* NETWORKING

handout posted on class web site

how is networking different (that, say, file systems):
- data comes from outside, and is unverified (even malicious)
- no full control, on incoming data, or once data is sent out (can't tell if data gets to its destination)
- "best effort": try best to recv/send data, but if can't, drop packets
- much wider range of possible latencies and data rates (than local I/O devices)

Layers:
- at top: VFS layer, e.g., read(2) and write(2) on sockets.

- also at top: Socket API, same level as VFS: socket specific syscalls like bind, listen, socket, select, poll, send[to], /recv[from], etc.

- socket layer: abstraction of network handles (includes TCP, but not just TCP)

- protocol drivers: more specific implementations. e.g., TCP/IP, UDP, Appletalk, NetBios, etc.

- queue discipline: traffic shaping or QoS control. Applied to all protocols.

- device drivers for network interface cards (NICs).

- many more layers "inside" each of those layers, including virtual NICs, etc. Also "TCP/IP" is actually TCP layer on top of IP, on top of Ethernet.

Layering:

1. pros: allows to develop abstractions independently, add features or implement new instances, at the "right" layer. And just have to focus on that layer's APIs above and below. Simpler development of code.

2. cons: for complete separation of layers, pass data up/down layers by copying it. Problem: too slow to copy data, and wastes memory.

Today, using a hybrid approach. In linux, we use a shared "SKbuff services". A shared pool/cache of objects. No data copies, but pass by ref or pointer. Faster to move data across layers.

SKbuff services:
- SKbuff: Simple Kernel buffer (same as "mbuf" in BSD/Solaris)
- a custom mem allocator for networking
- has tail/headroom: easy to add/remove headers when going up/down the layers.

SKbuff APIs:
- alloc_skbuf, free
- un/lock
- split an skbuf in two or more
- merge multiple skbufs into one

* Device drivers (receive and send data).

User vs. Kernel separation so far
Actually: User -> kernel -> hardware

NIC: hardware
- sits on a bus (PCI), and communicates with rest of computer
- Ethernet ports to talk to outside world
- has a processor
- memory (DRAM)
- a program running (firmware)
=> a NIC is a like a mini computer with a mind of its own.
=> every other peripheral device is also its own "mini computer"
=> so even on a single core CPU, you still have a "concurrent/distributed system"

NIC reading data:

1. NIC listens on wire (RJ45 Ethernet) for packets that belong to it
- look for data that is preceded by your own MAC (Ethernet) address (e.g., 48 bit)
2. If found bits that appear to belong to this NIC, then start to read them into my own memory buffers.
- if no free NIC buffer space, then "drop" packet (i.e., never receive it in the first place). Can be full b/c of previously received packets.
- if packet never received, then sender wouldn't get an ACK, then try to retransmit some time later.
- in TCP, you exponentially back off each time you don't get an ACK: senders essentially throttle themselves.
3. If have space in NIC mem, read entire packet into NIC ram and store it there. Maybe do basic validity checks.
4. NIC has to give packet to OS quickly, so it doesn't take up NIC ram for too long (don't want NIC ram to be full)
5. NIC would interrupt the main CPU, actual h/w interrupt.
---
inside OS
6. OS kernel will stop what's doing, save state, and invoke the interrupt handle routine for "networking".
7. networking interrupt handler: take packet from NIC and store in an SKbuf
7a. grab a free/unused skbuf (faster than trying to alloc a new skbuf)
7b. copy data from NIC to skbuf (fast)
- use actual I/O instructions (slow)
- or use DMA (direct mem access), which is async too.
7c.

while this interrupt handler runs, other interrupts are BLOCKED
- ihandler must finish work as quickly as possible!