Lab8

练习0：填写已有实验

本实验需要补充 `kern/process/proc.c` 中的 `alloc_proc`，`proc_run`，`do_fork` 和 `load_icode`：

`alloc_proc` 函数中新增：

```
proc->filesp = NULL;
//其中filesp是进程控制块PCB的新增成员变量，
//filesp是一个指向files_struct类型的指针，它指向当前进程的文件相关信息
```

`do_fork` 函数中新增：

```
    // LAB8 将当前进程的fs（File System）复制到fork出的进程中
    if (copy_files(clone_flags, proc) != 0) { //for LAB8
        goto bad_fork_cleanup_kstack;
    }
    //......
    // LAB8 如果复制失败，则需要重置原先的操作
    bad_fork_cleanup_fs:  //for LAB8
    put_files(proc);
```

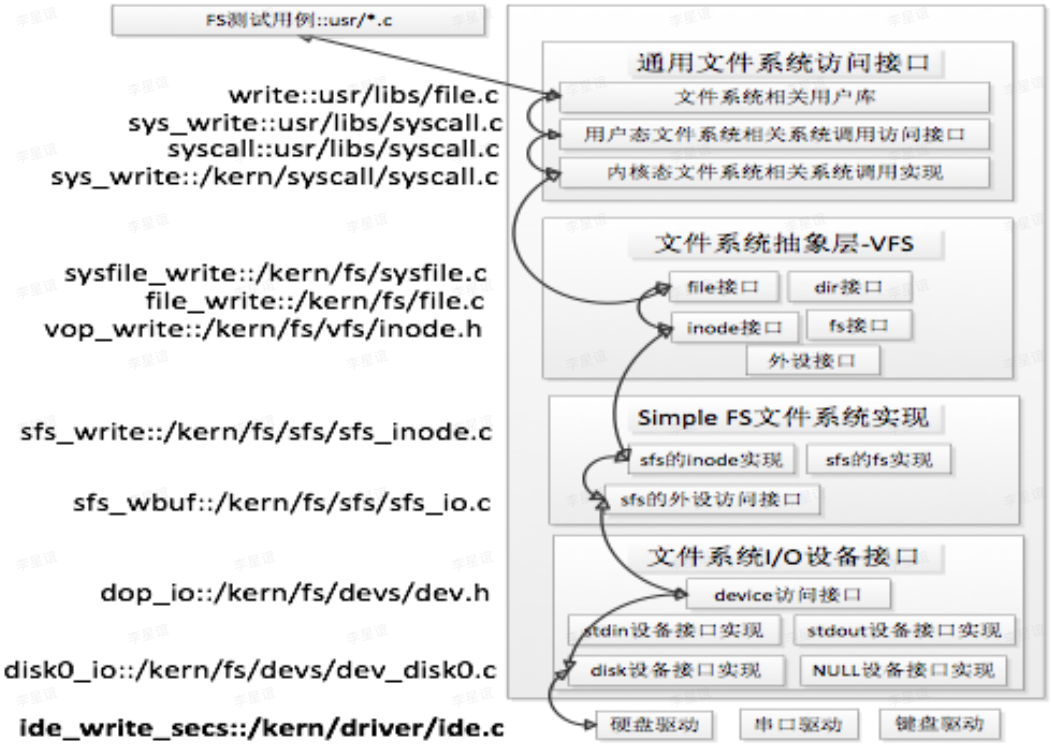
`proc_run` 函数中新增：

```
flush_tlb();//在switch_to之前应该先刷新TLB，在lab6中增加的
```

因为加载ELF文件方式不同，`load_icode` 在本实验中改动最大，具体见练习2。

练习1：完成读文件操作的实现

文件系统的访问处理过程（以用户态写函数write的执行过程为例）：



当进行文件读取/写入操作时，用户/内核从给定的通用文件系统访问接口中调用抽象文件系统函数，经过检查后交由具体文件系统（在ucore中是SFS）的具体实现函数实现，如：

