15.3 动态（基于硬件）重定位

为了更好地理解基于硬件的地址转换，我们先来讨论它的第一次应用。在 20 世纪 50 年代后期，它在首次出现的时分机器中引入，那时只是一个简单的思想，称为基址加界限机制（base and bound），有时又称为动态重定位（dynamic relocation），我们将互换使用这两个术语[SS74]。

具体来说，每个 CPU 需要两个硬件寄存器：基址（base）寄存器和界限（bound）寄存器，有时称为限制（limit）寄存器。这组基址和界限寄存器，让我们能够将地址空间放在物理内存的任何位置，同时又能确保进程只能访问自己的地址空间。

采用这种方式，在编写和编译程序时假设地址空间从零开始。但是，当程序真正执行时，操作系统会决定其在物理内存中的实际加载地址，并将起始地址记录在基址寄存器中。在上面的例子中，操作系统决定加载在物理地址 32KB 的进程，因此将基址寄存器设置为这个值。

当进程运行时，有趣的事情发生了。现在，该进程产生的所有内存引用，都会被处理器通过以下方式转换为物理地址：

```
physical address = virtual address + base
```

补充：基于软件的重定位

在早期，在硬件支持重定位之前，一些系统曾经采用纯软件的重定位方式。基本技术被称为静态重定位（static relocation），其中一个名为加载程序（loader）的软件接手将要运行的可执行程序，将它的地址重写到物理内存中期望的偏移位置。

例如，程序中有一条指令是从地址 1000 加载到寄存器（即 movl 1000，%eax），当整个程序的地址空间被加载到从 3000（不是程序认为的 0）开始的物理地址中，加载程序会重写指令中的地址（即 movl 4000, %eax），从而完成简单的静态重定位。

然而，静态重定位有许多问题，首先也是最重要的是不提供访问保护，进程中的错误地址可能导致对其他进程或操作系统内存的非法访问，一般来说，需要硬件支持来实现真正的访问保护[WL+93]。静态重定位的另一个缺点是一旦完成，稍后很难将内存空间重定位到其他位置 [M65]。

进程中使用的内存引用都是虚拟地址（virtual address），硬件接下来将虚拟地址加上基址寄存器中的内容，得到物理地址（physical address），再发给内存系统。

为了更好地理解，让我们追踪一条指令执行的情况。具体来看前面序列中的一条指令：

```
128: movl 0x0(%ebx), %eax
```

程序计数器（PC）首先被设置为 128。当硬件需要获取这条指令时，它先将这个值加上基址寄存器中的 32KB(32768)，得到实际的物理地址 32896，然后硬件从这个物理地址获取指令。接下来，处理器开始执行该指令。这时，进程发起从虚拟地址 15KB 的加载，处理器同样将虚拟地址加上基址寄存器内容（32KB），得到最终的物理地址 47KB，从而获得需要的数据。

将虚拟地址转换为物理地址，这正是所谓的地址转换（address translation）技术。也就是说，硬件取得进程认为它要访问的地址，将它转换成数据实际位于的物理地址。由于这

种重定位是在运行时发生的，而且我们甚至可以在进程开始运行后改变其地址空间，这种技术一般被称为动态重定位（dynamic relocation）[M65]。

提示：基于硬件的动态重定位

在动态重定位的过程中，只有很少的硬件参与，但获得了很好的效果。一个基址寄存器将虚拟地址转换为物理地址，一个界限寄存器确保这个地址在进程地址空间的范围内。它们一起提供了既简单又高效的虚拟内存机制。

现在你可能会问，界限（限制）寄存器去哪了？不是基址加界限机制吗？正如你猜测的那样，界限寄存器提供了访问保护。在上面的例子中，界限寄存器被置为 16KB。如果进程需要访问超过这个界限或者为负数的虚拟地址，CPU 将触发异常，进程最终可能被终止。界限寄存器的用处在于，它确保了进程产生的所有地址都在进程的地址“界限”中。

这种基址寄存器配合界限寄存器的硬件结构是芯片中的（每个 CPU 一对）。有时我们将 CPU 的这个负责地址转换的部分统称为内存管理单元（Memory Management Unit，MMU）。随着我们开发更复杂的内存管理技术，MMU 也将有更复杂的电路和功能。

关于界限寄存器再补充一点，它通常有两种使用方式。在一种方式中（像上面那样），它记录地址空间的大小，硬件在将虚拟地址与基址寄存器内容求和前，就检查这个界限。另一种方式是界限寄存器中记录地址空间结束的物理地址，硬件在转化虚拟地址到物理地址之后才去检查这个界限。这两种方式在逻辑上是等价的。简单起见，我们这里假设采用第一种方式。

