L41 - Lecture 5: The Network Stack (1)

Dr Robert N. M. Watson

19 January 2016

Reminder: where we left off in Michaelmas Term

Long, long ago, but in a galaxy not so far away:

- Lecture 3: The Process Model (1)
- Lecture 4: The Process Model (2)

- Lab 1: I/O performance

- Lab 2: IPC buffer size and probe effect
- Lab 3: Micro-architectural effects of IPC

Explored several implied (and rejected) hypotheses:

- Larger I/O and IPC buffer sizes amortize system-call overheads
- Micro-architecture is irrelevant
- The probe effect doesn't matter in real workloads

This time: Introduction to the Network Stack

Rapid tour across hardware and software:

1. Networking and the sockets API
2. Network-stack design principles: 1980s and today
3. Memory flow across hardware and software
4. Network-stack construction and work flows
5. Recent network-stack research

Networking: a key OS function

- Communication between distributed computer systems
  - *Local-area networking* (LANs) and *wide-area networking* (WANs)
- A network stack provides:
  - Sockets API and extensions
  - Interoperable, feature-rich, high-performance protocol implementations (e.g., IPv4, IPv6, ICMP, UDP, TCP, SCTP, ...)
  - Device drivers for Network Interface Cards (NICs)
  - Monitoring and management interfaces (BPF, `ioctl`)
  - Plethora of support libraries (e.g., DNS)
- Dramatic changes over 30 years:
  - 1980s: Early packet-switched networks, UDP+TCP/IP, Ethernet
  - 1990s: Large-scale migration to IP; Ethernet VLANs
  - 2000s: 1-Gigabit/s, then 10-Gigabit/s Ethernet; 802.11, GSM data
  - 2010s: Large-scale deployment of IPv6; 40/100-Gigabit/s Ethernet
- Vanishing technologies: UUCP, IPX/SPX, ATM, token ring, SLIP, ...

The Berkeley Sockets API (1983)

```
close()
read()
write()
...

accept()
bind()
connect()
getsockopt()
listen()
recv()
select()
send()
setsockopt()
socket()
...
```

- *The Design and Implementation of the 4.3BSD Operating System* (although appeared in 4.2)
- Now universal TCP/IP API (POSIX, Windows, ...)
- Kernel-resident network stack serves userspace networking applications via system calls
- Reuse file-descriptor abstraction
  - Same API for local and distributed IPC
  - Simple, synchronous, copying semantics
  - Blocking/non-blocking I/O, `select()`
- Multi-protocol (e.g., IPv4, IPv6, ISO, ...)
  - TCP-focused but not TCP-specific
  - Cross-protocol abstractions and libraries
  - Protocol-specific implementations
  - “Portable” applications

BSD network-stack principles (1980s-1990s)

A framework for **multi-protocol**, **packet-oriented** network research:

- Object-oriented: multiple protocols, multiple socket types, one API
  - **Protocol-independent**: streams vs. datagrams, sockets, socket buffers, socket addresses, network interfaces, routing table, packets
  - **Protocol-specific**: connection lists, address/routing specialization, routing, transport protocol itself – encapsulation, decapsulation, etc

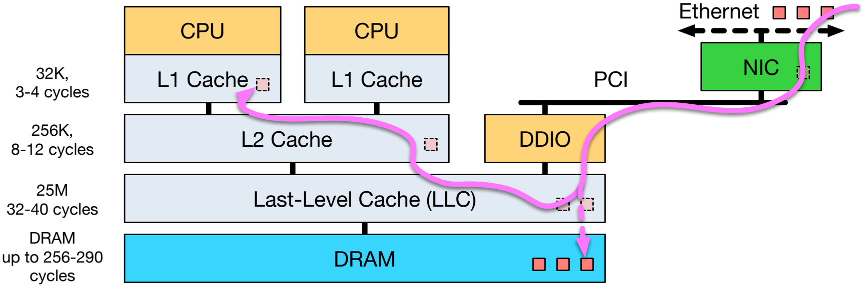
- Fundamentally packet-oriented design:
  - Packets and packet queueing as fundamental primitives
  - If there is a failure (overload, corruption) drop the packet
  - Work hard to maintain packet source ordering
  - Differentiate ‘receive’ from ‘deliver’ and ‘send’ from ‘transmit’
  - Heavy focus on TCP functionality and performance
  - Middle-node (forwarding), not just edge-node (I/O), functionality
  - High-performance packet capture: Berkeley Packet Filter (BPF)

FreeBSD network-stack principles (1990s-2010s)

All of the 1980s features and also ...

