به نام خدا

گزارش Lab5 سیستم عامل پیشرفته

استاد:

دکتر جوادی

دانشجویان:

سید علیرضا حسینی – ۴۰۱۱۳۱۰۰۵

امیررضا زارع – ۴۰۱۱۳۱۰۰۸

فهرست مطالب

مقدمه

در این آزمایش، ما یک فایل سیستم ساده مبتنی بر دیسک را پیاده سازی خواهیم کرد. خود سیستم فایل به صورت میکرو کرنل، خارج از هسته اما در محیط فضای کاربر خودش پیاده سازی خواهد شد. سایر محیط‌ها با درخواست IPC به این محیط فایل سیستم خاص به سیستم فایل دسترسی پیدا می‌کنند. ما spawn را پیاده سازی خواهیم کرد، یک فراخوان کتابخانه‌ای که فایل‌های اجرایی روی دیسک را بارگیری و اجرا می‌کند. سپس سیستم عامل هسته و کتابخانه خود را به اندازه کافی برای اجرای یک پوسته بر روی کنسول تکمیل می‌کنیم. جزء جدید اصلی برای این بخش از آزمایش، محیط فایل سیستم است که در فهرست جدید fs قرار دارد. در این آزمایش یک سری فایل جدید اضافه شده است که در شکل تصویر زیر مشاهده می‌کنید:

`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.

تمرین ۱:

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`.

سوال ۱.۱: آیا باید کار دیگری انجام دهید تا مطمئن شوید که این تنظیمات امتیاز ورودی/خروجی ذخیره شده و به درستی بازیابی می‌شود، زمانی که متعاقباً از یک محیط به محیط دیگر سوئیچ می‌کنید؟ چرا؟

پاسخ:

نیازی نیست، زیرا با تغییر محیط، مقدار eflag ها ذخیره می‌شود و مقدار eflag ها نیز tfrestored_pop_env می‌شود. تنظیمات امتیاز I/O بخشی از ثبت eflags است که به طور جداگانه برای هر محیط ذخیره می‌شود. بنابراین، اگر، برای مثال، از یک محیط دارای امتیاز I/O به یک محیط بدون I/O سوئیچ کنیم، رجیستر eflags دوباره بارگذاری می‌شود و محیط جدید و غیرمجاز نمی‌تواند به دستگاه‌های I/O دسترسی پیدا کند.

کاری که باید انجام دهیم این است که این خط ساده را به env_create() اضافه کنیم. در اینجا ما همچنین می‌توانیم FL_IOPL_3 را تنظیم کنیم زیرا محیط فایل سیستم یک محیط کاربری ویژه است.

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

        struct Env *e;
        int rc;
        if((rc = env_alloc(&e, 0)) != 0) {
            panic("env_create failed: env_alloc failed.\n");
        }
        // 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) {
                e->env_tf.tf_eflags |= FL_IOPL_MASK;
        }

        load_icode(e, binary);
        e->env_type = type;

}

//------------------------------------------------------------------------------
```

حال نتیجه این تمرین را در تصویر زیر مشاهده می‌کنید.

```
jos@Zare-Hosseini: ~/Desktop/jos-final/jos-lab5
+ cc[USER] user/ls.c
+ ld obj/user/ls
+ cc[USER] user/lsfd.c
+ ld obj/user/lsfd
+ cc[USER] user/num.c
+ ld obj/user/num
+ cc[USER] user/sh.c
+ ld obj/user/sh
+ mk obj/fs/clean-fs.img
+ cp obj/fs/clean-fs.img obj/fs/fs.img
make[1]: Leaving directory `/home/jos/Desktop/jos-final/jos-lab5'
internal FS tests [fs/test.c]: fatal: Not a git repository (or any of the parent
 directories): .git
fatal: Not a git repository (or any of the parent directories): .git
(1.3s)
  fs i/o: OK
```

تمرین ۲:

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".

موارد خواسته در این تمرین پیاده سازی شدند.

پیاده سازی bc_pgfault :

```
        // ------------------------------------------------------------------
        // LAB 5: you code here:

        int temp;
        addr = ROUNDDOWN(addr, PGSIZE);
        sys_page_alloc(0, addr, PTE_W|PTE_U|PTE_P);
        temp = ide_read(blockno * BLKSECTS, addr, BLKSECTS);
        if(temp<0){
                panic("ide_read: %e", temp);
        }

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


        // ------------------------------------------------------------------
```

پیاده سازی flush_block :

```
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.
        int temp;
        addr = ROUNDDOWN(addr, PGSIZE);//hint
        if(!va_is_mapped(addr) || !va_is_dirty(addr))return;
        if((temp = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0){
                panic("flush_block: ide_write failed! %e\n", temp);
                }
        if((temp = sys_page_map(0,addr,0,addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0){
                panic("flush_block: sys_page_map failed! %e\n",temp);
                }

}


//------------------------------------------------------------------------------
```

خروجی make grade برای این تمرین در تصویر زیر قابل مشاهده است.