转换示例

为了更好地理解基址加界限的地址转换的详细过程，我们来看一个例子。设想一个进程拥有 4KB 大小地址空间（是的，小得不切实际），它被加载到从 16KB 开始的物理内存中。一些地址转换结果见表 15.1。

表 15.1 地址转换结果

虚拟地址		物理地址
0	→	16KB
1KB	→	17KB
3000	→	19384
4400	→	错误（越界）

从例子中可以看到，通过基址加虚拟地址（可以看作是地址空间的偏移量）的方式，很容易得到物理地址。虚拟地址“过大”或者为负数时，会导致异常。

补充：数据结构——空闲列表

操作系统必须记录哪些空闲内存没有使用，以便能够为进程分配内存。很多不同的数据结构可以用于这项任务，其中最简单的（也是我们假定在这里采用的）是空闲列表（free list）。它就是一个列表，记录当前没有使用的物理内存的范围。

15.4 硬件支持：总结

我们来总结一下需要的硬件支持（见表 15.2）。首先，正如在 CPU 虚拟化的章节中提到的，我们需要两种 CPU 模式。操作系统在特权模式（privileged mode，或内核模式，kernel mode），可以访问整个机器资源。应用程序在用户模式（user mode）运行，只能做有限的操作。只要一个位，也许保存在处理器状态字（processor status word）中，就能说明当前的 CPU 运行模式。在一些特殊的时刻（如系统调用、异常或中断），CPU 会切换状态。

表 15.2 动态重定位：硬件要求

硬件要求	解释
特权模式	需要，以防用户模式的进程执行特权操作
基址/界限寄存器	每个 CPU 需要一对寄存器来支持地址转换和界限检查
能够转换虚拟地址并检查它是否越界	电路来完成转换和检查界限，在这种情况下，非常简单
修改基址/界限寄存器的特权指令	在让用户程序运行之前，操作系统必须能够设置这些值
注册异常处理程序的特权指令	操作系统必须能告诉硬件，如果异常发生，那么执行哪些代码
能够触发异常	如果进程试图使用特权指令或越界的内存

硬件还必须提供基址和界限寄存器（base and bounds register），因此每个 CPU 的内存管理单元（Memory Management Unit，MMU）都需要这两个额外的寄存器。用户程序运行时，硬件会转换每个地址，即将用户程序产生的虚拟地址加上基址寄存器的内容。硬件也必须能检查地址是否有用，通过界限寄存器和 CPU 内的一些电路来实现。

硬件应该提供一些特殊的指令，用于修改基址寄存器和界限寄存器，允许操作系统在切换进程时改变它们。这些指令是特权（privileged）指令，只有在内核模式下，才能修改这些寄存器。想象一下，如果用户进程在运行时可以随意更改基址寄存器，那么用户进程可能会造成严重破坏[①]。想象一下吧！然后迅速将这些阴暗的想法从你的头脑中赶走，因为它们很可怕，会导致噩梦。

最后，在用户程序尝试非法访问内存（越界访问）时，CPU 必须能够产生异常（exception）。在这种情况下，CPU 应该阻止用户程序的执行，并安排操作系统的“越界”异常处理程序（exception handler）去处理。操作系统的处理程序会做出正确的响应，比如在这种情况下终止进程。类似地，如果用户程序尝试修改基址或者界限寄存器时，CPU 也应该产生异常，并调用“用户模式尝试执行特权指令”的异常处理程序。CPU 还必须提供一种方法，来通知它这些处理程序的位置，因此又需要另一些特权指令。

15.5 操作系统的问题

为了支持动态重定位，硬件添加了新的功能，使得操作系统有了一些必须处理的新问

① 除了“严重破坏（havoc）”还有什么可以“造成（wreaked）”的吗？

题。硬件支持和操作系统管理结合在一起，实现了一个简单的虚拟内存。具体来说，在一些关键的时刻操作系统需要介入，以实现基址和界限方式的虚拟内存，见表 15.3。

第一，在进程创建时，操作系统必须采取行动，为进程的地址空间找到内存空间。由于我们假设每个进程的地址空间小于物理内存的大小，并且大小相同，这对操作系统来说很容易。它可以把整个物理内存看作一组槽块，标记了空闲或已用。当新进程创建时，操作系统检索这个数据结构（常被称为空闲列表，free list），为新地址空间找到位置，并将其标记为已用。如果地址空间可变，那么生活就会更复杂，我们将在后续章节中讨论。

我们来看一个例子。在图 15.2 中，操作系统将物理内存的第一个槽块分配给自己，然后将例子中的进程重定位到物理内存地址 32KB。另两个槽块（16～32KB，48～64KB）空闲，因此空闲列表（free list）就包含这两个槽块。

