南京航空航天大学

操作系统实践

Proj5

班级： 1618001
学号： 161840230
姓名： 王可

一、 题目简述

添加文件系统校验

当xv6启动时，在mpmain()调用scheduler()函数之前，你需要检验文件系统的一致性，并修正不一致的

部分。本实验中你仅需要考虑一种不一致性：**有些inode已经被使用了，但无法从根目录搜寻到它。**比如，/home/xzhu/1.txt对应4个inode，其中，根目录、home目录、xzhu目录、1.txt文件分别对应一个inode。此时，如果根目录的内容中删除了home目录的条目，则home目录、xzhu目录、1.txt文件的inode均有问题；反之，如果仅是home目录中的xzhu条目被删除，则xzhu目录、1.txt文件的inode有问题。修正方式是，归还这些有问题的inode，并归还inode指向的文件所占用的磁盘空间。或者说，将删除操作执行完。

- **问题解释在第二部分的第五小点展开**

二、 实验过程

1. 添加文件系统校验

1.1 全局考虑

1. 在完成文件系统的遍历时，首先需要了解xv6中FILE SYSTEM的结构，如下图，第一个分区为boot block，接下来是superblock，存储了整个文件系统的大小和分区结构，而后存储了题目中提到的200个inode，bitmap以及存放数据的data blocks：

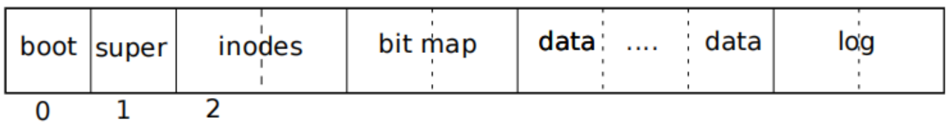


Figure 6-2. Structure of the xv6 file system. The header `fs.h` (3650) contains constants and data structures describing the exact layout of the file system.

然而，通过查询官方文档，我发现了如下描述(仅截取有用片段）：

The fifile system does not use block 0 (it holds the boot sector). Block 1 is called the superblock; it contains metadata about the fifile system (the fifile system size in blocks, the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold inodes, with multiple inodes per block.

该段表示xv6中并没有使用第一个分区，即boot block，因此在进行磁盘读写获取superblock的时候，需要注意偏移量offset为0即可。

2. 关于inode无法一次性放入内存的问题，考虑使用了栈缓冲的做法，栈的优势在于不需要程序员主动释放空间，这里算是偷了个小懒，但后面发现出了一点小小的意外（但针对此测试文件无影响）。事实上使用堆的方法与我使用的方法实现差别不大，因此改动几行代码便可以将此实现改为堆实现，所以可以理解为我这里栈的实现方式仅仅是为了尝试练习对指针、地址的控制而使用的。

3. 完成本次实验很重要的一点是需要了解inode的结构，从文档中了解结构如下图所示——除去前面的一些标志外，存放了13个addr。其中前12个为direct address，最后一个为single indirect

block，而single所指向的块存放了另外128个addr，因此一个inode最多可以指向140个地址。**若深入考虑，如果inode被释放时，indirect block存在，则需要将此块也释放。**出于时间原因，暂时没有实现。