```
jos@Zare-Hosseini: ~/Desktop/jos-final/jos-lab5
+ mk obj/kern/kernel.img
+ mk obj/fs/fsformat
+ cc[USER] user/init.c
+ ld obj/user/init
+ cc[USER] user/cat.c
+ ld obj/user/cat
+ cc[USER] user/echo.c
+ ld obj/user/echo
+ cc[USER] user/ls.c
+ ld obj/user/ls
+ cc[USER] user/lsfd.c
+ ld obj/user/lsfd
+ cc[USER] user/num.c
+ ld obj/user/num
+ cc[USER] user/sh.c
+ ld obj/user/sh
+ mk obj/fs/clean-fs.img
+ cp obj/fs/clean-fs.img obj/fs/fs.img
make[1]: Leaving directory `/home/jos/Desktop/jos-final/jos-lab5'
internal FS tests [fs/test.c]: fatal: Not a git repository (or any of the parent
 directories): .git
fatal: Not a git repository (or any of the parent directories): .git
(2.0s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
```

تمرین ۳:

Exercise 3. Use `free_block` as a model to implement `alloc_block`, 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".

alloc_block() یک بلوک جدید را به سیستم فایل اختصاص می‌دهد. از آنجایی که سیستم فایل از آرایه‌ای از بیت‌ها به نام بیت مپ برای ضبط استفاده از بلوک استفاده می‌کند، باید قسمت مربوطه بیت مپ را به‌روزرسانی کنیم و کش آن را پاک کنیم.

```
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.
        //search bitmap
        uint32_t temp;
        int temp_mod;
        int temp_div;
        for(temp = 3; temp < super->s_nblocks; temp=temp+1){
                if(block_is_free(temp) ==1 ){
                        temp_mod = temp%32;
                        temp_div = temp/32;

                        bitmap[temp_div] &= ~(1 << (temp_mod));
                        flush_block(&bitmap[temp_div]);
                        return temp;
                }
        }
        return -E_NO_DISK;
}

