ECE 373 Assignment #6
Spring 2022

Interrupts, descriptors, and saying hi to the world

Up until now, we've been controlling hardware locally by altering simple states. But for a device to be useful, it needs to process some sort of data, whether it's incoming, outgoing, or both. The core of a driver is doing just that; configuring a device, allocating resources for it, then pushing out data and processing incoming data via DMA, interrupts, and using descriptors.

The driver side

A good place to start is to re-use your assignment #4 codebase (the chardev plus PCI access to blink the LED). This will get your chardev and PCI layer ready, and you can start bolting the new code in this assignment to that.

We will be working with the same PCI Ethernet device, so make sure you have the manual ready!

Mainly, you will need to add the following:

1. A small set of legacy descriptors for the receive queue (16 descriptors) with buffers allocated to 2048 bytes.

2. A control mechanism for keeping track of the HEAD and TAIL of the receive queue

3. An interrupt handler, tied to the legacy interrupt source

4. A workqueue thread to handle deferred processing from the interrupt

5. Configuration code to put the chip into promiscuous mode, and force the link up at 1 Gigabit

6. Update your read() system call function to return a 16-bit unsigned integer, the lower 8-bits will be the value of the receive queue TAIL, the upper 8-bits will be the value of the receive queue HEAD.

The e1000e driver is a great reference to start with when looking at the chip initialization. A good order to follow in attacking this assignment is to make each of these steps work before doing the next:

1. Start with adding and allocating the descriptors and their buffers. It probably wouldn't be a bad idea to make a function to do this, and call that from probe() after all the BAR stuff is mapped.

2. Next add the interrupt handler and get the interrupt hooked up with request_irq(). If you were successful, you should see your interrupt registered in /proc/interrupts.

3. The workqueue thread should be added next, and have it invoked from the interrupt handler.

4. Next add the descriptor ring maintenance of HEAD and TAIL into the interrupt handler, and add the read() system call changes.

5. Next add the mechanism to re-arm the interrupt before exiting the handler.

6. Now you're ready to force the link up, and start testing with your link partner.

The link partner side

This can be the host machine running VirtualBox (aka the Hypervisor). Use 'ping' or other packet sending tools (playtcp, scapy, packeth, pktgen, etc) to send single packets.

Example from linux link partner: ping -I eth1 -c 1 -r 11.22.33.44

The meat of the driver

The idea is your driver will be receiving interrupts as packets come across from the link partner:

1. In the interrupt handler, turn on both green LED's, submit a workqueue thread to run, re-arm the interrupt, then return from the interrupt

2. In the workqueue thread, sleep for 0.5 seconds, then turn off both green LED's, and bump TAIL when done.

3. When a userspace program opens the file and reads it, return the packed unsigned int value of HEAD and TAIL, and unpack it in userspace to print the HEAD and TAIL values.

Do this a few times so you can see the two values changing, and to make sure the rings properly wrap. You might try sending several packets and let the workqueue thread process all the ready packets until TAIL is up to HEAD again.

Make sure your code properly tears down the interrupt, unpins all the descriptor memory, frees all memory, and cleanly exits.

How to finish

Turn in these materials to the Github Classroom repos by 11pm on **Monday**, **6-June-2022:**

1. Source code to your user program and Makefile.

2. Note that you will need to unload the current network driver!

3. A typescript of the user program running at least three times (partly to watch the HEAD and TAIL values change)

4. A note from someone (anyone, but our TA is a good resource) saying they actually saw it work.