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。

第 14 章　插叙：内存操作 API

在本章中，我们将介绍 UNIX 操作系统的内存分配接口。操作系统提供的接口非常简洁，因此本章简明扼要[①]。本章主要关注的问题是：

关键问题：如何分配和管理内存

在 UNIX/C 程序中，理解如何分配和管理内存是构建健壮和可靠软件的重要基础。通常使用哪些接口？哪些错误需要避免？

14.1　内存类型

在运行一个 C 程序的时候，会分配两种类型的内存。第一种称为栈内存，它的申请和释放操作是编译器来隐式管理的，所以有时也称为自动（automatic）内存。

C 中申请栈内存很容易。比如，假设需要在 func()函数中为一个整形变量 *x* 申请空间。为了声明这样的一块内存，只需要这样做：

```
void func() {
    int x; // declares an integer on the stack
    ...
}
```

编译器完成剩下的事情，确保在你进入 func() 函数的时候，在栈上开辟空间。当你从该函数退出时，编译器释放内存。因此，如果你希望某些信息存在于函数调用之外，建议不要将它们放在栈上。

就是这种对长期内存的需求，所以我们才需要第二种类型的内存，即所谓的堆（heap）内存，其中所有的申请和释放操作都由程序员显式地完成。毫无疑问，这是一项非常艰巨的任务！这确实导致了很多缺陷。但如果小心并加以注意，就会正确地使用这些接口，没有太多的麻烦。下面的例子展示了如何在堆上分配一个整数，得到指向它的指针：

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

关于这一小段代码有两点说明。首先，你可能会注意到栈和堆的分配都发生在这一行：首先编译器看到指针的声明（int * x）时，知道为一个整型指针分配空间，随后，当程序调

① 实际上，我们希望所有章节都简明扼要！但我们认为，本章更简明、更扼要。

用 malloc()时，它会在堆上请求整数的空间，函数返回这样一个整数的地址（成功时，失败时则返回 NULL），然后将其存储在栈中以供程序使用。

因为它的显式特性，以及它更富于变化的用法，堆内存对用户和系统提出了更大的挑战。所以这也是我们接下来讨论的重点。

14.2 malloc()调用

malloc 函数非常简单：传入要申请的堆空间的大小，它成功就返回一个指向新申请空间的指针，失败就返回 NULL[①]。

man 手册展示了使用 malloc 需要怎么做，在命令行输入 man malloc，你会看到：

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

从这段信息可以看到，只需要包含头文件 stdlib.h 就可以使用 malloc 了。但实际上，甚至都不需这样做，因为 C 库是 C 程序默认链接的，其中就有 mallock()的代码，加上这个头文件只是让编译器检查你是否正确调用了 malloc()（即传入参数的数目正确且类型正确）。

malloc 只需要一个 size_t 类型参数，该参数表示你需要多少个字节。然而，大多数程序员并不会直接传入数字（比如 10）。实际上，这样做会被认为是不太好的形式。替代方案是使用各种函数和宏。例如，为了给双精度浮点数分配空间，只要这样：

```
double *d = (double *) malloc(sizeof(double));
```

提示：如果困惑，动手试试

如果你不确定要用的一些函数或者操作符的行为，唯一的办法就是试一下，确保它的行为符合你的期望。虽然读手册或其他文档是有用的，但在实际中如何使用更为重要。实际上，我们正是通过这样做，来确保关于 sizeof()我们所说的都是真的！

啊，好多 double！对 malloc()的调用使用 sizeof() 操作符去申请正确大小的空间。在 C 中，这通常被认为是编译时操作符，意味着这个大小是在编译时就已知道，因此被替换成一个数（在本例中是 8，对于 double），作为 malloc()的参数。出于这个原因，sizeof() 被正确地认为是一个操作符，而不是一个函数调用（函数调用在运行时发生）。

