back to posts

Monitoring and Tuning the Linux Networking Stack: Receiving Data

Jun 22, 2016 • packagecloud

packagecloud linux kernel networking optimization tuning monitoring

- TL;DR
- Special thanks
- General advice on monitoring and tuning the Linux networking stack
- Overview
- Detailed Look
  - Network Device Driver
    - Initialization
      - PCI initialization
      - PCI probe
        - A peek into PCI initialization
      - More Linux PCI driver information
    - Network device initialization
      - `struct net_device_ops`
      - `ethtool` registration
      - IRQs
      - NAPI
      - NAPI initialization in the `igb` driver
    - Bringing a network device up
      - Preparing to receive data from the network
      - Enable NAPI
      - Register an interrupt handler
      - Enable Interrupts

        - The network device is now up
    - Monitoring network devices
        - Using `ethtool -S`
        - Using sysfs
        - Using `/proc/net/dev`
    - Tuning network devices
        - Check the number of RX queues being used
        - Adjusting the number of RX queues
        - Adjusting the size of the RX queues
        - Adjusting the processing weight of RX queues
        - Adjusting the rx hash fields for network flows
        - ntuple filtering for steering network flows
- SoftIRQs
    - What is a softirq?
    - `ksoftirqd`
    - `__do_softirq`
    - Monitoring
        - `/proc/softirqs`
- Linux network device subsystem
    - Initialization of network device subsystem
        - Initialization of `struct softnet_data` structures
        - Initialization of softirq handlers
    - Data arrives
        - Interrupt handler
        - NAPI and `napi_schedule`
        - A note about CPU and network data processing
        - Monitoring network data arrival
            - Hardware interrupt requests
        - Tuning network data arrival
            - Interrupt coalescing
            - Adjusting IRQ affinities
    - Network data processing begins
        - `net_rx_action` processing loop
        - NAPI `poll` function and `weight`
        - The NAPI / network device driver contract
        - Finishing the `net_rx_action` loop
        - Exiting the loop when limits are reached
        - NAPI poll
            - `igb_poll`
            - `igb_clean_rx_irq`
        - Monitoring network data processing
            - `/proc/net/softnet_stat`
        - Tuning network data processing
            - Adjusting the `net_rx_action` budget

- Generic Receive Offloading (GRO)
  - Tuning: Adjusting GRO settings with `ethtool`
  - `napi_gro_receive`
    - `dev_gro_receive`
  - `napi_skb_finish`
- Receive Packet Steering (RPS)
  - Tuning: Enabling RPS
- Receive Flow Steering (RFS)
  - Tuning: Enabling RFS
- Hardware accelerated Receive Flow Steering (aRFS)
  - Tuning: Enabling accelerated RFS (aRFS)
- Moving up the network stack with `netif_receive_skb`
  - Tuning: RX packet timestamping
- `netif_receive_skb`
  - Without RPS (default setting)
  - With RPS enabled
    - `enqueue_to_backlog`
    - Flow limits
    - Monitoring: Monitor drops due to full `input_pkt_queue` or flow limit
    - Tuning
      - Tuning: Adjusting `netdev_max_backlog` to prevent drops
      - Tuning: Adjust the NAPI weight of the backlog `poll` loop
      - Tuning: Enabling flow limits and tuning flow limit hash table size
  - backlog queue NAPI poller
  - `process_backlog`
  - `__netif_receive_skb_core` delivers data to packet taps and protocol layers
  - Packet tap delivery
  - Protocol layer delivery
- Protocol layer registration
  - IP protocol layer
    - `ip_rcv`
    - netfilter and iptables
    - `ip_rcv_finish`
      - Tuning: adjusting IP protocol early demux
    - `ip_local_deliver`
    - `ip_local_deliver_finish`
    - Monitoring: IP protocol layer statistics
  - Higher level protocol registration
  - UDP protocol layer
    - `udp_rcv`
    - `__udp4_lib_rcv`
    - `udp_queue_rcv_skb`
    - `sk_rcvqueues_full`
      - Tuning: Socket receive queue memory

        - `udp_queue_rcv_skb`
        - `__udp_queue_rcv_skb`
        - Monitoring: UDP protocol layer statistics
            - `/proc/net/snmp`
            - `/proc/net/udp`
    - Queuing data to a socket
- Extras
    - Timestamping
    - Busy polling for low latency sockets
    - Netpoll: support for networking in critical contexts
    - `SO_INCOMING_CPU`
    - DMA Engines
        - Intel's I/O Acceleration Technology (IOAT)
            - Direct cache access (DCA)
            - Monitoring IOAT DMA engine
            - Tuning IOAT DMA engine
- Conclusion
- Help with Linux networking or other systems
- Related posts

TL;DR

This blog post explains how computers running the Linux kernel receive packets, as well as how to monitor and tune each component of the networking stack as packets flow from the network toward userland programs.

It is impossible to tune or monitor the Linux networking stack without reading the source code of the kernel and having a deep understanding of what exactly is happening.

This blog post will hopefully serve as a reference to anyone looking to do this.

Special thanks

Special thanks to the folks at Private Internet Access who hired us to research this information in conjunction with other network research and who have graciously allowed us to build upon the research and publish this information.

The information presented here builds upon the work done for Private Internet Access, which was originally published as a 5 part series starting here.

General advice on monitoring and tuning the Linux networking stack

The networking stack is complex and there is no one size fits all solution. If the performance and health of your networking is critical to you or your business, you will have no choice but to invest a considerable amount of time, effort, and money into understanding how the various parts of the system interact.

Ideally, you should consider measuring packet drops at each layer of the network stack. That way you can determine and narrow down which component needs to be tuned.

This is where, I think, many operators go off track: the assumption is made that a set of sysctl settings or `/proc` values can simply be reused wholesale. In some case, perhaps, but it turns out that the entire system is so nuanced and intertwined that if you desire to have meaningful monitoring or tuning, you must strive to understand how the system functions at a deep level. Otherwise, you can simply use the default settings, which should be good enough until further optimization (and the required investment to deduce those settings) is necessary.

Many of the example settings provided in this blog post are used solely for illustrative purposes and are not a recommendation for or against a certain configuration or default setting. Before adjusting any setting, you should develop a frame of reference around what you need to be monitoring to notice a meaningful change.

Adjusting networking settings while connected to the machine over a network is dangerous; you could very easily lock yourself out or completely take out your networking. Do not adjust these settings on production machines; instead make adjustments on new machines and rotate them into production, if possible.

Overview

For reference, you may want to take have a copy of the device data sheet handy. This post will examine the Intel I350 Ethernet controller, controlled by the `igb` device driver. You can find that data sheet (warning: LARGE PDF) here for your reference.

The high level path a packet takes from arrival to socket receive buffer is as follows:

1. Driver is loaded and initialized.
2. Packet arrives at the NIC from the network.
3. Packet is copied (via DMA) to a ring buffer in kernel memory.
4. Hardware interrupt is generated to let the system know a packet is in memory.
5. Driver calls into NAPI to start a poll loop if one was not running already.
6. `ksoftirqd` processes run on each CPU on the system. They are registered at boot time. The `ksoftirqd` processes pull packets off the ring buffer by calling the NAPI `poll` function that the device driver registered during initialization.
7. Memory regions in the ring buffer that have had network data written to them are unmapped.
8. Data that was DMA'd into memory is passed up the networking layer as an 'skb' for more processing.
9. Incoming network data frames are distributed among multiple CPUs if packet steering is enabled or if the NIC has multiple receive queues.
10. Network data frames are handed to the protocol layers from the queues.
11. Protocol layers process data.
12. Data is added to receive buffers attached to sockets by protocol layers.

This entire flow will be examined in detail in the following sections.

The protocol layers examined below are the IP and UDP protocol layers. Much of the information presented will serve as a reference for other protocol layers, as well.

Detailed Look

This blog post will be examining the Linux kernel version 3.13.0 with links to code on GitHub and code snippets throughout this post.

Understanding exactly how packets are received in the Linux kernel is very involved. We'll need to closely examine and understand how a network driver works, so that parts of the network stack later are more clear.

This blog post will look at the `igb` network driver. This driver is used for a relatively common server NIC, the Intel Ethernet Controller I350. So, let's start by understanding how the `igb` network driver works.

Network Device Driver

Initialization

A driver registers an initialization function which is called by the kernel when the driver is loaded. This function is registered by using the `module_init` macro.

The `igb` initialization function (`igb_init_module`) and its registration with `module_init` can be found in drivers/net/ethernet/intel/igb/igb_main.c.

Both are fairly straightforward:

```
/**
 *  igb_init_module - Driver Registration Routine
 *
 *  igb_init_module is the first routine called when the driver is
 *  loaded. All it does is register with the PCI subsystem.
 **/
static int __init igb_init_module(void)
{
  int ret;
  pr_info("%s - version %s\n", igb_driver_string, igb_driver_version);
  pr_info("%s\n", igb_copyright);

  /* ... */

  ret = pci_register_driver(&igb_driver);
  return ret;
}

module_init(igb_init_module);
```

The bulk of the work to initialize the device happens with the call to `pci_register_driver` as we'll see next.

PCI initialization

The Intel I350 network card is a PCI express device.

PCI devices identify themselves with a series of registers in the PCI Configuration Space.

When a device driver is compiled, a macro named `MODULE_DEVICE_TABLE` (from `include/module.h`) is used to export a table of PCI device IDs identifying devices that the device driver can control. The table is also registered as part of a structure, as we'll see shortly.

The kernel uses this table to determine which device driver to load to control the device.

That's how the OS can figure out which devices are connected to the system and which driver should be used to talk to the device.

This table and the PCI device IDs for the `igb` driver can be found in `drivers/net/ethernet/intel/igb/igb_main.c` and `drivers/net/ethernet/intel/igb/e1000_hw.h`, respectively:

```
static DEFINE_PCI_DEVICE_TABLE(igb_pci_tbl) = {
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I354_BACKPLANE_1GBPS) },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I354_SGMII) },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I354_BACKPLANE_2_5GBPS) },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I211_COPPER), board_82575 },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I210_COPPER), board_82575 },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I210_FIBER), board_82575 },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I210_SERDES), board_82575 },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I210_SGMII), board_82575 },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I210_COPPER_FLASHLESS), board_82575 },
  { PCI_VDEVICE(INTEL, E1000_DEV_ID_I210_SERDES_FLASHLESS), board_82575 },

  /* ... */
};
MODULE_DEVICE_TABLE(pci, igb_pci_tbl);
```

As seen in the previous section, `pci_register_driver` is called in the driver's initialization function.

This function registers a structure of pointers. Most of the pointers are function pointers, but the PCI device ID table is also registered. The kernel uses the functions registered by the driver to bring the PCI device up.

From `drivers/net/ethernet/intel/igb/igb_main.c`:

```
static struct pci_driver igb_driver = {
  .name     = igb_driver_name,
  .id_table = igb_pci_tbl,
  .probe    = igb_probe,
  .remove   = igb_remove,

  /* ... */
```

```
};
```

PCI probe

Once a device has been identified by its PCI IDs, the kernel can then select the proper driver to use to control the device. Each PCI driver registers a probe function with the PCI system in the kernel. The kernel calls this function for devices which have not yet been claimed by a device driver. Once a device is claimed, other drivers will not be asked about the device. Most drivers have a lot of code that runs to get the device ready for use. The exact things done vary from driver to driver.

Some typical operations to perform include:

1. Enabling the PCI device.
2. Requesting memory ranges and IO ports.
3. Setting the DMA mask.
4. The ethtool (described more below) functions the driver supports are registered.
5. Any watchdog tasks needed (for example, e1000e has a watchdog task to check if the hardware is hung).
6. Other device specific stuff like workarounds or dealing with hardware specific quirks or similar.
7. The creation, initialization, and registration of a `struct net_device_ops` structure. This structure contains function pointers to the various functions needed for opening the device, sending data to the network, setting the MAC address, and more.
8. The creation, initialization, and registration of a high level `struct net_device` which represents a network device.

Let's take a quick look at some of these operations in the `igb` driver in the function `igb_probe`.

A peek into PCI initialization

The following code from the `igb_probe` function does some basic PCI configuration. From drivers/net/ethernet/intel/igb/igb_main.c:

```
 err = pci_enable_device_mem(pdev);

/* ... */

err = dma_set_mask_and_coherent(&pdev->dev, DMA_BIT_MASK(64));

/* ... */

err = pci_request_selected_regions(pdev, pci_select_bars(pdev,
          IORESOURCE_MEM),
          igb_driver_name);

pci_enable_pcie_error_reporting(pdev);

pci_set_master(pdev);
pci_save_state(pdev);
```

First, the device is initialized with `pci_enable_device_mem`. This will wake up the device if it is suspended, enable memory resources, and more.

Next, the DMA mask will be set. This device can read and write to 64bit memory addresses, so `dma_set_mask_and_coherent` is called with `DMA_BIT_MASK(64)`.

Memory regions will be reserved with a call to `pci_request_selected_regions`, PCI Express Advanced Error Reporting is enabled (if the PCI AER driver is loaded), DMA is enabled with a call to `pci_set_master`, and the PCI configuration space is saved with a call to `pci_save_state`.

Phew.

More Linux PCI driver information

Going into the full explanation of how PCI devices work is beyond the scope of this post, but this excellent talk, this wiki, and this text file from the linux kernel are excellent resources.

Network device initialization

The `igb_probe` function does some important network device initialization. In addition to the PCI specific work, it will do more general networking and network device work:

1. The `struct net_device_ops` is registered.
2. `ethtool` operations are registered.
3. The default MAC address is obtained from the NIC.
4. `net_device` feature flags are set.
5. And lots more.

Let's take a look at each of these as they will be interesting later.

struct net_device_ops

The `struct net_device_ops` contains function pointers to lots of important operations that the network subsystem needs to control the device. We'll be mentioning this structure many times throughout the rest of this post.

This `net_device_ops` structure is attached to a `struct net_device` in `igb_probe`. From drivers/net/ethernet/intel/igb/igb_main.c)

```
static int igb_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
  /* ... */


  netdev->netdev_ops = &igb_netdev_ops;
```

And the functions that this `net_device_ops` structure holds pointers to are set in the same file. From drivers/net/ethernet/intel/igb/igb_main.c:

```
static const struct net_device_ops igb_netdev_ops = {
  .ndo_open               = igb_open,
  .ndo_stop               = igb_close,
  .ndo_start_xmit          = igb_xmit_frame,
```

```
  .ndo_get_stats64        = igb_get_stats64,
  .ndo_set_rx_mode        = igb_set_rx_mode,
  .ndo_set_mac_address    = igb_set_mac,
  .ndo_change_mtu         = igb_change_mtu,
  .ndo_do_ioctl           = igb_ioctl,

  /* ... */
```

As you can see, there are several interesting fields in this `struct` like `ndo_open`, `ndo_stop`, `ndo_start_xmit`, and `ndo_get_stats64` which hold the addresses of functions implemented by the `igb` driver.

