OS Assignment 4: Documentation

December 25, 2013

Group 12
Group members: Ma Xuan(1110580), Wang Shaoyao(1110539)
Wu Zhengwei(1110548), Yuan Zhaozheng(1110556), Zhang Chao(1110557)

Introduction

本次作业的目的是实现一个简单的基于磁盘的文件系统, 然后编写代码加载和运行一个存储在这个磁盘文件系统中的可执行代码文件。文件系统本身将按照微内核的模式进行设计,位于内核之外,在用户空间中。其他的进程通过使用IPC 请求来进行文件访问。类似于Unix 系统中的 exec 这样的功能,在微内核模式设计中将作为用户空间库的一部分被链接到每一个想要使用它的应用程序上,而不是将exec 功能在内核或是文件系统中实现。

Exercise 1

Description

修改内核中的进程初始化函数, 在文件 env.c 中的 env_alloc(), 使其给予进程1以 I/O 特权, 而不将该权限赋予任何一个其他进程。

阅读在目录fs下的代码树。文件 fs/ide.c实现了最基本的基于PIO的设备驱动程序。文件 fs/serv.c包含着用于文件系统服务的umain函数。

要注意的是,新的 .bochsrc 文件设置本次试验中的 Bochs 使用 kern/bochs.img 作为disk0 的镜像(类似于Windows 中的C 盘), 使用新的文件obj/fs/fs.img 作为disk1(好比是D 盘)。这本次试验中文件系统只应当访问disk1; disk0 只用来启动内核。如果你破坏了这两个镜像文件,你可以将各自还原到原始的状态:

$ rm obj/kern/bochs obj/fs/fs/img

$ gmake

Solution

根据材料 X86 的处理器使用EFLAGS寄存器中的IOPL位来决定在保护模式下是否允许产生诸如IN和OUT之类的特殊设备I/O指令。

IOPL介绍：IOPL （I/O Privilege Level）在EFLAGS 的13,12 位。用于指出当前任务的I/O特权级。取值0-3 只有当前任务的特权级（CPL）小于或等于IOPL时，才可以存取I/O地址空间。该域只有在CPL为0是由POPF和IRET指令修改。任务切换时改变IOPL的值。

在inc/mmu.h 中，我们找到关于eflags的宏定义:

Listing 1: inc/mmu.h

```
// Eflags register
#define FL_CF          0x00000001 // Carry Flag
#define FL_PF          0x00000004 // Parity Flag
#define FL_AF          0x00000010 // Auxiliary carry Flag
#define FL_ZF          0x00000040 // Zero Flag
#define FL_SF          0x00000080 // Sign Flag
#define FL_TF          0x00000100 // Trap Flag
#define FL_IF          0x00000200 // Interrupt Flag
#define FL_DF          0x00000400 // Direction Flag
#define FL_OF          0x00000800 // Overflow Flag
#define FL_IOPL_MASK 0x00003000 // I/O Privilege Level bitmask
```

```
#define FL_IOPL_0   0x00000000  // IOPL == 0
#define FL_IOPL_1   0x00001000  // IOPL == 1
#define FL_IOPL_2   0x00002000  // IOPL == 2
#define FL_IOPL_3   0x00003000  // IOPL == 3
#define FL_NT       0x00004000  // Nested Task
#define FL_RF       0x00010000  // Resume Flag
#define FL_VM       0x00020000  // Virtual 8086 mode
#define FL_AC       0x00040000  // Alignment Check
#define FL_VIF      0x00080000  // Virtual Interrupt Flag
#define FL_VIP      0x00100000  // Virtual Interrupt Pending
#define FL_ID       0x00200000  // ID flag
```

找到 FL_IOPL_MASK，其他进程默认为 FL_IOPL_0.

Listing 2: kern/env.c

```
int
env_alloc(struct Env **newenv_store, envid_t parent_id)
{
    ...
    // If this is the file server (e == &envs[1])
    // give it I/O privileges.
    // LAB 5: Your code here.
    if (e == &envs[1])
        e->env_tf.tf_eflags |= FL_IOPL_3;
    ...
}
```

Exercise 2

Description

使用或者不使用DMA 来实现中断驱动的IDE 磁盘访问。

请自行决定是否将设备驱动加到内核中(它保存在用户空间的文件系统中)或者放到它们自身特定的环境中。

Solution

以ide_read为例进行研究：

硬盘读写完成后会发出一个中断信号。操作系统接收到这个信号后,会陷入内核,然后进行异常分发。在硬盘中断的异常分发中将进行读硬盘的系统调用的进程的状态设为ENV_RUNNABLE, 然后再次进行进程轮转。这时需要进行硬盘读写的进程是可能被运行的。当它被运行时开始读取IDE缓冲区的内容.

运行Bochs可以检测出IDE的中断是IRQ14，即trapno==IRQ_OFFSET+14;而可编程中断控制器8259A只有0、1、2、4未被屏蔽, 因此需要去掉屏蔽14号中断。之后IDE产生的中断就可以被CPU收到了。

```
/* Initialize the 8259A interrupt controllers. */
void
pic_init(void)
{
    ...
    if (irq_mask_8259A != 0xFFFF)
        irq_mask_8259A &= ~(1 << 14);
        irq_setmask_8259A(irq_mask_8259A);
}
```

之后编写系统调用sys_read_block, 在进程需要进行读磁盘的操作时使用该系统调用, 然后在内核中完成IDE驱动的操作，并修改用户空间的ide_read函数如下, 即可实现中断驱动的IDE磁盘读取。

```
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;
}
```

Exercise 3

Description

实现文件fs/fs.c中的read_block和write_block函数。

函数 read_block 首先要检查指定的块是否在内存中, 如果不在, 使用 ide_read 函数读入这个块并分配内存页。要注意的一点是每一个块中可能包含着多个扇区, read_block 函数需要返回请求块的映射位置。

函数 write_block 假定指定的块已经在内存中了,只需要写回磁盘即可。我们将使用VM 硬件来判断在上次被读入内存或者被写回磁盘后, 这个块有没有被修改。这一步骤只需要检查vpt中的 PTE_D 位是否被置位(PTE_D位由处理器进行设置,参看Intel 386参考手册的第5 章节的5.2.4.3)。在写回磁盘之后, write_block 函数应当使用 sys_page_map 函数将 PTE_D 位清空。

Solution

需要用到的函数:

// 返回块号相对应的虚拟地址。
char* diskaddr(uint32_t blockno)
// 判断虚拟地址va是否被映射到内存。
bool va_is_mapped(void *va)
// 检查磁盘块是否映射，否则分配一个页面保存磁盘块数据
int map_block(uint32_t blockno)
// 读取磁盘块secno中nesecs个扇区数据到地址dst处
Int ide_read (uint32_t secno, void *dst, size_t nsecs)

Listing 3: fs/fs.c: read_block()

```
static int
read_block(uint32_t blockno, char **blk)
{
    int r;
    char *addr;

    if (super && blockno >= super->s_nblocks)
        panic("reading non-existent block %08x\n",
        blockno);

    if (bitmap && block_is_free(blockno))
        panic("reading free block %08x\n",
        blockno);

    // LAB 5: Your code here.
    if ((r = map_block(blockno)) < 0)
        return r;

    addr = diskaddr(blockno);
    if ((r = ide_read(blockno * BLKSECTS,
    addr, BLKSECTS)) < 0)
        return r;

    if (blk)
        *blk = addr;

    return 0;
}
```

