第 13 章　抽象：地址空间

早期，构建计算机操作系统非常简单。你可能会问，为什么？因为用户对操作系统的期望不高。然而一些烦人的用户提出要“易于使用”“高性能”“可靠性”等，这导致了所有这些令人头痛的问题。下次你见到这些用户的时候，应该感谢他们，他们是这些问题的根源。

13.1　早期系统

从内存来看，早期的机器并没有提供多少抽象给用户。基本上，机器的物理内存看起来如图 13.1 所示。

操作系统曾经是一组函数（实际上是一个库），在内存中（在本例中，从物理地址 0 开始），然后有一个正在运行的程序（进程），目前在物理内存中（在本例中，从物理地址 64KB 开始），并使用剩余的内存。这里几乎没有抽象，用户对操作系统的要求也不多。那时候，操作系统开发人员的生活确实很容易，不是吗？

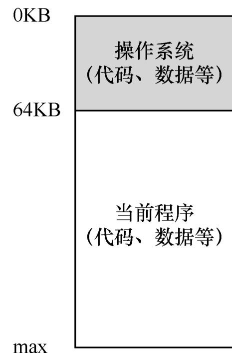

图 13.1　操作系统：早期

13.2　多道程序和时分共享

过了一段时间，由于机器昂贵，人们开始更有效地共享机器。因此，多道程序（multiprogramming）系统时代开启[DV66]，其中多个进程在给定时间准备运行，比如当有一个进程在等待 I/O 操作的时候，操作系统会切换这些进程，这样增加了 CPU 的有效利用率（utilization）。那时候，效率（efficiency）的提高尤其重要，因为每台机器的成本是数十万美元甚至数百万美元（现在你觉得你的 Mac 很贵！）

但很快，人们开始对机器要求更多，分时系统的时代诞生了[S59，L60，M62，M83]。具体来说，许多人意识到批量计算的局限性，尤其是程序员本身[CV65]，他们厌倦了长时间的（因此也是低效率的）编程—调试循环。交互性（interactivity）变得很重要，因为许多用户可能同时在使用机器，每个人都在等待（或希望）他们执行的任务及时响应。

一种实现时分共享的方法，是让一个进程单独占用全部内存运行一小段时间（见图 13.1），然后停止它，并将它所有的状态信息保存在磁盘上（包含所有的物理内存），加载其他进程的状态信息，再运行一段时间，这就实现了某种比较粗糙的机器共享[M+63]。

遗憾的是，这种方法有一个问题：太慢了，特别是当内存增长的时候。虽然保存和恢

复寄存器级的状态信息（程序计数器、通用寄存器等）相对较快，但将全部的内存信息保存到磁盘就太慢了。因此，在进程切换的时候，我们仍然将进程信息放在内存中，这样操作系统可以更有效率地实现时分共享（见图 13.2）。

图 13.2 3 个进程：共享内存

在图 13.2 中，有 3 个进程（A、B、C），每个进程拥有从 512KB 物理内存中切出来给它们的一小部分内存。假定只有一个 CPU，操作系统选择运行其中一个进程（比如 A），同时其他进程（B 和 C）则在队列中等待运行。

随着时分共享变得更流行，人们对操作系统又有了新的要求。特别是多个程序同时驻留在内存中，使保护（protection）成为重要问题。人们不希望一个进程可以读取其他进程的内存，更别说修改了。

13.3 地址空间

然而，我们必须将这些烦人的用户的需求放在心上。因此操作系统需要提供一个易用（easy to use）的物理内存抽象。这个抽象叫作地址空间（address space），是运行的程序看到的系统中的内存。理解这个基本的操作系统内存抽象，是了解内存虚拟化的关键。

一个进程的地址空间包含运行的程序的所有内存状态。比如：程序的代码（code，指令）必须在内存中，因此它们在地址空间里。当程序在运行的时候，利用栈（stack）来保存当前的函数调用信息，分配空间给局部变量，传递参数和函数返回值。最后，堆（heap）用于管理动态分配的、用户管理的内存，就像你从 C 语言中调用 malloc()或面向对象语言（如 C ++或 Java）中调用 new 获得内存。当然，还有其他的东西（例如，静态初始化的变量），但现在假设只有这 3 个部分：代码、栈和堆。

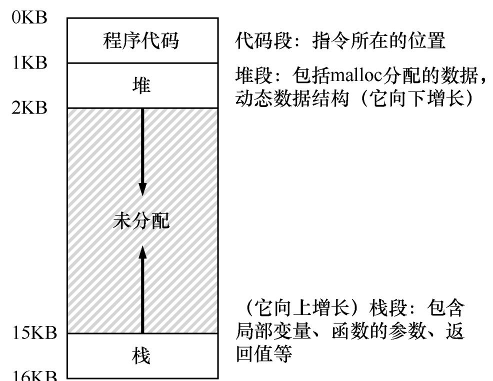

图 13.3 地址空间的例子

在图 13.3 的例子中，我们有一个很小的地址空间[①]（只有 16KB）。程序代码位于地址空间的顶部（在本例中从 0 开始，并且装入到地址空间的前 1KB）。代码是静态的（因此很容易放在内存中），所以可以将它放在地址空间的顶部，我们知道程序运行时不再需要新的空间。

