Chapter 2

Internet protocols' BSD software implementation

2.1 Introduction

This chapter's objective is to understand the influence that operating system design and implementation techniques have over the performance of the Internet protocols' BSD software implementation. Later chapters discuss how to apply this knowledge for building a performance model of a personal computer-based software router.

The chapter is organized as follows. The first three sections set the conceptual framework for the document but may be skipped by a reader familiar with BSD's networking subsystem. Section 2.2 presents a brief overview on BSD's interprocess communication facility. Section 2.3 presents BSD's networking architecture. And BSD's software interrupt mechanism is presented in section 2.4. Following sections present the chief features, components and structures of the Internet protocol's BSD software implementation. These sections present several ideas and diagrams that are referenced in latter document's sections and should not be skipped. Section 2.5 presents the software implementation while section 2.6 the run-time environment. Finally, section 2.7 presents brief descriptions of other system's networking architectures and section 2.8 summarizes.

2.2 Interprocess communication in the BSD operating system

The BSD operating system [McKusick et al. 1996] is a flavor of UNIX [Ritchie and Thompson 1978] and historically, UNIX systems were weak in the area of interprocess communication [Wright and Stevens 1995]. Before the 4.2 release of the BSD operating system, the only standard interprocess communication facility was the pipe. The requirements of the Internet [Stevens 1994] research community resulted in a significant effort to address the lack of a comprehensive set of interprocess communication facilities in UNIX. (At the time 4.2BSD was being designed there was no global Internet but an experimental computer network sponsored by the United States of America's Defense Advanced Research Projects Agency. Consequently, this computer network was known as the DARPANET.)

2.2.1 BSD's interprocess communication model

4.2BSD's interprocess communication facility was designed to provide a sufficiently general interface upon which distributed-computing applications—sharing of physically distributed computing resources, distributed parallel computing, computer supported telecommunications—could be constructed independently of the underlying communication protocols. This facility has outlasted and is present in the current 4.4 release. For now on, when referring to the BSD operating system, we mean the 4.2 release or any follow-on release like the current 4.4. While designing the interprocess-communication facilities that would support these goals, the developers identified the following requirements and developed unifying concepts for each:

- The system must support communication networks that use different sets of protocols, different naming conventions, different hardware, and so on. The notion of a ***communication domain*** was defined for these reasons.
- A unified abstraction for an endpoint of communication is needed that can be manipulated with a file descriptor. The ***socket*** is the abstract object from which messages are sent and received.
- The semantic aspects of communication must be made available to applications in a controlled and uniform way. So, all *sockets* are typed according to their communication semantics.
- Processes must be able to locate endpoints of communication so that they can rendezvous without being related. Hence, *sockets* can be named.

Figure 2.1 depicts the OMT Object Model [Rumbaugh et al. 1991] for these requirements.

2.2.2 Typical use of *sockets*

First, a *socket* must be created with the `socket` system call, which returns a file descriptor that is then used in later *socket* operations:

```
s = socket(domain, type, protocol);
int s, domain, type, protocol;
```

After a *socket* has been created, the next step depends on the type of *socket* being used. The most commonly used type of *socket* requires a ***connection*** before it can be used. The creation of a *connection* between two *sockets* usually requires that each *socket* have an address (name) bound to it. The address to be bound to a *socket* must be formulated in a ***socket address structure***.

```
error = bind(s, addr, addrlen);
int error, s, addrlen;
struct sockaddr *addr;
```

A *connection* is initiated with a `connect` system call:

```
error = connect(s, peeraddr, peeraddrlen);
int error, s, peeraddrlen;
struct sokaddr *peeraddr;
```

When a *socket* is to be used to wait for connection-requests to arrive, the system call pair `listen`/`accept` is used instead:

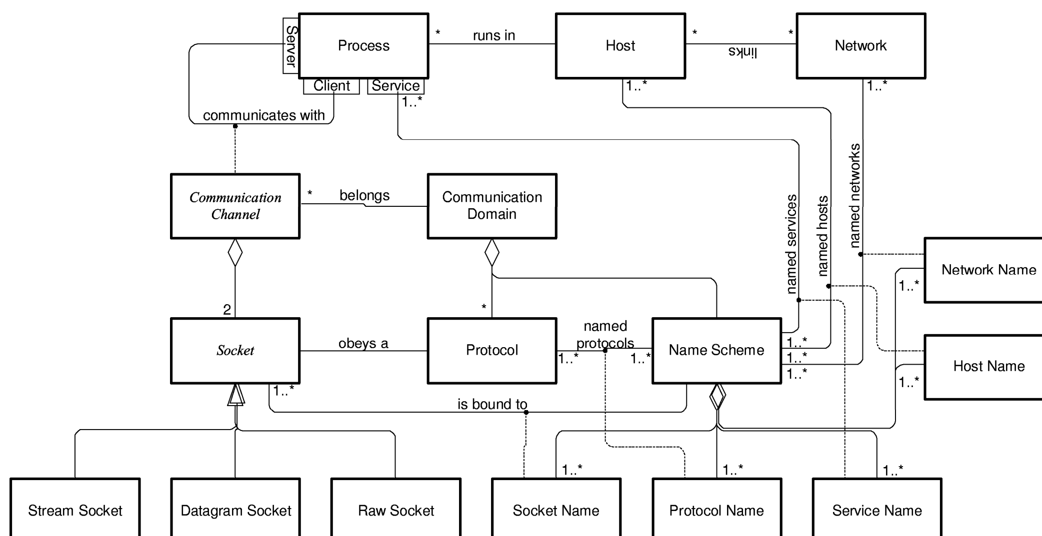
```
error = listen(s, backlog);
int error, s, backlog;
```

Connections are then received, one at a time, with:

```
snew = accept(s, peeraddr, peeraddrlen);
int snew, s, peeraddrlen;
struct sockaddr *peeraddr;
```

A variety of calls are available for transmitting and receiving data. The usual `read` and `write` system calls, as well as the newer `send` and `recv` system calls can be used with *sockets* that are in a connected state. The `sendto` and `recvfrom` system calls are

Figure 2.1—OMT object model for BSD IPC



most useful for connectionless *sockets*, where the peer's address is specified with each transmitted message. Finally, the `sendmsg` and `recvmsg` system calls support the full interface to the interprocess-communication facilities.

The `shutdown` system call terminates data transmission or reception at a *socket*. *Sockets* are discarded with the normal `close` system call.

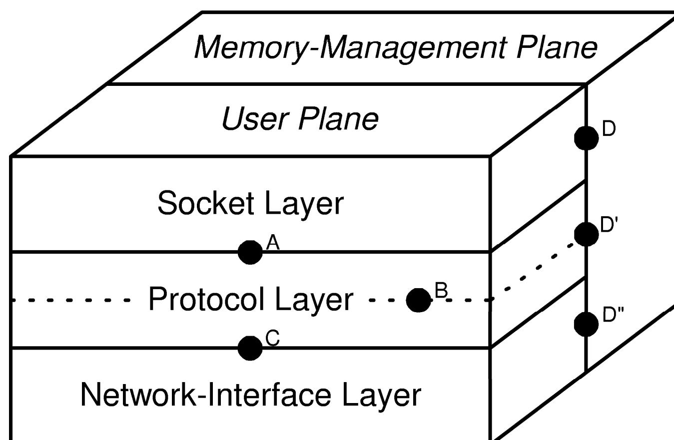
2.3 BSD's networking architecture

BSD's networking architecture has two planes, as shown in Figure 2.2: the ***user plane*** and the ***memory management plane***. The *user plane* defines a framework within which many *communication domains* may coexist and network services can be implemented. The *memory management plane* defines memory management policies and procedures that comply with the *user plane*'s memory requirements. More on this a little further.