先判断是否是脏页，如果是，就写入，并清除 PTE_D 位，如果不是直接返回。
这里 PTE_D 位的设置机制：当一个内存也被访问的时候，cpu就会对其进行标

记，把 PTE_D 位置1，但是cpu从来不会主动把PTE_D位置0，所以必须由程序设定。这里可以用 sys_page_map 来清除 PTE_D，PTE_D是脏读位。

用到的函数：
bool va_is_dirty(void *va) // 判断va的内容是否被修改了
sys_page_map() // 清除PTE_D

Listing 4: fs/fs.c: write_block()

```
void
write_block(uint32_t blockno)
{
    int r;
    char *addr;

    if (!block_is_mapped(blockno))
        panic("write unmapped block %08x", blockno);

    // Write the disk block and clear PTE_D.
    // LAB 5: Your code here.
    addr = diskaddr(blockno);

    if ((r = ide_write(blockno * BLKSECTS,
        addr, BLKSECTS)) < 0)
        panic("ide_write error: %e", r);

    if ((r = sys_page_map(0, addr, 0, addr, PTE_USER)) < 0)
        panic("sys_page_map error: %e", r);
}
```

Exercise 4

Description

实现函数read_bitmap。它首先检查所有文件系统中预留的块(block 0、block 1 和放置位图结构的块),并将它们设置为使用中。可使用提供的block_is_free函数。

Solution

用到的结构和函数：
// 系统中用一个全局变量指向映射到内存中的位示图块，方便查找
uint32_t *bitmap
// 检查磁盘块blockno的位图是否空闲
bool block_is_free(unit32_t blockno)

根据材料，每一个块用一个bit表示是否可用，即1表示可用，0表示可用。如果用一个块来存储这些信息，可以存 4096 * 8 = 32768 个。其中4096存在 BLKSIZE 中32768存在 BLKBITSIZE 中位示图在磁盘中的第二块磁盘块，从第二块开始检查，用block_is_free.

```
// Read and validate the file system bitmap.
void
read_bitmap(void)
{
    int r;
    uint32_t i;
    char *blk;

    for (i = 0; i * BLKBITSIZE < super->s_nblocks; i++) {
        if ((r = read_block(2+i, &blk)) < 0)
            panic("cannot read bitmap block %d: %e", i, r);
        if (i == 0)
            bitmap = (uint32_t*) blk;
        // Make sure all bitmap blocks are marked in-use
        assert(!block_is_free(2+i));
    }

    // Make sure the reserved and root blocks
    // are marked in-use.
    assert(!block_is_free(0));
    assert(!block_is_free(1));
    assert(bitmap);

    cprintf("read_bitmap is good\n");
}
```

代码说明：
在代码中加 0x7fff（32767）相当于 roudup 即向上舍入；循环体中加2是因为位示图在磁盘中的第二块磁盘块，从第二块开始检查代码。

Exercise 5

Description

按照 block_is_free 为例来实现 alloc_block_num, 它检查位图结构并寻找一个空闲的块, 设置此块为使用中并返回块号。当分配磁盘块时, 必须立刻使用write_block将对位图结构进行的修改存储到磁盘上,以保持文件系统的一致性。

Solution

```
int
alloc_block_num(void)
{
    // LAB 5: Your code here.
    if (super) {
        int i;

        for (i = 0; i < super->s_nblocks; i++) {
            if (block_is_free(i)) {
```

```
                bitmap[i/32] &= ~(1<<(i%32));
                // write back free block bitmap
                write_block(i/BLKBITSIZE+2);
                return i;
            }
        }
    }
    return -E_NO_DISK;
}
```

代码说明：
模32是因为在内存里存储时，32个bool值作为 unsigned int 保存; 参考 block_is_free.

Exercise 6

Description

完成文件fs/fs.c中的下列函数:
file_open, file_get_block, file_truncate_blocks, 和 file_flush。

除了上面实现的之外, 还有两类基本的文件操作没有完成: 命名、读取和写入。这是因为我们的文件系统进程不会直接为其他进程实现文件读写操作,而是使用内核提供的基于IPC 的页面共享措施来实现,将映射的页提交给其他进程,然后这些进程就可以直接读写这个页面了。

文件系统进程传递给其他进程的必定是在文件系统自己的缓冲内存中的内存块, 使用 file_get_block 获得。在后面我们将会实现用户级的read 和write 函数。

Solution

使用的函数：
static int wlak_path(const char *path, struct File **pdir, struct File**pf, char *lastelem)
// 从根目录开始查找，解析路径path，保存其路径指向的文件
int file_map_block(struct File *f, uint32_t filebno, uint32_t *diskbno, bool alloc)
// 将文件中第filebno磁盘块映射到文件系统中的diskbno块

实现函数 file_open；

利用 walk_path 打开路径path中的文件，获得其文件句柄。

```
// 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)
{
    int result = walk_path(path, 0, pf, 0);
```

```
    return result < 0 ? result : 0;
}
```

实现函数 file_get_block;

blk 指针指向文件F的第filebno个block，若不存在该block则创建。
调用 file_map_block() 函数，将block映射到内存，之后读取即可。
先用函数 file_map_block() 获得filebno的磁盘块数据diskbno;
再用函数 read_block() 获得的diskbno是blk指向其首地址。

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

    // Read in the block, leaving the pointer in *blk.
    // Hint: Use file_map_block and read_block.
    // LAB 5: Your code here.
    if ((r = file_map_block(f, filebno, &diskbno, 1)) < 0)
        return r;
    if ((r = read_block(diskbno, blk)) < 0)
        return r;
    return 0;
}
```

实现函数 file_truncate_blocks:

功能是减少文件大小。函数 file_truncate_blocks() 是将文件截为新的大小newsize，并释放掉文件中没有使用的磁盘块，包括间接磁盘块。
计算新的block数和旧的block数，然后调用file_clear_block() 删除。
还有间接块的问题，每个文件可以直接寻址10个block，之后用类似指针的办法间接寻址1024 个block，若新的block从10以上减少到10以下，则将指针删除，最后将文件的大小更新。

```
// Remove any blocks currently used by file 'f',
// but not necessary for a file of size 'newsize'.
static void
file_truncate_blocks(struct File *f, off_t newsize)
{
    int r;
    uint32_t bno, old_nblocks, new_nblocks;

    old_nblocks = (f->f_size + BLKSIZE - 1) / BLKSIZE;
    new_nblocks = (newsize + BLKSIZE - 1) / BLKSIZE;
    for (bno = new_nblocks; bno < old_nblocks; bno++)
        if ((r = file_clear_block(f, bno)) < 0)
            cprintf("warning: file_clear_block: %e", r);

    if (new_nblocks <= NDIRECT && f->f_indirect) {
        free_block(f->f_indirect);
        f->f_indirect = 0;
```

