MIT 6.828: Operating System Engineering

Lab Report

Lab 5: File system, Spawn and Shell

CSNLP
csnlp16@126.com
https://csnlp.github.io/

Contents

1 Introduction **3**
1.1 Auto Merge Failed . . . 3
1.1.1 env_create() in kern/env.c . . . 3
1.1.2 i386_init() in kern/init.c . . . 4
1.1.3 trap_dispatch() in kern/trap.c . . . 6
1.2 New Files . . . 8

2 File System Preliminaries **9**
2.1 On-Disk File System Structure . . . 9
2.1.1 Sectors and Blocks . . . 9
2.1.2 Super . . . 9
2.1.3 File Meta-data . . . 9
2.1.4 Directories versus Regular Files . . . 10

3 The File System **11**
3.1 Disk Access . . . 11
3.1.1 Exercise 1 . . . 11
3.2 The Block Cache . . . 12
3.2.1 Exercise 2 . . . 12
3.2.2 Implementation of bc_pgfault() in fs/bc.c . . . 13
3.2.3 Implementation of flush_block() . . . 15
3.2.4 Present Grade . . . 15
3.3 The Block Bitmap . . . 15
3.3.1 Exercise 3 . . . 15
3.3.2 The Implementation of alloc_block() . . . 15
3.3.3 Present Grade . . . 16
3.4 File Operations . . . 17
3.4.1 Exercise 4 . . . 17
3.4.2 Functions in fs/fs.c . . . 17
3.4.3 The Implementation of file_block_walk() in fs/fs.c . . . 19
3.4.4 The implementation of file_get_block() . . . 21
3.5 The File System Interface . . . 21
3.5.1 Some Background Knowledge . . . 21
3.5.2 Exercise 5 . . . 23
3.5.3 The Implementation of serve_read in fs/serv.c . . . 23
3.5.4 Present Grade . . . 24
3.5.5 Exercise 6 . . . 24
3.5.6 The Implementation of serve_write() in fs/serv.c . . . 25
3.5.7 The Implementation of devfile_write() in lib/file.c . . . 25
3.5.8 Present Grade . . . 26

4 Spawning Processes **27**
4.1 Spawn . . . 27
4.1.1 Exercise 7 . . . 27
4.1.2 DO NOT FORGET syscall in kern/syscall.c . . . 28
4.1.3 The Results . . . 28

4.2 Sharing Library State across Fork and Spawn 29
4.2.1 Background Knowledge 29
4.2.2 Exercise 8 29
4.2.3 The Implementation of duppage() in lib/fork.c: PTE_SHARE VERSION 29
4.2.4 The Implementation of copy_shared_pages() in lib/spawn.c: PTE_SHARE VERSION 31

5 The Keyboard Interface **32**
5.1 Exercise 9 32

6 The Shell **33**
6.1 runcmd in user/sh.c 33
6.2 Big BUG 36
6.2.1 Present Grade 36

7 Reference **37**

1 Introduction

1.1 Auto Merge Failed

Actually, there are some conflict happen when I try to auto merge.

```
Auto-merging lib/printfmt.c
Auto-merging kern/trap.c
CONFLICT (content): Merge conflict in kern/trap.c
Auto-merging kern/syscall.c
Auto-merging kern/sched.c
Auto-merging kern/pmap.c
Auto-merging kern/init.c
CONFLICT (content): Merge conflict in kern/init.c
Auto-merging kern/env.c
CONFLICT (content): Merge conflict in kern/env.c
Automatic merge failed; fix conflicts and then commit the result.
```

1.1.1 env_create() in kern/env.c

```
void
env_create(uint8_t *binary, enum EnvType type)
{
        // LAB 3: Your code here.
<<<<<<< HEAD

        // If this is the file server (type == ENV_TYPE_FS) give it I/O privileges.
        // LAB 5: Your code here.
=======
        struct Env *env;
        int rc;
        // step 1: allocates a new env with env_alloc
        rc = env_alloc(&env, 0);
        if(rc != 0)
                panic("env_create: env_alloc failed");
        // step 2: loads the named elf binary with load_icode
        load_icode(env, binary);
        // step 3: set env's env_type
        env->env_type = type;

>>>>>>> lab4
}
```

Modify as

```
void
env_create(uint8_t *binary, enum EnvType type)
{
        // LAB 3: Your code here.

        struct Env *env;
        int rc;
        // step 1: allocates a new env with env_alloc
        rc = env_alloc(&env, 0);
        if(rc != 0)
                panic("env_create: env_alloc failed");
        // step 2: loads the named elf binary with load_icode
        load_icode(env, binary);
        // step 3: set env's env_type
        env->env_type = type;

        // If this is the file server (type == ENV_TYPE_FS) give it I/O privileges.
        // LAB 5: Your code here.
}
```

1.1.2 i386_init() in kern/init.c

```
void
i386_init(void)
{
        // Initialize the console.
        // Can't call cprintf until after we do this!
        cons_init();

        cprintf("6828 decimal is %o octal!\n", 6828);

        // Lab 2 memory management initialization functions
        mem_init();

        // Lab 3 user environment initialization functions
        env_init();
        trap_init();

        // Lab 4 multiprocessor initialization functions
        mp_init();
        lapic_init();

        // Lab 4 multitasking initialization functions
        pic_init();

        // Acquire the big kernel lock before waking up APs
        // Your code here:
        lock_kernel();

        // Starting non-boot CPUs
        boot_aps();

        // Start fs.
        ENV_CREATE(fs_fs, ENV_TYPE_FS);
#if defined(TEST)
        // Don't touch -- used by grading script!
        ENV_CREATE(TEST, ENV_TYPE_USER);
#else
        // Touch all you want.
<<<<<<< HEAD
        ENV_CREATE(user_icode, ENV_TYPE_USER);
=======
        //ENV_CREATE(user_primes, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
>>>>>>> lab4
#endif // TEST*

        // Should not be necessary - drains keyboard because interrupt has given up.
        kbd_intr();

        // Schedule and run the first user environment!
        sched_yield();
}
```

Modify as:

```
void
i386_init(void)
{
        // Initialize the console.
        // Can't call cprintf until after we do this!
        cons_init();

        cprintf("6828 decimal is %o octal!\n", 6828);

        // Lab 2 memory management initialization functions
        mem_init();

        // Lab 3 user environment initialization functions
        env_init();
        trap_init();

        // Lab 4 multiprocessor initialization functions
        mp_init();
        lapic_init();

        // Lab 4 multitasking initialization functions
        pic_init();

        // Acquire the big kernel lock before waking up APs
        // Your code here:
        lock_kernel();

        // Starting non-boot CPUs
        boot_aps();

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

#if defined(TEST)
        // Don't touch -- used by grading script!
        ENV_CREATE(TEST, ENV_TYPE_USER);
#else
        // Touch all you want.
        ENV_CREATE(user_icode, ENV_TYPE_USER);
        //ENV_CREATE(user_primes, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
#endif // TEST*

        // Should not be necessary - drains keyboard because interrupt has given up.
        kbd_intr();

        // Schedule and run the first user environment!
        sched_yield();
}
```

1.1.3 trap_dispatch() in kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
	// Handle spurious interrupts
	// The hardware sometimes raises these because of noise on the
	// IRQ line or other reasons. We don't care.
	if (tf->tf_trapno == IRQ_OFFSET + IRQ_SPURIOUS) {
		cprintf("Spurious interrupt on irq 7\n");
		print_trapframe(tf);
		return;
	}
	// Handle processor exceptions.
	// LAB 3: Your code here.
	switch(tf->tf_trapno) {
		case T_PGFLT:
			page_fault_handler(tf);
			return;
		case T_BRKPT:
			monitor(tf);
			return;
		case T_SYSCALL:
			tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax,
				tf->tf_regs.reg_edx,
				tf->tf_regs.reg_ecx, tf->tf_regs.reg_ebx,
				tf->tf_regs.reg_edi, tf->tf_regs.reg_esi);
			return;
		// Handle clock interrupts. Don't forget to acknowledge the
		// interrupt using lapic_eoi() before calling the scheduler!
		// LAB 4: Your code here.
		case IRQ_OFFSET + IRQ_TIMER:
			lapic_eoi();
			sched_yield();
			return;
		default:
			// Unexpected trap: The user process or the kernel has a bug.
			print_trapframe(tf);
			if (tf->tf_cs == GD_KT)
				panic("unhandled trap in kernel");
			else {
				env_destroy(curenv);
				return;
			}
			break;
	}
<<<<<<< HEAD
	// Handle keyboard and serial interrupts.
	// LAB 5: Your code here.

