**ADRIAN PERRIG & TORSTEN HOEFLER**

# Networks and Operating Systems (252-0062-00)
# Chapter 10: I/O Subsystems (2)

## BREAKING FULL-DISK ENCRYPTION USING FIREWIRE

There have been a number of proof-of-concept hacks using IEEE1394 devices' DMA to elevate privileges on a host machine.
The most useful application of this technique is breaking into machines that use full-disk encryption.
Now there is a tool that will run from any Unix-Like host (Linux, OSX) and can unlock Windows XP,Vista,7,8,OSX 10.6,10.7,10.8, Ubuntu on both x86 and x64 hosts.

Inception is a FireWire physical memory manipulation and hacking tool exploiting IEEE 1394 SBP-2 DMA. The tool can unlock (any password accepted) and escalate privileges to Administrator/root on almost any machine you have physical access to.

It is primarily intended to do its magic against computers that utilize full disk encryption such as BitLocker, FileVault, TrueCrypt or Pointsec. There are plenty of other (and better) ways to hack a machine that doesn't pack encryption. Inception is also useful for incident response teams and digital forensics experts when faced with live machines.

**BE CAREFUL WITH I/O DEVICES!**

| OS | Version | Unlock lock screen | Escalate privileges | Dump memory < 4 GiB |
|---|---|---|---|---|
| Windows 8 | 8.1 | Yes | Yes | Yes |
| Windows 8 | 8.0 | Yes | Yes | Yes |
| Windows 7 | SP1 | Yes | Yes | Yes |
| Windows 7 | SP0 | Yes | Yes | Yes |
| Windows Vista | SP2 | Yes | Yes | Yes |
| Windows Vista | SP1 | Yes | Yes | Yes |
| Windows Vista | SP0 | Yes | Yes | Yes |
| Windows XP | SP3 | Yes | Yes | Yes |
| Windows XP | SP2 | Yes | Yes | Yes |
| Windows XP | SP1 | | | Yes |
| Windows XP | SP0 | | | Yes |
| Mac OS X | Mavericks | Yes (1) | Yes (1) | Yes (1) |
| Mac OS X | Mountain Lion | Yes (1) | Yes (1) | Yes (1) |
| Mac OS X | Lion | Yes (1) | Yes (1) | Yes (1) |
| Mac OS X | Snow Leopard | Yes | Yes | Yes |
| Mac OS X | Leopard | | | Yes |
| Ubuntu (2) | Saucy | Yes | Yes | Yes |
| Ubuntu | Raring | Yes | Yes | Yes |
| Ubuntu | Quantal | Yes | Yes | Yes |
| Ubuntu | Precise | Yes | Yes | Yes |
| Ubuntu | Oneiric | Yes | Yes | Yes |
| Ubuntu | Natty | Yes | Yes | Yes |
| Ubuntu | Maverick | Yes (3) | Yes (3) | Yes |
| Ubuntu | Lucid | Yes (3) | Yes (3) | Yes |
| Linux Mint | 13 | Yes | Yes | Yes |
| Linux Mint | 12 | Yes | Yes | Yes |
| Linux Mint | 12 | Yes | Yes | Yes |

FIREWIRE
A   B   B
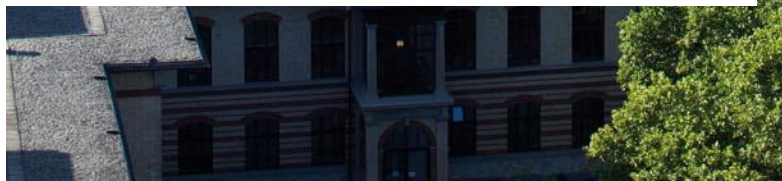
# Administrivia

- **If you're an exchange student**
  - … and very far away from Zurich during the exam period
  - … and you still want to take the exam

- **Then read: https://www.ethz.ch/students/en/studies/performance-assessments/exchange.html**

- **Constraints/boundary conditions:**
  - It has to be a written exam for fairness – same conditions for all students

- **Thus, what we offer is:**
  - We cannot offer preponement (cannot create 10+ different written exams)
  - But you can take the exam remotely
    *At the exact same time as in Zurich as coordinated by the office above*
    *Only in extreme cases, ETH may be able to move the time!*

# Our Small Quiz

- **True or false (raise hand)**
    - Open files are part of the process' address-space
    - Unified buffer caches improve the access times significantly
    - A partition table can unify the view of multiple disks
    - Unix enables to bind arbitrary file systems to arbitrary locations
    - The virtual file system interface improves modularity of OS code
    - Programmed I/O is efficient for the CPUs
    - DMA enables devices to access virtual memory of processes
    - IOMMUs enable memory protection for devices
    - IOMMUs improve memory access performance
    - First level interrupt handlers process the whole request from the hardware
    - Software interrupts reduce the request processing latency
    - Deferred procedure calls execute second-level interrupt handlers

# The I/O subsystem

# Generic I/O functionality

- **Device drivers essentially move data to and from I/O devices**
  - Abstract hardware
  - Manage asynchrony

- **OS I/O subsystem includes generic functions for dealing with this data**
  - Such as…

# The I/O Subsystem

- **Caching - fast memory holding copy of data**
  - Always just a copy
  - Key to performance

- **Spooling - hold output for a device**
  - If device can serve only one request at a time
  - E.g., printing

# The I/O Subsystem

- **Scheduling**
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness

- **Buffering - store data in memory while transferring between devices or memory**
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain "copy semantics"

# Naming and Discovery

- **What are the devices the OS needs to manage?**
  - Discovery (bus enumeration)
  - Hotplug / unplug events
  - Resource allocation (e.g., PCI BAR programming)

- **How to match driver code to devices?**
  - Driver instance ≠ driver module
  - One driver typically manages many models of device

- **How to name devices inside the kernel?**

- **How to name devices outside the kernel?**

# Matching drivers to devices

- **Devices have unique (model) identifiers**
  - E.g., PCI vendor/device identifiers

- **Drivers recognize particular identifiers**
  - Typically a list…

- **Kernel offers a device to each driver in turn**
  - Driver can "claim" a device it can handle
  - Creates driver instance for it.

# Naming devices in the Unix kernel

## (Actually, naming *device driver instances*)

- **Kernel creates identifiers for**
  - Block devices
  - Character devices
  - *[ Network devices – see later… ]*

- **Major device number:**
  - Class of device (e.g., disk, CD-ROM, keyboard)

- **Minor device number:**
  - Specific device within a class

# Unix Block Devices

- **Used for "structured I/O"**
  - Deal in large "blocks" of data at a time

- **Often look like files (seekable, mappable)**
  - Often use Unix' shared buffer cache

- **Mountable:**
  - File systems implemented above block devices

# Character Devices

- **Used for "unstructured I/O"**
  - Byte-stream interface – no block boundaries
  - Single character or short strings get/put
  - Buffering implemented by libraries


- **Examples:**
  - Keyboards, serial lines, mice


- **Distinction with block devices somewhat arbitrary…**

# Mid-lecture mini-quiz

- **Character or block device (raise hand)**
  - Console
  - USB stick
  - Microphone
  - Screen (graphics adapter)
  - Network drive
  - Webcam

# Naming devices outside the kernel

- **Device files: special type of *file***
  - Inode encodes <type, major num, minor num>
  - Created with `mknod()` system call

- **Devices are traditionally put in `/dev`**
  - `/dev/sda` – First SCSI/SATA/SAS disk
  - `/dev/sda5` – Fifth partition on the above
  - `/dev/cdrom0` – First DVD-ROM drive
  - `/dev/ttyS1` – Second UART

# Pseudo-devices in Unix

- **Devices with no hardware!**
- **Still have major/minor device numbers. Examples:**

```
/dev/stdin (was a device earlier, now link)
/dev/kmem
/dev/random
/dev/null
/dev/loop0
```

**etc.**

# Old-style Unix device configuration

- **All drivers compiled into the kernel**
- **Each driver probes for any supported devices**
- **System administrator populates `/dev`**
    - Manually types `mknod` when a new device is purchased!
- **Pseudo devices similarly hard-wired in kernel**
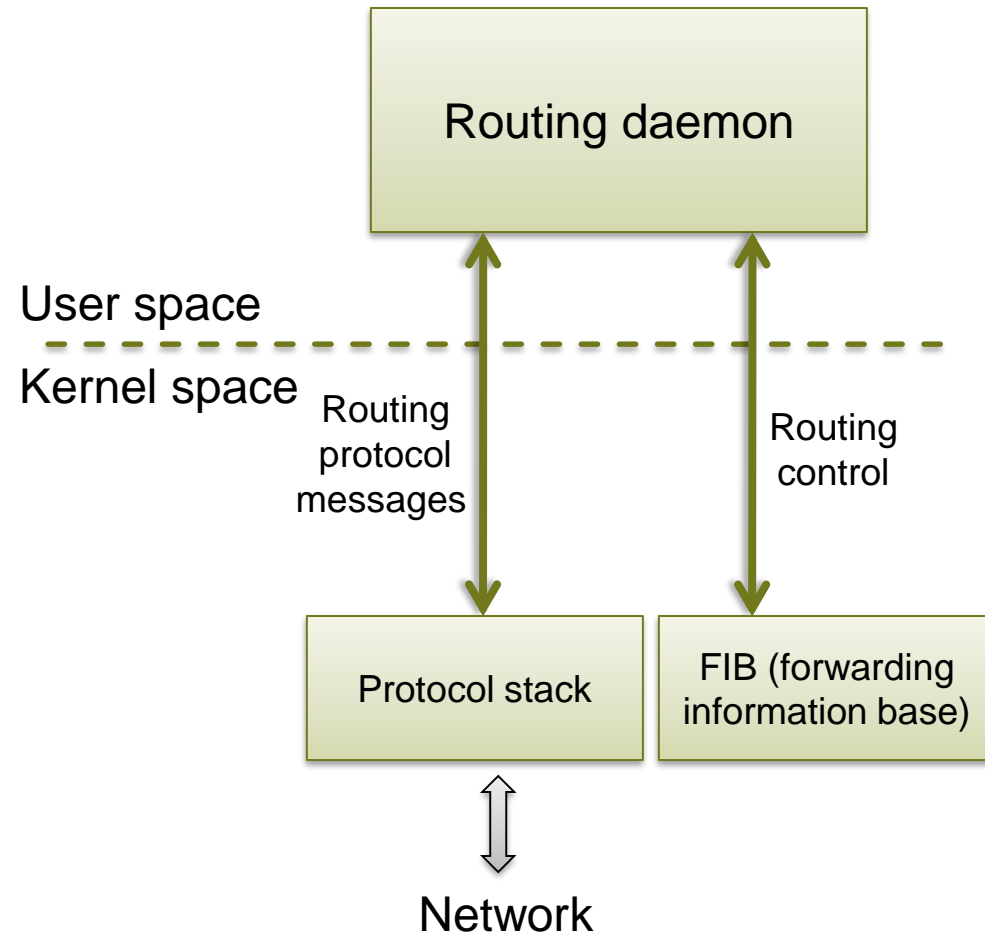
# Linux device configuration today

