Embedded Linux Part 2

Mark McDermott

Fall 2009

Agenda

- Memory Management
- File System Types
- Input and Output
- Interprocess Communication
- Kernel Time Keeping
- Networking

Memory Management

Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes
- Splits memory into 3 different zones due to hardware characteristics

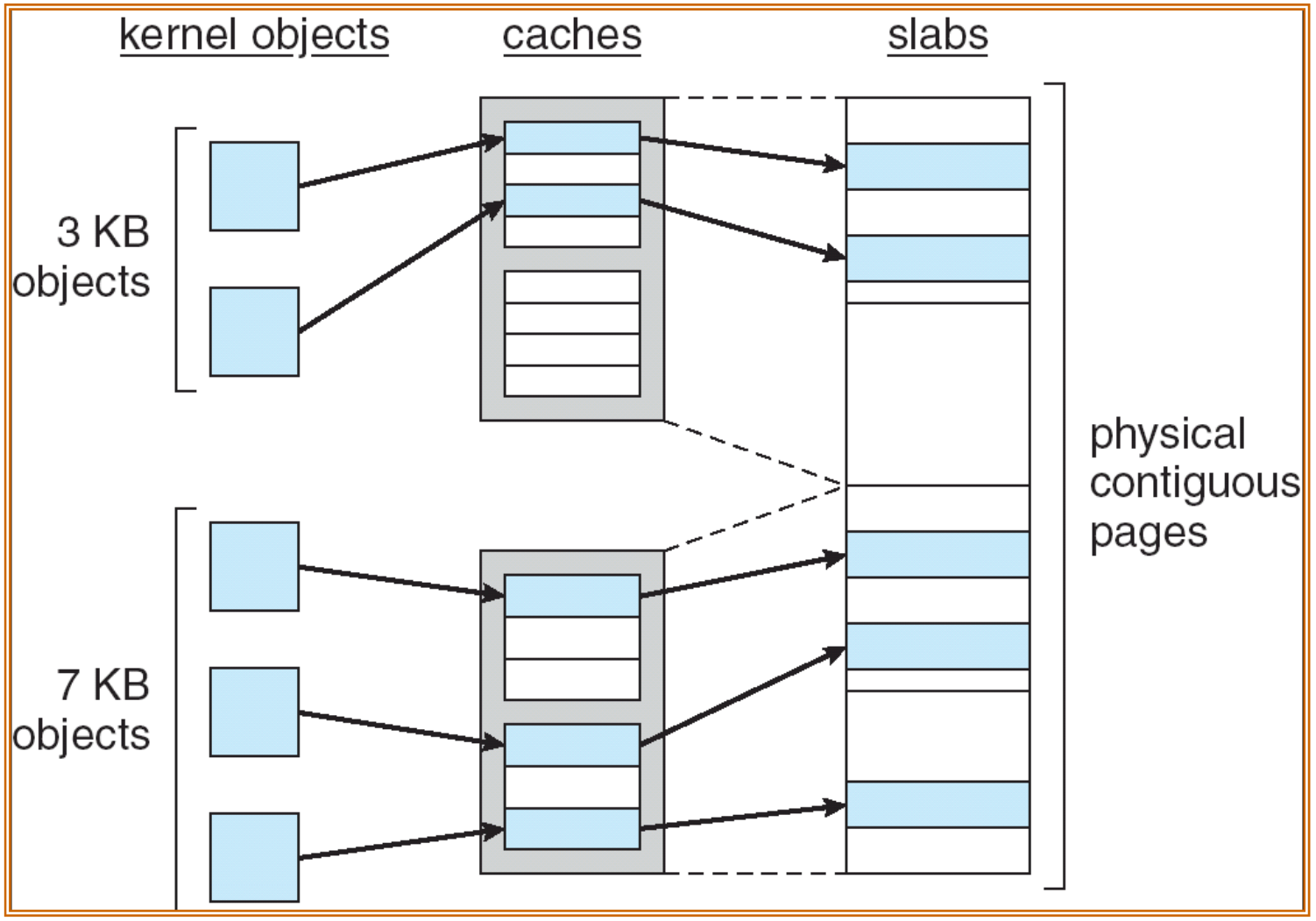
Relationship of Zones and Physical Addresses on 80x86

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

Managing Physical Memory

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request
- The allocator uses a buddy-heap algorithm to keep track of available physical pages
  - Each allocatable memory region is paired with an adjacent partner
  - Whenever two allocated partner regions are both freed up they are combined to form a larger region
  - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request
- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator)
  - Uses slab allocator for kernel memory
    - Slab allocation is a memory management algorithm that juxtaposes objects of the same type. A slab allocator will not fragment memory

Mapping to Physical memory



Virtual Memory

- The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required
- The VM manager maintains two separate views of a process's address space:
  - A logical view describing instructions concerning the layout of the address space
    - The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space
  - A physical view of each address space which is stored in the hardware page tables for the process

Virtual Memory (Cont.)

- **Virtual memory regions are characterized by:**
  - The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (demand-zero memory)
  - The region's reaction to writes (page sharing or copy-on-write)

- **The kernel creates a new virtual address space**
  - When a process runs a new program with the exec system call
  - Upon creation of a new process by the fork system call

Virtual Memory (Cont.)

- On executing a new program, the process is given a new, completely empty virtual-address space; the program-loading routines populate the address space with virtual-memory regions
- Creating a new process with fork involves creating a complete copy of the existing process's virtual address space
  - The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child
  - The parent's page tables are copied directly into the child's, with the reference count of each page covered being incremented
  - After the fork, the parent and child share the same physical pages of memory in their address spaces

Virtual Memory (Cont.)

- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else

- The VM paging system can be divided into two sections:
  - The pageout-policy algorithm decides which pages to write out to disk, and when
  - The paging mechanism actually carries out the transfer, and pages data back into physical memory as needed

Virtual Memory (Cont.)

- The Linux kernel reserves a constant, architecture-dependent region of the virtual address space of every process for its own internal use

- This kernel virtual-memory area contains two regions:
  - A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code
  - The remainder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory

File System

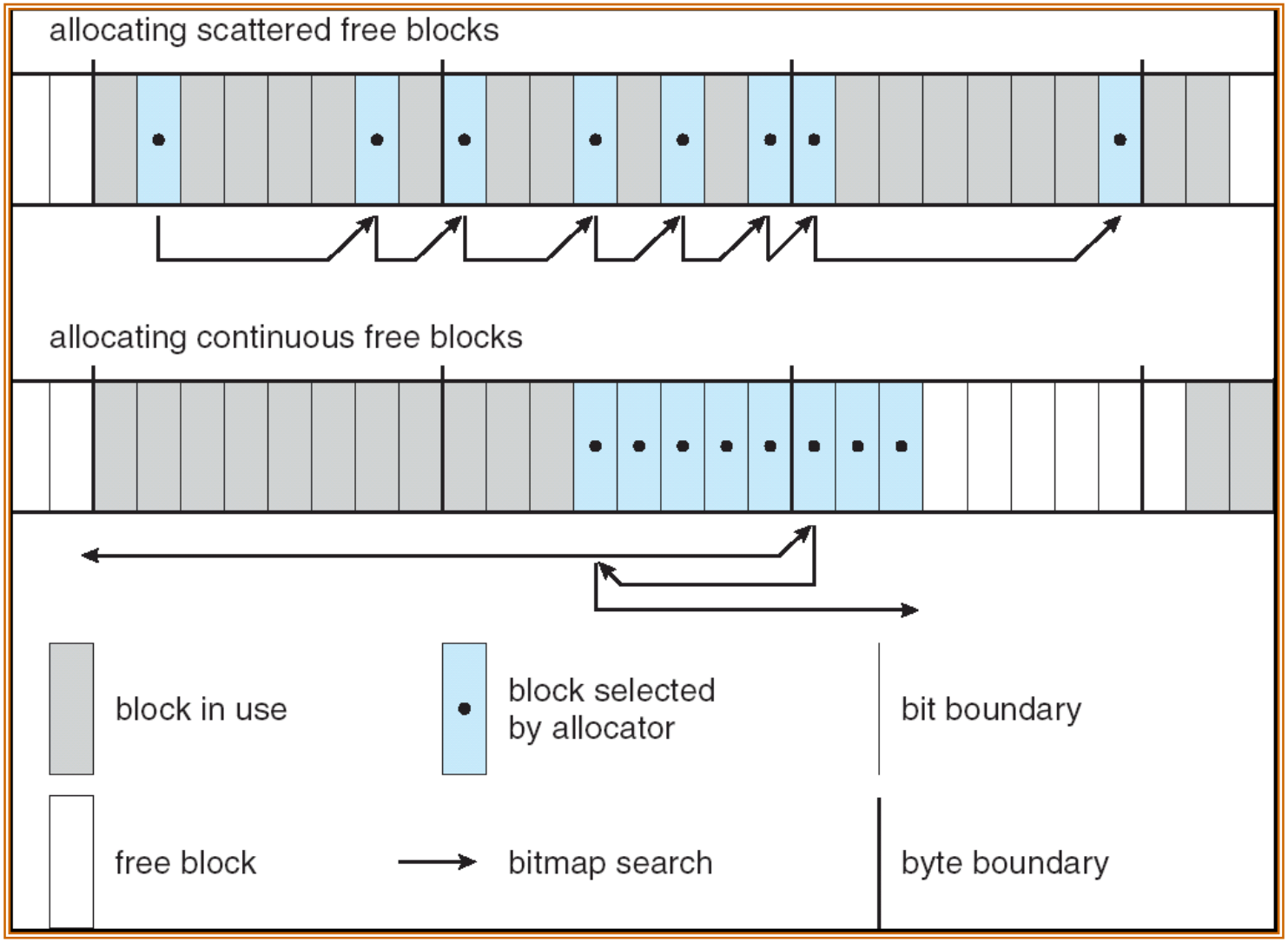
File Systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS)
- The Linux VFS is designed around object-oriented principles and is composed of two components:
  - A set of definitions that define what a file object is allowed to look like
    - The inode-object and the file-object structures represent individual files
    - the file system object represents an entire file system
  - A layer of software to manipulate those objects

The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file
- The main differences between ext2fs and ffs concern their disk allocation policies
  - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file
  - Ext2fs does not use fragments; it performs its allocations in smaller units
    - The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported
  - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation

Ext2fs Block-Allocation Policies



The Linux /proc File System

- The `/proc` file system does not store data, rather, its contents are computed on demand according to user file I/O requests
- `/proc` must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains
  - It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode
  - When data is read from one of these files, proc collects the appropriate information, formats it into text form and places it into the requesting process's read buffer

Input Output

Input and Output

- **The Linux device-oriented file system accesses disk storage through two caches:**
  - Data is cached in the page cache, which is unified with the virtual memory system
  - Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block
- **Linux splits all devices into three classes:**
  - block devices allow random access to completely independent, fixed size blocks of data
  - character devices include most other devices; they don't need to support the functionality of regular files
  - network devices are interfaced via the kernel's networking subsystem

