DPDK内存篇（一）：基本概念

DPDK社区 DPDK与SPDK开源社区 今天

作者简介

Anatoly Burakov：英特尔软件工程师，目前在维护DPDK中的VFIO和内存子系统

引言

内存管理是数据面开发套件（DPDK）的一个核心部分，以此为基础，DPDK的其他部分和用户应用得以发挥其最佳性能。本系列文章将详细介绍DPDK提供的各种内存管理的功能。

但在此之前，有必要先谈一谈为何DPDK中内存管理要以现有的方式运作，它背后又有怎样的原理，再进一步探讨DPDK具体能够提供哪些与内存相关的功能。本文将先介绍DPDK内存的基本原理，并解释它们是如何帮助DPDK实现高性能的。

请注意，虽然DPDK支持FreeBSD*，而且也会有正在运行的Windows*端口，但目前大多数与内存相关的功能仅适用于Linux*。

标准大页

现代CPU架构中，内存管理并不以单个字节进行，而是以页为单位，即虚拟和物理连续的内存块。这些内存块通常(但不是必须) 存储在RAM中。在英特尔®64和IA-32架构上，标准系统的页面大小为4KB。

基于安全性和通用性的考虑，软件的应用程序访问的内存位置使用的是操作系统分配的虚拟地址。运行代码时，该虚拟地址需要被转换为硬件使用的物理地址。这种转换是操作系统通过页表转换来完成的，页表在分页粒度级别上（即4KB一个粒度）将虚拟地址映射到物理地址。为了提高性能，最近一次使用的若干页面地址被保存在一个称为转换检测缓冲区(TLB)的高速缓存中。每一分页都占有TLB的一个条目。如果用户的代码访问(或最近访问过)16 KB的内存，即4页，这些页面很有可能会在TLB缓存中。

如果其中一个页面不在TLB缓存中，尝试访问该页面中包含的地址将导致TLB查询失败；也就是说，操作系统写入TLB的页地址必须是在它的全局页表中进行查询操作获取的。因此，TLB查询失败的代价也相对较高(某些情况下代价会非常高)，所以最好将当前活动的所有页面都置于TLB中以尽可能减少TLB查询失败。

然而，TLB的大小有限，而且实际上非常小，和DPDK通常处理的数据量(有时高达几十GB)比起来，在任一给定的时刻，4KB 标准页面大小的TLB所覆盖的内存量（几MB）微不足道。这意味着，如果DPDK采用常规内存，使用DPDK的应用会因为TLB频繁的查询失败在性能上大打折扣。

为解决这个问题，DPDK依赖于标准大页。从名字中很容易猜到，标准大页类似于普通的页面，只是会更大。有多大呢？在英特尔®64和1A-32架构上，目前可用的两种大页大小为2MB和1GB。也就是说，单个页面可以覆盖2 MB或1 GB大小的整个物理和虚拟连续的存储区域。

