半同步半异步

半同步半异步模式
一个架构模式，清晰的结构,高效并发的I/O

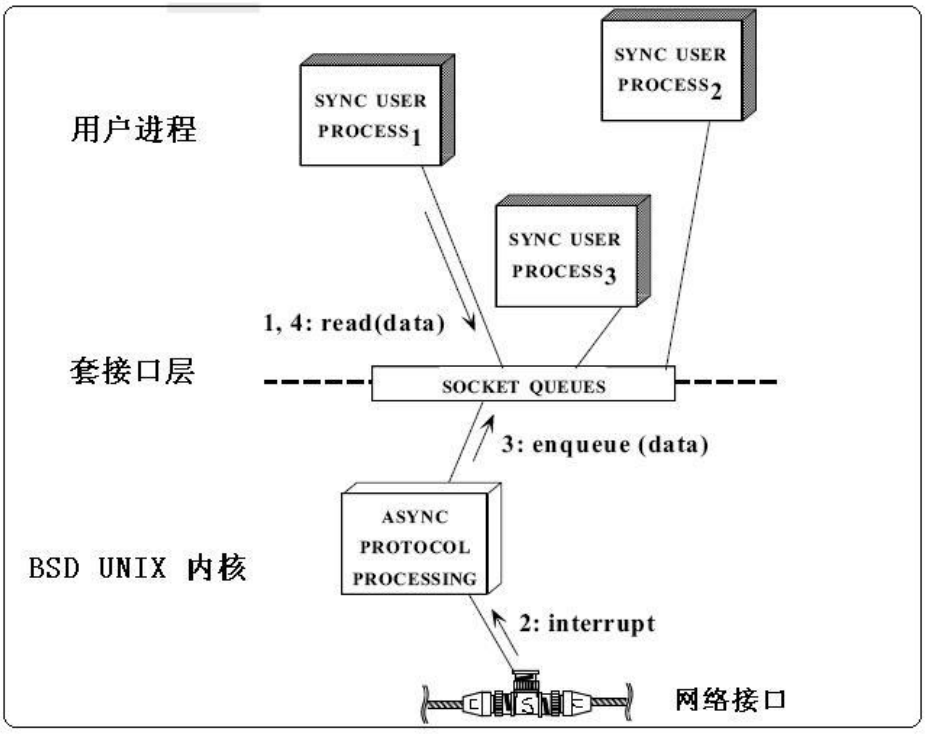
译者：cuichaox@gmail.com

英文原文：http://www.cs.wustl.edu/~schmidt/PDF/HS-HA.pdf

摘要
这 篇文字介绍了半同步半异步模式，这个模式运用在复杂的并行系统中，把同步和异步I/O模型集成在一起，既保持了编程简单又保证了执行的效率。这个模式中， 高层使用同步I/O模型，简化编程。低层使用异步I/O模型，高效执行。各种操作系统和其它复杂的并行系统中广泛使用这个模式，使用这个模式的操作系统 有：UNIX，Mach，Windows NT，and VMS。

1 意图
在系统中分离了同步I/O和异步I/O的两个过程，既简化了并行程序实现的复杂，又不会影响执行的效率。

2 动机
如 图一所示，为了说明半同步半异步模式，我们来考虑一个软件架构，BSD UNIX[1]的网络子系统。BSD UNIX的内核负责协调异步通信设备（如：网络适配器、远程终端）和应用程序之间的I/O行为。数据包随时到达设备，产生硬件中断，然后由中断处理例程把 数据交给操作系统内核。这些中断处理例程接收数据，触发上层的网络协议处理（如 IP，TCP和UDP）。应用数据包在套接口层排队。操作系统把这些数据包分发给每个用户进程。用户进程使用系统调用read，以同步的方式接收套接口层 的数据。用户进程可以在任意时刻调用read，当数据没有到达的时候，进程将一直阻塞等待。



图一：BSD UNIX 网络子系统的软件体系结构

在这个体系中，操作系统内核响应设备的中断，执行异步的I/O。而用户级的应用程序进行同步的I/O。这正是“半同步半异步”这个名字的由来，这个结构满足下面的两个需要：

编 程实现简单。异步I/O模型中，中断随时触发输入和输出操作，编程复杂。使用异步I/O模型，当中断处理例程拥有线程控制权时,会产生非常麻烦的时序和竟 争问题。而且，使用中断机制的程序要求额外的数据结构，这个数据结构用于在异步事件发生的时候保存进程上下文状态。并且，程序执行的时候，外部的事件会在 不固定的时间发生，程序不容易调试。

与其相比，使用同步I/O模型的时候，I/O操作在确定 的点发生，编程实现要容易的多。此外，同步I/O操作的程序会阻塞等待I/O操作的完成。进程运行上下文的活动记录自动在运行栈中保存，不必使用独立的数 据结构。因此，为了让编程简单，有强烈的理由使用同步I/O模型。

程序执行高效。在中断驱动的设备上，异步I/O模型运用带来了高效率。异步I/O让通信和计算同时进行。并且，因为程序运行状态的数据量相对较小，上下文切换的消耗被最小化[2]。因此，为了提高运行的性能，也有强烈理由使用异步I/O模型。

