**Networking**
Don Porter → Vyas Sekar
CSE 506

**Networking (2 parts)**
+
**Goals:**
++ Review networking basics
++ Discuss APIs
++ Trace how a packet gets from the network device to the application (and back)
++ Understand Receive livelock and NAPI

**Nomenclature**
+
Frame: hardware
+
Packet: IP
+
Segment: TCP/UDP
+
Message: Application

**TCP/IP Reality**
+
The OSI model is great for undergrad courses
+
TCP/IP (or UDP) is what the majority of programs use
+
Some random things (like networked disks) just use ethernet + some custom protocols

---

**Logical Diagram**

**4 to 7 layer diagram**
(from Understanding Linux Network Internals)
Ethernet (or 802.2 or 802.3)

+ All slight variations on a theme (3 different standards)
+ Simple packet layout:
  + Header: Type, source MAC address, destination MAC address, length, (and a few other fields)
  + Data block (payload)
  + Checksum
  + Higher-level protocols “nested” inside payload
  + “Unreliable” – no guarantee a packet will be delivered

Ethernet History

+ Originally designed for a shared wire (e.g., coax cable)
+ Each device listens to all traffic:
  + Hardware filters out traffic intended for other hosts
  + i.e., different destination MAC address
  + Can be put in “promiscuous” mode, and record everything (called a network sniffer)
+ Sending: Device hardware automatically detects if another device is sending at same time
  + Random back-off and retry

Early competition

+ Token-ring network: Devices passed a “token” around
  + Device with the token could send; all others listened
  + Like the “talking stick” in a kindergarten class
  + Send latencies increased proportionally to the number of hosts on the network
  + Even if they weren’t sending anything (still have to pass the token)
  + Ethernet has better latency under low contention and better throughput under high

Token ring

Source: [http://www.datacottage.com/tech/operation.htm](http://www.datacottage.com/tech/operation.htm)

Shared vs Switched

+ Modernethernetsare switched
+ What is a hub vs. a switch?
  + Both are a box that links multiple computers together
  + Hubs broadcast to all plugged-in computers (let computers filter traffic)
  + Switches track who is plugged in, only send to expected recipient
  + Makes sniffing harder 😐

Switched networks

Internet Protocol (IP)
+ 2 flavors: Version 4 and 6
+ Version 4 widely used in practice—today’s focus
+ Provides a network-wide unique device address (IP address)
+ This layer is responsible for routing data across multiple ethernet networks on the internet
+ Ethernet packet specifies its payload is IP
+ At each router, payload is copied into a new point-to-point ethernet frame and sent along

Transmission Control Protocol (TCP)
+ Higher-level protocol that layers end-to-end reliability, transparent to applications
+ Lots of packet acknowledgement messages, sequence numbers, automatic retry, etc.
+ Pretty complicated
+ Applications on a host are assigned a port number
+ A simple integer from 0-64k
+ Multiplexes many applications on one device
+ Ports below 1k reserved for privileged applications

User Datagram Protocol (UDP)
+ The simple alternative to TCP
+ None of the frills (no reliability guarantees)
+ Same port abstraction (1-64k)
+ But different ports
+ I.e., TCP port 22 isn’t the same port as UDP port 22

Some well-known ports
+ 80 –http
+ 22 –ssh
+ 53 –DNS
+ 25 –SMTP

Example (from Understanding Linux Network Internals)

Networking APIs
+ Programmers rarely create ethernet frames
+ Most applications use the socket abstraction
+ Stream of messages or bytes between two applications
+ Applications still specify: protocol (TCP vs. UDP), remote host address
+ Whether reads should return a stream of bytes or distinct messages
+ While many low-level details are abstracted, programmers must understand basics of low-level protocols
Sockets, cont.

