DMA and Descriptors

ECE 373

Class notes for today

- DMA – what is it, why do we care?
- How DMA works
- What existed before DMA?
- How DMA is setup in Linux
- Descriptors, huh?
- Bonus: Packing command buffers

K*alloc and friends

- Many ways to alloc heap memory
  - kmalloc(), kmalloc_array(), kmalloc_node()
  - kzalloc(), kzalloc_node(),
  - more....
- void *kmalloc(size_t size, gfp_t flags)
  - Flags: GFP_USER, GFP_KERNEL, GFP_ATOMIC, GFP_NOWAIT, GFP_DMA, ...
- void kfree(void *)

- See kernel source

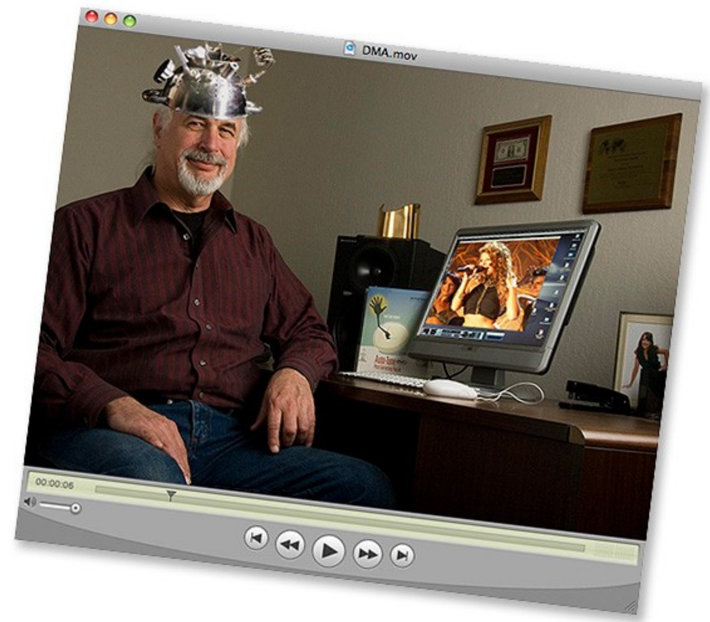
  http://lxr.free-electrons.com/source/include/linux/slab.h#L359

Direct Memory Access, the story

- DMA is a feature found in many pieces of hardware
- Allows direct access to system memory without the CPU
- Many devices live on slower busses
- DMA allows slower devices to transfer data at their leisure

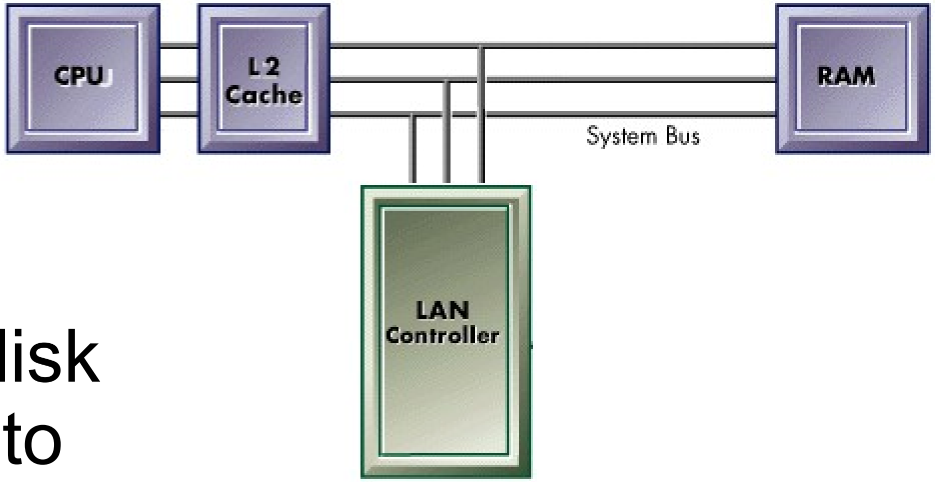
Why do we care about DMA?

- If devices can't move their own data, who does?
- CPU can only do one thing at a time
- If CPU is pre-occupied with copying data, system becomes sloooow



Back in the day...

