# 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 linux内存映射基础（上）

#### 3.3.3.1 linux内存映射实现（中）

#### 3.3.3.1 linux内存映射测试（下）