2.3.1 Memory management plane

It is well known [McKusick et al. 1996; Wright and Stevens 1995] that the requirements placed by interprocess communication and network protocols on a memory management scheme tend to be substantially different from those of other parts of the operating system. Basically, network messages processing require attaching and/or detaching protocol headers and/or trailers to messages. Moreover, some times these headers' and trailers' sizes vary with the communication session's state; some other times the number of these protocol elements is, a priori, unknown. Consequently, a special-purpose memory management facility was created by the BSD developing team for the use of the interprocess communication and networking systems.

Figure 2.2—BSD's two-plane networking architecture. The *user plane* is depicted with its layered structure, which is described in following sections. Bold circles in the figure represent defined interfaces between planes and layers: A) Socket-to-Protocol, B) Protocol-to-Protocol, C) Protocol-to-Network Interface, and D) User Layer-to-Memory Management. Observe that this architecture implies that layers share the responsibility of taking care of the storage associated with transmitted data



The memory management facilities revolve around a data structure called an `mbuf`. `Mbuf`s, or memory buffers, are 128 bytes long, with 100 or 108 of this space reserved for data storage. There are three sets of header fields that might be present in an `mbuf` and which are used for identifying and managing purposes. Multiple `mbuf`s can be linked forming `mbuf`-chains to hold an arbitrary quantity of data. For very large messages, the system can associate larger sections of data with an `mbuf` by referencing an external `mbuf-cluster` from a private virtual memory area. Data is stored either in the internal data area of the `mbuf` or in the external cluster, but never in both.

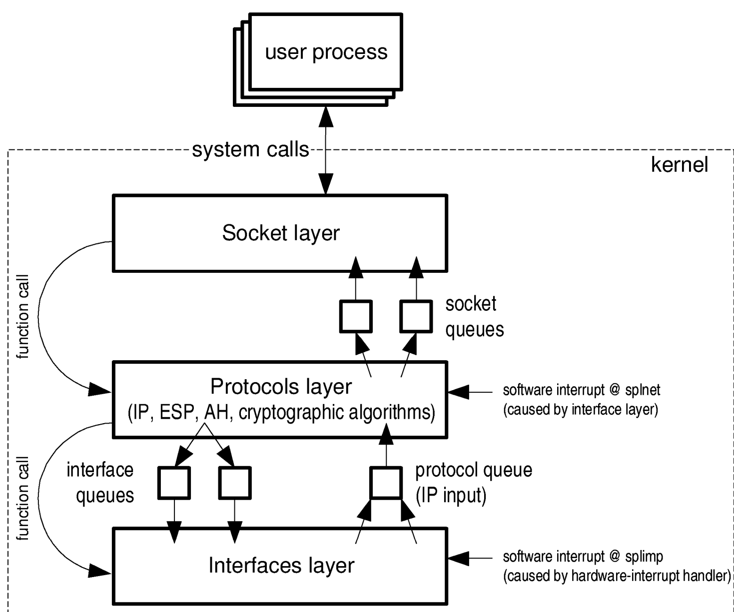
2.3.2 User plane

The *user plane*, as said before, provides a framework within which many *communication domains* may coexist and network services can be implemented. Networking facilities are accessed through the *socket* abstraction. These facilities include:

- A structured interface to the *socket layer*.
- A consistent interface to hardware devices.
- Network-independent support for message routing.

The BSD developing team devised a pipelined implementation for the *user plane* with three vertically delimited stages or layers. As Figure 2.2 and Figure 2.3 show, these layers are the ***sockets layer***, the ***protocols layer***, and the ***network-interfaces layer***. Jointly, the *protocols layer* and the *network-interfaces layer* are named the ***networking support***. Basically, the *sockets layer* is a protocol-independent interface used by applications to access the *networking support*. The *protocols layer* contains the implementation of the *communication domains* supported by the system, where each *communication domain* may have its own internal structure. Last but not least, the *network-interfaces layer* is mainly concerned with driving the transmission media involved.

Figure 2.3—BSD networking user plane's software organization



Entities at different layers communicate through well-defined interfaces and their execution is decoupled by means of message queues, as shown in Figure 2.3. Concurrent access to these message queues is controlled by the ***software interrupt*** mechanism, as explained in the next section.

2.4 The software interrupt mechanism and networking processing

Networking processing within the BSD operating system is pipelined and interrupt driven. To show how this works, let us describe the sequence of chief events occurred during message reception and transmission. If you feel lost during the first read, please keep an eye on Figure 2.3 during the second pass. It helps. During the following description, when we say "the system" we mean a computer system executing a BSD derived operating system.

2.4.1 Message reception

When a network interface card captures a message from a communications link, it posts a hardware interrupt to the system's central processing unit. Upon catching this interrupt—preempting any running application program and entering supervisor mode and the operating system kernel's address space—the system executes some *network-interfaces layer* task and marshals the message from the network interface card's local memory to a *protocols layer*'s `mbuf` queue in main memory. During this marshaling the system does any data-link protocol duties and determines to which *communication domain* the message is destined. Just after leaving the message in the selected protocol's `mbuf` queue and before terminating the hardware interrupt execution context, the system posts a software interrupt addressed to the corresponding *protocols layer* task. Considering that the arrived message is destined to a system's application program and that the addressed application has an opened *socket*, the system, upon catching the outstanding software interrupt, executes the corresponding *protocols layer* task and marshals the message from the protocol's `mbuf` queue to the addressed *socket*'s `mbuf` queue. All protocols processing within the corresponding *communication domain* takes place at this software interrupt's context. Just after leaving the message into the addressed *socket*'s `mbuf` queue and before terminating the software interrupt execution context, the system flags for execution any application program that might be sleeping over the addressed *socket*, waiting for a message to arrive. When the system is finished with all the interrupts execution contexts and its scheduler schedules for execution the application program that just received the message, the system executes the corresponding *sockets layer* task and marshals the message from the *socket*'s `mbuf` queue to the corresponding application's buffer in user address space. Afterwards, the system exits supervisor mode and the address space of the operating system's kernel and resumes the execution of the communicating application program.

Here let us spot a performance detail of the previous description. The message marshalling between `mbuf` queues does not always imply a data copy operation. There are copy-operations involve when marshalling messages between a network interface card's local memory and a protocol's `mbuf` queue and between a *socket*'s `mbuf` queue

and an application program's buffer. But there is no data copy operation between a protocol's and a *socket*'s `mbuf` queues. Here, only `mbuf` references—also know as pointers—are copied.

2.4.2 Message transmission

Message transmission network processing may be initiated by several events. For instance, by an application program issuing the `sendmsg`—or similar—system call. But it can also be initiated when forwarding a message, when a protocol timer expires or when the system has deferred messages.

When an application program issues the `sendmsg` system call, giving a data buffer as one of the arguments, (other arguments are, for instance, the *communication domain* identification and the destination *socket*'s address) the system enters into supervisor mode and into the operating system's kernel address space and executes some *sockets layer* task. This task builds an `mbuf` upon the selected *communication domain* and *socket* type and, considering that the given data buffer fits inside one `mbuf`, it copies into the built `mbuf`'s payload the contents of the given data buffer. In case that the communication channel protocol's state allows the system to immediately transmit a message, the system executes the appropriate *protocols layer* task and marshals the message through the arbitrary protocol structure of the corresponding *communication domain*. Among other protocol-dependent tasks, the system here selects a communication link for transmitting the message out. Considering that the network interface card attached to the selected communications link is idle, the system executes the appropriate *network-interfaces layer* task and marshals the message from the corresponding `mbuf` in main memory to the network interface card's local memory. At this point the system hands over the message delivery's responsibility to the network interface card.

Observe that under the considered situation the system executes the message transmission in a single execution context—that of the communicating application—and no intermediary buffering is required. On the contrary, if for instance the system finds an addressed network interface card busy, the system would place the `mbuf` in the corresponding network interface's `mbuf` queue and would defer the execution of the *network-interfaces layer* task. For cases like this, network interface cards are built to produce a hardware interrupt not just when receiving a message but at the end of every busy period. Moreover, network interface card's hardware interrupt handlers are built to always check for deferred message at the corresponding network interface's output `mbuf` queue. When deferred messages are found, the system does whatever is required to transmit them out. Observe that in this case the message transmission is done in the execution context of the network interface card's hardware interrupt.