- DMA didn't always exist
- Device RAM mapped into system memory
- CPU would do all the work moving data around
- Disk controllers worst, since disk spindles very slow compared to main memory
- If the CPU waits for data, it will stall
- Stall = Bad



Evolution of DMA

- DMA-capable devices showing up on the ISA bus
- Used a central DMA controller in between peripheral devices and memory
- DMA controller acted like a proxy
- Devices still using ISA DMA today are floppy disks and old parallel ports

Controller Driven DMA

1 – Data received from network

2 – Device sends IRQ

3 – IRQ handler starts DMA

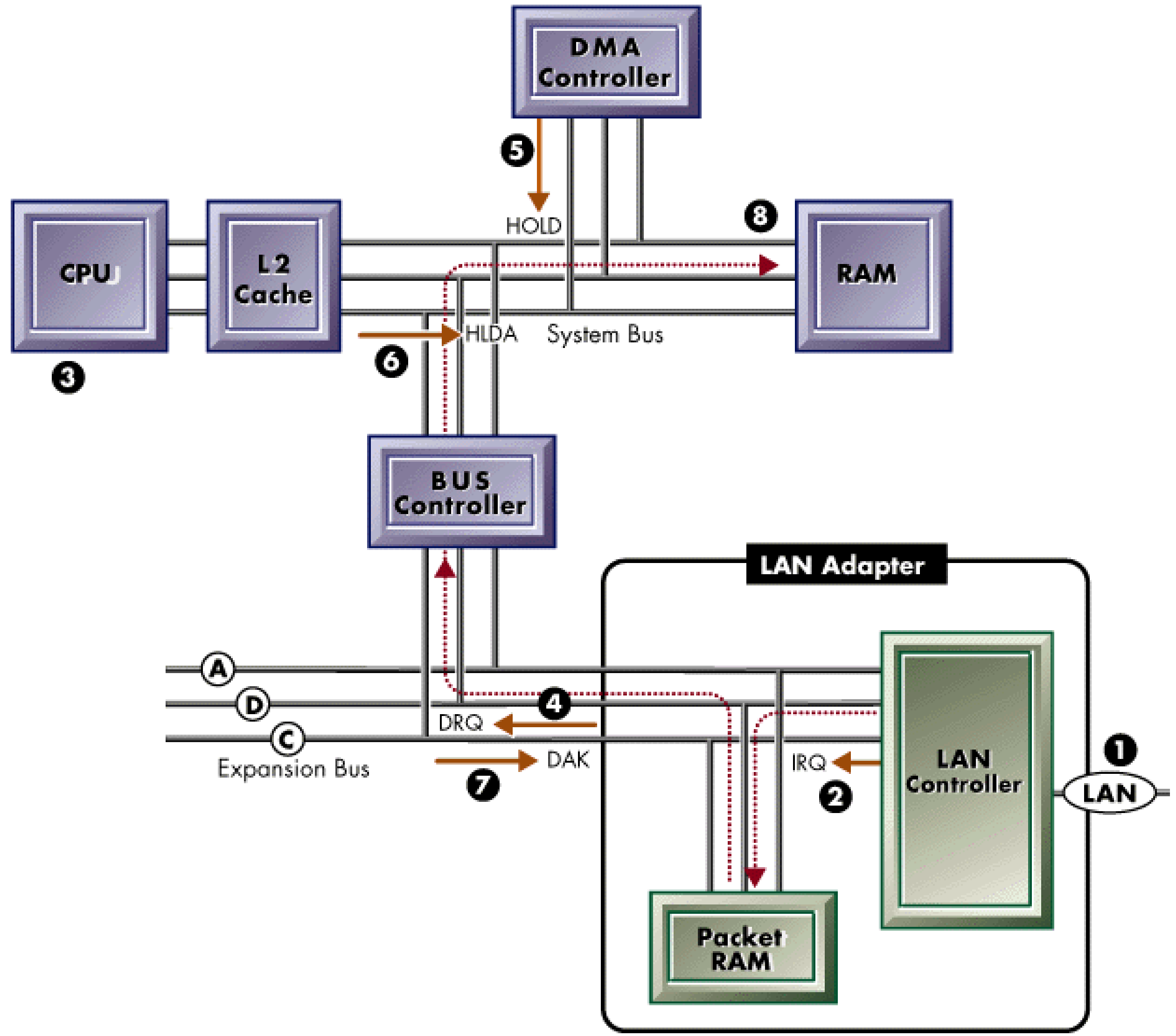
4 – Device requests DMA access

5 – DMA controller requests a bus hold

6 – CPU ack's the hold

7 – Bus controller ack's the DMA

8 – Device copies data to RAM



Further evolution of DMA

