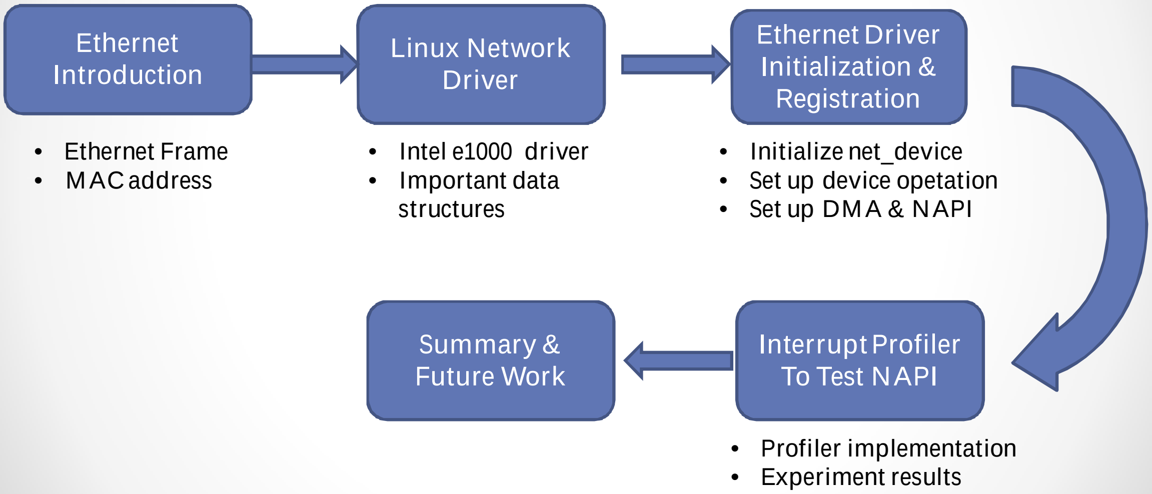
Understanding Linux Network Device Driver and NAPI Mechanism

Xinying Wang, Cong Xu

CS 423 Project

Outline



Ethernet Introduction

- Ethernet Frame
- MAC address

Linux Network Driver

- Intel e1000 driver
- Important data structures

Ethernet Driver Initialization & Registration

- Initialize net_device
- Set up device opetation
- Set up DMA & NAPI

Interrupt Profiler To Test NAPI

- Profiler implementation
- Experiment results

Summary & Future Work

Introduction to Ethernet

- A family of computer networking technologies for local area networks (LANs)
- Commercially introduced in 1980 and standardized in 1983 as IEEE 802.3.
- The most popular network with good degree of compatibility
- Features:
    - Ethernet frame
    - MAC Address

Ethernet frame

- Transported by Ethernet packet (a data packet on an Ethernet)
- Example of Ethernet frame structure through TCP socket:

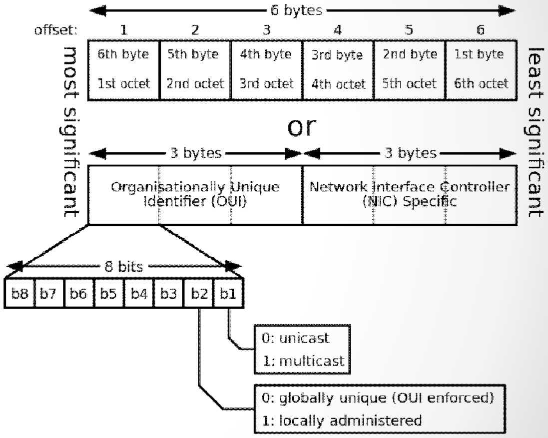
Ethernet header	IP header	TCP header	Data	Frame Check Sequence (FCS)

- Ethernet header
    - Header: a set of bytes (octets*) prepended to a packet
    - Include destination MAC address and source MAC address
- FCS: to detect any in-transit corruption of data

