

**页表技术**是理解Linux内核虚拟内存管理的核心，linux内存管理各个组件主要是围绕着页表来隔离各个进程的地址空间，掌握页表技术原理，是理解内存管理的各个组件的基础。

1. 什么是MMU，使用MMU的好处；

内存管理单元(Memory Management Unit)，负责将虚拟地址转化为物理地址，并进行权限管理；

好处：隔离用户地址空间和内核地址空间，隔离不同进程的地址空间。恶意进程无法访问其他进程及内核的地址空间；

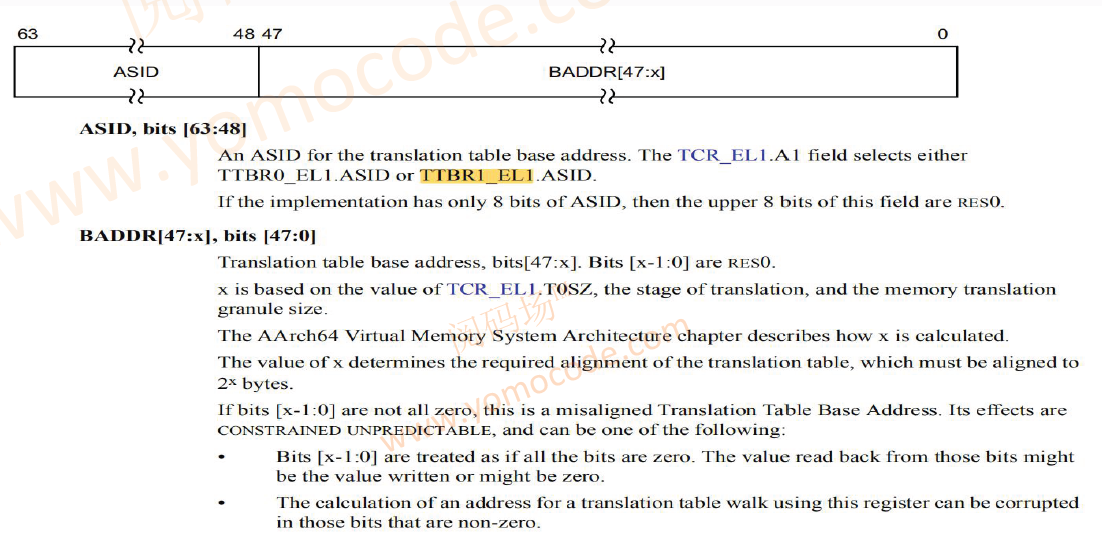
1. 什么是页表：**将虚拟地址映射到物理地址**的**数据结构**叫做页表，页表中存放的是虚拟地址中对应的物理页帧号和访问权限等信息；
2. 什么是tlb：页表缓存，缓存最近使用的页表项；
3. 使用多级页表的好处；按需分配，节省内存；
4. 硬件做什么；遍历页表从VA->PA
5. 软件做什么；填写页表；

