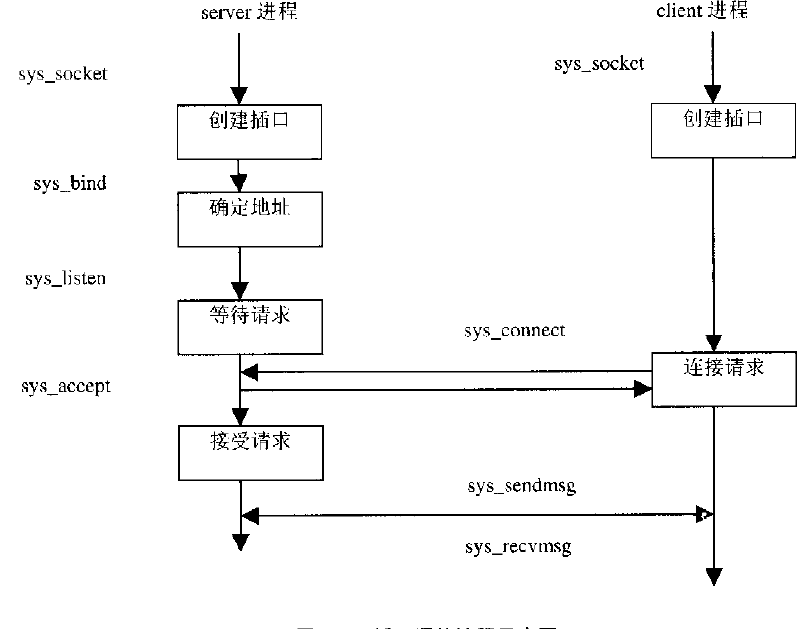
|  |
| --- |
| Marco |
| Linux Kernel Guide |
| * Source Code Analyze based on version 2.6.39 |
|  |
| **Xu Feng** |
| **2013/4/29** |

|  |
| --- |
| [在此处键入文档摘要。摘要通常为文档内容的简短概括。在此处键入文档摘要。摘要通常为文档内容的简短概括。] |

# Linux之socket

系统调用：sys\_socketcall  
                ----> sys\_socket       
                ----> sys\_bind  
                ----> sys\_connect  
                ----> sys\_listen  
                ----> sys\_accept  
                ----> sys\_getsockname  
                ----> sys\_getpeername  
                ----> sys\_socketpair  
                ----> sys\_send  
                ----> sys\_sendto  
                ----> sys\_recv  
                ----> sys\_recvfrom  
                ----> sys\_shutdown  
                ----> sys\_setsockopt  
                ----> sys\_getsockopt  
                ----> sys\_sendmsg  
                ----> sys\_recvmsg

****

SYSCALL\_DEFINE2(socketcall, int, call, unsigned long \_\_user \*, args)

    ...

Switch(call) {

case SYS\_SOCKET:

sys\_socket();

case SYS\_BIND:

sys\_bind();

case SYS\_CONNECT:

sys\_connect();

case SYS\_LISTEN:

sys\_listen();

case SYS\_ACCEPT:

sys\_accept4();

case SYS\_SEND:

sys\_send();

SYSCALL\_DEFINE4(send, int, fd, void \_\_user \*, buff, size\_t, len, unsigned, flags)

case SYS\_SENDTO:

sys\_sendto();

case SYS\_RECV:

sys\_recv();

case SYS\_SENDMSG:

sys\_sendmsg();

case SYS\_RECVMSG:

sys\_recvmsg();

…

}

* 1. **socket: net family, type, protocol**

  Socket数据机构





* 1. **Socket: socket**

sys\_socket

  ----> sock\_create  创建并设置socket 结构

       ----> \_\_sock\_create

           ----> sock\_alloc 为socket结构分配内存

               ----> new\_inode 调用sock\_mnt的alloc\_inode创建inode即sockfs\_ops.alloc\_inode，对socket来说他分配了一个socket\_alloc结构

----> pf->create = inet\_create

               ----> sk\_alloc  根据protocol类型设置socket结构，并为socket分配内存创建sock结构

----> sk\_prot\_alloc

              ----> sock\_init\_data  初始化sock结构中的队列等成员

----> sock\_map\_fd

* 1. **Socket: Bind**

sys\_bind

   ----> sockfd\_lookup\_light根据socket号查找对应的socket结构

----> fget\_light

----> sock\_from\_file

   ----> sock->ops->bind对inet的SOCK\_STREAM 来说是inet\_stream\_ops.bind即inet\_bind

----> sk->sk\_prot->bind 对inet的SOCK\_STREAM 来说是tcp\_prot.bind (目前为NULL)

      ----> inet\_addr\_type 检查地址类型，根据地址类型判断是否允许绑定此地址

      ----> sk->sk\_prot->get\_port 对TCP\_PROT说是tcp\_prot.get\_port即inet\_csk\_get\_port，函数判断此CPU上的tcp\_hashinfo中tcp\_bhash哈希表中指定端口是否被占用，若占用则绑定失败，返回提示绑定冲突。否则执行以下代码：

      ----> tcp\_bucket\_create 为端口创建tcp\_bind\_bucket结构

          ----> tcp\_bind\_hash 为tcp\_bind\_bucket与sock结构建立联系

---> ServiceManager.addService(Context.ACTIVITY\_SERVICE, m, true) 注册Binder

* 1. **Socket: Listen**

sys\_listen

----> sockfd\_lookup\_light 根据socket号查找对应的socket结构

---> sock\_from\_file

       ----> sock->ops->listen 对inet的SOCK\_STREAM 来说是inet\_stream\_ops.listen = inet\_listen

----> inet\_csk\_listen\_start Socket进入Listen状态

----> reqsk\_queue\_alloc 创建icsk\_accept\_queue

----> sk->sk\_prot->get\_port = inet\_csk\_get\_port

----> sk->sk\_prot->hash = inet\_hash 将socket记录近hash表  
       ----> fput\_light

* 1. **Socket: Accept**

From driver:

NET\_RX\_SOFTIRQ softirq

----> net\_rx\_action

----> n->poll = cp\_rx\_poll (对于8139网卡驱动poll注册此接口)

----> cp\_rx\_skb (分配skb用于保存接收的数据)

----> netif\_receive\_skb

----> \_\_netif\_receive\_skb

----> func = ip\_rcv

----> ip\_rcv\_finish

----> ip\_rcv\_options (查找路由表)

----> dst\_input

----> skb\_dst(skb)->input = ip\_local\_deliver

----> ip\_local\_deliver\_finish

----> ipprot->handler = tcp\_protocol.tcp\_v4\_rcv

----> tcp\_v4\_do\_rcv

----> tcp\_v4\_hnd\_req

---> tcp\_set\_state(sk, TCP\_ESTABLISHED)

----> tcp\_v4\_hnd\_req

----> inet\_csk\_search\_req

---> tcp\_check\_req

---> icsk\_af\_ops->syn\_recv\_sock = tcp\_v4\_syn\_recv\_sock

---> \_\_inet\_hash\_nolisten

----> tcp\_child\_process

----> parent->sk\_data\_ready = sock\_def\_readable 唤醒等待队列 (sk->sk\_wq)

IP layer

---> tcp\_v4\_do\_rcv   (For data)

        ---> tcp\_rcv\_established

            ---> \_\_skb\_queue\_tail

            ---> sk->sk\_data\_ready = sock\_def\_readable 唤醒等待队列 (sk->sk\_wq)

**sys\_accept4**     ----> sockfd\_lookup\_light 根据socket号查找对应的socket结构?  
     ----> sock\_alloc 调用sockfs\_ops.alloc\_inode为服务端创建socket\_alloc结构，内部包含一个socket结构和一个inode结构

----> sock\_alloc\_file  
     ----> sock->ops->accept 对INET说是inet\_stream\_ops.accept 即inet\_accept  
         ----> sk1->sk\_prot->accept 对TCP\_PROT说是tcp\_prot.accept 即inet\_csk\_accept  
                     ----> inet\_csk\_wait\_for\_connect若accept\_queue为空，则等待timeout后返回或直接返回（若NONBLOCK),否则：

                     ----> reqsk\_queue\_get\_child 获取accept\_queue中第一个request，调整accept\_queue为指向下一个请求  
                     ----> 读取请求对应的sock结构指针

