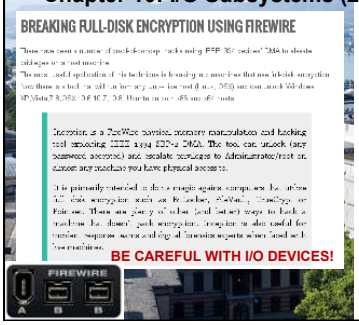
ADRIAN PERRIG & TORSTEN HOEFLER

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



Our Small Quiz

- **True or false (raise hand)**
  - Open files are part of the process' address-space
  - Unified buffer caches improve the access times
  - 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…**

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
/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

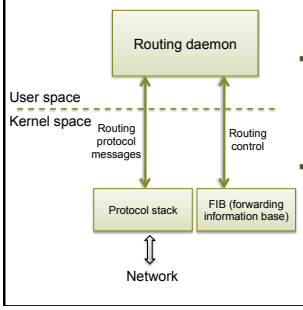
- **Physical hardware configuration readable from `/sys`**
  - Special fake file system: `sysfs`
  - Plug events delivered by a special socket
- **Drivers dynamically loaded as kernel modules**
  - Initial list given at boot time
  - User-space daemon can load more if required
- **`/dev` populated dynamically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

Interface to network I/O

Unix interface to network I/O

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

Software routing



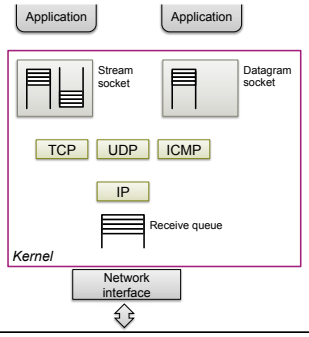
- **OS protocol stacks include routing functionality**
- **Routing *protocols* typically in a user-space daemon**
  - Non-critical
  - Easier to change
- ***Forwarding* information typically in kernel**
  - Needs to be fast
  - Integrated into protocol stack

Network stack implementation

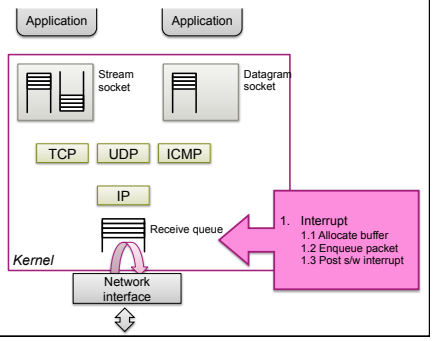
Networking stack

- **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)

