Lab5

Unix 的文件系统：

1.使用 inode 来管理所有的文件(目录也属于文件)，每个文件都有一个 inode。该文件的属性信息存储在 inode 的元数据中。

2.元数据是一种描述数据的数据。

3.多个文件名对应同一个 inode 称为硬连接。

Jos 文件系统：

jos 文件除去 inode 结点，而是将目录或者文件的信息存储在目录项中。（该信息表现为 inc/fs.h 中的 File 结构）

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

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

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

可以看得出来，该结构包含文件的名字(最大 128 个字节)。

我们需要注意的是 f_direct[NDIRECT]和 f_indirect 变量。这两个变量表示文件数据的具体位置。其中 f_direct[NDIRECT]为文件的直接块(一共 10 块，40K)，即如果文件小于等于 40K 则直接将文件数据块的位置存储于 f_direct[NDIRECT]中，如果要是大于 40K，则超出的部分存储在 f_indirect 指向的页面上，需要间接寻址找到该文件的具体数据块。

作业 1.

该问题要我们在 init.c 中使用 ENV_CREATE 创建文件系统进程。然后在 env.c 中赋予文件系统进程 I/Oprivilege 的权限，使其可以读取 I/O 映射空间。

代码：

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

```
// LAB 5: Your code here.
if (type == ENV_TYPE_FS) {
    e->env_tf.tf_eflags |= FL_IOPL_MASK; //I/O Privilege Level
}
```

其中权限为 FL_IOPL_MASK 为 inc/mmu.h 中的宏定义。

*现在我们分析 jos 是如何操作磁盘的。根据 jos 的介绍我们知道 jos 是基于 PIO 来实现对 I/O 设备的访问。

PIO(programmed I/O):是指运行在 CPU 的程序可以通过读写内存的指令来读取 I/O 设备的地址空间。而 PIO 的这种方式由要求 I/O 设备的地址空间是通过 MMIO 映射的，即内存和 I/O 在同一个地址空间中。JOS 就是这种映射方式，正如在前面实验看到的 IOPYMEM 与内存在同一地址空间。

作业 2.
fs/bc.c 的作用：
由于 JOS 最大支持 3G 的磁盘大小，虽然文件系统拥有独立的 4G 虚拟内存，而 JOS 并没有直接把所有的磁盘都一次性读入内存中，而是实现 bc.c 来实现磁盘的块缓存机制。具体来说就是当进程需要读取或者写磁盘文件时，首先在虚拟内存中检查对应的块是否被读入内存(block0 被映射到 0x10000000...),可以通过 vpt 来检查块是否被映射。如果对应的块没有被映射，则发出 bc.c 的缺页中断，将对应的块读入内存。
首先完成 bc_pgfault()函数：

```
void *addr = (void *) utf->utf_fault_va;
uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
int r;
    envid_t envid;
    void *blkaddr;
    envid = thisenv->env_id;
    blkaddr = ROUNDDOWN(addr, PGSIZE);

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

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

// Allocate a page in the disk map region, read the contents
// of the block from the disk into that page, and mark the
// page not-dirty (since reading the data from disk will mark
// the page dirty).
//
// LAB 5: Your code here
if (sys_page_alloc(envid, blkaddr, PTE_SYSCALL) < 0) {
    panic("bg_pgfault: can't allocate new page for disk block\n");
}

if (ide_read(blockno*BLKSECTS, blkaddr, BLKSECTS) < 0) {
    panic("bg_pgfault: failed to read disk block\n");
}

// 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);
```

分析：
1.由于磁盘空间被映射到了虚拟地址的 DISKMAP(0x10000000)到 DISKMAP+DISKSIZE 上，所以首先我们需要检查要读取的地址是否为磁盘地址，磁盘块号是否大于总的磁盘块数目。
其中 nblocks 和 super，bitmap 是在 fsformat.c 中定义和初始化的
2.接着对磁盘块分配一个 4K 的内存空间，使用 ide_read 读取磁盘文件到指定虚拟地址上。

