6.4 页帧分配和系统抖动

1. 最小帧数

在为进程分配帧时，首先应该考虑的问题是：能保证进程能正常运行所需的最少帧数（也称为最小物理块数）。若系统为某进程所分配的帧数少于此值时，进程将无法正常运行。

分配不少于最少数量帧的原因之一是性能。显然，随着分配给每个进程的帧数量的减少，缺页会增加，从而减慢进程的执行。另外，当在指令完成之前出现缺页时，该指令必须重新执行。因此，必须有足够的帧来容纳所有单个指令所引用的页。

例如，考虑这样一个机器，其所有机器指令只有一个内存地址。因此，至少需要一帧用于指令，另一帧用于内存引用。另外，如果允许一级间接引用(例如，一条在 16 页上的 load 指令引用了 0 页上的地址，而这个地址又间接引用了第 23 页)，那么每个进程至少需要 3 个帧。帧的最少数量是由给定计算机结构定义的。例如，PDP-11 的移动指令的长度在一些寻址模式下为多个字长，因此指令本身可能跨在 2 个页上。另外，它有 2 个操作数，而每个操作数都可能是间接引用，因此，共需要 6 个帧。

每个进程帧的最少数量是由体系结构决定的，而最大数量是由可用物理内存的数量来决定的。在这两者之间选择合理的帧数是操作系统重要的话题。

2. 帧分配算法

在 n 个进程之间分配 m 个帧的最为容易的方法是给每个一个平均值，即 m/n 帧。例如，有 93 个帧和 5 个进程，那么每个进程可得到 18 个帧，剩余 3 个帧可以放在空闲帧缓存池中。这种方案称为平均分配(equal allocation)，它是固定分配策略里最简单的一种。

另外一种方法是要认识到各个进程需要不同数量的内存。考虑一下这样的系统，其帧大小为 1KB。如果只有两个进程运行在系统上，且空闲帧数为 62，一个进程为具有 10KB 的学生进程，另一个进程为具有 127KB 的交互数据库，那么给每个进程各 31 个进程帧就没有道理了。学生进程所需要的帧不超过 10 个，因此其他 21 帧就完全浪费了。为了解决这个问题，可使用比例分配(proportional allocation)。根据进程大小，而将可用内存分配给每个进程。设进程 p_i 的虚拟内存大小为 s_i，且定义

$$S = \sum s_i$$

这样，如果可用帧的总数为 m，那么进程 p_i 可分配到 a_i 个帧，这里 a_i 近似为

$$a_i = \frac{s_i}{S} \times m$$

当然，必须调整 a_i 使之成为整数且大于指令集合所需要的帧的最少数量，并使所有帧不超过 m。

对于平均和比例分配，每个进程所分配的数量会随着多道程序的级别而有所变化。如果多道程序程度增加，那么每个进程会失去一些帧来提供给新进程使用。另一方面，如果多道程序程度降低，那么原来分配给离开进程的帧可以分配给剩余进程。

对于平均或比例分配，高优先级进程与低优先级进程一样处理。然而，根据定义，可能要给高优先级进程更多内存以加快其执行，这样就会损害到低优先级进程。所以另一个方法是使用比例分配的策略，但是不根据进程相对大小，而是根据进程优先级，或大小和优先级的组合：把内存中可供分配的所有帧分成两部分：一部分按比例分配给各进程；另一部分则根据各进程的优先级或者大小与优先级的组合，适当地增加其相应份额后，分配给各进程。

与固定分配相对的则是可变分配。可变分配中，各个进程的帧数并不是一开始就确定好，

操作系统本身会保持一个空闲帧队列，在进程缺页时从空闲帧队列里取出来分配给进程。当空闲帧队列里的帧用完时，则从内存中选择一些页调出。在整个进程的运行中，帧的数量始终不是固定的，而是由操作系统管理的。

3. 帧的置换策略

为各个进程分配帧的另一个重要因素是页置换。当有多个进程竞争帧时，可将页置换算法分为两大类：全局置换(global replacement) 和局部置换(local replacement) 。全局置换允许一个进程从所有帧集合中选择一个置换帧，而不管该帧是否已分配给其他进程，即一个进程可以从另一个进程中拿到帧。局部置换要求每个进程仅从其自己的分配帧中进行选择。

全局置换算法的一个问题是进程不能控制其缺页率。一个进程的位于内存的页集合不但取决于进程本身的调页行为，还取决于其他进程的调页行为。因此，相同进程由于外部环境不同，可能执行很不一样(有的执行可能需要 0.5 秒，而有的执行可能需要 10 秒)。局部置换算法就没有这样的问题。在局部置换下，进程内存中的页只受该进程本身的调页行为所影响。但是，因为局部置换不能使用其他进程的不常用的内存，所以会阻碍一个进程。因此，全局置换通常会有更好的系统吞吐量，且更为常用。

