已按照OS课程设计要求完成了XV6 2021年版的所有实验，下面的实验报告将包含如下内容：

1.环境搭建

2.每个实验的实验目的

3.每个实验的实验步骤及部分代码

4.实验过程中所遇到的问题及其解决方法

5.完成每个实验的心得感受

我的源码在https://github.com/litianlololo/xv6-lab-2021-lth

1. xv6实验介绍

1.1 xv6简介

xv6是一个教学目的的操作系统，旨在教授操作系统的基本概念和实现原理。它基于早期的UNIX版本（Version 6 UNIX），并由麻省理工学院（MIT）开发。

xv6的设计目标是提供一个清晰、简单和易于理解的操作系统实现，以帮助学生更好地理解操作系统的核心原理和概念。它被广泛应用于操作系统课程和教育领域，成为学习操作系统的重要工具之一。通过研究和修改Xv6的源代码，学生可以更深入地了解操作系统的内部工作原理，并实践实际的系统编程技能。

1.2 实验项目

本次OS课设项目选择xv6及Labs课程项目，版本为2021年版本

根据MIT的课程资源，进行了以下十个实验

1. 实用工具实验：Xv6 and Unix utilities
2. 系统调用实验：Lab: system calls
3. 页表实验：Lab: page tables
4. 中断实验：Lab: traps
5. 写时复制实验：Lab: copy on-write
6. 多线程实验：Lab: multithreading
7. 网卡驱动实验：Lab: network driver
8. 锁实验：Lab: lock
9. 文件系统实验：Lab: file system
10. 内存映射实验：Lab: mmap

1.3 实验平台

在虚拟机(VMware Workstation pro)运行Ubuntu 22.04，使用qemu来模拟RISC-V进行实验

2. 实验环境配置

2.1 虚拟机安装Ubuntu 22.04

安装VMware Workstation

下载Ubuntu系统镜像

在VMware Workstation中新建虚拟机，按提示步骤进行安装即可

2.2 配置 RISC-V 相关工具链

在终端中运行如下命令，安装RISC-V相关工具链

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

2.3 VS Code SSH

由于虚拟机性能和存储空间限制，通过SSH，使用宿主机的VS Code对虚拟机进行操作

配置过程不再赘述

3 Lab1 Utilities 实用工具实验

3.1 启动xv6

获取lab的 xv6 源代码

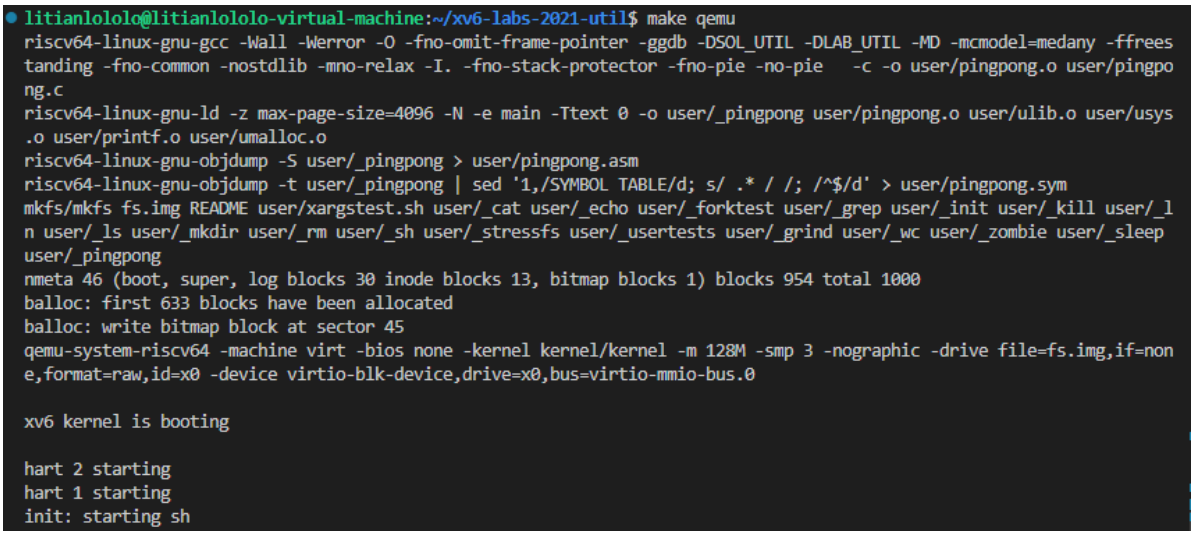
```
$ git clone git://g.csail.mit.edu/xv6-labs-2021
```

查看util分支

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

完成以上步骤，直接使用make qemu即可编译并在qemu中运行xv6

运行结果如下：



xv6 启动后，init 进程会启动一个 shell 等待用户的命令。若要结束运行xv6，需在键盘上同时按下 Ctrl+A 键，然后按下 X 键，即可结束运行。

3.2 sleep

1）实验目的

实现sleep工具，调用sleep system call实现，实现程序等待指定个时钟周期

2）实验步骤

分析

根据教程的提示，操作在文件`user/sleep.c`上进行，

另外user/user.h 中定义了如下系统调用

```
int sleep(int);
```

可知sleep只接受一个整型参数

检查是否提供了了足够的参数，如果是，则通过 `atoi` 函数将字符串参数 `argv[1]` 转换为整数值，并将结果存储在变量 `ticks` 中。调用`sleep`函数，使程序暂停执行指定的时间。

实现

1. 在user文件夹中新建文件sleep.c，将实现上述功能的源码添加入该文件；
2. 打开Makefile，在UPROGS中加入`$U/_sleep\`

测试

保存后在终端里执行 make qemu，重新编译

在终端中输入sleep 1,即睡眠1s

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util sleep ，即可进行自动评测

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-util$ ./grade-lab-util sleep
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (1.0s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
```

3）实验中遇到的问题和解决方法

第一次编写xv6代码，对文件结构和代码风格不太熟悉

通过参考users文件夹中其他代码实现，对结构和风格有了初步的了解

4）实验心得

在xv6中，UNIX程序"sleep"用于在一定时间内暂停程序的执行。它接受一个整数参数作为输入，表示要暂停的时间长度（以秒为单位）。当执行"sleep"命令时，程序会在调用时将指定的时间长度转换为以 tick 为单位的计数。然后，它使用计数器来进行定时操作，通过在每个tick 时检查计数器的值，来确定是否达到了指定的暂停时间。这个工具帮助我理解了进程调度和时间管理的概念，以及如何使用系统调用来控制程序的执行时间。

3.3 pingpong

1）实验目的

要实现一个名为 pingpong 的实用工具，用以验证 xv6 的进程通信的一些机制。该程序创建一个子进程，并使用管道与子进程通信

2）实验步骤

分析

"pingpong"是一个在操作系统教学中常用的示例程序，用于演示进程间通信和同步的概念。它模拟了两个进程之间传递一个消息（通常是一个字节）的过程。

该程序创建一个子进程，并使用管道与子进程通信："ping"进程首先发送一个消息（通常是一个字节）给"pong"进程，然后"pong"进程接收到消息后，再发送一个回复消息给"ping"进程。这个过程循环进行，实现了消息的来回传递。

实现

- 首先，定义两个管道`p1`和`p2`，用于进程间的通信。
- 然后，通过`pipe(p1)`和`pipe(p2)`创建了两个管道。
- 使用`fork()`创建了一个子进程。子进程中的`fork()`返回值为0，用于判断当前进程是否为子进程。

- 在子进程中，关闭了管道`p2`的写端和管道`p1`的读端。
- 子进程通过`read(p2[0], buf, 1)`从父进程读取一个字节的消息。
- 如果成功读取到一个字节的消息，子进程打印了`"<pid>: received ping"`的信息，其中`<pid>`是子进程的进程ID。
- 然后，子进程通过`write(p1[1], buf, 1)`将消息写回给父进程。
- 在父进程中，关闭了管道`p1`的写端和管道`p2`的读端。
- 父进程通过`write(p2[1], buf, 1)`向子进程写入一个字节的消息。
- 父进程通过`read(p1[0], buf, 1)`从子进程读取一个字节的消息。
- 如果成功读取到一个字节的消息，父进程打印了`"<pid>: received pong"`的信息，其中`<pid>`是父进程的进程ID。
- 最后，通过`exit(0)`正常退出程序。

在user文件夹中新建文件pingpong.c，将实现上述功能的源码添加入该文件；打开Makefile，在UPROGS中加入`$U/_pingpong\`

测试

保存后在终端里执行 make qemu，重新编译

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util pingpong ，即可进行自动评测

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-util$ ./grade-lab-util pingpong
make: “kernel/kernel”已是最新。
== Test pingpong == pingpong: OK (1.2s)
```

3）实验中遇到的问题和解决方法

实现 pingpong 需要使用管道在父子进程之间进行通信。使用管道通信需要正确的设置管道的读取端和写入端

在子进程中，关闭了管道`p2`的写端和管道`p1`的读端

在父进程中，关闭了管道`p1`的写端和管道`p2`的读端

p1用于子写父读

p2用于父写子读

4）实验心得

使用管道实现了一个简单的 ping-pong 程序，通过创建两个进程和两个管道，在父子进程之间传递字节并打印信息。这个工具帮助我学习了管道的使用，以及如何在进程间进行通信和同步。

