Linux Kernel Network Subsystem

Chih-che Lin

04/10/2006

Course Material Information

- http://www.csie.nctu.edu.tw/~jclin/nsd/
- Mini Project Deadline
- Final Examination
- Final Project
  - Implementing a packet filtering mechanism

Outline

- General Concept
  - Operating System
  - Device
  - Networking
- Linux Implementation
  - Socket Layer
  - Transport Layer
  - Network Layer
  - Device Layer

General Concept

Operating System Kernel

- Role
  - Resource manager
  - service provider
- Monolithic program vs. micro-kernel approach
- Dynamic loadable kernel module
- Component
  - Machine-independent
  - Machine-dependent
- System Entry

Machine-independent Part (80%)

- Basic kernel facility
  - Timer
  - Clock handling
  - Process management
  - Descriptor (file, socket, I/O, etc.) management
- Memory-management support
  - Paging
  - Swapping
- Generic system interface on descriptors
  - I/O (read/write)
  - Control (ioctl)
  - Multiplexing (select)
- File system
  - Files
  - Directories

Machine-independent Part .1

- Terminal Handling
  - Terminal interface driver (baud rate, cooked vs. raw mode, etc.)
- Interprocess communication
  - Socket
  - Pipe
  - FIFO
- Network communication
  - TCP/IP Protocol
  - Routing
- Accounting
  - CPU and I/O usage

Machine-dependent (20%)

- Low-level system startup actions
- Trap and fault handling
  - Specific processor instructions
  - Specific Interrupt mechanism
- Low-level manipulation of the run-time context of a process
  - Specific register set layout
- Configuration and initialization of hardware devices
- Run-Time support for I/O device
(open/read/write/close or send/receive)

User interface

- system call
  - A system call is identified by system call number
  - Various specific system calls
    - sys_read(), sys_write(), etc.
  - Generic system call:
    - Ioctl()
  - Socket interface
    - Netlink socket (provided by Linux 2.6)
- The global errno variable stores the status of the system call

System Entry

Hardware interrupt

- Example: from a network interface card
- Occurs asynchronously
- May not relate to the context of currently running process
- Have to do a context switching
- Stack used
  - The kernel stack of some running process, or
  - A system-wide interrupt stack
- Nested interrupt is allowed in Linux.
- In 4BSD, can not sleep (where to save the context?) , must run to completion

Hardware Exception And Trap

- Example: divide by zero
- May occur synchronously or asynchronously
- Related to the current executing process

System call

- May be blocked to wait for an event (e.g., disk I/O)
- May be interrupted by a signal when blocking
  - Can be aborted and return an error, or
  - Can be restarted
- Return from a system call
  - Check for a posted signal and execute its signal handler
  - Check for a higher priority process
    - Do a context switch if there is one
    - Later, return from the system call to the user process

Software interrupt

- Called as softirq in Linux
- Used by the system to force the scheduling of an event as soon as possible when the priority is lowered
- A high priority hardware interrupt creates a work of queue to be done at a lower priority level
- Triggered when the priority is dropped below the lower priority
- Can be implemented as
  - Hardware-generated interrupt or
  - A flag checked whenever the priority level drops
- Case study: The BSD and Linux network subsystem design
  - Advantage:
    - Absorb a burst of packets without losing any
  - Disadvantage:
    - At sustained high load, effective throughput drops considerably (receive livelock problem)

Clock interrupt

- Hardclock
  - Triggered 100 or 1000 times per second
  - The HZ variable controlling the frequency (100 or 1000)
  - The jiffies variable storing the ticks since system booted. (In Linux)
  - A kind of Hardware interrupt
  - Performed at a high priority
  - should be simple and fast
- Softclock
  - Issued when necessary
  - Implemented as software interrupt
  - Performed at a lower priority when returning from a hard clock
  - Can be longer and more complex
  - Used to provide timer functionality to user-level processes and other kernel components
  - Example
    - TCP Timeout

Run-Time Organization

- Top half
- Bottom half

Top Half

- Provide services to processes in response to system calls
- Can be thought of as a library of routines shared by all processes
- Execute in privileged execution mode
- Run on per-process kernel stack
- Has access to both kernel data structure and the context of user-level processes

Top Half .1

- Never preempted (single-threaded kernel)
- May block and give up the processor voluntarily if must wait for an event
- Multiple processes can thus be in the kernel simultaneously (one running, all others sleep).
- Execution may be interrupted by interrupts from the bottom half.
- Can block interrupts from the bottom half by raising the processor priority to ensure the consistency of the work queues and other data structures shared between the top and bottom halves
  - work for only a single-processor system, does not work well for a multi-processor system)
  - The protection of shared data in a multi-processor system is accomplished by coherence protocols.

Bottom Half

- Contain routines invoked to handle hardware interrupts
- Run on kernel stack in kernel address space
- Can not be blocked
- Activities occurs asynchronously, software can not depend on having a process running when an interrupt occurs
- When there is no running process, the system is idle and the kernel is executing an idle busy loop.
- The state information of the process that initiated the activity is not available

Communications Between The Half and Bottom Halves

- Work queue
- Managed by software interrupt
- Example:
  - A user process issues a READ system call.
  - The top half initializes an I/O operation and put the request descriptor to the appropriate work queue.
  - The requesting process sleeps. (The issued system is blocked)
  - A hardware interrupt occurs reporting the completion of the READ operation.
  - wake up the requesting process. (The system call resume its execution and is going to return.)
  - The requesting process resumes execution.

Shared Data Protection – Synchorization Primitives

- Per-CPU Variable
- Atomic operation (need support of Processor instruction)
- Memory barrier
- Spin lock
- Semaphore
- Local interrupt disabling
- Local software interrupt disabling

Read-copy-update (RCU)

- Get the pointer to the data structure that is to be read or written.
- Call Rcu_read_lock() equivalent to Preempt_disable()
- Rcu_read_unlock() equivalent to Preempt_enable()
- Read operation is fast
- Write operation:
  - Take a copy of the whole data structure first
  - Modify the copy of the data
  - Change the pointer of the old copy to the new one after finishing modification.
- A reader thread is not allowed to sleep until it calls rcu_read_unlock().
- RCU is introduced in Linux 2.6 and used in networking subsystem.

UNIX Kernel Properties

- Re-entrant
  - Several processes may be in the kernel at the same time.
  - Only one can be running, all others must be blocked.
- Non-preemptive
  - A system call has to run to completion, or block voluntarily, but not be preempted in the middle.
  - Can ensure data consistency
  - Time quantum may expire (not good for real-time applications)
  - For data consistency, still need to worry about three things
    - Blocking operation
      - Use locks to ensure data consistency across blocked system calls
    - Interrupts
      - Raise priority level to block interrupts temporarily
    - Multiprocessor
      - Two processes may execute in the kernel mode on different processors
      - Must use locks everywhere and anytime
        - resolved by coherence protocol
        - Performance may degrade substantially

Device Driver

Driver Task

- Used to control a device
- Know the details of a device
- Provide a uniform interface to the kernel and users to control the device

Three Main Components

- Auto-configuration and initialization routines
- Routine for servicing I/O requests (the top half)
- Interrupt service routine (ISR) (the bottom half)

Auto-configuration and Initialization Routines

- Probing the existence of a controller or a device

- If found, attach the controller/device by:
  - Allocating resources for it
  - Initialize the data structures

Routine for servicing I/O Requests -- The Top Half

- Invoked by system calls (e.g., read and write) or the virtual memory system
- Execute synchronously in the calling process's context
- Is permitted to block by calling sleep()

Main entry points -- Character Device

- **Open()**
  - **Make sure that the device is found during the auto-configuration phase**
  - **Make sure the media is present right now (for removable media such as floppy disk, CD-ROM, etc.)**
  - **May reset or bring the device on-line (power saving mode)**
  - **If the device can not be shared by multiple users, set an exclusive flag (e.g., a printer)**
- **Read()**
  - **Read data from the device**
- **Write()**
  - **Write data to the device**
- **Ioctl()**
  - **Do an operation other than read or write**
  - **E.g., get or set a device's control parameters**

Main entry points -- Character Device .1

- Poll() (in BSD: Select())
  - Check the device to see whether there are data available for reading, or space available for writing (e.g., socket-related system calls such as sendto() and recvfrom()).
  - Used by the select, poll, epoll system call
  - Meaningless for a raw device because it is always true (e.g., a hard disk)
- Release() (in BSD: Close())
  - Called after the final users interested in the device terminates
  - The exclusive flag should be cleared.

Main entry points -- Block Device

- Media_changed()
  - Checking whether the removable media has been changed. (e.g. a floopy disk)
- Revalidate_disk():
  - Checking whether the block device holds valid data().

Strategy() Routine (FreeBSD Version)

- Called by the file system bread() or bwrite(), not directly called by the users
- Do the work in “I/O request queue”
- So named because requests in the queue may be scheduled strategically to improve the I/O performance
- Disksort() uses two queues to implement the elevator scheduling algorithm

Strategy() Routine (Linux Version)

- Work on a request_queue structure and a bio discriptor.
- Invoked by the request_fn function pointer in the request_queue structure.
- The initialization of a request_queue is accomplished by blk_init_queue().
- The strategy() routine performs and optimizes IO operations based on the feature of the device.
  - e.g.
    - scsi_request_fn() for scsi devices
    - Scatter-gather DMA mode

IO Scheduler in Linux

- The optimization for read/write requests to a disk is performed by IO Scheduler (elevator).
- Four types of elevators in Linux
  - No operation (NOOP)
  - Complete Fairness Queuing (CFQ)
  - Deadline
  - Anticipatory (AS)
- The functions of an elevator is invoked via the make_request_fn function pointer in the request_queue structure.

Networking

Socket Interface

- A socket is an endpoint of a communication channel.

- Properties of a communication channel:
  - In-order delivery of data?
  - Unduplicated delivery of data?
  - Reliable delivery of data?
  - Preserve message boundary?
  - Support for out-of-band data?
  - Connection-oriented or connectionless?

Socket System Call

- Socket (int domain, int type, int protocol)
- Used to create a socket
- Commonly used domains:
  - AF_UNIX
  - AF_INET
  - AF_ROUTE (in FreeBSD)
  - AF_NETLINK (in Linux 2.6)
- Netlink is a new interface for communication between user-level program and kernel.

Commonly Used Types:

- Datagram
  - Unreliable, connectionless, preserve message boundary
- Stream
  - Reliable, connection-based, in-sequence delivery, byte stream
- Sequenced packet
  - Reliable, unduplicated, sequenced, preserve message boundary
  - This type is seldom used.
    - User applications prefer to achieve the service on top of UDP on their own.
- Raw socket
  - Used by some privileged programs such as “ping” and “route”.
  - Route
    - o s=socket(AF_ROUTE, SOCK_RAW, 0)
    - o Used to manipulate the routing table
  - Ping
    - o s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    - o Can build (forge!) your own IP header for a packet.

Protocol Field

- Valid protocol identifiers are defined in the /etc/protocols file.
- If protocol is specified as 0, the system will choose an appropriate one.
  - Datagram: UDP
  - Stream: TCP

Bind System Call

- Bind (int socket, const struct sockaddr *address, size_t address_len)
- Give a name (address) to a socket
- The address includes:
  - An IP address such as 140.113.17.1.
  - A port number such as 80.
- Usually needed only for a server.
  - A server need not specify an IP address number when binding its socket.
    - The system will accept all arriving packets for the server.
    - The server may not know the IP address(es) of the host.
    - If a host has multiple IP addresses and the server binds its socket to a particular IP address, the system will accept only those packets with the particular IP address as their destination addresses.
  - A server must choose a port to bind so that the system can deliver a client packets to it correctly.

Bind System Call .1

- Usually not needed for a client
  - A client need not call bind() at all.
    - A client need not know its own IP address and port number.
      - One host IP address and a random port number are chosen for the client socket.
      - The server uses these address information in a client's packet to send back its reply.
    - Of course, a client must know the IP address and port number of the server.

Listen System Call

- listen(int s, int backlog)
- Sys_listen() is finally handled by tcp_listen_start()
  - Express the willingness to start accepting incoming connection requests.
    - Used by a server
  - A listening socket has two queues:
    - Partially-established request queue
      - Hold the requests whose TCP 3-way handshaking have not finished yet.
    - Fully-established request queue
      - Hold the requests whose TCP 3-way handshaking have finished.

Listen System Call .1

- A request will first stay in the partially-established queue and then be moved to the fully-established queue when it becomes ready.
- Backlog specifies the maximum number of complete requests in the fully-established queue.
  - Why setting an upper bound?
    - Avoid two kinds of attack:
- An attacker can keep sending connection requests to exhaust the system's memory.
- Also, a connection lookup operation will become slower and slower when the queue length becomes larger and larger.
    - Why not setting backlog to a small number such as 1?
      - When a burst of connection requests come in, many of them will be unnecessarily dropped

Listen System Call .2

- The partially-established queue is the source of TCP denial-of-service attack
  - If there is an upper bound on the partially-established queue:
    - TCP connection setup needs 3-way handshaking.
    - The attacker can issue numerous connection requests. For each connection request, the attacker just sends the first SYN packet, then stop sending the other 2 packets.
    - The partially-established queue will be full of incomplete TCP connection requests.
    - When the number of partially-established requests exceeds the upper bound, no more TCP connection will be accepted.
    - All incoming requests are dropped, including those initiated by good users.

Listen System Call .3

- If there is no upper bound on the partially-established queue:
  - The number of requests in the partially-established queue will keep growing.
    - Eventually all of the system's memory will be exhausted.

- One approach to mitigate the problem:
  - Shorten the TCP 3-way handshaking timeout from 30 seconds to a shorter time such as 3 second
    - Reduce the number of requests in the partially-established queue from 30 (second) * attack_rate (SYN/second) to 3 * attack_rate.
    - Problem solved?
    - No! Why does not an attacker use a higher SYN rate?
  - So, let's further reduce the timeout from 3 to 1 second, or even 0.1 second.
    - The same problem is still there.
    - Worse yet! All normal wide-area TCP connection requests will be rejected because their RTT is larger than 0.1 second.

Listen System Call .4

- Linux 2.6 uses a hash table to store the SYN requests
  - fast searching
  - Alleviating the search overhead when network load is high
- The accept queue is a linked list
  - a fully-established queue need not be searched quickly.

Accept System Call

- accept(int s, struct sockaddr *addr, int *addrlen)
  - Take the first complete connection request from the request queue and return a new socket.
  - The listening socket is still used for accepting future incoming requests.
  - The address of the client that made the connection request is also returned.
    - The server then has a chance to decide whether to serve the client's request or not.

Connect System Call

- connect(int socket, const struct sockaddr *address, size_t address_len)
  - Used by a client to initiate a point-to-point connection to a server.
  - Even if a socket is a datagram socket, it can still use connect() to fix the server's address.
    - Future sendto() calls then do not need to provide the server address any more.

Sendto and Send System Calls

- sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, int tolen)
  - Send a packet to the specified receiver.
  - The socket is a datagram (connectionless) socket.

- send(int s, const void *msg, size_t len, int flags)
  - Send some number of bytes to the receiver
  - The peer address has been bound in connect().

Recvfrom and Recv System Calls

- recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, int *fromlen)
  - Receive a datagram packet. The packet sender's address is returned.
  - fromlen must be initialized to the size of struct sockaddr, otherwise "from" will not be filled in with the sender's address correctly!
- recv(int s, void *buf, size_t len, int flags)
  - Receive some number of bytes

Manipulate Socket Options

- getsockopt(int s, int level, int optname, void *optval, int *optlen)
  - Check the value of the socket option
  - Level can be TCP, IP, interface, etc.

- setsockopt(int s, int level, int optname, const void *optval, int optlen)
  - Set the value of a socket option.
  - Example:
    - Set the size of a socket's receive buffer
    - setsockopt(s, SOL_SOCKET, SO_RCVBUF, &bufsize, sizeof(bufsize))

A Socket Has a Send and a Receive Buffer

- Send buffer:
  - If the network interface is slow in sending data and the application program sends data too fast, data will be queued in the send buffer.
  - When the send buffer is full, the application program cannot send any more data to the socket.
  - The send buffer is only used for a stream (TCP) socket.
  - The send buffer is not used for a datagram (UDP) socket. UDP packets go all the way to the network interface. They may be dropped at the network interface.

A Socket Has a Send and a Receive Buffer .1

- Receive buffer
  - If the network interface is fast in receiving data and the application program receives data too slowly, data will be queued in the receive buffer.
  - When the receive buffer is full, TCP data will no longer come in. However, UDP data may keep coming in and get lost.

- A TCP connection's maximum possible throughput is limited by the minimum of the sender's socket send buffer size and the receiver's socket receive buffer size.
  - The socket receive buffer size physically limits a TCP connection's maximum window size.
    - Do not want to overflow the receiver's receive buffer
  - The socket send buffer size effectively limits a TCP connection's maximum number of outstanding bytes per its RTT.
    - Do not want to overflow the sender's send buffer.
  - Although the two reasons are different, their effects are the same!

