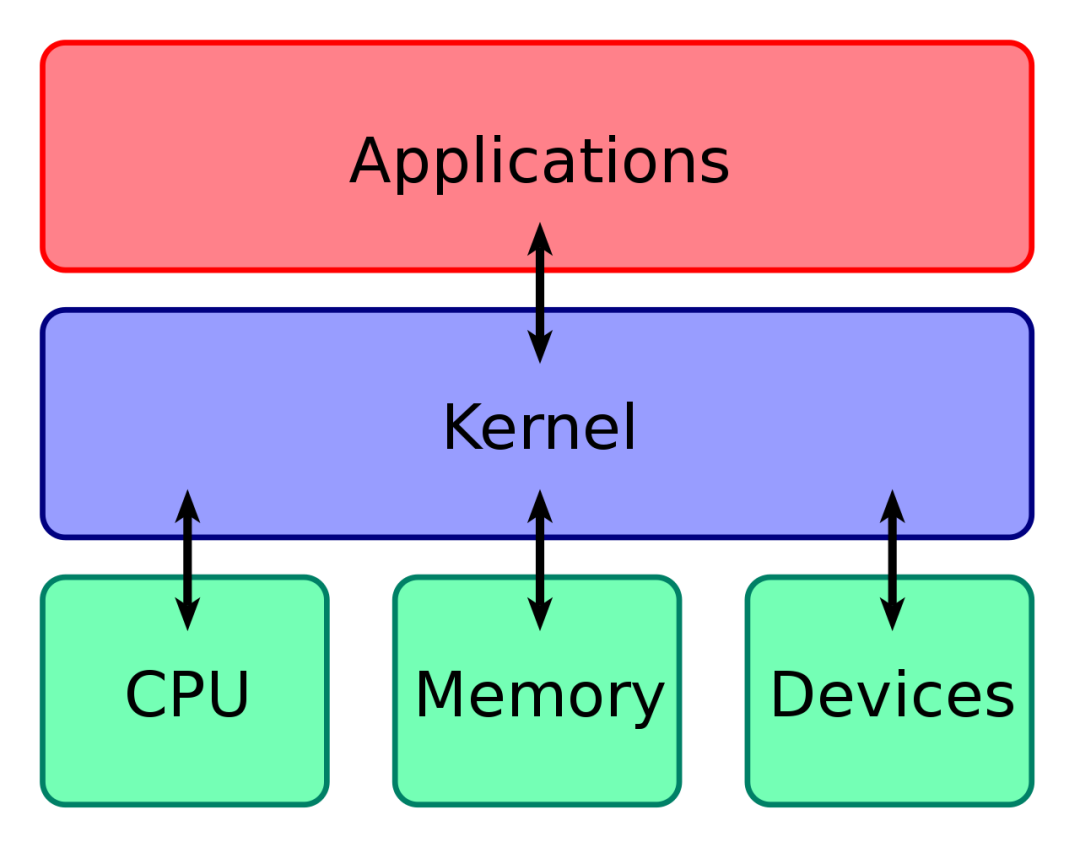
# 内核

## 什么是内核呢？

计算机是由各种外部硬件设备组成的，比如内存、cpu、硬盘等，如果每个应用都要和这些硬件设备对接通信协议，那这样太累了。

所以，这个中间人就由内核来负责，让内核作为应用连接硬件设备的桥梁，应用程序只需关心与内核交互，不用关心硬件的细节。



## 内核有哪些能力呢？

现代操作系统，内核一般会提供 4 个基本能力：

* 管理进程、线程，决定哪个进程、线程使用 CPU，也就是进程调度的能力；
* 管理内存，决定内存的分配和回收，也就是内存管理的能力；
* 管理硬件设备，为进程与硬件设备之间提供通信能力，也就是硬件通信能力；
* 提供系统调用，如果应用程序要运行更高权限运行的服务，那么就需要有系统调用，它是用户程序与操作系统之间的接口。

## 内核是怎么工作的？

内核具有很高的权限，可以控制 cpu、内存、硬盘等硬件，而应用程序具有的权限很小，因此大多数操作系统，把内存分成了两个区域：

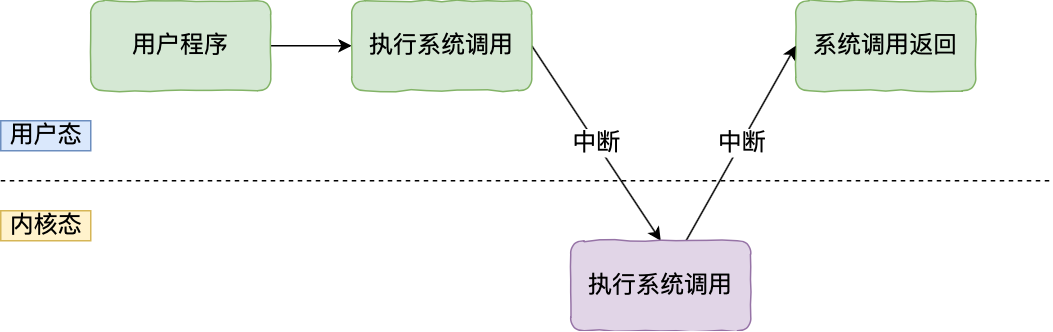
内核空间，这个内存空间只有内核程序可以访问；

用户空间，这个内存空间专门给应用程序使用；

用户空间的代码只能访问一个局部的内存空间，而内核空间的代码可以访问所有内存空间。

因此，当程序使用用户空间时，我们常说该程序在用户态执行，而当程序使内核空间时，程序则在内核态执行。

应用程序如果需要进入内核空间，就需要通过「系统调用」，下面来看看系统调用的过程：



内核程序执行在内核态，用户程序执行在用户态。当应用程序使用系统调用时，会产生一个中断。发生中断后， CPU 会中断当前在执行的用户程序，转而跳转到中断处理程序，也就是开始执行内核程序。内核处理完后，主动触发中断，把 CPU 执行权限交回给用户程序，回到用户态继续工作。

# Linux

## Linux 内核设计的理念主要有这几个点：

* MutiTask，多任务
* SMP，对称多处理
* ELF，可执行文件链接格式
* Monolithic Kernel，宏内核

## MutiTask多任务

MutiTask 的意思是多任务，代表着 Linux 是一个多任务的操作系统。

多任务意味着可以有多个任务同时执行，这里的「同时」可以是并发或并行：

对于单核 CPU 时，可以让每个任务执行一小段时间，时间到就切换另外一个任务，从宏观角度看，一段时间内执行了多个任务，这被称为并发。

对于多核 CPU 时，多个任务可以同时被不同核心的 CPU 同时执行，这被称为并行。

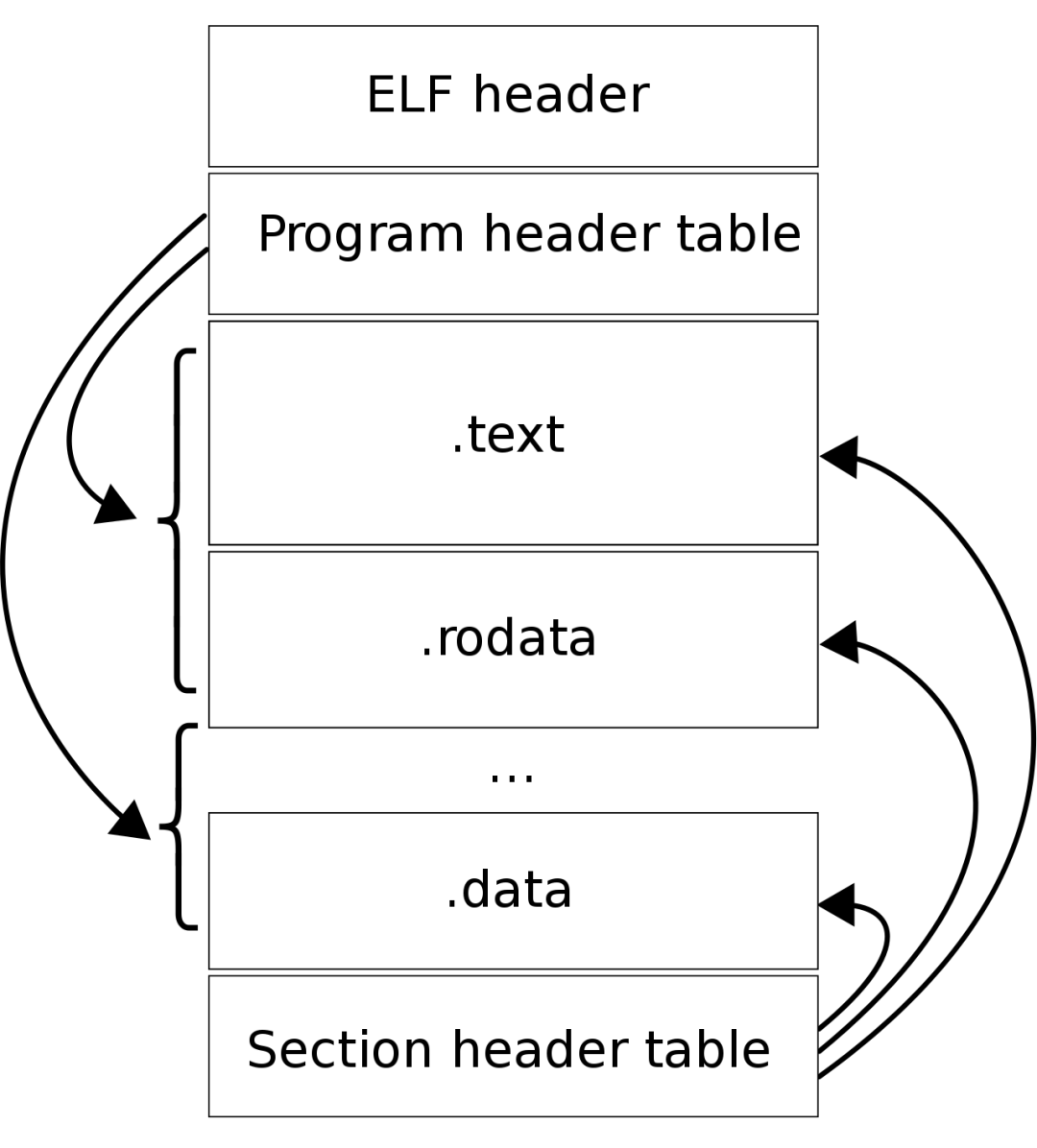
## SMP对称多处理

SMP 的意思是对称多处理，代表着每个 CPU 的地位是相等的，对资源的使用权限也是相同的，多个 CPU 共享同一个内存，每个 CPU 都可以访问完整的内存和硬件资源。

