这段代码运行时，不是一定会出现死锁的。当线程 1 占有锁 L1，上下文切换到线程 2。线程 2 锁住 L2，试图锁住 L1。这时才产生了死锁，两个线程互相等待。如图 32.1 所示，其中的圈（cycle）表明了死锁。

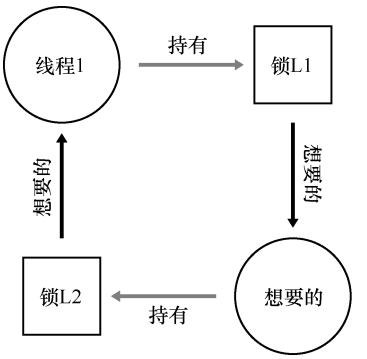

图 32.1 死锁依赖图

该图应该有助于描述清楚问题。程序员在编写代码中应该如何处理死锁呢？

关键问题：如何对付死锁

我们在实现系统时，如何避免或者能够检测、恢复死锁呢？这是目前系统中的真实问题吗？

为什么发生死锁

你可能在想，上文提到的这个死锁的例子，很容易就可以避免。例如，只要线程 1 和线程 2 都用相同的抢锁顺序，死锁就不会发生。那么，死锁为什么还会发生？

其中一个原因是在大型的代码库里，组件之间会有复杂的依赖。以操作系统为例。虚拟内存系统在需要访问文件系统才能从磁盘读到内存页；文件系统随后又要和虚拟内存交互，去申请一页内存，以便存放读到的块。因此，在设计大型系统的锁机制时，你必须要仔细地去避免循环依赖导致的死锁。

另一个原因是封装（encapsulation）。软件开发者一直倾向于隐藏实现细节，以模块化的方式让软件开发更容易。然而，模块化和锁不是很契合。Jula 等人指出[J+08]，某些看起来没有关系的接口可能会导致死锁。以 Java 的 Vector 类和 AddAll()方法为例，我们这样调用这个方法：

```
Vector v1, v2;
v1.AddAll(v2);
```

在内部，这个方法需要多线程安全，因此针对被添加向量（v1）和参数（v2）的锁都需要获取。假设这个方法，先给 v1 加锁，然后再给 v2 加锁。如果另外某个线程几乎同时在调用 v2.AddAll(v1)，就可能遇到死锁。

产生死锁的条件

死锁的产生需要如下 4 个条件[C+71]。

- 互斥：线程对于需要的资源进行互斥的访问（例如一个线程抢到锁）。
- 持有并等待：线程持有了资源（例如已将持有的锁），同时又在等待其他资源（例如，需要获得的锁）。
- 非抢占：线程获得的资源（例如锁），不能被抢占。
- 循环等待：线程之间存在一个环路，环路上每个线程都额外持有一个资源，而这个资源又是下一个线程要申请的。

如果这 4 个条件的任何一个没有满足，死锁就不会产生。因此，我们首先研究一下预防死锁的方法；每个策略都设法阻止某一个条件，从而解决死锁的问题。

预防

循环等待

也许最实用的预防技术（当然也是经常采用的），就是让代码不会产生循环等待。最直接的方法就是获取锁时提供一个全序（total ordering）。假如系统共有两个锁（L1 和 L2），那么我们每次都先申请 L1 然后申请 L2，就可以避免死锁。这样严格的顺序避免了循环等待，也就不会产生死锁。

当然，更复杂的系统中不会只有两个锁，锁的全序可能很难做到。因此，偏序（partial ordering）可能是一种有用的方法，安排锁的获取并避免死锁。Linux 中的内存映射代码就是一个偏序锁的好例子[T+94]。代码开头的注释表明了 10 组不同的加锁顺序，包括简单的关系，比如 i_mutex 早于 i_mmap_mutex，也包括复杂的关系，比如 i_mmap_mutex 早于 private_lock，早于 swap_lock，早于 mapping->tree_lock。

你可以想到，全序和偏序都需要细致的锁策略的设计和实现。另外，顺序只是一种约定，粗心的程序员很容易忽略，导致死锁。最后，有序加锁需要深入理解代码库，了解各种函数的调用关系，即使一个错误，也会导致“D”字[①]。