与 其相比，如果每种资源的事件（例如网卡，终端和计时器）占用一个独立的活动对象（进程或线程），一个完全同步I/O模型效率会低。每个活动对象包含多个资 源（例如栈，寄存器），每种资源都会让它阻塞，等待资源事件的发生。在创建，调度，分发和终止这些独立的活动对象，会消耗更多的时间和空间。

3 方案

编程简单和高效率执行是矛盾的，半同步半异步模式目的正是为了解决这个矛盾。这个模式集成了同步和异步两种I/O模式，结构清晰，效率高。在这个模式中， 上层的任务（如：数据库查询，文件传输）使用同步I/O模型，简化了编写并行程序的难度。而底层的任务（如网络控制器的中断处理）使用异步I/O模型，提 供了执行效率。一般情况下，上层的任务要比下层的任务多，使用一个简单的层次实现异步处理的复杂性，可以对外隐藏异步处理的细节。另外，同步层次和异步层 次任务间的通信使用一个队列来协调。

4 应用

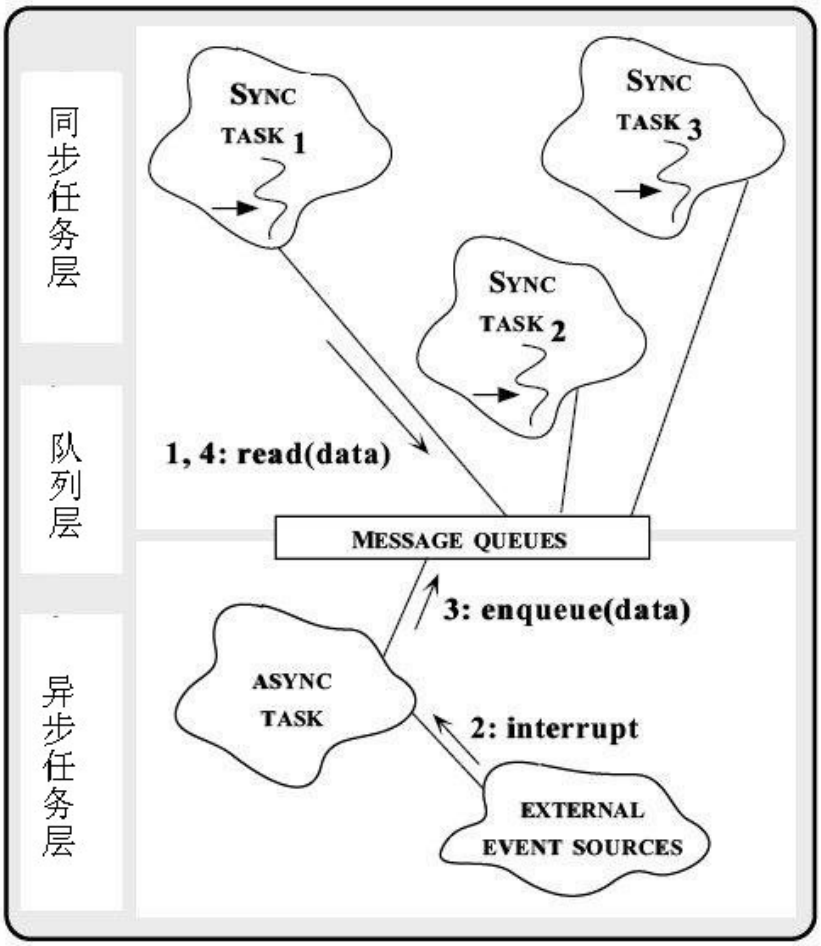
半同步半异步模式在下面的场景中使用。

× 一个系统中的进程有下面的特征：
系统必须响应和处理外部异步发生的事件，
如果为每一个外部资源的事件分派一个独立的线程同步处理I/O，效率很低。
如果上层的任务以同步方式处理I/O，实现起来简单。

× 一个或多个任务必须在单独的控制线程中执行，其它任务可以在多线程中执行。
例如，X Window和Sun RPC 的库函数许多是不可重入的。多个控制线程不能安全地同时调用这些函数。然而，为了使多个CPU提高服务的质量，有必要使用并行，不同的线程同时执行批量数 据的传输或数据库的查询。使用半同步半异步模式，可以在多线程中分离出单独的线程。在不改变现有代码的情况下，这个分离可以让不可重入的函数在并行环境正 确执行。

5 结构和组成单元

图二显示了半同步半异步模式的组成结构。这些组成单元如下描述



图二 半同步半异步模式的组成结构

× 同步任务层（用户级的进程）
本层的任务完成上层的I/O操作，使用同步I/O模型，通过队列层的队列中传输数据。和异步层不同，同步层的任务使用活动对象[3]执行，这些活动对象有自己运行栈和寄存器状态。当执行同步I/O的时候，他们会被阻塞/睡眠。

× 队列层（套接口层）
这个层在同步任务层和异步任务层之间，提供了同步控制和缓存的功能。异步任务的I/O事件被缓存到消息队列中，同步任务层在队列中提取这些事件（相反方向亦然）。

