OS 之道

操作系统简介

版本 0.7.4

Allen Downey(著)

杜文斌 (译)

Think OS

A Brief Introduction to Operating Systems

Version 0.7.4

Allen B. Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

The LaTeX source for this book is available from `http://greenteapress.com/thinkos`.

序

在很多计算机科学课程中, 操作系统始终是一个高级话题. 学生在学习它之前, 可能已经学习了如何用 C 编程, 甚至很可能已经学过计算机体系架构的课程. 通常本课程的目标是让学生了解操作系统的设计和实现, 同时期待其中一些学生在这个领域持续研究, 或者编写部分操作系统.

这本书希望可以满足不同人群的不同目标, 我编写此书是为了讲授欧林学院的软件系统的课程.

多数来上课的学生已经学过了如何用 Python 编程. 所以一个目标是能够帮助他们学习 C 语言. 对于这部分内容, 我使用 O'Reilly Media 出版社出版, Griffiths 和 Griffiths 编写的 *Head First C* 一书, 来充实本书的内容.

我的学生中很少会从事编写操作系统, 但是很多人将用 C 来编写底层应用, 以及进入嵌入式领域工作. 本课程包括操作系统, 网络, 数据库, 以及嵌入式系统, 但同时着重强调了程序员需要了解的主题.

本书不会假设你学过计算机架构. 随着课程学习, 我将会解释我们所需要了解的内容.

如果这本书是成功的, 那么它应该能够让你更好地理解程序运行时发生了什么, 以及做什么可以令程序运行更加高效.

第一章介绍了编译语言和解释语言的差异, 以及编译器工作原理. 推荐阅读: *Head First C* 第 1 章.

第二章解释了操作系统如何使用进程隔离程序运行.

第三章解释了虚拟内存和地址转换. 推荐阅读: *Head First C* 第 2 章.

第四章介绍了文件系统和数据流. 推荐阅读: *Head First C* 第 3 章.

第五章讲述了数字、字母和其他值如何编码, 并介绍了位运算符.

第六章介绍了动态内存管理的使用及其工作原理. 推荐阅读: *Head First C* 第 6 章.

第七章介绍了缓存和内存层次结构.

第八章介绍了多任务和调度.

第九章介绍了 POSIX 线程和互斥锁. 推荐阅读: *Head First C* 第 12 章以及 *Little Book of Semaphores* 第 1 和 2 章.

第十章介绍了 POSIX 条件变量和生产者/消费者问题.

第十一章介绍了 POSIX 信号量的使用以及在 C 语言中的实现.

补充

本书目前还是早期草稿版本. 我还在努力撰写文本, 但尚缺乏一些配图. 所以某些地方, 如果配图完备了, 我确信相关解释会得到改善.

0.1 代码使用

本书的代码示例可以从 `https://github.com/AllenDowney/ThinkOS`下载. Git 是个版本管理系统, 可以让你跟踪项目文件. Git 管理的文件集合叫做**仓库**. GitHub 是一个提供 Git 仓库存储以及友好 web 界面的托管服务.

我的仓库的 GitHub 主页, 提供了多种使用此代码的方法:

- 你可以点击 Fork 按钮, 在 GitHub 基于我的仓库创建一个副本. 如果你还没有 GitHub 账户, 你需要创建一个. 创建副本后, 你便拥有了自己的仓库, 便可以跟踪学习本书时所编写的代码了. 然后, 你可以克隆这个仓库, 这意味着文件会被复制到你的计算机上.
- 或者你也可以克隆我的仓库. 这样便无需 GitHub 账号, 但你便不能将你的变更写回 GitHub 了.
- 如果你一点都不想用 Git, 你可以使用 GitHub 页面的按钮下载 Zip 文件.

贡献者清单

如果你有建议或更正, 请发送邮件至 `downey@allendowney.com`. 如果我根据你的反馈进行更正, 我将把您添加到贡献者清单 (除非你要求省略).

如果您在反馈时包含了出现错误的部分语句, 那么我便可以快速定位. 提供页码或者章节编号也可以, 只是不太容易处理. 谢谢!

- 感谢参加 Olin 学院 2014 年春季软件系统课程的学生们, 他们在本书的早期草稿阶段, 进行了大量测试. 同时纠正了很多错误, 并提出相当多的建议. 我非常感激他们的开创精神!
- James P Giannoules 指出了一处复制粘贴错误.
- Andy Engle 指出了 GB 和 GiB 之间的区别.
- Aashish Karki 指出了一些语法问题.

其他发现错别字和其他错误的人包括 Jim Tyson, Donald Robertson, Jeremy Vermast, Yuzhong Huang 和 Ian Hill.

目录

Chapter 1

编译

1.1 编译语言和解释语言

人们通常将编程语言分为编译型或解释型. “编译” 意味着程序被翻译为机器语言, 然后硬件执行; “解释” 则意味着程序由软件解释器读取并执行. 通常认为 C 是编译型语言, 而 Python 是解释型语言. 但是两者差异并非泾渭分明.

首先, 很多语言是既可以被编译也可以被解释的. 例如, 也有 C 解释器和 Python 编译器. 其次, 像 Java 一样, 有些语言会使用混合方法, 将程序编译为一种中间语言, 然后在解释器中运行翻译后的程序. Java 使用了一种叫做 Java 字节码的中间语言, 和机器语言类似, 但它需要由软件解释器执行, 也就是 Java 虚拟机 (JVM).

所以被编译还是被解释, 并不是一个语言的内在特征; 然而无论如何, 编译语言和解释语言之间确实会存在一些普遍的差异.

1.2 静态类型

多数解释语言支持的是动态类型, 但是编译语言通常受限于静态类型. 在静态类型语言中, 你可以通过查看程序便能区分变量引用的类型. 而在动态类型语言中, 直到程序运行, 你才能确切知道变量类型. 通常, **静态** 是指在编译时发生的事情, 而**动态** 是指运行时发生的事情.

例如, 可以用 Python 编写如下函数:

```
def add(x, y):
    return x + y
```

看这个代码, 你很难在运行时确定 `x` 和 `y` 所引用的类型. 这个函数可能被调用多次, 每次都可能是不同类型的值. 任何支持加法运算的值都可以运行; 其他类型则会引发异常或**运行时错误**.

而在 C 中, 同样函数编写如下::

```
int add(int x, int y) {
    return x + y;
}
```

函数第一行包括对参数和返回值的**类型声明**: `x` 和 `y` 被声明为了整数, 这就意味着在编译时, 我们可以检查这种类型是否支持加法运算 (此处支持). 返回值也同样被声明为了整数.

因为这些声明的存在, 在程序其他位置调用这个函数时, 编译器便可以检查提供的参数是否具有正确的类型, 以及返回值是否使用正确.

这些检查发生在程序运行之前, 所以错误可以被提前发现. 更重要的, 可以在未执行的程序部分发现错误. 此外, 这些检查在运行时便无需发生, 这也是编译语言比解释语言运行更快的一个原因.

编译时声明类型还能节省空间. 在动态语言中, 程序运行时, 变量名称需要保存在内存中, 同时它们会被程序频繁访问. 例如, 在 Python 中, 内置函数 `locals` 会返回一个包含变量名称的字典. 下面一个用 Python 解释器执行的示例:

```
>>> x = 5
>>> print locals()
{'x': 5, '__builtins__': <module '__builtin__' (built-in)>,
'__name__': '__main__', '__doc__': None, '__package__': None}
```

这说明在程序运行时, 变量名是存储在内存中的 (以及一些作为默认运行环境一部分的其他值).

编译语言中, 变量名存在于编译时, 而不是运行时. 编译器会为每个变量选择一个位置, 同时将这些位置记录为编译后的程序的一部分.[1] 变量的位置被称为**地址**. 运行时, 每个变量的值被存储在它的地址上, 变量的名称则不会被存储 (除非编译器出于调试的目的而添加它们).

1.3 编译流程

作为程序员, 你应该对编译期间发生了什么有个大致脉络. 如果你熟悉这个过程, 便可以帮你了解错误信息, 调试代码, 以及避免常见的陷阱.

编译步骤如下:

[1] 这只是一个简单描述, 稍后我们会详细说明.

1. 预处理: C 是一种包括**预处理指令** 的语言, 这些指令是在编译之前起效的. 比如, `#include` 指令会将另一个文件的源代码插入到指令的位置.
2. 解析: 解析期间, 编译器会读取源码并构建程序的内在结构, 也就是**抽象语法树**. 在此期间检查出来的错误, 通常是语法错误.
3. 静态检查: 编译器会检查变量和值的类型是否正确, 调用函数是否使用了正确数量和类型的参数, 等等. 此阶段侦测到的错误通常是**静态语义**错误.
4. 代码生成: 编译器会读取程序的内在结构, 并生成机器码或字节码.
5. 链接: 如果程序使用了库中定义的值和函数, 编译器需要找到合适的库, 并囊括所需的代码.
6. 优化: 在整个流程的各个阶段, 编译器可以转换程序, 生成运行更快, 或更加节约空间的代码. 大部分的优化是一些简单的修改, 可以消除明显的耗费, 但有些编译器可以执行更复杂的分析和代码转换.

通常运行 `gcc` 时, 它会执行所有这些步骤, 生成一个可执行文件. 比如, 下面是一个极简的 C 程序代码:

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
}
```

如果你将上述代码保存为一个名为 `hello.c` 的文件, 你便能像这样对其进行编译和执行:

```
$ gcc hello.c
$ ./a.out
```

默认情况下, `gcc` 将可执行代码存储在名为 `a.out` (起初表示 “assembler output”) 的文件中. 第二行会运行可执行文件. 前缀 `./`是告诉 shell 在当前目录中查找.

通常最好使用`-o` 标识为可执行文件提供一个合适的名称:

```
$ gcc hello.c -o hello
$ ./hello
```

1.4 目标码

使用`-c` 标识, 可以令 `gcc` 编译程序并生成机器码, 但并不会链接库中函数, 也就不会生成可执行文件:

```
$ gcc hello.c -c
```

结果是一个名为 hello.o 的文件, 其中 o 表示**目标码**, 也就是编译后的程序. 目标码不可执行, 但可以链接到可执行文件中.

UNIX 命令 nm 可以读取目标文件, 并生成涉及其所定义和应用的名称的信息. 例如:

```
$ nm hello.o
0000000000000000 T main
                 U puts
```

输出结果表明 hello.o 定义了名称 main, 以及名为 puts 的函数, 这个名字表示 “put string”. 在这个例子中, gcc 通过将庞大复杂的 printf 函数, 替换为相对简单的 puts, 以进行优化.

你可以使用-O 标识来控制 gcc 进行多大程度的优化. 默认情况下, 它优化得很少, 这会令调试相对简单. 而选项-O1 会开启常用且安全得优化. 更高得数字会开启更多得优化, 而这也就会需要更长得编译时间.

理论上, 优化除了加速程序之外, 不应该改变程序的行为. 但如果你的程序存在微妙的错误, 那你可能会发现, 优化会令错误暴露或消失. 通常在开发新代码时, 最好关闭优化. 一旦程序正常运行, 并能通过适当的测试, 你便可以开启优化, 并确认测试仍然通过.

1.5 汇编码

类似于-c 标识, -S 标识会令 gcc 编译程序并生成汇编码, 而这基本上是一种人可读的机器码形式.

```
$ gcc hello.c -S
```

结果是一个名为 hello.s 的文件, 看起来像这样:

```
        .file   "hello.c"
        .section    .rodata
.LC0:
        .string     "Hello World"
        .text
        .globl      main
        .type       main, @function
main:
.LFB0:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
```

```
        .cfi_offset 6, -16
        movq %rsp, %rbp
        .cfi_def_cfa_register 6
        movl $.LC0, %edi
        call puts
        movl $0, %eax
        popq %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size        main, .-main
        .ident       "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section     .note.GNU-stack,"",@progbits
```

gcc 通常会被设置, 为你运行的机器生成相应的代码, 所以对我来说, 它会生成 x86 汇编语言, 这可以在 Intel, AMD 和其他厂商的多种处理器上运行. 如果你在不同架构上运行, 你可能会得到不同的代码.

1.6 预处理

现在我们在编译过程中向后再走一步. 你可以用-E 标识, 仅仅只运行预处理器:

```
$ gcc hello.c -E
```

结果是预处理器的输出. 这个例子中, 它包含了 stdio.h 中包含的代码, 以及 stdio.h 中包含的所有文件, 依此类推. 在我的机器上, 结果有 800 多行代码. 由于几乎每个 C 程序都包括 stdio.h, 所以这 800 行代码会被编译多次. 如果像很多 C 程序一样, 你也包含了 stdlib.h, 那么结果要超过 1800 行代码了.

1.7 理解错误

现在我们知道了编译过程的各个步骤, 便更容易理解错误信息了. 比如, 如果#include 指令中存在错误, 你会从预处理器得到下面这条信息:

```
hello.c:1:20: fatal error: stdioo.h: No such file or directory
compilation terminated.
```

如果存在语法错误, 你会从编译器得到下面信息:

```
hello.c: In function 'main':
hello.c:6:1: error: expected ';' before '}' token
```

如果使用了任何标准库都没有定义的函数, 你会从链接器得到下面错误信息:

```
/tmp/cc7iAUbN.o: In function `main':
hello.c:(.text+0xf): undefined reference to `printff'
collect2: error: ld returned 1 exit status
```

`ld` 是 UNIX 链接器的名称, 如此命名是因为 “loading” 是编译过程中和链接关系密切的另一步骤.

一旦程序运行, C 便几乎不会进行运行时检查, 所以你可能只会看到极少的运行时错误. 如果你除以零, 或者执行了其他非法的浮点操作, 你会得到一个 “Floating point exception(浮点异常).” 同时, 如果你尝试读取或写入一个错误的内存位置, 你会遇到 “Segmentation fault(段错误).”

Chapter 2

进程

2.1 抽象和虚拟化

在我们谈论进程之前, 我想先定义几个词汇:

- 抽象: 抽象是某个复杂事物的简化表达. 例如, 当你开车时, 你知道当你左转方向盘时, 车会左转, 反之亦然. 当然, 方向盘连接着一系列机械和 (通常是) 液压系统, 这些会使车轮转动, 同时车轮会以复杂的方式和路面互动. 作为一个司机, 你通常无需考虑这些细节. 你只需要一个简单的逻辑模型来进行驾驶. 你的这个逻辑模型便是一种抽象.

  同样地, 当你使用网络浏览器时, 你知道当你点击链接时, 浏览器会显示该链接所指的页面. 使这一切成为可能的软件和网络通信是复杂的. 但作为一个用户, 你不必了解其细节.

  软件工程中一个重要部分就是设计想这样的抽象, 这些抽象可以使用户和其他程序员能够使用强大而复杂的系统, 而无需了解其实现的细节.

- 虚拟化: 一种重要的抽象便是虚拟化, 这是创建一个令人满意的幻觉的过程.

  例如, 很多公共图书馆会参与图书馆之间的协作, 允许其相互借书. 当我请求一本图书时, 有时这本书是在我当地的图书馆的书架上, 但其他时候, 它需要从另外的地方转移过来. 无论哪种情况, 当书籍可以领取时, 我都会接到通知. 我不必知道它来自哪里, 我也不需要知道我的图书馆里有哪些书. 整个系统创建了一个假象, 让我相信我的图书馆拥有世界上的任意一本图书.

  我当地的图书馆里实际可用的书可能很少, 但对我来说, 可用的虚拟书籍包括合作的图书馆中的每一本书.

再举个例子, 大多数计算机只连接到一个网络, 但该网络会连接到其他网络, 依此类推. 我们所称的互联网是一组网络和协议, 它们可以将数据包从一个网络转发到下一个网络. 从用户或程序员的角度来看, 该系统表现得好像互联网上的每台计算机都与其他计算机相连. 所以物理连接的数量虽然很少, 但虚拟连接的数量是非常大的.

“虚拟” 一词通常在虚拟机的上下文中使用, 虚拟机是一种软件, 会给你一种假象, 彷佛你运行在一个运行着特定操作系统的专有机器上. 而实际上, 虚拟机可能和很多其他虚拟机一起运行在同一台计算机上, 这些虚拟机可能运行着不同的操作系统.

在虚拟化的上下文中, 我们有时称真实发生的事情为 “物理事件”, 而虚拟发生的事情为 “逻辑事件” 或者 “抽象事件”.

2.2 隔离

工程学最重要的一个原则便是隔离: 当你设计一个包括多个部件的系统时, 最好将其相互之间进行隔离, 如此一个部件的改变就不会给其他组件带来不良影响.

操作系统的一个重要目标便是将每个运行的程序和其他程序进行隔离, 如此程序员便无需关注任何交互的可能. 提供这种隔离性的软件对象叫做**进程**.

进程是一个表示当前运行程序的软件对象. 我说的 “软件对象” 是指面向对象编程中的对象; 通常, 对象包含数据以及操作数据的方法. 一个进程便是一个包含下面数据的对象:

- 程序的文本, 通常是一系列机器语言指令.
- 程序相关的数据, 包括静态数据 (编译时分配) 和动态数据 (运行时分配).
- 任何未完成的输入/输出操作的状态. 比如, 如果进程正在等待从磁盘读取数据, 或者等待网络数据到达, 这些操作的状态便是进程的一部分.
- 程序的硬件状态, 包括寄存器中的数据, 状态信息以及程序计数器. 程序计数器用来标识当前正在执行哪个指令.

通常一个进程运行一个程序, 但也可能会加载并运行新的程序.

还有可能, 也是很常见的情况是, 多个进程中运行同一个程序. 在这种情况下, 这些进程会共享相同的程序代码, 但通常数据和硬件状态不同.

多数操作系统会提供一组基础功能, 以使进程之间相互隔离:

- 多任务: 多数操作系统都具有一种能力, 可以在任何时刻将运行中的进程中断, 并保存其硬件状态, 然后稍后恢复进程. 通常, 程序员不必考虑这些中断. 程序的行为就像在专用处理器上连续运行一样, 只是指令之间的时间是不可预测的.
- 虚拟内存: 多数操作系统都会创建这样的假象, 即每个进程都有自己的一段空间, 与其他进程都隔离开来. 同样地, 程序员一般无需关注虚拟内存如何工作; 他们可以按照每个程序都有一段专门的内存来进行操作.
- 设备抽象: 运行在同一个计算机上的进程, 共享磁盘驱动器, 网络接口, 图形卡, 以及其他硬件. 如果进程不经协商便直接和硬件交互, 便会引发混乱. 例如, 本来要传递给某个进程的网络数据, 可能被其他进程读取, 或多个进程向硬盘上同一个位置存储数据. 这便有赖于操作系统提供适当的抽象, 以维护秩序.

作为程序员, 你并不需要知道这些功能实现的详情. 但如果你有兴趣, 你会发现在各种隐喻之下, 有很多有趣的事情正在发生. 如果你知晓发生的细节, 那你将成为更好的程序员.

2.3 UNIX 进程

当写这本书时, 我最关注的进程是我的文本编辑器, emacs. 每隔一段时间, 我会切换到终端窗口. 这是一个运行 UNIX shell 的窗口, 它提供了命令行界面.

当我移动鼠标, 窗口管理器便会被唤醒, 查看鼠标是否在终端窗口上, 然后又会唤醒终端. 终端唤醒 shell. 如果我在 shell 中输入 `make`, 它会创建一个新的进程来执行 Make, 创建另一个进程运行 LaTeX, 然后再创建一个进程来显示结果.

如果我需要查找一些东西, 我可能会切换到另一个桌面, 这也会再次唤醒窗口管理器. 如果我点击网络浏览器的图标, 窗口管理器会创建一个进程, 运行网络浏览器. 有些浏览器, 比如 Chrome, 会为每个窗口和每个标签页都创建新的进程.

这些还只是我知道的进程. 同时, 还有很多其他进程在**后台** 运行. 其中很多进程正在执行与操作系统相关的操作.

UNIX 命令 `ps` 可以打印正在运行的进程的信息. 如果你在终端执行它, 你可能会看到像这样的东西:

```
  PID TTY          TIME CMD
 2687 pts/1    00:00:00 bash
 2801 pts/1    00:01:24 emacs
24762 pts/1    00:00:00 ps
```

第一列是唯一的数字格式进程 ID. 第二列是创建进程的终端; “TTY” 代表电传打印机, 这是最早的机械终端.

第三列是进程使用的处理器总时间, 以小时、分钟和秒为单位. 在这个例子中, `bash` 是解释输入命令的 shell 的名称, emacs 是我的文本编辑器, ps 是生成此输出的程序.

默认情况下, `ps` 仅列出和当前终端相关的进程. 如果你使用`-e` 标志, 你便能获取每个进程 (包括属于其他用户的进程, 就我看来, 这是一个安全漏洞).

在我的系统上, 目前有 233 个进程. 下面是其中一些:

```
  PID TTY          TIME CMD
    1 ?        00:00:17 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:02 ksoftirqd/0
    4 ?        00:00:00 kworker/0:0
    8 ?        00:00:00 migration/0
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:16 rcu_sched
   47 ?        00:00:00 cpuset
   48 ?        00:00:00 khelper
   49 ?        00:00:00 kdevtmpfs
   50 ?        00:00:00 netns
   51 ?        00:00:00 bdi-default
   52 ?        00:00:00 kintegrityd
   53 ?        00:00:00 kblockd
   54 ?        00:00:00 ata_sff
   55 ?        00:00:00 khubd
   56 ?        00:00:00 md
   57 ?        00:00:00 devfreq_wq
```

`init` 是操作系统启动时创建的第一个进程. 它会创建许多其他进程, 然后闲置, 直到它创建的进程都结束.

`kthreadd` 是操作系统用来创建**线程** 的进程. 后续我们将更详细地讨论线程, 当前你可以将线程看作是一种进程. 开头的 `k` 代表**内核 (kernel)**, 这是操作系统符合核心功能 (比如创建线程) 的部分. 结尾的 `d` 代表**守护程序 (daemon)**, 这是运行在后台, 并提供操作系统服务的进程的另一个名称. 在此, “daemon” 是指一种乐于助人的精神, 没有任何邪恶的意思.

根据名称, 你可以推断出 `ksoftirqd` 也是一个内核守护程序; 具体来说, 它处理软件中断请求, 即 “软中断 (soft IRQ)”

`kworker` 是一个内核创建的工作进程, 用来为内核处理一些任务.

通常有多个进程运行着内核服务. 在我的系统上, 现在有 8 个 `ksoftirqd` 进程和 35 个 `kworker` 进程.

我不会详细讨论其他进程, 但如果你感兴趣, 你可以搜索更多关于它们的信息. 你应该在你的系统上运行 `ps`, 并将结果和我的进行对比.

Chapter 3

虚拟内存

3.1 一点信息论知识

比特 (bit) 是一个二进制的数位, 也是一个信息单元. 如果你有一比特, 便可以表示两种可能性中的一种, 通常写作 0 和 1. 如果你有两比特, 那边有 4 种组合, 分别是 00, 01, 10, 和 11. 通常, 如果你有 b 个比特, 你便可以表示 2^b 个值. 一个**字节 (byte)** 是 8 个比特, 所以可以表示 256 个值.

反过来想, 假设你想储存一个字母. 字母表有 26 个字母, 那么你需要几个比特呢? 如果用 4 个比特, 你可以表示 16 个值, 数量不够. 如果用 5 个比特, 你可以表示多达 32 个值, 这足够表示所有字母了, 而且还有剩余.

通常, 如果你想表示 N 个值, 你应该选择满足 $2^b \geq N$ 的最小 b 值. 两侧以 2 为底取对数, 便可以得到 $b \geq log_2 N$.

假设我掷一枚硬币, 并告诉你结果, 那么我便是给了你一比特的信息. 如果我掷一个六面骰子, 并告诉你结果, 那我便是给了你 $log_2 6$ 个比特的信息. 通常, 如果发生的概率是 N 中的 1 个, 那么结果便包含 $log_2 N$ 个比特信息.

同样的, 如果结果出现的概率是 p, 那么信息量便是 $-log_2 p$. 这个数量也被称为结果的**自信息**, 它是用来衡量结果的惊奇程度, 所以也被称为**惊奇度**. 如果你的马仅有 1/16 的概率获胜, 但是赢了, 你可以 4 比特的信息 (以及赔付). 但如果获胜的机率是 75%, 那么胜利的消息只包含 0.42 比特.

直观来说, 出乎意料的消息反而包含了大量的信息. 相反, 如果你已经确定某件事情, 那么证实它仅会提供少量信息.

在本书的诸多主题中, 我们需要在比特数, b, 和其能编码的值的数量 $N = 2^b$ 之间熟练转换.

3.2 内存和存储

进程运行时，其大部分数据存储在**主内存**，这通常是一种随机访问存储 (RAM). 当前多数计算机上，主存储都是**易失的**，也就意味着一旦计算机关机，主存储的内容便会丢失. 一台典型的桌面计算机有 2–8 GiB 内存. GiB 表示 “gibibyte,” 也就是 2^{30} 字节.

进程需要读写文件，而文件往往存储在硬盘驱动器 (HDD) 或者固态硬盘中. 这些存储设备都是**非易失的**，因此可以用作长期存储. 目前一台典型桌面计算机会拥有 500GB 到 2TB 容量的硬盘. 其中 GB 表示 “gigabyte,” 也就是 10^9 字节. TB 表示 “terabyte,” 也就是 10^{12} 字节.

你可能已经注意到，我描述主内存大小时，用二进制单位 GiB，描述硬盘大小时，用十进制单位 GB 和 TB. 由于一些历史和科技的原因，内存通常用二进制单位衡量，而硬盘则用十进制单位. 在本书中，我将谨慎区分二进制和十进制单位，但你也应该注意到，“gigabyte” 和缩写 GB 通常是混用的.

在非正式的使用中，“内存” 一词有时候会指 HDD 和 SSD，以及 RAM. 但这些设备的属性是不同的，所以我们需要仔细区分它们. 我将使用**存储** 来指代 HDD 和 SSD.

3.3 地址空间

主内存中的每个字节都由一个整数**物理地址**指定. 有效的物理地址集被称为物理**地址空间**. 通常从 0 到 $N-1$，其中 N 是内存的大小. 在 1 GiB 物理内存的系统上，最高的有效地址是 $2^{30}-1$，即十进制下的 1,073,741,823 ，或者十六进制下的 0x3fff ffff(前缀的 0x 表示十六进制数).

然而，大部分的操作系统会提供**虚拟内存**，这意味着程序永远不必处理物理地址，也不必知道由多少物理地址可用.

相反，程序会使用从 0 到 $M-1$ 编号的**虚拟地址**，其中 M 是有效虚拟地址的数量. 虚拟地址空间的大小是由操作系统和其运行的硬件共同决定的.

你可能听过别人谈论 32 位系统和 64 位系统. 这些术语表示的是寄存器的大小，通常也是虚拟地址的大小. 在 32 位系统上，虚拟地址为 32 比特，也就是虚拟地址空间从 0 到 0xffff ffff. 所以地址空间的大小为 2^{32} 字节，或 4 GiB.

而在 64 位系统上，虚拟地址空间是 2^{64} 字节，或 $2^4 \cdot 1024^6$ 字节. 这是 16 艾字节，约为当前物理内存的十亿倍. 虚拟地址空间比物理内存大这么多，这看起来很奇怪，但是我们稍后就会看到它是如何工作的.

当程序从内存读取或者向内存写入值时，它会生成虚拟地址. 在操作系统的帮助下，访问内存之前，硬件会被转换成物理地址. 这种转换是基于每个进程

进行的, 所以即使两个进程生成同样的虚拟地址, 它们也会映射到物理内存的不同位置.

最后, 虚拟内存是操作系统隔离进程的一种重要手段. 通常, 一个进程是无法访问另一个进程拥有的数据的, 因为如果某块物理内存已经被分配给了其他进程, 那么该进程便无法生成映射到这块物理内存的虚拟地址.

3.4 内存段

运行中得进程的数据会被组织成五个段:

- **代码段** 包含程序文本; 也就是构成程序的机器语言指令.
- **静态段** 包含不可变的值, 像字符串常量. 比如, 如果你的程序包含字符串`"Hello, World"`, 那这些字符会被保存在静态段.
- **全局段** 包含全局变量和被声明为 **static(静态类型)** 的局部变量.
- **堆段** 包含运行时被分配的内存块, 通常是通过调用 C 库函数 **malloc** 所分配.
- **栈段** 包含调用栈, 也就是一系列的堆栈帧. 每次函数被调用, 都会为其分配一个包含函数的参数和局部变量的堆栈帧. 当函数运行结束, 其堆栈帧会从栈中移除.

段的排序部分由编译器决定, 部分由操作系统决定. 具体细节取决于操作系统, 但常见的排列方式包括::

- 文本段位于内存接近 “底部” 的位置, 也就是靠近地址 0 的地方.
- 静态段通常在文本段上方, 也就是高地址位置.
- 全局段通常位于静态段上方.
- 堆通常在全局段上方. 随着堆的扩张, 它会向更大地址增长.
- 栈靠近内存的顶部; 也就是虚拟地址空间中最高地址附近. 栈扩展时, 会朝着更小地址方向增长.

若要确定你系统上这些段的布局, 尝试运行这个程序, 在本书仓库中的`aspace.c` 文件中 (详见第 0.1节).

```
#include <stdio.h>
#include <stdlib.h>

int global;

int main ()
{
    int local = 5;
    void *p = malloc(128);
    char *s = "Hello, World";

    printf ("Address of main is %p\n", main);
    printf ("Address of global is %p\n", &global);
    printf ("Address of local is %p\n", &local);
    printf ("p points to %p\n", p);
    printf ("s points to %p\n", s);
}
```

main 是这个函数的名称; 当作为变量使用时, 它指的是 main 中的第一条机器语言指令的地址, 同时我们希望它在文本段中.

global 是全局变量, 所以我们希望它在全局段中. local 是局部变量, 我们期望它存在于栈中.

s 指向 “字符串文本”, 它是程序的一部分 (与从文件读取, 或用户输入的字符串等不同). 我们希望字符串的位置在静态段中 (与指针 s 不同, 它是局部变量).

p 包含 malloc 返回的地址, 这个函数会在堆中分配空间. “malloc” 表示 “memory allocate(内存分配)”.

格式序列%p 会告诉 printf 将每个地址格式化为 “指针”, 因此它会以十六进制显示结果.

当我运行这个程序, 输出如下 (我增加了些空格, 以便于阅读):

```
Address of main is   0x      40057d
Address of global is 0x      60104c
Address of local is  0x7ffe6085443c
p points to          0x     16c3010
s points to          0x      4006a4
```

和预计的一样, main 的地址是最低的, 然后是字符串文本的位置. 接着是 global 的位置, 再然后是 p 指向的地址. local 的地址要大得多.

最大的地址有 12 位十六进制数. 每个十六进制数对应 4 个比特, 所以一共是 48 比特地址. 这便表示可用虚拟地址空间的大小为 2^{48} 字节.

做个练习, 在你电脑上运行这个程序, 并和我的结果进行比较. 再添加一个 `malloc`, 检查你系统上的堆是否向上增长了 (向更大地址方向). 再添加一个输出局部变量地址的函数, 并检查栈是否向下变化了.

3.5 静态局部变量

栈上的局部变量有时被称为**自动变量**, 因为它们会在函数调用时, 自动分配, 在函数返回时, 自动释放.

在 C 语言中, 存在另一种局部变量, 会在全局段中进行分配, 称为**静态变量**, 它会在程序启动时被初始化, 同时在函数调用之间保持其值不变.

例如, 下面的函数会跟踪其被调用的次数.

