内存之页表

目录

[1、 页表是什么 2](#_Toc99202724)

[2、 页面大小 2](#_Toc99202725)

[3、 页表的notes 3](#_Toc99202726)

[3.1、页表的作用 3](#_Toc99202727)

[3.2、页表的存放 3](#_Toc99202728)

[3.3、页表项中存放的是虚是实 3](#_Toc99202729)

[4、 页表的分级 3](#_Toc99202730)

[4.1、单级与多级页表 3](#_Toc99202731)

[4.2、优缺点 5](#_Toc99202732)

[4.3、快表 5](#_Toc99202733)

[4.4、arm64四级页表 7](#_Toc99202734)

[5、 页表的属性(特定于PTE的信息) 9](#_Toc99202735)

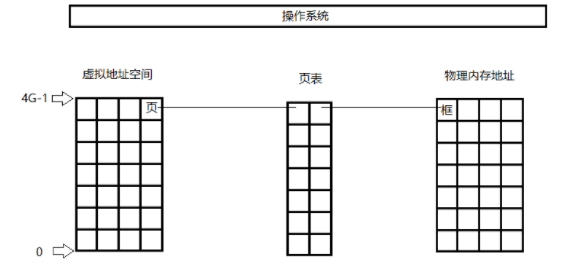
[6、 页表对应的内存管理 13](#_Toc99202736)

## 页表是什么

页表是一种特殊的数据结构，放在系统空间的页表区，用来存放逻辑页与物理页帧的对应关系。每一个进程都拥有一个自己的页表，PCB表中有指针指向页表。

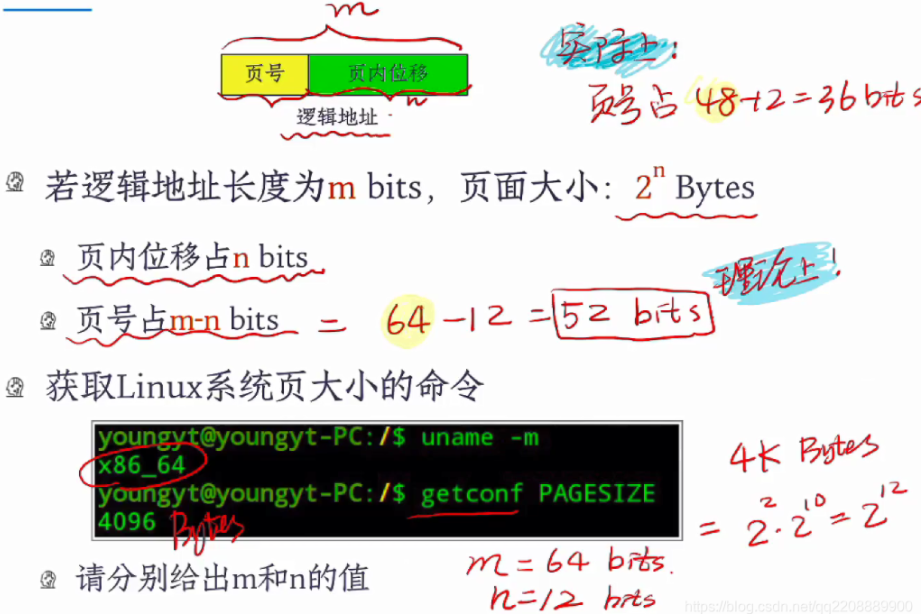
简单一句话：

带有权限属性的放在物理内存中，用来记录虚拟内存页与物理页映射关系的一张表。



## 页面大小

在64位系统中，页面大小为4K Bytes，即2的12次方，所以页内位移需要12位表示。那么页号也就是64 – 12 = 52位，但实际上只有用到其中的一部分。比如在内核中配置虚拟地址长度为48bit，则页号用到了48 – 12 = 36位。



## 页表的notes

### 3.1、页表的作用

1. 地址转换

将虚拟地址转换为物理地址

1. 权限管理

管理CPU对物理页的访问，如：读、写、可执行等权限

1. 隔离地址空间

隔离各个进程的地址空间，使其互不影响，提供系统的安全性

Note：

可以在不额外增加物理内存的情况下，将页换出到块设备来增加有效的可用内存空间。

### 3.2、页表的存放

页表存放在物理内存中，打开MMU后，如果需要修改页表，需要将页表所在的物理地址映射到虚拟地址才能访问页表(如内核初始化后会将物理内存线性映射，这样通过物理地址和虚拟地址的偏移就可以获得页表物理地址对应的虚拟地址)。