//------------------------------------------------------------------------
```

خروجی make grade برای این تمرین را در تصویر زیر مشاهده می‌کنید.

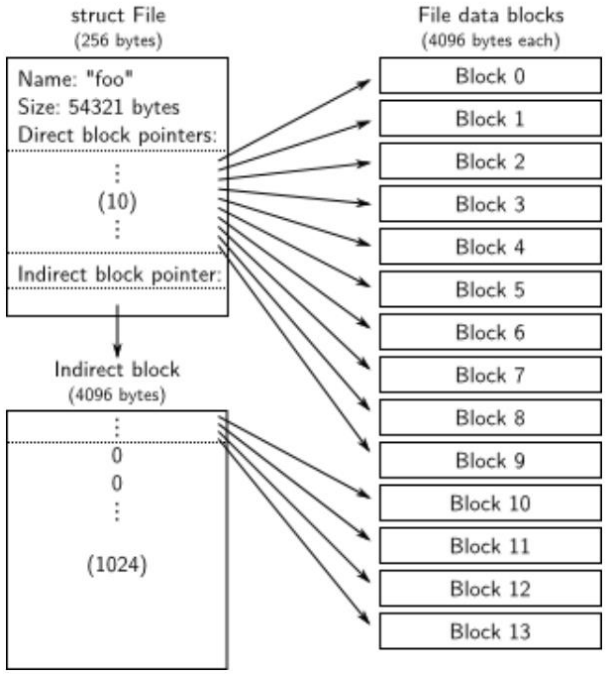
```
jos@Zare-Hosseini: ~/Desktop/jos-final/jos-lab5
+ cc[USER] user/ls.c
+ ld obj/user/ls
+ cc[USER] user/lsfd.c
+ ld obj/user/lsfd
+ cc[USER] user/num.c
+ ld obj/user/num
+ cc[USER] user/sh.c
+ ld obj/user/sh
+ mk obj/fs/clean-fs.img
+ cp obj/fs/clean-fs.img obj/fs/fs.img
make[1]: Leaving directory `/home/jos/Desktop/jos-final/jos-lab5'
internal FS tests [fs/test.c]: fatal: Not a git repository (or ar
 directories): .git
fatal: Not a git repository (or any of the parent directories):
(1.1s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
  alloc_block: OK
```

تمرین ۴:

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".

قبل از انجام این تمرین، بهتر است نگاهی به نحوه ذخیره یک فایل در سیستم فایل در File Meta-data صفحه آزمایش بیندازیم. ساختار File اطلاعات فایل و همچنین نشانگرهای ۱۰ بلوک اول را نگهداری می‌کند. اگر فایل فضای بیشتری اشغال کند، نشانگر بلوک غیرمستقیم به بلوکی اشاره می‌کند که نشانگرهای بلوک‌های زیر را نگه می‌دارد و می‌تواند تا ۱۰۲۴ اشاره‌گر را در خود جای دهد.



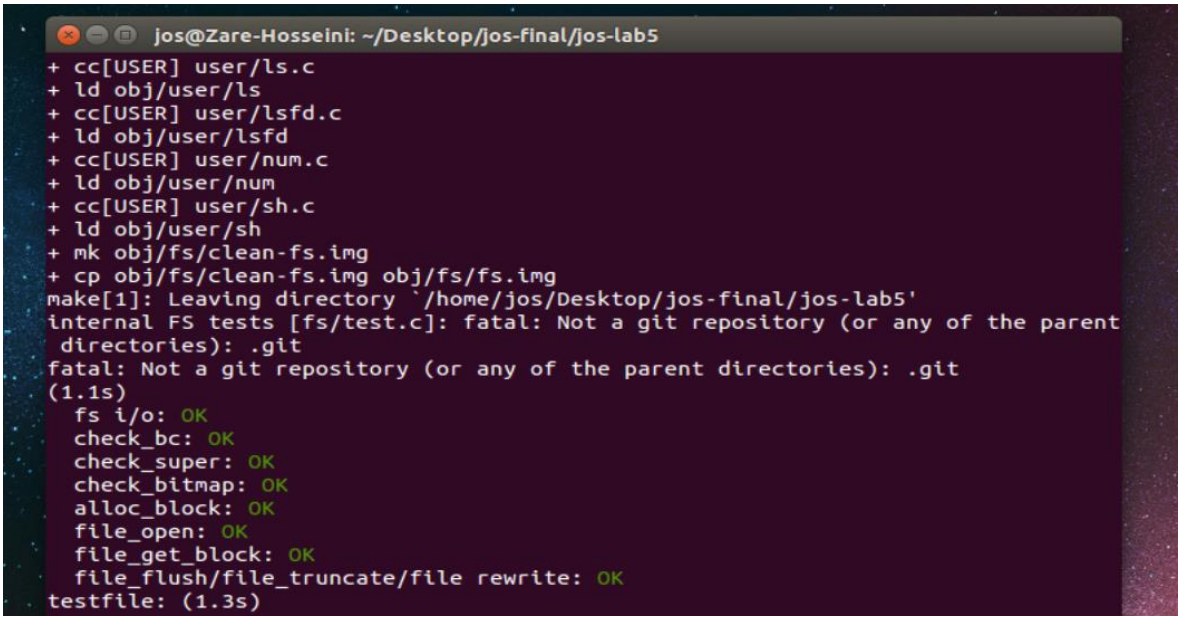
با اطلاعات بالا می‌توان این دو عملکرد را به راحتی پیاده سازی کرد. file_block_walk بلوک مشخص شده یک فایل را پیدا می‌کند که کاملاً شبیه به pgdir_walk است. اگر یک بلوک جدید را به عنوان یک بلوک غیرمستقیم اختصاص دهیم، چون بلوک جدید ممکن است کثیف باشد، باید آن را با ۰ پر کنیم.

کد پیاده سازی file_block_walk و file_get_block را در تصویر زیر مشاهده می‌کنید.