提示：通过锁的地址来强制锁的顺序

当一个函数要抢多个锁时，我们需要注意死锁。比如有一个函数：do_something(mutex t *m1, mutex t *m2)，如果函数总是先抢 m1，然后 m2，那么当一个线程调用 do_something(L1, L2)，而另一个线程调用 do_something(L2, L1)时，就可能会产生死锁。

为了避免这种特殊问题，聪明的程序员根据锁的地址作为获取锁的顺序。按照地址从高到低，或者从低到高的顺序加锁，do_something()函数就可以保证不论传入参数是什么顺序，函数都会用固定的顺序加锁。具体的代码如下：

```
if (m1 > m2) {  // grab locks in high-to-low address order
  pthread_mutex_lock(m1);
  pthread_mutex_lock(m2);
} else {
  pthread_mutex_lock(m2);
  pthread_mutex_lock(m1);
```

① “D”表示“Deadlock”。

```
}
// Code assumes that m1 != m2 (it is not the same lock)
```

在获取多个锁时，通过简单的技巧，就可以确保简单有效的无死锁实现。

持有并等待

死锁的持有并等待条件，可以通过原子地抢锁来避免。实践中，可以通过如下代码来实现：

```
1    lock(prevention);
2    lock(L1);
3    lock(L2);
4    ...
5    unlock(prevention);
```

先抢到 prevention 这个锁之后，代码保证了在抢锁的过程中，不会有不合时宜的线程切换，从而避免了死锁。当然，这需要任何线程在任何时候抢占锁时，先抢到全局的 prevention 锁。例如，如果另一个线程用不同的顺序抢锁 L1 和 L2，也不会有问题，因为此时，线程已经抢到了 prevention 锁。

注意，出于某些原因，这个方案也有问题。和之前一样，它不适用于封装：因为这个方案需要我们准确地知道要抢哪些锁，并且提前抢到这些锁。因为要提前抢到所有锁（同时），而不是在真正需要的时候，所以可能降低了并发。

非抢占

在调用 unlock 之前，都认为锁是被占有的，多个抢锁操作通常会带来麻烦，因为我们等待一个锁时，同时持有另一个锁。很多线程库提供更为灵活的接口来避免这种情况。具体来说，trylock()函数会尝试获得锁，或者返回−1，表示锁已经被占有。你可以稍后重试一下。

可以用这一接口来实现无死锁的加锁方法：

```
1    top:
2      lock(L1);
3      if (trylock(L2) == -1) {
4        unlock(L1);
5        goto top;
6    }
```

注意，另一个线程可以使用相同的加锁方式，但是不同的加锁顺序（L2 然后 L1），程序仍然不会产生死锁。但是会引来一个新的问题：活锁（livelock）。两个线程有可能一直重复这一序列，又同时都抢锁失败。这种情况下，系统一直在运行这段代码（因此不是死锁），但是又不会有进展，因此名为活锁。也有活锁的解决方法：例如，可以在循环结束的时候，先随机等待一个时间，然后再重复整个动作，这样可以降低线程之间的重复互相干扰。

关于这个方案的最后一点：使用 trylock 方法可能会有一些困难。第一个问题仍然是封

装：如果其中的某一个锁，是封装在函数内部的，那么这个跳回开始处就很难实现。如果代码在中途获取了某些资源，必须要确保也能释放这些资源。例如，在抢到 L1 后，我们的代码分配了一些内存，当抢 L2 失败时，并且在返回开头之前，需要释放这些内存。当然，在某些场景下（例如，之前提到的 Java 的 vector 方法），这种方法很有效。

互斥

最后的预防方法是完全避免互斥。通常来说，代码都会存在临界区，因此很难避免互斥。那么我们应该怎么做呢？

Herlihy 提出了设计各种无等待（wait-free）数据结构的思想[H91]。想法很简单：通过强大的硬件指令，我们可以构造出不需要锁的数据结构。

举个简单的例子，假设我们有比较并交换（compare-and-swap）指令，是一种由硬件提供的原子指令，做下面的事：

```
1    int CompareAndSwap(int *address, int expected, int new) {
2      if (*address == expected) {
3        *address = new;
4        return 1; // success
5      }
6      return 0;   // failure
7    }
```