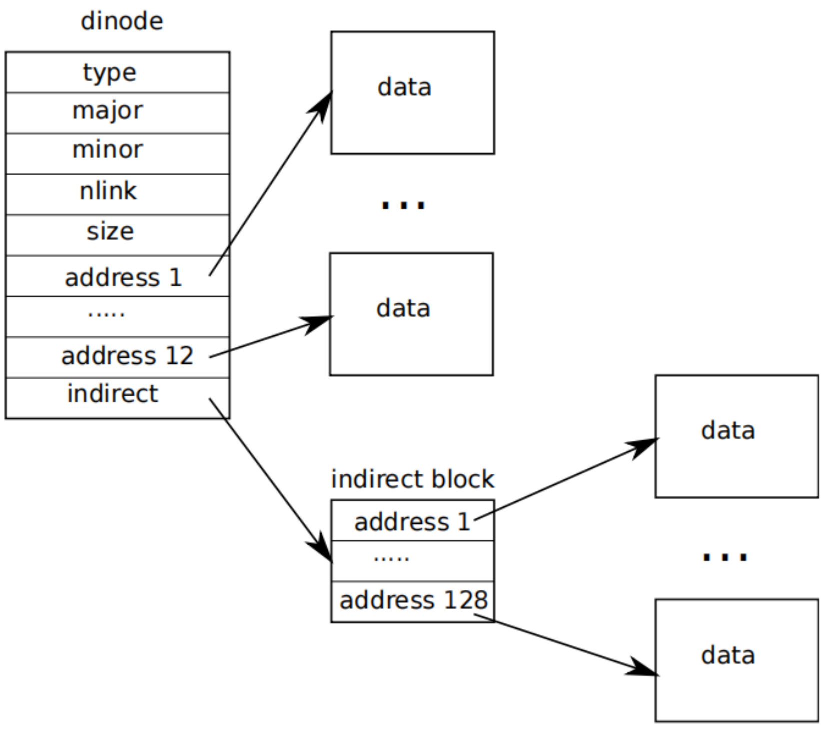


Figure 6-4. The representation of a file on disk.

1.2 读写

1.2.1 主要思路

读写磁盘的代码基本仿照了bootmain.c中的实现，具体代码如下，可以看出几处有限的修改：

1. readsec/writesec中将0xE0改为了0xF0,这是修改的磁盘目标，改为对img1的读写。
2. 写磁盘的部分将insl改为了outsl并将第二个参数改为了0x30.

1.2.2 实现代码

```
// copy from bootmain.c
void
waitdisk(void)
{
  // Wait for disk ready
  while((inb(0x1F7) & 0xC0) != 0x40);
}

// copy from bootmain.c
// Read a single sector at offset into dst.
void
readsec(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
```

```c
  outb(0x1F5, offset >> 16);
  //------------------------------------------
  // E0->F0  Read from disk img 1
  outb(0x1F6, (offset >> 24) | 0xF0);
  //------------------------------------------
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}

// Write a single sector at offset into dst.
void
Writesec(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  //------------------------------------------
  // E0->F0  Read from disk img 1
  outb(0x1F6, (offset >> 24) | 0xF0);
  //------------------------------------------
  outb(0x1F7, 0x30);  // cmd 0x30 - write sectors

  // Read data.
  waitdisk();
  outsl(0x1F0, dst, SECTSIZE/4);
}

// copy from bootmain.c
void
readblock(char* pa, uint count, uint offset)
{
  char* epa;

  epa = pa + count;

  // Round down to sector boundary.
  pa -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We'd write more to memory than asked, but it doesn't matter --
  // we load in increasing order.
  for(; pa < epa; pa += SECTSIZE, offset++)
    readsec(pa, offset);
}

int flag[250];

void
writeblock(char* pa, uint count, uint offset)
```

```c
{
  char* epa;

  epa = pa + count;

  // Round down to sector boundary.
  pa -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We'd write more to memory than asked, but it doesn't matter --
  // we load in increasing order.
  for(; pa < epa; pa += SECTSIZE, offset++)
    writesec(pa, offset);
}
```

1.3 第一次扫描

1.3.1 主要思路

进行第一次扫描时，我需要获取所有通过正常路径能够访问到的文件（inode），即从根目录可以访问到的文件，题目已给信息提到，根目录root存放在inum为1的dinode里，因此需要从此处进行遍历查找，获得所有合法inode，并将它们进行标记。此处使用了递归的方法，对于目录文件，首先遍历其直接地址，再考虑一次间接地址。代码的含义在下面有具体的注释：其一般流程为先获取inode中存放的地址，获取地址指向的block后，从中读取directory entry并搜索其后继，获取新的inode进行递归查找。

