Exercize 1

为文件系统进程添加 IO 权限，添加代码如下，作业 1 完成后，可以通过 `fs i/o` 测试。

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


        // If this is the file server (type == ENV_TYPE_FS) give it I/O privile
        // LAB 5: Your code here.
        struct Env* env=0;
        int r = env_alloc(&env, 0);
        if(r < 0)
                panic("env_create fault\n");
        load_icode(env, binary);
        env->env_type = type;
        if (type == ENV_TYPE_FS)
                env->env_tf.tf_eflags |= FL_IOPL_MASK;

}
```

Question 1

不需要做额外处理。因为不同进程有自己的 Trapframe，互不影响。

文件系统实现

本实验我们要完成的功能包括：

- 读取磁盘中的数据块到块缓存以及将块缓存中的数据刷回磁盘。
- 分配数据块。
- 映射文件偏移到磁盘数据块。
- 在 IPC 接口实现文件的 open，read，write，close。

文件系统镜像是在 fs/fsformat.c 中创建的，最终在 QEMU 中加载的文件系统镜像文件为 obj/fs/fs.img，其中内核镜像在磁盘 0，文件系统镜像在磁盘 1。

文件系统的第 0，1，2 数据块分别用于启动块，超级块，以及块位图。而因为

在文件系统中初始加入了 user 目录和 fs 目录的一些文件，一共用掉了 0-110 块，所以空闲块从 111 开始。

2.1 磁盘访问

不同于 Linux 等系统那样增加一个磁盘驱动并添加相关系统调用实现磁盘访问，JOS 的磁盘驱动是用用户级程序实现的，当然还是要对内核做一点修改，以支持文件系统进程(用户级进程)有权限访问磁盘。

在用户空间访问磁盘可以通过轮询的方式实现，而不是使用磁盘中断的方式，因为使用中断的方式会复杂不少。x86 处理器使用 EFLAGS 寄存器的 IOPL 位来控制磁盘访问权限(即 IN 和 OUT 指令)，用户代码能否访问 IO 空间就通过该标志来设置。JOS 在 i386_init()中运行了一个用户级的文件系统进程，该进程需要有磁盘访问权限。因此作业 1 就是在 env_create 中对 文件系统进程 这个特殊的运行在用户级的进程设置 IOPL 权限，而其他的用户进程不能设置该权限，根据进程类型设置权限即可。

ENV_CREATE(fs_fs, ENV_TYPE_FS);

特殊的文件系统进程代码在 fs/fs.c，它提供了 file_open，file_read，file_write，file_flush 文件操作函数以及 file_get_block, file_block_walk 数据块操作函数等。

2.2 块缓存

JOS 文件系统将 0x10000000(DISKMAP) 到 0xD0000000(DISKMAP+DISKMAX）这个区间的地址空间映射到磁盘，即 JOS 可以处理 3GB 的磁盘文件。如 0x1000000 映射到数据块 0，0x10001000 映射

到数据库 1。块缓存代码在 fs/bc.c 中，其中 diskaddr 函数可以完成数据块号到虚拟地址的转换。

因为文件系统进程自己有地理的虚拟地址空间，所以让它保留 3GB 虚拟空间地址用于映射文件是没问题的。当然我们不会一次将文件全部读到内存中，JOS 采用的是 demand paging，即访问对应的磁盘块发生了页错误时才分配物理页。具体实现在 bc_pgfault 函数中，有点类似 COW fork()的实现，ide_read()的单位是扇区，不是磁盘块，通过 outb 指令设置读取的扇区数，通过 insl 指令读取磁盘数据到对应的虚拟地址 addr 处。bc_pgfault 中分配了一页物理页，然后从磁盘中读取出错的 addr 那一块数据(8 个扇区）到分配的物理页中，然后清除分配页的 dirty 标记，最后调用 block_is_free 检查对应磁盘块确保磁盘块已经分配。注意这里检查磁盘块是否已经分配要在最后检查，是因为 bitmap 的值是在 fs_init 时指定的为 diskaddr(2)，即 0x10002000，在准备读取第二个磁盘块发生页错误进入 bgfault 时，此时 bitmap 对应块还没有从磁盘读取并映射好，所以要在最后检查。

flush_block()函数用于在写入磁盘数据到块缓存后，调用 ide_write() 写入块缓存数据到磁盘中。写入完成后，也要通过 sys_page_map() 清除块缓存的 dirty 标记(每次写入物理页的时候，处理器会自动标记该页为 dirty，即设置 PTE_D 标记)。注意，在 flush_block()中，如果该地址并没有映射或者并没有 dirty，则不需要做任何事情。

bc.c 中的 bc_init 用于完成块缓存初始化，它完成下面几件事：

- 1）设置页错误处理函数为 bc_pgfault。
- 2）调用 check_bc() 检查块缓存设置是否正确。

- 3）读取磁盘块 1 的数据到函数局部变量 super 对应的地址中。(这一步没有什么作用，super 变量也没有用到过，应该是老代码遗留问题)

2.3 块位图

在 fs_init 设置 bitmap 指针后，可以认为 bitmap 就是一个位数组，每个块占据一位。可以通过 block_is_free 检查块位图中的对应块是否空闲，如果为 1 表示空闲，为 0 已经使用。JOS 中第 0，1，2 块分别给 bootloader，superblock 以及 bitmap 使用了。此外，因为在文件系统中加入了 user 目录和 fs 目录的文件，导致 JOS 文件系统一共用掉了 0-110 这 111 个文件块，下一个空闲文件块从 111 开始。

Exercize 2

实现 `fs/bc.c` 中的 bc_pgfault 和 flush_block。注意这里 flush_block 中的 sys_page_map 使用的权限用 PTE_SYSCALL。作业 2 完成后，需要能通过 `"check_bc", "check_super", "check_bitmap"` 这三个测试。