- DMA in PCI and PCI Express much different than ISA
- No centralized DMA controller
- DMA is performed by peripheral device
- However, not all PCI devices allowed to talk at once

There can be only one...

- PCI devices utilize protocol called bus mastering
- Each device requests ownership of PCI
- PCI controller arbitrates over all bus masters
- Each bus master issues commands when he owns the bus

Device Driven DMA

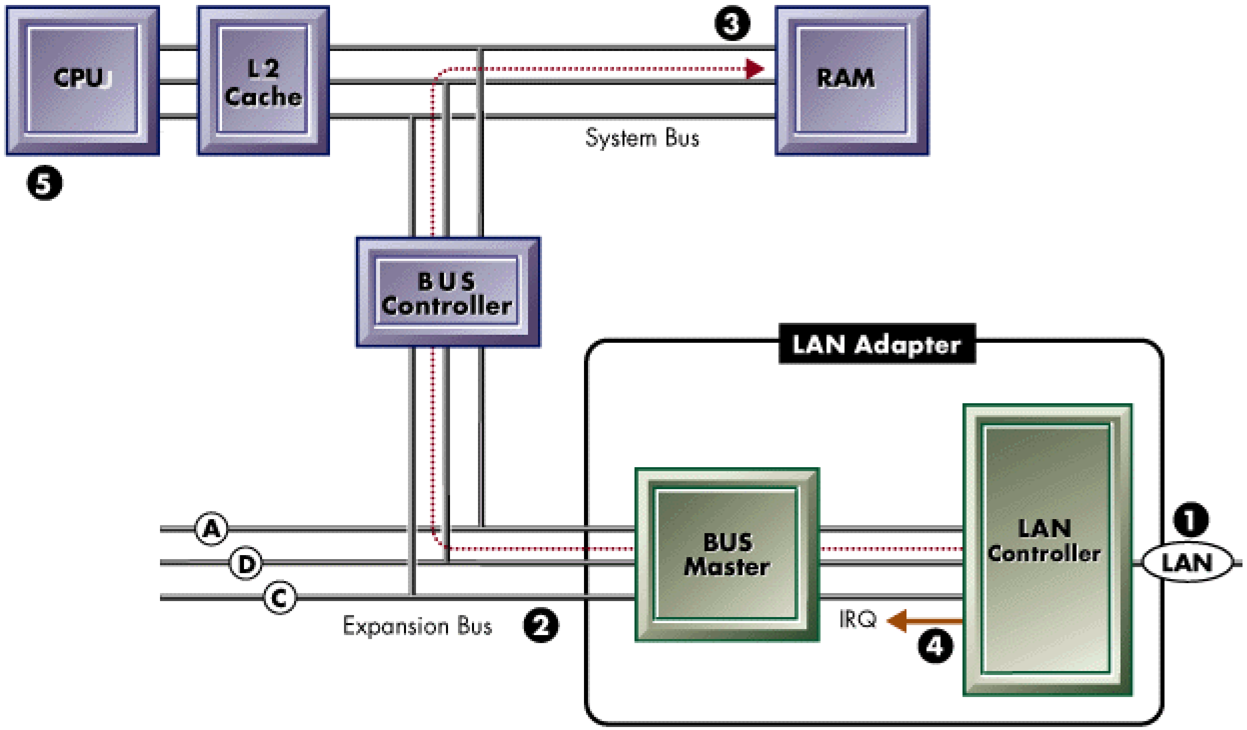
1 – Data received from network

2 – Bus master negotiates for bus access

3 – Device copies data to RAM

4 – Device sends IRQ

5 – IRQ handler processes data



DMA, how it works

- DMA-enabled device told which direction to DMA (to or from device, or bidirectional)
- Physical memory is mapped for DMA
- Device initiates the DMA (either to or from device)
- Memory unmapped for DMA after operation completes

