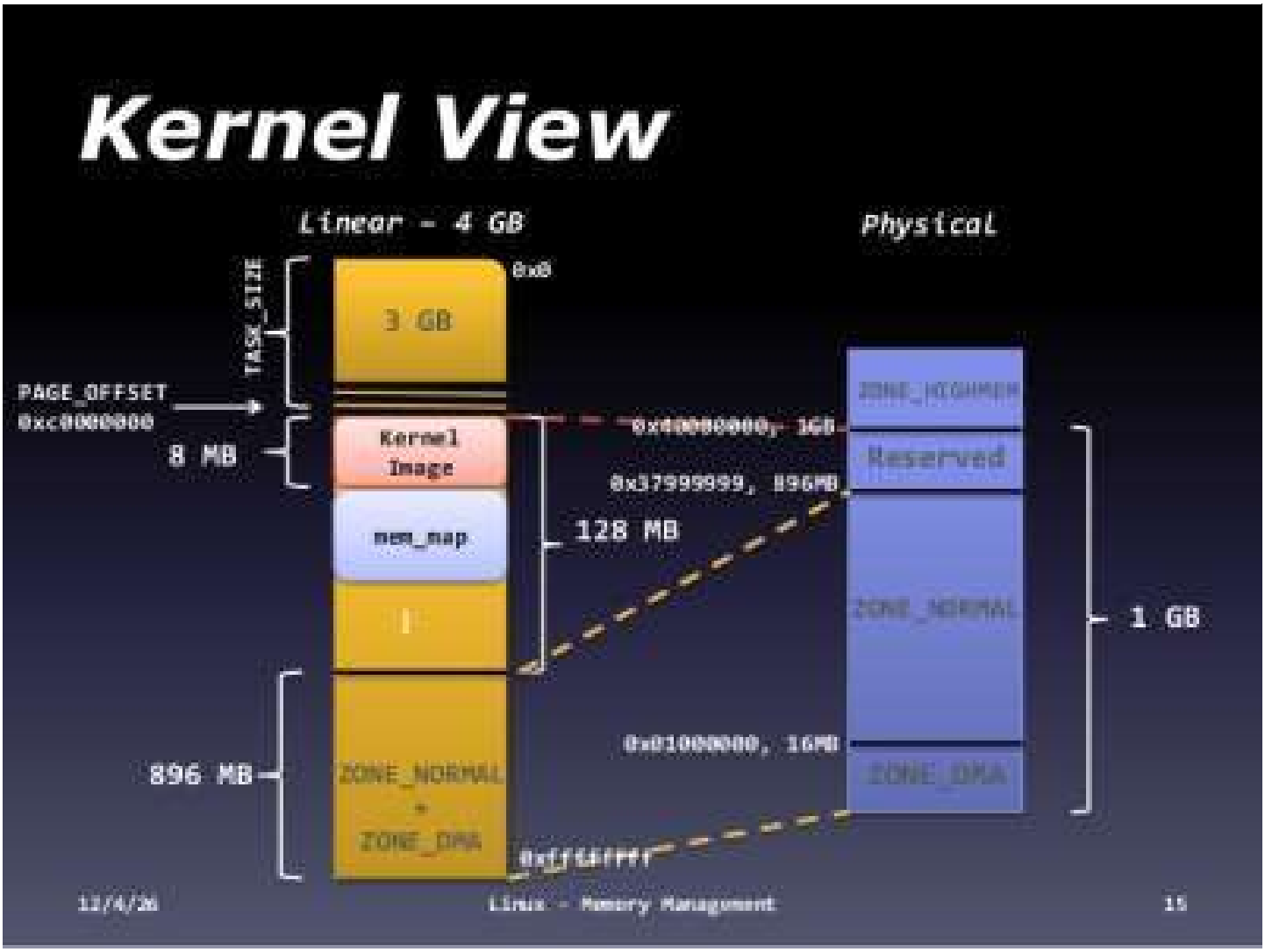


浅谈Linux内存管理

lecury
程序员，检索系统，搜索引擎。

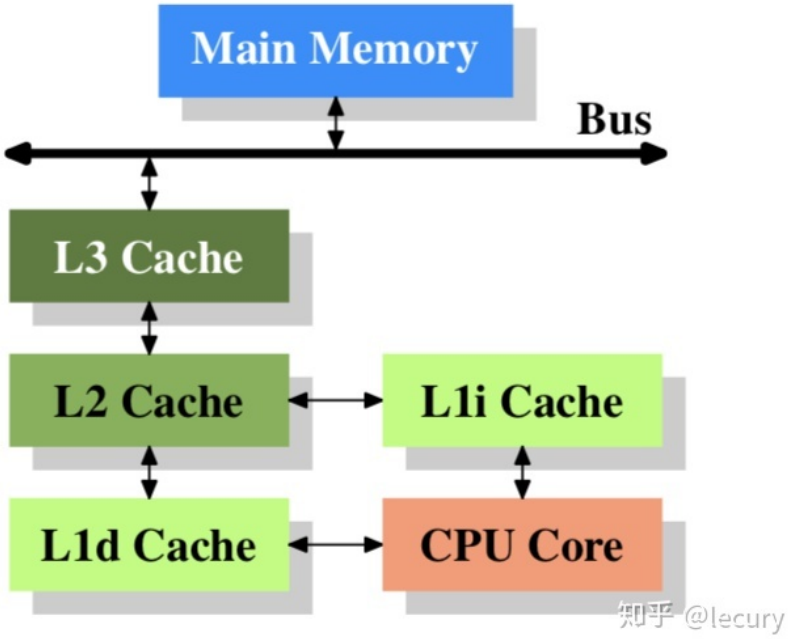
1. 扫盲篇

1.1 操作系统存储层次

常见的计算机存储层次如下：

- 寄存器：CPU提供的，读写ns级别，容量字节级别。
- CPU缓存：CPU和CPU间的缓存，读写10ns级别，容量较大一些，百到千节。
- 主存：动态内存，读写100ns级别，容量GB级别。
- 外部存储介质：磁盘、SSD，读写ms级别，容量可扩展到TB级别。

CPU内的缓存示意图如下：



其中 L1d 和 L1i 都是CPU内部的cache，

- L1d 是数据cache。
- L1i 是指令缓存。
- L2是CPU内部的，不区分指令和数据的。
- 由于现代PC有多个CPU，L3缓存多个核心共用一个。

对于编程人员来说，绝大部分观察主存和外部存储介质就可以了。如果要做极致的性能优化，可以关注L1、L2、L3的cache，比如nginx的绑核操作、pthread调度会影响CPU cache等。

1.2 内存管理概述

MMU（内存管理单元）：通过CPU将线性地址转换成物理地址。

1.2.1 虚拟内存

物理内存是有限的（即使支持了热插拔）、非连续的，不同的CPU架构对物理内存的组织都不同。这使得直接使用物理内存非常复杂，为了降低使用内存的复杂度，引入了虚拟内存机制。

虚拟内存抽象了应用程序物理内存的细节，只允许物理内存保存所需的信息（按需分页），并提供了一种保护和控制进程间数据共享数据的机制。有了虚拟内存机制之后，每次访问可以使用更易理解的虚拟地址，让CPU转换成实际的物理地址访问内存，降低了直接使用、管理物理内存的门槛。

物理内存按大小被分成页框、页，每块物理内存可以被映射为一个或多个虚拟内存页。这块映射关系，由操作系统的页表来保存，页表是有层级的。层级最低的页表，保存实际页面的物理地址，较高层级的页表包含指向低层级页表的物理地址，指向顶级的页表的地址，驻留在寄存器中。当执行地址转换时，先从寄存器获取顶级页表地址，然后依次索引，找到具体页面的物理地址。