```
static int file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc) {
//----------------------------------------------------------------------------------------------------------
        // LAB 5: Your code here.
        //3 bad walk

        if(filebno >= NDIRECT + NINDIRECT){
                return -E_INVAL;
        }
        if(filebno < NDIRECT){
                *ppdiskbno = f->f_direct + filebno;
        }else{
                if(alloc && (f->f_indirect == 0)){
                        int temp;
                        temp = alloc_block();
                        if(temp < 0){
                                return temp;
                        }
                        memset(diskaddr(temp), 0, BLKSIZE);
                        f->f_indirect = temp;
                }else if(f->f_indirect == 0){
                        return -E_NOT_FOUND;
                }
                //set *ppdiskbno
                *ppdiskbno = ((uint32_t*)diskaddr(f->f_indirect)) + filebno - NDIRECT;//diskaddr return char*
        }

                //return 0 success
        return 0;


}
//----------------------------------------------------------------------------------------------------------
```

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

//----------------------------------------------------------------------------------------------
        // LAB 5: Your code here.
        uint32_t *pdiskbno;
        int temp;
        temp = file_block_walk(f, filebno, &pdiskbno, 1);
        if(temp<0){
                return temp;
        }

        if(*pdiskbno == 0){

                temp = alloc_block();
                if(temp<0){
                        return temp;
                }

                *pdiskbno = temp;
        }

        *blk = (char*)diskaddr(*pdiskbno);
        flush_block(*blk);
        return 0;
}

//----------------------------------------------------------------------------------------------
```

خروجی make grade را برای این تمرین در تصویر زیر مشاهده می‌کنید.



تمرین ۵:

Exercise 5. Implement `serve_read` in `fs/serv.c` and `devfile_read` in `lib/file.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_stat` to get a general idea of how the server functions should be structured.

Likewise, `file_read` should pack its arguments into `fsipcbuf` for `serve_read`, call `fsipc`, and handle the result.

Your code should pass "serve_open/file_stat/file_close" and "file_read".

از آنجایی که محیط فایل سیستم یک محیط کاربری ویژه است، سایر محیط‌های کاربری باید از مکانیزم IPC (ارتباط بین فرآیندی) برای انجام عملیات فایل استفاده کنند، بنابراین سیستم فایل یک سرور را فراهم می‌کند. برنامه‌های کاربردی کاربر در JOS از یک توصیفگر فایل برای دسترسی به فایل استفاده می‌کنند و با ارسال درخواست باز به سرور فایل ایجاد می‌شود. سرور فایل از ساختار OpenFile برای ضبط یک فایل باز شده استفاده می‌کند و این رکوردها در opentab ذخیره می‌شوند. قبل از انجام عملیات فایل مانند خواندن یا نوشتن، ابتدا فایل باید باز شود. سرور همچنین از یک اتحادیه Fsipc برای نگهداری پیام IPC برای درخواست‌های عملیات فایل استفاده می‌کند. در inc/fs.h تعریف شده است. اکنون می‌توانیم serve_read را پیاده سازی کنیم. بررسی می‌کند که آیا فایل درخواستی باز شده است، فایل را می‌خواند، موقعیت جستجو را به‌روزرسانی می‌کند و پاسخی را ارسال می‌کند.

توابع serve_read و devfile_read را در تصاویر زیر قابل مشاهده است.

```
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 temp;
        temp = openfile_lookup(envid, req->req_fileid, &o);
        if(temp<0){
                return temp;
        }

        temp = file_read(o->o_file, ret->ret_buf, req->req_n, o->o_fd->fd_offset);
        if(temp<0){
                return temp;
        }

        o->o_fd->fd_offset = temp + o->o_fd->fd_offset;
        return temp;
}

//------------------------------------------------------------------------------------------
```

```
static ssize_t devfile_read(struct Fd *fd, void *buf, size_t n) {
        // Make an FSREQ_READ request to the file system server after
        // filling fsipcbuf.read with the request arguments.  The
        // bytes read will be written back to fsipcbuf by the file
        // system server.
        int temp;

        fsipcbuf.read.req_fileid = fd->fd_file.id;
        fsipcbuf.read.req_n = n;
        if ((temp = fsipc(FSREQ_READ, NULL)) < 0)
                return temp;
        assert(temp <= n);
        assert(temp <= PGSIZE);
        memmove(buf, fsipcbuf.readRet.ret_buf, temp);
        return temp;
}
```

خروجی دستور make grade را برای این تمرین در تصویر زیر مشاهده می‌کنید.