你也可以传入一个变量的名字（而不只是类型）给 sizeof()，但在一些情况下，可能得不到你要的结果，所以要小心使用。例如，看看下面的代码片段：

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

在第一行，我们为 10 个整数的数组声明了空间，这很好，很漂亮。但是，当我们在下一行使用 sizeof()时，它将返回一个较小的值，例如 4（在 32 位计算机上）或 8（在 64 位计

① 请注意，C 中的 NULL 实际上并不是什么特别的东西，只是一个值为 0 的宏。

算机上）。原因是在这种情况下，sizeof()认为我们只是问一个整数的指针有多大，而不是我们动态分配了多少内存。但是，有时 sizeof()的确如你所期望的那样工作：

```
int x[10];
printf("%d\n", sizeof(x));
```

在这种情况下，编译器有足够的静态信息，知道已经分配了 40 个字节。

另一个需要注意的地方是使用字符串。如果为一个字符串声明空间，请使用以下习惯用法：malloc(strlen(s) + 1)，它使用函数 strlen()获取字符串的长度，并加上 1，以便为字符串结束符留出空间。这里使用 sizeof()可能会导致麻烦。

你也许还注意到 malloc()返回一个指向 void 类型的指针。这样做只是 C 中传回地址的方式，让程序员决定如何处理它。程序员将进一步使用所谓的强制类型转换（cast），在我们上面的示例中，程序员将返回类型的 malloc()强制转换为指向 double 的指针。强制类型转换实际上没干什么事，只是告诉编译器和其他可能正在读你的代码的程序员：“是的，我知道我在做什么。”通过强制转换 malloc()的结果，程序员只是在给人一些信心，强制转换不是程序正确所必须的。

14.3 free()调用

事实证明，分配内存是等式的简单部分。知道何时、如何以及是否释放内存是困难的部分。要释放不再使用的堆内存，程序员只需调用 free()：

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

该函数接受一个参数，即一个由 malloc()返回的指针。

因此，你可能会注意到，分配区域的大小不会被用户传入，必须由内存分配库本身记录追踪。

14.4 常见错误

在使用 malloc()和 free()时会出现一些常见的错误。以下是我们在教授本科操作系统课程时反复看到的情形。所有这些例子都可以通过编译器的编译并运行。对于构建一个正确的 C 程序来说，通过编译是必要的，但这远远不够，你会懂的（通常在吃了很多苦头之后）。

实际上，正确的内存管理就是这样一个问题，许多新语言都支持自动内存管理（automatic memory management）。在这样的语言中，当你调用类似 malloc()的机制来分配内存时（通常用 new 或类似的东西来分配一个新对象），你永远不需要调用某些东西来释放空间。实际上，垃圾收集器（garbage collector）会运行，找出你不再引用的内存，替你释放它。

忘记分配内存

许多例程在调用之前，都希望你为它们分配内存。例如，例程 strcpy(dst, src)将源字符串中的字符串复制到目标指针。但是，如果不小心，你可能会这样做：

```
char *src = "hello";
char *dst;        // oops! unallocated
strcpy(dst, src); // segfault and die
```

运行这段代码时，可能会导致段错误（segmentation fault）[①]，这是一个很奇怪的术语，表示“你对内存犯了一个错误。你这个愚蠢的程序员。我很生气。”

提示：它编译过了或它运行了!=它对了

仅仅因为程序编译过了甚至正确运行了一次或多次，并不意味着程序是正确的。许多事件可能会让你相信它能工作，但是之后有些事情会发生变化，它停止了。学生常见的反应是说（或者叫喊）“但它以前是好的!”，然后责怪编译器、操作系统、硬件，甚至是（我们敢说）教授。但是，问题通常就像你认为的那样，在你的代码中。在指责别人之前，先撸起袖子调试一下。

