

(LK) 页表

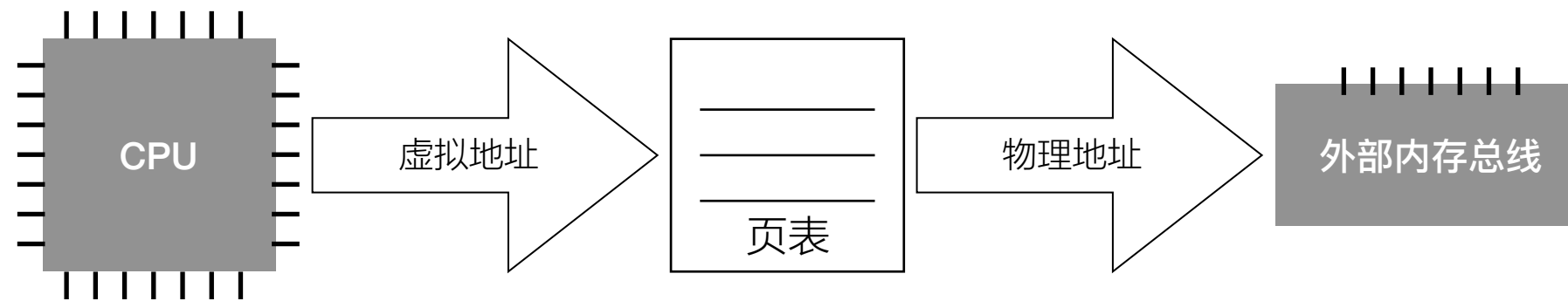
手稿

封面

本节简要介绍 Linux 内核中页表的基本概念和工作原理。

概述

页表将 CPU 所见的虚拟地址映射成外部内存总线所见的物理地址。



历史

分页式虚拟内存在 1962 年与虚拟内存概念一起被发明，诞生于 Ferranti Atlas 计算机，这是第一台使用分页虚拟内存的计算机。

随着时间的推移，该功能被迁移到了新的计算机，并成为所有类 Unix 系统事实上的标准功能。

1985 年，该功能被引入 Intel 80386，这也是 Linux 1.0 开发所用的 CPU。

物理页帧

对应于虚拟地址的物理地址常常被底层物理页帧引用。页帧号（page frame number 或者 pfn）是页的物理地址（外部内存总线所见），大小按 PAGE_SIZE 划分。

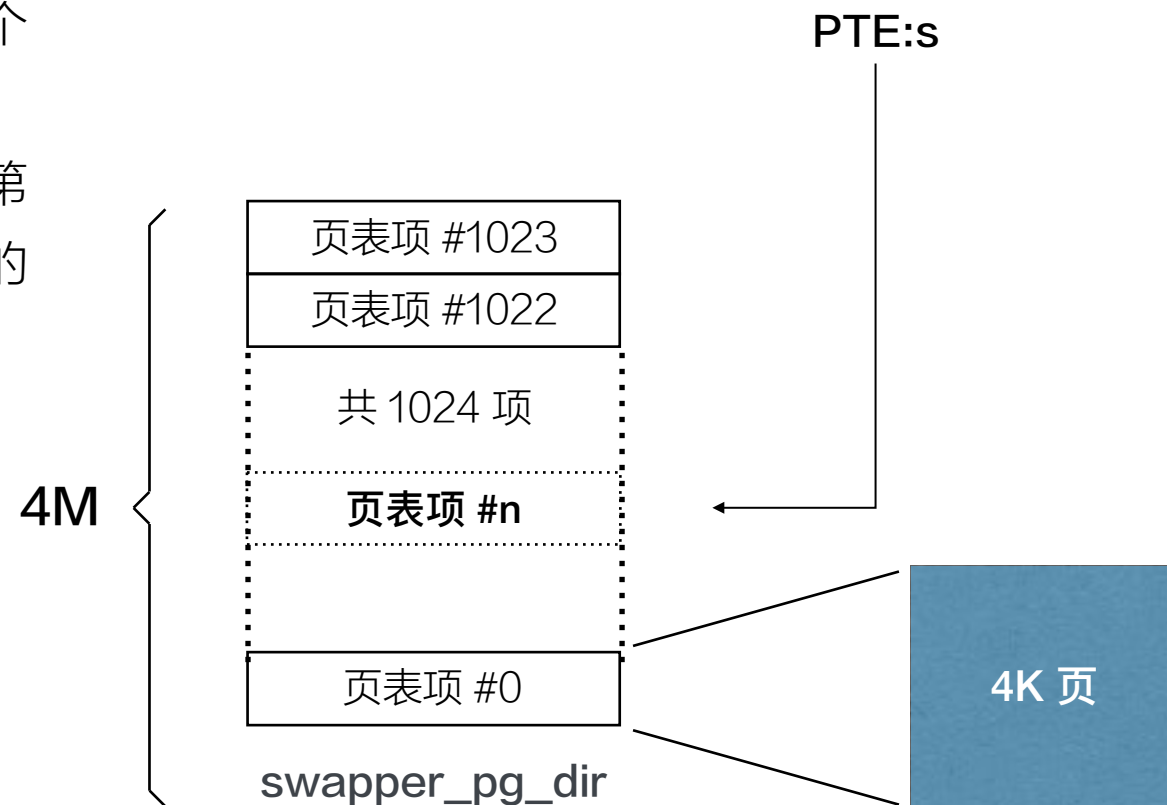
物理内存地址 0 对应 pfn 0，pfn 的最大值则是 CPU 的外部地址总线能够访问的最后一页。