	// Unexpected trap: The user process or the kernel has a bug.
	print_trapframe(tf);
	if (tf->tf_cs == GD_KT)
		panic("unhandled trap in kernel");
	else {
		env_destroy(curenv);
```

```
                return;
        }
=======
>>>>>>> lab4
}
```

Modify as

```
static void
trap_dispatch(struct Trapframe *tf)
{
        // Handle spurious interrupts
        // The hardware sometimes raises these because of noise on the
        // IRQ line or other reasons. We don't care.
        if (tf->tf_trapno == IRQ_OFFSET + IRQ_SPURIOUS) {
                cprintf("Spurious interrupt on irq 7\n");
                print_trapframe(tf);
                return;
        }
        // Handle processor exceptions.
        // LAB 3: Your code here.
        switch(tf->tf_trapno) {
                case T_PGFLT:
                        page_fault_handler(tf);
                        return;
                case T_BRKPT:
                        monitor(tf);
                        return;
                case T_SYSCALL:
                        tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax,
                                tf->tf_regs.reg_edx,
                                tf->tf_regs.reg_ecx, tf->tf_regs.reg_ebx,
                                tf->tf_regs.reg_edi, tf->tf_regs.reg_esi);
                        return;
                // Handle clock interrupts. Don't forget to acknowledge the
                // interrupt using lapic_eoi() before calling the scheduler!
                // LAB 4: Your code here.
               case IRQ_OFFSET + IRQ_TIMER:
                        lapic_eoi();
                        sched_yield();
                        return;
                default:
                        // Unexpected trap: The user process or the kernel has a bug.
                        print_trapframe(tf);
                        if (tf->tf_cs == GD_KT)
                                panic("unhandled trap in kernel");
                        else {
                                env_destroy(curenv);
                                return;
                        }
                        break;
        }


        // Unexpected trap: The user process or the kernel has a bug.
        print_trapframe(tf);
        if (tf->tf_cs == GD_KT)
                panic("unhandled trap in kernel");
        else {
                env_destroy(curenv);
                return;
        }

        // Handle keyboard and serial interrupts.
        // LAB 5: Your code here.
}
```

1.2 New Files

`fs/fs.c`	Code that mainipulates the file system's on–disk structure.
`fs/bc.c`	A simple block cache built on top of our user–level page fault handling facility.
`fs/ide.c`	Minimal PIO–based (non–interrupt–driven) IDE driver code.
`fs/serv.c`	The file system server that interacts with client environments using file system IPCs.
`lib/fd.c`	Code that implements the general UNIX–like file descriptor interface.
`lib/file.c`	The driver for on–disk file type, implemented as a file system IPC client.
`lib/console.c`	The driver for console input/output file type.
`lib/spawn.c`	Code skeleton of the `spawn` library call.

Figure 1: New files for Lab 5

2 File System Preliminaries

2.1 On-Disk File System Structure

Most UNIX file system divide available disk space into two main types of regions:

- inode regions.
- data regions.

2.1.1 Sectors and Blocks

- **sectors**: Most disks cannot perform reads and writes at byte granularity and instead perform reads and writes in units of sectors.
- **blocks:** File system actually allocate and use disk storage in units of blocks.

Be wary of the distinction between the two terms: {sector, block}. Sector size is a property of the disk hardware, whereas block size is an aspect of the OS using the disk. In this sense, a file system's block size must be a multiple of the sector size of the underlying disk.

2.1.2 Super

See its implementation:

```
struct Super {
        uint32_t s_magic;               // Magic number: FS_MAGIC
        uint32_t s_nblocks;             // Total number of blocks on disk
        struct File s_root;             // Root directory node
};
```

2.1.3 File Meta-data

This meta-data includes:

- file's name.
- file's size.
- file's type.
- pointers to the blocks comprising the file.

```
struct File {
        char f_name[MAXNAMELEN];        // filename
        off_t f_size;                   // file size in bytes
        uint32_t f_type;                // file type

        // Block pointers.
        // A block is allocated iff its value is != 0.
        uint32_t f_direct[NDIRECT];     // direct blocks
        uint32_t f_indirect;            // indirect block