- **Physical hardware configuration readable from `/sys`**
  - Special fake file system: `sysfs`
  - Plug events delivered by a special socket
- **Drivers dynamically loaded as kernel modules**
  - Initial list given at boot time
  - User-space daemon can load more if required
- **`/dev` populated dynamically by `udev`**
  - User-space daemon which polls `/sys`

# Interface to network I/O

# Unix interface to network I/O

- **You will soon know the data path**
  - BSD sockets
  - `bind(), listen(), accept(), connect(), send(), recv(),` etc.
- **Have not yet seen:**
  - Device driver interface
  - Configuration
  - Routing

# Software routing
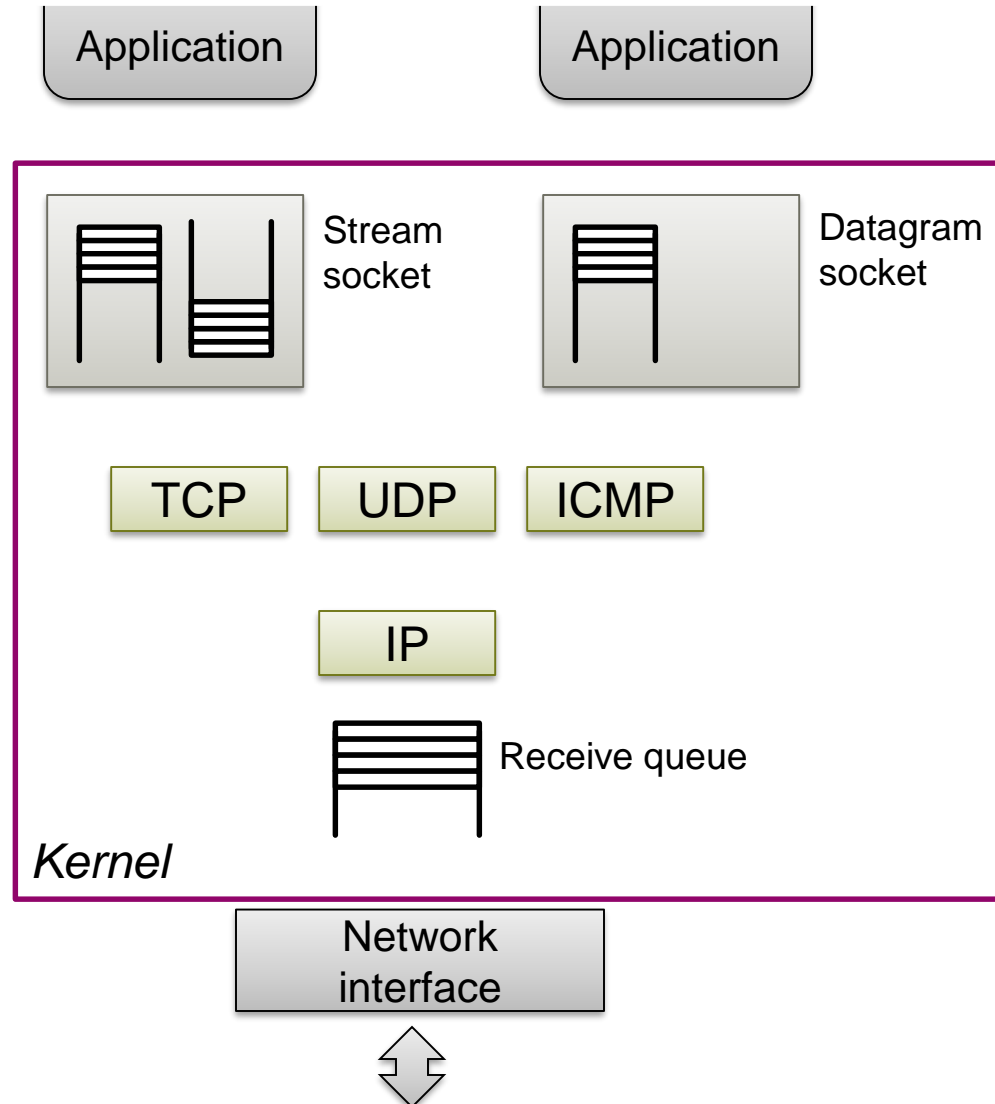
Routing daemon

User space
- - - - - - - - - - - - - - - - - - - - - - -
Kernel space

Routing protocol messages

Routing control

Protocol stack

FIB (forwarding information base)

Network

- **OS protocol stacks include routing functionality**
- **Routing *protocols* typically in a user-space daemon**
  - Non-critical
  - Easier to change
- ***Forwarding* information typically in kernel**
  - Needs to be fast
  - Integrated into protocol stack

