第17章 虚拟文件系统

实验目的

- 理解虚拟文件系统的概念和原理
- 理解虚拟文件系统对象及其数据结构
- 理解虚拟文件系统的操作接口
- 通过编程实现一个虚拟文件系统

主要内容

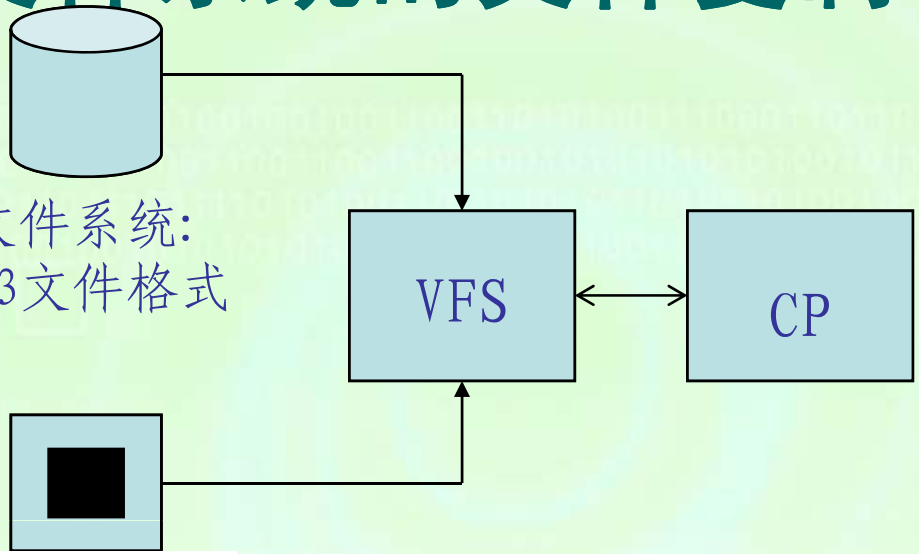
- 背景知识
  - 虚拟文件系统概念
  - VFS的组成(数据结构)
  - modutils软件包
- 实验内容
- 实现一个虚拟文件系统

虚拟文件系统实现目标

VFS作为内核子系统，其功能是将不同具体文件系统的接口统一起来，隐蔽它们的实现细节，为应用程序提供标准的、统一的、抽象的文件操作。

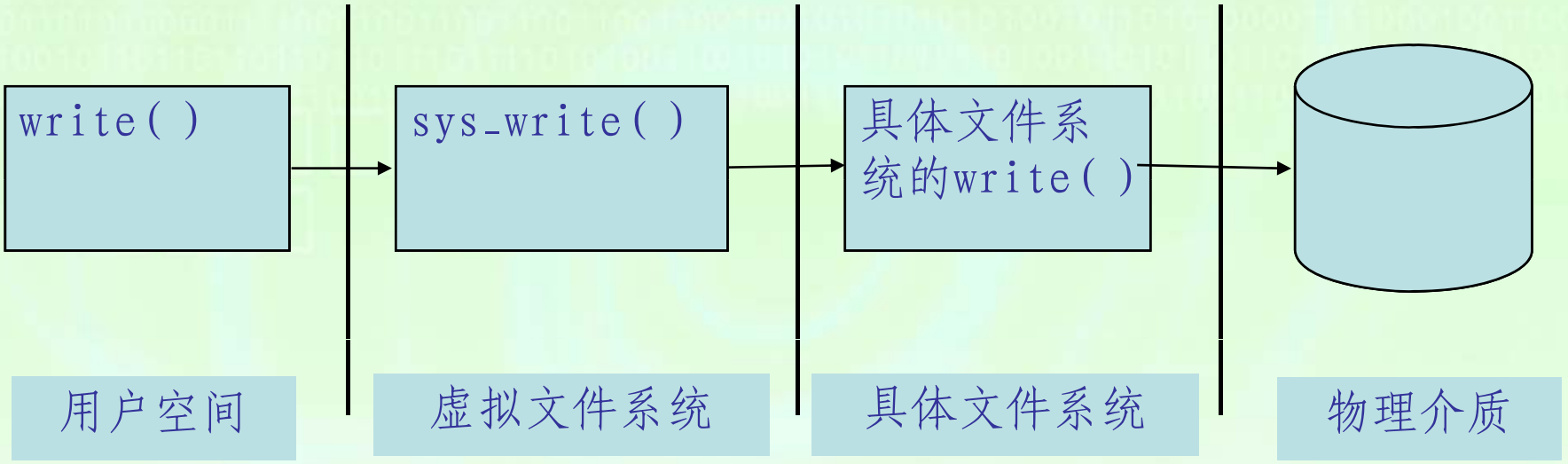
- 同时支持多种文件系统；且文件系统可交叉工作；
- 新开发出的文件系统可模块方式加入到操作系统中；
- 提供通过网络共享文件的支持，访问远程结点上的文件系统应与访问本地结点的文件系统一致；

跨文件系统的文件复制示意图




```
infile=open(" /user/test" ,O_RDONLY,0);
outfile=open(" /work/test" ,O_WRONLY | O_CREAT |
O_TRUNC,0600);
while ((charnum = read（ infile,buf,4096 ）)>0)
write（ outfile,buf,charnum ）
close(infile);
close(outfile);
```

VFS工作流程(1)