        // Pad out to 256 bytes; must do arithmetic in case we're compiling
        // fsformat on a 64-bit machine.
        uint8_t f_pad[256 - MAXNAMELEN - 8 - 4*NDIRECT - 4];
} __attribute__((packed));      // required only on some 64-bit machines
```

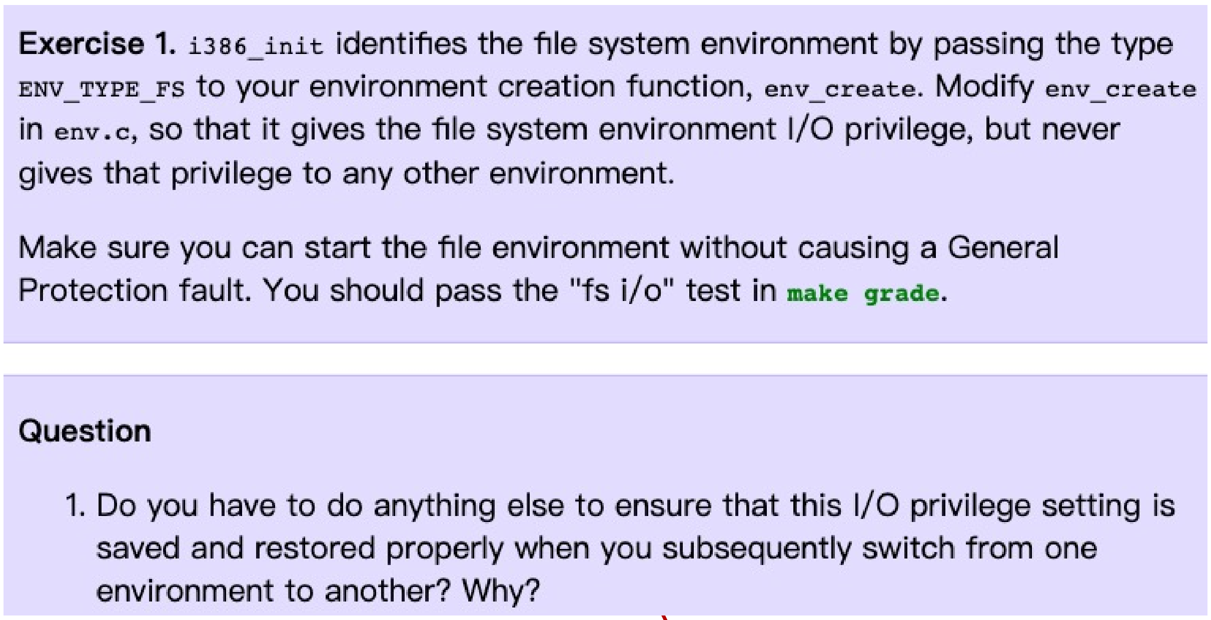
2.1.4 Directories versus Regular Files

A File structure in our file system can represent either a regular file or a directory; these two types of "files" are distinguished by the type field in the File structure. The file system manages regular files and directory-files in exactly the same way, except that it does not interpret the contents of the data blocks associated with regular files at all, whereas the file system interprets the contents of a directory-file as a series of File structures describing the files and subdirectories within the directory.

3 The File System

3.1 Disk Access

3.1.1 Exercise 1



Exercise 1. `i386_init` identifies the file system environment by passing the type `ENV_TYPE_FS` to your environment creation function, `env_create`. Modify `env_create` in `env.c`, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment.

Make sure you can start the file environment without causing a General Protection fault. You should pass the "fs i/o" test in `make grade`.

Question

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?

Figure 2: Exercise 1

3.1.1.1 See the background knowledge

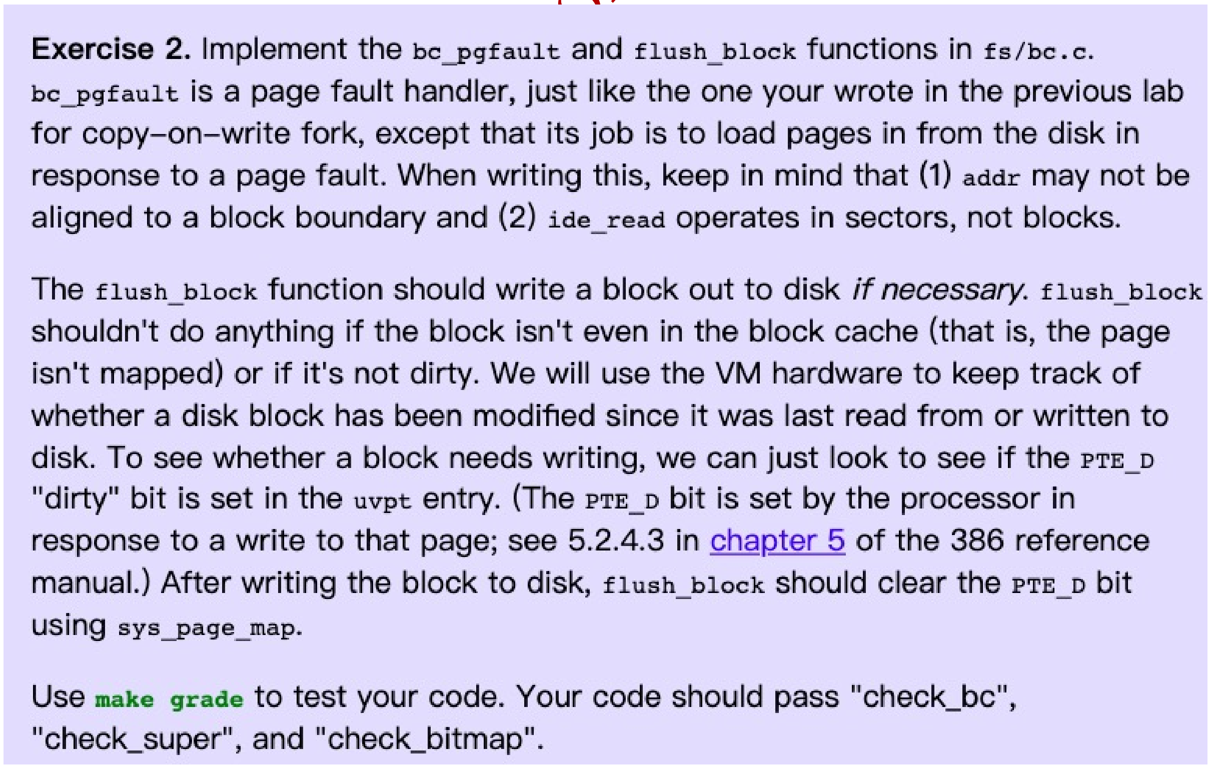
The x86 processor uses the IOPL bits in the EFLAGS register to determine whether protected-mode code is allowed to perform special device I/O instructions such as the IN and OUT instructions. Since all of the IDE disk registers we need to access are located in the x86's I/O space rather than being memory-mapped, giving "I/O privilege" to the file system environment is the only thing we need to do in order to allow the file system to access these registers. In effect, the IOPL bits in the EFLAGS register provides the kernel with a simple "all-or-nothing" method of controlling whether user-mode code can access I/O space. In our case, we want the file system environment to be able to access I/O space, but we do not want any other environments to be able to access I/O space at all.

```c
//
// Allocates a new env with env_alloc, loads the named elf
// binary into it with load_icode, and sets its env_type.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
// The new env's parent ID is set to 0.
//
void
env_create(uint8_t *binary, enum EnvType type)
{
        // LAB 3: Your code here.

        struct Env *env;
        int rc;
        // step 1: allocates a new env with env_alloc
        rc = env_alloc(&env, 0);
        if(rc != 0)
                panic("env_create: env_alloc failed");
        // step 2: loads the named elf binary with load_icode
        load_icode(env, binary);
        // step 3: set env's env_type
        env->env_type = type;

        // If this is the file server (type == ENV_TYPE_FS) give it I/O privileges.
        // LAB 5: Your code here.
        if (type == ENV_TYPE_FS)
                env->env_tf.tf_eflags |= FL_IOPL_MASK;

}
```

Exercise	Score
Exercise 1	25/150

3.2 The Block Cache

3.2.1 Exercise 2


Exercise 2. Implement the `bc_pgfault` and `flush_block` functions in `fs/bc.c`. `bc_pgfault` is a page fault handler, just like the one your wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) `addr` may not be aligned to a block boundary and (2) `ide_read` operates in sectors, not blocks.

The `flush_block` function should write a block out to disk *if necessary*. `flush_block` shouldn't do anything if the block isn't even in the block cache (that is, the page isn't mapped) or if it's not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the `PTE_D` "dirty" bit is set in the `uvpt` entry. (The `PTE_D` bit is set by the processor in response to a write to that page; see 5.2.4.3 in chapter 5 of the 386 reference manual.) After writing the block to disk, `flush_block` should clear the `PTE_D` bit using `sys_page_map`.

Use **`make grade`** to test your code. Your code should pass "check_bc", "check_super", and "check_bitmap".

Figure 3: Exercise 2

3.2.2 Implementation of bc_pgfault() in fs/bc.c

3.2.2.1 See its hints

```c
// Fault any disk block that is read in to memory by
// loading it from disk.
static void
bc_pgfault(struct UTrapframe *utf)
{
        void *addr = (void *) utf->utf_fault_va;
        uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
        int r;

        // Check that the fault was within the block cache region
        if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
                panic("page fault in FS: eip %08x, va %08x, err %04x",
                      utf->utf_eip, addr, utf->utf_err);

        // Sanity check the block number.
        if (super && blockno >= super->s_nblocks)
                panic("reading non-existent block %08x\n", blockno);

        // Allocate a page in the disk map region, read the contents
        // of the block from the disk into that page.
        // Hint: first round addr to page boundary. fs/ide.c has code to read
        // the disk.
        //
        // LAB 5: you code here:
        // Clear the dirty bit for the disk block page since we just read the
        // block from disk
        if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
                panic("in bc_pgfault, sys_page_map: %e", r);

        // Check that the block we read was allocated. (exercise for
        // the reader: why do we do this *after* reading the block
        // in?)
        if (bitmap && block_is_free(blockno))
                panic("reading free block %08x\n", blockno);
}
```

3.2.2.2 See its implementation

```c
static void
bc_pgfault(struct UTrapframe *utf)
{
        void *addr = (void *) utf->utf_fault_va;
        uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
        int r;

        // Check that the fault was within the block cache region
        if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
                panic("page fault in FS: eip %08x, va %08x, err %04x",
                      utf->utf_eip, addr, utf->utf_err);

        // Sanity check the block number.
        if (super && blockno >= super->s_nblocks)
                panic("reading non-existent block %08x\n", blockno);

        // Allocate a page in the disk map region, read the contents
        // of the block from the disk into that page.
        // Hint: first round addr to page boundary. fs/ide.c has code to read
        // the disk.
        //
        // LAB 5: you code here:
        // first round addr to page boundary
        addr = (void *) ROUNDDOWN(addr, PGSIZE);
        // Allocate a page at addr
        if ((r = sys_page_alloc(0, addr, PTE_P|PTE_W|PTE_U)) < 0)
                panic("bc_pgfault failed: sys_page_alloc failed: %e", r);
        // IDE read
        if((r = ide_read(blockno*BLKSECTS, addr, BLKSECTS)) < 0)
                panic("bc_pgfault failed: ide failed: %e", r);


