Enabling Linux* Network Support of Hardware Multiqueue Devices

Zhu Yi
Intel Corp.
yi.zhu@intel.com

Peter P. Waskiewicz, Jr.
Intel Corp.
peter.p.waskiewicz.jr@intel.com

Abstract

In the Linux kernel network subsystem, the Tx/Rx SoftIRQ and Qdisc are the connectors between the network stack and the net devices. A design limitation is that they assume there is only a single entry point for each Tx and Rx in the underlying hardware. Although they work well today, they won't in the future. Modern network devices (for example, E1000 [8] and IPW2200 [6]) equip two or more hardware Tx queues to enable transmission parallelization or MAC-level QoS. These hardware features cannot be supported easily with the current network subsystem.

This paper describes the design and implementation for the network multiqueue patches submitted to *netdev* [2] and *LKML* [1] mailing lists early this year, which involved the changes for the network scheduler, Qdisc, and generic network core APIs. It will also discuss breaking the `netdev->queue_lock` with fine-grained per-queue locks in the future. At the end of the paper, it takes the IPW2200 and E1000 drivers as an example to illustrate how the new network multiqueue features will be used by the network drivers.

1 A Brief Introduction for the Linux Network Subsystem

When a packet is passed from the user space into the kernel space through a socket, an *skb* (socket kernel buffer) is created to represent that packet in the kernel. The *skb* is passed through various layers of the network stack before it is handed to the device driver for transmission. Inside the Linux kernel, each network device is represented by a `struct net_device` structure. All the `struct net_device` instances are linked into a doubly linked list with a single pointer list head (called `hlist`); the list head is named `dev_base`. The `struct net_device` contains all information and function pointers for the device. Among them there is a `qdisc` item. Qdisc stands for queuing discipline. It defines how a packet is selected on the transmission path. A Qdisc normally contains one or more queues (`struct sk_buff_head`) and a set of operations (`struct Qdisc_ops`). The standard `.enqueue` and `.dequeue` operations are used to put and get packets from the queues by the core network layer. When a packet first arrives to the network stack, an attempt to enqueue occurs. The `.enqueue` routine can be a simple FIFO, or a complex traffic classification algorithm. It all depends on the type of Qdisc the system administrator has chosen and configured for the system. Once the enqueue is completed, the packet scheduler is invoked to pull an *skb* off a queue somewhere for transmission. This is the `.dequeue` operation. The *skb* is returned to the stack, and is then sent to the device driver for transmission on the wire. The network packets transmission can be started by either `dev_queue_xmit()` or the TX SoftIRQ (`net_tx_action()`) depending on whether the packet can be transmitted immediately or not. But both routines finally call `qdisc_run()` to dequeue an *skb* from the root Qdisc of the *netdev* and send it out by calling the `netdev->hard_start_xmit()` method.

When the hardware Tx queue of a network device is full (this can be caused by various reasons—e.g., carrier congestion, hardware errors, etc.), the driver should call `netif_stop_queue()` to indicate to the network scheduler that the device is currently unusable. So the `qdisk_restart()` function of the network scheduler won't try to transmit the packet (with the device's `.hard_start_xmit()` method) until the driver explicitly calls `netif_start_queue()` or `netif_wake_queue()` to indicate the network scheduler its hardware queue is available again. The `netdev->hard_start_xmit()` method is responsible for checking the hardware queue states. If the device hardware queue is full, it should call `netif_stop_queue()` and returns `NETDEV_TX_BUSY`. On the other side, when the network scheduler receives the

`NETDEV_TX_BUSY` as the return value for `netdev->hard_start_xmit()`, it will reschedule. Note that if a driver returns `NETDEV_TX_BUSY` without calling `netif_stop_queue()` in the `hard_start_xmit()` method when the hardware transmit queue is full, it will chew tons of CPU.

For a normal network device driver, the general rules to deal with the hardware Tx queue are:

- Driver detects queue is full and calls `netif_stop_queue()`;
- Network scheduler will not try to send more packets through the card any more;
- Even in some rare conditions (`dev->hard_start_xmit()` is still called), calls into the driver from top network layer always get back a `NETDEV_TX_BUSY`;
- EOT interupt happens and driver cleans up the TX hardware path to make more space so that the core network layer can send more packets (driver calls `netif_start_queue()`)
- Subsequent packets get queued to the hardware.

In this way, the network drivers use `netif_stop_queue()` and `netif_start_queue()` to provide feedback to the network scheduler so that neither packet starvation nor a CPU busy loop occurs.