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

تمرین ۶:

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".

serve_write مشابه serve_read است، با این تفاوت که پاسخ آن فقط حاوی یک مقدار است. devfile_write درخواست نوشتن را آماده می‌کند و به سرور ارسال می‌کند. در اینجا درخواست و پاسخ هر دو باید بررسی شوند. همانطور که تعریف ساختار Fsreq_write نشان می‌دهد req_buf هرگز نباید از (sizeof(int) + sizeof(size_t)) - PGSIZE تجاوز کند و مقدار پاسخ باید بالاتر از صفر باشد.

موارد خواسته شده پیاده سازی شده‌اند. کد serve_write و devfile_write را در تصاویر زیر مشاهده می‌کنید.

```
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 temp;
        temp = openfile_lookup(envid, req->req_fileid, &o);
        if(temp<0){
                return temp;
        }
        temp = file_write(o->o_file, req->req_buf, req->req_n, o->o_fd->fd_offset);
        if(temp<0){
                return temp;
        }
        o->o_fd->fd_offset = temp + o->o_fd->fd_offset;
        return temp;
}
//-------------------------------------------------------------------------------------------
```

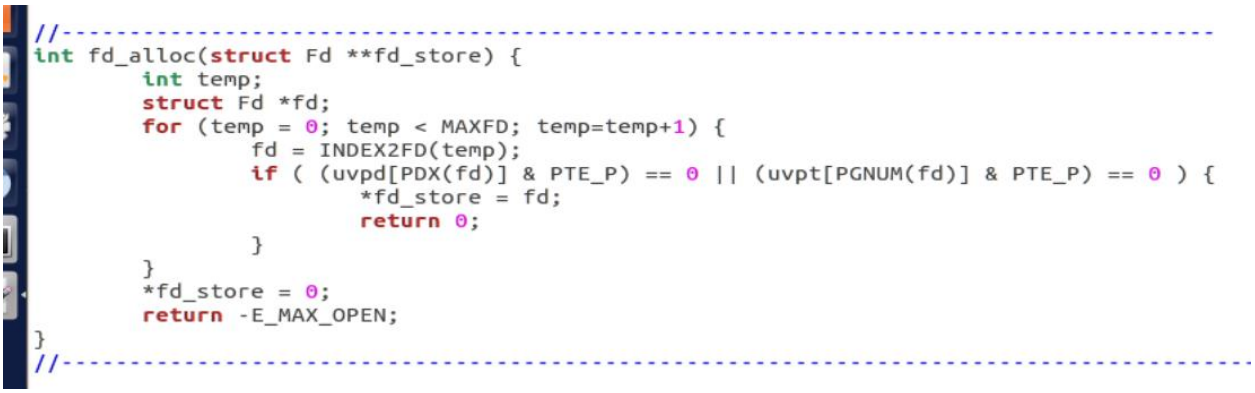
```
//-------------------------------------------------------------------------------
static ssize_t devfile_write(struct Fd *fd, const void *buf, size_t n) {
        // Make an FSREQ_WRITE request to the file system server.  Be
        // careful: fsipcbuf.write.req_buf is only so large, but
        // remember that write is always allowed to write *fewer*
        // bytes than requested.
        // LAB 5: Your code here
        int temp;

        fsipcbuf.write.req_fileid = fd->fd_file.id;
        fsipcbuf.write.req_n = n;
        memmove(fsipcbuf.write.req_buf, buf, n);
        if ((temp = fsipc(FSREQ_WRITE, NULL)) < 0){
                return temp;
        }
        assert(temp <= n);
        assert(temp <= PGSIZE);
        return temp;
}
//-------------------------------------------------------------------------------
```

تمرین ۷:

Exercise 7. Implement `open`. The `open` function must find an unused file descriptor using the `fd_alloc()` function we have provided, make an IPC request to the file system environment to open the file, and return the number of the allocated file descriptor. Be sure your code fails gracefully if the maximum number of files are already open, or if any of the IPC requests to the file system environment fails.

Use `make grade` to test your code. Your code should pass all file system tests at this point.