        // Clear the dirty bit for the disk block page since we just read the
        // block from disk
        if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
                panic("in bc_pgfault, sys_page_map: %e", r);

        // Check that the block we read was allocated. (exercise for
        // the reader: why do we do this *after* reading the block
        // in?)
        if (bitmap && block_is_free(blockno))
                panic("reading free block %08x\n", blockno);
}
```

3.2.2.3 ide_read()

It use ide_read(), let's see

```
int
ide_read(uint32_t secno, void *dst, size_t nsecs)
{
        int r;

        assert(nsecs <= 256);

        ide_wait_ready(0);

        outb(0x1F2, nsecs);
        outb(0x1F3, secno & 0xFF);
        outb(0x1F4, (secno >> 8) & 0xFF);
        outb(0x1F5, (secno >> 16) & 0xFF);
        outb(0x1F6, 0xE0 | ((diskno&1)<<4) | ((secno>>24)&0x0F));
        outb(0x1F7, 0x20);      // CMD 0x20 means read sector

        for (; nsecs > 0; nsecs--, dst += SECTSIZE) {
                if ((r = ide_wait_ready(1)) < 0)
                        return r;
                insl(0x1F0, dst, SECTSIZE/4);
        }

        return 0;
}
```

3.2.3 Implementation of flush_block()

```c
// Flush the contents of the block containing VA out to disk if
// necessary, then clear the PTE_D bit using sys_page_map.
// If the block is not in the block cache or is not dirty, does
// nothing.
// Hint: Use va_is_mapped, va_is_dirty, and ide_write.
// Hint: Use the PTE_SYSCALL constant when calling sys_page_map.
// Hint: Don't forget to round addr down.
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);

        // LAB 5: Your code here.
        panic("flush_block not implemented");
}
```

3.2.4 Present Grade

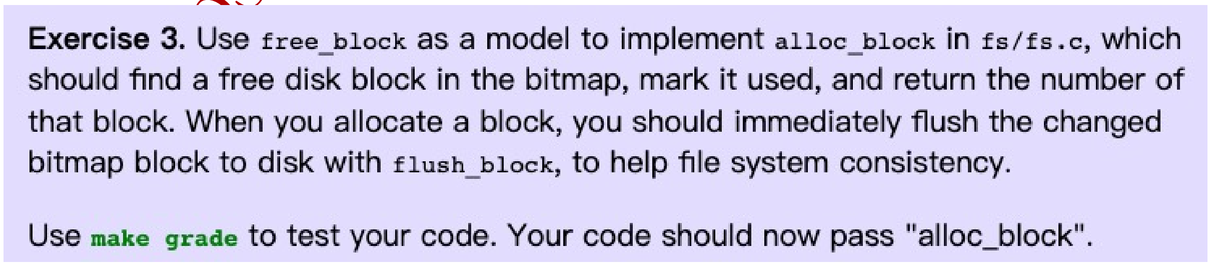
```
internal FS tests [fs/test.c]: OK (1.3s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
```

Grade: 40/150.

3.3 The Block Bitmap

After fs_init sets the bitmap pointer, we can treat bitmap as a packed array of bits, one for each block on the disk. See, for example, block_is_free, which simply checks whether a given block is marked free in the bitmap.

3.3.1 Exercise 3



Exercise 3. Use `free_block` as a model to implement `alloc_block` in `fs/fs.c`, which should find a free disk block in the bitmap, mark it used, and return the number of that block. When you allocate a block, you should immediately flush the changed bitmap block to disk with `flush_block`, to help file system consistency.

Use `make grade` to test your code. Your code should now pass "alloc_block".

Figure 4: Exercise 3

3.3.2 The Implementation of alloc_block()

3.3.2.1 See its hints

```c
// Search the bitmap for a free block and allocate it.  When you
// allocate a block, immediately flush the changed bitmap block
// to disk.
//
// Return block number allocated on success,
// -E_NO_DISK if we are out of blocks.
//
// Hint: use free_block as an example for manipulating the bitmap.
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.
        panic("alloc_block not implemented");
        return -E_NO_DISK;
}
```

3.3.2.2 See the hints from free_block()

```c
// Mark a block free in the bitmap
void
free_block(uint32_t blockno)
{
        // Blockno zero is the null pointer of block numbers.
        if (blockno == 0)
                panic("attempt to free zero block");
        bitmap[blockno/32] |= 1<<(blockno%32);
}
```

3.3.2.3 See its implementation

```c
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.
        size_t i;
        for(i = 1; i < super->s_nblocks; i++) {
                // find the free block
                if(block_is_free(i)) {
                        bitmap[i/32] ^= (1 << (i%32));
                        flush_block(&bitmap[i%32]);
                        return i;
                }
        }
        //panic("alloc_block not implemented");
        return -E_NO_DISK;
}
```

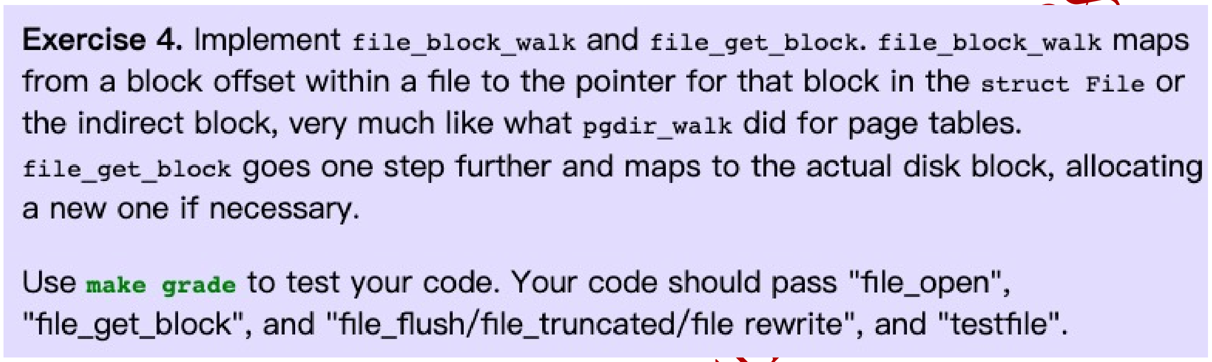
3.3.3 Present Grade

```
internal FS tests [fs/test.c]: OK (1.3s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
  alloc_block: OK
```

Grade: 45/150.

3.4 File Operations

3.4.1 Exercise 4



Exercise 4. Implement `file_block_walk` and `file_get_block`. `file_block_walk` maps from a block offset within a file to the pointer for that block in the `struct File` or the indirect block, very much like what `pgdir_walk` did for page tables. `file_get_block` goes one step further and maps to the actual disk block, allocating a new one if necessary.

Use `make grade` to test your code. Your code should pass "file_open", "file_get_block", and "file_flush/file_truncated/file rewrite", and "testfile".

Figure 5: Exercise 4

3.4.2 Functions in fs/fs.c

- check_super(): check the file system super-block.

```
// Validate the file system super-block.
void
check_super(void) {}
```

- block_is_free(): check if the block is free or not.

```
// Check to see if the block bitmap indicates that block 'blockno' is free.
// Return 1 if the block is free, 0 if not.
bool
block_is_free(uint32_t blockno)
```

- free_block(): mark a block free in the bit map.

```
// Mark a block free in the bitmap
void
free_block(uint32_t blockno)
```

- alloc_block(): search the bitmap for a free block and allocate it.

```
int
alloc_block(void)
```

- check_bitmap()

```c
void
check_bitmap(void)
```

- **fs_init()**: initialize the file system.

```c
void
fs_init(void)
```

- **file_block_walk()**: find the disk block number slot for the 'filebno'th block in file 'f'.

```c
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool
    ↪ alloc)
```

- **file_get_block()**: set *blk to the address in memory where the filebno'th block of file 'f' would be mapped.

```c
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
```

- **dir_lookup()**: try to find a file named "name" in dir. If so, set *file to it.

```c
static int
dir_lookup(struct File *dir, const char *name, struct File **file)
```

- **dir_alloc_file**: set *file to point at a free File structure in dir.