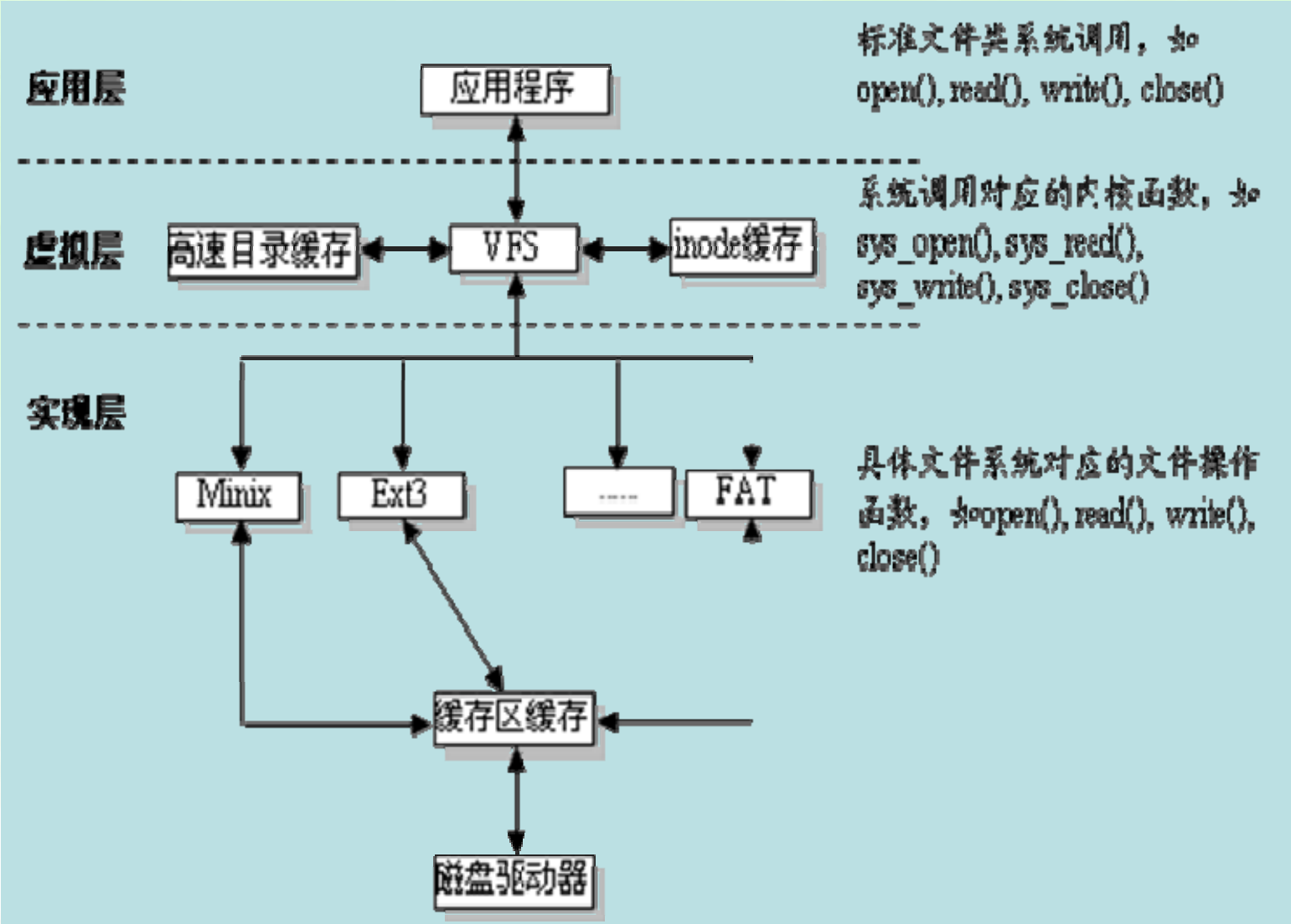


VFS工作流程(2)

读写操作的执行过程:

- 打开的文件用系统打开文件表file数据结构来表示，它有一个成员是指向文件操作表的指针file_operation *f_op，其中包含指向各种函数如read( )、write( )、open( )、close( )等的入口，每种具体文件系统都有自己的file operation结构，也就是有自己的操作函数。
- 当进程使用open( )打开文件时，将与具体文件建立连接，并以一个file结构作为纽带，将其中的f_op设置成指向某个具体的file_operation结构，也就指定了这个文件所属的文件系统。
- 当应用程序调用read( )函数时，就会陷入内核而调用sys_read( )内核函数，而sys_read( )就会通过file结构中的指针f_op去调用DOS文件系统的读函数，即:file→f_op→read( )函数。同样道理，write( )操作也会引发一个与输出文件相关的Ext3的file→f_op→write( )写函数的执行。

VFS结构


应用层
应用程序
标准文件类系统调用，如open(), read(), write(), close()
虚拟层
高速目录缓存
VFS
inode缓存
系统调用对应的内核函数，如sys_open(), sys_read(), sys_write(), sys_close()
实现层
Minix
Ext3
.....
FAT
具体文件系统对应的文件操作函数，如open(), read(), write(), close()
缓存区缓存
磁盘驱动器

虚拟文件系统的功能

VFS实质上是仅存于主存的，支持多种类型具体文件系统的运行环境，功能有：

- 记录安装的文件系统类型；
- 建立设备与文件系统的联系；
- 实现面向文件的通用操作；
- 涉及特定文件系统的操作时映射到具体文件系统中去。

主要内容

- 背景知识
  - 虚拟文件系统概念
  - **VFS的组成(数据结构)**
  - **modutils软件包**
- 实验内容
- 实现一个虚拟文件系统

VFS的组成

- 超级块对象-代表一个文件系统。
- 索引节点对象-代表一个文件。
- 目录项对象-代表路径中的一个组成部分。
- 文件对象-代表由进程已打开的一个文件。

VFS主要数据结构(1)

