5.1 内存管理基本概念

内存是计算机运行的中心，由一组字或字节组成。每个字或字节都有自己的地址。CPU就是估计程序计数器的值从内存提取指令，而这些指令可能会进一步引起对特定地址的内存读写操作。

1. 源程序的常规处理流程

通常程序以二进制可执行文件的形式存储在磁盘上。由于 CPU 所能直接访问的存储器只有内存和处理器内的寄存器。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。因此，执行指令以及指令使用的数据必须在这些直接可访问的存储设备上。如果不在内存中，那么在执行前，程序会被调入内存并放在进程空间内。

许多操作系统允许用户进程放在物理内存的任意位置。因此，虽然计算机的地址空间从00000 开始，但用户进程的开始地址不必也是 00000 。这种组织方式会影响用户程序能够使用的地址空间。所以，在绝大多数情况下，用户程序在成为进程前，需要经过好几个步骤的准备工作(参见图 5.1，其中有些步骤是可选的)，并且在这些步骤中它们使用了不同的地址表示形式：

(1) 编辑：形成源文件(用符号地址表示)

(2) 编译：形成目标模块(模块内符号地址解析)

(3) 链接：由多个目标模块或程序库生成可执行文件(模块间符号地址解析)

(4) 装入：构造 PCB，形成进程(使用物理地址)

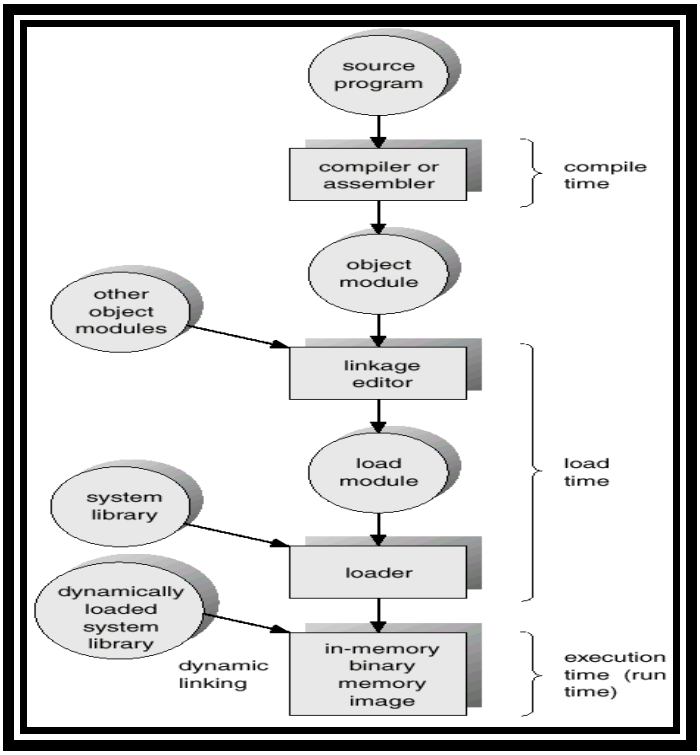

图 5.1 一个源程序的常规处理流程

源程序中的地址通常是用符号地址来表示的。编译器通常将这些符号地址映射到可重定位的模块内符号地址(类似“从本模块开始的第 X 个字节”)。链接程序或加载程序再将这些模块内符号地址映射成绝对地址。期间会完成多次从一个地址空间到另一个地址空间的映射。

根据所使用的内存管理方案，进程在执行时可以在磁盘和内存之间交换。在磁盘上等待调入内存以便执行的进程形成输入队列(input queue) 。通常的步骤是从输入队列中选取一个进程并装入内存。进程在执行时，会访问内存中的指令和数据。最后，进程终止，其地址空间将被释放。

2. 逻辑地址和物理地址

内存中存储单元的地址，即加载到内存地址寄存器(memory-address register)中的内存单元直接看到的地址通常称为物理地址 (physical address)。可以通过物理地址直接在内存中进行寻址。