          ----> sock\_graft 将新socket与accept请求的sock结构关联起来  
      ----> newsock->ops->getname 对INET说是inet\_stream\_ops.getname 即inet\_getname，获取对端的地址并拷贝到用户空间  
     ----> fd\_install 为新socket分配file结构及文件号，并与新socket结构关联

* 1. **Socket: Send**

Reference

<http://blog.csdn.net/jasenwan88/article/details/7695759>

<http://simohayha.iteye.com/blog/556168>

**sys\_send**

----> sys\_sendto 将缓冲中的数据通过指定socket发送出去

----> sockfd\_lookup\_light 通过fd查找对应的socket结构

----> sock\_sendmsg 通过socket发送消息

----> \_\_sock\_sendmsg

----> sock->ops->sendmsg 对INET来说即为inet\_stream\_ops.sendmsg = inet\_sendmsg

----> inet\_autobind 若socket未绑定，则进行自动绑定

----> sk->sk\_prot->get\_port = inet\_csk\_get\_port 获取未使用端口作为源端口

----> sk->sk\_prot->sendmsg 对TCP来说即为tcp\_prot.sendmsg = tcp\_sendmsg

----> tcp\_send\_mss 计算当前MSS值（最大分段）

----> tcp\_current\_mss

----> tcp\_write\_queue\_tail 获取sk->sk\_write\_queue队列队尾

----> tcp\_send\_head

----> sk\_stream\_alloc\_skb 为数据分配新的skb

----> skb\_entail 将新的skb入队sk->sk\_write\_queue

----> 若skb中还有空间则 skb\_add\_data 将用户空间数据拷入skb

----> 若skb没有空间则 sk\_stream\_alloc\_page 分配page

----> skb\_copy\_to\_page 将数据从用户空间拷贝到page中

----> \_\_tcp\_push\_pending\_frames

----> tcp\_write\_xmit

----> tcp\_transmit\_skb

----> icsk\_af\_ops->queue\_xmit = ip\_queue\_xmit将数据传入IP层

----> ip\_local\_out

----> \_\_ip\_local\_out

----> dst\_output

----> skb\_dst(skb)->output = ip\_output

----> ip\_finish\_output

* 1. **Socket: Recv**

From User: sys\_recv

---> sys\_recv

---> sys\_recv

--> sys\_recvfrom

----> sockfd\_lookup\_light通过fd查找对应的socket结构

----> sock\_recvmsg

---> \_\_sock\_recvmsg

----> \_\_sock\_recvmsg\_nosec

----> sock->ops->recvmsg = inet\_stream\_ops. inet\_recvmsg

----> sk->sk\_prot->recvmsg = tcp\_recvmsg

----> handle sk\_receive\_queue and copy data from skb to user space message

* 1. **Socket: Connect**

**sys\_connect**

----> sockfd\_lookup\_light 根据socket号查找对应的socket结构?

----> move\_addr\_to\_kernel 拷贝参数到内核空间

----> sock->ops->connect 对TCP\_PROT说是tcp\_prot.connect 即tcp\_v4\_connect

----> ip\_route\_connect 根据目的地址查找路由，如找到路由则：

----> tcp\_set\_state设置socket状态TCP\_SYN\_SENT

----> inet\_hash\_connect为connect操作分配一个端口，并为之创建inet\_bind\_bucket 结构

----> ip\_route\_newports

----> ip\_route\_output\_flow

----> \_\_ip\_route\_output\_key 遍历路由cache表rt\_hash\_table，根据源/目的地址查找路由

----> ip\_route\_output\_slow 若cache表中没有缓存路由则查找真正路由表fib\_table\_hash

----> fib\_lookup

----> \_\_mkroute\_output

----> rt\_intern\_hash 将找到的路由项保存到路由cache表里

----> tcp\_connect 发送SYN包

----> tcp\_connect\_init 初始化TCP链接

----> alloc\_skb\_fclone为SYN包创建sk\_buff结构

----> tcp\_init\_nondata\_skb 设置SYN位

----> \_\_tcp\_add\_write\_queue\_tail 将sk\_buff挂入sk\_write\_queue队列

----> tcp\_transmit\_skb 发送队列中的消息

----> icsk\_af\_ops->queue\_xmit = ip\_queue\_xmit

----> ip\_local\_out

* 1. **Socket: Select**

**sys\_select （select.c）**

----> copy\_from\_user 读取timeout时间到内核空间

----> core\_sys\_select

----> do\_select 对bits中的fd\_set进行select

----> 从0循环到max\_fd， **对select来说每次都要轮询所有的socket，对于大量的socket来说效率不高**

----> fget 找到对应socket的file结构

----> file->f\_op->poll 对socket来说即为socket\_file\_ops.poll = sock\_poll

----> sock->ops->poll 对INET来说即为inet\_stream\_ops.poll = tcp\_poll

----> 如果rcv\_nxt！=copied\_seq，设置POLLIN

----> sk\_stream\_wspace(sk) >= sk\_stream\_min\_wspace(sk) 如果当前剩余的发送缓冲大于最小需要的发送缓冲，设置POLLOUT

----> 循环下一fd

----> poll\_schedule\_timeout 若无时间发生在监听fd上则等待timeout时间

# **Epoll实现**

1. **数据结构**





**eventpoll\_init**

----> ep\_nested\_calls\_init

----> kmem\_cache\_create 创建内核内存cache，用于存放epitem

----> kmem\_cache\_create 创建内核内存cache，用于存放eppoll\_entry

----> register\_filesystem 注册epoll文件系统eventpoll\_fs\_type

* 1. **Epoll模块初始化**
  2. **Epoll创建**

**sys\_epoll\_create**

-----> sys\_epoll\_create1

----> ep\_alloc 创建eventpoll结构

----> anon\_inode\_getfd

----> anon\_inode\_getfile

----> get\_unused\_fd\_flags 获取可用的fd号

----> anon\_inode\_getfile为epoll创建file结构，文件号及inode结构，并将它们关联起来，同时将之前创建的eventpoll结构赋给private\_data，eventpoll内有一棵红黑树，维护所有监听的socket

1. **Epoll设置**

**sys\_epoll\_ctl**

----> copy\_from\_user 将要监听的时间拷贝到内核空间

----> fget 获取epoll对应的file结构

----> fget 获取要监听socket对应的file结构

----> ep\_find 根据监听socket的file结构和socket号在epoll红黑树中查找其对应的epitem

----> 根据op的类型执行以下操作

----> EPOLL\_CTL\_ADD：ep\_insert 没找到对应epitem则执行插入操作

----> init\_poll\_funcptr 设置poll\_table的qproc为ep\_ptable\_queue\_proc

----> tfile->f\_op->poll 对TCP\_PROT来说是socket\_file\_ops.poll即为sock\_poll

----> sock->ops->poll 对INET来说是inet\_stream\_ops.poll即为tcp\_poll

----> poll\_wait

----> p->qproc = ep\_ptable\_queue\_proc 设置epoll的callback函数

----> init\_waitqueue\_func\_entry 设置callback函数为ep\_poll\_callback

----> EPOLL\_CTL\_DEL：ep\_remove 若找到对应epitem则执行删除操作

----> EPOLL\_CTL\_MOD：ep\_modify 若找到对应epitem则执行修改操作

1. **Epoll监听**

ep\_insert

----> init\_poll\_funcptr

----> pt->qproc = ep\_ptable\_queue\_proc

----> tfile->f\_op->poll = ep\_eventpoll\_poll

----> poll\_wait

----> ep\_ptable\_queue\_proc

----> init\_waitqueue\_func\_entry 将新epitem加入poll\_wait队列

----> q->func = ep\_poll\_callback

sys\_epoll\_wait

----> fget 获取epoll对应的file结构

----> ep\_poll 对指定的epoll开始监听

----> ep\_events\_available 若没有监听事件产生，则挂起等待

----> ep\_send\_events 将事件发回用户空间

----> ep\_scan\_ready\_list

----> ep\_send\_events\_proc 将有事件的epitem链入ep->rdllist队列

----> f\_op->poll = sock\_poll读取对应socket的events

----> sock->ops->poll对INET来说即为inet\_stream\_ops.poll = tcp\_poll 读取event

----> ep\_poll\_safewake

----> ep\_call\_nested

----> ep\_poll\_wakeup\_proc

----> ep\_wake\_up\_nested

----> wake\_up\_poll

----> \_\_wake\_up

----> \_\_wake\_up\_common

----> curr->func = ep\_poll\_callback

----> list\_add\_tail(&epi->rdllink, &ep->rdllist)