3.4 primes

1）实验目的

利用管道理念实现质数筛

2）实验步骤

分析

使用管道实现并发素数筛选器。

并发素数筛选器是一种用于查找一定范围内所有素数的算法。它通过筛选法的方式，从2开始逐渐标记和排除合数，最终得到所有的素数。

实现

算法思路如下：

- 定义常量`MAX_NUM`，表示素数筛选的范围。
- 创建整型数组`p1`，用于管道的读取端和写入端。同时，定义了整型变量`fdr`和`fdw`，用于读取端和写入端的文件描述符。
- 定义整型变量`p`和`n`，分别用于存储当前筛选的素数和读取的数字。
- 定义整型变量`is_first`，用于判断是否为第一个进程。
- 在主函数`main()`中，首先判断是否为第一个进程。如果是，创建管道`p1`，并将读取端和写入端的文件描述符保存到`fdr`和`fdw`中。然后，从2到`MAX_NUM`之间的所有数依次写入管道。
- 接下来，如果是子进程，从管道中读取一个素数`p`。如果成功读取到一个素数，打印出来。然后，创建一个新的管道`p1`，并将写入端的文件描述符保存到`fdw`中。
- 在子进程中，通过循环从管道中读取数字`n`。如果读取成功且`n`不是`p`的倍数，就将`n`写入新的管道`p1`。
- 子进程将管道的读取端变为新管道的读取端`p1[0]`，然后关闭写入端的文件描述符。最后，通过递归调用`main()`函数，创建更多的子进程进行素数筛选。
- 如果是父进程，使用`wait(0)`等待子进程结束，然后关闭读取端的文件描述符。

详细代码实现详见源代码

在user文件夹中新建文件primes.c，将实现上述功能的源码添加入该文件；打开Makefile，在UPROGS中加入`$U/_primes\`

测试

保存后在终端里执行 make qemu，重新编译

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util primes，即可进行自动评测

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-util$ ./grade-lab-util primes
make: “kernel/kernel”已是最新。
== Test primes == primes: OK (1.4s)
```

3）实验中遇到的问题和解决方法

出现进程卡死

因为管道的写端一直没有释放导致的，在父进程wait(0)之前，需要将管道的写端释放

4）实验心得

多进程通信时，要时刻注意每个进程持有和需要的资源，及时释放，防止死锁

3.5 find

实现目的

实现UNIX find指令，用于在目录树中查找具有特定名称的所有文件。

实验步骤

实现

在函数 `fmt_name(char *path)` 中将路径格式化为文件名，将文件名提取出来并复制到缓冲区 `buf` 中。最后返回缓冲区 `buf`。

在函数 `find(char *path, char *findName)` 中进行查找，据文件的类型进行不同的处理：

- 如果是普通文件（`T_FILE`），则比较文件名 `fmtname(path)` 与目标文件名 `filename`，如果相同则打印该文件路径。
- 如果是目录类型 `T_DIR`
    - 遍历目录下的所有目录项，通过读取 `fd` 的方式逐个读取目录项 `de`。
    - 如果读取的目录项 `de.inum` 为0，则表示该目录项无效，继续下一个目录项的读取。
    - 将目录项的名称 `de.name` 复制到缓冲区 `buf` 中，并在名称后添加 `'/'` 作为路径分隔符。
    - 使用 `stat()` 函数获取路径 `buf` 对应文件的状态信息 `st`。
    - 判断路径的最后一个名称是否为 `.` 或 `..`，如果不是，则递归调用 `find()` 函数，传入新的目录路径 `buf` 和目标文件名 `filename`，继续查找具有特定名称的文件。

在主函数 `main()` 中，首先检查命令行参数数量是否为3个，如果不是，则打印使用说明并退出程序。

调用 `find()` 函数，传入指定的目录路径 `argv[1]` 和目标文件名 `argv[2]`，开始查找具有特定名称的文件。

详细代码实现详见源代码

在user文件夹中新建文件find.c，将实现上述功能的源码添加入该文件；打开Makefile，在UPROGS中加入 `$U/_find\`

测试

保存后在终端里执行 make qemu，重新编译

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util find，即可进行自动评测

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-util$ ./grade-lab-util find
make: “kernel/kernel”已是最新。
== Test find, in current directory == find, in current directory: OK (1.5s)
== Test find, recursive == find, recursive: OK (1.4s)
```

3.6 xargs

xargs程序，用于从标准输入读取行，并为每一行运行一个命令，并将该行作为命令的参数。

实验实现

算法思路如下：

在主函数 `main()` 中，首先检查命令行参数数量是否大于等于2，如果小于2，则打印使用说明并退出程序。如果合法，将命令行参数复制到 `args` 数组中，并将 `preargnum` 设置为命令行参数数量减1。进入循环，调用readline()函数读取输入的一行内容。如果读取成功，创建一个子进程。在循环的子进程中进行操作，执行传入的命令和参数。

readline()函数用于读取标准输入的内容，并将其按照空格或换行符进行分割，存储到args数组中。该函数会返回0表示输入结束，返回1表示读取到了一行数据。

详细代码实现详见源代码users/xargs.c

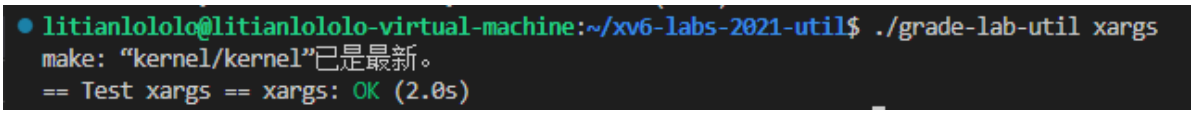
在user文件夹中新建文件xargs.c，将实现上述功能的源码添加入该文件；打开Makefile，在UPROGS中加入`$U/_xargs\`

实验测试

保存后在终端里执行 make qemu，重新编译

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util xargs，即可进行自动评测

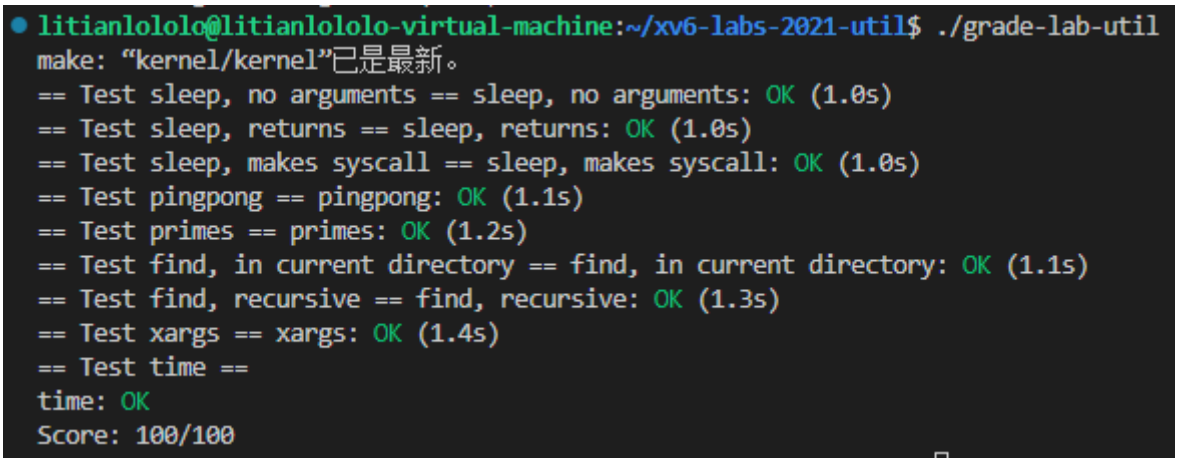
结果截图



3.7 完整测试

创建一个新文件`time.txt`，并在其中写入一个整数，表示在本实验所花费的小时数

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-util，即可进行自动评测



3.8 总结

通过第一个实验，熟悉了xv6系统的启动、测试操作，熟悉了 xv6 系统的结构和代码组织方式，对系统级编程和操作系统开发有了更深入的认识。

pingpong工具的实现帮助我学习了管道的使用，以及如何在进程间进行通信和同步。

Primes工具的实现帮助我深入理解了并发编程的概念和技术，以及如何使用进程间通信来协调并发任务。

通过实现这些工具，我加深了对操作系统原理、进程管理、文件系统和系统调用等概念的理解。

4 Lab2 system calls 系统调用实验

4.1 System call tracing

1）实验目的

添加一个跟踪系统调用的功能，创建一个新的trace系统调用来进行跟踪。用以追踪（打印）用户程序使用系统调用的情况。该系统调用接受一个参数，被称为 mask（掩码），用以设置被追踪的系统调用。

2）实验步骤

1.首先修改Makefile，将 $U/_trace添加到 UPROGS。

2.实现桩代码。在user/user.h中加入系统调用在用户态的入口

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

在user/usys.pl中加入在用户态执行的汇编代码(stubs)

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

在kernel/syscall.h为系统调用分配一个系统调用的编号

```
#define SYS_trace  22
```

3.接着在进程控制块中（kernel/proc.h/struct proc)添加一个新成员：int tracemask。表示当前进程需要跟踪哪些系统调用。而trace()系统调用的唯一功能就是设置这个tracemask。

添加一个sys_trace()函数,实现设置进程的tracemask的功能：

```
uint64
sys_trace(void)
{
int mask;
argint(0, &mask);
myproc()->tracemask = mask;
return 0;
}
```

可以看出它的功能就是获得传入trace系统调用的第一个参数，然后将其赋给当前进程的mask。

另外由于在fork时，子进程需要继承父进程的mask，因此在kernel/proc.c修改fork()函数，在其中合适位置添加：

```
np->mask=p->mask;
```

在kernel/syscall.c中添加sys_trace()的声明，以及在static uint64 (*syscalls[])(void)这个函数指针数组中添加sys_trace即可。

4.kernel/syscall.c中的syscall(void)函数是用户申请执行系统调用（执行ecall指令)后跳转到kernel space的统一入口。为了达到跟踪系统调用的效果，就需要扩展syscall(void)函数。

