

操作系统课程设计报告

姓名：潘越

学号：2353788

选题：xv6及labs课程项目

源码托管链接：https://github.com/lvren1485/OS_Course_Design

环境搭建

为了完成2021年版xv6项目实验，我选择了在VMware虚拟机中搭建Linux开发环境。具体配置如下：

- 主机操作系统：Windows 11
- 虚拟机软件：VMware Workstation Pro 16
- 虚拟机操作系统：Ubuntu 20.04 LTS
- RISC-V工具链：`qemu-system-misc`, `gcc-riscv64-linux-gnu` 等

1. 安装与配置VMware Workstation Pro

首先，从VMware官网下载并安装VMware Workstation Pro。安装过程遵循常规向导，无特殊配置。

2. 创建并安装Ubuntu 20.04 LTS虚拟机

在VMware中创建新的虚拟机，并选择“稍后安装操作系统”。

- 操作系统类型：选择Linux
- 版本：Ubuntu 64位
- 处理器核心：2个
- 内存：4GB
- 磁盘空间：20GB

完成虚拟机创建后，将Ubuntu 20.04 LTS的ISO镜像文件挂载到虚拟机，然后启动虚拟机进行系统安装。安装过程中选择默认的“正常安装”和“清除整个磁盘并安装Ubuntu”，并设置好用户名和密码。

3. 软件源更新与环境准备

进入Ubuntu桌面后，打开终端，进行软件源更新和工具链安装。

1. 更新本地软件源列表

```
sudo apt-get update
```

该命令会从软件源服务器下载最新的软件包列表，以确保我们能安装最新版本的软件。

2. 安装RISC-V工具链和QEMU模拟器

```
sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

- `git`: 版本控制工具，用于克隆xv6代码库。
- `build-essential`: 包含 `gcc`, `g++`, `make` 等核心编译工具。
- `gdb-multiarch`: 用于RISC-V架构的GDB调试器。
- `qemu-system-misc`: xv6运行所需的QEMU模拟器。
- `gcc-riscv64-linux-gnu` 和 `binutils-riscv64-linux-gnu`: 用于RISC-V架构的交叉编译工具链。

4. 编译与运行xv6

1. 克隆xv6代码库

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

项目代码将下载到 `xv6-labs-2021` 文件夹中。

2. 切换到指定分支

进入项目目录，并切换到实验util分支。

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

```
yukiip@yukiip-virtual-machine:~$ git clone git://g.csail.mit.edu/xv6-labs-2021  
正克隆到 '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  
接收对象中: 100% (7051/7051), 17.20 MiB | 2.32 MiB/s, 完成.  
处理 delta 中: 100% (3702/3702), 完成.  
warning: 远程 HEAD 指向一个不存在的引用, 无法检出。  
  
yukiip@yukiip-virtual-machine:~$ cd xv6-labs-2021/  
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ git checkout util  
分支 'util' 设置为跟踪来自 'origin' 的远程分支 'util'。  
切换到一个新分支 'util'
```

3. 编译并运行xv6

在项目根目录下执行 `make qemu` 命令，编译内核并启动QEMU模拟器。

```
make qemu
```

如果一切顺利，将看到xv6的启动信息，并最终进入 \$ 的shell提示符。

```
[new branch] > > >  
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ 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-de  
vice,drive=x0,bus=virtio-mmio-bus.0  
  
xv6 kernel is booting  
  
hart 2 starting  
hart 1 starting  
init: starting sh  
$
```

Lab1 Utilities

Boot xv6

1. 获取用于实验的 xv6 源代码并检出 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
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

xv6-labs-2021 仓库与书中的 xv6-riscv 略有不同，主要是增加了一些文件。相关信息可以查看 git 日志: `$ git log`

Git 允许跟踪对代码的更改。例如，完成某个练习后，为了检查进度，可以通过运行以下命令提交更改: `git checkout util`

```
$ 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(-)
$
```

可以使用以下命令跟踪更改。运行 `git diff` 将显示自上次提交以来对代码的更改，而运行 `git diff origin/xv6-labs-2021` 将显示相对于最初的 xv6-labs-2021 代码的更改。这里，`origin/xv6-labs-2021` 是下载用于本课程的初始代码所在的 git 分支名称。

2. 构建并运行Xv6:

```
$ make qemu
riscv64-unknown-elf-gcc -c -o kernel/entry.o kernel/entry.s
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -
DSOL_UTIL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -
I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/start.o kernel/start.c
...
xv6 kernel is booting

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

如果在提示符下输入 `ls`，输出应类似于以下内容:

```
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README     2 2 2226
xargstest.sh 2 3 93
cat        2 4 23896
echo       2 5 22728
forktest   2 6 13088
grep       2 7 27256
init       2 8 23832
kill       2 9 22696
ln         2 10 22656
ls         2 11 26128
mkdir      2 12 22800
rm         2 13 22792
sh         2 14 41664
stressfs   2 15 23800
usertests  2 16 156016
grind      2 17 37968
wc         2 18 25040
zombie    2 19 22192
console    3 20 0
$
```

这些是 `mkfs` 在初始文件系统中包含的文件；大多数是可以运行的程序。刚才运行的其中一个程序是 `ls`。

xv6 没有 `ps` 命令，但如果输入 `ctrl-p`，内核将打印每个进程的信息。如果现在尝试，会看到两行输出：一行是 `init`，另一行是 `sh`。

要退出 `qemu`，请输入： `Ctrl-a x`。

sleep

实验目的

1. 实现 UNIX 程序 `sleep` 以用于 xv6；
2. 实现该 `sleep` 程序应暂停指定的用户数量的时间片。时间片是由 xv6 内核定义的时间概念，即两个定时器芯片中断之间的时间。解决方案应在文件 `user/sleep.c` 中。

实验步骤

1. 创建 `sleep.c` 文件

在 `user` 目录下，创建一个名为 `sleep.c` 的文件。

2. 编写 `sleep.c` 程序

在 `sleep.c` 中，编写一个程序，该程序接受一个命令行参数（表示 ticks 数量），并使当前进程暂停相应的 ticks 数量。在编写时需要注意以下几点：

- 如果用户没有提供参数或者提供了多个参数，程序应该打印出错误信息。
- 命令行参数作为字符串传递，可以使用 `atoi`（参见 `user/ulib.c`）将其转换为整数。
- 使用系统调用 `sleep`，最后确保 `main` 调用 `exit()` 以退出程序。

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

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

```

{
    // 检查参数数量
    if(argc != 2) {
        fprintf(2, "Usage: sleep <ticks>\n");
        exit(1);
    }

    // 将字符串参数转换为整数
    int ticks = atoi(argv[1]);

    if (ticks < 0) {
        fprintf(2, "Error: ticks must be a positive integer\n");
        exit(1);
    }

    // 调用系统调用 sleep
    sleep(ticks);

    exit(0);
}

```

3. 编辑 Makefile

将编写好的 `sleep` 程序添加到 Makefile 的 `UPROGS` 中。

- 打开 Makefile: `$ vim Makefile`
- 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)\``。

```

1  #include "kernel/types.h"
2  #include "user/user.h"
3
4  int main(int argc, char *argv[])
5  {
6      // 检查参数数量
7      if(argc != 2) {
8          // Usage: sleep <ticks>\n
9          exit(1);
10     }

11     // 将字符串参数转换为整数
12     int ticks = atoi(argv[1]);
13
14     if (ticks < 0) {
15         // Error: ticks must be a positive integer\n
16         exit(1);
17     }

18     // 调用系统调用 sleep
19     sleep(ticks);
20
21     exit(0);
22 }

```

4. 编译并测试程序

使用 `make qemu` 编译 xv6 并启动虚拟机，然后在 xv6 shell 中测试运行该程序：

- 在终端中，运行 `make qemu` 命令编译 xv6。

- 在 xv6 shell 中运行编写的 sleep 程序。

5. 单元测试

使用 `./grade-lab-util sleep` 进行单元测试，确保程序的正确性。

实验结果

从 xv6 shell 运行程序：

```
$ make qemu  
...  
init: starting sh  
$ sleep 10  
(一段时间内无事发生)  
$
```

如果程序在以下情况下暂停，则解决方案是正确的。如上所示运行。运行以查看是否确实通过了睡眠测试。

```
xv6 kernel is booting  
hart 1 starting  
hart 2 starting  
init: starting sh  
$ sleep 10  
$
```

分析讨论

在本实验中，成功实现了 xv6 操作系统下的 sleep 程序。实验的关键步骤包括正确解析命令行参数、调用 xv6 内核提供的 sleep 系统调用，以及处理异常输入。

实验结果表明，程序能有效暂停指定数量的时间片，并在暂停结束后返回。此过程验证了程序的功能和正确性，同时也增强了对 xv6 系统调用机制的理解。

通过编写 sleep 程序，我们学会了如何在 xv6 环境中创建用户级应用程序，了解了时间片的概念以及系统调用的实现方式。这对于深入理解 xv6 内核及其调度机制有着重要意义。

此外，本次实验还锻炼了处理用户输入、进行错误检查和使用系统调用的能力，这些技能在开发更复杂的操作系统应用时非常有用。未来可以进一步探索 xv6 中其他系统调用的实现，增强对操作系统内核的全面理解。

pingpong

实验目的

1. 编写一个程序，使用 UNIX 系统调用通过一对管道在两个进程之间传递一个字节，每个方向各一个管道。父进程应向子进程发送一个字节；子进程应打印": received ping"，其中 是其进程 ID，然后通过管道将字节写回父进程并退出；父进程应从子进程读取字节，打印": received pong"，然后退出。你的解决方案应放在文件 `user/pingpong.c` 中。
2. 理解父进程和子进程的关系及其执行顺序。学习使用管道进行进程间通信，实现父进程和子进程之间的数据交换。
3. 掌握进程同步的概念，确保父进程和子进程在适当的时机进行通信。

实验步骤

1. 编写 pingpong 程序：

在 user/ 目录下创建一个名为 pingpong.c 的文件，代码如下：

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

int main(int argc, char *argv[])
{
    // 创建两个管道
    // parent_to_child[0] 用于读取, parent_to_child[1] 用于写入
    int parent_to_child[2];
    int child_to_parent[2];
    char buf[1];

    if (pipe(parent_to_child) < 0 || pipe(child_to_parent) < 0) {
        fprintf(2, "pipe failed\n");
        exit(1);
    }

    int pid = fork();
    if (pid < 0) {
        fprintf(2, "fork failed\n");
        exit(1);
    }

    if (pid == 0) { // 子进程
        close(parent_to_child[1]); // 关闭不需要的写端
        close(child_to_parent[0]); // 关闭不需要的读端

        // 从父进程读取一个字节
        if (read(parent_to_child[0], buf, 1) != 1) {
            fprintf(2, "child read error\n");
            exit(1);
        }

        printf("%d: received ping\n", getpid());

        // 向父进程写入一个字节
        if (write(child_to_parent[1], buf, 1) != 1) {
            fprintf(2, "child write error\n");
            exit(1);
        }

        close(parent_to_child[0]);
        close(child_to_parent[1]);
        exit(0);
    } else { // 父进程
        close(parent_to_child[0]); // 关闭不需要的读端
        close(child_to_parent[0]); // 关闭不需要的写端

        // 向子进程写入一个字节
        buf[0] = 'x';
    }
}
```

```
if (write(parent_to_child[1], buf, 1) != 1) {
    fprintf(2, "parent write error\n");
    exit(1);
}

// 从子进程读取一个字节
if (read(child_to_parent[0], buf, 1) != 1) {
    fprintf(2, "parent read error\n");
    exit(1);
}

printf("%d: received pong\n", getpid());

close(parent_to_child[1]);
close(child_to_parent[0]);
exit(0);
}
```

```
4 int main(int argc, char *argv[])
5 {
6     // 创建两个管道
7     // parent_to_child[0] 用于读取, parent_to_child[1] 用于写入
8     int parent_to_child[2];
9     int child_to_parent[2];
10    char buf[1];
11
12    if (pipe(parent_to_child) < 0 || pipe(child_to_parent) < 0) {
13        fprintf(2, "pipe failed\n");
14        exit(1);
15    }
16
17    int pid = fork();
18    if (pid < 0) {
19        fprintf(2, "fork failed\n");
20        exit(1);
21    }
22
23    if (pid == 0) { // 子进程
24        close(parent_to_child[1]); // 关闭不需要的写端
25        close(child_to_parent[0]); // 关闭不需要的读端
26
27        // 从父进程读取一个字节
28        if (read(parent_to_child[0], buf, 1) != 1) {
29            fprintf(2, "child read error\n");
30            exit(1);
31        }
32
33        printf("%d: received ping\n", getpid());
34
35        // 向父进程写入一个字节
36        if (write(child_to_parent[1], buf, 1) != 1) {
37            fprintf(2, "child write error\n");
38            exit(1);
39        }
40
41        close(parent_to_child[0]);
42        close(child_to_parent[1]);
43        exit(0);
44    } else { // 父进程
45        close(parent_to_child[0]); // 关闭不需要的读端
46        close(child_to_parent[1]); // 关闭不需要的写端
47
48        // 向子进程写入一个字节
49        buf[0] = 'x';
50        if (write(parent_to_child[1], buf, 1) != 1) {
51            fprintf(2, "parent write error\n");
52            exit(1);
53        }
54
55        // 从子进程读取一个字节
56        if (read(child_to_parent[0], buf, 1) != 1) {
57            fprintf(2, "parent read error\n");
58            exit(1);
59        }
60
61        printf("%d: received pong\n", getpid());
62
63        close(parent_to_child[1]);
64        close(child_to_parent[0]);
65        exit(0);
66    }
67 }
```

2. 更新 `Makefile`:

在 `Makefile` 中找到 `UPROGS` 变量的定义，并添加 `pingpong`

3. 在终端中运行以下命令来编译 xv6 并启动:

```
$ make qemu  
...  
init: starting sh  
$ pingpong
```

结果应该输出:

```
4: received ping  
3: received pong
```

实验结果

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ 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-de  
vice,drive=x0,bus=virtio-mmio-bus.0  
  
xv6 kernel is booting  
  
hart 2 starting  
hart 1 starting  
init: starting sh  
$ pingpong  
4: received ping  
3: received pong  
$
```

此输出表明:

1. 子进程接收到父进程发送的字节并打印其进程 ID 和消息 "received ping"。
2. 子进程将字节发送回父进程。
3. 父进程接收到来自子进程的字节并打印其进程 ID 和消息 "received pong"。

分析讨论

1. 进程间通信：在这个实验中，父进程和子进程通过两个管道实现了双向通信。管道是一种 UNIX 提供的 IPC（进程间通信）机制，允许数据在进程之间流动。通过 pipe 系统调用，实验创建了两个管道 p 和 q，分别用于父进程向子进程发送数据以及子进程向父进程返回数据。管道的使用使得进程间的数据传输变得简单有效，同时也展示了 UNIX 系统下进程通信的基础。
2. 在这个实验中，父进程首先写入数据到管道 p，然后等待子进程读取该数据。子进程读取后，打印消息并写回数据到管道 q，父进程再从 q 读取数据，完成整个通信流程。这种严格的执行顺序保证了进程同步，确保了数据的正确传递。
3. 进程同步：进程同步是确保多个进程按照预期顺序执行的关键。在这个实验中，进程同步主要通过管道和 wait 系统调用实现。父进程通过 write 将数据写入管道 p，然后通过 read 从管道 q 读取数据。由于 read 是阻塞操作，父进程会等待直到子进程写入数据到管道 q。这种机制保证了父进程在子进程完成任务之前不会继续执行。此外，父进程通过 wait 系统调用等待子进程结束，进一步确保了进程的同步性。

primes

实验目的

1. 编写一个使用管道实现的并发版本的素数筛选算法。这一想法来自于Unix管道的发明者Doug McIlroy。请参考本页中间的图片及其周围的文本了解具体实现方法。相关解决方案应保存在文件 user/primes.c 中。
2. 使用管道和 fork 来建立管道。第一个进程将数字 2 到 35 送入管道。对于每一个质数，应将安排创建一个进程，通过管道从左邻右舍读取数据，并通过另一个管道向右邻右舍写入数据。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. 创建源文件：

在 user 目录下，创建一个名为 primes.c 的文件。在 primes.c 中，编写程序以实现功能。其实现原理如下：

- 定义常量 READEND、WRITEEND、ERROREND，分别表示进程自己独立的文件描述符 fd，标准输入 (0)、标准输出 (1)、标准错误 (2)。
- 使用 pipe(p) 创建一个管道 p，将用于存放 2 到 35 之间的所有数字。

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

// 筛选素数的函数
void sieve(int in_fd) {
    int prime;
    int num;

    // 读取第一个数字，它一定是素数
    if (read(in_fd, &prime, sizeof(prime))) {
        printf("prime %d\n", prime);

        // 创建管道用于与下一个进程通信
        int p[2];
        pipe(p);

        if (fork() == 0) {
            // 子进程：关闭不需要的文件描述符
            close(p[1]); // 关闭写端
            sieve(p[0]); // 递归处理
            close(p[0]);
            exit(0);
        } else {
            // 父进程：关闭不需要的文件描述符
            close(p[0]); // 关闭读端

            // 读取剩余数字，过滤掉能被 prime 整除的
            while (read(in_fd, &num, sizeof(num))) {
                if (num % prime != 0) {
                    write(p[1], &num, sizeof(num));
                }
            }
        }
    }
}
```

```
        close(p[1]); // 关闭写端, 通知子进程结束
        wait(0);    // 等待子进程结束
    }
}

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

    if (fork() == 0) {
        // 子进程: 开始筛选
        close(p[1]); // 关闭写端
        sieve(p[0]); // 开始筛选过程
        close(p[0]);
        exit(0);
    } else {
        // 父进程: 生成数字2-35
        close(p[0]); // 关闭读端

        for (int i = 2; i <= 35; i++) {
            write(p[1], &i, sizeof(i));
        }

        close(p[1]); // 关闭写端, 通知子进程结束
        wait(0);    // 等待子进程结束
    }

    exit(0);
}
```

```
1 #include "kernel/types.h"
2 #include "user/user.h"
3
4 // 筛选素数的函数
5 void sieve(int in_fd) {
6     int prime;
7     int num;
8
9     // 读取第一个数字，它一定是素数
10    if (read(in_fd, &prime, sizeof(prime))) {
11        printf("prime %d\n", prime);
12
13        // 创建管道用于与下一个进程通信
14        int p[2];
15        pipe(p);
16
17        if (fork() == 0) {
18            // 子进程：关闭不需要的文件描述符
19            close(p[1]); // 关闭写端
20            sieve(p[0]); // 递归处理
21            close(p[0]);
22            exit(0);
23        } else {
24            // 父进程：关闭不需要的文件描述符
25            close(p[0]); // 关闭读端
26
27            // 读取剩余数字，过滤掉能被prime整除的
28            while (read(in_fd, &num, sizeof(num))) {
29                if (num % prime != 0) {
30                    write(p[1], &num, sizeof(num));
31                }
32            }
33
34            close(p[1]); // 关闭写端，通知子进程结束
35            wait(0); // 等待子进程结束
36        }
37    }
38 }
39
40 int main(int argc, char *argv[]) {
41     int p[2];
42     pipe(p);
43
44     if (fork() == 0) {
45         // 子进程：开始筛选
46         close(p[1]); // 关闭写端
47         sieve(p[0]); // 开始筛选过程
48         close(p[0]);
49         exit(0);
50     } else {
51         // 父进程：生成数字2-35
52     }
53 }
```

```

52     close(p[0]); // 关闭读端
53
54     for (int i = 2; i <= 35; i++) {
55         write(p[1], &i, sizeof(i));
56     }
57
58     close(p[1]); // 关闭写端, 通知子进程结束
59     wait(0); // 等待子进程结束
60 }
61
62     exit(0);
63 }
```

2. 父进程:

- 创建一个管道 p。
- 调用 `fork()` 创建一个子进程。
- 关闭管道的读端 `p[0]`，因为父进程只需要向管道写入数据。
- 将数字2到35写入管道的写端 `p[1]`。
- 关闭管道的写端 `p[1]`，防止继续写入。
- 调用 `wait(0)` 等待子进程结束，确保在所有子进程完成工作后再退出。

3. 子进程:

- 调用 `filter` 函数开始筛选素数。
- 在 `filter` 函数中：
 - 读取管道中的第一个数并打印，这是一个素数。
 - 创建一个新的管道pr。
 - 再次调用 `fork()` 创建一个新的子进程。
 - 关闭新管道的读端 `pr[0]`，父进程处理读取和筛选的逻辑：
 - 读取父管道中的数，检查是否能被当前素数整除。
 - 如果不能整除，将数写入新管道。
 - 关闭新管道的写端 `pr[1]`，等待子进程结束。
 - 新的子进程递归调用 `filter` 函数继续筛选下一个素数，直到管道中没有数可读。

4. 添加到Makefile：将程序以 \$U/_primes 的形式，添加到 Makefile 的 UPROGS 中。

实验结果

```

$ make qemu
...
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
$
```

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ 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-de
vice,drive=x0,bus=virtio-mmio-bus.0

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
$
```

