Half-Sync/Half-Async

An Architectural Pattern for Efficient and Well-structured Synchronous and Asynchronous Processing

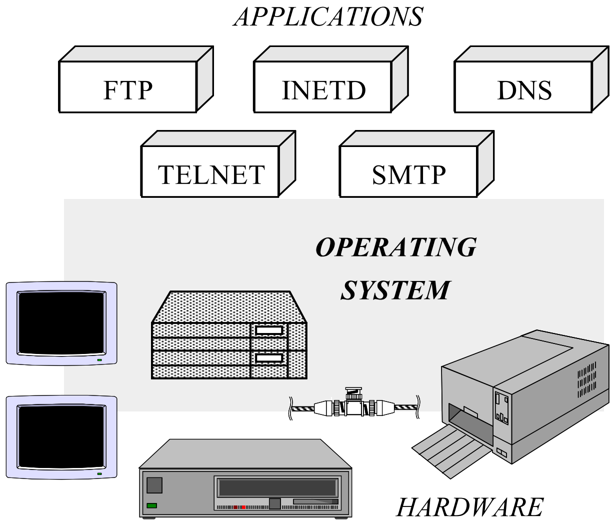
Douglas C. Schmidt and Charles D. Cranor
schmidt@cs.wustl.edu and chuck@maria.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-7538

1 Intent

The Half-Sync/Half-Async architecture pattern decouples synchronous tasks from asynchronous tasks in complex concurrent systems in order to simplify programming effort without degrading quality of service.

2 Example

An operating system is a widely-used example of a complex concurrent system that manages many different types of applications and hardware. For instance, the following figure illustrates how an operating system coordinates and demultiplexes the communication between standard Internet networking applications, such as TELNET, FTP, SMTP, DNS,



and INETD, and hardware I/O devices, such as network interfaces, disk controllers, end-user terminals, and printers. The operating system is responsible for managing the scheduling, communication, protection, and memory allocation of all its concurrently executing applications and hardware devices.

One way to implement a highly efficient operating system is to drive it entirely by asynchronous interrupts. This design is efficient because it maps directly onto the event notification mechanism used by most hardware devices. For instance, packets arriving on network interfaces can be delivered asynchronously to the operating system by hardware interrupts, processed asynchronously by higher-layer protocols, such as TCP/IP, and finally consumed by applications, such as TELNET or FTP, which can be notified by asynchronous signals, such as `SIGIO` [1].

Experience has shown, however, that it is hard to develop complex concurrent systems that are based entirely on asynchronous processing. In particular, the effort required to program, validate, debug, and maintain asynchronous applications can be prohibitively expensive, tedious, and error-prone. Therefore, many software developers prefer to use synchronous programming techniques, such as multi-threading or multi-processing, which are easier to learn and understand.

Unfortunately, basing complex concurrent systems entirely on synchronous processing often makes it hard to meet stringent quality of service requirements. This problem stems from the additional time and space overhead associated with implementations of synchronous multi-threading or multi-processing techniques. These synchronous techniques are typically less efficient because they add more abstraction layers between asynchronously triggered hardware devices, operating system services, and higher-level applications.

Thus, a key challenge facing developers of complex concurrent systems is how to structure asynchronous and synchronous processing in order to reconcile the need for programming simplicity with the need for high quality of service. There is a strong incentive to use a synchronous processing model to simplify application programming. Likewise, there is a strong incentive to use an asynchrony to improve quality of service.

3 Context

A complex concurrent system that performs both asynchronous and synchronous processing.

4 Problem

Complex concurrent systems are often characterized by a mixture of asynchronous and synchronous tasks that must be processed at various levels of abstraction. These types of systems require the simultaneous resolution of the following two *forces*:

The need for programming simplicity: Certain tasks in complex concurrent systems must be processed asynchronously in order to satisfy quality of service requirements. For instance, protocol processing within an operating system kernel is typically asynchronous because I/O devices are driven by interrupts that are triggered by the network interface hardware. If these asynchronous interrupts are not handled immediately the hardware may function incorrectly by dropping packets or corrupting hardware registers and memory buffers.