```
    }
}
```

实现函数 file_flush :

函数 file_flush() 将文件对象f中的内容刷新到磁盘上，只对文件中标记为dirty的数据操作。为了实现此函数，我们通过循环查看文件中所有的块（nblocks个块），利用三个系统提供的函数 file_map_block()、file_is_dirty() 和 write_block() 将有脏数据的文件写回磁盘。

```
void
file_flush(struct File *f)
{
    int i;
    uint32_t diskbno;

    for (i = 0; i < (f->f_size + BLKSIZE - 1) / BLKSIZE; i++) {
        if (file_map_block(f, i, &diskbno, 0) < 0)
            continue;
        if (block_is_dirty(diskbno))
            write_block(diskbno);
    }
}
```

Exercise 7

Description

服务器端位于文件服务器内部, 在 fs/serv.c 中实现。这些函数接受从客户端发来的IPC 请求, 解析参数, 之后使用 fs/fs.c 中的文件访问函数针对这些请求进行服务。我们已经为这个服务器端提供了一个框架,但是你需要填充它的部分内容。
参考 lib/fsipc.c 中的客户端以帮助你了解确切的服务器客户端协议。

Solution

inc/fs.h :
Fsreq_open :
客户进程向文件系统服务器发送打开文件请求，以req_omode模式打开文件。
Fsreq_map :
客户进程想文件系统服务器发送映射文件请求，大小为 req_offset。
Fsreq_set_size :
客户进程向文件系统服务器发送设置文件大小请求，大小设置为 req_size。
Fsreq_close :
客户进程向文件系统服务器发送关闭文件请求。
Fsreq_dirty :
客户进程向文件系统服务器发送数据修改请求，偏移量为req_offset。
Fsreq_remove :

客户进程向文件系统服务器发送删除文件请求，文件路径为req_path。

lib/fsipc.c :

fsipc_open :
向服务器发送打开文件请求，包括文件的路径path，打开方式omode，返回文件句柄fd。
fsipc_map :
向文件服务器发送块映射请求将文件fileid中要求的块映射到文件dstva地址处。
fsipc_set_size :
向文件服务器发送设置文件大小请求。
fsipc_close :
向文件服务器发送关闭文件请求。
fsipc_dirty :
向文件服务器发送文件中的块请求。
fsipc_remove :
向文件服务器发送删除文件请求。fsipc_sync :
向文件服务器发送打开文件同步请求，将修改的文件写到磁盘上。

serv.c :

openfile_lookup
根据fileid寻找openfile。
serve_open
打开一个文件，并填充OpenFile中的信息，然后发回去。
serve_set_size
设置文件的大小
serve_map
把服务器进程的一个block映射到用户进程。
serve_close
关闭一个文件。
serve_remove
删除一个文件。
serve_dirty
把一个页标记为脏页
serve_sync
把所有文件同步到磁盘。
serve
服务器进程主要函数，用来分配用户进程发来的请求。

在练习7 中，我们需要实现服务器端响应客户端IPC 请求的一系列函数。我们可以看出，fs/serv.c 中的函数都大同小异：参数1 为发出请求的函数ID，参数2 为发出的请求；服务器端的任务就是执行env 对应的请求，并将运行结果返回至env 中。这些函数的框架体系有个特点，每个函数最后两行，都为：
out:
ipc_send(envid,r,0,0);
之前的操作，如果有错误信息（r ¡ 0），则 goto out ，利用函数 ipc_send() ，返回错误代码r。

实现serve_map

使用函数 openfile_loopup() 打开对应文件，然后用 file_getblock() 得到需

要的block，之后根据 file open mode 设置权限为只读，或者可写。最后调用 ipc_send() 发送请求。

```
void
serve_map(envid_t envid, struct Fsreq_map *rq)
{
    int r;
    char *blk;
    struct OpenFile *o;
    int perm;

    if (debug)
        cprintf("serve_map %08x %08x %08x\n",
        envid, rq->req_fileid, rq->req_offset);

    // Map the requested block in the client's address space
    // by using ipc_send.
    // Map read-only unless the file's open
    // mode (o->o_mode) allows writes
    // (see the O_ flags in inc/lib.h).
    // LAB 5: Your code here.
    if ((r = openfile_lookup(envid, rq->req_fileid, &o)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }
    if ((r = file_get_block(o->o_file,
        rq->req_offset/BLKSIZE, &blk)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }

    perm = PTE_P | PTE_U | PTE_SHARE;
    if ((o->o_mode & O_WRONLY) ||
        (o->o_mode & O_RDWR))
        perm |= PTE_W;

    ipc_send(envid, 0, blk, perm);
}
```

实现 serve_close()

先用 openfile_loopup() 打开文件，然后调用 file_close()：

```
void
serve_close(envid_t envid, struct Fsreq_close *rq)
{
    struct OpenFile *o;
    int r;

    if (debug)
        cprintf("serve_close %08x %08x\n", envid,
                rq->req_fileid);

```

```
    if ((r = openfile_lookup(envid, rq->req_fileid, &o)) < 0)
        goto out;
    file_close(o->o_file);
    r = 0;

    out:
        ipc_send(envid, r, 0, 0);
}
```

实现serve_remove：

先将待删除文件的位置复制进临时变量path，因为字符串后可能没有结束标志符，之后调用 file_remove()‘函数执行删除。

```
void
serve_remove(envid_t envid, struct Fsreq_remove *rq)
{
    char path[MAXPATHLEN];
    int r;

    if (debug)
        cprintf("serve_remove %08x %s\n", envid, rq->req_path);

    // Copy in the path, making sure it's null-terminated
    memmove(path, rq->req_path, MAXPATHLEN);
    path[MAXPATHLEN-1] = 0;

    // Delete the specified file
    r = file_remove(path);
    ipc_send(envid, r, 0, 0);
}
```

实现serve_dirty()

用 openfile_lookup() 打开对应文件，然后用 file_dirty() 返回最终结果。

```
void
serve_dirty(envid_t envid, struct Fsreq_dirty *rq)
{
    struct OpenFile *o;
    int r;

    if (debug)
        cprintf("serve_dirty %08x %08x %08x\n", envid,
                rq->req_fileid, rq->req_offset);

    // Find the file and dirty the file at the requested
    // offset.
    // Send the return value back using ipc_send.
    // LAB 5: Your code here.
    if ((r = openfile_lookup(envid, rq->req_fileid, &o)) < 0)
        goto out;
```

```
    if ((r = file_dirty(o->o_file, rq->req_offset)) < 0)
        goto out;

20  out:
    ipc_send(envid, 0, 0, 0);
}
```

Exercise 8

Description

实现 fd_alloc 和 fd_lookup。函数fd_alloc找到一个未使用的文件描述符的编号,然后返回对应该描述符在描述符表中的指针。函数 fd_llokup 检查并确保给定的描述符是活动的, 并返回对应的描述符表项的指针。

Solution

这个练习需要我们完成lib/fd.c 中的fd_alloc() 和fd_lookup() 两个函数。这两个函数思路类似：先查找fdnum 对应的block ， 然后将结果返回至fd_store中。不过，我们需要先判断fdnum 是否合法，然后调用INDEX2FD 函数，得到FD 页的virtual address，并在内存中检查是否合法。

```
int
fd_alloc(struct Fd **fd_store)
{
        // LAB 5: Your code here.

