Report for lab5

Hongyu Wen, 1800013069

> All exercises finished.
>
> All questions answered.
>
> 1 Challenges completed.

Grade

```
internal FS tests [fs/test.c]: OK (1.4s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
  alloc_block: OK
  file_open: OK
  file_get_block: OK
  file_flush/file_truncate/file rewrite: OK
testfile: OK (1.0s)
  serve_open/file_stat/file_close: OK
  file_read: OK
  file_write: OK
  file_read after file_write: OK
  open: OK
  large file: OK
spawn via spawnhello: OK (1.7s)
Protection I/O space: OK (2.4s)
PTE_SHARE [testpteshare]: OK (1.4s)
PTE_SHARE [testfdsharing]: OK (1.3s)
start the shell [icode]: Timeout! OK (31.5s)
testshell: OK (2.5s)
    (Old jos.out.testshell failure log removed)
primespipe: OK (6.4s)
Score: 150/150
```

On-Disk File System Structure

- Our file system will not use inodes at all and instead will simply store all of a file's (or sub-directory's) meta-data within the (one and only) directory entry describing that file.
- Our file system does allow user environments to read directory meta-data directly (e.g., with read), which means that user environments can perform directory scanning operations themselves (e.g., to implement the ls program) rather than having to rely on additional special calls to the file system. The disadvantage of this approach to directory scanning, and the reason most modern UNIX variants discourage it, is that it makes application programs dependent on the format of directory meta-data, making it difficult to change the file system's internal layout without changing or at least recompiling application programs as well.
- Our file system will use a block size of 4096 bytes, conveniently matching the processor's page size.
- Our file system will have exactly one superblock, which will always be at block 1 on the disk. Its layout is defined by struct Super in `inc/fs.h`.
- As mentioned above, we do not have inodes, so this meta-data is stored in a directory entry on disk. Unlike in most "real" file systems, for simplicity we will use this one File structure to represent file meta-data as it

appears both on disk and in memory.
- Only support single-indirect blocks.

Disk Access

- Instead of taking the conventional "monolithic" operating system strategy of adding an IDE disk driver to the kernel along with the necessary system calls to allow the file system to access it, we instead implement the IDE disk driver as part of the **user-level file system environment**.
- It is easy to implement disk access in user space this way as long as we rely on polling, "programmed I/O" (PIO)-based disk access and do not use disk interrupts.
- The x86 processor uses the IOPL bits in the EFLAGS register to determine whether protected-mode code is allowed to perform special device I/O instructions such as the IN and OUT instructions.

Exercise 1

```
void
env_create(uint8_t *binary, enum EnvType type)
{
        ...
    if (type == ENV_TYPE_FS) {
        e->env_tf.tf_eflags |= FL_IOPL_MASK;
    }
}
```

Run `make grade` we get:

```
internal FS tests [fs/test.c]: OK (1.2s)
  fs i/o: OK
```

Questions

1. Do you have to do anything else to ensure that this I/O privilege setting is saved and restored properly when you subsequently switch from one environment to another? Why?

No. Because all the registers will be set well automatically.

The Block Cache

Exercise 2

`bc_pgfoult`:

```
    addr = ROUNDDOWN(addr, PGSIZE);
    if ((r = sys_page_alloc(0, addr, PTE_U | PTE_W | PTE_P)) < 0)
        panic("in bc_pgfault, sys_page_alloc: %e", r);
    if ((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
        panic("in bc_pgfault, ide_read: %e", r);
```

`flush_block`:

```
    addr = ROUNDDOWN(addr, PGSIZE);
    if (!va_is_mapped(addr) || !va_is_dirty(addr))
        return;

    int r;
    if ((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
        panic("in flush_block, ide_write: %e", r);

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

Run `make grade` we get:

```
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
```

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.

Simply remove the blocks without `PTE_A`.