Block Devices

- Provide the main interface to all disk devices in a system
- The *block buffer* cache serves two main purposes:
  - it acts as a pool of buffers for active I/O
  - it serves as a cache for completed I/O
- The *request manager* manages the reading and writing of buffer contents to and from a block device driver

Character Devices

- A “character device” driver which does not offer random access to fixed blocks of data
- A character device driver must register a set of functions which implement the driver’s various file I/O operations
- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device
- The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface

Interprocess Communications

Interprocess Communication

- Like UNIX, Linux informs processes that an event has occurred via signals
- There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process
- The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and wait.queue structures

Passing Data Between Processes

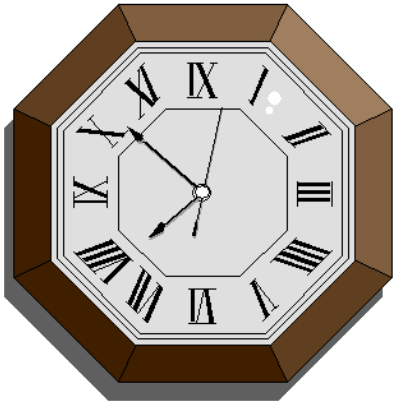
- The pipe mechanism allows a child process to inherit a communication channel to its parent, data written to one end of the pipe can be read at the other

- Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space

- To obtain synchronization, however, shared memory must be used in conjunction with another Interprocess-communication mechanism

The Linux Kernel: The Flow of Time

"What time is it?"



- Need timing measurements to:
  - Keep track of current time and date for use by e.g. gettimeofday().
    ```
    (void)gettimeofday(&start_timestamp, NULL);
    ```
  - Maintain timers that notify the kernel or a user program that an interval of time has elapsed.
- Timing measurements are performed by several hardware circuits, based on fixed frequency oscillators and counters.

Hardware Clocks

- **Real-Time Clock (RTC):**
  - Often integrated with CMOS RAM and Battery on separate chip from CPU: e.g., Motorola MC146818.
  - Issues periodic interrupts on IRQ line (IRQ 8) at programmed frequency (e.g., 2-8192 Hz).
  - In Linux, used to derive time and date.
  - Kernel accesses RTC through 0x70 and 0x71 I/O ports.

Timestamp Counter (TSC)

- Intel Pentium (and up), AMD K6 etc incorporate a TSC.
- Processor's CLK pin receives a signal from an external oscillator e.g., 400 MHz crystal.
- TSC register is incremented at each clock signal.
- Using `rdtsc` assembly instruction can obtain 64-bit timing value.
- Most accurate timing method on above platforms.

The "PIT"s

- Programmable Interrupt Timers (PITs):
  - e.g., 8254 chip.
- PIT issues *timer interrupts* at programmed frequency.
- In Linux, PC-based 8254 is programmed to interrupt Hz (=100) times per second on IRQ 0.
  - `Hz` defined in `<linux/param.h>`
  - PIT is accessed on ports `0x40-0x43`.
- Provides the system "heartbeat" or "clock tick".

"This'll only take a jiffy"

- `jiffies` is incremented every timer interrupt.
  - Number of clock ticks since OS was booted.
- Scheduling and preemption done at granularities of time-slices calculated in units of jiffies.

Timer Interrupt Handler

- **Every timer interrupt:**
  - Update jiffies.
  - Update time and date (in secs & μsecs since 1970).
    - There has been 1.2 billion seconds since 1970
  - Determine how long a process has been executing and preempt it, if it finishes its allocated timeslice.
  - Update resource usage statistics.
  - Invoke functions for elapsed interval timers.

PIT Interrupt Service Routine

- Signal on IRQ 0 is generated:
- `timer_interrupt()` is invoked w/ interrupts disabled (`SA_INTERRUPT` flag is set to denote this).
- `do_timer()` is ultimately executed:
  - Simply increments `jiffies` & allocates other tasks to "bottom half handlers".
  - Bottom half (bh) handlers update time and date, statistics, execute fns after specific elapsed intervals and invoke `schedule()` if necessary, for rescheduling processes.

Updating Time and Date

- `lost_ticks` (`lost_ticks_system`) store total (system) "ticks" since update to `xtime`, which stores *approximate* current time. This is needed since bh handlers run at convenient time and we need to keep track of when exactly they run to accurately update date & time.
- `TIMER_BH` refers to the queue of bottom halves invoked as a consequence of `do_timer()`.

Task Queues

- Often necessary to schedule kernel tasks at a later time without using interrupts.
  - Solution: Task Queues and kernel timers.
- A task queue is a list of *bottom half handlers*, each represented by a function pointer and argument.
- From `<linux/tqueue.h>`:

```
struct tq_struct {
    struct tq_struct *next;
    int sync; /* always 0 initially. */
    void (*routine)(void *);
    void *data;
}
```

Predefined Task Queues

- `tq_scheduler`: bottom half tasks in this queue are executed *whenever the scheduler runs*.
  - Both scheduler and bottom halves run in context of process being scheduled out.
- `tq_timer`: executed every timer tick at "interrupt time".
- `tq_immediate`: executed either on return from syscall or when scheduler is run.

Useful Task Queue Functions

- `void queue_task (struct tq_struct *task, task_queue *list);`
  - Each queued task is removed from its queue after it is executed.
  - A task must be re-queued if needed repeatedly.
- `void run_task_queue (task_queue *list);`
  - Not needed unless custom task queues are implemented.
  - Fn is called by `do_bottom_half()` for predefined task queues.