1.2.2 大页机制

虚拟地址转换的过程中，需要好几个内存访问，由于内存访问相对CPU较慢，为了提高性能，CPU维护了一个TLB地址转换的cache，TLB是比较重要且珍稀的缓存，对于大内存工作集的应用程序，会因TLB命中率低大大影响到性能。

为了减少TLB的压力，增加TLB缓存的命中率，有些系统会把页的大小设为MB或者GB，这样页的数目少了，需要转换的页表项也小了，足以把虚拟地址和物理地址的映射关系，全部保存于TLB中。

1.2.3 区域概念

通常硬件会对访问不同的物理内存的范围做出限制，在某些情况下设备无法对所有的内存区域做DMA。在其他情况下，物理内存的大小也会超过了虚拟内存的最大可寻址大小，需要执行特殊操作，才能访问这些区域。这些情况下，Linux对内存页的可能使用情况将其分组到各自的区域中（方便管理和限制）。比如ZONE_DMA用于指明哪些可以用于DMA的区域，ZONE_HIGHMEM包含未永久映射到内核地址空间的内存，ZONE_NORMAL标识正常的内存区域。

1.2.4 节点

多核CPU的系统中，通常是NUMA系统（非统一内存访问系统）。在这种系统中，内存被安排成具有不同访问延迟的存储组，这取决于与处理器的距离。每一个库，被称为一个节点，每个节点Linux构建了一个独立的内存管理子系统。一个节点有自己的区域集、可用页和已用页表和各种统计计数器。

1.2.5 page cache

从外部存储介质中加载数据到内存中，这个过程是比较耗时的，因为外部存储介质读写性能毫秒级。为了减少外部存储设备的读写，Linux内核提供了Page cache。最常见的操作，每次读取文件时，数据都会被放入页面缓存中，以避免后续读取时所进行昂贵的磁盘访问。同样，当写入文件时，数据被重新放置在缓存中，被标记为脏页，定期的更新到存储设备上，以提高读写性能。

1.2.6 匿名内存

匿名内存或者匿名映射表示不受文件系统支持的内存，比如程序的堆栈隐式创立的，或者显示通过mmap创立的。

1.2.7 内存回收

贯穿系统的生命周期，一个物理页可存储不同类型的数据，可以是内核的数据结构，或是DMA访问的buffer，或是从文件系统读取的数据，或是用户程序分配的内存等。

根据页面的使用情况，Linux内存管理对其进行了不同的处理，可以随时释放的页面，称之为可回收页面，这类页面为：页面缓存或者是匿名内存（被再次交换到硬盘上）

大多数情况下，保存内部内核数据并用DMA缓冲区的页面是不能重新被回收的，但是某些情况下，可以回收使用内核数据结构的页面。例如：文件系统元数据的内存缓存，当系统处于内存压力情况下，可以从主存中丢弃它们。

释放可回收的物理内存页的过程，被称之为回收，可以同步或者异步的回收操作。当系统负载增加到一定程序时，kswapd守护进程会异步的扫描物理页，可回收的物理页被释放，并逐出备份到存储设备。

1.2.8 compaction

系统运行一段时间，内存就会变得支离破碎。虽然使用虚拟村内可以将分散的物理页显示为连续的物理页，但有时需要分配较大的物理连续内存区域。比如设备驱动程序需要一个用于DMA的大缓冲区时，或者大页内存机制分页时。内存compact可以解决了内存碎片的问题，这个机制将被占用的页面，从内存区域合适的移动，以换取大块的空闲物理页的过程，由kcompactd守护进程完成。

