

# Xv6 Labs

2253551 李沅衡

学院：软件学院    专业：软件工程

源代码：<https://github.com/Yuen647/2024-Tongji-University-SSE-OS-xv6.git>

## Xv6 Labs

### 环境搭建

- 1. 安装 VMware Workstation
- 2. 安装 Ubuntu 20.04
- 3. 配置 Ubuntu 20.04
- 4. 安装 RISC-V 交叉编译工具
- 5. 下载并编译 xv6
- 6. 运行 xv6
- 7. 检查工具链
- 8. 测试

### lab1 Xv6 and Unix utilities

#### 实验总览

#### sleep

- 实验目的
- 实现步骤

#### pingpong

- 实验目的
- 实现步骤
- 说明
- 实验现象

#### primes

- 实验目标
- 实现步骤
- 说明
- 实验现象

#### find

- 实验目标
- 实现步骤
- 说明
- 实验现象

#### xargs

- 实验目标
- 实现步骤
- 说明
- 实验现象

#### 实验结果

#### 实验中遇到的问题及解决办法

#### 实验心得

### lab2 system calls

#### 实验总览

#### System call tracing

- 实验目的
- 实验过程

#### Sysinfo

- 实验目的
- 实验过程

#### 实验结果

实验中遇到的问题及解决办法

实验总结

lab3 Page tables

实验总览

Print a Page Table

实验目的

实验步骤

实验现象

A Kernel Page Table Per Process

实验目的

实验步骤

实验现象

Simplify copyin/copyinstr

实验目的

实验步骤

实验结果

实验中遇到的问题及解决办法

Speed Up System Calls

Print a Page Table

实验心得

lab4 Traps

实验综述

Backtrace

实验目的

实验步骤

Alarm

实验目的

实验步骤

实验结果

实验中遇到的问题和解决办法

实验心得

lab 5: lazy page allocation

实验目的

Eliminate Allocation from `sbrk()`

实验目的

实验步骤

Lazy Allocation

实验目的

实验步骤

实验中遇到的问题及解决办法

Lazytests and Usertests

实验目的

实验步骤

实验结果

实验中遇到的问题及解决办法

实验心得

lab 6: Multithreading

实验综述

Uthread: switching between threads

实验目的

实验步骤

Using Threads

实验目的

实验步骤

Barrier

实验目的

## 实验步骤

### 实验结果

### 实验中遇到的问题及解决方法

#### Uthread

1. 上下文切换问题
2. 线程调度问题

#### Barrier

1. 条件变量和互斥锁的结合使用

### 实验心得

1. 理解上下文切换的重要性
2. 掌握线程创建和调度的核心
3. 解决线程安全问题

## lab 7: Networking

### 实验目的

### 实验原理

### 实验步骤

1. E1000的交互方法
2. 接收描述符
3. 发送描述符
4. 寄存器操作
5. 实现代码

发送函数 `e1000_transmit`

接收函数 `e1000_recv`

### 实验结果

### 实验中遇到的问题及解决方法

1. 缺乏描述符空间
2. 多线程访问冲突
3. 内存分配失败

### 实验心得

## lab8: Locks

### 实验总览

#### Memory Allocator

### 实验目的

### 实验原理

### 实验步骤及分析

### 实验现象

#### Buffer Cache

### 实验目的

### 实验原理

### 实验步骤

### 实验现象

### 实验结果

### 实验中遇到的问题及解决方法

- 问题1: 锁争用高导致性能下降
- 问题2: 内存或资源不足
- 问题3: 并发一致性问题

### 实验心得

## lab 9: File System

### 实验总览

#### Large Files

### 实验目的

### 实验步骤

#### Symbolic Links

### 实验目的

### 实验步骤

实验结果  
实验中遇到的问题及解决方法  
    Large Files  
    Symbolic Links  
实验心得  
lab 10: mmap  
实验总览  
    内存映射文件的基本概念  
    虚拟内存与内存映射文件  
    mmap 和 munmap 系统调用  
    内存映射的操作流程  
实验目的  
实验步骤  
实验中遇到的问题及解决方法  
实验心得

## 环境搭建

---

### 1. 安装 VMware Workstation

访问 VMware 官方网站，下载 VMware Workstation 安装包。

### 2. 安装 Ubuntu 20.04

#### 1. 下载 Ubuntu 20.04 ISO 镜像

- 访问 Ubuntu 官方网站，下载 Ubuntu 20.04 的 ISO 镜像文件。

#### 2. 创建新的虚拟机

- 打开 VMware Workstation，选择 `File > New Virtual Machine`。
- 选择 `Typical (recommended)`，点击 `Next`。
- 选择 `Installer disc image file (iso)`，然后选择下载的 Ubuntu 20.04 ISO 镜像文件，点击 `Next`。
- 设置虚拟机的名称和位置，点击 `Next`。
- 设置磁盘大小，选择 `Store virtual disk as a single file`，点击 `Next`。
- 点击 `Finish`，完成虚拟机的创建。

#### 3. 安装 Ubuntu 20.04

- 启动刚刚创建的虚拟机，按照提示安装 Ubuntu 20.04。

### 3. 配置 Ubuntu 20.04

#### 1. 更新系统

- 打开终端，运行以下命令来更新系统：

```
1 | sudo apt update
2 | sudo apt upgrade -y
```

#### 2. 安装必要的软件包

- 为了进行 xv6 实验，需要安装一些必要的软件包。运行以下命令：

```
1 | sudo apt install build-essential qemu gdb git -y
```

## 4. 安装 RISC-V 交叉编译工具

### 1. 安装 RISC-V 工具链

- 在 Ubuntu 中，运行以下命令来安装 RISC-V 交叉编译工具：

```
1 | sudo apt install git build-essential gdb-multiarch qemu-system-misc  
gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu libglib2.0-dev  
libpixman-1-dev gcc-riscv64-unknown-elf
```

### 2. 安装 QEMU

- QEMU 用于在 x86 机器上模拟 RISC-V 架构的 CPU。运行以下命令安装 QEMU：

```
1 | wget https://download.qemu.org/qemu-5.1.0.tar.xz -P /home6s081  
2 | cd /home/qemu-5.1.0  
3 | tar xvjf qemu-5.1.0.tar.xz  
4 | cd qemu-5.1.0  
5 | ./configure --disable-kvm --disable-werror --prefix=/usr/local --  
target-list=riscv64-softmmu  
6 | make  
7 | sudo make install
```

## 5. 下载并编译 xv6

### 1. 克隆 xv6 源代码

- 在终端中运行以下命令来克隆 xv6 源代码：

```
1 | git clone https://github.com/mit-pdos/xv6-public.git  
2 | cd xv6-public
```

### 2. 编译 xv6

- 在 xv6 目录中运行以下命令来编译 xv6：

```
1 | make
```

## 6. 运行 xv6

### 1. 启动 xv6

- 在 xv6 目录中运行以下命令来启动 xv6：

```
1 | make qemu
```

### 2. 调试 xv6

可以使用 gdb 来调试 xv6。在一个终端中运行以下命令启动 xv6 并等待 gdb 连接：

```
1 | make qemu-gdb  
2 | sed "s/1234/26000/" < .gdbinit.tmpl-riscv > .gdbinit
```

在另一个终端中运行以下命令来启动 gdb:

```
1 | gdb-multiarch -q kernel/kernel
```

## 7. 检查工具链

### 1. 检查 RISC-V 工具链

- 在终端中运行以下命令检查 RISC-V 工具链的版本:

```
1 | riscv64-unknown-elf-gcc --version
```

- 预期输出:

```
1 | riscv64-unknown-elf-gcc (GCC) 10.1.0
```

### 2. 检查 QEMU 版本

- 在终端中运行以下命令检查 QEMU 版本:

```
1 | qemu-system-riscv64 --version
```

- 预期输出:

```
1 | QEMU emulator version 5.1.0
```

## 8. 测试

### 1. 运行 xv6 测试

- 在 xv6 目录中运行以下命令启动 xv6 测试:

```
1 | make qemu
```

- 如果能进入 xv6 的 shell, 看到 `xv6 kernel is booting` 则表示实验环境已搭建成功。

### 2. 调试 xv6

- 打开两个终端窗口, 在第一个窗口中运行以下命令启动 qemu 并等待 gdb 连接:

```
1 | make qemu-gdb
```

- 在另一个窗口中运行以下命令启动 gdb:

```
1 | gdb-multiarch -q kernel/kernel
```

- 如果看到如下内容, 则基本上环境没有问题:

```
1 | Reading symbols from kernel/kernel...
2 | The target architecture is set to "riscv:rv64".
3 | 0x00000000000001000 in ?? ()
4 | (gdb)
```

至此，MIT 6.S081 环境搭建完成！

## lab1 Xv6 and Unix utilities

### 实验总览

在本实验中，通过实现一系列用户级工具函数来深入学习系统调用的使用和进程间通信。

#### 1. `sleep`:

- 实现一个简单的功能，让进程休眠指定的时间。这个工具通过调用xv6的 `sleep` 系统调用，让调用它的进程暂停执行。

#### 2. `pingpong`:

- 这个实验涉及进程间的基本通信。父进程和子进程之间通过管道传递消息，父进程发送消息给子进程，子进程接收到后打印"ping"；反过来，子进程发送消息给父进程，父进程打印"pong"。

#### 3. `primes`:

- 使用多进程和管道技术实现一个质数计算器。这个实验通过创建多个进程链，每个进程负责筛选出一定范围内的质数，展示了进程间通信和数据流的处理。

#### 4. `find`:

- 实现一个查找工具，用于搜索并显示指定目录下的文件或目录名。这个工具需要遍历文件系统，并匹配目标文件名。

#### 5. `xargs`:

- 实现一个类似于Unix系统中的xargs工具，该工具能够从标准输入接收数据并构造成命令行参数执行其他命令。这是对进程创建和标准输入输出处理的良好练习。

```
1 | $ git checkout util
```

### sleep

#### 实验目的

- 实现一个简单的 UNIX 程序 `sleep`，该程序能够在 xv6 操作系统中按用户指定的 ticks 数暂停执行。
- 熟悉 xv6 操作系统的系统调用机制和时间管理机制。
- 所有代码应在 `user/sleep.c` 文件中实现。

#### 实现步骤

- 包括必要的头文件** 需要包含 `kernel/types.h`、`kernel/stat.h` 和 `user/user.h` 头文件以便使用 xv6 提供的类型和系统调用。
- 主程序结构** 主程序 `main` 应接受命令行参数并进行基本的错误检查。如果用户未指定暂停时间，程序应输出使用说明并退出。
- 时间参数转换** 从命令行参数获取的时间是字符串形式的，需要使用 `atoi` 函数将其转换为整数形式。

4. **调用系统调用** 使用 `sleep` 系统调用将程序暂停指定的 ticks 数。ticks 是 xv6 内核中定义的时间单位，代表定时器芯片两次中断之间的时间间隔。
5. **程序退出** 程序执行完毕后，应使用 `exit(0)` 进行退出，而不是 `return 0`，以确保在 xv6 环境下正确地退出程序。

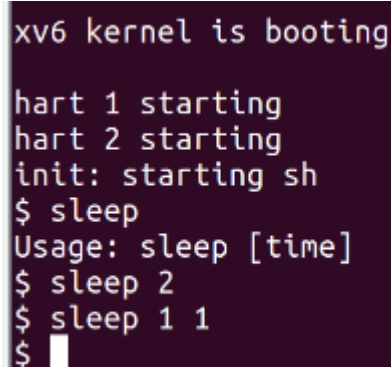
```
1 #include "kernel/types.h"
2 #include "kernel/stat.h"
3 #include "user/user.h"
4
5 int main(int argc, char *argv[]) {
6     // 检查是否提供了足够的命令行参数
7     if (argc < 2) {
8         fprintf(2, "Usage: sleep [time]\n"); // 提示正确的使用方法
9         exit(1); // 退出程序，返回错误状态码
10    }
11
12    // 将第一个命令行参数转换为整数，表示需要睡眠的时间
13    int sleep_time = atoi(argv[1]);
14
15    // 调用sleep函数，使程序暂停指定的时间
16    sleep(sleep_time);
17
18    // 程序正常退出，返回状态码0
19    exit(0);
20 }
21
```

## 说明

在 Unix 系统中，默认情况下 0 代表标准输入 (stdin)，1 代表标准输出 (stdout)，2 代表标准错误输出 (stderr)。这三个文件描述符在进程创建时已经打开，可以直接使用。在分配新的文件描述符时，系统会选择当前未使用的最小值。因此，可以通过关闭某个文件描述符并使用 `open` 或 `dup` 将该文件描述符重新分配给其他文件或管道，实现输入输出重定向。

需要注意的是，程序执行结束后应使用 `exit(0)` 来退出，而不是 `return 0`，以确保在 xv6 环境下正确地退出程序。

## 实验现象



```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ sleep
Usage: sleep [time]
$ sleep 2
$ sleep 1 1
$
```



# pingpong

## 实验目的

1. 实现一个简单的 `pingpong` 程序，使用管道（pipe）在父进程和子进程之间传递消息，以模拟 pingpong 通信。
2. 熟悉 xv6 操作系统中的进程创建和进程间通信机制。
3. 所有代码应在 `user/pingpong.c` 文件中实现。

## 实现步骤

1. **包括必要的头文件** 需要包含 `kernel/types.h`、`kernel/stat.h` 和 `user/user.h` 头文件以便使用 xv6 提供的类型和系统调用。
2. **创建管道** 使用 `pipe` 系统调用创建两个管道：一个用于父进程向子进程发送消息（p2c），另一个用于子进程向父进程发送消息（c2p）。若创建管道失败，应输出错误信息并退出程序。
3. **创建子进程** 使用 `fork` 系统调用创建一个子进程。`fork` 在父进程中返回子进程的 PID，在子进程中返回 0。
4. **进程间通信**
  - **子进程**：从父进程读取消息（通过管道 p2c），输出收到的消息并通过管道 c2p 向父进程发送回复消息。
  - **父进程**：向子进程发送消息（通过管道 p2c），从子进程读取回复消息（通过管道 c2p），并输出收到的回复消息。
5. **关闭管道** 进程在完成通信后，应关闭所有管道文件描述符以释放资源。
6. **程序退出** 程序执行完毕后，应使用 `exit(0)` 正常退出程序。

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int
6  main(int argc, char *argv[]){
7      int parentToChild[2]; // 父进程到子进程的管道
8      int childToParent[2]; // 子进程到父进程的管道
9
10     if(pipe(parentToChild) < 0){ // 创建父到子的管道
11         printf("pipe error");
12         exit(-1); // 如果创建管道失败，退出程序
13     }
14
15     if(pipe(childToParent) < 0){ // 创建子到父的管道
16         printf("pipe error");
17         exit(-1); // 如果创建管道失败，退出程序
18     }
19
20     int pid = fork(); // 创建子进程
21
22     if(pid == 0){
23         // 子进程
24         char readBuffer[10];
```

```

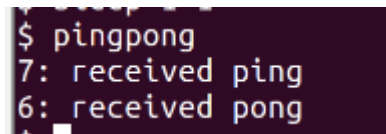
25     read(parentToChild[0], readBuffer, sizeof(readBuffer)); // 从父进程读
    取数据
26     printf("%d: received ping\n", getpid()); // 打印子进程收到的数据
27     write(childToParent[1], "o", 2); // 向父进程写入数据
28 }else if(pid > 0){
29     // 父进程
30     write(parentToChild[1], "p", 2); // 向子进程写入数据
31     char readBuffer[10];
32     read(childToParent[0], readBuffer, sizeof(readBuffer)); // 从子进程读
    取数据
33     printf("%d: received pong\n", getpid()); // 打印父进程收到的数据
34 }
35
36 close(parentToChild[0]); // 关闭父到子管道的读端
37 close(parentToChild[1]); // 关闭父到子管道的写端
38 close(childToParent[0]); // 关闭子到父管道的读端
39 close(childToParent[1]); // 关闭子到父管道的写端
40
41 exit(0); // 程序退出
42 }
43

```

## 说明

在这个实验中，`pipe` 系统调用用于创建进程间通信的管道，`fork` 系统调用用于创建子进程。父进程和子进程通过管道发送和接收消息，实现简单的 pingpong 通信机制。父进程首先向子进程发送 "ping" 消息，子进程收到后输出 "received ping"，然后向父进程发送 "pong" 消息，父进程收到后输出 "received pong"。

## 实验现象



```

$ pingpong
7: received ping
6: received pong

```

## primes

### 实验目标

1. 实现由管道发明者 Doug McIlroy 提出的素数计算方法，该方法类似于筛法，但使用管道和多进程实现。
2. 熟悉 xv6 操作系统中的进程创建、进程间通信和管道的高级用法。
3. 所有代码应在 `user/primes.c` 文件中实现。

### 实现步骤

1. **包括必要的头文件** 需要包含 `kernel/types.h`、`kernel/stat.h` 和 `user/user.h` 头文件，以便使用 xv6 提供的类型和系统调用。
2. **创建管道** 在 `main` 函数中，父进程创建一个管道（pipe），用于传递整数。
3. **父进程写入整数** 父进程在管道中写入从 2 到 35 的整数，并关闭写端。
4. **创建子进程** 父进程通过 `fork` 创建一个子进程，并让子进程调用 `prime` 函数处理管道中的数据。
5. **实现 prime 函数**

- **读取初始素数**: 子进程从管道中读取第一个素数, 并输出该素数。
  - **创建新管道和子进程**: 若读取到的数字超过一个, 子进程创建一个新管道, 并 fork 出一个新的子进程处理后续的素数计算。
  - **筛选非倍数**: 子进程在读取到新数字后, 若该数字不是已读取素数的倍数, 则将其写入新管道。
6. **递归处理** 每个子进程按照上述步骤继续创建新的子进程和管道, 直至所有数字处理完毕。
  7. **关闭管道** 在处理完管道中的所有数据后, 子进程关闭管道的读写端, 并等待所有子进程结束。
  8. **程序退出** 父进程和所有子进程在完成任务后, 均应调用 `exit(0)` 正常退出程序。

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  void primeFilter(int readDescriptor){
6      int primeNumber;
7      read(readDescriptor, &primeNumber, sizeof(primeNumber)); // 读取第一个素数
8      printf("prime %d\n", primeNumber);
9
10     int isPipeCreated = 0;
11     int pipeDescriptors[2];
12     int currentNumber;
13
14     while(read(readDescriptor, &currentNumber, sizeof(currentNumber)) != 0){
15         if(isPipeCreated == 0){
16             pipe(pipeDescriptors); // 创建新的管道
17             isPipeCreated = 1;
18             int pid = fork();
19             if(pid == 0){
20                 close(pipeDescriptors[1]); // 关闭写端
21                 primeFilter(pipeDescriptors[0]); // 递归调用
22                 return;
23             }else{
24                 close(pipeDescriptors[0]); // 关闭读端
25             }
26         }
27         if(currentNumber % primeNumber != 0){
28             write(pipeDescriptors[1], &currentNumber, sizeof(currentNumber));
29             // 将非倍数写入管道
30         }
31
32         close(readDescriptor); // 关闭读端
33         close(pipeDescriptors[1]); // 关闭写端
34         wait(0); // 等待子进程结束
35     }
36
37     int
38     main(int argc, char *argv[]){
39         int initialPipe[2];
40         pipe(initialPipe);
41
42         int pid = fork();
43         if(pid != 0){
```

```

44     // 父进程
45     close(initialPipe[0]); // 关闭读端
46     for(int i = 2; i <= 35; i++){
47         write(initialPipe[1], &i, sizeof(i)); // 写入数值
48     }
49     close(initialPipe[1]); // 关闭写端
50     wait(0); // 等待子进程结束
51 }else{
52     // 子进程
53     close(initialPipe[1]); // 关闭写端
54     primeFilter(initialPipe[0]); // 调用过滤函数
55     close(initialPipe[0]); // 关闭读端
56 }
57
58 exit(0); // 程序退出
59 }
60

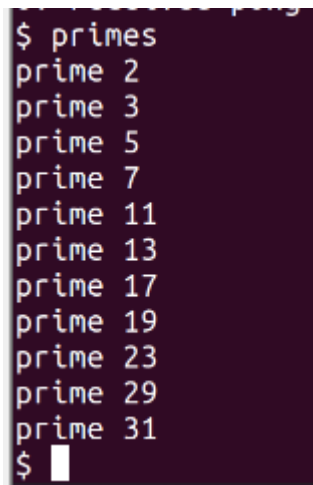
```

## 说明

该实验通过管道和多进程的结合，实现了一个计算素数的程序。主要思想是每个进程处理一个素数，并通过管道将非倍数的数字传递给下一个进程，以递归的方式实现筛法。

1. **管道**：用于在父进程和子进程之间传递数据。
2. **进程创建**：使用 `fork` 系统调用创建子进程，并通过管道进行进程间通信。
3. **递归处理**：每个子进程负责筛选一个素数，并将非倍数传递给下一个子进程，直至所有数字处理完毕。

## 实验现象



```

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

```

## find

### 实验目标

1. 实现一个类似于 Unix `find` 命令的程序，能够在指定目录及其子目录中递归查找匹配指定文件名的文件。
2. 熟悉 xv6 操作系统中的文件系统接口和目录遍历机制。
3. 所有代码应在 `user/find.c` 文件中实现。