× 异步任务层
处理低层的事件，这些事件由多个外部的事件源产生（例如网卡，终端）。和异步任务不同，此层的实体是被动对象，没有自己的运行栈，要求不能被阻塞。

× 外部事件源（网络接口）
外部设备产生事件，产生的事件先被异步任务层接收和处理。

6 协作

图三展现了一个动态的过程：当外部事件到达后，半同步半异步模式的各个组成单元协作和处理。把协作的过程分成下面三个阶段：

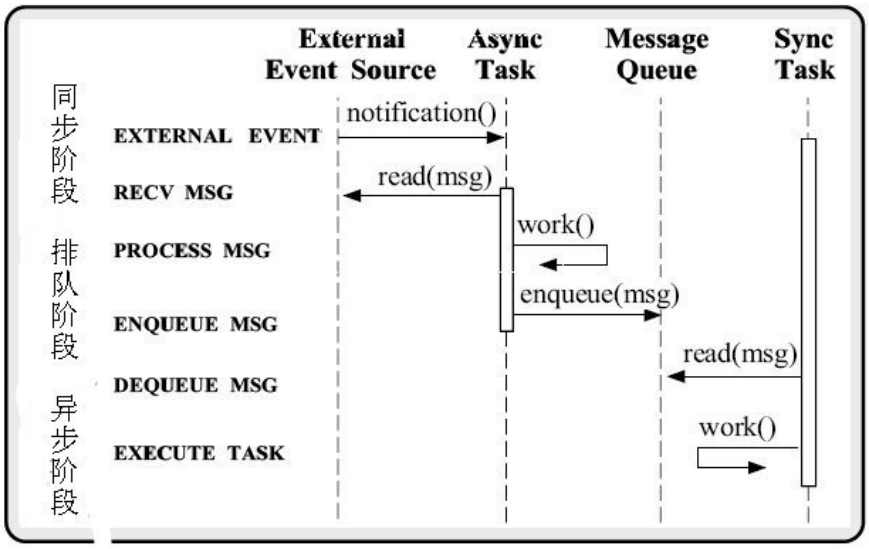
×

异步阶段。通过中断机制或异步事件通知，外部的事件源和异步任务层完成交互。

×排队阶段。队列层的队列提供了一个同步控制机制，响应输入事件，缓存同步层和异步层之间的消息，

×同步阶段。同步任务层在队列层提取消息。注意，同步层和异步层专递数据的协议是独立的，和队列层具体处理通信的方式无关。

在图三中，同步层和异步层之间的通信使用生产者/消费者模型。理解这个模型的关键是：完成同步任务的是活动对象。他们可以在任意时刻阻塞调用read或在 write。如果数据没有准备好，这些活动对象可以一直等待。相反的，异步任务层的实体是被动对象。他们不能被阻塞。这些对象被通知或外部对象触发。



图三 半同步半异步模式各层之间的协作

7 结果

半同步半异步模式有下面的优点

×上层的任务被简化，这是因为不需要再面对底层的异步I/O。复杂的并发控制，中断处理和计时操作都委托给异步任务层。异步层负责处理底层的细节，如，异步系统编程的中断处理。异步任务层管理与硬件相关的组件间（如DMA，内存管理，设备寄存器）的交互。

×不同层可以使用不同的同步策略。每一层不必使用相同的并行控制方式。例如，在单线程的BSD UNIX内核中，异步任务层使用低级的机构（如修改CPU的中断级别）。与此不同，用户级的进程使用高级的同步机构（如使用信号量，消息队列，条件变量，记录锁）。

×层间的通信被限制在单独的一点，因为所有的交互使用队列层协调。队列层对两层之间的消息进行缓存。如果直接通信，同步和异步任务层直接访问对方的内存，就必须使用复杂的锁和时序控制。

×在多处理器环境中提高了性能。使用同步的I/O可以简化编程和提高性能。例如，长时间的数据传输（如从数据库中加载一个大的医学图象）使用同步I/O效率更高。因为一个处理器单独为一个线程使用，可以更有效的使用CPU的指令和数据的缓存。

半同步半异步模式有下面的缺点

× 跨边界导致的性能消耗，这是因为同步控制，数据拷贝和上下文切换会过度地消耗资源。在同步和异步任务层之间使用队列层传送数据的时候，这种消耗往往会发 生。特别是，许多使用半同步半异步模式的操作系统，把队列层放到用户和内核域的边界上。到跨越这个边界的时候，往往会有明显的性能消耗。例如，BSD UNIX的套接口层，在很大比例上造成了TCP/IP网络的过载[4]。

× 上层任务缺少异步I/O的实现。依赖与系统接口的设计，可能导致上层无法直觉使用低层的异步I/O设备。因此，如果外部的设备支持异步的重叠的I/O方式，系统的I/O结构会防碍应用程序高效的使用。

