Linux-ARM64页表初始化

# 何为页表

## 1.1地址转换流程

Graphical user interface

Description automatically generated with low confidence

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。

Graphical user interface, text, application

Description automatically generated

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[\\lssrk-sh-win01\LSS\_China\Engineers\wangxuanfu\pagetable-test](file:///\\\\lssrk-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--ffffffff，DDRCS0\_1.BIN@100000000--17fffffff，DDRCS0\_2.BIN@180000000--1ffffffff，DDRCS1\_0.BIN@200000000--27fffffff，DDRCS1\_1.BIN@280000000--2ffffffff，DDRCS1\_2.BIN@300000000--37fffffff。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0xFFFFFFAB7F0626C0。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，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。

Text

Description automatically generated

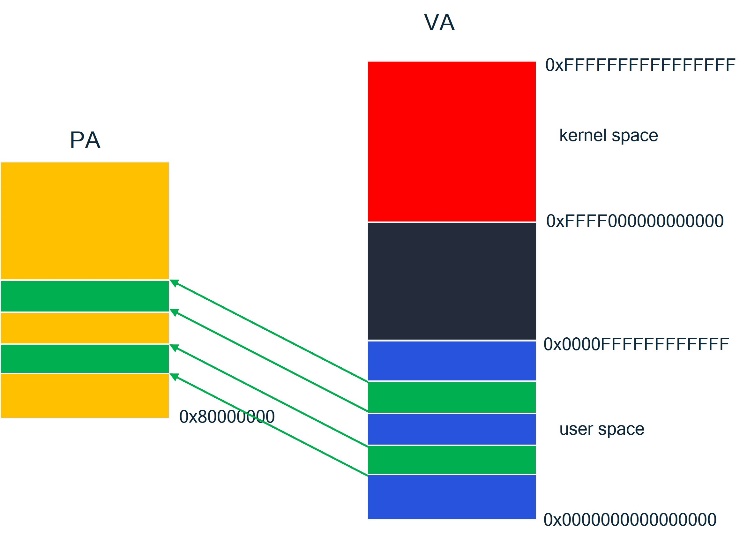
## 1.2 多级页表

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，每个用户进程可以的访问的虚拟地址空间包括用户态地址0x0000000000000000~0x0000FFFFFFFFFFFF和内核态地址0xFFFF000000000000~0xFFFFFFFFFFFFFFFF。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。

Diagram

Description automatically generated with medium confidence

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\*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候就确定的链接地址。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。

Table

Description automatically generated

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。

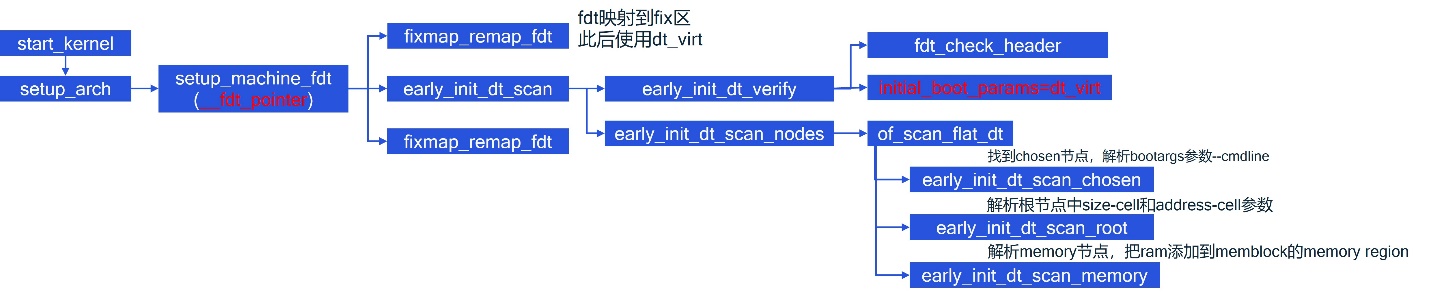
再回到.idmap.text和linux镜像的映射，两次映射的页表起始物理地址分别在变量idmap\_pg\_dir和init\_pg\_dir中，在使能MMU（设置sctlr寄存器，其中bit0为1表示enable mmu）之前，会把这两个物理地址分别保存到寄存器TTBR0\_EL1和TTBR1\_EL1中。这样在寻址的时候，如果虚拟地址的最高位是1，会使用TTBR1来查询页表，否则表示.idmap.text段，使用TTBR0来查询页表。