## 实现步骤

1. **包括必要的头文件** 需要包含 `kernel/types.h`、`kernel/stat.h`、`kernel/fs.h` 和 `user/user.h` 头文件，以便使用 xv6 提供的类型、系统调用和文件系统结构。
2. **实现子串匹配函数** 由于 xv6 库中没有提供 `strstr` 函数，需要自行实现一个  $O(n^2)$  的子串匹配算法，用于匹配文件路径中的文件名。
3. **实现目录递归查找函数**
  - **打开目录**：使用 `open` 系统调用打开指定目录，若打开失败则输出错误信息并返回。
  - **获取目录信息**：使用 `fstat` 系统调用获取目录的文件状态信息，若获取失败则关闭文件并返回。
  - 处理文件和目录
    - ：
    - 若当前路径为文件，则调用子串匹配函数进行匹配，并输出匹配结果。
    - 若当前路径为目录，则递归遍历目录中的所有文件和子目录，排除 `.` 和 `..` 目录，构造新的路径，并递归调用查找函数。
4. **在 main 函数中调用查找函数**
  - 检查命令行参数，确保用户输入了路径和文件名。
  - 调用查找函数对指定路径进行递归查找。

```
1 #include "kernel/types.h"
2 #include "kernel/stat.h"
3 #include "user/user.h"
4 #include "kernel/fs.h"
5
6 // 匹配路径中的文件名
7 void match(const char* path, const char* name) {
8     int pathIndex = 0;
9     int nameIndex = 0;
10
11     // 遍历路径字符串
12     while (path[pathIndex] != 0) {
13         nameIndex = 0;
14         int tempIndex = pathIndex;
15
16         // 检查子字符串是否与name匹配
17         while (name[nameIndex] != 0) {
18             if (name[nameIndex] == path[tempIndex]) {
19                 nameIndex++;
20                 tempIndex++;
21             } else {
22                 break;
23             }
24         }
25
26         // 如果完整匹配name，则打印路径
27         if (name[nameIndex] == 0) {
28             printf("%s\n", path);
29             return;
30         }
31         pathIndex++;
```

```

32     }
33 }
34
35 // 递归查找路径中的文件
36 void find(const char *path, const char *name) {
37     char buffer[512], *ptr;
38     int fileDescriptor;
39     struct dirent directoryEntry;
40     struct stat status;
41
42     // 打开目录
43     if ((fileDescriptor = open(path, 0)) < 0) {
44         fprintf(2, "无法打开 %s\n", path);
45         return;
46     }
47
48     // 获取目录信息
49     if (fstat(fileDescriptor, &status) < 0) {
50         fprintf(2, "无法获取状态 %s\n", path);
51         close(fileDescriptor);
52         return;
53     }
54
55     // 根据文件类型处理
56     switch (status.type) {
57         case T_FILE:
58             // 如果是文件, 尝试匹配
59             match(path, name);
60             break;
61
62         case T_DIR:
63             // 如果是目录, 递归查找
64             if (strlen(path) + 1 + DIRSIZ + 1 > sizeof(buffer)) {
65                 printf("路径过长\n");
66                 break;
67             }
68
69             strcpy(buffer, path);
70             ptr = buffer + strlen(buffer);
71             *ptr++ = '/';
72
73             // 读取目录内容
74             while (read(fileDescriptor, &directoryEntry,
75 sizeof(directoryEntry)) == sizeof(directoryEntry)) {
76                 if (directoryEntry.inum == 0)
77                     continue;
78
79                 // 跳过 "." 和 ".."
80                 if (directoryEntry.name[0] == '.' && directoryEntry.name[1]
81 == 0) continue;
82                 if (directoryEntry.name[0] == '.' && directoryEntry.name[1]
83 == '.' && directoryEntry.name[2] == 0) continue;
84
85                 memmove(ptr, directoryEntry.name, DIRSIZ);
86                 ptr[DIRSIZ] = 0;

```

```

85         // 获取目录项状态
86         if (stat(buffer, &status) < 0) {
87             printf("无法获取状态 %s\n", buffer);
88             continue;
89         }
90
91         // 递归查找
92         find(buffer, name);
93     }
94     break;
95 }
96
97 close(fileDescriptor); // 关闭目录
98 }
99
100 int main(int argc, char *argv[]) {
101     // 检查参数数量
102     if (argc < 3) {
103         printf("用法: find [路径] [文件名]\n");
104         exit(1);
105     }
106
107     // 调用查找函数
108     find(argv[1], argv[2]);
109     exit(0); // 正常退出程序
110 }
111

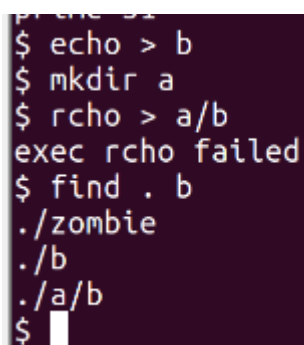
```

## 说明

该实验通过递归遍历目录和匹配文件名，实现了一个类似于 Unix `find` 命令的程序。主要步骤包括打开目录、获取目录信息、处理文件和目录、递归遍历子目录等。

1. **文件系统接口**：使用 `open`、`fstat`、`read` 和 `stat` 等系统调用实现目录和文件的操作。
2. **子串匹配**：自行实现子串匹配算法，用于匹配文件路径中的文件名。
3. **递归遍历**：通过递归遍历目录，实现对所有子目录和文件的查找。

## 实验现象



```

$ echo > b
$ mkdir a
$ rcho > a/b
exec rcho failed
$ find . b
./zombie
./b
./a/b
$

```

# xargs

## 实验目标

1. 实现一个类似于 Unix `xargs` 命令的程序，该程序从标准输入读取参数并构造新的 `argc` 数组，然后使用 `fork` 和 `exec` 执行指定的命令。
2. 熟悉 xv6 操作系统中的进程创建、进程间通信、标准输入读取和命令执行机制。
3. 所有代码应在 `user/xargs.c` 文件中实现。

## 实现步骤

1. **包括必要的头文件** 需要包含 `kernel/types.h`、`kernel/stat.h` 和 `user/user.h` 头文件，以便使用 xv6 提供的类型和系统调用。
2. **实现读取一行输入的函数** 由于 xv6 库中没有提供 `readline` 函数，需要自行实现一个函数，从标准输入读取一行文本并返回。
3. **解析输入** 自行实现将输入行拆分为单词的函数，并将其添加到命令参数数组中。
4. **构造新的 `argc` 数组** 从命令行参数和标准输入中读取的参数构造新的 `argc` 数组，用于传递给 `exec` 调用。
5. **创建子进程并执行命令**
  - 使用 `fork` 创建子进程。
  - 在子进程中使用 `exec` 执行命令，并传递构造好的 `argc` 数组。
  - 在父进程中使用 `wait` 等待子进程结束。
6. **内存管理** 注意对动态分配内存的管理，避免内存泄漏。

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  // 读取一行输入
6  char* readline() {
7      char* buffer = malloc(100);
8      char* p = buffer;
9
10     // 从标准输入读取字符，直到遇到换行符或字符串结束符
11     while (read(0, p, 1) != 0) {
12         if (*p == '\n' || *p == '\0') {
13             *p = '\0'; // 将换行符或字符串结束符替换为字符串结束符
14             return buffer;
15         }
16         p++;
17     }
18
19     // 如果缓冲区非空，返回缓冲区指针，否则释放内存
20     if (p != buffer) return buffer;
21     free(buffer);
22     return 0;
23 }
24
25 int main(int argc, char *argv[]) {
26     // 检查参数数量
```



```

27     if (argc < 2) {
28         printf("用法: xargs [命令]\n");
29         exit(1);
30     }
31
32     char* line;
33     argv++; // 跳过程序名参数
34     char* new_argv[16]; // 新的参数列表
35     char** new_argv_ptr = new_argv;
36     char** argv_ptr = argv;
37
38     // 将命令行参数复制到新的参数列表
39     while (*argv_ptr != 0) {
40         *new_argv_ptr = *argv_ptr;
41         new_argv_ptr++;
42         argv_ptr++;
43     }
44
45     // 读取每行输入
46     while ((line = readline()) != 0) {
47         char* p = line;
48         char* buffer = malloc(36);
49         char* buffer_head = buffer;
50         int new_argc = argc - 1;
51
52         // 解析输入行，将空格分隔的部分作为新的参数添加到参数列表中
53         while (*p != 0) {
54             if (*p == ' ' && buffer != buffer_head) {
55                 *buffer_head = '\0';
56                 new_argv[new_argc] = buffer;
57                 buffer = malloc(36);
58                 buffer_head = buffer;
59                 new_argc++;
60             } else {
61                 *buffer_head = *p;
62                 buffer_head++;
63             }
64             p++;
65         }
66
67         // 添加最后一个参数
68         if (buffer != buffer_head) {
69             new_argv[new_argc] = buffer;
70             new_argc++;
71         }
72         new_argv[new_argc] = 0;
73         free(line);
74
75         // 创建子进程并执行命令
76         int pid = fork();
77         if (pid == 0) {
78             exec(new_argv[0], new_argv);
79         } else {
80             wait(0); // 等待子进程结束
81         }
82     }

```

```
83     exit(0); // 正常退出程序
84 }
85
```

## 说明

该实验通过读取标准输入并构造新的命令行参数，实现了一个类似于 Unix `xargs` 命令的程序。主要步骤包括从标准输入读取输入行、解析输入、构造新的 `argc` 数组、创建子进程并执行命令。

1. **读取标准输入**：实现了一个 `getline` 函数，从标准输入读取一行文本并返回。
2. **解析输入**：将读取的输入行拆分为单词，并添加到命令参数数组中。
3. **进程创建和命令执行**：使用 `fork` 创建子进程，并在子进程中使用 `exec` 执行命令。父进程使用 `wait` 等待子进程结束。

## 实验现象

```
$ sh < xargstest.sh
$ mkdir: a failed to create
$ $ $ $ $ hello
hello
hello
```

## 实验结果

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (13.2s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (5.3s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (4.2s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (4.5s)
== Test primes ==
$ make qemu-gdb
primes: OK (2.8s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (2.9s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (5.2s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (5.5s)
== Test time ==
time: OK
Score: 100/100
(base) sheerio@ubuntu:~/xv6/lab1 util/xv6-labs-2021$
```

# 实验中遇到的问题及解决办法

## 1. 问题：在 `primes` 实验中，多进程的管理和管道使用出现错误。

- **描述：**在构建多进程进行质数筛选时，由于对管道使用不当，导致数据传递不正确，进程间的数据流失。
- **解决办法：**重新设计进程间的管道连接逻辑，确保每个进程正确关闭不需要的管道端点。此外，精确控制每个进程何时读取或写入管道，避免数据混乱。

## 2. 问题：`find` 命令中的递归搜索性能问题。

- **描述：**在实现文件系统递归搜索功能时，遇到了效率低下和处理大量文件时的性能瓶颈。
- **解决办法：**优化递归函数的实现，减少不必要的文件系统调用，并尝试减少递归深度，通过非递归方式改进搜索算法。

## 3. 问题：`xargs` 命令中参数处理和命令执行的复杂性。

- **描述：**在解析输入并转换为命令行参数时，处理引号和特殊字符带来了挑战。
- **解决办法：**编写更为精确的解析函数，以正确处理各种输入情况，包括带空格和特殊字符的参数。此外，确保生成的命令行正确地被子进程执行。

# 实验心得

在本次 Xv6 util 实验中，我实现了多个基于 Xv6 操作系统的实用程序，包括 `sleep`、`pingpong`、`primes` 和 `xargs`。每个程序都有其独特的挑战和学习价值，通过这些实验，我对操作系统的基本概念和系统调用机制有了更加深入的理解。

首先，实现 `sleep` 程序让我理解了如何在 Xv6 中使用系统调用来暂停进程。通过实现这个简单的功能，我熟悉了 Xv6 的时间管理机制，以及如何处理用户输入和错误。这个实验的一个关键点是正确使用 `fork` 和 `exec` 系统调用，以及在程序结束时使用 `exit` 来确保进程正确终止。

其次，在 `pingpong` 实验中，我学会了如何使用管道（pipe）实现进程间通信。通过创建两个管道并使用 `fork` 创建子进程，实现了父子进程之间的消息传递。这让我理解了管道在进程间通信中的重要性，以及如何通过 `read` 和 `write` 系统调用在管道中传递数据。同时，我也体会到了进程同步和资源管理的重要性。

在 `primes` 实验中，我实现了 Doug McIlroy 提出的使用管道和多进程计算素数的方法。这个实验类似于筛法，但通过管道和多进程实现。实现过程中，我学会了如何递归地创建子进程，并通过管道传递数据，从而筛选出素数。这个实验加深了我对进程创建、进程间通信以及递归处理的理解。

最后，在实现 `xargs` 程序时，我面临的主要挑战是处理标准输入并构造新的命令参数数组。由于 Xv6 没有提供现成的 `getline` 和 `split` 函数，我需要自己实现这些功能。这涉及到字符处理、字符串解析和动态内存管理。通过这个实验，我学会了如何从标准输入读取数据并解析输入行，将其转换为命令参数，并通过 `fork` 和 `exec` 系统调用执行命令。这大大提升了我处理字符串和管理内存的能力。

总的来说，这次 Xv6 util 实验让我全面了解了操作系统的基本机制，包括进程管理、进程间通信、系统调用和输入输出处理。每个实验都提供了独特的挑战，通过解决这些问题，我不仅巩固了理论知识，还提升了实际编程能力和调试技巧。特别是对动态内存管理、字符串处理和进程同步等方面的理解，使我在面对实际问题时更加得心应手。

# lab2 system calls

## 实验总览

在本实验中，将通过添加新的系统调用来深入理解xv6内核的工作机制及其内部特性，掌握系统调用的创建和管理过程。实验代码涉及的主要文件包括用户空间的 `user/user.h` 和 `user/usys.pl`，以及内核空间的 `kernel/syscall.h`、`kernel/syscall.c`、`kernel/proc.h` 和 `kernel/proc.c`。

```
1 | $ git checkout syscall
```

## System call tracing

### 实验目的

本实验旨在理解系统调用跟踪的原理和实现方法，以及如何通过添加新功能来修改操作系统，核心任务是创建一个名为 `trace` 的新系统调用，该调用接受一个整数参数“mask”。这个“mask”参数通过其位设置，指定了需要跟踪的系统调用，可以按需开启或关闭特定系统调用的跟踪。通过这个实验可以深入学习内核级编程，包括如何修改进程的数据结构、处理系统调用逻辑，以及管理跟踪掩码，增强对操作系统核心工作机制的理解。

### 实验过程

#### 1. 定义 `trace` 系统调用：

- 在 `user/user.h` 中添加 `trace` 系统调用的原型。

```
1  // system calls
2  int fork(void);
3  int exit(int) __attribute__((noreturn));
4  int wait(int*);
5  int pipe(int*);
6  int write(int, const void*, int);
7  int read(int, void*, int);
8  int close(int);
9  int kill(int);
10 int exec(char*, char**);
11 int open(const char*, int);
12 int mknod(const char*, short, short);
13 int unlink(const char*);
14 int fstat(int fd, struct stat*);
15 int link(const char*, const char*);
16 int mkdir(const char*);
17 int chdir(const char*);
18 int dup(int);
19 int getpid(void);
20 char* sbrk(int);
21 int sleep(int);
22 int uptime(void);
23 int trace(int);
```

- 在 `user/usys.pl` 脚本中添加 `trace` 对应的 entry，以自动生成系统调用的相关代码。

```

1  entry("fork");
2  entry("exit");
3  entry("wait");
4  entry("pipe");
5  entry("read");
6  entry("write");
7  entry("close");
8  entry("kill");
9  entry("exec");
10 entry("open");
11 entry("mknod");
12 entry("unlink");
13 entry("fstat");
14 entry("link");
15 entry("mkdir");
16 entry("chdir");
17 entry("dup");
18 entry("getpid");
19 entry("sbrk");
20 entry("sleep");
21 entry("uptime");
22 entry("trace");

```

- 在 `kernel/syscall.h` 中定义 `trace` 的系统调用号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

```

## 2. 编写系统调用函数:

- 在 `kernel/sysproc.c` 中实现 `sys_trace()` 函数。该函数使用 `argint(0, &mask)` 获取系统调用参数, 并将其存储在当前进程的 `proc` 结构中的新变量 `tracemask` 中。

```

1 // 从用户空间获取参数，并将其设置为进程的trace_mask
2 uint64
3 set_trace_mask(void) {
4     int traceMask;
5     // 获取追踪的mask
6     if (argint(0, &traceMask) < 0) {
7         return -1; // 如果获取参数失败，返回-1
8     }
9
10    // 将mask保存在当前进程的proc结构体中
11    struct proc *currentProc = myproc();
12    printf("trace pid: %d\n", currentProc->pid);
13    currentProc->trace_mask = traceMask;
14
15    return 0; // 成功设置后返回0
16 }

```

这个函数 `set_trace_mask` 的作用是从用户空间获取一个参数，并将其设置为当前进程的 `trace_mask`，从而实现系统调用的追踪。当在用户态调用 `trace()` 时，会传入一个 `mask` 值，这个系统调用的核心就是将这个传入的 `mask` 赋值给当前进程的 `struct proc` 结构体中的 `trace_mask` 字段。通过调用 `argint(0, &traceMask)` 来读取传入的第一个32位参数，这种方式的使用是因为内核与用户进程的页表不同，因此需要使用 `argaddr()`、`argint()`、`argstr()` 等系列函数来进行参数读取。函数首先声明并初始化一个 `int` 类型的变量 `traceMask` 用于存储从用户空间获取的参数，如果 `argint` 获取参数失败，则返回 `-1`。接着，通过 `myproc()` 函数获取当前进程的指针，并将 `traceMask` 的值赋给当前进程的 `trace_mask` 字段，最后打印当前进程的 PID 并返回 `0` 表示成功设置。

### 3. 修改进程管理相关代码：

- 修改 `kernel/proc.c` 中的 `fork()` 函数，确保父进程的跟踪掩码能被复制到子进程中。

```

1 // Create a new process, copying the parent.
2 // Sets up child kernel stack to return as if from fork() system
  call.
3 int
4 fork(void)
5 {
6     int i, pid;
7     struct proc *np;
8     struct proc *p = myproc();
9
10    // Allocate process.
11    if((np = allocproc()) == 0){
12        return -1;
13    }
14
15    // Copy user memory from parent to child.
16    if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
17        freeproc(np);
18        release(&np->lock);
19        return -1;
20    }
21    np->sz = p->sz;

```

```

22
23     np->parent = p;
24
25     // copy saved user registers.
26     *(np->trapframe) = *(p->trapframe);
27
28     // Cause fork to return 0 in the child.
29     np->trapframe->a0 = 0;
30
31     // increment reference counts on open file descriptors.
32     for(i = 0; i < NOFILE; i++)
33         if(p->ofile[i])
34             np->ofile[i] = filedup(p->ofile[i]);
35     np->cwd = idup(p->cwd);
36
37     safestrcpy(np->name, p->name, sizeof(p->name));
38
39     pid = np->pid;
40
41     np->state = RUNNABLE;
42
43     // trace
44     np->trace_mask = p->trace_mask;
45
46     release(&np->lock);
47
48     return pid;
49 }

```

#### 4. 修改系统调用处理逻辑:

- 在 kernel/syscall.c 中修改 syscall() 函数, 根据跟踪掩码打印相关的系统调用跟踪信息。

```

1 void
2 syscall(void)
3 {
4     int syscall_id;
5     struct proc *current_process = myproc(); // 获取当前进程的进程控制块
6
7     // 系统调用号存放在寄存器a7中
8     syscall_id = current_process->trapframe->a7;
9     if(syscall_id > 0 && syscall_id < NELEM(syscalls) &&
10        syscalls[syscall_id])
11     {
12         // 返回值保存在寄存器a0中
13         current_process->trapframe->a0 = syscalls[syscall_id]();
14         // 检查trace_mask是否包含当前调用号
15         if(current_process->trace_mask & (1 << syscall_id))
16         {
17             printf("%d: syscall %s -> %d\n", current_process->pid,
18                syscall_name[syscall_id], current_process->trapframe->a0);
19         }
20     }
21 }

```

```

19     else
20     {
21         printf("%d %s: unknown sys call %d\n",
22             current_process->pid, current_process->name,
23             syscall_id);
24         current_process->trapframe->a0 = -1;
25     }
26 }

```

- 更新 `syscalls` 数组和 `syscalls_name` 数组，添加对 `sys_trace()` 函数的引用。

```

1  static uint64 (*syscalls[])(void) = {
2  [SYS_fork]      sys_fork,
3  [SYS_exit]      sys_exit,
4  [SYS_wait]      sys_wait,
5  [SYS_pipe]      sys_pipe,
6  [SYS_read]      sys_read,
7  [SYS_kill]      sys_kill,
8  [SYS_exec]      sys_exec,
9  [SYS_fstat]     sys_fstat,
10 [SYS_chdir]     sys_chdir,
11 [SYS_dup]       sys_dup,
12 [SYS_getpid]    sys_getpid,
13 [SYS_sbrk]      sys_sbrk,
14 [SYS_sleep]     sys_sleep,
15 [SYS_uptime]    sys_uptime,
16 [SYS_open]      sys_open,
17 [SYS_write]     sys_write,
18 [SYS_mknod]     sys_mknod,
19 [SYS_unlink]    sys_unlink,
20 [SYS_link]      sys_link,
21 [SYS_mkdir]     sys_mkdir,
22 [SYS_close]     sys_close,
23 [SYS_trace]     sys_trace,
24 };

```

```

1  extern uint64 sys_trace(void);

```

## 5. 修改 Makefile:

- 在 `Makefile` 的 `UPROGS` 部分添加 `$U/_trace`，确保新的用户程序 `trace` 能被正确编译和链接。

## 6. 进行测试:



```
(base) sheerio@ubuntu:~/xv6/lab2 syscall/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 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
trace pid: 3
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$ trace 2147483647 grep hello README
trace pid: 4
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
trace pid: 6
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
9: syscall fork -> 11
9: syscall fork -> 12
9: syscall fork -> 14
10: syscall fork -> 13
9: syscall fork -> 15
9: syscall fork -> 16
10: syscall fork -> 17
10: syscall fork -> 19
9: syscall fork -> 18
10: syscall fork -> 20
9: syscall fork -> 21
9: syscall fork -> 22
```

