

实验 2：内存管理

本实验主要目的在于让同学们熟悉内核启动过程中对内存的初始化和内核启动后对物理内存和页表的管理，包括三个部分：内核启动页表、物理内存管理、页表管理。

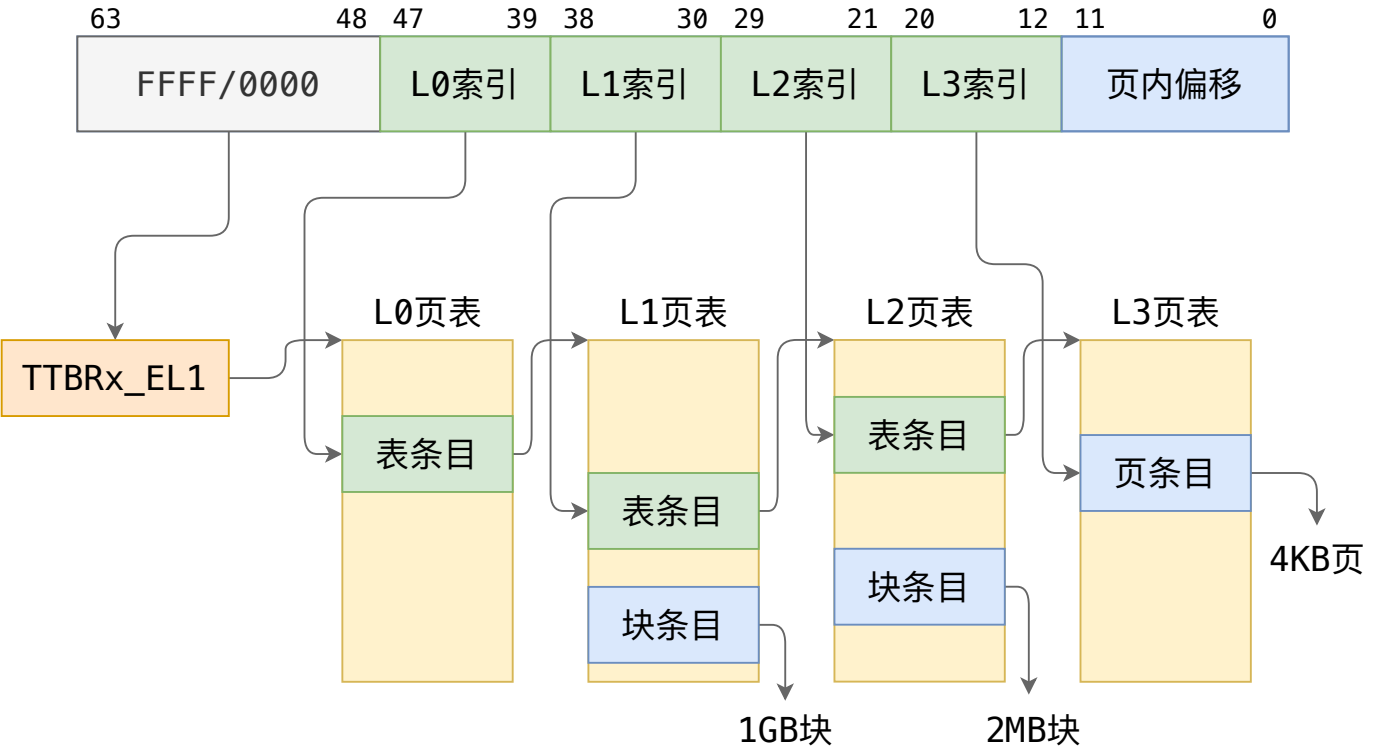
第一部分：内核启动页表

AArch64 地址翻译

在配置内核启动页表前，我们首先回顾实验涉及到的体系结构知识。这部分内容课堂上已经学习过，如果你已熟练掌握则可以直接跳过这里的介绍（但不要跳过思考题）。

在 AArch64 架构的 EL1 异常级别存在两个页表基址寄存器：`ttbr0_el1`¹ 和 `ttbr1_el1`²，分别用作虚拟地址空间低地址和高地址的翻译。那么什么地址范围称为“低地址”，什么地址范围称为“高地址”呢？这由 `ttcr_el1` 翻译控制寄存器³ 控制，该寄存器提供了丰富的可配置性，可决定 64 位虚拟地址的高多少位为 0 时，使用 `ttbr0_el1` 指向的页表进行翻译，高多少位为 1 时，使用 `ttbr1_el1` 指向的页表进行翻译⁴。