

实验报告

xv6-labs-2020

班级：软工四班

学号：2050992

姓名：法尔责娜

指导老师：王冬青

源码地址：<https://github.com/Angeldoo/xv6-labs-2020>

xv6 实验报告	5
1. 实验准备	5
1.1 实验环境	5
1.2 xv6 安装步骤	5
1.3 安装过程中遇到的问题	5
2. lab1:Xv6 and Unix utilities	6
2.1 实验目的	6
2.1.1 Boot xv6	6
2.1.2 sleep	6
2.1.3 pingpong	6
2.1.4 primes	6
2.1.5 find	6
2.1.6 xargs	7
2.2 实验步骤	7
2.2.1 Boot xv6	7
2.2.2 sleep	8
2.2.3 pingpong	9
2.2.4 primes	11
2.2.5 find	14
2.2.6 xargs	17
3. 实验中遇到的问题和解决办法	19
4. 实验心得	19
3.lab2:system calls	20
3.1 实验目的	20
3.1.1 System call tracing (moderate)	20
3.1.2 Sysinfo (moderate)	20
3.2 实验步骤	20
3.2.1 System call tracing (moderate)	20
3.2.2 Sysinfo (moderate)	23
3.3 实验中遇到的问题和解决方法	27
3.4 实验心得	28
4.lab3: page tables	28
4.1 实验目的	28
4.1.1 Print a page table (easy)	28
4.1.2 A kernel page table per process(hard)	29
4.1.3 Simplify copyin/copyinstr (hard)	29
4.2 实验步骤	29
4.2.1 Print a page table (easy)	29
4.2.2 A kernel page table per process(hard)	31

4.2.3 Simplify copyin/copyinstr (hard)	35
4.3 实验中遇到的问题及解决方法	40
4.4 实验心得	41
5.lab4 :Trap	41
5.1 实验目的	41
5.1.1 RISC-V assembly (easy)	41
5.1.2 Backtrace (moderate)	41
5.1.3 Alarm	41
5.2 实验步骤	42
5.2.1 RISC-V assembly (easy)	42
5.2.2 Backtrace (moderate)	44
5.2.3 Alarm	48
5.4 实验中遇到得问题及解决方式	53
5.4 实验心得	55
6.lab5:xv6 lazy page allocation	55
6.1 实验目的	55
6.1.1 Eliminate allocation from sbrk() (easy)	55
6.1.2 Lazy allocation (moderate)	56
6.1.3 Lazytests and Usertests (moderate)	56
6.2 实验步骤	56
6.2.1 Eliminate allocation from sbrk() (easy)	56
6.2.2 Lazy allocation (moderate)	57
6.2.3 Lazytests and Usertests (moderate)	58
6.3 实验中遇到的问题和解决方法	60
6.4 实验心得	62
7.lab6:Copy-on-Write Fork for xv6	63
7.1 实验目的	63
7.2 实验步骤	64
7.3 实验中遇到的问题及解决方法	68
7.4 实验心得	69
8.Lab7: Multithreading	69
8.1 实验目的	69
8.1.1 Uthread: switching between threads (moderate)	69
8.1.2 Using threads (moderate)	69

8.1.3 Barrier(moderate)	69
8.2 实验步骤	70
8.2.1 Uthread: switching between threads (moderate)	70
8.2.2 Using threads (moderate)	72
8.2.3 Barrier(moderate)	73
8.3 实验中遇到的问题及解决方法	74
8.4 实验心得	74
9.lab8:locks	75
9.1 实验目的	75
9.1.1 Memory allocator (moderate)	75
9.1.2 Buffer cache (hard)	75
9.2 实验步骤	75
9.2.1 Memory allocator (moderate)	76
9.2.2 Buffer cache (hard)	79
9.3 实验中遇到的问题及解决方法	86
9.4 实验心得	86
10.Lab9: file system	87
10.1 实验目的	87
10.1.1 Large files(moderate)	87
10.1.2 Symbolic links(moderate)	87
10.2 实验步骤	87
10.2.1 Large files(moderate)	87
10.2.2 Symbolic links (moderate)	89
10.3 遇到的问题及解决方法	92
11.Lab10: mmap	92
11.1 实验目的	92
11.2 实验步骤	92
11.3 实验中遇到的问题及解决方法	102
11.4 实验心得	103

xv6 实验报告

1. 实验准备

1.1 实验环境

windows 下的 VMware Workstation Pro
ubuntu 20.04
ssh:vscode
xv6-labs-2020

1.2 xv6 安装步骤

- 1.在 windows 环境下安装 VMware Workstation Pro
- 2.在 VMware Workstation 中建立自己虚拟机并将网上下载下来的 ubuntu 20.04 安装上去
- 3.打开虚拟机的终端，按照 lab 网页步骤的要求在终端输入指令下载
- 4.通过 vscode 的 ssh 连接到 ubuntu 的虚拟机上

1.3 安装过程中遇到的问题

1. 下载软件更新较慢

解决方法：更换/etc/apt/sources.list 文件里的源，Ubuntu 配置的默认源并不是国内的服务器，下载更新软件都比较慢。首先备份源列表文件 sources.list：

```
# 首先备份源列表
```

```
sudo cp /etc/apt/sources.list /etc/apt/sources.list_backup
```

在文件的最前面添加阿里云镜像（或中科大镜像），我的实验使用的是阿里云镜像
通过键入以下命令刷新，这样软件下载的速度一下子会很快

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

```
sudo apt-get install build-essential
```

2. apt-get 没有更新 `sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu`

解决方法：执行报错，遇到的这个错误都是 `apt-get` 没有即时更新导致的，一般在底下它都会提醒你用 `apt-get upgrade` 进行更新，更新之后就行了

3. make qemu 遇到的问题

```
root@host1f956f4ad8:~# riscv64-unknown-elf-gcc --version
-bash: riscv64-unknown-elf-gcc: command not found
```

这个意思是 `gcc` 还未安装完成，只有当环境安装好了才可以 `make qemu`

以上错误是遇到的安装过程中基本的错误，其他错误也基本解决，下面就可以通过 `git clone lab` 进行实验 1 了。

2. lab1:Xv6 and Unix utilities

2.1 实验目的

2.1.1 Boot xv6

切换到 `xv6-labs-2020` 代码的 `util` 分支，并利用 QEMU 模拟器启动 `xv6` 系统。

2.1.2 sleep

为 `xv6` 系统实现 UNIX 的 `sleep` 程序。你的 `sleep` 程序应该使当前进程暂停相应的时钟周期数，时钟周期数由用户指定。例如执行 `sleep 100`，则当前进程暂停，等待 100 个时钟周期后才继续执行。

2.1.3 pingpong

使用 UNIX 系统调用编写一个程序 `pingpong`，在一对管道上实现两个进程之间的通信。父进程应该通过第一个管道给子进程发送一个信息“ping”，子进程接收父进程的信息后打印 `<pid>: received ping`，其中是其进程 ID。然后子进程通过另一个管道发送一个信息“pong”给父进程，父进程接收子进程的信息然后打印 `<pid>: received pong`，然后退出。

2.1.4 primes

目标是使用 `pipe` 和 `fork` 来创建管道。第一个进程将数字 2 到 35 送入管道中。对于每个质数，要安排创建一个进程，从其左邻通过管道读取，并在另一条管道上写给右邻。由于 `xv6` 的文件描述符和进程数量有限，第一个进程可以停止在 35

2.1.5 find

编写一个简单的 UNIX `find` 程序，在目录树中查找包含特定名称的所有文件。解决方案应放在 `user/find.c`

2.1.6xargs

编写一个简单的 UNIX `xargs` 程序，从标准输入中读取行并为每一行运行一个命令，将该行作为命令的参数提供。解决方案应该放在 `user/xargs.c` 中。

2.2 实验步骤

2.2.1 Boot xv6

1.使用以下命令下载 `xv6-labs-2020` 到本地

```
git clone git://g.csail.mit.edu/xv6-labs-2020
```

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

```
git checkout util
```

2.make qemu 键入 `ls` 命令，就会看到 `xv6` 目录下的文件

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

```
.          1 1 1024
..         1 1 1024
README    2 2 2059
xargstest.sh 2 3 93
cat        2 4 24192
echo       2 5 23016
forktest   2 6 13240
grep       2 7 27496
init       2 8 23752
kill       2 9 22960
ln         2 10 22800
ls         2 11 26384
mkdir      2 12 23096
rm         2 13 23080
sh         2 14 41912
stressfs   2 15 23960
usertests  2 16 148376
grind      2 17 38064
wc         2 18 25280
zombie     2 19 22344
sleep      2 20 22920
pingpong   2 21 23888
primes     2 22 24768
find       2 23 26672
xargs      2 24 24704
console    3 25 0
```

3.ctrl+p

```
$
1 sleep init
2 sleep sh
```

3. 按 ctrl+a x 退

```
QEMU: Terminated
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$
```

2.2.2 sleep

1. 编写 sleep.c 的文件, 如下图, 对于输入的参数进行分析, 但是此时我所写的代码执行 make qemu 是无法执行代码命令的

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

int main(int argn, char *argv[]) {
    // 需要一个参数
    if (argn != 2) {
        fprintf(2, "must 1 argument for sleep\n");
        exit(1);
    }
    int sleepNum = atoi(argv[1]); // 获取需要睡眠的时间
    printf("(nothing happens for a little while)\n"); // 睡眠时间进行提示
    sleep(sleepNum);
    exit(0);
}
```

2. 修改 Makefile 文件中的 UPROGS 部分: 将 sleep 文件加进去

```
$U/_init\
$U/_kill\
$U/_ln\
$U/_ls\
$U/_mkdir\
$U/_rm\
$U/_sh\
$U/_stressfs\
$U/_usertests\
$U/_grind\
$U/_wc\
$U/_zombie\
$U/_sleep\
```


3. 执行测试：/grade-lab-util sleep，通过！

```
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.7s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
```

make qemu 也执行通过

```
$ sleep 10
(nothing happens for a little while)
$ sleep 20
(nothing happens for a little while)
```

2.2.3 pingpong

1. 编写所需要的 main 函数

- 首先引入头文件 `kernel/types.h` 和 `user/user.h`
- 定义两个文件描述符 `p1, p2` 的数组，创建两个管道，`pipe(p1)` 是写端的父进程，`pipe(p2)` 是写端的子进程，一个用于父进程传递信息给子进程，另一个用于子进程传递信息给父进程
- 创建缓冲区字符数组，存放传递的信息。
- 使用 `fork` 创建子进程，子进程的 `fork` 返回值为 0，父进程的 `fork` 返回子进程的进程 ID，所以通过 `if` 语句就能让父进程和子进程执行不同的代码块。
- 编写父进程执行的代码块。使用 `write()` 系统调用传入三个参数，把字符串 "ping" 写入管道一，第一个参数为管道的写入端文件描述符，第二个参数为写入的字符串，第三个参数为写入的字符串长度。然后调用 `wait()` 函数等待子进程完成操作后退出，传入参数 0 或者 `NULL`，不过后者需要引入头文件 `stddef.h`。使用 `read()` 系统调用传入三个参数，接收从管道二传来的信息，第一个参数为管道的读取端文件描述符，第二个参数为缓冲区字符数组，第三个参数为读取的字符串长度。最后调用 `printf()` 函数打印当前进程 ID 以及接收到的信息，即缓冲区的内容。
- 编写子进程执行的代码块。使用 `read()` 系统调用传入三个参数，接收从管道一传来的信息，第一个参数为管道的读取端文件描述符，第二个参数为缓冲区字符数组，第三个参数为读取的字符串长度。然后调用 `printf()` 函数打印当前进程 ID 以及接收到的信息，即缓冲区的内容。最后使用 `write()` 系统调用传入三个参数，把字符串 "pong" 写入管道二，第一个参数为管道的写入端文件描述符，第二个参数为写入的字符串，第三个参数为写入的字符串长度。
- 最后调用 `exit()` 系统调用使程序正常退出。

```
#include <kernel/types.h>
```

```
#include <user/user.h>
```

```
1. int main(){
```

```
2. //pipe1(p1): 写端父进程, 读端子进程
3. //pipe2(p2): 写端子进程, 读端父进程
4. int p1[2], p2[2];
5. //来回传输的字符数组: 一个字节
6. char buffer[] = {'X'};
7. //传输字符数组的长度
8. long length = sizeof(buffer);
9. //父进程写, 子进程读的 pipe
10. pipe(p1);
11. //子进程写, 父进程读的 pipe
12. pipe(p2);
13. //子进程
14. if(fork() == 0)
15. {
16.     //关掉不用的 p1[1]、p2[0]
17.     close(p1[1]);
18.     close(p2[0]);
19.     //子进程从 pipe1 的读端, 读取字符数组
20.     if(read(p1[0], buffer, length) != length){
21.         printf("a--->b read error!");
22.         exit(1);
23.     }
24.     //打印读取到的字符数组
25.     printf("%d: received ping\n", getpid());
26.     //子进程向 pipe2 的写端, 写入字符数组
27.     if(write(p2[1], buffer, length) != length){
28.         printf("a<---b write error!");
29.         exit(1);
30.     }
31.     exit(0);
32. }
33. //关掉不用的 p1[0]、p2[1]
34. close(p1[0]);
35. close(p2[1]);
36. //父进程向 pipe1 的写端, 写入字符数组
37. if(write(p1[1], buffer, length) != length){
38.     printf("a--->b write error!");
39.     exit(1);
40. }
41. //父进程从 pipe2 的读端, 读取字符数组
42. if(read(p2[0], buffer, length) != length){
```

```
43.     printf("a<---b read error!");
44.     exit(1);
45. }
46. //打印读取的字符数组
47. printf("%d: received pong\n", getpid());
48. //等待进程子退出
49. wait(0);
50. exit(0);
51. }
```

2.修改 Makefile 文件中的 UPROGS 部分

```
$U/_forktest\
$U/_grep\
$U/_init\
$U/_kill\
$U/_ln\
$U/_ls\
$U/_mkdir\
$U/_rm\
$U/_sh\
$U/_stressfs\
$U/_usertests\
$U/_grind\
$U/_wc\
$U/_zombie\
$U/_sleep\
$U/_pingpong\
```

4. 测试

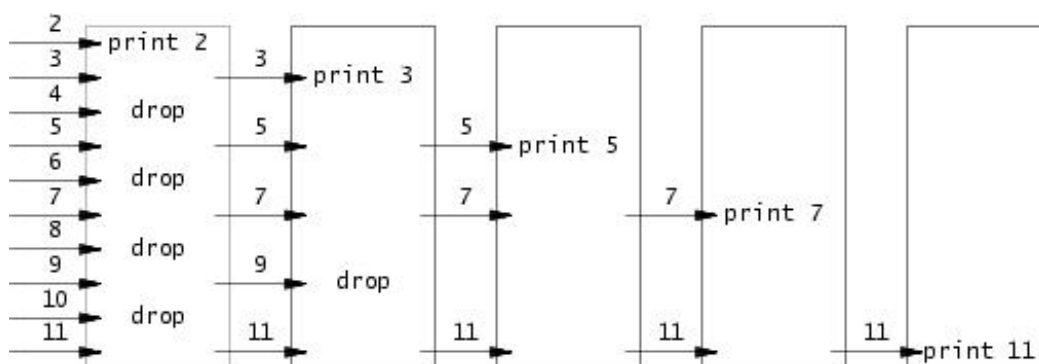
- xv6 中测试 pingpong

```
hart 1 starting
hart 2 starting
init: starting sh
$ sleep 20
(nothing happens for a little while)
$ pingpong
5: received ping
4: received pong
```

- ./grade-lab-util pingpong 单项测试:

```
farzana@farzana-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (3.3s)
```

2.2.4 primes



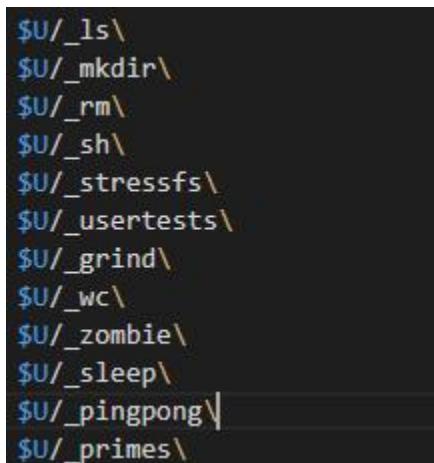
- 首先打开提示给出的链接，阅读并观察上面给出的图。由图可见，首先将数字全部输入到最左边的管道，然后第一个进程打印出输入管道的第一个数 2，并将管道中所有 2 的倍数的数剔除。接着把剔除后的所有数字输入到右边的管道，然后第二个进程打印出从第一个进程传入管道的第一个数 3，并将管道中所有 3 的倍数的数剔除。接着重复以上过程，最终打印出来的数都为素数。
- 首先，将创建管道，将 35 个数字全部放到这个管道当中，第二步创建一个 func() 的函数，在这个函数当中实现我想要完成的任务，通过递归的方式找到我们想要的质数，从小到大的方式进行编译，首先将 2 赋值给 prime，在数组当中找到是 2 的倍数的数并进行剔除，不是 2 的倍数从 3 开始对于这个管道进行赋值，下一次又从 3 开始，每一次都打印数组的第一个元素

```

1. #include "kernel/types.h"
2. #include "user/user.h"
3.
4. void func(int *input, int num){
5.     if(num == 1){
6.         printf("prime %d\n", *input);
7.         return;
8.     }
9.     int p[2], i;
10.    int prime = *input;
11.    int temp;
12.    printf("prime %d\n", prime);
13.    pipe(p);
14.    //每次给 temp 赋值，找出这个数组当中，是它的倍数的数，并且剔除
15.    if(fork() == 0){
16.        for(i = 0; i < num; i++){
17.            temp = *(input + i);
18.            write(p[1], (char *)&temp, 4);
19.        }
20.        exit(0);
21.    }
22.    close(p[1]);
23.    if(fork() == 0){
  
```

```
24.     int counter = 0;
25.     char buffer[4];
26.     while(read(p[0], buffer, 4) != 0){
27.         temp = *((int *)buffer);
28.         if(temp % prime != 0)// 如果不是 temp 的倍数, 就将这个数赋值到 input
           数组当中
29.         {
30.             *input = temp;
31.             input += 1;
32.             counter++;
33.         }
34.     }
35.     func(input - counter, counter);//递归调用
36.     exit(0);
37. }
38. wait(0);
39. wait(0);
40. }
41. int main(){
42.     int input[34];
43.     int i = 0;
44.     for(; i < 34; i++){
45.         input[i] = i+2;
46.     }
47.     func(input, 34);
48.     exit(0);
49. }
```

2. 在 Makefile 文件中, UPROGS 项追加一行 \$U/_primes\。编译并运行 xv6 进行测试。

A terminal window with a dark background and light-colored text. It displays a list of directories under the path \$U/. The directories are: _ls, _mkdir, _rm, _sh, _stressfs, _usertests, _grind, _wc, _zombie, _sleep, _pingpong, and _primes. The text is as follows:

```
$U/_ls\
$U/_mkdir\
$U/_rm\
$U/_sh\
$U/_stressfs\
$U/_usertests\
$U/_grind\
$U/_wc\
$U/_zombie\
$U/_sleep\
$U/_pingpong\
$U/_primes\
```

```
hart 2 starting
hart 1 starting
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
```

通过测试样例