DMA in Linux

- Function APIs exist for each driver type to control DMA
- DMA consists of mapping memory for DMA, unmapping when finished
- Mapping memory means pinning it down, not allowing it to be swapped out by memory manager
- DMA deals with physical address (or bus address)

DMA in PCI

- API for allocating and mapping memory in PCI:
  - dma_alloc_coherent(dev, size, dma_handle, flags)
- Coherent means can safely be used for both IN and OUT
- DMA handle returns (by reference) physical address of mapped memory
- Return type of function is kernel logical address of mapped memory
- Flags dictate mem allocation details: GFP_ATOMIC, GFP_KERNEL, __GFP_DMA

Contrived Example

```
/*
 * contrived DMA example
 */
static void ece_move_my_data(struct dev mydev, int len,
                              unsigned char *buf, void *hw_addr)
{
    dma_addr_t daddr;
    unsigned int reg;
    unsigned char *pmem;

    /* get and prepare the DMA memory */
    pmem = dma_alloc_coherent(mydev, len, &daddr, GFP_KERNEL);
    memcpy(pmem, buf, len);

    /* tell the device where the DMA space is */
    reg = (daddr >> 32) & 0xffffffff;
    writel(reg, hw_addr + ECE_DMA_HI_REGISTER);
    reg = daddr & 0xffffffff;
    writel(reg, hw_addr + ECE_DMA_LO_REGISTER);

    /* tell the device to copy the data */
    reg = 1;
    writel(reg, hw_addr + ECE_READ_DOORBELL);

    /* wait for DMA to complete */
    /* an interrupt handler would be better */
    usleep(10);
    dma_free_coherent(mydev, len, daddr);
}
```

Mapping existing memory

- kmalloc() returns kernel logical address
- Kernel logical addresses can be pinned
- dma_map_single(dev, ptr, size, flags) maps existing kernel logical address to pinned physical address
- Handy for mapping only when needed, keeps pinned memory space small
- Flags dictates copy direction, either DMA_TO_DEVICE, DMA_FROM_DEVICE, or DMA_BIDIRECTIONAL

Another Contrived Example

```
/*
 * contrived DMA example
 */
static void ece_move_my_data(struct dev mydev, int len,
                              unsigned char *buf, void *hw_addr)
{
    dma_addr_t daddr;
    unsigned int reg;
    unsigned char *pmem;

    /* pin the existing buffer */
    daddr = dma_map_single(mydev, buf, len, DMA_TO_DEVICE);

    /* tell the device where the DMA space is */
    reg = (daddr >> 32) & 0xffffffff;
    writel(reg, hw_addr + ECE_DMA_HI_REGISTER);
    reg = daddr & 0xffffffff;
    writel(reg, hw_addr + ECE_DMA_LO_REGISTER);

    /* tell the device to copy the data */
    reg = 1;
    writel(reg, hw_addr + ECE_READ_DOORBELL);

    /* wait for DMA to complete */
    /* an interrupt handler would be better */
    usleep(10);
    dma_unmap_single(mydev, daddr, len, DMA_TO_DEVICE);
}
```

DMA engines for non-devices

- Other areas of an OS require memory copies
- TCP stack needs to copy kernel memory to user memory (think copy_to_user())
- Intel I/O AT was DMA engine in chipset
- Changes made to OS's to offload these copies to I/O AT silicon
- Offloaded those particular memcpy operations from main CPU

Managing lots of DMA

- Most things that do DMA need to do lots of it
  - Disk controllers writing to and reading from disks
  - Network controllers receiving and transmitting
- We use lists of DMA requests
- Each request describes the activity
  - What data to move
  - How much data
  - In or Out
- These are called "descriptors"

Descriptors

- What is a descriptor?
  - Hardware field describing what work to do
  - Hardware field describing what work was done
- Carries bits and fields as instructions
- Carries pointers to buffers needing to be DMA'd into or out of hardware
- May return result information after the action

Example Descriptor layout

Table 7-24. Transmit Descriptor (TDESC) Fetch Layout - Legacy Mode

	63 48	47 40	39 36	35 32	31 24	23 16	15 0
0	Buffer Address [63:0]						
8	VLAN	CSS	Reserved	STA	CMD	CSO	Length