4K 页	0xffffffff000 pfn = 0xffffffff	16K 页	0xfffffc000 pfn = 0x3ffff
...		...	
4K 页	0x00002000 pfn = 2	16K 页	0x00008000 pfn = 2
4K 页	0x00001000 pfn = 1	16K 页	0x00004000 pfn = 1
4K 页	0x00000000 pfn = 0	16K 页	0x00000000 pfn = 0

如你所见，在 32 位平台，当页是 4KB 时，页基址使用地址的 12-31 位，这也是为什么这种情况下 PAGE_SHIFT 定义成 12 并且 PAGE_SIZE 按照 $(1 \ll \text{PAGE_SHIFT})$ 来定义的原因。

Linux 创建之初

在 Linux 创建之初，使用的是包含 1024 个页表项称为 `swapper_pg_dir` 的单一页表，覆盖 4M 的地址范围，恰好是 Torvald 的第一台计算机的物理内存大小。这个单表中的条目被称为 PTE:s -- 页表项。

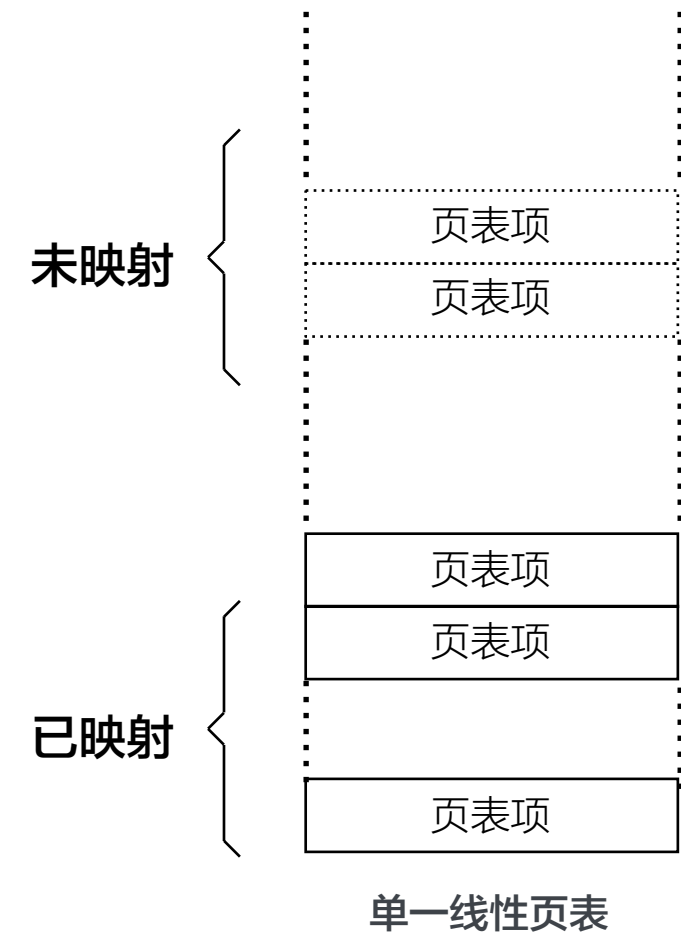


线性页表和层级页表

内存容量不断增加，页表层级也越来越深。这反映了如下事实：页表硬件变成了层级形式，如此可以节省页表内存并加快映射。

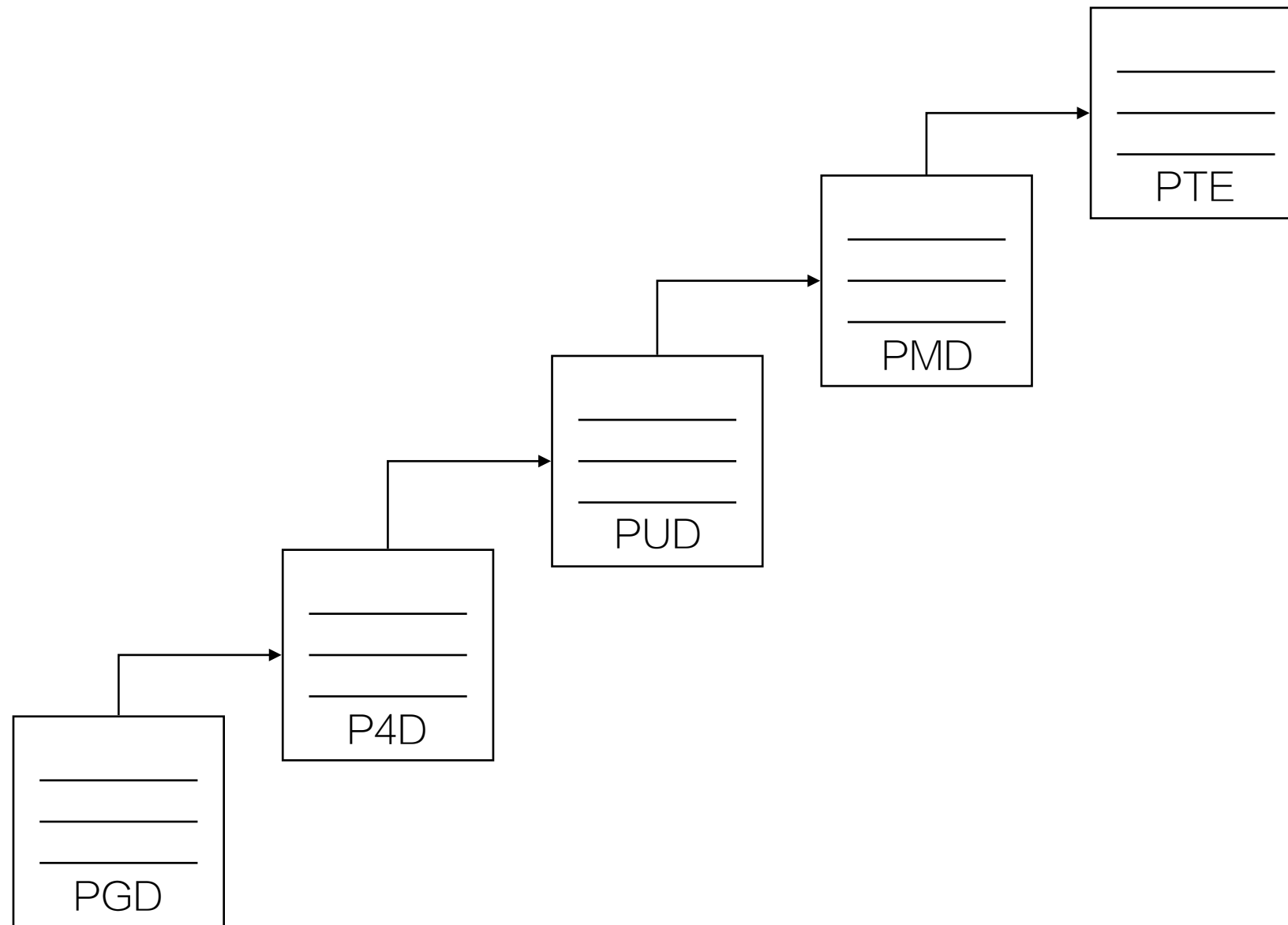
你当然可以想象有一个巨大的线性页表，包含非常多的页表项，它将整个内存分为单个页。这样的页表会非常稀疏，因为很大一部分虚拟内存通常没用。通过使用层级页表，虚拟地址间的空洞不会浪费珍贵的页表内存，因为它只要在页表的更高层级中将这一大块标记为未映射就可以了。