موارد خواسته شده پیاده سازی شده‌اند. کد تابع fd_alloc را در تصویر زیر مشاهده می‌کنید.



```
//-------------------------------------------------------------------------------
int fd_alloc(struct Fd **fd_store) {
        int temp;
        struct Fd *fd;
        for (temp = 0; temp < MAXFD; temp=temp+1) {
                fd = INDEX2FD(temp);
                if ( (uvpd[PDX(fd)] & PTE_P) == 0 || (uvpt[PGNUM(fd)] & PTE_P) == 0 ) {
                        *fd_store = fd;
                        return 0;
                }
        }
        *fd_store = 0;
        return -E_MAX_OPEN;
}
//-------------------------------------------------------------------------------
```

تمرین ۸:

Exercise 8. `spawn` relies on the new syscall `sys_env_set_trapframe` to initialize the state of the newly created environment. Test your code by running the `user/spawnhello` program from `kern/init.c`, which will attempt to spawn `/bin/hello` from the file system.

Use `make grade` to test your code.

موارد خواسته شده انجام شد.

تمرین ۹:

Exercise 9. 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.

موارد خواسته شده انجام شد.

تمرین ۱۰:

Exercise 10. 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`.

موارد خواسته شده انجام شد.

```
//----------------------------------------------------------------------
        // LAB 5: Your code here.
        if(tf->tf_trapno == IRQ_OFFSET + IRQ_KBD){
                // handle trap IRQ_OFFSET+IRQ_KBD
                kbd_intr();
                return;
        }
        if(tf->tf_trapno == IRQ_OFFSET + IRQ_SERIAL){
                // handle trap IRQ_OFFSET+IRQ_SERIAL
                serial_intr();
                return;
        }
//----------------------------------------------------------------------
```

چالش ۲:

Challenge 2! (5 points) 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, you can track approximate usage of disk blocks without the need to modify every place in the code that accesses the disk map region.

چالش ۲ شامل افزودن یک سیاست تخلیه به حافظه پنهان بلوک در سیستم عامل JOS است. هدف معرفی مکانیزمی است که بر اساس برخی سیاست‌های تخلیه، بلوک‌ها را از حافظه پنهان حذف می‌کند، نه اینکه هر بلوک را به طور نامحدود در حافظه نگه دارد. رویکرد پیشنهادی استفاده از بیت‌های «دسترسی» PTE_A در جداول صفحه برای ردیابی استفاده تقریبی از بلوک‌های دیسک است.

در ادامه یک راهنمای کلی در مورد نحوه برخورد با این چالش آورده شده است:

بلاک کش موجود را بشناسید: با اجرای فعلی بلاک کش در JOS آشنا شوید. نحوه ذخیره و دسترسی به بلوک ها را درک کنید.

درباره بیت‌های PTE_A (دسترسی) بیاموزید: هدف و استفاده از بیت‌های PTE_A (دسترسی) در جداول صفحه را درک کنید. این بیت‌ها معمولاً برای ردیابی اینکه آیا به یک صفحه دسترسی پیدا کرده است استفاده می‌شود.

تعریف سیاست اخراج: یک خط مشی اخراج را انتخاب یا طراحی کنید که تعیین می‌کند در صورت نیاز به فضا، کدام بلوک‌ها باید از حافظه پنهان خارج شوند. خط مشی‌های رایج اخراج عبارتند از: حداقل اخیراً استفاده شده (LRU)، اخیراً استفاده شده (MRU)، یا گونه‌ای که از بیت‌های دسترسی استفاده می‌کند.

تغییر کد بلوک کش: کد حافظه پنهان بلوک را برای گنجاندن خط مشی اخراج انتخابی تغییر دهید. این ممکن است شامل افزودن ساختارهای داده برای ردیابی تاریخچه دسترسی بلوک و به روز رسانی منطق مدیریت حافظه پنهان باشد.

از بیت های PTE_A برای ردیابی استفاده کنید: از بیت‌های PTE_A برای ردیابی وضعیت دسترسی بلوک‌های دیسک استفاده کنید. کد را تغییر دهید تا این بیت‌ها را به طور مناسب بر اساس الگوهای دسترسی بلوک تنظیم و پاک کنید.

اجرای منطق اخراج: منطق اخراج را به گونه‌ای اجرا کنید که خط مشی اخراج انتخابی و اطلاعات ارائه شده توسط بیت‌های PTE_A را در نظر بگیرد. وقتی حافظه پنهان بلاک پر شد، تصمیم بگیرید کدام بلوک (های) را بیرون کنید.