在实现中发现，递归层次若太多，则会产生栈溢出的问题。这个问题可以通过开全局变量（但是这样不加分）或者使用堆分配动态空间的方法解决该问题。

1.3.2 实现代码

```c
void
dfs(const char* name, const struct dinode* inode,int depth){
  // address of dataBlock going to access
  uint addr;
  // dataBlock
  char db[512];
  // single block

  // inode
  struct dinode INODES[8];
  // cprintf("before if\n");
  if(inode->type == 0) return;
  if(inode->type == T_DIR){
    // 12 direct address
    for(int i = 0; i < NDIRECT; i++){
      addr = inode->addrs[i];
      if(0 == addr) {
        continue;
      }
      // get the data block on the address
      readblock(&db[0],BSIZE,(addr-1) * BSIZE);
      struct dirent* entry = (struct dirent*)(&db);
      for(int j = 0; j < ENTNUM; j++){
        // get access to each directory entry saved in the data block
        if(entry->name[0]=='.'||
        entry->inum<=0||
        entry->inum>200){
          entry++;
          continue;
        }
        // get the inode metioned in the entry
        readblock((char*)(&INODES),BSIZE, BSIZE + logNum * BSIZE + (entry->inum/8 * BSIZE));
        struct dinode* next_inode = &INODES[entry->inum%8];
        // dfs into the newly got inode and search for more legal files
        if(next_inode==inode) continue;
        // flag the found dir/file
        flag[entry->inum] = 1;
        dfs(entry->name,next_inode,depth+1);
        entry++;
      }
    }

    addr = inode->addrs[NDIRECT];
    if(addr == 0) return;
    // get the single indirect block
    readblock(&single[0],4096,(addr-1) * BSIZE);
    // for the 128 addresses in the block, find all avaliable ones
    // the address is 4-Byte long, so the number i adds itself by 4 one time
    for(int i = 0;i < NINDIRECT;i+=4){
      addr = (uint)*(&single + i);
      if(addr == 0) continue;
      readblock(&db[0],BSIZE,(addr - 1) * BSIZE);
      struct dirent* entry = (struct dirent*)(&db);
      // if the address is avaliable, look for new files and the following is some familiar actions
      for(int j = 0; j < ENTNUM; j++,entry++){
        if(entry->name[0]=='.'||
        entry->inum==0||
        entry->inum>200){
          continue;
        }
        readblock((char*)(&INODES),BSIZE, BSIZE + logNum * BSIZE + (entry->inum/8 * BSIZE));
        struct dinode* next_inode = &INODES[entry->inum%8];
        flag[entry->inum] = 1;
        dfs(entry->name,next_inode,depth+1);
      }
    }
  }else{
      // cprintf("name :  %s\n",name);
  }
}
```

1.4 第二次扫描

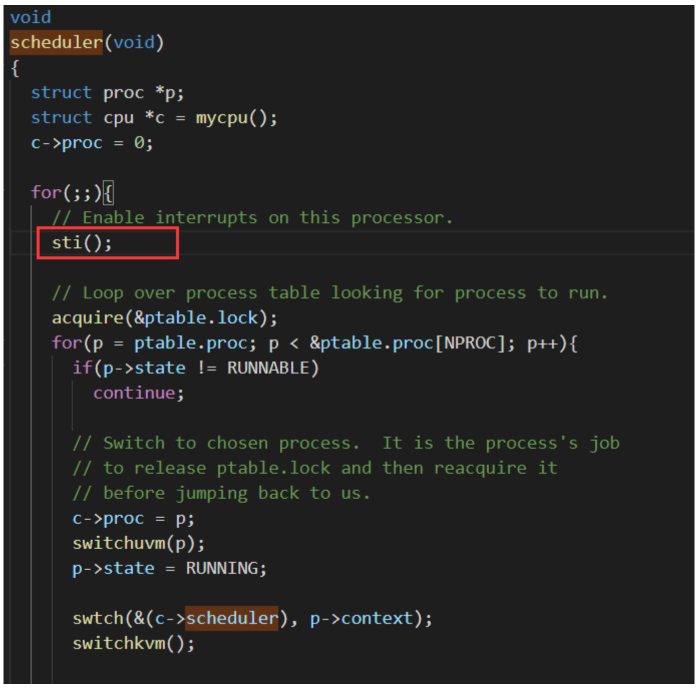
第二次扫描的目标是对所有inode进行扫描，对于那些自身生效，但是其指向的块儿没有在第一次扫描中被标记的inode，进行释放操作，具体操作为将其type置为0，并重新写回到磁盘中。

```c
void check_fs()
{
  struct superblock sb;
  struct dinode INODE[8];
  struct dinode *check;

  flag[ROOTINO] = 1;
    // check
  int checkFlag = 0;
    // False Num
  int Fnum = 0;

  readblock((char*)(&sb),BSIZE,0);
  readblock((char*)(&INODE),BSIZE,BSIZE + (sb.nlog) * BSIZE);
  logNum = sb.nlog;
  struct dinode * root = &INODE[1];
  dfs("/",root,1);


  for(int i = 1; i < sb.ninodes; i++){
    if(i % 8 == 0){
      readblock((char*)(&INODE),BSIZE,BSIZE + (sb.nlog) * BSIZE + i / 8 * BSIZE);
    }
    check = &INODE[i % 8];
    if(flag[i] == 0 && check->type != 0){
      check->type = 0;
      Fnum = Fnum + 1;
      checkFlag = 1;
      cprintf("\nfsck: inode %d is allocated but is not referenced by any dir!",i);
      writeblock((char*)(&INODE),BSIZE,BSIZE + (sb.nlog) * BSIZE + (i / 8) * BSIZE);
      cprintf("Fixing ... done");
    }
  }
  if(checkFlag){
    cprintf("\nfsck completed. Fixed %d inodes and freed %d disk blocks.\n",Fnum,Fnum);
  }else{
    cprintf(" fsck: no problem found. ");
  }
}
```

