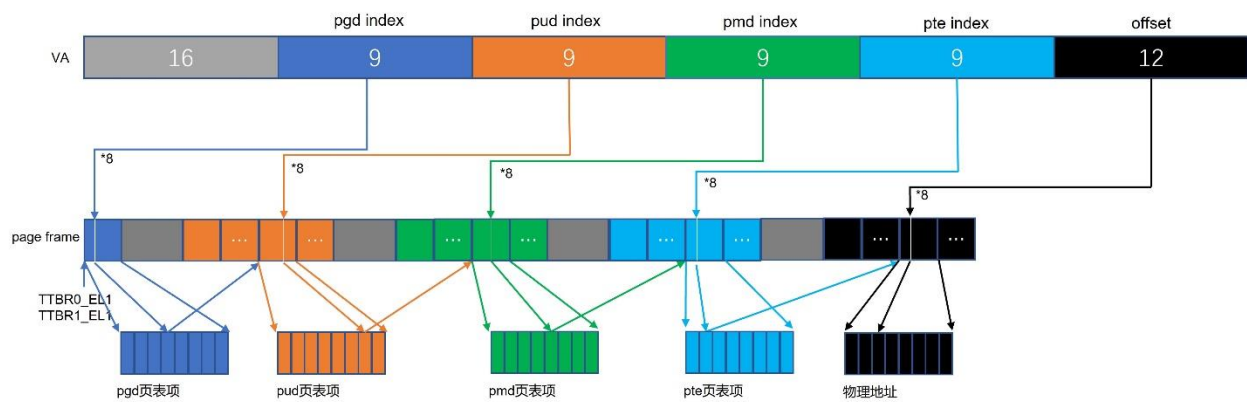


# Linux-ARM64 页表初始化

## 一、何为页表

### 1.1 地址转换流程



4KBpage，48 位有效虚拟地址四级页表转换过程示意图

这是 4KB 的页，使用 4 级页表的地址转换示意图，64 位的虚拟地址被分成 16bit+9bit\*4+12bit，低 12 位对应 4KB 的页大小，最后一级页表描述符中保存着这个虚拟地址所在的页对应的物理页框(物理页)，在加上低虚拟地址的低 12 位就得到虚拟地址对应的物理地址。每一级页表的描述符以数组的形式保存在内存中，上级页表描述符中记录有下一级页表项所在的页框，虚拟地址的 bit[47:39]、bit[38:30]、bit[29:21]和 bit[20:12]分别为四级页表的索引值。由于每条页表描述符占用 64 位（8 字节），所以把页表所在的页框加上对应索引值乘以 8 就得到虚拟地址在这一级页表中的描述符。最高级的页表描述符所在的页框保存在寄存器 TTBR0 和 TTBR1 中，分别用于用户态和内核态的页表翻译。虚拟地址的高 16 位在地址转换过程中用不到，有些位被用来作为特殊的 flag。

上面说到的页表描述符似乎有点抽象，具体格式可以看下图。bit[47:12]是下一级页表所在的页框号(pfn)，页框的大小是 4KB 的话，页框的地址就是页框号乘以 4096，也就是低 12 位全为 0。那么页表描述符低 12 位可以作为一下特殊用途，比如 bit0 为 0 时，表示无效的页表项；bit[1:0]为 01 时，表示 block 页表项，当 MMU 查找到 block 页表项或者最后一级页表项的时候就完成了页表的翻译；bit[1:0]为 11 时，表示 table 页表项，会继续往下一级页表检索。