这个特点决定了 Linux 操作系统不会有某个 CPU 单独服务应用程序或内核程序，而是每个程序都可以被分配到任意一个 CPU 上被执行。

## ELF

ELF 的意思是可执行文件链接格式，它是 Linux 操作系统中可执行文件的存储格式，你可以从下图看到它的结构：



ELF 把文件分成了一个个分段，每一个段都有自己的作用，具体每个段的作用这里我就不详细说明了，感兴趣的同学可以去看《程序员的自我修养——链接、装载和库》这本书。

另外，ELF 文件有两种索引，Program header table 中记录了「运行时」所需的段，而 Section header table 记录了二进制文件中各个「段的首地址」。

那 ELF 文件怎么生成的呢？

我们编写的代码，首先通过「编译器」编译成汇编代码，接着通过「汇编器」变成目标代码，也就是目标文件，最后通过「链接器」把多个目标文件以及调用的各种函数库链接起来，形成一个可执行文件，也就是 ELF 文件。

那 ELF 文件是怎么被执行的呢？

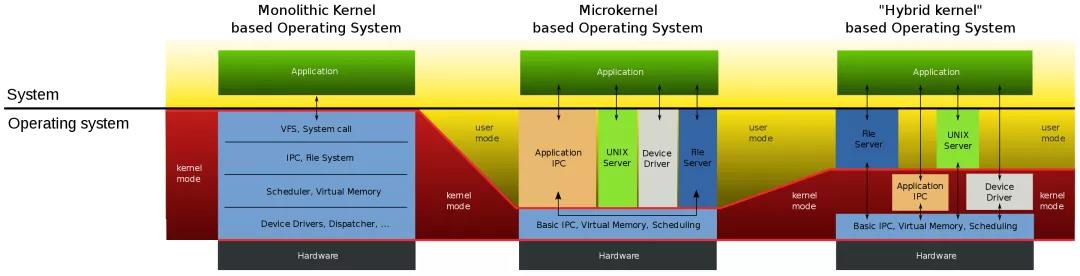
执行 ELF 文件的时候，会通过「装载器」把 ELF 文件装载到内存里，CPU 读取内存中的指令和数据，于是程序就被执行起来了。

## Monolithic Kernel宏内核

Monolithic Kernel 的意思是宏内核，Linux 内核架构就是宏内核，意味着 Linux 的内核是一个完整的可执行程序，且拥有最高的权限。

宏内核的特征是系统内核的所有模块，比如进程调度、内存管理、文件系统、设备驱动等，都运行在内核态。

不过，Linux 也实现了动态加载内核模块的功能，例如大部分设备驱动是以可加载模块的形式存在的，与内核其他模块解藕，让驱动开发和驱动加载更为方便、灵活。



与宏内核相反的是微内核，微内核架构的内核只保留最基本的能力，比如进程调度、虚拟机内存、中断等，把一些应用放到了用户空间，比如驱动程序、文件系统等。这样服务与服务之间是隔离的，单个服务出现故障或者完全攻击，也不会导致整个操作系统挂掉，提高了操作系统的稳定性和可靠性。

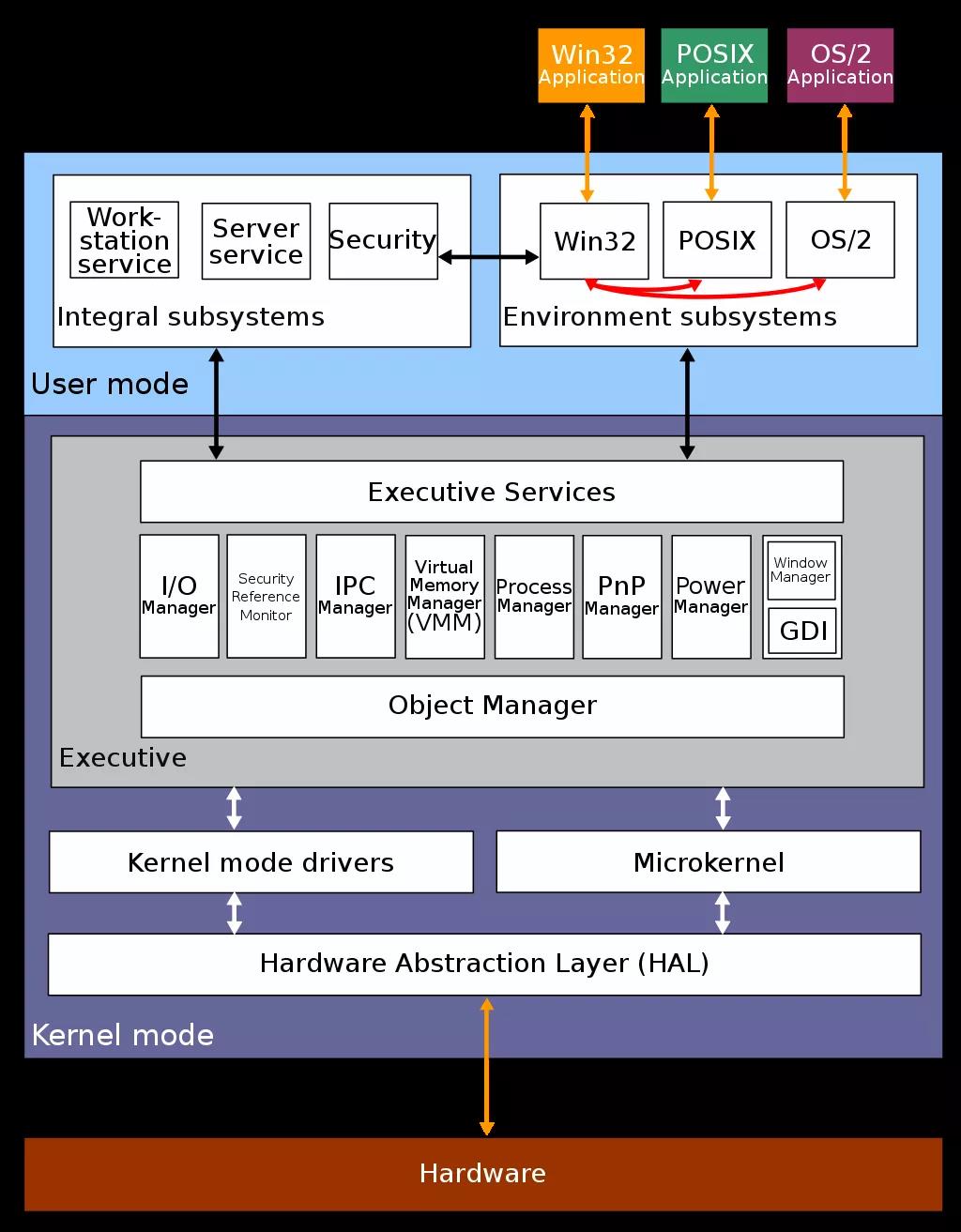
微内核内核功能少，可移植性高，相比宏内核有一点不好的地方在于，由于驱动程序不在内核中，而且驱动程序一般会频繁调用底层能力的，于是驱动和硬件设备交互就需要频繁切换到内核态，这样会带来性能损耗。华为的鸿蒙操作系统的内核架构就是微内核。

还有一种内核叫混合类型内核，它的架构有点像微内核，内核里面会有一个最小版本的内核，然后其他模块会在这个基础上搭建，然后实现的时候会跟宏内核类似，也就是把整个内核做成一个完整的程序，大部分服务都在内核中，这就像是宏内核的方式包裹着一个微内核。

# Windows设计

当今 Windows 7、Windows 10 使用的内核叫 Windows NT，NT 全称叫 New Technology。

下图是 Windows NT 的结构图片：

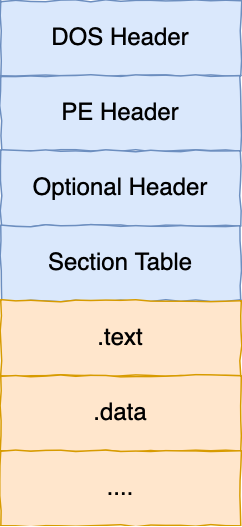


Windows 和 Linux 一样，同样支持 MutiTask 和 SMP，但不同的是，Windows 的内核设计是混合型内核，在上图你可以看到内核中有一个 MicroKernel 模块，这个就是最小版本的内核，而整个内核实现是一个完整的程序，含有非常多模块。

Windows 的可执行文件的格式与 Linux 也不同，所以这两个系统的可执行文件是不可以在对方上运行的。

Windows 的可执行文件格式叫 PE，称为可移植执行文件，扩展名通常是.exe、.dll、.sys等。

PE 的结构你可以从下图中看到，它与 ELF 结构有一点相似。



# 总结

对于内核的架构一般有这三种类型：

## 宏内核

包含多个模块，整个内核像一个完整的程序；

## 微内核

有一个最小版本的内核，一些模块和服务则由用户态管理；

## 混合内核

是宏内核和微内核的结合体，内核中抽象出了微内核的概念，也就是内核中会有一个小型的内核，其他模块就在这个基础上搭建，整个内核是个完整的程序；

Linux 的内核设计是采用了宏内核，Windows 的内核设计则是采用了混合内核。

这两个操作系统的可执行文件格式也不一样， Linux 可执行文件格式叫作 ELF，Windows 可执行文件格式叫作 PE。

# 核心

## **[Linux内核目录结构](https://www.cnblogs.com/preacher/p/4635599.html)**

arch 包括所有和体系结构相关的核心代码。

include 包括编译内核所需要的大部分头文件

init 包含内核的初始化代码(不是系统的引导代码)，有main.c和Version.c两个文件

mm 包含所有的内存管理代码

drivers 包含系统中所有的设备驱动程序

ipc 包含核心进程间的通信代码

fs 存放Linux支持的文件系统代码

kernel 包含内核管理的核心代码

net 内核的网络部分代码，其每个子目录对应于网络的一个方面

lib 包含核心的库代码

scripts 包含用于配置核心的脚本文件

Documentation 一些文档，是对每个目录作用的具体说明

## 通常Linux会有以下目录

arch 子目录包括所有和体系结构相关的核心代码。它还有更深的子目录，每一个代表一种支持的体系结构

include 子目录包括编译核心所需要的大部分 include 文件。它也有更深的子目录，每一个支持的体系结构一个。 include/asm 是这个体系结构所需要的真实的 include 目录的软链接，例如 include/asm-i386 。为了改变体系结构，你需要编辑核心的 makefile ，重新运行 Linux 的核心配置程序

init 这个目录包含核心的初始化代码，这时研究核心如何工作的一个非常好的起点

mm 这个目录包括所有的内存管理代码。和体系结构相关的内存管理代码位于 arch/\*/mm/

drivers 系统所有的设备驱动程序在这个目录。它们被划分成设备驱动程序类

ipc 这个目录包含核心的进程间通讯的代码

modules 这只是一个用来存放建立好的模块的目录

fs 所有的文件系统代码。被划分成子目录，每一个支持的文件系统一个

kernel 主要的核心代码。同样，和体系相关的核心代码放在 arch/\*/kernel

net 核心的网络代码

lib 这个目录放置核心的库代码。和体系结构相关的库代码在 arch/\*/lib/

scripts 这个目录包含脚本（例如 awk 和 tk 脚本），用于配置核心

### 按照以下顺序阅读源代码会轻松点

核心功能(kernel)