        //panic("fd_alloc not implemented");
        uint32_t i;
        for (i = 0; i < MAXFD; i ++) {
                *fd_store = INDEX2FD(i);
                if (!(vpd[VPD(*fd_store)] & PTE_P) || !(vpt[VPN(*fd_store)] & PTE_P))
                        return 0;
        }
        *fd_store=0;
        return -E_MAX_OPEN;
}
```

通过检查页是否分配来确定fd有没有被分配。

```
int
fd_lookup(int fdnum, struct Fd **fd_store)
{
        // LAB 5: Your code here.

        //panic("fd_lhokup not implemented");
        if (fdnum < 0 || fdnum >= MAXFD)        return -E_INVAL;
        *fd_store=INDEX2FD(fdnum);
        if (!(vpd[VPD(*fd_store)] & PTE_P) || !(vpt[VPN(*fd_store)] & PTE_P)) {
                *fd_store=0;
        return -E_INVAL;
        }
        return 0;
}
```

根据fdnum查找一个fd。

Exercise 9

Description

实现open。它必须首先使用fd_alloc找到一个未使用的文件描述符,发送一个IPC请求到文件系统服务器来打开文件, 然后映射所有的文件页到对应的客户程序的预留内存区域。

确保当已经打开了最大个数的文件或者IPC请求失败时,你的程序可以正常处理。

Solution

open() 函数在lib/file.c 里，它实现打开一个文件或目录的功能。提示中说要用到函数fmap，这个函数将文件的大小从oldsize 扩充到newsize ， 并映射缺少的页 。

首先我们利用fd_alloc() 申请一个新的Fd 页，然后向**Server** 发出一个要求打开文件的信息，接下来利用函数fmap()，将新的页映射，并且返回页编号。

```
int
open(const char *path, int mode)
{

        // LAB 5: Your code here.
        //panic("open() unimplemented!");
        struct Fd *fd;
        int r=fd_alloc(&fd);
        if (r < 0) return r;

        r=fsipc_open(path,mode,fd);
        if (r < 0) return r;

        uint32_t newsize = fd->fd_file.file.f_size;
        if ((r =fmap(fd,0,newsize)) < 0)
                return r;

        return fd2num(fd);
}
```

这里要注意，open的时候会把文件的内容都map到fd对应的数据区里面，具体请看用户模块图，文件最大为4M。

Exercise 10

Description

实现close。它必须首先通知文件系统服务器它所修改的页面,然后再请求关闭文件。

当文件系统服务器被要求关闭文件时,它会将新的数据写入磁盘(思考为什么文件系统不能根据自己的页表中的PTE_D位来检查映射的文件页是否是被修改过?)。最后,函数close应当取消前面被打开文件的所有映射页的映射关系,以保证程序不会在文件被关闭之后访问这些页面。

Solution

有了open() 的实现，close() 也好实现了：利用函数funmap() 先卸载装载了的文件，然后调用fsipc_close()，向Server 发出消息，要求关闭文件。

```
static int
file_close(struct Fd *fd)
{
        // Unmap any data mapped for the file,
        // then tell the file server that we have closed the file
        // (to free up its resources).

        // LAB 5: Your code here.
        //panic("close() unimplemented!");
        uint32_t oldsize=fd->fd_file.file.f_size;
        int r=funmap(fd,oldsize,0,1);
        if (r < 0) return r;
        r=fsipc_close(fd->fd_file.id);
        if (r < 0) return r;
        return 0;
}
```

这里会把fdunmap 掉，跟serv.c是不一样的。然后fd对应数据的映射也会被unmap.

funmap会使相应文件的所有页变为脏的，并且unmap掉。然后fsipc_close会令服务器把所有脏页写回磁盘。

Exercise 11

Description

请添加文件服务器和客户端方面对大于 4MB 文件的支持。

Solution

阅读源代码可知一个文件有10个直接块号和1014个间接块号,间接块号存在于indirect块中。为了使其支持的块数超过4 M,我们使用双重间接块,即indirect间接块中存在的不是文件块的块号,而是二级间接块的块号（指针）。

修改file_block_walk函数如下:

```
int
file_block_walk ( struct File *f, uint32_t filebno ,
uint32_t ** ppdiskbno , bool alloc)
{
    int r;
```

```
    uint32_t *ptr;
    char *blk;

    if ( filebno < NDIRECT )
        ptr = &f-> f_direct [ filebno ];
    else
    {
      filebno =filebno - NDIRECT ;
      if ( filebno < NINDIRECT2 ) {
        if (f-> f_indirect == 0) {
            //1 st level indirect block not exists
            if ( alloc == 0)
                return -E_NOT_FOUND ;
            if ((r = alloc_block ()) < 0)
                return r;
            f-> f_indirect = r;
            // the 1st level indirect block
        }
        else
            alloc = 0;
        // we did not allocate a block
        if ((r = read_block (f->f_indirect , &blk)) < 0)
            return r;
        assert (blk != 0);
        if ( alloc )
            // must clear any block allocated
            memset (blk , 0, BLKSIZE );
        ptr = ( uint32_t *) blk + filebno /1024;
        // then we try to get 2nd level indirect block
        if (* ptr ==0)
        {
            if ((r = alloc_block ()) < 0)
                return r;
            *ptr=r;
            alloc =1;
        }
        else
            alloc =0;
        if ((r = read_block (*ptr , &blk)) < 0)
            return r;
        assert (blk !=0);
        if ( alloc )
            memset (blk ,0, BLKSIZE );
        ptr = ( uint32_t *) blk + filebno %1024;
      }

      else
        return -E_INVAL ;
    }
    * ppdiskbno = ptr;
    return 0;
}
```

Exercise 12

Description

函数spawn的框架在文件 lib/spawn.c 中。我们将把参数传递放到下一个作业中。完成这个函数,其执行流程是:

1. 创建一个新的进程;
2. 使用init_stack在USTACKTOP-BY2PG处分配堆栈;
3. 在ELF文件的指定地方载入程序的正文、数据和bss部分。不要忘了把这些程序中不是从可执行文件中读取的部分设置为0;
4. 使用新的 sys_set_trapframe 系统调用来设置子进程的寄存器;
5. 在可执行文件头部中指定的位置开始运行子进程;

当你测试代码的时候, 可以通过从 kern/init.c 运行 user/icode 程序来从文件系统中“spawn”。

Solution

Spawn这个函数的作用是把程序从磁盘读出，并放到JOS上运行。实现的功能类似Unix的fork()后马上接exec()，即：创建一个新的进程并从文件系统中载入程序镜像，然后在子进程中独立运行这镜像，完成子进程的创建后，父进程会独立运行。

分析spawn()函数。