8 实现

这一节描述如何实现半同步半异步模式，系统中分成了同步任务层和异步任务层，两层之间使用队列层通信。

8.1 找到消耗时间的长时任务，使用同步I/O实现他们。

使用同步I/O方式，可以让系统任务的实现简单化，而系统中经常有会时间长的任务，如执行大量流数据碰[5]传输，或者是等待服务器响应的数据查询。

使 用活动对象[3]模型实现这些长时任务。活动对象拥有自己的运行栈和寄存器状态，在执行同步I/O的时候可以被阻塞。实现活动对象结构，要求一个控制线程 的切换机制，需要一个地方存放和恢复线程的状态（如寄存器的值，栈指针），这些功能足够实现一个非抢占的，没有内存保护的线程机构。“用户级线程”包一般 实现了此类的功能（译者：操作系统都实现了这个功能）。
然而，在一个要求健壮的多任务系统中，要使用进程或线程的方式实现活动对象，需要更多的功 能。这种情况下，每种线程需要自己的线程（或进程）空间，这些空间被处理器的内存管理单元（MMU）管理。当线程（或进程）切换的时候，新的地址空间信息 需要加载到MMU中。切换时还需要清空高速缓存，特别是高速缓存使用虚拟地址的情况下。除了地址空间，一个操作系统进程还可能包含一个“用户ID”。这个标识告诉操作系统这个进程的访问权限和占用的系统资源。
为了防止一个单独的线程（或进程）永久的占用系统，要有抢占的机制。经常使用计时器实现，计时器周期性的产生时钟中断。在中断处理期间，操作系统检查当前的进程是否应该被抢占。如果应该被抢占，就保存当前进程的状态，加载下一个进程的状态运行。

8.2 找到短时任务，使用异步I/O实现

系统中的一些任务不能阻塞过长的时间。这些任务一般和外界事件完成一个短期的交互过程（如图形用户界面或中断驱动的网络接口器），为了提高效率和保证响应速度，这些外部事件的处理一定不能阻塞。
实现这样的短期任务，要使用被动对象[6]模型。被动对象使用别人的控制线程（调用者的线程或者一个独立的中断栈）。因为他们不能常时间的等待，不能被阻塞。不阻塞的主要目标是保证其响应时间（如高优先级的硬件中断，如时钟中断）。
实现一个结构清晰的异步I/O框架，有多种方式。

× 使用反应堆模式[6]实现多路事件的处理，反应堆模式使用一个单线程的处理循环，把多路的事件派发给多个处理者。这个模式组合了单线程处理循环的简单性和 面向对象编程提供的可扩展性。反应堆模式在一个线程（或进程）中进行顺序的消息处理，常用来消除多线程同步和加锁处理的复杂性。
一个反应堆可以使用在同步或者异步的事件源上。但是它支持的事件处理者的行为要求是异步的。也就是说，为了不影响其它事件源的响应效率，事件处理者是不能阻塞的。

×实现一个多级的中断处理机构。这种机构下，当更高级别的任务（如硬中断）要求处理的时候，当前的进程可以被中断。为了防止共享的状态访问时被破坏，异步层使用的数据结构必须被保护（例如提升处理器级别或使用信号量）。

× 例如，在一个操作系统内核中，硬件中断的服务时间很大程度上决定了对多级中断处理机构的需要。如果这个时间能够显著的减少，可以把所有的处理放到硬中断的 层次上，这样就可以避免软中断的过度资源消耗。TCP/IP的实现中，就减少输入包的协议处理时间化费，让所有的包处理过程可以在两级中断机构实现。

8.3 实现一个队列层

队列层包含了一个同步控制点，缓存同步任务和异步任务之间交换的消息。在实现队列层的时候，要注意下面几点：

× 并行控制。如果同步任务和异步任务的执行是并行的（如论使用多CPU还是硬件中断），为了避免争用，共享的队列的状态变化必须是连续的。因此，实现队列层 的时候，经常使用并行控制机制，如信号量，互斥体和条件变量等。当消息在队列中插入或删除的时候，并行控制保证队列的内部数据结构不被破坏。

×层到层之间的流量控制。在队列层缓存消息，系统不能提供无限的资源。因此，必须控制同步和异步层之间传输的数据量。例如，在层到层的流控制下，避免同步层的数据量超过网络接口能够传输的极限。

同步任务可以被阻塞。因此，可以使用下面的策略：如果其排队超过一定的量，可以让任务阻塞。当异步任务层把排队数降低到一定的水平，就可以重新唤醒同步任 务继续执行。相对地，异步层地任务不能被阻塞，当处理过量地数据时，队列层要根据策略丢弃消息。这种情况下，如果使用了一个可靠的，面向连接的网络协议， 发送者最终会发现传输超时，要求重新传输数据。

×数据拷贝消耗。一些系统（如BSD UNIX），把队列层放到了用户和内核之间的保护边界上。为了分离不同的保护域，一般使用拷贝数据的方法。然而，这增加了系统总线和内存的负担。当大数据量的消息传输的时候，这可能会降低很大的性能。

一种减少数据拷贝的方式：分配一个专用的内存区，这个内存区被同步任务层和异步任务层共享[7]。这样，两层之间可以高效率的交换数据，不需要拷贝。例如 [8]，存在一个I/O子系统，使用轮询的中断机制改进连续的I/O流处理，最小化跨边界数据传送消耗。这种方法同时提供了一个缓存管理系统，准许高效率 的页影射，一个由用户进程，内核，设备使用的共享内存机构。

9 例子代码

这 节说明两个使用半同步半异步模式的例子，这是在BSD UNIX中两个不同部分。这些例子将说明：半同步半异步模式如果让用户进程同步操作，让内核进程异步操作。第一个例子，在网络子系统中使用这个模式，在以 太网上处理通过TCP/IP协议栈输入的数据。第二个例子，在文件系统上使用这个模式，在磁盘控制器上实现一个中断驱动的输出过程。

9.1 BSD网络子系统的例子

这个例子说明了如何把半主动半自动模式应用与read系统调用，实现同步的操作。异步地接收和处理到达网络接口的数据，同步完成read的调用。图一 说明了BSD UNIX中，这个模式的参与者和结构。关于BSD UNIX网络子系统的完整解释，请参考[9].