内存管理(mm)

文件系统(fs)

进程通讯(ipc)

网络(net)

系统启动和初始化(init/main和head.S)

其他等等

### System Startup and Initialization （系统启动和初始化）

在一个 Intel 系统上，当 loadlin.exe 或 LILO 把核心加载到内存并把控制权交给它的时候，核心开始启动。这一部分看 arch/i386/kernel/head.S 。 head.S 执行一些和体系结构相关的设置工作并跳到 init/main.c 中的 main() 例程。

### Memory Management （内存管理）

代码大多在 mm 但是和体系结构相关的代码在 arch/\*/mm 。 Page fault 处理代码在 mm/memory.c 中，内存映射和页缓存代码在 mm/filemap.c 中。 Buffer cache 在 mm/buffer.c 中实现，交换缓存在 mm/swap\_state.c 和 mm/swapfile.c 中。

### Kernel

大部分相对通用的代码在 kernel ，和体系结构相关的代码在 arch/\*/kernel 。调度程序在 kernel/sched.c，fork 代码在 kernel/fork.c 。 bottom half 处理代码在 include/linux/interrupt.h 。 task\_struct 数据结构可以在 include/linux/sched.h 中找到

### PCI

PCI 伪驱动程序在 drivers/pci/pci.c ，系统范围的定义在 include/linux/pci.h 。每一种体系结构都有一些特殊的 PCI BIOS 代码， Alpha AXP 的位于 arch/alpha/kernel/bios32.c

### Interprocess Communication

全部在 ipc 目录。所有系统 V IPC 对象都包括 ipc\_perm 数据结构，可以在 include/linux/ipc.h 中找到。系统 V 消息在 ipc/msg.c 中实现，共享内存在 ipc/shm.c 中，信号灯在 ipc/sem.c 。管道在 ipc/pipe.c 中实现。

### Interrupt Handling

核心的中断处理代码几乎都是和微处理器（通常也和平台）相关。 Intel 中断处理代码在 arch/i386/kernel/irq.c 它的定义在 incude/asm-i386/irq.h 。

### Device Drivers （设备驱动程序）

Linux 核心源代码的大部分代码行在它的设备驱动程序中。 Linux 所有的设备驱动程序源代码都在 drivers 中，但是它们被进一步分类：

/block 块设备驱动程序比如 ide （ ide.c ）。如果你希望查看所有可能包含文件系统的设备是如何初始化的，你可以看 drivers/block/genhd.c 中的 device\_setup() 。它不仅初始化硬盘，也初始化网络，因为你安装 nfs 文件系统的时候需要网络。块设备包括基于 IDE 和 SCSI 设备。

/char 这里可以查看基于字符的设备比如 tty ，串行口等。

/cdrom Linux 所有的 CDROM 代码。在这里可以找到特殊的 CDROM 设备（比如 Soundblaster CDROM ）。注意 ide CD 驱动程序是 drivers/block 中的 ide-cd.c ，而 SCSI CD 驱动程序在 drivers/scsi/scsi.c 中

/pci PCI 伪驱动程序。这是一个观察 PCI 子系统如何被映射和初始化的好地方。 Alpha AXP PCI 整理代码也值得在 arch/alpha/kernel/bios32.c 中查看

/scsi 在这里不但可以找到所有的 Linux 支持的 scsi 设备的驱动程序，也可以找到所有的 SCSI 代码

/net 在这里可以找到网络设备驱动程序比如 DEC Chip 21040 PCI 以太网驱动程序在 tulip.c 中

/sound 所有的声卡驱动程序的位置

### File Systems （文件系统）

EXT2 文件系统的源程序都在 fs/ext2/ 子目录，数据结构的定义在 include/linux/ext2\_fs.h,ext2\_fs\_i.h 和 ext2\_fs\_sb.h 中。虚拟文件系统的数据结构在 include/linux/fs.h 中描述，代码是 fs/\* 。 Buffer cache 和 update 核心守护进程都是用 fs/buffer.c 实现的

### Network （网络）

网络代码放在 net 子目录，大部分的 include 文件在 include/net 。 BSD socket 代码在 net/socket.c ， Ipv4 INET socket 代码在 net/ipv4/af\_inet.c 中。通用协议的支持代码（包括 sk\_buff 处理例程）在 net/core 中， TCP/IP 网络代码在 net/ipv4 。网络设备驱动程序在 drivers/net

### Modules （模块）

核心模块代码部分在核心，部分在 modules 包中。核心代码全部在 kernel/modules.c ，数据结果和核心守护进程 kerneld 的消息则分别在 include/linux/module.h 和 include/linux/kerneld.h 中。你可能也希望在 include/linux/elf.h 中查看一个 ELF 目标文件的结构

## 结构体

extern struct task\_struct init\_task;

extern struct task\_struct \*task[NR\_TASKS];

extern struct task\_struct \*last\_task\_used\_math;

extern struct task\_struct \*current; //表示当前用户

struct task\_struct {

/\* these are hardcoded - don't touch \*/

volatile long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/

long counter;

long priority;

unsigned long signal;

unsigned long blocked; /\* bitmap of masked signals \*/

unsigned long flags; /\* per process flags, defined below \*/

int errno;

int debugreg[8]; /\* Hardware debugging registers \*/

/\* various fields \*/

struct task\_struct \*next\_task, \*prev\_task;

struct sigaction sigaction[32];

unsigned long saved\_kernel\_stack;

unsigned long kernel\_stack\_page;

int exit\_code, exit\_signal;

int elf\_executable:1;

int dumpable:1;

int swappable:1;

int did\_exec:1;

unsigned long start\_code,end\_code,end\_data,start\_brk,brk,start\_stack,start\_mmap;

unsigned long arg\_start, arg\_end, env\_start, env\_end;

int pid,pgrp,session,leader;

int groups[NGROUPS];

/\*

\* pointers to (original) parent process, youngest child, younger sibling,

\* older sibling, respectively. (p->father can be replaced with

\* p->p\_pptr->pid)

\*/

struct task\_struct \*p\_opptr,\*p\_pptr, \*p\_cptr, \*p\_ysptr, \*p\_osptr;

struct wait\_queue \*wait\_chldexit; /\* for wait4() \*/

/\*

\* For ease of programming... Normal sleeps don't need to

\* keep track of a wait-queue: every task has an entry of its own

\*/

unsigned short uid,euid,suid;

unsigned short gid,egid,sgid;

unsigned long timeout;

unsigned long it\_real\_value, it\_prof\_value, it\_virt\_value;

unsigned long it\_real\_incr, it\_prof\_incr, it\_virt\_incr;

long utime,stime,cutime,cstime,start\_time;

unsigned long min\_flt, maj\_flt;

unsigned long cmin\_flt, cmaj\_flt;

struct rlimit rlim[RLIM\_NLIMITS];

unsigned short used\_math;

unsigned short rss; /\* number of resident pages \*/

char comm[16];

struct vm86\_struct \* vm86\_info;

unsigned long screen\_bitmap;

/\* file system info \*/

int link\_count;

int tty; /\* -1 if no tty, so it must be signed \*/

unsigned short umask;

struct inode \* pwd;

struct inode \* root;

struct inode \* executable;

struct vm\_area\_struct \* mmap;

struct shm\_desc \*shm;

struct sem\_undo \*semun;

struct file \* filp[NR\_OPEN];

fd\_set close\_on\_exec;

/\* ldt for this task - used by Wine. If NULL, default\_ldt is used \*/

struct desc\_struct \*ldt;

/\* tss for this task \*/

struct tss\_struct tss;

#ifdef NEW\_SWAP

unsigned long old\_maj\_flt; /\* old value of maj\_flt \*/

unsigned long dec\_flt; /\* page fault count of the last time \*/

unsigned long swap\_cnt; /\* number of pages to swap on next pass \*/

short swap\_table; /\* current page table \*/

short swap\_page; /\* current page \*/

#endif NEW\_SWAP

struct vm\_area\_struct \*stk\_vma;

};

包括struct inode \* pwd和struct inode \* root，表示当前及根节点inode。

# 核心

## 进程结构体task\_struct

struct task\_struct {

/\* these are hardcoded - don't touch \*/

volatile long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/

long counter;

long priority;

unsigned long signal;

unsigned long blocked; /\* bitmap of masked signals \*/

unsigned long flags; /\* per process flags, defined below \*/

int errno;

int debugreg[8]; /\* Hardware debugging registers \*/

/\* various fields \*/

struct task\_struct \*next\_task, \*prev\_task;

struct sigaction sigaction[32];

unsigned long saved\_kernel\_stack;

unsigned long kernel\_stack\_page;

int exit\_code, exit\_signal;

int elf\_executable:1;

int dumpable:1;

int swappable:1;

int did\_exec:1;

unsigned long start\_code,end\_code,end\_data,start\_brk,brk,start\_stack,start\_mmap;

unsigned long arg\_start, arg\_end, env\_start, env\_end;

int pid,pgrp,session,leader;

int groups[NGROUPS];

/\*

\* pointers to (original) parent process, youngest child, younger sibling,

\* older sibling, respectively. (p->father can be replaced with

\* p->p\_pptr->pid)

\*/

struct task\_struct \*p\_opptr,\*p\_pptr, \*p\_cptr, \*p\_ysptr, \*p\_osptr;

struct wait\_queue \*wait\_chldexit; /\* for wait4() \*/

/\*

\* For ease of programming... Normal sleeps don't need to

\* keep track of a wait-queue: every task has an entry of its own

\*/

unsigned short uid,euid,suid;

unsigned short gid,egid,sgid;

unsigned long timeout;

unsigned long it\_real\_value, it\_prof\_value, it\_virt\_value;

unsigned long it\_real\_incr, it\_prof\_incr, it\_virt\_incr;

long utime,stime,cutime,cstime,start\_time;

unsigned long min\_flt, maj\_flt;

unsigned long cmin\_flt, cmaj\_flt;

struct rlimit rlim[RLIM\_NLIMITS];

unsigned short used\_math;

unsigned short rss; /\* number of resident pages \*/

char comm[16];

struct vm86\_struct \* vm86\_info;

unsigned long screen\_bitmap;

/\* file system info \*/

int link\_count;

int tty; /\* -1 if no tty, so it must be signed \*/

unsigned short umask;

struct inode \* pwd;

struct inode \* root;

struct inode \* executable;

struct vm\_area\_struct \* mmap;

struct shm\_desc \*shm;

struct sem\_undo \*semun;

struct file \* filp[NR\_OPEN];

fd\_set close\_on\_exec;

/\* ldt for this task - used by Wine. If NULL, default\_ldt is used \*/

struct desc\_struct \*ldt;

/\* tss for this task \*/

struct tss\_struct tss;

#ifdef NEW\_SWAP

unsigned long old\_maj\_flt; /\* old value of maj\_flt \*/

unsigned long dec\_flt; /\* page fault count of the last time \*/

unsigned long swap\_cnt; /\* number of pages to swap on next pass \*/

short swap\_table; /\* current page table \*/

short swap\_page; /\* current page \*/

#endif NEW\_SWAP

struct vm\_area\_struct \*stk\_vma;