```
int times_called()
{
    static int counter = 0;
    counter++;
    return counter;
}
```

关键字 `static` 表示 `counter` 是一个静态局部变量. 初始化仅会在程序启动时, 发生一次.

如果你将此函数添加到 `aspace.c` 中, 你便可以确认 `counter` 是在全局段中与全局变量一起分配的, 而不是在栈中分配的.

3.6 地址转换

虚拟地址 (VA) 如何能转换成物理地址 (PA)? 基本机制很简单, 但简单的实现会太慢, 并消耗太多空间. 所以实际的实现有点复杂.

大部分处理器会提供位于 CPU 和主存之间的内存管理单元 (MMU). MMU 会执行 VA 和 PA 之间的快速转换.

1. 当程序读取或写入一个变量时, CPU 会生成一个 VA.
2. MMU 将 VA 分成两部分, 分别是页号和偏移量. “页” 是内存的一个块, 页的大小取决于操作系统和硬件, 常见大小一般是 1–4 KiB.

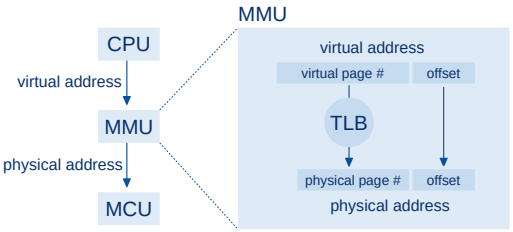

图 3.1: 地址转换过程示意图.

3. MMU 会从页表缓存 (TLB) 查找页号, 并获得相应的物理地址页号. 然后, 它将物理页号和偏移量结合起来, 构成 PA.

4. PA 会传递给主存, 主存基于给定位置进行读写.

TLB 包含来自页表 (存储在内核内存中) 数据的缓存副本. 页表包含虚拟页号到物理页号的映射. 由于每个进程都有自己的页表, 因此 TLB 必需确保它只使用正在运行的进程的页表中的条目.

图 3.1 是这个过程的示意图. 若要明了其工作原理, 假设 VA 是 32 比特, 物理内存是 1 GiB, 并划分为了 1 KiB 的页.

- 因为 1 GiB 是 2^{30} 字节, 1 KiB 是 2^{10} 字节, 所以有 2^{20} 个物理页, 有时也称为 “帧”.
- 虚拟地址空间大小是 2^{32} B, 单个页的大小是 2^{10} B, 所以有 2^{22} 个虚拟页.
- 偏移量的大小由页面大小决定. 这个例子中, 页面大小是 2^{10} B, 所以需要 10 比特来定位页面上的字节.
- 如果 VA 是 32 比特, 同时偏移量是 10 比特, 剩下的 22 比特便构成了虚拟页号.
- 由于有 2^{20} 个物理页, 每个物理页号是 20 比特. 加上 10 比特的偏移量, 得到的 PA 便是 30 比特.

至此, 一切似乎看起来可行. 但让我们想一下, 页表可能有多大. 页表的最简单实现是一个数组, 其中每个条目对应一个虚拟页. 每个条目包含物理页号, 本例中是 20 比特, 以及关于每帧的附加信息. 所以我们预计每个条目需要 3–4 字节的空间. 由于存在 2^{22} 个虚拟页, 页表将需要 2^{24} 字节, 即 16 MiB.

由于每个进程都需要一个页表, 那么运行 256 个进程的系统, 便需要 2^{32} 字节, 即 4GiB, 还只是页表空间! 而这仅仅是针对 32 位虚拟地址的情况. 对于 48 或 64 位的虚拟地址来说, 这个数字就太吓人了.

幸运的是, 我们实际上并不需要这么多空间, 因为多数进程甚至用不到虚拟地址空间的一小部分. 而且, 如果一个进程不使用虚拟页, 我们便不需要在页面中为其分配条目.

另一种表达同样观点的方式是页表是 “稀疏的”, 而这最简单的实现方式, 即一个页表条目数组, 但这是一个糟糕的注意. 幸运的是, 对于稀疏数组, 有其他高效的实现方案.

一种选择是多级页表, 这是许多操作系统 (包括 Linux) 使用的方案. 另一种选择是关联表, 其中每个条目包括虚拟页号和物理页号. 在软件中检索关联表很慢, 但在硬件中, 我们可以并行检索整个表, 所以其通常用于表示 TLB 中的页表条目.

你也可以从`http://en.wikipedia.org/wiki/Page_table`了解更多关于这些实现的信息; 你可能对这些细节感兴趣. 但核心思想为页表是稀疏的, 所以我们需要为稀疏数组选择一个好的实现方案.

前面我提到操作系统可以中断运行中的进程, 保存其状态, 然后运行另一个进程. 这个机制叫做**上下文切换**. 因为每个进程都有自己的页表, 操作系统需要和 MMU 协作, 从而确保每个进程都能获得正确的页表. 在老的机器上, MMU 中的页表信息需要在每次上下文切换时进行替换, 而这是很昂贵的. 在新的系统中, MMU 中的每个页表条目都包含进程 ID, 因此多个进程的页表可以同时存在于 MMU 中.

Chapter 4

文件和文件系统

当进程结束 (或崩溃), 存储在主内存中的任何数据都会丢失. 但存储在硬盘驱动器 (HDD) 或者固态硬盘驱动器上的数据是 “持久的”; 也就是说进程结束, 甚至计算机关闭, 也会存在.

硬盘驱动器是复杂的, 数据以块的形式存储, 这些块存在于扇区上, 扇区构成磁道, 磁道则排列在盘片的同心圆上.

固态硬盘在某种意义上相对简单, 因为块会按照顺序编号, 但它们也因此引入了一个不一样的复杂性: 每个块只能写入有限的次数, 然后就会变得不可靠.

作为个程序员, 你不会想要处理这些复杂性. 你想要的是对持久存储硬件的合理抽象. 而最常见的抽象就是 “文件系统”.

抽象地说:

- “文件系统” 是将每个文件名称映射到其内容的一种方式. 如果你将名称视为键, 内容当作值, 文件系统便是一种键-值数据库. (参见 `https://en.wikipedia.org/wiki/Key-value_database`).
- 一个 “文件” 便是一个字节序列.

文件名通常为字符串, 而且一般是 “分层次的”; 也就是说, 该字符串指定了一个从顶级目录 (或文件夹) 到特定文件的一系列子目录的路径.

抽象层以及底层机制之间最主要的区别是, 文件是基于字节的, 而持久层存储是基于块的. 操作系统将 C 库中基于字节的文件操作转换为对存储设备上基于块的操作. 典型的块大小一般为 1–8 KiB.

例如, 下面代码会打开文件并读取第一个字节:

```
FILE *fp = fopen("/home/downey/file.txt", "r");
char c = fgetc(fp);
fclose(fp);
```

上述代码运行时:

1. `fopen` 会使用文件名中的/ 确定顶级目录, 以及子目录 `home`, 以及下一级子目录 `downey`.

2. 它会找到名为 `file.txt` 的文件, 然后 “打开 (opens)” 文件以进行读取, 也就表示, 这会创建一个表示正在被读取的文件的数据结构. 这个数据结构除了其他事情, 还会跟踪文件已读取了多少, 也就是 “文件位置”.

   在 DOS 中, 这个数据结构叫做文件控制块 (FCB), 但我希望尽量避免使用这个术语, 因为在 UNIX 中, 它有其他含义. 在 UNIX 中, 似乎没有一个相应的好名称. 它是打开文件表中的一个条目, 所以我称其为打开文件表条目.

3. 当我调用 `fgetc` 时, 操作系统会检查文件的下一个字符是否已经存在于内存中. 如果已存在, 则会读取下一个字符, 更新文件位置, 并返回结果.

4. 如果下一个字符不在内存中, 操作系统会发出一个 I/O 请求, 获取下一个块. 磁盘驱动器速度较慢, 所以一个等待从磁盘获取块的进程通常会被中断, 以便另一个进程运行, 直到数据返回, 再回到等待的进程.

5. 当 I/O 操作结束, 新的数据块会被存储到内存, 同时进程继续运行. 读取第一个字符, 并将其存储为局部变量.

6. 当进程关闭文件, 操作系统会完成或者取消所有挂起的操作, 移除存储在内存中的数据, 并释放打开文件表条目 (OpenFileTableEntry).

写文件的过程类似, 只是会多一些额外步骤. 下面是一个打开文件进行写入并修改第一个字符的示例.

```
FILE *fp = fopen("/home/downey/file.txt", "w");
fputc('b', fp);
fclose(fp);
```

当代码运行时:

1. 再次, `fopen` 使用文件名定位文件. 如果文件不存在, 则会创建一个新文件, 并在父目录`/home/downey` 中添加一个条目.

2. 操作系统会创建一个打开文件表条目, 以标明此文件以写入方式打开, 同时设置文件位置为 0.

3. `fputc` 会尝试写入 (或重写) 文件的第一个字节. 如果文件已经存在, 操作系统需要将第一个块加载入内存. 否则, 它会在内存中分配一个新块, 同时在磁盘上申请一个新块.

4. 内存中的块被修改后不会立刻被复制回磁盘. 通常, 写入文件的数据是“缓冲的”, 意味着它会保存在内存, 直到至少有一个块需要写入, 才会写入磁盘.

5. 当文件关闭, 任何缓冲区的数据都会被写入磁盘, 同时释放打开文件表条目.

综上所述, C 库提供了一个从文件名映射到字节流的文件系统的抽象. 这个抽象建立在实际以块进行组织的存储设备之上.

4.1 磁盘性能

我之前提过磁盘驱动器很慢. 对于当前的 HDD 来说, 从磁盘读取一个块到内存的平均时间大约 5–25 毫秒 (参见 `https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics`). 而 SSD 则快很多, 读一个 4 KiB 的块, 只需要 25 μs , 写一个块需要 250 μs (参见 `http://en.wikipedia.org/wiki/Ssd#Controller`).

为了更好地理解这些数字, 我们将其和 CPU 的时钟周期进行比较. 时钟频率为 2GHz 的处理器每 0.5 ns 完成一个时钟周期. 从内存获取一个字节到 CPU 的时间通常约为 100 ns. 如果处理器每个时钟周期完成一条指令, 那么在等待从内存获取一个字节的时间可以完成 200 条指令.

在 1 微秒内, 处理器可以完成 2000 条指令, 所以等待从 SSD 获取一字节的 25 μs 内, 可以完成 50,000 条指令.

在 1 毫秒内, 处理器可以完成 2,000,000 条指令, 所以等待从 HDD 获取一字节的 20 ms 内, 处理器可以完成 4 千万条指令. 如果在等待期间, CPU 没有其他任务执行, 则它会处于空闲状态. 这也是为什么操作系统通常在等待磁盘数据时切换到另一个进程的原因.

主存和持久化存储之间的性能差距是计算机系统设计的主要挑战之一. 操作系统和硬件提供了几种特性, 以 “填补” 这个差距:

- 块传输: 从磁盘加载单个字节的时间是 5–25 ms. 相比之下, 加载 8 KiB 块的额外时间可以忽略不急. 所以系统通常每次访问磁盘时, 尽量读取大块数据.

- 预取: 有时候操作系统可以预测一个进程将会读取的块, 从而在请求之前便进行加载. 比如, 如果你打开一个文件并读取第一个块, 那么你很可能会继续读取第二个块. 操作系统可能会在被请求之前, 先加载其他块.

- 缓冲区: 正如我之前提到的, 当你写入一个文件, 操作系统会将数据存储在内存中, 稍后再将其写入磁盘. 如果当其在内存中时, 你多次修改了块, 系统只需要往磁盘写入一次.

- 缓存: 如果进程最近使用了某个块, 很可能再次使用它. 如果操作系统在内存中保留了该块的副本, 那便可以以内存速度处理将来的请求.

这些功能的有些实现是在硬件中. 比如, 有些磁盘驱动器会提供一个缓存, 来存储最近使用过的块, 同时很多磁盘驱动器即使只被请求了一个块, 也会一次夺取多个块.

这些机制通常会提高程序的性能, 但不会改变其行为. 通常程序员不需要考虑它们, 除非出现以下两种情况: (1) 如果程序性能很糟糕, 你可能需要了解这些机制以便诊断问题, 以及 (2) 当数据被缓冲存储了, 程序调试会变得困难. 例如, 如果一个程序打印一个值, 然后崩溃. 该值可能不会出现, 因为它可能在缓冲区. 同样, 如果一个程序往磁盘写数据, 然后计算机断电了, 如果数据还在缓存中, 尚未写入磁盘, 那么数据很可能会丢失.

4.2 磁盘元数据

构成文件的块在磁盘上可能是连续排列, 如此的话, 文件系统的性能一般会更好, 但多数操作系统不强求连续分配. 它们可以在磁盘的任意位置自由放置数据块, 并使用各种数据结构对其进行跟踪.

在很多 UNIX 文件系统中, 该数据结构被称为 “inode”, 即 “index node(索引节点)”. 通常, 关于文件的信息, 包括其数据块的位置, 被称为 “元数据”. (文件的内容是数据, 所以关于文件的信息便是关于数据的数据, 因此称为 “元数据”.)

由于 inode 一般与其他数据一起存储在磁盘上, 它们被设计成了完美适应磁盘块. UNIX 的 inode 包含文件的信息, 包括文件所有者的用户 ID; 表示谁可以读写或执行该文件的权限标注; 以及指示文件上次修改和访问的时间戳. 此外, 它还包含组成文件的前 12 个数据块编号.

如果块大小是 8 KiB, 那么前 12 个块总共占用 96 KiB 空间. 多数系统中, 这对于绝大多数文件来说已经足够大, 但仍不够容纳所有文件. 这也是为何 inode 会包含指向 “间接块” 的指针的原因, 该间接块仅包含指向其他块的指针.

间接块中的指针数量取决于块大小和块编号, 但通常是 1024. 对于具有 1024 个块编号, 并且块大小为 8 KiB 时, 那么一个间接块可以寻址 8 MiB 的空间. 这对于大多数大文件来说, 已经足够大, 但仍然不够容纳所有文件.

这也是为何 inode 也包含指向 “双重间接块” 的指针的缘由. 双重间接块包含指向间接块的指针. 通过 1024 个间接块, 我们便能寻址 8GiB 空间.

同时如果还是不够大, (最后) 还有三重间接块, 也就是包含指向双重间接块指针的块, 从而使文件大小达到 8 TiB. 当 UNIX 的 inode 设计时, 这似乎在很长时间内都足够大了. 但那是很久之前了.

有些文件系统, 比如 FAT, 会使用一个被称为 “簇” 的文件分配表作为间接块的替代方案, 这个表包含每个块的一个条目. 根目录包含每个文件中第一个簇的指针. 每个簇的 FAT 条目都会指向文件中的下一个簇, 和链表类似. 更多详细信息, 请参考 `http://en.wikipedia.org/wiki/File_Allocation_Table`.

4.3 块分配

文件系统需要跟踪哪些块属于某个文件; 也需要跟踪哪些块可以被使用. 当创建新文件时, 文件系统会找到一个可用的块并进行分配. 当删除了文件, 文件系统会将块标记为可用, 以重新分配.

块分配系统的目标是:

- 速度: 分配和释放块要快速.
- 最小化空间开销: 分配器使用的数据结构要尽量小, 为数据保留尽量多的空间.
- 最小化碎片: 如果有些块不再使用, 或者有些仅部分使用, 那么未使用的空间便是 “碎片”.
- 最大化连续性: 如果可以, 一起使用的数据最好在物理上是连续的, 以提高性能.

很难设计出一个满足上述所有目标的文件系统, 特别是文件系统的吸能取决于诸如文件大小, 访问模式等的 “负载特征”. 对于一个工作负载优化过的文件系统, 并不一定适用于另一个.

因此, 大部分操作系统支持多种文件系统, 同时文件系统设计也是一个研究和开发的活跃领域. 过去十年, Linux 系统已经从传统的 UNIX 文件系统 ext2, 迁移到了 ext3, ext3 是一个 “日志” 文件系统, 旨在提高速度和连续性, 最近又迁移到了 ext4, 这是一个可以处理更大的文件的文件系统. 在可预计的将来, 可能又会迁移到 B-树文件系统, Btrfs.

4.4 一切皆文件?

文件抽象实际是一个 “字节流” 抽象, 这种抽象对于很多事情都有用, 而不仅仅是文件系统.

一个例子便是 UNIX 管道, 这是一个简单的进程间通信方式. 进程可以设置为将一个进程的输出作为另一个进程的输入. 对于第一个进程来说, 这个管道就像一个以写入方式打开的文件, 所以它可以使用 C 库函数, 如 `fputs` 和 `fprintf`. 对于第二个进程来说, 管道就像一个以读取方式打开的文件, 所以它可以使用 `fgets` 和 `fscanf`.

网络通信也使用字节流抽象. UNIX 套接字便是一个表示不同计算机的进程间通信通道的数据结构. 同样, 进程可以使用 “文件” 处理函数, 从套接字读取以及写入数据.

复用文件抽象令程序员的工作变得简单, 因为他们只需要学习一个 API(应用程序编程接口). 同时也使程序更加灵活, 因为原本设计用于文件的程序, 也可以处理来自管道和其他来源的数据.

Chapter 5

位和字节

5.1 整数表示

你可能知道计算机是用二进制表示数字. 对于正数来说, 二进制表示很简单; 例如, 十进制的 5_{10} 可以用二进制表示为 $b101$.

对于负数来说, 最直观的表示是采用一个符号位来标识数字的正负. 但也有其他表示方法, 称为 “补码”, 因为其在硬件处理中更容易, 所以更加常用.

若要找到一个负数 $-x$ 的补码, 需要先找到 x 的二进制表示, 然后翻转所有未, 再加 1. 比如, 若要表示 -5_{10}, 可以从表示 5_{10} 开始, 如果写成 8 位版本, 即 $b00000101$. 将所有位翻转并加 1, 得到 $b11111011$.

在二进制补码中, 最左侧的位充当符号位; 正数的符号位是 0, 负数的符号位为 1.

若要将 8 位数字转为 16 位, 我们需要为正数添加更多的 0, 为负数添加更多的 1. 实际上, 我们需要将符号位复制到新的位上. 这个过程叫做 “符号扩展”.

在 C 语言中, 所有的整数类型都是有符号位的 (能够表示正数和负数), 除非声明其为无符号位. 这个声明的区别以及重要性在于, 无符号位的整数不需要符号扩展.

5.2 位运算符

人们学习 C 时, 有时会对位运算符`&` 和 `|` 满腹疑惑. 这些运算符将整数视为位向量, 并对相应的位进行逻辑运算.

比如, `&` 进行 AND(且) 运算, 这会在两个操作数都是 1 时, 结果为 1, 否则为 0. 下面是对两个 4-位数字应用`&`运算的例子:

```
  1100
& 1010
  ----
  1000
```

在 C 中, 这表示表达式 `12 & 10` 的结果为 8.

同样的, `|` 会进行 OR(或) 运算, 在两个操作数的其中一个为 1 时, 结果为 1, 否则为 0.