### 3.3、页表项中存放的是虚是实

页表基地址寄存器和各级页表中存放的都是物理地址，而不是虚拟地址。

## 页表的分级

### 4.1、单级与多级页表

在不同的架构体系中所使用的页表级数也不相同。页表为什么要分级？

Arm通常使用二级页表，arm64通常使用三级页表。

此处拿arm举例说明，

32位arm系统所支持的最大访问空间为4GB，页面大小为4KB。

<一级页表>



页内偏移为低12bit，页号占高20bit



若将4GB空间映射需要4GB/4KB个页表条目，每个条目占据4Byte则单个进程的4GB空间所对应的页表需要消耗4MB的物理内存。在多进程中物理内存将极大的被消耗掉。另外页表是不能被拆分的，因此页表的存放必须是物理地址连续的4MB空间(1KB 连续页框)。随着业务程序的不断运行，系统内存页将逐渐碎片化，很难在需要时找到一块如此大的连续内存，这种严苛的条件往往很难达到。

<二级页表>

若是采用二级页表，则逻辑地址应该怎样分段呢？i、j、k、m以及n分别是什么？



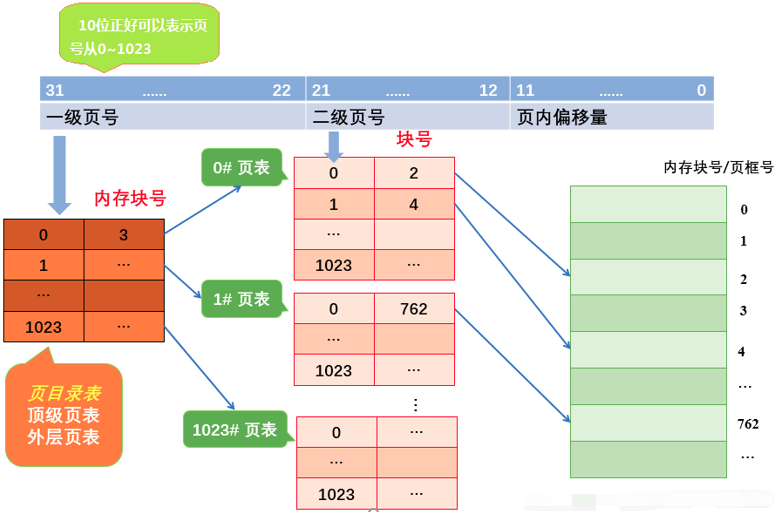
页面大小仍然使用最常用的4KB，因此页内偏移是0~11；

32位系统中页表项(条目)大小为4Byte，因此一页最多可存放1024个页表项，因此以10bit来划分高位的20bit页号段；j~k=12~21；m~n=22~31。

采用二级页表时，对应规则就会改变，二级页表将对应到物理内存上的4KB大小的页，一级页表此时变为映射物理地址为4MB。在实际的内存访问中，先找到一级页表，一级页表再和二级页表进行结合，二级页表相当于一级页表4MB分成了1024个4KB，找完后二级页表充当了offset的角色，此时定位到具体的4KB的页面，再用逻辑地址中的页内偏移offset找到具体的物理地址。

这样一个进程浪费掉的空间是一级页表占用的(4GB/4MB)\*4byte=4KB，二级页表浪费掉的是1kB (一个一级页表拥有的二级页表数量)\* 1kB(一级页表的数目4GB/4MB) \*4byte=4MB，加起来是4MB + 4KB，比光用一级页表要多4KB，但是二级页表是可以不存在的，比如此时程序只用了20%的页，那么4MB就需要乘以20%，这样一下内存消耗就比只用一级页表时少。

