同济大学软件学院

操作系统课程设计

6.S081/Fall 2021 实验报告

1953366 王思力

2022-8-12

目录

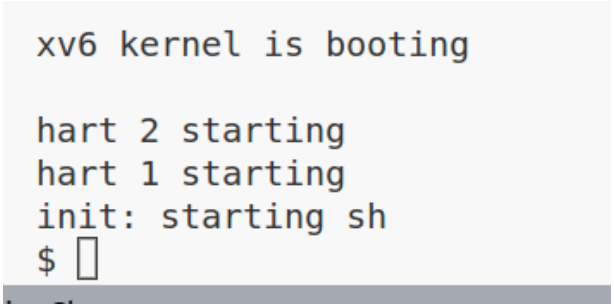
整体说明

实验环境：VMware, Ubuntu 系统虚拟机, VScode
源代码：已上传到 github 上，网址：TanglinQAQ/OS_6.S081 (github.com)
实验版本：为 6.S081/Fall 2021，网址：6.S081 / Fall 2021 (mit.edu)
其余说明：部分 Lab 中有需要在 answers-Labname.txt 文件中进行作答的理论题，作答的结果直接写在了实验报告中，未在 Lab 要求的 txt 中作答。

Lab 1：Xv6 and Unix utilities

1. Boot xv6

1.1 实验目的
尝试运行 xv6 系统。
1.2 实验内容
在环境配置好后，终端输入 make qemu，可见输出如下图。



2. sleep

2.1 实验目的
尝试在 xv6 系统编写程序，熟悉 xv6 系统，完成 user/sleep.c 程序。
2.2 实验内容
2.2.1 实验步骤
（1）把 sleep.c 加入到 makefile 文件中

```
$U/_zombie\
$U/_sleep\
```

（2）编写 sleep.c

```c
#include "kernel/types.h"
#include "user/user.h"
int main(int argc, char* argv[])
{
  if (argc != 2)
    exit(0);
  printf("test sleep\n");
  sleep(atoi(argv[1]));
  exit(0);
}
```

2.2.2 实验结果

```
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (1.4s)
== Test sleep, returns == sleep, returns: OK (1.2s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.7s)
tanglin@ubuntu:~/workspace/OS design/xv6-labs-2021$
```

3. pingpong

3.1 实验目的

编写 pingpong.c，使用 fork()创造父子进程，父进程通过 pipe 向子进程发"ping"，子进程收到后打印"ping"并向父进程发送"pong"，父进程收到后打印"pong"。

3.2 实验内容

3.2.1 实验步骤

（1）编写 pingpong.c

```c
#define WRITE 1
int main(int argc, char* argv[]) {
  int p[2], c[2];
  pipe(p);
  pipe(c);

  if (fork() == 0) {//子进程的fork()==0
    char c_s[10];
    close(c[READ]);
    close(p[WRITE]);
    read(p[READ], c_s, 4);
    printf("%d: received %s\n", getpid(), c_s);
    write(c[WRITE], "pong", 4);
    close(c[WRITE]);
  }
  else {//父进程fork()! =0
    char p_s[10];
    close(p[READ]);
    close(c[WRITE]);
    write(p[WRITE], "ping", 4);
    close(p[WRITE]);
    read(c[READ], p_s, 4);
    printf("%d: received %s\n", getpid(),p_s);
  }
  exit(0);
}
```

3.2.2 实验结果

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

3.3 心得体会

fork()的作用是生成一个子进程，子进程 fork()的返回值==0，父进程 fork()的返回值不为零，这样就能区分开两个进程。另外，在使用 pipe 时候写操作结束后要关闭写端，否则可能造成读端一直收不到 EOF 造成阻塞。

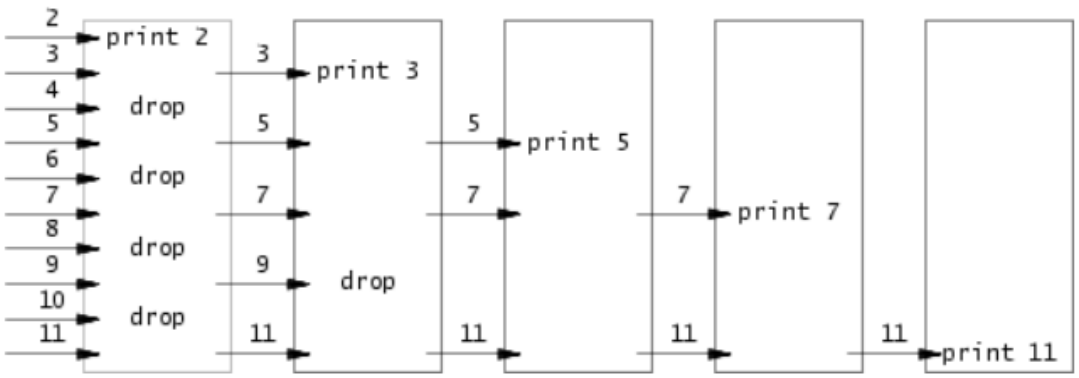
4. primes

4.1 实验目的

使用素数筛法、fork 和 pipe 实现打印 35 以内的所有素数。

4.2 实验内容

4.2.1 实验原理



上图为使用素数筛法求 2-11 的素数的示意图。已知 2 为素数，则选出 2，并把 2-11 中 2 的倍数全部筛除，此时备选数字中还剩下 3、5、7、9、11；剩下的首个数字必为素数，即 3 为选出的第二个素数，再次把剩下的数字中 3 的倍数全部筛除。以此类推，直到备选数字为 0 个。

4.2.2 实验步骤

（1）编写 primes.c

main 函数：调用 calc_prime 函数，传入存有 2-35 的数组 num 和当前备选数字计数 34。

```c
int main()
{
  int num[34];
  int i;
  for (i = 0;i < 34;i++)
    num[i] = i + 2;
  calc_prime(num, 34);
  exit(0);
}
```

（2）编写 calc_prime 函数：

当 count=0 时说明已经没有备选数字，素数筛选结束，否则 num 数组的第一个数就是素数。

fork()==0 的分支为子进程，负责把 num 数组中第二个数到最后一个数放入到管道中。

```
void calc_prime(int* num, int count)
{
  int pip[2], prime = 0;
  pipe(pip);
  if (count == 0)
    return;
  else {
    prime = num[0];
    printf("prime %d\n", prime);//打印第一个数，为素数
  }
  if (fork() == 0) {
    //子进程：管道第一个数即为素数
    close(pip[READ]);
    int i;
    for (i = 1;i < count;i++) {
      //printf("%d", *(num + i));
      write(pip[WRITE], (char*)(num + i), 4);
    }
```

fork()!=0 的分支为父进程，负责从管道中取出所有数字，若读到的数字不为之前取到的素数的倍数，就把这个数存到 num 数组中。

```
  else {
    close(pip[WRITE]);
    //父进程：取出num的倍数
    count = 0;
    int t;
    while (read(pip[READ], &t, 4)) {
      //printf("t:%d", t);
      if (t % prime != 0) {
        num[count] = t;
        count++;
      }
    }
    calc_prime(num, count);
    close(pip[READ]);
    wait(0);
  }
}
```

4.2.3 实验结果

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

4.3 遇到的问题及解决方案

与上一题（pingpong）不同，本题要对一个四字节的整数进行管道读写，而读写函数的原型是 read(int,void*,int)和 write(int,void*,int)和 write(int,void*,int)。考虑到可能变量类型会不匹配，最开始使用了 char t[4]这样的方式存入管道中读得的值，并且用*(int*)t 来将其转换成整数。后来经过尝试发现不存在上述问题，可以直接用 int t 来存，调用时使用&t 即可，无论是哪种数据类型的指针本质上就是一串地址，在传形参的时候不会受到影响。

4.4 心得体会

本题最大的收获是了解到了素数筛法，原本我求某一范围的所有素数时更倾向使用“判断所有比该数小的数是否能被该数整除的”方式，这种方法时间复杂度较高。而素数筛法蕴含减治的思想，不断减小问题的规模，是一种更优秀的算法。