这里值得注意的是 ide_read 的操作单位是扇区 512K，所以我们给的参数必须是指定扇区的起始地址和需要读取的扇区数目。
另外需要深刻理解 bitmap 的工作机制：(fs/fsformat.c)

```
nbitblocks = (nblocks + BLKBITSIZE - 1) / BLKBITSIZE;
bitmap = alloc(nbitblocks * BLKSIZE);
memset(bitmap, 0xFF, nbitblocks * BLKSIZE);
```

这是一个比较神奇的代码。nbitblocks 记录块的数目除以 32K（最小为 1），也就是说我们至少分配一个页来记录块的映射情况(该页中的每个 bit 记录一个 block 的映射情况)。如果块的数目大于 32K，则分配若干页。
如此以来 bitmap 维护一个 block 是否被映射的状态表。

接着完成 flush_block 函数，该函数完成向磁盘写文件块的操作。创建文件时，file_flush 会调用该函数把文件的 metaData 写进磁盘。

```
void
flush_block(void *addr)
{
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
    void *blkaddr;
    blkaddr = ROUNDDOWN(addr, PGSIZE);
    envid_t envid;
    if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
        panic("flush_block of bad va %08x", addr);

    // LAB 5: Your code here.
    envid = thisenv->env_id;
    if (!va_is_mapped(addr) || !va_is_dirty(addr)) {
        return;
    }

    if (ide_write(blockno*BLKSECTS, blkaddr, BLKSECTS) < 0)
       panic("flush_block: failed to write disk block");

    if (sys_page_map(envid, blkaddr, envid, blkaddr, PTE_SYSCALL) < 0)
       panic("flush_block: failed to mark disk page as non dirty\n");
}
```

分析：
1.检查地址是否越界
2.如果地址对应的磁盘块没有被映射到文件进程的虚拟空间，则意味着向一个不存在的块写，直接返回。
3.如果地址没有被修改过则不需要写回。
4.然后调用 ide_write 函数，其参数和 ide_read 一致
5.如果确实执行了写回操作，则此时我们需要将对应内存的标志位的 PTE_D 置为 0,该操作通过 sys_page_map 来完成。

作业 3.
要求根据 bitmap 返回一个空闲的 block。
其作用是在将磁盘写入内存时，我们需要首先在内存中分配一个块来存放磁盘数据(OS 以块进行读写)。
分析：
1.根据 bitmap 的每一个 bit 来检测该 bit 对应的 block 是否为空闲。bitmap 在 fsformat.c 初始化时空闲页被置为 1.
2.由于 bitmap 是指向 int 的指针，所以 bitmap 数组的每一项对应于 32 个 block。此时我们需要使用移位运算来检测每一位的状态。
3.最后根据要求，将改变的 bitmap 块写回磁盘。然而我们需要知道 bitmap 到底在哪一个 block 上，且 bitmap 占用了几个磁盘块。
在此之前我们需要了解 JOS 是如何模拟磁盘的。作业中提到如果想要将 jos 系统的 disk1 和 disk0 还原到初始状态，只需要将 obj/kern/kernl.img 和 obj/fs/fs.img 删除，然后 make 即可。

我们知道 make 指令是 unix/linux 的编译文件指令，make 根据 lab5 根目录下的 GNUmakefile 文件定义的规则来编译文件，我们也可以将编译规则分散到每个子文件下的 Makefrag 文件中，最后 GNUmakefile 文件中按顺序包含这些文件即可。
现在打开 fs 目录下的 Makefrag 文件

```
# How to build the file system image
$(OBJDIR)/fs/fsformat: fs/fsformat.c
	@echo + mk $(OBJDIR)/fs/fsformat
	$(V)mkdir -p $(@D)
	$(V)$(NCC) $(NATIVE_CFLAGS) -o $(OBJDIR)/fs/fsformat fs/fsformat.c

$(OBJDIR)/fs/clean-fs.img: $(OBJDIR)/fs/fsformat $(FSIMGFILES)
	@echo + mk $(OBJDIR)/fs/clean-fs.img
	$(V)mkdir -p $(@D)
	$(V)$(OBJDIR)/fs/fsformat $(OBJDIR)/fs/clean-fs.img 1024 $(FSIMGFILES)

$(OBJDIR)/fs/fs.img: $(OBJDIR)/fs/clean-fs.img
	@echo + cp $(OBJDIR)/fs/clean-fs.img $@
	$(V)cp $(OBJDIR)/fs/clean-fs.img $@

all: $(OBJDIR)/fs/fs.img
```

