Experiments With Highly Parallel Network Stack Processing

Robert N. M. Watson

17 May 2007

FreeBSD Developer Summit
BSDCan 2007
Introduction

• MPSAFE network stack in FreeBSD 5.3
  – November 2004

• Network stack safe to execute concurrently
  – Parallelism – many processors at a time
  – Preemption – low latency context switching
  – Direct dispatch – from interrupt thread context

• Opportunities are limited due to work model
  – Threads represent potential parallelism
  – So work must occur in multiple threads
## Socket to Interface Code Flow

<table>
<thead>
<tr>
<th>Layer</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>System call and socket</td>
<td>- <code>kern_send()</code>&lt;br&gt;- <code>sosend()</code>&lt;br&gt;- <code>sbappend()</code>&lt;br&gt;- <code>kern_recv()</code>&lt;br&gt;- <code>soreceive()</code>&lt;br&gt;- <code>sbappend()</code></td>
</tr>
<tr>
<td>TCP</td>
<td>- <code>tcp_send()</code>&lt;br&gt;- <code>tcp_output()</code>&lt;br&gt;- <code>tcp_reass()</code>&lt;br&gt;- <code>tcp_input()</code></td>
</tr>
<tr>
<td>IP</td>
<td>- <code>ip_output()</code>&lt;br&gt;- <code>ip_input()</code></td>
</tr>
<tr>
<td>Link Layer</td>
<td>- <code>ether_output()</code>&lt;br&gt;- <code>ether_input()</code>&lt;br&gt;- <code>em_start()</code>&lt;br&gt;- <code>em_intr()</code></td>
</tr>
</tbody>
</table>
Transmit

netblast

- sosend()
- send()
- ip_output()

em0 ithread

- udp_output()
- send() returns
- em_start()
- em_intr() preempts
- em_intr() returns
- em_clean_transmit_intr()
Input Path Parallelism

• Does have native concurrency
  – Work processed in ithread, optionally netisr, and user thread

• Cost of context switching measurable
  – Default in 7.x is direct dispatch as frequently faster

• Ordered with respect to source
  – Lack of parallelism for high bandwidth sources
Deferred netisr

recv()

netreceive blocks

netisr_dispatch()

swi_net()

ip_input()

em_intr() preempts

sbappend()

emalloc()

sowakeup()

recv() returns

netreceive wakes up

em_intr() returns

 ether_input()

idle

em0 ithread

netisr

netreceive
Direct netisr

- **recv()**
  - netreceive blocks
  - netisr_dispatch()
  - *em_intr() preempts*

- **netreceive wakes up**
  - recv() returns
  - *ip_input()*
  - *sbappend() sowakeup()*

- **em0 itherd**
  - *em_process_receive_interrupts()*
  - ether_input()
  - udp_input()
netisr2

• Function of netisr after direct dispatch
  – Encapsulation/decapsulation, loopback traffic

• Netisr2 re-implement netisr infrastructure
  – Per-cpu kernel worker threads
  – Maintain dispatch/queue distinction

• How to balance ordering and parallelism?
  – For direct dispatch, source ordering
  – For remote source input, per-protocol affinity lookup
  – For locally source, affinity passed down stack
TCP Input Parallelism

- Serious bottleneck for TCP input processing
- Two IP-layer locks per protocol
  - Global pcbinfo lock protecting inpcb lists
    - Insert, lookup, removal
  - Per-inpcb lock protects per-connection state
    - Global list manipulations involving connection
    - Per-connection state
- Input path acquires and holds tcbinfo lock
  - inpcb state may change requiring list changes
Addressing Locking Granularity

- Big issue is tcbinfo lock
  - Held over input paths that may reset connection
  - In 7.x, no longer over common case output paths

- Option 1: True reference counting on inpcbs
  - tcbinfo can be dropped and then safely re-acquired without race when inpcb lock is dropped

- Option 2: Decompose tcbinfo lock
  - Connection groups (ConnP-L per Willman, et al)
  - Less disruptive of current code
TODO

- Identify key code paths that would benefit from increased parallelism
  - IP forwarding path, netisr/loopback

- Develop new work management models to allow feedback from scheduler
  - Is it cheaper to direct dispatch or are we overloading the current thread/CPU?
  - Are there CPUs available to do additional work?
  - How should the network stack tell the scheduler about data/connection affinity?