CPU 所生成的地址通常称为逻辑地址(logical address)，它是由用户的程序经过汇编或编译后形成目标代码，目标代码通常采用相对地址的形式：

(1) 它的首地址为 0，其余指令中的地址都相对于首地址来编址。

(2) 不能通过逻辑地址在内存中读取信息。

逻辑地址也称为虚拟地址 (virtual address)。

由程序所生成的所有逻辑地址的集合称为逻辑地址空间(logical address space)，与这些逻辑地址相对应的所有物理地址的集合称为物理地址空间(physical address space)。如下图 5.2，可以通过地址绑定来完成逻辑地址空间和物理地址空间的映射。

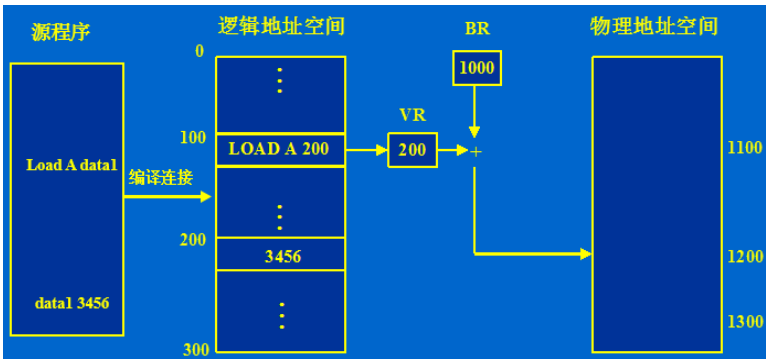

图 5.2 逻辑地址空间与物理地址空间的映射

3. 地址绑定

把相对地址转换成内存中的物理地址，这个过程称为地址绑定(address binding)。

通常，将指令和数据地址绑定到内存地址可以在源程序处理流程的三个不同的阶段发生：

(1) 编译时期(compile time)： 如果在编译时就知道进程将在内存中的驻留地址，那么就可以生成绝对代码(absolute code)。例如，如果事先就知道用户进程驻留在内存地址 R 处，那么所生成的编译代码就可以从该位置开始并向后扩展。如果将来开始地址发生变化，那么就必须重新编译代码。 MS-DOS 的.COM 格式程序就是在编译时绑定成绝对代码的。

(2) 装入时期(load time)： 如果在编译时并不知道进程将驻留在内存的什么地方，那么编译器就必须生成可重定位代码(relocatable code)。对于这种情况，最后绑定会延迟到加载时才进行。如果开始地址发生变化，只需重新加载用户代码以引入改变值。

(3) 执行时期(execution time)： 如果进程在执行时可以从一个内存段移到另一个内存段，那么绑定必须延迟到执行时才进行。采用这种方案需要特定的硬件对地址映射的支持。目前绝大多数通用计算机操作系统采用这种方法。

按照上面地址绑定的不同时期，可分为静态绑定和动态绑定。

静态绑定是在程序执行之前进行绑定，也就是在编译时期和装入时期。它根据执行程序将要装入的内存起始地址，直接修改执行程序中的有关使用地址的指令。在下图 5.3 中以“0”作为参考地址的执行程序，要装入以 10000 为起始地址的存储空间。在装入程序之前，程序必须做一些修改才能正确运行。