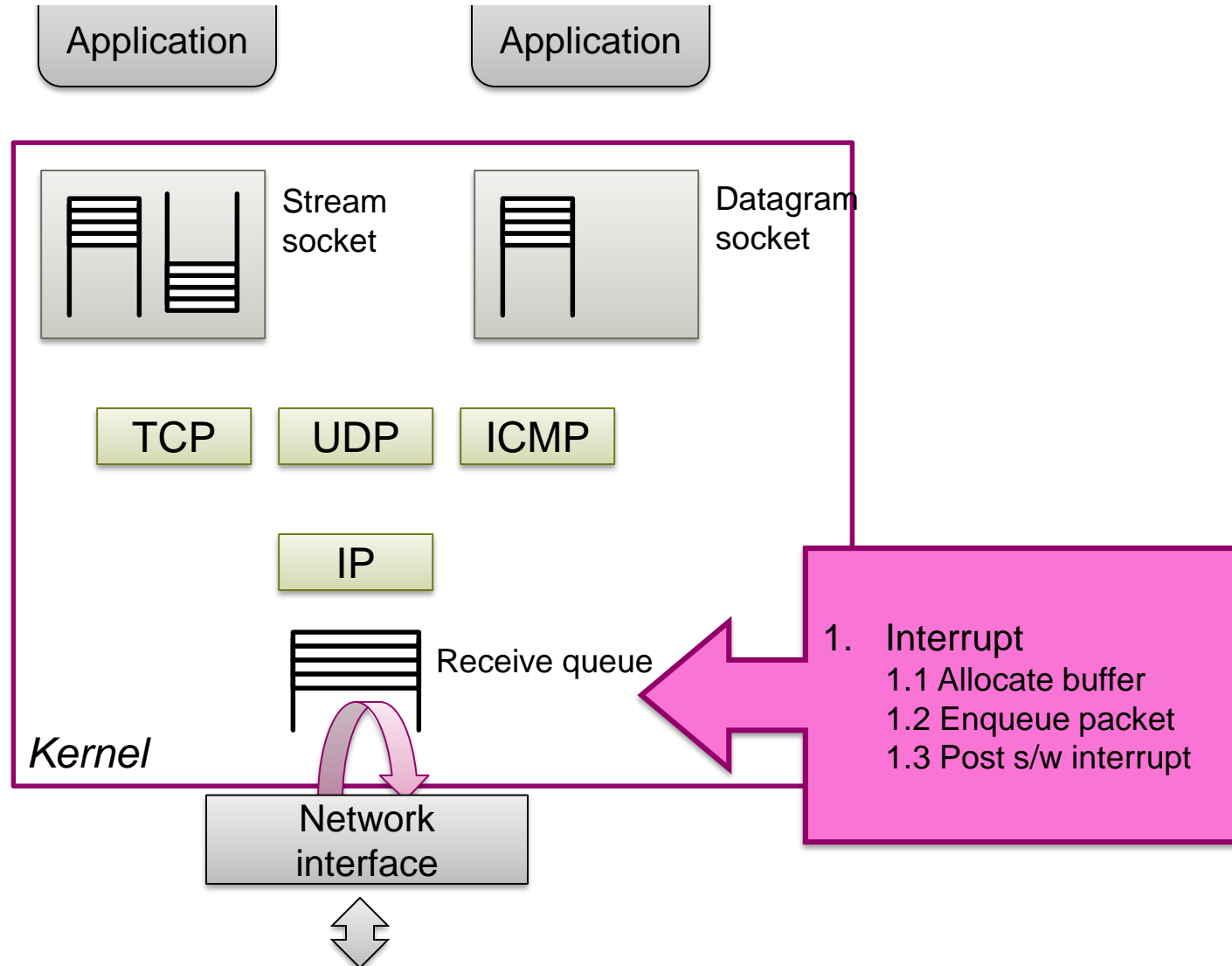# Network stack implementation

# Networking stack

- **Probably most important peripheral**
  - GPU is increasingly not a peripheral
  - Disk interfaces look increasingly like a network
- **But…**
  - NO standard OS textbook talks about the network stack!
- **Good references:**
  - The 4.4BSD book (for Unix at least)
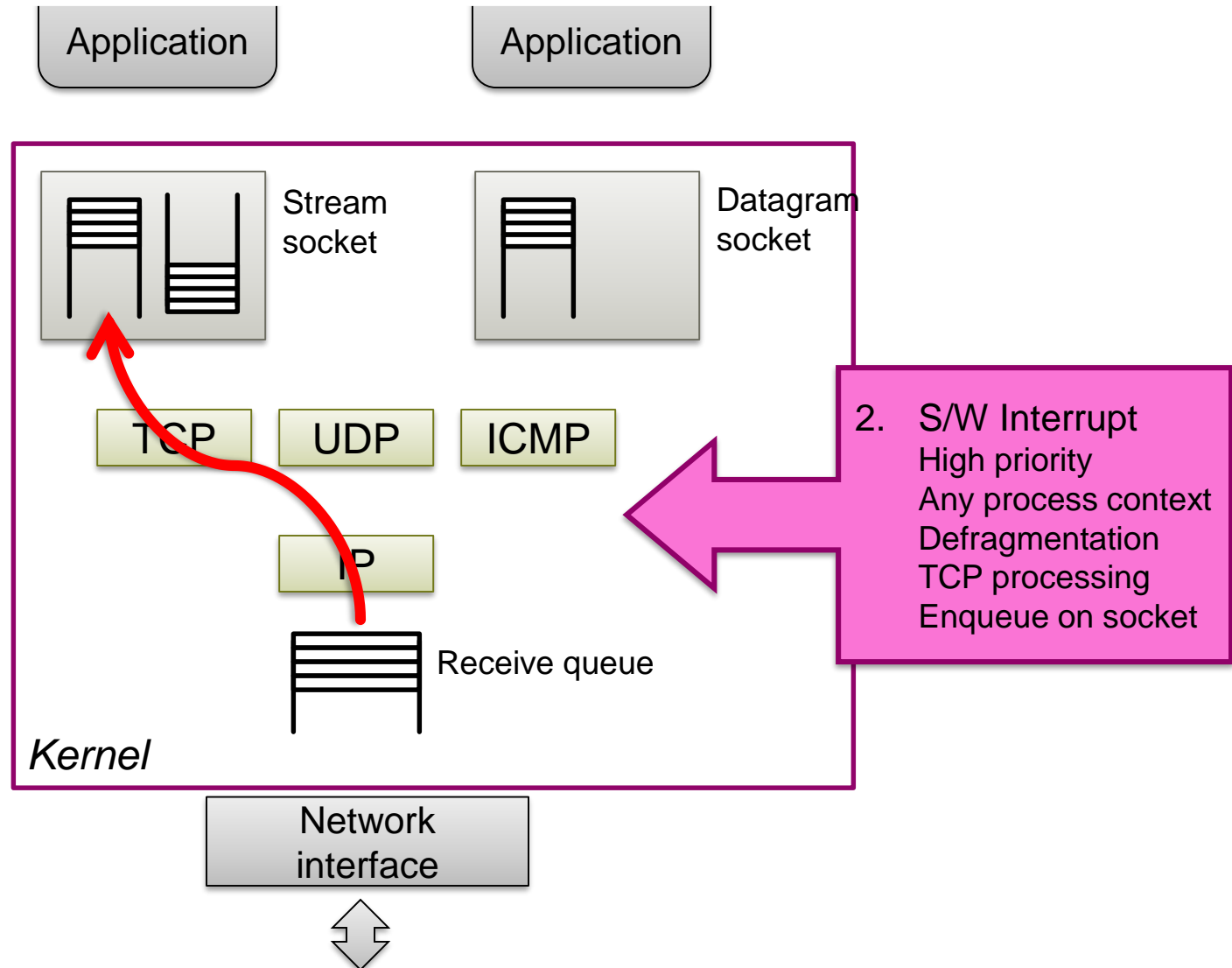  - George Varghese: "Network Algorithmics" (up to a point)
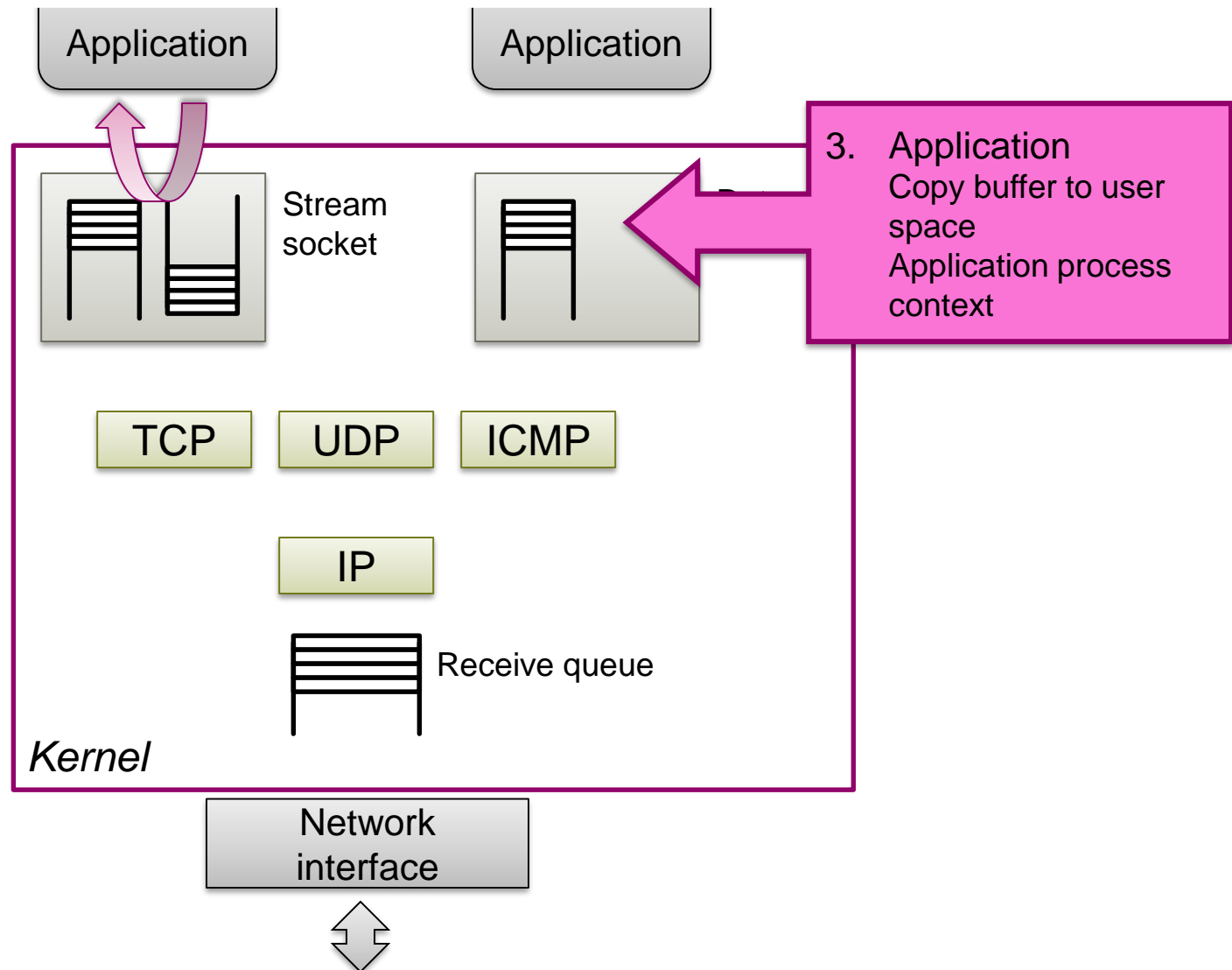
# Receiving a packet in BSD
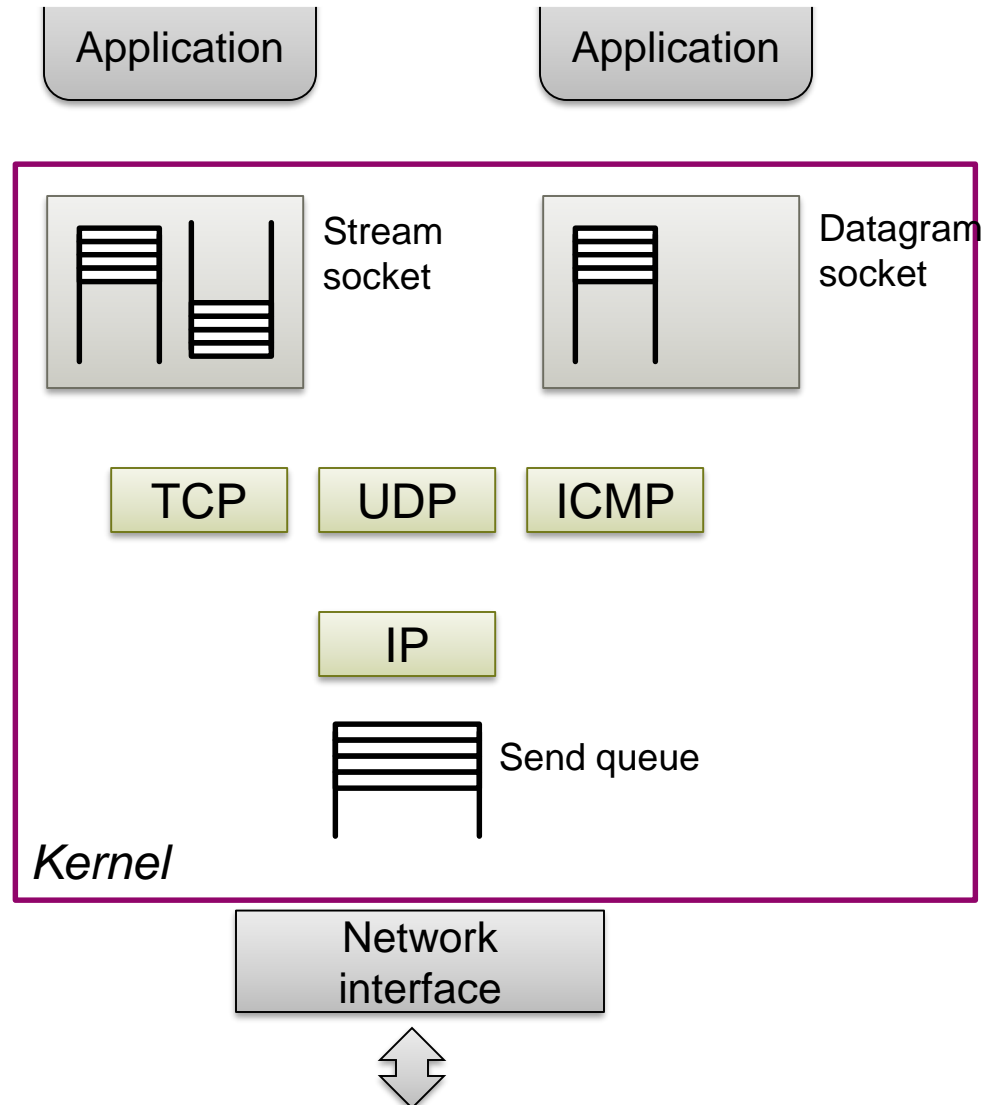
# Receiving a packet in BSD



