Operating Systems Design
16. Networking: Sockets

Paul Krzyzanowski
pxk@cs.rutgers.edu

1

## Sockets

- IP lets us send data between machines
- TCP & UDP are *transport layer* protocols
  - Contain **port number** to identify transport endpoint (application)
- The most popular abstraction for transport layer connectivity: **sockets**
  - Developed at UC Berkeley

2

## Sockets

Attempt at a generalized IPC model

Goals:
- communication between processes should not depend on whether they are on the same machine
- efficiency
- compatibility
- support different protocols and naming conventions
  - Not just IP networking

3

## Socket

Abstract object from which messages are sent and received
- Looks like a file descriptor
- Application can select particular style of communication
  - Virtual circuit, datagram, message-based, in-order delivery
- Unrelated processes should be able to locate communication endpoints
  - Sockets can have a *name*
  - Name should be meaningful in the communications domain

4

Programming with sockets

5

## Socket-related system calls

- Sockets are the interface the operating system provides for access to the network
- Next: a connection-oriented example

6

## Step 1

Create a socket

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

**AF_INET**

**SOCK_STREAM**
**SOCK_DGRAM**

useful if some families have more than one protocol to support a given service

Conceptually similar to open BUT
- *open* creates a new reference to a possibly existing object
- *socket* creates a new instance of an object

7

## Step 2

Name the socket (assign address, port)

```
int error = bind(s, addr, addrlen)
```

socket

Address structure
`struct sockaddr*`

length of address structure

8

## Step 3a (server)

Set socket to be able to accept connections

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

socket

queue length for pending connections

9

## Step 3b (server)

Wait for a connection from client

```
int snew = accept(s, clntaddr, &clntalen)
```

socket

pointer to address structure

length of address structure

new socket for this session

s is only used for managing the queue of connection requests

10

## Step 3 (client)

Connect to server

```
int error = connect(s, svraddr, svraddrlen)
```

socket

address structure
`struct sockaddr*`

length of address structure

11

## Step 4

Exchange data

Connection-oriented I/O
- **`read`/`write`**
- **`recv`/`send`** *(extra flags)*

Connectionless I/O: no need for *connect, listen, accept*
- **`sendto`/`recvfrom`**
- **`sendmsg`/`recvmsg`**

12

## Step 5

Close connection

```
shutdown(s, how)
```

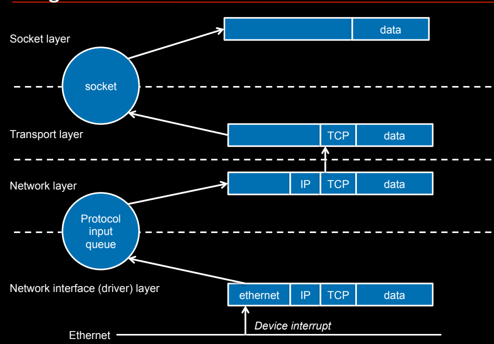
*how:*
0: can send but not receive
1: cannot send more data
2: cannot send or receive (=0+1)

13

## Socket Internals

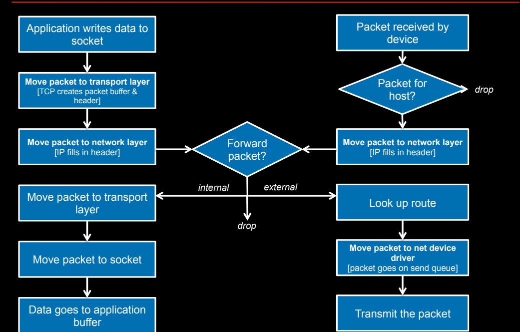
14

## Logical View




15

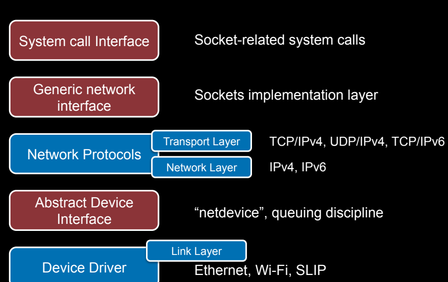
## Data Path




Adapted from http://kernelnewbies.org/Documents/LinuxIPNetworking

16

## OS Network Stack




17

## System call interfaces for accessing sockets

- Two ways to communicate with the network:
  - Socket-specific call (e.g., *socket, bind, shutdown*)
    - Directed to sys_socketcall (socket.c)
    - Goes to the target function
  - File descriptor call (e.g., *read, write, close*)
    - File descriptor ≡ socket
      - Sockets reside in the process's file table
    - Direct parallel of the VFS structure
      - A socket's *f_ops* field points to a set of functions for socket operations
- A `socket` structure acts as a queuing point for data being transmitted & received
  - A socket has send and receive queues associated with it
    - High & low watermarks to avoid resource exhaustion