第二，在进程终止时（正常退出，或因行为不端被强制终止），操作系统也必须做一些工作，回收它的所有内存，给其他进程或者操作系统使用。在进程终止时，操作系统会将这些内存放回到空闲列表，并根据需要清除相关的数据结构。

第三，在上下文切换时，操作系统也必须执行一些额外的操作。每个 CPU 毕竟只有一个基址寄存器和一个界限寄存器，但对于每个运行的程序，它们的值都不同，因为每个程序被加载到内存中不同的物理地址。因此，在切换进程时，操作系统必须保存和恢复基础和界限寄存器。具体来说，当操作系统决定中止当前的运行进程时，它必须将当前基址和界限寄存器中的内容保存在内存中，放在某种每个进程都有的结构中，如进程结构（process structure）或进程控制块（Process Control Block，PCB）中。类似地，当操作系统恢复执行某个进程时（或第一次执行），也必须给基址和界限寄存器设置正确的值。

表 15.3 动态重定位：操作系统的职责

操作系统的要求	解释
内存管理	需要为新进程分配内存 从终止的进程回收内存 一般通过空闲列表（free list）来管理内存
基址/界限管理	必须在上下文切换时正确设置基址/界限寄存器
异常处理	当异常发生时执行的代码，可能的动作是终止犯错的进程

需要注意，当进程停止时（即没有运行），操作系统可以改变其地址空间的物理位置，这很容易。要移动进程的地址空间，操作系统首先让进程停止运行，然后将地址空间拷贝到新位置，最后更新保存的基址寄存器（在进程结构中），指向新位置。当该进程恢复执行时，它的（新）基址寄存器会被恢复，它再次开始运行，显然它的指令和数据都在新的内存位置了。

第四，操作系统必须提供异常处理程序（exception handler），或要一些调用的函数，像上面提到的那样。操作系统在启动时加载这些处理程序（通过特权命令）。例如，当一个进程试图越界访问内存时，CPU 会触发异常。在这种异常产生时，操作系统必须准备采取行动。通常操作系统会做出充满敌意的反应：终止错误进程。操作系统应该尽力保护它运行的机器，因此它不会对那些企图访问非法地址或执行非法指令的进程客气。再见了，行为不端的进程，很高兴认识你。

表 15.4 为按时间线展示了大多数硬件与操作系统的交互。可以看出，操作系统在启动时

做了什么，为我们准备好机器，然后在进程（进程 A）开始运行时发生了什么。请注意，地址转换过程完全由硬件处理，没有操作系统的介入。在这个时候，发生时钟中断，操作系统切换到进程 B 运行，它执行了“错误的加载”（对一个非法内存地址），这时操作系统必须介入，终止该进程，清理并释放进程 B 占用的内存，将它从进程表中移除。从表中可以看出，我们仍然遵循受限直接访问（limited direct execution）的基本方法，大多数情况下，操作系统正确设置硬件后，就任凭进程直接运行在 CPU 上，只有进程行为不端时才介入。

表 15.4　　受限直接执行协议（动态重定位）

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 　系统调用处理程序 　时钟处理程序 　非法内存处理程序 　非常指令处理程序	
开始中断时钟		
	开始时钟，在 *x* ms 后中断	
初始化进程表 初始化空闲列表		
操作系统@运行（核心模式）	硬件	程序（用户模式）
为了启动进程 A： 　在进程表中分配条目 　为进程分配内存 　设置基址/界限寄存器 　从陷阱返回（进入 A）		
	恢复 A 的寄存器 转向用户模式 跳到 A（最初）的程序计数器	
		进程 A 运行 　获取指令
	转换虚拟地址并执行获取	
		执行指令
	如果显式加载/保存 　确保地址不越界 　转换虚拟地址并执行 　加载/保存	
		……
	时钟中断 转向内核模式 跳到中断处理程序	

续表

操作系统@启动（内核模式）	硬件	
处理陷阱 调用 switch()例程 　将寄存器（A）保存到进程结构（A） 　（包括基址/界限） 　从进程结构（B）恢复寄存器（B） 　（包括基址/界限） 从陷阱返回（进入 B）		
	恢复 B 的寄存器 转向用户模式 跳到 B 的程序计数器	
		进程 B 运行 　执行错误的加载
	加载越界 转向内核模式 跳到陷阱处理程序	
处理本期报告 　决定终止进程 B 　回收 B 的内存 　移除 B 在进程表中的条目		