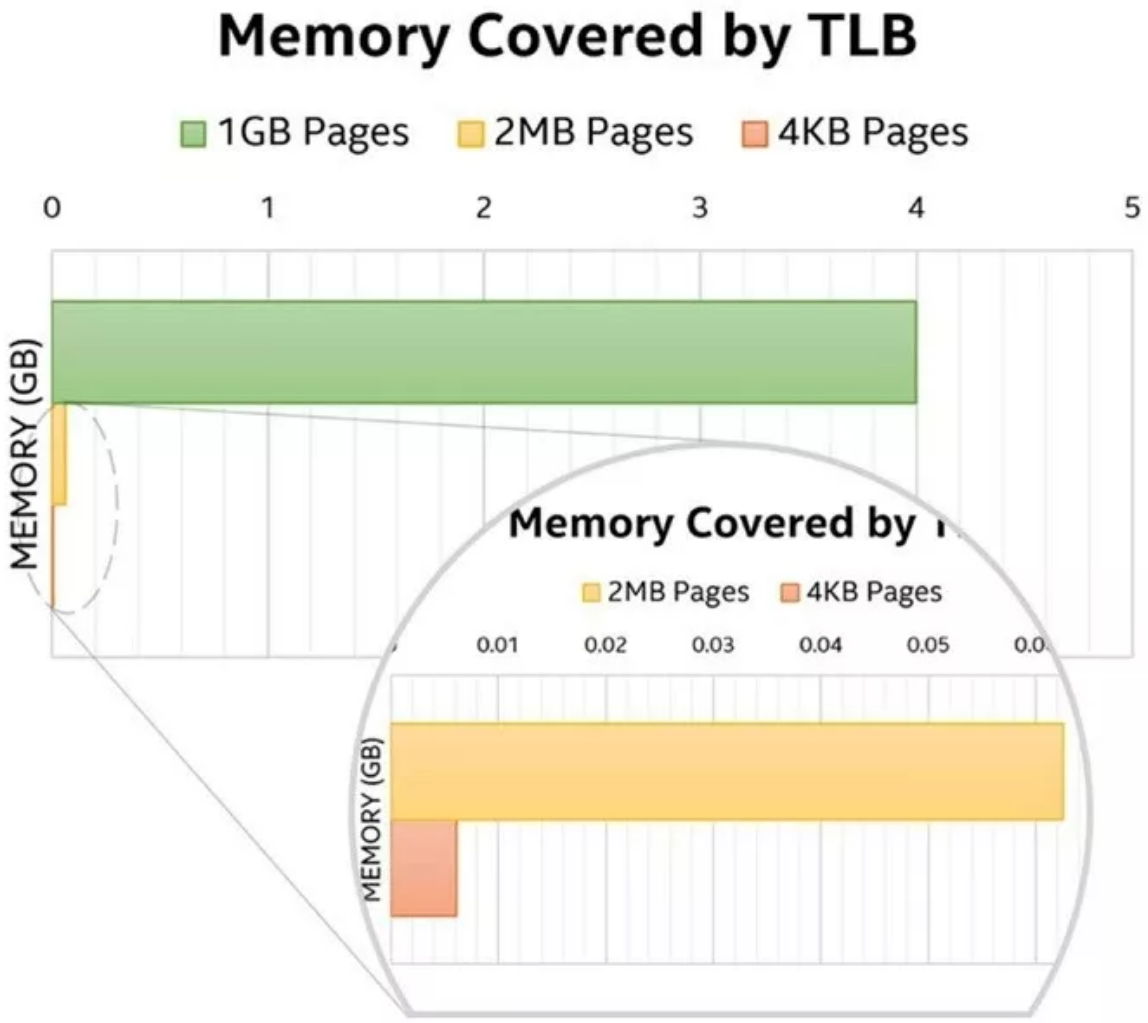


图1. TLB内存覆盖量比较。

这两种页面大小DPDK都可以支持。有了这样的页面大小，就可以更容易覆盖大内存区域，也同时避免(同样多的)TLB查询失败。反过来，在处理大内存区域时，更少的TLB查询失败也会使性能得到提升，DPDK的用例通常如此。

将内存固定到NUMA节点

当分配常规内存时，理论上，它可以被分配到RAM中的任何位置。这在单CPU系统上没有什么问题，但是许多DPDK用户是在支持非统一内存访问 (NUMA) 的多CPU系统上运行应用的。对于NUMA来说，所有内存都是不同的：某一个CPU对一些内存的访问（如不在该CPU所属NUMA NODE上的内存）将比其他内存访问花费更长的时间，这是由于它们相对于执行所述内存访问的CPU所在的物理位置不同。进行常规内存分配时，通常无法控制该内存分配到哪里，因此如果DPDK在这样的系统上使用常规内存，就可能会导致以下的情况：在一个CPU上执行的线程却在无意中访问属于非本地NUMA节点的内存。