```
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.7s)
```

2.2.5 find

- 在 `user/ls.c` 中，有一个名为 `fmtname()` 的函数，其目的是将路径格式化为文件名，也就是把名字变成前面没有左斜杠 `/`，仅仅保存文件名。
- 在 `user/ls.c` 中，有一个名为 `ls()` 的函数，首先函数里面声明了需要用到的变量，包括文件名缓冲区、文件描述符、文件相关的结构体等等。其次使用 `open()` 函数进入路径，判断此路径是文件还是文件名。
- 下面的参考实验代码只使用了 `find()` 函数，与 `ls()` 函数一样，首先声明了文件名缓冲区、文件描述符和文件相关的结构体。其次试探是否能进入给定路径，然后调用系统调用获得一个已存在文件的模式，并判断其类型。如果该路径不是目录类型就报错。接着把绝对路径进行拷贝，循环获取路径下的文件名，与要查找的文件名进行比较，如果类型为文件且名称与要查找的文件名相同则输出路径，如果是目录类型则递归调用 `find()` 函数继续查找。

```
1. #include "kernel/types.h"
2. #include "kernel/fcntl.h"
3. #include "kernel/stat.h"
4. #include "kernel/fs.h"
5. #include "user/user.h"
6.
7. /*
8.    将路径格式化为文件名
9. */
10. char* fmt_name(char *path){
11.     static char buf[DIRSIZ+1];
12.     char *p;
```

```
13. // Find first character after last slash.
14. for(p=path+strlen(path); p >= path && *p != '/'; p--);
15. p++;
16. memmove(buf, p, strlen(p)+1);
17. return buf;
18. }
19. /*
20.     系统文件名与要查找的文件名，若一致，打印系统文件完整路径
21. */
22. void eq_print(char *fileName, char *findName){
23.     if(strcmp(fmt_name(fileName), findName) == 0){
24.         printf("%s\n", fileName);
25.     }
26. }
27. /*
28.     在某路径中查找某文件
29. */
30. void find(char *path, char *findName){
31.     int fd;
32.     struct stat st;
33.     if((fd = open(path, O_RDONLY)) < 0){
34.         fprintf(2, "find: cannot open %s\n", path);
35.         return;
36.     }
37.     if(fstat(fd, &st) < 0){
38.         fprintf(2, "find: cannot stat %s\n", path);
39.         close(fd);
40.         return;
41.     }
42.     char buf[512], *p;
43.     struct dirent de;
44.     switch(st.type){
45.         case T_FILE: //是文件
46.             eq_print(path, findName); //路径和名字
47.             break;
48.         case T_DIR: //是目录
49.             if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
50.                 printf("find: path too long\n");
51.                 break;
52.             }
53.             strcpy(buf, path);
54.             p = buf+strlen(buf);
55.             *p++ = '/';
56.             while(read(fd, &de, sizeof(de)) == sizeof(de)){
```

```

57.         //printf("de.name:%s, de.inum:%d\n", de.name, de.inum);
58.         if(de.inum == 0 || de.inum == 1 || strcmp(de.name, ".")==0 |
| strcmp(de.name, "..")==0)
59.             continue;
60.             memmove(p, de.name, strlen(de.name));
61.             p[strlen(de.name)] = 0;
62.             find(buf, findName);
63.         }
64.         break;
65.     }
66.     close(fd);
67. }
68.
69. int main(int argc, char *argv[]){
70.     if(argc < 3){
71.         printf("find: find <path> <fileName>\n");
72.         exit(0);
73.     }
74.     find(argv[1], argv[2]);
75.     exit(0);
76. }

```

xv6 测试

```

$ echo > b
$ mkdir a
$ mkdir c
$ echo > c/a/b
open c/a/b failed
$ echo > a/b
$ echo > c/a/b
open c/a/b failed
$ find .b
find: find <path> <fileName>
$ find . b
./b
./a/b

```

```

$ echo > b
$ mkdir d
$ echo > d/b
$ find . d
$ find .d
find: find <path> <fileName>
$ find . b
./b
./a/b
./d/b

```

./grade-lab-util find 单项测试:


```
farzena@farzena-virtual-machine:~/work/xv6-labs-2026$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.8s)
== Test find, recursive == find, recursive: OK (1.3s)
```

2. 2. 6xargs

- 首先，我们定义一个字符数组，作为子进程的参数列表，其大小设置为 `kernel/param.h` 中定义的 `MAXARG`，用于存放子进程要执行的参数。而后，建立一个索引便于后面追加参数，并循环拷贝一份命令行参数，即拷贝 `xargs` 后面跟的参数。创建缓冲区，用于存放从管道读出的数据。
- 然后，循环读取管道中的数据，放入缓冲区，建立一个新的临时缓冲区存放追加的参数。把临时缓冲区追加到子进程参数列表后面。并循环获取缓冲区字符，当该字符不是换行符时，直接给临时缓冲区；否则创建一个子进程，把执行的命令和参数列表传入 `exec()` 函数中，执行命令。当然，这里一定要注意，父进程一定得等待子进程执行完毕。

```
1. #include "kernel/types.h"
2. #include "kernel/stat.h"
3. #include "user/user.h"
4. #include "kernel/param.h"
5.
6. #define MAXN 1024
7.
8. int
9. main(int argc, char *argv[])
10. {
11.     // 如果参数个数小于 2
12.     if (argc < 2) {
13.         // 打印参数错误提示
14.         fprintf(2, "usage: xargs command\n");
15.         // 异常退出
16.         exit(1);
17.     }
18.     // 存放子进程 exec 的参数
19.     char * argvs[MAXARG];
20.     // 索引
21.     int index = 0;
22.     // 略去 xargs，用来保存命令行参数
23.     for (int i = 1; i < argc; ++i) {
24.         argvs[index++] = argv[i];
25.     }
26.     // 缓冲区存放从管道读出的数据
27.     char buf[MAXN] = {"\0"};
28.
```

```
29.  int n;
30.  // 0 代表的是管道的 0, 也就是从管道循环读取数据
31.  while((n = read(0, buf, MAXN)) > 0 ) {
32.      // 临时缓冲区存放追加的参数
33.      char temp[MAXN] = {"\0"};
34.      // xargs 命令的参数后面再追加参数
35.      argvs[index] = temp;
36.      // 内循环获取追加的参数并创建子进程执行命令
37.      for(int i = 0; i < strlen(buf); ++i) {
38.          // 读取单个输入行, 当遇到换行符时, 创建子线程
39.          if(buf[i] == '\n') {
40.              // 创建子线程执行命令
41.              if (fork() == 0) {
42.                  exec(argv[1], argvs);
43.              }
44.              // 等待子线程执行完毕
45.              wait(0);
46.          } else {
47.              // 否则, 读取管道的输出作为输入
48.              temp[i] = buf[i];
49.          }
50.      }
51.  }
52.  // 正常退出
53.  exit(0);
54. }
```

测试结果

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ echo hello too | xargs echo bye
bye hello too
```

```
$ make qemu-gdb
sleep, no arguments: OK (5.6s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (1.0s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.0s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.0s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.0s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.2s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.2s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.7s)
== Test time ==
time: OK
Score: 100/100
```

3. 实验中遇到的问题和解决办法

- 写完函数文件发现编译不了
解决办法：编译不了的原因是没有在 `makefile` 里加入文件的名称
- 文件编译不成功，部分的功能无法实现
解决办法：有可能是头文件没有加进去，也有可能是函数文件有语法错误
- 在写后面两个文件的代码的时候出现瓶颈
解决办法：认真读实验说明，比如 `find.c` 文件要看 `ls.c` 中的函数方法，基本上就是将代码以另一种方式去实现，所以多去读网站上的实验说明。

4. 实验心得

第一次做操作系统的实验，感觉还是有很长的路要走，有一部分的时间是在网站上面先看的 MIT 哈佛的视频，有一部分的时间是在网站上面去读懂这个实验的目的和实验给出的方向，最后是进行代码的编写，学会了如何使用网络资源去正确的学习。

3. lab2:system calls

3.1 实验目的

3.1.1 System call tracing (moderate)

在本实验中，添加一个名为 `trace` 的系统调用。在以后调试代码时，该系统调用可能对你有帮助。`trace` 接受一个整型参数 `mask`，指定要跟踪的系统调用。例如，为了跟踪 `fork` 系统调用，程序调用 `trace(1 << SYS_fork)`。其中，`SYS_fork` 是 `kernel/syscall.h` 中 `fork` 的系统调用号。如果向 `mask` 传递了一个系统调用的编号，则必须修改 `xv6` 的内核，使得每个系统调用即将返回时打印出一行内容。该行内容应包含进程 `id`、系统调用的名称和返回值，你不需要打印系统调用参数。`trace` 系统调用应该对调用它的进程以及由它派生的任何子进程开启跟踪，但不应影响其他进程。

3.1.2 Sysinfo (moderate)

添加一个 `sysinfo` 系统调用，用来收集当前系统的相关信息。该系统调用接收 1 个参数：指向 `struct sysinfo` 的指针(参见 `kernel/sysinfo.h`)。内核应填写 `struct sysinfo` 的相关字段：`freemem` 字段应设置为空闲内存的字节数，`nproc` 字段应设置为 `state` 不是 `UNUSED` 的进程数。我们提供了一个测试程序 `sysinfotest`，如果它打印“`sysinfotest: OK`”，你就通过了这个任务。

3.2 实验步骤

3.2.1 System call tracing (moderate)

1. 在 `user/user.h` 中添加 `trace` 系统调用原型，函数原型参考 `user/trace.c` 中调用的形式。

```
char* sbrk(int);
int sleep(int);
int uptime(void);
int trace(int);
```

2. 在 `user/usys.pl` 脚本中添加 `trace` 对应的 `entry`

```
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("trace");
```

3. 在 kernel/syscall.h 中添加 trace 的系统调用号

```
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_trace 22
```

4. 按照要求在 kernel/sysproc.c 中写 trace 系统调用函数 sys_trace()

```
1. uint64
2. sys_trace(void)
3. {
4.     int mask;
5.     // 取 a0 寄存器中的值返回给 mask
6.     if(argint(0, &mask) < 0)
7.         return -1;
8.
9.     // 把 mask 传给现有进程的 mask
10.    myproc()->mask = mask;
11.    return 0;
12. }
```

5. 此时的 make qemu 是运行不成功的, 需要将 mask 加入到 struct proc 中

```
enum procstate state; // Process state
struct proc *parent; // Parent process
void *chan; // If non-zero, sleeping on chan
int killed; // If non-zero, have been killed
int xstate; // Exit status to be returned to parent's wait
int pid; // Process ID
int mask;

// these are private to the process, so p->lock need not be held.
uint64 kstack; // Virtual address of kernel stack
uint64 sz; // Size of process memory (bytes)
pagetable_t pagetable; // User page table
struct trapframe *trapframe; // data page for trampoline.S
```

6. 由于进程 fork 时需要复制进程的状态, 即结构体 proc 中相关的成员变量, 因此需要修改 kernel/proc.c 中的 fork() 函数

```
pid = np->pid;

np->state = RUNNABLE;
np->mask = p->mask;
release(&np->lock);
```

7. 掩码 mask 作为一个 64 位的变量, 从低位开始每一比特位对应一个系统调用, 因此通过移位和按位与操作即可判断当前系统调用是否被 trace, 在 syscall.c 中如下:

```
struct proc *p = myproc();

num = p->trapframe->a7;
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    // 下面是添加的部分
    if((1 << num) & p->mask) {
        printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], p->trapframe->a0);
    }
} else {
    printf("%d %s: unknown sys call %d\n",
        p->pid, p->name, num);
}
```

8. 对于需要输出的进程 PID 以及系统调用的返回值, 都在 proc 结构体的相应成员中有记录, 而系统调用名则需要自行构建一个系统调用名称的数组, 顺序与在 kernel/syscall.h 中定义的系统调用号是对应的, 而由于调用号从 1 开始, 因此数组第一个为空字符串来占位

```
static char *syscall_names[] = {
    "", "fork", "exit", "wait", "pipe",
    "read", "kill", "exec", "fstat", "chdir",
    "dup", "getpid", "sbrk", "sleep", "uptime",
    "open", "write", "mknod", "unlink", "link",
    "mkdir", "close", "trace", "sysinfo"};
```

8. sys_trace() 的外部声明

```
extern uint64 sys_unlink(void);
extern uint64 sys_wait(void);
extern uint64 sys_write(void);
extern uint64 sys_uptime(void);
extern uint64 sys_trace(void);
```

syscalls 的函数指针对应

[SYS_mknod]	uint64 sys_unlink(void)
[SYS_unlink]	sys_unlink,
[SYS_link]	sys_link,
[SYS_mkdir]	sys_mkdir,
[SYS_close]	sys_close,
[SYS_trace]	sys_trace,

9. make qemu 测试，按照网站给出的测试样例进行测试

```
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
```

单项测试

```
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (2.1s)
== Test trace all grep == trace all grep: OK (0.9s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (16.8s)
```

3.2.2 Sysinfo (moderate)

1、修改 Makefile

```
$U/_bin\
$U/_wc\
$U/_zombie\
$U/_trace\
$U/_sysinfotest\
$U/_sysinfo\
```

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

```
int uptime(void);  
int trace(int);  
int sysinfo(struct sysinfo *);
```

```
struct stat;  
struct rtcdate;  
struct sysinfo;
```

3、在 user/usys.pl 中添加 sysinfo 的用户态接口

```
entry("uptime");  
entry("trace");  
entry("sysinfo");
```

4、将 sysinfo 系统调用序号定义在 kernel/syscall.h 中。

```
#define SYS_sysinfo 23
```

5、在 kernel/syscall.c 中新增 sys_sysinfo 函数的定义、在函数指针数组中新增 sys_info 的函数指针、在函数名称数组中新增 sys_sysinfo 的函数调用名称

```
1. static char *syscall_names[] = {  
2.     "", "fork", "exit", "wait", "pipe",  
3.     "read", "kill", "exec", "fstat", "chdir",  
4.     "dup", "getpid", "sbrk", "sleep", "uptime",  
5.     "open", "write", "mknod", "unlink", "link",  
6.     "mkdir", "close", "trace", "sysinfo"};  
7.  
8. extern uint64 sys_sysinfo(void);  
9.  
10. [SYS_sysinfo] sys_sysinfo,
```

6.在 kernel/kalloc.c 中添加 free_mem 函数统计空余内存量，整体代码如下：

```
1. uint64  
2. free_mem(void)  
3. {  
4.     struct run *r;  
5.     // counting the number of free page  
6.     uint64 num = 0;  
7.     // add lock  
8.     acquire(&kmem.lock);  
9.     // r points to freelist  
10.    r = kmem.freelist;  
11.    // while r not null
```



```
12. while (r)
13. {
14.     // the num add one
15.     num++;
16.     // r points to the next
17.     r = r->next;
18. }
19. // release lock
20. release(&kmem.lock);
21. // page multiplicated 4096-byte page
22. return num * PGSIZE;
23. }
```

7、在 kernel/proc.c 中新增函数 nproc 如下，通过该函数以获取可用进程数目：

```
1. // Return the number of processes whose state is not UNUSED
2. uint64
3. nproc(void)
4. {
5.     struct proc *p;
6.     // counting the number of processes
7.     uint64 num = 0;
8.     // traverse all processes
9.     for (p = proc; p < &proc[NPROC]; p++)
10.    {
11.        // add lock
12.        acquire(&p->lock);
13.        // if the processes's state is not UNUSED
14.        if (p->state != UNUSED)
15.        {
16.            // the num add one
17.            num++;
18.        }
19.        // release lock
20.        release(&p->lock);
21.    }
22.    return num;
23. }
```

8、在 kernel/defs.h 中添加上述两个新增函数的声明：

9、开始写 sys_sysinfo(), 需要复制一个 struct sysinfo 返回用户空间。struct sysinfo 的定义在 kernel/sysinfo.h 中。

```
struct sysinfo {
    uint64 freemem;    // amount of free memory (bytes)
    uint64 nproc;      // number of process
};
```

11、在 kernel/sysproc.c 文件中添加 sys_sysinfo 函数的具体实现如下：

```
1.  uint64
2.  sys_sysinfo(void)
3.  {
4.      // addr is a user virtual address, pointing to a struct sysinfo
5.      uint64 addr;
6.      struct sysinfo info;
7.      struct proc *p = myproc();
8.
9.      if (argaddr(0, &addr) < 0)
10.         return -1;
11.     // get the number of bytes of free memory
12.     info.freemem = free_mem();
13.     // get the number of processes whose state is not UNUSED
14.     info.nproc = nproc();
15.
16.     if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
17.         return -1;
18.
19.     return 0;
20. }
```

测试结果

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
```

单项测试结果

```
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (3.8s)
```

make grade 两个实验测试

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (6.6s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.0s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (18.5s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (2.9s)
== Test time ==
time: OK (0.1s)
Score: 35/35
```

3.3 实验中遇到的问题和解决方法

usertest 测试通过，但是样例无反应

```
hart 2 starting
hart 1 starting
init: starting sh
$ syscall
exec syscall failed
$ trace 32 grep hello README
$ trace 2147483647 grep hello README
$ trace 2 usertests forkforkfork
usertests starting
test forkforkfork: OK
ALL TESTS PASSED
```

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

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

原因：没有将 mask 传给现有的进程进行追踪，改正方式就是将 mask 传给现有的进程
第三个例子和第四个例子输出不一样，为什么没有跟踪的进程却还要输出进程的跟踪？第四

个测试 fork 的子孙进程输出不对？为什么？

```
$ grep hello README
5: syscall sbrk -> 16384
5: syscall exec -> 3
5: syscall open -> 3
5: syscall read -> 1023
5: syscall read -> 966
5: syscall read -> 70
5: syscall read -> 0
5: syscall close -> 0
$ trace 2 usertests forkforkfork
6: syscall sbrk -> 16384
6: syscall exec -> 4
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
OK
6: syscall fork -> 69
ALL TESTS PASSED
```

```
np->state = RUNNABLE;
np->mask = p->mask;
release(&np->lock);
```

查看实验指导书，发现漏了一个步骤就是没有将进程的状态进行复制，虽然不知道为什么会这样的错误，但是通过实验指导书的方法去做没有这种错误
在做 sysinfo 实验时，写函数时不知道如何将内核态拷贝为用户态，最后实验指导书推荐的发现可以用 vm.c 中的 copyout() 函数实现。

3.4 实验心得

本次实验感觉按照实验指导书做下来是可以做出来的，但是如果不理解原理是会报很多错误的，还是需要学很多内核态和用户态相关的知识的。

4. lab3: page tables

4.1 实验目的

4.1.1 Print a page table (easy)

定义一个名为 vmprint() 的函数。它应该采用一个 pagetable_t 参数，并以描述的格式打印该可分页的格式。在返回 argc 之前插入 if(p->pid==1) 如果将打印(p->可分页)到 exec.c 中，以打印第一个进程的页面表。如果你通过了制作成绩的个人打印输出测试，你将获得这个作业的全部学分。

4.1.2 A kernel page table per process(hard)

您的第一个工作是修改内核，以便每个进程在内核中执行时都使用它自己的内核页表副本。修改结构流程以维护每个进程的内核页表，并修改调度程序以在切换进程时切换内核页表。对于此步骤，每个进程的内核页表都应该与现有的全局内核页表相同。如果用户测试运行正确，您可以通过实验室的这一部分。

4.1.3 Simplify copyin/copyinstr (hard)

在内核/复制体中调用 `copyin_new`(在内核/复制体中定义);对复制程序和 `copyinstr_new` 执行同样的操作。将用户地址的映射添加到每个进程的内核页表中，以便 `copyin_new` 和 `copyinstr_new` 可以工作。如果用户测试运行正确，并且所有的等级测试都通过，您可以通过此分配。

4.2 实验步骤

4.2.1 Print a page table (easy)

1.在 `kernel/vm.c` 中编写函数 `vmprint()`，在编写一个 `vmprinthelper()`函数，此处可以参考 `freewalk()` 函数的实现。

```
1. void vmprinthelper(pagetable_t pagetable, int level) {
2.     for (int i = 0; i < 512; ++i) {
3.         pte_t pte = pagetable[i];
4.         if (pte & PTE_V) {
5.             switch(level)
6.             {
7.                 case 3:
8.                     printf(".. ");
9.                 case 2:
10.                    printf(".. ");
11.                 case 1:
12.                    printf("...%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
13.            }
14.            pagetable_t child = (pagetable_t) PTE2PA(pte);
15.            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
16.                vmprinthelper(child, level + 1);
17.            }
18.        }
```

```

19.     }
20. }
21.
22. // print page tables lab3-1
23. void vmprint(pagetable_t pagetable) {
24.     printf("page table %p\n", pagetable);
25.     vmprinthelper(pagetable, 1);
26. }