*octet: a group of eight bits

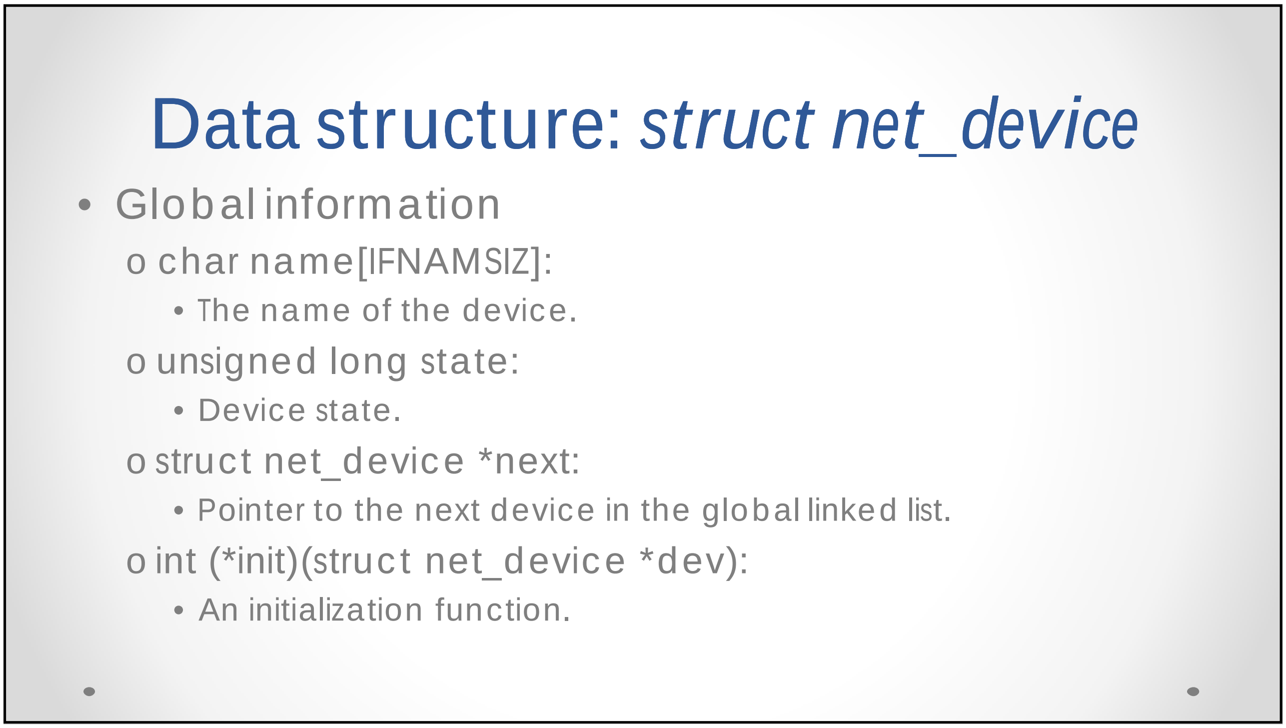
MAC address

- Media Access Control address
- Often stored in hardware's read-only memory
- First three octets: Organizationally Unique Identifier (OUI)
- Following octets: as long as unique



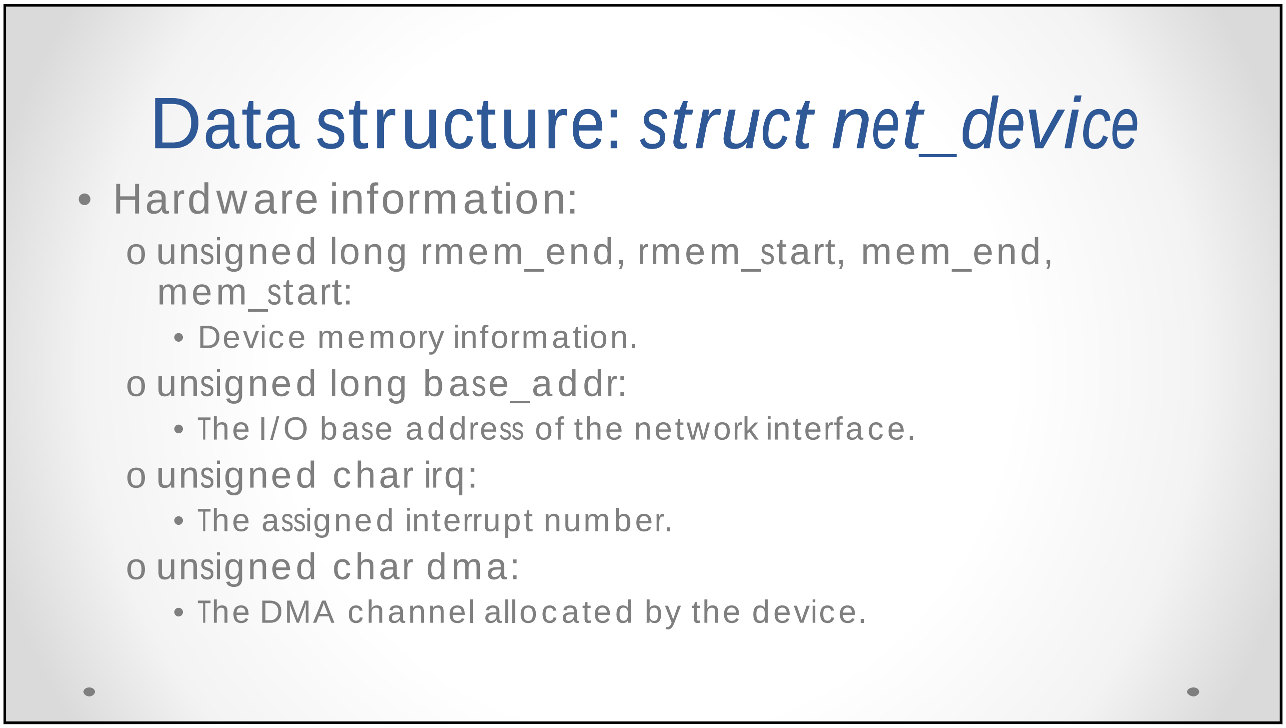
Linux network driver

- Linux kernel handles MAC address resolution.
- Network drivers are still needed
    - Kernel cannot do anything
    - Different from character drivers and block drivers
- Intel e1000 driver for Ethernet adapter
    - /drivers/net/ethernet/intel/e1000



Data structure: *struct net_device*

- Global information
  - o char name[IFNAMSIZ]:
    - The name of the device.
  - o unsigned long state:
    - Device state.
  - o struct net_device *next:
    - Pointer to the next device in the global linked list.
  - o int (*init)(struct net_device *dev):
    - An initialization function.