It is hard, however, to write complex programs where all tasks are triggered entirely by asynchronous interrupts. In particular, asynchrony can cause subtle timing problems and race conditions when a thread of control is preempted by an interrupt handler. Moreover, in addition to a run-time stack, asynchronous programs often require data structures that contain additional state information. When interrupts occur, programmers must save and restore these data structures explicitly, which is tedious and error-prone. In addition, it is hard to debug asynchronous programs because interrupts can occur at different, often non-repeatable, points of time during program execution.

In contrast, it may be easier to write synchronous programs because tasks can be constrainted to occur at well-defined points in the processing sequence. For instance, programs that use synchronous `read` and `write` calls can block awaiting the completion of I/O operations. Blocking I/O operations allow programs to maintain state information and execution history in their run-time activation record stack, rather than in separate data structures that must be managed by programmers explicitly.

The need for high quality of service: Although synchronous programs are generally easier to write, there are circumstances where quality of service requirements cannot be achieved if many or all tasks are processed synchronously within separate threads of control. The problem stems from the fact that each thread contains resources, such as a run-time stack, a set of registers, and thread-specific storage, that must be created, stored, retrieved, synchronized, and destroyed by a thread manager. A non-trivial amount of time and space may be required to manage threads. For example, if an operating system associates a separate thread to process each hardware I/O device synchronously within the kernel, the thread management overhead can be excessive.

In contrast, it may be more efficient to write asynchronous programs because tasks can be mapped directly onto hardware or software interrupt handlers. For instance, asynchronous I/O based on interrupt-driven DMA enables communication and computation to proceed simultaneously and efficiently. Likewise, interrupt-driven asynchronous systems may incur less context switching overhead [2] than synchronous threaded systems because the amount of information necessary to maintain program state can be reduced.

Although the need for both programming simplicity *and* high quality of service may appear to conflict, it is essential that both these forces be resolved effectively in complex concurrent systems.

5 Solution

Decompose the tasks in the system into three layers: *synchronous*, *asynchronous*, and *queueing*. Process higher-layer tasks, such as database queries or file transfers, *synchronously* in separate threads or processes in order to simplify concurrent programming. Conversely, process lower-layer tasks, such as servicing interrupts from network interfaces, *asynchronously* in order to enhance quality of service. Communication between tasks residing in the separate synchronous and asynchronous layers should be mediated by a *queueing* layer.

6 Structure

The participants in the Half-Sync/Half-Async pattern include the following:

Synchronous task layer:

- This layer performs high-level processing tasks. Tasks in the synchronous layer run in separate threads or processes that have their own run-time stack and registers. Therefore, they can block while performing synchronous operations, such as I/O. For instance, in an operating system, application processes can use `read` and `write` system calls to perform I/O synchronously for their high-level tasks.

Asynchronous task layer:

- This layer performs lower-level processing tasks, which typically emanate from multiple external I/O sources. Tasks in the asynchronous layer do not have their own dedicated run-time stack or registers. Thus, they cannot block indefinitely while performing asynchronous operations. For instance, I/O devices and protocol processing in operating system kernels typically perform their tasks asynchronously, often in interrupt handlers.

Queueing layer:

- This layer provides a buffering point between the synchronous task layer and the asynchronous task layer. Messages produced by asynchronous tasks are buffered at the queueing layer for subsequent retrieval by synchronous tasks and vice versa. In addition, the queueing layer is responsible for notifying tasks in one layer when messages are passed to them from the other layer. For instance, operating systems typically provide a socket layer that serves as the buffering and notification point between the synchronous application processes and the asynchronous I/O hardware devices in the kernel.

External I/O sources:

- External I/O sources generate events that are received and processed by the asynchronous task layer. For example, network interfaces, disk controllers, and end-user terminals are common sources of external I/O events for operating systems.

Figure 1 illustrates the structure and relationships between these participants.