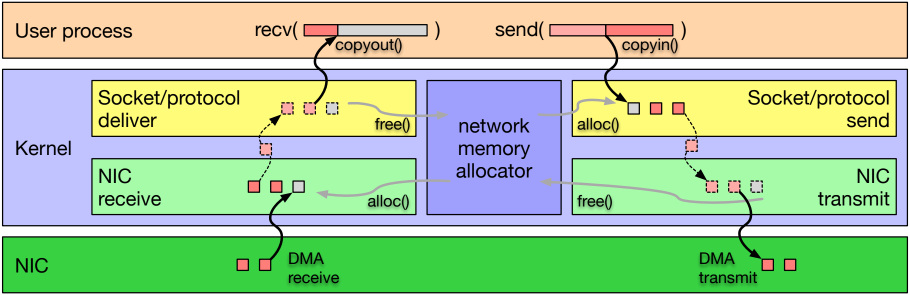
- Hardware:
  - Multi-processor scalability
  - NIC offload features (checksums, TSO/LRO, full TCP)
  - Multi-queue network cards with load balancing/flow direction
  - Performance to 10s or 100s of Gigabit/s
  - Wireless networking
- Protocols:
  - Dual-IPv4/IPv6
  - Security/privacy: firewalls, IPsec, ...
- Software model:
  - Flexible memory model integrates with VM for zero-copy
  - Network-stack virtualisation
  - Userspace networking via `netmap`

Memory flow in hardware



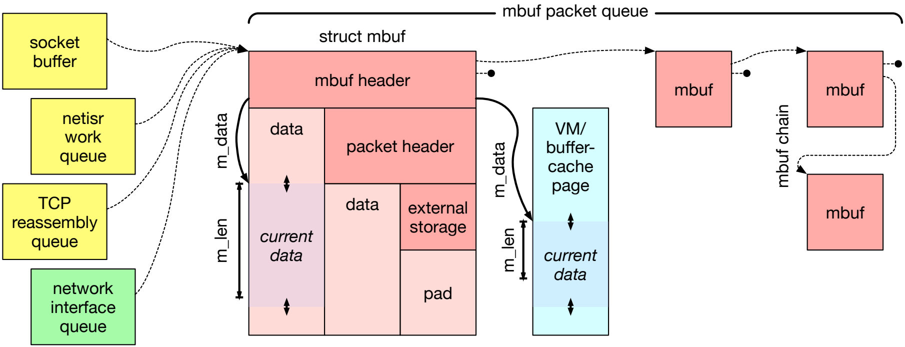
- Key idea: *follow the memory*
- Historically, memory copying avoided due to CPU cost
- Today, memory copying avoided due to cache footprint
- Recent Intel CPUs push and pull DMA via the LLC (“DDIO”)
- NB: if we differentiate ‘send’ and ‘transmit’, is this a good idea?

Memory flow in software



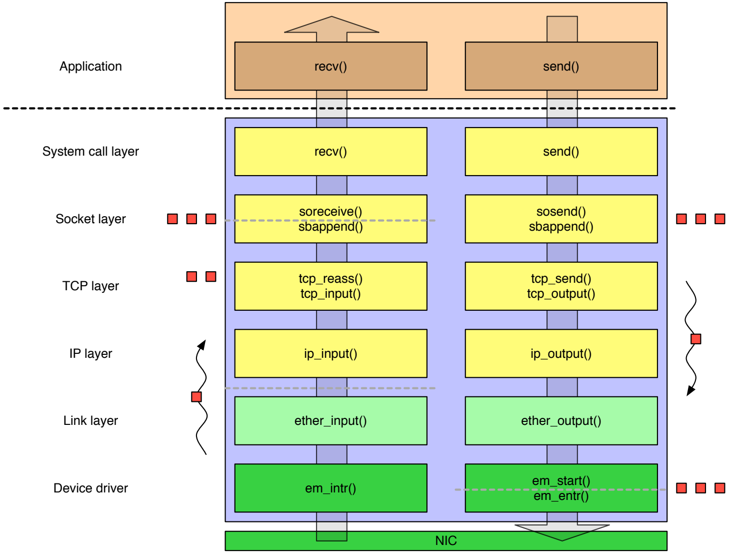
- Socket API implies one copy to/from user memory
  - Historically, zero-copy VM tricks for socket API ineffective
- Network buffers cycle through the slab allocator
  - Receive: allocate in NIC driver, free in socket layer
  - Transmit: allocate in socket layer, free in NIC driver
- DMA performs second copy; can affect cache/memory bandwidth
  - NB: what if packet-buffer working set is larger than the cache?

