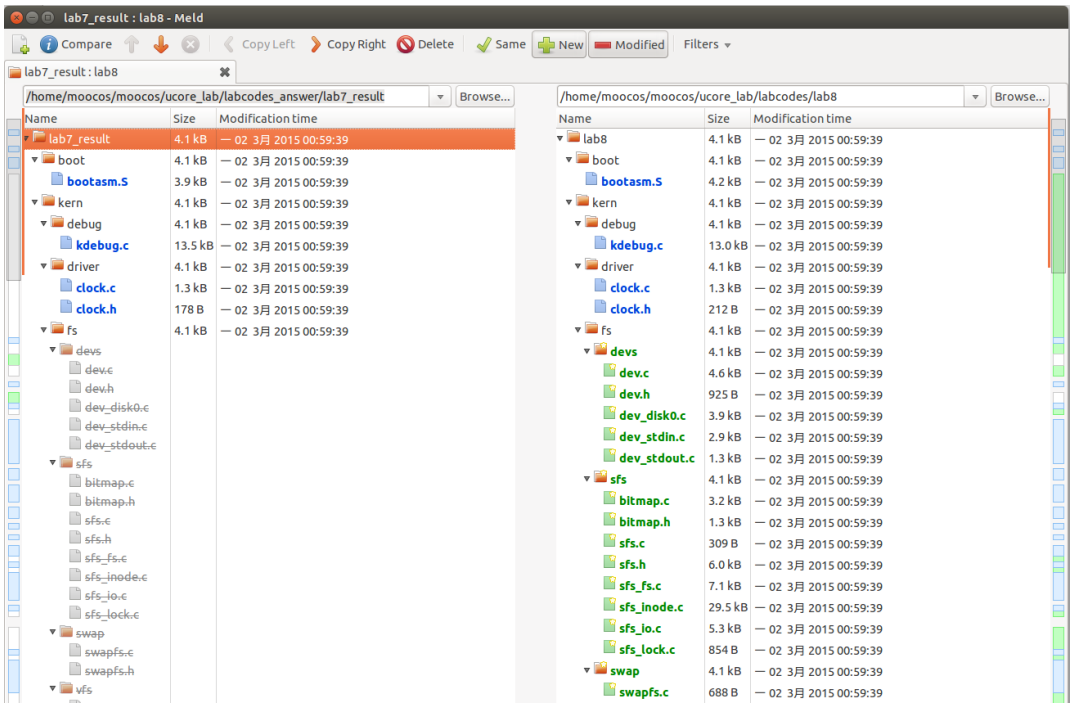
ucore lab8 文件系统实验报告

一、实验目的

1） 了解基本的文件系统系统调用的实现方法；
2） 了解一个基于索引节点组织方式的 Simple FS 文件系统的设计与实现；
3） 了解文件系统抽象层-VFS 的设计与实现；

二、实验内容

0）首先使用 meld 比较 lab7_result 和 lab8

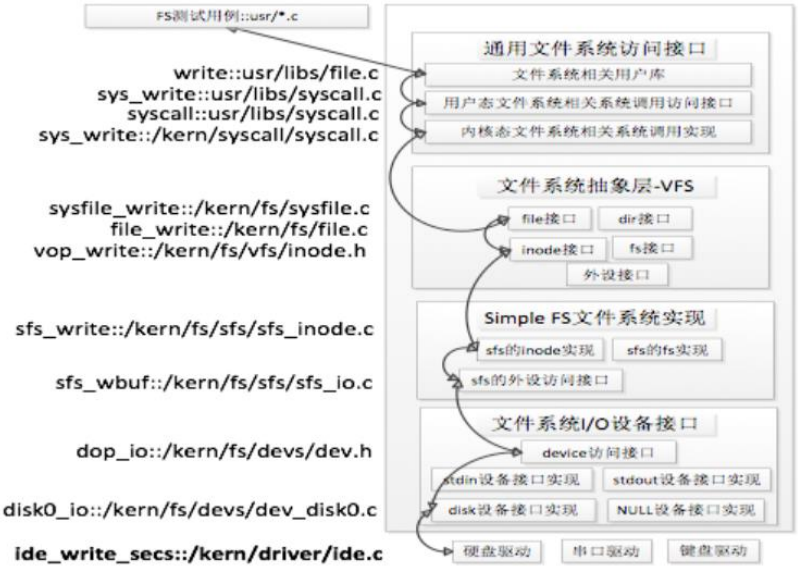


修改的文件：
kdebug.c, default_pmm.c, kmalloc.c, kmalloc.h, memlayout.h, pmm.c, swap_fifo.c, vmm.c, vmm.h, proc.c, default_sched.c, sched.c, check_sync.c, monitor.c, syscall.c, trap.c, atomic.h, grade.sh， systemcall[c,h]， spin.c
替换 Makefile

1）ucore 文件系统构成

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。



2）逐层分析及重要数据结构

① 通用文件系统访问接口

对文件的操作首先需要通过该接口从用户空间进入文件系统内部

文件操作和目录操作在打开和关闭上是相同的，因为在 ucore 中把目录看作是一个特殊的文件，所以 opendir 和 closedir 实际上就是调用与文件相关的 open 和 close 函数；而不同体现在 readdir 需要调用获取目录内容的特殊系统调用 sys_getdirentry

② 文件系统抽象层-VFS

文件系统抽象层是把不同文件系统的对外共性接口提取出来，形成一个**函数指针数组**，这样，通用文件系统访问接口层只需访问文件系统抽象层，而不需关心具体文件系统的实现细节和接口

- **file&dir 接口**定义了进程在内核中直接访问的文件相关信息

/kern/fs/file.h 中定义的 **file 数据结构：**

```
struct file {
    enum {
        FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
    } status;
    bool readable;
    bool writable;
    int fd;
    off_t pos;
    struct inode *node;
    int open_count;
};
```

- status：访问文件的执行状态
- readable：标记文件是否可读
- writable：标记文件是否可写
- fd：文件在 filemap 中的索引值
- pos：文件的当前位置
- node：文件对应的内存 inode 指针
- open_count：打开此文件的次数

- **inode 接口**(index node)是位于内存的索引节点，实际负责把不同文件系统的特定索引节点信息统一封装起来，避免进程直接访问具体文件系统

/kern/fs/vfs/inode.h 中定义 **inode 数据结构：**

```
29  struct inode {
30      union {
31          struct device __device_info;
32          struct sfs_inode __sfs_inode_info;
33      } in_info;
34      enum {
35          inode_type_device_info = 0x1234,
36          inode_type_sfs_inode_info,
37      } in_type;
38      int ref_count;
39      int open_count;
40      struct fs *in_fs;
41      const struct inode_ops *in_ops;
42  };
```