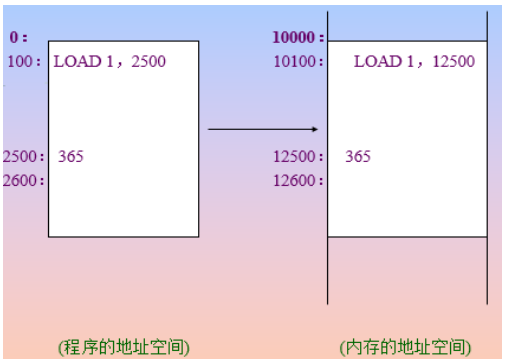

图 5.3 静态绑定：指令中的地址码要作相应的修改

动态绑定是指在程序执行时期进行地址绑定，即在每次访问内存单元前才进行地址变换。需要硬件 MMU 的支持。下图 5.4 给出了动态绑定的示意图。

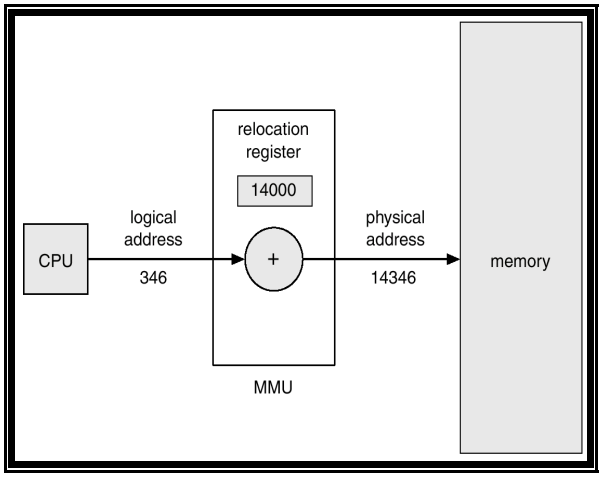

图 5.4 动态绑定：通过 MMU 进行地址变换

4. MMU

程序执行期间时从虚拟地址到物理地址的映射是由 CPU 中个被称为内存管理单元(memory-management unit, MMU) 的硬件设备来完成的。有许多可选择的方法来完成这种映射，其中一个比较简单通用的方法就是用一个简单的 MMU 方案来实现这种映射，它主要是通过一个重定位寄存器(relocation register)来实现地址的转换：用户进程所生成的地址在送交内存之前，都将加上重定位寄存器的值(如上图 5.4 所示)。例如，如果基地址为 14000，那么用户对位置 0 的访问将动态地重定位为位置 14000；对地址 346 的访问将映射为位置 14346。运行于 Intel 80x86 系列 CPU 的 MS-DOS 操作系统在加载和运行进程时，可使用 4 个重定位地址寄存器。

现代的多用户多进程操作系统，需要 MMU，才能达到每个用户进程都拥有自己独立的地址空间的目标。进程在读取地址时，这个地址不是被直接送到内存地址总线上，而是送到存储器管理单元 MMU，把虚拟地址映射为物理地址后再映射到物理内存里的内容。这就是 MMU 在当中进行地址转换所起的作用。

5. 动态加载

如果一个进程的整个程序和数据必须处于物理内存中才能执行，那么进程的大小就受物理内存大小的限制。因此为了获得更好的内存空间使用率，可以使用动态加载(dynamic loading)。采用动态加载时，一个子程序在调用之前不需要被加载，只有在调用时才被加载。所有子程序都以可重定位的形式保存在磁盘上，主程序装入内存并执行。当一个子程序需要调用另一个子程序时，调用子程序首先检查另一个子程序是否已加载。如果没有，可重定位的链接程序将用来加载所需要的子程序，并更新程序的地址表以反映这一变化。接着，控制传递给新加载的子程序。

动态加载的优点是不用的子程序决不会被加载。当需要大量的代码来处理不经常发生的事情(异常处理)时是非常有用的。对于这种情况，虽然总体上程序比较大，但是所使用的部分(即加载的部分)可能小很多。动态加载不需要操作系统提供特别的支持。利用这种方法来设计程序主要是用户的责任。不过，操作系统可以帮助程序员，如提供子程序库以实现动态加载。