```

2. 在 kernel/defs.h 文件中添加函数原型

```

uint64 walkaddr(pagetable_t, uint64);
int copyout(pagetable_t, uint64, char *, uint64);
int copyin(pagetable_t, char *, uint64, uint64);
int copyinstr(pagetable_t, char *, uint64, uint64);
void vmprint(pagetable_t pagetable); // lab3-1

```

3. 在 kernel/exec.c 的 exec 函数的 return argc 前插入 if(p->pid==1) vmprint(p->pagetable)

```

// print page table - lab3-1
if (p->pid == 1) {
    vmprint(p->pagetable);
}
return argc; // this ends up in a0, the first argument to main(argc, argv)

```

测试实验结果

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f64000
..0: pte 0x0000000021fd8001 pa 0x0000000087f60000
.. ..0: pte 0x0000000021fd7c01 pa 0x0000000087f5f000
.. .. ..0: pte 0x0000000021fd841f pa 0x0000000087f61000
.. .. ..1: pte 0x0000000021fd780f pa 0x0000000087f5e000
.. .. ..2: pte 0x0000000021fd741f pa 0x0000000087f5d000
..255: pte 0x0000000021fd8c01 pa 0x0000000087f63000
.. ..511: pte 0x0000000021fd8801 pa 0x0000000087f62000
.. .. ..510: pte 0x0000000021fed807 pa 0x0000000087fb6000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh

```

单项测试

```

farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-pgtbl pte
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (2.6s)

```


4.2.2 A kernel page table per process(hard)

1. 在 kernel/proc.h 中的 struct proc 结构体中添加成员变量 kpagetable

```
// these are private to the process, so p->lock need not be held.
uint64 kstack;           // Virtual address of kernel stack
uint64 sz;               // Size of process memory (bytes)
pagetable_t pagetable;   // User page table
struct trapframe *trapframe; // data page for trampoline.S
struct context context;   // swtch() here to run process
struct file *ofile[NOFILE]; // Open files
struct inode *cwd;        // Current directory
char name[16];            // Process name (debugging)
pagetable_t kpagetable;   // kernel pagetable - lab3-1
```

2. 构建进程的内核页表的映射函数 proc_kpagetable()

- 参考 kernel/vm.c 中的 kvminit() 函数, kvminit() 中通过调用 kvmmap() 函数完成 UART0、VIRTIO0、CLINT 等部分的映射, 其底层是调用的 mappages() 函数, 默认页表为全局的内核页表 kernel_pagetable. 因此此处不能直接调用该函数, 而是需要替换 kvmmap() 函数从而不是构建全局内核页表.
- 首先在 kernel/vm.c 中仿照 kvmmap() 编写了函数 uvmmmap(), 区别在于添加了 pagetable_t 的参数, 用于传入进程自身的内核页表结构体.

如下图: 可以看到两个函数

```
1. void
2. kvmmap(uint64 va, uint64 pa, uint64 sz, int perm)
3. {
4.     if(mappages(kernel_pagetable, va, sz, pa, perm) != 0)
5.         panic("kvmmap");
6. }
7.
8. // lab3-2
9. void uvmmmap(pagetable_t pagetable, uint64 va, uint64 pa, uint64 sz, int perm)
10. {
11.     if(mappages(pagetable, va, sz, pa, perm) != 0) {
12.         panic("uvmmmap");
13.     }
14. }
```

而 proc_kpagetable() 则基本参照 kvminit() 编写即可.

```
1. pagetable_t proc_kpagetable(struct proc *p) {
2.     pagetable_t kpagetable = uvmmcreate();
3.     if(kpagetable == 0){
4.         return 0;
```

```

5.     }
6.
7.     uvmmmap(kpagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
8.     uvmmmap(kpagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
9.     // uvmmmap(kpagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W); // lab3-3
10.    uvmmmap(kpagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
11.    uvmmmap(kpagetable, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
12.    uvmmmap(kpagetable, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);
13.    uvmmmap(kpagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
14.
15.    return kpagetable;
16. }

```

在 kernel/proc.c 的 allocproc() 函数中调用

```

// process's kernel page table - lab3-2
p->kpagetable = proc_kpagetable(p);
if (p->kpagetable == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
}

// Allocate a page for the process's kernel stack. - lab3-2
char *pa = kalloc();
if(pa == 0) {
    panic("kalloc");
}

```

4. 参考 kernel/proc.c 的 procinit() 函数，将其中为每个进程在全局内核页表中创建内核栈的部分代码注释掉，因为之后进程的内核栈被映射到了自身的内核页表中了。

```

// // Allocate a page for the process's kernel stack.
// // Map it high in memory, followed by an invalid
// // guard page.
// char *pa = kalloc();
// if(pa == 0)
//     panic("kalloc");
// uint64 va = KSTACK((int) (p - proc));
// kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
// p->kstack = va;
}
kvminithart();
}

```


将上述注释的代码复制到 `alloproc()` 函数当中

```
// Allocate a page for the process's kernel stack. - lab3-2
char *pa = kalloc();
if(pa == 0) {
    panic("kalloc");
}
uint64 va = KSTACK(0);
uvmmmap(p->kpagetable, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
p->kstack = va;

// Set up new context to start executing at forkret,
// which returns to user space.
memset(&p->context, 0, sizeof(p->context));
p->context.ra = (uint64)forkret;
```

4. 修改进程调度函数加载内核页表到 SATP 寄存器

代码直接参考 `kvminithart()`, 其中 `w_satp()` 函数用于设置最高级页目录地址的寄存器 SATP, `sfence_vam()` 用于清空当前 TLB.

```
1. void
2. scheduler(void)
3. {
4.     struct proc *p;
5.     struct cpu *c = mycpu();
6.
7.     c->proc = 0;
8.     for(;;){
9.         // Avoid deadlock by ensuring that devices can interrupt.
10.        intr_on();
11.
12.        int found = 0;
13.        for(p = proc; p < &proc[NPROC]; p++) {
14.            acquire(&p->lock);
15.            if(p->state == RUNNABLE) {
16.                // Switch to chosen process. It is the process's job
17.                // to release its lock and then reacquire it
18.                // before jumping back to us.
19.                p->state = RUNNING;
20.                c->proc = p;
21.                // load the process's kernel page table - lab3-2
22.                w_satp(MAKE_SATP(p->kpagetable));
23.                // flush the TLB - lab3-2
24.                sfence_vma();
```

```
25.     swtch(&c->context, &p->context);
26.
27.         // Process is done running for now.
28.         // use kernel_pagetable when no process is running - lab3-2
29.     kvmminithart();
30.         // It should have changed its p->state before coming back.
31.     c->proc = 0;
32.
33.     found = 1;
34. }
35.     release(&p->lock);
36. }
37. #if !defined (LAB_FS)
38.     if(found == 0) {
39.         intr_on();
40.         asm volatile("wfi");
41.     }
42. #else
43.     ;
44. #endif
45. }
46. }
```

5. 释放内核页表和内核栈

参考 `freewalk()` 函数之间进行三级页表结构的清除以及页目录内存释放. 与 `freewalk()` 不同的是, `freewalk()` 执行前需要通过 `uvmunmap()` 将最低级页目录的映射清除, 即最低级页目录的 PTE 均为 0, 负责清除前两级页目录的映射结构和三级页目录的物理内存. 而下述代码则是同时将最低级页目录的 PTE 清零.

```
1. void proc_freekpagetable(pagetable_t kpagetable)
2. {
3.
4.     for(int i = 0; i < 512; i++){
5.         pte_t pte = kpagetable[i];
6.         if((pte & PTE_V)){
7.             kpagetable[i] = 0;
8.             if ((pte & (PTE_R|PTE_W|PTE_X)) == 0) {
9.                 uint64 child = PTE2PA(pte);
10.                proc_freekpagetable((pagetable_t)child);
11.            }
12.        }
13.    }
14.    kfree((void*)kpagetable);
```

修改 `freeproc()` 代码，内核栈 `p->stack` 需要在内核页表 `p->kpagetable` 之前清除

```
// free kernel stack - lab3-2
if(p->kstack) {
    uvmunmap(p->kpagetable, p->kstack, 1, 1);
}
p->kstack = 0;
// free kernel page table without freeing physical memory - lab3-2
if(p->kpagetable){
    proc_freekpagetable(p->kpagetable);
}
```

6. 修改 `kvmppa()` 函数

```
uint64 off = va % PGSIZE;
pte_t *pte;
uint64 pa;

pte = walk(kpagetable, va, 0); // lab3-2
if(pte == 0)
    panic("kvmppa");
if((*pte & PTE_V) == 0)
    panic("kvmppa");
```

测试

```
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
```

4.2.3 Simplify `copyin/copyinstr` (hard)

通过添加用户地址空间的映射到进程的内核页表，这样在内核系统调用 `copyin()` 和

`copyinstr()` 就不用使用 `walk()` 函数让操作系统将虚拟地址换为物理地址进行字符拷贝(因为全局内核页表不知道用户地址空间的映射情况), 而是直接通过 MMU 完成虚拟地址到物理地址的转换. 这需要保证用户页表 `p->pagetable` 变动的同时修改 `p->kpagetable`, 根据指导书主要修改 `fork()`, `exec()`, `sbrk()` 三个函数.

1. 编写用户页表拷贝到内核页表的函数 `u2kvmcopy()`

参考 `kernel/vm.c` 中的 `uvmcopy()` 函数, 该函数是在 `fork()` 时用于将父进程用户页表拷贝到子进程.

此处的 `uvmcopy()` 在拷贝时并没有使用写时复制, 而是直接分配相应的页面(物理内存)并复制字节. 而将用户页表拷贝到内核页表时, 物理地址空间实际上是不变的, 只是多了一次映射, 因此 `u2kvmcopy()` 中就没有使用 `kalloc()` 分配页面, 而是直接复用 `walk()` 返回的物理地址.

```

1.  int u2kvmcopy(pagetable_t upagetable, pagetable_t kpagetable, uint64 begin,
    uint64 end) {
2.      pte_t *pte;
3.      uint64 pa, i;
4.      uint flags;
5.      uint64 begin_page = PGROUNDUP(begin);
6.      for(i = begin_page; i < end; i += PGSIZE){
7.          if((pte = walk(upagetable, i, 0)) == 0)
8.              panic("uvmcopy2kvm: pte should exist");
9.          if((*pte & PTE_V) == 0)
10.             panic("uvmcopy2kvm: page not present");
11.             pa = PTE2PA(*pte);
12.             flags = PTE_FLAGS(*pte) & (~PTE_U); // clear PTE_U flag
13.             // map to the physical memory same as user's pa instead of kalloc()

14.             if(mappages(kpagetable, i, PGSIZE, pa, flags) != 0){
15.                 goto err;
16.             }
17.         }
18.         return 0;
19.
20. err:
21.     uvmunmap(kpagetable, begin_page, (i - begin_page) / PGSIZE, 0);
22.     return -1;
23. }
```

2. 修改 `fork` 函数, `kernel/proc.c` 中的 `fork()` 函数用于创建子进程, 其中在用户页表从父进程复制到子进程之后, 同样要将子进程的用户页表拷贝到子进程的内核页表, 使用 `u2kvmcopy()`.

```

if((np = allocproc()) == 0){
    return -1;
}

// Copy user memory from parent to child.
if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
}
np->sz = p->sz;
// copy user page table to kernel page table - lab3-3
if(u2kvmcopy(np->pagetable, np->kpagetable, 0, np->sz) < 0) {
    freeproc(np);
    release(&np->lock);
    return -1;
}

```

3. 修改 exec() 函数

kernel/exec.c 中的 exec() 函数用于替换进程镜像，在替换之后会将原用户页表释放替换为新的用户页表，因此需要同样更新进程的内核页表。

由于内核页表中虚拟地址实际上指向的也是用户空间的物理地址，因此不需要像用户页表一样连同物理空间一并释放，而是使用 uvmunmap() 清除映射，然后使用 u2kvmcopy() 进行页表的复制。

```

p->sz = sz;
p->trapframe->epc = elf.entry; // initial program counter = main
p->trapframe->sp = sp; // initial stack pointer
proc_freepagetable(oldpagetable, oldsz);

// unmap old kernel page table, and copy the new one - lab3-3
uvmunmap(p->kpagetable, 0, PGROUNDUP(oldsz)/PGSIZE, 0);
if(u2kvmcopy(p->pagetable, p->kpagetable, 0, p->sz) < 0){
    goto bad;
}

// print page table - lab3-1
if (p->pid == 1) {
    vmprint(p->pagetable);
}
return argc; // this ends up in a0, the first argument to main(argc, argv)

```

4. 修改 growproc() 函数

sbrk() 函数即系统调用 sys_brk() 函数，最终会调用 kernel/proc.c 中的 growproc() 函数，用来增长或减少虚拟内存空间。根据指导书要求，要保证用户空间的大小在 PLIC 部分之下，因此，在 $n > 0$ 时要判断 $sz + n > \text{PLIC}$ 的情况，满足则返回失败。此处不取等，是因为 $sz + n$ 是个空间大小，在与 PLIC 相等时恰好未使用 PLIC 地址。


```

sz = p->sz;
if(n > 0){
    // prevent process from growing to PLIC address - lab3-3
    if(sz + n > PLIC){
        return -1;
    }
    if((sz = uvmmalloc(p->pagetable, sz, sz + n)) == 0) {
        return -1;
    }
    // copy the increase user page table to kernel page table - lab3-3
    if(u2kvmcopy(p->pagetable, p->kpagetable, p->sz, sz) < 0){
        return -1;
    }
} else if(n < 0){
    sz = uvmmdealloc(p->pagetable, sz, sz + n);
    // free process's kernel page table without free physical memory - lab3-3
    if (PGROUNDUP(sz) < PGROUNDUP(p->sz)) {
        uvmmunmap(p->kpagetable, PGROUNDUP(sz),
            (PGROUNDUP(p->sz) - PGROUNDUP(sz)) / PGSIZE, 0);
    }
}

```

5. 修改 userinit() 函数

kernel/proc.c 中的 userinit() 函数用于初始化 xv6 启动时第一个用户进程，该进程的加载是独立的，因此也需要将其用户页表拷贝到内核页表。

```

// and data into it.
uvminit(p->pagetable, initcode, sizeof(initcode));
p->sz = PGSIZE;
// init kernel pagetable - lab3-3
u2kvmcopy(p->pagetable, p->kpagetable, 0, p->sz);
// prepare for the very first "return" from kernel to user.
p->trapframe->epc = 0; // user program counter
p->trapframe->sp = PGSIZE; // user stack pointer
safestrcpy(p->name, "initcode", sizeof(p->name));

```

6. 替换 copyin() 和 copyinstr()

直接将 copyin() 和 copyinstr() 两个函数的原有代码注释掉，并分别调用已提供好的 kernel/vmcopyin.c 中的 copyin_new() 和 copyinstr_new() 函数

```

int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
    return copyin_new(pagetable, dst, srcva, len);
}
// uint64 n, va0, pa0;
//
// while(len > 0){
//     va0 = PGROUNDDOWN(srcva); // get page virtual address of srcva
//     pa0 = walkaddr(pagetable, va0); // get physical address of va0
//     if(pa0 == 0)
//         return -1;
// }

```

```
int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
    return copyinstr_new(pagetable, dst, srcva, max);
    // uint64 n, va0, pa0;
    // int got_null = 0;
    //
    // while(got_null == 0 && max > 0){
```

make qemu usertests

```
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
```

make grade 测试

```
count copyin: OK (1.1s)
== Test usertests ==
$ make qemu-gdb
(365.6s)
== 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
```

4.3 实验中遇到的问题及解决方法

1.xv6 卡住不能正常运行

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f62000
..0: pte 0x0000000021fd7801 pa 0x0000000087f5e000
.. ..0: pte 0x0000000021fd7401 pa 0x0000000087f5d000
.. ..0: pte 0x0000000021fd7c1f pa 0x0000000087f5f000
.. ..1: pte 0x0000000021fd700f pa 0x0000000087f5c000
.. ..2: pte 0x0000000021fd6c1f pa 0x0000000087f5b000
..255: pte 0x0000000021fd8401 pa 0x0000000087f61000
.. ..511: pte 0x0000000021fd8001 pa 0x0000000087f60000
.. ..510: pte 0x0000000021fed807 pa 0x0000000087fb6000
.. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

解决：这是由于 `u2kvmcopy()` 分配内存没有映射到用户空间

2.xv6 在实验 1 这块就卡住, `proc.c` 中的 `alloproc()` 函数中 `p->context.sp = p->kstack + PGSIZE`; 这一行代码设置了内核栈顶指针, 因此在这之前必须保证内核栈已经分配且映射完成.

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
█
```

3.usertests 报错 remap

```
.. ..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
$ usertests
usertests starting
panic: remap
█
```

解决方法：因为其存在重映射的可能, 也就从另一方面说明了该部分应该不会被实际映射, 具体两种的解决方法, 第一种是写个新的 `mappages` 将 `remap` 的 `panic` 注释掉, 或者是 `proc_kpagetable()` 的 `clint` 映射部分注释掉

4.4 实验心得

这个实验是做起来比较困难的，第一个实验按照实验指导书上的方案是可以做出来，比如三级页表以及可以借鉴 `freewalk` 函数，在做第二个实验的时候，需要理解用户地址在内核中无效，因此，当内核需要使用在系统调用中传递的用户指针（例如，传递给 `write()` 的缓冲区指针）时，内核必须首先将指针转换为物理地址，所以实验二和实验三大部分玩的是内核态和用户态的虚拟地址向物理地址的映射，但是难点在于需要修改很多的地方，所以可以单独写一个差不多的函数在这个函数上进行修改测试，不然容易和原有的代码进行混淆，实验三是将用户空间的映射添加到每个进程的内核页表（上一节中创建），以允许 `copyin`（和相关的字符串函数 `copyinstr`）直接解引用用户指针，感觉实验三都是在现有的函数上进行修改调用的，但是也花费了比较长的时间，主要原因是在做实验三是修改部分函数将实验二的一些函数也删除了，导致我后面 `debug` 的时候一直找不到错误，回过头才发现是实验二代码的问题，也有报错 `remap` 还有 `kernel` 卡住的情况，这个实验是做的时间最长并且我认为难度最大的实验

5. lab4 :Trap

5.1 实验目的

5.1.1 RISC-V assembly ([easy](#))

首先使用栈做一个热身练习，然后实现一个用户级陷阱处理的示例。

5.1.2 Backtrace ([moderate](#))

编译器向每一个栈帧中放置一个帧指针（`frame pointer`）保存调用者帧指针的地址。写一个 `backtrace` 函数，你的 `backtrace` 应当使用这些帧指针来遍历栈，并在每个栈帧中打印保存的返回地址。

5.1.3 Alarm

在这个练习中你将向 `XV6` 添加一个特性，在进程使用 `CPU` 的时间内，`XV6` 定期向进程发出警报。对于那些希望限制 `CPU` 时间消耗的受计算限制的进程，或者对于那些计算的同时执行某些周期性操作的进程可能很有用。更普遍的说，你将实现用户级中断/故障处理程序的一种初级形式。例如，你可以在应用程序中使用类似的一些东西处理页面故障。如

如果你的解决方案通过了 `alarmtest` 和 `usertests` 就是正确的。

5.2 实验步骤

5.2.1 RISC-V assembly ([easy](#))

1. 通过阅读网站和书籍对于汇编给出问题的回答，首先键入 `ls user | grep call` 再 `make fs.img` 产生 `call.asm` 文件

```
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ls user | grep call
call.c
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ make fs.img
gcc -DSOL_TRAPS -DLAB_TRAPS -Werror -Wall -I. -o mkfs/mkfs mkfs/mkfs.c
```

2. 在终端输入 `vim user/user.asm` 就可以看到汇编文件的代码，如下图所示

```
Disassembly of section .text:

0000000000000000 <g>:
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

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

000000000000000e <f>:

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);
```

于是，我将回答以下问题

Q: 哪些寄存器存储了函数调用的参数？举个例子，`main` 调用 `printf` 的时候，13 被存在了哪个寄存器中？ A: `a0-a7; a2;`

```

printf("%d %d\n", f(8)+1, 13):
24: 4635          li    a2,13
26: 45b1          ll    a1,12
28: 00000517      auipc  a0,0x0
2c: 7b850513      addi   a0,a0,1976 # 7e0 <main>
30: 00000097      auipc  ra,0x0
34: 608080e7      jalr   1544(ra) # 638 <printf>
// // Q5 - lab4-1

```

Q: main 中调用函数 f 对应的汇编代码在哪? 对 g 的调用呢? (提示: 编译器有可能会内链(inline)一些函数)

A: 没有这样的代码。g(x) 被内链到 f(x) 中, 然后 f(x) 又被进一步内链到 main() 中

Q: printf 函数所在的地址是? A: 0x0000000000000638, main 中使用 pc 相对寻址来计算得到这个地址。

```

30: 00000097      auipc  ra,0x0
34: 608080e7      jalr   1544(ra) # 638 <printf>
// // Q5 - lab4-1
// unsigned int i = 0x00646c72;

```

```

0000000000000638 <printf>:
void
printf(const char *fmt, ...)
{
638: 711d          addi   sp,sp,-96
63a: ec06          sd     ra,24(sp)
63c: e822          sd     s0,16(sp)
63e: 1000          addi   s0,sp,32
640: e40c          sd     a1,8(s0)
642: e810          sd     a2,16(s0)
644: ec14          sd     a3,24(s0)
646: f018          sd     a4,32(s0)
648: f41c          sd     a5,40(s0)
64a: 03043823      sd     a6,48(s0)
/638

```

Q: 在 main 中 jalr 跳转到 printf 之后, ra 的值是什么? A: 0x0000000000000038, jalr 指令的下一条汇编指令的地址。跳转 \$ra + 1544 处执行

```

30: 00000097      auipc  ra,0x0
34: 608080e7      jalr   1544(ra) # 638 <printf>
// // Q5 - lab4-1
// unsigned int i = 0x00646c72;

```

Q: 运行下面的代码 unsigned int i = 0x00646c72; printf("H%x Wo%s", 57616, &i); 输出是什么? 如果 RISC-V 是大端序的, 要实现同样的效果, 需要将 i 设置为什么? 需要将 57616 修改为别的值吗?