};

* 成员struct task\_struct \*next\_task, \*prev\_task可以看出其组织方式为链表。
* struct file \* filp[NR\_OPEN]是一个文件结构体的数组，表示其打开的文件。fd是其在表内的索引。

## 进程表

#define NR\_TASKS 128

struct task\_struct \* task[NR\_TASKS]；

进程表

struct task\_struct \*current；

表示当前正在运行的进程

## 创建进程

asmlinkage int sys\_fork(struct pt\_regs regs)

{

struct pt\_regs \* childregs;

struct task\_struct \*p;

int i,nr;

struct file \*f;

unsigned long clone\_flags = COPYVM | SIGCHLD;

if(!(p = (struct task\_struct\*)\_\_get\_free\_page(GFP\_KERNEL)))

goto bad\_fork;

nr = find\_empty\_process();

if (nr < 0)

goto bad\_fork\_free;

task[nr] = p;

\*p = \*current;

p->did\_exec = 0;

p->kernel\_stack\_page = 0;

p->state = TASK\_UNINTERRUPTIBLE;

p->flags &= ~(PF\_PTRACED|PF\_TRACESYS);

p->pid = last\_pid;

p->swappable = 1;

p->p\_pptr = p->p\_opptr = current;

p->p\_cptr = NULL;

SET\_LINKS(p);

p->signal = 0;

p->it\_real\_value = p->it\_virt\_value = p->it\_prof\_value = 0;

p->it\_real\_incr = p->it\_virt\_incr = p->it\_prof\_incr = 0;

p->leader = 0; /\* process leadership doesn't inherit \*/

p->utime = p->stime = 0;

p->cutime = p->cstime = 0;

p->min\_flt = p->maj\_flt = 0;

p->cmin\_flt = p->cmaj\_flt = 0;

p->start\_time = jiffies;

/\*

\* set up new TSS and kernel stack

\*/

if (!(p->kernel\_stack\_page = \_\_get\_free\_page(GFP\_KERNEL)))

goto bad\_fork\_cleanup;

p->tss.es = KERNEL\_DS;

p->tss.cs = KERNEL\_CS;

p->tss.ss = KERNEL\_DS;

p->tss.ds = KERNEL\_DS;

p->tss.fs = USER\_DS;

p->tss.gs = KERNEL\_DS;

p->tss.ss0 = KERNEL\_DS;

p->tss.esp0 = p->kernel\_stack\_page + PAGE\_SIZE;

p->tss.tr = \_TSS(nr);

childregs = ((struct pt\_regs \*) (p->kernel\_stack\_page + PAGE\_SIZE)) - 1;

p->tss.esp = (unsigned long) childregs;

p->tss.eip = (unsigned long) ret\_from\_sys\_call;

\*childregs = regs;

childregs->eax = 0;

p->tss.back\_link = 0;

p->tss.eflags = regs.eflags & 0xffffcfff; /\* iopl is always 0 for a new process \*/

if (IS\_CLONE) {

if (regs.ebx)

childregs->esp = regs.ebx;

clone\_flags = regs.ecx;

if (childregs->esp == regs.esp)

clone\_flags |= COPYVM;

}

p->exit\_signal = clone\_flags & CSIGNAL;

p->tss.ldt = \_LDT(nr);

if (p->ldt) {

p->ldt = (struct desc\_struct\*) vmalloc(LDT\_ENTRIES\*LDT\_ENTRY\_SIZE);

if (p->ldt != NULL)

memcpy(p->ldt, current->ldt, LDT\_ENTRIES\*LDT\_ENTRY\_SIZE);

}

p->tss.bitmap = offsetof(struct tss\_struct,io\_bitmap);

for (i = 0; i < IO\_BITMAP\_SIZE+1 ; i++) /\* IO bitmap is actually SIZE+1 \*/

p->tss.io\_bitmap[i] = ~0;

if (last\_task\_used\_math == current)

\_\_asm\_\_("clts ; fnsave %0 ; frstor %0":"=m" (p->tss.i387));

p->semun = NULL; p->shm = NULL;

if (copy\_vm(p) || shm\_fork(current, p))

goto bad\_fork\_cleanup;

if (clone\_flags & COPYFD) {

for (i=0; i<NR\_OPEN;i++)

if ((f = p->filp[i]) != NULL)

p->filp[i] = copy\_fd(f);

} else {

for (i=0; i<NR\_OPEN;i++)

if ((f = p->filp[i]) != NULL)

f->f\_count++;

}

if (current->pwd)

current->pwd->i\_count++;

if (current->root)

current->root->i\_count++;

if (current->executable)

current->executable->i\_count++;

dup\_mmap(p);

set\_tss\_desc(gdt+(nr<<1)+FIRST\_TSS\_ENTRY,&(p->tss));

if (p->ldt)

set\_ldt\_desc(gdt+(nr<<1)+FIRST\_LDT\_ENTRY,p->ldt, 512);

else

set\_ldt\_desc(gdt+(nr<<1)+FIRST\_LDT\_ENTRY,&default\_ldt, 1);

p->counter = current->counter >> 1;

p->state = TASK\_RUNNING; /\* do this last, just in case \*/

return p->pid;

bad\_fork\_cleanup:

task[nr] = NULL;

REMOVE\_LINKS(p);

free\_page(p->kernel\_stack\_page);

bad\_fork\_free:

free\_page((long) p);

bad\_fork:

return -EAGAIN;

}

* 他的主要任务是给进程结构体填写数据结构。
* 取得一个空闲页面存放结构体
* 找到一个空闲进程槽
* 为内存堆栈页取得空闲页面
* 拷贝父辈ldt到子进程
* 拷贝父进程mmap信息到子进程。

### 取得一个空闲页面存放结构体

unsigned long \_\_get\_free\_page(int priority)

{

extern unsigned long intr\_count;

unsigned long result, flag;

static unsigned long index = 0;

/\* this routine can be called at interrupt time via

malloc. We want to make sure that the critical

sections of code have interrupts disabled. -RAB

Is this code reentrant? \*/

if (intr\_count && priority != GFP\_ATOMIC) {

printk("gfp called nonatomically from interrupt %08lx\n",

((unsigned long \*)&priority)[-1]);

priority = GFP\_ATOMIC;

}

save\_flags(flag);

repeat:

REMOVE\_FROM\_MEM\_QUEUE(free\_page\_list,nr\_free\_pages);

if (priority == GFP\_BUFFER)

return 0;

if (priority != GFP\_ATOMIC)

if (try\_to\_free\_page())

goto repeat;

REMOVE\_FROM\_MEM\_QUEUE(secondary\_page\_list,nr\_secondary\_pages);

return 0;

}

* 保存task\_struct

### 找到一个空闲的进程槽find\_empty\_process

static int find\_empty\_process(void)

{

int free\_task;

int i, tasks\_free;

int this\_user\_tasks;

repeat:

if ((++last\_pid) & 0xffff8000)

last\_pid=1;

this\_user\_tasks = 0;

tasks\_free = 0;

free\_task = -EAGAIN;

i = NR\_TASKS;

while (--i > 0) {

if (!task[i]) {

free\_task = i;

tasks\_free++;

continue;

}

if (task[i]->uid == current->uid)

this\_user\_tasks++;

if (task[i]->pid == last\_pid || task[i]->pgrp == last\_pid ||

task[i]->session == last\_pid)

goto repeat;

}

if (tasks\_free <= MIN\_TASKS\_LEFT\_FOR\_ROOT ||

this\_user\_tasks > MAX\_TASKS\_PER\_USER)

if (current->uid)

return -EAGAIN;

return free\_task;

}

## 进程退出

### sys\_kill

asmlinkage int sys\_kill(int pid,int sig)

{

int err, retval = 0, count = 0;

if (!pid)

return(kill\_pg(current->pgrp,sig,0));

if (pid == -1) {

struct task\_struct \* p;

for\_each\_task(p) {

if (p->pid > 1 && p != current) {

++count;

if ((err = send\_sig(sig,p,0)) != -EPERM)

retval = err;

}

}

return(count ? retval : -ESRCH);

}

if (pid < 0)

return(kill\_pg(-pid,sig,0));

/\* Normal kill \*/

return(kill\_proc(pid,sig,0));

}

### sys\_exit

asmlinkage int sys\_exit(int error\_code)

{

do\_exit((error\_code&0xff)<<8);

}

#### do\_exit

NORET\_TYPE void do\_exit(long code)

{

struct task\_struct \*p;

int i;

fake\_volatile:

if (current->semun)

sem\_exit();

if (current->shm)

shm\_exit();

free\_page\_tables(current);

for (i=0 ; i<NR\_OPEN ; i++)

if (current->filp[i])

sys\_close(i);

forget\_original\_parent(current);

iput(current->pwd);

current->pwd = NULL;

iput(current->root);

current->root = NULL;

iput(current->executable);

current->executable = NULL;

/\* Release all of the old mmap stuff. \*/

{

struct vm\_area\_struct \* mpnt, \*mpnt1;

mpnt = current->mmap;

current->mmap = NULL;

while (mpnt) {

mpnt1 = mpnt->vm\_next;

if (mpnt->vm\_ops && mpnt->vm\_ops->close)

mpnt->vm\_ops->close(mpnt);

kfree(mpnt);

mpnt = mpnt1;

}

}

if (current->ldt) {

vfree(current->ldt);

current->ldt = NULL;

for (i=1 ; i<NR\_TASKS ; i++) {

if (task[i] == current) {

set\_ldt\_desc(gdt+(i<<1)+FIRST\_LDT\_ENTRY, &default\_ldt, 1);

load\_ldt(i);

}

}

}

current->state = TASK\_ZOMBIE;

current->exit\_code = code;

current->rss = 0;

/\*

\* Check to see if any process groups have become orphaned

\* as a result of our exiting, and if they have any stopped

\* jobs, send them a SIGUP and then a SIGCONT. (POSIX 3.2.2.2)

\*

\* Case i: Our father is in a different pgrp than we are

\* and we were the only connection outside, so our pgrp

\* is about to become orphaned.

\*/

if ((current->p\_pptr->pgrp != current->pgrp) &&

(current->p\_pptr->session == current->session) &&

is\_orphaned\_pgrp(current->pgrp) &&

has\_stopped\_jobs(current->pgrp)) {

kill\_pg(current->pgrp,SIGHUP,1);

kill\_pg(current->pgrp,SIGCONT,1);

}

/\* Let father know we died \*/

notify\_parent(current);

/\*

\* This loop does two things:

\*

\* A. Make init inherit all the child processes

\* B. Check to see if any process groups have become orphaned

\* as a result of our exiting, and if they have any stopped

\* jobs, send them a SIGHUP and then a SIGCONT. (POSIX 3.2.2.2)

\*/

while ((p = current->p\_cptr) != NULL) {

current->p\_cptr = p->p\_osptr;

p->p\_ysptr = NULL;

p->flags &= ~(PF\_PTRACED|PF\_TRACESYS);

if (task[1] && task[1] != current)

p->p\_pptr = task[1];

else

p->p\_pptr = task[0];

p->p\_osptr = p->p\_pptr->p\_cptr;

p->p\_osptr->p\_ysptr = p;

p->p\_pptr->p\_cptr = p;

if (p->state == TASK\_ZOMBIE)

notify\_parent(p);

/\*

\* process group orphan check

\* Case ii: Our child is in a different pgrp

\* than we are, and it was the only connection

\* outside, so the child pgrp is now orphaned.

