的一个主要问题（到目前为止我们一直在小心地回避这个问题）是，怎样在相互竞争的可运行进程之间分配内存。

如图3-23a所示，三个进程A、B、C构成了可运行进程的集合。假如A发生了缺页中断，页面置换算法在寻找最近最少使用的页面时是只考虑分配给A的6个页面呢？还是考虑所有在内存中的页面？如果只考虑分配给A的页面，生存时间值最小的页面是A5，于是将得到图3-23b所示的状态。

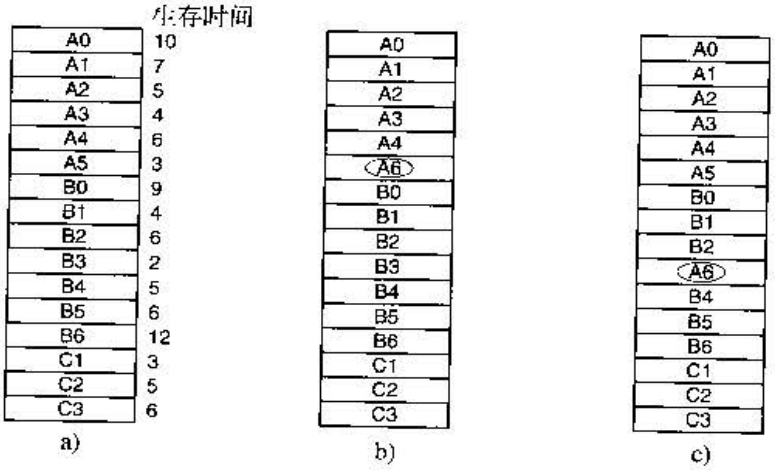

图3-23 局部页面置换与全局页面置换：a) 最初配置；b) 局部页面置换；c) 全局页面置换

另一方面，如果淘汰内存中生存时间值最小的页面，而不管它属于哪个进程，则将选中页面B3，于是将得到图3-23c所示的情况。图3-23b的算法被称为局部（local）页面置换算法，而图3-23c被称为全局（global）页面置换算法。局部算法可以有效地为每个进程分配固定的内存片段。全局算法在可运行进程之间动态地分配页框，因此分配给各个进程的页框数是随时间变化的。

全局算法在通常情况下工作得比局部算法好，当工作集的大小随进程运行时间发生变化时这种现象更加明显。若使用局部算法，即使有大量的空闲页框存在，工作集的增长也会导致颠簸。如果工作集缩小了，局部算法又会浪费内存。在使用全局算法时，系统必须不停地确定应该给每个进程分配多少页框。一种方法是监测工作集的大小，工作集大小由“老化”位指出，但这个方法并不能防止颠簸。因为工作集的大小可能在几微秒内就会发生改变，而老化位却要经历一定的时钟滴答数才会发生变化。

另一种途径是使用一个为进程分配页框的算法。其中一种方法是定期确定进程运行的数目并为它们分配相等的份额。例如，在有12 416个有效（即未被操作系统使用的）页框和10个进程时，每个进程将获得1241个页框，剩下的6个被放入到一个公用池中，当发生缺页中断时可以使用这些页面。

这个算法看起来好像很公平，但是给一个10KB的进程和一个300KB的进程分配同样大小的内存块是很不合理的。可以采用按照进程大小的比例来为它们分配相应数目的页面的方法来取代上一种方法，这样300KB的进程将得到10KB进程30倍的份额。比较明智的一个可行的做法是对每个进程都规定一个最小的页框数，这样不论多么小的进程都可以运行。例如，在某些机器上，一条两个操作数的指令会需要多达6个页面，因为指令自身、源操作数和目的操作数可能会跨越页面边界，若只给一条这样的指令分配了5个页面，则包含这样的指令的程序根本无法运行。

如果使用全局算法，根据进程的大小按比例为其分配页面也是可能的，但是该分配必须在程序运行时动态更新。管理内存动态分配的一种方法是使用PFF（Page Fault Frequency，缺页中断率）算法。它指出了何时增加或减少分配给一个进程的页面，但却完全没有说明在发生缺页中断时应该替换掉哪一个页面，它仅仅控制分配集的大小。

正如我们上面讨论过的，有一大类页面置换算法（包括LRU在内），缺页中断率都会随着分配的页面的增加而降低，这是PFF背后的假定。这一性质在图3-24中说明。

测量缺页中断率的方法是直截了当的：计算每秒的缺页中断数，可能也会将过去数秒的情况做连续平均。一个简单的方法是将当前这一秒的值加到当前的连续平均值上然后除以2。虚线A对应于一个高得不可接受的缺页中断率，虚线B则对应于一个低得可以假设进程拥有过多内存的缺页中断率。在这种情