Application

Application

Stream socket

Datagram socket

TCP    UDP    ICMP

IP

Receive queue

*Kernel*

Network interface

1. Interrupt
   1.1 Allocate buffer
   1.2 Enqueue packet
   1.3 Post s/w interrupt

# Receiving a packet in BSD



Application

Application

Stream socket

Datagram socket

TCP    UDP    ICMP

IP

Receive queue

*Kernel*

Network interface

2. S/W Interrupt
High priority
Any process context
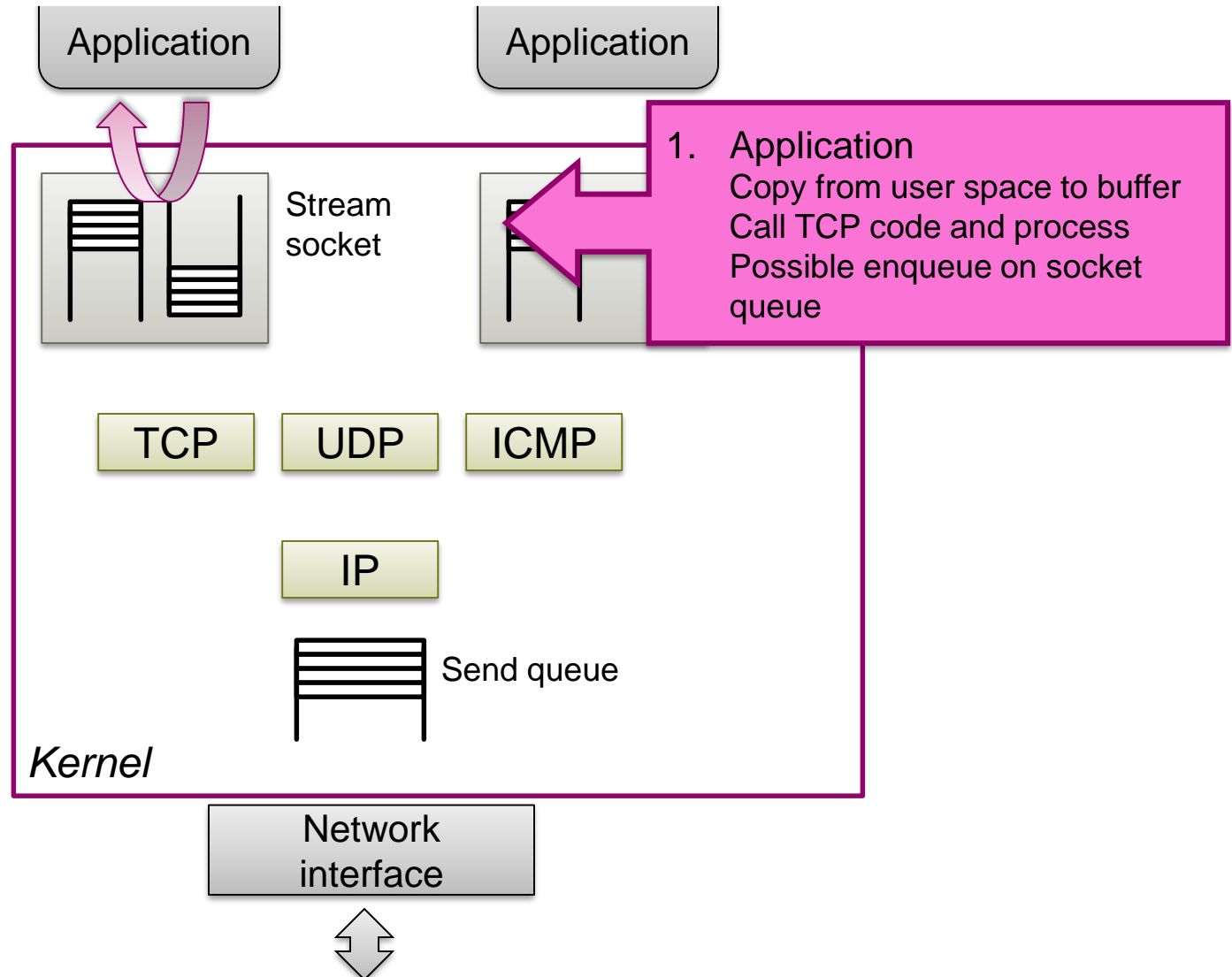Defragmentation
TCP processing
Enqueue on socket

# Receiving a packet in BSD

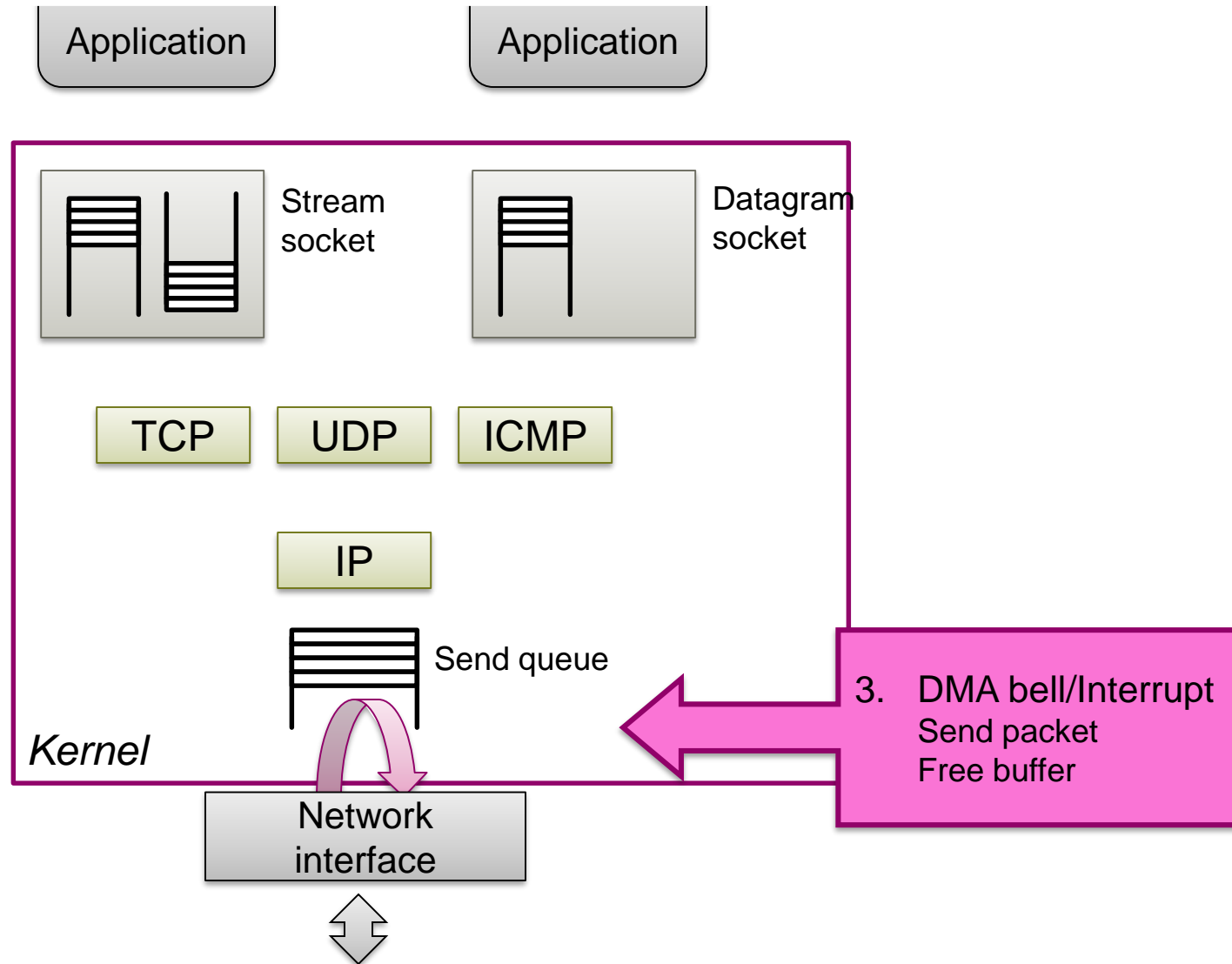# Sending a packet in BSD
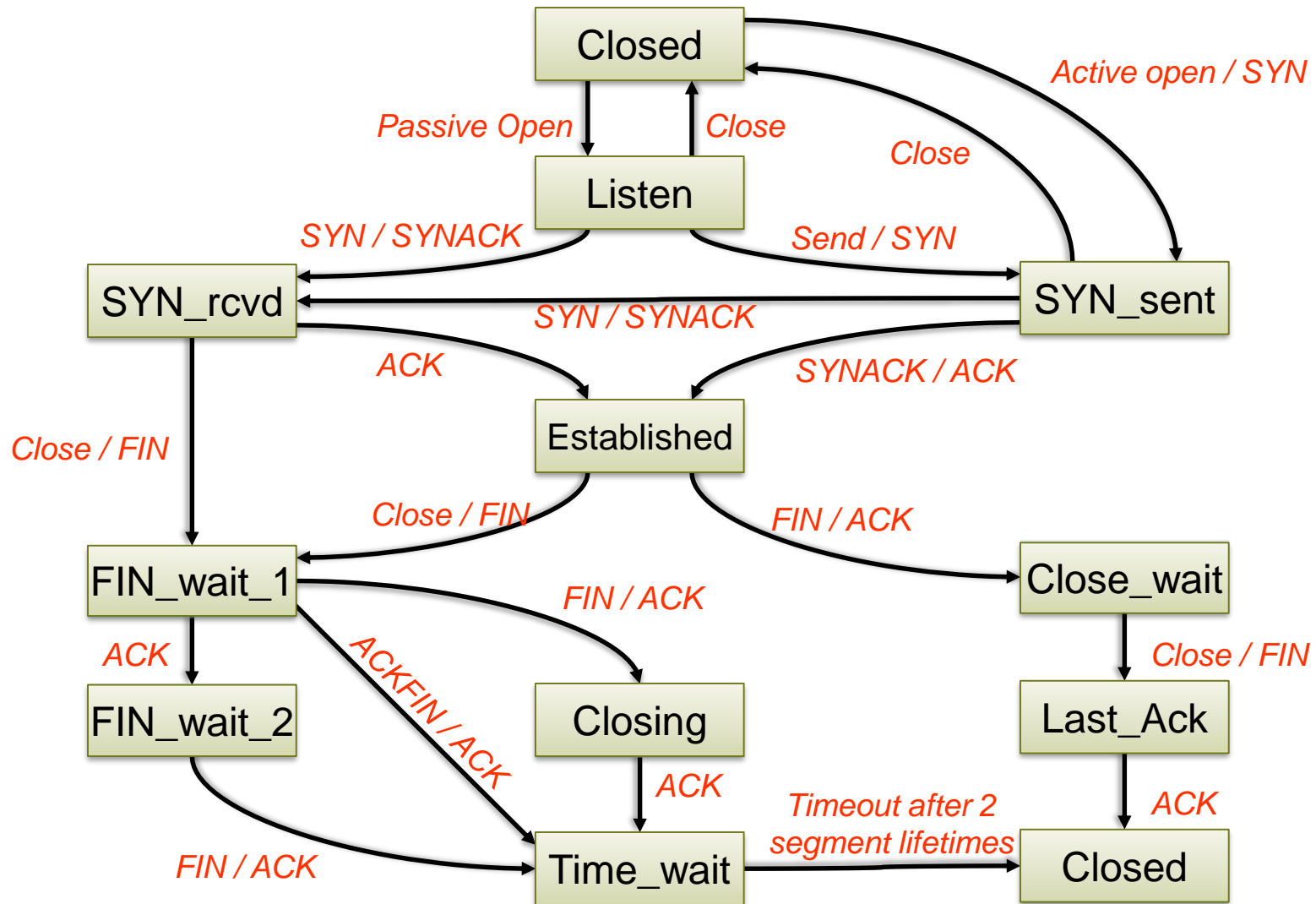
# Sending a packet in BSD