- 超级块(super block)对象-代表一个文件系统。存放已安装的文件系统信息，该对象对应于磁盘上的文件系统控制块，每个文件系统都对应一个超级块对象。如文件系统类型、超级块操作表指针、目录挂载点、设备和设备标识符、块大小。
- 索引节点(inode)对象-代表一个文件。存放通用的文件信息，该对象对应于磁盘上的文件FCB，即每个文件的inode对象，每个inode都有inode索引节点号，惟一地标识某个文件系统中的指定文件。

  如引用计数、硬链接数、用户ID、文件大小、文件属性、块大小、文件块数、超级块指针、inode操作表指针。

VFS主要数据结构(2)

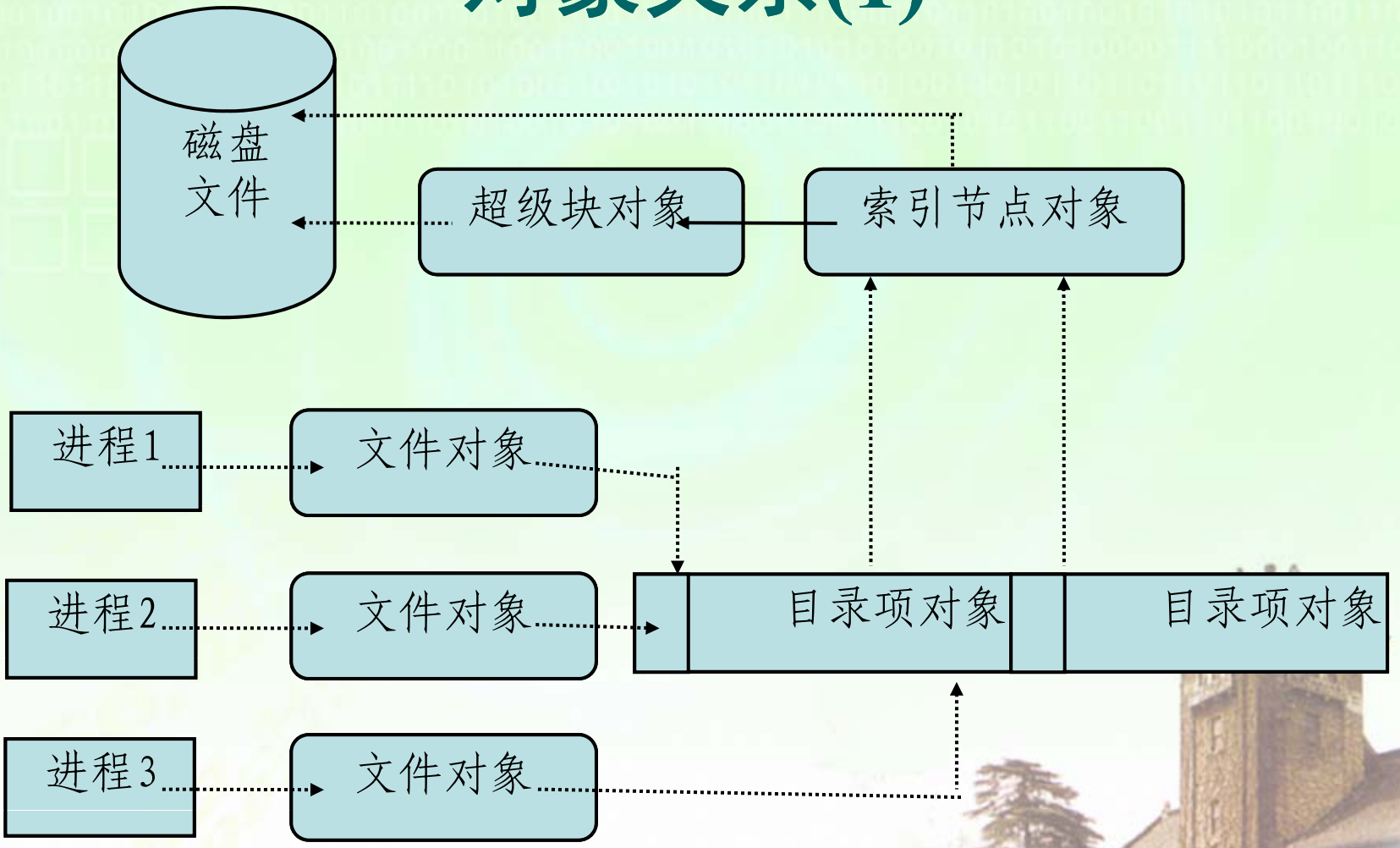
- 目录项(dentry)对象-代表路径中的一个组成部分。存放目录项与对应文件进行链接的信息，最近最常使用的dentry对象放在目录项高速缓存中，加快文件路径名搜索过程，提高系统性能。

  如相关inode、父目录/子目录信息、inode别名表、dentry操作表指针、超级块指针、 dentry标志、散列表。

- 文件(file)对象-代表由进程已打开的一个文件。存放已打开文件与进程的交互信息，这些信息仅当进程访问文件期间才存于主存中。

  如相关dentry、file操作表指针、引用次数、访问模式、当前位移、用户ID。

超级块、索引节点、目录项和文件对象关系(1)



超级块、索引节点、目录项和文件对象关系(2)