另外，在现代 CPU 上，更高级的页表项可以直接指向物理内存范围，这让几 M 甚至几 G 的连续范围可以在高层页表项中映射，短路虚拟内存到物理内存的映射：当你找到这样一个大范围映射后，没必要再去下一级遍历查找。



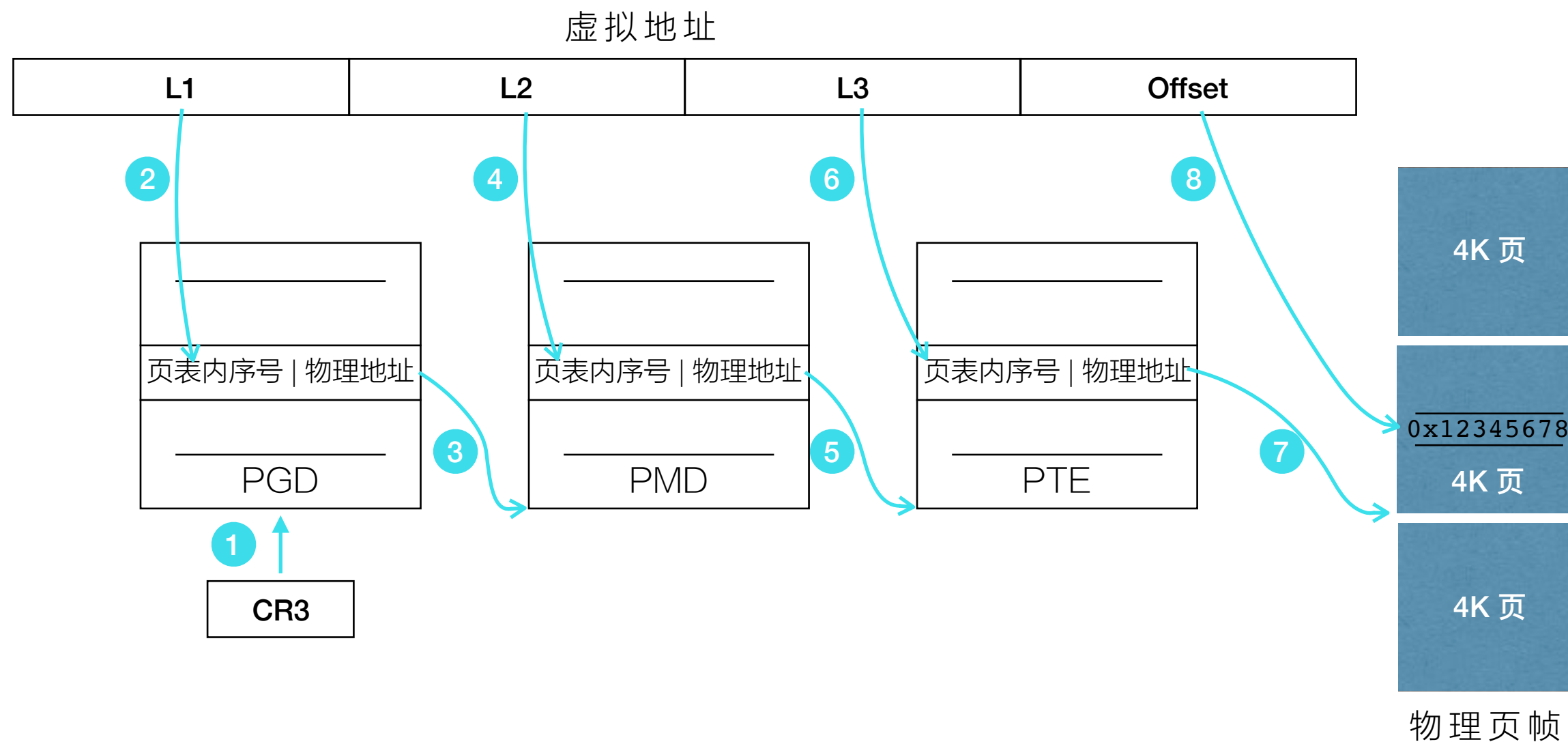
五级页表

Linux 以层级方式定义页表，目前支持 5 层。每种支持的架构对应的代码将根据硬件限制实现自己的映射。

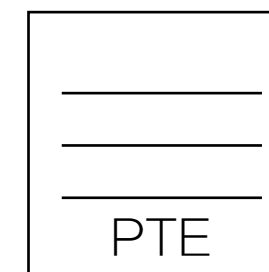


3 级页表寻址示意图

下面是 x86 架构下典型的 3 级页表寻址示意图。



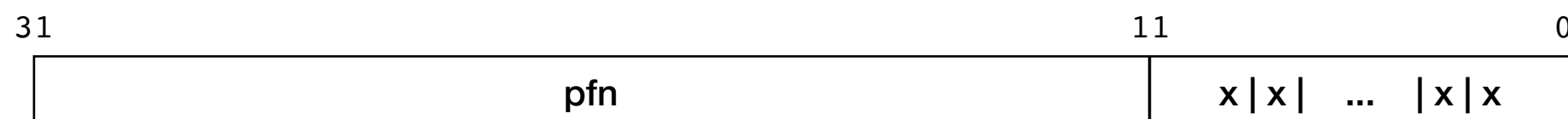
pte



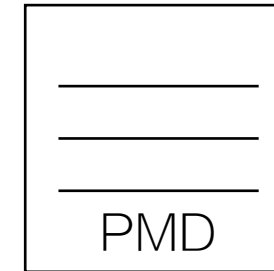
pte、pte_t、pteval_t 描述的是前面提到的页表项（**P**age **T**able **E**ntry）。pte 是 PTRS_PER_PTE 个 pteval_t 类型元素的数组，每个元素将单个虚拟内存页映射成单个物理内存页。pteval_t 的大小和内容依架构而定。

pteval_t 的一个典型例子是 32 位或 64 位值，高位是 pfn（页帧号），低位是架构专属位，比如内存保护。

