【概述】-Linux内核三驾马车之-内存管理

Linux Kernel内存管理是内核开发三驾马车之一，其余两个是进程和调度管理、IO管理。内存管理可能是这三个里面最晦涩难懂的，因为既涉及很多硬件MMU/IOMMU/TLB/Cache/DMA、又涉及到与硬件对应的数据结构和接口、还涉及到极多的算法，不容易学习更不容易精通.

click here back to Homepage

click here back to Category

click here back to Linux Kernel

本文着重记录内存管理知识点和脉络，其中部分内容是根据宋宝华老师《内存管理微课》并加入个人理解整理。如果没有特别指出均考虑是在32bit的ARM平台Cotex-A系列架构上的情况。

Linux内存管理映射关系图

下图描述内存管理的"虚拟地址"《==》"物理地址"统一关系图：

TODO...

Linux分页机制

内存分页机制（PAGING MECHANISM）是内存管理的基础，其原理来自于CPU的硬件实现.

内存寻址相关概念

各种地址的解释

① 物理地址：是一个整数，它不是一个指针，它是从0开始标记物理内存单元的标号。

② 虚拟地址：是一个指针，在开启MMU后从CPU看到的都是它。虚拟地址通过页表映射到物理地址。

③ 逻辑地址：是Intel平台这种使用段式内存管理的一个概念。

④ 线性地址：也是Intel平台段式内存管理产生的一个概念。

⑤ 总线地址：是给外设用的，从设备端看到的地址都是总线地址。

各种地址间关系

③逻辑地址、②虚拟地址：

- 在ARM平台上没有段式管理，尤其是Linux内核只靠页是管理就可以完成内存管理，因此，逻辑地址==虚拟地址

④线性地址、③逻辑地址、②虚拟地址：

- 在ARM平台上，线性地址==逻辑地址==虚拟地址

⑤总线地址、①物理地址：

- 总线地址和物理地址之间的转换关系是由系统设计决定的，kernel中的转换函数也是arch相关的，二者是一一映射的，通常是线性关系。而且在绝大多数平台上，物理地址和总线地址起始地址也是一样的。

MMU内存管理单元

MMU（Memory Management Unit），内存管理单元，一旦开启MMU之后，CPU只能看到虚拟地址（包含内存空间所有的部分，比如register、IO、memory），而MMU能访问到的是物理地址。

MMU的功能：

① 查询页表找虚拟地址映射到的物理地址

② 页访问权限控制

- RWX：读、写、执行
- Kernel/User：内核态访问、用户态访问，确保了“非特权模式”无法访问“特权模式”才能访问的内存，隔离内核与用户态

> Tips:
> CPU硬件漏洞 - 熔断攻击 meltdown：基于timing原理的旁路攻击，突破了硬件限制，在用户态读到了内核态内存。
> 三个关键点：
> 【1】 MMU触发缺页异常到CPU处理完该异常需要一定时间（a=*kernel_addr）；
> 【2】 CPU乱序执行此时还在不停的执行后续的攻击代码,导致内核地址kernel_addr上内存数据已经被读入cache（b=probe_array[a]）；
> 【3】 cache硬件特性又决定该cache单元的内容不会随指令被抛弃，可以利用cache命中速度去探测（for循环探测数组probe_array[]）；
> 这样的条件叠加在一起，才造成了可以通过对比probe_array[]各个元素的访问时间来找到数据下标a，这个a正是内核地址kernel_addr上的内存数据。

③ 产生缺页异常 - PageFault

- 第一种情况是：页表PageTable中虚拟地址项没有对应的物理地址（这里包含着着虚拟地址的vma合法/vma非法）
- 第二种情况是：页表PageTable中记录的访问页的权限不对（这里也包含着虚拟地址的vma合法/vma非法）

> Tips:
> 【1】缺页异常产生的直接原因是： MMU查找“页表内容”+判断“页表权限”，有不满足的，即产生PageFault
> 【2】缺页异常处理的动作是：Linux Kernel的异常处理handler函数中会去判断vma记录，只有vma合法（区间/权限）的PageFault内核才会给分配对应的物理页并且修改页表。

MPU内存保护单元

MPU（memory protection unit），内存保护单元，只能做内存权限保护，不能做虚拟地址到物理地址映射。

存在于某些ARM系列上面，比如Cortex-R系列。

MPU的功能：

① 访问权限保护

② 触发异常

PageTable页表

页表，它的软件实现依赖于硬件体系结构，arm和x86等等均不相同。

PageTable的功能：

① 记录虚拟地址到物理地址映射关系

② 记录页访问权限（RWX、Kernel/User）

TLB快表

TLB，页表高速缓存，类似于CPU的cache，空间有限，保存了部分最常用的页表项，用于MMU查询页表硬件加速。

TLB命中则MMU直接返回对应物理地址，TLB未命中则MMU需要再去查询内存中页表，然后刷新到TLB中某一项，再返回对应物理地址。

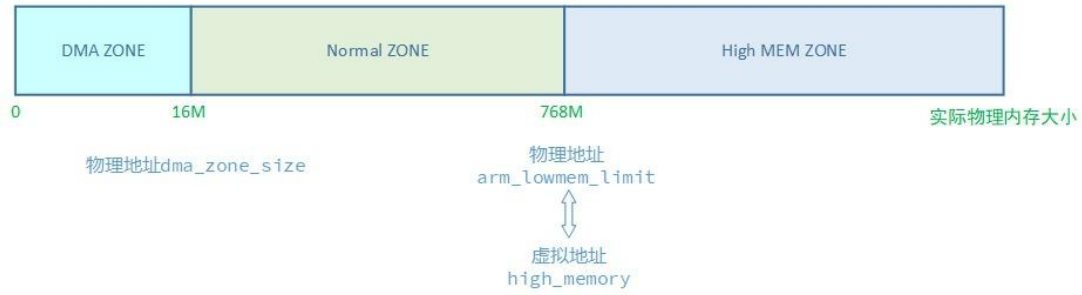
ASID与用户进程标记：

ARM使用asid来标记不同的用户进程空间的TLB，asid存在于mm_struct->mm_context_t->atomic64_t->counter的低8bit，它对应TLB中一个硬件buffer，该buffer表现为一个bitmap，它标识该TLB属于哪个用户进程。但是由于硬件buffer容量有限，只能最大标记2^8=256个进程，因此当超出时用counter的高位来标识generation++，这样软硬结合来实现所有用户进程的标识。

~~*关于MMU/TLB/PageTable更详细的描述见《MMU-AArch32》*~~

内存分区（ZONE）

内存分区指的是“物理内存”，而非“虚拟内存”



说明如下：

① 这是ARM平台上实际内存条物理地址；

② 0是没有加入实际平台PHY_OFFSET偏移的初始物理地址；

③ 16M、768M在不同平台并不一致，DMA ZONE大小在dts中配置，High MEM ZONE起始是根据Linux内核中配置的虚拟地址空间vmalloc映射区大小通过一系列计算得出来的，arm平台不配置vmalloc映射区大小时默认是240M，此时高端内存起始于768M；

> Tips:
> 网上很多资料写DMA是0~16M、Normal是16~896M，High MEM起始于896M，那只不过是32bit的x86平台固定的定义而已。
> 请忘掉网上各种无ARCH体系架构这一前提条件的资料中给出的896M这个破数字！不要在其它ARCH上混淆。

DMA ZONE

DMA ZONE产生原因：由于DMA IP的硬件缺陷引起的。

- DMA IP访问内存时会有一些限制，不能访问全部内存，也即当DMA IP挂在某些bus上时可能地址线只能访问到某个地址以下的部分，即便是被分配了该地址以上的内存这个DMA IP也访问不到。

  > Tips:
  > 比如，x86上ISA总线是16bit总线可以使用24bit内存地址，挂在它上的DMA IP就只能访问16M以下内存。

- DMA ZONE并不是给DMA专用的，可以用于任何用途，而只是Linux内核内存分配器需要了解的一个标识。

需要注意两个关键点：

① DMA ZONE值的配置

- 当SOC上只有1个DMA IP时，如果是有访问范围缺陷的，那么就配置成它能访问的范围。
- 当SOC上有多个DMA IP时，DMA ZONE需要配置成各个DMA IP能访问范围的最小值。

  > Tips:

> 比如，在x86-64上就有2个DMA ZONE，一个是“DMA32 ZONE”专门给32bit的DMA IP使用的，一个是“DMA ZONE”给有缺陷的64bit的DMA IP使用的。