根据注释，这一段为建立文件系统镜像文件。大致为流程为：
首先将 fs/format 编译为 obj/fs/fsformat 可执行文件,然后 fsformat 可执行文件生成 fs.img。我们只需要知道文件系统的磁盘镜像是由 fs/fsformat.c 生成的即可。这样我们便可以在 fs/fsformat 中查看 fsformat 是如何初始化 super 和 bitmap 的。
这里就不截图了：

- 首先在 opendisk()函数中打开磁盘镜像，使用 alloc 分配一个 block。然而这个块却没有使用。
- 接着再使用 alloc 为 super 分配一个超级块，所以 super 块并不是在磁盘的第 0 块。
- 接着初始化 super 块。
- 使用 alloc 为 bitmap 分配块（根据 nblocks 来确定分配的块数）。并将 bitmap 全部置为 1

代码如下：

```
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.
	uint32_t i, j;

	for (i = 0; i < super->s_nblocks/32; i++) {
	for (j = 0; j < 32; j++) {
		uint32_t match = (1 << j);
		if (bitmap[i] & match) {
		bitmap[i] &= ~match;
		flush_block(diskaddr((i * 32 | j)/BLKBITSIZE + 2));
		return (i * 32) | j;
		}
	}
	}

	return -E_NO_DISK;
}
```

作业 4.
完成 fs/fs.c 的 file_block_walk 和 file_get_block
（1）file_block_walk 原型：

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

1.该函数根据 f 和 fileno，查找文件 f 中第 fileno 块的入口地址。
即如果 fileno<10,则返回 f_direct[filebno]
如果 fileno>=10,则在间接目录项（f_indirect）中查找。
2.该函数还有一个 bool alloc 的参数：
该参数类似 kern/pmap.c 中 pgdir_walk 函数的 create 参数。即如果 fileno 大于等于 10，而间接目录项 f_indirect 为空，则使用 alloc_block 分配一个块作为间接目录表，使用 memeset 清空。类似 pgdir_walk 使用 page_alloc 分配一个页作为页表。

```
// LAB 5: Your code here.
uint32_t blockno;

if (filebno < NDIRECT) {
    *ppdiskbno = &f->f_direct[filebno];
} else if (filebno < (NINDIRECT + NDIRECT)) {
    if (!f->f_indirect) { // no indirect block set
        if (!alloc)
            return -E_NOT_FOUND;

        blockno = alloc_block();

        if (blockno < 0)
            return -E_NO_DISK;

        f->f_indirect = blockno;
        memset(diskaddr(blockno), 0, BLKSIZE); // clear block
    }

    *ppdiskbno = &((uintptr_t *) diskaddr(f->f_indirect))[filebno - NDIRECT];

} else
    return -E_INVAL;

return 0;
```

（2）file_get_block
该函数在 file_block_walk 的基础上进一步找到块的实际地址，如果该文件块对应的入口地址项为空，即标识该文件块此时没有对应的 block，使用 alloc_block 分配一个 block。
分析：
1.首先通过 file_get_block 获得块的入口地址
2.入口地址项的内容就是具体块的地址
3.如果发现该入口地址项为空，即该文件的第 fileno 块没有对应的具体块，则使用 alloc_block 分配
代码：

```
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
    // LAB 5: Your code here.
    uint32_t *blockno;
    uint32_t err;

    err = file_block_walk(f, filebno, &blockno, 1);
    if (err < 0)
        return err;

    if (!*blockno) {
        uint32_t block_no = alloc_block();
        if (block_no < 0) {
            return -E_NO_DISK;
        }
        *blockno = block_no;
    }

    *blk = (char *) diskaddr(*blockno);

    return 0;
}
```