A: "HE110 World", 0x726c6400, 不需要, 57616 的十六进制是 E110, 无论端序 (十六进制和内存中的表示不是同一个概念)

```
$ call
12 13
HE110 World
```

Q: 在下面的代码中, 'y=' 之后会答应什么? (note: 答案不是一个具体的值) 为什么?
printf("x=%d y=%d", 3);

A: 输出的是一个受调用前的代码影响的“随机”的值。因为 printf 尝试读的参数数量比提供的参数数量多。第二个参数 '3' 通过 a1 传递, 而第三个参数对应的寄存器 a2 在调用前不会被设置为任何具体的值, 而是会包含调用发生前的任何已经在里面的值。

5.2.2 Backtrace ([moderate](#))

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

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

2. 在 kernel/printf.c 中编写函数 backtrace() 输出所有栈帧, 如下图所示, 强制类型转换的对象要么为 (uint64 *) (fp-8) 和 (uint64 *) (fp-16), 因为 8 和 16 的单位是字节; 或者为 (uint64 *) fp-1 和 (uint64 *) fp-2, 因为此时 1 和 2 是以 (uint64 *) 指针大小(8 字节)为单位的。

```
// print the return address - lab4-2
void backtrace() {
    uint64 fp = r_fp();
    uint64 top = PGROUNDDUP(fp);
    uint64 bottom = PGROUNDDOWN(fp);
    for (; fp >= bottom && fp < top; fp = *((uint64 *) (fp - 16))) {
        printf("%p\n", *((uint64 *) (fp - 8)));
    }
}
```

3. 在 defs.h 中添加 backtrace 函数的原型

```
// printf.c
void printf(char*, ...);
void panic(char*) __attribute__((noreturn));
void printfinit(void);
void backtrace(); // lab4-2
```

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

```
ticks0 = ticks;
while(ticks - ticks0 < n){
    if(myproc()->killed){
        release(&tickslock);
        return -1;
    }
    sleep(&ticks, &tickslock);
}
release(&tickslock);
backtrace(); // lab4-2
return 0;
```

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

```
panic(char *s)
{
    pr.locking = 0;
    printf("panic: ");
    printf(s);
    printf("\n");
    backtrace(); // lab4-2
    panicked = 1; // freeze uart output from other CPUs
    for(;;)
        ;
}
```

调试 gdb, 如下连接 gdb

```
farzena@farzena-virtual-machine:~/Desktop$ ls
farzena@farzena-virtual-machine:~/Desktop$ cd
farzena@farzena-virtual-machine:~$ cd work
farzena@farzena-virtual-machine:~/work$ cd xv6-labs-2020
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ make qemu-gdb
*** Now run 'gdb' in another window.
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-devi
ce,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26000
```

在另一个窗口执行 gdb-multiarch kernel/kernel


```

farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ gdb-multiarch kernel/kernel
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel/kernel...
warning: File "/home/farzena/work/xv6-labs-2020/.gdbinit" auto-loading has been

```

两个窗口用 localhost 连接

```

(gdb) set architecture riscv:rv64
The target architecture is set to "riscv:rv64".
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000000000000100 in ?? ()
(gdb)

```

对刚刚的函数设置断点

```

(gdb) b backtrace
Breakpoint 1 at 0x80000574: file kernel/printf.c, line 138.
(gdb) c
Continuing.

```

在第一个窗口执行 bttest 指令

```

xv6 kernel is booting

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

```

第二个窗口显示入口地址

```

138     void backtrace() {
(gdb) info frame
Stack level 0, frame at 0x3fffff9f80:
pc = 0x80000574 in backtrace (kernel/printf.c:138); saved pc = 0x80002d7c
called by frame at 0x3fffff9fc0
source language c.
Arglist at 0x3fffff9f80, args:
Locals at 0x3fffff9f80, Previous frame's sp is 0x3fffff9f80
Could not fetch register "ustatus"; remote failure reply 'E14'

```

在窗口当中输入 if 0（查看当前的栈帧）


```

(gdb) i f 0
Stack frame at 0x3fffff9f80:
pc = 0x80000584 in backtrace (kernel/printf.c:139); saved pc = 0x80002d7c
called by frame at 0x3fffff9fc0
source language c.
Arglist at 0x3fffff9f80, args:
Locals at 0x3fffff9f80, Previous frame's sp is 0x3fffff9f80
Saved registers:
ra at 0x3fffff9f78, fp at 0x3fffff9f70, s1 at 0x3fffff9f68,
s2 at 0x3fffff9f60, s3 at 0x3fffff9f58, s4 at 0x3fffff9f50,
pc at 0x3fffff9f78Could not fetch register "ustatus"; remote failure reply 'E14'
(gdb) p *0x3fffff9f78

```

如上图所示的 0x3fffff9f78 就是下个函数执行的地址，下面测试一下，可以看到两张图的地址是对应的关系

```

(gdb) p *0x3fffff9f78
$1 = -2147472004
(gdb) p/x *0x3fffff9f78
$2 = 0x80002d7c
(gdb) p/x *0x3fffff9f60
$3 = 0x00000000

```

```

0x00000000080002d7c
0x00000000080002be0
0x00000000080002890
$ $ bttest

```

或者直接输入 bt（查看函数调用关系），可以看到所指的地址都是相应一致的

```

(gdb) x/i *0x80002d7c
0xffffffffff853e4781: Cannot access memory at address 0xfffffffff8
(gdb) x/i 0x80002d7c
0x80002d7c <sys_sleep+142>: li    a5,0
(gdb)
0x80002d7e <sys_sleep+144>: mv    a0,a5
(gdb) bt
#0  backtrace () at kernel/printf.c:140
#1  0x00000000080002d7c in sys_sleep () at kernel/sysproc.c:73
#2  0x00000000080002be0 in syscall () at kernel/syscall.c:144
#3  0x00000000080002890 in usertrap () at kernel/trap.c:67
#4  0x0000000000000012 in ?? ()
(gdb)

```

```

0x00000000080002d7c
0x00000000080002be0
0x00000000080002890
$ $ bttest

```

```
xv6 kernel is booting
```

```
hart 2 starting
hart 1 starting
init: starting sh
$ bttest
0x0000000080002d7c
0x0000000080002be0
0x0000000080002890
```

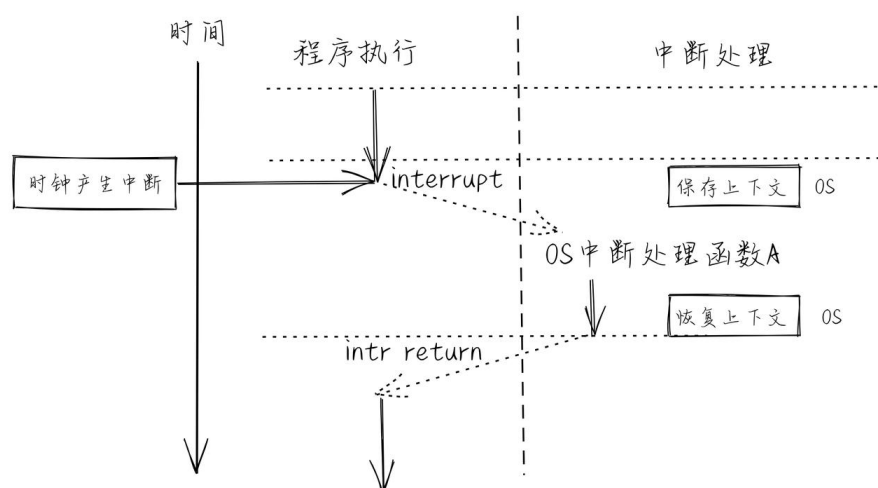
单项测试

```
● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-traps backtrace
make: 'kernel/kernel' is up to date.
== Test backtrace test == backtrace test: OK (2.1s)
```

```
● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ addr2line -e kernel/kernel 0x0000000080002d7c
/home/farzena/work/xv6-labs-2020/kernel/sysproc.c:74
● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ addr2line -e kernel/kernel 0x0000000080002be0
/home/farzena/work/xv6-labs-2020/kernel/syscall.c:144
● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ addr2line -e kernel/kernel 0x0000000080002890
/home/farzena/work/xv6-labs-2020/kernel/trap.c:76
```

5.2.3 Alarm

执行一个周期性的报警函数，对于限制 cpu 使用时间的函数会很有用，每当经过 interval 这个时钟周期后，就执行 handler 这个函数，handler 是一个函数指针



1. 在 XV6 的存储库中找到名为 user/alarmtest.c 的文件。将其添加到 Makefile

```
$U/_sh\  
$U/_stressfs\  
$U/_usertests\  
$U/_grind\  
$U/_wc\  
$U/_zombie\  
$U/_alarmtest\
```

2. 放入 user/user.h 的正确声明是:

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

```
int sigreturn(void);
```

```
int sleep(int);  
int uptime(void);  
int sigalarm(int ticks, void (*handler)()); // lab4-3  
int sigreturn(void); // lab4-3
```

4. 更新 user/usys.pl (此文件生成 user/usys.S)、kernel/syscall.h 和 kernel/syscall.c 以允许 alarmtest 调用 sigalarm 和 sigreturn 系统调用,

```
sigreturn:  
li a7, SYS_sigreturn  
ecall  
ret
```

```
21 #define SYS_mkdir 20  
22 #define SYS_close 21  
23 #define SYS_sigalarm 22 // lab4-3  
24 #define SYS_sigreturn 23 // lab4-3
```

```
[SYS_close] sys_close,  
[SYS_sigalarm] sys_sigalarm, // lab4-3  
[SYS_sigreturn] sys_sigreturn, // lab4-3  
};
```

```
extern uint64 sys_uptime(void);  
extern uint64 sys_sigalarm(void); // lab4-3  
extern uint64 sys_sigreturn(void); // lab4-3
```

4. 目前来说, 你的 sys_sigreturn 系统调用返回应该是零。

5. `sys_sigalarm()` 应该将报警间隔和指向处理程序函数的指针存储在 `struct proc` 的新字段中 (位于 `kernel/proc.h`)

```
// lab4-3
p = myproc(); // 用 myproc 函数获得当前的进程
p->interval = interval;
p->handler = handler;
p->passedticks = 0;
```

```
int interval;           // alarm interval - la
uint64 handler;         // pointer to the hand
int passedticks;        // ticks have passed s
struct trapframe* trapframecopy; // the copy of
```

6. 在 `struct proc` 新增一个新字段。用于跟踪自上一次调用 (或直到下一次调用) 到进程的报警处理程序间经历了多少滴答; 可以在 `proc.c` 的 `allocproc()` 中初始化 `proc` 字段。

```
p->context.sp = p->kstack + PGSIZE;
p->interval = 0; // lab4-3
p->handler = 0; // lab4-3
p->passedticks = 0; // lab4-3
p->trapframecopy = 0; // lab4-3
return p;
```

6. 每一个滴答声, 硬件时钟就会强制一个中断, 这个中断在 `kernel/trap.c` 中的 `usertrap()` 中处理, 这里需要注意到, 在 `usertrap()` 中时页表已经切换为内核页表 (切换工作在 `uservec` 函数中完成), 而 `handler` 很显然是用户空间下的函数虚拟地址, 因此不能直接调用。这里实际上并没有直接调用, 而是将 `p->trapframe->epc` 置为 `p->handler`, 这样在返回到用户空间时, 程序计数器为 `handler` 定时函数的地址, 便达到了执行定时函数的目的。

```
1. if(which_dev == 2){ // timer interrupt
2.     if(p->interval != 0 && ++p->passedticks == p->interval){
3.         p->passedticks = 0;
4.         p->trapframe->epc = p->handler; // user space
5.     }
6. }
```

此时 `test0` 是通过的

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



```

.....alarm!
test0 passed
test1 start
...alarm!
..alarm!
..alarm!
..alarm!
...alarm!
...alarm!
..alarm!
...alarm!
..alarm!
....alarm!

test1 failed: foo() executed fewer times than it was called
test1 failed: foo() executed fewer times than it was called
test1 failed: foo() executed fewer times than it was called

```

实验到此为止，test0 是可以通过的，但是其他两个 test1 和 test2 是不通过的，这是因为我们执行完中断的操作之后，并没有进行恢复 sigalarm(interval, handler) 和 sigreturn() 两个函数是配合使用的，在 handler 函数返回前会调用 sigreturn()。

根据 test0 的做法可以看到，调用定时函数 handler 实际上是通过修改 trapframe->epc 进而在返回到用户空间时调用定时函数。但这也同时产生了一个问题，即原本的 epc 已被覆盖，无法回到中断前的用户代码执行的位置，同时在执行 handler() 函数后，相关的寄存器的值也会受到影响。因此考虑要在 sigalarm() 函数中将寄存器值进行保存，在 sigreturn() 函数中进行恢复。这样在执行完 sigreturn() 后程序能够回到原来的执行位置。

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

```

char name[16];           // Process name (debugging)
int interval;            // alarm interval - lab4-3
uint64 handler;          // pointer to the handler f
int passedticks;         // ticks have passed since
struct trapframe* trapframecopy; // the copy of tra

```

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

```

1. void
2. usertrap(void)
3. {
4.     // ...
5.     // lab4-3
6.     if(which_dev == 2){

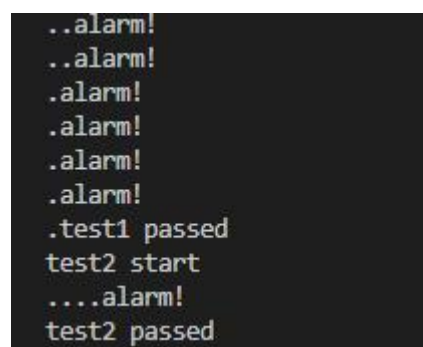
```

```
7. if(p->interval != 0 && ++p->passedticks == p->interval){
8.     // 使用 trapframe 后的一部分内存, trapframe 大小为 288B, 因此只要在
    trapframe 地址后 288 以上地址都可, 此处 512 只是为了取整数幂
9.     p->trapframecopy = p->trapframe + 512;
10.    memmove(p->trapframecopy, p->trapframe, sizeof(struct trapframe));
11.    p->trapframe->epc = p->handler;    // execute handler() when return to
    user space
12. }
13. }
14. // ...
15. }
```

3. 在 sys_sigreturn() 中将副本恢复到原 trapframe.

```
1. uint64 sys_sigreturn(void) {
2.     struct proc* p = myproc();
3.     // trapframecopy 必须是和 trapframe 一样的, 或者差个大小
4.     if(p->trapframecopy != p->trapframe + 512) {
5.         return -1;
6.     }
7.     memmove(p->trapframe, p->trapframecopy, sizeof(struct trapframe));    //
    拷贝
8.     p->passedticks = 0;    // 初始化, 为下一次恢复
9.     p->trapframecopy = 0;
10.    return 0;
11. }
```

测试: 两个测试均通过



```
..alarm!
..alarm!
.alarm!
.alarm!
.alarm!
.alarm!
.test1 passed
test2 start
....alarm!
test2 passed
```

usertests 也通过


```
0x0000000000000000
OK
test exitput: OK
test input: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

5.4 实验中遇到的问题及解决方式

1. make grade 测试时遇见的问题，测试超时

```
alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
Timeout! usertests: FAIL (300.5s)
...
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: qemu-system-riscv64: terminating on signal 15 from pid 46900 (make)
MISSING '^ALL TESTS PASSED$'
QEMU output saved to xv6.out.usertests
== Test time ==
time: OK
Score: 66/85
```

解决方法：在 lab-grade 文件当中修改时间大一点

2. 一开始不知道这个 jalr 是代表的什么意思，通过教程查询汇编代码可以看到，jalr 是指令的下一条汇编指令的地址，跳转 \$ra + 1544 处执行

```
30: 00000097          auipc   ra,0x0
34: 608080e7          jalr    1544(ra) # 638 <printf>
// // Q5 - lab4-1
// unsigned int i = 0x00646c72;
```

```

00000000000000638 <printf>:
void
printf(const char *fmt, ...)
{
638:    711d                addi    sp,sp,-96
63a:    ec06                sd      ra,24(sp)
63c:    e822                sd      s0,16(sp)
63e:    1000                addi    s0,sp,32
640:    e40c                sd      a1,8(s0)
642:    e810                sd      a2,16(s0)
644:    ec14                sd      a3,24(s0)
646:    f018                sd      a4,32(s0)
648:    f41c                sd      a5,40(s0)
64a:    03043823            sd      a6,48(s0)
/638

```

3. 不理解什么是大端序和小端序的意思

查阅资料：大端序：高位字节在低地址。

小端序：低位字节在低地址。

意思就是比如例子给出的 57616 它的二进制是 0x726c6400，这是小端序的，如果要改成大端序的话就是 0x0064c672

4. 在做 backtrace 实验的过程当中写 backtrace 函数的过程当中，不知道如何将二进制转化为 16 进制

解答：在阅读了参考书之后，看到 1 和 2 是以 (uint64*) 指针大小(8 字节)为单位的，所以强制类型转换的对象要么为 (uint64 *) (fp-8) 和 (uint64 *) (fp-16)，或者为 (uint64 *) fp - 1 和 (uint64 *) fp - 2，刚开始转化的代码是这样的，这样结果意思就是 fp 每次移动两个单位，而原则上是移动一个单位，所以是 (uint64 *) (fp-8)