5.find

5.1 实验目的

编写 user/find.c，使其实现能够在全部路径下找到具有某一特定文件名的文件。

5.2 实验内容

5.2.1 实验原理

根据实验指导书上的提示，可以参照 ls.c 来读取文件夹，ls 指令的作用是列出指定目录下的所有文件和文件夹。

ls.c 中的 fmtname 函数作用为从路径中取得目标文件、文件夹的名称并返回。

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

  // Find first character after last slash.
  for(p=path+strlen(path); p >= path && *p != '/'; p--)
    ;
  p++;
```

switch 中对于 st.type 做了不同的处理，其中 T_FILE 是文件，T_DIR 是路径。

```
switch(st.type){
case T_FILE:
  printf("%s %d %d %l\n", fmtname(path), st.type, st.ino, st.size);
  break;

case T_DIR:
  if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
    printf("ls: path too long\n");
```

本题要求在某一指定路径及其所有子文件夹下寻找指定名称的文件，代码的整体框架应该和 ls.c 是相同的，只需要修改原本的 ls.c 不会递归访问文件夹的逻辑即可。

5.2.2 实验步骤

（1）编写 find.c

直接复制粘贴 ls.c 的内容，并加以修改，修改的内容如下。

当前访问到的是文件，判断当前的文件名是否为目标文件名，若是目标文件名则打印。

```
case T_FILE:
  //是文件
  //文件名等于所求
  if (strcmp(fmtname(path), filename) == 0)
    printf("%s\n", path);
  break;
```

（2）当前访问到的是路径，除了判断路径为空，还需要加上忽略“.”和“..”的条件。

```
if(de.inum == 0|| strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
  continue;
```

（3）当前访问到的是路径，需要进入这个路径进行进一步的访问，递归调用 find 函数即可。

```
68        if(stat(buf, &st) < 0){
69          printf("find: cannot stat %s\n", buf);
70          continue;
71        }
72        find(buf, filename);//递归进入下一层文件夹
```

5.2.3 实验结果

```
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
```

5.3 实验中遇到的问题和解决方法

一开始编写程序时候没有加入忽略“.”和“..”的条件，导致产生了以下报错。

```
$ find . b
usertrap(): unexpected scause 0x000000000000000f pid=6
            sepc=0x00000000000008a2 stval=0x0000000000001e48
$QEMU: Terminated
```

经过分析后认为“.”代表当前路径，“..”代表上一级路径，若不忽略这两个路径就会不断重复地访问相同的路径，造成递归过深，最后报错。加入忽略这两个路径的条件后程序正确运行。

5.4 心得体会

文件系统中的文件都是靠路径定位，只要能够确定路径就能定位到某一具体文件。另外在操作系统中路径和文件的本质是差不多的，都可以看作是文件，区别在于 stat 结构体中的 type，若是 T_FILE 则为文件，若是 T_DIR 则为路径。

```
struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

6.xargs

6.1 实验目的

xargs 命令的作用是把一个命令的标准输出传递给另一个命令作为输入，本题需要我们实现这样的功能。

6.2 实验内容

6.2.1 实验原理

在 xv6 系统中，可以使用 read(0,void*,int)的方式读标准输出，从而获得上一条命令的输出。而要把这个输出变为下一条命令的参数并执行就需要用到 exec 系统调用，即 exec(*path,argv)，path 代表文件路径，argv 则是执行这条命令需要的参数。指要把上一条命令的输出和其它参数共同存在 argv 中再调用 exec 即可实现 xargs 的功能。

6.2.2 实验步骤

编写 xargs.c

```
//循环读取标准输入
while ((len = read(0, line, MAX)) > 0) {
  if (fork() == 0) {
    //子进程：执行命令
    int p = 0;
    for (i = 0;i < len;i++) {
      if (line[i] == ' ' || line[i] == '\n') {
        //一段命令结束
        params[count][p] = 0;
        count++;
        p = 0;
      }
      else {
        //指针！需要申请内存空间
        if (p == 0)
          params[count] = (char*)malloc(MAX);
        params[count][p] = line[i];
        p++;
      }
    }
    exec(cmd, params);
```

6.2.3 实验结果

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

```
tanglin@ubuntu:~/workspace/OS_design/xv6-labs-2021$ ./grade-lab-util
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (4.7s)
== Test sleep, returns == sleep, returns: OK (1.5s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.2s)
== Test pingpong == pingpong: OK (1.3s)
== Test primes == primes: OK (1.1s)
== Test find, in current directory == find, in current directory: OK (1.4s)
== Test find, recursive == find, recursive: OK (2.1s)
== Test xargs == xargs: OK (2.0s)
== Test time ==
time: OK
Score: 100/100
```

6.3 遇到的问题及解决方案

在初步写好代码调试时候发现打印单个 params（exec 中对应的 argv 数组）均为 null，但打印一个 params 的具体某一位的时候又能读到值。

```
$ sh < xargstest.sh
$ $ $ $ $ $ !(null)usage: grep pattern [file ...]
!(null)!(null)usage: grep pattern [file ...]
$ $ QEMU: Terminated
```

经过反复检查代码后发现 parmas 是指针数组，而指针本身没有分配内存空间，此时无论存储的字符串加没加’\0’都会变成 null，在对指针申请空间后问题解决。

```
char* params[MAXARG];//存参数
```

```
//指针！需要申请内存空间
if (p == 0)
  params[count] = (char*)malloc(MAX);
```

Lab2：System Calls

1. System call tracing

1.1 实验目的

实现 sys_trace()系统调用，使其能够追踪打印出调用的系统调用。

1.2 实验内容

1.2.1 实验步骤

（1）添加系统调用

```
############################
#修改：添加了两个需要的系统调用的入口
entry("trace");
entry("sysinfo");
```

```
//修改：增加两个新的系统调用（sysinfo需要声明结构体）
int trace(int);
```

```
//////////////////////////////
#define SYS_trace  22
```

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

```
//修改
[SYS_trace]   sys_trace,
```

（2）实现 sys_trace()

修改 proc.h 中的 proc 结构体，增加 tracemask 域，记录需要跟踪的系统调用号。

```
  char name[16];               // Process name (debug
  //修改
  int tracemask;               //记录需要跟踪的系统调用
};
```

编写 sys_trace()，使其能够改变 proc 中的 tracemask。

```
//修改
//sys_trace: 修改proc对象中的tracemask
uint64
sys_trace(void)
{
  int n;
  if(argint(0, &n) < 0)
    return -1;
  myproc()->tracemask = n;
  return 0;
}
```

（3）修改 fork()

在生成子进程时把 tracemask 也复制到子进程中。

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

（4）修改 syscall.c，实现系统调用的打印

想要实现打印系统调用的名称，首先要把各个系统调用的名称存起来。

```
//修改，添加一个数组用于存储系统调用的名称
static char* syscalls_name[25] = {
  [SYS_fork]    "fork",
  [SYS_exit]    "exit",
  [SYS_wait]    "wait",
  [SYS_pipe]    "pipe",
  [SYS_read]    "read",
  [SYS_kill]    "kill",
  [SYS_exec]    "exec",
  [SYS_fstat]   "fstat",
  [SYS_chdir]   "chdir",
  [SYS_dup]     "dup",
  [SYS_getpid]  "getpid",
  [SYS_sbrk]    "sbrk",
  [SYS_sleep]   "sleep",
  [SYS_uptime]  "uptime",
  [SYS_open]    "open",
  [SYS_write]   "write",
```

修改 syscall()，在每次被调用时打印该系统调用的名称。

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

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    if (p->tracemask & (1 << num)) {
      printf("%d: syscall %s -> %d\n", p->pid, syscalls_name[num], p->trapframe->a0);
    }
  }
  else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

1.2.2 实验结果

```
25: syscall fork -> 65
50: syscall fork -> 66
35: syscall fork -> 67
31: syscall fork -> 68
31: syscall fork -> -1
35: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED
```

1.3 心得体会

系统调用的传参方式比较特殊，是直接存在寄存器中的，核心态想要获取参数也是直接从寄存器中取得，而非我们平时熟悉的形参-实参传参。

2. Sysinfo

2.1 实验目的

实现 sysinfo()系统调用，从而读取当前系统的内存与进程信息。

2.2 实验内容

2.2.1 实验步骤

（1）在用户态添加 sysinfo 结构体以存储相关信息，并添加系统调用

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

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

```
//修改: 增加两个新的系统调用 (sysinfo需要声明结构体)
int trace(int);
struct sysinfo;
int sysinfo(struct sysinfo *);
```

