



2024-OS-summer_project-TongjiUniversity

刘继业 2252752 Tongji University, 2024 Summer, Du BoWen 2024/8/26-8/30

代码: https://github.com/salad14/2024-OS-summer_project-TongjiUniversity

各个实验的代码可切换不同的Branch查看

Tools

在Windows上安装WSL

下载[Windows Subsystem for Linux](#) 和 [Ubuntu on Windows](#)。

在Windows中, 您可以访问 `\\wsl$` 目录下的所有WSL文件。例如, Ubuntu 20.04安装的主目录应该位于 `\\wsl$\\Ubuntu-20.04\\home\\<username>`。

```
C:\Users\17912>wsl --list --verbose
```

NAME	STATE	VERSION
* Ubuntu	Stopped	2

软件源更新和环境准备

确保上一步已完成

启动Ubuntu, 运行下列代码

```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu bi
```

测试安装

如果安装正常, 运行下列命令行会显示如下的内容

```
$ qemu-system-riscv64 --version
QEMU emulator version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.22)
Copyright (c) 2003-2021 Fabrice Bellard and the QEMU Project developers
```

```
$ riscv64-linux-gnu-gcc --version
riscv64-linux-gnu-gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 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.
```

编译内核

下载XV6的源码

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

Guidance

调试提示

- 1.理解 C 和指针：建议阅读《C 程序设计语言》（第二版）并做指针练习。如果不熟悉 C 的指针，会在实验中遇到很大困难。特别注意几个常见的指针习惯用法，比如 `int p = (int)100` 和 `p[i]` 等。
- 2.Git 使用：代码部分运行正常时，请用 Git 提交，以便出错时能回滚到之前的状态。
- 3.通过插入打印语句理解代码行为。如果输出太多，使用 `script` 命令记录输出并进行搜索。
- 4.使用 GDB 调试 xv6：在一个窗口运行 `make qemu-gdb`，在另一个窗口运行 `gdb`，设置断点并继续执行。当程序崩溃时，可以用 `bt` 获取回溯信息。如果内核崩溃或挂起，使用 GDB 查找问题所在。
- 5.QEMU 监控器：按 `Ctrl-A C` 进入 QEMU 控制台，使用 `info mem` 查询页面表。可用 `cpu` 命令选择核，或用 `make CPUS=1 qemu` 只启动一个核。

Lab: Xv6 and Unix utilities

本实验将使您熟悉 xv6 及其系统调用。

Boot xv6

1. 获取实验室的 xv6 源代码并签出 util 分支：

```
$ git clone git://g.csail.mit.edu/xv6-labs-2021
Cloning into 'xv6-labs-2021'...
...
$ cd xv6-labs-2021
$ git checkout util
```

```
salad14@Salad:~$ git clone git://g.csail.mit.edu/xv6-labs-2021
Cloning into 'xv6-labs-2021'...
remote: Enumerating objects: 7051, done.
remote: Counting objects: 100% (7051/7051), done.
remote: Compressing objects: 100% (3423/3423), done.
remote: Total 7051 (delta 3702), reused 6830 (delta 3600), pack-reused 0
Receiving objects: 100% (7051/7051), 17.20 MiB | 2.99 MiB/s, done.
Resolving deltas: 100% (3702/3702), done.
warning: remote HEAD refers to nonexistent ref, unable to checkout.

salad14@Salad:~$ ls
xv6-labs-2021
salad14@Salad:~$ cd xv6-labs-2021/
salad14@Salad:~/xv6-labs-2021$ ls
salad14@Salad:~/xv6-labs-2021$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

xv6-labs-2021 仓库与本书的仓库略有不同 Xv6-RISCV;它主要添加一些文件。如果你好奇，可以通过下列命令查看git日志：

```
$ git log
```

Git 允许您跟踪对代码所做的更改。例如，如果您完成了其中一项练习，并且想要检查您的进度，您可以提交更改通过运行：

```
$ git commit -am 'my solution for util lab exercise 1'
Created commit 60d2135: my solution for util lab exercise 1
1 files changed, 1 insertions(+), 0 deletions(-)
```

2.构建并运行xv6

```
$ make qemu
riscv64-linux-gnu-gcc -c -o kernel/entry.o kernel/entry.S
.....
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive

xv6 kernel is booting

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

如果在提示符下键入 ls，则应看到类似的输出 到以下内容：

```
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
xargstest.sh 2 3 93
cat        2 4 23656
echo       2 5 22488
forktest   2 6 13232
grep       2 7 26808
init        2 8 23312
kill        2 9 22424
ln          2 10 22272
ls          2 11 25824
mkdir       2 12 22544
rm          2 13 22528
sh          2 14 40552
stressfs    2 15 23520
usertests   2 16 150240
grind       2 17 37032
wc          2 18 24624
zombie      2 19 21800
console     3 20 0
```

这些是 mkfs 包含在 初始文件系统;大多数是您可以运行的程序。我们刚刚运行了其中一个：ls。

xv6没有ps命令，但是如果输入 `Ctrl-p`，内核会打印每个进程的信息。如果你现在尝试一下，你会看到两行：一行是init，一行是sh。

要退出 qemu，请键入： `Ctrl-a x`。

Sleep

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

1.在开始编码之前，请阅读 [xv6 book](#)的第1章，并查看 `user/` 中的其他程序（例如 `user/echo.c`、`user/grep.c` 和 `user/rm.c`），了解如何获取传递命令行参数给程序。

2.在命令行中，您可以运行以下命令来打开文件并查看其内容：`$ vim user/echo.c` 可以使用任何文本编辑器打开文件，例如 Vim、Nano、Gedit 等。以Vim为例，在 Vim 编辑器中打开文件后，要退出并返回终端命令行界面，可以按照以下步骤操作：

1. 如果您处于编辑模式（**Insert Mode**），请按下 **Esc** 键，以确保切换到正常模式（**Normal Mode**）。

2. 在正常模式下，按 **:** 进入命令模式：

保存文件并退出：**wq** 或 **:x**

只保存文件：**:w**

退出且不保存：**:q!**

3.使用 ``cat`` 命令来显示文件的内容。例如，运行以下命令来查看文件内容：``cat user/echo.c``
``cat`` 命令会将文件的内容直接输出到终端。

4.使用 `less` 命令：``less`` 命令是一个分页查看器，用于逐页查看文件内容（使用空格键向下翻页，使用 **b** 键向上翻页）。

5.总结：

进入插入模式：**i**、**a**、**o**

退出插入模式：**Esc**

保存并退出：**:wq**

不保存退出：**:q!**

删除一行：**dd**

复制一行：**yy**

粘贴：**p**

查找：**/keyword**

撤销：**u**

3. 通过 `kernel/sysproc.c` 中的 `sys_sleep` 获取实现 `sleep` 系统调用的xv6内核代码；通过 `user/user.h` 获取可从用户程序调用 `sleep` 的C语言定义；通过 `user/usys.S` 获取从用户代码跳转到内核以实现 `sleep` 的汇编代码。

4.在程序中使用系统调用 `sleep`，其中命令行参数以字符串形式传递，可以使用 `atoi`（参见 `user/ulib.c`）将其转换为整数。最后，需要确保 `main` 调用 `exit()` 以退出程序。此外如果用户忘记传递参数，`sleep`应打印错误信息。

使用vim编辑器修改sleep程序：

```
int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(2, "need a param");
        exit(1);
    }
    int time = atoi(argv[1]);
    sleep(time);
    exit(0);
}
```

5.将编写好的睡眠程序添加到 Makefile 的 UPROGS 中;让 QEMU 编译你的程序，能够从 xv6 shell 运行它。

添加 `sleep` 目标程序：输入命令行 `$ vim Makefile` 打开 Makefile 文件，在 Makefile 中找到名为 `UPROGS` 的行，这是一个定义用户程序的变量。在 `UPROGS` 行中，添加 `sleep` 程序的目标名称：`$U/_sleep\`。

编译运行程序：在终端中，运行 `make qemu` 命令编译 xv6 并启动虚拟机，随后通过测试程序来检测sleep程序的正确性。

从xv6 shell运行程序：

```
$ make qemu
...
init: starting sh
$ sleep 10
(nothing happens for a little while)
$
```

```
salad14@Salad:~/xv6-labs-2021$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.6s)
== 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

为了使用sleep函数，需要包含相关头文件，我在阅读 `user/user.h` 等头文件并结合控制台报错信息来确定需要在 `main` 函数中使用的头文件，最终成功调用 `sleep` 函数。

实验心得

实验要求我们阅读相关的代码，正确调用需要程序依赖相关的文件，理清参数传递和头文件依赖关系，避免参数传递出错或者头文件缺少导致的错误。还需要让系统支持sleep的调用，添加的makefile里来正确编译

pingpong

实验目的

编写一个使用 UNIX 系统调用 "ping-pong" 的程序 一对管道上两个进程之间的字节，每个进程对应一个进程 方向。 父级应向子级发送一个字节; 孩子应打印“: 已接收ping”，其中 是其进程 ID， 将管道上的字节写入父级， 并退出; 父级应从子级读取字节， 打印 “: Received Pong”， 并退出。 你 解决方案应该在文件 `user/pingpong.c` 中。

实验步骤

1.创建管道


```
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);
pipe(p2);
```

2.使用fork创建子管道

3.使用read读管道数据, write写管道数据

4.使用getpid () 查找调用进程的id

5.重复之前的步骤并编译

6.运行并测试, 在xv6 shell中运行该程序, 输出结果如下:

```
$ pingpong
4: received ping
3: received pong
$
```

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);
    pipe(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);
    }
}

```

```
salad14@Salad:~/xv6-labs-2021$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (1.3s)
```

实验中遇到的问题

在最开始的代码中，我没有使用wait（）函数使父进程等待子进程，没有采用这样的同步机制有可能导致二者的冲突。后来我在仔细的阅读了实验要求后发现其对这个进行了规则，因此需要加上这个，否则会导致子进程失去父进程不能导致进一步的处理

实验心得

本次实验让我认识到进程间通信在多进程编程中的重要性。管道作为通信机制，可以在父进程和子进程之间传递数据，实现数据的共享和交换。我们需要加入一些同步机制来同步进程的顺序，确保数据能正确交换和打印顺序。

primes

实验目的

使用管道编写 Prime Sieve（主筛）的并发版本。这个想法这要归功于 Unix 管道的发明者 Doug McIlroy。

学习使用pipe和fork来设置管道。第一个进程将数字2到35输入管道。对于每个素数创建一个进程，该进程通过一个管道从左边的邻居读取数据，并通过另一个管道向右边的邻居写入数据。由于xv6的文件描述符和进程数量有限，第一个进程可以在35处停止。

实验步骤

- 1.创建父进程，父进程将数字2到35输入管道，此时不必创建其后所有进程，每一步迭代更新一对相对的父子进程（仅在需要时才创建管道中的进程）。
- 2.对于2-35中的每个素数创建一个进程，进程之间需要进行数据传递：该进程通过一个管道从左边的父进程读取数据，并通过另一个管道向右边子进程写入数据。

- 3.对于每一个生成的进程而言，当前进程最顶部的数即为素数；对每个进程中剩下的数进行检查，如果是素数则保留并写入下一进程，如果不是素数则跳过。
- 4.完成数据传递或更新时，需要及时关闭一个进程不需要的文件描述符（防止程序在父进程到达35之前耗尽xv6的资源）。
- 5.在数据传递的过程中，父进程需要等待子进程的结束，并回收共享的资源和数据等，即一旦第一个进程到达35，它应该等待直到整个管道终止，包括所有子进程、孙进程等。因此，主primes进程应该在所有输出都打印完毕，并且所有其他primes进程都退出后才退出。
- 6.编译程序，并且测试运行

源代码

```

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

void
sieve(int *left_pipe)
{
    int prime;
    if (read(left_pipe[0], &prime, sizeof(int)) == 0) {
        // 如果没有读取到内容，说明管道关闭了
        close(left_pipe[0]);
        exit(0);
    }

    printf("prime %d\n", prime);

    int right_pipe[2];
    pipe(right_pipe);

    if (fork() == 0) {
        // 子进程：继续处理筛选下一个质数
        close(right_pipe[1]); // 关闭子进程的写端
        sieve(right_pipe);
    } else {
        // 父进程：继续筛选
        close(right_pipe[0]); // 关闭父进程的读端
        int num;
        while (read(left_pipe[0], &num, sizeof(int)) > 0) {
            if (num % prime != 0) {
                write(right_pipe[1], &num, sizeof(int));
            }
        }
        close(left_pipe[0]);
        close(right_pipe[1]);
        wait(0); // 等待子进程结束
        exit(0);
    }
}

```

```

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

    if (fork() == 0) {
        // 子进程：启动筛选
        close(left_pipe[1]); // 关闭子进程的写端
        sieve(left_pipe);
    } else {
        // 父进程：向管道写入数字 2 到 35
        close(left_pipe[0]); // 关闭父进程的读端
        for (int i = 2; i <= 35; i++) {
            write(left_pipe[1], &i, sizeof(int));
        }
        close(left_pipe[1]); // 关闭写端，通知子进程没有更多数据了
        wait(0); // 等待子进程结束
        exit(0);
    }

    return 0;
}

```

```

salad14@Salad:~/xv6-labs-2021$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.4s)

```

实验中遇到的问题

fork()系统函数用于创建一个新的进程（子进程）作为当前进程（父进程）的副本，子进程会继承父进程的代码、数据、堆栈和文件描述符等资源的副本。子进程和父进程在 fork() 调用点之后的代码是独立执行的，并且拥有各自独立的地址空间。因此，父进程和子进程可以在 fork() 后继续执行不同的逻辑，实现并行或分支的程序控制流程。

在这个过程中，上一级的子会变成下一级的父，因此会能够解决其中的父子继承的关系。

实验心得

在这个函数中，我对管道数据传输和fork（）函数的调用有了新的理解。需要注意处理管道读写端的关闭和父子进程之间的关系，我对于并发编程有了深入的理解

find

实验目的

编写一个简单版本的UNIX查找程序：查找所有文件 在具有特定名称的目录树中。您的解决方案应位于文件 user/find.c 中。

实验步骤

- 1.首先查看user/lis.c以了解如何读取目录。
- 2.main 函数完成参数检查和功能函数的调用：检查命令行参数的数量，如果参数数量小于 3，则输出提示信息并退出程序。否则，将第一个参数作为路径，第二个参数作为要查找的文件名称，调用 find 函数进行查找，并最后退出程序。
- 3.编写一个fmtname函数，通过查找路径中最后一个 '/' 后的第一个字符来获取文件的名称部分，然后与给定的名称进行比较。如果匹配则返回文件名。

```

char* fmtname(char *path) {
    static char buf[DIRSIZ+1];
    char *p;

    // 找到最后一个斜杠后的部分
    for(p = path + strlen(path); p >= path && *p != '/'; p--)
        ;
    p++;

    // 返回文件名
    if(strlen(p) >= DIRSIZ)
        return p;
    memmove(buf, p, strlen(p));
    buf[strlen(p)] = 0;
    return buf;
}