2.ARM64虚拟地址空间

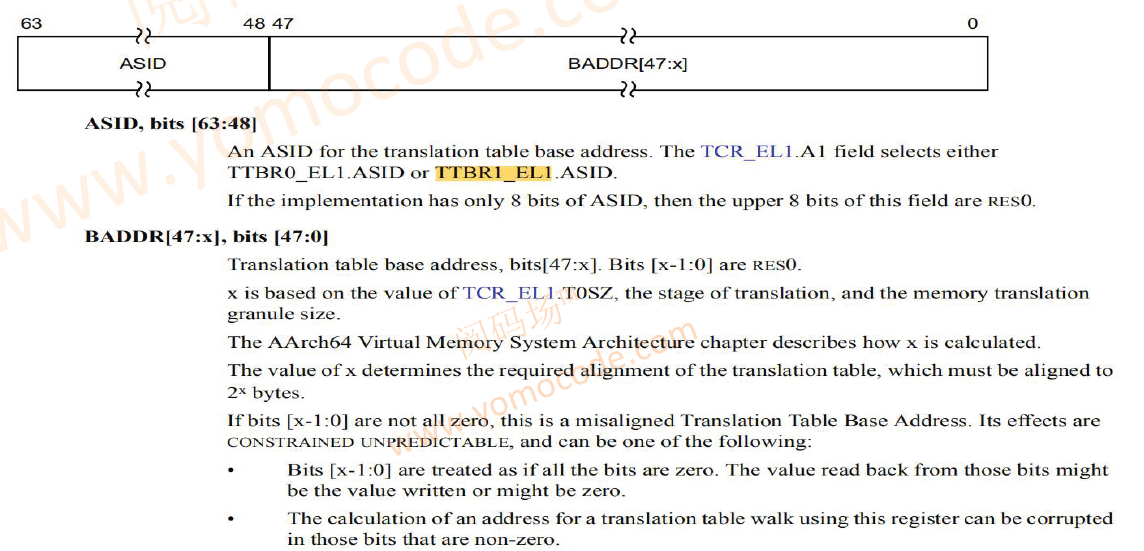


**相关重要寄存器**

ttbr0\_el1 转换表基址寄存器，存放**用户空间**页表及地址；



Ttbr1\_el1 转换表基址寄存器，存放**内核空间**页表及地址；



Sctlr\_dl1 系统控制寄存器el1，使能mmu

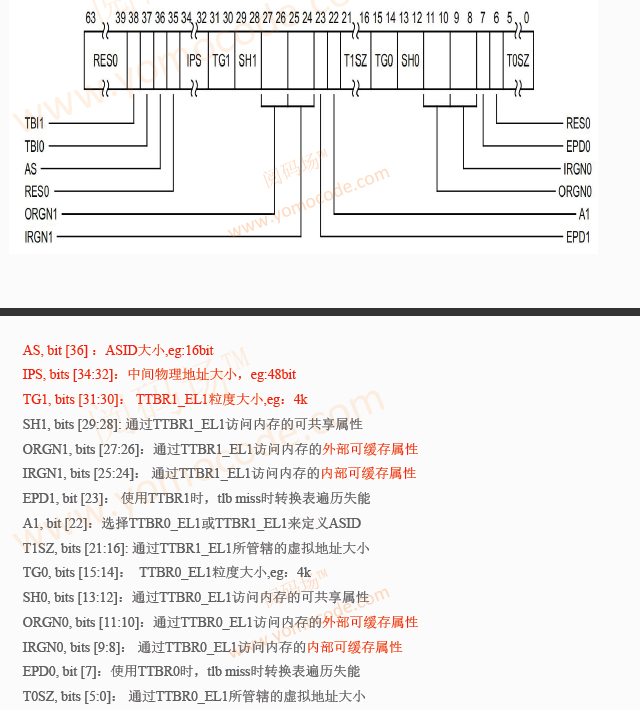


CPU启动过程，Bootload的数据可能被缓存，bootload指令与内核指令在不同地址；

指令cache，可关可不关；

**数据cache：必须关闭；**

TCR\_EL1 转换控制寄存器EL1，控制转换格式等



2，ARM64页表结构

2.1arm64页表格式

Arm64处理器将页表称为转化表(translation table),最多支持4级转换表，该表项称为描述符；

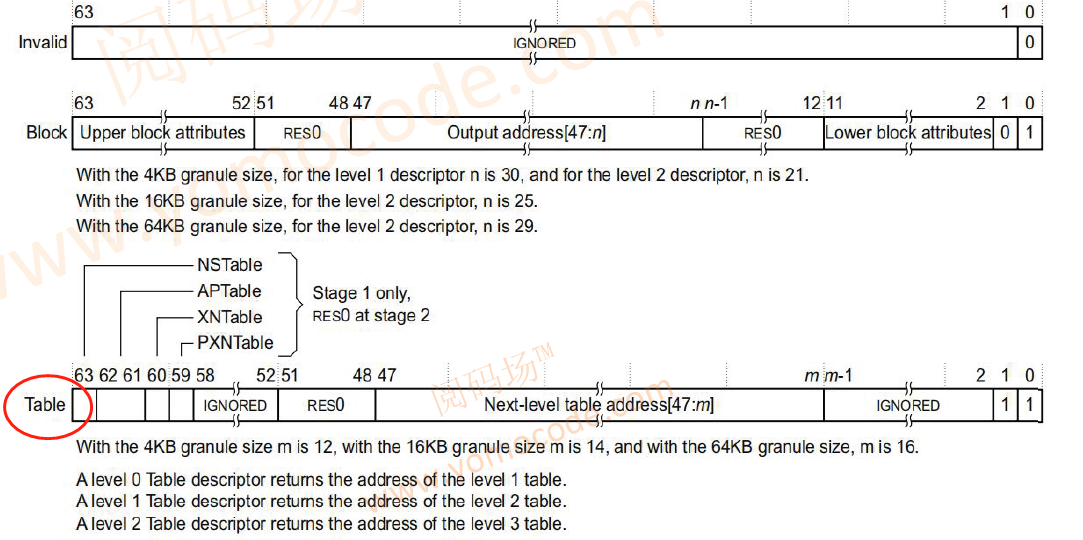
最多支持3种页长度：4K,16K, 64K；

描述符第0位表示描述符是否有效，1bit表示描述符类型；

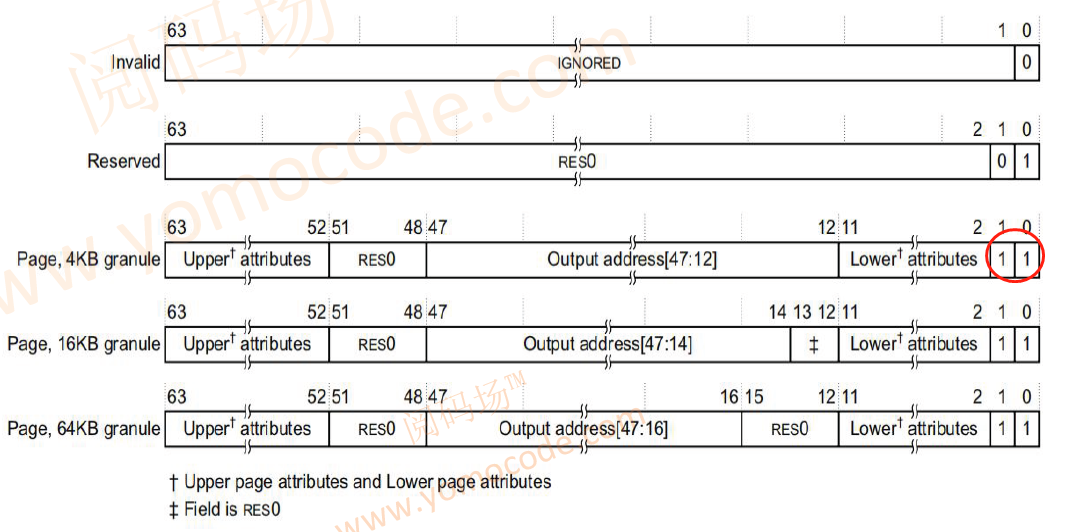
0-2级转换表，第1bit为0表示块描述符，1表示表描述符；

3级转换表，第1bit为0表示保留，1表示页描述符；

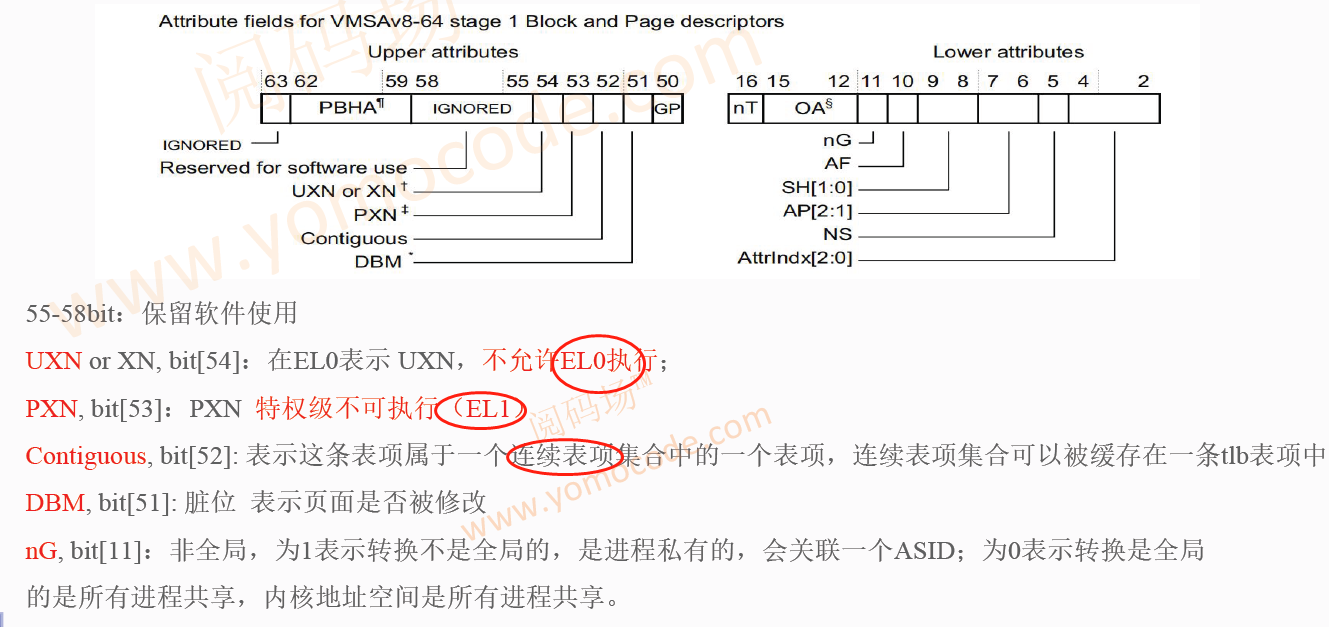
块描述符，常用于实现巨型页或段映射，表描述符存放下一级转换表地址(物理地址)，页描述符存放物理页帧号；