```
9: syscall fork -> 30
11: syscall fork -> 31
11: syscall fork -> 32
11: syscall fork -> 33
9: syscall fork -> 34
10: syscall fork -> 35
9: syscall fork -> 36
9: syscall fork -> 37
9: syscall fork -> 38
9: syscall fork -> 39
9: syscall fork -> 40
9: syscall fork -> 41
9: syscall fork -> 42
9: syscall fork -> 43
9: syscall fork -> 44
9: syscall fork -> 45
10: syscall fork -> 46
10: syscall fork -> 47
10: syscall fork -> 48
10: syscall fork -> 49
10: syscall fork -> 50
10: syscall fork -> 51
10: syscall fork -> 52
10: syscall fork -> 53
10: syscall fork -> 54
10: syscall fork -> 55
9: syscall fork -> 56
10: syscall fork -> 57
56: syscall fork -> 58
56: syscall fork -> 59
9: syscall fork -> 60
10: syscall fork -> 61
11: syscall fork -> 62
11: syscall fork -> 63
11: syscall fork -> 64
9: syscall fork -> 65
9: syscall fork -> 66
9: syscall fork -> 67
9: syscall fork -> 68
10: syscall fork -> -1
9: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED
$
```

## Sysinfo

### 实验目的

本次实验的目标是为 xv6 操作系统实现一个新的系统调用 `sysinfo`，用于收集当前系统的空闲内存和运行进程的数量。该系统调用接收一个指向 `struct sysinfo` 的指针，并在系统调用中将收集到的信息写入该结构体。通过此实验，您将学习如何在 xv6 中添加和实现系统调用，包括定义新的数据结构、修改内核代码以支持新系统调用，以及从内核中读取和操作相关系统数据，以便提供准确的系统信息。

### 实验过程

#### 1. 在 user/user.h 中声明 sysinfo

在 `user/user.h` 中声明 `sysinfo()` 的原型，并预先声明 `struct sysinfo` 的存在：

```
1 struct stat;
2 struct rtcdate;
3 struct sysinfo;
4
```

```

5 // system calls
6 int fork(void);
7 int exit(int) __attribute__((noreturn));
8 int wait(int*);
9 int pipe(int*);
10 int write(int, const void*, int);
11 int read(int, void*, int);
12 int close(int);
13 int kill(int);
14 int exec(char*, char**);
15 int open(const char*, int);
16 int mknod(const char*, short, short);
17 int unlink(const char*);
18 int fstat(int fd, struct stat*);
19 int link(const char*, const char*);
20 int mkdir(const char*);
21 int chdir(const char*);
22 int dup(int);
23 int getpid(void);
24 char* sbrk(int);
25 int sleep(int);
26 int uptime(void);
27 int trace(int);
28 int sysinfo(struct sysinfo *);

```

## 2. 实现 sysinfo 系统调用

在实现 `sysinfo` 系统调用时，需要将 `struct sysinfo` 复制到用户空间。参考 `sys_fstat()` (`kernel/sysfile.c`) 和 `filestat()` (`kernel/file.c`) 中的实现，使用 `copyout()` 函数完成复制。该函数在 `kernel/vm.c` 中定义。

```

1 int
2 copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
3 {
4     uint64 n, va0, pa0;
5
6     while(len > 0){
7         va0 = PGROUNDDOWN(dstva);
8         pa0 = walkaddr(pagetable, va0);
9         if(pa0 == 0)
10             return -1;
11         n = PGSIZE - (dstva - va0);
12         if(n > len)
13             n = len;
14         memmove((void *) (pa0 + (dstva - va0)), src, n);
15
16         len -= n;
17         src += n;
18         dstva = va0 + PGSIZE;
19     }
20     return 0;
21 }

```

## 3. 在 kernel/kalloc.c 中添加 getfreemem 函数

添加一个 `get_free_mem` 函数，用于收集可用内存的数量。参考 `kalloc()` 和 `kfree()` 等函数，通过遍历 `kmem.freelist` 链表计算空闲内存的字节数：

```

1  uint64
2  get_free_mem(void) // 获取空闲内存大小
3  {
4      struct run *free_node;
5      acquire(&kmem.lock); // 获取锁
6      free_node = kmem.freelist; // 获取空闲链表的头结点
7      int free_pages = 0;
8      while(free_node)
9      {
10         ++free_pages;
11         free_node = free_node->next;
12     }
13     release(&kmem.lock); // 释放锁
14     return free_pages * PGSIZE; // 计算并返回空闲内存大小
15 }

```

#### 4. 在 kernel/proc.c 中添加 getnproc 函数

添加一个 `get_proc_num` 函数，用于收集运行进程的数量。参考 `allocproc()` 和 `freeproc()` 等函数，通过遍历 `proc[NPROC]` 数组计算当前运行的进程数：

```

1  int
2  get_proc_num(void) // 获取非空闲进程数量
3  {
4      struct proc *process;
5      int active_proc_count = 0;
6      for(process = proc; process < &proc[NPROC]; process++) {
7          acquire(&process->lock);
8          if(process->state != UNUSED) { // 状态为UNUSED表示进程未被使用
9              ++active_proc_count;
10             }
11             release(&process->lock);
12         }
13         return active_proc_count;
14     }
15 }

```

#### 5. 在 kernel/defs.h 中添加函数原型

在 `kernel/defs.h` 中添加 `getfreemem` 和 `getnproc` 函数的原型声明：

```

1  int get_free_mem(void);
2  int get_proc_num(void);

```

#### 6. 实现 sysinfo 系统调用

在 `kernel/sysproc.c` 中实现 `sysinfo` 系统调用，调用 `getfreemem` 和 `getnproc` 函数获取系统信息，并将信息写入用户提供的 `struct sysinfo` 中：

```

1  uint64
2  sys_info(void)
3  {
4      uint64 addr;
5      if(argaddr(0, &addr) < 0) // 获取用户空间中第一个参数的地址
6          return -1;
7      struct sysinfo info;

```

```

8      info.freemem = get_free_mem();//获取系统信息（空闲内存大小、进程数目和可用的文件描述符数目）
9      info.nproc = get_proc_num();
10
11     //copyout 参数：进程页表，用户态目标地址，数据源地址，数据大小 返回值：数据大小
12     //将系统的状态信息返回给用户空间，以便用户可以方便地获取这些信息
13     if(copyout(myproc()->pagetable, addr, (char *)&info, sizeof(info)) < 0)
14         return -1;
15
16     return 0;
17 }

```

## 7. 在 Makefile 中添加目标

在 Makefile 的 UPROGS 中添加 `$U/_sysinfotest`，以便将测试程序包含在构建过程中。

## 8. 测试程序

```

$ sysinfotest
sysinfotest: start
sysinfotest: OK
$ S

```

## 实验结果

```

== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (6.4s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.0s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.1s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (17.2s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (3.6s)
== Test time ==
time: OK
Score: 35/35
(base) sheerio@ubuntu:~/xv6/lab2 syscall/xv6-labs-2021$

```

## 实验中遇到的问题及解决办法

### 1. 系统调用跟踪掩码：

- **问题：**最初，我在实现 trace 系统调用时，跟踪掩码无法正确过滤出指定的系统调用。这是因为在 `syscall()` 函数中对掩码的判断逻辑有误。
- **解决办法：**通过重新审查 xv6 的系统调用处理流程，我发现错误在于位掩码的应用方式。我修正了条件检查语句，确保只有当掩码与系统调用编号对应的位同时为 1 时，才记录该调用。

### 2. 进程信息传递至用户空间的正确性：

- **问题：**在实现 `sysinfo` 系统调用时，初次尝试从内核传递进程信息到用户空间时发生了数据错误，用户程序接收到的数据不正确。
- **解决办法：**问题出在数据复制到用户空间的函数 `copyout()` 使用不当。我通过详细阅读xv6的内存管理文档和检查代码实现，修改了对 `copyout()` 的调用参数，确保正确地从内核空间复制正确的数据量到用户空间。

## 实验总结

在完成xv6操作系统的Lab 2，特别是关于系统调用跟踪（System Call Tracing）和系统信息（Sysinfo）的部分后，我获得了许多关于操作系统内核如何运作以及系统调用如何被实现和管理的深刻见解。

首先，系统调用跟踪的部分使我深入理解了系统调用的内部机制。通过实现一个新的系统调用 `trace`，我学会了如何在内核中添加新的功能，并通过位掩码控制对特定系统调用的跟踪。这个过程不仅涉及到编码技巧，还需要对xv6内核的进程管理和系统调用处理流程有更深入的了解。实验中，我面临的最大挑战是确保跟踪功能正确无误地集成到现有的系统调用框架中。我需要确保新添加的代码不会影响到系统的其他部分，并且能够稳定运行。

对于sysinfo部分，通过添加一个能够返回系统运行信息的系统调用，我学习了如何从内核空间向用户空间传递信息。这不仅提高了我的系统编程能力，还加深了我对操作系统如何管理和监控其资源的理解。实现这一系统调用的过程中，我需要详细了解xv6内核如何管理进程、内存等资源，以便准确地收集和返回所需的信息。

## lab3 Page tables

### 实验总览

该实验是xv6操作系统中的Lab3，主要任务是通过改进系统调用的实现来提高系统性能，并实现打印页表和检测页表访问的功能。实验包括三个主要部分：加速系统调用、打印页表和检测页表访问。实验代码涉及的主要文件包括内核空间的`kernel/vm.c`、`kernel/memlayout.h`和`kernel/kalloc.c`。

```
1 | $ git checkout pagetable
```

### Print a Page Table

#### 实验目的

本实验旨在探索和修改页表，以简化将数据从用户空间复制到内核空间的过程。我们需要编写一个打印页表内容的函数 `vmprint()`，它应当接收一个 `pagetable_t` 作为参数，并以指定的格式打印该页表。打印页表信息有助于我们进行错误检查和调试。

#### 实验步骤

1. **文件准备：**将 `vmprint()` 函数放在 `kernel/vm.c` 中，并将其原型定义在 `kernel/defs.h` 中，以便在 `exec.c` 中调用。

```
1 | // vm.c
2 | void vmprint(pagetable_t);
```

2. **理解页表：**在计算机中，物理地址是内存单元的绝对地址，而虚拟地址是操作系统给用户态应用程序看到的假地址。虚拟地址与物理地址的对应关系通过页表实现。页表通过页号和页内偏移实现虚拟地址到物理地址的映射。

3. **实现 vmprint 函数**: 参考 `freewalk` 函数, 在 `vm.c` 中编写 `vmprint` 函数, 该函数将以指定格式递归打印页表的内容。

```
1 // 递归打印页表
2 void _vmprint(pagetable_t pagetable, int level)
3 {
4     for (int index = 0; index < 512; index++)
5     {
6         pte_t entry = pagetable[index]; // 获取页表项
7         if (entry & PTE_V) // 检查页表项是否有效
8         {
9             uint64 physicalAddr = PTE2PA(entry); // 获取物理地址
10            for (int indent = 0; indent < level; indent++)
11            {
12                if (indent)
13                    printf(" ");
14                printf("..");
15            }
16            // 打印页表项信息
17            printf("%d: pte %p pa %p\n", index, entry, physicalAddr);
18            if ((entry & (PTE_R | PTE_W | PTE_X)) == 0)
19            {
20                // 递归打印下一级页表
21                _vmprint((pagetable_t)physicalAddr, level + 1);
22            }
23        }
24    }
25 }
26
27 // 打印页表入口函数
28 void vmprint(pagetable_t pagetable) {
29     printf("page table %p\n", pagetable); // 打印页表地址
30     _vmprint(pagetable, 1);
31 }
```

4. **插入打印代码**: 在 `exec.c` 的 `exec()` 函数中插入条件语句, 以打印第一个进程的页表。

```
1 // Commit to the user image.
2 oldpagetable = p->pagetable;
3 p->pagetable = pagetable;
4 p->sz = sz;
5 p->trapframe->epc = elf.entry; // initial program counter = main
6 p->trapframe->sp = sp; // initial stack pointer
7 proc_freepagetable(oldpagetable, oldsz);
8 if (p->pid == 1) { vmprint(p->pagetable); }
9 return argc; // this ends up in a0, the first argument to main(argc,
// argv)
```

## 实验现象

通过在 `init` 进程创建时打印页表，我们能够清晰地看到页表的层级结构，以及虚拟地址到物理地址的映射关系。

```
(base) sheerio@ubuntu:~/xv6/lab3 pgtbl/xv6-labs-2021$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087fb5000
..0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
.. ..0: pte 0x0000000021fd9801 pa 0x0000000087f66000
.. .. ..0: pte 0x0000000021fda05f pa 0x0000000087f68000
..255: pte 0x0000000021fed001 pa 0x0000000087fb4000
.. ..511: pte 0x0000000021fecc01 pa 0x0000000087fb3000
.. .. ..510: pte 0x0000000021fed8c7 pa 0x0000000087fb6000
.. .. ..511: pte 0x0000000020001c4b pa 0x0000000080007000
page table 0x0000000087f63000
..0: pte 0x0000000021fd7c01 pa 0x0000000087f5f000
.. ..0: pte 0x0000000021fd7801 pa 0x0000000087f5e000
.. .. ..0: pte 0x0000000021fd801f pa 0x0000000087f60000
.. .. ..1: pte 0x0000000021fd740f pa 0x0000000087f5d000
.. .. ..2: pte 0x0000000021fd701f pa 0x0000000087f5c000
..255: pte 0x0000000021fd8801 pa 0x0000000087f62000
.. ..511: pte 0x0000000021fd8401 pa 0x0000000087f61000
.. .. ..510: pte 0x0000000021fed807 pa 0x0000000087fb6000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$
```

## A Kernel Page Table Per Process

### 实验目的

每个进程都有一个用户地址空间，内核有一个全局的内核地址空间。当前所有进程共享一个全局的内核页表，为了更好地管理和隔离内存，需要为每个进程维护一个独立的内核页表副本。

### 实验步骤

1. 修改 `struct proc`：在 `struct proc` 结构中添加一个新的字段 `pagetable_t kpagetable`，用于存储每个进程的内核页表。

```
1 // Per-process state
2 struct proc {
3     struct spinlock lock;
4     pagetable_t kpagetable; // 内核态页表
5
6     // p->lock must be held when using these:
7     enum procstate state; // Process state
8     struct proc *parent; // Parent process
9     void *chan; // If non-zero, sleeping on chan
10    int killed; // If non-zero, have been killed
```



```

11     int xstate;                // Exit status to be returned to parent's
wait
12     int pid;                   // Process ID
13
14     // these are private to the process, so p->lock need not be held.
15     uint64 kstack;             // Virtual address of kernel stack
16     uint64 sz;                 // Size of process memory (bytes)
17     pagetable_t pagetable;     // User page table
18     struct trapframe *trapframe; // data page for trampoline.s
19     struct context context;     // swtch() here to run process
20     struct file *ofile[NOFILE]; // Open files
21     struct inode *cwd;          // Current directory
22     char name[16];             // Process name (debugging)
23 };
24

```

2. 实现 `ukvmmap` 和 `ukvminit` 函数: 在 `kernel/vm.c` 中添加 `ukvmmap` 和 `ukvminit` 函数, 并在 `defs.h` 中声明。

```

1 // 映射内核页表
2 void ukvmmap(pagetable_t kpagetable, uint64 va, uint64 pa, uint64 sz,
int perm)
3 {
4     if(mappages(kpagetable, va, sz, pa, perm) != 0) // 映射页表条目
5         panic("ukvmmap");
6 }
7
8 // 初始化内核页表
9 pagetable_t ukvminit()
10 {
11     pagetable_t kpagetable = (pagetable_t) kalloc(); // 分配内核页表
12     memset(kpagetable, 0, PGSIZE); // 清空页表内存
13     // 映射内核需要的地址范围
14     ukvmmap(kpagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
15     ukvmmap(kpagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
16     ukvmmap(kpagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
17     ukvmmap(kpagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
18     ukvmmap(kpagetable, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R
| PTE_X);
19     ukvmmap(kpagetable, (uint64)etext, (uint64)etext, PHYSTOP-
(uint64)etext, PTE_R | PTE_W);
20     ukvmmap(kpagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R |
PTE_X);
21     return kpagetable; // 返回内核页表
22 }

```

3. 初始化内核页表: 在 `kernel/proc.c` `allocproc` 函数中初始化内核态页表和内核栈。

```

1 // An empty user kernel page table.
2 p->kpagetable = ukvminit();
3 if (p->kpagetable == 0)
4 {
5     freeproc(p);
6     release(&p->lock);
7     return 0;

```



```

8     }
9
10    // 初始化内核栈
11    char *pa = kalloc(); // 为进程分配一个物理页
12    if (pa == 0)
13        panic("kalloc");
14    uint64 va = KSTACK((int)(p - proc)); // 分配一个虚拟地址
15    ukvmmap(p->kpagetable, va, (uint64)pa, PGSIZE, PTE_R | PTE_W); // 内核态页
    表映射到物理地址
16    p->kstack = va; // 用户空间的内核栈的起始地址

```

4. **切换内核页表**: 在 `kernel/proc.c scheduler()` 函数中, 将当前进程的 `kpagetable` 加载到内核寄存器 `satp`。

```

1  if (p->state == RUNNABLE)
2      {
3          // Switch to chosen process. It is the process's job
4          // to release its lock and then reacquire it
5          // before jumping back to us.
6          p->state = RUNNING;
7          c->proc = p;
8          // 将 p->kpagetable 中的页表设置为当前 CPU 的页表
9          w_satp(MAKE_SATP(p->kpagetable));
10         sfence_vma();
11         swtch(&c->context, &p->context);
12         kvminithart();
13         // Process is done running for now.
14         // It should have changed its p->state before coming back.
15         c->proc = 0;
16
17         found = 1;
18     }

```

5. **释放内核页表**: 在 `kernel/proc.c freeproc` 函数中, 释放进程的内核页表和内核栈。

```

1  // 释放内核页表
2  void free_kernel_pagetable(pagetable_t pagetable)
3  {
4      for(int index = 0; index < 512; ++index) // 遍历页表的所有条目
5      {
6          pte_t entry = pagetable[index]; // 获取页表条目
7          if((entry & PTE_V) && (entry & (PTE_R|PTE_W|PTE_X)) == 0) // 检查条目
            是否有效且是页表目录项
8          {
9              pagetable[index] = 0; // 将页表目录项标记为无效
10             free_kernel_pagetable((pagetable_t)PTE2PA(entry)); // 递归释放子页表
11         }
12         else if(entry & PTE_V) // 如果是普通页表项
13             pagetable[index] = 0; // 标记为无效
14     }
15     kfree((void *)pagetable); // 释放页表所占的内存
16 }
17
18 // 释放进程的内核页表及其对应的内存
19 void proc_free_kernel_pagetable(struct proc *p)

```

```

20 {
21     if (p->kstack) // 检查进程的内核栈是否存在
22     {
23         pte_t *pte_ptr = walk(p->kpagetable, p->kstack, 0); // 获取与内核栈相关
// 的页表项
24         kfree((void *)PTE2PA(*pte_ptr)); // 释放与内核栈相关的物理内存
25         p->kstack = 0; // 将内核栈设置为 0
26     }
27     free_kernel_pagetable(p->kpagetable); // 释放进程的内核页表及其子页表
28 }
29
30 // 释放进程结构体及其相关的数据，包括用户页
31 // 必须持有 p->lock
32 static void freeproc(struct proc *proc)
33 {
34     if (proc->trapframe)
35         kfree((void *)proc->trapframe); // 释放trapframe
36     proc->trapframe = 0;
37     if (proc->pagetable)
38         proc_freepagetable(proc->pagetable, proc->sz); // 释放用户页表
39     proc->pagetable = 0;
40     proc->sz = 0;
41     proc->pid = 0;
42     proc->parent = 0;
43     proc->name[0] = 0;
44     proc->chan = 0;
45     proc->killed = 0;
46     proc->xstate = 0;
47     proc->state = UNUSED;
48     if (proc->kstack) // 释放内核栈
49     {
50         pte_t *pte_ptr = walk(proc->kpagetable, proc->kstack, 0); // 获取内核
// 栈相关的页表项
51         if (pte_ptr == 0)
52             panic("freeproc: walk");
53         kfree((void *)PTE2PA(*pte_ptr)); // 释放内核栈的物理内存
54     }
55     proc->kstack = 0;
56     if (proc->kpagetable) // 释放内核页表
57     {
58         proc_free_kernel_pagetable(proc);
59     }
60     proc->kpagetable = 0;
61 }
62

```

## 实验现象

每个进程在内核中执行时使用自己的内核页表副本，确保了进程之间的内存隔离和地址转换的正确性。

```

usertrap(): unexpected scause 0x000000000000000d pid=6247
sepc=0x0000000000000201a stval=0x00000000080186a00
usertrap(): unexpected scause 0x000000000000000d pid=6248
sepc=0x0000000000000201a stval=0x00000000080192d50
usertrap(): unexpected scause 0x000000000000000d pid=6249
sepc=0x0000000000000201a stval=0x0000000008019f0a0
usertrap(): unexpected scause 0x000000000000000d pid=6250
sepc=0x0000000000000201a stval=0x000000000801ab3f0
usertrap(): unexpected scause 0x000000000000000d pid=6251
sepc=0x0000000000000201a stval=0x000000000801b7740
usertrap(): unexpected scause 0x000000000000000d pid=6252
sepc=0x0000000000000201a stval=0x000000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6253
sepc=0x0000000000000201a stval=0x000000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6254
sepc=0x0000000000000201a stval=0x000000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6266
sepc=0x00000000000003e7a stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6270
sepc=0x00000000000002188 stval=0x000000000000fbc0
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$ █

```

## Simplify copyin/copyinstr

### 实验目的

`copyin` 函数用于将用户空间的数据复制到内核空间。通过将用户空间的映射添加到每个进程的内核页表中，可以提高 `copyin` 函数的效率。

### 实验步骤

1. **实现 `u2kvmcopy` 函数**：在 `kernel/vm.c` 中添加 `u2kvmcopy` 函数，将用户空间的映射复制到内核空间的页表中。