```

4.编写find函数，用于递归的在给定的目录中查找带有特定名称的文件

如果是 `T_FILE` 则使用 `fmtname` 函数检查文件名称是否与给定的名称匹配，如果匹配则打印文件的路径。

如果是 `T_DIR` 目录。先检查是否超过缓冲区大小，不超过将路径拷贝在 `buf` 中并添加 `/`。接下来，循环读取目录中的每个文件信息，注意不要递归到 `'.'` 和 `'..'`，跳过当前目录 `'.'` 和上级目录 `'..'`。将文件名拷贝到 `buf` 中，并使用递归调用 `find` 函数在子目录中进行进一步查找。

5.编译后运行

实验中遇到的问题

在最开始时，我忽视了 `ls.c` 的输入仅仅需要一个文件地址参数，文件查找需要路径和文件名两个参数，没有对函数的输入进行对应的修改。导致报错，在我更改函数的输入之后这一点得到了解决

实验心得

通过本次 `find` 实验的编写，我深入理解了文件系统中目录和文件之间的关系，学会了使用递归算法对目录树的深度遍历，并加强了自己在修正代码的能力，提高了对文件系统的理解与应用

xargs

实验目的

编写一个UNIX xargs程序的简单版本：从标准输入中读取行，并为每一行运行一个命令，将行作为参数提供给命令。解决方案位于文件user/xargs.c中。

实验步骤

1.通过示例理解xarg的行为：

此处的命令是“echo bye”和附加的 参数是“你好也是”，使命令“echo bye 你好也”， 输出“再见你好”。

```
$ echo hello too | xargs echo bye
bye hello too
$
```

请注意，UNIX 上的 xargs 会进行优化，它将一次向命令提供多个参数。我们不希望您进行此优化。要使 UNIX 上的 xargs

```
$ echo "1\n2" | xargs -n 1 echo line
line 1
line 2
$
```

2.程序从标准输入中读取数据，每次读取一行，并将其作为参数传递给命令。

3.使用 fork() 创建子进程，然后在子进程中使用 exec() 执行命令，并将输入的行作为命令的附加参数。父进程则使用 wait() 等待子进程完成后再处理下一行输入。

```

while((n = read(0, buf, sizeof(buf))) > 0) {
    int m = 0;
    while(m < n) {
        int start = m;
        while(m < n && buf[m] != '\n') {
            m++;
        }
        buf[m] = 0; // 将换行符替换为字符串结束符
        args[base_args] = buf + start; // 将当前行作为参数添加
        args[base_args + 1] = 0; // 确保 args 以 NULL 结尾

        if(fork() == 0) { // 创建子进程
            exec(args[0], args); // 执行命令
            exit(0); // 防止 exec 失败后继续执行
        } else {
            wait(0); // 等待子进程完成
        }
        m++;
    }
}

```

4.args 数组用于存储命令和附加的参数。初始的命令参数从 argv 中复制过来，标准输入的每一行被追加到 args 中作为命令参数。

5.编译运行

实验中遇到的问题

在最开始，由于我在最开始加了一个条件判断，导致有的时候无法读取到子文件，没能正确的输出hello。在查找到问题所在之后，我成功的处理了exec失败后继续执行的问题，正确的处理了文件之间的关系

实验心得

实验要求将输入按照空格拆分为多个参数，并将它们作为命令行参数传递给外部命令。我学会了如何处理命令行中的输入字符串，跳过空格，并将参数存储在适当的数据结构中。通过exec函数执行外部命令，我了解了进程创建替换的过程，学会了如何调试程序。

```
== Test sleep, no arguments ==  
$ make qemu-gdb  
sleep, no arguments: OK (3.0s)  
== Test sleep, returns ==  
$ make qemu-gdb  
sleep, returns: OK (1.0s)  
== Test sleep, makes syscall ==  
$ make qemu-gdb  
sleep, makes syscall: OK (0.8s)  
== Test pingpong ==  
$ make qemu-gdb  
pingpong: OK (1.0s)  
== Test primes ==  
$ make qemu-gdb  
primes: OK (1.4s)  
== Test find, in current directory ==  
$ make qemu-gdb  
find, in current directory: OK (0.8s)  
== Test find, recursive ==  
$ make qemu-gdb  
find, recursive: OK (1.0s)  
== Test xargs ==  
$ make qemu-gdb  
xargs: OK (1.4s)  
== Test time ==  
time: OK  
Score: 100/100
```

Lab: system calls

在本实验中，您将向 XV6 添加一些新的系统调用，这将有所帮助 您了解它们的工作原理，并会

让您接触到其中的一些 XV6 内核的内部结构。稍后将添加更多系统调用实验室。

在开始编码之前，请阅读 [XV6 book](#)，以及 第 4 章第 4.3 和 4.4 节，以及 相关源文件：

系统调用的用户空间代码是在 `user/user.h` 和 `user/usys.pl` 中。

内核空间代码是 `kernel/syscall.h`、`kernel/syscall.c`。

与进程相关的代码是 `kernel/proc.h` 和 `kernel/proc.c`。

实验前准备

若要启动实验室，请切换到 `syscall` 分支：

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

System call tracing

实验目的

旨在帮助了解系统调用跟踪的实现，以及如何修改 xv6 操作系统以添加新功能。我们需要添加一个有助于调试的新的 trace 系统调用。该功能包括创建一个名为 `trace` 的系统调用，并将整数 `"mask"` 作为参数。 `"mask"` 的位数表示要跟踪哪些系统调用。通过实验，熟悉内核级编程，包括修改进程结构、处理系统调用和管理跟踪掩码。

实验步骤

首先，我们需要声明新的 `trace` 系统调用：

1. 在 `kernel/syscall.h` 中声明系统调用号：在 `kernel/syscall.h` 中为 `SYS_trace` 添加一个新的条目。选择一个未使用的系统调用号

```
#define SYS_trace 22
```

2. 在 `user/user.h` 中声明 `trace` 系统调用的原型：

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

3.在 user/usys.pl 中添加 trace 系统调用的条目：

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

接下来，我们需要在内核中实现处理 trace 系统调用的函数：

4.实现 sys_trace 函数，将 trace 掩码存储在进程的 proc 结构中：

```
uint64
sys_trace(void)
{
    int mask;

    if(argint(0, &mask) < 0)
        return -1;

    struct proc *p = myproc();
    p->trace_mask = mask;

    return 0;
}
```

5.在 proc 结构体中添加一个新的字段 trace_mask，用于存储 trace 掩码：

```
int trace_mask; // 用于存储 trace 掩码的新字段
```

6.在 fork 函数中，将 trace_mask 从父进程复制到子进程：

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

7.在 kernel/syscall.c 中添加系统调用名称数组：

定义一个系统调用名称数组，方便打印系统调用的名称：

```
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"
};
```

8.更新 syscall 函数，添加打印跟踪信息的逻辑：

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

    num = p->trapframe->a7;

    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        p->trapframe->a0 = syscalls[num]();
        // 打印跟踪信息
        if ((1 << num) & p->trace_mask) {
            printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num],
                p->trapframe->a0);
        }
    }
    else
    {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}
```

9.编译运行

```
salad14@Salad:~/xv6-labs-2021$ ./grade-lab-syscall trace
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 (1.4s)
    (Old xv6.out.trace_all_grep failure log removed)
== Test trace nothing == trace nothing: OK (0.5s)
== Test trace children == trace children: OK (11.4s)
```

实验中遇到的问题

在最开始我编写完成后，出现了无法追踪全部系统调用的问题。后来发现问题出在没能正确的传递参数和数据上面，在查阅了相关的文档之后，解决了这个问题，避免了传参错误而导致的锁没能正确释放的问题。

实验心得

从这个实验中我学会了如何在xv6内核中添加新的系统调用，修改进程控制块和支持跟踪掩码。同时对于参数传递有了新的理解。

Sysinfo

实验目的

在本实验中将添加一个系统调用 sysinfo，用于收集运行系统的信息。系统调用需要一个参数：指向 struct sysinfo 的指针（参见kernel/sysinfo.h）。内核应填写该结构体的字段：freemem 字段应设置为可用内存的字节数，nproc 字段应设置为状态不是 UNUSED 的进程数。

实验步骤

1. 声明和定义系统调用

在 kernel/syscall.h 中为 SYS_sysinfo 添加一个新的系统调用编号：

```
#define SYS_sysinfo 23
```

由于 sysinfo 函数需要使用 struct sysinfo，我们需要在 user/user.h 中预先声明这个结构体，并添加 sysinfo 的函数原型：

```
struct sysinfo;

int sysinfo(struct sysinfo *);
```

在 user/usys.pl 中添加 sysinfo 的条目：

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

2.在内核中实现 sysinfo 系统调用，完成数据收集并将信息返回给用户空间。

在 kernel/sysproc.c 文件中实现 sysinfo 函数：

```
uint64
sys_sysinfo(void)
{
    struct sysinfo info;
    struct sysinfo *uinfo;

    if(argaddr(0, (uint64 *)&uinfo) < 0)
        return -1;

    // 获取空闲内存数
    info.freemem = free_mem();

    // 获取正在使用的进程数
    info.nproc = num_procs();

    // 将信息复制到用户空间
    if(copyout(myproc()->pagetable, (uint64)uinfo, (char *)&info, sizeof(info)) < 0)
        return -1;

    return 0;
}
```

在 kernel/kalloc.c 中实现一个函数，用于返回当前空闲的内存字节数：


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

在 kernel/proc.c 中实现一个函数，用于返回当前正在使用的进程数：

```
int
num_procs(void)
{
    struct proc *p;
    int count = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
        if(p->state != UNUSED)
            count++;
    }
    return count;
}
```

3.在Makefile中添加

\$U/_sysinfotest

4.编译调试运行

实验中遇到的问题

本次实验的难点在于从系统中收集运行的信息。我参考kalloc () 和kfree () 函数，学习得知内核通过 kmem.freelist 的一个链表维护未使用的内存，然后让链表每个结点对应的页表大小相乘得到可用内存数。在编写的过程中，我疏忽了外部调用的声明，给编写造成了麻烦。

实验心得

本次实验我成功为系统添加了一个新的系统调用 `sysinfo`，实现了收集运行系统的信息。在完成的过程中，我学会了分析问题，参考其他函数来收集自己需要的信息，同时也学会了查阅资料，深入研究。这次实验教育我在编写代码的时候需要细心，不要忽视外部调用需要声明。

```
== Test trace 32 grep ==  
$ make qemu-gdb  
trace 32 grep: OK (2.8s)  
== Test trace all grep ==  
$ make qemu-gdb  
trace all grep: OK (1.0s)  
== Test trace nothing ==  
$ make qemu-gdb  
trace nothing: OK (1.1s)  
== Test trace children ==  
$ make qemu-gdb  
trace children: OK (12.6s)  
== Test sysinfotest ==  
$ make qemu-gdb  
sysinfotest: OK (2.0s)  
== Test time ==  
time: OK  
Score: 35/35
```

Lab: page tables

在本实验中，您将浏览页表并将其修改为 加快某些系统调用的速度并检测已访问的页面。

`kern/memlayout.h`，用于捕获内存的布局。

`kern/vm.c`，其中包含大多数虚拟内存（VM）代码。

`kernel/kalloc.c`，其中包含用于分配和释放物理内存。

若要启动实验室，请切换到 pgtbl 分支：

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

Speed up system calls

实验目的

某些操作系统（例如 Linux）通过共享来加快某些系统调用的速度 用户空间和内核之间的只读区域中的数据。这消除了 执行这些系统调用时需要内核交叉。为了帮助您学习 如何将映射插入到页表中，您的首要任务就是实现这一点 对 xv6 中的 getpid () 系统调用进行了优化。

在 xv6 操作系统中优化 getpid() 系统调用，通过在用户空间和内核之间共享一个只读页面，从而减少内核切换的开销。具体而言，当每个进程被创建时，需要在 USYSCALL (memlayout.h 中定义的虚拟地址) 映射一个只读页面，并在页面的起始处存储一个 struct usyscall 结构体（也在 memlayout.h 中定义），并将其初始化为存储当前进程的 PID。实验要求的用户空间函数 ugetpid() 已经提供，并会自动使用 USYSCALL 映射。

实验步骤

1.在 kernel/proc.h 的proc 结构体中添加指针来保存这个共享页面的地址。

```
struct usyscall *usyscallpage;
```

2.在函数allocproc()中，为每一个新创建的进程分配一个只读页，使用 mappages() 来创建页表映射。

3.将 struct usyscall 结构放置在只读页的开头，并初始化其存储当前进程的 PID：

在 kernel/proc.c 的 proc_pagetable(struct proc *p) 中将这个映射（PTE）写入 pagetable 中。

4.在 kernel/proc.c 的 freeproc() 函数中，释放之前分配的只读页。

5.编译运行

```
salad14@Salad:~/xv6-labs-2021$ ./grade-lab-pgtbl ugetpid
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (1.6s)
== Test    pgtbltest: ugetpid ==
    pgtbltest: ugetpid: OK
```

实验中遇到的问题

在进行页面映射时，我遇到了编译出现的问题：

```
xv6 kernel is booting
```

```
panic: release
```

表明在某个地方，代码尝试释放一个未被持有的锁。后来经过调试发现，我在某处错误的使用了 `freeproc()` 释放了锁，后面又手动释放了锁，因此出现了冲突问题，在我修改了这个问题后得以解决

实验心得

这个实验使我更深入地理解了系统调用的工作原理以及它们是如何在用户空间和内核空间之间进行通信的。通过在每个进程的页表中插入只读页，我掌握了页表操作的方法，实现了用户空间和内核空间的共享

Print a page table

实验目的

本实验的目标是编写一个函数 `vmprint()`，用于打印 RISC-V 页表的内容。通过这个实验，您将能够可视化页表的结构，并为以后的调试提供帮助。

实验步骤

1. 在 `kernel/vm.c` 文件中定义 `vmprint()` 函数，该函数接受一个 `pagetable_t` 参数，并以特定格式打印该页表的内容。

```

void
vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable);
    vmprint_walk(pagetable, 0, 0);
}

void
vmprint_walk(pagetable_t pagetable, int depth, uint64 va)
{
    for (int i = 0; i < 512; i++) {
        pte_t pte = pagetable[i];
        if (pte & PTE_V) {
            uint64 pa = PTE2PA(pte);
            for (int j = 0; j < depth; j++) {
                printf("  .");
            }
            printf("%d: pte %p pa %p\n", i, pte, pa);
            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
                pagetable_t next_level = (pagetable_t)PTE2PA(pte);
                vmprint_walk(next_level, depth + 1, va + (i << (12 + 9 * depth)));
            }
        }
    }
}

```

2. 插入 vmprint() 调用:

在 exec.c 文件中找到 exec() 函数，在 return argc 语句之前插入 if(p->pid == 1) vmprint(p->pagetable); 代码，确保在初始化进程执行完 exec() 之后打印该进程的页表。

3. 实现 vmprint() 函数:

使用递归遍历页表的每一层，并按照缩进的格式打印页表项（PTE）。只打印有效的 PTE，忽略无效的 PTE。