在这个例子中，正确的代码可能像这样：

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

或者你可以用 strdup()，让生活更加轻松。阅读 strdup 的 man 手册页，了解更多信息。

没有分配足够的内存

另一个相关的错误是没有分配足够的内存，有时称为缓冲区溢出（buffer overflow）。在上面的例子中，一个常见的错误是为目标缓冲区留出“几乎”足够的空间。

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

奇怪的是，这个程序通常看起来会正确运行，这取决于如何实现 malloc 和许多其他细节。在某些情况下，当字符串拷贝执行时，它会在超过分配空间的末尾处写入一个字节，但在某些情况下，这是无害的，可能会覆盖不再使用的变量。在某些情况下，这些溢出可能具有令人难以置信的危害，实际上是系统中许多安全漏洞的来源[W06]。在其他情况下，malloc 库总是分配一些额外的空间，因此你的程序实际上不会在其他某个变量的值上涂写，并且工作得很好。还有一些情况下，该程序确实会发生故障和崩溃。因此，我们学到了另一个宝贵的教训：即使它正确运行过一次，也不意味着它是正确的。

① 尽管听起来很神秘，但你很快就会明白为什么这种非法的内存访问被称为段错误。如果这都不能刺激你继续读下去，那什么能呢？

忘记初始化分配的内存

在这个错误中，你正确地调用 malloc()，但忘记在新分配的数据类型中填写一些值。不要这样做！如果你忘记了，你的程序最终会遇到未初始化的读取（uninitialized read），它从堆中读取了一些未知值的数据。谁知道那里可能会有什么？如果走运，读到的值使程序仍然有效（例如，零）。如果不走运，会读到一些随机和有害的东西。

忘记释放内存

另一个常见错误称为内存泄露（memory leak），如果忘记释放内存，就会发生。在长时间运行的应用程序或系统（如操作系统本身）中，这是一个巨大的问题，因为缓慢泄露的内存会导致内存不足，此时需要重新启动。因此，一般来说，当你用完一段内存时，应该确保释放它。请注意，使用垃圾收集语言在这里没有什么帮助：如果你仍然拥有对某块内存的引用，那么垃圾收集器就不会释放它，因此即使在较现代的语言中，内存泄露仍然是一个问题。

在某些情况下，不调用 free()似乎是合理的。例如，你的程序运行时间很短，很快就会退出。在这种情况下，当进程死亡时，操作系统将清理其分配的所有页面，因此不会发生内存泄露。虽然这肯定“有效”（请参阅后面的补充），但这可能是一个坏习惯，所以请谨慎选择这样的策略。长远来看，作为程序员的目标之一是养成良好的习惯。其中一个习惯是理解如何管理内存，并在 C 这样的语言中，释放分配的内存块。即使你不这样做也可以逃脱惩罚，建议还是养成习惯，释放显式分配的每个字节。

在用完之前释放内存

有时候程序会在用完之前释放内存，这种错误称为悬挂指针（dangling pointer），正如你猜测的那样，这也是一件坏事。随后的使用可能会导致程序崩溃或覆盖有效的内存（例如，你调用了 free()，但随后再次调用 malloc()来分配其他内容，这重新利用了错误释放的内存）。

反复释放内存

程序有时还会不止一次地释放内存，这被称为重复释放（double free）。这样做的结果是未定义的。正如你所能想象的那样，内存分配库可能会感到困惑，并且会做各种奇怪的事情，崩溃是常见的结果。

错误地调用 free()

我们讨论的最后一个问题是 free()的调用错误。毕竟，free()期望你只传入之前从 malloc()得到的一个指针。如果传入一些其他的值，坏事就可能发生（并且会发生）。因此，这种无效的释放（invalid free）是危险的，当然也应该避免。

补充：为什么在你的进程退出时没有内存泄露

当你编写一个短时间运行的程序时，可能会使用 malloc()分配一些空间。程序运行并即将完成：是否需要在退出前调用几次 free()？虽然不释放似乎不对，但在真正的意义上，没有任何内存会“丢失”。原因很简单：系统中实际存在两级内存管理。