一般情况下，我们会将 `ttcr_el1` 配置为高低地址各有 48 位的地址范围，即，`0x0000_0000_0000_0000` ~ `0x0000_ffff_ffff_ffff` 为低地址，`0xffff_0000_0000_0000` ~ `0xffff_ffff_ffff_ffff` 为高地址。

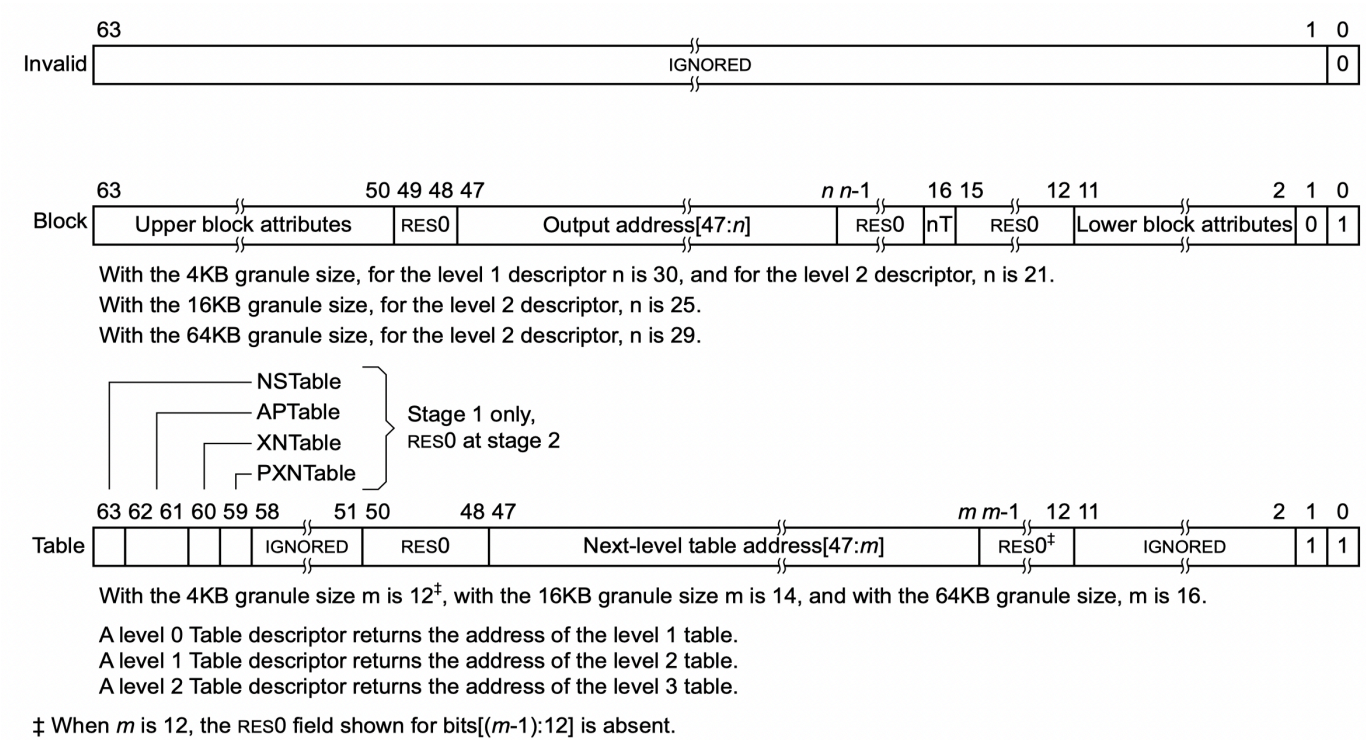
了解了如何决定使用 `ttbr0_el1` 还是 `ttbr1_el1` 指向的页表，再来看地址翻译过程如何进行。通常我们会将系统配置为使用 4KB 翻译粒度、4 级页表（L0 到 L3），同时在 L1 和 L2 页表中分别允许映射 2MB 和 1GB 大页（或称为块）⁵，因此地址翻译的过程如下图所示：