```c
static int
dir_alloc_file(struct File *dir, struct File **file)
```

- **skip_slash()**: Skip over slashes.

```c
static const char*
skip_slash(const char *p)
```

- **walk_path**:

```c
// Evaluate a path name, starting at the root.
// On success, set *pf to the file we found
// and set *pdir to the directory the file is in.
// If we cannot find the file but find the directory
// it should be in, set *pdir and copy the final path
// element into lastelem.
static int
walk_path(const char *path, struct File **pdir, struct File **pf, char *lastelem
    ↪ )
```

- **file_create**:

```c
// Create "path".  On success set *pf to point at the file and return 0.
// On error return < 0.
int
file_create(const char *path, struct File **pf)
```

- file_open:

```
// Open "path".  On success set *pf to point at the file and return 0.
// On error return < 0.
int
file_open(const char *path, struct File **pf)
```

- file_read

```
// Read count bytes from f into buf, starting from seek position
// offset.  This meant to mimic the standard pread function.
// Returns the number of bytes read, < 0 on error.
ssize_t
file_read(struct File *f, void *buf, size_t count, off_t offset)
```

- file_write

```
// Write count bytes from buf into f, starting at seek position
// offset.  This is meant to mimic the standard pwrite function.
// Extends the file if necessary.
// Returns the number of bytes written, < 0 on error.
int
file_write(struct File *f, const void *buf, size_t count, off_t offset)
```

- file_free_block():

```
// Remove a block from file f.  If it's not there, just silently succeed.
// Returns 0 on success, < 0 on error.
static int
file_free_block(struct File *f, uint32_t filebno)
```

- file_set_size(): set the size of file f, truncating or extending as necessary.

```
// Set the size of file f, truncating or extending as necessary.
int
file_set_size(struct File *f, off_t newsize)
```

- file_flush():

```
// Flush the contents and metadata of file f out to disk.
// Loop over all the blocks in file.
// Translate the file block number into a disk block number
// and then check whether that disk block is dirty.  If so, write it out.
void
file_flush(struct File *f)
```

3.4.3 The Implementation of file_block_walk() in fs/fs.c

3.4.3.1 See its hints

```
// Find the disk block number slot for the 'filebno'th block in file 'f'.
// Set '*ppdiskbno' to point to that slot.
// The slot will be one of the f->f_direct[] entries,
// or an entry in the indirect block.
// When 'alloc' is set, this function will allocate an indirect block
// if necessary.
```

```c
//
// Returns:
//      0 on success (but note that *ppdiskbno might equal 0).
//      -E_NOT_FOUND if the function needed to allocate an indirect block, but
//              alloc was 0.
//      -E_NO_DISK if there's no space on the disk for an indirect block.
//      -E_INVAL if filebno is out of range (it's >= NDIRECT + NINDIRECT).
//
// Analogy: This is like pgdir_walk for files.
// Hint: Don't forget to clear any block you allocate.
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");
}
```

3.4.3.2 See its implementation

```c
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
{
        // LAB 5: Your code here.
        // CASE 1: filebno is out of range, i.e., filebno >= NDIRECT + NINDIRECT
        if (filebno >= NDIRECT + NINDIRECT)
                return -E_INVAL;

        // CASE 2: direct
        if (filebno < NDIRECT) {
                // find the disk block number slot for the
                // 'filebno'th block in file 'f'. Set '*ppdiskbno'
                // to point to that slot.
                if(ppdiskbno)
                        *ppdiskbno = &(f->f_direct[filebno]);
                return 0;
        }

        // CASE 3: indirect
        /*
        when code come to this place, it means that
        NDIRECT <= blockno < NDIRECT + NINDIRECT
        */
        // alloc = 0: want to allocate a indirect
        // alloc = 1: want to allocate an direct
        int blockno = 0;
        // if f->f_indirect == 0, set the f->f_indirect
        if (f->f_indirect == 0) {
                // if the function needed to allocate an indirect block
                // but alloc was 0. Return -E_NOT_FOUND
                if (alloc == 0)
                        return -E_NOT_FOUND;
                // if there is no space on the disk for an indirect block
                if ((blockno = alloc_block()) < 0)
                        return -E_NO_DISK;

                memset(diskaddr(blockno), 0, BLKSIZE);
                flush_block(diskaddr(blockno));
                f->f_indirect = blockno;
        }
        if (ppdiskbno)
                *ppdiskbno = (uint32_t*)diskaddr(f->f_indirect) + filebno - NDIRECT;
        return 0;
        //panic("file_block_walk not implemented");
}
```

3.4.4 The implementation of file_get_block()

3.4.4.1 See its hints

```c
// Set *blk to the address in memory where the filebno'th
// block of file 'f' would be mapped.
//
// Returns 0 on success, < 0 on error.  Errors are:
//      -E_NO_DISK if a block needed to be allocated but the disk is full.
//      -E_INVAL if filebno is out of range.
//
// Hint: Use file_block_walk and alloc_block.
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
       // LAB 5: Your code here.
        panic("file_get_block not implemented");
}
```

3.4.4.2 See its implementation

```c
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
       // LAB 5: Your code here.
        uint32_t *ppdiskbno;
        int r;
        if ((r = file_block_walk(f, filebno, &ppdiskbno, true)) < 0)
                return r;
        // if (*ppdiskbno == 0), point nothing, pose it.
        if (*ppdiskbno == 0) {
                if ((r = alloc_block()) < 0)
                        return -E_NO_DISK;
                *ppdiskbno = r;
                memset(diskaddr(r), 0, BLKSIZE);
                flush_block(diskaddr(r));
        }
        *blk = diskaddr(*ppdiskbno);
        return 0;
        //panic("file_get_block not implemented");
}
```

3.5 The File System Interface

3.5.1 Some Background Knowledge

Now that we have the necessary functionality within the file system environment itself, we must make it accessible to other environments that wish to use the file system. Since other environments can't directly call functions in the file system environment, we'll expose access to the file system environment via a remote procedure call, or **RPC**, abstraction, built atop JOS's IPC mechanism. Graphically, here's what a call to the file system server (say, read) looks like:

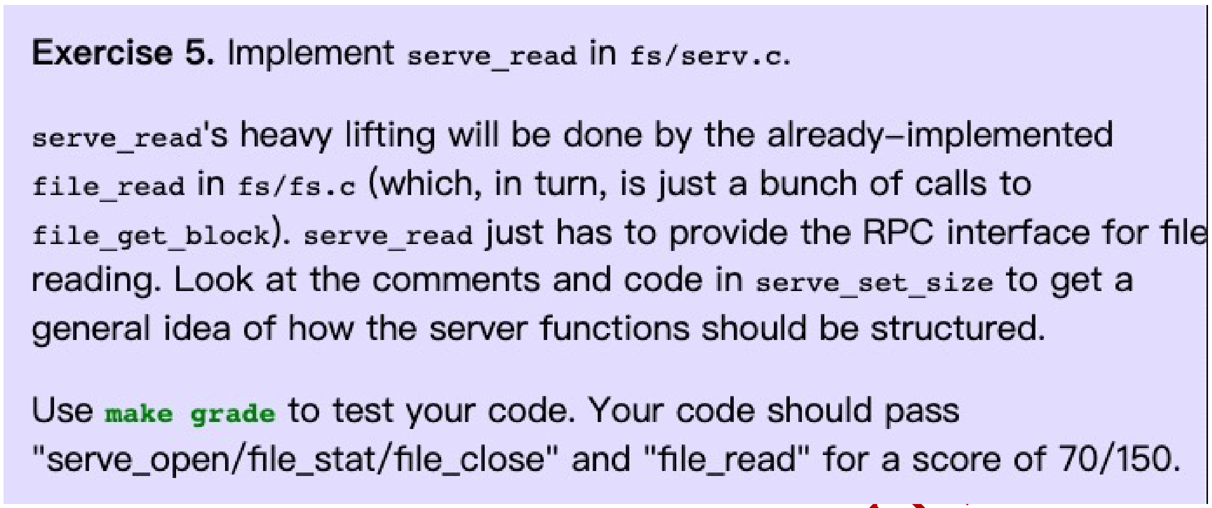
```
      Regular env           FS env
    +---------------+   +---------------+
    |      read     |   |   file_read   |
    |   (lib/fd.c)  |   |   (fs/fs.c)   |
....|.......|.......|...|.......^.......|...............
    |       v       |   |       |       | RPC mechanism
    |  devfile_read |   |  serve_read   |
    |  (lib/file.c) |   |  (fs/serv.c)  |
    |       |       |   |       ^       |
    |       v       |   |       |       |
    |     fsipc     |   |     serve     |
    |  (lib/file.c) |   |  (fs/serv.c)  |
    |       |       |   |       ^       |
    |       v       |   |       |       |
    |   ipc_send    |   |   ipc_recv    |
    |       |       |   |       ^       |
    +-------|-------+   +-------|-------+
            |                   |
            +-------------------+
```

Figure 6: File System Server

See the whole process of call to the file system server:

- **read** works on any file descriptor and simply dispatches to the appropriate device read function, in this case, the **devfile_read** function.
- **decfile_read** implements **read** specifically for on-disk files.
- This and the other **devfile_*** functions in **lib/file.c** implment the client side of the FS operations and all work in roughly the same way, bundling up arguments in a request structure, calling **fsipc** to send the IPC request, and unpacking and returning the results.

3.5.2 Exercise 5



Exercise 5. Implement `serve_read` in `fs/serv.c`.

`serve_read`'s heavy lifting will be done by the already–implemented `file_read` in `fs/fs.c` (which, in turn, is just a bunch of calls to `file_get_block`). `serve_read` just has to provide the RPC interface for file reading. Look at the comments and code in `serve_set_size` to get a general idea of how the server functions should be structured.

Use `make grade` to test your code. Your code should pass "serve_open/file_stat/file_close" and "file_read" for a score of 70/150.

Figure 7: Exercise 5

3.5.3 The Implementation of serve_read in fs/serv.c

3.5.3.1 See its hints