15.6 小结

本章通过虚拟内存使用的一种特殊机制，即地址转换（address translation），扩展了受限直接访问的概念。利用地址转换，操作系统可以控制进程的所有内存访问，确保访问在地址空间的界限内。这个技术高效的关键是硬件支持，硬件快速地将所有内存访问操作中的虚拟地址（进程自己看到的内存位置）转换为物理地址（实际位置）。所有的这一切对进程来说都是透明的，进程并不知道自己使用的内存引用已经被重定位，制造了美妙的假象。

我们还看到了一种特殊的虚拟化方式，称为基址加界限的动态重定位。基址加界限的虚拟化方式非常高效，因为只需要很少的硬件逻辑，就可以将虚拟地址和基址寄存器加起来，并检查进程产生的地址没有越界。基址加界限也提供了保护，操作系统和硬件的协作，确保没有进程能够访问其地址空间之外的内容。保护肯定是操作系统最重要的目标之一。没有保护，操作系统不可能控制机器（如果进程可以随意修改内存，它们就可以轻松地做出可怕的事情，比如重写陷阱表并完全接管系统）。

遗憾的是，这个简单的动态重定位技术有效率低下的问题。例如，从图 15.2 中可以看到，

重定位的进程使用了从 32KB 到 48KB 的物理内存，但由于该进程的栈区和堆区并不很大，导致这块内存区域中大量的空间被浪费。这种浪费通常称为内部碎片（internal fragmentation），指的是已经分配的内存单元内部有未使用的空间（即碎片），造成了浪费。在我们当前的方式中，即使有足够的物理内存容纳更多进程，但我们目前要求将地址空间放在固定大小的槽块中，因此会出现内部碎片[①]。所以，我们需要更复杂的机制，以便更好地利用物理内存，避免内部碎片。第一次尝试是将基址加界限的概念稍稍泛化，得到分段（segmentation）的概念，我们接下来将讨论。

参考资料

[M65]“On Dynamic Program Relocation”
W.C. McGee
IBM Systems Journal
Volume 4, Number 3, 1965, pages 184–199
本文对动态重定位的早期工作和静态重定位的一些基础知识进行了很好的总结。

[P90]“Relocating loader for MS-DOS .EXE executable files”Kenneth D. A. Pillay
Microprocessors & Microsystems archive Volume 14, Issue 7 (September 1990)
MS-DOS 重定位加载器的示例。不是第一个，而只是这样的系统如何工作的一个相对现代的例子。

[SS74]“The Protection of Information in Computer Systems”
J. Saltzer and M. Schroeder CACM, July 1974
摘自这篇论文：“在 1957 年至 1959 年间，在 3 个有不同目标的项目中，显然独立出现了基址和界限寄存器和硬件解释描述符的概念。在 MIT，McCarthy 建议将基址和界限的想法作为内存保护系统的一部分，以便让时分共享可行。IBM 独立开发了基本和界限寄存器，作为 Stretch（7030）计算机系统支持可靠多道程序的机制。在 Burroughs，R. Barton 建议硬件解释描述符可以直接支持 B5000 计算机系统中高级语言的命名范围规则。”我们在 Mark Smotherman 的超酷历史页面上找到了这段引用[S04]，更多信息请参见这些页面。

[S04]“System Call Support”
Mark Smotherman, May 2004
系统调用支持的简洁历史。Smotherman 还收集了一些早期历史，包括中断和其他有趣方面的计算历史。可以查看他的网页了解更多详情。

[WL+93]“Efficient Software-based Fault Isolation”
Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham SOSP ’93
关于如何在没有硬件支持的情况下，利用编译器支持限定从程序中引用内存的一篇极好的论文。该论文引

① 另一种解决方案可能会在地址空间内放置一个固定大小的栈，位于代码区域的下方，并在栈下面让堆增长。 但是，这限制了灵活性，让递归和深层嵌套函数调用变得具有挑战，因此我们希望避免这种情况。

发了人们对用于分离内存引用的软件技术的兴趣。

作业

程序 relocation.py 让你看到，在带有基址和边界寄存器的系统中，如何执行地址转换。详情请参阅 README 文件。

问题

1．用种子 1、2 和 3 运行，并计算进程生成的每个虚拟地址是处于界限内还是界限外？如果在界限内，请计算地址转换。

2．使用以下标志运行：-s 0 -n 10。为了确保所有生成的虚拟地址都处于边界内，要将 -l（界限寄存器）设置为什么值？

3．使用以下标志运行：-s 1 -n 10 -l 100。可以设置界限的最大值是多少，以便地址空间仍然完全放在物理内存中？

4．运行和第 3 题相同的操作，但使用较大的地址空间（-a）和物理内存（-p）。

5．作为边界寄存器的值的函数，随机生成的虚拟地址的哪一部分是有效的？画一个图，使用不同随机种子运行，限制值从 0 到最大地址空间大小。

第 16 章　分段

到目前为止，我们一直假设将所有进程的地址空间完整地加载到内存中。利用基址和界限寄存器，操作系统很容易将不同进程重定位到不同的物理内存区域。但是，对于这些内存区域，你可能已经注意到一件有趣的事：栈和堆之间，有一大块“空闲”空间。