1.5 问题解释

> 1. 为何OS尚未切换到scheduler时，读写磁盘不能用中断的方式进行?

在main函数中，操作系统正在进行初始化，若产生中断将使得部分功能未开启的环境下进入中断处理程序（理应是健全的），因此此时系统将中断使能关闭，而在scheduler函数中（如下图）调用了sli函数打开中断，此时才能触发中断。我们所进行的磁盘读取是在scheduler函数调用之前，因此不能使用中断。



```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();
```

> 2. 写出你遍历目录的伪代码。

函数： Search

输入： dinode* root

操作：

```
if(root->type == T_DIR)
    |for addr:root->addr[]:
    |  if addr != 0:
    |     db = getDataBlock();
    |     for entrys **in** db **by** sizeof(entry):
    |        if inum != 0 and name != "." or ".." :
    |           findNewInode();
    |           Search(new_inode);
    |if root->addr[13]!=0:
    |  single = getIndirectBlock();
    |   for addr **in** single by 4:
    |      if addr != 0:
    |        db = getDataBlock();
    |        for entrys **in** db **by** sizeof(entry):
    |           if inum != 0 and name != "." or ".." :
    |              findNewInode();
    |              Search(new_inode);
```

> 3. 你是使用全局变量存储inode，还是使用局部变量或者是用kalloc和kfree?

