www.yomocode.com

深入剖析Linux内核页表技术

(基于arm64处理器架构)

学习本课程将get什么？

1. 了解mmu的一些概念
2. 掌握arm64页表结构
3. 掌握页表遍历过程
4. 掌握ASID机制原理
5. 掌握TLB原理和操作
6. 理解内核页表和用户进程页表建立过程
7. 会分析页表遍历相关内核源代码

本课程有哪些特色？

深入理解页表技术

各阶段页表建立源代码分析

全面、深入解读运作机理

课程大纲

1. 概述
2. arm64页表结构
3. 页表遍历过程
4. TLB原理和操作
5. ASID机制
6. Linux内核页表操作相关定义
7. 启动阶段早期的页表创建
8. fixmap映射
9. 主内核页表创建
10. 用户进程页表创建
11. ioremap原理
12. 实践

1. 概述

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，掌握了页表技术原理，是理解内存管理的各个组件的基础。

学习理解Linux内核虚拟内存管理，必须理解页表机制！

注：课程中使用linux-5.4.78内核源代码！

1.1 概念

回答如下几个问题：

什么是mmu?使用mmu的好处是什么?

什么是页表?

什么是tlb?

使用多级页表的好处?

硬件做的事情?

软件做的事情?

什么是mmu？ 使用mmu的好处是什么？

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，隔离各个进程的地址空间。恶意进程不会访问到其他进程及内核的地址空间。

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，表项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。

什么是tlb?

页表缓存，缓存最近使用的页表项

使用多级页表的好处？

按需分配，节省内存

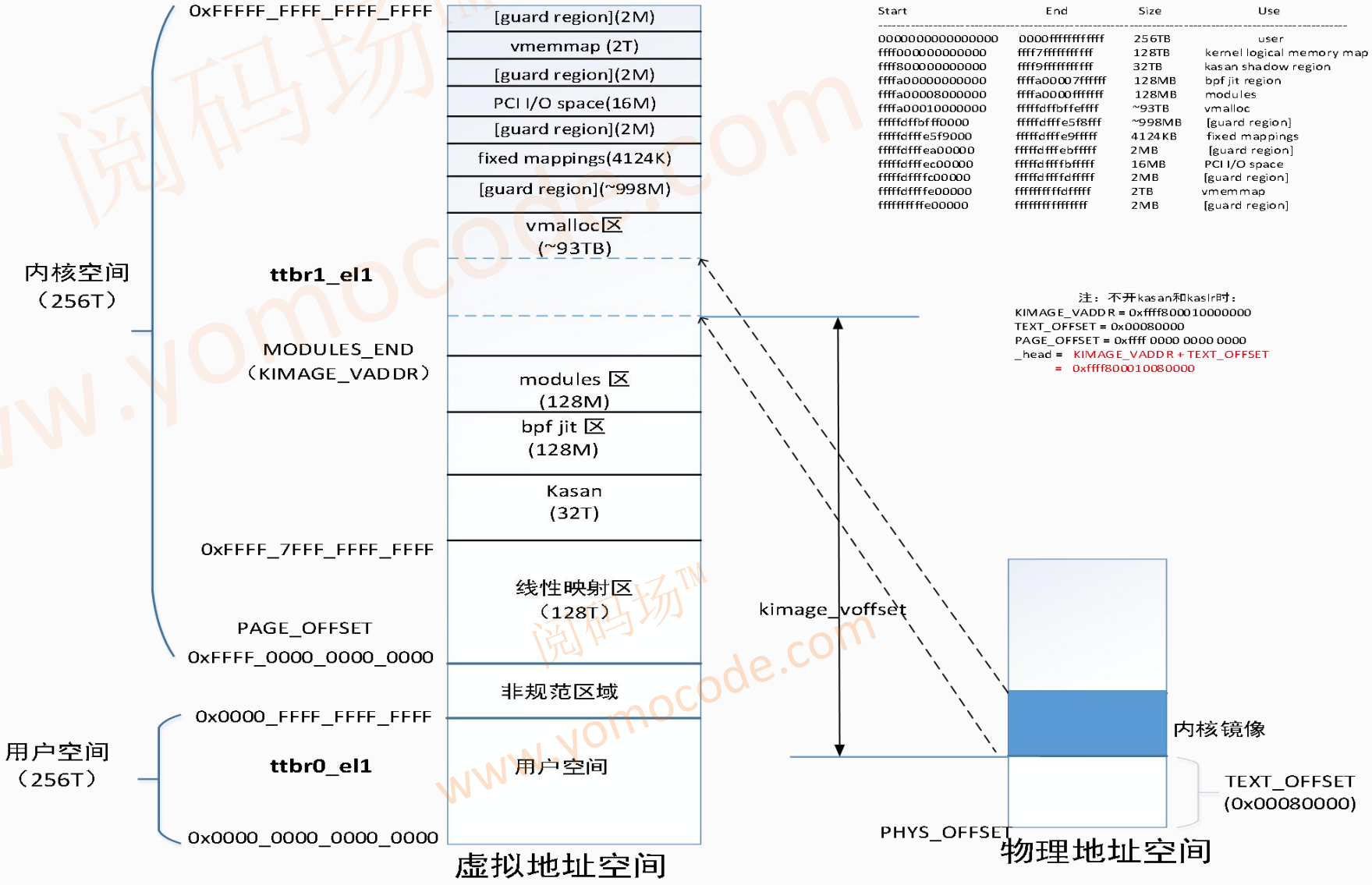
硬件做的事情？

遍历页表

软件做的事情？

填写页表

1.2 arm64虚拟地址空间

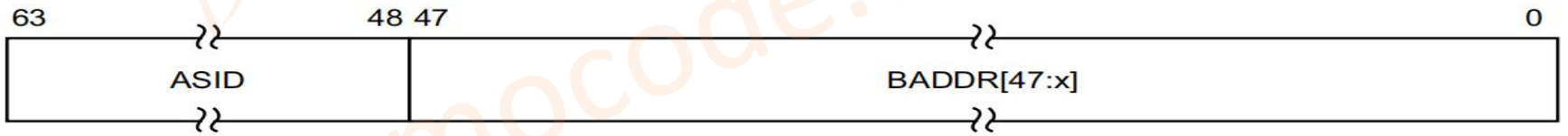


Start	End	Size	Use
0000000000000000	0000ffffffffffff	256TB	user
ffff000000000000	ffff7fffffffffff	128TB	kernel logical memory map
ffff800000000000	ffff9fffffffffff	32TB	kasan shadow region
ffffa00000000000	ffffa00007ffffff	128MB	bpf jit region
ffffa00008000000	ffffa0000fffffff	128MB	modules
ffffa00010000000	ffffdfffbffeffff	~93TB	vmalloc
ffffdfffbfff0000	ffffdfffe5f8ffff	~998MB	[guard region]
ffffdfffe5f90000	ffffdfffe9ffffff	4124KB	fixed mappings
ffffdfffea00000	ffffdfffebffffff	2MB	[guard region]
ffffdfffec00000	ffffdfffefffffff	16MB	PCI I/O space
ffffdfffefc00000	ffffdffffdffffff	2MB	[guard region]
ffffdffffe00000	fffffffffdffffff	2TB	vmemmap
fffffffffe00000	ffffffffffffffff	2MB	[guard region]

注：不开kasan和kaslr时：
KIMAGE_VADDR = 0xffff800010000000
TEXT_OFFSET = 0x00080000
PAGE_OFFSET = 0xffff 0000 0000 0000
_head = KIMAGE_VADDR + TEXT_OFFSET
= 0xffff800010080000

1.3 页表管理相关重要寄存器

ttbr0_el1 转换表基址寄存器0(el1)，存放用户地址空间页表基地址



ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

BADDR[47:x], bits [47:0]

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0.

x is based on the value of TCR_EL1.T0SZ, the stage of translation, and the memory translation granule size.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated.

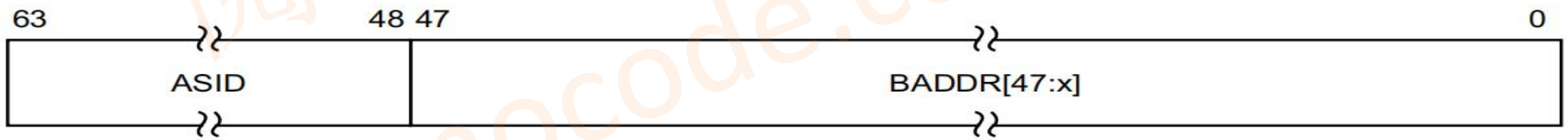
The value of x determines the required alignment of the translation table, which must be aligned to 2^x bytes.

If bits [x-1:0] are not all zero, this is a misaligned Translation Table Base Address. Its effects are CONSTRAINED UNPREDICTABLE, and can be one of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits might be the value written or might be zero.
- The calculation of an address for a translation table walk using this register can be corrupted in those bits that are non-zero.

1.3 页表管理相关重要寄存器

ttbr1_el1 转换表基址寄存器1(el1)，存放内核地址空间页表基地址



ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

BADDR[47:x], bits [47:0]

Translation table base address, bits[47:x]. Bits [x-1:0] are RES0.

x is based on the value of TCR_EL1.T0SZ, the stage of translation, and the memory translation granule size.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated.