测试

保存后在终端里执行 make qemu，重新编译

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-syscall trace，即可进行自动评测

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-syscall$ ./grade-lab-syscall trace
make: “kernel/kernel”已是最新。
== Test trace 32 grep == trace 32 grep: OK (1.5s)
== Test trace all grep == trace all grep: OK (0.7s)
== Test trace nothing == trace nothing: OK (0.7s)
== Test trace children == trace children: OK (16.9s)
```

3）实验注意事项

注意需要修改fork函数，使得子进程可以继承trace的tracemask

4.2 实现 Sysinfo 系统调用

1）实验目的

添加一个系统调用sysinfo，用以收集当前时刻下xv6的一些信息

2）实验步骤

实现

1.首先仿照trace实验修改Makefile和实现桩代码。

```
$U/_sysinfotest
```

2.在kernel/sysproc.c中添加sysinfo的系统调用函数sys_sysinfo()。我们要在kernel space中声明并填充一个struct sysinfo结构体。然后获取用户以系统调用参数给出的user space中的struct sysinfo结构体的地址，并使用copyout函数把我们填充的struct sysinfo复制回user space。

2.1 在kernel space中声明并填充一个struct sysinfo结构体。填充freemem的函数应写在kernel/kalloc.c中：kalloc.c是xv6的物理内存分配器。所有空闲页串在一个freelist链表中。因此只需遍历链表即可。

填充nproc的函数写在kernel/proc.c中：proc.c的起始处声明了struct proc proc[NPROC]数组，来管理所有进程。只需遍历此数组，检查每个进程的状态即可。

2.2 将struct sysinfo复制回user space

我们需要声明一个uint64变量作为用户空间指针，并用argaddr()用sysinfo系统调用的参数将其赋值。接着使用copyout()函数把我们填充好的结构体复制回user space。

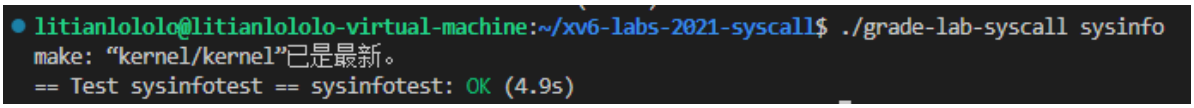
```
uint64
sys_sysinfo(void)
{
struct proc *p = myproc();
struct sysinfo info;
uint64 addr; // user pointer to struct stat
info.freemem = getfreemem();
info.nproc = getnproc();
if (argaddr(0, &addr) < 0)
return -1;
if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
```

```
return -1;
return 0;
}
```

测试

保存后在终端里执行 make qemu，重新编译

使用 xv6 实验自带的测评工具测评，在终端里输入 ./grade-lab-syscall sysinfo，即可进行自动评测



4.3 总结

实验中遇到的问题和解决

在tracing实验中，写完sys_trace函数，并修改完fork()后，make qemu进行测试，报错**未知的系统调用**

解决方案是在kernel/syscall.c中添加sys_trace()的声明，以及在**static uint64 (*syscalls[])(void)**这个**函数指针数组**中添加sys_trace

访问内核态中的数据结构时，最好进行获取锁和释放锁的操作，避免数据变“脏”：

```
acquire(&kmem.lock);
release(&kmem.lock);
```

实验心得

要在系统中实现一个新的系统调用，不仅仅是在内核中实现系统调用的实现代码，还需要将系统调用的原型添加到user/user.h，将桩代码添加到user /usys.pl和kernel/syscall.h的系统调用号。Makefile 调用 perl 脚本user/usys.pl，它产生user/usys.S，即实际的系统调用桩代码，它使用 RISC-V ecall指令转换到内核。

5 Lab3 page tables

5.1 Speed up system calls 加速系统调用

1）实验目的

通过在用户空间和内核之间共享只读区域中的数据，也就是映射一个页表，用来存储进程的pid，当获取pid时，就可以在该映射区域直接获取，而不用去切换到内核态，从而实现加速的功能

2）实验步骤

实现

1. 仿照trapframe，在进程控制块struct proc中添加struct usyscall *usyscall，作为共享页的物理地址。然后在kernel/proc.c中的allocproc()中完成共享页的分配和初始化

```
// Allocate a usyscall page.
  if((p->usyscall = (struct usyscall *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // Initialize the usyscall page.
  p->usyscall->pid=p->pid;
```

2. 接着在用户页表中添加共享页映射。在kernel/proc.c的proc_pagetable()中：

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

3. 最后处理共享页的释放。首先是释放共享页本身，在freeproc()中

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

4. 还要在proc_freepagetable()取消USYSCALL的映射

5.2 Print a page table打印页表

1）实验目的

可视化 RISC-V 页表，编写一个打印页表内容的函数。定义一个名为vmprint()的函数。它应该接受一个pagetable_t参数，并打印该页表。

2）实验步骤

实现

1. 要在kernel/vm.c中实现vmprint()函数，可以参考 freewalk 函数的实现。该函数通过递归的方式逐层释放页表。仿照这种递归方式，对该函数略做修改，我们可以构造遍历页表并按层数格式打印的函数 vmprintwalk(pagetable_t pagetable, int depth)
2. 然后实现 vmprint() 函数时，只需给定初始深度为 1 ，调用 vmprintwalk(pagetable_t pagetable, int depth) 即可

```
void vmprint(pagetable_t pagetable){
    printf("page table %p\n",pagetable);
    vmprintwalk(pagetable,1);
}
```

3. 最后按照要求在 kernel/exec.c 中的 exec() 函数的 return argc 前加入

```
if(p->pid==1) vmprint(p->pagetable);
return argc; //
```

详细代码见源代码

5.3 Detecting which pages have been accessed 检测哪些页面已被访问

1）实验目的

一些垃圾回收器（一种自动内存管理机制）需要知道关于**哪些页面已被访问（读取或写入）**的信息。向 xv6 添加一个新功能，该功能通过**检查 RISC-V 页表中的访问位**来检测并向user space报告此信息。RISC-V 硬件在 TLB 未命中时，在 PTE 中标记这些位。

需要实现pgaccess()，这是一个**报告哪些页面已被访问的系统调用**。系统调用需要三个参数。

- 第一个参数是需要检查的user page中，第一个page的起始虚拟地址。
- 第二个参数是需要检查的user page的数目。
- 第三个参数是用于存储结果的缓冲区的用户空间地址。结果会存储到一个位掩码中。这个位掩码从最低位对应检查的user page中的第一个page开始，用一个bit对应一个page。

2）实验步骤

实现

按照前文提及的方法添加系统调用以及其在 kernel/sysproc.c 中的实现 sys_pgaccess() ，并使用 argint() 和 argaddr() 获取传入的参数

```c
int sys_pgaccess(void){
    uint64 srcva, st;
    int len;
    uint64 buf = 0;
    struct proc *p = myproc();
    acquire(&p->lock);
    argaddr(0, &srcva);
    argint(1, &len);
    argaddr(2, &st);
    ......
    return 0;
}
```

接下来是实现系统调用功能的函数pgaccess()。我们可以规定可检查的页数上限。

接着for循环遍历从虚拟地址start_va开始的所有的页，用walk函数在页表中找到其PTE，检测PTE中的PTE_A位。若其为1，说明最近被访问过，将bitmask对应的位置1，并将PTE_A位重新置0。

for循环结束后，用copyout把bitmask写入系统调用第三个参数指定的目的虚拟地址

详细代码见源代码

5.4 总结

测试

编译并启动 xv6 后，在 shell 中运行 pgtbltest

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

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-pgtbl$ ./grade-lab-pgtbl pgtbltest
make: “kernel/kernel”已是最新。
== Test pgtbltest == (1.7s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
```

实验中遇到的问题和解决

在Detecting which pages have been accessed 检测哪些页面已被访问实验中，实现系统调用功能的函数pgaccess()的时候，检测到页被访问（PTE_A==1）后，需要将PTE_A重新置0，否则无法确定自上次调用pgaccess()以来是否访问过该页面

实验心得

对于实验中需要实现的一些函数，并不一定需要自己凭空写出来，也可以借鉴已经存在的用于实现其他功能的函数。

如在Print a page table打印页表实验中，要在kernel/vm.c中实现vmprint()函数，可以参考 freewalk 函数的实现。该函数通过递归的方式逐层释放页表。仿照这种递归方式，对该函数略做修改，我们可以构造遍历页表并按层数格式打印的函数 vmprintwalk(pagetable_t pagetable, int depth)。

6 Lab4 traps

6.1 assembly

1）实验目的

由于本次实验及之后的实验中，会涉及到机器指令级别的操作和调试，故而了解一些关于 RISC-V 汇编的知识将有助于实验的进行。第一题并不涉及程序，只是对MIT6.004知识的复习,主要内容是观察一些汇编代码和它们的行为。

2）实验步骤

我们先执行 make fs.img，xv6 的 Makefile 会自动生成反汇编后的代码。编译完成后，打开新生成的 user/call.asm，回答实验手册中的问题

Q1：

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

寄存器a0到a7用于存放函数调用的参数

通过查看汇编代码，可知参数13存放在寄存器a2中

```
void main(void) {
  1c: 1141                addi  sp,sp,-16
  1e: e406                sd ra,8(sp)
  20: e022                sd s0,0(sp)
  22: 0800                addi  s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                li a2,13
  26: 45b1                li a1,12
  28: 00000517            auipc a0,0x0
  2c: 7a050513            addi  a0,a0,1952 # 7c8 <malloc+0xe8>
  30: 00000097            auipc ra,0x0
  34: 5f8080e7            jalr  1528(ra) # 628 <printf>
  exit(0);
  38: 4501                li a0,0
  3a: 00000097            auipc ra,0x0
  3e: 274080e7            jalr  628(ra) # 2ae <exit>
```

Q2：

> Where is the call to function f in the assembly code for main? Where is the call to g?(Hint: the compiler may inline functions.)