Table 7-25. Transmit Descriptor (TDESC) Write-Back Layout - Legacy Mode

	63 48	47 40	39 36	35 32	31 24	23 16	15 0
0	Reserved				Reserved		
8	VLAN	CSS	Reserved	STA	CMD	CSO	Length

Note: For frames that span multiple descriptors, the *VLAN*, *CSS*, *CSO*, *CMD.VLE*, *CMD.IC*, and *CMD.IFCS* are valid only in the first descriptors and are ignored in the subsequent ones.

Descriptors in code, simple

```
/* Transmit Descriptor */
struct e1000_tx_desc {
        __le64 buffer_addr;      /* Address of the descriptor's data buffer */
        __le64 flags_fields;
};
```

… or …

```
struct e1000_tx_desc {
        __le64 buffer_addr;      /* Address of the descriptor's data buffer */
        __le32 lower_fields;
        __le32 upper_flags;
};
```

… or useful

```
/* Transmit Descriptor */
struct e1000_tx_desc {
	__le64 buffer_addr;			/* Address of the descriptor's data buffer */
	union {
		__le32 data;
		struct {
			__le16 length;		/* Data buffer length */
			u8 cso;			/* Checksum offset */
			u8 cmd;			/* Descriptor control */
		} flags;
	} lower;
	union {
		__le32 data;
		struct {
			u8 status;		/* Descriptor status */
			u8 css;			/* Checksum start */
			__le16 special;
		} fields;
	} upper;
};
```

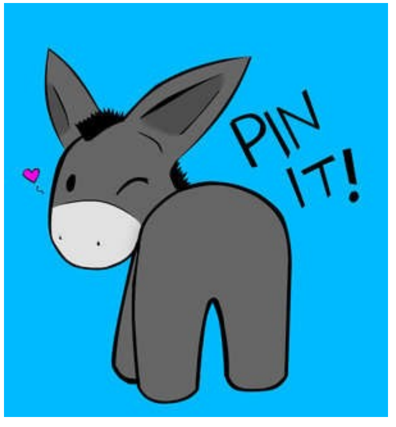
Filling it out

```
buffer_info->buf = databuf;
buffer_info->length = len;
buffer_info->dma = dma_map_single(mydev, databuf,
                                   len, DMA_TO_DEVICE);
```