Network Flow Overview

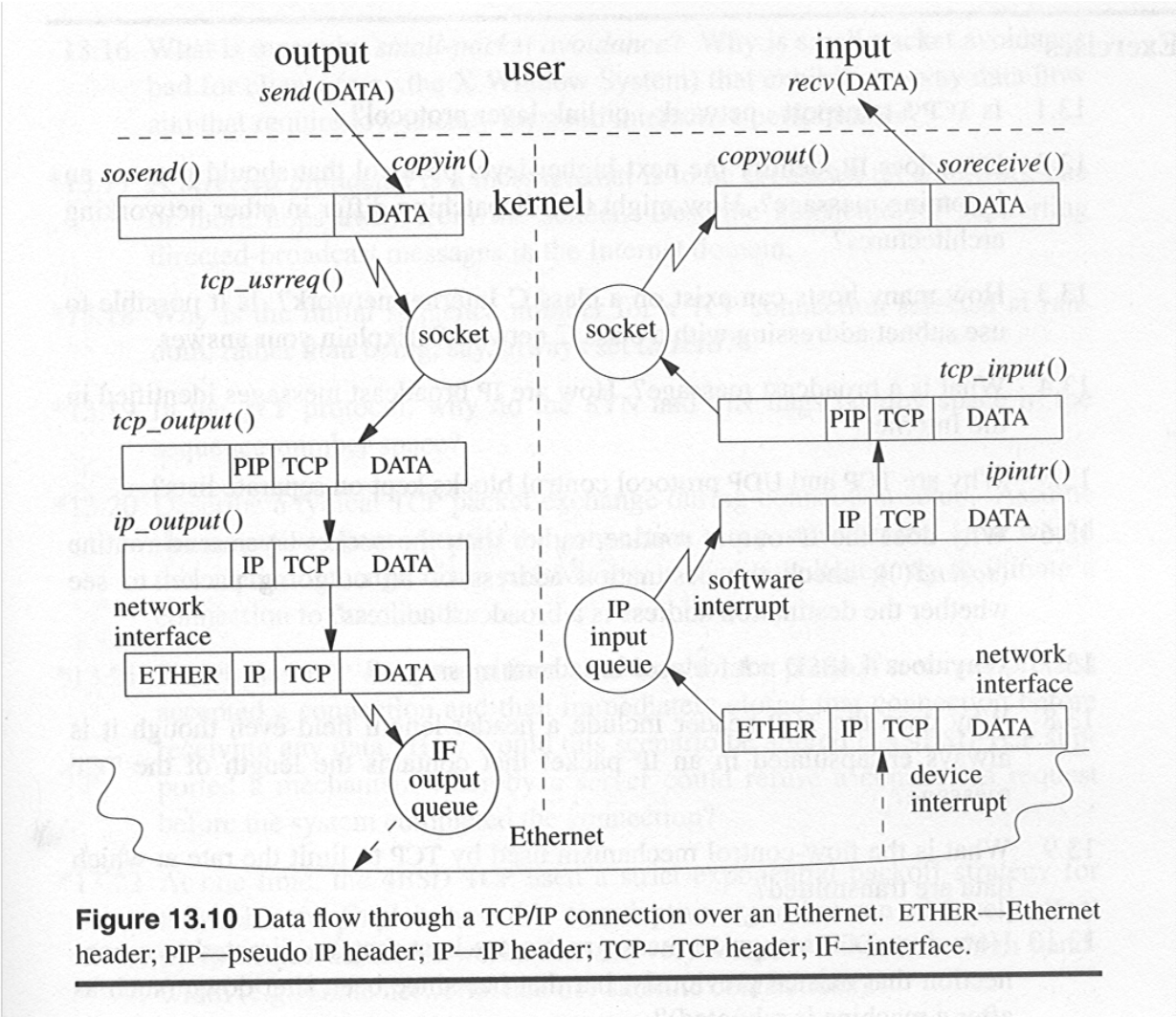


Figure 13.10 Data flow through a TCP/IP connection over an Ethernet. ETHER—Ethernet header; PIP—pseudo IP header; IP—IP header; TCP—TCP header; IF—interface.

Sending Side:

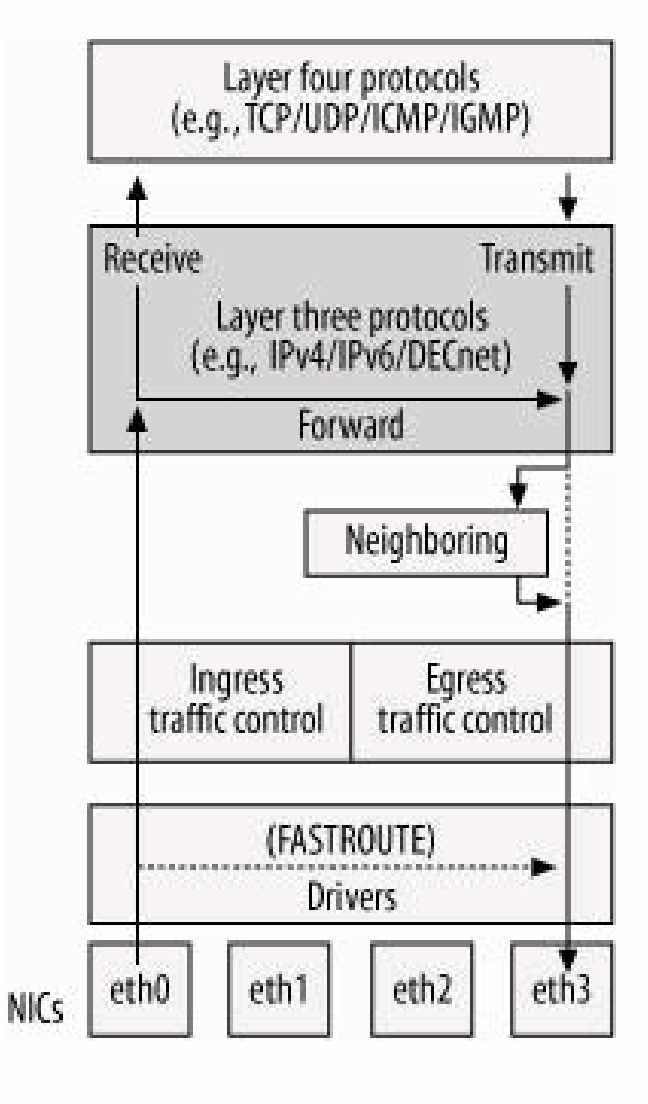
- Data sent through a TCP socket will be queued in the socket send buffer first.
  - Compared to UDP:
    - Data sent through a UDP socket will go all the way down to the network interface. The socket send buffer will not be used.
      - So, the UDP socket send buffer will never overflow.
      - Instead, the network interface's output queue may overflow. A UDP packet may be dropped at the interface.
        - The returned value of write() or sendto() will be -1 to immediately tell you about this.
  - Why?
    - Unlike UDP, TCP implements flow control.
      - Data to be sent thus may need a place to wait for its turn.
    - Unlike UDP, TCP provides a reliable delivery service.
      - Sent data may get lost, corrupted in a network. In this case, they need to be resent.
        - So even if a piece of data has been sent to the network, it still cannot be removed from the socket send buffer.
        - Only when the corresponding ACK has been received, will the sent data be removed.

- When the socket buffer is already full, the sending process is put to sleep.
  - A write(socket, buf, n) system call may not always copy N bytes to the send buffer.
    - You better check the returned value!
- So data will not be dropped in a TCP socket send buffer.
  - However, data may still get dropped at the network interface.
  - Some reasons:
    - When a TCP connection's throughput exceeds the link speed
      - Common when a 56 Kbps modem is used
      - Uncommon when a 100 Mbps Fast Ethernet is used.
    - When a TCP connection's slow start generates a long burst of packets
      - May happen
    - When there are many TCP connections on the system and their aggregate output throughput exceeds the link's speed
      - Common on a busy web server
  - When a TCP packet is dropped at the network interface, tcp_quench() will be called to reduce the TCP window size to only one packet.

- When some data is removed from the send buffer, the sleeping process will be waked up.
  - So that it can continue to dump more data to the socket.
  - To avoid waking up the process too frequently and unnecessarily, a low watermark is used.
    - Do not wake up the process if only a few bytes of buffer space become available.
    - Wakeup the process only when the buffer occupancy drops below the low watermark.

Linux Networking Implementation

Processing Flow Overview



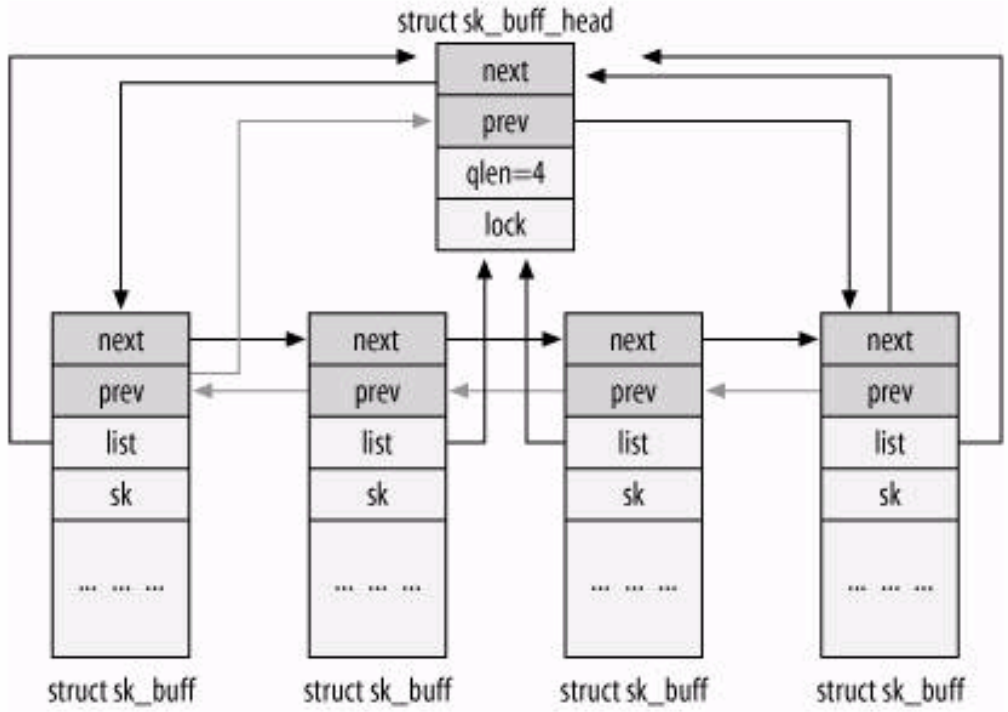
Fundamental Data Structure – sk_buff

- Defined in <*include/linux/skbuff.h*>
- Changed, reorganized, and expanded with new fields version by version
- Four categories of its fields
  - Layout
  - Socket Information
  - General purpose
  - Network feature specific
  - Management function

Layout Field

- Sk_buff_head
  - Global variable as the list head
  - Qlen denotes the number of sk_buff instances
  - Lock guarantees that only no users can access the list simultaneously.

- List maintenance fields
  - next
  - prev
  - list



Socket Information – "struct sock"

- Field in sk_buff: struct sock* sk
- a pointer to a sock data structure of the socket that owns this buffer.
- The control block for a socket
- A huge data structure
- needed when data is either locally generated or being received by a local process
  - User data and socket-related information is used by L4 (TCP or UDP) and by the user application.
  - Points to NULL if the sk_buff contains a forwarded pkt. (neither the source nor the destination is on the local machine)

The “proto” Structure in “sock”

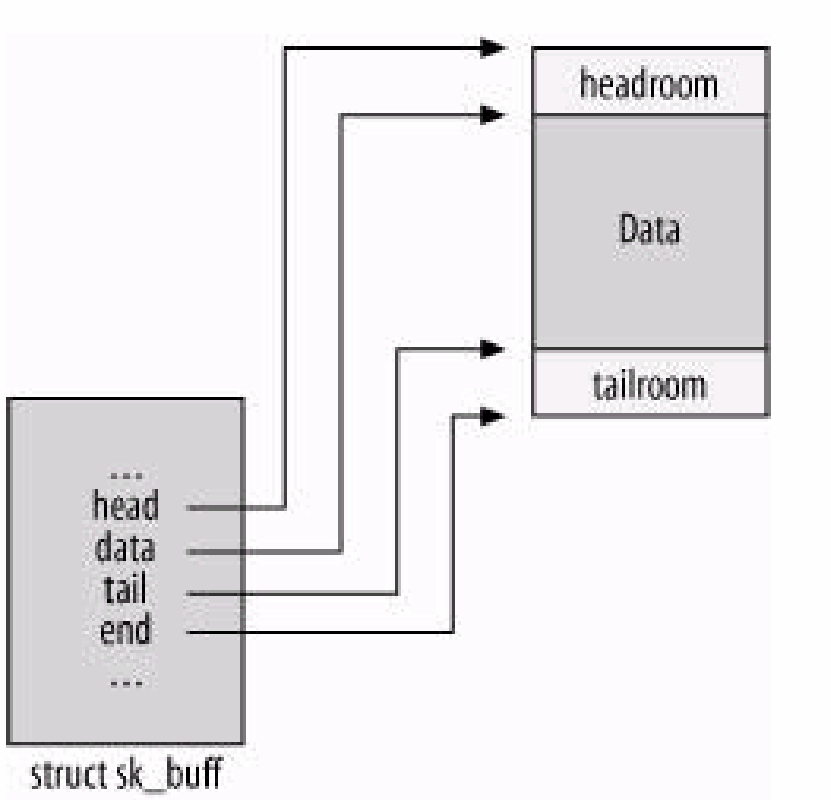
- Stores function pointers used by various socket operations
- Provides the entry points from socket layer to next layers
- Important fields
  - close
  - bind
  - connect
  - disconnect
  - accept
  - ioctl
  - init
  - setsockopt
  - getsockopt
  - sendmsg
  - recvmsg
  - backlog_rcv

Socket Buffer Layer

- unsigned char *head
- unsigned char *end
- unsigned char *data
- unsigned char *tail

- Headroom reserved for pre-pending headers of lower layers

- Tailroom reserved for error-checking redundancy data



The Header-related Fields

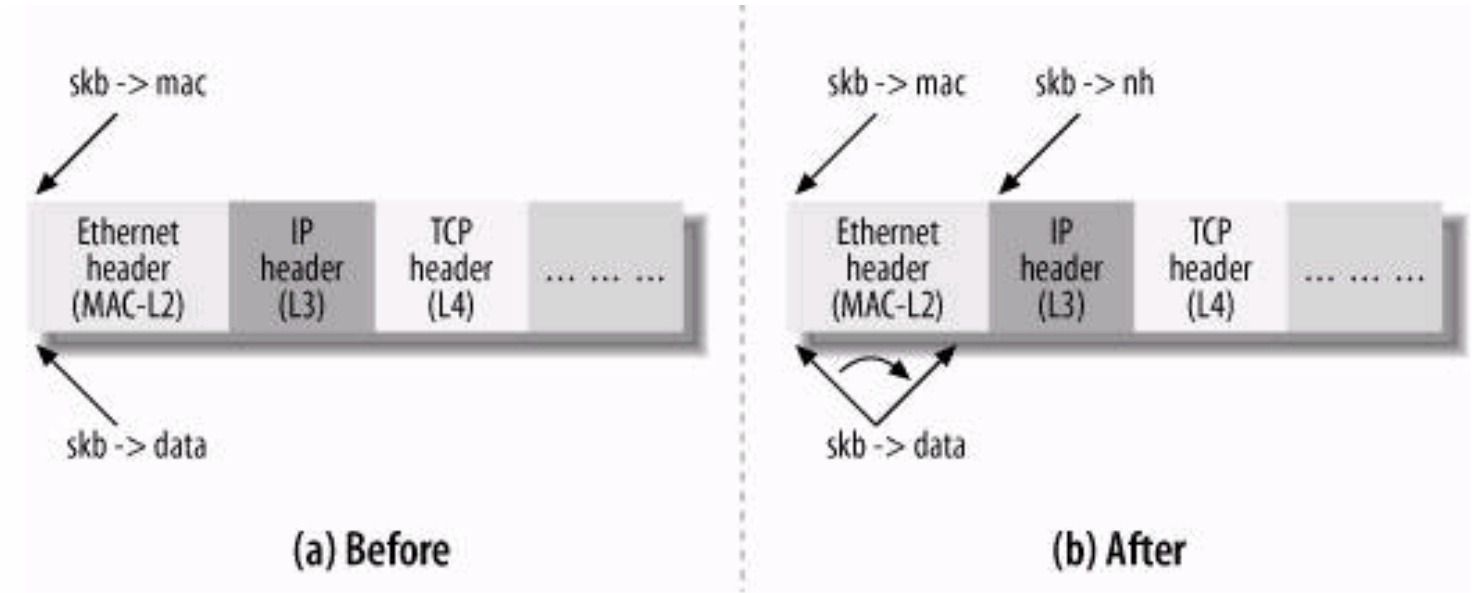
```
union {
        struct tcphdr   *th;
        struct udphdr   *uh;
        struct icmphdr  *icmph;
        struct igmphdr  *igmph;
        struct iphdr    *ipiph;
        struct ipv6hdr  *ipv6h;
        unsigned char   *raw;
} h;

union {
        struct iphdr    *iph;
        struct ipv6hdr  *ipv6h;
        struct arphdr   *arph;
        unsigned char   *raw;
} nh;

union {
        unsigned char   *raw;
} mac;
```

Header Stripping

- Carried out by only manipulating pointers.
- Efficient than copying a header-stripped packet.