接下来，在程序运行时，地址空间有两个区域可能增长（或者收缩）。它们就是堆（在顶部）和栈（在底部）。把它们放在那里，是因为它们都希望能够增长。通过将它们放在地址空间的两端，我们可以允许这样的增长：它们只需要在相反的方向增长。因此堆在代码（1KB）之下开始并向下增长（当用户通过 malloc()请求更多内存时），栈从 16KB 开始并向上增长

① 我们通常会使用这样的小例子，原因有二：①表示 32 位地址空间是一种痛苦；②数学计算更难。我们喜欢简单的数学。

（当用户进行程序调用时）。然而，堆栈和堆的这种放置方法只是一种约定，如果你愿意，可以用不同的方式安排地址空间 [稍后我们会看到，当多个线程（threads）在地址空间中共存时，就没有像这样分配空间的好办法了]。

当然，当我们描述地址空间时，所描述的是操作系统提供给运行程序的抽象（abstract）。程序不在物理地址 0～16KB 的内存中，而是加载在任意的物理地址。回顾图 13.2 中的进程 A、B 和 C，你可以看到每个进程如何加载到内存中的不同地址。因此问题来了：

关键问题：如何虚拟化内存

操作系统如何在单一的物理内存上为多个运行的进程（所有进程共享内存）构建一个私有的、可能很大的地址空间的抽象？

当操作系统这样做时，我们说操作系统在虚拟化内存（virtualizing memory），因为运行的程序认为它被加载到特定地址（例如 0）的内存中，并且具有非常大的地址空间（例如 32 位或 64 位）。现实很不一样。

例如，当图 13.2 中的进程 A 尝试在地址 0（我们将称其为虚拟地址，virtual address）执行加载操作时，然而操作系统在硬件的支持下，出于某种原因，必须确保不是加载到物理地址 0，而是物理地址 320KB（这是 A 载入内存的地址）。这是内存虚拟化的关键，这是世界上每一个现代计算机系统的基础。

提示：隔离原则

隔离是建立可靠系统的关键原则。如果两个实体相互隔离，这意味着一个实体的失败不会影响另一个实体。操作系统力求让进程彼此隔离，从而防止相互造成伤害。通过内存隔离，操作系统进一步确保运行程序不会影响底层操作系统的操作。一些现代操作系统通过将某些部分与操作系统的其他部分分离，实现进一步的隔离。这样的微内核（microkernel）[BH70，R+89，S+03] 可以比整体内核提供更大的可靠性。

13.4 目标

在这一章中，我们触及操作系统的工作——虚拟化内存。操作系统不仅虚拟化内存，还有一定的风格。为了确保操作系统这样做，我们需要一些目标来指导。以前我们已经看过这些目标（想想本章的前言），我们会再次看到它们，但它们肯定是值得重复的。

虚拟内存（VM）系统的一个主要目标是透明（transparency）[①]。操作系统实现虚拟内存的方式，应该让运行的程序看不见。因此，程序不应该感知到内存被虚拟化的事实，相反，程序的行为就好像它拥有自己的私有物理内存。在幕后，操作系统（和硬件）完成了所有的工作，让不同的工作复用内存，从而实现这个假象。

① 透明的这种用法有时令人困惑。一些学生认为“变得透明”意味着把所有事情都公之于众。在这里，“变得透明”意味着相反的情况：操作系统提供的假象不应该被应用程序看破。因此，按照通常的用法，透明系统是一个很难注意到的系统。

虚拟内存的另一个目标是效率（efficiency）。操作系统应该追求虚拟化尽可能高效（efficient），包括时间上（即不会使程序运行得更慢）和空间上（即不需要太多额外的内存来支持虚拟化）。在实现高效率虚拟化时，操作系统将不得不依靠硬件支持，包括 TLB 这样的硬件功能（我们将在适当的时候学习）。

最后，虚拟内存第三个目标是保护（protection）。操作系统应确保进程受到保护（protect），不会受其他进程影响，操作系统本身也不会受进程影响。当一个进程执行加载、存储或指令提取时，它不应该以任何方式访问或影响任何其他进程或操作系统本身的内存内容（即在它的地址空间之外的任何内容）。因此，保护让我们能够在进程之间提供隔离（isolation）的特性，每个进程都应该在自己的独立环境中运行，避免其他出错或恶意进程的影响。

补充：你看到的所有地址都不是真的

写过打印出指针的 C 程序吗？你看到的值（一些大数字，通常以十六进制打印）是虚拟地址（virtual address）。有没有想过你的程序代码在哪里找到？你也可以打印出来，是的，如果你可以打印它，它也是一个虚拟地址。实际上，作为用户级程序的程序员，可以看到的任何地址都是虚拟地址。只有操作系统，通过精妙的虚拟化内存技术，知道这些指令和数据所在的物理内存的位置。所以永远不要忘记：如果你在一个程序中打印出一个地址，那就是一个虚拟的地址。虚拟地址只是提供地址如何在内存中分布的假象，只有操作系统（和硬件）才知道物理地址。

