Volume 2, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, USENIX Association, Berkeley, CA (1986).

Rashid, 1980.

R. F. Rashid, "An Inter-Process Communication Facility for UNIX," Technical Report, Carnegie-Mellon University, Pittsburgh, PA (August 14, 1980).

Sunshine, 1977.

C. Sunshine, "Interprocess Communication Extensions for the UNIX Operating System: Design Considerations," Technical Report R-2064/1-AF, Rand Corporation, Santa Monica, CA (June 1977).

UPMV7, 1983.

UPMV7, *UNIX Programmer's Manual*, 7th ed, Volume 1 and 2, Holt, Rinehart & Winston, New York, NY (1983).

CHAPTER 11

Network Communication

In this chapter, we shall study the internal structure of the network subsystem provided in 4.3BSD. The networking facilities provide a framework within which many *network architectures* may coexist. A network architecture comprises a set of network-communication protocols, the *protocol family*, conventions for naming communication endpoints, the *address family* or *address format*, and any additional facilities that may fall outside the realm of connection management and data transfer. Networking facilities are accessed through the *socket* abstraction described in Chapter 10. The network subsystem provides a general-purpose framework within which to implement network services. These facilities include

- A structured interface to the socket level that allows the development of network-independent application software
- A consistent interface to the hardware devices used to transmit and receive data
- Network-independent support for message routing
- Memory management

We describe the internal structure of the network subsystem in Section 11.1. Then we discuss the interface between the socket layer and the network facilities, and examine the interfaces between the layers of software that make up the network subsystem. In Section 11.5, we discuss the routing database used by the network protocols; in Section 11.6, we describe the mechanisms provided to manage buffering and to control congestion. We present the *raw socket* interface that provides direct access to lower-level network protocols in Section 11.7. Finally, in Section 11.8 we discuss an assortment of issues and facilities, including out-of-band data, subnetwork addressing, the Address Resolution Protocol, trailer protocols, and support for network interfaces that reside on the VAX UNIBUS.

After we have discussed the framework in which the network protocols fit, we shall examine the implementations of existing network protocols in Chapter 12.

11.1 Internal Structure

The network subsystem is logically divided into three layers. These three layers manage the following tasks:

- Interprocess data transport
- Internetwork addressing and message routing
- Transmission-media support

The first two layers are made up of modules that implement communication protocols; the software in the third layer is structurally much like a device driver (see Chapter 8).

The topmost layer in the network subsystem is termed the *transport layer*. The transport layer must provide an addressing structure that permits communication between sockets and any protocol mechanisms necessary for socket semantics, such as reliable data delivery. The second layer, the *network layer*, is responsible for the delivery of data destined for remote transport or network-layer protocols. In providing internetwork delivery, the network layer must manage a private routing database or utilize the systemwide facility for routing messages to their destination host. The bottom layer, the *network-interface layer*, is responsible for transporting messages between hosts connected to a common transmission medium. The network-interface layer is mainly concerned with driving the transmission media involved and performing any necessary link-level protocol *encapsulation* and *decapsulation*.

The transport, network, and network-interface layers of the network subsystem most closely resemble the bottom three levels (2 through 0) of the Xerox Network System (NS) architecture. These layers correspond to the *transport*, *network*, and *link* layers of the ISO Open Systems Interconnection Reference Model [ISO, 1984], respectively. The internal structure of the networking software is not directly visible to users. Instead, all networking facilities are accessed through the socket layer described in Chapter 10. Each communication protocol that permits access to its facilities exports a user request routine to the socket layer. This routine is used by the socket layer in providing access to network services.

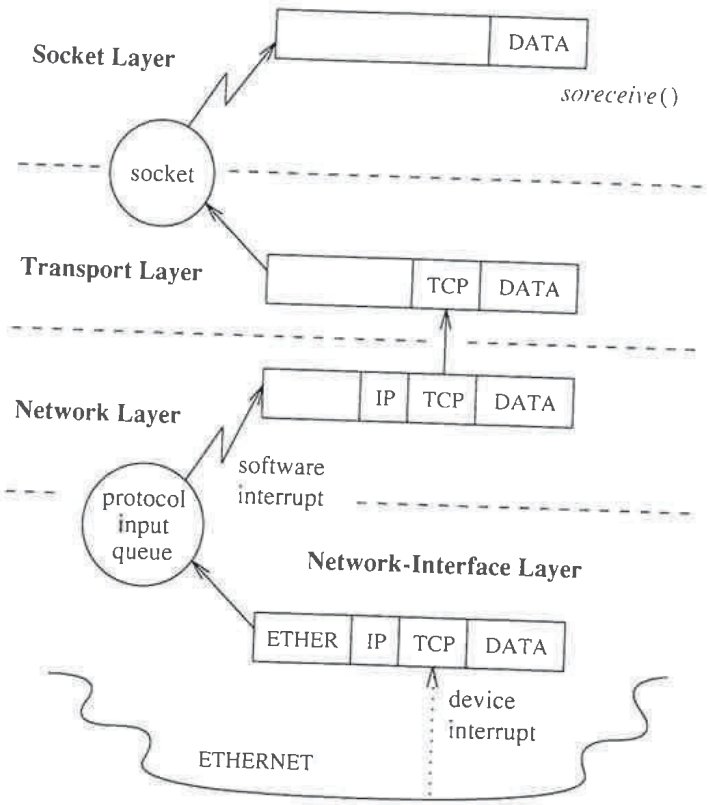
The layering described here is a *logical layering*. The software that implements network services may utilize more or fewer communication protocols according to the design of the network architecture being supported.

Data Flow

Data flow down to the network subsystem from the socket layer through calls to the transport-layer modules that support the socket abstraction. Data received at a

network interface flow upward through communication protocols until they are placed in the receive queue of the destination socket. The downward flow of data typically is initiated by system calls. Data flowing upward are received asynchronously and are passed from the network-interface layer to the appropriate communication protocol through per-protocol input message queues (see Fig. 11.1). The system schedules network protocol processing from the network-interface layer by marking a bit assigned to the protocol in the system's network-interrupt status word and posting a *software interrupt* reserved for triggering network activity. Software interrupts are used to schedule asynchronous network activity, rather than protocols being run as independent UNIX processes, to avoid context-switching overhead. If a message received by a communication protocol is destined for a higher-level protocol, this protocol is invoked directly at software-interrupt level to process the message. Alternatively, if the message is destined for another host and the system is prepared to forward the message, the message will be returned to the network-interface layer for retransmission.

Figure 11.1 Upward data flow in the network subsystem. ETHER—Ethernet header; IP—Internet Protocol header; TCP—Transmission Control Protocol header.



Communication Protocols

Each communication-protocol module is made up of a collection of procedures and private data structures. Protocols are described by a *protocol switch* structure that contains the set of externally visible entry points and certain attributes (see Fig. 11.2). The socket layer interacts with a communication protocol solely through the latter's protocol switch structure, recording the structure's address in a socket's *so_proto* field. This isolation of the socket layer from the networking subsystem is important in ensuring that the socket layer provides users with a consistent interface to all the protocols supported by a system.

Before a protocol is first used, the protocol's initialization routine is invoked. Thereafter, the protocol will be invoked for timer-based actions every 200 milliseconds if the *pr_fasttimo*() entry is present, and every 500 milliseconds if the *pr_slowtimo*() entry point is present. The *pr_drain*() entry is provided so that the system can notify the protocol if it is low on space and would like any noncritical data to be discarded.[1]

Protocols may pass data among themselves in chains of mbufs (see Section 10.3) using the *pr_input*() and *pr_output*() routines. The *pr_input*() routine is used to pass data *up* toward the user, whereas the *pr_output*() routine is used to pass data *down* toward the network. Similarly, control information passes up and down via the *pr_ctlinput*() and *pr_ctloutput*() routines. The *user request routine*, *pr_usrreq*(), is the interface between a protocol and the socket level; it is described in detail in Section 11.2.

In general, a protocol is responsible for storage space occupied by any of the arguments passed downward via these procedures and must either pass the space onward or dispose of it. On output, the lowest level reached must free space passed as arguments; on input, the highest level is responsible for freeing space passed up to it. Auxiliary storage needed by protocols is allocated from the mbuf store. This space is used temporarily to formulate messages or to hold data structures such as state control blocks. Mbufs allocated by a protocol for private use must be freed by that protocol when they are no longer in use.

The *pr_flags* field in a protocol's protocol switch structure describes the protocol's capabilities and certain aspects of its operation that are pertinent to the operation of the socket level (see Table 11.1). Protocols that are connection-based specify the PR_CONNREQUIRED flag so that socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notify the protocol when the user has removed data from a socket's receive queue. This notification allows a protocol to implement acknowledgment on user receipt, and also to update flow-control information based on the amount of space available in the receive queue. The PR_ADDR field indicates that any data placed in a socket's receive queue by the protocol will be preceded by the address of the sender. The PR_ATOMIC flag specifies that each *user* request to send data must be performed in a single

[1] The protocol *pr_drain* routines were never called in the original release of 4.3BSD.

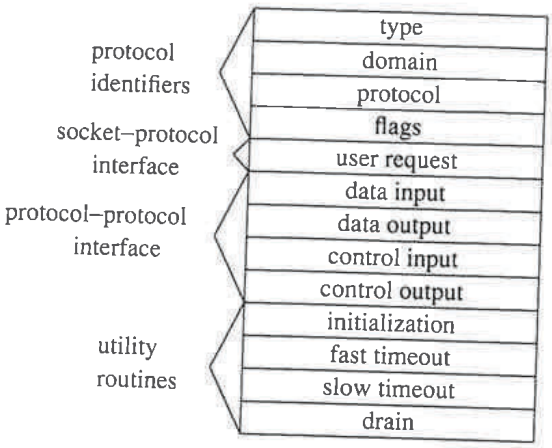


Figure 11.2 Protocol switch structure.

protocol send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates that the protocol supports the transfer of access rights; this flag is currently used only by the protocols in the UNIX communication domain (see Section 10.6).

Network Interfaces

Each network interface configured in a system defines a path through which messages can be sent and received. Normally, a hardware device is associated with this interface, although there is no requirement that one be (for example, all systems have a software *loopback* interface used for debugging and performance analysis). In addition to manipulating the hardware device, a network-interface module is responsible for encapsulation and decapsulation of any link-layer protocol header required to deliver a message to its destination. The selection of the interface to use in delivering a packet is a routing decision carried out at the

Table 11.1 Protocol flags.

Flag	Description
PR_ATOMIC	messages sent separately, each in a single packet
PR_ADDR	protocol presents address with each message
PR_CONNREQUIRED	connection required for data transfer
PR_WANTRCVD	protocol notified on user receipt of data
PR_RIGHTS	protocol supports passing access rights

network-protocol layer. An interface may have addresses in one or more address families. Each address is set at boot time using an *ioctl* system call on a socket in the appropriate domain; this operation is implemented by the protocol family after verifying the operation with an *ioctl* entry point provided by the network interface. The network-interface abstraction provides protocols with a consistent interface to all hardware devices that may be present on a machine.

An interface and its addresses are defined by the structures shown in Fig. 11.3. The network-interface module generally maintains the interface data structure as part of a larger structure that also contains information used in driving the underlying hardware device.

Each network interface has a queue of messages to be transmitted and routines used for initialization (*if_init*()), and output (*if_output*()). If the interface resides on a system bus, a reset routine will be invoked after a bus reset has been performed, so that it may reinitialize itself. An interface may also specify a timer routine and a timer value that (if it is nonzero) the system will decrement once per second, invoking the timer routine when the value expires. The timeout mechanism is typically used by interfaces to implement watchdog schemes for unreliable hardware and to collect statistics that reside on the hardware device.

The state of an interface and certain externally visible characteristics are stored in the *if_flags* field (see Table 11.2). The IFF_UP flag is set when the

Figure 11.3 Network-interface data structures.

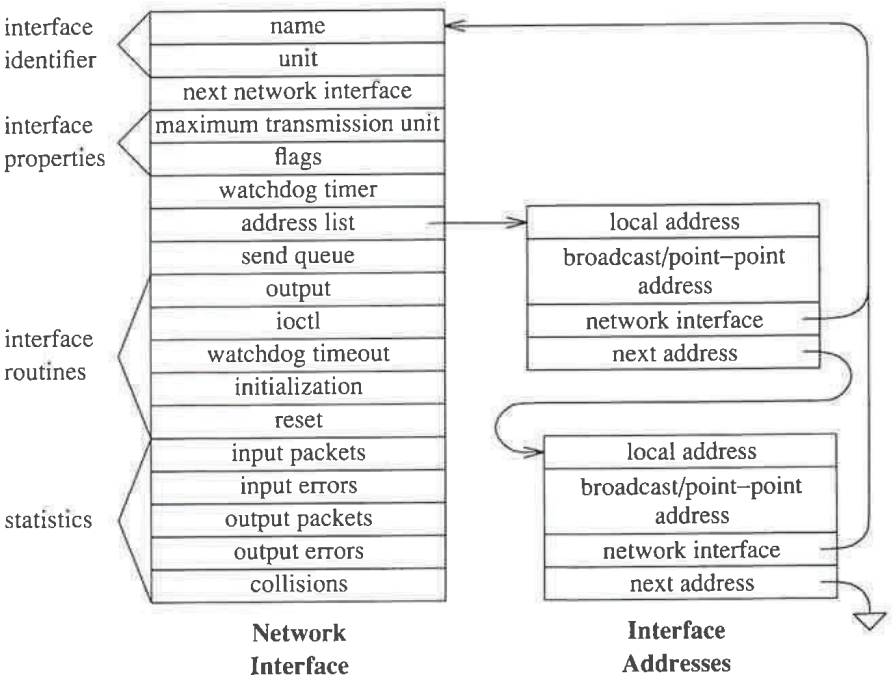


Table 11.2 Network interface flags.

Flag	Description
IFF_UP	interface is available for use
IFF_BROADCAST	broadcast is supported
IFF_DEBUG	enable debugging in the interface software
IFF_LOOPBACK	this is a software loopback interface
IFF_POINTOPOINT	interface is for a point-to-point link
IFF_NOTRAILERS	interface should not use trailer encapsulation
IFF_RUNNING	interface resources have been allocated
IFF_NOARP	interface should not use address-resolution protocol

interface is configured and ready to transmit messages. If an interface is connected to a network that supports transmission of *broadcast* messages, the IFF_BROADCAST flag will be set and the interface's address list will contain a broadcast address to use in sending and receiving such messages. If an interface is associated with a point-to-point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and the interface's address list will contain the address of the host on the other side of the connection. (Note that the broadcast and point-to-point attributes are mutually exclusive, and the two addresses share storage in the interface address structure.) These addresses and the local address of an interface are used by network-layer protocols in filtering incoming packets. An interface sets the IFF_RUNNING flag after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates that an interface should refrain from using *trailer encapsulations* on outgoing packets, or (where per-host negotiation of trailers is possible) that trailer encapsulations should not be requested; trailer protocols are described in Section 11.8. The IFF_NOARP flag indicates that an interface should not use an *Address Resolution Protocol* in mapping internetwork addresses to link-layer addresses; it also is described in Section 11.8.

Interface addresses and flags are set with *ioctl* requests. The SIOCSIFADDR request is used initially to define each interface's addresses. The SIOCSIFFLAGS request can be used to change an interface's state and to perform site-specific configuration. The destination address of a point-to-point link is set with the SIOCSIFDSTADDR request. Corresponding operations exist to read each value. Protocol families also can support operations to set and read the broadcast address. Finally, the SIOCGIFCONF request can be used to retrieve a list of interface names and protocol addresses for all interfaces and protocols configured in a running system; this request permits network processes such as the routing daemon to be constructed without detailed knowledge of the system's internal data structures.

11.2 Socket-to-Protocol Interface

The interface from the socket routines to the communication protocols is through the user request, *pr_usrreq*(), and control output, *pr_ctloutput*()—routines defined in the protocol switch table for each protocol. When the socket layer requires services of a supporting protocol, it makes a call to one of these two routines. The control-output routine is used to implement the *getsockopt* and *setsockopt* system calls; the user-request routine is used for all other operations. Calls to *pr_usrreq*() specify one of the requests shown in Fig. 11.3. Calls to *pr_ctloutput*() specify PRCO_GETOPT to get the current value of an option, or PRCO_SETOPT to set the value of an option.

Protocol User-Request Routine

Given a pointer to a protocol switch entry, *pr*, a call on the user-request routine is of the form,

```
error = (*pr->pr_usrreq)(so, req, m, addr, rights);
    struct socket *so;
    int req;
    struct mbuf *m, *addr, *rights;
```

The mbuf data chain *m* is supplied for output operations and for certain other operations where a result is to be returned. The address *addr* is supplied for address-oriented requests such as PRU_BIND, PRU_CONNECT, and PRU_SEND (when an address is specified; e.g., *sendto*). The *rights* parameter is a pointer to an optional mbuf chain containing user-specified access rights (see Section 10.6). Each protocol is responsible for disposal of the data mbuf chains on output operations. A nonzero return value from the user-request routine indicates a UNIX error number that should be passed to higher-level software. A description of each of the possible requests follows.

- **PRU_ATTACH: Attach protocol to socket** When a protocol is first bound to a socket (with the *socket* system call), the protocol module is called with the PRU_ATTACH request. It is the responsibility of the protocol module to allocate any resources necessary. The *attach* request will always precede any of the other requests, and will not occur more than once.
- **PRU_DETACH: Detach protocol from socket** This operation is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.
- **PRU_BIND: Bind address to socket** When a socket is initially created, it has no address bound to it. This request indicates that an address should be bound to an existing socket. The protocol module must verify that the requested address is valid and is available for use.

Table 11.3 *pr_usrreq* routine requests.

Request	Description
PRU_ABORT	abort connection and detach
PRU_ACCEPT	accept connection from peer
PRU_ATTACH	attach protocol to socket
PRU_BIND	bind name to socket
PRU_CONNECT	establish connection to peer
PRU_CONNECT2	connect two sockets
PRU_CONTROL	control protocol operation (*ioctl*)
PRU_DETACH	detach protocol from socket
PRU_DISCONNECT	disconnect from peer
PRU_FASTTIMO†	service 200-millisecond timeout
PRU_LISTEN	listen for connections
PRU_PEERADDR	fetch peer's address
PRU_PROTORCV†	receive from below
PRU_PROTOSEND†	send to below
PRU_RCVD	have taken data; more room now
PRU_RCVOOB	retrieve out-of-band data
PRU_SEND	send these data
PRU_SENDOOB	send out-of-band data
PRU_SENSE	sense socket status (*fstat*)
PRU_SHUTDOWN	will not send any more data
PRU_SLOWTIMO†	service 500-millisecond timeout
PRU_SOCKADDR	fetch socket's address

† Request used only internally by protocols.

- **PRU_LISTEN: Listen for incoming connections** The *listen request* indicates that the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A listen request always precedes any request to accept a connection.

- **PRU_CONNECT: Connect socket to peer** The *connect request* indicates that the user wants to a establish an association. The *addr* parameter supplied describes the peer to which to connect. The effect of a connect request may vary depending on the protocol. Virtual-circuit protocols use this request to initiate establishment of a network connection. Datagram protocols simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach.

- **PRU_ACCEPT: Accept pending connection** Following a successful listen request and the arrival of one or more connections, this request is made to indicate that the user is about to accept a socket from the queue of sockets with completed connections. The socket supplied as a parameter is the socket that is being *accepted*; the protocol module is expected to fill in the supplied buffer with the address of the peer connected to the socket.

- **PRU_DISCONNECT: Disconnect connected socket** Eliminate an association created with a connect request. This request is used with datagram sockets before a new association is created; it is used with connection-oriented protocols only when the socket is closed.

- **PRU_SHUTDOWN: Shut down socket data transmission** This call is used to indicate that no more data will be sent. The protocol may, at its discretion, deallocate any data structures related to the shutdown and/or notify a connected peer of the shutdown.

- **PRU_RCVD: Data were received by user** This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When the socket layer removes data from the receive queue and passes them to the user, this request will be sent to the protocol module. This request may be used by the protocol to trigger acknowledgments, to refresh windowing information, to initiate data transfer, and so on.

- **PRU_SEND: Send user data** Each user request to send data is translated into one or more PRU_SEND requests. A protocol may indicate that a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description. The data to be sent are presented to the protocol as a chain of mbufs, and an optional address is supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send them immediately or if it may need them at some later time (e.g., for retransmission).

- **PRU_ABORT: Abnormally terminate service** This request indicates an abnormal termination of service. The protocol should delete any existing associations.

- **PRU_CONTROL: Perform control operation** The *control request* is generated when a user performs an *ioctl* system call on a socket and the *ioctl* is not intercepted by the socket routines. This request allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code. The *rights* parameter contains a pointer to a network-interface structure if the *ioctl* operation pertains to a particular network interface.

- **PRU_SENSE: Sense socket status** The *sense request* is generated when the user makes an *fstat* system call on a socket; it requests the status of the associated socket. This call returns a standard *stat* structure that typically contains

only the optimal transfer size for the connection (based on buffer size, windowing information, and maximum packet size).

- **PRU_RCVOOB: Receive out-of-band data** Any *out-of-band* data presently available are to be returned. An mbuf is passed to the protocol module, and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf. An error may be returned if out-of-band data are not (yet) available or have already been consumed. The *addr* parameter contains any options, such as MSG_PEEK, that should be observed while this request is carried out.

- **PRU_SENDOOB: Send out-of-band data** This request is like the send request, but is used for out-of-band data.

- **PRU_SOCKADDR: Retrieve local socket address** The local address of the socket is returned, if one has been bound to the socket. The address (in a domain-specific format) is returned in the mbuf passed in the *addr* parameter.

- **PRU_PEERADDR: Retrieve peer socket address** The address of the peer to which the socket is connected is returned. The socket must be in a connected state for this request to be made to the protocol. The address (in a domain-specific format) is returned in the mbuf pointed to by the *addr* parameter.

- **PRU_CONNECT2: Connect two sockets without binding addresses** The protocol module is supplied two sockets and is requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the *socketpair* system call.

Internal Requests

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are used solely for convenience in tracing a protocol's operation (e.g., the slow timeout request).

- **PRU_FASTTIMO: Service fast timeout** A fast timeout has occurred. This request is made when a timeout occurs in the protocol's *pr_fasttimo*() routine. The *addr* parameter indicates which timer expired.

- **PRU_SLOWTIMO: Service slow timeout** A slow timeout has occurred. This request is made when a timeout occurs in the protocol's *pr_slowtimo*() routine. The *addr* parameter indicates which timer expired.

- **PRU_PROTORCV: Receive data for protocol** This request is used between protocols, not by the socket layer; it requests reception of data destined for a protocol and not for the user. No protocols currently use this facility.

- **PRU_PROTOSEND: Send data to protocol** This request allows a protocol to send data destined for another protocol module, not for a user. The details of how data are marked *addressed-to-protocol* instead of *addressed-to-user* are left to the protocol modules. No protocols currently use this facility.

Protocol Control-Output Routine

A call on the control-output routine is of the form

```
error = (*pr->pr_ctloutput)(op, so, level, optname, mp);
    struct socket *so;
    int level, optname;
    struct mbuf **mp;
```

where *op* is PRCO_SETOPT when setting an option's value, and PRCO_GETOPT when getting an option's value. The *level* parameter indicates the layer of software that should interpret the option request. A *level* of SOL_SOCKET is specified to control an option at the socket layer. When the option is to be processed by a protocol module below the socket layer, *level* is set to the appropriate protocol number (the same number used in the *socket* system call.) Each level has its own set of option names; this name is interpreted only by the targeted layer of software. The last parameter is a pointer to a pointer to an mbuf chain; the preexisting chain contains an option's new value when setting, and the pointer is used to return an mbuf chain that contains an option's value when getting. Mbufs passed to the control-output routine when setting an option value must be freed by the protocol. When getting an option value, mbufs used to return an option value are allocated by the protocol and returned to the socket layer, where they are freed after data are copied to the user.

In supporting the *getsockopt* and *setsockopt* system calls, the socket layer always invokes the control-output routine of the protocol attached to the socket. To access lower-level protocols each control-output routine must pass control-output requests that are not for itself *downward* to the next protocol in the protocol hierarchy. Chapter 12 describes some of the options provided by the protocols in the Internet and XNS communication domains.

11.3 Protocol–Protocol Interface

The interface between protocol modules uses the *pr_usrreq*(), *pr_input*(), *pr_output*(), *pr_ctlinput*(), and *pr_ctloutput*() routines. The *pr_usrreq*() and *pr_ctloutput*() routines are used by the socket layer to communicate with protocols and have standard calling conventions. The remaining routines are not normally accessed outside of a protocol family, and therefore different calling conventions have evolved.

Although imposing a standard calling convention for all of a protocol's entry points might theoretically permit an arbitrary interconnection of protocol modules, it would be very difficult in practice. Crossing a protocol-family boundary would require a network address to be converted from the format of the caller's domain to the format of the callee's domain. Consequently, connection of protocols in different communication domains is not generally supported, and calling conventions for the routines listed in the preceding paragraph are typically standardized on a per-domain basis.

In the following sections, we briefly examine the general framework and calling conventions of protocols. In Chapter 12, we examine specific protocols to see how they fit into this framework.

pr_output

The protocol output routine often uses a calling convention designed to send a single message on a connection; for example,

```
error = (*pr_output)(pcb, m);
    struct pcb *pcb;
    struct mbuf *m;
```

Lower-level protocol output routines may not always have protocol control blocks, and thus may require more explicit parameters.

pr_input

Upper-level protocol input routines are usually called at software-interrupt level once the network-level protocol has located the protocol identifier. They are generally more stylized than are output routines because they are often called via a protocol switch. Depending on the nature of the protocol family, they may receive a pointer to a control block identifying the connection, or they may have to locate the control block from information in the received packet. A typical calling convention is

```
(void) (*pr_input)(m, ifp);
    struct mbuf *m;
    struct ifnet *ifp;
```

In this example, the interface from which the packet was received is passed as the second parameter, and the protocol does the connection-level demultiplexing.

pr_ctlinput

This routine is used to pass *control* information (i.e., information that might be passed to the user, but does not consist of data) *upward* from one protocol module to another. The common calling convention for this routine is

```
(void) (*pr_ctlinput)(req, addr);
    int req;
    struct sockaddr *addr;
```

The *req* parameter is one of the values shown in Table 11.4. The *addr* parameter is the address to which the condition applies. Many of the requests have been derived from the Internet Control Message Protocol (ICMP) [Postel, 1981], and from error messages defined in the 1822 host/IMP convention [BBN, 1978].

Table 11.4 Control-input routine requests.

Request	Description
PRC_IFDOWN	network interface transition
PRC_ROUTEDEAD	select new route if possible
PRC_MSGSIZE	message size forced packet to be dropped
PRC_HOSTDEAD	remote host is down
PRC_HOSTUNREACH	remote host is unreachable
PRC_UNREACH_NET	no route to network
PRC_UNREACH_HOST	no route to host
PRC_UNREACH_PROTOCOL	protocol not supported by destination
PRC_UNREACH_PORT	port number not in use at destination
PRC_UNREACH_NEEDFRAG	fragmentation needed but not allowed
PRC_UNREACH_SRCFAIL	source route failed
PRC_REDIRECT_NET	routing redirect for a network
PRC_REDIRECT_HOST	routing redirect for a host
PRC_REDIRECT_TOSNET	routing redirect for type of service and network
PRC_REDIRECT_TOSHOST	routing redirect for type of service and host
PRC_TIMXCEED_INTRANS	packet lifetime expired in transit
PRC_TIMXCEED_REASS	lifetime expired on reassembly queue
PRC_PARAMPROB	header-parameter problem detected

11.4 Protocol–Network-Interface Interface

The lowest layer in the set of protocols that constitutes a protocol family must interact with one or more network interfaces to transmit and receive packets. It is assumed that any routing decisions have been made before a packet is sent to a network interface; in fact, a routing decision is necessary to locate any interface at all unless one uses a single *hardwired* interface. There are two cases with which we should be concerned in the interaction between protocols and network interfaces: transmission of a packet and receipt of a packet. We shall consider each separately.

Packet Transmission

Assuming that a protocol has chosen an interface identified by *ifp*, a pointer to a network interface structure, the protocol transmits a fully formatted network-level packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst);
    struct ifnet *ifp;
    struct mbuf *m;
    struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the protocol address specified in *dst*, or returns an error indication (a UNIX error number). In reality, transmission may not be immediate or successful; typically, the output routine validates the destination address, queues the packet on its send queue, and primes an interrupt-driven routine to transmit the packet. For unreliable media, such as the Ethernet, *successful* transmission simply means that the packet has been placed on the cable without a collision. In contrast, an ARPANET IMP interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promise of delivery to the packets presented to a network interface, and thus corresponds most closely to the Ethernet. Errors returned by the output routine are only those that can be detected immediately and are normally trivial in nature (no buffer space, address format not handled, etc.). If errors are detected after the call has returned, the protocol is not notified.

In transmitting messages, each network interface usually must formulate a link-layer address for each outgoing packet.[2] The interface layer must understand each protocol address format that it supports to formulate corresponding link-layer addresses. The network layer for each protocol family selects a destination address for each message and then uses that address to select the appropriate network interface to use. This destination address is passed to the interface's output routine in the form of a *sockaddr* structure. Presuming that the address format is supported by the interface, the interface must map the destination protocol address into an address for the link-layer protocol associated with the transmission medium the interface supports. This mapping may be a simple algorithm, it may require a table lookup, or it may require more involved techniques such as using the Address Resolution Protocol described in Section 11.8.

Packet Reception

Network interfaces receive packets and dispatch them to the appropriate network-layer protocol according to information encoded in the link-layer protocol header. Each protocol family must have one or more protocols that constitute the network layer described in Section 11.1. In this system, each network layer protocol has an input-packet queue assigned to it. Incoming packets received by a network interface are queued in a protocol's input packet queue, and a software interrupt is posted to initiate network-layer processing (see Fig. 11.4). Similar queues are used to store packets awaiting transmission by network-interface modules.

Four macros are available for manipulating packet queues:

- **IF_ENQUEUE(ifq, m)** Place the packet *m* at the tail of the queue *ifq*.
- **IF_DEQUEUE(ifq, m)** Place a pointer to the packet at the head of queue *ifq* in *m* and remove the packet from the queue; *m* will be zero if the queue is empty.

[2] A link-layer address may not be required for a point-to-point link.

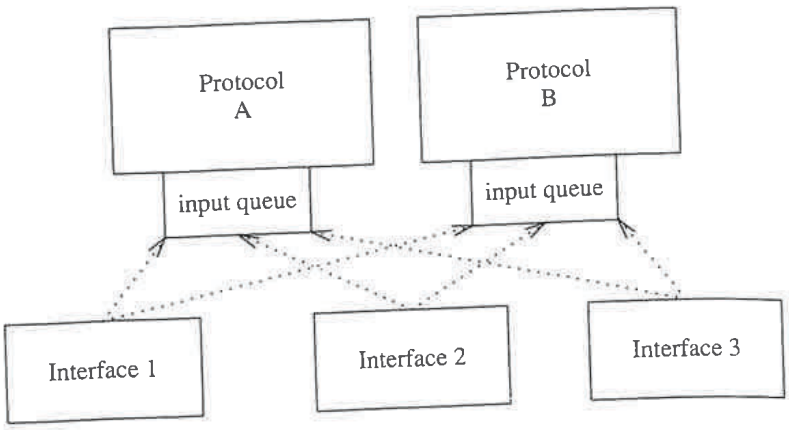


Figure 11.4 Input packets are dispatched to protocol input queues.

- **IF_DEQUEUEIF(ifq, m, ifp)** Remove the next packet from the head of a queue and return it in *m* and a pointer to the interface on which the packet was received in *ifp*.
- **IF_PREPEND(ifq, m)** Place the packet *m* at the head of the queue *ifq*.

Packet queues have a maximum length associated with them as a simple form of congestion control. The macro IF_QFULL() can be used to determine whether a queue is full, in which case another macro, IF_DROP(), can then be used to record the event in statistics kept for the queue. For example, the following code fragment is commonly found in a network interface's input routine:

```
if (IF_QFULL(inq)) {
    IF_DROP(inq);
    m_freem(m);        /* discard packet */
} else
    IF_ENQUEUE(inq, m);
```

On receiving a packet, a network interface decodes the packet type, strips the link-layer protocol header, attaches the identity of the receiving interface, and then dispatches the packet to the appropriate protocol. For example, packets are enqueued for the Internet domain with

```
schednetisr(NETISR_IP);   /* schedule IP input routine */
IF_ENQUEUE(&ipintr, m);   /* place message on IP's queue */
```

The *schednetisr*() macro marks a bit in a global status word and then posts a software interrupt. When the software interrupt occurs, the interrupt handler scans the status word, and for each preassigned bit that is set, invokes the associated protocol input routine. Bits in the status word are assigned according to the value of their protocol-family identifiers (see Table 11.5).

Table 11.5 Network-interrupt status-word bit assignments.

Status Bit	Value	Input Queue	Use
NETISR_RAW	PF_UNSPEC	*rawintrq*	unassigned/raw input
NETISR_IP	PF_INET	*ipintrq*	Internet IP protocol input
NETISR_IMP	PF_IMPLINK	*impintrq*	IMP/1822 error input
NETISR_NS	PF_NS	*nsintrq*	Xerox NS protocol input

Entries on a protocol's input queue contain both a packet and pointer to the network interface on which the packet was received. The pointer to the interface has many potential uses, such as deciding when to generate routing redirect messages. Input-handling routines that run at software-interrupt level are typically of the form

```
for (;;) {
    s = splimp();          /* block network from queue */
    IF_DEQUEUEIF(&xxintrq, m, ifp);
    splx(s);
    if (m == 0) break;     /* all packets processed */
    /* process packet and determine receiving protocol */
    (*pr_input)(m, ifp); /* invoke protocol */
}
```

While an entry is dequeued from an input queue, all network-interface input handling must be blocked by the processor's priority level being raised with *splimp*() to ensure that pointers in the queue data structure are not altered. Once a message is dequeued, it is processed and, if there is information in the packet for a higher-level protocol, the message is passed upward.

11.5 Routing

The networking system was designed for an internetwork environment in which a collection of local-area networks is connected at one or more points through hosts with multiple network interfaces (see Fig. 11.5). Hosts with multiple network interfaces, one on each local-area or long-haul network, are expected to act as gateways. In such an environment, issues related to gatewaying and packet-routing are important. Certain of these issues, such as congestion control, are handled in a simplistic manner (see Section 11.6). For others, the network system provides simple mechanisms on which more involved policies can be implemented. These mechanisms ensure that, as these problems become better understood, their solutions can be incorporated into the system.

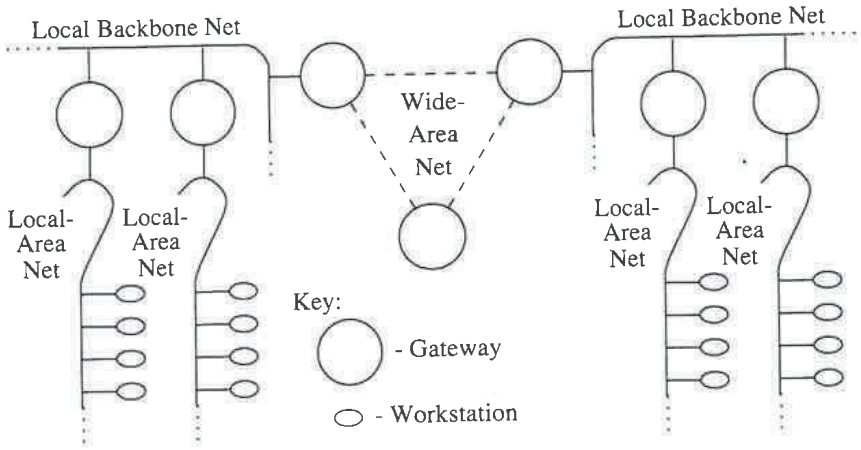


Figure 11.5 Example of the topology for which routing facilities were designed.

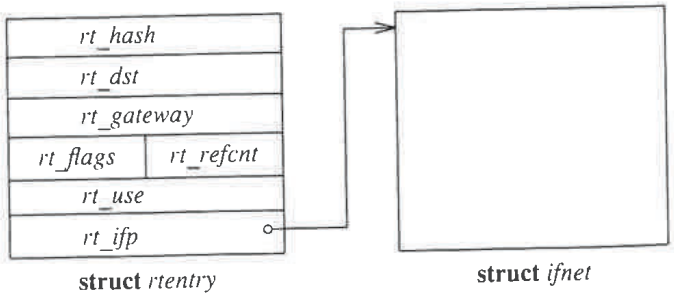
This section describes the facilities provided for message routing. Although there is nothing in the system that prevents protocols from managing their own routing information, the facilities described here were designed to support most needs. Note that the routing facilities included in the operating system do not impose *routing policies*, but instead support a *routing mechanism* by which externally defined policies can be implemented.

Routing Tables

The network system maintains a set of routing tables that is used by protocols in selecting a network interface to use in delivering a packet to its destination. These tables are composed of entries of the form shown in Fig. 11.6.

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. Hosts and networks are distinguished so that a single mechanism can be employed for broadcast and multidrop networks, and for networks built from point-to-point links (e.g., DECnet [DEC, 1980]).

Figure 11.6 Structure of a routing-table entry.



Each table is organized as a hashed set of linked lists. Two hash values are calculated by routines defined for each address family, one for the destination host and one for the network of the destination host. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash-table size), and the entire value is used as a key in scanning the list of routes. Lookups are done first in the routing table for hosts, then in the routing table for networks. If both lookups fail, a final lookup is made for a *wildcard route* (by convention, *network* 0). The first appropriate route discovered is used. Use of this algorithm allows for the presence of specific routes to hosts, as well as routes to the network in general. Also, a fallback network route can be defined to an *intelligent gateway* that can then perform more informed routing decisions.

Each routing-table entry contains a destination (the desired final destination), a gateway to which to send the packet, and flags that indicate the route's status and type (host or network). A count of the number of packets sent using the route also is kept, along with a count of references held to the dynamically allocated structure to ensure that memory reclamation occurs only when the route is no longer referenced. Finally, a pointer to a network interface is kept; packets sent using the route are passed to this interface's output routine.

Routes are typed in two ways: as either *host* or *network*, and as either *direct* or *indirect*. The host–network distinction determines how to compare the destination address during lookup. If the route is to a network, only the packet's destination *network* is compared to that of the entry stored in the table. Otherwise, the route is to a host and the addresses must match *completely*.

The distinction between direct and indirect routes indicates whether the destination is directly connected to the source. This distinction is needed when the link-layer encapsulation is done. If a packet is destined for a peer at a host or network that is not directly connected to the source, the internetwork packet header will contain the address of the eventual destination, whereas the link-layer protocol header will address the intervening gateway. If the destination is directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag in a routing-table entry indicates that the route is to an *indirect* gateway agent, and that the link-layer header should be filled in from the *rt_gateway* field instead of from the final internetwork destination address.

It is assumed that multiple routes to the same destination will not be present. If more than one route is present, only the most recently installed route is used.

Routing Redirects

A *routing redirect* is a control request from a protocol to the routing system to modify an existing routing-table entry, or to create a new routing-table entry. Protocols usually generate such requests in response to routing-redirect messages they receive from gateways. Gateways generate routing-redirect messages when they recognize that a better route exists for a packet that they have been asked to forward. For example, if two hosts A and B are on the same network, and host A

sends a packet to host B via a gateway C, then C will send a routing-redirect message to A indicating that A should send packets to B directly.

On hosts where exhaustive routing information is too expensive to maintain (e.g., workstations), the combination of wildcard routing entries and routing-redirect messages can be used to provide a simple routing-management scheme without the use of a higher-level policy process. Current connections may be rerouted after notification of the protocols by means of the protocols' *pr_ctlinput*() entries. Statistics are kept by the routing-table routines on the use of routing-redirect messages and on the latter's effect on the routing tables.

Routing-Table Interface

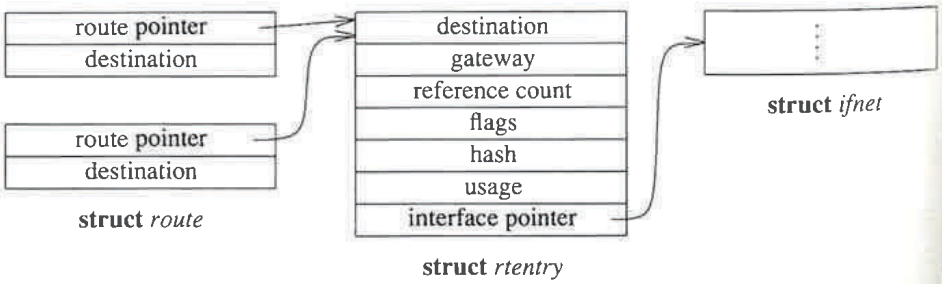
A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing-redirect control message. The routine *rtalloc*() allocates a route; it is called with a pointer to a *route* structure that contains the desired destination (see Fig. 11.7). The route returned is assumed to be *held* by the caller until released with a call to *rtfree*(). Protocols that implement virtual circuits, such as Transmission Control Protocol (TCP), hold on to routes for the duration of the circuit's lifetime, whereas connectionless protocols, such as User Datagram Protocol (UDP), allocate and free routes whenever the routes' destination address changes.

The routine *rtredirect*() is called to process a routing-redirect control message. It is called with a destination address, the new gateway to that destination, and the source of the redirect. Redirects are accepted from only the current router for the destination. If a nonwildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing-table entry is inserted that reflects the information supplied. Routes to interfaces and routes to gateways that are not directly accessible from the host are ignored.

User-Level Routing Policies

The kernel routing facilities deliberately refrain from making policy decisions. Instead, routing policies are determined by user processes, which then add or

Figure 11.7 Data structures used in allocating routes.



delete entries in the kernel routing tables through two *ioctl* system calls. The decision to place policy decisions in a user process implies that routing-table updates may lag a bit behind the identification of new routes, or the failure of existing routes. This period of instability is normally short, however, provided that the routing process is implemented properly. Internet-specific advisory information, such as ICMP error messages and IMP diagnostic messages, may also be read from raw sockets (described in Section 11.7).

Several routing-policy processes have been implemented. The system standard *routing daemon* uses a variant of the Xerox NS Routing Information Protocol [Xerox, 1981] to maintain up-to-date routing tables in a local environment. Interaction with other existing routing protocols, such as the Internet Exterior Gateway Protocol (EGP), has been accomplished using a similar scheme.

11.6 Buffering and Congestion Control

One of the major factors affecting the performance of a protocol is the buffering policy. Lack of a proper buffering policy can force packets to be dropped, cause false windowing information to be emitted by protocols, fragment host memory, and degrade the overall host performance. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for *normal* network operation.

The 4.3BSD networking system is not dramatically different in this respect. At boot time, a small, fixed amount of memory is allocated by the networking system. At later times, more system memory may be requested as the need arises; at no time, however, is memory ever returned to the system. It would be possible to reclaim memory from the network; however, for this memory reclamation to be performed safely, some portion of the network would have to be disabled while data structures were updated. In the environments where the system has been used, storage use has not been an issue and thus storage reclamation has been left unimplemented.

Protocol Buffering Policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket-creation time. These amounts define the high and low watermarks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory-management routines.