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。

# 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。

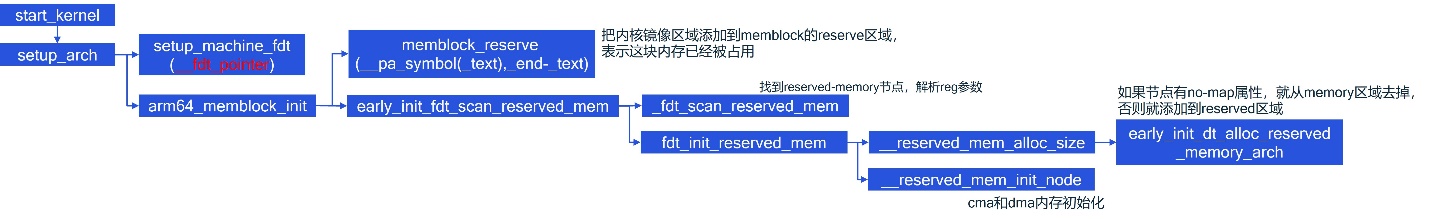
## 3.1解析内存大小



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。

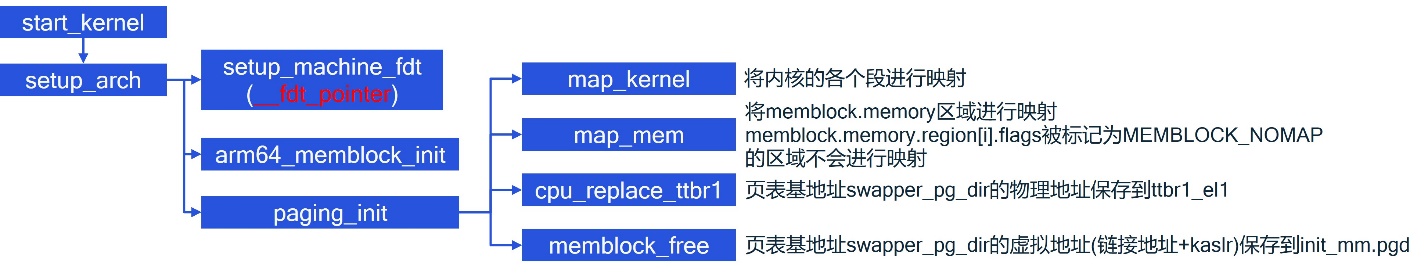
到目前为止，kernel总算看到可以访问的内存的大小。但是现在仅仅内核镜像部分建立了页表，其他地址都还不能访问。

## 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。

## 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。

## 3.4memblock\_alloc

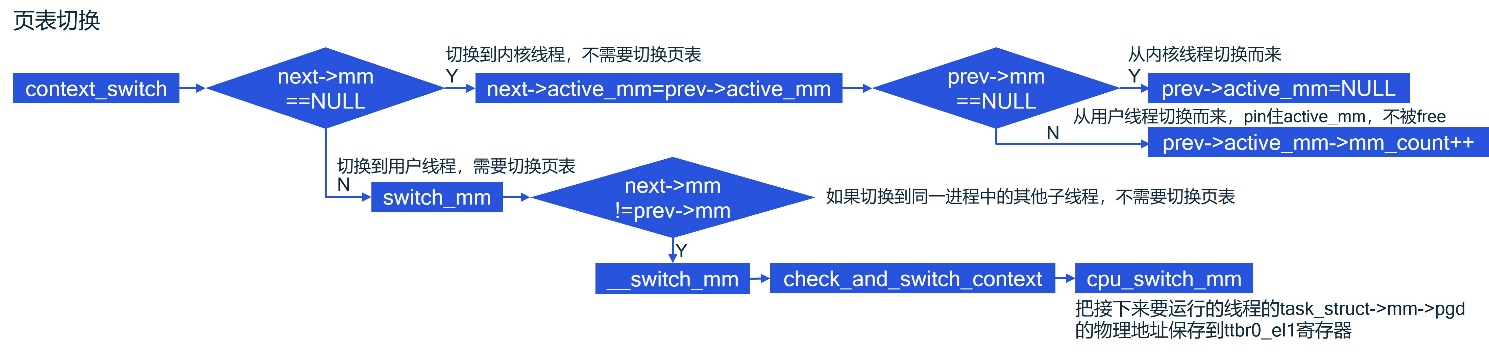
先从memblock.memory.region中找到满足所申请物理地址大小要求的内存块，然后遍历memblock.reserved.region，如果不在reserved区域，说明内存没有被占用，把此次申请的区域添加到reserved区域，并返回申请到的虚拟地址。如果已经在reserved区域，继续查找下一块满足要求的地址。

## 3.5memblock\_free

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

# 用户进程页表

## 4.1用户进程页表的切换



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，就会给子进程分配新的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))。

## 4.3copy\_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。

Text

Description automatically generated

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。