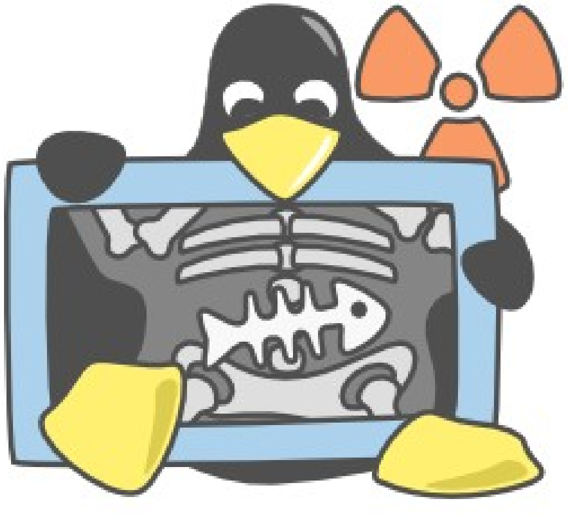
Linux Kernel Networking

Raoul Rivas



Kernel vs Application Programming

- No memory protection
  - We share memory with devices, scheduler
- Sometimes no preemption
  - Can hog the CPU
  - Concurrency is difficult
- No libraries
  - Printf, fopen
- No security descriptors
- In Linux no access to files
- Direct access to hardware

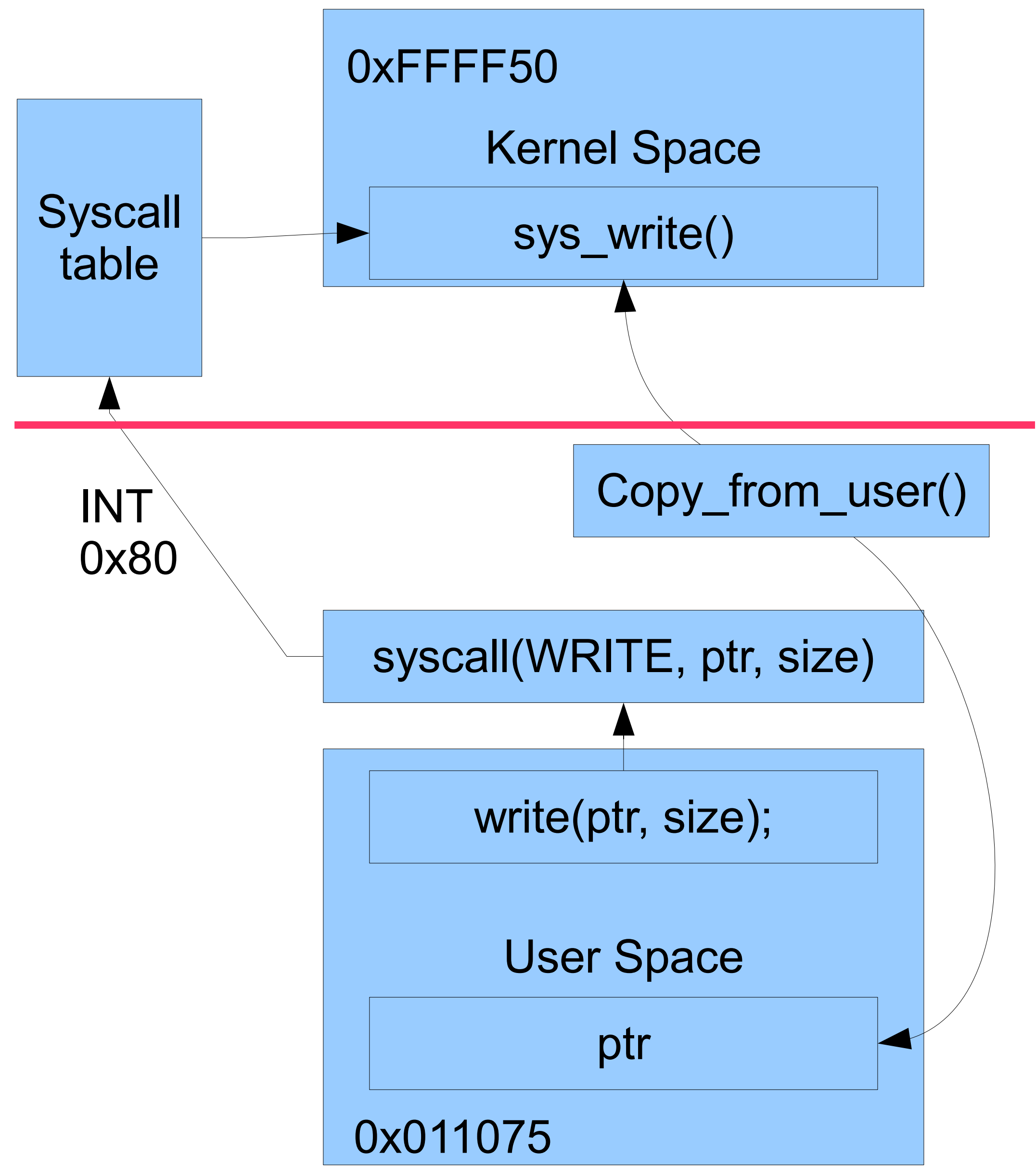
- Memory Protection
  - Segmentation Fault
- Preemption
  - Scheduling isn't our responsibility
- Signals (Control-C)
- Libraries
- Security Descriptors
- In Linux everything is a file descriptor
- Access to hardware as files