C/S 模式的文件系统访问
该部分的主要功能是添加文件系统的接口供其他普通进程使用，以便其他进程可以操作磁盘文件。JOS 采用远过程调用（RPC）来实现该功能，通过 IPC 传递参数和共享页。

下面我们来看一下一个普通进程操作文件的流程（以读文件为例）：

- 根据 JOS 的介绍，可以知道普通进程通过调用 lib/fd.c 的 read()函数，该函数根据 fdnum 使用 fd_lookup()查找 fd(文件描述符)，然后根据 fd 通过 dev_dev_id()查找该文件对应的设备文件（JOS 跟 Unix 类似，将 IO 抽象为文件）。在 inc/fd.h 中可以看到设备文件的结构：

```
// 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);
};
```

  可以看到设备文件结构定义了各种文件操作，这些操作都在 lib/file.c 中实现。在 lib/fd.c 的 read()函数最后调用 dev_read()函数。
- 在 lib/file.c 中，dev_read 被重命名为 devfile_read。该函数此时我们还没有实现。根据 JOS 提示，在 devfile_read()中我们需要调用 fsipc 来启动 IPC
- 在 lib/file.c 的 fsipc()中，调用 ipc_send()函数向 fs/serv.c 的 serve()函数发送文件操作请求,然后进入 ipc_recv()等待文件系统的回应
- 此时便来到了 fs/serv.c 的 serve()函数。在 serve()函数根据请求类型调用不同的处理函数

下面我们来仔细查看 fs/serv.c 文件：
fs/serv.c 文件在 umain()函数中通过一系列的初始化进入 serve()函数：
该函数部分代码：

```
req = ipc_recv((int32_t *) &whom, fsreq, &perm);
```

```
if (req == FSREQ_OPEN) {
    r = serve_open(whom, (struct Fsreq_open*)fsreq, &pg, &perm);
} else if (req < NHANDLERS && handlers[req]) {
    r = handlers[req](whom, fsreq);
} else {
    cprintf("Invalid request code %d from %08x\n", whom, req);
    r = -E_INVAL;
}
ipc_send(whom, r, pg, perm);
sys_page_unmap(0, fsreq);
```

分析：
1.serve()函数无限循环等待普通进程的文件操作请求。
其中 req 是客户端进程 ipc_send 发送的 type，（可以在 lib/file.c 的 fsipc()中看到）
2.然后根据 req 的类型调用不同的处理函数，serve()中的 handlers 是通过 typedef 重定义的函数名。分别对应不同的处理函数。
3.最后通过 ipc_send 唤醒客户端进程。

作业 5.
首先完成 serve_read():
该函数完成读文件操作，但是我们如何知道我们读哪一个文件和读多少字节？
在上述的客户端进程通过文件系统进程操作文件流程中我们忽略了一个细节。JOS 的文档中介绍，在客户端进程有一个 fsipcbuf 共享页，并且在调用 ipc_send 之前在共享页中填写相关的操作细节，接着通过 ipc_send 将该页(fsipcbuf)共享给文件系统进程，而且被映射到文件系统进程的 fsreq 共享页。

在文件系统进程中，该共享页通过联合体 Fsipc 来管理.该联合体在 Inc/fs.h 中定义。
在该结构体定义的最后一行，char _pad[PGSIZE]用来占位，保证 Fsipc 至少为一个页。注意到这一点，我们就可以理解为什么在 serve_read()的开始为什么可以使用联合体中的多个变量。