```
void
evict_block(void *addr)
{
    int r;
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;

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

    // ignore boot sector, super block and bitmap block.
    if(blockno <= 2 || !va_is_mapped(addr))
        return;

        // deal with dirty block
    if(uvpt[PGNUM(addr)] & PTE_D) {
        flush_block(addr);
    }

    // enture that addr was page-aligned
    addr = (void *)ROUNDDOWN(addr, PGSIZE);

        // evict this block to disk.
    if((r = sys_page_unmap(0, addr)) < 0){
        panic("evict_block_force:sys_page_unmap:%e", r);
```

```
    }
}

// This function will only evict those block
// loaded into memory but has never been accessed.
void
block_evict_policy()
{
    uint32_t blockno, nblocks = DISKSIZE / BLKSIZE;
    for(blockno = 3; blockno < nblocks; ++blockno) {
        if (!(uvpt[PGNUM(diskaddr(blockno))] & PTE_A)){
            evict_block(diskaddr(blockno));
            cprintf("block with blockno %d was evicted.\n");
        }
    }
}
```

The Block Bitmap

Exercise 3

```
int
alloc_block(void)
{
    int offset = 2; // 1 for super and 2 for bitmap
    for (uint32_t i = 0; i < super->s_nblocks; ++i) {
        if (block_is_free(i)) {
            bitmap[i/32] &= ~(1 << (i%32));

            // flush the bitmap
            flush_block(diskaddr(offset + i / 32 / NINDIRECT));
            return i;
        }
    }
    return -E_NO_DISK;
}
```

Now we can pass `alloc_block`.

File Operations

Exercise 4

```
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool
alloc)
{
       // LAB 5: Your code here.
       /* panic("file_block_walk not implemented"); */

    if (filebno >= NDIRECT + NINDIRECT) {
        // block number too large
        return -E_INVAL;
    }
```

```
    int bn;
    uint32_t *indir_addr;
    if (filebno < NDIRECT) {
        *ppdiskbno = &(f->f_direct[filebno]);
        // if *ppdiskbno = 0, which means the block has not been allocated
        // the block will be allocated in file_get_block
        // but we need to allocate f_indirect in this function
    } else {
        if (f->f_indirect) {
            indir_addr = diskaddr(f->f_indirect);
            *ppdiskbno = &indir_addr[filebno - NDIRECT];
            } else {
            if (alloc == 0) {
                return -E_NOT_FOUND;
            } else if ((bn = alloc_block()) < 0) {
                return bn;
            }

            f->f_indirect = bn;
            flush_block(diskaddr(bn));
            indir_addr = diskaddr(f->f_indirect);
            *ppdiskbno = &indir_addr[filebno - NDIRECT];

        }
    }

    return 0;
}
```

```
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
       // LAB 5: Your code here.
       /* panic("file_get_block not implemented"); */

    int r;
    uint32_t *ppdiskbno;
    if ((r = file_block_walk(f, filebno, &ppdiskbno, 1)) < 0) {
        return r;
    }

    int bn;
    if (*ppdiskbno == 0) {
        // not alloc yet
        if ((bn = alloc_block()) < 0) {
            return bn;
        }
        *ppdiskbno = bn;
        flush_block(diskaddr(bn));
    }
    *blk = diskaddr(*ppdiskbno);
    return 0;
}
```

Run `make grade`:

```
internal FS tests [fs/test.c]: OK (2.1s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
  alloc_block: OK
  file_open: OK
  file_get_block: OK
  file_flush/file_truncate/file rewrite: OK
testfile: OK (0.8s)
```

The file system interface

Exercise 5

```
int
serve_read(envid_t envid, union Fsipc *ipc)
{
        ...

    // Lab 5: Your code here:
    struct OpenFile *o;
    int r;
    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)
        // file_read in fs.c
        return r;
    o->o_fd->fd_offset += r;
    return r; // warp of file_read
}
```

```
  serve_open/file_stat/file_close: OK
  file_read: OK
```

Exercise 6

```
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.
    /* panic("serve_write not implemented"); */
    struct OpenFile *o;
    int r;

    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, o->o_fd->fd_offset))
< 0)
```

```
        return r;
    o->o_fd->fd_offset += r;
    return r;
    // the same as serve_read
}
```

