Half-Sync/Half-Async

An Architectural Pattern for Efficient and Well-structured Concurrent I/O

Douglas C. Schmidt and Charles D. Cranor
schmidt@cs.wustl.edu and chuck@maria.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-7538

An earlier version of this paper appeared in a chapter in the book "Pattern Languages of Program Design 2" ISBN 0-201-89527-7, edited by John Vlissides, Jim Coplien, and Norm Kerth published by Addison-Wesley, 1996.

Abstract

This paper describes the Half-Sync/Half-Async pattern, which integrates synchronous and asynchronous I/O models to support both programming simplicity and execution efficiency in complex concurrent software systems. In this pattern, higher-level tasks use a synchronous I/O model, which simplifies concurrent programming. In contrast, lower-level tasks use an asynchronous I/O model, which enhances execution efficiency. This pattern is widely used in operating systems such as UNIX, Mach, Windows NT, and VMS, as well as other complex concurrent systems.

1 Intent

The Half-Sync/Half-Async pattern decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency.

2 Motivation

To illustrate the Half-Sync/Half-Async pattern, consider the software architecture of the BSD UNIX [1] networking subsystem shown in Figure 1. The BSD UNIX kernel coordinates I/O between asynchronous communication devices (such as network adapters and terminals) and applications running on the OS. Packets arriving on communication devices are delivered to the OS kernel via interrupt handlers initiated asynchronously by hardware interrupts. These handlers receive packets from devices and trigger higher layer protocol processing (such as IP, TCP, and UDP). Valid packets containing application data are queued at the Socket layer. The OS then dispatches any user processes waiting to consume the data. These processes synchronously receive data from the Socket layer using the `read` system call. A user process can make `read` calls at any point; if the data is not available the process will sleep until the data arrives from the network.

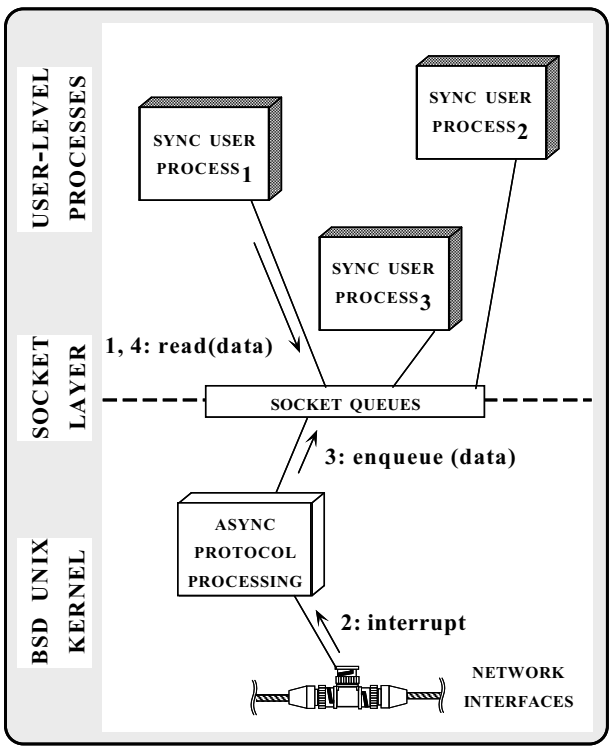

Figure 1: BSD UNIX Software Architecture

In the BSD architecture, the kernel performs I/O asynchronously in response to device interrupts. In contrast, user-level applications perform I/O synchronously. This separation of concerns into a "half synchronous and half asynchronous" concurrent I/O structure resolves the following two forces:

• **Need for programming simplicity:** Programming an asynchronous I/O model can be complex because input and output operations are triggered by interrupts. Asynchrony can cause subtle timing problems and race conditions when the current thread of control is preempted by an interrupt handler. Moreover, interrupt-driven programs require extra data structures in addition to the run-time stack. These data structures are used to save and restore state explicitly

when events occur asynchronously. In addition, debugging asynchronous programs is hard since external events occur at different points of time during program execution.

In contrast, programming applications with a synchronous I/O model is easier because I/O operations occur at well defined points in the processing sequence. Moreover, programs that use synchronous I/O can block awaiting the completion of I/O operations. The use of blocking I/O allows programs to maintain state information and execution history in a run-time stack of activation records, rather than in separate data structures. Thus, there is a strong incentive to use a synchronous I/O model to simplify programming.