Outline

- User Space and Kernel Space
- Running Context in the Kernel
- Locking
- Deferring Work
- Linux Network Architecture
- Sockets, Families and Protocols
- Packet Creation
- Fragmentation and Routing
- Data Link Layer and Packet Scheduling
- High Performance Networking

System Calls

- A system call is an interrupt
  - syscall(number, arguments)
- The kernel runs in a different address space
- Data must be copied back and forth
  - copy_to_user(), copy_from_user()
- Never directly dereference any pointer from user space



Context

	Kernel Context	Process Context	Interrupt Context
Preemptible	Yes	Yes	No
PID	Itself	Application PID	No
Can Sleep?	Yes	Yes	No
Example	Kernel Thread	System Call	Timer Interrupt

- Context: Entity whom the kernel is running code on behalf of
- Process context and Kernel Context are preemptible
- Interrupts cannot sleep and should be small
- They are all concurrent
- Process context and Kernel context have a PID:
  - Struct task_struct* current

Race Conditions

- Process context, Kernel Context and Interrupts run concurrently
- How to protect critical zones from race conditions?
  - Spinlocks
  - Mutex
  - Semaphores
  - Reader-Writer Locks (Mutex, Semaphores)
  - Reader-Writer Spinlocks

THE SPINLOCK SPINS...

THE MUTEX SLEEPS

Inside Locking Primitives

- Spinlock

```
//spinlock_lock:
disable_interrupts();
while(locked==true);

//critical region

//spinlock_unlock:
enable_interrupts();
locked=false;
```

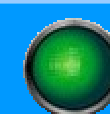
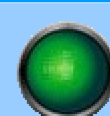
- Mutex

```
//mutex_lock:
If (locked==true)
{
    Enqueue(this);
    Yield();
}
locked=true;

//critical region

//mutex_unlock:
If !isEmpty(waitqueue)
{
 wakeup(Dequeue());
}
Else locked=false;
```

We can't sleep while the spinlock is locked!

We can't use a mutex in an interrupt because interrupts can't sleep!

When to use what?

	Mutex	**Spinlock**
Short Lock Time		
Long Lock Time		
Interrupt Context		
Sleeping		

- Usually functions that handle memory, user space or devices and scheduling sleep
  - Kmalloc, printk, copy_to_user, schedule
- wake_up_process does not sleep