根据帧的分配策略策略和置换策略可组合成以下三种策略：

(1) 固定分配局部置换策略：它基于进程的类型，或根据程序员、系统管理员的建议，为每个进程分配一固定帧的内存空间，在整个运行期间都不再改变。如果进程在运行中发现缺页，则只能从该进程在内存的固定帧中选出一页换出，然后再调入另一页，保证分配给该进程的内存空间不变。

(2) 可变分配全局置换策略：系统为每个进程分配一定数目的帧，而 OS 本身也保持一个空闲帧队列。当某进程发现缺页时，由系统从空闲帧队列中，取出一帧分配给该进程，并将欲调入的缺页装入其中。当空闲帧队列中的帧用完时，OS 才能从内存中选择一页调出，该页可能是系统中任一进程的页。

(3) 可变分配局部置换：根据进程的类型或程序员的要求，为每个进程分配一定数目的内存空间；但当某进程发生缺页时，只允许从该进程在内存的页面中选出一页换出，而不影响其它进程的运行。

4. 系统颠簸现象

如果低优先级进程所分配的帧数量少于计算机体系结构所要求的最少数量，那么必须暂停进程执行。接着应换出其他所有剩余页，以便使其所有分配的帧空闲。事实上，考虑那些没有"足够"帧的进程，如果进程没有它所需要的活跃使用的帧，那么它会很快产生页错误。这时，必须置换某个页。然而，其所有页都在使用，它置换一个页，但又立刻再次需要这个页。因此，它会一而再地产生页错误，置换一个页，而该页又立即出错且需要立即调进来。这种频繁的页调度行为称为颠簸(thrashing)。如果一个进程在换页上用的时间要多于执行时间，那么这个进程就在颠簸。

颠簸将导致严重的性能问题。考虑如下情况，这是基于早期调页系统的真实行为。操作系统在监视 CPU 的使用率。如果 CPU 使用率太低，那么向系统中引入新进程，以增加多道程序的程度。采用全局置换算法，它会置换页而不管这些页是属于哪个进程的。现在假设一个进程进入一个新执行阶段，需要更多的帧。它开始出现缺页，并从其他进程中拿到帧。然而，这些进程也需要这些页，所以它们也会出现缺页，从而从其他进程中拿到帧。这些缺页进程必须使用调页设备以换进和换出页。随着它们排队等待换页设备，就绪队列会变空，而进程等待调页设备，CPU 使用率就会降低。

CPU 调度程序发现 CPU 使用率降低，因此会在增加多道程序的程度。新进程试图从其

他运行进程中拿到帧，从而引起更多缺页，形成更长的调页设备的队列。因此，CPU 使用率进一步降低，CPU 调度程序试图再增加多道程序的程度。这样就出现了系统颠簸，系统吞吐量陡降，缺页显著增加。因此，有效内存访问时间增加。最终因为进程主要忙于调页，系统不能完成一件工作。

这种现象如图 6.17 所示，显示了 CPU 使用率与多道程序程度的关系。随着多道程序程度增加，CPU 使用率(虽然有点慢)增加，直到达到最大值。如果多道程序的程度还要继续增加，那么系统颠簸就开始了，且 CPU 使用率急剧下降。这时，为了增加 CPU 使用率和降低系统颠簸，必须降低多道程序的程度。