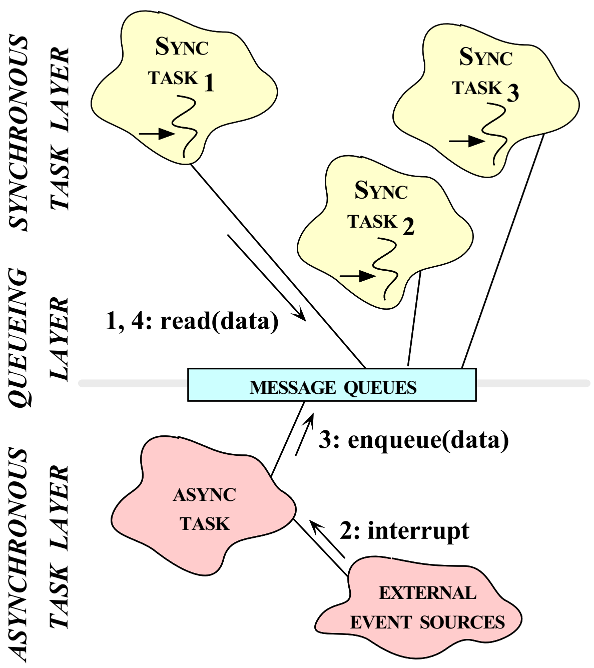


Figure 1: The Structure of Participants in the Half-Sync/Half-Async Pattern

7 Dynamics

Asynchronous and synchronous layers in the Half-Sync/Half-Async pattern interact in a "producer/consumer" manner by exchanging messages via a queueing layer. Below, we describe three phases of interactions that occur when input arrives "bottom-up" from external I/O sources, the collaborations between components in the Half-Sync/Half-Async model occurs. The interactions between layers and components is similar when output arrives "top-down" from tasks in the synchronous layer.

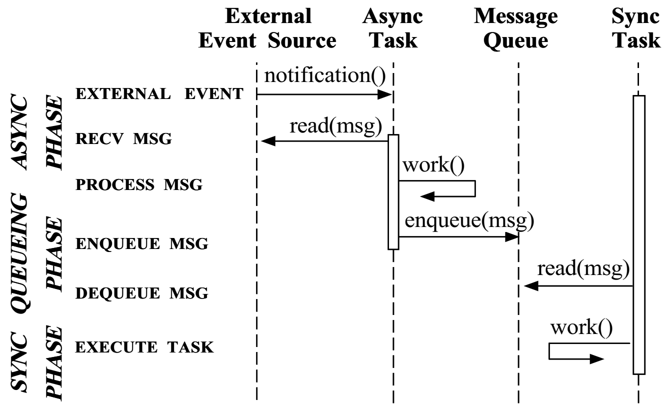


Figure 2: Collaboration between Layers in the Half-Sync/Half-Async Pattern

1. Asynchronous phase: In this phase, external sources of input interact with the asynchronous task layer via interrupts or asynchronous event notifications. When asynchronous tasks are finished processing their input they can communicate to the designated tasks in the synchronous layer via the queueing layer.

2. Queueing phase: In this phase, the queueing layer buffers input passed from the asynchronous layer to the synchronous layer and notifies the synchronous layer that the input is available.

3. Synchronous phase: In this phase, tasks in the synchronous layer retrieve and process input placed into the queueing layer by tasks in the asynchronous layer.

Figure 2 illustrates these collaborations among participants in the Half-Sync/Half-Async pattern.

8 Implementation

This section describes how to implement the Half-Sync/Half-Async pattern by factoring out tasks in a complex concurrent system into synchronous and asynchronous layers that communicate solely through a queueing layer. The following steps should be followed when implementing this pattern.

1. Decompose the Overall System into Three Layers: Synchronous, Asynchronous, and Queueing

The following criteria can be used to determine where various tasks are configured into a system architecture designed according to the Half-Sync/Half-Async pattern.

Identify higher-level or long-duration tasks and configure them into the synchronous layer: Many tasks in a complex concurrent system are easier to implement using synchronous processing. Often, these tasks perform relatively high-level or long-duration application processing, such as transferring large streams of data [3] or performing database queries. Tasks in the synchronous layer may block for prolonged periods awaiting responses in accordance with application protocols. If the data is not yet available, these synchronous tasks can block on the queueing layer until the data arrives.

Identify lower-level or short-duration tasks and configure them into the asynchronous layer: Other tasks in a system cannot block for prolonged periods of time. Often, these tasks perform lower-level or short-duration system processing that interacts 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. Instead, these tasks are triggered by asynchronous notifications or interrupts from external I/O sources and run to completion, at which point they can insert their results into the queueing layer.

Identify inter-layer communication tasks and configure them into the queueing layer: The queueing layer is a mediator that exchanges messages between tasks in the asynchronous and synchronous layers. The protocols used by tasks in the synchronous and asynchronous layers to exchange messages is orthogonal to the tasks performed by the queueing layer in order to mediate communication between the other two layers. Tasks associated with the queueing layer include buffering and notification, layer-to-layer flow control, and data copying.