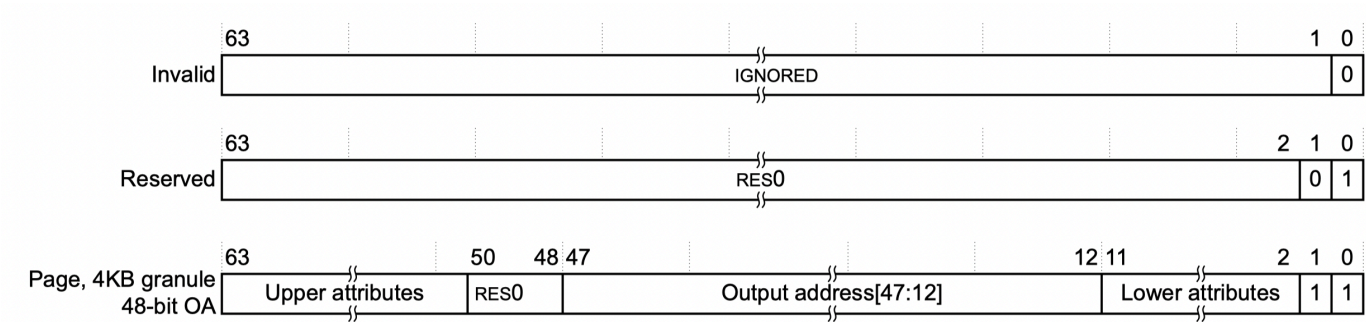
其中，当映射为 1GB 块或 2MB 块时，图中 L2、L3 索引或 L3 索引的位置和低 12 位共同组成块内偏移。

每一级的每一个页表占用一个 4KB 物理页，称为页表页（Page Table Page），其中有 512 个条目，每个条目占 64 位。AArch64 中，页表条目称为描述符（descriptor）⁶，最低位（bit[0]）为 1 时，描述符有效，否则无效。有效描述符有两种类型，一种指向下一级页表（称为表描述符），另一种指向物理块（大页）或物理页（称为块描述符或页描述符）。在上面所说的地址翻译配置下，描述符结构如下（“Output address”在这里即物理地址，一些地方称为物理页帧号（Page Frame Number，PFN））：

L0、L1、L2 页表描述符



L3 页表描述符



思考题 1：请思考多级页表相比单级页表带来的优势和劣势（如果有的话），并计算在 AArch64 页表中分别以 4KB 粒度和 2MB 粒度映射 0~4GB 地址范围所需的物理内存大小（或页表页数量）。

页表描述符中除了包含下一级页表或物理页/块的地址，还包含对内存访问进行控制的属性（attribute）。这里涉及到太多细节，本文档限于篇幅只介绍最常用的几个页/块描述符中的属性字段：

字段	位	描述
UXN	bit[54]	置为 1 表示非特权态无法执行（Unprivileged eXecute-Never）
PXN	bit[53]	置为 1 表示特权态无法执行（Privileged eXecute-Never）
nG	bit[11]	置为 1 表示该描述符在 TLB 中的缓存只对当前 ASID 有效
AF	bit[10]	置为 1 表示该页/块在上一次 AF 置 0 后被访问过
SH	bits[9:8]	表示可共享属性 ⁷
AP	bits[7:6]	表示读写等数据访问权限 ⁸
AttrIdx	bits[4:2]	表示内存属性索引，间接指向 mair_ell 寄存器中配置的属性 ⁹ ，用于控制将物理页映射为正常内存（normal memory）或设备内存（device memory），以及控制 cache 策略等

配置内核启动页表

有了关于页表配置的前置知识，我们终于可以开始配置内核的启动页表了（回顾：上一个实验的结尾我们在未配置启动页表时直接启用 MMU，内核崩溃，进入 translation fault 的死循环）。