6. 动态链接

动态链接的概念与动态加载相似。只是不是将加载延迟到运行时，而是将链接延迟到运行时。

如果有动态链接，二进制镜像中对每个库程序的引用都有一个存根(stub) 。存根是一小段代码，用来指出如何定位适当的内存驻留库程序，或如果该程序不在内存时应如何装入库。当执行存根时，它首先检查所需子程序是否已在内存中。如果不在，就将子程序装入内存。不管如何，存根会用子程序地址来替换自己，并开始执行子程序。因此，下次再执行该子程序代码时，就可以直接进行，而不会因动态链接产生任何开销。

这一特点通常用于系统库，如果没有动态链接，所有程序都需要一份库的副本，浪费了磁盘空间和内存空间。而且如果库被更新的版本替换时，所有用到该库的程序必须重新链接以便访问新的库。采用动态链接方案时，使用库的所有进程只需要一份库代码副本。

与动态加载不同，动态链接通常需要操作系统的帮助。如果内存中进程是彼此保护的，那么只有操作系统才可以检查所需子程序是否在其他进程内存空间内，或是允许多个进程访问同一内存地址。

7. 交换技术

交换技术最早用在MIT的分时系统CTSS中。所有用户作业都驻留在外存的后备队列中，每次只调入一个作业进入内存运行，此作业的时间片用完时，又将该作业调至外存，再将后备队列中的一个作业调入内存运行一个时间片。这是早期的简单分时系统，它采用早期的交换(调进 roll in / 调出 roll out)满足多个程序共享主存的需要。

在操作系统中，进程需要在内存中执行，不过，进程可以暂时从内存中交换(swap)到备份区(backing store)上，当需要再次执行时再换回到内存中。这就是交换技术。当一个进程的时间片用完，内存管理器开始将刚刚执行过的进程换出，将另一个进程换入到刚刚释放的内存空间中(如图 5.5) 。同时，CPU 调度器将时间片分配给其他已在内存中的进程。当每个进程用完时间片，它将与另一进程进行交换。其中备份区(也称交换区)是快速磁盘上一个固定的足够大的可以容纳所有用户内存映像的拷贝；它必须提供对这些内存镜像的直接访问。对于 Linux 和 Unix，通常存储中包含交换分区；Windows 则通常使用交换文件 pagefile.sys。

这种交换技术被用在操作系统的进程调度中。调出，调进的交换基于优先级的算法而不同，低优先级的进程被换出，这样高优先级的进程可以被装入和执行。系统有一个就绪队列，

它包括在备份区或在内存中准备运行的所有进程。当 CPU 调度程序决定执行进程时，它调用调度程序。调度程序检查队列中的下一进程是否在内存中。如果不在内存中且没有空闲内存空间，调度程序将一个已在内存中的进程交换出去，并换入所需要的进程。然后，它重新装载寄存器，并将控制转交给所选择的进程。

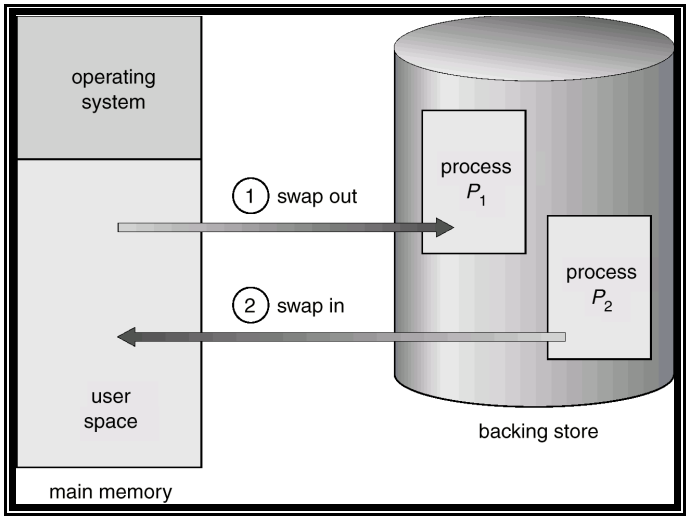

图 5.5 使用快速磁盘作为备份区的两个进程的交换

与内存访问相比，交换需要很大的时间开销，其中主要部分是传输时间，总的传输时间直接同内存交换的数量成正比。因此，知道一个用户进程所真正需要的内存空间，而不是其可能所需要的内存空间，是非常有用的。这样，只需要交换真正使用的内存，以减少变换时间。为有效使用这种方法，用户需要告诉系统其内存需求情况。因此，具有动态内存需求的进程要通过系统调用(请求内存和释放内存)来通知操作系统其内存需求变化情况。