Application

Application

Stream socket

TCP    UDP    ICMP

IP

Send queue

*Kernel*

Network interface

1. Application
   Copy from user space to buffer
   Call TCP code and process
   Possible enqueue on socket queue

# Sending a packet in BSD



Application

Application

Stream socket

Datagram socket

TCP     UDP     ICMP

IP

2.  S/W Interrupt
    Any process context
    Remaining TCP
    processing
    IP processing
    Enqueue on i/f queue

Send queue

*Kernel*

Network interface

# Sending a packet in BSD

# The TCP state machine

# OS TCP state machine

- **More complex!  Also needs to handle:**
  - Congestion control state (window, slow start, etc.)
  - Flow control (window size)
  - Retransmission timeouts
  - Etc.
- **State transitions triggered when:**
  - User request: `send`, `recv`, `connect`, `close`
  - Packet arrives
  - Timer expires
- **Actions include:**
  - Set or cancel a timer
  - Enqueue a packet on the transmit queue
  - Enqueue a packet on the socket receive queue
  - Create or destroy a *TCP control block*

# In-kernel protocol graph
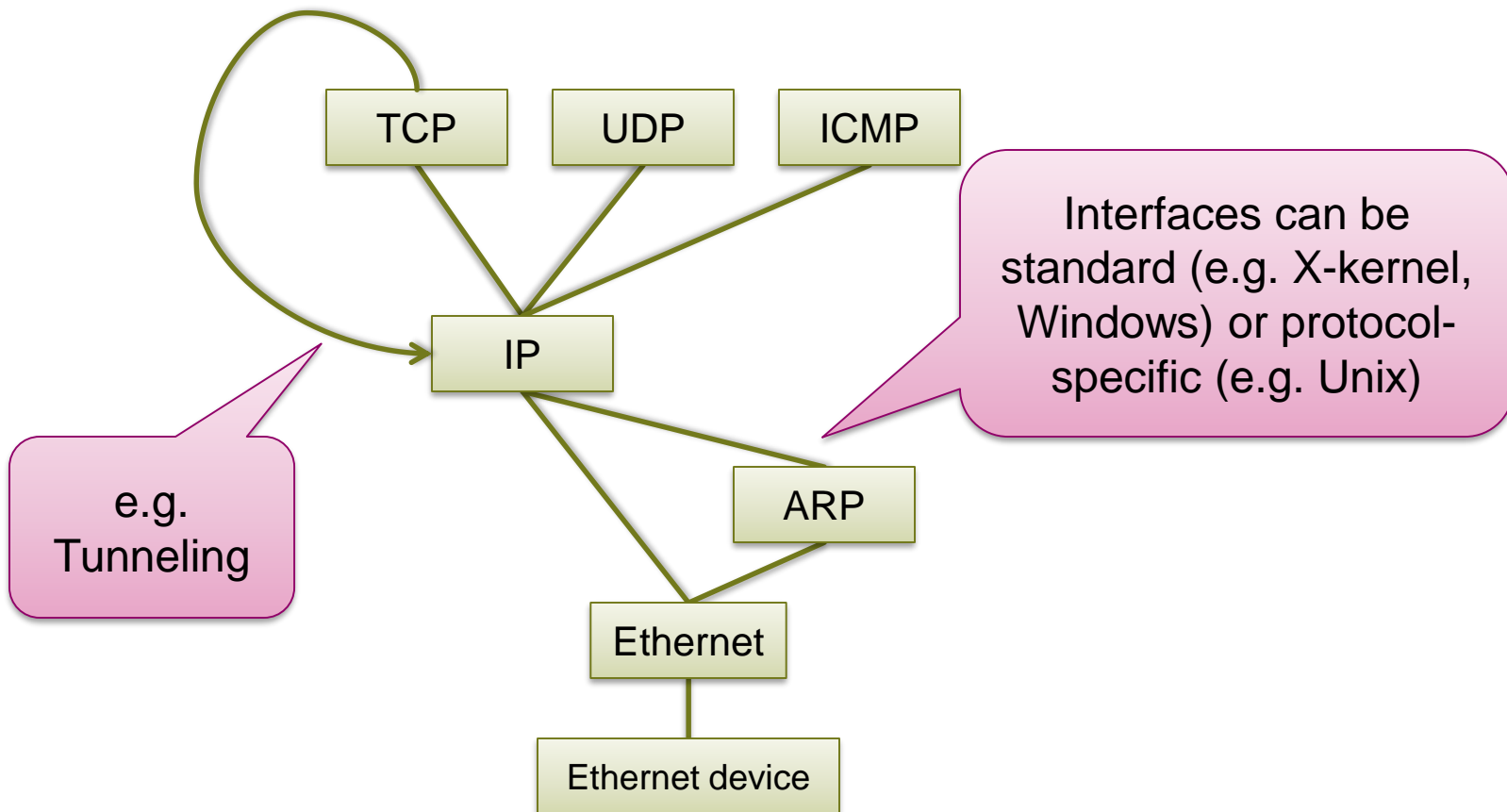
# Protocol graphs

**Graph nodes can be:**

- **Per-protocol (handle all flows)**
  - Packets are "tagged" with demux tags

- **Per-connection (instantiated dynamically)**
  - Multiple *interfaces* as well as connections
  - Ethernet $\leftrightarrow$ Ethernet $\Rightarrow$ bridging
  - IP $\leftrightarrow$ IP $\Rightarrow$ IP routing
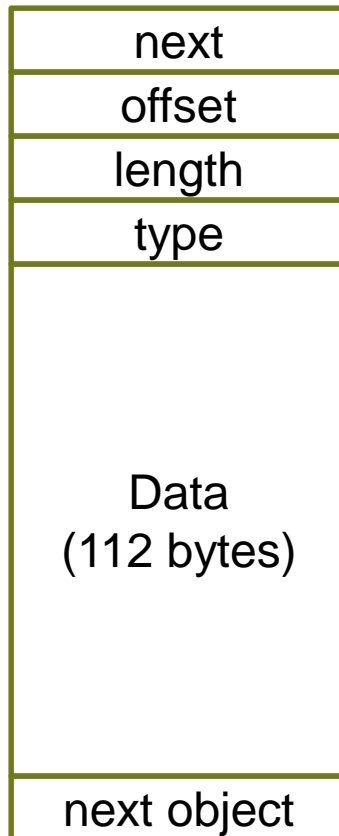
**ETH** zürich

# Memory management

# Memory management

- **Problem: how to ship packet data around**

- **Need a data structure that can:**
  - Easily add, remove headers
  - Avoid copying lots of payload
  - Uniformly refer to half-defined packets
  - Fragment large datasets into smaller units

- **Solution:**
  - Data is held in a linked list of "buffer structures"
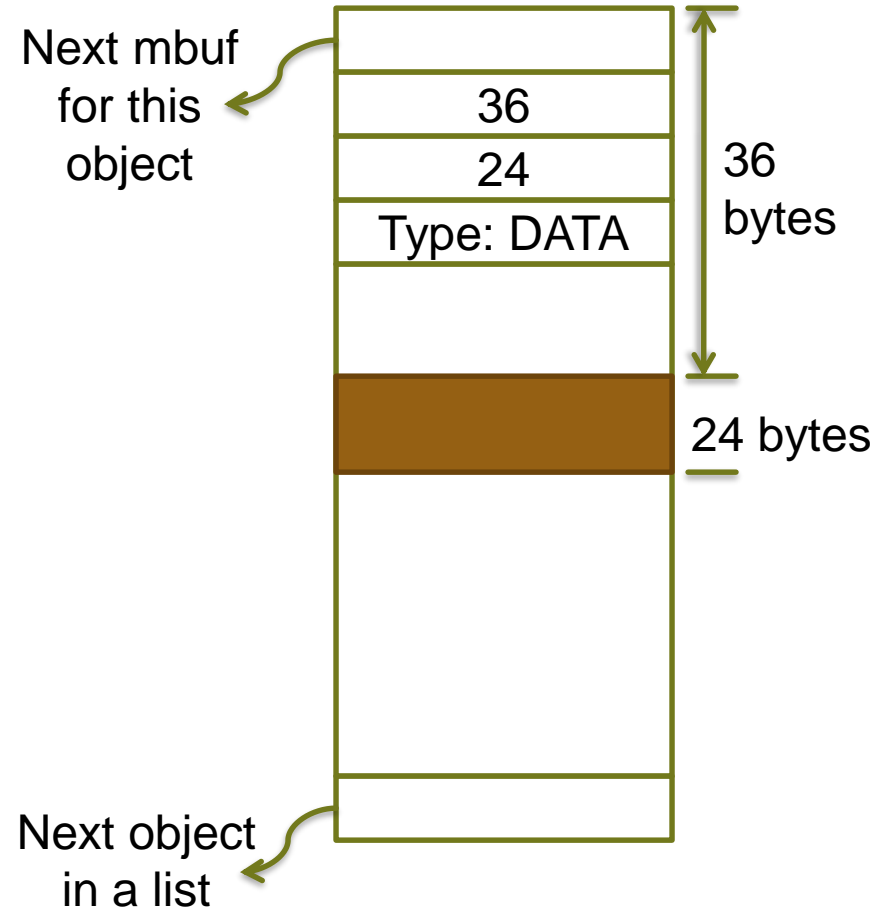
Destination:
Source:

| ETH | IP | TCP | User Data | ETH |
|-----|----|-----|-----------|-----|

# BSD Unix mbufs (Linux equivalent: sk_buffs)

| |
|---|
| next |
| offset |
| length |
| type |
| Data (112 bytes) |
| next object |

# BSD Unix mbufs (Linux equivalent: sk_buffs)

| next |
|:---:|
| offset |
| length |
| type |
| Data<br>(112 bytes) |
| next object |

Next mbuf for this object →

| |
|:---:|
| 36 |
| 24 |
| Type: DATA |
| |

36 bytes

24 bytes

Next object in a list →
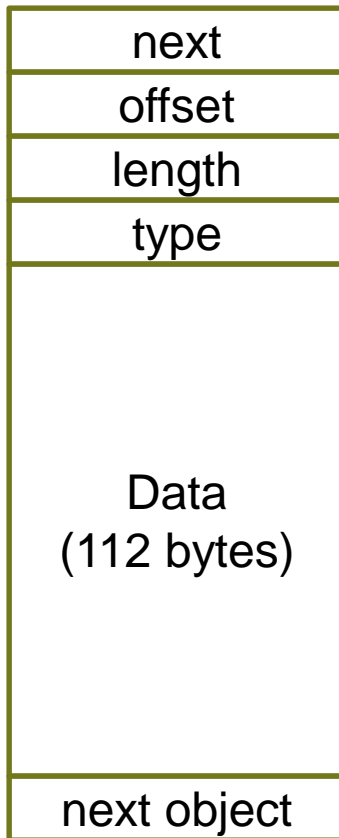
# Case Study: Linux 3.x

