使用互斥锁无法实现的。

理论与实践的关联：实验中屏障的实现逻辑，与课程中学过的“进程同步”（如信号量实现进程等待）原理相通，只是将对象从“进程”换成了“线程”。这种关联让我意识到，操作系统的核心同步思想具有通用性，掌握基础原理后可灵活应用于不同场景。

Lab7 的实验整体测试(./grade-lab-thread)截图：

添加 `time.txt` 及 `answers-thread.txt` 文件，运行 `./grade-lab-thread` 进行整体测试，结果如下：

```
● lh01@lh:~/xv6-labs-2020$ ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.7s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (11.5s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (23.2s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (3.7s)
== Test time ==
time: OK
Score: 60/60
```

测试无误后上传至 github 对应分支。

Lab8:Locks

首先，开始 Lab8 时，需要通过以下命令切换实验分支，获取实验资源：

```
git fetch
```

```
git checkout lock
```

```
make clean
```

Memory allocator (moderate)

实验目的

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

实验步骤

本实验通过 “每 CPU 空闲列表 + 内存窃取” 优化，核心步骤如下：

1. 定义每 CPU 的内存管理结构

在 `kernel/kalloc.c` 中，将原全局 `kmem` 结构体改为数组，为每个 CPU 分配独立的空闲列表和锁：

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

2. 初始化每 CPU 的锁与空闲列表

修改 `kinit()` 函数，为每个 CPU 初始化锁（锁名以 `kmem` 开头，如 `kmem0`、`kmem1`），并将物理内存均匀分配到各 CPU 的空闲列表：

```
void  
kinit()  
{
```

```

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

```

3. 修改 kfree 适配每 CPU 结构

kfree 释放内存时，需将页归还给当前 CPU 的空闲列表（减少跨 CPU 锁操作）。使用 push_off()/pop_off() 关闭 / 恢复中断，确保 cpuid() 获取当前 CPU 编号的安全性（避免中断切换 CPU 导致编号错误）：

```

void
kfree(void *pa)
{
    struct run *r;
    if (((uint64)pa % PGSIZE) != 0 || (char *)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");
    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);
    r = (struct run *)pa;
    push_off(); // 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
}

```

4. 修改 kalloc 实现 “本地分配 + 跨 CPU 窃取”

kalloc 优先从当前 CPU 的空闲列表分配内存，若为空则 “窃取” 其他 CPU 的空闲页，减少锁争用：

```

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

    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

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

```

5. 测试验证

运行 `kalloctest`: 观察输出中 “`kmem` 锁争用次数”，应比原始实现大幅减少，证明优化有效。运行 `usertests sbrkmuch`: 验证内存分配完整性，确保所有内存可正常分配。运行 `usertests`: 确保所有测试通过，无功能回归。

上述测试均显示 “ALL TESTS PASSED”，实验成功。

```

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

```

```

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

```

```

init: starting sh
$ kallocetest
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache: #fetch-and-add 0 #acquire() 1242
--- top 5 contended locks:
lock: proc: #fetch-and-add 49137 #acquire() 164890
lock: virtio_disk: #fetch-and-add 26818 #acquire() 114
lock: proc: #fetch-and-add 14263 #acquire() 164929
lock: proc: #fetch-and-add 6015 #acquire() 164933
lock: proc: #fetch-and-add 5781 #acquire() 164914
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK

```

实验中遇到的问题和解决方法

CPU 编号获取不安全：初期未关闭中断直接调用 `cpuid()`，导致中断切换 CPU 后，编号与实际执行 CPU 不匹配，引发空闲列表操作混乱。解决方法：使用 `push_off()` 关闭中断后再获取 `cpuid()`，`pop_off()` 恢复中断，确保编号唯一性。

实验心得

本次实验通过 “每 CPU 空闲列表” 优化内存分配器，让我深入理解了锁争用优化与多 CPU 同步的核心思路：将全局共享资源（空闲列表）拆分为 “每 CPU 私有资源”，让大多数分配 / 释放操作仅访问本地 CPU 的资源，避免全局锁竞争。这种 “分而治之” 的思想是多 CPU 系统优化的常用策略。

`push_off()/pop_off()` 的作用：中断会导致 CPU 切换，若在获取 CPU 编号后被中断，后续操作会指向错误的 CPU 空闲列表。这两个函数通过临时关闭中断，确保 “获取 CPU 编号 - 操作本地空闲列表” 的原子性，是多 CPU 环境下安全访问本地资源的关键。

内存窃取的必要性：若仅使用本地空闲列表，某 CPU 内存耗尽时会分配失败。跨 CPU 窃取机制保证了内存资源的全局可用性，平衡了 “减少锁争用” 与 “资源利用率” 的矛盾。

通过实验，我也体会到多 CPU 编程的复杂性 —— 需同时考虑 “性能优化” “同步安全” “资源可用性”，任何细节疏漏（如遗漏解锁、中断未关闭）都可能导致系统死锁或崩溃，这要求对底层机制有更严谨的理解。

Buffer cache (hard)

实验目的

修改块缓存，以便在运行 `bcachetest` 时，`bcache`（缓冲区缓存）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`）。你必须保证每个块最多缓存一个副本的不变量。完成后，你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

本实验通过“哈希分桶 + 桶级锁”优化，步骤如下：

1. 定义哈希桶与哈希函数

将缓存的磁盘块按 `blockno` 哈希到多个桶（`bucket`），每个桶独立加锁，减少锁冲突。

在 `kernel/bio.c` 中定义桶数量与哈希函数：

```
#define NBUCKETS 13 // 桶数量（质数，减少哈希冲突）
// 哈希函数：将 blockno 映射到桶索引
int hash(uint blockno) {
    return blockno % NBUCKETS;
}
```