从图 16.1 中可知，如果我们将整个地址空间放入物理内存，那么栈和堆之间的空间并没有被进程使用，却依然占用了实际的物理内存。因此，简单的通过基址寄存器和界限寄存器实现的虚拟内存很浪费。另外，如果剩余物理内存无法提供连续区域来放置完整的地址空间，进程便无法运行。这种基址加界限的方式看来并不像我们期望的那样灵活。因此：

关键问题：怎样支持大地址空间

怎样支持大地址空间，同时栈和堆之间（可能）有大量空闲空间？在之前的例子里，地址空间非常小，所以这种浪费并不明显。但设想一个 32 位（4GB）的地址空间，通常的程序只会使用几兆的内存，但需要整个地址空间都放在内存中。

16.1　分段：泛化的基址/界限

为了解决这个问题，分段（segmentation）的概念应运而生。分段并不是一个新概念，它甚至可以追溯到 20 世纪 60 年代初期[H61, G62]。这个想法很简单，在 MMU 中引入不止一个基址和界限寄存器对，而是给地址空间内的每个逻辑段（segment）一对。一个段只是地址空间里的一个连续定长的区域，在典型的地址空间里有 3 个逻辑不同的段：代码、栈和堆。分段的机制使得操作系统能够将不同的段放到不同的物理内存区域，从而避免了虚拟地址空间中的未使用部分占用物理内存。

我们来看一个例子。假设我们希望将图 16.1 中的地址空间放入物理内存。通过给每个段一对基址和界限寄存器，可以将每个段独立地放入物理内存。如图 16.2 所示，64KB 的物理内存中放置了 3 个段（为操作系统保留 16KB）。

从图中可以看到，只有已用的内存才在物理内存中分配空间，因此可以容纳巨大的地址空间，其中包含大量未使用的地址空间（有时又称为稀疏地址空间，sparse address spaces）。

你会想到，需要 MMU 中的硬件结构来支持分断：在这种情况下，需要一组 3 对基址和界限寄存器。表 16.1 展示了上面的例子中的寄存器值，每个界限寄存器记录了一个段的大小。

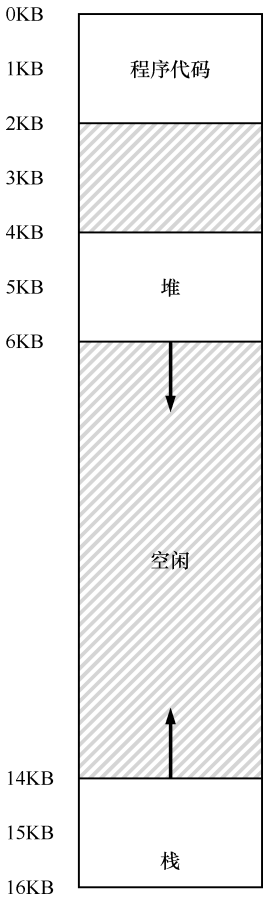


图 16.1 一个地址空间（复习）

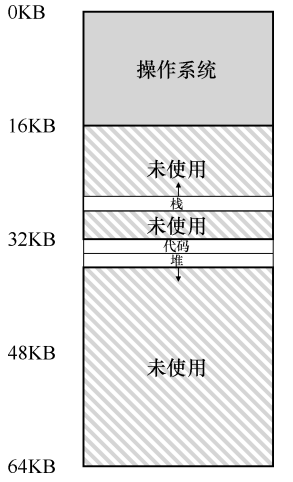


图 16.2 在物理内存中放置段

表 16.1 段寄存器的值

段	基址	大小
代码	32KB	2KB
堆	34KB	2KB
栈	28KB	2KB

如表 16.1 所示，代码段放在物理地址 32KB，大小是 2KB。堆在 34KB，大小也是 2KB。

利用图 16.1 中的地址空间，我们来看一个地址转换的例子。假设现在要引用虚拟地址 100（在代码段中），MMU 将基址值加上偏移量（100）得到实际的物理地址：100 + 32KB = 32868。然后它会检查该地址是否在界限内（100 小于 2KB），发现是的，于是发起对物理地址 32868 的引用。