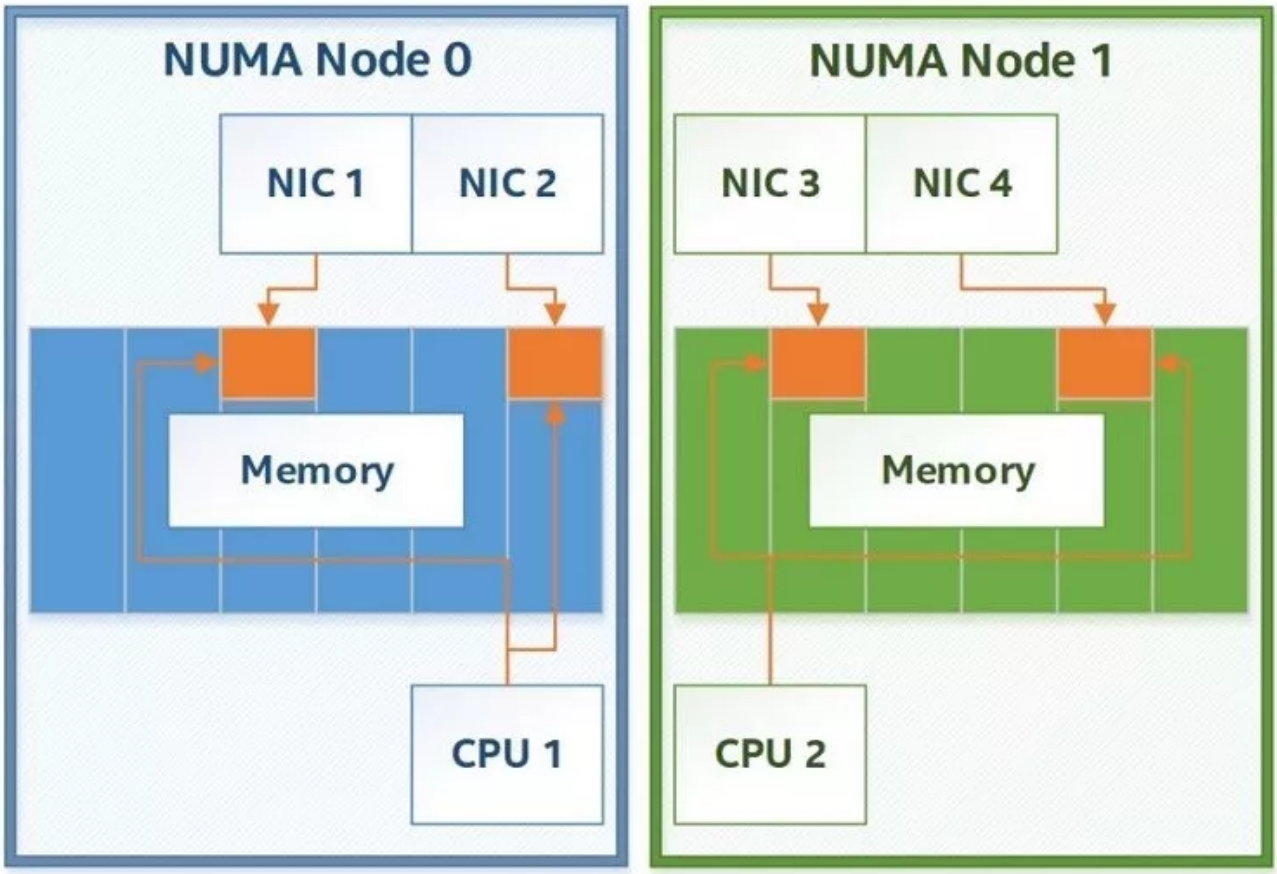


图2. 理想的NUMA节点分配。

虽然这种跨NUMA节点访问在所有现代操作系统上都比较少有，因为这样的访问都是都是NUMA感知的，而且即使没有DPDK还是有方法能对内存实施NUMA定位。但是DPDK带来的不仅仅是NUMA感知，事实上，整个DPDK API的构建都旨在为每个操作提供明确的NUMA感知。如果不明确请求NUMA节点访问（其中所述结构必须位于内存中），通常无法分配给定的DPDK数据结构。