2 What's the Problem if the Device has Multiple Hardware Queues?

The problem happens when the underlying hardware has multiple hardware queues. Multiple hardware queues provide QoS support from the hardware level. Wireless network adapters such as the Intel® PRO/Wireless 3945ABG, Intel® PRO/Wireless 2915ABG, and Intel® PRO/Wireless 2200BG Network Connections have already provided this feature in hardware. Other high speed ethernet adapters (i.e., e1000) also provide multiple hardware queues for better packet throughput by parallelizing the Tx and Rx paths. But the current Qdisc interface isn't multiple-hardware-queue aware. That is, the `.dequeue` method is not able to dequeue the correct *skb* according to the device hardware queue states. Take a device containing two hardware Tx queues for an example: if the high-priority queue is full while the low one is not, the Qdisc will still keep dequeueing the high priority *skb*. But it will always fail to transmit in the high-priority queue because the corresponding hardware queue is full. To make the situation even worse, `netif_stop_queue()` and friends are also ignorant of multiple hardware queues. There is no way to schedule Tx SoftIRQ based on hardware queues (vs. based on the global *netdev*). For example, if the low-priority hardware queue is full, should the driver call `netif_stop_queue()` or not? If the driver does, the high priority *skb*s will also be blocked (because the netdev Tx queue is stopped). If the driver doesn't, the high CPU usage problem we mentioned in Section 1 will happen when low priority *skb*s remain in the Qdisc.

3 How to Solve the Problem?

Since the root cause for this problem is that the network scheduler and Qdisc do not expect the network devices to have multiple hardware queues, the obvious fix is to add the support for multi-queue features to them.

4 The Design Considerations

The main goal of implementing support for multiple queues is to prevent one flow of traffic from interfering with another traffic flow. Therefore, if a hardware queue is full, the driver will need to stop the queue. With multiqueue, the driver should be able to stop an individual queue, and the network stack in the OS should know how to check individual queue states. If a queue is stopped, the network stack should be able to pull packets from another queue and send them to the driver for transmission.

One main consideration with this approach is how the stack will handle traffic from multiqueue devices and non-multiqueue devices in the same system. This must be transparent to devices, while maintaining little to no additional overhead.

5 The Implementation Details

The following details of implementation are under discussion and consideration in the Linux community at the writing of this paper. The concepts should remain the same, even if certain implementation details are changed to meet requests and suggestions from the community.

The first part of implementation is how to represent queues on the device. Today, there is a single queue state and lock in the `struct net_device`. Since we need to manage the queue's state, we will need a state for each queue. The queues need to be accessible from the `struct net_device` so they can be visible to both the stack and the driver. This is added to `include/linux/netdevice.h`:

Inside `struct net_device`:

```
struct net_device
{

 ...

 struct net_device_subqueue
   *egress_subqueue;

 unsigned long
   egress_subqueue_count;

 ...
}
```

Here, the *netdev* has the knowledge of the queues, and how many queues are supported by the device. For all non-multiqueue devices, there will be one queue allocated, with an `egress_subqueue_count` of 1. This is to help the stack run both non-multiqueue devices and multiqueue devices simultaneously. The details of this will be discussed later.

When a network driver is loaded, it needs to allocate a `struct net_device` and a structure representing itself. In the case of ethernet devices, the function `alloc_etherdev()` is called. A change to this API provides `alloc_etherdev_mq()`, which allows a driver to tell the kernel how many queues it wants to allocate for the device. `alloc_etherdev()` is now a macro that calls `alloc_etherdev_mq()` with a queue count of 1. This allows non-multiqueue drivers to transparently operate in the multiqueue stack without any changes to the driver.

Ultimately, `alloc_etherdev_mq()` calls the new `alloc_netdev_mq()`, which actually handles the `kzalloc()`. In here, `egress_subqueue` is assigned to the newly allocated memory, and `egress_subqueue_count` is assigned to the number of allocated queues.

On the deallocation side of things, `free_netdev()` now destroys the memory that was allocated for each queue.

The final part of the multiqueue solution comes with the driver being able to manipulate each queue's state. If one hardware queue runs out of descriptors for whatever reason, the driver will shut down that queue. This operation should not prevent other queues from transmitting traffic. To achieve this, the network stack should check the state for the global queue state, as well as the individual queue states, before deciding to transmit traffic on that queue.

Queue mapping in this implementation happens in the Qdisc, namely PRIO. The mapping is a combination of which PRIO band an *skb* is assigned to (based on TC filters and/or IP TOS to priority mapping), and which hardware queue is assigned to which band. Once the *skb* has been classified in `prio_classify()`, a lookup to the `band2queue` mapping is done, and assigned to a new field in the *skb*, namely `skb->queue_mapping`. The calls to `netif_subqueue_stopped()` will pass this queue mapping to know if the hardware queue to verify is running or not. In order to help performance and avoid unnecessary requeues, this check is done in the `prio_dequeue()` routine, prior to pulling an *skb* from the band. The check will be done again before the call to `hard_start_xmit()`, just as it is done today on the global queue. The *skb* is then passed to the device driver, which will need to look at the value of `skb->queue_mapping` to determine which Tx ring to place the *skb* on in the hardware. At this point, multiqueue flows have been established.