1.2.9 OOM killer

机器上的内存可能会被耗尽，并且内核将无法回收足够的内存用于运行新的程序，为了保存系统的其余部分，内核会调用OOM killer杀掉一些进程，以释放内存。

1.3 段页机制简介

段页机制是操作系统管理内存的一种方式，简单的来说，就是如何管理、组织系统中的内存。要理解这种机制，需要了解一下内存寻址的发展历程。

- 直接寻址：早期的内存很小，通过硬编码的形式，直接定位到内存地址。这种方式有着明显的缺点：可控性弱、难以重定位、难以维护
- 分段机制：8086处理器，寻址空间达到1MB，即地址线扩展了20位，由于制作20位的寄存器较

了 段 的概念，即 内存地

- 分页机制：随着寻址空间的进一步扩大、虚拟内存技术的引入，操作系统引入了分页机制。引入分页机制后，逻辑地址经过段机制转换得到的地址仅是中间地址，还需要通过页机制转换，才能得到实际的物理地址。 逻辑地址 -->(分段机制) 线性地址 -->(分页机制) 物理地址 。

段页机制详见：https://blog.lecury.cn/2017/05/05/内存寻址之段页存储机制分析/

2. 进阶篇

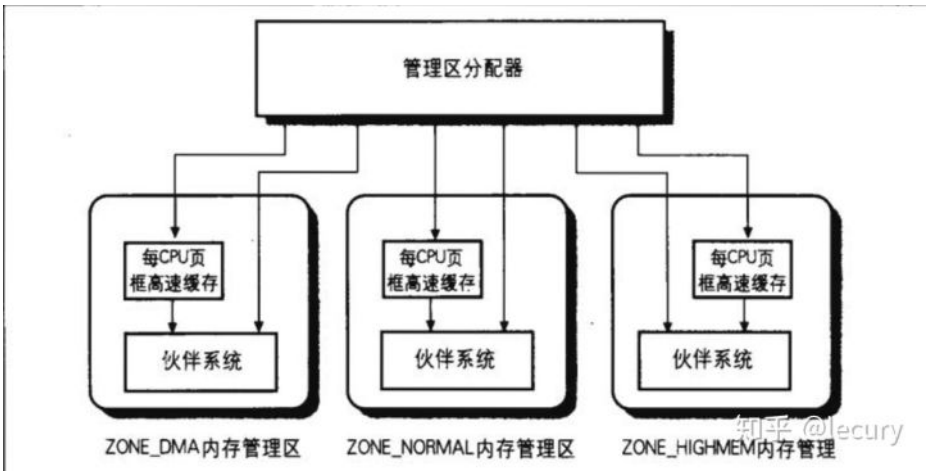
2.1 内存分配

2.1.1 大块内存的分配

扫盲篇也提到，Linux基于段页式机制管理物理内存，内存被分割成一个个页框，由多级页表管理。除此之外，由于硬件的约束：

- DMA处理器，只能对RAM的前16MB寻址。
- 32位机器CPU最大寻址空间，只有4GB，对于大容量超过4GB的RAM，无法访问所有的地址空间。

Linux还将物理内存划分为不同的管理区：ZONE_DMA、ZONE_NORMAL、ZONE_HIGHMEM，每个管理区都有自己的描述符，也有自己的页框分配器，示意图如下：



对于连续页框组的内存分配请求，是由管理区分配器完成，每个管理区的页框分配是通过伙伴系统算法来实现。内核经常请求和释放单个页框，为了提高性能，每个内存管理区，还定义了一个CPU页框高速缓存，包含一些预选分配的页框。

伙伴系统算法：内核为分配一组连续的页框而建立的一种健壮、高效的分配策略，这种策略缓解了内存碎片的发生。算法的核心思想：是把所有的空闲页框分组为11个块链表，每个块链表分别包含1、2、4、8、16、...、512、1024个连续页框。举个简单的例子，说明算法的工作过程。

假设需要256个页框的连续内存，算法先在256个页框的链表中，检查是否还有空闲块，如果有就分配出去。如果没有，算法会找到下一个更大的512页框的链表，如果存在空闲块，内核会把512页框分割成两部分，一半用来分配，另一半插入到256页框的链表中。