+ One application is the server, or listens on a pre-determined port for new connections
+ The client connects to the server to create a message channel
+ The server accepts the connection, and they begin exchanging messages

Creation APIs

+ int socket(domain, type, protocol) – create a file handle representing the communication endpoint
  + Domain is usually AF_INET (IP4), many other choices
  + Type can be STREAM, DGRAM, RAW
  + Protocol – usually 0
+ int bind(fd, addr, addrlen) – bind this socket to a specific port, specified by addr
  + Can be INADDR_ANY (don't care what port)

Server APIs

+ int listen(fd, backlog) – Indicate you want incoming connections
  + Backlog is how many pending connections to buffer until dropped
+ int accept(fd, addr, len, flags) – Blocks until you get a connection, returns where from in addr
  + Return value is a new file descriptor for child
  + If you don't like it, just close the new fd

Client APIs

+ Both client and server create endpoints using socket()
+ Server uses bind, listen, accept
+ Client uses connect(fd, addr, addrlen) to connect to server
+ Once a connection is established:
  + Both use send/recv
  + Pretty self-explanatory calls

Client/server toy example

+ Quick demo ..
  + Client/server code from http://www.linuxhowtos.org/C_C++/socket.htm

Linux implementation

+ Sockets implemented in the kernel
  + So are TCP, UDP and IP
+ Benefits:
  + Application doesn't need to be scheduled for TCP ACKs, retransmit, etc.
  + Kernel trusted with correct delivery of packets
  + A single system call (i386):
    + sys_socketcall(call, args)
      + Has a sub-table of calls, like bind, connect, etc.
Plumbing

Each message is put in a sk_buff structure
Between socket/application and device, the sk_buff is passed through a stack of protocol handlers
These handlers update internal bookkeeping, wrap payload in their headers, etc.
At the bottom is the device itself, which sends/receives the packets

Efficient packet processing

Moving pointers is more efficient than removing headers
Appending headers is more efficient than re-copy

Recap from last class

Layering
L2, L3, L4 basics
Packet walkthrough

Networking Part Two

Don Porter ➔ Vyas Sekar
CSE 506
Interrupt handler

- “Top half” responsible to:
  - Allocate a buffer (sk_buff)
  - Copy received data into the buffer
  - Initialize a few fields
  - Call “bottom half” handler
  - In some cases, sk_buff can be pre-allocated, and network card can copy data in (DMA) before firing the interrupt
  - Lab 6 will follow this design

Quick review

- Why top and bottom halves?
  - To minimize time in an interrupt handler with other interrupts disabled
  - Gives kernel more scheduling flexibility
  - Simplifies service routines (defer complicated operations to a more general processing context)

Digression: Softirqs

- A hardware IRQ is the hardware interrupt line
- Also used for hardware “top half”
- Soft IRQ is the associated software “interrupt” handler
  - Or, “bottom half”
- How are these implemented in Linux?
  - Two canonical ways: Softirq and Tasklet
  - More general than just networking

Softirqs

- Kernel’s view: per-CPU work lists
- Tuples of <function, data>
- At the right time, call function(data)
  - Right time: Return from exceptions/interrupts/sys. calls
  - Also, each CPU has a kernel thread ksoftirqd_CPU# that processes pending requests
  - ksoftirqd is nice +19. What does that mean?
    - Lowest priority – only called when nothing else to do

Softirqs, cont.

- Device programmer’s view:
  - Only one instance of a softirq function will run on a CPU at a time
  - Doesn’t need to be reentrant
    - reentrant if it can be interrupted in the middle of its execution and then safely called again (“re-entered”) before its previous invocations complete execution
  - If interrupted, won’t be called again by interrupt handler
  - Subsequent calls enqueued
  - One instance can run on each CPU concurrently, though
  - Must use locks

Tasklets

- For the faint of heart (and faint of locking prowess)
  - Constrained to only run one at a time on any CPU
  - Useful for poorly synchronized device drivers
    - Say those that assume a single CPU in the 90’s
  - Downside: If your driver uses tasklets, and you have multiple devices of the same type—-the bottom halves of different devices execute serially
Softirq priorities

- Actually, there are 6 queues per CPU; processed in priority order:
  - HI_SOFTIRQ (high/first)
  - TIMER
  - NET TX
  - NET RX
  - SCSI
  - TASKLET (low/last)

Observation 1

- Devices can decide whether their bottom half is higher or lower priority than network traffic (HI or TASKLET)
  - Example: Video capture device may want to run its bottom half at HI, to ensure quality of service
  - Example: Printer may not care

