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。

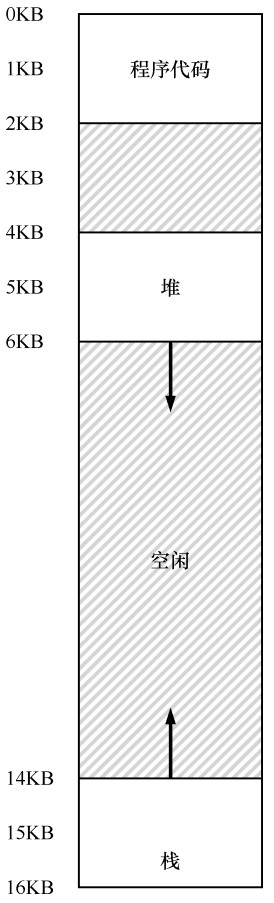


图 16.1 一个地址空间（复习）

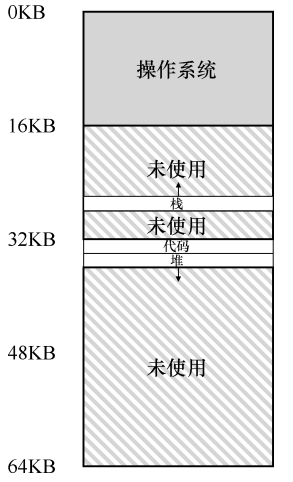


图 16.2 在物理内存中放置段

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

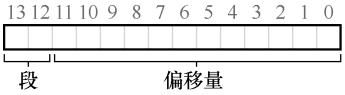
堆的基址（34KB），得到物理地址 39016，这不是正确的地址。我们首先应该先减去堆的偏移量，即该地址指的是这个段中的哪个字节。因为堆从虚拟地址 4K（4096）开始，4200 的偏移量实际上是 4200 减去 4096，即 104，然后用这个偏移量（104）加上基址寄存器中的物理地址（34KB），得到真正的物理地址 34920。

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）。

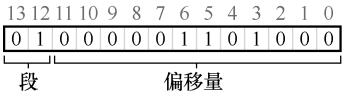
16.2 我们引用哪个段

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。

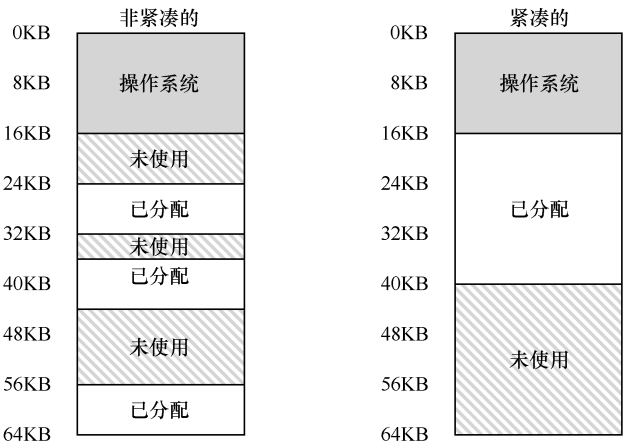


图 16.3 非紧凑和紧凑的内存

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

一篇关于分段及其一些用途的论文，发表时间非常早且难以阅读。

[I09]“Intel 64 and IA-32 Architectures Software Developer’s Manuals”Intel, 2009
尝试阅读这里的分段（第 3a 卷第 3 章），它会让你伤脑筋，至少有一点“头疼”。

[K68]“The Art of Computer Programming: Volume I”Donald Knuth
Addison-Wesley, 1968
Knuth 不仅因其早期关于计算机编程艺术的书而闻名，而且因其排版系统 TeX 而闻名。该系统仍然是当今专业人士使用的强大排版工具，并且排版了这本书。他关于算法的论述很早就引用了许多当今计算系统的算法。

[L83]“Hints for Computer Systems Design”Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
关于如何构建系统的宝贵建议。一下子读完这篇文章很难，每次读几页，就像品一杯美酒，或把它当作一本参考手册。

[LL82]“Virtual Memory Management in the VAX/VMS Operating System”Henry M. Levy and Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982)
一个经典的内存管理系统，在设计上有很多常识。我们将在后面的章节中更详细地研究它。

[RK68]“Dynamic Storage Allocation Systems”
B．Randell and C.J. Kuehner Communications of the ACM
Volume 11(5), pages 297-306, May 1968
对分页和分段两者的差异有一个很好的阐述，其中还有各种机器的历史讨论。

[R69]“A note on storage fragmentation and program segmentation”Brian Randell
Communications of the ACM
Volume 12(7), pages 365-372, July 1969
One of the earliest papers to discuss fragmentation.
最早讨论碎片问题的论文之一。

[W+95]“Dynamic Storage Allocation: A Survey and Critical Review”Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles In International Workshop on Memory Management
Scotland, United Kingdom, September 1995
一份关于内存分配程序的很棒的调查报告。

作业

该程序允许你查看在具有分段的系统中如何执行地址转换。详情请参阅README文件。

问题

1．先让我们用一个小地址空间来转换一些地址。这里有一组简单的参数和几个不同的随机种子。你可以转换这些地址吗？

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

2．现在，让我们看看是否理解了这个构建的小地址空间（使用上面问题的参数）。段 0 中最高的合法虚拟地址是什么？段 1 中最低的合法虚拟地址是什么？在整个地址空间中，最低和最高的非法地址是什么？最后，如何运行带有-A 标志的 segmentation.py 来测试你是否正确？

3．假设我们在一个 128 字节的物理内存中有一个很小的 16 字节地址空间。你会设置什么样的基址和界限，以便让模拟器为指定的地址流生成以下转换结果：有效，有效，违规，违反，有效，有效？假设用以下参数：

```
segmentation.py -a 16 -p 128
  -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
  --b0 ? --l0 ? --b1 ? --l1 ?
```

4．假设我们想要生成一个问题，其中大约 90%的随机生成的虚拟地址是有效的（即不产生段异常）。你应该如何配置模拟器来做到这一点？哪些参数很重要？

5．你可以运行模拟器，使所有虚拟地址无效吗？怎么做到？