```

uint64 top = PGROUNDUP(fp);
uint64 bottom = PGROUNDUP(fp);
for (; fp >= bottom && fp < top; fp = *((uint64 *) (fp - 16))) {
    printf("%p\n", *((uint64 *) (fp - 16)));
}

```

```

0x0000003fffff9fc0
0x0000003fffff9fe0
0x0000003fffffa000

```

5. 在做 alarm 的 test1 和 test2 时，单纯将保留进程的副本保存到了函数体内，并没有保存到进程的结构体当中，这就导致 alarmreturn 函数在执行获取副本的过程当中出现错误，报错如下图所示

```

kernel/proc.c: In function 'allocproc':
kernel/proc.c:132:6: error: 'struct proc' has no member named 'trapframecopy'; did you mean 'trapframe'?
132 | p->trapframecopy = 0; // lab4-3
    |      ^
    |      trapframe
make: *** [<built-in>: kernel/proc.o] Error 1

```

解决方法：在结构体当中声明

6. 对副本 `trapframecopy` 与 `trapframe` 之间的拷贝操作，直接结构体的赋值，拷贝比较慢，容易出错

```
$ alarmtetst
exec alarmtetst failed
$ alarmtest
test0 start
.....alarm!
usertrap(): unexpected scause 0x000000000000000c pid=5
sepc=0x0505050505050505 stval=0x0505050505050505
```

```
test1 start
..alarm!
.....
test1 failed: too few calls to the handler
test2 start
.....alarm!
test2 passed
```

解决方法：`memcpy()` 字节拷贝：即直接将结构体视为字节流，进行字节拷贝。经过实际测试，发现使用该方式比结构体赋值的速度更快。

5.4 实验心得

本实验主要是中断机制和汇编地址栈帧的转换学习，第一个实验主要是旨在让我去了解栈的指针，栈的地址，通过使用简单的汇编语句，知道每个语句的地址，平时函数中的变量保存在哪个寄存器当中，第二个实验是在实验 1 的基础上去找遍历每个函数的栈帧，第三个实验是在理解上个学期所学的中断机制的基础上，对在系统进行中断处理过程中中断前后系统做出的反应的一个实验，需要在中断前将中断的信息保留下来，为中断结束，操作系统继续执行做准备。

6. lab5: xv6 lazy page allocation

6.1 实验目的

6.1.1 Eliminate allocation from `sbrk()` (easy)

你的首项任务是删除 `sbrk(n)` 系统调用中的页面分配代码（位于 `sysproc.c` 中的函数 `sys_sbrk()`）。`sbrk(n)` 系统调用将进程的内存大小增加 `n` 个字节，然后返回新分配区域的开始部分（即旧的大小）。新的 `sbrk(n)` 应该只将进程的大小（`myproc()->sz`）增加 `n`，然后返回旧的大小。它不应该分配内存——因此您应该删除对 `growproc()` 的调用（但是您仍然需要增加进程的大小！）

6.1.2 Lazy allocation (moderate)

修改 `trap.c` 中的代码以响应来自用户空间的页面错误，方法是新分配一个物理页面并映射到发生错误的地址，然后返回到用户空间，让进程继续执行。您应该在生成“`usertrap(): ...`”消息的 `printf` 调用之前添加代码。你可以修改任何其他 `xv6` 内核代码，以使 `echo hi` 正常工作

6.1.3 Lazytests and Usertests (moderate)

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

6.2 实验步骤

6.2.1 Eliminate allocation from `sbrk()` (easy)

```
int addr;
int n;
struct proc *p;

if(argint(0, &n) < 0)
    return -1;
p = myproc();
addr = p->sz;
// if(growproc(n) < 0)
//     return -1;
myproc()->sz += n;
```

修改 `sys_sbrk()` 函数，将原本调用的 `growproc()` 函数注释掉，而是直接增加 `myproc()->sz`

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=
sepc=0x000000000000012ac stval=0x00000000
panic: uvmunmap: not mapped
```

可以看到 `SCAUSE` 寄存器的值为 15，如下在 RISC-V privileged instructions 中图所示，15 对应 Store/AMO page fault，而根据 `sepc` 的值在 `user/sh.asm` 文件中可以找到对应汇编代码，可以看到是一个 `sw` 指令，用于向内存中写入 1 个字，由于上述修改使得没有实际分配物理内存，从而引发 `page fault`。

0	9	Environment call from S-mode
0	10-11	Reserved
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	16-23	Reserved

```

2688      hp->s.size = nu;
2689      12ac: 01652423      sw    s6,8(a0)

```

6.2.2 Lazy allocation (moderate)

最终结果：在 kernel/trap.c 中添加代码来处理 page fault 分配物理内存。

1. 在 kernel/trap.c 的 usertrap() 中添加对 page fault 的处理.根据指导书提示,当 r_scause() 的值为 13 和 15 时为需要处理的 page fault 情况。
 2. 然后参考 growproc() 函数中调用的 uvmmalloc() 代码,调用 kalloc() 函数为引发 page fault 时记录在寄存器 STVAL 中出错的地址(由 r_stval() 获得)分配一个物理页,然后调用 mappages() 函数在用户页表中添加虚拟页到物理页的映射。
- 对于中途出现的错误,则是将 p->killed 置 1, 标记当前进程并在随后被杀死.

```

else if (r_scause() == 13 || r_scause() == 15) { // lab5-2
    char *pa;
    uint64 va = r_stval(); //设置一个64位整型值为出错的值
    // - lab5-3
    if (va >= p->sz) //不合法的虚拟地址
    {
        printf("usertrap(): invalid va=%p higher than p->sz=%p\n", va, p->sz);
        p->killed = 1;
        goto end;
    }

    if ((pa = kalloc()) == 0) //对页进行分配
    {
        printf("usertrap(): kalloc() failed\n"); //分配不成功
        p->killed = 1;
        goto end;
    }
    memset(pa, 0, PGSIZE); //成功获得, 就将页置为0
    // 下面就是对页表进行映射
    if (mappages(p->pagetable, PGROUNDOWN(va), PGSIZE, (uint64) pa, PTE_W | PTE_R | PTE_U) != 0) {
        kfree(pa);
        printf("usertrap(): mappages() failed\n");
        p->killed = 1;
        goto end;
    }
}

```

实验做到这个位置, 虽然问题是解决了, 但是会报出 panic 的问题, 这时找到 kernel/vm.c 的文件当中, 找到 uvmunmap 函数

在 xv6 的原本实现中, 由于是 Eager Allocation, 所以不存在分配的虚拟内存不对应物理内存的情况, 因此在 uvmunmap() 函数中取消映射时对于虚拟地址对应的 PTE 若是无效的, 便会引发 panic. 而此处改为 Lazy allocation 后, 对于用户进程的虚拟空间, 会出现有的虚拟内存并不对应实际的物理内存的情况, 即页表中的 PTE 是无效的. 因此此时便不能引发 panic, 而是选择跳过该 PTE。


```

    }
    if((*pte & PTE_V) == 0) {
        continue; // lab5-2
    }
    // panic("uvmunmap: not mapped"); // lab5-2
}
if(PTE_FLAGS(*pte) == PTE_V)

```

测试

```

hart 2 starting
hart 1 starting
init: starting sh
$ echo hi
hi
$ QEMU: Terminated

```

6.2.3 Lazytests and Usertests (moderate)

1. 处理 `sbrk()` 参数为负的情况，如果某个进程在高于 `sbrk()` 分配的任何虚拟内存地址上出现页错误，则终止该进程。

```

// 处理参数为负数的情况和进程高于虚拟地址的情况 lab5-3
p->sz = uvmdealloc(p->pagetable, addr, addr + n); //参考growproc()函数的方法
} else {
    return -1;
}

```

3. 在 `fork()` 中正确处理父到子内存拷贝。

`fork()` 是通过 `uvmcopy()` 来进行父进程页表即用户空间向子进程拷贝的。而对于 `uvmcopy()` 的处理和 `uvmunmap()` 是一致的，只需要将 PTE 不存在和无效的两种情况由引发 `panic` 改为 `continue` 跳过即可

```

for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0) {
        continue; // lab5-3
    }
    // panic("uvmcopy: pte should exist"); // lab5-3
}
if((*pte & PTE_V) == 0) {
    // lab5-3
    continue;
}
// panic("uvmcopy: page not present");
}

```



```

for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0) {
        continue; // lab5-3
    }
    // panic("uvmunmap: walk"); // lab5-3
}

```

4. 处理 page fault 的虚拟地址超过 p->sz 或低于用户栈的情况

va < p->trapframe->sp, va < PGROUNDDOWN(p->trapframe->sp) 或 va < PGROUNDUP(p->trapframe->sp) 三种条件都能通过测试, 猜测原因是这里的 Lazy allocation 只处理用户堆, 用户栈在 exec 系统调用时已经进行了分配

```

//lab5-3
if(va >= p->sz) //不合法的虚拟地址
{
    printf("usertrap(): invalid va=%p higher than p->sz=%p\n", va, p->sz);
    p->killed = 1;
    goto end;
}
if(va < PGROUNDUP(p->trapframe->sp)) { // lab5-3
    printf("usertrap(): invalid va=%p below the user stack sp=%p\n", va, p->trapframe->sp);
    p->killed = 1;
    goto end;
}

```

5. 处理 read()/write() 使用未分配物理内存的情况。通过这两个函数最终会调用 copyin() 和 copyout() 进行用户空间到内核空间的读写。而这两个函数对虚拟地址 va 的处理最终是通过 walkaddr() 函数得到物理地址而完成的, 这里的处理方式和 usertrap 处理的方式是一致的

```

pte = walk(pagetable, va, 0);
// lazy allocation - lab5-3
if(pte == 0 || (*pte & PTE_V) == 0) {
    // va is on the user heap
    if(va >= PGROUNDUP(p->trapframe->sp) && va < p->sz){
        char *pa;
        if ((pa = kalloc()) == 0) {
            return 0;
        }
        memset(pa, 0, PGSIZE);
        if (mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE,
            (uint64) pa, PTE_W | PTE_R | PTE_U) != 0) {
            kfree(pa);
            return 0;
        }
    }
    else {

```

测试

```

usertrap(): invalid va=0x000000003a004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003b004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003c004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003d004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003e004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003f004000 higher than p->sz=0x0000000000003000
test lazy unmap: OK
running test out of memory
usertrap(): invalid va=0xffffffff80003808 higher than p->sz=0x0000000081003810
test out of memory: OK
ALL TESTS PASSED

```

make grade 也是成功的

```

== Test   usertests: iref ==
   usertests: iref: OK
== Test   usertests: forktest ==
   usertests: forktest: OK
== Test time ==
time: OK
Score: 119/119

```

6.3 实验中遇到的问题和解决方法

1. 执行 echo hi 引发 freewalk: leaf 的 panic.

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ echo hi
panic: freewalk: leaf

```

解答: 是因为 usertrap() 中处理 page fault 时未对 va 使用 PGROUNDDOWN() 向下取整.

2. 在进行 lazytests 实验是可以通过的, 但是在进行 usertests 用户态实验测试时, 报 kerneltrap 的错误, 刚开始不太理解什么是处理 read()/write() 使用未分配物理内存的情况。于是找到 kernel/sysfile.c 找到 sys_write() 函数, 它返回的值是 filewrite() 函数, 于是在 file.c 中找到它

```
uint64
sys_write(void)
{
    struct file *f;
    int n;
    uint64 p;

    if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argad
        return -1;

    return filewrite(f, p, n);
}
```

它又调用了 pipewrite 函数，于是在 pipe.c 中找到 pipewrite 函数，可以发现，pipewrite 函数是调用 copyin

```
if(f->type == FD_PIPE){
    ret = pipewrite(f->pipe, addr, n);
} else if(f->type == FD_DEVICE){
    if(f->major < 0 || f->major >= NDEV || !devsw[f->major]
        return -1;
    ret = devsw[f->major].write(1, addr, n);
} else if(f->type == FD_INODE){
```

它接收的是四个参数，一个是页表，一个是物理地址，一个是原地址，还有一个原地址的长度，意思就是把 srcva 拷贝到 dst 上，它传过来的过程是一个虚拟的地址，在这个函数中我们可以看到 waikaddr 似乎是根据虚拟地址的值找到一个物理地址的操作，但是懒分配很有可能物理地址是 0，所以实际上这个函数对于懒分配是失败的，相同的失败同样会发生在 copyout 上，解决办法就是修改 waikaddr 函数，使它的值不为 0，得到一个正确的值，具体的修改方式和 usertrap 的修改方式是一样的

```
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
    uint64 n, va0, pa0;

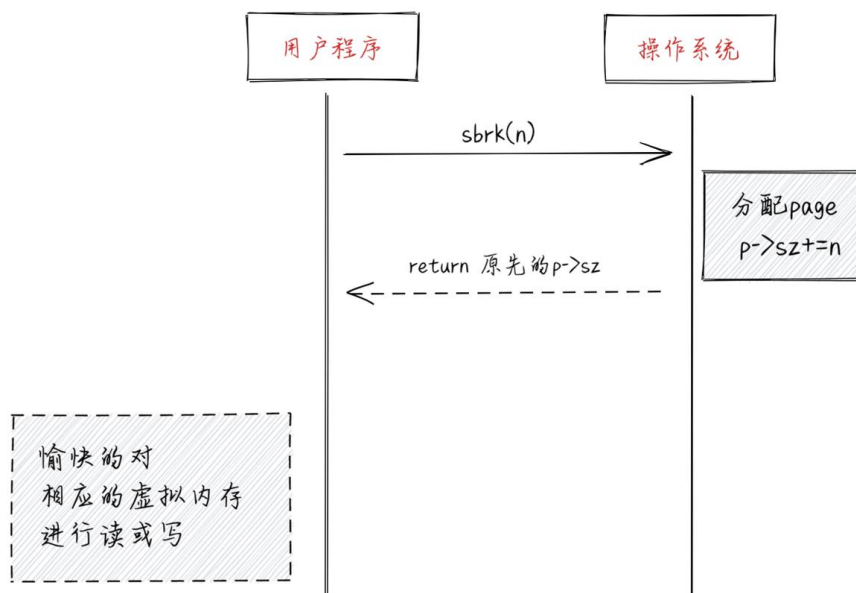
    while(len > 0){
        va0 = PGROUNDDOWN(srcva);
        pa0 = waikaddr(pagetable, va0);
        if(pa0 == 0)
            return -1;
        n = PGSIZE - (srcva - va0);
        if(n > len)
            n = len;
```

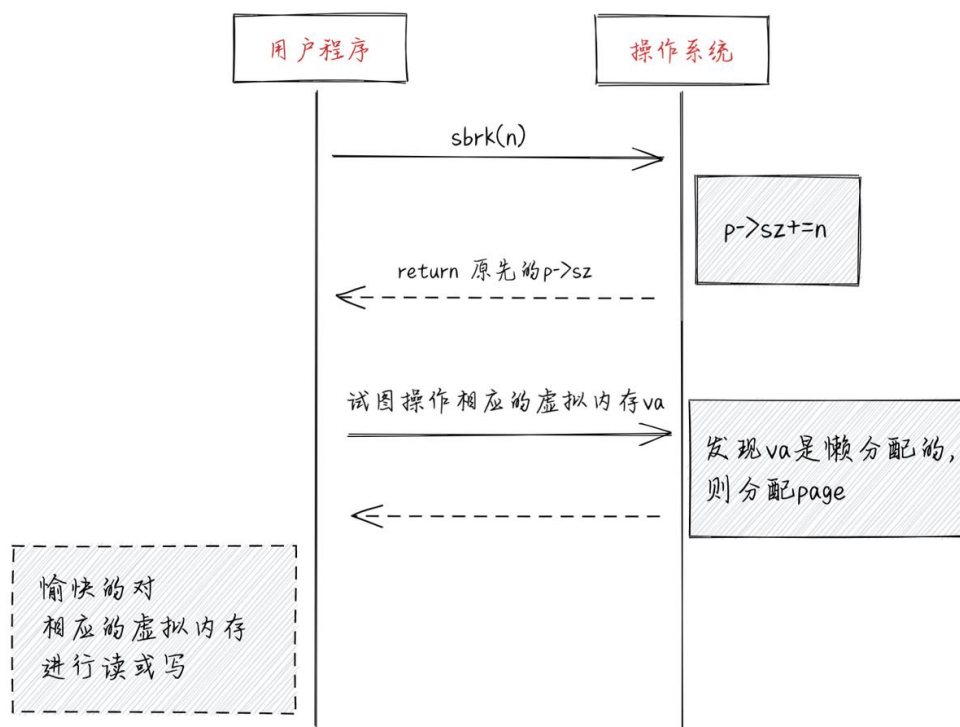
```
usertrap(): kalloc() failed
usertrap(): kalloc() failed
usertrap(): kalloc() failed
usertrap(): kalloc() failed
OK
test copyin: scause 0x000000000000000d
sepc=0x00000000800010a6 stval=0x0000000000000000
panic: kerneltrap
```

解决：通过查阅资料可知，对于函数的原本逻辑，PTE 无效或者不存在以及无 PTE_U 标志位都会返回 0 表示失败。但是在 Lazy allocation 的情况下，PTE 无效和不存在是可以被允许的，因此要对这两种情况进行处理。但并不是两种情况的所有情形都是允许的，由于 Lazy allocation 是针对用户堆空间的，因此需要判断虚拟地址 va 是否在用户堆空间的范围。

6.4 实验心得

首先我认为在做这个实验之前要知道什么是懒分配和正常分配以及他们的区别，可以把这两个分配认为是一个酒店的接待服务，有的酒店是在顾客走了之后立即打扫卫生以保持干净，而有的酒店是等顾客走，有了下一个顾客之后再打扫，第二种就是类似于懒分配的方式。





实验中遇到的问题还是蛮多的，比如不太清楚题目到底是想让我去修改哪一个函数，修改哪个地方，最后都是通过一个个的找函数调用，理解函数当中使用的方式，也可以借鉴现有的函数进行模仿，比如实验二可以模仿 `growproc()` 等。

7. lab6: Copy-on-Write Fork for xv6

7.1 实验目的

xv6 中的 `fork()` 系统调用将父进程的所有用户空间内存复制到子进程中。如果父进程较大，则复制可能需要很长时间。更糟糕的是，这项工作经常造成大量浪费；例如，子进程中的 `fork()` 后跟 `exec()` 将导致子进程丢弃复制的内存，而其中的大部分可能都从未使用过。另一方面，如果父子进程都使用一个页面，并且其中一个或两个对该页面有写操作，则确实需要复制。

copy-on-write (COW) `fork()` 的目标是推迟到子进程实际需要物理内存拷贝时再进行分配和复制物理内存页面。

COW `fork()` 只为子进程创建一个页表，用户内存的 PTE 指向父进程的物理页。COW `fork()` 将父进程和子进程中的所有用户 PTE 标记为不可写。当任一进程试图写入其中一个 COW 页时，CPU 将强制产生页面错误。内核页面错误处理程序检测到这种情况将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关 PTE 指向新的页面，将 PTE 标记为可写。当页面错误处理程序返回时，用户进程将能够写入其页面副本。

COW fork()将使得释放用户内存的物理页面变得更加棘手。给定的物理页可能会被多个进程的页表引用，并且只有在最后一个引用消失时才应该被释放。

7.2 实验步骤

按照实验指导书给出的方案：

1.修改 uvmcopy()将父进程的物理页映射到子进程，而不是分配新页。在子进程和父进程的 PTE 中清除 PTE_W 标志。