根据上面的分析，我们开始具体实现 serve_read()的代码：
1.首先根据 Fsipc 联合体中的 read 变量的 req_fileid，通过 openfile_lookup()函数查找打开的文件，如果该文件还没有打开则直接报错。因为必须先通过 FSREQ_OPEN 打开文件在读取。
该函数将 Openfile 结构赋值到引用参数 o 中。
下面来看一下 Openfile 的结构：

```
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
};
```

其中的 File 结构定义了文件的元数据。
其中的 Fd 结构是在 inc/fd.h 中定义。该结构中包含了文件设备的 ID 和 seek 的偏移等。
o_mode 为打开类型

2.然后使用 file_read()函数读取 nbytes 字节到指定的缓冲区 ret->ret_buf，该缓冲区位于共享页中。
struct Fsreq_read *req = &ipc->read; struct Fsret_read *ret = &ipc->readRet;
上述两句是为了获得共享页中 read()的详细参数。

```
    // LAB 5: Your code here
    r = openfile_lookup(envid, req->req_fileid, &o);
    if (r < 0) {
        cprintf("serve_read: failed to lookup open file id\n");
        return r;
    }

    nbytes = file_read(o->o_file, (void *) ret->ret_buf, MIN(req->req_n, PGS
IZE), o->o_fd->fd_offset);
    if (nbytes > 0) {
        o->o_fd->fd_offset += nbytes;
    }

    return nbytes;
```

完成 devfile_read()函数：
1.根据上述的分析，我们知道 lib/fd.c 的 read()函数调用 lib/file.c 的 devfile_read()函数，该函数的功能给出读操作的具体细节(将要读的文件 id，需要读取的字节数)。
2.然后调用 fsipc 通知文件系统进程，我们知道 fsipc 调用 ipc_send()之后进入 ipc_recv()的堵塞态。当再次返回到 devfile_read()函数中，我们将共享页的缓冲区内容复制到指定缓冲区。

```
// LAB 5: Your code here
int r;

fsipcbuf.read.req_fileid = fd->fd_file.id;
fsipcbuf.read.req_n = n;

r = fsipc(FSREQ_READ, NULL);
if (r < 0)
    return r;

memmove(buf, fsipcbuf.readRet.ret_buf, r);
return r;
```

作业 6.
这个作业要求完成 fs/serv.c 的 serve_write()函数和 lib/file.c 的 devfile_write()函数

```
// LAB 5: Your code here.
int r;
struct OpenFile *o;
size_t nbytes;

r = openfile_lookup(envid, req->req_fileid, &o);
if (r < 0) {
    cprintf("serve_write: failed to lookup open fileid:%d\n",req->req_fileid);
    return r;
}

nbytes = MIN(req->req_n,PGSIZE - (sizeof(int) + sizeof(size_t)));
nbytes = file_write(o->o_file, (void *) req->req_buf, nbytes, o->o_fd->fd_offset);
//更新seek
if (nbytes >= 0) {
    o->o_fd->fd_offset += nbytes;
}

return nbytes;
```

该函数可以参照 serve_read()函数，将其中的 file_read()函数改为 file_write()函数，然后在写完之后更新 seek 偏移量。

devfile_write()函数：
跟 devfile_read()类似，只要区别：
devfile_write()首先需要将要写的数据复制到共享页面，而 devfile_read()是将 ipc 返回的共享页面的缓冲区的内容复制到指定缓冲。

```
// LAB 5: Your code here
uint32_t max = PGSIZE - (sizeof(int) + sizeof(size_t));

if (n > max)
    n = max;

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);
```

客户进程访问文件访问：
1.在 JOS 中，每个设备都被抽象为文件(设备文件)，对应一个 Dev 结构。就像之前我们接触的 disk 设备文件，便是对 disk 的抽象。
2.文件描述符表(file descriptor table)
该描述表被维护在 FSTABLE 以上的空间。
(1) JOS 为每一个打开的文件分配一个文件描述符，该描述符实际上是一个 Fd 结构，该结构在 inc/fd.h 中定义。