```

1 // 将用户页表复制到内核页表
2 void u2kvmcopy(pagetable_t pagetable, pagetable_t kpagetable, uint64
  oldsz, uint64 newsz)
3 {
4     pte_t *srcPte, *dstPte; // 定义页表项指针

```

```

5   uint64 virtualAddr, physicalAddr; // 定义虚拟地址和物理地址
6   uint flags; // 定义页表项标志
7   if (newsz < oldsz) // 如果新大小小于旧大小, 直接返回
8       return;
9   oldsz = PGROUNDUP(oldsz); // 将旧大小向上取整到页边界
10  for (virtualAddr = oldsz; virtualAddr < newsz; virtualAddr += PGSIZE)
11  {
12      // 获取用户页表项
13      if ((srcPte = walk(pagetable, virtualAddr, 0)) == 0)
14          panic("u2kvmcopy: pte should exist");
15      // 获取内核页表项
16      if ((dstPte = walk(kpagetable, virtualAddr, 1)) == 0)
17          panic("u2kvmcopy: walk fails");
18      physicalAddr = PTE2PA(*srcPte); // 获取物理地址
19      flags = (PTE_FLAGS(*srcPte) & (~PTE_U)); // 获取标志并清除PTE_U
20      *dstPte = PA2PTE(physicalAddr) | flags; // 设置内核页表项
21  }
22  }

```

2. 修改相关函数: 在 `fork`、`exec` 和 `sbrk` 函数中, 确保用户空间的映射同步到内核页表。

```

1   int fork(void)
2   {
3       int i, pid;
4       struct proc *np;
5       struct proc *p = myproc();
6
7       // Allocate process.
8       if ((np = allocproc()) == 0)
9       {
10          return -1;
11      }
12
13      // Copy user memory from parent to child.
14      if (uvmcopy(p->pagetable, np->pagetable, p->sz) < 0)
15      {
16          freeproc(np);
17          release(&np->lock);
18          return -1;
19      }
20      np->sz = p->sz;
21
22      np->parent = p;
23
24      // copy saved user registers.
25      *(np->trapframe) = *(p->trapframe);
26
27      // Cause fork to return 0 in the child.
28      np->trapframe->a0 = 0;
29
30      // increment reference counts on open file descriptors.
31      for (i = 0; i < NOFILE; i++)
32          if (p->ofile[i])
33              np->ofile[i] = filedup(p->ofile[i]);
34      np->cwd = idup(p->cwd);
35

```

```

36     u2kvmcopy(np->pagetable, np->kpagetable, 0, np->sz);
37
38     safestrcpy(np->name, p->name, sizeof(p->name));
39
40     pid = np->pid;
41
42     np->state = RUNNABLE;
43
44     release(&np->lock);
45
46     return pid;
47 }
48
49 int exec(...)
50 {
51     ...
52     // Allocate two pages at the next page boundary.
53     // Use the second as the user stack.
54     sz = PGROUNDUP(sz);
55     uint64 sz1;
56     if((sz1 = uvmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
57         goto bad;
58     sz = sz1;
59     uvmclear(pagetable, sz-2*PGSIZE);
60     sp = sz;
61     stackbase = sp - PGSIZE;
62     u2kvmcopy(pagetable, p->kpagetable, 0, sz);
63     if(p->pid == 1)
64         vmprint(p->pagetable);
65     ...
66 }
67
68 int growproc(int n)
69 {
70     uint sz;
71     struct proc *p = myproc();
72     sz = p->sz;
73     if (n > 0)
74     {
75         if (PGROUNDUP(sz + n) >= PLIC)
76             return -1;
77         if ((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0)
78             return -1;
79         u2kvmcopy(p->pagetable, p->kpagetable, sz - n, sz);
80     }
81     else if (n < 0)
82         sz = uvdealloc(p->pagetable, sz, sz + n);
83     p->sz = sz;
84     return 0;
85 }

```

3. **重写 copyin 和 copyinstr 函数**: 在 kernel/vm.c 中重写 copyin 和 copyinstr 函数, 使其利用内核页表进行数据复制。

```

1  int copyin_new(pagetable_t kpagetable, char *dst, uint64 srcva, uint64
    len) {
2      return copyin_new(pagetable, dst, srcva, len);
3  }
4
5  int copyinstr_new(pagetable_t kpagetable, char *dst, uint64 srcva, uint64
    max) {
6      return copyinstr_new(pagetable, dst, srcva, max);
7  }

```

## 实验结果

```

== Test pte printout ==
$ make qemu-gdb
pte printout: OK (6.5s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.5s)
== Test usertests ==
$ make qemu-gdb
(263.1s)
== Test usertests: copyin ==
usertests: copyin: OK
== Test usertests: copyinstr1 ==
usertests: copyinstr1: OK
== Test usertests: copyinstr2 ==
usertests: copyinstr2: OK
== Test usertests: copyinstr3 ==
usertests: copyinstr3: OK
== Test usertests: sbrkmuch ==
usertests: sbrkmuch: OK
== Test usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: OK
Score: 66/66
(base) sheerio@ubuntu:~/xv6/lab3 pgtbl/xv6-labs-2021$

```

## 实验中遇到的问题及解决办法

### Speed Up System Calls

#### 1. 页面映射失败:

- **问题描述:** 在使用 `mappages()` 函数进行页面映射时, 有时会遇到映射失败的情况, 导致系统调用无法正常执行或出现崩溃。
- **解决办法:** 在 `proc_pagetable()` 函数中调用 `mappages()` 时, 如果映射失败, 确保及时清理已经映射的页面。使用 `uvmunmap()` 取消映射, 并调用 `uvmfree()` 释放内存。这可以防止部分映射成功但最终操作失败的情况。

```

1  if (mappages(pagetable, USYSCALL, PGSIZE, (uint64)(p->usyscall), PTE_R |
    PTE_U) < 0) {
2      uvmunmap(pagetable, TRAMPOLINE, 1, 0);
3      uvmunmap(pagetable, TRAPFRAME, 1, 0);
4      uvmfree(pagetable, 0);
5      return 0;
6  }

```

## Print a Page Table

### 1. 递归遍历错误:

- **问题描述:** 在递归遍历页表打印内容时, 设计不当可能导致函数无法正确地打印页表项。
- **解决办法:** 参考 `freewalk` 函数的递归实现, 确保在递归过程中正确处理页表项。在 `vmprint` 函数中, 通过递归遍历页表, 并在每层递归时打印页表项和层级信息。

```

1  void vmprint(pagetable_t pagetable, uint dep) {
2      if (dep == 0)
3          printf("page table %p\n", pagetable);
4      for (int i = 0; i < 512; i++) {
5          pte_t pte = pagetable[i];
6          if (pte & PTE_V) {
7              for (int j = 0; j < dep; j++)
8                  printf(".. ");
9              uint64 child = PTE2PA(pte);
10             printf("..%d: pte %p pa %p\n", i, pte, child);
11             if (dep < 2)
12                 vmprint((pagetable_t) child, dep + 1);
13         }
14     }
15 }

```

## 实验心得

通过此次 xv6 的 Lab 3 实验, 我对操作系统中页表的功能和内存管理机制有了更加深入的理解。在实验过程中, 面对多种技术挑战, 我不仅提高了编码能力, 还积累了许多解决实际问题的经验。

首先, 在实现加速系统调用的部分, 我体会到了优化系统性能的重要性。通过将数据共享在用户空间和内核之间的只读区域, 我们减少了用户态和内核态之间的频繁切换。这一优化思路在实际操作系统中广泛应用, 尤其是在需要高效处理大量系统调用的场景下, 更能体现其重要性。

在实现打印页表的过程中, 我深入学习了页表的层级结构以及虚拟地址到物理地址的映射原理。通过编写 `vmprint` 函数并调试输出内容, 我清晰地看到了页表是如何逐级映射的。这不仅帮助我巩固了操作系统课程中的理论知识, 还让我掌握了更多实际操作技能, 特别是在递归遍历和格式化输出方面的能力得到了提升。

最后, 在实现检测页表访问位的功能时, 我学会了如何通过位操作来处理访问标志位。这一过程不仅增强了我对位操作的理解, 也让我认识到内存管理和数据传输的复杂性和精确性。在解决访问位清除和数据传输的问题时, 我深刻体会到操作系统设计的严谨性和科学性。

# lab4 Traps

## 实验综述

本次实验的重点是了解和实现陷阱（trap）机制。陷阱是操作系统处理各种事件的关键部分，包括中断、异常和系统调用。通过这两个部分的实验，我们将深入探索陷阱的工作原理，并实现相应的功能。

```
1 | git checkout traps
```

## 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. **添加函数声明**：在 `kernel/defs.h` 中添加 `backtrace` 函数声明。

```
1 | //printf.c
2 | void backtrace(void);
```

2. **定义读取帧指针的函数**：在 `kernel/riscv.h` 中添加以下代码，通过内联汇编读取当前帧指针。

```
1 | //读取当前帧指针
2 | static inline uint64
3 | r_fp()
4 | {
5 |     uint64 x;
6 |     asm volatile("mv %0, s0" : "=r" (x) );
7 |     return x;
8 | }
```

3. **实现 backtrace 函数**：在 `kernel/printf.c` 中实现 `backtrace` 函数，遍历调用堆栈中的帧指针，输出保存在每个栈帧中的返回地址。

```
1 | void
2 | backtrace(void)
3 | {
4 |     printf("backtrace:\n");
5 |     uint64 addr = r_fp(); //获取当前函数栈帧的帧指针
6 |     while (PGROUNDUP(addr) - PGROUNDDOWN(addr) == PGSIZE)
7 |     {
8 |         uint64 ret_addr = *(uint64 *) (addr - 8);
9 |         printf("%p\n", ret_addr);
10 |        addr = *((uint64 *) (addr - 16)); //获取下一个栈帧的帧指针
11 |    }
12 | }
```

4. **调用 backtrace 函数**：在 `kernel/sysproc.c` `sys_sleep` 函数中调用 `backtrace` 函数。



```

1  uint64
2  sys_sleep(void)
3  {
4      backtrace();
5      int n;
6      uint ticks0;
7
8      if(argint(0, &n) < 0)
9          return -1;
10     acquire(&tickslock);
11     ticks0 = ticks;
12     while(ticks - ticks0 < n){
13         if(myproc()->killed){
14             release(&tickslock);
15             return -1;
16         }
17         sleep(&ticks, &tickslock);
18     }
19     release(&tickslock);
20     return 0;
21 }

```

5. **实验现象：**编译并运行 `bttest` 程序，该程序会调用 `sys_sleep`。

```

(base) sheerio@ubuntu:~/xv6/lab4 traps/xv6-labs-2021$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002d76
0x0000000080002c50
0x00000000800028d2
$

```

6. **添加回溯到 panic：**将 `backtrace` 函数添加到 `panic` 函数中，这样在内核发生 panic 时，可以看到内核的回溯信息。

```

1  void panic(char *s) {
2      // 原有的 panic 函数逻辑
3      backtrace();
4      // 继续原有的 panic 函数逻辑
5  }

```

# Alarm

## 实验目的

实现进程可以周期性地接收定时提醒功能。类似于用户级的中断/异常处理程序，允许进程在消耗一定的 CPU 时间后执行指定的函数，然后恢复执行。这对于限制计算密集型进程的 CPU 时间或提供周期性操作支持非常有用。

## 实验步骤

1. **添加 Makefile 入口：**在 `Makefile` 中添加 `alarmtest` 程序的编译。

```
1 | UPROGS += _alarmtest\
```

2. **声明库函数：**在 `user/user.h` 中声明 `sigalarm` 和 `sigreturn` 函数。

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

3. **更新系统调用入口：**在 `user/usys.pl` 中添加相应的用户态库函数入口。

```
1 | entry("sigalarm");  
2 | entry("sigreturn");
```

4. **声明系统调用：**在 `kernel/syscall.h` 中声明 `sigalarm` 和 `sigreturn` 的系统调用编号。

```
1 | #define SYS_sigalarm 22  
2 | #define SYS_sigreturn 23
```

5. **添加系统调用处理函数：**在 `kernel/sysproc.c` 中实现 `sys_sigalarm` 和 `sys_sigreturn` 的内核处理逻辑。

```
1 | extern uint64 sys_sigalarm(void);  
2 | extern uint64 sys_sigreturn(void);
```

6. **修改进程结构体：**在 `kernel/proc.h` 中修改 `proc` 结构体，添加新字段以存储警报信息。

```
1 | int interval;           // interval of alarm  
2 | int since_interval;     // last call until now  
3 | void (*handler)();      // function pointer  
4 | int running_hand;       // flag of running765  
5 | struct trapframe trapframe_cp; //copy of trapframe_cp
```

7. **初始化新字段：**在 `kernel/proc.c` 中的 `allocproc()` 函数中初始化这些新字段。

```

1 // Set up new context to start executing at forkret,
2 // which returns to user space.
3 memset(&p->context, 0, sizeof(p->context));
4 p->context.ra = (uint64)forkret;
5 p->context.sp = p->kstack + PGSIZE;
6
7 p->interval=0;
8 p->since_interval=0;
9 p->running_hand=0;

```

8. 实现 `sys_sigalarm` 函数: 在 `sysproc.c` 中实现 `sys_sigalarm` 函数。

```

1 uint64
2 sys_sigalarm(void) // 设置定时器并关联中断处理函数
3 {
4     int timeInterval; // 定时器的时间间隔
5     uint64 handlerAddr; // 中断处理函数的地址
6
7     if(argint(0, &timeInterval) < 0)
8         return -1;
9
10    if(timeInterval == 0) // 如果时间间隔为0, 则停止定时器
11        return 0;
12
13    if(argaddr(1, &handlerAddr) < 0)
14        return -1;
15
16    myproc()->handler = (void *)handlerAddr; // 设置中断处理函数
17    myproc()->since_interval = 0; // 初始化经过的时间
18    myproc()->running_hand = 0; // 确保没有其他中断处理程序正在运行
19    myproc()->interval = timeInterval; // 设置定时器的时间间隔
20
21    return 0;
22 }

```

9. 处理时钟中断: 在 `kernel/trap.c` 中的 `usertrap()` 函数中, 处理时钟中断, 触发警报处理函数。

```

1 if (which_dev == 2)
2 {
3     if (p->interval != 0)
4     {
5         if (!p->running_hand)
6             p->since_interval = p->since_interval + 1;
7         if (!p->running_hand && p->since_interval == p->interval)
8         {
9             printf("alarm!\n");
10            p->running_hand = 1;
11            p->trapframe_cp = *(p->trapframe);
12            p->trapframe->epc = (uint64)p->handler;
13        }
14    }
15    yield();
16 }

```

10. 实现 `sys_sigreturn` 函数: 在 `sysproc.c` 中实现 `sys_sigreturn` 函数, 用于恢复中断前的执行状态。

```
1 uint64
2 sys_sigreturn(void) // 恢复被中断的程序的执行, 并清除一些相关的状态信息
3 {
4     struct proc *p = myproc(); // 获取当前进程的指针
5
6     p->since_interval = 0; // 重置since_interval, 跟踪自上一次定时器中断以来的时间
7
8     *(p->trapframe) = p->trapframe_cp; // 恢复被中断的程序的执行上下文
9
10    p->running_hand = 0; // 确保没有其他中断处理程序正在运行
11
12    return 0;
13 }
```

11. 实验现象

```
xv6 kernel is booting

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

## 实验结果

```
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (6.7s)
== Test running alarmtest ==
$ make qemu-gdb
(4.4s)
== 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 (201.8s)
== Test time ==
time: OK
Score: 85/85
```

## 实验中遇到的问题和解决办法

### 1. 警报处理函数的返回

- **问题描述**: 在处理完警报处理函数后, 如何正确返回到中断前的执行状态。
- **解决办法**: 通过在 `sys_sigreturn` 函数中恢复保存的中断前的寄存器状态和程序计数器, 确保处理完警报后能正确恢复执行。

### 2. 帧指针的使用

- **问题描述**: 在理解帧指针在调用堆栈中的作用以及如何使用它来访问上一级函数的返回地址时遇到了困难。
- **解决办法**: 通过阅读相关资料和课堂笔记, 我了解到返回地址和帧指针之间的固定偏移。返回地址相对于帧指针的偏移量是 -8, 而保存的帧指针相对于当前帧指针的偏移量是 -16。利用这些信息, 我可以正确地遍历堆栈帧并输出返回地址。

## 实验心得

通过实现定时警报功能, 我学会了如何在操作系统内核中处理定时事件。这不仅包括设置和触发定时器, 还涉及到在中断处理程序中保存和恢复进程状态。

首先, 我理解了警报处理函数的执行和返回机制。通过设置警报间隔和处理函数, 在时钟中断时触发警报处理函数, 然后通过 `sys_sigreturn` 函数恢复原始状态, 确保程序能够继续执行。

其次, 时钟中断处理是实现定时功能的关键。通过在 `usertrap` 函数中处理时钟中断, 我们可以精确控制警报触发的时间间隔。这让我更加理解了中断处理程序的设计和实现。

此外, 本实验还让我体验到了操作系统的实时性要求。在处理定时事件时, 需要确保系统的响应速度和准确性, 这对提高操作系统的性能和稳定性具有重要意义。

# lab 5: lazy page allocation

## 实验目的

本次实验的重点是实现懒惰分配（Lazy Allocation），这一技术通过延迟分配内存页，直到实际访问时才进行分配，从而优化内存使用。通过实现这一功能，我们将探索操作系统中内存管理的高级技术，并提高对内存管理机制的理解。

```
1 | git checkout cow### 实验内容
```

## Eliminate Allocation from `sbrk()`

### 实验目的

删除 `sbrk()` 系统调用中实际分配内存的部分，仅调整进程的大小。这一修改为后续实现懒惰分配做准备。

### 实验步骤

首先，删除 `sbrk()` 中实际分配内存的部分，只保留更新进程大小的代码：

```
1 | uint64
2 | sys_sbrk(void)
3 | {
4 |     int addr;
5 |     int n;
6 |
7 |     if (argint(0, &n) < 0)
8 |         return -1;
9 |     addr = myproc()->sz;
10 |    // Remove actual memory allocation
11 |    // if (growproc(n) < 0)
12 |    //     return -1;
13 |    myproc()->sz += n; // Just increase the process size
14 |    return addr;
15 | }
```

删除 `growproc()` 的调用后，执行诸如 `echo hi` 的命令会导致 panic，因为没有实际分配内存。

## Lazy Allocation

### 实验目的

实现页表的懒分配，只有在发生缺页错误时才实际分配内存。通过这种方式，可以减少不必要的内存分配，提高内存使用效率。

### 实验步骤

1. **检测缺页错误**：在 `trap.c` 中检测缺页错误，查询 `scause` 寄存器，如果是 13 或 15 就进行下一步的处理。

```

1  else if ((which_dev = devintr()) != 0) {
2      // ok
3  } else if (r_scause() == 13 || r_scause() == 15) {
4      // Handle page fault
5      if (lazy_alloc(r_stval()) < 0) {
6          p->killed = 1;
7      }
8  } else {
9      printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p-
>pid);
10     printf("                sepc=%p stval=%p\n", r_sepc(), r_stval());
11     p->killed = 1;
12 }

```

2. 修改函数：在缺页错误时分配新内存。

```

1  else if (r_scause() == 15)
2  { // 写页面错
3      uint64 va = PGROUNDDOWN(r_stval());
4      pte_t *pte;
5      if (va >= MAXVA)
6      { // 虚拟地址错
7          printf("va is larger than MAXVA!\n");
8          p->killed = 1;
9          goto end;
10     }
11     if (va > p->sz)
12     { // 虚拟地址超出进程的地址空间
13         printf("va is larger than sz!\n");
14         p->killed = 1;
15         goto end;
16     }
17     if ((pte = walk(p->pagetable, va, 0)) == 0)
18     {
19         printf("usertrap(): page not found\n");
20         p->killed = 1;
21         goto end;
22     }
23     // 分配一个新页面
24     if (((*pte) & PTE_COW) == 0 || ((*pte) & PTE_V) == 0 || ((*pte) &
PTE_U) == 0)
25     {
26         printf("usertrap: pte not exist or it's not cow page\n");
27         p->killed = 1;
28         goto end;
29     }
30     uint64 pa = PTE2PA(*pte);
31     acquire_refcnt();
32     uint ref = kgetref((void *)pa);
33     if (ref == 1)
34     { // 直接使用该页
35         *pte = ((*pte) & (~PTE_COW)) | PTE_W;
36     }
37     else
38     { // 分配物理页

```

```

39     char *mem = kalloc();
40     if (mem == 0)
41     {
42         printf("usertrap(): memery alloc fault\n");
43         p->killed = 1;
44         release_refcnt();
45         goto end;
46     }
47
48     memmove(mem, (char *)pa, PGSIZE);
49     uint flag = (PTE_FLAGS(*pte) | PTE_W) & (~PTE_COW);
50     if (mappages(p->pagetable, va, PGSIZE, (uint64)mem, flag) != 0)
51     {
52         kfree(mem);
53         printf("usertrap(): can not map page\n");
54         p->killed = 1;
55         release_refcnt();
56         goto end;
57     }
58     kfree((void *)pa);
59 }
60 release_refcnt();
61 }
62 else
63 {
64     printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p-
>pid);
65     printf("          sepc=%p stval=%p\n", r_sepc(), r_stval());
66     p->killed = 1;
67 }
68 end:
69 if (p->killed)
70     exit(-1);
71 if (which_dev == 2)
72     yield();
73
74     usertrapret();