② GFP_DMA标记

- 在有缺陷的DMA IP申请内存时，需要给Linux内核内存分配器一个标记GFP_DMA，让内存分配器知道需要给该DMA IP分配DMA ZONE中的内存。

HighMEM ZONE

HighMEM ZONE产生原因：kernel能使用虚拟地址空间3~4G小于实际使用的物理内存（>1G）。

- 在虚拟地址空间划分时，0~3G是User Space空间，3~4G是Kernel Space空间。
- Linux内核为了简化，开机就会把物理内存做一一线性映射到3~4G空间，但如果物理内存大于1G就无法完成一一线性映射了。
- 在arm平台上，Linux就设定了一个域值arm_lowmem_limit变量，物理内存高于该阈值的部分，就叫HighMEM，它对应的ZONE就是HighMEM ZONE。该阈值选取不是固定的，而是通过计算需要保留的vmalloc区域大小得出的（vmalloc区域大小可以配置），该阈值对应到虚拟地址空间上就是high_memory变量。

需要注意的关键点：

- HighMEM ZONE中的内存在内核不是一一线性映射的。
- 内核一般不使用HighMEM ZONE，如果需要使用的话，必须先通过特殊API kmap做映射。
- 在调用kmap对HighMEM做映射时，x86是映射到虚拟地址空间3G以上部分，arm是映射到虚拟地址空间3G以下的（3G-2M）到3G部分

Normal ZONE

相对于HighMEM，物理内存低于该阈值的部分，就叫做LowMEM，它对应的ZONE包含DMA ZONE + Normal ZONE。

需要注意的关键点：

- LowMEM中内存虚实地址之间是一一线性映射的。可通过两个API直接线性转化： `phys_to_virt()` 和 `virt_to_phys()`
- LowMEM内存开机就做映射指的是建立好内核页表，而不是直接用掉了，内核用该内存时是需要申请的。
- Normal ZONE只是LowMEM的一部分，另外一部分是DMA ZONE。

内存分配（各级分配器）

BUDDY内存分配器

buddy内存分配器构建在物理内存之上，以内存分区ZONE为基础，每个ZONE都会使用一个同样的内存管理算法buddy来管理该ZONE中的空闲物理页。

buddy算法的数学原理：

任何一个正整数都可以拆解成多个2^n之和这种多项式的表示形式。

算法四个要点：

① 按页管理 —— 最小单元是1页（通常是4K），其它大小的单元均是1页的2^n倍的连续物理内存.

② 分组管理 —— 将空闲物理内存按照“符合(4K*(2^order))大小”分成11个组，order取值范围是0~10，即分成连续的1页、2页、4页、8页...1024页

③ 地址对齐 —— 组内页框起始地址对齐到(4K*(2^order))边界上，order即该组的order值。

④ 拆分合并 —— 当申请连续页框发现对应组已经空了时去拆分更高一阶的组内页，当两个连续的空闲页框符合地址对齐规律时合并成一个更高一阶的连续页框挂到对应组内。

算法缺陷：

buddy算法旨在解决物理内存的“外碎片”问题，该算法虽然优秀但是问题解决的并不彻底，缺陷依然存在于以下2个方面

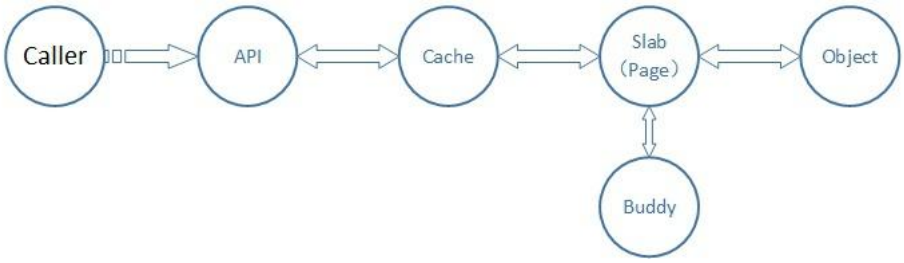
① 当没有连续空闲page页可以被合并时（例如，有很多非连续的4K页分散在内存当中无法合并）

② 有连续空闲page页但第一个空闲page页物理地址不符合(4K*(2^order))边界对齐规则而导致这些page页无法合并时

> Tips:
> 同时buddy算法还引入了另一个缺陷：内存“内碎片”问题!!!
> 因为算法局限，分配出的内存只能是(4K*(2^order))大小，如果直接使用buddy分配出的内存就可能造成有很大一部分内存是浪费掉的。

SLAB内存分配器

slab内存分配器构建于buddy内存分配器之上，针对于heap类型内存申请做二次分配管理，用于解决内存“内碎片”问题。



内核中的slab可以分成2类：

① 专用slab：比如TCPv6、UDPv6等内核需要经常分配和释放的常用数据结构

② 通用slab：比如kmalloc-64、kmalloc-128等，只区分大小、不区分用途

slab和buddy的关系概括成2点：

① slab的内存来自于buddy

② slab和buddy在内存分配算法上级别是对等的

> Tips：
> 【1】slab内存分配API kmalloc/kfree与buddy并不是一一对应关系
> 【2】同理，用户态c库函数malloc/free与内核内存分配也不是一一对应关系，在c库中有一个API函数mallopt可以控制一系列c库中free memory阈值来决定c库是否将已经从内核申请的内存再free给buddy。（这个特性在写实时进程RT编程时非常有用，先申请一个大的内存池、赋值、释放，让C库把这部分内存hold住，这样后续再malloc时都是在C库里面的用户态操作，不再和内核打交道，这样程序的实时性就好了）

~~关于slab分配器更详细的描述，可以查看《内存管理：内存分配器-slub》~~

CMA内存分配器

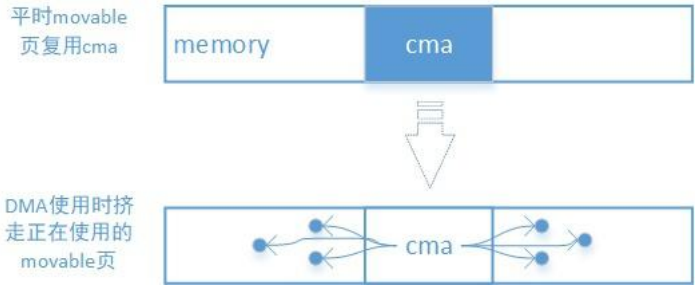
连续内存分配器，continuous memory allocater，是三星的同学发明的，它并非独立存在而是与DMA的API结合在一起的，是DMA内存申请API的后端。

cma目的：

为DMA提供连续的物理地址空间，避免因buddy内存碎片化导致DMA无法使用内存的情况。

cma原理：

- 预留物理内存给DMA，但可以被用户态进程（申请movable页面）复用
- 用户态进程物理内存迁移migration、以及用户态进程页表重建
  即，当DMA需要使用在dts中预留的cma内存时，cma分配器会从非cma区域申请对应数量的4K页，然后把cma区域中正在使用的物理地址copy过去，最后重建用户态进程页表让虚拟地址指向新的物理地址。整个过程是在内核中完成的，用户态进程是感受不到的。



内核文档/Documentation/devicetree/bindings/reserved-memory/reserved-memory.txt中详细记录了如何指定全局cma池，如何给特点设备指定cma池。

ION内存分配器

Google的发明，用于用户态和内核态共享内存（减少一次拷贝），可以配合iommu去完成外设内存访问：

- 对应用 —— 通过提供fd让应用在用户态做mmap来使用虚拟地址访问内存
- 对外设 —— 通过iommu driver中建立iommu pagetable完成虚实映射，外设可以通过iommu提供的虚拟地址访问内存

ion的heap

ion分为几个不同类别的heap，用于管理从不同的位置分配出的内存：

- ION_HEAP_TYPE_SYSTEM：内核通过vmalloc分配内存
- ION_HEAP_TYPE_SYSTEM_CONTIG：内核通过kmalloc分配内存
- ION_HEAP_TYPE_CARVEOUT：内核从reserved memory分配内存，物理连续空间
- ION_HEAP_TYPE_IOMMU：内核从IOMMU管理的内存范围分配内存，即已经建立IOMMU pagetable映射关系的内存范围
- ION_HEAP_TYPE_CP：内核从reserved memory分配内存，物理连续空间
- ION_HEAP_TYPE_DMA：内核通过DMA API分配内存
- ION_HEAP_TYPE_CUSTOM：用户自定义，永远放在最后一个
- ION_NUM_HEAPS：数量

ion的数据结构

TODO...