```
        //
        // LAX 5: you code here:
        addr = ROUNDDOWN(addr, PGSIZE);
        sys_page_alloc(0, addr, PTE_W|PTE_U|PTE_P);
        if ((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
                panic("ide_read: %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);
```

```
        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.
        addr = ROUNDDOWN(addr, PGSIZE);
        addr = ROUNDDOWN(addr, PGSIZE);
        if (va_is_mapped(addr) && va_is_dirty(addr)) {
                ide_write(BLKSECTS * blockno, addr, BLKSECTS);
                sys_page_map(0, addr, 0, addr, PTE_SYSCALL);
        }
```

Exercize 3-4

完成 alloc_block，file_block_walk 以及 file_get_block。作业 3，4 完成后，需要通过 "alloc_block", "file_open", "file_get_block", "file_flush/file_truncated/file rewrite", "testfile"。

```
        // LAB 5: Your code here.
        uint32_t bn;
        for (bn = 0; bn < super->s_nblocks; bn += 32) {
                if (bitmap[bn/32] != ~0) {
                        while (!(bitmap[bn/32] & 1<<(bn%32))) // search 1
                                bn++;
                        bitmap[bn/32] &= ~(1<<(bn%32)); // set to 0
                        flush_block(diskaddr(bn));
                        return bn;
                }
        }
        return -E_NO_DISK;
}
```

```
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
{
       // LAB 5: Your code here.
       int bn;
        uint32_t *indirects;
        if (filebno >= NDIRECT + NINDIRECT)
                return -E_INVAL;
        if (filebno < NDIRECT) {
                *ppdiskbno = &(f->f_direct[filebno]);
        } else {
                if (f->f_indirect) {
                        indirects = diskaddr(f->f_indirect);
                        *ppdiskbno = &(indirects[filebno - NDIRECT]);
                } else {
                        if (!alloc)
                                return -E_NOT_FOUND;
                        if ((bn = alloc_block()) < 0)
                                return bn;
                        f->f_indirect = bn;
                        flush_block(diskaddr(bn));
                        indirects = diskaddr(bn);
                        *ppdiskbno = &(indirects[filebno - NDIRECT]);
                }
        }
        return 0;
}
```