二级页表如何实现地址转换



大致流程：

(1) 按照地址结构将逻辑地址拆成三个部分。

(2) 从PCB中读取页目录起始地址，再根据一级页号查页目录表，找到下一级页表在内存中存放位置。

(3) 根据二级页号查表，找到最终想要访问的内存块号。

(4) 结合页内偏移量得到物理地址。

### 4.2、优缺点

1、一级页表的优缺点

优点：只需要2次访问内存(一次访问页表，一次访问数据)，效率高，实现简单

缺点：需要连续的大块内存存放每个进程的页表，浪费内存，虚拟内存越大页表越大，内存碎片化的时候很难分配到连续大块内存，大多数虚拟内存并没有使用。

2、多级页表的优缺点

优点：节省内存、可以按需分配各级页表、可以离散存储页表

缺点：需要遍历多级页表，需要多次访问内存，实现复杂度高。

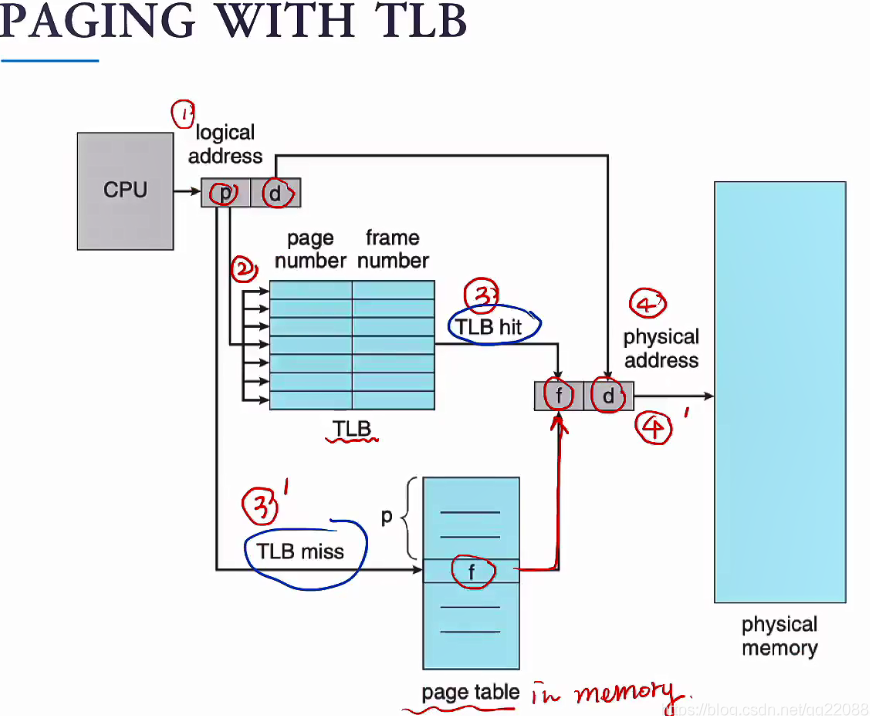
### 4.3、快表

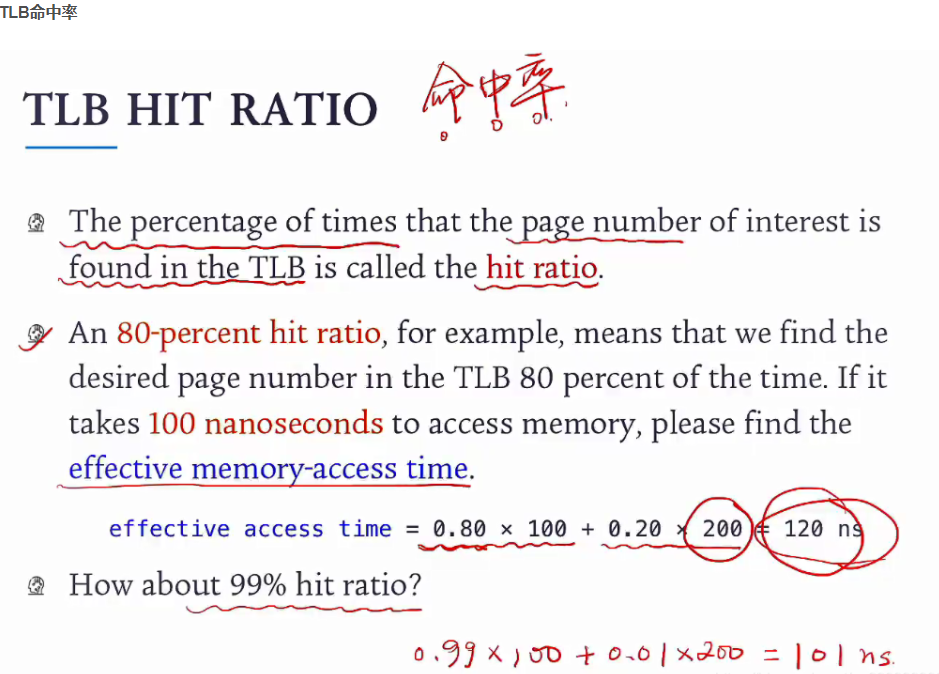
从上述内容可以看到在多级页表中，CPU预访问一个内存地址所需要的步骤比较多，如此带来的是性能的降低。为了提高访问内存的性能，硬件设计在MMU中增加了TLB。