2.1.2 小块内存的分配

伙伴系统算法采用页框作为基本的内存区，这适合于大块内存的请求。对于小块内存的分配，是采用的slab分配器算法来实现的。slab并没有脱离伙伴系统算法，而是基于伙伴系统分配的大内存基础上，进一步细分小内存对象的分配。slab 缓存分配器提供了很多优点，

- 首先，内核通常依赖于对小对象的分配，它们会在系统生命周期内进行无数次分配，slab 缓存分配器通过对类似大小的对象进行缓存，从而避免了常见的碎片问题。
- slab 分配器还支持通用对象的初始化，从而避免了为同一目而对一个对象重复进行初始化。
- 最后slab 分配器还可以支持硬件缓存对齐和着色，这允许不同缓存中的对象占用相同的缓存行，从而提高缓存的利用率并获得更好的性能。

备注: slab着色主要是为了更好的利用CPU L1 cache，所使用的地址偏移策略。如果slab分配对象后还有空间剩余，就会把剩余的空间进行着色处理，尽可能将slab对象分散在L1不同的cache line中。

2.1.3 非连续内存的分配

把内存区映射到一组连续的页框是最好的选择，这样会充分利用高速缓存。如果对内存区的请求不是很频繁，那么分配非连续的页框，会是比较好的选择，因为这样会避免外部碎片，缺点是内核的页表比较乱。Linux以下方面使用了非连续内存区：

- 为活动交换区分配数据结构。
- 给某些I/O驱动程序分配缓冲区。
- 等

2.2 实存、虚存

实存：进程分配的、加载到主存中的内存。包含来自共享库的内存，只要这些库占用的页框还在主存中，也包含所有正在使用的堆栈和堆内存。可以通过 ps -o rss 查看进程的实存大小。

虚存：包含进程可以访问的所有内存，包含被换出、已经分配但还未使用的内存，以及来自共享库的内存。可以通过 ps -o vsz 查看进程的虚存大小。

举个例子，如果进程A具有500K二进制文件并且链接到2500K共享库，则具有200K的堆栈/堆分配，其中100K实际上在内存中（其余是交换或未使用），并且它实际上只加载了1000K的共享库然后是400K自己的二进制文件：

```
RSS: 400K + 1000K + 100K = 1500K
VSZ: 500K + 2500K + 200K = 3200K
```

实存和虚存是怎么转换的呢？当程序尝试访问的地址未处于实存中时，就发生页面错误，操作系统必须以某种方式处理这种错误，从而使应用程序正常运行。这些操作可以是：

- 找到页面驻留在磁盘上的位置，并加载到主存中。
- 重新配置MMU，更新线性地址和物理地址的映射关系。
- 等。

随着进程页面错误的增长，主存中可用页面越来越少，为了防止内存完全耗尽，操作系统必须尽快释放主存中暂时不用的页面，以释放空间供以后使用，方式如下：

- 将修改后的页面写入到磁盘的专用区域上（调页空间或者交换区）。
- 将未修改的页面标记为空闲（没必要写入磁盘，因为没有被修改）。

调页或者交换是操作系统的正常部分，需要注意的是过度交换，这表示当前主存空间不足，页面换出抖动对系统极为不利，会导致CPU和I/O负载升高，极端情况下，会造成操作系统所有的资源花费在调页层面。

2.3 page cache

Linux中通过page cache机制来加速对磁盘文件的许多访问，当它首次读取或写入数据介质时，Linux会将数据存储在未使用的内存区中，通过这些区域充当缓存，如果再次读取这些数据时，直接从内存中快速获取该数据。当发生写操作时，Linux不会立刻执行磁盘写操作，而是把page cache中的页面标记为脏页，定期同步到存储设备中。

可以通过 `free -m` 来查看page cache情况：

```
total       used       free     shared    buffers     cached
Mem:         32013      31288        724          0        241      12000
-/+ buffers/cache:      19046      12966
Swap:        32767      23134       9633
```