Modernizing NetBSD Networking Facilities and Interrupt Handling

Ryota Ozaki <ozaki-r@iij.ad.jp>
Kengo Nakahara <k-nakahara@iij.ad.jp>
Overview of Our Work

1. MP-ify NetBSD networking facilities
2. Scale up NetBSD networking facilities

Goals

Targets

Layer 3 and above
- IPv4, IPv6, TCP, UDP, sockets, routing tables, etc.

Layer 2 and below
- Bridge, VLAN, BPF, device drivers, etc.

Software techniques
- Software interrupt, mutex, rwlock, passive serialization, etc.

Hardware technologies
- Multi-core, interrupt distribution, multi-queue, MSI/MSI-X, etc.
Contents

1. Current Status of Network Processing
2. MP-safe Networking
3. Interrupt Process Scaling
4. Multi-queue
5. Performance Measurement
6. Conclusion
Current Status of Network Processing - Outline

• Basic network processing
• Traditional mutual exclusion facilities
  – KERNEL_LOCK
  – IPL and SPL
• How each component works
  – A typical network device driver
  – Layer 2 forwarding
Basic Network Processing - TX

- Packets are passed from an upper layer to a lower layer one by one
- Enqueue packets to sender queue of a network interface driver (if_snd)
  - To delay TX when the device is busy
- All processes are down in a user process (LWP) context
  - Delayed TX may happen in HW interrupt context
Basic Network Processing - RX

• Hardware interrupt
  – Below Layer 2
  – Enqueue packets to pktqueue of a upper layer

• Software interrupt (softint)
  – Layer 3 and above (ipintr for IPv4 packets)
  – Dedicated softint for each protocol
    • IPv4, IPv6, ARP, etc.
Software Interrupt (softint)

• Special context to run low priority tasks of interrupts
• It can sleep/block
• It cannot allocate/free any memory
  – kmem(9) APIs aren’t allowed to use in softint context
  – Note that we can use malloc/free for now, but they are deprecated
• It doesn’t move between CPUs
Traditional Mutual Exclusion Facilities

• KERNEL_LOCK
• IPL and SPL
  – spl(9)
KERNEL_LOCK

• Big kernel lock
• Spin lock
  – It doesn’t sleep on acquisition
• To serialize activities on all CPUs
  – LWPs, HW interrupt handlers and softint handlers
• Easy to use
  – Can be used in HW interrupt context
  – Allow sleeping
  – Can use any other mutex facilities
  – Reentrant
KERNEL_LOCK (cont’d)

• Warning
  – It is unlocked when the LWP goes to sleep or is preempted
  – It doesn’t prevent any interrupts

• By default, interrupt handlers of network devices run with holding the lock
  – Passing MPSAFE flag to handler initialization functions allows handlers running without the lock
IPL and SPL

• IPL: interrupt priority level
  – See the below list

• SPL: system interrupt priority level
  – Prevents interrupts (IPL < SPL) from running

• `spl(9)` changes SPL
  – Enable atomic operations of data shared with interrupt handlers
  – E.g., `splnet` is to raise SPL to IPL_NET

• Limitation
  – Affects only interrupt handlers running on the current CPU

| IPL_*       | HIGH, SCHED, VM/NET, SOFTSERIAL, SOFTNET, SOFTBIO, SOFTCLOCK, NONE |
How Networking Facilities work - Outline

• vioif(4)
  – Device driver of virtio network device
  – Not complex

• bridge(4)
  – Pseudo device driver of network bridge
  – A Layer 2 networking facility
How vioif(4) Works

• Every interrupts are destined to CPU#0
  – No interrupt affinity / distribution facilities
  – Subsequent softint handlers are also run on CPU#0

• No fine-grain mutual exclusion for interrupt handlers
  – KERNEL_LOCK
How vioif(4) Works (cont’d)