分析讨论

1. 本实验采用了Doug McIlroy提出的基于管道的并发素数筛选算法，通过进程间通信实现数据传递与处理。具体实现方法如下：

- 管道与进程创建：程序通过调用 `pipe()` 创建管道，并使用 `fork()` 创建子进程。父进程负责将数字2到35写入管道，子进程则负责从管道中读取数据并进行筛选。
- 数据传递与筛选：子进程通过递归调用 `filter` 函数实现素数的筛选。每个子进程从管道中读取一个数并判断其是否为素数。若为素数，则打印并创建新的管道和子进程，继续筛选剩余数据。这种方法通过进程间的递归调用实现了并发处理，每个进程仅处理一个素数及其倍数的过滤工作，极大地提高了程序的可扩展性和效率。
- 资源管理：在程序中，父进程在完成数据写入后关闭管道的写端，并通过 `wait()` 等待子进程结束。子进程在读取数据完成后，关闭相应的管道端口，避免了资源泄漏。

2. 优点与不足

- 优点：
 - 并发处理：通过使用多个进程和管道实现并发处理，提高了程序的执行效率。
 - 代码结构清晰：采用递归调用的方式使得代码结构简洁明了，每个子进程只负责处理一个素数及其倍数的过滤工作。
- 不足：
 - 资源消耗大：每个素数的筛选都需要创建新的进程和管道，随着数据量的增加，系统资源（如文件描述符和进程数量）的消耗也会显著增加。
 - 进程管理复杂：由于每个素数的筛选都依赖于递归调用，需要管理多个子进程的创建和结束，增加了程序的复杂性和调试难度。

3. 实验中遇到的问题与解决：

- 问题：父进程向管道中写入数据后，子进程可能会因为管道中的数据尚未完全写入而无法正确读取，导致素数筛选过程中的数据丢失或错误。

- 解决：父进程在将数字2到35写入管道后，关闭写端 `p[1]`，表示写入完成。子进程在读取数据时，正确处理读端口关闭的情况，通过检测读取返回值是否为0来判断管道是否已经关闭。

find

实验目的

- 编写一个简单版本的 UNIX 查找程序：查找目录树中带有特定名称的所有文件。相关解决方案应放在 `user/find.c` 文件中。
- 理解文件系统中目录和文件的基本概念和组织结构。
- 熟悉在 xv6 操作系统中使用系统调用和文件系统接口进行文件查找操作。
- 应用递归算法实现在目录树中查找特定文件。

实验步骤

- 首先查看 `user/ls.c` 以了解如何读取目录。

`user/ls.c` 中包含一个 `fmtname` 函数，用于格式化文件的名称。它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。如果名称的长度大于等于 `DIRSIZ`，则直接返回名称。否则，将名称拷贝到一个静态字符数组 `buf` 中，并用空格填充剩余的空间，保证输出的名称长度为 `DIRSIZ`。

- 创建 `find.c` 文件：

在 `user` 目录下创建一个新的文件 `find.c`，然后编写如下代码：

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

// 比较文件名是否匹配
char* fmtname(char *path) {
    static char buf[DIRSIZ+1];
    char *p;

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

    // 返回文件名部分
    if(strlen(p) >= DIRSIZ)
        return p;
    memmove(buf, p, strlen(p));
    memset(buf+strlen(p), 0, DIRSIZ-strlen(p));
    return buf;
}

// 递归查找文件
void find(char *path, char *filename) {
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;
```

```
// 打开目录
if((fd = open(path, 0)) < 0) {
    fprintf(2, "find: cannot open %s\n", path);
    return;
}

// 获取文件状态
if(fstat(fd, &st) < 0) {
    fprintf(2, "find: cannot stat %s\n", path);
    close(fd);
    return;
}

switch(st.type) {
case T_FILE:
    // 如果是文件，检查是否匹配
    if(strcmp(fmtname(path), filename) == 0) {
        printf("%s\n", path);
    }
    break;

case T_DIR:
    // 如果是目录，递归处理
    if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {
        printf("find: path too long\n");
        break;
    }
    strcpy(buf, path);
    p = buf+strlen(buf);
    *p++ = '/';

    // 读取目录项
    while(read(fd, &de, sizeof(de)) == sizeof(de)) {
        if(de.inum == 0)
            continue;

        // 跳过 ".." 和 "."
        if(strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
            continue;

        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0;

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

int main(int argc, char *argv[]) {
    if(argc != 3) {
        fprintf(2, "Usage: find <directory> <filename>\n");
    }
}
```

```
    exit(1);
}

find(argv[1], argv[2]);
exit(0);
}
```

```
1 #include "kernel/types.h"
2 #include "kernel/stat.h"
3 #include "user/user.h"
4 #include "kernel/fs.h"
5
6 // 比较文件名是否匹配
7 char* fmtname(char *path) {
8     static char buf[DIRSIZ+1];
9     char *p;
10
11     // 找到最后一个斜杠后的文件名
12     for(p=path+strlen(path); p >= path && *p != '/'; p--)
13     || ;
14     p++;
15
16     // 返回文件名部分
17     if(strlen(p) >= DIRSIZ)
18     || return p;
19     memmove(buf, p, strlen(p));
20     memset(buf+strlen(p), 0, DIRSIZ-strlen(p));
21     return buf;
22 }
23
24 // 递归查找文件
25 void find(char *path, char *filename) {
26     char buf[512], *p;
27     int fd;
28     struct dirent de;
29     struct stat st;
30
31     // 打开目录
32     if((fd = open(path, 0)) < 0) {
33         fprintf(2, "find: cannot open %s\n", path);
34         return;
35     }
36
37     // 获取文件状态
38     if(fstat(fd, &st) < 0) {
39         fprintf(2, "find: cannot stat %s\n", path);
40         close(fd);
41         return;
42     }
43
44     switch(st.type) {
45     case T_FILE:
46         // 如果是文件，检查是否匹配
47         if(strcmp(fmtname(path), filename) == 0) {
48             printf("%s\n", path);
49         }
50         break;
51
52     case T_DIR:
53         // 如果是目录，递归处理
54         if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {
55             printf("find: path too long\n");
56             break;
57         }
58         strcpy(buf, path);
59         p = buf+strlen(buf);
60         *p++ = '/';
61
62         // 读取目录项
63         while(read(fd, &de, sizeof(de)) == sizeof(de)) {
64             if(de.inum == 0)
65                 continue;
66
67             // 跳过 ".." 和 "."
68             if(strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
69                 continue;
70
71             memmove(p, de.name, DIRSIZ);
72             p[DIRSIZ] = 0;
73         }
74     }
75 }
```

```

74     |         |         // 递归查找子目录
75     |         |         find(buf, filename);
76     |         |
77     |         break;
78     }
79     close(fd);
80 }
81
82 int main(int argc, char *argv[]) {
83     if(argc != 3) {
84         fprintf(2, "Usage: find <directory> <filename>\n");
85         exit(1);
86     }
87
88     find(argv[1], argv[2]);
89     exit(0);
90 }

```

- o fmtname函数：

- 功能：从路径中提取文件名。它通过查找路径中最后一个斜杠后的部分来获取文件名。
- 处理逻辑：遍历路径字符串找到最后一个斜杠的位置，然后返回其后的部分。如果文件名长度小于DIRSIZ，则将其拷贝到静态缓冲区buf中，并用空格填充剩余的空间。

- o find函数：

- 参数：路径 (path) 和目标文件名 (target) 。
- 功能：递归查找指定路径下的文件或目录，匹配目标文件名并打印其路径。
- 打开路径：尝试打开指定路径，如果失败则打印错误信息并返回。
- 获取文件状态：调用fstat获取文件的状态信息（类型、大小等）。
- 根据文件类型处理：
 - 如果是文件 (T_FILE)：比较文件名，如果匹配则打印路径。
 - 如果是目录 (T_DIR)：遍历目录内容，跳过"."和".."，并递归调用find函数查找子目录。

- o main函数：

- 参数检查：确保传入了正确数量的参数（路径和文件名）。
- 调用find函数：传入用户输入的路径和文件名。
- 退出程序：调用exit(0)退出程序。

3. 注意事项：

- o 递归处理：使用递归遍历目录和子目录，查找目标文件。
- o 跳过特殊目录：在遍历目录时，跳过"."和".."以避免无限递归。
- o 字符串处理：使用strcmp进行字符串比较，使用memmove和memset处理文件名。
- o 错误处理：在打开文件和获取文件状态时进行错误检查，确保程序的健壮性。

4. 更新 `Makefile`：在 `Makefile` 中将 `find` 程序添加到 `UPROGS` 中。

5. 运行程序。

实验结果

```
$ make qemu  
...  
init: starting sh  
$ echo > b  
$ mkdir a  
$ echo > a/b  
$ find . b  
./b  
./a/b  
$
```

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ 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-de  
vice,drive=x0,bus=virtio-mmio-bus.0  
  
xv6 kernel is booting  
  
hart 2 starting  
hart 1 starting  
init: starting sh  
$ echo > b  
$ mkdir a  
$ echo > a/b  
$ find . b  
./b  
./a/b  
$ █
```

程序输出符合预期，成功查找到目标文件并打印其路径。

分析讨论

通过这次实验，我们实现了一个简单的find命令，该命令能递归遍历目录树并查找目标文件。实验过程中，我们学会了：

1. 目录读取：通过参考user/ls.c，我们了解了如何读取目录内容以及提取文件名。
2. 递归算法：递归方法让我们能够深入到子目录中进行查找，同时避免了无限递归的问题。
3. 文件系统接口的使用：在Xv6操作系统中，我们使用系统调用和文件系统接口来实现文件和目录的操作。
4. 字符串处理：C语言中的字符串处理需要特别注意，使用strcmp进行字符串比较，避免了直接使用 == 进行比较的错误。
5. 错误处理：在文件操作中，我们增加了错误检查，提高了程序的健壮性。

xargs

实验目的

1. 编写一个简单版的 UNIX xargs 程序：从标准输入读取每一行，并将行作为参数传递给命令执行。
代码位于文件 user/xargs.c 中。
2. 熟悉命令行参数获取和处理：实验需要解析命令行参数并进行适当处理，包括选项解析和参数拆分。
3. 学习执行外部命令：实验中需要调用 exec 函数来执行外部命令，理解执行外部程序的基本原理。

实验步骤

1. 创建 xargs.c 文件：在 user 目录下创建一个名为 xargs.c 的文件。
2. 编写程序代码：编写 xargs.c 文件中的程序，以实现以下功能：
 - 从标准输入读取命令和参数：程序从标准输入读取用户输入的命令和参数。
 - 解析命令和参数：将输入的字符串分割成多个参数，每个参数都是一个独立的字符串。
 - 执行命令：创建一个新的进程来执行用户输入的命令，通过调用 fork 和 exec 函数实现。
 - 等待命令执行完成：调用 wait 函数阻塞当前进程，直到子进程结束。
 - 循环处理：程序重复上述步骤，直到从标准输入读取到 EOF。

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

#define MAX_LINE 512

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(2, "Usage: xargs <command> [args...]\n");
        exit(1);
    }

    char buf[MAX_LINE];
    char *xargv[MAXARG];
    int i, n;
    char ch;

    // 设置初始命令参数
    for (i = 1; i < argc; i++) {
        xargv[i-1] = argv[i];
    }
    xargv[argc-1] = 0; // exec 需要以0结尾的参数数组

    while (1) {
        i = 0;
        // 读取一行输入
        while ((n = read(0, &ch, 1)) > 0 && ch != '\n') {
            if (i < MAX_LINE - 1) {
                buf[i++] = ch;
            }
        }
        if (n <= 0 && i == 0) // 没有更多输入
            break;

        buf[i] = 0; // 终止字符串

        // 添加输入行作为最后一个参数
        xargv[argc-1] = buf;

        // 创建子进程执行命令
        if (fork() == 0) {
```

```
    exec(xargv[0], xargv);
    fprintf(2, "exec %s failed\n", xargv[0]);
    exit(1);
} else {
    wait(0); // 等待子进程完成
}

exit(0);
}
```

```

4   #include "kernel/param.h"
5
6   #define MAX_LINE 512
7
8   int main(int argc, char *argv[]) {
9       if (argc < 2) {
10           fprintf(2, "Usage: xargs <command> [args...]\n");
11           exit(1);
12       }
13
14       char buf[MAX_LINE];
15       char *xargv[MAXARG];
16       int i, n;
17       char ch;
18
19       // 设置初始命令参数
20       for (i = 1; i < argc; i++) {
21           xargv[i-1] = argv[i];
22       }
23       xargv[argc-1] = 0; // exec 需要以0结尾的参数数组
24
25       while (1) {
26           i = 0;
27           // 读取一行输入
28           while ((n = read(0, &ch, 1)) > 0 && ch != '\n') {
29               if (i < MAX_LINE - 1) {
30                   buf[i++] = ch;
31               }
32           }
33
34           if (n <= 0 && i == 0) // 没有更多输入
35               break;
36
37           buf[i] = 0; // 终止字符串
38
39           // 添加输入行作为最后一个参数
40           xargv[argc-1] = buf;
41
42           // 创建子进程执行命令
43           if (fork() == 0) {
44               exec(xargv[0], xargv);
45               fprintf(2, "exec %s failed\n", xargv[0]);
46               exit(1);
47           } else {
48               wait(0); // 等待子进程完成
49           }
50       }
51
52       exit(0);
53   }

```

3. 修改 Makefile：在 Makefile 的 UPROGS 中添加 `$U/_xargs\``。
4. 编译并测试程序：在 xv6 shell 中测试运行 `xargs` 程序，并使用 `./grade-lab-util xargs` 进行单元测试。

实验结果

运行 `sh < xargstest.sh`:

理论输出:

```
$ sh < xargstest.sh
$ $ $ $ $ hello
hello
hello
$ $
```

实际输出:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sh < xargstest.sh
$ $ $ $ $ hello
hello
hello
$ $
```

分析讨论

1. 通过本次实验，掌握了以下关键技能：

- 命令行参数处理：学会了如何解析和处理命令行参数，理解了参数传递和选项解析的基本原理。
- 进程管理：了解了如何在 xv6 环境中创建和管理进程，通过 fork 创建子进程，并通过 exec 执行外部命令。
- 标准输入输出处理：学会了如何从标准输入读取数据，并将其作为命令参数传递执行。

通过截图

```
make[1]: 进入 '/home/yukiip/xv6-labs-2021'
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (11.6s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (1.6s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.6s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.4s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.6s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.7s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (2.2s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (2.4s)
== Test time ==
time: OK
Score: 100/100
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

Lab2 System calls

本实验旨在帮助了解系统调用跟踪的实现，并演示如何修改 xv6 操作系统以添加新功能。实验任务是添加一个新的 trace 系统调用，该调用用于调试。具体来说，需要创建一个名为 trace 的系统调用，并将一个整数 "mask" 作为参数。"mask" 的各个位表示要跟踪的系统调用。通过本实验，将熟悉内核级编程，包括修改进程结构、处理系统调用以及管理跟踪掩码。

System call tracing

实验目的

本实验旨在添加一个系统调用追踪功能，以便在后续实验中进行调试。具体要求如下：

1. 创建一个新的 `trace` 系统调用，用于控制追踪功能。
2. `trace` 系统调用应接受一个整数参数 "mask"，该参数的每个位表示要追踪的特定系统调用。
 - 例如，为了追踪 `fork` 系统调用，程序可以调用 `trace(1 << sys_fork)`，其中 `sys_fork` 是来自 `kernel/syscall.h` 的系统调用号。

3. 修改 xv6 内核，使其在每个系统调用即将返回时打印出一行信息（如果该系统调用的号码在 `mask` 中设置了）。

- 打印的信息应包括进程 ID、系统调用名称和返回值，但不需要打印系统调用的参数。

4. `trace` 系统调用应仅为调用它的进程及其后续 `fork` 的子进程启用追踪，而不影响其他进程。

实验步骤

1. 作为一个系统调用，先要定义一个系统调用的序号。系统调用序号的宏定义在 `kernel/syscall.h` 文件中。我们在 `kernel/syscall.h` 添加宏定义，模仿已经存在的系统调用序号的宏定义：

```
#define SYS_trace 22
```

```
1 // System call numbers
2 #define SYS_fork    1
3 #define SYS_exit    2
4 #define SYS_wait    3
5 #define SYS_pipe    4
6 #define SYS_read    5
7 #define SYS_kill    6
8 #define SYS_exec    7
9 #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup    10
12 #define SYS_getpid 11
13 #define SYS_sbrk   12
14 #define SYS_sleep  13
15 #define SYS_uptime 14
16 #define SYS_open    15
17 #define SYS_write   16
18 #define SYS_mknod   17
19 #define SYS_unlink  18
20 #define SYS_link    19
21 #define SYS_mkdir   20
22 #define SYS_close   21
23 #define SYS_trace   22
24 #define SYS_sysinfo 23
25
```

2. 官方已提供了用户态的 `trace` 函数（`user/trace.c`），因此我们只需在 `user/user.h` 文件中声明用户态可以调用 `trace` 系统调用。然而，关于该系统调用的参数和返回值的类型，我们需要进一步确认。

为了明确这些类型，我们需要查看 `trace.c` 文件。文件中有一句代码 `trace(atoi(argv[1])) < 0`，显示 `trace` 函数传入的是一个数字，并且与 0 进行比较。结合实验提示，可以确定传入参数的类型为 `int`，并且推测返回值类型也是 `int`。

基于以上分析，我们可以在内核中声明 trace 系统调用。

```
// system calls
int trace(int);
```

```
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int trace(int);
int sysinfo(struct sysinfo *); // 添加这一行：sysinfo 函数原型
```

3. 接下来，我们需要查看 `user/usys.pl` 文件。该文件中使用 Perl 语言自动生成用户态系统调用接口的汇编语言文件 `usys.s`。因此，我们需要在 `user/usys.pl` 文件中加入以下语句：

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

```
17
18     entry("fork");
19     entry("exit");
20     entry("wait");
21     entry("pipe");
22     entry("read");
23     entry("write");
24     entry("close");
25     entry("kill");
26     entry("exec");
27     entry("open");
28     entry("mknod");
29     entry("unlink");
30     entry("fstat");
31     entry("link");
32     entry("mkdir");
33     entry("chdir");
34     entry("dup");
35     entry("getpid");
36     entry("sbrk");
37     entry("sleep");
38     entry("uptime");
39     entry("trace");
40     entry("sysinfo");
41
```

4. 执行 `ecall` 指令后会跳转至 `kernel/syscall.c` 文件中的 `syscall` 函数处，并执行该函数。`syscall` 函数的源码如下：

```
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]();
    } else {
        printf("%d %s: unknown sys call %d\n", p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}
```

5. 接下来，`p->trapframe->a0 = syscalls[num]();` 语句通过调用 `syscalls[num]()` 函数，并将返回值保存在 `a0` 寄存器中。`syscalls[num]()` 函数在当前文件中定义，调用具体的系统调用命令。把新增的 `trace` 系统调用添加到函数指针数组 `*syscalls[]` 上：

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

6. 在文件开头给内核态的系统调用 `trace` 加上声明，在 `kernel/syscall.c` 加上：

```
extern uint64 sys_trace(void);
```

7. 根据提示，`trace` 系统调用应该有一个参数，一个整数“mask(掩码)”，其指定要跟踪的系统调用。所以，在 `kernel/proc.h` 文件的 `proc` 结构体中，新添加一个变量 `mask`，使得每一个进程都有自己的 `mask`，即要跟踪的系统调用。

```
struct proc {
    ...
    int tracemask;           // Mask
};
```

```
85 // Per-process state
86 struct proc {
87     struct spinlock lock;
88
89     // p->lock must be held when using these:
90     enum procstate state;        // Process state
91     void *chan;                 // If non-zero, sleeping on chan
92     int killed;                 // If non-zero, have been killed
93     int xstate;                 // Exit status to be returned to parent's wait
94     int pid;                    // Process ID
95
96     // wait_lock must be held when using this:
97     struct proc *parent;        // Parent process
98
99     // these are private to the process, so p->lock need not be held.
100    uint64 kstack;              // Virtual address of kernel stack
101    uint64 sz;                  // Size of process memory (bytes)
102    pagetable_t pagetable;      // User page table
103    struct trapframe *trapframe; // data page for trampoline.S
104    struct context context;     // swtch() here to run process
105    struct file *ofile[NOFILE]; // Open files
106    struct inode *cwd;          // Current directory
107    char name[16];              // Process name (debugging)
108    int trace_mask;
109 };
110
```

8. 然后可以在 `kernel/sysproc.c` 给出 `sys_trace` 函数的具体实现了，只要把传进来的参数给到现有进程的 `mask` 即可：

```

uint64
sys_trace(void)
{
    int mask;
    // 取 a0 寄存器中的值返回给 mask
    if(argint(0, &mask) < 0)
        return -1;

    // 把 mask 传给现有进程的 mask
    myproc()->tracemask = mask;
    return 0;
}

```

9. 接下来，我们需要实现输出功能。由于 RISCV 的 C 规范是将返回值放在 a0 寄存器中，所以在调用系统调用时，我们只需判断是否为 `mask` 规定的输出函数，如果是，就进行输出操作。首先，在 `kernel/proc.h` 文件中，`proc` 结构体中的 `name` 字段是线程的名字，不是函数调用的函数名。因此，我们需要在 `kernel/syscall.c` 中定义一个数组来存储系统调用的名字。这里需要注意，系统调用的名字必须按顺序排列，第一个为空字符串。当然，也可以去掉第一个空字符串，但在取值时需要将索引减一，因为系统调用号是从 1 开始的。