TLB是一种很小但查找速度很快的缓存，TLB配合页表一起使用：

TLB存放了很多进程中的一部分页表条目(最近访问的)，这些条目就称为快表

当CPU转换逻辑地址时，会去TLB中进行查找，如果找到了，那么立刻就能将地址转换完成；如果没有找到，那么需要对内存进行多次访问来将逻辑地址转换为物理地址。



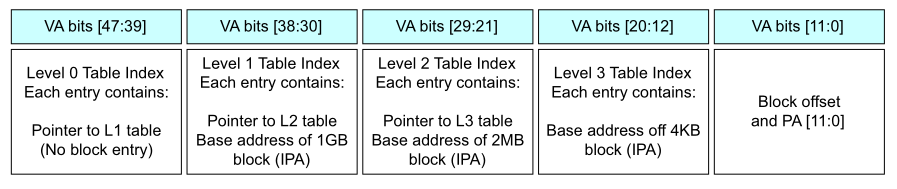


### 4.4、arm64四级页表

Arm64体系中虚拟地址为64bit，理论可访问的最大地址空间为2^64，这是一个超大的操作空间，现代的计算法并不会使用如此巨量的地址，在实际的应用中嵌入式平台上一般会配置为39bit，对应三级页表(pgd->pmd->pte)，当虚拟地址长度配置为48bit时，则必须使用四级页表(pgd->pud->pmd->pte)。

下面以内核配置虚拟地址长度为48bit，PAGE\_TABLE为4举例。

整个的逻辑地址按照页表层级划分如下：



可以看到其是9:9:9:9:12的地址划分结构，上面也有提到地址如何划分，这里再赘述下：

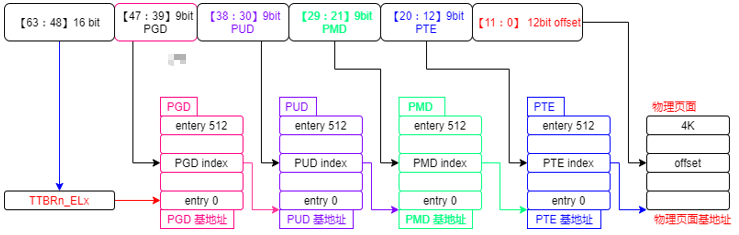
m = 逻辑地址位数 – 页内偏移量

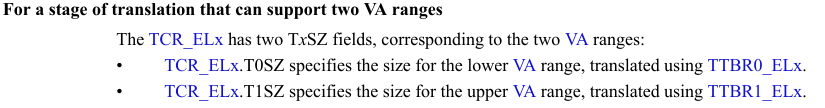
页大小为4KB，单个页表项占据8Byte，最大可容纳512个页表项。因此每级页表bit数不可超过9。

48bit ~ 63bit 在页表管理中是用不到的，但不是说没有意义。该段的数据代表着CPU访问的地址是用户空间还是内核空间，分别到TTBR0和TTBR1寄存器中获取Level0页表基地址。

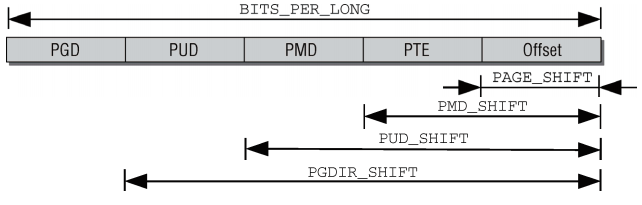
1. 47~39:Level 0页表中的索引，根据描述符中的内容获取Level 1页表基地址;
2. 38~30:Level 1页表中的索引，根据描述符中的内容获取Level 2页表基地址;
3. 29~21:Level 2页表中的索引，根据描述符中的内容获取Level 3页表基地址;
4. 20~12:Level 3页表中的索引，根据描述符中的内容获取物理地址的高36位，以4K地址对齐；
5. 11~00:是物理地址的偏移，结合获取的物理地址高位，最终得到物理地址。