```
  1100
| 1010
  ----
  1110
```

所以表达式 `12 | 10` 的结果为 14.

最后, 运算符 `^` 表示 XOR(异或) 操作, 在两个操作数有且只有一个为 1 时, 结果为 1.

```
  1100
^ 1010
  ----
  0110
```

所以表达式 `12 ^ 10` 的结果为 6.

最常见的用法, `&` 用来从位向量中清除一组位, `|` 用来设置位, 而 `^` 用来翻转或 “切换” 位. 下面是详细信息:

清除位: 对于任意 x, $x\&0$ 结果是 0, 而 $x\&1$ 的结果是 x. 所以如果你用 3 对一个向量执行 AND 操作, 则只会选择最右边的两位, 并将其他位置设置为 0.

```
  xxxx
& 0011
  ----
  00xx
```

在这里, 3 叫做 “掩码”, 因为它选择一些位, 并掩盖其余的位.

设置位: 同样, 对于任意 x, $x|0$ 结果是 x, 而 $x|1$ 结果是 1. 所以如果你用 3 对一个向量执行 OR 运算, 则会设置最右边的位, 并保持其余位不变:

```
  xxxx
| 0011
  ----
  xx11
```

翻转位: 最后, 如果你用 3 对一个向量执行 XOR 操作, 它会翻转最右边的位, 同时保持其余位不变. 作为练习, 看看你是否可以使用^计算出 12 的二进制补码. 提示: -1 的二进制补码表现形式是什么?

C 语言也提供了移位操作 `<<` 和 `>>`, 用于左移位和右移位. 每次左移会将数字翻倍, 所以 `5 << 1` 结果是 10, `5 << 2` 等于 20. 每次右移将会令数字除 2(向下取整), 所以 `5 >> 1` 的结果为 2, 同时 `2 >> 1` 结果为 1.

5.3 浮点数表示

浮点数使用二进制科学计数法表示. 在十进制表示法中, 大数字会被写成系数与 10 的指数幂相乘的形式. 例如, 光速用 m/s 表示, 大约为 $2.998 \cdot 10^8$.

多数计算机使用 IEEE 标准进行浮点数运算. C 语言中的 `float` 类型通常对应 32 位的 IEEE 标准; `double` 通常对应 64 位标准.

在 32 位标准中, 最左侧的位是符号位 s. 其后 8 位是指数 q, 最后的 23 位是系数 c. 那么一个浮点数的值便可以表示为:

$$(-1)^s c \cdot 2^q$$

基本正确, 但是小有差异. 浮点数通常是规范化的, 以确保小数点前有一个数字. 例如, 在十进制中, 我们一般写成 $2.998 \cdot 10^8$, 而不是 $2998 \cdot 10^5$, 或者其他等价表达式. 在二进制中, 规范的数字总是在二进制点前有一个数字. 由于该位置的数字始终是 1, 我们便可以在表示中省略它, 从而节省空间.

比如, 13_{10} 的整数表示是 $b1101$. 浮点数表示便是 $1.101 \cdot 2^3$, 指数是 3, 系数部分存储为 101(后跟 20 个零).

这基本正确, 但存在一个小细节. 指数是用 “偏移” 存储的. 在 32-位标准中, 偏移是 127, 所以指数 3 会被存储为 130.

在 C 中打包和解包浮点数, 我们可以用联合位运算操作. 这是一个例子:

```
union {
    float f;
    unsigned int u;
} p;

p.f = -13.0;
unsigned int sign = (p.u >> 31) & 1;
unsigned int exp = (p.u >> 23) & 0xff;

unsigned int coef_mask = (1 << 23) - 1;
unsigned int coef = p.u & coef_mask;

printf("%d\n", sign);
```

```
    printf("%d\n", exp);
    printf("0x%x\n", coef);
```

此代码存在于本书配套仓库中的 `float.c` 文件中 (见章节 0.1).

联合运算符允许我们使用 `p.f` 存储浮点数, 然后用 `p.u` 将其读取为无符号整数.

要获取符号位, 我们需要向右移 31 位, 然后用 1 位掩码来选择最右边的位.

若要获取指数, 我们右移 23 位, 然后选择最右侧的 8 位 (十六进制 `0xff` 有 8 个 1).

要获取系数, 我们需要提取最右侧 23 位并忽略其余位. 我们通过创建一个右侧 23 位均为 1, 左侧为 0 的掩码来实现. 最容易的方法是将 1 左移 23 位, 然后减去 1.

程序输出为:

```
1
130
0x500000
```

如预期一样, 负数的符号位是 1. 指数是 130, 包括偏差. 系数用十六进制表示, 是 101, 后跟 20 个零.

做个练习, 尝试聚合或者拆散用 64 位表示的 `double`, 请参阅`http://en.wikipedia.org/wiki/IEEE_floating_point`.

5.4 联合体和内存错误

C 联合体有两种常见用途. 一种是前面章节看到的, 用于访问数据的二进制表示. 另一种是用于存储异构数据. 比如, 你可以使用一个联合体来表示一个可能是整数, 浮点数, 复数或有理数的数字.

然而, 联合体是存在错误风险的. 这往往取决于你, 作为程序员, 往往需要跟踪联合体中数据类型; 如果你写了一个浮点数, 然后将其解释为了整数, 得到的结果便毫无意义了.

实际上, 如果你错误读取了内存中的某个位置, 也会发生同样的事情. 导致这种问题发生的一种方式是你读取了超出数组末尾的位置.

若要看到会发生什么, 我先创建一个在堆栈上分配数组, 并用 0 到 99 的数字进行填充的函数.

```
void f1() {
    int i;
    int array[100];

    for (i=0; i<100; i++) {
        array[i] = i;
    }
}
```

下一步我会定义一个函数, 以创建一个更小的数组, 并有意访问开始之前和结束之后的元素.

```
void f2() {
    int x = 17;
    int array[10];
    int y = 123;

    printf("%d\n", array[-2]);
    printf("%d\n", array[-1]);
    printf("%d\n", array[10]);
    printf("%d\n", array[11]);
}
```

如果我先调用 `f1`, 然后调用 `f2`, 我会得到下面结果:

```
17
123
98
99
```

这里细节取决于编译器, 它会在堆栈安排变量的位置. 从这些结果, 我们可以推断出编译器将 `x` 和 `y` 放在了一起, 位于数组的 “下方” (一个低地址的位置). 当我们读取超出数组的元素时, 似乎我们得到了前一个函数调用而留在堆栈上的值.

在这个例子中, 所有的变量都是整数, 所以相对容易弄清楚发生了什么. 但是通常当你读取超出数组边界的位置时, 你读到的值可能是任何类型. 例如, 如果我修改 `f1` 函数, 创建一个浮点数数组, 结果如下:

```
17
123
1120141312
1120272384
```

后两个值是你将浮点数作为整数解释得到的值. 如果你调试时遇到这种输出, 你很难确定发生了什么.

5.5 字符串表示

有时和字符串相关的问题也会出现. 首先, 记住 C 语言中的字符串是以 null 结尾的. 当你为字符串分配空间时, 不要忘记在末尾多留一个字节.

另外, C 语言中字符串内的字母*以及数字*是用 ASCII 编码的. 数字 “0” 到 “9” 的 ASCII 编码是 48 到 57, *而不是* 0 到 9. ASCII 码中的 0 是 NULL 字符, 用来标识字符串的结束. ASCII 码中的 1 到 9 是用于某些通讯协议的特殊字符. ASCII 码中的 7 是一个响铃符; 在某些终端上, 打印它会发出声音.

字母 “’A” 的 ASCII 码是 65; “a” 的编码是 97. 下面是这些编码的二进制表示:

```
65 = b0100 0001
97 = b0110 0001
```

仔细观察者会注意到他们只有一个位存在差异. 其他字母也是这种模式; 第六位 (从右边数) 充当的是 “大小写位”, 大写字母是 0, 小写字母是 1.

做个练习, 写个函数, 使其接收一个字符串, 通过反转第六位, 将其从小写转成大写. 作为挑战, 你可以通过一次性读取 32 位或 64 位的字符串, 而不是逐个字符读取, 来创建更快的版本. 如果字符串长度是 4 或 8 字节的倍数, 优化过程会更容易实现.

如果你读取超出了字符串的末尾, 你很可能会看到奇怪的字符. 相反, 如果你写入一个字符串, 然后意外将其当作整数或者浮点数读取, 结果会很难解释.

例如, 如果你运行:

```
char array[] = "allen";
float *p = array;
printf("%f\n", *p);
```

你会发现将我名字的前 8 个字符的 ASCII 表示, 解释为双精度的浮点数字, 结果是 6977971387880058545766 4.

Chapter 6

内存管理

C 语言为动态内存分配提供了 4 个函数:

- `malloc` 接收一个以字节为单位的整数, 返回一个 (至少) 指定大小的新分配内存块. 如果无法满足此条件, 则返回特殊指针 NULL.
- `calloc` 和 `malloc` 一样, 不同之处在于它会清除新分配的块, 也就是说, 它会将块中的所有字节设置为 0.
- `free` 会接受先前分配块的指针, 并释放它; 也就是说, 它可以将空间用于未来分配.
- `realloc` 会接收一个之前分配块的指针和一个新的大小. 它会根据新的大小分配内存块, 并将数据从旧的块拷贝到新的块, 释放旧的块, 返回新的块的指针.

这个 API 是出了名的容易出错且无法容忍. 内存管理一直是设计大型软件系统所面临的最大挑战之一, 而这也是为什么大多数现代语言提供类似垃圾回收的高级内存管理的特性的原因.

6.1 内存错误

C 语言的内存管理 API 和动画节目 *The Simpsons* 中的一个小人物 Jasper Beardly 有点像; 在少数的几集中, 他作为一个严厉的代课老师而出现, 对所有的违规行为都施加体罚—“打屁股”'.

下面是程序会做的一些需要被打屁股的事情:

- 访问 (读写) 任意未分配的块.

- 访问已释放的分配内存.
- 尝试释放未分配的内存.
- 重复释放同一块内存.
- 对未分配的内存, 或者分配后已释放的内存调用 `realloc` 函数.

遵循这些规则看起来不难, 但在大型程序中, 内存块会被某段程序分配, 又被不同部分代码使用, 再被其他部分释放. 所以, 程序中一部分的修改, 往往涉及其他很多地方.

同时, 程序的不同部分可能有很多对同一块内存的别称, 或者引用. 直到所有对某个内存块的引用都不再使用, 内存才能释放. 正确处理这点, 往往需要在程序所有地方都仔细分析, 这很困难, 而这也违反了良好软件工程的基本原则.

理想情况是, 每个分配内存的函数, 都应该在文档化的接口中包含如何释放该内存的信息. 成熟的库通常在这点做的很好, 但现实世界中, 软件工程实践往往和理想状态差之甚远.

更糟糕的是, 内存错误因为各种状况难以预测, 往往很难定位. 比如:

- 如果你从一个未分配的内存块中读取值, 系统可能会检测到错误, 触发一个叫 "分段错误" 的运行时错误, 并终止程序. 或者, 程序也可能并不会报错, 直接读取未分配的内存; 这种情况下, 获取的值是访问位置存储的值, 而这是无法预测的, 程序每次执行结果可能都不同.
- 如果你向一个未分配内存块写入值, 也没有遇到分段错误, 事情可能更糟糕. 你向一个无效的位置写入一个值, 可能经过很长时间才会读取, 并引发问题. 而这时很难发现问题的源头.

事情可能更糟! C 风格的内存管理中一个最常见的问题是用于实现 `malloc` 和 `free` (我们很快会看到) 的数据结构通常和分配的内存块存储在一起. 因此, 如果你意外地在动态分配内存的末尾之后写入, 你很可能会破坏这些数据结构. 系统通常会在你调用 `malloc` 或 `free`, 同时这些函数莫名其妙失败时, 才会检测到问题.

从中, 你可以得到的结论是, 安全内存管理需要设计和纪律. 如果你编写了一个库或者模块来分配内存, 你也应该提供释放它的接口, 同时, 内存管理应该从开始便作为 API 设计的一部分.

如果你使用一个库分配内存, 你应该在使用 API 时遵循纪律. 比如, 如果库提供了分配和释放存储的函数, 你应该先分配而不是先释放, 避免在一个没有执行 `malloc` 的内存块上调用 `free` 函数.

通常需要在安全内存管理和性能之间进行权衡. 比如, 内存错误最常见的一个来源便是超出数组边界进行写入. 解决这个问题最明显的方法是边界检查; 即每次数组访问都应该检查索引是否超出边界. 提供类似数组结构的高级库通常会执行边界检查. 但 C 数组和大多数底层库不会执行此类检查.

6.2 内存泄漏

还有一种内存错误, 可能需要注意, 也可能不需要在意. 如果你分配了一块内存, 但从未释放, 这便是 “内存泄漏”.

对有些程序来说, 内存泄漏可以接受. 比如, 你的程序分配内存, 执行计算, 然后退出, 那么可能没有必要释放分配的内存. 当程序退出时, 全部内存都会被操作系统释放掉. 在退出之前立刻释放内存可能感觉更加负责, 但这基本上是浪费时间.

但如果程序运行很长时间且存在内存泄漏, 它的总内存使用将无限增长. 这时候, 会发生以下情况:

- 某些时候, 系统会用光物理内存. 在没有虚拟内存的系统上, 再调用 `malloc` 会失败, 返回 NULL.
- 在具有虚拟内存的系统上, 操作系统可以将另一个进程的页从内存移动到磁盘, 然后为泄漏的进程分配其他空间. 我在 7.8节解释了这种机制.
- 单个进程可能存在分配空间的限额; 超出限额, `malloc` 会返回 NULL.
- 最终, 一个进程可能会填满其虚拟地址空间 (或可用部分). 然后, 便没有了跟多地址可供分配, `malloc` 则返回 NULL.

如果 `malloc` 返回 NULL, 但是你依然访问你认为已经分配的内存块, 那么会导致分段错误. 因此, 良好的编程风格应该是在使用 `malloc` 的结果之前先检查它. 一个方法是在每个 `malloc` 调用后添加一个类似下面的条件:

```
void *p = malloc(size);
if (p == NULL) {
    perror("malloc failed");
    exit(-1);
}
```

`perror` 声明在 `stdio.h` 中; 它会打印一个错误消息以及关于最后发生错误的附加信息.

`exit` 也在 `stdlib.h` 中声明, 它会导致进程终止. 参数是一个状态码, 表示进程终止方式. 按照惯例, 状态码 0 表示正常终止, -1 表示错误条件. 有时用不同代码表示不同错误条件.

错误代码检查可能令人厌烦, 而且令程序难以阅读. 你可以通过将库函数调用以及错误代码检查封装进自己的函数中, 来减轻这些问题. 比如, 下面是一个检查返回值的 malloc 封装函数.