假定我们想原子地给某个值增加特定的数量。我们可以这样实现：

```
1    void AtomicIncrement(int *value, int amount) {
2      do {
3        int old = *value;
4      } while (CompareAndSwap(value, old, old + amount) == 0);
5    }
```

无须获取锁，更新值，然后释放锁这些操作，我们使用比较并交换指令，反复尝试将值更新到新的值。这种方式没有使用锁，因此不会有死锁（有可能产生活锁）。

我们来考虑一个更复杂的例子：链表插入。这是在链表头部插入元素的代码：

```
1    void insert(int value) {
2      node_t *n = malloc(sizeof(node_t));
3      assert(n != NULL);
4      n->value = value;
5      n->next  = head;
6      head     = n;
7    }
```

这段代码在多线程同时调用的时候，会有临界区（看看你是否能弄清楚原因）。当然，我们可以通过给相关代码加锁，来解决这个问题：

```
1    void insert(int value) {
2      node_t *n = malloc(sizeof(node_t));
3      assert(n != NULL);
4      n->value = value;
```

```
5      lock(listlock);    // begin critical section
6      n->next  = head;
7      head     = n;
8      unlock(listlock); // end of critical section
9    }
```

上面的方案中，我们使用了传统的锁[①]。这里我们尝试用比较并交换指令（compare-and-swap）来实现插入操作。一种可能的实现是：

```
1    void insert(int value) {
2      node_t *n = malloc(sizeof(node_t));
3      assert(n != NULL);
4      n->value = value;
5      do {
6        n->next = head;
7      } while (CompareAndSwap(&head, n->next, n) == 0);
8    }
```

这段代码，首先把 next 指针指向当前的链表头（head），然后试着把新节点交换到链表头。但是，如果此时其他的线程成功地修改了 head 的值，这里的交换就会失败，导致这个线程根据新的 head 值重试。

当然，只有插入操作是不够的，要实现一个完善的链表还需要删除、查找等其他工作。如果你有兴趣，可以去查阅关于无等待同步的丰富文献。

通过调度避免死锁

除了死锁预防，某些场景更适合死锁避免（avoidance）。我们需要了解全局的信息，包括不同线程在运行中对锁的需求情况，从而使得后续的调度能够避免产生死锁。

例如，假设我们需要在两个处理器上调度 4 个线程。更进一步，假设我们知道线程 1（T1）需要用锁 L1 和 L2，T2 也需要抢 L1 和 L2，T3 只需要 L2，T4 不需要锁。我们用表 32.2 来表示线程对锁的需求。

表 32.2　　线程对锁的需求

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

一种比较聪明的调度方式是，只要 T1 和 T2 不同时运行，就不会产生死锁。下面就是这种方式：

请注意，T3 和 T1 重叠，或者和 T2 重叠都是可以的。虽然 T3 会抢占锁 L2，但是由于它只用到一把锁，和其他线程并发执行都不会产生死锁。

① 聪明的读者可能会问，为什么我们这么晚才抢锁，而不是就在进入 insert()时。聪明的读者，你可以弄清楚为什么这可能是正确的？

我们再来看另一个竞争更多的例子。在这个例子中，对同样的资源（又是锁 L1 和 L2）有更多的竞争。锁和线程的竞争如表 32.3 所示。

表 32.3 锁和线程的竞争

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

特别是，线程 T1、T2 和 T3 执行过程中，都需要持有锁 L1 和 L2。下面是一种不会产生死锁的可行方案：

你可以看到，T1、T2 和 T3 运行在同一个处理器上，这种保守的静态方案会明显增加完成任务的总时间。尽管有可能并发运行这些任务，但为了避免死锁，我们没有这样做，付出了性能的代价。

Dijkstra 提出的银行家算法[D64]是一种类似的著名解决方案，文献中也描述了其他类似的方案。遗憾的是，这些方案的适用场景很局限。例如，在嵌入式系统中，你知道所有任务以及它们需要的锁。另外，和上文的第二个例子一样，这种方法会限制并发。因此，通过调度来避免死锁不是广泛使用的通用方案。

检查和恢复

最后一种常用的策略就是允许死锁偶尔发生，检查到死锁时再采取行动。举个例子，如果一个操作系统一年死机一次，你会重启系统，然后愉快地（或者生气地）继续工作。如果死锁很少见，这种不是办法的办法也是很实用的。