General Purpose Field

- Timestamp
- Network Device Structure
  - dev, input dev, real dev
- Routing
- Header Descriptor
- Checksum
- Share flag
  - Indicate if this sk_buff is a clone of another one or not.
- Packet Type
- Priority
- Protocol security

The Field for Routing

- Field: struct dst_entry dst
  - Used by routing subsystem
  - Determines the next-hop of the packet contained in this socket buffer
- Discussed later

The Control Block Field

- Field: char cb[40]
- Used to store private data needed by each layer
- Example:
  - TCP use the cb field to store the seq number of a pacaket.

  - struct tcp_skb_cb {
    - ... ...
    - _ _u32 seq; /* Starting sequence number */
    - _ _u32 end_seq; /* SEQ + FIN + SYN + datalen*/
    - _ _u32 when; /* used to compute rtt's */
    - _ _u8 flags;
    - /* TCP header flags. */
    - ... ...

  - };

The Checksum Related Field

- Fields: skb->csum and skb->ip_summed
- Different meanings depending on whether skb points to a received packet or to an outgoing packet
- When a packet is received:
- The skb->csum field may hold its L4 checksum.
- The skb->ip_summed field keeps track of the status of the L4 checksum.
  - Required by interacting with NIC
  - Possible value
    - CHECKSUM_NONE
      - invalid checksum, need to recompute it at L4
    - CHECKSUM_HW
      - NIC's computed partial checksum, L4 needs to add the checksum of the pseudo-header
    - CHECKSUM_UNNECESSARY
      - NIC has done all checksum checking, including pseudo-header

The Checksum Related Field .1

- When a packet is to be transmitted:
- csum represents an offset.
  - used only if the checksum is calculated in hardware.
  - not stores the checksum itself
  - describes the place inside the buffer where the hardware card has to put the checksum it will compute

- The proactive protection on pseudo-header may make checksum checking failed.
  - For example, NAT manipulates the fields of the IP header used by the L4 layer to compute the so-called checksum on the pseudo-header would invalidate that data structure

The Packet Type Field

- PACKET_HOST
  - The destination address of the received frame is that of the receiving interface
  - That is, the packet has reached its destination.
- PACKET_MULTICAST
  - The destination address of the received frame is one of the multicast addresses to which the interface is registered.
- PACKET_BROADCAST
  - The destination address of the received frame is the broadcast address of the receiving interface.
- PACKET_OTHERHOST
  - The destination address of the received frame does not belong to the ones associated with the interface (unicast, multicast, and broadcast); thus, the frame will have to be forwarded if forwarding is enabled, and dropped otherwise.
- PACKET_OUTGOING
  - The packet is being sent out
- PACKET_LOOPBACK
  - The packet is being sent out to the loopback device.
  - Thanks to this flag, when dealing with the loopback device, the kernel can skip some operations needed for real devices.
- PACKET_FASTROUTE
  - The packet is being routed using the Fastroute feature. Fastroute support is not available anymore in 2.6 kernels.

The Priority Field

- indicates the QOS class of a packet being transmitted or forwarded.
- If the packet is generated locally, the socket layer defines the priority value.
- If the packet is being forwarded, the function rt_tos2priority (called from the ip_forward function) defines the value of the field according to the value of the TOS field in the IP header.
- DiffServ dose not use this field.

Protocol

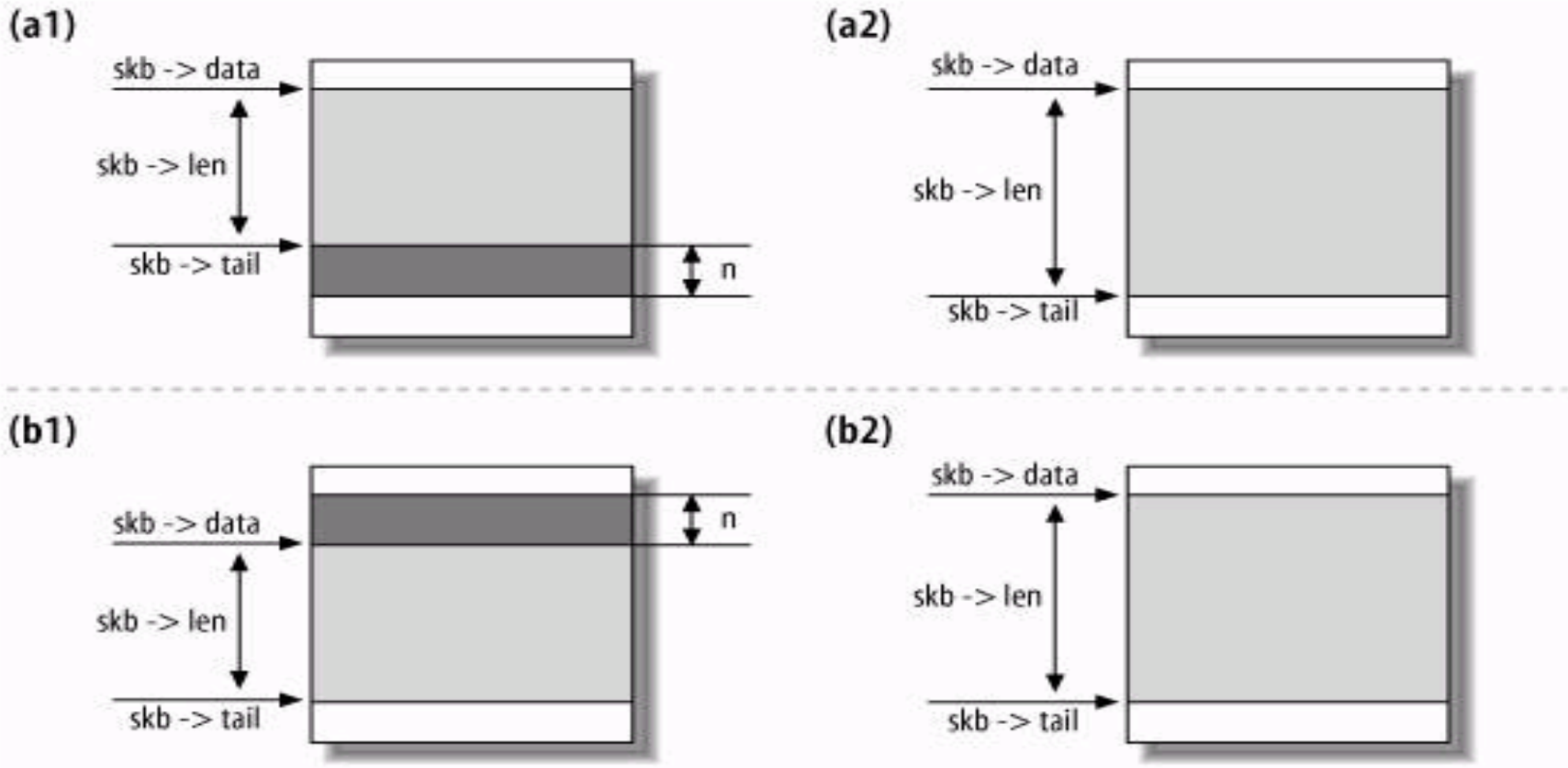
- Used by L2 to know which L3 protocol should handle this packet
- Typically, IP, IPv6, and ARP
- A complete list is available in *include/linux/if_ether.h*.
- Driver calls netif_rx to invoke the handler for the upper network layer indicated by this field

Feature Specific Field

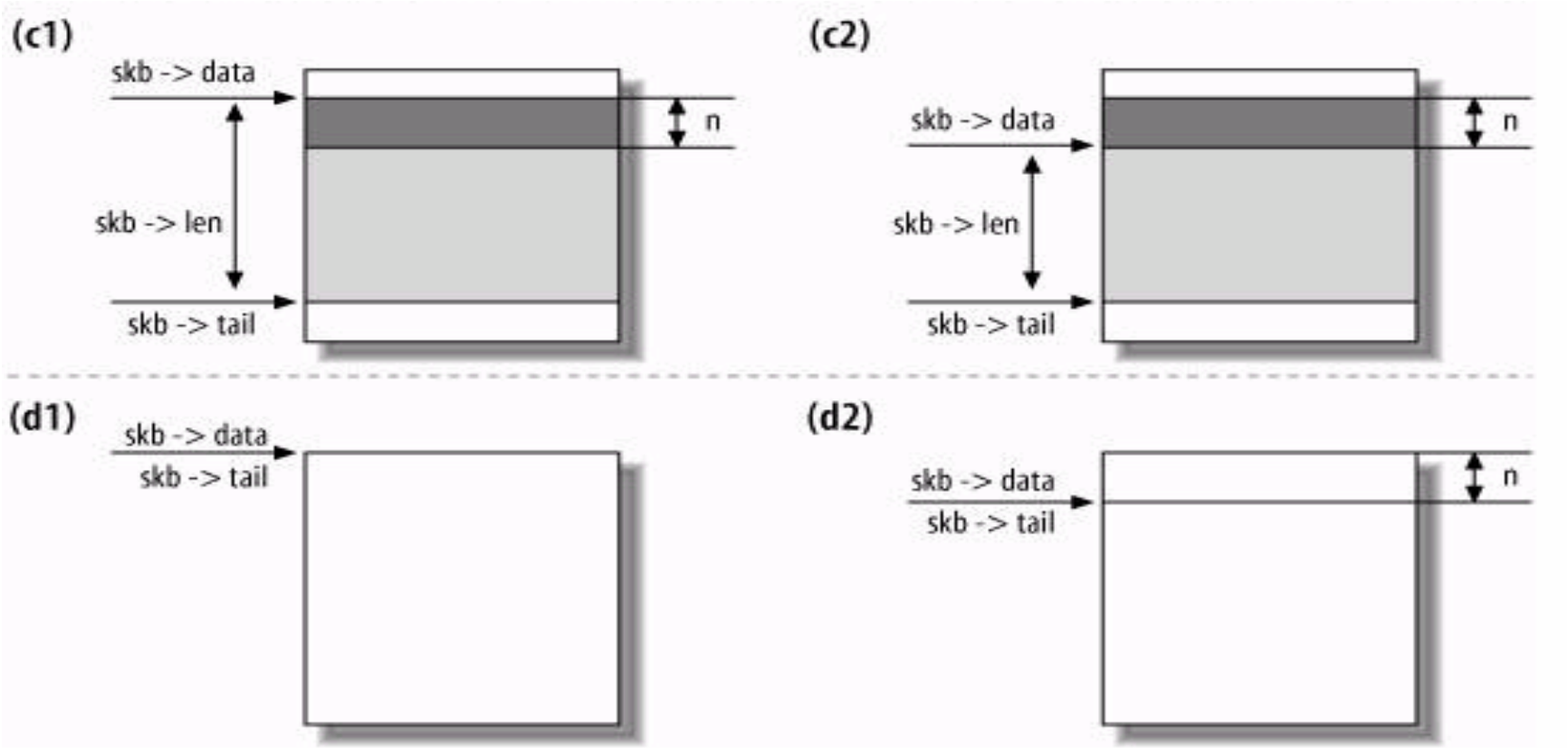
- A variety of specific fields dedicated for particular features.
  - Netfilter
  - Bridge
  - Traffic Control
  - IP Sec

Commonly Used Function For “sk_buff”

sk_put and sk_push

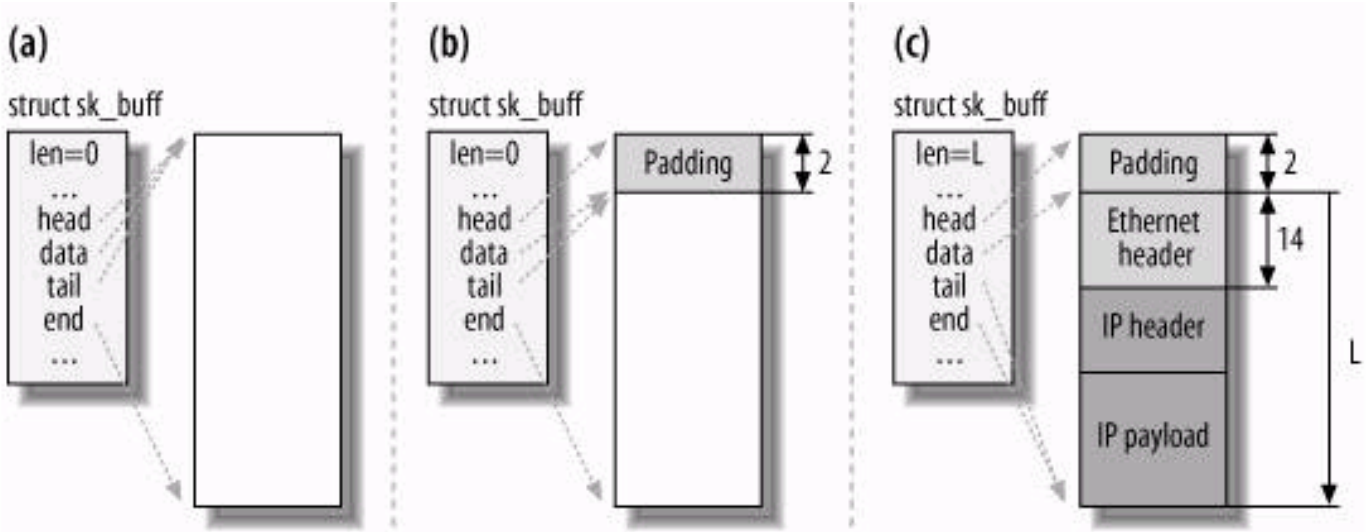


Sk_pull and sk_reserve



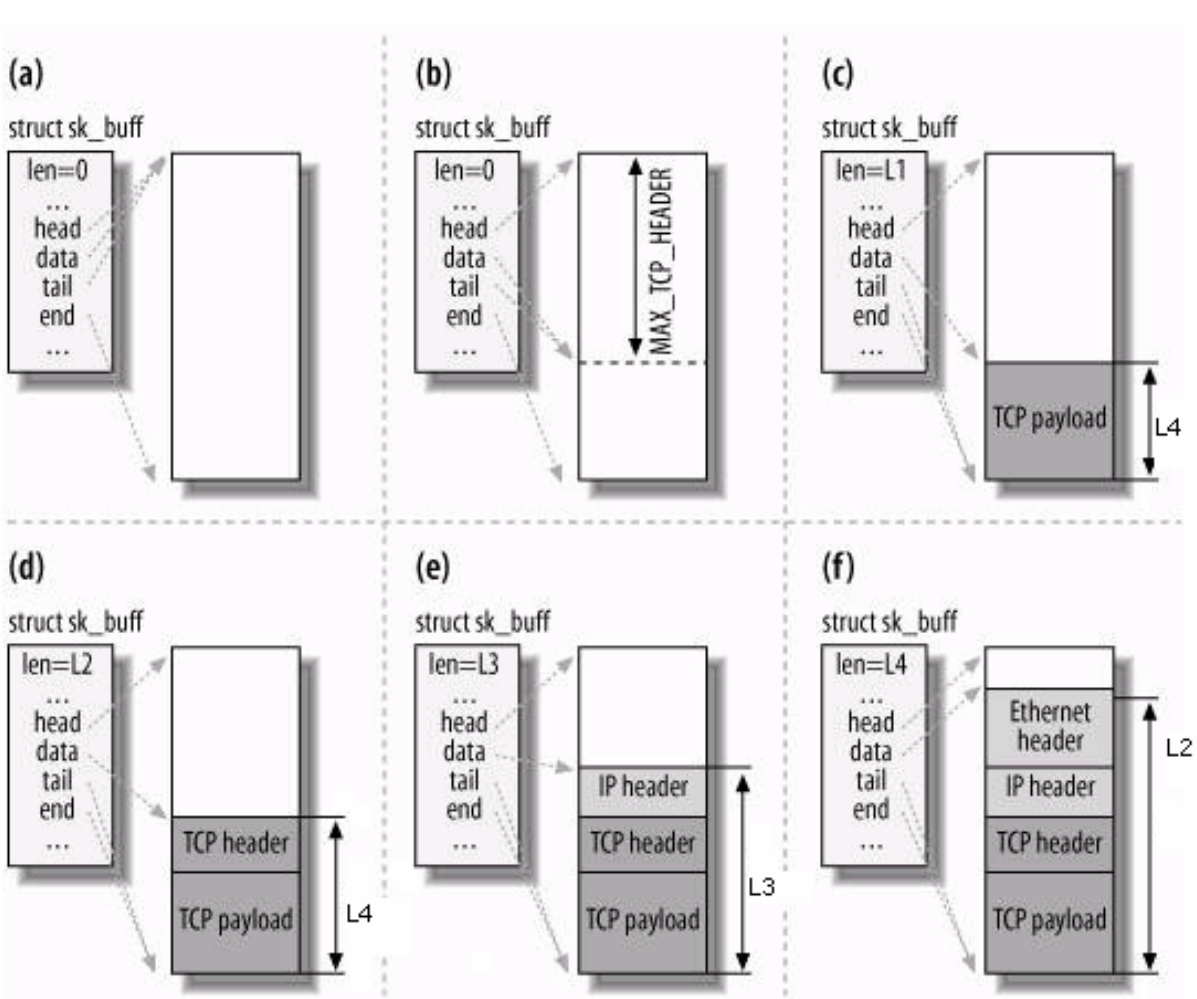
An Example of sk_reserve

- When driver receives a packet, it reserves 2 bytes with sk_reserve() to make IP header is stored with an alignment of 16-byte boundary.



