操作系统 lab8 实验报告

小组成员：

- 2110408 吴振华
- 2112426 怀硕
- 2113635 王祎宁

练习1: 完成读文件操作的实现（需要编码）

首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，填写在kern/fs/sfs/sfs_inode.c中 的sfs_io_nolock()函数，实现读文件中数据的代码。

从 ucore 操作系统不同的角度来看，ucore 中的文件系统架构包含四类主要的数据结构, 它们分别是：

- **超级块（SuperBlock）**，它主要从文件系统的全局角度描述特定文件系统的全局信息。它的作用范围是整个 OS 空间。
- **索引节点（inode）**：它主要从文件系统的单个文件的角度它描述了文件的各种属性和数据所在位置。它的作用范围是整个 OS 空间。
- **目录项（dentry）**：它主要从文件系统的文件路径的角度描述了文件路径中的一个特定的目录项（注：一系列目录项形成目录/文件路径）。它的作用范围是整个 OS 空间。对于 SFS 而言，inode(具体为 struct sfs_disk_inode)对应于物理磁盘上的具体对象，dentry（具体为 struct sfs_disk_entry）是一个内存实体，其中的 ino 成员指向对应的 inode number，另外一个成员是 file name(文件名).
- **文件（file）**，它主要从进程的角度描述了一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。它的作用范围是某一具体进程。

文件系统，会将磁盘上的文件（程序）读取到内存里面来，在用户空间里面变成进程去进一步执行或其他操作。通过一系列系统调用完成这个过程。

ucore 模仿了 UNIX 的文件系统设计，ucore 的文件系统架构主要由四部分组成：

- **通用文件系统访问接口层**：该层提供了一个从用户空间到文件系统的标准访问接口。这一层访问接口让应用程序能够通过一个简单的接口获得 ucore 内核的文件系统服务。
- **文件系统抽象层**：向上提供一个一致的接口给内核其他部分（文件系统相关的系统调用实现模块和其他内核功能模块）访问。向下提供一个同样的抽象函数指针列表和数据结构屏蔽不同文件系统的实现细节。

- Simple FS **文件系统层**：一个基于索引方式的简单文件系统实例。向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数。向下访问外设接口
- **外设接口层**：向上提供 device 访问接口屏蔽不同硬件细节。向下实现访问各种具体设备驱动的接口，比如 disk 设备接口/串口设备接口/键盘设备接口等。

此处我们可以通过下图来理解上述四个部分的关系：

接下来分析读文件的详细处理的流程。

Read系统调用执行过程

读文件其实就是读出目录中的目录项，首先假定文件在磁盘上且已经打开。用户进程有如下语句：

```
read(fd, data, len);
```

即读取fd对应文件，读取长度为len，存入data中。下面来分析一下读文件的实现。

通用文件访问接口层的处理流程

先进入通用文件访问接口层的处理流程，即进一步调用如下用户态函数：read->sys_read->syscall，从而引起系统调用进入到内核态。

```
int
read(int fd, void *base, size_t len) {
    return sys_read(fd, base, len);
}
```

到了内核态以后，通过中断处理例程，会调用到sys_read内核函数，并进一步调用sysfile_read内核函数，进入到文件系统抽象层处理流程完成进一步读文件的操作。

```
static int
sys_read(uint64_t arg[]) {
    int fd = (int)arg[0];
    void *base = (void *)arg[1];
    size_t len = (size_t)arg[2];
    return sysfile_read(fd, base, len);
}
```

文件系统抽象层的处理流程

1. 检查错误，即检查读取长度是否为0和文件是否可读。

2. 分配buffer空间，即调用kmalloc函数分配4096字节的buffer空间。

3. 读文件过程

[1] 实际读文件

循环读取文件，每次读取buffer大小。每次循环中，先检查剩余部分大小，若其小于4096字节，则只读取剩余部分的大小。然后调用file_read函数（详细分析见后）将文件内容读取到buffer中，alen为实际大小。调用copy_to_user函数将读到的内容拷贝到用户的内存空间中，调整各变量以进行下一次循环读取，直至指定长度读取完成。最后函数调用层层返回至用户程序，用户程序收到了读到的文件内容。