We'll be looking at some of these in more detail later.

ethtool registration

`ethtool` is a command line program you can use to get and set various driver and hardware options. You can install it on Ubuntu by running `apt-get install ethtool`.

A common use of `ethtool` is to gather detailed statistics from network devices. Other `ethtool` settings of interest will be described later.

The `ethtool` program talks to device drivers by using the `ioctl` system call. The device drivers register a series of functions that run for the `ethtool` operations and the kernel provides the glue.

When an `ioctl` call is made from `ethtool`, the kernel finds the `ethtool` structure registered by the appropriate driver and executes the functions registered. The driver's `ethtool` function implementation can do anything from change a simple software flag in the driver to adjusting how the actual NIC hardware works by writing register values to the device.

The `igb` driver registers its `ethtool` operations in `igb_probe` by calling `igb_set_ethtool_ops`:

```
static int igb_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
  /* ... */

  igb_set_ethtool_ops(netdev);
```

All of the `igb` driver's `ethtool` code can be found in the file `drivers/net/ethernet/intel/igb/igb_ethtool.c` along with the `igb_set_ethtool_ops` function.

From `drivers/net/ethernet/intel/igb/igb_ethtool.c`:

```
void igb_set_ethtool_ops(struct net_device *netdev)
{
```

```
  SET_ETHTOOL_OPS(netdev, &igb_ethtool_ops);
}
```

Above that, you can find the `igb_ethtool_ops` structure with the `ethtool` functions the `igb` driver supports set to the appropriate fields.

From `drivers/net/ethernet/intel/igb/igb_ethtool.c` :

```
static const struct ethtool_ops igb_ethtool_ops = {
  .get_settings           = igb_get_settings,
  .set_settings           = igb_set_settings,
  .get_drvinfo            = igb_get_drvinfo,
  .get_regs_len           = igb_get_regs_len,
  .get_regs               = igb_get_regs,
  /* ... */
```

It is up to the individual drivers to determine which `ethtool` functions are relevant and which should be implemented. Not all drivers implement all `ethtool` functions, unfortunately.

One interesting `ethtool` function is `get_ethtool_stats`, which (if implemented) produces detailed statistics counters that are tracked either in software in the driver or via the device itself.

The monitoring section below will show how to use `ethtool` to access these detailed statistics.

IRQs

When a data frame is written to RAM via DMA, how does the NIC tell the rest of the system that data is ready to be processed?

Traditionally, a NIC would generate an interrupt request (IRQ) indicating data had arrived. There are three common types of IRQs: MSI-X, MSI, and legacy IRQs. These will be touched upon shortly. A device generating an IRQ when data has been written to RAM via DMA is simple enough, but if large numbers of data frames arrive this can lead to a large number of IRQs being generated. The more IRQs that are generated, the less CPU time is available for higher level tasks like user processes.

The New Api (NAPI) was created as a mechanism for reducing the number of IRQs generated by network devices on packet arrival. While NAPI reduces the number of IRQs, it cannot eliminate them completely.

We'll see why that is, exactly, in later sections.

NAPI

NAPI differs from the legacy method of harvesting data in several important ways. NAPI allows a device driver to register a `poll` function that the NAPI subsystem will call to harvest data frames.

The intended use of NAPI in network device drivers is as follows:

1. NAPI is enabled by the driver, but is in the off position initially.
2. A packet arrives and is DMA'd to memory by the NIC.
3. An IRQ is generated by the NIC which triggers the IRQ handler in the driver.
4. The driver wakes up the NAPI subsystem using a softirq (more on these later). This will begin harvesting packets by calling the driver's registered `poll` function in a separate thread of execution.

5. The driver should disable further IRQs from the NIC. This is done to allow the NAPI subsystem to process packets without interruption from the device.
6. Once there is no more work to do, the NAPI subsystem is disabled and IRQs from the device are re-enabled.
7. The process starts back at step 2.

This method of gathering data frames has reduced overhead compared to the legacy method because many data frames can be consumed at a time without having to deal with processing each of them one IRQ at a time.

The device driver implements a `poll` function and registers it with NAPI by calling `netif_napi_add`. When registering a NAPI `poll` function with `netif_napi_add`, the driver will also specify the `weight`. Most of the drivers hardcode a value of `64`. This value and its meaning will be described in more detail below.

Typically, drivers register their NAPI `poll` functions during driver initialization.

NAPI initialization in the `igb` driver

The `igb` driver does this via a long call chain:

1. `igb_probe` calls `igb_sw_init`.
2. `igb_sw_init` calls `igb_init_interrupt_scheme`.
3. `igb_init_interrupt_scheme` calls `igb_alloc_q_vectors`.
4. `igb_alloc_q_vectors` calls `igb_alloc_q_vector`.
5. `igb_alloc_q_vector` calls `netif_napi_add`.

This call trace results in a few high level things happening:

1. If MSI-X is supported, it will be enabled with a call to `pci_enable_msix`.
2. Various settings are computed and initialized; most notably the number of transmit and receive queues that the device and driver will use for sending and receiving packets.
3. `igb_alloc_q_vector` is called once for every transmit and receive queue that will be created.
4. Each call to `igb_alloc_q_vector` calls `netif_napi_add` to register a `poll` function for that queue and an instance of `struct napi_struct` that will be passed to `poll` when called to harvest packets.

Let's take a look at `igb_alloc_q_vector` to see how the `poll` callback and its private data are registered.

From drivers/net/ethernet/intel/igb/igb_main.c:

```
static int igb_alloc_q_vector(struct igb_adapter *adapter,
                              int v_count, int v_idx,
                              int txr_count, int txr_idx,
                              int rxr_count, int rxr_idx)
{
  /* ... */

  /* allocate q_vector and rings */
  q_vector = kzalloc(size, GFP_KERNEL);
  if (!q_vector)
          return -ENOMEM;

  /* initialize NAPI */
```

```
  netif_napi_add(adapter->netdev, &q_vector->napi, igb_poll, 64);

  /* ... */
```

The above code is allocation memory for a receive queue and registering the function `igb_poll` with the NAPI subsystem. It provides a reference to the `struct napi_struct` associated with this newly created RX queue (`&q_vector->napi` above). This will be passed into `igb_poll` when called by the NAPI subsystem when it comes time to harvest packets from this RX queue.

This will be important later when we examine the flow of data from drivers up the network stack.

Bringing a network device up

Recall the `net_device_ops` structure we saw earlier which registered a set of functions for bringing the network device up, transmitting packets, setting the MAC address, etc.

When a network device is brought up (for example, with `ifconfig eth0 up`), the function attached to the `ndo_open` field of the `net_device_ops` structure is called.

The `ndo_open` function will typically do things like:

1. Allocate RX and TX queue memory
2. Enable NAPI
3. Register an interrupt handler
4. Enable hardware interrupts
5. And more.

In the case of the `igb` driver, the function attached to the `ndo_open` field of the `net_device_ops` structure is called `igb_open`.

Preparing to receive data from the network

Most NICs you'll find today will use DMA to write data directly into RAM where the OS can retrieve the data for processing. The data structure most NICs use for this purpose resembles a queue built on circular buffer (or a ring buffer).

In order to do this, the device driver must work with the OS to reserve a region of memory that the NIC hardware can use. Once this region is reserved, the hardware is informed of its location and incoming data will be written to RAM where it will later be picked up and processed by the networking subsystem.

This seems simple enough, but what if the packet rate was high enough that a single CPU was not able to properly process all incoming packets? The data structure is built on a fixed length region of memory, so incoming packets would be dropped.

This is where something known as known as Receive Side Scaling (RSS) or multiqueue can help.

Some devices have the ability to write incoming packets to several different regions of RAM simultaneously; each region is a separate queue. This allows the OS to use multiple CPUs to process incoming data in parallel, starting at the hardware level. This feature is not supported by all NICs.

The Intel I350 NIC does support multiple queues. We can see evidence of this in the `igb` driver. One of the first things the `igb` driver does when it is brought up is call a function named `igb_setup_all_rx_resources`. This function calls another function, `igb_setup_rx_resources`, once for each RX queue to arrange for DMA-able memory where the device will write incoming data.

If you are curious how exactly this works, please see the Linux kernel's DMA API HOWTO.

It turns out the number and size of the RX queues can be tuned by using `ethtool`. Tuning these values can have a noticeable impact on the number of frames which are processed vs the number of frames which are dropped.

The NIC uses a hash function on the packet header fields (like source, destination, port, etc) to determine which RX queue the data should be directed to.

Some NICs let you adjust the weight of the RX queues, so you can send more traffic to specific queues.

Fewer NICs let you adjust this hash function itself. If you can adjust the hash function, you can send certain flows to specific RX queues for processing or even drop the packages at the hardware level, if desired.

We'll take a look at how to tune these settings shortly.

Enable NAPI

When a network device is brought up, a driver will usually enable NAPI.

We saw earlier how drivers register `poll` functions with NAPI, but NAPI is not usually enabled until the device is brought up.

Enabling NAPI is relatively straight forward. A call to `napi_enable` will flip a bit in the `struct napi_struct` to indicate that it is now enabled. As mentioned above, while NAPI will be enabled it will be in the off position.

In the case of the `igb` driver, NAPI is enabled for each `q_vector` that was initialized when the driver was loaded or when the queue count or size are changed with `ethtool`.

From drivers/net/ethernet/intel/igb/igb_main.c:

```
for (i = 0; i < adapter->num_q_vectors; i++)
  napi_enable(&(adapter->q_vector[i]->napi));
```


Register an interrupt handler

After enabling NAPI, the next step is to register an interrupt handler. There are different methods a device can use to signal an interrupt: MSI-X, MSI, and legacy interrupts. As such, the code differs from device to device depending on what the supported interrupt methods are for a particular piece of hardware.

The driver must determine which method is supported by the device and register the appropriate handler function that will execute when the interrupt is received.

Some drivers, like the `igb` driver, will try to register an interrupt handler with each method, falling back to the next untested method on failure.

MSI-X interrupts are the preferred method, especially for NICs that support multiple RX queues. This is because each RX queue can have its own hardware interrupt assigned, which can then be handled by a specific CPU (with `irqbalance` or by modifying `/proc/irq/IRQ_NUMBER/smp_affinity`). As we'll see shortly, the CPU that handles the interrupt will be the CPU that processes the packet. In this way, arriving packets can be processed by separate CPUs from the hardware interrupt level up through the networking stack.

If MSI-X is unavailable, MSI still presents advantages over legacy interrupts and will be used by the driver if the device supports it. Read this useful wiki page for more information about MSI and MSI-X.

In the `igb` driver, the functions `igb_msix_ring`, `igb_intr_msi`, `igb_intr` are the interrupt handler methods for the MSI-X, MSI, and legacy interrupt modes, respectively.

You can find the code in the driver which attempts each interrupt method in drivers/net/ethernet/intel/igb/igb_main.c:

```
static int igb_request_irq(struct igb_adapter *adapter)
{
  struct net_device *netdev = adapter->netdev;
  struct pci_dev *pdev = adapter->pdev;
  int err = 0;

  if (adapter->msix_entries) {
    err = igb_request_msix(adapter);
    if (!err)
      goto request_done;
    /* fall back to MSI */

    /* ... */
  }

  /* ... */

  if (adapter->flags & IGB_FLAG_HAS_MSI) {
    err = request_irq(pdev->irq, igb_intr_msi, 0,
          netdev->name, adapter);
    if (!err)
      goto request_done;

    /* fall back to legacy interrupts */

    /* ... */
  }

  err = request_irq(pdev->irq, igb_intr, IRQF_SHARED,
        netdev->name, adapter);

  if (err)
    dev_err(&pdev->dev, "Error %d getting interrupt\n", err);
```

```
request_done:
  return err;
}
```

As you can see in the abbreviated code above, the driver first attempts to set an MSI-X interrupt handler with `igb_request_msix`, falling back to MSI on failure. Next, `request_irq` is used to register `igb_intr_msi`, the MSI interrupt handler. If this fails, the driver falls back to legacy interrupts. `request_irq` is used again to register the legacy interrupt handler `igb_intr`.

And this is how the `igb` driver registers a function that will be executed when the NIC raises an interrupt signaling that data has arrived and is ready for processing.

Enable Interrupts

At this point, almost everything is setup. The only thing left is to enable interrupts from the NIC and wait for data to arrive. Enabling interrupts is hardware specific, but the `igb` driver does this in `__igb_open` by calling a helper function named `igb_irq_enable`.

Interrupts are enabled for this device by writing to registers:

```
static void igb_irq_enable(struct igb_adapter *adapter)
{

  /* ... */

    wr32(E1000_IMS, IMS_ENABLE_MASK | E1000_IMS_DRSTA);
    wr32(E1000_IAM, IMS_ENABLE_MASK | E1000_IMS_DRSTA);

  /* ... */
}
```

The network device is now up

Drivers may do a few more things like start timers, work queues, or other hardware-specific setup. Once that is completed. the network device is up and ready for use.

Let's take a look at monitoring and tuning settings for network device drivers.

Monitoring network devices

There are several different ways to monitor your network devices offering different levels of granularity and complexity. Let's start with most granular and move to least granular.

Using ethtool -S

You can install `ethtool` on an Ubuntu system by running: `sudo apt-get install ethtool`.

Once it is installed, you can access the statistics by passing the `-S` flag along with the name of the network device you want statistics about.

Monitor detailed NIC device statistics (e.g., packet drops) with `ethtool -S`.

```
$ sudo ethtool -S eth0
NIC statistics:
     rx_packets: 597028087
     tx_packets: 5924278060
     rx_bytes: 112643393747
     tx_bytes: 990080156714
     rx_broadcast: 96
     tx_broadcast: 116
     rx_multicast: 20294528
     ....
```

Monitoring this data can be difficult. It is easy to obtain, but there is no standardization of the field values. Different drivers, or even different versions of the *same* driver might produce different field names that have the same meaning.

You should look for values with "drop", "buffer", "miss", etc in the label. Next, you will have to read your driver source. You'll be able to determine which values are accounted for totally in software (e.g., incremented when there is no memory) and which values come directly from hardware via a register read. In the case of a register value, you should consult the data sheet for your hardware to determine what the meaning of the counter really is; many of the labels given via `ethtool` can be misleading.

Using sysfs

sysfs also provides a lot of statistics values, but they are slightly higher level than the direct NIC level stats provided.

You can find the number of dropped incoming network data frames for, e.g. eth0 by using `cat` on a file.

Monitor higher level NIC statistics with sysfs.

```
$ cat /sys/class/net/eth0/statistics/rx_dropped
2
```

The counter values will be split into files like `collisions`, `rx_dropped`, `rx_errors`, `rx_missed_errors`, etc.

Unfortunately, it is up to the drivers to decide what the meaning of each field is, and thus, when to increment them and where the values come from. You may notice that some drivers count a certain type of error condition as a drop, but other drivers may count the same as a miss.