```
for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
        panic("uvmcopy: pte should exist");
    if((*pte & PTE_W) != 0)
        panic("uvmcopy: pte_t present");
    pa = PTE2PA(*pte);
    // clear PTE_W and add COW flags - lab6
    flags = (PTE_FLAGS(*pte) & (~PTE_W)) | PTE_COW;
    *pte = PA2PTE(pa) | flags; // update old pte - lab6
    // not allocate new page - lab6
    // if((mem = kalloc()) == 0)
    //     goto err;
    // memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, pa, flags) != 0){ // use the same pa as the old - lab6
        kfree(mem); // lab6
        goto err;
    }
}
```

2.修改 usertrap()以识别页面错误。当 COW 页面出现页面错误时，使用 kalloc()分配一个新页面，并将旧页面复制到新页面，然后将新页面添加到 PTE 中并设置 PTE_W。

```
syscall();
} else if(r_scause() == 15) { // COW - lab6
    if (walkcowaddr(p->pagetable, r_stval()) == 0) {
        goto bad;
    }
} else if(which_dev = devintr()) != 0){
    // ok
} else {
bad: // lab6
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("      sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
}
```

写 walkcowaddr()函数，当前面对 va 和 pte 的判断保留，然后添加对 PTE_W 标志位的判断，若无该标记，则进一步判断是否有 PTE_COW 标志位。因为无论是引发 page fault 还是 copyout(), 都是在写操作时才会考虑进行 COW 操作，读操作可以正常进行，而写操作时当前页面不可读，若无 PTE_COW 标记位则该物理页本身就不可写，直接返回 0 表示失败；反之有 PTE_COW 标记位则表明需要进行 COW 操作，接着分配新的物理页并重新映射的用户页表中，并返回新的物理地址。需要注意新的物理页的 PTE_COW 标志位需要移除，而 PTE_W 标志位需要添加，正好与 uvmcopy() 复制时是相反的。

1. // lab6

2. uint64 walkcowaddr(pagetable_t pagetable, uint64 va) {


```
3.  pte_t *pte;
4.  uint64 pa;
5.  char* mem;
6.  uint flags;
7.
8.  if (va >= MAXVA)
9.      return 0;
10.
11. pte = walk(pagetable, va, 0);
12. if (pte == 0)
13.     return 0;
14. if ((*pte & PTE_V) == 0)
15.     return 0;
16. if ((*pte & PTE_U) == 0)
17.     return 0;
18. pa = PTE2PA(*pte);
19. // 判断写标志位是否没有
20. if ((*pte & PTE_W) == 0) {
21.     // pte without COW flag cannot allocate page
22.     if ((*pte & PTE_COW) == 0) {
23.         return 0;
24.     }
25.     // 分配新物理页
26.     if ((mem = kalloc()) == 0) {
27.         return 0;
28.     }
29.     // 拷贝页表内容
30.     memmove(mem, (void*)pa, PGSIZE);
31.     // 更新标志位
32.     flags = (PTE_FLAGS(*pte) & (~PTE_COW)) | PTE_W;
33.     // 取消原映射
34.     uvmunmap(pagetable, PGROUNDDOWN(va), 1, 1);
35.     // 更新新映射
36.     if (mappages(pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem, flags) !=
        0) {
37.         kfree(mem);
38.         return 0;
39.     }
40.     return (uint64)mem; // COW 情况下返回新物理地址
41. }
42. return pa;
43. }
```

3. 确保每个物理页在最后一个 PTE 对它的引用撤销时被释放——而不是在此之前。这样做的一个好方法是为每个物理页保留引用该页面的用户页表数的“引用计数”。当 `kalloc()` 分配页时，将页的引用计数设置为 1。当 `fork` 导致子进程共享页面时，增加页的引用计数；每当任何进程从其页表中删除页面时，减少页的引用计数。`kfree()` 只应在引用计数为零时将页面放回空闲列表。可以将这些计数保存在一个固定大小的整型数组中。你必须制定一个如何索引数组以及如何选择数组大小的方案。例如，您可以用页的物理地址除以 4096 对数组进行索引，并为数组提供等同于 `kalloc.c` 中 `kinit()` 在空闲列表中放置的所有页面的最高物理地址的元素数。

- 在 `kernel/defs.h` 中声明加 1 减 1 的函数

```
// cow.c - lab6
void      increfcnt(uint64 pa);
uint8     decrefcnt(uint64 pa);
```

- 相应的 `makefile` 里面要添加相应的链接
- 在 `riscv.h` 中添加 `cow` 的标志位

```
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_COW (1L << 8) // COW flags - lab6
```

- 根据指导书，首先考虑的就是 `kernel/kalloc.c` 的 `kalloc()` 函数。在调用该函数时，则表明需要将一个物理页分配给一个进程，并对应一虚拟页。因此，需要调用 `increfcnt()` 函数对引用计数加 1，即从原本的 0 加至 1。

```
r = kmem.freelist;
if(r)
    kmem.freelist = r->next;
release(&kmem.lock);

increfcnt((uint64)r);

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

- 分配页表是加 1，那么释放页表就是减 1，`kalloc()` 函数对应的 `kernel/kalloc.c` 中的 `kfree()` 函数，用于物理页的释放

```
// and not place the page back if its reference count
// - lab6
if (decrefcnt((uint64) pa)) {
    return;
}
```

- 需要同样进行修改的还有 `kernel/kalloc.c` 中的 `freerange()` 函数。该函数被 `kinit()` 函数调用，其主要作用就是对物理内存空间中未使用的部分以物理页划分调用 `kfree()` 将其添

- 加至 kmem.freelist 中

```
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE) {
        increfcnt((uint64)p); // lab6
        kfree(p);
    }
}
```

- 4.修改 copyout()在遇到 COW 页面时使用与页面错误相同的方案。
将原本的 walkaddr() 更改为 walkcowaddr() 即可。

```
while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkcowaddr(pagetable, va0); // with COW - lab6
    if(pa0 == 0)
        return -1;
}
```

测试:

```

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

```

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
```

```

== 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
```

7.3 实验中遇到的问题及解决方法

1. make qemu 出现卡住的问题，问题在于不知道在分配页表时哪个函数进行修改

```
qemu-system-riscv64 -machine virt -bios none -kernel kern
d=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-b

xv6 kernel is booting
```

解决方法：这里的问题在于对于 cows 数组中的 ref_cnt 字段初始值为 0，在初始调用 freerange() 的 free() 函数时会将引用计数减 1，由于其类型为 uint8，会产生下溢变为 255，从而不能将物理页回收至 kmem.freelist 中，引发错误。因此，需要在调用 free() 之前再调用 increfcnt() 来先将引用计数变为 1，这样在 free() 时正好可以减至 0 进行回收。

```
char *p;
p = (char*)PGROUNDUP((uint64)pa_start);
for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE) {
    increfcnt((uint64)p); // lab6
    kfree(p);
}
```

所以需要在上面进行加 1 的操作。

2. 没有声明标志位

```
kernel/vm.c: In function 'walkcowaddr':
kernel/vm.c:134:17: error: 'PTE_COW' undeclared (first use in this function); did you mean 'PTE_W'?
134 |     if ((*pte & PTE_COW) == 0) {
    |                 ^~~~~~
    |                 PTE_W
kernel/vm.c:134:17: note: each undeclared identifier is reported only once for each function it appears in
```

解决方法：声明标志位（在 riscv.h 文件中）

```
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_COW (1L << 8) // COW flags - lab6
```

3. 报错：kalloc

```
xv6 kernel is booting

panic: kalloc
QEMU: Terminated
```

这个问题和问题 1 是同类型的，解决了问题 1，发现没有报错

4. 报错: remap，原因是在 walkcowaddr 函数当中没有取消原映射就对其进行映射进行更新

```
u=x0 -device virtio-blk-device  
  
xv6 kernel is booting  
  
hart 2 starting  
hart 1 starting  
coint: starting sh  
$ cowtests  
panic: remap
```

解决方法：该问题即出现了虚拟页重映射，原因在于 `walkcowaddr()` 中未调用 `uvmunmap()` 先将移除原映射，所以在函数当中调用 `uvmunmap()` 取消原映射

7.4 实验心得

本实验虽然不是很难，但是花费了我比较长的时间，主要原因是在进行标志位的建立以及映射修改的问题上出现了一些问题，主要是我没有特别的懂如何去更新标志位的方式，看了很多博主的文章和参考了一些方法，以及在进行测试的时候出现了很多的问题，也是一步一步修改测试出来的。

8. Lab7: Multithreading

8.1 实验目的

8.1.1 Uthread: switching between threads ([moderate](#))

您的工作是提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划。完成后，`make grade` 应该表明您的解决方案通过了 `uthread` 测试。

8.1.2 Using threads ([moderate](#))

在本作业中，您将探索使用哈希表的线程和锁的并行编程。您应该在具有多个内核的真实 Linux 或 MacOS 计算机（不是 xv6，不是 qemu）上执行此任务。最新的笔记本电脑都有多核处理器。

8.1.3 Barrier([moderate](#))

在本作业中，您将实现一个屏障 (Barrier)：应用程序中的一个点，所有参与的线程在

此点上必须等待，直到所有其他参与线程也达到该点。您将使用 `pthread` 条件变量，这是一种序列协调技术，类似于 `xv6` 的 `sleep` 和 `wakeup`

8.2 实验步骤

8.2.1 Uthread: switching between threads ([moderate](#))

您需要将代码添加到 `user/uthread.c` 中的 `thread_create()` 和 `thread_schedule()`，以及 `user/uthread_switch.S` 中的 `thread_switch`。一个目标是确保当 `thread_schedule()` 第一次运行给定线程时，该线程在自己的栈上执行传递给 `thread_create()` 的函数。另一个目标是确保 `thread_switch` 保存被切换线程的寄存器，恢复切换到线程的寄存器，并返回到后一个线程指令中最后停止的点。您必须决定保存/恢复寄存器的位置；修改 `struct thread` 以保存寄存器是一个很好的计划。您需要在 `thread_schedule` 中添加对 `thread_switch` 的调用；您可以将需要的任何参数传递给 `thread_switch`，但目的是将线程从 `t` 切换到 `next_thread`

1. 设置和 `kernel/proc.h` 中定义的 `struct context` 结构体一样的结构体，此时用户多线程切换需要保存的寄存器信息和 `xv6` 内核线程切换需要保存的寄存器信息

```
1. struct ctx {
2.     uint64 ra;
3.     uint64 sp;
4.     // callee-saved
5.     uint64 s0;
6.     uint64 s1;
7.     uint64 s2;
8.     uint64 s3;
9.     uint64 s4;
10.    uint64 s5;
11.    uint64 s6;
12.    uint64 s7;
13.    uint64 s8;
14.    uint64 s9;
15.    uint64 s10;
16.    uint64 s11;
17. };
18. struct thread {
19.     char    stack[STACK_SIZE]; /* the thread's stack */
20.     int     state;              /* FREE, RUNNING, RUNNABLE */
21.     struct ctx context;        // thread's context - lab7-1
22. };
```

2. 添加代码到 `thread_create()` 函数。

传递的 `thread_create()` 参数 `func` 需要记录，这样在线程运行时才能运行该函数，此外线程的栈结构是独立的，在运行函数时要在线程自己的栈上，因此也要初始化线程的栈指

针。而在线程进行调度切换时，同样需要保存和恢复寄存器状态，而上述二者实际上分别对应着 ra 和 sp 寄存器

```
// YOUR CODE HERE
// set thread's function address and thread's stack pointer - lab7-1
t->context.ra = (uint64) func;
t->context.sp = (uint64) t->stack + STACK_SIZE;
```

3. 添加代码到 thread_schedule() 函数

thread_schedule() 函数负责进行用户多线程间的调度。此处是通过函数的主动调用进行的线程切换。其主要工作就是从当前线程在线程数组的位置开始寻找一个 RUNNABLE 状态的线程进行运行。实际上与 kernel/proc.c 中的 scheduler() 函数是很相似的。而很明显在找到线程后就需要进行线程的切换，调用函数 thread_switch()。

```
if (current_thread != next_thread) {          /* switch threads? */
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    thread_switch(&t->context, &current_thread->context); // switch thread - lab7-1
} else
    next_thread = 0;
```

thread_switch() 根据其在 user/thread.c 中的外部声明以及指导书的要求可以推断出，该函数应该是定义在 user/uthread_switch.S，用汇编代码实现。因此其功能应该与 kernel/swtch.S 中的 swtch() 函数一致，进行线程切换时的寄存器代码的保存与恢复。

```
1. # same as swtch in swtch.S - lab7
2.     sd ra, 0(a0)
3.     sd sp, 8(a0)
4.     sd s0, 16(a0)
5.     sd s1, 24(a0)
6.     sd s2, 32(a0)
7.     sd s3, 40(a0)
8.     sd s4, 48(a0)
9.     sd s5, 56(a0)
10.    sd s6, 64(a0)
11.    sd s7, 72(a0)
12.    sd s8, 80(a0)
13.    sd s9, 88(a0)
14.    sd s10, 96(a0)
15.    sd s11, 104(a0)
16.
17.    ld ra, 0(a1)
18.    ld sp, 8(a1)
```

```
19. ld s0, 16(a1)
20.     ld s1, 24(a1)
21.     ld s2, 32(a1)
22.     ld s3, 40(a1)
23.     ld s4, 48(a1)
24.     ld s5, 56(a1)
25.     ld s6, 64(a1)
26.     ld s7, 72(a1)
27.     ld s8, 80(a1)
28.     ld s9, 88(a1)
29.     ld s10, 96(a1)
30.     ld s11, 104(a1)
31.     ret    /* return to ra */
```

测试

```
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
```

单项测试通过

```
farzena@farzena-virtual-machine:~/work/xv6
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (5.4s)
```

8.2.2 Using threads ([moderate](#))

如下图按照实验指导书

```
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ make ph
gcc -o ph -g -O2 notxv6/ph.c -pthread
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./ph 1
100000 puts, 15.838 seconds, 6314 puts/second
0: 0 keys missing
100000 gets, 14.878 seconds, 6721 gets/second
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./ph 2
100000 puts, 5.272 seconds, 18967 puts/second
0: 14020 keys missing
1: 14020 keys missing
200000 gets, 15.653 seconds, 12777 gets/second
```

```
pthread_mutex_t lock;           // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock);       // acquire lock
pthread_mutex_unlock(&lock);     // release lock
```

根据实验指导书给出的要求，声明互斥锁，初始化互斥锁，并进行加锁和解锁

```
int nthread = 1;
pthread_mutex_t locks[NBUCKET]; // lab7-2
```

```
} else {
    pthread_mutex_lock(&locks[i]); // lock - lab7-2
    // the new is new.
    insert(key, value, &table[i], table[i]);
    pthread_mutex_unlock(&locks[i]); // unlock - lab7-2
```

```
// initialize locks - lab7-2
for(int i = 0; i < NBUCKET; ++i) {
    pthread_mutex_init(&locks[i], NULL);
}
```

测试

```
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./ph 2
100000 puts, 5.134 seconds, 19477 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 11.491 seconds, 17405 gets/second
```

./grade-lab-thread ph_fast 单项测试:

```
farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-thread ph_fast
make: 'kernel/kernel' is up to date.
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (64.1s)
```

8.2.3 Barrier([moderate](#))

此处主要涉及互斥锁和条件变量配合达到线程同步。

首先条件变量的操作需要在互斥锁锁定的临界区内。

然后进行条件判断，此处即判断是否所有的线程都进入了 `barrier()` 函数，若不满足则使用 `pthread_cond_wait()` 将当前线程休眠，等待唤醒；若全部线程都已进入 `barrier()` 函数，则最后进入的线程会调用 `pthread_cond_broadcast()` 唤醒其他由条件变量休眠的线程继续运行。

需要注意的是，对于 `pthread_cond_wait()` 涉及三个操作：原子的释放拥有的锁并阻塞当前线程，这两个操作是原子的；第三个操作是由条件变量唤醒后会再次获取锁。

```

// then increment bstate.threads
//
// lab7-3
pthread_mutex_lock(&bstate.barrier_mutex);
// judge whether all threads reach the barrier
if(++bstate.nthread != nthread) { // not all threads reach
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex); // wait other threads
} else { // all threads reach
    bstate.nthread = 0; // reset nthread
    ++bstate.round; // increase round
    pthread_cond_broadcast(&bstate.barrier_cond); // wake up all sleeping threads
}
pthread_mutex_unlock(&bstate.barrier_mutex);
}

```

测试

```

● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./barrier 2
OK; passed
● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./barrier 3
OK; passed
● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./barrier 10
OK; passed

```

单项测试

```

● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-thread barrier
make: 'kernel/kernel' is up to date.
== Test barrier == make: 'barrier' is up to date.
barrier: OK (12.2s)

```

8.3 实验中遇到的问题及解决方法

1. 第三个实验报错：原因是没有确保在上一轮仍在使用的 `bstate.nthread` 时，离开 barrier 并循环运行的线程不会增加 `bstate.nthread`，同时 barrier 的轮数没有加 1

```

gcc -o barrier -g -O2 notxv6/barrier.c -pthread
● farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./barrier 2
barrier: notxv6/barrier.c:55: thread: Assertion `i == t' failed.
Aborted (core dumped)

```

改错

```

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

```

8.4 实验心得

刚开始不是很理解 using thread 当中为什么会有键的丢失，主要是不知道哈希表的结构，哈希表就是“数组(bucket)+链表”的经典实现方法，通过取余确定 bucket, put() 是使用前插法

插入键值对, `get()` 遍历 `bucket` 下的链表找到对应 `key` 的 `entry`。假设有 A 和 B 两个线程同时 `put()`, 由于该哈希表的桶数 `NBUCKET` 为 5, 哈希函数为 `key%NBUCKET`, 而插入的 `key` 为 `keys[b*n+i]`, 而 `b=NKEYS/nthread=100000/2=50000`, 而 `b%NBUCKET==0`, 因此对于 A 和 B 两个线程, 在 `i` 相同时实际上会在同一个 `bucket` 插入数据。假设 A 和 B 都运行到 `put()` 函数的 `insert()` 处, 还未进入该函数内部, 这就会导致两个线程 `insert()` 的后两个参数是相同的, 都是当前 `bucket` 的链表头, 如若线程 A 调用 `insert()` 插入完 `entry` 后, 切换到线程 B 再调用 `insert()` 插入 `entry`, 则会导致线程 A 刚刚插入的 `entry` 丢失。这个实验感觉代码上没有太多的要做的地方, 主要是要理解它的原理, 通过实验指导书肯定是可以做成功的, 但是不理解原理, 可能就不知道这个实验让我们干什么。

9. lab8:locks

9.1 实验目的

9.1.1 Memory allocator ([moderate](#))

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

9.1.2 Buffer cache ([hard](#))

修改块缓存, 以便在运行 `bcachetest` 时, `bcache` (buffer cache 的缩写) 中所有锁的 `acquire` 循环迭代次数接近于零。理想情况下, 块缓存中涉及的所有锁的计数总和应为零, 但只要总和小于 500 就可以。修改 `bget` 和 `brelse`, 以便 `bcache` 中不同块的并发查找和释放不太可能在锁上发生冲突 (例如, 不必全部等待 `bcache.lock`)。你必须保护每个块最多缓存一个副本的不变量。完成后, 您的输出应该与下面显示的类似 (尽管不完全相同)。确保 `usertests` 仍然通过。完成后, `make grade` 应该通过所有测试。

9.2 实验步骤

9.2.1 Memory allocator ([moderate](#))

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

