

内存管理策略

背景

基本硬件	内存由一个很大的字节数组来组成，每个字节都有各自的地址。CPU 根据程序计数器的值从内存中提取指令，这些指令可能引起对特定内存地址的额外加载与存储	
	CPU 可以直接访问的通用存储只有内存和处理单元内置的寄存器。机器指令可以用内存地址作为参数，而不能用磁盘地址作为参数。	CPU 内置寄存器通常可以在一个 CPU 时钟周期内完成访问。
	为了系统操作的正确，我们应保护操作系统，不被用户进程访问。在多用户系统上，我们还应保护用户进程不会互相影响。	完成内存的访问可能需要多个 CPU 时钟周期。在这种情况下，由于没有数据以便完成正在执行的指令，CPU 通常需要暂停
		补救措施是在 CPU 与内存之间，通常是在 CPU 芯片上，增加更快的内存；这称为高速缓存
地址绑定	根据采用的内存管理，进程在执行时可以在磁盘和内存之间移动。在磁盘上等待调到内存以便执行的进程形成了输入队列	我们需要确保每个进程都有一个单独的内存空间。单独的进程内存空间可以保护进程而不互相影响，
	源程序中的地址通常是用符号表示（如变量 count）。编译器通常将这些符号地址绑定到可重定位的地址	基址寄存器 含有最小的合法的物理内存地址，而界限地址寄存器 指定了范围的大小。
逻辑地址空间与物理地址空间	CPU 生成的地址通常称为逻辑地址，而内存单元看到的地址（即加载到内存地址寄存器的地址）通常称为物理地址	内存空间保护的实现是通过 CPU 硬件对对用户模式下产生的地址与寄存器的地址进行比较来完成的。
	由程序所生成的所有逻辑地址的集合称为逻辑地址空间，这些逻辑地址对应的所有物理地址的集合称为物理地址空间。	
动态加载	从虚拟地址到物理地址的运行时刻映射是由内存管理单元（MMU）的硬件设备来完成的。	
	进程的大小受限于内存的大小。为了获得更好的内存空间利用率，可以使用动态加载。采用动态加载时，一个程序只有在调用时才会加载。所有程序都可以可重定位加载格式保存在磁盘上。主程序被加载到内存，并执行。	
动态链接与共享库	动态链接的优点是，只有一个程序被需要时，它才会被加载。当大多数代码需要用来处理异常情况时，如错误处理，这种方法特别有用。	
	动态链接库 为系统库，可链接到用户程序，以便运行。有的操作系统只支持静态链接，它的系统库与其他目标模块一样，通过加载程序，被合并到二进制程序映像。动态链接类似于动态加载。这里，不是加载而是链接，会延迟到运行时。	

交换

标准交换	进程必须在内存中以便执行。不过，进程可以暂时从内存交换 到备份存储，当再次执行时再调回到内存中	
	标准交换在内存与备份存储之间移动进程。备份存储通常是快速磁盘它应足够大，以容纳所有用户的所有内存映像的副本；并且它应提供对这些存储器映像的直接访问。系统维护一个可运行的所有进程的就绪队列，它们的映像备份在备份存储或内存中。	
	当 CPU 调度器决定要执行一个进程时，它调用分派器。分派器检查队列中的下一个进程是否在内存中。如果不在，并且没有空闲内存区域，那么分派器会换出当前位于内存中的一个进程，并换入所需进程。然后，重新加载寄存器，并将控制权转移到所选进程。	请注意，交换时间的主要部分是传输时间。总的传输时间与交换的内存大小成正比。
移动系统的交换	这种交换系统的上下文切换时间相当高。	交换也受到其他因素的约束。如果我们想要换出一个进程，那么应确保该进程是完全处于空闲的。特别关注的是任何等待 I/O。
	移动设备通常采用闪存，而不是空间更大的硬盘作为它的永久存储U 导致的空间约束是移动操作系统设计者避免交换的原因之一。其他原因包括：闪存写入人次数的限制以及内存与闪存之间的吞吐量差。	
	当空闲内存降低到一定阈值以下时，苹果的 iOS，不是采用交换，而是要求应用程序自愿放弃分配的内存。	