9.1.1同步调用

考 虑一个用户进程，创建了一个被动模式的TCP流套接口，从连接的套接口描述子中接收TCP数据。对用户进程来说，read系统调用就是一个同步的操作，进 程调用然后数据返回。当read被调用时，它进入操作系统内核，进入网络套接口的实现代码。最终，控制线程会进入soreceive函数，这个函数完成半 同步部分的处理。soreceive负责把套接口队列的数据传递给用户。它必须能够处理多种类型的套接口（如数据报套接口和流套接口）。下面是简化后的样 子，重点强调了同步和异步任务层之间的边界。

```
/* Receive data from a socket. */
int soreceive ( ... )
{
  for (;;) {
      sblock (...); /* lock socket recv queue */
      /* mask off network interrupts to protect queue */
      s = splnet ();
      if (not enough data to satisfy read request) {
          sbunlock (...); /* unlock socket queue */
          /***** Note! *****
          * The following call forms the boundary
          * between the Sync and Async layers. */
          sbwait (...); /* wait for data */
          splx (s); /* drop splnet */
```

```
        }
        else
            break;
    }
    splx (s); /* drop splnet */
    /* copy data to user's buffer at normal priority */
    uiomove (...);
    s = splnet (); /* mask off network interrupts */
    sbunlock (...); /* unlock socket queue */
    splx (s); /* restore spl */
    return (error code); /* returns 0 if no error */
}
```

上面的代码展示了同步的用户进程和异步的内核进程之间的边界。用户进程可以阻塞等待数据，内核却不能被挂起，因为其它的用户进程或硬件设备需要它的服务。

soreceive处理read的请求有多种方式，这取决于套接口的属性和当前队列中的数据。

×完全同步。如果用户请求的数据已经在套接口队列中，数据将被立即取出，同步地完成操作。

×半同步半异步。如果请求的数据还没有到达内核，内核将调用sbwait函数，让用户进程阻塞至数据到达。

一旦sbwait让进程进入随眠状态，操作系统的调度者会立即把上下文切换到其它进程运行。但原来的用户进程看起来，read系统调用是同步执行的。当包含数据的包到达内核时，将被异步处理，见9.1.2的描述。当足够多的数据到达队列后，内核将唤醒原来的进程进行处理。

9.1.2 异步数据接收和协议处理

当 异步数据包到达网络接口的时候，半异步部分启动。所有输入的包在中断处理例程中处理。中断处理，没有进程上下文也没有控制线程，这期间是不能睡眠（被阻 塞）的。因此，一个中断处理例程必须借用调用者所在的控制线程（例如，它的运行栈和寄存器）执行。BSD UNIX 内核使用这个策略，借用发起系统调用的用户线程。

许多中断驱动的计算机给中断分配优先级。例 如，在SPARC体系中，一共有15个中断级别，最低的是Level 1 最高的是Level 15。其它的处理器有不同的级别数（例如，Motorola 68030有7个中断级别）。在BSD UNIX中，设计了与机器无关的中断级别，叫SPL级别（SPL这个词原于古老的PDP-11机器上的UNIX系统），把每个处理器特有的级别将影射到这 个SPL级别上。例如，最高的网络中断级别为SPLIMP，时钟中断为SPLCLOCK，最高的系统中断级别是SPLHIGH。每一个中断级别对应一个函 数名字，这个函数完成处理器对应中断级别的设置。所以，函数splimp调用后，将把所有的网络硬件中断排除在外。所有的spl*函数返回上一个处理器级 别，这个值供恢复使用。

传统的BSD UNIX版本使用两级的中断结构进行数据包的处理。硬件级关键处理在高级别（SPLIMP）上完成，较不关键的软件处理在低级别（SPLNET）上完成。 这种两个级别的体系，防止软件协议处理的过度消耗对硬中断处理造成延时。两个级别的包处理体系自然分成了两部分：硬件相关的处理和软件协议处理。当一个包 到达网络接口时，产生那个接口级别的中断，所有网络接口的中断优先级<=SPLIMP。 操作系统处理硬件的中断，把进入的数据包交给协议层处理（如IP协议）。当硬中断的处理完成，并且没有其它的未处理完的高级别中断，较低级别的软件中断 （SPLNET）开始进行剩余的协议的处理。BSD的内核被小心的设计，保证在软中断处理的过程中，如果又发生硬中断，不会造成数据丢失和缓存被破坏的情况。

考虑一个例子，一个主机安装了AMD LANCE的以太网卡，这个设备的驱动名字叫“le”。当数据到达的时候，中断处理中lerint函数被调用。它的工作是确认和清除中断，从包中提取数据，把数据拷贝到名字