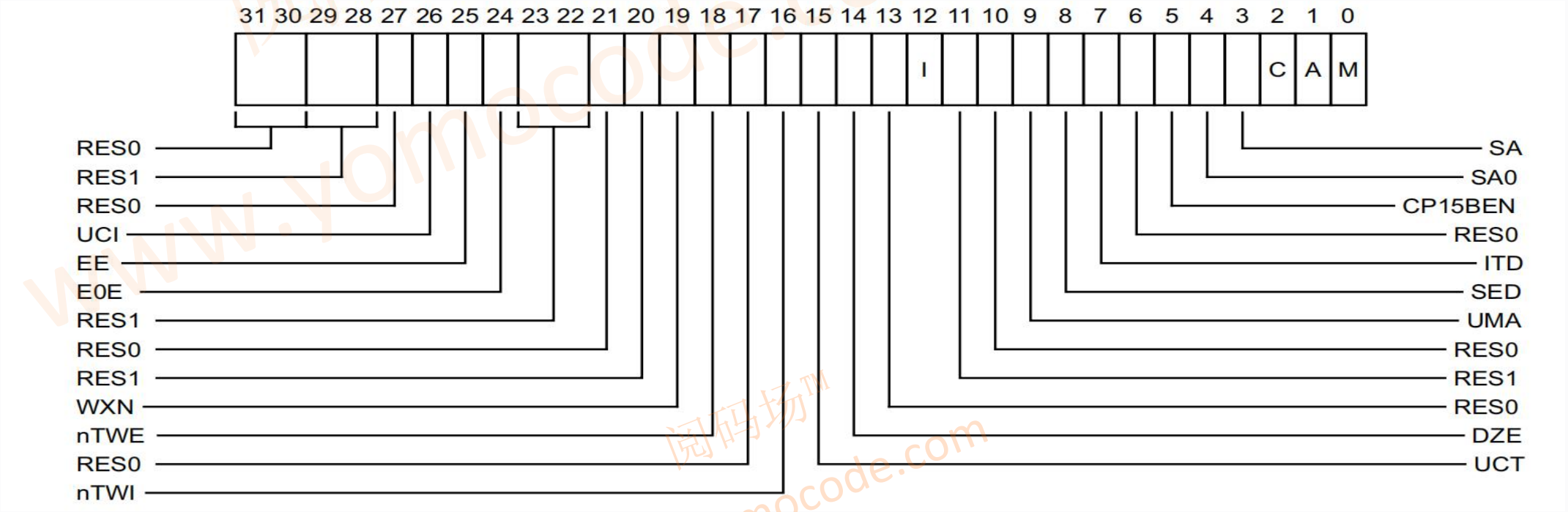
The value of x determines the required alignment of the translation table, which must be aligned to 2^x bytes.

If bits [x-1:0] are not all zero, this is a misaligned Translation Table Base Address. Its effects are CONSTRAINED UNPREDICTABLE, and can be one of the following:

- Bits [x-1:0] are treated as if all the bits are zero. The value read back from those bits might be the value written or might be zero.
- The calculation of an address for a translation table walk using this register can be corrupted in those bits that are non-zero.

1.3 页表管理相关重要寄存器

sctlr_el1 系统控制寄存器（el1），控制mmu使能



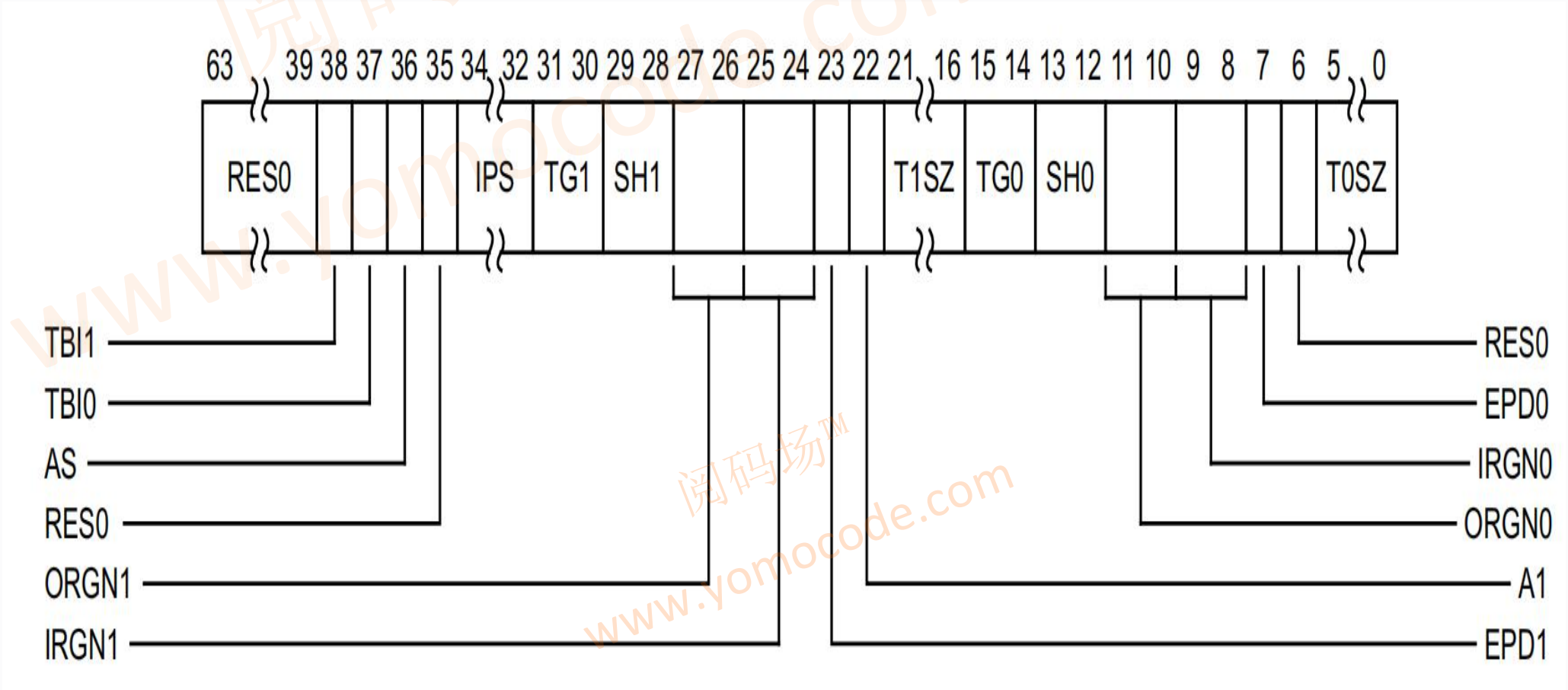
I, bit [12]：指令cache使能位

C, bit [2]：cache使能位，使能数据和统一cache

M, bit [0]: MMU使EL1和EL0阶段1地址转换使能位。

1.3 页表管理相关重要寄存器

TCR_EL1 转换控制寄存器（el1），控制转换格式等



AS, bit [36] ： ASID大小,eg:16bit

IPS, bits [34:32]： 中间物理地址大小， eg:48bit

TG1, bits [31:30]： TTBR1_EL1粒度大小,eg： 4k

SH1, bits [29:28]: 通过TTBR1_EL1访问内存的可共享属性

ORGN1, bits [27:26]： 通过TTBR1_EL1访问内存的外部可缓存属性

IRGN1, bits [25:24]： 通过TTBR1_EL1访问内存的内部可缓存属性

EPD1, bit [23]： 使用TTBR1时， tlb miss时转换表遍历失能

A1, bit [22]： 选择TTBR0_EL1或TTBR1_EL1来定义ASID

T1SZ, bits [21:16]: 通过TTBR1_EL1所管辖的虚拟地址大小

TG0, bits [15:14]： TTBR0_EL1粒度大小,eg： 4k

SH0, bits [13:12]： 通过TTBR0_EL1访问内存的可共享属性

ORGN0, bits [11:10]： 通过TTBR0_EL1访问内存的外部可缓存属性

IRGN0, bits [9:8]： 通过TTBR0_EL1访问内存的内部可缓存属性

EPD0, bit [7]： 使用TTBR0时， tlb miss时转换表遍历失能

T0SZ, bits [5:0]： 通过TTBR0_EL1所管辖的虚拟地址大小

2. arm64页表结构

2.1 arm64页表格式

1）arm64处理器将页表成为转化表（translation table），
最多支持4级转换表(L0-L3)，将表项称为描述符。

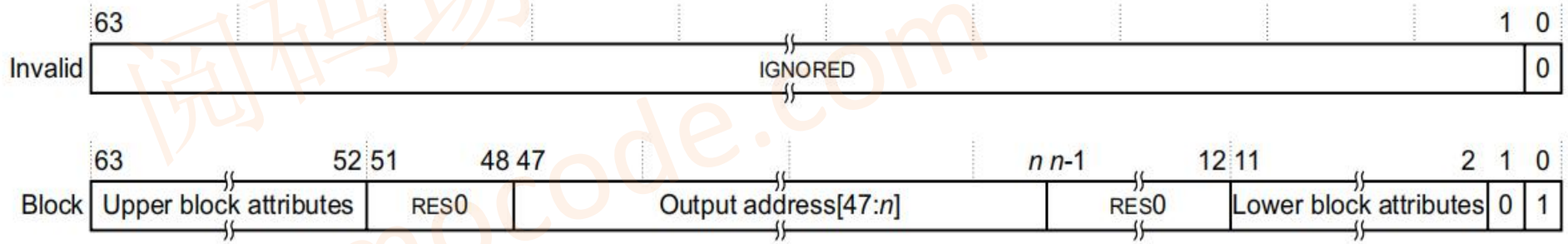
2）支持3种页长度：4k，16k, 46k。

3）描述符的第0bit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**，为1表示**表描述符**。
3级转换表，第1bit为0表示**保留描述符**，为1表示**页描述符**。