```c
// Read at most ipc->read.req_n bytes from the current seek position
// in ipc->read.req_fileid.  Return the bytes read from the file to
// the caller in ipc->readRet, then update the seek position.  Returns
// the number of bytes successfully read, or < 0 on error.
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:
        return 0;
}
```

The exercise5's hints suggest us to revisit the serve_set_size() code:

3.5.3.2 serve_set_size() code

```c
// Set the size of req->req_fileid to req->req_size bytes, truncating
// or extending the file as necessary.
int
serve_set_size(envid_t envid, struct Fsreq_set_size *req)
{
        struct OpenFile *o;
        int r;

        if (debug)
```

```
                cprintf("serve_set_size %08x %08x %08x\n", envid, req->req_fileid, req->req_size);

        // Every file system IPC call has the same general structure.
        // Here's how it goes.

        // First, use openfile_lookup to find the relevant open file.
        // On failure, return the error code to the client with ipc_send.
        if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
                return r;

        // Second, call the relevant file system function (from fs/fs.c).
        // On failure, return the error code to the client.
        return file_set_size(o->o_file, req->req_size);
}
```

Its steps:

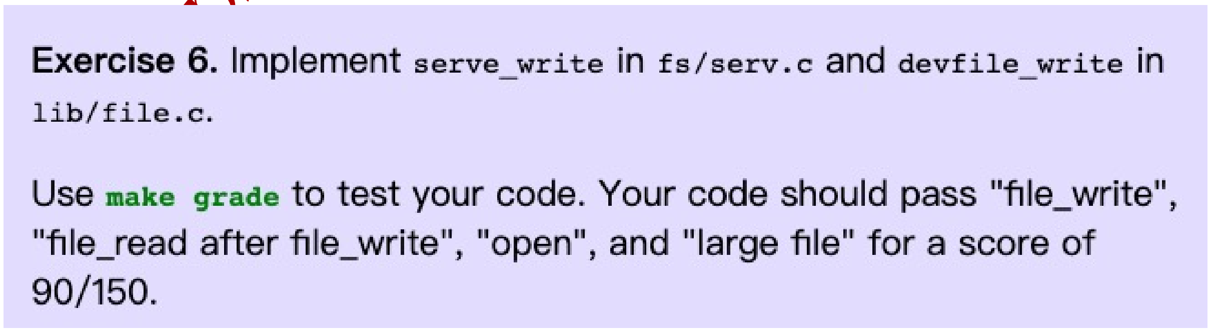
1. Use openfile_lookup to find the relevant open file.
2. Call the relevant file system function.

3.5.4 Present Grade

```
internal FS tests [fs/test.c]: OK (1.1s)
  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 (1.0s)
  serve_open/file_stat/file_close: OK
  file_read: OK
```

Grade: 70/150.

3.5.5 Exercise 6



Exercise 6. Implement `serve_write` in `fs/serv.c` and `devfile_write` in `lib/file.c`.

Use `make grade` to test your code. Your code should pass "file_write", "file_read after file_write", "open", and "large file" for a score of 90/150.

Figure 8: Exercise 6

3.5.6 The Implementation of serve_write() in fs/serv.c

3.5.6.1 Let's see its hints

```c
// Write req->req_n bytes from req->req_buf to req_fileid, starting at
// the current seek position, and update the seek position
// accordingly.  Extend the file if necessary.  Returns the number of
// bytes written, or < 0 on error.
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");
}
```

3.5.6.2 See its implementation

```c
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;
        int req_n;
        struct OpenFile *of;

        //
        if ((r = openfile_lookup(envid, req->req_fileid, &of)) < 0)
                return r;
        req_n = req->req_n > PGSIZE ? PGSIZE : req->req_n;

        // Write req_n bytes from req->req_buf to
        // req_fileid, starting at current seek position.
        if ((r = file_write(of->o_file, req->req_buf, req_n, of->o_fd->fd_offset)) < 0)
                return r;
        // update the seek position.
        of->o_fd->fd_offset += r;
        return r;

        //panic("serve_write not implemented");
}
```

3.5.7 The Implementation of devfile_write() in lib/file.c

3.5.7.1 Firstly, see its hints

```c
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

        panic("devfile_write not implemented");
}
```

3.5.7.2 See its implementation

```c
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;
        n = MIN(n, sizeof(fsipcbuf.write.req_buf));

        fsipcbuf.write.req_fileid = fd->fd_file.id;
        fsipcbuf.write.req_n = n;
        memmove(fsipcbuf.write.req_buf, buf, n);
        return fsipc(FSREQ_WRITE, NULL);
        //panic("devfile_write not implemented");
}
```

3.5.8 Present Grade

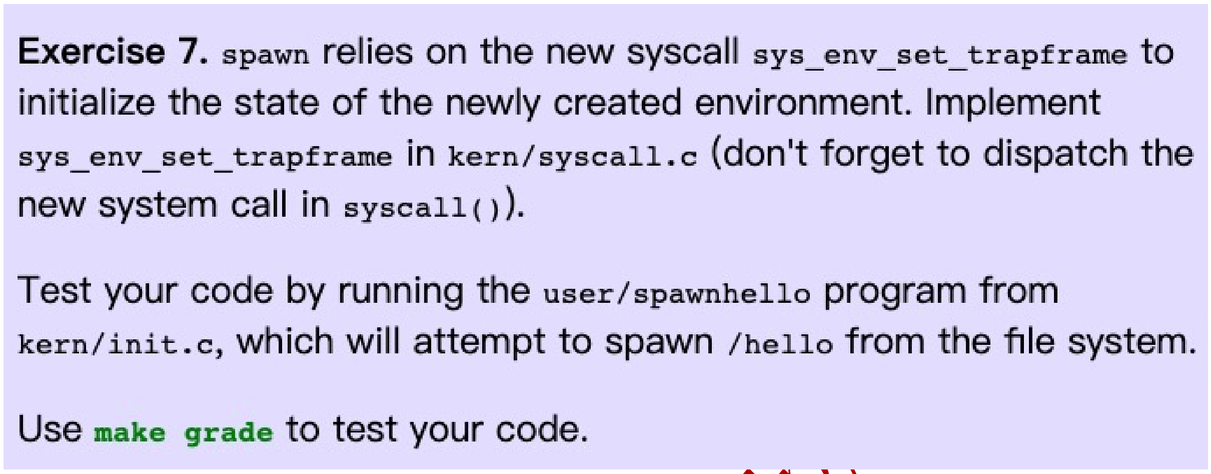
```
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
  alloc_block: OK
  file_open: OK
  file_get_block: OK
  file_flush/file_truncate/file rewrite: OK
testfile: OK (1.3s)
  serve_open/file_stat/file_close: OK
  file_read: OK
  file_write: OK
  file_read after file_write: OK
  open: OK
  large file: OK
```

Grade: 90/150.

4 Spawning Processes

4.1 Spawn

4.1.1 Exercise 7



Exercise 7. `spawn` relies on the new syscall `sys_env_set_trapframe` to initialize the state of the newly created environment. Implement `sys_env_set_trapframe` in `kern/syscall.c` (don't forget to dispatch the new system call in `syscall()`).

Test your code by running the `user/spawnhello` program from `kern/init.c`, which will attempt to spawn `/hello` from the file system.

Use `make grade` to test your code.

Figure 9: Exercise 7

4.1.1.1 Hints in sys_env_set_trapframe()