Protocols that provide connection-level flow control base their decisions on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, whereas receive windows are adjusted based on the number of data awaiting transmission in the send queue.

Queue Limiting

Incoming packets from the network are always received unless memory allocation fails. However, each network-layer protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance, if the host is acting as a gateway from a high-bandwidth network to a low-bandwidth network). As a *defense mechanism*, the queue limits can be adjusted to throttle network-traffic load on a host. Discarding packets is not always a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism. On the other hand, limiting *output* queue lengths can be valuable on hosts that gateway traffic from a high-bandwidth network to a low-bandwidth network.

11.7 Raw Sockets

A *raw socket* allows privileged users direct access to a protocol. Raw sockets are intended for knowledgeable processes that wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of protocols built atop existing protocols. For example, the Internet EGP [Mills, 1984] is implemented as a user-level process utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, possibly augmented by protocol-specific processing routines. The next sections describe only the core of the raw socket interface; details specific to particular protocols are not discussed.

Control Blocks

Every raw socket has a protocol control block of the form shown in Fig. 11.8. All control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The *rcb_proto* field contains the protocol family and protocol number with which the raw socket is associated. The protocol, family, and addresses are used to filter packets on input as described later. If any protocol-specific information is required, it can be attached to the control block through the *rcb_pcb* field. Protocol-specific options to be included in outgoing packets may be held with the *rcb_options* field.

A raw socket interface is datagram-oriented; each send or receive on the socket requires a destination address. Destination addresses may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present in the control block by inspection, two flags are used to indicate whether a local and foreign address are present. If routing is necessary, it must be performed by an underlying protocol.

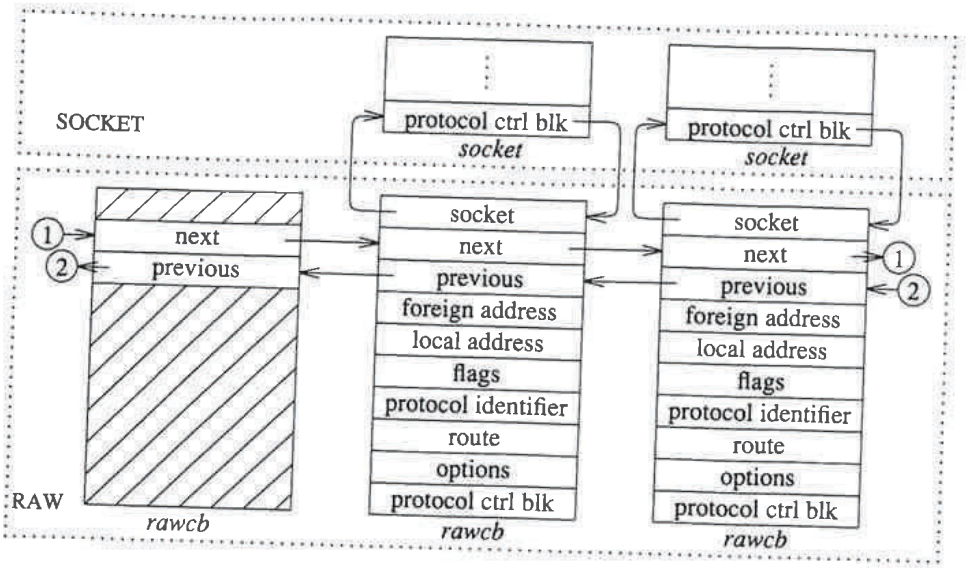


Figure 11.8 Raw socket control block.

Input Processing

Input packets are assigned to raw sockets based on a simple pattern-matching scheme. Each network interface or protocol gives unassigned packets to the raw input routine with the call[3]

```
raw_input(m, proto, src, dst)
    struct mbuf *m;
    struct sockproto *proto;
    struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form shown in Fig. 11.9, and it is placed in a packet queue for the raw input-protocol module. Packets taken from this queue are copied into all raw sockets that match the header according to the following rules:

1. The protocol family of the socket and header agree.
2. If the protocol number in the socket is nonzero, then it agrees with that found in the packet header.
3. If a local address is defined for the socket, the address format of the socket's local address is the same as the packet's destination address and the two addresses agree exactly.
4. Rule 3 is applied to the socket's foreign address and the packet's source address.

[3] In 4.3BSD, network interfaces do not pass unassigned packets to the raw input routine.

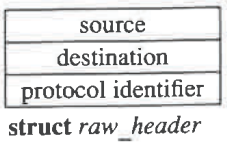


Figure 11.9 Raw input packet header.

A basic assumption in the pattern-matching scheme is that addresses present in the control block and packet header (as constructed by the network interface and any raw input-protocol module) are in a canonical form that can be compared on a bit-for-bit basis.

Output Processing

On output, each send request results in a call to the raw socket's user request routine; this routine is common to all raw sockets. The user request routine passes the packet and a pointer to the raw control block to a protocol-specific output routine where any necessary processing is done before the packet is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

11.8 Additional Network Subsystem Topics

In the following sections, we shall discuss several aspects of the network subsystem that are not easy to categorize.

Out-of-Band Data

The ability to process out-of-band data is a facility specific to the stream-socket abstraction. Little agreement appears to exist as to what out-of-band data's semantics should be. TCP defines a notion called *urgent data*, in which in-line data are marked for *urgent delivery*; the NBS protocols [Burruss, 1980] and numerous other protocols provide a fully independent logical transmission channel along which out-of-band data are sent. In addition, the number of the data that can be sent in an out-of-band message varies from protocol to protocol, from 1 bit to 16 bytes or more.

A stream socket's notion of out-of-band data has been defined as the lowest reasonable common denominator. Out-of-band data are expected to be transmitted out of the normal sequencing and flow-control constraints of the data stream. A minimum of 1 byte of out-of-band data and one outstanding out-of-band message is expected to be provided by protocols supporting a stream socket. It is a protocol's prerogative to support larger-sized messages, or more than one outstanding out-of-band message at a time.

Out-of-band data are maintained by the protocol and usually are not stored in

the socket's receive queue. A socket-level option, SO_OOBINLINE, is provided to force out-of-band data to be placed in the normal receive queue when urgent data are received. Placement of out-of-band data in the normal data stream can permit a protocol to hold several out-of-band messages simultaneously. This mechanism can avoid the loss of out-of-band messages caused by a user who responds slowly.

Address Resolution Protocol

The Address Resolution Protocol (ARP) is a link-level protocol that provides a dynamic address-translation mechanism for networks that support broadcast or multicast communication [Plummer, 1982]. ARP is used in 4.3BSD to map 32-bit Internet addresses to 48-bit Ethernet addresses. Although ARP is not specific either to Internet protocol addresses or to Ethernet, the 4.3BSD network subsystem supports only that combination. ARP is incorporated into the network-interface layer, although it logically sits *between* the network and network-interface layers.

The general idea of ARP is simple. A table of translations from network addresses to link-layer addresses is maintained. When an address-translation request is made to the ARP service by a network interface and the requested address is not in ARP's table of known translations, an ARP message is created that specifies the requested network address and an unknown link-layer address. This message is then broadcast by the interface in the expectation that one of the hosts attached to the network will know the translation—usually because the host is the intended target of the original message. If a response is received in a timely fashion, the ARP service uses the response to update its translation tables and to resolve the pending request, and the requesting network interface is then called to transmit the original message.

In practice, the simplicity of this algorithm is complicated by the necessity to minimize resources utilized by the name-translation tables and the network interfaces, to avoid stale translation data, and to deal with failed translation requests. The ARP translation tables are implemented as a cache of translations. Each entry in the cache maps a single Internet network address to its corresponding Ethernet address and has a timer associated with it. The entries are organized in a hash structure to speed lookups. A request is made to resolve an Internet address to an Ethernet address for an outgoing message by the call

```
result = arpresolve(ac, m, destip, desten, usetrailers);
    struct arpcom *ac;
    struct mbuf *m;
    struct in_addr *destip;
    u_char *desten;
    int *usetrailers;
```

ARP first checks its tables to see whether the Internet address, *destip*, is already present. If the address is known and the entry is *complete* (i.e., it is not waiting for a response), the Ethernet address is returned in *desten*. Otherwise, if the address was not found, ARP must allocate a new entry in the table, queue the

outgoing message for future transmission, and broadcast a message requesting the Internet address translation. At the same time, a timer associated with the entry is initialized to ensure that the mbuf chain in which the message is stored will be reclaimed if no response is received for the translation. If another translation request is made before a reply is received, the queued message is discarded and only the newer one is saved.

At a later time—preferably before the timer has expired on the queued message—ARP will receive a response to its translation request. The received message is processed first by the *arpinput*() routine, as invoked from each network-interface device that supports ARP, and the ARP packet is processed to locate the translation entry. If the message completes a pending translation, the entry is updated and the original message is passed back to the network interface for transmission. This time, the resultant call to *arpresolve*() will succeed without delay.

ARP input handling must cope with requests for the host's own address as well as responses to translation requests the host generated. The input module also watches for responses from other hosts that advertise a translation for its own Internet address. This monitoring is done to ensure that no two hosts on the same network believe they have the same Internet address (although this error may be detected, ARP's only recourse is to log a diagnostic message).

ARP normally times out completed translation entries in its cache after 20 minutes and incomplete translation entries after 3 minutes. Entries may be marked *permanent*, however, in which case they are never removed from the cache. Entries may also be marked *published*, allowing one host to act as a surrogate for other hosts that do not support ARP.

The use of ARP is important to the smooth operation of an Ethernet network that uses the Internet protocols. However, it can pose compatibility problems when hosts that do not support ARP are also present. Published entries can be used for hosts that do not support ARP, but if these hosts do not support a complete 32-bit to 48-bit mapping between Internet and Ethernet addresses, this is insufficient. Fortunately, most Internet hosts now support ARP.

VAX UNIBUS Interfaces

All VAX hardware-based interfaces supported by 4.3BSD reside on the UNIBUS. UNIBUS interfaces fall into two categories, according to whether host–device communication is by DMA or through shared UNIBUS memory. For devices that support DMA, a common set of utility routines for dealing with the UNIBUS is used; their operation is described here.

Each network device has at least two packet buffers for transmitting and receiving messages. All DMA transfers pass through a UNIBUS adapter to the UNIBUS on which the hardware device is located. The UNIBUS I/O mapping registers must be set up to map an I/O buffer's memory pages into a UNIBUS's address space (see Fig. 11.10).

Each UNIBUS-based interface utilizes an *if_ubinfo* structure that contains the general information about the device's I/O buffers on one UNIBUS. In addition,

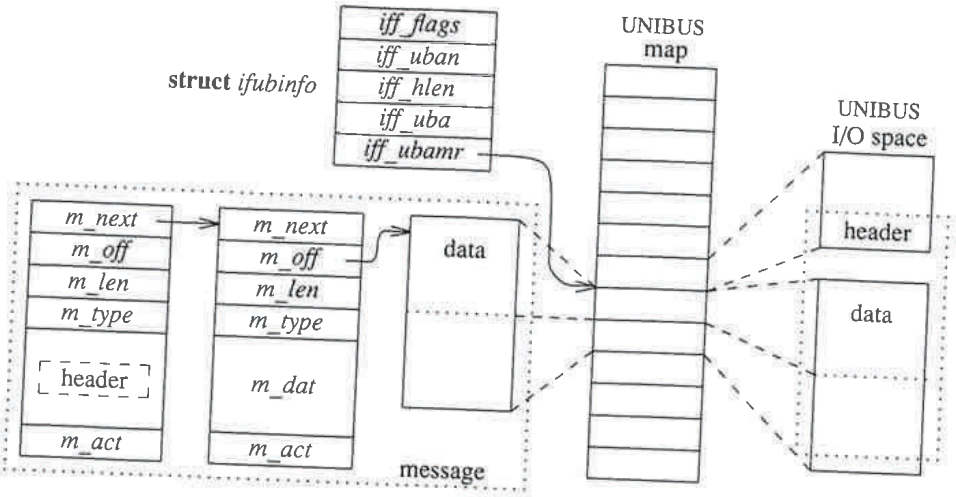


Figure 11.10 Mapping a message from VAX to UNIBUS address spaces.

there is a structure describing each buffer, including UNIBUS resources held by the interface to map the buffer into the UNIBUS's I/O space. At system initialization, enough memory pages and UNIBUS map registers are allocated for each buffer to map a full-sized packet and header into the UNIBUS address space. The VAX virtual address of the buffer is held in *ifrw_addr*, and the map registers begin at *ifrw_mr*. UNIBUS map register *ifrw_mr*[−1] maps the link-layer protocol header ending on a page boundary (for the benefit of the trailer encapsulation, described later). Additional structures are associated with each receive and transmit buffer—normally one each per interface.

When transmitting a message, the interface software must transfer the contents of an mbuf chain to a region of memory that is mapped contiguously in the UNIBUS address space. It effects this transfer by copying the data or, when possible, by setting up the UNIBUS adapter's mapping registers to map data pages held by mbufs into the appropriate region of the UNIBUS address space. In the latter case, map registers that normally map pages of the transmit buffer are changed to point at the mbuf data pages temporarily. The map registers are restored to reference the reserved pages of the transmit buffer when they are next used. The mbufs containing pages mapped for transfer are placed on a queue to be freed after transmission. Mbufs whose contents were copied are reclaimed immediately.

When a message that has been transferred into VAX memory is received, a similar process must be performed. If a transfer contains at least one-half of a page of data, page frames are allocated from the mbuf page pool and are traded with the pages already containing the data. These new pages are mapped into the receive buffer to replace the pages just placed in mbufs, making the device ready for the next UNIBUS transfer.

The remapping of pages improves the network performance on machines where the CPU overhead for copying memory is higher than that for remapping pages of memory. This remapping is possible only when the message sizes are large enough to permit such operations (most commonly, when transmissions utilize a local-area network, such as the Ethernet). The use of a trailer encapsulation also interacts well with this scheme, potentially permitting data to be transferred from device input to a user's address space without copying.

Trailer Protocols

Memory-to-memory copies can represent a significant portion of the overhead associated with network data-transport services [Watson & Mamrak, 1987]. Consequently, the designers devoted a great deal of effort to minimizing such operations in the design of the network software. Many architectures provide virtual-memory hardware organized in page-sized units and permit I/O transfers to be mapped on a page basis. An attractive scheme for reducing copy operations is to keep data in page-sized units on page-aligned boundaries, so that the system can move data in memory by altering page-table entries rather than by physically copying the data. The socket software attempts to place user data in mbufs that have pages associated with them whenever possible (see Section 10.6). VAX UNIBUS devices that support DMA utilize routines that remap pages into the UNIBUS address space in preparation for DMA whenever possible. This potentially permits data transmission to be accomplished with only a single copy operation. Data must be copied into the kernel's address space under the existing memory-management scheme; the copying is not necessary in some other systems to which the networking software has been ported.

Unfortunately, performing a similar operation when receiving data from the network is difficult. Consider the format of an incoming packet. A packet typically contains a link-level protocol header followed by one or more headers used by higher-level protocols. Finally, the data, if there are any, follow these headers. As the header information may be variable in length, ensuring that data destined for a user's receive buffer are properly aligned in memory is impossible without a priori knowledge of the format of the protocol headers (e.g., the protocol header length would be known if the system supported only a single protocol header format).

To allow a variable-length protocol header to be present and still to ensure page alignment of data, a special local network encapsulation can be used. This encapsulation, termed a *trailer protocol* [Leffler & Karels, 1984], places variable-length header information after the data (see Fig. 11.11). The link-layer header contains a new trailer-specific protocol-type value that encodes the size of the data portion, and a new *trailer-protocol header* is inserted before the variable-length header information. The trailer protocol contains the original packet type and the size of the protocol information that follows it. If the link-layer protocol header is of fixed size (as is usually the case), input DMA may be offset by the size of this header to ensure that data land in a page-aligned region of memory.

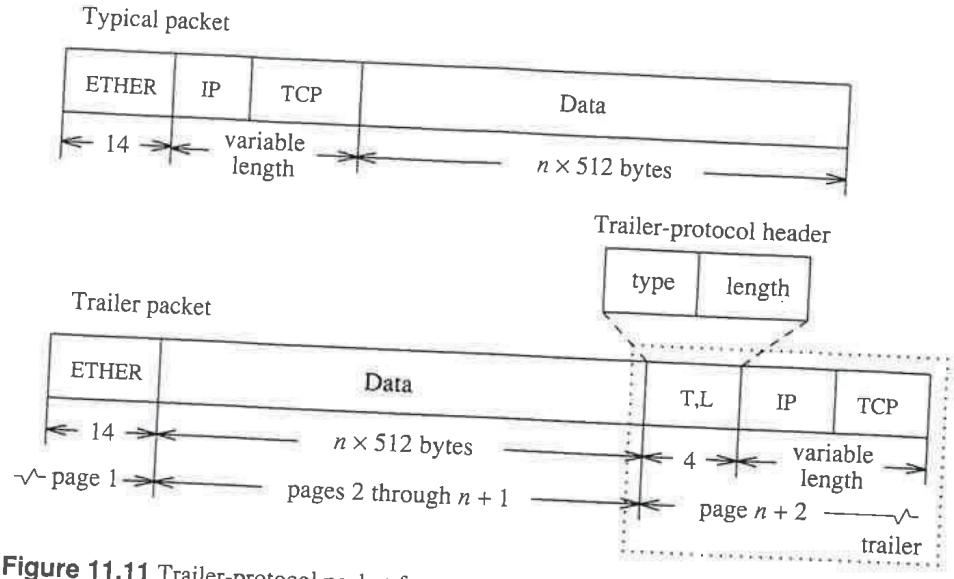


Figure 11.11 Trailer-protocol packet format.

The processing of a trailer protocol is simple. On output, the packet type in the link-layer header is set to indicate that a trailer encapsulation is being used. This type is selected from a set of types reserved for trailers according to the number of data pages in the message that precede the trailer-protocol header. The trailer-protocol header is initialized to contain the actual packet identifier and the size of the variable-length header, and is appended to the data along with the variable-length header information.

On input, the interface routines identify the trailer encapsulation by the packet type in the link-layer header and calculate the number of pages of data to find the beginning of the trailer. The trailing-protocol header information is then copied into a separate mbuf, and is linked to the front of the resultant packet. The data portion is left in the pages of the receive buffer, and new pages are allocated for the receive buffer.

Trailer protocols require cooperation between source and destination hosts. To permit trailers to be negotiated on a host-by-host basis, the ARP service includes negotiation of the use of trailer protocols. During the normal exchange of ARP request and response packets between a pair of hosts, each host may send an additional ARP response for the trailer-protocol type, indicating a desire to receive trailer-encapsulated packets.

For network interfaces that reside on an Ethernet and use ARP, each call to resolve an Internet address returns an indication of whether the receiving host wishes to receive trailer protocols. The system will send trailer encapsulations only if the host has indicated a willingness to receive them. Trailer-encapsulation packets are always accepted on networks that do not use ARP to negotiate the use of trailers; if hosts on a network are incapable of receiving trailers, a flag, IFF_NOTRAILERS, may be set in the interface structure to disable the use of trailers.

Exercises

11.1 Name two key data structures used in the networking subsystem that are important in ensuring that the socket-layer software is kept independent of the networking implementation.

11.2 Why are software interrupts used to trigger network protocol processing on receipt of data, rather than the protocol processing being encapsulated in separate processes?

11.3 What routines in the protocol switch are called by the socket layer? Explain why each of these routines is called.

11.4 Assume that a *reliably delivered message socket* (SOCK_RDM) is a connectionless socket that guarantees reliable delivery of data and that preserves message boundaries. Which flags would a protocol that supported this type of socket have set in the *pr_flags* field of its protocol-switch entry?

11.5 Give an example of a network interface that is useful without an underlying hardware device.

11.6 Give two reasons why the addresses of a network interface are *not* in the network-interface data structure.

11.7 Why is the name or address of a socket kept at the network layer and not at the socket layer?

11.8 Why does 4.3BSD not attempt to enforce a rigid protocol–protocol interface structure?

11.9 Describe two tasks performed by a network-interface output routine.

11.10 Why is the identity of the network interface on which each message is received passed upward with the message?

11.11 What routing *policies* are implemented in the kernel?

11.12 Why are two separate tables—one for hosts and one for networks—used in the routing facilities?

11.13 What routing facility is designed mainly to support workstations?

11.14 What is a routing redirect? For what is it used?

11.15 Why do the output-packet queues for each network interface have limits on the number of packets that may be queued?

11.16 What does the SO_OOBINLINE socket option do? Why does it exist?

*11.17 Explain why it is impossible to use the raw socket interface to support parallel protocol implementations—some in the kernel and some in user mode. What modifications to the system would be necessary to support this facility?

*11.18 Why are access rights provided to the user request routine at the same time as any associated data are provided, instead of being sent in a separate call?

*11.19 What is a trailer protocol? Should the use of trailer protocols improve the performance of a system that is I/O limited (e.g., a system with a slow hardware controller)? What about a system that is CPU-bound in its protocol processing?

References

BBN, 1978.

BBN, "Specification for the Interconnection of Host and IMP," Technical Report 1822, Bolt, Beranek, and Newman, Cambridge, MA (May 1978).

Burruss, 1980.

J. Burruss, "Features of the Transport and Session Protocols," Report No. ICST/HLNP-80-1, National Bureau of Standards, Washington, D.C. (March 1980).

DEC, 1980.

DEC, "DECnet DIGITAL Network Architecture—General Description," Report No. AA-K179A-TK, Digitial Equipment Corporation, Maynard, MA (October 1980).

ISO, 1984.

ISO, "Open Systems Interconnection—Basic Reference Model," ISO 7498, International Organization for Standardization (1984). Available from: American National Standards Institute, 1430 Broadway, New York, NY 10018.

Leffler & Karels, 1984.

S. J. Leffler & M. J. Karels, "Trailer Encapsulations," RFC 893, SRI Network Information Center, Menlo Park, CA (April 1984).

Mills, 1984.

D. L. Mills, "Exterior Gateway Protocol Formal Specification," RFC 904, SRI Network Information Center, Menlo Park, CA (April 1984).

Plummer, 1982.

D. Plummer, "An Ethernet Address Resolution Protocol," RFC 826, SRI Network Information Center, Menlo Park, CA (September 1982).

Postel, 1981.

J. Postel, "Internet Control Message Protocol," RFC 792, SRI Network Information Center, Menlo Park, CA (September 1981).

Watson & Mamrak, 1987.

R. W. Watson & S. A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems* **5**(2), pp. 97–120 (May 1987).

Xerox, 1981.

Xerox, "Internet Transport Protocols," Xerox System Integration Standard 028112, Xerox Corporation, Stamford, CT (December 1981).

CHAPTER 12

Network Protocols

Chapter 11 presented the network communications architecture of 4.3BSD. In this chapter, we examine the network protocols implemented within this framework. The 4.3BSD system supports three communication domains: DARPA Internet, Xerox Network Systems (NS), and UNIX. The UNIX domain does not include network protocols, as it operates entirely within a single system. The DARPA Internet protocol suite was the first set of protocols implemented within the network architecture of 4.2BSD. Following the release of 4.2BSD, several proprietary protocol families were implemented by vendors within the network architecture. However, it was not until the addition of the Xerox NS protocols in 4.3BSD that the system's ability to support multiple network protocol families was visibly demonstrated. Although some parts of the protocol interface were previously unused and thus unimplemented, the changes required to add a second network protocol family did not substantially modify the network architecture.

In this chapter, we shall concentrate on the organization and implementation of the DARPA Internet protocols. This protocol implementation is widely used, both in 4BSD systems and in many other systems, because it was publicly available when many vendors were looking for tuned and reliable communication protocols. Developers have implemented other protocols, including Xerox NS, by following the same general framework set forth by the Internet protocol routines. After describing the overall architecture of the Internet protocols, we shall examine their operation according to the structure defined in Chapter 11. We shall also describe the significant algorithms used by the Internet protocols. We then shall present an overview of the Xerox NS protocols and their implementation.

12.1 DARPA Internet Network Protocols

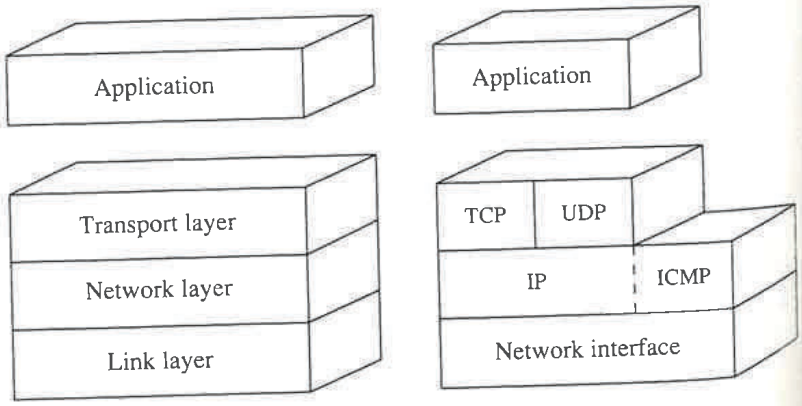
The DARPA Internet network protocols were developed under the sponsorship of DARPA, for use on the ARPANET [McQuillan & Walden, 1977; DARPA, 1983].

They are commonly known as TCP/IP, although TCP and IP are only two of the many protocols in the family. Unlike earlier protocols used within the ARPANET (the ARPANET Host-to-Host Protocol, sometimes called NCP, Network Control Program [Carr *et al.*, 1970]), these protocols do not assume a reliable subnetwork that ensures delivery of data. Instead, the Internet protocols were devised for a model in which hosts were connected to networks with varying characteristics, and the networks were interconnected by gateways. Such a model is called a *catenet* [Cerf, 1978]. The Internet protocols are designed for packet-switching networks ranging from the ARPANET, which provides reliable message delivery or notification of failure, to pure datagram networks such as Ethernet that provide no indication of datagram delivery.

This model leads to the use of at least two protocol layers. One layer operates end-to-end between two hosts involved in a conversation. It is based on a lower-level protocol that operates on a hop-by-hop basis, forwarding each message through intermediate gateways to the destination host. In general, there exists at least one protocol layer above the other two, which is the application layer. This three-level layering has been called the ARPANET Reference Model [Padlipsky, 1985]. The three layers correspond roughly to levels 3 (network), 4 (transport), and 7 (application) in the ISO Open Systems Interconnection reference model [ISO, 1984].

The Internet communications protocols that support this model have the layering illustrated in Fig. 12.1. The Internet Protocol, IP, is the lowest-level protocol in the ARPANET Reference Model; this corresponds to the ISO network layer. IP operates hop by hop as a datagram is sent from the originating host to its destination via any intermediate gateways. It provides the network-level services of host addressing, routing and, if necessary, packet fragmentation and reassembly if intervening networks cannot send an entire packet in one piece. All the other protocols use the services of IP. The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are transport-level protocols that provide

Figure 12.1 Internet protocol layering. TCP—Transmission Control Protocol; UDP—User Datagram Protocol; IP—Internet Protocol; ICMP—Internet Control Message Protocol.



additional facilities to IP. Each protocol adds a port identifier to IP's host address so that local and remote sockets can be identified. TCP provides reliable, unduplicated, and flow-controlled transmission of data; it supports the stream socket type in the Internet domain. UDP provides an optional data checksum for checking integrity in addition to a port identifier, but otherwise adds little to the services provided by IP. UDP is the protocol used by datagram sockets in the Internet domain. The Internet Control Message Protocol (ICMP) is used for error reporting and for other network-management tasks; it is logically a part of IP, but like the transport protocols is layered above IP. It is usually not accessed by users. Raw access to the IP and ICMP protocols is possible through *raw sockets*; see Section 11.7 for information on this facility.

The Internet protocols were designed to support heterogeneous host systems and architectures. These systems use a wide variety of internal data representations. Even the basic unit of data, the *byte*, was not the same on all host systems; one common type of host supported variable-sized bytes. The network protocols, however, require a standard representation. This representation is expressed in terms of the *octet*, an eight-bit byte. We shall use this term as it is used in the protocol specifications to describe network data, although we continue to use the term *byte* to refer to data or storage within the system. All fields in the Internet protocols that are larger than an octet are expressed in *network byte order*, with the most significant octet first. The 4.3BSD network implementation uses a set of routines or macros to convert short and long integer fields between host and network byte order on hosts (such as the VAX) that have a different native ordering.

Internet Addresses

An *Internet host address* is a 32-bit number that identifies both the network on which a host is located and the host on that network. Network identifiers are assigned by a central agency, whereas host identifiers are assigned by each network's administrator. It follows that a host with network interfaces attached to multiple networks has multiple addresses. Figure 12.2 shows the original addressing scheme that was tied to the subnetwork addressing used on the ARPANET: Each host was known by the number of the ARPANET IMP to which it was attached and by its host port number on that IMP (Interface Message Processor; see Section 12.9). The IMP and host numbers each occupied one octet of the address. One of the remaining two octets was used to designate the network, and the other was available for uses such as multiplexed host connections—thus the name logical host. This encoding of the address limits the number of networks to 255, a number that quickly proved to be too small. Figure 12.2 shows how the network portion of the address was encoded such that it could be variable in size. The most significant bits of the network part of the address determine the class of an address. Three classes are defined, A, B and C, with high-order bits of 0, 10, and 110;[1] they use 8, 16, and 24 bits, respectively, for the network part of the

[1] A fourth class, D, has been defined with high-order bits 1110. It is used in experimental multicast facilities.

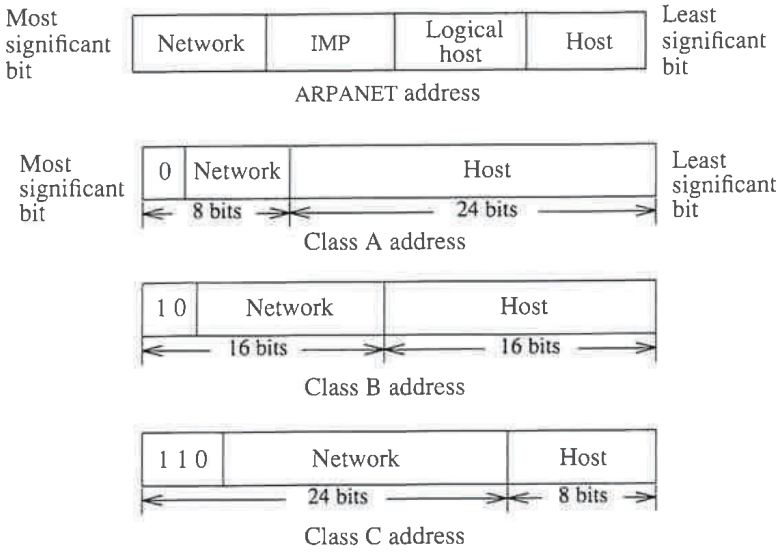


Figure 12.2 Internet addresses. IMP—Interface Message Processor.

address. Each class has fewer bits for the host part of each address and thus supports fewer hosts than do the higher classes. This form of frequency encoding supports a larger number of networks of varying size yet is compatible with the old encoding of ARPANET addresses.

Subnets

The basic Internet addressing scheme uses a 32-bit address that contains both a network and host identifier. All interconnected networks must be known to a central collection of routing agents for full connectivity. This scheme does not handle a large number of interconnected networks well because of the large amount of routing information necessary to ensure full connectivity. Furthermore, when networks are installed at a rapid pace, the administrative overhead is significant. However, many networks are installed at organizations such as universities, companies, and research centers that have a large number of interconnected local-area networks with only a few points of attachment to external networks. To handle these problems, the notion of a *subnet* addressing scheme was added [Mogul & Postel, 1985] that allows a collection of networks to be known by a single network number.

Subnets allow the addition of another level of hierarchy to the Internet address space. They partition a network assigned to an organization into multiple address spaces (see Fig. 12.3). This partitioning, each part of which is termed a *subnet*, is visible to only those hosts and gateways on the subnetted network. To hosts that are not on the subnetted network, the subnet structure is not visible. Instead, all hosts on subnets of a particular network are perceived externally as being on a single network. The scheme allows Internet routing to be performed

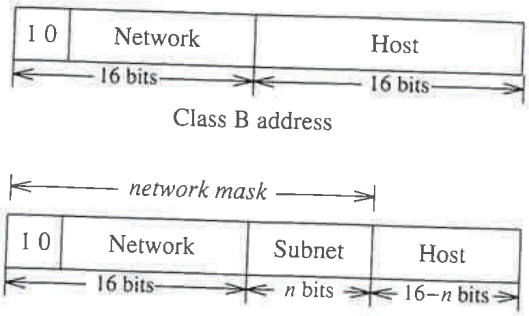


Figure 12.3 Subnet address partitioning.

on a site-by-site basis, as all hosts on a site's subnets appear to off-site hosts and gateways to be on a single Internet network. This partitioning scheme also permits sites to have greater local autonomy over the network topology at their site.

When a subnet addressing scheme is set up at a site, a partitioning of the assigned Internet address space for that site must be chosen. Consider Fig. 12.3. If a site has a class B network address assigned to it, it has 16 bits of the address in which to encode a subnet number and the identifier of a host on that subnet. An arbitrary subdivision of the 16 bits is permitted, but sites must balance the number of subnets they will need against the number of hosts that may be addressed on each subnet. To inform the system of the desired partitioning scheme, the site administrator specifies a *network mask* for each network interface. This mask shows which bits in the Internet address to include when extracting a network identifier from a local address. The mask includes the normal network portion as well as the subnet field. This mask also is used when the host part of an address is extracted. When interpreting an address that is not local, the system uses the mask corresponding to the class of the address.

The implementation of subnets is isolated, for the most part, to the routines that manipulate Internet addresses. Each Internet address assigned to a network interface is maintained in an *in_ifaddr* structure that contains an interface address structure and additional information for use in the Internet domain (see Fig. 12.4). When an interface's network mask is specified, it is recorded in the *ia_subnetmask* field of the address structure. The network mask, *ia_netmask*, is calculated based on the type of the network number (class A, B, or C) when the interface's address is assigned. For nonsubnetted networks, the two masks are identical. The system then interprets local Internet addresses using these values. An address is considered to be local if the field under the network mask matches the network field of an interface address. Thus, the subnet mask is assumed to be uniform throughout the subnetted network.

Broadcast Addresses

On networks capable of supporting broadcast datagrams, 4.2BSD used the address with a host part of zero for broadcasts. After 4.2BSD was released, the Internet broadcast address was defined as the address with a host part of all ones [Mogul,

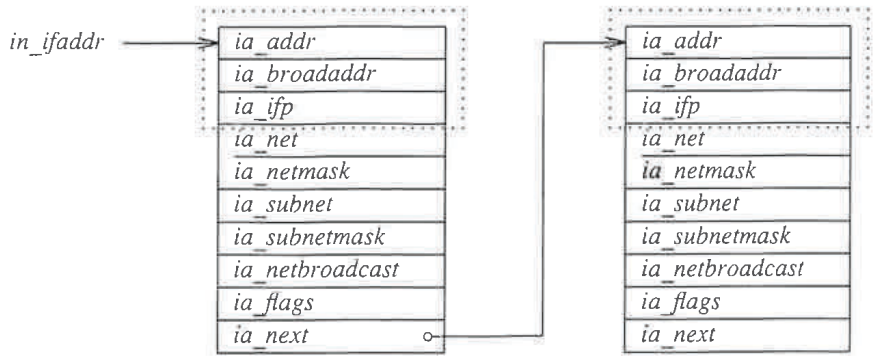


Figure 12.4 Internet interface address structure.

1984]. This change and the introduction of subnets both complicate the recognition of broadcast addresses. Hosts may use a host part of zero or ones to signify broadcast, and some may understand the presence of subnets, whereas others may not. For these reasons, 4.3BSD sets the broadcast address for each interface to be the host value of all ones, but allows the alternate address to be set for backward compatibility. If the network is subnetted, the subnet field of the broadcast address contains the normal subnet number. The *logical* broadcast address for the network also is calculated when the address is set; this would be the standard broadcast address if subnets were not in use. This address is needed by the IP input routine to filter input packets. On input, 4.3BSD recognizes and accepts subnet and network broadcast addresses with host parts of zeros or ones, as well as the address with 32 bits of one (''broadcast on this physical network'').

Internet Ports and Associations

At the IP level, packets are addressed to a host rather than to a process or communications port. However, each packet contains an eight-bit protocol number that identifies the next protocol that should receive the packet. Internet transport protocols use an additional identifier to designate the connection or communications port on the host. Most protocols (including TCP and UDP) use a 16-bit port number for this purpose. Each protocol maintains its own mapping of port numbers to processes or descriptors. Thus, an *association*, such as a connection, is fully specified by the tuple <source address, destination address, protocol number, source port, destination port>. Connection-oriented protocols such as TCP must enforce the uniqueness of associations; other protocols generally do so as well. When the local part of the address is set before the remote part, it is necessary to choose a unique port number to prevent collisions when the remote part is specified.

Protocol Control Blocks

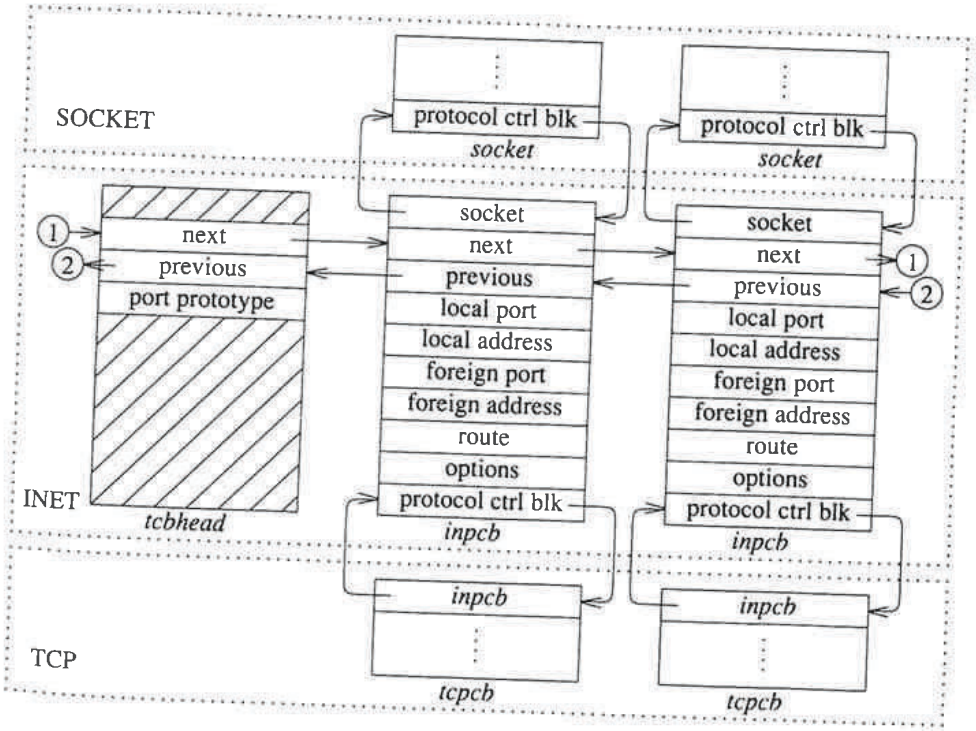
For each TCP- or UDP-based socket, an *Internet protocol control block* (an *inpcb* structure) is created to hold Internet network addresses, port numbers, routing

information, and pointers to any auxiliary data structures. TCP, in addition, creates a *TCP control block* (a *tcpcb* structure) to hold the wealth of protocol state information necessary for its implementation. Internet control blocks for use with TCP are maintained on a doubly linked list private to the TCP protocol module. Internet control blocks for use with UDP are kept on a similar list private to the UDP protocol module. Two separate lists are needed because each protocol in the Internet domain has a distinct space of port identifiers. Common routines are used by the individual protocols to add new control blocks to a list, to fix the local and remote parts of an association, to locate a control block by association, and to delete control blocks. IP demultiplexes message traffic based on the protocol identifier specified in its protocol header, and each higher-level protocol is then responsible for checking its list of Internet control blocks to direct a message to the appropriate socket. Figure 12.5 shows the linkage between the socket data structure and these protocol-specific data structures.

The implementation of the Internet protocols is rather tightly coupled, as befits the strong intertwining of the protocols. For example, the transport protocols send and receive packets including not only their own header, but also an IP pseudoheader containing the source and destination address, the protocol identifier, and a packet length. This pseudoheader is included in the transport-level packet checksum.

We are now ready to examine the operation of the Internet protocols. We begin with UDP, as it is far simpler than TCP.

Figure 12.5 Internet Protocol data structures.



12.2 User Datagram Protocol (UDP)

The *User Datagram Protocol* (*UDP*) is a simple unreliable datagram protocol that provides only peer-to-peer addressing and optional data checksums.[2] Its protocol headers are extremely simple, containing only the source and destination port numbers, the datagram length, and the data checksum. The host addresses for a datagram are provided by the IP pseudoheader.

Initialization

When a new datagram socket is created in the Internet domain, the socket layer locates the protocol-switch entry for UDP and calls the *udp_usrreq*() routine PRU_ATTACH entry with the socket as a parameter. UDP uses *in_pcballoc*() to create a new protocol control block on its list of current sockets. It also sets the default limits for the socket send and receive buffers. Although datagrams are never placed in the send buffer, the limit is set as an upper limit on datagram size; the UDP protocol-switch entry contains the flag PR_ATOMIC, requiring that all data in a send operation be presented to the protocol at one time.

If the application program wishes to bind a port number—for example, the well-known port for some datagram service—it calls the *bind* system call. This request reaches UDP as the PRU_BIND request to *udp_usrreq*(). The binding may also specify a specific host address, which must be an address of an interface on this host. Otherwise, the address will be left unspecified, matching any local address on input, and with an address chosen as appropriate on each output operation. The binding is done by *in_pcbbind*(), which verifies that the chosen port number (or address and port) is not in use, then records the local part of the association.

To send datagrams, the system must know the remote part of an association. This address and port may be specified with each send operation using *sendto* or *sendmsg*, or the specification may be done ahead of time with the *connect* system call. In either case, UDP uses the *in_pcbconnect*() function to record the destination address and port. If the local address was not bound, and if a route for the destination is found, the address of the outgoing interface is used as the local address. If no local port number was bound, one is chosen at this time.

Output

A system call that sends data reaches UDP as a call to *udp_usrreq*() with the PRU_SEND request and a chain of mbufs containing the data for the datagram. If the call provided a destination address, the address is passed as well; otherwise, the address from a prior *connect* call is used. The actual output operation is done by *udp_output*(),

[2] In 4.3BSD, checksums are enabled or disabled by a per-system option and cannot be enabled or disabled on individual sockets.

