Eliminating Receive Livelock in an Interrupt-Driven Kernel,
by Mogul and Ramakrishnan

Why are we reading this paper?
Most scheduling issues are boring, solvable with faster CPU.
This one is much more fundamental.
And it comes up a lot, in many guises.

What is the point of interrupts?
What is the problem the paper addresses?
For IP forwarding example.

Explain Figure 6-1.
why does it go up?
what determines how high the peak is?
why does it go down?
what determines how fast it goes down?

Suppose I wanted to test an NFS server for livelock.
Run client with this loop:
while(1){
    send NFS READ RPC;
    wait for response;
}
what would I see?
Is the NFS server probably subject to livelock?

Why not tell the O/S scheduler to give interrupts lower priority?

Could you fix this by making interrupts faster?
Maybe, if coupled with some limit on input rate.

Why not completely process each packet in the interrupt handler?
I.e. forward it?

What's the paper's solution?
No IP input queue.
Input processing and device input polling in kernel thread.
Device receive interrupt just wakes up thread.
And leaves interrupts *disabled* for that device.
Thread does all input processing,
then re-enables interrupts.

Why does this work?
What happens when packets arrive too fast?
What happens when packets arrive slowly?

Explain Figure 6-3.
why does "Polling (no quota)" work badly?
Input still starves xmit complete processing.
why does it immediately fall to zero, rather than gradually decreasing?
xmit complete processing must be very cheap compared to input.

Explain Figure 6-4.
why does "Polling, no feedback" behave badly?
There's a queue in front of screend.
we can still give 100% to input thread, 0% to screend.
why does "Polling w/ feedback" behave well?
Input thread yields when queue to screend fills.

Why are the two solutions different?
1. Polling thread *with quotas*.
2. Feedback from full queue.
(I believe they should have used #2 for both.)

If we apply their fixes, does the phenomenon totally go away?
E.g. for web server, waits for disk, &c.
Can the net device throw away packets without slowing down host?

What if processing has more complex structure?
Chain of processing stages with queues?
Does feedback work?
What happens when a late stage is slow?
Split at some point, multiple parallel paths?
No so great; one slow path blocks all paths.

Can we formulate any general principles?
Don't spend time on new work before completing existing work.
Or give new work lower priority than partially-completed work.