内存的映射过程如下：





**4.5、四级页表的格式**



内核提供了4个数据结构（定义在page.h中）来表示页表项的结构

❑ pgd\_t用于全局页目录项。

❑ pud\_t用于上层页目录项。

❑ pmd\_t用于中间页目录项。

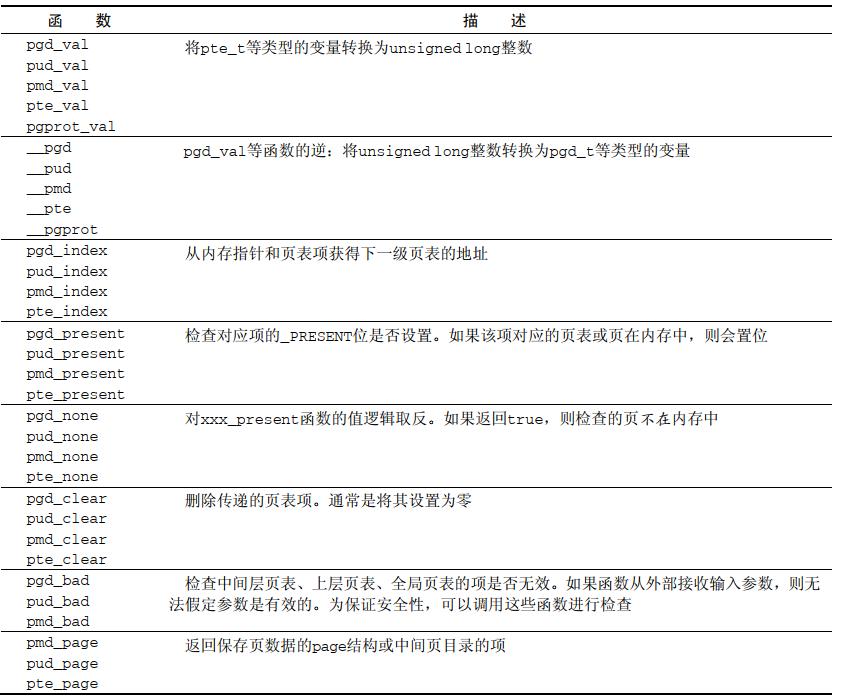
❑ pte\_t用于直接页表项。

PMD\_SHIFT指定了页内偏移量和最后一级页表项所需比特位的总数。该值减去PAGE\_SHIFT，可得最后一级页表项索引所需比特位的数目。更重要的是下述事实：该值表明了一个中间层页表项管理的部分地址空间的大小，即2^PMD\_SHIFT字节。

PUD\_SHIFT由 PMD\_SHIFT加 上 中 间 层 页 表 索 引 所 需 的 比 特 位 长 度 ， 而 PGDIR\_SHIFT则 由PUD\_SHIFT加上上层页表索引所需的比特位长度。对全局页目录中的一项所能寻址的部分地址空间长度计算以2为底的对数，即为PGDIR\_SHIFT。

在各级页目录/页表中所能存储的指针数目，也可以通过宏定义确定。 PTRS\_PER\_PGD指定了全局页目录中项的数目，PTRS\_PER\_PMD对应于中间页目录， PTRS\_PER\_PUD对应于上层页目录中项的数目，PTRS\_PER\_PTE则是页表中项的数目。

下表为分析页表项的函数



## 页表的属性(特定于PTE的信息)

在内存的访问过程中往往带有一些特殊的属性信息，比如可执行文件代码段映射的内存区是只读的，不允许用户去修改；一段内存可以同时映射到不同的程序中去使用；同一段物理地址可以分别映射成DEVICE和NORMAL，用于不同的场景中。这些属性权限都是记录在页表中，由页表在管理维护。

在不同的架构体系中页表项所占用的长度是不相同的，arm中是4Byte，arm64中是8Byte。如此仅看arm64架构的相关内容，本质与arm是相同的，只是相比arm可能有更丰富的管理属性。