```
error = udp_output(inp, m);
    struct inpcb *inp;
    struct mbuf *m;
```

where *inp* is an Internet protocol control block and *m* is an mbuf chain that contains the data to be sent. UDP simply prepends its header, fills in its fields and those of a prototype IP header, and calculates a checksum before passing the packet on to the IP module for output:

```
error = ip_output(m, opt, ro, flags);
    struct mbuf *m, *opt;
    struct route *ro;
    int flags;
```

The call to IP's output routine is more complicated than to UDP's because the IP routine cannot depend on having a protocol control block that contains information about the current sender and destination. The *m* parameter indicates the data to be sent, and the *opt* parameter may specify a list of IP options that should be placed in the IP packet header. IP options may be set for a socket with the *setsockopt* system call specifying the IP protocol level and option IP_OPTIONS. These options are stored in a separate mbuf, and a pointer to this mbuf is stored in the protocol control block for a socket; the pointer is passed to *ip_output*() with each packet sent. The *ro* parameter is optional; UDP passes a pointer to the route entry in the protocol control block for the socket. IP will determine a route and leave it in the route structure, so that it can be reused on subsequent calls. The final parameter, *flags*, indicates whether the user is allowed to transmit a broadcast message and if routing is to be performed for the message being sent (see Section 12.3). The broadcast flag may be inconsequential if the underlying hardware does not support broadcast transmissions.

Input

All Internet transport protocols that are layered directly on top of IP use the following calling convention when receiving input packets from IP:

```
(void) (*pr_input)(m, ifp);
    struct mbuf *m;
    struct ifnet *ifp;
```

Each mbuf chain passed is a single packet to be processed by the protocol module. The interface from which the packet was received is passed as the second parameter. The UDP input routine *udp_input*() is typical of protocol input routines. It first verifies that the length of the packet is at least as long as the IP plus UDP headers and uses *m_pullup*() to make the header contiguous. It then checks that the packet is the correct length and checksums the data if a checksum is present.

If any of these tests fail, the packet is simply discarded. Finally, the protocol control block for the socket that is to receive the data is located by *in_pcblookup*() from the addresses and port numbers in the packet. There might be multiple control blocks with the same local port number, but different local or remote addresses; if so, the control block with the best match is selected. An exact association matches best, but if none exists, a socket with the correct local port number but unspecified local address, remote port number, or remote address will match. A control block with unspecified local and/or remote addresses thus acts as a *wildcard* that receives packets for its port if no exact match is found. If a control block is located, the data and the address from which the packet was received are placed in the receive buffer of the indicated socket with *sbappendaddr*(). Otherwise, if no receiver is found and if the packet was not addressed to a broadcast address, an ICMP *port unreachable* error message is sent to the originator of the datagram.[3]

Control Operations

UDP supports few control operations. It supports no options in 4.3BSD, and passes calls to its *pr_ctloutput*() entry directly to IP. It has a simple *pr_ctlinput*() routine that receives notification of any asynchronous errors. Some errors simply cause cached routes to be flushed. Other errors are passed to any datagram socket with the indicated destination; only sockets with a destination fixed by a *connect* call may be notified of errors asynchronously. Such errors are simply noted in the appropriate socket, and socket wakeups are issued in case the process is selecting or sleeping while waiting for input.

When a UDP datagram socket is closed, the *udp_usrreq*() is called with the PRU_DETACH request. The protocol control block and its contents are simply deleted with *in_pcbdetach*(); no other processing is required.

12.3 Internet Protocol (IP)

Having examined the operation of a simple transport protocol, we continue with a discussion of the network-layer protocol [Postel, 1981a; Postel *et al.*, 1981]. The *Internet Protocol* (IP) is the level responsible for host-to-host addressing and routing, packet forwarding, and packet fragmentation and reassembly. Unlike the transport protocols, it does not always operate on behalf of a socket on the local host; it may forward packets, receive packets for which there is no local socket, or generate error packets in response to these situations.

[3] This error message generally has no effect, as the sender normally connects to this destination only temporarily, and destroys the association before new input is processed. However, if the sender still has a fully specified association, it may receive notification of the error. The host-name lookup routine in 4.3BSD uses this mechanism to detect the absence of a nameserver at boot time, allowing it to fall back to the local host file.

The functions performed by IP are illustrated by the contents of its packet header, shown in Fig. 12.6. The header identifies source and destination hosts and the destination protocol, and contains header and packet lengths. The identification and fragment fields are used when a packet or fragment must be broken into smaller sections for transmission on its next hop and to reassemble the fragments when they arrive at the destination. The fragmentation flags are *Don't Fragment* and *More Fragments*; the latter flag plus the offset are sufficient to assemble the fragments of the original packet at the destination.

IP options are present in an IP packet if the header length field has a value larger than the minimum. The *no-operation* option and the *end-of-option-list* option are each one octet in length. All other options are self-encoding, with a type and length preceding any additional data. Hosts and gateways are thus able to skip over options they do not implement. Examples of existing options are the *timestamp* and *record-route* options, which are updated by each gateway that forwards a packet, and the *source-route* options, which supply a complete or partial route to the destination.

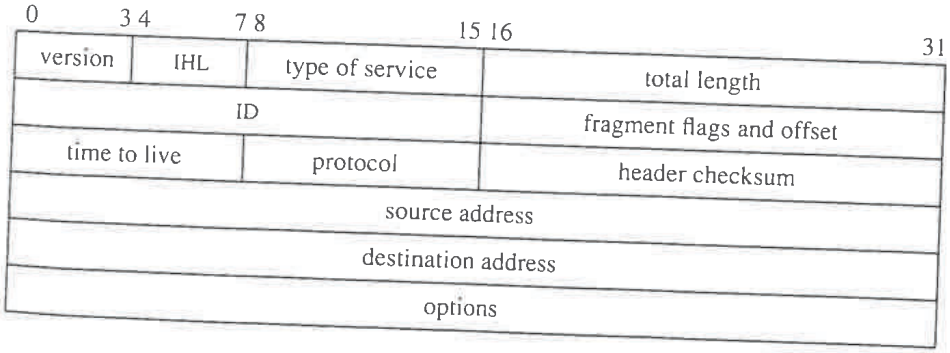
Output

We have already seen the calling convention for the IP output routine, which is

```
error = ip_output(m, opt, ro, flags);
    struct mbuf *m, *opt;
    struct route *ro;
    int flags;
```

As described earlier, the parameter *m* is an mbuf chain containing the packet to be sent, including a skeletal IP header; *opt* is an optional mbuf containing IP options to be inserted after the header. If the route *ro* is given, it is a reference to a routing entry (*rtentry* structure), which may contain a route to the destination from a

Figure 12.6 Internet Protocol header. IHL is the Internet header length specified in units of four octets. Options are delimited by IHL.



previous call, and in which any new route will be left for future use. The *flags* may allow the use of broadcast or may indicate that the routing tables should be bypassed.

The outline of the work done by *ip_output*() is as follows:

- Insert any IP options
- Fill in the remaining header fields (IP version, zero offset, header length, and a new packet identification)
- Determine the route (i.e., outgoing interface and next-hop destination)
- Check whether the destination is a broadcast address, and, if it is, whether broadcast is permitted
- If the packet size is no larger than the maximum packet size for the outgoing interface, compute checksum and call interface output routine
- If the packet size is larger than the maximum packet size for the outgoing interface, break the packet into fragments and send each in turn

We shall examine the routing step in more detail. First, if no route entry is passed as a parameter, an internal routing entry is used temporarily. A route entry that is passed from the caller is checked to see that it is a route to the same destination, and that it is still valid. If either test fails, the old route is freed. After these checks, if there is no route, *rtalloc*() is called to allocate a route. The route returned includes a pointer to the outgoing interface information. This information includes the maximum packet size, flags including broadcast capability, and the output routine. If the route is marked with the RTF_GATEWAY flag, the address of the next-hop gateway is given by the route; otherwise, the packet's destination is the next-hop destination. If routing is to be bypassed because of a MSG_DONTROUTE option (see Section 10.1) or a SO_DONTROUTE option, a directly attached network shared with the destination is found; if there is no directly attached network, an error is returned. Once the outgoing interface and next-hop destination are found, enough information is available to send the packet.

As described in Chapter 11, the interface output routine normally validates the destination address and places the packet on its output queue, returning errors only if the interface is down, the output queue is full, or the destination address is not understood.

Input

In Chapter 11, we described the reception of a packet by a network interface and the packet's placement on the input queue for the appropriate protocol. The network-interface handler then schedules the protocol to run by setting a corresponding bit in the network status word and scheduling a *software interrupt*. The IP input routine is invoked via this software interrupt when network interfaces

receive messages for an Internet protocol; consequently it is called without any parameters. The input routine, *ipintr*(), removes packets from its input queue one at a time and processes them to completion. A packet's processing is completed in one of four ways: it is passed as input to a higher-level protocol, it encounters an error which is reported back to the source, it is dropped because of an error, or it is forwarded along the path to its destination. In outline form, the steps in the processing of an IP packet on input are as follows:

1. Verify that the packet is at least as long as an IP header, and use *m_pullup*() to make the header contiguous.
2. Checksum the header of the packet and discard the packet if there is an error.
3. Verify that the packet is at least as long as the header indicates, and drop the packet if it is not. Trim any padding from the end of the packet.
4. Process any IP options in the header.
5. Check whether the packet is for this host. If so, continue processing the packet. If not, and if performing IP packet forwarding, try to forward the packet. Otherwise, drop the packet.
6. If the packet has been fragmented, keep it until all its fragments are received and reassembled or until it is too old to keep.
7. Pass the packet to the input routine of the next-higher-level protocol.

When the incoming packet is removed from the input queue, it is accompanied by an indication of the interface on which the packet was received. This information is passed to the next protocol, to the forwarding function, or to the error-reporting function. If any error is detected and is reported to the packet's originator, the source address of the error message will be set according to the packet's destination and the incoming interface.

The decision whether to receive a packet is not as simple as we might think. If a host has multiple addresses, the packet is received if its destination matches one of those addresses. If any of the attached networks support broadcast and the destination is a broadcast address, the packet is also received. (For reasons that are given in Section 12.1, there may be as many as five possible broadcast addresses for a given network.)

The IP input routine uses a simple and efficient scheme for locating the input routine for the receiving protocol of an incoming packet. The protocol field in the IP packet is eight bits long; thus, there are 256 possible protocols. Fewer than 256 protocols are actually defined or implemented, and the Internet protocol switch has far fewer than 256 entries. Therefore, IP input uses a 256-element mapping array to map from the protocol number to the protocol-switch entry of the receiving protocol. Each entry in the array is initially set to the index of a "raw" IP entry in the protocol switch. Then, for each protocol with a separate implementation in the system, the corresponding map entry is set to the index of the protocol

in the IP protocol switch. When a packet is received, IP simply uses the protocol field to index into the mapping array, and uses the value at that location as the index into the protocol-switch table for the receiving protocol.

Forwarding

Implementations of the Internet Protocol traditionally have been designed for use by either hosts or gateways, not by both. That is, a system was either an endpoint for IP packets (as source or destination) or a gateway (which forwards packets between hosts on different networks, but uses upper-level protocols only for maintenance functions). Traditional host systems do not incorporate packet-forwarding functions; instead, if they receive packets not addressed to them, they simply drop the packets. 4.2BSD was the first common IP implementation that attempted to provide both host and gateway services in normal operation. This approach had advantages and disadvantages. It meant that 4.2BSD hosts connected to multiple networks could serve as gateways as well as hosts, reducing the requirement for dedicated gateway machines. Early gateways were neither inexpensive nor especially powerful. On the other hand, the existence of gateway-function support in ordinary hosts made it more likely for misconfiguration errors to result in problems on the attached networks. The most serious problem had to do with forwarding of a broadcast packet due to misunderstanding by either the sender or the receiver of the packet's destination. The compromise used in 4.3BSD is that packet-forwarding gateway functions are enabled only if multiple network interfaces are present on a host; these functions may always be disabled. Hosts with only one network interface never attempt to forward packets or to return error messages in response to misdirected packets unless they are explicitly configured as gateways (in which case they are considered to be gateways with broken interfaces). As a result, far fewer misconfiguration problems are capable of causing synchronized or repetitive broadcasts on a local network ("broadcast storms").

The procedure for forwarding IP packets received at a host but destined for another host is the following:

1. Check that forwarding is enabled. If it is not, drop the packet.
2. Check that the destination address is one that allows forwarding. In particular, neither packets sent to network 0 or to network 127 (the official loopback network), nor those sent to addresses that are not on class A, B, or C networks, can be forwarded.[4]
3. Save at most 64 octets of the received message, in case an error message must be generated in response.

[4] Neither the use of class D addresses for multicast nor the implementation of multicast group agents were in existence when 4.3BSD was released. Forwarding of multicast datagrams is done by multicast agents rather than by standard packet-forwarding functions.

4. Determine the route to be used in forwarding the packet.
5. If the outgoing route uses the same interface as that on which the packet was received, and if the originating host is on that network, send an ICMP redirect message to the originating host. (ICMP is described in Section 12.8.)
6. Call *ip_output*() to send the packet to its destination or to the next-hop gateway.
7. If an error is detected, send an ICMP error message to the source host.

12.4 Transmission Control Protocol (TCP)

The major protocol of the Internet protocol suite is the *Transmission Control Protocol (TCP)* [Postel, 1981b; Cerf & Kahn, 1974]. TCP is the reliable connection-oriented stream transport protocol on which most application protocols are based. It includes several features not found in the other transport and network protocols described so far:

- Explicit and acknowledged connection initiation and termination
- Reliable, in-order, unduplicated delivery of data
- Flow control
- Out-of-band indication of urgent data

Because of these features, the TCP implementation is much more complicated than that of UDP or IP. These complications, along with the prevalence of the use of TCP, make the details of TCP's implementation both more critical and more interesting than are the implementations of the simpler protocols. We shall begin with an examination of the TCP itself, then continue with a description of its implementation in 4.3BSD.

A TCP connection may be viewed as a bidirectional, sequenced stream of data octets transferred between two peers. The data may be sent in packets of varying sizes and at varying intervals; for example, when they are used to support a login session over the network. The stream initiation and termination are explicit events at the start and end of the stream, and they occupy positions in the *sequence space* of the stream so that they can be acknowledged in the same manner as data are. Sequence numbers are 32-bit numbers from a circular space; that is, comparisons are made modulo 2^{32}, so that zero is the next sequence number after $2^{32}-1$. The sequence numbers for each direction start with an arbitrary value, called the *initial sequence number*. Each packet (*segment*) of a TCP connection carries the sequence number of its first datum and (except for messages during connection establishment) an acknowledgment of all contiguous data received. Acknowledgments are specified as the sequence number of the next datum not yet received. Acknowledgments are cumulative, and thus may

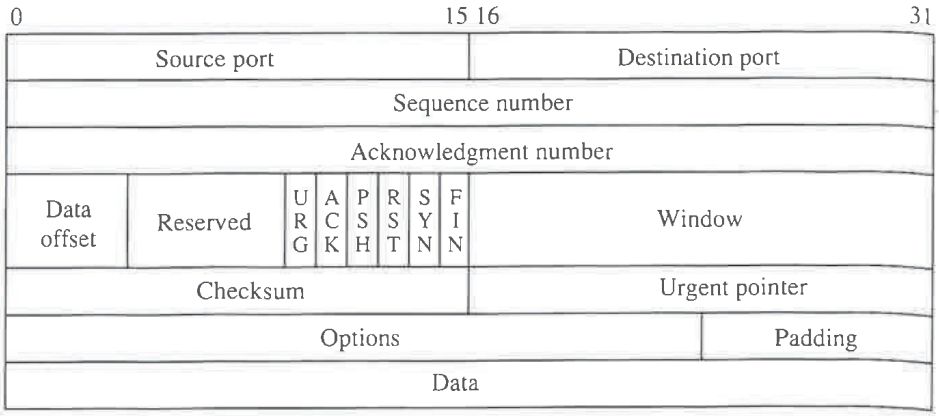


Figure 12.7 TCP packet header.

acknowledge data received in more than one (or part of one) packet. A packet may or may not contain data, but always contains the sequence number of the next datum to be sent.

Flow control in TCP is done with a *sliding-window scheme*. Each packet with an acknowledgment contains a window, which is the number of octets of data that the receiver is prepared to accept, beginning with the sequence number in the acknowledgment. Urgent data are handled similarly; if the flag indicating urgent data is set, the urgent-data pointer is used as a positive offset from the sequence number of the packet to indicate the extent of urgent data. Thus, TCP can send notification of urgent data without sending all intervening data, even if the flow control window would not allow the intervening data to be sent.

The complete header for a TCP packet is shown in Fig. 12.7. The flags include SYN and FIN, denoting the initiation (synchronization) and completion of a connection. Each of these flags occupies a sequence space of one. A complete connection thus consists of a SYN, zero or more octets of data, and a FIN sent from each peer and acknowledged by the other peer. Additional flags indicate whether the acknowledgment field (ACK) and urgent fields (URG) are valid, and include a connection-abort signal (RST). The header includes a header-length field so that the header can be extended with optional fields. Options are encoded in the same way as are IP options: the *no-operation* and *end-of-options* options are single octets, and all other options include a type and a length. The only option currently defined indicates the maximum segment (packet) size that a correspondent is willing to accept; this option is used only during initial connection establishment.

TCP Connection States

The connection-establishment and connection-completion mechanisms of TCP are designed for robustness. They serve to frame the data that are transferred during a connection, so that not only the data but also their extent are reliably

communicated. In addition, the procedure is designed to discover old connections that have not terminated correctly because of a crash of one peer or loss of network connectivity. If such a half-open connection is discovered, it is aborted. Hosts choose new initial sequence numbers for each connection to lessen the chances that an old packet may be confused with a current connection.

The normal connection-establishment procedure is known as a "three-way handshake." Each peer sends a SYN to the other, and each in turn acknowledges the other's SYN with an ACK. In practice, a connection is normally initiated by one of the two (the client) attempting to connect to the other (a server listening on a well-known port). The client choses a port number and initial sequence number and uses these in the initial packet with a SYN. The server creates a new connection block for the pending connection and sends a packet with its initial sequence number, a SYN, and an ACK of the client's SYN. The client responds with an ACK of the server's SYN, completing connection establishment. As the ACK of the first SYN is piggybacked on the second SYN, this procedure requires three packets, leading to the term "three-way handshake." (The protocol still operates correctly if both peers initiate the connection simultaneously, although it requires four packets in that case.)

After a connection is established, each peer includes an acknowledgment and window information in each packet. Each may send data according to the window it receives from its peer. As data are sent by one end, the window becomes filled. As data are received by the peer, acknowledgments may be sent so that the sender can discard the data from its send queue. If the receiver is prepared to accept additional data, perhaps because the receiving process has consumed the previous data, it will also advance the flow-control window. Data, acknowledgments and window updates may all be combined in a single message.

If a sender does not receive an acknowledgment within some reasonable time, it retransmits data that it presumes were lost. Duplicate data are discarded by the receiver but are acknowledged again in case the retransmission was caused by loss of the acknowledgment. If the data are received out of order, the receiver generally retains the out-of-order data for use when the missing segment is received. Out-of-order data may not be acknowledged.

Each peer may terminate data transmission at any time by sending a packet with the FIN bit. A FIN represents the end of the data (like an end-of-file indication). The FIN is acknowledged, advancing the sequence number by one. The connection may continue to carry data in the other direction until a FIN is sent in that direction. The acknowledgment of that FIN terminates the connection. To guarantee synchronization at the conclusion of the connection, the peer sending the last ACK of a FIN must retain state long enough that any retransmitted FIN packets would have reached it or have been discarded; otherwise, if the ACK were lost and a retransmitted FIN were received, the receiver would be unable to repeat the acknowledgment. This interval is arbitrarily set to twice the maximum expected segment lifetime (2MSL).

The TCP input-processing module and timer modules must maintain the state of a connection throughout that connection's lifetime. Thus, in addition to

Table 12.1 TCP connection states.

State	Description
States involved in establishing a connection	
CLOSED	closed
LISTEN	listening for connection
SYN SENT	active, have sent SYN
SYN RECEIVED	have sent and received SYN
State in an established connection	
ESTABLISHED	established
States involved when the remote end initiates a connection shutdown	
CLOSE WAIT	have received FIN, waiting for close
LAST ACK	have received FIN and close; awaiting FIN ACK
CLOSED	closed
States involved when the local end initiates a connection shutdown	
FIN WAIT 1	have closed, sent FIN
CLOSING	closed, exchanged FIN; awaiting FIN ACK
FIN WAIT 2	have closed, FIN is acknowledged; awaiting FIN
TIME WAIT	in 2MSL† quiet wait after close
CLOSED	closed

† twice maximum segment lifetime

processing data received on the connection, the input module must process SYN and FIN flags and other state transitions. The list of states for one end of a TCP connection is shown in Table 12.1. Figure 12.8 shows the finite state machine made up by these states, the events that cause transitions, and the actions during the transitions. An earlier version of the TCP implementation was actually implemented as an explicit state machine.

If a connection is lost because of a crash or timeout on one peer but is still considered established by the other, any data sent on the connection and received at the other end will cause the half-open connection to be discovered. When a half-open connection is detected, the receiving peer sends a packet with the RST flag and a sequence number derived from the incoming packet to signify that the connection is no longer in existence.

Sequence Variables

Each TCP connection maintains a large set of state variables in the TCP control block. This information includes the connection state, timers, options and state flags, a queue that holds data received out of order, and several sequence number variables. The sequence variables are used to define the send and receive sequence space, including the current *window* for each. The window is the range of data sequence numbers that are currently allowed to be sent, from the first octet

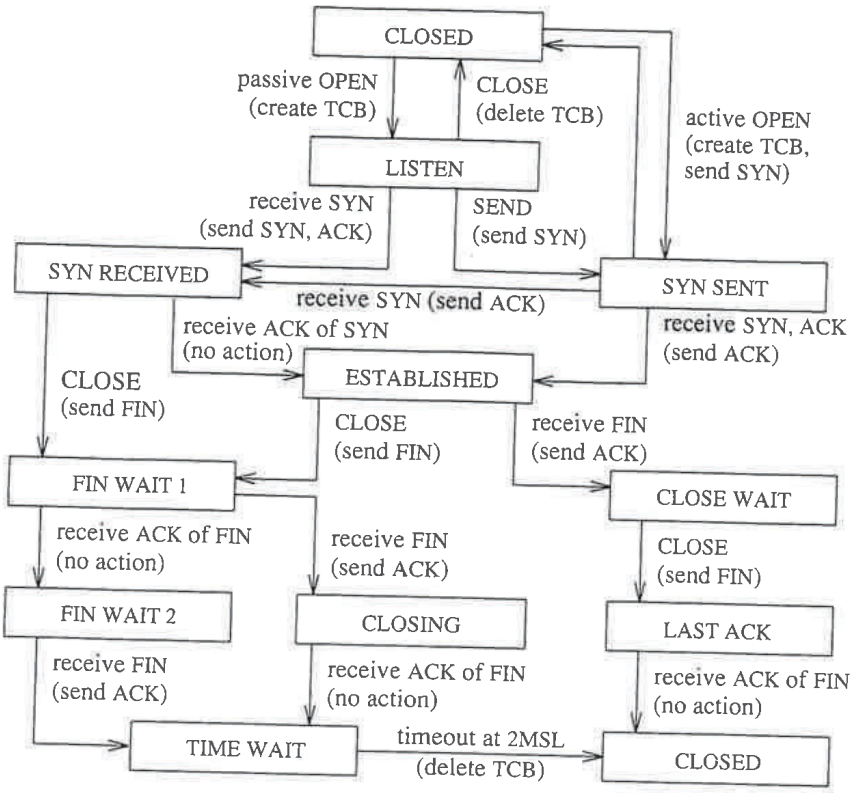


Figure 12.8 TCP state diagram. TCB—TCP control block; 2MSL—twice maximum segment lifetime.

of data not yet acknowledged up to the end of the range that has been offered in the window field of a header. The variables used to define the windows in 4.3BSD are a superset of those used in the protocol specification [Postel, 1981b]. The send and receive windows are shown in Fig. 12.9. The meanings of the sequence variables are listed in Table 12.2.

The area between *snd_nxt* and *snd_una* + *snd_wnd* is known as the *send window*. Data for the range *snd_una* to *snd_max* have been sent but not yet acknowledged, and are kept in the socket send buffer along with data not yet transmitted. The *snd_nxt* and *snd_max* values are normally maintained together except when retransmitting. The area between *rcv_nxt* and *rcv_nxt* + *rcv_wnd* is known as the *receive window*. These variables are used in the output module to decide whether data may be sent, and in the input module to decide whether data that are received may be accepted. When the receiver detects that a packet is not acceptable because the data are all outside the window, it drops the packet but sends a copy of its last acknowledgment. If the packet contained old data, the first acknowledgment may have been lost, and thus it must be repeated. The acknowledgment also includes a window update, synchronizing the sender's state with the receiver's state.

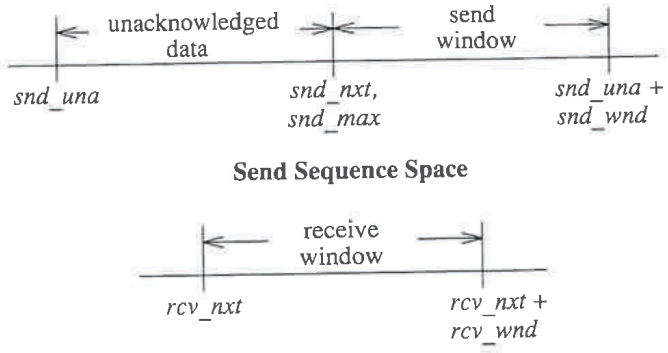


Figure 12.9 TCP sequence space.

12.5 TCP Algorithms

Now that we have introduced TCP, its state machine, and its sequence space, we can begin to examine the implementation of the protocol in 4.3BSD. Several aspects of the protocol implementation are global. The TCP connection state, output state, and state changes depend on external events and timers. TCP processing occurs in response to one of three events:

1. A request from the user, such as sending data, removing data from the socket receive buffer, or opening or closing a connection
2. The receipt of a packet for the connection
3. The expiration of a timer

Table 12.2 TCP sequence variables.

Variable	Description
snd_una	lowest send sequence number not yet acknowledged
snd_nxt	next data sequence to be sent
snd_wnd	number of data octets peer will receive, starting with *snd_una*
snd_max	highest sequence number sent
rcv_nxt	next receive sequence number expected
rcv_wnd	number of octets past *rcv_nxt* that may be accepted
rcv_adv	last octet of receive window advertised to peer

These events are handled in the routines *tcp_usrreq*(), *tcp_input*() and *tcp_timers*(), respectively. Each routine processes the current event and makes any required changes in the connection state. Then, for any transition that may require output, the *tcp_output*() routine is called to do any output that is necessary.

The criteria for sending a packet with data and/or control information are complicated, and therefore the TCP send policy is the most interesting and important part of the protocol implementation. For example, depending on the state- and flow-control parameters for a connection, any of the following may allow data to be sent that could not be sent previously:

- A user send call that places new data in the send queue
- The receipt of a window update from the peer TCP
- The expiration of the retransmission timer
- The expiration of the window-update timer

In addition, the *tcp_output*() routine may decide to send a packet with control information even if no data may be sent for any of these reasons:

- A change in connection state (e.g., open request, close request)
- Receipt of data that must be acknowledged
- A change in the receive window because of removal of data from the receive queue
- A send request with urgent data
- A connection abort

We shall consider most of these decisions in greater detail after we have described the states and timers involved. We begin with algorithms used for timing, connection setup, and shutdown, which are distributed through several parts of the code. We continue with the processing of new input and an overview of output processing and algorithms.

Timers

Unlike a UDP socket, a TCP connection maintains a significant amount of state information, and because of that state, some operations must be done asynchronously. For example, data might not be sent immediately when a process presents them, because of flow control. The requirement for reliable delivery implies that data must be retained after they are first transmitted so that they can be retransmitted if necessary. To prevent the protocol from hanging if packets are lost, each connection maintains a set of timers used to recover from losses or failures of the peer TCP. These timers are stored in the protocol control block for a connection. Whenever they are set, they are decremented every 500 milliseconds by the

tcp_slowtimo() routine (called as the TCP protocol switch *pr_slowtimo* routine) until they expire, triggering a call to *tcp_timers*().

Two timers are used for output processing. One is the *retransmit timer* (TCPT_REXMT). Whenever data are sent on a connection, the retransmit timer is started, unless it is already running. When all outstanding data are acknowledged, the timer is stopped. If the timer expires, the oldest unacknowledged data are resent (at most one full-sized packet) and the timer is restarted with a longer value. The rate at which the timer value is increased (the *timer backoff*) is determined by a table of multipliers that provide a near-exponential increase in timeout values.

The other timer used for maintaining output flow is the *persist timer* (TCPT_PERSIST). This timer protects against the other type of packet loss that could cause a connection to constipate—the loss of a window update that would allow more data to be sent. Whenever data are ready to be sent, but the send window is too small to bother sending (zero, or less than a reasonable amount) and no data are already outstanding (the retransmit timer is not set), the persist timer is started. If no window update is received before the timer expires, the output routine sends as large a segment as the window allows. If that size is zero, it sends a *window probe* (a single octet of data) and restarts the persist timer. If a window update was lost in the network, or if the receiver neglected to send a window update, the acknowledgment will contain current window information. On the other hand, if the receiver is still unable to accept additional data, it should send an acknowledgment for previous data with a still-closed window. The closed window might persist indefinitely; for example, the receiver might be a network-login client, and the user might stop terminal output and leave for lunch (or vacation).

The third timer used by TCP is a *keepalive timer* (TCPT_KEEP). The keepalive timer monitors idle connections that might no longer exist on the correspondent TCP because of timeout or a crash. If a socket-level option is set and the connection has been idle since the last keepalive timeout, the timer routine will send a *keepalive packet* designed to produce either an acknowledgment or a reset (RST) from the peer TCP. If a reset is received, the connection will be closed; if no response is received after several attempts, the connection will be dropped. This facility is designed so that network servers can avoid languishing forever if the client disappears without closing. Keepalive packets are not an explicit feature of the TCP protocol. The packets used for this purpose by 4.3BSD set the sequence number to one less than *snd_una*, which should elicit an acknowledgment from the correspondent TCP if the connection still exists.[5]

The final TCP timer is known as the *2MSL timer* (TCPT_2MSL; "twice the maximum segment lifetime"). TCP starts this timer when a connection is completed by sending an acknowledgment for a FIN (from FIN_WAIT_2 or CLOSING states, where the send side is already closed). Under these circumstances, the sender does

[5] In 4.3BSD, the keepalive packet contains no data unless the system is configured with a kernel option for compatibility with 4.2BSD, in which case a single null octet is sent. A bug prevents 4.2BSD from responding to a keepalive packet unless it contains data.

not know whether the acknowledgment was received. If the FIN is retransmitted, it is desirable that enough state remain that the acknowledgment can be repeated. Therefore, on entering the TIME_WAIT state, the 2MSL timer is started; on its expiration, the control block is deleted. To prevent this delay from blocking a process closing the connection, any process close request is returned successfully without the process waiting for the timer. Thus, a protocol control block may continue its existence even after the socket descriptor has been closed. In addition, 4.3BSD starts the 2MSL timer when FIN_WAIT_2 state is entered after the user has closed; if the connection is idle until the timer expires, it will be closed. Because the user has already closed, new data cannot be accepted on such a connection in any case. This timer is set because certain other TCP implementations (incorrectly) fail to send a FIN on a receive-only connection. Connections to such hosts will remain in FIN_WAIT_2 state forever unless the system times them out.

In addition to the four timers implemented by the TCP *tcp_slowtimo*() routine, TCP uses the protocol switch *pr_fasttimo* entry. The *tcp_fasttimo*() routine, called every 200 milliseconds, processes delayed acknowledgment requests. These will be illustrated in Section 12.6.

Estimation of Round-Trip Time

When connections must traverse slow networks that lose packets, an important decision determining connection throughput is the value to be used when the retransmission timer is set. If this value is too large, data flow will stop on the connection for an unnecessarily long time before the dropped packet is resent. Another round-trip time interval is required to receive an acknowledgment of the missing segment and a window update allowing new data to be sent. (With luck, only one segment will have been lost, and the acknowledgment will include the other segments that had been sent.) If the timeout value is too small, however, packets will be retransmitted needlessly. If the cause of the network slowness or packet loss is congestion, then unnecessary retransmission only exacerbates the problem. The traditional solution to this problem in TCP is to estimate the round-trip time, *rtt*, for the connection path by measuring the time required to receive acknowledgments for individual segments. An estimate of the round-trip time is maintained as a smoothed moving average, *srtt* [Postel, 1981b], using

$$srtt = (\text{ALPHA} \cdot srtt) + ((1-\text{ALPHA}) \cdot rtt).$$

The initial retransmission timeout is set to a multiple (BETA) of the current smoothed round-trip time. The version of TCP in the standard 4.3BSD tape distribution used this scheme, with a smoothing factor ALPHA of 0.9 (retaining 90 percent of the previous average) and a variance factor BETA of 2.

A newer version of TCP was included in the 4.3BSD Tahoe release that uses a somewhat more sophisticated algorithm. In addition to a smoothed estimate of the round-trip time, the new TCP keeps a smoothed variance (estimated as mean difference to avoid square-root calculations in the kernel). It employs an ALPHA value of 0.875 and a corresponding smoothing factor for the variance of 0.75. These values were chosen in part so that the smoothed averages could be

computed using shift operations on fixed-point values, instead of using-floating point values, as in the earlier system. (On many hardware architectures, it is expensive to use floating-point arithmetic in interrupt routines, as doing so forces floating-point registers and status to be saved and restored.) The initial retransmission timeout is then set to the current smoothed round-trip time plus twice the smoothed variance. The new algorithm is substantially better on long-delay paths with little variance in delay, such as satellite links, as it computes the BETA factor dynamically [Jacobson, 1988].

Connection Establishment

There are two ways in which a new TCP connection can be established. An active connection is initiated by a *connect* call, whereas a passive connection is created when a listening socket receives a connection request. We consider each in turn.

The initial steps of an active connection attempt are similar to the actions taken during the creation of a UDP socket. The process creates a new socket, resulting in a call to *tcp_usrreq*() with the PRU_ATTACH request. TCP creates an *inpcb* protocol control block just as does UDP, then creates an additional control block (a *tcpcb* structure), as described in Section 12.1. Some of the flow-control parameters in the *tcpcb* are initialized at this time. If the process explicitly binds an address or port number to the connection, the actions are identical to those for a UDP socket. Then, a *connect* call initiates the actual connection. The first step is to set up the association with *in_pcbconnect*(), again identically to this step in UDP. A packet header template is created for use in constructing each output packet. An initial sequence number is chosen from a sequence-number prototype, which is advanced by a substantial amount. The socket is then marked with *soisconnecting*(), the TCP connection state is set to TCPS_SYN_SENT, the keepalive timer is set (to 45 seconds) to limit the duration of the connection attempt, and *tcp_output*() is called for the first time.

The output-processing module *tcp_output*() uses an array of packet control flags indexed by the connection state to determine which control flags should be sent in each state. In the TCPS_SYN_SENT state, the SYN flag is sent. Because there is a control flag to send, a packet is sent immediately using the prototype just constructed and with the current flow-control parameters. If the connection is to a destination on the local network (or a subnet of that network, see Section 12.1), a maximum-segment-size option is sent in this packet specifying a multiple of 512 bytes that is less than the maximum transmission unit of the outgoing interface. If the option is not sent, the other side must send packets no larger than the default (576 octets including headers). The retransmit timer is set to the default value (6 seconds), as no round-trip time information is available yet.

With a bit of luck, a responding packet will be received from the target of the connection before the retransmit timer expires. If not, the packet is retransmitted and the retransmit timer is restarted with a greater value. If no response is received before the keepalive timer expires, the connection attempt is aborted with a "Connection timed out" error. If a response is received, however, it is checked for agreement with the outgoing request. It should acknowledge the SYN

that was sent, and should include a SYN. If it does both, the receive sequence variables are initialized and the connection state is advanced to TCPS_ESTABLISHED. If a maximum-segment-size option is present in the response, the maximum segment size for the connection is set to the minimum of the offered size and the maximum transmission unit of the outgoing interface; if the option is not present, the default size (512 data bytes) is recorded. The flag TF_ACKNOW is set in the TCP control block before the output routine is called, so that the SYN will be acknowledged immediately. The connection is now ready to transfer data.

The events that occur when a connection is created by a passive open are different. A socket is created and its address is bound as before. The socket is then marked by the *listen* call as willing to accept connections. When a packet arrives for a TCP socket in TCPS_LISTEN state, a new socket is created with *sonewconn*(), which calls the TCP PRU_ATTACH request to create the protocol control blocks for the new socket. The new socket is placed on the queue of partial connections headed by the listening socket. If the packet contains a SYN and is otherwise acceptable, the association of the new socket is bound, both the send and the receive sequence numbers are initialized, and the connection state is advanced to TCPS_SYN_RECEIVED. The keepalive timer is set as before, and the output routine is called after TF_ACKNOW has been set to force the SYN to be acknowledged; an outgoing SYN is sent as well. If this SYN is properly acknowledged, the new socket is moved from the queue of partial connections to the queue of completed connections. If the owner of the listening socket is sleeping in an *accept* call or does a *select*, the socket will indicate that a new connection is available. Again, the socket is finally ready to send data. Up to one window of data may already be received and acknowledged by the time the *accept* call completes.

Connection Shutdown

A TCP connection is symmetrical and full-duplex, so either side may initiate disconnection independently. As long as one direction of the connection can carry data, the connection remains open. A socket may indicate that it has completed sending data with the *shutdown* system call, which results in a call to the *tcp_usrreq*() routine with request PRU_SHUTDOWN. The response to this request is to advance the state of the connection; from the ESTABLISHED state, the state becomes FIN_WAIT_1. The ensuing output call will send a FIN, indicating an end-of-file. The receiving socket will advance to CLOSE_WAIT, but may continue to send. The procedure may be different if the process simply closes the socket; in that case, a FIN is sent immediately, but if new data are received, they cannot be delivered. Normally, higher-level protocols conclude their own transactions such that both sides know when to close. If they do not, however, TCP must refuse new data; it does so by sending a packet with RST set if new data are received after the user has closed. If data remain in the send buffer of the socket when the *close* is done, TCP will normally attempt to deliver them. If the socket option SO_LINGER was set with a linger time of zero, the send buffer is simply

flushed; otherwise, the user process is allowed to continue, and the protocol waits for delivery to conclude. Under these circumstances, the socket is marked with the state bit SS_NOFDREF (no file-descriptor reference). The completion of data transfer and the final close can take place an arbitrary amount of time later. When TCP finally completes the connection (or gives up because of timeout or other failure), it calls *tcp_close*(). The protocol control blocks and other dynamically allocated structures are freed at this time. The socket is also freed if the SS_NOFDREF flag has been set. Thus, the socket remains in existence as long as either a file descriptor or a protocol control block refers to it.

12.6 TCP Input Processing

Although TCP input processing is considerably more complicated than is UDP input handling, the preceding sections have provided the background we need to examine the actual operation. As always, the input routine is called with parameters