Another scenario happens if a communication channel protocol's state impedes the system to immediately transmit a message. For instance, when a TCP connection's transmission window is closed [Stevens, 1994]. In this case, the system would place the message's `mbuf` in the corresponding *socket*'s `mbuf` queue and defers the execution of the *protocols layer* task. Of course, the deferring protocol must have some built-in means for later transmitting or discarding any deferred message. For instance, TCP may open a connection's transmission window after receiving one or more segments from the other end. Upon opening the transmission window, TCP will start transmitting as

many deferred messages as possible as soon as possible—that is, just after finishing message reception. Observe that in this case the message transmission is done in the execution context of the *protocols layer* reception software interrupt. Also observe that when transmitting deferred messages at the *protocols layer*, the system may defer again at the *network-interfaces layer* as explained above.

Communications protocols generally defined timed operations that require the interchange of messages with peers, and thus require transmitting messages. For instance, TCP's delayed acknowledge mechanism [Stevens 1994]. In this cases, protocols may relay on the BSD ***callout*** mechanism [McKusick et al, 1996] and request for the system to execute some task at predefined times. The BSD *callout* mechanism uses the system's real-time clock for scheduling the execution of any enlisted task. It arranges itself to issue software interrupts every time an enlisted task is required to execute. If the called out task initiates the transmission of a networking message, this message transmission is done in the execution context of the *callout* mechanism software interrupt. Once again, as explained above, transmission hold off may happen at the *network-interfaces layer* as explained above.

Finally, let us consider the message-forwarding scenario. In this scenario some communications protocol—implemented at the *protocols layer*—is capable of forwarding messages; for instance, the Internet Protocol, IP, within the Internet *communication domain* [Stevens 1994]. During message reception, a protocol like IP may find out that the received message is not addressed to the local system but to another system to which it knows how to get to by means of a routing table. In this case, the protocol will launch a message transmission task upon the message being forwarded. Observe that this message transmission processing is done in the execution context of the *protocols layer* reception software-interrupt.

2.4.3 Interrupt priority levels

There is a priority level assigned to each hardware and software interrupt handler. The ordering of the different priority levels means that some interrupt handler preempts the execution of any lower-priority one. One concern with these different priority levels is how to handle data structures shared between interrupt handlers executed at different priority levels. The BSD interprocess communication facility code is sprinkled with calls to the functions `splimp` and `splnet`. These two calls are always paired with a call to `splx` to return the processor to the previous level. The result of this synchronization mechanism is a sequence of events like the one depicted in the Figure 2.4.

1) While a *sockets layer* task is executing at `spl0`, an Ethernet card receives a message and posts a hardware interrupt causing a *network interfaces layer* task—the Ethernet device driver—to execute at `splimp`. This interrupt preempts the *sockets layer* code.
2) While the Ethernet device driver is running, it places the received message into the appropriate *protocols layer*'s input `mbuf` queue—for instance IP—and schedules a software interrupt to occur at `splnet`. The software interrupt won't take effect immediately since the kernel is currently running at a higher priority level.

3) When the Ethernet device driver completes, the *protocols layer* executes at `splnet`.

4) A terminal device interrupt occurs—say the completion of a SLIP packet. It is handled immediately, preempting the *protocols layer*, since terminal processing's priority, at `spltty`, is higher than *protocols layer*'s.

5) The SLIP device driver places the received packet onto IP's input `mbuf` queue and schedules another software interrupt for the *protocols layer*.

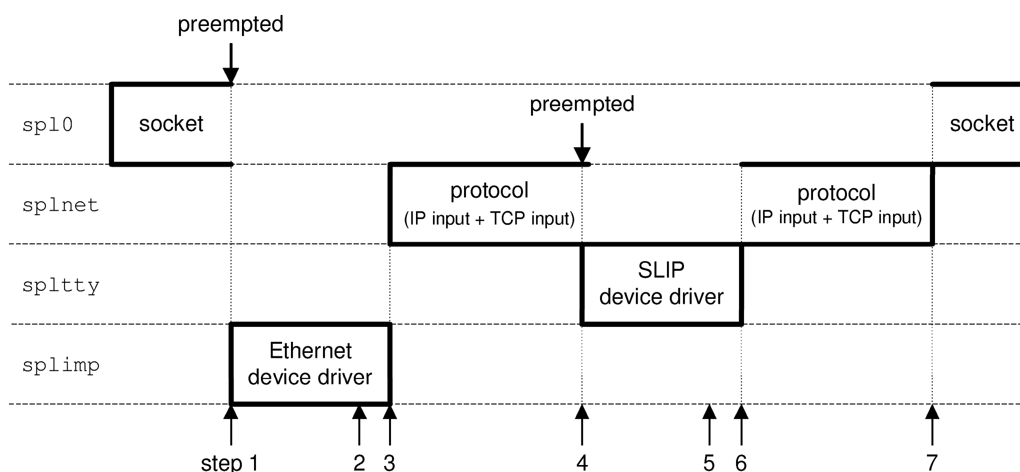
6) When the SLIP device driver completes, the preempted *protocols layer* task continues at `splnet` and finishes processing the message received from the Ethernet device driver. Then, it processes the message received form the SLIP device driver. Only when IP's input `mbuf` queue gets empty the *protocols layer* task will return control to whatever it preempted—the *sockets layer* task in this example.

7) The *sockets layer* task continues form where it was preempted.

2.5 BSD implementation of the Internet protocols suite

Figure 2.5 shows a control and data flow diagrams of the chief tasks that implement the Internet protocols suite within BSD. Furthermore, it shows its control and data associations with chief tasks at both the *sockets layer* and the *network-interfaces layer*. Within the 4.4BSD-lite source code distribution, the files implementing the Internet protocols suite are located at the `sys/netinet` subdirectory. On the other hand, the files implementing the *sockets layer* are located at the `sys/kern` subdirectory. The files implementing the *network-interfaces layer* are scattered among few subdirectories. The tasks implementing general data-link protocol tasks, such as Ethernet, the address resolution protocol or the point-to-point protocol, are located at the `sys/net` subdirectory. On the other hand, the tasks implementing hardware drivers are located at hardware de-