同样观察call.c，发现f被printf("%d %d\n", f(8)+1, 13);调用，故查看对应的汇编代码

```
void main(void) {
  1c: 1141                addi  sp,sp,-16
  1e: e406                sd ra,8(sp)
  20: e022                sd s0,0(sp)
  22: 0800                addi  s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                li a2,13
  26: 45b1                li a1,12
  28: 00000517            auipc a0,0x0
  2c: 7a050513            addi  a0,a0,1952 # 7c8 <malloc+0xe8>
  30: 00000097            auipc ra,0x0
  34: 5f8080e7            jalr  1528(ra) # 628 <printf>
  exit(0);
  38: 4501                li a0,0
  3a: 00000097            auipc ra,0x0
  3e: 274080e7            jalr  628(ra) # 2ae <exit>
```

在跳转至 printf 执行前，一条指令 li a1,12 直接将立即数 12 存放在寄存器 a1 中，而该立即数恰好是调用 f(8)+1 结果，可见编译器优化直接通过常量优化的方式将常量值计算出来填入了 printf 的参数中，而没有真正执行 f 。

对于g的调用

观察call.c，发现g被f调用，查看对应的汇编代码

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

在 f 中调用了 g ，编译器对于这种较短的函数，直接将函数内联至 f 中，以减少压栈和跳转的开销。实际上执行 g 的代码是 14: 250d addiw a0,a0,3 。

Q3:

> At what address is the function printf located?

在call.asm中查找printf

```
...
0000000000000628 <printf>:

void
printf(const char *fmt, ...)
{
    ...
}
```

函数地址是0x628

Q4:

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

ra是返回地址，所以在执行之后返回地址是下一条指令的地址为0x38

Q5:

> Run the following code.
>
> unsigned int i = 0x00646c72;
>
> printf("H%x Wo%s", 57616, &i);
>
> What is the output? Here's an ASCII table 2 that maps bytes to characters.
>
> The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

将上述代码加入到 user/call.c 中，编译运行，得到的输出为：HE110 World 。

57616转换成16进制是e110， 而646c72按小端法从低地址到高地址按入下表转换为rld.如果是大端序的话，将 i 改为 0x726c6400 即可，无需改变 57616。

Q6:

> In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?
>
> printf("x=%d y=%d", 3);

得到的输出为：x=3 y=1 。产生这样输出的原因在于恰好后一个 %d 访问的地址中的数据的十进制表示为 1

6.2 Backtrace

1）实验目的

在 kernel/printf.c 实现 backtrace() ，并且在 sys_sleep 调用中插入一个 backtrace() 函数，用以打印当时的栈。

2）实验步骤

首先，在 kernel/riscv.h 中加入以下内联汇编函数用于读取存储在fp中的栈帧指针

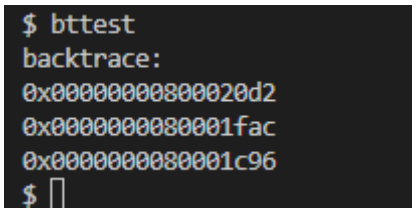
```
static inline uint64
r_fp()
{
    uint64 x;
    asm volatile("mv %0, s0" : "=r" (x) );
    return x;
}
```

然后利用上面的思想，在 kernel/printf.c 实现 backtrace()，在实现之前记得在defs.h中声明函数

backtrace()实现如下：

```
void
backtrace(void)
{
  printf("backtrace:\n");
  for (uint64 *fp = (uint64 *)r_fp();
       (uint64)fp < PGROUNDUP((uint64)fp);
         fp =(uint64 *)(*(fp-2)))
    printf("%p\n",*(fp-1));
}
```

然后即可编译运行xv6,执行bttest，得到输出如下：



6.3 Alarm

1）实验目的

为了让一个进程能够在消耗一定的 CPU 时间后被告知，我们需要实现 sigalarm(n, fn)系统调用。用户程序执行 sigalarm(n, fn) 系统调用后，将会在其每消耗 n 个 tick 的 CPU 时间后被中断并且运行其给定的函数 fn 。

- 实现一个间隔一段时间就报告进程使用CPU时间的alert
- 需要实现新的系统调用`sigalarm`
- 当应用调用sigalarm(n,fn)，那么在n个单位时间就需要调用一次函数fn
- sigalarm(0,0)表示停止间隔时间调用

2）实验步骤

添加`$U/_alarmtest`应用程序到`Makefile`中

先在 user/user.h 加入两个系统调用的入口，分别用于设置定时器和从定时器中断处理过程中返回：

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

在`user/usys.pl`，`kernel/syscall.h`, and `kernel/syscall.c`中添加系统调用原型（切不可遗漏，不然无法运行），具体代码略

第一步时，先按照实验指导说明中的暂时放置`sys_sigreturn`,只是return 0

在每个进程的控制块的数据结构中加入存储 sigalarm(n, fn) 中参数的项，即在 kernel/proc.h 的 struct proc中加入如下的项：

```
int alarminterval;              // sys_sigalarm() alarm interval in ticks
int alarmticks;                 // sys_sigalarm() alarm interval in ticks
void (*alarmhandler)();         // sys_sigalarm() pointer to the alarm handler
int sigreturned;
```

然后在 kernel/proc.c 的 allocproc(void) 中对这些变量进行初始化：

```
p->alarmticks = 0;
p->alarminterval = 0;
p->sigreturned = 1;
```

初始化后，在调用 sigalarm(n, fn) 系统调用时，执行的 kernel/sysproc.c 中的 sys_sigalarm() 需根据传入的参数设置 struct proc 的对应项：

```
uint64
sys_sigalarm(void)
{
    int ticks;
    uint64 handler;
    struct proc *p = myproc();
    if(argint(0, &ticks) < 0 || argaddr(1, &handler) < 0)
        return -1;
    p->alarminterval = ticks;
    p->alarmhandler = (void (*)())handler;
    p->alarmticks = 0;
    return 0;
}
```

接下来，需要为每个有sigalarm的进程，更新已经消耗的tick数，并判断该进程已经使用的 tick 数是否已经足以触发 alarm 。若 ticks 达到了预先设定的值，则将计数器清零，并使得用户进程跳至预设的处理过程 fn处执行。

首先在struct proc增设 alarmtrapframe 用于备份进程当前的上下文：

```
struct trapframe alarmtrapframe; // for saving registers
```

然后，对kernel/trap.c的usertrap()进行如下修改

```
if(which_dev == 2){
    p->alarmticks += 1;
    if ((p->alarmticks >= p->alarminterval) && (p->alarminterval > 0)){
        p->alarmticks = 0;
        if (p->sigreturned == 1){
            p->alarmtrapframe = *(p->trapframe);
            p->trapframe->epc = (uint64)p->alarmhandler;
            p->sigreturned = 0;
            usertrapret();
    }}
    yield();
}
```

当传入的 fn 执行完毕后，需要能够返回到进程正常的执行过程中。需要在内核态对应的 sys_sigreturn() 中将备份的上下文恢复，然后返回用户态。打开 kernel/sysproc.c，对sys_sigreturn做如下修改：

```
uint64
sys_sigreturn(void)
{
    struct proc *p = myproc();
    p->sigreturned = 1;
    *(p->trapframe) = p->alarmtrapframe;
    usertrapret();
    return 0;
}
```

编译并启动 xv6,z运行alarmtest，结果如下

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

6.4 总结

测试

终端输入 make grade

```
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (3.9s)
== Test running alarmtest ==
$ make qemu-gdb
(3.4s)
== 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 (249.1s)
== Test time ==
time: OK
Score: 85/85
```

实验中遇到的问题及解决方法

理解为什么需要保存trapframe的值，在什么时候保存trapframe的值。

这是因为再调用sys_sigalarm函数之前，已经把调用前所有的寄存器信息保存在了trapframe中，而sys_sigalarm的执行过程只是为alarmalarminterval等赋值。赋值后系统调用完成，trapframe的寄存器的值恢复，此时没有保存必要。

而在执行handler函数后，我们希望返回用户调用handler前的状态。但这个状态已经被用来调用handler了，而trapframe中存放的还是执行sys_sigalarm前的内容，此时无法满足状态恢复的要求。

需要注意的是，每个handler函数最后都会调用sigreturn函数。因此，我们需要在sigereturn中调用sigreturn时的trapframe，既可以在调用后恢复调用handler之前的状态。

实验心得

这个实验是我第一次接触到RISC-V汇编，在进行实验之前还需要预习一定的汇编和机器指令级别的操作和调试知识

通过这个实验极大的加强了我对epc、trap机制、栈帧的理解

相较于之前的实验，Alarm实验的难度更大，所运用的知识也更多，完整的走完alarm实验的流程，巩固并拓展了我学习xv6以来的知识

7 Lab5 Copy-on-Write Fork for xv6

7.1 Implement copy-on write

1）实验目的

写时复制（Copy-On-Write，COW）是一种内存管理技术，用于优化资源的使用，特别是在涉及进程复制和共享时。COW 通常应用于操作系统的进程管理和文件系统的实现中。

在传统的进程创建过程中，新创建的进程会复制父进程的内存内容，这意味着操作系统需要为新进程分配一块新的内存，并将父进程的内存内容复制到这块新的内存中。这会占用大量的时间和内存资源，尤其是当父进程的内存内容很大时。