Case Study 1 – The Processing For Transmitting A Packet

Socket Buffer Usage



An Example of Sending Data

- (1) send() → sys_send() → sys_sendto() →
- (2) sys_sendmsg():
  - copy data from user-space to a kernel buffer.
- (3) sock_sendmsg() → __sock_sendmsg()
- (4) sock->ops->sendmsg()
- (5) tcp_sendmsg() → tcp_push()

- (6) tcp_push_pending_frames()
- (7) __tcp_push_pending_frames
- (8) tcp_write_xmit()
  - Write a packet to the network layer
- (9) tcp_cwnd_validate()
  - congestion control validation

The tcp_write_xmit() Function

- Check tcp_state of the given sock structure sk:
- If sk->sk_state != TCP_CLOSE, then
  - while ((there is a valid socket buffer) and (tcp_snd_test() returns true) {
    - 1. if (skb->len > mss_len) call tcp_fragment() to fragment user-level data.
    - 2. call tcp_transmit_skb() to send a packet down to the network layer.
      - If an error occurs, break
    - 3. call update_send_head() to update the head of send buffer.
  - }
- Call tcp_cwnd_validate() to check the status of the congestion window.

The tcp_transmit_skb() function

- 1. add TCP header for a packet
- 2. Update some fields for congestion control mechanism and manipulate related timers.
  - call tcp_event_ack_sent() if the packet is an ACK packet.
  - call tcp_event_data_sent() if the packet is an DATA packet.
- 3. Call tp->af_specific->queue_xmit(skb) to indirectly invoke ip_queue_xmit()

- Tp is a pointer of struct tcp_sock
- Af_specific is pointer to struct tcp_func

TCP Checksum

- TCP/UDP and IP checksum routines are defined in include/asm-xxx/checksum.h
- TCP still computes checksum for pseudo-header.
  - Source IP address
  - Destination IP address
  - TCP protocol identifier in IP header
  - IP payload length

- Checksum routines used by TCP
  - (1) tcp_v4_send_check()
  - (2) tcp_v4_check(csum_partial(th))
  - (3) csum_tcpudp_magic()
  - (4) csum_fold(csum_tcpudp_nofold())

Example -- tcp_v4_send_check

```
/* This routine computes an IPv4 TCP checksum. */
void tcp_v4_send_check(struct sock *sk, struct tcphdr *th, int len,
                       struct sk_buff *skb)
{
        struct inet_sock *inet = inet_sk(sk);

        if (skb->ip_summed == CHECKSUM_HW) {
                th->check = ~tcp_v4_check(th, len, inet->saddr, inet->daddr, 0);
                skb->csum = offsetof(struct tcphdr, check);
        } else {
                th->check = tcp_v4_check(th, len, inet->saddr, inet->daddr,
                                         csum_partial((char *)th, th->doff << 2,skb->csum));
        }
}
```

- csum_partial is an assembly function that computes the checksum for a given memory region, e.g., a TCP header.

Csum_fold and Csum_tcpudp_nofold

```
static inline unsigned int csum_fold(unsigned int sum)
{
        __asm__(
                "addl %1, %0            ;\n"
                "adcl $0xffff, %0       ;\n"
                : "=r" (sum)
                : "r" (sum << 16), "0" (sum & 0xffff0000)
        );
        return (~sum) >> 16;
}
```

```
static inline unsigned long csum_tcpudp_nofold
(unsigned long saddr, unsigned long daddr, unsigned short len, unsigned  short proto, unsigned int sum) {

    __asm__(
        "addl %1, %0    ;\n"
        "adcl %2, %0    ;\n"
        "adcl %3, %0    ;\n"
        "adcl $0, %0    ;\n"
        : "=r" (sum)
        : "g" (daddr), "g"(saddr), "g"((ntohs(len)<<16)+proto*256), "0"(sum));
    return sum;

}
```

The Struct tcp_func Structure

- struct tcp_func ipv4_specific = {
-
- .queue_xmit = ip_queue_xmit,
- .send_check = tcp_v4_send_check,
- .rebuild_header = tcp_v4_rebuild_header,
- .conn_request = tcp_v4_conn_request,
- .syn_recv_sock = tcp_v4_syn_recv_sock,
- .remember_stamp = tcp_v4_remember_stamp,
- .net_header_len = sizeof(struct iphdr),
- .setsockopt = ip_setsockopt,
- .getsockopt = ip_getsockopt,
- .addr2sockaddr = v4_addr2sockaddr,
- .sockaddr_len = sizeof(struct sockaddr_in),
-
- };

TCP Vegas

- 31%~71% better throughputs and 20%~50% less losses compared against to TCP Reno.
- used by Linux
  - TCP Reno is used by BSD 4.4
- Features
  - New retransmission mechanism
  - New congestion avoidance algorithm
  - Modified slow start algorithm

The Retransmission Mechanism in TCP Vegas

- In TCP *Reno:*
  - *RTT* and *mean variance* estimates are computed using a coarse-grained timer (around *500ms*).
  - The *RTT* estimate is not very accurate.
  - This granularity influence also how often *TCP* checks to see if it should timeout on a segment.
  - An average of *1100ms* from the time it sent a segment that was lost, until it timed out and resent the segment, whereas less that *300ms* would have been the correct timeout interval *had a more accurate clock been used*.
- *TCP Vegas* then fixes this problem using a finer coarse-grained timer.

The Retransmission Mechanism in TCP Vegas .1

- extended retransmission mechanism:
- the *system clock* is read and recorded each time a segment is sent;
- when an *ACK* arrives, the clock is read again and the *RTT* calculation is done using this time and the timestamp recorded for the relevant segment.
- Using this more accurate *RTT*, retransmission is decided as follows:

The Retransmission Mechanism in TCP Vegas .2

- a.- When a *duplicated ACK* is received, *Vegas* checks to see if:
  - the new *RTT* (*current time - timestamp recorded*) is greater than *RTO*.
  - If it's, *Vegas* retransmits the segment without having to wait for the third duplicated ACK.
- In many cases third duplicated ACK is never received, and therefore, *Reno* would have to rely on the coarse-grained timeout to catch the loss.

The Retransmission Mechanism in TCP Vegas .3

- b.- When a *non-duplicated ACK* is received:
- if it is the first or second one after a retransmission, *Vegas* checks again to see if *RTT* > *RTO*;
- if so, then the segment is retransmitted.

- This will catch any other segment that may have lost previous to the retransmission without having to wait for a *duplicated ACK*.

- In other words, *Vegas* treats the receipt of certain *ACKs* as a trigger to check if a timeout should happen;

- but still contains *Reno's* coarse-grained timeout code in case these mechanism fail to recognize a lost segment.

The Retransmission Mechanism in TCP Vegas .4

- Finally, in *Reno*, it is possible to decrease the congestion window more than once for losses occurring during one *RTT* interval.
- In contrast, *Vegas* only decreases the congestion window if a retransmitted segment was sent after the last decrease.
- Any losses that happened before the last decrease are ignore, and therefore, do not imply that it should be decreased again.
- This change is needed because *Vegas* detects losses much sooner than *Reno*.

The Congestion Avoidance Algorithm in TCP Vegas

- *Reno's* congestion detection uses the loss of segments as a signal of congestion.
- It has no mechanism to detect the incipient stages of congestion - *before losses occur* - so they can be prevented.
- *Reno* is reactive, rather than proactive, in this respect.
- *Reno* needs to create losses to find the available bandwidth of the connection.

The Congestion Avoidance Algorithm in TCP Vegas .1

- *Vegas* implementation uses the idea to *measure and control the amount of extra data that a connection has in transit*,
  - extra data means data that would not have been sent if the bandwidth used by the connection exactly matches that available bandwidth of the link.
- *Vegas's* goal is to maintain the "right" amount of extra data in the network.
- if a connection is sending too much extra data, it will cause congestion;
- if it's sending too little extra data, it cannot respond rapidly enough to transient increase in the available bandwidth.

The Congestion Avoidance Algorithm in TCP Vegas .2

- Define a given connection's *BaseRTT* to be the *RTT* of a segment when the *connection is not congested*;
- in practice it sets *BaseRTT* to the minimum of all measured *RTTs*;
- it is commonly the *RTT* of the first segment sent by the connection, before the router queues increase due to traffic generated by this connection.
- If we asume we are not overflowing the connection, the expected throughput is given by:
  - *Expected* = *WndowSize* / *BaseRTT*, where *WindowSize* is the size of the current congestion window, which we assume to be equal to the number of bytes outstanding.
- Calculate the *current Actual sending rate* recording how many bytes are transmitted between the time that a segment is sent and its *ack* is received and its *RTT*, and dividing the number of bytes transmitted by the sample *RTT*.
- This calculation is done once per round-trip time.

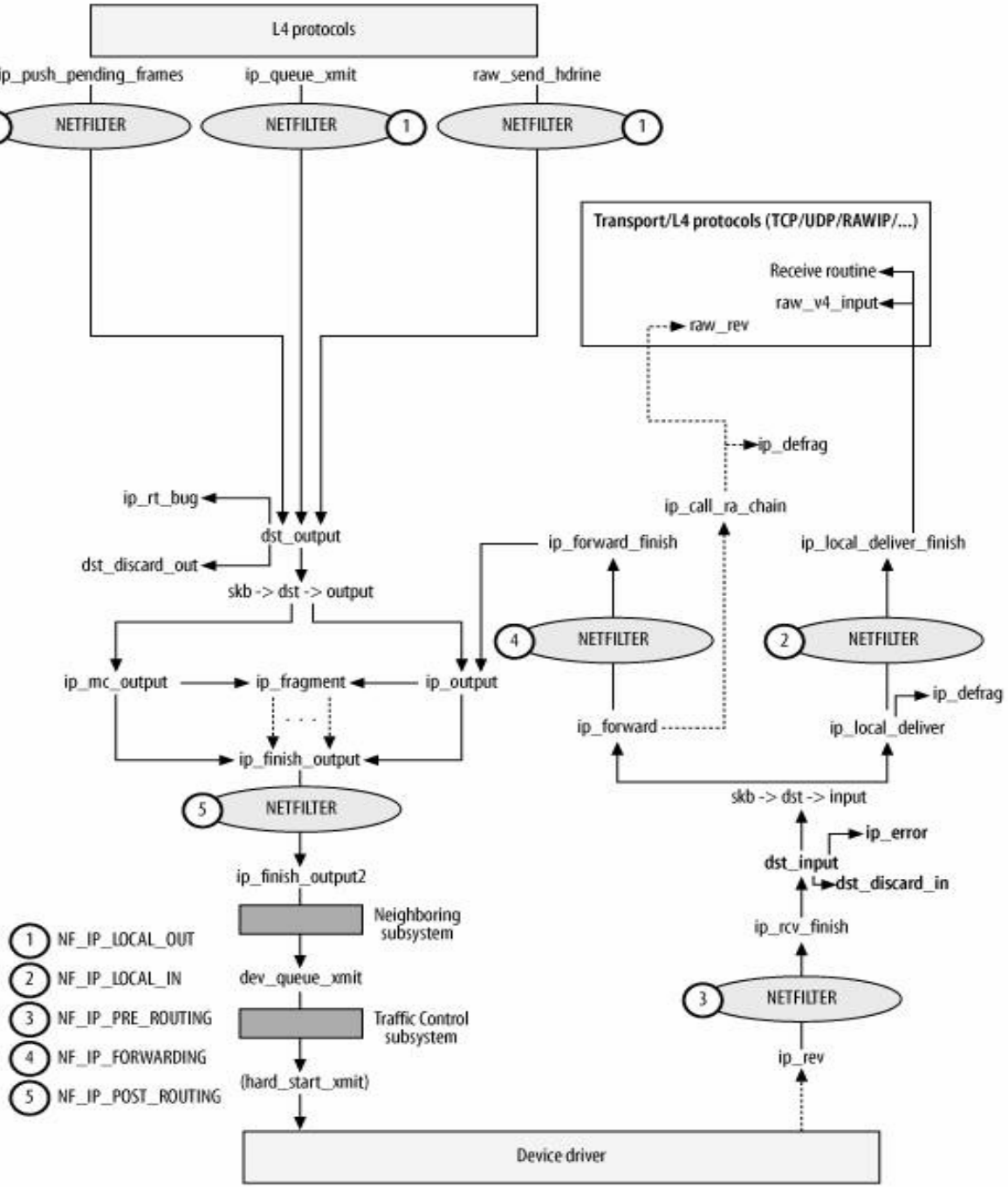
The Congestion Avoidance Algorithm in TCP Vegas .3

- Then compare *Actual to Expected* and *adjust the window accordingly*.
- Let *Diff* = *Expected* - *Actual*.
- *Diff* is positive or zero by definition, since *Actual* > *Expected* implies that we have to change *BaseRTT* to the latest sample *RTT*.
- Define two thresholds *a* and *b*, such that, *a* < *b*, roughly corresponding to having *too little* and *too much* extra data in the network, respectively.
- When *Diff* < *a*, *Vegas* increases the congestion window linearly during the next *RTT*
- when *Diff* > *b*, *Vegas* decrease the congestion window linearly during the next *RTT*.
- The congestion window is *left unchanged* when *a* < *Diff* < *b*.
- The overall goal is to *keep between a and b extrabytes in the network*

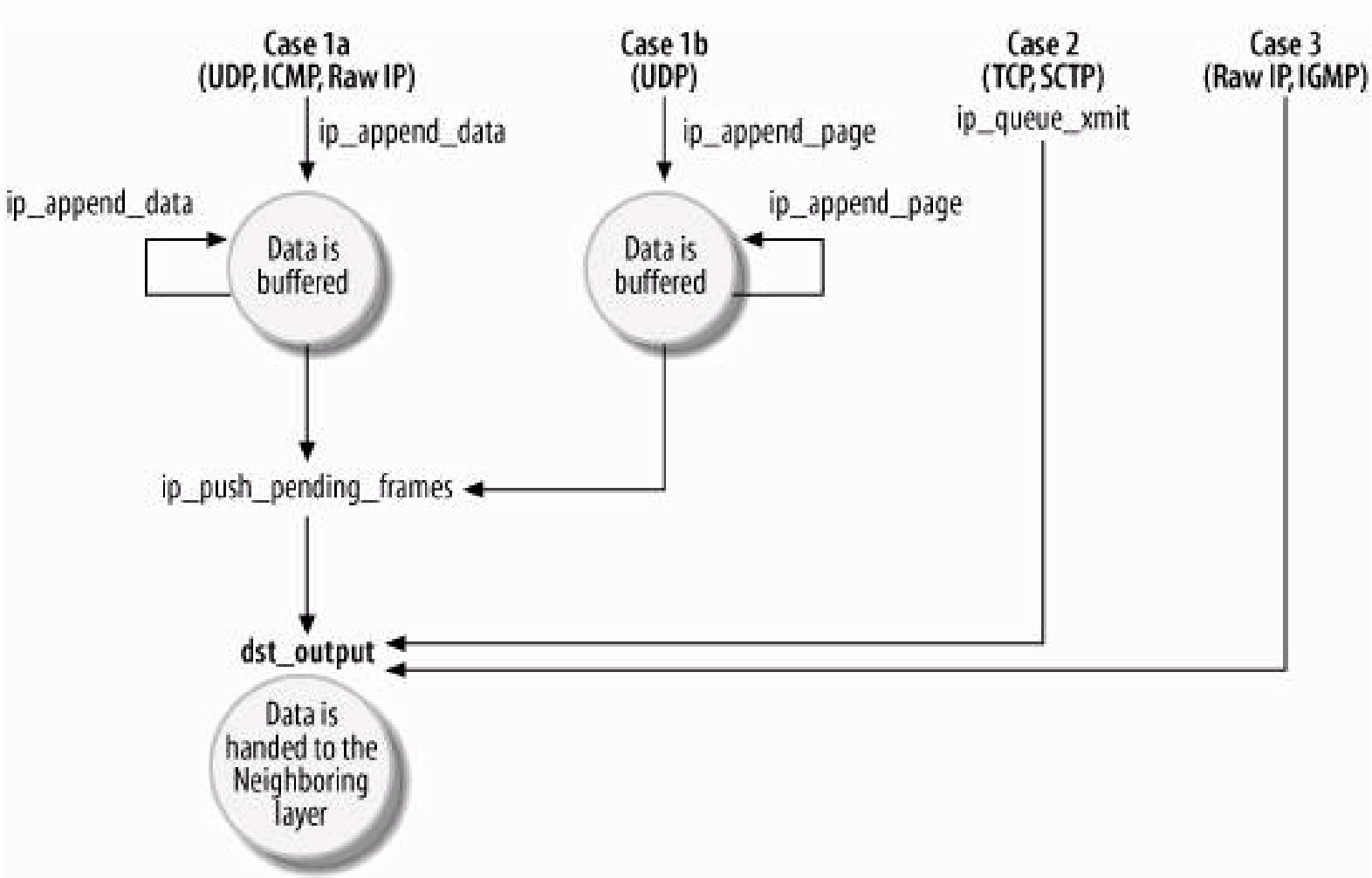
The Modified Slow Start Algorithm in TCP Vegas

- *Reno's slow-start* mechanism is very expensive in terms of losses
  - Double the size of the congestion window every *RTT* while there are no losses
  - equivalent to doubling the attempted throughput every *RTT*
  - when it finally overruns the connection bandwidth, we can expect losses in the order of half the current congestion window, more if we encounter a burst from another connection.
- *Vegas* tries to find a connection's available bandwidth that does not incur this kind of loss. Toward this end, the *congestion detection mechanism* is incorporated into *slow-start* with only minor modifications.
- To be able to detect and avoid congestion during *slow-start*, *Vegas* allows exponential growth only every other *RTT*.
- In between, the congestion window stays fixed
- a valid comparison of the expected and actual rates can be made.
- When the actual rate falls below the expected rate by a certain amount - the γ *threshold* - *Vegas* changes from *slow-start* mode to *linear increase/decrease mode*.