```c
// Set envid's trap frame to 'tf'.
// tf is modified to make sure that user environments always run at code
// protection level 3 (CPL 3), interrupts enabled, and IOPL of 0.
//
// Returns 0 on success, < 0 on error.  Errors are:
//      -E_BAD_ENV if environment envid doesn't currently exist,
//              or the caller doesn't have permission to change envid.
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");
}
```

4.1.1.2 See its implementation

```c
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!
        struct Env *env;
        int r;

        if ((r = envid2env(envid, &env, 1)) < 0)
```

```
                return -E_BAD_ENV;
        user_mem_assert(env, tf, sizeof(struct Trapframe), PTE_U);
        env->env_tf = *tf;
        env->env_tf.tf_cs |= 3;
        env->env_tf.tf_eflags |= FL_IF;
        return 0;
        //panic("sys_env_set_trapframe not implemented");
}
```

4.1.2 DO NOT FORGET syscall in kern/syscall.c

```
        // LAB 5: sys_env_set_trapframe
        case SYS_env_set_trapframe:
                return sys_env_set_trapframe((envid_t)a1, (struct Trapframe*)a2);
```

4.1.3 The Results

```
cui@cui-VirtualBox:~/mit6828/lab$ make run-spawnhello
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
make[1]: Entering directory '/home/cui/mit6828/lab'
+ cc kern/init.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
make[1]: 'obj/fs/fs.img' is up to date.
make[1]: Leaving directory '/home/cui/mit6828/lab'
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial
    ↪  mon:stdio -gdb tcp::26000 -D qemu.log -smp 1 -drive file=obj/fs/fs.img,index
    ↪ =1,media=disk,format=raw
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
FS is running
FS can do I/O
Device 1 presence: 1
i am parent environment 00001001
block cache is good
superblock is good
bitmap is good
alloc_block is good
file_open is good
file_get_block is good
file_flush is good
file_truncate is good
file rewrite is good
hello, world
i am environment 00001002
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

```
internal FS tests [fs/test.c]: OK (1.7s)
  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 (1.2s)
  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 (0.9s)
    (Old jos.out.spawn failure log removed)
```

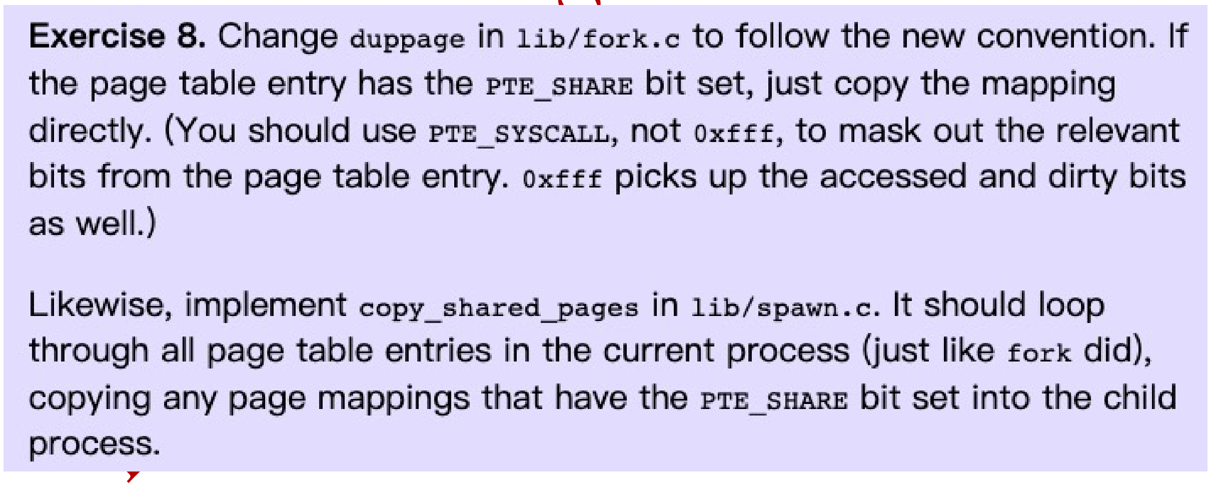
Grade: 95/150.

4.2 Sharing Library State across Fork and Spawn

4.2.1 Background Knowledge

In JOS, each of these device types has a corresponding struct Dev

4.2.2 Exercise 8



Exercise 8. Change `duppage` in `lib/fork.c` to follow the new convention. If the page table entry has the `PTE_SHARE` bit set, just copy the mapping directly. (You should use `PTE_SYSCALL`, not `0xfff`, to mask out the relevant bits from the page table entry. `0xfff` picks up the accessed and dirty bits as well.)

Likewise, implement `copy_shared_pages` in `lib/spawn.c`. It should loop through all page table entries in the current process (just like `fork` did), copying any page mappings that have the `PTE_SHARE` bit set into the child process.

Figure 10: Exercise 8

4.2.3 The Implementation of duppage() in lib/fork.c: PTE_SHARE VERSION

4.2.3.1 See the PTE_SHARE version

```
//
// Map our virtual page pn (address pn*PGSIZE) into the target envid
// at the same virtual address.  If the page is writable or copy-on-write,
```

```c
// the new mapping must be created copy-on-write, and then our mapping must be
// marked copy-on-write as well.  (Exercise: Why do we need to mark ours
// copy-on-write again if it was already copy-on-write at the beginning of
// this function?)
//
// Returns: 0 on success, < 0 on error.
// It is also OK to panic on error.
//
static int
duppage(envid_t envid, unsigned pn)
{
        int r;
        // LAB 4: Your code here.
//
//      void *addr;
//      pte_t pte;
//      int perm;
//
//      addr = (void *)((uint32_t)pn * PGSIZE);
//      pte = uvpt[pn];
//      perm = PTE_P | PTE_U;
//      if ((pte & PTE_W) || (pte & PTE_COW))
//              perm |= PTE_COW;
//      if ((r = sys_page_map(thisenv->env_id, addr, envid, addr, perm)) < 0) {
//              panic("duppage failed: page remapping failed: %e", r);
//              return r;
//      }
//      if (perm & PTE_COW) {
//              if ((r = sys_page_map(thisenv->env_id, addr, thisenv->env_id, addr,
    ↪ perm)) < 0) {
//                      panic("duppage failed: page remapping failed: %e", r);
//                      return r;
//              }
//      }
//      //panic("duppage not implemented");
//      return 0;
        // LAB 5: For exercise 8.
        void *addr;
        pte_t pte;
        int perm;

        addr = (void *)((uint32_t)pn * PGSIZE);
        pte = uvpt[pn];
        // If the page table entry has the PTE_SHARE bit set,
        // just copy the mapping directly.
        if (pte & PTE_SHARE) {
                if ((r = sys_page_map(sys_getenvid(), addr, envid, addr, pte &
    ↪ PTE_SYSCALL)) < 0) {
                        panic("duppage failed: sys_page_map failed: %e", r);
                        return r;
                }
        }
        else {
                perm = PTE_P | PTE_U;
                if ((pte & PTE_W) || (pte & PTE_COW))
                        perm |= PTE_COW;
                if ((r = sys_page_map(thisenv->env_id, addr, envid, addr, perm)) < 0)
    ↪ {
```

```
                        panic("duppage failed: page remapping failed: %e", r);
                        return r;
                }
                if (perm & PTE_COW) {
                        if ((r = sys_page_map(thisenv->env_id, addr, thisenv->env_id,
    ↪ addr, perm)) < 0) {
                                panic("duppage failed: page remapping failed: %e", r);
                                return r;
                        }
                }
        }
        return 0;
}
```