تست و رفع اشکال: کش بلوک اصلاح شده را به طور کامل تست کنید. اطمینان حاصل کنید که خط مشی اخراج همانطور که در نظر گرفته شده است کار می‌کند و در صورت نیاز بلوک‌ها به طور مناسب از حافظه پنهان حذف می‌شوند.

چالش ۳:

Challenge! (20 points) The file system is likely to be corrupted if it gets interrupted in the middle of an operation (for example, by a crash or a reboot). Implement soft updates or journalling to make the file system crash-resilient and demonstrate some situation where the old file system would get corrupted, but yours doesn't.

پیاده‌سازی انعطاف‌پذیری خرابی در یک سیستم فایل از طریق تکنیک‌هایی مانند به‌روزرسانی‌های نرم‌افزار یا ژورنال کردن، یک کار پیچیده است که شامل طراحی دقیق و اصلاح کد سیستم فایل است. در ادامه یک راهنمای کلی در مورد نحوه برخورد با این چالش آورده شده است.

مراحل پیاده سازی Crash Resilience (به روز رسانی نرم یا ژورنالینگ):

سیستم فایل موجود را بشناسید

به‌روزرسانی‌های نرم‌افزاری تحقیق یا ژورنال‌نگاری

یک تکنیک را انتخاب کنید: تصمیم بگیرید که آیا می خواهید به روز رسانی های نرم افزاری را پیاده سازی کنید یا ژورنالینگ. هر کدام رویکرد خاص خود را برای مدیریت خرابی ها و بازیابی دارند.

اصلاح به‌روزرسانی‌های فراداده: نقاط بحرانی را در کد سیستم فایل که در آن به روز رسانی ابرداده رخ می‌دهد، شناسایی کنید. این نقاط را تغییر دهید تا تکنیک انعطاف پذیری تصادف انتخابی را در خود جای دهد. این ممکن است شامل ثبت تغییرات قبل از اعمال یا حفظ اطلاعات وابستگی باشد.

پیاده سازی مکانیسم بازیابی: یک مکانیسم بازیابی را پیاده‌سازی کنید که تضمین می کند سیستم فایل می‌تواند پس از یک خرابی به حالت ثابت بازگردد. این شامل بررسی وضعیت فایل سیستم هنگام راه اندازی و اعمال هرگونه اصلاحات لازم است.

آزمایش کردن: اجرای انعطاف پذیری تصادف را به طور گسترده آزمایش کنید. خرابی‌ها یا راه‌اندازی مجدد را در طول عملیات سیستم فایل شبیه‌سازی کنید تا مطمئن شوید که مکانیسم بازیابی همانطور که انتظار می رود کار می‌کند.

سناریوی نمایش: سناریویی ایجاد کنید که مزایای انعطاف پذیری تصادف اجرا شده را نشان دهد. این می تواند شامل انجام یک سری عملیات سیستم فایل، قطع کردن سیستم در وسط، و سپس نشان دادن ثابت ماندن سیستم فایل پس از بازیابی باشد.

نکات اضافی:

به روز رسانی نرم در مقابل ژورنالینگ: به‌روزرسانی‌های نرم و ژورنالینگ دارای معاوضه‌های متفاوتی هستند. به‌روزرسانی‌های نرم از نیاز به مجله اجتناب می‌کنند، اما ممکن است شامل ردیابی وابستگی پیچیده باشد. ژورنال نویسی بازیابی را ساده می‌کند، اما هزینه سربار نگهداری یک مجله را اضافه می‌کند.

اتمی بودن: اطمینان حاصل کنید که به‌روزرسانی‌های سیستم فایل اتمی هستند یا در صورت خرابی قابل برگشت هستند. این برای حفظ ثبات بسیار مهم است.

زمان ریکاوری: مدت زمان لازم برای بازیابی فایل سیستم پس از خرابی را در نظر بگیرید. برای تجربه کاربری بهتر، زمان بازیابی را به حداقل برسانید.

این چالش نیاز به درک عمیقی از سیستم های داخلی فایل و تکنیک های بازیابی خرابی دارد.

چالش ۴:

Challenge! (5 points) Change the file system to keep most file meta-data in Unix-style inodes rather than in directory entries, and add support for hard links.

این چالش شامل تغییر سیستم فایل در JOS برای نگهداری بیشتر فراداده های فایل در inodeهای سبک یونیکس به جای ورودی‌های دایرکتوری است. علاوه بر این، ما وظیفه اضافه کردن پشتیبانی برای پیوندهای سخت را داریم. در ادامه یک راهنمای کلی در مورد نحوه برخورد با این چالش آورده شده است.