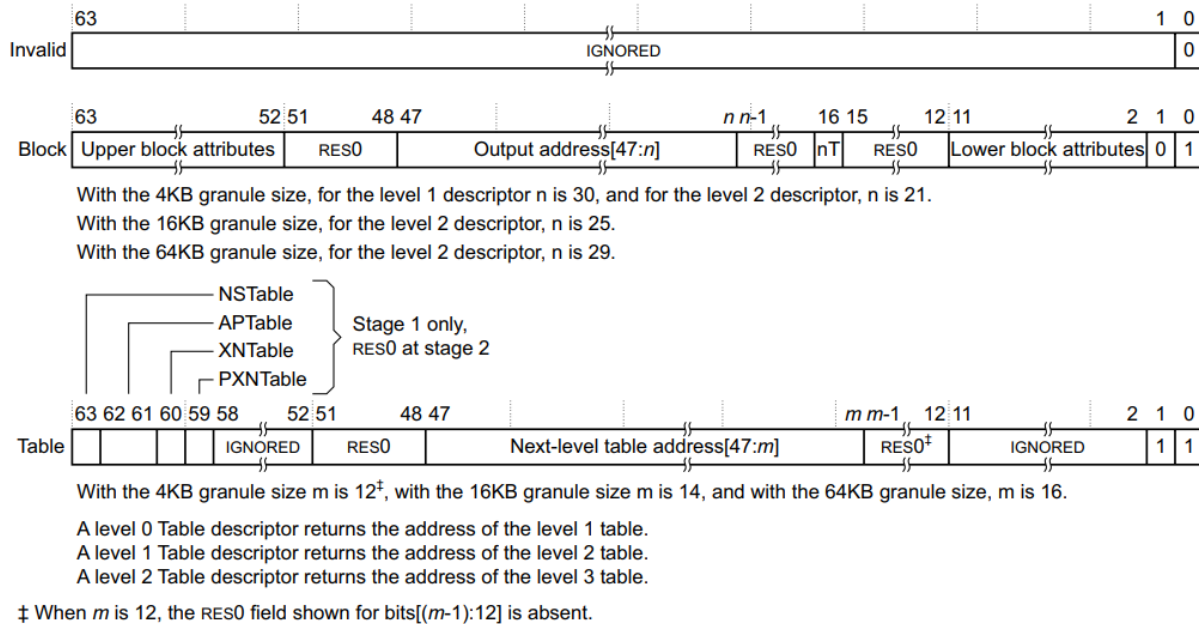


Figure D5-15 VMSAv8-64 level 0, level 1 and level 2 descriptor formats with 48-bit OAs

来自 ARM 官方文档的页表描述符类型及格式

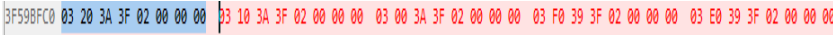
这里举个例子来更深入理解地址转换的过程。本例子中使用的是三级页表，因此虚拟地址有效位数是 39 位。环境：lahaina 平台 ramdump+Trace32+beyondcompare（或其他二进制编辑工具），dump 路径 [\\ssrk-sh-win01\LSS\\_China\Engineers\wangxuanfu\pagetable-test](#)。

从 dump 中可以知道这台手机有 6 片 DDR，每片 DDR 是 2GB 容量，相应的物理地址范围是 DDRCS0\_0.BIN@80000000--ffffff, DDRCS0\_1.BIN@100000000--17ffffff, DDRCS0\_2.BIN@180000000--1ffffff, DDRCS1\_0.BIN@200000000--27ffffff, DDRCS1\_1.BIN@280000000--2ffffff, DDRCS1\_2.BIN@300000000--37ffffff。首先，找一个虚拟地址作为研究对象。这里选取 printk 打印日志时候使用的全局变量 log\_buf，虚拟地址为 0xFFFFFAB7F0626C0。计算各级页表的索引值，pgd\_index=0xAD\*8=0x568, pmd\_index=0x1F8\*8=0xFC0, pte\_index=0x62\*8=0x310, offset=0x6C0。

然后查看 TTBR1\_EL1 寄存器的值为 0xA2634000。下面开始页表的检索。pgd 页表描述符的地址为 &pgd\_desc=TTBR1\_EL1+pgd\_index=0xA2634568，这个地址落在 DDRCS0\_0.BIN 中，在 bin 中的偏移地址为 0xA2634568-0x80000000=0x22634568，读出来页表描述符为 pgd\_desc=0x023F59B003，注意数据大小端转化

22634558 03 D0 99 3F 02 00 00 00 03 C0 79 3F 02 00 00 00 03 B0 59 3F 02 00 00 00 03 A0 60 2C 02 00 00 00 03 B0 07 2A 02 00 00 00