[2] file_read函数

这个函数是读文件的核心函数。函数有4个参数，fd是文件描述符，base是缓存的基地址，len是要读取的长度，copied_store存放实际读取的长度。函数首先调用fd2file函数找到对应的file结构，并检查是否可读。调用filemap_acquire函数使打开这个文件的计数加1。调用vop_read函数将文件内容读到iob中（详细分析见后）。调整文件指针偏移量pos的值，使其向后移动实际读到的字节数iobuf_used(iob)。最后调用filemap_release函数使打开这个文件的计数减1，若打开计数为0，则释放file。

SFS文件系统层的处理流程

vop_read函数实际上是对sfs_read的包装。在sfs_inode.c中sfs_node_fileops变量定义了.vop_read = sfs_read，所以下面来分析sfs_read函数的实现。

```
static int
sfs_read(struct inode *node, struct iobuf *iob) {
    return sfs_io(node, iob, 0);
}
```

sfs_read函数调用sfs_io函数。它有三个参数，node是对应文件的inode，iob是缓存，write表示是读还是写的布尔值（0表示读，1表示写），这里是0。函数先找到inode对应sfs和sin，然后调用sfs_io_nolock函数进行读取文件操作，最后调用iobuf_skip函数调整iobuf的指针。

```
/* * sfs_io - Rd/Wr file. the wrapper of sfs_io_nolock            with lock protect */
static inline int
sfs_io(struct inode *node, struct iobuf *iob, bool write) {
    struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
    struct sfs_inode *sin = vop_info(node, sfs_inode);
    int ret;
    lock_sin(sin);
    {
        size_t alen = iob->io_resid;
        ret = sfs_io_nolock(sfs, sin, iob->io_base, iob->io_offset, &alen, write);
        if (alen != 0) {
            iobuf_skip(iob, alen);
        }
    }
    unlock_sin(sin);
    return ret;
}
```

在sfs_io_nolock函数中完成操作如下：

1. 先计算一些辅助变量，并处理一些特殊情况（比如越界），然后有sfs_buf_op = sfs_rbuf,sfs_block_op = sfs_rblock，设置读取的函数操作。
2. 接着进行实际操作，先处理起始的没有对齐到块的部分，再以块为单位循环处理中间的部分，最后处理末尾剩余的部分。
3. 每部分中都调用sfs_bmap_load_nolock函数得到blkno对应的inode编号，并调用sfs_rbuf或sfs_rblock函数读取数据（中间部分调用sfs_rblock，起始和末尾部分调用sfs_rbuf），调整相关变量。
4. 完成后如果offset + alen > din->fileinfo.size（写文件时会出现这种情况，读文件时不会出现这种情况，alen为实际读写的长度），则调整文件大小为offset + alen并设置dirty变量。

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t offset,
size_t *alenp, bool write) {
    struct sfs_disk_inode *din = sin->din;
    assert(din->type != SFS_TYPE_DIR);
    off_t endpos = offset + *alenp, blkoff;
    *alenp = 0;
    // calculate the Rd/Wr end position
    if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
        return -E_INVAL;
    }
    if (offset == endpos) {
        return 0;
    }
    if (endpos > SFS_MAX_FILE_SIZE) {
        endpos = SFS_MAX_FILE_SIZE;
    }
    if (!write) {
        if (offset >= din->size) {
            return 0;
        }
        if (endpos > din->size) {
            endpos = din->size;
        }
    }

    int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t blkno, off_t
offset);
    int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno, uint32_t
nblks);
    if (write) {
        sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
    }
    else {
        sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
    }

    int ret = 0;
    size_t size, alen = 0;
    uint32_t ino;
    uint32_t blkno = offset / SFS_BLKSIZE;          // The NO. of Rd/Wr begin block
    uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // The size of Rd/Wr blocks