```
void *check_malloc(int size)
{
    void *p = malloc (size);
    if (p == NULL) {
        perror("malloc failed");
        exit(-1);
    }
    return p;
}
```

因为内存管理如此之难, 大多数的大型程序, 比如 web 浏览器, 会泄漏内存. 想要查看系统上哪些程序使用了最多的内存, 你可以用 UNIX 程序 ps 和 top.

6.3 实现

当一个进程启动时, 系统会为文本段和静态分配的数据分配空间, 为堆栈分配空间, 以及包含动态分配数据的堆空间.

并非所有程序都是动态分配数据, 因此堆的初始大小可能很小, 甚至是零. 初始时, 堆仅包含一个空闲块.

当调用 malloc 时, 函数会检查是否可以找到足够大的空闲块. 如果无法找到, 便需要从系统申请更多内存. 执行此操作的函数是 sbrk, 它会设置 “程序断点”, 你可以将其视为一个指向堆末尾的指针.

当 sbrk 被调用, 操作系统会分配新的物理内存的页, 更新进程的页表, 并设置程序断点.

理论上, 一个程序可以直接调用 sbrk(无需使用 malloc), 并自行管理堆. 但 malloc 更易于使用, 同时对于大多数内存使用模式来说, 它速度更快, 而且内存使用率高.

若要实现内存管理 API (即, 函数 malloc, free, calloc, 和 realloc), 大部分 Linux 系统使用 ptmalloc, 其源于 Doug Lea 写的 dlmalloc. 至于描述实现的关键要点的简短论文可以从http://gee.cs.oswego.edu/dl/html/malloc.html获取.

对于程序员来说, 需要关注的最重要的要素是:

- `malloc` 的运行时间通常并不取决于块的大小, 但可能取决于存在多少空闲块. 同时, 无论有多少空闲块, `free` 通常都很快. 而因为 `calloc` 需要清除块中的每个字节, 所以其运行时间取决于块大小 (也就是空闲块的数量).

  `realloc` 有时很快, 比如要分配的空间小于当前块大小, 或者有可用空间来扩展当前块. 如果不是这样, 便需要将数据从旧块复制到新块; 这种情况下, 运行时间便取决于旧块的大小.

- 边界标签 (Boundary tags): 当 `malloc` 分配块时, 会在开始和结尾添加空间, 以存储关于块的信息, 包括其大小和状态 (占用或空闲). 这些数据位被称为 "边界标签". `malloc` 通过使用这些标签, 可以轻易从任何一个块, 获取内存中其前后相邻的块. 此外, 空闲块被链接为一个双向链表; 每个空闲块包含在 "空闲列表" 中指向下一个和前一个块的指针.

  边界标签和空闲列表指针便组成了 `malloc` 的内部数据结构. 这些数据结构和程序数据交错在一起, 所以程序错误很容易便会破坏他们.

- 空间开销 (Space overhead): 边界标签和空闲列表指针会占用空间. 大部分系统的最小块大小是 16 字节. 因此对于非常小的块, `malloc` 无法高效利用空间. 如果你的程序需要大量小结构, 可能在数组中分配它们比较高效.

- 碎片化 (Fragmentation): 如果你以各种大小分配和释放块, 则堆很可能会变得碎片化. 也就是, 空闲块很可能会被拆分成很多小碎片. 碎片化会浪费空间; 同时也会通过降低内存缓存的效率, 而减慢程序运行速度.

- 分箱和缓存 (Binning and caching): 空闲列表会按照大小分箱, 所以当 `malloc` 检索特定大小的块时, 知道在哪个箱中进行检索. 如果你释放一个块, 然后立刻分配一个同样大小的块, `malloc` 通常会很快.

Chapter 7

缓存

7.1 程序如何运行

为了理解缓存, 你需要了解计算机是如何执行程序的. 要想对这个主题有深入了解, 你需要学习计算机体系架构. 本章的目标是提供一个程序执行的简单模型.

当程序启动时, 代码 (或文本) 通常是位于硬盘或固态硬盘上. 操作系统会创建一个进程来运行程序, 然后 “加载器” 会从存储中将文本拷贝到主内存中, 并通过调用 `main` 以启动程序.

当程序运行时, 大部分的数据存储在主内存中, 但有些数据存储在寄存器中, 寄存器是 CPU 上的小的内存单元. 这些寄存器包括:

- 程序计数器 (PC), 其包含程序中下一条指令的内存地址.
- 指令寄存器 (IR), 其包含当前运行的机器码指令.
- 栈指针 (SP), 其包含当前函数 (包括其参数和局部变量) 的栈帧地址.
- 通用寄存器, 用于保存程序当前处理的数据.
- 状态寄存器, 或标志寄存器, 其包含当前计算的有关信息. 比如, 标志寄存器通常包含一个位, 如果前一操作的结果为零, 则该位便被设置.

当程序运行时, CPU 执行以下步骤, 被称为 “指令周期”:

- 取指 (Fetch): 从内存获取下一指令, 并存储到指令寄存器中.
- 解码 (Decode): CPU 的一部分, 叫做 “控制单元”, 解码指令并向 CPU 其他部分发送信号.

- 执行 (Execute): 来自控制单元的信号会引发适当计算的执行.

多数计算机可以执行几百个不同的指令, 叫做 “指令集”. 但是大部分的指令可以归纳为以下几种通用类别:

- 加载 (Load): 将值从内存传输到寄存器.
- 算术/逻辑 (Arithmetic/logic): 从寄存器加载操作数, 执行数学运算, 将结果存储到寄存器.
- 存储 (Store): 将值从寄存器传输到内存.
- 跳转/分支 (Jump/branch): 更改程序计数器的值, 令执行流程跳转到程序的另一个位置. 分支通常是有条件的, 也就意味着它们会检查标志寄存器中的标志, 且仅在标志被设置时才跳转.

一些指令集, 包括无处不在的 x86, 提供了一些将加载和算术运算组合在一起的指令.

在每个指令周期中, 一条指令都是从程序文本中读取. 此外, 典型的程序中大约一半的指令是加载或存储数据. 因此正是在这里, 存在计算机架构的一个基本问题: “内存瓶颈”.

在现在计算机中, 典型的核心处理器可以在不到 1 纳秒内执行一条指令. 而它与内存交换数据则需要 100 纳秒. 如果 CPU 需要等待 100 纳秒来获取下一条指令, 又等待 100 纳秒以加载数据, 那么它完成指令的速度相比于其理论可行速度要慢 200 倍. 对于许多计算任务来说, 限制速度的因素是内存, 而不是 CPU.

7.2 缓存性能

这个问题的解决方案, 或至少一个部分解决方案, 便是缓存. “缓存” 是一种小型, 高速的内存, 在物理上靠近 CPU, 通常位于同一个芯片上.

实际上, 现代计算机通常有多级缓存: 一级缓存, 是最小最快的, 可能有 1–2 MiB, 访问时间接近 1 纳秒; 二级缓存访问时间可能接近 4 纳秒; 而三级缓存可能需要 16 纳秒.

当 CPU 从内存加载数据, 会将拷贝保存在缓存中. 如果同样的值需要再次加载, CPU 会取用缓存中的拷贝, 无需等待内存.

最终缓存会被填满. 然后, 为了引入新数据, 我们需要先淘汰一些旧数据. 所以如果 CPU 加载了一个值, 很久之后再次加载, 这个值可能已经不在缓存中了.

许多程序的性能受限于缓存效率. 如果 CPU 需要的指令和数据通常都在缓存中, 程序便能以接近 CPU 满速的性能执行. 如果 CPU 频繁需要不在缓存中的数据, 程序便受限于内存速度了.

缓存的 “命中率 (hit rate)”, h, 是在内存访问时找到数据在缓存中的比例; “未命中率 (miss rate)”, m, 是在内存访问时必须访问内存的比例. 如果处理缓存命中的时间是 T_h, 同时缓存未命中的时间是 T_m, 每次内存访问的平均时间为

$$hT_h + mT_m$$

同样的, 我们可以将 “未命中损失” 定义为处理未命中缓存的额外时间, $T_p = T_m - T_h$. 那么平均访问时间便是

$$T_h + mT_p$$

当未命中率较低时, 平均访问时间会接近于 T_h. 也就是说, 程序运行中内存访问的速度接近于缓存速度.

7.3 局部性

当程序首次读取一个字节, 缓存通常会加载包含请求字节及其相邻字节的 “块” 或 “行” 数据. 如果程序继续读取一个相邻字节, 那么它已经在缓存中了.

举个例子, 假设块大小是 64 B; 你读取一个长度 64 的字符串, 字符串的首个字节恰好位于块的开头. 当你加载第一个字节时, 会遇到缺失损失, 但之后的整个字符串都会存在于缓存中. 读取整个字符串, 命中率将会是 63/43, 大约 98%. 如果字符串横跨两个块, 你会遇到 2 次缺失损失. 但即便如此, 命中率仍将是 62/64, 接近 97%. 如果你之后再次读取相同字符串, 命中率将是 100%.

另一方面, 如果程序以不可预料的方式跳转, 从内存的分散位置读取数据, 并几乎不会再次访问同样位置, 那缓存性能将会变差.

程序多次使用相同数据的趋势叫做 “时间局部性”. 而使用临近位置数据的趋势叫做 “空间局部性”. 幸运的是, 很多程序天然地表现出了两种局部性:

- 多数程序包含没有跳转和分支的代码块. 在这些块中, 指令顺序运行, 所以访问模式具有空间局部性.
- 在循环中, 程序多次执行相同的指令, 因此访问模式具有空间局部性.
- 一个指令的结果通常会立刻用作下一个指令的操作数, 所以数据访问模式具有时间局部性.

- 当程序执行函数时, 其参数和局部变量都被存储在堆栈上; 访问这些值便具有空间局部性.
- 读取或写入数组元素的一个最常见的模式便是顺序操作; 这种模式也具有空间局部性.

下一节会探讨程序的访问模式和缓存性能之间的关系.

7.4 测量缓存性能

当我在加利福尼亚州伯克利分校攻读研究生时, 我是 Brian Harvey 的计算机架构课程的教学助教. 其中一个我最喜欢的练习是一个通过迭代数组来测量读写元素的平均时间的程序. 通过改变数组大小, 大概率可以推断出缓存大小, 块带下, 以及其他一些属性.

我关于这个程序的修订版本位于本书存储库的*缓存*目录中 (见第 0.1节)

程序中最重要的部分是这个循环:

```
    iters = 0;
    do {
        sec0 = get_seconds();

        for (index = 0; index < limit; index += stride)
            array[index] = array[index] + 1;

        iters = iters + 1;
        sec = sec + (get_seconds() - sec0);

    } while (sec < 0.1);
```

内部的 `for` 循环会遍历数组. `limit` 则决定了循环遍历数组的范围; `stride` 决定跳过多少元素. 例如, 如果 `limit` 是 16, 而 `stride` 是 4, 循环将访问第 0, 4, 8, 和 12 个元素.

`sec` 会记录内部循环使用的 CPU 总时间. 外部循环则运行到 `sec` 超过 0.1 秒为止, 而这时间已足以令我们用较高的精度计算平均时间.

`get_seconds` 使用系统调用 `clock_gettime`, 并转换为秒, 然后以 `double` 格式为结果返回:

```
double get_seconds(){
    struct timespec ts;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
    return ts.tv_sec + ts.tv_nsec / 1e9;
}
```

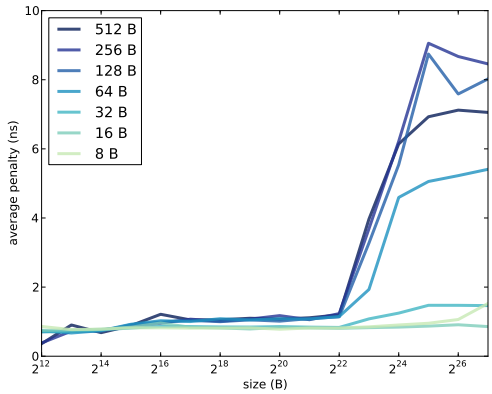

图 7.1: 数组大小和步长的平均缺失损失函数.

为了排除访问数组元素的时间, 程序运行了第二个循环, 和第一个循环几乎相同, 唯一的差异在于内部循环不涉及数组; 其始终递增同一个变量:

```
iters2 = 0;
do {
    sec0 = get_seconds();

    for (index = 0; index < limit; index += stride)
        temp = temp + index;

    iters2 = iters2 + 1;
    sec = sec - (get_seconds() - sec0);

} while (iters2 < iters);
```

第二个循环与第一个循环迭代相同次数. 每次迭代之后, 它从 `sec` 变量中减去 经过的时间. 当循环结束, `sec` 包含所有数组访问的总时间, 减去递增 `temp` 花费的时间. 这个差值便是所有访问产生的总缺失损失. 最后, 我们通过除以访问次数, 得到每次访问的平均缺失损失, 单位为纳秒:

```
sec * 1e9 / iters / limit * stride
```

如果你编译并运行 `cache.c`, 你会看到如下输出:

```
Size:     4096 Stride:        8 read+write: 0.8633 ns
Size:     4096 Stride:       16 read+write: 0.7023 ns
Size:     4096 Stride:       32 read+write: 0.7105 ns
Size:     4096 Stride:       64 read+write: 0.7058 ns
```

如果您已安装 Python 和 `matplotlib` , 便可以使用`graph_data.py` 来绘制结果. 图 7.1展示了我在戴尔 Optiplex 7010 上运行时的结果. 请注意, 数组大小和步长的单位是字节, 而不是数组元素的数量。

花一分钟来考虑这个图表, 看看您能从中推断出哪些关于缓存的信息. 以下是一些需要考虑的事项:

- 程序会遍历多次数组, 所以会有大量的时间局部性. 如果缓存包含了全部数据, 我们便可以预计平均缺失损失接近于 0.
- 当步长为 4 个字节, 我们会读取数组每个元素, 所以程序具有大量空间局部性. 如果块大小足够容纳 64 个元素, 即使缓存没有容纳整个数组, 命中率也是 63/64.
- 如果步长等于 (或大于) 块大小, 则空间局部性为零, 因为我们每次读取一个块, 仅仅需要访问一个元素. 如此情况, 我们可以预期会看到最大的缺失损失.

总之, 如果数组小于内存大小或步长小于块大小, 我们便可以相信内存性能良好. 只有当数组大于内存大小且步长较大时, 性能才会下降.

图 7.1中, 只要数组小于 2^{22} B, 对于所有步长, 内存性能都很好. 我们可以推断缓存大小接近 4 MiB; 实际上, 根据规格, 是 3 MiB.

当步长是 8, 16, 或 32 B 时, 缓存性能良好. 当步长 64B 时, 性能开始下降, 对于较大的步长, 内存损失大约 9 纳秒. 我们可以推断块大小接近 128B.

很多处理器使用 “多级缓存”, 其中包括一个小而快的缓存和一个大而慢的缓存. 在这个示例中, 当数组大小大于 2^{14} B 时, 缺失损失会略有增加, 所以很可能处理器还有一个 16KB 的缓存, 其访问时间小于 1 纳秒.

7.5 缓存性能编程

内存缓存是通过硬件实现, 所以多数时候, 程序员不需要对其了解太多. 但如果知道缓存的工作原理, 你可以写出更高效使用缓存的程序代码.

例如, 假设你正在处理一个大数组, 对其进行一次遍历, 同时对每个元素执行数个操作, 相比对数组遍历多次要更快一些.

如果你正在处理一个行存储的二维数组, 那么你遍历数组时, 逐行操作, 同时步长等于元素大小, 相比于逐列操作, 步长等于行长度的遍历方式, 要更快一些.

链式数据结构并不总是表现出空间局部性, 因为节点在内存中并不一定是连续的. 但如果你一次性分配了很多节点，它们通常在堆的相邻位置. 或者, 更好的操作是, 你分配一个节点数组, 你便知道他们会是连续的.

递归策略, 例如归并排序, 通常都有很好的缓存操作, 因为他们会将大数组拆成小片段处理. 有时候这些算法还可以通过微调来更充分地利用缓存操作.

对于性能优先的应用来说, 可以针对缓存大小, 块大小, 以及其他硬件特性来设计算法. 这样的算法叫做 “缓存感知” 算法. 而缓存感知算法的明显缺点则是他们是针对特定硬件的.

7.6 内存层次架构

在本章的某个节点, 你可能会遇到如下问题: “如果缓存比内存要快很多, 为什么不制造一个足够大的缓存来替代内存?”

在不涉及过多计算机架构的情况下, 有两个原因: 电子学和经济学. 缓存很快是因为它们小同时靠近 CPU, 这便最小化了因电容和信号传播而造成的延迟. 如果你制造一个大缓存, 速度便会慢很多.

此外, 缓存占据了处理器芯片上的空间, 而更大的芯片往往更加昂贵. 主存通常是动态随机访问存储 (DRAM), 每比特只需要一个晶体管和一个电容, 所以可以在相同空间内装入更多的内存. 但这种内存实现方式比缓存的实现方式在速度上要慢很多.

译者注

位和比特的翻译实在令我头疼, 后续将再次更正和优化.

此外, 主内存通常封装在一个双列直插内存模块 (DIMM) 中, 这个模块包括 16 个或者更多的芯片. 而多个小芯片相比一个大芯片价格上要更加便宜.

速度, 大小和成本之间的权衡是缓存实现的根本原因. 如果有一种快速, 容量大而且便宜的内存技术, 那我们便无需其他考虑了.

这个原则同样适用于存储和内存. 固态硬盘 (SSD) 速度很快, 但相比于硬盘驱动器 (HDD) 更加昂贵, 所以倾向于更小. 磁带驱动器比硬盘驱动器更慢, 但它们可以相对廉价地存储大量数据.

下表展示了每种技术的访问时间, 容量以及成本.

设备	访问时间	典型容量	成本
寄存器 (Register)	0.5 ns	256 B	?
缓存 (Cache)	1 ns	2 MiB	?
动态随机访问存储 (DRAM)	100 ns	4 GiB	$10 / GiB
固态硬盘 (SSD)	10 μs	100 GiB	$1 / GiB
硬盘驱动器 (HDD)	5 ms	500 GiB	$0.25 / GiB
磁带 (Tape)	minutes	1–2 TiB	$0.02 / GiB

译者注

对于一些常用词汇, 例如 `minutes`, `ns` 等, 如果也强求翻译, 那我建议先补充一下英文基础知识.

寄存器的数量和大小取决于架构的细节. 当前的计算机拥有大约 32 个通用寄存器, 每个存储一个 “单词”. 在 32 位计算机上, 一个单词是 32 比特或 4 字节. 在 64 位计算机上, 一个单词是 64 比特或 8 字节. 所以寄存器文件的总大小是 100–300 字节.

寄存器和缓存的成本是很难量化的. 它们会增加芯片的成本, 而消费者很难直接看到这些费用.

对于表中的其他数字, 我查阅了在线计算机硬件商店中销售的典型硬件的规格信息. 当你阅读本书时, 这些数字可能已经过时, 但它们可以让你了解, 在某个时间节点, 不同硬件的性能和成本之间的差距是什么样子.

这些技术构成了 “内存层次结构”(注意, 此处的 “内存” 也包括存储). 层次结构中的每一层级相比于上一层级都要更大, 更慢. 某种意义上, 每层都充当了下一层的缓存. 你可以将主存看作固态硬盘和硬盘驱动器上永久存储的程序和数据的缓存. 如果你正在处理存储在磁带上的大型数据集, 你可以使用硬盘驱动器来逐次缓存单个数据子集.

7.7 缓存策略

内存层次结构提出了一个缓存思维框架. 在每个层级, 我们都需要解决四个基本的缓存问题:

- 谁负责在层次结构中上下移动数据? 在层次结构顶部, 寄存器的分配通常由编译器完成. CPU 上的硬件会处理内存缓存. 当用户执行程序和打开文件时, 数据会默默地从存储移动到内存中. 但操作系统也会在内存和存储之间来回移动数据. 在层次结构的底部, 管理员需要清楚地在磁盘和磁带之间移动数据.
- 移动了什么? 通常层次结构顶层的块较小, 底层块较大. 在缓存中, 一个典型的块大小是 128 B. 内存中的页大小是 4KiB, 但操作系统从磁盘读取一个文件时, 它可能一次读取 10 个或 100 个块.
- 什么时候移动数据? 在最基本的缓存中, 数据首次使用时会被移动进缓存. 但很多缓存会使用某种形式的 “预取”, 意味着数据在被明确请求之前便被加载. 我们已经看到过一种预取方式: 当只请求部分块时也会加载整个块.

- 数据在缓存的哪个地方? 当缓存满了, 我们便需要先清除一些数据, 才能再次添加数据 (欲先添之，必先减之). 理想情况是, 我们希望保留即将再次使用的数据, 替掉我们不再使用的数据.

这些问题的答案构成了 “缓存机制”. 在层次结构顶部附近, 缓存机制往往简单, 因为它们必须要快速, 并能用硬件实现. 靠近层次结构底部, 可以有更多的决策时间, 而设计良好的策略则会影响重大.

大部分的缓存机制基于历史重演的原则; 如果我们有近期的历史信息, 我们便可以用它预测不久的将来. 例如, 如果一个数据块最近被使用了, 我们可以预期它很快会再次被使用. 这个原则引出了一个替换策略, 叫做 “最近最少使用” 策略, 或 LRU, 这个策略会将缓存中最近未被使用的数据块移除. 有关此主题的更多信息, 请参阅`http://en.wikipedia.org/wiki/Cache_algorithms`.

7.8 分页

在具有虚拟内存的系统中, 操作系统可以在内存和存储之间来回移动页面. 如我在 Section 6.2节提到的, 这个机制称作 “分页”, 有时也叫做 “交换”.

下面是这个过程的工作原理:

1. 假设进程 A 调用 `malloc` 分配一块内存. 如果堆中没有足够大小的空闲空间, `malloc` 会调用 `sbrk` 来请求操作系统分配更多内存.
2. 如果物理内存中有一个空闲页, 操作系统会将其添加到进程 A 的页表中, 创建一个新的有效虚拟地址范围.
3. 如果没有空闲页, 分页系统会选择一个属于进程 B 的 “受害页”. 将受害页中的内容从内存复制到磁盘, 然后修改进程 B 的页表, 表明该页已被 “交换出去”.
4. 一旦进程 B 的数据被写入, 该页便可以被重新分配给进程 A. 为了防止进程 A 读取进程 B 的数据, 该页应该被清空.
5. 此时, 对 `sbrk` 的调用便可以返回, 在堆中为 `malloc` 提供额外空间. 然后 `malloc` 分配请求的内存块并返回, 进程 A 可以继续.
6. 当进程 A 完成, 或被中断, 调度器可能会允许进程 B 继续. 当进程 B 访问一个已被交换出去的页时, 内存管理单元会注意到该页是 “无效” 的, 并引发一个中断.
7. 当操作系统处理中断时, 会发现该页已经被交换出去, 所以它会将该页从磁盘转移到内存.

8. 一旦该页被交换进来, 进程 B 便可以继续.

当分页正常工作, 它可以极大提高物理内存的利用率, 允许更多进程在较少空间运行. 原因如下:

- 大部分的进程不会使用分配给它们的全部内存. 文本段的许多部分从不会被执行, 或者执行一次后再也不被执行. 这些页可以被交换出去而不会引发任何问题.
- 如果程序存在内存泄漏, 可能会遗留被分配的空间, 再也不访问它. 通过交换这些页出去, 操作系统可以有效堵住内存泄漏.
- 在大部分系统中, 有一些像守护进程那样, 大部分时间处于空闲状态, 只偶尔被 “唤醒” 以响应事件. 当这些处于空闲时, 可以被交换出去.
- 用户可能打开了很多窗口, 但同一时间只有少数几个处于活动状态. 那么不活跃的进程便可以被交换出去.
- 此外, 可能有很多进程运行着相同程序. 这些进程可以共享同一文本和静态段, 避免在物理内存中保留多个副本.

如果你将分配给所有进程的内存相加, 可能远远超出物理内存的大小, 但系统依然表现良好.

在一定程度上是这样的.

当进程访问交换出去的页时, 需要从磁盘获取数据, 而这会耗费几毫秒的时间. 这种延迟通常是显而易见的. 如果你让一个窗口长时间处于空闲, 然后突然切换回去, 也许会启动较慢, 你可能会在页交换回来时听到磁盘驱动器运转起来.

偶尔出现这种延迟是可以接受的, 但如果你有太多的进程占用了太多的空间, 它们会开始相互影响. 当进程 A 运行, 它会驱逐进程 B 需要的页. 当进程 B 运行, 它又会驱逐进程 A 需要的页. 当这种情况发生, 两个进程都会变得缓慢, 系统变得不响应. 这种现象叫做 “抖动”.

理论上, 操作系统可以通过检测页的增加, 并堵塞或者杀掉进程, 直到系统再次响应, 来避免抖动. 但据我所知, 多数系统没有这样做, 或者做得不好; 通常情况是让用户限制他们物理内存的使用, 或在抖动发生时, 尝试恢复而已.

译者注

此处可以思考, 为什么多数操作系统不采取更有好的方式来避免抖动, 而是采取相对强硬或者暴力的措施?

Chapter 8

多任务

当前很多系统中, CPU 包括多个核, 也就表示系统可以同时运行多个进程. 另外, 每个核都能够进行 “多任务处理”, 意味着可以从一个进程快速切换到另一个进程, 从而产生多个进程同时运行的错觉.

操作系统实现多任务处理的部分是 “内核”. 在坚果或种子中, 内核是最内层的部分, 被壳所包围. 操作系统中, 内核是软件最底层的部分, 外部被其他层环绕, 其中包括一个叫做 “shell” 的接口层. 计算机专家往往喜欢使用这种类比的方式.

在最基本的功能中, 内核的工作是处理中断. 一个 “中断” 就是一个事件, 这个事件会停止正常指令周期, 并使执行流跳转到被称为 “中断处理程序” 的特定代码部分.

硬件中断是由设备向 CPU 发送信号引发的. 例如, 网络接口会在数据包到达时引起中断, 或着, 数据传输结束会, 磁盘驱动器也会触发中断. 大部分的系统还有计时器会在定期, 或某段时间后触发中断.

软件中断通常由运行中的程序触发. 例如, 如果某个指令因为某种原因无法完成, 便可能触发中断, 以便操作系统可以处理该状况. 有些浮点错误, 比如除零操作, 也是通过中断来处理.

当程序需要访问硬件设备时, 会进行一次**系统调用**, 类似于函数调用, 但不同之处在于, 它不会跳转到函数的开头, 而是执行某个特殊指令触发中断, 从而令执行流跳转到内核. 内核读取系统调用的参数, 执行请求的操作, 然后恢复被中断的进程.

8.1 硬件状态

处理中断需要硬件和软件之间的协作. 当中断发生时, 可能 CPU 正有多条指令在运行, 寄存器中正存储着数据, 以及其他的**硬件状态**.

通常, 硬件负责将 CPU 带到一个一致性的状态; 例如, 每个指令都应该完成, 或者如同从未开始. 任何指令都不应该被留在半完成的状态. 此外, 硬件负责保存程序计数器 (PC), 以便内核知道从何处恢复.

然后, 通常情况下, 中断处理程序需要在任何可能改变某个硬件状态的操作前, 保存其余的硬件状态, 然后在被中断进程恢复前, 还原被保存的状态.

以下是此事件序列的概要:

1. 当中断发生, 硬件将程序计数器保存在某个特殊的寄存器中, 同时跳转到适当的中断处理程序.

2. 中断处理程序将程序计数器和状态寄存器, 以及计划使用的数据寄存器的任何数据, 一并存储在内存中.

3. 中断处理程序运行处理中断所需的所有代码.

4. 然后它会恢复保存的寄存器的内容. 最后, 它恢复中断程序的程序计数器, 这会令执行流回到中断指令处.

如果这个机制运转正常, 通常情况下, 被中断的进程不会觉察到中断发生, 除非它检测到指令间的事件变化.

8.2 上下文切换

中断处理程序之所以快速, 是因为它们不必存储整个硬件状态; 只需要存储它们后续使用的寄存器.

但是当中断发生时, 内核并不总是会恢复中断的进程. 它也可以切换到另一个进程. 这种机制叫做 “上下文切换”.

一般来说, 内核并不知道进程要使用哪个寄存器, 所以需要保存所有寄存器. 此外, 当内核切换到一个新的进程, 可能需要清理存储在内存管理单元中的数据 (见第 3.6节). 在上下文切换后, 新进程需要一些时间将数据加载到缓存中. 因此, 上下文切换相对较慢, 大约需要上千个时间周期, 或者几微秒.

在一个多任务系统中, 每个进程都会运行一小段时间, 这段时间称为 “时间片” 或 “量子”. 在上下文切换期间, 内核会设置一个硬件定时器, 在时间片结束时引发中断. 当中断发生时, 内核可以切换到另一个进程或者恢复中断的进程. 操作系统中做这一决策的部分叫做 “调度程序”.

8.3 进程生命周期

当进程被创建时, 操作系统会分配一个包含进程相关信息的数据结构, 被称作“进程控制块 (process control block)” 或 PCB. 此外, PCB 还会追踪进程状态, 其中如下:

- 运行状态, 即进程当前正在核心上运行.
- 就绪状态, 即进程可以运行但还未运行, 通常是因为可运行的进程数超过了核心数.
- 阻塞状态, 即进程暂时无法运行, 通常因要等待一个将来的事件, 比如网络通信或磁盘读取.
- 完成状态, 即进程已经完成, 但还有退出状态信息未被读取.

下面是导致进程从一个状态转换到另一个状态的事件:

- 当运行的程序执行类似 `fork` 的系统调用时, 就会创建一个进程. 系统调用结束时, 新进程通常处于就绪状态. 然后调度程序会恢复原始程序 (“父进程”) 或者启动新进程 (“子进程”).
- 当一个进程被调度程序启动或恢复, 其状态便会从就绪变为运行.
- 如果一个进程执行一个无法立刻完成的系统调用时, 像磁盘请求, 便会被堵塞住, 同时调度程序通常会选择另一个进程执行.
- 当一个像磁盘请求这样的操作完成, 会触发中断. 中断处理程序会确定等待该请求的进程, 并将其状态从堵塞切换为就绪. 然后调度程序会选择是否恢复未堵塞的进程.
- 当一个进程调用 `exit`, 中断处理程序会将退出码存储在 PCB, 并将进程状态更改为完成.

8.4 调度

正如在第 2.3节中看到的, 计算机上可能有数百个进程, 但通常大多数进程都是被阻塞的。大多数时候, 只有少数进程处于就绪或运行状态. 当中断发生时, 调度程序决定启动或恢复某个进程。

在工作站或笔记本电脑上, 调度程序的主要目标是最小化响应时间, 也就是说, 计算机应该对用户操作快速做出响应. 在服务器上, 响应时间也很重要, 但此外, 调度程序可能会尝试最大化吞吐量, 即单位时间内完成的请求数量.

通常, 调度程序并不清楚进程正在做什么, 因此其决策基于一些启发式规则:

- 进程可能受限于不同的资源. 进行大量计算的进程是 CPU 密集型, 表示其运行时间取决于取得的 CPU 时间. 从网络或磁盘读取数据的进程是 I/O 密集型, 意味着如果数据输入输出速度越快, 便运行越快, 但即使取得更多 CPU 时间, 也不会更快. 最后, 与用户交互的进程很可能会被堵塞, 其大部分时间在等待用户操作.

  操作系统有时可以根据进程过去的行为, 对其进行分类, 并进行相应调度. 例如, 当一个交互进程解除堵塞时, 它应该立刻运行, 因为用户可能正在等待一个回复. 另一方面, 一个长时间运行的 CPU 密集型进程, 对于时间便不再敏感.

- 如果一个进程可能只会运行一小段时间, 然后会发出一个阻塞请求, 它可能应该立即运行, 原因有二: (1) 如果请求需要一些时间才能完成, 我们应该尽快开始; (2) 让一个耗时长的进程等待一个耗时短的进程, 要比反过来好很多.

  打个比方, 假设你正在做苹果派. 准备外皮需要 5 分钟, 但之后需要冷藏半小时. 准备馅料需要 20 分钟. 如果你先准备外皮, 你可以在外皮冷藏的时候准备馅料, 35 分钟就可以完成苹果派. 如果你先准备馅料, 整个过程需要 55 分钟.

  译者注

  好比制作包子, 和面需要 5 分钟, 之后醒面需要半小时, 准备馅需要 20 分钟. 如果先和面, 而后在醒面的时候准备馅, 35 分钟即可完成. 如果你先弄馅, 你需要 55 分钟才能完成.

多数调度程序使用基于优先级的调度策略, 其中每个进程都有一个根据时间变化调整的优先级. 当调度程序运行时, 它会选择具有最高优先级的可运行进程.

以下是确定进程优先级的一些因素:

- 进程通常以相对较高的优先级启动, 以便快速开始运行.
- 如果进程在它的时间片结束前, 产生了请求并堵塞, 很可能是因为交互或 I/O 密集, 因此其优先级应该提高.
- 如果一个进程运行了整个时间片, 它很可能是长时间运行且 CPU 密集, 所以其优先级应该降低.
- 如果一个任务堵塞较长时间后, 变为就绪状态, 它应该获得优先级提升, 以便其能快速响应等待的任意事件.

- 如果 A 进程因等待 B 进程而被堵塞, 例如, 它们通过管道连接, B 进程的优先级应该提高.
- 通过 `nice` 系统调用, 进程可以降低 (而不是提高) 其自身优先级, 程序员也可以向调度程序显式传递信息, 从而对进行优先级进行调整.

对多数负载正常的系统来说, 调度算法对性能没有实质性的影响. 通常简单的调度策略已绰绰有余.

8.5 实时调度

然而, 对于与现实世界交互的程序来说, 调度非常重要. 例如, 一个从传感器读取数据并控制电机的程序, 可能需要以最小频率完成重复任务, 并在某个最大响应时间限制内完成外部事件的响应. 这些要求通常用必须在 “截止日期” 内完成的 “任务” 来表示.

调度任务以满足截止时间, 称做 “实时调度”. 对于某些应用来说, 像 Linux 这种通用操作系统可以通过修改, 满足实时调度. 这些更改可能包括:

- 为任务优先级控制提供丰富 API.
- 修改调度器, 确保最高优先级的进程拥有固定时间.
- 重新调整中断处理程序, 以保证最大完成时间.
- 修改锁以及其他同步机制 (下一章介绍), 以允许高优先级任务可以抢占低优先级任务资源.
- 选择能保证最大完成时间的动态内存分配实现.

对于更加苛刻的应用, 特别是实时响应关乎生死存亡的领域, “实时操作系统” 提供了专门的能力, 通常比通用操作系统拥有更加简洁的设计.

译者注

实现实时操作系统, 需要对中断处理, 以及不同优先级的进程进行极为精细的时间控制, 避免超出最大完成时间, 从而保证操作系统的实时性能.

例如在电力行业应用的操作系统, 对实时性要求很高, 因为一旦电路出现问题, 即使系统响应延迟几毫秒, 在电的高速传输下，也会影响到数百公里, 甚至上千公里的范围.

Chapter 9

线程

当我在第 2.3节提到线程时, 我说线程就是一种进程. 现在我会提供一种更加谨慎的解释.

当你创建一个进程时, 操作系统会创建一个新的地址空间, 其中包括文本段, 静态段, 以及堆; 也会创建一个新的 “执行线程”, 其包括程序计数器和其他硬件状态, 以及调用堆栈.

我们目前见到的进程都是 “单线程的”, 也就是说每个地址空间中仅运行一个执行线程. 本章中, 我们会学习 “多线程的” 进程, 即同一地址空间中运行多个线程的进程.

在单个进程内, 所有线程共享相同的文本段, 因此它们执行相同的代码. 但不同的线程通常执行代码的不同部分.

同时, 它们共享同样的静态段, 所以如果一个线程修改了一个全局变量, 其他线程也会看到变化. 它们还共享堆, 所以线程可以共享动态分配的块.

但每个线程拥有自己的堆栈, 因此线程可以互不干扰地调用函数. 通常线程不会访问彼此的局部变量 (有时候它们也不能).

本章的样例代码在本书的代码仓库中, 名为 `counter` 的目录下. 有关下载代码的更多信息, 见第 0.1节.

9.1 创建线程

在 C 语言中, 最流行的线程标准是 POSIX 线程, 简称为 Pthread. POSIX 标准定义了一个线程模型, 以及创建和控制线程的接口. 多数 UNIX 版本都提供了 Pthread 的实现.

使用 Pthread 就像使用多数 C 库一样:

- 在程序的开头包括头文件.
- 编写调用基于 Pthread 定义的函数代码.
- 当编译程序时, 将其与 Pthread 库链接.

比如我的例子, 包含以下头文件:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```

前两个是标准库; 第三个是用于 Pthreads 的, 第四个是用于信号量的. 若要在 gcc 中使用 Pthread 库编译, 你可以在命令行使用 -l 选项:

```
gcc -g -O2 -o array array.c -lpthread
```

这将编译名为 array.c 的源文件, 包含调试信息和优化, 链接 Pthread 库, 以及生成名为 array 的可执行文件.

9.2 创建线程

创建线程的 Pthread 函数叫做pthread_create. 下面函数展示了如何使用它:

```
pthread_t make_thread(void *(*entry)(void *), Shared *shared)
{
    int n;
    pthread_t thread;

    n = pthread_create(&thread, NULL, entry, (void *)shared);
    if (n != 0) {
        perror("pthread_create failed");
        exit(-1);
    }
    return thread;
}
```

make_thread 是我编写的一个包装函数, 以便 pthread_create 更易用, 同时也提供错误检查.

pthread_create 的返回类型是 pthread_t, 你可以将其视为新线程的标识或“句柄”.

如果 pthread_create 执行成功, 会返回 0, make_thread 则会返回新线程的句柄. 如果有错误发生, pthread_create 会返回错误码, 而make_thread 会打印错误信息并退出.

make_thread 的参数需要一些解释. 从第二个参数开始, Shared 是我定义的一个结构体, 用于存储线程间共享的值. 下面的 typedef 语句会创建新的类型:

```
typedef struct {
    int counter;
} Shared;
```

在这种情况下, 唯一的共享变量是 counter. 而 make_shared 则会为 Shared 分配空间并初始化其内容:

```
Shared *make_shared()
{
    Shared *shared = check_malloc(sizeof (Shared));
    shared->counter = 0;
    return shared;
}
```

现在, 我们有了一个共享的数据结构, 让我们回到make_thread. 第一个参数是个指向函数的指针, 被指向的函数会接收一个 void 指针, 同时返回一个 void 指针. 如果这种类型声明的语法令你眼花缭乱, 那你并不孤单. 无论如何, 这个参数的目的是指定新线程开始执行的函数. 依照惯例, 这个函数被命名为 entry:

```
void *entry(void *arg)
{
    Shared *shared = (Shared *) arg;
    child_code(shared);
    pthread_exit(NULL);
}
```

entry 的参数必须声明为一个 void 指针, 但在这个程序中, 我们知道它只是一个指向 Shared 结构的指针. 因此我们进行相应的类型转换, 然后将其传递给执行实际工作的 child_code.

作为一个简单的例子, child_code 会打印共享计数器的值, 并递增它.

```
void child_code(Shared *shared)
{
    printf("counter = %d\n", shared->counter);
    shared->counter++;
}
```

当 child_code 返回时, entry 调用pthread_exit, 其用来传递参数给与此线程相连接的线程. 这个例子中, 子线程没什么需要做的, 所以我们传递了 NULL.

最后, 下面是创建子线程的代码:

```
    int i;
    pthread_t child[NUM_CHILDREN];

    Shared *shared = make_shared(1000000);

    for (i=0; i<NUM_CHILDREN; i++) {
        child[i] = make_thread(entry, shared);
    }
```

NUM_CHILDREN 是一个编译时间常量, 用来决定子线程的数量. child 是一个线程句柄的数组.

9.3 线程等待

当一个线程希望等待另一个线程完成, 它会调用 pthread_join. 下面是我对 pthread_join 的封装:

```
void join_thread(pthread_t thread)
{
    int ret = pthread_join(thread, NULL);
    if (ret == -1) {
      perror("pthread_join failed");
      exit(-1);
    }
}
```

参数是你等待的线程的句柄. 封装函数的全部作用是调用 pthread_join, 并检查结果.

任何线程都可以等待任何其他线程, 但在最常见的模式中, 父线程会创建并等待所有的子线程. 继续上一章的示例, 下面是等待子线程的代码:

```
    for (i=0; i<NUM_CHILDREN; i++) {
        join_thread(child[i]);
    }
```

这个循环按照子线程创建的顺序依次等待. 虽然无法保证子线程按照顺序完成, 但循环依然会正确工作. 如果一个子线程完成较晚, 循环可能需要等待, 而其他子线程可能会在此期间完成. 但无论如何, 只有在所有子线程结束后, 循环才会退出.

如果你下载了本书的代码库 (参见第 0.1节), 你会在 counter/counter.c 看到这个示例. 你可以像这样编译和运行它:

```
$ make counter
gcc -Wall counter.c -o counter -lpthread
$ ./counter
```

当我用 5 个子线程运行时, 我得到以下输出:

```
counter = 0
counter = 0
counter = 1
counter = 0
counter = 3
```

当你又运行它, 你可能会得到不同结果. 如果你再次运行, 你可能每次都会得到不同的结果. 发生了什么?

9.4 同步错误

前一个程序的问题在于, 子线程在没有同步的情况下, 访问了共享变量 counter. 所以任何线程在任意线程对 counter 递增之前, 都可以读到相同的值.

下面是可以解释上一节输出的事件序列:

```
Child A reads 0
Child B reads 0
Child C reads 0
Child A prints   0
Child B prints   0
Child A sets counter=1
Child D reads 1
Child D prints   1
Child C prints   0
Child A sets counter=1
Child B sets counter=2
Child C sets counter=3
Child E reads 3
Child E prints   3
Child D sets counter=4
Child E sets counter=5
```

每次运行程序, 线程都可能在不同的点被中断, 或者调度器可能会选择不同的线程运行, 所以事件序列, 以及结果, 都可能不同.

假如我们希望施加一些顺序. 例如, 我们可能希望每个线程读取 counter 的不同值, 并递增它, 以便 counter 的值可以反映执行child_code的线程数量.

为了强制满足这个需求, 我们使用 “互斥锁”, 这是一个为一段代码保证 “互斥访问” 的对象; 也就是说, 同一时间只有一个线程可以执行这段代码.

我已经编写了一个提供互斥对象, 称作 mutex.c 的小模块. 我先展示如何使用它; 然后解释其如何工作.

下面是使用互斥锁来同步线程的child_code 的一个版本:

```
void child_code(Shared *shared)
{
    mutex_lock(shared->mutex);
    printf("counter = %d\n", shared->counter);
    shared->counter++;
    mutex_unlock(shared->mutex);
}
```

在任何线程访问 counter 之前, 它会 “锁定” 互斥锁, 而这将会阻止所有其他线程. 假如线程 A 锁定了互斥锁, 并且执行到了child_code的中间部分. 如果线程 B 到达并执行mutex_lock, 则会被堵塞.

当线程 A 执行结束, 它会执行mutex_unlock, 而这会允许线程 B 继续执行. 实际上, 线程顺序执行child_code, 一次一个, 它们不会相互干扰. 当我用 5 个子线程执行这段代码, 会得到:

```
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
```

这满足了要求. 为了让这个解决方案生效, 我需要将互斥锁加到 Shared 结构中:

```
typedef struct {
    int counter;
    Mutex *mutex;
} Shared;
```

同时在make_shared中对其初始化:

```
Shared *make_shared(int end)
{
    Shared *shared = check_malloc(sizeof(Shared));
    shared->counter = 0;
    shared->mutex = make_mutex();   //-- this line is new
    return shared;
}
```

本节的代码位于counter_mutex.c中. Mutex 定义在 mutex.c 内, 我会在下一节解释.

9.5 互斥锁

我对 Mutex(互斥锁) 的定义是对pthread_mutex_t类型的封装, 该类型定义在了 POSIX 线程 API 中.

若要创建一个 POSIX 互斥锁, 你需要为pthread_mutex_t类型分配空间, 然后调用 pthread_mutex_init.

该 API 的一个问题是 pthread_mutex_t表现得像一个结构, 所以如果你将它作为参数传递, 它会创建一个副本, 而这会导致互斥锁的行为错误. 为了避免这种情况, 你需要通过地址传递pthread_mutex_t.

我的代码简化了这一过程. 定义了一个类型, Mutex, 只是pthread_mutex_t的更易读的名字:

```
#include <pthread.h>

typedef pthread_mutex_t Mutex;
```

然后定义了make_mutex, 它会分配空间并初始化互斥锁:

```
Mutex *make_mutex()
{
    Mutex *mutex = check_malloc(sizeof(Mutex));
    int n = pthread_mutex_init(mutex, NULL);
    if (n != 0) perror_exit("make_lock failed");
    return mutex;
}
```

返回值是个指针, 你可以将其作为参数传递, 而不会引起不必要的复制.

锁定和解锁互斥锁的函数是对 POSIX 函数的简单封装:

```
void mutex_lock(Mutex *mutex)
{
    int n = pthread_mutex_lock(mutex);
    if (n != 0) perror_exit("lock failed");
}

void mutex_unlock(Mutex *mutex)
{
    int n = pthread_mutex_unlock(mutex);
    if (n != 0) perror_exit("unlock failed");
}
```

这些代码在 mutex.c 和同文件 mutex.h 中.

Chapter 10

条件变量

许多简单的同步问题都可以使用上一章讲述的互斥锁来解决. 在本章我将介绍一个更大的挑战, 即著名的 “生产者-消费者问题”, 以及解决它的新工具: 条件变量.

10.1 工作对列

在一些多线程问题中, 线程被用来执行不同的任务. 他们通常使用队列进行相互通信, 其中一些线程叫做 “生产者”, 会将数据放入对列中, 而其他线程被称为 “消费者”, 从队列中取出数据.

例如, 在带有用户交互界面的应用中, 可能有一个线程运行 GUI, 以响应用户事件, 另一个线程处理用户请求. 这种情况下, GUI 线程会将请求放入队列, 而 “后端” 线程则从队列中取出请求并进行处理.

为了支持这种组织方式, 我们需要一个 “线程安全” 的队列实现, 也就是两个线程 (或多于两个) 可以同时访问队列. 此外, 我们还需要能处理特殊情况, 队列为空时, 以及队列大小有限制, 而队列已满时.

我将从一个不是线程安全的简单队列开始, 然后我们会看到它哪里出错并修复它. 该示例的代码位于本书的代码库中, 名为 `queue` 的文件夹. 文件 `queue.c` 包含一个环形缓冲区的基本实现, 你可以阅读这里`https://en.wikipedia.org/wiki/Circular_buffer`了解更多.

以下是结构定义:

```
typedef struct {
    int *array;
    int length;
```

```
    int next_in;
    int next_out;
} Queue;
```

array 是包含队列元素的数组. 在这个例子中, 元素是整数, 但通常情况下, 它们可能是包含用户事件, 工作项等的结构体.

length 是数组的长度. next_in 是数组的索引, 指示下一个元素应该添加的位置; 类似的, next_out 是下一个应该移除元素的索引.

make_queue 为这个结构体分配了空间, 并初始化了各个字段:

```
Queue *make_queue(int length)
{
    Queue *queue = (Queue *) malloc(sizeof(Queue));
    queue->length = length + 1;
    queue->array = (int *) malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    return queue;
}
```

next_out的初始值需要解释一下. 由于队列初始为空, 没有下一个元素可以移除, 所以 next_out无效. 令next_out == next_in 设置为特殊情况, 表示队列为空, 所以我们可以写成:

```
int queue_empty(Queue *queue)
{
    return (queue->next_in == queue->next_out);
}
```

现在我们可以使用queue_push, 向队列添加元素:

```
void queue_push(Queue *queue, int item) {
    if (queue_full(queue)) {
        perror_exit("queue is full");
    }

    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);
}
```

如果队列满了, queue_push 会打印一条错误信息并退出. 我稍后会解释 queue_full.

如果队列未满, queue_push 会插入新元素, 然后使用 queue_incr 增加 next_in:

```
int queue_incr(Queue *queue, int i)
{
    return (i+1) % queue->length;
}
```

当索引 i 到达数组结尾, 它会绕回到 0. 这才是我们遇到的棘手之处. 如果我们依然向队列中添加元素, 最后 next_in 会绕回去追上 next_out. 但如果 next_in == next_out, 我们又会错误认为队列为空.

为了避免此情况, 我们定义其他方案来表示队列已满:

```
int queue_full(Queue *queue)
{
    return (queue_incr(queue, queue->next_in) == queue->next_out);
}
```

如果递增 next_in 会导致与 next_out 重合, 意味着我们无法添加元素, 否则队列便看起来空了. 所以我们在 "末尾" 的前一元素的位置停止插入 (注意队列的末尾可以在任何位置, 不一定是数组的末尾).

现在我们可以编写queue_pop函数, 它会从队列移除并返回下一个元素:

```
int queue_pop(Queue *queue) {
    if (queue_empty(queue)) {
        perror_exit("queue is empty");
    }

    int item = queue->array[queue->next_out];
    queue->next_out = queue_incr(queue, queue->next_out);
    return item;
}
```

如果你尝试从空队列中弹出元素, queue_pop会打印错误信息并退出.

10.2 生产者与消费者

现在让我们创造一些线程来访问队列. 下面是生产者代码:

```
void *producer_entry(void *arg) {
    Shared *shared = (Shared *) arg;

    for (int i=0; i<QUEUE_LENGTH-1; i++) {
        printf("adding item %d\n", i);
        queue_push(shared->queue, i);
    }
    pthread_exit(NULL);
}
```

下面是消费者代码:

```
void *consumer_entry(void *arg) {
    int item;
    Shared *shared = (Shared *) arg;

    for (int i=0; i<QUEUE_LENGTH-1; i++) {
        item = queue_pop(shared->queue);
        printf("consuming item %d\n", item);
    }
    pthread_exit(NULL);
}
```

这是启动线程并等待它们的父线程代码:

```
    pthread_t child[NUM_CHILDREN];

    Shared *shared = make_shared();

    child[0] = make_thread(producer_entry, shared);
    child[1] = make_thread(consumer_entry, shared);

    for (int i=0; i<NUM_CHILDREN; i++) {
        join_thread(child[i]);
    }
```

最后这是包含队列的共享结构:

```
typedef struct {
    Queue *queue;
} Shared;