内存分配API间关系澄清

kmalloc、vmalloc、ioremap之间关系：

① kmalloc是slab内存申请接口，申请到的是kernel可以直接使用的内存的虚拟地址空间，即“Low MEM映射区”或者叫“一一映射区”，它不需要修改页表，因为开机内核已经做了一一线性映射，它申请到的虚拟地址是连续的、物理地址也是连续的。

> Tips:
> 当kmalloc使用的gfp_flags=__GFP_HIGHMEM时也可以从高端内存分配，但是一般不这么做。

② vmalloc是内核申请非常大块内存的接口，申请的内存使用的虚拟地址空间是“vmalloc映射区”，该区域与物理地址空间（Low MEM/High MEM）开机并未做任何映射，因此申请时就需要修改内核页表，它申请到的虚拟地址是连续的、物理地址可以是离散的。

vmalloc函数是调用__vmalloc_node_flags(__GFP_HIGHMEM)实现的，它分配物理内存Zone顺序是High MEM Zone->Normal Zone->DMA Zone。

> Tips:
> 这里需要说明的是：
> 【1】 尽管vmalloc优先申请High MEM但这并不是这个函数族的实际用途，因为高端内存的页不能永久地映射到内核地址空间，vmalloc分配的高端内存通常用于用户过程中，内核自身会尽量避免使用High MEM。
> 【2】 如果需要将High MEM长期映射(作为持久映射)到内核地址空间中，必须使用kmap(*page)函数，对应的虚拟地址映射区是KMAP映射区，长期高端内存映射使用着内核页表中一个专门的页表, 其地址存放在变量pkmap_page_table中。
> 【3】 正是因为vmalloc函数总是通过调用alloc_pages(__GFP_HIGHMEM)来实现首先映射高端内存，所以看似与kmap功能重复，这其实确实是造成了理解上的混乱.

③ ioremap通常给寄存器使用，它会把register物理地址映射到“vmalloc映射区”中虚拟地址上，也需要修改内核页表。

> Tips:
> 【1】 “Low MEM”与“Low MEM映射区”不是一个概念：前一个指的是实际物理内存上一段，后一个指的是内存虚拟地址空间0~4G上一段；两者的联系是，Low MEM会被映射到Low MEM映射区。
> 【2】 “内存”与“内存空间”不是一个概念：内存肯定是属于内存空间的，但是register也是属于内存空间的，因为软件访问它们的汇编指令都是LDR/STR并且CPU访问时的地址都是通过virt->mmu->phys转换而来的，基本没有区别。

内存分配特点

Lazy行为：

内核总是以最lazy的方式给应用程序分配内存.即，不到最后向内存写数据的那一刻应用程序都拿不到真实内存，应用程序认为分配成功了只是内核告之应用程序可以分配这么多vss内存的一个“假象”，只有当应用程序写一次申请内存任意一页时才会真的申请成功一页。

以应用程序用C库malloc申请100M内存为例，内核执行了如下操作：

- 内核调用brk/sbrk将堆vss扩大并反馈给用户进程内存申请成功；
- 内核将用户申请的100M内存每一个4K页，都以只读方式（页表记录只读）映射到同一个全部清零的4K物理页面；
- 当每一个4K写数据时，MMU会检查页表并发现是只读的，这就会产生一个PageFault，当内核收到PageFault时，对应的handler处理函数会从硬件寄存器读取PageFault发生的地址和原因；
- 内核检查发生PageFault的地址和原因，发现是写一个malloc区域并且原因是写只读的Page，这时内核就会新申请一个物理页，然后执行copy on write将之前全清零的物理页拷贝到新申请的物理页；

- 内核修改进程页表，将虚拟地址指向新申请物理页的物理地址，同时将该Page权限修改成R+W；

这就是内核“欺骗”应用程序的lazy行为，但是需要注意的是内核不欺骗自己，内核调用kmalloc、vmalloc去申请内存时，当返回addr时就是已经建立好了页表拿到了物理内存。

OOM：

正是因为内核前面lazy行为“欺骗”了应用程序，可能会因为时间差，导致造成应用程序以为有内存但是真正去写的时候却发现拿不到内存，因此内核会启动oom这个机制。

> Tips:
> 这里需要注意的是，虽然内核有lazy分配行为，但是内核也提供了一个feature叫overcommit，会产生一个/proc/sys/vm/overcommit_memory节点，当该节点值=0时会去检查是否有足够的物理内存共应用程序的虚拟内存去映射，检查失败就会导致应该挂掉，当该节点值=1时不去检查物理内存剩余量，直接返回成功。

内核运行时，会对每个task进行oom打分，主要依据是耗费内存大小，通常耗费内存越大打分会越高。同时，内核会照顾用户权限、以及oom_adj因子。综合算出一个分数来，这样，打分最高的task就是最该死的task。

LMK:

TOOD...

文件系统内存与页交换

两种不同的page页面

① anonymous page：

用户进程使用的，没有文件背景的页面，比如堆、栈、写时拷贝的页面等

> Tips:
> 数据段在写过一次时就变成匿名页了，没有写过的是有文件背景的，与磁盘读出来的一致的

② file-backed page：

有文件背景的页面，指的是在磁盘里面有一个file文件，任何时候从磁盘读该文件，内核都会从内存页中给申请一个pagecache。然后将该文件先读到pagecache里面，这里pagecache是内存充当了硬盘的缓存功能，当file文件被读入pagecache时，下次再读的时候，就会非常快了。比如代码段。

读/写file文件的两种方式

① read/write：

- 当读文件时，内核先申请一片内存充当pagecache，接下来read会将pagecache拷贝到用户空间buffer。
- 当写文件时，是将用户空间的buffer内容拷贝到内核空间pagecache中。

② mmap：

- 如果想省去read/write的一次用户空间<==>内核空间内存拷贝过程，可以使用mmap直接把file文件映射成一个虚拟地址（位于0~3G），实质上是这个虚拟地址指向了内核申请的这片pagecache的。
- 接下来就可以直接用这个虚拟地址操作pagecache，内核知道这个pagecache与硬盘中file文件对应关系，所以，当应用程序用虚拟地址写完pagecache后实际相当于完成了硬盘写操作。

> Tips:
> 【1】进程代码段的本质 —— 将ELF文件text段mmap到用户空间0~3G的某段虚拟地址上（vma）；
> 【2】进程stack段（栈）—— 在ELF文件mmap之后生成的vma会带VM_GROWSDOWN标记，这样在PageFault函数中检测到VM_GROWSDOWN后会先调用expand_stack(vma, address)扩展vma，然后分配物理页，这就实现了栈随着使用的自动生长；（更进一步：对于进程process生成的线程thread而言，stak并不带增长标记，thread栈是在进程地址空间mmap出来的固定vma大小的，可以mmap时设置大小但不能生长）
> 【3】进程heap段（堆）—— 在ELF文件mmap之后生成vma并不带以上标记，因此需要用户态通过malloc触发brk/sbrk系统调用，扩展heap对应的vma区域然后才能在使用时触发PageFault分配物理页，实现堆生长；

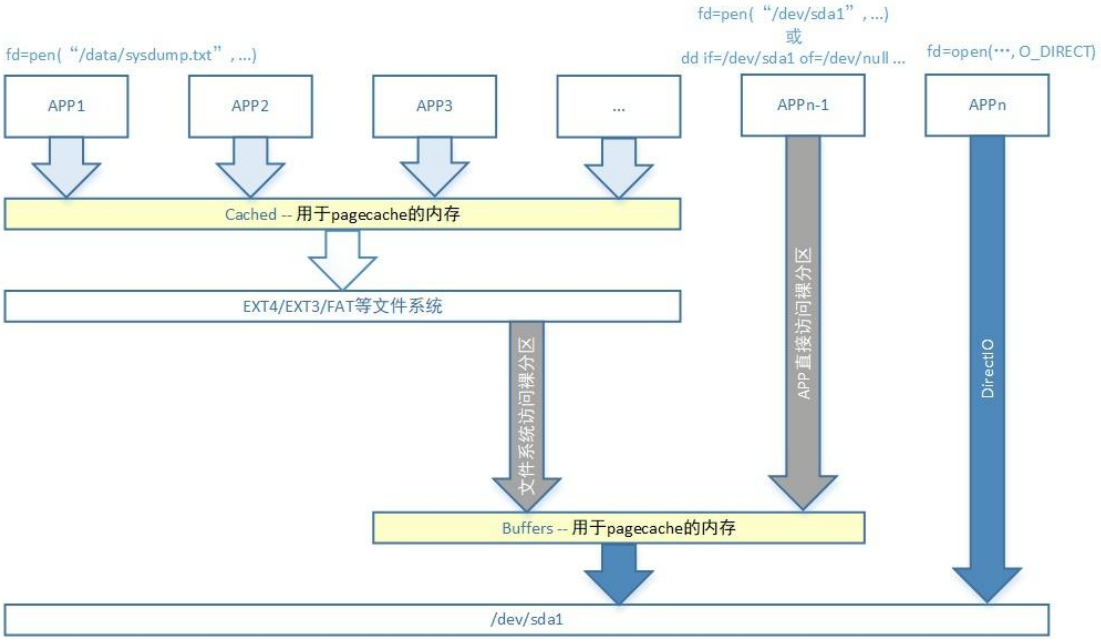
以上两种方式给我们的启示是:

- 对于应用程序而言，它所谓的写文件，只不过是将文件从用户态拷贝到pagecache/或者直接写到pagecache内。
  然后由内核将pagecache在合适的时间点同步到磁盘上，比如调用sync时/pagecache dirty page到达一定数量时等等。

- pagecache的存在能极大的提高系统的整体性能，当一个file被读过一次后，下次再读速度会直接命中pagecache，因此非常快。但是pagecache不能一直占用内存，会使用LRU（最近最少使用算法）来进行替换。

pagecache的两种形式

cached与buffers这两个都是文件系统缓存（pagecache），这两个没有本质区别，唯一的区别是“背景”不一样，如下图所示：



说明：

- 当通过ext4/ext3/fat等文件系统去访问file时以“文件系统为背景”，产生的pagecache就是属于cached
- 当直接访问/dev/sda1这种裸分区上file时以“裸分区为背景”，产生的pagecache就是属于buffers

> Tips：
> 只是通常情况下我们是不会访问裸分区的，裸分区的用户一般有2个：
> 【1】文件系统本身会访问裸分区；
> 【2】应用可能直接访问裸分区；

下面这幅图能很好的说明buffers与cached区别：

```
root@baohua-VirtualBox:~/develop/training/memory-courses/day4# sync
root@baohua-VirtualBox:~/develop/training/memory-courses/day4# echo 3 > /proc/sy
s/vm/drop_caches
root@baohua-VirtualBox:~/develop/training/memory-courses/day4# free
             total       used       free     shared    buffers     cached
Mem:       1024700     485868     538832       2816       2020      69916
-/+ buffers/cache:     413932     610768
Swap:            0          0          0
root@baohua-VirtualBox:~/develop/training/memory-courses/day4# cat /dev/sda1 > /
dev/null
^C
root@baohua-VirtualBox:~/develop/training/memory-courses/day4# free
             total       used       free     shared    buffers     cached
Mem:       1024700     873868     150832       2816     389060      69892
-/+ buffers/cache:     414916     609784
Swap:            0          0          0
```

说明：

- 先通过drop_caches节点回收不用的pagecache页面
- 然后用cat命令读一段时间的裸分区
- 对比cat命令前后两次free输出结果可以看到
  ① used增加free减少，这时大家基本都能猜测到的
  ② cached基本没有变化，说明此时读裸分区没有使用cached类型的pagecache
  ③ buffers的增加基本等于free的减少，说明读裸分区用的是buffers类型的pagecache
  ④ 包含buffers/cached在内的free基本没有变化

swap页交换

页交换通常指的是当物理内存不足时，内核会将一些不常用、不正在使用的页面暂时保存到磁盘上，以便腾出内存空间供更有需要的进程运行。通常需要被交换的页面就是上面我们提到的两种file-backed page和anonymous page。

① 有文件背景的页面(file-backed page)：

也即用于pagecache的页面，当Linux发现内存剩余量达到内存回收水位时会被swap到磁盘。

② 没有文件背景的页面（anonymous page）：

只能常驻内存，但Linux为了防止这些页面常驻内存会从磁盘分一个区“swap分区”或者在磁盘建立一个文件“swapfile文件”，把这个分区或者文件当作匿名页的“文件背景”，也即Linux给匿名页伪造了一个文件背景。比如创建一个swapfile命令：

```
dd if=/dev/zero of=/var/tmp/oomswap bs=1M count=32768
#or fallocate -l 32g /var/tmp/oomswap
chmod 600 /var/tmp/oomswap
mkswap /var/tmp/oomswap
swapon /var/tmp/oomswap
```

需要注意的是：

① 有文件背景的页面、匿名页都需要被交换，有文件背景的页面swap到磁盘文件，匿名页swap到swapfile（或swap分区）。

② 即便关闭内核中CONFIG_SWAP选项，kswapd线程也在工作，CONFIG_SWAP只控制匿名页swap，而有文件背景的页面swap是必须做的。

> Tips:
> kswapd线程功能：（内存回收）
> 【1】有文件背景页面的swap ==》 swap到 ==》 磁盘文件
> 【2】匿名页的swap ==》 swap到 ==》 swapfile文件/swap分区

zram透明压缩

zram会占用一部分内存，将自己模拟成一个硬盘分区作为swap分区，该分区自带“透明压缩功能”。

> Tips:
> 所谓“透明压缩”即：当将anonymous page交换写入该分区时，zram会“自动”把这些匿名页做压缩，而反之，当这些匿名页被应用程序继续访问时会再次触发PageFault（注意这个PageFault将会是Major PageFault），这时内核会从zram分区将这些匿名页“自动”解压出来放入内存。这个过程是不被察觉的，因此叫透明的。

这样zram以压缩形式做swap分区解决了2个问题：

- 增大内存的同时，以内存做swap分区，速度比磁盘快很多
- 增加磁盘寿命

但引入了1个问题：

- 牺牲了CPU使用率，因为需要CPU去做压缩和解压

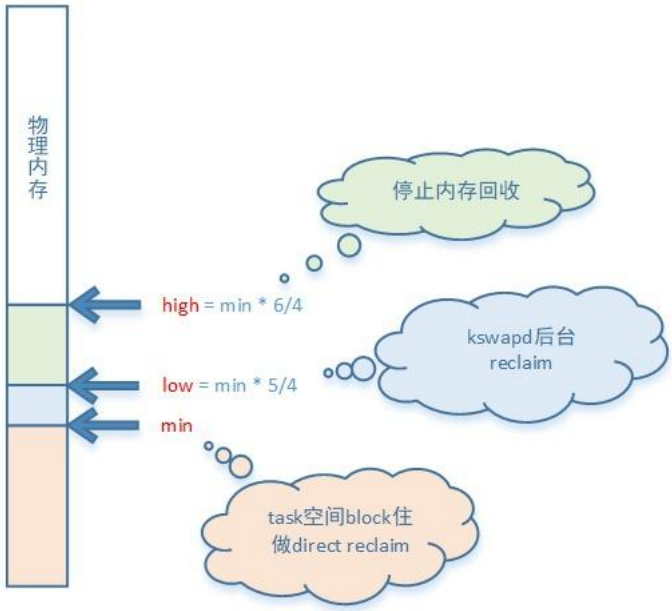
内存回收

回收水位

每个ZONE都会有自己的三个内存回收水位，都是根据min_free_kbytes值配置的：

- min —— 触发直接内存回收： 每个ZONE上的计算 min = (min_free_kbytes * (ZONE_Memory / Total_Memory))
- low —— 触发后台回收： 每个ZONE上计算 low = min * (5/4)
- high —— 停止后台回收： 每个ZONE上计算 high = min * (6/4)

默认情况下，min_free_kbytes值是根据memory总量开平方得来的，设置最小水线值min是因为内核必须要预留出来一些紧急内存，来给一些紧急任务利用ALLOC_NO_WATERMARKS参数/或者Pfmemalloc接口去申请使用，比如回收内存代码等。



回收方式

当物理内存低于low回收水位时内核就开启后台内存回收，低于min内核开始占用当前task做直接回收，到high水位内核停止后台回收。

即，内存回收有两种方式：

- 后台回收（kswapd）- Background reclaim
  指的是剩余物理内存值达到中间的low水位时kswapd线程开始启动回收，当剩余物理内存值达到high水位时kswapd线程停止回收。

- 直接回收 - Direct reclaim
  指的是剩余物理内存值达到最低的min水位时，内核内存分配代码会直接阻塞住当前task，直接在task代码路径上开始做内存回收，这个时间会比较久。

回收对象

内存回收会在pagecache和anonymous page两者同时做回收，两者回收的比例由swappiness值决定，它在节点/proc/sys/vm/swappiness：

- swappiness越大时(->100)倾向于回收anonymous page多一些 (=100积极回收匿名页，即使用swap分区/swapfile)
- swappiness越小时(->0)倾向于回收pagecache多一些 (=0积极将文件swap回磁盘，则仅在内存不足时才使用swap空间)

从上面可以看出该节点值范围是0~100，一般该节点默认会被配置成60，这只是一个经验值。

> Tips：
> 位于用户态、内核态内存分配器（非buddy）上中的未使用page，属于是anonymous page，他们也在回收之列，只不过他们的回收与它所在的内存分配器特点有一定的关系
> 比如slab page，在内存紧张时会做kmem_cache_shrink去压缩重排每个slab cache中的kmem_cache_node链表，释放掉一些没有在使用的空闲页