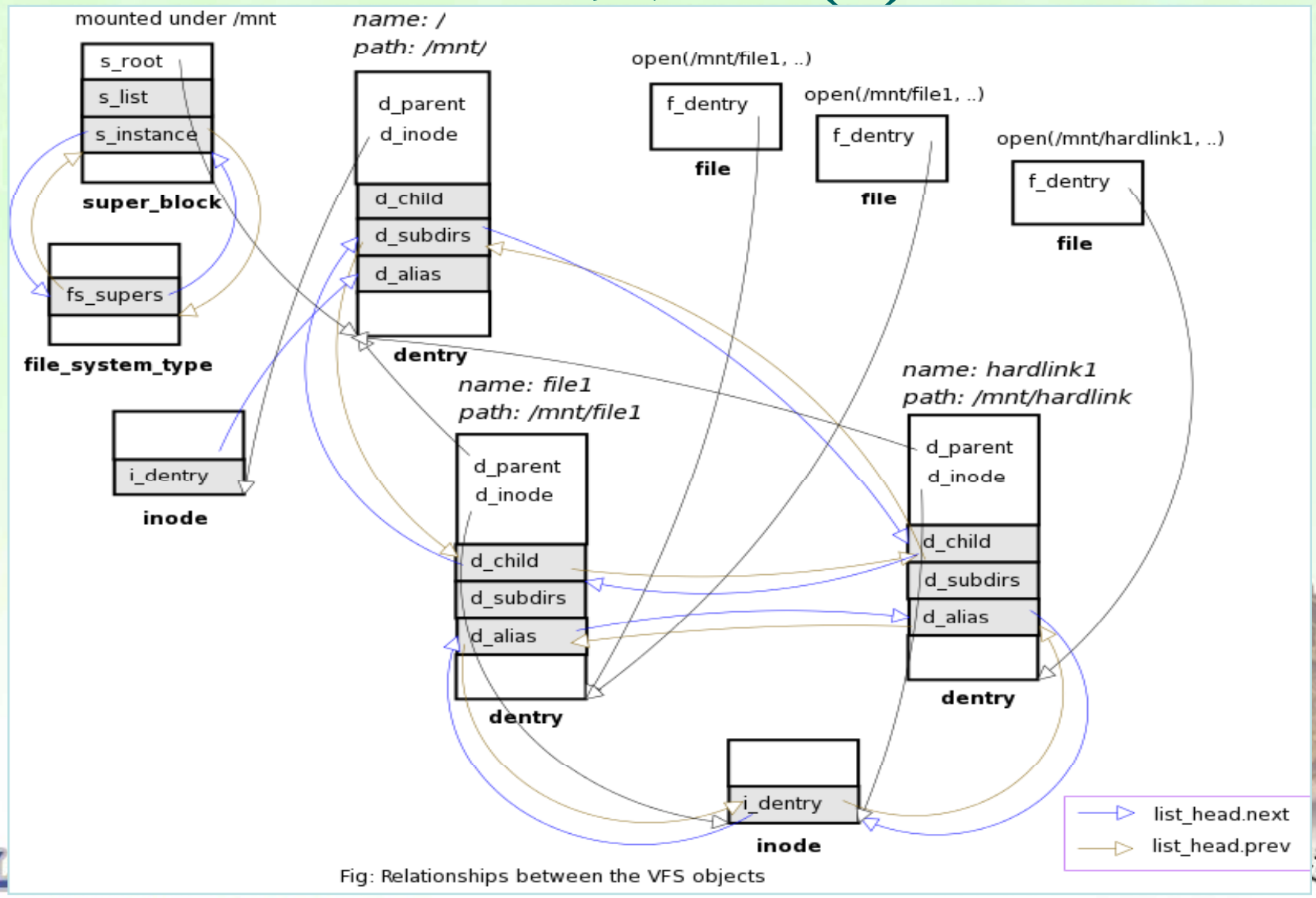
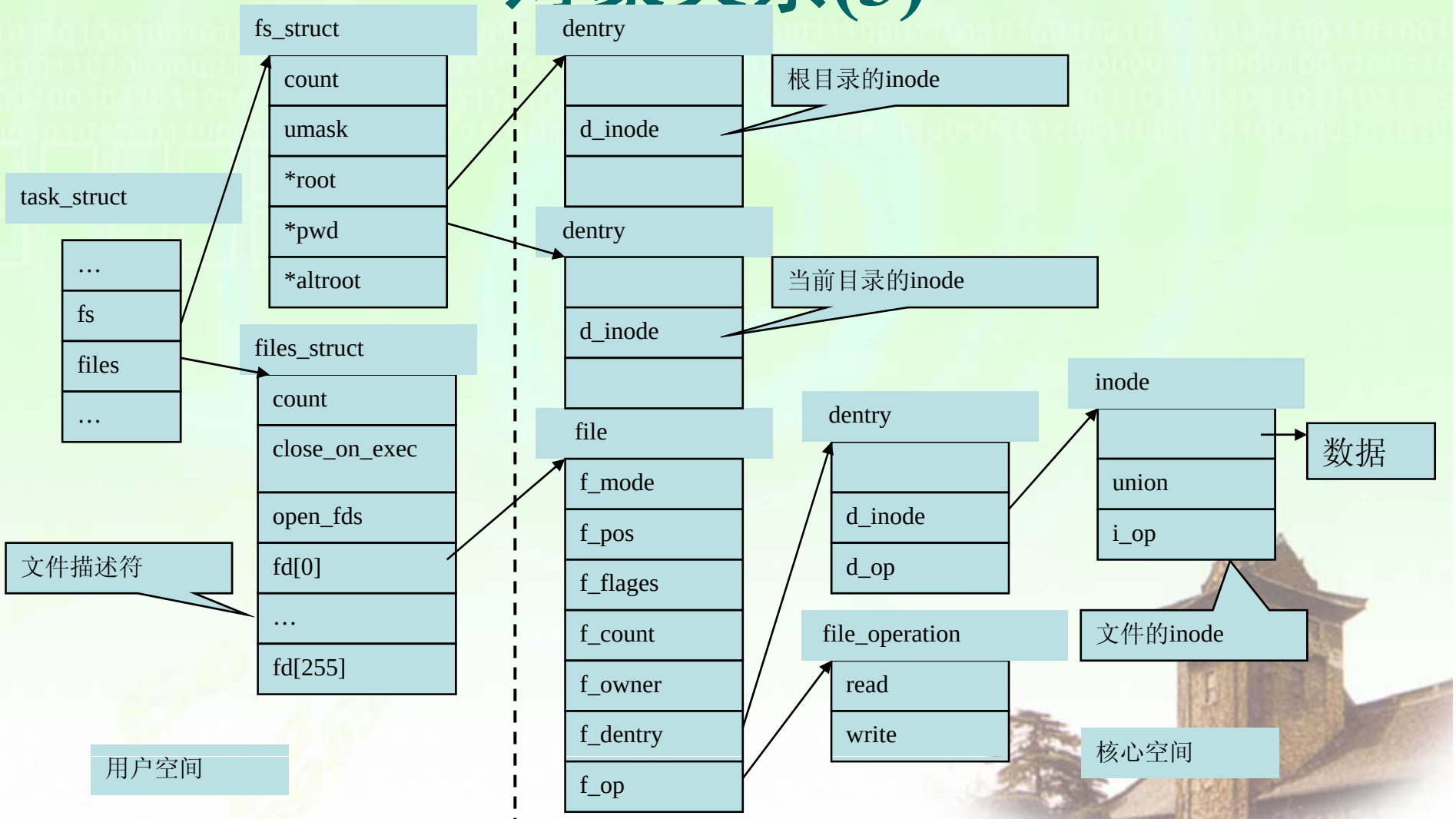