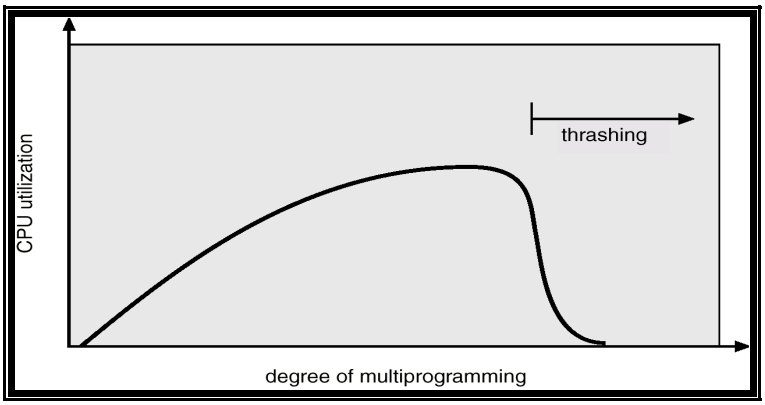

图 6.17 系统颠簸

通过局部置换算法(local replacement algorithm) (或优先置换算法，priority replacement algorithm)能限制系统颠簸。采用局部置换，如果一个进程开始颠簸，那么它不能从其他进程拿到帧，且不能使后者也颠簸。然而这个问题还没有完全得到解决。如果进程颠簸，那么绝大多数时间内也会排队来等待调页设备。由于调页设备的更长的平均队列，缺页的平均等待时间也会增加。因此，即使对没有颠簸的进程，其有效访问时间也会增加。为了防止颠簸，可以选择挂起若干进程，或者必须提供进程所需的足够多的帧，使得产生缺页的平均时间等于系统处理进程缺页的平均时间。如何知道进程"需要"多少帧，在这里可以让 CPU 调度引入工作集算法，它研究一个进程实际正在使用多少帧。这种方法定义了进程执行的局部模型(locality model)。局部模型的原理是缓存的基础。如果对任何数据类型的访问是随机的而没有一定的模式，那么缓存就没有用了。

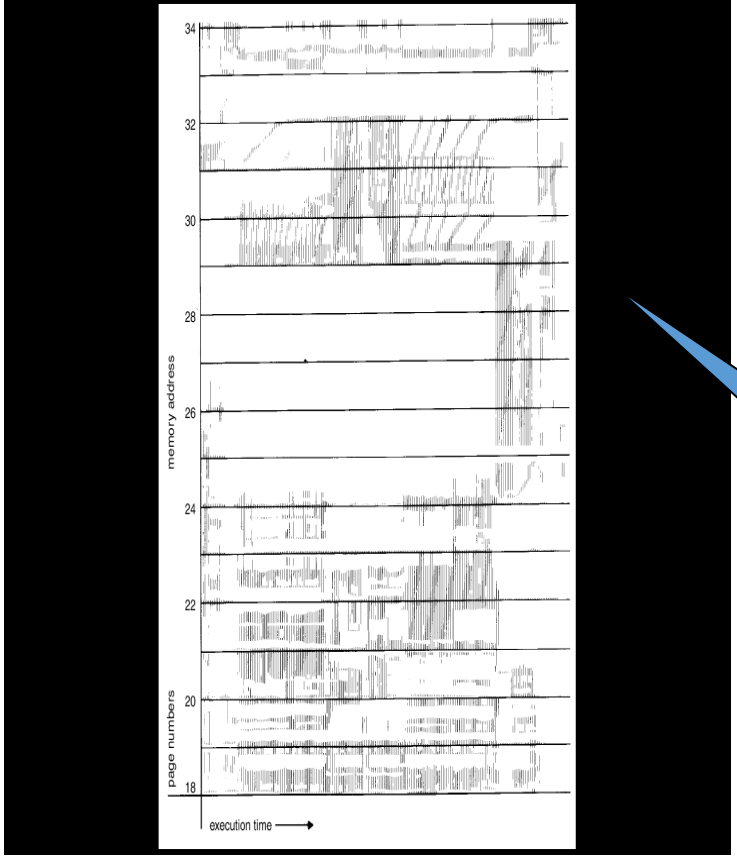

图 6.18 内存引用模式中的局部性

从图 6.18 可以看到，进程执行时访问的内存具有一定的时间局部性和空间局部性。局部模型说明，当进程执行时，它从一个局部移向另一个局部。局部是一个经常使用页的集合。一个程序通常由多个不同局部组成，它们可能重叠。局部是由程序结构和数据结构来定义的。局部模型说明了所有程序都具有这种基本的内存引用结构。

假设为每个进程都分配了可以满足其当前局部的帧。该进程在其局部内会出现缺页，直到所有页均在内存中：接着它不再会出现缺页直到它改变局部为止。如果分配的帧数少于现有局部的大小，那么进程会颠簸，这是因为它不能将所有经常使用的页放在内存中。

5. 工作集模型

工作集模型(working-set model)是基于局部性假设的。该模型使用参数Δ定义工作集窗口(working-set window)。其思想是检查最近Δ个页的引用。这最近Δ个引用的页集合称为工作集(working set)(如图 6.19 所示)。如果一个页正在使用中，那么它就在工作集内。如果它不再使用，那么它会在其上次引用的Δ时间单位后从工作集中删除。因此，工作集是程序局部的近似。

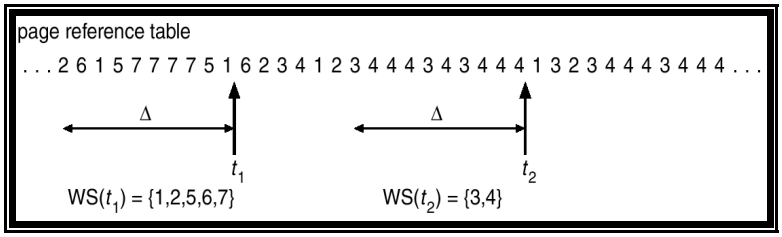

图 6.19 工作集模型

例如，对于如图 9.20 所示的内存引用序列，如果Δ为 10 个内存引用，那么 t_1 时的工作集为{1, 2, 5, 6, 7}。到 t_2 时，工作集则为{3, 4}。

