另一个重要因素是用户级线程可以使用专为应用程序定制的线程调度程序。例如，考虑图2-8中的Web服务器。假设一个工作线程刚刚被阻塞，而分派线程和另外两个工作线程是就绪的。那么应该运行哪一个呢？由于运行系统了解所有线程的作用，所以会直接选择分派线程接着运行，这样分派线程就会启动另一个工作线程运行。在一个工作线程经常阻塞在磁盘I/O上的环境中，这个策略将并行度最大化。而在内核级线程中，内核从来不了解每个线程的作用（虽然它们被赋予了不同的优先级）。不过，一般而言，应用定制的线程调度程序能够比内核更好地满足应用的需要。

2.5 经典的IPC问题

操作系统文献中有许多广为讨论和分析的有趣问题，它们与同步方法的使用相关。以下几节我们将讨论其中两个最著名的问题。

2.5.1 哲学家就餐问题

1965年，Dijkstra提出并解决了一个他称之为*哲学家就餐*的同步问题。从那时起，每个发明新的同步原语的人都希望通过解决哲学家就餐问题来展示其同步原语的精妙之处。这个问题可以简单地描述如下：五个哲学家围坐在一张圆桌周围，每个哲学家面前都有一盘通心粉。由于通心粉很滑，所以需要两把叉子才能夹住。相邻两个盘子之间放有一把叉子，餐桌如图2-44所示。

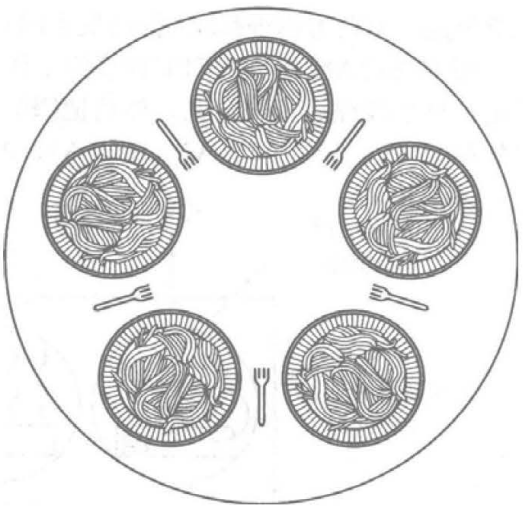

图2-44 哲学家的午餐时间

哲学家的生活中有两种交替活动时段：即吃饭和思考（这只是一种抽象，即对哲学家而言其他活动都无关紧要）。当一个哲学家觉得饿了时，他就试图分两次去取其左边和右边的叉子，每次拿一把，但不分次序。如果成功地得到了两把叉子，就开始吃饭，吃完后放下叉子继续思考。关键问题是：能为每一个哲学家写一段描述其行为的程序，且决不会死锁吗？（要求拿两把叉子是人为规定的，我们也可以将意大利面条换成中国菜，用米饭代替通心粉，用筷子代替叉子。）

图2-45给出了一种直观的解法。过程take_fork将一直等到所指定的叉子可用，然后将其取用。不过，这种显然的解法是错误的。如果五位哲学家同时拿起左面的叉子，就没有人能够拿到他们右面的叉子，于是发生死锁。

我们可以将这个程序修改一下，这样在拿到左叉后，程序要查看右面的叉子是否可用。如果不可用，则该哲学家先放下左叉，等一段时间，再重复整个过程。但这种解法也是错误的，尽管与前一种原因不同。可能在某一个瞬间，所有的哲学家都同时开始这个算法，拿起其左叉，看到右叉不可用，又都放下左叉，等一会儿，又同时拿起左叉，如此这样永远重复下去。对于这种情况，所有的程序都在不停地运行，但都无法取得进展，就称为饥饿（starvation）。（即使问题不发生在意大利餐馆或中国餐馆，也被称为饥饿。）

```
#define N 5                                    /* 哲学家的数目 */

void philosopher(int i)                        /* i：哲学家编号，从0到4 */
{
    while (TRUE) {
        think( );                              /* 哲学家在思考 */
        take_fork(i);                          /* 拿起左边叉子 */
        take_fork((i+1) % N);                  /* 拿起右边叉子，%是模运算 */
        eat( );                                /* 进食 */
        put_fork(i);                           /* 将左叉放回桌上 */
        put_fork((i+1) % N);                   /* 将右叉放回桌上 */
    }
}
```

图2-45 哲学家就餐问题的一种错误解法