pte 名称中的 e（entry）有点令人困惑，因为在 Linux 1.0 时它确实是单一顶层页表中的单个页表项，但是当引入 2 层页表开始它就演变成了映射元素的数组，因此 pte 是页表的最底层，而不是页表项。

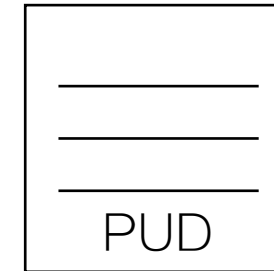


pmd



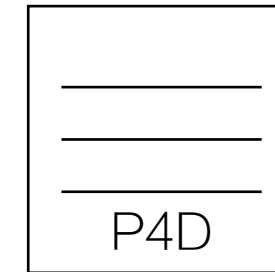
pmd、pmd_t、pmdval_t 描述的是页中间目录（**P**age **M**iddle **D**irectory），即紧靠着 pte 上面的一层，有着 PTRS_PER_PMD 条对 pte:s 的引用。

pud



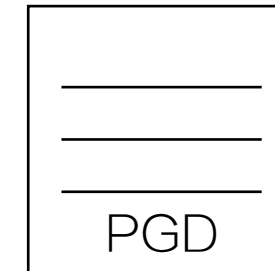
pud、pud_t、pudval_t 描述的是页上层目录（**P**age **U**pper **D**irectory），它在其他层之后被引入，用于处理 4 层页表。这一层可能不会被用到，或者被折叠起来。

p4d



p4d、p4d_t、p4dval_t 描述的是第四层页目录（**P**age Level **4 D**irectory），它在 pud 之后被引入，目的是为了处理 5 层页表。很明显，我们不能再用临时名称了，需要用数字来替换 pgd、pmd、pud 等，以此来表明目录等级。只有真正拥有 5 层页表的系统才会用到这一层，否则它将被折叠。

