据，它会**阻塞**。这意味着，该进程会停止运行，操作系统可以运行其他进程。但是，许多程序能够继续工作即使当前运行的部分必须等待某些资源。例如：航班预定系统在一个用户因为等待座位图而阻塞时，可为另一个用户提供可用的航线查询。**线程**为程序员提供了一种机制，将程序划分为多个大致独立的任务，当某个任务阻塞时能执行其他任务。此外，在大多数系统中，线程间的切换比进程间的切换更快。因为线程相对于进程而言是“轻量级”的。线程包含在进程中，所以线程可以使用相同的可执行代码，共享相同的内存和相同的 I/O 设备。实际上，当两个线程共属于一个进程时，它们共享进程的大多数资源。它们之间最大的差别是各自需要一个私有的程序计数器和函数调用栈，使它们能够独立运行。

如果进程是执行的“主线程”，其他线程由主线程启动和停止，那么我们可以设想进程和它的子线程如下进行，当一个线程开始时，它从进程中**派生**（fork）出来；当一个线程结束，它**合并**（join）到进程中，如图 2-2 所示。

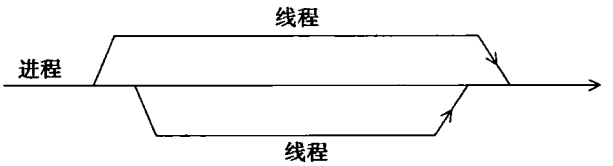

图 2-2　一个进程与两个线程

2.2　对冯·诺依曼模型的改进

正如我们前面提到的，第一台电子数字计算机是在 20 世纪 40 年代诞生的，计算机科学家和计算机工程师对基本的冯·诺依曼结构已经做了很多改进。许多改进都是为了解决冯·诺依曼瓶颈，也有许多改进只是为了使 CPU 更快。本节中，我们会看到三种改进措施：缓存（caching）、
[18] 虚拟存储器（或虚拟内存）、低层次并行。

2.2.1　Cache 基础知识

缓存是解决冯·诺依曼瓶颈而最广泛使用的方法之一。为了理解缓存背后的思想，我们再来回忆前面的例子。一个大型企业在某个镇上有一个工厂（CPU），在另一个镇上有一个仓库（主存）。在工厂和仓库之间只有一条双车道公路。有许多方法可以解决工厂和仓库之间的原材料和制成品运输难题。其中之一是加宽公路，另一个方法是迁移工厂或者仓库，或者建立一个统一的工厂和仓库。缓存结合了这两种方法。不再是一次传输一条指令或者一个数据，而是将互连通路加宽，使得一次内存访问能够传送更多的数据或者更多的指令。而且，不再是将所有的数据和指令存储在主存中，可以将部分数据块或者代码块存储在一个靠近 CPU 寄存器的特殊存储器里。

一般来说，对**高速缓冲存储器**（cache，简称缓存）的访问时间比其他存储区域的访问时间短。在本书中，当谈到缓存时，一般指的是 **CPU 缓存**（CPU Cache）。CPU Cache 是一组相比于主存，CPU 能更快速地访问的内存区域。CPU Cache 位于与 CPU 同一块的芯片或者位于其他芯片上，但比普通的内存芯片能更快地访问。

