[A-12]ARMv8/ARMv9-Memory-页表描述符(Translation table descriptor)

原创 基层架构师 浩瀚架构师 2024年09月21日 18:36 辽宁

ver 0.4

前言

我们在前序的文章中，介绍了页表的基本概念，也介绍了多级页表架构。基于前文的中介绍的内容，我们应该搞清楚了如下几个事实：

(1) 从地址空间的角度出发，可以将页表理解为一个账本，记录着虚拟地址空间和物理地址空间的映射关系。

(2) 页表在地址翻译的过程中扮演着重要角色，可以说是连接着软件(由软件负责初始化和更新页表)和硬件(由MMU来遍历和权限检查页表项)的纽带。

(3) 单级页表由于自身的局限性(需要消耗大量连续物理内存保存页表)，需要引入多级页表架构完善整个内存管理的系统架构，从而推高整体资源(MMU、主存)的工作和使用效率。

对于以上的基础设计，还没有起码的感觉的话，建议大家先看一下前序的文章，再回到本文继续阅读。本文主要是在前文的基础上，进一步放大页表项，看一下每个页表项内部到底放了哪些内容，然后在这个基础上，完成一次多级页表架构下的地址翻译(页表映射)的闭环。

正文

1. 页表描述符

在介绍页表描述符之前，我们还是要把视角拉回系统架构的角度看一下我们要讨论的课题，以前的一个老领导经常教育我们说：大家不要光顾着埋头干活，也要抬头看路。我们也一样，在分析具体的课题之前，还是要具备全局的视野，对于我们一个嵌入式的工程师而言，这个全局就是我们的Soc系统架构。现在的我们已经不是普通的我们了，是经过精确的计算后，解决了单级页表缺陷，搞出了多级页表架构的我们了，哈哈哈。这里可以对前序文章中的系统框图进一步做更准确的描述了，如图1-1所示。

图1-1 典型的多级页表架构拓扑

通过上图，可以清晰地看出多级页表在主存中的布局。页表本身也要存储在内存当中，并且通过页表项中存储的物理地址实现多级页表的级联，最终在末端连接到实际程序要用到的物理内存，根据虚拟地址的分配策略，这个物理内存有两种形式：Page 和 Block。前面我们说过页表作为MMU管理内存资源的纽带，连接着软件和硬件。芯片的硬件(MMU)设计好完成、流片量产之后只能通过系统寄存器进行有限度的配置从而影响硬件的实际表现，聚焦到当前的主题，MMU使用的页表的格式就需要在量产的时候规定好，否则就乱套了。比如一款ARM的芯片上可以运行各种操作系统，例如Linux和QNX，硬件是不能改变的，那就要求Linux和QNX的软件开发者必须要按照MMU能够使用的形式去设计页表并编码实现，也就是说软件和硬件的开发者要做一个约定，这个约定一般情况下会体现在硬件芯片手册中。关于页表的相关约定，我们就称之为页表描述符(Translation table descriptor)。

1.1 基本概念

通过前文，大家应该能够理解页表描述符就是软硬件世界的一个约定，就像身份证一样，上面要体现生日、性别、ID...等等。在多级页表的架构下，页表描述符就变得稍微复杂了一些，这里需要通过一个多级页表架构的模型讲清楚，如图1-2所示。

图1-2 多级页表架构模型

通过上图可以清晰地看出，ARM多级页表架构体系下页表描述符是一个集合，具体包括D_Table、D_Block、D_Page。这些描述符规定了软件在初始化或者更新页表时候所需要做的具体工作，其实就是按照规定填充页表项。那么们再具象化一下多级架构下各级页表的页表项，如图1-3所示。

图1-3 High-Level页表描述符

通过上图，结合手册看一下页表描述符的分类。

A translation descriptor has one of the following formats:
• An invalid descriptor format.
• A Table descriptor format that points to the next-level translation table.
• A Block descriptor or Page descriptor format that defines the memory access properties.
• A reserved format.