Data structure: *struct net_device*

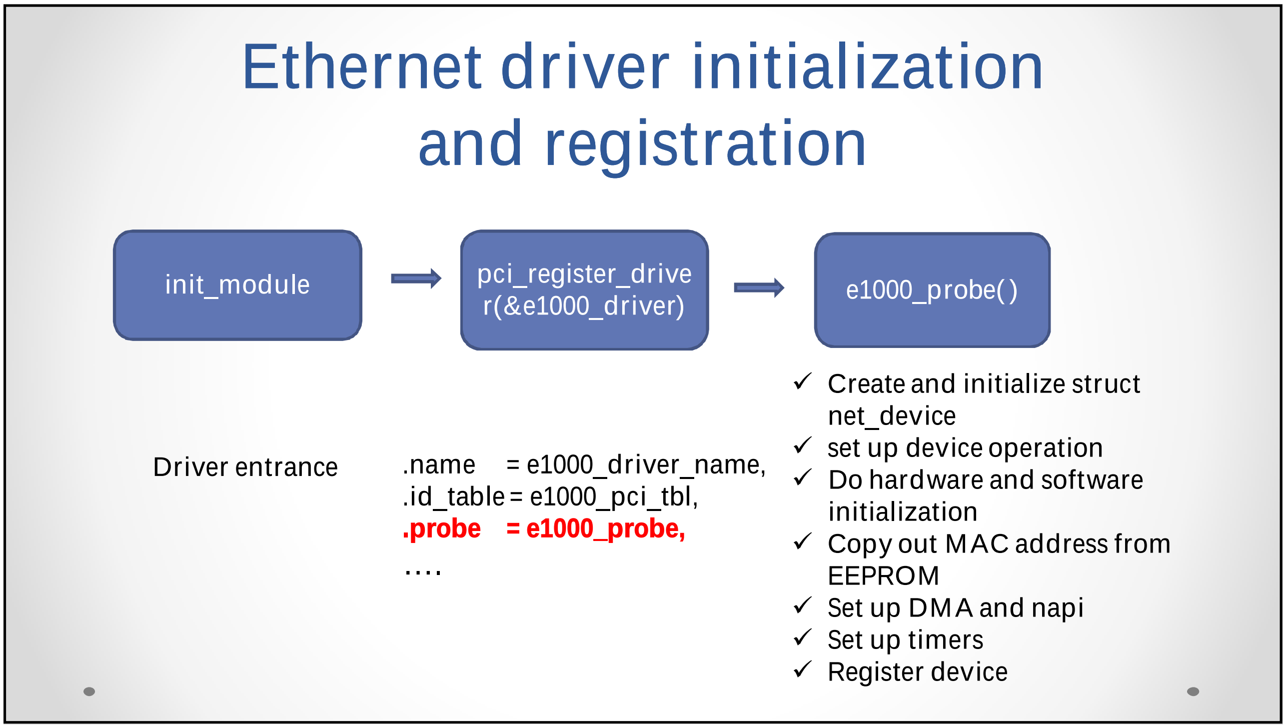
- Hardware information:
  - o unsigned long rmem_end, rmem_start, mem_end, mem_start:
    - Device memory information.
  - o unsigned long base_addr:
    - The I/O base address of the network interface.
  - o unsigned char irq:
    - The assigned interrupt number.
  - o unsigned char dma:
    - The DMA channel allocated by the device.

Data structure: *struct e1000_adapter*

- struct net_device *netdev;
    - Pointer to net_device struct
- struct pci_dev *pdev;
    - Pointer to pci_device struct
- struct e1000_hw hw;
    - An e1000_hw struct
- struct e1000_hw_stats stats;
    - Statistics counters collected by the MAC

Data structure: *struct e1000_hw*

- e1000_mac_type mac_type;
    - An enum for currently available devices
- u8 mac_addr[NODE_ADDRESS_SIZE];
    - MAC address
- u16 device_id;
    - Device identification information
- u16 vendor_id;
    - Vendor information