2. Implement Each Layer Identified in Step 1

Implement the synchronous layer: A straightforward way to implement the higher-level or long-duration tasks is to use the Active Object pattern [4]. The Active Object pattern decouples method execution from method invocation to simplify synchronized access to a shared resource by methods invoked in different threads of control. Thus, active objects can block, such as when they perform synchronous I/O, because they have their own run-time stack and registers.

Implement the asynchronous layer: These lower-level or shorter-duration tasks can be implemented using *passive objects*, which borrow their thread of control from elsewhere, such as the calling thread or a separate interrupt stack. Therefore, these tasks must run asynchronously because they cannot block for long periods of time in order to ensure adequate response time for other system tasks, such as high-priority hardware interrupts. There are several ways to process asynchronous tasks:

- **Demultiplex events from external sources using the Reactor or Proactor patterns:** The Reactor [5] pattern is responsible for demultiplexing and dispatching of multiple event handlers that are triggered when it is possible to initiate an operation without blocking. The Proactor [6] pattern supports the demultiplexing and dispatching of multiple event handlers that are triggered by the completion of asynchronous operations. The behavior of both patterns is similar, however, in that a handler cannot block for a long-duration without disrupting the processing of other event sources.

- **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. To prevent interrupt handlers from corrupting shared state while they are being accessed, data structures used by the asynchronous layer must be protected, *e.g.*, by raising the processor priority or by using semaphores.

Implement the queueing layer: For input processing, messages are typically "pushed" from the asynchronous task layer to the queueing layer, where the synchronous task layer then "pulls" the messages from the queueing layer. These roles are reversed for output processing. The following topics must be addressed when implementing the queueing layer.

- **Define the queueing layer's buffering and notification mechanisms:** The presence of multiple CPUs, preemptive multi-threading, or asynchronous hardware interrupts can cause tasks in the asynchronous and synchronous layer to execute simultaneously. Therefore, it is necessary to ensure that concurrent access to internal queue message buffers is serialized to avoid race conditions. In addition, it is necessary to notify a task in one layer when messages are passed to it from another layer.

  The buffering and notification mechanisms provided by the queueing layer are typically implemented using synchronization primitives, such as semaphores, mutexes, and condition variables. These mechanisms ensure that messages can be inserted and removed to and from the queueing layer's message buffers without corrupting internal data structures. Likewise, these synchronization mechanisms also can be used to notify the appropriate tasks when data has arrived for them in the queueing layer.

  The `Message_Queue` component in the Monitor [**?**] or Active Object [4] patterns illustrates various ways to implement a queueing layer.

- **Define a layer-to-layer flow control mechanism:** 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.

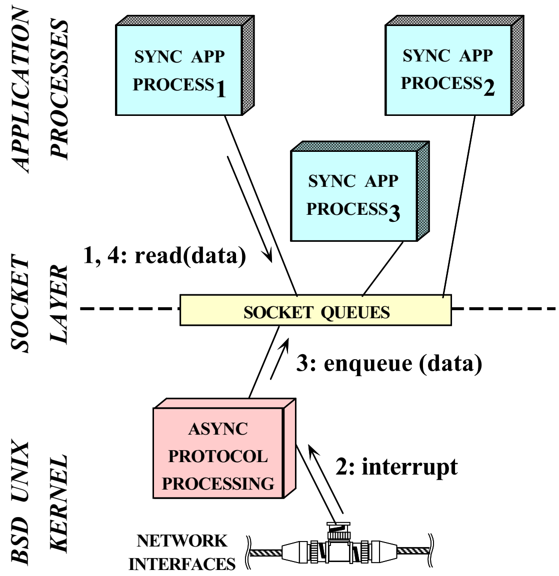
  Tasks in the synchronous layer can block. Therefore, a common flow control policy is to put a task to sleep if it produces and queues more than a certain amount of data. When the asynchronous task layer drains the queue below a certain level the synchronous task can be awakened to continue. In contrast, 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.

• **Minimize data copying overhead:** Some systems, such as BSD UNIX, place the queueing layer at the boundary between user-level and kernel-level 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 design 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 application processes, the kernel, and its devices.