COW 技术解决了这个问题，通过延迟实际的内存复制，从而节省了内存和时间。当一个新进程被创建时，操作系统不会立即为它分配新的内存，而是共享父进程的内存。只有在父进程或子进程试图修改共享的内存内容时，操作系统才会进行实际的内存复制，将需要修改的内存内容复制到新的内存页中。

本次实验需要实现写时复制的Fork系统调用。

2）实验步骤

因为COW中一个物理页面不仅仅被映射给一个页表，所以需要在完成COW实现之前，要对对应的数据结构进行修改。

首先，在 kalloc.c 中

```
struct spinlock reflock;
uint8 referencecount[PHYSTOP/PGSIZE];
```

还需要在初始化内存管理器的 kinit 中引入计数锁

```
void
kinit()
{
    initlock(&kmem.lock, "kmem");
    initlock(&reflock, "ref");
    freerange(end, (void*)PHYSTOP);
}
```

计数锁在释放物理页面时，将发挥作用。对freerange函数进行修改，在原先每次释放的位置将引用计数减一，仅在引用计数归零后释放页面。代码略。

接下来，要实现将父进程的页面映射给子进程，并将父进程和子进程的页表设为只读。

在proc.c 中找到fork()函数的实现，发现使用到了uvmcopy()函数，找到 vm.c中的实现。

理解uvmcopy()函数实现后，着手对该函数的修改，使用 mappages 只映射页面、增加引用计数并修改权限为只读，不分配页面

使用了分页机制中为软件使用保留的第 8 位作为标志位，若其为 1 ，则说明其为 COW 页。

```
int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz){
...
for(i = 0; i < sz; i += PGSIZE){
```

```
    ......
    *pte &= ~(PTE_W);
    *pte |= PTE_COW;
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){goto err;}
    acquire(&reflock);
    referencecount[PGROUNDUP((uint64)pa)/PGSIZE]++;
    release(&reflock);
    }
...
```

因为在前文中引入了引用计数的概念，在删除页面映射的过程中也需要做出对应的修改。具体实现在uvmunmap（）函数中。

```
if(do_free && referencecount[PGROUNDUP((PTE2PA(*pte)))/PGSIZE] < 1)
```

这是一个条件判断，检查是否需要释放物理页。如果 `do_free` 为真，并且对应物理页的引用计数已经小于1，那么意味着没有其他映射在使用该物理页，可以释放它。

若父子进程其中一个进程尝试写入内存，则会引发页面权限错误，从而产生一个中断最终调用usertrap()。因此需要修改usertrap（）；写页面权限造成的错误引发的中断会使得 SCAUSE 寄存器被设为 12 或 15。当cow页面中出现page fault，释放新页面，复制旧页面到新页面中，并且设新的PTE的PTE_W位为true。

```
else if (r_scause() == 12 || r_scause() == 15){
    //处理写页面权限造成的错误 代码略
}
```

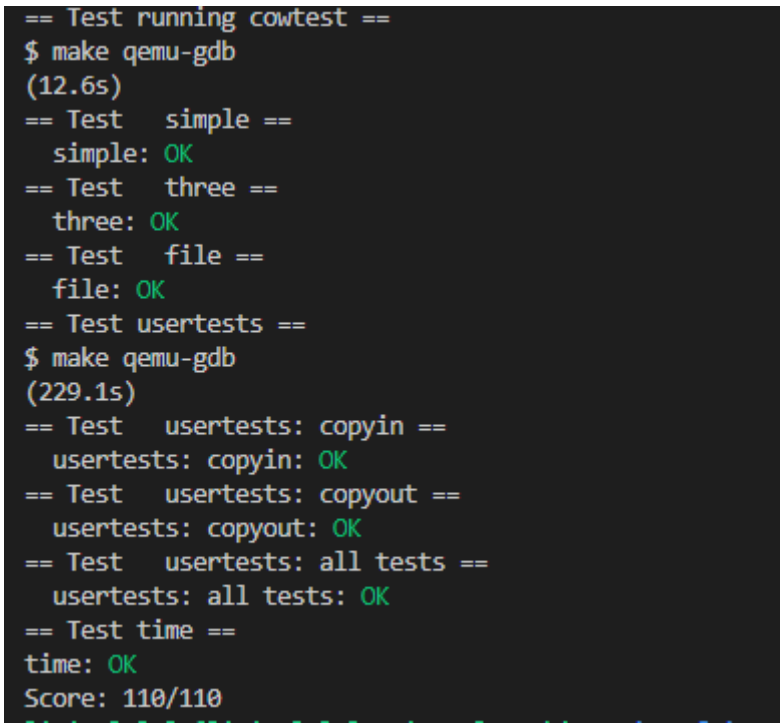
根据实验手册的提醒，完成上述工作后，还需要修改copyout()

前文提到使用标志位来判断页面是否为COW页面，因此在copyout（）中，可以根据#define PTE_COW标志位来决定执行什么操作

```
if (*pte & PTE_COW)
{
    //确定是COW页面
    //进行分配页面、复制数据并进行释放页面、修改页面权限和标志等操作
    //代码略
}
```

7.2 总结

测试



实验中遇到的问题及解决方法

1. 没能把握好释放物理页的时机

分情况讨论，时刻注意修改对物理页的引用计数，确保释放时机正确

2. 注意加锁

对于物理页的分配和引用计数时需要加锁。

实验心得

实现写时复制涉及到了比较多的知识点。COW 通常涉及进程的复制和内存共享。需要确保在进程复制时，子进程和父进程共享相同的物理内存页，直到其中一个进程尝试修改页内容为止。COW还设计页表的知识，通过实验，将内存映射、页表权限、页故障处理等知识进行了实战，加深理解。

8 Lab6 Multithreading

8.1 Uthread: switching between threads

1）实验目的

xv6 已经为该实验提供了基本的代码：user/uthread.c 和 user/uthread_switch.S 。为完成用户态线程库中的功能，需要在 user/uthread.c 中实现 thread_create() 和 thread_schedule() ，并且在 user/uthread_switch.S 中实现 thread_switch用于切换上下文。

2）实验步骤

首先观察user/uthread.c中的代码

在struct thread 中，我们还需要增加一个数据结构用于保存每个线程的上下文。参照内核中关于进程上下文的代码，做出一下修改：

```
struct context {
  uint64 ra;
  uint64 sp;
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct context context;
};
```

仿照 kernel/trampoline.S的结构，按照 struct context 各项在内存中的位置，在 user/uthread_switch.S 中加入如下的代码：

```
thread_switch:
    /* YOUR CODE HERE */
# addi ra, ra, 8; # t1 = t1 + imm

    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
```

```
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)

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

根据实验手册的提示，在`proc.c`可以找到类似的操作，于是我们仿照设置ra, sp和 s0（fp），对thread_create进行修改

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

  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->context.ra = (uint64)func;
  t->context.sp = (uint64)&t->stack[STACK_SIZE];
  t->context.s0 = (uint64)&t->stack[STACK_SIZE];
}
```

仿照`swtch.S`中的swtch的函数内容，在`thread_schedule`中添加：

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

编译运行xv6，输入uthread，部分结果如下：

```
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
```

8.2 Using threads线程的使用

1）实验目的

`notxv6/ph.c`中给出了一个hashtable的实现（不并发），需要通过pthread库实现并发

2）实验步骤

在多线程同时读写的情况下，部分数据会因为竞争访问，导致不能被正确写入到哈希表中。需要我们为这些共享数据结构加上互斥锁。

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-thread$ ./ph 1
100000 puts, 5.573 seconds, 17945 puts/second
0: 0 keys missing
100000 gets, 5.764 seconds, 17350 gets/second
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-thread$ ./ph 2
100000 puts, 3.122 seconds, 32027 puts/second
1: 16362 keys missing
0: 16362 keys missing
200000 gets, 7.244 seconds, 27609 gets/second
```

首先定义一个保护哈希表的互斥锁，并在开始线程前对其进行初始化：

```
pthread_mutex_t lock;
...
pthread_mutex_init(&lock, NULL);
```

接着需要将互斥锁用在写入的过程中，在写入操作哈希表时加上获取锁的操作，并在写入完成后释放锁

```
pthread_mutex_lock(&lock);
insert(key, value, &table[i], table[i]);
pthread_mutex_unlock(&lock);
```

再次使用make ph编译，运行./ph 2，结果如下;

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-thread$ make ph
make: “ph”已是最新。
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-thread$ ./ph 2
100000 puts, 2.893 seconds, 34563 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 5.370 seconds, 37244 gets/second
```

8.3 Barrier线程屏障

线程屏障（Thread Barrier）是一种多线程编程的同步机制，用于控制多个线程在某个点上等待，直到所有线程都达到了这个点才能继续执行。它通常用于在多线程程序中创建分阶段的同步，确保多个线程在某个阶段完成后再一起继续执行。

1）实验目的

需要实现barrier，使得所有已创建线程到达某点后才会一起开始执行

2）实验步骤

首先运行 make barrier 编译 notxv6/barrier.c ， 运行 ./barrier 2 ,输出如下，说明该 barrier() 并未正确实现

```
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-thread$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
litianlololo@litianlololo-virtual-machine:~/xv6-labs-2021-thread$ ./barrier 2
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
已放弃 (核心已转储)
```