- in_info：__device_info 是设备文件系统内存 inode 信息，__sfs_inode_info 是 SFS 文件系统内存 inode 信息
- in_type：此 inode 所属文件系统类型
- ref_count：此 inode 引用次数
- open_count：打开此 inode 对应文件的个数
- in_fs：抽象的文件系统，包含访问文件系统的函数指针
- in_ops：抽象的 inode 操作，包含访问 inode 的函数指针（对此 inode 的操作函数指针列表，对于某一具体文件系统，只需实现相关函数即可访问）

```
169 struct inode_ops {
170     unsigned long vop_magic;
171     int (*vop_open)(struct inode *node, uint32_t open_flags);
172     int (*vop_close)(struct inode *node);
173     int (*vop_read)(struct inode *node, struct iobuf *iob);
174     int (*vop_write)(struct inode *node, struct iobuf *iob);
175     int (*vop_fstat)(struct inode *node, struct stat *stat);
176     int (*vop_fsync)(struct inode *node);
177     int (*vop_namefile)(struct inode *node, struct iobuf *iob);
178     int (*vop_getdirentry)(struct inode *node, struct iobuf *iob);
179     int (*vop_reclaim)(struct inode *node);
180     int (*vop_gettype)(struct inode *node, uint32_t *type_store);
181     int (*vop_tryseek)(struct inode *node, off_t pos);
182     int (*vop_truncate)(struct inode *node, off_t len);
183     int (*vop_create)(struct inode *node, const char *name, bool excl, struct inode **node_store);
184     int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
185     int (*vop_ioctl)(struct inode *node, int op, void *data);
186 };
```

③ Simple FS 文件系统层

文件系统通常保存在磁盘上，本实验中，第三个磁盘（即 disk0）用于存放一个 SFS 文件系统。通常文件系统中，磁盘的使用是以扇区（Sector）为单位的，但是为了实现简便，SFS 中以 block（4K，与内存 page 大小相等）为基本单位。

- **超级块（superblock）**

包含了关于文件系统的所有关键参数，当计算机被启动或文件系统被首次接触时，超级块的内容就会被装入内存

在/kern/fs/sfs/sfs.h 中 **sfs_super 结构定义如下**：

```
40  struct sfs_super {
41      uint32_t magic;                             /* magic number, should be SFS_MAGIC */
42      uint32_t blocks;                            /* # of blocks in fs */
43      uint32_t unused_blocks;                     /* # of unused blocks in fs */
44      char info[SFS_MAX_INFO_LEN + 1];            /* infomation for sfs  */
45  };
```

- magic：内核通过魔数来检查磁盘镜像是否合法
- blocks：记录 SFS 中所有 block 数量
- unused_block：记录还没有被使用 block 数量
- info：字符串"simple file system"

root-dir 的 inode，用来记录根目录的相关信息；
根据 SFS 中所有块的数量，用 1 个 bit 来表示一个块的占用和未被占用的情况。这个区域称为 SFS 的 **freemap 区域；**
最后在**剩余的磁盘空间**中，存放了所有其他目录和文件的 inode 信息和内容数据信息

- **索引节点**

在 SFS 文件系统中，需要记录文件内容的存储位置以及文件名与文件内容的对应关系
sfs_disk_inode 记录了文件或目录的内容存储的索引信息，该数据结构在硬盘里储存，需要时读入内存;
sfs_disk_entry 表示一个目录中的一个文件或目录，包含该项所对应 inode 的位置和文件名，同样也在硬盘里储存，需要时读入内存

inode 的目录操作函数和文件操作函数

```
998  // The sfs specific DIR operations correspond to the abstract operations on a inode.
999  static const struct inode_ops sfs_node_dirops = {
1000     .vop_magic                      = VOP_MAGIC,
1001     .vop_open                       = sfs_opendir,
1002     .vop_close                      = sfs_close,
1003     .vop_fstat                      = sfs_fstat,
1004     .vop_fsync                      = sfs_fsync,
1005     .vop_namefile                   = sfs_namefile,
1006     .vop_getdirentry                = sfs_getdirentry,
1007     .vop_reclaim                    = sfs_reclaim,
1008     .vop_gettype                    = sfs_gettype,
1009     .vop_lookup                     = sfs_lookup,
1010 };
1011 /// The sfs specific FILE operations correspond to the abstract operations on a inode.
1012 static const struct inode_ops sfs_node_fileops = {
1013     .vop_magic                      = VOP_MAGIC,
1014     .vop_open                       = sfs_openfile,
1015     .vop_close                      = sfs_close,
1016     .vop_read                       = sfs_read,
1017     .vop_write                      = sfs_write,
1018     .vop_fstat                      = sfs_fstat,
1019     .vop_fsync                      = sfs_fsync,
1020     .vop_reclaim                    = sfs_reclaim,
1021     .vop_gettype                    = sfs_gettype,
1022     .vop_tryseek                    = sfs_tryseek,
1023     .vop_truncate                   = sfs_truncfile,
1024 };
```

3）实验执行流程

lab8 增加了加载可执行文件到内存运行的功能，导致对进程管理相关实现的调整

总控函数 kern_init()中增加了 **fs_init()函数**的调用

```
22  int
23  kern_init(void) {
24      extern char edata[], end[];
25      memset(edata, 0, end - edata);
26
27      cons_init();                // init the console
28
29      const char *message = "(THU.CST) os is loading ...";
30      cprintf("%s\n\n", message);
31
32      print_kerninfo();
33
34      grade_backtrace();
35
36      pmm_init();                 // init physical memory management
37
38      pic_init();                 // init interrupt controller
39      idt_init();                 // init interrupt descriptor table
40
41      vmm_init();                 // init virtual memory management
42      sched_init();               // init scheduler
43      proc_init();                // init process table
44
45      ide_init();                 // init ide devices
46      swap_init();                // init swap
47      fs_init();                  // init fs
```

fs_init()函数是文件系统初始化的总控函数，进一步调用了虚拟文件系统初始化函数，设备初始化函数和 SFS 文件系统初始化函数

```
10  //called when init_main proc start
11  void
12  fs_init(void) {
13      vfs_init();
14      dev_init();
15      sfs_init();
16  }
```

vfs_init 主要建立了 一个 devicelist 双向链表 vdev_list，为后续具体设备（键盘、串口、磁盘）以文件的形式呈现建立查找访问通道。

dev_init 通过进一步调用 disk0/stdin/stdout_device_init 完成对具体设 备的初始化，把它们抽象成一个设备文件，并建立对应的 inode 数据结构，最后把它们链入到 vdev_list 中。这样通过虚拟文件系统就可以方便地以文件的形式访问这些设备了。

sfs_init 是完成对 Simple FS 的初始化工作，并把此实例文件系统挂在虚拟文件系统中，从而让 ucore 的其他部分能够通过访问虚拟文件系统的接口来进一步访问到 SFS 实例文件系统。