If these values are critical to you, you will need to read your driver source to understand exactly what your driver thinks each of these values means.

Using `/proc/net/dev`

An even higher level file is `/proc/net/dev` which provides high-level summary-esque information for each network adapter on the system.

Monitor high level NIC statistics by reading `/proc/net/dev`.

```
$ cat /proc/net/dev
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    p
ackets errs drop fifo colls carrier compressed
  eth0: 110346752214 597737500    0    2    0     0          0  20963860 990
```

```
024805984 6066582604     0     0     0      0         0           0
    lo: 428349463836 1579868535     0     0     0      0           0          0 428
349463836 1579868535     0     0     0      0         0           0
```

This file shows a subset of the values you'll find in the sysfs files mentioned above, but it may serve as a useful general reference.

The caveat mentioned above applies here, as well: if these values are important to you, you will still need to read your driver source to understand exactly when, where, and why they are incremented to ensure your understanding of an error, drop, or fifo are the same as your driver.

Tuning network devices

Check the number of RX queues being used

If your NIC and the device driver loaded on your system support RSS / multiqueue, you can usually adjust the number of RX queues (also called RX channels), by using `ethtool`.

Check the number of NIC receive queues with `ethtool`

```
$ sudo ethtool -l eth0
Channel parameters for eth0:
Pre-set maximums:
RX:    0
TX:    0
Other:    0
Combined: 8
Current hardware settings:
RX:    0
TX:    0
Other:    0
Combined: 4
```

This output is displaying the pre-set maximums (enforced by the driver and the hardware) and the current settings.

Note: not all device drivers will have support for this operation.

Error seen if your NIC doesn't support this operation.

```
$ sudo ethtool -l eth0
Channel parameters for eth0:
Cannot get device channel parameters
: Operation not supported
```

This means that your driver has not implemented the ethtool `get_channels` operation. This could be because the NIC doesn't support adjusting the number of queues, doesn't support RSS / multiqueue, or your driver has not been updated to handle this feature.

Adjusting the number of RX queues

Once you've found the current and maximum queue count, you can adjust the values by using `sudo ethtool -L`.

Note: some devices and their drivers only support combined queues that are paired for transmit and receive, as in the example in the above section.

Set combined NIC transmit and receive queues to 8 with `ethtool -L`

```
$ sudo ethtool -L eth0 combined 8
```

If your device and driver support individual settings for RX and TX and you'd like to change only the RX queue count to 8, you would run:

Set the number of NIC receive queues to 8 with `ethtool -L`.

```
$ sudo ethtool -L eth0 rx 8
```

Note: making these changes will, for most drivers, take the interface down and then bring it back up; connections to this interface will be interrupted. This may not matter much for a one-time change, though.

Adjusting the size of the RX queues

Some NICs and their drivers also support adjusting the size of the RX queue. Exactly how this works is hardware specific, but luckily `ethtool` provides a generic way for users to adjust the size. Increasing the size of the RX queue can help prevent network data drops at the NIC during periods where large numbers of data frames are received. Data may still be dropped in software, though, and other tuning is required to reduce or eliminate drops completely.

Check current NIC queue sizes with `ethtool -g`

```
$ sudo ethtool -g eth0
Ring parameters for eth0:
Pre-set maximums:
RX:   4096
RX Mini:  0
RX Jumbo: 0
TX:   4096
Current hardware settings:
RX:   512
RX Mini:  0
RX Jumbo: 0
TX:   512
```

the above output indicates that the hardware supports up to 4096 receive and transmit descriptors, but it is currently only using 512.

Increase size of each RX queue to 4096 with `ethtool -G`

```
$ sudo ethtool -G eth0 rx 4096
```

Note: making these changes will, for most drivers, take the interface down and then bring it back up; connections to this interface will be interrupted. This may not matter much for a one-time change, though.

Adjusting the processing weight of RX queues

Some NICs support the ability to adjust the distribution of network data among the RX queues by setting a weight.

You can configure this if:

- Your NIC supports flow indirection.
- Your driver implements the `ethtool` functions `get_rxfh_indir_size` and `get_rxfh_indir`.
- You are running a new enough version of `ethtool` that has support for the command line options `-x` and `-X` to show and set the indirection table, respectively.

Check the RX flow indirection table with `ethtool -x`

```
$ sudo ethtool -x eth0
RX flow hash indirection table for eth3 with 2 RX ring(s):
0: 0 1 0 1 0 1 0 1
8: 0 1 0 1 0 1 0 1
16: 0 1 0 1 0 1 0 1
24: 0 1 0 1 0 1 0 1
```

This output shows packet hash values on the left, with receive queue 0 and 1 listed. So, a packet which hashes to 2 will be delivered to receive queue 0, while a packet which hashes to 3 will be delivered to receive queue 1.

Example: spread processing evenly between first 2 RX queues

```
$ sudo ethtool -X eth0 equal 2
```

If you want to set custom weights to alter the number of packets which hit certain receive queues (and thus CPUs), you can specify those on the command line, as well:

Set custom RX queue weights with `ethtool -X`

```
$ sudo ethtool -X eth0 weight 6 2
```

The above command specifies a weight of 6 for rx queue 0 and 2 for rx queue 1, pushing much more data to be processed on queue 0.

Some NICs will also let you adjust the fields which be used in the hash algorithm, as we'll see now.

Adjusting the rx hash fields for network flows

You can use `ethtool` to adjust the fields that will be used when computing a hash for use with RSS.

Check which fields are used for UDP RX flow hash with `ethtool -n`.

```
$ sudo ethtool -n eth0 rx-flow-hash udp4
UDP over IPV4 flows use these fields for computing Hash flow key:
IP SA
IP DA
```

For eth0, the fields that are used for computing a hash on UDP flows is the IPv4 source and destination addresses. Let's include the source and destination ports:

Set UDP RX flow hash fields with `ethtool -N`.

```
$ sudo ethtool -N eth0 rx-flow-hash udp4 sdfn
```

The `sdfn` string is a bit cryptic; check the `ethtool` man page for an explanation of each letter.

Adjusting the fields to take a hash on is useful, but `ntuple` filtering is even more useful for finer grained control over which flows will be handled by which RX queue.

ntuple filtering for steering network flows

Some NICs support a feature known as "ntuple filtering." This feature allows the user to specify (via `ethtool`) a set of parameters to use to filter incoming network data in hardware and queue it to a particular RX queue. For example, the user can specify that TCP packets destined to a particular port should be sent to RX queue 1.

On Intel NICs this feature is commonly known as Intel Ethernet Flow Director. Other NIC vendors may have other marketing names for this feature.

As we'll see later, ntuple filtering is a crucial component of another feature called Accelerated Receive Flow Steering (aRFS), which makes using ntuple much easier if your NIC supports it. aRFS will be covered later.

This feature can be useful if the operational requirements of the system involve maximizing data locality with the hope of increasing CPU cache hit rates when processing network data. For example consider the following configuration for a webserver running on port 80:

- A webserver running on port 80 is pinned to run on CPU 2.
- IRQs for an RX queue are assigned to be processed by CPU 2.
- TCP traffic destined to port 80 is 'filtered' with ntuple to CPU 2.
- All incoming traffic to port 80 is then processed by CPU 2 starting at data arrival to the userland program.
- Careful monitoring of the system including cache hit rates and networking stack latency will be needed to determine effectiveness.

As mentioned, ntuple filtering can be configured with `ethtool`, but first, you'll need to ensure that this feature is enabled on your device.

Check if ntuple filters are enabled with `ethtool -k`

```
$ sudo ethtool -k eth0
Offload parameters for eth0:
...
ntuple-filters: off
receive-hashing: on
```

As you can see, `ntuple-filters` are set to off on this device.

Enable ntuple filters with `ethtool -K`

```
$ sudo ethtool -K eth0 ntuple on
```

Once you've enabled ntuple filters, or verified that it is enabled, you can check the existing ntuple rules by using `ethtool`:

Check existing ntuple filters with `ethtool -u`

```
$ sudo ethtool -u eth0
40 RX rings available
Total 0 rules
```

As you can see, this device has no ntuple filter rules. You can add a rule by specifying it on the command line to `ethtool`. Let's add a rule to direct all TCP traffic with a destination port of 80 to RX queue 2:

Add ntuple filter to send TCP flows with destination port 80 to RX queue 2

```
$ sudo ethtool -U eth0 flow-type tcp4 dst-port 80 action 2
```

You can also use ntuple filtering to drop packets for particular flows at the hardware level. This can be useful for mitigating heavy incoming traffic from specific IP addresses. For more information about configuring ntuple filter rules, see the `ethtool` man page.

You can usually get statistics about the success (or failure) of your ntuple rules by checking values output from `ethtool -S [device name]`. For example, on Intel NICs, the statistics `fdir_match` and `fdir_miss` calculate the number of matches and misses for your ntuple filtering rules. Consult your device driver source and device data sheet for tracking down statistics counters (if available).

SoftIRQs

Before examining the network stack, we'll need to take a short detour to examine something in the Linux kernel called SoftIRQs.

What is a softirq?

The softirq system in the Linux kernel is a mechanism for executing code outside of the context of an interrupt handler implemented in a driver. This system is important because hardware interrupts may be disabled during all or part of the execution of an interrupt handler. The longer interrupts are disabled, the greater chance that events may be missed. So, it is important to defer any long running actions outside of the interrupt handler so that it can complete as quickly as possible and re-enable interrupts from the device.

There are other mechanisms that can be used for deferring work in the kernel, but for the purposes of the networking stack, we'll be looking at softirqs.

The softirq system can be imagined as a series of kernel threads (one per CPU) that run handler functions which have been registered for different softirq events. If you've ever looked at top and seen `ksoftirqd/0` in the list of kernel threads, you were looking at the softirq kernel thread running on CPU 0.

Kernel subsystems (like networking) can register a softirq handler by executing the `open_softirq` function. We'll see later how the networking system registers its softirq handlers. For now, let's learn a bit more about how softirqs work.

ksoftirqd

Since softirqs are so important for deferring the work of device drivers, you might imagine that the `ksoftirqd` process is spawned pretty early in the life cycle of the kernel and you'd be correct.

Looking at the code found in kernel/softirq.c reveals how the `ksoftirqd` system is initialized:

```
static struct smp_hotplug_thread softirq_threads = {
  .store              = &ksoftirqd,
  .thread_should_run  = ksoftirqd_should_run,
  .thread_fn          = run_ksoftirqd,
  .thread_comm        = "ksoftirqd/%u",
};

static __init int spawn_ksoftirqd(void)
{
  register_cpu_notifier(&cpu_nfb);

  BUG_ON(smpboot_register_percpu_thread(&softirq_threads));

  return 0;
}
early_initcall(spawn_ksoftirqd);
```

As you can see from the `struct smp_hotplug_thread` definition above, there are two function pointers being registered: `ksoftirqd_should_run` and `run_ksoftirqd`.

Both of these functions are called from kernel/smpboot.c as part of something which resembles an event loop.

The code in `kernel/smpboot.c` first calls `ksoftirqd_should_run` which determines if there are any pending softirqs and, if there are pending softirqs, `run_ksoftirqd` is executed. The `run_ksoftirqd` does some minor bookkeeping before it calls `__do_softirq`.

__do_softirq

The `__do_softirq` function does a few interesting things:

- determines which softirq is pending
- softirq time is accounted for statistics purposes
- softirq execution statistics are incremented
- the softirq handler for the pending softirq (which was registered with a call to `open_softirq`) is executed.

So, when you look at graphs of CPU usage and see `softirq` or `si` you now know that this is measuring the amount of CPU usage happening in a deferred work context.

Monitoring

/proc/softirqs

The `softirq` system increments statistic counters which can be read from `/proc/softirqs` Monitoring these statistics can give you a sense for the rate at which softirqs for various events are being generated.

Check softIRQ stats by reading `/proc/softirqs`.

```
$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3
          HI:           0          0          0          0
       TIMER: 2831512516 1337085411 1103326083 1423923272
      NET_TX:   15774435     779806     733217     749512
      NET_RX: 1671622615 1257853535 2088429526 2674732223
       BLOCK: 1800253852    1466177    1791366     634534
BLOCK_IOPOLL:           0          0          0          0
     TASKLET:          25          0          0          0
       SCHED: 2642378225 1711756029  629040543  682215771
     HRTIMER:    2547911    2046898    1558136    1521176
         RCU: 2056528783 4231862865 3545088730  844379888
```

This file can give you an idea of how your network receive (`NET_RX`) processing is currently distributed across your CPUs. If it is distributed unevenly, you will see a larger count value for some CPUs than others. This is one indicator that you might be able to benefit from Receive Packet Steering / Receive Flow Steering described below. Be careful using just this file when monitoring your performance: during periods of high network activity you would expect to see the rate `NET_RX` increments increase, but this isn't necessarily the case. It turns out that this is a bit nuanced, because there are additional tuning knobs in the network stack that can affect the rate at which `NET_RX` softirqs will fire, which we'll see soon.

You should be aware of this, however, so that if you adjust the other tuning knobs you will know to examine `/proc/softirqs` and expect to see a change.

Now, let's move on to the networking stack and trace how network data is received from top to bottom.

Linux network device subsystem

Now that we've taken a look in to how network drivers and softirqs work, let's see how the Linux network device subsystem is initialized. Then, we can follow the path of a packet starting with its arrival.

Initialization of network device subsystem

The network device (netdev) subsystem is initialized in the function `net_dev_init`. Lots of interesting things happen in this initialization function.

Initialization of `struct softnet_data` structures

`net_dev_init` creates a set of `struct softnet_data` structures for each CPU on the system. These structures will hold pointers to several important things for processing network data:

- List for NAPI structures to be registered to this CPU.
- A backlog for data processing.
- The processing `weight`.
- The receive offload structure list.
- Receive packet steering settings.
- And more.

Each of these will be examined in greater detail later as we progress up the stack.

Initialization of softirq handlers

`net_dev_init` registers a transmit and receive softirq handler which will be used to process incoming or outgoing network data. The code for this is pretty straight forward:

```
static int __init net_dev_init(void)
{
  /* ... */

  open_softirq(NET_TX_SOFTIRQ, net_tx_action);
  open_softirq(NET_RX_SOFTIRQ, net_rx_action);

 /* ... */
}
```

We'll see soon how the driver's interrupt handler will "raise" (or trigger) the `net_rx_action` function registered to the `NET_RX_SOFTIRQ` softirq.

Data arrives

At long last; network data arrives!

Assuming that the RX queue has enough available descriptors, the packet is written to RAM via DMA. The device then raises the interrupt that is assigned to it (or in the case of MSI-X, the interrupt tied to the rx queue the packet arrived on).

Interrupt handler

In general, the interrupt handler which runs when an interrupt is raised should try to defer as much processing as possible to happen outside the interrupt context. This is crucial because while an interrupt is being processed, other interrupts may be blocked.