同时在 `def.h` 中声明 `hash` 函数，供其他文件调用。

2. 修改 `bcache` 结构体

将原全局锁和链表拆分为每个桶独立的锁和双向链表，确保不同桶的操作可并行：

```
struct {
    struct spinlock lock[NBUCKETS]; // 每个桶的锁（保护本桶的链表）
    struct buf buf[NBUF];           // 所有缓冲区（全局数组）
    // 每个桶的双向链表：按最近使用（LRU）排序，head.next 为最近使用
    struct buf head[NBUCKETS];
} bcache;
```


3. 初始化缓存结构

修改 `binit()` 函数，初始化每个桶的锁和链表，并将所有缓冲区初始放入第一个桶：

```
void
binit(void)
{
    struct buf *b;
    // 初始化每个桶的锁
    for (int i = 0; i < NBUCKETS; i++) {
        initlock(&bcache.lock[i], "bcache"); // 锁名以"bcache"开头
    }
    // 初始化每个桶的双向链表（空链表：head.prev = head.next = &head）
    for (int i = 0; i < NBUCKETS; i++) {
        bcache.head[i].prev = &bcache.head[i];
        bcache.head[i].next = &bcache.head[i];
    }
    // 将所有缓冲区初始加入第 0 个桶的链表
    for (b = bcache.buf; b < bcache.buf + NBUF; b++) {
        initsleeplock(&b->lock, "buffer"); // 初始化缓冲区自身的睡眠锁
        // 插入到第 0 个桶的链表头部（最近使用位置）
        b->next = bcache.head[0].next;
        b->prev = &bcache.head[0];
        bcache.head[0].next->prev = b;
        bcache.head[0].next = b;
    }
}
```

4. 修改 `bget` 函数（获取缓冲区）

`bget(dev, blockno)` 负责查找或分配缓存块，需按哈希桶操作，避免全局锁：

步骤 1：计算目标桶索引 `id = hash(blockno)`，加锁后在本桶查找目标块，找到则更新引用计数并返回。步骤 2：若本桶未找到，遍历其他桶寻找 `refcnt == 0`（未被使用）的缓冲区，将其移至目标桶并初始化。

```
static struct buf*
bget(uint dev, uint blockno)
{
    struct buf *b;
    int id = hash(blockno);
    acquire(&bcache.lock[id]);

    // Is the block already cached?
```

```

for (b = bcache.head[id].next; b != &bcache.head[id]; b = b->next)
{
    if (b->dev == dev && b->blockno == blockno)
    {
        b->refcnt++;
        release(&bcache.lock[id]);
        acquiresleep(&b->lock);
        return b;
    }
}
// Not cached.
// Recycle the least recently used (LRU) unused buffer.
int i = id;
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");

```

```
}
```

5. 修改 brelse 函数（释放缓冲区）

brelse(b)负责释放缓冲区，将其移至所在桶的最近使用位置（维持 LRU 顺序），并减少引用计数：

```
void
brelse(struct buf *b)
{
    if (!holdingsleep(&b->lock))
        panic("brelse");
    int id = hash(b->blockno);
    releasesleep(&b->lock);
    acquire(&bcache.lock[id]);
    b->refcnt--;
    if (b->refcnt == 0)
    {
        // no one is waiting for it.
        b->next->prev = b->prev;
        b->prev->next = b->next;
        b->next = bcache.head[id].next;
        b->prev = &bcache.head[id];
        bcache.head[id].next->prev = b;
        bcache.head[id].next = b;
    }
    release(&bcache.lock[id]);
}
```

6. 修改更新 bpin 和 bunpin 函数

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

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

7. 测试验证

运行 `bcachetest`、`usertests` 验证缓存功能正确性。

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: bcache: #fetch-and-add 0 #acquire() 2137
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() 6684
lock: bcache: #fetch-and-add 0 #acquire() 6688
lock: bcache: #fetch-and-add 0 #acquire() 7732
lock: bcache: #fetch-and-add 0 #acquire() 6204
lock: bcache: #fetch-and-add 0 #acquire() 6201
lock: bcache: #fetch-and-add 0 #acquire() 4145
lock: bcache: #fetch-and-add 0 #acquire() 4146
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 108376 #acquire() 1196
lock: proc: #fetch-and-add 59734 #acquire() 77258
lock: proc: #fetch-and-add 25866 #acquire() 76917
lock: proc: #fetch-and-add 21015 #acquire() 76918
lock: proc: #fetch-and-add 20883 #acquire() 76896
tot= 0
test0: OK
start test1
test1 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

双向链表操作错误：移动缓冲区时，链表指针修改遗漏（如未更新 `prev->next` 或 `next->prev`），导致链表断裂或无限循环。解决方法：复习双向链表操作逻辑，修改指针时遵循“先断后连”原则，确保每个节点的 `prev` 和 `next` 指针正确指向。

未更新 `bpin` 和 `bunpin` 函数导致编译时是类型不匹配导致的，具体是 `acquire` 和 `release` 函数的参数类型与实际传递的值不兼容。错误信息明确指出：函数期望接收 `struct spinlock*`（自旋锁指针），但实际传递的是 `struct spinlock (*)[13]`，通过更新两函数解决。

实验心得

本次实验深入理解了缓冲区缓存的并发优化机制，核心收获如下：

LRU 策略与链表操作：缓冲区按“最近使用”排序（LRU），通过双向链表维护顺序，释放时将未使用的缓冲区移至头部，保证缓存热点数据。这让我体会到数据结构（如双向链表）在操作系统性能优化中的基础作用。