DPDK API提供的这种明确的NUMA感知有助于确保用户应用在每个操作中都能考虑到NUMA感知；换句话说，DPDK API可以减少写出编写性能差的代码的可能性。

硬件、物理地址和直接内存存取（DMA）

DPDK被认为是一组用户态的网络包输入/输出库，到目前为止，它基本上保持了最初的任务声明。但是，电脑上的硬件不能处理用户空间的虚拟地址，因为它不能感知任何用户态的进程和其所分配到的用户空间虚拟地址。相反，它只能访问真实的物理地址上的内存，也就是CPU、RAM和系统所有其他的部分用来相互通信的地址。

出于对效率的考量，现代硬件几乎总是使用直接内存存取（DMA）事务。通常，为了执行一个DMA事务，内核需要参与创建一个支持DMA的存储区域，将进程内虚拟地址转换成硬件能够理解的真实物理地址，并启动DMA事务。这是大多数现代操作系统中输入输出的工作方式；然而，这是一个耗时的过程，需要上下文切换、转换和查找操作，这不利于高性能输入/输出。

DPDK的内存管理以一种简单的方式解决了这个问题。每当一个内存区域可供DPDK使用时，DPDK就通过询问内核来计算它的物理地址。由于DPDK使用锁定内存，通常以大页的形式，底层内存区域的物理地址预计不会改变，因此硬件可以依赖这些物理地址始终有效，即使内存本身有一段时间没有使用。然后，DPDK会在准备由硬件完成的输入/输出事务时使用这些物理地

址，并以允许硬件自己启动DMA事务的方式配置硬件。这使DPDK避免不必要的开销，并且完全从用户空间执行输入/输出。

IOMMU和IOVA

默认情况下，任何硬件都可以访问整个系统，因此它可以在任何地方执行DMA 事务。这有许多安全隐患。例如，流氓和/或不可信进程(包括在VM (虚拟机)内运行的进程)可能使用硬件设备来读写内核空间，和几乎其他任何存储位置。为了解决这个问题，现代系统配备了输入输出内存管理单元(IOMMU)。这是一种硬件设备，提供DMA地址转换和设备隔离功能，因此只允许特定设备执行进出特定内存区域(由IOMMU指定)的DMA 事务，而不能访问系统内存地址空间的其余部分。

由于IOMMU的参与，硬件使用的物理地址可能不是真实的物理地址，而是IOMMU分配给硬件的(完全任意的)输入输出虚拟地址(IOVA)。一般来说，DPDK社区可以互换使用物理地址和IOVA这两个术语，但是根据上下文，这两者之间的区别可能很重要。例如，DPDK 17.11和更新的DPDK长期支持(LTS)版本在某些情况下可能根本不使用实际的物理地址，而是使用用户空间虚拟地址(甚至完全任意的地址)来实现DMA。IOMMU负责地址转换，因此硬件永远不会注意到两者之间的差异。