```
(void) tcp_input(m, ifp);
    struct mbuf *m;
    struct ifnet *ifp;
```

The first few steps are beginning to sound familiar:

1. Locate the TCP header in the received IP datagram. Make sure that the packet is at least as long as a TCP header and use *m_pullup*() if necessary to make it contiguous.
2. Compute the packet length, set up the IP pseudoheader, and checksum the TCP header and data. Discard the packet if the checksum is bad.
3. Check the TCP header length; if it is larger than a minimal header, make sure that the whole header is contiguous, allocate an extra mbuf, and copy the TCP options into it.
4. Locate the protocol control block for the connection with the port number specified. If none exists, send a packet containing the reset flag RST and drop the packet.
5. Check whether the socket is listening for connections; if it is, follow the procedure described for passive connection establishment.
6. Process any TCP options from the packet header.
7. Clear the idle time for the connection and set the keepalive timer to its normal value.

At this point, the normal checks have been made, and we are prepared to deal with data and control flags in the received packet. There are still many

consistency checks that must be made during normal processing; for example, the SYN flag must be present if we are still establishing a connection, and must not be present if the connection has been established. We shall omit most of these checks from our discussion, but the tests are important to prevent wayward packets from causing confusion and possible data corruption.

The next step in checking a TCP packet is to see whether the packet is acceptable according to the receive window. It is important that this step be done before control flags, in particular RST, are examined, as old or extraneous packets should not affect the current connection unless they are clearly relevant in the current context. A segment is acceptable if the receive window has nonzero size, and if at least some of the sequence space occupied by the packet falls within the receive window. If the packet contains data, some of the data must fall within the window; portions of the data that precede the window are trimmed, as they have already been received, and portions that exceed the window also are discarded, as they have been sent prematurely. If the receive window is closed (*rcv_wnd* is zero), then only segments with no data and with a sequence number equal to *rcv_nxt* are acceptable. If an incoming segment is not acceptable, it is dropped after an acknowledgment is sent.

The way in which these tests are applied in the 4.3BSD Tahoe release version of TCP is somewhat different from the way they are done in the original 4.3BSD tape distribution. We shall use the tests used in the Tahoe release here, as they are more correct. The older version performed similar tests, but in a slightly different order. The processing steps are as follows:

1. Check whether the packet begins before *rcv_nxt*. If it does, ignore any SYN in the packet, and trim any data that fall before *rcv_nxt*. If no data remain, send a current acknowledgment and drop the packet. (The packet is presumed to be a duplicate transmission.)

2. If the packet still contains data after trimming, and the process that created the socket has already closed the socket, send a reset (RST) and drop the connection. This reset is necessary to abort connections that cannot complete; it typically is sent when a remote-login client disconnects while data are being received.

3. If the end of the segment falls after the window, trim any data beyond the window. If the window was closed and the packet sequence number is *rcv_nxt*, the packet is treated as a window probe; TF_ACKNOW is set to send a current acknowledgment and window update, and the remainder of the packet is processed. If SYN is set and the connection was in TIME_WAIT state, this is really a new connection request, and the old connection is dropped; this procedure has been called *rapid connection reuse*. Otherwise, if no data remain, send an acknowledgment and drop the packet.

The remaining steps of TCP input processing check the following flags and fields and take the appropriate actions: RST, ACK, window, URG, data, and FIN.

As the packet has already been confirmed to be acceptable, these actions can be done in a straightforward manner:

4. If RST is set, close the connection and drop the packet.
5. If ACK is not set, drop the packet.
6. If the acknowledgment-field value is higher than previous acknowledgments, new data have been acknowledged. If the connection was in SYN_RECEIVED state and the packet acknowledges our SYN, enter ESTABLISHED state. If the sequence range that was newly acknowledged includes the sequence number for which the round-trip time was being measured, average the time sample into the smoothed round-trip time estimate for the connection. If all outstanding data have been acknowledged, stop the retransmission timer; otherwise, set it back to the current timeout value. Finally, drop the data that were acknowledged from the send queue in the socket. If a FIN has been sent and was acknowledged, advance the state machine.
7. Check the window field to see whether it advances the known send window. First, check whether this is a new window update. If the sequence number of the packet is greater than that of the previous window update, or the sequence number is the same but the acknowledgment is higher, or if both sequence and acknowledgment are the same but the window is larger, record the new window.
8. If the urgent-data flag URG is set, compare the urgent pointer in the packet to the last-received urgent pointer. If it is different, new urgent data have been sent. The urgent pointer is used to compute *so_oobmark*, the offset from the beginning of the socket receive buffer to the urgent mark (Section 10.6), and the socket is notified with *sohasoutofband*(). If the urgent pointer is less than the packet length, the urgent data have all been received. TCP normally removes the data octet sent in urgent mode (the last octet before the urgent pointer[6]) and places it in the protocol control block until it is requested with a PRU_RCVOOB request. A socket option may request that urgent data be left with the normal data.
9. At long last, examine the data field in the received packet. If the data begin with *rcv_nxt*, then they can be placed directly into the socket receive buffer with *sbappend*(). Otherwise, the packet is retained in a per-connection queue until the intervening data arrive. The flag TF_DELACK is set in the protocol control block to indicate that an acknowledgment is needed, but the latter is not sent immediately in hope that it can be piggybacked on any packets sent soon (presumably in response to the incoming data) or combined with acknowledgment of other data received soon. The next time that the *tcp_fasttimo*() routine runs, it will change the flag to TF_ACKNOW and call the *tcp_output*() routine to send the acknowledgment.

[6] The actual position of the "urgent" octet is a subject of disagreement; the 4.2BSD and 4.3BSD interpretation follows the original TCP specification.

10. As the final step in processing a received packet, check for the FIN flag. If it is present, the connection state machine may have to be advanced, and the socket is marked with *socantrcvmore*() to convey the end-of-file indication. If the send side has already closed (a FIN was sent and acknowledged), the socket is now considered closed, and it is so marked with *soisdisconnected*(). The TF_ACKNOW flag is set to force immediate acknowledgment.

Step 10 completes the actions taken when a new packet is received by *tcp_input*(). However, as noted earlier, receipt of input may require new output. In particular, acknowledgment of all outstanding data or a new window update requires either new output or a state change by the output module. Also, several special conditions set the TF_ACKNOW flag. In these cases, *tcp_output*() is called at the conclusion of input processing.

12.7 TCP Output Processing

We are finally ready to investigate the most interesting part of the TCP implementation—the send policy. As we saw earlier, a TCP packet contains an acknowledgment and a window field as well as data, and a single packet may be sent if any of these three change. A naive TCP send policy might send many more packets than necessary. For example, consider what happens when a user types one character to a remote-terminal connection that uses remote echo. The server-side TCP receives a single-character packet. It might send an immediate acknowledgment of the character. Then, milliseconds later, the login server would read the character, removing it from the receive buffer; the TCP might immediately send a window update noting that one additional octet of send window was available. After another millisecond or so, the login server would send an echoed character back to the client, necessitating a third packet sent in response to the single character of input. It is obvious that all three responses (the acknowledgment, the window update, and the data returns) could be sent in a single packet. However, if the server were not echoing input data, the acknowledgment could not be withheld for too long a time, or the client-side TCP would begin to retransmit. The algorithms used in the send policy to minimize network traffic yet to maximize throughput are the most subtle part of a TCP implementation. The send policy used in 4.3BSD includes several standard algorithms as well as some approaches suggested by the network research community. We shall examine each part of the send policy.

As we saw earlier, there are several different events that may trigger the sending of data on a connection; in addition, packets must be sent to communicate acknowledgments and window updates (consider a one-way connection!).

Sending Data

The most obvious reason that the tcp output module *tcp_output*() is called is that the user has written new data to the socket. Write operations are done with a call

to *tcp_usrreq*() with the PRU_SEND request. (Recall that *sosend*() waits for enough space in the socket send buffer if necessary, then copies the user's data into a chain of mbufs that is passed to the protocol with the PRU_SEND request.) The action in *tcp_usrreq*() is simply to place the new output data in the socket's send buffer with *sbappend*() and to call *tcp_output*(). If flow control permits, *tcp_output*() will send the data immediately.

The actual send operation is not very different from one for a UDP datagram socket; the differences are that the header is more complicated, and additional fields must be initialized, and that the data sent are simply a copy of the user's data.[7] A copy must be retained in the socket's send buffer in case of retransmission. Also, if the number of data octets is larger than the size of a single maximum-sized segment, multiple packets will be constructed and sent in a single call.

The *tcp_output*() routine allocates an mbuf to contain the output packet header and copies the contents of the header template into that mbuf. The data to be sent are added as a separate chain of mbufs obtained with an *m_copy*() operation from the appropriate part of the send buffer. The sequence number for the packet is set from *snd_nxt*, and the acknowledgment is set from *rcv_nxt*. The flags are obtained from an array containing the flags to be sent in each connection state. The window to be advertised is computed from the amount of space remaining in the socket's receive buffer; however, if that amount is small (less than one-fourth of the buffer and less than one segment), it is set to zero. The window is never allowed to end at a smaller sequence number than it did in previous packets. If urgent data have been sent, the urgent pointer and flag are set accordingly. One other flag must be set: The PUSH flag on a packet indicates that data should be passed to the user, something like an end-of-record mark or buffer-flush request. This flag is generally considered obsolete, but is set whenever all the data in the send buffer have been sent; 4.3BSD ignores this flag on input. Once the header is filled in, the packet is checksummed. The remaining parts of the IP header are initialized, and the packet is sent with *ip_output*(). The retransmission timer is started if it it is not already running, and the *snd_nxt* and *snd_max* values for the connection are updated.

Avoidance of the Silly-Window Syndrome

Silly-window syndrome is the name given to a potential problem in a window-based flow-control scheme in which a receiver sends several allocations rather than waiting for a reasonable-sized window to become available [Clark, 1982]. For example, if a network-login client program has a total receive buffer size of 4096 octets, and the user stops terminal output during a large printout, the buffer will become nearly full as new full-sized segments are received. If the remaining buffer space dropped to 10 bytes, it would not be useful for the receiver to volunteer to receive an additional 10 octets. If the user then allowed a small

[7] However, for send operations large enough for *sosend*() to place the data in external mbuf clusters, the copy is done by creation of a new reference to the data cluster.

number of characters to print and stopped output again, it still would not be useful for the receiving TCP to send a window update allowing another 14 octets. Instead, it is desirable to wait until a reasonably large packet can be sent, as the receive buffer already contains enough data for the next several pages of output. Avoidance of the silly-window syndrome is desirable in both the receiver and the sender of a flow-controlled connection, as either end can prevent silly small windows from being used. Receiver avoidance of the silly-window syndrome has been described already; when a packet is sent, the receive window is advertised as zero if it is less than one packet and less than one-fourth of the receive buffer. For sender avoidance of the silly-window syndrome, an output operation is delayed if at least a full packet of data is ready to be sent, but less than one full packet can be sent because of the size of the send window. Instead of sending, the output state is set to persist state by the persist timer being started. If no window update has been received by the time the timer expires, the allowable data are sent in the hope that the acknowledgment will include a larger window. If not, the connection stays in persist state, sending a window probe periodically until the window is opened.

An initial implementation of sender avoidance of the silly-window syndrome produced large delays and low throughput over connections to hosts using TCP implementations with tiny buffers. Unfortunately, those implementations *always* advertised receive windows less than 512 octets, which was considered silly by this implementation. As a result of this problem, the 4.3BSD TCP keeps a record of the largest receive window offered by a peer in the protocol-control-block variable *max_sndwnd*. When at least one-half of *max_sndwnd* may be sent, a new segment is sent. This technique improved performance when a 4.3BSD system was communicating with these primitive hosts.

Avoidance of Small Packets

Network traffic exhibits a bimodal distribution of sizes. Bulk data transfers tend to use the largest possible packets for maximum throughput. Network-login services tend to use small packets, however, often containing only a single data character. On a fast local-area network such as an Ethernet, the use of single-character packets generally is not a problem, as the network bandwidth usually is not saturated. On slower long-haul networks such as the ARPANET or on networks interconnected by slow links, it is desirable to collect input over some period and then to send it in a single network packet. Various schemes have been devised for collecting input over a fixed time, usually about 50 to 100 milliseconds, and then sending it in a single packet. These schemes noticeably slow character echo times on fast networks, however, and often save few packets on slow networks. In contrast, a simple and elegant scheme for reducing small-packet traffic was suggested by Nagle [Nagle, 1984]. This scheme allows the first octet output to be sent alone in a packet with no delay. Until this packet is acknowledged, however, no new small packets may be sent. If enough new data arrive to fill a maximum-sized packet, another packet is sent. As soon as the outstanding data are acknowledged, the input that was queued while waiting for the

first packet may be sent. Only one small packet may ever be outstanding on a connection at one time. The net result is that small numbers of data are queued during one round-trip time. If the round-trip time is less than the intercharacter arrival time, as with a remote-terminal session on a local-area network, transmissions are never delayed, and response time remains low. When a slow network intervenes, input after the first character is queued, and the next packet contains the input received during the preceding round-trip time. This algorithm is attractive both because of its simplicity and because of its self-tuning nature.

Eventually, people discovered that this algorithm did not work well for certain classes of network clients that sent streams of small requests that could not be batched. One such client was the network-based X Window System [Scheifler & Gettys, 1986], which required immediate delivery of small messages to get real-time feedback for user interfaces such as rubber-banding to sweep out a new window. Hence an option, TCP_NODELAY, was added to TCP to defeat this algorithm on a connection. This option can be set with a *setsockopt* call, which reaches TCP via the *tcp_ctloutput*() routine.

Window Updates

TCP packets must be sent for reasons other than data transmission. On a one-way connection, the receiving TCP must still send packets to acknowledge received data and to advance the sender's send window. The time at which these packets are sent is a determining factor for network throughput. For example, if the receiver simply set the TF_DELACK flag each time data were received on a bulk-data connection, acknowledgments would be sent every 200 milliseconds. Using 4096-octet windows on a 10-Mbit/s Ethernet, this would result in a maximum throughput of 160 Kbit/s, or 1.6 percent of the physical network bandwidth. Clearly, once the sender has filled the send window that it has been given, it must stop until the receiver acknowledges the old data (allowing them to be removed from the send buffer and new data to replace them) and provides a window update (allowing the new data to be sent).

Because TCP's window-based flow control is limited by the space in the socket receive buffer, TCP has the PR_RCVD flag set in its protocol-switch entry so that the protocol will be called (via the PRU_RCVD request of *tcp_usrreq*()) when the user has done a receive call that has removed data from the receive buffer. The PRU_RCVD entry simply calls *tcp_output*(). (In 4.2BSD, *tcp_output*() was also called after each packet was received by *tcp_input*(); that call is eliminated under most circumstances in 4.3BSD.) Whenever *tcp_output*() determines that a window update sent under the current circumstances would provide sufficient new send window to the sender to be worthwhile, it sends an acknowledgment and window update. If the receiver really had to wait until the window was full, the sender would already have been idle for some time when it finally received a window update. Therefore, the window-update strategy in 4.3BSD is based on a fraction of the receive window. Whenever a new window update would expose at least 35 percent of the total receive buffer, the window update is sent. Using the default 4096-byte buffers and 1024-octet segments on

an Ethernet, this criterion causes every other segment to be acknowledged. This window-update strategy produces a two-fold reduction in acknowledgment traffic and a two-fold reduction in input processing for the sender. This strategy assumes that the sender in each connection has a send buffer of a similar size to the receiver's receive buffer. Unfortunately, this assumption was not always correct, and if the sender's buffer was less than 35 percent of the size of the receiver's receive buffer, window updates were never triggered except when *tcp_fasttimo*() forced out delayed acknowledgments. Therefore, in the 4.3BSD Tahoe release of TCP, the strategy was changed to send a window update if at least twice the segment size was added to the sender's window.

Retransmit State

When the retransmit timer expires while a sender is awaiting acknowledgment of transmitted data, *tcp_output*() is called to retransmit. The retransmit timer is first set to the next multiple of the round-trip time in the backoff series. The variable *snd_nxt* is moved back from its current sequence number to *snd_una*. A single packet is then sent containing the oldest data in the transmit queue. Unlike some other systems, 4.3BSD does not keep copies of the packets that have been sent on a connection, but retains only the data. Thus, although only a single packet is retransmitted, that packet may contain more data than does the oldest outstanding packet. On a slow connection with small numbers of data, this may mean that a single-octet packet that is lost may be retransmitted with all the data queued since they were first transmitted.

If a single packet was lost in the network, the retransmitted packet will elicit an acknowledgment of all data yet transmitted. If more than one packet was lost, the next acknowledgment will include the retransmitted packet and possibly some of the intervening data. It may also include a new window update. Thus, when an acknowledgment is received after a retransmit timeout, any old data that were not acknowledged will be resent, and some new data may be sent as well.

Source-Quench Processing and Congestion Control

If a gateway along the route used by a connection receives more packets than it can send along this path, it may be forced to drop packets. When this happens, the gateway may send an ICMP *Source Quench* error message to hosts whose packets have been dropped to indicate that the senders should slow their transmissions. Although this message indicates that some change should be made, it provides no information on how much of a change must be made or for how long the change should take effect. One algorithm for handling source quench was proposed by Nagle [Nagle, 1984] and implemented in 4.3BSD. As the receipt of a source quench indicates that too many packets are outstanding in the network, the host receiving a source quench must reduce the number of packets it sends at one time. The algorithm uses a second window, like the send window but maintained separately, called the congestion window (*snd_cwnd*). Initially, the congestion-window size is set to some large amount. When a source quench is received for

some destination, the congestion-window size is set to a fraction of the data outstanding, but to no less than the maximum segment size; in 4.3BSD, the fraction is 0.8. The send policy is then modified so that new data are sent if allowed by both the normal and congestion send windows. As the network congestion might be reduced with time, the congestion window is opened by a small amount (e.g., 10 percent), whenever a new acknowledgment is received. If this opening results in too large a congestion window at some point, a new source quench should be received. Thus, the number of data outstanding should modulate slowly around the limit that can be tolerated by the network.

The implementation of this scheme in 4.3BSD suffered from two flaws. First, although the congestion window needed to be kept in bytes to maintain significance, the send policy should have truncated this limit to a multiple of the segment size. The original implementation would send a smaller segment at the end of the congestion window. This produced no less congestion on the network, but fragmented buffers at the receiver and led to the use of odd packet sizes for some time. This problem was corrected in the 4.3BSD Tahoe release, as described in the following two sections. Second, many gateways do not send source-quench messages for each packet dropped.

Slow Start

Many TCP connections traverse several networks between source and destination. When some of the networks are slower than others, the entry gateway to the slowest network often is presented with more traffic than can be handled. It may buffer some number of input packets to avoid dropping packets due to sudden changes in flow, but eventually its buffers will fill and it must begin dropping packets. When a TCP connection first starts sending data across a fast network to a gateway to a slower network, it may find that the gateway's buffers are already nearly full. In the original send policy used in 4.3BSD, a bulk-data transfer would start out by sending a full window of packets once the connection was established. These packets could be sent at the full speed of the network to the bottleneck gateway, but that gateway could transmit them at only a much slower rate. As a result, the initial burst of packets was highly likely to overflow the gateway's buffers, and some of the packets would be lost. If such a connection used an expanded window size in an attempt to gain performance—for example, when traversing a satellite-based network with a long round-trip time—this problem would be even more severe. However, if the connection could once reach steady state, a full window of data often could be accommodated by the network if the packets were spread evenly throughout the path. At steady state, new packets would be injected into the network only when previous packets were acknowledged, and the number of packets in the network would be constant. In addition, even if packets arrived at the outgoing gateway in a cluster, they would be spread out when traversing the network by at least their transmission times in the slowest network. If the receiver sent acknowledgments when each packet was received, the acknowledgments would return to the sender with approximately the correct spacing. The sender would then have a self-clocking means for transmitting at the correct rate for the network without sending bursts of packets that the bottleneck could not buffer.

An algorithm named slow-start brings a TCP connection to this steady state [Jacobson, 1988]. It is called *slow start* because it is necessary to start data transmission slowly when traversing a slow network. The scheme is simple; a connection starts out with a limit of just one outstanding packet. Each time an acknowledgment is received, the limit is increased by one packet. If the acknowledgment also carries a window update, two packets can be sent in response. This process continues until the window is fully open. During the slow-start phase of the connection, if each packet was acknowledged separately, the limit would be doubled during each exchange, resulting in an exponential opening of the window. Delayed acknowledgments might cause acknowledgments to be coalesced if more than one packet could arrive at the receiver within 200 milliseconds, slowing the window opening slightly. However, the sender never sends bursts of more than two or three packets during the opening phase, and sends only one or two packets at a time once the window has opened.

The implementation of the slow-start algorithm uses the same congestion-window field in the connection block that was originally used for source-quench handling in 4.3BSD, *snd_cwnd*. This field is set to the size of one packet whenever the connection comes to a halt. It is initialized to one packet for any connection to a destination not on a shared network, and is set to one packet whenever transmission stops due to a timeout. Otherwise, once a dropped packet was acknowledged, the resulting window update might allow a full window of data to be sent, which would once again overrun intervening gateways. This slow start after a retransmission timeout eliminates the need for a test in the output routine to limit output to one packet on the initial timeout. In addition, the timeout may indicate that the network has become slower due to congestion, and temporary reduction of the window may help the network recover from its condition. The connection is forced to reestablish its clock of acknowledgments after the connection has come to a halt, and the slow start accomplishes this as well.

The use of the slow-start algorithm after retransmission timeouts allows a connection to respond correctly to a dropped packet, whether or not a source quench is received to indicate the loss. The action on receipt of a source quench in this version of TCP is simply to anticipate the timeout due to the dropped packet, setting the congestion window to one packet. This action prevents new packets from being sent until the dropped packet is resent at the next timeout. At that time, the slow start will begin again.

Avoidance of Congestion with Slow Start

The addition of the slow-start algorithm to TCP allows a connection to send packets at a rate that the network can tolerate, reaching a steady state at which packets are sent only when another packet has exited the network. A single connection may reasonably use a larger window without flooding the entry gateway to the slow network on startup. For some paths, using slow start and a large window can lead to much better performance than could be achieved previously. For example, paths that traverse satellite links have long intrinsic delay, even though the bandwidth may be high, but the throughput is limited to one window of data per round-trip time. Even with such a path, gateways with limited buffering, such as most ARPANET gateways, between the source and the satellite uplink would be

flooded without slow start. It is highly desirable for a TCP connection to be self-tuning, however, as the characteristics of the path are seldom known at the end-points. A window size large enough to give good performance when a long-delay link is in the path will overrun the network when most of the round-trip time is in queuing delays. As a connection opens the window during a slow start, it injects packets into the network until the network links are kept busy, and then additional widening of the window will simply increase the size of the queues in the gateway. Opening the window past the point at which all the transmission links are kept busy does not increase throughput, but it does increase the delay seen by the connection. If a connection expands to too large a window for a path, or if additional load on the network collectively exceeds the capacity, gateway queues will build until packets must be dropped. At this point, the connection will close the congestion window to one packet and will initiate a slow start. If the window is simply too large for the path, however, this process will repeat once the window is opened too far.

The connection can learn from this problem, and adjust its behavior accordingly with another algorithm installed at the same time as slow start was. This algorithm keeps a new state variable for each connection, *t_ssthresh* (slow-start threshold), which is an estimate of the usable window for the path. When a packet is dropped, as evidenced by a retransmission timeout, this window estimate is set to one-half the number of the outstanding data octets. The current window is obviously too large at the moment, and the decrease in window utilization must be large enough that congestion will decrease rather than stabilizing. At the same time, the slow-start window (*snd_cwnd*) is set to one segment to restart. The connection starts up as before, opening the window exponentially until it reaches the *t_ssthresh* limit. To test for improvement in the network, it continues to expand the window slowly; as long as this expansion succeeds, the connection can continue to take advantage of reduced network load. The expansion of the window in this phase is linear, adding one additional full-sized segment to the current window for each full window of data transmitted. This slow increase allows the connection to discover when it is safe to resume use of a larger window while reducing the loss in throughput due to the wait after the loss of a packet before transmission can resume. Note that the increase in window size during this phase of the connection is linear as long as no packets are lost, but the decrease in window size when signs of congestion appear is exponential (it is divided by 2 on each timeout). With the use of this dynamic window-sizing algorithm, it is possible to use larger default window sizes for connection to all destinations without overrunning networks that cannot support them.

12.8 Internet Control Message Protocol (ICMP)

The *Internet Control Message Protocol (ICMP)* [Postel, 1981c] is the control- and error-message protocol for IP. Although it is layered above IP for input and output operations much like UDP, it is really an integral part of IP. Unlike those of UDP, most ICMP messages are received and implemented by the kernel. ICMP

messages may also be sent and received using a raw IP socket (see Section 11.7).

ICMP messages fall into three general classes. One class includes various errors that may occur somewhere in the network and that may be reported back to the originator of the packet provoking the error. Such errors include routing failures (network or host unreachable), expiration of the time-to-live field in a packet, or a report by the destination host that the target protocol or port number is not available. The second message class may be considered as gateway-to-host control messages. The two instances of such messages are the source-quench message that reports excessive output and packet loss, and the routing redirect that informs a host that a better route is available for a host or network via a different gateway. The final message class includes network management, testing, and measurement packets. These include a network address request and reply, a network mask request and reply, an echo request and reply, and a timestamp request and reply.

All the actions and replies required by an incoming ICMP message are done by the kernel ICMP layer. ICMP packets are received from IP via the normal protocol-input entry point, as ICMP has its own IP protocol number. The ICMP input routine formulates responses to any requests and passes the reply to *ip_output*() to be returned to the sender. When error indications or source quenches are received, a generic address is constructed in a *sockaddr* structure, and the address and error code are reported to each network protocol's control-input entry *pr_ctlinput*() by *pfctlinput*().

Routing changes indicated by redirect messages are processed by the *rtredirect*() routine. It verifies that the gateway from which the message was received was the next-hop gateway in use for the destination, and it checks that the new gateway is on a directly attached network. If these tests succeed, the kernel routing tables are modified accordingly. If the new route is of equivalent scope to the previous route (e.g., both are for the destination network), the gateway in the route is changed to the new gateway. If the scope of the new route is smaller than that of the original route (either a host redirect is received when a network route was used, or the old route used a wildcard route), a new route is created in the kernel table. Routes that are created or modified by redirects are marked with the flags RTF_DYNAMIC and RTF_MODIFIED, respectively. Once the routing tables are updated, the protocols are notified by *pfctlinput*(), using a redirect code rather than an error code. TCP and UDP simply flush any cached route from the protocol control block when a redirect is received. The next packet sent on the socket will thus reallocate a route, choosing the new route if it is now the best route.

Once an incoming ICMP message has been processed by the kernel, it is passed to *raw_input*() for any ICMP raw sockets. The raw sockets can also be used to send ICMP messages. The low-level network test program **ping** works by sending ICMP echo requests on a raw socket and listening for corresponding replies.

ICMP is also used by other Internet network protocols to generate error messages. UDP sends only ICMP Port Unreachable error messages, and TCP uses other means to report such errors. However, many different errors may be

detected by IP, especially on systems used as IP gateways. The *icmp_error*() function constructs an error message of a specified type in response to an IP packet. Most error messages include a portion of the original packet that caused the error as well as the type and code for the error. The identity of the interface on which the original packet was received is also passed to *icmp_error*() in case the destination of the original packet was a different host, as occurs when forwarding is done; the source address of the error message can then be set to the address of the gateway on the network closest to (or shared with) the originating host. Also, when IP forwards a packet via the same network interface on which that packet was received, it may send a redirect message to the originating host if that host is on the same network. The *icmp_error*() routine accepts an additional parameter for redirect messages—the address of the new gateway to be used by the host.

12.9 ARPANET Host Interface

The protocols we have described so far were designed for use in an Internet environment centered on the ARPANET. The ARPANET (and similar networks, including the Milnet) are packet-switched networks consisting of minicomputers interconnected by 56-Kbit/s trunks. Hosts and gateways are connected to the switching nodes, which receive packets from a host, forward them via some number of intermediate nodes if necessary, and deliver them to the destination host or gateway. The *Packet Switch Node* (*PSN*), previously known as an *Interface Message Processor* (IMP) [Heart *et al.*, 1970], provides the host interface and packet-switching functions. It supports reliable message delivery with flow control. Messages are presented to the receiving PSN with a header containing the destination host (as a PSN number and the host's number on the PSN). The PSNs use a dynamic routing protocol to determine the route to the destination PSN [McQuillan *et al.*, 1980]. Messages may be fragmented into subnetwork packets of at most 128 octets, then reassembled by the destination PSN. The end-to-end protocol used by the PSNs manages buffer allocation, message delivery, and acknowledgment. Once a message is accepted by the source PSN, the source host will receive either an acknowledgment or a failure message within a specified time (30 seconds). Flow control is implemented by hosts being limited to a maximum of eight outstanding messages to each other host. Thus, the acknowledgment of one message allows another message to be sent, and the acknowledgment is known as *Ready for Next Message* (RFNM).

Two major types of host connections to PSNs are in current use. The original host interface to the IMP is known as an 1822 interface, after the number of the report containing the specification for its message protocol and electrical signaling [BBN, 1978]. The other interface now used for connection to a PSN is X.25, which we shall describe later.

The *ARPANET Host Access Protocol* (AHIP), is the newer name for the 1822 host protocol. This protocol is implemented by common routines in the **netimp**

directory of the kernel source. Several types of hardware interfaces supply connections to a PSN using the 1822 protocol. The packet formatting and interpretation, as well as the per-host flow control, are done by the common layer, whereas the drivers are responsible for only sending and receiving messages. The AHIP protocol uses a simple header containing a message type, destination IMP and host numbers, and a link type. Some messages are exchanged with the local PSN, for example the initialization messages. The link type is used in data messages to identify the protocol that is to receive the message, and thus corresponds with the type field of an Ethernet header. There are several message types for control information, including the RFNM message, the failure message (*Incomplete Message*), and messages that indicate that destination hosts are down or unreachable.

The IMP interface software maintains a list of entries for hosts with which it is communicating. This host entry contains a count of messages that have been sent to the host for which RFNM (or Incomplete Message) messages have not been received. When packets in excess of the limit are received for transmission to the same host, up to eight additional packets are enqueued in the host entry for transmission when RFNMs are received. Messages beyond this limit are dropped, returning the error that indicates buffer space has been exhausted. On gateways this error causes an ICMP source-quench message to be generated for the originating host. If a *Host Dead* or a *Host Unreachable* message is received, indicating that a destination is down, the host entry is marked with this information and a timer is set. Until the timer expires, attempts to send to that destination will produce errors of the appropriate type, and the packets are dropped with no attempt made to send them. Gateways may thus send appropriate ICMP error messages to hosts attempting to send packets to unreachable destinations.

The other type of host interface that may used for connection to a PSN is X.25 [CCITT, 1980], similar to the connection to a public packet-switched data network. An X.25 interface to the Milnet (Defense Data Network, DDN) may be used to connect a host to a PSN for use with the Internet protocols. In this mode, the X.25 interface is used to create virtual circuits to the local PSN for communications with other hosts, rather than to make X.25 virtual circuits terminating at the destination hosts. Hosts with X.25 connections may communicate with hosts that have 1822 connections; the PSNs map both access protocols onto the internal end-to-end protocol, and each host uses the flow-control mechanism appropriate to its access method. 4.3BSD does not include drivers for hardware using an X.25 interface to a PSN, although such drivers are available.

12.10 Xerox Network Systems Communication Domain (XNS)

The Xerox Network Systems (NS) communication domain supports three communication protocols. *IDP* is the *Internetwork Datagram Protocol*, which forms the basis for the level-1 transport protocol in the NS architecture. IDP is responsible for the internet packet format, internet addressing, and routing. The

Sequenced Packet Protocol (SPP), is a level-2 transport protocol layered on top of IDP. SPP provides reliable, unduplicated, and flow-controlled transmission of messages. The third protocol is the *Error Protocol*, which reports errors; this protocol is used as a diagnostic tool as well as as a means of improving performance; it is not normally accessed by users.

SPP is used to support the operation of both stream and sequenced packet sockets created in the NS domain. The standard datagram service, called *Packet Exchange (PEX)*, is implemented by a user-level library using IDP datagram sockets. Raw access to the Error Protocol and IDP protocols is possible through *raw sockets*.

For each SPP- or IDP-based socket, an *NS protocol control block* is created to hold NS information similar to that found in the Internet protocol control block. Also like TCP, SPP creates an *SPP control block* to hold the protocol state information necessary for its implementation. Unlike those in the Internet domain, the NS protocol control blocks for all protocols are maintained on *one* doubly linked list. In the NS domain, packet demultiplexing at the network-protocol level is done first according to the network address and port numbers and then according to the communication protocol. The linkage between the socket data structure and the protocol-specific data structures is identical to that found in the Internet communication domain.

The NS and Internet protocol implementations are similar. Consequently, understanding the data structures, organization, and algorithms of one protocol family makes understanding the other much easier.

The SPP protocol output routine uses the following convention:

```
error = spp_output(spcb, m);
    struct sppcb *spcb;
    struct mbuf *m;
```

where the *spcb* is an SPP protocol control block and *m* is the mbuf chain that contains the data to be sent.

The IDP protocol uses a similar convention:

```
error = idp_output(nsp, m);
    struct nspcb *nsp;
    struct mbuf *m;
```

which differs only in the use of an NS protocol control block. Both SPP and IDP pass packets to the common NS module used for packet transmission with

```
error = ns_output(m, ro, flags);
    struct mbuf *m;
    struct route *ro;
    int flags;
```

As in the previous example, the *m* parameter is an mbuf chain containing the message to be transmitted. The *ro* parameter is used in making routing decisions (for IDP, this will always be zero, as it does not hold routing information between packets transmitted). The *flags* parameter indicates whether the user is allowed to transmit a broadcast packet and whether routing is to be performed. As noted before, the broadcast flag may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure is detected immediately.

All NS protocol input is received by the NS interrupt routine, *nsintr*(), which is invoked at software-interrupt level when a network interface receives a message to be processed by an NS protocol module. The interrupt routine performs consistency checks on the packet, and then determines whether to receive, forward, or discard the packet. If the packet is to be received, *nsintr*() locates the NS protocol control block for the receiving socket according to the sender's address and the destination port number. It then passes the packet to the receiving protocol.

Both IDP and SPP use the following calling convention:

```
(void) (*pr_input)(m, nsp, ifp);
    struct mbuf *m;
    struct nspcb *nsp;
    struct ifnet *ifp;
```

Each mbuf chain is a single packet to be processed by the protocol module. The interface from which the packet was received is passed as the last parameter.

The IDP *pr_ctlinput*() routine is mainly used to pass control information to the IDP and SPP protocol modules from the Error protocol's input routine. (It may also be invoked from the Internet ICMP module if NS is being encapsulated inside IP.) The common calling convention for this routine is