\*/

if ((p->pgrp != current->pgrp) &&

(p->session == current->session) &&

is\_orphaned\_pgrp(p->pgrp) &&

has\_stopped\_jobs(p->pgrp)) {

kill\_pg(p->pgrp,SIGHUP,1);

kill\_pg(p->pgrp,SIGCONT,1);

}

}

if (current->leader)

disassociate\_ctty(1);

if (last\_task\_used\_math == current)

last\_task\_used\_math = NULL;

#ifdef DEBUG\_PROC\_TREE

audit\_ptree();

#endif

schedule();

/\*

\* In order to get rid of the "volatile function does return" message

\* I did this little loop that confuses gcc to think do\_exit really

\* is volatile. In fact it's schedule() that is volatile in some

\* circumstances: when current->state = ZOMBIE, schedule() never

\* returns.

\*

\* In fact the natural way to do all this is to have the label and the

\* goto right after each other, but I put the fake\_volatile label at

\* the start of the function just in case something /really/ bad

\* happens, and the schedule returns. This way we can try again. I'm

\* not paranoid: it's just that everybody is out to get me.

\*/

goto fake\_volatile;

}

### sys\_waitpid

asmlinkage int sys\_waitpid(pid\_t pid,unsigned long \* stat\_addr, int options)

{

return sys\_wait4(pid, stat\_addr, options, NULL);

}

#### sys\_wait4

asmlinkage int sys\_wait4(pid\_t pid,unsigned long \* stat\_addr, int options, struct rusage \* ru)

{

int flag, retval;

struct wait\_queue wait = { current, NULL };

struct task\_struct \*p;

if (stat\_addr) {

flag = verify\_area(VERIFY\_WRITE, stat\_addr, 4);

if (flag)

return flag;

}

add\_wait\_queue(&current->wait\_chldexit,&wait);

repeat:

flag=0;

for (p = current->p\_cptr ; p ; p = p->p\_osptr) {

if (pid>0) {

if (p->pid != pid)

continue;

} else if (!pid) {

if (p->pgrp != current->pgrp)

continue;

} else if (pid != -1) {

if (p->pgrp != -pid)

continue;

}

/\* wait for cloned processes iff the \_\_WCLONE flag is set \*/

if ((p->exit\_signal != SIGCHLD) ^ ((options & \_\_WCLONE) != 0))

continue;

flag = 1;

switch (p->state) {

case TASK\_STOPPED:

if (!p->exit\_code)

continue;

if (!(options & WUNTRACED) && !(p->flags & PF\_PTRACED))

continue;

if (stat\_addr)

put\_fs\_long((p->exit\_code << 8) | 0x7f,

stat\_addr);

p->exit\_code = 0;

if (ru != NULL)

getrusage(p, RUSAGE\_BOTH, ru);

retval = p->pid;

goto end\_wait4;

case TASK\_ZOMBIE:

current->cutime += p->utime + p->cutime;

current->cstime += p->stime + p->cstime;

current->cmin\_flt += p->min\_flt + p->cmin\_flt;

current->cmaj\_flt += p->maj\_flt + p->cmaj\_flt;

if (ru != NULL)

getrusage(p, RUSAGE\_BOTH, ru);

flag = p->pid;

if (stat\_addr)

put\_fs\_long(p->exit\_code, stat\_addr);

if (p->p\_opptr != p->p\_pptr) {

REMOVE\_LINKS(p);

p->p\_pptr = p->p\_opptr;

SET\_LINKS(p);

notify\_parent(p);

} else

release(p);

#ifdef DEBUG\_PROC\_TREE

audit\_ptree();

#endif

retval = flag;

goto end\_wait4;

default:

continue;

}

}

if (flag) {

retval = 0;

if (options & WNOHANG)

goto end\_wait4;

current->state=TASK\_INTERRUPTIBLE;

schedule();

current->signal &= ~(1<<(SIGCHLD-1));

retval = -ERESTARTSYS;

if (current->signal & ~current->blocked)

goto end\_wait4;

goto repeat;

}

retval = -ECHILD;

end\_wait4:

remove\_wait\_queue(&current->wait\_chldexit,&wait);

return retval;

}

## 执行

exec系统定位该执行文件的二进制镜像，加载并执行它。

### sys\_execve

asmlinkage int sys\_execve(struct pt\_regs regs)

{

int error;

char \* filename;

error = getname((char \*) regs.ebx, &filename);

if (error)

return error;

error = do\_execve(filename, (char \*\*) regs.ecx, (char \*\*) regs.edx, &regs);

putname(filename);

return error;

}

#### do\_execve

static int do\_execve(char \* filename, char \*\* argv, char \*\* envp, struct pt\_regs \* regs)

{

struct linux\_binprm bprm;

struct linux\_binfmt \* fmt;

unsigned long old\_fs;

int i;

int retval;

int sh\_bang = 0;

if (regs->cs != USER\_CS)

return -EINVAL;

bprm.p = PAGE\_SIZE\*MAX\_ARG\_PAGES-4;

for (i=0 ; i<MAX\_ARG\_PAGES ; i++) /\* clear page-table \*/

bprm.page[i] = 0;

retval = open\_namei(filename, 0, 0, &bprm.inode, NULL);

if (retval)

return retval;

bprm.filename = filename;

bprm.argc = count(argv);

bprm.envc = count(envp);

restart\_interp:

if (!S\_ISREG(bprm.inode->i\_mode)) { /\* must be regular file \*/

retval = -EACCES;

goto exec\_error2;

}

if (IS\_NOEXEC(bprm.inode)) { /\* FS mustn't be mounted noexec \*/

retval = -EPERM;

goto exec\_error2;

}

if (!bprm.inode->i\_sb) {

retval = -EACCES;

goto exec\_error2;

}

i = bprm.inode->i\_mode;

if (IS\_NOSUID(bprm.inode) && (((i & S\_ISUID) && bprm.inode->i\_uid != current->

euid) || ((i & S\_ISGID) && !in\_group\_p(bprm.inode->i\_gid))) &&

!suser()) {

retval = -EPERM;

goto exec\_error2;

}

/\* make sure we don't let suid, sgid files be ptraced. \*/

if (current->flags & PF\_PTRACED) {

bprm.e\_uid = current->euid;

bprm.e\_gid = current->egid;

} else {

bprm.e\_uid = (i & S\_ISUID) ? bprm.inode->i\_uid : current->euid;

bprm.e\_gid = (i & S\_ISGID) ? bprm.inode->i\_gid : current->egid;

}

if (current->euid == bprm.inode->i\_uid)

i >>= 6;

else if (in\_group\_p(bprm.inode->i\_gid))

i >>= 3;

if (!(i & 1) &&

!((bprm.inode->i\_mode & 0111) && suser())) {

retval = -EACCES;

goto exec\_error2;

}

memset(bprm.buf,0,sizeof(bprm.buf));

old\_fs = get\_fs();

set\_fs(get\_ds());

retval = read\_exec(bprm.inode,0,bprm.buf,128);

set\_fs(old\_fs);

if (retval < 0)

goto exec\_error2;

if ((bprm.buf[0] == '#') && (bprm.buf[1] == '!') && (!sh\_bang)) {

/\*

\* This section does the #! interpretation.

\* Sorta complicated, but hopefully it will work. -TYT

\*/

char \*cp, \*interp, \*i\_name, \*i\_arg;

iput(bprm.inode);

bprm.buf[127] = '\0';

if ((cp = strchr(bprm.buf, '\n')) == NULL)

cp = bprm.buf+127;

\*cp = '\0';

while (cp > bprm.buf) {

cp--;

if ((\*cp == ' ') || (\*cp == '\t'))

\*cp = '\0';

else

break;

}

for (cp = bprm.buf+2; (\*cp == ' ') || (\*cp == '\t'); cp++);

if (!cp || \*cp == '\0') {

retval = -ENOEXEC; /\* No interpreter name found \*/

goto exec\_error1;

}

interp = i\_name = cp;

i\_arg = 0;

for ( ; \*cp && (\*cp != ' ') && (\*cp != '\t'); cp++) {

if (\*cp == '/')

i\_name = cp+1;

}

while ((\*cp == ' ') || (\*cp == '\t'))

\*cp++ = '\0';

if (\*cp)

i\_arg = cp;

/\*

\* OK, we've parsed out the interpreter name and

\* (optional) argument.

\*/

if (sh\_bang++ == 0) {

bprm.p = copy\_strings(bprm.envc, envp, bprm.page, bprm.p, 0);

bprm.p = copy\_strings(--bprm.argc, argv+1, bprm.page, bprm.p, 0);

}

/\*

\* Splice in (1) the interpreter's name for argv[0]

\* (2) (optional) argument to interpreter

\* (3) filename of shell script

\*

\* This is done in reverse order, because of how the

\* user environment and arguments are stored.

\*/

bprm.p = copy\_strings(1, &bprm.filename, bprm.page, bprm.p, 2);

bprm.argc++;

if (i\_arg) {

bprm.p = copy\_strings(1, &i\_arg, bprm.page, bprm.p, 2);

bprm.argc++;

}

bprm.p = copy\_strings(1, &i\_name, bprm.page, bprm.p, 2);

bprm.argc++;

if (!bprm.p) {

retval = -E2BIG;

goto exec\_error1;

}

/\*

\* OK, now restart the process with the interpreter's inode.

\* Note that we use open\_namei() as the name is now in kernel

\* space, and we don't need to copy it.