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

```
[SYS_sysinfo] sys_sysinfo
```

（2）编写 count_freemem()和 count_proc()，获取内存与进程相关信息。

```
//修改
int count_freemem()
{
  struct run* r;
  r = kmem.freelist;
  int count = 0;
  while (r) {
    count++;
    r = r->next;
  }
  return count*PGSIZE;
}
```

```
//修改
int count_nproc()
{
  int count = 0, i;
  for (i = 0;i < NPROC;i++) {
    if (proc[i].state != UNUSED)
      count++;
  }
  return count;
}
```

（2）编写 sys_sysinfo()

```
//修改
int count_freemem();
int count_nproc();
uint64 sys_sysinfo(void)
{
  uint64 addr;//存储用户态变量地址
  struct sysinfo info;
  struct proc* p = myproc();
  if (argaddr(0, &addr) < 0)
    return -1;
  info.freemem = count_freemem();
  info.nproc = count_nproc();
  if (copyout(p->pagetable, addr, (char*)&info, sizeof(info)) < 0)//复制
    return -1;
  return 0;
}
```

2.2.2 实验结果

```
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$
```

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (9.7s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.3s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (0.9s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (25.9s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (5.1s)
== Test time ==
time: OK
Score: 35/35
```

2.3 心得体会

用户态和核心态之间不能直接传递数据，在核心态向用户态传数据需要使用copyout()函数。

Lab3：Page Tables

1. Speed up system calls

1.1 实验目的

在核心态与用户态之间建立一个只读页，把内核的信息写在这个只读页中，方便用户态的快速读取，达到加速系统调用 getpid()的效果。

1.2 实验内容

1.2.1 实验步骤

（1）在 proc 结构体中增加 usyspage 指针，指向分配的页表。

```
  char name[16];               // Process name (debug
  struct usyscall* usyspage;   //修改：指向分配的页表
};
```

（2）修改 proc.c 中的 proc_pagetable()，完成对 USYSCALL 的映射。

```
  }
  //修改：对USYSCALL映射
  if(mappages(pagetable, USYSCALL, PGSIZE,
    (uint64)(p->usyspage), PTE_R | PTE_U) < 0) {
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);//撤销前面的映射
    uvmunmap(pagetable, TRAPFRAME, 1, 0);//撤销前面的映射
    uvmfree(pagetable, 0);
    return 0;
  }
  return pagetable;
```

（3）修改 allocproc()、freeproc()、proc_freepagetable()，使进程在初始化和回收时做出相应的改变。

```
//修改：为usyscall申请一页
if((p->usyspage = (struct usyscall *)kalloc()) == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
p->usyspage->pid = p->pid;
```

```
  p->trapframe = 0;
  //修改
  if (p->usyspage)
    kfree((void*)p->usyspage);
  p->usyspage = 0;
```

```
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  //修改：释放申请的页表
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmfree(pagetable, sz);
}
```

1.2.2 实验结果

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

1.3 心得体会

在 Lab2 中我了解到核心态想向用户态传值只能通过使用 copyout()的方式，用户态切换核心态也要经过把各值保存到寄存器里的过程，这样的速度比较慢。本题让我了解到还可以建立一个只读页表，使用户态不必通过麻烦的系统调用就能读到核心态传出的值。

2. Print a page table

2.1 实验目的

实现 vmprint()函数，使其能够打印页表。

2.2 实验内容

2.2.1 实验步骤

根据实验指导书中的提示，freewalk()是递归释放页表的函数，参考它即可实现递归遍历页表并打印。

```
//修改：vmprint
void recurse_print(pagetable_t pagetable, int level)
{
  for (int i = 0; i < 512; i++) {
    pte_t pte = pagetable[i];
    if (pte & PTE_V) {
      for (int j = 0; j <= level; j++) {
        if (j == 0)
          printf("..");
        else
          printf(" ..");
      }
      uint64 child = PTE2PA(pte);
      printf("%d: pte %p pa %p\n", i, pte, child);
      if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0)
        recurse_print((pagetable_t)child, level + 1);
    }
  }
}
```

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

}
```

2.2.2 实验结果

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

3. Detecting which pages have been accessed

3.1 实验目的

实现系统调用 pgaccess()，使其能够搜索到所有被访问过的页面。

3.2 实验内容

3.2.1 实验步骤

（1）修改 riscv.h，添加 PTE_A 标志位，代表该 PTE 被访问过

```
#define PTE_U (1L << 4) //
//修改
#define PTE_A (1L << 6)
```

（2）修改 sys_pgaccess()

```
int i;
for (i = 0;i < len;i++) {
  if (va > MAXVA)
    return -1;
  pte_t* pte = walk(p->pagetable, va, 0);
  if (pte == 0)
    return -1;
  if (*pte & PTE_A) {
    res |= (1 << i);//结果对应的那一位置为1
    *pte &= (~PTE_A);//把PTE_A置为0
  }
  va += PGSIZE;
}
if (copyout(p->pagetable, target_addr, (char*)&res, sizeof(res)) < 0)
  return -1;
```

3.2.2 实验结果

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

```
$ make qemu-gdb
(4.1s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(240.3s)
    (Old xv6.out.usertests failure log removed)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

3.3 心得体会

页表项（PTE）中通过掩码的方式可以存储很多信息，比如本题通过自己添加 PTE_A 这样一个新的标志位就能实现判断某个页面是否被访问过。

Lab 4：Lab Traps

1. RISC-V assembly

1.1 实验目的

了解基本的汇编语言，能读懂汇编程序。

1.2 实验内容

阅读 call.asm 的代码，回答以下问题。

（1）哪些寄存器包含函数的参数？例如，哪个寄存器在 main 对 printf 的调用中存储 13？

main()：

a2：13

a1：12

```
24: 4635                    li a2,13
26: 45b1                    li a1,12
```

a0：main(), g(), f()的返回值，PC 值

```
exit(0);
38: 4501                    li a0,0
```

```
6: 250d                   addiw a0,a0,3
```

```
14: 250d                  addiw a0,a0,3
```

```
28: 00000517               auipc a0,0x0
```

（2）在 main 的汇编代码中，对 f()和 g()的调用在哪里？

没有对于 f()和 g()的调用，而是直接内联优化。

```
26: 45b1                    li a1,12
```

```
14: 250d                      addiw a0,a0,3
```

（3）print 函数的地址是多少？

地址为 630

```
0000000000000630 <printf>:

void
printf(const char *fmt, ...)
{
 630: 711d                addi  sp,sp,-96
```

（4）在 main 函数中，在 jalr 到 printf 后寄存器 ra 的值是多少？

ra=PC+4=0x34+4=0x38

（5）运行以下代码

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

输出是什么？输出取决于 RISC-V 是小端序，如果为大端序设置什么值能获得相同的输出？

输出为：HE110 World

如果为大端序需要把 i=0x00646c72 改为 0x726c6400，57616 不需要改变。

（6）在下面的代码中，y 会打印出什么，为什么是这样的？

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

会打印出寄存器 a2 中的值，因为 printf 会从寄存器 a2 中取出值作为 y 的值。

1.3 实验心得

汇编语言相对高级语言更加接近底层，通过查看汇编代码我们可以了解到一个程序在运行时数据是怎样流动的，以及编译器对于高级语言的代码进行了怎样的优化。

2. Backtrace

2.1 实验目的

在一个程序栈的多个帧中实现返回地址的回溯，在这个过程中，把这些返回地址打印出来。

2.2 实验内容

2.2.1 实验步骤

（1）在 kernel/defs.h 中声明 backtrace 函数，以便 sys_sleep()调用。

```
83   //修改
84   void backtrace();
```

（2）在 riscv.h 中声明 r_fp 函数，以便 backtrace()调用。

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

（3）编写 backtrace 函数。