注意对于swappiness的调整是一个经验活儿，没有最佳值，看当前系统状态和性能要求：

- 如果系统中内存使用以anonymous page为主、同时实时性/性能要求又高，那么可以倾向于将swappiness设置的小一些，比如从60->40什么的去尝试
- 如果系统中内存使用以pagecache为主、同时实时性要求不高，那么可以尝试将swappiness设置大一些

回收算法

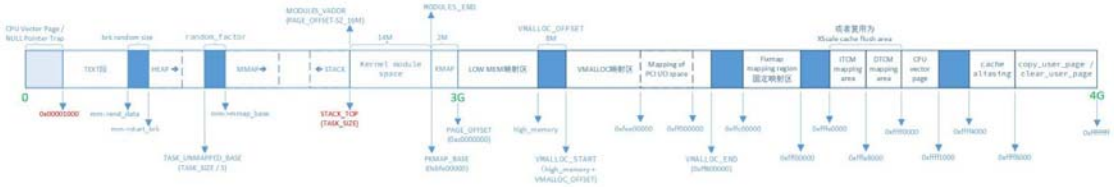
不管回收哪类page页，在做回收时内核都采用LRU算法，即最近最少使用算法，找到一个最不活跃的page去做回收。

进程内存统计

当评估一个进程内存消耗时，从来不说内核空间的，都是指的用户进程消耗的，对于进程调用系统调用使用了多少内存的情况也计算到内核空间，而不是算在用户进程头上。

进程虚拟地址空间分布

下图是32bit ARM平台上Linux的0~3G~4G的虚拟地址空间分布图：



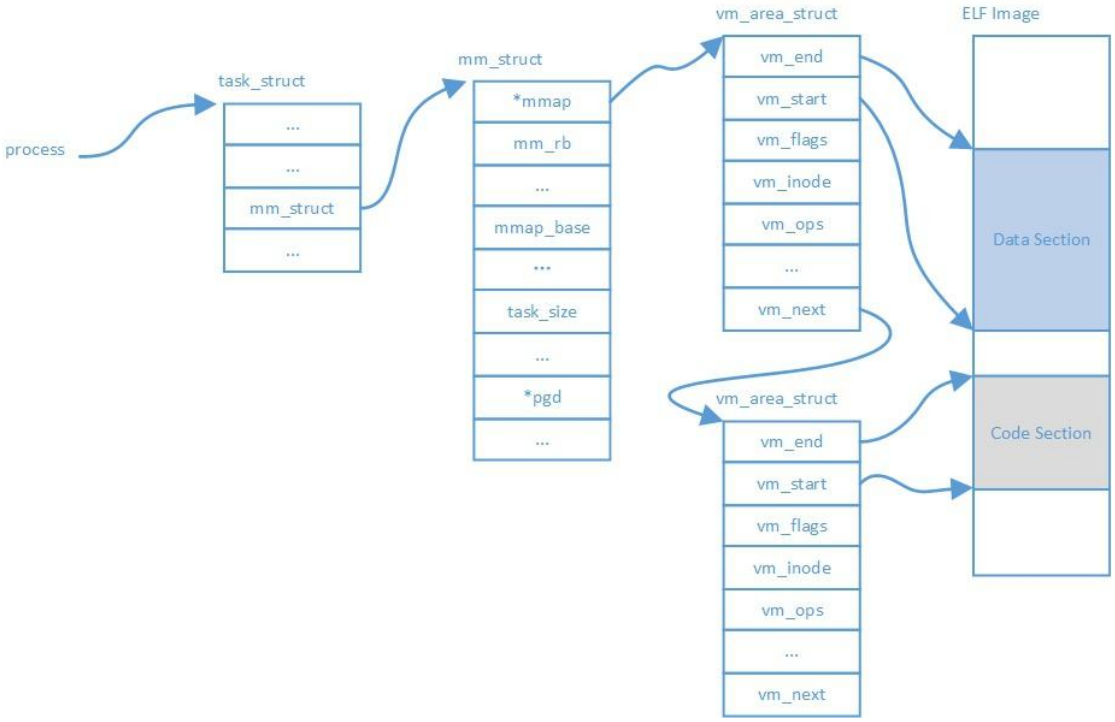
对于该图解释如下：

- 我们通常所说，用户空间0~3G，内核空间3G~4G。
- 用户空间进程真正使用的地址空间是上图0~3G内两个红色边界之间的部分，arm架构kmap和module映射会占用3G以下16M部分。
- 0x1000这个地址间距并非固定的，不同体系结构有不同值。arm平台上该值是0x1000。一般用户进程不会访问低于这个地址的空间，如果一定需要访问得自己实现open()和mmap()。
- 用户空间heap增加是有上限的，一直增加会踏到用户空间mmap映射区。
- 用户空间3G以下16M空间是给内核使用的module insmod映射区和kmap永久映射区。
- 固定映射区是指fix_to_virt() 转换而来的虚拟地址所在区域。
- TCM和XScale不一定存在，看CPU硬件平台；并且两个不会同时存在。
- PCI I/O space是一个在vmalloc映射区内固定专用区域。
- 在带MMU的32bit的ARM平台机器上vmalloc映射区域大小是有限制的，最小16M，最大984M，默认768M。

进程虚拟地址空间上的VMA

进程跑起来后，就是各种vma充斥在上图所示的用户空间0~3G段

下图是Linux用户进程虚拟地址空间vma结构：task_struct=>mm_struct=>vm_area_struct



其中mm_struct->mm_rb是所有vm_area_struct组成的红黑树，而mm_struct->mmap是所有vm_area_struct组成的链表，vma这个数据结构被双重管理主要是为了加速查找速度。

> Tips:
> 需要注意的是：无用户空间的纯内核线程task_struct->mm_struct==NULL

红黑树/链表上每个节点都表示该进程的一个vma虚拟地址空间，进程由多个这种vma虚拟地址空间组成，这些vma虚拟地址空间呈现出一段一段的离散分布在进程的0~3G虚拟地址空间上。例如：

pmap

```
acean acean ~ > LinuxKernel > linux-stable > mm > pmap 8741
8741:   bash
0000000000400000    976K r-x-- bash
0000000000f3000       4K r---- bash
0000000000f4000      36K rw--- bash
0000000000fd000      24K rw---   [ anon ]
0000000000f36000   2092K rw---   [ anon ]
00007f11e7adb000     44K r-x-- libnss_files-2.23.so
00007f11e7ae6000   2044K ----- libnss_files-2.23.so
00007f11e7ce5000      4K r---- libnss_files-2.23.so
00007f11e7ce6000      4K rw--- libnss_files-2.23.so
00007f11e7ce7000     24K rw---   [ anon ]
00007f11e7cf3000     44K r-x-- libnss_nis-2.23.so
00007f11e7cfe000   2044K ----- libnss_nis-2.23.so
00007f11e7efd000      4K r---- libnss_nis-2.23.so
00007f11e7efe000      4K rw--- libnss_nis-2.23.so
00007f11e7f03000     88K r-x-- libnsl-2.23.so
00007f11e7f19000   2044K ----- libnsl-2.23.so
00007f11e8118000      4K r---- libnsl-2.23.so
00007f11e8119000      4K rw--- libnsl-2.23.so
```

