lab8 report

练习0：填写已有实验

使用`meld`填写已有实验。

对之前实验代码的修改：

`proc.c:alloc_proc`

增加对进程相关文件信息的初始化

```
proc->filesp = NULL;
```

`proc.c:do_fork`

fork时增加对打开文件信息的复制：

```c
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
...
    //    3. call copy_mm to dup OR share mm according clone_flag
    if(copy_mm(clone_flags,proc)!=0){
        goto bad_fork_cleanup_kstack;
    }

    // for lab8
    if(copy_files(clone_flags,proc)!=0){
        goto bad_fork_cleanup_fs;
    }

    //    4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc,stack,tf);
...
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;
}
```

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，编写在sfs_inode.c中sfs_io_nolock读文件中数据的实现代码。

请在实验报告中给出设计实现"UNIX的PIPE机制"的概要设方案，鼓励给出详细设计方案

实现思路

在sfs_io_nolock函数中，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。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。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。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。

相关函数

`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。

关键代码

先处理起始的没有对齐到块的部分

```
blkoff = offset % SFS_BLKSIZE;
if (blkoff != 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 --;
}
```

再以块为单位循环处理中间的部分

```
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 --;
}
```

最后处理末尾剩余的部分

```
    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;
    }
```

UNIX的PIPE机制

用先进先出队列做缓冲区。进程写的数据时，若缓冲区没有满，入队，否则将进程阻塞。当缓冲区不是空时，进程可以读数据，出队，否则也将进程阻塞。需要考虑到同步互斥问题，可用信号量或者管程来实现。

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。

请在实验报告中给出设计实现基于"UNIX的硬链接和软链接机制"的概要设方案，鼓励给出详细设计方案

`load_icode`

按照提示将原来从内存读取、拷贝程序的操作转换为从磁盘读取、拷贝。具体步骤见注释。

```
static int
load_icode(int fd, int argc, char **kargv) {
    /* LAB8:EXERCISE2 2014011336  HINT:how to load the file with handler fd in to process's memory? how to setup argc/argv?
     * MACROs or Functions:
     *  mm_create        - create a mm
     *  setup_pgdir      - setup pgdir in mm
     *  load_icode_read  - read raw data content of program file
     *  mm_map           - build new vma
     *  pgdir_alloc_page - allocate new memory for  TEXT/DATA/BSS/stack parts
     *  lcr3             - update Page Directory Addr Register -- CR3
     */
    /* (1) create a new mm for current process
     * (2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
     * (3) copy TEXT/DATA/BSS parts in binary to memory space of process
     *    (3.1) read raw data content in file and resolve elfhdr
```

```
     *    (3.2) read raw data content in file and resolve proghdr based on info in elfhdr
     *    (3.3) call mm_map to build vma related to TEXT/DATA
     *    (3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read contents in file
     *          and copy them into the new allocated pages
     *    (3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero in these pages
     * (4) call mm_map to setup user stack, and put parameters into user stack
     * (5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)
     * (6) setup uargc and uargv in user stacks
     * (7) setup trapframe for user environment
     * (8) if up steps failed, you should cleanup the env.
     */
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
    struct mm_struct *mm;
    //(1) create a new mm for current process
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    //(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    //(3) copy TEXT/DATA/BSS parts in binary to memory space of process
    struct Page *page;

    //(3.1) read raw data content in file and resolve elfhdr
    struct elfhdr __elf, *elf = &__elf;
    if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
        goto bad_elf_cleanup_pgdir;
    }

    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;
    for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
    //(3.2) read raw data content in file and resolve proghdr based on info in elfhdr
```

```
        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 ;
        }
    //(3.3) call mm_map to build vma related to TEXT/DATA
        vm_flags = 0, perm = PTE_U;
        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;
        if (vm_flags & VM_WRITE) perm |= PTE_W;
        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;
     //(3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read contents in file
     //          and copy them into the new allocated pages
        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;
```

```
        }

      //(3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero in these pages
        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_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;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
        }
    }
    sysfile_close(fd);

    //(4) call mm_map to setup user stack, and put parameters into user stack
    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    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);
```

```c
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);

    //(5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    //(6) setup uargc and uargv in user stacks
    uint32_t argv_size=0, i;
    for (i = 0; i < argc; i ++) {
        argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }

    uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
    char** uargv=(char **)(stacktop  - argc * sizeof(char *));

    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;

    //(7) setup trapframe for user environment
    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));
    /* LAB5:EXERCISE1 2014011336
     * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
     * NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE from kernel. So
     *          tf_cs should be USER_CS segment (see memlayout.h)
     *          tf_ds=tf_es=tf_ss should be USER_DS segment
     *          tf_esp should be the top addr of user stack (USTACKTOP)
     *          tf_eip should be the entry point of this binary program (elf->e_entry)
     *          tf_eflags should be set to enable computer to produce Interrupt
     */
    tf->tf_cs = USER_CS;
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
    tf->tf_esp = stacktop;
    tf->tf_eip = elf->e_entry;
    tf->tf_eflags = FL_IF;
    ret = 0;
```

```
    //(8) if up steps failed, you should cleanup the env.
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;
}
```