\*/

retval = open\_namei(interp, 0, 0, &bprm.inode, NULL);

if (retval)

goto exec\_error1;

goto restart\_interp;

}

if (!sh\_bang) {

bprm.p = copy\_strings(bprm.envc,envp,bprm.page,bprm.p,0);

bprm.p = copy\_strings(bprm.argc,argv,bprm.page,bprm.p,0);

if (!bprm.p) {

retval = -E2BIG;

goto exec\_error2;

}

}

bprm.sh\_bang = sh\_bang;

fmt = formats;

do {

int (\*fn)(struct linux\_binprm \*, struct pt\_regs \*) = fmt->load\_binary;

if (!fn)

break;

retval = fn(&bprm, regs);

if (retval == 0) {

iput(bprm.inode);

current->did\_exec = 1;

return 0;

}

fmt++;

} while (retval == -ENOEXEC);

exec\_error2:

iput(bprm.inode);

exec\_error1:

for (i=0 ; i<MAX\_ARG\_PAGES ; i++)

free\_page(bprm.page[i]);

return(retval);

}

# 内存管理

实现为虚拟内存和物理内存，虚拟内存通过MMU，唯一映射到一块物理内存。若数据不在物理内存（被交换出去了），则产生缺页中断，将需要的数据换入。

## 虚拟内存

### vm\_struct

struct vm\_struct {

unsigned long flags;

void \* addr;

unsigned long size;

struct vm\_struct \* next;

};

虚拟内存结构体，包含：

* 标志位
* 虚拟内存地址
* 内存块大小
* 指向下一个结构体的指着（单链表）

### 页面

struct page\_descriptor {

struct page\_descriptor \*next;

struct block\_header \*firstfree;

int order;

int nfree;

};

## 物理内存

### block\_header

struct **block\_header** {

unsigned long bh\_flags;

union {

unsigned long ubh\_length;

struct block\_header \*fbh\_next;

} vp;

};

### 快速适配

struct size\_descriptor {

struct page\_descriptor \*firstfree;

int size;

int nblocks;

int nmallocs;

int nfrees;

int nbytesmalloced;

int npages;

};

struct size\_descriptor sizes[] = {

{ NULL, 32,127, 0,0,0,0 },

{ NULL, 64, 63, 0,0,0,0 },

{ NULL, 128, 31, 0,0,0,0 },

{ NULL, 252, 16, 0,0,0,0 },

{ NULL, 508, 8, 0,0,0,0 },

{ NULL,1020, 4, 0,0,0,0 },

{ NULL,2040, 2, 0,0,0,0 },

{ NULL,4080, 1, 0,0,0,0 },

{ NULL, 0, 0, 0,0,0,0 }

};

为常用大小的空闲区维护单独的链表。

### 交换

static struct swap\_info\_struct {

unsigned long flags;

struct inode \* swap\_file;

unsigned int swap\_device;

unsigned char \* swap\_map;

unsigned char \* swap\_lockmap;

int pages;

int lowest\_bit;

int highest\_bit;

unsigned long max;

} swap\_info[MAX\_SWAPFILES];

内存数据和磁盘块的一一映射。

# 文件系统

将上层虚拟文件系统的操作，映射到物理介质。

## 虚拟文件系统

Linu提供一个标准层，使得能在内核和每种fs之间交换数据。这个接口层成为虚拟文件系统。

上层关于内核表格的管理和数据结构，下层是与各具体文件系统相关的函数集合。

虚拟文件系统由层次较高的操作组成，实际在做事的是每种文件系统的函数。

### file\_operations

struct file\_operations {

int (\*lseek) (struct inode \*, struct file \*, off\_t, int);

int (\*read) (struct inode \*, struct file \*, char \*, int);

int (\*write) (struct inode \*, struct file \*, char \*, int);

int (\*readdir) (struct inode \*, struct file \*, struct dirent \*, int);

int (\*select) (struct inode \*, struct file \*, int, select\_table \*);

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

int (\*mmap) (struct inode \*, struct file \*, unsigned long, size\_t, int, unsigned long);

int (\*open) (struct inode \*, struct file \*);

void (\*release) (struct inode \*, struct file \*);

int (\*fsync) (struct inode \*, struct file \*);

};

包含文件操作函数的指针，指向具体文件系统的函数。

### inode\_operations

struct inode\_operations {

struct file\_operations \* default\_file\_ops;

int (\*create) (struct inode \*,const char \*,int,int,struct inode \*\*);

int (\*lookup) (struct inode \*,const char \*,int,struct inode \*\*);

int (\*link) (struct inode \*,struct inode \*,const char \*,int);

int (\*unlink) (struct inode \*,const char \*,int);

int (\*symlink) (struct inode \*,const char \*,int,const char \*);

int (\*mkdir) (struct inode \*,const char \*,int,int);

int (\*rmdir) (struct inode \*,const char \*,int);

int (\*mknod) (struct inode \*,const char \*,int,int,int);

int (\*rename) (struct inode \*,const char \*,int,struct inode \*,const char \*,int);

int (\*readlink) (struct inode \*,char \*,int);

int (\*follow\_link) (struct inode \*,struct inode \*,int,int,struct inode \*\*);

int (\*bmap) (struct inode \*,int);

void (\*truncate) (struct inode \*);

int (\*permission) (struct inode \*, int);

};

包含inode操作函数的指针，指向具体文件系统的函数。

### super\_operations

struct super\_operations {

void (\*read\_inode) (struct inode \*);

int (\*notify\_change) (int flags, struct inode \*);

void (\*write\_inode) (struct inode \*);

void (\*put\_inode) (struct inode \*);

void (\*put\_super) (struct super\_block \*);

void (\*write\_super) (struct super\_block \*);

void (\*statfs) (struct super\_block \*, struct statfs \*);

int (\*remount\_fs) (struct super\_block \*, int \*, char \*);

};

包含超级块操作函数的指针，指向具体文件系统的函数

### file\_system\_type

struct file\_system\_type {

struct super\_block \*(\*read\_super) (struct super\_block \*, void \*, int);

char \*name;

int requires\_dev;

};

#### 文件系统数组

struct file\_system\_type file\_systems[]

现已转化为链表，以允许加载新的文件系统类型。

### 职责

* 管理缓冲区寄存器
* fcntl,iocntl
* pipe,fifo
* 管理文件和inode表
* 文件锁
* 将文件名（目录项）映射到inode
* select函数
* 提供信息——stat函数
* 加载和卸载文件系统
* exec函数加载二进制程序

## 超级块super\_block

一个超级块对应一个文件系统

struct super\_block {

dev\_t s\_dev;

unsigned long s\_blocksize;

unsigned char s\_blocksize\_bits;

unsigned char s\_lock;

unsigned char s\_rd\_only;

unsigned char s\_dirt;

struct super\_operations \*s\_op;

unsigned long s\_flags;

unsigned long s\_magic;

unsigned long s\_time;

struct inode \* s\_covered;

struct inode \* s\_mounted;

struct wait\_queue \* s\_wait;

union {

struct minix\_sb\_info minix\_sb;

struct ext\_sb\_info ext\_sb;

struct ext2\_sb\_info ext2\_sb;

struct hpfs\_sb\_info hpfs\_sb;

struct msdos\_sb\_info msdos\_sb;

struct isofs\_sb\_info isofs\_sb;

struct nfs\_sb\_info nfs\_sb;

struct xiafs\_sb\_info xiafs\_sb;

struct sysv\_sb\_info sysv\_sb;

} u;

};

* 超级块代表了整个文件系统，超级块是文件系统的控制块，有整个文件系统信息，一个文件系统所有的inode都要连接到超级块上，可以说，一个超级块就代表了一个文件系统。
* s\_magic区别于其他文件系统的标识
* s\_covered应该表示其覆盖的i节点，s\_mounted应该表示挂载的i节点。
* s\_op表示对其进行操作的函数集

struct super\_operations {

void (\*read\_inode) (struct inode \*);

int (\*notify\_change) (int flags, struct inode \*);

void (\*write\_inode) (struct inode \*);

void (\*put\_inode) (struct inode \*);

void (\*put\_super) (struct super\_block \*);

void (\*write\_super) (struct super\_block \*);

void (\*statfs) (struct super\_block \*, struct statfs \*);

int (\*remount\_fs) (struct super\_block \*, int \*, char \*);

};

包括读i节点集，写inode,写super\_block，监听inode变化等。

## 索引节点inode

保存元数据，映射一块磁盘区块。

struct inode {

dev\_t i\_dev;

unsigned long i\_ino;

umode\_t i\_mode;

nlink\_t i\_nlink;

uid\_t i\_uid;

gid\_t i\_gid;

dev\_t i\_rdev;

off\_t i\_size;

time\_t i\_atime;

time\_t i\_mtime;

time\_t i\_ctime;

unsigned long i\_blksize;

unsigned long i\_blocks;

struct semaphore i\_sem;

struct inode\_operations \* i\_op;

struct super\_block \* i\_sb;

struct wait\_queue \* i\_wait;

struct file\_lock \* i\_flock;

struct vm\_area\_struct \* i\_mmap;

struct inode \* i\_next, \* i\_prev;

struct inode \* i\_hash\_next, \* i\_hash\_prev;

struct inode \* i\_bound\_to, \* i\_bound\_by;

struct inode \* i\_mount;

struct socket \* i\_socket;

unsigned short i\_count;

unsigned short i\_flags;

unsigned char i\_lock;

unsigned char i\_dirt;

unsigned char i\_pipe;

unsigned char i\_seek;

unsigned char i\_update;

union {

struct pipe\_inode\_info pipe\_i;

struct minix\_inode\_info minix\_i;

struct ext\_inode\_info ext\_i;

struct ext2\_inode\_info ext2\_i;

struct hpfs\_inode\_info hpfs\_i;

struct msdos\_inode\_info msdos\_i;

struct iso\_inode\_info isofs\_i;

struct nfs\_inode\_info nfs\_i;

struct xiafs\_inode\_info xiafs\_i;

struct sysv\_inode\_info sysv\_i;

} u;

};

* 保存的其实是实际的数据的一些信息，这些信息称为“元数据”(也就是对文件属性的描述)。例如：文件大小，设备标识符，用户标识符，用户组标识符，文件模式，扩展属性，文件读取或修改的时间戳，链接数量，指向存储该内容的磁盘区块的指针，文件分类等等。( 注意数据分成：元数据+数据本身 )
* 同时注意：inode有两种，一种是虚拟文件系统的inode，一种是具体文件系统的inode。前者在内存中，后者在磁盘中。所以每次其实是将磁盘中的inode调进填充内存中的inode，这样才是算使用了磁盘文件inode。
* 注意inode怎样生成的：每个inode节点的大小，一般是128字节或256字节。inode节点的总数，在格式化时就给定(现代OS可以动态变化)，一般每2KB就设置一个inode。一般文件系统中很少有文件小于2KB的，所以预定按照2KB分，一般inode是用不完的。所以inode在文件系统安装的时候会有一个默认数量，后期会根据实际的需要发生变化。
* 注意inode号：inode号是唯一的，表示不同的文件。其实在Linux内部的时候，访问文件都是通过inode号来进行的，所谓文件名仅仅是给用户容易使用的。当我们打开一个文件的时候，首先，系统找到这个文件名对应的inode号；然后，通过inode号，得到inode信息，最后，由inode找到文件数据所在的block，现在可以处理文件数据了。
* inode和文件的关系：当创建一个文件的时候，就给文件分配了一个inode。一个inode只对应一个实际文件，一个文件也会只有一个inode。inodes最大数量就是文件的最大数量。
* 保存文件的管理信息（文件属性），在内核通过inoe\*访问。
* 进程通过fd找到struct file\*，再通过struct file\*找到struct iode\*
* 包含i\_dev文件的设备号，i\_ino i节点号，i\_rdev（若设备文件）的设备编号
* 包含ls -l能查到的：i\_mode权限和文件类型，i\_nlink硬连接计数，i\_gid所属组ID，i\_size文件字节大小，i\_atime访问时间，i\_mtime修改时间，i\_ctime属性修改时间，i\_blksize块大小，i\_blocks块数。
* 包含i\_sem信号量，i\_flock文件锁（应该用于互斥读写该文件），i\_mmap共享内存。
* i\_count引用计数
* 一系列inode\*实现数据结构（或许是链表？）
* i\_op是对inode节点操作函数的函数指针集合。

struct inode\_operations {

struct file\_operations \* default\_file\_ops;

int (\*create) (struct inode \*,const char \*,int,int,struct inode \*\*);

int (\*lookup) (struct inode \*,const char \*,int,struct inode \*\*);

int (\*link) (struct inode \*,struct inode \*,const char \*,int);

int (\*unlink) (struct inode \*,const char \*,int);

int (\*symlink) (struct inode \*,const char \*,int,const char \*);

int (\*mkdir) (struct inode \*,const char \*,int,int);

int (\*rmdir) (struct inode \*,const char \*,int);

int (\*mknod) (struct inode \*,const char \*,int,int,int);

int (\*rename) (struct inode \*,const char \*,int,struct inode \*,const char \*,int);

int (\*readlink) (struct inode \*,char \*,int);

int (\*follow\_link) (struct inode \*,struct inode \*,int,int,struct inode \*\*);

int (\*bmap) (struct inode \*,int);

void (\*truncate) (struct inode \*);

int (\*permission) (struct inode \*, int);