```

void
vmprint_walk(pagetable_t pagetable, int depth, uint64 va)
{
    for (int i = 0; i < 512; i++) {
        pte_t pte = pagetable[i];
        if (pte & PTE_V) { // 仅打印有效的 PTE
            uint64 pa = PTE2PA(pte);

            // 为每一层页表项打印适当的缩进
            for (int j = 0; j <= depth; j++) {
                printf("  .");
            }

            printf("%d: pte %p pa %p\n", i, pte, pa);

            // 如果 PTE 指向下一级页表，递归打印
            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
                pagetable_t next_level = (pagetable_t)PTE2PA(pte);
                vmprint_walk(next_level, depth + 1, va + (i << (12 + 9 * depth)));
            }
        }
    }
}

```

4.使用 printf 打印 PTE 索引、PTE 位和物理地址，确保输出格式与实验要求一致。

5.编译运行

实验中遇到的问题

在最开始递归打印的时候，我没能正确处理好递归层级的问题，导致打印错误。后来我参考了 freewalk 函数的形式，成功的解决了问题。在理解页表的层次结构时，可能显得过于抽象，但通过观察源码、结合课程教材中的图 3-4 和解释 vmprint() 输出的页表内容，我可以更清晰地了解每个级别的页表是如何映射虚拟地址到物理地址的。

实验心得

通过本次实验，我可以清晰地通过 vmprint() 的输出来查看页表的层次结构，从根页表开始，逐

级向下指向不同级别的页表页，最终到达最底层的页表页，其中包含了实际的物理页框映射信息。

这个实验加深了我对页表结构的理解，学会了如何在内核中操作位操作和宏定义，以及如何通过递归遍历页表来打印出整个页表的内容。

Detecting which pages have been accessed

实验目的

一些垃圾回收器（一种自动内存管理形式）可以从中受益 从有关已访问（读取或写入）的页面的信息中。作为实验的一部分，您将向 Xv6 添加一个新功能，用于检测并报告此问题 通过检查 RISC-V 页表中的访问位，将信息发送到用户空间。每当 RISC-V 硬件页面行走器解析时，它都会在 PTE 中标记这些位 TLB 未命中。

你的工作是实现 `pgaccess()`，这是一个报告已访问页面函数。系统调用采用三个参数。首先，它需要检查的第一个用户页面的起始虚拟地址。其次，它需要检查的页数。最后，它将用户地址带到缓冲区进行存储 将结果转换为 Bitmask（一种每页使用一位的数据结构，其中 第一页对应于最低有效位）。您将收到完整的 如果 `PGCaccess` 测试用例在以下情况下通过，则为实验室的这一部分提供信用 运行 `pgtbltest`。

实验步骤

1. 在 `kernel/riscv.h` 中定义访问位 `PTE_A`。根据 RISC-V 手册，我们得知 `PTE_A` 位的值为 `0x040`

2. 解析系统调用 `sys_pgaccess()` 的三个参数

`start_va`: 第一个用户页面的起始虚拟地址。

`num_pages`: 要检查的页面数量。

`user_mask`: 一个用户空间的地址，用于存储结果的位掩码。

通过 `argaddr()` 和 `argint()` 函数来解析传递进来的参数。如果参数解析失败，返回 -1 表示错误。

```

uint64 start_va;
int num_pages;
uint64 user_mask;

// 解析系统调用的参数
if(argaddr(0, &start_va) < 0)
    return -1;
if(argint(1, &num_pages) < 0)
    return -1;
if(argaddr(2, &user_mask) < 0)
    return -1;

```

3.为了避免处理过大的内存范围，函数限制可以检查的最大页面数量。这里设定最大检查的页面数为 64。如果请求的页面数超过这个限制，则返回 -1。

```

if(num_pages > 64)
    return -1;

```

4.遍历指定范围的页面并检查访问位。

walk() 函数用于获取页表项（PTE），其中包含了虚拟地址到物理地址的映射信息。

如果 pte 为 0，说明没有找到对应的页表项，跳过该页面。

如果 pte 存在且其访问位（PTE_A）被设置，表示该页面已被访问。此时，将对应位（ $1L \ll i$ ）设置到 mask 中。

检查访问位后，清除 PTE_A 位，以便下次调用时可以正确检测新访问的页面。

```

for(int i = 0; i < num_pages; i++) {
    pte_t *pte = walk(p->pagetable, start_va + i * PGSIZE, 0);
    if(pte == 0)
        continue;

    // 检查访问位并更新掩码
    if(*pte & PTE_A) {
        mask |= (1L << i);
        // 清除访问位
        *pte &= ~PTE_A;
    }
}

```


5.将结果从内核空间复制到用户空间即可

6.实现完成，编译运行

实验中遇到的问题

在实验中如何清除PTE_A的访问位困扰了我很久，在多次查阅资料后得知，我可以采用检测pte的方式，如果pte为0，说明没有找到对应的页表项，跳过该页面。如果pte存在且其访问位（PTE_A）被设置，表示该页面已被访问。此时，将对应位（ $1L \ll i$ ）设置到mask中。检查访问位后，清除PTE_A位，以便下次调用时可以正确检测新访问的页面。我成功的解决了问题

实验心得

通过这个实验，我学习了操作系统的内存管理机制，包括页表的结构和作用；理解了如何为进程分配页表，映射虚拟地址到物理地址，以及如何使用页表权限来实现不同的访问控制。我学会了如何查看代码文档来解决问题的能力。

```
== Test pgtbltest ==  
$ make qemu-gdb  
(3.4s)  
== Test    pgtbltest: ugetpid ==  
    pgtbltest: ugetpid: OK  
== Test    pgtbltest: pgaccess ==  
    pgtbltest: pgaccess: OK  
== Test pte printout ==  
$ make qemu-gdb  
pte printout: OK (1.2s)  
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK  
== Test usertests ==  
$ make qemu-gdb  
(141.0s)  
== Test    usertests: all tests ==  
    usertests: all tests: OK  
== Test time ==  
time: OK  
Score: 46/46
```

Lab: traps

本实验探讨了如何使用陷阱实现系统调用。您将首先使用堆栈进行热身练习，然后您将实现用户级陷阱处理的示例。

若要启动实验室，请切换到陷阱分支：

```
$ git fetch  
$ git checkout traps  
$ make clean
```

RISC-V assembly

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

阅读 `call.asm` 中的 `g`，`f`，和 `main` 函数。

回答下面的问题：

Q. 01

Which registers contain arguments to functions? For example, which register holds 13 in main's call to `printf`?

在 RISC-V 架构中，函数调用的前8个参数是通过寄存器传递的，分别是 `a0` 到 `a7`。具体来说：

`a0`：第1个参数

`a1`：第2个参数

依此类推

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。

```
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
```

Q. 02

**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.)**

查看 `call.asm` 文件中的 `f` 和 `g` 函数可知，函数 `f` 调用函数 `g`；函数 `g` 使传入的参数加 3 后返回。

```

int f(int x) {
    e:    1141
    10:    e422
    12:    0800
    return g(x);
}
    14:    250d
    16:    6422
    18:    0141
    1a:    8082

```

编译器会进行内联优化，即一些编译时可以计算的数据会在编译时得出结果，而不是进行函数调用。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，所以对于 main 函数的汇编代码来说，其并没有调用函数 f 和 g，而是在运行之前由编译器对其进行了计算。

```

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:    7a050513      addi    a0,a0,1952 # 7c8 <malloc+0xe8>
    30:    00000097      auipc   ra,0x0
    34:    5f8080e7      jalr    1528(ra) # 628 <printf>
    exit(0);

```

Q. 03

At what address is the function printf located?

```

void
printf(const char *fmt, ...)
{
    628:    711d          addi    sp, sp, -96

```

查阅得到其地址在 0x628。

Q. 04

What value is in the register ra just after the jalr to printf in main?

0: 使用 `auipc ra,0x0` 将当前程序计数器 pc 的值存入 ra 中。

34: `jalr 1536(ra)` 跳转到偏移地址 `printf` 处, 也就是 0x630 的位置。

根据 reference1 中的信息, 在执行完这句命令之后, 寄存器 ra 的值设置为 `pc + 4`, 也就是 return address 返回地址 0x38。即 `jalr` 指令执行完毕之后, ra 的值为 0x38。

30:	00000097	<code>auipc</code>	<code>ra, 0x0</code>
34:	5f8080e7	<code>jalr</code>	<code>1528(ra) # 628 <printf></code>
<code>exit(0);</code>			
38:	4501	<code>li</code>	<code>a0, 0</code>

Q. 05

Run the following code.

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

What is the output? [Here's an ASCII table]([ASCII Table - ASCII Character Codes, HTML, Octal, Hex, Decimal](#)) that maps bytes to characters.

The output depends on that fact that the RISC-V is little-endian. 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?

[Here's a description of little- and big-endian](#) and [a more whimsical description](#).

运行结果: 打印出了 He110 World。

`i` 的值是 0x00646c72, 它表示的字符串是 `rl`, 因为 RISC-V 是小端序, 所以字节顺序是反的。

57616 在十六进制中是 e110。

故输出 He110 World

Q. 06

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);
```

printf 函数期望有两个参数：x 和 y。

由于代码中只传递了一个参数 3，所以 y 部分的输出将是未定义的行为，可能会打印一些垃圾值，因为第二个参数 y 没有提供。

可能导致 y 打印出一个随机的值或导致程序崩溃。

Backtrace

实验目的

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

实验步骤

1. 在 kernel/printf.c 中实现 backtrace() 函数

backtrace() 函数需要遍历当前堆栈帧，找到每个调用者的返回地址并将其打印出来。

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

    while(fp != PGROUNDDOWN(fp)) {
        printf("%p\n", *(uint64*)(fp-8));
        fp = *(uint64*)(fp - 16);
    }
}
```

2. 为了在其他地方调用 backtrace()，我们需要在 kernel/defs.h 中添加其原型。

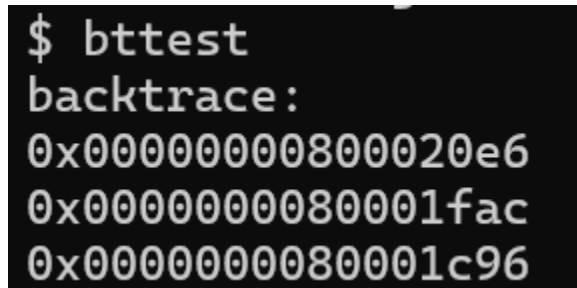
```
void backtrace(void);
```

3. `r_fp()` 函数用于获取当前函数的帧指针（即 `s0` 寄存器的值）。这是一个内联汇编函数，可以帮助我们获取当前栈帧指针。

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

4. 打开 `kernel/sysproc.c` 文件，并在 `sys_sleep()` 函数中调用 `backtrace()`。这样，我们可以测试 `backtrace()`。

5. 编译运行后，运行 `bttest`

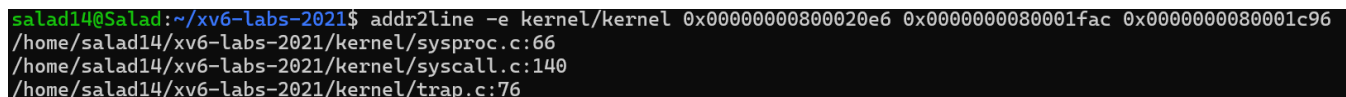


```
$ bttest
backtrace:
0x00000000800020e6
0x0000000080001fac
0x0000000080001c96
```

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

```
""bash
addr2line -e kernel/kernel 0x00000000800020e6 0x0000000080001fac 0x0000000080001c96
""
```

我们可以看到如下输出



```
salad14@Salad:~/xv6-labs-2021$ addr2line -e kernel/kernel 0x00000000800020e6 0x0000000080001fac 0x0000000080001c96
/home/salad14/xv6-labs-2021/kernel/sysproc.c:66
/home/salad14/xv6-labs-2021/kernel/syscall.c:140
/home/salad14/xv6-labs-2021/kernel/trap.c:76
```

7. 为了在内核崩溃时获取调用栈信息，可以在 `panic()` 函数中调用 `backtrace()`。

打开 `kernel/printf.c` 文件，找到 `panic()` 函数，并添加对 `backtrace()` 的调用。

实验中遇到的问题

起初，我在 `backtrace()` 函数中使用了错误的格式化字符串，导致地址打印不正确。通过调试，我意识到在内核环境下需要更加小心地处理打印格式，并确保栈帧的指针在遍历过程中始终指向有效的内存区域。

此外，通过使用 `addr2line` 工具将返回地址映射回源码行，我成功地验证了 `backtrace()` 输出的正确性。这使我进一步体会到工具在调试和分析代码中的重要性。

实验心得

在这次实验中，我实现了 `backtrace()` 函数，用于在 RISC-V 架构下的 xv6 操作系统中获取并打印当前的函数调用栈。当内核遇到错误或者需要调试时，`backtrace()` 函数可以帮助我们跟踪并分析函数调用的历史，定位问题的根源。通过这一实验，我对操作系统内核的栈帧结构、函数调用的工作机制以及如何调试内核有了更深入的理解。

在实现 `backtrace()` 过程中，我学习并理解了 RISC-V 架构下栈帧的布局和使用方式。每个栈帧保存了函数调用的返回地址和前一个栈帧的指针（帧指针 `fp`），这使得可以沿着帧指针链逐步回溯，找到调用链中的每个函数。这种理解对我来说是非常宝贵的，因为它不仅是操作系统内核中重要的概念之一，也是调试和分析软件问题的基础。

Alarm

实验目的

在本练习中，您将向 xv6 添加一个定期发出警报的功能 一个进程，因为它使用 CPU 时间。这对于计算受限可能很有用 想要限制它们消耗的 CPU 时间的进程，或者 想要计算但也希望定期进行一些计算的进程 行动。更一般地说，您将实现 用户级中断/故障处理程序;你可以使用类似的东西 例如，用于处理应用程序中的页面错误。您的解决方案 如果它通过了 `alarmtest` 和 `userTests`，则是正确的。

实验步骤

1.在 Makefile 中添加 `alarmtest`，使其可以作为用户程序进行编译。

2.定义 `sigalarm` 和 `sigreturn` 系统调用

在 `user/user.h` 中添加以下声明：


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

在 user/usys.pl 中添加以下行：

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

3.在 kernel/syscall.h 中添加 sigalarm 和 sigreturn 的系统调用号：

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

在 kernel/syscall.c 中添加对应的系统调用函数：

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

4.在kernel/proc.h, 将警报间隔和处理函数的指针存储在 proc 结构体的新字段中

```
uint64 handler_va;  
int alarm_interval;  
int passed_ticks;  
struct trapframe saved_trapframe;  
int have_return;
```

5.每次时钟中断发生时，硬件时钟会产生一个中断，这将在 usertrap() 函数中进行处理（位于 kernel/trap.c）。因此接下来需要修改 usertrap 函数，使得硬件时钟每滴答一次都会强制中断一次。