----> ep\_poll\_safewake

# **IP Routing**

* 1. **IP: Routing数据结构**

rt\_hash\_table[rt\_hash\_bucket] -->{ chain [rtable] }

fib\_tables[fib\_table]+ [fn\_hash] -->{ fn\_zones [fn\_zone], ---> { fz\_next [fn\_zone]

fn\_zone\_list [fn\_zone ]

fz\_hash [hlist\_head] ---> [fib\_node]

} }





* 1. **添加路由**

dev\_open

---> call\_netdevice\_notifiers(NETDEV\_UP, dev);

---> inetdev\_event

---> inet\_insert\_ifa

---> \_\_inet\_insert\_ifa

---> rtmsg\_ifa(RTM\_NEWADDR)

---> blocking\_notifier\_call\_chain(NETDEV\_UP)

---> notifier\_call\_chain

---> nb->notifier\_call = fib\_netdev\_event

---> fib\_add\_ifaddr

---> fib\_magic(RTM\_NEWROUTE, RTN\_LOCAL, addr, 32, prim)

---> fib\_table\_insert

---> rtmsg\_fib(RTM\_NEWROUTE)

---> rtnl\_notify

---> nlmsg\_notify (send message to netlink socket net->rtnl)

Handle message received on netlink socket

rtnetlink\_rcv\_msg

---> doit = inet\_rtm\_newroute

---> rtm\_to\_fib\_config

---> fib\_new\_table

---> fib\_table\_insert

* 1. **ioctl调用设置路由**

sys\_ioctl

--> filp->f\_op->ioctl (socket\_file\_ops.ioctl = sock\_ioctl)

--> sock->ops->ioctl (inet\_stream\_ops.inet\_ioctl)

--> ip\_rt\_ioctl

--> tb->tb\_insert/tb->tb\_delete

# **系统进程相关**

   系统调用列表： sys\_call\_table

sys\_fork

---> do\_fork

---> copy\_process

---> dup\_task\_struct 为新task分配task\_struct结构

---> copy\_semundo

---> copy\_files 拷贝当前进程的files结构

---> dup\_fd

---> copy\_fs

---> copy\_fs\_struct

---> copy\_sighand

---> copy\_signal

---> copy\_mm 拷贝当前进程的mm\_struct结构

---> dup\_mm

---> dup\_mmap

---> copy\_io

---> copy\_thread

---> wake\_up\_new\_task

---> activate\_task

---> enqueue\_task

---> p->sched\_class->enqueue\_task

# **文件操作**

* 1. **打开文件**

SYSCALL\_DEFINE3(open, …)

---> do\_sys\_open

---> getname

---> get\_unused\_fd\_flags

---> files\_fdtable

---> expand\_files

---> do\_filp\_open

---> path\_openat

---> get\_empty\_filp

---> path\_init

* 1. **写文件**
  2. **读文件**
  3. **关闭文件**

# **Linux系统权限管理**

# **Linux内存管理**

# **系统基础知识**

   1. 内存地址0xC000 0000 ~ 0xFFFF FFFF是内核空间 0x0000 0000~ 0xBFFF FFFF为用户空间

   2. 系统进程哈希表

   3.  物理空间管理数据结构： mem\_map 管理所有物理页面page结构的数组

      page: 表示一个物理页面

      zone\_struct: 代表不同的管理区所管理的页面

     free\_area\_t： 每个管理区有一组队列，用于管理不同大小的连续物理页面

   4.  虚拟空间管理数据结构： vm\_area\_struct: 表示一个虚存区间，该区间的虚存具有相同属性例如访问权限，同时它是一个AVL树的节点，方便查找

      mm\_struct: 进程整个用户空间的抽象，

      zone\_struct: 代表不同的管理区所管理的页面

      free\_area\_t： 每个管理区有一组队列，用于管理不同大小的连续物理页面

# **Linux启动及初始化**

<https://www.ibm.com/developerworks/cn/linux/l-linuxboot/>

BIOS 固定地址引导程序

--> stage 1 512B加载程序

--> stage2 GRUB

--> 解压内核并运行 Head.s: startup\_32

--> start\_kernel

--> setup\_arch/rest\_init/trap\_init/init\_IRQ

--> kernel\_thread --> init

main.c: init

--> do\_basic\_setup

--> sock\_init/do\_initcalls

init.c: paging\_init

-->pagetable\_init

　 physmem.c: init\_maps

startup\_32

---> decompress\_kernel

---> startup\_32 (/arch/i386/kernel/head\_32.S) <process 0>

---> i386\_start\_kernel

---> start\_kernel

---> rest\_init

---> kernel\_thread

---> do\_fork

---> kernel\_init

---> sys\_access

---> init\_post

---> run\_init\_process(ramdisk\_execute\_command); = '/init'

---> init.rc / init.xx.rc

---> service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server

---> run\_init\_process(execute\_command);

---> run\_init\_process("/sbin/init");

---> run\_init\_process("/etc/init");

---> run\_init\_process("/bin/init");

---> run\_init\_process("/bin/sh");

---> cpu\_idle

# **X86平台寄存器用途说明**

     CR3: 当前进程页目录地址（物理地址），该地址对应内存虚地址在内存中，需要转换成物理地址写入此寄存器中

     TSS: 不是寄存器，他是一个与进程对应的结构，用于保存进程运行现场，包括所有与进程相关的寄存器内容及三个堆栈指针

     TR:  任务寄存器，用于指向当前进程的TSS

     EFLAGS: 记录当前状态位，例如IF标志位0表示关闭中断，在中断处理程序中设置该位0防止中断嵌套，在处理自陷时保持该位为1.

     CS: 当前进程代码段寄存器。

     IDTR: 指向当前中断向量表IDT

     ES:  附加段寄存器

     DS: 数据段寄存器

     CS:  代码段寄存器

     SS:  堆栈段寄存器 task\_struct中记录有运行级别为0， 1, 2时对应的SS和ESP内容

     ESP: 指向当前进程堆栈顶部

     EAX: 通用寄存器

     EBP: 通用寄存器， 当前进程栈底指针

     EIP: 存放CPU的下一条指令地址

     EDI: 通用寄存器

     ESI: 通用寄存器

     EDX: 通用寄存器

     ECX: 通用寄存器

     EBX: 通用寄存器

# **Linux中断处理**

     IDT: 中断描述表，位长度256的数组，里面保存中断描述符

     0~20号中断为系统占用中断，中断描述符中包含中断处理程序入口地址

     其他中断为第二种，可支持多个中断源公用，中断描述符中的中断入口地址为统一入口地址，内部将便利数组irq\_descs内的对应中断处理队列，根据中断源找到对应的中断处理程序指向。

     初始化： trap\_init/init\_IRQ --> init\_ISA\_irqs

     注册： request\_irq函数用于等级中断处理程序到irq\_descs数组中

# **Linux进程调度**

<http://blog.csdn.net/shinesi/article/details/1937622>

     2.6内核： 每个CPU都有一个runqueues结构，位于每个CPU的专有数据区中，他在start\_kernel时通过setup\_per\_cpu\_areas分配，runqueues用于存放记录有所有进程的队列数组runqueues ->active->queue[MAX\_PRIO]，该数组按优先级将所有进程入队管理，便于系统调度。

    实时进程优先级：0~100

    普通进程优先级：101 ~140, 普通进程权值为优先级和进程平均等待时间sleep\_avg 计算的奖励值之和，但不能小于101

    sleep\_avg更新算法参见 <http://hi.baidu.com/renling456000/item/11a4dc4d2b0f7ee5a4c0665c>