```
static ssize_t
devfile_write(struct Fd *fd, const void *buf, size_t n)
{
    int r;

    fsipcbuf.write.req_fileid = fd->fd_file.id;
    if (n > sizeof(fsipcbuf.write.req_buf))
        n = sizeof(fsipcbuf.write.req_buf);
        // write fewer bytes
    fsipcbuf.write.req_n = n;
    memmove(fsipcbuf.write.req_buf, buf, n);
    if ((r = fsipc(FSREQ_WRITE, NULL)) < 0)
        return r;
    return r;
}
```

Spawning Processes

Exercise 7

```
static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
    // LAB 5: Your code here.
    // Remember to check whether the user has supplied us with a good
    // address!
    /* panic("sys_env_set_trapframe not implemented"); */

    int r;
    struct Env *e;
    if ((r = envid2env(envid, &e, 1)) < 0) {
        return r;
    }

    tf->tf_eflags = FL_IF; // Enable interrupts
    tf->tf_eflags &= ~FL_IOPL_MASK; // No I/O priority
    tf->tf_cs = GD_UT | 3; // User mode
    e->env_tf = *tf;
    return 0;
}
```

```
spawn via spawnhello: OK (1.7s)
    (Old jos.out.spawn failure log removed)
Protection I/O space: OK (2.2s)
```

Exercise 8

```
static int
duppage(envid_t envid, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    void *addr = (void *) (pn * PGSIZE);
    if (uvpt[pn] & PTE_SHARE) { // shared page
        if ((r = sys_page_map(0, addr, envid, addr, PTE_SYSCALL)) < 0)
            panic("sys_page_map: %e", r);
    } else if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW)) {
        if ((r = sys_page_map(0, addr, envid, addr, PTE_COW | PTE_U | PTE_P)) <
0)
            panic("sys_page_map: %e", r);
        if ((r = sys_page_map(0, addr, 0, addr, PTE_COW | PTE_U | PTE_P)) < 0)
            panic("sys_page_map: %e", r);

    } else { // read-only page
        if ((r = sys_page_map(0, addr, envid, addr, PTE_U | PTE_P)) < 0)
            panic("sys_page_map: %e", r);
    }
    return 0;
}
```

```
static int
copy_shared_pages(envid_t child)
{
    // LAB 5: Your code here.

    int r;
    uintptr_t addr;
    for (addr = 0; addr < UTOP; addr += PGSIZE) {
        if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P) &&
            (uvpt[PGNUM(addr)] & PTE_U) && (uvpt[PGNUM(addr)] & PTE_SHARE)) {
            r = sys_page_map(0, (void*)addr, child, (void*)addr,
(uvpt[PGNUM(addr)] & PTE_SYSCALL));
            if (r) return r;
        }
    }
    return 0;
}
```

The Shell

Exercise 9

```
    if (tf->tf_trapno == IRQ_OFFSET + IRQ_KBD) {
                kbd_intr();
                return;
        }
        if (tf->tf_trapno == IRQ_OFFSET + IRQ_SERIAL) {
                serial_intr();
                return;
        }
```

Exercise 10

```
            if ((fd = open(t, O_RDONLY)) < 0) {
                                cprintf("open %s for read: %e", t, fd);
                                exit();
                        }
            if (fd != 0) { // standard input
                dup(fd, 0);
                close(fd);
            }
```

Run `make grade` and we have:

```
internal FS tests [fs/test.c]: OK (1.4s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
  alloc_block: OK
  file_open: OK
  file_get_block: OK
  file_flush/file_truncate/file rewrite: OK
testfile: OK (1.0s)
  serve_open/file_stat/file_close: OK
  file_read: OK
  file_write: OK
  file_read after file_write: OK
  open: OK
  large file: OK
spawn via spawnhello: OK (1.7s)
Protection I/O space: OK (2.4s)
PTE_SHARE [testpteshare]: OK (1.4s)
PTE_SHARE [testfdsharing]: OK (1.3s)
start the shell [icode]: Timeout! OK (31.5s)
testshell: OK (2.5s)
    (Old jos.out.testshell failure log removed)
primespipe: OK (6.4s)
Score: 150/150
```