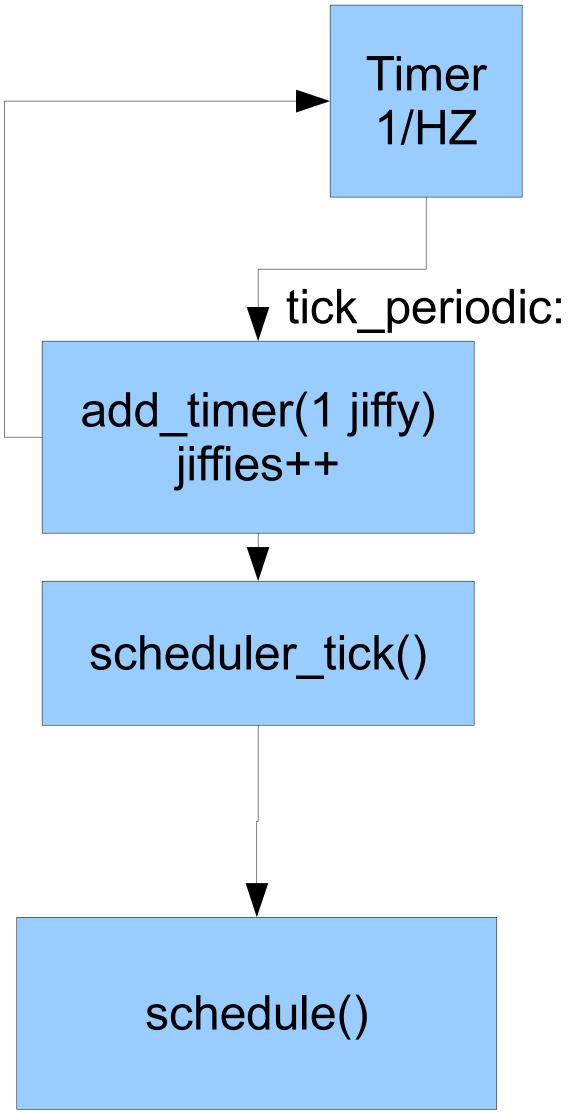
Linux Kernel Modules

- Extensibility
  - Ideally you don't want to patch but build a kernel module
- Separate Compilation
- Runtime-Linkage
- Entry and Exit Functions
  - Run in Process Context
- LKM “Hello-World”

```
#define MODULE

#define LINUX

#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init myinit(void)
{
   printk(KERN_ALERT ”Hello,
world\n”);
   Return 0;
}

static void __exit myexit(void)
{
   printk(KERN_ALERT ”Goodbye,
world\n”);
}

module_init(myinit);
module_exit(myexit);
MODULE_LICENSE(”GPL”);
```

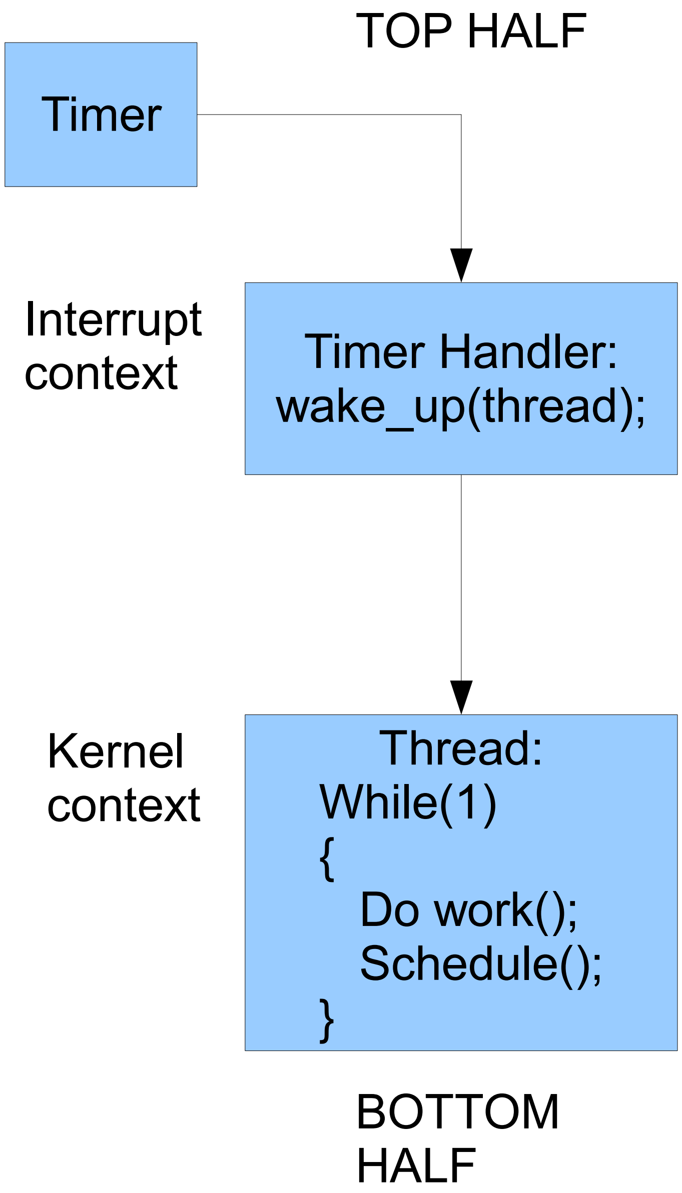
The Kernel Loop

- The Linux kernel uses the concept of jiffies to measure time
- Inside the kernel there is a loop to measure time and preempt tasks
- A jiffy is the period at which the timer in this loop is triggered
  - Varies from system to system 100 Hz, 250 Hz, 1000 Hz.
  - Use the variable HZ to get the value.
- The schedule function is the function that preempts tasks