• TX routines run on arbitrary CPUs
• Layer 2 and below are serialized with KERNEL_LOCK
• splnet(9) is used to protect shared data with interrupt handlers
  – E.g., ioctl doesn’t take KERNEL_LOCK
• vioif_rx_softint
  – A softint to fill receive buffers
  – It, LWPs and HW interrupt handlers are serialized with KERNEL_LOCK
How Layer 2 Forwarding Works
How Layer 2 Forwarding Works

• bridge(4) runs in both HW interrupt context and softint context

• Mutual exclusion
  – bridge_input: KERNEL_LOCK
  – bridge_forward: KERNEL_LOCK, splnet and softnet_lock
How Layer 2 Forwarding Works

Hardware interrupt:
- bridge
  - bridge_input
  - if_input
  - vioif_rx_deq
  - vioif_rx_vq_done
- RX
  - device

Software interrupt:
- bridge
  - bridge_forward
  - queue
  - if_snd
- if_start
  - TX
  - device

Other operations:
- schedule
- softint
- KERNEL_LOCK
- splnet
- softnet_lock
MP-safe Networking - Outline

• Mutual exclusion facilities for MP-safe
  – mutex(9)
  – rwlock(9)
  – pserialize(9)

• Case studies
  – Making vioif MP-safe
  – Making bridge MP-safe
mutex(9)

• It provides exclusive accesses to shared data
  – between mutex_enter and mutex_exit

• Two mutexes: spin and adaptive
  – The type is determined by its IPL
    • HIGH, SCHED, VM/NET => spin
    • SOFT* and NONE => adaptive

• Spin mutex
  – Busy-wait for the holder to release the mutex
  – Can be used in HW interrupt context
  – Raise SPL to its IPL when it has been acquired
    • So it can be used a replacement of spl APIs
    • For MP-safe, we should replace spl APIs with spin mutexes
mutex(9)

• Adaptive mutex
  – First busy-wait for some time
    • If the holder is running on another CPU
  – If couldn’t acquire, then go to sleep
  – Cannot be used in HW interrupt context
  – Turnstile
    • for the priority inversion problem

• No reentrancy
**rwlock(9)**

- Multiple readers and single writer
- Similar to adaptive mutex
  - Busy-wait then sleep
  - Cannot be used in HW interrupt context
  - Turnstile
    - for the priority inversion problem
  - No reentrancy
- Suit for cases read >>>> write
pserialize(9)

- pserialize = passive serialization
- Similar to Linux RCU
- Motivation
  - Provide high scalable data access on read-most workload
- Approach
  - Reduce/Remove exclusive data accesses by locks
  - Lockless data structure
pserialize(9) (cont’d)

• Issue
  – How to safely deallocate/free objects that readers may or may not reference
  – Using reference counting is a solution but it still suffers from data access contentions

• Solution
  – Provide a mechanism to wait for readers to dereference objects without interfering the readers
  – … with some expensive operations

Reader

Writer

.oO(When can I free this?)
pserialize(9) Implementation

• How to ensure readers left?
  – Assumption: a reader never block/sleep in reader’s critical section (CS)
  – If a reader LWP is switched to another LWP, we can ensure that the reader has left the CS and dereferenced a target object
  – If all LWPs on all CPUs are context-switched, we can ensure no reader is referencing the target object
pserialize(9) Implementation (cont’d)

• pserialize_read_{enter,exit}
  – Used the beginning and ending of critical sections
  – Equivalent to splsoftserial(9)
    • to prevent unexpected context switches
  – Programmers must ensure readers never sleep/block in pserialize critical sections

• pserialize_perform
  – Wait until all CPUs conduct context switches two times
Example Use of pserialize(9)

Reader

s = pserialize_read_enter();
/* Refer an object in a collection and use it here */
pserialize_read_exit(s);