9 Example Resolved

The BSD UNIX operating system [9, 10] demultiplexes and coordinates communication between application processes and I/O device hardware controlled by the BSD UNIX kernel. In the following example, we illustrate how the BSD UNIX applies the Half-Sync/Half-Async pattern to receive data through its TCP/IP protocol stack over Ethernet. We focus on 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, which are shown in the following figure:



9.1 Synchronous Invocation

To reduce the complexity of network programming, application processes commonly perform I/O *synchronously* via `read` and `write` operations. For instance, an application process can receive its data from the socket layer synchronously using the `read` system call. Moreover, an application can make `read` calls at any point during its execution. If the data is not available yet, the operating system kernel can put the process to sleep until the data arrives from the network, thereby allowing other application processes to proceed concurrently.

For instance, consider an application process that receives TCP data from the connected socket handle. To the application process, the `read` system call on the connection appears to be a synchronous operation, *i.e.,* the process invokes `read` and the data is subsequently returned. Many steps occur to implement this system call, however, as we describe below.

When a `read` call is issued it traps into the kernel, which vectors it synchronously into the network socket code. The thread of control ends up in the kernel's `soreceive` function, which handles many types of sockets, such as datagram sockets and stream sockets, and transfers the data from the socket queue to the user. In addition, this function performs the "Half-Sync" part of the BSD UNIX operating system processing.

A simplified description of `soreceive` is shown below:

```
/* Receive data from a socket after being invoked
   synchronously on behalf of an application
   process's read() system call. */

int soreceive ( ... )
{
  for (;;) {
    if (not enough data to satisfy read request) {
      /***** Note! *****
        The following call forms the boundary
        between the Sync and Async layers. */
      sbwait (...); /* wait for data */
    }
    else
      break;
  }

  /* copy data to user's buffer at normal priority */
  uiomove (...);

  return (error code);  /* returns 0 if no error */
}
```

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

There are two ways the user's `read` operation can be handled by `soreceive`, depending on the characteristics of the socket and the amount of data in the socket queue:

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

Once `sbwait` puts the process to sleep, the operating system scheduler will context switch to another process that is ready to run. To the original application 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 9.2. 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.

9.2 Asynchronous Reception and Protocol Processing

To maximize performance within the BSD UNIX kernel, all processing tasks are executed *asynchronously* because I/O devices are driven by hardware interrupts. For instance, packets arriving on network interfaces are delivered to the kernel via interrupt handlers initiated asynchronously by the hardware. These handlers receive packets from devices and trigger subsequent asynchronous processing of higher-layer protocols, such as IP, TCP, or UDP. Valid packets containing application data are ultimately queued at the socket layer, where the BSD UNIX kernel schedules and dispatches application processes waiting to produce or consume the data.

For instance, the "half-async" processing associated with an application's `read` system call starts with a packet arriving on a network interface, which causes a hardware interrupt. All incoming packet processing is done in the context of an interrupt handler. It is not possible to sleep in the BSD UNIX kernel during an interrupt because there is no application 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 application processes that perform system calls.

Conventional versions of BSD UNIX use a two-level interrupt scheme to handle packet processing. Hardware critical processing is done at a high priority and less time critical software processing is done at a lower priority level. 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 *software protocol processing*. When a packet arrives on a network interface it causes an interrupt at that interface's interrupt priority. 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. 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.

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 flow of control in the BSD UNIX kernel, starting with `ipintr`, up through `tcp_input`, to `sowakeup`, which forms the boundary between the asynchronous and synchronous 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 (;;) {
    IF_DEQUEUE (&ipintrq, m); /* dequeue next packet */
    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(),
         to pass the packet up to the transport
         protocol. */
      (*inetsw[ip_protox[ip->ip_p]].pr_input)
        (m, hlen);
    }
  }
}
```

When a TCP/IP packet is received by `ipintr`, the `inetsw` "protocol switch" invokes the `tcp_input` function, which processes inbound TCP packets:

```
int tcp_input (m, iphlen)
{
  /* Much complicated protocol processing omitted... */

  /* We come here to pass data up to the application. */
  sbappend (&so->so_rcv, m);
  sowakeup((so), &(so)->so_rcv);
  /* ... */
}
```

The `sowakeup` function wakes up the application process that was asleep in `read` waiting for data to arrive from a TCP connection. As discussed below, this function forms the boundary between the asynchronous and synchronous layers.

9.3 Synchronous Completion

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

```
void sowakeup (so, sb)
{
  /* ... */
  if (an application 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, 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` operation. If all the data requested by the application has arrived `soreceive` will copy the data to the user's buffer and the system call will return.