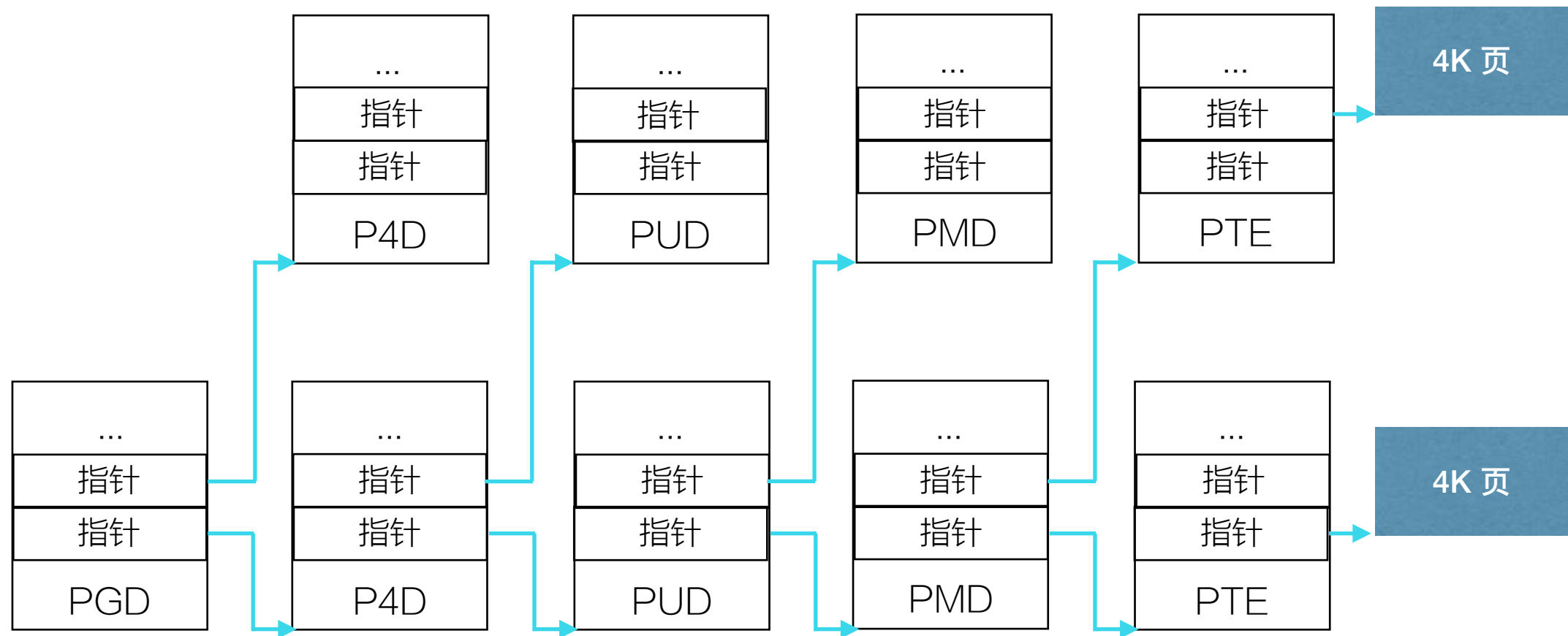
pgd



pgd、pgd_t、pgdval_t 描述的是全局页目录（**P**age **G**lobal **D**irectory），处理针对内核内存 PGD 的 Linux 内核主页表仍然位于 swapper_pg_dir，但系统中每个用户空间的进程同样拥有自己的内存上下文和它们自己的 pgd，你可以在 struct mm_struct 中找到它，而 struct mm_struct 则被每个 struct task_struct 所引用。因此，任务有自己 struct mm_struct 形式的内存上下文，它用一个 struct pgt_t *pgd 指针指向对应的页表全局目录。