```
(void) (*pr_ctlinput)(req, arg);
    int req;
    caddr_t arg;
```

The *req* will usually be a value shown in Table 11.4, although only a restricted subset of these requests is mapped from Error Protocol codes in *ns_err_input*(). The *arg* parameter is a pointer to an Error Protocol packet when a mapped request is supplied. Otherwise, the *arg* is a pointer to a socket address structure.

XNS Control Operations

The SPP and IDP *pr_ctloutput*() routines provide access to NS-specific options that control the behavior of SPP and/or IDP in processing data transmitted and received through a socket. Options usable with both SPP and IDP protocols are SO_HEADERS_ON_INPUT, to have protocol headers returned on each message with data; SO_MTU, to set the maximum size of a message that will be sent or

received; and SO_DEFAULT_HEADERS, to establish a default header for outgoing messages.

12.11 Summary

In this section, we shall tie together much of the material presented in the last three chapters. To do this, we shall describe the operation of the socket and network layers during normal use.

There are three stages in the lifetime of a socket. Initially, the socket is created and is associated with some communication domain. During its lifetime, data passes through it to one or more other sockets. When the socket is no longer needed, it must go through an orderly shutdown process in which its resources are freed.

Creating a Communication Channel

Sockets are created by users with the *socket* system call and internally with the *socreate*() routine. To create a socket, the user must give a communication domain and socket type, and it also may request a specific communication protocol within that domain. The socket routines first locate the domain structure for the communication domain from a global list initialized at boot time for each configured domain. The table of protocols that constitute the domain's protocol family is located in the domain structure. This table of protocol-switch entries is then scanned for an appropriate protocol to support the type of socket being created (or for a specific protocol, if one was specified). The socket routine performs this search by examining the *pr_type* field, which contains a possible socket type (e.g., SOCK_STREAM), and the *pr_protocol* field, which contains the protocol number of the protocol, normally a well-known value. If a suitable protocol is found, a reference to the protocol's protocol-switch entry is then recorded in the socket's *so_proto* field, and all requests for network services are made through the appropriate procedure identified in the structure.

After locating a handle on a protocol, *socreate*() allocates an mbuf and creates the socket data structure with the socket's initial state. To complete the creation process, *socreate*() makes a PRU_ATTACH request to the protocol's user request routine so that the protocol may *attach* itself to the new socket.

Next, an address may be bound to a socket. Binding of an address is done internally by *sobind*(), which makes a PRU_BIND request to the socket's supporting protocol. The Internet and XNS domains each provide a routine that manages their address spaces. Addresses in the UNIX domain are names in the filesystem name space, and consequently name requests go through the filesystem name-lookup routine, *namei*().

For a socket to be ready to accept connections, the socket layer must inform the protocols with a PRU_LISTEN request. This request obviously has no meaning for connectionless protocols such as UDP and IDP. For connection-oriented protocols such as TCP and SPP, however, a listen request causes a protocol state

transition. Before effecting this state change, protocols verify that the socket has an address bound to it; if it has none, the protocol module chooses one for it.

In the UNIX domain a listen request causes no state change, but a check is made to ensure that the socket has a name. Unlike the other protocols, however, the UNIX domain will not select a name for the socket.

Soconnect() is invoked to establish a connection, generating a PRU_CONNECT request to the protocol. For connectionless protocols, the address is recorded as a default address to be used when data are sent on the socket (i.e., the process does a *write* or *send*, instead of a *sendto*). Setting the address does not require any peer communication, and the protocol module returns immediately.

For a connection-based protocol, the peer's address is verified, and a local address is assigned for unbound sockets. Instead of entering a *connected* state immediately, the socket is marked as *connecting* with *soisconnecting*(). The protocol then initiates a handshake with the peer by transmitting a connection-request message. When a connection request of this sort is completed, usually on receipt of a message by the protocol input routine, the socket's state is changed with a call to *soisconnected*().

From a user's perspective all connection requests appear synchronous because the *connect* system call invokes *soconnect*() to initiate a connection and then, at the socket level, puts the calling process to sleep if the connection request has not been completed. Alternatively, if the socket has been made nonblocking with *fcntl*, *connect* returns the error EINPROGRESS once the connection has been initiated successfully. The caller may test the completion of the connection with a *select* call testing for ability to write to the socket.

For connection-based communication, a process must accept an incoming connection request on a listening socket by calling *accept*, which in turn calls *soaccept*(). This call returns the next completed connection from the socket receive queue.

Sending and Receiving Data

Once a socket has been created, data can begin to flow through it. A typical TCP/IP connection is shown in Fig. 12.10. The *sosend*() routine is responsible for copying data from the sending process's address space into mbufs. It then presents the data to the network layer with one or more calls to the protocol's PRU_SEND request. The network may choose to send the data immediately, or to wait until a more auspicious time. If the protocol delays, or if it must retain a copy of the data for possible retransmission, it may store the data in the socket's send buffer. Eventually, the data are passed down through TCP and IP to the appropriate interface driver as one or more packets; at each layer, an appropriate header is added. Each packet is sent out over the network to its destination machine.

On receipt at the destination machine, the interface driver's receiver-interrupt handler verifies and removes its own header and places the packet onto an appropriate network-protocol input queue. Later, the network-level input processing module (e.g., IP or IDP) is invoked by a software interrupt; it runs at a lower

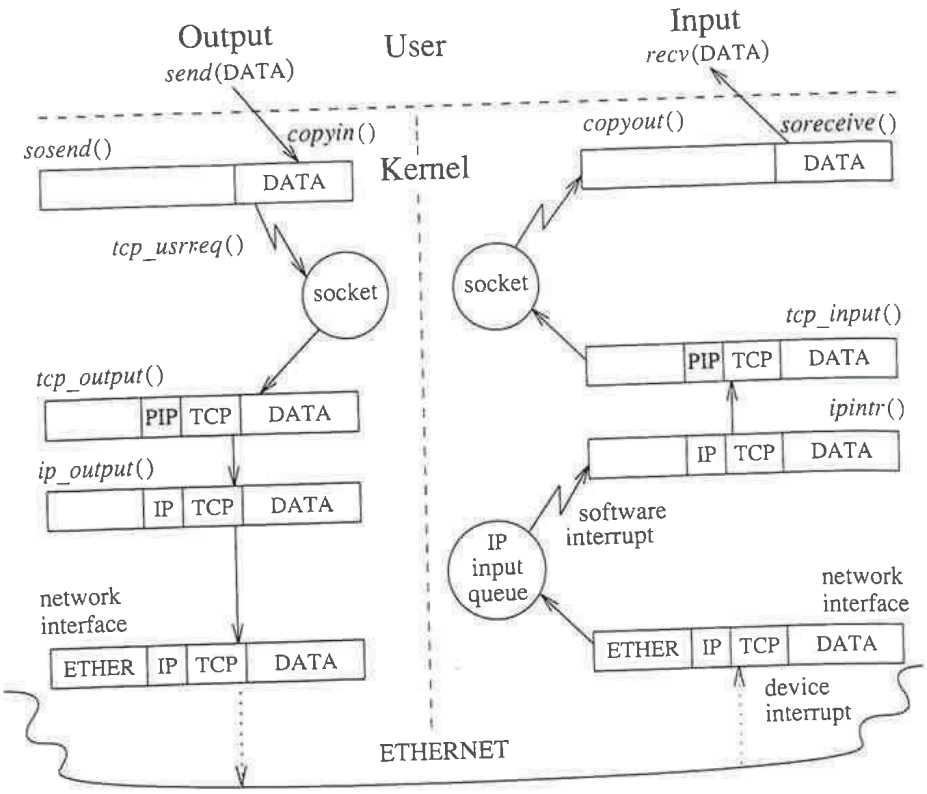


Figure 12.10 Data flow through a TCP/IP connection over an Ethernet. ETHER—Ethernet header; PIP—pseudo IP header; IP—IP header; TCP—TCP header.

interrupt priority level than that of the hardware network interface interrupt. In this example, the packets on the input queue are processed first by IP and then by TCP, each of which verifies and removes its header. If received in order, the data are then placed on the appropriate socket's input queue, ready to be copied out by *soreceive*() on receipt of a *read* request.

Terminating Data Transmission and/or Reception

The *soshutdown*() routine stops data flow at a socket. Shutting down a socket for reading is a simple matter of flushing the receive queue and marking the socket as unable to receive more data; this is done with a call to *sorflush*(), which in turn invokes *socantrcvmore*() to change the socket state, and then releases any resources associated with the receive queue. Shutting down a socket for writing, however, involves notifying the protocol with a PRU_SHUTDOWN request. For reliable connections, any data remaining in the send queue must be drained before the connection can finish shutting down. If a protocol supports the notion of a *half-open* connection (i.e., a connection in which unidirectional data flow is possible), the socket may continue to be usable; otherwise, the protocol may start a

disconnect sequence. Once a socket has been shut down in both directions, the protocol starts a disconnect sequence. When the disconnect completes, first the resources associated with the network and then those associated with the socket are freed.

Exercises

12.1 Is TCP a transport-, network-, or link-layer protocol?

12.2 How does IP identify the next-higher-level protocol that should process an incoming message? How does this dispatching differ in the Xerox NS architecture?

12.3 How many hosts can exist on a class C Internet network? Is it possible to use subnet addressing with a class C network?

12.4 What is a broadcast message? How are broadcast messages identified in the DARPA Internet?

12.5 Why are TCP and UDP protocol control blocks kept on separate lists?

12.6 Why does the IP output routine check the destination address of an outgoing packet to see whether it is a broadcast address instead of the socket-layer send routine (*sosend*())?

12.7 Why does 4.3BSD not forward broadcast messages?

12.8 Why does the TCP header include a header-length field even though it is always encapsulated in an IP packet that contains the length of the TCP message?

12.9 What is the flow-control mechanism used by TCP to limit the rate at which data are transmitted?

12.10 How does TCP recognize messages from a host that are directed to a connection that previously existed, but that has since been shut down (such as after a machine is rebooted)?

12.11 When is the size of the TCP receive window for a connection not equal to the amount of space available in the associated socket's receive buffer? Why are these values not equal at that time?

12.12 What are keepalive messages and for what does TCP use them? Why are keepalive messages implemented in the kernel rather than, say, in each application that wants this facility?

12.13 Why is calculating a *smoothed* round-trip time important, rather than, for example, just averaging calculated round-trip times?

12.14 Why does TCP delay acknowledgments for received data? What is the maximum period of time that TCP will delay an acknowledgment?

12.15 Explain what the silly-window syndrome is and give an example in which its avoidance is important to good protocol performance. How does the 4.3BSD TCP avoid this problem?

12.16 What is meant by *small-packet avoidance*? Why is small-packet avoidance bad for clients that exhibit one-way data flow and that require low latency for good interactive performance (e.g., the X Window System)?

*12.17 A *directed broadcast* is a message that is to be broadcast on a network one or more hops away from the sender. Describe a scheme for supporting directed-broadcast messages in the Internet domain.

*12.18 Why is the initial sequence number for a TCP connection selected at random, rather than being, say, always set to zero?

*12.19 In the TCP protocol, why do the SYN and FIN flags occupy space in the sequence-number space?

*12.20 Describe a typical TCP packet exchange during connection setup. Assume an active client initiated the connection to a passive server. How would this scenario change if the passive server tried simultaneously to initiate a connection to the client?

*12.21 Sketch the TCP state transitions that would take place if a server process accepted a connection and then immediately closed it before receiving any data. How would this scenario be altered if 4.3BSD supported a mechanism whereby a server could refuse a connection request before the system completed the connection?

*12.22 At one time, the 4BSD TCP used a strict exponential backoff strategy for transmission. Explain how this nonadaptive algorithm can adversely affect performance across networks that are very *lossy*, but that have high bandwidth (e.g. some networks that utilize satellite connections).

*12.23 Why does UDP match the completely specified destination addresses of incoming messages to sockets with incomplete local and remote destination addresses?

*12.24 Why might a sender set the *Don't Fragment* flag in the header of an IP packet?

*12.25 The *maximum segment lifetime* (MSL) is the maximum time a message may exist in a network; that is, the maximum time that a message may be in transit on some hardware medium or queued in a gateway. What does TCP do to ensure that TCP messages have a limited MSL? What does IP do to enforce a limited MSL [Fletcher & Watson, 1978]?

**12.26 Describe a protocol for calculating a bound on the maximum segment lifetime of messages in an internet environment. How might TCP use a bound on the MSL (see Exercise 12.25) for a message to minimize the overhead associated with shutting down a TCP connection?

References

BBN, 1978.

BBN, "Specification for the Interconnection of Host and IMP," Technical Report 1822, Bolt, Beranek, and Newman, Cambridge, MA (May 1978).

CCITT, 1980.

CCITT, "Recommendation X.25: Interface between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) for Terminals Operating in the Packet Mode on Public Data Networks," in *Public Data Networks, Orange Book, vol. VIII.2* (1980).

Carr *et al.*, 1970.

S. Carr, S. Crocker, & V. Cerf, "HOST–HOST Communication Protocol in the ARPA Network," *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 589–597 (1970).

Cerf, 1978.

V. Cerf, "The Catenet Model for Internetworking," Technical Report IEN 48, SRI Network Information Center, Menlo Park, CA (July 1978).

Cerf & Kahn, 1974.

V. G. Cerf & R. E. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications* **22**(5), pp. 637–648 (May 1974).

Clark, 1982.

D. D. Clark, "Window and Acknowledgment Strategy in TCP," RFC 813, SRI Network Information Center, Menlo Park, CA (July 1982).

DARPA, 1983.

DARPA, "A History of the ARPANET: The First Decade," Technical Report, Bolt, Beranek, and Newman, Cambridge, MA (April, 1983).

Fletcher & Watson, 1978.

J. Fletcher & R. Watson, "Mechanisms for a Reliable Timer-Based Protocol," pp. 271–290 in *Computer Networks 2*, North-Holland, Amsterdam, The Netherlands (1978).

Heart *et al.*, 1970.

F. E. Heart, R. E. Kahn, S. Ornstein, W. Crowther, & D. Walden, "The Interface Message Processor for the ARPA Computer Network," *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 551–567 (1970).

ISO, 1984.

ISO, "Open Systems Interconnection—Basic Reference Model," ISO 7498, International Organization for Standardization (1984). Available from: American National Standards Institute, 1430 Broadway, New York, NY 10018.

Jacobson, 1988.

V. Jacobson, "Congestion Avoidance and Control," *Proceedings of the ACM SIGCOMM Conference*, pp. 314–329 (August 1988).

McQuillan & Walden, 1977.

J. M. McQuillan & D. C. Walden, "The ARPA Network Design

Decisions,'' *Computer Networks* **1**(5), pp. 243–289 (1977).

McQuillan *et al.*, 1980.
J. M. McQuillan, I. Richer, & E. Rosen, ''The New Routing Algorithm for the ARPANET,'' *IEEE Transactions on Communications* **28**(5), pp. 711–719 (May 1980).

Mogul, 1984.
J. Mogul, ''Broadcasting Internet Datagrams,'' RFC 919, SRI Network Information Center, Menlo Park, CA (October 1984).

Mogul & Postel, 1985.
J. Mogul & J. Postel, ''Internet Standard Subnetting Procedure,'' RFC 919, SRI Network Information Center, Menlo Park, CA (August 1985).

Nagle, 1984.
J. Nagle, ''Congestion Control in IP/TCP Internetworks,'' RFC 896, SRI Network Information Center, Menlo Park, CA (January 1984).

Padlipsky, 1985.
M. A. Padlipsky, *The Elements of Networking Style,* Prentice-Hall, Englewood Cliffs, NJ (1985).

Postel, 1980.
J. Postel, ''User Datagram Protocol,'' RFC 768, SRI Network Information Center, Menlo Park, CA (August 1980).

Postel, 1981a.
J. Postel, ''Internet Protocol,'' RFC 791, SRI Network Information Center, Menlo Park, CA (September 1981).

Postel, 1981b.
J. Postel, ''Transmission Control Protocol,'' RFC 793, SRI Network Information Center, Menlo Park, CA (September 1981).

Postel, 1981c.
J. Postel, ''Internet Control Message Protocol,'' RFC 792, SRI Network Information Center, Menlo Park, CA (September 1981).

Postel *et al.*, 1981.
J. B. Postel, C. A. Sunshine, & D. Cohen, ''The ARPA Internet Protocol,'' *Computer Networks* **5**(4), pp. 261–271 (July 1981).

Scheifler & Gettys, 1986.
R. W. Scheifler & J. Gettys, ''The X Window System,'' *ACM Transactions on Graphics* **5**(2), pp. 79–109 (April 1986).

PART 5

System Operation

CHAPTER 13

System Startup

When a computer is powered on, there is nothing running on the CPU. For a program to be set running, the binary image of the program must first be loaded into memory from a storage device. Many microprocessor systems automatically start programs that reside in nonvolatile storage devices such as programmable read-only memories (PROMs). Once the image of the program is loaded, the CPU must be directed to start execution at the first memory location of the loaded program. This process of *bootstrapping* a program into execution is used to start any program running on a CPU.

In this chapter, we examine how the UNIX kernel, or any other similar program, is bootstrapped. We then study the operation of the system during the initialization phase, which takes the system from a *cold start* to the point at which user-mode programs can be run. A final section examines topics that are related to the startup procedure. These topics include configuring the kernel load image, shutting down a running system, and debugging system failures.

13.1 Overview

The UNIX kernel is only a program, albeit a large one. Like any UNIX program, its binary image resides in a file in the filesystem until it is loaded and set running. 4.3BSD presumes the executable image of the kernel resides in a file named **vmunix** on the filesystem that is designated as the *root filesystem*. When UNIX is bootstrapped on a machine, a special program, named **boot**, is first loaded and is started running. The **boot** program's task is to load and initialize the executable image of a program and to start it running. **Boot** may come from the same storage device as the file it bootstraps, or it may be loaded from a storage device supported by the machine's console processor specifically for bootstrapping purposes.

The **boot** program reads the binary image of a program to be bootstrapped into main memory and then initializes the CPU so that the loaded program can be started. Programs loaded by **boot** are set running with virtual-address translation and hardware interrupts disabled. The loaded program is responsible for enabling these facilities and any additional hardware, such as I/O devices, that it intends to use.

When the UNIX kernel is loaded by the **boot** program, the kernel goes through several stages of hardware and software initialization in preparation for normal system operation. The first stage is responsible for initializing the state of the CPU, including the run-time stack and virtual-memory mapping. Memory mapping, including virtual-address translation, is enabled early in the startup procedure to minimize the amount of special-purpose assembly-language code that must be written. Once virtual-memory mapping is enabled, the system does machine-dependent initializations and then machine-independent initializations. The machine-dependent operations include setting up virtual-memory page tables and configuring I/O devices; the machine-independent actions include mounting the root filesystem and initializing the myriad system data structures. This order is necessary because many of the machine-independent initializations depend on the I/O devices being initialized properly.

Following the setup of the machine-independent portions of the kernel, the system is in operational status. System processes are created and made runnable and user-level programs are brought in from the filesystems to execute. At this point, the system is ready to run normal applications.

13.2 Bootstrapping

Bootstrapping a program is a machine-dependent operation. On most machines, this operation is supported either by a secondary processor termed the *console processor*, or by a *console monitor*. The console-monitor program is resident in nonvolatile storage and is invoked automatically when the CPU is reset. The console facilities are expected to support the bootstrap of standalone programs. Most console processors and monitors also execute diagnostic operations when a machine is reset to ensure that the hardware is functioning properly.

The boot Program

The console processor or console monitor often does not understand the format of the 4.3BSD filesystem. Instead, a vendor's proprietary filesystem format, or some other filesystem format, is interpreted by the software that supports bootstrapping operations. For this reason, the UNIX kernel historically has been bootstrapped through the **boot** program. This program is a general-purpose *standalone* program that can be used to load and execute other standalone programs. A *standalone program* is a program that is capable of operating without the assistance of the UNIX kernel. Standalone programs usually are linked with the *standalone I/O library*, a library that supports a UNIX-like I/O interface on a variety of hardware devices. The standalone I/O library provides these facilities through a collection

of *standalone device drivers* and a library of routines that support reading of files from UNIX filesystems that reside on the devices.

Once the **boot** program has been loaded and started, it must load the file containing the executable image of the program to be bootstrapped and then start the loaded program running. To load the appropriate file, **boot** must know the pathname of the file to be loaded and the hardware device on which the file resides. This information is communicated to **boot** in one of two ways. On the VAX, two hardware registers are initialized to contain values that specify how **boot** should operate and which device should be used when program images are loaded. Other machines communicate the bootstrapping information to the **boot** program by initializing the run-time stack and then placing the parameters on the stack in the same way the UNIX kernel passes arguments to programs.

The registers used on the VAX are called the *boothowto register* and the *bootdevice register*. The boothowto register is interpreted by **boot** when the latter starts running; depending on the register's value, **boot** will either load the file **vmunix** from a device specified in the bootdevice register, or will prompt at the console terminal for the name of a file to load and a device from which to obtain the specified file. The value placed in the boothowto register is defined from the flags shown in Table 13.1. The value placed in the bootdevice register to identify the hardware device to use for bootstrapping is dependent on the configuration of the standalone I/O library with which **boot** is linked. Since the composition of this library is dependent on the hardware devices available on a machine, the contents of the bootdevice register can vary among different types of machines. For our discussion, it is sufficient to know that the register's contents specify a default device and filesystem partition from which to read programs.

Boot always loads programs at memory location 0. Since **boot** is initially loaded in memory at location 0, it must copy its own image to another place in

Table 13.1 Flags passed to the **boot** program and to the *reboot* system call.

Flag	Description
RB_ASKNAME	prompt for a filename to bootstrap
RB_SINGLE	bootstrap a system to single-user operation
RB_NOSYNC†	do not sync before rebooting
RB_HALT†	do not reboot, just halt
RB_INITNAME	use the name given instead of **/etc/init**
RB_DFLTROOT	use the compiled-in root device
RB_KDB	give control to the kernel debugger
RB_RDONLY	mount the root filesystem read-only
RB_DUMP†	dump kernel memory before rebooting

† Meaningful to only the *reboot* system call.

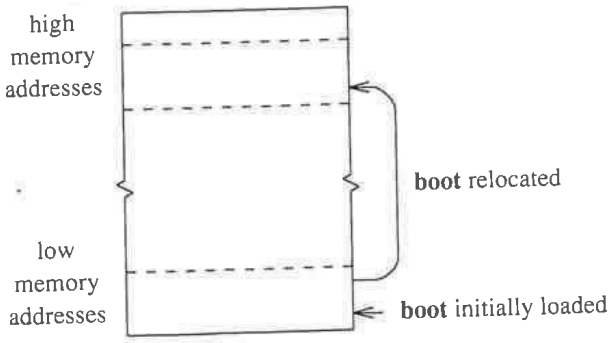


Figure 13.1 Placement of the **boot** program in memory.

memory to avoid loading the image of the program it bootstraps on top of itself (see Fig. 13.1). This *relocation* implies that the **boot** program must be created with its starting address set to the memory location at which it will be copied; otherwise, references to data structures in the **boot** program will access the wrong memory locations after **boot** is copied (remember that **boot** operates with virtual-address translation disabled).

VAX Console Media

Most VAX computers include a console processor and some form of secondary storage that is accessible from the console processor. The VAX–11/780 console subsystem includes a floppy-disk drive, whereas the VAX–11/750 and VAX–11/730 console subsystems include a small cartridge-tape drive.

The exact sequence of steps taken during bootstrapping a VAX varies from machine to machine. In normal operation, the VAX console processor searches for a file called **DEFBOO.CMD** on its console media. This file contains commands that the console processor should interpret in order to bootstrap the CPU. The commands in this file typically reset the CPU and I/O subsystems, initialize registers to known values, and then load a file from the console storage device and start it running. The user can request alternative actions on some VAX machines by setting front panel switches, or by typing special commands on the console terminal.

When the console storage device is used in the bootstrapping process, the **boot** program must reside on this storage device. 4.3BSD supports two utility programs that manipulate the filesystem format used by the console processor—**arff** and **flcopy**. These programs are used to place programs such as **boot** on VAX console storage media.

13.3 Kernel Initialization

When the UNIX kernel is started by the **boot** program, it does an initialization in preparation for the execution of application programs. The initialization process

is roughly divided into three stages. The first stage is written entirely in assembly language and does the work necessary for non–assembly-language code to operate. The second stage does machine-dependent operations including the configuration and initialization of the I/O devices on the machine. The third stage does machine-independent operations, completing its work by starting up the system-resident processes that compose the foundation for the normal UNIX run-time environment.

Assembly-Language Startup

The first steps taken by the system during initialization are carried out by assembly-language code. This work is very machine-dependent; it includes

- Setting up the run-time stack
- Identifying the type of CPU on which the system is executing
- Calculating the amount of physical memory on the machine
- Enabling the virtual-address–translation hardware
- Initializing the memory-management hardware
- Crafting the hardware context for process 0
- Invoking the initial C-based entry point of the system

Our discussion of these steps will concentrate on the work required to support the VAX; similar actions are necessary for any machine on which 4.3BSD runs.

When the **boot** program starts the UNIX kernel running, it sets up only two components of the machine state:

1. The interrupt priority is set at its highest level so that all hardware interrupts are blocked.
2. The hardware address-translation facility is disabled so that all memory references are to physical memory locations.

The UNIX kernel presumes nothing else about the state of the machine on which it is running.

The kernel is loaded into physical memory at location 0. In normal operation, the VAX address-translation hardware is enabled and the kernel image is mapped into virtual memory starting at hexadecimal location 80000000. In preparation for enabling address translation, the physical memory used by the kernel is mapped into the identical virtual locations. This *double mapping* is necessary because the VAX will not enter mapped mode until address translation is enabled *and* a branch instruction is executed. Double mapping of the physical memory associated with the kernel ensures that a branch instruction to a succeeding instruction (specified as a virtual address) will go to the correct physical location.

A second task of the startup code is to identify the type of VAX on which the system is executing. Although all VAX computers support a common instruction set, the I/O architecture varies among machines. Furthermore, some microprocessor implementations of the VAX architecture directly support only a subset of the complete VAX instruction set. For these machines, the kernel must *emulate* the missing hardware instructions in software.

The kernel identifies the type of VAX on which it is executing by interrogating the contents of the *system identification register* (SID). The contents of this register includes a well-known constant that uniquely identifies the type of VAX. 4.3BSD uses the system type

- To distinguish among the different types of console storage media and console processor interfaces
- To decide which hardware devices and I/O interfaces might be configured on a machine
- To handle exceptional machine conditions (*machine checks*)
- To calculate real-time delays implemented with *spin loops*
- To decide how to set up interrupt handlers

4.3BSD can be configured such that a single kernel load image can support a number of different VAXes.

Machine-Dependent Initialization

After the assembly-language code has completed its work, it calls the first kernel routine written in C, the *main*() routine. One parameter is passed to this routine—the number of the first available page of memory. The system calculates this value, taking into account the size of the kernel's load image and any pages allocated to run-time stacks and to other data structures. The *main*() routine immediately calls the *startup*() routine to do machine-dependent initializations. *Startup*() is supplied the same value *main*() received from the assembly-language code.

The tasks of the machine-dependent startup code include

- Initialization of the *error-message buffer*
- Allocation of memory for system data structures
- Initialization of the kernel's memory allocator
- Autoconfiguration and initialization of I/O devices

A few other hardware-specific parts of the machine are initialized after the call to *startup*() returns; these operations are described later, in our discussion of the machine-independent startup code.

Message Buffer

The *message buffer* is a 4 Kbyte circular buffer located at the top of physical memory. Diagnostic messages displayed on the console with the *printf*() routine (or one of its variants) are kept in this buffer as an aid in tracking problems. Prior to 4.3BSD, the message buffer was accessible only through the **/dev/kmem** special device. Furthermore, utilities such as the **dmesg** program that read the message buffer and copy the buffer's contents to an administrative log file were unable to synchronize their activities properly with the generation of new diagnostic messages. For these reasons, 4.3BSD added a special device, **/dev/log**. This device provides a read-only interface to the message buffer that supports the *select* system call. In addition, most system diagnostics are now generated in a format that is interpreted by the **syslogd** program. These changes ensure that system diagnostics are saved reliably in log files.

Initialization of the message buffer is straightforward. First, the system allocates memory for the buffer by deducting the size of the message buffer from the size of physical memory calculated by the assembly-language code. Then, a private page map, *msgbufmap*, is initialized to reference the physical pages just allocated.

System Data Structures

Allocation of memory for the system data structures is easy at this point in the startup procedure. The *startup*() routine is supplied the identity of the first available page of physical memory that follows the resident kernel, *firstaddr*. In addition, the kernel is known to reside in virtual memory starting at hexadecimal location 80000000. Thus, to allocate space for contiguous data structures, the system simply assigns the virtual address of the next available block of physical memory to each data structure,

```
base = 0x80000000 | (firstaddr * NBPG)
```

and then increments the value of *firstaddr* by the size of the data structure. Memory allocated to data structures in this way is not necessarily initialized to zero; initialization routines called from *main*() ensure that the contents of each data structure are set up properly.

The technique just described is used to allocate memory for each contiguous system data structure. Most of these data structures are sized at the time the system is configured, with the sizes based on the peak number of users expected (see also Section 13.5). The buffer cache and core map, however, are sized according to the amount of physical memory available on the machine. The buffer-cache size is calculated as 10 percent of the first 2 Mbyte of physical memory and 5 percent of the remaining memory. The system ensures that there are a minimum of 16 buffers, although this lower limit should never be a problem unless the system is configured with very large filesystem block sizes. In addition to the buffers dedicated to the buffer cache, the system must also allocate buffer headers for

swapping operations: one-half of the number of file I/O buffer headers is allocated for use in swap and paging operations. The system must calculate the size of the core map after allocating the buffer cache and static data structures, because that value is used to map all physical memory not otherwise allocated to the system. Once the core-map and system-memory allocator have been initialized (described in the following section), the normal system memory-allocation mechanisms must be used.

Memory Allocator

The last step in the machine-dependent startup procedure prior to autoconfiguration is the initialization of the system's memory allocator. The kernel has already allocated all the physical memory it requires to hold system data structures; any remaining memory is made available for general use. The memory allocator is initialized by a call to the *meminit*() routine with the first and last available pages of memory. The core map structure is initialized to reference this memory and the *firstfree*, *maxfree*, *freemem*, and *avefree* variables used by the memory-management system (see Section 5.12) are set up based on the available memory.

Autoconfiguration

Autoconfiguration is the procedure carried out by the system to recognize and enable the hardware devices present in a system. This process was introduced in Section 8.5, where the requirements it places on device drivers were described. In this section, we shall concentrate on the central workings of the autoconfiguration process. In particular, we shall study the data structures that are used to describe a system's hardware topology and the way in which devices are accessed by the system.

To understand the autoconfiguration process, we must first gain some familiarity with the structure of a VAX computer. The *VAX architecture* is a specification that pertains mostly to the operation of the CPU. That is, each VAX computer supports a common instruction set, memory-management facilities, mechanisms for handling traps and interrupts, and so on. A common subset of the instruction set is implemented in hardware, with any remaining instructions emulated in software. I/O devices for a VAX reside on I/O buses that have well-defined specifications that are *not* part of the basic architectural specification. Each *VAX system* combines an *implementation* of a VAX CPU with a *system bus*. I/O devices may be directly connected to the computer's system bus or, more commonly, to existing I/O buses that are connected to the system buses through hardware *bus adapters* (see Fig. 13.2). Each physical connection to the system bus of a VAX, including the bus adapters, is made through a *nexus*. A nexus does any system-bus arbitration that devices need to access the system bus. Each nexus also provides an 8-Kbyte bank of registers on the system bus through which the CPU can access and control the associated device. The registers associated with a nexus are located in the I/O portion of the *system region*. Nexuses are

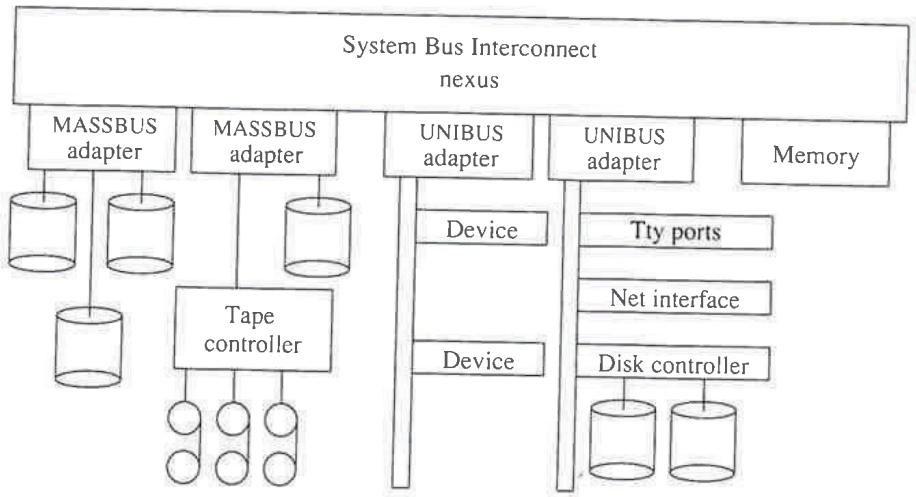


Figure 13.2 Basic VAX computer architecture.

self-describing; one register in each nexus register bank contains a value that uniquely identifies the type of device attached to that nexus.

Autoconfiguration works by systematically probing the possible nexuses on a VAX looking for active connections to the system bus. For each nexus that is in use, the type of device attached to the nexus is interpreted and, depending on this type, the necessary actions are taken to initialize and configure the device. For memory controllers attached to a nexus, the system must initialize the controller and enable the reporting of memory errors. For bus adapters, the configuration routines must probe for I/O devices that might be present on the associated I/O buses.

To configure an I/O device, the system requires that the device be specified at *configuration time* as potentially present in the system (see Section 13.5). The set of possible devices is organized into two *initialization tables*, one for devices that might reside on a MASSBUS and one for devices that might be present on a UNIBUS. When an I/O adapter is encountered at autoconfiguration time, the system searches the initialization tables for devices that might be configured on that adapter. For each matching device, a probing operation is made either by the device driver (for UNIBUS devices) or by the autoconfiguration code (for MASSBUS devices) and, if the probe is successful, the device is recorded in the system data structures as a *configured device* (see Section 8.5).

Aside from the work involved in probing for devices, the configuration process must also identify where I/O devices will interrupt, and then must install the appropriate interrupt service routines so that future interrupts from the devices will be handled by the supporting device drivers. The dynamic configuration of I/O devices and interrupt handlers permits a single kernel to be configured for a number of different hardware configurations; this flexibility is important in minimizing system maintenance.

To configure interrupt handlers dynamically, the system initializes all the possible interrupt vectors for an I/O adapter to point to a special-purpose routine. This routine simply records the interrupt vector and the interrupt priority level at which an interrupt occurs in well-known registers, and then returns. The device-driver probe routines are written such that they define a usage for these registers, but they never actually load a value into the registers. At boot time, the system modifies the probe routines such that they will not save the affected registers on entry. This permits values placed in the registers before a probe routine is called to be altered by an invocation of the special-purpose interrupt handler. To probe for a device and to configure its interrupt vectors, the system can simply initialize the registers to known values, call the device driver to probe the device and to generate an interrupt, and then check the values in the registers to see whether the device has interrupted. For example, the probe routine for the *up* device driver is

```
upprobe(reg)
    caddr_t reg;
{
    register int br, cvec;          /* must be r11, r10 */

                /* probe device and generate interrupt */
    ((struct updevice *)reg)->upcs1 = UP_IE|UP_RDY;
    DELAY(10);
                    /* disable interrupt */
    ((struct updevice *)reg)->upcs1 = 0;
    return (sizeof (struct updevice));
}
```

On the VAX, the declaration of the variables *br* and *cvec* cause the C compiler to reserve registers R11 and R10 for use within the routine. Normally, these registers would be saved on entry to the routine, but alteration of the VAX *register save mask* for the *upprobe*() routine allows this action to be defeated. When *upprobe*() is called and the device interrupts, the registers will be set to contain the interrupt priority level and interrupt vector for the device. Because the save mask has been altered to prevent R10 and R11 from being restored to their previous values, these registers will still contain the interrupt priority level and interrupt vector for the device in the routine that called *upprobe*().

The technique used to configure interrupt vectors depends on the 4.3BSD C compiler's implementation of the *register* declaration. If instead register declarations were interpreted as *advisory*, or the registers allocated to the variables were assigned in an indeterminate fashion, then this scheme would not work. When an ANSI C compiler is used, it is impossible to force a particular variable to be in a register. However, it is possible to force a variable to be allocated to memory with use of the *volatile* keyword. When this feature is used, the calling routine has to arrange to pass the address of a volatile variable in which to place the result.

Interrupt vectors that are not assigned to devices configured at boot time remain pointing to the initial interrupt handler. A global variable, *cold*, is set to 1 by the system until after the autoconfiguration phase of the startup procedure is completed. The interrupt handler checks this variable, and, if the value is nonzero, it takes the actions described; otherwise, the handler generates a diagnostic on the console.

Although VMS can load device drivers after the system is completely booted, 4.3BSD cannot load device drivers because

- 4.3BSD device drivers use some shared global data structures
- Some buses, such as the UNIBUS, are not globally addressable
- A new device might interrupt at the same location as an existing device, leading to confusion

MASSBUS devices do not have these problems, and could be loaded later. The developers did not implement specific support for this capability in 4.3BSD because they did not think it would be useful.

Machine-Independent Initialization

With the static system data structures allocated and the I/O devices configured and initialized, the system is ready to complete the initialization procedure and to start up the first few processes. The first action of the *main*() routine on return from *startup*() is to set up the context for process 0; the process that will eventually implement the swapping policy of the virtual-memory system. The first entry in the process table is allocated for process 0, and the process is marked runnable and is installed as the currently running process (see Chapter 4). The user structure, run-time stack, and process control block for this process were initialized in the assembly-language startup code, so only minor work is required to complete the initialization. The last page of the run-time stack is marked read-only to act as a *red zone*; this ensures that the process will not expand its stack beyond the fixed space allocated to it without causing a system trap. Various pointers in the user structure are initialized, and system default parameters in the user structure that are inherited across a *fork* system call are established. The latter include the resource limits, the file-creation mask, and the group identifier array.

When process 0 has been crafted, various routines are called to initialize each system data structure:

- The *vminit*() routine sets up the parameters used by the paging system. These parameters are dependent on the amount of available physical memory and the number of I/O devices that will be used for paging and swapping. The resource limits on a process's stack and data segments, as well as the resident-set size, are installed in the user structure of process 0. These limits will then be automatically inherited by all other processes, since they are descendents of process 0.

- The disk quota resource-control system is initialized; it includes the cache of quotas for resident processes as well as the cache of per-user data structures.
- The real-time clock is started through a call to *startrtclock*(). This routine primes the necessary hardware that supplies regular interrupts to the system. The clock rate, if programmable, is set up according to the *hz* variable that is defined at the time the system is configured. By default, 4.3BSD runs with a 100-hertz real-time clock. This value can be altered, but selecting a frequency of less than 50 hertz degrades the system's response time to I/O devices that are polled. For some hardware terminal multiplexers, lowering the clock frequency can permit high data-flow rates to swamp input buffers. A poorly chosen clock frequency can also cause roundoff errors in certain calculations. For example, with a 60-hertz clock rate, integer calculations involving the clock frequency will skew. The system normally avoids this problem by maintaining timeout values in terms of clock ticks. A final consideration in choosing a clock frequency is that the frequency defines the minimal observable time interval in the system. This is important for statistical calculations, such as program profiling and accounting, where *entire* clock ticks are charged to a process- or program-counter value at the time a real-time clock interrupt is serviced. In general, the clock frequency should be selected to be as high as possible without too much system overhead being incurred.

Following the initialization of the clock, the network memory-management system is initialized with a call to *mbinit*() (see Chapter 10). The character list data structures used by the terminal I/O facilities are set up through a call to *cinit*(), and subsequent calls are then made to initialize

- The network interfaces
- The communication domains and network communication protocols
- The process-management data structures
- The text table
- The hashed inode cache and inode table
- The swap-space management data structures
- The systemwide filesystem name cache

Before the system can reach single-user operation, it must still mount the root filesystem, and create process 1 (the process that executes **/etc/init**) and process 2 (the process that copies pages of memory to secondary storage in support of the virtual-memory system). The identity of the root filesystem is defined by the value of the *rootdev* variable. This value is initially defined at configuration time, although users may change it at boot time by bootstrapping the system from a device other than the configured one (see the previous discussion of autoconfiguration).

The root inode of the mounted root filesystem is needed to initialize the

current working directory and root directory for process 0. In addition, the last modification date in the superblock of the root filesystem is used in initializing the system's time of day. The timestamp from the superblock is compared to any current value for the time of day available in hardware, and the current time of day is constrained to be within 6 months of the time in the filesystem (unless the filesystem time is completely unbelievable). This consistency check ensures that the system will be bootstrapped with a reasonably accurate time of day. User-level facilities—such as the *timedaemon*, **timed**—support time synchronization and recalibration in a network environment.

Finally, the system is ready to execute user-mode programs. Process 1 is created with a call to *newproc*(), then the contents of a small assembly-language program that is compiled into the kernel are copied into the process's user address space and the process is set executing. The contents of this program are basically as follows:

```
main() {
        char *argv[2];

        argv[0] = "init";
        argv[1] = 0;
        exit(execv("/etc/init", argv));
}
```

That is, the program starts up the **/etc/init** program, exiting if the call to *execv* fails. This makes **/etc/init** critical to operation: If it does not run correctly, if it is not there, or if the parts of the filesystem necessary to reach it are damaged, the system cannot be booted from that filesystem. This error is more serious than is an incorrect **/vmunix**, because the bootstrap allows naming of a different object file for the kernel, but there is no direct way to specify a different **init** program. When enough disk space is available, the usual backup is to keep a different filesystem available for such pathological cases.

The second process to be started is the *pagedaemon*, with process identifier 2. This process executes entirely in kernel mode by invoking the *pageout*() routine, a procedure that never returns. Like process 0, the *pagedaemon* marks its process structure to ensure that that structure will not be removed from memory. The *pagedaemon* also expands its data segment to provide itself room to map pages of memory it will be writing to secondary storage (see Section 5.12).

The last action of *main*() is to call the *sched*() routine within process 0. Like the *pagedaemon*, this process executes entirely in kernel mode, and the call to *sched*() never returns. The *sched*() routine implements the swapping policy of the virtual-memory system; it is described in Section 5.13.

13.4 User-Level Initialization

With the start of execution of process 1, most of the system is operating and functional. There are several additional steps taken between this point and the time a

user sees a prompt to sign on to the system. All these actions are done by user-level programs that use the standard UNIX system-call interface that has been described in previous chapters. We shall briefly examine the steps that take place in a typical 4.3BSD system; be aware, however, that these steps can vary drastically among UNIX systems.

/etc/init

The **/etc/init** program is invoked as the last step in the bootstrapping procedure. The parameters specified at the time UNIX was boostrapped are passed to **init** in a machine-dependent fashion. On the VAX, the flags passed to the **boot** program are passed to **init** in the same hardware register as that in which they were passed to the kernel. **Init** uses the values of these flags to determine whether it should bring the system up to single- or to multiuser operation. In single-user operation, **init** forks a process that invokes the standard UNIX shell, **/bin/sh**. The standard input, output, and error descriptors of the process are directed to the system's console terminal, **/dev/console**. This shell then operates normally, but with superuser privileges, until it terminates.

In multiuser operation, **init** first spawns a shell to interpret the commands in the file **/etc/rc**. These commands do filesystem consistency checks, start up system processes, and initialize database files, such as the **/etc/mtab** file that contains the list of mounted filesystems. If the **/etc/rc** script completes successfully, **init** then forks a copy of itself for each terminal device that is marked for use in the file **/etc/ttys**. These copies of **init** invoke other system programs, such as **/etc/getty**, to manage the standard sign on procedure. Process 1 always acts as the master coordinating process for system operation. It is responsible for spawning new processes as terminal sessions are terminated, and for managing the shutdown of a running system.

/etc/rc

The **/etc/rc** command script first checks the integrity of the filesystems. This check is necessary to ensure that any damage that might have occurred from a previous system failure is repaired. The filesystem support within the kernel is concerned solely with reading and writing existing filesystems. Any inconsistencies in a filesystem are repaired by user-level programs.

The program **/etc/fsck** is the major tool used in checking filesystem consistency and in repairing damaged filesystems. Normally, **fsck** is invoked from the **/etc/rc** script to examine and repair each filesystem before it is mounted. If the root filesystem, mounted by the kernel before **/etc/rc** is invoked, requires repairs, 4.3BSD must reboot itself to ensure consistency between the data in the buffer cache and the data in the filesystem. This reboot operation is done automatically, as described later.

Following the filesystem checks, the filesystems are mounted and any devices that are to be used for swapping and paging are enabled. Disk quotas are then checked and enabled, and the system starts the background processes that implement various system services. These processes include **/etc/update**, the program

that flushes the disk writes from the buffer cache every 30 seconds; **/etc/cron**, the program that executes commands at periodic time intervals; **/etc/accton**, the program that enables system accounting; and **/etc/syslogd**, the system error-logging process. Some of these processes are started from the command script **/etc/rc.local**. The commands in **/etc/rc.local** are tailored according to the needs of each host, whereas the commands in **/etc/rc** are common to all hosts. For example, processes that provide network services are typically started up from the **/etc/rc.local** command file.

/etc/getty

The **/etc/getty** program is spawned by **/etc/init** for each hardware terminal line on a system. This program is responsible for opening and initializing the terminal line. Opening the terminal causes the system to assign a process group (if the process already had a process group, **getty** uses the *vhangup* system call first to zero it). The **getty** program sets the initial parameters for a terminal line and establishes the type of terminal attached to the line. For lines attached to a modem, **getty** can be directed to accept connections at a variety of baud rates. **Getty** selects this baud rate by changing the speed of the terminal line in response to a break character or a framing error, typically generated as a result of the user hitting a break key. Successive break keys can be hit by a user to cycle through a number of line speeds until the proper one is found. **Getty**'s actions are driven by a terminal-configuration database that is located in the file **/etc/gettytab**.

Getty finally reads a login name, and invokes the **/bin/login** program to complete a login sequence.

/bin/login

The **login** program is responsible for signing a user onto the system; it is usually invoked by **/etc/getty** with the name of the user who wishes to log in to the system. **Login** prompts the user for a password (after turning off terminal echoing if possible). If the password supplied by the user encrypts to the same value as that stored in the master password file **/etc/passwd**, **login** writes a record of the sign on in various accounting files, initializes the user and group identifiers to those specified in the password and **/etc/group** files, and changes to the user's login directory. Finally, **login** uses *exec* to overlay itself with the user's shell.

The **login** program is also invoked when a user enters the system through a network connection; **getty** and **init** are bypassed for such connections. Before this complication was added (in 4.2BSD), the system could recognize a login shell by noticing that the shell's parent process was **init**. This was useful for accounting and for limiting the number of simultaneous logins. The flag SLOGIN was added to the proc structure for **login** to set to mark a login shell. The 4.3BSD **login** does not set or use this flag, but the flag has been used elsewhere [Elz, 1984].

13.5 System Startup Topics

In this section, we consider topics that are related to the system startup procedure.

Kernel Configuration

The software that makes up a 4.3BSD kernel is defined by a *configuration file* that is interpreted by the **/etc/config** program. Configuration files specify the hardware and software components that should be supported by a kernel. The **config** program uses the configuration file to generate a set of data files and an input file for the **make** program that are used to compile and link a kernel load image.

A sample configuration file for a VAX computer is shown in Fig. 13.3. Among the most important *global* configuration parameters are

- The type of machine on which the system is run; *vax* in our example.
- The type of CPU to be supported. Many CPU types may be specified, and the system will attempt to support the one on which it has been bootstrapped. The CPU types specified in our example are *VAX780* and *VAX750*.
- The time zone in which the system will operate and, optionally, a daylight-saving-time correction algorithm. The system normally runs on Universal Coordinated Time (UTC); the time zone and daylight-saving-time correction algorithm are used by applications in calculating the local time of day. In Fig. 13.3, the Pacific standard time zone is specified as 8 hours past UTC, and the standard daylight-saving-time algorithm is requested (see also Section 3.5).
- The maximum expected number of simultaneously active users on the system. This value is used to calculate the sizes of numerous data structures.

Additional parameters can be specified to enable the inclusion of optional code.

Figure 13.3 A sample configuration file for a VAX.

```
machine     vax
cpu         "VAX780"
cpu         "VAX750"
timezone    8 dst
maxusers    16