# **Linux文件系统**

数据结构

task\_struct ->fs和 task\_struct->files-> {

                   file\_operations f\_op,

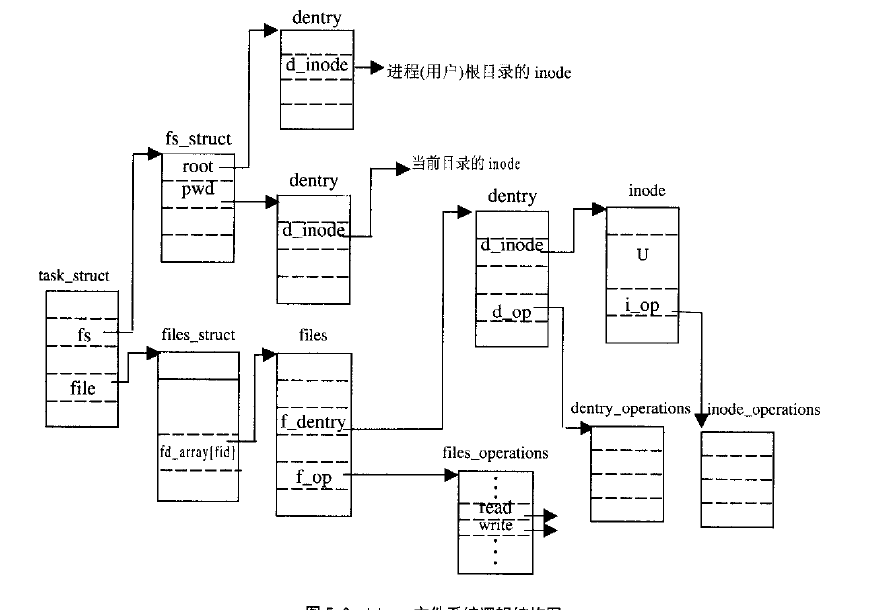
                   dentry f\_dentry --------------------> {

inode\* d\_inode,

dentry\_operations \*d\_op

}

}

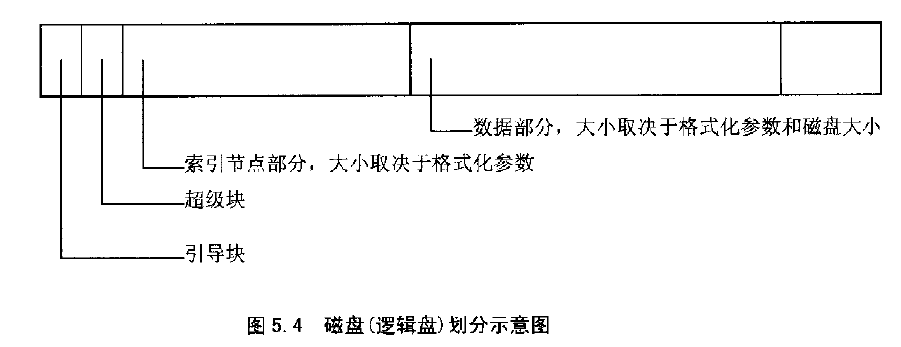


              磁盘结构                   内存结构

             ext2\_inode ---------- inode

            ext2\_dir\_entry\_2 ------ dentry

     磁盘第一逻辑块为引导块，用于引导系统启动，第二逻辑块即为文件系统根设备，即为 super\_block。

内核函数\_\_user\_walk用于根据给定路径获取其inode。

dentry\_hashtable：内存dentry哈希表，保存当前所有dentry结构

inode\_hashtable：内存inode哈希表，保存已创建的inode结构

dentry内队列说明：

* d\_hash 挂入全局dentry\_hashtable 表中
* d\_lru 共享为0是挂入dentry\_unused表中
* d\_child 挂入父节点的d\_subdirs队列中
* d\_parent指向父节点的dentry结构
* d\_subdirs 包含子节点dentry队列
* d\_alias 挂入对应inode的i\_identry队列

d\_vfsmount 该目录项对应的设备vfsmount结构队列

inode结构说明：

* i\_identry 为所有引用该inode的dentry队列
* d\_sb: 设备super\_block
* i\_rdev：节点所代表的设备号
* i\_dev： 节点所在的设备号 如/dev
* i\_bdev：如果此结构代表块设备，该指针指向一个block\_device结构，为该设备的控制结构
* bd\_op：块设备函数跳转表

vfsmount结构：

* mnt\_mountpoint安装点的dentry结构
* mnt\_root为所安装设备上的根目录dentry结构
* mnt\_hash连入到dentry的d\_vfsmount 上
* mnt\_sb为所安装设备的super\_block结构
* mnt\_instances连入到其super\_block的s\_mounts上
* mnt\_parent指向上一层vfsmount结构



# **Linux用户及权限管理**

* 1. **数据结构：**
* task\_struct ->uid：当前用户id
* task\_struct->suid: 暂时改变euid是记录原来的euid值以便以后恢复
* task\_struct->euid: 当前有效的用户id
* task\_struct->fsuid: 当前访问文件系统的用户id
* 哈希表uidhash\_table：所有task\_struct根据其uid的哈希值挂入其某个队列中。
* 队列super\_blocks: 包含所有super\_block结构
* file\_systems： 为所有文件类型队列，该结构在系统初始化时通过register\_filesystem注册，对于ext2文件系统其type结构为ext2\_fs\_type



* 1. **mount文件系统过程：**

sys\_mount [系统调用，拷贝type, dir\_name及dev\_name等参数到内核空间后]

                     --> do\_mount  [参数检查，根据flags决定后续操作类型]

                          or --> 若flags设置MS\_REMOUNT： do\_remount

                          or --> 若flags设置MS\_BIND： do\_loopback

                          or --> 若flags设置MS\_MOVE： do\_move\_mount

                          or --> 其他： do\_new\_mount [须有CAP\_SYS\_ADMIN权限]

                                 --> do\_kern\_mount

                                       --> get\_fs\_type [调用find\_filesystem获取文件系统类型] file\_systems为所有文件类型队列，该结构在系统初始化时通过register\_filesystem注册，对于ext2文件系统其type结构为ext2\_fs\_type

                                       --> alloc\_vfsmnt [分配vfsmount结构 ]

                                       --> type->get\_sb (ext2\_get\_sb) [获取文件设备的super\_block]

                                           --> get\_sb\_bdev

                                               --> open\_bdev\_excl --> blkdev\_get --> do\_open

                                               --> fill\_super (ext2\_fill\_super)

                                                    --> sb\_bread

                                       --> security\_sb\_kern\_mount

                                 --> do\_add\_mount

# **Linux通讯之管道实现**

系统调用sys\_pipe，返回fd数组，fd[0]为读文件，fd[1]为写文件

           sys\_pipe

                  ----> do\_pipe

                        ----> get\_empty\_filp 为f1, f2创建对应的file结构

                        ----> get\_pipe\_inode 为管道文件创建对应的inode结构

                               ----> new\_inode 创建管道inode结构

                             ----> pipe\_new 为管道inode创建成员i\_pipe指向的pipe\_node\_info结构，并分配一页内存赋给i\_pipe.base

                        ----> get\_unused\_fd 为f1，f2获取空闲的fd号

                        ----> d\_alloc 为管道文件创建dentry结构

                  ----> copy\_to\_user 拷贝创建的fd数组到用户空间

pipe\_mnt： 全局变量，代表管道的vfsmount结构

管道文件inode的file\_operations： rdwr\_pipe\_fops

管道文件的dentry\_operations: pipefs\_dentry\_operations

管道两端文件操作的file\_operations： read\_pipe\_fops & write\_pipe\_fops

注：一般文件系统中file->f\_op和inode->i\_fop指向同一个file\_operations结构，但对于pipe来说其inode->i\_fop 指向rdwr\_pipe\_fops而file->f\_op分别指向read\_pipe\_fops & write\_pipe\_fops ，从而实现一边只读一边只写。

管道文件类型： pipe\_fs\_type

命名管道：

创建命名管道文件： mknod mypipe p

管道文件类型： pipe\_fs\_type

管道文件inode的file\_operations： def\_fifo\_fops

管道文件的dentry\_operations: pipefs\_dentry\_operations

管道两端文件操作的file\_operations： read\_fifo\_fops & write\_fifo\_fops & rdwr\_fifo\_fops

# **Linux通讯**

**18. Linux通讯之信号**

     每个进程的task\_struct->sig指向一个signal\_struct结构，该结构为信号向量表

     task\_struct->sighand ---> {

                                                      k\_sigaction action[\_NSIG] --->   {

                                                 }                                                                 \_\_sighandler\_t sa\_handler   信号服务程序

                                                                                                                   unsigned long sa\_flags

                                                                                                                    sigset\_t sa\_mask   位图，1表示在服务程序中屏蔽对应信号

                                                                                                           }

     task\_struct->signal ---> {

                                                sigpending shared\_pending --->   {

                                              }                                                                 sigset\_t signal 信号位图