第一级是由操作系统执行的内存管理，操作系统在进程运行时将内存交给进程，并在进程退出（或以其他方式结束）时将其回收。第二级管理在每个进程中，例如在调用 malloc()和 free()时，在堆内管理。即使你没有调用 free()（并因此泄露了堆中的内存），操作系统也会在程序结束运行时，收回进程的所有内存（包括用于代码、栈，以及相关堆的内存页）。无论地址空间中堆的状态如何，操作系统都会在进程终止时收回所有这些页面，从而确保即使没有释放内存，也不会丢失内存。

因此，对于短时间运行的程序，泄露内存通常不会导致任何操作问题（尽管它可能被认为是不好的形式）。如果你编写一个长期运行的服务器（例如 Web 服务器或数据库管理系统，它永远不会退出），泄露内存就是很大的问题，最终会导致应用程序在内存不足时崩溃。当然，在某个程序内部泄露内存是一个更大的问题：操作系统本身。这再次向我们展示：编写内核代码的人，工作是辛苦的……

小结

如你所见，有很多方法滥用内存。由于内存出错很常见，整个工具生态圈已经开发出来，可以帮助你在代码中找到这些问题。请查看 purify [HJ92]和 valgrind [SN05]，在帮助你找到与内存有关的问题的根源方面，两者都非常出色。一旦你习惯于使用这些强大的工具，就会想知道，没有它们时，你是如何活下来的。

14.5 底层操作系统支持

你可能已经注意到，在讨论 malloc()和 free()时，我们没有讨论系统调用。原因很简单：它们不是系统调用，而是库调用。因此，malloc 库管理虚拟地址空间内的空间，但是它本身是建立在一些系统调用之上的，这些系统调用会进入操作系统，来请求更多内存或者将一些内容释放回系统。

一个这样的系统调用叫作 brk，它被用来改变程序分断（break）的位置：堆结束的位置。它需要一个参数（新分断的地址），从而根据新分断是大于还是小于当前分断，来增加或减小堆的大小。另一个调用 sbrk 要求传入一个增量，但目的是类似的。

请注意，你不应该直接调用 brk 或 sbrk。它们被内存分配库使用。如果你尝试使用它们，很可能会犯一些错误。建议坚持使用 malloc()和 free()。

最后，你还可以通过 mmap()调用从操作系统获取内存。通过传入正确的参数，mmap()可以在程序中创建一个匿名（anonymous）内存区域——这个区域不与任何特定文件相关联，而是与交换空间（swap space）相关联，稍后我们将在虚拟内存中详细讨论。这种内存也可以像堆一样对待并管理。阅读 mmap()的手册页以获取更多详细信息。

14.6 其他调用

内存分配库还支持一些其他调用。例如，calloc()分配内存，并在返回之前将其置零。如果你认为内存已归零并忘记自己初始化它，这可以防止出现一些错误（请参阅 14.4 节中“忘记初始化分配的内存”的内容）。当你为某些东西（比如一个数组）分配空间，然后需要添加一些东西时，例程 realloc()也会很有用：realloc()创建一个新的更大的内存区域，将旧区域复制到其中，并返回新区域的指针。

14.7 小结

我们介绍了一些处理内存分配的 API。与往常一样，我们只介绍了基本知识。更多细节可在其他地方获得。请阅读 C 语言的书[KR88]和 Stevens [SR05]（第 7 章）以获取更多信息。有关如何自动检测和纠正这些问题的很酷的现代论文，请参阅 Novark 等人的论文[N+07]。这篇文章还包含了对常见问题的很好的总结，以及关于如何查找和修复它们的一些简洁办法。

参考资料

[HJ92] Purify: Fast Detection of Memory Leaks and Access Errors
R. Hastings and B. Joyce USENIX Winter ’92
很酷的 Purify 工具背后的文章。Purify 现在是商业产品。

[KR88]“The C Programming Language”Brian Kernighan and Dennis Ritchie Prentice-Hall 1988
C 之书，由 C 的开发者编写。读一遍，编一些程序，然后再读一遍，让它成为你的案头手册。

[N+07]“Exterminator: Automatically Correcting Memory Errors with High Probability”Gene Novark, Emery D. Berger, and Benjamin G. Zorn
PLDI 2007
一篇很酷的文章，包含自动查找和纠正内存错误，以及 C 和 C ++程序中许多常见错误的概述。

