Report for lab5, YiChen Mao

Report for lab5, YiChen Mao
- The File System
  - Exercise 1.
  - Exercise 2.
  - Exercise 3.
  - Exercise 4.
  - Exercise 5.
  - Exercise 6.
  - Exercise 7.
  - Exercise 8.
  - Exercise 9.
  - Exercise 10.
  - challenge

The File System

主要需要实现的功能，从磁盘中读取块到块缓存中，或从缓存中写回到磁盘中，分配磁盘块，将文件偏移映射到磁盘块中，在 IPC 接口时间读写打开。

Exercise 1.

Task. 修改 env.c 中 env_create 使得赋予环境 I/O 权限。

```
在这个 lab 中新加入了一个环境类型 ENV_TYPE_FS，只有环境类型为 ENV_TYPE_FS 才能赋予 I/O 权限，ENV_TYPE_USER 则不行。
具体修改只需要修改 eflags 对应位，在 mmu.h 中可以找到所有关于 eflags 的宏定义。
void
env_create(uint8_t *binary, enum EnvType type)
{
  ...
  if (type == ENV_TYPE_FS){
    e->env_tf.tf_eflags |= FL_IOPL_MASK;
  }
}
```

Question. 在切换环境的时候是否需要额外操作保证 I/O 权限正确性。

答：我认为是不需要的，因为在切换的过程中，会修改对应的 env 包括其中的 eflags 因此保证了正确。

Exercise 2.

Task. 实现 fs/bc.c 中函数 bc_pgfault 和 flush_block

```
磁盘的 page_fault 处理，将数据从磁盘读取到对应的内存
static void
bc_pgfault(struct UTrapframe *utf)
{
  ...
  addr = (void*)ROUNDDOWN(addr,BLKSIZE);
  if ((r = sys_page_alloc(0,addr,PTE_P|PTE_U|PTE_W))<0)
    panic("bc_pgfault: sys_page_alloc (add) is %x\n",addr);
  if ((r = ide_read(blockno * BLKSECTS,addr,BLKSECTS))<0)
    panic("bc_pgfault: ide_raed (blockno) is %x, (addr) is
%x\n",blockno,addr);

  if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] &
PTE_SYSCALL)) < 0)
    panic("in bc_pgfault, sys_page_map: %e", r);


  if (bitmap && block_is_free(blockno))
    panic("reading free block %08x\n", blockno);
}
```

```
将 cache 中的内容写回到磁盘之中去。
条件：当前页被映射了且被修改了，那么需要再写回保证数据正确性。
同时修改当前 cache 中的标记位，以防重复写回未修改的数据。
void
flush_block(void *addr)
{
  uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;

  if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
    panic("flush_block of bad va %08x", addr);

  int r;
```

```
  addr = (void*)ROUNDDOWN(addr,BLKSIZE);
  if (va_is_mapped(addr)&&va_is_dirty(addr)){
    if ((r = ide_write(blockno * BLKSECTS,addr,BLKSECTS))<0)
      panic("flush_block: ide_write (blockno) is %x, (addr) is
%x",blockno,addr);
    if ((r = sys_page_map(0,addr,0,addr,uvpt[PGNUM(addr)]&PTE_SYSCALL))
<0)
      panic("flush_block: sys_page_map (addr) is %x\n",addr);
  }
  return;
  panic("flush_block not implemented");
}
```

Exercise 3.

Task. 实现 fs/fs 函数 alloc_block()。

```
allock_block 找到一个空闲的磁盘块并分配它。
如果未能分配块就 返回 -E_NO_DISK

从 第 2 个块开始查找空闲块。这里用 block_is_free 判断一个块是否空间，维护了一个
bitset，第 i 位对应第 i 个块是否为空。
映射后修改 bitset 并写回到磁盘中。
int
alloc_block(void)
{
  // The bitmap consists of one or more blocks.  A single bitmap block
  // contains the in-use bits for BLKBITSIZE blocks.  There are
  // super->s_nblocks blocks in the disk altogether.

  // LAB 5: Your code here.
  int free_block = 2;
  while (free_block < super->s_nblocks &&
!block_is_free(free_block))free_block++;
  if (free_block == super->s_nblocks)return -E_NO_DISK;
  bitmap[free_block/32] &= ~(1<<(free_block%32));
  flush_block(&bitmap[free_block/32]);
  return free_block;

  panic("alloc_block not implemented");
  return -E_NO_DISK;
}
```

Exercise 4.