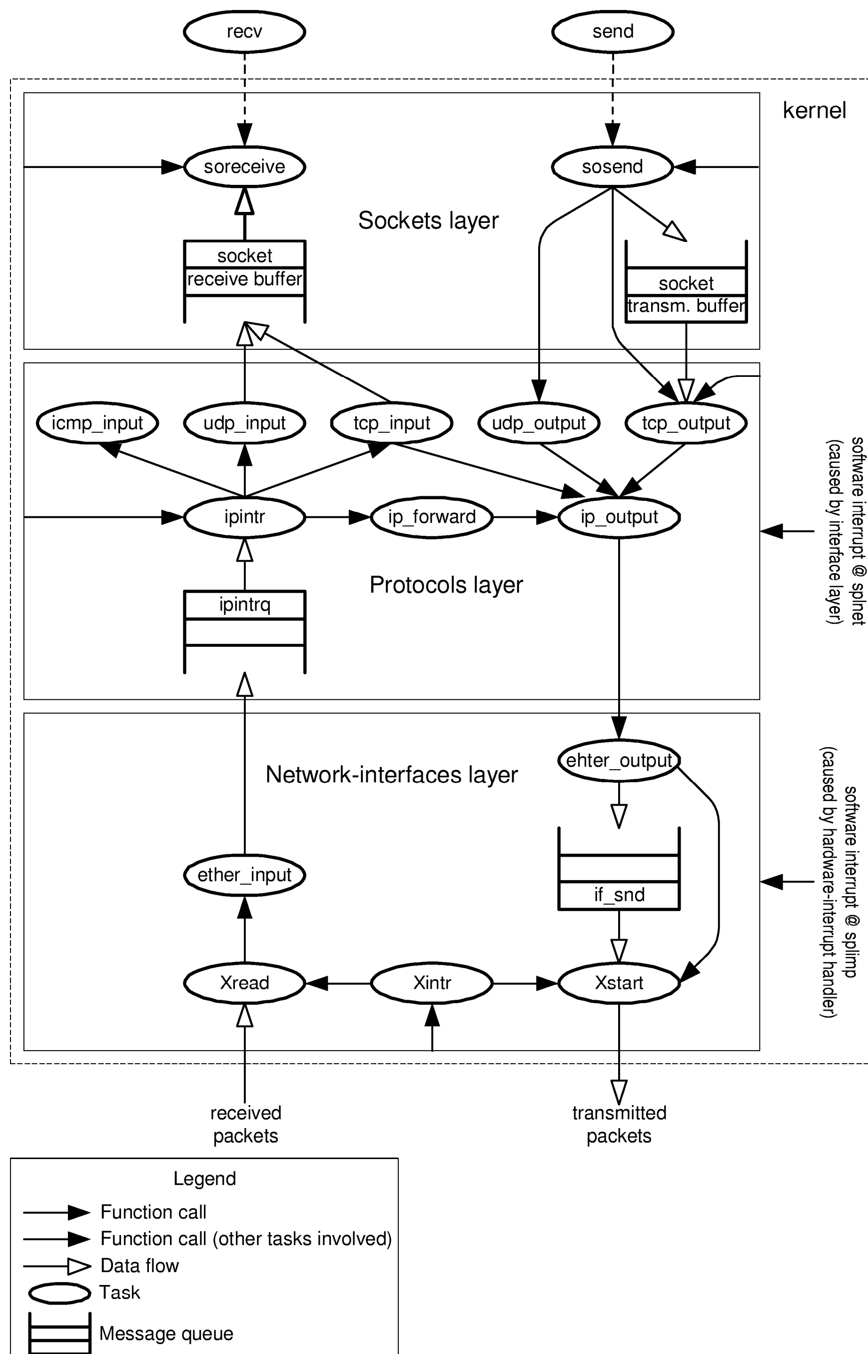
Figure 2.4—Example of priority levels and kernel processing



pendent subdirectories, such as the `sys/i386/isa` or the `sys/pci` subdirectories.

As can be seen in Figure 2.5, the protocols implementation, in general, provides output and input tasks per protocol. In addition, the IP protocol has a special `ip_forwarding` task. It can also be seen that the IP protocol does not have an input task. Instead, the implementation comes with an `ipintr` task. The fact that IP input processing is started by a software interrupt may be the cause of this apparent fault to

Figure 2.5—BSD implementation of the Internet protocol suite. Only chief tasks, message queues and associations are shown. Please note that some control flow arrows are sourced at the bounds of the squares delimiting the implementation layers. This is for denoting that a task is executed after an external event, such as an interrupt or a CPU scheduler event



the general rule. (The FreeBSD operating system drops the `ipintr` task in favor of an `ip_input` task.) Observe that the figure depicts all the control and data flows corresponding to the message reception and message transmission scenarios described in the previous section.

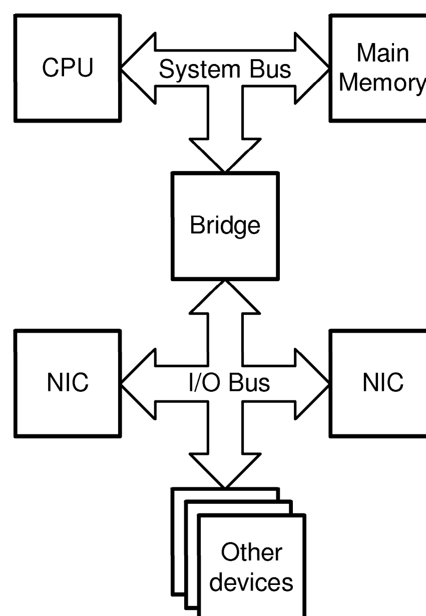
In order to complete the description let me note some facts on the *network-interfaces layer*. The tasks shown at the bottom half of the layer depict hardware dependent tasks. The names depicted, `Xintr`, `Xread` and `Xstart` are not actual task names but name templates. For building actual task names the capital "x" is substituted by the name of a hardware device. For example, the FreeBSD source code distribution has `xlintr`, `xlread` and `xlstart` for the `xl` device driver, which is the device deriver used for the 3COM's 3C900 and 3C905 families of PCI/Fast-Ethernet network interface cards.

2.6 Run-time environment: the host's hardware

The BSD operating system was devised to run on a computing hardware with an organization much like the one shown in Figure 2.6. This computing hardware organization is widely used for building personal computers, low-end servers and workstations or high-end embedded systems. The shown organization is an instance of the classical stored-program computer architecture with the following features [Hennessy and Patterson 1995]:

- A single central processing unit
- A four-level, hierarchic memory (not shown)
- A two-tier, hierarchic bus
- Interrupt driven input/output processing (not shown)

Figure 2.6—Chief components in a general purpose computing hardware.



- Programmable or direct memory access network interface cards

2.6.1 The central processing unit and the memory hierarchy

Nowadays, personal computers and the like computing systems are provisioned with high-performance microprocessors. These microprocessors in general leverage the following technologies: very-low operation cycle period, pipelines, multiple instruction issues, out-of-order and speculative execution, data prefetching or trace caches.

In order to sustain a high operation throughput, this kind of microprocessors requires very fast access to instructions and data. Unfortunately, current memory technology lags behind microprocessor technology in its performance/price ratio. That is, low latency memory components have to be small in order to remain economically feasible. Consequently, personal computers and the like computing systems—but also other computing systems using high-performance microprocessors—are suited with hierarchically organized memory. Ever faster and thus smaller memory components are placed lower in the hierarchy and thus closer to the microprocessor. Several caching techniques are used for mapping large address spaces onto the smaller and faster memory components, which in consequence are named cache memories [Hennessy and Patterson 1995]. These caching techniques mainly consist of replacement policies for swapping out of the cache memory computer program's address space sections (named address space pages) that are not expected to be used in the near future in favor of active ones. The caching techniques also determine what to do with the swapped out address space sections—it may or may not be stored in the memory component at the next higher level, considering that the computer program's complete address space is always resident in the memory component at the top of the hierarchy.

Another important aspect of the microprocessor-memory relationship is the wire latency. That is, the time required for a data signal to travel from the output ports of a memory component to the input ports of a microprocessor, or vice versa. Nowadays, the lowest wire latencies are obtained when placing a microprocessor and a cache memory in the same chip. The next worst step happens when placing these components within a single package. The next worst step occurs when the cache memory is part of the main memory component and thus it is at the opposite side of the system bus with respect to the microprocessor.

Let us cite some related performance numbers of an example microprocessor. The Intel's Pentium 4 microprocessor is available at speeds ranging from 1.6 to 2.4 GHz. It has a pipelined, multiple issue, speculative, and out-of-order engine. It has a 20 KB, on-chip, separated data/instruction level-one cache, whose wire latency is estimated at two clock cycles. And it has a 512 or 256 KB on-chip and unified level-two cache, whose wire latency is estimated at 10 clock cycles.

2.6.2 The busses organization

For reasons not relevant to this discussion, the use of a hierarchical organization of busses is attractive. Nowadays, personal computers and the like computing systems come with two-tier busses. One bus, the so named system bus, links the central processing unit and the main memory through a very fast point-to-point bus. The second bus,

named the input/output bus, links all input/output components or periphery devices, like network interface cards and video or disk controllers, through a relatively slower multi-drop input/output bus.