连续内存分配

内存保护	内存通常分为两个区域：一个用于驻留操作系统，另一个用于用户进程。操作系统可以放在低内存，也可放在高内存，影响这一决定的主要因素是中断向量的位置	
	在采用连续内存分配时，每个进程位于一个连续的内存区域，与包含下一个进程的内存相连。	
	如果一个系统有重定位寄存器和界限寄存器，则可以防止进程访问不属于它的内存	
	每个逻辑地址应在界限寄存器规定的范围内。MMU 通过动态地将逻辑地址加上重定位寄存器的值，来进行映射。映射后的地址再发送到内存	
内存分配	当 CPU 调度器选择一个进程来执行时，作为上下文切换工作的一部分，分派器会用正确的值来加载重定位寄存器和界限寄存器。	
	重定位寄存器方案提供了一种有效方式，以便允许操作系统动态改变大小。许多情况都需要这一灵活性。	
	现在我们讨论内存分配。最为简单的内存分配方法之一，就是将内存分为多个固定大小的分区。每个分区可以只包含一个进程。因此，多道程序的程度受限于分区数。	
	对于可变分区方案，操作系统有一个表，用于记录哪些内存可用和哪些内存已用。所有内存都可用于用户进程，因此可以作为一大块的可用内存，称为孔。	
碎片	通用动态存储分配问题(根据一组空闲孔来分配大小为n 的请求)的一个特例，这个问题有许多解决方法。从一组可用孔中选择一个空闲孔的最为常用方法包括：	首次适应：分配首个足够大的孔。查找可以从头开始，也可以从上次首次适应结束时开始。一旦找到足够大的空闲孔，就可以停止。
	用于内存分配的首次适应和最优适应算法都有外部碎片的问题。随着进程加载到内存和从内存退出，空闲内存空间被分为小的片段。当总的可用内存之和可以满足请求但并不连续时，这就出现了外部碎片问题：存储被分成了大量的小孔。	最优适应：分配最小的足够大的孔。应查找整个列表，除非列表按大小排序。这种方法可以产生最小剩余孔。
	根据内存空间总的大小和平均进程大小的不同，外部碎片问题或许次要或许重要。	最差适应：分配最大的孔。同样，应查找整个列表，除非列表按大小排序。这种方法可以产生最大剩余孔，该孔可能比最优适应产生的较小剩余孔更为适用。
	内存碎片可以是内部的，也可以是外部的。进程所分配的内存可能比所需的要大。这两个数字之差称为内部碎片，这部分内存	
	外部碎片问题的一种解决方法是紧缩。它的目的是移动内存内容，以便将所有空闲空间合并成一块。	选择首次适应者或最优适应，可能会影响碎片数量。
	外部碎片化问题的另一个可能的解决方案是：允许进程的逻辑地址空间是不连续的；这样，只要有物理内存可用，就允许为进程分配内存。	另一因素是从空闲块的哪端开始分配。

分段

基本方法	分段 就是支持这种用户视图的内存管理方案。逻辑地址空间是由一组 段构成。每个段都有名称和长度。地址指定了段名称和段内偏移。
分段硬件	映射用户定义的二维地址到一维物理地址。这个地址是通过段表 来实现的。段表的每个条目都有段基址和段界限。段基址地址包含该段在内存中的开始物理地址，而段界限指定该段的长度

分段允许进程的物理地址空间是非连续的。分页是提供这种优势的另一种内存管理方案。然而，分页避免了外部碎片和紧缩，而分段不可以。分页也避免了将不同大小的内存块匹配到交换空间的麻烦问题。

