Eliminating Receive Livelock in an Interrupt-Driven Kernel

Administrivia

- Lab 2 grading is underway!
- Last academic week.
  - It's been fun. :)
- Final Exam next Monday, 3:30pm - 6:30pm, Skilling Auditorium
  - Open notes, same as midterm.
  - Wholistic. Everything we've covered is fair game.
    - Slight emphasis on things we didn't cover on the midterm.
  - Excellent grades will be accounted for especially.
    - Can make up for poor performance on midterm.

Back to papers! What's the deal with this one?

* Jeff Mogul, K.K. Ramakrishnan
  - KK Ramakrishnan: DEC 11 years, then ATT/Bell Labs almost 20 years, now UCR
  - Jeff Mogul: DEC, Compaq, HP, Google (author on HTTP 1.1 RFC)
  - USENIX ATC, 1996
* Huge industry folks! What does that mean for this paper?
  - This is real stuff! This shipped!

Q: What was the max ethernet speed in 1996?
A: 10mbit, typically. 100mbit (introduced in '95), if you were lucky.

- 1gbit not introduced until 1998.

Q: How fast were processors?
A: Pentium's were ~150mhz. DEC alpha was at 333mhz.

- 3 cycles per bit (24/byte) to max 100mbit link.
- Packet size is, say, 1K. So ~24k cycles/packet.
- CPUs now are 10x faster, likely much more due to arch. differences.

Let's get some background on the OS networking stack.

Q: We start with a NIC; what does it look like the OS?
A: Basically a queue of packets.

```
                 ---------------
  EXTERNAL -->      | ~ | ~ |
                 ---------------
```

Q: How do we know there's something in the queue?
A: We get an interrupt.

Q: How often do we get interrupts?
A: Configurable. Usually want an interrupt everytime there's something there, or some acceptable time thereafter, to minimize latency.

What about polling-based implementations?

Instead of getting an interrupt from the NIC when there's a packet ready, we could periodically check for ready packets.

Q: What's the main problem with polling?
A: Tradeoff between interactivity and latency.
high interactity = low poll rate = high latency
low interactity = high poll rate = low latency

Ok, now we know how the OS interfaces with the NIC.

Q: What happens once the OS gets notified?
A: More queues!

```
                   NIC                     PROTO              DRAIN (?)
              ---------------         ---------------     ---------------
  EXTERNAL -->   | ~ | ~ |      -->     | ~ | ~ |     -->    | ~ | ~ |      --> ?
              ---------------         ---------------     ---------------

                          DEVICE INT          SOFTWARE INT
                           [driver]            [in-kernel]
```

In BSD 4.2:

```
  0) NIC interrupt comes in.
  1) The OS finds the driver for the NIC, gives it the interrupt.
    - In reality, the driver gets the INT directly.
  2) The driver copies the packet from NIC into a protocol-specific queue.
    2b) The driver issues a software interrupt.
  3) Some protocol code in the kernel wakes up, processes packet.
    3b) The protocol code finally copies the packet into some drain.
      - Could be a user-level socket queue.
      - Could be an outbound queue (for routing, for instance)
```

```
  on NIC interrupt (in driver):
    packet = NIC_queue.pop()
    copy packet -> kernel_buffer
    protocol_queue.push(kernel_buffer)
    emit software interrupt
```

Back to the paper.

When we talked about scalability, we had this ideal graph:

```
      |           /
    w |          /
    o |         /
    r |        /
    k |      /
      | /
     ----------------
       worker count
```

Q: For this paper, what do we label the X and Y axis?
A: X = input rate, Y = output rate

Q: Why can't we get this ideal version?
A: There are physical limits. Our NICs/CPUs/electrons are only so fast.

Q: So what do we want?
A: Reach MLFRR, sustain it.

```
    |
  o |       _______   <--- (MLFRR = Maximm Loss Free Receive Rate)
  u |      /
  t |    /
    | /
   ----------------
      input rate
```

Q: What does this paper show us we get unless we're clever?
A: * = livelock! Reach MLFRR, decline to 0.

```
    |
  o |
  u |     ____
```

```
    t |   /      \
      | /         \
      -------------*--
         input rate
```

Q: Why is called livelock?
A: Play on "dead"lock, except you’re still doing work. So alive, but not
   making progress.