按照实验手册和代码中的提示，在barrier（）函数中实现线程屏蔽的功能

```
static void
barrier()
{
  // YOUR CODE HERE
  pthread_mutex_lock(&bstate.barrier_mutex);
  ++ bstate.nthread;
  if(bstate.nthread < nthread) {
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  } else {
    bstate.nthread = 0;
    bstate.round ++;
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

再次运行 make barrier ， 运行 ./barrier 2 ,输出如下

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

8.4 总结

测试

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (5.1s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/litianlololo/xv6-labs-2021-thread”
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录“/home/litianlololo/xv6-labs-2021-thread”
ph_safe: OK (8.4s)
== Test ph_fast == make[1]: 进入目录“/home/litianlololo/xv6-labs-2021-thread”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/litianlololo/xv6-labs-2021-thread”
ph_fast: OK (21.7s)
== Test barrier == make[1]: 进入目录“/home/litianlololo/xv6-labs-2021-thread”
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录“/home/litianlololo/xv6-labs-2021-thread”
barrier: OK (11.1s)
== Test time ==
time: OK
Score: 60/60
```

实验中遇到的问题及解决方法

注意区分进程、线程和协程的区别：

xv6中一个进程只有一个线程，故进程和线程在本实验中区分不大。而本实验的用户线程切换时，不进入内核态，直接让用户线程主动让出CPU，比较像协程的概念。

实验心得

本次实验，主要涉及了线程的知识，考查了上下文切换、互斥和同步。

上下文切换是指操作系统在多任务环境下，将当前执行的进程或线程的状态（包括寄存器值、程序计数器等）保存起来，并加载另一个进程或线程的状态以切换执行。上下文切换允许多个任务在单个 CPU 上交替执行，实现并发。

互斥是一种同步机制，用于防止多个线程同时访问共享资源，从而避免数据竞争和不一致性。通过使用互斥锁或信号量等，只允许一个线程在特定时间内访问临界区，其他线程需要等待。

同步是协调多个线程或进程之间的执行顺序，以确保正确的交互和协作。同步机制包括互斥、条件变量、信号量等，用于保证线程之间按照特定的顺序或规则进行通信和交互。

9 Lab7 network driver

9.1 Your Job

1）实验目的

实现 `e1000_transmit` 和 `e1000_recv`,这两个函数可以实现传递和接收包

2）实验步骤

根据实验手册的提醒，我们主要需要实现kernel/e1000.c 中的两个函数：用于发送数据包的 e1000_transmit() 和用于接收数据包的 e1000_recv() 。

首先实现用于发送数据包的 e1000_transmit()。我们只需要将需要发送的数据包放入环形缓冲区中，设置好对应的参数并更新管理缓冲区的寄存器，即可视为完成了数据包的发送。

打开kernel/e1000.c,对e1000_transmit函数做出如下实现：

```
int
e1000_transmit(struct mbuf *m){
  // Your code here.
  acquire(&e1000_lock);
  uint32 idx = regs[E1000_TDT];
  if (tx_ring[idx].status != E1000_TXD_STAT_DD)
  {
    printf("e1000_transmit: tx queue full\n");
    __sync_synchronize();
    release(&e1000_lock);
    return -1;
  } else {
    if (tx_mbufs[idx] != 0)
      mbuffree(tx_mbufs[idx]);
    tx_ring[idx].addr = (uint64) m->head;
    tx_ring[idx].length = (uint16) m->len;
    tx_ring[idx].cso = 0;
    tx_ring[idx].css = 0;
    tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
    tx_mbufs[idx] = m;
    regs[E1000_TDT] = (regs[E1000_TDT] + 1) % TX_RING_SIZE;
  }
  __sync_synchronize();
  release(&e1000_lock);
  return 0;
}
```

在实现e1000_recv()时，根据 xv6 实验手册的提示，我们无需自己实现该拷贝过程，只需要调用 net.c 中的net_rx() 即可。而当发生中断时，中断处理过程e1000_intr会执行regs[E1000_ICR]表示已完成此次中断中所有数据包的处理，因此，需要在函数实现中将rx_ring的所有内容拷贝出去。同时，需要加入互斥锁，保证接受数据包的顺序正确。代码实现如下：

```
extern void net_rx(struct mbuf *);
static void
e1000_recv(void)
{
```

```
  // Your code here
  uint32 idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
  struct rx_desc* dest = &rx_ring[idx];
  while (rx_ring[idx].status & E1000_RXD_STAT_DD)
  {
    acquire(&e1000_lock);
    struct mbuf *buf = rx_mbufs[idx];
    mbufput(buf, dest->length);
    if (!(rx_mbufs[idx] = mbufalloc(0)))
      panic("mbuf alloc failed");
    dest->addr = (uint64)rx_mbufs[idx]->head;
    dest->status = 0;
    regs[E1000_RDT] = idx;
    __sync_synchronize();
    release(&e1000_lock);
    net_rx(buf);
    idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    dest = &rx_ring[idx];
  }
}
```

9.2 总结

测试

```
== Test running nettests ==
$ make qemu-gdb
(5.9s)
== Test   nettest: ping ==
  nettest: ping: OK
== Test   nettest: single process ==
  nettest: single process: OK
== Test   nettest: multi-process ==
  nettest: multi-process: OK
== Test   nettest: DNS ==
  nettest: DNS: OK
== Test time ==
time: OK
Score: 100/100
```

实验心得

实现网络驱动程序让我领略了实时性和并发编程的挑战。处理网络数据包需要高效地管理缓冲区、处理中断和保证数据的可靠传输。

此次实验虽然只需要完成两个函数的实现，但在试图解决问题的过程中，我也对网卡及其驱动程序有了新的认识。

10 Lab8 locks

10.1 Memory allocator

1）实验目的

在未修改前，所有内存块由一个锁管理，若有多个进程并发地获取内存，则会造成非常多的锁等待，kalloctest则统计了锁冲突的次数。修改内存块策略，以减少锁冲突，修改后的锁冲突应该大幅减少。

2）实验步骤

首先执行kalloctest，会产生三个进程不断地分配并释放内存。在这个过程中，锁会产生阻塞，因此会有额外的开销。结果如下：

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 5960650 #acquire() 433016
lock: bcache: #test-and-set 0 #acquire() 1254
--- top 5 contended locks:
lock: kmem: #test-and-set 5960650 #acquire() 433016
lock: proc: #test-and-set 1270610 #acquire() 330408
lock: proc: #test-and-set 1135031 #acquire() 330408
lock: proc: #test-and-set 1085724 #acquire() 330408
lock: proc: #test-and-set 1061916 #acquire() 330409
tot= 5960650
test1 FAIL
start test2
total free number of pages: 32499 (out of 32768)
```

需要通过为每个CPU分配一个空闲链表和对应的锁，当分配内存时会优先获取本空闲链表的锁，再进行分配内存；只有在必要情况才会到其它CPU的空闲链表借空闲页面。

首先将原来的单freelist改为多freelist，如下所示：

```
struct {

    struct spinlock lock;

    struct run *freelist;

} kmem[NCPU];
```

修改初始化操作，在kinit中将原来的初始化修改为for循环，对kmem的每个元素进行初始分配；

```
for (int i = 0; i < NCPU; i++)
    initlock(&kmem[i].lock, "kmem");
```

在获取cpuid时要关闭中断，防止线程切换，因此要对kfree进行修改。同时将释放的页面加入到对应CPU的空闲

页面链表(freelist)的头部。

```
push_off();
int id = cpuid();
pop_off();
acquire(&kmem[id].lock);
r->next = kmem[id].freelist;
kmem[id].freelist = r;
release(&kmem[id].lock);
```

修改kalloc，修改分配内存的逻辑。如果当前的CPU中有空闲的内存块，则直接分配；如果没有，则从其它CPU对应的freelist中借取一块：

```
if(r)
    kmem[id].freelist = r->next;
else {
    for (int i = 0; i < NCPU; i++) {
    if (i == id) continue;
        acquire(&kmem[i].lock);
        r = kmem[i].freelist;
        if(r)
            kmem[i].freelist = r->next;
        release(&kmem[i].lock);
        if(r) break;
    }
}
```

运行kcalloctest，结果如下：

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 184144
lock: kmem: #test-and-set 0 #acquire() 115875
lock: kmem: #test-and-set 0 #acquire() 133031
lock: bcache: #test-and-set 0 #acquire() 346
--- top 5 contended locks:
lock: proc: #test-and-set 4923845 #acquire() 942582
lock: proc: #test-and-set 4090254 #acquire() 942584
lock: proc: #test-and-set 3693092 #acquire() 942582
lock: proc: #test-and-set 3648955 #acquire() 942582
lock: proc: #test-and-set 3610787 #acquire() 942699
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

3）实验注意事项

在获取cpuid时要注意使用push_off和pop_off，关闭中断，防止线程切换。

10.2 Buffer cache

由于磁盘等外存设备普遍读写速度较慢，故很多操作系统都会在其文件系统的机制中设置外存块的缓存，以提高整个系统的性能。在原始的 xv6 中，对于缓存的读写是由单一的锁 bcache.lock 来保护的，这就导致了如果系统中有多个进程在进行 IO 操作，则等待获取 bcache.lock 的开销就会较大。

1）实验目的

xv6 的实验手册提示我们可以将缓存分为几个桶，为每个同单独设置一个锁，这样如果两个进程访问的缓存块在不同的桶中，则可以同时获得两个锁从而进行操作，而无需等待加锁。

本实验需要根据实验手册的提示修改cache管理策略，降低锁冲突。

2）实验步骤

首先根据实验指导书的提示，将锁修改为13份。在param.h中添加：

```
#define NBUCKET 13
```

在bio.c中修改结构体：

```
struct {
    struct spinlock lock[NBUCKET];
    struct buf buf[NBUF];
    struct buf head[NBUCKET];
} bcache;
```

修改bcache初始化函数binit()，使其与修改后的数据结构相适应：