```

3. 修改 `uvmunmap` 函数：防止试图解除未分配的页表映射时引发 panic。

```

1 void
2 uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
3 {
4     uint64 a;
5     pte_t *pte;
6
7     if((va % PGSIZE) != 0)
8         panic("uvmunmap: not aligned");
9
10    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
11        if((pte = walk(pagetable, a, 0)) == 0)
12            panic("uvmunmap: walk");
13        if((*pte & PTE_V) == 0)
14            panic("uvmunmap: not mapped");
15        if(PTE_FLAGS(*pte) == PTE_V)
16            panic("uvmunmap: not a leaf");

```



```

17     if(do_free){
18         uint64 pa = PTE2PA(*pte);
19         kfree((void*)pa);
20     }
21     *pte = 0;
22 }
23 }

```

## 实验中遇到的问题及解决办法

### 1. 缺页错误处理:

- **问题描述:** 在执行命令时发生缺页错误, 需要处理这些错误。
- **解决办法:** 实现 `lazy_alloc` 函数, 在缺页错误发生时分配新内存, 并将其映射到正确的虚拟地址。

### 2. 未分配页的解除映射:

- **问题描述:** 在解除映射未分配的页时, 会引发 panic。
- **解决办法:** 修改 `uvmunmap` 函数, 在遇到未分配的页时跳过解除映射操作。

## Lazytests and Usertests

### 实验目的

通过 usertests 和 lazytests 检验懒分配功能的正确性, 并修复其中的 bug。

### 实验步骤

1. **正确处理 fork 中的内存拷贝:** 在 `vm.c` 中的 `uvmcopy` 函数中正确处理父进程的懒分配页。

```

1  int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
2  {
3      pte_t *pte;
4      uint64 pa, i;
5      uint flags;
6
7      for (i = 0; i < sz; i += PGSIZE)
8      {
9          if ((pte = walk(old, i, 0)) == 0) // 查找旧页表中虚拟地址为i的页表项
10             panic("uvmcopy: pte should exist");
11          if ((*pte & PTE_V) == 0) // 检查页表项是否有效
12             panic("uvmcopy: page not present");
13
14          *pte = ((*pte) & (~PTE_W)) | PTE_COW; // 将页表项设置为COW (写时复制)
15          flags = PTE_FLAGS(*pte); // 获取页表项的属性
16          pa = PTE2PA(*pte); // 获取物理地址
17
18          // 在新页表中映射子进程的虚拟地址到父进程的物理地址
19          if (mappages(new, i, PGSIZE, pa, flags) != 0)
20          {
21              goto err; // 映射失败时跳转到错误处理部分
22          }
23          kaddref((void *)pa); // 增加物理页的引用计数
24      }
25      return 0;
26

```

```

27 err:
28     // 发生错误时取消映射
29     uvmunmap(new, 0, i / PGSIZE, 1);
30     return -1;
31 }
32

```

## 2. 处理系统调用中的懒分配页：在 `copyin` 和 `copyout` 函数中处理懒分配页

```

1 int copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
2 {
3     uint64 numBytes, virtAddr, physAddr;
4     pte_t *pte; // 页表项指针
5
6     while (len > 0)
7     {
8         virtAddr = PGROUNDDOWN(dstva); // 将虚拟地址向下舍入到页边界
9         if (virtAddr >= MAXVA) // 检查虚拟地址是否超出范围
10             return -1;
11         if ((pte = walk(pagetable, virtAddr, 0)) == 0) // 查找页表项
12             return -1;
13         if (((*pte & PTE_V) == 0) || ((*pte & PTE_U) == 0)) // 检查页表项是否有
            效且用户可访问
14             return -1;
15         physAddr = PTE2PA(*pte); // 获取物理地址
16         if (((*pte & PTE_W) == 0) && (*pte & PTE_COW)) // 处理写时复制
17         {
18             acquire_refcnt(); // 获取引用计数锁
19             if (kgetref((void *)physAddr) == 1) // 如果引用计数为1，直接设置为可写
20             {
21                 *pte = (*pte | PTE_W) & (~PTE_COW);
22             }
23             else
24             { // 分配新物理页并复制数据
25                 char *newMem = kalloc();
26                 if (newMem == 0)
27                 {
28                     printf("copyout(): memory alloc fault\n");
29                     release_refcnt();
30                     return -1;
31                 }
32                 memmove(newMem, (void *)physAddr, PGSIZE);
33                 uint newFlags = (PTE_FLAGS(*pte) & (~PTE_COW)) | PTE_W;
34                 if (mappages(pagetable, virtAddr, PGSIZE, (uint64)newMem,
                    newFlags) != 0)
35                 {
36                     kfree(newMem);
37                     release_refcnt();
38                     return -1;
39                 }
40                 kfree((void *)physAddr); // 释放旧物理页
41             }
42             release_refcnt(); // 释放引用计数锁
43         }
44         physAddr = walkaddr(pagetable, virtAddr); // 获取物理地址
45         if (physAddr == 0)

```

```

46     return -1;
47     numBytes = PGSIZE - (dstva - virtAddr); // 计算需要复制的字节数
48     if (numBytes > len)
49         numBytes = len;
50     memmove((void *) (physAddr + (dstva - virtAddr)), src, numBytes); //
复制数据
51
52     len -= numBytes;
53     src += numBytes;
54     dstva = virtAddr + PGSIZE;
55 }
56 return 0;
57 }
58 int
59 copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
60 {
61     uint64 n, va0, pa0;
62
63     while(len > 0){
64         va0 = PGROUNDDOWN(srcva);
65         pa0 = walkaddr(pagetable, va0);
66         if(pa0 == 0)
67             return -1;
68         n = PGSIZE - (srcva - va0);
69         if(n > len)
70             n = len;
71         memmove(dst, (void *) (pa0 + (srcva - va0)), n);
72
73         len -= n;
74         dst += n;
75         srcva = va0 + PGSIZE;
76     }
77     return 0;
78 }
79

```

### 3. 处理walkaddr

```

1  uint64
2  walkaddr(pagetable_t pagetable, uint64 va)
3  {
4      pte_t *pte;
5      uint64 pa;
6
7      if(va >= MAXVA)
8          return 0;
9
10     pte = walk(pagetable, va, 0);
11     if(pte == 0)
12         return 0;
13     if((*pte & PTE_V) == 0)
14         return 0;
15     if((*pte & PTE_U) == 0)
16         return 0;
17     pa = PTE2PA(*pte);
18     return pa;

```

## 实验结果

```

== Test running cowtest ==
$ make qemu-gdb
(24.3s)
== Test    simple ==
    simple: OK
== Test    three ==
    three: OK
== Test    file ==
    file: OK
== Test usertests ==
$ make qemu-gdb
(288.8s)
== 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
(base) sheerio@ubuntu:~/xv6/lab5 cow/xv6-labs-2021$

```

## 实验中遇到的问题及解决办法

### 1. 处理 `sbrk` 的负数参数:

- **问题描述:** 负数参数表示减少进程的地址空间，需要实际释放内存。
- **解决办法:** 在 `sys_sbrk` 中处理负数参数，通过 `growproc` 函数实际释放内存。

### 2. 正确处理 `fork` 中的内存拷贝:

- **问题描述:** 在 `fork` 中，需要正确处理父进程的懒分配页，防止拷贝未分配的页。
- **解决办法:** 在 `uvmcopy` 函数中跳过未分配的页，确保只拷贝已分配的页。

## 实验心得

### 1. 内存管理机制的复杂性:

- 在实现懒分配的过程中，我体会到了内存管理的复杂性。操作系统需要处理各种不同的情况，确保内存分配和释放的正确性，以避免内存泄漏和其他错误。

### 2. 页表的灵活性和重要性:

- 页表在内存管理中起着至关重要的作用。通过操作页表，我们可以实现各种高级功能，如懒分配、虚拟内存保护等。本次实验让我更深入地理解了页表的工作原理及其应用。

### 3. 错误处理的重要性:

- 在实现懒分配过程中，处理各种可能的错误情况是非常重要的。例如，内存不足、无效地址访问等。正确处理这些错误，不仅可以提高系统的稳定性，还可以增强系统的鲁棒性。

# lab 6: Multithreading

## 实验综述

本次实验的目标是实现用户态线程和同步屏障。通过该实验，我们可以深入理解多线程编程和线程间同步的基本概念和实现方法。

```
1 | git checkout thread
```

## Uthread: switching between threads

### 实验目的

实现用户态的多线程机制。用户态线程在实现上类似于内核态线程，但它们在用户态运行，不依赖于操作系统内核的线程管理。这使得线程的创建、调度和切换都在用户态完成，从而减轻了内核的负担，提高了效率。

### 实验步骤

#### 1. 定义线程结构体：

在 `user/uthread.c` 中定义线程结构体 `thread`，其中包含了线程的栈、状态和上下文。

```
1 | struct Context{
2 |     uint64 ra;
3 |     uint64 sp;
4 |
5 |     // callee-saved
6 |     uint64 s0;
7 |     uint64 s1;
8 |     uint64 s2;
9 |     uint64 s3;
10 |    uint64 s4;
11 |    uint64 s5;
12 |    uint64 s6;
13 |    uint64 s7;
14 |    uint64 s8;
15 |    uint64 s9;
16 |    uint64 s10;
17 |    uint64 s11;
18 | };
19 |
20 | struct thread {
21 |     char    stack[STACK_SIZE];
22 |     int     state;
23 |     struct Context ctx;
24 | };
```

#### 2. 实现 `thread_switch` 函数：

该函数用于保存当前线程的上下文，并恢复下一个线程的上下文。其实现与内核中的 `switch` 函数类似。

```
1 | .text
```

```

2  .globl thread_switch
3  thread_switch:
4      sd ra, 0(a0)
5      sd sp, 8(a0)
6      sd s0, 16(a0)
7      sd s1, 24(a0)
8      sd s2, 32(a0)
9      sd s3, 40(a0)
10     sd s4, 48(a0)
11     sd s5, 56(a0)
12     sd s6, 64(a0)
13     sd s7, 72(a0)
14     sd s8, 80(a0)
15     sd s9, 88(a0)
16     sd s10, 96(a0)
17     sd s11, 104(a0)
18
19     ld ra, 0(a1)
20     ld sp, 8(a1)
21     ld s0, 16(a1)
22     ld s1, 24(a1)
23     ld s2, 32(a1)
24     ld s3, 40(a1)
25     ld s4, 48(a1)
26     ld s5, 56(a1)
27     ld s6, 64(a1)
28     ld s7, 72(a1)
29     ld s8, 80(a1)
30     ld s9, 88(a1)
31     ld s10, 96(a1)
32     ld s11, 104(a1)
33
34     ret

```

### 3. 实现 `thread_create` 函数:

该函数用于创建新的线程。为每个线程分配栈空间，并初始化线程的上下文，使其能够正确地执行用户提供的函数。

```

1  void thread_create(void (*func)()) {
2      struct thread *t;
3
4      for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
5          if (t->state == FREE) break;
6      }
7      t->state = RUNNABLE;
8      t->ctx.ra = (uint64) func;
9      t->ctx.sp = (uint64) &t->stack + (STACK_SIZE - 1);
10 }

```

### 4. 实现 `thread_schedule` 函数:

该函数用于调度线程。遍历所有线程，找到一个 `RUNNABLE` 状态的线程，并切换到该线程执行。

```

1  void thread_schedule(void) {
2      struct thread *t, *next_thread;
3

```

```

4   next_thread = 0;
5   t = current_thread + 1;
6   for (int i = 0; i < MAX_THREAD; i++) {
7       if (t >= all_thread + MAX_THREAD)
8           t = all_thread;
9       if (t->state == RUNNABLE) {
10          next_thread = t;
11          break;
12      }
13      t = t + 1;
14  }
15
16  if (next_thread == 0) {
17      printf("thread_schedule: no runnable threads\n");
18      exit(-1);
19  }
20
21  if (current_thread != next_thread) {
22      next_thread->state = RUNNING;
23      t = current_thread;
24      current_thread = next_thread;
25      thread_switch((uint64) &t->ctx, (uint64) &next_thread->ctx);
26  } else {
27      next_thread = 0;
28  }
29  }

```

##### 5. 实验现象:

命令行输入make qemu 运行 xv6, 然后执行 uthread

```

thread_c 94
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
$ 

```

# Using Threads

## 实验目的

使用用户态线程在多线程环境中正确工作，特别是在共享资源的情况下，通过加锁机制确保线程安全。

## 实验步骤

### 1. 理解散列表实现：

阅读 `user/uthread.c` 中的散列表实现代码，找出可能引发线程安全问题的地方。

### 2. 为关键部分加锁：

在 `put` 和 `get` 函数中添加互斥锁，以确保对共享资源的访问是线程安全的。

```
1 pthread_mutex_t bkt_lock[NBUCKET];
2
3 static
4 void put(int key, int value)
5 {
6     int i = key % NBUCKET;
7
8     pthread_mutex_lock(&bkt_lock[i]);
9     struct entry *e = 0;
10    for (e = table[i]; e != 0; e = e->next) {
11        if (e->key == key)
12            break;
13    }
14    if(e){
15        e->value = value;
16    } else {
17        insert(key, value, &table[i], table[i]); // 在 table[i] 的最前面插入一个 key val 对
18    }
19    pthread_mutex_unlock(&bkt_lock[i]);
20 }
21
22 static struct entry*
23 get(int key)
24 {
25     int i = key % NBUCKET;
26
27     pthread_mutex_lock(&bkt_lock[i]);
28
29     struct entry *e = 0;
30     for (e = table[i]; e != 0; e = e->next) {
31         if (e->key == key) break;
32     }
33     pthread_mutex_unlock(&bkt_lock[i]);
34     return e;
35 }
```

### 3. 实验现象



```

(base) sheerio@ubuntu:~/xv6/lab6 thread/xv6-labs-2021$ make ph
make: 'ph' is up to date.
(base) sheerio@ubuntu:~/xv6/lab6 thread/xv6-labs-2021$ ./ph 1
100000 puts, 17.009 seconds, 5879 puts/second
0: 0 keys missing
100000 gets, 16.809 seconds, 5949 gets/second
(base) sheerio@ubuntu:~/xv6/lab6 thread/xv6-labs-2021$ ./ph 2
100000 puts, 9.888 seconds, 10113 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 19.810 seconds, 10096 gets/second
(base) sheerio@ubuntu:~/xv6/lab6 thread/xv6-labs-2021$

```

## Barrier

### 实验目的

实现同步屏障，使得一组线程能够在执行到同步点时等待，直到所有线程都到达该同步点后再继续执行。

### 实验步骤

#### 1. 定义屏障结构体 (barrier.c) :

定义包含互斥锁和条件变量的屏障结构体，以便实现线程同步。

```

1 struct barrier {
2     pthread_mutex_t barrier_mutex;
3     pthread_cond_t barrier_cond;
4     int nthread;      // Number of threads that have reached this round of
the barrier
5     int round;        // Barrier round
6 } bstate;

```

#### 2. 实现屏障函数:

实现 `barrier` 函数，使用互斥锁和条件变量来实现同步屏障。

```

1 static void
2 barrier()
3 {
4     // YOUR CODE HERE
5     //
6     // Block until all threads have called barrier() and
7     // then increment bstate.round.
8     //
9     pthread_mutex_lock(&bstate.barrier_mutex);
10    bstate.nthread++;
11    if(bstate.nthread < nthread){
12        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
13    }else{
14        bstate.nthread = 0;
15        bstate.round++;
16        pthread_cond_broadcast(&bstate.barrier_cond);
17    }
18    pthread_mutex_unlock(&bstate.barrier_mutex);
19 }

```

## 实验结果

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (8.2s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/sheerio/xv6/lab6 thread/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/sheerio/xv6/lab6 thread/xv6-labs-2021'
ph_safe: OK (29.9s)
== Test ph_fast == make[1]: Entering directory '/home/sheerio/xv6/lab6 thread/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/sheerio/xv6/lab6 thread/xv6-labs-2021'
ph_fast: OK (62.7s)
== Test barrier == make[1]: Entering directory '/home/sheerio/xv6/lab6 thread/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/sheerio/xv6/lab6 thread/xv6-labs-2021'
barrier: OK (12.3s)
== Test time ==
time: OK
Score: 60/60
(base) sheerio@ubuntu:~/xv6/lab6 thread/xv6-labs-2021$
```

## 实验中遇到的问题及解决方法

### Uthread

#### 1. 上下文切换问题

**问题：**实现 `thread_switch` 函数时，必须正确保存和恢复所有必要的寄存器状态，包括返回地址 (ra) 和栈指针 (sp)。

**解决方法：**参考内核中的 `swtch` 函数，通过汇编代码实现寄存器的保存和恢复。确保在切换时保存当前线程的上下文并恢复下一个线程的上下文。

```
1  .text
2  .globl thread_switch
3  thread_switch:
4      sd ra, 0(a0)
5      sd sp, 8(a0)
6      sd s0, 16(a0)
7      sd s1, 24(a0)
8      sd s2, 32(a0)
9      sd s3, 40(a0)
10     sd s4, 48(a0)
11     sd s5, 56(a0)
12     sd s6, 64(a0)
13     sd s7, 72(a0)
14     sd s8, 80(a0)
15     sd s9, 88(a0)
16     sd s10, 96(a0)
17     sd s11, 104(a0)
18
19     ld ra, 0(a1)
20     ld sp, 8(a1)
21     ld s0, 16(a1)
22     ld s1, 24(a1)
23     ld s2, 32(a1)
24     ld s3, 40(a1)
25     ld s4, 48(a1)
26     ld s5, 56(a1)
27     ld s6, 64(a1)
```

```

28     ld s7, 72(a1)
29     ld s8, 80(a1)
30     ld s9, 88(a1)
31     ld s10, 96(a1)
32     ld s11, 104(a1)
33     ret

```

## 2. 线程调度问题

**问题：**在 `thread_schedule` 函数中，如何找到一个可运行的线程并切换到该线程。

**解决方法：**遍历所有线程，找到状态为 `RUNNABLE` 的线程，并调用 `thread_switch` 函数进行上下文切换。

```

1 void thread_schedule(void) {
2     struct thread *t, *next_thread;
3
4     next_thread = 0;
5     t = current_thread + 1;
6     for (int i = 0; i < MAX_THREAD; i++) {
7         if (t >= all_thread + MAX_THREAD)
8             t = all_thread;
9         if (t->state == RUNNABLE) {
10             next_thread = t;
11             break;
12         }
13         t = t + 1;
14     }
15
16     if (next_thread == 0) {
17         printf("thread_schedule: no runnable threads\n");
18         exit(-1);
19     }
20
21     if (current_thread != next_thread) {
22         next_thread->state = RUNNING;
23         t = current_thread;
24         current_thread = next_thread;
25         thread_switch((uint64) &t->ctx, (uint64) &next_thread->ctx);
26     } else {
27         next_thread = 0;
28     }
29 }

```

## Barrier

### 1. 条件变量和互斥锁的结合使用

**问题：**条件变量的 `wait` 和 `broadcast` 如何与互斥锁结合使用，以确保同步操作的原子性。

**解决方法：**在 `pthread_cond_wait` 中，解锁和加入等待队列的操作是原子的，可以确保在释放锁的同时将线程加入等待队列。使用 `pthread_cond_broadcast` 唤醒所有等待的线程。

```
1 pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
2 pthread_cond_broadcast(&bstate.barrier_cond);
```

## 实验心得

通过本次实验，我对多线程编程有了更加深入的理解和实践体验。整个实验涵盖了用户态线程的实现、线程安全的共享数据访问以及同步屏障的应用，以下是我在实验过程中总结的一些心得体会：

### 1. 理解上下文切换的重要性

在实现用户态线程时，上下文切换是一个关键部分。通过编写 `thread_switch` 函数，我深刻体会到保存和恢复寄存器状态的重要性。每个线程在被切换时，其当前的状态必须被保存，以便下次运行时能够从中断的地方继续执行。这一过程不仅仅是对函数调用栈的管理，更是对整个线程执行环境的维护。

### 2. 掌握线程创建和调度的核心

实现 `thread_create` 和 `thread_schedule` 函数，使我理解了线程创建和调度的核心逻辑。在用户态创建线程时，必须正确设置栈指针和返回地址，以确保线程能够从指定的起点开始执行。在调度线程时，找到一个可运行的线程并切换到该线程，是实现多线程并发执行的关键。这一部分的实现让我更深刻地理解了操作系统内核中线程管理的机制。

### 3. 解决线程安全问题

在多线程环境下访问共享数据，容易出现数据竞争问题。通过为关键部分加锁，确保对共享资源的访问是线程安全的，这让我认识到并发编程中同步原语的重要性。在实现哈希表的线程安全操作时，使用互斥锁来保护共享数据，防止多个线程同时修改数据造成的不一致性，这一经验对于编写可靠的并发程序非常有帮助。

## lab 7: Networking

### 实验目的

本实验旨在实现E1000网卡驱动程序中的 `transmit` 和 `recv` 函数。这些函数负责处理数据包的发送和接收操作。通过实现这些函数，了解DMA（Direct Memory Access）技术在网卡数据传输中的应用，并掌握网络驱动程序的基本开发方法。

```
1 git checkout net
```

### 实验原理

E1000网卡使用DMA技术直接与计算机内存进行数据包的交互。DMA允许网卡在不需要CPU参与的情况下直接将数据从内存读取或写入内存，从而提高了数据传输的效率。在发送和接收数据包时，E1000使用描述符数组来管理数据包。每个描述符包含一个指向数据缓冲区的指针和一些状态信息。

在接收数据包时，网卡会将数据写入接收描述符所指向的缓冲区，并更新描述符的状态。当软件读取这些数据时，需要检查描述符的状态，以确定哪些数据包已经被接收。

在发送数据包时，软件将数据写入发送描述符所指向的缓冲区，并更新描述符的状态。网卡会读取这些描述符，并将数据包发送出去。当数据包发送完成后，网卡会更新描述符的状态。

## 实验步骤

### 1. E1000的交互方法

E1000网卡使用DMA技术直接与计算机内存进行数据包的交互。通过描述符数组来描述数据包，以实现接收和发送数据包的操作。

### 2. 接收描述符

接收描述符的定义如下：