config      vmunix      root on hp0

controller  mba0        at nexus ?
controller  uba0        at nexus ?
controller  uba1        at nexus ?
disk        hp0         at mba? drive 0
disk        hp1         at mba? drive 1
controller  hk0         at uba? csr 0177440              vector rkintr
disk        rk0         at hk0 drive 0
device      dz0         at uba? csr 0160120 flags 0xff   vector dzrint dzxint
controller  zs0         at uba? csr 0172520              vector tsintr
device      ts0         at zs0 drive 0
```

For example, the filesystem disk-quota support is not normally included in a system; to force it to be compiled into the system, the line "options QUOTA" would be specified.

The *config* specification in Fig. 13.3 indicates that **/etc/config** should arrange to create a kernel load image in the file **vmunix** that has its root filesystem on the disk drive named *hp0*. The remainder of the configuration file describes the hardware devices that may be present on the machine. Specifications that include question marks act as *wildcards*. For example, *uba?* specifies that a device might be present on either of the two UNIBUS adapters described in the configuration file.

Disks and tapes are configured in terms of controllers and devices. This division reflects the actual hardware topology present on a machine. The *csr* specifications identify the base address of a controller's bank of registers. The *vector* specification identifies the name and number of interrupt handlers (or vectors) for a device. These names are actually the names of the interrupt service routines in the device drivers that have been written to support the hardware.

The configuration file is used by **/etc/config** to generate a number of output files, several of which are compiled and linked into the kernel's load image:

- A file that describes the hardware device topology and the devices that might be present on the machine
- A file that includes assembly-language routines that connect the hardware interrupt-vector entry points to the device-driver interrupt handlers specified in the configuration file
- A file that defines the devices to use for the root filesystem and for swapping and paging
- A number of small header files that control conditional compilations of source code
- A file for the *make* program that is used to compile and link the kernel load image

A complete description of the configuration process and **/etc/config** can be found in [Leffler & Karels, 1986].

System Shutdown and Autoreboot

4.3BSD provides several utility programs to halt, reboot, or bring a system from multiuser to single-user operation. Safe halting and rebooting of a system requires support from the kernel. This support is provided by a *reboot* system call.

The *reboot* system call is a privileged call. A single parameter specifies how the system should be shut down and rebooted. This parameter is a superset of the flags passed by the **boot** program to the system when the latter is initially bootstrapped (see Table 13.1). A system can be brought to a halt (typically by its being forced to execute an infinite loop), or it can be rebooted to single- or multiuser operation. There are additional controls that can be used to force a crash

dump before rebooting (see the next section for information about crash dumps), and to disable the writing of data that are in the buffer cache to disk (in case the information in the buffer cache is wrong).

On most hardware, rebooting requires support from the console processor or monitor. Typically, a reboot operation is initiated by a command being passed to the console processor. This command causes the system to be rebooted as though someone had typed the appropriate commands on the console terminal. Automatic rebooting is also commonly done by the console processor when a catastrophic failure is recognized. The system will automatically reboot itself if it recognizes an unrecoverable failure during normal operation. Failures of this sort, termed *panics*, are all handled by the *panic*() subroutine. 4.1BSD was one of the first UNIX systems to be able to recover *automatically* from catastrophic failures by rebooting, repairing any filesystem damage, and then restarting normal operation. Facilities to *checkpoint* the state of active processes and automatically to resume the processes' execution after a system reboot have been added by some vendors, such as Cray Research.

System Debugging

4.3BSD provides several facilities for debugging system failures. The most commonly used facility is the *crash dump*, a copy of memory that is saved on secondary storage by the kernel in the event of a catastrophic failure. Crash dumps are created by the *doadump*() routine. They occur if a reboot system call is made in which the RB_DUMP flag is specified, or if the system encounters an unrecoverable, and unexpected, error.

The *doadump*() routine disables virtual-address translation, raises the processor priority level to its highest value to block out all device interrupts, and then invokes the *dumpsys*() routine to write the contents of physical memory to secondary storage. The precise location of a crash dump is configurable; most systems place the information at the back of the primary swap partition. The device driver's dump entry point is used to do this operation.

A crash dump is retrieved from its location on disk after the system is rebooted and the filesystems have been checked. The **/etc/savecore** program exists solely for this purpose. It creates a file into which the crash-dump image is copied. **Savecore** also makes a copy of the initial kernel load image, **/vmunix**, for use in debugging. Crash dumps can be examined with either of the 4.3BSD debugging programs, **adb** or **dbx**. Both debuggers can also be used to examine and modify a running system through the **/dev/mem** special file. More information about the use of **adb** for kernel debugging can be found in [Leffler & Joy, 1986].

Exercises

13.1 What is the purpose of the **boot** program?

13.2 What is the job of the machine-language startup? Why is this program written in machine language?

13.3 What is the purpose of the kernel's message buffer?

13.4 What are the first three processes started when the system is booted?

13.5 Assume that **/boot** is read in from the console media. Name the three other files that must be present to boot the system to single-user mode.

13.6 Why are kernel data structures sized when the system boots, instead of when the kernel is compiled?

13.7 What information is supplied in a configuration file that cannot be determined at the time the system is booted?

13.8 The *reboot* system call causes the system to halt or reboot. Give three reasons why this system call is useful.

*13.9 Suppose a machine does not have a battery-backup time-of-day clock. Propose a method for determining that the time-of-day clock is incorrect and describe a way to initialize its time of day. What are the limitations of your method?

References

Elz, 1984.
K. R. Elz, "Resource Controls, Privileges, and Other MUSH," *USENIX Association Conference Proceedings*, pp. 183–191 (June 1984).

Leffler & Karels, 1986.
S. J. Leffler & M. J. Karels, "Building Berkeley UNIX Systems with Config," pp. 2:1–32 in *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, USENIX Association, Berkeley, CA (1986).

Leffler & Joy, 1986.
S. J. Leffler & W. N. Joy, "Using ADB to Debug the UNIX Kernel," pp. 3:1–11 in *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, USENIX Association, Berkeley, CA (1986).

Glossary

absolute pathname *See* pathname.

access rights In an operating system, the rights of processes to access system-maintained objects. For example, the ability to write data into a file. Rights are recognized and enforced by the system and typically are associated with capabilities. The passing of access rights in messages is supported by the 4.3BSD interprocess-communication facilities. For example, the UNIX communication domain supports the transmission of file descriptors and their associated access rights.

adb A program debugger that was first introduced in the Seventh Edition UNIX system.

address family A collection of related address formats, as found in a communication domain.

address format A set of rules used in creating network addresses of a particular format. For example, in the Internet communication domain, a host address is a 32-bit value that is encoded using one of four rules according to the type of network on which the host resides.

Address Resolution Protocol (ARP) A communication protocol used to map one network address to another dynamically. For example, ARP is used in 4.3BSD to map DARPA Internet addresses into Ethernet addresses dynamically.

address translation A mechanism, typically implemented in hardware, that translates memory addresses supplied by a program into physical memory addresses. This facility is important in supporting multiprogramming because it allows an operating system to load programs into different areas of memory, and yet to have each program execute as though it were loaded at a single, fixed memory location.

advisory lock A lock that is enforced only when a process explicitly requests its enforcement. An advisory lock is contrasted with a hard lock, the enforcement of which is mandatory.

AGE buffer list A list in the filesystem buffer cache. This list holds buffers whose contents have not yet proven useful—for example, read-ahead blocks.

ARP *See* Address Resolution Protocol.

ARPANET The Advanced Research Projects Agency Network. A packet-switching network established in December 1969 by DARPA.

ARPANET Host Access Protocol A protocol used to communicate between a host and an ARPANET PSN.

association In the interprocess-communication facilities, a logical binding between two communication endpoints that must be established before communication can take place. Associations may be long-lived, such as in virtual-circuit–based communication, or short-lived, such as in a datagram-based communication paradigm.

AST *See* asynchronous system trap.

asynchronous system trap (AST) On the VAX, a software-initiated interrupt to a service routine. ASTs enable a process to be notified of the occurrence of a specific event asynchronously with respect to its execution. In 4.3BSD, ASTs are used to initiate process rescheduling.

attach routine A device-driver routine that is used during the autoconfiguration phase to establish the software state for a device. Attach routines are called for each device that the system has probed successfully.

autoconfiguration phase A phase of operation the system goes through when bootstrapping itself into operation. In the autoconfiguration phase, the system probes for hardware devices that might be present in the machine, and attaches each device that it locates. *See also* attach routine; probe routine; slave routine.

background process In job-control–oriented process-management systems, a process whose process group is different from that of its controlling terminal, and thus, is currently blocked from most terminal access. Otherwise, a background process is one for which the command interpreter is not waiting; that is, the process was set running with the "&" operator. The opposite of a background process is a *foreground process*.

backing storage Storage that is used to hold objects that are removed from main memory during paging and swapping operations. *See also* secondary storage.

bad sector A sector with an uncorrectable media defect.

bad-sector forwarding A facility whereby bad sectors on a disk are transparently replaced with good sectors. The system does this replacement by *forwarding* requests for each bad sector to the replacement good sector. The good sectors are taken from reserved areas of a disk. Bad-sector forwarding

can be done on the host in the device driver, or, as is becoming commonplace, in the disk controller. *See also* slip-sector forwarding.

bad-sector table A table used in bad-sector forwarding. During bad-sector forwarding, the system looks up the location of the good (forwarded) sector in the bad-sector table, using the bad-sector number as a key.

Berknet An obsolete network developed at the University of California, Berkeley that used serial lines for communication.

block In the filesystem, a unit of allocation. The filesystem allocates space in block-size units, or in fragments of block-size units.

block device A random-access mass-storage device that supports a block-oriented interface; for example, a disk drive.

block-device interface The conventions established for accessing block devices within the kernel. These conventions include the set of procedures that can be called to perform I/O operations, as well as the parameters that must be passed in each call.

block-device table A table within the kernel in which the device-driver routines that support the block-device interface for each device are recorded. The ordering of entries in the block-device table is important, since it defines the major-device number for block devices.

block I/O I/O to a block device.

block size The natural unit of space allocated to a file (*filesystem block size*), or the smallest unit of I/O that a block device can perform (for disk devices, usually the sector size). In 4.3BSD, the filesystem block size is a parameter of the filesystem that is fixed at the time the filesystem is created.

bootdevice register The register in which the first-level bootstrap passes the identity of the device from which the system should be bootstrapped. On the VAX, the bootdevice register is R10.

boothowto register The register in which the first-level bootstrap passes a code that indicates whether the system should be bootstrapped to multiuser or to single-user operation, and whether or not to prompt for the name of the file to bootstrap. On the VAX, the boothowto register is R11.

bootstrapping Bringing a system up into an operational state. When a machine is first powered on, it is typically not running any program. Bootstrapping initializes the machine, loads a program from secondary storage into main memory, and sets that program running.

bottom half With regard to system operation, the collection of routines in the kernel that are invoked as a result of interrupts. These routines cannot depend on any per-process state and, as a result, cannot block by calling the *sleep*() routine. *See also* top half.

breakpoint fault A hardware trap that is generated when a process executes a breakpoint instruction.

broadcast To transmit to all parties. In a network, a *broadcast message* is transmitted to all stations attached to a common communication medium.

bss segment The portion of a program that is to be initialized to zero at the time the program is loaded into memory. The name *bss* is an abbreviation for "block started by symbol."

buffered As in "buffered I/O"; a technique whereby data are held, or buffered, to minimize the number of I/O operations that are performed. For example, the standard I/O library buffers output to files by accumulating data to be written until there is a full filesystem block to write, or until a flush request is made by the application.

buffered data path A data path in a UNIBUS adapter that buffers data transfers between the UNIBUS and a VAX's system bus during DMA operations. *See also* unbuffered data path.

bus A standardized electrical and mechanical interconnection for components of a computer.

bus adapter A hardware device that permits one style of I/O bus to be connected to another. *See also* MASSBUS adapter and UNIBUS adapter.

byte A unit of measure applied to data. A byte is almost always 8 bits. *See also* octet.

C-block The data buffer that holds the actual data in a C-list data structure.

C-list A linked-list data structure used by the system in supporting terminal I/O.

capability Data presented by a process to gain access to an object. *See also* access rights.

catenet A network in which hosts are connected to networks with varying characteristics, and the networks are interconnected by gateways. The DARPA Internet is an example of a catenet.

cathode ray tube (CRT) A screen-display device commonly used in computer terminals. A terminal that includes a CRT is often called a CRT.

caught signal A signal is said to be "caught" if the delivery of the signal to a process results in a signal-handler procedure being invoked. A signal handler is installed by a process with the *sigvec* system call.

cbreak mode A mode of operation for a terminal device whereby processes reading from the terminal receive input immediately as it is typed. This mode differs from raw mode in that certain input processing, such as interpreting the interrupt character, is still performed by the system. *See also* cooked mode and raw mode.

central processing unit (CPU) The primary computational unit in a computer. The CPU is the processing unit that executes applications. Additional processing units may be present in a computer for handling, for example, I/O.

character A datum that represents a single printable or control symbol. Characters are usually 8 or 16 bits long. *See also* byte and octet.

character device A device that provides either a character-stream oriented I/O interface or, alternatively, an unstructured (raw) interface. For example, a

terminal multiplexer is a character device that exhibits a character-oriented I/O interface, whereas all magnetic-tape devices support a character-device interface that provides a raw interface to the hardware. Devices that are not character devices are usually block devices.

character-device interface The conventions established for accessing character-oriented devices within the kernel. These conventions include the set of procedures that can be called to do I/O operations, as well as the parameters that must be passed in each call.

character-device table A table within the kernel in which the device-driver routines that support the character-device interface for each device are recorded. The ordering of entries in the character-device table is important, since it defines the major-device number for character devices.

checksum The value of a mathematical function computed for a block of data used to detect corruption of the data block.

child process A process that is a direct descendent of another process as a result of being created with a *fork* system call.

client process In the client-server model of communication, a process that contacts a server process to request services. A client process is usually unrelated to a server process; the client process's only association with the server process is through a communication channel.

CLOCK algorithm *See* global CLOCK algorithm.

clock hand An abstraction used in describing the global CLOCK algorithm used by the virtual-memory system in its page-replacement strategy. In this algorithm, all the pages of main memory are envisioned in a circular loop, and two pointers shuttle through the pages in the loop as clock hands move on the face of a clock.

cluster The logical grouping of contiguous physical pages of memory. In 4.3BSD, this grouping is used by the virtual-memory system to simulate memory pages with sizes larger than the physical page size supported by the hardware.

cold start The initial phase of a bootstrap procedure. The term is derived from the fact that the software assumes nothing about the state of the machine—as though the machine had just been turned on and were cold.

communication domain An abstraction used by the interprocess-communication facilities to organize the properties of a communication network or similar facility. A communication domain includes a set of protocols, termed the *protocol family*; rules for manipulating and interpreting names; the *address family*; and, possibly, other intrinsic properties, such as the ability to transmit access rights. The facilities provided by the system for interprocess communication are defined such that they are independent of the communication domains supported by the system. This design makes it possible for applications to be written in a communication-domain–independent manner.

communication protocol A set of conventions and rules used by two communicating processes.

configuration file A file that contains parameters for the system-configuration program **/etc/config**. This file describes the hardware devices and topology that the system should be able to support, as well as miscellaneous parameters, such as the maximum number of users that are expected to be using the system simultaneously.

configuration procedure The procedure followed by a system administrator in configuring a kernel for a machine, or for a collection of machines. The configuration procedure involves defining a configuration file, which is then supplied to the **/etc/config** program to create the necessary data files for building a kernel.

connect request A request passed to the user-request routine of a communication-protocol module as a result of a process making a *connect* system call on a socket. The request causes the system to attempt to establish an association between a local and a remote socket.

console monitor The terminal attached to a console-terminal interface.

console processor An auxiliary processor to the main CPU that allows an operator to start and stop the system, to monitor system operation, and to run hardware diagnostics.

context switching The action of interrupting the currently running process and switching to another process. Context switching occurs as one process after another is scheduled for execution. On the VAX, an interrupted process's hardware context is saved in that process's process-control block using the *svpctx* instruction, and another process's hardware context is loaded with the *ldpctx* instruction.

continue signal Signal 19 (SIGCONT). A signal that, when delivered to a stopped or sleeping process, causes that process to resume execution.

control request A request passed to the user-request routine of a communication-protocol module as a result of a process making an *ioctl* system call on a socket.

controlling terminal The terminal device for a process from which keyboard-related signals may be generated. The controlling terminal for a process is normally inherited from the process's parent. If the parent process has no controlling terminal, the controlling terminal is the first terminal device a process opens.

cooked mode A mode of operation for a terminal device whereby processes reading from the terminal receive input only after a special end-of-line character, usually a carriage-return, is typed. Cooked mode is the normal mode of operation for a terminal. It is used, for example, by most command interpreters. *See also* cbreak mode and raw mode.

copy-on-write A technique whereby multiple references to a common object are maintained until the object is modified (written). Before the object is written, a copy is made and the modification is made to the copy, rather than to the

original. In virtual-memory management, copy-on-write is a common scheme used to manage pages shared by multiple processes. All the PTEs mapping a shared page are set such that the first write reference to the page causes a page fault. In servicing the page fault, the faulted page is replaced with a private copy, which is writable.

core file A file (named **core**) that is created by the system when certain signals are delivered to a process. The file contains a record of the state of the process at the time the signal occurred. This record includes the contents of the process's virtual address space and, on most systems, the user structure.

core map The data structure used by the 4.3BSD kernel to manage main memory. The core map contains one entry for each cluster of main memory.

CPU *See* central processing unit.

crash Among computer scientists, an unexpected system failure.

crash dump A record of the state of a machine at the time of a crash. This record is usually written to a place on secondary storage that is thought to be safe, to be saved until the information can be recovered.

CRT *See* cathode ray tube.

current working directory The directory from which relative pathnames are interpreted for a process. The current working directory for a process is set with the *chdir* system call.

cylinder The tracks of a disk that are accessible from one position of the head assembly.

cylinder group In the filesystem, a collection of cylinders on a disk drive that are grouped together for the purpose of localizing information. That is, the filesystem allocates inodes and data blocks on a per–cylinder-group basis.

daemon A long-lived process that provides a system-related service. There are daemon processes that execute in kernel mode (e.g., the *pagedaemon*), and daemon processes that execute in user mode (e.g., the *routing daemon*). The old English term, *daemon*, means "a deified being", as distinguished from the term, *demon*, which means an "evil spirit."

DARPA Defense Advanced Research Projects Agency. An agency of the U.S. Department of Defense that is responsible for managing defense-sponsored research in the United States.

data path In a UNIBUS adapter, a pathway between the UNIBUS and the VAX's system bus through which data are transferred during a DMA operation.

data segment The segment of a process's address space that contains the initialized and uninitialized data portions of a program.

datagram socket A type of socket that models potentially unreliable connectionless packet communication.

dbx The name of a program debugger that was first introduced in 4.2BSD.

decapsulation In network communication, the removal of the outermost header information on a message. The inverse of encapsulation.

demand paging A memory-management technique in which memory is divided into pages and the pages are provided to processes as needed—that is, *on demand*. *See also* pure demand paging.

descriptor An integer assigned by the system when a file is referenced by the *open* system call, or when a socket is created with the *socket*, *pipe*, or *socketpair* system calls. The integer uniquely identifies an access path to the file or socket from a given process, or from any of that process's children. Descriptors can also be duplicated with the *dup* and *fcntl* system calls.

descriptor table A per-process table that holds references to objects on which I/O may be performed. I/O descriptors are indices into this table.

device In UNIX, a peripheral connected to the CPU.

device driver A software module that is part of the kernel and that supports access to a peripheral device.

device flags Data specified in a system configuration file and passed to a device driver. The use of these flags varies between device drivers. Device drivers for terminal devices use the flags to indicate the terminal lines on which the driver should ignore modem-control signals on input.

device number A number that uniquely identifies a device within the block or character device classes. A device number comprises two parts: a major-device number and a minor-device number.

device special file A file through which processes can access hardware devices on a machine. For example, a tape drive is accessed through such a file.

direct memory access (DMA) A facility whereby a peripheral device can access main memory without the assistance of the CPU. DMA is typically used to transfer contiguous blocks of data between main memory and a peripheral device.

directed broadcast A message that is to be broadcast on a network to which the sender is not connected directly.

directory In UNIX, a special type of file that contains entries that are references to other files. By convention, a directory contains at least two entries, dot (.) and dot-dot (..). Dot refers to the directory itself; dot-dot refers to the parent directory.

directory entry An entry in a directory that is represented by a variable-length record structure in the directory file. Each structure holds an ASCII string that is the filename, the number of bytes of space provided for the string, the number of bytes of space provided for the entry, and the number of the inode associated with the filename. By convention, a directory entry with a zero inode number is treated as unallocated, and the space held by the entry is available for use.

dirty In computer systems, an object that has been modified is said to be "dirty." A system usually tracks whether or not an object is dirty because it needs to save the object's contents before reusing the space held by the object. For example, in the filesystem, a buffer in the buffer cache is dirty if

its contents have been modified. Dirty buffers must eventually be written back to the disk.

disk partition A contiguous region of a disk drive that is used as a swap area or to hold a filesystem.

distributed program A program that is partitioned among multiple processes, possibly spread across multiple machines.

DMA *See* direct memory access.

double indirect block *See* indirect block.

ECC correction A procedure whereby corrupted data are detected and corrected through the use of a redundant error-correction code (ECC). ECC correction is common in disk subsystems. The ECC is recorded with each disk sector and is used to identify and correct bit errors in the recorded data.

effective GID *See* effective group identifier.

effective group identifier (effective GID) The GID that the system uses, with the GIDs in the groups array, to check filesystem group access permission. The effective GID is set when a set-group-identifier program is executed. *See also* group identifier and real group identifier.

effective UID *See* effective user identifier.

effective user identifier (effective UID) The UID that the system uses to check many user permissions. For example, the effective UID is used by the filesystem when checking owner access permission on files. The effective UID is set when a set-user-identifier program is executed. *See also* user identifier and real user identifier.

elevator sorting algorithm An algorithm used by the device drivers for I/O requests for moving head disks. The algorithm sorts requests into a cyclic ascending order based on the cylinder number of the request. The algorithm's name is derived from the fact that it orders disk requests in a manner similar to the way ride requests would best be handled for an elevator.

emulate To simulate.

encapsulation In network communication, the procedure by which a message is created that has an existing message enclosed in it as data. A protocol normally encapsulates a message by crafting a leading protocol header that indicates the original message is to be treated as data. The inverse of *decapsulation*.

erase character The ASCII character that is recognized by the terminal handler in cooked mode to mean "delete the last character in the line of input." Each terminal can have a different erase character, and that erase character can be changed at any time with an *ioctl* system call. The terminal handler does not recognize the erase character on terminals that are in raw or cbreak mode. *See also* word-erase character and kill character.

errno The global variable in C programs that holds an error code that indicates why a system call failed. On the VAX, the value placed in errno is returned by the kernel in register R0; it is moved to *errno* by code in the C run-time library.

error-message buffer *See* message buffer.

fault rate The rate at which a process generates page faults. For a reference string, the fault rate is defined to be time-independent by its being specified as the number of page faults divided by the length of the reference string.

fetch policy The policy used by a demand-paged virtual-memory–management system in processing page faults. Fetch policies differ primarily in the way they handle prepaging of data.

ffs An instruction on the VAX that locates the first bit that is set to one in a string of bits.

FIFO In the UNIX filesystem, a type of file that can be used for interprocess communication. Data written by one process to a FIFO are read by another in the order in which they were sent. The name refers to the fact that data are transferred in a first-in-first-out fashion.

file An object in the filesystem that is treated as a linear array of bytes. A file has at least one name, and it exists until all its names are deleted explicitly.

file offset A byte offset associated with an open file descriptor. The file offset for a file descriptor is set explicitly with the *lseek* system call, or implicitly as a result of a *read* or *write* system call.

file structure The data structure used by the kernel to hold the information associated with one or more open file descriptors that reference a file. In most cases, each open file descriptor references a unique file structure. File structures may be shared, however, when open descriptors are duplicated with the *dup* and *dup2* system calls, inherited across a *fork* system call, or received in a message through the interprocess-communication facilities.

filename A string of ASCII characters that is used to name an ordinary file, special file, or directory. The characters in a filename cannot include null (0) or the ASCII code for slash ('/'). (On 4.3BSD the eighth bit, or parity bit, must also be 0.)

filesystem A collection of files. The UNIX filesystem is hierarchical, with files organized into directories, and filesystems, in most cases, restricted to a single physical hardware device such as a disk drive. Filesystems typically include facilities for naming files and for controlling access to files.

fill-on-demand klustering A technique used by the virtual-memory system for prepaging data. When a page fault is encountered for a fill-on-demand page, the page-fault handler attempts to read in the desired page together with adjacent pages on either side of the page for which the fault occurred. *See also* kluster.

fill-on-demand page fault The first page fault for an individual page; it must be resolved by retrieval of data from the filesystem or by allocation of a zero-filled page.

fill-on-demand page-table entry A PTE for a page whose reference will generate a fill-on-demand page fault.

finite-state machine A technique for expressing a set of decision-making rules that is based solely on a *current state* and one or more input variables.

first-level bootstrap The initial code that is executed in a multilevel bootstrapping operation. Usually, the first-level bootstrap is limited in size and does little more than bootstrap into operation a larger, more intelligent, program. For example, the first-level-bootstrap on the VAX-11/750 is limited to 512 bytes; its sole task is to bootstrap the **boot** program so that **boot** can, in turn, bootstrap the kernel.

foreground process In job-control–oriented process-management systems, a process whose process group is the same as that of its controlling terminal; thus the process is allowed to read from and write to the terminal. Otherwise, a foreground process is one for which the command interpreter is currently waiting. The opposite of a foreground process is a *background process*.

fragment In the filesystem, a part of a block. The filesystem allocates new disk space to a file as a full block or as one or more fragments of a block. The filesystem uses fragments, rather than allocating space in only full block-size units, to reduce wasted space when the size of a full block is large.

fragment-descriptor table A data structure in the filesystem that describes the fragments that are free in an entry of the allocation map. The filesystem uses the fragment-descriptor table by taking a byte in the allocation map and using the byte to index into the fragment-descriptor table. The value in the fragment-descriptor table indicates how many fragments of a particular size are available in the entry of the allocation map. By doing a logical AND with the bit corresponding to the desired fragment size, the system can determine quickly whether a desired fragment is contained within the allocation-map entry.

free list In the memory-management system, the list of available clusters of physical memory (also called the *memory free list*). There are similar free lists in the system for many data structures: inodes, file-table entries, disk-quota structures, and so on.

free-space reserve A percentage of space in a filesystem that is held in reserve to ensure that certain allocation algorithms used by the filesystem will work well. By default, 10 percent of the available space in a filesystem is held in reserve.

garbage collection A memory-management facility in which unused portions of memory are reclaimed without an application having to release them explicitly.

GID *See* group identifier.

global CLOCK algorithm An algorithm used by the virtual-memory–management system in selecting candidate pages for replacement. The algorithm treats all the pages of main memory as though they were organized in a circle. Two logical hands (in the modified algorithm used in 4.3BSD) then sweep through the pages, with each page at the front hand being marked unreferenced and each page at the back hand checked to see whether it was referenced since the front hand passed over it. If a page is found to be unreferenced when it is checked by the back hand, it is treated as a candidate for replacement; otherwise, it is ignored.

go routine A routine supplied by a device driver for a UNIBUS device. The routine is expected to initiate an I/O operation on a device after necessary resources have been allocated by the UNIBUS adapter-support routines.

group identifier (GID) A 16-bit integer value that uniquely identifies a collection of users. GIDs are used in the access-control facilities provided by the filesystem. *See also* effective group identifier and real group identifier.

half-open A connection that is thought to be open by only one of the two endpoints is said to be "half-open."

handler A procedure that is invoked in response to an event such as a signal.

hard limit A limit that cannot be exceeded. *See also* soft limit.

hard link A directory entry that directly references an inode. If there are multiple hard links to a single inode and if one of the links is deleted, the remaining links still reference the inode. By contrast, a symbolic link is a file that holds a pathname that is used to reference a file.

hard lock A lock that cannot be ignored or avoided. *See also* advisory lock.

heap In UNIX, the region of a process that can be dynamically expanded with the *sbrk* system call. Its name is derived from the disorderly fashion in which data are placed in the region.

high watermark A data-flow control that specifies an upper bound on the number of data that may be buffered. In the interprocess-communication facilities, each socket's data buffer has a high watermark that specifies the maximum number of data that may be queued in the data buffer before a request to send data will block the process (or will return an error if nonblocking I/O is being used).

hole In a file, a region that is part of the file, but that has no associated data blocks. The filesystem returns zero-valued data when a process reads from a hole in a file. A hole is created in a file when a process positions the file pointer past the current end-of-file, writes some data, and then closes the file.

home directory The current working directory that is set for a user's shell when the user logs into a system. This directory is usually private to the user. The home directory for a user is specified in a field in the password-file entry for the user.

Host Dead message A message from an ARPANET PSN that indicates that the host to which a previous message was directed is unavailable because the host is *down*.

host identifier A 32-bit value that is intended to identify uniquely a host within a network of machines. This value is set with the *sethostid* system call at the time the system is bootstrapped. A process may retrieve the value of the host identifier with the *gethostid* system call. Host identifiers are intended for use in constructing networkwide unique identifiers for objects such as files, processes, and users.

Host Unreachable message An ICMP message that indicates that the host to which a previous message was directed is unavailable because there is no known path to the desired host.

ICMP *See* Internet Control Message Protocol.

IDP *See* Internetwork Datagram Protocol.

IMP *See* Interface Message Processor.

indirect block In the filesystem, the first 12 blocks of the file are pointed to directly by the inode. Additional data blocks are described with a pointer from the inode to an *indirect data block*; the system must first fetch the indirect block that holds the number of the data block. In 4.3BSD, as many as two indirect blocks may have to be fetched to locate the desired data block. An indirect block that contains data block numbers is termed a *single-level indirect block*; an indirect block that contains block numbers of single-level indirect blocks is called a *double level indirect block*. 4.3BSD has provision for, but does not use, an indirect block that contains block numbers of double-level indirect blocks; such a block is termed a *triple-level indirect block*.

init The first user program that is started up when the system is bootstrapped. This program (**/etc/init**) is mainly responsible for managing the terminal lines attached to a machine.

initial sequence number *See* sequence space.

inode A data structure used by the filesystem to describe a file. The contents of an inode include the file's type, the UID of the file's owner, and a list of the disk blocks and fragments that make up the file. Note that inodes do not have names; directory entries are used to associate a name with an inode.

inode table The data structure used by the kernel to hold the active inodes in the system. The inode table is always resident in main memory. Inactive entries in the table are reused on an LRU basis.

Input/Output (I/O) The transfer of data between the computer and its peripheral devices.

intelligent gateway A gateway machine that is capable of making intelligent decisions about routing network data. Such machines usually participate in a scheme whereby routing information is updated dynamically to reflect

changes in network topology. An intelligent gateway is also expected to respond to hosts that make poor routing decisions with routing redirect messages.

interactive program A program that must periodically obtain user input to do its work. A screen-oriented text editor is an example of an interactive program.

Interface Message Processor (IMP) The original name for a hardware device used in the construction of packet-switched networks such as the ARPANET. *See also* Packet Switch Node.

Internet Control Message Protocol (ICMP) A host-to-host communication protocol used in the DARPA Internet for reporting errors and controlling the operation of IP.

internet domain A communication domain in the interprocess-communication facilities that supports the DARPA Internet architecture. This architecture supports both stream and datagram-oriented styles of communication between processes on machines in an internet.

internet host address In the DARPA Internet, a 32-bit number that identifies both the network on which a host is located and the host on that network.

Internet Protocol (IP) The network-layer communication protocol used in the DARPA Internet. IP is responsible for host-to-host addressing and routing, packet forwarding, and packet fragmentation and reassembly.

Internetwork Datagram Protocol (IDP) In the Xerox Network Systems architecture, a communication protocol that serves as the level-1 protocol. IDP is responsible for the internet packet format, internet addressing, and routing.

interprocess communication (IPC) The transfer of data between processes. Most facilities for interprocess communication are designed such that data are transferred between objects other than processes. An interprocess-communication model that is not directly process-oriented is advantageous because it is possible to model scenarios in which communication endpoints are location-independent and, possibly, dynamically migrated. For example, in 4.3BSD, communication is between sockets, not processes.

interrupt In computer systems, an event external to the currently executing process that causes a change in the normal flow of instruction execution. Interrupts usually are generated by hardware devices that are external to the CPU.

interrupt priority level The priority that is associated with a device interrupt. This value is usually defined by switches or jumpers located on a device controller and transmitted with each interrupt request made by the hardware device. *See also* processor priority level.

interrupt stack A run-time stack that is used by procedures that are invoked to respond to interrupts and traps. On the VAX, a systemwide interrupt stack is provided that is independent of the normal kernel–run-time stack located in the user structure of each process.

interrupt vector A storage location known to the system that contains the starting address of a procedure to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting device controller and for classes of exceptions.

interrupt-vector table A table of interrupt vectors. On most machines that support vectored interrupts, the vector supplied by the interrupting device is used to index into an interrupt-vector table. The VAX uses several interrupt-vector tables; for example, each UNIBUS adapter has its own interrupt-vector table. *See also* vectored interrupt.

I/O *See* Input/Output.

I/O page A section of an address space in which I/O device control registers are accessible. On the VAX, the last 8 Kbyte of the address space of each UNIBUS adapter is reserved as an I/O page through which UNIBUS-device control registers are accessed. *See also* memory-mapped I/O architecture.

I/O redirection The redirection of an I/O stream from its default assignment. For example, all the standard shells permit users to redirect the standard output stream to a file or process.

I/O stream A stream of data directed to, or generated from, a process. Most I/O streams in UNIX have a single common data format that permits users to write programs in a tool-oriented fashion and to combine these programs in pipelines by directing the standard output stream of one program to the standard input stream of another.

I/O vector *See* iovec.

iovec A data structure used to specify user I/O requests made to the kernel. Each structure holds the address of a data buffer and the number of bytes of data to be read or written. Arrays of such structures are passed to the kernel in *readv* and *writev* system calls.

IP *See* Internet Protocol.

IPC *See* interprocess communication.

job In UNIX, a set of processes that all have the same process-group identifier. Jobs that have multiple processes are normally created with a pipeline. A job is the fundamental object that is manipulated with job control.

job control A facility for managing jobs. With job control, a job may be started, stopped, and killed, as well as moved between the foreground and the background. The terminal handler provides facilities for automatically stopping a background job that tries to access the controlling terminal, and for notifying a job's controlling process when such an event occurs.

keepalive packet A type of packet used by TCP to maintain information about whether or not a destination host is up. Keepalive packets are sent to a remote host, which, if it is up, must respond. If a response is not received in a reasonable period of time to any of several keepalive packets, then the

connection is terminated. Keepalive packets are used only on TCP connections that have been created for sockets that have the SO_KEEPALIVE option set on them.

keepalive timer A timer used by the TCP protocol in conjunction with keepalive packets. The timer is set when a keepalive packet is transmitted. If a response to the packet is not received before the timer expires several times, then the connection is shut down.

kernel The central controlling program that provides basic system facilities. The UNIX kernel creates and manages processes, provides functions to access the filesystem, and supplies communication facilities. The UNIX kernel is the only part of UNIX that a user cannot replace.

kernel map The resource map that is used by the 4.3BSD kernel in allocating the PTEs that are used to map the virtual address spaces of user processes.

kernel mode The most privileged processor-access mode. The UNIX kernel operates in kernel mode.

kernel process A process that executes with the processor in kernel mode. The *pagedaemon* and *swapper* processes are examples of kernel processes.

kernel state The run-time execution state for the kernel. This state, which includes the program counter, general-purpose registers, and run-time stack, must be saved and restored on each context switch.

kill character The ASCII character that is recognized by the terminal handler in cooked mode to mean ''delete everything typed on this terminal after the last end-of-line character.'' Each terminal can have a different kill character, and that kill character can be changed at any time with an *ioctl* system call. The terminal handler does not recognize the kill character on terminals that are in raw or cbreak mode. *See also* erase character and word-erase character.

kluster A group of pages related by their locality in a process's virtual address space. This grouping is performed by the virtual-memory system when servicing certain page faults for the purpose of prepaging. The name is spelled with a 'k' to distinguish it from the clustering of pages. *See also* fill-on-demand klustering.

ldpctx A VAX instruction that causes the current VAX process state to be loaded from the process-control block pointed to by the PCBB register. *See also* svpctx.

least recently used (LRU) A policy of reuse whereby the least recently used items are reused first. For example, in the filesystem, there are a fixed number of data buffers available for doing I/O. Buffers that hold valid data are reallocated in an LRU order in the hope that the data held in the buffer may be reused by a subsequent read request.

line discipline A processing module in the kernel that is used to provide semantics on an asynchronous serial interface or on a software emulation of such an interface. Line disciplines are described by a procedural interface whose entry points are stored in a *linesw* data structure.

line mode *See* cooked mode.

link layer Layer 2 in the ISO Open Systems Interconnection Reference Model. In this model, the link layer is responsible for the (possibly unreliable) delivery of messages within a single physical network. The link layer corresponds most closely to the network-interface layer of the 4.3BSD network subsystem.

listen request A request passed to the user-request routine of a communication-protocol module as a result of a process making a *listen* system call on a socket. This request indicates that the system should listen for requests to establish a connection to the socket. Otherwise, the system will reject any connection requests it receives for the socket.

load average A measure of the CPU load on the system. The load average in 4.3BSD is defined as an average of the number of processes ready to run or waiting for disk I/O to complete, as sampled over the previous 1-minute interval of system operation.

locality of reference A phenomenon whereby memory references of a running program are localized within the virtual address space over short periods of time. Most programs tend to exhibit some degree of locality of reference. This locality of reference makes it worthwhile for the system to prefetch pages that are adjacent to a page that is faulted, so as to reduce the fault rate of a running program.

logical block A block defined by dividing a file's linear extent by the underlying filesystem block size. Each logical block of a file is mapped into a physical block. This additional level of mapping permits physical blocks to be placed on disk without concern for the linear organization of the logical blocks in a file.

logical drive partitions A software scheme that is used to divide a disk drive into one or more linear extents or partitions.