- `sfs_lookup` 实现以“/”为分割符，从左至右逐一分解path获得各个子目录和最终文件对应的inode节点；
- `sfs_io_nolock` 实现读取文件操作；需要注意的是，`sfs_read`，`sfs_write` 都会调用 `sfs_io`，进而在其中调用 `sfs_io_nolock`。

因此在文件读写操作的执行流中，一定会调用 `sfs_io_nolock` 函数。在该函数中，我们要完成对设备上基础块数据的**读取与写入**。

在进行读取/写入前，我们需要先将数据与基础块对齐，以便于使用 `sfs_block_op` 函数来操作基础块，提高读取/写入效率。但将数据对齐后会出现以下问题：

- 待操作数据的前一小部分有可能在最前的一个基础块的末尾位置
- 待操作数据的后一小部分有可能在最后的一个基础块的起始位置

我们需要分别对这**第一**和**最后**这两个位置的基础块进行读写/写入，因为**这两个位置的基础块所涉及到的数据都是部分的**。而中间的数据由于已经对齐好基础块了，所以可以直接调用 `sfs_block_op` 来读取/写入数据。以下是相关操作的实现：

```
    // 对齐偏移。如果偏移没有对齐第一个基础块，读取/写入第一个基础块的末尾数据
    if ((blkoff = offset % SFS_BLKSIZE) != 0) {

        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
        //如果有多个基础块(nblks不等于0)，则需要读取/写入第一个基础块的末尾数据
(SFS_BLKSIZE - blkoff);
```

```
        //否则，只需要读取/写入到文件结束位置(endpos - offset)

        // 获取第一个基础块所对应的block的编号`ino`
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0)
            goto out;
        // 通过上一步取出的`ino`，读取/写入一部分第一个基础块的末尾数据
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0)
            goto out;
        alen += size;
        if (nblks == 0)
            goto out;
        buf += size, blkno ++, nblks --;
    }

    // 循环读取/写入对齐好的数据
    size = SFS_BLKSIZE;
    while (nblks != 0) {
        // 获取inode对应的基础块编号
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0)
            goto out;
        // 单次读取/写入一基础块的数据
        if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0)
            goto out;
        alen += size, buf += size, blkno ++, nblks --;
    }
    // 如果末尾位置没有与最后一个基础块对齐，读取/写入一点末尾基础块的数据
    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;
    }
```