• **Need for execution efficiency:** The asynchronous I/O model maps efficiently onto hardware devices that are driven by interrupts. Asynchronous I/O enables communication and computation to proceed simultaneously. In addition, context switching overhead is minimized because the amount of information necessary to maintain program state is relatively small [2]. Thus, there is a strong incentive to use an asynchronous I/O model to improve run-time performance.

In contrast, a completely synchronous I/O model may be inefficient if each source of events (such as network adapter, terminal, or timer) is associated with a separate active object (such as a process or thread). Each of these active objects contain a number of resources (such as a stack and a set of registers) that allow it to block while waiting on its source of events. Thus, this synchronous I/O model increases the time and space required to create, schedule, dispatch, and terminate separate active objects.

3 Solution

To resolve the tension between the need for concurrent programming simplicity and execution efficiency use the *Half-Sync/Half-Async pattern*. This pattern integrates synchronous and asynchronous I/O models in an efficient and well-structured manner. In this pattern, higher-level tasks (such as database queries or file transfers) use a synchronous I/O model, which simplifies concurrent programming. In contrast, lower-level tasks (such as servicing interrupts from network controllers) use an asynchronous I/O model, which enhances execution efficiency. Because there are usually more high-level tasks than low-level tasks in a system, this pattern localizes the complexity of asynchronous processing within a single layer of a software architecture. Communication between tasks in the Synchronous and Asynchronous layers is mediated by a Queueing layer.

4 Applicability

Use the Half-Sync/Half-Async pattern when

- A system possesses the following characteristics:
  - The system must perform tasks in respond to external events that occur asynchronously, and
  - it is inefficient to dedicate a separate thread of control to perform synchronous I/O for each source of external events, and
  - the higher-level tasks in the system can be simplified significantly if I/O is performed synchronously.
- One or more tasks in a system *must* run in a single-thread of control, while other tasks may benefit from multi-threading.
  - For example, legacy libraries like X windows and Sun RPC are often non-reentrant. Therefore, multiple threads of control cannot safely invoke these library functions concurrently. However, to ensure quality of service or to take advantages of multiple CPUs, it may be necessary to perform bulk data transfers or database queries in separate threads. The Half-Sync/Half-Async pattern can be used to decouple the single-threaded portions of an application from the multi-threaded portions. This decoupling enables non-reentrant functions to be used correctly, without requiring changes to existing code.

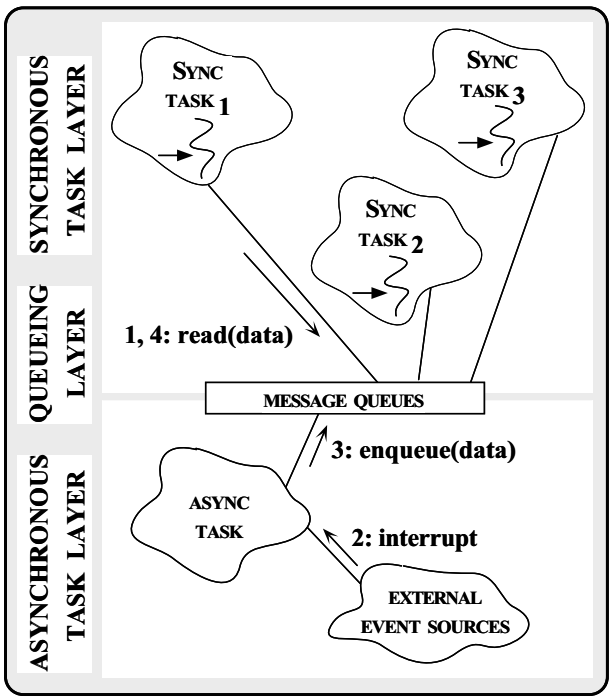

Figure 2: The Structure of Participants in the Half-Sync/Half-Async Pattern

5 Structure and Participants

Figure 2 illustrates the structure of participants in the Half-Sync/Half-Async pattern. These participants are described below.

- **Synchronous task layer** (`User processes`)

– The tasks in this layer perform high-level I/O operations that transfer data synchronously to message queues in the Queueing layer. Unlike the Asynchronous layer, tasks in the Synchronous layer are *active objects* [3] that have their own run-time stack and registers. Therefore, they can block while performing synchronous I/O.