```
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
        // LAB 5: Your code here.
        int r;
        uint32_t *pdiskbno;
        if ((r = file_block_walk(f, filebno, &pdiskbno, true)) < 0)
                return r;
        int bn;
        if (*pdiskbno == 0) {
                if ((bn = alloc_block()) < 0)
                        return bn;
                *pdiskbno = bn;
                flush_block(diskaddr(bn));
        }
        *blk = diskaddr(*pdiskbno);
        return 0;
}
```

2.4 文件操作

在 fs/fs.c 中有很多文件操作相关的函数，这里的主要几个结构体要说明下：

- struct File 用于存储文件元数据，前面提到过。
- struct Fd 用于文件模拟层，类似文件描述符，如文件 ID，文件打开模式，文件偏移都存储在 Fd 中。一个进程同时最多打开 MAXFD(32) 个文件。
- 文件系统进程还维护了一个打开文件的描述符表，即 opentab 数组，数组元素为 struct OpenFile。OpenFile 结构体用于存储打开文件信息，包括文件 ID，struct File 以及 struct Fd。JOS 同时打开的文件数一共为 MAXOPEN(1024) 个。

```
•   struct OpenFile {
•       uint32_t o_fileid;  // file id
•       struct File *o_file;    // mapped descriptor for open file
```

```
    int o_mode;     // open mode
    struct Fd *o_fd;    // Fd page
};

struct Fd {
    int fd_dev_id;
    off_t fd_offset;
    int fd_omode;
    union {
        // File server files
        struct FdFile fd_file;
    };
};
```

文件操作函数如下：

- file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)

  这个函数是查找文件第 filebno 块的数据块的地址，查到的地址存储在 ppdiskbno 中。注意这里要检查间接块，如果 alloc 为 1 且寻址的块号>=NDIRECT，而间接块没有分配的话需要分配一个间接块。

- file_get_block(struct File *f, uint32_t filebno, char **blk)

  查找文件第 filebno 块的块地址，并将块地址在虚拟内存中映射的地址存储在 blk 中(即将 diskaddr(blockno)存到 blk 中)。

- dir_lookup(struct File *dir, const char *name, struct File **file)

  在目录 dir 中查找名为 name 的文件，如果找到了设置*file 为找到的文件。因为目录的数据块存储的是 struct File 列表，可以据此来查找文件。

- file_open(const char *path, struct File **pf)

打开文件，设置*pf 为查找到的文件指针。

- file_create(const char *path, struct File **pf)* 创建路径/文件，在* pf 存储创建好的文件指针。
- file_read(struct File *f, void *buf, size_t count, off_t offset)

  从文件的 offset 处开始读取 count 个字节到 buf 中，返回实际读取的字节数。
- file_write(struct File *f, const void *buf, size_t count, off_t offset)

  从文件 offset 处开始写入 buf 中的 count 字节，返回实际写入的字节数。

写文件过程类似，流程是 `devfile_write -> serve_write -> file_write`。这里分析几个例子看下 JOS 读写文件流程：

直接读文件

这里跳过文件描述符层，直接打开文件并读取

i. fs 进程首先调用 serve_init 完成 opentab 的初始化，然后在 地址 0x0ffff000 处 接收 IPC 的页。

- 2）测试进程通过 IPC 发送 FSREQ_OPEN 请求，请求参数在 fsipcbuf 所在页中，然后在 FVA (0xCCCCC000)处接收 fs 进程的 IPC 页。
- 3）fs 进程的 serve() 接收到 FSREQ_OPEN 请求，调用 serve_open() 处理该请求。会先分配一个 OpenFile 结构给文件，设置 o_file 为文件指针，o_fd 为文件描述符等，IPC 映射的页的权限为 PTE_SHARE 等，然后将文件描述符所在的页作为参数发送 IPC 请求给测试进程。
-

iv. 测试进程在 FVA 处读取打开的文件描述符信息，然后返回。

直接读取文件

- 1）调用 devfile_read 发送 fsipc 到文件系统进程。
- 2）fs 进程通过 ipc_recv 接收 fsipc 请求，然后传给 serve 函数处理。serve 函数根据 fspic 请求类型，调用 serve_read 处理请求。
- 3）fs 系统进程最终通过 file_read 完成文件读取。文件读取结果存储到了 fsipcbuf 中的 readRet 中，恰好是一页的大小，而且这个是测试进程一开始就映射了的页面，可以直接读取。

直接写入文件

与读取文件类似，只是不用返回读取结果了，在 IPC 中返回写入字节数即可。

路径是 `serve()->serve_write()->devfile_write()->file_write()`

通过文件描述符打开/读取/写入文件

- 通过文件描述符打开文件时，测试进程会先通过 fd_alloc() 分配一个文件描述符，然后在文件描述符 fd 处接收 fs 进程的 IPC 页，分配的 fd 的地址为 (0xD0000000 + i * PGSIZE)。后面的流程跟之前直接操作类似。
- 通过文件描述符读取写入文件，会先通过 fd_lookup() 找到文件描述符对应的文件信息，然后再根据设备类型调用相应的读写操作。如文件就是 devfile_read/devfile_write，console 就是 devcons_read/devcons_write。

Exercize 5-6

完成 serve_read 和 serve_write 以及 devfile_write 函数。完成后，可以通过 `serve_open/file_stat/file_close"，"file_read"，"file_write"，"file_read after file_write"，"open"，"large file"，` 可得 90/150。

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