提示：不要总是完美（TOM WEST 定律）

Tom West 是经典的计算机行业小说《Soul of a New Machine》[K81]的主人公，有一句很棒的工程格言：“不是所有值得做的事情都值得做好”。如果坏事很少发生，并且造成的影响很小，那么我们不应该去花费大量的精力去预防它。当然，如果你在制造航天飞机，事故会导致航天飞机爆炸，那么你应该忽略这个建议。

很多数据库系统使用了死锁检测和恢复技术。死锁检测器会定期运行，通过构建资源图来检查循环。当循环（死锁）发生时，系统需要重启。如果还需要更复杂的数据结构相关的修复，那么需要人工参与。

读者可以在其他地方找到更多的关于数据库并发、死锁和相关问题的资料[B+87，K87]。阅读这些著作，当然最好可以通过学习数据库的课程，深入地了解这一有趣而且丰富的主题。

32.4 小结

在本章中，我们学习了并发编程中出现的缺陷的类型。第一种是非常常见的，非死锁

缺陷，通常也很容易修复。这种问题包括：违法原子性，即应该一起执行的指令序列没有一起执行；违反顺序，即两个线程所需的顺序没有强制保证。

同时，我们简要地讨论了死锁：为何会发生，以及如何处理。这个问题几乎和并发一样古老，已经有成百上千的相关论文了。实践中是自行设计抢锁的顺序，从而避免死锁发生。无等待的方案也很有希望，在一些通用库和系统中，包括 Linux，都已经有了一些无等待的实现。然而，这种方案不够通用，并且设计一个新的无等待的数据结构极其复杂，以至于不够实用。也许，最好的解决方案是开发一种新的并发编程模型：在类似 MapReduce（来自 Google）[GD02]这样的系统中，程序员可以完成一些类型的并行计算，无须任何锁。锁必然带来各种困难，也许我们应该尽可能地避免使用锁，除非确信必须使用。

参考资料

[B+87]“Concurrency Control and Recovery in Database Systems”Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman Addison-Wesley, 1987
数据库管理系统中并发性的经典教材。如你所知，理解数据库领域的并发性、死锁和其他主题本身就是一个世界。研究它，自己探索这个世界。

[C+71]“System Deadlocks”
E.G. Coffman, M.J. Elphick, A. Shoshani ACM Computing Surveys, 3:2, June 1971
这篇经典论文概述了死锁的条件以及如何处理它。当然有一些关于这个话题的早期论文，详细信息请参阅该论文的参考文献。

[D64]“Een algorithme ter voorkoming van de dodelijke omarming”Circulated privately, around 1964
事实上，Dijkstra 不仅提出了死锁问题的一些解决方案，更重要的是他首先注意到了死锁的存在，至少是以书面形式。然而，他称之为“致命的拥抱”，（幸好）没有流行起来。

[GD02]“MapReduce: Simplified Data Processing on Large Clusters”Sanjay Ghemawhat and Jeff Dean
OSDI ’04, San Francisco, CA, October 2004
MapReduce 论文迎来了大规模数据处理时代，提出了一个框架，在通常不可靠的机器群集上执行这样的计算。

[H91]“Wait-free Synchronization”Maurice Herlihy
ACM TOPLAS, 13(1), pages 124-149, January 1991
Herlihy 的工作开创了无等待方式编写并发程序的想法。这些方法往往复杂而艰难，通常比正确使用锁更困难，可能会限制它们在现实世界中的成功。

[J+08]“Deadlock Immunity: Enabling Systems To Defend Against Deadlocks”Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea
OSDI ’08, San Diego, CA, December 2008
最近的优秀文章，关于死锁以及如何避免在特定系统中一次又一次地陷入同一个问题。

[K81]“Soul of a New Machine”Tracy Kidder, 1980
任何系统建造者或工程师都必须阅读，详细介绍 Tom West 领导的 Data General（DG）内部团队如何制造“新机器”的早期工作。Kidder 的其他图书也非常出色，其中包括《Mountains beyond Mountains》。 或者，也许你不同意我们的观点？

[K87]“Deadlock Detection in Distributed Databases”Edgar Knapp
ACM Computing Surveys, Volume 19, Number 4, December 1987
分布式数据库系统中死锁检测的极好概述，也指出了一些其他相关的工作，因此是开始阅读的好文章。