  //LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf,
sfs_rblock,etc. read different kind of blocks in file
    /*     * (1) If offset isn't aligned with the first block, Rd/Wr some content from
offset to the end of the first block     *  NOTICE: useful function:
sfs_bmap_load_nolock, sfs_buf_op     *  Rd/Wr size = (nblks != 0) ? (SFS_BLKSIZE -
blkoff) : (endpos - offset)     * (2) Rd/Wr aligned blocks      *  NOTICE: useful
function: sfs_bmap_load_nolock, sfs_block_op     * (3) If end position isn't aligned
```

```
with the last block, Rd/Wr some content from begin to the (endpos % SFS_BLKSIZE) of
the last block     *  NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
*/

out:
    *alenp = alen;
    if (offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1;
    }
    return ret;
}
```

sfs_bmap_load_nolock函数将对应sfs_inode的第index个索引指向的block的索引值取出存到相应的指针指向的单元（ino_store）。它调用sfs_bmap_get_nolock来完成相应的操作。sfs_rbuf和sfs_rblock函数最终都调用sfs_rwblock_nolock函数完成操作，而sfs_rwblock_nolock函数调用dop_io->disk0_io->disk0_read_blks_nolock->ide_read_secs完成对磁盘的操作。

至此已对文件打开的处理流程有了大致的了解，可以完成对 `sfs_io_nolock` 的填写了，填充部分与相应解析如下：

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t offset,
size_t *alenp, bool write) {
    ...
    if ((blkoff = offset % SFS_BLKSIZE) != 0) {
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }
        alen += size;
        if (nblks == 0) {
            goto out;
        }
        buf += size, blkno ++, nblks --;
    }

    size = SFS_BLKSIZE;

    while (nblks != 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
            goto out;
        }
        alen += size, buf += size, blkno ++, nblks --;
    }

    if ((size = endpos % SFS_BLKSIZE) != 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
            goto out;
        }
        alen += size;
    }
    ...
}
```

- **对齐处理（Unaligned Part）**：

```
if ((blkoff = offset % SFS_BLKSIZE) != 0) {
    size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
        goto out;
    }
    alen += size;
    if (nblks == 0) {
        goto out;
    }
    buf += size, blkno ++, nblks --;
}
```

- 如果 `offset` 不是块对齐的，需要先处理不对齐的部分。
- `blkoff` 记录了在当前块内的偏移量。
- `size` 记录了需要读写的长度，可能是当前块的剩余部分，也可能是直到文件末尾的部分。
- 调用 `sfs_bmap_load_nolock` 获取块映射，并通过 `sfs_buf_op` 读写对应的块。
- 更新 `alen`，如果没有块了，直接跳出。

- **对齐块读写（Aligned Block I/O）**：

```
size = SFS_BLKSIZE;
while (nblks != 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
        goto out;
    }
    alen += size, buf += size, blkno ++, nblks --;
}
```

- 处理对齐的块，循环读写每个块。
- 调用 `sfs_bmap_load_nolock` 获取块映射，通过 `sfs_block_op` 读写对应的块。
- 更新 `alen`，移动缓冲区指针和块号，继续处理下一个块。

- **最后不对齐部分处理（Trailing Unaligned Part）**：

```
if ((size = endpos % SFS_BLKSIZE) != 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
        goto out;
    }
    alen += size;
}
```

- 处理最后不对齐的部分，即文件末尾的不足一个块的部分。
- 调用 `sfs_bmap_load_nolock` 获取块映射，并通过 `sfs_buf_op` 读写对应的块。
- 更新 `alen` 。

练习2: 完成基于文件系统的执行程序机制的实现（需要编码）

改写proc.c中的load_icode函数和其他相关函数，实现基于文件系统的执行程序机制。执行：make qemu。如果能看看到sh用户程序的执行界面，则基本成功了。如果在sh用户界面上可以执行”ls”,”hello”等其他放置在sfs文件系统中的其他执行程序，则可以认为本实验基本成功。

此处直接在 lab5 的基础上完成适配文件系统的几处修改即可，具体涉及到了 `proc.c` 中的如下函数：

```
alloc_proc
proc_run
do_fork
load_icode
```