为mbuf的内存缓存区。如下:

```
int lerint (...)
{
  /* perform hardware sanity checks */
  while (inbound buffers to process) {
     /* get length and clear interrupt ... */
     /* read the packet into mbufs */
     ether_input (interface, ether_type, packet);
     /* free buffer */
  }
}
```

mbuf被传递给以太函数ether_init

```
int ether_input (char *intf, int etype, struct mbuf *packet)
{
   switch (etype) {
      case ETHERTYPE_IP:
      /* schedule network interrupt */
       schednetisr (NETISR_IP);
       inq = &ipintrq;
      break;
      /* etc... */
   }
   s = splimp ();
   /* Try to insert the packet onto the IP queue. */
   if (IF_QFULL (inq)) {
      /* queue full, drop packet */
      IF_DROP (inq);
      m_freem (packet);
   }else
      /* queue packet for net interrupt */
      IF_ENQUEUE (inq, m);
      splx (s);
}
```

每一个网络协议拥有一个数据包队列（例如，IP 包队列），ether_input函数先检测应该使用哪一个网络协议，把数据包入队到正确的协议的队列中。然后，一个软中断发生，这个中断是较低的SPLNET级别。到了这里，硬件中断的处理已经完成，中断服务例程退出。

硬 中断完成后，当没有更高级别中断的时候，SPLNET级别的软中断发生。如果输入的数据包是一个IP数据包，内核将调用IP中断例程（ipintr）。在 这个例程中进行IP协议的处理（如：消息头解析，包转发，分解和重组）。如果数据包确定要发给本机的进程，把数据包传递给传输层。传输层会进行更多的协议 处理（如TCP协议级的重组和确认）。最终传输层把数据放到套接口队列中，调用sbwakeup，这个调用将唤醒原来的用户进程，这个进程之前在 soreceive调用中阻塞。当这些工作完成，软中断的包处理完成。

下面的代码对ipintr的处理流程进行了说明，从tcp_input,到sowakeup，组成了同步和异步层的边界，第一个函数ipintr，处理输入的数据包。

```
int ipintr (...)
{
   int s;
   struct mbuf *m;
   /* loop, until there are no more packets */
```

```
  for (;;) {
    s = splimp ();
    IF_DEQUEUE (&ipintrq, m); /* dequeue next packet */
    splx(s);
    if (m == 0) return; /* return if no more packets */
    if (packet not for us) {
       /* route and forward packet */
    } else {
      /* packet for us... reassemble */
      /* call protocol input, which is tcp_input() */
      (*inetsw[ip_protox[ip->ip_p]].pr_input)(m, hlen);
    }
  }
}
```

在我们的例子中，要处理的是一个TCP/IP的包，inetsw函数会根据判断结果调用tcp_input函数。这个函数处理一个输入的TCP数据包。

```
int tcp_input (m, iphlen)
{
  /* lots of complicated protocol processing... */
  /* We come here to pass data up to the user */
  sbappend (&so->so_rcv, m);
  sowakeup((so), &(so)->so_rcv);
  /* ... */
}
```

函数sowakeup将唤醒在read调用中睡眠的用户进程，这个进程之前一直在等待数据包的到达。在下面一节的讨论中将会看到，这个函数组成了同步和异步层的边界。

9.1.3 同步阶段的完成

当数据添加到队列后，如果有一个用户进程正在睡眠中等待这个数据，sowakeup函数将被调用。

```
void sowakeup (so, sb)
{
  /* ... */
  if (a user process is asleep on this queue) {
    /***** Note! *****
    The following call forms the boundary
    between the Async and Sync layers. */
    wakeup ((caddr_t) &sb->sb_cc);
  }
}
```

当 一个进程陷入睡眠后，进程会和一个“Handle”绑定在一起。要唤醒这个睡眠的进程，wakeup以这个Handle为参数。一个等待事件的线程，一般 使用和这个事件相关的数据结构的地址作为Handle。在我们的例子中，套接口接收队列的地址（sb->sc_cc）作为Handle。

如 果套接口队列上没有等待数据的进程，什么也不会发生。在我们的例子中，如9.1.1的说明，原来的进程阻塞在soreceive调用上。内核将要唤醒这个 进程，它循环检测是否有足够多的数据到达供read使用。如果足够的数据到达，soreceive把数据拷贝到用户的缓冲区，系统调用read将会返回。

对应用户进程，read调用看起来就是同步的。然而，这不过是半同步半异步模式造成的幻觉。特别地，异步进程和上下文的切换，在用户进程睡眠的过程中默默地发生。注意，内核不会阻塞，会一直在干活，总

有一些东西在运行，哪怕是空闲进程行。

9.2 磁盘控制器的例子

这 个例子说明了在BSD UNIX的文件子系统中使用半主动半被动模式的情况。上面的例子说明了这个模式中，数据在网络接口卡输入通过TCP/IP的协议栈一直传递到用户进程。下 面这个例子将说明如何输出数据，数据来自用户的进程，通过了BSD UNIX 的I/O子系统， 最后到达磁盘。
访问UNIX的磁盘类的存储设 备，有两种方式。一种是通过/dev下的块设备文件，另外一种是通过字符设备文件。通过块设备文件访问的时候，要通过一个软件实现的磁盘块缓存区。与此相 反，通过字符设备（叫“raw”I/O）访问的时候，会绕过缓存系统直接进行I/O操作。挂接一个文件系统前，Raw I/O往往用来做完整性检查。一些用户级别的数据库，想自己实现磁盘缓存机制，也会使用raw I/O。

