# 第一章概述

## 1.1 linux内核双链表结构

### 1.1.1原理

1）内核的链表的优点在于，如果仅仅在头部做插入与删除 就是一个栈，如果头部插入，尾部删除就是一个队列。也可以退化一个二叉树。

2）函数解释

### 1.1.1.2.1

//链表结构体

struct list\_head {

struct list\_head \*next, \*prev;

};

//初始化一个链表

static inline void init\_list\_head(struct list\_head \*list)

{

list->next = list;

list->prev = list;

}

//添加一个节点进入链表—头部插入--

static void \_\_list\_add(struct list\_head \*new, struct list\_head \*prev ,struct list\_head \*next)

{

next->prev = new;

new->next = next;

new->prev = prev;

prev->next = new;

}



//遍历所有的结构体

#define list\_for\_each(pos ,head) \

for(pos = (head)->next ; pos != (head) ;pos = pos->next)

//求出在结构体中的偏移量

#define offsetof(type, member) ((size\_t) &((type \*)0)->member)

//计算结构体的首地址

#define container\_of(ptr, type, member) ({ \

const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \

(type \*)( (char \*)\_\_mptr - offsetof(type,member) );})

//计算结构体的首地址

#define list\_entry(ptr, type, member) \

container\_of(ptr, type, member)

//list\_for\_each\_entry = list\_for\_each + list\_entry

#define list\_for\_each\_entry(pos, head, member) \

for (pos = list\_entry((head)->next, typeof(\*pos), member); \

&pos->member != (head); \

pos = list\_entry(pos->member.next, typeof(\*pos), member))

//对于list\_for\_each\_entry的安全操作 (n和pos属于同一种类型)

#define list\_for\_each\_entry\_safe(pos, n, head, member) \

for (pos = list\_entry((head)->next, typeof(\*pos), member), \

n = list\_entry(pos->member.next, typeof(\*pos), member); \

&pos->member != (head); \

pos = n, n = list\_entry(n->member.next, typeof(\*n), member))

### 1.1.2 实验

## 1.2 linux内核中的哈希表

### 1.2.1原理

### 1.2.2 实验

## 总结:

# 第二章进程管理

## 2.1进程创建

正文

## 2.2 进程调度

正文

## 面试总结:

# 第三章内存管理

## 3.1页面机制

一级页表

### 3.3.1一级页表

#### 3.3.1.1 32位一级页表

32位一级页表的缺陷：

页面4k=2^12(页内偏移至少需要12位)

对于32位机器如下理解：4G的内存空间(虚拟内存空间)--页表个数=(2^20)-4G(2^32)/4K(2^12)

需要存储2^20个页表项，一个页表项占用32位（4字节）总共需要内存：2^20\*4(4M)内存

也就是一个进程占用4M的大小，如果多个进程引起更大的空间占用浪费。

|  |
| --- |
| 0 |
| 1 |
| 2 |
| ... |
| 2^20-1 |

一级页表：

|  |  |  |
| --- | --- | --- |
| 序号 | 31~12位 | 11~0位 |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |
| .... |  |  |
| 2^20-1 |  |  |

#### 3.3.1.2 64位一级页表

### 3.3.2二级页表

#### 3.3.2.1 32位系统二级页表

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| ... |
| ... |
| 2^20-1 |

二级页表：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 10位(dir) | 10位(pag) | 12位（offset） |
| 序号 | 31~22 | 21~12 | 11~0 |
| 0 | 0 | 1 | 12 |
| 1 |  |  |  |
| 2 |  |  |  |
| .... |  |  |  |
| .... |  |  |  |
| 1024 |  |  |  |

二级页表原理：将一级页表的20位拆分两部分：10（第一层）+10（第二层）

一个进程需要的最小内存页表是4K=2^10 + 2^10+ 2^12，

在PCB中可以获取一级页目录的开始位置，系统初始化建立一个对应关系。

这样的页目录需要连续的4M空间，

将页目录拆分如下图：

|  |  |
| --- | --- |
| 0#页表 | |
| 0 | … |
| 1 | 762 |
| 1023 | … |

|  |  |
| --- | --- |
| 存放数据 | 物理块号 |
|  | 1 |
|  | 2 |
| 0#页表位置 | 3 |
|  | … |
|  | 762 |
|  | ... |
|  | 2^20-1 |

|  |  |
| --- | --- |
| 页表号 | 内存块号 |
| 0 | 3 |
| 1 | … |
| … | … |
| 1023 | ….. |