```
void backtrace()
{
  printf("backtrace:\n");
  uint64 fp = r_fp();//当前栈帧
  uint64 bottom = PGROUNDUP(fp);//栈底
  while (fp < bottom) {
    //一直到栈底
    printf("%p\n", *(uint64*)(fp - 8));
    fp = *(uint64*)(fp - 16);
  }
}
```

2.2.2 实验结果

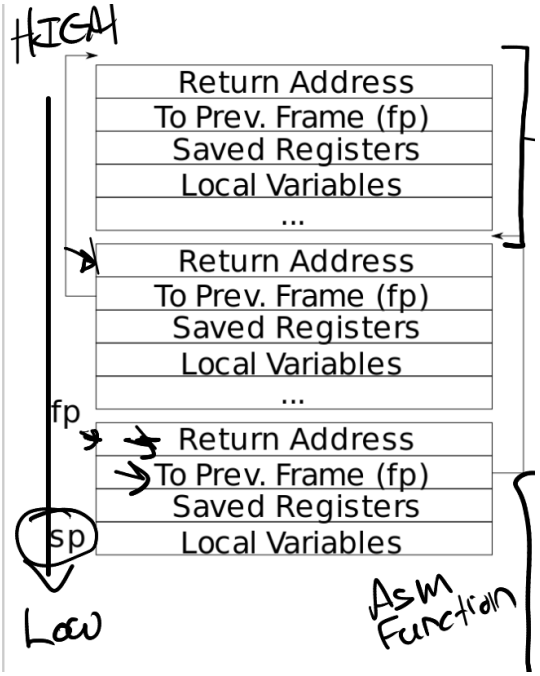
```
$ bttest
backtrace:
0x0000000080002046
0x0000000080001ce2
```

2.3 实验中遇到的问题及解决办法

在 make qemu 后输入 bttest 没有反应，一开始以为是 backtrace 内部的问题或者系统调用没有成功执行，于是在 sys_sleep()和 backtrace()中分别加入了两句打印语句来查错，运行后仍然没有反应。检查 makefile 文件后发现 UPROGS 中没有包含 bttest，也就是说 bttest 没有参与编译，也就不会运行。在把 bttest 加入到 UPROGS 中后运行成功。

```
$U/_bttest\
```

2.4 心得体会



xv6 栈帧的结构如上图所示，每个栈帧都有一个 fp 指针指向，从这个指针向上依次存有函数的返回地址、fp 指针（用于指向上一层函数的栈帧起始地址）、寄存器值、函数的本地变量。通过这样的一个结构，就能实现函数的层层调用、保存现场和恢复现场。

3. Alarm

3.1 实验目的

理解时钟中断的过程与原理，实现两个系统调用 sigalarm(int ticks, void (*handler)())和 sigreturn()。其中，系统调用 sigalarm(int ticks, void (*handler)())的作用是在当前进程运行了 ticks 个时钟周期后转而运行 handler 所指向的函数，而 sigreturn()是 handler 函数调用的系统调用，作用为恢复进程在跳转运行 handler 函数之前的现场。

3.2 实验内容

3.2.1 实验步骤

（1）修改 makefile，把 alarmtest.c 加到 UPROGS 中

```
$U/_alarmtest\
```

（2）在 user.h 中声明函数 sigalarm 和 sigreturn

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

（3）在 user/usys.pl, kernel/syscall.h 和 kernel/syscall.c 中做相应的修改。

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

```
//修改
#define SYS_SIGALARM 22
#define SYS_SIGRETURN 23
```

```
[SYS_SIGALARM]   sys_sigalarm,
[SYS_SIGRETURN] sys_sigreturn
};
```

（4）在 proc.h 的 struct proc 中添加所需要的值，并修改相应的 allocproc()和 freeproc()中的内容。

```
  //修改
  int ticks;//记录目标时钟周期数
  void (*handler)();//记录要调用的函数
  int passed_ticks;//记录已经过去的时钟周期数
  struct trapframe* pretrapframe;//保存现场的指针
};
```

```
  //修改
  p->ticks = 0;
  p->passed_ticks = 0;
  p->handler = 0;

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

（5）实现 sys_sigalarm()，作用为读入用户态传入的参数到 struct proc 的相应域

中。

```
//修改
//系统调用sys_sigalarm，读入参数ticks和handler
uint64 sys_sigalarm(void)
{
  int ticks;
  uint64 handler;
  if(argint(0, &ticks) < 0)
    return -1;
  if(argaddr(1, &handler) < 0)
    return -1;
  struct proc *p = myproc();
  p->ticks = ticks;
  p->handler = (void*)handler;
  return 0;
}
```

（5）在 trap.c 中修改 usertrap()陷入处理函数，使其实现计算时钟周期并在到达目标时钟周期后跳转的功能。

```
// give up the CPU if this is a timer interrupt.
//修改:
if (which_dev == 2) {//==2为时钟中断
  if (p->ticks) {
      p->passed_ticks++;
      if (p->passed_ticks == p->ticks) {
        *p->pretrapframe = *p->trapframe;
        //若间隔的时钟周期数==设定的时钟周期数，调用handler
        p->trapframe->epc = (uint64)p->handler;
      }
  }
  yield();
}
```

这一段的基本逻辑是：

which_dev==2 时代表该中断为时钟中断，每一个时钟周期进行一次时钟中断，意味着在时钟中断时可以判断调用了系统调用后的时钟周期个数。若从系统调用的开始到现在经历的时钟周期个数 passed_ticks==目标周期数 ticks，那么就调用目标函数 handler，即把 handler 的地址传入 PC 中。

在 xv6 系统中，产生中断前的各寄存器状态会存在一个结构体 trapframe 中，当回到用户态的时候通过 trapframe 恢复状态。若直接调用 handler，则调用 sigalarm()之前的寄存器数据会被刷掉，无法恢复原本的状态，此时应当把各寄

存器的值存在 pretrapframe 中，起到保存现场的效果。

（6）实现 sys_sigreturn()，起到从 pretrapframe 中恢复现场的效果。

```
uint64 sys_sigreturn(void)
{
  struct proc* p = myproc();
  p->passed_ticks = 0;
  *p->trapframe = *p->pretrapframe;//恢复现场
  return 0;
}
```

3.2.2 实验结果

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

（下图 FAIL 是因为有理论题，已经写在了实验报告中，没写在 Lab 环境里）

```
make[1]: 离开目录“/home/tanglin/workspace/OS_design/xv6-labs
== Test answers-traps.txt == answers-traps.txt: FAIL
    answers-traps.txt does not seem to contain enough text
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (4.9s)
== Test running alarmtest ==
$ make qemu-gdb
(4.3s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (256.4s)
== Test time ==
time: OK
Score: 80/85
make: *** [Makefile:336: grade] 错误 1
```

3.3 实验中遇到的问题和解决方案

在初步做好实验调试时候发现 test0 会在第一行输出正确后一直循环地打印 alarm!，无法进入 test1。经过查错发现是*p->pretrapframe = *p->trapframe;和 p->trapframe->epc = (uint64)p->handler;的顺序写错了，导致保存了 PC 修改成 handler 地址的数据，从而会循环地执行 handler 函数里面的内容，也就是打印”alarm!"。将这两句代码调换位置后问题解决。

3.4 心得体会

usertrap()为陷入处理程序，负责在核心态中对于不同的中断进行处理调度。usertrap()函数中使用 which_dev 参数对不同的中断分类，其中 2 代表时钟中断。在切换到核心态时，用户态的各寄存器的值会被存在 proc 结构体的 trapframe 中，待回到用户态时恢复现场。

Lab5：Copy-on-Write Fork for xv6

1. Implement copy-on write

1.1 实验目的

在 shell 中执行指令时，首先会 fork 一个子进程，然后在子进程中使用 exec 执行 shell 中的指令。在这个过程中，fork 需要完整的拷贝所有父进程的地址空间，但在 exec 执行时，又会完全丢弃这个地址空间，创建一个新的，因此会造成很大的浪费。本题需要实现一个写时复制机制（COW），通俗地讲就是只有当有写入机制的时候才去申请空间，否则就和父进程共享空间。