- **Queueing layer** (`Socket layer`)
  – This layer provides a synchronization and buffering point between the Synchronous task layer and the Asynchronous task layer. I/O events processed by asynchronous tasks are buffered in message queues at the Queueing layer for subsequent retrieval by synchronous tasks (and vice versa).
- **Asynchronous task layer** (`BSD UNIX kernel`)
  – The tasks in this layer handle lower-level events from multiple external event sources (such as network interfaces or terminals). Unlike the Synchronous layer, tasks in the Asynchronous layer are *passive objects* that do not have their own run-time stack or registers. Thus, they cannot block indefinitely on any single source of events.
- **External event sources** (`Network interfaces`)
  – External devices (such as network interfaces and disk controllers) generate events that are received and processed by the Asynchronous task layer.

6 Collaborations

Figure 3 illustrates the dynamic collaboration among participants in the Half-Sync/Half-Async pattern when input events arrive at an external event source (output event processing is similar). These collaborations are divided into the following three phases:

- *Async phase* – in this phase external sources of events interact with the Asynchronous task layer via interrupts or asynchronous event notifications.
- *Queueing phase* – in this phase the Queueing layer provides a well-defined synchronization point that buffers messages passed between the Synchronous and Asynchronous task layers in response to input events.
- *Sync phase* – in this phase tasks in the Synchronous layer retrieve messages placed into the Queueing layer by tasks in the Asynchronous layer. Note that the protocol used to determine how data is passed between the Synchronous and Asynchronous task layers is orthogonal to how the Queueing layer mediates communication between the two layers.

The Asynchronous and Synchronous layers in Figure 3 communicate in a "producer/consumer" manner by passing

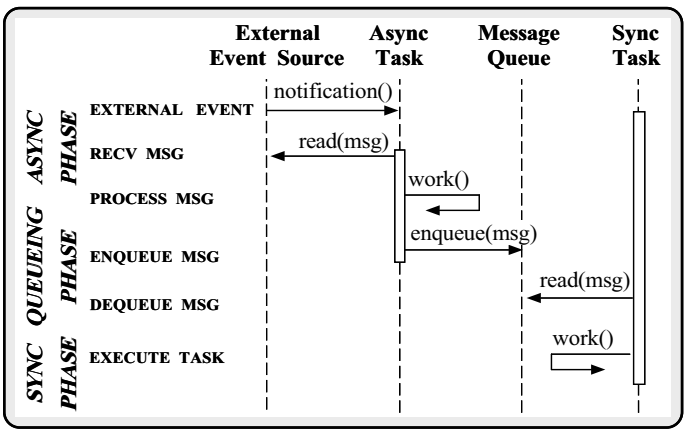

Figure 3: Collaboration between Layers in the Half-Sync/Half-Async Pattern

messages. The key to understanding the pattern is to recognize that Synchronous tasks are active objects. Thus, they can make blocking `read` or `write` calls at any point in accordance with their protocol. If the data is not yet available tasks implemented as active objects can sleep until the data arrives. In contrast, tasks in the Asynchronous layer are passive objects. Thus, they cannot block on `read` calls. Instead, tasks implemented as passive objects are triggered by notifications or interrupts from external sources of events.

7 Consequences

The Half-Sync/Half-Async pattern yields the following benefits:

- *Higher-level tasks are simplified* because they are shielded from lower-level asynchronous I/O. Complex concurrency control, interrupt handling, and timing issues are delegated to the Asynchronous task layer. This layer handles the low-level details (such as interrupt handling) of programming an asynchronous I/O system. The Asynchronous layer also manages the interaction with hardware-specific components (such as DMA, memory management, and device registers).
- *Synchronization policies in each layer are decoupled.* Therefore each layer need not use the same concurrency control strategies. For example, in the single-threaded BSD UNIX kernel the Asynchronous task layer implements concurrency control via low-level mechanisms (such as raising and lowering CPU interrupt levels). In contrast, user processes in the Synchronous task layer implement concurrency control via higher-level synchronization constructs (such as semaphores, message queues, condition variables, and record locks).
- *Inter-layer communication is localized at a single point* because all interaction is mediated by the Queueing layer. The Queueing layer buffers messages passed

between the other two layers. This eliminates the complexity of locking and serialization that would occur if the Synchronous and Asynchronous task layers directly accessed each other's memory.