Fig: Relationships between the VFS objects

超级块、索引节点、目录项和文件对象关系(3)



函数和数据对象、以及数据对象中操作对象的关系

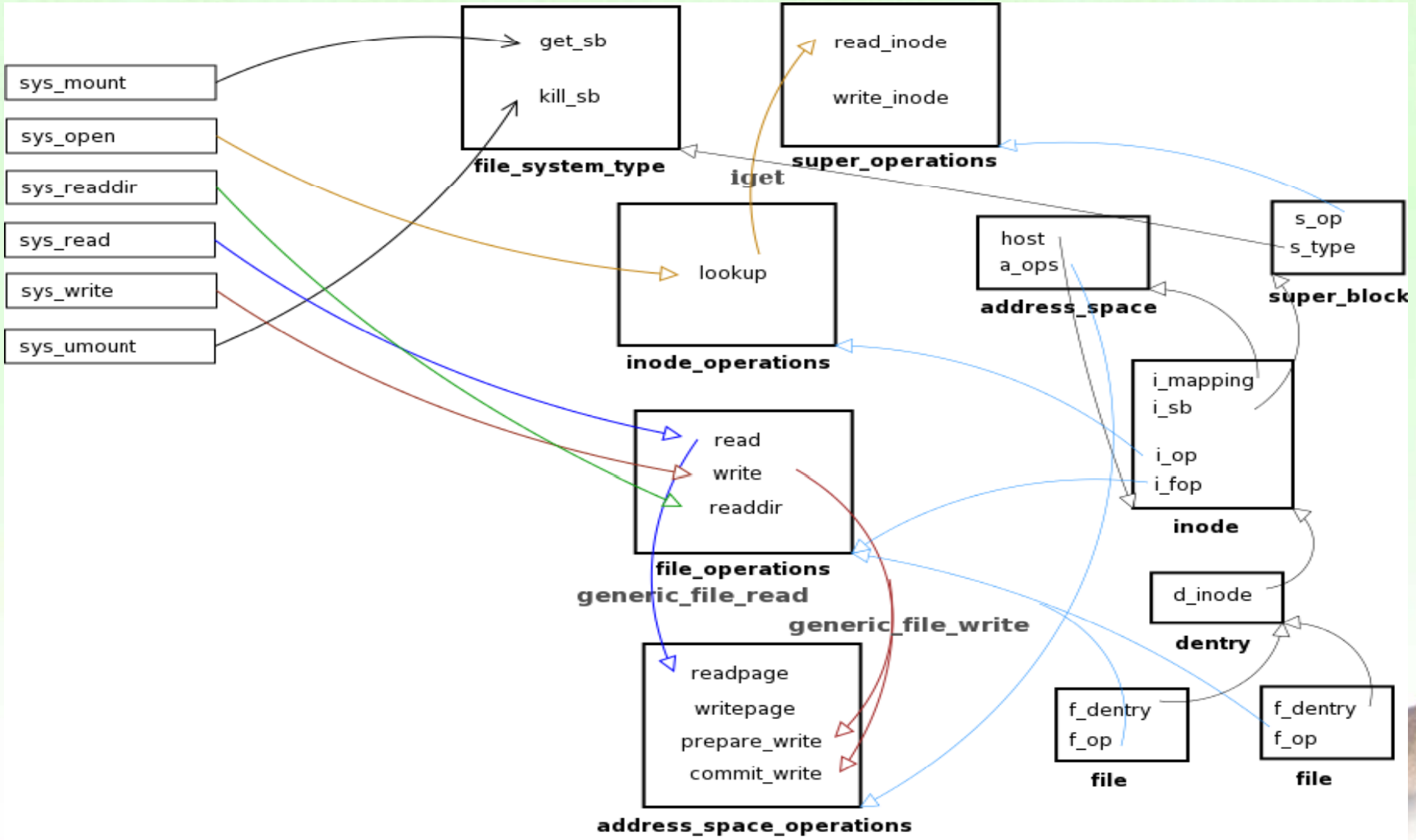
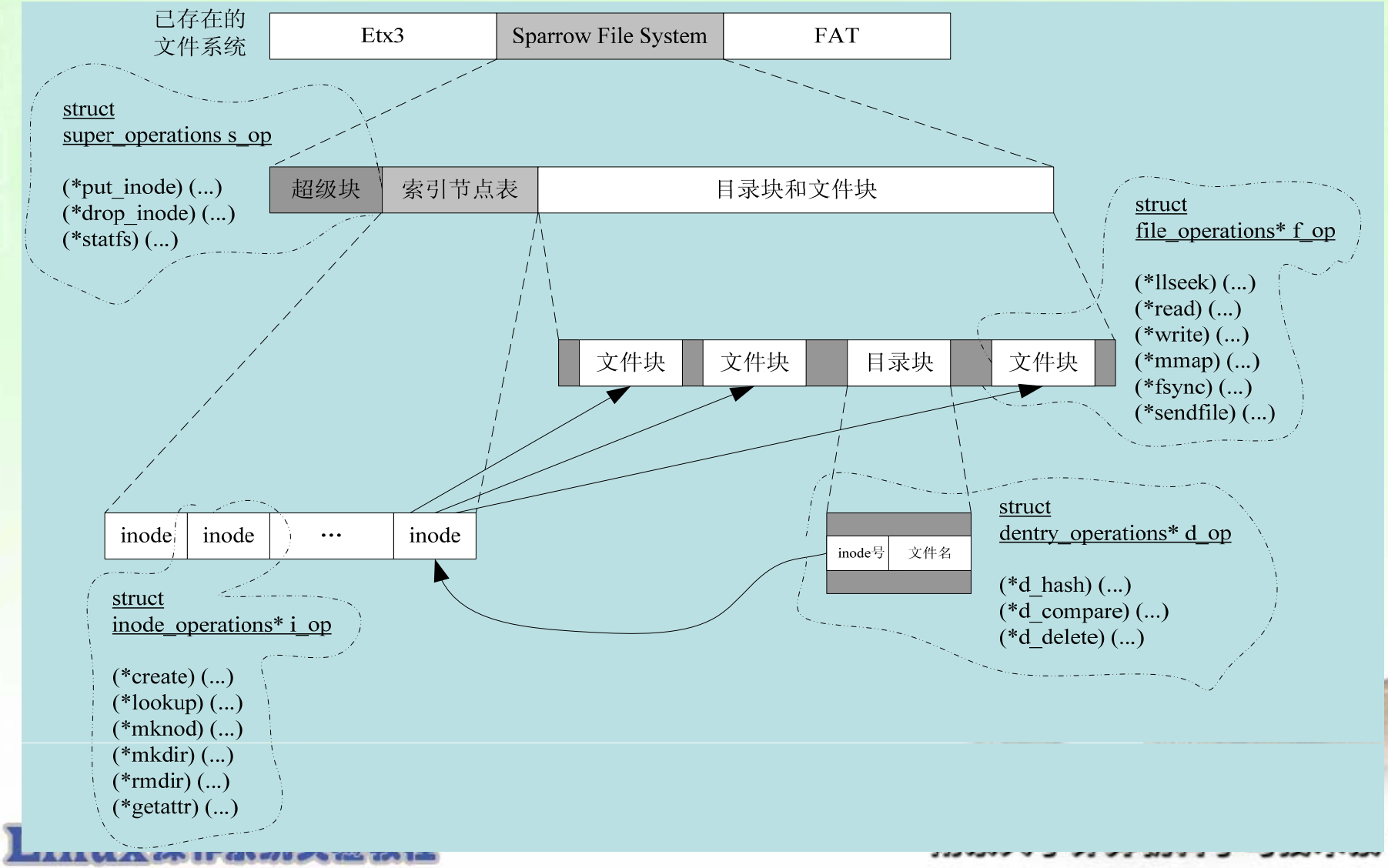


Fig: System call mapping and data structure relationships

Sparrow文件系统中4个基本对象需要实现的函数



主要内容

- 背景知识
  - 虚拟文件系统概念
  - VFS的组成(数据结构)
  - modutils软件包
- 实验内容
- 实现一个虚拟文件系统

modutils软件包

- insmod命令-载入模块
- rmmod命令-卸载模块
- lsmod命令-显示模块信息(模块模块名、大小和引用计数等)

主要内容

- 背景知识
  - 虚拟文件系统概念
  - VFS的组成(数据结构)
  - modutils软件包
- 实验内容
- 实现一个虚拟文件系统

实验 实现一个虚拟文件系统

需要完成以下内容:

- 1）完成Sparrow文件系统的编程、调试和挂载
- 2）让Sparrow文件系统具有文件链接功能
  - 软链接（Symbolic Links，又称为符号链接）
  - 硬链接（Hard Links）。

Sparrow文件系统

- Sparrow文件系统是基于Linux操作系统平台的，仅具有基本功能的文件系统，能够支持也仅仅支持以下功能:
  - 1）文件系统的初始化和安装、挂载/卸载;
  - 2）文件的创建和删除;
  - 3）目录的创建和删除。
- Sparrow文件系统是驻留在主存中的文件系统，当被卸载时，其中的文件将会丢失。
- Sparrow文件使用Linux内核模块技术来安装和挂载，首先，需要在内核中加载Sparrow文件系统这个内核模块，然后，使用mount命令把它添加进内核。

参考源程序(1)