COW 共有两个应用场景，一是用户进程写入内存，此时会报写缺页异常，产生 15 号中断；二是在核心态写入内存，此时不会触发中断，但仍然需要 COW。

1.2 实验内容

1.2.1 实验步骤

（1）在 kalloc.h 中添加一个 page_ref 数组，用于存储一个物理界面对应了几个进程。

```
//修改:
//page_ref数组记录一个物理页面对应几个进程
uint page_ref[(PHYSTOP - KERNBASE) / PGSIZE];
```

（2）在 risv.h 中添加宏定义

PTE_COW 为 PTE 的标志位

COW_INDEX(pa) 为 page_ref 的索引，pa 为物理地址，page_ref[COW_INDEX(pa)]即可得到 pa 这个地址的页面有几个进程在使用。

```
//修改
#define PTE_COW (1L << 8)  // copy on write
#define COW_INDEX(pa) (((uint64)(pa) - KERNBASE) >> PGSHIFT)
```

（3）修改 kalloc()和 kfree()，使其符合 COW 的逻辑——kalloc()为申请内存页面，需要将 ref_page 对应项置为 1；kfree()为释放内存，需要判断对应的物理页面是否还有其余的进程在使用，若没有则释放，若有则 ref_page 对应项-1。

```
if (r) {
  page_ref[COW_INDEX(r)] = 1;//第一次申请页面，置为1
  memset((char*)r, 5, PGSIZE); // fill with junk
}
```

```
//修改
//若不止一个虚拟页面对应该物理页面，则不应该free，计数-1即可,否则置为0
if (page_ref[COW_INDEX(pa)] > 1) {
  page_ref[COW_INDEX(pa)]--;
  return;
}
else
  page_ref[COW_INDEX(pa)] = 0;
```

（4）修改 uvmcopy()，使其逻辑变为复制时候不申请内存，而是与要复制的页面共享内存。

```
    panic("uvmcopy: page not present");
  pa = PTE2PA(*pte);//获取父进程的PTE地址
  /*修改：这里注释掉，不需要申请空间，而是直接映射
  flags = PTE_FLAGS(*pte);
  if((mem = kalloc()) == 0)
    goto err;
  memmove(mem, (char*)pa, PGSIZE);*/
  *pte = (*pte & ~PTE_W) | PTE_COW;
  flags = PTE_FLAGS(*pte);
  if (mappages(new, i, PGSIZE, pa, flags) != 0) {
    goto err;
  }
  page_ref[COW_INDEX(pa)]++;//映射时候+1
}
```

（5）修改 usertrap()陷入处理函数，使发生写缺页异常时处理相应的中断。

```
//修改
else if (r_scause() == 15) {
  //13号为读缺页异常，15号为写缺页异常，COW的情况为写缺页
  uint64 va = r_stval();//r_stval()函数：取得引起缺页的虚拟地址
  if(va >= p->sz)
    p->killed = 1;
  else if(cow_alloc(p->pagetable, va) != 0)//调入缺少的页面
    p->killed = 1;
}
```

（6）实现 cow_alloc()，作用为在需要时候申请内存。

```c
//修改
int cow_alloc(pagetable_t pagetable, uint64 va)
{
  va = PGROUNDDOWN(va);//地址对齐
  if (va >= MAXVA)
    return -1;
  pte_t* pte = walk(pagetable, va, 0);
  if (pte == 0)
    return -1;
  uint64 pa = PTE2PA(*pte);
  if (pa == 0)
    return -1;
  uint64 flags = PTE_FLAGS(*pte);
  if (flags & PTE_COW) {
    uint64 mem = (uint64)kalloc();
    if (mem == 0)
      return -1;
    memmove((char*)mem, (char*)pa, PGSIZE);
    uvmunmap(pagetable, va, 1, 1);
```

（7）修改 copyout()，对应在内核态写入内存的情况

```c
while(len > 0){
  va0 = PGROUNDDOWN(dstva);
  //修改
  if (cow_alloc(pagetable, va0) != 0)
    return -1;
  pa0 = walkaddr(pagetable, va0);
```

1.2.2 实验结果

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

1.3 遇到的问题与解决方案

```
$ cowtest
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x00000000000009da stval=0x0000000000003f98
usertrap(): unexpected scause 0x000000000000000f pid=2
            sepc=0x0000000000000002 stval=0x0000000000003f98
usertrap(): unexpected scause 0x000000000000000f pid=1
            sepc=0x00000000000006e8 stval=0x0000000000002f78
panic: init exiting
QEMU: Terminated
```

最开始调试时会遇到以上情况，与期望输出不符。这个输出大概是在说产生了没有预料到的中断，0xf 中断，也就是 15 号中断。但在这个时候我已经写好了usertrap()中对于 15 号中断的处理，因此判断可能是中断判断逻辑出了问题。查代码后发现是因为判断 scause 时候单独用了一个 if 语句，而后面还有其它的 if 语句继续执行，也就会运行到下图语句，和错误输出相符。

```
else {
  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;
```

把原本的 if 语句改为 else if 后测试成功。

1.4 心得体会

在申请内存的时候不必一次性调入所有页面，这样可能造成内存的浪费，使用 COW 即需要的时候再把页面调入内存是一种节约内存的方式。

Lab6：Multithreading

1. Uthread: switching between threads

1.1 实验目的
了解线程切换的原理，完成 uthread.c、uthread_switch.S 的代码。
1.2 实验内容
1.2.1 实验原理
本题要求线程之间切换时候要保存之前的线程中各个寄存器的状态，可以把状态存在 struct thread 这个结构体中。而 uthread_switch()函数的在调用时可以从结构体中取出各寄存器的状态，并为寄存器赋值，从而实现线程切换。
1.2.2 实验步骤
（1）完成 uthread_switch.S（参考 kernel/switch.s）

```
    .globl thread_switch
thread_switch:
    /* YOUR CODE HERE */
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
```

（2）借鉴 kernel/proc.h，定义 contenxt 结构体，表示要存储状态的寄存器，并在 struct thread 中声明相关的域。

```
//修改
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
```

```
struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct context contex;//各寄存器状态
};
```

（3）完成 uthread.c 中需要补充的部分

thread_schedule()：

```
/* YOUR CODE HERE
 * Invoke thread_switch to switch from t to next_thread:
 * thread_switch(??, ??);
 */
thread_switch((uint64)&t->contex, (uint64)&current_thread->contex);//进程切换，修改寄存器
```

thread_create()：

```
t->state = RUNNABLE;
// YOUR CODE HERE
t->contex.ra = (uint64)func;//ra存PC，即为线程代表的函数
t->contex.sp = (uint64)&t->stack + (STACK_SIZE - 1);//存栈顶指针
```

1.2.3 实验结果

```
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

1.3 心得体会

进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位。通过本实验我了解到线程切换的本质就是更改相关的各个寄存器的状态，另外在线程创建时需要将线程的代码地址传入 ra 寄存器（PC）、并让 sp 保存栈顶指针。

2. Using threads

2.1 实验目的

了解多线程和线程锁的概念，理解为什么会产生冲突。完成 not xv6/ph.c，使其运行时保持“0 keys missing”。

2.2 实验内容

2.2.1 实验原理

输入./ph 1 和./ph 2，得到下图所示的输出。发现线程 2 会出现 keys missing 的情况。原因在于线程 1 和 2 共同使用一个哈希表来存值，两个线程同时运行就可能造成冲突，可以通过给哈希表上锁来解决这个问题。

```
tanglin@ubuntu:~/workspace/OS_design/xv6-labs-2021$ ./ph 1
100000 puts, 6.622 seconds, 15102 puts/second
0: 0 keys missing
100000 gets, 6.626 seconds, 15093 gets/second
tanglin@ubuntu:~/workspace/OS_design/xv6-labs-2021$ ./ph 2
100000 puts, 5.322 seconds, 18791 puts/second
0: 15440 keys missing
1: 15440 keys missing
200000 gets, 7.017 seconds, 28503 gets/second
```

2.2.2 回答问题

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

答：

假设键 k1、k2 属于同个 bucket

thread 1: 尝试设置 k1，发现 k1 不存在，尝试在 bucket 末尾插入 k1。

此时线程切换到 thread 2

thread 2: 尝试设置 k2，发现 k2 不存在，尝试在 bucket 末尾插入 k2。分配 entry，在桶末尾插入 k2

此时线程切换回 thread 1

thread 1: 分配 entry，没有意识到 k2 的存在，在其认为的 “桶末尾”（实际为 k2 所处位置）插入 k1

k1 被插入，但是由于被 k1 覆盖，k2 从桶中消失了，引发了键值丢失。

2.2.3 实验步骤

完成对 put 和 get 操作上锁