[SN05]“Using Valgrind to Detect Undefined Value Errors with Bit-precision”
J. Seward and N. Nethercote USENIX ’05
如何使用 valgrind 来查找某些类型的错误。

[SR05]“Advanced Programming in the UNIX Environment”
W. Richard Stevens and Stephen A. Rago Addison-Wesley, 2005
我们之前已经说过了，这里再重申一遍：读这本书很多遍，并在有疑问时将其用作参考。本书的两位作者

总是很惊讶，每次读这本书时都会学到一些新东西，即使具有多年的 C 语言编程经验的程序员。

[W06]“Survey on Buffer Overflow Attacks and Countermeasures”Tim Werthman
一份很好的调查报告，关于缓冲区溢出及其造成的一些安全问题。文中指出了许多著名的漏洞。

作业（编码）

在这个作业中，你会对内存分配有所了解。首先，你会写一些错误的程序（好玩!）。然后，利用一些工具来帮助你找到其中的错误。最后，你会意识到这些工具有多棒，并在将来使用它们，从而使你更加快乐和高效。

你要使用的第一个工具是调试器 gdb。关于这个调试器有很多需要了解的知识，在这里，我们只是浅尝辄止。

你要使用的第二个工具是 valgrind [SN05]。该工具可以帮助查找程序中的内存泄露和其他隐藏的内存问题。如果你的系统上没有安装，请访问 valgrind 网站并安装它。

问题

1．首先，编写一个名为 null.c 的简单程序，它创建一个指向整数的指针，将其设置为 NULL，然后尝试对其进行释放内存操作。把它编译成一个名为 null 的可执行文件。当你运行这个程序时会发生什么？

2．接下来，编译该程序，其中包含符号信息（使用-g 标志）。这样做可以将更多信息放入可执行文件中，使调试器可以访问有关变量名称等的更多有用信息。通过输入 gdb null，在调试器下运行该程序，然后，一旦 gdb 运行，输入 run。gdb 显示什么信息？

3．最后，对这个程序使用 valgrind 工具。我们将使用属于 valgrind 的 memcheck 工具来分析发生的情况。输入以下命令来运行程序：valgrind --leak-check=yes null。当你运行它时会发生什么？你能解释工具的输出吗？

4．编写一个使用 malloc()来分配内存的简单程序，但在退出之前忘记释放它。这个程序运行时会发生什么？你可以用 gdb 来查找它的任何问题吗？用 valgrind 呢（再次使用 --leak-check=yes 标志）？

5．编写一个程序，使用 malloc 创建一个名为 data、大小为 100 的整数数组。然后，将 data[100]设置为 0。当你运行这个程序时会发生什么？当你使用 valgrind 运行这个程序时会发生什么？程序是否正确？

6．创建一个分配整数数组的程序（如上所述），释放它们，然后尝试打印数组中某个元素的值。程序会运行吗？当你使用 valgrind 时会发生什么？

7．现在传递一个有趣的值来释放（例如，在上面分配的数组中间的一个指针）。会发生什么？你是否需要工具来找到这种类型的问题？

8．尝试一些其他接口来分配内存。例如，创建一个简单的向量似的数据结构，以及使用 realloc()来管理向量的相关函数。使用数组来存储向量元素。当用户在向量中添加条目时，请使用realloc()为其分配更多空间。这样的向量表现如何？它与链表相比如何？使用valgrind来帮助你发现错误。

9．花更多时间阅读有关使用 gdb 和 valgrind 的信息。了解你的工具至关重要，花时间学习如何成为 UNIX 和 C 环境中的调试器专家。

第 15 章　机制：地址转换

在实现 CPU 虚拟化时，我们遵循的一般准则被称为受限直接访问（Limited Direct Execution，LDE）。LDE 背后的想法很简单：让程序运行的大部分指令直接访问硬件，只在一些关键点（如进程发起系统调用或发生时钟中断）由操作系统介入来确保“在正确时间，正确的地点，做正确的事”。为了实现高效的虚拟化，操作系统应该尽量让程序自己运行，同时通过在关键点的及时介入（interposing），来保持对硬件的控制。高效和控制是现代操作系统的两个主要目标。