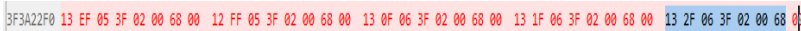
，对照页表描述符的格式，这是一个有效的 table 类型的页表，下一级 pmd 页表所在的页框是 pmd=0x23F59B000。Pmd 页表描述符地址为 &pmd\_desc=pmd+pmd\_index=0x23F59BFC0。这个地址落在 DDRCS1\_0.BIN 的偏移地址 0x3F59BFC0 处。读出来页表描述符为

pmd\_desc=0x023F3A2003  ,

这也是一个有效的 table 类型的页表，下一级 pte 页表所在的页框是 pte=0x23F3A2000。

Pte 页表描述符地址为 &pte\_desc=pte+pte\_index=0x23F3A2310，这个地址落在

DDRCS1\_0.BIN 的偏移地址 0x3F3A2310 处。读出来页表描述符为

pte\_desc=0x6800023F062F13  ,

这也是一个有效的 table 类型的页表，到这里就已经找到物理地址所在的页框为

page\_frame=0x23F062000。最后一步得到物理地址 pa=page\_frame+offset=0x23F0626C0。

作为验证，使用 crash 工具执行 vtop 命令进行虚拟地址到物理地址的转换，可以看到结果与我们手动检索的完全一致。

```
crash_64> sym log_buf
fffffffec33cd0208 (d) log_buf
crash_64>
crash_64> rd ffffffec33cd0208
fffffffec33cd0208: ffffffffab7f0626c0          .&.....
crash_64>
crash_64> vtop ffffffffab7f0626c0
VIRTUAL      PHYSICAL
fffffffab7f0626c0 23f0626c0

PAGE DIRECTORY: ffffffec33634000
PGD: ffffffec33634568 => 23f59b003
PMD: ffffffffab7f59bfc0 => 23f3a2003
PTE: ffffffffab7f3a2310 => 6800023f062f13
PAGE: 23f062000

      PTE      PHYSICAL      FLAGS
6800023f062f13 23f062000 (VALID|SHARED|AF|NG|PXN|UXN)

      PAGE      PHYSICAL      MAPPING      INDEX CNT FLAGS
ffffffffffaddc1880 23f062000          0          0 1 1000 reserved
crash_64>
```

## 1.2 多级页表

Linux 为每个进程提供的虚拟地址空间是完全相同的，对于 64 位系统，设置有效虚拟地址位数为 48 的话，每个用户进程可以访问的虚拟地址空间包括用户态地址

0x0000000000000000~0x0000FFFFFFFFFFFFFF 和内核态地址

0xFFFFF00000000000~0xFFFFFFFFFFFFFFFF。实际上所有进程的内核态地址空间是共享的，用户态地址也并不是全部会被使用，而是申请多少才会分配多少。

**TTBR0\_ELx** points to the initial translation table for the lower VA range, that starts at address 0x0000\_0000\_0000\_0000,

**TTBR1\_ELx** points to the initial translation table for the upper VA range, that runs up to address 0xFFFFFFFFFFFFFFFF.

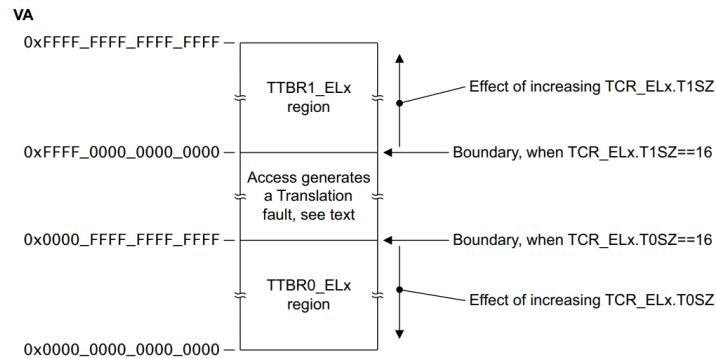
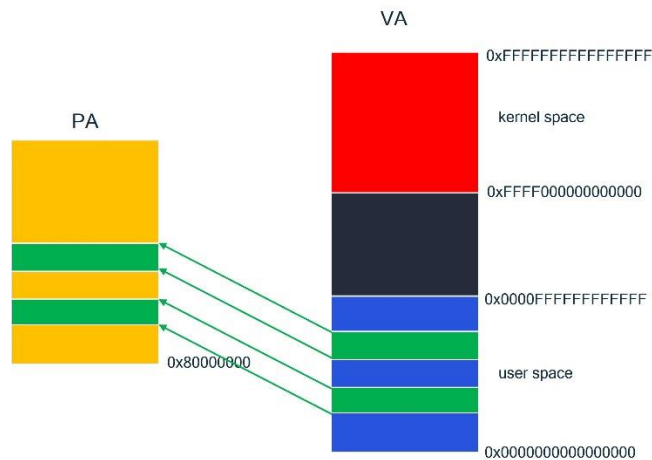