ARM提供了4大类页表描述，他们可以通过描述符最后两位进行区别，我们会在后面章节重点讨论“11”和“01”两种类型，具体的配置如图1-4所示。

图1-4

VMSAv8-64的页表分类

通过图1-4和图1-3 将页表描述符分类之后，还需要关注页表的如下几个方面：Table size， Table alignment，Table endianness，Memory Attributes。

1.1.1 Translation table size

我们先来看一下手册中的描述：

The descriptor size in a translation table entry is one of the following:
• For the VMSAv8-64 translation system, an eight-byte, or 64-bit, object.
• For the VMSAv9-128 translation system, a 16-byte, or 128-bit, object.
If n is the number of bits resolved by a lookup level, then the number of translation table entries required at that lookup level is 2^n .
The size of a translation table in bytes is determined by multiplying the number of entries by the descriptor size.

(1) 芯片的VMSA体系不同，页表项的Size有区别，目前有两种：64-bit和128-bit，本文只聚焦VMSAv8-64体系下的描述符格式。

(2) 页表的Size是和当前级别页表项的数目相关的(TTE_Size * TTE_Count).

(3) 而每一级页表项的数目(TTE_Count)需要是2^n，这个后面讲解具体描述符格式时会讲到一部分，具体会在讲解页表映射的时候详细阐述，这里先留下一个伏笔。(这里面的n其实是虚拟地址切分后，每一段的虚拟地址的位数)

1.1.2 Translation table alignment

我们在前面的文章中反复说过，页表的存储需要连续的存储空间，这里就给出了具体的规定：

A translation table is required to be aligned to one of the following:
• For the VMSAv8-64 translation system, if the translation table has fewer than eight entries and an OA size greater than 48 bits is used, then the table is aligned to 64 bytes.
• Otherwise, the translation table is aligned to the size of that translation table.

(1) 虚拟地址翻译的过程中，MMU的Table walk模块需要快速的遍历页表，越快越好，因此在物理内存中要求连续的存储各级页表。

(2) 基于上述的原因，页表的存储自然就要求内存，这里要求最低对齐的单位为64Bytes。

(3) 巧不巧，还记得我们讲Cache的时候，现在一般ARM处理器的Cache Line Size 也是64Bytes。所以一切都是为了效率。

1.1.3 Translation table lookup endianness

这一部分对身经百战的嵌入式工程师来说绝不陌生，就不展开说了，总之就这么8个字节的一条的页表项，你不要让他们盲目的安放就行，具体的参考如下手册的描述。

When a translation table lookup reads a translation table entry, all of the following apply:
• If the VMSAv8-64 translation system is used, the read is an 8-byte single-copy atomic access.
• If the VMSAv9-128 translation system is used, the read is a 16-byte single-copy atomic access.
The endianness of a translation table lookup is determined by the appropriate SCTLR_ELx.EE bit.

1.1.4 Translation table lookup memory attributes

“这里有两根金条，你告诉我哪根是高尚的”-谢若琳斯基当年的这句灵魂拷问，同样适用于内存。物理内存本来都是一样的放在那里，是软件在运行时根据不同的上下文赋予了它们不同的属性。

For a translation table lookup in an address translation stage, the TCR_ELx and VTCR_EL2 registers determine the memory Cacheability and Shareability attributes that apply.
For a translation table lookup in an address translation stage, the required memory type is Normal memory.
For a two stage translation regime, when a translation table lookup from stage 1 occurs, all of the following apply:
• Arm recommends that the stage 2 translation of the stage 1 translation table lookup does not map to Device memory.
• Software can configure HCR_EL2.PTW to protect stage 2 table walks from mapping stage 1 translation tables to Device memory.

(1) 内存的属性包括但不限于，内存的类型、Cache属性、共享属性、访问权限、安全状态等等。

(2) 要支持虚拟化，MMU是支持两阶段的地址翻译的，页表在内存中也有两份，这就涉及到内存属性合并的问题，如图1-5所示，如在stage-1阶段为Device类型的内存，在Stage-2阶段要怎么处理，并通过MMU的检查通知CPU呢？这些都需要相当的篇幅才能够讲清楚。

(3) 页表的属性初始化和更新涉及到具体上下文(异常等级、执行状态等)，情况还是比较复杂的，我们会专门通过一篇文章来阐述，这里不展开讨论，大家只需要了解页表也要记录被分配空间的属性。