操作系统内核通常需要“运行在”虚拟内存的高地址（如前所述，0xffff_0000_0000_0000 之后的虚拟地址），这里“运行在”的意思是，通过对内核页表的配置，将虚拟内存高地址映射到内核所在的物理内存，在执行内核代码时，PC 寄存器的值是高地址，对全局变量、栈等的访问都使用高地址。在内核运行时，除了需要访问内核代码和数据等，往往还需要能够对任意物理内存和外设内存（MMIO）进行读写，这种读写同样通过高地址进行。

因此，在内核启动时，首先需要对内核自身、其余可用物理内存和外设内存进行虚拟地址映射，最简单的映射方式是一对一的映射，即将虚拟地址 0xffff_0000_0000_0000 + addr 映射到 addr。需要注意的是，在 ChCore 实验中我们使用了 0xffff_ff00_0000_0000 作为内核虚拟地址的开始（注意开头 f 数量的区别），不过这不影响我们对知识点的理解。

在树莓派 3B+ 机器上，物理地址空间分布如下¹⁰：

物理地址范围	对应设备
0x00000000 ~ 0x3f000000	物理内存（SDRAM）
0x3f000000 ~ 0x40000000	共享外设内存
0x40000000 ~ 0xffffffff	本地（每个 CPU 核独立）外设内存

现在将目光转移到 kernel/arch/aarch64/boot/raspi3/init/mmu.c 文件，我们需要在 init_boot_pt 为内核配置从 0x00000000 到 0x80000000（0x40000000 后的 1G，ChCore 只需使用这部分地址中的本地外设）的映射，其中 0x00000000 到 0x3f000000 映射为 normal memory，0x3f000000 到 0x80000000 映射为 device memory，其中 0x00000000 到 0x40000000 以 2MB 块粒度映射，0x40000000 到 0x80000000 以 1GB 块粒度映射。

练习题 2: 请在 `init_boot_pt` 函数的 LAB 2 TODO 1 处配置内核高地址页表 (`boot_ttbr1_10`、`boot_ttbr1_11` 和 `boot_ttbr1_12`)，以 2MB 粒度映射。

思考题 3: 请思考在 `init_boot_pt` 函数中为什么还要为低地址配置页表，并尝试验证自己的解释。

完成 `init_boot_pt` 函数后，ChCore 内核便可以在 `el1_mmu_activate` 启用 MMU 后继续执行，并通过 `start_kernel` 跳转到高地址，进而跳转到内核的 `main` 函数（位于 `kernel/arch/aarch64/main.c`）。

第二部分：物理内存管理

内核启动过程结束后（我们一般认为跳到内核 `main` 函数即完成了启动（boot）过程），需要对内存管理模块进行初始化（`mm_init` 函数），首先需要把物理内存管起来，从而使内核代码可以动态地分配内存。

ChCore 使用伙伴系统（buddy system）¹¹ 对物理页进行管理，在 `mm_init` 中对伙伴系统进行了初始化。为了使物理内存的管理可扩展，ChCore 在 `mm_init` 的开头首先调用平台特定的 `parse_mem_map` 函数（实验中目前只有树莓派 3B+ 平台的实现），该函数解析并返回了可用物理内存区域，然后再对各可用内存区域初始化伙伴系统。

伙伴系统中的每个内存块都有一个阶（order），阶是从 0 到指定上限 `BUDDY_MAX_ORDER` 的整数。一个 n 阶的块的大小为 $2^n \times \text{PAGE_SIZE}$ ，因此这些内存块的大小正好是比它小一个阶的内存块的大小的两倍。内存块的大小是 2 次幂对齐，使地址计算变得简单。当一个较大的内存块被分割时，它被分成两个较小的内存块，这两个小内存块相互成为唯一的伙伴。一个分割的内存块也只能与它唯一的伙伴块进行合并（合并成他们分割前的块）。

ChCore 中每个由伙伴系统管理的内存区域称为一个 `struct phys_mem_pool`，该结构体中包含物理页元信息的起始地址（`page_metadata`）、伙伴系统各阶内存块的空闲链表（`free_lists`）等。