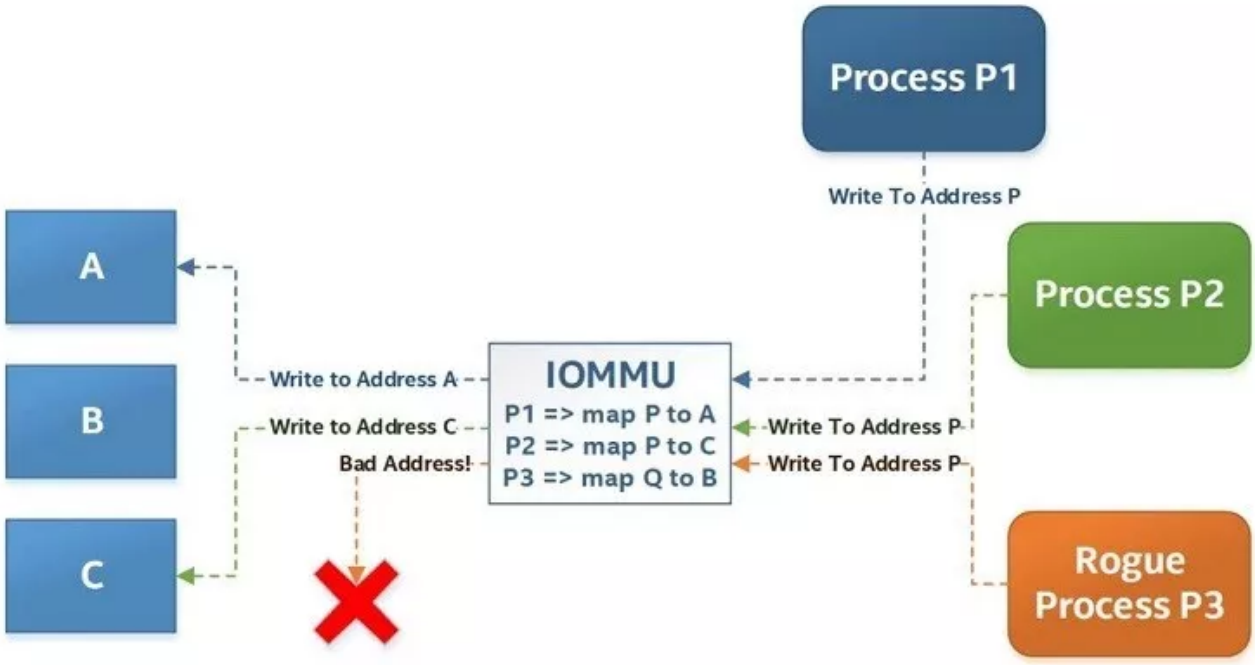
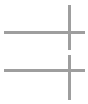


图3 .IOMMU将物理地址重新映射到IOVA地址的示例。

根据DPDK的初始化方式，IOVA地址可能代表也可能不代表实际的物理地址，但有一点始终是正确的:DPDK知道底层内存布局，因此可以利用这一点。例如，它可以以创建IOVA连续虚拟区域的方式映射页面，或者甚至利用IOMMU来重新排列内存映射，以使内存看起来IOVA连续，即使底层物理内存可能不连续。

因此，这种对底层物理内存区域的感知是DPDK工具包中的又一个利器。大多数数据结构不关心IOVA地址，但当它们关心时，DPDK为软件和硬件提供了利用物理内存布局的工具，并针对不同的用例进行优化。

请注意，IOMMU不会自行设置任何映射。相反，平台、硬件和操作系统必须进行配置，来使用IOMMU。 这种配置说明超出了本系列文章的范围，但是在DPDK文档和其他地方有相关说明。一旦系统和硬件设置为使用IOMMU，DPDK就可以使用IOMMU为DPDK分配的任何内存区域设置DMA映射。使用IOMMU是运行DPDK的推荐方法，因为这样做更安全，并且它提供了可用性优势。



内存分配和管理

DPDK不使用常规内存分配函数，如malloc()。相反，DPDK管理自己的内存。更具体地说，DPDK分配大页并在此内存中创建一个堆(heap)并将其提供给用户应用程序并用于存取应用程序内部的数据结构。

使用自定义内存分配器有许多优点。最明显的一个是终端应用程序的性能优势：DPDK创建应用程序要使用的内存区域，并且应用程序可以原生支持大页、NUMA节点亲和性、对DMA地址的访问、IOVA连续性等等性能优势，而无需任何额外的开发。