For quantitatively putting these busses on perspective, let us note some performance numbers of two widely deployed busses: the system bus of Intel's Pentium 4 microprocessor and the almost omnipresent Peripheral Component Interconnect input/output bus. [Shanley and Anderson 2000] The specification for the Pentium 4's system bus states a speed operation of 400 MHz and a theoretical maximum throughput of 3.2 Gigabytes per second. (Here, 1 Gigabytes equals 10^9 bytes.) On the other hand, the PCI bus specification states a selection of path widths between 32 and 64 bits and a selection of speed operations between 33 and 66 MHz. Consequently, the theoretical maximum throughput for the PCI bus stays between 132 and 528 Mbytes per second for the 33-MHz/32-bit PCI and the 66-MHz/64-bit PCI, respectively. (Here, 1 Mbytes equals 10^6 bytes.)

2.6.3 The input/output bus' arbitration scheme

One more important aspect to mention with respect to the input/output bus is its arbitration scheme. Because the input/output bus is a multi-drop bus, its path is shared by all components attached to it and thus some access protocol is required.

The omnipresent PCI bus [Shanley and Anderson 2000] uses a set of signals for implementing a use-by-request master-slave arbitration scheme. These signals are emitted through a set of wires separated from the address/data wires. There is a request/grant pair of wires for each bus attachment and a set of shared wires for signaling an initiator-ready event, (FRAME and IRDY) a target-ready event, (TRDY and DEVSEL) and for issuing commands (three wires).

A periphery device attached to the PCI bus (device for short) that wants to transfer some data, requests the PCI bus mastership by emitting a request signal to the PCI bus arbiter. (Bus arbiter for short.) The bus arbiter grants the bus mastership by emitting a grant signal to a requesting device. A granted device becomes the bus master and drives the data transfer by addressing a slave device and issuing to it read or writes commands. A device may request bus mastership and the bus arbiter may grant it at any time, even when other device is currently performing a bus transaction, in what is called "hidden bus arbitration." This seams a natural way to improve performance. However, devices may experience reduced performance or malfunctioning if bus masters are preempted to quickly. Next subsection discusses this and other issues regarding performance and latency.

The PCI bus specification does not defines how the bus arbiter should behave when receiving simultaneous requests. The 2.1PCI bus specification only states that the arbiter is required to implement a fairness algorithm to avoid deadlocks. Generally, some kind of bi-level round robin policy is implemented. Under this policy, devices are separated in two groups: a fast access and a slow access group. The bus arbiter rotates grants through the fast access group allowing one grant to the slow access group at the end of each cycle. Grants for slow access devices are also rotated. Figure 2.7 depicts this policy.

2.6.4 PCI hidden bus arbitration's influence on latency

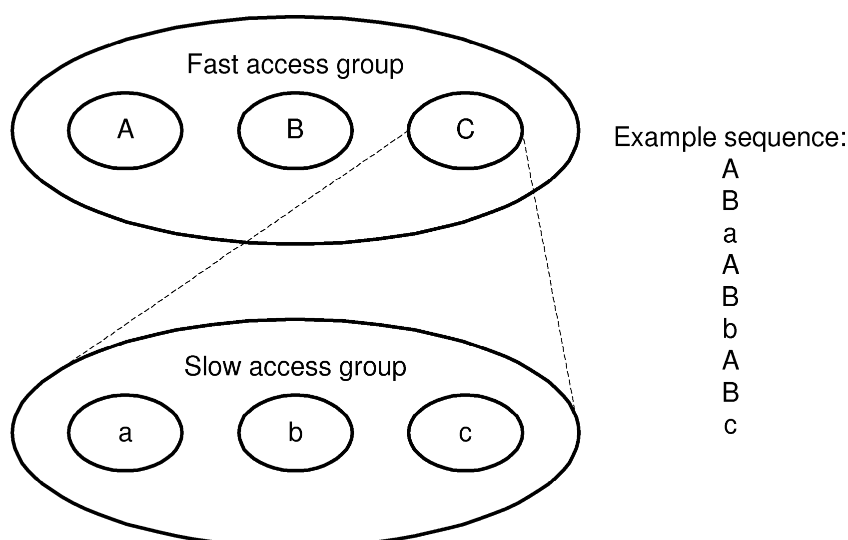
PCI bus masters should always use burst transfers to transfer blocks of data between themselves and target devices. If a bus master is in the midst of a burst transaction and the bus arbiter removes its grant signal, this indicates that the bus arbiter has detected a request from another bus master and is granting bus mastership for the next transaction to the other device. In other words, the current bus master has bee preempted. Due to PCI's hidden bus arbitration this could happen any moment, even one bus cycle before the current bus master has initiated its transaction. Evidently this hampers PCI's burst transactions support and leads to bad performance.

In order to avoid this the 2.1PCI bus specification mandates the use of a master latency timer per PCI device. The value contained in this latency timer defines the minimum amount of time that a bus master is permitted to retain bus mastership. Therefore, a current bus master retains bus mastership until either it completes its burst transaction or its latency timer expires.

Note that independently of the latency timer a PCI device must be capable of managing bus transaction preemption; that is, it must be capable of "remembering" the state of a transaction so it may continue where it left off.

The latency timer is implemented as a configuration register in a PCI device's configuration space. It is either initialized by the system's configuration software at startup, or contains a hardwire value. It may equal zero, in which case a device can only enforce single data phase transactions. Configuration software computes latency timer values for PCI devices not having it hardwire from its knowledge of the bus speed and each PCI device's target value, stored in another PCI configuration register.

Figure 2.7—Example PCI arbitration algorithm



2.6.5 Network interface card's system interface

There are two different techniques for interfacing a computer system with periphery devices like network interface cards. If using the programmable input/output technique, a periphery device interchanges data between its local memory and the system's main memory by means of a program executed by the central processing unit. This program articulates either input/output or memory instructions that read or write data from or to particular main memory's locations. These locations were previously allocated and initialized by the system's configuration software at startup. The periphery device's and motherboard's organizations determine the use of either input/output or memory instructions. When using this technique, periphery devices interrupt the central processing unit when they want to initiate a data interchange.

With the direct memory access (DMA) technique the data interchange is carried out without the central processing unit intervention. Instead, a DMA periphery device uses a pair of specialized electronic engines for performing the data interchange with the system's main memory. The one specialized engine is part of the same periphery device and the other is part of the bridge chipset; see Figure 2.6. Evidently, the input/output bus must support DMA transactions. In a DMA transaction, one engine assumes the bus master role and issues read or write commands; the other engine's role is as servant and follows commands. Generally, the DMA engine at the bridge chipset may assume both roles. When incorporating a master DMA engine, a periphery device interrupts the central processing unit after finishing a data interchange. It is important to note that DMA engines do not allocate nor initialize the main memory's locations from or to where data is read or written. Instead, the corresponding device driver is responsible of that and somehow communicates the location's addresses to the master DMA engine. Next subsection further explains this.

Periphery devices' system interface may incorporate both previously described techniques. For instance, they may relay on programmable input/output for setup and performance statistics gathering tasks and on DMA for input/output data interchange.