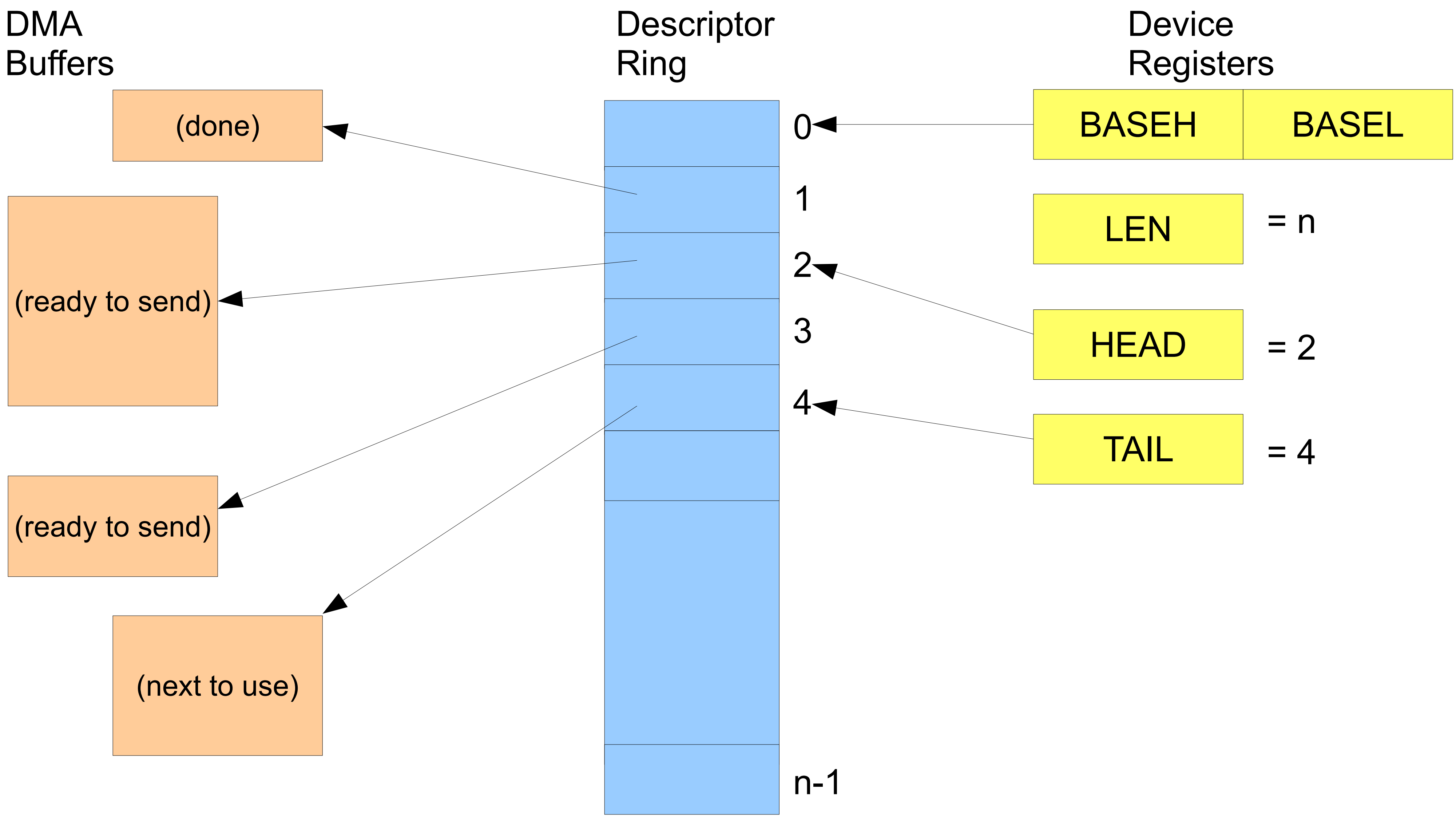
```
tx_desc = E1000_TX_DESC(*tx_ring, i);
tx_desc->buffer_addr = cpu_to_le64(buffer_info->dma);
tx_desc->lower.flags.length = cpu_to_le16(buffer_info->length);
tx_desc->lower.flags.cso = 0;
tx_desc->lower.flags.cmd = E1000_TXCMD_EOP;
tx_desc->upper.data = 0;
tx_desc->upper.fields.special = cpu_to_le16(vlan_id);
```

Rings!!

