Interprocess Communication

The original UNIX systems had pipes as the only interprocess communication method.

An improved interprocess communications interface was designed for 4.2BSD.

The primary goales for the 4.2BSD interprocess communication were:

1. Standardized interface for network communication.
2. Making communication between unrelated local processes possible.
3. Provide communication facilities suitable for local area networks for access to services such as fileservers.

Interprocess Communication

Specific goals for the new BSD interprocess communication:

Transparency: Communication between processes should not depend on the processes being at the same machine.

Efficiency: Network implementations need to be efficient. Extra layers should not be used in the implementation unless absolutely necessary for proper functioning.

Compatibility: Existing *naive processes* should be able to use the interface without change. More advanced facilities are provided for *sophisticated processes* (that need to be written for the new interface).

Interprocess Communication

The goals led to the following requirements:

- The system must support networks that use different communication protocols. The notion of a *communication domain* was defined for this purpose.
- A unified abstraction for a communication endpoint is needed that can be manipulated with a file descriptor. This communication endpoint was called a *socket*.
- The semantic aspects of communication must be made available to applications in a controlled way. The *socket types* were introduced for this purpose.
- Processes must be able to locate communication endpoints so that they can communicate without being related. Thus it must be possible to attach a name to a socket.

Common socket types are *stream sockets*, *datagram sockets* and *sequenced packet sockets*.

Socket - Server code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
int main()
{
   int sd, ns, len;
   struct sockaddr_un server_address, client_address;
   char buf[256];
   //create a socket for the client
   sd = socket(AF_UNIX, SOCK_STREAM, 0);
   //Name the socket
   server_address.sun_family = AF_UNIX;
   strcpy(server_address.sun_path, "sockname");
   len = sizeof(server_address);
   if (bind(sd, (struct sockaddr *)&server_address, len) < 0)
       printf("Cannot bind\n");

   //Create a connection queue and wait for clients.
   listen(sd, 2);
   while(1) {
      //Accept a connection.
      len = sizeof(client_address);
      ns = accept(sd, (struct sockaddr *)&client_address, &len);
      if (fork() == 0)
      { /* child */
         close(sd);
         read(ns, buf, sizeof(buf));
         printf("server read %s\n",buf);
      }
      close(ns);
   }
}
```

Socket - Client code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
int main ()
{
   int sd, len;
   struct sockaddr_un address;
   int result;

   //create a socket for the client
   sd = socket(AF_UNIX, SOCK_STREAM, 0);

   //Name the socket as agreed with the server
   address.sun_family=AF_UNIX;
   strcpy(address.sun_path,"sockaddr");
   len = sizeof(address);

   //Now connect our socket to the server's socket
   result = connect(sd, (struct sockaddr *)&address, len);
   if(result == -1) exit (1);
   write(sd, "hi guy", 6);
   //Close the socked
   close(sd);
   return (0);
}
```

Socket System Calls

Several new read and write system calls were implemented for the socket interface (Table 11.1).

There are also several other new system calls such as:

getsockname Return the locally bound name of a socket.

getpeername Return the address of the remote end of the socket connection.

There are also two *ioctl*-style calls - *setsockopt* and *getsockopt* - to set and retrieve various parameters that control the operation of a socket.

Implementation structure

- The interprocess-communication facilities are layered on top of the networking facilities (Figure 11.2).
- Data flows through the socket layer to the network protocols layer and down to the network interfaces layer and up again in the receiving end.
- State required by the socket level is fully encapsulated within the socket layer and state related to the protocols are maintained in data structures specific to each protocol.
- Within the socket layer, the *socket* data structure (Fig. 11.7) is the focus of all activity.
- For system calls related to sockets, most of the work in the system calls are performed by a number of second-level routines.
- This second-level routines all have a name with the prefix so (Table 11.2).

Memory Management

- Protocol implementations must frequently prepend headers or remove headers from data.
- As packets are sent, buffered data may need to be divided into packets and received data may need to be combined to a single record.
- A special purpose memory-management facility exist in the kernel for use by the networking systems.
- This memory management facility is based on the *mbuf* data structure.