9.2.1 同步调用

如 果一个进程打开一个字符设备文件（例如：/dev/rdk0a），进行一个写动作，当设备驱动真正完成写动作后才会结束。这是半同步的处理部分，多数原始 磁盘设备有一个write动作的执行入口点，入口点作为一个全局I/O的例程存储在全局的cdevsw向量中。如下所示。

```
/* Do a write on a device for a user process. */
int raw_write (dev_t dev, struct uio *uio)
{
   return physio (cdevsw[major(dev)].d_strategy,
               (struct buf *) NULL,
               dev, B_WRITE, minphys, uio);
}
```

这个函数同步调用了physio，physio在用户进程的请求下完成物理I/O动作，物理I/O直接从裸设备写入到用户缓存区，绕过系统的高速缓存。如下实现

```
int physio (int (*strategy)(),
          struct buf *bp,
          dev_t dev,
          int flags,
          u_int (*minphys)(),
          struct uio *uio)
{
   struct iovec *iovp;
   struct proc *p = curproc;
   int error, done, i, nobuf, s, todo;
   /* ... */
   /* read and write, from above */
   flags &= B_READ | B_WRITE;
   bp->b_flags = B_BUSY | B_PHYS | B_RAW | flags;
   /* call driver's strategy to start the transfer */
   (*strategy) (bp);
   /***** Note! *****
    The following call forms the boundary
    between the Sync and Async layers. */
    while ((bp->b_flags & B_DONE) == 0)
       /* Wait for the transfer to complete */
       tsleep ((caddr_t) bp, PRIBIO + 1, "physio", 0);
    /* ... */
}
```

这 个physio例程使用一个用户缓冲区，一个设备和设备的strategy函数指针作为参数。这个例程的任务

是：发起一个读或写操作，然后立即返回。因为 执行用户缓冲去的指针是用户提供的，第一步必须认。一旦确认，就把缓冲区封装到一个buf的结构体中。buf结构体的标志被设置说明其是一个写还是一个读 操作。同时有标志说明了这是一个raw I/O的操作。当buf的结构体设置好，就传递到strategy，这个例程调度I/O操作并返回。下一步，physio睡眠直到I/O操作完成。

9.2.2 异步处理

带缓存的和raw I/O的请求，都是同步的进入设备的驱动，通过strategy函数。

```
void strategy (struct buf *bp)
{
  /* ... */
  s = splbio (); /* protect the queues */
  /* sort the buffer structure into the
  driver's queue (e.g., using disksort()) */
  if (drive is busy) { splx (s); return; }
      /* flow control is here.... if the
      drive is busy the request stays in the queue */
      /* start first request on the queue */
      /* done! */
      splx (s);
      return;
}
```

strategy被设计成通用的函数，大部分设备的I/O都使用这个接口。上面的例子假定驱动一次只处理一个请求。一个设备可能在一个时间处理多个请求，在这种情况下，需要使用多个列表保存哪一个缓存区是激活的，哪一个正在等待I/O。

9.2.3 同步完成阶段。

当磁盘控制器完成操作后，会产生中断。这将触发一个中断处理例程，这个例程联系了同步任务层和异步任务层。如下面的代码

```
int intr (void *v)
{
    struct buf *bp;
    /* get current request into "bp" */
    /***** Note! *****
     The following ties the Async layer back
     into the Sync layer. */
    biodone (bp); /* Wakeup the sleep in physio(). */
    /* start next request on queue */
    return (1); /* done */
}
```

这个中断处理函数，完成中断服务并清除硬件中断。查看驱动的状态表检测一个I/O是不是完成。一个I/O的请求使用异步buf结构描述，一旦boidone函数被调用，将通知高一级的内核软件，write请求已经完成。这会导致进程的tsleep调用返回。

10 变化

使用半同步半异步模式的传统方式是：来自异步任务层的输入使用“pash-driver”的I/O，来自同步任务层的输入使用“pull-driver”的I/O，处理输出使用相反的方式。在一些系统中有下面的变化。

×组合异步通知和同步I/O。当数据在排队层缓存后，通知同步层。UNIX的SIGIO就是实现的这种信号驱动的I/O。在这种情况下，一个信号发给上层的用户进程，然后用户进程使用read读出数据。

×当要求异步处理的时候新产生线程。组合异步通知和同步I/O的另外一种方式，当异步事件发生的时候

产生一个新的线程。

× 在上层任务中使用异步I/O。一些系统扩展了原来的模型，准许把传递数据的通知发给上层任务。UNIX SYSTEM V 的第4版本使用这种方法扩展了信号接口。一个缓冲区指针作为信号处理函数的参数传递。Windows NT 使用重叠I/O的方式实现了相似的结构。这种结构中，一个异步事件中包含一个重叠的IO结构，结构中包含完成事件的标识和对应的数据。

× 底层任务实现同步的I/O。单线程的操作系统（如BSD UNIX）。通常只对上层的任务使用混合的同步/异步模型。在这些系统中，底层的任务必须严格的使用异步I/O。如果使用线程实现上下文切换，系统准许在 内核中使用同步的I/O。在高性能的媒体系统中，使用一个内核线程在固定的时间轮询共享内存特定的域，使用这种轮询的中断机制，可以减少上下文切换。
如果异步任务层拥有自己的控制线程，他可以自动运行把消息通过队列层传递给同步任务层。微内核操作系统一般使用这种设计。在微内核中使用一个独立的进程和用户进程交换数据。