```
int
spawn(const char *prog, const char **argv)
{
        unsigned char elf_buf[512];
        struct Trapframe child_tf;
        envid_t child;
        int r;
        int fdnum;
```

prog：需要执行的程序的路径。

argv：指向一个以NULL结尾的字符串组数组，是需要执行的程序的参数

elf_buf：用于保存读取到的程序镜像的ELF_header

child_tf：子进程开始运行时的运行时信息

child：子进程的id

fdnum：通过prog打开文件的file descriptor id

调用之前完成实现的open()函数打开程序，获得file descriptor id。如果产生错误，下同。

在注释的提示下，我们查看了kern/env.c中load_icode()的实现。参考它的实现完成了这里要求的读取ELF头信息，并通过magic_number检查它是否是ELF文件。结构elf中存储了prog的开始512字节的数据。

```
if ((r = read(fdnum, (void *)elf_buf, 512)) < 0) {
        close(fdnum);
        return r;
}
struct Elf *elf = (struct Elf *)elf_buf;
if (elf->e_magic != ELF_MAGIC)
        panic("bad elf format!");
```

使用sys_exofork()系统调用创建新进程并取得子进程的id

```
if ((child = sys_exofork()) < 0)
        return child;
```

用调用sys_exofork()时候创建的envs[ENVX(child)].env_tf来初始化child_tf。通过比较load_icode()，应该将子进程的eip设为elf-¿e_entry.

```
child_tf = envs[ENVX(child)].env_tf;
child_tf.tf_eip = elf->e_entry;
```

调用init_stack()初始化需要执行的程序（prog）的堆栈，简单的说，就是将它的参数压入它的地址空间的栈中。这个函数将在后面详细分析，值得一提的是，由于我们已经创建了子进程打开了文件，所以这里如果产生了错误，我们得先销毁子进程并关闭文件后再返回这个错误，因此这里跳到了这个函数结尾的标号error处完成。

```
if ((r = init_stack(child, argv, &child_tf.tf_esp)) < 0)
        goto error;
```

接下来这几段程序，要实现将程序的各段以合适的权限映射到虚拟内存首先爱你取得程序第一段和最后一个段地址。

```
struct Proghdr *ph = (struct Proghdr *)((uint8_t *)elf + elf->e_phoff),
                        *eph = ph + elf->e_phnum;
```

然后遍历各段，如果这一段的类型为ELF_PROG_LOAD则是我们需要处理的段。同时我们需要检查这段在内存中的大小是否不小于它的文件大小，ELF文件格式要求ph-¿p_filesz ¡= ph-¿p_memsz。

```
for ( ; ph < eph; ph++) {
        if (ph->p_type == ELF_PROG_LOAD) {
                if (ph->p_memsz < ph->p_filesz)
                        panic("spawn: file size too large!!");
```

我们需要从ph-¿p_offset开始将程序这段的数据一页一页的映射到ph-¿p_va这个虚拟地址，所以需要先将这两个地址对齐。

```
uint32_t va
        = ROUNDDOWN(ph->p_va, PGSIZE),
        start = ROUNDDOWN(ph->p_offset, PGSIZE);
```

此处我们得区分这一段是否可写。如果不可写，即'elf'的标志位ph-¿p_flags不包含ELF_PROG_FLAG_WRITE，则这一段仅仅包含程序的正文及只读数据，我们只用将这段只读权限一页一页地从磁盘读入并映射到子进程的地址空间。

```
if ((ph->p_flags & ELF_PROG_FLAG_WRITE) == 0) {
        uint32_t end = ROUNDUP(ph->p_offset + ph->p_filesz, PGSIZE);
        for ( ; start < end; start += PGSIZE, va += PGSIZE) {
                void *blk;
                if ((r = read_map(fdnum, start, &blk)) < 0)
                        goto error;
                if ((r = sys_page_map(0, blk, child, (void *)va, PTE_P
                                                        | PTE_U)) < 0)
                        goto error;
        }
}
```

但是，如果ph-¿p_flags包含ELF_PROG_FLAG_WRITE，这一段可写，情况就麻烦一些了。这里有个隐晦的地方就是我们需要用seek()函数将文件的位置指针修正到这一段的偏移位置ph-¿p_offset，没有调用seek()将会发生读取错误，子进程无法正常运行（因为实际上，上一段的ph-¿pfilesz到ph-¿p_memsz这一段是没有文件读取操作的，文件位置指针并不在这一段的首地址而在上一段的ph-¿p_offset+ph-¿p_fliesz处）。

```
else {
        if ((r = seek(fdnum, ph->p_offset)) < 0)
                goto error;
        uint32_t end = ROUNDUP(ph->p_offset + ph->p_memsz, PGSIZE);
```

遍序这一段的每一页，由于这些数据可写，所以我们需要在子进程的地址空间中为它们申请空间，所以这里的处理方式是在UTEMP处申请一个临时的页，如果当前页（start）地址在ph-¿p_offset到ph-¿p_offset+ph-¿p_filesz(注意，如果不是最后一页，则不能按照一页的数据读入，正确的大小为ph-¿p_offset+ph-¿p_filesz-start)，从磁盘将其读入UTEMP后映射到子进程的地址空间，否则，需要将UTEMP初始化为0后映射，在映射结束后，将UTEMP反映射（unmap）以释放空间。

加载的时候请注意：可写的段需要分配物理页，不可写的直接映射就可以了。具体代码如下：

```
for ( ; start < end; start += PGSIZE, va += PGSIZE) {
        if ((r = sys_page_alloc(0, UTEMP, PTE_P | PTE_U | PTE_W
                                                ) < 0))
                goto error;
        memset(UTEMP, 0, PGSIZE);
        if (start < ph->p_offset + ph->p_filesz) {
                size_t readsz = MIN(PGSIZE, ph->p_offset + ph->
                                        p_filesz - start);
                if ((r = read(fdnum, UTEMP, readsz)) < 0)
                        goto error;
        }
        if ((r = sys_page_map(0, UTEMP, child, (void *)va,
PTE_P | PTE_U | PTE_W)) < 0)
                goto error;
        if ((r = sys_page_unmap(0, UTEMP)) < 0)
                goto error;
}
}
        }
    }
```

至此，我们完成了程序镜像的载入和映射。下一步需要设置子进程运行时的信息，重点是设置eip和esp两个寄存器。最后还需要将子进程的运行状态设置为ENV_RUNNABLE等待调度。

```
        if ((r = sys_env_set_trapframe(child, &child_tf)) < 0)
                goto error;
        if ((r = sys_env_set_status(child, ENV_RUNNABLE)) < 0)
                goto error;
        close(fdnum);
        return child;
```

如果在打开文件、创建子进程之后发生错误。则需要关闭文件、销毁子进程后再返回错误。

```
error:
        sys_env_destroy(child);
        close(fdnum);
        return r;
```

那么，一个完整的swapn()函数如下：

```
int
spawn(const char *prog, const char **argv)
{
        unsigned char elf_buf[512];
        struct Trapframe child_tf;
        envid_t child;
        int r;
        int fdnum;

        if ((fdnum = open(prog, O_RDONLY)) < 0)
                return fdnum;
        if ((r = read(fdnum, (void *)elf_buf, 512)) < 0) {
                close(fdnum);
                return r;
        }
```