4）文件操作实现

首先学习打开文件流程（没有找到讲义中说的 sfs_filetest1.c 文件，就以 user/sh.c 为例）

用户进程会调用如下语句：

```
109     if ((ret = open(name, O_RDONLY)) < 0) {
```

如果 ucore 能正常找到这个 name 文件，就会返回一个代表文件的文件描述符

具体调用过程：

① 通用文件访问接口层

首先用户会在进程中调用 open()函数，然后依次调用如下函数：open->sys_open->syscall，从而引起系统调用进入到内核态。到了内核态后，通过中断处理例程，会调用到 sys_open 内核函数，并进一步调用 sysfile_open 内核函数。到了这里，需要把位于用户空间的字符串”/test/testfile”拷贝到内核空间中的字符串 path 中，并进入到文件系统抽象层的处理流程完成进一步的打开文件操作中

open()函数调用 sys_open()

```
int
open(const char *path, uint32_t open_flags) {
    return sys_open(path, open_flags);
}
```

sys_open()函数调用 syscall()

```
int
sys_open(const char *path, uint32_t open_flags) {
    return syscall(SYS_open, path, open_flags);
}
```

syscall()参数为 SYS_open

```
static inline int
syscall(int num, ...) {
    va_list ap;
    va_start(ap, num);
    uint32_t a[MAX_ARGS];
    int i, ret;
    for (i = 0; i < MAX_ARGS; i ++) {
        a[i] = va_arg(ap, uint32_t);
    }
    va_end(ap);

    asm volatile (
        "int %1;"
```

SYS_open 实际对应 sys_open()函数

```
static int (*syscalls[])(uint32_t arg[]) = {
    [SYS_exit]              sys_exit,
    [SYS_fork]              sys_fork,
    [SYS_wait]              sys_wait,
    [SYS_exec]              sys_exec,
    [SYS_yield]             sys_yield,
    [SYS_kill]              sys_kill,
    [SYS_getpid]            sys_getpid,
    [SYS_putc]              sys_putc,
    [SYS_pgdir]             sys_pgdir,
    [SYS_gettime]           sys_gettime,
    [SYS_lab6_set_priority] sys_lab6_set_priority,
    [SYS_sleep]             sys_sleep,
    [SYS_open]              sys_open,
    [SYS_close]             sys_close
```

sys_open()函数调用 sysfile_open()

```
static int
sys_open(uint32_t arg[]) {
    const char *path = (const char *)arg[0];
    uint32_t open_flags = (uint32_t)arg[1];
    return sysfile_open(path, open_flags);
}
```

sysfile_open()内核函数

```
int
sysfile_open(const char *__path, uint32_t open_flags) {
    int ret;
    char *path;
    if ((ret = copy_path(&path, __path)) != 0) {
        return ret;
    }
    ret = file_open(path, open_flags);
    kfree(path);
    return ret;
}
```

把位于用户空间的字符串拷贝到内核空间中的字符串 path 中，并进入到文件系统抽象层的处理流程完成进一步的打开文件操作

② 文件系统抽象层

系统会分配一个 file 数据结构的变量，但是分配完了之后还不能找到对应的文件结点。所以系统在该层调用了 vfs_open 函数通过调用 vfs_lookup 找到 path 对应文件的 inode，然后调用 vop_open 函数打开文件。然后层层返回，通过执行语句 file->node=node;，就把当前进

程的 current->fs_struct->filemap[fd]（即 file 所指变量）的成员变量 node 指针指向了代表文件的索引节点 node。这时返回 fd。最后完成打开文件的操作。

file_open()调用 vfs_open()

```
156 int
157 file_open(char *path, uint32_t open_flags) {
158     bool readable = 0, writable = 0;
159     switch (open_flags & O_ACCMODE) {
160     case O_RDONLY: readable = 1; break;
161     case O_WRONLY: writable = 1; break;
162     case O_RDWR:
163         readable = writable = 1;
164         break;
165     default:
166         return -E_INVAL;
167     }
168
169     int ret;
170     struct file *file;
171     if ((ret = fd_array_alloc(NO_FD, &file)) != 0) {
172         return ret;
173     }
174
175     struct inode *node;
176     if ((ret = vfs_open(path, open_flags, &node)) != 0) {
```

vfs_open()调用 vfs_lookup()找到 path 对应文件的 inode

```
11 int
12 vfs_open(char *path, uint32_t open_flags, struct inode **node_store) {
13     bool can_write = 0;
14     switch (open_flags & O_ACCMODE) {
15     case O_RDONLY:
16         break;
17     case O_WRONLY:
18     case O_RDWR:
19         can_write = 1;
20         break;
21     default:
22         return -E_INVAL;
23     }
24
25     if (open_flags & O_TRUNC) {
26         if (!can_write) {
27             return -E_INVAL;
28         }
29     }
30
31     int ret;
32     struct inode *node;
33     bool excl = (open_flags & O_EXCL) != 0;
34     bool create = (open_flags & O_CREAT) != 0;
35     ret = vfs_lookup(path, &node);
```

③ SFS 文件系统层

在第 2 步中，调用了 SFS 文件系统层的 vfs_lookup 函数去寻找 node

vfs_lookup()调用 vop_lookup()

```
71 int
72 vfs_lookup(char *path, struct inode **node_store) {
73     int ret;
74     struct inode *node;
75     if ((ret = get_device(path, &path, &node)) != 0) {
76         return ret;
77     }
78     if (*path != '\0') {
79         ret = vop_lookup(node, path, node_store);
80         vop_ref_dec(node);
81         return ret;
82     }
83     *node_store = node;
84     return 0;
85 }
```

可以看到.vop_lookup = sfs_lookup

```
184     int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
```

```
1009     .vop_lookup                     = sfs_lookup,
```

sfs_lookup()函数

```
977  static int
978  sfs_lookup(struct inode *node, char *path, struct inode **node_store) {
979      struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
980      assert(*path != '\0' && *path != '/');
981      vop_ref_inc(node);
982      struct sfs_inode *sin = vop_info(node, sfs_inode);
983      if (sin->din->type != SFS_TYPE_DIR) {
984          vop_ref_dec(node);
985          return -E_NOTDIR;
986      }
987      struct inode *subnode;
988      int ret = sfs_lookup_once(sfs, sin, path, &subnode, NULL);
989
990      vop_ref_dec(node);
991      if (ret != 0) {
992          return ret;
993      }
994      *node_store = subnode;
995      return 0;
996  }
```

三个参数中 node 是根目录“/”所对应的 inode 节点；path 是文件的绝对路径（例如“/test/file”），而 node_store 是经过查找获得的 file 所对应的 inode 节点。
函数以“/”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。在本例中是分解出“test”子目录，并调用 sfs_lookup_once 函数获得“test”子目录对应的 inode 节点 subnode，然后循环进一步调用 sfs_lookup_once 查找以“test”子目录下的文件“testfile1”所对应的 inode 节点。当无法分解 path 后，就意味着找到了 testfile1 对应的 inode 节点，就可顺利返回了

sfs_lookup_once()函数调用 **sfs_direct_search_nolock()函数**来查找与路径名匹配的目录项，如果找到目录项，则根据目录项中记录的 inode 所处的数据块索引值找到路径名对应的 SFS 磁盘 inode，并读入 SFS 磁盘 inode 对的内容，创建 SFS 内存 inode。

```
498  static int
499  sfs_lookup_once(struct sfs_fs *sfs, struct sfs_inode *sin, const char *name, struct inode **node_store, int *slot) {
500      int ret;
501      uint32_t ino;
502      lock_sin(sin);
503      {   // find the NO. of disk block and logical index of file entry
504          ret = sfs_dirent_search_nolock(sfs, sin, name, &ino, slot, NULL);
```

```
440  static int
441  sfs_dirent_search_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, const char *name, uint32_t *ino_store,
442      assert(strlen(name) <= SFS_MAX_FNAME_LEN);
443      struct sfs_disk_entry *entry;
444      if ((entry = kmalloc(sizeof(struct sfs_disk_entry))) == NULL) {
445          return -E_NO_MEM;
```