logical unit An integer that specifies the unit number of a hardware device. The hardware device and unit number are specified in terms of logical devices and units as discovered by the system during the autoconfiguration phase of its bootstrap sequence. For example, a reference to partition 1 on disk drive 2 typically refers to partition 1 on the third disk drive identified at boot time (devices are numbered starting at 0). The actual mapping between logical unit numbers and physical devices is defined by the configuration file that is used to build a kernel. For flexibility, most systems are configured to support a reasonably dynamic mapping between physical and logical devices. This dynamic mapping permits, for example, system administrators to move a disk drive from one controller to another without having to reconfigure a new kernel or to reconstruct the associated special files for the device.

long-term–scheduling algorithm *See* short-term–scheduling algorithm.

loop In the virtual-memory system, the circular collection of pages that make up the part of main memory that is used by pageable processes. A global CLOCK algorithm is used by the virtual-memory system to manage pages in the loop.

lossy A communication medium that has a high rate of data loss is said to be ''lossy.''

low watermark A data-flow control that specifies the minimum number of data that must be present before an action can be taken. In the interprocess-communication facilities, each socket's data buffer has a low watermark that specifies the minimum number of data that must be present in the data buffer before a reception request will be satisfied.

LRU *See* least recently used.

machine check On the VAX, an exceptional machine condition that indicates that the CPU detected an internal error in itself. For example, a machine check is generated if a parity error is detected in a cache memory.

magic number The number located in the first few bytes of an executable file that specifies the type of executable file that file is. On the VAX, the magic number is present in the first four bytes of each executable file.

main memory The primary memory system on a machine.

major-device number An integer number that uniquely identifies the type of a device. This number is defined as the index into the array of device-driver entry points for the device and is used, for example, when a user creates a device special file with the *mknod* system call.

mapped An object is said to be mapped into the virtual address space of a process when page tables have been set up that map the pages of the object into the process's address space.

mark and sweep algorithm A garbage-collection algorithm that works by sweeping through the set of collectable objects, marking each object that is referenced. If, after this marking phase, there are any objects that are unmarked, they are reclaimed.

masked A signal is said to be ''masked'' if it has been specified in a *sigblock* or *sigsetmask* system call. When a signal is masked, its delivery is delayed until it is unmasked. In addition, in 4.3BSD, the system automatically masks a caught signal while that signal is being handled.

MASSBUS An I/O bus designed by DEC for higher-speed mass-storage peripherals than could be supported by the UNIBUS. The MASSBUS is found mainly on older models of PDP-11 and VAX computers.

MASSBUS adapter A hardware device that permits a MASSBUS to be attached to the system bus of a VAX.

MASSBUS-adapter support routines The set of routines in the kernel that are used in implementing device drivers for MASSBUS peripherals. These routines schedule the use of MASSBUS-adapter resources: the memory-mapping registers and the data paths.

MASSBUS unit number An integer number used by a device driver to identify a MASSBUS adapter. The MASSBUS unit number used by a device driver is a

logical number that is mapped into a physical unit number by a table that is constructed at the time the system is bootstrapped. This mapping is done to provide flexibility in the configuration of devices on a machine.

master device *See* slave device.

maximum segment lifetime (MSL) The maximum time a segment of data may exist in the network. *See also* 2MSL timer.

mbuf A data structure that describes a block of data; mbufs are used in the interprocess-communication facilities. "Mbuf" is shorthand for "memory buffer."

memory address A number that specifies a memory location. Memory addresses are often categorized as *physical* or *virtual* according to whether they reference *physical memory* or *virtual memory*.

memory free list *See* free list.

memory-management system The part of the operating system that is responsible for the management of memory resources available on a machine.

memory-management unit A hardware device that implements memory-management–related tasks such as address translation and memory protection. Most contemporary memory-management units also provide support for demand-paged virtual-memory management.

memory-mapped I/O architecture An architecture where the data structures associated with hardware devices are accessed as part of the physical address space of a machine. Such devices are said to be memory mapped. This style of architecture is contrasted with one in which special machine instructions are used to access hardware devices. *See also* I/O page.

message buffer A circular buffer in which the system records all kernel messages directed to the console terminal. The device **/dev/klog** can be used by a user program to read data from this buffer in a manner that ensures that no data will be lost. On the VAX, the message buffer is allocated early in the bootstrapping of the system, and it is placed in high memory so that it can be located after a reboot, allowing messages printed out just before a crash to be saved.

minor-device number An integer number that uniquely identifies a subunit of a device. For example, the minor-device number for a disk device specifies a subunit termed a *partition*, whereas the minor-device number for a terminal multiplexer identifies a specific terminal line. The minor-device number is interpreted on a per-device basis and is used, for example, when a user creates a device special file with the *mknod* system call.

modem control For data-communication equipment, the support of a set of signals used to ensure reliable initiation and termination of connections over asynchronous serial lines governed by the RS-232 standard. Support for modem control is normally important only for serial lines that are accessed via dialup modems.

MSL *See* maximum segment lifetime.

multilevel feedback queue A queueing scheme in which requests are partitioned into multiple prioritized subqueues, with requests moving between subqueues based on dynamically varying criterion. The 4.3BSD kernel uses a multilevel feedback queueing scheme for scheduling the execution of processes.

multiplexed file A type of file used for interprocess communication that was supported in the Seventh Edition UNIX system.

network address A number that specifies a host machine.

network architecture The collection of protocols, facilities, and conventions (such as the format of a network address) that define a network. Like machine architectures, network architectures may be realized in different ways. For example, some network architectures are specifically designed to permit their implementation in hardware devices.

network byte order The order defined by a network for the transmission of protocol fields that are larger than one octet. In the DARPA Internet protocols, this order is "most significant octet first."

network-interface layer The layer of software in the 4.3BSD network subsystem that is responsible for transporting messages between hosts connected to a common transmission medium. This layer is mainly concerned with driving the transmission media involved and performing any necessary link-level protocol encapsulation and decapsulation.

network layer The layer of software in the 4.3BSD network subsystem that is responsible for the delivery of data destined for remote transport or network-layer protocols.

network mask A value that is used in the subnet addressing scheme of the DARPA Internet. A network mask specifies which bits in a local Internet address the system should include when extracting a network identifier from a local address.

network virtual terminal A terminal device that receives and transmits data across a network connection.

new terminal driver The terminal line discipline that extends the old terminal driver to support job control and logical line-editing facilities.

nexus On the VAX, a physical connection to the main system bus that supports the transfer of data and control information (such as interrupt vectors).

nice In UNIX, a user-controllable process scheduling parameter. The value of a process's *nice* variable is used in calculating that process's scheduling priority. Positive values of *nice* mean that the process is willing to receive less than its share of the processor. Negative values of *nice* mean that the process should be given more than its share of the processor.

nonblocking I/O A mode in which a descriptor may be placed, whereby the system will return an error if any I/O operation on the descriptor would cause the

process to block. For example, if a *read* system call is performed on a descriptor that is in nonblocking I/O mode, and no data are available, the system will return the error code EWOULDBLOCK rather than block the process until data arrive. *See also* signal-driven I/O.

nonlocal goto A transfer in control that circumvents the normal flow of execution in a program across routine boundaries. For example, if procedure A calls procedure B, and B calls C, then a direct transfer of control from C back to A (bypassing B) would be a nonlocal goto.

nonresident An object that is not present in main memory is said to be "nonresident." For example, a page in the virtual address space of a process may be nonresident if the page has never been referenced.

octet A basic unit of data representation; an eight-bit byte. The term *octet* is used instead of *byte* in the definition of many network protocols because some machines use other byte sizes.

old terminal driver The terminal line discipline present in UNIX 32V and the Seventh Edition UNIX. This interface supports only a very simple line-editing facility based on erase and kill characters. More advanced line-editing facilities, such as the word erase and literal characters, as well as support for job control, are present in the new terminal driver.

Optimal Replacement Policy A replacement policy that optimizes the performance of a demand-paging virtual-memory system. In this book, the Optimal Replacement Policy refers to the policy in which advanced knowledge of the full reference string of a program is known, and pages are selected such that the number of page faults is minimized.

out-of-band data Data transmitted and received out of the normal flow of data. Stream sockets support a logically separate out-of-band data channel through which at least one message of at least one byte of data may be sent. The system immediately notifies a receiving process of the presence of out-of-band data, and out-of-band data may be retrieved out of the received order of data.

overlay In computer systems, a region of code or data that may be replaced with other such regions on demand. Overlays are usually loaded into a process's address space on demand, possibly on top of another overlay. Overlays are a commonly used scheme for programs that are too large to fit in the address space of a machine that does not support virtual memory.

P0 base register (P0BR) On the VAX, the processor register, or its saved value in a process-control block, that contains the virtual address of the page table for a process's P0 region. (The P0 region maps a process's text, data, and heap.)

P0 length register (P0LR) On the VAX, the processor register, or its saved value in a process-control block, that contains the number of entries in the page table for a process's P0 region.

P0BR *See* P0 base register.

P0LR *See* P0 length register.

P1 base register (P1BR) On the VAX, the processor register, or its saved value in a process-control block, that contains the virtual address of a process's P1 region page table. (The P1 region maps a process's stack and user area.)

P1 length register (P1LR) On the VAX, the processor register, or its saved value in a process-control block, that contains the number of nonexistent page-table entries for pages in a process's P1 region.

P1BR *See* P1 base register.

P1LR *See* P1 length register.

Packet Exchange (PEX) The standard datagram service in the Xerox Network Systems architecture. This facility is implemented in 4.3BSD by a user-level library using IDP datagram sockets.

Packet Switch Node (PSN) A hardware device used in the construction of packet-switched networks such as the ARPANET. Each PSN utilizes a protocol that provides reliable, flow-controlled, packet-switching functionality. *See also* Interface Message Processor.

page In memory-management, the fixed-sized unit of measure used to divide a physical or virtual address space. *See also* demand paging.

page fault An exception generated by a process reference to a page of its virtual address space that is not marked as resident in memory.

page push A pageout of a dirty page.

page reclaim A page fault, where the page that was faulted is located in memory, usually on the memory free list.

page-table entry (PTE) The data structure that identifies the location and status of a page of a virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the PTE contains the information needed to create the page or to locate the page on secondary storage.

pagedaemon In 4.3BSD, the name of the process that is responsible for writing parts of the address space of a process to secondary storage to support the paging facilities of the virtual-memory system. *See also* swapper.

pagein An operation performed by the virtual-memory system in which the contents of a page are read from secondary storage.

pageout An operation performed by the virtual-memory system in which the contents of a page are written to secondary storage.

pageout in progress A pageout operation that has been started, but has not yet been completed. In 4.3BSD, the system tracks the number of pageouts for each shared text segment, to ensure there are no other references when it reclaims the resources associated with such a segment.

paging The actions of bringing pages of an executing process into main memory when they are referenced, and removing them from memory when they are replaced. When a process executes, all its pages are said to reside in virtual

memory. Only the actively used pages, however, need to reside in main memory. The remaining pages can reside on disk until they are needed.

panic In UNIX, an unrecoverable system failure detected by the kernel. 4.3BSD automatically recovers from a panic by rebooting the machine, repairing any filesystem damage, and then restarting normal operation. *See also* crash dump.

parent process A process that is a direct relative of another process as a result of a *fork* system call.

partition *See* disk partition.

pathname A null-terminated character string starting with an optional slash (''/''), followed by zero or more directory names separated by slashes, and optionally followed by a filename. If a pathname begins with a slash, the pathname is said to be an *absolute pathname* and the path search begins at the root directory. Otherwise, the pathname is said to be a *relative pathname* and the path search begins at the current working directory of the process. A slash by itself names the root directory. A null pathname refers to the current working directory.

PCB *See* process control block.

PCBB *See* Process Control Block Base.

persist timer A timer used by TCP for maintaining output flow on a connection. This timer is started whenever data are ready to be sent, but the send window is too small to bother sending and no data are already outstanding. If no window update is received before the timer expires, a window probe is sent.

PEX *See* Packet Exchange.

physical block One or more contiguous disk sectors to which the system maps a logical block.

physical unit number An integer that is recognized by a device controller as specifying a particular hardware unit. For example, a tape transport has a physical unit number that a user must use in referring to the device when making requests to an associated tape controller. The physical unit number is often different from the logical unit number used by the system.

PID *See* process identifier.

pipe An interprocess-communication facility that supports the unidirectional flow of data between related processes. Data transfer is stream-oriented, reliable, and flow controlled. A pipe is specified to the shell with the ''|'' symbol. For example, to connect the standard output of a program named **a** to the standard input of a program named **b**, the command ''a | b'' would be used.

pipeline A collection of processes in which the standard output of one process is connected to the standard input of the next with a pipe.

placement policy The policy used by the virtual-memory system to place pages in main memory when servicing a page fault.

POSIX The portable operating-system interface standards group, P1003, established by the IEEE. Their first established standard was the kernel interface,

1003.1, which was ratified in 1988.

post a signal To notify a process that a signal is pending for it. Since most of the actions associated with a signal are performed by the receiving process, posting a signal usually does little more than to record the pending signal in the process's proc structure and to arrange for the process to be run.

prefetching The retrieval of data before they are needed. Many machines prefetch machine instructions so that they can overlap the time spent fetching instructions from memory with the time spent decoding instructions.

prepaging The prefetching of pages of memory. Prepaging is a technique used by virtual-memory systems to reduce the number of page faults.

probe routine A device-driver routine is responsible for deciding whether or not a hardware device is present on a machine and for attempting to make the device interrupt.

probing The operation of checking to see whether a hardware device is present on a machine. Each different type of hardware device usually requires its own technique for probing.

/proc filesystem A filesystem-based interface to active processes that can be used to construct process-debugging facilities. Each process is represented by a directory entry in a pseudo-directory named **/proc**. Applications access the virtual address space of a process by opening the file in **/proc** that is associated with the process, and then using the *read* and *write* system calls as though the process were a regular file. This facility is part of the Eighth Edition UNIX system.

proc structure A data structure maintained by the kernel for each active process in the system. The proc structure for a process is always resident, as opposed to the user structure, which is moved to secondary storage when the process is swapped out.

process In operating systems, a task or thread of execution. In UNIX, user processes are created with the *fork* system call.

process control block (PCB) A data structure used to hold process context. On the VAX, the hardware PCB contains the hardware portion of this context. The software PCB contains the software portion, and is located in memory immediately after the hardware PCB.

Process Control Block Base (PCBB) On the VAX, a register that holds the physical address of the process-control block for the currently running process.

process group A collection of processes on a single machine that all have the same process-group identifier. This grouping is used to arbitrate between multiple jobs contending for the same terminal.

process-group identifier A positive integer used to identify uniquely each active process group in the system. Process-group identifiers are typically defined to be the PID of the process-group leader. Process-group identifiers are used by command interpreters in implementing job control, when broadcasting signals with the *killpg* system call, and when altering the scheduling priority of all processes in a process group with the *setpriority* system call.

process-group leader The process in a process group whose PID is used as the process-group identifier. This process is typically the first process in a pipeline.

process identifier (PID) A nonnegative integer used to identify uniquely each active process in the system.

process open file table *See* descriptor table.

process priority A parameter used by the kernel to schedule the execution of processes. The priority for a process changes dynamically according to the operation of the process. In addition, the *nice* parameter can be set for a process to weight the overall scheduling priority for the process.

process table The collection of proc structures that describe all the active processes that exist in the system. The process table is always resident in main memory.

processor priority level A priority that is used to control the delivery of interrupts to the CPU. Most machines support multiple priority levels at which the processor may execute. Similarly, interrupts also occur at multiple levels. When an interrupt is posted to the processor, if the priority level of the interrupt is greater than that of the processor, then the interrupt is recognized by the processor and execution is diverted to service the interrupt. Otherwise, the interrupt is not acknowledged by the CPU, and it is held pending until the processor priority drops to a level that permits the interrupt to be acknowledged. Changing the processor priority level is usually a privileged operation that can be done only when the processor is executing in kernel mode.

Processor Status Longword (PSL) On the VAX, a processor register that consists of a word of privileged processor status and the *Processor Status Word* (PSW). The privileged processor status information includes the current processor priority level, the current and previous access modes, and the interrupt stack bit. The PSW contains unprivileged processor status information, such as the condition codes, the arithmetic trap-enable flags, and the trace-trap–enable bit.

protocol family A collection of communication protocols that are related by being part of a single network architecture. For example the TCP, UDP, IP, and ICMP protocols are part of the protocol family for the DARPA Internet.

protocol switch structure A data structure that holds all the entry points for a communication protocol supported by the kernel.

pseudo-DMA A technique in which DMA is simulated in software. This technique is usually used by device drivers that need to receive data from or send data to a device that has real-time constraints that cannot be met under normal operating circumstances.

PSL *See* Processor Status Longword.

PSN *See* Packet Switch Node.

PTE *See* page-table entry.

pure demand paging Demand paging without prepaging.

raw-device interface The character-device interface for block-oriented devices such as disks and tapes. This interface provides raw access to the underlying device, arranging for direct I/O between a process and the device.

raw mode A mode of operation for a terminal device whereby processes reading from the terminal receive input immediately as it is typed and without an interpretation of the input characters. *See also* cooked mode and cbreak mode.

raw socket A socket that provides direct access to a lower-level communication protocol.

Ready for Next Message (RFNM) A type of message in the protocol used between a host and a PSN. This type of message is sent from a PSN to the local host to which it is connected when the PSN receives an acknowledgment for an outstanding message. The message from the PSN to the local host indicates that the PSN is ready to accept a new message for transmission to the remote host. RFNM messages are part of the host-to-host flow-control facility provided by PSNs.

real GID *See* real group identifier.

real group identifier (real GID) The GID that is recorded in the accounting record when a process terminates. The real GID for a process is initially set at the time a user logs into a system, and is then inherited by child processes across subsequent *fork* and *execve* system calls (irrespective of whether or not a program is set-group-identifier). The real GID can be made the effective GID with the *setregid* system call. *See also* effective group identifier and real group identifier.

real UID *See* real user identifier.

real user identifier (real UID) With respect to a process, the true identity of the user that owns the process. The real UID for a process is initially set at the time a user logs into a system, and is then inherited by child processes across subsequent *fork* and *execve* system calls (irrespective of whether or not a program is set-user-identifier). The real UID is recorded in the accounting record when a process terminates. The real UID can be made the effective user identifier with the *setreuid* system call. *See also* effective user identifier and real user identifier.

receive window In TCP, the range of sequence numbers that defines the data that the system will accept for a connection. Any data with sequence numbers outside this range that are received are dropped. *See also* sliding-window scheme.

reclaim *See* page reclaim.

reclaim from free A page-reclaim operation from the memory free list. This scenario can take place if a page is reclaimed by the page-replacement algorithm, but the page is not reassigned before a process faults on it.

red zone A read-only region of memory immediately below the last page of the per-process run-time stack (on the VAX). The red zone is set up by the system so that a fault will occur if a process overruns its kernel-mode run-time stack.

reference bit A bit in a PTE that indicates whether a process has accessed the associated page. The VAX architecture does not support a reference bit, so the bit is simulated in software.

reference string A dataset that describes the pages referenced by a process over the time of the process's execution. This description represents the memory-related behavior of the process at discrete times during its lifetime.

referenced In the virtual-memory system, a page that is read from or written to is considered to have been referenced.

register save mask On the VAX, a word that describes the registers to be saved on entry to a procedure that is called using a VAX call instruction.

relative pathname *See* pathname.

reliably delivered message socket A type of socket that guarantees reliable data delivery and preservation of message boundaries, and that is not connection-based.

relocation The copying of a program's contents from one place in an address space to another. This copying may be accompanied by modifications to the image of the program, so that memory references encoded in the program remain correct after that program is copied. Code that is not bound to a particular starting memory address is said to be *relocatable*.

replacement policy The policy that a demand-paged virtual-memory–management system uses to select pages for reuse when memory is otherwise unavailable.

resident An object that is present in main memory is said to be "resident." For example, a page in the virtual address space of a process is resident if its contents is present in main memory.

resource map A data structure used by the system to manage the allocation of a resource that can be described by a set of linear extents.

retransmit timer A timer used by TCP to trigger the retransmission of data. This timer is set each time data are transmitted to a remote host. It is set to a value that is expected to be greater than the time it will take the receiving host to receive the data and return an acknowledgement.

RFNM *See* Ready for Next Message.

root directory The directory used in resolving absolute pathnames. Each process has a root directory that can be set with the *chroot* system call, and the system has a unique root directory, the identity of which is set at the time the system is bootstrapped.

root filesystem The filesystem containing the root directory that is considered the root of all filesystems on a machine. The identity of a default root filesystem is compiled into a kernel, although the actual root filesystem used by a system may be set to some other filesystem at the time a system is bootstrapped.

rotational-layout table A per-filesystem data structure that describes the rotational position of blocks in a filesystem. The filesystem uses the rotational-

layout table for a filesystem in selecting rotationally optimal blocks for allocation to a file.

round robin In queueing, an algorithm in which each requester is serviced for a fixed period of time in a first-come-first-served order; requests are placed at the end of the queue if they are incomplete after service.

route In packet-switched–network communication, a route to a destination specifies the host or hosts through which data must be transmitted to reach the destination.

routing daemon The process in 4.3BSD that provides a routing-management service for the system. This service uses a protocol that implements a distributed database of routing information that is dynamically updated to reflect changes in topological connectivity.

routing redirect message A message generated by a gateway when the latter recognizes that a message that it has received can be delivered with a more direct route.

run queue The queue of processes that are ready to execute.

SBR *See* system base register.

scatter loading A memory-management technique used in UNIX 32V. With scatter loading, when a process is loaded into memory, the contents of its address space are loaded into (possibly) discontiguous regions of memory. Scatter loading tends to be more space-efficient than is loading into a single contiguous region, but can result in memory fragmentation.

scheduling In operating systems, the planning used to share a resource. For example, process scheduling is used to share the CPU and main memory.

scheduling priority A per-process parameter maintained by the kernel that specifies the priority with which the latter will schedule the execution of a process. When a process is executing in user mode, the system periodically calculates the scheduling priority, using the process priority and the *nice* parameter.

secondary storage Storage that is used to hold data that do not fit in main memory. Secondary storage is usually located on rotating magnetic media, such as disk drives. *See also* backing storage.

sector The smallest contiguous region on a disk that may be accessed with a single I/O operation.

segment A contiguous range of data defined by a base and an extent. In memory management, a segment is defined by a contiguous range of memory addresses, and is used to describe a portion of a process's address space or a portion of main memory. In communication protocols, a segment is defined by a contiguous range of sequence numbers for which there are associated data.

send window In TCP, the range of sequence numbers that defines the data that the system may transmit on a connection and be assured that the receiving

party has space to hold them on receipt. Any data with sequence numbers prior to the start of the send window have already been sent and acknowledged. Any data with sequence numbers after the end of the window will not be sent until the send window changes to include them. *See also* sliding-window scheme.

sense request A request passed to the user-request routine of a communication-protocol module as a result of a process making a *stat* system call on a socket.

sequence space The range of sequence numbers that are assigned to data transmitted over a TCP connection. In TCP, sequence numbers are taken from a 32-bit circular space that starts with an arbitrary value called the *initial sequence number*.

Sequenced Packet Protocol (SPP) In the Xerox Network Systems architecture, a communication protocol that serves as a level-2 protocol. SPP is layered on top of IDP and provides reliable, unduplicated, and flow-controlled transmission of messages.

sequenced packet socket A type of socket that models sequenced, reliable, unduplicated, connection-based communication that preserves message boundaries.

Serial Line IP (SLIP) An encapsulation used to transfer IP datagrams over asynchronous serial lines. Also, the line discipline that implements this encapsulation.

server process A process that provides services to client processes via an interprocess-communication facility.

session In job control, all the processes started during a single login session.

set-group-identifier program A program that runs with an additional group privilege. Set-group-identifier programs are indicated by a bit in the inode of the file. When a process specifies such a file in an *execve* system call, the GID of the file is made the effective GID of the process.

set-priority-level A request that sets the current *processor priority level*. In 4.3BSD on the VAX, all such requests are made with calls to routines that have a name with the prefix ''spl''. For example, to set the processor priority level high enough to block interrupts that cause terminal processing, the *spltty*() routine would be called.

set-user-identifier program A program that runs with an UID different from that of the process that started it running. Set-user-identifier programs are indicated by a bit in the inode of the file. When a process specifies such a file in an *execve* system call, the UID of the file is made the effective UID of the process.

shadow swap map A data structure in the user structure of each process that is used temporarily to record new swap space allocated to a process while that process is holding existing space.

shared text segment A text segment that is shared by more than one process. Shared text segments can be created only from certain types of programs.

Such programs are identified by their magic number. These programs are created with a text segment that is rounded to a size that makes sharing possible. *See also* text segment.

shell A program that interprets and executes user commands. When a user logs into a UNIX system, a shell process is normally created with its standard input, standard output, and standard error descriptors directed to the terminal or network virtual terminal on which the user logged in.

short-term–scheduling algorithm The algorithm used by the system to select the next process to run from among the set of processes that are deemed runnable. The *long-term–scheduling algorithm*, on the other hand, can influence the set of runnable processes by swapping processes in and out of main memory (and thus in and out of the set of runnable processes).

signal In UNIX, a software event; in 4.3BSD, this event is modeled after a hardware interrupt.

signal-driven I/O A mode in which a descriptor may be placed, whereby the system will deliver a SIGIO signal to a process whenever I/O is possible on the descriptor. *See also* nonblocking I/O.

signal handler A procedure that is invoked in response to a signal.

signal-trampoline code The piece of code located in the user structure of each process that is used to invoke a signal handler. The signal-trampoline code contains instructions that set up parameters for calling a signal handler, do the actual call to the signal handler, and, on return, do a *sigreturn* system call to reset kernel state and resume execution of the process after the signal is handled.

silly-window syndrome A condition observed in window-based flow-control schemes in which a receiver sends several small (i.e., silly) window allocations rather than waiting for a reasonable-sized window to become available.

single indirect block *See* indirect block.

slave device A hardware device that is controlled by a *master device*. For example, a disk drive is a slave device to a disk controller. The distinction between master and slave devices is used by the autoconfiguration system. A slave device is assumed to be accessible only if its corresponding master device is present.

slave routine A device-driver routine that is responsible for deciding whether or not a slave device is present on a machine. Slave routines are never called unless the master device for the slave has been probed successfully.

sleep queue The queue of processes that are blocked awaiting an event. The name is derived from the *sleep*() routine that places processes on this queue.

sliding-window scheme A flow-control scheme in which the receiver limits the number of data it is willing to receive. This limit is expressed as a contiguous range of sequence numbers termed the *receive window* and is periodically communicated to the sender, who is expected to transmit only data that are

within the window. As data are received and acknowledged, the window *slides* forward in the sequence space. *See also* sequence space; receive window; send window.

SLIP *See* Serial Line IP.

slip sector A sector that is used to replace a bad sector on the same track.

slip-sector forwarding The procedure of retargeting a bad sector to a *slip sector*.

SLR *See* system length register.

small-packet avoidance Avoiding the transmission of packets that are so small that they are inefficient.

socket In the 4.3BSD interprocess-communication model, an endpoint of communication. Also, the data structure that is used to implement the socket abstraction, and the system call that is used to create a socket.

soft limit A limit that may be temporarily exceeded, or exceeded a limited number of times. A soft limit is typically used in conjunction with a hard limit.

soft link *See* symbolic link.

software interrupt A software-initiated interrupt. The VAX supports 15 prioritized software interrupts. The kernel can request these interrupts by setting bits in the Software Interrupt Request Register (SIRR).

software-interrupt process A process that is set running in response to a software interrupt. In 4.3BSD, input processing for each transport-layer communication protocol is embodied in a software-interrupt process.

special file *See* device special file.

spin loop A sequence of instructions that causes the processor to do a specific operation repeatedly. Standalone device drivers use spin loops to implement real-time delays.

SPP *See* Sequenced Packet Protocol.

stack An area of memory set aside for temporary storage, or for procedure and interrupt-service linkages. A stack uses the last-in first-out (LIFO) concept. On the VAX, the stack grows from high memory addresses to low memory addresses. As items are added to (''pushed onto'') the stack, the stack pointer decrements; as items are retrieved from (''popped off'') the stack, the stack pointer increments.

stack segment A segment that holds a stack.

stale translation A translation or mapping that was true at some time, but that is no longer valid. For example, on machines that have a translation lookaside buffer, if a PTE in memory is changed to alter the mapping, any address translation for that page that is present in the TLB must be flushed to avoid a stale translation.

standalone Software that can run without the support of an operating system is said to be capable of ''standalone'' operation.

standalone device driver A device driver that is used in a standalone program. A standalone device driver usually differs from a device driver used in an operating system in that it does not have interrupt services, memory management, or full support for virtual-memory mapping. In the UNIX standalone I/O library, for example, a standalone device driver polls a device to decide when an operation has completed, and is responsible for setting up its own memory mapping when doing DMA.

standalone I/O library A library of software that is used in writing standalone programs. This library includes standalone device drivers that are used to do I/O.

standard error The I/O stream on which error messages are conventionally placed. This stream is usually associated with descriptor 2 in a process.

standard input The I/O stream on which input is conventionally received. This stream is usually associated with descriptor 0 in a process.

standard output The I/O stream to which output is conventionally directed. This stream is usually associated with descriptor 1 in a process.

start routine A device-driver routine that is responsible for starting a device operation after the system has acquired all the resources that are required for the operation.

sticky bit The bit in an inode that indicates that the text segment of the program is to be treated as a *sticky text*.

sticky text A text segment that is saved in the swap area for future reuse, rather than its being reloaded from the file from which it is was originally obtained. *See also* sticky bit.

stream I/O system A facility in the Eighth Edition UNIX system that permits the flexible configuration of processing for streams of data. In this system, it is possible to connect kernel-resident modules dynamically in a stack-oriented fashion, and to have these modules process data sent and received on an I/O stream.

stream socket A type of socket that models a reliable, connection-based, byte stream that may support out-of-band data transmission.

subnetwork A physical network that is a part of a larger logical network with a single shared network address. The subnet is assigned a subset of the logical network's address space.

superblock A data structure in the filesystem that gives the basic parameters of the filesystem.

superuser The user whose UID is 0. Processes owned by the superuser are granted special privileges by UNIX. The superuser's login name is usually *root*.

svpctx A VAX instruction that causes the current VAX process state to be loaded into the process-control block pointed to by the PCBB register. *See also* ldpctx.

swap area A region on secondary storage that is used for swapping and paging.

swap device A device on which a swap area resides.

swap map The data structure that describes available space in the swap area.

swap space *See* swap area.

swapper In 4.3BSD, the name of the process that implements the swapping portion of the memory-management facilities. Historically, the swapper is process 0. *See also* pagedaemon.

swapping A memory-management policy in which entire processes are moved to and from secondary storage when main memory is in short supply.

symbolic link A file whose *contents* are interpreted as a pathname when it is supplied as a component of a pathname. Also called a *soft link*.

synchronous Synchronized with the current process. For example, in UNIX all I/O operations appear to be synchronous: the *read* and *write* system calls do not return until the operation has been completed. (For a *write*, however, the data may not actually be written to their final destination until some time later—for example, in writing to a disk file.)

system activity An entry into the kernel. System activities can be categorized according to the event or action that initiates them: system calls, hardware interrupts, hardware traps, and software-initiated traps or interrupts.

system base register (SBR) On the VAX, the processor register that contains the physical address of the base of the system page table.

system bus The central bus in a machine. On many machines, the system bus is used to connect the CPU to main memory and to peripheral devices. On some machines, such as the VAX, many peripheral devices are not directly connected to the system bus, but instead are connected to an *I/O bus*, which is connected to the system bus through a bus adapter; see, for example, *UNIBUS* and *MASSBUS*.

system call In operating systems, a request to the system for service; also called a *system service request*.

system clock The device that is used to maintain the system's notion of time-of-day. On most systems, this device is an interval timer that periodically interrupts the CPU. The system uses these interrupts to maintain the current time-of-day, as well as to perform periodic functions such as process scheduling.

system identification register On the VAX, a processor register that contains the processor type and serial number.

system length register (SLR) On the VAX, the processor register that contains the number of PTEs in the system page table.

system mode *See* kernel mode.

system page table The page table used to map the virtual address space of the kernel. *See also* system base register and system length register.

system region On the VAX, the third quarter of the virtual address space of a process. Virtual addresses in the system region are shareable between processes.

TCP *See* Transmission Control Protocol.

terminal In computer systems, a device used to enter and receive data interactively from a computer. Most terminals include a CRT, which is used display data that is received from a computer. In the Electrical Industry Association (EIA) standard RS-232-C for connecting computers and Data Terminal Equipment (DTE), a terminal is a device that is placed at the other end of a wire that is connected to Data Communications Equipment (DCE). In this standard, a terminal might be any kind of device, not just the kind of device on which people type.

terminal multiplexer A hardware device that connects multiple serial lines to a computer. These serial lines may be used to connect terminals, modems, printers, and similar devices.

text segment The segment of a program that holds machine instructions. The system usually makes a program's text segment read-only and shareable by multiple processes when the program image is loaded into memory. *See also* shared text segment.

text structure A data structure used by the kernel to manage shared text segments.

thrashing A condition where requested memory utilization far exceeds the availability of memory. When a machine is thrashing, it usually spends more time performing system-related tasks than executing application code in user mode.

tick An interrupt by the system clock.

time quantum In a timesharing environment, the period of time that the process scheduler gives a process to run before it preempts that process so that another process can execute. Also called a *time slice*.

time slice *See* time quantum.

timer backoff The rate at which a timer value is increased. For example, in TCP, the value of the retransmit timer is determined by a table of multipliers that provide a near-exponential increase in timeout values.

TLB *See* translation lookaside buffer.

top half With regard to system operation, the collection of routines in the kernel that are invoked synchronously as a result of a system call or trap. These routines, the top half of the kernel, depend on per-process state and can block by calling *sleep*(). *See also* bottom half.

trace trap On the VAX, a trap caused by setting the *trace* bit in the PSL. This trap is used by the system to implement single-stepping in program debuggers. The kernel sets the trace bit in the PSL of the process being debugged so that a trace trap will be generated after the process executes one instruction. This trap is fielded by the kernel, which stops the process and returns control to the debugging process.

track In computer systems, the sectors of a disk that are accessible by one head at one of its seek positions.

trailer protocol A protocol in which control information is placed *after* the data. 4.3BSD uses a trailer protocol for transmitting packets on an Ethernet when a packet contains at least 512 bytes of data and the data size of the packet is a multiple of 512 bytes (and when the receiving host is willing to accept the protocol). This protocol is designed to permit the receiving host to receive data in page-aligned areas of memory and, thereby, to avoid memory-to-memory copy operations.

translation lookaside buffer (TLB) A processor cache containing translations for recently used virtual addresses.

translation-not-valid fault On the VAX, a fault generated when a process references a page in memory whose PTE is not marked valid—that is, the valid bit in the PTE is zero.

Transmission Control Protocol (TCP) A connection-oriented transport protocol used in the DARPA Internet. TCP provides for the reliable transfer of data, as well as the out-of-band indication of urgent data.

transport layer The layer of software in the network subsystem that provides the addressing structure required for communication between sockets, as well as any protocol mechanisms necessary for socket semantics such as reliable data delivery.

triple indirect block *See* indirect block.

tty driver The software module that implements the semantics associated with a terminal device. In 4.3BSD, the terminal-handling software is implemented in several line disciplines.

2MSL timer A timer used by the TCP protocol during connection shutdown. The name refers to the fact that the timer is set for twice the maximum time a segment may exist in the network. This value is chosen to ensure that future shutdown actions on the connection are done only after all segments associated with the connection no longer exist.

type-ahead The ability to transmit data to a system, usually by a user typing at a keyboard, before they are requested by a process.

u-dot *See* user structure.

UDP *See* User Datagram Protocol.

UID *See* user identifier.

uio A data structure used by the system to describe an I/O operation. This structure contains an array of *iovec* structures, the file offset at which the operation should start, the sum of the lengths of the I/O vectors, a flag showing whether the operation is a read or a write, and a flag showing whether the source and destination are both in the kernel's address space, or whether the source and destination are split between user and kernel address spaces.

unbuffered data path A data path in a UNIBUS adapter that has no buffering. Transfers through a UNIBUS adapter that use an unbuffered data path

generate one or more system-bus data transfers for each UNIBUS transfer. *See also* buffered data path.

UNIBUS An I/O bus designed by DEC for medium- and low-speed peripherals.

UNIBUS adapter A hardware device that permits a UNIBUS to be attached to the system bus of a VAX.

unit start routine A device-driver routine that is responsible for initiating an operation on a unit. This routine typically does any preparatory work necessary for the operation, and then asks the system to allocate resources for the operation and schedule a call to the device driver's *go routine* that will tell the hardware device to do the operation.

UNIX domain A communication domain in the interprocess-communication facilities that supports stream and datagram-oriented styles of communication between processes on a single machine.

urgent data In TCP, data that are marked for urgent delivery.

user area *See* user structure.

User Datagram Protocol (UDP) A simple, unreliable, datagram protocol used in the DARPA Internet. UDP provides only peer-to-peer addressing and optional data checksums.

user identifier (UID) A 16-bit nonnegative integer that uniquely identifies a user. UIDs are used in the access-control facilities provided by the filesystem. *See also* effective user identifier; real user identifier; set-user-identifier program.

user mode On the VAX, the least privileged processor access mode. User processes run in user mode.

user page table The page table used to map the virtual address space of a process. On the VAX, user page tables reside in the virtual address space of the kernel. Therefore, the translation of virtual addresses by a process executing in *user mode* requires two page tables: the user page table for the process and the kernel or system page table.

user-request routine A routine provided by each communication protocol that directly supports a socket (a protocol that indirectly supports a socket is layered underneath a protocol that directly supports a socket). This routine serves as the main interface between the layer of software that implements sockets and the communication protocol. The interprocess-communication facilities make calls to the user-request routine for most socket-related system calls. *See also* connect request; control request; listen request; sense request.

user structure A data structure maintained by the kernel for each active process in the system. The user structure contains, among many things, the UID and GIDs of the process, the table of open descriptors, and the kernel mode run-time stack. Unlike the proc structure, the user structure for a process is moved to secondary storage if the process is swapped out. Also referred to as the *u-dot area* and *user area*.

VAX architecture The functionality found in all VAX machines, no matter what implementation technique is used. This functionality is usually defined by the attributes visible to the machine-level programmer and can be characterized by the VAX instruction set.