The `mbuf` abstraction

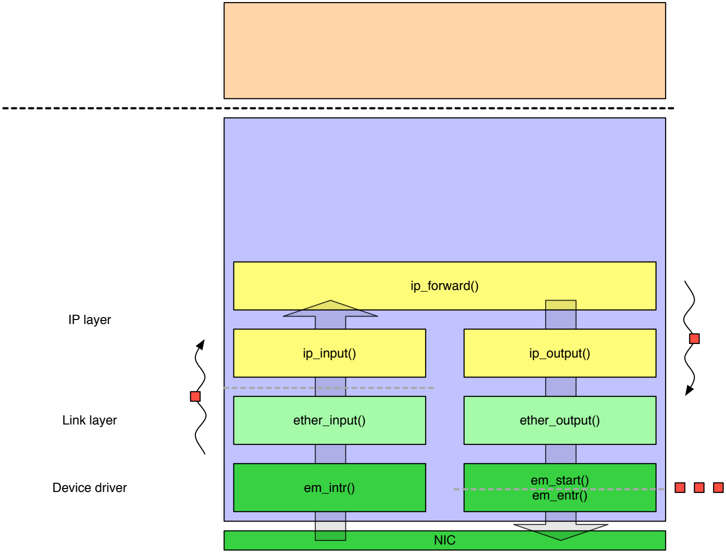


- `mbuf` chains represent in-flight packets, streams, etc.
  - Ops: alloc, free, prepend, append, truncate, enqueue, dequeue
  - Internal or external data buffer (e.g., VM page)
  - Bi-modal packet-size distribution; e.g., TCP ACKs vs. data
- Unit of **work allocation and distribution** throughout the stack
- Similar structures in other OSes – e.g., `skbuff` in Linux

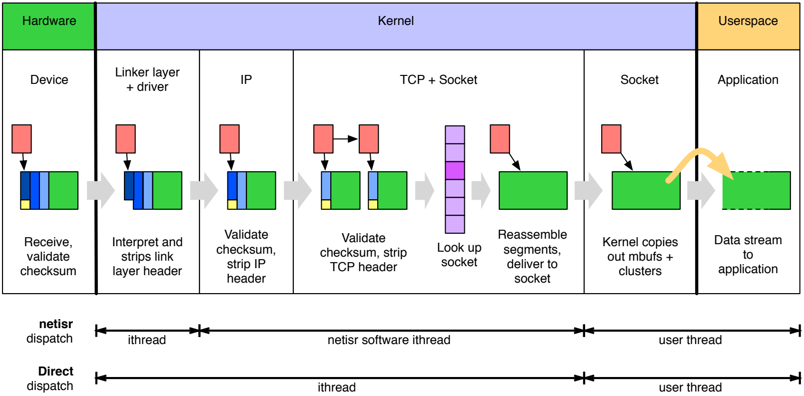
Local send/receive paths in the network stack