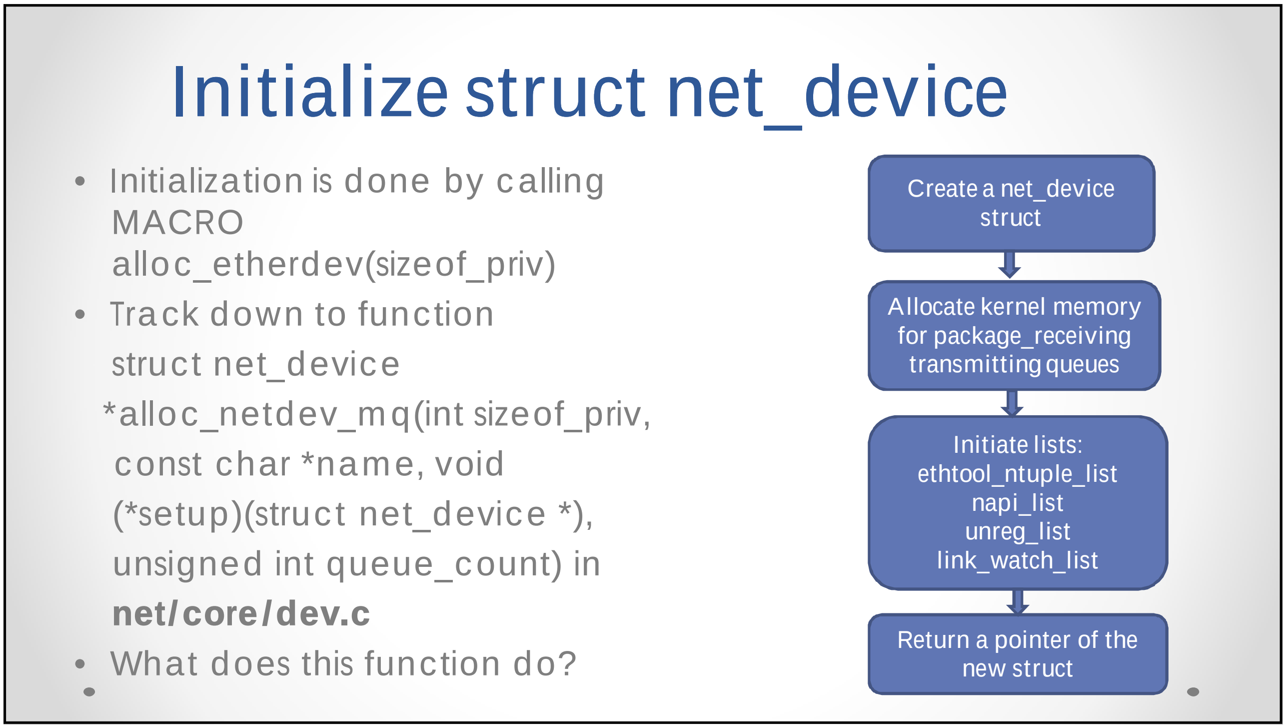
Ethernet driver initialization and registration

init_module → pci_register_driver(&e1000_driver) → e1000_probe()

Driver entrance

.name = e1000_driver_name,
.id_table = e1000_pci_tbl,
.probe = e1000_probe,
….

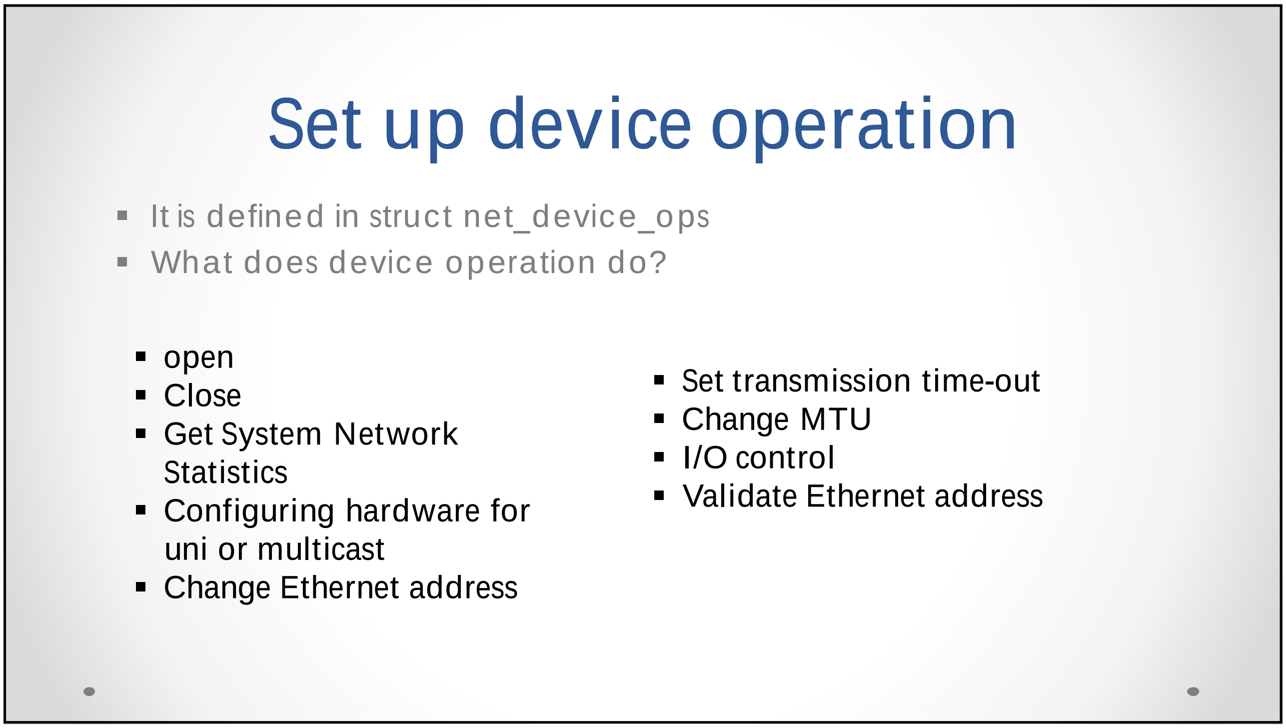
- ✓ Create and initialize struct net_device
- ✓ set up device operation
- ✓ Do hardware and software initialization
- ✓ Copy out MAC address from EEPROM
- ✓ Set up DMA and napi
- ✓ Set up timers
- ✓ Register device



Initialize struct net_device

- Initialization is done by calling MACRO alloc_etherdev(sizeof_priv)
- Track down to function struct net_device *alloc_netdev_mq(int sizeof_priv, const char *name, void (*setup)(struct net_device *), unsigned int queue_count) in **net/core/dev.c**
- What does this function do?