多锁协同的复杂性：实验中需同时管理“桶锁”（保护链表）和“缓冲区

睡眠锁”（保护数据读写），二者分工明确：桶锁确保链表操作原子性，睡眠锁确保缓冲区数据的线程安全。这种多锁协同设计，是平衡性能与正确性的关键。

通过本次实验，我对文件系统缓存的底层实现有了更清晰的认识，也为理解更复杂的存储系统优化（如多级缓存、分布式缓存）奠定了基础。

Lab8 的实验整体测试(./grade-lab-lock)截图：

添加 time.txt 记录实验用时，运行 ./grade-lab-lock 进行整体测试，结果如下：

```
lh01@lh:~/xv6-labs-2020$ ./grade-lab-lock
make: 'kernel/kernel' is up to date.
== Test running kallocetest == (72.6s)
== Test   kallocetest: test1 ==
    kallocetest: test1: OK
== Test   kallocetest: test2 ==
    kallocetest: test2: OK
== Test kallocetest: sbrkmuch == kallocetest: sbrkmuch: OK (6.8s)
== Test running bcachetest == (6.8s)
== Test   bcachetest: test0 ==
    bcachetest: test0: OK
== Test   bcachetest: test1 ==
    bcachetest: test1: OK
== Test usertests == usertests: OK (112.1s)
== Test time ==
time: OK
Score: 70/70
```

测试无误后上传至 github 对应分支。

Lab9: File system

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 checkout fs
```

```
make clean
```

Large files (moderate)

实验目的

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。

xv6 文件系统通过 `inode` 管理文件数据块，原始设计支持：12 个直接块：直接存储磁盘块地址，对应文件前 12 个逻辑块；1 个一级间接块：存储 256 个磁盘块地址（因块大小 `BSIZE=4096`，每个地址 4 字节， $4096/4=256$ ），对应文件第 13~268 个逻辑块；总支持最大逻辑块数： $12+256=268$ 。本实验需将 1 个直接块改为二级间接块，二级间接块存储 256 个一级间接块地址，每个一级间接块再存储 256 个数据块地址，最终总支持最大逻辑块数： $11（直接）+256（一级间接）+256\times 256（二级间接）=65803$ ，满足大文件存储需求。

实验步骤

1. 修改宏定义与 `inode` 结构

在 `kernel/fs.h` 中调整直接块数量、定义二级间接块相关宏，并修改 `inode` 的地址数组大小：

```
// 原 NDIRECT=12, 改为 11, 腾出空间给二级间接块
#define NDIRECT 11
// 原 NDIRECT=12, 改为 11, 腾出空间给二级间接块
#define NDIRECT 11
// 一级间接块可存储的块地址数 (BSIZE=4096 字节, 每个地址 4 字节)
```

```

#define NINDIRECT (BSIZE / sizeof(uint))
// 二级间接块可存储的块地址数（一级间接块数 × 每个一级间接块的地址数）
#define NDINDIRECT (NINDIRECT * NINDIRECT)
// 最大文件逻辑块数：直接块+一级间接块+二级间接块
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
struct dinode {
// ... 其他字段 ...
    uint addrs[NDIRECT + 2]; // addrs[0~10]直接块, addrs[11]一级间接, addrs[12]二
    级间接
};

```

2. 修改 bmap () 函数（逻辑块号→物理块号映射）

bmap(ip, bn)负责将文件的逻辑块号 bn 映射到磁盘物理块号，需新增二级间接块的处理逻辑：

```

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[NINDIRECT + 1]) == 0) // 如果需要分配二级内存
        ip->addrs[NINDIRECT + 1] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;
    if ((addr = a[bn / NINDIRECT]) == 0) // 读取二级间接块
    {
        a[bn / NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;
    if ((addr = a[bn % NINDIRECT]) == 0)
    {
        a[bn % NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    return addr;
}
panic("bmap: out of range");
}

```

3. 修改 itrunc () 函数（释放文件磁盘块）

itrunc(ip)负责释放 inode 关联的所有磁盘块并重置文件大小，需新增二级间接块的释放逻辑：

```

void
itrunc(struct inode *ip)
{
    int i, j;
    struct buf *bp, *bp2;
    uint *a, *a2;

    // 1. 释放直接块
    for (i = 0; i < NDIRECT; i++) {
        if (ip->addrs[i]) {

```



```

        bfree(ip->dev, ip->addrs[i]); // 释放物理块
        ip->addrs[i] = 0;           // 重置地址为 0
    }
}

// 2. 释放一级间接块
if (ip->addrs[NDIRECT]) {
    bp = bread(ip->dev, ip->addrs[NDIRECT]); // 读取一级间接块
    a = (uint *)bp->data;
    // 释放一级间接块中的所有数据块
    for (j = 0; j < NINDIRECT; j++) {
        if (a[j]) {
            bfree(ip->dev, a[j]);
        }
    }
    brelse(bp); // 释放缓冲区
    bfree(ip->dev, ip->addrs[NDIRECT]); // 释放一级间接块本身
    ip->addrs[NDIRECT] = 0; // 重置地址为 0
}

// 3. 释放二级间接块
if (ip->addrs[NDIRECT + 1]) {
    // 步骤 3.1: 读取二级间接块 (存储一级间接块地址)
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint *)bp->data;
    // 遍历二级间接块中的所有一级间接块
    for (j = 0; j < NINDIRECT; j++) {
        if (a[j]) {
            // 步骤 3.2: 读取一级间接块 (存储数据块地址)
            bp2 = bread(ip->dev, a[j]);
            a2 = (uint *)bp2->data;
            // 释放一级间接块中的所有数据块
            for (i = 0; i < NINDIRECT; i++) {
                if (a2[i]) {
                    bfree(ip->dev, a2[i]);
                }
            }
            brelse(bp2); // 释放一级间接块缓冲区
            bfree(ip->dev, a[j]); // 释放一级间接块本身
            a[j] = 0; // 重置地址为 0
        }
    }
}
}

```