};

* 包括创建、创建硬链接、断开硬链接、创建符号链接、指定文件大小、访问权限检查、读符号链接等。
* 还有些感觉像目录和目录项的函数：mkdir,rmdir,rename等。

## 文件对象file

进程打开的文件对象

struct file {

mode\_t f\_mode;

dev\_t f\_rdev; /\* needed for /dev/tty \*/

off\_t f\_pos;

unsigned short f\_flags;

unsigned short f\_count;

unsigned short f\_reada;

struct file \*f\_next, \*f\_prev;

struct inode \* f\_inode;

struct file\_operations \* f\_op;

};

* 与进程关联。进程通过PCB的文件描述符表的表值file\*，找到该结构体。
* 这个结构体与进程对应，表示该进程打开的文件。
* f\_rdev表示其设备号，f\_mode表示文件打开标志，f\_pos表示当前文件偏移量。
* F\_next和f\_prev应该是其数据组织方式：链表。f\_inode是其指向的i节点指针。
* f\_op是对file结构体及其inode节点操作函数的函数指针集合。

struct file\_operations {

int (\*lseek) (struct inode \*, struct file \*, off\_t, int);

int (\*read) (struct inode \*, struct file \*, char \*, int);

int (\*write) (struct inode \*, struct file \*, char \*, int);

int (\*readdir) (struct inode \*, struct file \*, struct dirent \*, int);

int (\*select) (struct inode \*, struct file \*, int, select\_table \*);

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

int (\*mmap) (struct inode \*, struct file \*, unsigned long, size\_t, int, unsigned long);

int (\*open) (struct inode \*, struct file \*);

void (\*release) (struct inode \*, struct file \*);

int (\*fsync) (struct inode \*, struct file \*);

};

包括读写文件，读目录，打开和释放文件，mmap，ioctl等。

## 挂载卸载

Linux无论几个分区，他只有一个根目录。他的每个分区都是用来组成整个文件系统的一部分，他采用载入的方法将分区和文件系统关联。

一个文件系统中包含一整套目录和文件，将一个目录如/mnt/Space和一个分区联系起来。

### sys\_mount

asmlinkage int sys\_mount(char \* dev\_name, char \* dir\_name, char \* type,

unsigned long new\_flags, void \* data)

{

struct file\_system\_type \* fstype;

struct inode \* inode;

struct file\_operations \* fops;

dev\_t dev;

int retval;

char \* t;

unsigned long flags = 0;

unsigned long page = 0;

if (!suser())

return -EPERM;

if ((new\_flags &

(MS\_MGC\_MSK | MS\_REMOUNT)) == (MS\_MGC\_VAL | MS\_REMOUNT)) {

retval = copy\_mount\_options (data, &page);

if (retval < 0)

return retval;

retval = do\_remount(dir\_name,

new\_flags & ~MS\_MGC\_MSK & ~MS\_REMOUNT,

(char \*) page);

free\_page(page);

return retval;

}

retval = copy\_mount\_options (type, &page);

if (retval < 0)

return retval;

fstype = get\_fs\_type((char \*) page);

free\_page(page);

if (!fstype)

return -ENODEV;

t = fstype->name;

if (fstype->requires\_dev) {

retval = namei(dev\_name,&inode);

if (retval)

return retval;

if (!S\_ISBLK(inode->i\_mode)) {

iput(inode);

return -ENOTBLK;

}

if (IS\_NODEV(inode)) {

iput(inode);

return -EACCES;

}

dev = inode->i\_rdev;

if (MAJOR(dev) >= MAX\_BLKDEV) {

iput(inode);

return -ENXIO;

}

} else {

if (!(dev = get\_unnamed\_dev()))

return -EMFILE;

inode = NULL;

}

fops = get\_blkfops(MAJOR(dev));

if (fops && fops->open) {

retval = fops->open(inode,NULL);

if (retval) {

iput(inode);

return retval;

}

}

page = 0;

if ((new\_flags & MS\_MGC\_MSK) == MS\_MGC\_VAL) {

flags = new\_flags & ~MS\_MGC\_MSK;

retval = copy\_mount\_options(data, &page);

if (retval < 0) {

iput(inode);

return retval;

}

}

retval = do\_mount(dev,dir\_name,t,flags,(void \*) page);

free\_page(page);

if (retval && fops && fops->release)

fops->release(inode,NULL);

iput(inode);

return retval;

}

## 具体文件系统

struct file\_system\_type {

struct super\_block \*(\*read\_super) (struct super\_block \*, void \*, int);

char \*name;

int requires\_dev;

};

文件系统结构体：read\_super函数负责填写超级块；name表示文件系统名称。

### 组成

* filesystem.c的file\_systems[]的一个项
* 超级块头文件type\_fs\_sb.h（例如minix\_fs\_sb.h）
* i节点头文件type\_fs\_i.h（例如minix\_fs\_i.h）
* 普通的自己专用的头文件type\_fs.h（例如minix\_fs.h）

### 具体

#### minix

struct minix\_sb\_info {

unsigned long s\_ninodes;

unsigned long s\_nzones;

unsigned long s\_imap\_blocks;

unsigned long s\_zmap\_blocks;

unsigned long s\_firstdatazone;

unsigned long s\_log\_zone\_size;

unsigned long s\_max\_size;

struct buffer\_head \* s\_imap[8];

struct buffer\_head \* s\_zmap[8];

unsigned long s\_dirsize;

unsigned long s\_namelen;

struct buffer\_head \* s\_sbh;

struct minix\_super\_block \* s\_ms;

unsigned short s\_mount\_state;

};

#### extended-fs

struct ext\_sb\_info {

unsigned long s\_ninodes;

unsigned long s\_nzones;

unsigned long s\_firstdatazone;

unsigned long s\_log\_zone\_size;

unsigned long s\_max\_size;

unsigned long s\_firstfreeblocknumber;

unsigned long s\_freeblockscount;

struct buffer\_head \* s\_firstfreeblock;

unsigned long s\_firstfreeinodenumber;

unsigned long s\_freeinodescount;

struct buffer\_head \* s\_firstfreeinodeblock;

};

#### ext2

struct ext2\_sb\_info {

unsigned long s\_frag\_size; /\* Size of a fragment in bytes \*/

unsigned long s\_frags\_per\_block;/\* Number of fragments per block \*/

unsigned long s\_inodes\_per\_block;/\* Number of inodes per block \*/

unsigned long s\_frags\_per\_group;/\* Number of fragments in a group \*/

unsigned long s\_blocks\_per\_group;/\* Number of blocks in a group \*/

unsigned long s\_inodes\_per\_group;/\* Number of inodes in a group \*/

unsigned long s\_itb\_per\_group; /\* Number of inode table blocks per group \*/

unsigned long s\_desc\_per\_block; /\* Number of group descriptors per block \*/

unsigned long s\_groups\_count; /\* Number of groups in the fs \*/

struct buffer\_head \* s\_sbh; /\* Buffer containing the super block \*/

struct ext2\_super\_block \* s\_es; /\* Pointer to the super block in the buffer \*/

struct buffer\_head \* s\_group\_desc[EXT2\_MAX\_GROUP\_DESC];

unsigned short s\_loaded\_inode\_bitmaps;

unsigned short s\_loaded\_block\_bitmaps;

unsigned long s\_inode\_bitmap\_number[EXT2\_MAX\_GROUP\_LOADED];

struct buffer\_head \* s\_inode\_bitmap[EXT2\_MAX\_GROUP\_LOADED];

unsigned long s\_block\_bitmap\_number[EXT2\_MAX\_GROUP\_LOADED];

struct buffer\_head \* s\_block\_bitmap[EXT2\_MAX\_GROUP\_LOADED];

int s\_rename\_lock;

struct wait\_queue \* s\_rename\_wait;

unsigned long s\_mount\_opt;

unsigned short s\_mount\_state;

};

#### hpfs

struct hpfs\_sb\_info {

ino\_t sb\_root; /\* inode number of root dir \*/

unsigned sb\_fs\_size; /\* file system size, sectors \*/

unsigned sb\_bitmaps; /\* sector number of bitmap list \*/

unsigned sb\_dirband\_size; /\* directory band size, dnodes \*/

unsigned sb\_dmap; /\* sector number of dnode bit map \*/

unsigned sb\_n\_free; /\* free blocks for statfs, or -1 \*/

unsigned sb\_n\_free\_dnodes; /\* free dnodes for statfs, or -1 \*/

uid\_t sb\_uid; /\* uid from mount options \*/

gid\_t sb\_gid; /\* gid from mount options \*/

umode\_t sb\_mode; /\* mode from mount options \*/

unsigned sb\_lowercase : 1; /\* downcase filenames hackery \*/

unsigned sb\_conv : 2; /\* crlf->newline hackery \*/

};

#### MS-DOS

struct msdos\_sb\_info {

unsigned short cluster\_size; /\* sectors/cluster \*/

unsigned char fats,fat\_bits; /\* number of FATs, FAT bits (12 or 16) \*/

unsigned short fat\_start,fat\_length; /\* FAT start & length (sec.) \*/

unsigned short dir\_start,dir\_entries; /\* root dir start & entries \*/

unsigned short data\_start; /\* first data sector \*/

unsigned long clusters; /\* number of clusters \*/

uid\_t fs\_uid;

gid\_t fs\_gid;

int quiet; /\* fake successful chmods and chowns \*/

unsigned short fs\_umask;

unsigned char name\_check; /\* r = relaxed, n = normal, s = strict \*/

unsigned char conversion; /\* b = binary, t = text, a = auto \*/

struct wait\_queue \*fat\_wait;

int fat\_lock;

int prev\_free; /\* previously returned free cluster number \*/

int free\_clusters; /\* -1 if undefined \*/

};