Mbufs

- Mbufs (Memory buffers) are fixed size data buffers. All mbufs has the same basic header (Fig. 11.3).
- The *m_len* field shows the number of valid bytes in the mbuf starting at the location pointed to by *m_data*.
- The data structure makes it easy to remove data at the start or at the end of the buffer.
  - ➝ To remove data at the start of the *mbuf*, the *m_data* field is increased and *m_len* is decreased.
  - ➝ To remove data at the end of an mbuf, the *m_len* field is decreased.
- Multiple mbufs linked together by the *m_next* field is treated as a single object.
- Chains of mbufs linked together with the *m_nextpkt* field is called a queue.

Mbufs

The structure of an *mbuf* can be modified by the m_flags field:

M_EXT The mbuf uses a storage area external to the mbuf (fig. 11.5).

M_PKTHDR This mbuf have an extra header that gives more information about the packet stored in the mbuf chain. Only used at the first mbuf in a chain (fig. 11.4).

M_EOR This mbuf completes a record

The external data area used by an *mbuf* is called a *mbuf cluster*. Several *mbufs* can refer to the same external data area.

Storage-Management Algorithms

- Older single-processor versions of BSD used the standard kernel memory allocator to allocate memory for mbufs.
- This method is not efficient on a SMP machine.
- FreeBSD 5.2 gives each CPU a private container of mbufs and clusters using the zone allocator.
- There is also a shared general pool of mbufs that are used if the per-CPU list is empty.

The following routines are defined for mbuf allocation:

m_get()	Allocate an mbuf.
m_gethdr()	Allocate an mbuf with an extra header.
m_clget()	Add an external cluster to an mbuf.
m_free()	Free a single mbuf.
m_freem()	Free a chain of mbufs.

Mbuf Utility Routines

Many routines are defined within the kernel for manipulating mbufs:

m_copym()	Make a copy of a chain of mbufs to another chain of mbufs. It the mbufs refer to an external cluster, the copy will reference the same data.
m_copydata()	Copy a chain of mbufs to a normal memory area.
m_adj()	Adjust (decrease) the data in an mbuf (by adjusting the m_len and m_data fields)
M_PREPEND()	Macro that prepends a specified number of data bytes to an mbuf.

Data structures

- Sockets are described by a *socket struct* that is dynamically created at the time of a *socket* system call.
- The four socket types currently supported are shown in Table 11.3.
- Communication domains are described by a *domain* data structure (fig. 11.6) that is statically allocated.
- Communication protocols within a domain are described by a *protosw struct* that is also statically allocated for each protocol.
- The domain struct is described in Fig. 11.6.

Some fields in the domain struct:

dom_name	ASCII name of the communication domain.
dom_family	Identifies the *address family* used by the domain (Table 11.4).
dom_protosw	Points to a table of functions that implement the protocol routines.

Sockets

- The *socket struct* is described in Fig. 11.7.
- Storage for the *socket struct* is allocated by the UMA zone allocator.
- The *socket struct* contains information about the socket's type, protocol and state (Table 11.5).
- At system call level, sockets are located via the file descriptor.
- Data being sent or received are queued at the socket as a list of mbufs.
- Data that is written on a socket is passed to the network subsystem as a chain of mbufs for immediate transmission.
- Received data are passed up to the socket layer in mbuf chains, where it is queued until the application makes a read system call.
- The maximum amount of data that may be queued in a socket data buffer is limited by the *high watermark*.
- The network protocols can examine the high watermark and use it for flow control.
- There is also a *low watermark* in each socket data buffer.

Sockets and Accept

Sockets used to accept incoming connections maintain two queues of sockets associated with connection requests:

- The *so_incomp* field represents a queue of sockets that need to be completed at the protocol level.
- The *so_comp* field heads a list of sockets that are ready to be returned to the listening process.