- *Performance is improved on multi-processors.* The use of synchronous I/O can simplify programming *and* improve performance on multi-processor platforms. For example, long-duration data transfers (such as downloading a large medical image from a database) can be simplified and performed efficiently by using synchronous I/O. One processor can be dedicated to the thread transferring the data, which enables the instruction and data cache of that CPU to be associated with the entire transfer operation.

The Half-Sync/Half-Async pattern has the following drawbacks:

- *A boundary-crossing penalty may be incurred* from synchronization, data copying, and context switching overhead. This overhead typically occurs when data is transferred between the Synchronous and Asynchronous task layer via the Queueing layer. In particular, most operating systems that use the Half-Sync/Half-Async pattern place the Queueing layer at the boundary between the user and kernel protection domains. A significant performance penalty may be incurred when crossing this boundary. For example, the socket layer in BSD UNIX accounts for a large percentage of the overall TCP/IP networking overhead [4].
- *Asynchronous I/O for higher-level tasks is lacking.* Depending on the design of system interfaces, it may not be possible for higher-level tasks to utilize low-level asynchronous I/O devices. Thus, the system I/O structure may prevent applications from utilizing the hardware efficiently, even if external devices support asynchronous overlap of computation and communication.

8 Implementation

This section describes how to implement the Half-Sync/Half-Async pattern by factoring tasks in the system into Synchronous and Asynchronous layers that communicate through a Queueing layer.

8.1 Identify Long-duration Tasks and Implement Them Using Synchronous I/O

Many tasks in a system can be simplified by allowing them to perform synchronous I/O. Often, these are long-duration tasks that transfer large streams of data [5] or perform database queries that may block for prolonged periods awaiting responses from servers.

Implement these long-duration tasks using an active object model [3]. Since active objects have their own run-time stack and registers they can block while performing synchronous I/O. Implementing an active object mechanism requires a method of switching between different threads of control. At the lowest level, this means having a place to *store* the current thread's hardware state (*e.g.,* the values in all its registers, including its stack pointer) and *load* in the state of the new thread. This functionality is sufficient to implement a non-premptive threading mechanism with no memory protection. "User level threads" packages typically provide this type of functionality.

However, more functionality is required to implement active objects as threads and processes in a robust, multi-tasking operating system. In this case, each thread of control has its own address space that is managed by the processor's memory management unit (MMU). When switching between threads the new process's address space info must be loaded into the MMU. Cache flushing may also be required, especially with certain types of virtually addressed cache. In addition to an address space, an OS process often has a "user identification." This tells the operating system what access rights the process has and how much system resources it can consume.

To prevent a single process from taking over the system indefinitely, there must be a way to preempt it. Preemption is generally done with a timer. Periodically (*e.g.,* 1/100 of a second) the timer generates an clock interrupt. During this interrupt the operating systems checks to see if the currently running process needs to be preempted. If so, it saves the process's state and loads the state of the next process to run. When the interrupt returns, the new process will be running.

8.2 Identify Short-duration Tasks and Implement Them Using Asynchronous I/O

Certain tasks in a system cannot block for prolonged periods of time. Often, these tasks run for a short-duration and interact with external sources of events (such as graphical user interfaces or interrupt-driven hardware network interfaces). To increase efficiency and ensure response-time, these sources of events must be serviced rapidly without blocking.

Implement these short-duration tasks using a reactive, passive object model [6]. Passive objects borrow their thread of control from elsewhere (such as the caller or a separate interrupt stack). Therefore, these tasks must use asynchronous I/O since they cannot block for long periods of time. The primary motivation for not blocking is to ensure adequate response time for other system tasks (such as high-priority hardware interrupts like clock timers).

There are several ways to develop a well-structured framework for asynchronous I/O:

- *Demultiplex events using the Reactor pattern* – The Reactor pattern [6] manages a single-threaded event loop that supports the demultiplexing and dispatching of multiple *event handlers*, which are triggered concurrently by multiple events. This pattern combines the simplicity of single-threaded event loops with the extensibility

offered by object-oriented programming. The Reactor pattern serializes event handling within a process or thread and often eliminates the need for more complicated threading, synchronization, or locking.

A Reactor may be implemented to run atop synchronous and/or asynchronous sources of events. The behavior it provides to its event handlers, however, is distinctly asynchronous. Thus, a handler cannot block without disrupting the response time for other sources of events.