【TODO 1】完成读文件操作
类似上面打开文件的过程，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

步骤：① 首先需要通过文件系统的**通用文件系统访问接口层**给用户空间提供的访问接口进入文件系统内部；
② 接着由**文件系统抽象层**把访问请求转发给某一具体文件系统（如 SFS 文件系统）；
③ **具体文件系统（Simple FS 文件系统层）**把应用程序的访问请求转化为对磁盘上的 block 的处理请求；
④ 并通过**外设接口层**交给磁盘驱动例程来完成具体的磁盘操作

下面结合调用了 read()函数的/user/sh.c 文件分析读文件处理流程:

```
Getting Started | sh.c | file.c | syscall.c | syscall.c | sysfile.c | file.c | inode.h | sfs_inode.c
46      return token;
47  }
48
49  char *
50  readline(const char *prompt) {
51      static char buffer[BUFSIZE];
52      if (prompt != NULL) {
53          printf("%s", prompt);
54      }
55      int ret, i = 0;
56      while (1) {
57          char c;
58          if ((ret = read(0, &c, sizeof(char))) < 0) {
```

①通用文件访问接口层的处理流程

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

在/user/libs/file.c 中实现的 **read()函数**调用了 sys_read()函数

```
Getting Started | sh.c | file.c | syscall.c | syscall.c | sysfile.c | file.c | inode.h | sfs_inode.c
19  int
20  read(int fd, void *base, size_t len) {
21      return sys_read(fd, base, len);
22  }
```

在/user/libs/syscall.c 中实现了 **sys_read()函数**

```
Getting Started | sh.c | file.c | syscall.c | syscall.c | sysfile.c | file.c | inode.h | sfs_inode.c
107 int
108 sys_read(int fd, void *base, size_t len) {
109     return syscall(SYS_read, fd, base, len);
110 }
```

syscall()函数是对系统调用的封装

```
Getting Started | sh.c | file.c | syscall.c | syscall.c | sysfile.c | file.c | inode.h | sfs_inode.c
10
11  static inline int
12  syscall(int num, ...) {
13      va_list ap;
14      va_start(ap, num);
15      uint32_t a[MAX_ARGS];
16      int i, ret;
17      for (i = 0; i < MAX_ARGS; i ++) {
18          a[i] = va_arg(ap, uint32_t);
19      }
20      va_end(ap);
21
22      asm volatile (
23          "int %1;"
24          : "=a" (ret)
25          : "i" (T_SYSCALL),
26            "a" (num),
27            "d" (a[0]),
28            "c" (a[1]),
29            "b" (a[2]),
30            "D" (a[3]),
31            "S" (a[4])
32          : "cc", "memory");
33      return ret;
34  }
```

参数 **SYS_read** 对应的系统调用号为 102

```
6   /* syscall number */
7   #define SYS_exit            1
8   #define SYS_fork            2
9   #define SYS_wait            3
10  #define SYS_exec            4
11  #define SYS_clone           5
12  #define SYS_yield           10
13  #define SYS_sleep           11
14  #define SYS_kill            12
15  #define SYS_gettime         17
16  #define SYS_getpid          18
17  #define SYS_mmap            20
18  #define SYS_munmap          21
19  #define SYS_shmem           22
20  #define SYS_putc            30
21  #define SYS_pgdir           31
22  #define SYS_open            100
23  #define SYS_close           101
24  #define SYS_read            102
25  #define SYS_write           103
```

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**，

```
[SYS_close]             sys_close,
[SYS_read]              sys_read,
[SYS_write]             sys_write,
```

sys_read()内核函数进一步调用 sysfile_read 内核函数，进入到文件系统抽象层处理流程

```
static int
sys_read(uint32_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);
}
```

② **文件系统抽象层处理流程**

i) 检检查错误，即检查读取长度是否为 0 和文件是否可读

Getting Started | sh.c | file.c | syscall.c | syscall.c | sysfile.c | file.c | inode.h | sfs_inode.c | unistd.h

```
sysfile_read(int fd, void *base, size_t len) {
    struct mm_struct *mm = current->mm;
    if (len == 0) {
        return 0;
    }
    if (!file_testfd(fd, 1, 0)) {
        return -E_INVAL;
    }
```

ii) 分配 buffer 空间，即调用 kmalloc 函数分配 4096 字节的 buffer 空间

```
    void *buffer;
    if ((buffer = kmalloc(IOBUF_SIZE)) == NULL) {
        return -E_NO_MEM;
    }
```

iii) 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。

```
    int ret = 0;
    size_t copied = 0, alen;
    while (len != 0) {
        if ((alen = IOBUF_SIZE) > len) {//IOBUF_SIZE=4096,len为剩余部分长度
            alen = len;//如果剩余部分长度<4096,只读取剩余部分
        }
        ret = file_read(fd, buffer, alen, &alen);//将文件内容读取到buffer中
        if (alen != 0) {
            lock_mm(mm);
            {
                if (copy_to_user(mm, base, buffer, alen)) {//将读取到的内容拷贝到用户的内存空间中
                    assert(len >= alen);
                    base += alen, len -= alen, copied += alen;
                }
                else if (ret == 0) {
                    ret = -E_INVAL;
                }
            }
            unlock_mm(mm);
        }
        if (ret != 0 || alen == 0) {
            goto out;
        }
    }
```

```
out:
    kfree(buffer);
    if (copied != 0) {
        return copied;
    }
    return ret;
}
```

[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。

```
211 // read file
212 int
213 file_read(int fd, void *base, size_t len, size_t *copied_store) {
214     int ret;
215     struct file *file;
216     *copied_store = 0;
217     if ((ret = fd2file(fd, &file)) != 0) {//找到对应的file结构
218         return ret;
219     }
220     if (!file->readable) {//检查是否可读
221         return -E_INVAL;
222     }
223     fd_array_acquire(file);//打开文件数+1
224
225     struct iobuf __iob, *iob = iobuf_init(&__iob, base, len, file->pos);
226     ret = vop_read(file->node, iob);//将文件内容读入iob中
227
228     size_t copied = iobuf_used(iob);//实际读到的字节数
229     if (file->status == FD_OPENED) {
230         file->pos += copied;//文件指针后移
231     }
232     *copied_store = copied;
233     fd_array_release(file);//打开该文件计数-1
234     return ret;
235 }
```

③ 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，**

```
973 /// The sfs specific FILE operations correspond to the abstract operations on a inode.
974 static const struct inode_ops sfs_node_fileops = {
975     .vop_magic                      = VOP_MAGIC,
976     .vop_open                       = sfs_openfile,
977     .vop_close                      = sfs_close,
978     .vop_read                       = sfs_read,
979     .vop_write                      = sfs_write,
```