Figure D5-14 AArch64 TTBRn boundaries and VA ranges for 48-bit VAs

## ARM64 地址空间



## 进程的地址映射

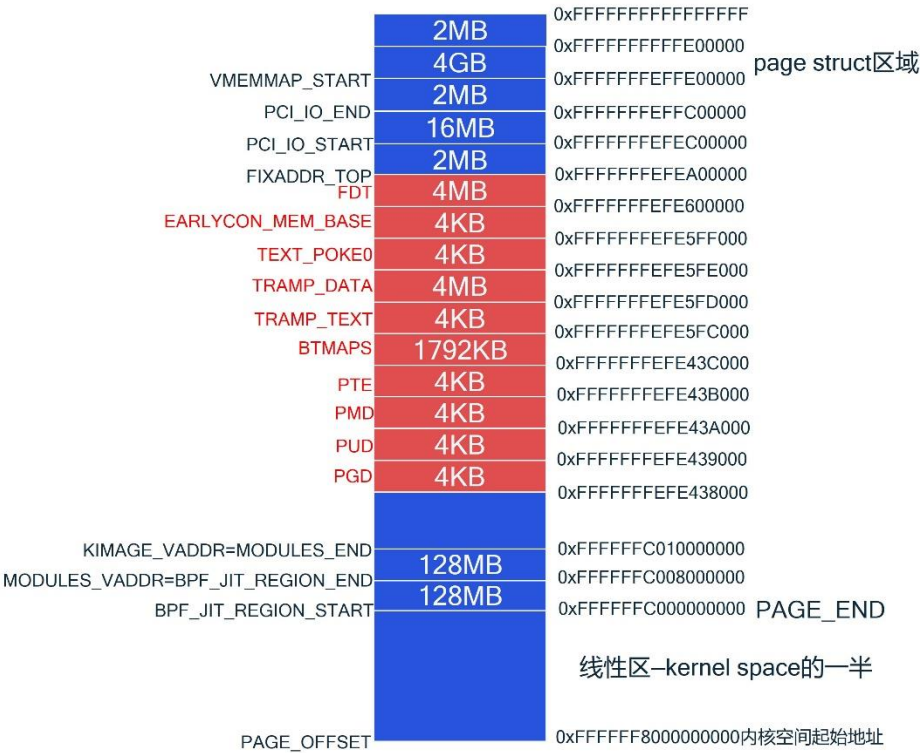
假如一个进程在用户态只使用了绿色部分的内存，显然只需要对这部分内存页建立页表就可以了，而不需要对整个虚拟地址空间做映射。为了保证检索效率，页表是以数组（随机访问的时间复杂度为  $O(1)$ ）的形式保存在内存中的，无法把中间没用到的地址页表去掉（形象的说就是数组的下标不可能是 0, 1, 2, 5, 6..., 而没有了 3, 4）。而采用多级页表之后，仅仅是同一级的页表是以数组的形式保存，下一级的页表地址保存在上级页表的描述符中，这样就可以灵活的对小块内存做映射。如果使用上述例子中的三级页表的话，每个页框中可以保存 512 条页表项，每个 PTE 页表项可以对应一个页框，也就是 1 个 page 的内存中保存的 PTE 页表可以实现  $512 \times 4KB = 2MB$  的地址的转换。假设进程使用的地址是 0~0x200000 和 0x10000000~0x10200000，那么需要 1 个 page 的 PGD 页表项，2 个 page 的 PMD 页表项和 2 个 page 的 PTE 页表项，总共需要 5 个 page（20KB）保存页表。但是如果只有一级页表，至少需要  $0x10200000 / 4KB / 512 = 81$  个 page 来保存页