练习2：完成基于文件系统的执行程序机制的实现

相比于lab5，lab8中的 `load_icode` 主要有两点改动：

1. 从文件系统中读取ELF header，而不是之前的内存；
2. 对 `do_execve` 所执行的程序传入参数。

其余步骤中，也有基于文件系统新添加的一些操作，具体的代码实现如下，思路见注释：

```
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proc的内存管理结构
    struct mm_struct *mm;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    struct Page *page;
    // LAB8 这里要从文件中读取ELF header，而不是Lab7中的内存了
    struct elfhdr __elf, *elf = &__elf;
    if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
        goto bad_elf_cleanup_pgdir;
    }
    // 判断读取入的elf header是否正确
    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }
    // 根据每一段的大小和基地址来分配不同的内存空间
    struct proghdr __ph, *ph = &__ph;
    uint32_t vm_flags, perm, phnum;
    for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
        // LAB8 从文件特定偏移处读取每个段的详细信息（包括大小、基地址等等）
        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 ;
        }
        vm_flags = 0, perm = PTE_U;
```

```
50          if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
51          if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
52          if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
53          if (vm_flags & VM_WRITE) perm |= PTE_W;
54          // 为当前段分配内存空间
55          if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
56              goto bad_cleanup_mmap;
57          }
58          off_t offset = ph->p_offset;
59          size_t off, size;
60          uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);
61
62          ret = -E_NO_MEM;
63
64          end = ph->p_va + ph->p_filesz;
65          while (start < end) {
66              // 设置该内存所对应的页表项
67              if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
68                  ret = -E_NO_MEM;
69                  goto bad_cleanup_mmap;
70              }
71              off = start - la, size = PGSIZE - off, la += PGSIZE;
72              if (end < la) {
73                  size -= la - end;
74              }
75              // LAB8 读取elf对应段内的数据并写入至该内存中
76              if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset))
77                  goto bad_cleanup_mmap;
78              }
79              start += size, offset += size;
80          }
81          end = ph->p_va + ph->p_memsz;
82          // 对于段中当前页中剩余的空间（复制elf数据后剩下的空间），将其置为0
83          if (start < la) {
84              /* ph->p_memsz == ph->p_filesz */
85              if (start == end) {
86                  continue ;
87              }
88              off = start + PGSIZE - la, size = PGSIZE - off;
89              if (end < la) {
90                  size -= la - end;
91              }
92              memset(page2kva(page) + off, 0, size);
93              start += size;
94              assert((end < la && start == end) || (end >= la && start == la));
95          }
96          // 对于段中剩余页中的空间（复制elf数据后的多余页面），将其置为0
```

```
97          while (start < end) {
98              if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
99                  ret = -E_NO_MEM;
100                 goto bad_cleanup_mmap;
101             }
102             off = start - la, size = PGSIZE - off, la += PGSIZE;
103             if (end < la) {
104                 size -= la - end;
105             }
106             memset(page2kva(page) + off, 0, size);
107             start += size;
108         }
109     }
110     // 关闭读取的ELF
111     sysfile_close(fd);
112
113     // 设置栈内存
114     vm_flags = VM_READ | VM_WRITE | VM_STACK;
115     if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) !
116         goto bad_cleanup_mmap;
117     }
118     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);
119     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE , PTE_USER) != NULL);
120     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE , PTE_USER) != NULL);
121     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);
122
123     mm_count_inc(mm);
124     //(5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)
125     // 设置CR3页表相关寄存器，同lab5
126     current->mm = mm;
127     current->cr3 = PADDR(mm->pgdir);
128     lcr3(PADDR(mm->pgdir));
129
130     //(6) setup uargc and uargv in user stacks
131     // LAB8 设置execve所启动的程序参数
132     uint32_t argv_size=0, i;
133     for (i = 0; i < argc; i ++) {
134         argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
135     }
136
137     uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
138     // 直接将传入的参数压入至新栈的底部
139     char** uargv=(char **)(stacktop  - argc * sizeof(char *));
140
141     argv_size = 0;
142     for (i = 0; i < argc; i ++) {
143         uargv[i] = strcpy((char *)(stacktop + argv_size ), kargv[i]);
```

```
144         argv_size +=  strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
145     }
146 
147     stacktop = (uintptr_t)uargv - sizeof(int);
148     *(int *)stacktop = argc;
149 
150     //(7) setup trapframe for user environment
151     struct trapframe *tf = current->tf;
152 
153     uintptr_t sstatus = tf->status;
154     memset(tf, 0, sizeof(struct trapframe));
155 
156     tf->gpr.sp = USTACKTOP;// tf->gpr.sp 指向用户栈栈顶
157     tf->epc = elf->e_entry;// tf->epc指向用户程序入口点
158     tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
159 
160     ret = 0;
161 out:
162     return ret;
163 bad_cleanup_mmap:
164     exit_mmap(mm);
165 bad_elf_cleanup_pgdir:
166     put_pgdir(mm);
167 bad_pgdir_cleanup_mm:
168     mm_destroy(mm);
169 bad_mm:
170     goto out;
171 }
```

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

UNIX的管道（Pipe）机制是一种进程间通信的方式，它允许**一个进程的输出直接成为另一个进程的输入**，从而实现它们之间的数据传输。管道可以用于在不同进程之间传递数据，使得这些进程能够协同工作。