IP Layer


L4 protocols
ip_push_pending_frames
ip_queue_xmit
raw_send_hdrine
NETFILTER
NETFILTER
NETFILTER
Transport/L4 protocols (TCP/UDP/RAWIP/...)
Receive routine
raw_v4_input
raw_rev
ip_defrag
ip_rt_bug
dst_output
dst_discard_out
skb -> dst -> output
ip_call_ra_chain
ip_forward_finish
ip_local_deliver_finish
NETFILTER
NETFILTER
ip_mc_output
ip_fragment
ip_output
ip_defrag
ip_forward
ip_local_deliver
ip_finish_output
skb -> dst -> input
NETFILTER
ip_error
dst_input
dst_discard_in
ip_finish_output2
Neighboring subsystem
ip_rcv_finish
dev_queue_xmit
Traffic Control subsystem
NETFILTER
(hard_start_xmit)
ip_rev
Device driver
1 NF_IP_LOCAL_OUT
2 NF_IP_LOCAL_IN
3 NF_IP_PRE_ROUTING
4 NF_IP_FORWARDING
5 NF_IP_POST_ROUTING

Sending Flow



The ip_queue_xmit Function – used by TCP and SCTP

- The TCP and SCTP protocol deals with packet fragmentation by their own in advance.
- Tasks performed in ip_queue_xmit is rarely simple.
  - Setting the route for the packet.
  - Building the IP header

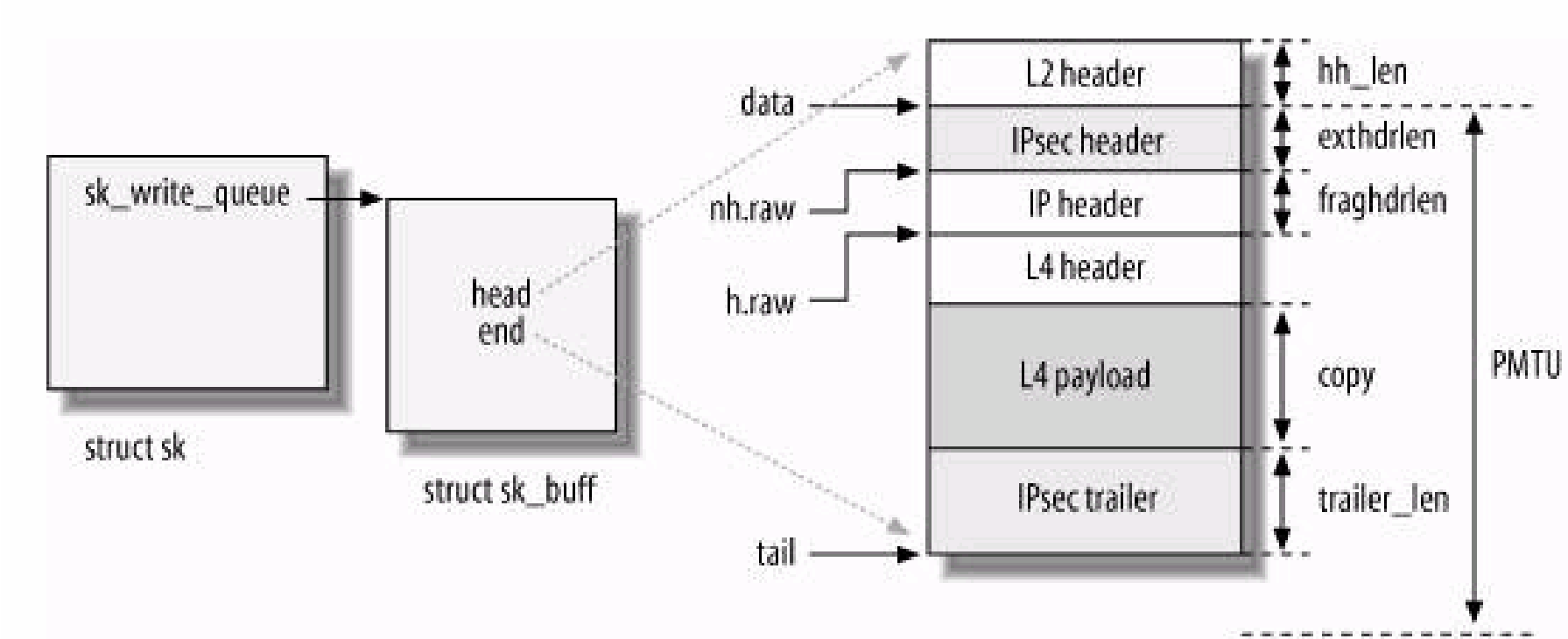
The ip_append_data Function

- Take fragmentation into account
  - PMTU is obtained from routing subsystem
- Consider the support of scatter/gather DMA I/O
- Only allocate buffers and fragment L4 segments
- Not really transmit a packet out
- The transmission is carried out by the ip_push_pending_frames function

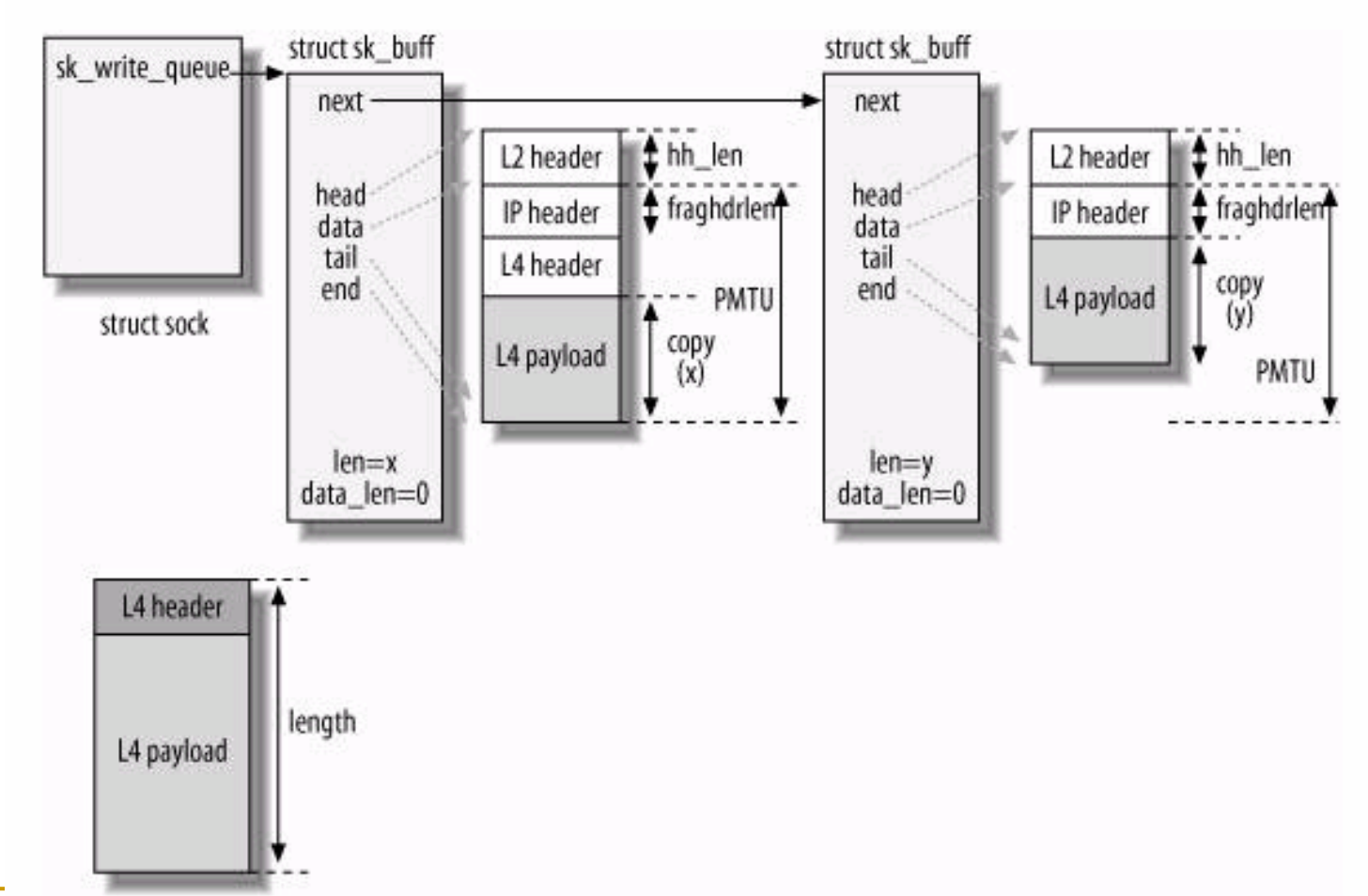
Step

- (1) Set the context
  - Set the size of fragments as PMTU
- (2) Get ready for fragment generation
- (3) Buffer Allocation
- (4) Copy data into the fragments

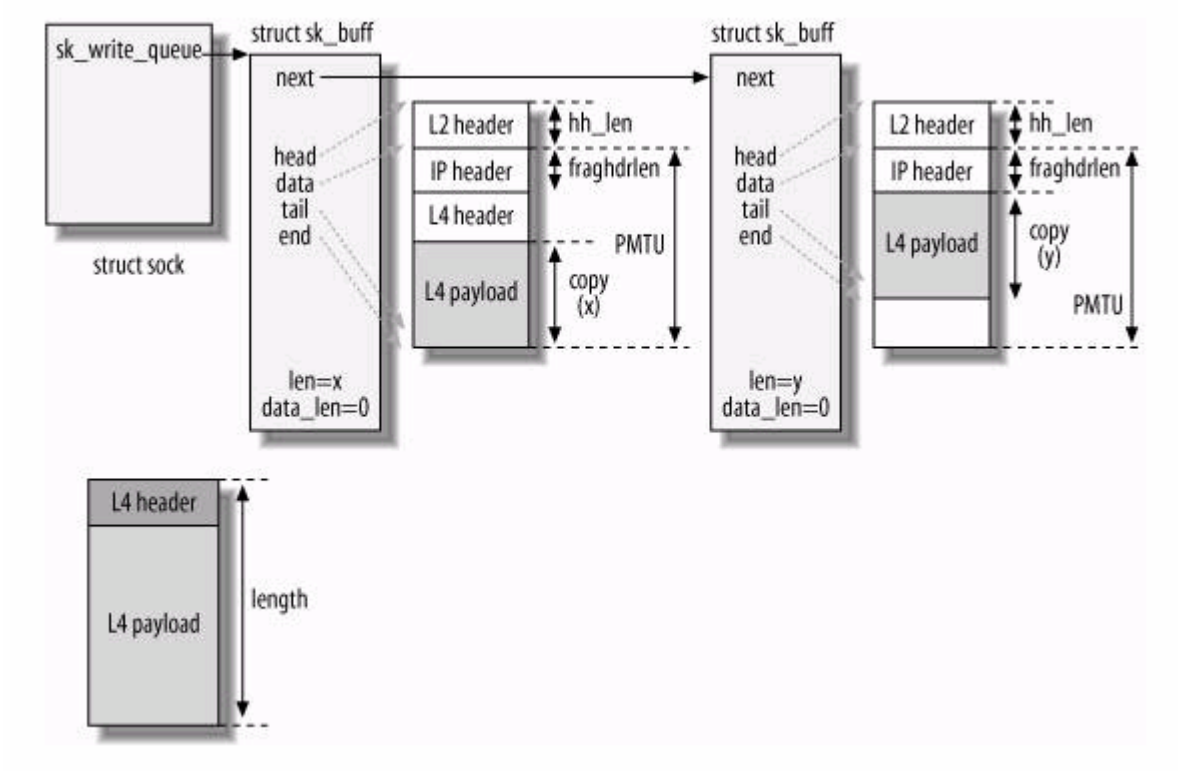
IP Packet with IP Sec Option Without Having To Be Fragmented



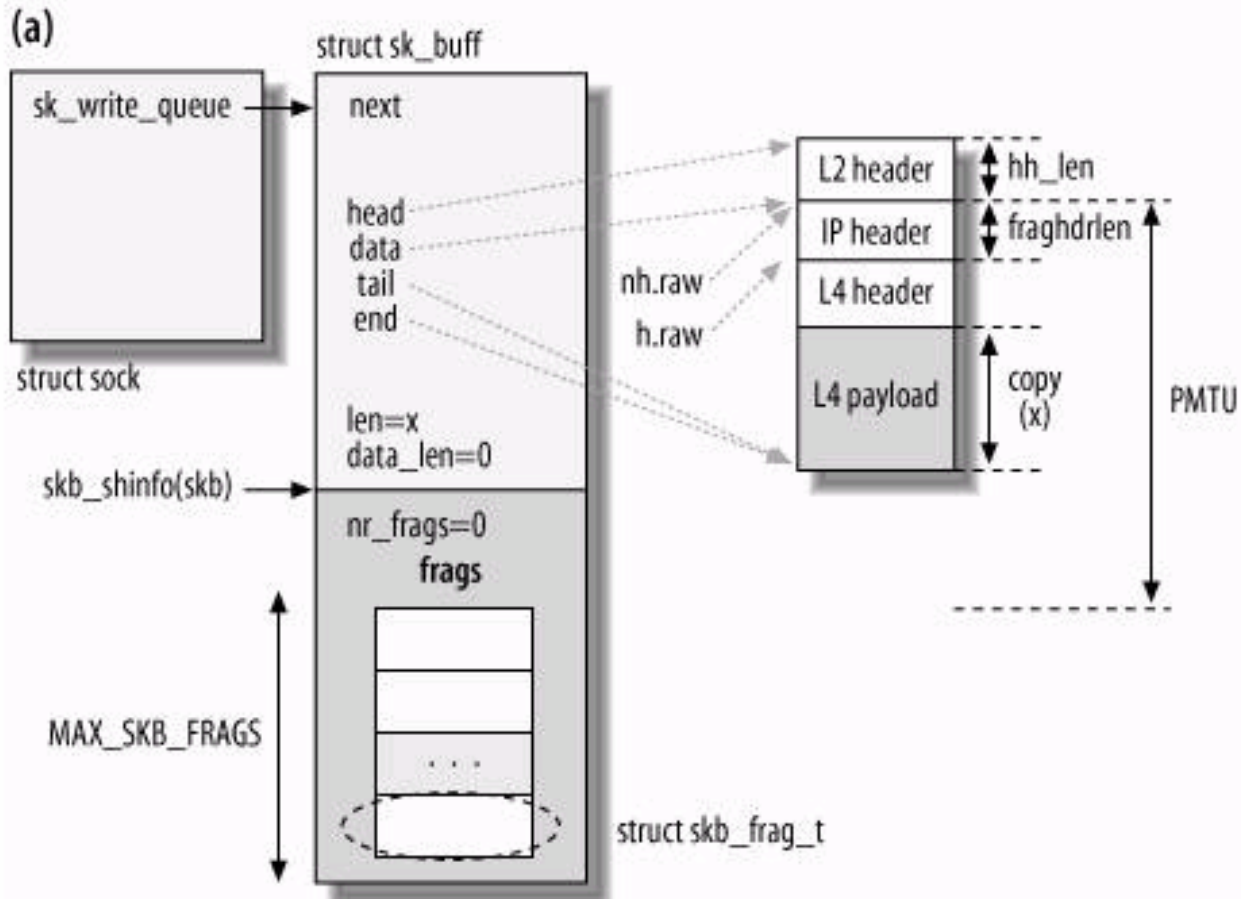
IP Fragmentation Without Scatter-gather I/O And MSG_MORE