```
void
binit(void)
{
    struct buf *b;
    for (int i = 0; i < NBUCKET; i++) {
        initlock(&(bcache.lock[i]), "bcache.hash");
        }
    for (int i = 0; i < NBUCKET; i++) {
        bcache.head[i].next = &bcache.head[i];
        bcache.head[i].prev = &bcache.head[i];
    }
    for(b = bcache.buf; b < bcache.buf+NBUF; b++){
        b->next = bcache.head[0].next;
        b->prev = &bcache.head[0];
        initsleeplock(&b->lock, "buffer");
        bcache.head[0].next->prev = b;
        bcache.head[0].next = b;
    }
}
```

修改brelse、bpin、bunpin函数，在必要的位置加锁；

```
int id = hash(b->blockno);
acquire(&bcache.lock[id]);
......
release(&bcache.lock[id]);
```

最后修改bget，要求在找到相应缓冲区后直接返回，若没有找到，则去其他桶中取出一个来放在自己的缓冲区

详细代码略

```
if(b->refcnt == 0) {
    b->dev = dev;
    b->blockno = blockno;
    b->valid = 0;
    b->refcnt = 1;
    // 将别的hash桶里缓冲区放入当前id的缓冲区中
    b->next->prev = b->prev;
    b->prev->next = b->next;
    release(&bcache.lock[j]);
    b->next = bcache.head[id].next;
    b->prev = &bcache.head[id];
    bcache.head[id].next->prev = b;
    bcache.head[id].next = b;
    release(&bcache.lock[id]);
    acquiresleep(&b->lock);
    return b;
}
```

执行bcachetest，结果如下：

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 32997
lock: kmem: #test-and-set 0 #acquire() 55
lock: kmem: #test-and-set 0 #acquire() 55
lock: bcache.hash: #test-and-set 0 #acquire() 6230
lock: bcache.hash: #test-and-set 0 #acquire() 4130
lock: bcache.hash: #test-and-set 0 #acquire() 4295
lock: bcache.hash: #test-and-set 0 #acquire() 2277
lock: bcache.hash: #test-and-set 0 #acquire() 4288
lock: bcache.hash: #test-and-set 0 #acquire() 2264
lock: bcache.hash: #test-and-set 0 #acquire() 4700
lock: bcache.hash: #test-and-set 0 #acquire() 4729
lock: bcache.hash: #test-and-set 0 #acquire() 8484
lock: bcache.hash: #test-and-set 0 #acquire() 6202
lock: bcache.hash: #test-and-set 0 #acquire() 6204
lock: bcache.hash: #test-and-set 0 #acquire() 6206
lock: bcache.hash: #test-and-set 0 #acquire() 6208
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 9317287 #acquire() 1271
lock: proc: #test-and-set 4797994 #acquire() 736699
lock: proc: #test-and-set 4615152 #acquire() 736701
lock: proc: #test-and-set 4613176 #acquire() 736700
lock: proc: #test-and-set 4590283 #acquire() 736121
tot= 0
test0: OK
start test1
test1 OK
```

10.3 总结

测试

```
== Test running kalloctest ==
$ make qemu-gdb
(155.4s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (19.7s)
== Test running bcachetest ==
$ make qemu-gdb
(25.4s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (255.9s)
== Test time ==
time: OK
Score: 70/70
```

实验心得

通过这次实验，我深入了解了锁的相关操作。在避免竞争、减少加锁开销时，可以采用资源重复和小粒度两种方式进行对锁的优化，从而避免竞争。

在Buffer cache实验中，发现使用固定大小的bucket，使用素数(如13)来减少哈希冲突；通过该实验经历产生死锁并解决死锁的过程，熟悉了对lock的处理。

11 Lab9 file system

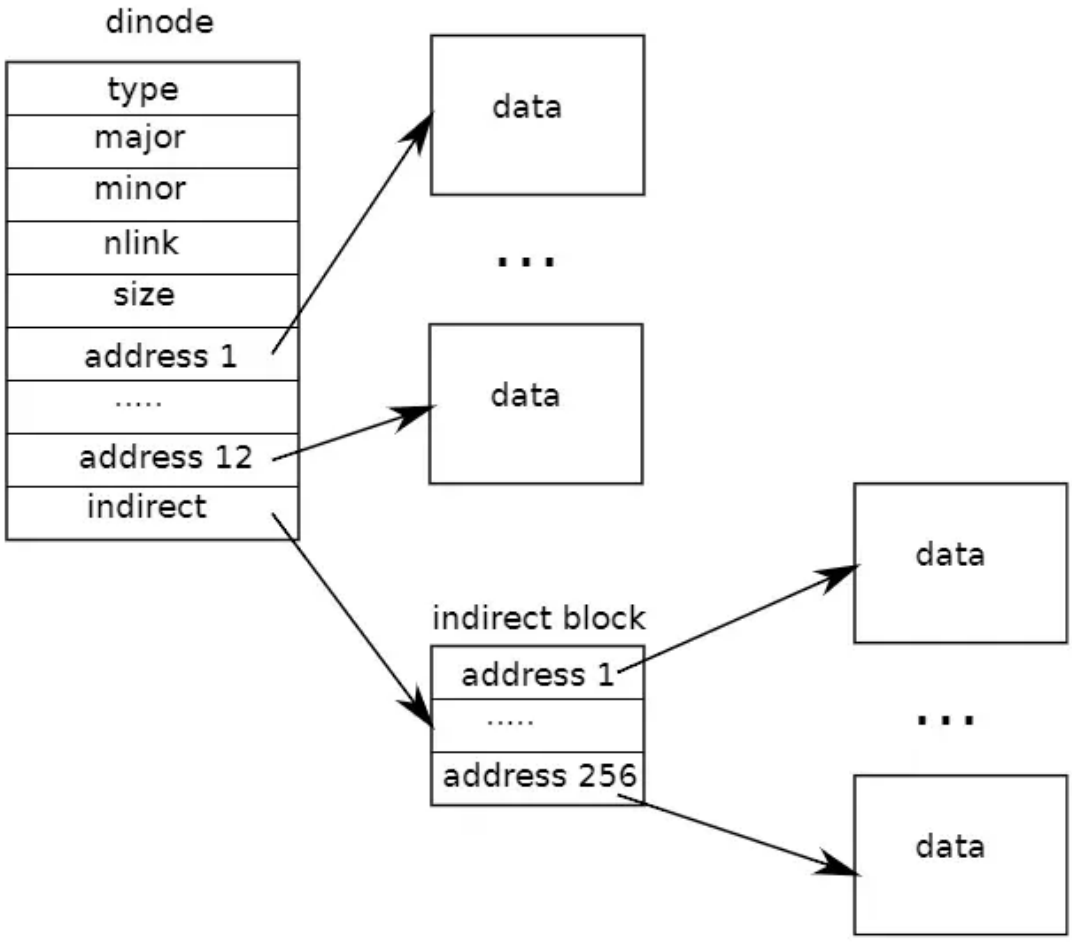
11.1 Large files

1）实验目的

在本实验中，测试程序将创建一个65803个磁盘块大小的文件，而未修改前的xv6中，inode为两级索引，有12个直接索引块和1个简介索引块(其指向256个数据块)。因此一个文件最多拥有268个数据块。需要修改文件系统结构和分配磁盘块代码以支持更大的文件。

2）实验步骤

使用三级索引(11个直接索引，1个间接索引块和1个二级间接索引块)的方式进行实验，总计支持的文件大小为11 + 256 + 256 × 256 =65803 块



因为要使用三级索引，根据三级索引的需求，对fs.h中的定义进行如下修改：

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT -1 + NINDIRECT + NINDIRECT * NINDIRECT)
```

修改 fs.h 中的struct dinode和 file.h 中的struct inode，修改如下：

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses
};
```

文件的寻址通过bmap()执行，设立三级索引后需要对寻址方式进行修改

寻址条件如下：

1. 判断逻辑块号，如果小于NDIRECT，则直接获取相应的物理块地址；
2. 如果小于NINDIRECT * NINDIRECT，则在一级目录中进行寻找，根据偏移确认对应的物理块地址；
3. 找到实际的物理块地址，在二级目录中查找，根据偏移确认对应的文件数据块的物理块地址。

详细代码略。

最后修改itrunc，处理释放问题。详细代码略。

运行编译xv6,执行bigfile，结果如下：

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

3）实验注意事项

1.最后记得处理释放问题

2.记得修改file.h中的struct inode

11.2 Symbolic links

为了方便文件管理，很多系统提供了符号链接的功能。符号链接（或软链接）是指通过路径名链接的文件；当一个符号链接被打开时，内核会跟随链接指向被引用的文件。Symbolic links可以指向另一个文件或目录，类似于一个路径的别名。通过使用符号链接，用户可以创建一个文件或目录的指向，而不是直接访问原始文件。

1）实验目的

通过实现一个系统调用symlink来创建符号链接。

2）实验步骤

根据实验指导手册的提示，在 stat.h 中添加

```
  #define T_SYMLINK 4;
```

在 fcntl.h 中添加

```
#define O_NOFOLLOW 0x010
```

实现系统调用函数sys_symlink，从寄存器中读取参数，并将目标路径和符号链接路径复制到对应的数组中，新建一个inode，将目标路径写入链接的inode节点即可。

```
uint64
sys_symlink(void)
{
    char target[MAXPATH],path[MAXPATH];
    struct inode *ip;
    //从用户态获取参数并将目标路径和符号链接路径复制到对应的数组中
    if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
        return -1;
    begin_op();
    if((ip = namei(path)) == 0){
        ip = create(path, T_SYMLINK, 0, 0);
        iunlock(ip);
    }
    ilock(ip);
    //将目标路径写入符号链接的inode节点
    if(writei(ip, 0, (uint64)target, ip->size, MAXPATH) != MAXPATH){
        return -1;
    }
    iunlockput(ip);
    end_op();
    return 0;
}
```