图1-5 典型的两级翻译场景

1.2 页表描述符的格式

上面章节我们已经对页表描述符的概念、分类、以及描述符所涵盖的内容做了介绍，本节我们会具象化页表描述符对其内部的细节做一下介绍。这里需要说明一点：ARM 的 Virtual Memory System Architecture(VMSA) 在AArch64的执行状态下，涵盖VMSAv8-64和 VMSAv9-128两种翻译体系的设计，本文会主要目的是依托VMSAv8-64体系的核心页表项的格式介绍ARM页表设计的相关原理和特点，对于VMSAv9-128体系，感兴趣的同学可以自行查阅手册。

>If an implementation is executing in AArch64 state, then that implementation uses either or both of the VMSAv8-64 and the VMSAv9-128 translation systems.
R RKHJV All of the following determine whether a translation stage uses the VMSAv9-128 translation system:
• For stage 1 translations in the EL1&0 translation regime, the Effective value of TCR2_EL1.D128 is 1.
• For stage 1 translations in the EL2&0 translation regime, the Effective value of TCR2_EL2.D128 is 1.

• For stage 1 translations in the EL3 translation regime, the Effective value of TCR_EL3.D128 is 1.
• For stage 2 translations, the Effective value of VTCR_EL2.D128 is 1.
Otherwise, the translation stage uses the VMSAv8-64 translation system.

1.2.1 VMSAv8-64 Table descriptor format(D_Table)

看一下多级页表架构的D_Table的格式，如图1-6所示。

图1-6 VMSAv8-64 Table descriptor format

结合图1-2、1-3，我们对D_Table归纳如下：

(1) D_Table处于多级页表架构的中间层(头是系统寄存器TTBR，尾是D_Block或者D_Page)，是用来描述下一级页表的Entry。

(2) D_Table 有效位中主要涵盖了3方面信息：该页表项的类型、下一级页表的物理地址、当前页表项的属性。

(3) OA的位宽(52bit/48)和末端Page的颗粒度(4k、16k、64k)不同，也会影响D_Block的格式。

OA(Output Address)在不考虑虚拟化的前提下可以简单理解为地址翻译之后得到的物理地址(OA永远不可能超过TCR_ELx配置的物理地址上限，否则MMU在地址翻译的过程中会报错)，这个在前序文章<内存空间>的物理地址空间的Size的章节有过详细的论述，大家可以参考。

1.2.2 VMSAv8-64 Block descriptor formats(D_Block)

看一下多级页表架构的D_Block的格式，如图1-7所示。

图1-7 VMSAv8-64 Block descriptor format

结合图1-2、1-3，我们对D_Block归纳如下：

(1) D_Block处于多级页表架构的尾部，它不会再指向下一级页表，而是真正分配给虚拟地址空间做映射的一块连续物理地址空间。

(2) D_Block 有效位中主要涵盖了3方面信息：该页表项的类型、块物理空间的起始地址、当前物理地址空间块的属性。

(3) OA的位宽(52bit/48)和末端Page的颗粒度(4k、16k、64k)不同，也会影响D_Block的格式。(这一点和D_Table一样)

(4) 关于Output address中的“n”，虽然手册中做了解释，但是还是要先简单思考一下(详细的会在下一篇文章中阐述)。

• 这里的Output address和前面的OA没有区别，给MMU输入一个IA(虚拟地址)之后，它千辛万苦的要找的其实就是这个OA。
• 这个OA寻找的过程，其实是一个不断拼接的过程：实际物理页或者块的基地址 + Offset(虚拟地址的一部分)。
• D_Block里面的OA[xx:n]就是这个基地址，而n其实是Offset和基地址的分界线，n越小那么Block空间就越小，n越大那么Block空间就越大，提高具体场景下的多样性。

(5) Block Entry 映射了连续的物理内存区域到虚拟内存中，可以提高内存访问的效率，因为它减少了页表的层级和相关的内存访问代价。

(6) 使用 Block Entry 可以在保持足够页表粒度的同时减少页表的大小和深度，这提高了内存映射的灵活性和效率，特别是对于大量使用连续物理内存区域的应用场景（如大型数据库或多媒体处理），这种方法非常有利。