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/pagemap.h>
#include <linux/version.h>
#include <linux/nls.h>
#include <linux/proc_fs.h>
#include <linux/backing-dev.h>
#include "sfs.h"
#define SAMPLEFS_MAGIC     0x73616d70
extern struct inode_operations sfs_dir_inode_ops;
extern struct inode_operations sfs_file_inode_ops;
extern struct file_operations sfs_file_operations;
extern struct address_space_operations sfs_aops;
```

参考源程序(2)

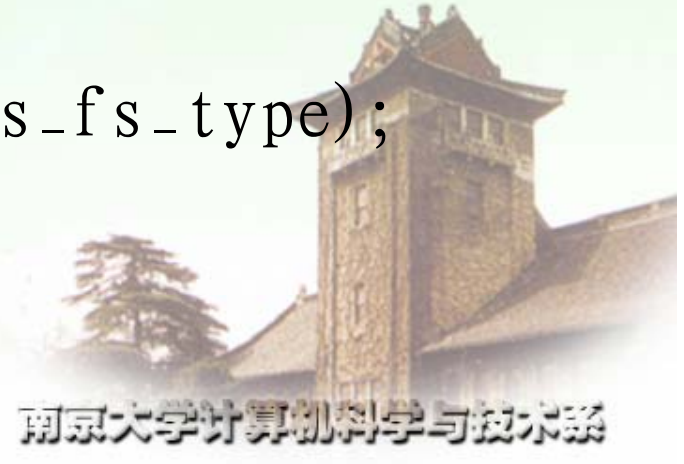
- 1) init_module函数

```
static struct file_system_type sfs_fs_type = {
.owner = THIS_MODULE,
.name = "sfs",
.get_sb = sfs_get_sb,
.kill_sb = kill_litter_super,
};
static int __init init_sfs_fs(void) {
printk(KERN_INFO "init samplefs\n");
/* some filesystems pass optional parms at load time */
if(sample_parm > 256) {
printk("sample_parm %d too large, reset to 10\n", sample_parm);
sample_parm = 10;
}
return register_filesystem(&samplefs_fs_type);
}
module_init(init_sfs_fs);
```

参考源程序(3)

- 2) cleanup_module函数

```
static void __exit exit_sfs_fs(void)
{
    printk(KERN_INFO "unloading sfs\n");
#ifdef CONFIG_PROC_FS
    //sfs_proc_clean( );
#endif
    unregister_filesystem(&sfs_fs_type);
}
module_exit(exit_sfs_fs)
```



参考源程序(4)

- 3)file_system_type.get_sb函数

```
static int sfs_fill_super(struct super_block * sb, void * data, int silent)
{
struct inode * inode;
struct sfs_sb_info * sfs_sb;
sb->s_maxbytes = MAX_LFS_FILESIZE;
sb->s_blocksize = PAGE_CACHE_SIZE;
sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
sb->s_magic = SAMPLEFS_MAGIC;
sb->s_op = &sfs_super_ops;
sb->s_time_gran = 1;
sb->s_fs_info = kzalloc(sizeof(struct sfs_sb_info),GFP_KERNEL);
sfs_sb = SFS_SB(sb);
if(!sfs_sb) {
return -ENOMEM;
}
inode = sfs_get_inode(sb, S_IFDIR | 0755, 0);
if(!inode) {
kfree(sfs_sb);
return -ENOMEM;
}
sb->s_root = d_alloc_root(inode);
if (!sb->s_root) {
iput(inode);
kfree(sfs_sb);
return -ENOMEM;
}
```

参考源程序(5)

- 4) file_system_type.kill_sb函数

```
static struct file_system_type sfs_fs_type = {
.owner = THIS_MODULE,
.name = "sfs",
.get_sb = sfs_get_sb,
.kill_sb = kill_litter_super,
};
```

页高速缓存(6)

- 5)address_space_operations.readpage函数
- 6)address_space_operations.commit_write函数
- struct address_space_operations sfs_aops = {
- .readpage = simple_readpage,
- .prepare_write = simple_prepare_write,
- .commit_write = simple_commit_write

参考源程序(7)

- 7) inode_operations.lookup函数

```
static struct
dentry *sfs_lookup(struct inode *dir, struct dentry *dentry, struct nameidata *nd)
{
struct sfs_sb_info * sfs_sb = SFS_SB(dir->i_sb);
if (dentry->d_name.len > NAME_MAX)
return ERR_PTR(-ENAMETOOLONG);
if(sfs_sb->flags & SFS_MNT_CASE)
dentry->d_op = &sfs_ci_dentry_ops;
else
dentry->d_op = &sfs_dentry_ops;
d_add(dentry, NULL);
return NULL;
}
```

参考源程序(8)

- 8) file_operations.read函数
- 9) file_operations.write函数

```
struct file_operations sfs_file_operations = {
    .read           = do_sync_read,
    .aio_read       = generic_file_aio_read,
    .write          = do_sync_write,
    .aio_write      = generic_file_aio_write,
    .mmap           = generic_file_mmap,
    .fsync          = simple_sync_file,
    .sendfile       = generic_file_sendfile,
    .llseek         = generic_file_llseek,
};
```

参考源程序(9)

```
struct inode_operations sfs_dir_inode_ops = {
    .create         = sfs_create,
    .lookup         = sfs_lookup,
    .link           = simple_link,
    .unlink         = simple_unlink,
    .symlink        = sfs_symlink,
    .mkdir          = sfs_mkdir,
    .rmdir          = simple_rmdir,
    .mknod          = sfs_mknod,
    .rename         = simple_rename,
};
```

参考源程序(10)

- 10)创建特殊文件