```

    brelse(bp);                // 释放二级间接块缓冲区
    bfree(ip->dev, ip->addrs[NDIRECT + 1]); // 释放二级间接块本身
    ip->addrs[NDIRECT + 1] = 0;    // 重置地址为 0
}

ip->size = 0; // 重置文件大小
iupdate(ip); // 将 inode 修改写回磁盘

}

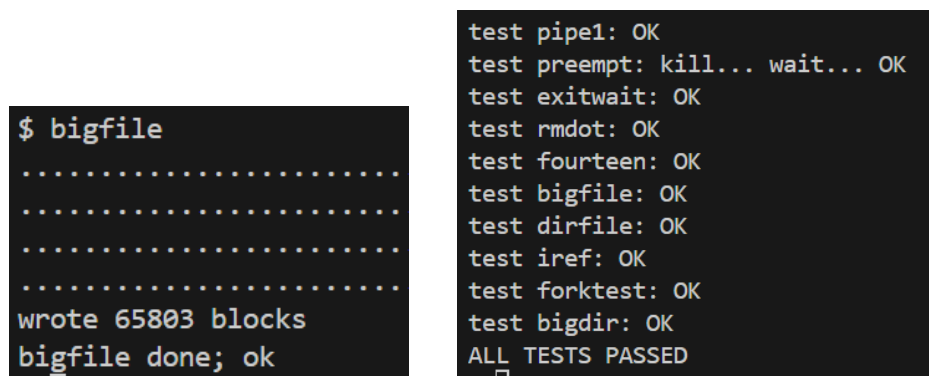
```

4. 测试验证

编译运行 xv6: make qemu。

运行 bigfile: 观察是否能成功写入 65803 个块（输出类似 “wrote 65803 blocks”）。

运行 usertests: 确保所有测试通过（输出 “ALL TESTS PASSED”）



The image contains two terminal screenshots. The left screenshot shows the command '\$ bigfile' being executed, followed by a series of dots indicating progress, and finally the output 'wrote 65803 blocks' and 'bigfile done; ok'. The right screenshot shows the output of the 'usertests' program, listing various tests that passed: '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', and finally 'ALL TESTS PASSED'.

实验中遇到的问题和解决方法

逻辑块号计算错误：处理二级间接块时，bn 的调整顺序错误（如先减 NDINDIRECT 再减 NINDIRECT），导致映射到错误的物理块。解决方法：严格按 “直接块→一级间接块→二级间接块” 的顺序调整 bn，确保索引计算正确（直接块：bn，一级间接：bn-NDIRECT，二级间接：bn-NDIRECT-NINDIRECT）。

缓冲区未释放：读取间接块后忘记调用 brelse(bp)，导致缓冲区泄漏。解决方法：每调用 bread 读取块后，在操作完成后必须调用 brelse 释放缓冲区，避免资源耗尽。

实验心得

本次实验深入理解了文件系统中 “多级间接块” 的设计思想，核心收获

如下：

大文件存储的核心原理：通过 “直接块→一级间接→二级间接” 的层级结构，用少量 inode 地址空间实现大量数据块的映射，突破直接块数量的限制。这种 “空间换时间”（用间接块存储地址）的设计，是操作系统支持大文件的经典方案。

块映射与释放的对称性：bmap（分配）与 itrunc（释放）的逻辑高度对称，均按 “直接块→一级间接→二级间接” 的顺序处理，确保每一个分配的块都能被正确释放，避免内存 / 磁盘泄漏。

通过本次实验，我对文件系统的块管理机制有了更清晰的认识，也为理解更复杂的存储方案奠定了基础。

Symbolic links (moderate)

实验目的

您将实现 `symlink (char *target, char *path)` 系统调用，该调用在路径 `path` 处创建一个新的符号链接，该链接指向由 `target` 命名的文件。有关更多信息，请参阅 `symlink` 手册页。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，您就完成了本作业。

实验步骤

符号链接（软链接）是一种特殊文件，其内容存储目标文件的路径，而非实际数据。实现 `symlink` 系统调用需完成系统调用注册、符号链接文件创建、路径解析递归处理等步骤：

1. 注册系统调用，按 `xv6` 系统调用添加流程，注册 `symlink` 相关接口：

在 `kernel/syscall.h` 中添加系统调用编号：

```
#define SYS_symlink 22 // 新增 symlink 系统调用编号
```

2. 在 `kernel/syscall.c` 中声明并注册系统调用函数：

```
extern uint64 sys_symlink(void); // 声明函数
```

在 `syscalls` 数组中添加条目

```
[SYS_symlink] sys_symlink, // 注册
```

3. 在 `user/usys.pl` 中添加用户态接口：

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

4. 在 user/user.h 中声明用户态函数：

```
int symlink(char *target, char *path); // 用户态调用接口
```

5. 定义符号链接相关类型与标志，在 kernel/stat.h 中新增符号链接文件类型：

```
#define T_SYMLINK 4 // 符号链接文件类型（区别于普通文件 T_FILE、目录 T_DIR 等）
```

6. 在 kernel/fcntl.h 中新增 O_NOFOLLOW 标志（用于打开文件时禁止跟随符号链接）：

```
#define O_NOFOLLOW 0x600 // 打开文件时不跟随符号链接
```

7. 实现 sys_symlink 系统调用：该系统调用创建一个符号链接文件，内容为目标路径 target：