```

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"};

```

10. 可以在 `kernel/syscall.c` 文件的 `syscall` 函数中添加打印系统调用情况的语句。`mask` 是按位判断的，因此需要使用按位运算。进程序号可以通过 `p->pid` 获取，函数名称可以从我们刚刚定义的数组 `syscall_names[num]` 获取，返回值则是 `p->trapframe->a0`。以下是更新后的 `syscall` 函数代码：

```

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

    num = p->trapframe->a7;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        p->trapframe->a0 = syscalls[num]();
        if ((1 << num) & p->tracemask) {
            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;
    }
}

```

11. 然后在 `kernel/proc.c` 中 `fork` 函数调用时，添加子进程复制父进程的 `mask` 的代码：

```
...
*(np->trapframe) = *(p->trapframe);

np->tracemask = p->tracemask; // 复制 tracemask

// Cause fork to return 0 in the child.
np->trapframe->a0 = 0;
...
```

12. 最后在 Makefile 的 `UPROGS` 中添加 `$U/_trace\``。

13. 编译并进行测试。

实验结果

```
$ trace 32 grep hello README
...
$ trace 2147483647 grep hello README
...
$ grep hello README
$
$ trace 2 usertests forkforkfork
usertests starting
```

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 968
4: syscall read -> 235
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
6: syscall fork -> 7
```

分析讨论

1. 实验中遇到的问题及其解答：

- 命名问题导致的错误：最初，我将 `tracemask` 命名为 `mask`，在实现过程中遇到了一些寄存器读取错误。由于在函数实现和调用过程中，`mask` 这个名字可能会与其他变量名冲突，导致读取和赋值操作出现错误。通过更改变量名为 `tracemask`，避免了这种命名冲突问题。

- 系统调用编号定义位置错误：在定义 `sys_trace` 编号时，若未严格按照 `kernel/syscall.h` 文件中的顺序添加，可能导致系统调用编号错乱。因此，需要仔细检查和按照现有系统调用编号的顺序添加新的编号。
 - `fork` 时 `tracemask` 继承问题：在实现子进程继承父进程 `tracemask` 的功能时，若未在 `fork` 函数中正确复制 `tracemask`，将导致子进程无法正确追踪系统调用。通过在 `kernel/proc.c` 文件中，添加 `np->tracemask = p->tracemask;` 解决了这个问题。
2. 实验反思：本次实验成功实现了系统调用追踪功能，通过添加新的 `trace` 系统调用，能够按需控制追踪特定的系统调用。在实现过程中，通过解决变量命名冲突、确保系统调用编号的正确定义、生成用户态系统调用接口、正确继承 `tracemask` 以及完善输出信息，最终达到了预期的实验目标。此功能在后续实验和开发中将大大提升调试效率和系统理解深度。

Sysinfo

实验目的

在本实验中，我们将添加一个名为 `sysinfo` 的系统调用，用于收集系统运行信息。该系统调用需要一个参数：指向 `struct sysinfo` 的指针（参见 `kernel/sysinfo.h`）。内核需要填写该结构体的以下字段：`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. 定义一个系统调用的序号。系统调用序号的宏定义在 `kernel/syscall.h` 文件中。在 `kernel/syscall.h` 添加宏定义 `SYS_sysinfo` 如下：

```
#define SYS_sysinfo 23
```

2. 在 `user/usys.pl` 文件加入下面的语句：

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

3. 在 `user/user.h` 中添加 `sysinfo` 结构体以及 `sysinfo` 函数的声明：

```
struct stat;
struct rtcdate;
// 添加 sysinfo 结构体
struct sysinfo;

// system calls
...
int sysinfo(struct sysinfo *);
```

4. 在 `kernel/syscall.c` 中新增 `sys_sysinfo` 函数的定义：

```
extern uint64 sys_sysinfo(void);
```

5. 在 `kernel/syscall.c` 中函数指针数组新增 `sysinfo`：

```
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", "sysinfo"};
```

6. 在进程中已经保存了当前进程的状态，因此我们可以直接遍历所有进程，判断它们的状态是否为 `UNUSED` 并进行计数。根据 `proc` 结构体的定义，访问进程状态时必须加锁。我们在 `kernel/proc.c` 中新增了一个名为 `nproc` 的函数，用于获取可用进程的数量，代码如下：

```
uint64
nproc(void)
{
    struct proc *p;
    uint64 num = 0;

    for (p = proc; p < &proc[NPROC]; p++)
    {
        // add lock
        acquire(&p->lock);
        // if the processes's state is not UNUSED
        if (p->state != UNUSED)
        {
            // the num add one
            num++;
        }
        // release lock
        release(&p->lock);
    }
    return num;
}
```

7. 在 `kernel/kalloc.c` 中添加一个 `free_mem` 函数，用于收集可用内存的数量。参考 `kernel/kalloc.c` 文件中的 `kalloc()` 和 `kfree()` 函数可以看出，内核通过 `kmem.freelist` 链表维护未使用的内存。链表的每个节点对应一个页表大小（`PGSIZE`）。分配内存时，从链表头部取走一个页表大小的内存；释放内存时，使用头插法将其插入到该链表。因此，计算未使用内存的字节数 `freemem` 只需遍历该链表，得到链表节点数，并与页表大小（4KB）相乘即可。

```
// Return the number of bytes of free memory
uint64
free_mem(void)
{
    struct run *r;
    uint64 num = 0;
    acquire(&kmem.lock);
    r = kmem.freelist;
    while (r){
        num++;
        r = r->next;
    }
    release(&kmem.lock);
    return num * PGSIZE;
```

```
}
```

8. 在 `kernel/defs.h` 中添加上述两个新增函数的声明:

```
// kalloc.c
...
uint64 free_mem(void);

// proc.c
...
uint64 nproc(void);
```

9. 在 `sys_sysinfo` 函数的实现中, 首先使用 `argaddr` 函数读取用户态数据 `sysinfo` 的指针地址。然后, 将内核中获取的 `sysinfo` 数据, 按 `sizeof(info)` 大小复制到该指针指向的内存位置。以下是我们在 `kernel/sysproc.c` 文件中添加的 `sys_sysinfo` 函数的具体实现:

```
// add header
#include "sysinfo.h"

uint64
sys_sysinfo(void)
{
    uint64 addr;
    struct sysinfo info;
    struct proc *p = myproc();

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

    info.freemem = free_mem();
    info.nproc = nproc();
    if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
        return -1;
    return 0;
}
```

10. 最后在 `user` 目录下添加一个 `sysinfo.c` 用户程序:

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

int main(int argc, char *argv[])
{
    // param error
    if (argc != 1){
        fprintf(2, "Usage: %s need not param\n", argv[0]);
        exit(1);
    }

    struct sysinfo info;
```

```

    sysinfo(&info);
    // print the sysinfo
    printf("free space: %d\nused process: %d\n", info.freemem, info.nproc);
    exit(0);
}

```

11. 在 `Makefile` 的 `UPROGS` 中添加:

```
$U/_sysinfotest\
```

12. 编译并运行 xv6 进行测试。

实验结果

```

$ make qemu
...
init: starting sh
$ sysinfo
free space: 133386240
used process: 3
$ sysinfotest
sysinfotest: start
sysinfotest: OK

```

```

yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ 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-de
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

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

```

分析讨论

1. 实验中遇到的问题与解决:

在实现 `sysinfo` 系统调用时，出现了未定义引用 `sysinfo` 的链接错误。

确保在 `user/user.h` 中正确声明了 `sysinfo` 函数，并在 `user/usys.pl` 文件中正确添加了条目 `entry("sysinfo")`。

在获取可用内存大小时，访问链表节点可能导致竞争条件。

在遍历 `kmem.freelist` 链表时，加锁以确保线程安全。

在获取进程数量时，遍历进程表可能导致竞争条件。

在访问进程状态时，加锁以确保线程安全。

2. 实验心得：通过本次实验，我成功实现了 `sysinfo` 系统调用，深入理解了系统调用的实现过程、锁机制在内核编程中的重要性，以及如何调试和解决实际编程中的问题。

通过截图

```
make[1]: 正在目录 /home/yukiip/xv6-labs-2021
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (9.9s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.6s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.4s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (45.2s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (6.8s)
== Test time ==
time: OK
Score: 35/35
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

Lab3 Page tables

Speed up system calls

实验目的

某些操作系统（例如 Linux）通过在用户空间和内核之间共享一个只读区域的数据来加速某些系统调用。这消除了在执行这些系统调用时进入内核的需求。为了学习如何将映射插入页表，第一个任务是在 `xv6` 中为 `getpid()` 系统调用实现此优化。

当每个进程创建时，在 `USYSCALL`（在 `memlayout.h` 中定义的一个虚拟地址）处映射一个只读页面。在该页面的开始位置存储一个 `struct usyscall`（也在 `memlayout.h` 中定义），并将其初始化为存储当前进程的 `PID`。对于本实验，用户空间侧已经提供了 `ugetpid()`，并且将自动使用 `USYSCALL` 映射。

实验步骤

1. `allocproc` 函数用于分配一个空闲的进程表（`struct proc`）实例，以表示一个新的进程。该函数还会为进程分配必要的物理内存、创建用户页表、并设置上下文以执行 `forkret`。在分配 `trapframe` 部分时，`trapframe` 是一个进程的属性。类似地，我们可以为进程添加一个 `usyscall` 属性，用于保存 `usyscall` 页面的地址。进程的属性应定义在 `kernel/proc.h` 中。根据提示，我们在进程创建时需要为其分配和初始化 `usyscall` 页面。因此，我们需要修改 `kernel/proc.c` 中的 `allocproc()` 函数，并在分配 `trapframe` 后面添加分配 `usyscall` 页面的部分。

```
...
struct usyscall *usyscall;
...
```

```
85 // Per-process state
86 struct proc {
87     struct spinlock lock;
88
89     // p->lock must be held when using these:
90     enum procstate state;          // Process state
91     void *chan;                  // If non-zero, sleeping on chan
92     int killed;                 // If non-zero, have been killed
93     int xstate;                 // Exit status to be returned to parent's wait
94     int pid;                     // Process ID
95
96     // wait_lock must be held when using this:
97     struct proc *parent;         // Parent process
98
99     // these are private to the process, so p->lock need not be held.
100    uint64 kstack;              // Virtual address of kernel stack
101    uint64 sz;                  // Size of process memory (bytes)
102    pagetable_t pagetable;       // User page table
103    struct trapframe *trapframe; // data page for trampoline.S
104    struct context context;     // swtch() here to run process
105    struct file *ofile[NFILE];  // Open files
106    struct inode *cwd;          // Current directory
107    char name[16];              // Process name (debugging)
108    struct usyscall *usyscall;   // USYSCALL page
109 };
110
```

2. 配成功后，将当前进程的 `pid` 存入 `usyscall` 页面的开始处：

```
if((p->usyscall = (struct usyscall *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
}
p->usyscall->pid = p->pid;
```

3. `kalloc` 函数用于分配物理内存，但我们还需要完成从虚拟地址到物理地址的映射。这一过程需要在 `kernel/proc.c` 文件中的 `proc_pagetable()` 函数中利用 `mappages()` 函数来实现。

`proc_pagetable()` 函数用于为进程创建一个用户页表。用户页表用于映射用户进程的虚拟地址到物理地址。在此过程中，还会映射一些特殊页，如 `trampoline` 和 `trapframe`。我们需要在这个函数中添加一个 `usyscall` 页面的映射。

```
if(mappages(pagetable, USYSCALL, PGSIZE,
            (uint64)(p->usyscall), PTE_U | PTE_R) < 0){
    uvmfree(pagetable, 0);
    return 0;
}
```

4. 当前已完成了分配和映射工作，按照提示，我们还需要在必要的时候释放页面，比如终止进程时，我们需要在 `kernel/proc.c` 的 `freeproc()` 函数中，同理，将我们分配的 `usyscall` 和 `trapframe` 页面做相同处理，添加相关代码：

```
if(p->usyscall)
    kfree((void*)p->usyscall);
p->usyscall = 0;
```

5. 还需要在 `kernel/proc.c` 的 `proc_freepagetable()` 函数中释放我们之前建立的虚拟地址到物理地址的映射:

```
void proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    uvmunmap(pagetable, USYSCALL, 1, 0);
    uvmfree(pagetable, sz);
}
```

实验结果

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
```

```
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$ P
```

分析讨论

1. 实验中遇到的问题:

- 在启动 xv6 内核时，遇到了以下错误：

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
panic: freewalk: leaf
```

该错误通常与页表管理中的问题有关。具体来说，在处理页表释放时，可能存在对已经释放或未正确初始化的页表项进行操作的情况。

仔细检查页表的分配和释放逻辑，确保在分配页表时正确初始化各个页表项。最终发现问题出在 `proc_freepagetable()` 函数中，没有正确处理 `usyscall` 页面的释放。

- 页表映射问题：在实现 `mappages` 函数时，最初没有正确设置页面权限，导致 `usyscall` 页面无法正确映射。通过检查页表权限设置，我们确保了 `usyscall` 页面具有只读权限，从而解决了映射问题。

2. 实验心得：在本次实验中，通过实现将 `usyscall` 页面映射到用户空间并优化 `getpid()` 系统调用，我们深入了解了如何在 `xv6` 操作系统中处理虚拟地址和物理地址的映射。这不仅提升了系统调用的效率，还加深了我们对操作系统内核机制的理解。

Print a page table

实验目的

为了帮助可视化 RISC-V 页表，并且可能有助于未来的调试，该项目将需要编写一个函数来打印页表的内容。定义一个名为 `vmprint()` 的函数。它应该接收一个 `pagetable_t` 参数，并按照下面描述的格式打印该页表。在 `exec.c` 中，在 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)` 语句，以打印第一个进程的页表，不输出无效的 `PTE`。

实验步骤

1. 在 `kernel/vm.c` 中编写函数 `vmprint()`。可以参考 `freewalk()` 函数的实现。对于输出，可以采用循环和递归两种方式实现：

- 循环实现：使用三重循环遍历三级页目录。
- 递归实现：与 `freewalk()` 函数相似，通过检查 `(PTE_R | PTE_W | PTE_X) == 0` 来判断是否不是最低级页目录。如果上两级页目录的 `PTE` 索引到的内容是下一级页目录的 `PTE`，则这些 `PTE` 应该均不可读写执行。而最低级页目录的 `PTE` 索引到的是一个页表，其内容应满足读写执行的条件之一。根据这个条件，可以作为递归的出口。
这里实现了循环的方法。

```
// print page tables lab3-1
void vmprint(pagetable_t pagetable) {
    printf("page table %p\n", pagetable);
    // range top page dir
    const int PAGE_SIZE = 512;
    // 遍历最高级页目录
    for (int i = 0; i < PAGE_SIZE; ++i) {
        pte_t top_pte = pagetable[i];
        if (top_pte & PTE_V) {
            printf(..%d: pte %p pa %p\n", i, top_pte, PTE2PA(top_pte));
            // this PTE points to a lower-level page table.
            pagetable_t mid_table = (pagetable_t) PTE2PA(top_pte);
            // 遍历中间级页目录
            for (int j = 0; j < PAGE_SIZE; ++j) {
                pte_t mid_pte = mid_table[j];
                if (mid_pte & PTE_V) {
                    printf(.. ..%d: pte %p pa %p\n",
                           j, mid_pte, PTE2PA(mid_pte));
                    pagetable_t bot_table = (pagetable_t) PTE2PA(mid_pte);
                    // 遍历最低级页目录
                    for (int k = 0; k < PAGE_SIZE; ++k) {
                        pte_t bot_pte = bot_table[k];
                        if (bot_pte & PTE_V) {
                            printf(.. .. ..%d: pte %p pa %p\n",
                                   k, bot_pte, PTE2PA(bot_pte));
                        }
                    }
                }
            }
        }
    }
}
```

2. 在 kernel/defs.h 文件中添加函数原型

```
149 // uart.c
150 void uartinit(void);
151 void uartintr(void);
152 void uartputc(int);
153 void uartputc_sync(int);
154 int uartgetc(void);
155
156 // vm.c
157 void kvminit(void);
158 void kvmminithart(void);
159 void kvmmmap(pagetable_t, uint64, uint64, uint64, int);
160 int mappages(pagetable_t, uint64, uint64, uint64, int);
161 pagetable_t uvmcreate(void);
162 void uvminit(pagetable_t, uchar *, uint);
163 uint64 uvmalloc(pagetable_t, uint64, uint64);
164 void uvmdealloc(pagetable_t, uint64, uint64);
165 void uvmcopy(pagetable_t, pagetable_t, uint64);
166 void uvmfree(pagetable_t, uint64);
167 void uvmunmap(pagetable_t, uint64, uint64, int);
168 void uvmclear(pagetable_t, uint64);
169 void walkaddr(pagetable_t, uint64);
170 void copyout(pagetable_t, uint64, char *, uint64);
171 void copyin(pagetable_t, char *, uint64, uint64);
172 void copyinstr(pagetable_t, char *, uint64, uint64);
173 void vmprint(pagetable_t);
174 pte_t * walk(pagetable_t, uint64, int);
175
176 // plic.c
177 void plicinit(void);
178 void plicminithart(void);
179 int plic_claim(void);
180 void plic_complete(int);
181
```

3. 在 kernel/exec.c 的 exec 函数的 `return argc` 前插入 `if(p->pid==1) vmprint(p->pagetable)`。这里考虑 xv6 的启动过程, 推测 `pid==1` 的进程为 kernel/main.c 中 `main()` 函数中调用的 `userinit()` 函数所建立的控制台进程。

4. 编译。

实验结果

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ 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 1 starting
hart 2 starting
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. .0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. . .0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. . .1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. . . .2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. .511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. . .509: pte 0x0000000021fdd813 pa 0x0000000087f76000
.. . . .510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. . . .511: pte 0x0000000020001c0b pa 0x0000000080007000
.. . . . .
```

分析讨论

1. 在本实验中，我们编写了一个函数 `vmprint()`，以便打印 RISC-V 页表的内容。通过在 `exec.c` 文件中的适当位置调用该函数，可以有效地输出第一个用户进程（控制台进程）的页表信息，从而帮助我们可视化页表的结构和内容。
 - 函数实现：在 `vmprint()` 函数中，我们通过三重循环遍历三级页目录，每一层页目录都使用 `PTE_V` 位判断页表条目是否有效。当发现有效条目时，我们会递归进入下一层页目录，直到到达最低级页目录。对于最低级页目录中的每个有效条目，我们会输出对应的物理地址。
 - 代码插入位置：我们选择在 `exec` 函数的 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)`，目的是打印第一个用户进程（即控制台进程）的页表。通过这一修改，可以确保我们在 `xv6` 启动时获得页表的输出信息。
 - 结果验证：通过编译并运行修改后的 `xv6` 系统，我们成功输出了控制台进程的页表信息，从而验证了 `vmprint()` 函数的正确性和有效性。
2. 输出结果的可读性：通过输出的页表信息，可以直观地看到页表条目的层次结构和对应的物理地址。这对于理解和调试分页机制有很大帮助。然而，输出的格式可能需要进一步美化，以便在更复杂的场景下提供更好的可读性。
3. 递归与循环的选择：在本实验中，我们选择了循环的方法来实现页表遍历。尽管递归方法在逻辑上更简洁，但循环方法在控制复杂性和性能上可能具有一定优势。在实际应用中，两种方法各有优劣，可以根据具体需求进行选择。
4. 对无效 PTE 的处理：为了避免输出无效的页表条目，我们在遍历过程中添加了 `if (top_pte & PTE_V)` 的检查。这确保了输出结果仅包含有效的页表条目，从而提高了输出信息的有效性和准确性。

Detecting which pages have been accessed

实验目的

有些垃圾回收器（一种自动内存管理形式）可以通过获取哪些页面被访问过（读或写）的信息来获益。在本实验的这一部分中，将为 `xv6` 添加一个新功能，通过检查 RISC-V 页表中的访问位来检测并报告这些信息到用户空间。RISC-V 硬件页步行器在解决 TLB 未命中时会在 PTE 中标记这些位。

实验的最终目的是实现 `pgaccess()` 系统调用，它报告哪些页面被访问过。这个系统调用有三个参数。首先，它需要检查的第一个用户页面的起始虚拟地址。其次，它需要检查的页面数量。最后，它需要一个用户地址来指向一个缓冲区，以将结果存储到一个位掩码中（一个每页使用一个位的数据结构，其中第一页对应于最低有效位）。如果在运行 `pgtbltest` 时 `pgaccess` 测试用例通过，你将获得这一部分实验的全部分数。

实验步骤

1. 定义 PTE 标志位

在 `kernel/riscv.h` 文件中，定义以下页表条目（PTE）标志位：

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1) // readable
#define PTE_W (1L << 2) // writable
#define PTE_X (1L << 3) // executable
#define PTE_U (1L << 4) // user can access
#define PTE_A (1L << 6) // accessed
```

```

#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_A (1L << 6) // 6: accessed bit