/proc/pid/smaps

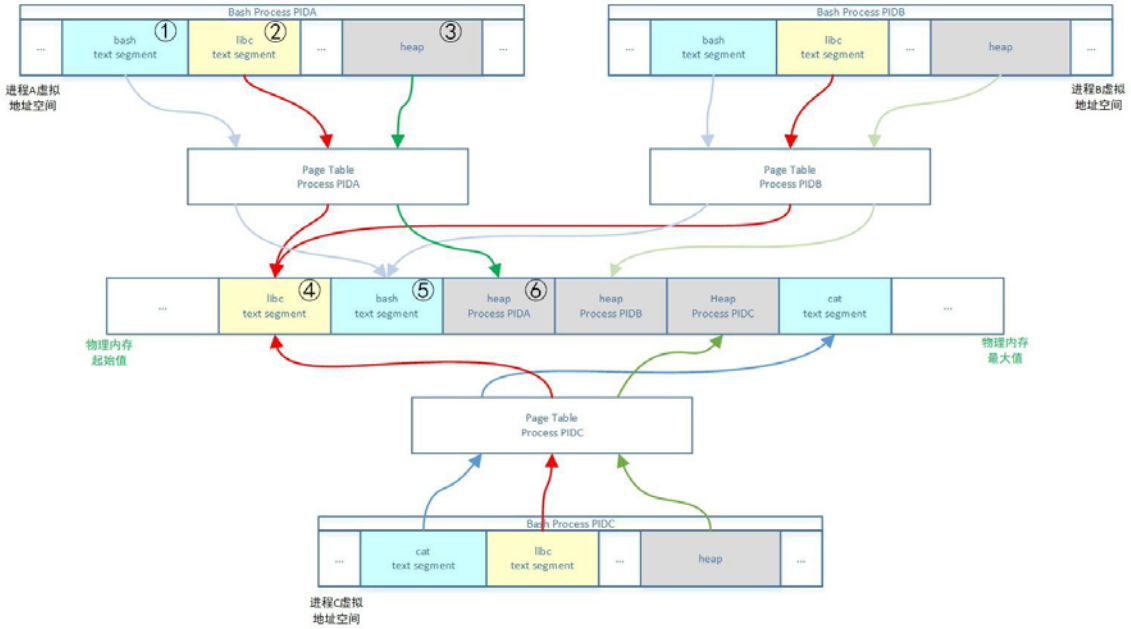
```
acean acean ~ > LinuxKernel > linux-stable > mm > cat /proc/8741/smaps
00400000-004f4000 r-xp 00000000 08:01 11010050                           /bin/bash
Size:                976 kB
Rss:                 888 kB
Pss:                 295 kB
Shared_Clean:        888 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:          888 kB
Anonymous:             0 kB
LazyFree:              0 kB
AnonHugePages:         0 kB
ShmemPmdMapped:        0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB
VmFlags: rd ex mr mw me dw sd
006f3000-006f4000 r--p 000f3000 08:01 11010050                           /bin/bash
Size:                  4 kB
Rss:                   4 kB
```

/proc/pid/maps

```
acean acean ~ > LinuxKernel > linux-stable > mm > cat /proc/8741/maps
00400000-004f4000 r-xp 00000000 08:01 11010050                           /bin/bash
006f3000-006f4000 r--p 000f3000 08:01 11010050                           /bin/bash
006f4000-006fd000 rw-p 000f4000 08:01 11010050                           /bin/bash
006fd000-00703000 rw-p 00000000 00:00 0
00f36000-01141000 rw-p 00000000 00:00 0                                  [heap]
7f11e7adb000-7f11e7ae6000 r-xp 00000000 08:01 7078604                    /lib/x86_64-linux-gnu/libnss_files-2.23.so
7f11e7ae6000-7f11e7ce5000 ---p 0000b000 08:01 7078604                    /lib/x86_64-linux-gnu/libnss_files-2.23.so
7f11e7ce5000-7f11e7ce6000 r--p 0000a000 08:01 7078604                    /lib/x86_64-linux-gnu/libnss_files-2.23.so
7f11e7ce6000-7f11e7ce7000 rw-p 0000b000 08:01 7078604                    /lib/x86_64-linux-gnu/libnss_files-2.23.so
7f11e7ce7000-7f11e7ced000 rw-p 00000000 00:00 0
7f11e7cf3000-7f11e7cfe000 r-xp 00000000 08:01 7078608                    /lib/x86_64-linux-gnu/libnss_nis-2.23.so
7f11e7cfe000-7f11e7efd000 ---p 0000b000 08:01 7078608                    /lib/x86_64-linux-gnu/libnss_nis-2.23.so
7f11e7efd000-7f11e7efe000 r--p 0000a000 08:01 7078608                    /lib/x86_64-linux-gnu/libnss_nis-2.23.so
7f11e7efe000-7f11e7eff000 rw-p 0000b000 08:01 7078608                    /lib/x86_64-linux-gnu/libnss_nis-2.23.so
7f11e7f03000-7f11e7f19000 r-xp 00000000 08:01 7078248                    /lib/x86_64-linux-gnu/libnsl-2.23.so
7f11e7f19000-7f11e8118000 ---p 00016000 08:01 7078248                    /lib/x86_64-linux-gnu/libnsl-2.23.so
7f11e8118000-7f11e8119000 r--p 00015000 08:01 7078248                    /lib/x86_64-linux-gnu/libnsl-2.23.so
7f11e8119000-7f11e811a000 rw-p 00016000 08:01 7078248                    /lib/x86_64-linux-gnu/libnsl-2.23.so
```

其中pmap、/proc/8741/smaps、/proc/8741/maps都能看出8741进程的vma是一段一段离散分布的。而这些没有用到的地址空间空白区域对于该进程都是非法的。

进程如何通过vma瓜分了物理内存，如下图显示：



关于这个图示说明如下：

- 进程A、B、C有独立的libc text段对应的vma，但是共享物理内存上vma映射的text段区域，因为是只读的
- 进程A、B有独立的bash text段对应的vma，但是共享物理内存上vma映射的text段区域，因为是只读的
- 进程A、B、C的heap段无论是vma还是映射到的物理内存都是独立的，因为是读写权限的
- 进程有自己独立的用户态页表，共享同一份内核态页表
- 进程VSS=①+②+③
- 进程RSS=④+⑤+⑥
- 进程PSS=④/3+⑤/2+⑥
- 进程USS=⑥

对于vma与物理内存关系需要澄清的是：

- 应用程序访问的虚拟内存virt_addr必须落在一个vma内
- virt_addr在vma内的不一定在物理内存有内容
    - 比如用户调用malloc是分配了vma，但是并没给实际物理内存
- virt_addr落在一个vma内的访问权限也不一定是正确的

- 需要看vma和pagetable记录的一软、一硬双重权限匹配

对于vma记录的页访问权限和page table记录的页访问权限关系：

- vma记录page访问权限 <=对应=> Linux对于虚拟内存的管理
- page table记录page访问权限 <=对应=> MMU对于物理页面管理
- 软硬件协同工作处理页面分配的过程大概是这样的：
  ① 进程读/写内存时，CPU通过MMU访问页表，发现访问权限不对
  ② MMU触发缺页异常PageFault给CPU处理
  ③ CPU收到PageFault后跳入异常处理流程，Linux内核调用PageFault处理handler
  ④ handler里面从硬件读取发生PageFault的地址和原因，并检查被访问页是否落入合法VMA、以及VMA中记录的权限是否正确
  ⑤ 如果被访问page没有落入合法VMA区间，内核干掉进程
  ⑥ 如果被访问page落入合法VMA区间内，但是VMA中记录的权限不对，内核干掉进程
  ⑦ 如果被访问page落入合法VMA区间内，并且VMA访问权限正确，handler会给进程申请新的一个物理页
  ⑧ 内核修改页表将虚拟地址指向新的物理地址

对于vma记录正确能够产生物理内存分配的PageFault分成如下2类：

- Major缺页：产生IO行为，从硬盘读取数据，比如加载代码段，处理时间长
- Minor缺页：不产生IO行为，比如去写mallo的内存，处理时间短
- Major缺页处理时间 >> Minor缺页处理时间