- **Implementing a simple protocol over Ethernet**
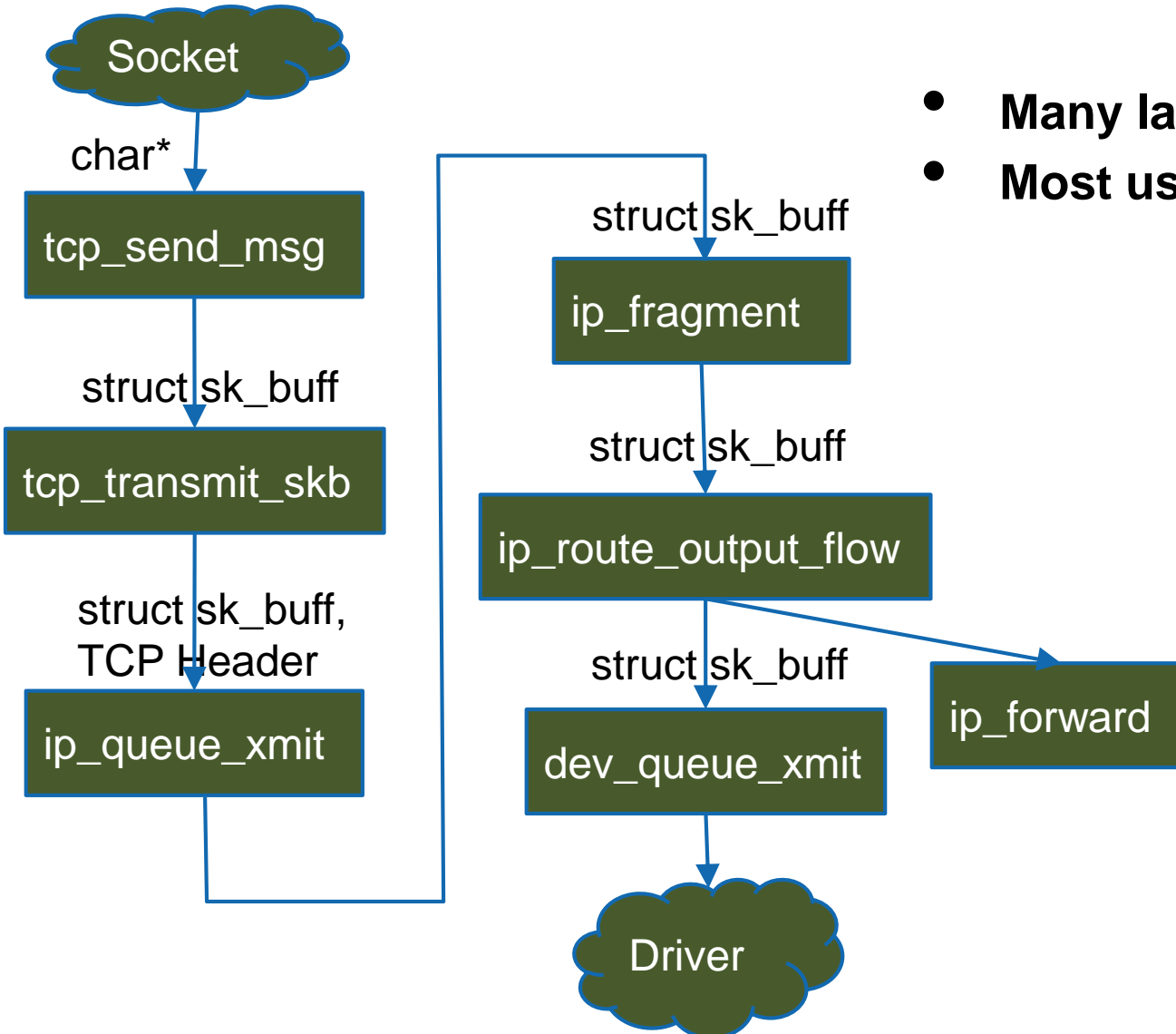- **Why?**
  - You want to play with networking equipment (well, RAW sockets are easier)
  - You want to develop specialized protocols

    *E.g., application-specific "TCP"*

    *E.g., for low-latency cluster computing*
  - You'll understand how it works!

# Sending Data in Linux 3.x

Socket

char*

tcp_send_msg

struct sk_buff

tcp_transmit_skb

struct sk_buff,
TCP Header

ip_queue_xmit

struct sk_buff

ip_fragment

struct sk_buff

ip_route_output_flow

struct sk_buff

dev_queue_xmit

ip_forward

Driver

- **Many layers**
- **Most use the sk_buf struct**

# Receiving Data: register receive hooks

- **Fill packet_type struct:**
    - .type = your ethertype
    - .func = your receive function
- **Receive handler recv_hook(...)**
    - Gets sk_buff, packet_type, net_device, ...
    - Called for each incoming frame
    - Reads skb->data field and processes protocols

Receive hook table:

| | |
|---|---|
| 0x0800 | IPv4 hdlr. |
| 0x8864 | PPPOE hdlr. |
| 0x8915 | RoCE hdlr. |

…

# Interaction with applications

- **Socket Interface**
    - Need to implement handlers for connect(), bind(), listen(), etc.

- **Call sock_register(struct net_proto_family*)**
    - Register a protocol family
    - Enables user to create socket of this type

# Anatomy of struct sk_buff

- **Called "skb" in Linux jargon**
  - Allocate via alloc_skb() (or dev_alloc_skb() if in driver)
  - Free with kfree_skb() (dev_kfree_skb())
  - Use pskb_may_pull(skb, len) to check if data is available
  - skb_pull(skb, len) to advance the data pointer
    
    *... it even has a webpage! http://www.skbuff.net/*

# SKB fields

- **Double-linked list, each skb has .next/.prev**
  - .data contains payload (size of data field is set by skb_alloc)
  - .sk is the socket this skb is owned by
  - .mac_header, .network_header, .transport_header contain headers of various layers
  - .dev is the device this skb uses
  - ... 58 member fields total

# Case study: IP fragmenting

- **Linux <2.0.32:**
  - Two fragments:
  - #1
    *Offset: 0*
    *Length: 100*
  - #2
    *Offset 100*
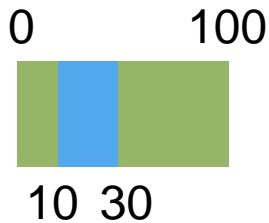    *Length: 100*

0        100        200
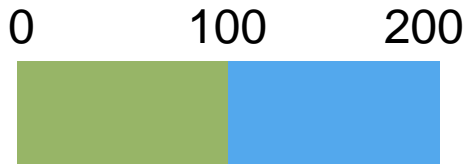
```
// Determine the position of this fragment.
end = offset + iph->tot_len - ihl;          100  100
// Check for overlap with preceding fragment, and, if needed,
// align things so that any overlaps are eliminated.
if (prev != NULL && offset < prev->end) {
  i = prev->end - offset;
  offset += i;    /* ptr into datagram */
  ptr += i;       /* ptr into fragment data */
}
// initialize segment structure
fp->offset = offset;                   0   100
fp->end = end;                        100   200
fp->len = end - offset;               100   100
.... // collect multiple such fragments in queue
// process each fragment
if(count+fp->len > skb->len)  {
  error_to_big;
}
memcpy((ptr + fp->offset), fp->ptr, fp->len);
count += fp->len;
fp = fp->next;
```

# Case study: IP fragmenting

- **Linux <2.0.32:**
  - Two fragments:
  - #1
    *Offset: 0*
    *Length: 100*
  - #2
    *Offset 10*
    *Length: 20*

```
// Determine the position of this fragment.
end = offset + iph->tot_len - ihl;        [100][30]
// Check for overlap with preceding fragment, and, if needed,
// align things so that any overlaps are eliminated.
if (prev != NULL && offset < prev->end) {
  i = prev->end - offset;                           [100-10=90]
  offset += i;    /* ptr into datagram */
  ptr += i;       /* ptr into fragment data */     [100]
}
// initialize segment structure                    [0][100]
fp->offset = offset;
fp->end = end;                                      [100][30]
fp->len = end - offset;                             [100][-70]
.... // collect multiple such fragments in queue
// process each fragment                            (size_t)-70 = 4294967226
if(count+fp->len > skb->len)  {
  error_to_big;
}
memcpy((ptr + fp->offset), fp->ptr, fp->len);
count += fp->len;
fp = fp->next;
```

0    100

10  30

# Case study: IP fragmenting

# 2.0.32 … that's so last century!

Security TechCenter > Security Bulletins > Microsoft Security Bulletin MS09-050

# Microsoft Security Bulletin MS09-050 - Critical

## Vulnerabilities in SMBv2 Could Allow Remote Code Execution (975517)

Published: Tuesday, October 13, 2009 | Updated: Wednesday, October 14, 2009

**Version:** 1.1

### General Information

#### Executive Summary

This security update resolves one publicly disclosed and two privately reported vulnerabilities in Server Message Block Version 2 (SMBv2). The most severe of the vulnerabilities could allow remote code execution if an attacker sent a specially crafted SMB packet to a computer running the Server service. Firewall best practices and standard default firewall configurations can help protect networks from attacks that originate from outside the enterprise perimeter. Best practices recommend that systems that are connected to the Internet have a minimal number of ports exposed.

This security update is rated Critical for supported editions of Windows Vista and Windows Server 2008. For more information, see the subsection, **Affected and Non-Affected Software**, in this section.

The security update addresses the vulnerabilities by correctly validating the fields inside the SMBv2 packets, correcting the way that SMB handles the command value in SMB packets, and correcting the way that SMB parses specially crafted SMB packets. For more information about the vulnerabilities, see the Frequently Asked Questions (FAQ) subsection for the specific vulnerability entry under the next section, **Vulnerability Information**.