Task Queue Example

```
struct wait_queue *waitq=null;

void wakeup_function(void *data) {
        wakeup_interruptible(&waitq);
}

void foo() {
        struct tq_struct bh;
        bh.next=null;
        bh.sync=0;
        bh.routine=wakeup_function;
        bh.data=(void *)some_data;
        queue_task(&bh,&tq_scheduler);
        interruptible_sleep_on(&waitq);
}
```

Kernel Timers

- Like task queues but timer bottom halves execute at predefined times.
- From `<linux/timer.h>`:

```
struct timer_list {
    struct timer_list *next;
    struct timer_list *prev;
    unsigned long expires; /* timeout in jiffies. */
    unsigned long data;
    void (*function)(unsigned long);
}
```

Useful Kernel Timer Functions

- `void init_timer(struct timer_list *timer);`
  - Zeroes `prev` & `next` pointers in doubly-linked timer queue.
- `void add_timer(struct timer_list *timer);`
  - Adds timer bottom half to kernel timer queue.
- `int del_timer(struct timer_list *timer);`
  - Removes timer before it expires.

Kernel Timer Example

```
struct wait_queue *waitq=null;

void wakeup_function(unsigned long data) {
        wakeup_interruptible(&waitq);
}

void foo() {
        struct timer_list bh;
        init_timer(&bh);
        bh.function=wakeup_function;
        bh.data=(unsigned long)some_data;
        bh.expires=jiffies+10*HZ; /* in 10 seconds. */
        add_timer(&bh);
        interruptible_sleep_on(&waitq);
}
```

Networking

Network Structure

- **Networking is a key area of functionality for Linux.**
  - It supports the standard Internet protocols for UNIX to UNIX communications
  - It also implements protocols native to nonUNIX operating systems, in particular, protocols used on PC networks, such as Appletalk and IPX

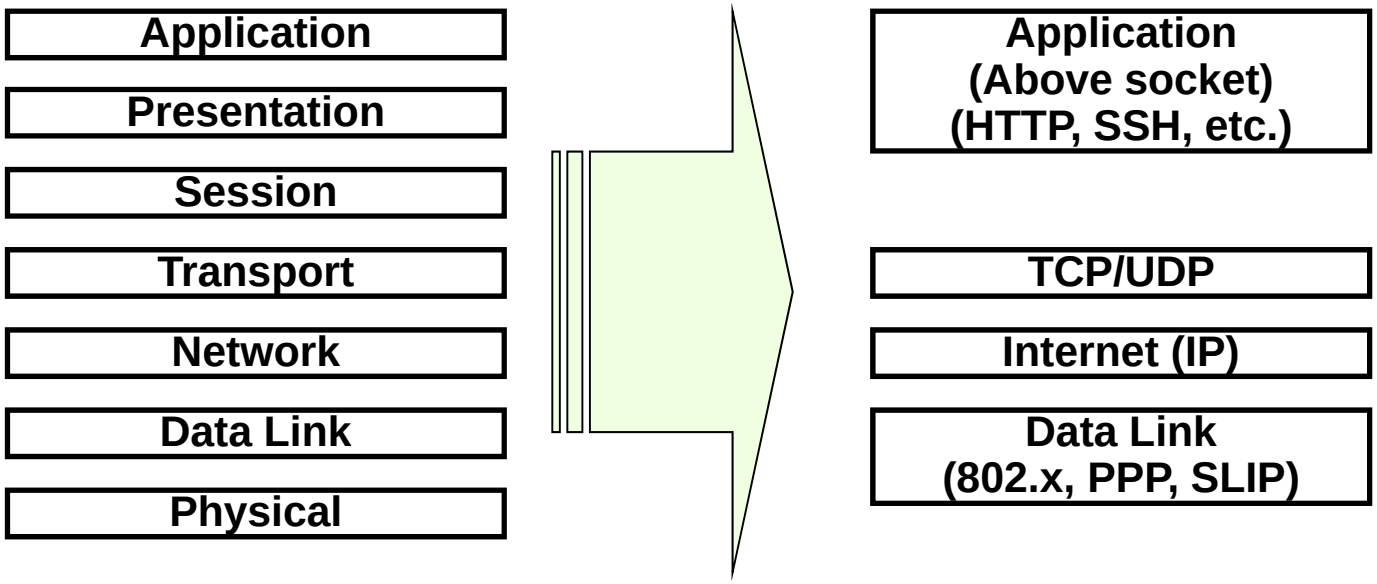
- **Internally, networking in the Linux kernel is implemented by three layers of software:**
  - The socket interface
  - Protocol drivers
  - Network device drivers

Network Structure (Cont.)

- The most important set of protocols in the Linux networking system is the internet protocol suite
  - It implements routing between different hosts anywhere on the network
  - On top of the routing protocol are built the UDP, TCP and ICMP protocols