Shared *make_shared()
{
    Shared *shared = check_malloc(sizeof(Shared));
    shared->queue = make_queue(QUEUE_LENGTH);
    return shared;
}
```

到目前为止, 我们的代码是个好的开始, 但仍然有几个问题:

- 访问队列不是线程安全的. 不同的线程可以同时访问 `array`, `next_in`, 和 `next_out`, 从而使队列处于破损的, "不一致" 的状态.
- 如果消费者先被调度, 它发现队列未空, 便打印错误信息并退出. 我们更希望消费者一直堵塞到队列不为空. 同样的, 我们也希望生产者在队列满的时候进行堵塞等待.

下一节, 我们使用 Mutex 来解决第一个问题. 后续章节, 我们通过条件变量解决第二个问题.

10.3 互斥

我们可以使用互斥锁（mutex）来保证线程安全。该版本的代码位于 queue_mutex.c 中。

首先，我们在队列结构中添加一个 Mutex 指针:

```
typedef struct {
    int *array;
    int length;
    int next_in;
    int next_out;
    Mutex *mutex;          //-- this line is new
} Queue;
```

同时在make_queue中初始化 Mutex:

```
Queue *make_queue(int length) {
    Queue *queue = (Queue *) malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *) malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_mutex();   //-- new
    return queue;
}
```

然后我们在 queue_push中添加同步代码:

```
void queue_push(Queue *queue, int item) {
    mutex_lock(queue->mutex);   //-- new
    if (queue_full(queue)) {
      mutex_unlock(queue->mutex);   //-- new
      perror_exit("queue is full");
    }

    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);
    mutex_unlock(queue->mutex);   //-- new
}
```

检查队列是否已满之前，我们需要先锁定 Mutex。如果队列已满，则在退出之前必须先解锁 Mutex；否则，该线程将保持锁定状态，其他线程将无法继续执行。

queue_pop 的同步代码与此类似：

```
int queue_pop(Queue *queue) {
    mutex_lock(queue->mutex);
    if (queue_empty(queue)) {
      mutex_unlock(queue->mutex);
      perror_exit("queue is empty");
    }

    int item = queue->array[queue->next_out];
    queue->next_out = queue_incr(queue, queue->next_out);
    mutex_unlock(queue->mutex);
    return item;
}
```

注意，其他 Queue 相关函数——包括 queue_full、queue_empty 和 queue_incr——都不会尝试去锁定互斥锁。任何调用这些函数的线程都需要先手动加锁；这是这些函数接口文档中的一部分约定。

通过添加这些代码，现在队列已经是线程安全的；如果你运行它，应该不会再看到任何同步错误。但是，很可能在某个时刻消费者会因为队列为空而退出，或者生产者因为队列已满而退出，或者两者都会发生。

下一步，我们将引入条件变量。

10.4 条件变量

条件变量是一种与“条件”相关联的数据结构；它允许线程在条件变为真之前一直阻塞等待。例如，thread_pop 可能想先检查队列是否为空，如果为空，就等待“队列非空”这一条件。

类似地，thread_push 可能需要检查队列是否已满，如果已满，就阻塞直到队列不再是满的。

这里我先处理第一个条件，而第二个条件将作为练习留给你完成。

首先，我们在 Queue 结构中添加一个条件变量：

```
typedef struct {
    int *array;
    int length;
```

```
    int next_in;
    int next_out;
    Mutex *mutex;
    Cond *nonempty;   //-- new
} Queue;
```

并在 make_queue 中对其进行初始化：

```
Queue *make_queue(int length)
{
    Queue *queue = (Queue *) malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *) malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_mutex();
    queue->nonempty = make_cond();   //-- new
    return queue;
}
```

现在，在 queue_pop 中，如果发现队列为空，我们不再直接退出；而是使用条件变量进行阻塞等待：

```
int queue_pop(Queue *queue) {
  mutex_lock(queue->mutex);
  while (queue_empty(queue)) {
    cond_wait(queue->nonempty, queue->mutex);  //-- new
  }

  int item = queue->array[queue->next_out];
  queue->next_out = queue_incr(queue, queue->next_out);
  mutex_unlock(queue->mutex);
  cond_signal(queue->nonfull);   //-- new
  return item;
}
```

cond_wait 的行为比较复杂，因此我们一步一步来看。第一个参数是条件变量，在这里我们等待的条件是“队列非空”。第二个参数是用于保护队列的互斥锁。

当已经锁定互斥锁的线程调用 cond_wait 时，它会先解锁互斥锁，然后进入阻塞状态。这一点非常重要。如果 cond_wait 在阻塞前没有释放互斥锁，那么其他线程将无法访问队列，也就无法继续添加新元素，队列将永远保持为空。

因此，当消费者线程阻塞在 nonempty 上时，生产者线程是可以继续运行的。来看一下生产者运行 queue_push 时发生了什么：

```
void queue_push(Queue *queue, int item) {
    mutex_lock(queue->mutex);
    if (queue_full(queue)) {
        mutex_unlock(queue->mutex);
        perror_exit("queue is full");
    }
    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);
    mutex_unlock(queue->mutex);
    cond_signal(queue->nonempty);    //-- new
}
```

与之前一样，queue_push 会先锁定 Mutex，并检查队列是否已满。在假设队列未满的情况下，它会向队列中添加一个新元素，然后解锁 Mutex。

但在返回之前，它还会多做一件事：对条件变量 nonempty 进行一次“信号通知（signal）”。

对条件变量发出信号，通常意味着对应的条件已经为真。如果此时没有线程在该条件变量上等待，那么该信号不会产生任何影响。

如果有线程正在该条件变量上等待，则其中一个线程会被唤醒，并继续执行 cond_wait 之后的代码。但是，在被唤醒的线程真正从 cond_wait 返回之前，它必须重新获取并锁定 Mutex。

现在回到 queue_pop，看看线程从 cond_wait 返回时会发生什么。它会回到 while 循环的开头，再次检查条件。稍后我会解释原因，但现在先假设条件为真，也就是说队列中已经有元素。

当消费者线程跳出 while 循环时，我们可以确定两点：(1) 条件为真，即队列中至少有一个元素；(2) Mutex 已经被锁定，因此可以安全地访问队列。

在取出一个元素之后，queue_pop 会解锁互斥锁并返回。

在下一节中，我会展示 Cond 的具体实现方式，但在此之前，我想先回答两个常见问题：

- 为什么 cond_wait 要放在 while 循环中，而不是 if 语句中？也就是说，为什么从 cond_wait 返回后还需要再次检查条件？

  主要原因在于“信号被截获”的可能性。假设线程 A 正在 nonempty 上等待。线程 B 向队列中添加了一个元素，并对 nonempty 发出了信号。线程 A 被唤醒并试图获取互斥锁，但在它成功获取之前，邪恶的线程 C 抢先一步，锁定互斥锁，从队列中取走了这个元素，然后解锁。此时队列再次变为空，但线程 A 已经不再处于阻塞状态。线程 A 随后获取互斥锁，并从 cond_wait 返回。如果线程 A 不再次检查条件，它就会从一个空队列中取数据，从而很可能导致错误。

- 另一个常见问题是:“条件变量如何知道它对应的具体条件是什么?”

  这是可以理解的，因为在 Cond 结构体与其所关联的条件之间，并不存在显式的绑定关系。它们之间的关联是通过使用方式隐式建立的。

  可以这样理解：当你调用 cond_wait 时，条件是“假”的；当你调用 cond_signal 时，条件是“真”的。

由于线程在从 cond_wait 返回后都会重新检查条件，因此实际上并不要求只在条件一定为真时才调用 cond_signal。只要你有理由认为条件“可能”为真，就可以调用 cond_signal 作为一种提示，提示现在是检查条件的合适时机。

10.5 条件变量实现

前一节中使用的 Cond 结构，是对 POSIX 线程 API 中 pthread_cond_t 类型的一个封装。这与 Mutex 非常相似，Mutex 是 pthread_mutex_t 的封装。这两个封装都定义在 utils.c 和 utils.h 中。

其类型定义如下:

```
typedef pthread_cond_t Cond;
```

make_cond 负责分配空间、初始化条件变量，并返回一个指针:

```
Cond *make_cond() {
    Cond *cond = check_malloc(sizeof(Cond));
    int n = pthread_cond_init(cond, NULL);
    if (n != 0) perror_exit("make_cond failed");

    return cond;
}
```

下面是 cond_wait 和 cond_signal 的封装实现:

```
void cond_wait(Cond *cond, Mutex *mutex) {
    int n = pthread_cond_wait(cond, mutex);
    if (n != 0) perror_exit("cond_wait failed");
}

void cond_signal(Cond *cond) {
    int n = pthread_cond_signal(cond);
    if (n != 0) perror_exit("cond_signal failed");
}
```

到这里为止，这些代码应该都不会让你感到意外。

Chapter 11

C 中的信号量

Semaphores are a good way to learn about synchronization, but they are not as widely used, in practice, as mutexes and condition variables.

Nevertheless, there are some synchronization problems that can be solved simply with semaphores, yielding solutions that are more demonstrably correct.

This chapter presents a C API for working with semaphores and my code for making it easier to work with. And it presents a final challenge: can you write an implementation of a semaphore using mutexes and condition variables?

The code for this chapter is in directory `semaphore` in the repository for this book (see Section 0.1).

11.1 POSIX Semaphores

A semaphore is a data structure used to help threads work together without interfering with each other.

The POSIX standard specifies an interface for semaphores; it is not part of Pthreads, but most UNIXes that implement Pthreads also provide semaphores.

POSIX semaphores have type `sem_t`. As usual, I put a wrapper around `sem_t` to make it easier to use. The interface is defined in `sem.h`:

```
typedef sem_t Semaphore;

Semaphore *make_semaphore(int value);
void semaphore_wait(Semaphore *sem);
void semaphore_signal(Semaphore *sem);
```

`Semaphore` is a synonym for `sem_t`, but I find it more readable, and the capital letter reminds me to treat it like an object and pass it by pointer.

The implementation of these functions is in `sem.c`:

```
Semaphore *make_semaphore(int value)
{
    Semaphore *sem = check_malloc(sizeof(Semaphore));
    int n = sem_init(sem, 0, value);
    if (n != 0) perror_exit("sem_init failed");
    return sem;
}
```

`make_semaphore` takes the initial value of the semaphore as a parameter. It allocates space for a Semaphore, initializes it, and returns a pointer to `Semaphore`.

`sem_init` returns 0 if it succeeds and -1 if anything goes wrong. One nice thing about using wrapper functions is that you can encapsulate the error-checking code, which makes the code that uses these functions more readable.

Here is the implementation of `semaphore_wait`:

```
void semaphore_wait(Semaphore *sem)
{
    int n = sem_wait(sem);
    if (n != 0) perror_exit("sem_wait failed");
}
```

And here is `semaphore_signal`:

```
void semaphore_signal(Semaphore *sem)
{
    int n = sem_post(sem);
    if (n != 0) perror_exit("sem_post failed");
}
```

I prefer to call this operation “signal” rather than “post”, although both terms are common.

Here’s an example that shows how to use a semaphore as a mutex:

```
Semaphore *mutex = make_semaphore(1);

semaphore_wait(mutex);
  // protected code goes here
semaphore_signal(mutex);
```

When you use a semaphore as a mutex, you usually initialize it to 1 to indicate that the mutex is unlocked; that is, one thread can pass the semaphore without blocking.

Here I am using the variable name `mutex` to indicate that the semaphore is being used as a mutex. But remember that the behavior of a semaphore is not the same as a Pthread mutex.

11.2 Producers and consumers with semaphores

Using these semaphore wrapper functions, we can write a solution to the Producer-Consumer problem from Section 10.2. The code in this section is in `queue_sem.c`.

Here's the new definition of `Queue`, replacing the mutex and condition variables with semaphores:

```
typedef struct {
    int *array;
    int length;
    int next_in;
    int next_out;
    Semaphore *mutex;       //-- new
    Semaphore *items;       //-- new
    Semaphore *spaces;      //-- new
} Queue;
```

And here's the new version of `make_queue`:

```
Queue *make_queue(int length)
{
    Queue *queue = (Queue *) malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *) malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_semaphore(1);
    queue->items = make_semaphore(0);
    queue->spaces = make_semaphore(length-1);
    return queue;
}
```

`mutex` is used to guarantee exclusive access to the queue; the initial value is 1, so the mutex is initially unlocked.

`items` is the number of items in the queue, which is also the number of consumer threads that can execute `queue_pop` without blocking. Initially there are no items in the queue.

`spaces` is the number of empty spaces in the queue, which is the number of producer threads that can execute `queue_push` without blocking. Initially the number of spaces is the capacity of the queue, which is `length-1`, as explained in Section 10.1.

Here is the new version of `queue_push`, which is run by producer threads:

```
void queue_push(Queue *queue, int item) {
  semaphore_wait(queue->spaces);
  semaphore_wait(queue->mutex);

  queue->array[queue->next_in] = item;
  queue->next_in = queue_incr(queue, queue->next_in);

  semaphore_signal(queue->mutex);
  semaphore_signal(queue->items);
}
```

Notice that `queue_push` doesn't have to call `queue_full` any more; instead, the semaphore keeps track of how many spaces are available and blocks producers if the queue is full.

Here is the new version of `queue_pop`:

```
int queue_pop(Queue *queue) {
  semaphore_wait(queue->items);
  semaphore_wait(queue->mutex);

  int item = queue->array[queue->next_out];
  queue->next_out = queue_incr(queue, queue->next_out);

  semaphore_signal(queue->mutex);
  semaphore_signal(queue->spaces);

  return item;
}
```

This solution is explained, using pseudo-code, in Chapter 4 of *The Little Book of Semaphores*.

Using the code in the repository for this book, you should be able to compile and run this solution like this:

```
$ make queue_sem
$ ./queue_sem
```

11.3 Make your own semaphores

Any problem that can be solved with semaphores can also be solved with condition variables and mutexes. We can prove that's true by using condition variables and mutexes to implement a semaphore.

Before you go on, you might want to try this as an exercise: write functions that implement the semaphore API in `sem.h` using using condition variables and mutexes. In the repository for this book, you'll find my solution in `mysem_soln.c` and `mysem_soln.h`.

If you have trouble getting started, you can use the following structure definition, from my solution, as a hint:

```
typedef struct {
  int value, wakeups;
  Mutex *mutex;
  Cond *cond;
} Semaphore;
```

`value` is the value of the semaphore. `wakeups` counts the number of pending signals; that is, the number of threads that have been woken but have not yet resumed execution. The reason for wakeups is to make sure that our semaphores have Property 3, described in `The Little Book of Semaphores`.

`mutex` provides exclusive access to `value` and `wakeups`; `cond` is the condition variable threads wait on if they wait on the semaphore.

Here is the initialization code for this structure:

```
Semaphore *make_semaphore(int value)
{
  Semaphore *semaphore = check_malloc(sizeof(Semaphore));
  semaphore->value = value;
  semaphore->wakeups = 0;
  semaphore->mutex = make_mutex();
  semaphore->cond = make_cond();
  return semaphore;
}
```

11.3.1 Semaphore implementation

Here is my implementation of semaphores using POSIX mutexes and condition variables:

```
void semaphore_wait(Semaphore *semaphore)
{
  mutex_lock(semaphore->mutex);
  semaphore->value--;

  if (semaphore->value < 0) {
    do {
      cond_wait(semaphore->cond, semaphore->mutex);
    } while (semaphore->wakeups < 1);
    semaphore->wakeups--;
  }
  mutex_unlock(semaphore->mutex);
}
```

When a thread waits on the semaphore, it has to lock the mutex before it decrements `value`. If the value of the semaphore becomes negative, the thread blocks until a "wakeup" is available. While it is blocked, the mutex is unlocked, so another thread can signal.

Here is the code for `semaphore_signal`:

```
void semaphore_signal(Semaphore *semaphore)
{
  mutex_lock(semaphore->mutex);
  semaphore->value++;

  if (semaphore->value <= 0) {
    semaphore->wakeups++;
    cond_signal(semaphore->cond);
  }
  mutex_unlock(semaphore->mutex);
}
```

Again, a thread has to lock the mutex before it increments `value`. If the semaphore was negative, that means threads are waiting, so the signalling thread increments `wakeups` and signals the condition variable.

At this point one of the waiting threads might wake up, but the mutex is still locked until the signalling thread unlocks it.

At that point, one of the waiting threads returns from `cond_wait` and checks

whether a wakeup is still available. If not, it loops and waits on the condition variable again. If so, it decrements `wakeups`, unlocks the mutex, and exits.

One thing about this solution that might not be obvious is the use of a `do...while` loop. Can you figure out why it is not a more conventional `while` loop? What would go wrong?

The problem is that with a `while` loop this implementation would not have Property 3. It would be possible for a thread to signal and then run around and catch its own signal.

With the `do...while` loop, it is guaranteed[1] that when a thread signals, one of the waiting threads will get the signal, even if the signalling thread runs around and gets the mutex before one of the waiting threads resumes.

[1]Well, almost. It turns out that a well-timed spurious wakeup (see `http://en.wikipedia.org/wiki/Spurious_wakeup`) can violate this guarantee.