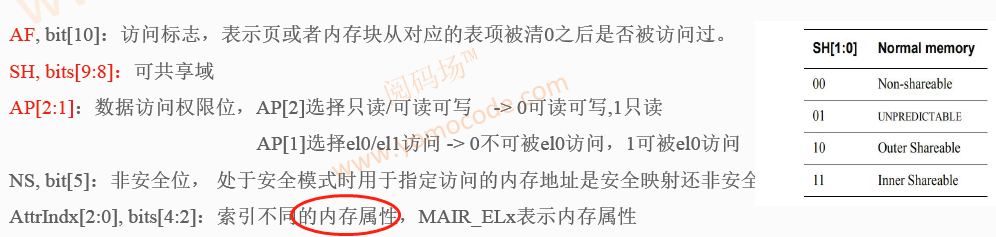


3级转换表格式



2.2页表属性

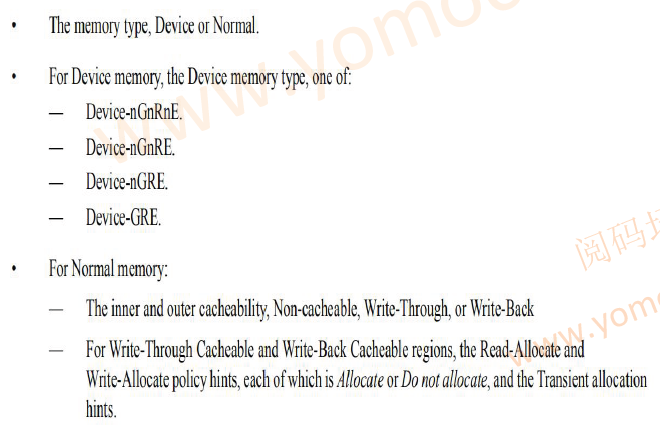




2.3内存属性

Normal Memory：普通内存，如DDR,SRAM;

Device Memory：设备内存，如内存映射IO寄存器，总是non cacheable的，而且是outer shareable的(CPU,DMA全局共享)；



/\*

\* Memory types available.

\*/

#define MT\_DEVICE\_nGnRnE 0

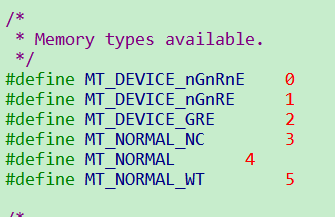
#define MT\_DEVICE\_nGnRE 1

#define MT\_DEVICE\_GRE 2

#define MT\_NORMAL\_NC 3

#define MT\_NORMAL 4

#define MT\_NORMAL\_WT 5



/\*

\* Memory region attributes for LPAE:

\*

\* n = AttrIndx[2:0]

\* n MAIR

\* DEVICE\_nGnRnE 000 00000000

\* DEVICE\_nGnRE 001 00000100

\* DEVICE\_GRE 010 00001100

\* NORMAL\_NC 011 01000100

\* NORMAL 100 11111111

\* NORMAL\_WT 101 10111011

\*/

ldr x5, =MAIR(0x00, MT\_DEVICE\_nGnRnE) | \

MAIR(0x04, MT\_DEVICE\_nGnRE) | \

MAIR(0x0c, MT\_DEVICE\_GRE) | \

MAIR(0x44, MT\_NORMAL\_NC) | \

MAIR(0xff, MT\_NORMAL) | \

MAIR(0xbb, MT\_NORMAL\_WT)

msr mair\_el1, x5

如何划分shareable domain是和系统设计相关，我们假设一个系统的domain分配如下：

（1）所有的cpu core属于一个inner shareable domain

（2）所有的cpu core和dma controller属于一个outer shareable domain

在ARM architecture中，对一个normal memory location而言，是否是coherent是和它的页表中的shareability attribute的设定相关。

（1）**non-shareable**。根本不会再多个agent之间共享，不存在coherent的问题。不与其他CPU或DMA共享；

（2）**inner-shareable**。说明inner shareable domain中的所有的agent在对该内存进行数据访问的时候，硬件会保证coherent。

（3）**outer-shareable**。说明outer shareable domain中的所有的agent在对该内存进行数据访问的时候，硬件会保证coherent。

**Wrte-through（直写模式）**在数据更新时，同时写入缓存Cache和后端存储。

优点是操作简单；缺点是因为数据修改需要同时写入存储，数据写入速度较慢。

**Write-back（回写模式）**在数据更新时只写入缓存Cache。只在数据被替换出缓存时，被修改的缓存数据才会被写到后端存储。

优点是数据写入速度快，因为不需要写存储；缺点是一旦更新后的数据未被写入存储时出现系统掉电的情况，数据将无法找回。

**对于device type，其总是non cacheable的，而且是outer shareable**，因此它的attribute不多，主要有下面几种

附加的特性：

（1）Gathering 或者non Gathering (G or nG)。

这个特性表示对多个memory的访问是否可以合并，如果是nG，表示处理器必须严格按照代码中内存访问来进行，不能把两次访问合并成一次。例如：代码中有2次对同样的一个地址的读访问，那么处理器必须严格进行两次read transaction。

（2）Re-ordering (R or nR)。

这个特性用来表示是否允许处理器对内存访问指令进行重排。nR表示必须严格执行program order。

（3）Early Write Acknowledgement (E or nE)。

PE访问memory是有问有答的(更专业的术语叫做transaction)，对于write而言，PE需要write ack操作以便确定完成一个write transaction。为了加快写的速度，系统的中间环节可能会设定一些write buffer。nE表示写操作的ack必须来自最终的目的地而不是中间的write buffer。

2.4内核中的页表属性

**//设备内存页表属性**

#define PROT\_DEVICE\_nGnRnE (PROT\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_DIRTY | PTE\_WRITE | PTE\_ATTRINDX(MT\_DEVICE\_nGnRnE))

#define PROT\_DEVICE\_nGnRE (PROT\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_DIRTY | PTE\_WRITE | PTE\_ATTRINDX(MT\_DEVICE\_nGnRE))

#define PROT\_NORMAL\_NC (PROT\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_DIRTY | PTE\_WRITE | PTE\_ATTRINDX(MT\_NORMAL\_NC))