VAX system A particular implementation of the VAX architecture. The implementation includes the actual hardware structure used in realizing the architecture; for example, the algorithms used in controlling the machine as it interprets the architecture. The implementation also encompasses all the details associated with the physical aspects of a machine; for example, the kind of logic and how that logic is packaged and interconnected.

vectored interrupt A type of interrupt. When a device generates a vectored interrupt, it transmits a value to the CPU that is used to select an interrupt handler from an *interrupt vector table*.

virtual address An address that references a location in a *virtual address space*.

virtual address space A contiguous range of virtual-memory locations.

virtual machine A machine whose architecture is emulated in software.

virtual memory A facility whereby the effective range of addressable memory locations provided to a process is independent of the size of main memory; that is, the virtual address space of a process is independent of the physical address space of the CPU.

wait The UNIX system call that is used to wait for the termination of a descendent process.

wait channel A value used to identify an event for which a process is waiting. In most situations, a wait channel is defined as the address of a data structure related to the event for which a process is waiting. For example, if a process is waiting for the completion of a disk read, the wait channel is specified as the address of the data structure supplied to the block I/O system.

wildcard route A route that is used if there is no explicit route to a destination.

window probe In TCP, a message that is transmitted when data are queued for transmission, the send window is too small for TCP to bother sending data, and no message containing an update for the send window has been received in a long time. A window probe message contains a single octet of data.

word-erase character The ASCII character that is recognized by the terminal handler in cooked mode to mean "delete the last word typed on this terminal," where a word is defined as a consecutive string of characters that does not include a space or horizontal tab. Each terminal can have a different word-erase character and that character can be changed at any time with an *ioctl* system call. The terminal handler does not recognize the word-erase character on terminals that are in raw or cbreak mode. *See also* erase character and kill character.

working directory *See* current working directory.

working set The set of pages in a process's virtual address space to which memory references have been made over some period of time. Most processes exhibit some locality of reference and the size of their working set is typically less than one-half of their total virtual memory size.

XNS domain A communication domain in the interprocess-communication facilities that supports the Xerox Network System (XNS) architecture. This architecture supports both stream and datagram-oriented styles of communication between processes on machines in an internet.

zombie process A process that has terminated but whose exit status has not yet been received by its parent process (or by *init*).

Index

#!, 56

A

absolute pathname, 35, 413, 435
Accent operating system, 20, 281, 300
accept system call, 286, 296, 298–299, 308, 367, 385
access control, filesystem, 36
access rights, 39, 283, 293, 302, 413
  passing, 304–306
  receiving, 304
accounting, process resource, 51, 62–63, 93
address family, 311, 413
address format, 311, 413
Address Resolution Protocol, 317, 325, 335–336, 339, 413–414
  implementation of, 335–336
  purpose of, 335
address, socket, 296–298
address space. *See* virtual address space
address structure
  internet, 297
  socket, 284–285
  UNIX domain, 297
  XNS, 297
address translation, 110, 413
  VAX, 120–122
adjtime system call, 54
AGE buffer list, 211–213, 222, 414
algorithm
  CLOCK, 150–153, 163, 417, 429
  daylight saving time, 54
  for *disksort*(), 245
  elevator sorting, 244
  mark-and-sweep garbage collection, 306
  MASSBUS adapter interrupt handling, 255
  for physical I/O, 232
  TCP slow-start, 376–378
Alliant, 86
alloc(), 217–218
allocation
  descriptor, 299
  directory space, 188
  file block, 193–195, 199, 207–208, 218–219
  filesystem fragment, 218–219
  inode, 194
  kernel memory, 28–29
  swap space, 129
  user page table, 131
allocbuf(), 212–213
ANSI C compiler, 402
argdev, 141
ARP. *See* Address Resolution Protocol
ARPANET, 7, 11, 14, 325, 343–346, 373, 377, 380, 414, 425–426, 434
  Host Access Protocol, 380, 414
  host interface, 380–381
  Reference Model, 344

arpinput(), 336
arpresolve(), 336
assembly-language startup, 397–398
assembly language in the kernel, 22, 46–47, 90, 229, 405
AST. *See* asynchronous system trap
asynchronous I/O, 174
  in *pageout*(), 155
asynchronous system trap, 44, 90, 101, 414
  use in context switching, 90
attach routine, 236, 239, 414
autoconfiguration, 41, 400–403
  alternative schemes for, 403
  contribution of, 9
  device driver support for, 227, 234–237
  of interrupt vector, 236, 401–403
  of MASSBUS device, 253
  operation of, 401–403
  phase, 234–235, 414
  up device driver, 239–242, 402
avefree, 400

B

B programming language, 3–4
Babaoğlu, Özalp, 7
background process, 103, 265, 414, 423
backing storage, 109, 414
bad sector handling, 248, 414–415
bawrite(), 210
BCPL, 4
bdwrite(), 210
Bell Laboratories, 3–4, 13, 115
Berknet, 276, 415
/bin/login, 59, 407
/bin/sh, 56, 406
bind system call, 350
biodone(), 247
blkpref(), 217
block device, 37, 171–172, 226, 229–230, 415
  table, 227, 415
block-device interface, 169, 226–227, 229, 233, 238, 415
block I/O, 171, 192–193, 415
block size, 196, 227, 415
bmap(), 142, 206–208, 213, 217
Bolt Beranek and Newman, 7, 40, 291
boot, 393–397, 406, 409, 423
  flags, 395
  operation of, 394–396
bootdevice register, 395, 415
boothowto register, 395, 415
bootstrapping, 22, 40, 124, 230, 234, 393–396, 415
  setting time when, 53
  see also initialization, **boot**
bottom half of, 44, 415
  device driver, 228
  kernel, 44–45, 85
  terminal driver, 261
  terminal driver input, 271–273
  terminal driver output, 271
Bourne shell, 56
bread(), 209, 212, 230
break character, 272
breakpoint fault, 104, 415
brealloc(), 212
brelse(), 210
broadcast message, 317, 351–352, 356, 388, 415
  address, 317, 347–348
  IP handling of, 354
bss segment, 56, 416
b_to_q(), 266, 269
buffer cache, 130, 147, 171–172, 195, 226–227, 231
  consistency, 212
  effectiveness, 208
  implementation of, 211–213
  interface, 209–210
  management, 208–213
  memory allocation, 212–213
  structure of, 210–211
buffer list
  AGE, 211–213, 222, 414
  EMPTY, 211, 213
  LOCKED, 211
  LRU, 211, 222
buffered data path, 237, 249, 416
buffering
  filesystem, 171–172, 208–209
  network, 331–332
  terminal, 266–267
bus adapter, 225, 400, 416
bus architecture, VAX, 225
bwrite(), 210, 230

C

C-block, 266, 416
C library, 54
  system calls in the, 47
C-list, 170, 228, 266, 270, 416
C programming language, 3–4, 15, 23, 47
C++ programming language, 4
C shell, 102
C70, 291
cache
  buffer, 130, 147, 171–172, 195, 226–227, 231
  directory offset, 220
  filesystem name, 220–221
  inode, 205–206
  text-page, 127, 148
callout queue, 51–53
capability, 220, 304, 416
Carnegie-Mellon University, 281
Case Western Reserve University, 115
castor oil, 266
catq(), 267
caught signal, 25, 95, 416
cbreak mode, 38, 260, 263, 416
character device, 37–38, 170–171, 226, 230–234, 416
  table, 227, 417
character-device interface, 169, 226–227, 230, 233, 238, 261, 417
chdir system call, 36, 419
checksum, 345, 349–351, 354, 368, 372, 417
child process, 24, 75, 91, 417
chmod system call, 32
chown system call, 32
chroot system call, 36, 439
cinit(), 404
cleanup(), 155
client process, 285, 417
CLOCK algorithm, 150–153, 163, 417, 429
  changes in 4.3BSD, 153
  clock hand, 150, 417
clock, alternate, 55
clock initialization, real-time, 404
clock interrupt handling, 50–53
clock interrupt rate, 50, 404
clock, real-time, 43
close-on-exec, 175–176
close system call, 30, 32, 175, 189, 261, 287, 306, 367
closedir(), 189
cluster, 117, 417
  synchronization, 125
cold start, 393, 417
communication domain, 38, 283, 293, 417
  data structures, 293
communication protocol. *See* protocol
Computer Consoles, Inc., 12
Computer Systems Research Group, vii, 8–12, 14–15, 40, 54
configuration
  device, 234, 409
  file, 408, 418
  kernel, 408–409
  procedure, 234, 418
  time, 401
  time zone, 408
congestion control
  network, 331–332
  TCP, 376–378
  see also network buffering
connect request, 319, 418
connect system call, 285–286, 298–299, 350, 352, 366, 385, 418
connection
  queueing, socket, 296, 299
  setup, TCP, 359, 366–367
  shutdown, TCP, 359, 367–368
  states, TCP, 360
console
  media, VAX, 396
  monitor, 394, 418
  processor, 394, 418
context switching, 49, 70, 79–86, 129, 418
  AST use in, 90
  intraprocess, 83–84
  involuntary, 79, 90
  process state, 80
  during swapping, 84
  VAX influence on, 90
  voluntary, 79, 81–83
control-output routine, protocol, 322
control request, 320, 418
controlling terminal, 26, 103, 418
cooked mode, 38, 259, 263, 418
copy-on-write, 6, 28, 138, 162, 418
copyin(), 119
copyout(), 119
copyseg(), 138

core(), 101
core file, 25, 94, 419
core map, 124–128, 148–149, 419
Cornell University, 40
crash, 172, 419
crash dump, 228, 230, 410, 419
creat system call, 32
CSRG. *See* Computer Systems Research Group
CTSS operating system, 3
current working directory, 35, 190, 419
cylinder group, 197–198, 419

D

DARPA, 7, 9–11, 40, 281–282, 298, 343, 387, 413–414, 416, 419, 426, 432, 437, 447–448
  Internet, 7, 39
  steering committee, 7
data path, 237, 419
  buffered, 237, 249, 416
  unbuffered, 237, 249, 447
  UNIBUS, 237–238
data segment, 26, 55–56, 142, 419
  expansion, 142
data structures
  communication domain, 293
  filesystem, 203–208
  interprocess communication, 292–298
  socket, 294–296
datagram socket, 284, 419
daylight saving time algorithm, 54
DCD, 267–268
DCE, 267, 446
deadlock avoidance
  in file locking, 178
  during *fork* system call, 137
  when locking resources, 85–86
debugging
  information in exec header, 56
  process, 100, 103–105
  system, 410
  see also ptrace system call
decapsulation, 312, 315, 419, 421
DECnet, 328
DEFBOO.CMD, 396
Defense Data Network, 381
deficit, 157
demand paging. *See* paging
descriptor, 30, 420
  allocation, 299
  duplication, 176
  locking, 177–179
  management, 172–177
  multiplexing, 179–181
  passing, 175
  table, 30, 173, 420
desfree, 152, 157
design
  4.2BSD, 11
  4.2BSD IPC, 9, 282–283
  4.2BSD network, 13, 39–40
  mbuf, 290–291
  memory-management, 27–28
/dev/console, 406
/dev/klog, 431
/dev/kmem, 186, 399
/dev/log, 399
/dev/mem, 170, 410
/dev/null, 170
device, 37–38, 225
  configuration, 234, 409
  flags, 268, 420
  interrupt, 49–50, 226
  interrupt handler, 49
  MASSBUS, 253–256
  naming, 236–237
  number, 170, 227, 420
  slave, 236, 442
  special file, 33, 420
  swap, 114, 127, 445
  UNIBUS, 237–252
device driver, 37, 49, 169–171, 225–256, 420
  attach routine, 236
  autoconfiguration, *up*, 239–242, 402
  bottom half of, 228
  I/O strategy, MASSBUS, 253–254
  I/O strategy, *up*, 244
  internal interface to, 225
  interrupt handling, MASSBUS, 254–256
  interrupt handling, *up*, 247–249
  maximum transfer size, 171
  probe routine, 235–236
  sections of a, 227
  seek optimization in *up*, 245–246

slave routine, 236
support for autoconfiguration, 227, 234–237
support for *select* system call, 181–183
top half of, 228
use of disk geometry, 241
Digital Equipment Corporation, 7
direct memory access, 147, 232, 234, 237–239, 246, 249, 253, 270–271, 274, 336, 338, 416, 419–421, 437, 444
directed broadcast, 388, 420
directory, 34, 187, 420
entry, 35, 195, 420
offset cache, 220
operations, 32–33, 189
space allocation, 188
structure, 187–189
dirty, 420
disk geometry, device driver use of, 241
disk geometry, filesystem use of, 201
disk labels, 241–242
disk partition, 187, 230, 241, 421, 431
disksort(), 243–244, 256–257
algorithm for, 245
distributed filesystem, 13, 32
DMA. *See* direct memory access
dmesg, 399
dmmax, 128, 135
dmmin, 135, 154
dmtext, 136
doadump(), 410
domain. *See* communication domain
double indirect block, 194, 421, 425
dquot table, 214–216
DTE, 267, 446
dtom(), 291–292
DTR, 267–268, 275–276
dumpsys(), 410
dup system call, 31, 174–176, 305, 420, 422
implementation of, 176
dup2 system call, 31, 176, 422

E

ECC correction, 238, 248, 257, 421
effective GID. *See* effective group identifier
effective group identifier, 59, 421
effective UID. *See* effective user identifier
effective user identifier, 59, 421
EGP. *See* Exterior Gateway Protocol
Eighth Edition UNIX, 6, 13, 39, 104
elevator sorting algorithm, 244, 421
Elz, Robert, 10, 191
EMPTY buffer list, 211, 213
Emulex SC-21V, 238–239, 243
encapsulation, 312, 315, 317, 338, 421. *See also* trailer protocol
Encore, 86
environment, location of process, 57
erase character, 260, 421
errno, 23, 47–48, 422
error-message buffer, 398, 422
Error Protocol, 382
/etc/accton, 407
/etc/config, 234, 408–409, 418
files generated by, 409
/etc/cron, 407
/etc/fsck, 196, 231, 256, 406
/etc/getty, 406–407
/etc/gettytab, 407
/etc/group, 407
/etc/init, 404–407, 425
initial startup of, 405
/etc/mknod, 236
/etc/mtab, 406
/etc/passwd, 407
/etc/rc, 406–407
/etc/rc.local, 407
/etc/savecore, 410
/etc/syslogd, 399, 407
/etc/timed, 405
/etc/ttys, 406
/etc/update, 406
Ethernet, 7, 11, 39–40, 313, 344
exec header, 56
exec system call, 31, 59, 63, 69, 136–142, 144, 159–162, 175, 405, 407
execve system call, 24, 79, 104, 140, 438, 441
operation of, 140–142
swap space allocation during, 141
exit(), 93, 101, 144, 159
exit system call, 24, 93, 118, 139, 144–145, 159–160
operation of, 93–94, 144
status, 24–25, 75, 93
Exterior Gateway Protocol, 331–332

F

fault rate, 112, 422
fchmod system call, 32
fchown system call, 32
fcntl system call, 9, 174–176, 272, 385, 420
Federal Information Processing Standard, 9
fetch policy, 112, 422
ffs, 90, 422
FIFO, 33, 422
file, 30, 187, 422
  access validation, 58–59
  executable, 56
  hole in, 36, 424
  offset, 31, 174, 422
  permission bits, 58
file block
  allocation, 193–195, 199, 207–208, 218–219
  allocation, implementation of, 217–219
  extension, 217
  locality of reference, 202
  lookup, 206–207
  mapping, 206
  numbers for paging, precomputing, 132
  reading, 207
  writing, 207
file locking, 174, 177–179
  deadlock avoidance in, 178
  implementation of, 178–179
file structure, 31, 173, 296, 422
file table, 173–175
  flag, 174, 176, 272
  handling during *fork* system call, 175
  implementation of, 174–175
  object oriented, 173–174, 176
  operations, 173
filename, 34–35, 422
filesystem, 169, 422
  abstraction, 191–193
  access control, 36
  buffering, 171–172, 208–209
  comparison with other systems, 200
  data structures, 203–208
  disk structure, 187–191
  distributed, 13, 32
  fragment allocation, 218–219
  fragmentation, 198–200
  implementation of, 191–195
  initialization, 404, 406
  interaction with virtual memory, 141
  layout policies, 201–203
  limits, 127
  links, 189–191
  local allocation routines, 202–203
  name cache, 220–221
  name length, 37
  name translation, 36, 203–205, 219–221
  old, 196
  organization, 196–198
  overview, 34–37
  parameterization, 200–201
  /proc, 104–105, 436
  quotas, 9, 191, 213–217, 404, 408
  redesign, 195–203
  redundant information in, 197
  resource locking, 86
  storage optimization, 198–200
  use of disk geometry, 201
  see also buffer cache, quotas
fill-on-demand
  klustering, 146–147, 422
  page fault, 132, 146, 423
  page table entry, 132–133, 423
First Edition UNIX, 69
firstaddr, 399
firstfree, 124, 400
floating point in the kernel, use of, 366
flow control in TCP, 358
fodkluster(), 146
foreground process, 103, 265, 414, 423
fork system call, 3, 24, 31, 63, 69, 74, 81, 91–92, 104, 136–139, 151, 162, 175, 186, 403, 417, 422, 435–436, 438
  deadlock avoidance during, 137
  file-table handling during, 175
  implementation of, 136–138
  implementation issues, 138
  see also process creation
Forkmap, 137, 139
Fortran programming language, 15, 32
4.0BSD, 7
4.1aBSD, 14
4.1BSD, 7–8
  memory management in, 117–118
4.2BSD, 7–8
  design, 11
  IPC design, 9, 282–283
  network design, 13, 39–40
  virtual-memory interface, 8

4.3BSD, 4, 7–8
compatibility of, 12
kernel, division of software in, 21
memory management in, 118
network additions in, 40
portability of, 21
Tahoe release, 7, 10, 12, 29, 54, 119–120, 150, 232, 241–242, 365, 369, 375
virtual-memory system deficiencies, 12
fragment-descriptor table, 218, 423
fragmentation
filesystem, 198–200
system page table, 155, 157
free(), 29
free list, 195, 423
memory, 125–126, 423, 431
free-space reserve, 200, 423
freemem, 128, 149, 157, 400
fseek(), 15
fstat system call, 32, 320
fsync system call, 172, 210

G

garbage collection, 305, 423
gateway handling, 329
GENIE operating system, 3
getblk(), 212
getc(), 266, 273
gethostid system call, 425
getpeername system call, 287
getrusage system call, 61
getsockname system call, 287
getsockopt system call, 287, 318, 322
gettimeofday system call, 53–54
GID. *See* group identifier
go routine, 249, 424, 448
Greenwich time. *See* Universal Coordinated Time
group identifier, 58–60, 62, 421, 424, 438, 441, 448
use in file-access validation, 59
gsignal(), 99

H

hard limit, 62, 424
hard link, 189, 424
hardclock(), 51–52, 55, 61–62, 88–89
Harris, Guy, 10
Harvard University, 7
heap, 57, 424
high watermark on, 424
socket, 296, 301, 331
terminal, 269
history of
4.3BSD memory management, 115–118
job control, 8, 103
process management, 69
UNIX, 3–8
home directory, 36, 424
Host Dead message, 381, 425
host identifier, 60, 425
host name, 60
Host Unreachable message, 381, 425
ht device driver, 253
htstrategy(), 253
Hyperchannel, 11
hz, 404

I

I/O, 425, 427
asynchronous, 174
nonblocking, 175, 180, 299–301, 303, 432
physical, 126, 158, 231
queueing, 228
redirection, 31, 427
scatter/gather, 34, 185
signal driven, 176, 180, 442
types of kernel, 169–170
I/O buffer, 228, 230
I/O page, 237, 427
I/O stream, 30, 427
I/O strategy
MASSBUS device driver, 253–254
up device driver, 244
I/O vector, 184, 427
ICMP. *See* Internet Control Message Protocol
icmp_error(), 380
IDP. *See* Internetwork Datagram Protocol
IEEE, 8–9, 435
if_init(), 316
if_output(), 316
IMP. *See* Interface Message Processor
implementation of
ARP, 335–336
buffer cache, 211–213

dup system call, 176
file block allocation, 217–219
file locking, 178–179
file table, 174–175
filesystem, 191–195
fork system call, 136–138
ioctl system call, 176–177
pipe, 30
quotas, 191, 213–217
select system call, 181–184
trailer protocol, 338–339
uiomove(), 184–185
UNIBUS network interface, 336–338
Ingres database system, 282
init, 25, 46, 76, 151, 405, 425, 450
initial sequence number, 357, 425, 441
initial startup of **/etc/init**, 405
initialization
filesystem, 404, 406
kernel, 396–405
machine-dependent, 398–403
machine-independent, 403–405
memory allocator, 400
pagedaemon, 405
paging system, 403
real-time clock, 404
system data structures, 399–400
system processes, 403, 405
user-level system, 405–407
see also bootstrapping
inode, 187–208, 213–217, 219–223, 236, 296, 404, 425
allocation, 194
cache, 205–206
contents, 195
definition, 194–195
locality of reference, 202
management, 205–206
table, 205, 425
in_pcballoc(), 350
in_pcbbind(), 350
in_pcbconnect(), 350, 366
in_pcbdetach(), 352
in_pcblookup(), operation of, 352
intelligent gateway, 329, 425
interactive program, 71, 426
Interdata 8/32, 6
interface
buffer cache, 209–210
capabilities, network, 316–317
line switch, 261, 428
network, 315–317
protocol, 293
protocol–network-interface, 324–327
protocol-protocol, 322–323
socket-to-protocol, 318–322
Interface Message Processor, 323, 325, 331, 345–346, 380–381, 425–426
interleaving, swap space, 128
International Organization for Standardization, 9, 13, 312, 344, 429
protocol suite, 40
internet addresses
broadcast, 347–348
host, 345–348, 426
packet demultiplexing, 348
structure, 297
subnet, 346–347
Internet Control Message Protocol, 331, 344–345, 352, 357, 375, 378–381, 383, 425–426, 437
interaction with routing, 379
port unreachable message, 352
internet domain, 39, 426
Internet Protocol, 40, 276, 313, 332, 344–345, 348–357, 368, 372, 378–380, 387–388, 426–427
handling of broadcast message, 354
input processing, 354–357
options, 353
output processing, 353–354
packet demultiplexing, 349
packet forwarding, 356–357, 380
protocol header, 353
responsibilities of, 352
Internetwork Datagram Protocol, 381–385, 425–426
interprocess communication, viii, 6, 12–13, 19, 30, 33, 38–40, 61, 172, 281–307, 426–427
connection setup, 298–300
data structures, 292–298
data transfer, 300–306
design, 4.2BSD, 9, 282–283
facilities, interface design, 287
memory management in, 289–292
model of, 282–287
receiving data, 302–304
reliable delivery, 301
socket shutdown, 306–307

transmitting data, 301–302
interrupt, 426
device, 49–50, 226
priority level, 45, 85, 426
software, 50, 313, 354, 443
stack, 45, 50, 78, 96, 426
interrupt handling, 49–50, 226, 229
algorithm, MASSBUS adapter, 255
clock, 50–53
MASSBUS device driver, 254–256
up device driver, 247–249
interrupt vector, 226, 427
autoconfiguration of, 236, 401–403
table, 226, 427, 449
interrupted system call, 47–48, 98
involuntary
context switching, 79, 90
swapping, 157–158
ioctl system call, 33, 97, 103, 173, 176, 261–263, 274, 276, 304, 316–317, 320, 331, 418, 421, 428, 449
implementation of, 176–177
iovec, 184–185, 427, 447
IP. *See* Internet Protocol
IPC. *See* interprocess communication
ipintr(), operation of, 354–357
ip_output(), 351, 354, 357, 372, 379
operation of, 353–354
ISO. *See* International Organization for Standardization
issig(), operation of, 101
ITS operating system, 8

J

job, 427
job control, 26, 76, 97, 102–103, 427
history of, 8, 103
signals in 4.3BSD, 25
terminal driver support for, 265, 269, 273
use of process group, 26
Joy, William, 7

K

keepalive packet, 364, 427
keepalive timer, 364, 428
kernel, 20, 428
assembly language in the, 22, 46–47, 90, 229, 405
bottom half of, 44–45, 85
configuration, 408–409
I/O, types of, 169–170
initialization, 396–405
memory allocation, 28–29
mode, 69, 114, 428
organization, 20–23
partitioning, reason for, 20
preemption, 45
process, 45–46, 428
state, 70, 428
top half of, 44–45, 85
kernel stack, 50
growth of, 78
location of, 57, 78
size of, 78
kernelmap, 130, 137, 145, 428
initialization of, 130
kill character, 260, 428
kill system call, 96
killpg system call, 76, 96, 103, 436
klustering, 149, 428
Korn shell, 102
Kulp, Jim, 103

L

lbolt, 270
ldctx(), 84, 92
ldpctx, 81, 418, 428
least recently used, 117, 126, 150, 205–206, 211, 425, 428, 430
buffer list, 211, 222
lightweight process, 107
limits in system, 127, 191
line discipline, 260–262, 268, 276–277, 428
Berknet, 276
close(), 275–276
output(), 268–270
SLIP, 276–277
tablet, 277
line mode, 259, 429
line switch interface, 261, 428
link layer, 312, 429
link system call, 32. *See also* filesystem links
LISP programming language, 7, 115
listen request, 319, 429
listen system call, 286, 298–299, 367, 429
load average, 87–88, 429

locality of reference, 113, 146, 201–202, 429
LOCKED buffer list, 211
locking
  descriptor, 177–179
  on page, 126
  resources on a shared-memory multiprocessor, 86
  resources, deadlock avoidance when, 85–86
  socket data buffer, 302
logical
  block, 192, 429
  device unit, 236–237
  drive partitions, 243, 429
  unit, 242, 429
longjmp(), 83–84, 160
loop, 150, 429
lotsfree, 152, 154
low watermark on, 430
  socket, 296, 331
  terminal, 269, 271
LRU. *See* least recently used
lseek system call, 31, 174, 189, 422
lstat system call, 191

M

Mach operating system, 20
machine check, 398, 430
machine-dependent initialization, 398–403
machine-independent initialization, 403–405
m_adj(), 292
magic number, 56, 430
main(), 398–399, 403, 405
main memory, 109, 430
major device number, 170, 227, 236, 430
malloc(), 29, 57
mark-and-sweep algorithm, 306, 430
Massachusetts Institute of Technology, 3, 8
MASSBUS, 225–227, 403, 416, 430, 445
  device, 253–256
  device, autoconfiguration of, 253
  device driver I/O strategy, 253–254
  device driver interrupt handling, 254–256
  unit number, 229, 430
MASSBUS adapter, 253, 430
  interrupt handling algorithm, 255
  support routines, 253, 430
maxfree, 124, 400
maximum segment lifetime, 359, 361, 388, 431–432. *See also* 2MSL timer
maximum-segment-size option, TCP, 366
maxslp, 158
mbinit(), 404
Mbmap, 291
mbstart(), 254
mbuf, 172, 289–291, 431
  allocation of, 291
  design, 290–291
  utility routines, 292
mbustart(), 253–254
MC68000, 119
m_copy(), 292, 372
memall(), 127, 131
memfree(), 127
meminit(), 400
memory allocation
  buffer cache, 212–213
  kernel, 28–29
  routines, 127–128
memory allocator initialization, 400
memory free list, 125–126, 423, 431
memory management, 109–161
  in 3BSD, 116–117
  in 4.1BSD, 117–118
  in 4.3BSD, 118
  design, 27–28
  evolution of 4.3BSD, 115–118
  hardware, use of, 70
  hardware, VAX, 27, 118–124
  in IPC, 289–292
  portability of, 21, 27
  in Seventh Edition UNIX, 115–116
  system, 109, 431
  unit, 111, 431
  in UNIX/32V, 116
memory-mapped I/O architecture, 226, 431
memory overlay, 111
memory size limit, 127
message buffer, 399, 431
m_free(), 292
m_freem(), 292
m_get(), 291
m_getclr(), 291
Microsoft Corporation, 7
Milnet, X.25 interface to, 381
minfree, 152, 157
MINIX operating system, 8
minor device number, 170, 227, 236, 431

minphys(), 231
mkdir system call, 9, 32, 41
mknod system call, 33, 430–431
mmap system call, 27–28, 234
modem control, 267–268, 431
mount system call, 229
m_pullup(), 292, 351, 355, 368
MS-DOS operating system, 189
msgbufmap, 399
MSL. *See* maximum segment lifetime
mtod(), 292
MT XINU, 10
Multics operating system, 3, 8
multilevel feedback queue, 86, 432
multiplexed file, 281, 432
multiprocessor
  locking resources on a shared-memory, 86
  virtual memory for a shared-memory, 27
multiprogramming, 69–71

N

Nagle, John, 373, 375
name cache, filesystem, 220–221
name length, filesystem, 37
name translation, filesystem, 36, 203–205, 219–221
namei(), 86, 206, 219–221, 384
National Bureau of Standards, 9, 334
NCP. *See* Network Control Program
ndflush(), 271
ndqb(), 270
network
  additions in 4.3BSD, 40
  architecture, 311, 432
  buffering, 331–332
  byte order, 345, 432
  congestion control, 331–332
  data flow, 172, 312–313
  design, 4.2BSD, 13, 39–40
  layer, 312, 432
  layering, 312
  mask, 347, 432
  protocol capabilities, 314–315
  queue limiting, 332
  time synchronization, 53–54
  timer, 52, 314
  virtual terminal, 13, 432
Network Control Program, 344
Network File System, Sun Microsystems, 13
network interface, 315–317
  capabilities, 316–317
  layer, 312, 432
  UNIBUS, 336–338
new terminal driver, 262, 269, 276, 432
newbuf(), 212–213
newproc(), 405
nextc(), 267
nexus, 400, 432
nice, 25, 60, 156, 432, 437, 440
Ninth Edition UNIX, 6
nodev(), 239
nonblocking I/O, 175, 180, 299–301, 303, 432
nonlocal goto, 48, 83–84, 92, 433
noproc, 90, 93
notavail(), 212
ns_err_input(), 383
NSFnet, 39
nsintr(), 383
null modem connection, 267
nulldev(), 238–239

O

object oriented file table, 173–174, 176
octet, 345, 433
old filesystem, 196
old terminal driver, 262, 268, 276, 433
Olson, Arthur, 10
open system call, 30–32, 174, 189–190, 205–206, 229, 256, 261, 285, 287, 420
opendir(), 189
Optimal Replacement Policy, 113, 433
out-of-band data, 302, 321, 334–335, 433
  receipt of, 303
  transmission of, 301
overlay, 22, 115, 433

P

P0 base register, 122–123, 433–434
P0 length register, 122–123, 433–434
P0BR. *See* P0 base register
P0LR. *See* P0 length register
P1 base register, 122, 434
P1 length register, 122, 434
P1BR. *See* P1 base register
P1LR. *See* P1 length register

packet
  forwarding, IP, 356–357, 380
  queue, 325–327
  reception, 325–327
  transmission, 324–325
packet demultiplexing
  internet addresses, 348
  IP, 349
  XNS, 382
Packet Exchange, 382, 434–435
Packet Switch Node, 380–381, 425, 434, 437
page fault, 111, 120, 123, 126, 434
  fill-on-demand, 132, 146, 423
  on the VAX, 122–124
page, locking on, 126
page push, 154, 434
page reclaim, 148, 434
page replacement, 7, 112–113, 149–155
  in the VMS operating system, 151
  VAX influence on, 149
page table, 79
  allocation, user, 131
  expansion, 143
  layout of process, 131
  paging of, 131
  text, 134, 148, 159–161
  VAX, 119–120
page table entry, 119, 122, 131, 419, 423, 428, 434, 437, 439, 443, 445, 447
  fill-on-demand, 132–133, 423
  format of a VAX, 120
  normal, 132
  types, 131–133
  valid bit, 119, 132
page in-transit, 126, 149
pagedaemon, 45–46, 151–152, 154, 157, 162, 247, 405, 419, 428, 434
  initialization, 405
  operation of the, 151
pagein(), 61, 126, 145–146, 148–149, 434
  operation of, 145–149
  of swapped pages, 147–149
pageout(), 62, 148, 151, 153–155, 405, 434
  asynchronous I/O in, 155
  operation of, 153–155
pageout in progress, 155, 434
paging, 7, 27, 57, 112, 114, 145–149, 420, 434
  of page table, 131
  parameters, 151–152
  system initialization, 403
  systems, characteristics of, 112
panic(), 410
panic, 410, 435
parent directory, 36
parent process, 24, 75, 91, 435
partition. *See* disk partition
pathname, 35, 435
PCB. *See* process control block
PCBB. *See* Process Control Block Base
PDP-11, vii, 6, 8, 48, 69, 103, 115–116, 171, 226, 237, 276, 430
PDP-7, 3, 69
performance. *See* system performance
Perkin Elmer, 45
persist timer, 364, 435
PEX. *See* Packet Exchange
pfctlinput(), 379
physical block, 193, 435
physical I/O, 126, 158, 231
  algorithm for, 232
  restrictions on, 231
physical unit number, 229, 435
physio(), 231, 233, 249, 257
  operation of, 231
PID. *See* process identifier
pipe, 30, 281–282, 435
  implementation of, 30
  system call, 30–31, 420
pipeline, 26, 31, 435
placement policy, 112, 435
portability of
  4.3BSD, 21
  memory management, 21, 27
  Seventh Edition UNIX, 6
POSIX, viii, 8–9, 12–13, 33, 96, 99, 103, 277, 435
  signal handling, 99
pr_ctlinput(), 314, 322–323, 330, 352, 379, 383
pr_ctloutput(), 314, 318, 322, 352, 383
pr_drain(), 314
preemption
  kernel, 45
  process, 85, 90
prefetching, 146, 436
prepaging, 112, 118, 436
pr_fasttimo(), 314, 321
pr_input(), 314, 322–323

printf(), 399
priority level, interrupt, 45, 85, 426
probe routine, 235–236, 239, 436
probing, 227, 235, 436
/**proc** filesystem, 104–105, 436
proc structure, 70, 72–77, 80, 131, 436
procdup(), 92
process, 23, 69, 436
  checkpoint a, 410
  creation, 91–92, 136–140
  debugging, 100, 103–105
  flags, 74, 104
  kernel, 45–46, 428
  lightweight, 107
  limit, 127
  open file table, 205, 437
  page table, layout of, 131
  preemption, 85, 90
  profiling, 49, 55
  queues, 74
  resource accounting, 51, 62–63, 93
  scheduling, 43, 53, 58, 71–72, 83, 86–91
  state, 74, 80
  state, change of, 83, 93, 100–101, 104
  synchronization, 83
  table, 57, 437
  termination, 93–94, 144–145
process control block, 77–81, 92, 122, 131, 435–436
Process Control Block Base, 81, 435–436
process group, 26, 76, 96–97, 102, 436
  hierarchy, 75
  identifier, 76, 96, 176, 295, 436
  job-control use of, 26
  socket, 97, 295
  terminal, 97, 265, 268, 272–273, 275
process identifier, 24–26, 74–75, 91, 103, 265, 268, 435–437
  allocation during process creation, 91
process management, 23–26, 55–58, 69–105
  history of, 69
process priority, 25, 48, 60–61, 74, 76, 81, 437
  calculation of, 51, 83, 87–88
  ordering of, 77
  while sleeping, 76, 82
processor priority level, 45, 437, 441
Processor Status Longword, 46–47, 78, 437, 446
profil system call, 64
profiling
  process, 49, 55
  timer, 51, 55
program relocation, 396, 439
programming language
  B, 3–4
  BCPL, 4
  C, 3–4, 15, 23, 47
  C++, 4
  Fortran, 15, 32
  LISP, 7, 115
protocol, 39, 314–315, 418
  capabilities, network, 314–315
  control-output routine, 322
  interface, 293
  network-interface interface, 324–327
  protocol interface, 322–323
  remote procedure call, 13
  switch structure, 314, 437
protocol family, 284, 293–294, 311, 437
pr_output(), 314, 322–323
pr_slowtimo(), 314, 321
pr_usrreq(), 314, 318, 322
pseudo-DMA, 271, 437
psig(), 100–101
  operation of, 101
psignal(), 99–101
  operation of, 100–101
PSL. *See* Processor Status Longword
PSN. *See* Packet Switch Node
PTE. *See* page table entry
ptrace system call, 83, 103–104
  limitations of, 104
Purdue University, 7, 115
pure demand-paging, 112, 437
Pushmap, 154
putc(), 266

Q

q_to_b(), 266
queue limiting, network, 332
quota table, 216–217
quotachk(), 213–215
quotas
  configuration of, 404, 408
  contribution of, 9
  format of record, 214
  implementation of, 191, 213–217
  limits, 191

R

Rand Corporation, 7, 281
raw device interface, 170, 438
raw mode, 38, 260, 263, 438
raw socket, 37, 311, 332–334, 345, 379, 382, 438
  control block, 332
  input processing, 333–334
  output processing, 334
raw_input(), 379
read system call, 30, 32, 34, 39, 97, 103–104, 172, 174, 185, 189, 261, 272–273, 286–287, 300, 386, 422, 433, 436, 445
readdir(), 189
readv system call, 34, 184, 286, 427
Ready for Next Message, 380–381, 438–439
real GID. *See* real group identifier
real group identifier, 59, 438
real-time clock, 43
  initialization, 404
real-time system, UNIX as a, 90
real-time timer, 51, 55, 74
real UID. *See* real user identifier
real user identifier, 59, 438
realloccg(), 217
reboot, 409–410
reboot system call, 395, 409, 411
receive window, 361, 438, 442
reclaim, 438
  from free, 148, 438
recv system call, 34, 172, 286
recvfrom system call, 34, 286, 300
recvit(), 301
recvmsg system call, 34, 286, 300
  data structures for, 287
red zone, 72, 403, 438
reference bit, 120, 149, 439
  simulation of a, 133, 150
reference string, 112, 439
register save mask, 402, 439
relative pathname, 35, 435, 439
release engineering, 14–15
reliably delivered message socket, 340, 439
remote procedure call protocol, 13
remque, 90
remrq(), 89
rename system call, 9, 32
  addition of, 32
replacement policy, 112, 439
resident set size, 73, 151
resource accounting, process, 51, 62–63, 93
resource limit, 24, 60–62
resource map, 130, 439
resume(), 80–81, 90, 92, 160
retransmit timer, 364, 366, 439
return from interrupt instruction, 47
rewinddir(), 189
RFNM. *See* Ready for Next Message
Ritchie, Dennis, 3–4, 8
rmalloc(), 130
rmdir system call, 9, 32
rmfree(), 130, 145
root directory, 35, 439
root filesystem, 36, 393, 439
root user, 58, 444
rotational delay, 201
rotational-layout table, 218, 439
round robin, 86, 440
round-trip time, 365
  TCP estimation of, 365–366
roundrobin(), 88, 90
routing, 327–331
  daemon, 331, 419, 440
  interaction with ICMP, 379
  mechanism, 328
  policy, 328, 330–331
  redirect, 329–330, 440
  tables, 328–330
  types of, 329
RS-232 serial line, 259, 267, 446
rtalloc(), 330, 354
rtfree(), 330
rtredirect(), 330, 379
run queue, 74, 86, 440
  management of, 89–91
  VAX influence on, 89
runrun, 90
rwip(), 206–207

S

savecore, 410
savectx(), 84, 92
sbappend(), 370, 372
sbappendaddr(), 352
sblock(), 302
SBR. *See* system base register
sbrk system call, 57, 142, 424

sbunlock(), 302
SC-21V, Emulex, 238–239, 243
scanc(), 269
scatter/gather I/O, 34, 185
scatter-loading, 116, 440
sched(), 156, 405
schedcpu(), 88, 90
schednetisr(), 326
schedpaging(), 152
scheduling, 70, 440
  parameters, 24
  priority, 76, 440
  process, 43, 53, 58, 71–72, 83, 86–91
secondary storage, 109, 440
seek optimization, 241, 256
  in *up* device driver, 245–246
seekdir(), 189
segment, 110, 357, 440
  bss, 56, 416
  data, 26, 55–56, 142, 419
  stack, 26, 55, 142, 443
  text, 26, 55–56, 127, 446
select system call, 12, 181–183, 186, 233, 261, 367, 385, 399
  device driver support for, 181–183
  implementation of, 181–184
seltrue(), 234
selwait, 182–183
selwakeup(), 181–183, 271
send system call, 34, 39, 172, 286, 296, 385
send window, 361, 440
sendit(), 301
sendmsg system call, 34, 286, 300, 350
  data structures for, 287
sendsig(), 101–102
sendto system call, 34, 286, 300, 318, 350, 385
sense request, 320, 441
sequence numbers, TCP, 357
sequence space, 357, 441
sequence variables, TCP, 360–361
Sequenced Packet Protocol, 381–384, 441, 443
sequenced packet socket, 284, 441
Sequent, 86
Serial Line IP, 276–278, 441, 443
server process, 285, 441
session, 103, 441
set-group-identifier program, 59, 441
set-priority-level, 85, 441
set-user-identifier program, 59, 441
sethostid system call, 425
setjmp(), 83–84
setpgrp system call, 96
setpri(), 88–90
setpriority system call, 96, 436
setrq(), 89
setrun(), 89–90, 100
setsockopt system call, 287, 307, 318, 322, 351, 374
settimeofday system call, 54
Seventh Edition UNIX, 6–8, 12, 281
  memory management in, 115–116
  portability of, 6
shadow swap map, 141, 441
shared text segment, 6, 73, 79, 441
  limit, 127
shell, 20, 442
  Bourne, 56
  C, 102
  Korn, 102
short-term scheduling algorithm, 87, 442
shutdown system call, 287, 303, 367
SIGALRM, 55
sigblock system call, 96, 98, 430
SIGBUS, 123
SIGCHLD, 100, 104
SIGCONT, 96, 100, 275, 418
SIGHUP, 275
SIGIO, 174, 272, 295, 442
SIGKILL, 25, 95–96, 101
signal, 25–26, 73, 94–103, 442
  checking for a pending, 48
  comparison with other systems, 97–98
  delivering, 57, 101–102
  driven I/O, 176, 180, 442
  handler, 25, 94, 442
  handling, POSIX, 99
  posting, 96, 99–101
  problems addressed by 4.3BSD, 97
  restrictions on posting, 96
  stack, 25, 96
  trampoline code, 78, 102, 129, 442
sigpause system call, 96
SIGPROF, 55, 64
sigreturn system call, 96, 102, 442
SIGSEGV, 123
sigsetmask system call, 96, 98, 430
sigstack system call, 96, 98
SIGSTOP, 25, 95–96

SIGTRAP, 104
SIGTSTP, 106, 273
SIGTTIN, 273
SIGTTOU, 100, 269
SIGURG, 295
sigvec system call, 94, 97–98, 101, 416
SIGVTALRM, 55
SIGWINCH, 265
SIGXCPU, 51
silly-window syndrome, 372, 442
  TCP handling of, 372–373
single indirect block, 194, 425, 442
Sixth Edition UNIX, 4, 6, 8, 12, 77
slave device, 236, 442
slave routine, 236, 239, 442
sleep(), 77, 79, 81–86, 88, 90, 96, 99, 106, 151, 228, 256, 300, 415, 442, 446
  operation of, 82
sleep queue, 74, 442
sliding-window scheme, 358, 442
slip sector, 248, 443
  forwarding, 248, 443
SLIP. *See* Serial Line IP
slow-start algorithm, TCP, 376–378
SLR. *See* system length register
small-packet avoidance, 388, 443
  TCP implementation of, 373–374
soaccept(), 385
sobind(), 384
socantrcvmore(), 371, 386
socket, 30, 33, 39, 169, 172–173, 283, 293, 311, 443
  address, 296–298
  address structure, 284–285
  connection queueing, 296, 299
  data buffer locking, 302
  data buffering, 296, 301, 303
  data structures, 294–296
  error handling, 300
  options, 318
  process group, 97, 295
  shutdown, 306–307
  state transitions during rendezvous, 298
  state transitions during shutdown, 307
  states, 295
  types, 283, 293
  using a, 284–287
socket system call, 9, 13, 30–31, 39, 284–285, 293, 298, 318, 322, 384, 420
socket-to-protocol interface, 318–322
socketpair system call, 287, 321, 420
soconnect(), 300, 385
socreate(), 384
soft limit, 62, 443
soft link, 190, 443, 445
softclock(), 51–52, 55
software interrupt, 50, 313, 354, 443
sohasoutofband(), 370
soisconnected(), 300, 385
soisconnecting(), 366, 385
soisdisconnected(), 371
solisten(), 299
sonewconn(), 299, 367
soreceive(), 302–305, 308, 386
sorflush(), 386
sorwakeup(), 303
sosend(), 301–302, 305, 308, 372, 385, 387
soshutdown(), 386
source-quench processing, TCP, 375–376
special file, 33, 37, 173, 443
spin loop, 398, 443
splbio(), 228, 257
splimp(), 327
splnet(), 302
spltty(), 228, 441
SPP. *See* Sequenced Packet Protocol
stack, 443
  growth on VAX, 57
  kernel, 50
  location of kernel, 57
  segment, 26, 55, 142, 443
  segment expansion, 142
  zero filling of user, 57
stale translation, 124, 443
standalone, 443
  device driver, 395, 444
  I/O library, 394, 444
  program, 394–396
standard error, 31, 444
standard input, 31, 444
standard output, 31, 444
Stanford University, 14
startrtclock(), 404
startup(), 398–399, 403
stat system call, 32, 190–191, 320, 441
statistics collection, 51, 55, 61
sticky bit, 118, 148, 162, 444
sticky text, 148, 444

stop character, 234, 270
stream I/O system, 6, 13, 444
stream socket, 284, 444
subnet, 12, 346
  internet addresses, 346–347
Sun Microsystems, 119, 265
  Network File System, 13
superblock, 196, 444
superuser, 58, 177, 444
svpctx, 81, 418, 444
swap(), 154
swap area, 114, 444
swap device, 114, 127, 445
swap map, 135, 445
swap space, 114, 445
  allocation, 129
  allocation during *execve* system call, 141
  allocation during *fork* system call, 136
  expansion, 135
  interleaving, 128
  management, 128–129, 134–136
swapin(), 83
  operation of, 159–160
swapon system call, 129
swapout(), operation of, 158–159
swapper, 45, 156–159, 428, 445
  operation of, 156
swapping, 27, 58, 113, 137, 151, 155–161, 445
  in 4.3BSD, reasons for, 155
  context switching during, 84
  involuntary, 157–158
  a text segment, 160–161
swdone(), 154
swtch(), 80–82, 90, 94
  operation of, 90
symbolic link, 190–191, 445
sync system call, 172, 186, 207
synchronization, 84–86
  cluster, 125
  network time, 53–54
syscall(), 46
Sysmap, 130
system activity, 43–44, 445
system base register, 120–121, 440, 445
system bus, 400, 445
system call, 20, 23, 44–45, 445
  handling, 28, 46–49, 80
  interrupted, 47–48, 98
  result handling, 47–48
system calls
  accept, 286, 296, 298–299, 308, 367, 385
  adjtime, 54
  bind, 350
  chdir, 36, 419
  chmod, 32
  chown, 32
  chroot, 36, 439
  close, 30, 32, 175, 189, 261, 287, 306, 367
  connect, 285–286, 298–299, 350, 352, 366, 385, 418
  creat, 32
  dup, 31, 174–176, 305, 420, 422
  dup2, 31, 176, 422
  exec, 31, 59, 63, 69, 136–142, 144, 159–162, 175, 407
  execve, 24, 79, 104, 140, 438, 441
  exit, 24, 93, 118, 139, 144–145, 159–160
  fchmod, 32
  fchown, 32
  fcntl, 9, 174–176, 272, 385, 420
  fork, 3, 24, 31, 63, 69, 74, 81, 91–92, 104, 136–139, 151, 162, 175, 186, 403, 417, 422, 435–436, 438
  fstat, 32, 320
  fsync, 172, 210
  gethostid, 425
  getpeername, 287
  getrusage, 61
  getsockname, 287
  getsockopt, 287, 318, 322
  gettimeofday, 53–54
  ioctl, 33, 97, 103, 173, 176, 261–263, 274, 276, 304, 316–317, 320, 331, 418, 421, 428, 449
  kill, 96
  killpg, 76, 96, 103, 436
  link, 32
  listen, 286, 298–299, 367, 429
  lseek, 31, 174, 189, 422
  lstat, 191
  mkdir, 9, 32, 41
  mknod, 33, 430–431
  mmap, 27–28, 234
  mount, 229
  open, 30–32, 174, 189–190, 205–206, 229, 256, 261, 285, 287, 420

pipe, 30–31, 420
profil, 64
ptrace, 83, 103–104
read, 30, 32, 34, 39, 97, 103–104, 172, 174, 185, 189, 261, 272–273, 286–287, 300, 386, 422, 433, 436, 445
readv, 34, 184, 286, 427
reboot, 395, 409, 411
recv, 34, 172, 286
recvfrom, 34, 286, 300
recvmsg, 34, 286, 300
rename, 9, 32
rmdir, 9, 32
sbrk, 57, 142, 424
select, 12, 181–183, 186, 233, 261, 367, 385, 399
send, 34, 39, 172, 286, 296, 385
sendmsg, 34, 286, 300, 350
sendto, 34, 286, 300, 318, 350, 385
sethostid, 425
setpgrp, 96
setpriority, 96, 436
setsockopt, 287, 307, 318, 322, 351, 374
settimeofday, 54
shutdown, 287, 303, 367
sigblock, 96, 98, 430
sigpause, 96
sigreturn, 96, 102, 442
sigsetmask, 96, 98, 430
sigstack, 96, 98
sigvec, 94, 97–98, 101, 416
socket, 9, 13, 30–31, 39, 284–285, 293, 298, 318, 322, 384, 420
socketpair, 287, 321, 420
stat, 32, 190–191, 320, 441
swapon, 129
sync, 172, 186, 207
truncate, 32–33
unlink, 32
vfork, 91, 134, 136, 138–139, 144, 162
vhangup, 407
wait, 25, 69, 93–94, 104, 139, 145
wait3, 25, 93–94, 104
write, 30, 32–34, 39, 97, 103–104, 172, 174, 185, 199, 261, 270, 286–287, 296, 300, 385, 422, 436, 445
writev, 34, 184, 286, 427
system debugging, 410
system identification register, 398, 445
use of the, 398
system length register, 120–121, 443, 445
system page table, 120, 129, 445
allocation, 130
fragmentation, 155, 157
layout, 130
system performance, 12, 47, 50–52, 54, 57, 70, 90, 301, 404
system processes initialization, 403, 405
system region, 119, 445
system shutdown, 409–410
system startup, 393–394
initial state, 397

T

Tahoe release. *See* 4.3BSD Tahoe release
TCP. *See* Transmission Control Protocol
tcp_close(), 368
tcp_ctloutput(), 374
tcp_fasttimo(), 365, 370, 375
tcp_input(), 363, 371, 374
operation of, 368–371
tcp_output(), 363, 366, 370–372, 374–375
operation of, 372
tcp_slowtimo(), 363, 365
tcp_timers(), 363–364
tcp_usrreq(), 363, 366–367, 372, 374
telldir(), 189
TENEX operating system, 8, 103
terminal, 38, 446
buffering, 266–267
multiplexer, 226, 259, 446
terminal driver, 234, 260, 262, 447
bottom half of, 261
close(), 275–276
data queues, 264, 266–267, 269–274
hardware state, 263–264
input, 271–273
input, bottom half of, 271–273
input silo, 271–272
input, top half of, 273
ioctl(), 262–263, 274
modem control, 267–268
modem transitions, 275
modes, 259–260, 264, 272
open(), 268
output, 270–271
output, bottom half of, 271
output, *stop*(), 273–274
output, top half of, 270–271

software state, 265
special characters, 259–260, 264
start(), 270
top half of, 261
user interface, 8, 262–263
window size, 265, 268
terminal process group, 97, 265, 268, 272–273, 275
text-page cache, 127, 148
text page table, 134, 148, 159–161
initialization, 141
text segment, 26, 55–56, 127, 446
swapping a, 160–161
see also shared text segment
text structure, 72, 79–80, 446
Thompson, Ken, 3–4, 8, 20
thrashing, 71, 446
3BSD, memory management in, 116–117
tick, 50, 446
time, 53–55
algorithm, daylight saving, 54
of day, 43
of day register, 53
quantum, 86, 446
representation, 54–55
slice, 71, 86, 446
synchronization, network, 53–54
time zone, 53–54
configuration, 408
handling, 10, 54
timeout(), 51–53, 270
timer
2MSL, 364–365, 447
backoff, 364, 446
network, 52, 314
profiling, 51, 55
real-time, 51, 55, 74
resolution of, 54
virtual-time, 55
watchdog, 52
TLB. *See* translation lookaside buffer
top half of, 44, 446
device driver, 228
kernel, 44–45, 85
terminal driver, 261
terminal driver input, 273
terminal driver output, 270–271
TOPS-20 operating system, 8
trace trap, 104, 446
traced process, 100, 104
trailer protocol, 317, 338–339, 447
implementation of, 338–339
negotiation, 339
translation lookaside buffer, 124, 446–447
translation-not-valid fault, 120, 447
Transmission Control Protocol, viii, 12, 40, 313, 330, 334, 344–345, 348, 357–379, 384, 387–388, 446–447
congestion control, 376–378
connection setup, 359, 366–367
connection shutdown, 359, 367–368
connection states, 360
delayed acknowledgements in, 370
estimation of round-trip time, 365–366
features of, 357
flow control in, 358
handling of silly-window syndrome, 372–373
handling of urgent data, 370
implementation of small packet avoidance, 373–374
implementation, use of 4BSD, 9
input processing, 368–371
maximum-segment-size option, 366
options, 358
output processing, 371–378
packet header, 358
retransmission handling, 375
send policy, 363, 371–378
sequence numbers, 357
sequence variables, 360–361
slow-start algorithm, 376–378
source-quench processing, 375–376
state diagram, 361
timers, 363–365
window updates, 374–375
transport layer, 312, 447
trap(), 46
trap handling, 44, 46–47, 49–50, 80
trap type code, 46
triple indirect block, 194, 425, 447
truncate system call, 32–33
addition of, 32
ttread(), 273
ttrstrt(), 271
ttselect(), 234, 261
ttstart(), 270, 277
ttwakeup(), 272
ttwrite(), 269–270, 273
tty driver. *See* terminal driver

tty structure, 263–265
ttyclose(), 276
ttyinput(), 272
ttylclose(), 275
ttymodem(), 275
ttyopen(), 268
ttyoutput(), 269–270
Tunis operating system, 8, 20
2MSL timer, 364–365, 447. *See also* maximum segment lifetime
type-ahead, 259, 447

U

u-dot. *See* user structure
ubadone(), 249–250
ubago(), 247, 249–250, 254
ubarelse(), 251–252
ubasetup(), 250
UDP. *See* User Datagram Protocol
udp_input(), 351
udp_output(), 350
udp_usrreq(), 350, 352
UID. *See* user identifier
uio, 184–185, 231–232, 268, 273, 447
uiomove(), 185, 233, 269
  implementation of, 184–185
Ultrix operating system, 23
unbuffered data path, 237, 249, 447
UNIBUS, 225–227, 270–271, 403, 416, 445, 448
  data path, 237–238
  device, 237–252
  network interface, 336–338
UNIBUS adapter, 237, 448
  support routines, 249–252
Universal Coordinated Time, 53–54, 64, 408
University of California at Berkeley, 7, 115
University of Illinois, 7
University of Maryland, 40
UNIX/32V, 6–7, 11
  memory management in, 116
UNIX as a real-time system, 90
UNIX domain, 38, 448
  address structure, 297
  passing access rights in the, 305–306
UNIX, history of, 3–8
UNIX Programmer's Manual, 4
UNIX Support Group, 6
UNIX System III, 6–9, 39
UNIX System V, 4, 6–9, 177, 195
  Release 3, 6, 13, 103
unlink system call, 32
unp_gc(), 306
unputc(), 267
unselect(), 182
up device driver, 236, 238–249, 253, 256–257
  autoconfiguration, 239–242, 402
  I/O strategy, 244
  interrupt handling, 247–249
  seek optimization in, 245–246
update, 172, 186, 207
updatepri(), 89
ureadc(), 273
urgent data, 334, 448
  TCP handling of, 370
  transmission, styles of, 302
USENET, 10
user area. *See* user structure
User Datagram Protocol, 330, 344–345, 348–352, 366, 372, 379, 384, 387–388, 447–448
user identifier, 58–60, 62, 64, 92, 421, 425, 438, 441, 444, 447–448
  use in file-access validation, 58
user-level system initialization, 405–407
user mode, 69, 114, 448
user page table, 129, 448
  allocation, 131
user request routine, 314, 318–321, 448
  operations, 318–321
user structure, 44, 57, 70, 72, 77–78, 131, 448
  contents of, 77
  location of, 57
  mapping on VAX, 78
Usrptmap, 130–131
UTC. *See* Universal Coordinated Time

V

V Kernel operating system, 20
valid bit, PTE, 119, 132
VAX, vii, viii, 6–7, 11–12, 21, 45
  VAX-11/750, 117
  VAX-11/780, 6, 116
  address translation, 120–122

architecture, 400, 449
bus architecture, 225
console media, 396
influence on context switching, 90
influence on *fork* system call implementation, 138
influence on page replacement, 149
memory management hardware, 27, 118–124
page table, 119–120
stack growth on, 57
system, 400, 449
user structure mapping on, 78
VAX peripheral
DH-11, 270, 278
DMR-11, 317
DZ-11, 271, 278
vectored interrupt, 226, 449
vfork system call, 91, 134, 136, 138–139, 144, 162
operation of, 139–140
see also process creation
vgetpt(), 137
vhangup system call, 407
virtual address space, 110, 449
4.3BSD process, 129
layout of user, 55–58
on the VAX, 118–119
virtual memory, 7, 111, 449
for a shared-memory multiprocessor, 27
advantages of, 114
hardware requirements for, 114–115
interaction with filesystem, 141
interface, 4.2BSD, 8
system deficiencies, 4.3BSD, 12
virtual-time timer, 55
vmemall(), 127, 131, 137
vmemfree(), 127, 144–145
vminit(), 403
VMS operating system, viii, 8–9, 151, 403
page replacement in the, 151
vmunix, 393, 395, 405, 409–410
voluntary context switching, 79, 81–83

W

wait channel, 73, 81, 84, 449
wait system call, 25, 69, 93–94, 104, 139, 145, 449
operation of, 94
wait3 system call, 25, 93–94, 104
differences from *wait*, 93
operation of, 94
wakeup(), 82–83, 85, 89–90, 104, 155, 160
operation of, 83
watchdog timer, 52
wildcard route, 329, 449
window probe, 364, 449
window size, 265, 268
window system, 102, 265. *See also* X Window System
word-erase character, 260, 449
working set, 113, 450
workstation, 109, 118, 147, 163
write system call, 30, 32–34, 39, 97, 103–104, 172, 174, 185, 199, 261, 270, 286–287, 296, 300, 385, 422, 436, 445
writev system call, 34, 184, 286, 427

X

X/OPEN, 9
X Window System, 265, 374
X.25 interface to Milnet, 381
XENIX operating system, 7
Xerox Network System, 12, 39–40, 322, 381–384, 450
address structure, 297
domain, 39, 381–384, 450
packet demultiplexing, 382
XINU operating system, 8
XNS. *See* Xerox Network System

Z

zero filling of user stack, 57
zombie process, 74, 93–94, 450

The Design and Implementation of the

4.3BSD UNIX® Operating System

Samuel J. Leffler
Silicon Graphics, Inc.

Marshall Kirk McKusick
University of California, Berkeley

Michael J. Karels
University of California, Berkeley

John S. Quarterman
Texas Internet Consulting

This book is the first authoritative description of the design and implementation of the research versions of the UNIX® System developed at the University of California at Berkeley. It covers the *internal* structure of the 4.3BSD system and the concepts, data structures, and algorithms used in implementing the system facilities. The book also includes a chapter on the implementation of TCP/IP—a networking protocol suite widely implemented throughout the world.

Both philosophical and design issues of 4.3BSD are discussed, as well as details of the actual implementation. In most cases, the discussion starts at the system-call level and descends from the interface to the kernel down to the hardware itself. The kernel includes system facilities such as process management, memory management, the I/O system, the filesystem, the socket IPC mechanism, and network-protocol implementations.

The Design and Implementation of the 4.3BSD UNIX® Operating System is an in-depth study of a contemporary operating system. This book assumes that the reader understands basic operating-system terminology and has some familiarity with any version of the UNIX® System and with the C programming language. Therefore, this book is suitable for operating-system implementors, systems programmers, and UNIX® application developers.

X004GQV9PF
The Design and Implementati...
USED PK G
Box # 13
06196

Addison-Wesley Publishing Company