Writer

mutex_enter(&writer_lock);
/* remove a object from the collection */

pserialize_perform(psz);
/* Here we can guarantee that no reader is touching the object */

mutex_exit(&writer_lock);
/* So we can free the object safely */
# Mutual Exclusion Facilities

<table>
<thead>
<tr>
<th></th>
<th>Can use in HW intr context?</th>
<th>Sleepable in its critical sections?</th>
<th>Reentrant</th>
<th>Can use in its critical sections?</th>
</tr>
</thead>
<tbody>
<tr>
<td>KERNEL_LOCK</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>all</td>
</tr>
<tr>
<td>spl</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>all (*1)</td>
</tr>
<tr>
<td>mutex (spin)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>mutex (spin)</td>
</tr>
<tr>
<td>mutex (adaptive)</td>
<td>no</td>
<td>yes (*2)</td>
<td>no</td>
<td>all</td>
</tr>
<tr>
<td>rwlock</td>
<td>no</td>
<td>yes (*2)</td>
<td>no</td>
<td>all</td>
</tr>
<tr>
<td>pserialize (read)</td>
<td>no</td>
<td>no</td>
<td>no (*3)</td>
<td>mutex (spin)</td>
</tr>
</tbody>
</table>

(*1) Should not lower SPL
(*2) Possible but not recommended
(*3) Possible but not expected
Case Studies - Outline

• vioif(4)
  – Device driver of virtio network device
  – A typical network device driver

• bridge(4)
  – Pseudo device driver of network bridge
  – A Layer 2 networking facility
Make vioif(4) MP-safe

• What to do: introduce fine-grain locking and remove KERNEL_LOCK
• Two spin mutexes for TX and RX
  – Serialize whole TX and RX routines
  – RX mutex is released when processing upper protocols (if_input)
• Graceful shutdown
  – Introduce “now stopping” flag
  – Need to check it on every mutex acquisitions
Make bridge(4) MP-safe

- Use pserialize(9) for scalable Layer 2 forwarding

- Two resources to protect
  - Bridge member list
    - A linked list to manage interfaces connected to the bridge
  - MAC address table
    - A hash list to manage caches of MAC addresses of frames passing the bridge
Bridge Member List

• Access characteristics
  – No modification on fast path
  – May sleep/block holding a bridge member

• Reader
  – pserialize(9) + reference counting (refcount)
  – Increments refcount of a member within pserialize’s critical section
  – Decrements after using the member
    • Wakes up a waiter (writer) via convar(9) if needed
    • Need a spin mutex(9) for condvar(9)
Bridge Member List (cont’d)

• Writer
  – Remove a member from the list and `pserialize_perform`
    • with holding the adaptive mutex
  – Then, sleep using `condvar(9)` if someone is still referencing the member
  – After that, we can free the member safely
MAC Address Table

• Access characteristics
  – Caches can be added on fast path
    • deleted on slow path (ioctl and timer)
  – No sleep/block with holding caches
• Reader
  – pserialize(9) for list accesses
• Writer
  – Spin mutex(9) for list updates
    • Caches can be added in HW interrupt context
  – Adaptive mutex(9) for pserialize_perform
static void
bridge_rtdelete(struct bridge_softc *sc, struct ifnet *ifp)
{
    struct bridge_rtnode *brt, *nbrt;

    BRIDGE_RT_LOCK(sc);
    BRIDGE_RT_INTR_LOCK(sc);
    LIST_FOREACH_SAFE(brt, &sc->sc_rtlist, brt_list, nbrt) {
        if (brt->brt_ifp == ifp)
            break;
    }
    if (brt == NULL)
        // snip error handling
    bridge_rtnode_remove(sc, brt);
    BRIDGE_RT_INTR_UNLOCK(sc);
    BRIDGE_RT_PSZ_PERFORM(sc);
    BRIDGE_RT_UNLOCK(sc);

    bridge_rtnode_destroy(brt);
}
Layer 2 Forwarding with pserialize(9)