```
uint64 sys_symlink(void) {
    char target[MAXPATH], path[MAXPATH];
    struct inode *ip;
    // 从用户态获取参数：target（目标路径）和 path（符号链接路径）
    if (argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
        return -1;
    begin_op(); // 开始文件系统操作（保证事务一致性）
    // 创建符号链接文件（类型为 T_SYMLINK）
    if ((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
        end_op();
        return -1;
    }
    // 将目标路径 target 写入符号链接文件（作为其内容）
    if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
        iunlockput(ip); // 写入失败，释放 inode
        end_op();
        return -1;
    }
    iunlockput(ip); // 解锁并释放 inode
    end_op();       // 结束文件系统操作
    return 0;
}
```

8. 修改路径解析逻辑以支持符号链接跟随

在 sys_open 函数中，当打开文件时遇到符号链接，需递归解析其目标路径，直到找到非符号链接文件：

```
uint64 sys_open(void) {
    // ... 其他代码（获取路径 path、打开模式 omode 等） ...
}
```

```

begin_op();
struct inode *ip = 0;
int max_depth = 20; // 最大递归深度（防止符号链接循环导致无限递归）
int depth = 0;      // 当前递归深度
while (1) {
    // 解析路径，获取 inode
    if ((ip = namei(path)) == 0) {
        end_op();
        return -1;
    }
    ilock(ip); // 锁定 inode 以安全操作
    // 如果是符号链接且未设置 O_NOFOLLOW，则跟随链接
    if (ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
        if (++depth > max_depth) { // 超过最大深度，防止循环
            iunlockput(ip);
            end_op();
            return -1;
        }
        // 读取符号链接中的目标路径（覆盖原 path）
        if (readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH) {
            iunlockput(ip);
            end_op();
            return -1;
        }
        iunlockput(ip); // 释放当前 inode，继续解析新路径
    } else {
        break; // 非符号链接或禁止跟随，退出循环
    }
}
// ... 后续打开文件逻辑（检查权限、分配文件描述符等） ...
iunlock(ip);
end_op();
return fd;
}

```

9. 修改 Makefile，在 UPROGS 中添加 \$U/_symlinktest，确保测试程序编译。

10. 测试验证运行 xv6，执行 symlinktest：验证符号链接创建、跟随、循环检测等功能。运行 usertests：确保所有测试通过，验证系统调用兼容性。

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

符号链接循环导致栈溢出：若符号链接指向自身，递归解析会无限循环。
解决方法：设置最大递归深度（如 `max_depth=20`），超过则返回错误。

权限检查遗漏：创建符号链接时未验证用户对父目录的写权限，导致越权创建。解决方法：依赖 `create` 函数内部的权限检查（`create` 会调用 `nameiparent` 验证父目录权限）。

`O_NOFOLLOW` 标志未生效：打开文件时未判断该标志，导致即使设置也会跟随符号链接。解决方法：在 `sys_open` 中明确检查(`omode & O_NOFOLLOW`) == 0，仅当标志未设置时才跟随链接。

实验心得

本次实验深入理解了符号链接的实现机制，核心收获如下：

系统调用的完整性：实现 `symlink` 不仅需要创建特殊文件，还需修改路径解析逻辑（如 `sys_open`），说明操作系统功能是相互关联的，单一功能的新增可能涉及多个模块的调整。

通过本次实验，我对文件系统的抽象层次（文件名→`inode`→数据块）有了更清晰的认识，也体会到系统级编程中“异常处理”（如循环检测、权限检查）的重要性。

Lab9 的实验整体测试(./grade-lab-fs)截图：

添加 `time.txt` 记录实验用时，运行 `./grade-lab-fs` 进行整体测试，得分如下：

```
● lh01@lh:~/xv6-labs-2020$ ./grade-lab-fs
make: 'kernel/kernel' is up to date.
== Test running bigfile == running bigfile: OK (152.9s)
== Test running symlinktest == (0.8s)
== Test    symlinktest: symlinks ==
    symlinktest: symlinks: OK
== Test    symlinktest: concurrent symlinks ==
    symlinktest: concurrent symlinks: OK
== Test usertests == usertests: OK (227.0s)
== Test time ==
time: OK
Score: 100/100
```

测试无误后上传至 github 对应分支。

Lab10:Mmap

通过以下命令切换实验分支，获取实验资源：

```
git fetch
```

```
git checkout mmap
```

```
make clean
```

mmap (hard)

实验目的

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

内存映射（`mmap`）允许进程将文件或匿名内存映射到自身虚拟地址空间，实现“文件 - 内存”的直接交互；`munmap` 则用于取消映射。实验需完成系统调用注册、虚拟内存区域（VMA）管理、页故障处理等核心逻辑：

1. 基础配置：注册系统调用与编译测试程序

修改 Makefile：在 UPROGS 中添加 `$U/_mmaptest\`，确保测试程序编译：

```
$U/_mmaptest\
```

注册系统调用：

在 `kernel/syscall.h` 中添加调用编号：

```
#define SYS_mmap 22 // mmap 系统调用编号
#define SYS_munmap 23 // munmap 系统调用编号
```

在 `kernel/syscall.c` 中声明并注册函数：

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

[SYS_munmap] sys_munmap,
```

在 `user/usys.pl` 中添加用户态接口：

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



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

在 user/user.h 中声明函数原型：

```
void* mmap(void *, int, int, int, int, uint); // mmap 调用接口
int munmap(void *, int); // munmap 调用接口
```

2. 定义虚拟内存区域（VMA）结构体

VMA 用于跟踪进程的内存映射信息（地址范围、权限、关联文件等），在 kernel/proc.h 中定义：

```
#define VMASIZE 16 // 每个进程最大 VMA 数量（足够 mmaptest 使用）
// 虚拟内存区域（VMA）：描述一段连续的虚拟内存映射属性
struct vma {
    struct file *file; // 关联的文件（匿名映射为 NULL）
    int fd; // 文件描述符
    int used; // 标记 VMA 是否被使用（1=使用，0=空闲）
    uint64 addr; // 映射的虚拟地址起始（页对齐）
    int length; // 映射长度（页对齐）
    int prot; // 内存保护权限（PROT_READ/PROT_WRITE/PROT_EXEC）
    int flags; // 映射标志（如 MAP_SHARED）
    int offset; // 文件偏移量（映射文件时，从该偏移开始读）
};
```