Observation 2

- Transmit traffic prioritized above receive. Why?
  - The ability to send packets may stem the tide of incoming packets
  - Obviously eliminates retransmit requests based on timeout
  - Can also send “back-off” messages

Receive bottom half

- For each pending sk_buff:
  - Pass a copy to any taps (sniffers)
  - Do any MAC-layer processing, like bridging
  - Pass a copy to the appropriate protocol handler (e.g., IP)
    - Recur on protocol handler until you get to a port
      - Perform some handling transparently (filtering, ACK, retry)
    - If good, deliver to associated socket
    - If bad, drop

Socket delivery

- Once the bottom half/protocol handler moves a payload into a socket:
  - Check and see if the task is blocked on input for this socket
  - If so, wake it up
  - Read/recv system calls copy data into application

Socket sending

- Send/write system calls copy data into socket
  - Allocate sk_buff for data
  - Be sure to leave plenty of head and tail room!
  - System call does protocol handling during application’s timeslice
    - Note that receive handling done during ksoftirqd timeslice
  - Last protocol handler enqueues a softirq to transmit
Transmission
- Softirq can go ahead and invoke low-level driver to do a send
- Interrupt usually signals completion
  - Interrupt handler just frees the sk_buff

Switching gears
- We’ve seen the path network data takes through the kernel in some detail
- Now, let’s talk about how network drivers handle heavy loads

Our cup runneth over
- Suppose an interrupt fires every time a packet comes in
  - This takes $N$ ms to process the interrupt
  - What happens when packets arrive at a frequency approaching or exceeding $N$?
    - You spend all of your time handling interrupts!
  - Will the bottom halves for any of these packets get executed?
    - No. They are lower-priority than new packets

Receive livelock
- The condition that the system never makes progress because it spends all of its time starting to process new packets
- Real problem: Hard to prioritize other work over interrupts
- Principle: Better to process one packet to completion than to run just the top half on a million

Receive livelock in practice
- If you can’t process all incoming packets, you must drop some
- Principle: If you are going to drop some packets, better do it early!
- If you quit taking packets off of the network card, the network card will drop packets once its buffers get full

Source: Mogul & Ramakrishnan, TinCS 96
Idea

+ Under heavy load, disable the network card’s interrupts
+ Use polling instead
  + Ask if there is more work once you’ve done the first batch
  + This allows a packet to make it all the way through all of the bottom half processing, the application, and get a response back out
+ Ensuring some progress! Yay!

Why not poll all the time?

+ If polling is so great, why even bother with interrupts?
+ Latency: When incoming traffic is rare, we want high-priority, latency-sensitive applications to get their data ASAP

General insight

+ If the expected input rate is low, interrupts are better
+ When the expected input rate gets above a certain threshold, polling is better
+ Just need to figure out a way to dynamically switch between the two methods…

Why haven’t we seen this before?

+ Why don’t disks have this problem?
+ Inherently rate limited
+ If the CPU is bogged down processing previous disk requests, it can’t issue more
+ An external CPU can generate all sorts of network inputs

Pictorially..

Source: download.intel.com/design/control/PAPERS/323704.pdf

Linux NAPI

+ Or New API. Seriously.
+ Every driver provides a poll() method that does the low-level receive
  + Called in first step of softirq RX function
  + Top half just schedules poll() to do the receive as softirq
+ Can disable the interrupt under heavy loads; use timer interrupt to schedule a poll
+ Bonus: Some rare NICs have a timer; can fire an interrupt periodically, only if something to say!
NAPI

- Gives kernel control to throttle network input
- Slow adoption – means some measure of driver rewriting
- Backwards compatibility solution:
  - Old top half still creates sk_buffs and puts them in a queue
  - Queue assigned to a fake “backlog” device
  - Backlog poll device is scheduled by NAPI softirq
  - Interrupts can still be disabled

NAPI Summary

- Too much input is a real problem
- NAPI lets kernel throttle interrupts until current packets processed
- Softirq priorities let some devices run their bottom halves before net TX/RX
  - Net TX handled before RX

General summary

- Networking basics and APIs
- Idea of plumbing from socket to driver
  - Through protocol handlers and softirq poll methods
- NAPI and input throttling