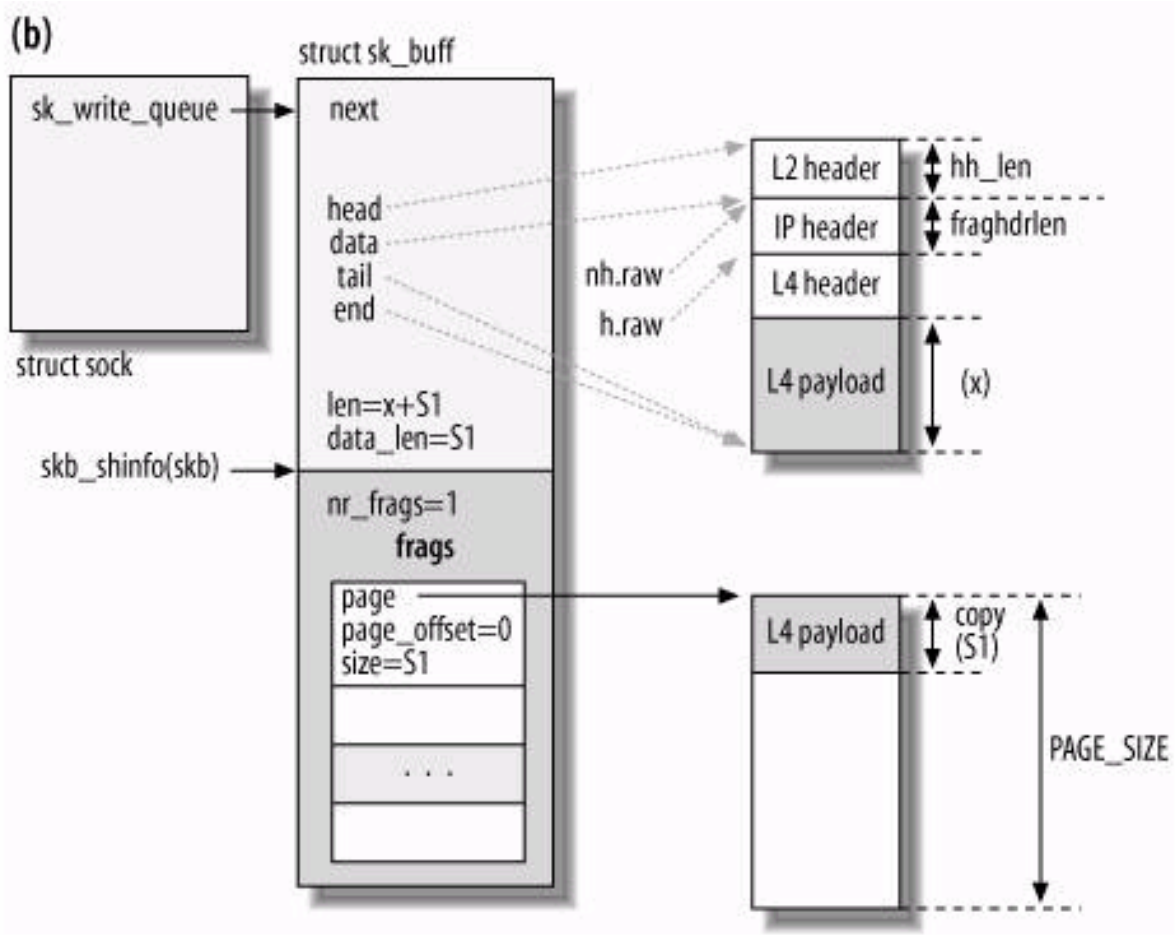
IP Fragmentation With Scatter-gather I/O And MSG_MORE



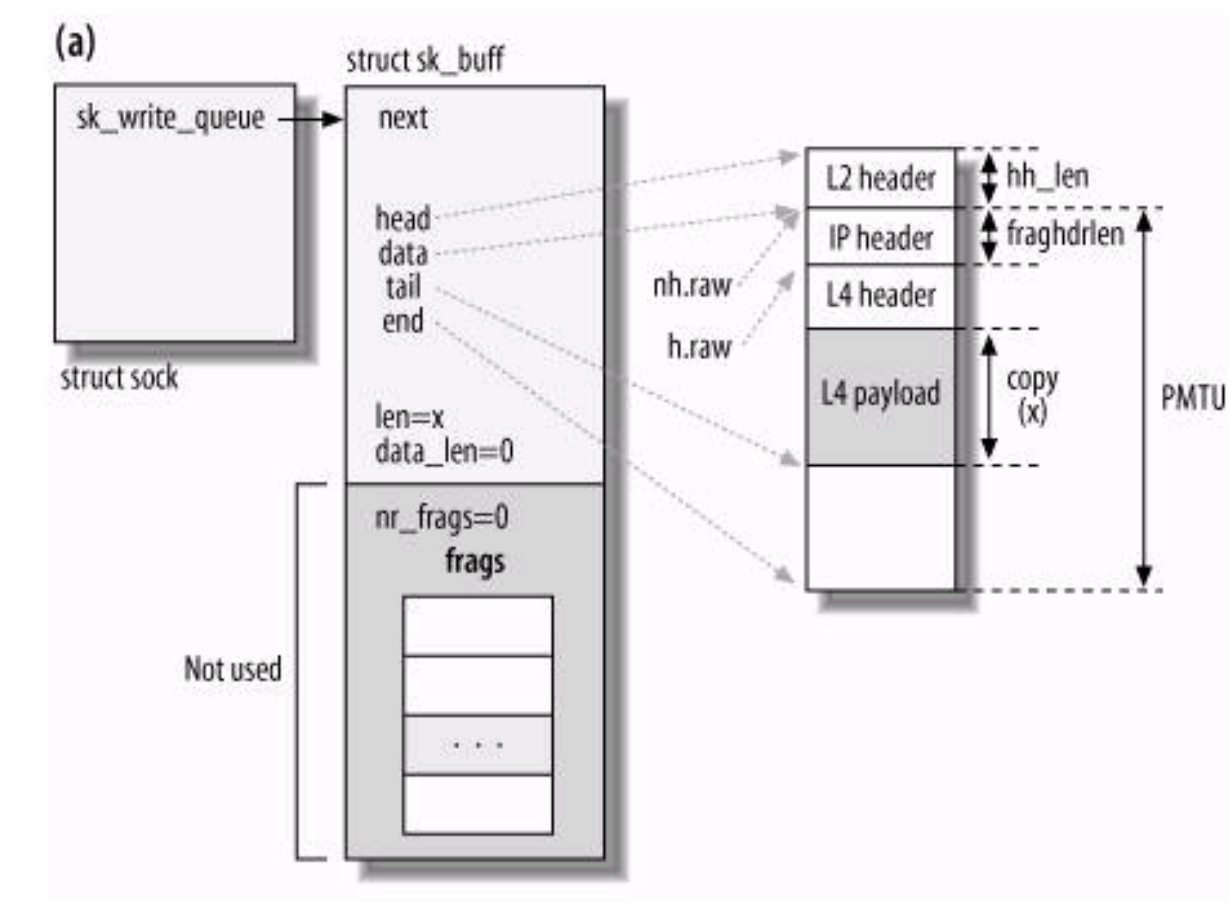
IP Packet Without Fragmentation using Scatter-gather IO (a)



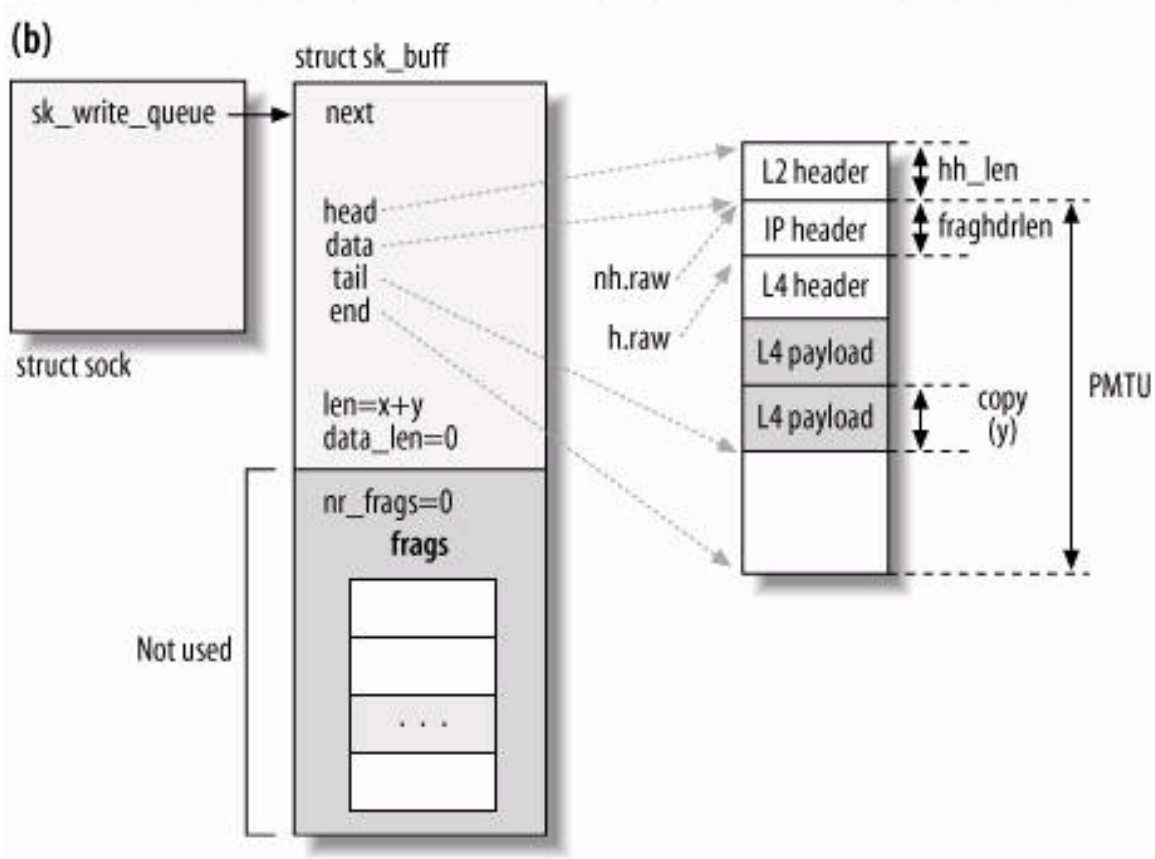
IP Packet Without Fragmentation using Scatter-gather IO (b)



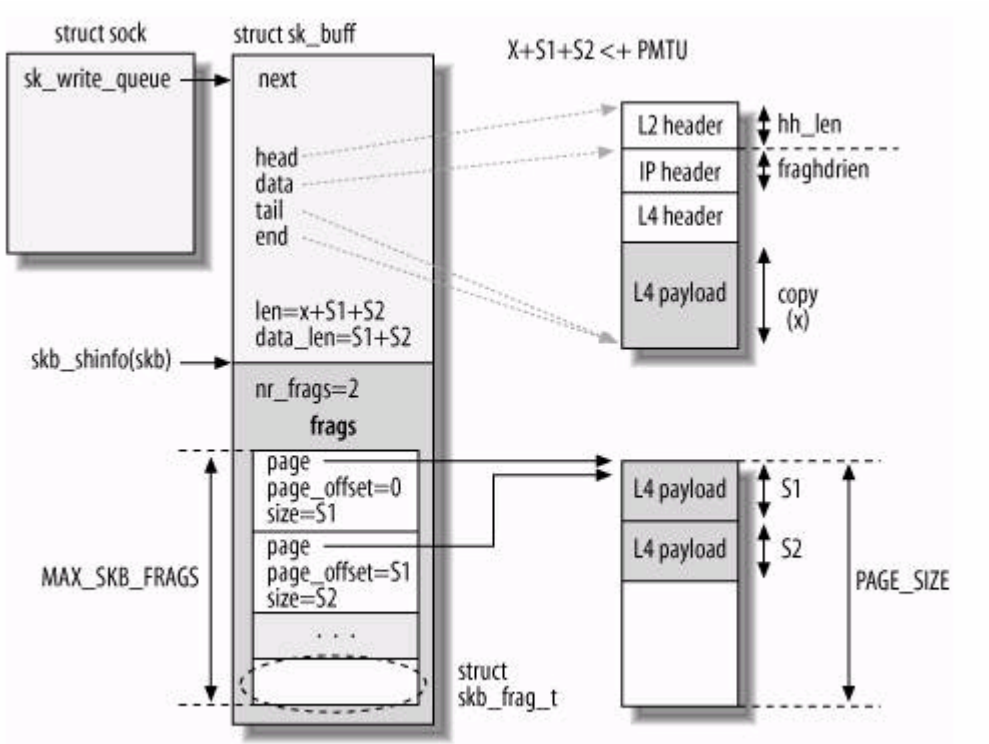
IP Packet Without Fragmentation using Traditional DMA (a)



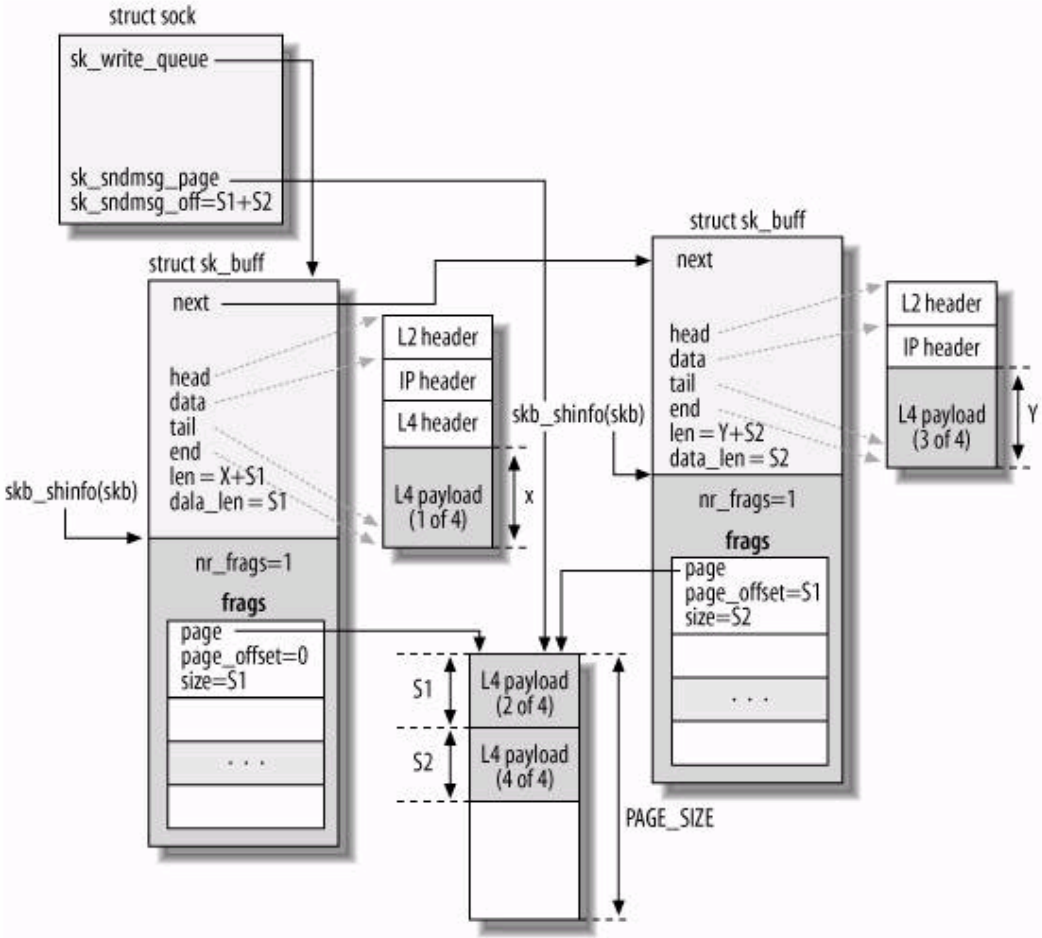
IP Packet Without Fragmentation using Traditional DMA (b)



Fragments With Scatter-gather IO



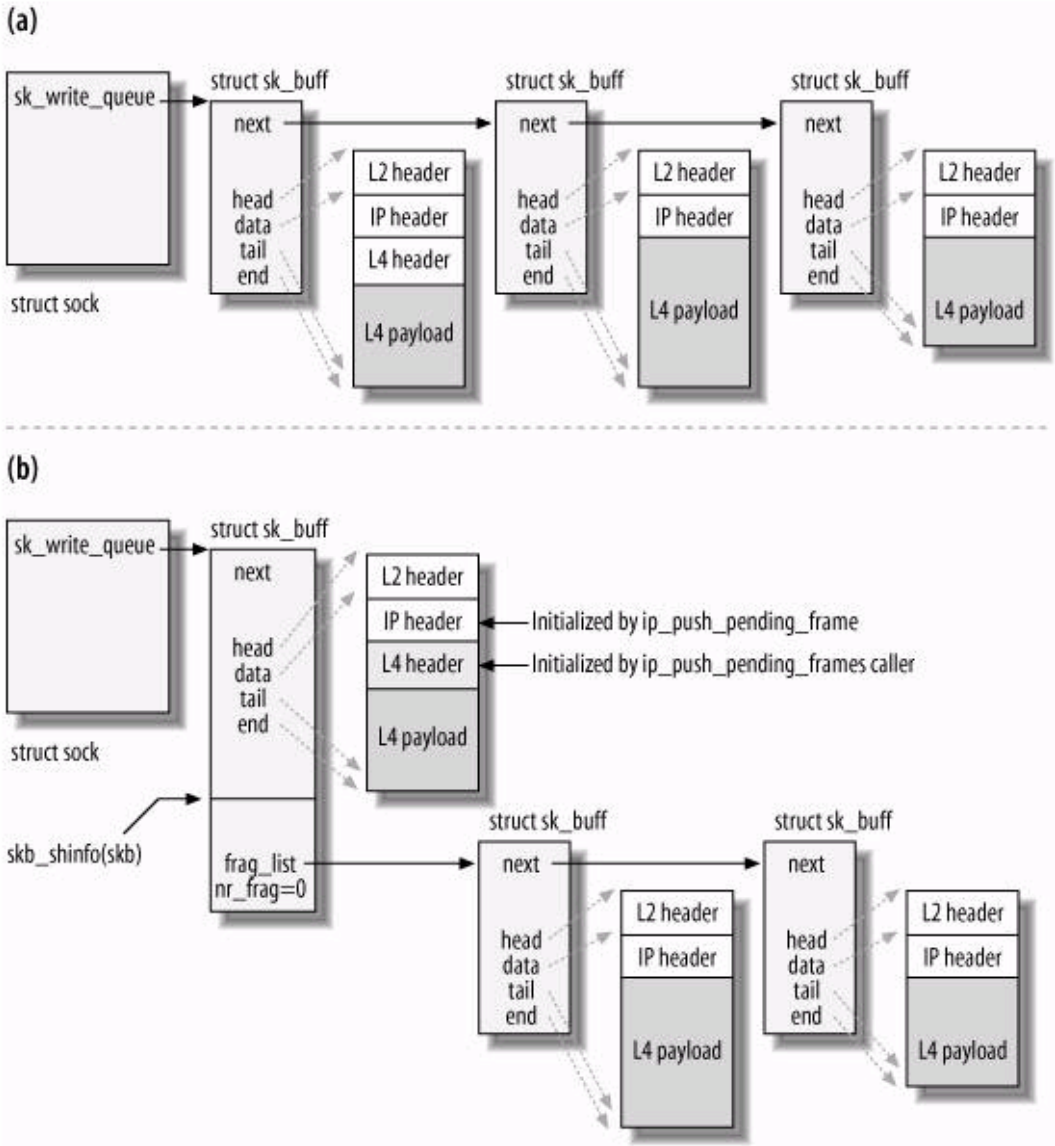
Memory Pages Shared By Fragments Using Scatter-gather IO



The ip_push_pending_frames Function

- (1) dequeue from sk_write_buffe queue to chain fragments belonging to the same packet.
- (2) build the IP header for the first fragment.
- (3) Let Netfilter get a chance to decide the fate of this packet
- (4) Invoke dst_output() using NF_HOOK.