This paper gives us two things (1-2 punch):
  1) Why we get the above behavior.
  2) How to avoid the behavior.
  BONUS) Experience shipping software/hardware that avoids this.

Q: Now we have some background. What’s the problem?
A: Spend all of our time handling interrupts, none processing packets.
  - This is known as livelock.

Q: What happen in the code, exactly?
A:
  1) Interrupt handler runs, reads packet, tries to push onto next queue.
  2) At some point, queue is full, so packet just gets dropped.
  3) This happens ad infinitum. No useful work is done.

Q: Why doesn’t the non-interrupt handling code get to run?
A: Interrupt handling code runs at a higher priority. If there’s an interrupt,
   it’ll be handled. This is the main issue!
A: Conclusion of paper states is beautifully:
   Systems progressively reduce the priority of processing a packet as it goes
   further into the system.

Aside from Livelock avoidance, we want other properties from our system.

- High throughput
  - rate at which packets are delivered to consumer (app, etc.)
  - useful throughput also depends on transmitting packets

- Low latency
  - high caused by long queues

- Low jitter
  - high caused by bursty scheduling
  - Variance in latency

- Fair resource allocation
  - Among packet reception, transmission, protocol processing, I/O processing,
    housekeeping, application processing
  - Includes not starving application processing

How do we accomplish our goals?

Let’s focus on the paper’s specific example: a router.

```
                 NIC                      PROTO                 OUTPUT
           ---------------          ---------------       ---------------
  EXT -->     | ~ | ~ |       -->      | ~ | ~ |     -->     | ~ | ~ |      --> EXT
           ---------------          ---------------       ---------------

                        DEVICE INT          SOFTWARE INT
                         [driver]            [in-kernel]
```

Q: Which of these queues are in kernel-space, and which in user-space?
A: All of them are in kernel space!

Q: Without making changes, which graph do we get?
A: Figure 6-1:

```
     |
   o |
   u |     ____          -> 4.5k pkts/s
   t |   /      \
     | /         \
    -------------*--  * -> ~12k pkts/s
       input rate
```

How do we fix this?

1) Disable interrupts if spending too much time managing them
   - Detect livelock can/is happen(ning), disable interrupts.
   - Proposed solution:
     a) disable when you're about to drop a packet because queue is full
     Q: When do we reenable interrupts?
     A: When there's space in the next queue, ideally, or some time later. Can check either condition on timer interrupts.

   Q: Does this solve livelock? (think about the last queue)
   A: No! Doing the above doesn't guarantee progress. Progress means doing useful work. Here, sending packets. We could move packets from the proto queue to the output queue and never take them off that queue.

2) On interrupt, disable interrupts and poll across all sources and drains.
   - Note: we poll the NIC for the ability to receive or transmit. To be able to transmit means there's free space in the NIC's transmit queue, so needed to have transmitted before.
   - Can now ensure that if packets get to the end, that we act on them.

   Q: Does this solve livelock? (think about the last queue)
   A: We've handled the beginning and end, but the code that does the middle part still runs at a lower priortiy than the interrupt handler, so can get interrupted while processing packets.
   A: Probably doesn't livelock anymore, but severely compromises performance (latency): not moving packets through even though they're almost done.

3) Ensure that (in-kernel) processing code can't get interrupted.
   a) Run it at the same IPL as interrupt handler.
   b) On interrupt from a given source/drain:
      - Disbale interrupts (for that ONE). Set servicing needed flag for ONE.
      - Poll all sources with flags.
        - Can still get interrutped at most once per source/drain.
   c) kill the second buffer since it's no longer needed.

   Q: Does this solve livelock?
   A: Yes, for in-kernel code.

What do they actually do?

1) Kill the middle buffer.
2) Do the service flag.
3) Don't do feedback thing.
4) Do a 'quota' thing.

Q: What's going in figure 6-3, especially re: quotas?

----

5) Disable interrupts to give user-level code chance to run

- If fraction time processing packets > some threshold, disable interrupts
    - can use timer interrupts to see how much time, OR
    - can timestamp enter/exit of kernel packet processing