TCP/IP Stack Introduction:
Looking Under the Hood!

Shiv Kalyanaraman
Rensselaer Polytechnic Institute
shivkuma@ecse.rpi.edu
http://www.ecse.rpi.edu/Homepages/shivkuma
Example program

1. Create UDP datagram socket; fill in server address
2. Send datagram to a server
3. Read datagram returned by server
Example Program: UDP Datagram

```c
13 int
14 main()
15 {
16   struct sockaddr_in serv;
17   char   buff[BUFFSIZE];
18   int    sockfd, n;
19   if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
20     err_sys("socket error");
21   bzero((char *) &serv, sizeof(serv));
22   serv.sin_family = AF_INET;
23   serv.sin_addr.s_addr = inet_addr("140.252.1.32");
24   serv.sin_port = htons(13);
25   if (sendto(sockfd, buff, BUFFSIZE, 0,
26             (struct sockaddr *) &serv, sizeof(serv)) != BUFFSIZE)
27     err_sys("sendto error");
28   if ((n = recvfrom(sockfd, buff, BUFFSIZE, 0,
29             (struct sockaddr *) NULL, (int *) NULL)) < 2)
30     err_sys("recvfrom error");
31   buff[n - 2] = 0; /* null terminate */
32   printf("%s\n", buff);
33   exit(0);
34 }
```

**Figure 1.2** Example program: send a datagram to the UDP daytime server and read a response.
Organization of Networking Code

Figure 1.3 The general organization of networking code in Net/3.
Descriptor → Kernel Data Structures

Figure 1.4 Fundamental relationship between kernel data structures starting with a descriptor.
Descriptors

- Socket: is like other UNIX descriptors
  - Can read(), write(), dup(), fcntl(), close(); shared between parent and child after fork()

- Socket descriptor → file{} → socket{}
  - Fileops{}: convenient place for socket operations
  - Doubly linked list of protocol control blocks (PCBs)
    - Sendto: Access PCBs in top-down manner
    - Receive: PCBs searched for port-number; then socket{} accessed through inp_socket
Figure 1.5: Kernel data structures after call to socket in example program.
Memory Buffers (mbufs)

- *Used for passing socket address structs, data etc*
- 128-bytes, with 20 byte mbuf header
- Mbuf chain with mbuf packet header
  - `m_next`: links mbufs within a packet
  - `m_nextpkt`: links multiple pkts (queue of mbufs)
- 2048-byte *clusters* used if data > 208 bytes

- Nice examples: Figure 1.7, 1.8 (pg 16, 17)
Example: mbuf chain with data

- 150 bytes of data divided into 2 mbufs: 100 + 50 bytes
- Note: pkthdr fields (and mflags = M_PKTHDR) only in first mbuf

Figure 1.7 Two mbufs holding 150 bytes of data.
Mbufs: header and data

- Protocol (IP + UDP) headers prepended in another mbuf (avoid copies in future)! 
- Pkthdr fields moved to first mbuf in this new chain. 
- Headers are in the end of the first mbuf: allows space for ethernet header prepending 
- M_data points to appropriate location

Figure 1.8  Mbuf chain from Figure 1.7 with another mbuf for IP and UDP headers prepended.
TCP/IP Stack

Figure 1.12 Communication between the layers for network input and output.
Output Functions at Various Levels

- **UDP** output routine fills in UDP header and as much of the IP header as possible (e.g., IP checksum not filled; but destination address & UDP checksum filled)
  - 2 copy passes on 150 bytes of data so far:
    - A) to copy from process to kernel mbuf;
    - B) to compute UDP checksum
- **IP** fills in remaining fields, determines outgoing interface, fragments IP datagram if necessary, and calls the interface output function
- **Ethernet output**: converts IP addr to Ethernet address (ARP); prepend 14-byte Enet header; add mbuf chain to output queue for interface; “start” interface if not busy.
- **Interface** copies data to transmit buffer (3rd copy pass), initiates output & release mbuf chain
Input processing

- **Recvfrom()**: input is asynchronous: interrupt driven & not system-call driven
- Device interrupt => ethernet device driver scheduled
  - Ethernet driver reads data into an mbuf chain (in our case 54 bytes received: 26 byte payload, 28 bytes of UDP/IP headers)
  - Pkthdr.rcvif field in mbuf points to interface structure (used only for input)
- Device driver passes mbuf to a general Ethernet input routine which does de-multiplexing based upon the type field.
  - Add to IP queue, and schedule a software interrupt for IP
- **IP input** triggered by software interrupt.
  - IP processes each datagram in its input queue: verifies chksum, processes IP options, sanity checks.
  - Either forward pkt (if router), else demux it to the appropriate protocol (UDP)
- **UDP** sanity checks, decides whether a process should receive dgram.
  - Goes thru UDP PCBs looking for inp_lport (local port) & append two mbufs to **socket receive queue**
- 1st mbuf: MT_SONAME
- The length fields refer to the data only (just a quick pointer manipulation based upon fig 1.10)
## Functions to block interrupts

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>spl0</td>
<td>normal operating mode, nothing blocked</td>
<td>(lowest)</td>
</tr>
<tr>
<td>splsoftclock</td>
<td>low-priority clock processing</td>
<td></td>
</tr>
<tr>
<td>splnet</td>
<td>network protocol processing</td>
<td></td>
</tr>
<tr>
<td>spltty</td>
<td>terminal I/O</td>
<td></td>
</tr>
<tr>
<td>splbio</td>
<td>disk and tape I/O</td>
<td></td>
</tr>
<tr>
<td>splimp</td>
<td>network device I/O</td>
<td></td>
</tr>
<tr>
<td>splclock</td>
<td>high-priority clock processing</td>
<td></td>
</tr>
<tr>
<td>splhigh</td>
<td>all interrupts blocked</td>
<td>(highest)</td>
</tr>
<tr>
<td>splx(s)</td>
<td>(see text)</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 1.13  Kernel functions that block selected interrupts.*
Interrupt Priority Levels

Device interrupt -> interface layer executes …
Software interrupt -> protocol layer executes…
then the socket code executes…

Figure 1.14

Figure 1.14 Example of priority levels and kernel processing.