```
struct {
    struct spinlock lock;
    struct run *freelist;
    char lockname[8];    // save lock's name - lab8-1
} kmems[NCPU]; // a free list and a lock per CPU - lab8-1
```

2.修改初始化函数 kinit()。

kinit() 函数中主要会初始化 kmem 的锁并调用 freearrange() 初始化分配物理页。

由于此处 kmems 是一个数组, 因此这里需要将原本对 kmem 中锁的初始化替换为一个初始化 kmems 数组中锁的循环。

这里利用了 snprintf() 函数来设置每个锁的名称, 将名称存储到 lockname 字段. 之所以这样做, 是因为在 initlock() 函数中, 锁名称的记录是指针的浅拷贝 lk->name=name, 因此对于每个锁的名称需要使用全局的内存进行记录而非函数的局部变量, 以防止内存丢失. 此外, 为了配合 kalloc test 的输出, 需要保证每个锁的名称以"kmem"开头。

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

3.修改 kfree() 函数。

kfree() 函数用于回收物理页到 freelist. 指导书要求初始时 freearrange() 将空闲内存分配给当前运行 CPU 的 freelist. 而 freearrange() 内部就是调用 kfree() 进行的内存回收. 同样的, 这里我们也将每次调用 kfree() 释放的物理页由当前运行的 CPU 的 freelist 进行回收。

```
1. void
2. kfree(void *pa)
3. {
4.     struct run *r;
5.     int c;    // cpuid - lab8-1
6.
```



```

7.  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
8.      panic("kfree");
9.
10. // Fill with junk to catch dangling refs.
11. memset(pa, 1, PGSIZE);
12.
13. r = (struct run*)pa;
14.
15. // get the current core number - lab8-1
16. push_off();
17. c = cpuid();
18. pop_off();
19. // free the page to the current cpu's freelist - lab8-1
20. acquire(&kmems[c].lock);
21. r->next = kmems[c].freelist;
22. kmems[c].freelist = r;
23. release(&kmems[c].lock);
24. }

```

4.修改 kalloc() 函数.

与 kfree() 函数回收物理页相对的, 就是 kalloc() 函数进行物理页的分配.

此处先不考虑偷取其他 CPU 的空闲物理页. 同样的, 需要调用 cpuid() 获取当前 CPU 核心的序号, 使用 kmems 中对应的锁和 freelist 进行物理页的分配。

```

{
    struct run *r;
    // lab8-1
    int c;
    push_off();
    c = cpuid();
    pop_off();
    // get the page from the current cpu's freelist
    acquire(&kmems[c].lock);
    r = kmems[c].freelist;
    if(r)
        kmems[c].freelist = r->next;
    release(&kmems[c].lock);
    // steal page - lab8-1
    if(!r && (r = steal(c))) {
        acquire(&kmems[c].lock);
        kmems[c].freelist = r->next;
        release(&kmems[c].lock);
    }
}

```

6. 编写偷取物理页函数 steal().

7. 最后考虑偷取物理页的情况。当前 CPU 的空闲物理页链表 freelist 为空，但此时其他 CPU 可能仍有空闲物理页，因此需要当前 CPU 去偷取其他 CPU 的部分物理页。

```
1. struct run *steal(int cpu_id) {
2.     int i;
3.     int c = cpu_id;
4.     struct run *fast, *slow, *head;
5.     // 若传递的 cpuid 和实际运行的 cpuid 出现不一致,则引发 panic
6.     // 加入该判断以检查在 kalloc()调用 steal 时 CPU 不会被切换
7.     if(cpu_id != cpuid()) {
8.         panic("steal");
9.     }
10.    // 遍历其他 NCPU-1 个 CPU 的空闲物理页链表
11.    for (i = 1; i < NCPU; ++i) {
12.        if (++c == NCPU) {
13.            c = 0;
14.        }
15.        acquire(&kmems[c].lock);
16.        if (kmems[c].freelist) {
17.            slow = head = kmems[c].freelist;
18.            fast = slow->next;
19.            while (fast) {
20.                fast = fast->next;
21.                if (fast) {
22.                    slow = slow->next;
23.                    fast = fast->next;
24.                }
25.            }
26.            kmems[c].freelist = slow->next;
27.            release(&kmems[c].lock);
28.            slow->next = 0;
29.            return head;
30.        }
31.        release(&kmems[c].lock);
32.    }
33.    return 0;
34. }
```

在 xv6 中执行 kalloctest, 输出如下

```

lock: proc: #fetch-and-add 147759 #acquire() 112905
lock: proc: #fetch-and-add 93246 #acquire() 112905
lock: proc: #fetch-and-add 40419 #acquire() 112705
lock: proc: #fetch-and-add 31284 #acquire() 112689
lock: proc: #fetch-and-add 25524 #acquire() 112690
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK

```

在 xv6 中执行 usertests sbrkmuch 进行测试:

```

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

```

在 xv6 中执行 usertests 测试:

```

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

```

./grade-lab-lock kallocetest 单项测试:

```

● == Test kallocetest: sbrkmuch == ^Cfarzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-lock ka
make: 'kernel/kernel' is up to date.
== Test running kallocetest == (293.5s)
== Test kallocetest: test1 ==
kallocetest: test1: OK
== Test kallocetest: test2 ==
kallocetest: test2: OK
== Test kallocetest: sbrkmuch == kallocetest: sbrkmuch: OK (38.0s)

```

9.2.2 Buffer cache (hard)

1. 修改 buf 和 bcache 结构体

修改 kernel/buf.h 中的 buf 结构体。

由于此处不再使用双向链表而采用哈希表, 对于哈希表 bucket 中的链式结构, 此处笔者使用的是单向链表(当然双向链表同样可以满足), 所以不再需要 prev 字段。

```
uint blockno;
struct sleeplock lock;
uint refcnt;
// struct buf *prev; // LRU cache list - lab8-2
struct buf *next; // hash list
uchar data[BSIZE];
uint timestamp; // the buf last using time - lab8-2
};
```

修改 kernel/bio.c 中的 bcache 结构体。

```
int size; // record the count of used buf - lab8-2
struct buf buckets[NBUCKET]; // lab8-2
struct spinlock locks[NBUCKET]; // buckets' locks - lab8-2
struct spinlock hashlock; // the hash table's lock - lab8-2
// Linked list of all buffers, through prev/next.
```

2. 修改非主要函数

修改 kernel/bio.c 中的 binit() 函数。

该函数主要用于缓存块和相关锁的初始化。

由于不再使用双向链表, 因此相关的代码即可注释掉。

```
1. void
2. binit(void)
3. {
4.     int i;
5.     struct buf *b;
6.
7.     bcache.size = 0; // lab8-2
8.     initlock(&bcache.lock, "bcache");
9.     initlock(&bcache.hashlock, "bcache_hash"); // init hash lock - lab8-2
10.    // init all buckets' locks - lab8-2
11.    for(i = 0; i < NBUCKET; ++i) {
12.        initlock(&bcache.locks[i], "bcache_bucket");
13.    }
14.
15.    // lab8-2
16.    // // Create linked list of buffers
17.    // bcache.head.prev = &bcache.head;
18.    // bcache.head.next = &bcache.head;
19.    for(b = bcache.buf; b < bcache.buf+NBUF; b++){
20.    // lab8-2
```

```
21. //    b->next = bcache.head.next;
22. //    b->prev = &bcache.head;
23.    initsleeplock(&b->lock, "buffer");
24. //    bcache.head.next->prev = b;
25. //    bcache.head.next = b;
26. }
27. }
```

修改 kernel/bio.c 中的 `brelse()` 函数。

该函数用于释放缓存块。在原本的实现中，若其引用计数为 0，则将其移至双向链表表头，这样双向链表表头是最近使用的，表尾是最近未使用的，构成一个 LRU 序列，方便 `bget()` 函数寻找缓存块

```
1. void
2. brelse(struct buf *b)
3. {
4.     int idx;
5.     if(!holdingsleep(&b->lock))
6.         panic("brelse");
7.
8.     releasesleep(&b->lock);
9.
10.    // change the lock - lab8-2
11.    idx = HASH(b->blockno);
12.    acquire(&bcache.locks[idx]);
13.    b->refcnt--;
14.    if (b->refcnt == 0) {
15.        // no one is waiting for it.
16.        // lab8-2
17.        b->next->prev = b->prev;
18.        b->prev->next = b->next;
19.        b->next = bcache.head.next;
20.        b->prev = &bcache.head;
21.        bcache.head.next->prev = b;
22.        bcache.head.next = b;
23.        b->timestamp = ticks;
24.    }
25.
26.    release(&bcache.locks[idx]);
27. }
```

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

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

```
void
bunpin(struct buf *b) {
    // change the lock - lab8-2
    int idx = HASH(b->blockno);
    acquire(&bcache.locks[idx]);
    b->refcnt--;
    release(&bcache.locks[idx]);
}
```

```
void
bpin(struct buf *b) {
    // change the lock - lab8-2
    int idx = HASH(b->blockno);
    acquire(&bcache.locks[idx]);
    b->refcnt++;
    release(&bcache.locks[idx]);
}
```

3. 修改 bget() 函数

```
1. static struct buf*
2. bget(uint dev, uint blockno)
3. {
4.     struct buf *b;
5.     // lab8-2
6.     int idx = HASH(blockno);
7.     struct buf *pre, *minb = 0, *minpre;
8.     uint mintimestamp;
9.     int i;
10.
11.     // loop up the buf in the buckets[idx]
12.     acquire(&bcache.locks[idx]); // lab8-2
13.     for(b = bcache.buckets[idx].next; b; b = b->next){
14.         if(b->dev == dev && b->blockno == blockno){
15.             b->refcnt++;
16.             release(&bcache.locks[idx]); // lab8-2
17.             acquiresleep(&b->lock);
18.             return b;
19.         }
20.     }
21.
22.     // Not cached.
23.     // check if there is a buf not used -lab8-2
24.     acquire(&bcache.lock);
```



```
25. if(bcache.size < NBUF) {
26.     b = &bcache.buf[bcache.size++];
27.     release(&bcache.lock);
28.     b->dev = dev;
29.     b->blockno = blockno;
30.     b->valid = 0;
31.     b->refcnt = 1;
32.     b->next = bcache.buckets[idx].next;
33.     bcache.buckets[idx].next = b;
34.     release(&bcache.locks[idx]);
35.     acquiresleep(&b->lock);
36.     return b;
37. }
38. release(&bcache.lock);
39. release(&bcache.locks[idx]);
40.
41. // select the last-recently used block into the bucket
42. //based on the timestamp - lab8-2
43. acquire(&bcache.hashlock);
44. for(i = 0; i < NBUCKET; ++i) {
45.     mintimestamp = -1;
46.     acquire(&bcache.locks[idx]);
47.     for(pre = &bcache.buckets[idx], b = pre->next; b; pre = b, b = b->next)
48.     {
49.         // research the block
50.         if(idx == HASH(blockno) && b->dev == dev && b->blockno == blockno)
51.         {
52.             b->refcnt++;
53.             release(&bcache.locks[idx]);
54.             release(&bcache.hashlock);
55.             acquiresleep(&b->lock);
56.             return b;
57.         }
58.         if(b->refcnt == 0 && b->timestamp < mintimestamp) {
59.             minb = b;
60.             minpre = pre;
61.             mintimestamp = b->timestamp;
62.         }
63.     }
64.     // find an unused block
65.     if(minb) {
66.         minb->dev = dev;
67.         minb->blockno = blockno;
68.         minb->valid = 0;
```

```
67.         minb->refcnt = 1;
68.         // if block in another bucket, we should move it to correct bucket

69.         if(idx != HASH(blockno)) {
70.             minpre->next = minb->next;    // remove block
71.             release(&bcache.locks[idx]);
72.             idx = HASH(blockno); // the correct bucket index
73.             acquire(&bcache.locks[idx]);
74.             minb->next = bcache.buckets[idx].next;    // move block to cor
rect bucket
75.             bcache.buckets[idx].next = minb;
76.         }
77.         release(&bcache.locks[idx]);
78.         release(&bcache.hashlock);
79.         acquiresleep(&minb->lock);
80.         return minb;
81.     }
82.     release(&bcache.locks[idx]);
83.     if(++idx == NBUCKET) {
84.         idx = 0;
85.     }
86. }
87. // lab8-2
88. // // Recycle the least recently used (LRU) unused buffer.
89. // for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
90. //     if(b->refcnt == 0) {
91. //         b->dev = dev;
92. //         b->blockno = blockno;
93. //         b->valid = 0;
94. //         b->refcnt = 1;
95. //         release(&bcache.lock);
96. //         acquiresleep(&b->lock);
97. //         return b;
98. //     }
99. // }
100. panic("bget: no buffers");
101. }
```

测试 bcachetest

可以看到 bcache 相关锁的争用情况大幅下降, acquire() 整体次数大幅减少

```
ock: bcache: #fetch-and-add 0 #acquire() 112
ock: bcache_hash: #fetch-and-add 0 #acquire() 82
ock: bcache_bucket: #fetch-and-add 0 #acquire() 4894
ock: bcache_bucket: #fetch-and-add 0 #acquire() 2882
ock: bcache_bucket: #fetch-and-add 0 #acquire() 4977
ock: bcache_bucket: #fetch-and-add 0 #acquire() 4320
ock: bcache_bucket: #fetch-and-add 0 #acquire() 6335
ock: bcache_bucket: #fetch-and-add 0 #acquire() 6327
ock: bcache_bucket: #fetch-and-add 0 #acquire() 6683
ock: bcache_bucket: #fetch-and-add 0 #acquire() 6403
ock: bcache_bucket: #fetch-and-add 0 #acquire() 9506
ock: bcache_bucket: #fetch-and-add 0 #acquire() 7143
ock: bcache_bucket: #fetch-and-add 0 #acquire() 5340
ock: bcache_bucket: #fetch-and-add 0 #acquire() 4890
ock: bcache_bucket: #fetch-and-add 0 #acquire() 2882
-- top 5 contended locks:
ock: proc: #fetch-and-add 802636 #acquire() 104705
ock: proc: #fetch-and-add 663854 #acquire() 104705
ock: proc: #fetch-and-add 499717 #acquire() 104704
ock: proc: #fetch-and-add 452488 #acquire() 104704
ock: proc: #fetch-and-add 440314 #acquire() 104706
```

usertests 测试

```
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
```

make grade 测试通过

```
$ make qemu-gdb
(299.9s)
== Test  kallocetest: test1 ==
kallocetest: test1: OK
== Test  kallocetest: test2 ==
kallocetest: test2: OK
== Test kallocetest: sbrkmuch ==
$ make qemu-gdb
kallocetest: sbrkmuch: OK (34.1s)
== Test running bcachetest ==
$ make qemu-gdb
(63.3s)
== Test  bcachetest: test0 ==
bcachetest: test0: OK
```

9.3 实验中遇到的问题及解决方法

1. `bget` 中重新分配可能要持有两个锁，如果桶 `a` 持有自己的锁，再申请桶 `b` 的锁，与此同时如果桶 `b` 持有自己的锁，再申请桶 `a` 的锁就会造成死锁！因此代码中使用了 `if(!holding(&bcache.bucket[i].lock))` 来进行检查。此外，代码优先从自己的桶中获取缓冲区，如果自身没有依次向后查找这样的方式也尽可能地避免了前面的情况。

```
$ usertests
usertests starting
test manywrites:
```

2. 在 `bget` 中搜索缓冲区并在找不到缓冲区时为该缓冲区分配条目必须是原子的！在提示中说 `bget` 如果未找到而进行分配的操作可以是串行化的，也就是说多个 `CPU` 中未找到，应当串行的执行分配，同时还应当避免死锁。

3. `usertests` 中测试报错：panic: freeing free block

当在目标 `bucket` 找到可重用的缓存块时错误地在 `bucket` 中移除了缓存块，而此处不应该移除，否则会导致缓存块的丢失。

```
$ bcachetest
start test0
panic: freeing free block
QEMU: Terminated
```

9.4 实验心得

本次实验在实验 2 和实验 3 上面花费的时间是最多的，主要是按照实验指导书走没有太多的方法，于是就一头扎进去写，发现如果不懂原理是根本写不出代码的，要么就是代码报错，要么就是卡住，所以还是原理比较重要。

10. Lab9: file system

10.1 实验目的

10.1.1 Large files(moderate)

在本作业中,您将增加 xv6 文件的最大大小。目前,xv6 文件限制为 268 个块或 $268 \times \text{BSIZE}$ 字节(在 xv6 中 BSIZE 为 1024)。此限制来自以下事实: 一个 xv6 inode 包含 12 个“直接”块号和一个“间接”块号,“一级间接”块指一个最多可容纳 256 个块号的块, 总共 $12+256=268$ 个块。

更改 xv6 文件系统代码, 以支持每个 inode 中可包含 256 个一级间接块地址的“二级间接”块, 每个一级间接块最多可以包含 256 个数据块地址。结果将是一个文件将能够包含多达 65803 个块, 或 $256 \times 256 + 256 + 11$ 个块(11 而不是 12, 因为我们将为二级间接块牺牲一个直接块号)。

10.1.2 Symbolic links(moderate)

您将实现 `symlink(char *target, char *path)` 系统调用, 该调用在引用由 target 命名的文件的路径处创建一个新的符号链接。有关更多信息, 请参阅 `symlink` 手册页(注: 执行 `man symlink`)。要进行测试, 请将 `symlinktest` 添加到 Makefile 并运行它。当测试产生以下输出(包括 `usertests` 运行成功)时, 您就完成本作业了。

10.2 实验步骤

10.2.1 Large files(moderate)

1.修改 `kernel/fs.h` 中的直接块号的宏定义 `NDIRECT` 为 11.

根据实验要求, inode 中原本 12 个直接块号被修改为了 11 个.

```
#define NDIRECT 11 // lab9-1
```

2.修改 inode 相关结构体的块号数组. 具体包括 `kernel/fs.h` 中的磁盘 inode 结构体 `struct dinode` 的 `addrs` 字段; 和 `kernel/file.h` 中的内存 inode 结构体 `struct inode` 的 `addrs` 字段. 将二者数组大小设置为 `NDIRECT+2`, 因为实际 inode 的块号总数没有改变, 但 `NDIRECT` 减少了 1

```
uint addrs[NDIRECT+2]; // Data block addresses // lab9-1
```

```
uint addrs[NDIRECT+2]; // lab9-1
```

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

```
#define NDOUBLYINDIRECT (NINDIRECT * NINDIRECT)
```

4. 修改 kernel/fs.c 中的 bmap() 函数.

该函数用于返回 inode 的相对块号对应的磁盘中的块号.