#define PROT\_NORMAL\_WT (PROT\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_DIRTY | PTE\_WRITE | PTE\_ATTRINDX(MT\_NORMAL\_WT))

#define PROT\_NORMAL (PROT\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_DIRTY | PTE\_WRITE | PTE\_ATTRINDX(MT\_NORMAL))

#define PROT\_SECT\_DEVICE\_nGnRE (PROT\_SECT\_DEFAULT | PMD\_SECT\_PXN | PMD\_SECT\_UXN | PMD\_ATTRINDX(MT\_DEVICE\_nGnRE))

#define PROT\_SECT\_NORMAL (PROT\_SECT\_DEFAULT | PMD\_SECT\_PXN | PMD\_SECT\_UXN | PMD\_ATTRINDX(MT\_NORMAL))

#define PROT\_SECT\_NORMAL\_EXEC (PROT\_SECT\_DEFAULT | PMD\_SECT\_UXN | PMD\_ATTRINDX(MT\_NORMAL))

#define \_PAGE\_DEFAULT (PROT\_DEFAULT | PTE\_ATTRINDX(MT\_NORMAL))

**//内核普通内存页表属性**

#define PAGE\_KERNEL \_\_pgprot(\_PAGE\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_DIRTY | PTE\_WRITE) // 可读写，不可执行

#define PAGE\_KERNEL\_RO \_\_pgprot(\_PAGE\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_DIRTY | PTE\_RDONLY) //只读

#define PAGE\_KERNEL\_ROX \_\_pgprot(\_PAGE\_DEFAULT | PTE\_UXN | PTE\_DIRTY | PTE\_RDONLY)

#define PAGE\_KERNEL\_EXEC \_\_pgprot(\_PAGE\_DEFAULT | PTE\_UXN | PTE\_DIRTY | PTE\_WRITE)

#define PAGE\_KERNEL\_EXEC\_CONT \_\_pgprot(\_PAGE\_DEFAULT | PTE\_UXN | PTE\_DIRTY | PTE\_WRITE | PTE\_CONT)

/\*

\* I/O memory mapping functions.

\*/

extern void \_\_iomem \*\_\_ioremap(phys\_addr\_t phys\_addr, size\_t size, pgprot\_t prot);

extern void \_\_iounmap(volatile void \_\_iomem \*addr);

extern void \_\_iomem \*ioremap\_cache(phys\_addr\_t phys\_addr, size\_t size);

#define ioremap(addr, size) \_\_ioremap((addr), (size), \_\_pgprot(PROT\_DEVICE\_nGnRE))

#define ioremap\_nocache(addr, size) \_\_ioremap((addr), (size), \_\_pgprot(PROT\_DEVICE\_nGnRE))

#define ioremap\_wc(addr, size) \_\_ioremap((addr), (size), \_\_pgprot(PROT\_NORMAL\_NC))

#define ioremap\_wt(addr, size) \_\_ioremap((addr), (size), \_\_pgprot(PROT\_DEVICE\_nGnRE))

#define iounmap \_\_iounmap

用户进程页表属性：

Do\_mmap->mmap\_region->vma\_set\_page\_prot->vm\_pgprot\_modify->vm\_get\_page\_prot

WRITE\_ONCE(vma->vm\_page\_prot,vm\_page\_prot)

pgprot\_t vm\_get\_page\_prot(unsigned long vm\_flags)

{

return \_\_pgprot(pgprot\_val(protection\_map[vm\_flags &

(VM\_READ|VM\_WRITE|VM\_EXEC|VM\_SHARED)]) |

pgprot\_val(arch\_vm\_get\_page\_prot(vm\_flags)));