在实现虚拟内存时，我们将追求类似的战略，在实现高效和控制的同时，提供期望的虚拟化。高效决定了我们要利用硬件的支持，这在开始的时候非常初级（如使用一些寄存器），但会变得相当复杂（比如我们会讲到的 TLB、页表等）。控制意味着操作系统要确保应用程序只能访问它自己的内存空间。因此，要保护应用程序不会相互影响，也不会影响操作系统，我们需要硬件的帮助。最后，我们对虚拟内存还有一点要求，即灵活性。具体来说，我们希望程序能以任何方式访问它自己的地址空间，从而让系统更容易编程。所以，关键问题在于：

关键问题：如何高效、灵活地虚拟化内存

如何实现高效的内存虚拟化？如何提供应用程序所需的灵活性？如何保持控制应用程序可访问的内存位置，从而确保应用程序的内存访问受到合理的限制？如何高效地实现这一切？

我们利用了一种通用技术，有时被称为基于硬件的地址转换（hardware-based address translation），简称为地址转换（address translation）。它可以看成是受限直接执行这种一般方法的补充。利用地址转换，硬件对每次内存访问进行处理（即指令获取、数据读取或写入），将指令中的虚拟（virtual）地址转换为数据实际存储的物理（physical）地址。因此，在每次内存引用时，硬件都会进行地址转换，将应用程序的内存引用重定位到内存中实际的位置。

当然，仅仅依靠硬件不足以实现虚拟内存，因为它只是提供了底层机制来提高效率。操作系统必须在关键的位置介入，设置好硬件，以便完成正确的地址转换。因此它必须管理内存（manage memory），记录被占用和空闲的内存位置，并明智而谨慎地介入，保持对内存使用的控制。

同样，所有这些工作都是为了创造一种美丽的假象：每个程序都拥有私有的内存，那里存放着它自己的代码和数据。虚拟现实的背后是丑陋的物理事实：许多程序其实是在同一时间共享着内存，就像 CPU（或多个 CPU）在不同的程序间切换运行。通过虚拟化，操作系统（在硬件的帮助下）将丑陋的机器现实转化成一种有用的、强大的、易于使用的抽象。

15.1 假设

我们对内存虚拟化的第一次尝试非常简单，甚至有点可笑。如果你觉得可笑就笑吧，很快就轮到操作系统嘲笑你了。当你试图理解 TLB 的换入换出、多级页表，和其他技术一样有奇迹之处的时候。不喜欢操作系统嘲笑你？很不幸，但这就是操作系统的运行方式。

具体来说，我们先假设用户的地址空间必须连续地放在物理内存中。同时，为了简单，我们假设地址空间不是很大，具体来说，小于物理内存的大小。最后，假设每个地址空间的大小完全一样。别担心这些假设听起来不切实际，我们会逐步地放宽这些假设，从而得到现实的内存虚拟化。

15.2 一个例子

为了更好地理解实现地址转换需要什么，以及为什么需要，我们先来看一个简单的例子。设想一个进程的地址空间如图 15.1 所示。这里我们要检查一小段代码，它从内存中加载一个值，对它加 3，然后将它存回内存。你可以设想，这段代码的 C 语言形式可能像这样：

```
void func() {
    int x;
    x = x + 3; // this is the line of code we are interested in
```

编译器将这行代码转化为汇编语句，可能像下面这样（x86 汇编）。我们可以用 Linux 的 objdump 或者 Mac 的 otool 将它反汇编：

```
128: movl 0x0(%ebx), %eax   ;load 0+ebx into eax
132: addl $0x03, %eax       ;add 3 to eax register
135: movl %eax, 0x0(%ebx)   ;store eax back to mem
```