在UNIX中，管道是由操作系统内核提供的一种特殊的文件类型。它可以连接两个进程，其中一个进程将数据写入管道，而另一个进程从管道中读取数据。数据在管道中按照先进先出（FIFO）的顺序进行传递，并遵循“阅后即焚”原则，在读取后直接被清除。

管道机制的主要特点包括：

- 管道是单向的（半双工通信方式），只能支持一个方向的数据传输。
- 通常只能在有亲缘关系的进程之间进行通信。
- 管道是基于字节流的，没有固定的消息边界。
- 管道的大小是有限的，当管道被填满时，写入操作会被阻塞，直到有空间可用。

要在ucore中加入UNIX的管道机制，至少需要定义以下数据结构和接口：

1. 管道**数据结构**（Pipe struct）：用于表示管道的状态和属性，包括管道的读写位置、缓冲区、大小等信息。
2. 创建管道接口（create_pipe()）：用于创建一个新的管道实例，并返回其文件描述符。
3. 写入数据接口（write_pipe()）：将数据写入管道中的写入端，如果管道已满，则阻塞等待直到有空间可用。
4. 读取数据接口（read_pipe()）：从管道中的读取端读取数据，如果管道为空，则阻塞等待直到有数据可读。
5. 关闭管道接口（close_pipe()）：关闭管道并释放相关资源。

在设计实现"UNIX的PIPE机制"时，需要考虑同步和互斥问题。增加互斥锁之后，管道数据结构的成员变量如下：

1. `buffer`：表示管道的缓冲区，用于存储数据的传输。可以使用一个字符数组或者其他合适的数据结构来实现。
2. `buffer_size`：表示管道缓冲区的大小，用于限制数据的容量。
3. `read_index`：表示管道读取位置的索引，用于指示下一个读取操作的位置。
4. `write_index`：表示管道写入位置的索引，用于指示下一个写入操作的位置。
5. `data_count`：表示当前管道中有效数据的数量，用于判断管道是否为空或已满。
6. `mutex`：**互斥锁，用于保护对管道的访问，确保同一时间只有一个进程可以访问管道**。
7. `read_cond`：读取条件变量，用于控制在管道为空时读取操作的等待。
8. `write_cond`：写入条件变量，用于控制在管道已满时写入操作的等待。

综上所述，结构体和相关函数的实现如下：

```
// 管道数据结构
typedef struct {
    char* buffer;
    int buffer_size;
    int read_index;
    int write_index;
    int data_count;
    pthread_mutex_t mutex;
    pthread_cond_t read_cond;
    pthread_cond_t write_cond;
} Pipe;

// 创建管道接口
int create_pipe(Pipe* pipe, int buffer_size) {
    // 初始化管道结构体成员
```

```
    pipe->buffer = malloc(buffer_size);
    pipe->buffer_size = buffer_size;
    pipe->read_index = 0;
    pipe->write_index = 0;
    pipe->data_count = 0;
    pthread_mutex_init(&(pipe->mutex), NULL);
    pthread_cond_init(&(pipe->read_cond), NULL);
    pthread_cond_init(&(pipe->write_cond), NULL);
    return 0; // 返回文件描述符或其他标识符
}

// 写入数据接口
void write_pipe(Pipe* pipe, char* data, int length) {
    pthread_mutex_lock(&(pipe->mutex));
    // 等待直到管道有足够的空间可用
    while (pipe->data_count == pipe->buffer_size) {
        pthread_cond_wait(&(pipe->write_cond), &(pipe->mutex));
    }
    // 写入数据到管道缓冲区
    for (int i = 0; i < length; i++) {
        pipe->buffer[pipe->write_index] = data[i];
        pipe->write_index = (pipe->write_index + 1) % pipe->buffer_size;
        pipe->data_count++;
    }
    // 通知等待读取的线程
    pthread_cond_signal(&(pipe->read_cond));
    pthread_mutex_unlock(&(pipe->mutex));
}

// 读取数据接口
void read_pipe(Pipe* pipe, char* buffer, int length) {
    pthread_mutex_lock(&(pipe->mutex));
    // 等待直到管道有数据可读
    while (pipe->data_count == 0) {
        pthread_cond_wait(&(pipe->read_cond), &(pipe->mutex));
    }
    // 从管道缓冲区读取数据
    for (int i = 0; i < length; i++) {
        buffer[i] = pipe->buffer[pipe->read_index];
        pipe->read_index = (pipe->read_index + 1) % pipe->buffer_size;
        pipe->data_count--;
    }
    // 通知等待写入的线程
    pthread_cond_signal(&(pipe->write_cond));
    pthread_mutex_unlock(&(pipe->mutex));
}