```
        struct Elf *elf = (struct Elf *)elf_buf;
        if (elf->e_magic != ELF_MAGIC)
                panic("bad elf format!");
        if ((child = sys_exofork()) < 0)
                return child;
        child_tf = envs[ENVX(child)].env_tf;
        child_tf.tf_eip = elf->e_entry;
        if ((r = init_stack(child, argv, &child_tf.tf_esp)) < 0)
                goto error;
        struct Proghdr *ph = (struct Proghdr *)((uint8_t *)elf + elf->e_phoff),
                                *eph = ph + elf->e_phnum;
```

```
        for ( ; ph < eph; ph++) {
                if (ph->p_type == ELF_PROG_LOAD) {
                        if (ph->p_memsz < ph->p_filesz)
                                panic("spawn: file size too large!!");
                        uint32_t va
                                = ROUNDDOWN(ph->p_va, PGSIZE),
                                start = ROUNDDOWN(ph->p_offset, PGSIZE);
                        if ((ph->p_flags & ELF_PROG_FLAG_WRITE) == 0) {
                                uint32_t end = ROUNDUP(ph->p_offset + ph->p_filesz, PGSIZE);
                                for ( ; start < end; start += PGSIZE, va += PGSIZE) {
                                        void *blk;
                                        if ((r = read_map(fdnum, start, &blk)) < 0)
                                                goto error;
                                        if ((r = sys_page_map(0, blk, child, (void *)va, PTE_P
                                                                        | PTE_U)) < 0)
                                                goto error;
                                }
                        }
```

```
                        else {
                                if ((r = seek(fdnum, ph->p_offset)) < 0)
                                        goto error;
                                uint32_t end = ROUNDUP(ph->p_offset + ph->p_memsz, PGSIZE);
                                for ( ; start < end; start += PGSIZE, va += PGSIZE) {
                                        if ((r = sys_page_alloc(0, UTEMP, PTE_P | PTE_U | PTE_W
                                                                        ) < 0))
                                                goto error;
                                        memset(UTEMP, 0, PGSIZE);
                                        if (start < ph->p_offset + ph->p_filesz) {
                                                size_t readsz = MIN(PGSIZE, ph->p_offset + ph->
                                                                        p_filesz - start);
                                                if ((r = read(fdnum, UTEMP, readsz)) < 0)
                                                        goto error;
                                        }
```

```
                                        if ((r = sys_page_map(0, UTEMP, child, (void *)va,
PTE_P | PTE_U | PTE_W)) < 0)
                                                goto error;
                                        if ((r = sys_page_unmap(0, UTEMP)) < 0)
                                                goto error;
                                }
                        }
                }
        }
        if ((r = sys_env_set_trapframe(child, &child_tf)) < 0)
                goto error;
        if ((r = sys_env_set_status(child, ENV_RUNNABLE)) < 0)
                goto error;
        close(fdnum);
        return child;

error:
        sys_env_destroy(child);
        close(fdnum);
        return r;
}
```

Exercise 13

Description

实现Unix 风格的exec.

Solution

参考《操作系统（第2版）》第13章UNIX：案例分析，得知内核通过以下步骤实现该系统调用：

(1) Exec系统调用要求以参数形式提供可执行文件名，并存储该参数以备将来使用。连同文件名一起，还要提供和存储其他参数，例如如果shell命令是"ls -l"，那么ls作为文件名，-l作为选项，一同存储起来以备将来使用。

(2) 现在，内核解析文件路径名，从而得到该文件的索引节点号。然后，访问和读取该索引节点。内核知道对任何shell命令而言，它要先在/bin目录中搜索。

(3) 内核确定用户类别(是所有者、组还是其他)。然后从索引节点得到对应该可执行文件用户类别的执行(X)权限。内核检查该进程是否有权执行该文件。如果不可以，内核提示错误消息并退出。

(4) 如果一切正常，它访问可执行文件的头部。

(5) 现在，内核要将期望使用的程序(例如本例中的ls)的可执行文件加载到子进程的区域中。但"ls"所需的不同区域的大小与子进程已经存在的区域不同，因为它们是从父进程中复制过来的。因此，内核释放所有与子进程相关的区域。这是准备将可执行镜像中的新程序加载到子进程的区域中。在为仅仅存储在内存中的该系统调用存储参数后释放空间。进行存储是为了避免"ls"的可执行代码覆盖它们而导致它们丢失。根据实现方式的不同，在适当的地方进行存储。例如，如果"ls"是命令，"-l"是它的参数，那么就将"-l"存储在内核区。/bin目录中"ls"实用程序的二进制代码就是内核要加载到子进程内存空间中的内容。

(6) 然后，内核查询可执行文件(例如ls)镜像的头部之后分配所需大小的新区域。此时，建立区域表和页面映射表之间的链接。

(7) 内核将这些区域和子进程关联起来，也就是创建区域表和P区表之间的链接。由图13-36可知，子进程的P区表已经由分支程序创建好了。

(8) 然后，内核将实际区域中的内容加载到分配的内存中。

(9) 内核使用可执行文件头部中的寄存器初始值创建保存寄存器上下文。

(10) 此时，子进程("ls"程序)已经运行。因此，内核根据子进程优先级，将其插到"准备就绪"进程列表的合适位置。最终，调度这个子进程。

(11) 在调度该子进程后，由前述(9)中介绍的保存寄存器上下文生成该进程的上下文。然后，PC、SP等就有了正确的值。

(12) 然后，内核跳转到PC指明的地址。这就是要执行的程序中第一个可执行指令的地址。现在开始执行"ls"这样的新程序。内核从步骤(5)中存储的预先确定的区域中得到参数，然后生成所需的输出。如果子进程在前台执行，父进程会一直等到子进程终止；否则它会继续执行。

(13) 子进程终止，进入僵尸状态，期望使用的程序已经完成。现在，内核向父进程发送信号，指明"子进程死亡"，这样现在就可以唤醒父进程了(参见图13-35)。

如果这个子进程打开新文件，那么这个子进程的UFDT、FT和IT结构就和父进程的不同。如果该子进程调用另一个子程序，就会重复执行/分支进程。这样就会创建不同深度层次的进程结构。

用户进程从地址空间中保留的部分读取所需要的字节,然后通过kern的帮助将页映射到合适的位置。可以保留部分的地址空间开始地址为0x40000000。其他部分参照spawn修改设置。

Exercise 14

Description

实现共享库的某一种加载机制, 并且把用户级的库代码移动到共享库内。

Solution

共享库的概念

共享库以.so结尾. (so == share object) 在程序链接的时候并不像静态库那样从库中拷贝使用的函数代码到生成的可执行文件中，而只是作些标记，然后在程序开始启动运行的时候，动态地加载所需库（模块）。所以，应用程序在运行的时候仍然需要共享库的支持。

共享库链接出来的可执行文件比静态库链接出来的要小得多，运行多个程序时占用内存空间比也比静态库方式链接少(因为内存中只有一份共享库代码的拷贝)，但是由于有一个动态加载的过程所以速度稍慢。

Exercise 15

Description

我们已经设置 spawn() 来调用函数 init_stack() 来设置子进程的堆栈。大部分的代码已经实现好了: 分配一个临时的页面并映射到父进程的固定地址(从TMPPAGE到TMPPAGETOP-1)。然后(在那个你需要添加代码的地方之后), 将这个页面映射到子进程的 USTACKTOP。你只需要将参数数组拷贝到这个页面。一定要修改那行设置 *init_esp 的代码以设置子进程的堆栈指针。子进程的初始堆栈指针应当指向它的argc参数, 上面的图表中已经指出位置了。

2.子进程部分: 检查在start标签下面的子进程的指令。此时, libmain() 和 umain() 都接受参数 (int argc,char * atgv[])。函数 libmain() 只是简单的将自己的参数传递给umain。你需要注意考虑当一个新进程是从内核被创建的情况,在那种情况下是没有参数被传递的。

注意: 实际上只有argc 和argv ptr 必须被设置在新进程的堆栈上。指针argv ptr 必须指向&argv[0], &argv[1].. &argv[n], 每一个都指向一个字符串。因而, argv [0]..argv[n]可以被放置在新进程地址空间的任何地方。

Solution

先看看这个函数。

