# JOS Lab5

**Jen-tse Huang, 1500017836**

## Exercise 1

First, look at `init.c`:

```
// Start fs.
ENV_CREATE(fs_fs, ENV_TYPE_FS);
```

If `EnvType type == ENV_TYPE_FS`, give it I/O privilege. Modify `env.c`:

```
if (type == ENV_TYPE_FS) {
    e->env_tf.tf_eflags |= FL_IOPL_MASK;
}
```

After `make grade`, we can see:

```
internal FS tests [fs/test.c]: OK (1.0s)
  fs i/o: OK
```

- Q: 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?

  A: No, because `iret` will automatically save the registers.

## Exercise 2

- `bc_pgfault()`:

Notice that we are in user mode. We cannot use `page_alloc()` directly. Instead, we can use the `sys_page_alloc()` in `inc/lib.h`.

After read the codes in `ide.c`, we can find that the function `ide_read()` need the beginning sector's number and the total number of sectors needed. So modify `bc_pgfault()`:

```
addr = ROUNDDOWN(addr, BLKSIZE);
if (sys_page_alloc(0, addr, PTE_P | PTE_W | PTE_U) < 0) {
    panic("bc_fault:sys_page_alloc failed!");
}
if (ide_read(blockno * BLKSECTS, addr, BLKSECTS) < 0) {
    panic("bc_pgfault: ide_read failed!");
}
```

- `flush_block()`

Remember to use the functions in `bc.c`

```
addr = ROUNDDOWN(addr, BLKSIZE);
if (!va_is_mapped(addr) || !va_is_dirty(addr)) {
   return ;
}
if (ide_write(blockno * BLKSECTS, addr, BLKSECTS) < 0) {
   panic("flush_block: ide_write failed!");
}
if (sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL) < 0) {
   panic("flush_block: sys_page_map failed!");
}
```

After `make grade`, we can see:

```
check_bc: OK
check_super: OK
check_bitmap: OK
```

## Challenge 2

- `alloc_block()`

In this function, if it runs out of space, it simply returns an error value. I add a eviction policy when this happens. First write back the dirty pages and clear the dirty bit. Then unmap the not-accessing pages and free those blocks. Finally invoke `alloc_block()` itself.

```
for (size_t blockno = 0; blockno < super->s_nblocks; ++ blockno) {
   if (block_is_free(blockno)) {
      bitmap[blockno / 32] &= ~(1 << (blockno % 32));
      flush_block(bitmap);
      return blockno;
   }
}
// No space
for (uint32_t va = DISKMAP; va < DISKMAP + DISKMAPMAX; va += BLKSIZE) {
   if ((uvpd[PDX(va)] & PTE_P) && (uvpt[PGNUM(va)] & PTE_P)) {
      if (uvpt[PGNUM(va)] & PTE_D) {
         flush_block((void *)va);
         sys_page_map(0, (void *)va, 0, (void *)va, (uvpt[PGNUM(va)] & ~PTE_D));
      }
      if ((uvpt[PGNUM(va)] & PTE_A) == 0) {
         sys_page_unmap(0, (void *)va);
         free_block((va - DISKMAP) / BLKSIZE);
      }
   }
}
return alloc_block();
```

## Exercise 3

Use `block_is_free()`:

```
for (size_t blockno = 0; blockno < super->s_nblocks; ++ blockno) {
    if (block_is_free(blockno)) {
        bitmap[blockno / 32] &= ~(1 << (blockno % 32));
        flush_block(bitmap);
        return blockno;
    }
}
return -E_NO_DISK;
```

After `make grade`, we can see:

```
alloc_block: OK
```

## Exercise 4

- `file_block_walk()`

If we need to allocate a indirect block, remember to initialize it and `flush_block()` it.

```
if (filebno < NDIRECT) {
    // No indirect block
    *ppdiskbno = &(f->f_direct[filebno]);
} else if (filebno < NDIRECT + NINDIRECT) {
    // Need indirect block
    if (f->f_indirect == 0) {
        // Not found
        if (alloc) {
            int blockno = alloc_block();
            if (blockno < 0) {
                return -E_NO_DISK;
            }
            memset(diskaddr(blockno), 0, BLKSIZE);
            flush_block(diskaddr(blockno));
            f->f_indirect = blockno;
            *ppdiskbno = &(((uint32_t *)diskaddr(blockno))[filebno - NDIRECT]);
        } else {
            // alloc not set
            return -E_NOT_FOUND;
        }
    } else {
        *ppdiskbno = &(((uint32_t *)diskaddr(f->f_indirect))[filebno - NDIRECT]);
    }
} else {
    // filebno is out of range
    return -E_INVAL;
}
return 0;
```