18

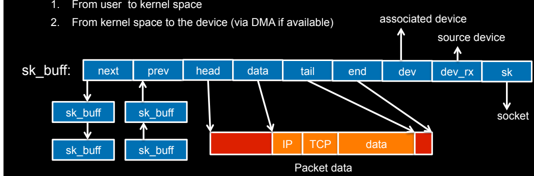
## Sockets layer

- All network communication takes place via a socket
- Two socket structures – one within another
  1. Generic sockets (*aka* BSD sockets) – `struct socket`
  2. Protocol-specific sockets (INET socket) – `struct sock`
- *`socket`* structure
  - Keeps all the state of a socket including the protocol and operations that can be performed on it
  - Some key members of the structure:
    - struct proto_ops *ops: protocol-specific functions that implement socket operations
      - Common functions to support a variety of protocols
      - TCP, UDP, IP, raw ethernet, other networks
    - struct inode *inode: points to in-memory inode associated with the socket
    - struct sock *sk: protocol-specific (e.g., INET) socket
      - E.g., this contains TCP/IP and UDP/IP specific data for an INET (Internet Address Domain) socket

19

## Socket Buffer: `struct sk_buff`

- Component for managing the data movement for sockets through the networking layers
  - Contains packet & state data for multiple layers of the protocol stack
- Don't waste time copying parameters & packet data from layer to layer of the network stack
- Data sits in a socket buffer (struct sk_buff)
- As we move through layers, data is only copied twice:
  1. From user to kernel space
  2. From kernel space to the device (via DMA if available)




20

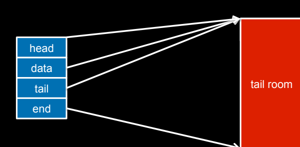
## Socket Buffer: `struct sk_buff`

- Each sent or received packet is associated with an sk_buff:
  - Packet data in *`data->`*, *`tail->`*
  - Total packet buffer in *`head->`*, *`end->`*
  - Header pointers (MAC, IP, TCP header, etc.)

  *Add or remove headers without reallocating memory*
- Identifies device structure (`net_device`)
  - `rx_dev`: points to the network device that received the packet
  - `dev`: identifies net device on which the buffer operates
    - If a routing decision has been made, this is the outbound interface
- Each socket (connection stream) is associated with a linked list of sk_buffs

21

## Keeping track of packet data



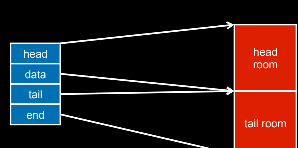

Allocate new socket buffer data

skb = alloc_skb(len, GFP_KERNEL);

No packet data: head = data = tail

22

## Keeping track of packet data



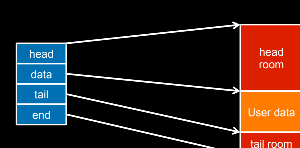

Make room for protocol headers.
skb_reserve(skb, header_len)

For IPv4, use sk->sk_prot->max_header

Data size is still 0

23

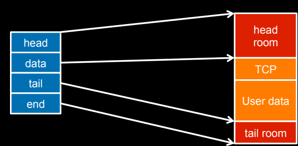
## Keeping track of packet data




Add user data

24

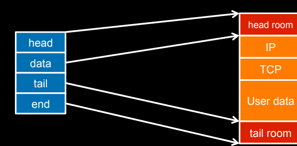
## Keeping track of packet data




Add TCP header

25

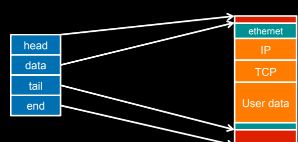
## Keeping track of packet data




Add IP header

26

## Keeping track of packet data




Add ethernet header

The outbound packet is complete!

27

## Network protocols

- Define the specific protocols available (e.g., TCP, UDP)
- Each networking protocol has a structure called *proto*
  - Associated with an "address family" (e.g., AF_INET)
  - Address family is specified by the programmer when creating the socket
  - Defines socket operations that can be performed from the sockets layer to the transport layer
    - *Close, connect, disconnect, accept, shutdown, sendmsg, recvmsg*, etc.
- Modular: one module may define one or more protocols
- Initialized & registered at startup
  - Initialization function: registers a family of protocols
  - The *register* function adds the protocol to the active protocol list
- Additional protocols can be added by calling *inet_register_proto_sw*

28

## Abstract device interface

- Layer above network device drivers
- Common set of functions for low-level network device drivers to operate with the higher-level protocol stack