```
static int
sfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
{
        struct inode * inode = sfs_get_inode(dir->i_sb, mode, dev);
        int error = -ENOSPC;

        printk(KERN_INFO "sfs: mknod\n");
        if (inode) {
                if (dir->i_mode & S_ISGID) {
                        inode->i_gid = dir->i_gid;
                        if (S_ISDIR(mode))
                                inode->i_mode |= S_ISGID;
                }
                d_instantiate(dentry, inode);
                dget(dentry);   /* Extra count - pin the dentry in core */
                error = 0;
                dir->i_mtime = dir->i_ctime = CURRENT_TIME;

                /* real filesystems would normally use i_size_write function */
                dir->i_size += 0x20;  /* bogus small size for each dir entry */
        }
        return error;
}
```

参考源程序(11)

```
static int sfs_create(struct inode *dir, struct
dentry *dentry, int mode, struct nameidata *nd) {
    /*调用sfs_mknod函数 */
}
    /*用以支持链接功能*/
static int sfs_symlink(struct inode * dir, struct
dentry *dentry, const char * symname) {
.....
}
```

参考源程序(12)

```
/*挂载时返回超级块对象*/
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,18)
struct super_block * sfs_get_sb(struct file_system_type
*fs_type, int flags, const char *dev_name, void *data)
{
    return get_sb_nodev(fs_type, flags, data, sfs_fill_super);
}
#else
int sfs_get_sb(struct file_system_type *fs_type,int flags,
const char *dev_name, void *data, struct vfsmount *mnt)
{
    return get_sb_nodev(fs_type, flags, data, sfs_fill_super,
mnt);
}
#endif
```

参考源程序(13)

- 13)创建目录

```
static int sfs_mkdir(struct inode * dir, struct dentry * dentry, int mode)
{
        int retval = 0;
        retval = sfs_mknod(dir, dentry, mode | S_IFDIR, 0);
        /* link count is two for dir, for dot and dot dot */
        if (!retval)
                dir->i_nlink++;
        return retval;
}
```

参考源程序(14)

- 14)创建新的inode
- static int sfs_create(struct inode *dir, struct dentry *dentry, int mode,
- struct nameidata *nd)
- {
- return sfs_mknod(dir, dentry, mode | S_IFREG, 0);
- }

参考源程序(15)

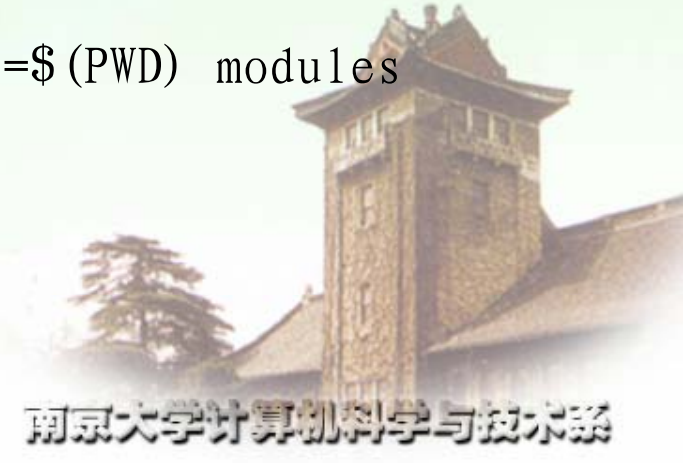
- 15)建立符号联结

```
static int sfs_symlink(struct inode * dir, struct dentry *dentry, const char
* symname)
{
        struct inode *inode;
        int error = -ENOSPC;

        inode = sfs_get_inode(dir->i_sb, S_IFLNK|S_IRWXUGO, 0);
        if (inode) {
                int l = strlen(symname)+1;
                error = page_symlink(inode, symname, l);
                if (!error) {
                        if (dir->i_mode & S_ISGID)
                                inode->i_gid = dir->i_gid;
                        d_instantiate(dentry, inode);
                        dget(dentry);
                        dir->i_mtime = dir->i_ctime = CURRENT_TIME;
                } else
                        iput(inode);
        }
        return error;
}
```

实验操作(1)

- 创建一个makefile文件

```
ifneq ($(KERNELRELEASE),)
obj-m += samplefs.o
samplefs-objs := super.o inode.o file.o
else
KERNEL_SOURCE := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
	$(MAKE) -C ${KERNEL_SOURCE} SUBDIRS=$(PWD) modules
clean :
	rm *.o *.ko
endif
```



实验操作(2)

- 1) $ make
  - 编译的结果是产生sfs.ko文件。
- 2) 加载内核模块
  - 以root身份加载该内核模块(使用sudo命令，后面的所有操作都使用sudo):
  - $ insmod samplefs.ko
  - 加载完成后，可使用lsmod命令查看加载是否成功。
- 3) 挂载文件系统
  - 将文件系统挂载到根目录树的/mnt路径下:
  - $ mount -t samplefs /junk /mnt
  - 挂载完成后，可使用mount命令查看是否挂载成功
  - 在/mnt路径下的文件系统就是Sparrow文件系统，可使用mkdir、rmdir、touch等命令创建/删除文件、创建/删除文件夹以及修改文件。

实验操作(3)

- 4) 卸载文件系统
  - 卸载文件系统使用命令:
  - $ umount /mnt
- 5) 卸载内核模块
  - 卸载Sfs内核模块使用命令:
  - $ rmmod sfs

实验验证

- 挂载文件系统: mount -t samplefs /junk /mnt
-
- 创建和删除文件夹: mkdir dir
- rmdir dir
-
- 创建文件: touch file
- 删除文件: rm file
-
- 创建硬链接: ln file hardLink
- 创建软链接: ln -s file symbolLink
-
- 从sfs拷贝到根文件系统ext3: cp file /home/demo/file