- *Implement a multi-level interrupt scheme* – These implementations allow non-time critical processing to be interrupted by higher-priority tasks (such as hardware interrupts) if higher priority events must be handled before the current processing is done. Data structures used by the Asynchronous layer must be protected (*e.g.,* by raising the processor priority or using semaphores) to prevent interrupt handlers from corrupting shared state while they are being accessed.

  For example, in an operating system kernel the need for a multi-level interrupt scheme is strongly influenced by the hardware interrupt service time. If this time can be reduced significantly it may be more efficient to perform all processing at the hardware interrupt level to avoid the overhead of an extra software interrupt. Implementations of TCP/IP have reduced inbound packet protocol processing overhead to the point where the cost of the two-level interrupt scheme dominates the overall packet processing time.

8.3 Implement a Queueing Layer

The Queueing layer provides a synchronization point for buffering messages exchanged by tasks in the Asynchronous and Synchronous layers. The following are several topics that must be addressed when designing the Queueing layer:

- *Concurrency control* – If tasks in the Asynchronous and Synchronous layer execute concurrently (either due to multiple CPUs or hardware interrupts) it is necessary to ensure that concurrent access to shared queue state is serialized to avoid race conditions. Thus, the Queueing layer is typically implemented using concurrency control mechanisms such as semaphores, mutexes, and condition variables. These mechanisms ensure that messages can be inserted and removed to and from the Queueing layer without corrupting internal queue data structures.
- *Layer-to-layer flow control* – Systems cannot devote an unlimited amount of resources to buffer messages in the Queueing layer. Therefore, it is necessary to regulate the amount of data that is passed between the Synchronous and Asynchronous layers. For example, layer-to-layer flow control prevents Synchronous tasks from flooding the Asynchronous layer with more messages than can be transmitted on network interfaces.

  Tasks in the Synchronous layer can block. Therefore, a common flow control policy is to put a task to sleep if it produces and queues more than a certain amount of data. When the Asynchronous task layer drains the queue below a certain level the Synchronous task can be awakened to continue. In contrast, tasks in the Asynchronous layer cannot block. Therefore, if they produce an excessive amount of data a common flow control policy is to have the Queueing layer discard messages. If the messages are associated with a reliable connection-oriented network protocol the sender will eventually timeout and retransmit.
- *Data copying overhead* – Some systems (such as BSD UNIX) place the Queueing layer at the boundary between the user and kernel protection domains. A common way to decouple these protection domains is to copy messages from user to kernel and vice versa. However, this increases system bus and memory load, which may degrade performance significantly when large messages are moved across domains.

  One way to reduce data copying is to allocate a region of memory that is shared between the Synchronous task layer and the Asynchronous task layer [7]. This allows the two layers to exchange data directly, without copying data in the Queueing layer. For example, [8] presents an I/O subsystem that minimizes boundary-crossing penalties by using polled interrupts to improve the handling of continuous media I/O streams. This approach also provides a buffer management system that allows efficient page remapping and shared memory mechanisms to be used between user processes, the kernel, and its devices.

9 Sample Code

This section illustrates examples of the Half-Sync/Half-Async pattern in two different parts of the BSD UNIX operating system [1]. These examples illustrate how the Half-Sync/Half-Async pattern is used by the BSD kernel to enable user processes to operate synchronously, while ensuring the kernel operates asynchronously. The first example illustrates how this pattern is used in the networking subsystem to input data through the TCP/IP protocol stack over Ethernet. The second example illustrates how this pattern is used in the file subsystem to implement interrupt-driven output for disk controllers.

9.1 BSD Networking Subsystem Example

This example illustrates how the Half-Sync/Half-Async pattern is used to structure the synchronous invocation of a `read` system call, asynchronous reception and protocol processing of data arriving on a network interface, and synchronous completion of the `read` call. Figure 1 illustrates the participants and structure of this pattern in BSD UNIX. For a

comprehensive explanation of the BSD UNIX networking subsystem see [9].