各模块修改与对应解析如下：

`alloc_proc`

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1 YOUR CODE
    /*
     * below fields in proc_struct need to be initialized
     *       enum proc_state state;                      // Process state
     *       int pid;                                    // Process ID
     *       int runs;                                   // the running times of
Proces
     *       uintptr_t kstack;                           // Process kernel stack
     *       volatile bool need_resched;                 // bool value: need to be
rescheduled to release CPU?
     *       struct proc_struct *parent;                 // the parent process
     *       struct mm_struct *mm;                       // Process's memory
management field
     *       struct context context;                     // Switch here to run process
     *       struct trapframe *tf;                       // Trap frame for current
interrupt
     *       uintptr_t cr3;                              // CR3 register: the base
addr of Page Directroy Table(PDT)
     *       uint32_t flags;                             // Process flag
     *       char name[PROC_NAME_LEN + 1];               // Process name
     */
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);

     //LAB5 YOUR CODE : (update LAB4 steps)
    /*
     * below fields(add in LAB5) in proc_struct need to be initialized
     *       uint32_t wait_state;                        // waiting state
     *       struct proc_struct *cptr, *yptr, *optr;     // re lations between
processes
     */
        proc->wait_state = 0;  // 进程等待状态初始化为0
        proc->cptr = proc->yptr = proc->optr = NULL;  // 进程间指针初始化为NULL

    //LAB6 YOUR CODE : (update LAB5 steps)
```

```
    /*
     * below fields(add in LAB6) in proc_struct need to be initialized
     *     struct run_queue *rq;                       // running queue contains
Process
     *     list_entry_t run_link;                      // the entry linked in run
queue
     *     int time_slice;                             // time slice for occupying the
CPU
     *     skew_heap_entry_t lab6_run_pool;            // FOR LAB6 ONLY: the entry in
the run pool
     *     uint32_t lab6_stride;                       // FOR LAB6 ONLY: the current
stride of the process
     *     uint32_t lab6_priority;                     // FOR LAB6 ONLY: the priority
of process, set by lab6_set_priority(uint32_t)
     */

     //LAB8 YOUR CODE : (update LAB6 steps)
      /*
     * below fields(add in LAB6) in proc_struct need to be initialized
     *       struct files_struct * filesp;               file struct point
     */
        proc->filesp = NULL; // filesp为指向文件描述符表的指针，初始化为 NULL表示该进程还
没有打开任何文件
    }
    return proc;
}
```

与 lab5 相比增加了 `lab8` 相关字段的初始化：

- `proc->filesp = NULL;`：
    - `filesp` 为指向文件描述符表的指针，这里将其初始化为 `NULL`，表示该进程还没有打开任何文件。

`proc_run`

```
// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load  base addr of "proc"'s new PDT
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
         // LAB4:EXERCISE3 YOUR CODE
        /*
        * Some Useful MACROs, Functions and DEFINEs, you can use them in below
implementation.
        * MACROs or Functions:
        *   local_intr_save():        Disable interrupts
        *   local_intr_restore():     Enable Interrupts
        *   lcr3():                   Modify the value of CR3 register
        *   switch_to():              Context switching between two processes
        */
        // 禁用中断
        bool intr_flag;
        struct proc_struct *prev = current;  // prev指向当前正在运行的进程
        local_intr_save(intr_flag);
        {
            // 切换当前进程为要运行的进程
            current = proc;
            // 切换页表
            lcr3(proc->cr3);

            flush_tlb(); //页目录表的切换可能导致 TLB 中的缓存的映射关系不再有效

            // 切换上下文，只切换context，不使用tf
            switch_to(&(prev->context), &(proc->context)); // prev.ra <- 返回值，转移到
forkret
        }
        local_intr_restore(intr_flag);

       //LAB8 YOUR CODE : (update LAB4 steps)
      /*
     * below fields(add in LAB6) in proc_struct need to be initialized
    *       before switch_to();you should flush the tlb
    *        MACROs or Functions:
     *       flush_tlb():          flush the tlb
     */