DPDK内存分配总是在CPU高速缓存行(cache line)的边界上对齐，每个分配的起始地址将是系统高速缓存行大小的倍数。这种方法防止了许多常见的性能问题，例如未对齐的访问和错误的数据共享，其中单个高速缓存行无意中包含(可能不相关的)多个内核同时访问的数据。对于需要这种对齐的用例(例如，分配硬件环结构)，也支持任何其他二次幂值 (当然> =高速缓存行大小)。

DPDK中的任何内存分配也是线程安全的。这意味着在任何CPU核心上发生的任何分配都是原子的，不会干扰任何其他分配。这可能看起来很无足轻重 (毕竟，常规glibc内存分配例程通常也是线程安全的)，但是一旦在多处理环境中考虑，它的重要性就会变得更加清晰。

DPDK支持特定风格的协同多处理，其中主进程管理所有DPDK资源，多个辅助进程可以连接到主进程，并共享由主进程管理的资源的访问。

DPDK的共享内存实现不仅通过映射不同进程中的相同资源 (类似于shmget () 机制) 来实现，还通过复制另一个进程中主进程的地址空间来实现。因此，由于两个进程中的所有内容都位于相同的地址，指向DPDK内存对象的任何指针都将跨进程工作，无需任何地址转换。这对于跨进程传递数据时的性能非常重要。

表1. 操作系统和DPDK分配器的比较。

	常规 Linux*分配器 1	DPDK rte_malloc
大页内存支持	未强制执行	系统默认值
NUMA 节点定位	未强制执行	系统默认值
访问 IOVA 地址	否	是
IOVA 连续存储	否	是
缓存对齐的分配	未强制执行	强制执行
分配的任意对齐	是	是
用于多处理的全共享内存	否	是
多进程线程安全	否	是

由于DPDK内存的共享性质，DPDK堆的线程安全则非常重要；任何线程不仅可以同时与任何其他线程分配和解除分配数据，而且任何进程都可以同时与多个其他进程分配和释放内存，而不引起任何竞争(race condition)。因为整个DPDK内存堆是跨进程共享的，所以在一个进程中分配内存并在另一个进程中引用或释放它也是非常安全的。

内存池

DPDK也有一个内存池管理器，在整个DPDK中广泛用于管理大型对象池，对象大小固定。它的用途很多——包输入/输出、加密操作、事件调度和许多其他需要快速分配或解除分配固定大小缓冲区的用例。DPDK内存池针对性能进行了高度优化，并支持可选的线程安全(如果用户不需要线程安全，则无需为之付费)和批量操作，所有这些都会导致每个缓冲区的分配或空闲操作周期计数达到两位数以下。

也就是说，即使DPDK内存池的主题出现在几乎所有关于DPDK内存管理的讨论中，从技术上讲，内存池管理器是一个建立在常规DPDK内存分配器之上的库。它不是标准DPDK内存分配工具的一部分，它的内部工作与DPDK内存管理例程完全分离 (并且非常不同) 。因此，这超出了本系列文章的范围。但是，有关DPDK内存池管理器库的更多信息可以在DPDK文档中找到。

结论

本文介绍了构成DPDK内存管理子系统基础的许多核心原理，并证明了DPDK的高性能并不是偶然，而是其体系架构的必然结果。

本系列接下来的文章将深入探讨IOVA寻址及其在DPDK中的使用；以历史的视角，回顾DPDK长期支持（LTS）版本17.11及更早版本中提供的内存管理功能；同时也会介绍18.11及更高版

本DPDK版本中做出的更改和提供的新功能。

本系列文章:

DPDK内存篇（一）:基本概念

DPDK内存篇（二）:深入学习IOVA

DPDK内存篇（三）:DPDK17.11及早期版本

DPDK内存篇（四）:DPDK 18.11及其他

推荐阅读

绝对干货！初学者也能看懂的DPDK解析

基于DPDK的Open vSwitch概述

DPDK19.05 已发布，新版本都有哪些新功能和变化？