这里有一个小程序，打印出 main() 函数（代码所在地方）的地址，由 malloc()返回的堆空间分配的值，以及栈上一个整数的地址：

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    int main(int argc, char *argv[]) {
4        printf("location of code  : %p\n", (void *) main);
5        printf("location of heap  : %p\n", (void *) malloc(1));
6        int x = 3;
7        printf("location of stack : %p\n", (void *) &x);
8        return x;
9    }
```

在 64 位的 Mac 上面运行时，我们得到以下输出：

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

从这里，你可以看到代码在地址空间开头，然后是堆，而栈在这个大型虚拟地址空间的另一端。所有这些地址都是虚拟的，并且将由操作系统和硬件翻译成物理地址，以便从真实的物理位置获取该地址的值。

在接下来的章节中，我们将重点介绍虚拟化内存所需的基本机制（mechanism），包括硬件和操作系统的支持。我们还将研究一些较相关的策略（policy），你会在操作系统中遇到它们，包括如何管理可用空间，以及在空间不足时哪些页面该释放。通过这些内容，你会逐渐理解现代虚拟内存系统真正的工作原理[①]。

① 或者，我们会说服你放弃课程。但请坚持下去，如果你坚持学完虚拟内存系统，很可能会坚持到底！

13.5 小结

我们介绍了操作系统的一个重要子系统：虚拟内存。虚拟内存系统负责为程序提供一个巨大的、稀疏的、私有的地址空间的假象，其中保存了程序的所有指令和数据。操作系统在专门硬件的帮助下，通过每一个虚拟内存的索引，将其转换为物理地址，物理内存根据获得的物理地址去获取所需的信息。操作系统会同时对许多进程执行此操作，并且确保程序之间互相不会受到影响，也不会影响操作系统。整个方法需要大量的机制（很多底层机制）和一些关键的策略。我们将自底向上，先描述关键机制。我们继续吧！

参考资料

[BH70]“The Nucleus of a Multiprogramming System”Per Brinch Hansen
Communications of the ACM, 13:4, April 1970
第一篇建议 OS 或内核应该是构建定制操作系统的最小且灵活的基础的论文，这个主题将在整个 OS 研究历史中重新被关注。

[CV65]“Introduction and Overview of the Multics System”
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
一篇卓越的早期 Multics 论文。下面是关于时分共享的一句名言：“时分共享的动力首先来自专业程序员，因为他们在批处理系统中调试程序时经常感到沮丧。因此，时分共享计算机最初的目标，是以允许几个人同时使用，并为他们每个人提供使用整台机器的假象。”

[DV66]“Programming Semantics for Multiprogrammed Computations”Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
关于多道程序系统的早期论文（但不是第一篇）。

[L60]“Man-Computer Symbiosis”
J. C. R. Licklider
IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960
一篇关于计算机和人类如何进入共生时代的趣味论文，显然超越了它的时代，但仍然令人着迷。

[M62]“Time-Sharing Computer Systems”
J. McCarthy
Management and the Computer of the Future, MIT Press, Cambridge, Mass, 1962
可能是 McCarthy 最早的关于时分共享的论文。然而，在另一篇论文[M83]中，他声称自 1957 年以来一直在思考这个想法。McCarthy 离开了系统领域，并在斯坦福大学成为人工智能领域的巨人，其工作包括创建

LISP 编程语言。查看 McCarthy 的主页可以了解更多信息。

[M+63]“A Time-Sharing Debugging System for a Small Computer”
J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider AFIPS ’63 (Spring), New York, NY, May 1963
这是一个很好的早期系统例子，当程序没有运行时将程序存储器交换到“鼓”，然后在运行时回到“核心”存储器。

[M83]“Reminiscences on the History of Time Sharing”John McCarthy
Winter or Spring of 1983
关于时分共享思想可能来自何处的一个了不起的历史记录，包括针对那些引用 Strachey 的作品[S59]作为这一领域开拓性工作的人的一些怀疑。

[NS07]“Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”Nicholas Nethercote and Julian Seward
PLDI 2007, San Diego, California, June 2007
对于那些使用 C 这样的不安全语言的人来说，Valgrind 是程序的救星。阅读本文以了解其非常酷的二进制探测技术——这真是令人印象深刻。

[R+89]“Mach: A System Software kernel”
Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones
COMPCON 89, February 1989
尽管这不是微内核的第一个项目，但 CMU 的 Mach 项目是众所周知的、有影响力的。它仍然深深扎根于 macOS X 的深处。

[S59]“Time Sharing in Large Fast Computers”
C. Strachey
Proceedings of the International Conference on Information Processing, UNESCO, June 1959
关于时分共享的最早参考文献之一。

[S+03]“Improving the Reliability of Commodity Operating Systems”Michael M. Swift, Brian N. Bershad, Henry M. Levy
SOSP 2003
第一篇介绍微内核思想如何提高操作系统可靠性的论文。