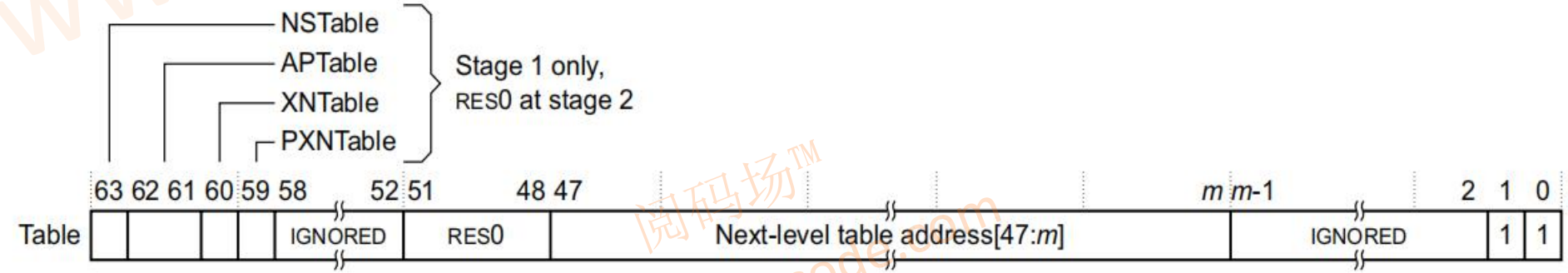
CONFIG_PGTABLE_LEVELS=4
CONFIG_ARM64_4K_PAGES=y
CONFIG_ARM64_VA_BITS_48=y
CONFIG_ARM64_VA_BITS=48
CONFIG_ARM64_PA_BITS_48=y
CONFIG_ARM64_PA_BITS=48

注：块描述符通常用于实现巨型页或段映射，表描述符存放有下一级转换表地址，页描述符存放有物理页帧号。
且讲解主要集中在Non-secure EL1和EL0、stage 1 translation、VA和PA的地址宽度都是48个bit。

48位虚拟地址宽度为例：



With the 4KB granule size, for the level 1 descriptor n is 30, and for the level 2 descriptor, n is 21.
With the 16KB granule size, for the level 2 descriptor, n is 25.
With the 64KB granule size, for the level 2 descriptor, n is 29.

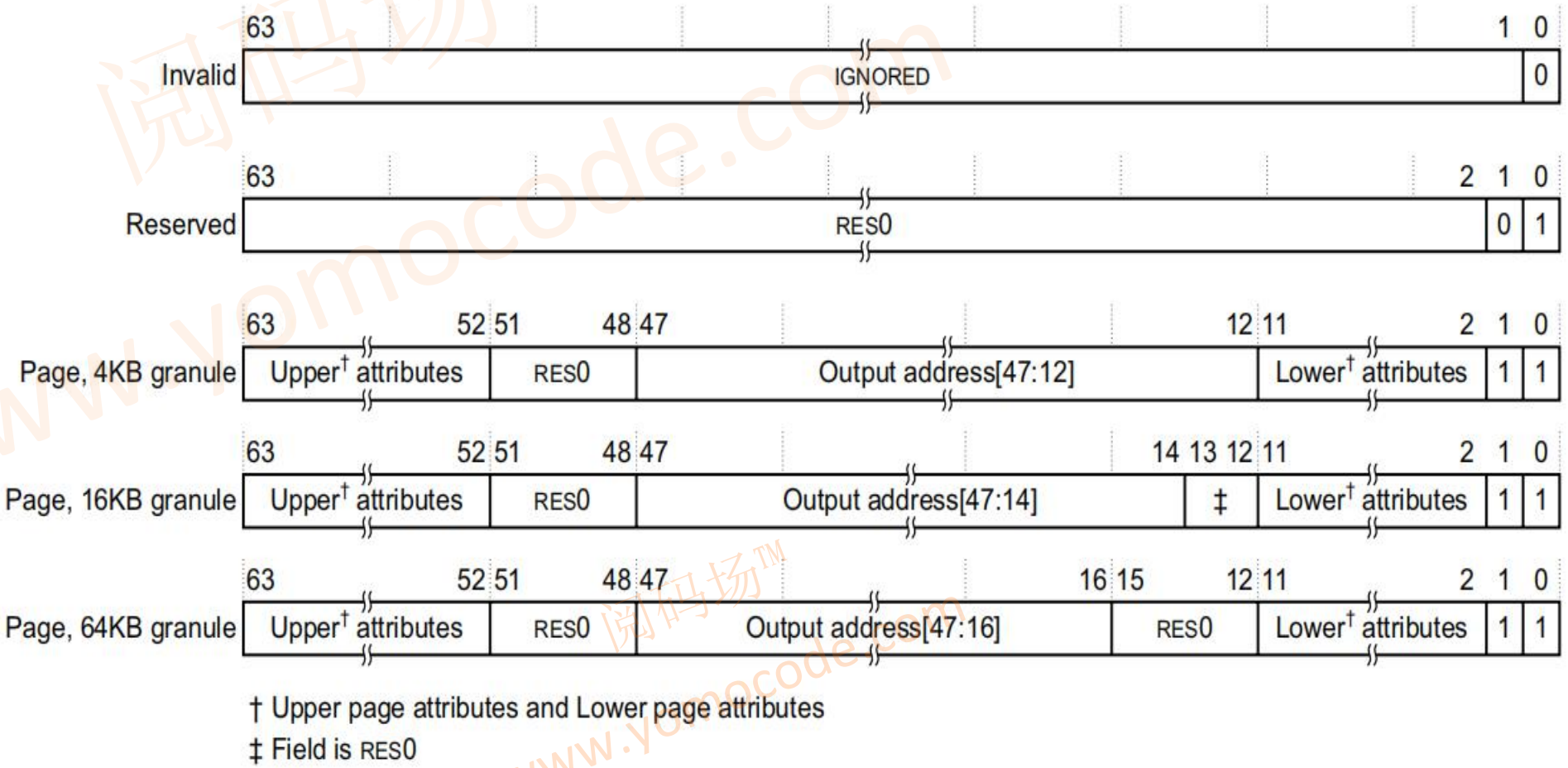


With the 4KB granule size m is 12, with the 16KB granule size m is 14, and with the 64KB granule size, m is 16.

A level 0 Table descriptor returns the address of the level 1 table.
A level 1 Table descriptor returns the address of the level 2 table.
A level 2 Table descriptor returns the address of the level 3 table.

Figure D4-28 VMSAv8-64 level 0, level 1, and level 2 descriptor formats

Figure D4-29 shows the ARMv8 level 3 descriptor formats.

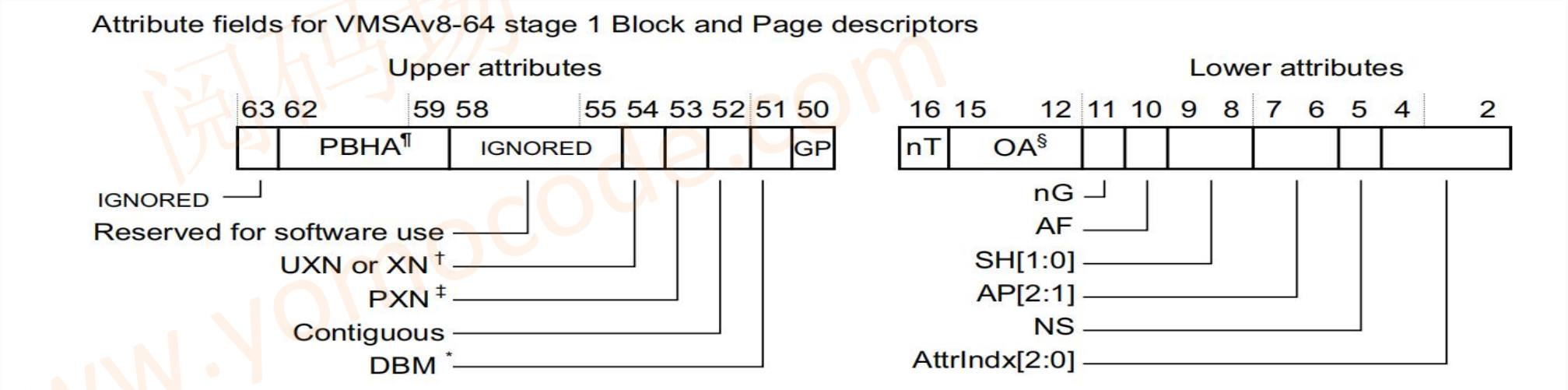


† Upper page attributes and Lower page attributes

‡ Field is RES0

Figure D4-29 VMSAv8-64 level 3 descriptor format

2.2 页表属性



55-58bit：保留软件使用

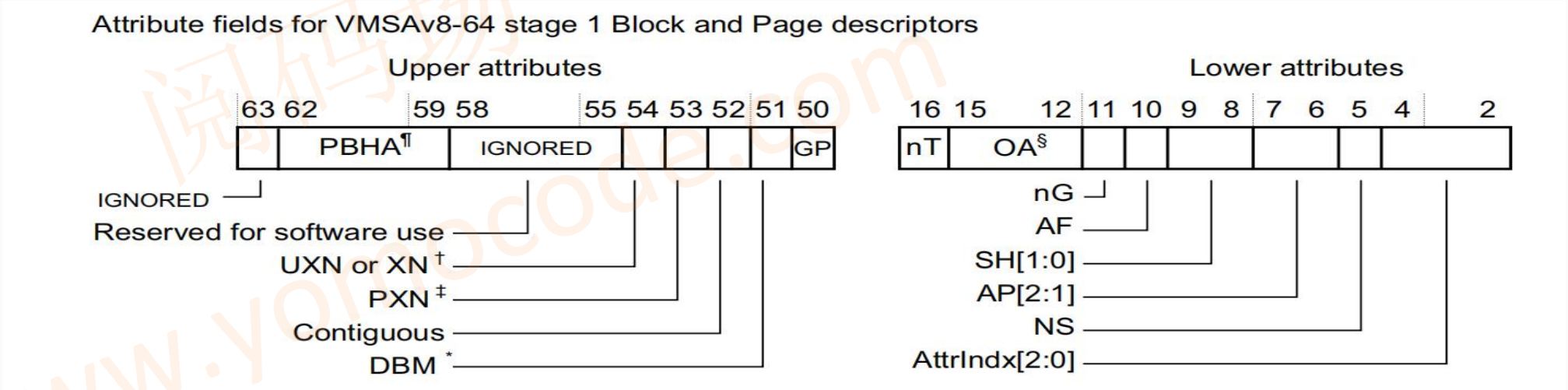
UXN or XN, bit[54]：在EL0表示 UXN，不允许EL0执行；

PXN, bit[53]：PXN 特权级不可执行（EL1）

Contiguous, bit[52]: 表示这条表项属于一个连续表项集合中的一个表项，连续表项集合可以被缓存在一条tlb表项中。

DBM, bit[51]: 脏位 表示页面是否被修改

nG, bit[11]：非全局，为1表示转换不是全局的，是进程私有的，会关联一个ASID；为0表示转换是全局的是所有进程共享，内核地址空间是所有进程共享。



AF, bit[10]：访问标志，表示页或者内存块从对应的表项被清0之后是否被访问过。

SH, bits[9:8]：可共享域

AP[2:1]：数据访问权限位，AP[2]选择只读/可读可写 -> 0可读可写,1只读

AP[1]选择el0/el1访问 -> 0不可被el0访问，1可被el0访问

NS, bit[5]：非安全位， 处于安全模式时用于指定访问的内存地址是安全映射还非安全

AttrIndx[2:0], bits[4:2]：索引不同的内存属性，MAIR_ELx表示内存属性

SH[1:0]	Normal memory
00	Non-shareable
01	UNPREDICTABLE
10	Outer Shareable
11	Inner Shareable