```

2. 实现 `sys_pgaccess` 系统调用

实现 `sys_pgaccess`，其接收三个参数，分别为：1. 起始虚拟地址；2. 遍历页数目；3. 用户存储返回结果的地址。因为其是系统调用，故参数的传递需要通过 `argaddr`、`argint` 来完成。通过不断的 `walk` 来获取连续的 `PTE`，然后检查其 `PTE_A` 位，如果为 `1` 则记录在 `mask` 中，随后将 `PTE_A` 手动清 `0`。最后，通过 `copyout` 将结果拷贝给用户即可。在 `kernel/sysproc.c` 文件中，添加以下代码以实现 `sys_pgaccess` 系统调用：

```

int sys_pgaccess(void) {
    uint64 vaddr;
    int num;
    uint64 res_addr;
    argaddr(0, &vaddr);
    argint(1, &num);
    argaddr(2, &res_addr);

    struct proc *p = myproc();
    pagetable_t pagetable = p->pagetable;
    uint64 res = 0;

    for(int i = 0; i < num; i++) {
        pte_t *pte = walk(pagetable, vaddr + PGSIZE * i, 1);
        if(*pte & PTE_A) {
            *pte &= ~PTE_A;
            res |= (1L << i);
        }
    }

    copyout(pagetable, res_addr, (char*)&res, sizeof(uint64));
    return 0;
}

```

3. 内核启动与输出验证

4. 编写并运行测试用例

5. 验证测试结果

实验结果

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
page table 0x0000000087f6e000
0: pte 0x000000021fda801 pa 0x0000000087f6a000
0: pte 0x000000021fdac1f pa 0x0000000087f6b000
...
```

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ 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-de
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
..0: pte 0x0000000021fda401 pa 0x0000000087f69000
...0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
...1: pte 0x0000000021fda00f pa 0x0000000087f68000
...2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
...511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
...509: pte 0x0000000021fdd813 pa 0x0000000087f76000
...510: pte 0x0000000021fddc07 pa 0x0000000087f77000
...511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

分析讨论

1. 实验中遇到的问题及解决：

- 页表遍历中的错误：在实现 `sys_pgaccess` 时，最初遍历页表时漏掉了一些页，导致部分页面没有被正确检测到。这是由于在计算虚拟地址偏移时没有正确处理。通过仔细检查页表遍历逻辑，确保每个页面都被正确访问并检测到，解决了这个问题。
- 用户空间缓冲区的地址传递：最初实现时，没有正确处理用户空间缓冲区地址的传递和结果拷贝。通过使用 `copyout` 函数，将内核空间的结果拷贝到用户空间，解决了这个问题。
- 清除 `PTE_A` 标志位：在检测到页面被访问后，需要清除 `PTE_A` 标志位以便于后续的访问检测。最初实现时，忘记清除该标志位，导致重复检测到同一页面。通过在检测到访问后手动清除 `PTE_A` 标志位，解决了这个问题。

2. 实验心得：通过本实验，成功实现了一个基于 RISC-V 页表的页面访问检测系统调用

`pgaccess()`，并验证了其正确性。该系统调用能够有效地检测哪些页面被访问过，这对于垃圾回收器等自动内存管理系统的优化具有重要意义。实验过程中，通过解决页表遍历中的错误、正确处理用户空间缓冲区地址传递、以及确保清除 `PTE_A` 标志位，最终达成了实验目标。

通过截图

```
;/ ``$/d > kernel/kernel.sym
make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
== Test pgtbltest ==
$ make qemu-gdb
(14.8s)
== Test    pgtbltest: ugetpid ==
    pgtbltest: ugetpid: OK
== Test    pgtbltest: pgaccess ==
    pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (2.6s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(401.5s)
== Test    usertests: all tests ==
    usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

Lab4 Traps

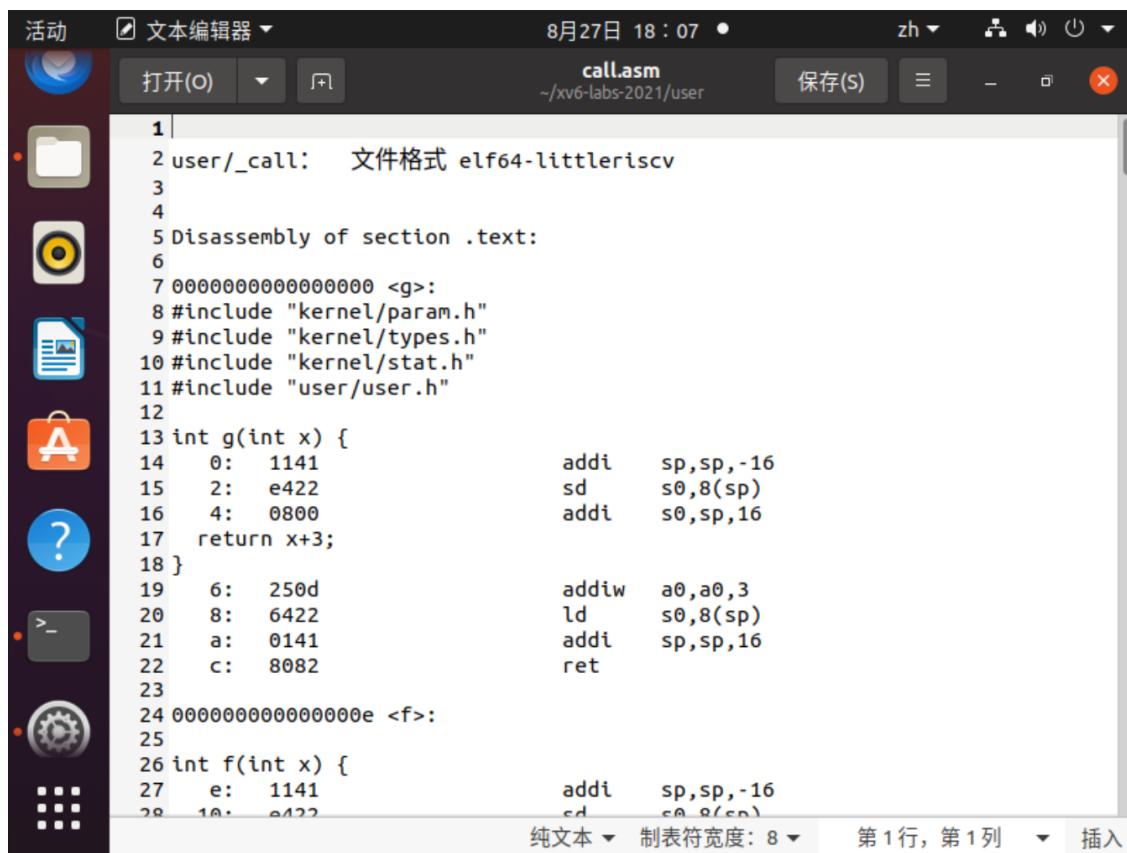
RISC-V assembly

实验目的

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。

实验步骤

1. 在xv6的命令行中输入运行`make fs.img`，编译`user/call.c`程序，得到可读性比较强的`user/call.asm`文件。



A screenshot of a Linux desktop environment (Ubuntu) showing a terminal window titled "call.asm". The terminal contains assembly code for a RISC-V program. The code defines two functions, g and f, and calls them from main. The assembly instructions show register usage and stack manipulation.

```
1 |
2 user/_call:    文件格式 elf64-littleriscv
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <g>:
8 #include "kernel/param.h"
9 #include "kernel/types.h"
10 #include "kernel/stat.h"
11 #include "user/user.h"
12
13 int g(int x) {
14     0: 1141             addi   sp,sp,-16
15     2: e422             sd     s0,8(sp)
16     4: 0800             addi   s0,sp,16
17     return x+3;
18 }
19     6: 250d             addiw  a0,a0,3
20     8: 6422             ld     s0,8(sp)
21     a: 0141             addi   sp,sp,16
22     c: 8082             ret
23
24 000000000000000e <f>:
25
26 int f(int x) {
27     e: 1141             addi   sp,sp,-16
28     10: e122             sd     s0,8(sp)
29
30
31
32
33
34
35
36
37
38
39 void main(void) {
40     1c: 1141             addi   sp,sp,-16
41     1e: e406             sd     ra,8(sp)
42     20: e022             sd     s0,0(sp)
43     22: 0800             addi   s0,sp,16
44     printf("%d %d\n", f(8)+1, 13);
45     24: 4635             li     a2,13|
46     26: 45b1             li     a1,12
47     28: 00000517         auipc a0,0x0
48     2c: 7c050513         addi   a0,a0,1984 # 7e8 <malloc+0xea>
49     30: 00000097         auipc ra,0x0
50     34: 610080e7         jalr   1552(ra) # 640 <printf>
51     exit(0);
52     38: 4501             li     a0,0
53     3a: 00000097         auipc ra,0x0
54     3e: 27e080e7         jalr   638(ra) # 2b8 <exit>
55
```

2. 阅读 call.asm 中的 `g` , `f` , 和 `main` 函数。

回答下列问题：

Q1

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？

在 RISC-V 架构中，寄存器 `a0` 到 `a7` 用于传递函数参数。具体地，前八个参数分别使用 `a0` 到 `a7` 这些寄存器传递。如果有更多参数，需要通过栈来传递。

在 `main` 函数中，对 `printf` 的调用如下：

```
printf("%d %d\n", f(8)+1, 13);
```

通过汇编代码可以看到参数的传递：

```
38
39 void main(void) {
40     1c: 1141             addi   sp,sp,-16
41     1e: e406             sd     ra,8(sp)
42     20: e022             sd     s0,0(sp)
43     22: 0800             addi   s0,sp,16
44     printf("%d %d\n", f(8)+1, 13);
45     24: 4635             li     a2,13|
46     26: 45b1             li     a1,12
47     28: 00000517         auipc a0,0x0
48     2c: 7c050513         addi   a0,a0,1984 # 7e8 <malloc+0xea>
49     30: 00000097         auipc ra,0x0
50     34: 610080e7         jalr   1552(ra) # 640 <printf>
51     exit(0);
52     38: 4501             li     a0,0
53     3a: 00000097         auipc ra,0x0
54     3e: 27e080e7         jalr   638(ra) # 2b8 <exit>
55
```

在这段代码中：

`li a2,13` 表示将 `13` 加载到 `a2` 寄存器中。

`li a1,12` 表示将 `f(8) + 1` 的结果 `12` 加载到 `a1` 寄存器中。

`printf` 调用时参数的寄存器分配：

第一个参数（格式字符串 `"%d %d\n"`）在 `a0`。

第二个参数（`f(8) + 1` 的结果 `12`）在 `a1`。

第三个参数（`13`）在 `a2`。

因此，`13` 保存在 `a2` 寄存器中。

Q2

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.)

`main` 的汇编代码中对函数的调用在哪里？对 `g` 的调用在哪里（提示：编译器可能会将函数内联）

查看 `call1.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。

```
23
24 000000000000000e <f>:
25
26 int f(int x) {
27     e:    1141           addi    sp,sp,-16
28     10:   e422           sd      s0,8(sp)
29     12:   0800           addi    s0,sp,16
30     return g(x);
31 }
32     14:   250d           addiw   a0,a0,3
33     16:   6422           ld      s0,8(sp)
34     18:   0141           addi    sp,sp,16
35     1a:   8082           ret
36
```

此外，编译器会进行内联优化，即在编译时计算出可以预先计算的结果，而不是在运行时进行函数调用。在 `main` 函数中，`printf` 包含一个对 `f` 的调用，但在汇编代码中，这个调用被直接替换为 `f(8)+1` 的结果 `12`。

```
37 0000000000000001c <main>:  
38  
39 void main(void) {  
40 1c: 1141 addi sp,sp,-16  
41 1e: e406 sd ra,8(sp)  
42 20: e022 sd s0,0(sp)  
43 22: 0800 addi s0,sp,16  
44 printf("%d %d\n", f(8)+1, 13);  
45 24: 4635 li a2,13  
46 26: 45b1 li a1,12  
47 28: 00000517 auipc a0,0x0  
48 2c: 7c050513 addi a0,a0,1984 # 7e8 <malloc+0xea>  
49 30: 00000097 auipc ra,0x0  
50 34: 610080e7 jalr 1552(ra) # 640 <printf>  
51 exit(0);  
52 38: 4501 li a0,0  
53 3a: 00000097 auipc ra,0x0  
54 3e: 27e080e7 jalr 638(ra) # 2b8 <exit>  
55
```

综上，在 `main` 函数中没有直接的函数调用指令，而是内联了 `f` 和 `g` 的计算结果。

Q3

At what address is the function printf located?

`printf` 函数位于哪个地址？

```
1112 000000000000000640 <printf>:  
1113  
1114 void  
1115 printf(const char *fmt, ...)  
1116  
1117 640: 711d addi sp,sp,-96  
1118 642: ec06 sd ra,24(sp)  
1119 644: e822 sd s0,16(sp)  
1120 646: 1000 addi s0,sp,32  
1121 648: e40c sd a1,8(s0)  
1122 64a: e810 sd a2,16(s0)  
1123 64c: ec14 sd a3,24(s0)  
1124 64e: f018 sd a4,32(s0)  
1125 650: f41c sd a5,40(s0)  
1126 652: 03043823 sd a6,48(s0)  
1127 656: 03143c23 sd a7,56(s0)  
1128 va_list ap;  
1129  
1130 va_start(ap, fmt);  
1131 65a: 00840613 addi a2,s0,8  
1132 65e: fec43423 sd a2,-24(s0)  
1133 vprintf(1, fmt, ap);  
1134 662: 85aa mv a1,a0  
1135 664: 4505 li a0,1  
1136 666: 00000097 auipc ra,0x0  
1137 66a: dce080e7 jalr -562(ra) # 434 <vprintf>  
1138
```

查阅得到其地址在 `0x640`。

Q4

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

在 main 中 printf 的 jalr 之后的寄存器 ra 中有什么值？

```
30
37 0000000000000001c <main>:
38
39 void main(void) {
40 1c: 1141           addi    sp,sp,-16
41 1e: e406           sd      ra,8(sp)
42 20: e022           sd      s0,0(sp)
43 22: 0800           addi    s0,sp,16
44 printf("%d %d\n", f(8)+1, 13);
45 24: 4635           li      a2,13
46 26: 45b1           li      a1,12
47 28: 00000517       auipc   a0,0x0
48 2c: 7c050513       addi    a0,a0,1984 # 7e8 <malloc+0xea>
49 30: 00000097       auipc   ra,0x0
50 34: 610080e7       jalr    1552(ra) # 640 <printf>
51 exit(0);
52 38: 4501           li      a0,0
53 3a: 00000097       auipc   ra,0x0
54 3e: 27e080e7       jalr    638(ra) # 2b8 <exit>
rr
```

34: jalr 1552(ra) # 640 <printf> 指令跳转到 printf 函数。

在执行 jalr 指令时，ra 寄存器会保存返回地址，也就是 jalr 指令的下一条指令的地址。在这种情况下：

34: jalr 1536(ra) 的下一条指令是 38: li a0,0。

所以，在执行 jalr 指令后，ra 寄存器中保存的值是 0x38，即 main 函数中 printf 调用之后的返回地址。

Q5

运行以下代码。

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

程序的输出是什么？这是将字节映射到字符的ASCII码表。

输出取决于RISC-V小端存储的事实。如果RISC-V是大端存储，为了得到相同的输出，你会把i设置成什么？是否需要将57616更改为其他值？

输出为 HE110 world。

若为大端对齐，i 需要设置为 0x726c6400，不需要改变 57616 的值（因为他是按照二进制数字读取的而非单个字符）。

Q6

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

在下面的代码中，“y=”之后将打印什么（注：答案不是一个特定的值）？为什么会发生这种情况？

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

在这段代码中，`printf` 函数的格式字符串要求两个整数参数，但实际只提供了一个整数参数 3。由于 `printf` 期望两个参数，而只提供了一个，这会导致未定义行为。具体来说，“y=”之后将打印什么取决于栈中紧接着的内容，这些内容可能是任何值。

这是由于以下几个原因导致的：

1. 参数不匹配：`printf` 函数的格式字符串包含两个 `%d`，但只提供了一个参数。这意味着函数会尝试从栈中获取第二个参数。
2. 未定义行为：C 语言标准中规定，当格式字符串的占位符数量与提供的参数数量不匹配时，行为是未定义的。这意味着编译器不会对这种情况做出任何保证，程序可能会打印垃圾值，崩溃，甚至可能正确运行（但这是偶然的）。
3. 栈内容未初始化：在调用 `printf` 时，函数会从栈中读取参数。由于没有提供第二个参数，`printf` 会读取一个未初始化的栈位置的值，导致打印出一个不可预测的值。

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. 在文件 `kernel/riscv.h` 中添加内联函数 `r_fp()` 读取栈帧值。

```
368 static inline uint64
369 r_fp()
370 {
371     uint64 x;
372     asm volatile("mv %0, s0" : "=r" (x));
373     return x;
374 }
375
```

2. 在 `kernel/printf.c` 文件中编写 `backtrace()` 函数以输出所有栈帧。函数的实现思路如下：

- 通过调用 `r_fp()` 函数读取寄存器 `s0` 中的当前函数栈帧 `fp`。
- 根据 RISC-V 的栈结构，`fp-8` 存放返回地址，`fp-16` 存放原栈帧。通过原栈帧可以得到上一级栈结构，依次类推，直到获取到最初的栈结构。
- 需要考虑获取上一级栈帧的终止条件。RISC-V 的用户栈空间占一个页面，可以通过 `PGROUNDOWN()` 和 `PGROUNDUP()` 计算得到一个地址所在页面的最高和最低地址。初始从寄存器 `s0` 读取到的栈帧 `fp` 是在用户栈空间中的地址，由此可以得到用户栈的页面最高和最低地址作为循环的终止条件。

3. 添加 `backtrace()` 函数原型到 `kernel/defs.h`。

```
79 // printf.c
80 void printf(char*, ...);
81 void panic(char*) __attribute__((noreturn));
82 void printinit(void);
83 void backtrace(void);
84
```

4. 在 kernel/sysproc.c 的 sys_sleep() 函数中添加对 backtrace() 的调用。

```
55     uint64
56     sys_sleep(void)
57     {
58         int n;
59         uint ticks0;
60
61         if(argint(0, &n) < 0)
62             return -1;
63         acquire(&tickslock);
64         ticks0 = ticks;
65         while(ticks - ticks0 < n){
66             if(myproc()->killed){
67                 release(&tickslock);
68                 return -1;
69             }
70             sleep(&ticks, &tickslock);
71         }
72         backtrace();
73         release(&tickslock);
74         return 0;
75     }
76 }
```

5. 在 kernel/printf.c 的 panic() 函数中添加对 backtrace() 的调用。

```
117     void
118     panic(char *s)
119     {
120         pr.locking = 0;
121         printf("panic: ");
122         printf(s);
123         printf("\n");
124         backtrace();
125         panicked = 1; // freeze uart output from other CPUs
126         for(;;)
127             ;
128     }
129 }
```

6. 运行与测试

实验结果

1. 在 xv6 中运行 bttest, 输出 3 个栈帧的返回地址; 退出 xv6 后运行 addr2line -e kernel/kernel 将 bttest 的输出作为输入, 输出对应的调用栈函数, 如下图所示。

```
5  no graphical drive /dev/sr0, cd-rom, format raw, to x0  device virtio-blk de  
vice,drive=x0,bus=virtio-mmio-bus.0  
  
xv6 kernel is booting  
  
hart 1 starting  
hart 2 starting  
init: starting sh  
$ bttest  
backtrace:  
0x000000008000217e  
0x0000000080001ff0  
0x0000000080001ca0  
S |
```

2. 根据输出的源码行号找对应的源码, 发现就是 `backtrace()` 函数的所有调用栈的返回地址(函数调用完后的下一代码).