To the application 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.

10 Variants

The following are common variations of the Half-Sync/Half-Async pattern:

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 [1]. In this case, a signal is used to "push" a notification to higher-level application 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 notifications with synchronous I/O is to spawn a thread on-demand when an asynchronous operation is invoked. 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 (SVR4). 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 [11]. In this case, when an asynchronous operation completes an overlapped I/O structure contains an indication of the operation that completed, along with its associated data. The Proactor pattern [6] describes how to structure applications to take advantage of asynchronous operations and overlapped I/O.

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 application processes [12].

11 Known Uses

UNIX Networking Subsystems: The BSD UNIX networking subsystem [9] and the original System V UNIX STREAMS communication framework [13] use the Half-Sync/Half-Async pattern to structure the concurrent I/O architecture of application processes and the operating system 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 application processes is synchronous. Most UNIX applications are developed as application processes that call the synchronous higher-level `read`/`write` interfaces. This design shields developers from the complexity of asynchronous I/O handled by the kernel. There are provisions for notifications (via the SIGIO signal) that asynchronously trigger synchronous I/O.

CORBA ORBs: The multi-threaded version of Orbix (MT-Orbix) [14] 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.

ACE: The ACE framework [15] uses the Half-Sync/Half-Async pattern in an application-level Gateway [16] that routes messages between peers in a distributed system. A `Reactor` [5] is used to implement an object-oriented demultiplexing and dispatching mechanism that handles indication events asynchronously, the ACE `Message Queue` class implements the Queueing layer, and the ACE `Task` class implements active objects in the Synchronous task layer.

Conduit: The Conduit communication framework [17] from the Choices operating system project [18] implements an object-oriented version of the Half-Sync/Half-Async pattern. Application processes are synchronous active objects, an Adapter Conduit serves as the Queueing layer, and the Conduit micro-kernel operates asynchronously.

12 See Also

The Half-Sync/Half-Async pattern can be viewed as a composite pattern that combines Reactor/Proactor patterns with the Active Object pattern. The synchronous task layer can use the Active Object pattern [4]. The Active Object pattern decouples method execution from method invocation to simplify synchronized access to a shared resource by methods invoked in different threads of control.

The asynchronous task layer may use the Reactor pattern [5] or Proactor pattern [6] to demultiplex events from multiple sources of events. The Reactor and Proactor patterns are responsible for demultiplexing and dispatching event and completion handlers, respectively. These pattern can be used in conjunction with the Active Object pattern to form the Half-Sync/Half-Async pattern.

13 Consequences

The Half-Sync/Half-Async pattern has the following **benefits**:

Higher-level synchronous processing tasks are simplified, without degrading overall system performance: In complex concurrent systems, there are often many more high-layer processing tasks than low-layer tasks. Therefore, decoupling higher-layer synchronous tasks from lower-level asynchronous processing tasks simplifies the overall system because complex concurrency control, interrupt handling, and timing tasks can be localized within the asynchronous task layer. The asynchronous layer also handles low-level details, such as interrupt handling, that are harder for application developers to program. In addition, the asynchronous layer can manage the interaction with hardware-specific components, such as DMA, memory management, and I/O 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 synchronization via low-level mechanisms, such as raising and lowering CPU interrupt levels. In contrast, application processes in the synchronous task layer implement synchronization via higher-level mechanisms, such as monitors and message queues.

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 of synchronization policies in each layer enables non-reentrant functions to be used correctly, without requiring changes to existing code.

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 accessed each other's memory directly.

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 [3], 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.

However, the Half-Sync/Half-Async pattern also has the following **liabilities**:

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-level and kernel-level 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 [19].

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.

Acknowledgements

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

References

[1] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[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] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[4] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[5] 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.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[6] I. Pyarali, T. H. Harrison, D. C. Schmidt, and T. D. Jordan, "Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events," in *Pattern Languages of Program Design* (B. Foote, N. Harrison, and H. Rohnert, eds.), Reading, MA: Addison-Wesley, 1999.

[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] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

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

[11] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

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

[13] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[14] S. Baker, *CORBA Distributed Objects using Orbix*. Addison Wesley Longman, 1997.

[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] 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.

[17] 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.

[18] 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.

[19] 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.