```
1 struct rx_desc {
2     uint64 addr;    // Address of the descriptor's data buffer
3     uint16 length;  // Length of data DMAed into data buffer
4     uint16 csum;    // Packet checksum
5     uint8 status;   // Descriptor status
6     uint8 errors;   // Descriptor Errors
7     uint16 special;
8 };
```

描述符数组形成环形队列，网卡将接收到的数据包写入当前 head 位置的描述符缓冲区，并设置状态和长度。环形队列的 head 和 tail 用于管理缓冲区。

### 3. 发送描述符

发送描述符的定义如下：

```
1 struct tx_desc {
2     uint64 addr;
3     uint16 length;
4     uint8 cso;      // checksum offset
5     uint8 cmd;      // command field
6     uint8 status;   //
7     uint8 css;      // checksum start field
8     uint16 special; //
9 };
```

发送描述符的 addr 和 length 分别表示数据缓冲区的地址和长度，cmd 用于发送命令，status 标志发送状态。

### 4. 寄存器操作

通过内存映射访问E1000的控制寄存器，在 e1000\_dev.h 中定义了这些寄存器的偏移量。

### 5. 实现代码

发送函数 e1000\_transmit

1. 获取当前环形队列的 tail 位置。
2. 检查 tail 位置的描述符状态，如果没有 DD（Descriptor Done）标志位，表示队列已满。
3. 清理已发送的描述符缓存，并设置新的描述符地址和长度。
4. 设置发送命令，并更新 tail 位置。

```
1 int
2 e1000_transmit(struct mbuf *m)
```

```

3  {
4      //
5      // Your code here.
6      //
7      // the mbuf contains an ethernet frame; program it into
8      // the TX descriptor ring so that the e1000 sends it. Stash
9      // a pointer so that it can be freed after sending.
10     //
11     acquire(&e1000_lock);
12     uint idx = regs[E1000_TDT]; //第一个空闲的环形描述符
13     struct tx_desc *desc = &tx_ring[idx];
14
15     if(!(desc->status & E1000_TXD_STAT_DD)){
16         release(&e1000_lock);
17         return -1;
18     }
19
20     if(tx_mbufs[idx] != NULL){ // 要发的数据包
21         // tx_mbufs直接指向 m 这个参数
22         mbuf_free(tx_mbufs[idx]);
23         tx_mbufs[idx] = NULL;
24     }
25
26     desc->addr = m->head;
27     desc->length = m->len;
28
29     desc->cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
30
31     tx_mbufs[idx] = m;
32
33     regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;
34
35     release(&e1000_lock);
36     return 0;
37 }

```

### 接收函数 e1000\_rcv

1. 读取环形队列的 `tail` 位置。
2. 检查 `tail` 位置的描述符状态，如果有 `DD` 标志位，表示数据包已接收。
3. 调用 `net_rx` 函数处理接收到的数据包，并分配新的 `mbuf`。
4. 更新描述符地址和状态，更新 `tail` 位置。

```

1  static void
2  e1000_rcv(void)
3  {
4      //
5      // Your code here.
6      //
7      // Check for packets that have arrived from the e1000
8      // Create and deliver an mbuf for each packet (using net_rx()).
9      //
10     while(1){
11         uint idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;

```

```

12
13     struct rx_desc *desc = &rx_ring[idx];
14     if(!(desc->status & E1000_RXD_STAT_DD)){
15         return;
16     }
17     rx_mbufs[idx]->len = desc->length;
18
19     net_rx(rx_mbufs[idx]);
20
21     rx_mbufs[idx] = mbufalloc(0);
22     desc->addr = rx_mbufs[idx]->head;
23     desc->status = 0;
24
25     regs[E1000_RDT] = idx;
26 }
27
28 }
29

```

## 实验结果

```

== 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
(base) sheerio@ubuntu:~/xv6/lab7 net/xv6-labs-2021$

```

## 实验中遇到的问题及解决方法

### 1. 缺乏描述符空间

**问题：**在发送函数中，如果描述符队列已满，即所有描述符都没有设置DD标志位，无法继续发送数据。

**解决方法：**在发送函数 `e1000_transmit` 中，检查描述符的DD标志位。如果队列已满，直接返回错误。

```

1  if (!(desc->status & E1000_TXD_STAT_DD)) {
2      release(&e1000_lock);
3      return -1;
4  }

```

### 2. 多线程访问冲突

**问题：**多个线程同时访问E1000的寄存器和描述符队列，导致数据竞争。

**解决方法：**使用锁（如 `e1000_lock`）来确保同时只有一个线程在访问和修改寄存器和描述符队列。

```

1  acquire(&e1000_lock);
2  // critical section
3  release(&e1000_lock);

```

### 3. 内存分配失败

**问题：**在接收函数中，为描述符分配新的 mbuf 时，可能会发生内存分配失败。

**解决方法：**在每次分配 mbuf 后检查返回值，如果分配失败，记录错误并处理。

```
1 rx_mbufs[idx] = mbufalloc(0);
2 if (!rx_mbufs[idx]) {
3     panic("e1000: mbuf alloc failed");
4 }
```

## 实验心得

通过本次实验，我对网络编程特别是网卡驱动的开发有了更加深入的理解和实践经验。以下是我在实验过程中的一些体会：

#### 1. 深入理解DMA和内存映射：

- DMA技术的应用使得网卡可以直接访问主内存，提高了数据传输的效率。通过内存映射访问控制寄存器，使得编程更加灵活。

#### 2. 描述符的使用和管理：

- 描述符是网卡与主机之间进行数据交换的核心。通过环形队列管理描述符，可以有效地利用内存资源，提高数据传输的效率。

#### 3. 多线程编程中的同步问题：

- 在实现网卡驱动时，多线程的访问冲突问题需要通过锁机制来解决，确保数据的一致性和正确性。

## lab8: Locks

### 实验总览

在多核环境中，锁的竞争是一个主要的性能瓶颈。锁竞争会导致处理器在等待锁时浪费时间，从而降低系统的整体性能。通过对锁的优化和设计，可以有效地减少锁竞争，提高系统的并发性能。本次实验的目标是通过修改内存分配器和缓冲区缓存的设计，减少锁竞争，提高多核系统中的性能。

```
1 git checkout lock
```

## Memory Allocator

### 实验目的

本实验的目标是优化多核系统中的内存分配器，以降低锁的竞争和提高系统性能。具体措施包括为每个CPU核心实现一个独立的自由列表（free list），并为每个列表配置独自の锁。此外，当某个CPU的自由列表耗尽时，实验还需探索如何从其他CPU的自由列表中有效地借用内存页。



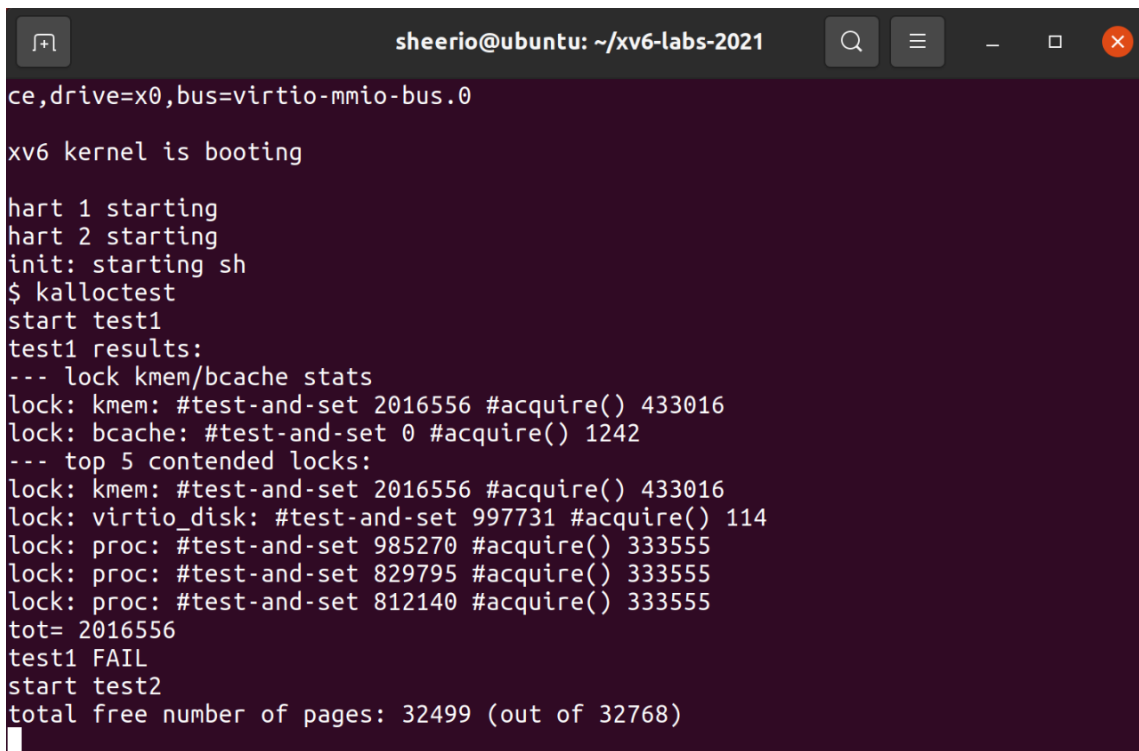
## 实验原理

在多核处理器系统中，所有核心共用一个内存分配器和其锁会导致严重的锁竞争，从而降低系统性能。通过为每个核心分配独立的内存分配列表和锁，可以使各核心在执行内存分配和释放操作时互不干扰，显著减少锁竞争，提高系统效率。

## 实验步骤及分析

### 1. 分析现有内存分配器问题：

- 在实验前，通过运行 `kalloctest` 测试程序，分析了现有内存分配器的性能问题。结果显示，内存分配器中的全局锁 `kmem.lock` 出现了大量的锁竞争。
- 测试显示，大量的 `#test-and-set` 和 `#acquire()` 操作频繁发生，指出了明显的锁争用问题。



```
sheerio@ubuntu: ~/xv6-labs-2021
ce,drive=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 2016556 #acquire() 433016
lock: bcache: #test-and-set 0 #acquire() 1242
--- top 5 contended locks:
lock: kmem: #test-and-set 2016556 #acquire() 433016
lock: virtio_disk: #test-and-set 997731 #acquire() 114
lock: proc: #test-and-set 985270 #acquire() 333555
lock: proc: #test-and-set 829795 #acquire() 333555
lock: proc: #test-and-set 812140 #acquire() 333555
tot= 2016556
test1 FAIL
start test2
total free number of pages: 32499 (out of 32768)
```

### 2. 重构内存分配器设计：

- 分析 `kalloc.c` 和 `kalloc.h` 中的内存分配器代码。
- 修改内存分配器的结构，为每个CPU创建独立的自由列表和锁。
- 在 `param.h` 中定义的 `NCPU` 参数基础上，结构体 `kmem` 被重构为包含多个自由列表和锁的数组。

```
1 struct {
2     struct spinlock lock;
3     struct run *freelist; // 空闲资源链表
4     char lock_name[7];
5 } kmem[NCPU]; // 修改成数组形式
```

### 3. 实现内存分配优化：

- 每个CPU在执行内存分配或释放时，只操作自己的自由列表和锁，减少了与其他CPU的锁竞争。
- 实现了“借用”机制，允许一个CPU在自己的列表耗尽时，尝试从其他CPU的自由列表中借用内存页。

#### 4. 处理内存借用逻辑:

- 设计一个循环遍历所有CPU, 从其他CPU的自由列表中借用内存页。
- 为每个操作加锁, 确保在修改列表时不会产生数据竞争。

#### 5. 代码修改与测试:

- 调整 `kalloc.c` 中的函数, 使之适应新的内存分配器结构。

```
1 void
2 kinit()
3 {
4     for (int idx = 0; idx < NCPU; idx++) {
5         snprintf(kmem[idx].lock_name, sizeof(kmem[idx].lock_name),
6             "kmem_%d", idx);
7         initlock(&kmem[idx].lock, kmem[idx].lock_name); // 初始化锁
8     }
9     freerange(end, (void*)PHYSTOP);
10 }
11 void
12 freerange(void *start_addr, void *end_addr)
13 {
14     char *mem_ptr;
15     mem_ptr = (char*)PGROUNDUP((uint64)start_addr);
16     for (; mem_ptr + PGSIZE <= (char*)end_addr; mem_ptr += PGSIZE)
17         kfree(mem_ptr);
18 }
19
20 void
21 kfree(void *phys_addr)
22 {
23     struct run *node;
24     if(((uint64)phys_addr % PGSIZE) != 0 || (char*)phys_addr < end ||
25         (uint64)phys_addr >= PHYSTOP)
26         panic("kfree"); // 检查释放的内存块是否合法
27     memset(phys_addr, 1, PGSIZE);
28     node = (struct run*)phys_addr;
29     push_off();
30     int cpu_id = cpuid(); // 获取当前CPU的ID
31
32     acquire(&kmem[cpu_id].lock); // 获取锁
33     node->next = kmem[cpu_id].freelist;
34     kmem[cpu_id].freelist = node;
35     release(&kmem[cpu_id].lock); // 释放锁
36
37     pop_off();
38 }
39 void *
40 kalloc(void)
41 {
42     struct run *node;
43
44     push_off();
45     int cpu_id = cpuid();
```

```

46
47     acquire(&kmem[cpu_id].lock);
48     node = kmem[cpu_id].freelist; // 将当前CPU空闲资源链表的头节点赋值给
node
49     if(node) {
50         kmem[cpu_id].freelist = node->next;
51     }
52     else {
53         int success = 0;
54         for(int i = 0; i < NCPU; i++) {
55             if (i == cpu_id) continue;
56             acquire(&kmem[i].lock);
57             struct run *temp = kmem[i].freelist;
58             if(temp) {
59                 struct run *half_node = temp;
60                 struct run *prev = temp;
61                 while (half_node && half_node->next) {
62                     half_node = half_node->next->next;
63                     prev = temp;
64                     temp = temp->next;
65                 }
66                 kmem[cpu_id].freelist = kmem[i].freelist; // 将窃取的一半内存分
配给当前CPU
67                 if (temp == kmem[i].freelist) {
68                     kmem[i].freelist = 0;
69                 }
70                 else {
71                     kmem[i].freelist = temp; // 更新其他CPU空闲资源链表的头指针
72                     prev->next = 0;
73                 }
74                 success = 1;
75             }
76             release(&kmem[i].lock);
77             if (success) {
78                 node = kmem[cpu_id].freelist;
79                 kmem[cpu_id].freelist = node->next; // 更新当前CPU空闲资源链表的
头指针
80                 break;
81             }
82         }
83     }
84     release(&kmem[cpu_id].lock);
85     pop_off();
86
87     if(node)
88         memset((char*)node, 5, PGSIZE);
89     return (void*)node;
90 }
91

```

- 重新运行 `kalloctest`，观察锁竞争的减少情况。测试结果显示锁的争用大幅减少，验证了优化的有效性。

## 6. 完整性验证:

- 运行 `usertests_sbrkmuch` 和 `usertests`，确保内存分配器的修改不影响系统的其他功能和稳定性。

## 实验现象

通过为每个CPU实现独立的内存分配器，显著降低了锁的竞争，增强了系统的并行处理能力。测试表明，在高并发情况下，内存分配和释放的效率得到了提升。此外，通过借用机制保持了系统的灵活性和高效性，使得内存资源得到了更合理的利用。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$
```

```
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 293
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 18
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 18
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 18
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 18
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 18
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 21
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 30
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 28
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 31
lock: bcache_eviction: #fetch-and-add 0 #acquire() 18
--- top 5 contended locks:
lock: proc: #fetch-and-add 342619 #acquire() 174113
lock: proc: #fetch-and-add 244299 #acquire() 174112
lock: proc: #fetch-and-add 215340 #acquire() 174112
lock: proc: #fetch-and-add 205113 #acquire() 174112
lock: proc: #fetch-and-add 194741 #acquire() 174113
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
$
```

## Buffer Cache

### 实验目的

本实验的目的是优化 xv6 操作系统中的缓冲区缓存（buffer cache），以减少多进程间对缓冲区缓存锁的竞争，从而提高系统的性能和并发能力。实验中，设计和实现了一种高效的缓冲区管理机制，使得不同的进程可以更有效地使用和管理缓冲区，减少锁竞争和性能瓶颈。

## 实验原理

缓冲区缓存是操作系统用于管理磁盘块和内存之间交换数据的一种机制。通过缓存频繁访问的磁盘块，系统可以减少直接磁盘访问次数，从而加快数据访问速度。然而，多进程环境下对同一缓冲区缓存的竞争会严重影响系统性能。为此，本实验通过引入哈希分桶机制来改进缓冲区管理，每个桶独立管理一部分缓冲区，并拥有自己的锁，这样可以显著减少锁的竞争。

## 实验步骤

### 1. 缓冲区缓存了解与问题分析：

- 通过阅读文档和源代码，深入理解了 xv6 中的缓冲区缓存的工作原理和数据结构。
- 运行 `bcachetest` 测试程序，观察并分析缓冲区缓存的锁竞争情况。测试显示，多个进程在尝试获取 `bcache.lock` 时产生了高频的锁竞争，影响了系统性能。

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 33036
lock: bcache: #test-and-set 412483 #acquire() 65022
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 4755734 #acquire() 1136
lock: proc: #test-and-set 701251 #acquire() 170124
lock: proc: #test-and-set 633129 #acquire() 170124
lock: proc: #test-and-set 549093 #acquire() 170124
lock: proc: #test-and-set 495733 #acquire() 170126
tot= 412483
test0: FAIL
start test1
test1 OK
$
```

### 2. 缓冲区管理机制优化设计：

- 引入 `struct bucket` 结构体，用于实现缓冲区的哈希分桶机制，每个桶包含一个链表存储缓冲区，并通过 `struct spinlock` 锁保护。
- 使用哈希函数将缓冲区映射到不同的分桶中，减少对整个缓冲区数组的搜索，提高访问效率。

```
1 struct bucket {
2     struct spinlock lock;
3     struct buf head;
4 };
5
6 struct {
7     struct buf buf[NBUF]; // 缓冲区数组
8     struct spinlock eviction_lock; // 自旋锁，保护缓冲区
9     struct buf bufmap[NBUFMAP_BUCKET]; // 缓冲区桶
10    struct spinlock bufmap_locks[NBUFMAP_BUCKET]; // 桶的锁
11 } bcache; // 块缓存
```

### 3. 缓冲区数据结构初始化:

- 初始化全局 `bcache` 结构体, 包括 `buf` 数组和 `bucket` 数组。
- 对每个缓冲区和每个分桶进行初始化, 确保每个分桶的自旋锁和链表头正确设置。

```
1 void binit(void) {
2     for(int i = 0; i < NBUFMAP_BUCKET; i++) {
3         initlock(&bcache.bufmap_locks[i], "bcache_bufmap"); // 初始化bufmap的
        锁
4         bcache.bufmap[i].next = 0; // 初始化bufmap的next字段
5     }
6
7     for(int i = 0; i < NBUF; i++){
8         struct buf *b = &bcache.buf[i];
9         initsleeplock(&b->lock, "buffer"); // 初始化缓冲区的锁
10        b->lastuse = 0; // 初始化lastuse字段
11        b->refcnt = 0;
12        b->next = bcache.bufmap[0].next;
13        bcache.bufmap[0].next = b;
14    }
15
16    initlock(&bcache.eviction_lock, "bcache_eviction"); // 初始化eviction锁
17 }
18
```

### 4. 缓冲区获取与释放逻辑实现:

- 实现 `bget` 函数, 使用哈希函数定位缓冲区所属分桶, 通过自旋锁同步访问, 减少锁争用。

```
1 // 查找或分配一个锁定的缓冲区
2 static struct buf* bget(uint dev, uint blockno) {
3     struct buf *b;
4     uint key = BUFMAP_HASH(dev, blockno); // 计算哈希值确定索引
5
6     acquire(&bcache.bufmap_locks[key]); // 获取bufmap锁
7
8     for(b = bcache.bufmap[key].next; b; b = b->next) {
9         if(b->dev == dev && b->blockno == blockno) {
10            b->refcnt++; // 引用计数加1
11            release(&bcache.bufmap_locks[key]); // 释放锁
12            acquiresleep(&b->lock); // 获取缓冲区锁
13            return b; // 返回缓冲区
14        }
15    }
16    release(&bcache.bufmap_locks[key]); // 释放锁
17    acquire(&bcache.eviction_lock); // 获取eviction锁
18
19    for(b = bcache.bufmap[key].next; b; b = b->next) {
20        if(b->dev == dev && b->blockno == blockno) {
21            acquire(&bcache.bufmap_locks[key]); // 获取bufmap锁
22            b->refcnt++;
23            release(&bcache.bufmap_locks[key]); // 释放锁
24            release(&bcache.eviction_lock); // 释放eviction锁
25            acquiresleep(&b->lock); // 获取缓冲区锁

```