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

```
/*
 * 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
```

- The memory type, Device or Normal.
- For Device memory, the Device memory type, one of:
  - Device-nGnRnE.
  - Device-nGnRE.
  - Device-nGRE.
  - Device-GRE.
- For Normal memory:
  - The inner and outer cacheability, Non-cacheable, Write-Through, or Write-Back
  - For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate policy hints, each of which is *Allocate* or *Do not allocate*, and the Transient allocation hints.

arch/arm64/mm/proc.S： __cpu_setup

```
/*
 * 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。

（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

内核页表属性：

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

```
#define PROT_NORMAL             (PROT_DEFAULT | PTE_PXN | PTE_UXN | PTE_WRITE |
PTE_ATTRINDX(MT_NORMAL))

#define PAGE_KERNEL             __pgprot(PROT_NORMAL)#define PAGE_KERNEL_ROX
__pgprot((PROT_NORMAL & ~(PTE_WRITE | PTE_PXN)) | PTE_RDONLY)
#define PAGE_KERNEL_EXEC        __pgprot(PROT_NORMAL & ~PTE_PXN)
#define PAGE_KERNEL_EXEC_CONT   __pgprot((PROT_NORMAL & ~PTE_PXN) | PTE_CONT)
```


设备页表属性：

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

```
#define _PROT_DEFAULT          (PTE_TYPE_PAGE | PTE_AF | PTE_SHARED)
#define PROT_DEFAULT           (_PROT_DEFAULT | PTE_MAYBE_NG)

#define PROT_DEVICE_nGnRE      (PROT_DEFAULT | PTE_PXN | PTE_UXN | PTE_WRITE |
PTE_ATTRINDX(MT_DEVICE_nGnRE))
```

arch/arm64/include/asm/io.h

```
#define ioremap(addr, size)          __ioremap((addr), (size), __pgprot(PROT_DEVICE_nGnRE))
```


用户进程页表属性：

do_mmap_pgoff //mm/mmap.c

->do_mmap

->mmap_region

->vma_set_page_prot

->vm_pgprot_modify

->vm_get_page_prot

WRITE_ONCE(vma->vm_page_prot, vm_page_prot)

```
/* description of effects of mapping type and prot in current implementation.
 * this is due to the limited x86 page protection hardware.  The expected
 * behavior is in parens:
 *
 * map_type     prot
 *              PROT_NONE       PROT_READ       PROT_WRITE      PROT_EXEC
 * MAP_SHARED   r: (no) no      r: (yes) yes    r: (no) yes     r: (no) yes
 *              w: (no) no      w: (no) no      w: (yes) yes    w: (no) no
 *              x: (no) no      x: (no) yes     x: (no) yes     x: (yes) yes
 *
 * MAP_PRIVATE  r: (no) no      r: (yes) yes    r: (no) yes     r: (no) yes
 *              w: (no) no      w: (no) no      w: (copy) copy  w: (no) no
 *              x: (no) no      x: (no) yes     x: (no) yes     x: (yes) yes
 */
pgprot_t protection_map[16] __ro_after_init = {
        __P000, __P001, __P010, __P011, __P100, __P101, __P110, __P111,
        __S000, __S001, __S010, __S011, __S100, __S101, __S110, __S111
};
```

```
pgprot_t vm_get_page_prot(unsigned long vm_flags)
{
        pgprot_t ret = __pgprot(pgprot_val(protection_map[vm_flags &
                                (VM_READ|VM_WRITE|VM_EXEC|VM_SHARED)]) |
                        pgprot_val(arch_vm_get_page_prot(vm_flags)));

        return arch_filter_pgprot(ret);
}
```

```
#define _PROT_DEFAULT		(PTE_TYPE_PAGE | PTE_AF | PTE_SHARED)
#define _PAGE_DEFAULT		(_PROT_DEFAULT | PTE_ATTRINDX(MT_NORMAL))

#define PAGE_NONE		__pgprot(((_PAGE_DEFAULT) & ~PTE_VALID) | PTE_PROT_NONE | PTE_RDONLY | PTE_NG | PTE_PXN | PTE_UX
/* shared+writable pages are clean by default, hence PTE_RDONLY|PTE_WRITE */
#define PAGE_SHARED		__pgprot(_PAGE_DEFAULT | PTE_USER | PTE_RDONLY | PTE_NG | PTE_PXN | PTE_UXN | PTE_WRITE)
#define PAGE_SHARED_EXEC	__pgprot(_PAGE_DEFAULT | PTE_USER | PTE_RDONLY | PTE_NG | PTE_PXN | PTE_WRITE)
#define PAGE_READONLY		__pgprot(_PAGE_DEFAULT | PTE_USER | PTE_RDONLY | PTE_NG | PTE_PXN | PTE_UXN)
#define PAGE_READONLY_EXEC	__pgprot(_PAGE_DEFAULT | PTE_USER | PTE_RDONLY | PTE_NG | PTE_PXN)

#define __P000  PAGE_NONE
#define __P001  PAGE_READONLY
#define __P010  PAGE_READONLY
#define __P011  PAGE_READONLY
#define __P100  PAGE_READONLY_EXEC
#define __P101  PAGE_READONLY_EXEC
#define __P110  PAGE_READONLY_EXEC
#define __P111  PAGE_READONLY_EXEC