9.1.1 Synchronous Invocation

Consider a user process that creates a passive-mode TCP stream socket, accepts a connection, and receives TCP data from the connected socket descriptor. To the user process, the `read` system call on the connection appears to be a synchronous operation, *i.e.,* the process makes the call and the data is returned. However, many steps occur to implement this system call. When the `read` call is issued it traps into the kernel and gets vectored into the network socket code synchronously. The thread of control ends up in the kernel's `soreceive` function, which performs the Half-Sync part of the processing. The `soreceive` function is responsible for transfering the data from the socket queue to the user. It must handle many types of sockets (such as datagram sockets and stream sockets). A simplified view of what `soreceive` does is shown below, with emphasis on the boundary between the Sync and Async layers:

```
/* 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 */
}
```

The code above illustrates the boundary between the synchronous user layer process and the asynchronous kernel layer. Although the user process can sleep while waiting for data, the kernel cannot be suspended because other user processes and devices in the system may require its services.

There are several ways the user's read request is handled by `soreceive`, depending on the characteristics of the socket and the amount of data in the socket queue:

- *Completely synchronous* – If the data requested by the user is in the socket queue it is copied out immediately and the operation completes synchronously.
- *Half-synchronous and half-asynchronous* – If the data requested by the user has not yet arrived the kernel will call the `sbwait` function to put the user process to sleep until the requested data arrives.

Once `sbwait` puts the process to sleep, the OS scheduler will context switch to another process that is ready to run. To the original user process, however, the `read` system call appears to execute synchronously. When packet(s) containing the requested data arrive the kernel will process them asynchronously as described in Section 9.1.2. When enough data has been placed in the socket queue to satisfy the user's request the kernel will wakeup the original process, which completes the `read` system call.

9.1.2 Asynchronous Reception and Protocol Processing

The Half-Async part of the user's read request starts with a packet arriving on a network interface, which causes a hardware interrupt. All inbound packet processing is done in the context of an interrupt handler. It is not possible to sleep during an interrupt because there is no UNIX process context and no separate thread of control. Therefore, an interrupt handler must borrow the caller's thread of control (*i.e.*, its stack and registers). The BSD UNIX kernel uses this strategy to borrow the thread of control from interrupt handlers and from user processes that perform system calls.

Most interrupt-driven computers assign priority levels to the interrupts. For example, on a SPARC there are fifteen interrupt levels with level one being the lowest level and level fifteen being the highest level. Other processors have different levels (*e.g.,* the Motorola 68030 has seven interrupt levels). Under BSD UNIX, processor-specific interrupt levels are assigned machine independent symbolic names called `SPL` levels (the term `SPL` originated in the PDP-11 days of UNIX). For example, the highest network hardware interrupt level is called `SPLIMP`, the clock interrupt is called `SPLCLOCK`, and the highest possible interrupt level is called `SPLHIGH`. For each of these levels there is a corresponding function of the same name that sets the processor interrupt level to that value. Thus, the `splimp` function is called to block out all network hardware level interrupts. All the `spl*` functions will return the previous processor priority level, which represents what the priority should be restored to when the operation completes.

Conventional versions of BSD UNIX use a two-level interrupt scheme to handle packet processing. Hardware critical processing is done at a high priority (`SPLIMP`) and less time critical software processing is done at a lower priority level (`SPLNET`). This two-level interrupt scheme prevents the overhead of software protocol processing from delaying the servicing of other hardware interrupts. The two-level BSD UNIX packet processing scheme is divided into hardware specific processing and protocol processing. When a

packet arrives on a network interface it causes an interrupt at that interface's interrupt priority. All networking interfaces have priority of $\leq$ SPLIMP.

The operating system services the hardware interrupt and then enqueues the packet on the input queue in the protocol layer (such as the IP protocol). A network software interrupt is then scheduled to service that queue at a lower priority (*e.g.,* SPLNET). Once the network hardware interrupt is serviced, the rest of the protocol processing is done at the lower priority level as long as there are no other higher level interrupts pending. The BSD kernel is carefully designed to allow hardware interrupts to occur during a software interrupt without losing data or corrupting buffers.

As an example, consider a host with an AMD LANCE Ethernet NIC chip. The device driver for this chip is called "le" (for "LANCE Ethernet"). On packet arrival the lerint function is called from the interrupt handler. It's job is to acknowledge and clear the interrupt. It then extracts the packet from the network interface and copies it into memory buffers called mbufs, as follows:

```
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 */
  }
}
```

The mbufs are then handed off from lerint to the following Ethernet function called ether_input:

```
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);
}
```

Each network protocol has a packet queue associated with it (*e.g.,* the IP packet queue). The ether_input function first determines which network protocol is being used and puts the packet on the correct queue. It then arranges for a network software level interrupt to occur. This interrupt will occur at the lower priority SPLNET level. At this point the hardware interrupt has been handled and the interrupt service routine exits.