表。当然页表级数增加之后，页表翻译的过程也会耗时增加。页表的级数和有效虚拟地址位数是相关的，采用三级页表时，有效虚拟地址位数是  $12+3*9=39$  位；采用 4 级页表时，有效虚拟地址位数是  $12+4*9=48$  位。

## 二、启动阶段页表

### 2.1 内核地址空间布局

从 bootloader 跳转到 linux 之后，MMU 是关闭的，CPU 直接访问物理地址。一旦打开 MMU 之后，CPU 访问的都是虚拟地址，包括 PC，LR，SP 等寄存器，只有经过 MMU 翻译页表之后才能访问到对应的物理地址。为了打开 MMU 之后代码能够正常运行，linux 必须先进行页表的初始化，把 linux 镜像建立物理地址到虚拟地址的映射。那么问题来了，对于已经初始化的系统，线程/驱动申请内存的时候，会从内存管理子系统中返回一个虚拟地址，似乎是天经地义的事情。但是启动阶段内存管理子系统都没初始化，从哪里得到虚拟地址呢？这时候的虚拟地址其实是编译时就确定的链接地址。可以打开编译生成的 system.map 文件，linux 的所有符号和全局变量以及各 section 的地址都在里面，也可以参考 vmlinux.lds.s 源文件。下图是 linux5.4 内核的一部分内存布局，没有考虑 kaslr 和 kasan。所谓 kaslr 跟本文的主题关系不大，不详细展开了。简单的说就是出于安全考虑，通过从 bootloader 传递一个随机的 kaslr-seed 参数，内核启动阶段会把这个随机数加到内核镜像的起始地址上，从而实现起始地址随机化。



内核地址空间布局

---

## 2.2 初始页表

在 ARM 平台打开 MMU 的代码被放到一个特殊的 `.idmap.text` 段中，这一区域会被映射两次，一次是对这一区域进行平行映射，也就是虚拟地址等于物理地址，其实这时候的虚拟地址落在地址空间的低地址处（也是系统初始化完成之后的用户态地址空间）。在打开 MMU 的指令执行的时候，CPU 的流水线可能已经把打开 MMU 之后的代码取指完成，而且其地址是物理地址。那么在打开 MMU 之后，那些已经取指完成的代码地址由于缺少页表就会无法访问到对应的物理地址。这里用虚拟地址等于物理地址的方式进行规避。第二次映射是对整个 linux 镜像区域（从地址 `_text` 到 `_end`）进行映射到内核地址空间，由于 `.idmap.text` 是镜像的一部分，所以会被再次映射。

前述说到 TTBR0 和 TTBR1 寄存器用来保存 pgd 页表的起始地址，实际上 ARM64 在寻址的时候，在 kasan 没有打开的情况下，如果虚拟地址的最高位 `bit[63]` 为 0，选择 TTBR0\_EL1 作为 pgd 起始地址，如果 `bit[63]` 为 1，选择 TTBR1\_EL1 作为 pgd 起始地址。Kasan 打开的情况下，虚拟地址的高位会被用作 tag，这时候根据虚拟地址的 `bit[55]` 来选择 TTBR0 还是 TTBR1 寄存器作为 pgd 起始地址。

再回到 `.idmap.text` 和 linux 镜像的映射，两次映射的页表起始物理地址分别在变量 `idmap_pg_dir` 和 `init_pg_dir` 中，在使能 MMU（设置 `sctlr` 寄存器，其中 `bit0` 为 1 表示 `enable mmu`）之前，会把这两个物理地址分别保存到寄存器 TTBR0\_EL1 和 TTBR1\_EL1 中。这样在寻址的时候，如果虚拟地址的最高位是 1，会使用 TTBR1 来查询页表，否则表示 `.idmap.text` 段，使用 TTBR0 来查询页表。