```
pthread_mutex_t lock[NBUCKET];//声明锁
```

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

  // is the key already present?
  //修改: 上锁
  pthread_mutex_lock(&lock[i]);
  struct entry* e = 0;
```

```
  //修改: 开锁
  pthread_mutex_unlock(&lock[i]);
}
```

```
get(int key)
{
  int i = key % NBUCKET;

  //修改: 上锁
  pthread_mutex_lock(&lock[i]);
  struct entry *e = 0;
//修改: 开锁
pthread_mutex_unlock(&lock[i]);
return e;
```

2.2.4 实验结果

```
tanglin@ubuntu:~/workspace/OS_design/xv6-labs-2021$ ./ph 1
100000 puts, 6.388 seconds, 15654 puts/second
0: 0 keys missing
100000 gets, 6.315 seconds, 15835 gets/second
tanglin@ubuntu:~/workspace/OS_design/xv6-labs-2021$ ./ph 2
100000 puts, 4.815 seconds, 20766 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 9.230 seconds, 21668 gets/second
```

```
ph_safe: OK (17.4s)
```

2.3 遇到的问题和解决方案

在上锁时候考虑过可以把整个哈希表都上锁，但这样就会导致在同一时间只有一个线程可以访问并修改哈希表，哪怕另一个线程不冲突也不能运行，这样会非常影响多线程的效率。因此把锁的形式改成哈希表的每一个项都有一个单独的锁，这样只有在某一项发生冲突的时候会让线程停下来。

2.4 心得体会

多线程使一种有效提高程序运行效率的方式，但存在着各个线程之间同步和冲突的问题，需要注意。

3. Barrier

3.1 实验目的

屏障 (Barrier) 允许每个线程都处于等待状态，直到所有的合作线程都达到某一执行点，然后从该点继续执行。本实验需要完成 Barrier()函数，从而实现一个屏障。

3.2 实验内容

3.2.1 实验步骤

完成 Barrier()函数

```
    // then increment bstate.round.
    //
    pthread_mutex_lock(&bstate.barrier_mutex);
    if (++bstate.nthread == nthread) {//达到条件：所有的线程都达到了屏障
      bstate.nthread = 0;
      bstate.round++;
      pthread_cond_broadcast(&bstate.barrier_cond);//唤醒所有在wait的线程
    }
    else {//不满足条件
      pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);//挂起线程
    }
    pthread_mutex_unlock(&bstate.barrier_mutex);
  }
```

3.2.2 实验结果

```
tanglin@ubuntu:~/workspace/OS_design/xv6-labs-2021$ ./barrier 2
OK; passed
tanglin@ubuntu:~/workspace/OS_design/xv6-labs-2021$
```

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (11.0s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/tanglin/workspace/OS_design/xv6-labs-2021”
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录“/home/tanglin/workspace/OS_design/xv6-labs-2021”
ph_safe: OK (15.9s)
== Test ph_fast == make[1]: 进入目录“/home/tanglin/workspace/OS_design/xv6-labs-2021”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/tanglin/workspace/OS_design/xv6-labs-2021”
ph_fast: OK (28.3s)
== Test barrier == make[1]: 进入目录“/home/tanglin/workspace/OS_design/xv6-labs-2021”
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录“/home/tanglin/workspace/OS_design/xv6-labs-2021”
barrier: OK (11.8s)
== Test time ==
time: OK
Score: 60/60
```

Lab7：Lock

1. Memory allocator

1.1 实验目的

当只使用一个 freelist 时，会产生内存块的竞争问题，为了解决这个问题，需要重新安排内存管理器的形式。具体地讲，本实验需要把原本只有一个 freelist 的方式改成每个 CPU 都有自己的一个 freelist 的方式，并且若某个 CPU 的内存不足，可以从其它有空闲的 CPU 处“偷”一块内存。

1.2 实验内容

1.2.1 实验步骤

（1）修改 kmem 结构体，使每个 CPU 都有这样一个结构体

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU];//修改：使每个CPU都有都有一个freelist
```

（2）根据新的 kmem 结构修改对应的 kinit(), kfree(), kalloc()

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

```
  //修改
  push_off();//关中断
  int id = cpuid();//获取当前cpuid
  pop_off();//开中断
  acquire(&kmem[id].lock);
  r->next = kmem[id].freelist;
  kmem[id].freelist = r;
  release(&kmem[id].lock);
}
```

```
struct run *r;
//修改
push_off();//关中断
int id = cpuid();//获取当前cpuid
pop_off();//开中断
acquire(&kmem[id].lock);
r = kmem[id].freelist;
if (r)//本cpu有空闲
  kmem[id].freelist = r->next;
else {
  //否则从其它cpu找
  int i;
  for (i = 0;i < NCPU;i++) {
    if (i == id)
      continue;
    acquire(&kmem[i].lock);
    r = kmem[i].freelist;
    if (r) {//i有空闲
      kmem[i].freelist = r->next;
    }
    release(&kmem[i].lock);
```

1.2.2 实验结果

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 186344
lock: kmem: #test-and-set 0 #acquire() 114084
lock: kmem: #test-and-set 0 #acquire() 132640
lock: bcache: #test-and-set 0 #acquire() 1248
--- top 5 contended locks:
lock: proc: #test-and-set 3887546 #acquire() 840717
lock: proc: #test-and-set 3488488 #acquire() 840717
lock: proc: #test-and-set 2927636 #acquire() 840717
lock: proc: #test-and-set 2758048 #acquire() 840717
lock: proc: #test-and-set 2713789 #acquire() 840717
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

1.3 心得体会

每个 CPU 都可以有自己的内存，给每个 CPU 一个 freelist 和对应的锁是可以提高计算机的并行度，与上一个实验一样，都是采用了把粗粒度变成细粒度从而提高并行度的思想。

2. Buffer cache

2.1 实验目的

cache 缓存块存在竞争的问题，本题需要修改块缓存，以便在运行 bcachetest 时，bcache 中所有锁的 acquire 循环迭代次数尽可能地小。

2.2 实验内容

2.2.1 实验原理

本题和上一题的思路类似，缓存锁竞争过多的主要原因是锁的粒度过大，可以通过减小粒度来缓解这个问题。与上一题不同的是，如果只是减小锁的粒度可能会造成死锁。比如，一个进程给自己的 bucket 上锁，在自己的 bucket 中没找到对应的缓存块，去另一个 bucket 中寻找，而这个 bucket 对应的并发进程同样给自己的 bucket 上了锁，试图寻找对应的缓存块。这样这两个进程就造成了死锁现象。可以通过给整个 buf 上一把大锁来解决这个问题。

此外，缓存块的搜索遵循以下逻辑：

若能在本 bucket 中找到命中的缓存块，则直接返回；若未命中，则在本 bucket 中遵循 LRU 的原则找空闲块；若本 bucket 中没有空闲块，则去其它 bucket 中找；若还找不到空闲块则缓存块分配失败。

2.2.2 实验步骤

（1）修改 buf 的结构，把 buf 分成 NBUCKET 个部分，并添加 biglock

```
struct {
  //struct spinlock biglock;
   struct spinlock biglock;
  struct spinlock lock[NBUCKET];
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // Sorted by how recently the buffer was used.
  // head.next is most recent, head.prev is least.
  struct buf head[NBUCKET];
} bcache;
```

（2）添加 hash 函数，便于检索 bucket

```
int hash(int n)
{
  return n % NBUCKET;
}
```

（3）修改 binit、bpin、bunpin，使其符合修改后的 buf 结构

```
binit(void)
{
  struct buf *b;
  int i;
  initlock(&bcache.biglock, "bcache_biglock");
  for (i = 0;i < NBUCKET;i++)
    initlock(&bcache.lock[i], "bcache");

  // Create linked list of buffers
  for (i = 0;i < NBUCKET;i++) {
    bcache.head[i].prev = &bcache.head[i];
    bcache.head[i].next = &bcache.head[i];
  }
  for (b = bcache.buf; b < bcache.buf + NBUF; b++) {
    b->next = bcache.head[0].next;
    b->prev = &bcache.head[0];
    initsleeplock(&b->lock, "buffer");
```

```
void
bpin(struct buf* b) {
  int i = hash(b->blockno);
  acquire(&bcache.lock[i]);
  b->refcnt++;
  release(&bcache.lock[i]);
}

void
bunpin(struct buf* b) {
  int i = hash(b->blockno);
  acquire(&bcache.lock[i]);
  b->refcnt--;
  release(&bcache.lock[i]);
}
```