                                                                                                                list\_head list    信号队列

                                                                                                           }

     task\_struct->sas\_ss\_sp 用户控件执行信号处理程序时堆栈位置

     task\_struct->sas\_ss\_size 用户控件执行信号处理程序时堆栈大小

     task\_struct->blocked 屏蔽信号位图，再是组织信号处理，屏蔽解除后信号还要得到处理

     系统调用设置信号 sys\_signal -----> do\_sigaction

     系统调用发送信号 sys\_kill -----> kill\_something\_info (pid=0 发给同组所有进程 pid=-1发给系统所有进程)

                                                       ----> (pid > 0) kill\_proc\_info

     系统调用发送信号 sys\_signalqueue -----> sys\_rt\_sigqueueinfo ----> kill\_proc\_info  ----> group\_send\_sig\_info

                                                      ----> \_\_group\_send\_sig\_info ----> send\_signal ----> 挂入目标进程shared\_pending队列并设置位图signal

     进程在从系统调用，中断或异常处理返回到用户控件前夕，又或者被从睡眠中唤醒时要检查是否有信号，并处理之。

     系统空间返回用户空间汇编代码ret\_with\_reschedule中compl $0 , sigpending(%ebx)  jne signal\_return 并执行do\_signal函数

        do\_signal ----> get\_signal\_to\_deliver

                         ----> handle\_signal

                                 ----> setup\_frame 在执行信号处理程序之前，将系统空间堆栈原始框架复制到用户堆栈中，并修改系统堆栈esp

指向修改后的用户堆栈顶部，更改eip指向信号处理程序地址ka->sa.sa\_handler，这样从系统空间返回用户空间后将从信号处理程序开始执行。

信号处理返回：

      系统调用sys\_sigreturn

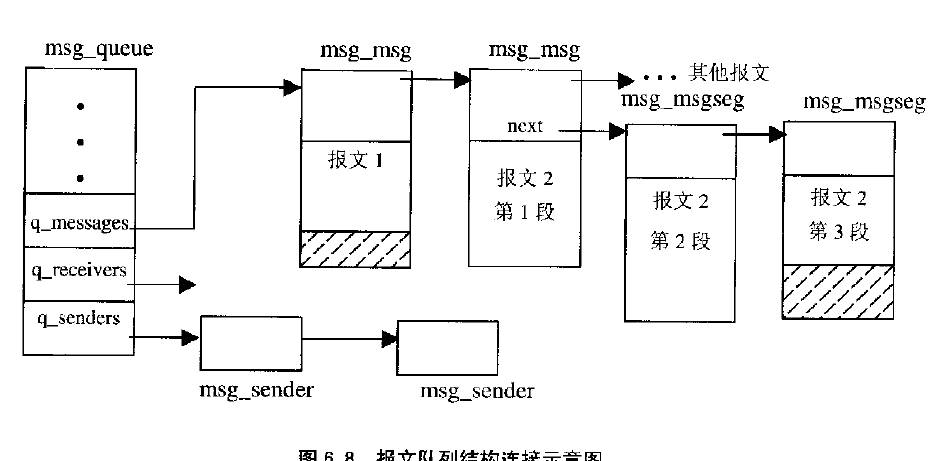
**18. Linux通讯之报文**

     重要数据结构：

           msg\_ids： 内核全局数据结构，用来管理报文队列。

           msg\_ids->ipc\_id\* entries: 用于保存报文队列结构ipc\_id的数组

           msg\_queue： 该结构对应着一个报文队列，其成员q\_perm的地址被写入一个ipc\_id机构并被填入msg\_ids->entries数组便于管理。



     系统调用sys\_ipc，实际上该系统调用处理的操作多达11种，包括报文，信号量及共享内存等。对于报文处理的分支，代码路径如下：

      case: MSGSND

                sys\_msgsnd

                      ----> get\_user 拷贝msg\_type到内核

                      ----> load\_msg 拷贝msg内容到内核

      case: MSGRCV

                sys\_msgrcv

      case: MSGGET

                sys\_msgget  根据给定键值创建或获取报文队列

     case: MSGCTL

                sys\_msgctl

**18. Linux通讯之共享内存**

     重要数据结构：

           shm\_ids： 内核全局数据结构，用来管理共享内存区。

           shm\_ids ->ipc\_id\* entries: 用于保存共享内存区结构ipc\_id的数组

           shmid\_kernel： 该结构对应着一个共享内存区，其成员shm\_perm的地址被写入一个ipc\_id机构并被填入shm\_ids ->entries数组便于管理。

           shm\_mnt： 内核全局变量，指向共享内存文件系统的vfsmount结构

           共享内存文件的f\_op指向shmem\_file\_operations，而他的inode的i\_fop指向shmem\_file\_operations

     共享内存通过文件映射实现，对应的文件类型为shmem\_fs\_type，该文件系统在系统初始化时通过kern\_mount安装

     系统调用sys\_ipc，共享内存分支代码路径如下：

      case: SHMGET

                sys\_shmget 共享内存区的创建及查找，创建共享内存映射文件并将共享内存区对应shmid\_kernel 挂入shm\_ids

      case: SHMAT

                do\_shmat 建立共享内存区映射

                      ----> do\_mmap

      case: SHMDT

                sys\_shmdt

     case: SHMCTL

                sys\_shmctl

**18. Linux通讯之信号量**

重要数据结构：

sem\_ids: 结构为ipc\_ids，成员entries为以sem id为下标的数组

系统调用：

sys\_semget

   ----> newary

sys\_semctl

sys\_semop

   ----> sys\_semtimedop

**19. Linux之内存管理**

     重要数据结构：

           pgd\_t: 页目录

           pmd\_t： 中间目录

           pte\_t： 页表项

           mem\_map：内核全局变量， 该数组代表系统全部物理页面，其内容在系统初始化时创建。

           pglist\_data： 存储节点链表

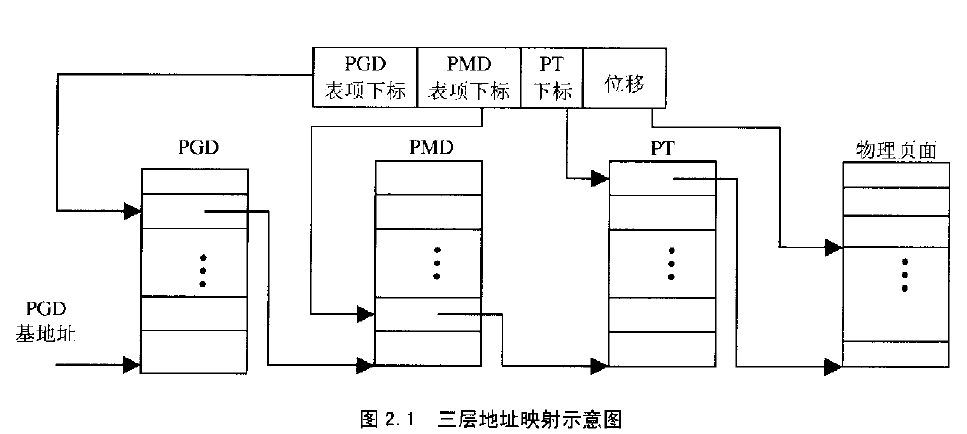
           zone：管理区结构

           zone.free\_area: 一个数组，数组下标为连续空闲页面大小2,4,6,8...开2的幂，分别指向对应大小物理块队列

           vm\_area\_struct：虚存区间数据结构

           node\_data：定义为struct pglist\_data \*node\_data[]，内核全局数组包含所有物理页面

     linux内核页面映射机制被设计成3层，页目录PGD, 中间目录PMD 和页表 PT



               各表项下标所占位数可设置，一般为2层设置，因此PGD为10位而PT为10位，位移为12位。

               内核占据地址0xC0000000~0xFFFFFFFF的1G空间，内核虚地址映射即为vaddr + 0xC0000000 = paddr

**全局段描述表GDT：**

               段寄存器中13位用作GDT下标，因而GDT有8192个表项。

               固定表项1：永远为空

               固定表项2：内核代码段

               固定表项3：内核数据段

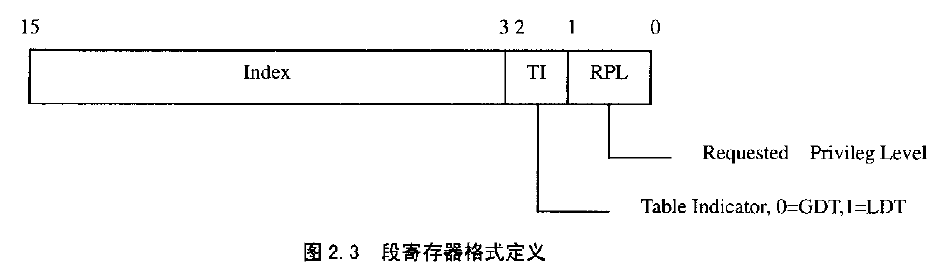
               固定表项4：当前进程代码段

               固定表项5：当前进程数据段

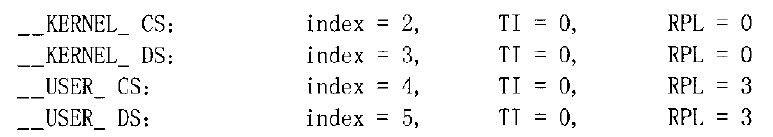
               ...... 除其他系统开销外还剩余8180项，每一个进程占用2项（代码段和数据段），因此理论上系统最多支持4090个进程。