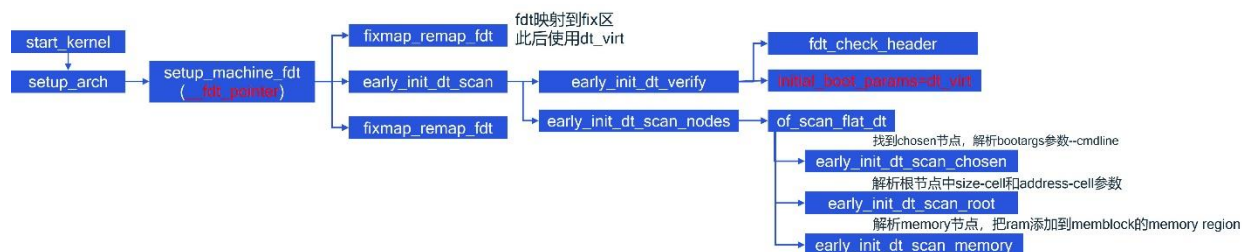
当系统运行到这时候，还是无法分配内存，原因有二。一是系统现在只建立了内核镜像的页表，除此之外的内存都无法访问。二是系统甚至还不知道内存有多大，更别提分配内存了。下面就进行这两方面的初始化工作。

### 三、Memblock

在正式的内存管理系统——伙伴系统初始化之前，内核使用 memblock 来完成早期的内存管理工作。Memblock 有两个区域（region），memory 区域和 reserved 区域。Memory 区域用来管理所有建立页表的内存，reserved 区域用来管理已经使用的内存。每个区域有若干个数组，按照起始地址高低排序，每个数组保存一块内存的起始物理地址和大小。调用 `memblock_alloc` 申请内存的时候会从 `memblock.memory.regions[]` 数组中找到可以涵盖所申请的内存的一个数组元素，然后遍历 `memblock.reserved.regions[]` 数组，如果不在 reserved 区域中表明这块内存没被使用，会把这块内存添加到 reserved 区域，然后返回申请成功的物理地址，最后把物理地址转化为虚拟地址。



### 3.1 解析内存大小



从设备树的 memory 节点中解析出 RAM 的起始地址和大小，全部添加到 memblock 的 memory 区域中。而这个 RAM 的大小并不一定是全部 DDR 的大小，而是经 hypervisor 指定的可供 AP 侧 kernel 访问的内存。由于每次遍历设备树都是从根节点开始，为了快速从设备树中找到 memory 节点，一般编写设备树源文件的时候把 memory 节点尽可能靠近根节点，而从 bootloader 中传递而来的 bootargs 参数需要更早被解析，所以一般 memory 节点紧跟在包含 bootargs 属性的 chosen 节点之后。

到目前为止，kernel 总算看到可以访问的内存的大小。但是现在仅仅内核镜像部分建立了页表，其他地址都还不能访问。

### 3.2 解析 reserved 内存



从设备树的 reserved-memory 节点中解析出 reserved 内存的起始地址和大小，添加到 memblock 的 reserved 区域，这些一般是作为特殊用途的内存，比如 modem，audio 等子系统加载启动镜像的地址，在启动阶段不允许其他人使用。这其中有 no-map 属性的节点会被从 memblock 的 memory 区域中去掉，在后面进行正式页表创建的时候不会包含这部分内存，也就意味着不允许运行时候 AP 侧的 kernel 来访问。这里在从 memblock 中移除内存的时候有个很严重的 bug，已经被

<https://github.com/torvalds/linux/commit/8a5a75e5e9e55de1cef5d83ca3589cb4899193ef> 修复

了。如果设备树中定义的 no-map 的 reserved-memory 恰好和 kernel 的代码段和数据段重合的话，系统会随机的挂掉。因为在创建页表的时候，当 kernel 的代码段和数据段建立页表之后，会把 kernel 代码段和数据段已经建立的映射区域在 memblock 管理的区域中标记为 nomap，保证之后不会被映射到线性区。但是由于其中的部分内存可能已经被从 memblock 中移除，这时候是无法从 memblock 中找到对应的区域的，也就不会被标记成 nomap，kernel 的代码段和数据段会被重新映射到线性区，代码和数据都映射到错误的地