This security update also addresses the vulnerability first described in Microsoft Security Advisory 975497.
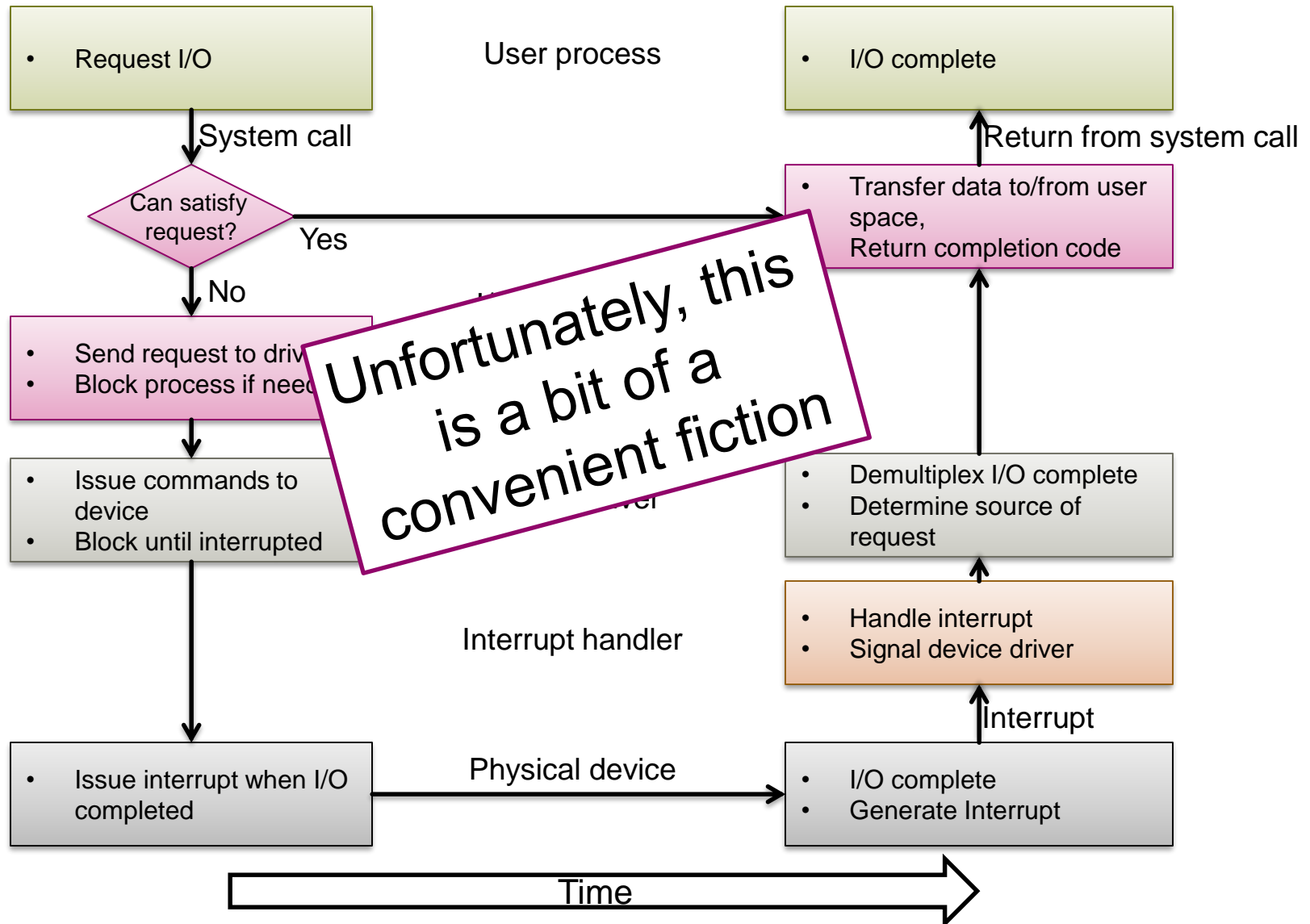
**Recommendation.** The majority of customers have automatic updating enabled and will not need to take any action because this security update will be downloaded and installed automatically. Customers who have not enabled automatic updating need to check for updates and install this update manually. For information about specific configuration options in automatic updating, see Microsoft Knowledge Base Article 294871.

For administrators and enterprise installations, or end users who want to install this security update manually, Microsoft recommends that customers apply the update immediately using update management software, or by checking for updates using the Microsoft Update service.

See also the section, **Detection and Deployment Tools and Guidance**, later in this bulletin.

# Performance issues

# Life cycle of an I/O request (each packet!)



**User process**

- Request I/O

System call

- I/O complete

Return from system call

Can satisfy request?

Yes

- Transfer data to/from user space,
- Return completion code

No

- Send request to driver
- Block process if needed

- Issue commands to device
- Block until interrupted

- Demultiplex I/O complete
- Determine source of request

**Interrupt handler**

- Handle interrupt
- Signal device driver

Interrupt

- Issue interrupt when I/O completed

**Physical device**

- I/O complete
- Generate Interrupt

Time

*Unfortunately, this is a bit of a convenient fiction*

# Consider 10 Gb/s Ethernet

# At full line rate for 1 x 10 Gb port

- **~1GB (gigabyte) per second**
  - ⇒ ~ 700k full-size Ethernet frames per second
  - ⇒ At 2GHz, must process a packet in ≤ 3000 cycles

- **This includes:**
  - IP and TCP checksums
  - TCP window calculations and flow control
  - Copying packet to user space

# A few numbers…

- **L3 cache miss (64-byte lines) ≈ 300 cycles**

    $\Rightarrow$ *At most* 10 cache misses per packet
    Note: DMA ensures cache is cold for the packet!

- **Interrupt latency ≈ 500 cycles**

    - Kernel entry/exit

    - Hardware access

    - Context switch / DPC

    - Etc.

# Plus…

- **You also have to send packets.**
  - Card is full duplex $\Rightarrow$ can send at 10 Gb/s
- **You have to do something useful with the packets!**
  - Can an application make use of 1.5kB of data every 1000 machine cycles or so?
- **This card has two 10 Gb/s ports.**

# And Plus …

- **And if you thought that was fast …**
  - Mellanox FDR 200 Gb/s Adapter
  - Impossible to use without advanced features
    *RDMA*
    *SR-IOV*
    *TOE*
    *Interrupt coalescing*

# What to do?

- **TCP offload (TOE)**
  - Put TCP processing into hardware on the card
- **Buffering**
  - Transfer lots of packets in a single transaction
- **Interrupt coalescing / throttling**
  - Don't interrupt on every packet
  - Don't interrupt at all if load is very high
- **Receive-side scaling**
  - Parallelize: direct interrupts and data to different cores

# Linux New API (NAPI)

- **Mitigate interrupt pressure**
  1. Each packet interrupts the CPU  vs.
  2. CPU polls driver
     - NAPI switches between the two!
- **NAPI-compliant drivers**
  - Offer a poll() function
  - Which calls back into the receive path …
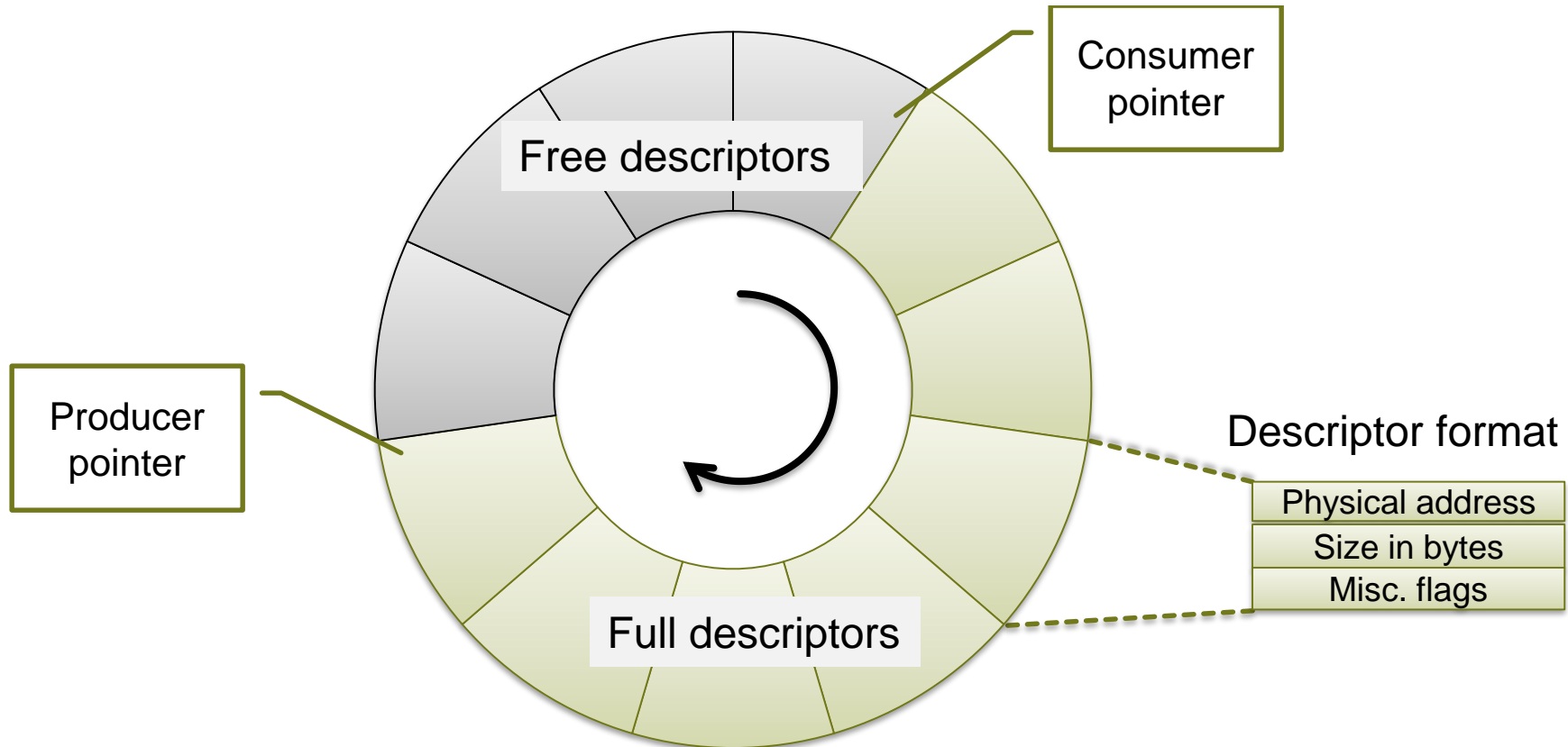
# Linux NAPI Balancing

- **Driver enables polling with netif_rx_schedule(struct net_device *dev)**
  - Disables interrupts

- **Driver deactivates polling with netif_rx_complete(struct net_device *dev)**
  - Re-enable interrupts.