有了 Cache 后，一个很明显的问题是什么样的数据和指令能够存储在 Cache 中。通用的准则基于下面的原理：程序接下来可能会用到的指令和数据与最近访问过的指令和数据在物理上是邻近存放的。在执行完一条指令后，程序通常会执行下一条指令。同样，当程序访问一个内存区域后，通常会访问物理位置靠近的下一个区域。一个例子是在使用数组时，考虑下面的循环：

```
float z[1000];
. . .
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

数组在内存中是连续分配的，所以存储 z[1] 数据的内存区域紧接在存储 z[0] 的内存区域后

面。因此，只要 i <999，在读完 z[i] 的数据之后总是立即读 z[i +1] 的数据。

程序访问完一个存储区域往往会访问接下来的区域，这个原理称为**局部性**。在访问完一个内存区域（指令或者数据），程序会在不久的将来（**时间**局部性）访问邻近的区域（**空间**局部性）。

为了运用局部性原理，系统使用更宽的互连结构来访问数据和指令。也就是：一次内存访问能存取一整块代码和数据，而不只是单条指令和单条数据。这些块称为**高速缓存块**或者**高速缓存行**。一个典型的高速缓存行能存储 8 ~ 16 倍单个内存区域的信息。在我们的例子中，如果一个高 [19] 速缓存行可以存放 16 个浮点数，当执行 sum + =z[0] 时，系统可能把数组 z 最开始的 16 个元素：z[0]、z[1]、…、z[15] 从主存读到 Cache 中。因此，在后面的 15 次加法运算中，需要使用的数据已经在 Cache 中。

从概念上，很容易把 CPU Cache 认为是单一结构，但实际上，Cache 分为不同的**层**（level）。第一层（L1）最小但最快，更高层 Cache（L2、L3、…）更大但相对较慢。2010 年，大多数系统拥有至少两层 Cache，有三层 Cache 是非常普遍的。Cache 通常是用来存储速度较慢的存储器中信息的副本，可以认为低层 Cache（更快、更小）是高层 Cache 的 Cache。所以，一个变量存储在 L1 Cache 中，也会存储在 L2 Cache 中。但是，有些多层 Cache 不会复制已经在其他层 Cache 中存在的信息。对于这种 Cache，L1 Cache 中的变量不会存储在其他层 Cache 中，但会存储在主存中。

当 CPU 需要访问指令或者数据时，它会沿着 Cache 的层次结构向下查询：首先查询 L1 Cache，接着 L2 Cache，以此类推。最后，如果 Cache 中没有所需要的信息，就会访问主存。当向 Cache 查询信息时，如果 Cache 中有信息，则称为 **Cache 命中**或者**命中**；如果信息不存在，则称为 **Cache 缺失**或者**缺失**。命中和缺失是相对 Cache 层而言的。例如，当 CPU 试图访问某个变量时，很可能 L1 Cache 缺失，而 L2 Cache 命中。

注意，存储器访问的术语**读**（read）和**写**（write）也适用于 Cache，例如我们可以从 L2 Cache 中读一条指令，也可以向 L1 Cache 写数据。

当 CPU 尝试读数据或者指令时，如果发生 Cache 缺失，那么就会从主存中读出包含所需信息的整个高速缓存块。这时 CPU 会阻塞，因为它需要等待速度相对较慢的主存：处理器可以停止执行当前程序的指令，直到从主存中取出所需的数据或者指令。例如，当我们读 z[0] 时，处理器会阻塞直到包括 z[0] 的高速缓存块从主存传送到 Cache 中。

当 CPU 向 Cache 中写数据时，Cache 中的值与主存中的值就会不同或者**不一致**（inconsistent）。有两种方法来解决这个不一致性问题。在**写直达**（write-through）Cache 中，当 CPU 向 Cache 写数据时，高速缓存行会立即写入主存中。在**写回**（write-back）Cache 中，数据不是立即更新到主存中，而是将发生数据更新的高速缓存行标记成脏（dirty）。当发生高速缓存行替换时，标记为脏的高速缓存行被写入主存中。

2.2.2　Cache 映射

在 Cache 设计中，另一个问题是高速缓存行应该存储在什么位置。当从主存中取出一个高速缓存行时，应该把这个高速缓存行放置到 Cache 中的什么位置？这个问题的答案因系统而异。一个极端是**全相联**（fully associative）Cache，每个高速缓存行能够放置在 Cache 中的任意位置。另 [20] 一个极端是**直接映射**（directed mapped）Cache，每个高速缓存行在 Cache 中有唯一的位置。处于两种极端中间的方案是 ***n* 路组相联**（n-way set associated）。在 *n* 路组相联 Cache 中，每个高速缓存行都能放置到 Cache 中 *n* 个不同区域位置中的一个。例如，在 2 路组相联 Cache 中，每个高速缓存行可以映射到 2 个位置中的一个。

假设主存有 16 行，分别用 0 ~ 15 标记，Cache 有 4 行，用 0 ~ 3 标记。在全相联映射中，主存中的 0 号行能够映射到 Cache 中的 0、1、2、3 任意一行。在直接映射 Cache 中，可以根据主存中