    }
}
```

添加 `flush_tlb` 语句，以在切换页目录表后对 TLB 刷新。从理论角度这是必要的，因为页目录表的切换可能导致 TLB 中的缓存的映射关系不再有效。通过刷新 TLB，可以确保新的进程的地址空间映射关系得到正确加载，避免出现地址转换错误。

`do_fork`

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;

    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = current;
    assert(current->wait_state == 0);

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    if (copy_files(clone_flags, proc) != 0) { //for LAB8
        goto bad_fork_cleanup_kstack;
    }
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_fs;
    }
    copy_thread(proc, stack, tf);

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        set_links(proc);
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);

    ret = proc->pid;
fork_out:
    return ret;

bad_fork_cleanup_fs:  //for LAB8
    put_files(proc);
bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

添加 `copy_files` 语句来处理文件描述符的复制。在其实现中，当 `copy_files` 函数返回非零值时，表示文件描述符的复制出现了错误。此时，通过 `goto bad_fork_cleanup_kstack;` 将控制流转移到 `bad_fork_cleanup_kstack` 标签，执行相应的清理工作，确保在出现错误的情况下，不会留下未释放的资源，并返回错误码，以便上层调用处理异常。

`load_icode`

```c
static int
load_icode(int fd, int argc, char **kargv) {
    assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
    //创建新的 mm（内存管理结构）
    struct mm_struct *mm;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    //设置新的页目录
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    struct Page *page;

    struct elfhdr __elf, *elf = &__elf;
    //读取 ELF 头部信息
    if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
        goto bad_elf_cleanup_pgdir;
    }
    //验证 ELF 头部
    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    struct proghdr __ph, *ph = &__ph;
    uint32_t vm_flags, perm, phnum;
    //循环处理程序头部
    for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
        off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;
        if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0) {
            goto bad_cleanup_mmap;
        }
        if (ph->p_type != ELF_PT_LOAD) {
            continue ;
        }
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) {
            // continue ;
            // do nothing here since static variables may not occupy any space
        }
```

```
        vm_flags = 0, perm = PTE_U | PTE_V;
        if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
        // modify the perm bits here for RISC-V
        if (vm_flags & VM_READ) perm |= PTE_R;
        if (vm_flags & VM_WRITE) perm |= (PTE_W | PTE_R);
        if (vm_flags & VM_EXEC) perm |= PTE_X;
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
            goto bad_cleanup_mmap;
        }
        off_t offset = ph->p_offset;
        size_t off, size;
        uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);

        ret = -E_NO_MEM;

        end = ph->p_va + ph->p_filesz;
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                ret = -E_NO_MEM;
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) != 0)
{
                goto bad_cleanup_mmap;
            }
            start += size, offset += size;
        }
        end = ph->p_va + ph->p_memsz;

        if (start < la) {
            /* ph->p_memsz == ph->p_filesz */
            if (start == end) {
                continue ;
            }
            off = start + PGSIZE - la, size = PGSIZE - off;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
            assert((end < la && start == end) || (end >= la && start == la));
        }
        while (start < end) {
            if ((page = pgdir_alloc_pagFe(mm->pgdir, la, perm)) == NULL) {
                ret = -E_NO_MEM;
```

```
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
        }
    }
    //关闭文件
    sysfile_close(fd);

    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    //建立用户栈
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) != 0) {
        goto bad_cleanup_mmap;
    }
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);

    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    //setup argc, argv
    //设置用户程序参数
    uint32_t argv_size=0, i;
    for (i = 0; i < argc; i ++) {
        argv_size += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1)+1;
    }
    //计算参数在用户栈的位置
    uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
    char** uargv=(char **)(stacktop  - argc * sizeof(char *));

    //将参数字符串拷贝到用户栈中
    argv_size = 0;
    for (i = 0; i < argc; i ++) {
        uargv[i] = strcpy((char *)(stacktop + argv_size ), kargv[i]);
        argv_size +=  strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }

    stacktop = (uintptr_t)uargv - sizeof(int);
    *(int *)stacktop = argc;

    struct trapframe *tf = current->tf;
    // Keep sstatus
    uintptr_t sstatus = tf->status;
```

```
    memset(tf, 0, sizeof(struct trapframe));
    tf->gpr.sp = stacktop;
    tf->epc = elf->e_entry;
    tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
    ret = 0;
out:
    return ret;
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;