- **→ but where does the data go???**

# Buffering

**Key ideas:**

- **Decouple sending and receiving**
    - Neither side should wait for the other
    - Only use interrupts to unblock host

- **Batch requests together**
    - Spread cost of transfer over several packets

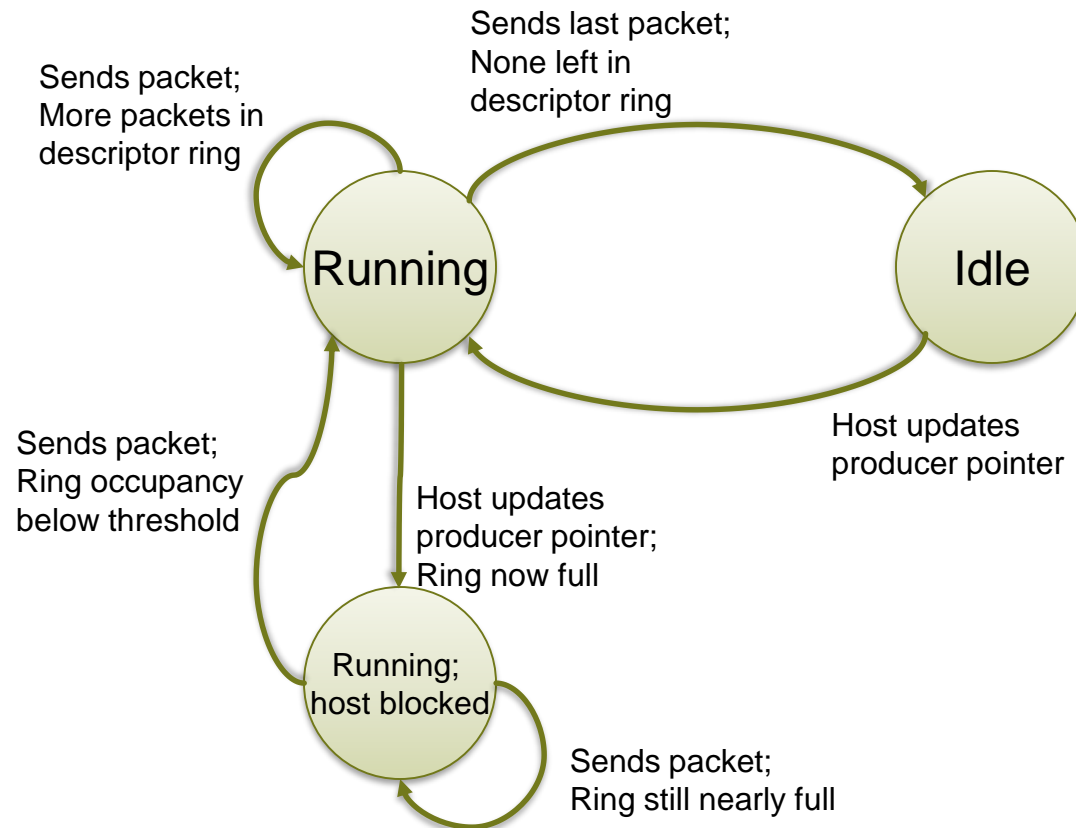# Producer-consumer buffer descriptor rings



Consumer pointer

Free descriptors

Producer pointer

Full descriptors

Descriptor format

| Physical address |
| Size in bytes |
| Misc. flags |

# Buffering for network cards

**Producer, consumer pointers are NIC registers**

- **Transmit path:**
  - Host updates producer pointer,
    adds packets to ring
  - Device updates consumer pointer

- **Receive path:**
  - Host updates consumer pointer,
    adds empty buffers to ring
  - Device updates producer pointer,
    fills buffers with received packets.

**More complex protocols are possible…**

# Example transmit state machine

Sends packet;
More packets in
descriptor ring

Sends last packet;
None left in
descriptor ring

**Running**

**Idle**

Host updates
producer pointer

Sends packet;
Ring occupancy
below threshold

Host updates
producer pointer;
Ring now full

**Running;
host blocked**

Sends packet;
Ring still nearly full

# Transmit interrupts

## Ring empty

$\Rightarrow$ all packets sent

$\Rightarrow$ device going idle

Sends packet;
More packets in
descriptor ring

Sends last packet;
None left in
descriptor ring

Running

Idle

Sends packet;
Ring occupancy
below threshold

Host updates
producer pointer

Host updates
producer pointer;
Ring now full
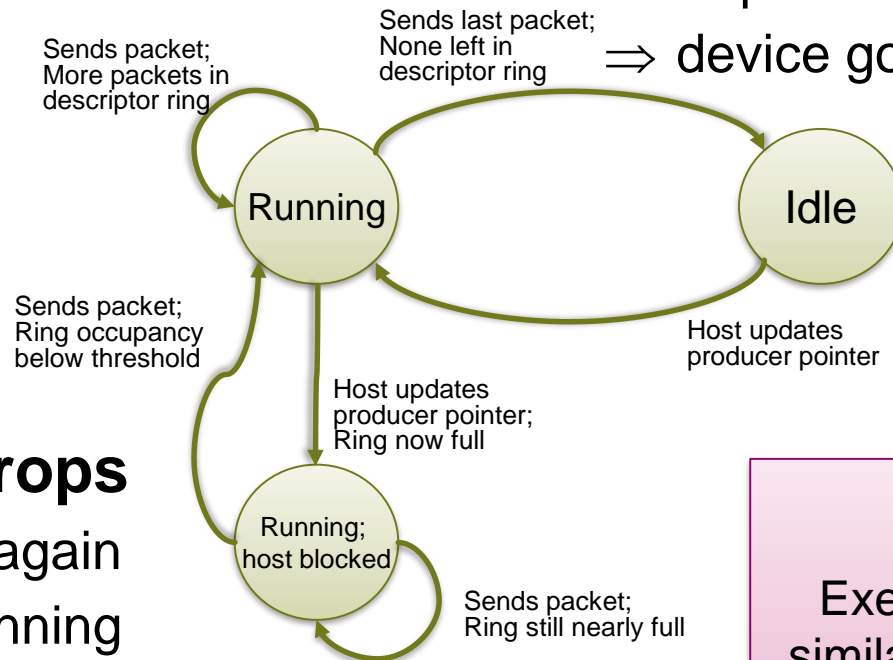
## Ring occupancy drops

$\Rightarrow$ host can now send again

$\Rightarrow$ device continues running

Running;
host blocked

Sends packet;
Ring still nearly full

Exercise: devise a
similar state machine
for receive!

# Buffering summary

- **DMA used *twice***
  - Data transfer
  - Reading and writing descriptors
- **Similar schemes used for any fast DMA device**
  - SATA/SAS interfaces (such as AHCI)
  - USB2/USB3 controllers
  - etc.
- **Descriptors send *ownership* of memory regions**
- **Flexible – many variations possible:**
  - Host can send lots of regions in advance
  - Device might allocate out of regions, send back subsets
  - Buffers might be used out-of-order
- **Particularly powerful with multiple send and receive queues…**

# Receive-side scaling

- **Insight:**
  - Too much traffic for one core to handle
  - Cores aren't getting any faster
    $\Rightarrow$ Must parallelize across cores
- **Key idea: handle different flows on different cores**
  - But: how to determine flow for each packet?
  - Can't do this on a core: same problem!
- **Solution: demultiplex on the NIC**
  - DMA packets to per-flow buffers / queues
  - Send interrupt only to core handling flow

# Receive-side scaling
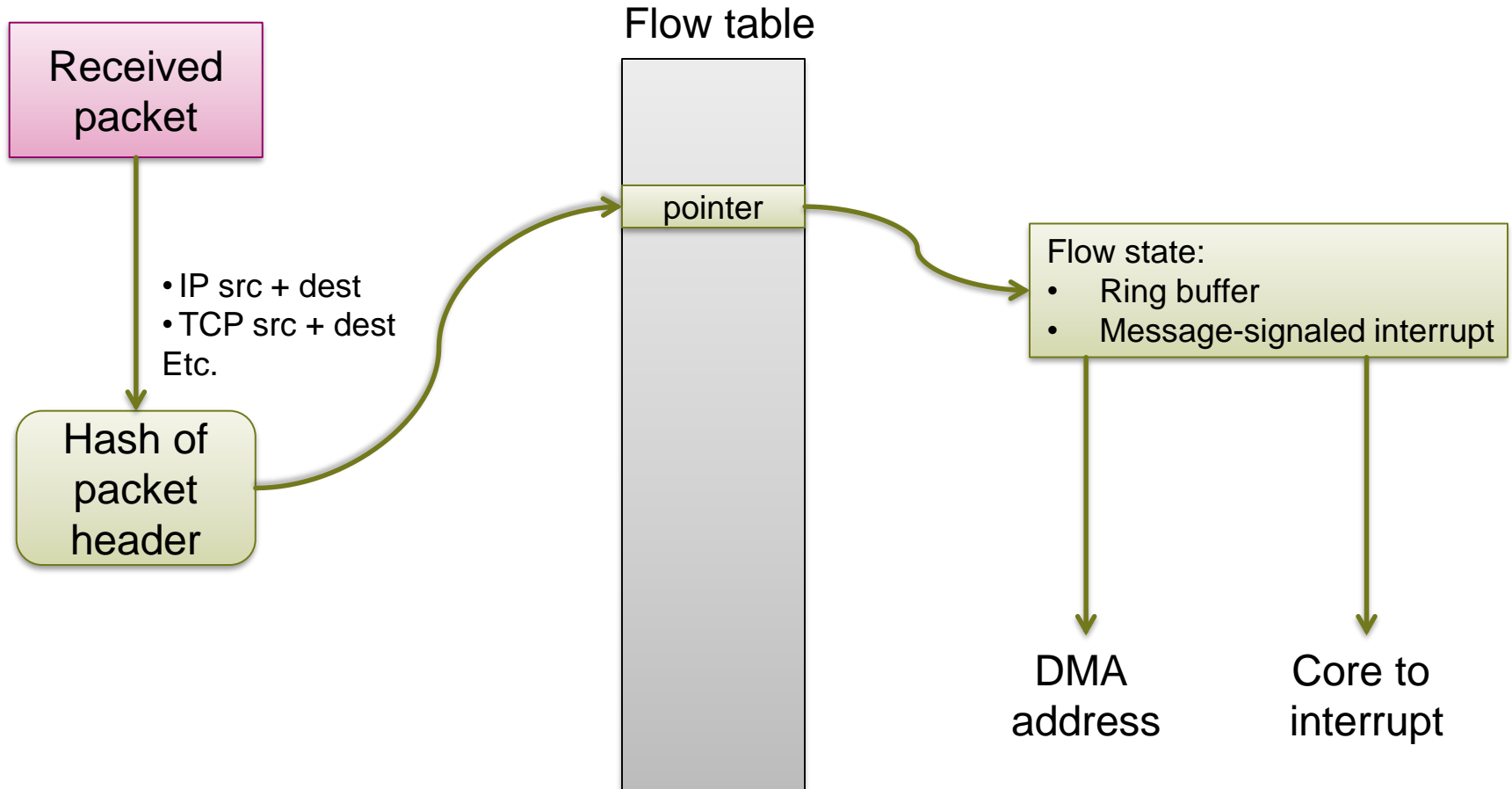
# Receive-side scaling

- **Can balance flows across cores**
  - Note: doesn't help with one big flow!
- **Assumes:**
  - *n* cores processing *m* flows is faster than one core
- **Hence:**
  - Network stack and protocol graph must *scale* on a multiprocessor.
- **Multiprocessor scaling: topic for later**

# Next (final) week

- **Virtual machines**
- **Multiprocessor operating systems**