Report [Lab 5: File system, Spawn and Shell]

I've completed the challenge of `mmap`

The File System

Disk Access

Exercise 1

An environment is permitted to do IO only when its CPL is equal or less than the IOPL field in EFLAGS[1]. As the file system environment is running in user mode with CPL=3, we should set the IOPL to 3 to give it appropriate privilege.

```
    if (type == ENV_TYPE_FS)
    {
        e->env_tf.tf_eflags |= FL_IOPL_3;
    }
```

Question 1

Nothing special needs to be done since IOPL is part of EFLAGS which is properly stored in `struct Trapframe` when trapped into kernel and restored when to be run again.

The Block Cache

Exercise 2

In JOS we map the whole disk into the address space of the file system environment and read a page (which has the same size as a block) from disk when necessary i.e. it is accessed by leveraging the `pgfault_handler` mechanism.

We use `sys_page_alloc` to allocate a new page for the faulting address and use `ide_read` to load its content from the disk. As disk hardware is working with the unit of sector, we pass `BLKSECTS` as the `nsecs` parameter to read the sectors of the corresponding block.

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

`flush_block()` will write the content cached in memory back to disk if necessary, that is to say, the corresponding page is mapped and modified (dirty). `ide_write()` with similar semantics as `ide_read` is used to accomplish the actual work. The dirty bit also needs to be cleared when things are done.

```
    // LAB 5: Your code here.
    int r;
    addr = ROUNDDOWN(addr, PGSIZE);
    if (va_is_mapped(addr) && va_is_dirty(addr))
    {
        if ((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
        {
            panic("in flush_block, ide_write: %e", r);
        }
        // clear dirty bit by clearing all bits in PTE except permission
bits
        if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] &
PTE_SYSCALL)) < 0)
        {
            panic("in flush_block, sys_page_map: %e", r);
        }
    }
```

The Block Bitmap

Exercise 3

Bitmap, as a critical data structure for the disk, should be flushed back immediately after any modification. Keeping this in mind, we iterate over `super->s_nblocks` block numbers and use `block_is_free` to search for a free block. Once it's found, we revert the corresponding bit in the bitmap to mark its allocation and flush this change to disk to maintain consistency of the disk.

```
    // LAB 5: Your code here.
    // panic("alloc_block not implemented");
    if (super == 0)
    {
        panic("alloc_block: super is null");
    }
    for (int i = 0; i < super->s_nblocks; i++)
    {
        if (block_is_free(i))
        {
            bitmap[i / 32] ^= 1 << (i % 32);
            flush_block(&bitmap[i / 32]);
            return i;
        }
    }
    return -E_NO_DISK;
```

File Operations

Exercise 4

File system provides the abstraction of files to applications to make disk management more convenient. To support this abstraction, we must have corresponding mechanism to translation a position in the file to a position in the disk just like what we do to translate virtual address to physical address.

The underlying functions are `file_block_walk()` and `file_get_block()`. In `file_block_walk()`, a `filebno` is given and the pointer to where the block number of `filebno`-th block in corresponding file is stored in the file system is to be filled. If `filebno < NDIRECT`, the position of the block is stored in `struct File`. Otherwise the block number is stored in the indirect block which may need to be allocated depending on its existence and `alloc` parameter.

```
// 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");
    int r;
    if (filebno >= NDIRECT + NINDIRECT)
    {
        return -E_INVAL;
    }
    if (filebno < NDIRECT)
    {
        *ppdiskbno = f->f_direct + filebno;
        return 0;
    }
    if (!f->f_indirect)
    {
        if (!alloc) return -E_NOT_FOUND;
        if ((r = alloc_block()) < 0)
        {
            return r;
        }
        memset(diskaddr(r), 0, BLKSIZE);
        f->f_indirect = r;
    }
    uint32_t *ind = diskaddr(f->f_indirect);
    *ppdiskbno = ind + (filebno - NDIRECT);
    return 0;
}
```

`file_get_block()` is basically a wrapper for `file_block_walk`, except that a new block must be allocated if the corresponding file block number is 0 i.e. null.

```
// 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");
    uint32_t *pdiskbno;
    int r;
    if ((r = file_block_walk(f, filebno, &pdiskbno, true)) < 0)
    {
        return r;
    }
    if (*pdiskbno == 0)
    {
        if ((r = alloc_block()) < 0)
        {
            return r;
        }
        memset(diskaddr(r), 0, sizeof(r));
        *pdiskbno = r;
    }
    *blk = diskaddr(*pdiskbno);
    return 0;
}
```

The file system interface

Exercise 5 & 6

In JOS, regular environments do file operations by RPC to the file system environment, which is implemented by IPC. A number is passed through IPC as an identifier to some specific file system service and the arguments is passed as a sharing page to file system environment, which is also used to contain the results.