补充：段错误

段错误指的是在支持分段的机器上发生了非法的内存访问。有趣的是，即使在不支持分段的机器上这个术语依然保留。但如果你弄不清楚为什么代码老是出错，就没那么有趣了。

来看一个堆中的地址，虚拟地址 4200（同样参考图 16.1）。如果用虚拟地址 4200 加上

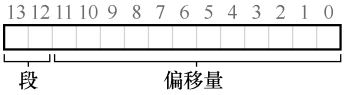
堆的基址（34KB），得到物理地址 39016，这不是正确的地址。我们首先应该先减去堆的偏移量，即该地址指的是这个段中的哪个字节。因为堆从虚拟地址 4K（4096）开始，4200 的偏移量实际上是 4200 减去 4096，即 104，然后用这个偏移量（104）加上基址寄存器中的物理地址（34KB），得到真正的物理地址 34920。

如果我们试图访问非法的地址，例如 7KB，它超出了堆的边界呢？你可以想象发生的情况：硬件会发现该地址越界，因此陷入操作系统，很可能导致终止出错进程。这就是每个 C 程序员都感到恐慌的术语的来源：段异常（segmentation violation）或段错误（segmentation fault）。

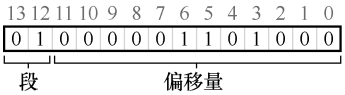
16.2 我们引用哪个段

硬件在地址转换时使用段寄存器。它如何知道段内的偏移量，以及地址引用了哪个段？

一种常见的方式，有时称为显式（explicit）方式，就是用虚拟地址的开头几位来标识不同的段，VAX/VMS 系统使用了这种技术[LL82]。在我们之前的例子中，有 3 个段，因此需要两位来标识。如果我们用 14 位虚拟地址的前两位来标识，那么虚拟地址如下所示：



那么在我们的例子中，如果前两位是 00，硬件就知道这是属于代码段的地址，因此使用代码段的基址和界限来重定位到正确的物理地址。如果前两位是 01，则是堆地址，对应地，使用堆的基址和界限。下面来看一个 4200 之上的堆虚拟地址，进行进制转换，确保弄清楚这些内容。虚拟地址 4200 的二进制形式如下：



从图中可以看到，前两位（01）告诉硬件我们引用哪个段。剩下的 12 位是段内偏移：0000 0110 1000（即十六进制 0x068 或十进制 104）。因此，硬件就用前两位来决定使用哪个段寄存器，然后用后 12 位作为段内偏移。偏移量与基址寄存器相加，硬件就得到了最终的物理地址。请注意，偏移量也简化了对段边界的判断。我们只要检查偏移量是否小于界限，大于界限的为非法地址。因此，如果基址和界限放在数组中（每个段一项），为了获得需要的物理地址，硬件会做下面这样的事：

```
1    // get top 2 bits of 14-bit VA
2    Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3    // now get offset
4    Offset  = VirtualAddress & OFFSET_MASK
5    if (Offset >= Bounds[Segment])
6        RaiseException(PROTECTION_FAULT)
7    else
8        PhysAddr = Base[Segment] + Offset
9        Register = AccessMemory(PhysAddr)
```

在我们的例子中，可以为上面的常量填上值。具体来说，SEG_MASK 为 0x3000，SEG_SHIFT 为 12，OFFSET_MASK 为 0xFFF。

你或许已经注意到，上面使用两位来区分段，但实际只有 3 个段（代码、堆、栈），因此有一个段的地址空间被浪费。因此有些系统中会将堆和栈当作同一个段，因此只需要一位来做标识[LL82]。

硬件还有其他方法来决定特定地址在哪个段。在隐式（implicit）方式中，硬件通过地址产生的方式来确定段。例如，如果地址由程序计数器产生（即它是指令获取），那么地址在代码段。如果基于栈或基址指针，它一定在栈段。其他地址则在堆段。

16.3 栈怎么办

到目前为止，我们一直没有讲地址空间中的一个重要部分：栈。在表 16.1 中，栈被重定位到物理地址 28KB。但有一点关键区别，它反向增长。在物理内存中，它始于 28KB，增长回到 26KB，相应虚拟地址从 16KB 到 14KB。地址转换必须有所不同。

首先，我们需要一点硬件支持。除了基址和界限外，硬件还需要知道段的增长方向（用一位区分，比如 1 代表自小而大增长，0 反之）。在表 16.2 中，我们更新了硬件记录的视图。