所以下面来分析 sfs_read 函数的实现：

```
632 // sfs_read - read file
633 static int
634 sfs_read(struct inode *node, struct iobuf *iob) {
635     return sfs_io(node, iob, 0);
636 }
```

sfs_read 函数调用 **sfs_io 函数**

```
615 static inline int
616 sfs_io(struct inode *node, struct iobuf *iob, bool write) {
617     struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
618     struct sfs_inode *sin = vop_info(node, sfs_inode);
619     int ret;
620     lock_sin(sin);
621     {
622         size_t alen = iob->io_resid;
623         ret = sfs_io_nolock(sfs, sin, iob->io_base, iob->io_offset, &alen, write);
624         if (alen != 0) {
625             iobuf_skip(iob, alen);
626         }
627     }
628     unlock_sin(sin);
629     return ret;
630 }
```

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_nolock 函数主要用来将磁盘中的一段数据读入到内存中或者将内存中的一段数据写入磁盘。需要补充的代码用来对一段地址对应磁盘块的读或写

```
/*
 * sfs_io_nolock - Rd/Wr a file contentfrom offset position to offset+ length  disk blocks<-->buffer (in memroy)
 * @sfs:      sfs file system
 * @sin:      sfs inode in memory
 * @buf:      the buffer Rd/Wr
 * @offset:   the offset of file
 * @alenp:    the length need to read (is a pointer). and will RETURN the really Rd/Wr lenght
 * @write:    BOOL, 0 read, 1 write
 */
```

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

```
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
```

（blkno 是读写开始的块，nblks 是读写块的数量）

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。每部分中都调用 **sfs_bmap_load_nolock 函数**得到 block 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

```
//LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf, sfs_rblock,etc. read different ki
  /*
   * (1) If offset isn't aligned with the first block, Rd/Wr some content from offset to the end of the
   *       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 (endpo
   *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
  */
```

```
603     //读取第一页数据
604     if ((blkoff = offset % SFS_BLKSIZE) != 0) {
605         size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);//计算第一个数据块大小
606         if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {//找到内存文件索引对应的block编号
607             goto out;
608         }
609         if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
610             goto out;
611         }
612         //完成实际读写操作
613         alen += size;
614         if (nblks == 0) {
615             goto out;
616         }
617         buf += size;
618         blkno ++;
619         nblks --;
620     }

621     //如果超过一页，读取第2~n-1页数据，将其分成大小为size的块，一次读一块直至读完
622         size = SFS_BLKSIZE;
623         while(nblks!=0){
624             if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
625                 goto out;
626             }
627             if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
628                 goto out;
629             }
630             alen += size;
631             buf+=size;
632             blkno++;
633             nblks--;
634         }
635     //读取最后一页数据
636     if ((size = endpos % SFS_BLKSIZE) != 0) {
637         if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
638             goto out;
639         }
640         if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
641             goto out;
642         }
643         alen += size;
644     }
645 out:
646     *alenp = alen;
647     if (offset + alen > sin->din->size) {
648         sin->din->size = offset + alen;
649         sin->dirty = 1;
650     }
651     return ret;
652 }
```

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

（下面是用 understand 观察的调用过程）

```
353 static int
354 sfs_bmap_load_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, uint32_t index, uint32_t *ino_store) {
355     struct sfs_disk_inode *din = sin->din;
356     assert(index <= din->blocks);
357     int ret;
358     uint32_t ino;
359     bool create = (index == din->blocks);
360     if ((ret = sfs_bmap_get_nolock(sfs, sin, index, create, &ino)) != 0) {
361         return ret;
362     }
363     assert(sfs_block_inuse(sfs, ino));
364     if (create) {
365         din->blocks ++;
366     }
367     if (ino_store != NULL) {
368         *ino_store = ino;
369     }
370     return 0;
371 }
```

sfs_bmap_get_nolock

```
256 static int
257 sfs_bmap_get_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, uint32_t index, bool create, uint32_t *ino_store) {
258     struct sfs_disk_inode *din = sin->din;
259     int ret;
260     uint32_t ent, ino;
261     // the index of disk block is in the fist SFS_NDIRECT  direct blocks
262     if (index < SFS_NDIRECT) {
263         if ((ino = din->direct[index]) == 0 && create) {
264             if ((ret = sfs_block_alloc(sfs, &ino)) != 0) {
265                 return ret;
266             }
267             din->direct[index] = ino;
268             sin->dirty = 1;
269         }
270         goto out;
271     }
272     // the index of disk block is in the indirect blocks.
```

sfs_rbuf

```
83  int
84  sfs_rbuf(struct sfs_fs *sfs, void *buf, size_t len, uint32_t blkno, off_t offset) {
85      assert(offset >= 0 && offset < SFS_BLKSIZE && offset + len <= SFS_BLKSIZE);
86      int ret;
87      lock_sfs_io(sfs);
88      {
89          if ((ret = sfs_rwblock_nolock(sfs, sfs->sfs_buffer, blkno, 0, 1)) == 0) {
90              memcpy(buf, sfs->sfs_buffer + offset, len);
91          }
92      }
93      unlock_sfs_io(sfs);
94      return ret;
95  }
```

sfs_rwblock_nolock

```
19  static int
20  sfs_rwblock_nolock(struct sfs_fs *sfs, void *buf, uint32_t blkno, bool write, bool check) {
21      assert((blkno != 0 || !check) && blkno < sfs->super.blocks);
22      struct iobuf __iob, *iob = iobuf_init(&__iob, buf, SFS_BLKSIZE, blkno * SFS_BLKSIZE);
23      return dop_io(sfs->dev, iob, write);
24  }
```

dop_io

```
13  struct device {
14      size_t d_blocks;
15      size_t d_blocksize;
16      int (*d_open)(struct device *dev, uint32_t open_flags);
17      int (*d_close)(struct device *dev);
18      int (*d_io)(struct device *dev, struct iobuf *iob, bool write);
19      int (*d_ioctl)(struct device *dev, int op, void *data);
20  };
21
22  #define dop_open(dev, open_flags)           ((dev)->d_open(dev, open_flags))
23  #define dop_close(dev)                      ((dev)->d_close(dev))
24  #define dop_io(dev, iob, write)             ((dev)->d_io(dev, iob, write))
25  #define dop_ioctl(dev, op, data)            ((dev)->d_ioctl(dev, op, data))
```

disk0_io

```
111 static void
112 disk0_device_init(struct device *dev) {
113     static_assert(DISK0_BLKSIZE % SECTSIZE == 0);
114     if (!ide_device_valid(DISK0_DEV_NO)) {
115         panic("disk0 device isn't available.\n");
116     }
117     dev->d_blocks = ide_device_size(DISK0_DEV_NO) / DISK0_BLK_NSECT;
118     dev->d_blocksize = DISK0_BLKSIZE;
119     dev->d_open = disk0_open;
120     dev->d_close = disk0_close;
121     dev->d_io = disk0_io;
122     dev->d_ioctl = disk0_ioctl;
123     sem_init(&(disk0_sem), 1);
124
125     static_assert(DISK0_BUFSIZE % DISK0_BLKSIZE == 0);
126     if ((disk0_buffer = kmalloc(DISK0_BUFSIZE)) == NULL) {
127         panic("disk0 alloc buffer failed.\n");
128     }
129 }
```

```
96              disk0_read_blks_nolock(blkno, nblks);
```