8. 内存分区

内存必须容纳操作系统和各种用户进程，因此应该尽可能有效地分配内存的各个部分。通常需要将多个进程同时放在内存中，因此需要考虑如何为输入队列中需要调入内存的进程分配内存空间。采用连续内存分配(contiguous memory allocation) 时，每个进程位于一个连续的内存区域。

一种简单的内存管理方式通常把内存分为两个部分：一个用于驻留操作系统，另一个用于用户进程。操作系统可以位于低内存，也可位于高内存。影响这一决定的主要因素是中断矢量的位置。由于中断矢量通常位于低内存，因此程序员通常将操作系统也放在低内存，把用户进程保存在内存高端。这种内存管理方式叫做单一分区分配，是一种最简单的存储管理方式，但只能用于单用户、单任务的操作系统，如在 8 位和 16 位微机上 CP/M 和 MS-DOS 操作系统。

为了满足多道程序的需求，分区式管理是另一种简单的存储管理方案。它的基本思想是将内存划分成若干个连续区域，称为内存分区。每个分区只能存储一个程序，且程序也只能在它所驻留的分区中运行。这种方式叫多分区分配。

9. 内存保护

内存保护(Memory protection)是操作系统对电脑上的内存进行访问权限管理的一个机制。内存保护的主要目的是防止某个进程去访问不是操作系统配置给它的寻址空间。这个机制可以防止某个进程，因为某些程序错误或问题，而有意或无意地影响到其他进程或是操作系

统本身的运行状态和数据。

通过采用前面提到的重定位寄存器(也称基地址寄存器)和界限地址寄存器，可以实现这种保护。重定位寄存器策略用来保护用户进程同其他进程和改变的操作系统代码和数据分开。重定位寄存器含有最小的物理地址值；界限地址寄存器含有逻辑地址的范围值。有了重定位寄存器和界限地址寄存器，每个逻辑地址必须小于界限地址寄存器。MMU 动态地将逻辑地址加上重定位寄存器的值后映射成物理地址，这个过程叫做地址映射。映射后的物理地址才会送交内存单元(见图 5.6) 。

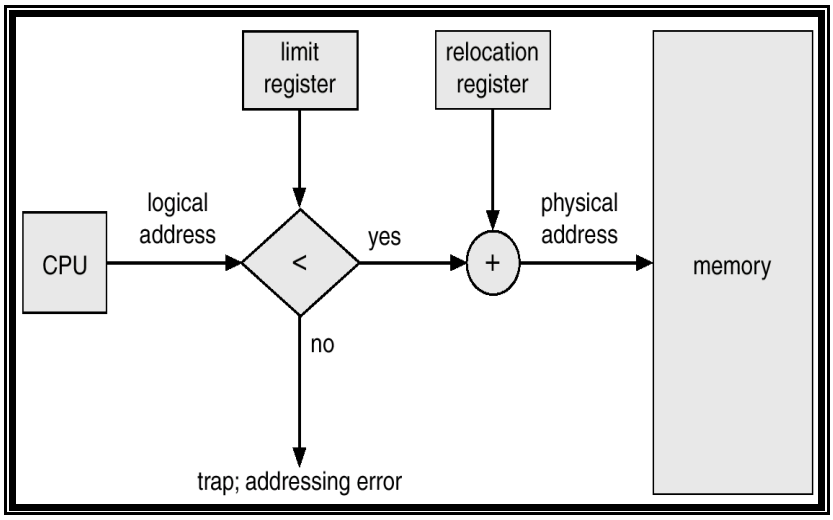

图 5.6 重定位寄存器和界限地址寄存器

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，调度程序会用正确的值来初始化重定位寄存器和界限地址寄存器。由于 CPU 所产生的每一地址都需要与寄存器进行核对，所以可以保证操作系统和其他用户程序和数据不受该进程的运行所影响。