- `file_get_block()`

Much like the function above:

```
    // LAB 5: Your code here.
    if (filebno >= NDIRECT + NINDIRECT) {
        // filebno is out of range
        return -E_INVAL;
    }
    uint32_t *ppdiskbno = NULL;
    int r = file_block_walk(f, filebno, &ppdiskbno, true);
    if (r < 0) {
        return r;
    }
    if (*ppdiskbno == 0) {
        // Need to allocate
        int blockno = alloc_block();
        if (blockno < 0) {
            return -E_NO_DISK;
        }
        memset(diskaddr(blockno), 0, BLKSIZE);
        flush_block(diskaddr(blockno));
        *ppdiskbno = blockno;
    }
    *blk = diskaddr(*ppdiskbno);
    return 0;
```

After `make grade`, we can see:

```
  file_open: OK
  file_get_block: OK
  file_flush/file_truncate/file rewrite: OK
testfile: OK (1.0s)
```

## Exercise 5

- `serve_read()`

First, as what `serve_set_size()` does, use `openfile_lookup()` to find the file. Then invoke relevant function, which is (in this case) `file_read()`.

Remember to maintain the `OpenFile.o_fd->fd_offset` because it records the current reading position of this file.

```
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,
                   MIN(req->req_n, BLKSIZE), o->o_fd->fd_offset)) < 0) {
   return r;
}
o->o_fd->fd_offset += r;
return r;
```

After `make grade`, we can see:

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

**In this exercise, I found there is some bugs in Lab 4. Please refer to the codes.**

## Exercise 6

- `serve_write()`

The same as `serve_read()` above:

```
int r;
struct OpenFile *o;
if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0) {
   return r;
}
if ((r = file_write(o->o_file, req->req_buf,
                    MIN(req->req_n, BLKSIZE), o->o_fd->fd_offset)) < 0) {
   return r;
}
o->o_fd->fd_offset += r;
return r;
```

- `devfile_write()`

Read the function `devfile_read()` above as a template.

Remember that `fsipcbuf.write.req_buf` is smaller than `buf`:

```
int r = 0;
void *p = (void *)buf;
fsipcbuf.write.req_fileid = fd->fd_file.id;

while (n > 0) {
   fsipcbuf.write.req_n = MIN(n, sizeof(fsipcbuf.write.req_buf));

   memmove(fsipcbuf.write.req_buf, p, fsipcbuf.write.req_n);
```

```
        if ((r = fsipc(FSREQ_WRITE, NULL)) < 0) {
            return r;
        }
        n -= r, p += r;
    }

    return r;
```

After `make grade`, we can see:

```
file_write: OK
file_read after file_write: OK
open: OK
large file: OK
```

## Exercise 7

- `syscall()`

First, modify `syscall()` to dispatch the system call:

```
    case SYS_env_set_trapframe:
        return sys_env_set_trapframe((envid_t)a1, (struct Trapframe *)a2);
```

- `sys_env_set_trapframe()`

Remember to set some restrictions:

```
    struct Env *e;
    if (envid2env(envid, &e, 1) < 0) {
        return -E_BAD_ENV;
    }
    e->env_tf = *tf;
    // CPL 3
    e->env_tf.tf_cs |= 3;
    // Enable interrupts
    e->env_tf.tf_eflags |= FL_IF;
    // Set IOPL zero
    e->env_tf.tf_eflags &= ~FL_IOPL_MASK;
    return 0;
```

Modify `i386_init()` in `kern/init.c`:

```
    ENV_CREATE(user_spawnhello, ENV_TYPE_USER);
    //ENV_CREATE(user_icode, ENV_TYPE_USER);
```

After `make qemu`, we can see:

```
hello, world
i am environment 00001002
No runnable environments in the system!
```

We successfully run the `/hello` from disk. Remember to change `i386_init()` in `kern/init.c` back.

After `make grade`, we can see:

```
spawn via spawnhello: OK (1.5s)
Protection I/O space: OK (2.1s)
```

## Exercise 8

We want to map pages with `PTE_SHARE` to children directly.