disk0_read_blks_nolock

```
40  static void
41  disk0_read_blks_nolock(uint32_t blkno, uint32_t nblks) {
42      int ret;
43      uint32_t sectno = blkno * DISK0_BLK_NSECT, nsecs = nblks * DISK0_BLK_NSECT;
44      if ((ret = ide_read_secs(DISK0_DEV_NO, sectno, disk0_buffer, nsecs)) != 0) {
45          panic("disk0: read blkno = %d (sectno = %d), nblks = %d (nsecs = %d): 0x%08x.\n",
46                  blkno, sectno, nblks, nsecs, ret);
47      }
48  }
```

ide_read_secs

```
157 int
158 ide_read_secs(unsigned short ideno, uint32_t secno, void *dst, size_t nsecs) {
159     assert(nsecs <= MAX_NSECS && VALID_IDE(ideno));
160     assert(secno < MAX_DISK_NSECS && secno + nsecs <= MAX_DISK_NSECS);
161     unsigned short iobase = IO_BASE(ideno), ioctrl = IO_CTRL(ideno);
162
163     ide_wait_ready(iobase, 0);
164
165     // generate interrupt
166     outb(ioctrl + ISA_CTRL, 0);
167     outb(iobase + ISA_SECCNT, nsecs);
168     outb(iobase + ISA_SECTOR, secno & 0xFF);
169     outb(iobase + ISA_CYL_LO, (secno >> 8) & 0xFF);
170     outb(iobase + ISA_CYL_HI, (secno >> 16) & 0xFF);
171     outb(iobase + ISA_SDH, 0xE0 | ((ideno & 1) << 4) | ((secno >> 24) & 0xF));
```

[TODO2] 完成基于文件系统的执行程序机制

proc_struct 进程控制块结构中多出一个成员变量 filesp，是一个文件指针

```
70    struct files_struct *filesp;                // the file related info(pwd, files_count, files_array, fs_semaphore) of process
```

在 proc.c 中的 alloc_proc()函数中初始化为空

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

load_icode()函数主要是将文件加载到内存中执行，从上面的注释可知分为了一共七个步骤：

1、建立内存管理器

2、建立页目录

3、将文件逐个段加载到内存中，这里要注意设置虚拟地址与物理地址之间的映射

4、建立相应的虚拟内存映射表

5、建立并初始化用户堆栈

6、处理用户栈中传入的参数

7、最后很关键的一步是设置用户进程的中断帧

8、发生错误还需要进行错误处理

```
601 static int
602 load_icode(int fd, int argc, char **kargv) {
603     /* LAB8:EXERCISE2 YOUR CODE  HINT:how to load the file with handler fd  in to process's memory
604      * MACROs or Functions:
605      *  mm_create        - create a mm
606      *  setup_pgdir      - setup pgdir in mm
607      *  load_icode_read  - read raw data content of program file
608      *  mm_map           - build new vma
609      *  pgdir_alloc_page - allocate new memory for  TEXT/DATA/BSS/stack parts
610      *  lcr3             - update Page Directory Addr Register -- CR3
611      */
612     /* (1) create a new mm for current process
613      * (2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
614      * (3) copy TEXT/DATA/BSS parts in binary to memory space of process
615      *    (3.1) read raw data content in file and resolve elfhdr
616      *    (3.2) read raw data content in file and resolve proghdr based on info in elfhdr
617      *    (3.3) call mm_map to build vma related to TEXT/DATA
618      *    (3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read contents in file
619      *          and copy them into the new allocated pages
620      *    (3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero in these pages
621      * (4) call mm_map to setup user stack, and put parameters into user stack
622      * (5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)
623      * (6) setup uargc and uargv in user stacks
624      * (7) setup trapframe for user environment
625      * (8) if up steps failed, you should cleanup the env.
626      */
```

```
627     assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
628
629     //(1) 当前内存管理器为空时，新建内存管理器
630     if (current->mm != NULL) {
631         panic("load_icode: current->mm must be empty.\n");
632     }
633
634     int ret = -E_NO_MEM;//E_NO_MEM代表因为存储设备产生的请求错误
635     struct mm_struct *mm;
636     if ((mm = mm_create()) == NULL) {
637         goto bad_mm;
638     }
639     //(2) 建立页目录
640     if (setup_pgdir(mm) != 0) {
641         goto bad_pgdir_cleanup_mm;
642     }
643     //(3) 从文件加载程序到内存
644     struct Page *page;
645
646     struct elfhdr __elf, *elf = &__elf;
647     if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
648         goto bad_elf_cleanup_pgdir;
649     }
650
651     if (elf->e_magic != ELF_MAGIC) {
652         ret = -E_INVAL_ELF;
653         goto bad_elf_cleanup_pgdir;
654     }
655
656     struct proghdr __ph, *ph = &__ph;
657     uint32_t vm_flags, perm, phnum;
658     for (phnum = 0; phnum < elf->e_phnum; phnum ++) {//e_phnum代表程序段入口地址数目
659         off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;//循环读取程序每个段的头部
660         if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0) {
661             goto bad_cleanup_mmap;
662         }
663         if (ph->p_type != ELF_PT_LOAD) {
664             continue ;
665         }
666         if (ph->p_filesz > ph->p_memsz) {
667             ret = -E_INVAL_ELF;
668             goto bad_cleanup_mmap;
669         }
670         if (ph->p_filesz == 0) {
671             continue ;
672         }
673         vm_flags = 0, perm = PTE_U;//建立虚拟地址和物理地址之间的映射
674         if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
675         if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
676         if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
677         if (vm_flags & VM_WRITE) perm |= PTE_W;
678         if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
679             goto bad_cleanup_mmap;
680         }
681         off_t offset = ph->p_offset;
682         size_t off, size;
683         uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);
684
685         ret = -E_NO_MEM;
686         //复制数据段和代码段
687         end = ph->p_va + ph->p_filesz;//数据段和代码段终止地址
688         while (start < end) {
689             if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
690                 ret = -E_NO_MEM;
691                 goto bad_cleanup_mmap;
692             }
693             off = start - la, size = PGSIZE - off, la += PGSIZE;
694             if (end < la) {
695                 size -= la - end;
696             }
697             //每次读取size大小的块，直至全部读完
698             if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) != 0) {//load_icode_read读取文件
699                 goto bad_cleanup_mmap;
700             }
```

```
            start += size, offset += size;
        }
        //建立bss段
        end = ph->p_va + ph->p_memsz;//终止地址

        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);//每次操作size大小的块
            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);//关闭文件，加载程序结束


    //(4) 建立相应的虚拟内存映射表
    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);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);

    //(5)设置用户栈
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    //(6)处理用户栈中传入的参数（argc为参数个数，argv为参数内容的地址）
    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 ++) {//将参数取出
        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);//栈顶
    *(int *)stacktop = argc;

```

```
768     //(7)设置进程的中断帧
769     struct trapframe *tf = current->tf;
770     memset(tf, 0, sizeof(struct trapframe));
771     tf->tf_cs = USER_CS;
772     tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
773     tf->tf_esp = stacktop;
774     tf->tf_eip = elf->e_entry;
775     tf->tf_eflags = FL_IF;
776     ret = 0;
777     //(8)错误处理
778 out:
779     return ret;
780 bad_cleanup_mmap:
781     exit_mmap(mm);
782 bad_elf_cleanup_pgdir:
783     put_pgdir(mm);
784 bad_pgdir_cleanup_mm:
785     mm_destroy(mm);
786 bad_mm:
787     goto out;
788 }
```