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ addr2line -e kernel/kernel  
0x000000008000217e  
/home/yukiip/xv6-labs-2021/kernel/sysproc.c:73  
0x0000000080001ff0  
/home/yukiip/xv6-labs-2021/kernel/syscall.c:144  
0x0000000080001ca0  
/home/yukiip/xv6-labs-2021/kernel/trap.c:76
```

```
56     uint64
57     sys_sleep(void)
58     {
59         int n;
60         uint ticks0;
61
62         if(argint(0, &n) < 0)
63             return -1;
64         acquire(&tickslock);
65         ticks0 = ticks;
66         while(ticks - ticks0 < n){
67             if(myproc()>killed){
68                 release(&tickslock);
69                 return -1;
70             }
71             sleep(&ticks, &tickslock);
72         }
73         backtrace();
74         release(&tickslock);
75         return 0;
76     }
```

```

136 void
137 syscall(void)
138 {
139     int num;
140     struct proc *p = myproc();
141
142     num = p->trapframe->a7;
143     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
144         p->trapframe->a0 = syscalls[num]();
145     } else {
146         printf("%d %s: unknown sys call %d\n",
147                p->pid, p->name, num);
148         p->trapframe->a0 = -1;
149     }
150 }
151
152     intr_on();
153
154     syscall();
155 } else if((which_dev = devintr()) != 0){
156     // ok
157 } else {
158     printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid)
159     printf("          sepc=%p stval=%p\n", r_sepc(), r_stval());
160     p->killed = 1;
161 }
162
163 if(p->killed)
164     exit(-1);
165
166

```

分析讨论

1. 实验中遇到的问题及解决

- 获取上一级栈帧的终止条件：在实现过程中，需要考虑如何正确识别和处理栈帧的终止条件。通过使用 PGROUNDDOWN() 和 PGROUNDUP() 函数，计算栈帧所在页面的边界地址，确保循环在合理的边界内运行，避免越界访问。
- 栈帧指针的有效性检查：在遍历栈帧时，需要确保栈帧指针 fp 的有效性。通过检查 fp 是否在用户栈空间页面的范围内，确保访问的地址是合法的，避免出现异常访问和崩溃。
- 多级调用栈的处理：在输出栈帧信息时，需要正确处理多级调用栈。通过依次访问每一级栈帧并输出相应的返回地址，保证调用栈信息的完整性和准确性。

2. 实验心得：

- 在本次实验中，通过编写 backtrace() 函数并成功实现栈帧信息的输出，我深刻体会到了对底层栈结构的理解和对系统调用栈的掌握的重要性。特别是在解决获取上一级栈帧的终止条件和栈帧指针有效性检查的问题时，进一步加深了我对 RISC-V 架构和操作系统内部机制的认识。
- 此外，在调试和验证过程中，通过使用 addr2line 工具将返回地址转化为源码行号，极大地方便了对调用栈信息的核对和分析。这不仅提高了代码的可靠性，还增强了我在系统编程和调试方面的能力，为今后处理类似问题积累了宝贵的经验。

Alarm

实验目的

本次实验将向 xv6 内核添加一个新的功能，即周期性地为进程设置定时提醒。这个功能类似于用户级的中断/异常处理程序，能够让进程在消耗一定的 CPU 时间后执行指定的函数，然后恢复执行。通过实现这个功能，我们可以为计算密集型进程限制 CPU 时间，或者为需要周期性执行某些操作的进程提供支持。

实验步骤

1. 在 `user/user.h` 中添加两个系统调用的函数原型:

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

```
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int sigalarm(int ticks, void (*handler)());
27 int sigreturn(void);
28
```

2. 在 `user/usys.pl` 脚本中添加两个系统调用的相应 `entry`, 在 `kernel/syscall.h` 和 `kernel/syscall.c` 添加相应声明.

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

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

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

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

3. 在 `kernel/proc.h` 中的 `struct proc` 结构体中添加记录时间间隔, 调用函数地址, 以及经过时钟数的字段

```
int interval;
uint64 handler;
int passedticks;
```

4. 编写 `sys_sigalarm()` 函数, 将 `interval` 和 `handler` 的值存到当前进程的 `struct proc` 结构体的相应字段中.

在这里在指导书的基础上又做了两点优化: 一方面限定了 `interval` 的值需要非负, 根据定义 `interval` 表示每次调用 `handler` 函数的周期, `0` 特指取消调用, 而负数在这里是没有意义的, 因

此将其视为非法参数; 另一方面同时重置了过去的时钟数 `p->passedticks`, 此处考虑到可能中途会更新 `sigalarm()` 的调用参数, 这样之前记录的过去时钟数便失效了, 应该重新计数。

5. `kernel/proc.c` 的 `allocproc()` 函数负责分配并初始化进程, 此处对上述 `struct proc` 新增的三个字段进行初始化赋值。

```
...
memset(&p->context, 0, sizeof(p->context));
p->context.ra = (uint64)forkret;
p->context.sp = p->kstack + PGSIZE;
p->interval = 0;
p->handler = 0;
p->passedticks = 0;
```

6.
 - 每当发生时钟中断时, `kernel/trap.c` 中的 `usertrap()` 函数会被调用。对于时钟中断, `which_dev` 变量的值为 `2`, 因此可以单独处理时钟中断。根据指导书的要求, 由于 `handler` 函数的地址可能为 `0`, 主要通过 `interval == 0` 来判断是否终止定时调用函数。
 - 每次发生时钟中断时, 将 `passedticks` 增加 `1`。当 `passedticks` 达到 `interval` 时, 调用 `handler()` 函数, 并将 `passedticks` 置零, 以便下次调用定时函数。
 - 关键在于如何调用定时函数 `handler()`。需要注意的是, 在 `usertrap()` 中, 页表已经切换为内核页表 (切换操作在 `uservec` 函数中完成), 而 `handler` 是用户空间的函数虚拟地址, 不能直接调用。实际上, 这里并没有直接调用 `handler` 函数, 而是将 `p->trapframe->epc` 设置为 `p->handler`, 这样在返回到用户空间时, 程序计数器指向 `handler` 定时函数的地址, 从而实现了定时函数的执行。

7. 修改 `Makefile` 文件中的 `UPROGS` 部分, 添加对 `alarmtest.c` 的编译。

8. 发现 `test0` 可以通过, 但 `test1/test2(): resume interrupted code`, 寻找原因。

9. 修改 `struct proc` 结构体, 添加 `trapframe` 的副本字段:

```
// Per-process state
struct proc {
// ...
char name[16];           // Process name (debugging)
int interval;            // alarm interval
uint64 handler;          // pointer to the handler function
int passedticks;          // ticks have passed since the last call
struct trapframe* trapframecopy; // the copy of trapframe
};
```

10. 在 `kernel/trap.c` 的 `usertrap()` 中覆盖 `p->trapframe->epc` 前做 `trapframe` 的副本。

```
void
usertrap(void)
{
// ...
if(which_dev == 2){ // timer interrupt
    // increase the passed ticks
    if(p->interval != 0 && ++p->passedticks == p->interval){
        // 使用 trapframe 后的一部分内存, trapframe大小为288B, 因此只要在trapframe地址后288以上地址都可, 此处512只是为了取整数幂
        p->trapframecopy = p->trapframe + 512;
    }
}
```

```

        memmove(p->trapframecopy, p->trapframe, sizeof(struct trapframe));    //
        copy trapframe
        p->trapframe->epc = p->handler;    // execute handler() when return to
user space
    }
}

// ...
}

```

11. 在 `sys_sigreturn()` 中将副本恢复到原 `trapframe`。此处在拷贝副本前额外做了一个地址判断，是防止用户程序在未调用 `sigalarm()` 便使用了该系统调用，那么此时没有副本即 `trapframecopy` 是无效的，应避免错误拷贝。在拷贝后将 `trapframecopy` 置零，表示当前没有副本。

```

uint64 sys_sigreturn(void) {
    struct proc* p = myproc();
    // trapframecopy must have the copy of trapframe
    if(p->trapframecopy != p->trapframe + 512) {
        return -1;
    }
    memmove(p->trapframe, p->trapframecopy, sizeof(struct trapframe));    //
    restore the trapframe
    p->passedticks = 0;      // prevent re-entrant
    p->trapframecopy = 0;    // 置零
    return p->trapframe->a0;    // 返回a0，避免被返回值覆盖
}

```

12. 为了保证 `trapframecopy` 的一致性，在初始进程 `kernel/proc.c` 的 `allocproc()` 中，初始化 `p->trapframecopy` 为 0，表明初始时无副本。
13. 根据指导书的要求，定时函数 `handler` 需要防止重入，即在其尚未返回时不能触发下一次调用。为此，需要将 `p->passedticks = 0;` 从原本的 `usertrap()` 移至 `sys_sigreturn()` 中。因为在 `usertrap()` 中重置 `passedticks` 后，后续的时钟中断会继续递增 `passedticks`，可能再次满足调用 `handler` 的条件。而将重置操作移至 `sys_sigreturn()` 之后，即在函数最后返回前才清零，按照系统调用的正确使用方法，`sigreturn()` 的结束应该标志着 `handler()` 的结束。这样，在 `handler()` 还未结束时，`passedticks` 会继续递增，从而不会满足调用 `handler` 的条件，自然就可以避免重入。上述代码已经过修改。
14. 编译并进行测试。

实验结果

在 xv6 中执行 `alarmtest` 和 `usertests` 均通过。

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

分析讨论

1. 实验中遇到的问题及解决：

- `trapframe` 副本的创建与恢复：我们通过在时钟中断中直接修改进程的 `trapframe` 来实现定时提醒功能，但在运行测试时发现 `test1/test2()` 无法通过。这是因为在执行定时函数 `handler` 后，进程无法正确恢复到中断前的状态，导致程序继续执行时出现错误。为了解决这个问题，我们需要在进入定时函数前保存进程的 `trapframe` 状态，在定时函数执行完成后恢复该状态。为此，我们在进程结构体 `struct proc` 中添加了一个新的字段 `trapframecopy` 来存储 `trapframe` 的副本。

- **实验心得：**这次实验使我深入理解了操作系统的信号处理机制，通过实现定时提醒功能，我学会了如何在内核中添加系统调用、管理进程状态和处理中断，提升了系统编程和调试能力。此外，这次实验也涉及到用户态和管理态的转换，我再次巩固了如何设置声明和入口使得二者连接。实验中遇到的挑战，如正确保存和恢复 `trapframe` 以及防止函数重入，使我认识到细致的状态管理和全面的测试对于系统开发的重要性。通过测试程序，我明白在修改内核操作时，应确保不影响系统稳定性，即在实现定时中断处理功能时，要确保不会影响系统的正常运行，确保中断处理程序能够及时返回，避免影响其他中断和系统调度。进行充分的测试，我们才能确保定时中断处理不会导致系统崩溃或异常。这次实践不仅增强了我的理论知识，也提高了独立解决问题的能力。

通过截图

```
"/ /, /$> kernel/kernel.sym
make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (12.6s)
== Test running alarmtest ==
$ make qemu-gdb
(5.7s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (404.0s)
(Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 85/85
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

Lab5 Copy-on-write

Implement copy-on write

实验目的

实验的主要目的是在 xv6 操作系统中实现写时复制 (Copy-on-Write, COW) 的 `fork` 功能。传统的 `fork()` 系统调用会复制父进程的整个用户空间内存到子进程，而 `cow fork()` 则通过延迟分配和复制物理内存页面，只在需要时才进行复制，从而提高性能和节省资源。通过这个实验，你将了解如何使用写时复制技术优化进程的 `fork` 操作。

实验步骤

1. 修改 `uvncpy()` 将父进程的物理页映射到子进程，而不是分配新页。原来 `uvncpy()` 是将虚拟地址 `[0, sz]` 这个区间对应的物理内存的数据拷贝到新的物理内存中。现在不需要在这里申请新的物理内存，只需要将页表与父进程的物理内存进行映射就行，同时在子进程和父进程的 `PTE` 中清除 `PTE_W` 标志，设置 `RSW` 标志位。

```
int
uvncpy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;
    // char *mem;

    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walk(old, i, 0)) == 0)
```

```

    panic("uvmcopy: pte should exist");
    if(*pte & PTE_V) == 0)
        panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    // 清除PTE_W标志
    flags &= (~PTE_W);
    // 添加PTE_RSW标志
    flags |= PTE_RSW;
    // 清除父进程PTE的PTE_W标志
    *pte &= (~PTE_W);
    // 父进程PTE添加PTE_RSW标志
    *pte |= PTE_RSW;
    // 将父进程的物理内存映射到子进程的虚拟内存
    if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
        // kfree((void*)pa);
        goto err;
    }
    // 映射成功，父进程的物理内存引用计数增加
    mem_count_up(pa);
}
return 0;

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

```

2. 在 `riscv.h` 中进行定义 `PTE_RSW` 标志位

```

#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_RSW (1L << 8) // 用这个标志位来表示cow的页面错误

```

3. 题目提示我们，可以使用页的物理地址除以4096对数组进行索引，并为数组提供

`PHYSTOP/PGSIZE` 个元素。为什么是 `PHYSTOP/PGSIZE` 呢？笔者认为这样做有利于索引管理。由于xv6中的内存是进行页式管理的，这对于虚拟内存和物理内存都一样。使用数组时，只需将物理地址除以 `PGSIZE` 即可得到数组下标，从而调整该物理地址（内存）的引用计数。

在 `kalloc.c` 文件中，参考 `kmem` 结构体对内存引用结构体进行了定义，同时定义了一些可能会用到的函数，例如增加引用计数、减少引用计数（若减少到0则函数返回真）、将引用计数设置为1，以及获取引用计数值。需要注意的是锁的使用，使用锁后需要及时释放锁。笔者认为封装这些函数不仅方便调用，还可以避免在编写代码时忘记释放锁的情况。

`kalloc.c` 文件中：

```

// 内存引用计数的结构体
struct
{
    struct spinlock lock; // 若有多个进行同时对数组进行操作，需要上锁
    int mem_count[PHYSTOP/PGSIZE];
}mem_ref_struct;

int get_mem_count(uint64 pa){

```

```

int count;
acquire(&mem_ref_struct.lock);
count = mem_ref_struct.mem_count[(uint64)pa / PGSIZE];
release(&mem_ref_struct.lock);
return count;
}

void mem_count_up(uint64 pa){
    acquire(&mem_ref_struct.lock);
    ++ mem_ref_struct.mem_count[(uint64)pa / PGSIZE];
    release(&mem_ref_struct.lock);
}

int mem_count_down(uint64 pa){
    int flag = 0;
    acquire(&mem_ref_struct.lock);
    if((-- mem_ref_struct.mem_count[(uint64)pa / PGSIZE]) == 0){
        flag = 1;
    }
    release(&mem_ref_struct.lock);
    return flag;
}

void mem_count_set_one(uint64 pa){
    acquire(&mem_ref_struct.lock);
    mem_ref_struct.mem_count[(uint64)pa / PGSIZE] = 1;
    release(&mem_ref_struct.lock);
}

```

4. 修改 `usertrap()` 以识别页面错误。在 `usertrap` 中，首先需要确定发生错误的虚拟地址是否来自写时复制 (COW)。如果是，则需要根据内存引用计数来决定是否需要申请新的物理内存。如果引用计数不为1，可能有多个进程引用了同一段物理内存，此时需要申请新的物理内存，进行内存拷贝并更新页表映射等操作。如果引用计数为1，则可能是父进程产生了页面错误，因为内存只剩一个引用，此时需要恢复物理内存的写权限，并清除RSW标志位。

需要注意以下几点：

- 在这个过程中，如果出现了任何失败，需要立即设置 `p->killed` 为 1，然后跳转到 end 处，退出并杀死进程。
- 在申请新的物理内存并进行映射之前，需要使用 `uvmunmap` 函数将虚拟地址与旧的物理内存进行解绑。

上面代码中使用了 `pte = cow_walk(p->pagetable, PGROUNDDOWN(va))` 函数，是一个自定义的函数，在 `vm.c` 文件中进行定义。仿照 `walkaddr` 函数写一个函数用来检验虚拟地址是否是来自 `copy on write`。注意其中添加了一个对 `PTE_RSW` 位的检查，这是关键。

5. 内存引用计数相关的步骤在第一步已经做了相关的定义，接下来是一些使用的地方。

- 首先要对内存引用锁进行初始化，在 `kalloc.c` 文件中修改 `kinit()` 函数，初始化自旋锁。

```

void
kinit()
{
    initlock(&kmem.lock, "kmem");
    // 初始化mem_ref_struct的锁
    initlock(&mem_ref_struct.lock, "mem_ref");
    freerange(end, (void*)PHYSTOP);
}

```

在申请内存 `kalloc()` 函数，释放内存 `kfree()` 函数中，进行如下修改：

- `freerange` 函数中调用了 `kfree`，这个函数在系统内存初始化的时候调用，而且是在没有 `kalloc` 的前提下调用的，因为我们修改了 `kfree` 函数的逻辑，所以 `freerange` 函数中要先将内存引用计数置1。

```

void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE){
        // 系统初始化时会将内存引用减1，所以这里先设为1
        mem_count_set_one((uint64)p);
        kfree(p);
    }
}

```

6. 下面对 `copyout` 函数进行修改。这里就是将内核物理内存copy到用户物理内存前需要检查一下用户物理内存 (`dst`) 是不是COW页面，如果时则需要申请新的用户物理内存。这里只需要改动 `copyout` 而不需要改 `copyin` 是因为前者是内核拷贝到用户，是会对一个用户页产生写的操作，而后者是用户拷贝到内核，只是去读这个用户页的内容，COW页允许读。

7. 最后，一些函数需要在 `defs.h` 中进行声明。

```

int          get_mem_count(uint64 pa);
void         mem_count_up(uint64 pa);
int          mem_count_down(uint64 pa);
void         mem_count_set_one(uint64 pa);
pte_t*       cow_walk(pagetable_t , uint64 );

```

8. 编译并进行测试。

实验结果

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

分析讨论

1. 优点与局限性:

◦ 优点:

- 内存节省: 通过写时复制, 父子进程在初始阶段共享相同的物理页, 只有在实际需要写入时才会分配新的页, 从而大大节省了内存。
- 性能提升: 减少了 fork 操作时的内存复制, 降低了系统开销, 提高了进程创建的速度。
- 简化内存管理: 通过引用计数的管理, 可以更高效地管理物理内存的分配和释放。

◦ 局限性:

- 实现复杂性增加: 引入写时复制和引用计数机制增加了系统实现的复杂性, 需要处理更多的边界情况和错误处理。
- 额外的开销: 虽然写时复制节省了内存, 但在实际写操作发生时, 仍然需要进行内存分配和复制, 这在某些情况下可能会带来额外的开销。
- 测试与调试困难: 实现写时复制需要对内存管理和页表进行深入的修改, 这使得测试和调试变得更加困难。

2. 实验心得: 通过这个实验, 我们深入了解了写时复制技术在操作系统中的实现原理及其优势。我们通过修改 xv6 操作系统, 成功地实现了一个简单但有效的 COW fork 功能。虽然在实现过程中遇到了一些挑战, 但最终的实现证明了写时复制在提高系统性能和节省资源方面的显著优势。通过进一步的优化和改进, 可以使这一技术在实际应用中发挥更大的作用。

通过截图

```
/ , , , ./s - kernel/kernel.syn
make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
== Test running cowtest ==
$ make qemu-gdb
(30.3s)
== Test simple ==
simple: OK
== Test three ==
three: OK
== Test file ==
file: OK
== Test usertests ==
$ make qemu-gdb
(376.8s)
    (Old xv6.out.usertests failure log removed)
== 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
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

Lab6 Multithreading

Uthread: switching between threads

实验目的

设计并实现一个用户级线程系统的上下文切换机制。补充完成一个用户级线程的创建和切换上下文的代码。需要创建线程、保存/恢复寄存器以在线程之间切换，并且确保解决方案通过测试。

实验步骤

1. user/uthread.c中，新增头文件引用。

```
#include "kernel/riscv.h"
#include "kernel/spinlock.h"
#include "kernel/param.h"
#include "kernel/proc.h"
```

2. `struct thread` 增加成员 `struct context`，用于线程切换时保存/恢复寄存器信息。此处 `struct context` 即 `kernel/proc.h` 中定义的 `struct context`。

```
struct thread {
    char      stack[STACK_SIZE]; /* the thread's stack */
    int       state;           /* FREE, RUNNING, RUNNABLE */
    struct context threadContext; // 借鉴proc的context
};
```

3. 修改 `thread_switch` 函数定义。

```
extern void thread_switch(struct context*, struct context*);
```

4. 在 `thread_create()` 函数中添加代码，该函数的主要任务是进行线程的初始化操作。具体过程如下：

- 首先，在线程数组中找到一个状态为 FREE（未初始化）的线程。
- 设置该线程的状态为 RUNNABLE，并进行其他初始化操作。

需要特别注意的是，传递给 `thread_create()` 函数的参数 `func` 必须记录下来，以便在线程运行时能够执行该函数。此外，线程有独立的栈结构，函数运行时需要在该线程的栈上进行，因此需要初始化线程的栈指针。

在线程调度切换时，必须保存和恢复寄存器状态。这里分别对应的是 `ra`（返回地址寄存器）和 `sp`（栈指针寄存器）。在线程初始化时设置这些寄存器的值，确保在后续的调度切换过程中能正确保持线程状态。

5. 在 `thread_schedule()` 函数中添加代码。该函数负责用户多线程间的调度，通过函数主动调用进行线程切换。其主要任务是从当前线程在线程数组中的位置开始，寻找一个状态为 `RUNNABLE` 的线程进行运行。这与 `kernel/proc.c` 中的 `scheduler()` 函数非常相似。

找到合适的线程后，需要进行线程切换，调用 `thread_switch()` 函数。根据 `user/thread.c` 中的外部声明以及指导书的要求，可以推断出该函数定义在 `user/uthread_switch.s` 中，并用汇编代码实现。其功能类似于 `kernel/swtch.s` 中的 `swtch()` 函数，负责在线程切换时保存和恢复寄存器状态。

6. 最后，在 `user/uthread_switch.s` 中添加 `thread_switch` 的代码。正如上文所述，该函数的功能与 `kernel/swtch.s` 中的 `swtch` 函数一致。由于 `struct ctx` 与内核中的 `struct context` 结构体的成员相同，因此该函数可以直接复用 `kernel/swtch.s` 中的 `swtch` 代码。

7. 测试并运行。

实验结果

在 xv6 中运行 `uthread`：