**段映射**

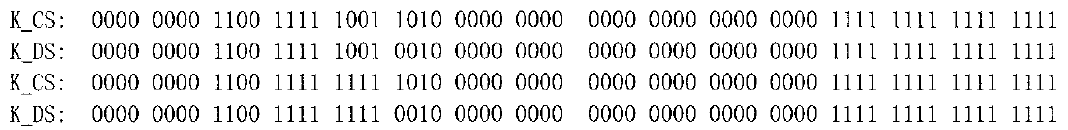
                  代码段寄存器CS，根据CS内容映射段描述表

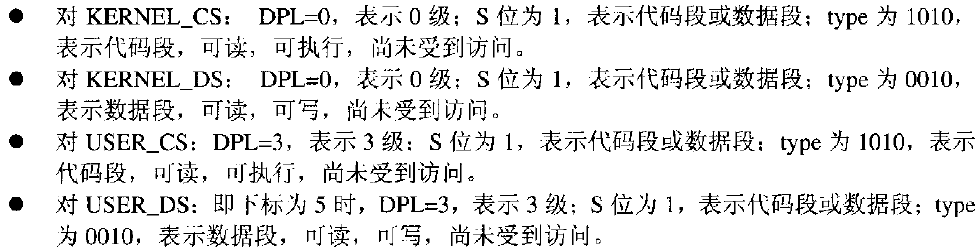


                    段寄存器值：



                   TI都为0，因此内核中全部使用GDT。GDT初始内容在head.S中定义：





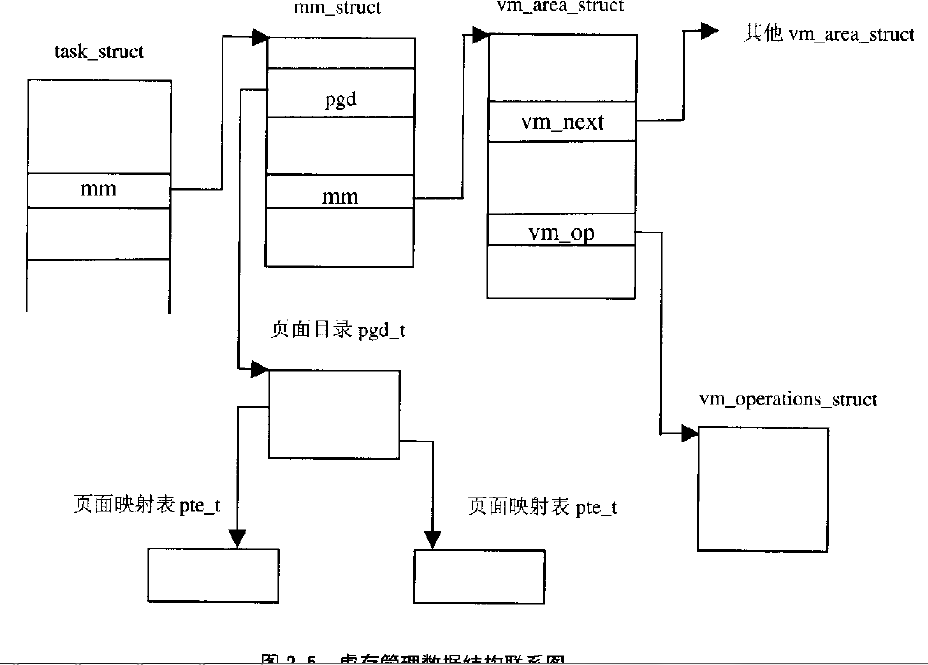
            B0~B15和B16~B31为段基址，但段描述符中各个段基址都为0，因而linux中各段基址都相同，虚地址段映射的结构都为其自身，到此段映射结束，下一步开始也映射。

**页映射**

                  页映射根据页表完成，每个进程都有其页表结构PGD，页表地址保存在进程的mm\_struct结构中。每当一个进程将要被运行时，内核都要讲起页表地址设置到寄存器CR3中。

                  虚地址高10位用于页目录下标找到对应页表，虚地址中间10位用于页表下标找到对应目录，剩余12位为对应页中的偏移。

                  映射中MMU先检查表项中状态位P，1表示该页在内存中，如果为0，MMU将因无法映射而产生一次缺页异常。如果映射成功，高20位将可以被用来作为下标在数组mem\_map中找到对应的物理页面。



**虚存管理**

虚存以vm\_area\_struct结构进行管理，该结构代表虚存地址区间，内核以红黑树的形式保存所有虚存区间，该红黑树的根节点保存在进程对应的mm\_struct结构中。内核根据要访问的虚地址在红黑树种查找对应的虚存区间，如果找不到则会产生一个页面出错异常，通过异常处理响应机制，最终走到处理程序主体do\_fault\_page函数。

             越界访问通常是有问题的，只有一种情况除外，那就是堆栈访问，堆栈越界访问可能将触发堆栈伸展。

**物理页面管理**

            数据结构：

                zone

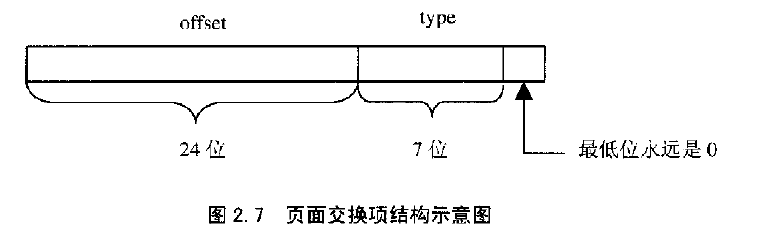
                swap\_info\_struct： 描述和管理用于交换的文件及设备

                swap\_info\_struct.swap\_map： 数组，数组中每一个元素代表一个盘上页面

                swap\_info：swap\_info\_struct结构的数组，因为系统可能存在多个交换设备，该数组管理系统中所有交换设备

                swap\_list：系统中所有交换设备按优先级排列的队列

                swp\_entry\_t：盘上页面对应的页面表项，他是一个32位无符号整数，具体结构如下：



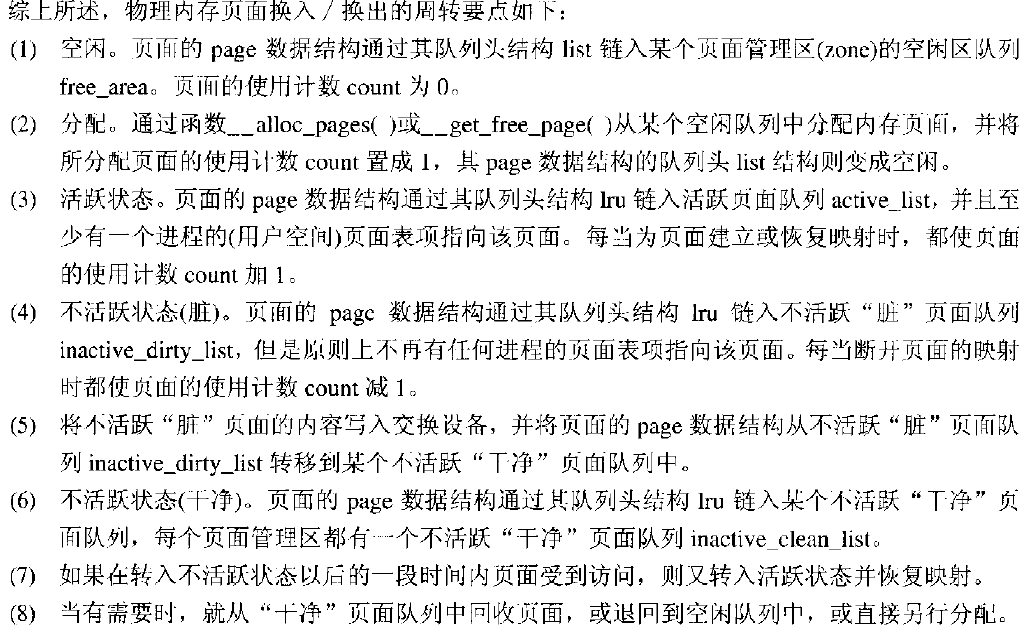
                                  offset：表示页面在磁盘设备或文件中的位置，即逻辑页面号