进程内存泄漏

内存泄漏讲的是程序运行的越久耗费的内存就越多，即分配和释放不是成对的。

内存泄漏的确认方法：连续多点采样法。

> Tips:
> 一般情况，如果确认内存泄漏，只需要判读USS就够了，主要就是看heap内存

内存泄漏的检测2个常用工具：valgrind、addresssanitizer

内存管理工程实践

DMA

DMA在对内存的操作上与CPU是对等的：

早些时候DMA是通过物理地址直接访问连续的物理内存（DMA ZONE / CMA）

有了IOMMU后DMA可以通过访问IOMMU转化的虚拟地址来使用不连续物理内存

Cache Coherent（缓存一致性）

dma cache的一致性有两个方向：

① 指CPU从cache读到的内容是旧的，而memory内容的是新被DMA改写的。即，memory更新而cache没有invalid。

② 指CPU写到cache的内容是新的，而DMA读到外设的memory内容是旧的。即，cache更新而没有flush到memory。

DMA API

dma cache一致性解决主要靠两类API：

① 一致性DMA缓冲区API：Coherent DMA buffers

原理是：驱动自己申请memory，然后映射成uncached的，即将PageTable中对应的页表项配置/修改成uncached属性。

- dma_alloc_coherent() 它返回给CPU使用的内存虚拟地址，它申请的dma内存会被映射到vmalloc映射区。（与vmalloc/ioremap类似）

```
1
2  dma_alloc_coherent()
3      => dma_alloc_attrs()
4          |=> ① dma_alloc_from_dev_coherent() 平台通用
```

```
5          |      |=> dev_get_coherent_memory()
6          |      |     => dev->dma_mem
7          |      |         <= dma_init_coherent_memory
8          |      |             <= rmem_dma_device_init
9          |      |                 <= rmem_dma_setup
10         |      |                    <= RESERVEDMEM_OF_DECLARE(dma, "shared-dma-pool", rmem_dma_setup); 解析dts中shared-dma-pool配置
11         |      |=> __dma_alloc_from_coherent(dev->dma_mem)
12         |=> 或者② get_dma_ops(dev)->alloc() 绑定到设备的，比如过IOMMU的、配置用CMA的等等
13               |=> ① if (dev && dev->dma_ops) dev->dma_ops;
14               |     |<= set_dma_ops
15               |     |<= ① if arm_get_iommu_dma_map_ops(coherent)有IOMMU的设备
16               |     |<= 或者② else arm_get_dma_map_ops(coherent)没有IOMMU的设备
17               |              |        |=> ① arm_dma_alloc()申请的页带cache的！，需要由软件/或硬件CCI做同步cache<=>memory
18               |              |              => __dma_alloc(coherent=false) 如下面：
19               |              |        |=> 或者② arm_coherent_dma_alloc()申请的页不带cache的！
20               |              |              => __dma_alloc(coherent=true)
21               |              |                 |=> ① if (cma) cma_allocator;从cma内存分配器分配，重建/修改页表
22               |              |                 |=> 或者② else if (is_coherent) simple_allocator;从buddy直接分配
23               |              |                 |=> 或者③ else if (allowblock) remap_allocator;从buddy直接分配映射到userspace使用，
24               |              |                 |=> 或者④ else pool_allocator;从预留的内存池范培
25               |              |
26               |              |<= arch_setup_dma_ops
27               |                  <= of_dma_configure 驱动设备解析dts配置
28               |=> 或者② else get_arch_dma_ops(dev ? dev->bus : NULL);
                       => arm_dma_ops;
```

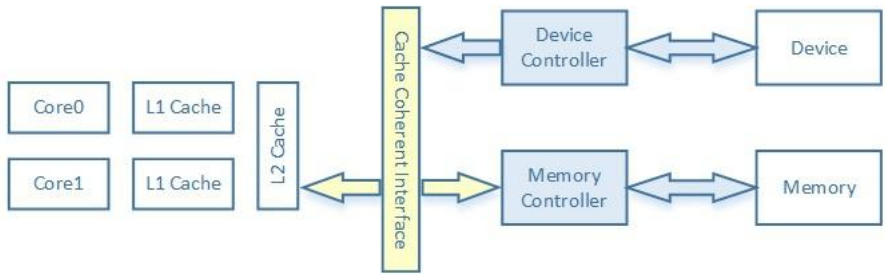
- dma_free_coherent()

对于这一对API的进一步说明：

该API是“前端接口”，“后端实现”是多种多样的，如下图

当SOC上没有硬件cache同步单元时，比如ARM CCI（cache coherent interface），那事情比较简单，我们通过这套API申请的dma buffer需要是非cache的；

当SOC上有硬件同步单元时，DMA引擎可以踏到cache/或者由硬件cache同步，那这套API的“后端实现”（比如以上cma_allocator()等函数就是后端）会被IC公司在自己“arch/arm/...”内重构实现，这样就可以去使用带cache的dma buffer；



② 流式DMA映射API：DMA Streaming Mapping

适用于做DMA的内存不是驱动自己分配的，而是上面传下来的地址，因为协议层或者应用层一般不了解该内存是否需要做DMA，这就造成了这样的内存一般都是带cache的。这时，每次在做DMA搬移之前需要内核中驱动程序首先调用dma_map_sg/或dma_man_single，并带一个参数表征是做memory=>device/还是device=>memory的DMA搬移，内核会首先做cache=>memory的flush/或者memory=>cache的invalid，然后才开始DMA搬移。选择sg或single主要看SOC上DMA IP是否支持聚集散列（非连续的 - sg）操作。

- dma_map_sg()
- dma_unmap_sg()
- dma_map_single()
- dma_unmap_single()

在做dma_unmap_sg/或者dma_unmap_single之前，即在以上每对API的两个函数之间，CPU是不允许再操作这片用于DMA的内存了，只允许DMA IP去操作。

DMA Memory

DMA物理内存特点：

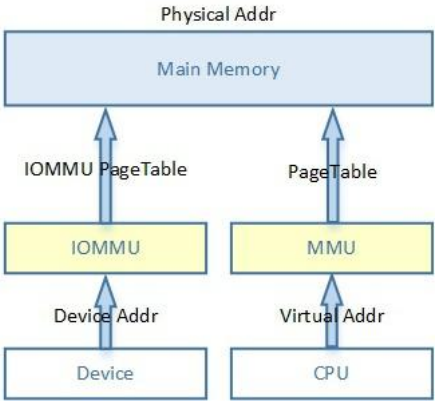
- 连续的--SOC无IOMMU；非连续的--SOC带IOMMU；
- 带cache的--SOC无CCI；不带cache--SOC带CCI；

DMA接口使用要领：

- 如果是驱动自己申请memory使用dma_alloc_coherent；
- 如果是上层传下来的memory使用sg/或者single；

IOMMU

当SOC上DMA过IOMMU时，DMA就不再需要申请连续的物理内存，IOMMU驱动可以把不连续的物理内存建立IOMMU Pagetable，然后通过IOMMU解析该页表去转换成连续的“虚拟内存”，这个虚拟内存就可以给DMA IP使用了。比如：Android内核中通过ION申请的内存。



CCI

TODO...

SMP

TODO...

CGROUP

非常有用的Linux内核资源管理特性，控制memory、cpu、io、tlb、net等等资源使用门限值、比率等等。

如下图显示各个cgroup子系统：

```
acean? /? sys? fs? cgroup? ls
blkio  cpu  cpuacct  cpu,cpuacct  cpuset  devices  freezer  hugetlb  memory  net_cls  net_cls,net_prio  net_prio  perf_event  pids  rdma  systemd
```

可以在cgroup的memory子系统中建立一个目录，cgroup就会自动在该目录内生成各种控制节点，这就相当于建立了一个“内存控制组”，通过修改该group中控制节点的属性值去控制加入该group的task对于memory使用，如下图：

```
acean? … fs? cgroup? memory? sudo mkdir acegroup
[sudo] password for acean:
acean? … fs? cgroup? memory? ls
acegroup              memory.force_empty            memory.kmem.tcp.max_usage_in_bytes  memory.oom_control          notify_on_release
cgroup.clone_children memory.kmem.failcnt          memory.kmem.tcp.usage_in_bytes      memory.pressure_level       release_agent
cgroup.event_control  memory.kmem.limit_in_bytes    memory.kmem.usage_in_bytes          memory.soft_limit_in_bytes  system.slice
cgroup.procs          memory.kmem.max_usage_in_bytes memory.limit_in_bytes               memory.stat                 tasks
cgroup.sane_behavior  memory.kmem.slabinfo          memory.max_usage_in_bytes           memory.swappiness           user.slice
init.scope            memory.kmem.tcp.failcnt       memory.move_charge_at_immigrate     memory.usage_in_bytes
memory.failcnt        memory.kmem.tcp.limit_in_bytes memory.numa_stat                   memory.use_hierarchy
acean? … fs? cgroup? memory? cd acegroup/
acean? … cgroup? memory? acegroup? ls
cgroup.clone_children memory.kmem.limit_in_bytes      memory.kmem.tcp.usage_in_bytes   memory.oom_control          memory.use_hierarchy
cgroup.event_control  memory.kmem.max_usage_in_bytes  memory.kmem.usage_in_bytes       memory.pressure_level       notify_on_release
cgroup.procs          memory.kmem.slabinfo           memory.limit_in_bytes            memory.soft_limit_in_bytes  tasks
memory.failcnt        memory.kmem.tcp.failcnt        memory.max_usage_in_bytes        memory.stat
memory.force_empty    memory.kmem.tcp.limit_in_bytes memory.move_charge_at_immigrate  memory.swappiness
memory.kmem.failcnt   memory.kmem.tcp.max_usage_in_bytes memory.numa_stat             memory.usage_in_bytes
```

cgroup与task数据结构关系是：task_struct->css_set->cgroup_subsys_state->cgroup

可以用如下图大概描述（从网上扣来借用的图^_^）

借助cgroup-tools命令集，将task加入一个cgroup组的方法有如下几种：

① task运行前，可以用gcexec去运行task来加入cgroup组：

```
1  cgexec -g memory:acegroup ./a.out
```

② task运行后，可以用cgclassify命令将task加入cgroup组：

```
cgclassify -g memory:acegroup 871
```

该工具集命令如下：

KSM

TODO...

RTCC

TODO...

COMPACTION

TODO...