The network driver will need to manage the queues using the new `netif_{start|stop|wake}_subqueue()` APIs. This way full independance between queues can be established.

6 Using the Multiqueue Features

6.1 Intel$^{®}$ PRO/Wireless 2200BG Network Connection Driver

This adapter supports the IEEE 802.11e standard [3] for Quality of Service (QoS) on the Medium Access Control (MAC) layer. Figure 1 shows the hardware implementation. Before a MSDU (MAC Service Data Unit) is passed to the hardware, the network stack maps the

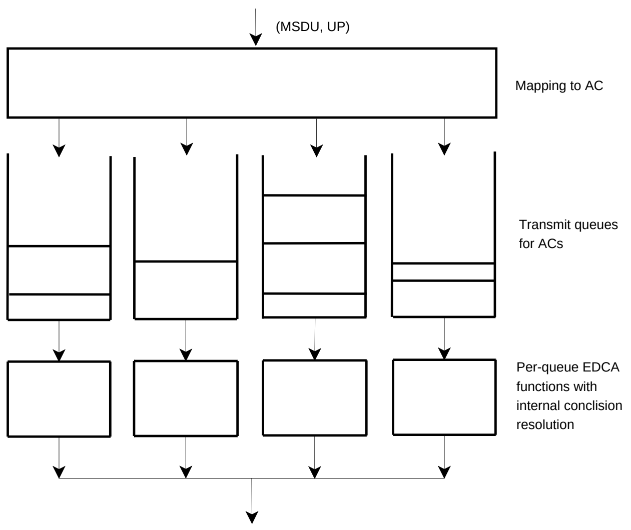
(MSDU, UP)
Mapping to AC
Transmit queues for ACs
Per-queue EDCA functions with internal conclision resolution

Figure 1: MAC level QoS implementation

frame type or User Priority (UP) to Access Category (AC). The NIC driver then pushes the frame to the corresponding one of the four transmit queues according to the AC. There are four independent EDCA (enhanced distributed channel access) functions in the hardware, one for each queue. The EDCA function resolves internal collisions and determines when a frame in the transmit queue is permitted to be transmitted via the wireless medium.

With the multiqueue devices supported by the network subsystem, such hardware-level packet scheduling is easy to enable. First, a specific PRIO Qdisc queue mapping for all the IEEE 802.11 wireless devices is created. It maps the frame type or UP to AC according to the mapping algorithm defined in [3]). With this specific queue mapping, the IEEE 802.11 frames with higher priority are always guaranteed to be scheduled before the lower priority ones by the network scheduler when both transmit queues are active at the time. In other cases, for example, the higher priority transmit queue is inactive while the lower priority transmit queue is active, and the lower priority frame is scheduled (dequeued). This is the intention because the hardware is not going to transmit any higher priority frames at this time. When the `dev->hard_start_xmit()` is invoked by the network scheduler, the `skb->queue_mapping` is already set to the corresponding transmit queue index for the *skb* (by the Qdisc `.dequeue` method). The driver then just needs to read this value and move the *skb* to the target transmit queue accordingly. In the normal

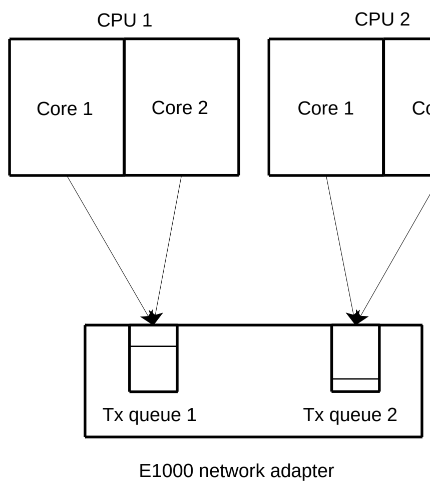

Figure 2: Mapping CPU cores to multiple Tx queues on SMP system

cases, the queue will still remain active since the network scheduler has just checked its state in the Qdisc `.dequeue` method. But in some rare cases, a race condition would still happen during this period to make the queue state inconsistent. This is the case where the Qdisc `.requeue` method is invoked by the network scheduler.

6.2 Intel® PRO/ 1000 Adapter Driver

This adapter with MAC types of 82571 and higher supports multiple Tx and Rx queues in hardware. A big advantage with these multiple hardware queues is to achieve packets transmission and reception parallelization. This is especially useful on SMP and multi-core systems with a lot of processes running network loads on different CPUs or cores. Figure 2 shows an e1000 network adapter with two hardware Tx queues on a multi-core SMP system.