}

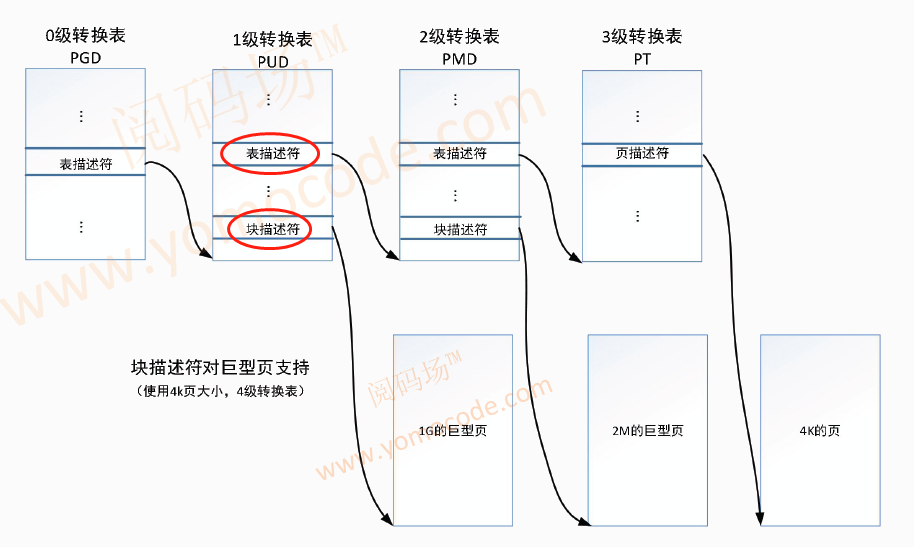
## 2.5页表对巨型页的支持

**1)块描述符支持巨型页**

**如果页的大小为4k**, 则使用4级转换表，0级转换表不能使用块描述符，1级转换表可以使用块描述符指向1G的巨型页，2级转换表可以使用块描述符指向2M的巨型页。

**如果页的大小为16k,** 则使用4级转换表，0级转换表不能使用块描述符，1级转换表不能使用块描述符，2级转换表可以使用块描述符指向32M的巨型页。

**如果页的大小为64k**, 则使用3级转换表，0级转换表不能使用块描述符，1级转换表不能使用块描述符，2级转换表可以使用块描述符指向512M的巨型页。



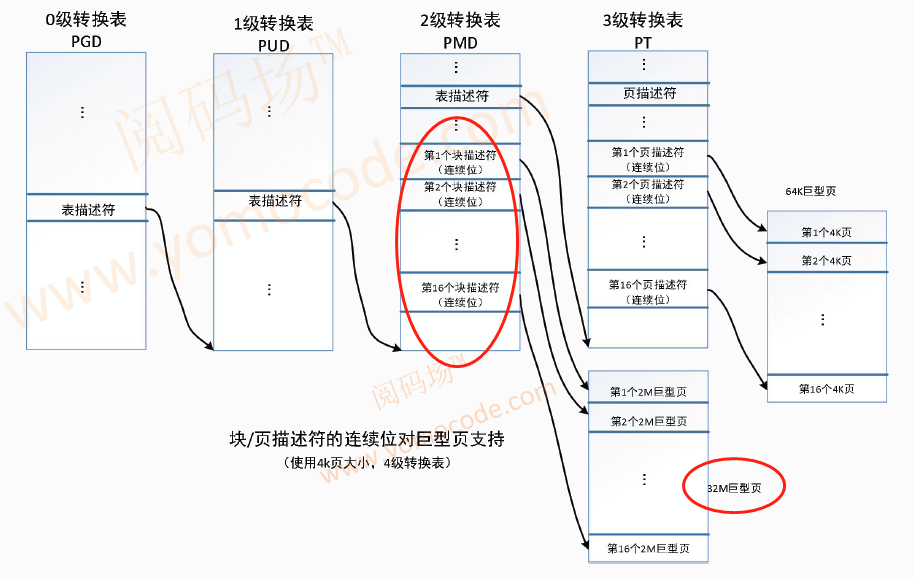
**2)块/页描述符连续位支持巨型页**

块/页描述符的连续位（Contiguous）表示的表项是一个表项集合中的一个表项，一个表项集合可以被缓存在一个tlb表项中。

**如果页的大小为4k**, 则使用4级转换表，1级转换表的块描述符不能使用连续位，2级转换表支持16个连续位，即是16\*2M=32M的巨型页，3级转换表支持16个连续位，即是16\*4K=64K的巨型页。

**如果页的大小为16k**, 则使用4级转换表，2级转换表支持32个连续位，即是32\*32M=1G的巨型页，3级转换表支持128个连续位，即是128\*16K=2M的巨型页。

**如果页的大小为64k**, 则使用3级转换表，2级转换表的块描述符不能使用连续位，3级转换表支持32个连续位，即是32\*64K=2M的巨型页。



## 3.页表遍历过程

### 3.1 页表遍历原理

Linux 4.11之前，Linux内核将页表分为4级：

页全局目录（Page Global Directory, PGD）

页上级目录（Page Upper Directory, PUD）

页中间目录（Page Middle Directory, PMD）

直接页表（Page Table, PT）

Linux 4.11之后，将页表扩展到5级，在页全局目录和页上级目录之间增加了页四级目录（Page 4th Directory, P4D）

而各个处理器架构可以选择使用

5级（pgd, p4d, pud, pmd, pt）、

4级（pgd, pud, pmd, pt）、

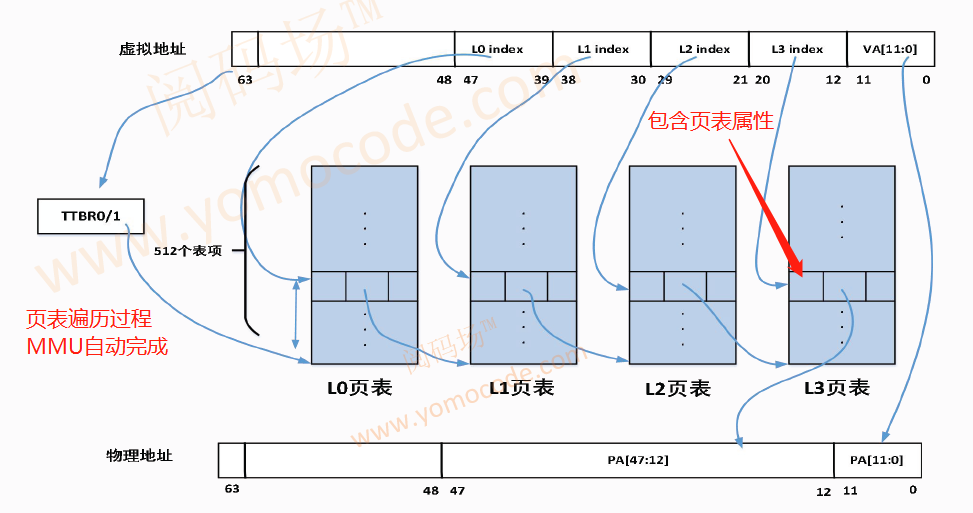
3级（pgd, pmd, pt）、

2级（pgd, pt）页表，

使用CONFIG\_PGTABLE\_LEVELS来配置页表级数，如arm64一般配置为4级。

include/asm-generic/pgtable-nopXd.h //X表示4或u或m 来模拟不使用的表项

假如配置4级页表，不使用pud, 则：



### 3.2 内核中遍历实现

软件也可以遍历页表，eg：缺页异常中遍历各级表项：

\_\_handle\_mm\_fault //mm/memory.c

->pgd = pgd\_offset(mm, address);

->p4d = p4d\_alloc(mm, pgd, address); //p4d = pgd

->vmf.pud = pud\_alloc(mm, p4d, address); //分配不存在的页表项pud

->vmf.pmd = pmd\_alloc(mm, vmf.pud, address); //分配不存在的页表项pmd

->handle\_pte\_fault

-> vmf->pte = pte\_offset\_map(vmf->pmd, vmf->address); //获取新分配的页帧号

## 4.TLB原理和操作

### 4.1 相关概念

mmu把虚拟地址转化为物理地址，为了改进转化速度，避免每次转换都从内存查询多级页表，处理器厂商就在mmu中增加了一个称为TLB的高速缓存。

**TLB（Translation Lookaside Buffer）转换旁路缓冲区**，又称为**块表**，可以理解为**页表缓存**，用来缓存最近使用的页表项。

有些处理器有两级TLB：

第一级TLB分为指令TLB和数据TLB, 访问指令和数据可以并行执行；

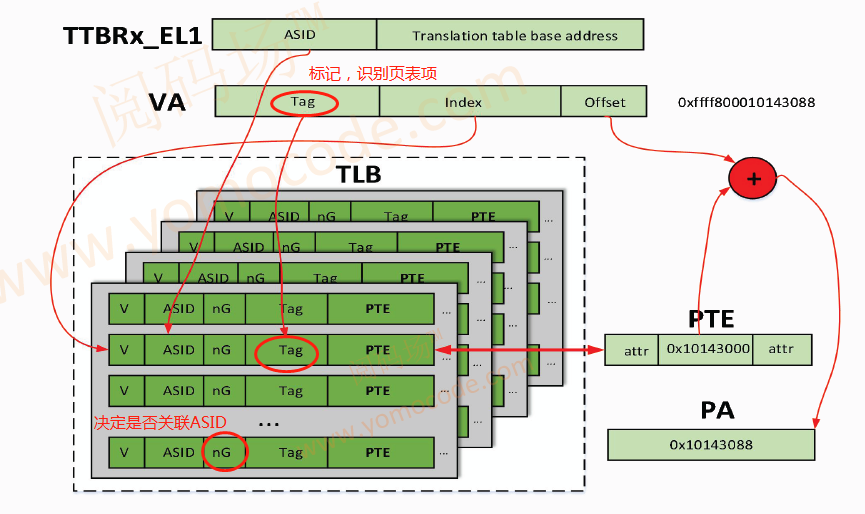
第二级的TLB为统一的tlb(Unfied TLB), 指令和数据共用TLB。

### 4.2 tlb原理

原理概述：

当cpu发出虚拟地址访问时，mmu首先查询TLB ，看TLB 中是否缓存了虚拟地址对应的页表

项，如果缓存了直接获得页表项，称为**TLB hit**，如果没有缓存则成为**TLB miss,** 需要在内存中遍历各级页表获得页表项，然后页表项填充到TLB中，如果TLB已经满了，那么还要设计替换算法来决定让哪一个TLB entry失效，从而加载新的页表项。