29

## Abstract device interface

- Send a packet to a device
  - Send sk_buff from the protocol layer to a device
    - dev_queue_xmit function
    - enqueues an sk_buff for transmission to the underlying driver
    - Device is defined in sk_buff
      - Device structure contains a method hard_start_xmit: driver function for actually transmitting the data in the sk_buff
- Receive a packet from a device & send to protocol stack
  - Receive an sk_buff from a device
    - Driver receives a packet and places it into an allocated sk_buff
    - Sk_buff passed to the network layer with a call to netif_rx
    - Function enqueues the sk_buff to an upper-layer protocol's queue for processing through netif_rx_schedule

30

## Device drivers

- Drivers to access the network device
  - Examples: ethernet, 802.11b/g/n, SLIP
- Modular, like other devices
  - Described by `struct net_device`
- Initialization
  - Driver allocates a `net_device` structure
  - Initializes it with its functions
    - `dev->hard_start_xmit`: defines how to transmit a packet
      - Typically the packet is moved to a hardware queue
    - Register interrupt service routine
  - Calls *`register_netdevice`* to make the device available to the network stack

31

## Sending a message

- Write data to socket
- Socket calls appropriate *send* function (typically INET)
  - Send function verifies status of socket & protocol type
  - Sends data to transport layer routine (typically TCP or UDP)
- Transport layer
  - Creates a socket buffer (struct sk_buff)
  - Copies data from user's memory; fills in header (port #, options, checksum)
  - Passes buffer to the network layer (typically IP)
- Network layer
  - Fills in buffer with its own headers (IP address, options, checksum)
  - Look up destination route
  - IP layer may fragment data into multiple packets
  - Passes buffer to link layer: to destination route's device output function
- Link layer (part 1): move packet to the device's xmit queue
- Network driver (link layer part 2)
  - Wait for scheduler to run the device driver
  - Sends the link header
  - Transmit packet via DMA

32

## Routing

IP Network layer

Two structures:

1. Forwarding Information Base (FIB)
   Keeps track of details for every known route
2. Cache for destinations in use (hash table)
   If not found here then check FIB.

33

## Receiving a message – part 1

- Interrupt from network card: packet received
- Network driver – top half
  - Allocate new `sk_buff`
  - Get data from the hardware buffer into the `sk_buff` (DMA)
  - Call *netif_rx*, the generic network reception handler
    - This moves the sk_buff to protocol processing
    - When netif_rx returns, the service routine is finished
  - Repeat until no more packets in the device buffers
- If the packet queue is full, the packet is discarded
- *netif_rx* is called in the interrupt service routine
  - Must be quick. Main goal: queue the packet.

34

## Receiving a packet – part 2

Bottom half

- Kernel schedules work to go through pending packet queue
- Call *net_rx_action()*
  - Dequeue first `sk_buff` (packet)
  - Go through list of protocol handlers
    - Each protocol handler registers itself
    - Identifies which protocol type they handle
    - Go through each generic handler first
    - Then go through the *receive* function registered for the packet's protocol

35

## Receiving an IP packet – part 3

Network layer

- IP is a registered as a protocol handler for ETH_P_IP packets
  - IP handler will either route the packet, deliver locally, or discard
    - Send either to an outgoing queue (if routing) or to the transport layer
  - Look at protocol field inside the IP packet
    - Calls transport-lever handlers (*tcp_v4_rcv*, *udp_rcv*, *icmp_rcv*, …)
  - IP handler includes Netfilter hooks

36

## Receiving an IP packet – part 4

Transport layer

- Next stage (usually): *tcp_v4_rcv()* or *udp_rcv()*
  - Check for transport layer errors
  - Look for a socket that should receive this packet (match local & remote addresses and ports)
  - Call *tcp_v4_do_rcv*: passing it the `sk_buff` and socket (sock structure)
    - Adds `sk_buff` to the end of that socket's receive queue
    - The socket may have specific processing options defined
      - If so, apply them
- Wake up the process (ready state) if it was blocked on the socket

37

## Lots of Interrupts!

- Assume:
  - Non-jumbo maximum payload size: 1500 bytes
  - TCP acknowledgement (no data): 40 bytes
  - Median packet size: 413 bytes
- Assume a steady amount of network traffic at:
  - 1 Gbps: ~300 thousand packets/second
  - 100 Mbps: ~30 thousand packets/second
  - 10 Mbps: ~3 thousand packets /second
- Even 9000-byte jumbo frames
  - 1 Gbps: 14,000 packets per second → 14,000 interrupts/second

One interrupt per received packet

Network traffic can generate a LOT of interrupts!!

38

## Interrupt Mitigation: Linux NAPI

- Linux NAPI: "New API" (c. 2009)
- Avoid getting thousands of interrupts per second
  - Disable network device interrupts during high traffic
  - Re-enable interrupts when there are no more packets
- Also, packet throttling:
  - If we get more packets than we can process, leave them in the network card's buffer and let them get overwritten (same as dropping a packet)

39

## The End

40