```

if(which_dev == 2) {
    struct proc *proc = myproc();
    if (proc->alarm_interval && proc->have_return) {
        if (++proc->passed_ticks == 2) {
            proc->saved_trapframe = *p->trapframe;
            proc->trapframe->epc = proc->handler_va;
            proc->passed_ticks = 0;
            proc->have_return = 0;
        }
    }
    yield();
}

```

6.完成第一步骤的工作后，我们可能遇到 alarmtest 在打印 "alarm!"后在 test0 或 test1 中崩溃，或者是 alarmtest（最终）打印 "test1 失败"，或者是 alarmtest 退出时没有打印 "test1 通过"。

因此，这一步涉及确保在处理完警报处理函数后，控制能够返回到被定时中断中断的用户程序指令处，同时保证寄存器的内容被恢复，以使用户程序能够在警报处理之后继续执行。最后，你需要在每次警报触发后 "重新激活" 定时器计数器，以便定时器定期触发处理函数的调用。

用户警报处理程序在完成后必须调用 sigreturn 系统调用。请看 alarmtest.c 中的 periodic 示例。这意味着您可以在 usertrap 和 sys_sigreturn 中添加代码，使用户进程在处理完警报后恢复正常。

我们可以编写一个符合要求的sys_sigreturn 函数：

```

uint64
sys_sigreturn(void)
{
    struct proc* proc = myproc();
    *proc->trapframe = proc->saved_trapframe;
    proc->have_return = 1; // true
    return proc->trapframe->a0;
}

```

7.测试

运行 alarmtest 并确保其能够正确地输出 "alarm!"。

```

$ 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` 来确保你的修改没有影响到内核的其他部分。

```

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

在最后测试时出现了 `panic: balloc: out of blocks` 的错误

```
test writebig: panic: balloc: out of blocks
```

表示在分配磁盘块时，文件系统上的可用块已经耗尽。这通常与文件系统的大小配置和测试过程中创建的文件数量有关。

我查阅资料得知 `writebig`，会尝试写入大量数据以测试文件系统的极限。在块数不足的情况下，

系统无法再为新数据分配块，从而触发 `balloc` 函数的错误。因此在我对宏定义 `#define FSSIZE` 修改后，将文件系统的块数增加到了2000，解决了这个问题

实验心得

在本次实验中，我深入理解并实现了 xv6 操作系统中的一个关键功能：定期警报机制。通过为 xv6 添加 `sigalarm` 和 `sigreturn` 系统调用，我学到了如何在操作系统层面管理定时事件，并让用户空间的进程能够对这些事件做出响应。这种机制的实现对于增强操作系统的灵活性和功能性具有重要意义，尤其是在处理实时系统或需要定期执行任务的场景中。

在实现过程中，最大的挑战是正确处理寄存器的保存和恢复，以及在处理程序执行完毕后确保进程能够从中断点继续无缝执行。这要求我对 xv6 的中断处理流程、进程状态管理以及系统调用机制有深入的理解。

通过这次实验，我不仅加深了对操作系统内部机制的认识，还提高了调试和解决复杂问题的能力，特别是在涉及底层系统编程的场景中。这些经验将对我今后处理更复杂的操作系统问题提供宝贵的帮助。

Lab: 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 PTE 标记为无效或只读来拦截内存引用，导致页面错误，并且可以通过修改 PTE 来更改地址的含义。计算机系统中有一句话说，任何系统问题都可能是通过一定程度的间接解决。惰性分配实验室提供了一个例。本实验探讨了另一个示

例：复制写入分叉。

若要启动实验室，请切换到 `cow` 分支：

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

问题

xv6 中的 `fork()` 系统调用复制了所有父进程的用户空间内存到子项中。如果父级较大，则复

制可以 需要很长时间。更糟糕的是，工作往往被大量浪费;例如 子项中的 `fork()` 后跟 `exec()` 将导致子项 丢弃复制的内存，可能从未使用过其中的大部分。另一方面，如果父级和子级都使用一个页面，并且一个或两个 写出来，确实需要一份副本。

解决方案

写入时复制 (COW) `fork()` 的目标是延迟分配和 复制子项的物理内存页，直到副本实际为止 需要，如果有的话。

COW `fork()` 只为子项创建一个页面表，并为用户创建 PTE 指向父级物理页的内存。COW `fork()` 标记所有 父项和子项中的用户 PTE 均为不可写。当 进程尝试写入这些 COW 页面之一，CPU 将强制执行 页面错误。内核页面错误处理程序检测到这种情况，分配 故障进程的一页物理内存，复制 将原页面放入新页面，并在 故障进程引用新页面，这次使用 PTE 标记为可写。当页面错误处理程序返回时，用户 进程将能够编写其页面副本。

COW `fork()` 释放了实现用户的物理页面 记忆有点棘手。给定的物理页面可以通过以下方式引用 多个进程的页表，并且只应在最后一个进程的页表中释放 引用消失。

Implement copy-on write

实验目的

本实验的目的是在 xv6 操作系统内核中实现写入时复制 (Copy-on-Write, COW) 机制。通过实现这一机制，我们旨在优化内存管理，特别是在进程创建 (如 `fork()` 调用) 过程中，减少物理内存的消耗。通过共享父进程和子进程之间的物理页，COW 可以显著降低内存开销，并在写操作发生时动态分配新页面。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. 打开 kernel/riscv.h 文件，定位到 PTE 标志的定义部分。确认 RISC-V 页表项中存在一个 RSW (Reserved for Software) 位，可以用来标记 COW 页。

```
#define PTE_RSW (1L << 8) // 使用第 8 位作为 COW 页的标志
```

2. 在 fork() 调用中，uvmcopy() 函数负责将父进程的地址空间复制到子进程。为了实现 COW，我们将页面标记为只读并共享，同时将可写的页面标记为 COW 页。找到 uvmcopy() 函数，修改代码，使得父子进程共享物理页面，并设置 PTE 为只读和 COW 标志。

```
int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz) {
    ...
    if (flags & PTE_W) { // 如果页面可写
        flags &= ~PTE_W; // 清除 PTE_W 标志
        flags |= PTE_COW; // 标记为 COW 页
        incref(pa);      // 增加页面引用计数
    }
    ...
}
```

3. 在 kalloc 和 kfree 函数中实现引用计数机制，确保正确管理物理页面的分配和释放。引用计数机制可以帮助操作系统跟踪有多少个进程共享同一物理页面，当没有进程使用时，系统可以安全地释放页面。编辑 kernel/kalloc.c 文件，添加引用计数数组和自旋锁。

```
int useReference[PHYSTOP/PGSIZE]; // 全局引用计数数组
struct spinlock ref_count_lock;    // 自旋锁保护引用计数数组
```

在 kalloc() 函数中初始化新分配的页面的引用计数为 1。

```

void *kalloc(void) {
    ...
    if(r) {
        acquire(&ref_count_lock);
        useReference[(uint64)r / PGSIZE] = 1; // 初始化引用计数
        release(&ref_count_lock);
    }
    ...
}

```

在 kfree() 函数中减少页面的引用计数，当引用计数为 0 时释放页面。

```

void kfree(void *pa) {
    ...
    acquire(&ref_count_lock);
    useReference[(uint64)pa/PGSIZE] -= 1;
    int temp = useReference[(uint64)pa/PGSIZE];
    release(&ref_count_lock);

    if (temp > 0)
        return; // 引用计数不为 0，不释放页面
    ...
}

```

4.在 usertrap() 中处理 COW 页面错误。当子进程或父进程尝试写入一个 COW 页面时，会触发页面错误，此时需要为该进程分配一个新的物理页面，并复制旧页面的数据。

在 kernel/trap.c 中定义一个 cowhandler() 函数，用于处理 COW 页面错误。

```

int cowhandler(pagetable_t pagetable, uint64 va) {
    ...
    // 分配新页面
    char *mem = kalloc();
    if(mem == 0)
        return -1;
    // 复制页面数据
    memmove(mem, (char*)pa, PGSIZE);
    kfree((void*)pa); // 释放旧页面
    *pte = PA2PTE(mem) | (flags | PTE_W) & ~PTE_COW; // 更新 PTE
    sfence_vma(); // 刷新 TLB
    return 0;
}

```

在 usertrap() 函数中调用 cowhandler() 处理页面错误。

```

if(r_scause() == 15) { // 页面错误
    if (cowhandler(p->pagetable, r_stval()) != 0) {
        p->killed = 1;
    }
}

```

5.处理 copyout() 函数中的 COW 页面

修改 copyout() 函数，使其在写入 COW 页面时触发 cowhandler()。copyout() 函数用于将数据从内核空间复制到用户空间。如果遇到 COW 页面，应该触发 cowhandler() 进行处理。

```

int copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len) {
    ...
    if((*pte & PTE_W) == 0 && (*pte & PTE_COW)) {
        if (cowhandler(pagetable, va0) != 0)
            return -1;
        pte = walk(pagetable, va0, 0);
    }
    ...
}

```

6.测试并验证

在 shell 中运行 cowtest 和 usertests。

cowtest:

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

usertests:

```
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

在实现引用计数管理时，最初在 `kfree()` 函数中对引用计数的处理出现了逻辑错误。我将所有页面的引用计数初始化为 1，但在某些情况下引用计数可能会被错误地减少到 0，从而导致页面过早被释放。经过仔细调试和分析，最终通过在 `kfree()` 函数中添加判断条件，确保只有在引用计数真正为 0 时才释放页面，这个问题得以解决。

实验心得

通过此次实验，我深刻体会到操作系统中内存管理的重要性和复杂性。写入时复制（Copy-on-Write, COW）作为一种优化内存使用的技术，极大地减少了 `fork()` 操作时的内存开销，这在资源有限的系统中尤为关键。实验中遇到的各种问题不仅帮助我加深了对 COW 机制的理解，也让我在调试和问题排查方面积累了宝贵的经验。我深入理解了引用计数的作用和实现方式，并且认

识到在资源共享场景下，正确的引用计数管理对资源的有效利用至关重要。

Lab: Multi-threading

本实验将使您熟悉多线程处理。你会 在用户级线程包中实现线程之间的切换，使用多个线程来加速程序，并实现屏障。

若要启动实验室，请切换到线程分支：

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

Uthread: switching between threads

实验目的

在本练习中，您将设计上下文切换机制 用户级线程系统，然后实现它。为了得到你 启动时，您的 xv6 有两个文件 user/uthread.c 和 用户/uthread_switch.S 和 Makefile 中的规则来构建 uthread 程序。uthread.c 包含大部分用户级线程包，以及三个简单测试线程的代码。线程包缺少一些用于创建线程和切换的代码 线程之间。

实验步骤

1.首先仔细阅读 uthread.c 和 uthread_switch.S 文件的现有代码。理解这些文件中已有的线程测试逻辑以及用户级线程的基本操作，确保对代码的运行机制有一个全面的理解。

2.设计线程结构体

在 uthread.c 文件中，设计一个用于表示线程的结构体。在这个结构体中，包括保存线程状态和寄存器信息的字段。这些字段将用于在线程切换过程中保存和恢复线程的上下文信息。

3.. 实现线程创建功能

在 user/uthread.c 文件中完善 thread_create() 函数的实现。

通过设置 ra 寄存器来保存线程函数的地址，使得在线程被切换时能够跳转到正确的函数执行。并将 sp 寄存器指向线程的栈底，以确保栈指针的正确性。

```
t->context.ra = (uint64)func;
t->context.sp = (uint64)(t->stack + STACK_SIZE); // 栈指针指向栈底
```

4. 编写线程切换逻辑

在 `user/uthread_switch.S` 文件中，编写实现线程切换的汇编代码。在这个函数中，需要保存当前线程的寄存器状态，然后切换到下一个线程，并恢复下一个线程的寄存器状态。可以参考 `swtch.S` 文件中的实现方法，确保在切换过程中正确处理所有需要保存的寄存器。

5. 实现调度器功能

在 `user/uthread.c` 中实现 `thread_schedule` 函数。这个函数的作用是调用 `thread_switch` 来实现线程的切换。需要传递当前线程的上下文和下一个要运行的线程的上下文指针作为参数。

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

6. 运行和验证

通过 `make qemu` 来编译并运行 `xv6` 系统，执行 `uthread` 程序进行测试。观察程序输出，以确认线程切换是否按预期工作。

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

实验中遇到的问题

在实验初期，运行 `uthread` 程序时，我遇到了一个 `usertrap` 错误，提示非法内存访问。经过调试，我发现这是由于栈指针 `sp` 的初始化错误导致的。在 `thread_create` 函数中，我最初将栈指针设置为栈的顶部位置，而不是底部，这导致线程在切换时试图访问无效的内存地址。

为了修复这个问题，我将 `sp` 寄存器指向栈的底部，即将栈指针初始化为 `t->stack + STACK_SIZE`。这样，当线程切换时，栈指针将正确指向线程的栈空间，从而避免了非法内存访问错误。

```
t->context.sp = (uint64)(t->stack + STACK_SIZE); // 正确指向栈底
```

实验心得

通过本次实验，我深入了解了用户级线程的实现和上下文切换的原理。特别是在编写 `thread_switch` 和调度器时，我体会到了如何在用户空间保存和恢复线程的执行状态。实验中遇到的错误让我意识到，线程切换的正确性取决于对栈指针和返回地址的准确管理。通过反复调试和修正代码，我学会了如何在复杂的线程环境下处理内存和寄存器的操作。

此外，这次实验还让我深刻理解了用户级线程与内核级线程的不同之处。用户级线程的实现虽然较为轻量，但需要开发者自己处理上下文切换和调度机制，这对代码的正确性提出了更高的要求。总的来说，这次实验不仅增强了我对多线程编程的理解，也让我在调试复杂系统时积累了宝贵的经验。

Using threads

实验目的

理解多线程编程中的数据一致性问题：通过在多线程环境下操作共享数据结构，体会并发现数据竞争和不一致性的问题。

学习并掌握线程同步机制：通过使用互斥锁（`pthread_mutex_t`），确保多线程环境下共享数据的正确性。

优化并发性能：在保证数据一致性的前提下，尝试通过细粒度锁机制提高并发操作的性能。

实践并验证并发程序的正确性和效率：通过编写、运行和测试代码，验证所实现的并发哈希表在不同线程数下的正确性和性能表现。

实验步骤

1. 确保您处于 xv6 项目的根目录

运行以下命令编译和执行初始代码：

```
make ph
./ph 1
```

出现如下结果：

```
100000 puts, 7.859 seconds, 12724 puts/second
0: 0 keys missing
100000 gets, 7.434 seconds, 13451 gets/second
```

这表示在单线程情况下，哈希表的插入和获取操作都是正确的，没有丢失任何键。

运行以下命令以使用两个线程测试：

```
./ph 2
```

出现如下结果：

```
100000 puts, 3.092 seconds, 32338 puts/second
1: 16588 keys missing
0: 16588 keys missing
200000 gets, 7.275 seconds, 27493 gets/second
```

可以观察到，在两个线程的情况下，出现了大量的丢失键，这表明在多线程环境下，哈希表操作存在数据竞争和不一致性问题。

2.