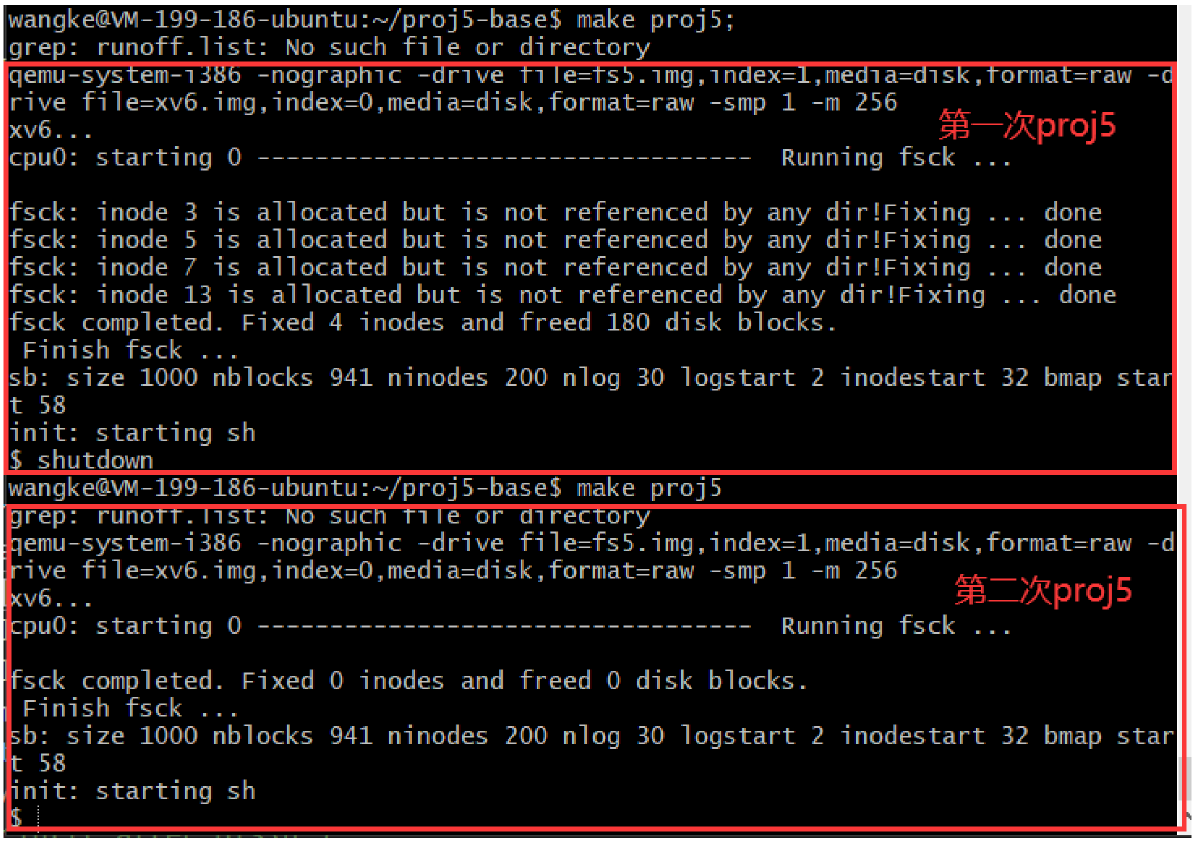
我使用的是局部变量存储inode，但事实上使用kalloc进行存储在代码上的区别不大，使用局部变量只是为了熟悉*和&操作符。