```
bn -= NINDIRECT;
if(bn < NDOUBLYINDIRECT) {
    // get the address of doubly-indirect block
    if((addr = ip->addrs[NINDIRECT + 1]) == 0) {
        ip->addrs[NINDIRECT + 1] = addr = balloc(ip->dev);
    }
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    // get the address of singly-indirect block
    if((addr = a[bn / NINDIRECT]) == 0) {
        a[bn / NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    bn %= NINDIRECT;
    // get the address of direct block
    if((addr = a[bn]) == 0) {
        a[bn] = addr = balloc(ip->dev);
        log_write(bp);
    }
}
```

6. 修改 kernel/fs.c 中的 itrunc() 函数, 该函数用于释放 inode 的数据块

```
int i, j, k; // lab9-1
struct buf *bp, *bp2; // lab9-1
uint *a, *a2; // lab9-1
```



```

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

```

7. 修改 kernel/fs.h 中的文件最大大小的宏定义 MAXFILE。

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

测试

```

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

```

./grade-lab-fs bigfile 单项测试:

```

farzena@farzena-virtual-machine:~/work/xv6-labs-2020$ ./grade-lab-fs bigfile
make: 'kernel/kernel' is up to date.
== Test running bigfile == running bigfile: OK (180.6s)
(Old xv6.out.bigfile failure log removed)

```

10.2.2 Symbolic links (moderate)

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

```

#define SYS_close 21
#define SYS_symlink 22 // lab9-2

```

```

extern uint64 sys_uptime(void);
extern uint64 sys_symlink(void); // lab9-2

```

```
[SYS_close] sys_close,  
[SYS_symlink] sys_symlink, // lab9-2
```

```
entry("uptime");  
entry("symlink"); # lab9-2
```

```
int uptime(void);  
int symlink(char *target, char *path); // lab9-2
```

2. 向 kernel/stat.h 添加新的文件类型 (T_SYMLINK) 以表示符号链接

```
#define T_DEVICE 5 // Device  
#define T_SYMLINK 4 // Soft symbolic link - lab9-2
```

3. 在 kernel/fcntl.h 中添加一个新标志 (O_NOFOLLOW)，该标志可用于 open 系统调用

```
#define O_RDONLY 0x0001 // Read only  
#define O_NOFOLLOW 0x0004 // lab9-2
```

4. 在 kernel/sysfile.c 中实现 sys_symlink() 函数.

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

```

25.   end_op();
26.   return 0;
27. }

```

5. 修改 kernel/sysfile 的 sys_open() 函数。

该函数使用来打开文件的，对于符号链接一般情况下需要打开的是其链接的目标文件，因此需要对符号链接文件进行额外处理

```

if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
    if((ip = follow_symlink(ip)) == 0) {
        end_op();
        return -1;
    }
}

```

对于链接的深度，此处 kernel/fs.h 中定义了 NSYMLINK 用于表示最大的符号链接深度，超过该深度将不会继续跟踪而是返回错误

```

#define NSYMLINK 10

```

6. 最后在 Makefile 中添加对测试文件 symlinktest.c 的编译。

```

$U/_usertests\
$U/_grind\
$U/_wc\
$U/_zombie\
$U/_symlinktest\

```

测试：

```

$ symlinktest
Start: test symlinks
open_symlink: path "/testsymlink/a" is not exist
open_symlink: links form a cycle
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok

```

```

test tput: 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

```

usertests 测试

10.3 遇到的问题及解决方法

在 xv6 中执行 bigfile 通过, 但执行 usertests 出现 virtio_disk_intr status 的 panic

```
$ usertests
panic: virtio_disk_intr status
QEMU: Terminated
```

原因: 未修改 NDIRECT 为 11, 而是保持 12, 将 NDIRECT 的值改为 11 以后, 不在出现 panic 在 make qemu 的过程中, 出现还未出现输入指令界面时, kernel 卡住

解决方式: 有可能是代码的问题, 代码如果语法上没有错误, 但是逻辑上出现错误就会导致循环问题上面的错误卡住, 所以需要反复检查代码逻辑上的问题。

11. Lab10: mmap

11.1 实验目的

mmap 和 munmap 系统调用允许 UNIX 程序对其地址空间进行详细控制。它们可用于在进程之间共享内存, 将文件映射到进程地址空间, 并作为用户级页面错误方案的一部分, 如本课程中讨论的垃圾收集算法。在本实验室中, 您将把 mmap 和 munmap 添加到 xv6 中, 重点关注内存映射文件 (memory-mapped files)。

11.2 实验步骤

1. 首先, 向 UPROGS 添加 _mmaptest, 以及 mmap 和 munmap 系统调用, 以便让 user/mmaptest.c 进行编译。现在, 只需从 mmap 和 munmap 返回错误。我们在 kernel/fcntl.h 中为您定义了 PROT_READ 等。运行 mmaptest, 它将在第一次 mmap 调用时失败

```
$U/_grep\
$U/_init\
$U/_kill\
$U/_ln\
$U/_ls\
$U/_mkdir\
$U/_rm\
$U/_sh\
$U/_stressfs\
$U/_usertests\
$U/_grind\
$U/_wc\
$U/_zombie\
$U/_mmaptest
```

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

```
#define SYS_close 21
#define SYS_mmap 22 // lab10
#define SYS_munmap 23 // lab10

extern uint64 sys_uptime(void);
extern uint64 sys_mmap(void); // lab10
extern uint64 sys_munmap(void); // lab10

[SYS_close] sys_close,
[SYS_mmap] sys_mmap, // lab10
[SYS_munmap] sys_munmap, // lab10
];

entry("uptime");
entry("mmap"); # lab10
entry("munmap"); # lab10

void *mmap(void *addr, int length, int prot, int flags,
           int fd, int offset); // lab10
int munmap(void *add, int length); // lab10
```

2. 定义 vm_area 结构体及其数组

在 kernel/proc.h 中定义 struct vm_area 结构体

处用于表示使用 mmap 系统调用文件映射的虚拟内存的区域的位置、大小、权限等(实际上在 Linux 中会用来表示整个虚拟内存区域的相关信息, 包括堆、栈等, 而在此部分实验中出于简单考虑只用于表示文件映射部分内存)。struct vm_area 中具体记录的内存信息和 mmap 系统调用的参数基本对应, 包括映射的起始地址、映射内存长度(大小)、权限、mmap 标志位、文件偏移以及指向的文件结构体指针。

```
struct vm_area {
    uint64 addr; // 起始地址
    int len; // 映射内存长度(大小)
    int prot; // 权限
    int flags; // mmap 标志位
    int offset; // the file offset
    struct file* f; // pointer to the mapped file
};
```

3. 由于 xv6 内核中没有内存分配器, 因此可以声明一个固定大小的 VMA 数组, 并根据需要从该数组进行分配。大小为 16 应该就足够了

```
#define NVMA 16
```


4.编写 mmap 系统调用

在 kernel/sysfile.c 中实现系统调用 sys_mmap()

```
1. uint64 sys_mmap(void) {
2.     uint64 addr;
3.     int len, prot, flags, offset;
4.     struct file *f;
5.     struct vm_area *vma = 0;
6.     struct proc *p = myproc();
7.     int i;
8.
9.     if (argaddr(0, &addr) < 0 || argint(1, &len) < 0
10.        || argint(2, &prot) < 0 || argint(3, &flags) < 0
11.        || argfd(4, 0, &f) < 0 || argint(5, &offset) < 0) {
12.         return -1;
13.     }
14.     if (flags != MAP_SHARED && flags != MAP_PRIVATE) {
15.         return -1;
16.     }
17.     // the file must be written when flag is MAP_SHARED
18.     if (flags == MAP_SHARED && f->writable == 0 && (prot & PROT_WRITE)) {
19.         return -1;
20.     }
21.     // offset must be a multiple of the page size
22.     if (len < 0 || offset < 0 || offset % PGSIZE) {
23.         return -1;
24.     }
25.
26.     // allocate a VMA for the mapped memory
27.     for (i = 0; i < NVMA; ++i) {
28.         if (!p->vma[i].addr) {
29.             vma = &p->vma[i];
30.             break;
31.         }
32.     }
33.     if (!vma) {
34.         return -1;
35.     }
36.
37.     // assume that addr will always be 0, the kernel
```



```

38. //choose the page-aligned address at which to create
39. //the mapping
40. addr = MMAPMINADDR;
41. for (i = 0; i < NVMA; ++i) {
42.     if (p->vma[i].addr) {

43. // get the max address of the mapped memory
44.     addr = max(addr, p->vma[i].addr + p->vma[i].len);
45.     }
46. }
47. addr = PGROUNDUP(addr);
48. if (addr + len > TRAPFRAME) {
49.     return -1;
50. }
51. vma->addr = addr;
52. vma->len = len;
53. vma->prot = prot;
54. vma->flags = flags;
55. vma->offset = offset;
56. vma->f = f;
57. filedup(f); // increase the file's reference count
58.
59. return addr;
60. }

```

5. 编写 page fault 处理代码

由于在 `sys_mmap()` 中对文件映射的内存采用的是 Lazy allocation, 因此需要对访问文件映射内存产生的 page fault 进行处理. 和之前 Lazy allocation 和 COW 的实验相同, 即修改 `kernel/trap.c` 中 `usertrap()` 的代码.

首先就是添加对 page fault 情况的 trap 的检查, 由于映射的内存未分配, 而且该内存读写执行都是有可能的, 因此所有类型的 page fault 都可能发生, 因此 `r_scause()` 的值包括 12, 13 和 15 三种情况。

0	9	Environment call from S-mode
0	10-11	Reserved
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	16-23	Reserved

接下来则是根据发生 page fault 的地址去当前进程的 VMA 数组中找对应的 VMA 结构体. 然后就是进行 Lazy allocation, 使用 `kalloc()` 先分配一个物理页, 并使用 `memset()` 进行清空. 使用 `readi` 读取文件, 它接受一个偏移量参数, 在该偏移处读取文件 (但必须 lock/unlock 传

递给 readi 的索引结点)

访问权限根据 VMA 中记录的 mmap 的 prot 参数转换为 PTE 的权限标志位。对于读和执行

权限比较简单, 对于写权限此处只有本次是 Store Page fault 时才会设置。

最后即可使用 mappages() 将物理页映射到用户进程的页面值。

```

1.  else if (r_scause() == 12 || r_scause() == 13
2.          || r_scause() == 15) { // mmap page fault - lab10
3.      char *pa;
4.      uint64 va = PGROUNDDOWN(r_stval());
5.      struct vm_area *vma = 0;
6.      int flags = PTE_U;
7.      int i;
8.      // find the VMA
9.      for (i = 0; i < NVMA; ++i) {
10.         // like the Linux mmap, it can modify the remaining bytes in
11.         //the end of mapped page
12.         if (p->vma[i].addr && va >= p->vma[i].addr
13.             && va < p->vma[i].addr + p->vma[i].len) {
14.             vma = &p->vma[i];
15.             break;
16.         }
17.     }
18.     if (!vma) {
19.         goto err;
20.     }
21.     // set write flag and dirty flag to the mapped page's PTE
22.     if (r_scause() == 15 && (vma->prot & PROT_WRITE)
23.         && walkaddr(p->pagetable, va)) {
24.         if (uvmsetdirtywrite(p->pagetable, va)) {
25.             goto err;
26.         }
27.     } else {
28.         if ((pa = kalloc()) == 0) {
29.             goto err;
30.         }
31.         memset(pa, 0, PGSIZE);
32.         ilock(vma->f->ip);
33.         if (readi(vma->f->ip, 0, (uint64) pa, va - vma->addr + vma->offset, PGS
34.             IZE) < 0) {
35.             iunlock(vma->f->ip);
36.             goto err;
37.         }
38.         iunlock(vma->f->ip);

```

```

38.     if ((vma->prot & PROT_READ)) {
39.         flags |= PTE_R;
40.     }
41.     // only store page fault and the mapped page can be written
42.     //set the PTE write flag and dirty flag otherwise don't set
43.     //these two flag until next store page fault
44.     if (r_scause() == 15 && (vma->prot & PROT_WRITE)) {
45.         flags |= PTE_W | PTE_D;
46.     }
47.     if ((vma->prot & PROT_EXEC)) {
48.         flags |= PTE_X;
49.     }
50.     if (mappages(p->pagetable, va, PGSIZE, (uint64) pa, flags) != 0) {
51.         kfree(pa);
52.         goto err;
53.     }
54. }
55. }

```

6.编写 munmap 系统调用

在 kernel/sysfile.c 中实现系统调用 sys_munmap()。根据实验要求，该系统调用即将映射的部分内存进行取消映射，同时若为 MAP_SHARED，则需要将对文件映射内存的修改会写到文件中。

- 首先也是对参数的提取，munmap 只有 addr 和 length 两个参数。然后对参数进行简单的检查。len 需要非负；此外根据 Linux 手册，addr 需要是 PGSIZE 的整数倍。
- 接下来同 usertrap() 中类似，根据 addr 和 length 找到对应的 VMA 结构体。未找到则返回失败。
- 然后主要就是判断当前取消映射的部分是否有 MAP_SHARED 标志位，有的话则需要将该部分写回文件。当然在此之前先判断 len 是否为 0，若是的话则直接返回成功，无需后续的工作。
- 该部分为该系统调用的最为复杂的部分，需要考虑的有两点：一是哪些页面需要写入，另一方面是每次写回文件的写入大小。对于前者，根据实验指导，选择使用脏页标志位 (PTE_D) 进行记录，拥有该标志位则表明该部分被修改过，则需要将该页面写回文件中，具体脏页标志位的设置见后续。对于后者，由于是根据脏页会写文件的，因此能够想到写入大小为 PGSIZE，但由于 len 可能不为 PGSIZE 的整倍数，此处还需要进行判断。此外，根据实验指导，参考 filewrite() 函数，一次写入文件的大小还受日志 block 的影响，因此可能会再分批写入文件(实际上 PGSIZE 大于 maxsz，整页需要写回文件时会被分为两次)。
- 在将修改内容写回文件后，便可以使用 uvmunmap() 将改部分页面在用户页表中取消映射。
- 这里采用的是向上页面取整的取消页表映射方法，实际上感觉有些不合理，因为可能有一部分页面仍未取消文件映射，但考虑到 addr 需要页面对齐，因此不会在页面中间取消文件映射，否则后半部分将无法取消文件映射。
- 此外，此处修改了 uvmunmap() 函数的 PTE_V 标志位的检查部分，和之前 Lazy

allocation 的实验相同, 取消映射的页面可能并未实际分配, 此时跳过即可。

1. // lab10

```
2. uint64 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.    // find the VMA
18.    for (i = 0; i < NVMA; ++i) {
19.        if (p->vma[i].addr && addr >= p->vma[i].addr
20.            && addr + len <= p->vma[i].addr + p->vma[i].len) {
21.            vma = &p->vma[i];
22.            break;
23.        }
24.    }
25.    if (!vma) {
26.        return -1;
27.    }
28.
29.    if (len == 0) {
30.        return 0;
31.    }
32.
33.    if ((vma->flags & MAP_SHARED)) {
34.        // the max size once can write to the disk
35.        maxsz = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
36.        for (va = addr; va < addr + len; va += PGSIZE) {
37.            if (uvmgetdirty(p->pagetable, va) == 0) {
38.                continue;
39.            }
40.            // only write the dirty page back to the mapped file
41.            n = min(PGSIZE, addr + len - va);
```

```

42.     for (i = 0; i < n; i += n1) {
43.         n1 = min(maxsz, n - i);

44.     begin_op();
45.         ilock(vma->f->ip);
46.         if (writei(vma->f->ip, 1, va + i, va - vma->addr + vma->offset + i,
47.             n1) != n1) {
48.             iunlock(vma->f->ip);
49.             end_op();
50.             return -1;
51.         }
52.         iunlock(vma->f->ip);
53.         end_op();
54.     }
55. }
56. uvmunmap(p->pagetable, addr, (len - 1) / PGSIZE + 1, 1);
57. // update the vma
58. if (addr == vma->addr && len == vma->len) {
59.     vma->addr = 0;
60.     vma->len = 0;
61.     vma->offset = 0;
62.     vma->flags = 0;
63.     vma->prot = 0;
64.     fclose(vma->f);
65.     vma->f = 0;
66. } else if (addr == vma->addr) {
67.     vma->addr += len;
68.     vma->offset += len;
69.     vma->len -= len;
70. } else if (addr + len == vma->addr + vma->len) {
71.     vma->len -= len;
72. } else {
73.     panic("unexpected munmap");
74. }
75. return 0;
76. }

```

7. 脏页标志位设置

首先是在 kernel/riscv.h 中定义了脏页标志位 PTE_D

```
#define PTE_D (1L << 7) // dirty flag - lab10
```

接下来再 kernel/vm.c 中定义了 uvmgetdirty() 以及 uvmsetdirtywrite() 两个函数。前者用于读取脏页标志位，后者用于写入脏页标志位和写标志位

```
// get the dirty flag of the va's PTE - lab10
int uvmgetdirty(pagetable_t pagetable, uint64 va) {
    pte_t *pte = walk(pagetable, va, 0);
    if(pte == 0) {
        return 0;
    }
    return (*pte & PTE_D);
}
```

```
int uvmsetdirtywrite(pagetable_t pagetable, uint64 va) {
    pte_t *pte = walk(pagetable, va, 0);
    if(pte == 0) {
        return -1;
    }
    *pte |= PTE_D | PTE_W;
    return 0;
}
```

8.修改 exit 和 fork 系统调用

修改 kernel/proc.c 中的 exit() 函数,添加的代码与 munmap() 中部分基本系统, 区别在于需要遍历 VMA 数组对所有文件映射内存进行取消映射, 而且是整个部分取消

```
// unmap the mapped memory - lab10
for (i = 0; i < NVMA; ++i) {
    if (p->vma[i].addr == 0) {
        continue;
    }
    vma = &p->vma[i];
    if ((vma->flags & MAP_SHARED)) {
        for (va = vma->addr; va < vma->addr + vma->len; va += PGSIZE) {
            if (uvmgetdirty(p->pagetable, va) == 0) {
                continue;
            }
            n = min(PGSIZE, vma->addr + vma->len - va);
            for (r = 0; r < n; r += n1) {
                n1 = min(maxsz, n - i);
                begin_op();
                ilock(vma->f->ip);
                if (writei(vma->f->ip, 1, va + i, va - vma->addr + vma->offset + i, n1) != n1) {
                    iunlock(vma->f->ip);
                    end_op();
                    panic("exit: writei failed");
                }
                iunlock(vma->f->ip);
                end_op();
            }
        }
    }
}
```

修改 kernel/proc.c 中的 fork() 函数.

在使用 fork() 创建子进程时, 需要将父进程的 VMA 结构体进行拷贝, 从而获得相同的文件映射内存


```
for (i = 0; i < NVMA; ++i) {  
    if (p->vma[i].addr) {  
        np->vma[i] = p->vma[i];  
        filedup(np->vma[i].f);  
    }  
}
```

实行 mmaptest 测试

```
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
```

usertests 通过

```
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
```

make grade 测试: 通过

```
== 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 (238.5s)
(Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 140/140
```

11.3 实验中遇到的问题及解决方法

1.在写 page fault 代码的过程当中，pagefault 的数据使用错误导致报错，但是侥幸通过测试，第二次没通过

```
} else if (r_scause() == 12 || r_scause() == 13
|| r_scause() == 15) { // mmap page fault - lab10
char *pa;
```

	0	9	Environment call from S-mode
	0	10-11	Reserved
	0	12	Instruction page fault
	0	13	Load page fault
	0	14	Reserved
	0	15	Store/AMO page fault
	0	16-23	Reserved

如上图所示，pagefault 有三个值，12，13，15

2.mmaptest 测试失败，

```
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
mmaptest: fork_test failed: mmap (4), pid=3
```

找到报错的地方

```
char *p1 = mmap(0, PGSIZE*2, PROT_READ, MAP_SHARED, fd, 0);
if (p1 == MAP_FAILED)
    err("mmap (4)");
```

解决方法，添加对映射权限 `prot & PROT_WRITE` 的检查

3.脏页标志位的设计：

对于 `MAP_SHARED` 以及可写的文件，并不将所有内容都回写，而是只将修改的部分回写；另一方面，由于文件映射内存是 `Lazy allocation`，脏页因为进行了修改因此一定分配了内存，因而在取消映射时无需考虑当前页面是否真正被分配。

11.4 实验心得

实验做到这里，感觉自己还是非常有收获的，下面总结一下这个假期我学习 `xv6` 的一些感想，首先就是一个实验可以通过实验指导书的提示去做完，但是实验的原理不一定可以弄清楚，实验的原理可以通过每次实验的过程中查找资料以及推荐的实验书来进行学习，同时如果实在是看不懂可以去看哈佛的 *MIT* 视频，其次就是在实验的过程当中，实验指导书所指的每一个步骤都要非常仔细的去浏览和弄懂它所指的意思，我在做实验的过程当中每次遇到问题首先会把这个方法所使用的代码先拷贝一份，不要去动源码，不然会导致代码改不成原有的代码也找不到的问题，其次就是在写函数的过程当中每写一个实现的方式，可以 `make qemu` 一次，这样最后可能错误会少一点，但是也不会避免逻辑上错误的问题，还有一个技巧就是可以多去借鉴内核源码，有些可以直接用，有些是可以在原有的基础上进行修改的，最后就是一些心得和感想：做不出来一定不要放弃，多去找原因才能成功，学习如此，生活亦是如此。