Let's take a look at the source for the MSI-X interrupt handler; it will really help illustrate the idea that the interrupt handler does as little work as possible.

From drivers/net/ethernet/intel/igb/igb_main.c:

```
static irqreturn_t igb_msix_ring(int irq, void *data)
{
  struct igb_q_vector *q_vector = data;

  /* Write the ITR value calculated from the previous interrupt. */
  igb_write_itr(q_vector);

  napi_schedule(&q_vector->napi);

  return IRQ_HANDLED;
}
```

This interrupt handler is very short and performs 2 very quick operations before returning.

First, this function calls `igb_write_itr` which simply updates a hardware specific register. In this case, the register that is updated is one which is used to track the rate hardware interrupts are arriving.

This register is used in conjunction with a hardware feature called "Interrupt Throttling" (also called "Interrupt Coalescing") which can be used to to pace the delivery of interrupts to the CPU. We'll see soon how `ethtool` provides a mechanism for adjusting the rate at which IRQs fire.

Secondly, `napi_schedule` is called which wakes up the NAPI processing loop if it was not already active. Note that the NAPI processing loop executes in a softirq; the NAPI processing loop does not execute from the interrupt handler. The interrupt handler simply causes it to start executing if it was not already.

The actual code showing exactly how this works is important; it will guide our understanding of how network data is processed on multi-CPU systems.

NAPI and `napi_schedule`

Let's figure out how the `napi_schedule` call from the hardware interrupt handler works.

Remember, NAPI exists specifically to harvest network data without needing interrupts from the NIC to signal that data is ready for processing. As mentioned earlier, the NAPI `poll` loop is bootstrapped by receiving a hardware interrupt. In other words: NAPI is enabled, but off, until the first packet arrives at which point the NIC raises an IRQ and NAPI is started. There are a few other cases, as we'll see soon, where NAPI can be disabled and will need a hardware interrupt to be raised before it will be started again.

The NAPI poll loop is started when the interrupt handler in the driver calls `napi_schedule`. `napi_schedule` is actually just a wrapper function defined in a header file which calls down to `__napi_schedule`.

From net/core/dev.c:

```
/**
 * __napi_schedule - schedule for receive
 * @n: entry to schedule
 *
 * The entry's receive function will be scheduled to run
 */
void __napi_schedule(struct napi_struct *n)
{
  unsigned long flags;

  local_irq_save(flags);
  ____napi_schedule(&__get_cpu_var(softnet_data), n);
  local_irq_restore(flags);
}
EXPORT_SYMBOL(__napi_schedule);
```

This code is using `__get_cpu_var` to get the `softnet_data` structure that is registered to the current CPU. This `softnet_data` structure and the `struct napi_struct` structure handed up from the driver are passed into `____napi_schedule`. Wow, that's a lot of underscores ;)

Let's take a look at `____napi_schedule`, from net/core/dev.c:

```
/* Called with irq disabled */
static inline void ____napi_schedule(struct softnet_data *sd,
                                     struct napi_struct *napi)
{
  list_add_tail(&napi->poll_list, &sd->poll_list);
  __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

This code does two important things:

1. The `struct napi_struct` handed up from the device driver's interrupt handler code is added to the `poll_list` attached to the `softnet_data` structure associated with the current CPU.
2. `__raise_softirq_irqoff` is used to "raise" (or trigger) a NET_RX_SOFTIRQ softirq. This will cause the `net_rx_action` registered during the network device subsystem initialization to be executed, if it's not currently being executed.

As we'll see shortly, the softirq handler function `net_rx_action` will call the NAPI `poll` function to harvest packets.

A note about CPU and network data processing

Note that all the code we've seen so far to defer work from a hardware interrupt handler to a softirq has been using structures associated with the current CPU.

While the driver's IRQ handler itself does very little work itself, the softirq handler will execute on the same CPU as the driver's IRQ handler.

This why setting the CPU a particular IRQ will be handled by is important: that CPU will be used not only to execute the interrupt handler in the driver, but the same CPU will also be used when harvesting packets in a softirq via NAPI.

As we'll see later, things like Receive Packet Steering can distribute some of this work to other CPUs further up the network stack.

Monitoring network data arrival

Hardware interrupt requests

Note: monitoring hardware IRQs does not give a complete picture of packet processing health. Many drivers turn off hardware IRQs while NAPI is running, as we'll see later. It is one important part of your whole monitoring solution.

Check hardware interrupt stats by reading `/proc/interrupts`.

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
   0:          46          0          0          0 IR-IO-APIC-edge      timer
   1:           3          0          0          0 IR-IO-APIC-edge      i8042
  30: 3361234770          0          0          0 IR-IO-APIC-fasteoi   aacra
id
  64:          0          0          0          0 DMAR_MSI-edge      dmar0
  65:          1          0          0          0 IR-PCI-MSI-edge      eth0
  66:  863649703          0          0          0 IR-PCI-MSI-edge      eth0-
```

```
TxRx-0
  67:  986285573          0          0          0 IR-PCI-MSI-edge      eth0-
TxRx-1
  68:         45          0          0          0 IR-PCI-MSI-edge      eth0-
TxRx-2
  69:        394          0          0          0 IR-PCI-MSI-edge      eth0-
TxRx-3
 NMI:    9729927    4008190    3068645    3375402  Non-maskable interrupts
 LOC: 2913290785 1585321306 1495872829 1803524526  Local timer interrupts
```

You can monitor the statistics in `/proc/interrupts` to see how the number and rate of hardware interrupts change as packets arrive and to ensure that each RX queue for your NIC is being handled by an appropriate CPU. As we'll see shortly, this number only tells us how many hardware interrupts have happened, but it is *not* necessarily a good metric for understanding how much data has been received or processed as many drivers will disable NIC IRQs as part of their contract with the NAPI subsystem. Further, using interrupt coalescing will also affect the statistics gathered from this file. Monitoring this file can help you determine if the interrupt coalescing settings you select are actually working.r

To get a more complete picture of your network processing health, you'll need to monitor `/proc/softirqs` (as mentioned above) and additional files in `/proc` that we'll cover below.

Tuning network data arrival

Interrupt coalescing

Interrupt coalescing is a method of preventing interrupts from being raised by a device to a CPU until a specific amount of work or number of events are pending.

This can help prevent interrupt storms and can help increase throughput or latency, depending on the settings used. Fewer interrupts generated result in higher throughput, increased latency, and lower CPU usage. More interrupts generated result in the opposite: lower latency, lower throughput, but also increased CPU usage.

Historically, earlier versions of the `igb`, `e1000`, and other drivers included support for a parameter called `InterruptThrottleRate`. This parameter has been replaced in more recent drivers with a generic `ethtool` function.

Get the current IRQ coalescing settings with `ethtool -c`.

```
$ sudo ethtool -c eth0
Coalesce parameters for eth0:
Adaptive RX: off  TX: off
stats-block-usecs: 0
sample-interval: 0
pkt-rate-low: 0
pkt-rate-high: 0
...
```

`ethtool` provides a generic interface for setting various coalescing settings. Keep in mind, however, that not every device or driver will support every setting. You should check your driver documentation or driver source code to determine what is, or is not, supported. As per the ethtool documentation: "Anything not implemented by the driver

causes these values to be silently ignored."

One interesting option that some drivers support is "adaptive RX/TX IRQ coalescing." This option is typically implemented in hardware. The driver usually needs to do some work to inform the NIC that this feature is enabled and some bookkeeping as well (as seen in the `igb` driver code above).

The result of enabling adaptive RX/TX IRQ coalescing is that interrupt delivery will be adjusted to improve latency when packet rate is low and also improve throughput when packet rate is high.

Enable adaptive RX IRQ coalescing with `ethtool -C`

```
$ sudo ethtool -C eth0 adaptive-rx on
```

You can also use `ethtool -C` to set several options. Some of the more common options to set are:

- `rx-usecs`: How many usecs to delay an RX interrupt after a packet arrives.
- `rx-frames`: Maximum number of data frames to receive before an RX interrupt.
- `rx-usecs-irq`: How many usecs to delay an RX interrupt while an interrupt is being serviced by the host.
- `rx-frames-irq`: Maximum number of data frames to receive before an RX interrupt is generated while the system is servicing an interrupt.

And many, many more.

Reminder that your hardware and driver may only support a subset of the options listed above. You should consult your driver source code and your hardware data sheet for more information on supported coalescing options.

Unfortunately, the options you can set aren't well documented anywhere except in a header file. Check the source of include/uapi/linux/ethtool.h to find an explanation of each option supported by `ethtool` (but not necessarily your driver and NIC).

Note: while interrupt coalescing seems to be a very useful optimization at first glance, the rest of the networking stack internals also come into the fold when attempting to optimize. Interrupt coalescing can be useful in some cases, but you should ensure that the rest of your networking stack is also tuned properly. Simply modifying your coalescing settings alone will likely provide minimal benefit in and of itself.

Adjusting IRQ affinities

If your NIC supports RSS / multiqueue or if you are attempting to optimize for data locality, you may wish to use a specific set of CPUs for handling interrupts generated by your NIC.

Setting specific CPUs allows you to segment which CPUs will be used for processing which IRQs. These changes may affect how upper layers operate, as we've seen for the networking stack.

If you do decide to adjust your IRQ affinities, you should first check if you running the `irqbalance` daemon. This daemon tries to automatically balance IRQs to CPUs and it may overwrite your settings. If you are running `irqbalance`, you should either disable `irqbalance` or use the `--banirq` in conjunction with `IRQBALANCE_BANNED_CPUS` to let `irqbalance` know that it shouldn't touch a set of IRQs and CPUs that you want to assign yourself.

Next, you should check the file `/proc/interrupts` for a list of the IRQ numbers for each network RX queue for your NIC.

Finally, you can adjust the which CPUs each of those IRQs will be handled by modifying `/proc/irq/IRQ_NUMBER/smp_affinity` for each IRQ number.

You simply write a hexadecimal bitmask to this file to instruct the kernel which CPUs it should use for handling the IRQ.

Example: Set the IRQ affinity for IRQ 8 to CPU 0

```
$ sudo bash -c 'echo 1 > /proc/irq/8/smp_affinity'
```

Network data processing begins

Once the softirq code determines that a softirq is pending, begins processing, and executes `net_rx_action`, network data processing begins.

Let's take a look at portions of the `net_rx_action` processing loop to understand how it works, which pieces are tunable, and what can be monitored.

`net_rx_action` processing loop

`net_rx_action` begins the processing of packets from the memory the packets were DMA'd into by the device.

The function iterates through the list of NAPI structures that are queued for the current CPU, dequeuing each structure, and operating on it.

The processing loop bounds the amount of work and execution time that can be consumed by the registered NAPI `poll` functions. It does this in two ways:

1. By keeping track of a work `budget` (which can be adjusted), and
2. Checking the elapsed time

From net/core/dev.c:

```
while (!list_empty(&sd->poll_list)) {
    struct napi_struct *n;
    int work, weight;


    /* If softirq window is exhausted then punt.
     * Allow this to run for 2 jiffies since which will allow
     * an average latency of 1.5/HZ.
     */
    if (unlikely(budget <= 0 || time_after_eq(jiffies, time_limit)))
      goto softnet_break;
```

This is how the kernel prevents packet processing from consuming the entire CPU. The `budget` above is the total available budget that will be spent among each of the available NAPI structures registered to this CPU.

This is another reason why multiqueue NICs should have the IRQ affinity carefully tuned. Recall that the CPU which handles the IRQ from the device will be the CPU where the softirq handler will execute and, as a result, will also be the CPU where the above loop and budget computation runs.

Systems with multiple NICs each with multiple queues can end up in a situation where multiple NAPI structs are registered to the same CPU. Data processing for all NAPI structs on the same CPU spend from the same `budget`.

If you don't have enough CPUs to distribute your NIC's IRQs, you can consider increasing the `net_rx_action` `budget` to allow for more packet processing for each CPU. Increasing the budget will increase CPU usage (specifically `sitime` or `si` in `top` or other programs), but should reduce latency as data will be processed more promptly.

Note: the CPU will still be bounded by a time limit of 2 jiffies, regardless of the assigned budget.

NAPI poll function and weight

Recall that network device drivers use `netif_napi_add` for registering `poll` function. As we saw earlier in this post, the `igb` driver has a piece of code like this:

```
/* initialize NAPI */
 netif_napi_add(adapter->netdev, &q_vector->napi, igb_poll, 64);
```

This registers a NAPI structure with a hardcoded weight of 64. We'll see now how this is used in the `net_rx_action` processing loop.

From net/core/dev.c:

```
 weight = n->weight;


work = 0;
if (test_bit(NAPI_STATE_SCHED, &n->state)) {
        work = n->poll(n, weight);
        trace_napi_poll(n);
}


WARN_ON_ONCE(work > weight);


budget -= work;
```

This code obtains the weight which was registered to the NAPI struct (`64` in the above driver code) and passes it into the `poll` function which was also registered to the NAPI struct (`igb_poll` in the above code).

The `poll` function returns the number of data frames that were processed. This amount is saved above as `work`, which is then subtracted from the overall `budget`.

So, assuming:

1. You are using a weight of `64` from your driver (all drivers were hardcoded with this value in Linux 3.13.0), and
2. You have your `budget` set to the default of `300`

Your system would stop processing data when either:

1. The `igb_poll` function was called at most 5 times (less if no data to process as we'll see next), OR
2. At least 2 jiffies of time have elapsed.

The NAPI / network device driver contract

One important piece of information about the contract between the NAPI subsystem and device drivers which has not

been mentioned yet are the requirements around shutting down NAPI.

This part of the contract is as follows:

- If a driver's `poll` function consumes its entire weight (which is hardcoded to `64`) it must NOT modify NAPI state. The `net_rx_action` loop will take over.
- If a driver's `poll` function does NOT consume its entire weight, it must disable NAPI. NAPI will be re-enabled next time an IRQ is received and the driver's IRQ handler calls `napi_schedule`.

We'll see how `net_rx_action` deals with the first part of that contract now. Next, the `poll` function is examined, we'll see how the second part of that contract is handled.

Finishing the `net_rx_action` loop

The `net_rx_action` processing loop finishes up with one last section of code that deals with the first part of the NAPI contract explained in the previous section. From net/core/dev.c:

```
 /* Drivers must not modify the NAPI state if they
  * consume the entire weight.  In such cases this code
  * still "owns" the NAPI instance and therefore can
  * move the instance around on the list at-will.
  */
if (unlikely(work == weight)) {
  if (unlikely(napi_disable_pending(n))) {
    local_irq_enable();
    napi_complete(n);
    local_irq_disable();
  } else {
    if (n->gro_list) {
      /* flush too old packets
       * If HZ < 1000, flush all packets.
       */
      local_irq_enable();
      napi_gro_flush(n, HZ >= 1000);
      local_irq_disable();
    }
    list_move_tail(&n->poll_list, &sd->poll_list);
  }
}
```

If the entire work is consumed, there are two cases that `net_rx_action` handles:

1. The network device should be shutdown (e.g. because the user ran `ifconfig eth0 down`),
2. If the device is *not* being shutdown, check if there's a generic receive offload (GRO) list. If the timer tick rate is >= 1000, all GRO'd network flows that were recently updated will be flushed. We'll dig into GRO in detail later. Move the NAPI structure to the end of the list for this CPU so the next iteration of the loop will get the next NAPI structure registered.