```

```
63 // 关闭管道接口
64 void close_pipe(Pipe* pipe) {
65     pthread_mutex_destroy(&(pipe->mutex));
66     pthread_cond_destroy(&(pipe->read_cond));
67     pthread_cond_destroy(&(pipe->write_cond));
68     free(pipe->buffer);
69 }
```

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

软连接是一种特殊的文件，它包含了指向另一个文件或目录的路径。软连接可以跨越文件系统边界，甚至可以链接到不存在的文件或目录。软连接**类似于Windows系统中的快捷方式**。

软连接的创建和删除不会影响原始文件或目录，它只是提供了一个指向原始文件或目录的符号链接。软连接使用原始文件或目录的路径名作为链接的内容，当访问软连接时，实际上是通过路径名找到原始文件或目录进行访问。其可以用于创建文件的备份、跨目录链接文件等场景。

软连接的实现：

- 新建一个 `sfs_disk_inode` 结构（即建立一个全新的文件）。之后，将 `old_path` （原始文件的路径）写入该文件中，并标注 `sfs_disk_inode` 的 `type` 为 `SFS_TYPE_LINK` 即可。
- 删除时，将对应的 `sfs_disk_entry` 和 `sfs_disk_inode` 结构删除。

硬连接是在文件系统中创建一个新的链接，它与原始文件或目录**共享**相同的索引节点（inode）。这意味着硬连接与原始文件或目录具有相同的文件名和文件内容，它们之间没有实质上的区别。

与软连接不同，硬连接只能在同一文件系统中创建，且不能链接到目录。硬连接的创建和删除**不应该**影响原始文件或目录，因为它们共享相同的inode。当删除任何一个硬连接时，其他硬连接仍然可以访问和使用原始文件或目录。

硬连接常用于创建多个文件名引用同一文件的情况，可以节省存储空间并提高文件的访问效率。

硬连接机制的实现：

- 创建硬连接时，仍然为 `new_path` 建立一个 `sfs_disk_entry` 结构，但该结构的内部 `ino` 成员指向 `old_path` 的磁盘索引结点，并使该磁盘索引节点的 `nlinks` 引用计数成员**加一**即可。
- 删除硬连接时，令对应磁盘结点 `sfs_disk_inode` 中的 `nlinks` 减一，同时删除硬连接的 `sfs_disk_entry` 结构即可。

需要注意的是，当最后一个硬连接被删除时，与该硬连接关过联的inode的计数器（也称为链接计数）会减少。如果该inode的链接计数器减少到零，表示没有任何硬链接指向该inode，这时该inode所占用的磁盘空间会被释放，并且文件系统会将其标记为可重用。**然而**，如果在硬连接被删除之前就已经打开了该文件或目录的句柄（文件描述符），那么即使最后一个硬连接被删除，该文件或目录仍然可以通过打开的句柄继续访问和使用，因为打开的句柄会保持对该inode的引用。

这种情况就导致了“悬空inode”（orphaned inode）现象的出现，即inode没有任何硬链接指向它，但仍然可以通过打开的句柄进行访问。

为了解决这个问题，文件系统通常会使用垃圾回收机制来定期扫描文件系统，检查是否存在悬空inode，并将其清理和回收。