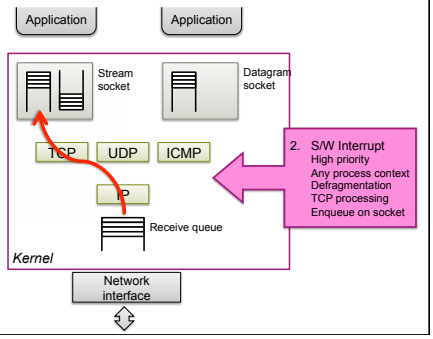
Receiving a packet in BSD



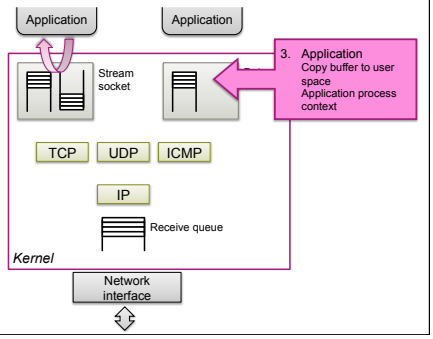
Receiving a packet in BSD



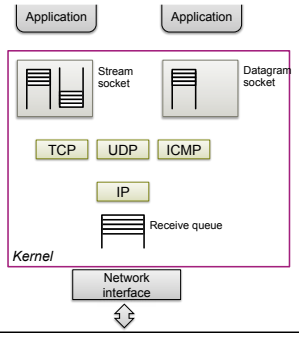
Receiving a packet in BSD



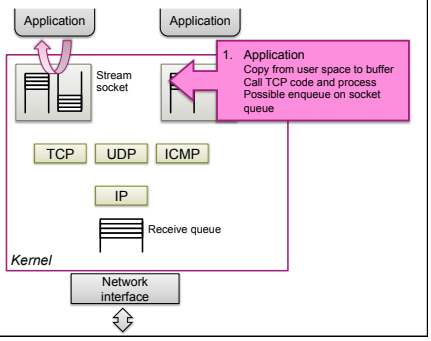
Receiving a packet in BSD



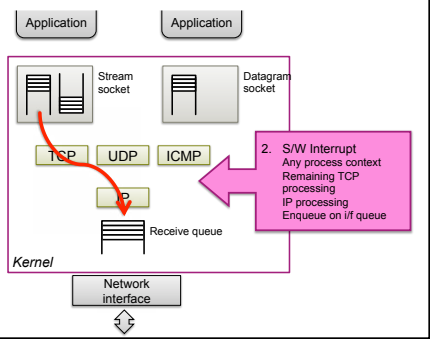
Sending a packet in BSD



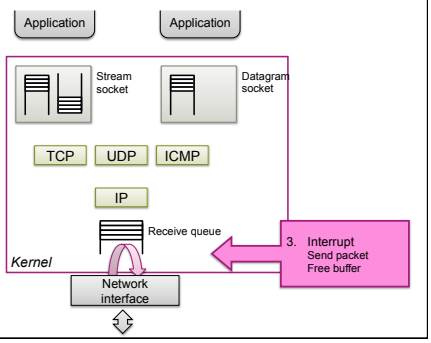
Sending a packet in BSD



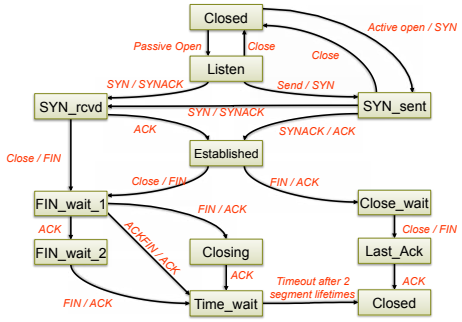
Sending a packet in BSD



Sending a packet in BSD



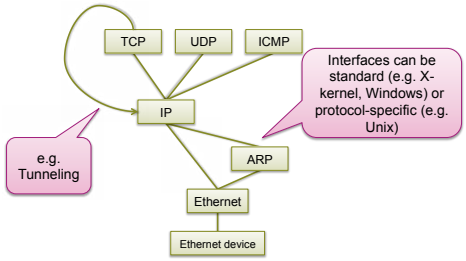
The TCP state machine



OS TCP state machine

- **More complex! Also needs to handle:**
  - Congestion control state (window, slow start, etc.)
  - Flow control window
  - 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:

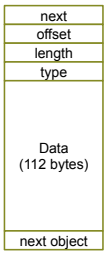
- **Per-protocol (handle all flows)**
  - Packets are "tagged" with demux tags
- **Per-connection (instantiated dynamically)**
  - Multiple *interfaces* as well as connections
  - Ethernet ↔ Ethernet ⇒ bridging
  - IP ↔ IP ⇒ IP routing

Memory management

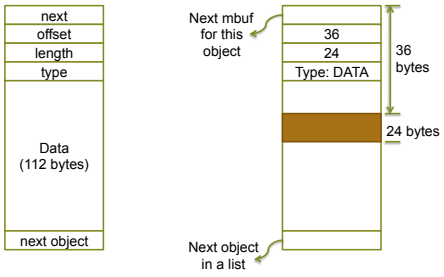
Memory management

- **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"

BSD Unix mbufs (Linux equivalent: sk_buffs)



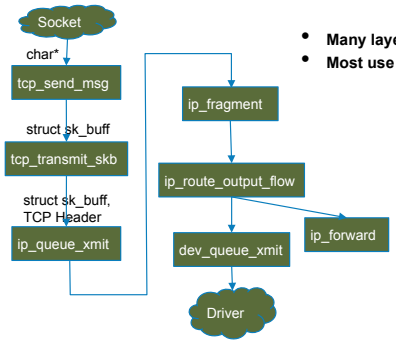
BSD Unix mbufs (Linux equivalent: sk_buffs)



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



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

Register a receive hook

- **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: TCP Fragmenting

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

```
// Determine the position of this fragment.
end = offset + iph->tot_len - ihl;        #1: 100, #2: 200
// 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;        #1: 0, #2: 100
fp->end = end;              #1: 100, #2: 200
fp->len = end - offset;     #1: 100, #2: 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;
```