And that is how the packet processing loop invokes the driver's registered `poll` function to process packets. As we'll see shortly, the `poll` function will harvest network data and send it up the stack to be processed.

Exiting the loop when limits are reached

The `net_rx_action` loop will exit when either:

- The poll list registered for this CPU has no more NAPI structures (`!list_empty(&sd->poll_list)`), or
- The remaining budget is <= 0, or
- The time limit of 2 jiffies has been reached

Here's this code we saw earlier again:

```
/* If softirq window is exhausted then punt.
 * Allow this to run for 2 jiffies since which will allow
 * an average latency of 1.5/HZ.
 */
if (unlikely(budget <= 0 || time_after_eq(jiffies, time_limit)))
  goto softnet_break;
```

If you follow the `softnet_break` label you stumble upon something interesting. From net/core/dev.c:

```
softnet_break:
  sd->time_squeeze++;
  __raise_softirq_irqoff(NET_RX_SOFTIRQ);
  goto out;
```

The `struct softnet_data` structure has some statistics incremented and the softirq `NET_RX_SOFTIRQ` is shut down. The `time_squeeze` field is a measure of the number of times `net_rx_action` had more work to do but either the budget was exhausted or the time limit was reached before it could be completed. This is a tremendously useful counter for understanding bottlenecks in network processing. We'll see shortly how to monitor this value. The `NET_RX_SOFTIRQ` is disabled to free up processing time for other tasks. This makes sense as this small stub of code is only executed when more work could have been done, but we don't want to monopolize the CPU.

Execution is then transferred to the `out` label. Execution can also make it to the `out` label if there were no more NAPI structures to process, in other words, there is more budget than there is network activity and all the drivers have shut NAPI off and there is nothing left for `net_rx_action` to do.

The `out` section does one important thing before returning from `net_rx_action`: it calls `net_rps_action_and_irq_enable`. This function serves an important purpose if Receive Packet Steering is enabled; it wakes up remote CPUs to start processing network data.

We'll see more about how RPS works later. For now, let's see how to monitor the health of the `net_rx_action` processing loop and move on to the inner working of NAPI `poll` functions so we can progress up the network stack.

NAPI poll

Recall in previous sections that device drivers allocate a region of memory for the device to perform DMA to incoming packets. Just as it is the responsibility of the driver to allocate those regions, it is also the responsibility of the driver to unmap those regions, harvest the data, and send it up the network stack.

Let's take a look at how the `igb` driver does this to get an idea of how this works in practice.

igb_poll

At long last, we can finally examine our friend `igb_poll`. It turns out the code for `igb_poll` is deceptively simple. Let's take a look. From drivers/net/ethernet/intel/igb/igb_main.c:

```
/**
 *  igb_poll - NAPI Rx polling callback
 *  @napi: napi polling structure
 *  @budget: count of how many packets we should handle
 **/
static int igb_poll(struct napi_struct *napi, int budget)
{
        struct igb_q_vector *q_vector = container_of(napi,
                                                     struct igb_q_vector,
                                                     napi);
        bool clean_complete = true;

#ifdef CONFIG_IGB_DCA
        if (q_vector->adapter->flags & IGB_FLAG_DCA_ENABLED)
                igb_update_dca(q_vector);
#endif

        /* ... */

        if (q_vector->rx.ring)
                clean_complete &= igb_clean_rx_irq(q_vector, budget);

        /* If all work not completed, return budget and keep polling */
        if (!clean_complete)
                return budget;

        /* If not enough Rx work done, exit the polling mode */
        napi_complete(napi);
        igb_ring_irq_enable(q_vector);
```

```
        return 0;
}
```

This code does a few interesting things:

- If Direct Cache Access (DCA) support is enabled in the kernel, the CPU cache is warmed so that accesses to the RX ring will hit CPU cache. You can read more about DCA in the Extras section at the end of this blog post.
- Next, `igb_clean_rx_irq` is called which does the heavy lifting, as we'll see next.
- Next, `clean_complete` is checked to determine if there was still more work that could have been done. If so, the `budget` (remember, this was hardcoded to `64`) is returned. As we saw earlier, `net_rx_action` will move this NAPI structure to the end of the poll list.
- Otherwise, the driver turns off NAPI by calling `napi_complete` and re-enables interrupts by calling `igb_ring_irq_enable`. The next interrupt that arrives will re-enable NAPI.

Let's see how `igb_clean_rx_irq` sends network data up the stack.

igb_clean_rx_irq

The `igb_clean_rx_irq` function is a loop which processes one packet at a time until the `budget` is reached or no additional data is left to process.

The loop in this function does a few important things:

1. Allocates additional buffers for receiving data as used buffers are cleaned out. Additional buffers are added `IGB_RX_BUFFER_WRITE` (16) at a time.
2. Fetch a buffer from the RX queue and store it in an `skb` structure.
3. Check if the buffer is an "End of Packet" buffer. If so, continue processing. Otherwise, continue fetching additional buffers from the RX queue, adding them to the `skb`. This is necessary if a received data frame is larger than the buffer size.
4. Verify that the layout and headers of the data are correct.
5. The number of bytes processed statistic counter is increased by `skb->len`.
6. Set the hash, checksum, timestamp, VLAN id, and protocol fields of the skb. The hash, checksum, timestamp, and VLAN id are provided by the hardware. If the hardware is signaling a checksum error, the `csum_error` statistic is incremented. If the checksum succeeded and the data is UDP or TCP data, the `skb` is marked as `CHECKSUM_UNNECESSARY`. If the checksum failed, the protocol stacks are left to deal with this packet. The protocol is computed with a call to `eth_type_trans` and stored in the `skb` struct.
7. The constructed `skb` is handed up the network stack with a call to `napi_gro_receive`.
8. The number of packets processed statistics counter is incremented.
9. The loop continues until the number of packets processed reaches the budget.

Once the loop terminates, the function assigns statistics counters for rx packets and bytes processed.

Now it's time to take two detours prior to proceeding up the network stack. First, let's see how to monitor and tune the network subsystem's softirqs. Next, let's talk about Generic Receive Offloading (GRO). After that, the rest of the networking stack will make more sense as we enter `napi_gro_receive`.

Monitoring network data processing

/proc/net/softnet_stat

As seen in the previous section, `net_rx_action` increments a statistic when exiting the `net_rx_action` loop and when additional work could have been done, but either the `budget` or the time limit for the softirq was hit. This statistic is tracked as part of the `struct softnet_data` associated with the CPU.

These statistics are output to a file in proc: `/proc/net/softnet_stat` for which there is, unfortunately, very little documentation. The fields in the file in proc are not labeled and could change between kernel releases.

In Linux 3.13.0, you can find which values map to which field in `/proc/net/softnet_stat` by reading the kernel source. From net/core/net-procfs.c:

```
seq_printf(seq,
      "%08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x\n",
      sd->processed, sd->dropped, sd->time_squeeze, 0,
      0, 0, 0, 0, /* was fastroute */
      sd->cpu_collision, sd->received_rps, flow_limit_count);
```

Many of these statistics have confusing names and are incremented in places where you might not expect. An explanation of when and where each of these is incremented will be provided as the network stack is examined. Since the `squeeze_time` statistic was seen in `net_rx_action`, I thought it made sense to document this file now.

Monitor network data processing statistics by reading `/proc/net/softnet_stat`.

```
$ cat /proc/net/softnet_stat
6dcad223 00000000 00000001 00000000 00000000 00000000 00000000 00000000 0000
0000 00000000
6f0e1565 00000000 00000002 00000000 00000000 00000000 00000000 00000000 0000
0000 00000000
660774ec 00000000 00000003 00000000 00000000 00000000 00000000 00000000 0000
0000 00000000
61c99331 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000
0000 00000000
6794b1b3 00000000 00000005 00000000 00000000 00000000 00000000 00000000 0000
0000 00000000
6488cb92 00000000 00000001 00000000 00000000 00000000 00000000 00000000 0000
0000 00000000
```

Important details about `/proc/net/softnet_stat`:

- Each line of `/proc/net/softnet_stat` corresponds to a `struct softnet_data` structure, of which there is 1 per CPU.
- The values are separated by a single space and are displayed in hexadecimal
- The first value, `sd->processed`, is the number of network frames processed. This can be more than the total number of network frames received if you are using ethernet bonding. There are cases where the ethernet bonding driver will trigger network data to be re-processed, which would increment the `sd->processed` count more than once for the same packet.
- The second value, `sd->dropped`, is the number of network frames dropped because there was no room on the processing queue. More on this later.
- The third value, `sd->time_squeeze`, is (as we saw) the number of times the `net_rx_action` loop terminated because the budget was consumed or the time limit was reached, but more work could have been. Increasing the `budget` as explained earlier can help reduce this.
- The next 5 values are always 0.
- The ninth value, `sd->cpu_collision`, is a count of the number of times a collision occurred when trying to obtain a device lock when transmitting packets. This article is about receive, so this statistic will not be seen below.
- The tenth value, `sd->received_rps`, is a count of the number of times this CPU has been woken up to process

packets via an Inter-processor Interrupt

- The last value, `flow_limit_count`, is a count of the number of times the flow limit has been reached. Flow limiting is an optional Receive Packet Steering feature that will be examined shortly.

If you decide to monitor this file and graph the results, you must be extremely careful that the ordering of these fields hasn't changed and that the meaning of each field has been preserved. You will need to read the kernel source to verify this.

Tuning network data processing

Adjusting the `net_rx_action` budget

You can adjust the `net_rx_action` budget, which determines how much packet processing can be spent among all NAPI structures registered to a CPU by setting a sysctl value named `net.core.netdev_budget`.

Example: set the overall packet processing budget to 600.

```
$ sudo sysctl -w net.core.netdev_budget=600
```

You may also want to write this setting to your `/etc/sysctl.conf` file so that changes persist between reboots.

The default value on Linux 3.13.0 is 300.

Generic Receive Offloading (GRO)

Generic Receive Offloading (GRO) is a software implementation of a hardware optimization that is known as Large Receive Offloading (LRO).

The main idea behind both methods is that reducing the number of packets passed up the network stack by combining "similar enough" packets together can reduce CPU usage. For example, imagine a case where a large file transfer is occurring and most of the packets contain chunks of data in the file. Instead of sending small packets up the stack one at a time, the incoming packets can be combined into one packet with a huge payload. That packet can then be passed up the stack. This allows the protocol layers to process a single packet's headers while delivering bigger chunks of data to the user program.

The problem with this sort of optimization is, of course, information loss. If a packet had some important option or flag set, that option or flag could be lost if the packet is coalesced into another. And this is exactly why most people don't use or encourage the use of LRO. LRO implementations, generally speaking, had very lax rules for coalescing packets.

GRO was introduced as an implementation of LRO in software, but with more strict rules around which packets can be coalesced.

By the way: if you have ever used `tcpdump` and seen unrealistically large incoming packet sizes, it is most likely because your system has GRO enabled. As you'll see soon, packet capture taps are inserted further up the stack, after GRO has already happened.

Tuning: Adjusting GRO settings with `ethtool`

You can use `ethtool` to check if GRO is enabled and also to adjust the setting.

Use `ethtool -k` to check your GRO settings.

```
$ ethtool -k eth0 | grep generic-receive-offload
generic-receive-offload: on
```

As you can see, on this system I have `generic-receive-offload` set to on.

Use `ethtool -K` to enable (or disable) GRO.

```
$ sudo ethtool -K eth0 gro on
```

Note: making these changes will, for most drivers, take the interface down and then bring it back up; connections to this interface will be interrupted. This may not matter much for a one-time change, though.

napi_gro_receive

The function `napi_gro_receive` deals processing network data for GRO (if GRO is enabled for the system) and sending the data up the stack toward the protocol layers. Much of this logic is handled in a function called `dev_gro_receive`.

dev_gro_receive

This function begins by checking if GRO is enabled and, if so, preparing to do GRO. In the case where GRO is enabled, a list of GRO offload filters is traversed to allow the higher level protocol stacks to act on a piece of data which is being considered for GRO. This is done so that the protocol layers can let the network device layer know if this packet is part of a network flow that is currently being receive offloaded and handle anything protocol specific that should happen for GRO. For example, the TCP protocol will need to decide if/when to ACK a packet that is being coalesced into an existing packet.

Here's the code from `net/core/dev.c` which does this:

```
list_for_each_entry_rcu(ptype, head, list) {
  if (ptype->type != type || !ptype->callbacks.gro_receive)
    continue;

  skb_set_network_header(skb, skb_gro_offset(skb));
  skb_reset_mac_len(skb);
  NAPI_GRO_CB(skb)->same_flow = 0;
  NAPI_GRO_CB(skb)->flush = 0;
  NAPI_GRO_CB(skb)->free = 0;

  pp = ptype->callbacks.gro_receive(&napi->gro_list, skb);
  break;
}
```

If the protocol layers indicated that it is time to flush the GRO'd packet, that is taken care of next. This happens with a call to `napi_gro_complete`, which calls a `gro_complete` callback for the protocol layers and then passes the packet up the stack by calling `netif_receive_skb`.

Here's the code from `net/core/dev.c` which does this:

```
if (pp) {
```

```
  struct sk_buff *nskb = *pp;

  *pp = nskb->next;
  nskb->next = NULL;
  napi_gro_complete(nskb);
  napi->gro_count--;
}
```

Next, if the protocol layers merged this packet to an existing flow, `napi_gro_receive` simply returns as there's nothing else to do.

If the packet was not merged and there are fewer than `MAX_GRO_SKBS` (8) GRO flows on the system, a new entry is added to the `gro_list` on the NAPI structure for this CPU.

Here's the code from `net/core/dev.c` which does this:

```
if (NAPI_GRO_CB(skb)->flush || napi->gro_count >= MAX_GRO_SKBS)
  goto normal;

napi->gro_count++;
NAPI_GRO_CB(skb)->count = 1;
NAPI_GRO_CB(skb)->age = jiffies;
skb_shinfo(skb)->gso_size = skb_gro_len(skb);
skb->next = napi->gro_list;
napi->gro_list = skb;
ret = GRO_HELD;
```

And that is how the GRO system in the Linux networking stack works.

napi_skb_finish

Once `dev_gro_receive` completes, `napi_skb_finish` is called which either frees unneeded data structures because a packet has been merged, or calls `netif_receive_skb` to pass the data up the network stack (because there were already `MAX_GRO_SKBS` flows being GRO'd).