为了更深入地理解多线程问题，分析可能导致 keys missing 的原因。在多线程环境下，多个线程可能同时对同一个哈希桶进行操作，例如两个线程在几乎同时调用 insert 函数，而第四个参

数 n 代表的链表头可能是相同的，导致一个线程的插入操作覆盖或丢失另一个线程的操作。

观察 ph.c 中与插入操作相关的代码段：

```
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]);
    }
}
```

在上述代码中，如果多个线程同时执行 put 函数，并且它们处理的哈希桶是相同的，那么可能会发生竞争条件，导致插入操作的漏插或覆盖。

3.根据分析结果，我们可以通过添加锁来保护共享资源，防止多线程竞争导致的数据不一致问题。具体方法如下：

在 main 函数中，使用 pthread_mutex_init 为每个哈希桶初始化一个锁：

```
pthread_mutex_t lock[NBUCKET];

for (int i = 0; i < NBUCKET; ++i)
    pthread_mutex_init(&lock[i], NULL);
```

在 put 函数中，使用 pthread_mutex_lock 和 pthread_mutex_unlock 对插入操作进行保护，确保每个桶的插入操作是原子性的：

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

    // 加锁
    pthread_mutex_lock(&lock[i]);

    // 检查键是否已经存在
    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next) {
        if (e->key == key)
            break;
    }
    if (e) {
        // 更新已存在的键
        e->value = value;
    } else {
        // 插入新的键值对
        insert(key, value, &table[i], table[i]);
    }

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

类似地，在 `get` 函数中也添加锁保护，以防止并发读取时的数据竞争：

```

static struct entry* get(int key)
{
    int i = key % NBUCKET;

    // 加锁
    pthread_mutex_lock(&lock[i]);

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

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

    return e;
}

```

在程序结束时，销毁锁以释放资源：

```

for (int i = 0; i < NBUCKET; ++i)
    pthread_mutex_destroy(&lock[i]);

```

4.在添加锁保护后，重新编译和运行程序：

```

make ph
./ph 2

```

出现如下的结果

```

100000 puts, 4.040 seconds, 24752 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 7.856 seconds, 25459 gets/second

```


说明所有键都已成功插入哈希表。锁的添加正确

5.进行make grade 确保通过了 ph_safe 测试，这表明代码在多线程环境下是线程安全的。通过 ph_fast 测试，检查代码在多线程环境下是否获得了显著的性能提升。ph_fast 测试要求两个线程每秒的 put 操作数至少是一个线程的 1.25 倍。

```
ph_safe: OK (12.4s)
== Test ph_fast == make[1]: Entering directory '/home/salad14/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/salad14/xv6-labs-2021'
ph_fast: OK (27.9s)
== Test barrier == make[1]: Entering directory '/home/salad14/xv6-labs-2021'
```

实验中遇到的问题

在本次实验中，旨在实现一个线程安全的哈希表，以便在多线程环境中能够正确且高效地执行插入和查询操作。然而，在初步测试中，我发现在多线程环境下运行实验程序时，会出现键值对丢失的情况，表现为 keys missing 的数量显著增加。

具体来说，当我使用两个线程同时向哈希表插入数据时，尽管程序的运行速度明显加快，但仍然有一部分键没有被正确插入。这导致了在后续的查询操作中，这些键无法被找到，出现了预期以外的 keys missing 错误。

经过对程序的深入分析，我发现问题的根源在于多线程环境下的竞争条件。具体而言，当两个线程几乎同时尝试在同一个哈希桶中插入不同的键时，由于缺乏适当的同步机制，可能会导致以下情况：

两个线程几乎同时进入 put 函数，且目标哈希桶相同。

第一个线程尚未完成对哈希桶的更新，第二个线程便开始操作，导致数据被覆盖或链表结构被破坏。

由于没有正确的锁机制保护，最终导致部分插入操作失败，键值对丢失。

这种竞争条件是多线程编程中的典型问题，当多个线程同时操作共享资源且没有正确同步时，可能会导致数据不一致或丢失。

为了解决上述问题，我选择为每个哈希桶引入互斥锁 (pthread_mutex_t)，以确保每次只有一个线程可以访问和修改特定的哈希桶。这种方式可以有效防止线程之间的竞争条件，保证数据操作的原子性和一致性。

实验心得

通过本次实验，我深入理解了线程安全的重要性，并学会了使用 pthread 库和锁机制来保证多线程环境下的数据一致性。同时，我还尝试通过优化锁的使用来提高程序的并行性能。本次实验不仅帮助我掌握了多线程编程的基本技能，也为进一步的并发编程打下了坚实的基础。

Barrier

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。

解决线程竞争问题：理解和处理多个线程在不同同步障碍点之间竞争的复杂情况，确保每一轮次的障碍行为独立且正确。

实验步骤

1.理解屏障（Barrier）：

屏障是一种同步机制，使得一组线程在执行到达特定点之前，必须等待其他所有线程也达到这一点。在所有线程都到达障碍点之前，没有线程能够继续执行。

2.初步分析 barrier.c 文件：

barrier.c 文件中包含了一个损坏的屏障实现。当前实现的问题在于，当有线程到达障碍点后继续执行，而其他线程尚未到达时，会导致同步失败，最终导致断言失败。

3.理解条件变量和互斥锁：

条件变量 (pthread_cond_t)：允许线程在某些条件满足时被唤醒。在条件满足之前，线程会阻塞等待。

互斥锁 (pthread_mutex_t)：确保共享资源在同一时间内只能被一个线程访问，用于保护临界区。

4.实现barrier()函数：为了实现 barrier，需要用到 UNIX 提供的条件变量以及 wait/broadcast 机制。

```
pthread_mutex_lock(&bstate.barrier_mutex);
if (++bstate.nthread < nthread) {
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
} else {
    bstate.nthread = 0;
    ++bstate.round;
    pthread_cond_broadcast(&bstate.barrier_cond);
}
pthread_mutex_unlock(&bstate.barrier_mutex);
```

在这个函数中，bstate.nthread 表示当前已经达到屏障的线程数量。当一个线程进入屏障时，会将这个计数值加一，以记录达到屏障的线程数量。

如果还有线程未达到屏障，这个线程就会调用 pthread_cond_wait() 来等待在条件变量 barrier_cond 上。在调用这个函数之前，线程会释放之前获取的锁 barrier_mutex，以允许其他线程在这个锁上等待。

当最后一个线程到达屏障，bstate.nthread 的值会等于 nthread，就会进入 else 分支。在这里，首先重置 bstate.nthread 为 0，以便下一轮的屏障计数。然后，增加 bstate.round 表示进入了下一个屏障的轮次。最后，通过调用 pthread_cond_broadcast() 向所有在条件变量上等待的线程发出信号，表示可以继续执行。

5.处理多轮次和竞争条件：使用bstate.round来记录当前轮次，确保每次所有线程都达到屏障后增加轮次。避免在上一轮仍在使bstate.nthread时，另一个线程增加了该值。

6.编译测试

实验中遇到的问题

在本次实验中，我们的目标是实现一个线程同步的屏障（Barrier），确保所有线程在继续执行之前都能够达到屏障点。然而，在实验过程中，程序运行时多次出现了 assert 失败的情况，提示 assert (i == t) 不成立。这意味着某些线程在其他线程到达屏障点之前已经继续执行，导致了同步失败。

经过对代码的深入分析，我发现问题主要出在以下几个方面：

竞争条件 (Race Condition) :

如果在某些线程进入 `barrier()` 函数并等待其他线程到达时，有线程在释放锁后立即获得调度权，并且该线程可能会开始下一轮的同步。这可能会导致某些线程在本轮次屏障中未被阻塞，从而触发 `assert` 失败。

轮次管理不当：

由于我们使用了一个全局的 `round` 变量来管理同步的轮次，如果某个线程过早地进入下一轮次，则其他线程可能仍然停留在上一轮次。这种情况会导致线程无法正确同步，最终导致 `assert` 失败。

为了修复这些问题，我们对代码进行了以下调整和优化：

改进 `barrier()` 函数中的等待逻辑：

我们确保线程在 `pthread_cond_wait()` 函数中阻塞时，能够准确等待所有线程到达当前屏障点。通过维护 `round` 变量的值，使每个线程都等待当前轮次结束，避免线程过早进入下一轮次。

确保轮次的一致性：

在 `barrier()` 中，我们使用 `++bstate.round` 来更新当前轮次，并确保所有线程在进入下一轮次之前都已经完成了上一轮次的同步。通过使用 `pthread_cond_broadcast()` 唤醒所有等待的线程，我们保证所有线程在轮次切换时的一致性。

实验心得

在本次实验中，我成功实现了一个多线程同步的屏障 (Barrier)，并通过 Pthreads 库中的互斥锁和条件变量来协调多个线程的执行顺序。这次实验不仅使我进一步掌握了多线程编程的基本概念，还让我深入理解了线程同步在实际应用中的重要性和复杂性。

1. 多线程编程的挑战

在实验的初期，我遇到了线程同步失败的问题，即部分线程在其他线程到达屏障点之前继续执行。这暴露了多线程编程中常见的竞争条件问题。通过这次实验，我更加认识到，在多线程环境中，即使是看似简单的同步任务，如果没有正确的同步机制，都会引发潜在的错误。这促使我更加重视代码中的线程安全性。

2. 条件变量与互斥锁的使用

通过实验，我掌握了条件变量 (`pthread_cond_t`) 和互斥锁 (`pthread_mutex_t`) 的使用方法。条件变量和互斥锁的结合使用是解决线程同步问题的有效手段。尤其是在实现屏障时，通过条件变量让线程进入等待状态，并通过广播唤醒所有线程，确保了线程的正确同步。这种机制不仅能够确保线程同步的正确性，还能够有效避免死

锁和资源争用。

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (6.4s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/salad14/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/salad14/xv6-labs-2021'
ph_safe: OK (13.3s)
== Test ph_fast == make[1]: Entering directory '/home/salad14/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/salad14/xv6-labs-2021'
ph_fast: OK (26.9s)
== Test barrier == make[1]: Entering directory '/home/salad14/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/salad14/xv6-labs-2021'
barrier: OK (7.1s)
== Test time ==
time: OK
Score: 60/60
```

Lab: networking

在本实验中，您将为网络接口编写 xv6 设备驱动程序卡（NIC）。

获取实验室的 xv6 源代码并查看 net 分支：

```
$ git fetch
$ git checkout net
$ make clean
```

背景

您将使用名为 E1000 的网络设备来处理网络通信。对于 xv6（以及您编写的驱动程序），E1000 看起来像一个真实的连接到真实以太网局域网（LAN）的硬件。在事实上，您的驾驶员将与之交谈的 E1000 是 qemu 提供的仿真，连接到也由 qemu 模拟的 LAN。在此模拟 LAN 上，xv6（“来宾”）IP 地址为 10.0.2.15。Qemu 还为运行 qemu 的计算机进行了安排以 IP 地址 10.0.2.2。当 xv6 使用 E1000 发送一个数据包发送到 10.0.2.2，QEMU 将数据包传送到运行 QEMU 的（真实）计算机（“主机”）。

您将使用 QEMU 的“用户模式网络堆栈”。QEMU 的文档中有更多关于用户模式的信息 [堆栈在这里](#)。我们更新了 Makefile 以启用 QEMU 的用户模式网络堆栈和 E1000 网卡。

Makefile 将 QEMU 配置为记录所有传入和传出 数据包复制到 Lab 目录中的文件 packets.pcap。复习可能会有所帮助 这些录音用于确认 XV6 正在传输和接收您的数据包 期望。要显示录制的数据包：

```
tcpdump -XXnr packets.pcap
```

我们已将一些文件添加到此实验室的 xv6 存储库中。文件 kernel/e1000.c 包含初始化 E1000 的代码以及 E1000 的空函数 发送和接收数据包，您将填写这些数据包。kernel/e1000_dev.h 包含以下定义 E1000 定义的寄存器和标志位，以及 在英特尔 E1000 软件开发人员手册中进行了描述。kernel/net.c 和 kernel/net.h 包含一个简单的网络堆栈，用于实现 IP、UDP 和 ARP 协议。这些文件还包含用于灵活 保存数据包的数据结构，称为 MBUF。最后，kernel/pci.c 包含以下代码：当 xv6 启动时，在 PCI 总线上搜索 E1000 卡。

Your Job

你的工作是完成 e1000_transmit () 和 e1000_recv () ， 都在 kernel/e1000.c 中， 以便驱动程序可以发送和接收数据包。当 make grade 说你的 solution 通过所有测试。

在编写代码时，您会发现自己在参考 E1000 软件开发人员手册。特别有帮助的是以下部分：

第 2 节是必不可少的，它概述了整个设备。

第 3.2 节概述了数据包接收。

第 3.3 节和第 3.4 节一起概述了数据包传输。

第 13 节概述了 E1000 使用的寄存器。

第 14 节可以帮助您了解我们提供的 init 代码。

实验步骤

1.了解网络设备：使用E1000的网络设备来处理网络通信。这个虚拟设备模拟了一个真实的硬件连接到真实的以太网局域网（LAN）。在xv6中，E1000看起来就像连接到真实LAN的硬件设备。你需要了解如何在xv6中与这个虚拟设备进行通信。

2.实现驱动函数：在kernel/e1000.c文件中完成e1000_transmit()和e1000_recv()函数。

e1000_transmit()函数用于发送数据包，而e1000_recv()函数用于接收数据包。我们在 e1000.c

中提供的 `e1000_init()` 函数可配置 E1000 从 RAM 中读取要传输的数据包，并将接收到的数据包写入 RAM。这种技术称为 DMA（直接内存访问），指的是 E1000 硬件直接将数据包写入/读出 RAM。

3.e1000_transmit() 函数实现

该函数的主要功能是将数据包写入发送描述符环（TX ring）并通知 E1000 设备开始传输。

获取传输环索引并检查描述符状态：

```
acquire(&e1000_lock);
int idx = regs[E1000_TDT];
if ((tx_ring[idx].status & E1000_TXD_STAT_DD) == 0) {
    release(&e1000_lock);
    return -1; // 如果描述符未完成传输，则返回
}
```

`acquire(&e1000_lock);`：我们使用自旋锁来保护共享资源（如描述符环和相关寄存器）免受并发访问的影响。在多线程或多进程环境中，这样可以防止多个线程同时访问和修改同一块数据，从而避免竞争条件。

`int idx = regs[E1000_TDT];`：E1000_TDT（Transmit Descriptor Tail）寄存器保存了发送描述符环中的当前尾索引，即下一个将被使用的描述符的位置。通过读取该寄存器，我们得知 E1000 设备期望发送的下一个数据包应存放在哪个位置。

`if ((tx_ring[idx].status & E1000_TXD_STAT_DD) == 0)`：E1000_TXD_STAT_DD 是描述符状态中的一个位，表示数据包是否已经传输完成。如果该位未设置，说明 E1000 仍在传输之前的数据包，因此当前描述符尚未准备好被复用。如果遇到这种情况，我们应该返回错误并退出，以避免覆盖未完成传输的数据。

释放之前使用的 mbuf 并更新描述符：