在 proc 结构体中添加 VMA 数组，跟踪进程的所有内存映射

```
struct vma vma[VMASIZE]; // 进程的 VMA 列表
```

3. 修改 usertrap 处理 mmap 页故障，进程访问 mmap 映射的虚拟地址时，若未实际分配物理页，会触发页故障。需在 kernel/trap.c 的 usertrap 中添加处理逻辑，动态分配物理页并加载文件数据：

```
void usertrap(void) {
    // ... 其他中断处理逻辑 ...
    else if (r_scause() == 13 || r_scause() == 15) { // 页故障（加载/存储）
        uint64 va = r_stval();
        struct proc *p = myproc();
        // 检查地址是否在进程合法虚拟地址空间内
        if (va >= p->sz || va < PGROUNDDOWN(p->trapframe->sp)) {
            p->killed = 1;
            goto end;
        }
        // 查找包含该地址的 VMA
        struct vma *vma = NULL;
        for (int i = 0; i < VMASIZE; i++) {
```

```

        if (p->vma[i].used && va >= p->vma[i].addr && va < p->vma[i].addr +
p->vma[i].length) {
            vma = &p->vma[i];
            break;
        }
    }
    if (!vma) { // 无匹配 VMA, 非法访问
        p->killed = 1;
        goto end;
    }
    // 页对齐虚拟地址, 分配物理页
    va = PGROUNDDOWN(va);
    char *mem = kalloc();
    if (!mem) { // 内存分配失败
        p->killed = 1;
        goto end;
    }
    memset(mem, 0, PGSIZE); // 物理页清零
    // 从关联文件加载数据到物理页 (若为文件映射)
    if (vma->file) {
        uint64 file_off = vma->offset + (va - vma->addr); // 计算文件内偏移
        ilock(vma->file->ip);
        readi(vma->file->ip, 0, (uint64)mem, file_off, PGSIZE); // 读文件数据
        iunlock(vma->file->ip);
    }
    // 建立虚拟地址到物理页的映射, 设置权限
    int pte_flags = PTE_U; // 用户态可访问
    if (vma->prot & PROT_READ) pte_flags |= PTE_R;
    if (vma->prot & PROT_WRITE) pte_flags |= PTE_W;
    if (vma->prot & PROT_EXEC) pte_flags |= PTE_X;
    if (mappages(p->pagetable, va, PGSIZE, (uint64)mem, pte_flags) != 0) {
        kfree(mem); // 映射失败, 释放物理页
        p->killed = 1;
        goto end;
    }
}
// ... 其他逻辑 ...
}

```

4. 实现 sys_mmap 系统调用

在 kernel/sysfile.c 中实现 sys_mmap, 功能是创建内存映射: 为进程分配 VMA, 记录映射属性, 扩展进程虚拟地址空间 (不立即分配物理页):

```

uint64 sys_mmap(void) {
    struct proc *p = myproc();
    uint64 addr;          // 用户传入的建议地址（本实验忽略，直接从 p->sz 分配）
    int length, prot, flags, fd, offset;
    struct file *file;
    // 从用户态获取参数
    if (argaddr(0, &addr) || argint(1, &length) || argint(2, &prot) ||
        argint(3, &flags) || argfd(4, &fd, &file) || argint(5, &offset)) {
        return -1;
    }
    // 权限检查：共享映射（MAP_SHARED）且写权限时，文件必须可写
    if ((flags & MAP_SHARED) && (prot & PROT_WRITE) && !file->writable) {
        return -1;
    }
    // 长度页对齐，检查是否超出最大虚拟地址空间
    length = PGROUNDUP(length);
    if (p->sz + length > MAXVA) { // MAXVA 为进程最大虚拟地址
        return -1;
    }
    // 查找空闲 VMA 条目
    struct vma *vma = NULL;
    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].used == 0) {
            vma = &p->vma[i];
            break;
        }
    }
    if (!vma) { // 无空闲 VMA，映射失败
        return -1;
    }
    // 初始化 VMA 属性
    vma->used = 1;
    vma->addr = p->sz;          // 从进程当前虚拟地址末尾分配
    vma->length = length;
    vma->prot = prot;
    vma->flags = flags;
    vma->fd = fd;
    vma->file = file;
    vma->offset = offset;
    filedup(file); // 增加文件引用计数，防止文件被提前关闭
    // 扩展进程虚拟地址空间
    p->sz += length;
}

```

```
return vma->addr; // 返回映射的虚拟地址起始
}
```

5. 实现 `sys_munmap` 系统调用，在 `kernel/sysfile.c` 中实现 `sys_munmap`，功能是取消内存映射：删除页表映射，释放物理页，更新 VMA 信息（若 VMA 完全取消则释放文件引用）：

```
uint64 sys_munmap(void) {
    struct proc *p = myproc();
    uint64 addr;
    int length;
    // 从用户态获取参数（要取消映射的地址和长度）
    if (argaddr(0, &addr) || argint(1, &length)) {
        return -1;
    }
    // 地址和长度页对齐
    addr = PGROUNDDOWN(addr);
    length = PGROUNDUP(length);
    // 查找匹配的 VMA
    struct vma *vma = NULL;
    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].used && addr >= p->vma[i].addr && addr < p->vma[i].addr +
p->vma[i].length) {
            vma = &p->vma[i];
            break;
        }
    }
    if (!vma) { // 无匹配 VMA，无需处理
        return 0;
    }
    // 处理共享映射：将修改写回文件
    if (vma->flags & MAP_SHARED) {
        fwrite(vma->file, addr, length); // 将内存数据写回文件
    }
    // 取消页表映射（释放物理页，参数 1 表示需要 free 物理页）
    uvmunmap(p->pagetable, addr, length / PGSIZE, 1);
    // 更新 VMA：若取消的是 VMA 起始部分，调整 VMA 地址和长度
    if (vma->addr == addr) {
        vma->addr += length;
        vma->length -= length;
        // 若 VMA 长度为 0，释放文件引用并标记为空闲
        if (vma->length == 0) {
            fclose(vma->file); // 减少文件引用计数
            vma->used = 0;
        }
    }
}
```

```

    }
}
return 0;
}