|  |  |
| --- | --- |
| 1#页表 | |
| 0 | …. |
| ... |  |
|  |  |
| 1023 | … |

|  |  |
| --- | --- |
| 页号 | 块号 |
| 0 | 2 |
| 1 | 4 |
| ... | ... |
| 1023 | 762 |
| ..... | ... |
| 2^20-1 | .... |

|  |  |
| --- | --- |
| 2#页表 | |
| 0 | … |
| ... |  |
| 1023 | … |



比如：32位系统 一个0000000000 0000000001 000000001111 逻辑地址转换物理地址？

762\*1024+(1111)偏移

#### 3.3.2.2 64位系统多级页表

64位系统需要页表偏移是2^64 = 2^34G = 2^24T 但是虚拟并不是2^34内存空间（如果这么大有点吓人）。注意一个概念long( 32位4字节 && 64位8字节 )

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [63:48] | [47:39] 9 | [38:30] 9 | [29:21] 9 | [20:12] 9 | [11:0]12 |
| 预留 | PGD | PUD | PMD | PTE | Offset |

如果：[63]=1 内核空间地址 && [63]=0 用户空间地址

在linux划分中各占用9位，各有2^9=512个表项



#define PAGE\_SHIFT 12 //也就是偏移量 2^12=4K(一个页面的大小)

问题：1、如何求出PGD基地址？

2、00 01 01 01 01 02 ？物理地址是多少？

00首位”0” 用户空间地址

01(PGD=000000001)索引1

FF 01 01 01 01 02 ？物理地址是多少？

## 3.3 虚拟地址转换物理地址

物理页面的切换就是取出mm中的PGD参数放置在cr3寄存器中。

pgd\_offset() 求出的页目录项低12位没有用保存的是页目录项的一些属性。



[root@localhost ~]# dmesg -c

[ 289.469352] \*\*\*\*\*\*\*start\*\*\*\*\*\*\*\*

[ 289.469356] cr0= 80050033 cr3= 7b3e0000

[ 289.469359] PGDIR\_SHIFT = 39

[ 289.469362] PUD\_SHIFT = 30

[ 289.469364] PMD\_SHIFT = 21

[ 289.469366] PAGE\_SHIFT = 12

[ 289.469369] PTRS\_PER\_PGD = 512

[ 289.469371] PTRS\_PER\_PUD = 512

[ 289.469373] PTRS\_PER\_PMD = 512

[ 289.469375] PTRS\_PER\_PTE = 512

[ 289.469378] PAGE\_MASK = fffffffffffff000

[ 289.469382] vaddr = ffff99fdf65df000

[ 289.469385] pgd\_val = 4051067 , pgd\_index = 133

[ 289.469387] pud\_val = 4055067 , pud\_index = 1f7

[ 289.469390] pmd\_val = 7648a063 , pmd\_index = 1b2

[ 289.469392] pte\_val = 80000000765df063 , pte\_index = 1df

[ 289.469395] page\_addr = 80000000765df000 , page\_offset = 0

[ 289.469397] vaddr = ffff99fdf65df000 , paddr = 80000000765df000

下面是内存的真是存在数值:

第一步：寄存器cr3+PGD\*8B = 0x7b3e0000+0x133\*8 = 0x7b3e998

0x7b3e998地址存放的数据0x4051067

000007B3E0980 0000000000000000 0000000000000000 ................

000007B3E0990 0000000000000000 0000000004051067 ........g.......

000007B3E09A0 0000000000000000 0000000000000000 ................

第二步：基址0x4051000+0x1f7\*8 = 0x4051fb8

0x4051fb8地址存放的数据是0x4055067

0000004051FA0 0000000000000000 0000000000000000 ................

0000004051FB0 0000000004052067 0000000004055067 g......gP......

0000004051FC0 0000000000000000 0000000000000000 ................

第三步：0x4055000+0x1b2\*8=0x4055D90

0x4055D90地址存放的数据是0x7648A063

0000004055D80 80000000760000E3 000000007638C063 ...v....c.8v....

0000004055D90 000000007648A063 00000000327E8063 c.Hv....c.~2....

0000004055DA0 80000000768000E3 8000000076A000E3 ...v.......v....

第四步：0x7648A000+0x1df\*8 = 0x7648Aef8

0x7648Aef8地址存放的数据是0x765DF063

000007648AEE0 80000000765DC063 80000000765DD063 c.]v....c.]v....

000007648AEF0 80000000765DE063 80000000765DF063 c.]v....c.]v....

000007648AF00 80000000765E0063 80000000765E1063 c.^v....c.^v....

第五步：0x765DF000+0x0\*0 =0x765DF000

00000765DEFF0 0000000000000000 0000000000000000 ...............