With the multiqueue devices supported by the network subsystem, the e1000 driver can export all its Tx queues and bind them to different CPU cores. Figure 3 illustrates how this is done in the e1000 multiqueue patch [4]. The per-CPU variable `adapter->cpu_tx_ring` points to the mapped Tx queue for the current CPU. After the e1000 queue mapping has been setup, the access for the Tx queues should always be referenced by the `adapter->cpu_tx_ring` instead of manipulating the `adapter->tx_ring` array directly. With spreading CPUs on multiple hardware Tx

```
netdev = alloc_etherdev_mq(sizeof(struct e1000_adapter), 2);
netdev->features |= NETIF_IF_MULTI_QUEUE;

...

adapter->cpu_tx_ring = alloc_percpu(struct e1000_tx_ring *);

lock_cpu_hotplug();
i = 0;
for_each_online_cpu(cpu) {
  *per_cpu_ptr(adapter->cpu_tx_ring, cpu) =
               &adapter->tx_ring[i % adapter->num_tx_queues];
  i++;
}
unlock_cpu_hotplug();
```

Figure 3: E1000 driver binds CPUs to multiple hardware Tx queues

queues, transmission parallelization is achieved. Since the CPUs mapped to different Tx queues don't contend for the same lock for packet transmission, `LLTX` lock contention is also reduced. With breaking the `netdev->queue_lock` into per-queue locks in the future, this usage model will perform and scale even better. The same parallelization is also true for packet reception.

Comparing with the multiqueue usage model for the hardware QoS scheduler by the wireless devices, the e1000 usage model doesn't require a special frame type to queue mapping algorithm in the Qdisc. So any type of multiqueue-aware Qdiscs can be configured on top of e1000 hardware by the system administrator with command *tc*, which is part of the `iproute2` [7] package. For example, to add the PRIO Qdisc to your network device, assuming the device is called eth0, run the following command:

```
# tc qdisc add dev eth0 root \
          handle 1: prio
```

As of the writing of this paper, there are already patches in the linux-netdev mailing list [5] to enable the multiqueue features for the `pfifo_fast` and `PRIO` Qdiscs.

7 Future Work

In order to extend flexibility of multiqueue network device support, work on the Qdisc APIs can be done. This is needed to remove serialization of access to the Qdisc itself. Today, the Qdisc may only have one transmitter inside it, governed by the `__LINK_STATE_QDISC_RUNNING` bit set on the global queue state. This bit will need to be set per queue, not per device. Implications with Qdisc statistics will need to be resolved, such as the number of packets sent by the Qdisc, etc.

Per-queue locking may also need to be implemented in the future. This is dependent on performance of higher speed network adapters becoming throttled by the single device queue lock. If this is determined to be a source of contention, the stack will need to change to know how to independently lock and unlock each queue during a transmit.

8 Conclusion

The multiqueue devices support for network scheduler and Qdisc enables modern network interface controllers to provide advanced features like hardware-level packet scheduling and Tx/Rx parallelization. As processor packages ship with more and more cores nowadays, there is also a trend that the network adapter hardware may equip more and more Tx and Rx queues in the future. The multiqueue patch provides the fundamental features for the network core to enable these devices.

Legal

References

[1] Linux kernel mailing list. `linux-kernel@vger.kernel.org`.

[2] Linux network development mailing list. `linux-netdev@vger.kernel.org`.

[3] IEEE Computer Society. LAN/MAN Committee. *IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications Amendment 8: Medium Access Control (MAC) Quality of Service Enhancements*. 3 Park Avenue New York, NY 10016-5997, USA, November 2005.

[4] Peter P Waskiewicz Jr. E1000 example implementation of multiqueue network device api. `http://marc.info/?l=linux-netdev&m=117642254323203&w=2`.

[5] Peter P Waskiewicz Jr. Multiqueue network device support implementation. `http://marc.info/?l=linux-netdev&m=117642230823028&w=2`.

[6] James Ketrenos and the ipw2200 developers. *Intel® PRO/Wireless 2200BG Driver for Linux*. `http://ipw2200.sf.net`.

[7] Alexey Kuznetsov and Stephen Hemminger. *Iproute2: a collection of utilities for controlling TCP / IP networking and Traffic Control in Linux*. `http://linux-net.osdl.org/index.php/Iproute2`.

[8] John Ronciak, Auke Kok, and the e1000 developers. *The Intel® PRO/10/100/1000/10GbE Drivers*. `http://e1000.sf.net`.

Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007
Ottawa, Ontario
Canada