表 16.2 段寄存器（支持反向增长）

段	基址	大小	是否反向增长
代码	32KB	2KB	1
堆	34KB	2KB	1
栈	28KB	2KB	0

硬件理解段可以反向增长后，这种虚拟地址的地址转换必须有点不同。下面来看一个栈虚拟地址的例子，将它进行转换，以理解这个过程：

在这个例子中，假设要访问虚拟地址 15KB，它应该映射到物理地址 27KB。该虚拟地址的二进制形式是：11 1100 0000 0000（十六进制 0x3C00）。硬件利用前两位（11）来指定段，但然后我们要处理偏移量 3KB。为了得到正确的反向偏移，我们必须从 3KB 中减去最大的段地址：在这个例子中，段可以是 4KB，因此正确的偏移量是 3KB 减去 4KB，即−1KB。只要用这个反向偏移量（−1KB）加上基址（28KB），就得到了正确的物理地址 27KB。用户可以进行界限检查，确保反向偏移量的绝对值小于段的大小。

16.4 支持共享

随着分段机制的不断改进，系统设计人员很快意识到，通过再多一点的硬件支持，就能实现新的效率提升。具体来说，要节省内存，有时候在地址空间之间共享（share）某些内存段是有用的。尤其是，代码共享很常见，今天的系统仍然在使用。

为了支持共享，需要一些额外的硬件支持，这就是保护位（protection bit）。基本为每个

段增加了几个位，标识程序是否能够读写该段，或执行其中的代码。通过将代码段标记为只读，同样的代码可以被多个进程共享，而不用担心破坏隔离。虽然每个进程都认为自己独占这块内存，但操作系统秘密地共享了内存，进程不能修改这些内存，所以假象得以保持。

表 16.3 展示了一个例子，是硬件（和操作系统）记录的额外信息。可以看到，代码段的权限是可读和可执行，因此物理内存中的一个段可以映射到多个虚拟地址空间。

表 16.3 段寄存器的值（有保护）

段	基址	大小	是否反向增长	保护
代码	32KB	2KB	1	读—执行
堆	34KB	2KB	1	读—写
栈	28KB	2KB	0	读—写

有了保护位，前面描述的硬件算法也必须改变。除了检查虚拟地址是否越界，硬件还需要检查特定访问是否允许。如果用户进程试图写入只读段，或从非执行段执行指令，硬件会触发异常，让操作系统来处理出错进程。

16.5 细粒度与粗粒度的分段

到目前为止，我们的例子大多针对只有很少的几个段的系统（即代码、栈、堆）。我们可以认为这种分段是粗粒度的（coarse-grained），因为它将地址空间分成较大的、粗粒度的块。但是，一些早期系统（如 Multics[CV65, DD68]）更灵活，允许将地址空间划分为大量较小的段，这被称为细粒度（fine-grained）分段。

支持许多段需要进一步的硬件支持，并在内存中保存某种段表（segment table）。这种段表通常支持创建非常多的段，因此系统使用段的方式，可以比之前讨论的方式更灵活。例如，像 Burroughs B5000 这样的早期机器可以支持成千上万的段，有了操作系统和硬件的支持，编译器可以将代码段和数据段划分为许多不同的部分[RK68]。当时的考虑是，通过更细粒度的段，操作系统可以更好地了解哪些段在使用哪些没有，从而可以更高效地利用内存。

16.6 操作系统支持

现在你应该大致了解了分段的基本原理。系统运行时，地址空间中的不同段被重定位到物理内存中。与我们之前介绍的整个地址空间只有一个基址/界限寄存器对的方式相比，大量节省了物理内存。具体来说，栈和堆之间没有使用的区域就不需要再分配物理内存，让我们能将更多地址空间放进物理内存。

然而，分段也带来了一些新的问题。我们先介绍必须关注的操作系统新问题。第一个是老问题：操作系统在上下文切换时应该做什么？你可能已经猜到了：各个段寄存器中的内容必须保存和恢复。显然，每个进程都有自己独立的虚拟地址空间，操作系统必须在进程运行前，确保这些寄存器被正确地赋值。

第二个问题更重要，即管理物理内存的空闲空间。新的地址空间被创建时，操作系统需要在物理内存中为它的段找到空间。之前，我们假设所有的地址空间大小相同，物理内存可以被认为是一些槽块，进程可以放进去。现在，每个进程都有一些段，每个段的大小也可能不同。

一般会遇到的问题是，物理内存很快充满了许多空闲空间的小洞，因而很难分配给新的段，或扩大已有的段。这种问题被称为外部碎片（external fragmentation）[R69]，如图 16.3（左边）所示。

