之间的选择（或者使用这种原语或者另一种原语），当然也有少数系统中两种原语同时可用，而让用户决定其喜好。

但是，非阻塞原语所提供的性能优点被其严重的缺点所抵消了：直到消息被送出发送者才能修改消息缓冲区。进程在传输过程中重写消息的后果是如此可怕以致不得不慎重考虑。更糟的是，发送进程不知道传输何时会结束，所以根本不知道什么时候重用缓冲区是安全的。不可能永远避免再碰缓冲区。

有三种可能的解决方案。第一种方案是，让内核复制这个消息到内部的内核缓冲区，然后让进程继续，如图8-19b所示。从发送者的观点来看，这个机制与阻塞调用相同：只要进程获得控制，就可以随意重用缓冲区了。当然，消息还没有发送出去，但是发送者是不会被这种情况所妨碍的。这个方案的缺点是对每个送出的消息都必须将其从用户空间复制进内核空间。面对大量的网络接口，消息最终要复制进硬件的传输缓冲区中，所以第一次的复制实质上是浪费。额外的复制会明显地降低系统的性能。

第二种方案是，当消息发送之后中断发送者，告知缓冲区又可以使用了。这里不需要复制。从而节省了时间，但是用户级中断使编写程序变得棘手，并可能会要处理竞争条件，这些都使得该方案难以设计并且几乎无法调试。

第三种方案是，让缓冲区写时复制（copy on write），也就是说，在消息发送出去之前将其标记为只读。在消息发送出去之前，如果缓冲区被重用，则进行复制。这个方案的问题是，除非缓冲区被孤立在自己的页面上，否则对临近变量的写操作也会导致复制。此外，需要有额外的管理，因为这样的发送消息行为隐含着对页面读/写状态的影响。最后，该页面迟早会再次被写入，它会触发一次不再必要的复制。

这样，在发送端的选择是

1) 阻塞发送（CPU在消息传输期间空闲）。

2) 带有复制操作的非阻塞发送（CPU时间浪费在额外的复制上）。

3) 带有中断操作的非阻塞发送（造成编程困难）。

4) 写时复制（最终可能也会需要额外的复制）。

在正常条件下，第一种选择是最好的，特别是在有多线程的情况下，此时当一个线程由于试图发送被阻塞后，其他线程还可以继续工作。它也不需要管理任何内核缓冲区。而且，正如将图8-19a和图8-19b进行比较所见到的，如果不需要复制，通常消息会被更快地发出。

请注意，有必要指出，有些作者使用不同的判别标准区分同步和异步原语。另一种观点认为，只有发送者一直被阻塞到消息已被接收并且有响应发送回来时为止，才是同步的（Andrews，1991）。但是，在实时通信领域中，同步有着其他的含义，不幸的是，它可能会导致混淆。

正如send可以是阻塞的和非阻塞的一样，receive也同样可以是阻塞的和非阻塞的。阻塞调用就是挂起调用者直到消息到达为止。如果有多线程可用，这是一种简单的方法。另外，非阻塞receive只是通知内核缓冲区所在的位置，并几乎立即返回控制。可以使用中断来告知消息已经到达。然而，中断方式编程困难，并且速度很慢，所以也许对于接收者来说，更好的方法是使用一个过程poll轮询进来的消息。该过程报告是否有消息正在等待。若是，调用者可调用get_message，它返回第一个到达的消息。在有些系统中，编译器可以在代码中合适的地方插入poll调用，不过，要掌握以怎样的频度使用poll则是需要技巧的。

还有另一个选择，其机制是在接收者进程的地址空间中，一个消息的到达自然地引起一个新线程的创建。这样的线程称为弹出式线程（pop-up thread）。这个线程运行一个预定义的过程，其参数是一个指向进来消息的指针。在处理完这个消息之后，该线程直接退出并被自动撤销。

这一想法的变种是，在中断处理程序中直接运行接收者代码，从而避免了创建弹出线程的麻烦。要使这个方法更快，消息自身可以带有该处理程序的句柄（handler），这样当消息到达时，只在少数几个指令中可以调用处理程序。这样做的最大好处在于再也不需要复制了。处理程序从接口板取到消息并且即时处理。这种方式称为主动消息（active messages，Von Eicken等人，1992）。由于每条消息中都有处理程序的句柄，主动消息方式只能在发送者和接收者彼此完全信任的条件下工作。

8.2.4 远程过程调用

尽管消息传递模型提供了一种构造多计算机操作系统的便利方式，但是它有不可救药的缺陷：构造所有通信的范型（paradigm）都是输入/输出。过程send和 receive 基本上在做I/O工作，而许多人认为