Dequeue from sk_write_queue queue



Example – udp_sendmsg

- int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
- size_t len) {
  - ... ...
  - struct udp_opt *up = udp_sk(sk);
  - ... ...
  - int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;
  - ... ...
  - err = ip_append_data (sk, ip_generic_getfrag, msg->msg_iov,
  - ulen, sizeof(struct udphdr),
  - &ipc, rt,
  - corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
  - if (err)
    - udp_flush_pending_frames(sk);
  - else if (!corkreq)
    - err = udp_push_pending_frames(sk, up);

The Last Mile at IP Layer – ip_finish_output and ip_finish_output2

- int ip_finish_output(struct sk_buff *skb) {
  - struct net_device *dev = skb->dst->dev;
  - skb->dev = dev;
  - skb->protocol = _ _constant_htons(ETH_P_IP);
  - return NF_HOOK(PF_INET,
  - NF_IP_POST_ROUTING,
  - skb, NULL, dev,
  - ip_finish_output2);
- }
- ip_finish_output2() invokes routines of the ARP protocol

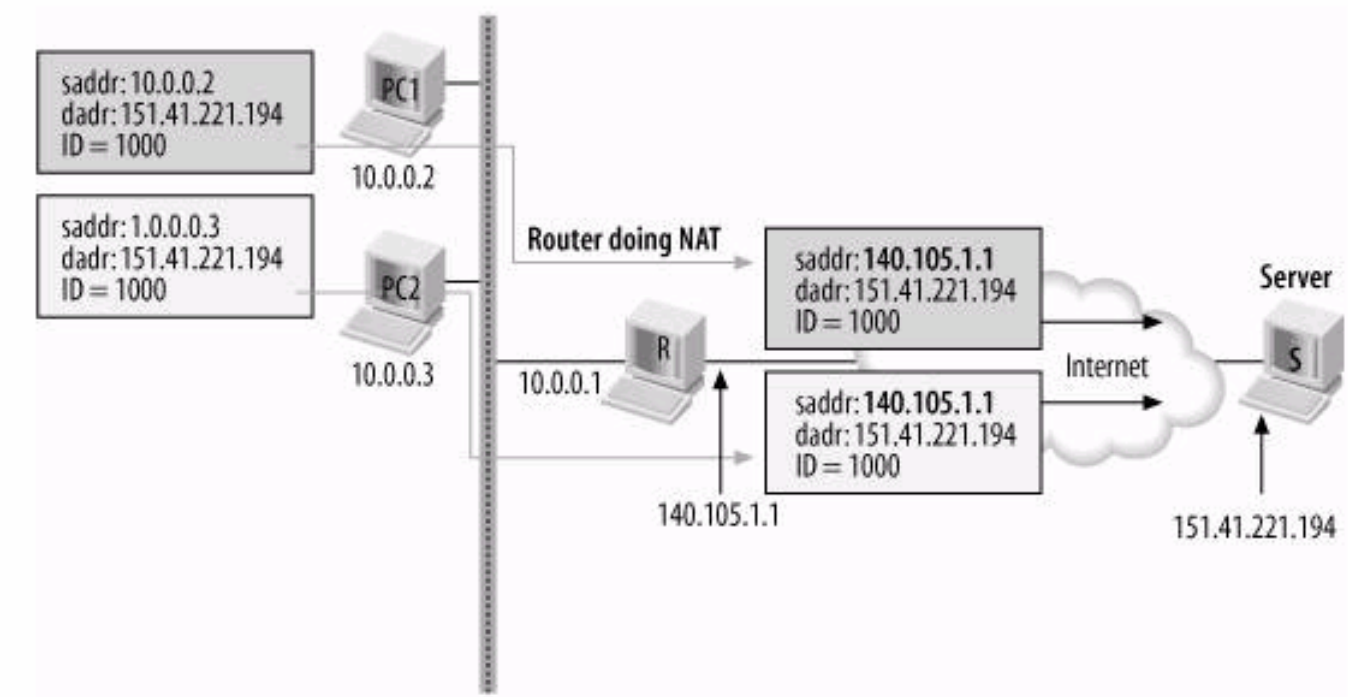
Problems of Fragmentation And Defragmentation

- Fragments are resolved and identified by three fields in IP header
  - Identification field
  - DF/MF flags
  - Fragment Offset

- Problems
  - Packet ID Generation
    - ID is 16-bit long and easy wrapped around.
    - Linux: associate IDs with different destination IP address
  - Network Address Translation

Problems Generated by NAT

- Source IP address is modified by NAT gateway
- No good solution for this problem
- IPV6 only allows the fragmentation to the end hosts.
  - Intermediate router is not allowed to do fragmentation
  - May provide a way to map packet ID, too



Path MTU Discovery

- Eliminate excessive fragmentation
- Cannot be less than 68 byte defined in RFC 1191
  - IP header: 20 bytes
  - IP options: at most 40 bytes
  - Minimum fragment length: 8 bytes
- Discovery Process
  - Try-and-error basis until receiving an ICMP FRAGMENT NEEDED message

Receiving Flow at IP Layer

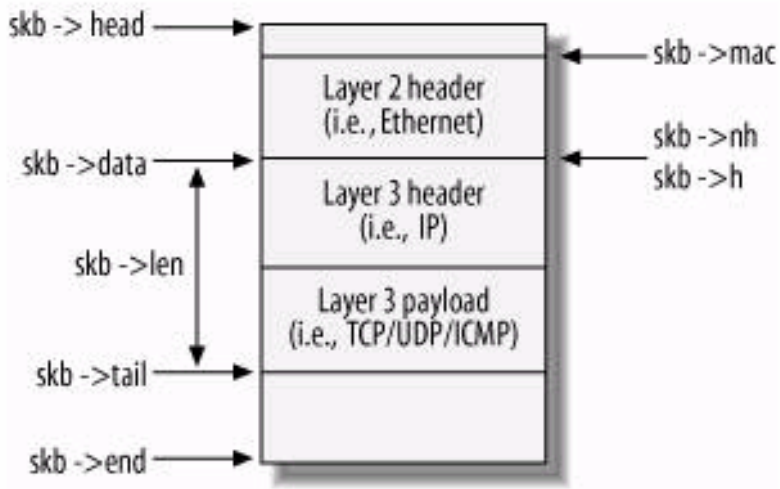
- (1) ip_rcv
- (2) ip_rcv_finish
- (3) dst_input
- (4) skb->dst_input

- (5-a) ip_local_deliver
- (6-a) ip_local_deliver_finish
- (7-a) L4 receiving handler

- (5-b) ip_forward
- (6-b) ip_forward_finish
- (7-b) ip_output

The ip_rcv function

- Invoked by netif_receive_skb() at L2
- Perform necessary sanity check
  - Packet type
  - Header size
  - Header fields: IP protocol version
  - Checksum examination
- Invoke its second-stage routine ip_rcv_finish()



The ip_rcv_finish function

- (1) find the destination of this packet:
  - if no routing entry for it: drop the packet
  - else set the dst->input routine propery
    - ip_local_deliver or ip_forward
- (2)call ip_options_compile()
  - initialize necessary structures for ip options.
- (3) Invoke dst_input
  - Call dst->input()
  - The function is either ip_local_deliver or ip_forward based on the destination of the packet.

The ip_local_deliver Function

- int ip_local_deliver (struct sk_buff *skb) {
- **/***
- ****** ***Reassemble IP fragments.***
- ****/***

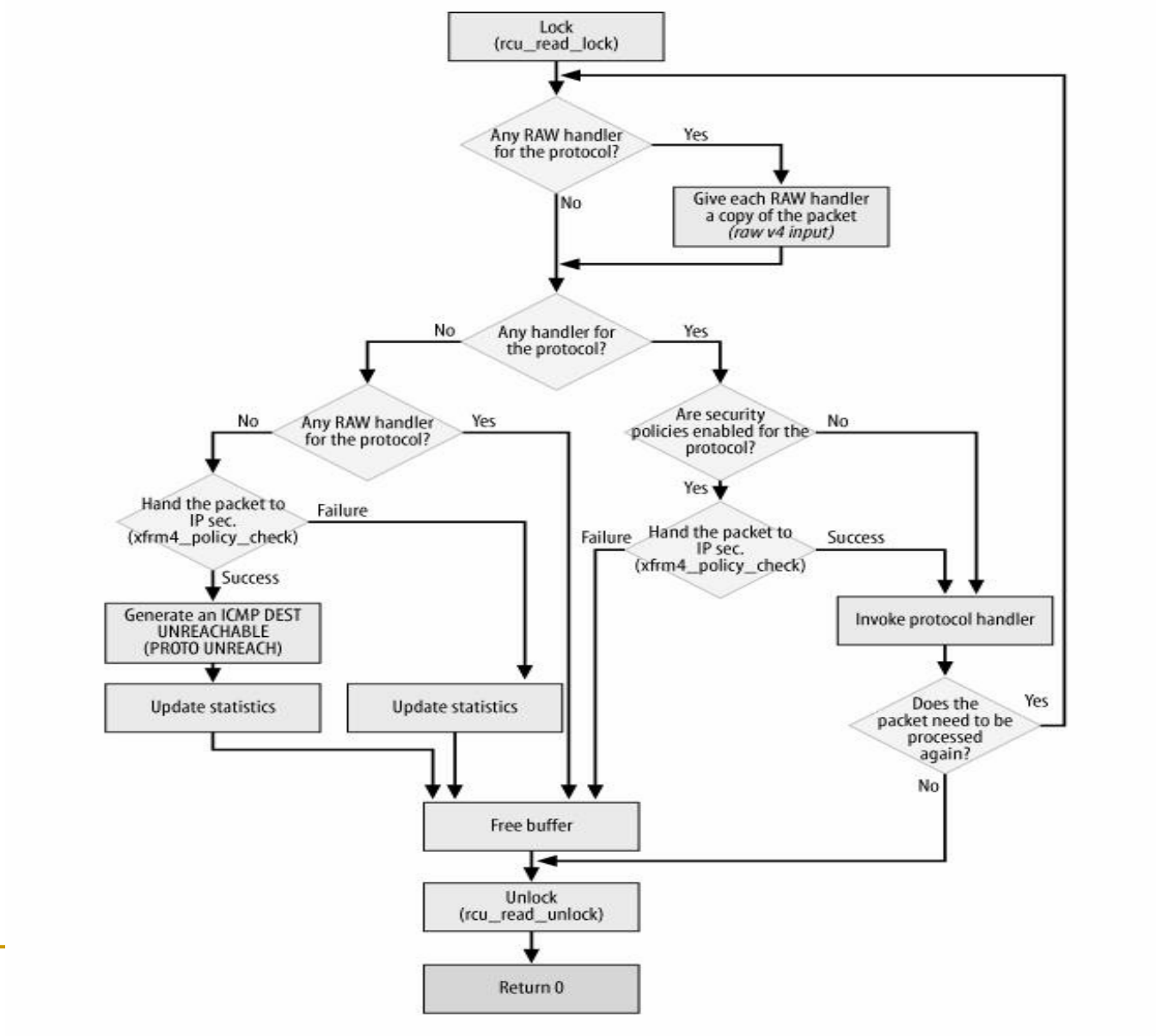
- if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
  - skb = ip_defrag (skb, IP_DEFRAG_LOCAL_DELIVER);
  - if (!skb) return 0;

- }

- return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL, ip_local_deliver_finish);