11 Known Uses

×The BSD UNIX networking subsystem [1] and the original System V UNIX STREAMS communication framework [12] use the Half-Sync/Half-Async pattern to structure the concurrent I/O architecture of user processes and the OS kernel. All I/O in these kernels is asynchronous and triggered by interrupts. The Queueing layer is implemented by the Socket layer in BSD and by STREAM heads in System V STREAMS. I/O for user processes is synchronous. Most UNIX applications are developed as user processes that call the synchronous higher-level read/write interfaces. This design shields developers from the complexity of asynchronous OS handled by the kernel. There are provisions for notifications (via the SIGIO signal) that asynchronously trigger synchronous I/O.
×The multi-threaded version of Orbix 1.3 (MT-Orbix) [13] uses several variations of the Half-Sync/Half-Async pattern to dispatch CORBA remote operations in a concurrent server. In the Asynchronous layer of MTOrbix a separate thread is associated with each HANDLE that is connected to a client. Each thread blocks synchronously readingCORBArequests from the client. When a request is received it is formatted and then enqueued at the Queueing layer. An active object thread in the Synchronous layer then wakes up, dequeues the request, and processes it to completion by performing an upcall on the CORBA object implementation.
×The Motorola Iridium system uses the Half-Sync/Half-Async pattern in an application-level Gateway that routes messages between satellites and ground control stations [14]. The Iridium Gateway implements the Half-Sync/Half-Async pattern with the ADAPTIVE Service eXecutive (ASX) framework [15]. The Reactor [6] class category from the ASX framework implements an object-oriented demultiplexing and dispatching mechanism that handles events asynchronously. The ASX Message Queue class implements the Queueing layer, and the ASX Task class implements active objects in the Synchronous task layer.
×The Conduit communication framework [16] from the Choices OS project [17] implements an object-oriented version of the Half-Sync/Half-Async pattern. User processes are synchronous active objects, an Adapter Conduit serves as the Queueing layer, and the Conduit micro-kernel operates asynchronously.

12 Related Patterns

×The Synchronous task layer uses the Active Object pattern[3].（这个模式，我也已经翻译了）
×The Asynchronous task layer may use the Reactor pattern [6] to demultiplex events from multiple sources of events.
×The Queueing layer provides a Facade [18] that simplifies the interface to the Asynchronous task layer of the system.

×The Queueing layer is also a Mediator [18] that coordinates the exchange of data between the Asynchronous and Synchronous task layers.

Acknowledgements

We would like to thank Lorrie Cranor and Paul McKenney for comments and suggestions for improving this paper.

References

[1] S. J. Leffler,M.McKusick,M. Karels, and J.Quarterman, The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley, 1989.

[2] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in Proceedings of the Conference on Computer Communications (INFOCOM),(Boston, MA), pp. 624-633, IEEE, April 1995.

[3] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs, (Monticello, Illinois), pp. 1-7, September 1995.

[4] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An Architecture for ImplementingNetwork Protocols," IEEE Transactions on Software Engineering, vol. 17, pp. 64-76, January 1991.

[5] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming,"in Proceedings of the 1st Conference on Object-Oriented Technologies and Systems, (Monterey, CA),USENIX, June 1995.

[6] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in Pattern Languages of Program Design (J. O.Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[7] P. Druschel and L. L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," in Proceedings of the 14th Symposium on Operating System Principles (SOSP), Dec.1993.

[8] C. Cranor and G. Parulkar, Design of Universal Continuous Media I/O, in Proceedings of the 5th InternationalWorkshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV (Durham, New Hampshire),pp. 83-86, Apr. 1995.

[9] W. R. Stevens, TCP/IP Illustrated, Volume 2. Reading, Massachusetts:AddisonWesley, 1993.

[10] H. Custer, Inside Windows NT. Redmond, Washington: MicrosoftPress, 1993.

[11] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System, IEEE Computer,vol. 23, pp. 23-33,May 1990.

[12] D. Ritchie, A Stream Input-Output System,"AT&TBell Labs Technical Journal, vol. 63, pp. 311-314,Oct. 1984.

[13] C. Horn, "The Orbix Architecture," tech. rep., IONA Technologies,August 1993.

[14] D. C. Schmidt, A Family of Design Patterns for Applicationlevel Gateways," The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages), vol. 2, no. 1, 1996.

[15] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications, Proceedings of the 6th USENIX C++ Technical Conference, (Cambridge, Massachusetts),USENIX Association, April 1994.

[16] J. M. Zweig, "The Conduit: a Communication Abstraction in C++," in Proceedings of the 2nd USENIX C++ Conference,pp. 191-203,USENIX Association, April 1990.

[17] R. Campbell, N. Islam, D. Raila, and P. Madany, "Designing and Implementing Choices: an Object-Oriented System in C++," Communications of the ACM, vol. 36, pp. 117-126,Sept. 1993.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software.Reading,MA: Addison-Wesley, 1995.

原文地址 http://www.cs.wustl.edu/~schmidt/PDF/HS-HA.pdf

发表于: 2007-02-10，修改于: 2007-02-10 21:34，已浏览6448次，有评论1条 推荐 投诉