第一次 make qemu 之后输入 ls 和 hello 说不存在指令，发现这个 **do_fork()函数**，需要将父进程的文件系统信息复制到子进程中去

```
443 int
444 do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
445     int ret = -E_NO_FREE_PROC;
446     struct proc_struct *proc;
447     if (nr_process >= MAX_PROCESS) {
448         goto fork_out;
449     }
450     ret = -E_NO_MEM;
451     //LAB4:EXERCISE2 YOUR CODE
452     //LAB8:EXERCISE2 YOUR CODE  HINT:how to copy the fs in parent's proc_struct?
```

```
495     if (copy_files(clone_flags, proc) != 0) { //for LAB8
496         goto bad_fork_cleanup_fs;
497     }
```

```
519 bad_fork_cleanup_fs:  //for LAB8
520     put_files(proc);
```

再次 make qemu 成功

```
No.0 philosopher_condvar quit
No.2 philosopher_condvar quit
phi_test_condvar: state_condvar[1] will eating
phi_test_condvar: signal self_cv[1]
cond_signal begin: cvp c03559f4, cvp->count 0, cvp->owner->next_count 0
cond_signal end: cvp c03559f4, cvp->count 0, cvp->owner->next_count 0
Iter 4, No.1 philosopher_condvar is eating
Iter 4, No.2 philosopher_sema is thinking
No.1 philosopher_condvar quit
No.4 philosopher_condvar quit
Iter 4, No.2 philosopher_sema is eating
No.2 philosopher_sema quit
```

输入 ls，打印所有用户程序

```
@ 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)    40315(s)   softint
  [-]   1(h)       11(b)    44503(s)   priority
  [-]   1(h)       11(b)    44516(s)   matrix
  [-]   1(h)       10(b)    40323(s)   faultreadkernel
  [-]   1(h)       10(b)    40313(s)   hello
  [-]   1(h)       10(b)    40314(s)   badarg
  [-]   1(h)       10(b)    40336(s)   sleep
```

```
   [-]   1(h)       11(b)    44626(s)   sh
   [-]   1(h)       10(b)    40312(s)   spin
   [-]   1(h)       11(b)    44572(s)   ls
   [-]   1(h)       10(b)    40318(s)   badsegment
   [-]   1(h)       10(b)    40367(s)   forktree
   [-]   1(h)       10(b)    40342(s)   forktest
   [-]   1(h)       10(b)    40448(s)   waitkill
   [-]   1(h)       10(b)    40336(s)   divzero
   [-]   1(h)       10(b)    40313(s)   pgdir
   [-]   1(h)       10(b)    40317(s)   sleepkill
   [-]   1(h)       10(b)    40340(s)   testbss
   [-]   1(h)       10(b)    40313(s)   yield
   [-]   1(h)       10(b)    40338(s)   exit
   [-]   1(h)       10(b)    40317(s)   faultread
lsdir: step 4
$
```

输入 hello

```
lsdir: step 4
Hello world!!.
I am process 14.
hello pass.
$
```

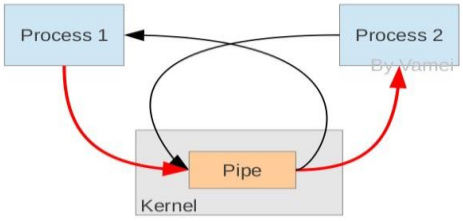
[TODO3] Unix 中 Pipe 机制以及软硬链接的设计

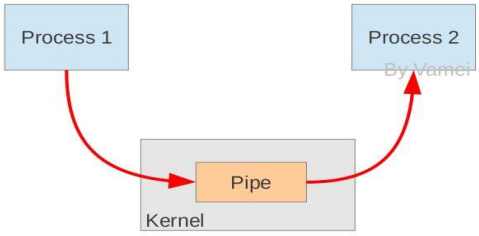
- **Pipe 机制简介**

管道技术是进程间通信的一种方式，和文件不同，管道以及管道的变种不会占据任何的磁盘空间，而只是使用了内核中的一个缓冲区。如果缓冲区满了，那么进程会被阻塞

UNIX 系统中创建管道的方式有两种：pipe()和 mkfifo()。前者创建了一个匿名的管道，后者创建了一个命名的管道，两者的唯一区别就是命名管道会一直留在文件系统中直到被删除，而匿名管道不存在于文件系统中，当文件描述符被关闭后将消失

从原理上，管道利用 fork 机制建立，从而让两个进程可以连接到同一个 PIPE 上。最开始的时候，上面的两个箭头都连接在同一个进程 Process 1 上(连接在 Process 1 上的两个箭头)。当 fork 复制进程的时候，会将这两个连接也复制到新的进程(Process 2)。随后，每个进程关闭自己不需要的一个连接（两个黑色的箭头被关闭; Process 1 关闭从 PIPE 来的输入连接，Process 2 关闭输出到 PIPE 的连接)，这样，剩下的红色连接就构成了如下图的 PIPE。



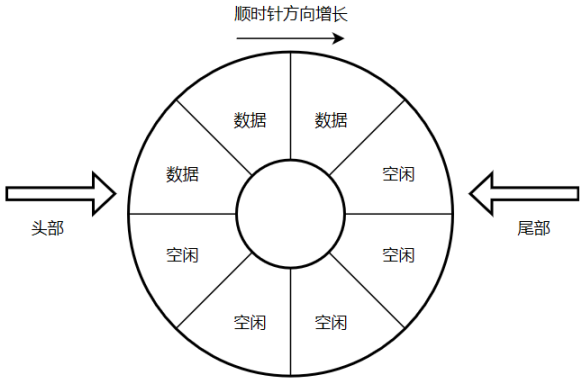


具体实现

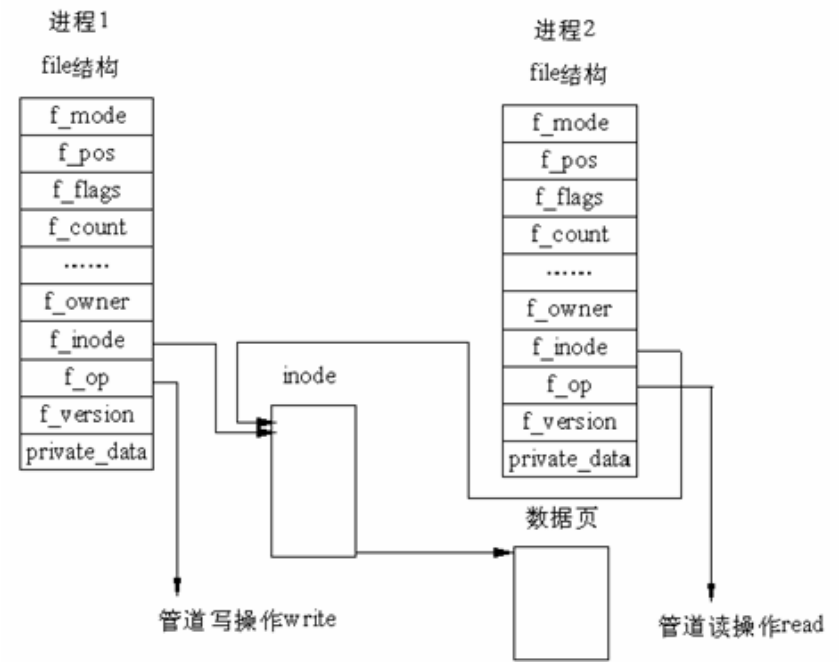
- **数据结构**

管道本质上就是一个操作系统内核管理的**环形缓冲区**，所以需要一块内存作为缓冲区，然后记录环形缓冲区的头部和尾部。当一个进程尝试从空管道读取或者向满管道写入数据时，操作系统内核需要将进程阻塞，所以还需要一个读取等待队列和写入等待队列