（4）修改 bget

在本 bucket 中有命中的缓存块，直接返回

```
for (b = bcache.head[i].next; b != &bcache.head[i]; b = b->next) {
  if (b->dev == dev && b->blockno == blockno) {
    //在缓存中，直接返回即可
    b->refcnt++;
    release(&bcache.lock[i]);
    acquiresleep(&b->lock);
    return b;
  }
}
```

若未命中，则在本 bucket 中找空闲块

```
  }
  //不在缓存中，LRU找空闲块
  for (b = bcache.head[i].next;b != &bcache.head[i];b = b->next) {
    if (b->refcnt == 0 && (b2 == 0 || b->lastuse < min_ticks)) {
      min_ticks = b->lastuse;
      b2 = b;
    }
  }
  if (b2) {
    //找到空闲块了
    b2->dev = dev;
    b2->blockno = blockno;
    b2->refcnt++;
    b2->valid = 0;
    release(&bcache.lock[i]);
    release(&bcache.biglock);
    acquiresleep(&b2->lock);
    return b2;
  }
  else {
```

若没有空闲块再去其它 bucket 中找

```
else {
  //没找到，去其它bucket里找
  int j;
  min_ticks = 0xfffffff;
  for (j = 0;j < NBUCKET;j++) {
    if (j == hash(blockno))
      continue;
    acquire(&bcache.lock[j]);
    for (b = bcache.head[j].next; b != &bcache.head[j]; b = b->next) {
      if (b->refcnt == 0 && (b2 == 0 || b->lastuse < min_ticks)) {
        min_ticks = b->lastuse;
        b2 = b;
      }
    }
    if (b2) {
```

2.2.3 实验结果

```
lock: kmem: #test-and-set 0 #acquire() 51
lock: kmem: #test-and-set 0 #acquire() 51
lock: kmem: #test-and-set 0 #acquire() 51
lock: bcache: #test-and-set 0 #acquire() 6203
lock: bcache: #test-and-set 0 #acquire() 6178
lock: bcache: #test-and-set 0 #acquire() 4282
lock: bcache: #test-and-set 0 #acquire() 4280
lock: bcache: #test-and-set 0 #acquire() 2264
lock: bcache: #test-and-set 0 #acquire() 4260
lock: bcache: #test-and-set 0 #acquire() 2680
lock: bcache: #test-and-set 0 #acquire() 7092
lock: bcache: #test-and-set 0 #acquire() 4174
lock: bcache: #test-and-set 0 #acquire() 6176
lock: bcache: #test-and-set 0 #acquire() 6176
lock: bcache: #test-and-set 0 #acquire() 6176
lock: bcache: #test-and-set 0 #acquire() 6174
--- top 5 contended locks:
lock: proc: #test-and-set 21113506 #acquire() 7121350
lock: proc: #test-and-set 20993186 #acquire() 7121350
lock: proc: #test-and-set 19608866 #acquire() 7121350
lock: proc: #test-and-set 19569414 #acquire() 7121350
lock: proc: #test-and-set 18727463 #acquire() 7121349
tot= 0
test0: OK
start test1
test1 OK
```

```
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (25.7s)
== Test running bcachetest ==
$ make qemu-gdb
(30.9s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
```

2.3 遇到的问题与解决方案

一开始写的版本在本 bucket 没命中后直接去找 bucket 中的空闲块，结果锁冲突的次数仍然很高，无法通过测试。后来在查了相关资料后了解到未命中这一段时间内其它进程也有可能把对应的内容写入缓存，因此需要重新再遍历一遍缓存块，加入以下代码后实验通过。

```
for (b = bcache.head[i].next; b != &bcache.head[i]; b = b->next)
  if (b->dev == dev && b->blockno == blockno) {
    b->refcnt++;
    release(&bcache.lock[i]);
    release(&bcache.biglock);
    acquiresleep(&b->lock);
    return b;
  }
}
```

2.4 心得体会

锁的冲突过多很有可能是锁的粒度过大造成的，这两个题目的核心思想均为减小锁的粒度从而降低冲突的频率。但在细分粒度的同时也要注意潜在的死锁问题。

Lab8：File System

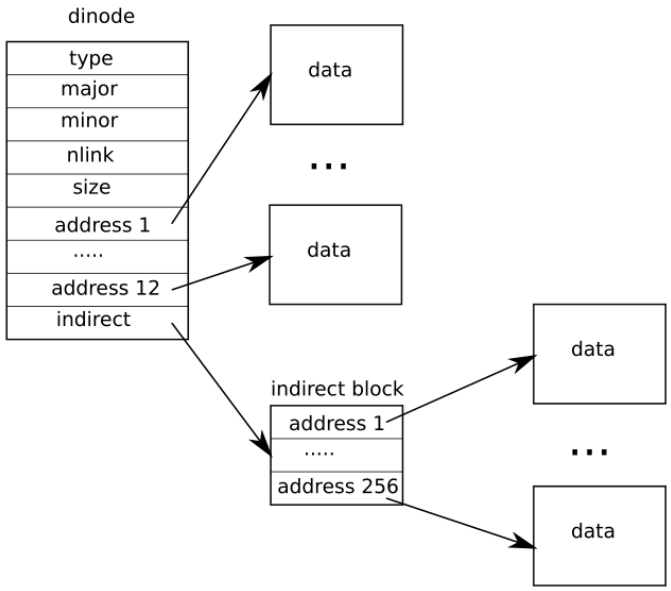
1. Large Files

1.1 实验目的

修改 xv6 的文件系统，使其支持大文件。

1.2 实验内容

1.2.1 实验原理



xv6 的文件系统如上图所示，dinode 是磁盘上存储文件信息的结点，其中 address1-address12 为直接索引，指向一个数据块，indirect 为一级索引，指向 indirect block，indirect block 中的每一项又指向对应的一个数据块。因此，一个文件最多能够占用 12+256=268 个数据块，想要对其进行扩充就需要修改索引的结构，使其支持二级索引。根据实验指导书上的内容，本题需要把 address12 改为一级索引，indirect 的部分改为二级索引。

1.2.2 实验步骤

（1）修改 inode 和 dinode

减少一个直接索引，间接索引变为一个一级索引、一个二级索引，其中二级索引代表的数据块数目=一级索引对应的数据块数的平方。

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

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

（2）修改 bmap

参照 bmap 代码中实现一级索引的部分，实现二级索引。

```
//实现二级索引
bn -= NINDIRECT;
if (bn < NDINDIRECT) {
  if ((addr = ip->addrs[NDIRECT + 1]) == 0)
    ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  //一级索引
  if ((addr = a[bn / NINDIRECT]) == 0) {
    a[bn / NINDIRECT] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
  // 二级索引
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if ((addr = a[bn % NINDIRECT]) == 0) {
    a[bn % NINDIRECT] = addr = balloc(ip->dev);
    log_write(bp);
  }
```

（3）修改 itrunc

当文件内容被删除时，需要使用 itrunc 对各个数据块进行释放。参照更改一级索引的部分，使 itrunc 支持二级索引。

```
  //二级索引
  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]) {//一级索引里的项不为空
        bp2 = bread(ip->dev, a[j]);
        a2 = (uint*)bp2->data;
        for(i = 0; i < NINDIRECT; i++) {
          if (a2[i])//释放数据块
            bfree(ip->dev, a2[i]);
        }
        brelse(bp2);
        //释放二级索引
        bfree(ip->dev, a[j]);
        a[j] = 0;
      }
    }
    brelse(bp);
    //释放一级索引
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
```