Most routines run in parallel
Motivation

- **Issue:** all RX routines run on CPU#0
  - Including subsequent softints
- **Interrupt distribution**
  - Handle interrupts on other than CPU#0
- **MSI/MSI-X**
  - Assign RX and TX interrupts to different CPUs
- **Multi-queue**
  - Multiple RX and TX hardware queues
  - Assign hardware queues to different CPUs
Interrupt Process Scaling - Outline

- MSI/MSI-X
- Interrupt Distribution
MSI/MSI-X Support

• MSI is Message Signaled Interrupt
  – Interrupts occur as memory writes
  – MSI-X is an extension of MSI
    • It allows parallel interrupts from one device

• Current Status
  – NetBSD/ppc supports MSI
  – NetBSD does not support MSI-X

• We have implemented MSI-X support for x86/ppc
  – We have integrated existing MSI implementation into ours
Interrupt Distribution Support

• By default, all interrupts are destined to CPU#0

• How to distribute
  – Device drivers can re-assign hardware interrupts to other CPUs by a kernel API (intr_distribute)
  – System administrators can re-assign hardware interrupts to other CPUs by a userland command (intrctl(8))
Example of Interrupt Distribution (1)

- intrctl(8)
  - show interrupts list by intrctl list

<table>
<thead>
<tr>
<th>interrupt name</th>
<th>CPU#00(+)</th>
<th>CPU#01(+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ioapic0 pin 16</td>
<td>10*</td>
<td>0</td>
</tr>
<tr>
<td>msi0 vec 0</td>
<td>49115*</td>
<td>0</td>
</tr>
<tr>
<td>msi0 vec 1</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>msi0 vec 2</td>
<td>359017*</td>
<td>0</td>
</tr>
<tr>
<td>msi0 vec 3</td>
<td>477*</td>
<td>0</td>
</tr>
<tr>
<td>msi0 vec 4</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>msi1 vec 0</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>msi1 vec 1</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>msi1 vec 2</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>msi1 vec 3</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>msi1 vec 4</td>
<td>0*</td>
<td>0</td>
</tr>
</tbody>
</table>

wm0: legacy
vmx0: tx 0
vmx0: tx 1
vmx0: rx 0
vmx0: rx 1
vmx0: link
wm1: tx0
wm1: tx1
wm1: rx0
wm1: rx1
wm1: link
Example of Interrupt Distribution (2)

- intrctl(8)
  - set affinity by
    intrctl affinity –i “interrupt id” –c “cpuid”

```
# intrctl affinity -i 'msix0 vec 2' -c 1
# intrctl list | grep -e CPU -e vmx -e wm
interrupt name        CPU#00(+)     CPU#01(+)   
ioapic0 pin 16       43*           0
msix0 vec 0           49154*        0
msix0 vec 1           0*            0
msix0 vec 2           359081        155*
msix0 vec 3           574*          0
msix0 vec 4           0*            0
msix1 vec 0           0*            0
msix1 vec 1           0*            0
msix1 vec 2           0*            0
msix1 vec 3           0*            0
msix1 vec 4           0*            0
```
Result - No intrctl(8)
Result - With intrctl(8)
Result - With intrctl(8) + MSI-X
Multi-queue - Outline

• About Multi-queue
• Implementation
  – Common Part
  – Receive Side
  – Transmit Side
About Multi-queue

• Modern 1 Gigabit and more Ethernet controllers have more than one TX/RX queues, it is called “multi-queue”.