Dirty Page Writeback

所谓脏页dirty page，即有文件背景的页面(file-backed page)被改写但未同步到磁盘上，即pagecache里面内容与磁盘不一致了。

脏页不能长期存在memory中，因为一是会占用内存，二是断电会导致数据丢失。那么内核会将脏页定期/不定期写回到磁盘上。

既然是脏页，那么可以想象一定有以下几种直观的属性：

- 脏页的时长 - 从pagecache被改写到被同步到磁盘这段持续的时间跨度
- 脏页的比例 - 未被写回的pagecache占总内存的比例
- 脏页的总量 - 未被写回的pagecache共用多大内存

内核就是利用以上“属性”特点来完成脏页写回机制：一是通过时间控制，二是通过空间控制：

① 时间： 目的是防止脏页在memory时间太久而导致数据丢失

- dirty_expire_centisecs - 当脏页存在时长超过该水线值时，会被内核flusher线程写回磁盘
- dirty_writeback_centisecs - 内核flusher线程会以这个时间为周期被唤醒，检查脏页时长，将所有“超标”的脏页写回磁盘

一般默认会被设置成30s，嵌入式上经常被设置为8s，设置的越短写磁盘的操作越密集，过于频繁会可能造成系统IO升高，性能下降；

② 空间： 目的是防止一次性写回对系统IO产生压力、以及对内存的压力

- dirty_backgroud_ratio - 当脏页占内存比例一旦达到该水线值，后台内核线程会开始进行脏页回写
- dirty_ratio - 当脏页占内存比例一旦达到该水线值，说明后台脏页回写速度已经不能满足，应用程序task会被直接block住，内核在该task进程空间开始做脏页回写

> Tips:
> 需要注意的是： 脏页写回 != 内存回收

DEBUG

记录内存管理相关的一些调试节点、命令、工具.

调试节点

节点	描述
/proc/buddyinfo	这个proc节点显示每个ZONE，同时显示该ZONE中从order=0到order=10的每个组当前空闲的页面数量

节点	描述
/proc/slabinfo	这个proc节点但显示当前kernel中所有的slab cache以及它们包含的slab、object等详细信息
/proc/vmallocinfo	这个proc节点显示内核所有vmalloc申请的映射区域、所有ioremap映射区域详细信息
/proc/pid/oom_score	这个proc节点显示该进程oom打分分数
/proc/pid/oom_score_adj	可以调整task打分因子，范围是-1000~1000，该值会被用于oom值的加减
/proc/pid/oom_adj	可以调整task打分因子，范围是-17~15，该值会被用于oom值乘除法，越大相当于乘以的分数越多，打分越高。这两个的换算关系：
——	① 当oom_adj = 15, 则oom_score_adj=1000;
——	② 当oom_adj < 15, 则oom_score_adj= oom_adj * 1000/17;
/proc/sys/vm/overcommit_memory	该节点是设置内核是否会检查剩余物理内存来判定申请内存是否可以成功。写0是检查，写1是不检查
/proc/pid/smaps	查看pid进程的vma虚拟内存分布情况，并且将每一段详细信息都输出来
/proc/pid/maps	查看pid进程的vma虚拟内存分布情况
/proc/sys/vm/drop_caches	用于回收不使用的pagecache页面
/proc/sys/vm/swappiness	用于调整内核做内存回收时对于pagecache和anonymous page的回收比例：
——	① swappiness大时回收anonymous page多一些
——	② swappiness小时回收pagecache多一些

> Tips:
> 对于/proc/sys/vm/drop_caches节点，下面英文描述很清楚：
> Writing to this file causes the kernel to drop clean caches, dentries and inodes from memory, causing that memory to become free.
> To free pagecache, use echo 1 > /proc/sys/vm/drop_caches; to free dentries and inodes, use echo 2 > /proc/sys/vm/drop_caches; to free pagecache, dentries and inodes, use echo 3 > /proc/sys/vm/drop_caches.
> Because this is a non-destructive operation and dirty objects are not freeable, the user should run sync first.

调试工具和命令

swap off

关闭交换分区，让匿名页不做交换。

pmap

查看pid进程的vma虚拟内存分布情况。命令是

```
pmap PID
```

time

查看某个程序执行的时间，有两种用法如下：

第一种命令如下，显示如下图

```
time ps
```

显示了该程序执行时实际时间、用户态时间、内核态时间

第二种命令如下，显示如下图

```
\time -v ps
```

这种方式显示的更为详细，包括时间都消耗在哪些方面了，比如下面Major PageFault发生了2095次

free

查看当前系统剩余内存情况，如下

pidof

查看某个进程名字是name的进程pid，比如

```
pidof firefox
```

slabtop

读slabinfo节点并显示的工具。显示效果如下

smem

用于查看一个进程耗费的内存，能列出USS、PSS、RSS如下

该工具在ubuntu环境下需要安装，它还能通过图形来显示进程内存占用情况，命令是：

```
smem --pie=command
```

显示如下图所示，看起来更直观一些：

valgrind

会让进程在一个虚拟机里面跑，会检测程序的所有内存行为，被检测程序不需要被重新编译即可。

但是缺点是该工具会将程序运行速度放慢二三十倍，有不少时候无法接受该工具的性能

```
valgrind --tool=memcheck --leak-check=yes ./a.out
```

AddressSanitizer

检测内存泄漏的另外一个工具是addresssannitizer（地址清洁剂），它里面有一个工具lsan（leaksanitizer）可以检测内存泄漏，可代码中直接调用lsan检测函数做手动检查。需要重新编译被检测程序。gcc4.9之后的版本才支持的。

```
gcc -g -fsanitize=address ./example.c
```

在Linue内核中，其实这个工具就是对应我们做kernel稳定性分析时常用的KASAN（Kernel Address Sanitizer），在内核中可以通过配置CONFIG_KASAN选项来开启。

getdelays

该工具作用是内核提供的测试进程时间消耗，在sched、IO、swap、reclaim各个点上的等待情况，适用于评估单个应用执行情况；

工具路径：/kernel/Documentation/accounting/getdelays.c

> Tips:
> 新版本linux内核已经将此类工具都修改到如下路径了/kernel/tools/
> 该工具在/kernel/tools/accounting/下面

工具使用需要开启内核选项:

- CONFIG_TASK_DELAY_ACCT = y
- CONFIG_TASKSTATS = y

假设将getdelays.c编译成了可执行文件getdelays，那么命令如下：

```
./getdelays -d -c ./a.out
```

生成类似如下样子的报告：

包含了CPU调度、IO、SWAP、内存回收4种行为发生的次数和总时长、平均每次时长

vmstat

查看当前系统中CPU、IO、MEM、SWAP等使用情况

REFERENCE

宋宝华内存管理一

宋宝华内存管理二

宋宝华内存管理三

宋宝华内存管理四

宋宝华内存管理五

宋宝华内存管理六

Kernel ARM Memory Layout

Kernel ARM64 Memory Layout

ARM64 Linux内核空间的内存布局图

Linux虚拟地址空间布局

高端内存映射之kmap持久内核映射

ARM平台下elf文件超详细的分析与解读

Intel线性地址逻辑地址和虚拟地址

内核PageFault处理流程

内核地址申请不能发生PageFault

Memory Management

Spectre/Meltdown演义--通俗篇

宋宝华：meltdown漏洞五分钟视频（原理与案例）

郭建：Linux内存模型——平坦、非连续与稀疏

#kernel #memory @3000-01-01 00:00

发表评论

撰写评论

super.zed reply
2019-06-22 18:05:32

dma_alloc_coherent()

=> dma_alloc_attrs()

|=> ① dma_alloc_from_dev_coherent() 平台通用

| |=> dev_get_coherent_memory()

| | => dev->dma_mem

| | <= dma_init_coherent_memory

| | <= rmem_dma_device_init

| | <= rmem_dma_setup

| | <= RESERVEDMEM_OF_DECLARE(dma, "shared-dma-pool", rmem_dma_setup); 解析dts中shared-dma-pool配置（一般该路径很少使用了）

| |=> __dma_alloc_from_coherent(dev->dma_mem)

|=> 或者② get_dma_ops(dev)->alloc() 绑定到设备的，比如过IOMMU的、配置用CMA的等等

楼主，请问这种格式的call tree是什么工具生成的么？我猜想应该不是手工画的。不过我看cflow或者egypt出来的都不是这种形式。请指教。

ace reply
2019-09-05 09:03:26

@super.zed 这个格式显然不是工具整理的calltree，整理成这种形式只是为了比较直观，用cflow或其它工具生成后自己再手动整理一下

firo reply
2019-09-01 19:43:45

请问作者整理这些内容花了多少时间?

ace reply
2019-09-05 09:04:36

@firo 最初整理倒是没有花费太久，一个多小时。后面慢慢补充了几次。

Powered by Bitcron