在armv-X相关手册中都有定义属性类型，Linux内核源码中同样存在。

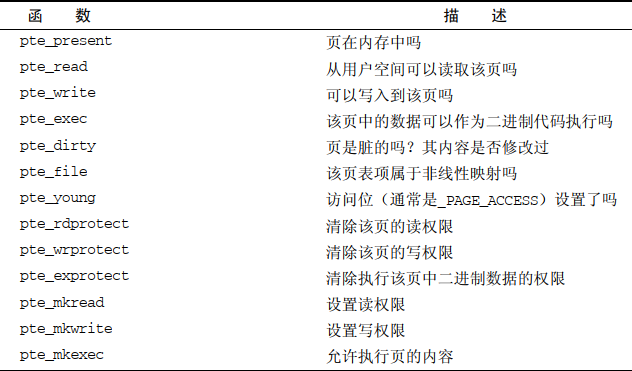
最后一级页表中的项不仅包含了指向页的内存位置的指针，还在上述的多余比特位包含了与页有关的附加信息。尽管这些数据是特定于CPU的，它们至少提供了有关访问控制的一些信息。

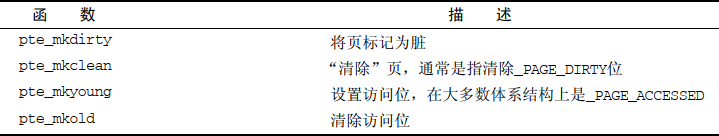
下列位在Linux内核支持的大多数CPU中都可以找到。  
❑\_PAGE\_PRESENT指定了虚拟内存页是否存在于内存中。页不见得总是在内存中，页可能换出到交换区。如果页不在内存中，那么页表项的结构通常会有所不同，因为不需要描述页在内存中的位置。相反，需要信息来标识并找到换出的页。  
❑CPU每次访问页时，会自动设置\_PAGE\_ACCESSED。内核会定期检查该比特位，以确认页使用的活跃程度（不经常使用的页，比较适合于换出）。在读或写访问之后会设置该比特位。  
❑\_PAGE\_DIRTY表示该页是否是“脏的”，即页的内容是否已经修改过。  
❑\_PAGE\_FILE的数值与\_PAGE\_DIRTY相同，但用于不同的上下文，即页不在内存中的时候。显然，不存在的页不可能是脏的，因此可以重新解释该比特位。如果没有设置，则该项指向一  
个换出页的位置。如果该项属于非线性文件映射，则需要设置\_PAGE\_FILE。  
❑如果设置了\_PAGE\_USER，则允许用户空间代码访问该页。否则只有内核才能访问（或CPU处  
于系统状态的时候）。  
❑\_PAGE\_READ、 \_PAGE\_WRITE和\_PAGE\_EXECUTE指定了普通的用户进程是否允许读取、写入、  
执行该页中的机器代码。内核内存中的页必须防止用户进程写入。但即使属于用户进程的页，也无法保证可以写入，这可能是有意如此，也可能是无意偶合。例如，其中可能包含了不能修改的可执行代码。对于访问权限粒度不那么细的体系结构而言，如果没有进一步的准则可区分读写访问权限，则会定义\_PAGE\_RW常数，用于同时允许或禁止读写访问。

IA-32和AMD64提供了\_PAGE\_BIT\_NX，用于将页标记为不可执行的（在IA-32系统上，只有启用了可寻址64 GiB内存的页面地址扩展（ page address extension， PAE)功能时，才能使用该保护位。例如，它可以防止执行栈页上的代码。否则，恶意代码可能通过缓冲区溢出手段在栈  
上执行代码，导致程序的安全漏洞。 NX位无法防止缓冲器溢出，但可以抑制其效果，因为进  
程会拒绝执行恶意代码。当然，如果体系结构本身对内存页提供了良好的访问授权设置，也  
可以实现同样的效果，某些处理器就是这样（令人遗憾的是，这些处理器不怎么常见）。

---内容摘录自深入Linux内核架构

用于处理内存页的体系结构相关状态的函数





<相关数据结构>

在内核代码pgtable-hwdef.h(arch/arm64/include/asm)中可见PTE相关定义

#define PTE\_VALID (\_AT(pteval\_t, 1) << 0)

#define PTE\_WRITE (PTE\_DBM) /\* same as DBM (51) \*/

#define PTE\_DIRTY (\_AT(pteval\_t, 1) << 55)

#define PTE\_SPECIAL (\_AT(pteval\_t, 1) << 56)

#define PTE\_PROT\_NONE (\_AT(pteval\_t, 1) << 58) /\* only when !PTE\_VALID \*/

#define PTE\_TYPE\_MASK (\_AT(pteval\_t, 3) << 0)