#define __S000  PAGE_NONE
#define __S001  PAGE_READONLY
#define __S010  PAGE_SHARED
#define __S011  PAGE_SHARED
#define __S100  PAGE_READONLY_EXEC
#define __S101  PAGE_READONLY_EXEC
#define __S110  PAGE_SHARED_EXEC
#define __S111  PAGE_SHARED_EXEC
```

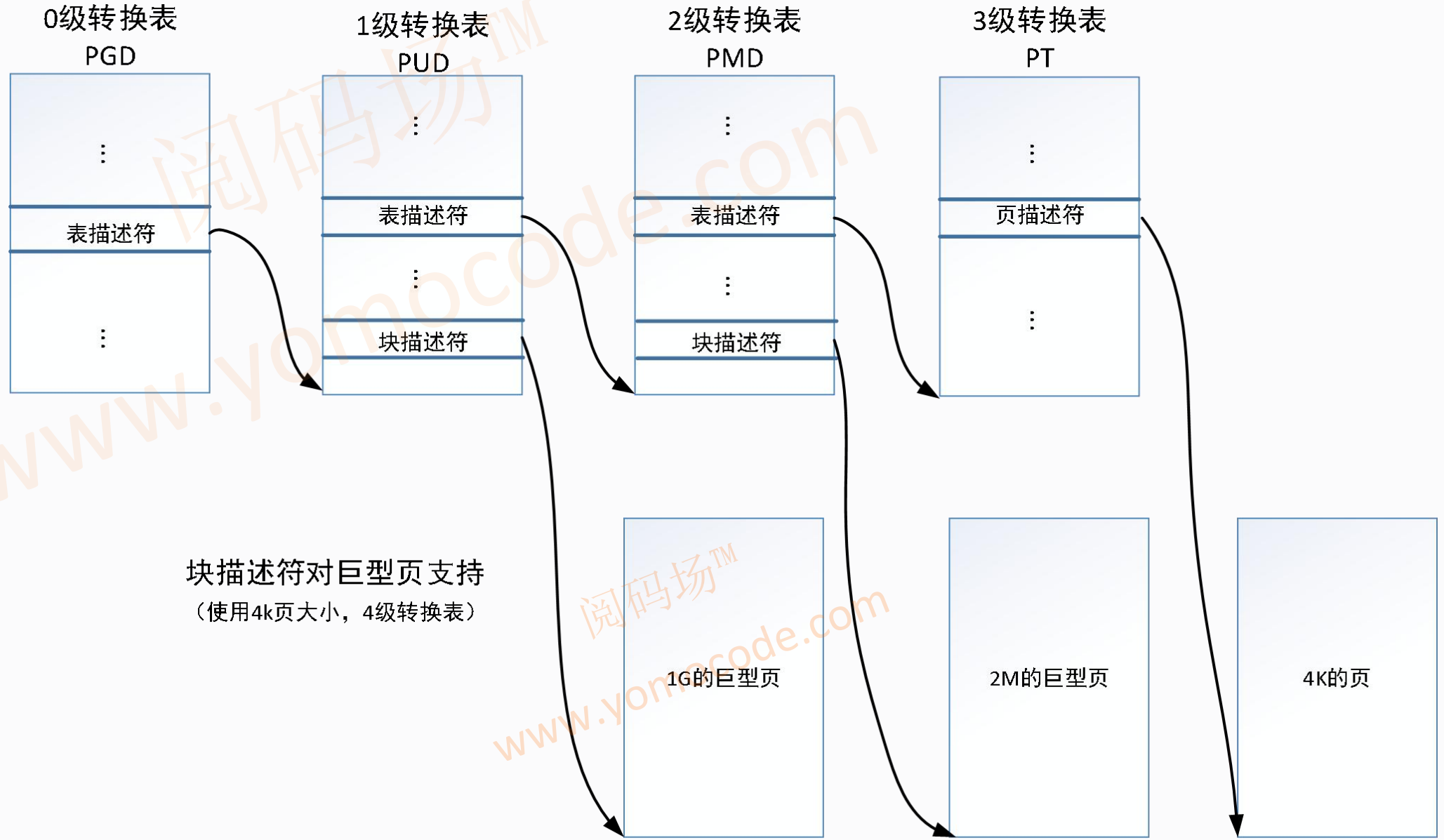

2.5 页表对巨型页的支持

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**。



块描述符对巨型页支持

（使用4k页大小，4级转换表）

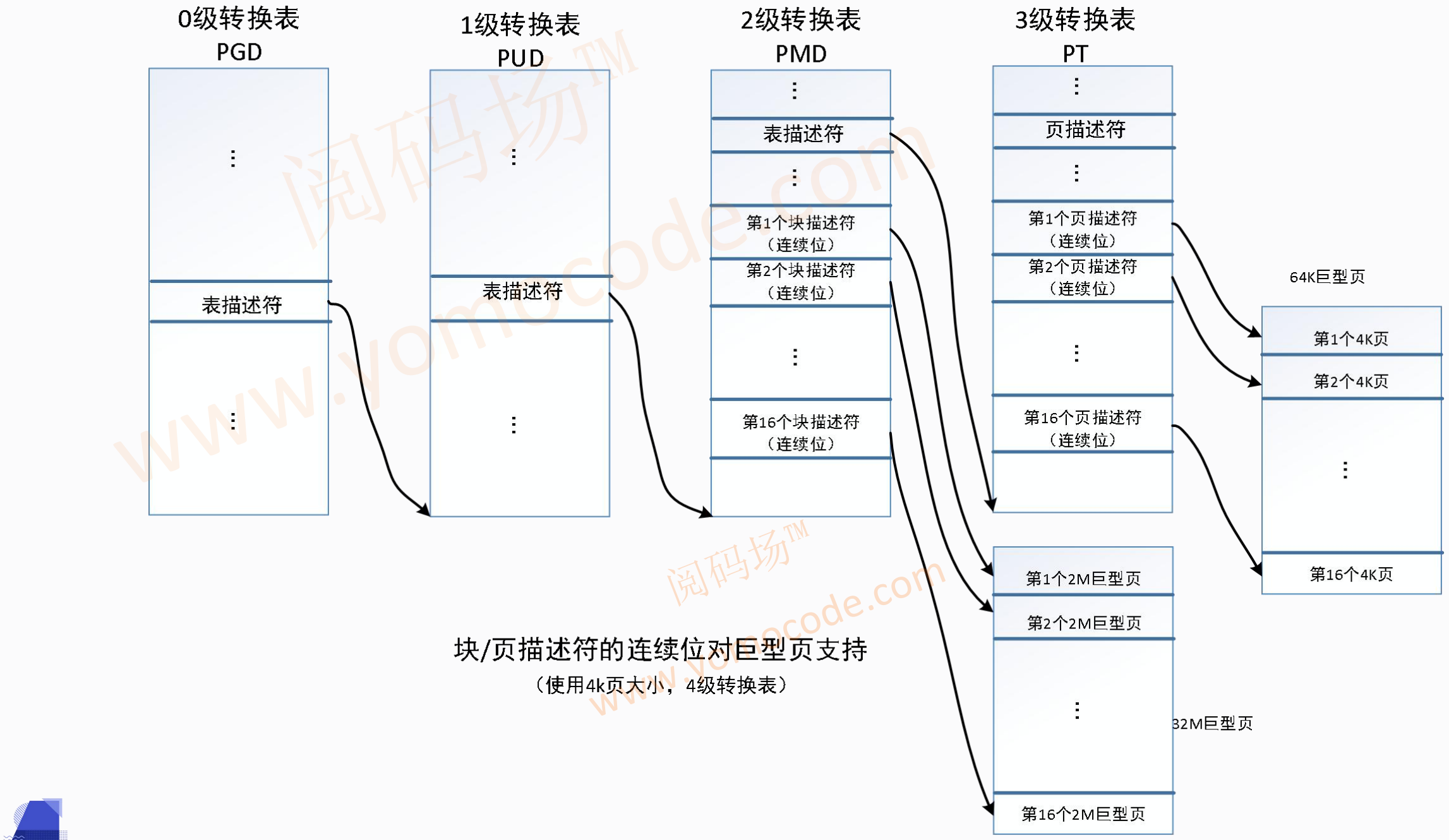
2)块/页描述符连续位支持巨型页

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。



块/页描述符的连续位对巨型页支持

（使用4k页大小，4级转换表）

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）

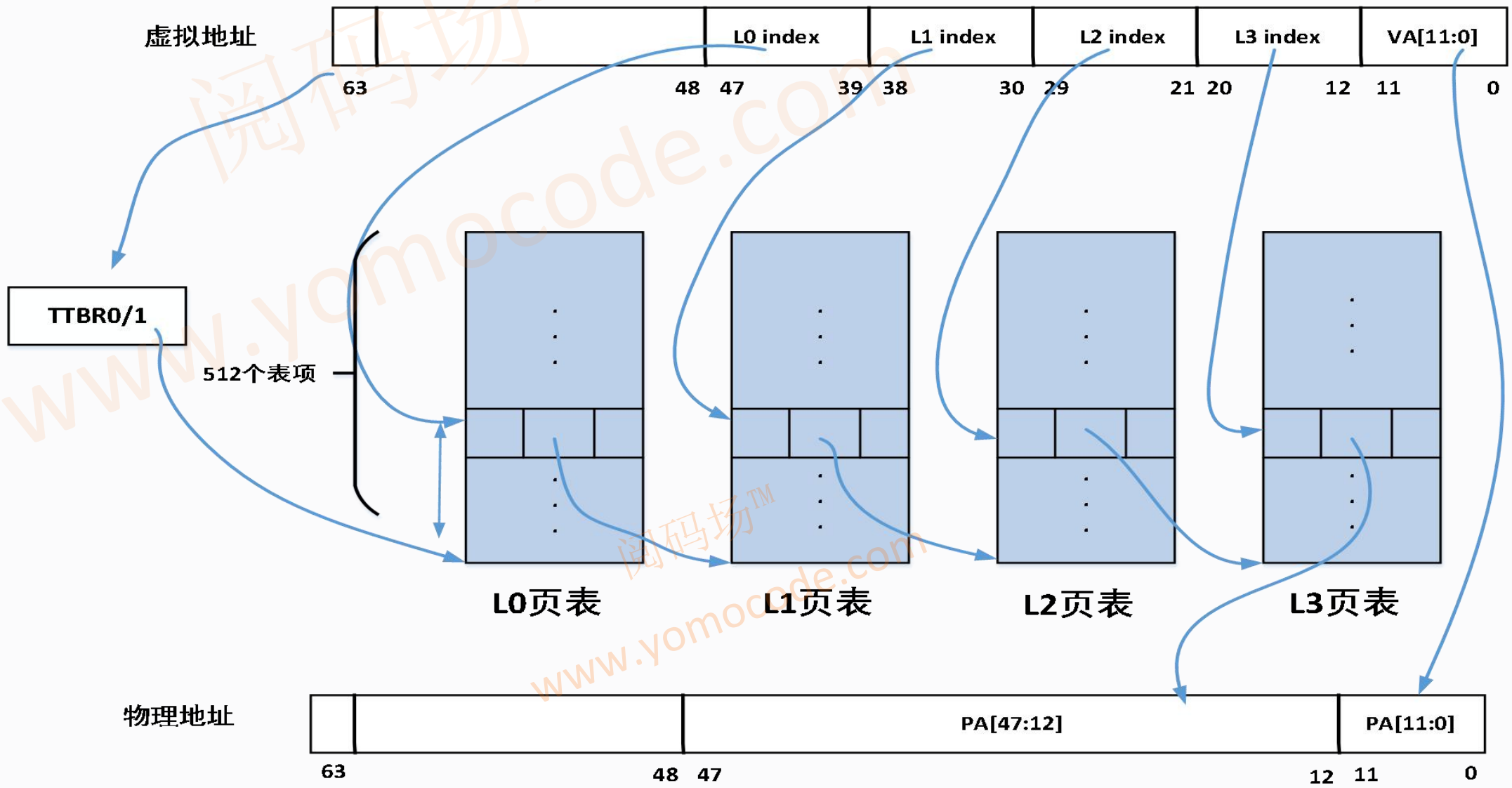
Linux 4.11之后，将页表扩展到5级，在页全局目录和页上级目录之间增加了页四级目录（Page 4th Directory, P4D）

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：