高速缓存行的标记值除以 4 求余，获得在 Cache 中的索引。因此主存中 0、4、8 号行会映射到 Cache 的 0 号行，主存中的 1、5、9 号行映射到 Cache 的 1 号行，以此类推。在 2 路组相联 Cache 中，将 Cache 分成两组，0 号行和 1 号行构成一个组，称为 0 号组；2 号行和 3 号行构成另一个组，称为 1 号组。根据主存中行的标记对 2 取模从而获得 Cache 中组的索引号。主存的 0 号行可以映射到 Cache 中第 0 组中的 0 号行或者 1 号行。详见表 2-1。

表 2-1 将 16 行的主存映射到 4 行的 Cache 上

内存索引	高速缓存中的位置		
	全相联	直接映射	2 路组相联
0	0，1，2 或 3	0	0 或 1
1	0，1，2 或 3	1	2 或 3
2	0，1，2 或 3	2	0 或 1
3	0，1，2 或 3	3	2 或 3
4	0，1，2 或 3	0	0 或 1
5	0，1，2 或 3	1	2 或 3
6	0，1，2 或 3	2	0 或 1
7	0，1，2 或 3	3	2 或 3
8	0，1，2 或 3	0	0 或 1
9	0，1，2 或 3	1	2 或 3
10	0，1，2 或 3	2	0 或 1
11	0，1，2 或 3	3	2 或 3
12	0，1，2 或 3	0	0 或 1
13	0，1，2 或 3	1	2 或 3
14	0，1，2 或 3	2	0 或 1
15	0，1，2 或 3	3	2 或 3

当内存中的行（多于一行）能被映射到 Cache 中的多个不同位置（全相联和 n 路组相联）时，需要决定替换或者**驱逐** Cache 中的哪一行。在前面的 2 路组相联的例子中，假设主存中的 0 号行已存储在 Cache 的 0 号行，主存中的 2 号行已存储在 Cache 的 1 号行，那么主存的 4 号行
[21] 应该存储在哪里呢？最常用的替换方案是**最近最少使用**（least recently used）。顾名思义，Cache 记录各个块被访问的次数，替换最近访问次数最少的块。如果近来主存的 0 号行比 2 号行访问的更多，那么 2 号行在就会被替换出 Cache，它在原来 Cache 的位置就被替换成用来存储 4 号主存行。

2.2.3 Cache 和程序：一个实例

非常重要的一点是：CPU Cache 是由系统硬件来控制的，而编程人员并不能直接决定什么数据和什么指令应该在 Cache 中。但是，了解空间局部性和时间局部性原理可以让我们对 Cache 有些许间接的控制。例如，C 语言以“行主序”来存储二维数组。尽管二维数组看上去是一个矩形块，但是内存是巨大的一维数组。在行主序存储模式下，先存储二维数组的第一行，接着第二行，以次类推。下面的两段代码中，第一个嵌套循环比第二个嵌套循环有更好的性能，因为它顺序访问二维数组中的数据。

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

为了更好地理解，假设 MAX 等于 4，那么数组 A 中元素的存储位置为：

高速缓存行	数组 A 中的元素			
0	A [0] [0]	A [0] [1]	A [0] [2]	A [0] [3]
1	A [1] [0]	A [1] [1]	A [1] [2]	A [1] [3]
2	A [2] [0]	A [2] [1]	A [2] [2]	A [2] [3]
3	A [3] [0]	A [3] [1]	A [3] [2]	A [3] [3]

所以，A[0][1]存储在 A[0][0]的后面，而 A[1][0]存储在 A[0][3]的后面。

假设刚开始时，Cache 中没有 A 数组的任何元素，一个高速缓存行可以存放 A 的 4 个元素，并且 A[0][0]是高速缓存行中的第一个元素。最后，假设 Cache 是直接映射的，只能存储 A 数组的 8 个元素，即两个高速缓存行（我们不关注 x 和 y）。 22

两个循环都尝试首先访问 A[0][0]，因为它不在 Cache 中，所以这将导致一次 Cache 缺失，然后系统将包含 A[0][0]、A[0][1]、A[0][2]、A[0][3]的行从内存中读出并写入 Cache 中。第一个循环接下来会依次访问 A[0][1]、A[0][2]、A[0][3]，它们都在 Cache 中，而下一次 Cache 缺失就会发生在代码访问 A[1][0]的时候。按照上面的分析，我们可以看到第一个循环在访问数组 A 的元素时，总共发生 4 次 Cache 缺失，每行发生一次。值得注意的是，我们假想的 Cache 只能存储两个高速缓存行，即 8 个元素，当读第三行的第一个元素和第四行的第一个元素时，Cache 中已存在的行必须被替换出去。一旦某行被替换出去，第一个循环就不会再次访问该行的元素。

在将第一行元素读出并写入 Cache 中后，第二个循环需要访问 A[1][0]、A[2][0]、A[3][0]，但是它们都不在 Cache 中。所以下面三次访问 A 都将导致 Cache 缺失。此外，因为 Cache 比较小，读 A[2][0]和 A[3][0]会导致 Cache 中原有的行被替换出去。因为 A[2][0]在 2 号行中，读该行会导致 0 号高速缓存行被替换出去，而读 A[3][0]会导致 1 号高速缓存行被替换出去。在执行完一次内部循环后，要访问 A[0][1]，但它原来所在的 0 号高速缓存行已经被替换出去了。所以我们可以看到，每次读 A 的元素，就会发生一次 Cache 缺失，所以第二个循环总共发生 16 次缺失。

因此，我们可以预测第一个嵌套循环比第二个运行速度快。在实际运行的时候，当 MAX = 1000，第一个嵌套循环的运行速度近 3 倍快于第二个嵌套循环。

2.2.4　虚拟存储器

Cache 使得 CPU 快速访问主存中的指令和数据变成可能。但是，如果运行一个大型的程序，或者程序需要访问大型数据集，那么所有的指令或者数据可能在主存中放不下。这种情况在多任