址上了，系统很快就挂掉了。

### 3.3 正式的内核页表



在 memblock 解析完 ram 信息之后就可以着手创建正式的内核页表。正式内核页表的基地址是 swapper\_pg\_dir，然后以 page 为单位逐级完成页表的创建。如前所述，创建完成之后会把 swapper\_pg\_dir 的物理地址保存到寄存器 ttbr1\_el1，而虚拟地址会保存到全局变量 init\_mm.pgd。到这时候内核才可以随心所欲的访问内核空间的地址，但是申请/释放内存还只能通过 memblock\_alloc/memblock\_free 来完成，伙伴系统和 slab 分配器都还没建立。

### 3.4 memblock\_alloc

先从 memblock.memory.region 中找到满足所申请物理地址大小要求的内存块，然后遍历 memblock.reserved.region，如果不在 reserved 区域，说明内存没有被占用，把此次申请的区域添加到 reserved 区域，并返回申请到的虚拟地址。如果已经在 reserved 区域，继续查找下一块满足要求的地址。

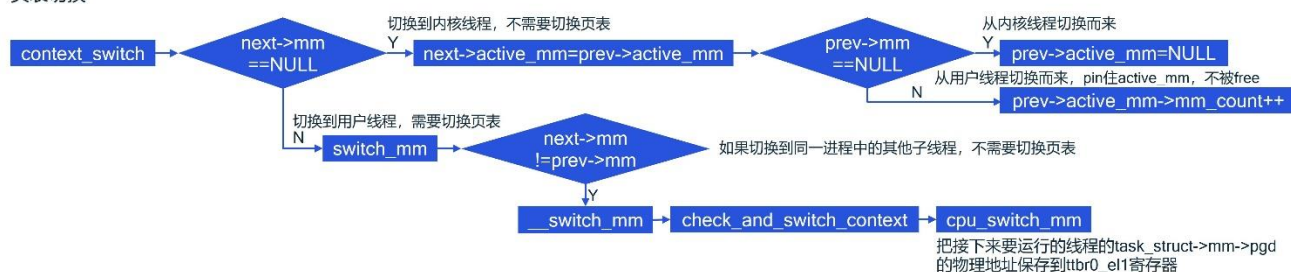
### 3.5 memblock\_free

把一块内存从 memblock.reserved.region 数组中去掉即可。

## 四、用户进程页表

### 4.1 用户进程页表的切换

页表切换



在调度器切换线程的时候会进行页表的切换，但是并不是所有情况都需要切换页表。由于内核线程共用一份页表，所以切换到内核线程的时候不需要切换页表，在初始化的时候就已经把内核页表的起始地址保存到了 ttbr1\_el1 寄存器中。当切换到用户线程的时候，同一进程的多个子线程也是共用页表的，在线程组的两个子线程之间进行线程切换的时候也是不需要切换页表的，只有在不同进程之间进行切换的时候才会有页表的切换。从



之前的介绍可以知道，用户态页表的起始地址保存在 ttbr0\_el1 寄存器中，页表的切换也就是把每个进程的页表起始物理地址写入 ttbr0\_el1 寄存器。而用户进程页表起始地址同时也会保存在 task\_struct->mm->pgd 中，页表切换的过程也就转化成把线程的 task\_struct->mm->pgd 写入 ttbr0\_el1 寄存器的过程。cpu\_switch\_mm 代码短小精悍，直接上代码：

```
static inline void cpu_switch_mm(pgd_t *pgd, struct mm_struct *mm)
{
    BUG_ON(pgd == swapper_pg_dir);
    cpu_set_reserved_ttbr0();
    cpu_do_switch_mm(virt_to_phys(pgd), mm);
}
/*
 *  cpu_do_switch_mm(pgd_phys, tsk)
 *  Set the translation table base pointer to be pgd_phys.
 *  - pgd_phys - physical address of new TTB
 *  x0 = phys of mm->pgd, x1 = mm
 */
ENTRY(cpu_do_switch_mm)
    mrs    x2, ttbr1_el1          //x2=ttbr1_el1
    mmid   x1, x1                 // get mm->context.id, x1=mm->context.id.co,8 位有效的 asid
    phys_to_ttbr x3, x0          //x3 = x0 = phys of mm->pgd
alternative_if ARM64_HAS_CNP
    cbz     x1, 1f                // skip CNP for reserved ASID
    orr     x3, x3, #TTBR_CNP_BIT
1:
alternative_else_nop_endif
#ifdef CONFIG_ARM64_SW_TTBR0_PAN
    bfi     x3, x1, #48, #16 //set the ASID field in TTBR0,x3[63:48]=x1=mm->context.id.counter
#endif
    bfi     x2, x1, #48, #16      // set the ASID, x2[63:48] = x1 = mm->context.id
    msr     ttbr1_el1, x2        // in TTBR1 (since TCR.A1 is set), ttbr1_el1 = x2, 相比切换
                                    //前的 ttbr1 寄存器更新了 ASID 域 (bit[63:48])

    isb

    msr     ttbr0_el1, x3        // now update TTBR0, ttbr0_el1=x3=phys_of_pgd

    isb

    b       post_ttbr_update_workaround // Back to C code...
ENDPROC(cpu_do_switch_mm)
```