Task. 实现 file_block_walk 和 file_get_block。

```
file_block_walk 在文件 f 中找到第 filebno 个 block 所对应的磁盘块，将对应 slot 写到 ppdiskbno 中。

分几类情况讨论：
1. 首先 filebno <NDIRECT(10) ，此时直接从当前文件的 f_direct 即可获得对应的指针地址。
2. 其次 filebno > NDIRECT 此时要到 f_indirect 对应的磁盘块查找地址。如果 f_indirect 未分配则先分配一个磁盘页。建立映射，并返回新的磁盘页的指针地址。
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
{
  // LAB 5: Your code here.
  int r;
  if (filebno >= NDIRECT + NINDIRECT)
    return -E_INVAL;
  if (filebno >= NDIRECT && f->f_indirect == 0){
    if (!alloc)return -E_NOT_FOUND;
    if ((r = alloc_block()) < 0)return r;

    f->f_indirect = r;
    memset(diskaddr(r),0,BLKSIZE);
    flush_block(diskaddr(r));
  }
  *ppdiskbno = (filebno < NDIRECT)?&f->f_direct[filebno]:(uint32_t*)diskaddr(f->f_indirect)+(filebno - NDIRECT);
  return 0;
    panic("file_block_walk not implemented");
}
```

```
这里找到真正对应的磁盘块。 file_block_walk 只是找到文件对应块的磁盘块的地址。
相应的，如果该地址未分配对应的磁盘块则分配块，并建立地址和磁盘块之间的映射。
最后返回地址对应的磁盘块即可。
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
```

```
  int r;
  uint32_t *pdiskbno;
  if ((r = file_block_walk(f,filebno,&pdiskbno,true)) < 0)
    return r;
  // cprintf("file_get_block: find\n");
  if (*pdiskbno == 0){
    if ((r = alloc_block()) < 0)
      return r;
    *pdiskbno = r;
    flush_block(diskaddr(r));
  }
  // cprintf("file_get_block: find\n");
  *blk = diskaddr(*pdiskbno);
  return 0;
    panic("file_get_block not implemented");
}
```

Exercise 5.

Task. 实现 fs/ servlet .c 中 serve_read

```
首先 通过 openfile_lookup 找到对应的 OpenFile。
然后调用 file_read 将需要读取的内容放入到 ipc->readRet->ret_buf 中。
并修改 offset。

int
serve_read(envid_t envid, union Fsipc *ipc)
{
  struct Fsreq_read *req = &ipc->read;
  struct Fsret_read *ret = &ipc->readRet;

  if (debug)
    cprintf("serve_read %08x %08x %08x\n", envid, req->req_fileid, req->req_n);

  // Lab 5: Your code here:
  int r;
  struct OpenFile *o;
  if ((r = openfile_lookup(envid,req->req_fileid,&o)) < 0)
    return r;
  if ((r = file_read(o->o_file,ret->ret_buf,req->req_n,o->o_fd->fd_offset)) < 0)
```

```
    return r;
  o->o_fd->fd_offset += r;
  return r;
}

```

Exercise 6.

Task. 实现 fs/serv.c 中的 serve_write 和 lib/file.c 中的 devfile_write。

```
和 serve_read 相似，找到 OpenFile 并写入，最后修改 offset。

int
serve_write(envid_t envid, struct Fsreq_write *req)
{
  if (debug)
    cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid,
req->req_n);

  // LAB 5: Your code here.
  int r;
  struct OpenFile *o;
  // cprintf("server_write: %x %x %x %x\n",envid,req->req_fileid,req-
>req_buf,&o);
  if ((r = openfile_lookup(envid,req->req_fileid,&o)) < 0)
    return r;
  if ((r = file_write(o->o_file,req->req_buf,(req->req_n>PGSIZE)?
PGSIZE:req->req_n,o->o_fd->fd_offset)) < 0)
    return r;
  // cprintf("find\n");
  o->o_fd->fd_offset += r;
  // cprintf("serve_write fd_offset: %x\n",o->o_fd->fd_offset);
  return r;
  panic("serve_write not implemented");
}