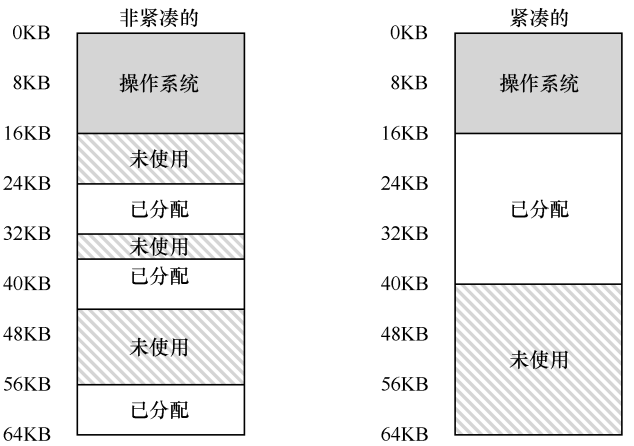


图 16.3 非紧凑和紧凑的内存

在这个例子中，一个进程需要分配一个 20KB 的段。当前有 24KB 空闲，但并不连续（是 3 个不相邻的块）。因此，操作系统无法满足这个 20KB 的请求。

该问题的一种解决方案是紧凑（compact）物理内存，重新安排原有的段。例如，操作系统先终止运行的进程，将它们的数据复制到连续的内存区域中去，改变它们的段寄存器中的值，指向新的物理地址，从而得到了足够大的连续空闲空间。这样做，操作系统能让新的内存分配请求成功。但是，内存紧凑成本很高，因为拷贝段是内存密集型的，一般会占用大量的处理器时间。图 16.3（右边）是紧凑后的物理内存。

一种更简单的做法是利用空闲列表管理算法，试图保留大的内存块用于分配。相关的算法可能有成百上千种，包括传统的最优匹配（best-fit，从空闲链表中找最接近需要分配空间的空闲块返回）、最坏匹配（worst-fit）、首次匹配（first-fit）以及像伙伴算法（buddy algorithm）[K68]这样更复杂的算法。Wilson 等人做过一个很好的调查[W+95]，如果你想对这些算法了解更多，可以从它开始，或者等到第 17 章，我们将介绍一些基本知识。但遗憾的是，无论算法多么精妙，都无法完全消除外部碎片，因此，好的算法只是试图减小它。

提示：如果有一千个解决方案，就没有特别好的

存在如此多不同的算法来尝试减少外部碎片，正说明了解决这个问题没有最好的办法。因此我们满足于找到一个合理的足够好的方案。唯一真正的解决办法就是（我们会在后续章节看到），完全避免这个问题，永远不要分配不同大小的内存块。

16.7 小结

分段解决了一些问题，帮助我们实现了更高效的虚拟内存。不只是动态重定位，通过避免地址空间的逻辑段之间的大量潜在的内存浪费，分段能更好地支持稀疏地址空间。它还很快，因为分段要求的算法很容易，很适合硬件完成，地址转换的开销极小。分段还有一个附加的好处：代码共享。如果代码放在独立的段中，这样的段就可能被多个运行的程序共享。

但我们已经知道，在内存中分配不同大小的段会导致一些问题，我们希望克服。首先，是我们上面讨论的外部碎片。由于段的大小不同，空闲内存被割裂成各种奇怪的大小，因此满足内存分配请求可能会很难。用户可以尝试采用聪明的算法[W+95]，或定期紧凑内存，但问题很根本，难以避免。

第二个问题也许更重要，分段还是不足以支持更一般化的稀疏地址空间。例如，如果有一个很大但是稀疏的堆，都在一个逻辑段中，整个堆仍然必须完整地加载到内存中。换言之，如果使用地址空间的方式不能很好地匹配底层分段的设计目标，分段就不能很好地工作。因此我们需要找到新的解决方案。你准备好了吗？

参考资料

[CV65]“Introduction and Overview of the Multics System”
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
在秋季联合计算机大会上发表的关于 Multics 的 5 篇论文之一。啊，多希望那天我在那个房间里！

[DD68]“Virtual Memory, Processes, and Sharing in Multics”Robert C. Daley and Jack B. Dennis
Communications of the ACM, Volume 11, Issue 5, May 1968
一篇关于如何在 Multics 中进行动态链接的早期文章。文中的内容远远领先于它的时代。随着大型 X-windows 库的要求，动态链接最终在 20 年后回到系统中。有人说，这些大型的 X11 库是 MIT 对 UNIX 的早期版本中取消对动态链接支持的“报复”!

[G62]“Fact Segmentation”
M. N. Greenfield
Proceedings of the SJCC, Volume 21, May 1962
另一篇关于分段的早期论文，发表的时间太早了，以至于没有引用其他人的工作。

[H61]“Program Organization and Record Keeping for Dynamic Storage”
A. W. Holt
Communications of the ACM, Volume 4, Issue 10, October 1961