```
static int
init_stack(envid_t child, const char **argv, uintptr_t *init_esp)
{
        size_t string_size;
        int argc, i, r;
        char *string_store;
        uintptr_t *argv_store;
```

Init_stack()函数的三个参数及开头声明的变量的意义如下：

child：子进程id
argv：参数表
Init_esp：需要得到的初始化后的esp
String_size：参数表的长度
argc：参数个数
String_store：用于将参数表串接成一个字符串存储
Argv_store：用于在栈中存储参数指针

首先获取参数表的总长度，由于字符串以'0'结尾，所以每个字符串的长度都会额外多1。

```
        string_size = 0;
        for (argc = 0; argv[argc] != 0; argc++)
                string_size += strlen(argv[argc]) + 1;
```

决定strong_store和argv_store的位置，具体关系见上图。这两部份先从UTEMP分配空间，等最后再映射到子进程的地址空间，然后父进程这里需要反映射（unmap）。

```
        // Determine where to place the strings and the argv array.
        // Set up pointers into the temporary page 'UTEMP'; we'll map a page
        // there later, then remap that page into the child environment
        // at (USTACKTOP - PGSIZE).
        // strings is the topmost thing on the stack.
        string_store = (char*) UTEMP + PGSIZE - string_size;
        // argv is below that.  There's one argument pointer per argument, plus
        // a null pointer.
        argv_store = (uintptr_t*) (ROUNDDOWN(string_store, 4) - 4 * (argc + 1));
```

栈的最后两个位置需要存储argc和argv所以需要检查栈空间是否足够。如果足够，则为这个栈申请一页空间。

```
        // Make sure that argv, strings, and the 2 words that hold 'argc'
        // and 'argv' themselves will all fit in a single stack page.
        if ((void*) (argv_store - 2) < (void*) UTEMP)
                return -E_NO_MEM;

        // Allocate the single stack page at UTEMP.
        if ((r = sys_page_alloc(0, (void*) UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
                return r;
```

接下来我们需要将传入并存在了argv字符串数组中的参数转存到string_store字符串，由于字符串最后额外有有一位'0'，所以string_store的增量为strlen(argv[i]+1)。并使用UTEMP2USTACK()，使argv_store[i]指向正确的地址。

```
        for (i = 0; i < argc; i++) {
                size_t len = strlen(argv[i]);
                memmove(string_store, argv[i], len);
                argv_store[i] = UTEMP2USTACK(string_store);
                string_store += len + 1;
        }
```

在参数表的最后补充一个'0'表示字符串的结束。最后设置argv和argc的位置，至此，子进程的栈空间设置完毕。可以将init_esp赋值为UTEMP2USTACK(argv_store-2)。

```
        argv_store[argc] = 0;

        *(argv_store - 1) = UTEMP2USTACK(argv_store);
        *(argv_store - 2) = argc;
        *init_esp = UTEMP2USTACK(argv_store - 2);
```

最后便是要将这个栈映射到子进程的地址空间，并且反映父进程的这一页。

```
        // After completing the stack, map it into the child's address space
        // and unmap it from ours!
        if ((r = sys_page_map(0, UTEMP, child, (void*) (USTACKTOP - PGSIZE), PTE_P | PTE_U |
PTE_W)) < 0)
                goto error;
        if ((r = sys_page_unmap(0, UTEMP)) < 0)
                goto error;

        return 0;

error:
        sys_page_unmap(0, UTEMP);
        return r;
}
```

给出完整的init_stack()函数如下：

```
static int
init_stack(envid_t child, const char **argv, uintptr_t *init_esp)
{
        size_t string_size;
        int argc, i, r;
        char *string_store;
        uintptr_t *argv_store;

        // Count the number of arguments (argc)
        // and the total amount of space needed for strings (string_size).
        string_size = 0;
        for (argc = 0; argv[argc] != 0; argc++)
                string_size += strlen(argv[argc]) + 1;


        string_store = (char*) UTEMP + PGSIZE - string_size;
        // argv is below that.  There's one argument pointer per argument, plus
        // a null pointer.
        argv_store = (uintptr_t*) (ROUNDDOWN(string_store, 4) - 4 * (argc + 1));
```

```
        // Make sure that argv, strings, and the 2 words that hold 'argc'
        // and 'argv' themselves will all fit in a single stack page.
        if ((void*) (argv_store - 2) < (void*) UTEMP)
                return -E_NO_MEM;

        // Allocate the single stack page at UTEMP.
        if ((r = sys_page_alloc(0, (void*) UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
                return r;

        for (i = 0; i < argc; i++) {
                size_t len = strlen(argv[i]);
                memmove(string_store, argv[i], len);
                argv_store[i] = UTEMP2USTACK(string_store);
                string_store += len + 1;
        }
```

```
        argv_store[argc] = 0;

        *(argv_store - 1) = UTEMP2USTACK(argv_store);
        *(argv_store - 2) = argc;
        *init_esp = UTEMP2USTACK(argv_store - 2);

        // After completing the stack, map it into the child's address space
        // and unmap it from ours!
        if ((r = sys_page_map(0, UTEMP, child, (void*) (USTACKTOP - PGSIZE), PTE_P | PTE_U |
PTE_W)) < 0)
                goto error;
        if ((r = sys_page_unmap(0, UTEMP)) < 0)
                goto error;

        return 0;

error:
        sys_page_unmap(0, UTEMP);
        return r;
}
```

随着最后一个练习的完成，整个实验到此结束。运行make grade截图：