}
```

`load_icoed` 函数主要用于从磁盘上（此处即同之前实现的主要区别，不是从内存直接读取）加载用户程序的二进制文件（ELF格式）到当前进程的内存中，并设置进程的上下文，包括内存映射、栈设置、参数传递等。具体的分步解析如下：

1. **创建新的 mm（内存管理结构）**：调用 `mm_create` 创建一个新的 `mm_struct` 结构。
2. **设置新的页目录**：调用 `setup_pgdir` 在新的 `mm` 结构中设置页目录，并将其赋给 `mm->pgdir`。
3. **读取 ELF 头部信息**：使用 `load_icode_read` 从文件中读取 ELF 头部信息，包括程序入口地址等。
4. **验证 ELF 头部**：检查 ELF 头部的魔数是否正确，确保文件是合法的 ELF 文件。
5. **循环处理程序头部（Program Header）**：遍历 ELF 文件中的程序头部，处理每个程序头部。
   a. **判断程序头部类型**：如果是 `ELF_PT_LOAD` 类型，表示这是一个可加载的段。
   b. **根据类型处理**：
      - 计算 `vm_flags` 和 `perm` 标志：这里涉及虚拟内存区域的权限标志。
      - 使用 `mm_map` 创建新的虚拟内存区域，并映射到对应的物理内存。
      - 使用 `pgdir_alloc_page` 分配物理页面，并将 ELF 文件中的内容复制到这些页面上。
6. **关闭文件**：使用 `sysfile_close` 关闭文件。

7. **建立用户栈**：使用 `mm_map` 在用户栈的位置建立一个新的虚拟内存区域，即用户栈。同时，分配了一些页面用于用户栈。

8. **设置当前进程的 mm、cr3 寄存器**：将新创建的 `mm` 结构关联到当前进程，并设置当前进程的页目录地址。

9. **设置用户程序参数**：

   a. 计算参数在用户栈的位置。

   b. 将参数字符串拷贝到用户栈中。

   c. 设置用户栈指针（`sp`）和用户程序入口地址（`epc`）。

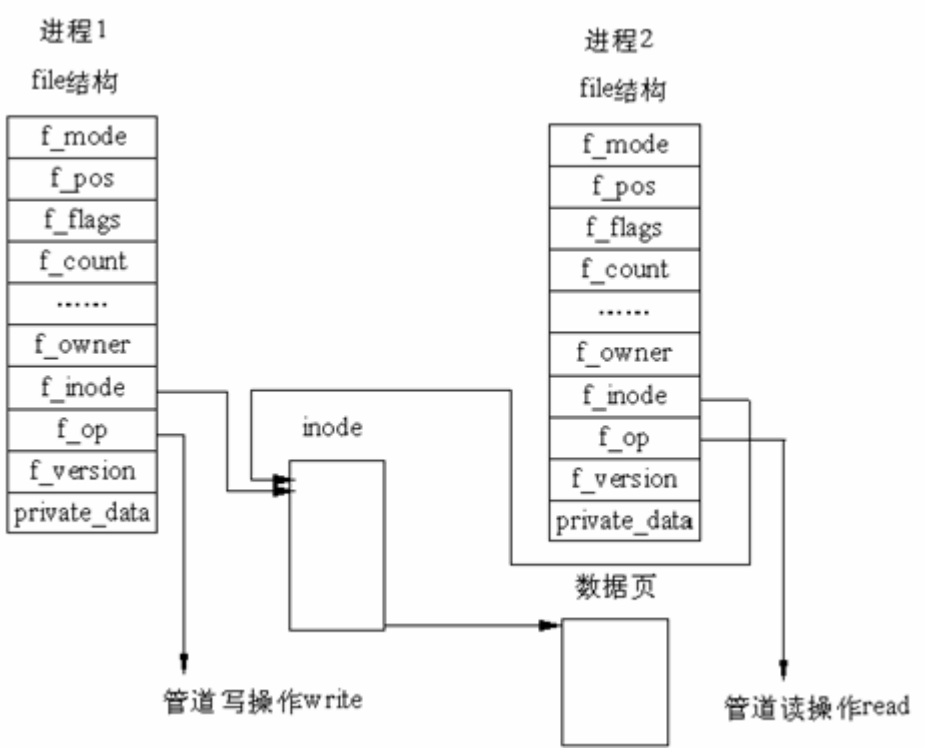
   d. 设置状态寄存器 `sstatus`。

10. **清理**：如果发生错误，执行相应的清理工作，包括释放内存、关闭文件等。

扩展练习 Challenge1：完成基于 UNIX 的 PIPE 机制 的设计方案

如果要在ucore里加入UNIX的管道（Pipe)机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个(或多个）具体的C语言struct定义。在网络上查找相关的Linux资料和实现，请在实验报告中给出设计实现”UNIX的PIPE机制“的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。）

在 Linux 中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的 file 结构和 VFS 的索引节点 inode。通过将两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面从而实现的.



管道可以看作是由内核管理的一个缓冲区，一端连接进程 A 的输出，另一端连接进程 B 的输入。进程 A 会向管道中放入信息，而进程 B 会取出被放入管道的信息。