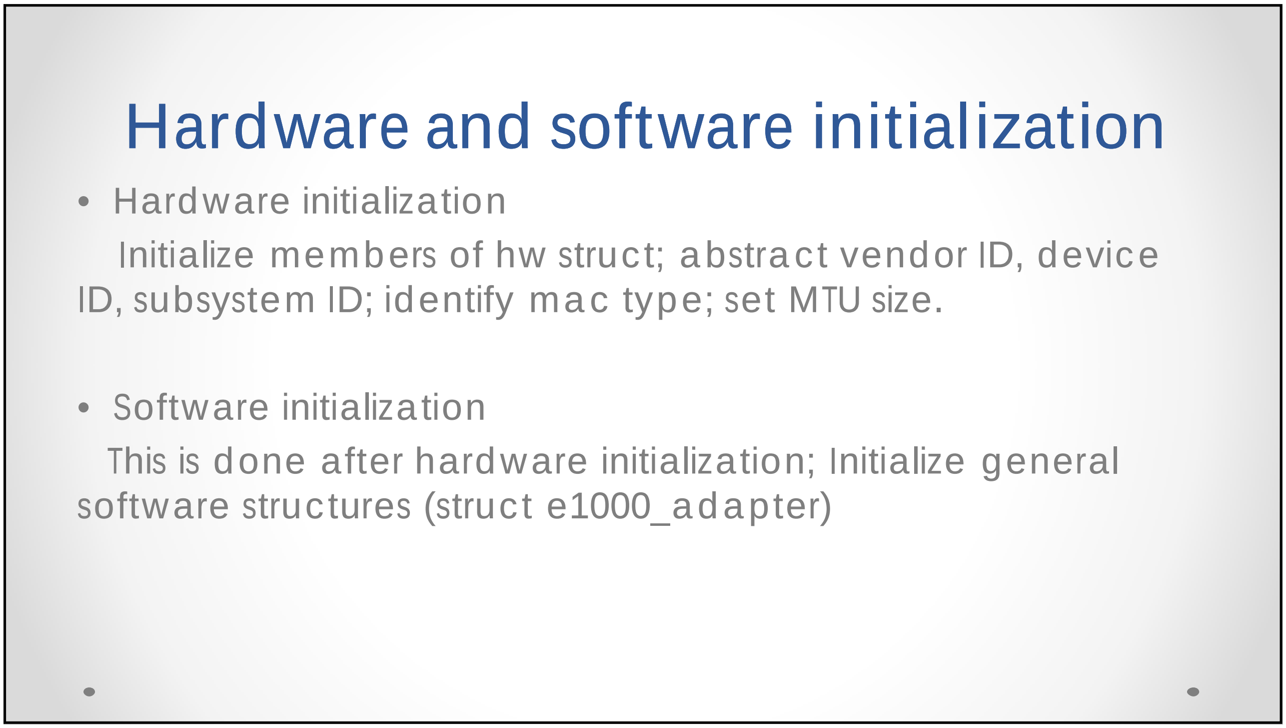
Create a net_device struct
↓
Allocate kernel memory for package_receiving transmitting queues
↓
Initiate lists:
ethtool_ntuple_list
napi_list
unreg_list
link_watch_list
↓
Return a pointer of the new struct



Set up device operation

- It is defined in struct net_device_ops
- What does device operation do?

- open
- Close
- Get System Network Statistics
- Configuring hardware for uni or multicast
- Change Ethernet address
- Set transmission time-out
- Change MTU
- I/O control
- Validate Ethernet address