```
if (tx_mbufs[idx]) mbuf_free(tx_mbufs[idx]);
tx_mbufs[idx] = m;
tx_ring[idx].length = m->len;
tx_ring[idx].addr = (uint64)m->head;
tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
```

`if (tx_mbufs[idx]) mbuf_free(tx_mbufs[idx]);`：如果之前描述符指向的 mbuf（内存缓冲区）不为

空，我们需要释放它，以便新的数据包可以被存储。这是为了防止内存泄漏，确保每个数据包都能在发送后被正确释放。

`tx_mbufs[idx] = m;`：将当前要发送的 mbuf 指针存储在 `tx_mbufs` 数组中，以便在发送完成后可以访问并释放这个缓冲区。

`tx_ring[idx].length = m->len;`：设置描述符的长度字段为数据包的长度。这样，E1000 设备在发送数据包时知道需要发送多少字节的数据。

`tx_ring[idx].addr = (uint64)m->head;`：设置描述符的地址字段为数据包的起始地址。这是 E1000 设备在传输数据包时需要访问的内存位置。

`tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;`：设置描述符的命令字段。E1000_TXD_CMD_RS 表示在数据包发送完成后，E1000 设备应更新描述符的状态字段。E1000_TXD_CMD_EOP 表示这是一个完整的数据包（即数据包的结束部分），无需再追加其他数据包。

更新传输描述符尾寄存器并释放锁：

```
regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;  
release(&e1000_lock);  
return 0;
```

`regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;`：将传输描述符尾寄存器 E1000_TDT 更新为下一个位置。这告诉 E1000 设备我们已经准备好发送数据包，设备应该开始传输数据。通过 mod 操作（`% TX_RING_SIZE`），我们确保索引在描述符环内循环。

`release(&e1000_lock);`：释放自旋锁，以允许其他线程或进程访问共享资源。

`return 0;`：成功返回，表示数据包已成功添加到传输队列中。

4.e1000_recv() 函数实现

该函数的主要功能是从接收描述符环（RX ring）中读取数据包并传递给网络协议栈处理。

获取接收环索引并检查描述符状态：


```
int idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
while(rx_ring[idx].status & E1000_RXD_STAT_DD) {
    rx_mbufs[idx]->len = rx_ring[idx].length;
    net_rx(rx_mbufs[idx]); // 将 mbuf 传递给上层协议栈
}
```

`int idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;` `E1000_RDT` (Receive Descriptor Tail) 寄存器保存了接收描述符环中的尾索引。通过读取并增加 1，我们可以得到下一个应处理的数据包的位置。这里同样通过 `mod` 操作保证索引在描述符环内循环。

`while(rx_ring[idx].status & E1000_RXD_STAT_DD):` `E1000_RXD_STAT_DD` 是接收描述符状态中的一个位，表示数据包是否已经被接收并写入内存。如果该位被设置，表示有新数据包可供处理。`while` 循环会遍历所有可用的接收描述符，处理每一个接收到的数据包。

`rx_mbufs[idx]->len = rx_ring[idx].length;`：将接收描述符中的数据包长度记录到 `mbuf` 中，以便上层协议栈能够正确处理数据包。

`net_rx(rx_mbufs[idx]);`：`net_rx()` 是一个网络协议栈的入口函数，负责处理接收到的以太网帧。它会解析数据包，并根据其类型（如 IP、ARP）将数据包分发到相应的处理函数。

分配新的 `mbuf` 并更新描述符：

```
rx_mbufs[idx] = mbufalloc(0);
rx_ring[idx].status = 0;
rx_ring[idx].addr = (uint64)rx_mbufs[idx]->head;
regs[E1000_RDT] = idx;
idx = (idx + 1) % RX_RING_SIZE;
}
```

`rx_mbufs[idx] = mbufalloc(0);`：接收描述符环中的每个描述符需要有一个对应的 `mbuf` 缓冲区，用于存放接收到的数据。在处理完当前数据包后，我们为描述符分配一个新的 `mbuf` 缓冲区，以便接收下一个数据包。

`rx_ring[idx].status = 0;`：清除描述符的状态位，以便下次可以重复使用该描述符。

`rx_ring[idx].addr = (uint64)rx_mbufs[idx]->head;`：将新的 `mbuf` 缓冲区地址存储到接收描述符中，以便 `E1000` 设备在下次接收数据包时知道将数据写入何处。

`regs[E1000_RDT] = idx;`：更新接收描述符尾寄存器 `E1000_RDT`，通知 `E1000` 设备我们已处理

完此数据包并已准备好接收新的数据包。

`idx = (idx + 1) % RX_RING_SIZE;` 循环更新索引，准备处理下一个描述符。

5.编译调试，使用make grade测试

```
== Test running nettests ==
$ make qemu-gdb
(5.5s)
== Test    nettest: ping ==
    nettest: ping: OK
== Test    nettest: single process ==
    nettest: single process: OK
== Test    nettest: multi-process ==
    nettest: multi-process: OK
== Test    nettest: DNS ==
    nettest: DNS: OK
== Test time ==
time: OK
Score: 100/100
```

实验中遇到的问题

在实验过程中，我遇到了一个问题，当运行 nettests 时，测试卡在了 ping 部分，没有任何输出，系统似乎在等待某些操作完成。我怀疑问题出在 E1000 驱动程序的 `e1000_transmit()` 或 `e1000_recv()` 函数中，因为这些函数负责数据包的发送和接收。如果这两个函数没有正确工作，数据包就无法传输或接收，导致测试卡住。

为了解决这个问题，我首先在 `e1000_transmit()` 和 `e1000_recv()` 函数中添加了调试输出，以确认它们是否被正确调用，以及描述符环是否正常更新。通过观察调试输出，我发现 `e1000_transmit()` 中的描述符状态未正确更新，导致 E1000 设备无法发送数据包。我修改了相关代码，确保在每次发送数据包后正确更新 E1000_TDT 寄存器，并在 `e1000_recv()` 中正确处理接收到的数据包，最终问题得以解决，nettests 测试顺利通过。

实验心得

在本次实验中，我深入探索了 E1000 网卡驱动程序的实现，并且通过解决实际问题，进一步理解了计算机网络和操作系统之间的协作机制。实验过程中，我遇到了 nettests 测试卡住的问题，这促使我反思驱动程序中数据包的发送与接收逻辑。通过在 `e1000_transmit()` 和 `e1000_recv()` 函数中添加调试输出，我得以逐步排查错误，发现问题的根源在于描述符环的更新不正确以及接收数据包时的处理逻辑不完善。

在修复这些问题的过程中，我不仅加深了对描述符环和寄存器操作的理解，还体会到了驱动程序开发中细节管理的重要性。从这个实验中，我学会了如何通过调试和迭代改进代码，同时也认识到在驱动程序设计中，确保每一个细节都得到正确处理是多么关键。通过这次实验，我的动手能力和问题解决技巧得到了进一步提升，也为我将来更复杂的系统编程打下了坚实的基础。

Lab: locks

在本实验中，您将获得重新设计代码的经验，以提高 排比。多核并行性差的常见症状 machines 是高锁争用。提高并行度通常涉及 更改数据结构和锁定策略，以便 减少争用。您将对 xv6 内存分配器和 块缓存。

实验开始前，请切换到lock分支

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

Memory allocator

实验目的

本实验的目的是通过重新设计 xv6 操作系统的内存分配器，以减少由于锁争用导致的性能瓶颈。当前的内存分配器使用单一的空闲列表，由单个锁保护，多个 CPU 在并发访问时容易产生锁争用，导致性能下降。通过实现每个 CPU 独立的空闲列表，并处理空闲列表为空时的内存窃取问题，我们希望显著减少锁争用，提高系统的并发性能。

实验步骤

1.修改kmem为kmem[NCPU] kmem[NCPU] 是一个包含 NCPU 个元素的数组，每个元素都包含一个自旋锁 lock 和一个指向空闲内存块的指针 freelist。NCPU 表示系统中的 CPU 数量。这样设计的目的是为每个 CPU 提供独立的内存管理结构，减少全局锁争用。

2.修改 kinit() 函数:

kinit() 函数负责初始化内存分配器。在多核环境下，我们需要为每个 CPU 分配独立的空闲列表和锁，并将可用的内存页分配到这些列表中。

```
void
kinit()
{
    // 初始化每个 CPU 的 kmem 锁
    for(int i = 0; i < NCPU; i++) {
        initlock(&kmem[i].lock, "kmem");
        kmem[i].freelist = 0;
    }

    // 将可用的内存范围分配到各个 CPU 的空闲列表中
    freerange(end, (void*)PHYSTOP);
}
```

我们首先通过循环初始化每个 CPU 的自旋锁 kmem[i].lock，并将 freelist 指针初始化为 0。然后，通过调用 freerange() 函数，将可用的物理内存页均匀分配到各个 CPU 的空闲列表中。

3.修改 freerange() 函数

freerange() 函数负责将一段物理内存释放到空闲列表中。在多核环境下，这些内存页应被分配到调用 CPU 的空闲列表中。

```

void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);

    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE) {
        int cpu = cpuid(); // 获取当前 CPU 编号

        push_off(); // 关闭中断，防止并发问题
        acquire(&kmem[cpu].lock); // 获取当前 CPU 的 kmem 锁

        // 将页添加到当前 CPU 的空闲列表中
        struct run *r = (struct run *)p;
        r->next = kmem[cpu].freelist;
        kmem[cpu].freelist = r;

        release(&kmem[cpu].lock); // 释放锁
        pop_off(); // 恢复中断
    }
}

```

通过使用 `cpuid()` 获取当前 CPU 的编号，并将内存页释放到相应 CPU 的空闲列表中。通过 `push_off()` 和 `pop_off()` 确保操作过程中不受中断干扰，从而保证并发的安全性。

4.修改 `kalloc()` 函数

`kalloc()` 函数负责分配一个内存页。我们首先尝试从当前 CPU 的空闲列表中分配内存，如果当前 CPU 的空闲列表为空，则尝试从其他 CPU 的空闲列表中窃取内存。

```

void *
kalloc(void)
{
    struct run *r;
    int cpu = cpuid();

    push_off(); // 关闭中断，防止并发问题
    acquire(&kmem[cpu].lock); // 获取当前 CPU 的 kmem 锁

    r = kmem[cpu].freelist; // 尝试从当前 CPU 的空闲列表中获取页面
    if(r)
        kmem[cpu].freelist = r->next; // 如果成功获取页面，将空闲列表指针移到下一个页面

    release(&kmem[cpu].lock); // 释放当前 CPU 的锁

    if(!r) { // 如果当前 CPU 的空闲列表为空，尝试从其他 CPU 的空闲列表中获取页面
        for(int i = 0; i < NCPU; i++) {
            if(i == cpu) // 跳过当前 CPU
                continue;

            acquire(&kmem[i].lock); // 获取其他 CPU 的 kmem 锁
            r = kmem[i].freelist; // 尝试从其他 CPU 的空闲列表中获取页面
            if(r) {
                kmem[i].freelist = r->next; // 如果成功获取页面，将空闲列表指针移到下一个页面
                release(&kmem[i].lock); // 释放其他 CPU 的锁
                break;
            }
            release(&kmem[i].lock); // 如果没有获取到页面，继续尝试下一个 CPU
        }
    }

    pop_off(); // 恢复中断

    if(r)
        memset((char*)r, 5, PGSIZE); // 填充页面内容以帮助调试
    return (void*)r;
}

```

首先尝试从当前 CPU 的空闲列表中获取内存页，如果当前 CPU 的列表为空，则循环遍历其他

CPU 的空闲列表进行内存窃取。push_off() 和 pop_off() 用于保证整个分配过程中的原子性。

5.

修改 kfree() 函数

kfree() 函数负责释放一个内存页，将其返回到当前 CPU 的空闲列表中。

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

    // 检查释放的页面是否对齐到页边界，并确保地址在合法的物理内存范围内
    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");

    // 用垃圾数据填充内存以捕获悬空引用
    memset(pa, 1, PGSIZE);

    r = (struct run*)pa;

    int cpu = cpuid(); // 获取当前 CPU 编号
    push_off(); // 关闭中断，防止并发问题
    acquire(&kmem[cpu].lock); // 获取当前 CPU 的 kmem 锁

    // 将页面添加到当前 CPU 的空闲列表中
    r->next = kmem[cpu].freelist;
    kmem[cpu].freelist = r;

    release(&kmem[cpu].lock); // 释放锁
    pop_off(); // 恢复中断
}
```

6.运行 kalloc test 测试，查看锁的竞争情况是否减少，确认你的实现是否成功降低了锁的争用。

```
$ kallocetest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 65792
lock: kmem: #test-and-set 0 #acquire() 181942
lock: kmem: #test-and-set 0 #acquire() 185326
lock: bcache: #test-and-set 0 #acquire() 1254
--- top 5 contended locks:
lock: proc: #test-and-set 181221 #acquire() 2097346
lock: proc: #test-and-set 162696 #acquire() 2097284
lock: proc: #test-and-set 162293 #acquire() 2097285
lock: proc: #test-and-set 161512 #acquire() 2097284
lock: proc: #test-and-set 159230 #acquire() 2097285
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

7.运行 usertests sbrkmuch 测试，确保内存分配器仍然能够正确分配所有内存。

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

8.运行 usertests 测试，确保所有的用户测试都通过。


```
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

在实验过程中，我遇到了一个意外的问题。当我运行 `kalloctest` 测试时，发现系统在尝试分配和释放内存时表现出异常的锁争用，导致测试的性能远低于预期。我意识到，虽然我已经为每个 CPU 创建了独立的空闲列表和锁，但在实际操作中，由于内存页的释放和分配频繁发生在不同的 CPU 上，仍然存在大量的锁争用，特别是在多个 CPU 之间频繁窃取内存页时。

为了解决这个问题，我进一步优化了 `kalloc()` 和 `kfree()` 函数，减少了跨 CPU 的内存窃取操作，并且在 `kfree()` 中引入了更细粒度的锁定机制。具体而言，我确保内存页尽可能在其所属的 CPU 上被分配和释放，避免不必要的跨 CPU 操作。这种改进显著减少了锁争用，提高了系统的整体并发性能，最终使得 `kalloctest` 测试顺利通过，并且性能大大提升。

实验心得

在此次实验中，我深入探索了 xv6 操作系统中的内存分配机制，并成功地优化了其在多核环境下的并发性能。通过为每个 CPU 创建独立的空闲列表和锁，我学会了如何有效减少锁争用问题，提升系统的并发处理能力。这一过程不仅加深了我对操作系统内存管理的理解，也让我体会到了并行编程中的挑战与复杂性。

Buffer cache

实验目的

本实验的目的是优化 xv6 操作系统中的块缓存（block cache）系统，减少在多进程密集使用文件系统时对 bcache.lock 的锁争用。通过引入哈希表结构，并对每个哈希桶（hash bucket）使用单独的锁，我们期望显著降低 bcache.lock 的获取次数，从而提高系统的并发性能。最终，我们希望通过运行 bcachetest 测试，确保所有 bcache 相关的锁争用次数接近零（或总和少于 500），并确保系统在并发情况下仍然能正确运行。

实验步骤

1. 了解现有的 bcache 实现

首先，阅读 kernel/bio.c 中的现有代码，了解 bcache 的工作原理。当前 bcache.lock 保护了缓存块的列表、缓存块的引用计数（b->refcnt）以及缓存块的标识符（b->dev 和 b->blockno）。