                                  type: 指页面在哪一个文件中

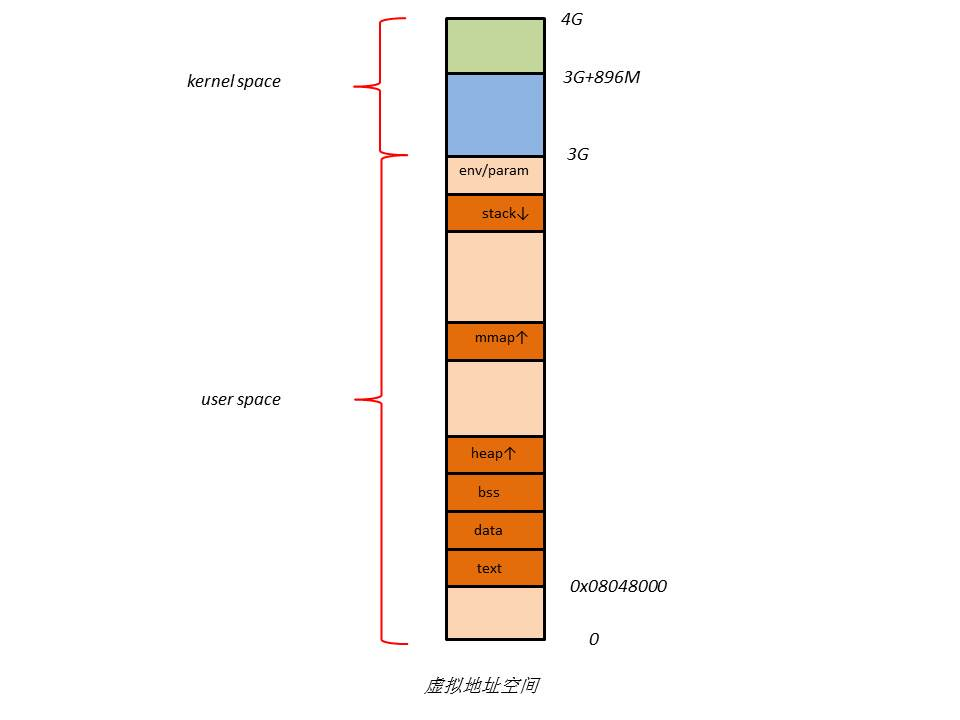
             物理页面：内存页面（内存中）和盘上页面（磁盘上）

             内存页面：系统初始化时根据检测到的内存大小，按页面大小将内存分成若干页面，并创建page结构与之对应并管理。

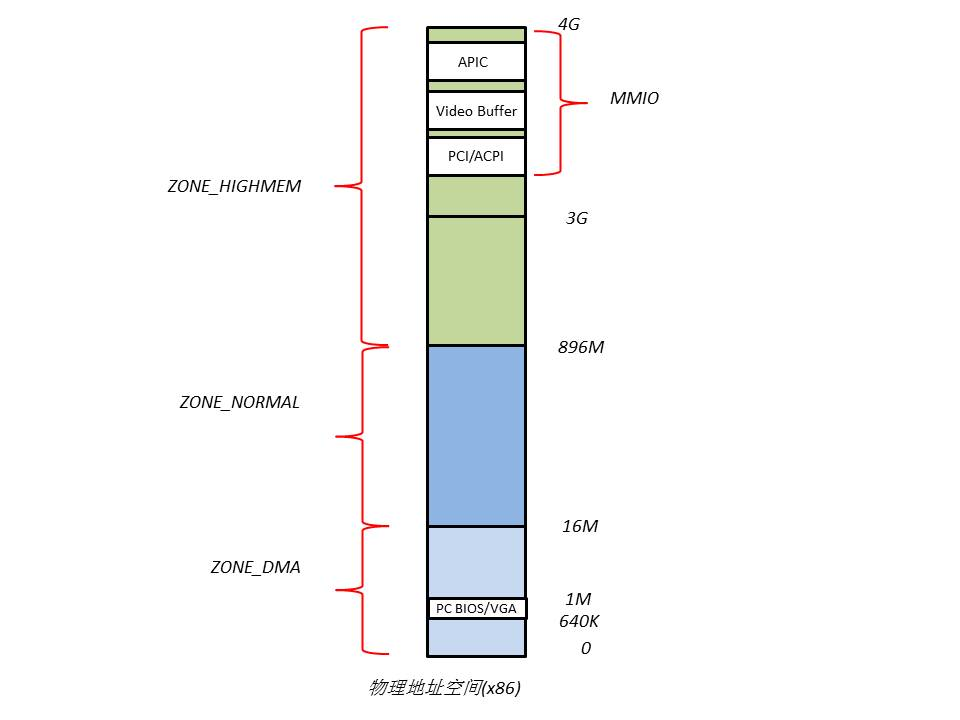
             页面周转包含两方面的意思，一方面是页面分配，使用和回收，另一方面是盘区交换。并不是所有页面都可以交换出去，事实上只有用户空间页面可以交换出去。



              内存用户控件布局：



               内存物理空间布局：



**20. Linux之设备驱动**

     重要数据结构：

          chrdevs：系统中字符设备数组

      linux设备分为两大类，一类为像磁盘一样已记录块为单位输入输出的设备，成为块设备，一类为想键盘一样以字符或字节为单位输入输出的，成为字符设备。设备文件节点记载着设备建立连接所需的信息，分为三部分，第一部分是设备类型，第二部分是主设备号，第三部分为次设备号。

**系统调用mknod**： 用于创建设备文件

             sys\_mknod

                      -----> vfs\_mknod

                              ----> dir->i\_op->mknod (ext2\_mknod)

                                      ----> ext2\_new\_inode 创建对应设备文件的inode

                                      ----> init\_special\_inode 设置设备文件inode结构的i\_fop和i\_rdev

                                      ----> mark\_inode\_dirty 将inode标记为dirty

                                      ----> ext2\_add\_nondir 将inode加入其所在目录的文件队列中，并将本文件的inode和dentry挂钩

**系统可安装模块**

         系统调用： 1. create\_module：2.6已经删除该系统调用

                            2. sys\_init\_module

                                         ----> load\_module 创建module数据结构

                                         ----> mod->init 调用module的init\_module函数对module进行初始化

                            3. query\_module： 2.6已经删除此系统调用

                            4. sys\_delete\_module

          数据结构：

                  module\_symbol: 描述一个符号，包括符号名及其地址

                  module\_ref: 描述模块间的关系

                  module： 对应一个可安装模块，内核中的module通过next指针练成队列

                  module.name：模块名

                  module.size：模块大小

                  module.init：指向模块的init\_module函数

                  module.cleanup：指向模块的cleanup\_module函数

                  module.syms：指向一个module\_symbol数组，即为模块符号表

                  module.nsyms：指明syms数组大小

                  module.deps：指向一个module\_ref数组

                  module.ndeps：指明deps 数组大小

                  modules：全局数组，用于保存系统所有module

                  kernel\_module：内核对应的module结构

**安装sparcaudio声卡驱动实例：**

                   sparcaudio\_init 模块入口函数，对可安装模块他是init\_module函数

                       ---> devfs\_register\_chdev 注册字符设备

                           ---> register\_chrdev  注册字符设备，将创建的字符设备写入chrdev数组

                       ---> devfs\_mk\_dir 创建设备文件所在的目录

**20. Linux之系统引导及初始化**

     i386系统：

          1. 加电后从地址0xFFFF0执行程序指令，故此位置上必须为不挥发存储器。

          2. 此程序一般很小，称为引导程序，用于从读入引导扇区，引导扇区上包含操作系统相关的数据（例如系统映像的位置）和程序，引导扇区上还可能是另一个程序，称为“引导装入程序”，由它引导装入操作系统映像。具体：BIOS读取引导扇区读入0x7c00，然后从0x7c00处开始执行引导扇区代码