```
fs i/o [testfsipc]: OK (4.7s)
read_block [testfsipc]: OK (4.7s)
write_block [testfsipc]: OK (4.7s)
read_bitmap [testfsipc]: OK (4.7s)
alloc_block [testfsipc]: OK (4.7s)
file_open [testfsipc]: OK (4.7s)
file_get_block [testfsipc]: OK (4.7s)
file_truncate [testfsipc]: OK (4.7s)
file_flush [testfsipc]: OK (4.7s)
file_rewrite [testfsipc]: OK (4.7s)
serv_* [testfsipc]: OK (4.7s)
PART A SCORE: 55/55
motd display [writemotd]: OK (3.6s)
motd change [writemotd]: OK (3.6s)
spawn via icode [icode]: OK (3.6s)
PART B SCORE: 45/45
shiki@shiki-desktop:~/桌面/lab5$
```

Bonus Exercise 1

Description

请在作业3(抢占式调度)工程基础上完成以下作业:

Job1.
扩展内核代码, 使得页错误和各种在用户模式下产生的处理器异常都可以重定向到用户模式的异常处理函数中; 然后写一个用户模式的测试程序来测试这些异常的处理(如:除0错误)。

Job2.
实现一个称为sfork()的共享内存fork(), 它的功能是使得父进程和儿子进程都共享通用内存空间(栈空间除外,因为它们是使用copy-on-write 方式来处理)。修改user/forktree.c 源码,用sfork()来替代fork().

要是你已经完成了以前留过的IPC 作业,那么你现在可以用sfork()来运行user/pingpongs. 当然,你需要采用新的方式提供全局env 指针的功能。

Solution

第一步共享文件：具体点描述就是父进程开了一个文件，然后fork一个子进程，然后子进程改文件的东西，父进程读文件的时候，文件的内容跟子进程改变之后的是一样的。回顾lab4中的copy on write机制：当父进程或者子进程要写它们共有的东西的时候，系统会新开一页，然后把原来那页的内容复制过去，这样做能使父子进程完全相互独立，就是说子进程开始之后，父进程不能影响子进程（如果子进程不愿意的话）。

现在我们要做的事情就是如果父进程有打开文件的话，父子进程共享这个文件，无论父进程还是子进程修改这个文件，父子进程读这个文件的时候都会是改变之后的文件（仅限于文件所在内存页是这样，其它还是相互独立的）。说白了就是通过文件的共享特性实现进程之间的通信。如果是PTE_SHARE类型的话，映射到共同的物理页。把父进程打开的文件映射到子进程里面。这里主要作用是用来作为输入和输出文件的。系统调用并非直接和程序员或系统管理员打交道，它仅仅是一个通过软中断机制（我们后面讲述）向内核提交请求，获取内核服务的接口。而在实际使用中程序员调用的多是用户编程接口——API，而管理员使用的则多是系统命令。

用户编程接口其实是一个函数定义，说明了如何获得一个给定的服务，比如read( )、malloc( )、free( )、abs( )等。它有可能和系统调用形式上一致，比如read()接口就和read系统调用对应，但这种对应并非一一对应，往往会出现几种不同的API内部用到同一个系统调用，比如malloc( )、free( )内部利用brk( )系统调用来扩大或缩小进程的堆；或一个API利用了好几个系统调用组合完成服务。更有些API甚至不需要任何系统调用——因为它并不是必需要使用内核服务，如计算整数绝对值的abs（）接口。

另外要补充的是Linux的用户编程接口遵循了在Unix世界中最流行的应用编程界面标准——POSIX标准，这套标准定义了一系列API。在Linux中（Unix也如此），这些API主要是通过C库（libc）实现的，它除了定义的一

些标准的C函数外，一个很重要的任务就是提供了一套封装例程（wrapper routine）将系统调用在用户空间包装后供用户编程使用。

下一个需要解释一下的问题是内核函数和系统调用的关系。大家不要把内核函数想像的过于复杂，其实它们和普通函数很像，只不过在内核实现，因此要满足一些内核编程的要求。系统调用是一层用户进入内核的接口，它本身并非内核函数，进入内核后，不同的系统调用会找到对应到各自的内核函数——换个专业说法就叫：系统调用服务例程。实际上针对请求提供服务的是内核函数而非调用接口。比如系统调用getpid实际上就是调用内核函数sys_getpid。

Linux系统中存在许多内核函数，有些是内核文件中自己使用的，有些则是可以export出来供内核其他部分共同使用的，具体情况自己决定。

具体实现代码如下：

```
/*******************************sduppage*****************************/
static int
sduppage(envid_t envid, unsigned pn, int need_cow)
{
    int r;
    void * addr = (void *) ((uint32_t) pn * PGSIZE);
    pte_t pte = vpt[VPN(addr)];
    if (need_cow || (pte & PTE_COW) > 0) {
        if ((r = sys_page_map (0, addr, envid, addr, PTE_U|PTE_P|PTE_COW)) < 0)
            panic ("duppage: page re-mapping failed at 1 : %e", r);
        if ((r = sys_page_map (0, addr, 0, addr, PTE_U|PTE_P|PTE_COW)) < 0)
            panic ("duppage: page re-mapping failed at 2 : %e", r);
    } else
    if ((pte & PTE_W) > 0) {
        if ((r = sys_page_map (0,  addr, envid, addr, PTE_U|PTE_W|PTE_P)) < 0)
            panic ("duppage: page re-mapping failed at 3 : %e", r);
    } else {
        if ((r = sys_page_map (0,  addr, envid, addr, PTE_U|PTE_P)) < 0)
            panic ("duppage: page re-mapping failed at 4 : %e", r);
    }
    return 0;
}
```

```
/*****************************************************************************/
int
sfork(void)
{
	set_pgfault_handler (pgfault);
	envid_t envid;
	uint32_t addr;
	int r;
	envid = sys_exofork();
	if (envid < 0)
	    panic("sys_exofork: %e", envid);
	// We're the child
	if (envid == 0) {
	    env = &envs[ENVX(sys_getenvid())];
	    return 0;
	}
	// We're the parent.
	int stackarea = 1;
	for (addr = USTACKTOP - PGSIZE; addr >= UTEXT; addr -= PGSIZE) {
		if ((vpd[VPD(addr)] & PTE_P) > 0 && (vpt[VPN(addr)] & PTE_P) > 0 && (vpt[VPN(addr)] & PTE_U) > 0)
			sduppage (envid, VPN(addr), stackarea);
		else
			stackarea = 0;
	}
	if ((r = sys_page_alloc (envid, (void *)(UXSTACKTOP - PGSIZE), PTE_U|PTE_W|PTE_P)) < 0)
		panic("fork: page allocation failed : %e", r);
	extern void _pgfault_upcall (void);
	sys_env_set_pgfault_upcall (envid, _pgfault_upcall);
	// Start the child environment running
	if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
	    panic("fork: set child env status failed : %e", r);
	return envid;
```

Bonus Exercise 2

请在作业2(内管管理)工程基础上完成以下作业:

Job3.

写一篇概述性文档用来描述内核如何实现在用户环境下无限制地使用所有4GB 虚拟的和线性的空间。

这种实现技术有时也被称为“follow the bouncing kernel ”。在你的文档中,需要清楚的说明处理器在内核和用户模式之间切换时到底发生了哪些变化,以及内核是如何完成这些切换的; 同样,也需要描述内核在这种机制下是如何访问物理内存和IO 设备的, 以及内核在系统调用情况下是如何访问用户环境的虚拟地址空间的,等等。最后,从灵活性、性能、内核复杂度及你所关心的各种指标方面分析一下这种方案的优缺点。

Acknowledgement

袁兆争(1110556), 张超(1110557) 完成了前7题的代码及文档;

马璇(1110580)，王少尧(1110539)，武政伟(1110548）完成了后8题及附加题代码及文档。