مراحل پیاده سازی اینودها و پیوندهای سخت به سبک یونیکس:

درک ساختار فایل سیستم موجود: با ساختار سیستم فایل فعلی در JOS از جمله نحوه ذخیره ابرداده ها و نحوه مدیریت ورودی‌های دایرکتوری آشنا شوید.

تحقیق در مورد Inodeهای سبک یونیکس: Inodeهای سبک یونیکس را مطالعه کنید و بدانید که چگونه از آنها برای ذخیره ابرداده برای فایل ها استفاده می شود. Inode ها معمولاً حاوی اطلاعاتی مانند اندازه فایل، مجوزها، مُهرهای زمانی و نشانگرهای بلوک های داده هستند.

تغییر کد فایل سیستم: کد سیستم فایل را طوری تغییر دهید که inodeهای سبک یونیکس را در خود جای دهد. یک ساختار جدید برای نمایش inode ها ایجاد کنید و کد را برای استفاده از این ساختار برای ذخیره ابرداده فایل به روز کنید.

پیاده سازی لینک های سخت: پیوندهای سخت به چندین ورودی دایرکتوری اجازه می‌دهند به یک inode اشاره کنند. کد سیستم فایل را برای پشتیبانی از ایجاد پیوندهای سخت و به روز رسانی تعداد مراجع در inodes تغییر دهید.

کنترل تعداد حذف و مراجع: پیاده سازی منطق برای مدیریت ایجاد و حذف لینک های سخت. هنگامی که یک پیوند سخت ایجاد می شود، تعداد مراجع را در inode مربوطه به روز کنید. هنگامی که یک فایل حذف می شود، تعداد مراجع را کاهش دهید، و تنها زمانی که تعداد مراجع به صفر رسید، منابع را آزاد کنید.

به روز رسانی ورودی های دایرکتوری: کد مسئول مدیریت ورودی های دایرکتوری را به گونه ای تغییر دهید که به جای ذخیره کردن تمام ابرداده ها مستقیماً در ورودی های دایرکتوری، از inode استفاده کند. اکنون ورودی های دایرکتوری به جای تکرار ابرداده باید شامل شماره inode باشند.

آزمایش کردن: کد فایل سیستم اصلاح شده را به طور کامل تست کنید. فایل‌ها، پیوندهای سخت و دایرکتوری‌ها را ایجاد کنید تا اطمینان حاصل کنید که تغییرات به‌درستی inodeهای سبک یونیکس و پیوندهای سخت را مدیریت می‌کنند.

سناریوی نمایش: سناریویی ایجاد کنید که استفاده از پیوندهای سخت و تفکیک ابرداده ها را به inodeهای سبک یونیکس نشان دهد. مزایای این رویکرد را از نظر انعطاف پذیری و اشتراک منابع به نمایش بگذارید.

نکات اضافی:

ساختار inode: ساختار inode های خود را طوری طراحی کنید که شامل اطلاعات لازم مانند اندازه فایل، مجوزها، مُهرهای زمانی و نشانگرهای بلوک های داده باشد.

تعداد مراجع: برای مدیریت صحیح پیوندهای سخت، تعداد مراجع را در اینودها پیگیری کنید. هنگام ایجاد یا حذف پیوندهای سخت، تعداد مراجع را به روز کنید.

ثبات: اطمینان حاصل کنید که سیستم فایل پس از عملیاتی که شامل پیوندهای سخت، حذف ها و تغییرات در inode ها می شود، ثابت می ماند.

این چالش شامل تغییرات قابل توجهی در ساختار و رفتار سیستم فایل است، بنابراین آزمایش کامل و اجرای دقیق ضروری است.

خروجی make grade **و نمره نهایی تمرین**

ما یک بخش از آزمایش را که **بخشی از تمرین ۶** است را به دلیل نتوانستیم انجام دهیم، برای اینکه دستور make grade بتواند درست اجرا شود و نمره دهد، کدی که به آن بخش نمره میداد را **کامنت** کردیم. برای همین **از ۱۸ حساب شده است**. در واقع نمره ما **۱۸ از ۲۰** هست. خروجی دستور make grade را در تصویر زیر مشاهده می‌کنید.