Next, it's time for `netif_receive_skb` to see how data is handed off to the protocol layers. Before this can be examined, we'll need to take a look at Receive Packet Steering (RPS) first.

Receive Packet Steering (RPS)

Recall earlier how we discussed that network device drivers register a NAPI `poll` function. Each `NAPI` poller instance is executed in the context of a softirq of which there is one per CPU. Further recall that the CPU which the driver's IRQ handler runs on will wake its softirq processing loop to process packets.

In other words: a single CPU processes the hardware interrupt and polls for packets to process incoming data.

Some NICs (like the Intel I350) support multiple queues at the hardware level. This means incoming packets can be

DMA'd to a separate memory region for each queue, with a separate NAPI structure to manage polling this region, as well. Thus multiple CPUs will handle interrupts from the device and also process packets.

This feature is typically called Receive Side Scaling (RSS).

Receive Packet Steering (RPS) is a software implementation of RSS. Since it is implemented in software, this means it can be enabled for any NIC, even NICs which have only a single RX queue. However, since it is in software, this means that RPS can only enter into the flow after a packet has been harvested from the DMA memory region.

This means that you wouldn't notice a decrease in CPU time spent handling IRQs or the NAPI `poll` loop, but you can distribute the load for processing the packet after it's been harvested and reduce CPU time from there up the network stack.

RPS works by generating a hash for incoming data to determine which CPU should process the data. The data is then enqueued to the per-CPU receive network backlog to be processed. An Inter-processor Interrupt (IPI) is delivered to the CPU owning the backlog. This helps to kick-start backlog processing if it is not currently processing data on the backlog. The `/proc/net/softnet_stat` contains a count of the number of times each `softnet_data` struct has received an IPI (the `received_rps` field).

Thus, `netif_receive_skb` will either continue sending network data up the networking stack, or hand it over to RPS for processing on a different CPU.

Tuning: Enabling RPS

For RPS to work, it must be enabled in the kernel configuration (it is on Ubuntu for kernel 3.13.0), and a bitmask describing which CPUs should process packets for a given interface and RX queue.

You can find some documentation about these bitmasks in the kernel documentation.

In short, the bitmasks to modify are found in:

`/sys/class/net/DEVICE_NAME/queues/QUEUE/rps_cpus`

So, for `eth0` and receive queue 0, you would modify the file: `/sys/class/net/eth0/queues/rx-0/rps_cpus` with a hexadecimal number indicating which CPUs should process packets from `eth0`'s receive queue 0. As the documentation points out, RPS may be unnecessary in certain configurations.

> **Note:** enabling RPS to distribute packet processing to CPUs which were previously not processing packets will cause the number of `NET_RX` softirqs to increase for that CPU, as well as the `si` or `sitime` in the CPU usage graph. You can compare before and after of your softirq and CPU usage graphs to confirm that RPS is configured properly to your liking.

Receive Flow Steering (RFS)

Receive flow steering (RFS) is used in conjunction with RPS. RPS attempts to distribute incoming packet load amongst multiple CPUs, but does not take into account any data locality issues for maximizing CPU cache hit rates. You can use RFS to help increase cache hit rates by directing packets for the same flow to the same CPU for processing.

Tuning: Enabling RFS

For RFS to work, you must have RPS enabled and configured.

RFS keeps track of a global hash table of all flows and the size of this hash table can be adjusted by setting the

`net.core.rps_sock_flow_entries` sysctl.

Increase the size of the RFS socket flow hash by setting a `sysctl`.

```
$ sudo sysctl -w net.core.rps_sock_flow_entries=32768
```

Next, you can also set the number of flows per RX queue by writing this value to the sysfs file named `rps_flow_cnt` for each RX queue.

Example: increase the number of flows for RX queue 0 on eth0 to 2048.

```
$ sudo bash -c 'echo 2048 > /sys/class/net/eth0/queues/rx-0/rps_flow_cnt'
```

Hardware accelerated Receive Flow Steering (aRFS)

RFS can be sped up with the use of hardware acceleration; the NIC and the kernel can work together to determine which flows should be processed on which CPUs. To use this feature, it must be supported by the NIC and your driver.

Consult your NIC's data sheet to determine if this feature is supported. If your NIC's driver exposes a function called `ndo_rx_flow_steer`, then the driver has support for accelerated RFS.

Tuning: Enabling accelerated RFS (aRFS)

Assuming that your NIC and driver support it, you can enable accelerated RFS by enabling and configuring a set of things:

1. Have RPS enabled and configured.
2. Have RFS enabled and configure.
3. Your kernel has `CONFIG_RFS_ACCEL` enabled at compile time. The Ubuntu kernel 3.13.0 does.
4. Have ntuple support enabled for the device, as described previously. You can use `ethtool` to verify that ntuple support is enabled for the device.
5. Configure your IRQ settings to ensure each RX queue is handled by one of your desired network processing CPUs.

Once the above is configured, accelerated RFS will be used to automatically move data to the RX queue tied to a CPU core that is processing data for that flow and you won't need to specify an ntuple filter rule manually for each flow.

Moving up the network stack with `netif_receive_skb`

Picking up where we left off with `netif_receive_skb`, which is called from a few places. The two most common (and also the two we've already looked at):

- `napi_skb_finish` if the packet is not going to be merged to an existing GRO'd flow, OR
- `napi_gro_complete` if the protocol layers indicated that it's time to flush the flow, OR

Reminder: `netif_receive_skb` and its descendants are operating in the context of a the softirq processing loop and you'll see the time spent here accounted for as `sitime` or `si` with tools like `top`.

`netif_receive_skb` begins by first checking a `sysctl` value to determine if the user has requested receive timestamping before or after a packet hits the backlog queue. If this setting is enabled, the data is timestamped now,

prior to it hitting RPS (and the CPU's associated backlog queue). If this setting is disabled, it will be timestamped after it hits the queue. This can be used to distribute the load of timestamping amongst multiple CPUs if RPS is enabled, but will introduce some delay as a result.

Tuning: RX packet timestamping

You can tune when packets will be timestamped after they are received by adjusting a sysctl named `net.core.netdev_tstamp_prequeue`:

Disable timestamping for RX packets by adjusting a `sysctl`

```
$ sudo sysctl -w net.core.netdev_tstamp_prequeue=0
```

The default value is 1. Please see the previous section for an explanation as to what this setting means, exactly.

netif_receive_skb

After the timestamping is dealt with, `netif_receive_skb` operates differently depending on whether or not RPS is enabled. Let's start with the simpler path first: RPS disabled.

Without RPS (default setting)

If RPS is not enabled, `__netif_receive_skb` is called which does some bookkeeping and then calls `__netif_receive_skb_core` to move data closer to the protocol stacks.

We'll see precisely how `__netif_receive_skb_core` works, but first let's see how the RPS enabled code path works, as that code will also call `__netif_receive_skb_core`.

With RPS enabled

If RPS is enabled, after the timestamping options mentioned above are dealt with, `netif_receive_skb` will perform some computations to determine which CPU's backlog queue should be used. This is done by using the function `get_rps_cpu`. From net/core/dev.c:

```
 cpu = get_rps_cpu(skb->dev, skb, &rflow);


if (cpu >= 0) {
  ret = enqueue_to_backlog(skb, cpu, &rflow->last_qtail);
  rcu_read_unlock();
  return ret;
}
```

`get_rps_cpu` will take into account RFS and aRFS settings as described above to ensure the the data gets queued to the desired CPU's backlog with a call to `enqueue_to_backlog`.

enqueue_to_backlog

This function begins by getting a pointer to the remote CPU's `softnet_data` structure, which contains a pointer to the `input_pkt_queue`. Next, the queue length of the `input_pkt_queue` of the remote CPU is checked. From net/core/dev.c:

```
qlen = skb_queue_len(&sd->input_pkt_queue);
if (qlen <= netdev_max_backlog && !skb_flow_limit(skb, qlen)) {
```

The length of `input_pkt_queue` is first compared to `netdev_max_backlog`. If the queue is longer than this value, the data is dropped. Similarly, the flow limit is checked and if it has been exceeded, the data is dropped. In both cases the drop count on the `softnet_data` structure is incremented. Note that this is the `softnet_data` structure of the CPU the data was going to be queued to. Read the section above about `/proc/net/softnet_stat` to learn how to get the drop count for monitoring purposes.

`enqueue_to_backlog` is not called in many places. It is called for RPS-enabled packet processing and also from `netif_rx`. Most drivers should not be using `netif_rx` and should instead be using `netif_receive_skb`. If you are not using RPS and your driver is not using `netif_rx`, increasing the backlog won't produce any noticeable effect on your system as it is not used.

> **Note**: You need to check the driver you are using. If it calls `netif_receive_skb` and you are **not** using RPS, increasing the `netdev_max_backlog` will not yield any performance improvement because no data will ever make it to the `input_pkt_queue`.

Assuming that the `input_pkt_queue` is small enough and the flow limit (more about this, next) hasn't been reached (or is disabled), the data can be queued. The logic here is a bit funny, but can be summarized as:

- If the queue is empty: check if NAPI has been started on the remote CPU. If not, check if an IPI is queued to be sent. If not, queue one and start the NAPI processing loop by calling `____napi_schedule`. Proceed to queuing the data.
- If the queue is not empty, or the previously described operation has completed, enqueue the data.

The code is a bit tricky with its use of `goto`, so read it carefully. From net/core/dev.c:

```
if (skb_queue_len(&sd->input_pkt_queue)) {
enqueue:
        __skb_queue_tail(&sd->input_pkt_queue, skb);
        input_queue_tail_incr_save(sd, qtail);
        rps_unlock(sd);
        local_irq_restore(flags);
        return NET_RX_SUCCESS;
 }

 /* Schedule NAPI for backlog device
  * We can use non atomic operation since we own the queue lock
  */
```

```
if (!__test_and_set_bit(NAPI_STATE_SCHED, &sd->backlog.state)) {
        if (!rps_ipi_queued(sd))
                ____napi_schedule(sd, &sd->backlog);
}
goto enqueue;
```

Flow limits

RPS distributes packet processing load amongst multiple CPUs, but a single large flow can monopolize CPU processing time and starve smaller flows. Flow limits are a feature that can be used to limit the number of packets queued to the backlog for each flow to a certain amount. This can help ensure that smaller flows are processed even though much larger flows are pushing packets in.

The if statement above from net/core/dev.c checks the flow limit with a call to `skb_flow_limit`:

```
if (qlen <= netdev_max_backlog && !skb_flow_limit(skb, qlen)) {
```

This code is checking that there is still room in the queue and that the flow limit has not been reached. By default, flow limits are disabled. In order to enable flow limits, you must specify a bitmap (similar to RPS' bitmap).

Monitoring: Monitor drops due to full `input_pkt_queue` or flow limit

See the section above about monitoring `/proc/net/softnet_stat`. The `dropped` field is a counter that gets incremented each time data is dropped instead of queued to a CPU's `input_pkt_queue`.

Tuning

Tuning: Adjusting `netdev_max_backlog` to prevent drops

Before adjusting this tuning value, see the note in the previous section.

You can help prevent drops in `enqueue_to_backlog` by increasing the `netdev_max_backlog` if you are using RPS or if your driver calls `netif_rx`.

Example: increase backlog to 3000 with `sysctl`.

```
$ sudo sysctl -w net.core.netdev_max_backlog=3000
```

The default value is 1000.

Tuning: Adjust the NAPI weight of the backlog `poll` loop

You can adjust the weight of the backlog's NAPI poller by setting the `net.core.dev_weight` sysctl. Adjusting this value determines how much of the overall budget the backlog `poll` loop can consume (see the section above about adjusting `net.core.netdev_budget`):

Example: increase the NAPI `poll` backlog processing loop with `sysctl`.

```
$ sudo sysctl -w net.core.dev_weight=600
```

The default value is 64.

Remember, backlog processing runs in the softirq context similar to the device driver's registered `poll` function and will be limited by the overall `budget` and a time limit, as described in previous sections.

Tuning: Enabling flow limits and tuning flow limit hash table size

Set the size of the flow limit table with a `sysctl`.

```
$ sudo sysctl -w net.core.flow_limit_table_len=8192
```

The default value is 4096.

This change only affects newly allocated flow hash tables. So, if you'd like to increase the table size, you should do it before you enable flow limits.

To enable flow limits you should specify a bitmask in `/proc/sys/net/core/flow_limit_cpu_bitmap` similar to the RPS bitmask which indicates which CPUs have flow limits enabled.

backlog queue NAPI poller

The per-CPU backlog queue plugs into NAPI the same way a device driver does. A `poll` function is provided that is used to process packets from the softirq context. A `weight` is also provided, just as a device driver would.

This NAPI struct is provided during initialization of the networking system. From `net_dev_init` in `net/core/dev.c`:

```
 sd->backlog.poll = process_backlog;
sd->backlog.weight = weight_p;
sd->backlog.gro_list = NULL;
sd->backlog.gro_count = 0;
```

The backlog NAPI structure differs from the device driver NAPI structure in that the `weight` parameter is adjustable, where as drivers hardcode their NAPI weight to 64. We'll see in the tuning section below how to adjust the weight using a `sysctl`.

process_backlog

The `process_backlog` function is a loop which runs until its weight (as described in the previous section) has been consumed or no more data remains on the backlog.

Each piece of data on the backlog queue is removed from the backlog queue and passed on to `__netif_receive_skb`. The code path once the data hits `__netif_receive_skb` is the same as explained above for the RPS disabled case. Namely, `__netif_receive_skb` does some bookkeeping prior to calling `__netif_receive_skb_core` to pass network data up to the protocol layers.

`process_backlog` follows the same contract with NAPI that device drivers do, which is: NAPI is disabled if the total weight will not be used. The poller is restarted with the call to `____napi_schedule` from `enqueue_to_backlog` as described above.

The function returns the amount of work done, which `net_rx_action` (described above) will subtract from the budget (which is adjusted with the `net.core.netdev_budget`, as described above).

__netif_receive_skb_core delivers data to packet taps and protocol layers

`__netif_receive_skb_core` performs the heavy lifting of delivering the data to protocol stacks. Before it does this, it checks if any packet taps have been installed which are catching all incoming packets. One example of something that does this is the `AF_PACKET` address family, typically used via the libpcap library.

If such a tap exists, the data is delivered there first then to the protocol layers next.

Packet tap delivery

If a packet tap is installed (usually via libpcap), the packet is delivered there with the following code from net/core/dev.c:

```
list_for_each_entry_rcu(ptype, &ptype_all, list) {
  if (!ptype->dev || ptype->dev == skb->dev) {
    if (pt_prev)
      ret = deliver_skb(skb, pt_prev, orig_dev);
    pt_prev = ptype;
  }
}
```

If you are curious about how the path of the data through pcap, read net/packet/af_packet.c.

Protocol layer delivery

Once the taps have been satisfied, `__netif_receive_skb_core` delivers data to protocol layers. It does this by obtaining the protocol field from the data and iterating across a list of deliver functions registered for that protocol type.