```
static inline p4d_t *p4d_offset(pgd_t *pgd, unsigned long address)
{
        return (p4d_t *)pgd;
}
```


虚拟地址
L0 index
L1 index
L2 index
L3 index
VA[11:0]
63
48
47
39
38
30
29
21
20
12
11
0
TTBR0/1
512个表项
L0页表
L1页表
L2页表
L3页表
物理地址
PA[47:12]
PA[11:0]
63
48
47
12
11
0

3.2 内核中遍历实现

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);

->vmf.pud = pud_alloc(mm, p4d, address);

->vmf.pmd = pmd_alloc(mm, vmf.pud, address);

->handle_pte_fault

-> vmf->pte = pte_offset_map(vmf->pmd, vmf->address);

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。

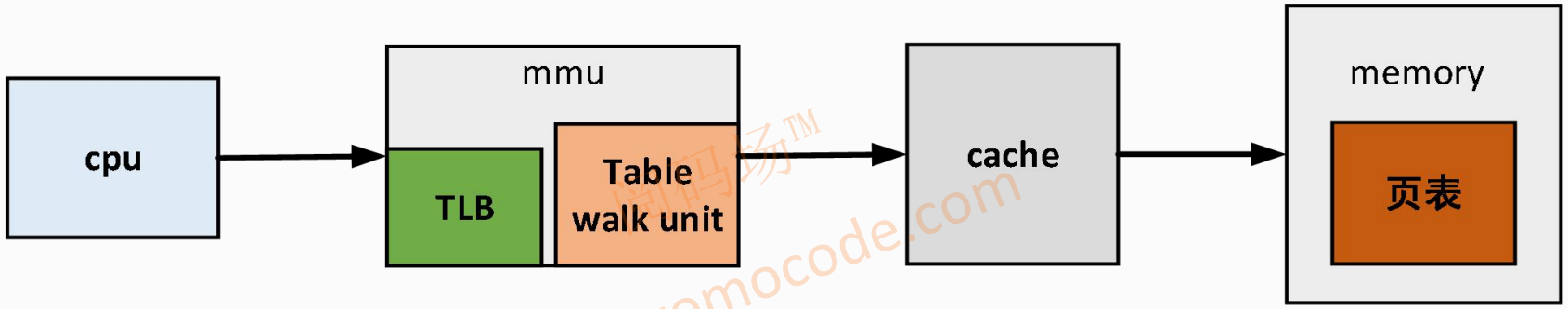
TLB（Translation Lookaside Buffer）转换旁路缓冲区，又成为块表，可以理解为页表缓存，用来缓存最近使用的页表项。

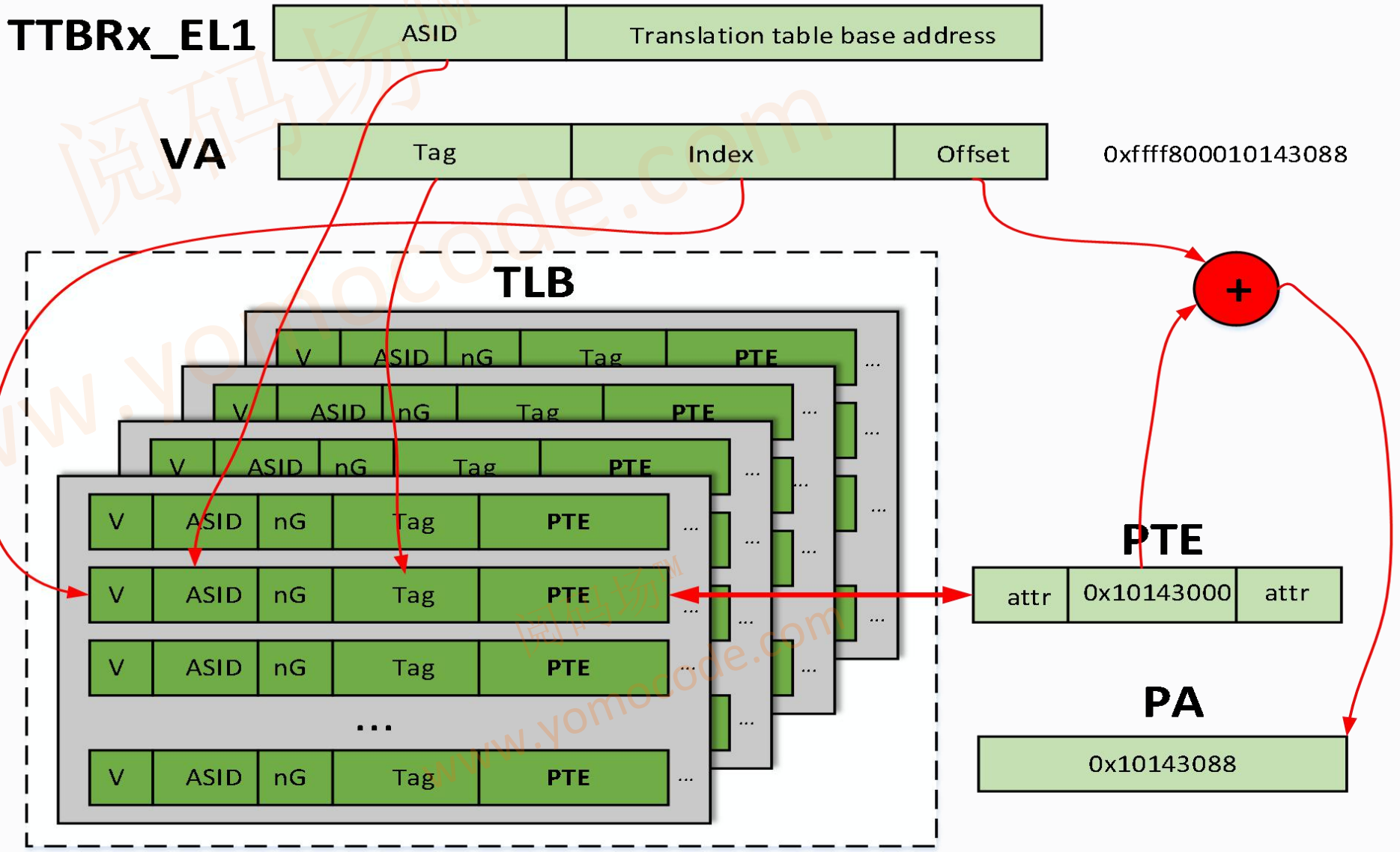
有些处理器有两级TLB：第一级TLB分为指令TLB和数据TLB, 访问指令和数据可以并行执行；
第二级的TLB为统一的tlb(Unfied TLB), 指令和数据公用TLB。

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，成为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。




TTBRx_EL1
ASID
Translation table base address
VA
Tag
Index
Offset
0xffff800010143088
TLB
V
ASID
nG
Tag
PTE
+
PTE
attr
0x10143000
attr
PA
0x10143088

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：

代码路径	api	说明
arch/arm64/include/asm/tlbflush.h	void flush_tlb_all(void) / void local_flush_tlb_all(void)	使所有核的所有的tlb表项无效 / 当前核所有的tlb表项无效
	void flush_tlb_mm(struct mm_struct *mm)	指定的地址空间mm的所有tlb表项无效
	void flush_tlb_range(struct vm_area_struct *vma,unsigned long start, unsigned long end)	指定的用户地址空间的某个范围tlb表项无效
	flush_tlb_page(struct vm_area_struct *vma,unsigned long uaddr)	指定的用户地址空间的虚拟页tlb表项无效
	flush_tlb_kernel_range(unsigned long start, unsigned long end)	内核的虚拟地址空间范围的tlb表项无效
arch/arm64/include/asm/pgtable.h	update_mmu_cache	修改页表项之后将页表项设置到tlb 软件管理tlb的处理器必须实现，如mips架构。 ARM64架构函数为空。

4.4 flush tlb的arm64实现

当tlb miss的时候，ARM64处理器架构的mmu会自动遍历内存中的各级页表，然后将页表项缓存到tlb中，不需要软件去写页表项到tlb,所有ARM64没有写tlb的指令，但是提供将tlb表项无效的指令：

TLBI <type><level>{IS} {, <Xt>}

各字段的含义：
->type

ALL	所有表项
VMALL	当前虚拟机的阶段1的所有表项，表项的VMID是当前虚拟机 的VMID
VMALLS12	当前虚拟机的阶段1和阶段2的所有表项，表项的VMID是当前虚拟机 的VMID
ASID	Xt指定的ASID
VA	Xt指定的虚拟地址和ASID
VAA	Xt指定的虚拟地址，ASID可以时任意的

->level
E1 ： 异常级别1
E2：异常级别2
E3：异常级别3
->IS
表示内部共享(Inner Shareable),即多个核共享，不使用IS，表示非共享，仅仅被一个核使用。在SMP系统中TLBI指令没有IS字段表示使当前核tlb表项无效，如果有IS字段表示时所有核的tlb表项无效。
->Xt
X0-X31任何一个寄存器

```c
static inline void flush_tlb_all(void)
{
        dsb(ishst);
        __tlbi(vmalle1is);
        dsb(ish);
        isb();
}
```

```c
static inline void local_flush_tlb_all(void)
{
        dsb(nshst);
        __tlbi(vmalle1);
        dsb(nsh);
        isb();
}
```

```c
#define __TLBI_0(op, arg) asm ("tlbi " #op "\n"                                 \
                ALTERNATIVE("nop\n                  nop",                       \
                            "dsb ish\n              tlbi " #op,                 \
                            ARM64_WORKAROUND_REPEAT_TLBI,                       \
                            CONFIG_ARM64_WORKAROUND_REPEAT_TLBI)                \
                            : : )

#define __TLBI_1(op, arg) asm ("tlbi " #op ", %0\n"                             \
                ALTERNATIVE("nop\n                  nop",                       \
                            "dsb ish\n              tlbi " #op ", %0",          \
                            ARM64_WORKAROUND_REPEAT_TLBI,                       \
                            CONFIG_ARM64_WORKAROUND_REPEAT_TLBI)                \
                            : : "r" (arg))

#define __TLBI_N(op, arg, n, ...) __TLBI_##n(op, arg)