> 4. 你需要再实验报告中说明，make proj5和make proj5i分别修复了多少个inode，释放了多少个 disk blocks。

	修复inode数量	释放diskblocks数量
make proj5	4	180?
make proj5i	7	7

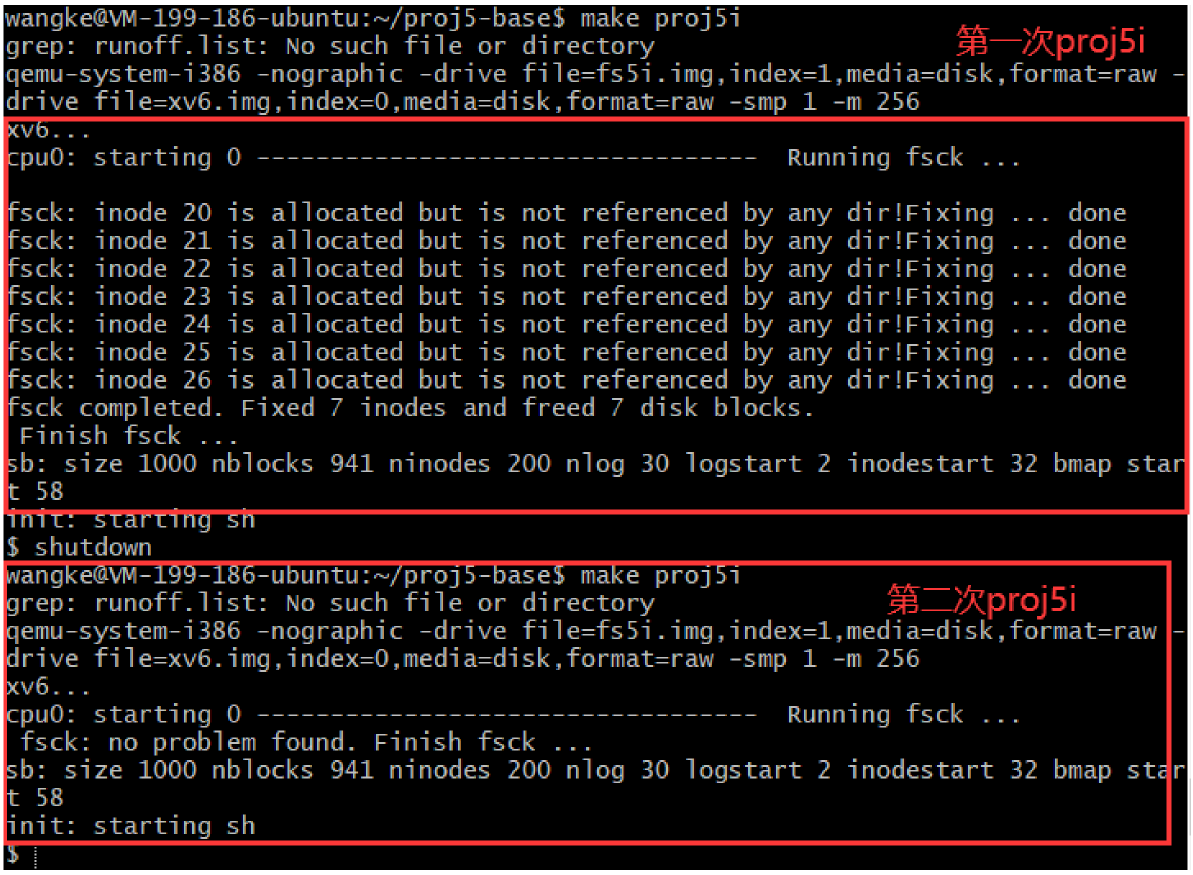
三、 实验结果

1. 在git bash中输入两次make proj5，按要求输出了指定内容：



```
wangke@VM-199-186-ubuntu:~/proj5-base$ make proj5;
grep: runoff.list: No such file or directory
qemu-system-i386 -nographic -drive file=fs5.img,index=1,media=disk,format=raw -d
rive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 256
xv6...
cpu0: starting 0 ----------------------------------  Running fsck ...

fsck: inode 3 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 5 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 7 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 13 is allocated but is not referenced by any dir!Fixing ... done
fsck completed. Fixed 4 inodes and freed 180 disk blocks.
 Finish fsck ...
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ shutdown
wangke@VM-199-186-ubuntu:~/proj5-base$ make proj5
grep: runoff.list: No such file or directory
qemu-system-i386 -nographic -drive file=fs5.img,index=1,media=disk,format=raw -d
rive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 256
xv6...
cpu0: starting 0 ----------------------------------  Running fsck ...

fsck completed. Fixed 0 inodes and freed 0 disk blocks.
 Finish fsck ...
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$
```

2. 在git bash中输入两次make proj5i，按要求输出了指定内容：



```
wangke@VM-199-186-ubuntu:~/proj5-base$ make proj5i
grep: runoff.list: No such file or directory
qemu-system-i386 -nographic -drive file=fs5i.img,index=1,media=disk,format=raw -
drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 256
xv6...
cpu0: starting 0 ----------------------------------  Running fsck ...

fsck: inode 20 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 21 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 22 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 23 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 24 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 25 is allocated but is not referenced by any dir!Fixing ... done
fsck: inode 26 is allocated but is not referenced by any dir!Fixing ... done
fsck completed. Fixed 7 inodes and freed 7 disk blocks.
 Finish fsck ...
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ shutdown
wangke@VM-199-186-ubuntu:~/proj5-base$ make proj5i
grep: runoff.list: No such file or directory
qemu-system-i386 -nographic -drive file=fs5i.img,index=1,media=disk,format=raw -
drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 256
xv6...
cpu0: starting 0 ----------------------------------  Running fsck ...
 fsck: no problem found. Finish fsck ...
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$
```

四、实验心得

本次实验中，所花的时间大多在资料的阅读还有代码的阅读上，实际编写代码的时间较少，思路比较清晰。后期debug花费时间比较长且遇到比较离奇的错误就是爆栈，事实上存放inode的缓冲区在栈中还是可以安全存放，但存放singleIndirectBlock的4KB则太大了，因此最后我把这部分空间开成了全局变量，但这部分也可以作为动态使用，因为使用次数并不多。而本次实验我也是为了方便一次性缓冲了8个inode，事实上如果一次读取一个inode的话，应该可以使得递归层数限制翻3翻。