          3. 引导扇区程序将其自身搬运到0x90000处，并跳转到那里继续执行，然后从磁盘上读入setup和内核映像，setup被读到0x90200

          4. 跳转到0x90200处执行，做好执行内核代码准备，包括映像解压缩

          5. 内存的头640K（0x00000~0xA0000），0x90000~0xA0000用于存放bootsec和setup程序，0x00000~0x10000保留给BIOS，剩余8个64k即512K空间，除去4k用于引导命令行和传递内核参数，剩余508k可用于存放内核映像

          6. 对于小映像zImage被装载在508K内，对于大映像bzImage则被装载在地址0x100000，不管大映像还是小映像，解压后都放在地址0x100000

          7. setup完成内核映像解压后，跳转到0x100000处执行内核本身

**内核启动阶段 1：**

          8.  内核映像代码入口startup\_32，其物理地址为0x100000，虚地址为0xC0100000

          9.  创建临时页表

          10. 创建进程0

          11. 创建中断描述表IDT

          12. start\_kernel

**内核启动阶段 2：**

          13.  页表，内存等资源初始化

          14.  kernel\_thread启动init进程即进程1

          15.  网络等设备初始化，文件系统建立

**20. Linux之内存分配**

   sys\_brk  
        ----> 若堆地址小于当前堆地址，则缩减堆  
            ----> do\_munmap  
        ----> 若扩大堆  
        ----> find\_vma\_intersection 检验新对地址是否在存在的VMA中，若没有则：  
        ----> do\_brk  
            -----> find\_vma\_prepare 找到起始地址所在的VMA，若找到则先umap新虚地址段，然后repeat此步骤  
            -----> vma\_merge 将新的虚地址段merge到现有的VMA中，若不能merge则需要创建新的VMA  
            -----> kmem\_cache\_alloc 分配新的VMA  
            -----> vma\_link 把新的VMA加入到红黑树种  
        ----> 更新brk到mm\_struct

**21. Linux之epoll实现**

POLL：

sys\_poll  系统调用

     ----> poll\_initwait 创建该POLL的poll\_wqueues，并设置其poll\_table.qproc = \_\_pollwait

     ----> while() {}  将传入的fd拷入内核空间并将其加入poll\_list队列中 （此处拷贝fd是性能瓶颈）

     ----> do\_poll

           ----> while() {} 对poll\_list 中的fd执行POLL操作（此处顺序poll是性能瓶颈）

                  ----> tcp\_poll 对tcp socket调用此poll函数

                         ----> poll\_wait

                               ----> p->qproc 此处调用注册的\_\_pollwait 函数

                                    ----> 将当前进程current挂入socket的设备等待队列sk->sk\_sleep

      ----> while() {} 对poll\_list中所有fd读取其revents，并拷贝到用户空间

EPOLL：

    eventpoll\_init 初始化epoll module

          ----> kmem\_cache\_create  创建cache用于分配epitem

          ----> kmem\_cache\_create  创建cache用于分配eppoll\_entry

          ----> register\_filesystem 创建epoll的文件系统，类型为eventpoll\_fs\_type

          -----> kern\_mount 创建虚拟文件系统的根eventpoll\_mnt

   sys\_epoll\_create 创建新的epoll

          ----> ep\_getfd 在epoll文件系统中创建新的文件

                ----> get\_empty\_filp

                ----> ep\_eventpoll\_inode

                ----> get\_unused\_fd

          ----> ep\_file\_init 分配并初始化eventpoll，并将其挂入file->private\_data

sys\_epoll\_ctl  设置epoll

          ----> ep\_find 根据传入socket的fd在file->private\_data 内的红黑树种查找

          ----> (EPOLL\_CTL\_ADD:ep\_insert) 插入操作

                 ----> EPI\_MEM\_ALLOC 分配epitem

                 ----> init\_poll\_funcptr 设置epq.pt.qproc = ep\_ptable\_queue\_proc

                 ----> tfile->f\_op->poll 对于tcp socket调用tcp\_poll函数

                        ----> poll\_wait

                               ----> ep\_ptable\_queue\_proc

                                      ----> init\_waitqueue\_func\_entry 注册回调函数到ep\_poll\_callback到epoll的等待队列，当设备有数据时，回调函数将对应epitem挂入epoll的rdllist

                                      ----> add\_wait\_queue

                 -----> ep\_rbtree\_insert 将新的epoll节点插入红黑树

          ----> (EPOLL\_CTL\_DEL:ep\_remove) 删除操作

          ----->  (EPOLL\_CTL\_MOD:ep\_modify) 修改操作

sys\_epoll\_wait  读取epoll上的事件

          ----> ep\_poll

               ----> ep\_events\_transfer

                      ----> ep\_collect\_ready\_items 循环rdllist队列，将ready的epitem的txlink挂入txlist

                      ----> ep\_send\_events

                           ----> list\_for\_each 循环txlist

                           ----> epi->ffd.file->f\_op->poll 重新poll对应socket

                           ----> \_\_put\_user 将socket上的event拷贝到用户空间

对于POLL的两处瓶颈：

1. 每次调用POLL都要拷贝fd数组到内核空间，如果数量巨大则会造成性能瓶颈。对于EPOLL，fd通过epoll\_ctrl传入内核并保存在红黑树种，以后epoll\_wait不够用再次拷贝，规避了拷贝的瓶颈。

2. POLL每次调用都要循环调用所有fd的poll操作，同样在fd数量巨大时会造成瓶颈。对于EPOLL，poll操作只会在有数据的fd上进行（异步操作，通过回调函数），并且poll操作不会被epoll\_wait时被调用，epoll\_wait只是读取当前ready的fd的事件。

            ----> sock\_map\_fd  为socket创建file结构接分配文件号，并建立文件与socket之间的关系

                  ----> get\_unused\_fd  获取当前未用的文件号

                  ----> get\_empty\_filp 为file结构获取内存

                  ----> d\_alloc  为该socket文件创建dentry结构

                  ----> d\_add 将创建的dentry与创建的inode关联

                  ----> fd\_install 将创建的file结构与获取的文件号关联，即将file结构按fd下标挂入当前进程的打开文件数组current->files中

**22. Linux锁实现**

<http://www.searchtb.com/2011/01/pthreads-mutex-vs-pthread-spinlock.html>

**23. Linux Domain Socket实现**

* **Create Socket**

sys\_socket

  ----> sock\_create  创建并设置socket 结构

       ----> \_\_sock\_create

           ----> sock\_alloc 为socket结构分配内存

               ----> new\_inode 调用sock\_mnt的alloc\_inode创建inode即sockfs\_ops.alloc\_inode，对socket来说他分配了一个socket\_alloc结构

----> pf->create = unix\_create

               ----> unix\_create1

----> sk\_alloc

----> sk\_prot\_alloc

              ----> sock\_init\_data  初始化sock结构中的队列等成员

----> sock\_map\_fd

* **Bind Socket**

sys\_bind

   ----> sockfd\_lookup\_light根据socket号查找对应的socket结构

----> fget\_light

----> sock\_from\_file

   ----> sock->ops->bind对PF\_UNIX的SOCK\_STREAM 来说是unix\_stream\_ops.bind即unix\_bind

----> unix\_mkname

----> kern\_path\_parent

----> lookup\_create 创建Dentry

----> fput\_light

* **Send data over Socket**

**sys\_send**

----> sys\_sendto 将缓冲中的数据通过指定socket发送出去

----> sockfd\_lookup\_light 通过fd查找对应的socket结构

----> sock\_sendmsg 通过socket发送消息

----> \_\_sock\_sendmsg

----> sock->ops->sendmsg 对PF\_UNIX来说即为unix\_stream\_ops.sendmsg = unix\_stream\_sendmsg

----> scm\_send

----> sk->sk\_prot->get\_port = inet\_csk\_get\_port 获取未使用端口作为源端口

----> sock\_alloc\_send\_skb

----> sock\_alloc\_send\_pskb

----> alloc\_skb

----> alloc\_pages

----> unix\_scm\_to\_skb 将数据拷贝到SKB的buffer中

----> skb\_queue\_tail 将SKB挂入到目标socket的sk\_receive\_queue队尾

----> other->sk\_data\_ready = sock\_def\_readable 调用data\_ready函数唤醒目标socket等待队列