```
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.
        struct OpenFile *o;
        int r;
        if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
                return r;
        int total = 0;
        while (1) {
                r = file_write(o->o_file, req->req_buf, req->req_n, o->o_fd->fd_offset);
                if (r < 0) return r;
                total += r;
                o->o_fd->fd_offset += r;
                if (req->req_n <= total)
                        break;
        }
        return total;
}
```

```
//       < 0 on error.
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
        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);
}
```

3 Spawning 进程

spawn 代码用于创建一个子进程，然后从磁盘中加载一个程序代码镜像并在子进程运行加载的程序。这有点类似 Unix 的 fork+exec，但是又有所不同，因为我们的 spawn 进程运行在用户空间，我们通过一个新的系统调用 `sys_env_set_trapframe` 简化了一些操作。

在 fork 和 spawn 中，JOS 需要实现文件描述符的共享，这里引入了一个新的 PTE_SHARE 标识，用于标识共享页，这样在拷贝时可以进行统一处理，不再类似 fork 那样用 COW，而是直接共享。因为用 fork 的话，如子进程修改了文件数据，此时会新分配一个页来保存修改数据，而父进程里面对应页面是没有变化的，这样无法在父子进程共享文件的变化。

spawn 的流程如下：

- 打开文件，获取文件描述符 fd。
- 读取 ELF 头部，检查 ELF 文件魔数。
- 调用 sys_exofork() 创建一个子进程。
- child_tf 设置，主要是设置了 eip 为 ELF 文件的入口点 e_entry，设置 esp 为 init_stack()分配的栈空间。
- 最后将 ELF 文件映射到子进程的地址空间，并根据 ELF 的读写段来设置读写权限。
- 拷贝共享的页。
- 调用 sys_env_set_trapframe()设置子进程的 env_tf 位 child_tf。
- 调用 sys_env_set_status() 设置子进程为 RUNNABLE 状态。

之前的 fork 在 duppage 时拷贝代码空间是以程序代码的 end 为结束的，现在看来这是有问题的，因为文件系统映射的地址并不在其中，需要将 end 改为 USTACKTOP-PGSIZE。此外，在 duppage 和 spawn.c 中的 copy_shared_pages 中要对 PTE_SHARE 做处理，直接映射即可，权限要用

PTE_SYSCALL，因为文件系统相关的页权限都是用的 PTE_SYSCALL，否则会检查失败。

Exercize 7

完成 sys_env_set_trapframe()。

```
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 *e;
        int r;
        if ((r = envid2env(envid, &e, 1)) < 0) {
                return -E_BAD_ENV;
        }
        tf->tf_eflags = FL_IF;
        tf->tf_cs = GD_UT | 3;
        e->env_tf = *tf;
        return 0;
}
```

```
        case SYS_env_set_status:ret = sys_env_set_status(a1, a2);
                                        break;
        case SYS_env_set_trapframe:
                                ret = sys_env_set_trapframe(a1, (struct Trapframe *)a2);
                                        break;
```

Exercize 8

修改 `lib/fork.c` 和 `lib/spawn.c`，支持共享页面。这里修复之前映射的一个问题，之前是以进程 end 作为映射结束位置，为了支持文件系统，改成 USTACKTOP-PGSIZE。