在环形缓冲中，头部指针指向下次需要取出的数据，尾部指针指向下次放入的区域。按照图 1 的设计，当头部指针与尾部指针相等的时候，缓冲区空，管道中没有信息，从管道中读取的进程会等待，直到另一端的进程放入信息；当头部指针恰好在尾部指针前一个位置的时候，缓冲区为满，尝试放入信息的进程会等待，直到另一端的进程取出信息；当两个进程都终结的时候，管道也自动消失



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而实现的。如下图



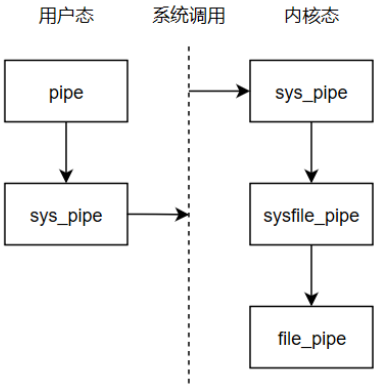
有两个 file 数据结构，但它们定义文件操作例程地址是不同的，其中一个是向管道中写入数据的例程地址，而另一个是从管道中读出数据的例程地址。这样，用户程序的系统调用仍然是通常的文件操作，而内核却利用这种抽象机制实现了管道这一特殊操作

- **管道操作**

除了管道的创建过程需要使用独立的系统调用外，管道的其他操作都是由虚拟文件系统控制的，这也就意味着需要提供一组操作接口供虚拟文件系统使用。

- **管道创建**

操作系统内核需要以系统调用的形式提供一个创建管道的接口
首先需要创建管道的信息节点，然后创建两个文件描述符用于读写（一个只读、一个只写）



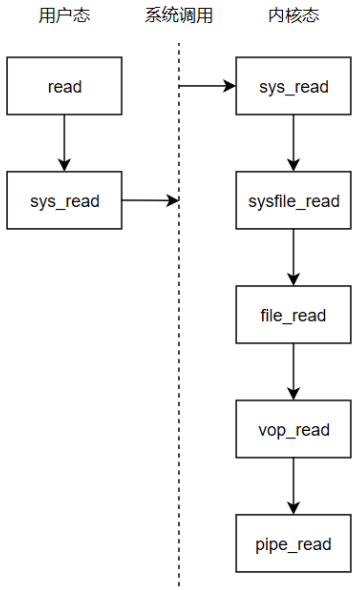
- **管道关闭和回收**

当管道信息节点的打开计数变为 0 后，管道关闭操作就会被虚拟文件系统调用，对于管道来说，引用计数值等于打开计数值，当管道相关的文件描述符全部关闭后，由回收操作进行内存资源回收，关闭操作不需要做任何工作。当管道的全部文件描述符关闭后，虚拟文件系统调用管道回收操作回收缓冲区内存以及信息节点使用的内存。

- **管道读取和写入**

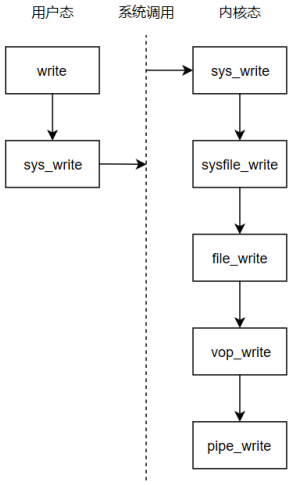
读取过程采用逐字节读取的方式读取数据，对于每一个字节的数据：

如果缓冲区非空，那么取出一个字节后，唤醒处于写入等待队列中的进程，然后尝试下一个字节；如果缓冲区为空， 那么进程将自己加入读取等待队列，主动进入阻塞状态，等待进程被唤醒后再次尝试；如果全部要求的数据全部读取完毕，那么结束读取过程，返回读取到的字节数



写入过程和读取过程非常类似，对于每一个字节的数据：

如果缓冲区非满，那么写入一个字节后，唤醒处于读取等待队列中的进程，然后尝试下一个字节；如果缓冲区为满， 那么进程将自己加入写入等待队列，主动进入阻塞状态，等待进程被唤醒后再次尝试；如果全部要求的数据全部写入完毕，那么结束读取过程，返回写入的字节数



- **设计基于“Unix 的硬链接和软连接机制”的方案**

硬链接和软连接的区别：

1）原理上：

- 硬链接(hard link)：A 是 B 的硬链接（A 和 B 都是文件名），则 A 的目录项中的 inode 节点号与 B 的目录项中的 **inode 节点号相同，即一个 inode 节点对应两个不同的文件名，两个文件名指向同一个文件**，A 和 B 对文件系统来说是完全平等的

如果删除了其中一个，对另外一个没有影响。每增加一个文件名，inode 节点上的链接数增加一，每删除一个对应的文件名，inode 节点上的链接数减一，直到为 0，inode 节点和对应的数据块被回收

- 软链接(soft link)：A 是 B 的软链接，A 的目录项中的 inode 节点号与 B 的目录项中的 **inode 节点号不相同，A 和 B 指向的是两个不同的 inode，继而指向两块不同的数据块。但是 A 的数据块中存放的只是 B 的路径名**（可以根据这个找到 B 的目录项）

A 和 B 之间是“主从”关系，如果 B 被删除了，A 仍然存在（因为两个是不同的文件），但指向的是一个无效的链接

2）使用限制上：

- 硬链接：a.不能对目录创建硬链接，原因：文件系统不能存在链接环,存在环的后果会导致例如文件遍历等操作的混乱；b：不能对不同的文件系统创建硬链接,即两个文件名要在相同的文件系统下；c：不能对不存在的文件创建硬链接

- 软链接：a.可以对目录创建软链接，遍历操作会忽略目录的软链接；b:可以跨文件系统 c:可以对不存在的文件创建软链接，因为放的只是一个字符串

三、实验体会

lab8 主要巩固了文件系统的知识，了解了 ucore 文件系统的架构、重要数据结构，实现了读文件操作和执行程序机制。数据结构和函数比较繁琐，但是分层看比较清晰。这是 ucore 的最后一次实验了！整一学期都在结合操作系统理论课一点一点学习，是真的有点儿难，真的花了不少时间，但也收获了不少吧！感谢帮助过自己的老师助教同学们~