[L+08]“Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics”
Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou
ASPLOS ’08, March 2008, Seattle, Washington
首次深入研究真实软件中的并发错误，也是本章的基础。参见 Y.Y. Zhou 或 Shan Lu 的网页，有许多关于缺陷的更有趣的论文。

[T+94]“Linux File Memory Map Code”Linus Torvalds and many others
感谢 Michael Walfish（纽约大学）指出这个宝贵的例子。真实的世界，就像你在这个文件中看到的那样，可能比教科书中的简单、清晰更复杂一些。

第 33 章　基于事件的并发（进阶）

目前为止，我们提到的并发，似乎只能用线程来实现。就像生活中的许多事，这不完全对。具体来说，一些基于图形用户界面（GUI）的应用[O96]，或某些类型的网络服务器[PDZ99]，常常采用另一种并发方式。这种方式称为基于事件的并发（event-based concurrency），在一些现代系统中较为流行，比如 node.js[N13]，但它源自于 C/UNIX 系统，我们下面将讨论。

基于事件的并发针对两方面的问题。一方面是多线程应用中，正确处理并发很有难度。正如我们讨论的，忘加锁、死锁和其他烦人的问题会发生。另一方面，开发者无法控制多线程在某一时刻的调度。程序员只是创建了线程，然后就依赖操作系统能够合理地调度线程。要实现一个在各种不同负载下，都能够良好运行的通用调度程序，是极有难度的。因此，某些时候操作系统的调度并不是最优的。关键问题如下。

关键问题：不用线程，如何构建并发服务器

不用线程，同时保证对并发的控制，避免多线程应用中出现的问题，我们应该如何构建一个并发服务器？

33.1　基本想法：事件循环

我们使用的基本方法就是基于事件的并发（event-based concurrency）。该方法很简单：我们等待某事（即“事件”）发生；当它发生时，检查事件类型，然后做少量的相应工作（可能是 I/O 请求，或者调度其他事件准备后续处理）。这就好了！

在深入细节之前，我们先看一个典型的基于事件的服务器。这种应用都是基于一个简单的结构，称为事件循环（event loop）。事件循环的伪代码如下：

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

它确实如此简单。主循环等待某些事件发生（通过 getEvents()调用），然后依次处理这些发生的事件。处理事件的代码叫作事件处理程序（event handler）。重要的是，处理程序在处理一个事件时，它是系统中发生的唯一活动。因此，调度就是决定接下来处理哪个事件。这种对调度的显式控制，是基于事件方法的一个重要优点。

但这也带来一个更大的问题：基于事件的服务器如何决定哪个事件发生，尤其是对于

网络和磁盘 I/O？具体来说，事件服务器如何确定是否有它的消息已经到达？

33.2 重要 API：select()（或 poll()）

知道了基本的事件循环，我们接下来必须解决如何接收事件的问题。大多数系统提供了基本的 API，即通过 select()或 poll()系统调用。

这些接口对程序的支持很简单：检查是否有任何应该关注的进入 I/O。例如，假定网络应用程序（如 Web 服务器）希望检查是否有网络数据包已到达，以便为它们提供服务。这些系统调用就让你做到这一点。