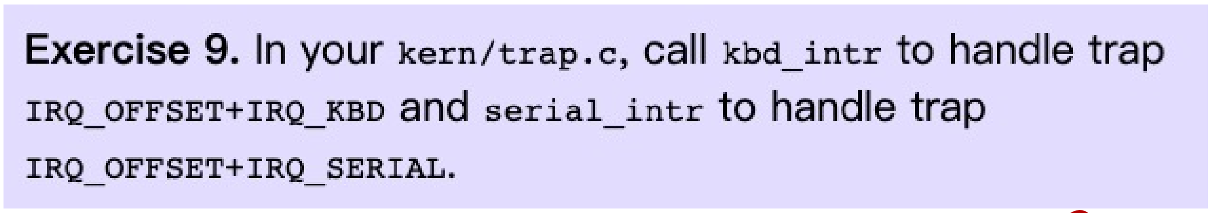
4.2.4 The Implementation of copy_shared_pages() in lib/spawn.c: PTE_SHARE VERSION

It loops through all page table entries in the current process (just like the fork did), copying any page mappings that have the PTE_SHARE bit set into the child process.

4.2.4.1 See its hints

5 The Keyboard Interface

5.1 Exercise 9



Exercise 9. In your `kern/trap.c`, call `kbd_intr` to handle trap `IRQ_OFFSET+IRQ_KBD` and `serial_intr` to handle trap `IRQ_OFFSET+IRQ_SERIAL`.

Figure 11: Exercise 9

See the **trap_dispatch()**

```c
		// Handle keyboard and serial interrupts.
		// LAB 5: Your code here.
		case (IRQ_OFFSET + IRQ_KBD):
			lapic_eoi();
			kbd_intr();
			return;
		case (IRQ_OFFSET + IRQ_SERIAL):
			lapic_eoi();
			serial_intr();
			return;
```

6 The Shell

6.1 runcmd in user/sh.c

```c
// Parse a shell command from string 's' and execute it.
// Do not return until the shell command is finished.
// runcmd() is called in a forked child,
// so it's OK to manipulate file descriptor state.
#define MAXARGS 16
void
runcmd(char* s)
{
        char *argv[MAXARGS], *t, argv0buf[BUFSIZ];
        int argc, c, i, r, p[2], fd, pipe_child;

        pipe_child = 0;
        gettoken(s, 0);

again:
        argc = 0;
        while (1) {
                switch ((c = gettoken(0, &t))) {

                case 'w':       // Add an argument
                        if (argc == MAXARGS) {
                                cprintf("too many arguments\n");
                                exit();
                        }
                        argv[argc++] = t;
                        break;
                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.
                        panic("< redirection not implemented");
                        break;
                case '>':       // Output redirection
                        // Grab the filename from the argument list
                        if (gettoken(0, &t) != 'w') {
                                cprintf("syntax error: > not followed by word\n");
                                exit();
                        }
                        if ((fd = open(t, O_WRONLY|O_CREAT|O_TRUNC)) < 0) {
                                cprintf("open %s for write: %e", t, fd);
                                exit();
                        }
```

```
				if (fd != 1) {
					dup(fd, 1);
					close(fd);
				}
				break;
			case '|':	// Pipe
				if ((r = pipe(p)) < 0) {
					cprintf("pipe: %e", r);
					exit();
				}
				if (debug)
					cprintf("PIPE: %d %d\n", p[0], p[1]);
				if ((r = fork()) < 0) {
					cprintf("fork: %e", r);
					exit();
				}
				if (r == 0) {
					if (p[0] != 0) {
						dup(p[0], 0);
						close(p[0]);
					}
					close(p[1]);
					goto again;
				} else {
					pipe_child = r;
					if (p[1] != 1) {
						dup(p[1], 1);
						close(p[1]);
					}
					close(p[0]);
					goto runit;
				}
				panic("| not implemented");
				break;

			case 0:		// String is complete
				// Run the current command!
				goto runit;

			default:
				panic("bad return %d from gettoken", c);
				break;

		}
	}

runit:
	// Return immediately if command line was empty.
	if(argc == 0) {
		if (debug)
			cprintf("EMPTY COMMAND\n");
		return;
	}

	// Clean up command line.
	// Read all commands from the filesystem: add an initial '/' to
	// the command name.
	// This essentially acts like 'PATH=/'.
```

```c
	if (argv[0][0] != '/') {
		argv0buf[0] = '/';
		strcpy(argv0buf + 1, argv[0]);
		argv[0] = argv0buf;
	}
	argv[argc] = 0;

	// Print the command.
	if (debug) {
		cprintf("[%08x] SPAWN:", thisenv->env_id);
		for (i = 0; argv[i]; i++)
			cprintf(" %s", argv[i]);
		cprintf("\n");
	}
	// Spawn the command!
	if ((r = spawn(argv[0], (const char**) argv)) < 0)
		cprintf("spawn %s: %e\n", argv[0], r);

	// In the parent, close all file descriptors and wait for the
	// spawned command to exit.
	close_all();
	if (r >= 0) {
		if (debug)
			cprintf("[%08x] WAIT %s %08x\n", thisenv->env_id, argv[0], r);
		wait(r);
		if (debug)
			cprintf("[%08x] wait finished\n", thisenv->env_id);
	}

	// If we were the left-hand part of a pipe,
	// wait for the right-hand part to finish.
	if (pipe_child) {
		if (debug)
			cprintf("[%08x] WAIT pipe_child %08x\n", thisenv->env_id, pipe_child);
		wait(pipe_child);
		if (debug)
			cprintf("[%08x] wait finished\n", thisenv->env_id);
	}

	// Done!
	exit();
}
```

See the implementation:

```c
			// 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("open file: %s failed: %e", t, fd);
				exit();
			}
```

```
                        if (fd != 0) {
                                dup(fd, 0);
                                close(fd);
                        }
                        //panic("< redirection not implemented");
                        break;
```

6.2 Big BUG

6.2.1 Present Grade

```
make[1]: Leaving directory ‘/home/cui/mit6828/lab’
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 (1.3s)
  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 (1.1s)
Protection I/O space: FAIL (1.3s)
    AssertionError: ...
         (null): made it here --- bug
         No runnable environments in the system!
         Welcome to the JOS kernel monitor!
         Type 'help' for a list of commands.
         qemu: terminating on signal 15 from pid 26196
    MISSING 'TRAP'

    QEMU output saved to jos.out.faultio
PTE_SHARE [testpteshare]: OK (1.2s)
PTE_SHARE [testfdsharing]: OK (1.9s)
start the shell [icode]: Timeout! OK (30.7s)
testshell: OK (3.2s)
primespipe: OK (8.9s)
Score: 145/150
make: *** [grade] Error 1
```

7 Reference

1. bysui's github and blog
   - github: https://github.com/bysui/mit6.828
   - blog: https://blog.csdn.net/bysui
2. SmallPond's github and blog
   - github: https://github.com/SmallPond/MIT6.828_OS
   - blog: https://me.csdn.net/Small_Pond
3. SimpCosm's github
   - github: https://github.com/SimpCosm/6.828
4. fatsheep9146's blog
   - blog: https://www.cnblogs.com/fatsheep9146/category/769143.html