Ultrafast userspace packet blasting: Why should the kernel have all the fun?

Cody Harris

<2022-04-14 Thu>

Outline

But Why?

- Writing kernel code is really hard.
  - Small mistakes crash the whole system.
  - Libraries are different, less featured.
- Debugging kernel code is even harder than writing it.
- Userspace code is just as fast as the kernel on a properly-configured system.
  - Actualy, userspace code can be even faster than the kernel –it can focus on just one thing.

What's Wrong with the Kernel Network Stack?

- Nothing is wrong with the kernel network stack, but sometimes you run up against its limits.
  - Want to do weird things it wasn't meant to do
  - Need higher performance than it was designed to provide (eg. use a commodity server as a network device)
  - The kernel network stack is mostly interrupt-driven
    - A reasonable choice for a general-purpose system!
    - But interrupt-driven is a compromise when net throughput is highest priority
    - Polling (repeatedly checking) network card is faster than waiting to be told that packets have arrived

But How? (Applies to any High-Performance System)

- Configure the system for high performance.
  - Disable features that throttle back the CPU clock
  - Configure huge pages to improve memory performance
  - Disable SMI interrupts from the BIOS
- Configure the kernel so it's out of the way
  - Pin your processes to specific cores
  - Configure the kernel to never schedule other processes (inc. itself) on your cores (via isolcpus)
  - Go tickless, disable RCU callbacks so the kernel doesn't interrupt
  - You now have the whole core to yourself!
- Don't use system calls in high-performance code
  - Just use counters, no logging in normal operation
  - Ok, major errors and RPC can go to a shared-memory ring-buffer, but NO printf allowed!

But How? (Network Specifics)

- Get the kernel network stack out of the way
- Unbind the network card from the kernel network driver
- Bind the network card to a userspace i/o driver
  - This memory-maps the network card's address space into userspace. You can talk to the card just by reading and writing to memory. PCIe subsystem handles the bus transparently. Feels like embedded system programming.
- Use a userspace driver (just some C code) to talk to the network card with a nicer API
- One downside –you no longer have a network stack

But How? (Network Card Specifics)

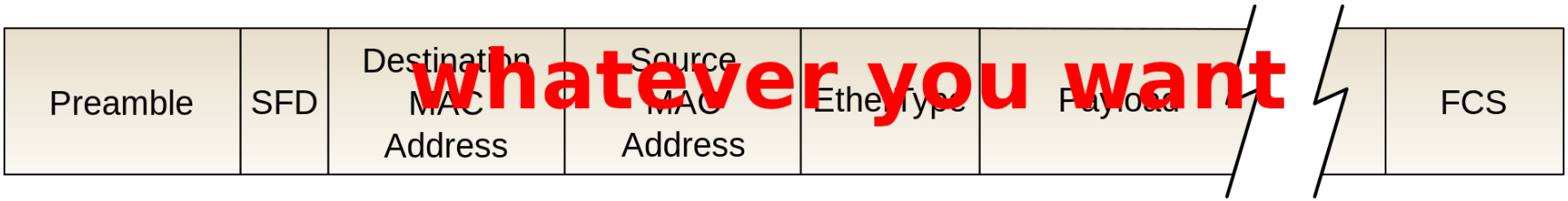
- Configure network card checksum offloads to save some work
  - Cards have logic to verify or generate:
    - Ethernet frame CRC
    - IP header checksum
    - UDP or TCP checksums
- Split the network card into multiple queues
  - A modern high-performance server CPU can only forward about 1 Gbps of traffic per core.
  - Eg. make your 10GbE card act like 20x 0.5 Gbps pipes.
    - Gives you headroom to actually do something with the packets!
- Set up memory for the network card
  - Message buffers –where packet data goes
  - Descriptors –a card-specific data structure understood by the network card. One descriptor per packet. These describe the DMA setup and tells the network card where the message buffers are.
  - Descriptors are in a ring buffer (a linked list where the tail points back to the head)

The Data Plane Development Kit (DPDK)