10. 动态分区管理

对于内存的分区式管理，最简单的方法之一就是操作系统预先将可分配的主存空间分为若干个连续的分区。每个分区只能容纳一个进程。当一个分区空闲时，可以从输入队列中选择一个进程，以调入到空闲分区。当进程终止时，其分区可以被其他进程所使用。不过一旦分好，则每个分区的大小固定不再变化，且分区的个数也不再改变。一个分区只能容纳一个作业。这种分区方式叫固定分区，最初为 IBM 08/360 操作系统所使用，现在已不再使用。

而动态分区正好与上面相对，在作业装入内存时把可用内存“切出”一个连续的区域分配给该作业，且分区大小正好适合作业的需要。

在这种动态分区的方案里，操作系统有一个表，用于记录哪些内存可用和哪些内存已被占用。一开始，所有内存都可用于用户进程，因此可以作为一大块可用内存，称为孔(hole) 。当有新进程需要内存时，为该进程查找足够大的孔。如果找到，可以从该孔为该进程分配所需的内存，孔内未分配的内存可以下次再用。

随着进程进入系统，它们将被加入到输入队列。操作系统根据所有进程的内存需要和现有可用内存情况来决定哪些进程可分配内存。当进程分配到空间时，它就装入内存，并开始竞争 CPU 。当进程终止时，它将释放内存，该内存可以被操作系统分配给输入队列中的其他进程。

在任意时候，有一组可用孔(块)大小列表和输入队列。操作系统根据调度算法来对输入队列进行排序。内存不断地分配给进程，直到下一个进程的内存需求不能满足为止，这时没

有足够大的可用孔来装入进程。操作系统可以等到有足够大的空间，或者往下扫描输入队列以确定是否有其他内存需求较小的进程可以被满足。

11. 动态存储分配算法

在动态分区管理中，通常有一组不同大小的孔分散在内存中。当新进程需要内存时，系统为该进程查找足够大的孔。如果孔太大，那么就分为两块：一块分配给新进程，另一块还回到孔集合。当进程终止时，它将释放其内存，该内存将还给孔集合。如果新孔与其他孔相邻，那么将这些孔合并成大孔。这时，系统可以检查是否有进程在等待内存空间，新合并的内存空间是否满足等待进程。

这种方法是通用动态存储分配问题的一种情况，这个问题有许多解决方法，在这里把它叫做动态存储分配算法。动态存储分配算法解决的问题就是怎样从一个空闲的分区列表中满足一个申请需要。其中最为常用方法有首次适应(first-fit)、最佳适应(best-fit)、最差适应(worst-fit)和下次适应(next-fit)。

(1) 首次适应：从头开始查找，只要找到一个足够大的孔，就把它划分后分配出去。

设作业分配序列： A：12K， B：10K，C： 3K， 则分配过程如下：

图 5.7 首次适应算法下的分配过程

(2) 最佳适应：分配最小的足够大的孔。必须查找整个列表，除非列表按大小排序。这种方法可以产生最小剩余孔。

设作业分配序列： A：12K， B：10K，C： 3K， 则分配过程如下：

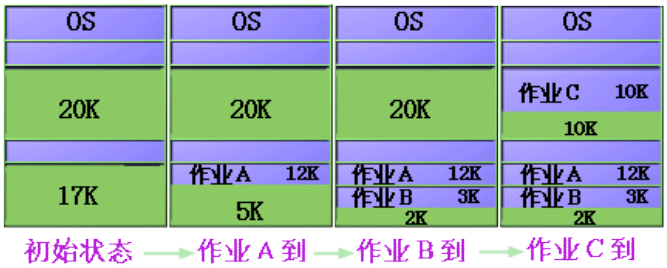

图 5.8 最佳适应算法下的分配过程

(3) 最差适应：分配最大的孔。同样，必须查找整个列表，除非列表按大小排序。这种方法可以产生最大剩余孔，该孔可能比最佳适应方法产生的较小剩余孔更为有用。但在速度和存储的利用上，首次适应和最佳适应要比最差适应好。