#define PTE\_TYPE\_FAULT (\_AT(pteval\_t, 0) << 0)

#define PTE\_TYPE\_PAGE (\_AT(pteval\_t, 3) << 0)

#define PTE\_TABLE\_BIT (\_AT(pteval\_t, 1) << 1)

#define PTE\_USER (\_AT(pteval\_t, 1) << 6) /\* AP[1] \*/ 用户空间可访问

#define PTE\_RDONLY (\_AT(pteval\_t, 1) << 7) /\* AP[2] \*/ 只读

#define PTE\_SHARED (\_AT(pteval\_t, 3) << 8) /\* SH[1:0], inner shareable \*/

#define PTE\_AF (\_AT(pteval\_t, 1) << 10) /\* Access Flag \*/ 在内存中

#define PTE\_NG (\_AT(pteval\_t, 1) << 11) /\* nG \*/ 不可合并

#define PTE\_DBM (\_AT(pteval\_t, 1) << 51) /\* Dirty Bit Management \*/

#define PTE\_CONT (\_AT(pteval\_t, 1) << 52) /\* Contiguous range \*/

#define PTE\_PXN (\_AT(pteval\_t, 1) << 53) /\* Privileged XN \*/

#define PTE\_UXN (\_AT(pteval\_t, 1) << 54) /\* User XN \*/ 用户无执行权限

#define PTE\_HYP\_XN (\_AT(pteval\_t, 1) << 54) /\* HYP XN \*/

/\*

\* Memory types available.

\*/

#define MT\_DEVICE\_nGnRnE 0

#define MT\_DEVICE\_nGnRE 1

#define MT\_DEVICE\_GRE 2

#define MT\_NORMAL\_NC 3

#define MT\_NORMAL 4

#define MT\_NORMAL\_WT 5

<相关功能接口定义>

#define pte\_pfn(pte) (\_\_pte\_to\_phys(pte) >> PAGE\_SHIFT)

#define pfn\_pte(pfn,prot) \

\_\_pte(\_\_phys\_to\_pte\_val((phys\_addr\_t)(pfn) << PAGE\_SHIFT) | pgprot\_val(prot))

#define pte\_none(pte) (!pte\_val(pte))

#define pte\_clear(mm,addr,ptep) set\_pte(ptep, \_\_pte(0))

#define pte\_page(pte) (pfn\_to\_page(pte\_pfn(pte)))

/\*

\* The following only work if pte\_present(). Undefined behaviour otherwise.

\*/

#define pte\_present(pte) (!!(pte\_val(pte) & (PTE\_VALID | PTE\_PROT\_NONE)))

#define pte\_young(pte) (!!(pte\_val(pte) & PTE\_AF))

#define pte\_special(pte) (!!(pte\_val(pte) & PTE\_SPECIAL))

#define pte\_write(pte) (!!(pte\_val(pte) & PTE\_WRITE))

#define pte\_user\_exec(pte) (!(pte\_val(pte) & PTE\_UXN))

#define pte\_cont(pte) (!!(pte\_val(pte) & PTE\_CONT))

#define pte\_hw\_dirty(pte) (pte\_write(pte) && !(pte\_val(pte) & PTE\_RDONLY))

#define pte\_sw\_dirty(pte) (!!(pte\_val(pte) & PTE\_DIRTY))

#define pte\_dirty(pte) (pte\_sw\_dirty(pte) || pte\_hw\_dirty(pte))

#define pte\_valid(pte) (!!(pte\_val(pte) & PTE\_VALID))

#define pte\_valid\_not\_user(pte) \

((pte\_val(pte) & (PTE\_VALID | PTE\_USER)) == PTE\_VALID)

#define pte\_valid\_young(pte) \

((pte\_val(pte) & (PTE\_VALID | PTE\_AF)) == (PTE\_VALID | PTE\_AF))

#define pte\_valid\_user(pte) \

((pte\_val(pte) & (PTE\_VALID | PTE\_USER)) == (PTE\_VALID | PTE\_USER))

static inline pte\_t pte\_wrprotect(pte\_t pte)

{

pte = clear\_pte\_bit(pte, \_\_pgprot(PTE\_WRITE));

pte = set\_pte\_bit(pte, \_\_pgprot(PTE\_RDONLY));

return pte;

}

static inline pte\_t pte\_mkwrite(pte\_t pte)