## 4.2 用户进程页表起始地址

每个用户线程把自己的页表起始地址保存在 task\_struct->mm->pgd 中，在创建线程的

---

时候, fork 函数需要带上 CLONE\_VM 的 flags, 这样在内核的 clone 系统调用函数中 copy\_mm 的时候直接把父进程(转化为主线程)的 mm 拷贝给新线程的 mm, 这也就是多线程共享地址空间的意思。如果没有 CLONE\_VM 的 flags, 就会给予进程分配新的 pgd。也就是说页表是每个进程有一份的, 如果一个进程有多个子线程就共享一份页表。确认一下 android 使用的 bionic/libc 中创建线程和进程时候的 flags 的差异, pthread\_create 函数中的 flags 是 int flags = CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND | CLONE\_THREAD | CLONE\_SYSVSEM | CLONE\_SETTLS | CLONE\_PARENT\_SETTID | CLONE\_CHILD\_CLEARTID; fork 函数中的 flags 是 CLONE\_CHILD\_SETTID | CLONE\_CHILD\_CLEARTID | SIGCHLD : clone(nullptr, nullptr, (CLONE\_CHILD\_SETTID | CLONE\_CHILD\_CLEARTID | SIGCHLD), nullptr, nullptr, nullptr, &(self->tid))。

#### 4.3 copy\_from\_user/copy\_to\_user

当 linux 打开 CONFIG\_ARM64\_PAN 选项的时候, 在 cpu 初始化阶段 setup\_cpu\_capabilities 会把 cpu 系统寄存器 SPSR 的 PAN (privileged access never) 域置 1, 如果从 EL1 或 EL2 访问 EL0 的虚拟地址的时候, MMU 会抛出 permission fault。但是系统调用的时候经常会从用户态传递参数到内核态, 对应的虚拟地址也是用户态地址, 如果内核(工作在 EL1)直接访问的话, 就会报错。这时候就需要用到 copy\_from\_user 函数。在 copy\_from\_user 函数中访问 user space 的地址之前, 会调用 SET\_PSTATE\_PAN (0) 宏, 把系统寄存器 SPSR 的 PAN 位清零, 临时运行内核访问用户态地址。在数据拷贝完之后再调用 SET\_PSTATE\_PAN (1) 把寄存器 SPSR 的 PAN 位置 1。另外 copy\_from\_user 还有 access\_ok 的检查, 只有用户态地址才会执行 copy 操作。

##### D5.4.2 About PSTATE.PAN

When the value of PSTATE.PAN is 1, any privileged data access from EL1 or EL2 to a virtual memory address that is accessible at EL0 generates a Permission fault.

When the value of PSTATE.PAN is 0, the translation system is the same as in ARMv8.0.

##### ARM 官方文档对 PAN 的解释

既然在内核态拷贝自然是拷贝到了内核地址, 那也就引出了另外一个问题, 如果不拷贝的话会怎样。用户态的地址是跟进程相关的, 当发生进程切换的时候, 用于用户态地址翻译的寄存器 TTBR0\_EL1 会被修改掉, 当再次切换到原来的进程的时候, 这个虚拟地址翻译得到的物理地址已经不是当初的物理地址了, 地址中的值当然也就不是需要的值。