练习题 4: 完成 `kernel/mm/buddy.c` 中的 `split_page`、`buddy_get_pages`、`merge_page` 和 `buddy_free_pages` 函数中的 LAB 2 TODO 2 部分，其中 `buddy_get_pages` 用于分配指定阶大小的连续物理页，`buddy_free_pages` 用于释放已分配的连续物理页。

提示：

- 可以使用 `kernel/include/common/list.h` 中提供的链表相关函数如 `init_list_head`、`list_add`、`list_del`、`list_entry` 来对伙伴系统中的空闲链表进行操作
- 可使用 `get_buddy_chunk` 函数获得某个物理内存块的伙伴块
- 更多提示见代码注释

有了基于伙伴系统的物理内存管理，我们便可以在内核中进行动态内存分配，也就是可以实现 `kmalloc` 了。

ChCore 的 `kmalloc` 对于较小的分配需求采用 SLAB 分配器¹²，对于较大的分配需求则直接从伙伴系统中分配物理页。动态分配出的物理页被转换为内核虚拟地址（Kernel Virtual Address, KVA），也就是在实验第一部分中我们映射的 `0xffff_ff00_0000_0000` 之后的地址。现在你可以测试内核中是否已经能够正常使用 `kmalloc` 和 `kfree` 了！

第三部分：页表管理

在第一部分我们已经详细介绍了 AArch64 的地址翻译过程，并介绍了各级页表和不同类型的页表描述符，最后在内核启动阶段配置了一个粗粒度的启动页表。现在，在迎接下一个实验中将要引入的第一个用户态进程之前，我们需要为其准备一个更细粒度的页表实现，提供映射、取消映射、查询等功能。

练习题 5: 完成 `kernel/arch/aarch64/mm/page_table.c` 中的

`query_in_pgtbl`、`map_range_in_pgtbl`、`unmap_range_in_pgtbl` 函数中的 LAB 2 TODO 3 部分, 分别实现页表查询、映射、取消映射操作。

提示:

- 暂时不用考虑 TLB 刷新, 目前实现的只是页表作为内存上的数据结构的管理操作, 还没有真的设置到页表基址寄存器 (TTBR)
- 实现中可以使用 `get_next_ptp`、`set_pte_flags`、`GET_LX_INDEX` 等已经给定的函数和宏
- 更多提示见代码注释

练习题 6: 在上一个练习的函数中支持大页 (2M、1G 页) 映射, 可假设取消映射的地址范围一定是某次映射的完整地址范围, 即不会先映射一大块, 再取消映射其中一小块。

思考题 7: 阅读 Arm Architecture Reference Manual, 思考要在操作系统中支持写时拷贝 (Copy-on-Write, CoW) ¹³ 需要配置页表描述符的哪个/哪些字段, 并在发生缺页异常 (实际上是 permission fault) 时如何处理。

思考题 8: 为了简单起见, 在 ChCore 实验中没有为内核页表使用细粒度的映射, 而是直接沿用了启动时的粗粒度页表, 请思考这样做有什么问题。

挑战题 9: 使用前面实现的 `page_table.c` 中的函数, 在内核启动后重新配置内核页表, 进行细粒度的映射。

-
1. Arm Architecture Reference Manual, D13.2.144 [↗](#)
 2. Arm Architecture Reference Manual, D13.2.147 [↗](#)
 3. Arm Architecture Reference Manual, D13.2.131 [↗](#)
 4. Arm Architecture Reference Manual, D5.2 Figure D5-13 [↗](#)
 5. 现代操作系统: 原理与实现, 4.3.5 大页 [↗](#)
 6. Arm Architecture Reference Manual, D5.3 [↗](#)
 7. Arm Architecture Reference Manual, D5.5 [↗](#)
 8. Arm Architecture Reference Manual, D5.4 [↗](#)
 9. Arm Architecture Reference Manual, D13.2.97 [↗](#)
 10. [bcm2836-peripherals.pdf](#) & Raspberry Pi Hardware - Peripheral Addresses [↗](#)
 11. 现代操作系统: 原理与实现, 4.4.2 伙伴系统 [↗](#)
 12. 现代操作系统: 原理与实现, 4.4.3 SLAB 分配器 [↗](#)
 13. 现代操作系统: 原理与实现, 4.3.2 写时拷贝 [↗](#)