Once the hardware interrupt is done a network software interrupt occurs at the SPLNET level (provided there are no higher level interrupts pending). If the inbound packet is an IP packet the kernel calls the IP interrupt routine (ipintr). IP protocol processing (such as header parsing, packet forwarding, fragmentation, and reassembly) is done in this routine. If the packet is destined for a local process then it is handed off to the transport protocol layer. The transport layer performs additional protocol processing (such as TCP segment reassembly and acknowledgements). Eventually, the transport layer appends the data to the receive socket queue and calls sbwakeup. This call wakes up the original process that was sleeping in soreceive waiting for data on that socket queue. Once this is done, the software interrupt is finished processing the packet.

The following code illustrates the general logic of the thread of control running from ipintr, up through tcp_input, to sowakeup, which forms the boundary between the Async and Sync layers. The first function is ipintr, which handles inbound IP packets:

```
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);
    }
  }
}
```

Since our current example involves a TCP/IP packet, the "protocol switch" inetsw invokes the tcp_input function, which handles an inbound tcp packet:

```
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);
  /* ... */
}
```

The sowakeup function wakes up the user process that was asleep in read waiting for the packet to arrive. As discussed in the following subsection, this function forms the boundary between the Async and Sync layers.

9.1.3 Synchronous Completion

When the data is appended to the socket queue, the `sowakeup` is invoked if a user process is asleep waiting for data to be placed into its buffer.

```
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);
  }
}
```

When a process goes to sleep there is a "handle" associated with that process. To wake up a sleeping process the `wakeup` call is invoked on that handle. A process waiting for an event will typically use the address of the data structure related to that event as its handle. In the current example, the address of the socket receive queue (`sb->sc_cc`) is used as a handle.

If there are no processes waiting for data on a socket queue nothing interesting will happen. However, in the example shown in Section 9.1.1, the original process was sleeping in `soreceive` waiting for data. The kernel will wake up this process in the `soreceive` function, which loops back to check if enough data has arrived to satisfy the `read` request. If all the data requested by the user has arrived `soreceive` will copy the data to the user's buffer and the system call will return.

To the user process the `read` call appeared to be synchronous. However, this was an illusion supported by the Half-Sync/Half-Async pattern. In particular, asynchronous processing and context switching were performed while the process was sleeping. Note that the kernel never blocks and is always doing something, even if that something is running an "idle" process.

9.2 Disk Controller Example

This example illustrates another example of the Half-Sync/Half-Async pattern in the context of the BSD UNIX file subsystem. The previous example illustrates how the pattern is used to *input* data from the Ethernet interface, through the TCP/IP protocol stack, and up to a user process. This example illustrates how the pattern is used to *output* data from a user process, through the BSD UNIX raw I/O subsystem, to a disk.

There are two ways to access UNIX storage devices such as disks. One is through their block-special devices in `/dev`, the other is through their character-special devices. Accesses through the block-special devices go through a layer of software that buffers disk blocks. This buffering takes advantage of the locality of data references. In contrast, access through the character-special device (called "raw" I/O) bypasses the buffering system and directly accesses the disk for each I/O operation. Raw I/O is useful for checking the integrity of a filesystem before mounting it, or for user-level databases that have their own buffering schemes.

9.2.1 Synchronous Invocation

If a process does an `open` on a character-special file (*e.g.*, `/dev/rdk0a`) and then does a `write`, the thread of control will end up in the device driver's `write` entry point. This performs the Half-Sync part of the processing. Most raw disk devices have a `write` entry point that references a global raw I/O routine stored in the `cdevsw` vector. The following illustrates this entry:

```
/* 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);
}
```

This entry point is a synchronous redirect into `physio`, which is a routine that does physical I/O on behalf of a user process. Physical I/O writes directly from the raw device to user buffers, bypassing the buffer cache. The `physio` routine is implemented as follows:

```
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);

  /* ... */
}
```

The `physio` routine is given a user buffer, a device, and that device's `strategy` routine. The strategy routine's job is to initiate a read or write operation on a buffer and return immediately. Because the pointer to the user's buffer is provided by the user process, `physio` must first validate the buffer's address. Once the buffer has been validated it is encapsulated in a `buf` structure. The flags in the `buf` structure are set to indicate if this is a read or a write operation.