这段代码相对简单，它假定 *x* 的地址已经存入寄存器 ebx，之后通过 movl 指令将这个地址的值加载到通用寄存器 eax（长字移动）。下一条指令对 eax 的内容加 3。最后一条指令将 eax 中的值写回到内存的同一位置。

提示：介入（Interposition）很强大

介入是一种很常见又很有用的技术，计算机系统中使用介入常常能带来很好的效果。在虚拟内存中，硬件可以介入到每次内存访问中，将进程提供的虚拟地址转换为数据实际存储的物理地址。但是，一般化的介入技术有更广阔的应用空间，实际上几乎所有良好定义的接口都应该提供功能介入机制，以便增加功能或者在其他方面提升系统。这种方式最基本的优点是透明（transparency），介入完成时通常不需要改动接口的客户端，因此客户端不需要任何改动。

在图 15.1 中，可以看到代码和数据都位于进程的地址空间，3 条指令序列位于地址 128（靠近头部的代码段），变量 *x* 的值位于地址 15KB（在靠近底部的栈中）。如图 15.1 所示，*x* 的初始值是 3000。

如果这 3 条指令执行，从进程的角度来看，发生了以下几次内存访问：

- 从地址 128 获取指令；
- 执行指令（从地址 15KB 加载数据）；
- 从地址 132 获取命令；
- 执行命令（没有内存访问）；
- 从地址 135 获取指令；
- 执行指令（新值存入地址 15KB）。

从程序的角度来看，它的地址空间（address space）从 0 开始到 16KB 结束。它包含的所有内存引用都应该在这个范围内。然而，对虚拟内存来说，操作系统希望将这个进程地址空间放在物理内存的其他位置，并不一定从地址 0 开始。因此我们遇到了如下问题：怎样在内存中重定位这个进程，同时对该进程透明（transparent）？怎么样提供一种虚拟地址空间从 0 开始的假象，而实际上地址空间位于另外某个物理地址？

图 15.2 展示了一个例子，说明这个进程的地址空间被放入物理内存后可能的样子。从图 15.2 中可以看到，操作系统将第一块物理内存留给了自己，并将上述例子中的进程地址空间重定位到从 32KB 开始的物理内存地址。剩下的两块内存空闲（16～32KB 和 48～64KB）。

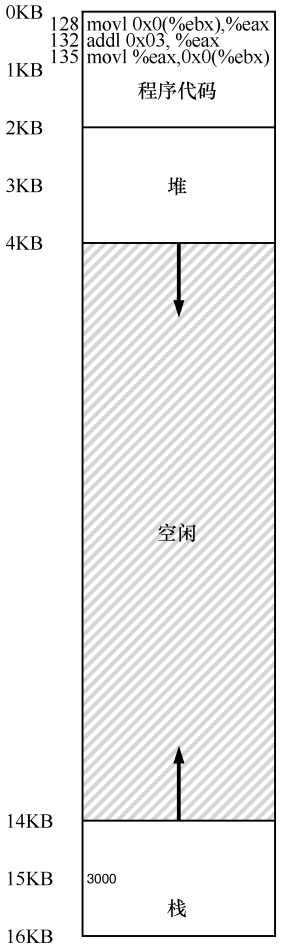

图 15.1 进程及其地址空间

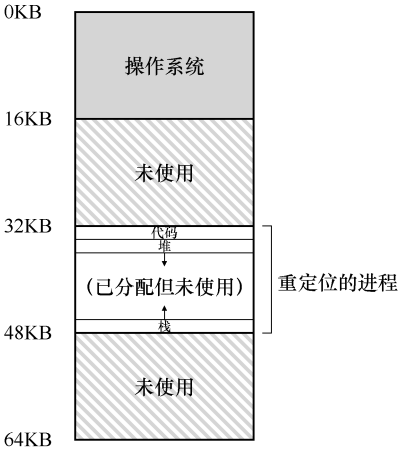

图 15.2 物理内存和单个重定位的进程