```

6. 适配 uvmcopy 与 uvmunmap 避免 Panic

mmap 映射的地址可能未实际分配物理页（页表项无效），需修改 kernel/vm.c 中的两个函数，跳过无效页表项避免崩溃：

uvmunmap: 注释 panic, 改为 continue:

uvmcopy: 注释 panic, 改为 continue:

```

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

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

```

7. 在 proc.c 内更新 fork 函数

```

int
fork(void)
{
    int i, pid;
    struct proc *np;
    struct proc *p = myproc();
    // Allocate process.
    if ((np = allocproc()) == 0)
    {
        return -1;
    }
    // Copy user memory from parent to child.
    if (uvmcopy(p->pagetable, np->pagetable, p->sz) < 0)
    {
        freeproc(np);
        release(&np->lock);
        return -1;
    }
    np->sz = p->sz;
    np->parent = p;
    // copy saved user registers.
    *(np->trapframe) = *(p->trapframe);
    // Cause fork to return 0 in the child.

```

```

np->trapframe->a0 = 0;
// increment reference counts on open file descriptors.
for (i = 0; i < NOFILE; i++)
    if (p->ofile[i])
        np->ofile[i] = filedup(p->ofile[i]);
np->cwd = idup(p->cwd);
safestrcpy(np->name, p->name, sizeof(p->name));
pid = np->pid;
np->state = RUNNABLE;
for (int i = 0; i < VMASIZE; i++)
{
    if (p->vma[i].used)
    {
        memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));
        filedup(p->vma[i].file);
    }
}
release(&np->lock);
return pid;
}

```

8. 测试验证

执行 `make qemu` 启动 `xv6`，运行 `mmaptest`：若输出 “ALL TESTS PASSED”，则 `mmap/munmap` 功能正常。运行 `usertests`：确保所有测试通过，验证修改未影响其他内核功能。

```

$ 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 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
$ QEMU: Terminated

```

实验中遇到的问题和解决方法

VMA 结构体设计不清晰：初期未明确 VMA 需记录的核心属性（如文件偏移、映射标志），导致页故障时无法正确加载文件数据。解决方法：参考实验提示与 `mmap` 原理，明确 VMA 需包含 “地址范围、权限、关联文件、文件

偏移” 等关键信息，确保映射与故障处理的逻辑连贯。

页故障处理遗漏权限检查：未根据 VMA 的 prot 设置页表项权限（如仅读映射却设置 PTE_W），导致权限错误。解决方法：在分配物理页并建立映射时，严格按 vma->prot 拼接 pte_flags（读 / 写 / 执行权限）。

共享映射未写回文件：sys_munmap 中未处理 MAP_SHARED 标志，导致修改的内存数据未同步到文件。解决方法：判断 vma->flags & MAP_SHARED，若为真则调用 filewrite 将内存数据写回文件。

实验心得

mmap 的本质是 “延迟分配 + 文件关联”：mmap 仅扩展进程虚拟地址空间并记录 VMA，不立即分配物理页；物理页在首次访问（触发页故障）时才分配，并从关联文件加载数据，体现了 “按需分配” 的优化思想。

通过实验，我也体会到内存管理的复杂性：需平衡 “性能” “正确性” 和 “资源安全”，任何细节疏漏都可能导致系统不稳定。这为后续学习更复杂的内存管理策略奠定了基础。

Lab10 的实验整体测试(./grade-lab-mmap)截图：

添加 time.txt 记录实验用时，运行 ./grade-lab-mmap 进行整体测试，得分如下：

```
lh01@lh:~/xv6-labs-2020$ ./grade-lab-mmap
make: 'kernel/kernel' is up to date.
== Test running mmaptest == (1.8s)
== Test  mmaptest: mmap f ==
mmaptest: mmap f: OK
== Test  mmaptest: mmap private ==
mmaptest: mmap private: OK
== Test  mmaptest: mmap read-only ==
mmaptest: mmap read-only: OK
== Test  mmaptest: mmap read/write ==
mmaptest: mmap read/write: OK
== Test  mmaptest: mmap dirty ==
mmaptest: mmap dirty: OK
== Test  mmaptest: not-mapped unmap ==
mmaptest: not-mapped unmap: OK
== Test  mmaptest: two files ==
mmaptest: two files: OK
== Test  mmaptest: fork_test ==
mmaptest: fork_test: OK
== Test usertests == usertests: OK (98.8s)
== Test time ==
time: OK
Score: 140/140
```

测试无误后上传至 github 对应分支。

Lab11: Networking

通过以下命令切换实验分支，获取实验资源：

```
git fetch
```

```
git checkout net
```

```
make clean
```

Your Job (hard)

实验目的

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。

`e1000` 是 Intel 的千兆以太网控制器，`xv6` 通过驱动程序与该硬件交互，实现数据包的发送（TX）和接收（RX）。驱动核心依赖描述符环（Descriptor Ring）机制：发送描述符环（TX Ring）：内核将待发送的数据包信息（地址、长度、命令）写入描述符，硬件从环中读取并发送，发送完成后更新描述符状态；接收描述符环（RX Ring）：内核预先分配空缓冲区（mbuf）并写入接收描述符，硬件接收数据包后将数据写入缓冲区并更新状态，内核读取状态后处理数据包。

