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。