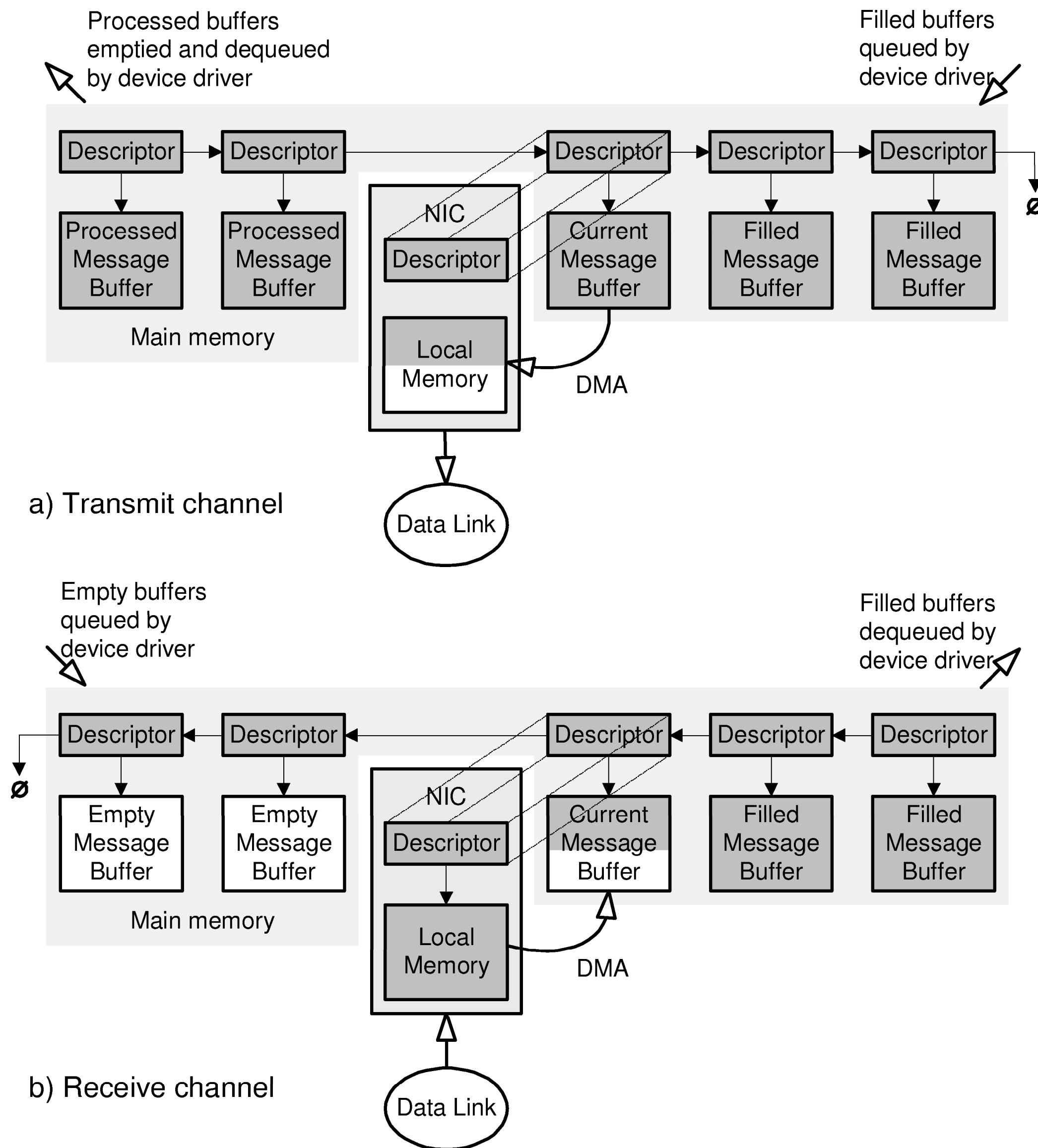
2.6.6 Main memory allocation for direct memory access network interface cards

Generally, a DMA capable network interface card supports the usage of a linked list of message buffers, `mbufs`, for data interchange with main memory; see Figure 2.8. During startup, the corresponding device driver builds two of this `mbufs` lists, one for handling packets exiting the system, named the transmit channel, and the other for handling packet entering it, named the receive channel. `Mbuf`s in these lists are wrapped with descriptors that hold additional list management information, including the `mbuf`'s main memory start address and size. Network interface cards maintain a local copy of the current `mbuf`'s list descriptor. They use the list descriptor's data to marshal DMA transfers. A network interface card may get list descriptors either autonomously, by DMA, or by device driver command, by programmable input/output code. The method used depends on context as explained in the next paragraphs. Generally, a list descriptor includes a "next element" field. If a network interface card supports it, it uses this field to fetch the next current `mbuf`'s list descriptor. This field is set to zero to instruct the network interface card to stop doing DMA through a channel.

Transmit channel. At system startup, any transmit channel is empty and device drivers zero network interface cards' local copy of the current `mbuf` descriptor. When there is a network message to transmit, a device driver queues the corresponding `mbuf` to a transmit channel and does whatever necessary so the appropriate network interface card gets its copy of the new current `mbuf`'s list descriptor. This means that device drivers either copies the list descriptor by programmable input/output code or instruct network interface cards to DMA copy it. It may also happen that device drivers do not have to signal network interface cards because the latter are programmed to periodically poll transmit channels looking for new elements. Generally, list descriptors have a "message processed" field required for transmit channel operation. After a network interface card DMA copies a message from the transmit channel, it sets the "message processed" field and, after transmitting the message through the data link, signals the appropriate device driver to notify of a message transmission. (Signaling may be batched for improved performance.) When acknowledging an end-of-transmission signal, a device driver will walk the transmit channel dequeuing each list element that has it "message processed" field set.

Receive channel. At system startup device drivers provide receive channels with a predefined number of empty `mbuf`s. Naturally, this number is a trade-off between channel-overrun probability and memory wastage, which in turn depends on the operation velocity difference between the host computer and the data link. Continuing with system startup, device drivers make whatever necessary so network interface cards get their copy of the new current `mbuf`'s list descriptor of the appropriate receive channel. This means that device drivers either copy the list descriptor by programmable input/output code or instruct network interface cards to DMA copy it. It may also happen that device drivers do not have to signal network interface cards because the latter are programmed to periodically poll receive channels looking for new elements. After receiving and DMA coping one or more network messages, a network interface card signal the appropriate device driver to notify of a message reception. When acknowledging a reception signal, a device driver will walk the receive channel dequeuing and processing each list element that has its "message size" field greater than zero. Moreover, a device driver must provide its receive channel with more empty `mbuf`s as soon as possible for avoiding the corresponding network interface card to stall; a situation that may result in network message losses.

Figure 2.8—Main memory allocation for direct memory access network interface cards



2.7 Other system's networking architectures

BSD-like networking architectures are known to have good and bad qualities almost since its inception [Clark 1982]. While communications links worked at relatively low speeds the tradeoff between modularity and efficiency was positive. With the advent of multi-megabit data communication technologies this started not to hold true. Worst yet, since some five years ago networking application programs are not improving performance proportionally to the central processing unit's and communication link's speeds. Others [Abbot and Peterson 1993; Coulson and Blair 1995; Druschel and Banga 1996; Druschel and Peterson 1993; Eicken et al. 1992; Geist and Westall 1998; Hutchinson and Peterson 1991; Mosberger and Peterson 1996] have pointed that operating system overheads and networking software not exploiting cache memory features cause the problem. These same people have proposed new networking architectures to improve overall networking application programs' performance. Strikingly, although these new networking architectures are relatively old none have been deployed in production systems. Arguably, the reason is that most of these networking architectures required large changes in networking application programs, at best. In the worst case they also require changes in communication protocols' design and implementation.

In this section we briefly explore post-BSD networking architectures. We think this is interesting in the context of this document because we believe that these relatively new networking architectures have several similarities with the BSD networking architecture. Consequently, the methodology defined in this document may easily be applied for studying the performance of these other systems.

2.7.1 Active Messages [Eicken et al. 1992]

An active message is a message that incorporates the name of the remote procedure that will process it. When an active message arrives to the system, the operating system does not have to buffer the data because it can learn from the active message the name of the software module where the data goes. Like traditional communications protocol messages, active messages are encapsulated. Differently form traditional protocol layers, each software layer can learn the next procedure that will process the message and directly call it to run. This can avoid copy memory operations and reduce central processing unit context switches. Unfortunately, this requires a network wide naming space for procedures that is not standard in current protocol specifications. Fast Sockets [Rodrigues, Anderson and Culler 1997] is protocol stack with active messages that implements a BSD socket like network application program interface and can inter-operate with legacy TCP/IP systems. However, this interoperation is limited.