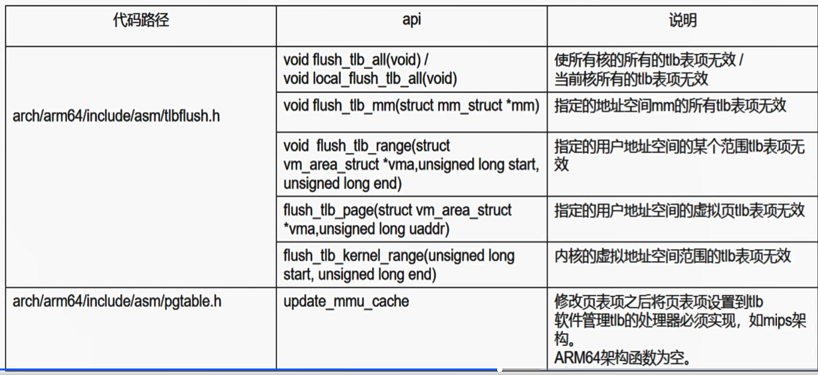


### 4.3 tlb操作

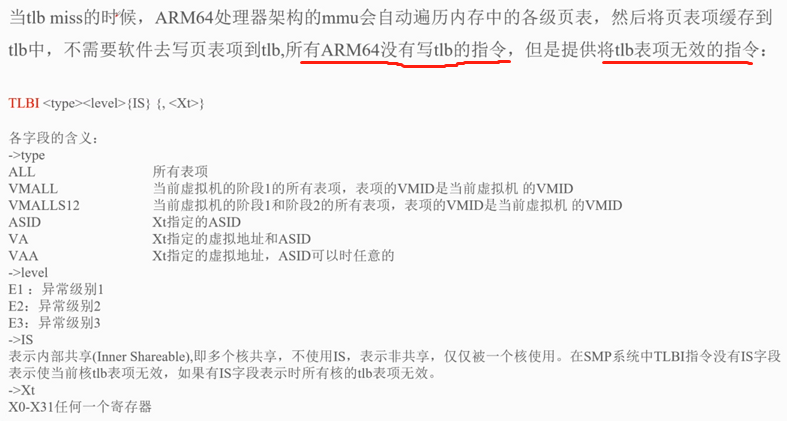
内核中如果修改了缓存在tlb中的页表项，那么内核就需要使旧的tlb表项无效（invalidate tlb,也叫做flush tlb）。

下面为内核定义的一些tlb无效的api, 由各个处理器架构来实现：

Arch/arm64/include/asm/tlbflush.h



### 4.4 flush tlb的arm64实现



## 5 ASID机制

### 5.1概念

**地址空间标识符**（Address Space Identifier, ASID）

是为了减少在进程切换时清空tlb的操作，arm64处理器在tlb中增加了nG位区别内核和用户进程的页表项，**使用ASID区别不同的用户进程的页表项**。

注：ASID机制存在之前，由于进程切换时可能存在的是上一个进程的tlb表项，需要清空tlb，但是被切换的进程获得的是全空的tlb,这样进程地址转换需要遍历多级页表，极大的影响系统性能。

**ASID的长度**：arm64处理器可以选择8bit或16bit，

**ID\_AA64MMFR0\_EL1** 的ASIDBits, bits [7:4] 存放处理器支持的ASID长度。

通过TCR\_EL1寄存器的AS位来选择实际使用的长度，0表示使用8bit的ASID, 1表示使用16bit的ASID。

ASID的存放位置：保存到进程的task\_struct的mm->contex->id。

### 5.2 ASID机制实现原理

管理变量：

asid\_generation ：表示ASID版本号，全局变量，区别当前分配是否同一批次；

每处理器变量active\_asid：处理器正在使用的ASID；

asid\_bits**:**ASID长度

每处理器变量reserved\_asids :存放保留的ASID（ASID版本号加1时保存 处理

器正在执行进程的ASID）

tlb\_flush\_pending需要刷tlb的cpu位掩码

asid\_map ASID分配位图，记录哪些ASID被分配

static int asids\_init(void)

{

asid\_bits = get\_cpu\_asid\_bits();//获取大小

atomic64\_set(&asid\_generation, ASID\_FIRST\_VERSION);//获取版本号

asid\_map = kcalloc(BITS\_TO\_LONGS(NUM\_USER\_ASIDS), sizeof(\*asid\_map),

GFP\_KERNEL);

if (!asid\_map)

panic("Failed to allocate bitmap for %lu ASIDs\n",

NUM\_USER\_ASIDS);

pinned\_asid\_map = kcalloc(BITS\_TO\_LONGS(NUM\_USER\_ASIDS),

sizeof(\*pinned\_asid\_map), GFP\_KERNEL);

nr\_pinned\_asids = 0;

/\*

\* We cannot call set\_reserved\_asid\_bits() here because CPU

\* caps are not finalized yet, so it is safer to assume KPTI

\* and reserve kernel ASID's from beginning.

\*/

if (IS\_ENABLED(CONFIG\_UNMAP\_KERNEL\_AT\_EL0))

set\_kpti\_asid\_bits(asid\_map);

return 0;

}

early\_initcall(asids\_init);

**1)内核初始化时：**

arch/arm64/mm/context.c

start\_kernel

->early\_initcall(asids\_init)

**2)fork时初始化进程ASID**

\_do\_fork //kernel/fork.c

-> copy\_process

->dup\_mm

->mm\_init

->init\_new\_context

#define init\_new\_context(tsk,mm) ({ atomic64\_set(&(mm)->context.id, 0); 0; }) //初始为0

**3)进程切换时分配ASID**

\_\_schedule //kernel/sched/core.c

->context\_switch

->switch\_mm\_irqs\_off

->switch\_mm

->\_\_switch\_mm //arch/arm64/include/asm/mmu\_context.h

->check\_and\_switch\_context //arch/arm64/mm/context.c

->new\_context //为进程分配新的ASID

**4）ASID分配满之后刷tlb:**

check\_and\_switch\_context

->if (cpumask\_test\_and\_clear\_cpu(cpu, &tlb\_flush\_pending))

local\_flush\_tlb\_all();

**5) 切换地址空间**

check\_and\_switch\_context

->cpu\_switch\_mm(mm->pgd, mm) //切换pgd和ASID

void check\_and\_switch\_context(struct mm\_struct \*mm)

{

unsigned long flags;

unsigned int cpu;

u64 asid, old\_active\_asid;

if (system\_supports\_cnp())

cpu\_set\_reserved\_ttbr0();

asid = atomic64\_read(&mm->context.id);//获得ASID

/\*

\* The memory ordering here is subtle.