```

```
写入至多 n byte 从 buf 到 fd. 可以 仿照 devfile_read。
static ssize_t
devfile_write(struct Fd *fd, const void *buf, size_t n)
```

```
{
  // Make an FSREQ_WRITE request to the file system server.  Be
  // careful: fsipcbuf.write.req_buf is only so large, but
  // remember that write is always allowed to write *fewer*
  // bytes than requested.
  // LAB 5: Your code here
  int r;
  fsipcbuf.write.req_fileid = fd->fd_file.id;
  fsipcbuf.write.req_n = (n>sizeof(fsipcbuf.write.req_buf))?
sizeof(fsipcbuf.write.req_buf):n;
  memcpy(fsipcbuf.write.req_buf,buf,fsipcbuf.write.req_n);
  // cprintf("write\n");
  if ((r = fsipc(FSREQ_WRITE,NULL)) < 0)
    return r;
  // cprintf("r:%x\n",r);
  assert(r <= n);
  assert(r <= PGSIZE);
  // cprintf("r:%x\n",r);
  return r;
  panic("devfile_write not implemented");
}
```

Exercise 7.

Task. 实现 sys_env_set_trapframe 函数。

```
实现从文件中加载一个子进程，这里只需要实现 设置 tf 即可。
由于之前都已经处理好，需要检查用户空间是否合法，然后设置 eflag 即可。
static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
  int r;
  struct Env *e;
  if ((r = envid2env(envid,&e,true)) < 0)
    return r;

  user_mem_assert(e,(const void*)tf,sizeof(struct Trapframe),PTE_U);
  tf->tf_eflags = (tf->tf_eflags | FL_IF) & ~FL_IOPL_MASK;
  tf->tf_cs |= 3;
  e->env_tf = *tf;
  return 0;
  panic("sys_env_set_trapframe not implemented");
```

```
}
```

Exercise 8.

Task. 处理 duppage() 中的PTE_SHARE 页，如之前所述，采用直接复制映射的方式。对于 copy_shared_page

```
在 duppage() 中添加对于 PTE_SHARE 位处理，直接映射，采用 PTE_SYSCALL 作为 perm。
if (pte&PTE_SHARE){
    if ((r = sys_page_map(envid_parent,(void*)va,envid, (void*)va,PTE_SYSCALL))<0)
      return r;
    return 0;
  }
```

```
枚举用户空间所在页，对于设置 PTE_SHARE 的页面，同样直接映射，操用 PTE_SYSCALL 作为 perm。
static int
copy_shared_pages(envid_t child)
{
  // LAB 5: Your code here.
  int r;
  for (uintptr_t va = 0; va < UTOP; va+=PGSIZE){
    if ((uvpd[PDX(va)] & PTE_P)&&(uvpt[PGNUM(va)] & PTE_P)&&(uvpt[PGNUM(va)]&PTE_U)&&(uvpt[PGNUM(va)]&PTE_SHARE)){
      if ((r = sys_page_map(0,(void*)va,child,(void*)va,PTE_SYSCALL))<0);
    }
  }
  return 0;
}

```

Exercise 9.

Task. 调用 kern/trap.c，处理 IRQ_OFFSET+IRQ_KBD 和IRQ_OFFSET+IRQ_SERIAL。

```
kbd_intr() 和 serial_intr() 已经实现好了，因此我们只需要根据 tf_trapno 调用即可。
switch (tf->tf_trapno){
    case IRQ_OFFSET + IRQ_TIMER: {
      lapic_eoi();
          sched_yield();
          return;
    }
    case IRQ_OFFSET + IRQ_KBD:{
      kbd_intr();
      return;
    }
    case IRQ_OFFSET + IRQ_SERIAL: {
      serial_intr();
      return;
    }
  }
```

Exercise 10.

Task. 实现 shell 对于 < 符号的重定向。

```
< 是对于 读文件，因此参数只需要 O_RDONLY。
如果 fd 不为 0，则将 fd dup 到 0并关闭原 fd。
      case '<': // Input redirection
      // Grab the filename from the argument list
      if (gettoken(0, &t) != 'w') {
        cprintf("syntax error: < not followed by word\n");
        exit();
      }
      // Open 't' for reading as file descriptor 0
      // (which environments use as standard input).
      // We can't open a file onto a particular descriptor,
      // so open the file as 'fd',
      // then check whether 'fd' is 0.
      // If not, dup 'fd' onto file descriptor 0,
      // then close the original 'fd'.

      // LAB 5: Your code here.
      if ((fd = open(t,O_RDONLY))<0){
```

```
        cprintf("fd open error");
        exit();
      }
      if (fd){
        dup(fd,0);
        close(fd);
      }
      break;
```