- Open source project started by Intel
- Provides libraries to handle the really tedious parts of everything just mentioned
- A hardware abstraction layer and the userspace network card drivers.
  - They've implemented userspace poll-mode drivers (PMDs) for popular network cards
    - (In contrast to more common interrupt-driven drivers)
- Contains useful lockless data structures

No Gods No Masters

- You are no longer bound by the rules of society
- Any frame/packet headers you want, including addresses:



  - Any ethernet src/dst (pretend to be many computers for the lulz)
  - Anything layer-3 payload want inside the ethernet frame. IP is nice if you want your packets to get onto the internet, but there are many interesting layer-3 protocols: https://en.wikipedia.org/wiki/EtherType has some interesting ideas for protocols to implement:
    - GOOSE, SVE, GSE –SCADA and industrial control!
    - IPX, DECnet –Network with vintage computers, bridge them to the internet?
    - Precision Time Protocol (PTP) client or server?

No Gods No Masters (Continued)

- Make IP packets, send them to the internet...
  - Fake the source address, see what happens
  - Ping everybody on the internet: $\frac{3.7 \cdot 10^{9} \text{addresses}}{1.49 \cdot 10^{6} \text{packets/second}}$
    - 2500 seconds to ping every public ip4 address with a 1 Gbps link!
  - Implement a scary fast port scanner?

Slightly More Practical Ideas

- Implement NAT (network address translation) to bridge networks.
- Network wiretap and deep-packet inspection device that mirrors packet off to some secret cable :nsa:
- Implement a high-performance network tunnel endpoint, eg. IPSec VPN
  - Network virtualization! Become your own cloud provider!
- Build a firewall, switch, router, etc. in software

Download and install DPDK

- Get it from dpdk.org
- My opinion: work from a release tag, not HEAD, for the best experience

Configure machine for hugepages

What is a page?

- Basically a chunk of RAM. 4KB in modern Linux machines.
- Why divide memory up into pages? Virtual memory. Outside scope of this talk, but a fascinating topic.

What are hugepages?

- Like pages, but huger. 2MB or 1GB chunks (configurable) rather than 4KB.
- Why would I do use hugepages? Hardware reasons. Sort-of out of scope.

But Hardware is Cool, so if there's Time:

- The translation lookaside buffer (TLB) is a hardware component in the CPU's virtual memory system.
- TLB speeds up virtual memory via table of mappings between virtual addresses and physical addresses.
  - Pages cached in the TLB can be accessed immediately.
  - Pages that aren't in the TLB can still be accessed, but there's a performance impact because the kernel needs to step in and help (context switch).
- This all happens transparently to a userspace program, but there is a performance impact to TLB misses.
- Bigger pages -> fewer pages. fewer pages -> fewer opportunities for tlb misses. tlb miss requires context switch to kernel, which is slow. Default pages are 4KB. Huge pages are 2MB or 1GB on modern CPUs

Configuring Hugepages

- First, requires build-time support from kernel
- I had to edit /etc/sysctl.conf and add $\text{vm.nr}_{\text{hugepages}} = \ldots$ (set a different option if you want to use 1G hugepages) then reboot
- You can check for success in /proc/meminfo

```
# grep Huge /proc/meminfo
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     256
HugePages_Free:      245
HugePages_Rsvd:      125
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          524288 kB
```

Bind network card to a vfio driver

- What? The network card "owns" a section of memory. In non-DPDK world, the kernel driver reads and writes this memory to make it do network things. In DPDK world, we need to read/write this memory from userspace. Basically, the vfio driver mmaps the network card so we can mess with the hardware directly from userspace.

How To Do It

```
# sudo modprobe uio_pci_generic # load userspace PCI driver
# ./usertools/dpdk-devbind.py -s # list the current bindings
Network devices using kernel driver
==================================
0000:00:1f.6 [...] drv=e1000e unused=uio_pci_generic

# ./usertools/dpdk-devbind.py -u 00:1f.6 # unbind NIC driver

# bind uio to ethernet NIC instead of kernel network driver
# ./usertools/dpdk-devbind.py -b uio_pci_generic 00:1f.6

#./usertools/dpdk-devbind.py -s # list bindings again
Network devices using DPDK-compatible driver
============================================
0000:00:1f.6 [...] drv=uio_pci_generic # woohoo
```