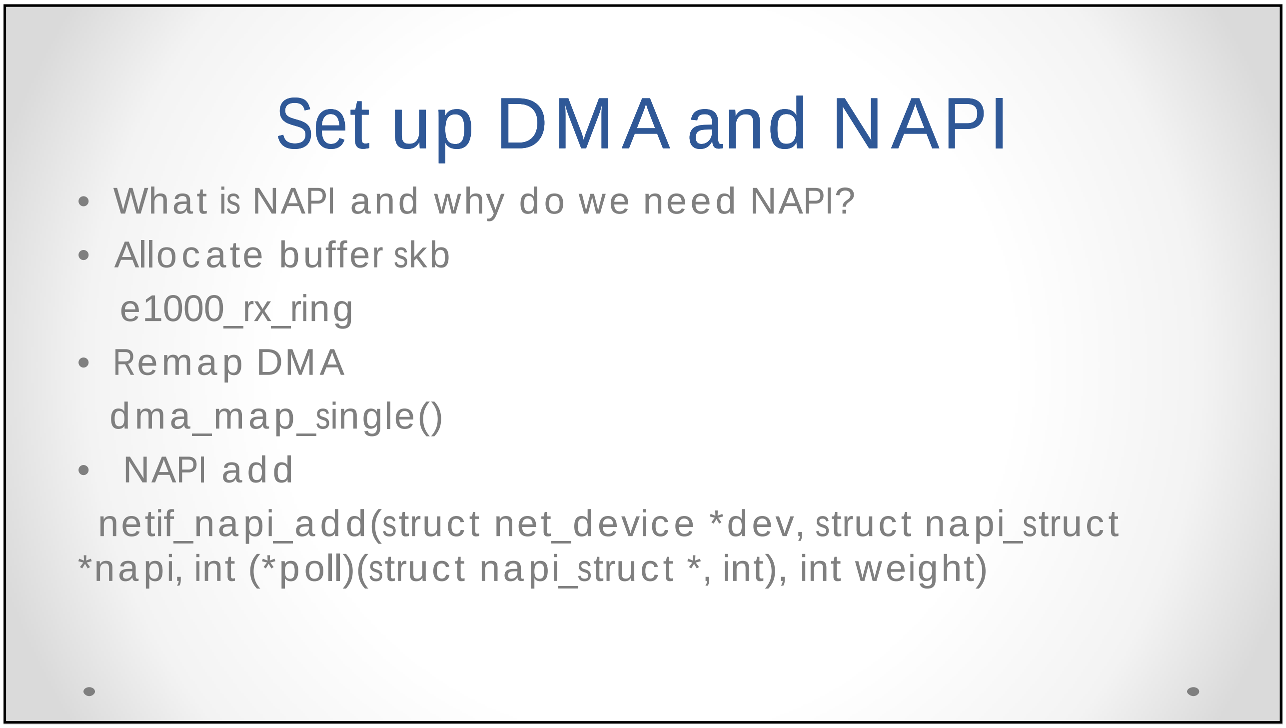
Hardware and software initialization

- Hardware initialization

Initialize members of hw struct; abstract vendor ID, device ID, subsystem ID; identify mac type; set MTU size.

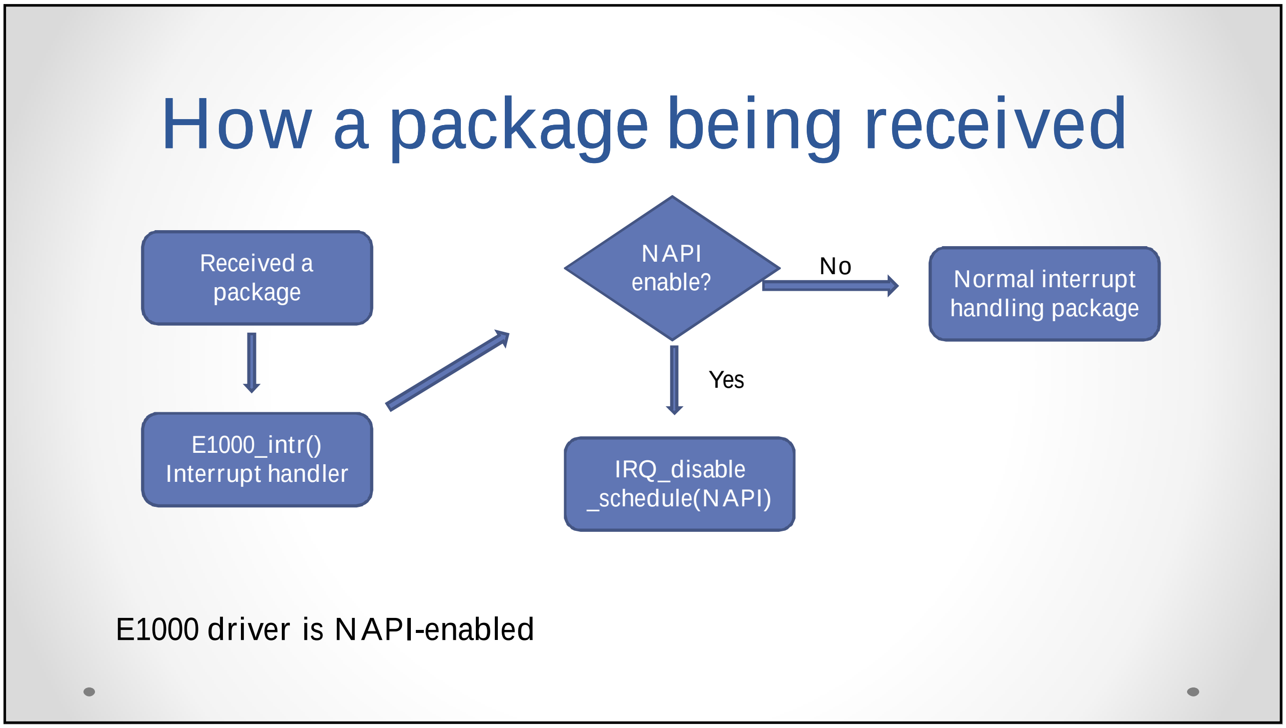
- Software initialization

This is done after hardware initialization; Initialize general software structures (struct e1000_adapter)



Set up DMA and NAPI

- What is NAPI and why do we need NAPI?
- Allocate buffer skb
  e1000_rx_ring
- Remap DMA
  dma_map_single()
- NAPI add
  netif_napi_add(struct net_device *dev, struct napi_struct *napi, int (*poll)(struct napi_struct *, int), int weight)



How a package being received

Received a package

E1000_intr()
Interrupt handler

NAPI enable?