示例执行结果

ls

```
ls
 @ is  [directory] 2(hlinks) 23(blocks) 5888(bytes) : @'.'
   [d]   2(h)       23(b)     5888(s)   .
   [d]   2(h)       23(b)     5888(s)   ..
   [-]   1(h)       10(b)    40224(s)   testbss
   [-]   1(h)       10(b)    40204(s)   badsegment
   [-]   1(h)       10(b)    40204(s)   faultread
   [-]   1(h)       10(b)    40360(s)   ls
   [-]   1(h)       10(b)    40200(s)   softint
   [-]   1(h)       10(b)    40292(s)   priority
   [-]   1(h)       10(b)    40332(s)   waitkill
   [-]   1(h)       10(b)    40220(s)   divzero
   [-]   1(h)       10(b)    40192(s)   pgdir
   [-]   1(h)       10(b)    40204(s)   sleepkill
   [-]   1(h)       10(b)    40252(s)   forktree
   [-]   1(h)       10(b)    40208(s)   faultreadkernel
   [-]   1(h)       10(b)    40200(s)   badarg
   [-]   1(h)       10(b)    40200(s)   hello
   [-]   1(h)       10(b)    40220(s)   sleep
   [-]   1(h)       10(b)    40228(s)   forktest
   [-]   1(h)       10(b)    40224(s)   exit
   [-]   1(h)       10(b)    40304(s)   matrix
   [-]   1(h)       10(b)    40196(s)   spin
   [-]   1(h)       10(b)    40200(s)   yield
   [-]   1(h)       11(b)    44508(s)   sh
lsdir: step 4
```

forktest

```
$ forktest
main-loop: WARNING: I/O thread spun for 1000 iterations
I am child 31
I am child 30
I am child 29
I am child 28
I am child 27
I am child 26
I am child 25
I am child 24
I am child 23
I am child 22
I am child 21
I am child 20
I am child 19
I am child 18
I am child 17
I am child 16
I am child 15
I am child 14
I am child 13
I am child 12
I am child 11
I am child 10
I am child 9
I am child 8
I am child 7
I am child 6
I am child 5
I am child 4
I am child 3
I am child 2
I am child 1
I am child 0
forktest pass.
```

UNIX的硬链接和软链接机制

- 硬链接：在文件描述符中加入引用标记和引用文件指针，标记位为1表示硬链接，指针指向链接的文件。删除文件时进行判断，如果被引用文件的引用次数标记为0则删除该文件。
- 软链接：直接拷贝文件对应的 `inode` 信息。

与参考答案的区别

练习0

参考答案在 `copy_mm` 失败时会跳到 `bad_fork_cleanup_fs`，而在 `copy_files` 失败的时候会跳到 `bad_fork_cleanup_kstack`，明显写反了，希望能改过来。

练习1,2

基本相同

知识点

列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

- 文件系统架构。原理中很清楚的文件系统架构到了具体ucore的实现中则需要好好捋一捋。我自下而上的分析了文件系统，收获很大。
- 进程的文件管理。打开文件，读写操作在实验中都有体现。

列出你认为OS原理中很重要，但在实验中没有对应上的知识点

- 文件系统的挂载
- 路径访问