**DMA Access Process in Bare-metal Setting**

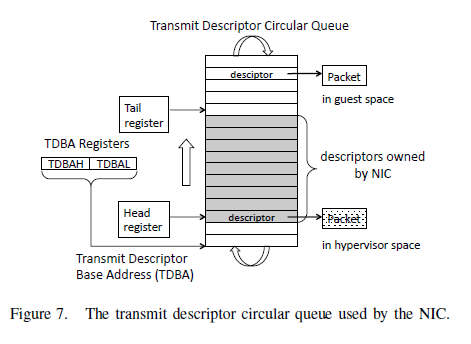
Modern server I/O devices, including disk and network controllers, utilize direct memory access (DMA) to move data between the host’s main memory and the device’s on-board buffers. The device uses DMA to access memory independently of the host CPU, so such accesses must be controlled and protected. To initiate a DMA operation, the device driver within the operating system creates DMA descriptors that refer to regions of memory. Each DMA descriptor typically includes an address, a length, and a few device-specific flags. In commodity x86 systems, devices lack support for virtual-to-physical address translation, so DMA descriptors always contain physical addresses for main memory. Once created, the device driver passes the descriptors to the device, which will later use the descriptors to transfer data to or from the indicated memory regions autonomously. When the requested I/O operations have been completed, the device raises an interrupt to notify the device driver

For example, to transmit a network packet, the network interface’s device driver might create two DMA descriptors. The first descriptor might point to the packet headers and the second descriptor might point to the packet payload. Once created, the device driver would then notify the network interface that there are new DMA descriptors available. The precise mechanism of that notification depends on the particular network interface, but typically involves a programmed I/O operation to the device telling it the location of the new descriptors. The network interface would then retrieve the descriptors from main memory using DMA—if they were not written to the device directly by programmed I/O. The network interface would then retrieve the two memory regions that compose the network packet and transmit them over the network. Finally, the network interface would interrupt the host to indicate that the packet has been transmitted. In practice, notifications from the device driver and interrupts from the network interface would likely be aggregated to cover multiple packets for efficiency.

**More details**

The packet transmission mechanism is illustrated in Figure 7. The NIC makes use of a ring buffer (essentially a circular queue) to store transmit descriptors which point to the packets to transmit. The ring buffer has its base address saved in the TDBAL and TDBAH registers, has its size saved in the TDLENL and TDLENH registers, and has a head register and a tail register pointing to the queue head and tail respectively. The NIC always dequeues the descriptor pointed by the head register, and then fetches the corresponding packet. After retrieval, it advances the head pointer. The tail pointer is maintained by the device driver. To send a new packet, the driver enqueues one or multiple descriptors. Then, the tail pointer is also advanced. The NIC only uses the descriptors between the head and the tail. It stops transmission when the two pointers collide.

The packet receiving mechanism is analogous to the transmission mechanism. It also has a ring buffer storing receive descriptors, and has its own base address registers, length registers, and the head and tail registers. Initially, the driver allocates a set of fixed length DMA buffers, and enqueues the corresponding descriptors into the ring queue. When receiving packets, the NIC stores them into those preallocated DMA buffers, updates the corresponding descriptors, and advances the head pointer accordingly. Finally, it throws out an interrupt to notify the driver to fetch the packets according to the descriptors.



**DMA Access Process for Network Packets Transmission involving VMM**

To carry out an I/O transaction using a single-use mapping strategy, the virtual machine monitor (VMM), untrusted guest operating system (GOS), and the device (DEV) carry out the following steps:

1. GOS: The guest OS requests an IOMMU mapping for the memory buffer involved in the I/O transaction.

2. VMM: The VMM validates that the requesting guest OS has appropriate read or write permission for each memory page in the buffer to be mapped.

3. VMM: The VMM marks the memory buffer as “in I/O use”, which prevents the buffer from being reallocated to another guest OS during an I/O transaction.

4. VMM: The VMM creates one or more IOMMU mappings for the buffer. As with virtual memory management units, one mapping is usually required for each memory page in the buffer.

5. GOS: The guest OS creates a DMA descriptor with the IOMMU-mapped address that was returned by the VMM.

6. DEV: The device carries out its I/O transaction as directed by the DMA descriptor and it notifies the driver upon completion.

7. GOS: The driver requests destruction of the corresponding IOMMU mapping(s).

8. VMM: The VMM validates that the mappings belong to the guest OS making the request.

9. VMM: The VMM destroys the IOMMU mappings.

10. VMM: The VMM clears the “in I/O use” marker associated with each memory page referred to by the recently-destroyed mapping(s).

注：需要看下PV下的 Linux NIC driver 和 xen 是不是遵循上述这个模式。

**Possible Attacks**

Three potential memory access violations can occur on every I/O transfer initiated using this DMA architecture:

1. The device driver could create a DMA descriptor with an incorrect address (a “bad-address” fault).

2. The operating system could repurpose the memory referenced by a DMA descriptor, or the device driver could later reuse a valid DMA descriptor without permission (an “invalid-use” fault).

3. The device itself could initiate a DMA transfer to a memory address not referenced by the DMA descriptor (a “bad-device” fault).

初步目标：我们先考虑上述所说的第二种攻击，修改descriptor指向的packet地址，使它指向某个页表页（without write permission），覆盖该页的内容，先造成domain里面某个进程的crash，再考虑如何将该页表页的内容修改成合理的payload。

当前应先看看PV模式下的 NIC driver 相关的函数，根据上述的机制理出一个思路。