Fast Packet Processing

Lecture 11
Srinivas Narayana

http://www.cs.rutgers.edu/~sn624/553-S23

Some slides were adapted from those of Gianni Antichi
Context: Networking for Internet services

Data center transport

Fast packet processing

Interconnect
Packet processing on Linux
Modern NICs and architectures can also do direct cache access (DCA)
Interrupt mitigation

• Interrupt processing at high rate and priority prevents any other part of the system from progressing (receive livelock).
• Mitigations:
  • (1) Interrupt coalescing:
    • Wait (at NIC) for more packets or a timeout until interrupting
  • (2) Polling to schedule work across different sources of processing
    • Avoid preemption
  • (3) CPU or packet quotas on polling to ensure other parts of the system (user space app) can progress
    • Re-enable interrupts if there is less work than allotted quota
Revisiting network I/O APIs: The netmap framework. CACM'12
Allocate packet data structures in memory (sk_buff, mbufs, …)

(Optional) receive packet steering
Socket buffers

• Allocate in arbitrary chunks (multiples of 64 bytes)
• Support arbitrary packet sizes, fragments, deferred processing
Other things that happen afterward

- Netfilter: tracking TCP connection state, firewalling, NAT, …
- IP protocol processing: routing
- Transport processing (UDP/TCP protocol layer)

Some stateless, per-packet work can be done by the NIC:
  - TSO: TCP segmentation offload
  - LRO: Large Receive Offload (also applicable in software)
  - IP checksum
  - Ethernet CRC computation
FreeBSD `sendto()` code path

Overheads are sprinkled throughout the packet processing stack.

<table>
<thead>
<tr>
<th>File</th>
<th>Function/description</th>
<th>time ns</th>
<th>delta ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>user program</td>
<td><code>sendto</code> system call</td>
<td>8</td>
<td>96</td>
</tr>
<tr>
<td>uipc_syscalls.c</td>
<td><code>sys_sendto</code></td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>uipc_syscalls.c</td>
<td><code>sendit</code></td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>uipc_syscalls.c</td>
<td><code>kern_sendit</code></td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>uipc_socket.c</td>
<td><code>sosend</code></td>
<td>146</td>
<td>137</td>
</tr>
<tr>
<td>uipc_socket.c</td>
<td><code>sosend_dgram</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sockbuf locking, mbuf allocation, copyin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>udp_usrreq.c</td>
<td><code>udp_send</code></td>
<td>273</td>
<td>57</td>
</tr>
<tr>
<td>udp_usrreq.c</td>
<td><code>udp_output</code></td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>ip_output.c</td>
<td><code>ip_output</code> route lookup, ip header setup</td>
<td>330</td>
<td>198</td>
</tr>
<tr>
<td>if_ETHERSUBR.C</td>
<td><code>ether_output</code> MAC header lookup and copy, loopback</td>
<td>528</td>
<td>162</td>
</tr>
<tr>
<td>if_ETHERSUBR.C</td>
<td><code>ether_output_frame</code></td>
<td>690</td>
<td></td>
</tr>
<tr>
<td>ixgbe.c</td>
<td><code>ixgbe_mq_start</code></td>
<td>698</td>
<td>220</td>
</tr>
<tr>
<td>ixgbe.c</td>
<td><code>ixgbe_mq_start_locked</code></td>
<td>720</td>
<td></td>
</tr>
<tr>
<td>ixgbe.c</td>
<td><code>ixgbe_xmit</code> mbuf mangling, device programming</td>
<td>730</td>
<td></td>
</tr>
<tr>
<td>–</td>
<td><code>on wire</code></td>
<td>950</td>
<td></td>
</tr>
</tbody>
</table>
(1) Shared memory: avoid per-byte costs

- Remove user-kernel data copies
- Other systems use similar ideas:
  - Finish processing entirely within the kernel (e.g., click-kernel, eBPF)
  - Expose kernel buffers directly to user space (PF_RING)
(2) Data representation: pre-allocated fixed size buffers and rings

- Avoid per-byte costs by pre-allocating chunks of a fixed size (max packet size)
- No allocation and freeing mbuf/sk_buff at run time
• Validate netmap ring inputs provided by applications
• System call still needed to copy netmap ring descriptor to NIC ring descriptor (per-packet operation)
• Some systems avoid even this (DPDK, PF-RING, Solarflare openonload) by having apps directly program NIC rings (security & fault implications)
(4) Amortize system calls by batching

- Notify the kernel about packets written for transmission or available for reception

\[
\text{ioctl}(...) \text{, NIOCTXSYNC)
\]

\[
\text{ioctl}(...) \text{, NIOCRXSYNC)
\]

\[
\text{select}()/\text{poll}()
\]
Pkt gen

- Associate shared buffers with fd’s
- Poll file descriptor
- Walk through the netmap ring to identify available packet buffers. Write and notify
- Poll automatically synchronizes rings. No more system calls needed

```c
fds.fd = open("/dev/netmap", O_RDWR);
strcpy(nmr.nm_name, "ix0");
ioctl(fds.fd, NIOCREG, &nmr);
p = mmap(0, nmr.memsiz, fds.fd);
nifp = NETMAP_IF(p, nmr.offset);
fds.events = POLLOUT;
for (;;) {
poll(fds, 1, -1);
  for (r = 0; r < nmr.num_queues; r++) {
    ring = NETMAP_TXRING(nifp, r);
    while (ring->avail-- > 0) {
      i = ring->cur;
      buf = NETMAP_BUF(ring, ring->slot[i].buf_index);
      // ... store the payload into buf ...
      ring->slot[i].len = ... // set packet length
      ring->cur = NETMAP_NEXT(ring, i);
    }
  }
}
```
DPDK basic forwarding

```c
for (;;) {
    /*
    * Receive packets on a port and forward them on the paired
    * port. The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.
    */
    RTE_ETH_FOREACH_DEV(port) {

        /* Get burst of RX packets, from first port of pair. */
        struct rte_mbuf *bufs[BURST_SIZE];
        const uint16_t nb_rx = rte_eth_rx_burst(port, 0, 
                                                bufs, BURST_SIZE);

        if (unlikely(nb_rx == 0))
            continue;

        /* Send burst of RX packets, to second port of pair. */
        const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0, 
                                                bufs, nb_rx);

        /* Free any unsent packets. */
        if (unlikely(nb_tx < nb_rx)) {
            uint16_t buf;
            for (buf = nb_tx; buf < nb_rx; buf++)
                rte_pktmbuf_free(bufs[buf]);
        }
    }
}
```
Forwarding between two interfaces

- Move descriptors, no data copies

```
...  
src = &src_nifp->slot[i]; /* locate src and dst slots */
dst = &dst_nifp->slot[j];
/* swap the buffers */
tmp = dst->buf_index;
dst->buf_index = src->buf_index;
src->buf_index = tmp;
/* update length and flags */
dst->len = src->len;
/* tell kernel to update addresses in the NIC rings */
dst->flags = src->flags = BUF_CHANGED;
...
```
Performance (pkt gen throughput)

- Vary clock rate to make the workload CPU bound
Varying packet size
Performance with batching

![Graph showing performance with batching]
Outlook: fast packet processing

• Get rid of software if you can
• Application-kernel API change: application must be modified
• Device drivers must often be modified
• Utilities in the host networking stack?
  • Libpcap, Netfilter, Routing, Socket lookup/packet demuxing?
• Multitenancy: serious implications to weakening fault isolation
• Can we get isolation with efficiency?