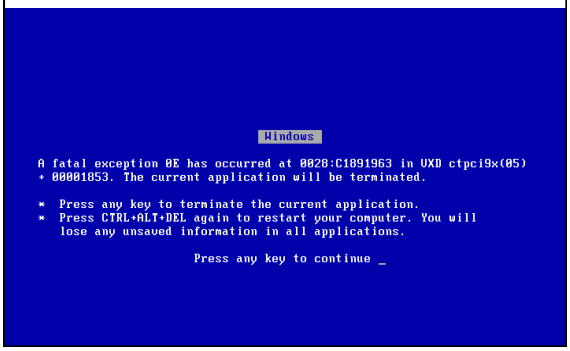
Case Study: TCP Fragmenting

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

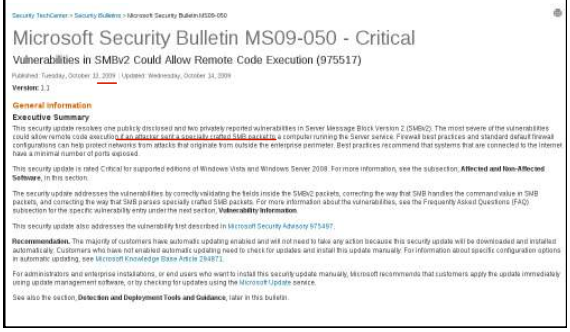
```
// Determine the position of this fragment.
end = offset + iph->tot_len - ihl;        #1: 100, #2: 30
// Check for overlap with preceding fragment, and, if needed,
// align things so that any overlaps are eliminated.
if (prev != NULL && offset < prev->end) {
  i = prev->end - offset;                 #2: 100-10=90
  offset += i;    /* ptr into datagram */ #2: 100
  ptr += i;       /* ptr into fragment data */
}
// initialize segment structure
fp->offset = offset;        #1: 0, #2: 100
fp->end = end;              #1: 100, #2: 30
fp->len = end - offset;     #1: 100, #2: -70
.... // 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;
```

(size_t)-70 = 4294967226

Case Study: TCP Fragmenting

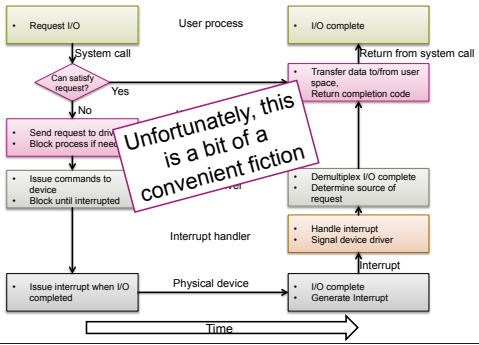


2.0.32 … that's so last century!



Performance issues

Life Cycle of an I/O Request



Consider 10 Gb/s Ethernet

At full line rate for 1 x 10Gb 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**
  ⇒ *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 ⇒ can send at 10Gb/s
- **You have to do something useful with the packets!**
  - Can an application can make use of 1.5kB of data every 1000 machine cycles or so?
- **This card has two 10Gb/s ports.**

And Plus …

- **And if you thought that was fast …**
  - Mellanox 100 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
  2. Vs. 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

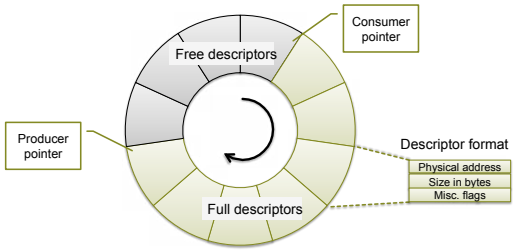
- **Driver enables polling with netif_rx_schedule(struct net_device *dev)**
  - Disables interrupts
- **Driver deactivates polling with netif_rx_complete(struct net_device *dev)**
  - Re-enable interrupts.
- **→ but where does the data go???**

Buffering

Key ideas:

- **Decouple sending and receiving**
  - Neither side should wait for the other
  - Only use interrupts to unblock host
- **Batch together requests**
  - Spread cost of transfer over several packets

Producer-consumer buffer descriptor rings



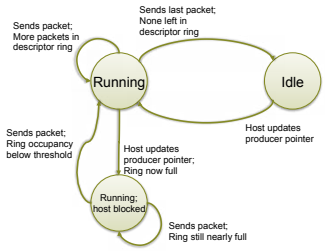
Buffering for network cards

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.

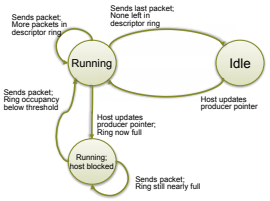
More complex protocols are possible…

Example transmit state machine



Transmit interrupts

- **Ring empty**
  - ⇒ all packets sent
  - ⇒ device going idle
- **Ring occupancy drops**
  - ⇒ host can now send again
  - ⇒ device continues running



Exercise: devise a similar state machine for receive!

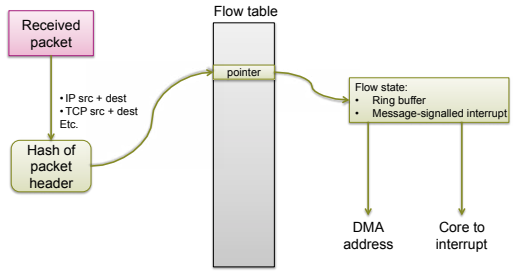
Buffering summary

- **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
    - ⇒ 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**

Tomorrow

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