00000765DF000 726F77206C6C6568 206D6F726620646C hellworld from

00000765DF010 00006C656E72656B 0000000000000030 kernel..0.......

真实的物理地址是：0x765DF000

## 3.3 内存管理

引用地址：<http://kerneltravel.net>

#### 3.3.3.1内存管理mm\_struct

mm\_struct 被称为内存描述符，对整个进程的空间的描述，但是mm\_struct包含多个vm\_area\_struct（虚拟区域）如下图所示：



bss属于所有未初始化存放的位置



struct mm\_struct {

struct vm\_area\_struct \* mmap; //指向虚拟区间(VMA)的链表

struct rb\_root mm\_rb; //指向线性区对象红黑树的根

struct vm\_area\_struct \* mmap\_cache; //指向最近找到的虚拟区间

unsigned long(\*get\_unmapped\_area) (struct file \*filp,

unsigned long addr, unsigned long len,

unsigned long pgoff, unsigned long flags);//在进程地址空间中搜索有效线性地址区

unsigned long(\*get\_unmapped\_exec\_area) (struct file \*filp,

unsigned long addr, unsigned long len,

unsigned long pgoff, unsigned long flags);

void(\*unmap\_area) (struct mm\_struct \*mm, unsigned long addr);//释放线性地址区间时调用的方法

unsigned long mmap\_base; /\* base of mmap area \*/

unsigned long task\_size; /\* size of task vm space \*/

unsigned long cached\_hole\_size;

unsigned long free\_area\_cache; //内核从这个地址开始搜索进程地址空间中线性地址的空闲区域

pgd\_t \* pgd; //指向页全局目录

atomic\_t mm\_users; //次使用计数器，使用这块空间的个数

atomic\_t mm\_count; //主使用计数器

int map\_count; //线性的个数

struct rw\_semaphore mmap\_sem; //线性区的读/写信号量

spinlock\_t page\_table\_lock; //线性区的自旋锁和页表的自旋锁

struct list\_head mmlist; //指向内存描述符链表中的相邻元素

/\* Special counters, in some configurations protected by the

\* page\_table\_lock, in other configurations by being atomic.

\*/

mm\_counter\_t \_file\_rss; //mm\_counter\_t代表的类型实际是typedef atomic\_long\_t

mm\_counter\_t \_anon\_rss;

mm\_counter\_t \_swap\_usage;

unsigned long hiwater\_rss; //进程所拥有的最大页框数

unsigned long hiwater\_vm; //进程线性区中最大页数

unsigned long total\_vm, locked\_vm, shared\_vm, exec\_vm;

//total\_vm 进程地址空间的大小(页数）

//locked\_vm 锁住而不能换出的页的个数

//shared\_vm 共享文件内存映射中的页数

unsigned long stack\_vm, reserved\_vm, def\_flags, nr\_ptes;

//stack\_vm 用户堆栈中的页数

//reserved\_vm 在保留区中的页数或者在特殊线性区中的页数

//def\_flags 线性区默认的访问标志

//nr\_ptes 进程的页表数

unsigned long start\_code, end\_code, start\_data, end\_data;

//start\_code 可执行代码的起始地址

//end\_code 可执行代码的最后地址

//start\_data已初始化数据的起始地址

// end\_data已初始化数据的最后地址

unsigned long start\_brk, brk, start\_stack;

//start\_stack堆的起始位置

//brk堆的当前的最后地址

//用户堆栈的起始地址

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

//arg\_start 命令行参数的起始地址

//arg\_end命令行参数的起始地址

//env\_start环境变量的起始地址

//env\_end环境变量的最后地址

unsigned long saved\_auxv[AT\_VECTOR\_SIZE]; /\* for /proc/PID/auxv \*/

struct linux\_binfmt \*binfmt;

cpumask\_t cpu\_vm\_mask; //用于惰性TLB交换的位掩码

/\* Architecture-specific MM context \*/

mm\_context\_t context; //指向有关特定结构体系信息的表

unsigned int faultstamp;

unsigned int token\_priority;

unsigned int last\_interval;

unsigned long flags; /\* Must use atomic bitops to access the bits \*/

struct core\_state \*core\_state; /\* coredumping support \*/

#ifdef CONFIG\_AIO

spinlock\_t ioctx\_lock; //用于保护异步I/O上下文链表的锁

struct hlist\_head ioctx\_list;//异步I/O上下文

#endif

#ifdef CONFIG\_MM\_OWNER

struct task\_struct \*owner;

#endif

#ifdef CONFIG\_PROC\_FS

unsigned long num\_exe\_file\_vmas;