```
process1 -[write]-> pipe(in kernel) -[read]-> process2
```

当管道中没有信息，进程 B 会等待，直到进程 A 放入信息。当管道被放满信息的时候，进程A会等待，直到进程 B 取出信息。当两个进程都结束的时候，管道也自动消失。管道基于fork 机制建立，从而让两个进程可以连接到同一个 PIPE 上。

基于上述理论基础，实现上，可以借助文件系统的file结构和VFS的索引节点inode。
将两个file结构指向同一个**临时的VFS索引节点**，而这个VFS索引节点指向一个物理数据页。

```
file1.inode   ---> inode <---   file2.inode
       write         ↓         read
                  data page
```

当进程向管道写入时，利用标准库函数`write()`，系统根据库函数传递的文件描述符找到文件的file结构。
file结构拿到特定函数地址进行写入，写入时锁定内存，接着将进程地址空间的数据复制到内存。
若不能获取到锁或不能写入，则休眠，进入等待队列。
当有空间可以写入或内存解锁时，读取进程唤醒写入进程。
写入进程收到信号，写入数据后，唤醒休眠的读取进程进行读取。

扩展练习 Challenge2：完成 基于 UNIX 的软连接和硬连接机制的设计方案

如果要在ucore里加入UNIX的软连接和硬连接机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个(或多个）具体的C语言struct定义。在网络上查找相关的Linux资料和实现，请在实验报告中给出设计实现"UNIX的软连接和硬连接机制"的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。）

分别对硬链接与软链接进行分析：

硬链接存在以下几点特性：

- 文件有相同的 inode 及 data block；
- 只能对已存在的文件进行创建；
- 不能交叉文件系统进行硬链接的创建；
- 不能对目录进行创建，只可对文件创建；
- 删除一个硬链接文件并不影响其他有相同 inode 号的文件

而软链接的创建与使用没有类似硬链接的诸多限制：

- 软链接有自己的文件属性及权限等；
- 可对不存在的文件或目录创建软链接；
- 软链接可交叉文件系统；
- 软链接可对文件或目录创建；
- 创建软链接时，链接计数 i_nlink 不会增加；
- 删除软链接并不影响被指向的文件，但若被指向的原文件被删除，则相关软连接被称为死链接（即 dangling link，若被指向路径文件被重新创建，死链接可恢复为正常的软链接）。

且由于存在磁盘上的 inode 信息均存在一个 `nlinks` 变量用于表示当前文件的被链接的计数，因而支持实现硬链接和软链接机制；

- **创建硬链接 link 时**，为 new_path 创建对应的 file，并把其 inode 指向 old_path 所对应的 inode，inode 的引用计数加1。
- **创建软链接 link 时**，创建一个新的文件（inode 不同），并把 old_path的内容存放到- 文件的内容中去，将该文件保存在磁盘上时 disk_inode 类型为 `SFS_TYPE_LINK`，再完善对于该类型 inode 的操作即可。
- **删除一个软链接 B 时**，直接将其在磁盘上的 inode 删掉即可
- **删除一个硬链接 B 时**，除了需要删除掉 B 的 inode 之外，还需要将 B 指向的文件 A 的被链接计数减1，如果减到了0，则需要将A删除掉；
- 访问软/硬链接的方式是一致的；

具体设计方案如下：

- **创建硬链接：**