```

26     return b;
27 }
28 }
29
30 struct buf *least_recently_used = 0;
31 uint holding_bucket = -1;
32
33 for(int i = 0; i < NBUFMAP_BUCKET; i++){
34     acquire(&bcache.bufmap_locks[i]); // 获取桶锁
35     int found = 0;
36     for(b = &bcache.bufmap[i]; b->next; b = b->next) {
37         if(b->next->refcnt == 0 && (!least_recently_used || b->next-
38 >lastuse < least_recently_used->next->lastuse)) {
39             least_recently_used = b;
40             found = 1;
41         }
42     }
43     if(!found) {
44         release(&bcache.bufmap_locks[i]);
45     } else {
46         if(holding_bucket != -1)
47             release(&bcache.bufmap_locks[holding_bucket]);
48         holding_bucket = i;
49     }
50 }
51
52 if(!least_recently_used) {
53     panic("bget: no buffers");
54 }
55 b = least_recently_used->next;
56
57 if(holding_bucket != key) {
58     least_recently_used->next = b->next;
59     release(&bcache.bufmap_locks[holding_bucket]);
60     acquire(&bcache.bufmap_locks[key]);
61     b->next = bcache.bufmap[key].next;
62     bcache.bufmap[key].next = b;
63 }
64
65 b->dev = dev;
66 b->blockno = blockno;
67 b->refcnt = 1;
68 b->valid = 0;
69 release(&bcache.bufmap_locks[key]);
70 release(&bcache.eviction_lock);
71 acquiresleep(&b->lock);
72 return b;
73 }

```

- 在缓冲区未被缓存时，选择一个未使用的缓冲区进行分配，并更新链表头表示最近使用。
- 实现缓冲区释放逻辑，只有拥有休眠锁的线程能释放缓冲区，通过原子操作清除 "used" 标志。

```

1 // 返回一个包含指定块内容的锁定缓冲区
2 struct buf* bread(uint dev, uint blockno) {
3     struct buf *b;
4     b = bget(dev, blockno);
5     if(!b->valid) {
6         virtio_disk_rw(b, 0);
7         b->valid = 1;
8     }
9     return b;
10 }
11
12 // 将缓冲区内容写入磁盘
13 void bwrite(struct buf *b) {
14     if(!holdingsleep(&b->lock))
15         panic("bwrite");
16     virtio_disk_rw(b, 1);
17 }
18
19 // 释放一个被锁定的缓冲区
20 void brelease(struct buf *b) {
21     if(!holdingsleep(&b->lock))
22         panic("brelease");
23
24     releasesleep(&b->lock); // 释放锁
25
26     uint key = BUFMAP_HASH(b->dev, b->blockno); // 计算哈希值
27     acquire(&bcache.bufmap_locks[key]); // 获取bufmap锁
28     b->refcnt--; // 引用计数减1
29     if (b->refcnt == 0) {
30         b->lastuse = ticks; // 更新最后使用时间
31     }
32     release(&bcache.bufmap_locks[key]); // 释放锁
33 }
34
35 // 将缓冲区锁定，避免回收
36 void bpin(struct buf *b) {
37     uint key = BUFMAP_HASH(b->dev, b->blockno);
38     acquire(&bcache.bufmap_locks[key]);
39     b->refcnt++;
40     release(&bcache.bufmap_locks[key]);
41 }
42
43 // 取消对缓冲区的锁定，允许回收
44 void bunpin(struct buf *b) {
45     uint key = BUFMAP_HASH(b->dev, b->blockno);
46     acquire(&bcache.bufmap_locks[key]);
47     b->refcnt--;
48     release(&bcache.bufmap_locks[key]);
49 }

```

## 5. 锁竞争与性能测试:

- 重新运行 `bcachetest` 测试程序，观察优化后的锁竞争情况。测试结果表明，锁竞争显著减少，系统响应速度得到改善。
- 运行 `usertests` 确保所有用户级测试通过，验证缓冲区管理修改的正确性和稳定性。



## 实验现象

优化后的缓冲区缓存系统在并发访问时展现出更低的锁竞争和更高的性能。通过将缓冲区分布到多个分桶中，每个分桶实施独立的锁管理，有效降低了进程间的锁争用。测试结果证实了优化措施的有效性，显示出系统处理速度的提升和响应时间的缩短。

```
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 6719
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 6729
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 7041
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 6305
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 6299
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4235
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4235
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 2227
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4238
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 2382
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4391
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4437
lock: bcache_eviction: #fetch-and-add 0 #acquire() 110
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 5902487 #acquire() 1219
lock: proc: #fetch-and-add 887474 #acquire() 104884
lock: proc: #fetch-and-add 458867 #acquire() 104553
lock: proc: #fetch-and-add 433400 #acquire() 104534
lock: proc: #fetch-and-add 406206 #acquire() 104534
tot= 0
test0: OK
start test1
test1 OK
$
```

## 实验结果

```
sheerio@ubuntu: ~/xv6/lab8 lock/xv6-labs-2021
== Test running kallocetest ==
$ make qemu-gdb
(241.3s)
== Test kallocetest: test1 ==
kallocetest: test1: OK
== Test kallocetest: test2 ==
kallocetest: test2: OK
== Test kallocetest: sbrkmuch ==
$ make qemu-gdb
kallocetest: sbrkmuch: OK (31.1s)
== Test running bcachetest ==
$ make qemu-gdb
(79.6s)
== Test bcachetest: test0 ==
bcachetest: test0: OK
== Test bcachetest: test1 ==
bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (440.5s)
== Test time ==
time: OK
Score: 70/70
(base) sheerio@ubuntu:~/xv6/lab8 lock/xv6-labs-2021$ S
```

## 实验中遇到的问题及解决方法

### 问题1: 锁争用高导致性能下降

在 **Memory Allocator** 实验中, 通过 `kalloc_test` 测试发现, 大量的锁请求导致高度锁争用。这种现象同样在 **Buffer Cache** 实验的 `bcache_test` 中观察到, 多个进程频繁尝试获取同一锁, 造成性能瓶颈。

解决方法:

- **Memory Allocator:** 引入每 CPU 独立的自由列表和锁机制, 使不同 CPU 上的线程可以并行执行内存分配和释放, 极大减少锁争用。
- **Buffer Cache:** 实施了缓冲区的哈希分桶机制, 每个分桶拥有自己的锁。这种设计降低了不同进程间的锁竞争, 提高了访问效率。

### 问题2: 内存或资源不足

在进行 **Memory Allocator** 实验时, 若一个 CPU 的自由列表耗尽, 需要从其他 CPU 的列表中借用内存页面, 初步实现中存在效率问题。

解决方法:

- 实现了一种高效的“借用”机制, 允许 CPUs 在缺页时动态地从其他 CPU 的自由列表中获取页面。通过优化借用策略, 确保借用操作不会因频繁的锁请求而影响性能。

### 问题3: 并发一致性问题

在 **Buffer Cache** 实验的实现中, 原初设计未能有效处理多线程环境下的数据一致性, 导致缓冲区管理出现资源分配冲突和竞态条件。

解决方法:

- 引入细粒度的锁机制, 为每个分桶引入自旋锁, 确保操作缓冲区时的线程安全。同时, 使用休眠锁同步对缓冲区数据的访问, 保证只有一个线程可以操作特定的缓冲区。

## 实验心得

通过参与内存分配器和缓冲区缓存的优化实验, 我不仅提升了对操作系统内核管理资源和处理并发的理解, 还深入体会了理论与实践的结合如何揭示系统性能瓶颈。这些实验中遇到的并发控制挑战, 如锁竞争和数据一致性问题, 加强了我的问题解析和解决技巧, 尤其是在应用细粒度锁和其他同步机制来保证系统稳定性方面。设计权衡的实践, 例如在减少锁竞争与最大化性能之间找到平衡, 以及与同伴的交流, 极大地丰富了我的学习经验。

## lab 9: File System

### 实验总览

本实验涉及对xv6文件系统的两个主要扩展: 支持更大的文件和实现符号链接。实验要求阅读相关的文件系统章节, 并修改文件系统的代码以支持新的功能。

1. **大文件支持:** 扩展xv6文件系统, 使其能够支持超过当前限制的文件大小。原系统中文件大小最大为  $268 * 1024$  字节, 由inode中的12个直接块和1个单间接块组成。实验中将实现双间接块, 大幅提高文件最大大小。
2. **符号链接:** 在xv6中实现符号链接功能, 允许创建指向其他文件或目录的链接, 支持跨文件系统的引用。

```
1 | git checkout fs
```

## Large Files

### 实验目的

本实验部分的目标是扩展xv6文件系统以支持更大的文件。由于原有的xv6文件系统中文件大小受限于268个磁盘块，这是因为每个inode只能直接引用12个块和一个单间接块，后者又可以引用256个块。为了扩大文件大小的限制，引入了双间接块的概念。

### 实验步骤

1. **修改inode结构**：首先在 `kernel/fs.h` 中的 `struct dinode` 定义中修改，减少直接块的数量从12到11，并增加一个双间接块的引用。

```
1 // On-disk inode structure
2 struct dinode {
3     short type;           // File type
4     short major;          // Major device number (T_DEVICE only)
5     short minor;          // Minor device number (T_DEVICE only)
6     short nlink;          // Number of links to inode in file system
7     uint size;            // Size of file (bytes)
8     uint addrs[NDIRECT+2]; // Data block addresses
9 };
```

2. **更新宏定义**：

- `NDIRECT` 减少到11。
- `NINDIRECT` 保持不变，表示单间接块可以引用的块数量。
- 新增 `NDBL_INDIRECT` 表示通过双间接块可以引用的最大块数 ( $256 * 256$ )。
- `MAXFILE` 更新为 `NDIRECT + NINDIRECT + NDBL_INDIRECT`。

```
1 #define NDIRECT 11
2 #define NINDIRECT (BSIZE / sizeof(uint))
3 #define NDBL_INDIRECT (NINDIRECT * NINDIRECT)
4 #define MAXFILE (NDIRECT + NINDIRECT + NDBL_INDIRECT)
```

3. **修改地址数组**：在 `struct dinode` 和 `struct inode` 中的 `addrs[]` 数组，调整数组大小以适应新的双间接块。这需要保证数组有足够的空间存储额外的块引用。

```
1 struct dinode {
2     short type;           // File type
3     ...
4     uint addrs[NDIRECT+2]; // Data block addresses
5 };
6
7 struct inode {
```

```

8  uint dev;           // Device number
9  uint inum;          // Inode number
10 int ref;            // Reference count
11 struct sleeplock lock; // protects everything below here
12 int valid;          // inode has been read from disk?
13
14 short type;         // copy of disk inode
15 short major;
16 short minor;
17 short nlink;
18 uint size;
19 uint addrs[NDIRECT+2];
20 };

```

4. **修改bmap()函数**：在kernel/fs.c中调整此函数以支持通过双间接块映射。在处理超过单间接块的块号时，需要计算双间接块的映射。

```

1  static uint
2  bmap(struct inode *ip, uint bn)
3  {
4      uint addr, *a;
5      struct buf *bp;
6
7      if(bn < NDIRECT){
8          if((addr = ip->addrs[bn]) == 0)
9              ip->addrs[bn] = addr = balloc(ip->dev);
10         return addr;
11     }
12     bn -= NDIRECT;
13
14     if(bn < NINDIRECT){
15         // Load indirect block, allocating if necessary.
16         if((addr = ip->addrs[NDIRECT]) == 0)
17             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
18         bp = bread(ip->dev, addr);
19         a = (uint*)bp->data;
20         if((addr = a[bn]) == 0){
21             a[bn] = addr = balloc(ip->dev);
22             log_write(bp);
23         }
24         brelse(bp);
25         return addr;
26     }
27
28     bn -= NINDIRECT;
29     if (bn < NDBL_INDIRECT) {
30         if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
31             addr = balloc(ip->dev);
32             if (addr == 0)
33                 return 0;
34             ip->addrs[NDIRECT + 1] = addr;
35         }
36
37         // 读取双间接块

```

```

38     bp = bread(ip->dev, addr);
39     a = (uint*)bp->data;
40
41     // 计算在单间接块数组中的索引
42     uint index1 = bn / NINDIRECT;
43
44     if ((addr = a[index1]) == 0) {
45         addr = balloc(ip->dev);
46         if (addr == 0)
47             return 0;
48         a[bn / NINDIRECT] = addr;
49         log_write(bp); // Record changes in the log
50     }
51     brelse(bp);
52     bp = bread(ip->dev, addr);
53     a = (uint *)bp->data;
54
55     uint index2 = bn % NINDIRECT;
56
57     // 如果这个数据块不存在，则分配一个
58     if ((addr = a[index2]) == 0) {
59         addr = balloc(ip->dev);
60         if (addr == 0)
61             return 0;
62         a[bn % NINDIRECT] = addr;
63         log_write(bp); // Record changes in the log
64     }
65     brelse(bp);
66     return addr; // Returns the actual data block
67 }
68
69 panic("bmap: out of range");
70 }

```

5. **更新文件截断函数itrunc()**: 确保能正确释放包括双间接块在内的所有块。需要遍历双间接块引用的所有单间接块，并释放它们引用的所有直接块。

```

1  void
2  itrunc(struct inode *ip)
3  {
4      int i, j;
5      struct buf *bp;
6      uint *a;
7
8      for(i = 0; i < NDIRECT; i++){
9          if(ip->addrs[i]){
10             bfree(ip->dev, ip->addrs[i]);
11             ip->addrs[i] = 0;
12         }
13     }
14
15     if(ip->addrs[NDIRECT]){
16         bp = bread(ip->dev, ip->addrs[NDIRECT]);
17         a = (uint*)bp->data;

```

```

18     for(j = 0; j < NINDIRECT; j++){
19         if(a[j])
20             bfree(ip->dev, a[j]);
21     }
22     brelse(bp);
23     bfree(ip->dev, ip->addrs[NDIRECT]);
24     ip->addrs[NDIRECT] = 0;
25 }
26
27 if (ip->addrs[NDIRECT + 1]) {
28     // 读取双间接块
29     bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
30     a = (uint*)bp->data;
31
32     for (i = 0; i < NINDIRECT; ++i) {
33         if (a[i] == 0) continue;
34
35         // 读取单间接块
36         struct buf* bp2 = bread(ip->dev, a[i]);
37         uint* b = (uint*)bp2->data;
38         for (j = 0; j < NINDIRECT; ++j) {
39             if (b[j])
40                 bfree(ip->dev, b[j]);
41         }
42         brelse(bp2);
43
44         bfree(ip->dev, a[i]);
45         a[i] = 0;
46     }
47     brelse(bp);
48
49     bfree(ip->dev, ip->addrs[NDIRECT + 1]);
50     ip->addrs[NDIRECT + 1] = 0;
51 }
52
53 ip->size = 0;
54 iupdate(ip);
55 }

```

6. 确保文件写操作正确使用**bmap()**: 在 `filewrite()` 函数中, 调用修改后的 `bmap()` 以正确处理大文件的写入。

```

1  int
2  filewrite(struct file *f, uint64 addr, int n)
3  {
4      int r, ret = 0;
5
6      if(f->writable == 0)
7          return -1;
8
9      if(f->type == FD_PIPE){
10         ret = pipewrite(f->pipe, addr, n);
11     } else if(f->type == FD_DEVICE){
12         if(f->major < 0 || f->major >= NDEV || !devsw[f->major].write)

```

```

13     return -1;
14     ret = devsw[f->major].write(1, addr, n);
15 } else if(f->type == FD_INODE){
16     // write a few blocks at a time to avoid exceeding
17     // the maximum log transaction size, including
18     // i-node, indirect block, allocation blocks,
19     // and 2 blocks of slop for non-aligned writes.
20     // this really belongs lower down, since writei()
21     // might be writing a device like the console.
22     int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;
23     int i = 0;
24     while(i < n){
25         int n1 = n - i;
26         if(n1 > max)
27             n1 = max;
28
29         begin_op();
30         ilock(f->ip);
31         if ((r = writei(f->ip, 1, addr + i, f->off, n1)) > 0)
32             f->off += r;
33         iunlock(f->ip);
34         end_op();
35
36         if(r != n1){
37             // error from writei
38             break;
39         }
40         i += r;
41     }
42     ret = (i == n ? n : -1);
43 } else {
44     panic("filewrite");
45 }
46
47 return ret;
48 }

```

7. **测试**：运行 `bigfile` 测试以验证系统能够正确创建并处理大文件。此外，运行 `usertests` 测试套件确保所有其他功能仍正常工作。

## Symbolic Links

### 实验目的

本实验部分旨在 xv6 操作系统中实现符号链接（软链接）功能。符号链接允许通过路径名引用其他文件，不同于硬链接，它们能够跨越不同的文件系统。

### 实验步骤

#### 1. 系统调用的添加：

- 在 `kernel/syscall.h`、`kernel/syscall.c`、`user/usys.pl` 和 `user/user.h` 中添加关于 `symlink` 系统调用的必要声明和定义。

```
1 | #define SYS_symlink 22
```

```
1 | extern uint64 SYS_symlink(void);
2 | static uint64 (*syscalls[])(void) = {
3 |     ...
4 |     [SYS_symlink] sys_symlink,
5 | };
```

```
1 | entry("symlink"); 1
```

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

## 2. 文件类型定义:

- 在 kernel/stat.h 中定义新的文件类型 T\_SYMLINK，用于标识符号链接。

```
1 | #define T_SYMLINK 4
```

## 3. 打开标志添加:

- 在 kernel/fcntl.h 中添加 O\_NOFOLLOW 打开标志，以允许 open 系统调用在需要时不跟随链接。

```
1 | #define O_NOFOLLOW 0x004
```

## 4. 实现 sys\_symlink 函数:

- 在 kernel/sysfile.c 中，实现 sys\_symlink 函数以创建符号链接。这包括使用 create() 函数创建符号链接的inode，并使用 writei() 将目标文件路径写入该inode。

```
1 | uint64
2 | sys_symlink(void) {
3 |     char target[MAXPATH], path[MAXPATH];
4 |     struct inode *ip;
5 |     int n;
6 |
7 |     if ((n = argstr(0, target, MAXPATH)) < 0
8 |         || argstr(1, path, MAXPATH) < 0) {
9 |         return -1;
10 |    }
11 |
12 |    begin_op();
13 |    if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
14 |        end_op();
15 |        return -1;
16 |    }
17 |    if(writei(ip, 0, (uint64)target, 0, n) != n) {
18 |        iunlockput(ip);
19 |        end_op();
20 |        return -1;
21 |    }
22 |
23 |    iunlockput(ip);
```



```

24     end_op();
25     return 0;
26 }

```

## 5. 修改 open 系统调用：

- 调整 `sys_open` 以正确处理符号链接。特别是，添加对 `O_NOFOLLOW` 的支持，使其在指定时不跟随符号链接。

```

1  uint64
2  sys_open(void)
3  {
4      char path[MAXPATH];
5      int fd, omode;
6      struct file *f;
7      struct inode *ip;
8      int n;
9
10     if((n = argstr(0, path, MAXPATH)) < 0 || argint(1, &omode) < 0)
11         return -1;
12
13     begin_op();
14
15     if(omode & O_CREATE){
16         ip = create(path, T_FILE, 0, 0);
17         if(ip == 0){
18             end_op();
19             return -1;
20         }
21     } else {
22         if((ip = namei(path)) == 0){
23             end_op();
24             return -1;
25         }
26         ilock(ip);
27         if(ip->type == T_DIR && omode != O_RDONLY){
28             iunlockput(ip);
29             end_op();
30             return -1;
31         }
32     }
33
34     if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
35         iunlockput(ip);
36         end_op();
37         return -1;
38     }
39
40     // handle the symlink - lab 9.2
41     if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
42         if((ip = follow_symlink(ip)) == 0) {
43             end_op();
44             return -1;
45         }

```

```

46     }
47
48     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
49         if(f)
50             fileclose(f);
51         iunlockput(ip);
52         end_op();
53         return -1;
54     }
55
56     if(ip->type == T_DEVICE){
57         f->type = FD_DEVICE;
58         f->major = ip->major;
59     } else {
60         f->type = FD_INODE;
61         f->off = 0;
62     }
63     f->ip = ip;
64     f->readable = !(omode & O_WRONLY);
65     f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
66
67     if((omode & O_TRUNC) && ip->type == T_FILE){
68         itrunc(ip);
69     }
70
71     iunlock(ip);
72     end_op();
73
74     return fd;
75 }

```

## 6. 递归链接处理：

- 实现递归链接跟踪，并在符号链接深度过大或检测到环形链接时返回错误。

## 7. 测试和验证：

- 编译并运行 `symlinktest` 以测试符号链接功能。同时，运行 `usertests` 和其他相关测试确保系统其他部分未受影响。

## 实验结果