This can be seen in `__netif_receive_skb_core` in net/core/dev.c:

```
 type = skb->protocol;
list_for_each_entry_rcu(ptype,
                &ptype_base[ntohs(type) & PTYPE_HASH_MASK], list) {
        if (ptype->type == type &&
            (ptype->dev == null_or_dev || ptype->dev == skb->dev ||
             ptype->dev == orig_dev)) {
                if (pt_prev)
                        ret = deliver_skb(skb, pt_prev, orig_dev);
                pt_prev = ptype;
        }
}
```

The `ptype_base` identifier above is defined as a hash table of lists in net/core/dev.c:

```
struct list_head ptype_base[PTYPE_HASH_SIZE] __read_mostly;
```

Each protocol layer adds a filter to a list at a given slot in the hash table, computed with a helper function called `ptype_head`:

```
static inline struct list_head *ptype_head(const struct packet_type *pt)
{
        if (pt->type == htons(ETH_P_ALL))
                return &ptype_all;
        else
                return &ptype_base[ntohs(pt->type) & PTYPE_HASH_MASK];
}
```

Adding a filter to the list is accomplished with a call to `dev_add_pack`. That is how protocol layers register themselves for network data delivery for their protocol type.

And now you know how network data gets from the NIC to the protocol layer.

Protocol layer registration

Now that we know how data is delivered to the protocol stacks from the network device subsystem, let's see how a protocol layer registers itself.

This blog post is going to examine the IP protocol stack as it is a commonly used protocol and will be relevant to most readers.

IP protocol layer

The IP protocol layer plugs itself into the `ptype_base` hash table so that data will be delivered to it from the network device layer described in previous sections.

This happens in the function `inet_init` from net/ipv4/af_inet.c:

```
dev_add_pack(&ip_packet_type);
```

This registers the IP packet type structure defined at net/ipv4/af_inet.c:

```
static struct packet_type ip_packet_type __read_mostly = {
        .type = cpu_to_be16(ETH_P_IP),
        .func = ip_rcv,
};
```

`__netif_receive_skb_core` calls `deliver_skb` (as seen in the above section), which calls `func` (in this case, `ip_rcv`).

ip_rcv

The `ip_rcv` function is pretty straight-forward at a high level. There are several integrity checks to ensure the data is valid. Statistics counters are bumped, as well.

`ip_rcv` ends by passing the packet to `ip_rcv_finish` by way of netfilter. This is done so that any iptables rules that should be matched at the IP protocol layer can take a look at the packet before it continues on.

We can see the code which hands the data over to netfilter at the end of `ip_rcv` in net/ipv4/ip_input.c:

```
return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL, ip_rcv_finish);
```

netfilter and iptables

In the interest of brevity (and my RSI), I've decided to skip my deep dive into netfilter, iptables, and conntrack.

The short version is that `NF_HOOK_THRESH` will check if any filters are installed and attempt to return execution back to the IP protocol layer to avoid going deeper into netfilter and anything that hooks in below that like iptables and conntrack.

Keep in mind: if you have numerous or very complex netfilter or iptables rules, those rules will be executed in the softirq context and can lead to latency in your network stack. This may be unavoidable, though, if you need to have a particular set of rules installed.

ip_rcv_finish

Once net filter has had a chance to take a look at the data and decide what to do with it, `ip_rcv_finish` is called. This only happens if the data is not being dropped by netfilter, of course.

`ip_rcv_finish` begins with an optimization. In order to deliver the packet to proper place, a `dst_entry` from the routing system needs to be in place. In order to obtain one, the code initially attempts to call the `early_demux` function from the higher level protocol that this data is destined for.

The `early_demux` routine is an optimization which attempts to find the `dst_entry` needed to deliver the packet by checking if a `dst_entry` is cached on the socket structure.

Here's what that looks like from net/ipv4/ip_input.c:

```
if (sysctl_ip_early_demux && !skb_dst(skb) && skb->sk == NULL) {
  const struct net_protocol *ipprot;
  int protocol = iph->protocol;

  ipprot = rcu_dereference(inet_protos[protocol]);
  if (ipprot && ipprot->early_demux) {
    ipprot->early_demux(skb);
    /* must reload iph, skb->head might have changed */
    iph = ip_hdr(skb);
  }
}
```

As you can see above, this code is guarded by a sysctl `sysctl_ip_early_demux`. By default `early_demux` is enabled. The next section includes information about how to disable it and why you might want to.

If the optimization is enabled and there is no cached entry (because this is the first packet arriving), the packet will be handed off to the routing system in the kernel where the `dst_entry` will be computed and assigned.

Once the routing layer completes, statistics counters are updated and the function ends by calling `dst_input(skb)` which in turn calls the input function pointer on the packet's `dst_entry` structure that was affixed by the routing system.

If the packet's final destination is the local system, the routing system will attach the function `ip_local_deliver` to the input function pointer in the `dst_entry` structure on the packet.

Tuning: adjusting IP protocol early demux

Disable the `early_demux` optimization by setting a `sysctl`.

```
$ sudo sysctl -w net.ipv4.ip_early_demux=0
```

The default value is 1; `early_demux` is enabled.

This sysctl was added as some users saw a ~5% decrease in throughput with the `early_demux` optimization in some cases.

ip_local_deliver

Recall how we saw the following pattern in the IP protocol layer:

1. Calls to `ip_rcv` do some initial bookkeeping.
2. Packet is handed off to netfilter for processing, with a pointer to a callback to be executed when processing finishes.
3. `ip_rcv_finish` is the callback which finished processing and continued working toward pushing the packet up the networking stack.

`ip_local_deliver` has the same pattern. From net/ipv4/ip_input.c:

```
/*
 *      Deliver IP Packets to the higher protocol layers.
 */
int ip_local_deliver(struct sk_buff *skb)
{
        /*
         *      Reassemble IP fragments.
         */

        if (ip_is_fragment(ip_hdr(skb))) {
                if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
                        return 0;
        }

        return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
                       ip_local_deliver_finish);
}
```

Once netfilter has had a chance to take a look at the data, `ip_local_deliver_finish` will be called, assuming the data is not dropped first by netfilter.

ip_local_deliver_finish

`ip_local_deliver_finish` obtains the protocol from the packet, looks up a `net_protocol` structure registered for that protocol, and calls the function pointed to by `handler` in the `net_protocol` structure.

This hands the packet up to the higher level protocol layer.

Monitoring: IP protocol layer statistics

Monitor detailed IP protocol statistics by reading `/proc/net/snmp`.

```
$ cat /proc/net/snmp
Ip: Forwarding DefaultTTL InReceives InHdrErrors InAddrErrors ForwDatagrams
InUnknownProtos InDiscards InDelivers OutRequests OutDiscards OutNoRoutes Re
asmTimeout ReasmReqds ReasmOKs ReasmFails FragOKs FragFails FragCreates
Ip: 1 64 25922988125 0 0 15771700 0 0 25898327616 22789396404 12987882 51 1
10129840 2196520 1 0 0 0
...
```

This file contains statistics for several protocol layers. The IP protocol layer appears first. The first line contains space separate names for each of the corresponding values in the next line.

In the IP protocol layer, you will find statistics counters being bumped. Those counters are referenced by a C enum. All of the valid enum values and the field names they correspond to in `/proc/net/snmp` can be found in include/uapi/linux/snmp.h:

```
enum
{
  IPSTATS_MIB_NUM = 0,
/* frequently written fields in fast path, kept in same cache line */
  IPSTATS_MIB_INPKTS,     /* InReceives */
  IPSTATS_MIB_INOCTETS,     /* InOctets */
  IPSTATS_MIB_INDELIVERS,     /* InDelivers */
  IPSTATS_MIB_OUTFORWDATAGRAMS,   /* OutForwDatagrams */
  IPSTATS_MIB_OUTPKTS,      /* OutRequests */
  IPSTATS_MIB_OUTOCTETS,      /* OutOctets */


  /* ... */
```

Monitor extended IP protocol statistics by reading `/proc/net/netstat`.

```
$ cat /proc/net/netstat | grep IpExt
IpExt: InNoRoutes InTruncatedPkts InMcastPkts OutMcastPkts InBcastPkts OutBc
astPkts InOctets OutOctets InMcastOctets OutMcastOctets InBcastOctets OutBca
stOctets InCsumErrors InNoECTPkts InECT0Pktsu InCEPkts
```

```
IpExt: 0 0 0 0 277959 0 14568040307695 32991309088496 0 0 58649349 0 0 0 0 0
```

The format is similar to `/proc/net/snmp`, except the lines are prefixed with `IpExt`.

Some interesting statistics:

- `InReceives`: The total number of IP packets that reached `ip_rcv` before any data integrity checks.
- `InHdrErrors`: Total number of IP packets with corrupted headers. The header was too short, too long, non-existent, had the wrong IP protocol version number, etc.
- `InAddrErrors`: Total number of IP packets where the host was unreachable.
- `ForwDatagrams`: Total number of IP packets that have been forwarded.
- `InUnknownProtos`: Total number of IP packets with unknown or unsupported protocol specified in the header.
- `InDiscards`: Total number of IP packets discarded due to memory allocation failure or checksum failure when packets are trimmed.
- `InDelivers`: Total number of IP packets successfully delivered to higher protocol layers. Keep in mind that those protocol layers may drop data even if the IP layer does not.
- `InCsumErrors`: Total number of IP Packets with checksum errors.

Note that each of these is incremented in really specific locations in the IP layer. Code gets moved around from time to time and double counting errors or other accounting bugs can sneak in. If these statistics are important to you, you are strongly encouraged to read the IP protocol layer source code for the metrics that are important to you so you understand when they are (and are not) being incremented.

Higher level protocol registration

This blog post will examine UDP, but the TCP protocol handler is registered the same way and at the same time as the UDP protocol handler.

In `net/ipv4/af_inet.c`, the structure definitions which contain the handler functions for connecting the UDP, TCP , and ICMP protocols to the IP protocol layer can be found. From net/ipv4/af_inet.c:

```
static const struct net_protocol tcp_protocol = {
        .early_demux    =       tcp_v4_early_demux,
        .handler        =       tcp_v4_rcv,
        .err_handler    =       tcp_v4_err,
        .no_policy      =       1,
        .netns_ok       =       1,
};

static const struct net_protocol udp_protocol = {
        .early_demux =  udp_v4_early_demux,
        .handler =      udp_rcv,
        .err_handler =  udp_err,
        .no_policy =    1,
        .netns_ok =     1,
};
```

```
static const struct net_protocol icmp_protocol = {
        .handler =      icmp_rcv,
        .err_handler =  icmp_err,
        .no_policy =    1,
        .netns_ok =     1,
};
```

These structures are registered in the initialization code of the inet address family. From net/ipv4/af_inet.c:

```
/*
 *      Add all the base protocols.
 */

 if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
        pr_crit("%s: Cannot add ICMP protocol\n", __func__);
 if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
        pr_crit("%s: Cannot add UDP protocol\n", __func__);
 if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
        pr_crit("%s: Cannot add TCP protocol\n", __func__);
```

We're going to be looking at the UDP protocol layer. As seen above, the `handler` function for UDP is called `udp_rcv`.

This is the entry point into the UDP layer where the IP layer hands data. Let's continue our journey there.

UDP protocol layer

The code for the UDP protocol layer can be found in: net/ipv4/udp.c.

udp_rcv

The code for the `udp_rcv` function is just a single line which calls directly into `__udp4_lib_rcv` to handle receiving the datagram.

__udp4_lib_rcv

The `__udp4_lib_rcv` function will check to ensure the packet is valid and obtain the UDP header, UDP datagram length, source address, and destination address. Next, are some additional integrity checks and checksum verification.

Recall that earlier in the IP protocol layer, we saw that an optimization is performed to attach a `dst_entry` to the packet before it is handed off to the upper layer protocol (UDP in our case).

If a socket and corresponding `dst_entry` were found, `__udp4_lib_rcv` will queue the packet to the socket:

```
 sk = skb_steal_sock(skb);
if (sk) {
  struct dst_entry *dst = skb_dst(skb);
```

```
  int ret;

  if (unlikely(sk->sk_rx_dst != dst))
    udp_sk_rx_dst_set(sk, dst);

  ret = udp_queue_rcv_skb(sk, skb);
  sock_put(sk);
  /* a return value > 0 means to resubmit the input, but
   * it wants the return to be -protocol, or 0
   */
  if (ret > 0)
    return -ret;
  return 0;
} else {
```

If there is no socket attached from the early_demux operation, a receiving socket will now be looked up by calling `__udp4_lib_lookup_skb`.

In both cases described above, the datagram will be queued to the socket:

```
 ret = udp_queue_rcv_skb(sk, skb);
sock_put(sk);
```

If no socket was found, the datagram will be dropped:

```
 /* No socket. Drop packet silently, if checksum is wrong */
if (udp_lib_checksum_complete(skb))
        goto csum_error;

UDP_INC_STATS_BH(net, UDP_MIB_NOPORTS, proto == IPPROTO_UDPLITE);
icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);

/*
 * Hmm.  We got an UDP packet to a port to which we
 * don't wanna listen.  Ignore it.
 */
kfree_skb(skb);
return 0;
```

udp_queue_rcv_skb

The initial parts of this function are as follows:

1. Determine if the socket associated with the datagram is an encapsulation socket. If so, pass the packet up to

that layer's handler function before proceeding.
2. Determine if the datagram is a UDP-Lite datagram and do some integrity checks.
3. Verify the UDP checksum of the datagram and drop it if the checksum fails.

Finally, we arrive at the receive queue logic which begins by checking if the receive queue for the socket is full. From `net/ipv4/udp.c` :

```
if (sk_rcvqueues_full(sk, skb, sk->sk_rcvbuf))
  goto drop;
```

sk_rcvqueues_full

The `sk_rcvqueues_full` function checks the socket's backlog length and the socket's `sk_rmem_alloc` to determine if the sum is greater than the `sk_rcvbuf` for the socket (`sk->sk_rcvbuf` in the above code snippet):

```
/*
 * Take into account size of receive queue and backlog queue
 * Do not take into account this skb truesize,
 * to allow even a single big packet to come.
 */
static inline bool sk_rcvqueues_full(const struct sock *sk, const struct sk_buff *s
kb,
                                     unsigned int limit)
{
        unsigned int qsize = sk->sk_backlog.len + atomic_read(&sk->sk_rmem_alloc);

        return qsize > limit;
}
```

Tuning these values is a bit tricky as there are many things that can be adjusted.

Tuning: Socket receive queue memory

The `sk->sk_rcvbuf` (called limit in `sk_rcvqueues_full` above) value can be increased to whatever the sysctl `net.core.rmem_max` is set to.