2.7.2 Integrated Layer Processing [Abbot and Peterson 1993]

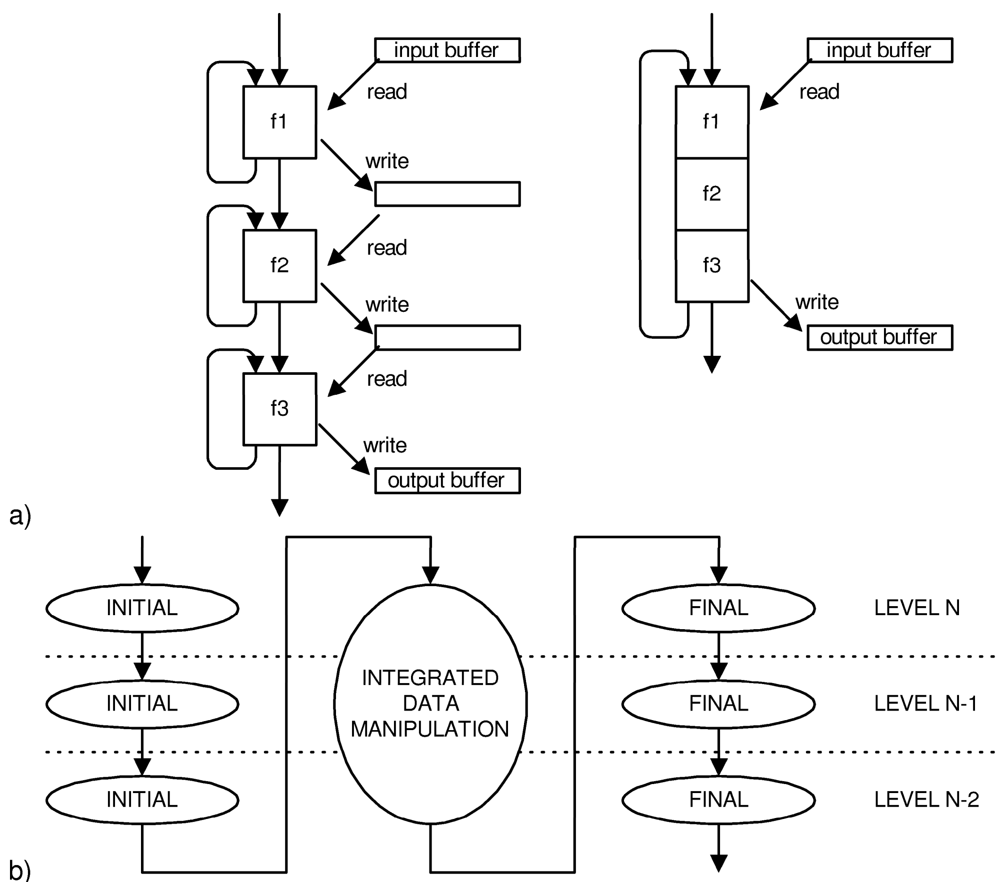
Integrated Layer Processing (ILP) reduces the copy memory operations and creates a running pipeline of layers' code by means of a proposed dynamic code hooking. This dynamic hooking, as showed in Figure 2.9, allows integrally running independently constructed layers' code and eliminates interlayer buffering. The result is improved performance, due to better cache behavior, less copy memory operations and

less central processing unit context switches, all without scarifying modularity. A brief description of this technique follows.

Dynamic code hooking requires that software modules implement a special interface. Also, the hooked modules have to agree on the data type that they process, i.e., a machine word. This means that ILP cannot be applied to legacy protocol stacks but new protocols can be designed to meet the interface and data specification. Basically, the module interface is a mixed form of code-in-lining and loop unrolling with an interlayer communication mechanism implemented with central processing unit's registers. (this can be done only if all modules process data one word at a time.) That is, the programmer must implement each module as a loop that processes incoming metadata a word at a time. Inside the loop, he must place an explicit hook that the runtime environment will use to dynamically link the next layer's code and transfer data. The first and last modules in the pipeline are special because they read/write information from/to a buffer in memory.

There is another restriction to ILP. Protocol metadata is encapsulated; that is, one layer's header is another one's data. Furthermore, these layer's header could be no-existent for a third one. So, for this technique to work, the integrated processing can only be applied to the part of a message that all layers agree exists and has the same meaning; that is, the user application data. This means another change in protocol specification; protocol processing must be divided in three parts: (1) protocol initialization, (2) data processing and (3) protocol consolidation. The first and third parts implement

Figure 2.9—Integrated layering processing [Abbot and Peterson 1993]



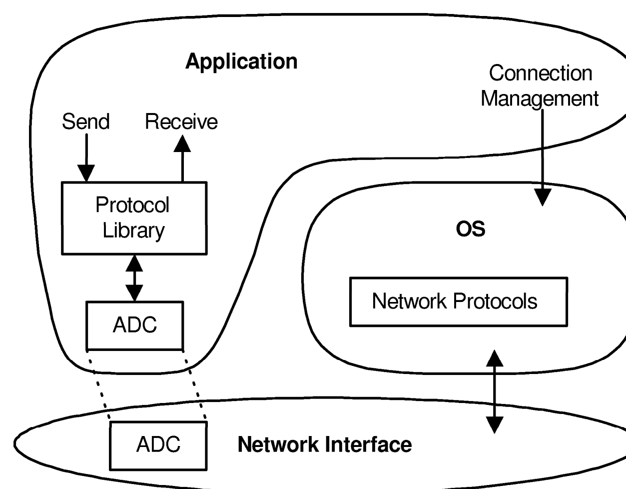
protocol control so they process control data in message headers and/or trailers. Those parts have to be run in the traditional serial order. Only data processing parts can be integrated. This is shown in Figure 8

2.7.3 Application Device Channels [Druschel 1996]

This technique improve overall telematic system performance by allowing common telecommunication operations bypass operating system's address space; that is, send and receive data operations are done by layered protocol code at user application's address space. System security is preserved because connection establishment and tire down are still controlled by protocol code within operating system's kernel.

Figure 2.10 shows a typical scenario for application device channels (ADCs). There, it can be seen that an ADC is formed by a pair of code stubs that cooperate to send and receive messages to and from the network. One stub is at the adapter driver and the other at the user application. The user-level layered protocols can process messages using single-address-space procedure-calls to improve overall performance. Also, because the operating system's scheduler is not aware of the layered structure of user-level protocols, it is unlikely to interleave its run with other processes and overall performance is improved. Moreover, because there is no need to general-purpose protocols, user-level protocols can easily be optimized to meet user application needs, further improving performance. Finally, there is still a need to implement some protocol code inside the kernel because the operating system controls ADC allocation and connection establishment and tired down.