The flags are also set to indicate that this is a raw I/O operation. Once the `buf` structure is set up, it is passed to the device-specific `strategy` routine. The `strategy` routine schedules the I/O operation and returns. Next, `physio` sleeps until the I/O operation is done.

9.2.2 Asynchronous Processing

Both buffered and raw I/O requests enter the device driver synchronously via the device's `strategy` routine:

```
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;
}
```

The `strategy` routine is designed to be general so that most device I/O can be routed through this interface (the exception being some `ioctl` calls that perform control operations on a device such as formatting a cylinder on a disk). The bookkeeping information required to store state information during the asynchronous I/O is stored in a data structure accessible to the driver. The example above assumes that the driver only handles one request at a time. It is possible to have a device that handles multiple requests at a time. In that case, multiple lists would keep track of which buffers are active and which are waiting for I/O.

9.2.3 Synchronous Completion

A hardware interrupt is generated by disk controller when the write request completes. This triggers an interrupt routine that ties the Asynchronous task layer back into the Synchronous task layer, as follows:

```
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 */
}
```

The interrupt function services and clears the hardware interrupt. This involves looking in the driver's state table to determine which I/O request has completed. The I/O request is represented by a `buf` structure. Once the `buf` structure has been identified the `biodone` function is called to signal the higher level kernel software that the `write` request is complete. This causes the sleeping process to return from `tsleep`. The interrupt function must also start any queue'd `write` requests if necessary.

10 Variations

The conventional form of the Half-Sync/Half-Async pattern for input uses "push-driven" I/O from the Asynchronous task layer to the Queueing layer and "pull-driven" I/O from the Synchronous task layer to the Queueing layer. These roles are reversed for output. The following variations appear in some systems:

- *Combining asynchronous notification with synchronous I/O* – it is possible for the Synchronous task layer to be notified asynchronously when data is buffered at the Queueing layer. This is how signal-driven I/O is implemented by the UNIX SIGIO mechanism. In this case, a signal is used to "push" a notification to higher-level user processes. These processes then use `read` to "pull" the data synchronously from the queueing layer.
- *Spawning synchronous threads on-demand from asynchronous handlers* – Another way to combine asynchronous notification with synchronous I/O is to spawn a thread on-demand when an asynchronous event occurs. I/O is then performed synchronously in the new thread. This approach ensures that the resources devoted to I/O tasks are a function of the number of work requests being processed in the system.
- *Providing asynchronous I/O to higher-level tasks* – Some systems extend the preceding model still further by allowing notifications to push data along to the higher-level tasks. This approach is used in the extended signal interface for UNIX System V Release 4. In this case, a buffer pointer is passed along with the signal handler function. Windows NT supports a similar scheme using overlapped I/O and I/O completion ports [10]. In this case, when an asynchronous event completes an overlapped I/O structure contains an indication of the event that completed, along with the associated data.
- *Providing synchronous I/O to lower-level tasks* – Single-threaded operating systems (such as BSD UNIX) usually support a hybrid synchronous/asynchronous I/O model only for higher-level application tasks. In these systems, lower-level kernel tasks are restricted to asynchronous I/O. Multi-threaded systems permit synchronous I/O operations in the kernel if multiple wait contexts are supported via threads. This is useful for implementing polled interrupts, which reduce the amount of context

switching for high-performance continuous media systems by dedicating a kernel thread to poll a field in shared memory at regular intervals [8].

If the Asynchronous task layer possesses its own thread of control it can run autonomously and use the Queueing layer to pass messages to the Synchronous task layer. Micro-kernel operating systems typically use this design. The micro-kernel runs as a separate process that exchanges messages with user processes [11].

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 MT-Orbix a separate thread is associated with each HANDLE that is connected to a client. Each thread blocks synchronously reading CORBA requests 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].
- 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 2^{nd} 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 Implementing Network 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 1^{st} 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 14^{th} 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 International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '95)*, (Durham, New Hampshire), pp. 83–86, Apr. 1995.

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

[10] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 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&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

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

[14] D. C. Schmidt, "A Family of Design Patterns for Application-level 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," in *Proceedings of the* 6^{th} *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* 2^{nd} *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.