challenge

Challenge! The block cache has no eviction policy. Once a block gets faulted in to it, it never gets removed and will remain in memory forevermore. Add eviction to the buffer cache. Using the PTE_A "accessed" bits in the page tables, which the hardware sets on any access to a page, you can track approximate usage of disk blocks without the need to modify every place in the code that accesses the disk map region. Be careful with dirty blocks.

```
#define NBUF 64

对于 cache 构建一个双向链表结构，unused 表示空闲的 block， used 表示正在使用的 block。
typedef struct MyLink{
    struct MyLink *p,*s;
  void *addr;
}MyLink;
void MyLink_init(MyLink *p){p->p=p->s=p;p->addr=0;}
MyLink* MyLink_delete(MyLink *p){p->p->s=p->s;p->s->p=p->p;p->p=p->s=p;return p;}
void MyLink_insert(MyLink *p,MyLink *q,void *addr){
  //cprintf("link %x %x\n",p,q);
  q->p=p;q->s=p->s;p->s=q->s->p=q;q->addr=addr;
}
MyLink buf[NBUF],unused,used;


int nbuf=0,t=0;

将一个 block 移出 cache。
```

如果该块被修改过，那么要将其写回到磁盘中，并且取消内存映射，并将其从使用链表中删除并加入到空闲链表。

```
int buf_remove(MyLink *l){
  // cprintf("remove %x %x\n",l,l->addr);
  int r;
  if (uvpt[PGNUM(l->addr)]&PTE_D)flush_block(l->addr);
  if (r=sys_page_unmap(0,l->addr),r<0)
    return r;
  nbuf--;
  // cprintf("fffff  %x %x\n",unused.p,l);
  MyLink_insert(unused.p,MyLink_delete(l),0);
  return 0;
}
```

但 cache 满时，需要将一个 block 移除。

选择的策略是 根据是否最近访问过，该标记由 PTE_A 维护，如果一个 block 最近未访问过，则将其删除，否则删除一个最早加入 cache 的block。

```
int buf_evict(void){
  // cprintf("%x %x\n",&used,used.s);
  int r;
  for (MyLink *l=used.s;l!=&used;){
    void *addr=l->addr;
    if (!(uvpt[PGNUM(addr)]&PTE_A)){
      //cprintf("%x %x %x %x\n",addr,l,buf,&used);
      MyLink* tmp = l->s;
      if((r=buf_remove(l))&&r<0)
        return r;
      l = tmp;
    }else l = l->s;
  }
  if (nbuf==NBUF&&(r=buf_remove(used.s))&&r<0)
    return r;
  return 0;
}
```