#define __tlbi(op, ...)         __TLBI_N(op, ##__VA_ARGS__, 1, 0)
```

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区别不用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　　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

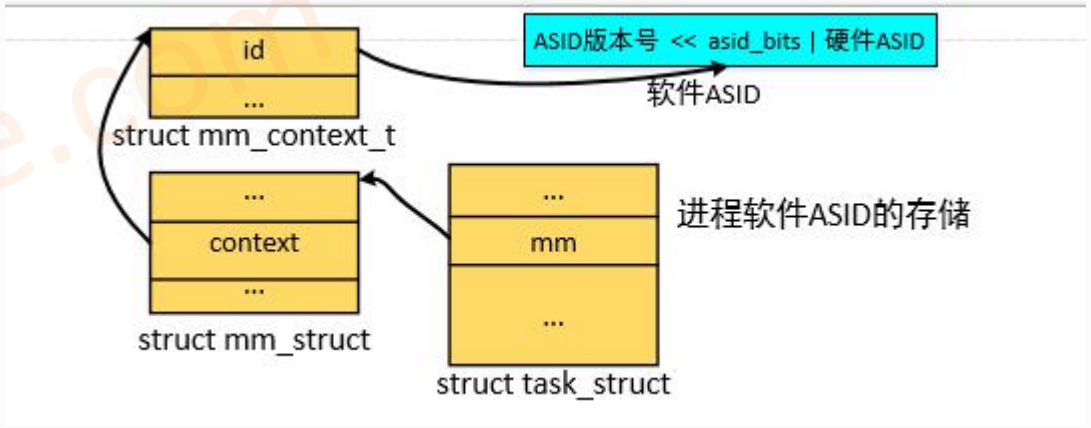
```
static int asids_init(void)
{
        asid_bits = get_cpu_asid_bits();
        /*
        * Expect allocation after rollover to fail if we don't have at least
        * one more ASID than CPUs. ASID #0 is reserved for init_mm.
        */
        WARN_ON(NUM_USER_ASIDS - 1 <= num_possible_cpus());
        atomic64_set(&asid_generation, ASID_FIRST_VERSION);
        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);

        pr_info("ASID allocator initialised with %lu entries\n", NUM_USER_ASIDS);
        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; })

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();

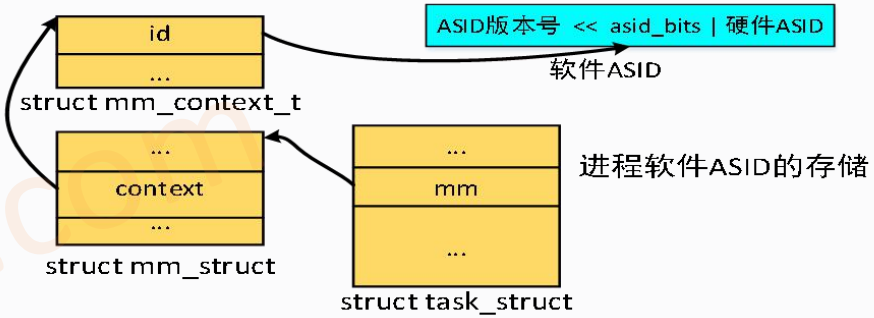
5) 切换地址空间

check_and_switch_context

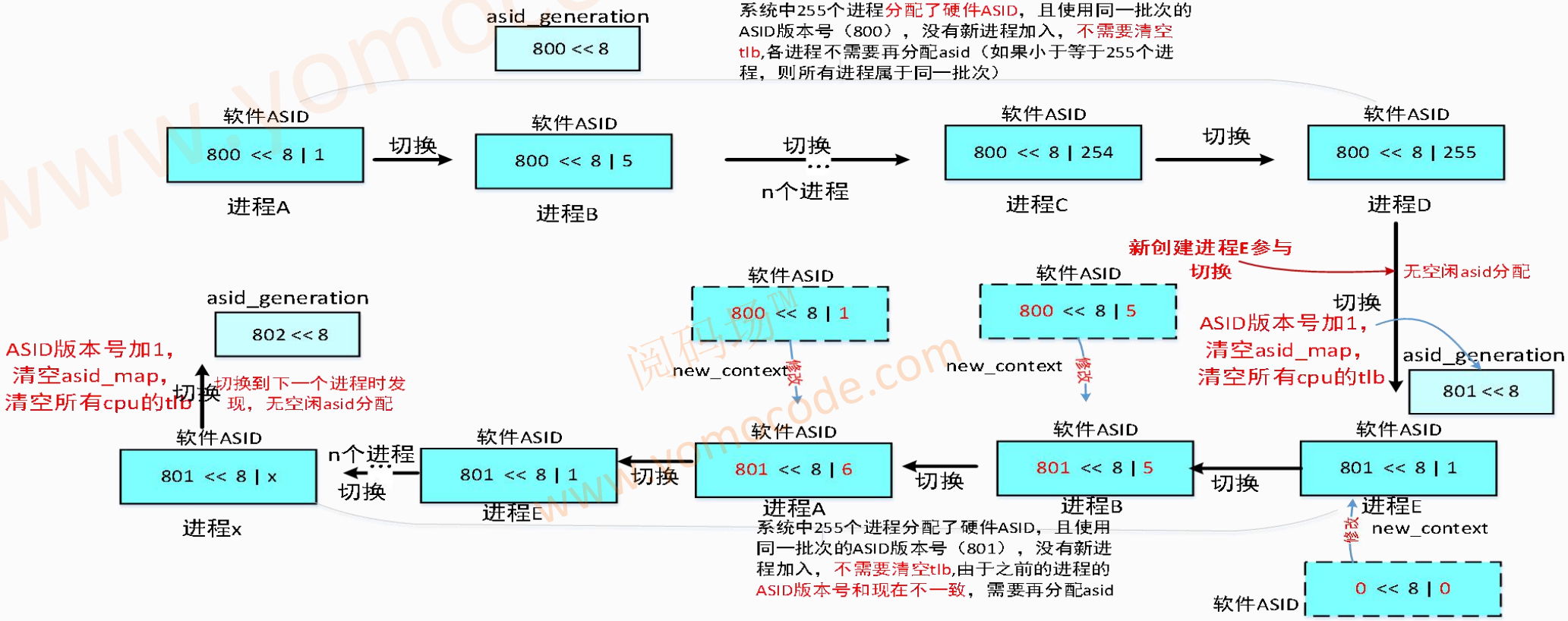
->cpu_switch_mm(mm->pgd, mm) //切换pgd和ASID

5.3 实例讲解

约定：系统中有N个进程，采样时的asid_generation（全局的ASID版本号）为800，asid_bits（ASID长度）为8位（可以在同一批次为255 个进程分配硬件ASID，0号为保留）。