1.2.3 实验结果

```
init: starting sh
$ bigfile
..........................................................
..........................................................
..........................................................
..........................................................
..........................................................
..........................................
wrote 65803 blocks
bigfile done; ok
```

1.3 心得体会

xv6 文件系统采用的是多级索引结构，小文件只需要使用直接索引，大文件则需要使用一级乃至更多级的索引，而这些信息只需要存储在大小固定的 inode 中。

2. Symbolic links

2.1 实验目的

实现系统调用 symlink()，即软链接功能。

2.2 实验内容

2.2.1 实验原理

访问一个文件一定要访问这个文件对应的 inode，inode 记录了文件的类型、数据块的索引，软链接也会经历同样的过程。因此可以把软链接当作一种文件类型，目标链接的路径存在文件数据块里，再对打开文件的逻辑进行修改，就可以做到在访问到软链接文件时继续访问它所指向的路径的文件。

2.2.2 实验步骤

（1）添加系统调用

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

```
//修改: 添加symlink
int symlink(const char*, const char*);
```

```
//修改: 添加系统调用
#define SYS_symlink 22
```

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

```
[SYS_symlink] sys_symlink
```

（2）编写 sys_symlink()

```c
sys_symlink(void)
{
  char path[MAXPATH], target[MAXPATH];
  struct inode *ip;
  // 读取参数
  if(argstr(0, target, MAXPATH) < 0)
    return -1;
  if(argstr(1, path, MAXPATH) < 0)
    return -1;
  // 开启事务
  begin_op();
  // 为这个符号链接新建一个 inode
  if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
    end_op();
    return -1;
  }
  // 在符号链接的 data 中写入被链接的文件
  if(writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
```

（3）修改 sys_open()，使其支持打开软链接类型的文件。

```c
sysfile.c > sys_symlink(void)
//修改.
while (ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
  // 如果访问深度过大，则退出
  if (depth++ >= 20) {
    iunlockput(ip);
    end_op();
    return -1;
  }
  // 读取对应的 inode
  if(readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH) {
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlockput(ip);
  // 根据文件名称找到对应的 inode
  if((ip = namei(path)) == 0) {
    end_op();
    return -1;
  }
  ilock(ip);
```

2.2.3 实验结果

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

3.3 心得体会

系统中的文件和我们平时直观感受到的文件是不一样的，有数据文件、路径等类型，也可以自定义类似软链接这样的文件类型，本质上都是由 inode 存储、索引的一些信息，但解析方式有一定的区别。

Lab9 mmap

1. mmap

1.1 实验目的
实现 mmap()和 munmap()系统调用。
1.2 实验内容
1.2.1 实验原理
mmap()系统调用可以将文件映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系，使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以向访问普通内存一样对文件进行访问，不必再调用 read()，write（）等操作。而 munmap()系统调用则是清除虚拟内存空间与磁盘地址之间的映射。
1.2.2 实验步骤
（1）添加系统调用

```
//修改: 添加系统调用
void* mmap(void* addr, int length, int prot, int flags, int fd, uint offset);
int munmap(void *addr, int length);
```

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

```
//添加系统调用
#define SYS_mmap   22
#define SYS_munmap 23
```

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

```
[SYS_mmap]    sys_mmap,
[SYS_munmap]  sys_munmap
```

（2）添加 VMA 结构体，表示虚拟内存空间

```
#define VMASIZE 16
struct vma {
  int used;
  uint64 addr;
  int length;
  int prot;
  int flags;
  int fd;
  int offset;
  struct file* file;
};
```

```
struct vma vma[VMASIZE];
```

（3）编写 sys_mmap()

```
uint64
sys_mmap(void)
{
  uint64 addr;
  int length, prot, flags, fd, offset;
  struct file* file;
  struct proc* p = myproc();
  //读取参数
  if (argaddr(0, &addr) || argint(1, &length) || argint(2, &prot) ||
    argint(3, &flags) || argfd(4, &fd, &file) || argint(5, &offset)) {
    return -1;
  }
  if (!file->writable && (prot & PROT_WRITE) && flags == MAP_SHARED)
    return -1;
  length = PGROUNDUP(length);//内存对齐
  if (p->sz > MAXVA - length)//过大
    return -1;
```

```
  //遍历VMA数组寻找空闲VMA
  for (int i = 0; i < VMASIZE; i++) {
    if (p->vma[i].used == 0) {
      //若为空闲，对这个VMA初始化
      p->vma[i].used = 1;
      p->vma[i].addr = p->sz;
      p->vma[i].length = length;
      p->vma[i].prot = prot;
      p->vma[i].flags = flags;
      p->vma[i].fd = fd;
      p->vma[i].file = file;
      p->vma[i].offset = offset;
      filedup(file);
      p->sz += length;
      return p->vma[i].addr;//返回地址
    }
  }
  return -1;
}
```

（4）修改 usertrap，实现发生缺页异常时自动读入页面（与 Lab5 类似）。

```
else if (r_scause() == 13 || r_scause() == 15) {
  uint64 va = r_stval();
  if (va >= p->sz || va > MAXVA || PGROUNDUP(va) == PGROUNDDOWN(p->trapframe->sp)) p->killed = 1;
  else {
    struct vma* vma = 0;
    for (int i = 0; i < VMASIZE; i++) {
      if (p->vma[i].used == 1 && va >= p->vma[i].addr && va < p->vma[i].addr + p->vma[i].length) {
        //为所求的VMA则把值赋给vma
        vma = &p->vma[i];
        break;
      }
    }
    if (vma) {
      va = PGROUNDDOWN(va);
      uint64 offset = va - vma->addr;
      uint64 mem = (uint64)kalloc();
      if (mem == 0) {
        p->killed = 1;
      }
      else {
        memset((void*)mem, 0, PGSIZE);
        //读入inode
        ilock(vma->file->ip);
        readi(vma->file->ip, 0, mem, offset, PGSIZE);
        iunlock(vma->file->ip);
```

（5）修改 uvmunmap()和 uvmcopy()循环中的判断条件，使其符合缺页时调入的逻辑

```
if ((*pte & PTE_V) == 0)
  continue;
```

（6）编写 sys_munmap()，使其能够取消虚拟内存空间的映射。

```
uint64
sys_munmap(void)
{
  uint64 addr;
  int length;
  struct proc* p = myproc();
  struct vma* vma = 0;
  //读参数
  if (argaddr(0, &addr) || argint(1, &length))
    return -1;
  addr = PGROUNDDOWN(addr);
  length = PGROUNDUP(length);
  for (int i = 0; i < VMASIZE; i++) {
    if (addr >= p->vma[i].addr || addr < p->vma[i].addr + p->vma[i].length) {
      vma = &p->vma[i];
      break;
    }
  }
```

```
if (vma == 0) return 0;
if (vma->addr == addr) {
  vma->addr += length;
  vma->length -= length;
  //写回
  if (vma->flags & MAP_SHARED)
    filewrite(vma->file, addr, length);
  //取消映射
  uvmunmap(p->pagetable, addr, length / PGSIZE, 1);
  //清除占用的VMA
  if (vma->length == 0) {
    fileclose(vma->file);
    vma->used = 0;
  }
}
return 0;
```

（7）修改 fork()和 exit()，在进程创建和退出时，需要复制和清空相应的文件映射。

```
np->state = RUNNABLE;
//修改：复制VMA到子进程
for (int i = 0; i < VMASIZE; i++) {
  if (p->vma[i].used) {
    memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));
    filedup(p->vma[i].file);
  }
}
release(&np->lock);
```

```
//修改：清除VMA
for (int i = 0; i < VMASIZE; i++) {
  if (p->vma[i].used) {
    if (p->vma[i].flags & MAP_SHARED)
      filewrite(p->vma[i].file, p->vma[i].addr, p->vma[i].length);
    fileclose(p->vma[i].file);
    uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].length / PGSIZE, 1);
    p->vma[i].used = 0;
  }
}
```

1.2.3 实验结果

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

1.3 心得体会

mmap 给我的感觉是所有实验里面最难的实验，需要综合到前面几个实验的内容，比如 Lab5 里利用缺页中断读入页面、比如编写系统调用的方法。另外我对 mmap()和 munmap()这两个系统调用不太熟悉，通过查资料才了解到了它们具体的作用。