输出函数，能够将所有当前在 cache 中 的块输出。

同时输出空间利用率，这里将 PTE_A 标记为 1 的认为是当前使用的 block，而在 cache 中而而未标记 PTE_A 的则认为是未使用 void buf_print_used(void){

```
  static int u=0,un=0;
  for (MyLink *l = used.s;l!=&used;l=l->s){
    cprintf("%x ",l->addr);
    if (uvpt[PGNUM(l->addr)]&PTE_A)u++;
    else un++;
  }cprintf("\n");
  cprintf("%d %d %d\n",u,u+un,(int)((float)u/(u+un)*1000000000));
```

```
}
从 cache 中分配一个 block。首先忽略 前两个 block，因为他们不能被换出。
如果 没有空闲 则从 buf 中移除一个。
然后从空闲删除一个并加入到使用链表中。存储对应的地址。
int buf_alloc(void *addr){
  // cprintf("alloc %x\n",addr);
  int r;
  if (addr<diskaddr(2))return 0;
  if ((nbuf==NBUF)&&(r=buf_evict())&&r<0)
    return r;
  nbuf++;
  //cprintf("important %x %x\n",&used,&unused);
  MyLink * l = MyLink_delete(unused.s);
  MyLink_insert(used.p,l,addr);
  return 0;
}
初始化，unused 链表中是所有 cache ，而 used 链表为空。
void buf_init(void){
  MyLink_init(&unused);MyLink_init(&used);
  for (int i=0;i<NBUF;++i) {
    MyLink_init(&buf[i]);
    MyLink_insert(unused.p,&buf[i],0);
  }
  /*
  cprintf("init:\n");
  for (MyLink *l=unused.s;l!=&unused;l=l->s){
    cprintf("%x ",l);
  }
  cprintf("\n");*/
}
block 访问，每次对于 block 的访问都会导致 visit 次数加 1，当达到一定次数之后，回将 PTE_A 标记清空。
int buf_visit(void){
  int r;
  if (++t<NBUF)return 0;t=0;
  int cused = 0,cunused = 0;
  for (MyLink *l=used.s;l!=&used;l=l->s)cused++;
  for (MyLink *l=unused.s;l!=&unused;l=l->s)cunused++;
  // cprintf("%x %x\n",cused,(cused+cunused));
  for (MyLink *l=used.s;l!=&used;l=l->s){
    void *addr=l->addr;
    if (uvpt[PGNUM(addr)]&PTE_A){
      if (uvpt[PGNUM(addr)]&PTE_D)flush_block(addr);
```

```
      if ((r=sys_page_map(0,addr,0,addr,uvpt[PGNUM(addr)]&PTE_SYSCALL))&&r<0)
        return r;
    }
  }
  return 0;
}
这里是对于 block_free 的时候需要将对应地址的 block 从 cache 中删除。
int buf_delete(void*addr){
  for (MyLink *l=used.s;l!=&used;l=l->s){
    if (l->addr==addr){
      MyLink_insert(unused.p,MyLink_delete(l),0);
      return 0;
    }
  }
  return -1;
}

```

```
在 bc_init() 中 调用 buf_init() 初始化。
void
bc_init(void)
{
  struct Super super;
  set_pgfault_handler(bc_pgfault);
  check_bc();
#ifdef BUF_CACHE_OPEN
  buf_init();
#endif
  // cache the super block by reading it once
  memmove(&super, diskaddr(1), sizeof super);
}

在 bc_pgfault 中会将一个块移入内存中，因此需要在 cache 中加入该 block
static void
bc_pgfault(struct UTrapframe *utf)
{
...
#ifdef BUF_CACHE_OPEN
  if ((r = buf_alloc(addr))&&r < 0)
    panic("in bc_pgfault, buf_alloc: %e", r);
#endif
```

```
24
25  }
26
27  free_block 中需要将其从 cache 中删除。
28  void
29  free_block(uint32_t blockno)
30  {
31    // Blockno zero is the null pointer of block numbers.
32    if (blockno == 0)
33      panic("attempt to free zero block");
34    bitmap[blockno/32] |= 1<<(blockno%32);
35
36  #ifdef BUF_CACHE_OPEN
37    int r;
38    if ((r = sys_page_unmap(0, diskaddr(blockno)))&&r < 0)
39      panic("free_block: %e", r);
40    extern int buf_delete(void*);
41    if ((r=buf_delete(diskaddr(blockno)))&&r<0)
42      panic("free_block: %e", r);
43
44  #endif
45    //cprintf("free_block %x\n",blockno);
46  }
```

测试。用 testfile , testpteshare 和 testshell 文件测试（因为这三个测试的访问 block 较多）

测试数据	NBUF=64	NBUF=32	NBUF=16	NBUF=8	NBUF=4
testfile	37/0/0.3523	69/51/0.4277	69/53/0.5031	70/64/0.6885	214/210/0.9199
testpteshare	11/0/0.8440	11/0/0.8440	11/0/0.8440	13/9/0.8119	18/14/0.8333
testshell	34/0/0.1214	70/44/0.1397	89/74/0.1969	146/139/0.4501	289/285/0.6945

表中第一项数据为cache 中 分配block 次数，第二项为 删除block 的次数，第三项可以认为是空间利用率。

计算方式如下：

用 PTE_A 表示该块是否最近使用过，对于一个随机时刻，记录分配块中最近使用和未使用的块的个数。多次采样取平均值。（这里的采样是对于每次访问就采一次样）

总体而言，利用率随着 BUF 减小而增大，而切换次数也逐渐增多。

（testpteshare 行利用率局部的降低是因为我的程序 NBUF 越少，我刷新 PTE_A 的频率就会越块，因此较小的 BUF 在 刷新 PTE_A 之后导致 used 中块大多数变成最近未使用了。）