Increase the maximum receive buffer size by setting a `sysctl`.

```
$ sudo sysctl -w net.core.rmem_max=8388608
```

`sk->sk_rcvbuf` starts at the `net.core.rmem_default` value, which can also be adjusted by setting a sysctl, like so:

Adjust the default initial receive buffer size by setting a `sysctl`.

```
$ sudo sysctl -w net.core.rmem_default=8388608
```

You can also set the `sk->sk_rcvbuf` size by calling `setsockopt` from your application and passing `SO_RCVBUF`. The maximum you can set with `setsockopt` is `net.core.rmem_max`.

However, you can override the `net.core.rmem_max` limit by calling `setsockopt` and passing `SO_RCVBUFFORCE`, but the user running the application will need the `CAP_NET_ADMIN` capability.

The `sk->sk_rmem_alloc` value is incremented by calls to `skb_set_owner_r` which set the owner socket of a datagram. We'll see this called later in the UDP layer.

The `sk->sk_backlog.len` is incremented by calls to `sk_add_backlog`, which we'll see next.

udp_queue_rcv_skb

Once it's been verified that the queue is not full, progress toward queuing the datagram can continue. From net/ipv4/udp.c:

```
 bh_lock_sock(sk);
if (!sock_owned_by_user(sk))
  rc = __udp_queue_rcv_skb(sk, skb);
else if (sk_add_backlog(sk, skb, sk->sk_rcvbuf)) {
  bh_unlock_sock(sk);
  goto drop;
}
bh_unlock_sock(sk);


return rc;
```

The first step is determine if the socket currently has any system calls against it from a userland program. If it does not, the datagram can be added to the receive queue with a call to `__udp_queue_rcv_skb`. If it does, the datagram is queued to the backlog with a call to `sk_add_backlog`.

The datagrams on the backlog are added to the receive queue when socket system calls release the socket with a call to `release_sock` in the kernel.

__udp_queue_rcv_skb

The `__udp_queue_rcv_skb` function adds datagrams to the receive queue by calling `sock_queue_rcv_skb` and bumps statistics counters if the datagram could not be added to the receive queue for the socket.

From net/ipv4/udp.c:

```
 rc = sock_queue_rcv_skb(sk, skb);
if (rc < 0) {
  int is_udplite = IS_UDPLITE(sk);


  /* Note that an ENOMEM error is charged twice */
  if (rc == -ENOMEM)
    UDP_INC_STATS_BH(sock_net(sk), UDP_MIB_RCVBUFERRORS,is_udplite);


  UDP_INC_STATS_BH(sock_net(sk), UDP_MIB_INERRORS, is_udplite);
```

```
  kfree_skb(skb);
  trace_udp_fail_queue_rcv_skb(rc, sk);
  return -1;
}
```

Monitoring: UDP protocol layer statistics

Two very useful files for getting UDP protocol statistics are:

- `/proc/net/snmp`
- `/proc/net/udp`

/proc/net/snmp

Monitor detailed UDP protocol statistics by reading `/proc/net/snmp`.

```
$ cat /proc/net/snmp | grep Udp\:
Udp: InDatagrams NoPorts InErrors OutDatagrams RcvbufErrors SndbufErrors
Udp: 16314 0 0 17161 0 0
```

Much like the detailed statistics found in this file for the IP protocol, you will need to read the protocol layer source to determine exactly when and where these values are incremented.

- `InDatagrams`: Incremented when `recvmsg` was used by a userland program to read datagram. Also incremented when a UDP packet is encapsulated and sent back for processing.
- `NoPorts`: Incrememnted when UDP packets arrive destined for a port where no program is listening.
- `InErrors`: Incremented in several cases: no memory in the receive queue, when a bad checksum is seen, and if `sk_add_backlog` fails to add the datagram.
- `OutDatagrams`: Incremented when a UDP packet is handed down without error to the IP protocol layer to be sent.
- `RcvbufErrors`: Incremented when `sock_queue_rcv_skb` reports that no memory is available; this happens if `sk->sk_rmem_alloc` is greater than or equal to `sk->sk_rcvbuf`.
- `SndbufErrors`: Incremented if the IP protocol layer reported an error when trying to send the packet, if there's no kernel memory, or no send buffer space.
- `InCsumErrors`: Incremented when a UDP checksum failure is detected. Note that in all cases I could find, `InCsumErrors` is incrememnted at the same time as `InErrors`. Thus, `InErrors` - `InCsumErros` should yield the count of memory related errors.

/proc/net/udp

Monitor UDP socket statistics by reading `/proc/net/udp`

```
$ cat /proc/net/udp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt
  uid  timeout inode ref pointer drops
  515: 00000000:B346 00000000:0000 07 00000000:00000000 00:00000000 00000000
   104        0 7518 2 0000000000000000 0
  558: 00000000:0371 00000000:0000 07 00000000:00000000 00:00000000 00000000
     0        0 7408 2 0000000000000000 0
  588: 0100007F:038F 00000000:0000 07 00000000:00000000 00:00000000 00000000
     0        0 7511 2 0000000000000000 0
  769: 00000000:0044 00000000:0000 07 00000000:00000000 00:00000000 00000000
```

```
    0         0 7673 2 0000000000000000 0
812: 00000000:006F 00000000:0000 07 00000000:00000000 00:00000000 00000000
    0         0 7407 2 0000000000000000 0
```

The first line describes each of the fields in the lines following:

- `sl`: Kernel hash slot for the socket
- `local_address`: Hexadecimal local address of the socket and port number, separated by `:`.
- `rem_address`: Hexadecimal remote address of the socket and port number, separated by `:`.
- `st`: The state of the socket. Oddly enough, the UDP protocol layer seems to use some TCP socket states. In the example above, `7` is `TCP_CLOSE`.
- `tx_queue`: The amount of memory allocated in the kernel for outgoing UDP datagrams.
- `rx_queue`: The amount of memory allocated in the kernel for incoming UDP datagrams.
- `tr`, `tm->when`, `retrnsmt`: These fields are unused by the UDP protocol layer.
- `uid`: The effective user id of the user who created this socket.
- `timeout`: Unused by the UDP protocol layer.
- `inode`: The inode number corresponding to this socket. You can use this to help you determine which user process has this socket open. Check `/proc/[pid]/fd`, which will contain symlinks to `socket[:inode]`.
- `ref`: The current reference count for the socket.
- `pointer`: The memory address in the kernel of the `struct sock`.
- `drops`: The number of datagram drops associated with this socket.

The code which outputs this can be found in `net/ipv4/udp.c`.

Queuing data to a socket

Network data is queued to a socket with a call to `sock_queue_rcv`. This function does a few things before adding the datagram to the queue:

1. The socket's allocated memory is checked to determine if it has exceeded the receive buffer size. If so, the drop count for the socket is incremented.
2. Next `sk_filter` is used to process any Berkeley Packet Filter filters that have been applied to the socket.
3. `sk_rmem_schedule` is run to ensure sufficient receive buffer space exists to accept this datagram.
4. Next the size of the datagram is charged to the socket with a call to `skb_set_owner_r`. This increments `sk->sk_rmem_alloc`.
5. The data is added to the queue with a call to `__skb_queue_tail`.
6. Finally, any processes waiting on data to arrive in the socket are notified with a call to the `sk_data_ready` notification handler function.

And that is how data arrives at a system and traverses the network stack until it reaches a socket and is ready to be read by a user program.

Extras

There are a few extra things worth mentioning that are worth mentioning which didn't seem quite right anywhere else.

Timestamping

As mentioned in the above blog post, the networking stack can collect timestamps of incoming data. There are sysctl

values controlling when/how to collect timestamps when used in conjunction with RPS; see the above post for more information on RPS, timestamping, and where, exactly, in the network stack receive timestamping happens. Some NICs even support timestamping in hardware, too.

This is a useful feature if you'd like to try to determine how much latency the kernel network stack is adding to receiving packets.

The kernel documentation about timestamping is excellent and there is even an included sample program and Makefile you can check out!.

Determine which timestamp modes your driver and device support with `ethtool -T`.

```
$ sudo ethtool -T eth0
Time stamping parameters for eth0:
Capabilities:
  software-transmit     (SOF_TIMESTAMPING_TX_SOFTWARE)
  software-receive      (SOF_TIMESTAMPING_RX_SOFTWARE)
  software-system-clock (SOF_TIMESTAMPING_SOFTWARE)
PTP Hardware Clock: none
Hardware Transmit Timestamp Modes: none
Hardware Receive Filter Modes: none
```

This NIC, unfortunately, does not support hardware receive timestamping, but software timestamping can still be used on this system to help me determine how much latency the kernel is adding to my packet receive path.

Busy polling for low latency sockets

It is possible to use a socket option called `SO_BUSY_POLL` which will cause the kernel to busy poll for new data when a blocking receive is done and there is no data.

IMPORTANT NOTE: For this option to work, your device driver must support it. Linux kernel 3.13.0's `igb` driver does not support this option. The `ixgbe` driver, however, does. If your driver has a function set to the `ndo_busy_poll` field of its `struct net_device_ops` structure (mentioned in the above blog post), it supports `SO_BUSY_POLL`.

A great paper explaining how this works and how to use it is available from Intel.

When using this socket option for a single socket, you should pass a time value in microseconds as the amount of time to busy poll in the device driver's receive queue for new data. When you issue a blocking read to this socket after setting this value, the kernel will busy poll for new data.

You can also set the sysctl value `net.core.busy_poll` to a time value in microseconds of how long calls with `poll` or `select` should busy poll waiting for new data to arrive, as well.

This option can reduce latency, but will increase CPU usage and power consumption.

Netpoll: support for networking in critical contexts

The Linux kernel provides a way for device drivers to be used to send and receive data on a NIC when the kernel has crashed. The API for this is called Netpoll and it is used by a few things, but most notably: kgdb, netconsole.

Most drivers support Netpoll; your driver needs to implement the `ndo_poll_controller` function and attach it to the `struct net_device_ops` that is registered during probe (as seen above).

When the networking device subsystem performs operations on incoming or outgoing data, the netpoll system is

checked first to determine if the packet is destined for netpoll.

For example, we can see the following code in `__netif_receive_skb_core` from `net/dev/core.c` :

```
static int __netif_receive_skb_core(struct sk_buff *skb, bool pfmemalloc)
{

  /* ... */

  /* if we've gotten here through NAPI, check netpoll */
  if (netpoll_receive_skb(skb))
    goto out;

  /* ... */
}
```

The Netpoll checks happen early in most of the Linux network device subsystem code that deals with transmitting or receiving network data.

Consumers of the Netpoll API can register `struct netpoll` structures by calling `netpoll_setup`. The `struct netpoll` structure has function pointers for attaching receive hooks, and the API exports a function for sending data.

If you are interested in using the Netpoll API, you should take a look at the `netconsole` driver, the Netpoll API header file, 'include/linux/netpoll.h`, and this excellent talk.

SO_INCOMING_CPU

The `SO_INCOMING_CPU` flag was not added until Linux 3.19, but it is useful enough that it should be included in this blog post.

You can use `getsockopt` with the `SO_INCOMING_CPU` option to determine which CPU is processing network packets for a particular socket. Your application can then use this information to hand sockets off to threads running on the desired CPU to help increase data locality and CPU cache hits.

The mailing list message introducing `SO_INCOMING_CPU` provides a short example architecture where this option is useful.

DMA Engines

A DMA engine is a piece of hardware that allows the CPU to offload large copy operations. This frees the CPU to do other tasks while memory copies are done with hardware. Enabling the use of a DMA engine and running code that takes advantage of it, should yield reduced CPU usage.

The Linux kernel has a generic DMA engine interface that DMA engine driver authors can plug into. You can read more about the Linux DMA engine interface in the kernel source Documentation.

While there are a few DMA engines that the kernel supports, we're going to discuss one in particular that is quite common: the Intel IOAT DMA engine.

Intel's I/O Acceleration Technology (IOAT)

Many servers include the Intel I/O AT bundle, which is comprised of a series of performance changes.

One of those changes is the inclusion of a hardware DMA engine. You can check your `dmesg` output for `ioatdma` to determine if the module is being loaded and if it has found supported hardware.

The DMA offload engine is used in a few places, most notably in the TCP stack.

Support for the Intel IOAT DMA engine was included in Linux 2.6.18, but was disabled later in 3.13.11.10 due to some unfortunate data corruption bugs.

Users on kernels before 3.13.11.10 may be using the `ioatdma` module on their servers by default. Perhaps this will be fixed in future kernel releases.

Direct cache access (DCA)

Another interesting feature included with the Intel I/O AT bundle is Direct Cache Access (DCA).

This feature allows network devices (via their drivers) to place network data directly in the CPU cache. How this works, exactly, is driver specific. For the `igb` driver, you can check the code for the function `igb_update_dca`, as well as the code for `igb_update_rx_dca`. The `igb` driver uses DCA by writing a register value to the NIC.

To use DCA, you will need to ensure that DCA is enabled in your BIOS, the `dca` module is loaded, and that your network card and driver both support DCA.

Monitoring IOAT DMA engine

If you are using the `ioatdma` module despite the risk of data corruption mentioned above, you can monitor it by examining some entries in `sysfs`.

Monitor the total number of offloaded `memcpy` operations for a DMA channel.

```
$ cat /sys/class/dma/dma0chan0/memcpy_count
123205655
```

Similarly, to get the number of bytes offloaded by this DMA channel, you'd run a command like:

Monitor total number of bytes transferred for a DMA channel.

```
$ cat /sys/class/dma/dma0chan0/bytes_transferred
131791916307
```

Tuning IOAT DMA engine

The IOAT DMA engine is only used when packet size is above a certain threshold. That threshold is called the `copybreak`. This check is in place because for small copies, the overhead of setting up and using the DMA engine is not worth the accelerated transfer.

Adjust the DMA engine copybreak with a `sysctl`.

```
$ sudo sysctl -w net.ipv4.tcp_dma_copybreak=2048
```

The default value is 4096.

Conclusion

The Linux networking stack is complicated.

It is impossible to monitor or tune it (or any other complex piece of software) without understanding at a deep level exactly what's going on. Often, out in the wild of the Internet, you may stumble across a sample `sysctl.conf` that contains a set of sysctl values that should be copied and pasted on to your computer. This is probably not the best way to optimize your networking stack.

Monitoring the networking stack requires careful accounting of network data at every layer. Starting with the drivers and proceeding up. That way you can determine where exactly drops and errors are occurring and then adjust settings to determine how to reduce the errors you are seeing.

There is, unfortunately, no easy way out.

Help with Linux networking or other systems

Need some extra help navigating the network stack? Have questions about anything in this post or related things not covered? Take a look at our services page and let us know how we can help.

Related posts

If you enjoyed this post, you may enjoy some of our other low-level technical posts:

- The Definitive Guide to Linux System Calls
- How does `strace` work?
- How does `ltrace` work?
- APT Hash sum mismatch
- HOWTO: GPG sign and verify deb packages and APT repositories
- HOWTO: GPG sign and verify RPM packages and yum repositories