下面以 select()为例，手册页（在 macOS X 上）以这种方式描述 API：

```
int select(int nfds,
           fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

手册页中的实际描述：select()检查 I/O 描述符集合，它们的地址通过 readfds、writefds 和 errorfds 传入，分别查看它们中的某些描述符是否已准备好读取，是否准备好写入，或有异常情况待处理。在每个集合中检查前 nfds 个描述符，即检查描述符集合中从 0 到 nfds-1 的描述符。返回时，select()用给定请求操作准备好的描述符组成的子集替换给定的描述符集合。select()返回所有集合中就绪描述符的总数。

补充：阻塞与非阻塞接口

阻塞（或同步，synchronous）接口在返回给调用者之前完成所有工作。非阻塞（或异步，asynchronous）接口开始一些工作，但立即返回，从而让所有需要完成的工作都在后台完成。

通常阻塞调用的主犯是某种 I/O。例如，如果一个调用必须从磁盘读取才能完成，它可能会阻塞，等待发送到磁盘的 I/O 请求返回。

非阻塞接口可用于任何类型的编程（例如，使用线程），但在基于事件的方法中非常重要，因为阻塞的调用会阻止所有进展。

关于 select()有几点要注意。首先，请注意，它可以让你检查描述符是否可以读取和写入。前者让服务器确定新数据包已到达并且需要处理，而后者则让服务知道何时可以回复（即出站队列未满）。

其次，请注意超时参数。这里的一个常见用法是将超时设置为 NULL，这会导致 select()无限期地阻塞，直到某个描述符准备就绪。但是，更健壮的服务器通常会指定某种超时。一种常见的技术是将超时设置为零，因此让调用 select()立即返回。

poll()系统调用非常相似。有关详细信息，请参阅其手册页或 Stevens 和 Rago 的书[SR05]。

无论哪种方式，这些基本原语为我们提供了一种构建非阻塞事件循环的方法，它可以简单地检查传入数据包，从带有消息的套接字中读取数据，并根据需要进行回复。

33.3 使用 select()

为了让这更具体，我们来看看如何使用 select()来查看哪些网络描述符在它们上面有传入消息。图 33.1 展示了一个简单的例子。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <sys/time.h>
4    #include <sys/types.h>
5    #include <unistd.h>
6
7    int main(void) {
8        // open and set up a bunch of sockets (not shown)
9        // main loop
10       while (1) {
11           // initialize the fd_set to all zero
12           fd_set readFDs;
13           FD_ZERO(&readFDs);
14
15           // now set the bits for the descriptors
16           // this server is interested in
17           // (for simplicity, all of them from min to max)
18           int fd;
19           for (fd = minFD; fd < maxFD; fd++)
20               FD_SET(fd, &readFDs);
21
22           // do the select
23           int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);
24
25           // check which actually have data using FD_ISSET()
26           int fd;
27           for (fd = minFD; fd < maxFD; fd++)
28               if (FD_ISSET(fd, &readFDs))
29                   processFD(fd);
30       }
31   }
```

图 33.1 使用 select()的简单代码

这段代码实际上很容易理解。初始化完成后，服务器进入无限循环。在循环内部，它使用 FD_ZERO()宏首先清除文件描述符集合，然后使用 FD_SET()将所有从 minFD 到 maxFD 的文件描述符包含到集合中。例如，这组描述符可能表示服务器正在关注的所有网络套接字。最后，服务器调用 select()来查看哪些连接有可用的数据。然后，通过在循环中使用 FD_ISSET()，事件服务器可以查看哪些描述符已准备好数据并处理传入的数据。

当然，真正的服务器会比这更复杂，并且在发送消息、发出磁盘 I/O 和许多其他细节时需要使用逻辑。想了解更多信息，请参阅 Stevens 和 Rago 的书 [SR05]，了解 API 信息，或

Pai 等人的论文、Welsh 等人的论文[PDZ99，WCB01]，以便对基于事件的服务器的一般流程有一个很好的总体了解。

33.4 为何更简单？无须锁

使用单个 CPU 和基于事件的应用程序，并发程序中发现的问题不再存在。具体来说，因为一次只处理一个事件，所以不需要获取或释放锁。基于事件的服务器不能被另一个线程中断，因为它确实是单线程的。因此，线程化程序中常见的并发性错误并没有出现在基本的基于事件的方法中。

提示：请勿阻塞基于事件的服务器

基于事件的服务器可以对任务调度进行细粒度的控制。但是，为了保持这种控制，不可以有阻止调用者执行的调用。如果不遵守这个设计提示，将导致基于事件的服务器阻塞，客户心塞，并严重质疑你是否读过本书的这部分内容。

33.5 一个问题：阻塞系统调用

到目前为止，基于事件的编程听起来很棒，对吧？编写一个简单的循环，然后在事件发生时处理事件。甚至不需要考虑锁！但是有一个问题：如果某个事件要求你发出可能会阻塞的系统调用，该怎么办？

例如，假定一个请求从客户端进入服务器，要从磁盘读取文件并将其内容返回给发出请求的客户端（很像简单的 HTTP 请求）。为了处理这样的请求，某些事件处理程序最终将不得不发出 open()系统调用来打开文件，然后通过一系列 read()调用来读取文件。当文件被读入内存时，服务器可能会开始将结果发送到客户端。

open()和 read()调用都可能向存储系统发出 I/O 请求（当所需的元数据或数据不在内存中时），因此可能需要很长时间才能提供服务。使用基于线程的服务器时，这不是问题：在发出 I/O 请求的线程挂起（等待 I/O 完成）时，其他线程可以运行，从而使服务器能够取得进展。事实上，I/O 和其他计算的自然重叠（overlap）使得基于线程的编程非常自然和直接。