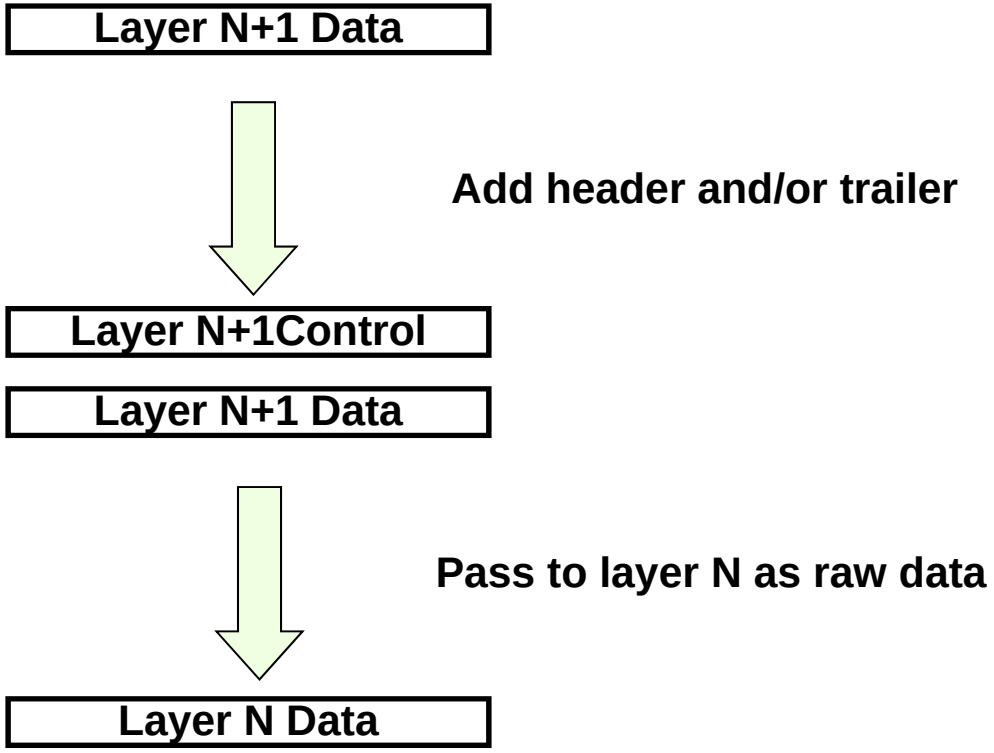
OSI Model

- The Linux kernel adheres closely to the OSI 7-layer networking model



OSI Model (Interplay)

Layers generally interact in the same manner, no matter where placed



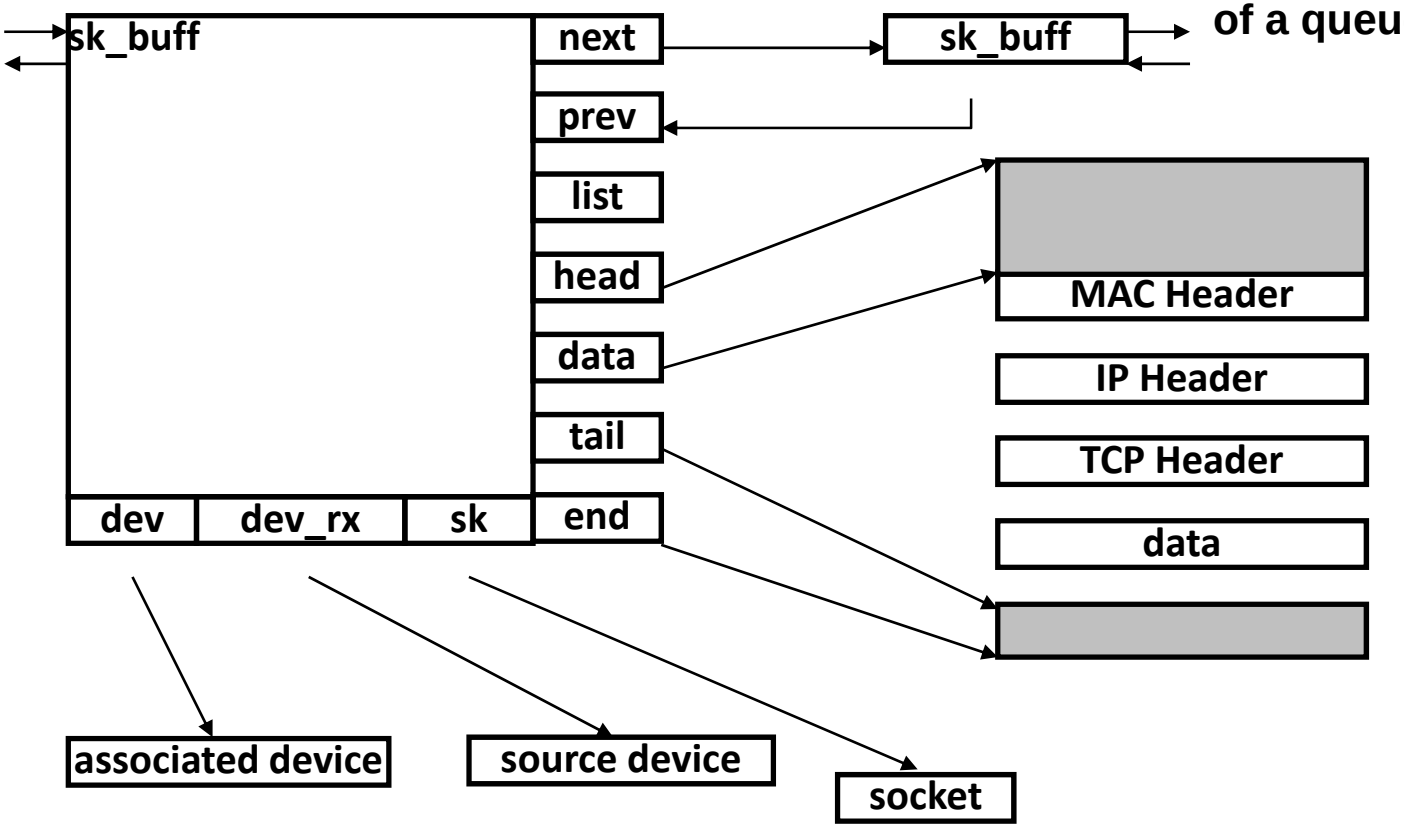
Socket Buffer

When discussing the data path through the linux kernel, the data being passed is stored in `sk_buff` structures (socket buffer).