#endif

#ifdef CONFIG\_MMU\_NOTIFIER

struct mmu\_notifier\_mm \*mmu\_notifier\_mm;

#endif

#ifdef CONFIG\_TRANSPARENT\_HUGEPAGE

pgtable\_t pmd\_huge\_pte; /\* protected by page\_table\_lock \*/

#endif

#ifdef \_\_GENKSYMS\_\_

unsigned long rh\_reserved[2];

#else

//有多少任务分享这个mm OOM\_DISABLE

union {

unsigned long rh\_reserved\_aux;

atomic\_t oom\_disable\_count;

};

/\* base of lib map area (ASCII armour) \*/

unsigned long shlib\_base;

#endif

};

#### 3.3.3.1内存管理vm\_area\_struct

是最小的内存管理结构，表示一段属性一样的内存区域。

struct vm\_area\_struct {

/\* The first cache line has the info for VMA tree walking.

第一个缓存行具有VMA树移动的信息\*/

unsigned long vm\_start; /\* vm\_mm内的起始地址。 \*/  
unsigned long vm\_end; /\* 在vm\_mm内结束地址之后的第一个字节的地址。 \*

/\* linked list of VM areas per task, sorted by address

每个任务的VM区域的链接列表，按地址排序\*/

struct vm\_area\_struct \*vm\_next, \*vm\_prev;

struct rb\_node vm\_rb;

/\*

此VMA左侧最大的可用内存间隙（以字节为单位）。

在此VMA和vma-> vm\_prev之间，

或者在VMA rbtree中我们下面的一个VMA与其->vm\_prev之间。

这有助于get\_unmapped\_area找到合适大小的空闲区域。

\*/

unsigned long rb\_subtree\_gap;

/\* Second cache line starts here.

第二个缓存行从这里开始\*/

struct mm\_struct \*vm\_mm; /\* 我们所属的address space\*/

pgprot\_t vm\_page\_prot; /\* 此VMA的访问权限 \*/

unsigned long vm\_flags; /\* Flags, see mm.h. \*/

/\*

对于具有地址空间（address apace）和后备存储(backing store)的区域，

链接到address\_space->i\_mmap间隔树，或者链接到address\_space-> i\_mmap\_nonlinear列表中的vma。

\*/

union {

struct {

struct rb\_node rb;

unsigned long rb\_subtree\_last;

} linear;

struct list\_head nonlinear;

} shared;

/\*

在其中一个文件页面的COW之后，文件的MAP\_PRIVATE vma可以在i\_mmap树和anon\_vma列表中。

MAP\_SHARED vma只能位于i\_mmap树中。

匿名MAP\_PRIVATE，堆栈或brk vma（带有NULL文件）只能位于anon\_vma列表中。

\*/

struct list\_head anon\_vma\_chain; /\* Serialized by mmap\_sem & \* page\_table\_lock

由mmap\_sem和\* page\_table\_lock序列化\*/

struct anon\_vma \*anon\_vma; /\* Serialized by page\_table\_lock 由page\_table\_lock序列化\*/

/\* 用于处理此结构体的函数指针 \*/

const struct vm\_operations\_struct \*vm\_ops;

/\* 后备存储（backing store）的信息: \*/

unsigned long vm\_pgoff; /\* 以PAGE\_SIZE为单位的偏移量（在vm\_file中），\*不是\* PAGE\_CACHE\_SIZE\*/

struct file \* vm\_file; /\* 我们映射到文件（可以为NULL）\*/

void \* vm\_private\_data; /\* 是vm\_pte（共享内存） \*/

#ifndef CONFIG\_MMU

struct vm\_region \*vm\_region; /\* NOMMU映射区域 \*/

#endif

#ifdef CONFIG\_NUMA

struct mempolicy \*vm\_policy; /\* 针对VMA的NUMA政策 \*/

#endif

#### 3.3.3.1内存管理vm\_operations\_struct

#### 3.3.3.1 linux内存映射原理

为什么需要内存映射？

IO速度远远没有，读取虚拟内存的速度快。

页框号和物理内存有关系吗？

同一个虚拟地址，可以对应不同的物理地址吗？

如何查找物理内存有多少个page？每个page的地址是多少？

用户空间使用mmap()映射函数。

Mmap由用户空间映射一段文件（磁盘空间）的方法。

但是mmap的原理是什么呢？

猜测1：用户空间打开一个文件，计算出大小之后，申请页框，设置页框的属性不能被内核在使用。

虚拟内存空间如何映射文件的？

#### 3.3.3.1 linux内存映射实现

#### 3.3.3.1 linux内存映射测试

正文