- `duppage()`

Add this in the beginning of the function:

```
if (uvpt[pn] & PTE_SHARE) {
    sys_page_map(0, addr, envid, addr, PTE_SYSCALL);
    return 0;
}
```

- `copy_shared_pages()`

Just as what `fork()` in `lib/fork.c` does:

```
for (uint32_t addr = UTEXT; addr < UTOP; addr += PGSIZE) {
    // For parent's exception stack
    if (addr != UXSTACKTOP - PGSIZE) {
        if (((uvpd[PDX(addr)] & PTE_P) != 0) &&
            ((~uvpt[PGNUM(addr)] & (PTE_P | PTE_U | PTE_SHARE)) == 0)) {
            sys_page_map(0, (void *)addr, child, (void *)addr,
                         (uvpt[PGNUM(addr)] & PTE_SYSCALL));
        }
    }
}
return 0;
```

After `make run-testpteshare`:

```
fork handles PTE_SHARE right
spawn handles PTE_SHARE right
```

After `make run-testfdsharing`:

```
read in child succeeded
read in parent succeeded
```

## Exercise 9

- `trap_dispatch()`

Easy to implement, remember to return:

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

After `make run-testkbd`, I type in both the console and the graphical window:

```
Type a line: penguin
penguin
Type a line: penguin
penguin
```

## Exercise 10

After `make run-icode`, I type in several commands:

```
$ echo hello world | cat
hello world
$ cat lorem |cat
Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit
in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim
id est laborum.
$ cat lorem |num
    1 Lorem ipsum dolor sit amet, consectetur
    2 adipisicing elit, sed do eiusmod tempor
    3 incididunt ut labore et dolore magna

    4 aliqua. Ut enim ad minim veniam, quis
```

```
 5 nostrud exercitation ullamco laboris
 6 nisi ut aliquip ex ea commodo consequat.
 7 Duis aute irure dolor in reprehenderit
 8 in voluptate velit esse cillum dolore eu
 9 fugiat nulla pariatur. Excepteur sint
10 occaecat cupidatat non proident, sunt in
11 culpa qui officia deserunt mollit anim
12 id est laborum.
$ cat lorem |num |num |num |num |num
  1   1   1   1   1 Lorem ipsum dolor sit amet, consectetur
  2   2   2   2   2 adipisicing elit, sed do eiusmod tempor
  3   3   3   3   3 incididunt ut labore et dolore magna
  4   4   4   4   4 aliqua. Ut enim ad minim veniam, quis
  5   5   5   5   5 nostrud exercitation ullamco laboris
  6   6   6   6   6 nisi ut aliquip ex ea commodo consequat.
  7   7   7   7   7 Duis aute irure dolor in reprehenderit
  8   8   8   8   8 in voluptate velit esse cillum dolore eu
  9   9   9   9   9 fugiat nulla pariatur. Excepteur sint
 10  10  10  10  10 occaecat cupidatat non proident, sunt in
 11  11  11  11  11 culpa qui officia deserunt mollit anim
 12  12  12  12  12 id est laborum.
$ lsfd
fd 0: name <cons> isdir 0 size 0 dev cons
fd 1: name <cons> isdir 0 size 0 dev cons
```

- `runcmd()`

Simply use `case '>':` as a template:

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

After `make run-icode`, I type `sh <script`:

```
This is from the script.
 1 Lorem ipsum dolor sit amet, consectetur
 2 adipisicing elit, sed do eiusmod tempor
 3 incididunt ut labore et dolore magna
 4 aliqua. Ut enim ad minim veniam, quis
 5 nostrud exercitation ullamco laboris
 6 nisi ut aliquip ex ea commodo consequat.
 7 Duis aute irure dolor in reprehenderit
 8 in voluptate velit esse cillum dolore eu
 9 fugiat nulla pariatur. Excepteur sint
10 occaecat cupidatat non proident, sunt in
```

```
   11 culpa qui officia deserunt mollit anim
   12 id est laborum.
These are my file descriptors.
fd 0: name script isdir 0 size 132 dev file
fd 1: name <cons> isdir 0 size 0 dev cons
This is the end of the script.
```

After `make run-testshell`:

```
running sh -x < testshell.sh | cat
shell ran correctly
```

---

## Score

```
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 (2.1s)
  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 (2.0s)
Protection I/O space: OK (0.8s)
PTE_SHARE [testpteshare]: OK (2.0s)
PTE_SHARE [testfdsharing]: OK (2.1s)
start the shell [icode]: Timeout! OK (30.8s)
testshell: OK (1.9s)
primespipe: OK (6.1s)
Score: 150/150
```