但是，使用基于事件的方法时，没有其他线程可以运行：只是主事件循环。这意味着如果一个事件处理程序发出一个阻塞的调用，整个服务器就会这样做：阻塞直到调用完成。当事件循环阻塞时，系统处于闲置状态，因此是潜在的巨大资源浪费。因此，我们在基于事件的系统中必须遵守一条规则：不允许阻塞调用。

33.6 解决方案：异步 I/O

为了克服这个限制，许多现代操作系统已经引入了新的方法来向磁盘系统发出 I/O 请求，

一般称为异步 I/O（asynchronous I/O）。这些接口使应用程序能够发出 I/O 请求，并在 I/O 完成之前立即将控制权返回给调用者，另外的接口让应用程序能够确定各种 I/O 是否已完成。

例如，让我们来看看在 macOS X 上提供的接口（其他系统有类似的 API）。这些 API 围绕着一个基本的结构，即 struct aiocb 或 AIO 控制块（AIO control block）。该结构的简化版本如下所示（有关详细信息，请参阅手册页）：

```
struct aiocb {
    int              aio_fildes;    /* File descriptor */
    off_t            aio_offset;    /* File offset */
    volatile void    *aio_buf;      /* Location of buffer */
    size_t           aio_nbytes;    /* Length of transfer */
};
```

要向文件发出异步读取，应用程序应首先用相关信息填充此结构：要读取文件的文件描述符（aio_fildes），文件内的偏移量（ai_offset）以及长度的请求（aio_nbytes），最后是应该复制读取结果的目标内存位置（aio_buf）。

在填充此结构后，应用程序必须发出异步调用来读取文件。在 macOS X 上，此 API 就是异步读取（asynchronous read）API：

```
int aio_read(struct aiocb *aiocbp);
```

该调用尝试发出 I/O。如果成功，它会立即返回并且应用程序（即基于事件的服务器）可以继续其工作。

然而，我们必须解决最后一个难题。我们如何知道 I/O 何时完成，并且缓冲区（由 aio buf 指向）现在有了请求的数据？

还需要最后一个 API。在 macOS X 上，它被称为 aio_error()（有点令人困惑）。API 看起来像这样：

```
int aio_error(const struct aiocb *aiocbp);
```

该系统调用检查 aiocbp 引用的请求是否已完成。如果有，则函数返回成功（用零表示）。如果不是，则返回 EINPROGRESS。因此，对于每个未完成的异步 I/O，应用程序可以通过调用 aio_error()来周期性地轮询（poll）系统，以确定所述 I/O 是否尚未完成。

你可能已经注意到，检查一个 I/O 是否已经完成是很痛苦的。如果一个程序在某个特定时间点发出数十或数百个 I/O，是否应该重复检查它们中的每一个，或者先等待一会儿，或者……

为了解决这个问题，一些系统提供了基于中断（interrupt）的方法。此方法使用 UNIX 信号（signal）在异步 I/O 完成时通知应用程序，从而消除了重复询问系统的需要。这种轮询与中断问题也可以在设备中看到，正如你将在 I/O 设备章节中看到的（或已经看到的）。

补充：UNIX 信号

所有现代 UNIX 变体都有一个称为信号（signal）的巨大而迷人的基础设施。最简单的信号提供了一种与进程进行通信的方式。具体来说，可以将信号传递给应用程序。这样做会让应用程序停止当前的任何工作，开始运行信号处理程序（signal handler），即应用程序中某些处理该信号的代码。完成后，该进程就恢复其先前的行为。

每个信号都有一个名称，如 HUP（挂断）、INT（中断）、SEGV（段违规）等。有关详细信息，请参阅手册页。有趣的是，有时是内核本身发出信号。例如，当你的程序遇到段违规时，操作系统发送一个 SIGSEGV（在信号名称之前加上 SIG 是很常见的）。如果你的程序配置为捕获该信号，则实际上可以运行一些代码来响应这种错误的程序行为（这可能对调试有用）。当一个信号被发送到一个没有配置处理该信号的进程时，一些默认行为就会生效。对于 SEGV 来说，这个进程会被杀死。