分页

基本方法	实现分页的基本方法涉及将物理内存分为固定大小的块，称为帧或页帧；而将逻辑内存也分为同样大小的块，称为页或页面。当需要执行一个进程时，它的页从文件系统或备份存储等源处，加载到内存的可用帧。	
	由 CPU 生成的每个地址分为两部分：页码和页偏移 页码作为页表的索引。页表包含每页所在物理内存的基址。这个基址与页偏移的组合就形成了物理内存地址，可发送到物理单元。	
	页表的硬件实现有多种方法。最为简单的一种方法是，将页表作为一组专用的寄存器来实现。这些寄存器应用高速逻辑电路来构造，以高效地进行分页地址的转换。由于每次访问内存都要经过分页映射，因此效率是一个重要的考虑因素。	如果页表比较小（例如 256 个条目），那么页表使用寄存器还是令人满意的。
	对于允许页表非常大的这些机器，采用快速寄存器来实现页表就不行了。因而需要将页表放在内存中，并将页表基址寄存器指向页表。改变页表只需要改变这一寄存器就可以，这也大大降低了上下文切换的时间	采用这种方法的问题是访问用户内存位置的所需时间。如果需要访问位置 / 那么应首先利用 PTBR 的值，再加上 / 的页码作为偏移，来查找页表。这一任务需要内存访问。访问一个字节需要两次内存访问（一次用于页表条目，一次用于字节）
硬件支持	这个问题的标准解决方案是采用专用的、小的、查找快速的高速硬件缓冲，它称为转换表缓冲（TLB）。TLB 是关联的高速内存。TLB 条目由两部分组成：键（标签）和值。	在 TLB 中查找到感兴趣页码的次数的百分比称为命中率
	用一个位可以定义一个页是可读可写或只可读。每次内存引用都要通过页表，来查找正确的帧码；在计算物理地址的同时，可以通过检查保护位来验证有没有对只可读页进行操作。	
保护	分页环境下的内存保护是通过与每个帧关联的保护位来实现的。通常，这些位保存在页表中。	还有一个位通常与页表中的每一条目相关联：有效无效位。当该位为有效时，该值表示相关的页在进程的逻辑地址空间内，因此是合法（或有效）的页
	如果代码是可重入代码或纯代码，则可以共享。可重入代码是不能自我修改的代码：它在执行期间不会改变。因此，两个或更多个进程可以同时执行相同代码。每个进程都有它自己的寄存器副本和数据存储，以便保存进程执行的数据。当然，两个不同进程的数据不同。	
共享页		

页表结构

分层分页	不想在内存中连续地分配这个页表。这个问题的一个简单解决方法是将页表划分为更小的块。完成这种划分有多种方法。	一种方法是使用两层分页算法，就是将页表再分页
		对于 64 位的逻辑地址空间的系统，两层分页方案就不再适合。为了说明这一点，假设系统的页面大小为 4KB（212）。这时，页表可由多达 252 个条目组成。如果采用两层分页，那么内部页表可方便地定为一页长
哈希页表	处理大于 32 位地址空间的常用方法是使用哈希页表。采用虚拟页码作为哈希值。哈希页表的每一个条目都包括一个链表，该链表的元素哈希到同一位置（该链表用来解决处理碰撞）。每个元素由三个字段组成：1）虚拟页码，2）映射的帧码，3）指向链表内下一个元素的指针	外部页表的划分有很多方法。例如，我们可以对外部页表再分页，进而得到三层分页方案。
	因为进程是通过虚拟地址来引用页的。操作系统应将这种引用转换成物理内存的地址。由于页表是按虚拟地址排序的，操作系统可计算出所对应条目在页表中的位置，可以直接使用该值	
倒置页表	为了解决这个问题，我们可以使用倒置页表 对于每个真正的内存页或帧，倒置页表才有一个条目。每个条目包含保存在真正内存位置上的页的虚拟地址，以及拥有该页进程的信息。	这种方法的缺点之一是，每个页表可能包含数以百万计的条目。这些表可能需要大量的物理内存，以跟踪其他物理内存是如何使用的。
		因此，整个系统只有一个页表，并且每个物理内存的页只有一条相应的条目