\* If our active\_asids is non-zero and the ASID matches the current

\* generation, then we update the active\_asids entry with a relaxed

\* cmpxchg. Racing with a concurrent rollover means that either:

\*

\* - We get a zero back from the cmpxchg and end up waiting on the

\* lock. Taking the lock synchronises with the rollover and so

\* we are forced to see the updated generation.

\*

\* - We get a valid ASID back from the cmpxchg, which means the

\* relaxed xchg in flush\_context will treat us as reserved

\* because atomic RmWs are totally ordered for a given location.

\*/

old\_active\_asid = atomic64\_read(this\_cpu\_ptr(&active\_asids));//当前CPU活跃的ASID

/\*

\* asid\_gen\_match:版本号

\*/

if (old\_active\_asid && asid\_gen\_match(asid) &&

atomic64\_cmpxchg\_relaxed(this\_cpu\_ptr(&active\_asids),

old\_active\_asid, asid))

goto switch\_mm\_fastpath;//版本号是是同批次，不需要分配ASID

raw\_spin\_lock\_irqsave(&cpu\_asid\_lock, flags);

/\* Check that our ASID belongs to the current generation. \*/

asid = atomic64\_read(&mm->context.id);

if (!asid\_gen\_match(asid)) { //批次不同，分配新ASID

asid = new\_context(mm);

atomic64\_set(&mm->context.id, asid);

}

cpu = smp\_processor\_id();

if (cpumask\_test\_and\_clear\_cpu(cpu, &tlb\_flush\_pending))

local\_flush\_tlb\_all();

atomic64\_set(this\_cpu\_ptr(&active\_asids), asid);

raw\_spin\_unlock\_irqrestore(&cpu\_asid\_lock, flags);

switch\_mm\_fastpath:

arm64\_apply\_bp\_hardening();

/\*

\* Defer TTBR0\_EL1 setting for user threads to uaccess\_enable() when

\* emulating PAN.

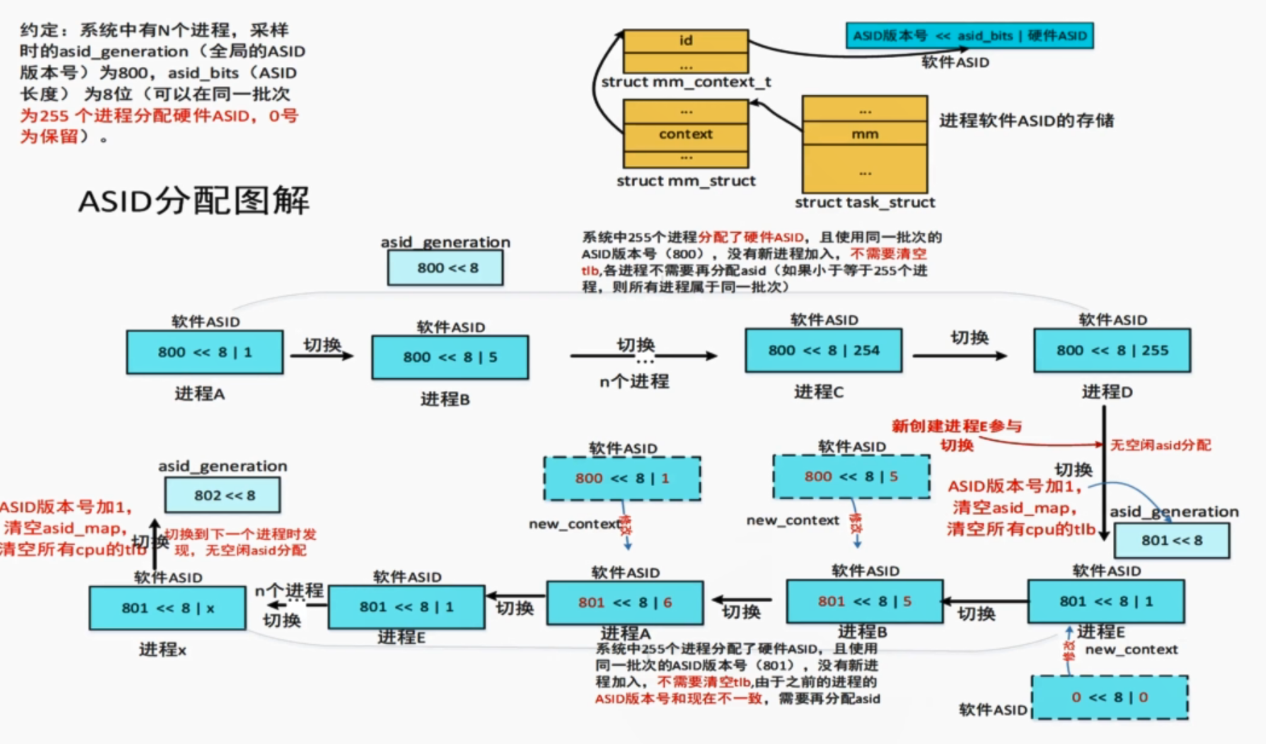
\*/

if (!system\_uses\_ttbr0\_pan())

cpu\_switch\_mm(mm->pgd, mm);

}

### 5.3分配ASID实例



补充：

(1)线程之间不存在内存切换；

(2)内核线程之间，不存在地址切换；

(3)进程切换时，mm传入pgd和asid;

会根据pgd计算实际物理地址；

根据VA和asid查找TLB表项是否命中；

## 6.Linux内核页表操作相关定义

1）表项数据类型定义：//arch/arm64/include/asm/pgtable-types.h

pgd\_t p4d\_t pud\_t pmd\_t pte\_t

eg：

typedef u64 pgdval\_t;

typedef struct { pgdval\_t pgd; } pgd\_t;

2）类型转换 //arch/arm64/include/asm/pgtable-types.h

pgd\_val p4d\_val pud\_val pmd\_val pte\_val //pXd\_t转化为u64

\_\_pgd \_\_p4d \_\_pud \_\_pmd \_\_pte //u64转化为pXd\_t

eg:

#define pte\_val(x) ((x).pte)

#define \_\_pte(x) ((pte\_t) { (x) } )