工作集的精确度与Δ的选择有关。如果Δ太小，那么它不能包含整个局部；如果Δ太大，那么它可能包含多个局部。在最为极端的情况下，如果Δ为无穷大，那么工作集为进程执行所

接触到的所有页的集合。

最为重要的工作集的属性是其大小。如果经计算而得到系统内每个进程的工作集为 WSS_i，那么就得到

$D=\Sigma WSS_i$

其中 D 为总的帧需求量。每个进程都经常要使用位于其工作集内的页。因此，进程 i 需要 WSS_i 帧。如果总的需求大于可用帧的数量(D>m)，那么有的进程就会得不到足够的帧，从而会出现颠簸，可以选择暂停其中的一个进程。

一旦确定了Δ，那么工作集模型的使用就较为简单。操作系统跟踪每个进程的工作集，并为进程分配大于其工作集的帧数。如果还有空闲帧，那么可启动另一进程。如果所有工作集之和的增加超过了可用帧的总数，那么操作系统会选择暂停一个进程。该进程的页被写出，且其帧可分配给其他进程。挂起的进程可以在以后重启。

这种工作集策略防止了颠簸，并尽可能地提高了多道程序的程度。因此，它优化了 CPU 使用率。

实现工作集模型的困难是跟踪工作集。工作集窗口是移动窗口。在每次引用时，会增加新引用，而最老的引用会失去。如果一个页在工作集窗口内被引用过，那么它就处于工作集内。

通过固定定时中断和引用位，能近似模拟工作集模型。例如，假设Δ为 10000 个引用，且每 5000 个引用会产生定时中断。当出现定时中断时，先复制再清除所有页的引用位。这样，当出现缺页时，可以检查当前引用位和位于内存内的两个位，从而确定在过去的 10000 到 15000 个引用之间该页是否被引用过。如果使用过，至少有一个位会为 1。如果没有使用过，那么所有这 3 个位均为 0。只要有 1 个位为 1，那么就可认为处于工作集中。注意，这种安排并不完全准确，这是因为并不知道在 5000 个引用的什么位置出现了引用。通过增加历史位的位数和中断频率(例如，10 位和每 1000 个引用就产生中断),可以降低这一不确定性。然而，处理这些更为经常的中断的时间也会增加。

6. 缺页频率

工作集知识能用于预先调页，但是用于控制颠簸有点不太灵活。更为直接的方法是采用缺页频率(page fault frequency, PFF)策略。

这里的问题是如何防止颠簸。颠簸具有高的缺页率。因此，需要控制缺页率。当缺页率太高时，进程需要更多帧。类似地，如果缺页率太低，那么进程可能有太多的帧。可以为所期望的缺页率设置上限和下限(见图 6.20)。如果实际缺页率超过上限，那么为进程分配更多的帧；如果实际缺页率低于下限，那么可从该进程中回收一些帧。因此，可以直接测量和控制缺页率以防止颠簸。

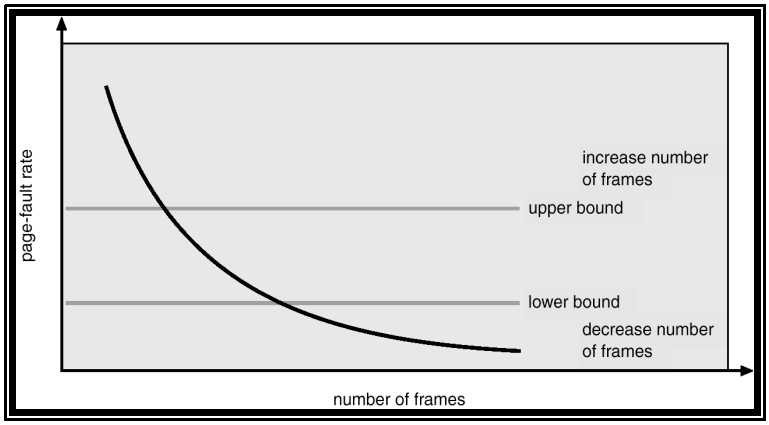

图 6.20 缺页频率

与工作集策略类似，操作系统也可选择暂停一个进程。如果缺页增加且没有可用帧，那么必须选择一个进程暂停。接着，可将释放的帧分配给那些具有高缺页率的进程。