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

该结构体中的 fd_dev_id 标识设备文件 ID。在上面的读操作流程中，我们在 fd.c 的 read()可以看到，该函数根据 Fd 结构中的 fd_dev_id 通过 dev_loopup()查找指定文件的 Dev 结构(该结构中包含该种设备文件的读写删等操作函数的指针)。

(2)在查看 fd.c 的 read()函数时，可以看到 read()接受的 fdnum,如何通过 fdnum 来查找 fd 结构体？在这里就体现了 file descriptor table 的作用了。追踪到 fd_lookup 我们可以看到该函数使用宏定义 INDEX2FD，该宏定义利用 fd 到 fdt 首部的偏移来查找 fd 结构体。

(3)file descriptor table 的上限是 MAXFD(32)，也就是说应用程序最多可以同时打开 32 个文件。

作业 7.

要求我们完成 lib/file.c 的 open()函数，该函数打开指定路径的文件。

分析：

1.根据文件描述符表(file descriptor table)的分析，我们知道应用进程最多可以同时打开 32 个文件，所以在每次打开文件时首先使用 fd_alloc 检查 file descriptor table 是否还有空余（即是否已经到了打开文件的上限）.

2.然后在共享页中指定打开文件的类型和路径(因为 ipc_send 只能传递一个值，在该过程中用于传递请求命令 FSREQ_OPEN)

3,最后返回文件描述符的索引。

```
// LAB 5: Your code here.
int r;
struct Fd* fd;
if (strlen(path) >= MAXPATHLEN) {
    return -E_BAD_PATH;
}

r = fd_alloc(&fd);
if (r < 0)
    return r;

fsipcbuf.open.req_omode = mode;
strcpy(fsipcbuf.open.req_path, path);

if ((r = fsipc(FSREQ_OPEN, fd)) < 0) {
    fd_close(fd, 0);
    return r;
}

return fd2num(fd);
```

作业 8

根据 Spawning Process 的叙述，lib/spawn.c 创建一个子进程，然后从文件系统加载一个程序镜像，然后启动子进程来运行这个程序。

spawn 依赖 sys_env_set_trapframe。打开 kern/syscall.c，查看 sys_env_trapframe()函数，根据注释提示，我们只需要将指定 envid 的 tf 结构赋值为指定的 tf 即可。
然后在 syscall()函数中添加分支语句。

```
// LAB 5: Your code here.
// Remember to check whether the user has supplied us with a good
// address!
struct Env *e;
if (envid2env(envid, &e, 1) < 0)
    return -E_BAD_ENV;

if ((tf->tf_eip >= UTOP))
    return -1;

e->env_tf = *tf;
e->env_tf.tf_eflags |= FL_IF;
return 0;
```

lib/spawn.c 的 spawn()函数的具体流程：
1.打开程序镜像文件
2.读取镜像文件的头部，确保是可加载代码
3.使用 sys_exofork()创建一个子进程
4.把 child_tf 设置为初始的结构 Trapframe，其 eip 被设置为 elf_entry
5.然后使用 sys_set_trapframe()将子进程的 tf 栈赋值为 chilf_tf 的值
6.最后启动子进程，子进程便可以运行父进程为子进程指定的程序了。

make grade:

```
fs i/o [fs]: OK (1.0s)
check_bc [fs]: OK (1.0s)
check_super [fs]: OK (1.0s)
check_bitmap [fs]: OK (1.0s)
alloc_block [fs]: OK (1.0s)
file_open [fs]: OK (1.0s)
file_get_block [fs]: OK (1.0s)
file_flush/file_truncate/file rewrite [fs]: OK (1.0s)
lib/file.c [testfile]: OK (1.0s)
file_read [testfile]: OK (1.0s)
file_write [testfile]: OK (1.0s)
file_read after file_write [testfile]: OK (1.0s)
open [testfile]: OK (1.0s)
large file [testfile]: OK (1.0s)
motd display [writemotd]: OK (1.0s)
motd change [writemotd]: OK (1.0s)
spawn via icode [icode]: OK (1.0s)
Score: 105/105
```