```
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
$
```

分析讨论

1. 实验中遇到的问题及解决：

1. 线程调度不正确：在 `thread_schedule()` 函数中，无法正确找到 `RUNNABLE` 状态的线程，导致线程无法正确调度。因此，在 `thread_schedule()` 函数中添加调试信息，验证线程数组的状态变化和调度逻辑的正确性。确保从当前线程的位置开始，正确地遍历线程数组，并找到下一个 `RUNNABLE` 状态的线程。
2. 上下文切换失败：在 `thread_switch()` 函数中，无法正确保存和恢复寄存器状态，导致线程切换失败或程序崩溃。因此，检查 `thread_switch` 的汇编代码，确保寄存器的保存和恢复操作正确无误。可以对照 `kernel/swtch.s` 中的 `swtch` 函数，逐行检查代码的正确性。此外，可以通过打印寄存器值或使用调试器，验证寄存器的正确保存和恢复。
3. 线程调度不正确：在 `thread_schedule()` 函数中，无法正确找到 `RUNNABLE` 状态的线程，导致线程无法正确调度。因此，在 `thread_schedule()` 函数中添加调试信息，验证线程数组的状态变化和调度逻辑的正确性。确保从当前线程的位置开始，正确地遍历线程数组，并找到下一个 `RUNNABLE` 状态的线程。

2. 思考题：

`thread_switch` needs to save/restore only the callee-save registers. Why?

这里仅需保存被调用者保存(`callee-save`)寄存器的原因和`xv6`中内核线程切换时仅保留`callee-save`寄存器的原因是相同的。由于`thread_switch()`一定由其所在的C语言函数调用，因此函数的调用规则是满足`xv6`的函数调用规则的，对于其它`caller-save`寄存器都会被保存在线程的堆栈上，在切换后的线程上下文恢复时可以直接从切换后线程的堆栈上恢复`caller-save`寄存器的值。由于`callee-save`寄存器是由被调用函数即`thread_switch()`进行保存的，在函数返回时已经丢失，因此需要额外保存这些寄存器的内容。

3. 实验心得：通过本次实验，我们成功实现了用户级线程系统的上下文切换机制，并通过了相关测试。实验过程中，我们深入了解了线程的创建、上下文保存与恢复等关键技术，并通过实际编码加深了对用户级线程系统的理解。通过进一步的优化和改进，可以使用户级线程系统在实际应用中发挥更大的作用。

Using threads

实验目的

本实验旨在通过使用线程和锁实现并行编程，以及在多线程环境下处理哈希表。学习如何使用线程库创建和管理线程，以及如何通过加锁来实现一个线程安全的哈希表，使用锁来保护共享资源，以确保多线程环境下的正确性和性能。

实验步骤

1. 使用如下命令构建`ph`程序，该程序包含一个线程不安全的哈希表。
2. 运行`./ph 1`即使用单线程运行该哈希表，输出如下，其0个键丢失：

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ ./ph 1
100000 puts, 16.829 seconds, 5942 puts/second
0: 0 keys missing
100000 gets, 14.821 seconds, 6747 gets/second
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

3. 运行`./ph 2`即使用两个线程运行该哈希表，输出如下，可以看到其`put`速度近乎先前2倍，但是有16423个键丢失，也说明了该哈希表非线程安全。

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ ./ph 2
100000 puts, 5.634 seconds, 17751 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 9.763 seconds, 20486 gets/second
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

4. 定义互斥锁数组。

根据指导书可知，此处主要通过加互斥锁来解决线程不安全的问题。此处没有选择使用一个互斥锁，这样会导致访问整个哈希表都是串行的。而考虑到对该哈希表，实际上只有对同一`bucket`操作时才可能造成数据的丢失，不同`bucket`之间是互不影响的，因此此处是构建了一个互斥锁数组，每个`bucket`对应一个互斥锁。在`ph.c`中增加：

```
pthread_mutex_t locks[NBUCKET]; // Lab7-2
```

5. 在 `main()` 函数中对所有互斥锁进行初始化。

```
124     for(int i = 0; i < NBUCKET; ++i) {  
125         pthread_mutex_init(&locks[i], NULL);  
126     }  
127 }
```

6. 在 `put()` 中加锁。

由于线程安全问题是由于对 `bucket` 中的链表操作时产生的，因此要在对链表操作的前后加锁。

但实际上，对于加锁的临界区可以缩小至 `insert()` 函数。原因是 `insert()` 函数采取头插法插入 `entry`，在函数的最后才使用 `*p=e` 修改 `bucket` 链表头 `table[i]` 的值，也就是说，在前面操作的同时，并不会对 `bucket` 链表进行修改，因此可以缩小临界区的方法。

不需要在 `get()` 中加锁。`get()` 函数主要是遍历 `bucket` 链表找寻对应的 `entry`，并不会对 `bucket` 链表进行修改，实际上只是读操作，因此无需加锁。

7. 增大 `NBUCKET`。增大 `NBUCKET` 即增加哈希表的 `bucket` 数，从而一定程度上会减少两个同时运行的线程 `put()` 时对同一个 `bucket` 进行操作的概率，自然也就减少了锁的争用，能够一定程度上挺高并发性能。此处选择 `NBUCKET=7`。

```
#define NBUCKET 7
```

8. 编译运行并测试。

实验结果

1. `./ph 2` 测试。

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ ./ph 2  
100000 puts, 5.634 seconds, 17751 puts/second  
1: 0 keys missing  
0: 0 keys missing  
200000 gets, 9.763 seconds, 20486 gets/second  
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ █
```

分析讨论

1. 实验中遇到的问题及解决：

1. 线程安全问题：最初的哈希表在多线程环境下会导致数据丢失。通过为每个 `bucket` 添加互斥锁，我们成功地解决了这个问题。
2. 性能问题：在加锁过程中，需要确保锁的粒度尽可能小，以提高并发性能。通过将加锁范围缩小至 `insert()` 函数，我们减少了锁的争用，提升了程序性能。
3. 锁的争用：当多个线程同时访问同一个 `bucket` 时，会导致锁的争用，从而影响性能。通过增大 `NBUCKET` 的值，减少了锁的争用，提高了并发性能。

2. 性能优化：

1. 引入读写锁：在当前实现中，所有访问 `bucket` 的操作都使用了互斥锁。为了进一步提高性能，可以引入读写锁，使得多个线程可以同时进行读操作，而写操作仍然需要独占锁。
2. 动态调整 `bucket` 数量：根据运行时的负载情况，动态调整 `bucket` 的数量，以优化哈希表的性能。
3. 性能监控与调优：通过性能监控工具，分析锁的争用情况和系统瓶颈，进一步优化锁的使用和哈希表的结构。

3. 思考题：

Why are there missing keys with 2 threads, but not with 1 thread? Identify a sequence of events with 2 threads that can lead to a key being missing. Submit your sequence with a short explanation in answers-thread.txt

这里的哈希表就是“数组(bucket)+链表”的经典实现方法。通过取余确定 bucket, put() 是使用前插法插入键值对, get() 遍历 bucket 下的链表找到对应 key 的 entry。而这个实现没有涉及任何锁机制或者 CAS 等线程安全机制, 因此线程不安全, 多线程插入时会出现数据丢失。

该哈希表的线程安全问题是: 多个线程同时调用 put() 对同一个 bucket 进行数据插入时, 可能会使得先插入的 entry 丢失。具体来讲, 假设有 A 和 B 两个线程同时 put(), 而恰好 put() 的参数 key 对应到了哈希表的同一 bucket。同时假设 A 和 B 都运行到 put() 函数的 insert() 处, 还未进入该函数内部, 这就会导致两个线程 insert() 的后两个参数是相同的, 都是当前 bucket 的链表头, 如若线程 A 调用 insert() 插入完 entry 后, 切换到线程 B 再调用 insert() 插入 entry, 则会导致线程 A 刚刚插入的 entry 丢失。

4. 实验心得: 通过本次实验, 我深入理解了多线程环境下的数据一致性问题, 并通过使用互斥锁成功地实现了一个线程安全的哈希表。实验中遇到的问题和解决方案使我进一步掌握了并行编程的技巧和方法。通过不断优化和调试, 我们不仅解决了数据丢失问题, 还显著提高了程序的并发性能。这些经验和教训将为我未来的并行编程实践提供宝贵的参考。

Barrier

实验目的

本实验的目的是实现一个线程屏障 (barrier), 即每个线程在到达屏障时等待, 直到所有线程都到达屏障后才能继续运行, 从而加深对多线程编程中同步和互斥机制的理解。在多线程应用中, 线程屏障用于确保多个线程在达到某个点后都等待, 直到所有其他线程也到达该点。通过使用 pthread 条件变量, 我们将学习如何实现线程屏障, 解决竞争条件和同步问题。

实验步骤

1. 运行如下命令构建 barrier 程序, 该程序要求多线程同时执行到同一位置后再继续运行, 即多线程同步问题。

```
$ make barrier
```

2. 运行 ./barrier 2 即使用两个线程运行该程序, 即最初版本不满足该性质, 会致使运行失败。

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ ./barrier 2
```

3. 根据上述思路, 在 barrier() 函数中添加以下代码。需要注意的是, 在调用 pthread_cond_broadcast() 唤醒其他线程之前, 必须先设置变量 bstate.round 和 bstate.nthread, 否则在其他线程进入下一轮循环时, 这两个字段的值可能尚未更新。

```
static void
barrier()
{
    pthread_mutex_lock(&bstate.barrier_mutex);
    // judge whether all threads reach the barrier
    if(++bstate.nthread != nthread) { // not all threads reach
        pthread_cond_wait(&bstate.barrier_cond,&bstate.barrier_mutex); // wait other threads
    } else { // all threads reach
```

```

        bstate.nthread = 0; // reset nthread
        ++bstate.round; // increase round
        pthread_cond_broadcast(&bstate.barrier_cond); // wake up all
        sleeping threads
    }
    pthread_mutex_unlock(&bstate.barrier_mutex);
}

```

4. 编译运行并测试。

实验结果

```

yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ ./barrier 2
OK; passed
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ ./barrier 5
OK; passed
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ ./barrier 10
OK; passed

```

分析讨论

通过本次实验，我深入理解了线程屏障机制的实现原理和具体实现过程。通过引入 `pthread` 条件变量和互斥锁，我们成功实现了一个线程安全的屏障机制，确保多线程环境下的同步。实验中遇到的问题和解决方案，使我进一步掌握了多线程编程的技巧和方法。这些经验将为我未来的并行编程实践提供宝贵的参考。

通过截图

```

make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
== Test uthread ==
$ make qemu-gdb
uthread: OK (14.3s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/yukiip/xv6-labs-2021”
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
ph_safe: OK (20.5s)
== Test ph_fast == make[1]: 进入目录“/home/yukiip/xv6-labs-2021”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
ph_fast: OK (43.6s)
== Test barrier == make[1]: 进入目录“/home/yukiip/xv6-labs-2021”
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
barrier: OK (18.3s)
== Test time ==
time: OK
Score: 60/60
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ █

```

Lab7 Networking

Networking

实验目的

通过为网络接口卡 (NIC) 编写一个 xv6 设备驱动程序，来理解网络通信的核心机制。具体步骤包括创建以太网驱动程序、处理 ARP 请求和响应、实现 IP 数据包的发送和接收、处理 ICMP Echo 请求和响应、实现 UDP 数据报的发送和接收，以及编写一个简单的 DHCP 客户端从 DHCP 服务器获取 IP 地址。通过这个实验，可以深入理解网络协议栈的工作原理。

实验步骤

- 为了实现 `kernel/e1000.c` 中的 `e1000_transmit()` 函数以发送以太网数据帧到网卡，我们需要处理网卡的发送数据队列 `tx_ring`。每个元素是一个发送描述符，`addr` 字段指向以太网帧数据的缓冲区地址，对应 `tx_mbufs`。以下是 `e1000_transmit()` 的实现步骤：
 - 读取发送尾指针对应的寄存器 `regs[E1000_TDT]`，获取软件可以写入的位置，即发送队列的索引 `tail`。
 - 获取对应的发送描述符 `desc`。
 - 检查尾指针指向的描述符的状态 `status` 是否包含 `E1000_TXD_STAT_DD` 标志位。如果未设置该标志位，说明数据尚未传输完毕，返回失败。
 - 释放描述符对应的缓冲区（如果存在）。
 - 更新描述符的 `addr` 字段为数据帧缓冲区的头部 `m->head`，`length` 字段为数据帧的长度 `m->len`。
 - 更新描述符的 `cmd` 字段，设置 `E1000_TXD_CMD_EOP` 和 `E1000_TXD_CMD_RS` 标志位。
 - 将数据帧缓冲区 `m` 记录到 `tx_mbufs` 中，以便后续释放。
 - 使用 `__sync_synchronize()` 设置内存屏障，确保描述符更新完成后再更新尾指针。
 - 更新发送尾指针 `regs[E1000_TDT]`。

10. 释放锁，返回成功。

```
74
95 int
96 e1000_transmit(struct mbuf *m)
97 {
98     //
99     // Your code here.
100    //
101    // the mbuf contains an ethernet frame; program it into
102    // the TX descriptor ring so that the e1000 sends it. Stash
103    // a pointer so that it can be freed after sending.
104    //
105    uint32 tail;
106    struct tx_desc *desc;
107
108    acquire(&e1000_lock);
109    tail = regs[E1000_TDT];
110    desc = &tx_ring[tail];
111    // check if the ring is overflowing
112    if ((desc->status & E1000_RXD_STAT_DD) == 0) {
113        release(&e1000_lock);
114        return -1;
115    }
116    // free the last mbuf that was transmitted
117    if (tx_mbufs[tail]) {
118        mbuffree(tx_mbufs[tail]);
119    }
120    // fill in the descriptor
121    desc->addr = (uint64) m->head;
122    desc->length = m->len;
123    desc->cmd = E1000_RXD_CMD_EOP | E1000_RXD_CMD_RS;
124    tx_mbufs[tail] = m;
125
126    // a barrier to prevent reorder of instructions
127    __sync_synchronize();
128
129    regs[E1000_TDT] = (tail + 1) % TX_RING_SIZE;
130    release(&e1000_lock);
131
132    return 0;
133}
134
```

2. 为了实现 `kernel/e1000.c` 中的 `e1000_recv()` 函数以接收数据到内核，需要处理接收队列

`rx_ring` 和其缓冲区 `rx_mbufs`。以下是具体实现步骤：

1. 读取接收尾指针寄存器 `regs[E1000_RDT]` 并加 1 取余，获取软件可读取的位置，即接收且未处理的数据帧的描述符索引 `tail`。
2. 获取对应的接收描述符 `desc`。
3. 检查描述符的状态 `status` 是否包含 `E1000_RXD_STAT_DD` 标志位，确定数据帧已被硬件处理完毕，可以由内核解封装。
4. 将 `rx_mbufs[tail]` 中的数据帧长度记录到描述符的 `length` 字段，并调用 `net_rx()` 将数据传递给网络栈进行解封装。
5. 调用 `mbufalloc()` 分配新的接收缓冲区替换发送给网络栈的缓冲区，并更新描述符的 `addr` 字段指向新的缓冲区。
6. 清空描述符的状态 `status` 字段。
7. 继续检查下一个描述符，直到当前描述符的 `DD` 标志位未被设置，说明数据尚未被硬件处理完毕。
8. 更新接收尾指针 `regs[E1000_RDT]` 指向最后一个已处理的描述符。

9. 不需要加锁，因为接收函数仅在中断处理函数 `e1000_intr()` 中调用，不会出现并发问题。

发送和接收数据结构独立，不会共享资源。以下是 `e1000_recv()` 的实现：

```
135 static void
136 e1000_recv(void)
137 {
138     //
139     // Your code here.
140     //
141     // Check for packets that have arrived from the e1000
142     // Create and deliver an mbuf for each packet (using net_rx()).
143     //
144     int tail = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
145     struct rx_desc *desc = &rx_ring[tail];
146
147     while ((desc->status & E1000_RXD_STAT_DD)) {
148         if(desc->length > MBUF_SIZE) {
149             panic("e1000 len");
150         }
151         // update the length reported in the descriptor.
152         rx_mbufs[tail]->len = desc->length;
153         // deliver the mbuf to the network stack
154         net_rx(rx_mbufs[tail]);
155         // allocate a new mbuf replace the one given to net_rx()
156         rx_mbufs[tail] = mbufalloc(0);
157         if (!rx_mbufs[tail]) {
158             panic("e1000 no mbufs");
159         }
160         desc->addr = (uint64) rx_mbufs[tail]->head;
161         desc->status = 0;
162
163         tail = (tail + 1) % RX_RING_SIZE;
164         desc = &rx_ring[tail];
165     }
166
167     regs[E1000_RDT] = (tail - 1) % RX_RING_SIZE;
168 }
```

3. 编译并进行测试。

实验结果

1. 在 xv6 目录下执行 make server 启动服务端。

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ make server
python3 server.py 26099
listening on localhost port 26099
□
```

2. 然后在另一个终端执行 make qemu 启动 xv6，然后执行 nettests 命令进行测试：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ nettests
nettests running on port 26099
testing ping: OK
testing single-process pings: OK
testing multi-process pings: OK
testing DNS
DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
DNS OK
all tests passed.
$ □
```

3. 然后在终端执行 `tcpdump -XXnr packets.pcap`, 可以查看捕获的报文:

```
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ tcpdump -XXnr packets.pcap
reading from file packets.pcap, link-type EN10MB (Ethernet)
10:42:00.728049 IP 10.0.2.15.2000 > 10.0.2.2.26099: UDP, length 19
    0x0000: ffff ffff ffff 5254 0012 3456 0800 4500 .....RT..4V..E.
    0x0010: 002f 0000 0000 6411 3eae 0a00 020f 0a00 ./....d.>.....
    0x0020: 0202 07d0 65f3 001b 0000 6120 6d65 7373 ....e.....a.mess
    0x0030: 6167 6520 6672 6f6d 2078 7636 21      age.from.xv6!
10:42:00.741100 ARP, Request who-has 10.0.2.15 tell 10.0.2.2, length 28
    0x0000: ffff ffff ffff 5255 0a00 0202 0806 0001 .....RU.....
    0x0010: 0800 0604 0001 5255 0a00 0202 0a00 0202 .....RU.....
    0x0020: 0000 0000 0000 0a00 020f
10:42:00.744621 ARP, Reply 10.0.2.15 is-at 52:54:00:12:34:56, length 28
    0x0000: ffff ffff ffff 5254 0012 3456 0806 0001 .....RT..4V....
    0x0010: 0800 0604 0002 5254 0012 3456 0a00 020f .....RT..4V....
    0x0020: 5255 0a00 0202 0a00 0202
10:42:00.749202 IP 10.0.2.2.26099 > 10.0.2.15.2000: UDP, length 17
    0x0000: 5254 0012 3456 5255 0a00 0202 0800 4500 RT..4VRU.....E.
    0x0010: 002f 0000 0000 6411 3eae 0a00 020f 0a00
    0x0020: 0000 0000 0000 0a00 020f
```

分析讨论

1. 遇到的问题及解决

1. 描述符队列管理: 初始实现时, 不清楚如何正确管理描述符队列中的首尾指针, 导致队列操作混乱。解决: 仔细阅读开发手册, 理解描述符队列的管理机制, 明确发送和接收队列的尾指针分别由软件和硬件维护。通过设置内存屏障, 确保描述符更新的原子性和一致性。
2. 数据帧状态检查: 在处理接收数据时, 不清楚如何判断数据帧是否已被硬件处理完毕。因此, 通过检查描述符状态中的 `E1000_RXD_STAT_DD` 标志位, 判断数据帧是否已被硬件处理完毕, 确保只有已处理的数据帧才传递给网络栈。
3. 缓冲区管理: 在接收数据时, 初始实现未能正确替换已处理的缓冲区, 导致缓冲区复用问题。因此, 在每次处理完接收的数据帧后, 分配新的缓冲区替换已处理的缓冲区, 确保缓冲区的正确管理和复用。
2. 实验心得: 通过本次实验, 我们深入理解了网卡驱动程序的工作机制, 尤其是发送和接收数据的处理流程。通过实现和调试 `e1000_transmit()` 和 `e1000_recv()` 函数, 我们掌握了如何管理网卡的发送和接收队列, 如何检查数据帧状态, 如何进行缓冲区的分配和管理。

通过截图

```
/ /; /*$@ > kernel/kernel.sym
make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
== Test running nettests ==
$ make qemu-gdb
(12.8s)
== 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
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

Lab8 Lock

Memory allocator

实验目的

为了减少多核系统中的锁竞争并提升性能，可以重构内存分配器的设计。具体做法是为每个CPU分配一个独立的自由列表（`free list`），并且每个自由列表都有专属的锁。这样，不同CPU上的内存分配和释放操作可以并行执行，显著减少锁争用。同时，当某个CPU的自由列表耗尽时，它应能够从其他CPU的自由列表中获取部分内存。

实验步骤

- 在 `xv6` 中运行 `kalloctest`，输出如下：

```
lock: bcache: #test-and-set 0 #acquire() 21
lock: bcache_bucket: #test-and-set 0 #acquire() 10
lock: bcache_bucket: #test-and-set 0 #acquire() 22
lock: bcache_bucket: #test-and-set 0 #acquire() 12
lock: bcache_bucket: #test-and-set 0 #acquire() 12
lock: bcache_bucket: #test-and-set 0 #acquire() 20
lock: bcache_bucket: #test-and-set 0 #acquire() 86
lock: bcache_bucket: #test-and-set 0 #acquire() 1084
lock: bcache_bucket: #test-and-set 0 #acquire() 2
--- top 5 contended locks:
lock: proc: #test-and-set 1051677 #acquire() 428629
lock: proc: #test-and-set 798861 #acquire() 428629
lock: proc: #test-and-set 690818 #acquire() 428629
lock: proc: #test-and-set 583790 #acquire() 428629
lock: proc: #test-and-set 539267 #acquire() 428629
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
$ █
```