修改sys_open，在打开文件时，如果遇到符号链接，直接打开对应的文件即可。

运行编译xv6,执行symlinktest，结果如下：

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

3）实验注意事项

需要修改file.h中的inode，如果没修改的话，ip->addr最后一个地址有值，会报 panic:virtio_disk_intr status ；

11.3 总结

测试

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (352.0s)
    (Old xv6.out.bigfile failure log removed)
== Test running symlinktest ==
$ make qemu-gdb
(1.0s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
```

实验心得

通过这次实验，对先前操作系统理论学习中的三级索引进行了一次实战，加深了对多级索引的理解。

接触了符号链接的概念，并通过实验深入了解了符号链接的实现。

在进行最近的实验学习中，我掌握了关于磁盘存储文件的重要知识。在磁盘上，文件的存储主要涉及两种类型的数据块：inode（索引节点）和data（数据块）。在操作系统中，文件的相关信息被存储在inode中，每个文件都对应一个唯一的inode。inode内部包含了指向存储文件内容的磁盘块的索引信息，通过这些信息，系统能够精确定位文件在磁盘上的存储位置。

特别是，在这个过程中，我理解到inode块实际上是一个储存多个文件的inode结构的集合，它为每个文件提供了一种结构化的方式来存储文件的元数据。这使得系统可以高效地管理大量文件，而不需要为每个文件都单独维护一个inode。这种组织方式使得操作系统能够在不浪费空间的情况下存储大量文件的信息，从而更有效地管理磁盘空间。而在学习过程中，我也意识到inode的设计对于文件系统的性能和可扩展性非常重要，因为它直接影响了文件的查找、访问和管理过程。

12 Lab10 mmap

12.1

在 xv6 操作系统中，`mmap`（内存映射）是一种机制，允许用户程序将一个文件或设备映射到它的地址空间，从而可以通过内存访问文件的内容，就好像访问内存一样。这种映射可以是读取、写入或读写的，允许用户程序以类似于内存访问的方式来处理文件或设备。

在 xv6 中，`mmap` 提供了一种访问文件的灵活方法，它的主要步骤如下：

1. **打开文件：** 首先，用户程序需要打开要映射的文件，通过系统调用如 `open` 或 `create`。
2. **使用 `mmap`：** 一旦文件打开，用户程序可以使用 `mmap` 系统调用将文件映射到其地址空间。`mmap` 函数接受一些参数，如文件描述符、映射起始地址、映射长度、映射权限等。
3. **访问文件内容：** 一旦文件被映射到用户程序的地址空间，程序可以像访问内存一样来访问文件的内容。读取或写入映射区域会直接影响到文件内容。
4. **解除映射：** 在不再需要文件映射时，用户程序可以使用 `munmap` 系统调用来解除映射。这会使映射的内存区域不再与文件关联，并释放相应的资源

1）实验目的

在实验环境中实现 *mmap()* 系统调用

2）实验步骤

首先定义`vma`结构体用于保存内存映射信息

```
struct vma{
  uint64 start;
  uint64 end;
  uint64 length; // 0 means vma not used
  uint64 off;
  int permission;
  int flags;
  struct file *file;
  struct vma *next;

  struct spinlock lock;
};
```

在`proc`结构体中加入`struct vma *vma`指针：

```
struct proc {
  ...
  struct vma *vma;
  ...
};
```

接着实现对`vma`分配的代码：

```
struct vma vma_list[NVMA];
struct vma* vma_alloc(){
  for(int i = 0; i < NVMA; i++){
    acquire(&vma_list[i].lock);
    if(vma_list[i].length == 0){
      return &vma_list[i];
    }else{
      release(&vma_list[i].lock);
    }
  }
  panic("no enough vma");
}
```

在mmap函数中，需要主动申请一个vma，之后查找一块空闲内存，将vma插入到进程的vma的链表中

详细实现略

接下来在`usertrap`中对缺页中断进行处理：查找进程的`vma`链表，判断该地址是否为映射地址，如果不是就说明出错，直接返回；如果在`vma`链表中，就申请并映射一个页面，之后根据`vma`从对应的文件中读取数据：

详细代码略

```
  if(v == 0) return -1;                                    // not mmap addr
  if(scause == 13 && !(v->permission & PTE_R)) return -2; // unreadable vma
  if(scause == 15 && !(v->permission & PTE_W)) return -3; // unwritable vma

  // load page from file
  va = PGROUNDDOWN(va);
  char* mem = kalloc();
  if (mem == 0) return -4; // kalloc failed

  memset(mem, 0, PGSIZE);

  if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, v->permission) != 0){
    kfree(mem);
    return -5; // map page failed
  }
```

接下来实现munmap，限制链表中找出对应的vma结构体。分别处理头部、尾部、整个三种情况，写回并释放对应的页面并更新vma，如果整个区域都被释放就将vma和文件释放。

详细代码略

```
if(addr == v->start){
    writeback(v, addr, length);
    uvmunmap(p->pagetable, addr, length / PGSIZE, 1);
    if(length == v->length){
      // free all
      fileclose(v->file);
      if(pre == 0){
        p->vma = v->next; // head
      }else{
        pre->next = v->next;
        v->next = 0;
```

```
      }
      acquire(&v->lock);
      v->length = 0;
      release(&v->lock);
    }else{
      // free head
      v->start -= length;
      v->off += length;
      v->length -= length;
    }
  }else{
    // free tail
    v->length -= length;
    v->end -= length;
  }
```

写回函数先判断是否需要写回，当需要写回时就仿照filewrite的实现，将数据写回到对应的文件当中去；反之，不必写回。

详细代码略，如下为判断语句

```
if(!(v->permission & PTE_W) || (v->flags & MAP_PRIVATE)) // no need to writeback
    return;
```

最后在fork中复制vma到子进程

```
int
fork(void)
{
  ...
  np->state = RUNNABLE;

  np->vma = 0;
  struct vma *pv = p->vma;
  struct vma *pre = 0;
  while(pv){
    struct vma *vma = vma_alloc();
    vma->start = pv->start;
    vma->end = pv->end;
    vma->off = pv->off;
    vma->length = pv->length;
    vma->permission = pv->permission;
    vma->flags = pv->flags;
    vma->file = pv->file;
    filedup(vma->file);
    vma->next = 0;
    if(pre == 0){
      np->vma = vma;
    }else{
      pre->next = vma;
    }
    pre = vma;
    release(&vma->lock);
    pv = pv->next;
  }
```

```
  ...
}
```

在exit中将当前进程的vma链表释放，同时在exit时对页面进行写回处理

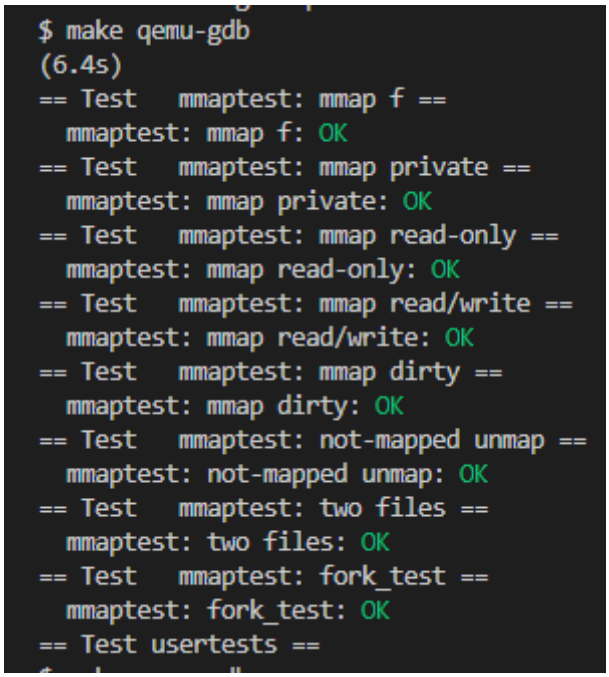
```
void
exit(int status)
{
  struct proc *p = myproc();

  if(p == initproc)
    panic("init exiting");

  // munmap all mmap vma
  struct vma* v = p->vma;
  struct vma* pv;
  while(v){
    writeback(v, v->start, v->length);
    uvmunmap(p->pagetable, v->start, PGROUNDUP(v->length) / PGSIZE, 1);
    fileclose(v->file);
    pv = v->next;
    acquire(&v->lock);
    v->next = 0;
    v->length = 0;
    release(&v->lock);
    v = pv;
  }
  ...
}
```

运行编译，执行mmaptest，结果如下

```
$ 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
```

12.2 总结

测试



实验心得

在这次实验中，了解了mmap是什么、怎么实现，对进程地址空间和文件的关系有了更深入的认识。

通过 `mmap`，用户程序可以避免显式的文件读取和写入，而是将文件的部分或全部内容直接映射到内存，从而提供了高效的数据访问方法。这对于处理大型文件、共享内存和实现一些特殊的文件操作非常有用。

总结

至此，已完成了全部的xv6 2021版实验内容，结合先前学习的操作系统理论课程学习，再进行实验的“动手”操作，让我意识到了此前理论学习的浅尝辄止，也庆幸有这个机会再次进行操作系统知识的学习。

通过这次实验，我养成了阅读实验指导手册以及上网搜索学习的习惯。遇到困难，先试着结合实验指导手册自己钻研，辅以查阅他人博客的先进经验。

在实验的过程中，我感受到了自己的进步以及学习到的知识，希望在之后的学习生涯中能够再进行一些操作系统的实验与实践，继续巩固并提高自己的操作系统方面的知识。