实验步骤

1. 实现 `e1000_transmit`（发送数据包）

该函数将用户 / 内核的数据包（mbuf 格式）写入 TX 描述符环，触发硬件发送，核心逻辑是 “检查描述符可用性→填充描述符→更新硬件指针”。

```
int e1000_transmit(struct mbuf *m) {
    // 加锁：避免多线程并发访问 TX 环导致数据混乱
    acquire(&e1000_lock);
    // 1. 获取当前可用的 TX 描述符索引（从 E1000_TDT 寄存器读取尾部指针）
    uint32 next_idx = regs[E1000_TDT]; // TDT: Transmit Descriptor Tail
    struct e1000_tx_desc *tx_desc = &tx_ring[next_idx];
    // 2. 检查描述符是否可用（DD 位：Descriptor Done, 1 表示硬件已处理完该描述符）
    if ((tx_desc->status & E1000_TXD_STAT_DD) == 0) {
        release(&e1000_lock); // 发送队列满，释放锁后返回失败
    }
}
```



```

    return -1;
}
// 3. 释放上一次发送遗留的 mbuf (若存在)
if (tx_mbufs[next_idx] != NULL) {
    mbuf_free(tx_mbufs[next_idx]);
}
// 4. 填充 TX 描述符: 数据包地址、长度、命令
tx_desc->addr = (uint64)m->head;    // mbuf 头部地址 (数据包起始)
tx_desc->length = (uint16)m->len;    // 数据包长度 (以太网帧长度)
// 命令: EOP (End of Packet, 标记帧结束) + RS (Report Status, 发送后更新状态)
tx_desc->cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;
// 5. 保存 mbuf 指针 (发送完成后需释放)
tx_mbufs[next_idx] = m;
// 6. 更新 TX 环尾部指针 (硬件从 TDT 到 TDH 之间读取描述符)
regs[E1000_TDT] = (next_idx + 1) % TX_RING_SIZE;
// 解锁: 允许其他线程访问 TX 环
release(&e1000_lock);
return 0; // 发送成功
}

```

2. 实现 e1000_recv (接收数据包)

该函数从 RX 描述符环读取硬件接收的数据包，将数据封装为 mbuf 传递给网络栈 (net_rx)，核心逻辑是 “检查描述符状态→处理数据包→重置描述符”。

```

static void e1000_recv(void) {
    // 1. 计算第一个待处理的 RX 描述符索引 (RDT+1, RDT: Receive Descriptor Tail)
    uint32 next_idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    struct e1000_rx_desc *rx_desc = &rx_ring[next_idx];
    // 2. 循环处理所有已接收数据包的描述符 (DD 位为 1 表示有新数据)
    while (rx_desc->status & E1000_RXD_STAT_DD) {
        // 检查数据包长度是否超过 mbuf 最大容量 (避免溢出)
        if (rx_desc->length > MBUF_SIZE) {
            panic("e1000_recv: packet too large (exceeds MBUF_SIZE)");
        }
        // 3. 填充 mbuf: 设置数据包长度
        struct mbuf *m = rx_mbufs[next_idx];
        m->len = rx_desc->length;
        // 4. 将 mbuf 传递给网络栈 (处理以太网帧、IP 包等)
        net_rx(m);
        // 5. 分配新 mbuf, 重置 RX 描述符 (为下一次接收做准备)
        struct mbuf *new_m = mbuf_alloc(0); // 分配空 mbuf
        if (new_m == NULL) {

```



```
lh01@lh:~/xv6-labs-2020$ make server
python3 server.py 26099
listening on localhost port 26099
Traceback (most recent call last):
  File "server.py", line 7, in <module>
    sock.bind(addr)
OSError: [Errno 98] Address already in use
make: *** [Makefile:319: server] Error 1
lh01@lh:~/xv6-labs-2020$ sudo lsof -i :26099
[sudo] password for lh01:
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF NODE NAME
python3 24072 lh01   3u   IPv4 113878      0t0  UDP localhost:26099
lh01@lh:~/xv6-labs-2020$ kill -9 24072
```

实验心得

本次实验深入理解了网络设备驱动的核心机制。硬件与软件的协作模式：描述符环是硬件与内核的“通信桥梁”——内核通过填充描述符告知硬件“发送什么 / 接收数据存哪里”，硬件通过更新描述符状态告知内核“发送完成 / 有新数据”，这种异步协作模式避免了 CPU 轮询，提升了效率；锁与并发安全：e1000 驱动可能被多线程（如多个进程发送数据包）访问，e1000_lock 确保对描述符环的操作原子性，避免并发写导致的描述符混乱，这之前实验中“桶级锁”“VMA 锁”的设计思想一致，均是通过锁保护共享资源。

驱动程序将硬件接收的原始数据包（以太网帧）传递给网络栈，网络栈再进行帧解析、IP 转发等上层处理，这种分层设计让我更清晰地认识到“硬件 - 驱动 - 内核 - 应用”的完整数据流转链路。

Lab11 的实验整体测试(./grade-lab-net)截图：

time.txt 记录实验用时，运行 ./grade-lab-net 进行整体测试，得分如下：

```
lh01@lh:~/xv6-labs-2020$ ./grade-lab-net
make: 'kernel/kernel' is up to date.
== Test running nettests == (3.0s)
== 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
```

测试无误后上传至 github 对应分支。

项目源码 github 仓库链接:

<https://github.com/Effulgence12/OS-design-XV6-Operation-System.git>