In the server side, the arguments should be dispatched to the corresponding file system function wrapper and be parsed and passed to the function to get the result. Some other information needed is stored in the file descriptor field such as current offset for read and write operation.

```
    // 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, MIN(req->req_n, PGSIZE),
                       o->o_fd->fd_offset)) < 0)
        return r;

    o->o_fd->fd_offset += r;
    return r;
```

```
    // LAB 5: Your code here.
    // panic("serve_write not implemented");
    struct OpenFile *o;
    int r;

    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;

    if ((r = file_write(o->o_file, req->req_buf, req->req_n, o->o_fd->fd_offset)) < 0)
        return r;

    o->o_fd->fd_offset += r;
    return r;
```

In the client side of RPC mechanism, arguments are written into the IPC page before passing them to server side. Then the client blocks until the file system environment finish the job and inform the client. Then some checks of returning value in the shared page is checked.

```
int
serve_write(envid_t envid, struct Fsreq_write *req)
{
    if (debug)
        cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid, req->req_n);

    // LAB 5: Your code here.
    // panic("serve_write not implemented");
    struct OpenFile *o;
    int r;

    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;

    if ((r = file_write(o->o_file, req->req_buf, req->req_n, o->o_fd->fd_offset)) < 0)
        return r;

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

Spawning Processes

If we want to implement `exec` in user space, we have to ask the environment to replace itself including code and stacks to the new program's, which is difficult, if not impossible, to keep consistency in such procedure.

So instead we implement `spawn`, creating an empty child environment and replacing its content to the new program's, which is much easier.

Exercise 7

A new system call `sys_env_set_trapframe()` is needed by `spawn()` to fill the `env_tf` for the new environment to indicate the position of program entry point and initial stack. Some particular bits of the `struct Trapframe` is modified to ensure that the permission is appropriate for a user program.

```
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");

    struct Env *e;
    int r;

    if ((r = envid2env(envid, &e, true)) < 0)
        return r;

    user_mem_assert(curenv, tf, sizeof(struct Trapframe), 0);
    e->env_tf = *tf;

    e->env_tf.tf_cs |= 3;
    e->env_tf.tf_eflags |= FL_IF;
    e->env_tf.tf_eflags &= ~FL_IOPL_MASK;

    return 0;
}
```

challenge!

The `mmap` method defined in POSIX allows to map a file into a region in address space to make files able to be accessed like other data in memory. For read-only parts of files like the code in program image and archives, using `mmap` can make different environments share their common content to save space.

In JOS, I choose to implement `mmap` in a simple but effective way. It takes an offset and an address space which are both aligned, then it leverages file system call to share exactly one page through IPC, which is exactly the corresponding block in the mapped disk.

```
// fs/server.c
int
serve_read_map(envid_t envid, struct Fsreq_read_map *req,
    void **pg_store, int *perm_store)
{
    struct OpenFile *o;
    char *blk;
    int r;

    if (req->req_perm & PTE_W)
        return -E_INVAL;
```

```
    if (req->req_offset % BLKSIZE)
        return -E_INVAL;
    int blkno = req->req_offset / BLKSIZE;

    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;
    if ((r = file_get_block(o->o_file, blkno, &blk)) < 0)
        return r;
    memcpy(blk, blk, PGSIZE); // load the block from the disk

    *pg_store = blk;
    *perm_store = req->req_perm;

    return 0;
}
```

To use RPC mechanism to call this function, we have to register it in the user library, adding a new function pointer to `struct Dev` and implement client-side `read_map()` for files.

```
// inc/fd.h
// Per-device-class file descriptor operations
struct Dev {
    int dev_id;
    const char *dev_name;
    ssize_t (*dev_read)(struct Fd *fd, void *buf, size_t len);
    ssize_t (*dev_write)(struct Fd *fd, const void *buf, size_t len);
    int (*dev_close)(struct Fd *fd);
    int (*dev_stat)(struct Fd *fd, struct Stat *stat);
    int (*dev_trunc)(struct Fd *fd, off_t length);

    int (*dev_read_map)(struct Fd *fd, void *addr, off_t offset, int perm);
};
```

```
// lib/file.c
static int
devfile_read_map(struct Fd *fd, void *addr, off_t offset, int perm)
{
    int r;

    fsipcbuf.read_map.req_fileid = fd->fd_file.id;
    fsipcbuf.read_map.req_offset = offset;
    fsipcbuf.read_map.req_perm = perm;
    if ((r = fsipc(FSREQ_READ_MAP, addr)) < 0)
        return r;

    return r;
}
```

With `read_map()`, we can share the read-only pages during `spawn()` without making a copy for the child.