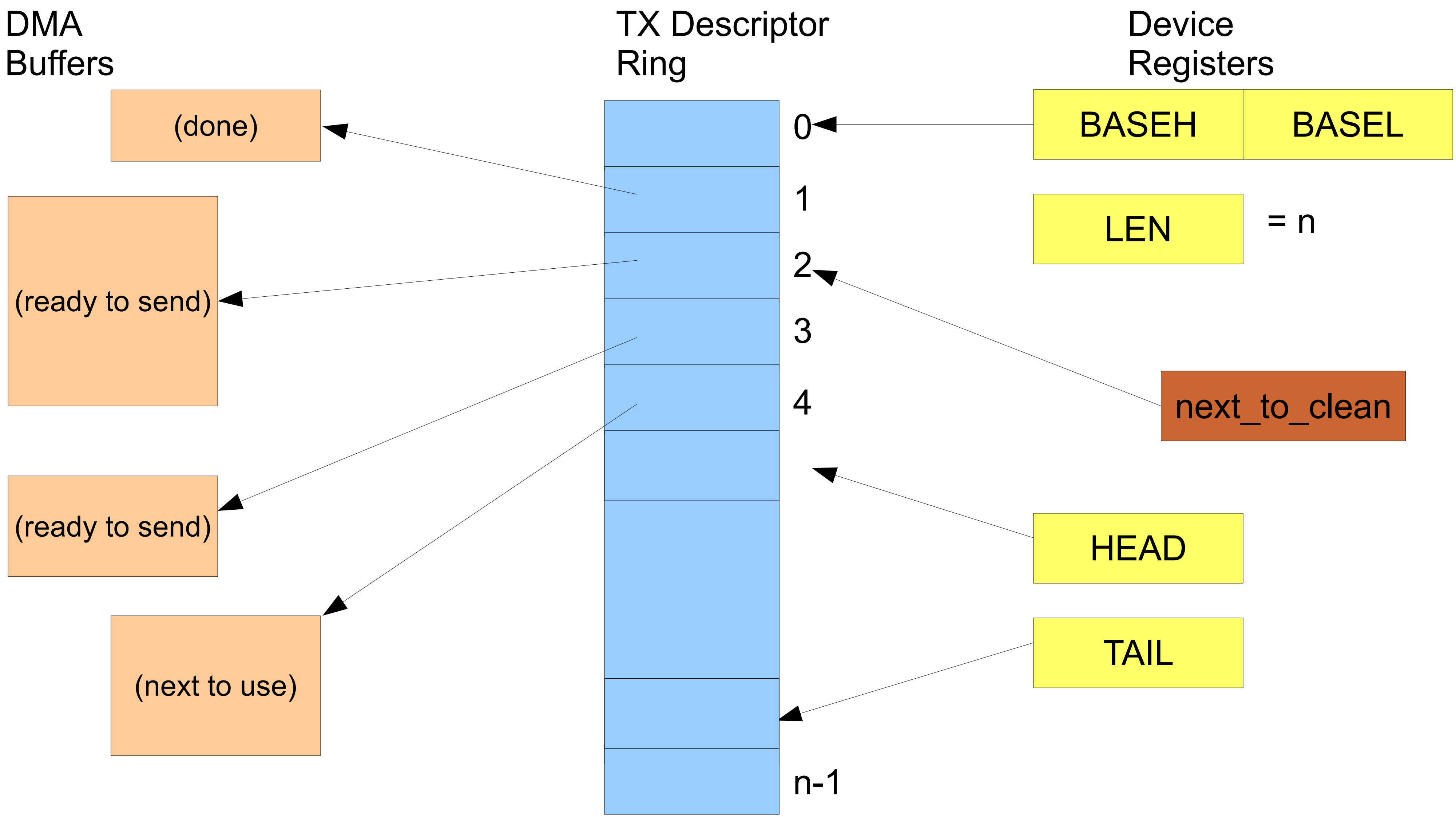
- Descriptors stored in ring buffer
  - Long array of descriptors
  - HW uses each through the array, then loops back
- Driver sets up each descriptor in ring as needed
- In order for hardware to use, what must we do?
- Pin it!