{

pte = set\_pte\_bit(pte, \_\_pgprot(PTE\_WRITE));

pte = clear\_pte\_bit(pte, \_\_pgprot(PTE\_RDONLY));

return pte;

}

static inline pte\_t pte\_mkclean(pte\_t pte)

{

pte = clear\_pte\_bit(pte, \_\_pgprot(PTE\_DIRTY));

pte = set\_pte\_bit(pte, \_\_pgprot(PTE\_RDONLY));

return pte;

}

static inline pte\_t pte\_mkdirty(pte\_t pte)

{

pte = set\_pte\_bit(pte, \_\_pgprot(PTE\_DIRTY));

if (pte\_write(pte))

pte = clear\_pte\_bit(pte, \_\_pgprot(PTE\_RDONLY));

return pte;

}

static inline pte\_t pte\_mkold(pte\_t pte)

{

return clear\_pte\_bit(pte, \_\_pgprot(PTE\_AF));

}

static inline pte\_t pte\_mkyoung(pte\_t pte)

{

return set\_pte\_bit(pte, \_\_pgprot(PTE\_AF));

}

static inline pte\_t pte\_mkspecial(pte\_t pte)

{

return set\_pte\_bit(pte, \_\_pgprot(PTE\_SPECIAL));

}

static inline pte\_t pte\_mkcont(pte\_t pte)

{

pte = set\_pte\_bit(pte, \_\_pgprot(PTE\_CONT));

return set\_pte\_bit(pte, \_\_pgprot(PTE\_TYPE\_PAGE));

}

static inline pte\_t pte\_mknoncont(pte\_t pte)

{

return clear\_pte\_bit(pte, \_\_pgprot(PTE\_CONT));

}

static inline pte\_t pte\_mkpresent(pte\_t pte)

{

return set\_pte\_bit(pte, \_\_pgprot(PTE\_VALID));

}

static inline pmd\_t pmd\_mkcont(pmd\_t pmd)

{

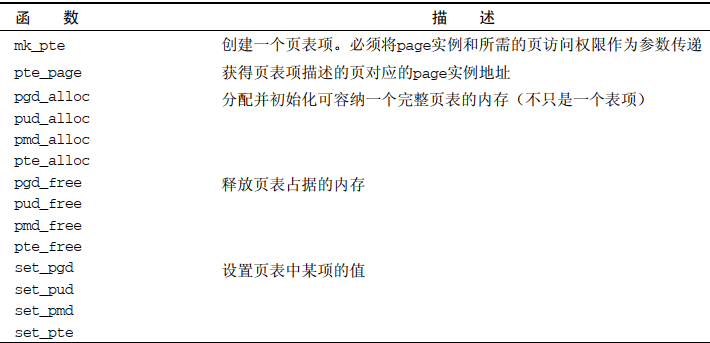
return \_\_pmd(pmd\_val(pmd) | PMD\_SECT\_CONT);

}

## 页表对应的内存管理

上面有提到页表是存放在物理内存中的，因此页表的建立也是需要消耗内存的，其对应的内存分配有专门的接口函数。

用于创建新页表项的函数



<接口定义>

pgd\_t \*pgd\_alloc(struct mm\_struct \*mm)

{

if (PGD\_SIZE == PAGE\_SIZE)

return (pgd\_t \*)\_\_get\_free\_page(PGALLOC\_GFP); //页申请

else

return kmem\_cache\_alloc(pgd\_cache, PGALLOC\_GFP);

}

void pgd\_free(struct mm\_struct \*mm, pgd\_t \*pgd)

{

if (PGD\_SIZE == PAGE\_SIZE)

free\_page((unsigned long)pgd);

else

kmem\_cache\_free(pgd\_cache, pgd);

}

static inline pmd\_t \*pmd\_alloc\_one(struct mm\_struct \*mm, unsigned long addr)

{

return (pmd\_t \*)\_\_get\_free\_page(PGALLOC\_GFP);

}

static inline pgtable\_t pte\_alloc\_one(struct mm\_struct \*mm, unsigned long addr)

{

struct page \*pte;

pte = alloc\_pages(PGALLOC\_GFP, 0);

if (!pte)

return NULL;

if (!pgtable\_page\_ctor(pte)) {

\_\_free\_page(pte);

return NULL;

}

return pte;

}

pmd\_populate是什么意思？