设作业分配序列： A：12K， B：10K，C： 3K， 则分配过程如下：

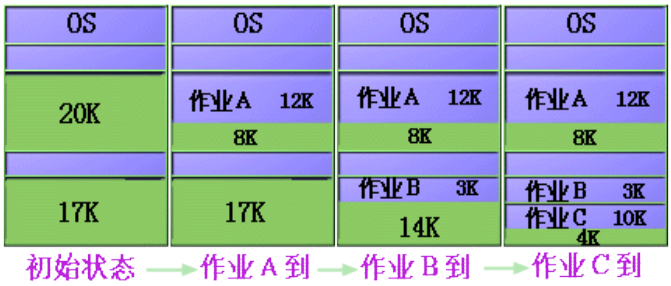

图 5.9 最差适应算法下的分配过程

(4) 下次适应：类似首次适应，只要找到一个足够大的孔，就把它划分后分配出去。但每次分区时，都改为从上次查找结束的地方开始。

设作业分配序列： A：12K， B：10K，C： 3K， 则分配过程如下：

图 5.10 下次适应算法下的分配过程

12. 碎片

在系统不断地分配和回收中，必定会出现一些不连续的小的空闲区，这种存在于分区之间的空闲块不能被充分利用。 这部分空闲块叫做外部碎片(external fragmentation)。

首次适应法和最佳适应方法算法都有外部碎片问题。随着进程装入和移出内存，空闲内存空间被分为小片段。当所有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，该问题可能很严重。在最坏情况下，每两个进程之间就有空闲块(或浪费)。如果这些内存是一整块，那么就可以再运行多个进程。在首次适应和最佳适应之间的选择可能会影响碎片的量(对一些系统来说首次适应更好，对另一些系统最佳适应更好)。另一个影响因素是从空闲块的哪端开始分配(顶端还是末端)。不管使用哪种算法，外部碎片始终是个问题。

一种解决外部碎片间题的方法是紧缩(compaction) 。紧缩的目的是移动内存内容，以便所有空闲空间合并成一整块。但紧缩并非总是可能的。如果重定位是静态的，并且在汇编时或装入时进行的，那么就不能紧缩。紧缩仅在重定位是动态并在运行时可采用。如果地址被动态重定位，可以首先移动程序和数据，然后再根据新基地址的值来改变基地址寄存器。如果能采用紧缩，还需要评估其开销。最简单的合并算法是简单地将所有进程移到内存的一端，而将所有的孔移到内存的另一端，以生成一个大的空闲块。这种方案开销较大。

另一种可能解决外部碎片问题的方法是允许物理地址空间为非连续，这样只要有物理内存就可为进程分配。这种方案有两种互补的实现技术：分页和分段，将在后面的课程进行介绍。这种两种技术也可合并。

内存碎片还可以是内部的。设想有一个 18464B 大小的孔，并采用多分区分配方案。假如有一个进程需要 18462B 。如果只准确分配所要求的块，那么还剩下一个 2B 的孔。维护这一小孔的开销要比孔本身大得多。因此，通常将内存以固定大小的块为单元(而不是字节)来分配。采用这种方案，进程所分配的内存可能比所要的要大。这两个数字之差称为内部碎片，

这部分内存在分区内，但又不能被充分利用。在这种动态存储分配的方法中，内部碎片是无法消除的。