ASID分配图解



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))
```

```
//页表项判断
/*页是否存在内存中*/
#define pte_present(pte)       (!!(pte_val(pte) & (PTE_VALID | PTE_PROT_NONE)))
/*页是否被访问过*/
#define pte_young(pte)         (!!(pte_val(pte) & PTE_AF))
/*页是否是特殊的*/
#define pte_special(pte)       (!!(pte_val(pte) & PTE_SPECIAL))
/*页是否可写*/
#define pte_write(pte)         (!!(pte_val(pte) & PTE_WRITE))
/*页是否存用户可执行*/
#define pte_user_exec(pte)     (!(pte_val(pte) & PTE_UXN))
/*页是否为连续表项中的一项*/
#define pte_cont(pte)          (!!(pte_val(pte) & PTE_CONT))
/*页是否为脏，被修改过*/
#define pte_dirty(pte)         (pte_sw_dirty(pte) || pte_hw_dirty(pte))
/*页是否有效*/
#define pte_valid(pte)         (!!(pte_val(pte) & PTE_VALID))
```

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. 启动阶段早期的页表创建

7.1 建立恒等映射

arch/arm64/kernel/head.S:

stext

->__create_page_tables

->map_memory xxx

(段映射，3级页表结构,pgd->pud->pmd，一个表项映射2M大小内存)

idmap_pg_dir 恒等映射（va = pa）的pgd页目录地址(建立三个表项)，会保存到ttbr0_el1

7.2 建立粗粒度内核页表映射

arch/arm64/kernel/head.S:

stext

->__create_page_tables

->map_memory xxx

(段映射，3级页表结构,pgd->pud->pmd，一个表项映射2M大小内存)

init_pg_dir 内核镜像做段映射的pgd页目录地址，会保存到ttbr1_el1

7.2 打开mmu

arch/arm64/kernel/head.S:

stext

->__cpu_setup //arch/arm64/mm/proc.S

->设置mair_el1来设置内存属性

->__primary_switch

  ->__enable_mmu //打开mmu

设置init_pg_dir到ttbr1_el1

设置idmap_pg_dir到ttbr0_el1

设置sctlr_el1 的m位，来打开mmu！！！

8. fixmap映射

8.1 fixmap

内核初始化的早期，内核已经运行在虚拟地址上，建立恒等映射和粗粒度内核页表映射只能保证内核镜像的可以正常访问，这个时候如何我们想访问bootloader传递过来的dtb怎么办？这个时候内存管理子系统还没有准备好，想通过ioremap访问外设的寄存器怎么办？为此kernel提出fixmap。

代码路径：

start_kernel

->setup_arch　　//arch/arm64/kernel/setup.c

->early_fixmap_init //fixmap区映射

->early_ioremap_init //早期ioremap映射

->setup_machine_fdt(__fdt_pointer) //设备树映射和设置

8.2 早期ioremap和earlycon

早期点灯，串口打印等！

代码路径：

start_kernel

->setup_arch　　//arch/arm64/kernel/setup.c

  ->early_fixmap_init //fixmap区映射

  ->early_ioremap_init //早期ioremap映射

  ->setup_machine_fdt(__fdt_pointer) //设备树映射和设置

eg：早期串口

early_param("earlycon", param_setup_earlycon)　//drivers/tty/serial/earlycon.c

param_setup_earlycon

->setup_earlycon

  ->register_earlycon

    ->earlycon_map

      ->set_fixmap_io

8.3 dtb映射

映射设备树，通过虚拟地址获得内存信息和板级信息。

代码路径：

start_kernel

->setup_arch　　//arch/arm64/kernel/setup.c

->early_fixmap_init //fixmap区映射

->early_ioremap_init //早期ioremap映射

->setup_machine_fdt(__fdt_pointer) //设备树映射和设置

setup_machine_fdt

->fixmap_remap_fdt(dt_phys, &size, PAGE_KERNEL)　//映射设备树到fixmap

-> -> create_mapping_noalloc(round_down(dt_phys, SWAPPER_BLOCK_SIZE),

dt_virt_base, SWAPPER_BLOCK_SIZE, prot);

->fixmap_remap_fdt(dt_phys, &size, PAGE_KERNEL_RO)

9. 主内核页表创建

9.1 细粒度内核页表建立

start_kernel

->setup_arch　　//arch/arm64/kernel/setup.c

->early_fixmap_init //fixmap区映射

->early_ioremap_init //早期ioremap映射

->paging_init　　//arch/arm64/mm/mmu.c

->**map_kernel**　**//细粒度内核页表建立**

->map_mem　　//线性映射区页表建立

swapper_pg_dir 保存主内核页表pgd页目录地址。

进入el1时，访问内核地址空间，如访问内核函数。

9.2 线性映射区页表建立

start_kernel

->setup_arch　　//arch/arm64/kernel/setup.c

->early_fixmap_init //fixmap区映射

->early_ioremap_init //早期ioremap映射

->paging_init　　//arch/arm64/mm/mmu.c

->map_kernel　　//细粒度内核页表建立

->**map_mem**　　**//线性映射区页表建立**

swapper_pg_dir 保存主内核页表pgd页目录地址。

通过__va访问pa。

10. 用户进程页表创建

10.1 创建进程fork时

1.fork时分配进程pgd页目录

_do_fork

->copy_process

->copy_mm

->dup_mm

->mm_init

->mm_alloc_pgd

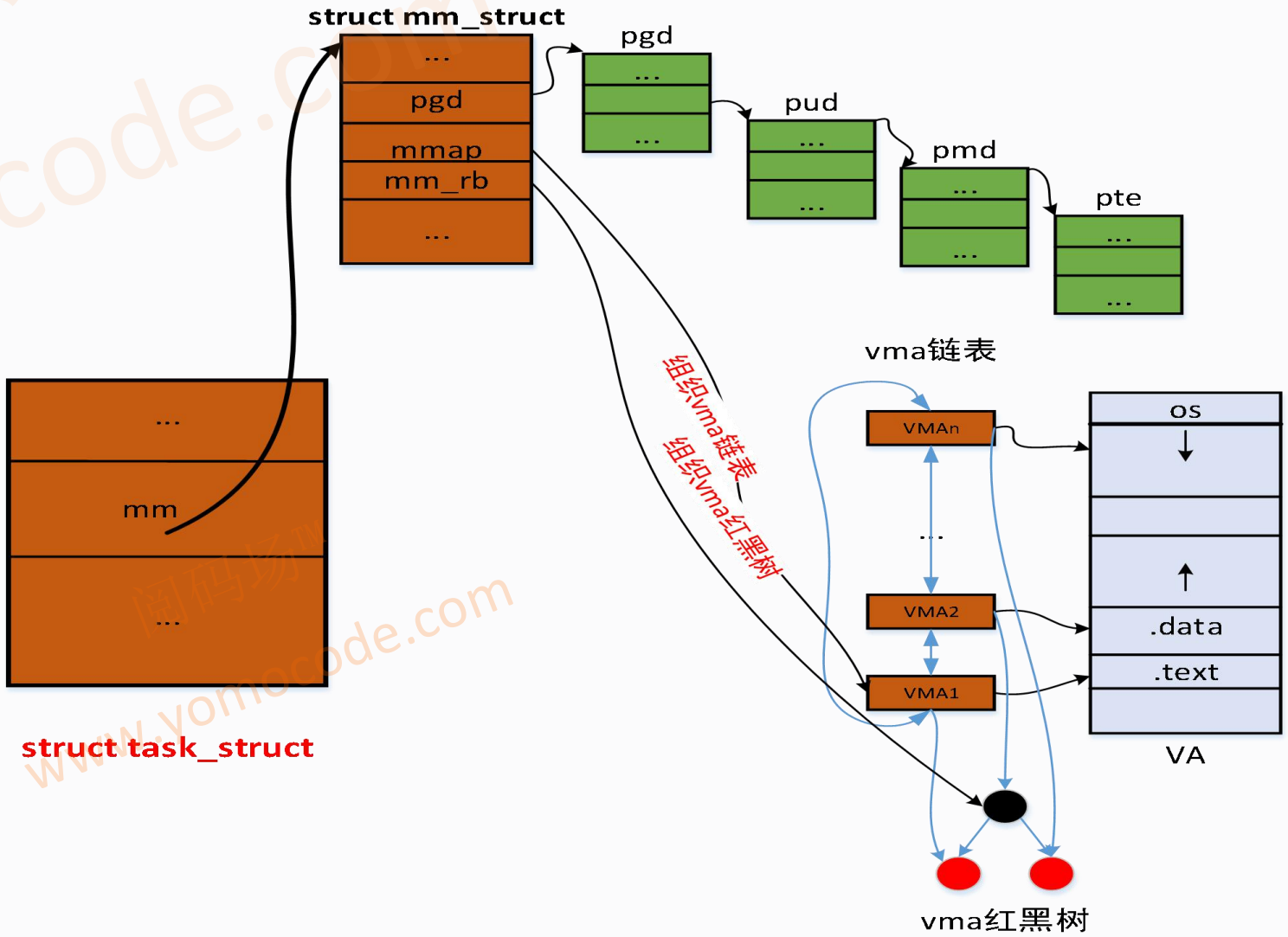
2.拷贝父进程页表

->copy_mm

->dup_mm

->dup_mmap

->copy_page_range



10.2 缺页异常时

按需分配/按需掉页，建立虚拟地址和物理地址映射关系。

handle_pte_fault　　mm/memory.c

->do_anonymous_page //匿名映射页缺页为例

->alloc_zeroed_user_highpage_movable //分配物理页

->mk_pte //组合页表项

->set_pte_at //设置页表项

10.3 进程切换时

__schedule //kernel/sched/core.c 主调度器

->context_switch //进程上下文切换

 ->switch_mm_irqs_off //进程地址空间切换

  ->switch_mm //arch/arm64/include/asm/mmu_context.h

   ->__switch_mm

    ->cpu_switch_mm(mm->pgd, mm)

     ->cpu_do_switch_mm(virt_to_phys(pgd),mm) //arch/arm64/mm/proc.S

```
145 ENTRY(cpu_do_switch_mm)
146         mrs     x2, ttbr1_el1
147         mmid    x1, x1                          // get mm->context.id
148         phys_to_ttbr x3, x0
149
150 alternative_if ARM64_HAS_CNP
151         cbz     x1, 1f                          // skip CNP for reserved ASID
152         orr     x3, x3, #TTBR_CNP_BIT
153 1:
154 alternative_else_nop_endif
155 #ifdef CONFIG_ARM64_SW_TTBR0_PAN
156         bfi     x3, x1, #48, #16                // set the ASID field in TTBR0
157 #endif
158         bfi     x2, x1, #48, #16                // set the ASID
159         msr     ttbr1_el1, x2                   // in TTBR1 (since TCR.A1 is set)
160         isb
161         msr     ttbr0_el1, x3                   // now update TTBR0
162         isb
163         b       post_ttbr_update_workaround     // Back to C code...
164 ENDPROC(cpu_do_switch_mm)
```

设置进程的ASID到ttbr1_el1,

设置mm->pgd到ttbr0_el1完成了地址空间切换！！！

11. ioremap原理

11.1 ioremap相关概念

一般情况下，处理器访问外设通过读写设备的寄存器来进行，如控制寄存器、状态寄存器、数据寄存器等。

根据处理器架构的不同，处理器对IO端口的编址方式有两种：

1） I/O 映射方式（I/O-mapped）

典型地，如X86处理器为外设专门实现了一个单独的地址空间，称为"I/O地址空间"或者"I/O端口空间"，CPU通过专门的I/O指令（如X86的IN和OUT指令）来访问这一空间中的地址单元。

2） 内存映射方式（Memory-mapped）

RISC指令系统的CPU（如ARM、PowerPC等）通常只实现一个物理地址空间，外设I/O端口成为内存的一部分。此时，CPU可以象访问一个内存单元那样访问外设I/O端口，而不需要设立专门的外设I/O指令。

一般来说，在系统运行时，外设的I/O内存资源的物理地址是已知的，由硬件的设计决定。打开mmu后，cpu访问的是虚拟地址，所以必须将外设I/O的物理地址映射为虚拟地址（通过页表的方式），这时才能通过虚拟地址来访问外设的I/O内存资源（控制外设的寄存器）。

内核中提供ioremap相关api供驱动使用来映射外设寄存器：

```
#include <linux/io.h>

 #define ioremap(addr, size)          __ioremap((addr), (size), __pgprot(PROT_DEVICE_nGnRE))
void iounmap(volatile void __iomem *io_addr)
```


11.2 ioremap原理

arch/arm64/include/asm/io.h

#define ioremap(addr, size)		__ioremap((addr), (size), __pgprot(PROT_DEVICE_nGnRE))

->__ioremap		//arch/arm64/mm/ioremap.c

->__ioremap_caller

-> area = get_vm_area_caller(size, VM_IOREMAP, caller) //获得一个vmalloc区

->ioremap_page_range(addr, addr + size, phys_addr, prot) //物理地址映射到vmalloc区的虚拟地址

12. 实践

通过debugfs查看内核页表：

添加配置选项：

CONFIG_ARM64_PTDUMP_CORE=y

CONFIG_ARM64_PTDUMP_DEBUGFS=y

源码实现路径：arch/arm64/mm/dump.c

用户层接口：/sys/kernel/debug/kernel_page_tables

Thanks!!!

唯有热爱可抵岁月漫长!!!

阅码场出品

www.yomocode.com