Packet Data
Management Information

- The `sk_buff` is first created incomplete, then filled in during passage through the kernel, both for received packets and for sent packets.
- Packet data is normally never copied. We just pass around pointers to the `sk_buff` and change structure members

Socket Buffer



struct sk_buff is defined in:
include/linux/skbuff.h

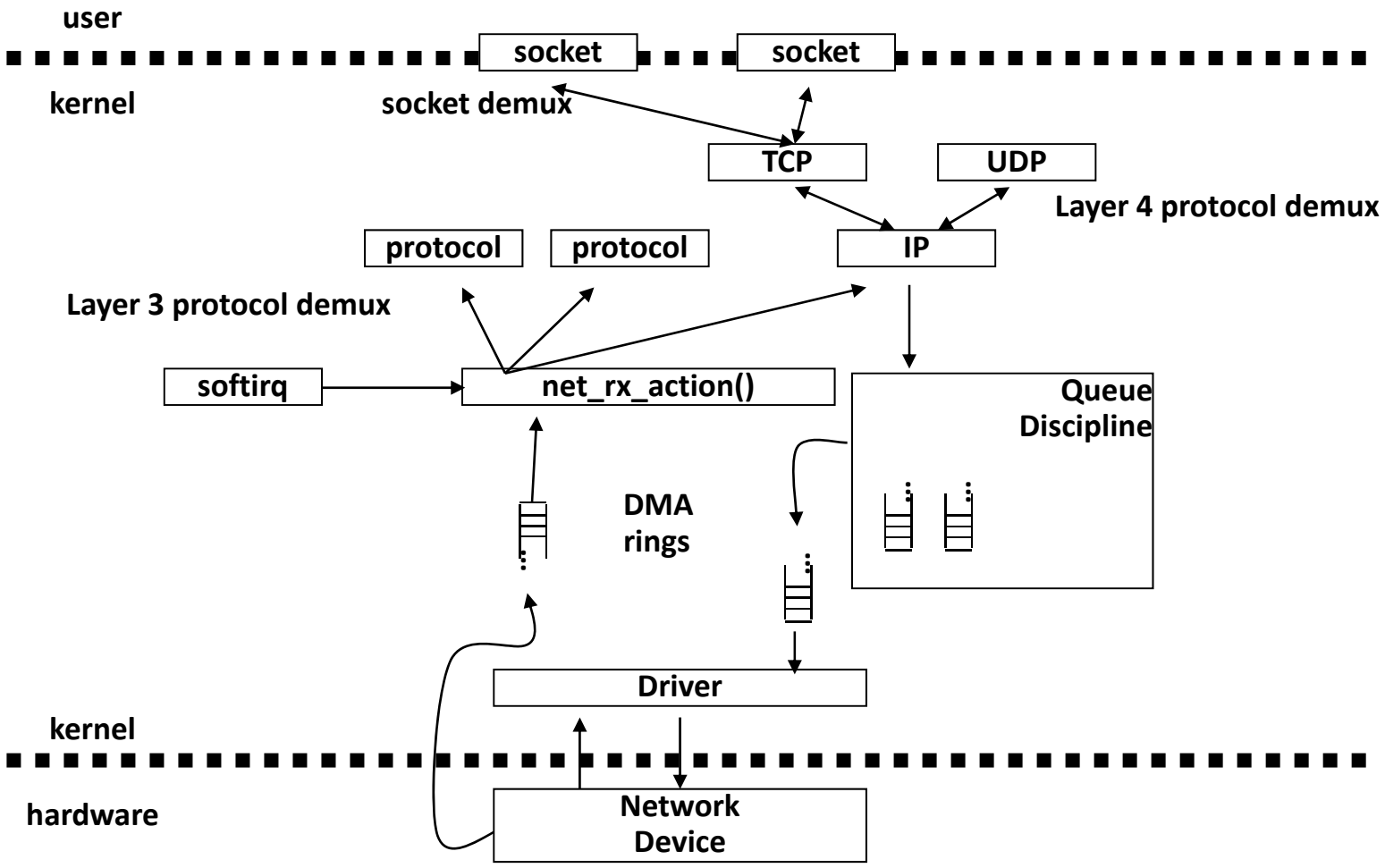
Socket Buffer

`sk_buff` features:

- Reference counts for cloned buffers
- Separate allocation pool and support
- Functions for manipulating the data space
- *Very* "feature-rich" – this is a very complex, detailed structure, encapsulating information from protocols at multiple layers

There are also numerous support functions for queues of `sk_buff`'s.

Data Path Overview



OSI Layers 1&2 – Data Link

- The code presented resides mostly in the following files:

include/linux/netdevice.h

net/core/skbuff.c

net/core/dev.c

net/dev/core.c

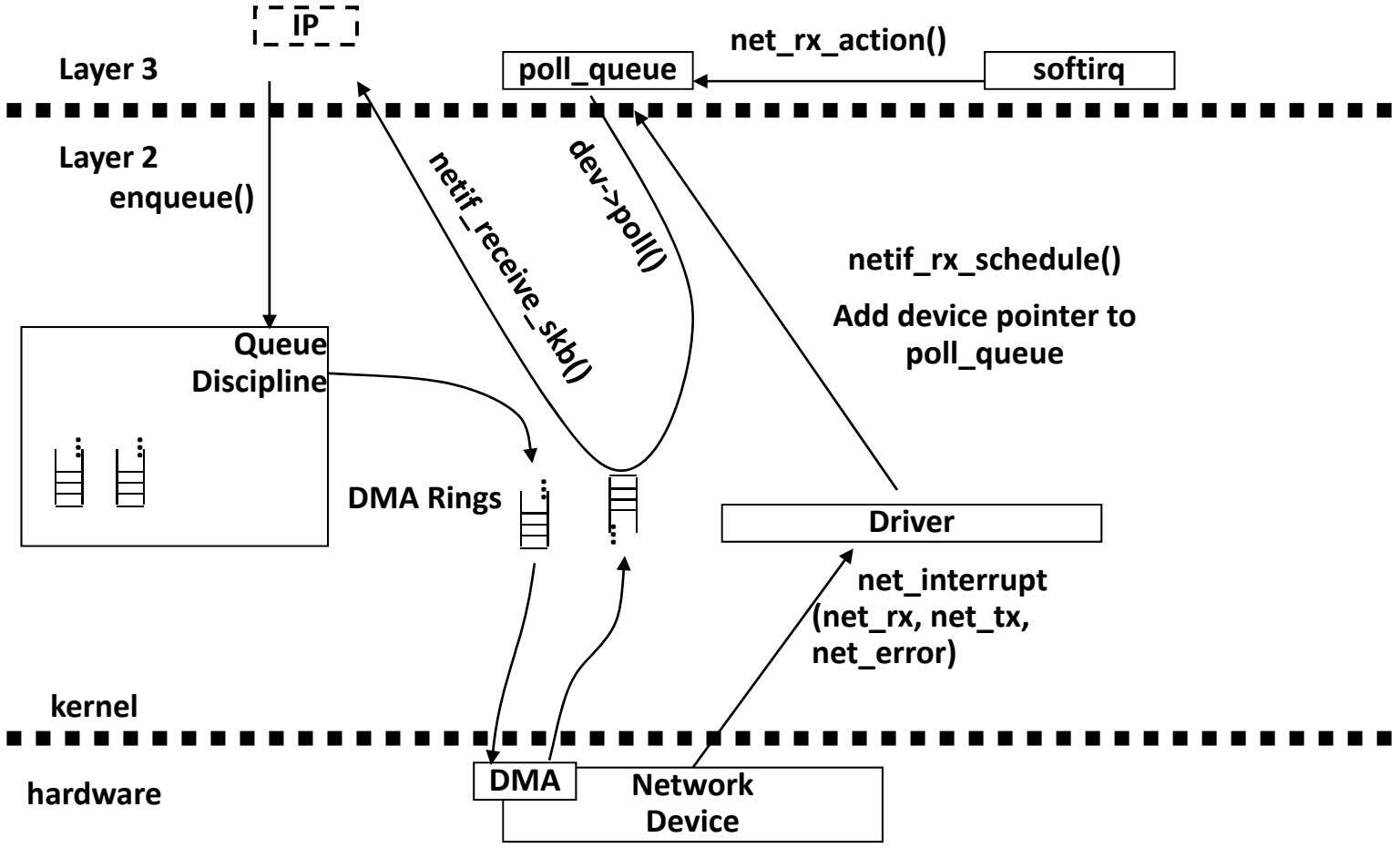
arch/i386/irq.c

drivers/net/net_init.c

net/sched/sch_generic.c

net/ethernet/eth.c (for layer 3 demux)

Data Link – Data Path



Data Link – Features

- NAPI
  - Old API would reach interrupt livelock under 60 MBps
  - New API ensures earliest possible drop under overload
    - Packet received at NIC
    - NIC copies to DMA ring (struct skbuff *rx_ring[])
    - NIC raises interrupt via netif_rx_schedule()
    - Further interrupts are blocked
    - Clock-based softirq calls softirq_rx(), which calls dev->poll()
    - dev->poll() calls netif_receive_skb(), which does protocol demux (usually calling ip_rcv())
  - Backward compatibility for non-DMA interfaces maintained
    - All legacy devices use the same backlog (equivalent to DMA ring)
    - Backlog queue is treated just like all other modern devices
- Per-CPU poll_list of devices to poll
  - Ensures no packet re-ordering necessary
- No memory copies in kernel – packet stays in the sk_buff at the same memory location until passed to user space

OSI Layer 3: Internet

The code presented resides mostly in the following files:

net/ipv4/ip_input.c – process packet arrivals

net/ipv4/ip_output.c – process packet departures

net/ipv4/ip_forward.c – process packet traversal

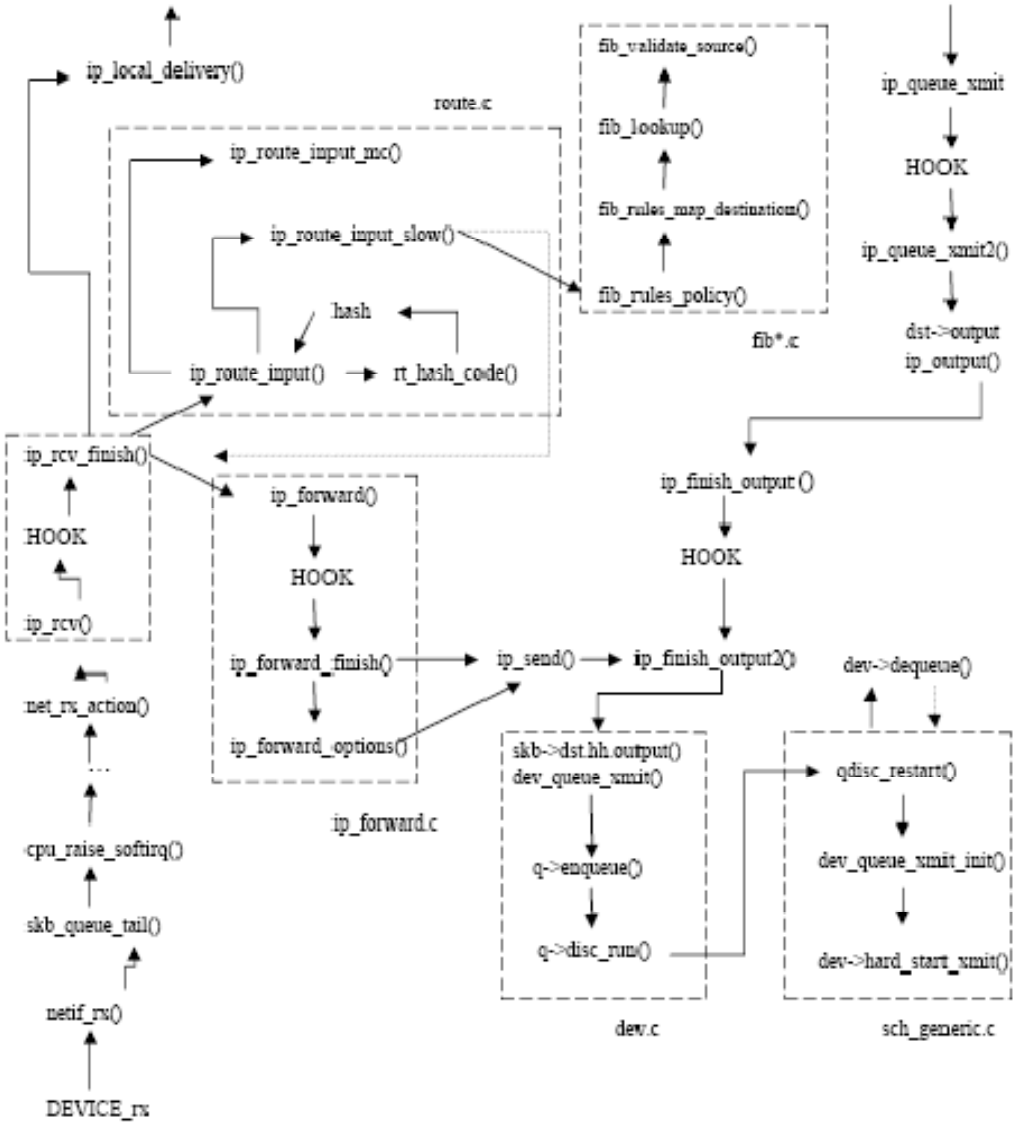
net/ipv4/ip_fragment.c – IP packet fragmentation

net/ipv4/ip_options.c – IP options

net/ipv4/ipmr.c – IP multicast

net/ipv4/ipip.c – IP over IP, also good virtual interface example

Internet: Data Path



From: "A Map of the Networking Code in the Linux Kernel"

Internet: Features

- Netfilter hooks in many places
  - INPUT, OUTPUT, FORWARD (iptables)
  - NF_IP_PRE_ROUTING – ip_rcv()
  - NF_IP_LOCAL_IN – ip_local_deliver()
  - NF_IP_FORWARD – ip_forward()
  - NF_IP_LOCAL_OUT – ip_build_and_send_pkt()
  - NF_IP_POST_ROUTING – ip_finish_output()
- Connection tracking in IPv4, not in TCP/UDP/ICMP.
  - Used for NAT, which must maintain connection state in violation of OSI Layering
  - Can also gather statistics for networking usage, but all of this functionality comes from the netfilter module

Socket Structure and System Call Mapping

The following files are useful:

- include/linux/net.h
- net/socket.c

There are two significant data structures involved, the socket and the net_proto_family. Both involve arrays of function pointers to handle each system call type that is relevant.

System Call: socket

- From user space, an application calls socket(family,type, protocol)
- The kernel calls sys_socket(), which calls sock_create()
- sock_create references net_families[family], an array of network protocol families, to find the corresponding protocol family, loading any modules necessary on the fly.
  - If the module is loaded, it is loaded as "net_pf_<num>", where the protocol family number is used directly in the string. For TCP, the family is PF_INET (was: AF_INET), and the type is SOCK_STREAM
  - Note: linux has a hard limit of 32 protocol families. (These include PF_INET, PF_PACKET, PF_NETLNK, PF_INET6, etc.)
  - Layer 4 Protocols are registered in inet_add_protocol() (include/net/protocol.h), and socket interfaces are registered by inet_register_protosw(). Raw IP datagram sockets are registered like any other Layer 4 protocol.

System Call: socket (cont)

- Once the correct family is found, sock_create allocates an empty socket, obtains a mutex, and calls net_families[family]->create(). This is protocol-specific, and filles in the socket structure. The socket structure includes another function array, ops, which maps all system calls valid on file descriptors.
- sys_socket() calls sock_map_fd() to map the new socket to a file descriptor, and returns it.

Other socket System Calls

Subsequent socket system calls are passed to the appropriate function in socket->ops[]. These include (exhaustive list):

- release
- bind
- connect
- socketpair
- accept
- getname
- poll
- ioctl
- listen
- shutdown
- setsockopt
- getsockopt
- sendmsg
- recvmsg
- mmap
- sendpage

Technically, Linux offers only one socket system call, sys_socket-call(), which multiplexes to all other system calls via the first parameter. This means that socket-based protocols could provide new and different system calls via a library and a mux, although this is never done in practice.