```
sheerio@ubuntu: ~/xv6/lab9 fs/xv6-labs-2021
lic.o kernel/virtio_disk.o kernel/start.o kernel/console.o kernel/printf.o kern
el/uart.o kernel/spinlock.o
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* /
/; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/sheerio/xv6/lab9 fs/xv6-labs-2021'
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (295.9s)
== Test running symlinktest ==
$ make qemu-gdb
(1.5s)
== Test symlinktest: symlinks ==
symlinktest: symlinks: OK
== Test symlinktest: concurrent symlinks ==
symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (504.1s)
== Test time ==
time: OK
Score: 100/100
```

## 实验中遇到的问题及解决方法

### Large Files

- 问题：** 在实现双间接块支持时，对数据结构的修改引入了数组边界的问题。
  - 解决方法：** 谨慎地修改了 `struct dinode` 中的 `addrs[]` 数组定义，确保不超过结构的总大小，并适当调整了直接和间接块的数量，以适配双间接块。
- 问题：** 在修改 `bmap()` 函数以支持双间接块时，处理块映射逻辑较为复杂。
  - 解决方法：** 通过详细地分层逻辑块号的计算，逐步实现了映射过程，从直接块到单间接块，再到双间接块。必要时，重构代码以简化逻辑并保证清晰。
- 问题：** 在释放文件块时，需要确保双间接块及其内容被正确释放。
  - 解决方法：** 在 `itrunc()` 函数中增加了对双间接块的处理，通过嵌套循环访问并释放所有关联的数据块，确保不会有内存泄漏或磁盘空间浪费。

### Symbolic Links

- 问题：** 符号链接的递归解析可能导致无限循环，特别是在符号链接成环时。
  - 解决方法：** 引入了深度限制（例如限制为10次解析），并通过记录每次解析的inode编号来检测循环。一旦检测到循环，立即终止解析并返回错误。
- 问题：** `sys_open()` 函数在处理符号链接时必须正确解析链接目标。
  - 解决方法：** 实现了 `follow_symlink()` 函数，该函数递归地解析符号链接并获取最终目标的inode。修改 `sys_open()` 以使用此函数获取正确的inode，而不是符号链接自身的inode。

## 实验心得

在这两个实验中，我深入了解和扩展了xv6操作系统的文件系统，特别是通过实现大文件支持和符号链接功能，显著增强了系统的功能性和灵活性。

1. **大文件实验** 让我理解了文件系统如何处理存储和检索数据的基本机制。通过实现双间接块，我学习到了如何扩展文件系统的存储容量，使其能够处理比以往更大的文件。这不仅需要对现有数据结构进行精确的修改，还要确保所有相关功能（如数据块映射和文件截断）都能适应新的结构。这个过程提高了我对文件系统架构深度和复杂性的认识。
2. **符号链接实验** 强化了我对文件系统中路径解析和文件链接概念的理解。通过实现符号链接，我学习了文件系统如何处理不同类型的文件节点，并探索了路径名解析的内部工作原理。这一功能的实现尤其强调了处理边界情况和异常的重要性，例如循环链接和深度限制，这些都是确保文件系统稳定性和可靠性的关键因素。

## lab 10: mmap

### 实验总览

#### 内存映射文件的基本概念

内存映射文件（Memory-mapped file）技术允许应用程序通过内存访问的方式来访问存储在硬盘上的文件。这种技术将文件内容映射到进程的地址空间中，实现文件和内存之间的直接映射。通过这种方式，文件数据可以像访问常规内存那样被访问，从而提高文件操作的效率，并简化编程模型。

#### 虚拟内存与内存映射文件

内存映射文件依赖于操作系统的虚拟内存系统。虚拟内存系统通过使用硬盘空间作为额外的虚拟内存来扩展物理内存，使得计算机可以运行大于物理内存的应用程序。当进程访问其虚拟地址空间中的某个地址时，如果该地址关联的数据不在物理内存中，操作系统将处理这一缺页（page fault），并将数据从硬盘加载到物理内存中。

#### mmap 和 munmap 系统调用

- **mmap 系统调用**：mmap 允许应用程序指定一个文件区域，并将其映射到进程的地址空间。应用程序可以指定映射的起始地址、文件的偏移量、映射的大小以及访问权限等。映射建立后，应用程序可以直接通过指针访问这部分内存，操作系统负责将文件数据同步到这段内存中。
- **munmap 系统调用**：与 mmap 相对应，munmap 用于撤销内存映射。当映射不再需要时，munmap 将释放相关的资源，并可能将修改过的数据写回文件。

#### 内存映射的操作流程

1. **映射创建**：当应用程序调用 mmap 时，系统不会立即分配物理内存或加载文件数据，而是记录映射的元数据，并在进程的地址空间中保留对应的虚拟内存区域。
2. **懒惰加载**：当程序首次访问映射区域的某部分时，会触发缺页中断。操作系统此时才会分配物理内存，并从文件中加载必要的文件数据。
3. **数据同步**：对映射区域的修改可以通过 msync 调用显式地同步回文件，或者在取消映射时隐式同步。
4. **映射撤销**：当调用 munmap 或进程终止时，映射会被撤销。如果映射是共享的并且被修改过，系统需要将变更写回文件。

```
1 | git checkout mmap
```

## 实验目的

本实验的目的是在 xv6 操作系统中引入 `mmap` 和 `munmap` 系统调用。这些功能允许将文件内容映射到进程的虚拟地址空间，从而实现内存映射文件功能。

## 实验步骤

### 1. 系统调用添加：

- 修改 `kernel/syscall.h` 和 `kernel/syscall.c`，引入 `mmap` 和 `munmap` 的相关定义和函数指针映射。

`kernel/syscall.h`：

```
1 | #define SYS_mmap    22
2 | #define SYS_munmap  23
```

`kernel/syscall.c`

```
1 | extern uint64 sys_mmap(void);
2 | extern uint64 sys_munmap(void);
3 |
4 | static uint64 (*syscalls[])(void) = {
5 |     [SYS_fork]    sys_fork,
6 |     [SYS_exit]    sys_exit,
7 |     [SYS_wait]    sys_wait,
8 |     [SYS_pipe]    sys_pipe,
9 |     [SYS_read]    sys_read,
10 |    [SYS_kill]    sys_kill,
11 |    [SYS_exec]    sys_exec,
12 |    [SYS_fstat]   sys_fstat,
13 |    [SYS_chdir]   sys_chdir,
14 |    [SYS_dup]     sys_dup,
15 |    [SYS_getpid]  sys_getpid,
16 |    [SYS_sbrk]    sys_sbrk,
17 |    [SYS_sleep]   sys_sleep,
18 |    [SYS_uptime]  sys_uptime,
19 |    [SYS_open]    sys_open,
20 |    [SYS_write]   sys_write,
21 |    [SYS_mknod]   sys_mknod,
22 |    [SYS_unlink]  sys_unlink,
23 |    [SYS_link]    sys_link,
24 |    [SYS_mkdir]   sys_mkdir,
25 |    [SYS_close]   sys_close,
26 |    [SYS_mmap]    sys_mmap,
27 |    [SYS_munmap]  sys_munmap,
28 | };
```

- 更新 `user/usys.pl` 以自动生成用户空间的系统调用包装函数。

```

1 entry("mmap");
2 entry("munmap");

```

- 在 `user/user.h` 中添加用户接口函数原型，方便在用户程序中调用。

```

1 // ulib.c
2 int stat(const char*, struct stat*);
3 char* strcpy(char*, const char*);
4 void *memmove(void*, const void*, int);
5 char* strchr(const char*, char c);
6 int strcmp(const char*, const char*);
7 void fprintf(int, const char*, ...);
8 void printf(const char*, ...);
9 char* gets(char*, int max);
10 uint strlen(const char*);
11 void* memset(void*, int, uint);
12 void* malloc(uint);
13 void free(void*);
14 int atoi(const char*);
15 int memcmp(const void *, const void *, uint);
16 void *memcpy(void *, const void *, uint);
17 void *mmap(void *, int, int, int, int, int);
18 int munmap(void *, int);

```

## 2. 虚拟内存区域 (VMA) 结构定义：

- 在 `kernel/proc.h` 中定义 `struct vm_area`，用于描述由 `mmap` 创建的虚拟内存区域。此结构包括映射起始地址、长度、权限、映射标志、文件偏移量和关联的文件指针。

```

1 struct vm_area {
2     uint64 addr;
3     int len;
4     int prot;
5     int flags;
6     int offset;
7     struct file* f;
8 };

```

- 修改 `struct proc` 结构，为每个进程添加一个 VMA 数组，用于跟踪该进程的所有内存映射。

```

1 struct proc {
2     struct spinlock lock;
3
4     // p->lock must be held when using these:
5     enum procstate state;           // Process state
6     void *chan;                     // If non-zero, sleeping on chan
7     int killed;                     // If non-zero, have been killed
8     int xstate;                     // Exit status to be returned to
    parent's wait
9     int pid;                         // Process ID

```

```

10
11 // wait_lock must be held when using this:
12 struct proc *parent; // Parent process
13
14 // these are private to the process, so p->lock need not be held.
15 uint64 kstack; // Virtual address of kernel stack
16 uint64 sz; // Size of process memory (bytes)
17 pagetable_t pagetable; // User page table
18 struct trapframe *trapframe; // data page for trampoline.S
19 struct context context; // swtch() here to run process
20 struct file *ofile[NOFILE]; // Open files
21 struct inode *cwd; // Current directory
22 char name[16]; // Process name (debugging)
23 struct vm_area vma[NVMA];
24 };
25

```

### 3. 权限和映射标志实现:

- 在 `kernel/fcntl.h` 中定义了 `PROT_READ`, `PROT_WRITE` 以及 `MAP_SHARED` 和 `MAP_PRIVATE`。
- 这些定义控制映射的访问权限和映射的共享性, 决定了内存写入时的行为 (如写时复制)。

```

1 #define O_RDONLY 0x000
2 #define O_WRONLY 0x001
3 #define O_RDWR 0x002
4 #define O_CREATE 0x200
5 #define O_TRUNC 0x400
6
7 #ifndef LAB_MMAP
8 #define PROT_NONE 0x0
9 #define PROT_READ 0x1
10 #define PROT_WRITE 0x2
11 #define PROT_EXEC 0x4
12
13 #define MAP_SHARED 0x01
14 #define MAP_PRIVATE 0x02
15 #endif

```

### 4. 懒惰加载实现:

- `mmap` 调用时并不立即分配物理内存或从文件中加载数据, 而是在发生页面错误时才进行加载。
- 修改 `kernel/trap.c` 中的页面错误处理逻辑, 以便根据访问的虚拟地址动态加载数据。

### 5. 实现 `mmap` 系统调用:

- `sys_mmap()` 在适当的虚拟地址空间找到未使用的区域进行映射。
- 它更新进程的 VMA 表, 记录映射详细信息, 并在需要时处理文件引用和权限设置。

```

1 uint64
2 sys_mmap(void) {
3     uint64 addr;
4     int len, prot, flags, offset;
5     struct file *f;
6     struct vm_area *vma = 0;

```

```

7   struct proc *p = myproc();
8   int i;
9
10  if (argaddr(0, &addr) < 0 || argint(1, &len) < 0
11      || argint(2, &prot) < 0 || argint(3, &flags) < 0
12      || argfd(4, 0, &f) < 0 || argint(5, &offset) < 0) {
13      return -1;
14  }
15  if (flags != MAP_SHARED && flags != MAP_PRIVATE) {
16      return -1;
17  }
18  if (flags == MAP_SHARED && f->writable == 0 && (prot &
PROT_WRITE)) {
19      return -1;
20  }
21  if (len < 0 || offset < 0 || offset % PGSIZE) {
22      return -1;
23  }
24
25  for (i = 0; i < NVMA; ++i) {
26      if (!p->vma[i].addr) {
27          vma = &p->vma[i];
28          break;
29      }
30  }
31  if (!vma) {
32      return -1;
33  }
34
35  addr = MMAPMINADDR;
36  for (i = 0; i < NVMA; ++i) {
37      if (p->vma[i].addr) {
38          addr = max(addr, p->vma[i].addr + p->vma[i].len);
39      }
40  }
41  addr = PGROUNDUP(addr);
42  if (addr + len > TRAPFRAME) {
43      return -1;
44  }
45  vma->addr = addr;
46  vma->len = len;
47  vma->prot = prot;
48  vma->flags = flags;
49  vma->offset = offset;
50  vma->f = f;
51  filedup(f);
52
53  return addr;
54  }

```

#### 6. 页面错误处理:

- 对于通过 `mmap` 映射的内存区域发生的页面错误, 系统需要加载正确的文件内容到新分配的物理页面。
- 实现了将文件内容按需加载到内存中, 处理了权限和标志位的相关逻辑。

#### 7. 实现 `munmap` 系统调用:



- `sys_munmap()` 处理取消映射的请求，释放相应的物理内存并更新进程的 VMA 表。
- 如果映射是 `MAP_SHARED`，并且页面已修改，则需要将更改写回文件。

```
1  uint64
2  sys_munmap(void) {
3      uint64 addr, va;
4      int len;
5      struct proc *p = myproc();
6      struct vm_area *vma = 0;
7      uint maxsz, n, n1;
8      int i;
9
10     if (argaddr(0, &addr) < 0 || argint(1, &len) < 0) {
11         return -1;
12     }
13     if (addr % PGSIZE || len < 0) {
14         return -1;
15     }
16
17     for (i = 0; i < NVMA; ++i) {
18         if (p->vma[i].addr && addr >= p->vma[i].addr
19             && addr + len <= p->vma[i].addr + p->vma[i].len) {
20             vma = &p->vma[i];
21             break;
22         }
23     }
24     if (!vma) {
25         return -1;
26     }
27
28     if (len == 0) {
29         return 0;
30     }
31
32     if ((vma->flags & MAP_SHARED)) {
33         maxsz = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
34         for (va = addr; va < addr + len; va += PGSIZE) {
35             if (uvmgetdirty(p->pagetable, va) == 0) {
36                 continue;
37             }
38             n = min(PGSIZE, addr + len - va);
39             for (i = 0; i < n; i += n1) {
40                 n1 = min(maxsz, n - i);
41                 begin_op();
42                 ilock(vma->f->ip);
43                 if (writei(vma->f->ip, 1, va + i, va - vma->addr + vma-
44 >offset + i, n1) != n1) {
45                     iunlock(vma->f->ip);
46                     end_op();
47                     return -1;
48                 }
49                 iunlock(vma->f->ip);
50                 end_op();
51             }
52         }
53     }
```

```

52     }
53     uvmunmap(p->pagetable, addr, (len - 1) / PGSIZE + 1, 1);
54     if (addr == vma->addr && len == vma->len) {
55         vma->addr = 0;
56         vma->len = 0;
57         vma->offset = 0;
58         vma->flags = 0;
59         vma->prot = 0;
60         fclose(vma->f);
61         vma->f = 0;
62     } else if (addr == vma->addr) {
63         vma->addr += len;
64         vma->offset += len;
65         vma->len -= len;
66     } else if (addr + len == vma->addr + vma->len) {
67         vma->len -= len;
68     } else {
69         panic("unexpected munmap");
70     }
71     return 0;
72 }

```

## 8. 设置脏页标志位

### ◦ 定义脏页标志位:

- 在 `kernel/riscv.h` 中定义脏页标志位 `PTE_D`。该标志位用于表示页面已被修改，是对页面内容的修改标记。定义如下：

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

### ◦ 实现脏页检测和设置函数:

- 在 `kernel/vm.c` 中实现两个函数：`uvmgetdirty()` 和 `uvmsetdirtywrite()`。这两个函数分别用于读取和设置页表项的脏页标志。
  - `uvmgetdirty()`：检查指定虚拟地址对应的页表项是否标记为脏。
  - `uvmsetdirtywrite()`：设置指定虚拟地址的页表项的脏标志，通常在写操作发生时调用。

```

1 int uvmgetdirty(pagetable_t pagetable, uint64 va) {
2     pte_t *pte = walk(pagetable, va, 0);
3     if(pte == 0) {
4         return 0;
5     }
6     return (*pte & PTE_D);
7 }
8
9 int uvmsetdirtywrite(pagetable_t pagetable, uint64 va) {
10    pte_t *pte = walk(pagetable, va, 0);
11    if(pte == 0) {
12        return -1;
13    }
14    *pte |= PTE_D | PTE_W;
15    return 0;
16 }

```

- 在异常处理中应用脏页标志:
  - 修改 `kernel/trap.c` 中的 `usertrap()` 函数，增加对页错误 (Page Fault) 的处理。如果是因写入操作到未映射或只读页面导致的页错误，就设置脏页标志位。
    - 检查触发页错误的原因 (如 `r_scause()` 返回值是 12、13 或 15)，判断是否由写操作引起。
    - 根据 VMA (虚拟内存区域) 设置的权限，决定是否给相应的页表项添加 `PTE_D` 标志。
- 处理写操作引起的页错误:
  - 当进行写操作且页面未映射时，通过页错误懒惰分配 (Lazy Allocation) 新的物理页，并设置脏页标志位。使用 `kalloc()` 为新页分配内存，并用 `memset()` 初始化新分配的页。
- 在 `munmap` 中检查和处理脏页:
  - 在 `kernel/sysfile.c` 中的 `sys_munmap()` 函数中，使用 `uvmgetdirty()` 检查需要取消映射的内存页是否被修改过。
  - 如果是 `MAP_SHARED` 类型映射且页面被修改，则需要将数据写回映射的文件中，确保数据的一致性。

```

1  if ((vma->flags & MAP_SHARED)) {
2      maxsz = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
3      for (va = addr; va < addr + len; va += PGSIZE) {
4          if (uvmgetdirty(p->pagetable, va) == 0) {
5              continue;
6          }
7          n = min(PGSIZE, addr + len - va);
8          for (i = 0; i < n; i += n1) {
9              n1 = min(maxsz, n - i);
10             begin_op();
11             ilock(vma->f->ip);
12             if (writei(vma->f->ip, 1, va + i, va - vma->addr + vma-
>offset + i, n1) != n1) {
13                 iunlock(vma->f->ip);
14                 end_op();
15                 return -1;
16             }
17             iunlock(vma->f->ip);
18             end_op();
19         }
20     }
21 }

```

## 修改 `exit` 和 `fork`

### 1. 处理进程退出:

- 修改 `kernel/proc.c` 中的 `exit()` 函数，在进程退出时对所有文件映射内存进行取消映射处理。这类似于 `munmap()` 的操作，但需要遍历所有 VMA 来释放所有映射区域。

### 2. 处理进程复制:

- 在 `fork()` 函数中，确保子进程继承父进程的所有内存映射区域。复制 VMA 结构时，使用 `filedup()` 增加文件的引用计数，确保文件描述符在父子进程间共享。

- 在子进程的页错误处理中，如果是因映射区域访问导致的页错误，分配新的物理页而不是共享父进程的页，从而支持写时复制（Copy-On-Write, COW）策略。

## 9. 实验结果：

- 执行 `mmaptest` 测试程序，验证 `mmap` 和 `munmap` 功能的正确性和效率。

```
sheerio@ubuntu: ~/xv6/lab10 mmap/xv6-labs-2021
part 2 starting
part 1 starting
init: starting sh
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded
$
```

- 执行 `make grade`。

```
sheerio@ubuntu: ~/xv6/lab10 mmap/xv6-labs-2021
(11.2s)
== Test  mmaptest: mmap f ==
mmaptest: mmap f: OK
== Test  mmaptest: mmap private ==
mmaptest: mmap private: OK
== Test  mmaptest: mmap read-only ==
mmaptest: mmap read-only: OK
== Test  mmaptest: mmap read/write ==
mmaptest: mmap read/write: OK
== Test  mmaptest: mmap dirty ==
mmaptest: mmap dirty: OK
== Test  mmaptest: not-mapped unmap ==
mmaptest: not-mapped unmap: OK
== Test  mmaptest: two files ==
mmaptest: two files: OK
== Test  mmaptest: fork_test ==
mmaptest: fork_test: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (550.9s)
== Test time ==
time: OK
Score: 140/140
(base) sheerio@ubuntu: ~/xv6/lab10 mmap/xv6-labs-2021$
```

## 实验中遇到的问题及解决方法

### 1. 问题：页表脏位标志的正确设置

- **描述：**在实现 `mmap` 功能时，需要正确管理页表的脏位（`PTE_D`），特别是在写操作触发的页错误中。初始实现中未能正确设置脏位，导致对映射内存的修改未被适当追踪，影响了 `munmap` 函数的正确执行。

- **解决方法**：在 `kernel/trap.c` 中的 `usertrap()` 函数处理页错误时，增加对写操作的检测。如果写入地址对应的页表项未设置为可写，首先验证该地址是否属于某个 `mmap` 的范围，若属于则设置脏位，并进行惰性分配。

## 2. 问题：munmap 的正确实现

- **描述**：在实现 `munmap` 时遇到的问题是如何正确处理映射内存的回收以及潜在的文件回写。尤其是在处理 `MAP_SHARED` 映射时，必须将更改同步回文件。
- **解决方法**：为 `munmap` 实现添加检测脏页的步骤，并对有更改的页进行回写操作。使用 `uvmgetdirty()` 检查每页是否被修改，并使用文件写入函数将数据写回映射的文件。

## 3. 问题：复制写时错误和父子进程的文件映射共享

- **描述**：在 `fork` 过程中，需要确保子进程得到父进程映射区域的精确副本。初次实现中，子进程的页表直接指向父进程的物理页，未能正确处理写时复制逻辑。
- **解决方法**：修改 `fork` 的实现，确保在处理页错误时，如果涉及写操作且页面属于共享映射，则为子进程分配新的物理页，而不是共享父进程的页。同时确保更新 VMA 结构以反映新的映射关系。

# 实验心得

在完成这次关于 `mmap` 和 `munmap` 的实验中，我深刻体会到了操作系统虚拟内存管理的复杂性和灵活性。通过实现内存映射文件功能，我不仅加深了对虚拟内存、页表、和内存映射之间交互作用的理解，还学习到了如何处理内存中的权限控制和错误管理。

1. **虚拟内存的动态管理**：通过手动控制内存映射区域（VMA），我学习到了如何动态管理进程的虚拟地址空间。这包括如何为新的内存映射寻找合适的地址、如何在不同映射之间处理地址冲突，以及如何在 `munmap` 调用后清理资源。
2. **懒加载和页错误处理**：实现懒加载机制，即在实际需要时才分配内存和加载数据，让我深入理解了页错误处理的重要性。这一机制不仅优化了内存使用，还提高了系统的响应速度和效率。
3. **文件映射与共享内存**：通过 `mmap` 的实现，我掌握了如何将文件直接映射到进程的地址空间。这个功能使得进程可以像访问普通内存一样直接访问文件内容，极大地简化了文件 I/O 操作。同时，实现 `MAP_SHARED` 和 `MAP_PRIVATE` 两种映射类型，加深了我对共享内存和写时复制技术的理解。
4. **系统调用的整合**：在实现过程中，我需要确保 `mmap` 和 `munmap` 与系统中的其他部分（如页错误处理、进程创建和退出）正确集成。这要求我对 xv6 操作系统的内部工作机制有更全面的把握，并且能够对系统调用进行适当的修改和扩展。