#### isofs

struct isofs\_sb\_info {

unsigned long s\_ninodes;

unsigned long s\_nzones;

unsigned long s\_firstdatazone;

unsigned long s\_log\_zone\_size;

unsigned long s\_max\_size;

unsigned char s\_high\_sierra; /\* A simple flag \*/

unsigned char s\_mapping;

unsigned char s\_conversion;

unsigned char s\_rock;

unsigned char s\_cruft; /\* Broken disks with high

byte of length containing

junk \*/

};

#### nfs

struct nfs\_sb\_info {

struct nfs\_server s\_server;

struct nfs\_fh s\_root;

};

struct nfs\_server {

struct file \*file;

int lock;

struct wait\_queue \*wait;

int flags;

int rsize;

int wsize;

int timeo;

int retrans;

int acregmin;

int acregmax;

int acdirmin;

int acdirmax;

char hostname[256];

};

struct nfs\_fh {

char data[NFS\_FHSIZE];

};

#### xiafs

struct xiafs\_sb\_info {

u\_long s\_nzones;

u\_long s\_ninodes;

u\_long s\_ndatazones;

u\_long s\_imap\_zones;

u\_long s\_zmap\_zones;

u\_long s\_firstdatazone;

u\_long s\_zone\_shift;

u\_long s\_max\_size; /\* 32 bytes \*/

struct buffer\_head \* s\_imap\_buf[\_XIAFS\_IMAP\_SLOTS]; /\* 32 bytes \*/

struct buffer\_head \* s\_zmap\_buf[\_XIAFS\_ZMAP\_SLOTS]; /\* 128 bytes \*/

int s\_imap\_iznr[\_XIAFS\_IMAP\_SLOTS]; /\* 32 bytes \*/

int s\_zmap\_zznr[\_XIAFS\_ZMAP\_SLOTS]; /\* 128 bytes \*/

u\_char s\_imap\_cached; /\* flag for cached imap \*/

u\_char s\_zmap\_cached; /\* flag for cached imap \*/

};

#### SystemV/Coherent

struct sysv\_sb\_info {

int s\_type; /\* file system type: FSTYPE\_{XENIX|SYSV|COH} \*/

unsigned int s\_block\_size; /\* zone size, = 512 or = 1024 \*/

unsigned int s\_block\_size\_1; /\* block\_size - 1 \*/

unsigned int s\_block\_size\_bits; /\* log2(block\_size) \*/

unsigned int s\_block\_size\_ratio; /\* BLOCK\_SIZE / block\_size \*/

unsigned int s\_block\_size\_ratio\_1; /\* block\_size\_ratio - 1 \*/

unsigned int s\_block\_size\_ratio\_bits; /\* log2(block\_size\_ratio) \*/

char s\_convert; /\* flag whether byte ordering requires conversion \*/

char s\_kludge\_symlinks; /\* flag whether symlinks have a kludgey mode \*/

char s\_truncate; /\* if 1: names > SYSV\_NAMELEN chars are truncated \*/

/\* if 0: they are disallowed (ENAMETOOLONG) \*/

nlink\_t s\_link\_max; /\* max number of hard links to a file \*/

unsigned int s\_inodes\_per\_block; /\* number of inodes per block \*/

unsigned int s\_inodes\_per\_block\_1; /\* inodes\_per\_block - 1 \*/

unsigned int s\_inodes\_per\_block\_bits; /\* log2(inodes\_per\_block) \*/

unsigned int s\_ind\_per\_block; /\* number of indirections per block \*/

unsigned int s\_ind\_per\_block\_1; /\* ind\_per\_block - 1 \*/

unsigned int s\_ind\_per\_block\_bits; /\* log2(ind\_per\_block) \*/

unsigned int s\_ind\_per\_block\_2; /\* ind\_per\_block ^ 2 \*/

unsigned int s\_ind\_per\_block\_2\_1; /\* ind\_per\_block ^ 2 - 1 \*/

unsigned int s\_ind\_per\_block\_2\_bits; /\* log2(ind\_per\_block^2) \*/

unsigned int s\_ind\_per\_block\_3; /\* ind\_per\_block ^ 3 \*/

unsigned int s\_toobig\_block; /\* 10 + ipb + ipb^2 + ipb^3 \*/

unsigned int s\_block\_base; /\* physical block number of block 0 \*/

unsigned short s\_fic\_size; /\* free inode cache size, NICINOD \*/

unsigned short s\_flc\_size; /\* free block list chunk size, NICFREE \*/

/\* The superblock is kept in a disk buffer: \*/

struct buffer\_head \*s\_bh;

/\* These are pointers into the disk buffer, to compensate for

different superblock layout. \*/

char \* s\_sbd; /\* entire superblock data \*/

unsigned short \*s\_sb\_fic\_count; /\* pointer to s\_sbd->s\_ninode \*/

unsigned short \*s\_sb\_fic\_inodes; /\* pointer to s\_sbd->s\_inode \*/

unsigned short \*s\_sb\_total\_free\_inodes; /\* pointer to s\_sbd->s\_tinode \*/

unsigned short \*s\_sb\_flc\_count; /\* pointer to s\_sbd->s\_nfree \*/

unsigned long \*s\_sb\_flc\_blocks; /\* pointer to s\_sbd->s\_free \*/

unsigned long \*s\_sb\_total\_free\_blocks;/\* pointer to s\_sbd->s\_tfree \*/

unsigned long \*s\_sb\_time; /\* pointer to s\_sbd->s\_time \*/

/\* We keep those superblock entities that don't change here;

this saves us an indirection and perhaps a conversion. \*/

unsigned long s\_firstinodezone; /\* index of first inode zone \*/

unsigned long s\_firstdatazone; /\* same as s\_sbd->s\_isize \*/

unsigned long s\_ninodes; /\* total number of inodes \*/

unsigned long s\_ndatazones; /\* total number of data zones \*/

unsigned long s\_nzones; /\* same as s\_sbd->s\_fsize \*/

};

## 文件锁

struct file\_lock {

struct file\_lock \*fl\_next; /\* singly linked list \*/

struct task\_struct \*fl\_owner; /\* NULL if on free list, for sanity checks \*/

unsigned int fl\_fd; /\* File descriptor for this lock \*/

struct wait\_queue \*fl\_wait;

char fl\_type;

char fl\_whence;

off\_t fl\_start;

off\_t fl\_end;

};

* 实现对文件的互斥访问
* 实现为链表形式，fl\_next指向链表的下一个节点。
* fl\_fd表示锁的文件描述符。
* fl\_type表示锁类型（F\_WRLCK），l\_whence表示从哪里开始锁（SEEK\_SET），l\_start表示开始位置偏移量， fl\_end表示锁到哪里。
* 对应用户区数据结构struct flock（UBuntu 18.04）

struct flock {

short l\_type;

short l\_whence;

off\_t l\_start;

off\_t l\_len;

pid\_t l\_pid;

};

## 函数

### sys\_open打开文件

asmlinkage int sys\_open(const char \* filename,int flags,int mode)

{

char \* tmp;

int error;

error = getname(filename, &tmp);

if (error)

return error;

error = do\_open(tmp,flags,mode);

putname(tmp);

return error;

}

将传入的文件名写入到内核区的内存，然后调用do\_open打开文件。

#### getname

/\*

\* In order to reduce some races, while at the same time doing additional

\* checking and hopefully speeding things up, we copy filenames to the

\* kernel data space before using them..

\*

\* POSIX.1 2.4: an empty pathname is invalid (ENOENT).

\*/

int getname(const char \* filename, char \*\*result)；

做文件名检查和转换，在内核区申请内存以写入文件名，然后将文件名返回。

#### do\_open

/\*

\* Note that while the flag value (low two bits) for sys\_open means:

\* 00 - read-only

\* 01 - write-only

\* 10 - read-write

\* 11 - special

\* it is changed into

\* 00 - no permissions needed

\* 01 - read-permission

\* 10 - write-permission

\* 11 - read-write

\* for the internal routines (ie open\_namei()/follow\_link() etc). 00 is

\* used by symlinks.

\*/

int do\_open(const char \* filename,int flags,int mode)

{

struct inode \* inode;

struct file \* f;

int flag,error,fd;

for(fd=0 ; fd<NR\_OPEN ; fd++)

if (!current->filp[fd])

break;

if (fd>=NR\_OPEN)

return -EMFILE;

FD\_CLR(fd,&current->close\_on\_exec);

f = get\_empty\_filp();

if (!f)

return -ENFILE;

current->filp[fd] = f;

f->f\_flags = flag = flags;

f->f\_mode = (flag+1) & O\_ACCMODE;

if (f->f\_mode)

flag++;

if (flag & (O\_TRUNC | O\_CREAT))

flag |= 2;

error = open\_namei(filename,flag,mode,&inode,NULL);

if (error) {

current->filp[fd]=NULL;

f->f\_count--;

return error;

}

f->f\_inode = inode;

f->f\_pos = 0;

#### open\_namei

\* namei for open - this is in fact almost the whole open-routine.

\*

\* Note that the low bits of "flag" aren't the same as in the open

\* system call - they are 00 - no permissions needed

\* 01 - read permission needed

\* 10 - write permission needed

\* 11 - read/write permissions needed

\* which is a lot more logical, and also allows the "no perm" needed

\* for symlinks (where the permissions are checked later).

\*/

int open\_namei(const char \* pathname, int flag, int mode,

struct inode \*\* res\_inode, struct inode \* base)

{

const char \* basename;

int namelen,error;

struct inode \* dir, \*inode;

。。。

dir\_namei(pathname,&namelen,&basename,base,&dir);

。。。

}

#### dir\_namei

\* dir\_namei() returns the inode of the directory of the

\* specified name, and the name within that directory.

从目录（linux1.0的目录用inode表示）返回文件名所指向的i节点指针。

static int dir\_namei(const char \* pathname, int \* namelen, const char \*\* name,

struct inode \* base, struct inode \*\* res\_inode)

{

char c;

const char \* thisname;

int len,error;

struct inode \* inode;

\*res\_inode = NULL;

if (!base) {

base = current->pwd;

base->i\_count++;

}

if ((c = \*pathname) == '/') {

iput(base);

base = current->root;

pathname++;

base->i\_count++;

}

while (1) {

thisname = pathname;

for(len=0;(c = \*(pathname++))&&(c != '/');len++)

/\* nothing \*/ ;

if (!c)

break;

base->i\_count++;

error = lookup(base,thisname,len,&inode);

if (error) {

iput(base);

return error;

}

error = follow\_link(base,inode,0,0,&base);

if (error)

return error;

}

if (!base->i\_op || !base->i\_op->lookup) {

iput(base);

return -ENOTDIR;

}

\*name = thisname;

\*namelen = len;

\*res\_inode = base;

return 0;

}

# 内存管理