```
duppage(envid_t envid, unsigned pn)
{
        int r;
        void *addr = (void *)(pn * PGSIZE);
        if (uvpt[pn] & PTE_SHARE) {
                // cprintf("dup share page :%d\n", pn);
                if ((r = sys_page_map(0, addr, envid, addr, PTE_SYSCALL)) < 0)
                        panic("duppage sys_page_map:%e", r);
        } else if (uvpt[pn] & (PTE_W|PTE_COW)) {
                if ((r = sys_page_map(0, addr, envid, addr, PTE_COW|PTE_U|PTE_P)) < 0)
                        panic("sys_page_map COW:%e", r);
                if ((r = sys_page_map(0, addr, 0, addr, PTE_COW|PTE_U|PTE_P)) < 0)
                        panic("sys_page_map COW:%e", r);
        } else {
                if ((r = sys_page_map(0, addr, envid, addr, PTE_U|PTE_P)) < 0)
                        panic("sys_page_map UP:%e", r);
        }
        return 0;
```

```
static int
copy_shared_pages(envid_t child)
{
        // LAB 5: Your code here.
        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)) {
                        cprintf("copy shared page %d to env:%x\n", PGNUM(addr), child);
                    sys_page_map(0, (void*)addr, child, (void*)addr, (uvpt[PGNUM(addr)] & PTE_SYSCALL));
                }
        }
        return 0;
}
```

4 键盘接口

用户键盘输入会产生键盘中断 IRQ_KBD(通过 QEMU 图形界面输入触发)或者串口中断 IRQ_SERIAL(通过 console 触发)，为此要在 trap.c 中处理这两个中断，分别调用 kbd_intr() 和 serial_intr() 即可。其实在 cons_getc()中调用了 kbd_intr() 和 serial_intr() 这两个函数，因此在我们之前的实验内核的 monitor 中已经屏蔽了中断，一样可以读取到键盘输入。

Exercize 9

处理键盘和串口中断，这里即便去掉 kbd_intr() 和 serial_intr() 也不会有影响，因为 cons_getc()有调用它们。`kern/trap.c/trap_dispatch(struct Trapframe *tf)`

```
// Handle keyboard and serial interrupts.
// LAB 5: Your code here.
if (tf->tf_trapno == IRQ_OFFSET + IRQ_KBD) {
        kbd_intr();
        return;
}
if (tf->tf_trapno == IRQ_OFFSET + IRQ_SERIAL) {
        serial_intr();
        return;
}
```

输入/输出重定向

输入重定向：跟 Linux 一样，使用 < 语法。实现原理就是使用 dup(fd，0) 将文件描述符拷贝到标准输入 0，然后关闭 fd，最后从标准输入中读取文件内容即可。如 `cat < script` 便是先将 script 文件重定向到标准输入 0，最后 spawn 执行 cat 从标准输入读取内容并输出到标准输出。而如果使用 `sh < script`，又是不同的，此时 spawn 进程执行的是 sh 进程，它会先读取 script 文件内容，然后对 script 文件内容一行行命令 spawn 执行。

输出重定向：使用 > 语法。实现原理就是使用 dup(fd, 1)将文件描述符拷贝到标准输出 1，然后关闭 fd，这样输出到标准输出就相当于输出到文件了。如 `echo haha > motd`，会将 motd 文件内容改为 haha。

管道

JOS 管道实现在 `lib/pipe.c`，它分配两个文件描述符作为管道输入输出端，设备类型为管道，对应的数据页部分映射到了同样的物理页，只是设置的文件描述符的权限不同，pipe[0]对应的文件描述符为只读，而 pipe[1]可写。然后 fork()创建一个子进程，子进程中将 pipe[0] 拷贝到标准输入，然后重新读取输入运行管道右边的命令。父进程中则是将 pipe[1] 拷贝到标准输出。父进程会

先 spawn 运行左边命令，输出会重定向到标准输出，即 pipe[1]这个 fd。而子进程接着从标准输入读取输入，也就是从 pipe[0]这个 fd 读取输入，然后输出结果。管道读写使用方法是 devpipe_read 和 devpipe_write，如果管道没有数据可读，则会 sys_yield() 调度其他进程先运行。

Exercize 10

修改 `user/sh.c` 支持输入重定向，参照输出重定向修改即可。

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

Make grade 后:

```
internal FS tests [fs/test.c]: OK (2.6s)
  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 (2.7s)
  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 (1.3s)
PTE_SHARE [testpteshare]: OK (1.7s)
PTE_SHARE [testfdsharing]: OK (1.4s)
start the shell [icode]: OK (1.9s)
testshell: OK (4.5s)
```