No

Normal interrupt handling package

Yes

IRQ_disable
_schedule(NAPI)

E1000 driver is NAPI-enabled

NAPI implementation

In interrupt handler function e1000_entr()

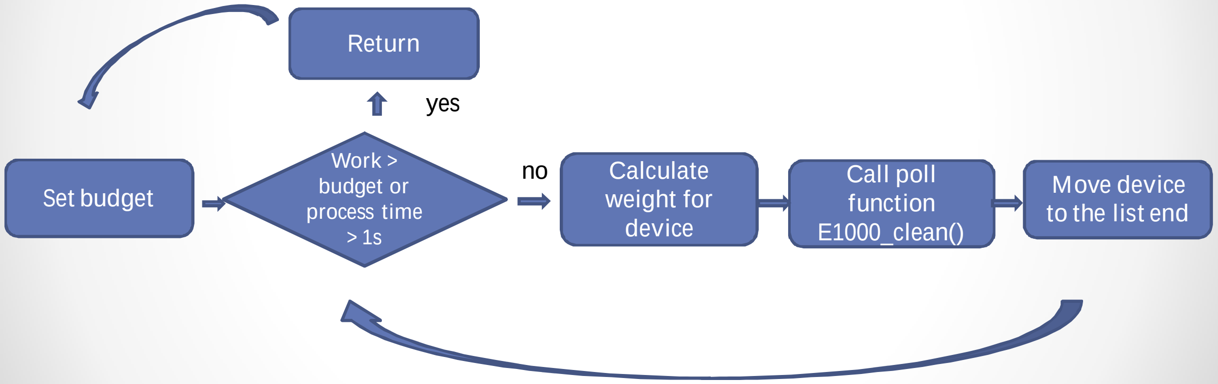
Make sure the net device is working properly
netif_rx_schedule_prep()

Add net device into poll list
_netif_rx_schedule()->napi_schedule()

_raise_softirq_irqoff(NET_RX_SOFTIRQ) for switching to bottom half

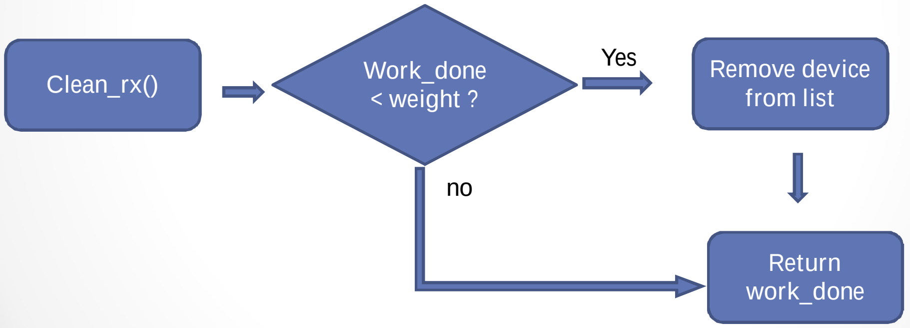
NAPI implementation

- Bottom half function net_rx_action()



NAPI implementation

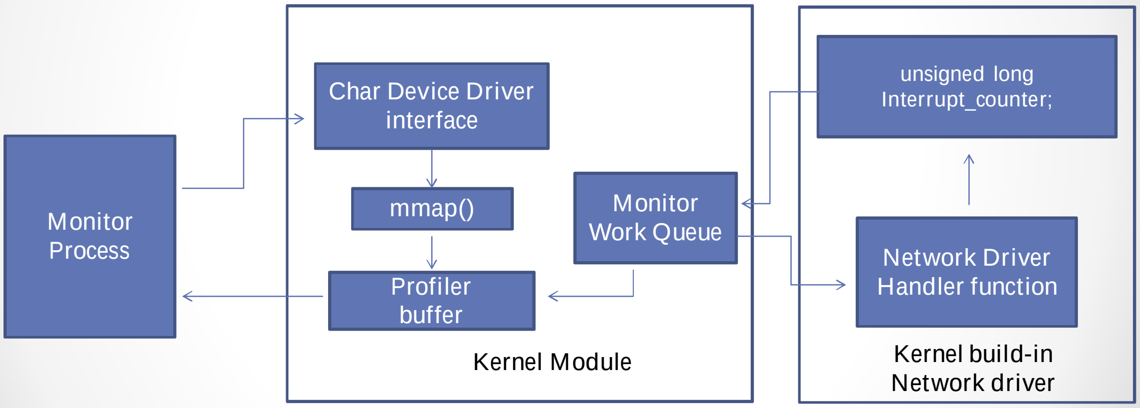
- Poll function e1000_clean()



Experiment

- An experiment is designed to test NAPI mechanism
- A interrupt profiler is designed to profile the interrupt counts in a designed period
- Linux kernel 3.13.6 was employed to fulfill the experiment
- Experiment platform: CPU: Intel core i5 dual core 2.53Ghz; Memory: 4G; Network card: Intel-82577 Gb card

Profiler implementation



Monitor Process

Char Device Driver interface

mmap()