• We can assign interrupts of multiple hardware queues to different CPUs
  – Incoming packets are classified to different hardware queues based on, e.g., flows
  – We can handle packet flows that come to an Ethernet port on different CPUs in parallel
Multi-queue and MSI-X

- Each hardware queue is assigned to a MSI-X vector
- Each MSI-X vector can be bound to an interrupt destination (normally CPU)
  - by changing a corresponding entry in a MSI-X table
Implementation - Assigning MSI-X vectors

- `intrctl(8)` changes bindings between MSI-X vectors and CPUs
Implementation -
Receive Side Modifications

• if_wm
  – Split data structures per hardware queue
  – Setup multiple RX hardware queues
  – Assign RX handlers to different CPUs
  – Have spin mutex(9) per hardware queue

• if_bridge
  – Have per-cpu queues
    • between bridge_input and bridge_forward
Result - Receive Side (before)
Result - Receive Side (after)
Implementation - Transmit Side Modifications

- **if_wm**
  - (HW queues, data structures, mutex)
  - Have multiple TX queues (txq_snd)
    - one queue per TX hardware queue
    - (was if_snd queue)

- **if_bridge**
  - Enqueue packets to if_wm’s new TX queues (txq_snd)
  - TX queue selection logic (tentative)
    - queue ID = (CPU ID) % (# of TX hardware queues)
Result - Transmit Side (before)

CPU#0

sc_fwd_pktq

bridge_forward

if_snd

bridge_forward

if_start

wm_start

NIC queue

device

not used : (# of CPUs) - 1

not used : (# of TX queues) - 1

in the system

out of the system
Result - Transmit Side (after)

CPU#0

sc_fwd_pktq

<table>
<thead>
<tr>
<th>bridge_forward</th>
<th>if_transmit</th>
<th>wm_transmit</th>
</tr>
</thead>
</table>

CPU#1

<table>
<thead>
<tr>
<th>bridge_forward</th>
<th>if_transmit</th>
<th>wm_transmit</th>
</tr>
</thead>
</table>

CPU#2

<table>
<thead>
<tr>
<th>bridge_forward</th>
<th>if_transmit</th>
<th>wm_transmit</th>
</tr>
</thead>
</table>

CPU#3

<table>
<thead>
<tr>
<th>bridge_forward</th>
<th>if_transmit</th>
<th>wm_transmit</th>
</tr>
</thead>
</table>

(CPU id) % (# of TX queues) => TX queue id

not used:
(# of TX queues) - (# of CPUs)

NIC queue

device

in the system

out of the system
Contents

1. Current Status of Network Processing
2. MP-safe Networking
3. Interrupt Process Scaling
4. Multi-queue
5. Performance Measurement
6. Conclusion
Performance Measurement - Outline

• Setting
• Results
Setting -
DUT (Device Under Test)

- **Hardware**
  - Supermicro A1SRi-2758F
    - 8 core Atom C2758 SoC
    - 4 port I354 Ethernet adapter (each port has 8 TX/RX queues)

- **Kernel**
  - GENERIC
    - NetBSD current at 2015-01-07
    - built with GENERIC config
  - NET_MPSAFE
    - GENERIC plus our MP-safe implementation
    - built with GENERIC plus NET_MPSAFE enabled config
Setting - Environment

• DUT as a bridge
  – two ports used

• RFC2544 throughput

• Send UDP packets
  – port 8000 - 9000
  – bi-directionally
Results (1)
frame size vs kfps
Results (1)
frame size vs kfps

- Achieve wire rate @ frame size 256 byte
- 384 byte
- 512 byte
- 768 byte
- NOT achieve wire rate
Results (2)

# of cores vs kfps
Results (2)

# of cores vs kfps

well scaling

saturated

saturated

saturated
Conclusion

• We want to run networking facilities in parallel
• Our contributions
  – Made bridge(4) and some device driver MP-safe
  – Supported MSI/MSI-X
  – Modified if_wm to support multi-queue with MSI/MSI-X
• We have measured our implementations
  – Our implementation scales well
Current Status

• Most bridge(4) modifications have already been merged to NetBSD-current
  – To use our bridge(4) implementation, enable "NET_MPSAFE" option

• MSI/MSI-X support is not merged yet
  – Therefore, if_wm multi-queue support is not merged yet, either
Thank you for listening!

Any questions?