Socket Address

- Sockets may be assigned an address so that peers can connect to them.
- The socket layer do not store addresses, but pass them down to the protocol layer.
- Each protocol has its own kind of address and the socket layer must be able to handle them all.
- The *sockaddr struct* (Fig. 11.8) is used to exchange addresses between the socket layer and the lower layers.
- The *sockaddr struct* always starts with a *sa_len* field and a *sa_family* field, the remainder is dependent on the sa_family field (Fig 11.9).

Connection Setup

- For two processes to pass information between them, an *association* must be set up.
- The steps in creating an association involves the system calls, *socket*, *connect*, *listen* and *accept*.
- We only considers connection based protocols here.
- Connection establishment in a client-server model is asymmetric.
- If a socket is used to accept incoming calls, a *listen* system call must be used.
- The listen system call invokes *solisten()*:
  - → Establish an empty queue at *so_comp* and set the socket in SO_ACCEPTCON state.
  - → Inform the supporting protocol that the socket will be receiving connections.
  - → A parameter to *listen* specified the max number of connections that will be queued at *so_comp*.

Connection Setup

Once a socket is set up to establish connections, the remainder is performed by the protocol layers.

- For each connection established at the server side a new socket is created with the *sonewconn()* routine.
  - → These new sockets may be placed at the *so_incomp* queue while processed or put directly at the *so_comp* queue awaiting an accept system call (Fig. 11.11).
- When an accept system call is made, the system checks if a connection is present on the *so_comp* queue.
  - → If no connection is present, the process is put to sleep until one arrives.
  - → If a connection is present, the socket is removed from the *so_comp* queue and a file descriptor is allocated to reference the socket and returned to the caller.

Client side.

An application that wants to connect uses the *connect* system call.

- If the connecting socket is in unconnected state, soconnect() makes a request to the protocol layer to initiate a connection.

The state transitions during connection is shown in Fig. 11.10.

Date Transfer

Sending and receiving data can be done by any of several different system calls (Table 11.1).

The system calls *read* and *write* can only handle sockets that use connection oriented protocols, since they require a file descriptor.

Datagram oriented protocols need extended system calls like *sendmsg* and *recvmsg* that can take an address as parameter.

Internally all transmit and receive requests are converted to a uniform format and passed to the socket-layer *sendit()* and *recvit()* routines.

Transmitting Data

The *sendit()* routine is responsible for gathering all system-call parameters into the kernel's address space (with exception for the data) and call *sosend()*.

sosend()

- Sosend() copies data from the user's address space into mbufs in the kernel's address space and calls the protocol layer to transmit the data.
- Sosend() is also responsible for putting processes to sleep if it is insufficient space in the socket's send buffer.
- For sockets that guarantee reliable data delivery, a protocol will normally maintain a copy of all transmitted data in the sockets send queue until it acknowledged by the receiver.

Receiving Data

The *recvit()* routine does similar tasks as *sendit()* and then calls *soreceive()*.

soreceive()

- Soreceive() checks the socket's state for incoming data, errors or state changes and processes queued data according to its type and actions specified by the caller.
- Data in the receive buffers are differently organized for stream, datagram and sequenced-packet sockets.
- In the general case, the receive buffers are organized as a list of buffers (Fig. 11.12).
- Receive calls normally return as soon as the *low watermark* (default 1 byte) is reached.
- If no data or error exists, *soreceive()* puts the process to sleep.
- When data arrive for a socket, the supporting protocol notifies the socket layer by calling *sorwakeup()*.

Socket Shutdown

Closing a socket may be a complicated operation for the following reason:

- In certain situations as when a process exits a close call is never expected to fail.
- However, if a socket promising reliable data delivery is closed with data still queued, the system must attempt to transmit the data perhaps indefinitely.

The socket layer compromises in an effort to address this problem yet maintaining the semantics of close.

- State transitions for close is shown in Fig 11.13.
- If a socket in connected state is closed, a disconnect is initiated, the socket marked to indicate that the filedescriptor is no longer referencing it and the close returns successfully.
- When the disconnect request completes, the network routines notifies the socket layer and the resources are reclaimed.
- Connection oriented protocols normally attempt to transmit any queued data after close returns.