下面一个进入无限循环的简单程序，但首先设置一个信号处理程序来捕捉 SIGHUP：

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

void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

你可以用 kill 命令行工具向其发送信号（是的，这是一个奇怪而富有攻击性的名称）。这样做会中断程序中的主 while 循环并运行处理程序代码 handle()：

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

要了解信号还有很多事情要做，以至于单个页面，甚至单独的章节，都远远不够。与往常一样，有一个重要来源：Stevens 和 Rago 的书[SR05]。如果感兴趣，请阅读。

在没有异步 I/O 的系统中，纯基于事件的方法无法实现。然而，聪明的研究人员已经推出了相当适合他们的方法。例如，Pai 等人 [PDZ99]描述了一种使用事件处理网络数据包的混合方法，并且使用线程池来管理未完成的 I/O。详情请阅读他们的论文。

33.7 另一个问题：状态管理

基于事件的方法的另一个问题是，这种代码通常比传统的基于线程的代码更复杂。原因如下：当事件处理程序发出异步 I/O 时，它必须打包一些程序状态，以便下一个事件处理

程序在 I/O 最终完成时使用。这个额外的工作在基于线程的程序中是不需要的，因为程序需要的状态在线程栈中。Adya 等人称之为手工栈管理（manual stack management），这是基于事件编程的基础[A + 02]。

为了使这一点更加具体一些，我们来看一个简单的例子，在这个例子中，一个基于线程的服务器需要从文件描述符（fd）中读取数据，一旦完成，将从文件中读取的数据写入网络套接字描述符 SD）。代码（忽略错误检查）如下所示：

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

如你所见，在一个多线程程序中，做这种工作很容易。当 read()最终返回时，代码立即知道要写入哪个套接字，因为该信息位于线程堆栈中（在变量 sd 中）。

在基于事件的系统中，生活并没有那么容易。为了执行相同的任务，我们首先使用上面描述的 AIO 调用异步地发出读取。假设我们使用 aio_error()调用定期检查读取的完成情况。当该调用告诉我们读取完成时，基于事件的服务器如何知道该怎么做？

解决方案，如 Adya 等人[A+02]所述，是使用一种称为“延续（continuation）”的老编程语言结构[FHK84]。虽然听起来很复杂，但这个想法很简单：基本上，在某些数据结构中，记录完成处理该事件需要的信息。当事件发生时（即磁盘 I/O 完成时），查找所需信息并处理事件。

在这个特定例子中，解决方案是将套接字描述符（sd）记录在由文件描述符（fd）索引的某种数据结构（例如，散列表）中。当磁盘 I/O 完成时，事件处理程序将使用文件描述符来查找延续，这会将套接字描述符的值返回给调用者。此时（最后），服务器可以完成最后的工作将数据写入套接字。

33.8 什么事情仍然很难

基于事件的方法还有其他一些困难，我们应该指出。例如，当系统从单个 CPU 转向多个 CPU 时，基于事件的方法的一些简单性就消失了。具体来说，为了利用多个 CPU，事件服务器必须并行运行多个事件处理程序。发生这种情况时，就会出现常见的同步问题（例如临界区），并且必须采用通常的解决方案（例如锁定）。因此，在现代多核系统上，无锁的简单事件处理已不再可能。

基于事件的方法的另一个问题是，它不能很好地与某些类型的系统活动集成，如分页（paging）。例如，如果事件处理程序发生页错误，它将被阻塞，并且因此服务器在页错误完成之前不会有进展。尽管服务器的结构可以避免显式阻塞，但由于页错误导致的这种隐式阻塞很难避免，因此在频繁发生时可能会导致较大的性能问题。

还有一个问题是随着时间的推移，基于事件的代码可能很难管理，因为各种函数的确切语义发生了变化[A+02]。例如，如果函数从非阻塞变为阻塞，调用该例程的事件处理程序也必须更改以适应其新性质，方法是将其自身分解为两部分。由于阻塞对于基于事件的服务器而言是灾难性的，因此程序员必须始终注意每个事件使用的 API 语义的这种变化。