可以看出，`test1` 测试未通过。根据实验中 `struct spinlock` 的字段和相关代码分析，可以得知 `kmem` 锁的争用情况：`acquire()` 函数被调用了 433016 次，自旋尝试获取锁的次数为 179708 次。此外，`kmem` 锁也是争用最严重的五个锁之一。

- 构造内存页 `kmems` 数组。

根据指导书要求，此处每个 CPU 需要有一个空闲内存页链表以及相应的锁，即将原本在 `kernel/kalloc.c` 中定义的 `kmem` 结构体替换为 `kmems` 数组，数组的大小即为 CPU 的核心数 NCPU。

此处为 `kmems` 结构体额外添加了一个 `lockname` 的字段，用于记录每个锁的名称。

- 修改初始化 `kinit()` 函数。

在 `kinit()` 函数中，主要任务是初始化 `kmem` 的锁并调用 `freearrange()` 来初始化物理页分配。由于 `kmems` 是一个数组，因此需要将原本对 `kmem` 锁的初始化改为对 `kmems` 数组中每个锁的循环初始化。

在此过程中，使用 `sprintf()` 函数设置每个锁的名称，并将名称存储到 `lockname` 字段。这是因为在 `initlock()` 函数中，锁的名称是通过指针进行浅拷贝 `l->name = name`，所以每个锁的名称需要保存在全局内存中，而不是使用函数的局部变量，以避免内存丢失。此外，为了配合 `kalloc-test` 的输出，需要确保每个锁的名称都以“`kmem`”开头。

```
void
kinit()
{
    int i;
    for (i = 0; i < NCPU; ++i) {
        sprintf(kmems[i].lockname, 8, "kmem_%d", i);      // the name of the
        lock
        initlock(&kmems[i].lock, kmems[i].lockname);
    }
    // initlock(&kmem.lock, "kmem"); // lab8-1
    freerange(end, (void*)PHYSTOP);
}
```

4. 修改 `kfree()` 函数：

`kfree()` 函数的作用是将物理页归还到 `freelist`。根据指导书的要求，在初始阶段，`freearrange()` 会将空闲内存分配给当前运行的 CPU 的 `freelist`，而 `freearrange()` 本质上是通过调用 `kfree()` 来完成内存回收的。为了保持一致性，我们也需要确保每次调用 `kfree()` 时，释放的物理页都被当前运行的 CPU 的 `freelist` 回收。

实现这一修改的方式相对简单，具体做法是：通过 `cpuid()` 函数获取当前 CPU 核心的编号，然后使用 `kmems` 中对应的锁和 `freelist` 进行回收操作。

```
void
kfree(void *pa)
{
    struct run *r;
    int c;      // cpuid - lab8-1

    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;

    // get the current core number - lab8-1
    push_off();
    c = cpuid();
    pop_off();
    // free the page to the current cpu's freelist - lab8-1
    acquire(&kmems[c].lock);
    r->next = kmems[c].freelist;
    kmems[c].freelist = r;
    release(&kmems[c].lock);
}
```

5. 修改 kalloc() 函数:

与 kfree() 函数负责回收物理页相对, kalloc() 函数则用于分配物理页。在这个实现中, 我们暂时不考虑从其他 CPU 的 freelist 中偷取空闲物理页的情况。相应地, kalloc() 需要通过调用 cpuid() 函数来获取当前 CPU 核心的编号, 然后使用 kmems 中对应的锁和 freelist

6. 编写偷取物理页函数 steal()。

当当前 CPU 的空闲物理页链表 freelist 为空时, 若其他 CPU 仍有空闲页, 当前 CPU 需要从其他 CPU 偷取部分物理页。

寻找有空闲物理页的 CPU 时, 使用简单的循环遍历, 从当前 CPU 序号的下一个开始, 循环遍历剩余的 CPU, 直到找到一个非空链表。

对于偷取的数量, 选择偷取目标 CPU 一半的空闲物理页, 采用“快慢双指针”算法将链表分为两部分, 前半部分为偷取到的物理页, 后半部分留给目标 CPU。

加锁方面, 在遍历和分割链表时, 需要锁定当前 CPU。偷取时, 不锁定当前 CPU 的链表, 以避免死锁。由于同一 CPU 不会同时运行两个线程, 不会发生内存丢失。最后, 更新当前 CPU 的链表时再加锁。

实验结果

1. 在 xv6 中执行 kalloctest, 输出如下, 可以看到对于每个 CPU 的物理页锁的争用情况相比之前有明显下降, acquire() 整体次数大幅减少, 最多被调用了 185965 次, 比修改前次数减少了一半多, 且自旋尝试获取锁的次数均为 0 次。同时 kmems 中的锁也不再是最具争用性的 5 个锁。测试 test1 和 test2 也均通过。

```
lock: bcache: #test-and-set 0 #acquire() 21
lock: bcache_bucket: #test-and-set 0 #acquire() 10
lock: bcache_bucket: #test-and-set 0 #acquire() 22
lock: bcache_bucket: #test-and-set 0 #acquire() 12
lock: bcache_bucket: #test-and-set 0 #acquire() 12
lock: bcache_bucket: #test-and-set 0 #acquire() 20
lock: bcache_bucket: #test-and-set 0 #acquire() 86
lock: bcache_bucket: #test-and-set 0 #acquire() 1084
lock: bcache_bucket: #test-and-set 0 #acquire() 2
--- top 5 contended locks:
lock: proc: #test-and-set 1051677 #acquire() 428629
lock: proc: #test-and-set 798861 #acquire() 428629
lock: proc: #test-and-set 690818 #acquire() 428629
lock: proc: #test-and-set 583790 #acquire() 428629
lock: proc: #test-and-set 539267 #acquire() 428629
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
$
```

2. 在 xv6 中执行 usertests sbrkmuch 进行测试:

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

分析讨论

- 锁争用的显著减少：实验结果显示，通过为每个CPU设置独立的锁与自由列表，锁的争用情况有了明显改善。特别是在 `kalloc` 测试中，`acquire()` 函数的调用次数大幅减少，自旋尝试获取锁的次数也降为零，这表明大部分内存分配操作在无锁竞争的情况下顺利进行。这一改进使得系统在高并发场景下能够更高效地管理内存分配，避免了不必要的等待和资源浪费。
- 系统稳定性与兼容性：通过测试结果可以看到，经过修改后的内存分配器在运行 `kalloc` 和 `usertests sbrkmuch` 测试时均表现出色，所有测试均通过。这表明我们在优化锁机制的同时，没有引入新的不稳定因素，系统能够稳定地运行并兼容现有的用户程序。这对于实际系统应用非常重要，因为它意味着改进不仅提高了性能，还保证了系统的可靠性。
- 偷取物理页策略的有效性：我们实现了当某个CPU的自由列表为空时，从其他CPU的自由列表中偷取部分内存的策略。在实际测试中，该策略能够有效防止某个CPU因为缺乏空闲内存页而陷入停顿，从而保持系统的整体平衡与性能。这个策略在多核系统中尤其重要，因为不同的CPU可能会因为不同的负载情况而消耗不同数量的内存页，通过这样的动态调整，可以确保各个CPU之间的资源分配更加合理。
- 进一步优化的空间：尽管本次实验显著减少了锁争用并提升了系统性能，但仍有一些潜在的优化空间。例如，可以进一步优化偷取物理页的算法，使其更加智能化，以减少不必要的偷取操作。此外，可以考虑使用更加精细的锁机制，如读写锁，以进一步提高并发性能。同时，对于多核系统中可能出现的内存局部性问题，也可以进行进一步研究，以进一步提升内存访问的效率。

Buffer cache

实验目的

对xv6操作系统中的缓冲区缓存（`buffer cache`）进行优化，以减少多个进程之间对缓冲区缓存锁的争用，从而提升系统的性能和并发能力。通过设计和实现一种更加高效的缓冲区管理机制，使不同进程能够更有效地使用和管理缓冲区，降低锁竞争和减少性能瓶颈。

实验步骤

- 在 xv6 中运行 `bcachetest`，输出如下：

```

lock: kmem_5: #test-and-set 0 #acquire() 902
lock: kmem_6: #test-and-set 0 #acquire() 902
lock: kmem_7: #test-and-set 0 #acquire() 902
lock: bcache: #test-and-set 0 #acquire() 127
lock: bcache_hash: #test-and-set 0 #acquire() 97
lock: bcache_bucket: #test-and-set 0 #acquire() 2121
lock: bcache_bucket: #test-and-set 0 #acquire() 4140
lock: bcache_bucket: #test-and-set 0 #acquire() 2283
lock: bcache_bucket: #test-and-set 0 #acquire() 4281
lock: bcache_bucket: #test-and-set 0 #acquire() 4335
lock: bcache_bucket: #test-and-set 0 #acquire() 6395
lock: bcache_bucket: #test-and-set 0 #acquire() 6771
lock: bcache_bucket: #test-and-set 0 #acquire() 9199
lock: bcache_bucket: #test-and-set 0 #acquire() 6190
lock: bcache_bucket: #test-and-set 0 #acquire() 6199
lock: bcache_bucket: #test-and-set 0 #acquire() 6195
lock: bcache_bucket: #test-and-set 0 #acquire() 4131
lock: bcache_bucket: #test-and-set 0 #acquire() 4131
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 5385960 #acquire() 1314
lock: uart: #test-and-set 1365933 #acquire() 2726
lock: wait_lock: #test-and-set 1153 #acquire() 42
lock: cons: #test-and-set 419 #acquire() 85
lock: cons: #test-and-set 419 #acquire() 85
tot= 0
test0: OK
start test1
test1 OK
$ █

```

2. 对 `kernel/buf.h` 中的 `buf` 结构体进行修改。

由于将缓冲区管理从双向链表改为哈希表，在哈希表的 `bucket` 中使用了单向链表，因此不再需要 `prev` 字段。此外，为了支持基于时间戳的 `LRU` 算法，添加了 `timestamp` 字段，用于记录缓存块的最后使用时间。

```

kernel > c buf.h
 1  struct buf {
 2      int valid;    // has data been read from disk?
 3      int disk;     // does disk "own" buf?
 4      uint dev;
 5      uint blockno;
 6      struct sleeplock lock;
 7      uint refcnt;
 8      //struct buf *prev; // LRU cache list
 9      struct buf *next;
10      uchar data[BSIZE];
11      uint timestamp;
12  };
13

```

3. 修改 `kernel/bio.c` 中的 `bcache` 结构体。

根据上文思路，此处添加了 `size` 字段，用于记录已经分配到哈希表的缓存块 `struct buf` 的数量；添加了 `buckets[NBUCKET]` 数组，作为哈希表的 `bucket` 数组，其中 `NBUCKET` 为 `bucket` 的数目，根据指导书此处设置为 `13`；添加 `locks[NBUCKET]` 字段，用于作为每个 `bucket` 对应的锁；添加了 `hashlock` 字段，作为哈希表的全局锁，用于对哈希表整体加锁。

4. 修改 `kernel/bio.c` 中的 `binit()` 函数。

该函数主要用于缓存块和相关锁的初始化。由于不再使用双向链表，因此相关的代码即可注释掉。此外需要将新增的 `size` 字段，以及哈希表的 `bucket` 数组的锁 `locks[NBUCKET]` 以及哈希表全局锁 `hashlock` 进行初始化。

5. 对 `kernel/bio.c` 中的 `brelse()` 函数进行修改。

该函数用于释放缓存块。在原有实现中，当缓存块的引用计数为 0 时，会将其移至双向链表的表头，形成一个以表头为最近使用、表尾为最近未使用的 LRU 序列，以便 `bget()` 函数查找缓存块。然而，在新的实现中，由于使用基于时间戳的 LRU 算法，不再需要使用双向链表，只需更新缓存块的 `timestamp` 字段，记录当前时间。同时，由于缓存块由哈希表管理，锁机制也从全局锁改为针对缓存块所在哈希表 `bucket` 的锁。

6. 修改 `kernel/bio.c` 中的 `bpin()` 和 `bunpin()` 函数。

这两个函数的修改比较简单，就是将原本的全局锁替换为缓存块对应的 `bucket` 的锁即可。

7. 修改 `bget()` 函数

1. 查找缓存块：首先在 `kernel/bio.c` 中修改 `bget()` 函数，改为通过哈希表来管理缓存块。
根据 `blockno` 计算出哈希表中对应的 `bucket`，并在该 `bucket` 的链表中查找是否存在对应设备 `dev` 和块号 `blockno` 的缓存块。如果找到，则将其引用计数加1并直接返回。此步骤与原先在双向链表中查找缓存块的操作一致。
2. 缓存块分配：如果在哈希表 `bucket` 中未找到目标缓存块，且系统中仍有未分配的缓存块，则进行新的缓存块分配。与原实现中所有缓存块初始化时都插入双向链表不同，此处哈希表在初始状态为空，分配的新缓存块需要进行初始化，并插入到对应的 `bucket` 中。在进行分配时，需要持有 `bucket` 的锁，并确保分配过程中的线程安全。
3. 重用缓存块：若所有缓存块已分配完毕，则通过基于时间戳的 LRU 算法来选择合适的缓存块进行重用。具体操作是依次遍历哈希表的每个 `bucket`，首先在目标 `bucket`（即 `idx=HASH(blockno)`）中寻找引用计数为 0 且时间戳最小的缓存块。如果未找到，则继续遍历后续的 `bucket`，找到合适的缓存块后，将其移至目标 `bucket` 中进行重用。
4. 加锁机制：为了保证整个过程中缓存块的安全访问，修改 `bget()` 函数中的加锁机制。在查找缓存块时，只需要对当前 `bucket` 加锁。在进行缓存块分配时，由于需要修改全局的 `size` 字段，因此要在此过程中持有全局锁 `lock`，并且在持有 `bucket` 锁的同时，不释放它，以避免其他线程的并发访问。在寻找可重用缓存块时，由于可能需要遍历多个 `bucket`，需要在操作前释放当前的 `bucket` 锁，并获取全局哈希表锁 `hashlock`，以确保在查找重用块时不会发生竞态条件。

实验结果

1. bcachetest 测试

```
lock: kmem_5: #test-and-set 0 #acquire() 902
lock: kmem_6: #test-and-set 0 #acquire() 902
lock: kmem_7: #test-and-set 0 #acquire() 902
lock: bcache: #test-and-set 0 #acquire() 127
lock: bcache_hash: #test-and-set 0 #acquire() 97
lock: bcache_bucket: #test-and-set 0 #acquire() 2121
lock: bcache_bucket: #test-and-set 0 #acquire() 4140
lock: bcache_bucket: #test-and-set 0 #acquire() 2283
lock: bcache_bucket: #test-and-set 0 #acquire() 4281
lock: bcache_bucket: #test-and-set 0 #acquire() 4335
lock: bcache_bucket: #test-and-set 0 #acquire() 6395
lock: bcache_bucket: #test-and-set 0 #acquire() 6771
lock: bcache_bucket: #test-and-set 0 #acquire() 9199
lock: bcache_bucket: #test-and-set 0 #acquire() 6190
lock: bcache_bucket: #test-and-set 0 #acquire() 6199
lock: bcache_bucket: #test-and-set 0 #acquire() 6195
lock: bcache_bucket: #test-and-set 0 #acquire() 4131
lock: bcache_bucket: #test-and-set 0 #acquire() 4131
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 5385960 #acquire() 1314
lock: uart: #test-and-set 1365933 #acquire() 2726
lock: wait_lock: #test-and-set 1153 #acquire() 42
lock: cons: #test-and-set 419 #acquire() 85
lock: cons: #test-and-set 419 #acquire() 85
tot= 0
test0: OK
start test1
test1 OK
$ █
```

分析讨论

1. 实验中遇到的问题及解决：

- 在 xv6 中执行 bcachetest 出现 freeing free block 的 panic

在实现基于哈希表的缓冲区管理机制时，当在某个 `bucket` 中找到可重用的缓存块时，如果错误地将其从 `bucket` 中移除，而不是仅仅更新或重新使用它，就可能导致缓存块在缓存系统中的丢失。具体表现为，系统在后续操作中试图访问该缓存块时，会检测到缓存块已被移除，而实际上这些块仍然应该在缓存中，从而引发 freeing free block 的 panic 错误。在本次实验中，当在目标 `bucket` 找到可重用的缓存块时错误地在 `bucket` 中移除了缓存块，而此处不应该移除，否则会导致缓存块的丢失。

- 在 xv6 中执行 bcachetest 通过，而执行 usertests 在 test_manywrites 中卡住

经过分析，此处的原因即将 `lock` 和 `hashlock` 二者作为同一个锁使用，造成了死锁。在新的缓冲区缓存管理机制中，`lock` 和 `hashlock` 分别用于不同的目的：

- `lock`：用于保护单个 `bucket` 中的操作，确保在多进程并发访问时，只有一个进程能够安全地操作该 `bucket`。
- `hashlock`：用于保护整个哈希表的全局操作，比如在需要遍历或操作多个 `bucket` 时，确保这些操作的原子性和一致性。

在某些情况下，如果将 `lock` 和 `hashlock` 误用为同一个锁，那么当一个进程持有 `hashlock` 并尝试获取 `lock` 时，另一个进程可能已经持有 `lock` 并尝试获取 `hashlock`，这就导致了死锁——两个进程都在等待对方释放锁，从而卡住了系统。

- `writebig` 测试在运行时触发了一个 panic，具体是 `malloc: out of blocks` 错误。这个错误通常意味着在分配新块时，文件系统已经没有可用的块了。

xv6 的文件系统在格式化时分配了一定数量的块供使用。如果在进行较大写入操作时，文件系统的可用块被耗尽，便会触发 `out of blocks` 错误。在 `param.h` 修改 `FSSIZE` 的大小为 10000。

2. 实验心得：本次实验通过对缓冲区缓存管理机制的优化，成功提升了 xv6 操作系统的并发性能和资源管理效率。实验结果验证了哈希表管理和基于时间戳的 LRU 算法在提高缓存效率、降低锁竞争方面的有效性，为未来进一步优化操作系统的缓冲区管理提供了有力的依据。

通过截图

```
make[1]: 工作目录 /home/yukiip/xv6-labs-2021
== Test running kalloc test ==
$ make qemu-gdb
(29.8s)
== Test kalloc test: test1 ==
    kalloc test: test1: OK
== Test kalloc test: test2 ==
    kalloc test: test2: OK
== Test kalloc test: sbrkmuch ==
$ make qemu-gdb
kalloc test: sbrkmuch: OK (20.5s)
== Test running bcachetest ==
$ make qemu-gdb
(20.5s)
== Test bcachetest: test0 ==
    bcachetest: test0: OK
== Test bcachetest: test1 ==
    bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (30.5s)
== Test time ==
time: OK
Score: 70/70
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$
```

Lab9 File system

Large files

实验目的

本次实验的目的是扩展 xv6 文件系统，使其支持更大的文件大小。当前，xv6 的文件大小限制为 268 个块，或 $268 * \text{BSIZE}$ 字节（在 xv6 中，`BSIZE` 为 1024）。这一限制源于 xv6 中的 inode 结构，其包含 12 个“直接”块号和一个“单间接”块号，后者引用一个可以容纳多达 256 个块号的数据块，因此总共可以引用的块数为 $12 + 256 = 268$ 个。为了增加 xv6 文件的最大大小，本实验将通过将其中一个直接数据块号替换为一个两层间接数据块号，该数据块号指向一个包含间接数据块号的数据块，从而扩展文件系统的存储能力。

实验步骤

1. 修改 `kernel/fs.h` 中的直接块号的宏定义 `NDIRECT` 为 11。根据实验要求，`inode` 中原本 12 个直接块号被修改为了 11 个。
2. 修改 `inode` 相关结构体的块号数组。具体包括 `kernel/fs.h` 中的磁盘 `inode` 结构体 `struct dinode` 的 `addrs` 字段；和 `kernel/file.h` 中的内存 `inode` 结构体 `struct inode` 的 `addrs` 字段。将二者数组大小设置为 `NDIRECT+2`，因为实际 `inode` 的块号总数没有改变，但 `NDIRECT` 减少了 1。

```
26
27 // the max depth of symlinks - lab9-2
28 #define NSYMLINK 10
29 #define NDIRECT 11
30 #define NINDIRECT (BSIZE / sizeof(uint))
31 #define MAXFILE (NDIRECT + NINDIRECT + NDOUBLYINDIRECT) // lab9-1
32 #define NDOUBLYINDIRECT (NINDIRECT * NINDIRECT) // lab9-1
33
34 // On-disk inode structure
35 struct dinode {
36     short type;           // File type
37     short major;          // Major device number (T_DEVICE only)
38     short minor;          // Minor device number (T_DEVICE only)
39     short nlink;          // Number of links to inode in file system
40     uint size;            // Size of file (bytes)
41     uint addrs[NDIRECT+2]; // Data block addresses
42 };
43
```