Profiler buffer

Monitor Work Queue

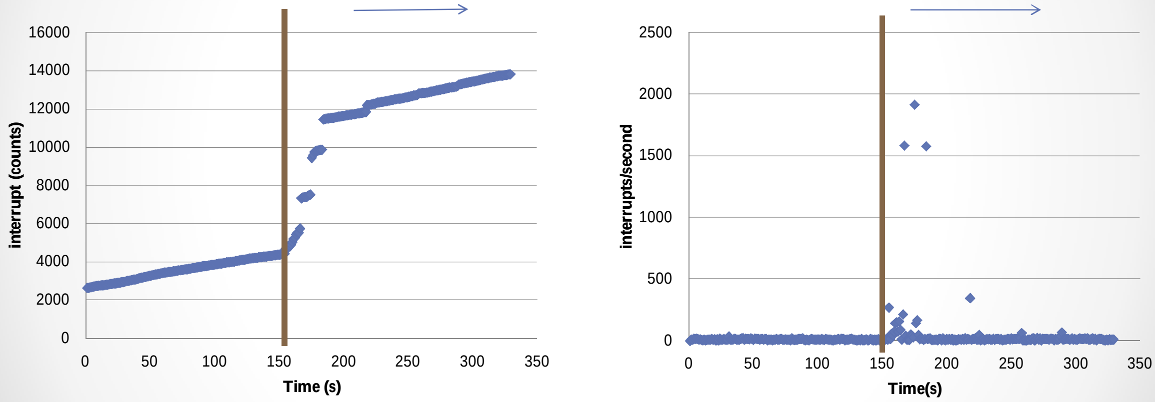
Kernel Module

unsigned long Interrupt_counter;

Network Driver Handler function

Kernel build-in Network driver

Results



streaming YouTube video

streaming YouTube video

Accumulated interrupts

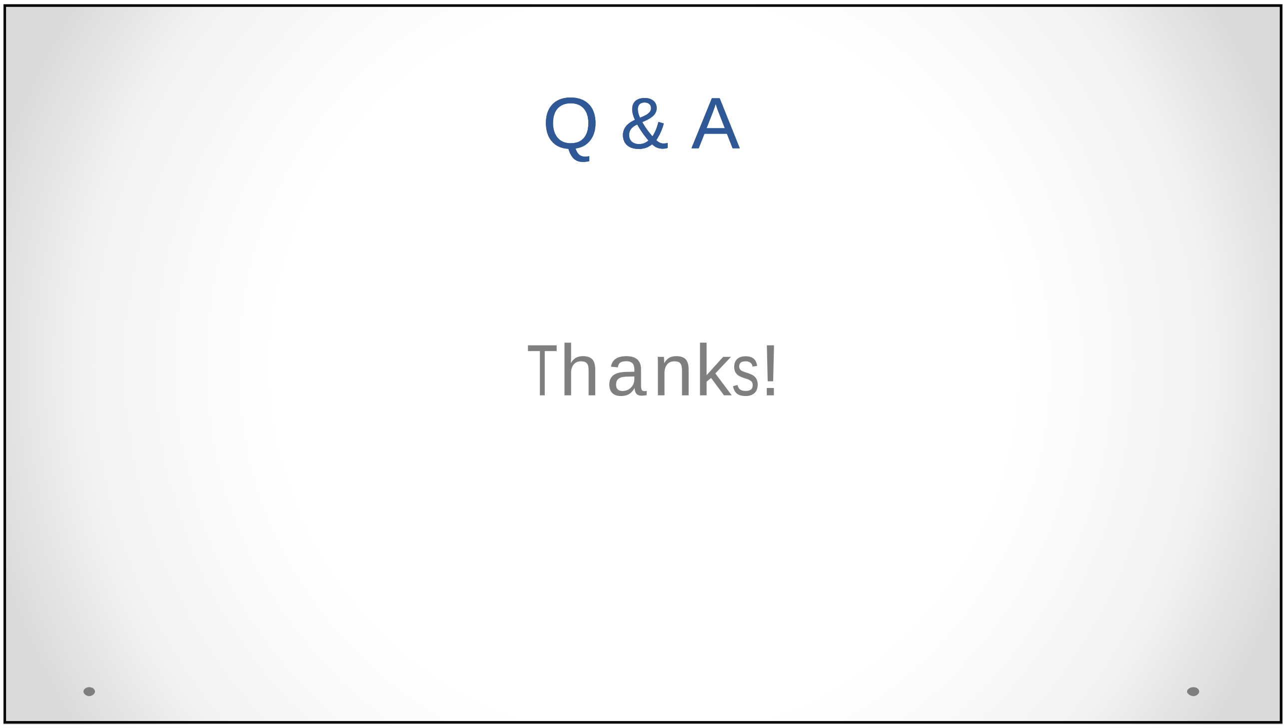
Differentiated interrupts

Summary and future work

- The Linux network device driver was analyzed base on Intel E1000 driver code files.
- The mechanism and implementation of NAPI was detailed
- An experiment was designed to further understand the NAPI mechanism
- A thorough understanding the Linux network device driver could be done for the future by further analysis of more sub functions.

Reference

- Branden Moore, Thomas Slabach, Lambert Schaelicke, Profiling Interrupt Handler Performance through Kernel Instrumentation, Proceedings of the 21st international conference on computer design
- Lambert Schaelicke, Al Davis, and Sally A. Mckee, Profiling I/O Interrupts in Modern Architectures
- Linux kernel source code (version 3.13.6)



Q & A

Thanks!