2. 定义哈希桶结构并初始化

在 buf.h 中定义哈希桶结构 struct bucket。每个桶包含一个自旋锁和一个双向链表头，用于管理缓存的块缓冲区。

在 bio.c 中创建 bcache 结构体，其中包含缓冲区数组和哈希桶数组。

```
struct bucket {
    struct spinlock lock;
    struct buf head; // 双向链表头，用于管理缓存的块缓冲区
};

struct {
    struct buf buf[NBUF]; // 缓冲区数组
    struct bucket bucket[NBUCKET]; // 哈希桶数组
} bcache;
```

NBUF 是缓冲区的数量，NBUCKET 是哈希桶的数量。哈希桶用于减少锁争用。我们能在官方的文档中看到建议的哈希桶数量为13，能减少概率

其中NBUCKET需要我们在param.h 定义为13

3. 初始化哈希桶

在 binit() 函数中，初始化每个缓冲区的睡眠锁。
初始化每个哈希桶的自旋锁和双向链表。

```
void binit(void) {  
    for (int i = 0; i < NBUF; ++i) {  
        initsleeplock(&bcache.buf[i].lock, "buffer");  
    }  
    for (int i = 0; i < NBUCKET; ++i) {  
        initbucket(&bcache.bucket[i]);  
    }  
}
```

initbucket() 函数初始化每个哈希桶的自旋锁，并将链表头指向自己。

4.实现缓冲区查找和获取

在 bget() 函数中，根据块号计算哈希值，将缓冲区映射到哈希桶中。
首先检查目标块是否已经缓存在对应的哈希桶中；如果存在，增加引用计数并返回。
如果未缓存，查找未使用的缓冲区，并将其放入哈希桶中。

5.实现缓冲区释放

在 brelse() 函数中，释放缓冲区时，首先锁定对应的哈希桶。
如果引用计数为 0，将缓冲区移出链表，并清除 used 标志。

```

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

    releasesleep(&b->lock);

    uint v = hash_v(b->blockno);
    struct bucket* bucket = &bcache.bucket[v];
    acquire(&bucket->lock);

    b->refcnt--;
    if (b->refcnt == 0) {
        b->next->prev = b->prev;
        b->prev->next = b->next;
        __atomic_clear(&b->used, __ATOMIC_RELEASE);
    }

    release(&bucket->lock);
}

```

使用 refcnt 管理缓冲区的引用计数，确保缓冲区被释放后重新进入缓存池。

6.运行bcachetest：运行修改后的bcachetest测试程序，观察锁竞争情况和测试结果。优化后，锁竞争应该大幅减少。

```
$ bcachtest
exec bcachtest failed
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 33009
lock: kmem: #test-and-set 0 #acquire() 108
lock: kmem: #test-and-set 0 #acquire() 52
lock: bcache.bucket: #test-and-set 0 #acquire() 4120
lock: bcache.bucket: #test-and-set 0 #acquire() 4132
lock: bcache.bucket: #test-and-set 0 #acquire() 2270
lock: bcache.bucket: #test-and-set 0 #acquire() 4276
lock: bcache.bucket: #test-and-set 0 #acquire() 2262
lock: bcache.bucket: #test-and-set 0 #acquire() 4256
lock: bcache.bucket: #test-and-set 0 #acquire() 4734
lock: bcache.bucket: #test-and-set 0 #acquire() 9132
lock: bcache.bucket: #test-and-set 0 #acquire() 6174
lock: bcache.bucket: #test-and-set 0 #acquire() 6174
lock: bcache.bucket: #test-and-set 0 #acquire() 6174
lock: bcache.bucket: #test-and-set 0 #acquire() 6174
lock: bcache.bucket: #test-and-set 0 #acquire() 6194
--- top 5 contended locks:
lock: proc: #test-and-set 373668 #acquire() 3612906
lock: proc: #test-and-set 334868 #acquire() 3617773
lock: proc: #test-and-set 330304 #acquire() 3574035
lock: proc: #test-and-set 328605 #acquire() 3573845
lock: proc: #test-and-set 323841 #acquire() 3573942
tot= 0
test0: OK
start test1
test1 OK
```

7.运行usertests：运行usertests测试程序，确保修改不会影响其他部分的正常运行。

```
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

在进行缓冲区缓存实验时，我遇到了一个困惑的问题：在运行 `bcachetest` 时，系统频繁出现“panic: bget: no buffers”错误。这个错误让我怀疑系统无法正确分配和管理缓冲区，导致在高并发访问时，缓存中没有可用的缓冲区可供使用。我检查了代码中的 `bget` 函数，发现问题可能出在缓冲区的哈希桶机制上，特别是在高并发情况下，缓冲区可能无法及时从其他哈希桶中获取，导致缓存无法分配新的缓冲区。

为了解决这个问题，我首先修改了 `bget` 函数，确保当当前哈希桶没有可用的缓冲区时，系统可以从其他哈希桶中尝试获取可用的缓冲区。这一改动在一定程度上缓解了问题，但系统仍然会偶尔出现相同的错误。经过进一步的调试和分析，我意识到问题可能还在于缓冲区释放的机制上，因此我在 `brelease` 函数中加入了对引用计数的更严格检查，确保缓冲区在真正空闲时才会被回收。这些改动最终解决了问题，系统不再出现缓冲区耗尽的情况，`bcachetest` 测试顺利通过。

在进行缓冲区缓存实验时，我遇到了一个困惑的问题：在运行 `bcachetest` 时，系统频繁出现“panic: bget: no buffers”错误。这个错误让我怀疑系统无法正确分配和管理缓冲区，导致在高并发访问时，缓存中没有可用的缓冲区可供使用。我检查了代码中的 `bget` 函数，发现问题可能出在缓冲区的哈希桶机制上，特别是在高并发情况下，缓冲区可能无法及时从其他哈希桶中获取，导致缓存无法分配新的缓冲区。

实验心得

在完成缓冲区缓存实验的过程中，我对系统的内存管理和并发处理有了更深入的理解。这个实验让我意识到，在高并发环境中，如何有效地管理和分配资源是非常重要的，特别是在处理像缓冲区这样的共享资源时，设计合理的锁机制和缓存策略至关重要。

实验中遇到的问题和挑战也帮助我认识到调试和优化代码的重要性。通过多次分析和修改，我最终解决了系统在高并发访问下出现的缓存耗尽问题。这不仅让我学到了具体的编程技巧，也让我深刻理解了设计模式和数据结构在实际应用中的重要性。总的来说，这次实验不仅增强了我对操作系统内核的理解，也提高了我处理复杂问题的能力。

Lab: file system

在本实验中，您将向 xv6 添加大文件和符号链接 文件系统。

开始实验前，请更改到fs分支

```
$ git fetch
$ git checkout fs
$ make clean
```

Large files

实验目的

本实验旨在修改 xv6 文件系统，以支持更大的文件大小。具体来说，将实现双重间接块（double-indirect block），使得文件可以包含最多 65,803 个块。这一扩展将增加 xv6 文件系统的最大文件大小，从 268 个块（268 KB）扩展到 65,803 个块（约 65 MB）。

实验步骤

1.修改 bmap 函数

bmap 函数的作用是将文件的逻辑块号映射到磁盘上的物理块号。我们需要修改 bmap 函数以支持双重间接块的处理。

```

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

```

如果逻辑块号 `bn` 小于直接块的数量 `NDIRECT`，则直接访问 `ip->addrs[bn]` 获取相应的物理块号。如果该物理块号为空，则分配一个新的块并存储在 `ip->addrs[bn]` 中。

处理单间接块：

```

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

```

首先将 `bn` 减去 `NDIRECT`，因为前 `NDIRECT` 个块是直接块。接下来，如果逻辑块号在单间接块的范围内，则访问单间接块的索引块，读取其中的物理块号。如果该块未分配，则分配一个新块，并在索引中更新。最后，返回该物理块号。

处理双重间接块：


```

bn -= NINDIRECT;
if (bn < NDOUBLE) {
    if ((addr = ip->addrs[NDIRECT+1]) == 0)
        ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);

    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;
    uint double_bn = bn / NINDIRECT;
    uint single_bn = bn % NINDIRECT;

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

    bp2 = bread(ip->dev, addr);
    a = (uint *)bp2->data;
    if ((addr = a[single_bn]) == 0) {
        a[single_bn] = addr = balloc(ip->dev);
        log_write(bp2);
    }
    brelse(bp2);
    return addr;
}

```

首先将 bn 减去 NINDIRECT，然后检查逻辑块号是否在双重间接块的范围内。如果是，则从双重间接块开始，通过两级索引定位到实际的数据块。在每一级索引中，检查块是否已分配，未分配则进行分配并更新索引。

2.修改 itrunc 函数

我们在修改bmap后，也要对应的修改itrunc，才能实现功能的完善

itrunc 函数的作用是在文件被删除或截断时释放与该文件相关的所有块。

释放直接块：

```

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

```

依次检查每个直接块号，如果它已分配，则释放相应的物理块并将块号置为0。

释放单间接块：

```

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

```

读取单间接块的索引块，释放其中指向的所有物理块，然后释放索引块自身。

释放双重间接块：

```


if (ip->addr[NDIRECT+1]) {
    bp = bread(ip->dev, ip->addr[NDIRECT+1]);
    a = (uint *)bp->data;
    for (j = 0; j < NINDIRECT; j++) {
        if (a[j]) {
            bp2 = bread(ip->dev, a[j]);
            a2 = (uint *)bp2->data;
            for (k = 0; k < NINDIRECT; k++) {
                if (a2[k])
                    bfree(ip->dev, a2[k]);
            }
            brelse(bp2);
            bfree(ip->dev, a[j]);
        }
    }
    brelse(bp);
    bfree(ip->dev, ip->addr[NDIRECT+1]);
    ip->addr[NDIRECT+1] = 0;
}

```

读取双重间接块的索引块，遍历其中的每个单间接块，依次释放它们指向的所有物理块，并最终释放双重间接块自身。

3.在 kernel/file.c 文件中，找到 filewrite() 函数的实现。确保在写文件时正确调用了 bmap() 函数。

4.运行 bigfile 测试，确保它可以成功创建文件，并报告正确的文件大小。这个测试可能会花费一些时间。



```

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

```

5.运行 usertests 测试套件，确保所有的测试都通过。这可以验证你的修改是否没有引入错误，并且 xv6 正常运行。

```
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

在实验的过程中，我遇到了一个摸不清的问题。当我修改完 bmap 函数以支持双重间接块后，运行测试时，系统在执行 test manywrites 时卡住了。经过仔细检查，我发现问题出在 bmap 函数的逻辑上，特别是在处理双重间接块的部分。由于双重间接块的逻辑比较复杂，我不小心在索引块的分配和释放时遗漏了一些关键步骤，导致系统在执行写操作时进入了死循环。

为了解决这个问题，我首先仔细梳理了 bmap 函数的逻辑，逐行检查代码的执行路径。我发现，在为双重间接块分配新的物理块时，可能会因为未正确更新索引而导致死锁。为此，我在每次分配新块后，确保立即调用 log_write 函数将索引块写入日志，并在适当时机释放相关的缓冲区。经过这些修正，我再次编译并运行测试，发现问题得到了解决，所有测试均通过。这次问题的解决不仅让我更深入理解了文件系统的块管理机制，也让我意识到在处理复杂数据结构时必须非常谨慎。

实验心得

在这次实验中，我深入探讨并修改了 xv6 文件系统的块管理机制，以支持更大文件的存储。通过增加双重间接块，我成功扩展了文件系统的容量，突破了原有的 268 块限制。在这个过程中，我不仅学习了文件系统的核心结构，还锻炼了调试复杂代码的能力。

最具挑战性的一部分是理解并正确实现双重间接块的逻辑。在反复调试和测试中，我逐渐掌握了 xv6 的 inode 结构和块映射机制。特别是在解决死锁和逻辑错误时，我学会了如何分析问题根源并进行有效修正。这次实验让我认识到，面对复杂的系统架构时，细心和耐心是不可或缺的。此

外，这次实验也让我对文件系统的内部工作原理有了更深刻的理解，为后续的学习和研究打下了坚实的基础。

Symbolic links

实验目的

本实验的目的是在 xv6 操作系统中添加符号链接的支持。符号链接是一种文件类型，它通过路径名引用另一个文件或目录。实现符号链接有助于理解文件系统路径解析的工作原理，并加深对 xv6 操作系统内部机制的理解。通过完成此实验，您将学习如何扩展文件系统功能，并处理文件系统中的路径解析和递归查找等问题。

实验步骤

1. 我们需要为 symlink 系统调用分配一个系统调用号，并在相应的文件中声明。系统调用号在 kernel/syscall.h 中定义。

在 kernel/syscall.h 中添加以下代码，为 symlink 分配系统调用号：

```
#define SYS_symlink 22 // 实验 9.2
```

在 kernel/syscall.c 文件中，我们需要将 sys_symlink 函数加入到系统调用表中，添加 sys_symlink 函数的声明，以便系统能够识别新的系统调用：

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

2. 在 user/usys.pl 中添加 symlink 的入口声明：

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

在 user/user.h 文件中添加以下声明，以使用户程序可以调用该系统调用：

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

通过这些步骤，我们已经成功地将 symlink 系统调用集成到 xv6 中。

3. 定义文件类型

为了区分普通文件和符号链接，我们在 kernel/stat.h 文件中添加一个新的文件类型 T_SYMLINK：

```
#define T_SYMLINK 4    // 表示符号链接的文件类型
```

这个定义允许内核识别和处理符号链接，并将其与常规文件区分开来。

4. 添加打开标志

在 kernel/fcntl.h 文件中添加新的打开标志 O_NOFOLLOW，这个标志用于指示 open 系统调用是否应当跟随符号链接：

```
#define O_NOFOLLOW 0x004    // 防止跟随符号链接
```

O_NOFOLLOW 标志可以与 open 系统调用一起使用。当指定该标志时，open 应该打开符号链接本身，而不是链接指向的目标文件。

5. 实现 sys_symlink 系统调用

在 kernel/sysfile.c 文件中实现 sys_symlink 系统调用，该函数负责创建符号链接。符号链接的实现涉及创建一个特殊类型的 inode，并将目标路径写入 inode 的数据块中。

```

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

    if ((n = argstr(0, target, MAXPATH)) < 0 || argstr(1, path, MAXPATH) < 0) {
        return -1;
    }

    begin_op();
    // 创建符号链接的 inode
    if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
        end_op();
        return -1;
    }
    // 将目标路径写入节点
    if(writei(ip, 0, (uint64)target, 0, n) != n) {
        iunlockput(ip);
        end_op();
        return -1;
    }

    iunlockput(ip);
    end_op();
    return 0;
}

```

在这个函数中，首先通过 `create` 函数创建一个 `T_SYMLINK` 类型的 `inode`，然后将目标路径写入 `inode` 的数据块中。需要特别注意的是，在进行文件系统操作时，确保正确的加锁和解锁，以防止并发访问带来的问题。

6.修改 sys_open 函数以处理符号链接

在 `kernel/sysfile.c` 中修改 `sys_open` 函数，使其能够正确处理符号链接。如果文件路径指向符号链接，并且未指定 `O_NOFOLLOW` 标志，那么 `open` 应该解析符号链接并打开目标文件。