Cheesy Picture



Another Cheesy Picture



Filling out a ring

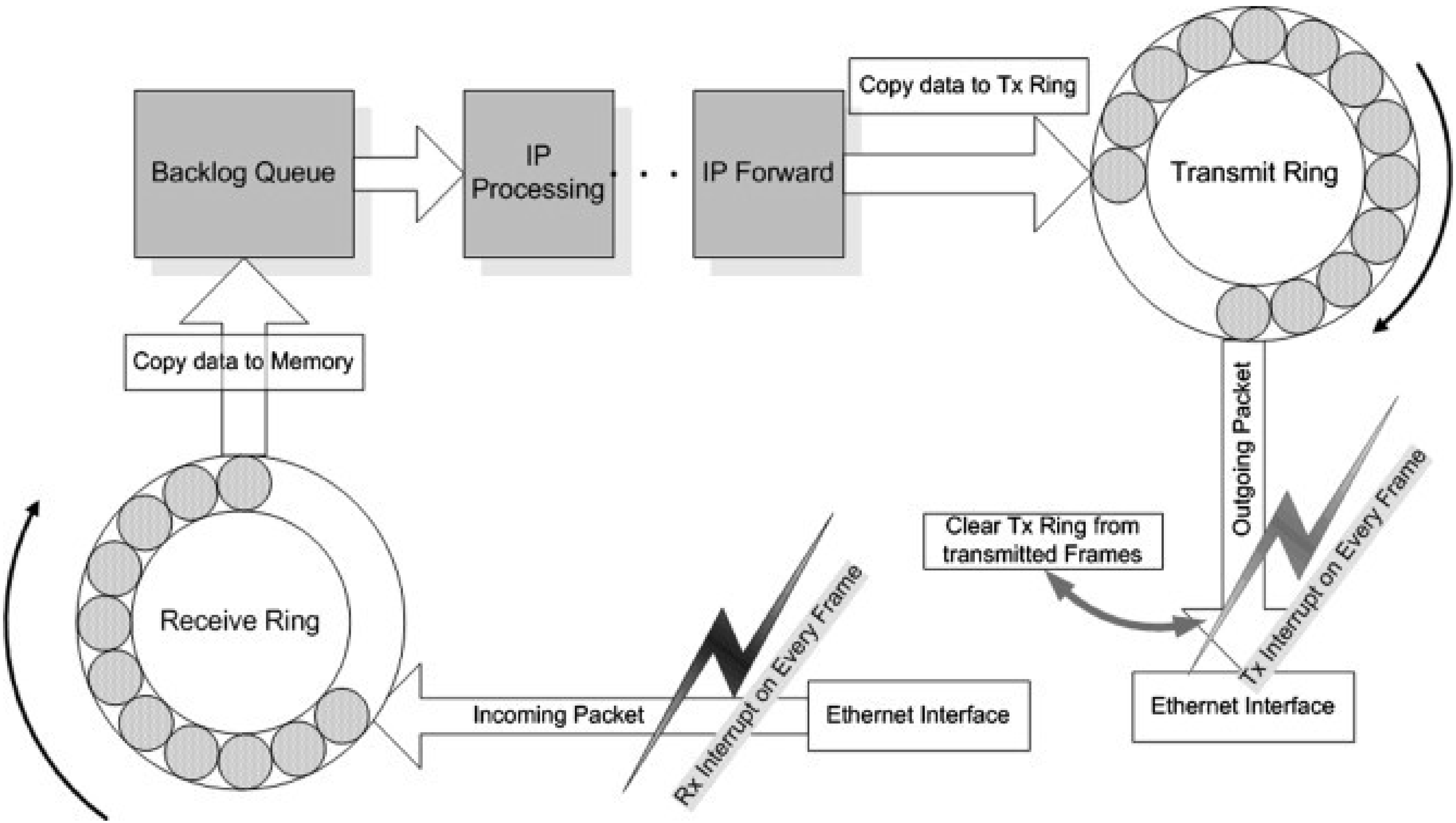
```
do {
        buffer_info = &tx_ring->buffer_info[i];
        tx_desc = E1000_TX_DESC(*tx_ring, i);
        tx_desc->buffer_addr = cpu_to_le64(buffer_info->dma);
        tx_desc->lower.data = cpu_to_le32(txd_lower | buffer_info->length);
        tx_desc->upper.data = cpu_to_le32(txd_upper);

        i++;
        if (i == tx_ring->count)
                i = 0;
} while (--count > 0);

tx_desc->lower.data |= cpu_to_le32(adapter->txd_cmd);

writel(i, E1000_TDT);    /* bump tail */
```

Managing rings



Setting up a network device

- Reset device
- Create descriptors
- Set promiscuous mode
- Set autoneg speed
- Force link up
- Start interrupt handlers

Sniffing around

- Wireshark
  - Good frontend to packet tracing applications
  - Used for anything that can stream data
- pcap
  - Packet capture file
- tcpdump
  - Unix/Linux-based packet tracer
- Save as a standard libpcap!

Play the packet trace

- Linux: tcpreplay
- Windows: PlayCap

- Takes pcap from previous trace, plays it back on the network
- Good for repeatable testing

Sending test packets

- Single packet (linux)
  - ping -I ethX -r 11.22.33.44
  - Other arguments let you shape the packet

- Packet streams
  - Linux: tcpreplay, Windows: PlayCap
  - Plays back packet capture from previous trace (tcpdump)
  - Good for repeatable testing

Readings

- LLD3 pg 440-453
- ELDD pg 301-310
  - Uses pci_xxx() rather than dma_xxx() API
- http://lxr.linux.no/#linux+v2.6.35.9/Documentation/DMA-API-HOWTO.txt
- http://lxr.linux.no/#linux+v2.6.35.9/Documentation/DMA-API.txt