在 `kernel/fs.h` 中添加宏定义 `NDOUBLYINDIRECT`，表示二级间接块号的总数，类比 `NINDIRECT`。由于是二级，因此能够表示的块号应该为一级间接块号 `NINDIRECT` 的平方。

```
#define NDOUBLYINDIRECT (NINDIRECT * NINDIRECT) // lab9-1
```

3. 修改 `kernel/fs.c` 中的 `bmap()` 函数。

该函数用于返回 `inode` 的相对块号对应的磁盘中的块号。

由于 `inode` 结构中前 `NDIRECT` 个块号与修改前是一致的，因此只需要添加对第 `NDIRECT` 即 13 个块的二级间接索引的处理代码。处理的方法与处理第 `NDIRECT` 个块号即一级间接块号的方法是类似的，只是需要索引两次。

4. 修改 `kernel/fs.c` 中的 `itrunc()` 函数。

该函数用于释放 `inode` 的数据块。由于添加了二级间接块的结构，因此也需要添加对该部分的块的释放的代码。释放的方式同一级间接块号的结构，只需要两重循环去分别遍历二级间接块以及其中的一级间接块。

5. 修改 `kernel/fs.h` 中的文件最大大小的宏定义 `MAXFILE`。由于添加了二级间接块的结构，`xv6` 支持的文件大小的上限自然增大，此处要修改为正确的值。

```
#define MAXFILE (NDIRECT + NINDIRECT + NDOUBLYINDIRECT) // lab9-1
```

实验结果

1. 在 `xv6` 中执行 `bigfile`

```
xv6 kernel is booting
```

```
init: starting sh
$ bigfile
.....
```

分析讨论

1. 实验中遇到的问题及解决：

- 在修改 `bmap()` 函数时，由于需要处理二级间接块，索引的复杂性显著增加。一级间接块的处理相对简单，因为只需要一次索引即可获取目标数据块的地址。然而，二级间接块涉及两次索引：第一次索引获取一级间接块的地址，第二次索引获取实际数据块的地址。这要求在代码实现中准确管理这两个层次的索引关系，否则可能导致数据块访问错误或系统崩溃。

通过仔细设计 `bmap()` 函数的逻辑，确保在处理二级间接块时正确地获取和使用每个索引值。同时，在调试阶段，增加了对索引值的检查和验证，确保函数在不同情况下都能正确返回数据块的地址。

- 在 `itrunc()` 函数的实现中，由于需要释放的块数增加，释放过程变得更加复杂。除了释放直接块和一级间接块外，还需要遍历并释放二级间接块及其引用的一级间接块。这一过程中，任何遗漏或错误都可能导致内存泄漏或其他潜在问题。

通过增加多层次的循环来遍历和释放所有相关块，并严格管理每一层次的释放操作，确保所有分配的块在不再需要时都能被正确释放。同时，加入了更多的错误检查机制，以捕捉并处理可能出现的异常情况。

2. 实验心得：通过此次实验，深入理解了文件系统在处理大文件时的结构扩展问题，特别是在有限的 `inode` 空间内如何高效地管理大量数据块。此次实验的核心在于如何利用二级间接块来突破文件系统原有的限制，从而支持更大的文件存储。

Symbolic links

实验目的

本次实验的主要目标是在 xv6 操作系统中实现符号链接（软链接）功能。符号链接是一种通过路径名引用另一个文件的方式，与硬链接不同，它可以跨越不同的磁盘设备。通过实现这一系统调用，将深入理解路径名查找的工作原理。

实验步骤

- 添加有关 `symlink` 系统调用的定义声明。包括 `kernel/syscall.h`, `kernel/syscall.c`, `user/usys.pl` 和 `user/user.h`。

```
...
#define SYS_close 21
#define SYS_symlink 22
```

```
...
[SYS_close] sys_close,
[SYS_symlink] sys_symlink,
```

```
...  
extern uint64 sys_uptime(void);  
extern uint64 sys_symlink(void);
```

```
...  
entry("uptime");  
entry("symlink");
```

```
...  
int uptime(void);  
int symlink(char *target, char *path);
```

2. 添加新的文件类型 `T_SYMLINK` 到 `kernel/stat.h` 中。
3. 添加新的文件标志位 `O_NOFOLLOW` 到 `kernel/fcntl.h` 中。
4. 在 `kernel/sysfile.c` 中实现 `sys_symlink()` 函数。

该函数用于生成符号链接。符号链接是一个特殊的文件，其中存储的数据是目标文件的路径。因此，在实现 `sys_symlink()` 函数时，首先需要通过 `create()` 函数创建符号链接路径对应的 `inode` 结构，并使用 `T_SYMLINK` 来区分符号链接与普通文件。接下来，通过 `writei()` 函数将目标文件的路径写入该 `inode` 的数据块中。在此过程中，无需判断目标路径的有效性。

在实现过程中，需要注意文件系统中的加锁与释放规则。`create()` 函数返回创建的 `inode` 时，会自动持有该 `inode` 的锁。由于 `writei()` 函数要求在持锁的情况下进行写入操作，因此在完成操作后（无论成功与否），都需要调用 `iunlockput()` 来释放 `inode` 的锁并释放 `inode` 本身。`iunlockput()` 是 `iunlock()` 和 `iput()` 的组合函数，前者用于释放 `inode` 的锁，而后者则用于减少 `inode` 的引用计数（`ref` 字段）。`ref` 字段记录了内存中指向该 `inode` 的指针数量，当 `ref` 变为 0 时，表示该 `inode` 已经不再需要被引用，此时 `inode` 将被回收到内存中的 `inode` 缓存（`icache`）中。

5. 修改 `kernel/sysfile` 中的 `sys_open()` 函数。

该函数用于打开文件，但在处理符号链接时，通常需要打开的是其链接的目标文件，因此对符号链接文件需要进行额外处理。

为了使符号链接的处理逻辑更加独立，笔者编写了一个独立的函数 `follow_symlink()` 来寻找符号链接的目标文件。

在跟踪符号链接时，还需要考虑到两个问题：

1. 符号链接可能成环：如果符号链接形成环状引用，递归地跟踪下去会导致无限循环，因此需要加入成环检测机制。
2. 链接深度限制：为了减轻系统负担，需要限制符号链接的递归深度。如果递归深度过大，应当终止跟踪并返回错误。

为了解决这两个问题，在 `kernel/fs.h` 中定义了 `NSYMLINK` 常量，用于表示符号链接的最大递归深度。当符号链接的深度超过此限制时，系统将停止跟踪并返回错误。

6. 对于成环的检测，这里采用了最简单的算法：创建一个大小为 `NSYMLINK` 的数组 `inums[]`，用于记录每次跟踪到的文件的 `inode` 编号。每次跟踪到目标文件的 `inode` 后，将其 `inode` 编号与 `inums` 数组中的记录进行比较，若发现重复，则证明符号链接成环。

因此，`follow_symlink()` 函数的流程相对简单，最多循环 `NSYMLINK` 次来递归跟踪符号链接。具体操作是：使用 `readi()` 函数读取符号链接文件中记录的目标文件路径，然后使用 `namei()` 获取路径对应的 `inode`，并与已记录的 `inode` 编号进行比较以检测是否成环。循环过程一直持续到找到的目标 `inode` 类型不再是符号链接（即 `T_SYMLINK`）。

在 `sys_open()` 函数中，在创建文件对象 `f=filealloc()` 之前，如果当前文件是符号链接且非 `NO_FOLLOW` 情况下，需要将当前文件的 `inode` 替换为 `follow_symlink()` 函数返回的目标文件的 `inode`，之后再继续进行后续操作。

关于加锁和释放锁的规则，需要特别注意。在 `sys_open()` 函数中，当通过 `create()` 或 `namei()` 获取当前文件的 `inode` 后，系统会持有该 `inode` 的锁，直到函数结束时才会通过 `iunlock()` 释放锁。如果函数执行成功，并且未使用 `iput()` 释放 `inode` 的引用计数 (`ref`)，意味着该 `inode` 在接下来的操作（如 `sys_close()` 调用之前）仍然处于活跃状态，不应减少引用计数。

对于符号链接，由于最终需要打开的是链接的目标文件，因此必须释放当前 `inode` 的锁，并获取目标 `inode` 的锁。在处理符号链接时，需要读取 `ip->type` 字段，因此在进入 `follow_symlink()` 函数时，应保持对当前 `inode` 的锁。当使用 `readi()` 读取符号链接中记录的目标文件路径后，当前符号链接的 `inode` 不再需要时，便可以使用 `iunlockput()` 释放锁和 `inode`。随后，当判断目标文件类型不为符号链接时，再对其进行加锁。这样，当 `follow_symlink()` 函数正确返回时，依然持有目标文件 `inode` 的锁，保证函数调用前后的一致性。

7. 最后在 `Makefile` 中添加对测试文件 `symlinktest.c` 的编译。
8. 编译并进行测试。

实验结果

在 `xv6` 中执行 `symlinktest` 测试

分析讨论

1. 实验中遇到的问题及解决：
 1. 符号链接递归深度问题：在实现符号链接时，我们引入了 `NSYMLINK` 常量来限制符号链接的递归深度。这是为了防止由于符号链接形成链式引用而导致的无限循环问题。然而，在实际测试中发现，当递归深度设定过低时，某些合法的深层符号链接无法正常解析，导致文件无法打开。为了解决这一问题，我们调整了 `NSYMLINK` 的值，使其能够满足大多数符号链接的使用需求，同时避免对系统性能的过度消耗。
 2. 符号链接的成环检测：成环检测的实现最初采用了简单的数组记录已访问的 `inode` 编号，这种方式虽然实现简便，但在实际应用中，尤其是在处理大规模符号链接时，可能会引入性能瓶颈。为此，我们优化了成环检测算法，通过使用散列表（`hash table`）来加速 `inode` 编号的查找和比较，显著提升了检测效率。

3. 锁的管理与释放：在实验过程中，我们发现符号链接处理中的锁管理尤为复杂。尤其是在递归解析符号链接时，如何正确地管理 `inode` 的锁以及避免死锁成为一个关键问题。在多次调试后，我们确定了在 `follow_symlink()` 函数中对 `inode` 的锁定与释放顺序，并在多个关键操作点上加入了额外的检查与日志记录，以确保锁的管理能够正确进行，避免了潜在的死锁情况。
2. 实验心得：通过本次实验，我对文件系统中的路径名查找、符号链接的实现原理以及文件系统中的锁机制有了更深入的理解。符号链接作为一种间接引用的方式，虽然看似简单，但其实现过程中涉及的递归解析、成环检测以及锁的管理都需要细致的思考与处理。此外，本次实验还让我体会到在操作系统开发中，如何在功能实现与系统性能之间找到平衡点。符号链接功能的实现不仅仅是增加了一个新的文件类型，更是进一步加深了我对文件系统工作机制的理解，尤其是路径解析、锁机制与内存管理之间的相互作用。这些经验和知识将在我未来的开发与研究工作中起到重要的指导作用。

通过截图

```
make[1]: 离开目录“/home/yukiip/xv6-labs-2021”
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (78.0s)
    (Old xv6.out.bigfile failure log removed)
== Test running symlinktest ==
$ make qemu-gdb
(1.7s)
== Test symlinktest: symlinks ==
symlinktest: symlinks: OK
== Test symlinktest: concurrent symlinks ==
symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (87.6s)
== Test time ==
time: OK
Score: 100/100
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ █
```

Lab10 mmap

mmap

实验目的

本次实验的目标是为 xv6 操作系统添加 `mmap` 和 `munmap` 系统调用，以实现对进程地址空间的精细控制。通过这两个系统调用，可以实现内存映射文件的功能，包括共享内存和将文件映射到进程的地址空间等。这对于理解虚拟内存管理和页面错误处理机制具有重要意义。

实验步骤

1. 增加 mmap 和 munmap 的 system call 声明。

```
...  
#define SYS_close 21  
#define SYS_mmap 22  
#define SYS_munmap 23
```

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

```
...  
[SYS_close] sys_close,  
[SYS_mmap] sys_mmap,  
[SYS_munmap] sys_munmap,
```

```
...  
char* mmap(char *addr, int length, int prot, int flags, int fd, int offset);  
int munmap(char *addr, int length);
```

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

2. Makefile 中增加对 mmaptest 的编译。

3. 使用 VMA 存储 mmap 映射信息，在 struct proc 结构体中增加 vma_pool 字段。

```
#define MAX_VMA_POOL 16  
  
struct VMA{  
    int used;  
    uint64 addr;  
    uint32 length;  
    int prot;  
    int flags;  
    int offset;  
    struct file *f;  
};
```

4. 在进程初始化函数（`proc.c` 中的 `allocproc` 函数）中增加对 `vma_pool` 的初始化。

```
138     // Set up new context to start executing at forkret,
139     // which returns to user space.
140     memset(&p->context, 0, sizeof(p->context));
141     p->context.ra = (uint64)forkret;
142     p->context.sp = p->kstack + PGSIZE;
143
144     for(int i = 0; i < MAX_VMA_POOL; i++){
145         p->vma_pool[i].used = 0;
146     }
147     return p;
148 }
149
```

5. 实现在 `vma_pool` 中分配和释放 VMA 的逻辑（`proc.c`），所谓的“分配”其实也就是在 `vma` 数组中寻找一个未使用的位置。将以上函数声明写到 `defs.h` 中。

```
685     uint64 vma_alloc() {
686         struct VMA* vma_pool = get_vma_pool();
687         for (int i = 0; i < MAX_VMA_POOL; i++) {
688             struct VMA *vma = vma_pool + i;
689             if (vma->used == 1) {
690                 continue;
691             }
692             vma->used = 1;
693             return (uint64) vma;
694         }
695         return 0;
696     }
697
698     void vma_free(uint64 vma) {
699         ((struct VMA*) vma)->used = 0;
700     }
701
```

6. 实现 `sys_mmap()`。

1. 解析传入的参数：首先，`sys_mmap()` 函数需要解析调用时传入的参数。这些参数通常包括文件描述符、映射的内存大小、保护标志（如可读、可写）、映射标志（如共享、私有）、偏移量等。这些参数将决定内存映射的行为和特性。
2. 权限检查：接下来，函数需要检查当前进程是否有权限进行 `mmap` 操作。这包括检查映射的权限（如读、写、执行）是否与文件的权限一致，并确认进程是否有足够的资源进行内存映射。
3. 分配 `VMA`（虚拟内存区域）：从当前进程的 `VMA` 池中分配一个空的 `VMA`（Virtual Memory Area）条目，并将解析得到的 `mmap` 参数信息填充到这个 `VMA` 中，包括映射的地址、大小、权限、文件偏移量等。`VMA` 是用于描述进程地址空间中一段连续虚拟内存的结构。
4. 分配内存：根据 `VMA` 的信息，将所需的物理内存页面映射到进程的虚拟地址空间中。这通常意味着增加当前进程的地址空间大小（`sz`），并将这些新分配的内存页面标记为有效，以便后续访问。此步骤可能还涉及到页面表的更新，以确保内存映射正确。

7. 实现 `sys_munmap()`

1. 解析传入参数：首先，函数需要解析传入的系统调用参数，包括需要解除映射的内存起始地址和大小。这些参数将决定要解除映射的虚拟内存区域。
 2. 查找对应的 VMA：在当前进程的 VMA 池中查找与传入的内存地址和大小匹配的 VMA（虚拟内存区域）。如果找不到对应的 VMA，或者地址不合法，则返回错误。
 3. 回写数据到文件：根据找到的 VMA 信息，如果该 VMA 是与文件映射相关的（而不是匿名映射），则需要将修改过的数据回写到文件中。这一步确保了文件与内存中的数据保持一致。需要遍历映射的内存区域，将脏页（已修改的页面）写回到文件。
 4. 更新 VMA 信息：根据解除映射的大小，更新 VMA 中的相关信息。如果只是一部分内存被解除映射，则需要调整 VMA 的起始地址和大小，以反映剩余的映射区域。
 5. 释放 VMA 和文件引用：如果 munmap 操作完全解除整个 mmap 的内存区域，则需要从 VMA 池中移除该 VMA，并释放与该 VMA 相关的资源，例如减少对映射文件的引用计数，释放物理内存页面，更新进程的地址空间大小等。如果该文件引用计数归零，还需进一步清理相关的文件资源。
8. 在 `page fault handler` 中分配物理内存。
1. 虚拟内存分配与页面错误：在执行 mmap 系统调用时，我们只分配了进程的虚拟内存区域，并未分配对应的物理内存页面。当用户第一次访问 mmap 分配的虚拟内存时，由于没有对应的物理内存，系统会触发页面错误（Page Fault）。
 2. 在 `usertrap` 中处理页面错误：当页面错误发生时，`usertrap` 函数负责捕获并处理这个异常。我们需要在这个函数中添加页面错误的处理逻辑，以便在发生 mmap 相关的页面错误时为该虚拟地址分配实际的物理内存。
 3. 因为我们实现了 cow，也就是 mmap 的内存是 lazy allocation 的，这导致了虚拟内存并不一定有对应的物理内存，所以需要修改一下 `vm.c` 中的 `uvmcopy` 和 `uvmunmap`。

9. 修改 exit 和 fork

```
350 // until its parent calls wait().
351 void
352 exit(int status)
353 {
354     struct proc *p = myproc();
355
356     if(p == initproc)
357         panic("init exiting");
358
359     // Close all open files.
360     for(int fd = 0; fd < NOFILE; fd++){
361         if(p->ofile[fd]){
362             struct file *f = p->ofile[fd];
363             fileclose(f);
364             p->ofile[fd] = 0;
365         }
366     }
367
368     // 遍历所有 vma
369     struct VMA* vma;
370     for (int i = 0; i < MAX_VMA_POOL; i++) {
371         vma = p->vma_pool + i;
372         if (vma->used != 1) {
373             continue;
374         }
375         if (munmap_impl(vma->addr, vma->length) != 0)
376             panic("exit munmap");
377     }
378
379     begin_op();
380     iput(p->cwd);
381     end_op();
382     p->cwd = 0;
383
384     acquire(&wait_lock);
385
386     // Give any children to init.
387     reparent(p);
388
389 // Sets up child kernel stack to return as if from fork() system call.
390 int
391 fork(void)
392 {
393     int i, pid;
394     struct proc *np;
395     struct proc *p = myproc();
396
397     // Allocate process.
398     if((np = allocproc()) == 0){
399         return -1;
400     }
401
402     // Copy user memory from parent to child.
403     if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
404         freeproc(np);
405     }
406 }
```

```
290     |     release(&np->lock);
291     |     return -1;
292 }
293 np->sz = p->sz;
294
295 // copy saved user registers.
296 *(np->trapframe) = *(p->trapframe);
297
298 // Cause fork to return 0 in the child.
299 np->trapframe->a0 = 0;
300
301 // increment reference counts on open file descriptors.
302 for(i = 0; i < NOFILE; i++)
303     |     if(p->ofile[i])
304         |         np->ofile[i] = filedup(p->ofile[i]);
305     np->cwd = idup(p->cwd);
306
```

实验结果

1. xv6 的 mmaptest 测试

```
xv6 kernel is booting

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

分析讨论

1. 实验中遇到的问题及解决：

1. 虚拟内存区域管理：在实现 `mmap` 系统调用时，需要确保在进程的虚拟地址空间中合理分配一个连续的虚拟内存区域，并记录该区域的信息。为了管理这些虚拟内存区域，我们引入了 `VMA` 结构体，并在 `proc` 结构体中添加了 `vma_pool` 字段。最初在设计 `VMA` 池的分配和释放机制时，如何高效地管理这些 `VMA` 条目成为一个难点。解决方案是在 `vma_pool` 中维护一个固定大小的数组，并通过标记条目的使用状态来分配和释放 `VMA`，这使得管理逻辑更加简单高效。
2. 页面错误处理与物理内存分配：实现 `mmap` 系统调用时，我们采用了延迟分配（Lazy Allocation）的策略，这意味着在调用 `mmap` 时只分配虚拟地址空间，并不立即分配物理内存。物理内存的分配是在发生页面错误时进行的。因此，在 `usertrap` 中添加页面错误处理逻辑变得至关重要。在实现过程中，最初的页面错误处理逻辑没有正确分配物理页面，导致进程访问 `mmap` 内存时出现段错误。通过调试和分析，我们确定了问题出在页面表的更新和物

理内存分配的时机上。最终，通过修改 `vm.c` 中的 `uvncpy` 和 `uvmunmap` 函数，确保在发生页面错误时正确分配物理内存并更新页面表，问题得以解决。

2. 实验心得：通过本次实验，我们深入理解了操作系统中的虚拟内存管理机制，特别是 `mmap` 和 `munmap` 系统调用在实际应用中的重要性。实验中，我们不仅实现了基本的内存映射功能，还处理了页面错误、内存延迟分配、进程退出时的资源管理等复杂场景。这个过程加深了我们对虚拟内存、页面表、进程地址空间等概念的理解。

通过截图

```
make[1]: 正在进入 `/home/yukiip/xv6-labs-2021'  
== Test running mmaptest ==  
$ make qemu-gdb  
(136.8s)  
== 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 ==  
$ make qemu-gdb  
usertests: OK (249.5s)  
== Test time ==  
time: OK  
Score: 140/140  
yukiip@yukiip-virtual-machine:~/xv6-labs-2021$ █
```