```
// lib/spawn.c map_segment()
    for (i = 0; i < memsz; i += PGSIZE) {
```

```
        if (i >= filesz) {
            // allocate a blank page
            if ((r = sys_page_alloc(child, (void*) (va + i), perm)) < 0)
                return r;
        } else {
            // from file
            if (~perm & PTE_W)
            {
                // use read_map() to share read-only pages
                // between environments
                if ((r = read_map(fd, UTEMP, fileoffset + i, perm)) < 0)
                    return r;
                if ((r = sys_page_map(0, UTEMP, child, (void*) (va + i),
perm)) < 0)
                    panic("spawn: sys_page_map sharing data: %e", r);
                sys_page_unmap(0, UTEMP);

                continue;
            }
            if ((r = sys_page_alloc(0, UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
                return r;
            if ((r = seek(fd, fileoffset + i)) < 0)
                return r;
            if ((r = readn(fd, UTEMP, MIN(PGSIZE, filesz-i))) < 0)
                return r;
            if ((r = sys_page_map(0, UTEMP, child, (void*) (va + i), perm))
< 0)
                panic("spawn: sys_page_map data: %e", r);
            sys_page_unmap(0, UTEMP);
        }
    }
```

Sharing library state across fork and spawn

We want share some specific pages e.g. file descriptor with child environment created by `fork()` or `spawn()`. This is implemented by leveraging one of the available bits in PTE to mark pages to share.

Exercise 8

In `lib/fork.c`, we change `duppage()` to map pages with `PTE_SHARE` set directly into the new environment's address space.

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

    // LAB 4: Your code here.
    pte_t *pte = PGADDR(PDX(UVPT), pn / (PGSIZE >> 2), (pn % (PGSIZE >> 2))
<< 2);
    void *addr = (void *)(pn * PGSIZE);

    if (((*pte) & (PTE_W | PTE_COW)) && (~(*pte) & PTE_SHARE))
```

```
    {
        // writable or copy-on-write page
        int perm = (((*pte) & PTE_SYSCALL)  & (~PTE_W)) | PTE_COW;
        if ((r = sys_page_map(0, addr, envid, addr, perm)))
        {
            panic("duppage: %e\n",  r);
        }
        if ((r = sys_page_map(0, addr, 0, addr, perm)))
        {
            panic("duppage: %e\n", r);
        }
    }
    else
    {
        // read-only page or shared page
        int perm = (*pte) & PTE_SYSCALL;
        if ((r = sys_page_map(0, addr, envid, addr, perm)))
        {
            panic("duppage: %e\n", r);
        }
    }
    // panic("duppage not implemented");
    return 0;
}
```

In `lib/spawn.c`, we check every existent page in user space and copy those pages with `PTE_SHARE` set in `copy_shared_pages()`.

```
// Copy the mappings for shared pages into the child address space.
static int
copy_shared_pages(envid_t child)
{
    // LAB 5: Your code here.
    int r;
    for (int i = 0; i < PDX(UTOP); i++) if (uvpd[i] & PTE_P)
    {
        for (int j = 0; j < NPTENTRIES; j++)
        {
            void *addr = PGADDR(i, j, 0);
            pte_t pte = uvpt[PGNUM(addr)];
            if ((pte & PTE_P) && (pte & PTE_SHARE))
            {
                int perm = pte & PTE_SYSCALL;
                if ((r = sys_page_map(0, addr, child, addr, perm)) < 0)
                    return r;
            }
        }
    }
    return 0;
}
```

The keyboard interface

Exercise 9

Now to provide input function for user environments, we dispatch the hardware interrupt from keyboard and serial port to `kbd_intr()` and `serial_intr()` respectively, in which `cons_intr()` is called with corresponding function for requiring data from the hardware. In `cons_intr()`, input characters are stored in a ring buffer which is to be drained by the console file type defined in `lib/console.c`

```
    // 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;
    }
```

The Shell

Exercise 10

In order to support IO redirection, we need to make the file descriptor ID of the file provided to be 0 or 1. `init` has set up console as file descriptors 0 and 1 and this is inherited by the new environment. So after opening the file we must exploit the `dup(int oldfd, int newfd)` method to close the original descriptor 0 and set it to point to our provided file. Then we close the temporary file descriptor to prevent leak.

```
            // 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;
```

Reference

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Section 2.3