Forwarding path in the network stack

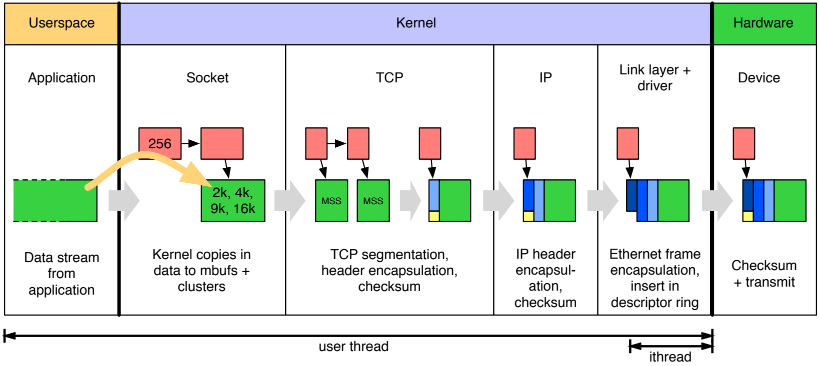


Work dispatch: input path



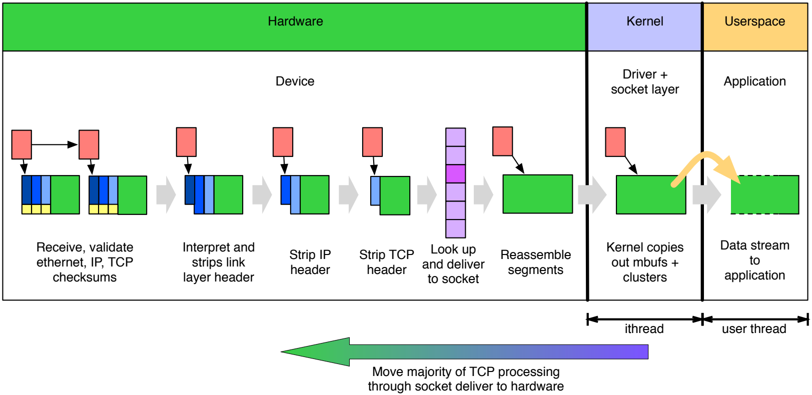
- Deferred dispatch - *ithread -> netisr thread -> user thread*
- Now: direct dispatch - *ithread -> user thread*
  - Pros: reduced latency, better cache locality, drop overload early
  - Cons: reduced parallelism and work placement opportunities

Work dispatch: output path



- Fewer deferred dispatch opportunities implemented
- Gradual shift of work from software to hardware
  - Checksum calculation, segmentation, ...

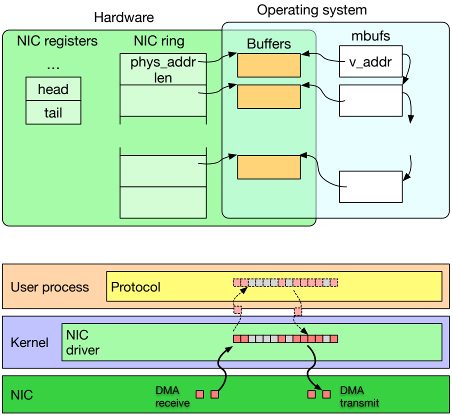
Work dispatch: TOE input path



- Full TCP offload
  - Kernel provides socket buffers and resource allocation
  - Remainder, including state, retransmits, reassembly, etc, in NIC
- But: Two network stacks? Less flexible/updateable structure?
- Better with an explicit HW/SW architecture – e.g., Microsoft Chimney?

netmap: a novel framework for fast packet I/O

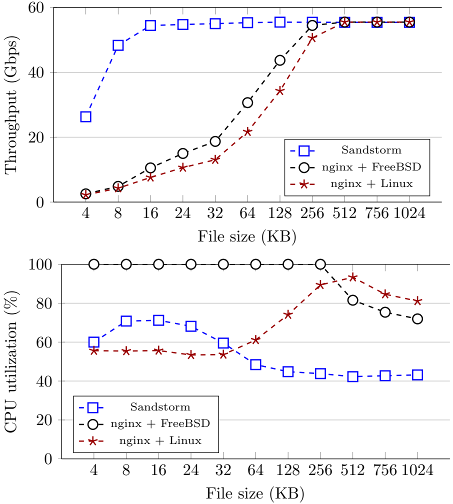
Luigi Rizzo, USENIX ATC 2012 (best paper).



- Map NIC buffers directly into user process memory
- Zero copy to/from app.
- System calls initiate DMA, block for NIC events
- Packets can be reinjected into normal stack
- Ships in FreeBSD, patch available for Linux
- Userspace network stack can be **specialized** to task (e.g., packet forwarding)

Network Stack Specialization for Performance

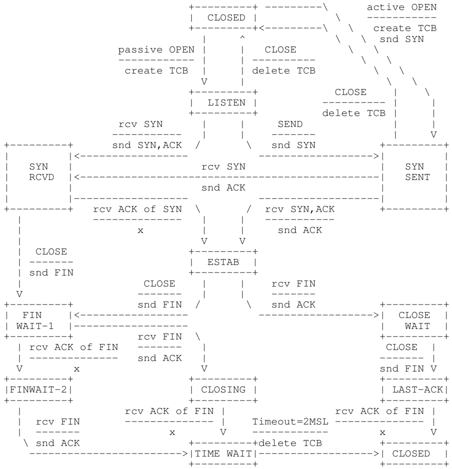
Ilias Marinos, Robert N.M. Watson, Mark Handley, SIGCOMM 2014.



- 30 years since network-stack design developed
- Massive changes in architecture, micro-architecture, memory, buses, NICs
  - Optimising compilers
  - Cache-centered CPUs
  - Multiprocessing, NUMA
  - DMA, multiqueue
  - 10 Gigabit/s Ethernet
- Revisit fundamentals through clean-slate stack

Next time: Socket buffers and TCP

September 1981 Transmission Control Protocol
Functional Specification



TCP Connection State Diagram
Figure 6.

- McKusick, et al: Chapter 14 (*Transport-Layer Protocols*)
- Transmission Control Protocol (TCP)
- TCP implementation
  - Buffers and input processing
  - Parallelism and performance
  - DoS resistance
- The final two labs