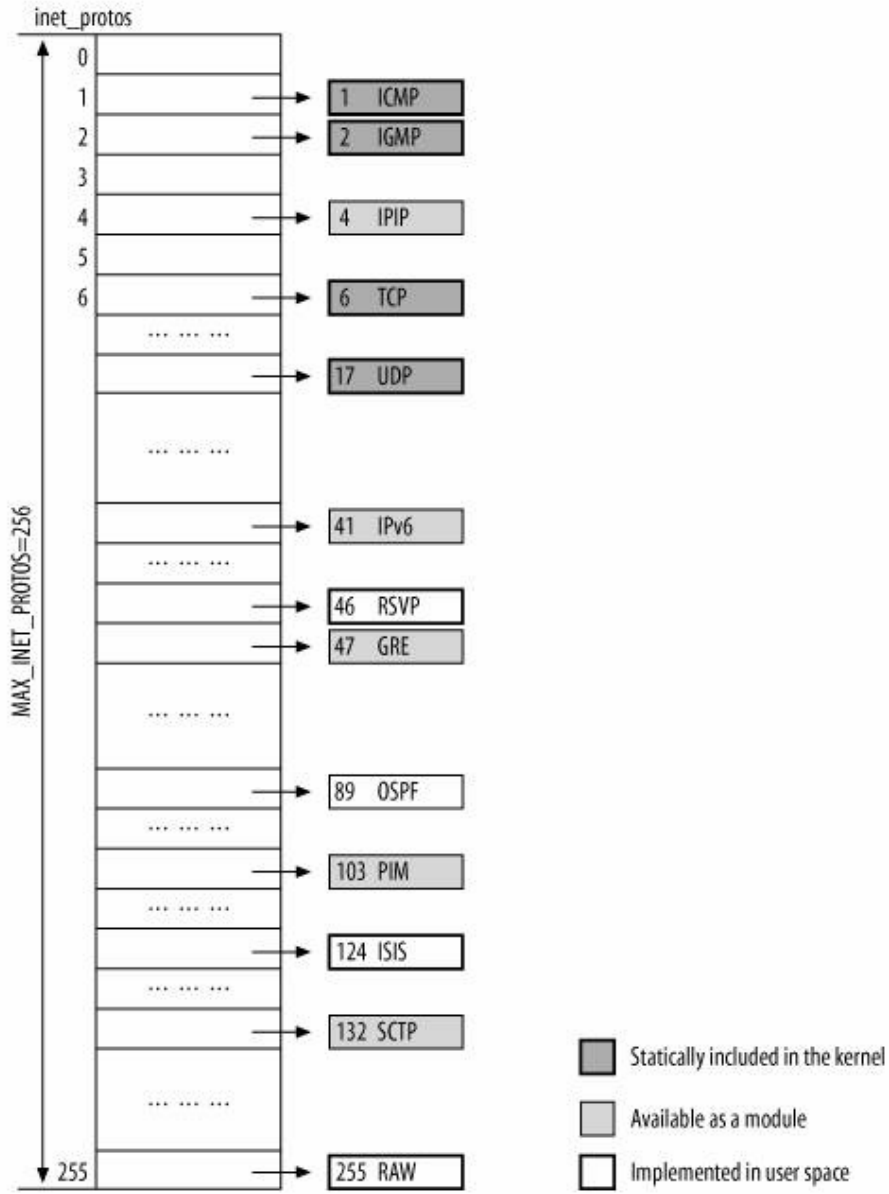
- }

The ip_local_deliver_finish Function

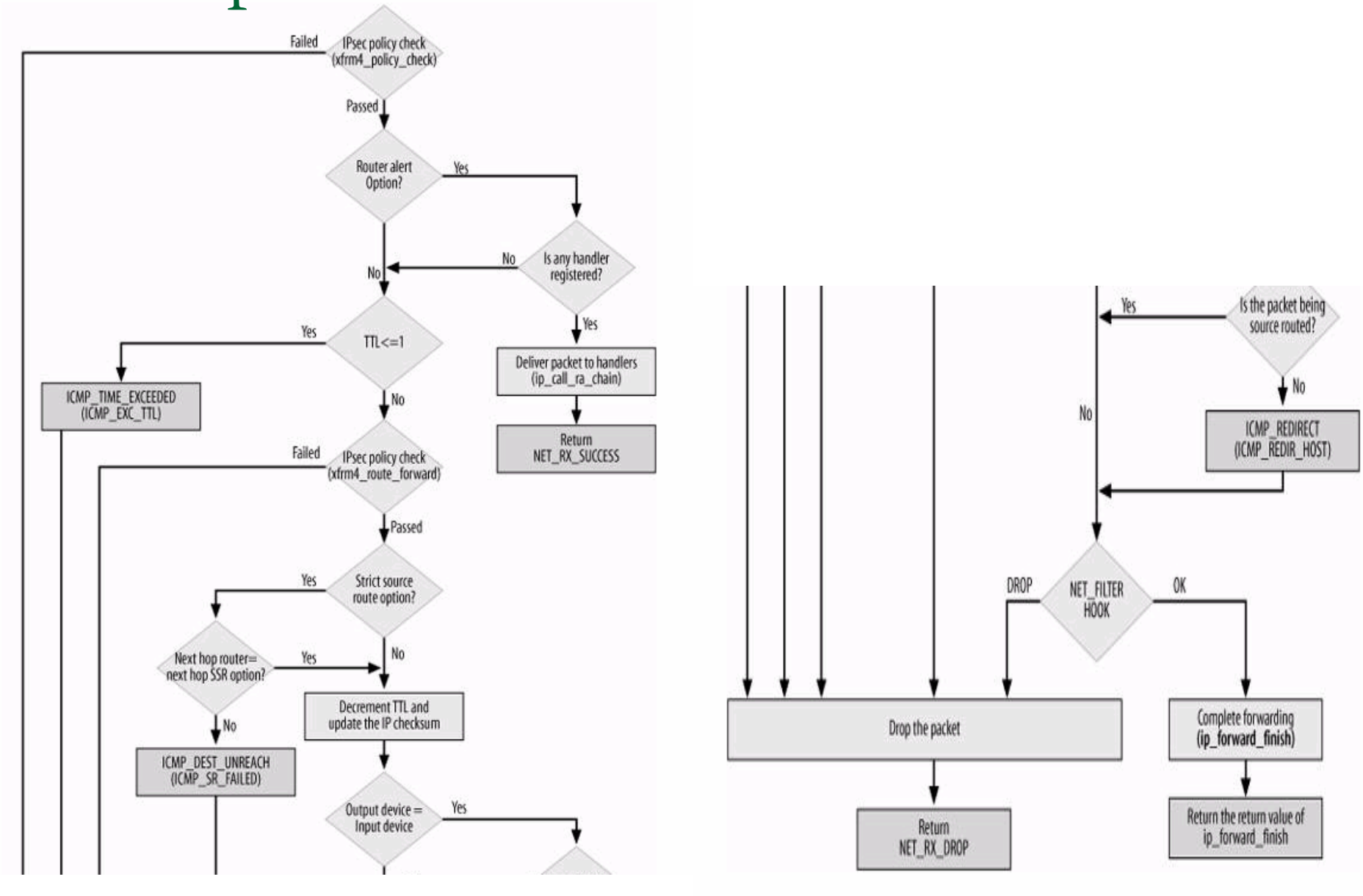


The inet_protos table

- used to reference the L4 protocol control block



The ip_forward function



The ip_forward_finish function

- static inline int ip_forward_finish (struct sk_buff *skb) {
  - struct ip_options * opt = &(IPCB(skb)->opt);
  - IP_INC_STATS_BH (IPSTATS_MIB_OUTFORWDDATAGRAMS);
  - if (unlikely(opt->optlen)
    - ip_forward_options(skb);
  - return dst_output(skb);
- }

The dst_output Function

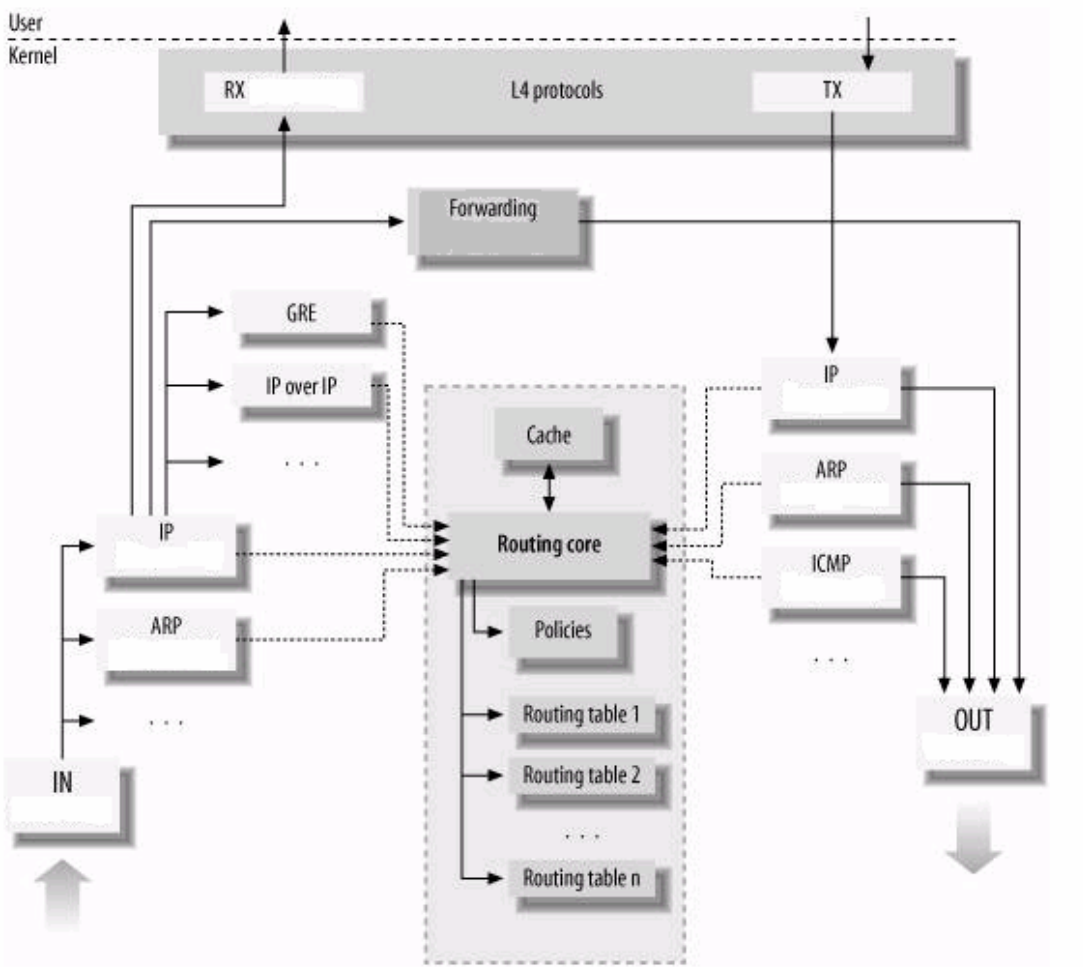
- static inline int dst_output (struct sk_buff *skb) {
  - int err;
  - for (;;) {
    - err = skb->dst->output(&skb);
    - if (likely(err = 0))
    - return err;
    - if (unlikely(err != NET_XMIT_BYPASS))
    - return err;
  - }
- }

Routing

- Goal
  - Determine if a packet reaches its destination or not.
  - Determine the next-hop if a packet has not reached its destination host.
- Efficiency
  - Routing lookup is a main bottleneck for Internet routers several years ago
  - How to fast lookup a route?
  - Tree-based algorithm
  - Hash-based algorithm (currently used by Linux)
- Dynamic change

Interaction With Other Components

- IP layer
  - For routing decision
- ARP layer
  - For sanity checking

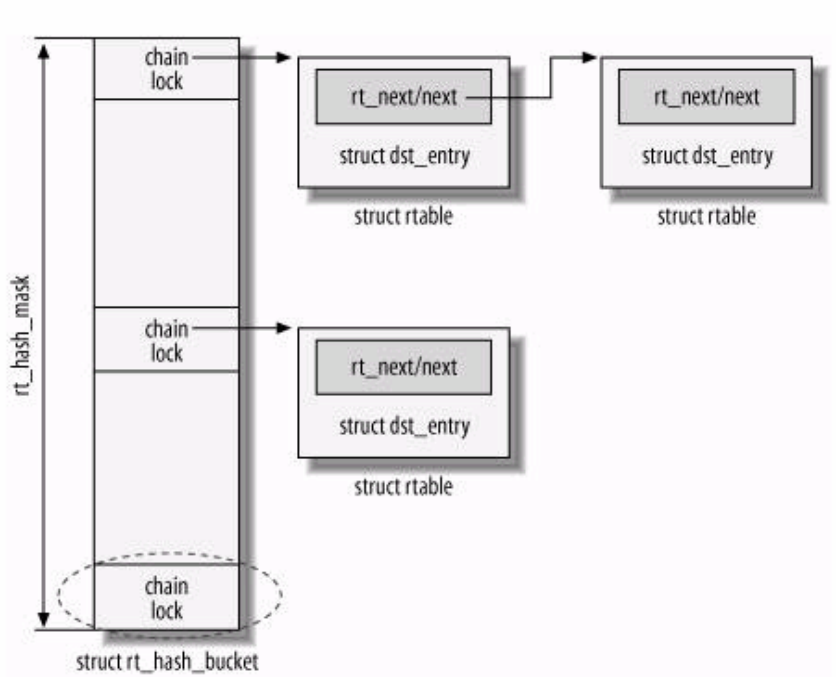


Main Components

- Route Cache
  - Each entry representing an IP address
  - Lookup method – complete matching
    - Cache hit or miss
- Route Table
  - Various structures used for fast lookup
  - Lookup method -- Longest-prefix matching
- Hash Table is highly used for both components

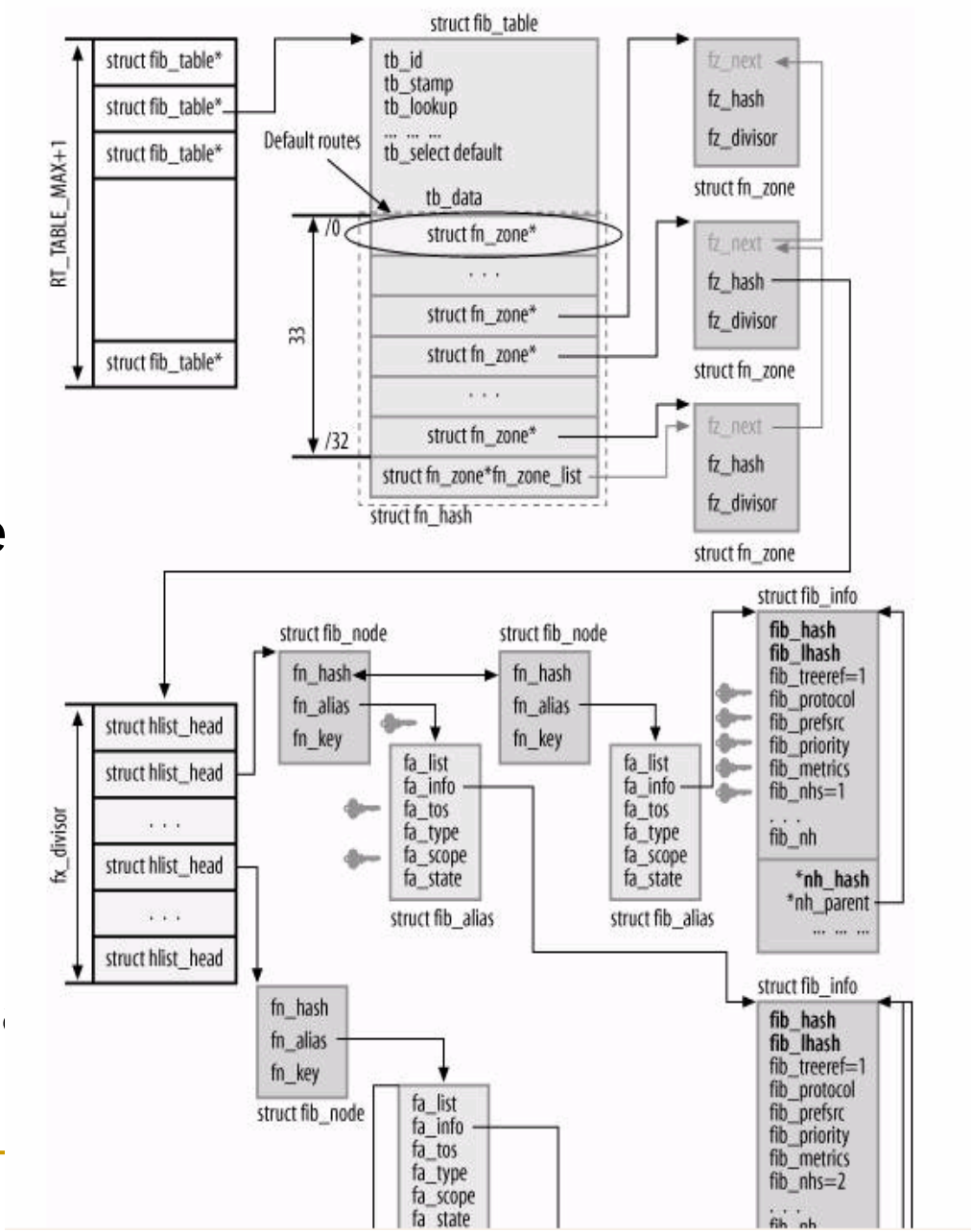
Routing Cache

- dst_entry
  - Protocol-independent information
    - Outgoing network device structure
    - Metrics
    - Expiration time



Routing Table

- fib_hash_table
  - First-level hash table
- Fib_table
  - Contains 33 vectors, each of which for route with netmasks of the same length
- Fn_zone
  - stores routes with netmasks of the same length
  - Organized routes with hash table



Routing Table .1

- Fib_node
  - Associated with routes for a subnet
  - Fib_alias stores routes to the same subnet but with different parameters.
- Fib-info
  - Main routing entry
  - indexed by two hash tables
    - Fib_info_hash
      - Mainly used for route lookup
    - Fib_info_laddrhash
      - Mainly used for fast manipulating routing entries related to local interfaces.

Dynamic Changes To Routes

- Routes for remote hosts
  - The responsibility of routing protocols
  - Netlink socket is used to manipulate routing tables
- Notification chain
  - Widely used in kernel
  - Used to notify kernel components of an occurrence for a specific event
  - E.g.
    - an “UP” and “Down” of a local NIC
    - A link failure event

Notification Chain

- Simplify the complexity of kernel codes handling various asynchronous events

- Registration
  - notifier_chain_register()
- Unregistration
  - notifier_chain_unregister()
- Notification
  - notifier_call_chain()

- Wrapper functions
  - register_inetaddr_notifier, unregister_inetaddr_notifier
  - register_inet6addr_notifier, unregister_inet6addr_notifier
  - register_netdevice_notifier, unregister_netdevice_notifier

Registration

- int notifier_chain_register(struct notifier_block **list, struct notifier_block *n) {
  - write_lock(&notifier_lock);
  - while(*list) {
    - if(n->priority > (*list)->priority) break;
    - list= &((*list)->next);
  - }
  - n->next = *list;
  - *list=n;
  - write_unlock(&notifier_lock);
  - return 0;
- }

Notification Call

- int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
  {
  - int ret = NOTIFY_DONE;
  - struct notifier_block *nb = *n;
    - while (nb) {
      - ret = nb->notifier_call(nb, val, v);
      - if (ret & NOTIFY_STOP_MASK) {
        - return ret;
      - }
      - nb = nb->next;
    - }
  - return ret;
- }

Main Description Block – Notifier Block

- struct notifier_block {
  - int (*notifier_call) (struct notifier_block *self, unsigned long, void *);
  - struct notifier_block *next;
  - int priority;
- };

An Example Used in Network Subsystem

- static struct notifier_block fib_inetaddr_notifier = {
  - .notifier_call = fib_inetaddr_event,
- };

- static struct notifier_block fib_netdev_notifier = {
  - .notifier_call = fib_netdev_event,
- };
-
- void _ _init ip_fib_init(void) {
  - ... ...
  - register_netdevice_notifier(&fib_netdev_notifier);
  - register_inetaddr_notifier(&fib_inetaddr_notifier);
- }

Device Layer (Not Discussed Here)

- Device I/O Method
  - Programmed I/O
  - DMA and Scatter-gather DMA
- Notification Model
  - Polling vs. Interrupt
- Notification Chain
- Linux Driver Model
- Netdevice structure

Reference

- Benvenuti, “Understanding Linux Network Internal,” 2006.
- Bovet and Cesati, “Understanding Linux Kernel,” 3rd edition.
- Corbet, Rubini, and Kroah-Hartman, “Linux Device Drivers,” 3rd edition.
- Linux kernel source code 2.6.11
- Prof. S.Y. Wang’s slides