在这个实现中，我们在 `sys_open` 函数中添加了对符号链接的处理逻辑。当 `open` 系统调用遇到符号链接时，如果没有指定 `O_NOFOLLOW` 标志，它会递归解析符号链接，直到找到目标文件或达到指定的递归深度。

7.修改 namex 函数以处理符号链接

在 kernel/namei.c 文件中修改 namex 函数，使其在路径查找过程中能够正确处理符号链接，并递归解析实际文件。以下是修改后的 namex 函数：


```

static struct inode* namex(char *path, int nameiparent, char *name) {
    struct inode *ip, *next;
    int depth = 0; // 用于限制递归深度，避免循环符号链接
    char symlink_path[MAXPATH];

    if(*path == '/')
        ip = iget(ROOTDEV, ROOTINO);
    else
        ip = idup(myproc()->cwd);

    while((path = skipelem(path, name)) != 0){
        ilock(ip);

        // 处理符号链接
        while (ip->type == T_SYMLINK && depth < 10) {
            if (readi(ip, 0, (uint64)symlink_path, 0, ip->size) != ip->size) {
                iunlockput(ip);
                return 0;
            }
            symlink_path[ip->size] = '\0';
            iunlockput(ip);

            ip = namei(symlink_path);
            if (ip == 0) {
                return 0;
            }

            ilock(ip);
            depth++;
        }

        if(ip->type != T_DIR){
            iunlockput(ip);
            return 0;
        }

        if(nameiparent && *path == '\0'){
            iunlock(ip);
            return ip;
        }
    }
}

```

```

    }

    if((next = dirlookup(ip, name, 0)) == 0){
        iunlockput(ip);
        return 0;
    }

    iunlockput(ip);
    ip = next;
}

if(nameiparent){
    iput(ip);
    return 0;
}
return ip;
}

```

在 `namex` 函数中，我们通过递归的方式处理符号链接，并且增加了对循环符号链接的检测。如果检测到循环或递归深度超过设定值（如 10 次），函数将返回错误，防止程序陷入无限循环。

8.运行 `symlinktest` 确认所有测试都通过，验证符号链接功能的正确性：

```
./grade-lab-fs symlinktest
```

```

salad14@Salad:~/xv6-labs-2021$ ./grade-lab-fs symlinktest
make: 'kernel/kernel' is up to date.
== Test running symlinktest == (1.4s)
== Test    symlinktest: symlinks ==
    symlinktest: symlinks: OK
== Test    symlinktest: concurrent symlinks ==
    symlinktest: concurrent symlinks: OK

```

实验中遇到的问题

在实验的过程中，我遇到了一个问题，就是在实现 `sys_open` 系统调用时，程序在处理符号链接时会出现循环引用，导致栈溢出并触发系统崩溃。具体来说，当我创建了两个符号链接相互引用对方时，系统在解析路径时陷入了无限递归。起初，我并未考虑到符号链接可能会形成循环引

用，导致系统在遇到这种情况时无法正常处理。这个问题让我意识到，必须在解析符号链接的过程中加入循环检测机制，以避免这种递归陷阱。

为了解决这个问题，我在 `sys_open` 和 `namex` 函数中增加了对符号链接解析深度的限制。在每次递归解析符号链接时，我增加一个计数器，并且将这个计数器的最大值限制在 10 以内。如果在解析过程中符号链接的深度超过了这个阈值，系统将返回一个错误，并停止进一步解析。通过这种方法，我成功地解决了符号链接循环引用的问题，确保了系统的稳定性和正确性。经过修改和测试，`symlinktest` 测试通过了所有的验证。

实验心得

在这次实验中，我深入理解了符号链接的实现原理以及文件系统路径解析的复杂性。通过实现 `sys_symlink` 系统调用，我不仅增强了对 xv6 文件系统的理解，还进一步熟悉了操作系统在处理文件和目录路径时的底层机制。

最大的收获在于，我意识到处理符号链接时，需要特别注意可能的循环引用问题。通过引入深度限制和递归解析策略，我成功避免了符号链接可能引发的无限递归问题。在解决这些问题的过程中，我不仅提升了代码编写的严谨性，还学会了如何在代码设计中考虑边界情况和潜在的陷阱。这次实验给了我一次难得的机会，让我更加全面地理解了操作系统的设计原则和实现细节，同时也增强了我在面对复杂系统时的调试能力和解决问题的信心。

Lab: mmap

`mmap` 和 `munmap` 系统调用允许 UNIX 程序 对他们的地址空间进行详细控制。他们可以使用在进程之间共享内存，将文件映射到进程地址 空格，以及作为用户级页面错误方案的一部分，例如 讲座中讨论了垃圾收集算法。在本实验中，您将向 xv6 添加 `mmap` 和 `munmap`，重点介绍内存映射文件。

实验开始前请切换到mmap分支

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

手册页（`man 2 mmap`）显示了 `mmap` 的以下声明：

```
void *mmap(void *addr, size_t length, int prot, int flags,  
           int fd, off_t offset);
```

mmap 可以通过多种方式调用 但此实验室只需要 与内存映射文件相关的功能子集。您可以假设 addr 始终为零, 这意味着 kernel 应该决定映射文件的虚拟地址。mmap 返回该地址, 或者 0xffffffff 它失败了。length 是要映射的字节数;可能不是 与文件的长度相同。prot 指示是否应映射内存 可读、可写和/或可执行;你可以假设 该 prot 是 PROT_READ 或 PROT_WRITE 或两者兼而有之。flags 将为 MAP_SHARED、 这意味着对映射内存的修改应该 写回文件, 或者 MAP_PRIVATE 这意味着他们不应该。您不必实施任何 Flags 中的其他位。fd 是要映射的文件的打开文件描述符。您可以假设 offset 为零 (这是起点 在要映射的文件中)。

如果映射同一 MAP_SHARED 文件的进程不共享物理页, 那也没关系。

munmap (addr, length) 应该删除 指示的地址范围。如果进程修改了内存和 将其映射 MAP_SHARED, 则修改应首先为 写入文件。munmap 调用可能只涵盖 部分, 但您可以假设它 将在开头、结尾或整个片段 (但不是 区域中间的孔)。

实验目的

本次实验旨在向 xv6 操作系统添加 mmap 和 munmap 系统调用, 实现对进程地址空间的详细控制。通过实现这两个系统调用, 我们可以实现内存映射文件的功能, 包括共享内存、将文件映射到进程地址空间等。这有助于理解虚拟内存管理和页面错误处理的机制。

实验步骤

1.添加 mmap 和 munmap 系统调用号和声明。

修改 kernel/syscall.h:

```
#define SYS_mmap    22  
#define SYS_munmap  23
```

修改 kernel/syscall.c:

```
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);

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

修改 user/usys.pl:

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

修改 user/user.h:

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

上述步骤是为了在系统调用机制中为 mmap 和 munmap 系统调用分配编号、声明函数，并确保这些系统调用可以在用户空间调用时正确映射到内核中的相应函数实现。

2. 定义表示内存映射区域的结构体 vm_area，并在进程结构中添加 VMA 数组。

vm_area 结构体用于表示通过 mmap 系统调用映射到进程地址空间的文件区域。定义 vma 数组是为了记录每个进程的所有内存映射区域，并在需要时对其进行管理，例如处理页面错误、取消映射等操作。

```

#define NVMA 16 // 定义进程最大支持的 VMA 数量

struct vm_area {
    uint64 addr;    // mmap address
    int len;        // mmap memory length
    int prot;       // 权限（读/写）
    int flags;      // mmap 标志位（MAP_SHARED 或 MAP_PRIVATE）
    int offset;     // 文件偏移
    struct file* f; // 映射的文件指针
};

struct proc {
    ...
    struct vm_area vma[NVMA]; // VMA 数组，用于存储进程的虚拟内存区域
};

```

3.实现 mmap 系统调用所需的权限和标志位。

在 kernel/fcntl.h 中定义 MAP_SHARED 和 MAP_PRIVATE 标志位：

```

#define MAP_SHARED 0x01 // 共享映射
#define MAP_PRIVATE 0x02 // 私有映射

```

MAP_SHARED 和 MAP_PRIVATE 标志位分别用于定义映射区域的共享和私有性质。这些标志位将在 mmap 系统调用实现中用于设置映射区域的行为。

4.实现 mmap 系统调用，处理内存映射的相关操作。

在 kernel/sysfile.c 中实现 sys_mmap() 函数：

```

uint64 sys_mmap(void) {
    struct file *f;
    int len, prot, flags, fd, offset;
    uint64 addr;

    if (argint(1, &len) < 0 || argint(2, &prot) < 0 || argint(3, &flags) < 0 ||
        argfd(4, &fd, &f) < 0 || argint(5, &offset) < 0)
        return -1;

    // 检查标志位合法性
    if (flags != MAP_SHARED && flags != MAP_PRIVATE)
        return -1;

    // 分配 VMA 结构并初始化
    struct proc *p = myproc();
    for (int i = 0; i < NVMA; i++) {
        if (!p->vma[i].f) {
            addr = MMAPMINADDR;
            p->vma[i].addr = addr;
            p->vma[i].len = len;
            p->vma[i].prot = prot;
            p->vma[i].flags = flags;
            p->vma[i].offset = offset;
            p->vma[i].f = filedup(f); // 增加文件引用计数
            return addr;
        }
    }

    return -1; // 没有可用的 VMA
}

```

sys_mmap 函数实现了内存映射的核心功能，包括参数检查、分配 VMA 结构、记录映射区域信息，并返回映射的起始地址。使用 filedup 增加文件引用计数，确保文件不会在映射期间被关闭。

5.实现 munmap 系统调用，处理内存取消映射的相关操作。

在 kernel/sysfile.c 中实现 sys_munmap() 函数：

```

uint64 sys_munmap(void) {
    uint64 addr;
    int length;

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

    struct proc *p = myproc();
    for (int i = 0; i < NVMA; i++) {
        if (p->vma[i].addr == addr && p->vma[i].len == length) {
            uvmunmap(p->pagetable, addr, length/PGSIZE, 1);
            fileclose(p->vma[i].f);
            p->vma[i].f = 0; // 释放 VMA
            return 0;
        }
    }

    return -1; // 未找到匹配的 VMA
}

```

sys_munmap 函数用于处理内存区域的取消映射。通过查找与给定地址和长度匹配的 VMA，取消映射相应的内存区域，并关闭文件句柄，释放 VMA 结构。

6. 修改 exit 和 fork 系统调用，以支持 VMA 处理。
7. 在发生页面错误时，正确处理 mmap 映射的内存。

修改 kernel/trap.c 中的 usertrap() 函数：


```

void usertrap(void) {
    ...
    uint64 va = r_stval();
    struct proc *p = myproc();

    for (int i = 0; i < NVMA; i++) {
        struct vm_area *v = &p->vma[i];
        if (va >= v->addr && va < v->addr + v->len) {
            char *mem = kalloc();
            memset(mem, 0, PGSIZE);
            if (readi(v->f->ip, 0, (uint64)mem, v->offset + (va - v->addr), PGSIZE) < 0) {
                kfree(mem);
                break;
            }
            if (mappages(p->pagetable, va, PGSIZE, (uint64)mem, PTE_W | PTE_R | PTE_U) < 0) {
                kfree(mem);
                break;
            }
            return;
        }
    }
    ...
}

```

在页面错误处理函数中，检查触发错误的地址是否属于 mmap 映射的区域。如果是，则分配一个新的物理页面，并将相应文件内容加载到页面中，最后将页面映射到用户地址空间中。

8. 设置脏页标志位

在 kernel/riscv.h 中定义脏页标志位 PTE_D，表示页面已被修改。

```
#define PTE_D (1L << 7)
```

在 kernel/vm.c 中实现 uvmgetdirty() 和 uvmsetdirtywrite() 两个函数，用于读取和设置脏页标志位。

9. 编译调试

mmaptest 测试：

```
$ 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
```

make grade

```
== Test running mmaptest ==
$ make qemu-gdb
(4.2s)
== 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 ==
$ make qemu-gdb
usertests: OK (158.7s)
== Test time ==
time: OK
Score: 140/140
```

实验中遇到的问题

在进行 mmap 和 munmap 实验的过程中，我遇到了几个问题，其中一个较大的挑战是实现页面错误处理中的内存映射问题。在最初的实现中，当发生页面错误时，我直接使用 readi 函数从文件中读取数据并映射到用户空间，结果系统经常崩溃，报出非法内存访问的错误。经过调试，发现问题在于页面错误处理中，文件指针 file 和文件 inode 之间的关系没有处理好。readi 函数期望的是一个 inode 指针，而不是文件指针，这导致了参数传递中的不匹配，引发了系统崩溃。

为了解决这个问题，我在处理页面错误时，先从文件指针中提取 inode 指针，再调用 readi 函数读取文件内容。这样，问题解决了，页面错误处理也正常了，mmap 和 munmap 功能顺利实现。

在实现 exit 和 fork 函数的修改时，程序在 fork 之后有时会出现文件引用计数错误，导致文件关闭时出现资源泄露或者进程退出时系统崩溃。分析后发现，在 fork 函数中，复制父进程的虚拟内存区域（VMA）时，没有正确处理文件的引用计数，导致多个进程共享同一个文件时，引用计数不一致。在修改过程中，我在复制 VMA 的同时，增加了对文件指针的引用计数，并确保在进程退出时正确释放资源，解决了资源泄露的问题。

实验心得

在这个实验中，我深入研究并实现了 mmap 和 munmap 系统调用，这个过程对我来说充满了挑战，同时也带来了许多学习和成长的机会。通过这个实验，我不仅在理论上理解了内存管理的原理，还在实践中体验到了如何将这些原理应用于操作系统的实际开发中。

实验的开始阶段，我主要集中在了解 mmap 和 munmap 的基本概念以及它们在操作系统中的重要性。mmap 允许程序将文件映射到进程的地址空间，从而实现更高效的文件访问和内存共享，而 munmap 则负责取消这种映射。在实现 mmap 时，我特别关注了懒加载内存的方式，通过在页面错误时动态分配内存，避免了预先分配所有内存所带来的资源浪费。这种设计在处理大文件时尤其重要，它不仅提高了内存使用效率，还保证了系统的稳定性。

在实际编码过程中，我遇到了许多意想不到的问题。例如，在处理页面错误时，我发现需要仔细管理页表和内存映射，否则很容易导致系统崩溃或者内存泄漏。在调试时，我多次遇到由于未正确处理内存页标志位而导致的错误。这促使我深入研究了 RISC-V 的页面管理机制，并进一步理解了如何通过标志位控制内存页的读写权限。最终，通过对代码的不断调整和优化，我成功实现了 mmap 和 munmap，并保证它们能够正确处理各种边界情况。

除了技术上的挑战，这个实验还让我体会到了良好的代码结构和清晰的逻辑对于系统开发的重要性。在实现 mmap 和 munmap 的过程中，我学会了如何合理地设计数据结构，如何有效地管理资源，以及如何处理进程之间的内存共享。这些经验不仅增强了我对操作系统内核的理解，也提升了我在系统编程中的整体能力。