页表中的指针

再说一次：页表层级的每一级都是指针数组，因此 pgd 包含 PTRS_PER_PGD 个指向下一级的指针，p4d 包含 PTRS_PER_P4D 个指向 pud 的指针，以此类推。每一层中指针的数量依架构而定。



页表折叠

如果架构没有使用所有的页表层级，那么它们可以折叠，意味着被跳过，在页表上执行的所有操作将在编译时扩展为：当访问下一层时，直接跳过这一层。例如：

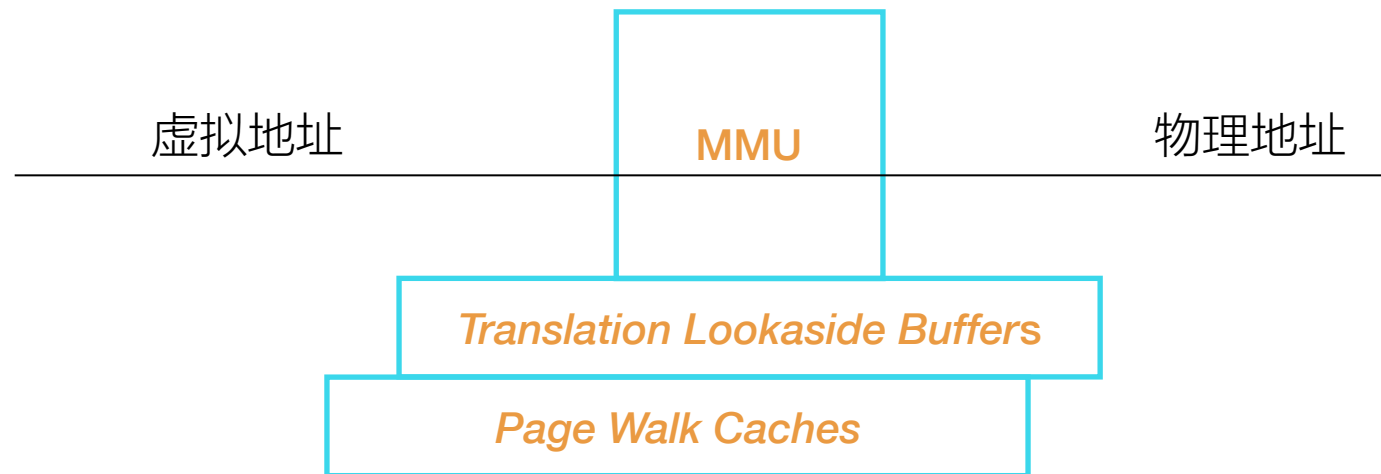
```
#define __PAGETABLE_P4D_FOLDED
```

架构无关的页表处理代码，比如虚拟内存管理器，需要这样写，以便能够遍历所有 5 层页表。与架构相关的代码也应该优先使用这种形式，增强鲁棒性应对将来的变动。

MMU

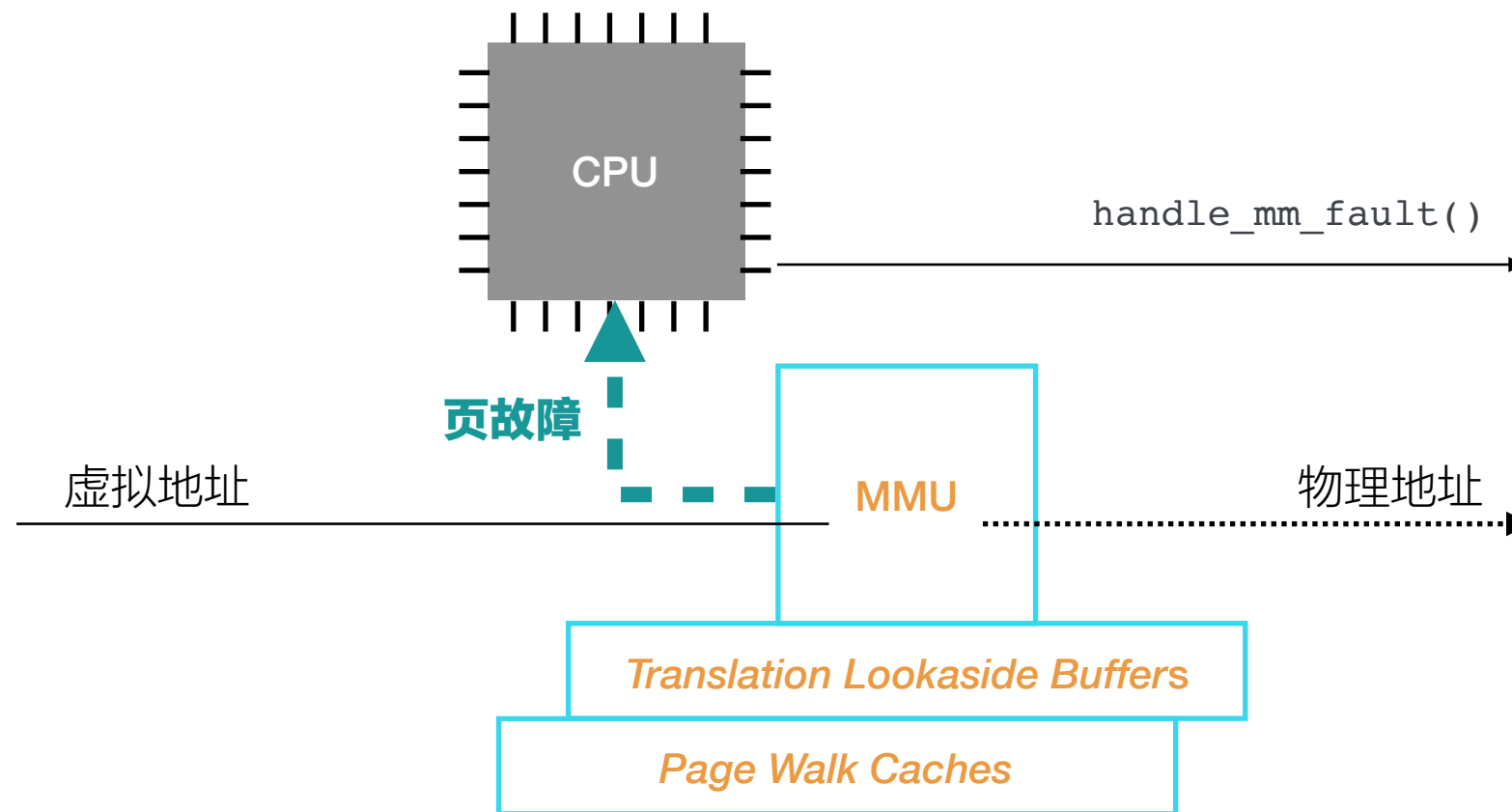
内存管理单元（**M**emory **M**anagement **U**nit, MMU）是处理虚拟地址到物理地址翻译的硬件，它可能会使用硬件中容量相对较小的缓存来加速翻译过程，它们称为快表（**T**ranslation **L**ookaside **B**uffers, TLB）和页遍历缓存（**P**age **W**alk **C**aches）。

当 CPU 访问内存位置时，它向 MMU 提供一个虚拟地址，MMU 会首先在 TLB 或者页遍历缓存（如果架构支持的话）中查找已有翻译。如果没有找到匹配项，MMU 再通过页表遍历确定物理地址并创建映射。



页故障

当 MMU 无法找到对应的翻译时，比如因为没有访问权限或页交换，它会触发页故障。页故障分为普通和可预见两种，它们由“惰性分配”和“写时拷贝”这样的进程管理技术触发，目的是为了改进内存效率、减少延迟以及最小化空间占用。如果内核自身无法为数据分配空间，它会调用 OOM（killer）杀死低优先级进程。如果发生在用户空间，内核向当前线程发送 SIGSEGV 信号。页故障的最后，所有架构都会调用 `handle_mm_fault()`，进而调用真正的处理函数 `__handle_mm_fault()`，它通过调用几个其他函数找到上层页表的对应条目的偏移量，然后分配页表并更新 MMU 和它的缓存。





参考资料

参考资料：

https://docs.kernel.org/mm/page_tables.html