Figure 2.10—Application Device Channels [Druschel 1996]

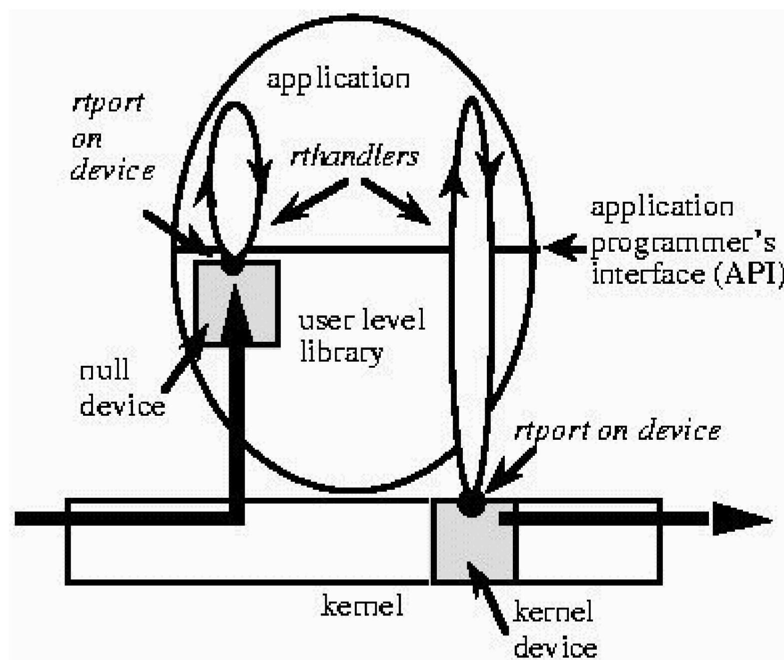


2.7.4 Microkernel operating systems' extensions for improved networking [Coulson et al. 1994; Coulson and Blair 1995]

Even though mechanism like Application Device Channels can exploit the notion of user-level resource management to improve traditional telematic system's performance, other requirements of modern telematic systems cannot be met. Telematic systems like distributed multimedia, loosely couple parallel computing and wide-area network management, require real time behavior and end-to-end communications quality assurance mechanisms, better known as QoS mechanisms. While legacy operating systems cannot provide these requirements [Vahalia 1996], the so named ***microkernel*** operating systems can [Liedtke 1996]. Furthermore, *microkernel* operating systems lend themselves to user-level resource management and can have multiple personalities that permit concurrently run modern and legacy telematic systems.

Under this trend, the SUMO project at Lancaster University (http://www.comp.lancs.ac.uk/computing/research/sumo/sumo.html, current as 2 July 2002) have extend a *microkernel* OS to support an end-to-end quality of service architecture over ATM networks for multimedia distributed computing. Figure 2.11 shows the *microkernel* SUMO extensions. SUMO extends the basic *microkernel* concepts with `rtports`, `rthandlers`, QoS controlled connections, and QoS handlers. This set of abstractions promote a event driven style of programming that, with the help of a split-level central processing unit scheduling mechanism; a QoS external memory mapper; a connection oriented network actor; and a flow management actor, allows the implementation of QoS controlled multimedia distributed systems.

Figure 2.11—SUMO extensions to a *microkernel* operating system [Coulson et al. 1994; Coulson and Blair 1995]



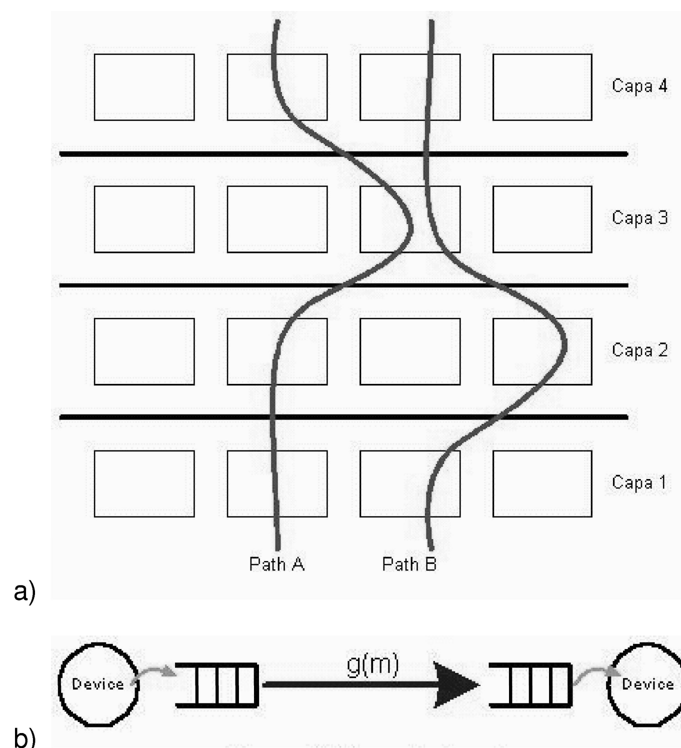
2.7.5 Communications oriented operating systems [Mosberger and Peterson 1996]

While the techniques just described look for better telematic system's performance in computation-oriented operating systems, the Scout project at Princeton University (formerly at University of Arizona) is aimed at building a communication-oriented operating system. This project is motivated by the fact that telematic systems workloads have only a low percentage of computations. Here, communication operations are the common case.

Just as computation oriented operating systems extend the use of the processor by means of software abstractions like processes or threads, the Scout operating system extend the use of physical telecommunication channels by means of a software abstraction called "path". This path is the next step consequence of all the previously described techniques and can be seen as the virtual trail drawn by a message that travels through a layered system from a source module to a sink one; see Figure 2.12. This path has a pair of buffers, one at each end, and a transformation rule. The buffers are used the regular way; the transformation rule represents the set of operations that act over each message that travels through the path.

Within Scout, all resource management and allocation is done on behalf of a path. So, it is possible to obtain the following benefits: (1) early work segregation for better resource management, (2) allow central processing unit scheduling for the hole path which improves performance, (3) allow easy implementation of admission control mechanism and (4) early work discard for reduced waste of resources.

Figure 2.12—Making paths explicit in the Scout operating system [Mosberger and Peterson 1996]



2.7.6 Network processors

Although the network processor concept is not a networking architecture solution to the problem of better telematic systems we decided to include a brief mention to it because it is the current hot topic on programmable-by-software hardware for communications systems. (We got 15 references after a search at IEEE Explore of the pattern "'network processors' in <ti> not 'neural network' in <ti>" restricted to not being before 1999. Besides, see Geppert, L. Editor "The new chips on the block," IEEE Spectrum, January 2001, p.66-68.) Network processors are microprocessors specialized for executing communications software. They exploit communications software's intrinsic parallelism at several levels: data, task, thread and instruction [Verdú et al. 2002]. Data-level parallelism (DLP) exists because, in general, one packet processing is independent of others—previous or subsequent. Vector processor and SIMD chip multiprocessors are known to effectively leverage this kind of parallelism. They are also known to require processing regularity, however, to avoid load unbalancing that hampers performance. Communication software does not exhibit enough processing regularity—in general, the number of instructions required to process one packet has no correlation with others. One work around for this problem exploits thread-level parallelism (TLP) conjunctionally with DLP. The idea then is to simultaneously combine several execution contexts within the microprocessor so if a microprocessor's execution unit stalls for whatever reason, (the address resolution is not complete or the forwarding data has to be fetched form main memory or current communication session's state impedes further processing) the hardware may automatically change the execution context and proceed processing another thread—packet. This technique, which has in multitasking its software counterpart, is known as simultaneous multi-threading [Eggers et al. 1997].

2.8 Summary

- Internet protocols' BSD implementation is networking's de facto standard. In some way or another, most available systems derive their structure, concepts, and/or code from it.
- Networking processing within a BSD system is pipelined and governed by the software interrupt mechanism.
- Message queues are used by a BSD system for decoupling processing between networking pipeline's stages.
- The software interrupt mechanism defines preemptive interrupt levels.
- All of the above suggests to us that widely used, single-queue performance models of software routers incur in significant error. We further discuss this in chapter 3,when proposing a networking queue performance model for these systems.
- Networking processing in PC-based telematic systems has some random variability due to microprocessor features like pipelines, multiple instruction issues, out-of-order and speculative execution, data prefetching, and trace caches. We take this into account when defining a characterization process of these systems, as discuss in chapter 3.
- The PCI I/O bus uses simple bi-level round robin policy for bus arbitration. We will show in chapter 3 that this may hamper a PC-based software router of sustaining system-wide QoS behavior, and in chapter 4 we propose a solution to this problem.
- Devices and device drivers supporting DMA transactions use a pair of message buffer lists, called receive and transmit channels, which may be used to implement a credit-based flow-control mechanism for fairly sharing I/O bus bandwidth, as will be discussed later in chapter 4.