1.2.3 VMSAv8-64 Page descriptor formats(D_Page)

看一下多级页表架构的D_Page的格式，如图1-7所示。

图1-8 VMSAv8-64 Page descriptor format

结合图1-2、1-3，我们对D_Page归纳如下：

(1) D_Page处于多级页表架构的尾部，它不会再指向下一级页表，而是真正分配给虚拟地址空间做映射的一页物理地址空间。

(2) D_Page 有效位中主要涵盖了3方面信息：该页表项的类型、页物理空间的起始地址、当前物理地址空间页的属性。

(3) OA的位宽(52bit/48)和末端Page的颗粒度(4k、16k、64k)不同，也会影响D_Block的格式。(这一点和D_Table一样)

(4) 关于Output address的理解D_Block一样，这里就不再赘述。

(5) 灵活性和效率都被D_Block占据了，D_Page就得守住精确性这个特征了，显然页的分配和释放代价更小，更加方便操作系统把控内存资源的使用，从而提高整个应用空间的使用体验在设计操作系统或底层驱动时，合理利用块描述符和页描述符，可以有效地控制内存访问权限，提高系统的安全性和效率。在设计操作系统或底层驱动时，合理利用块描述符和页描述符，可以有效地控制内存访问权限，提高系统的安全性和效率。

结语

本文着重介绍了页表描述符的具体格式，梳理了围绕多级页表架构体系下的相关的概念。在ARMv8 & ARMv9架构中，内存管理单元(MMU)使用描述符(Descriptors)和软件的世界做了约定，通过页表描述符定义内存区域的属性和记录虚拟地址空间和物理地址空间的映射关系。而在多级页表框架下引入了管理D_Table来管理D_Page和D_Block所在的页表区域，实现了高效率的级联管理，而块描述符(Block Descriptor)和页描述符(Page Descriptor)作为多级页表架构末端直接用来管理实际被上下文(进程、异常处理)使用的物理内存区域。

本文同样对篇幅做了压缩，对于页表映射的相关内容会独立成篇，在后续文章中做讲解，请大家保持关注。

Reference

[00] <corelink_dmc520_technical_reference_manual_100000_0202_00_en.pdf>
[01] <corelink_dmc620_dynamic_memory_controller_TRM.pdf>
[02] <IP-Controller/DDI0331G_dmc340_r4p0_trm.pdf>
[03] <80-ARM-IP-cs0001_ARMv8基础篇-400系列控制器IP.pdf>
[04] <arm_cortex_a725_core_trm_107652_0001_04_en.pdf>
[05] <arm_cortex_a720_core_trm_102530_0001_04_en.pdf>
[06] <DDI0487K_a_a-profile_architecture_reference_manual.pdf>
[07] <armv8_a_address_translation.pdf>
[08] <cortex_a55_trm_100442_0200_02_en.pdf>
[09] <learn_the_architecture_aarch64_memory_management_guide.pdf>
[10] <learn_the_architecture_armv8-a_memory_systems.pdf>
[11] <80-ARM-MM-cs0001_mmu表描述符和块描述符.pdf>
[12] <80-ARM-MM-wx0008_页表属性page-descriptor的详细介绍.pdf>
[13] <80-ARM-MM-wx0017_深度学习arm-MMU一篇就够了.pdf>
[14] <80-ARM-MM-wx0004_Arm64-MMU及页表映射.pdf>

Glossary

MMU - Memory Management Unit
TLB - translation lookaside buffer
VIPT - Virtual Index Physical Tag
VIVT - Virtual Index Virtual Tag
PIPT - Physical Index Physical Tag
VA - Virtual Address
PA - Physical Address
IPS - Intermediate Physical Space
IPA - Intermediate Physical Address
VMID - virtual machine identifier
TLB - translation lookaside buffer(地址变换高速缓存)
VTTBR_EL2 - Virtualization Translation Table Base Registers(ArmV8 寄存器)
ASID - Address Space Identifier (ASID)
DMC - Dynamic Memory Controller
DDR SDRAM - Double Data Rate Synchronous Dynamic Random Access Memory，双数据率同步动态随机存储器
IA - Input Address
OA - Output Address
VMSA - Virtual Memory System Architecture

ARMv8/v9 47 内存 14

ARMv8/v9 · 目录

上一篇
[A-11]ARMv8/ARMv9-Memory-多级页表架构

下一篇
[A-13]ARMv8/ARMv9-Memory-虚拟地址翻译(页表映射过程)

个人观点，仅供参考