  - 在创建硬链接时，为新的链接路径 `new_path` 创建一个新的文件（`inode`）。
  - 将新文件的 `inode` 指向被链接路径 `old_path` 所对应的 `inode`。
  - 增加 `old_path` 对应 `inode` 的被链接计数 `nlinks`。
- **创建软链接：**
  - 在创建软链接时，创建一个新的文件（`inode` 不同），并将 `target_path` 的内容存放到文件的内容中。
  - 在磁盘上保存时，文件的 `disk_inode` 类型标记为 `SFS_TYPE_LINK`，需要完善对该类型 `inode` 的操作。
- **删除链接：**
  - 在删除软链接时，直接删除其在磁盘上的 `inode`。
  - 在删除硬链接时，除了删除 `new_path` 的 `inode` 外，还需减少被链接文件 `old_path` 的 `inode` 的被链接计数 `nlinks`。
    - 如果 `nlinks` 减到 0，需要将 `old_path` 的 `inode` 删除。
- **访问链接：**
  - 访问软链接和硬链接的方式是一致的，通过链接路径即可访问到被链接的文件。
- **同步互斥问题处理：**
  - 在修改 `nlinks` 和删除链接等操作上使用锁机制，确保对链接结构的访问是原子的。
  - 对硬链接计数进行适当的同步，防止多个进程同时修改硬链接计数导致数据不一致。

硬链接和软链接的区别：

1. **物理位置**：
   - **硬链接**：多个硬链接指向同一个 inode（索引节点），它们在磁盘上实际上是同一个文件。删除其中一个硬链接并不会影响其他硬链接，只有当所有硬链接都被删除时，文件的内容才会被释放。
   - **软链接**：软链接是一个特殊的文件，其中包含指向另一个文件的路径。软链接本身有一个独立的 inode，指向的文件有一个独立的 inode。删除原始文件后，软链接仍然存在，但失效，称为悬挂链接。
2. **跨文件系统**：

- **硬链接**：只能在同一文件系统中创建硬链接，因为硬链接实际上是同一个 inode 的多个引用，而 inode 是与文件系统紧密相关的。
- **软链接**：可以跨越文件系统，因为软链接只包含目标文件的路径信息，而不是 inode 信息。

3. **文件类型**：
   - **硬链接**：只能链接普通文件，不能链接目录或特殊文件。
   - **软链接**：可以链接普通文件、目录，甚至是跨文件系统的文件。

4. **大小和权限**：
   - **硬链接**：多个硬链接共享相同的 inode 和文件内容，因此它们在磁盘上占用相同的空间。硬链接不能更改文件权限，因为它们指向同一个 inode。
   - **软链接**：软链接占据磁盘上的额外空间，因为它们有自己的 inode 和路径信息。软链接有自己的权限设置，可以有不同的权限。

5. **创建和删除**：
   - **硬链接**：只能在同一文件系统中创建。
   - **软链接**：可以在不同文件系统中创建。

硬链接与软链接的数目问题：

在实现上，软链接的数量通常可以比硬链接更多，因为软链接是独立的文件，每个软链接都有自己的 inode。相比之下，硬链接共享相同的 inode，因此受到文件系统和操作系统的限制，其数量可能受到一些限制。

硬链接数量的限制主要取决于文件系统的实现。一些文件系统对硬链接数量有较低的限制，而另一些文件系统可能允许更多的硬链接。在大多数情况下，硬链接数量的限制可能是文件系统实现的技术限制，而不是硬链接本身的限制。

软链接的数量理论上可以非常大，因为每个软链接都是一个独立的文件，有自己的 inode。软链接数量受到文件系统和存储空间的限制，但这些限制通常相对较高，因此在实践中，软链接的数量通常可以很大。