3) 表项大小定义

arch/arm64/include/asm/pgtable-hwdef.h

XXX\_SHIFT 表示各级页表索引在虚拟地址中的偏移 eg:12

XXX\_SIZE 表示各级页表表项描述的地址空间大小 eg:4096

XXX\_MASK 表示各级页表屏蔽位掩码

PTRS\_PER\_XXX 表示各级页表存放的表项个数 eg:512

pgd pud pmd pte 表项分别映射大小为512G 1G 2M 4k内存.

pgd页目录映射512表项\* 512G = 256T

eg:#if CONFIG\_PGTABLE\_LEVELS > 3

#define PUD\_SHIFT ARM64\_HW\_PGTABLE\_LEVEL\_SHIFT(1)

#define PUD\_SIZE (\_AC(1, UL) << PUD\_SHIFT)

#define PUD\_MASK (~(PUD\_SIZE-1))

#define PTRS\_PER\_PUD PTRS\_PER\_PTE

#endif

4）获得表项索引：arch/arm64/include/asm/pgtable.h

XXX\_index //从虚拟地址中分解出XXX索引

eg:

#define pmd\_index(addr) (((addr) >> PMD\_SHIFT) & (PTRS\_PER\_PMD - 1))

5) 获得表项地址：arch/arm64/include/asm/pgtable.h

XXX\_offset //从指定的地址空间/目录表中获得XXX表项地址

eg:

#define pud\_offset(dir, addr) ((pud\_t \*)\_\_va(pud\_offset\_phys((dir), (addr))))

#define pgd\_offset\_k(addr) pgd\_offset(&init\_mm, addr)

6)表项状态判断：arch/arm64/include/asm/pgtable.h

XXX\_none //判断是否为空表项

XXX\_bad //判断是否为坏的表项

XXX\_present //判断表项是否存在

eg:

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

#define pgd\_bad(pgd) (!(pgd\_val(pgd) & 2))

#define pgd\_present(pgd) (pgd\_val(pgd))

7）表项设置：arch/arm64/include/asm/pgtable.h

pte\_wrprotect //设置为写保护

pte\_mkwrite //设置为可写

pte\_mkclean //清除脏标志

pte\_mkdirty //设置脏标志

pte\_mkyoung //设置为访问标志

pte\_mkold //清除访问标志

set\_pte(pte\_t \*ptep, pte\_t pte) //设置pte到ptep

#define pte\_pfn(pte) (\_\_pte\_to\_phys(pte) >> PAGE\_SHIFT) //页表项中取出页帧号

#define pfn\_pte(pfn,prot) \ //页帧号和标志组合成页表项

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

8）页目录/页表分配和释放：include/linux/mm.h

XXX\_alloc //XXX页表分配 如分配页全局目录，分配页表等

eg:

#define pte\_alloc(mm, pmd) (unlikely(pmd\_none(\*(pmd))) && \_\_pte\_alloc(mm, pmd))

XXX\_free //XXX页表释放 如释放页表

eg:

static inline void pte\_free(struct mm\_struct \*mm, struct page \*pte\_page)

{

pgtable\_pte\_page\_dtor(pte\_page);

\_\_free\_page(pte\_page);

}

7.启动阶段段早期的页表创建

# VIM用法:

X删除当前字符；

“；”，进入低行模式；

把光标建映射成Nop

noremap <Up> <Nop>

noremap <Down> <Nop>

noremap <Left> <Nop>

noremap <Right> <Nop>

cw：替换光标所在单词；

移动光标：

0：到行头；

^:到本行第一个不是blank(空格，tab，换行，回车等)字符的位置；

$:到本行行尾；

g\_:到本行最后一个不是blank字符的位置；

/xxx:搜索xxx字符串，若搜出多个，可以按n键到下一个；

拷贝粘贴撤销：

P:p/P

Yy:拷贝

U:撤销

Ctrl+r： redo

文件操作：

:e: xx.c

:w, 存文件

：save as path\_xxx

:wq,保存并退出

:q!, 退出不保存，:qa!强行退出所有的正在编辑文件，就算别的文件有更改；

:bn或:bp, 同事打开很多个文件，切换下一个文件；

“.”，重复上个命令；

Num cmd:重复某个命令N此，比如7 yy

取消高亮

:noh

更快移动光标：

nG,: n移到第N行；

gg, 第一行；

G，最后一行；

w, 下一个单词的开头；

e，到下一个单词的结尾；

%, 匹配括号的移动，(,{,[等；

#/\*，匹配光标当前所在的单词，移动到下一个(\*)匹配单词,(#)上一个

F x，到下一个为x的字符处；

T x, 到x前的第一个字符

3fx,在当前行查找第三个出现的x

F/T, f/t相同；

可视化

V,按行可视化

Ctrl + v,按列可视化

I,按列插入字符//，ESC退出；每行都添加//

d,按列删除指定块

ctrl+v, $,选中可视化块，A,在末尾添加字符hello，ESE，每行莫为添加hello

分屏：

Split

Vsplit

Ctrl+ h/j/k/l

Ctrl+w，+/- 调整屏幕尺寸；

Sudo apt

Vim

:PluginInstall 自动下载安装插件

YouCompleteMe 代码补全，vim --version >7.4

~/.vim/bundle/YouCompleteMe/

Sudo apt install build-essential cmake python-dev python3-dev

./install.py –clang-completer

Cp third\_party/ycmd/examples/.ycm\_extra\_conf.py ~/.vim

.vimrc 添加

Ycm

Sudo apt install ctags //记录索引，查找，跳转

Sudo apt install cscope //比ctags强大，贝尔实验室开发，开源

Tagbar:动态显示函数名等

Nerdtree

Vim-airline

Export ARCH=arm

Export CROSS\_COMPLIE=arm-linux-guneabi-

Make express\_defconfig

Make tags cscope TAGS //生成索引文件

VIM8 极限IDE

1.ctags不能实时更新；

2.YCM如何实现对本地工程代码的索引；

3.代码动态语法检测；

4.函数列表查找；

5.函数参数提示；

VIM8

1.异步IO，可以在后台启动任务，这样可以便浏览文件，便运行长时间编译任务；

2.时钟机制：可以创建时钟，比如每隔100ms调用某个插件函数；

Ubuntu6.04：

Sudo add-apt-repository ppa:jonathonf/vim

Sudo apt update

Sudo apt install vim

自动索引 vim-gutentags

Ycm添加对本地工程代码的支持；

修改~/.vim/.ycm\_extra\_conf.py

动态检测 ALE

Plugin ‘w0rp/ale’

关闭YCM的语法检测

Let g:ycm\_show\_diagnostics\_ui=0

可以动态检测简单语法错误；

函数列表查找：LeaderF

模糊查找文件

:Leaderffile

Xxx 模糊查找

查找函数名

:LeaderfFunction //列出所有函数名

查找变量：

:LeaderfBufTag

函数参数提示插件echofunc

关闭YCM

在下方显示

Func\_xxx()自动显示参数