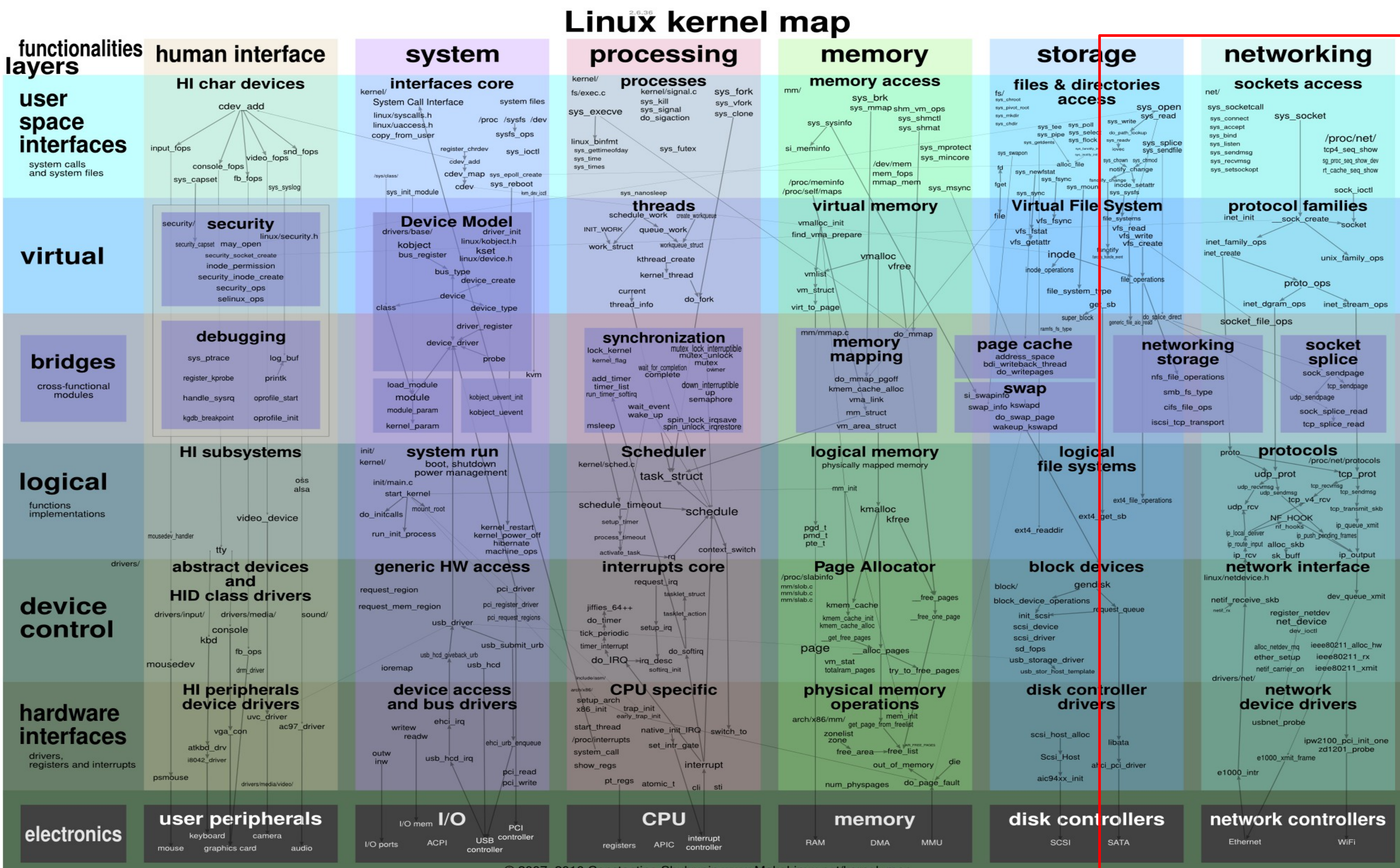


Deferring Work / Two Halves

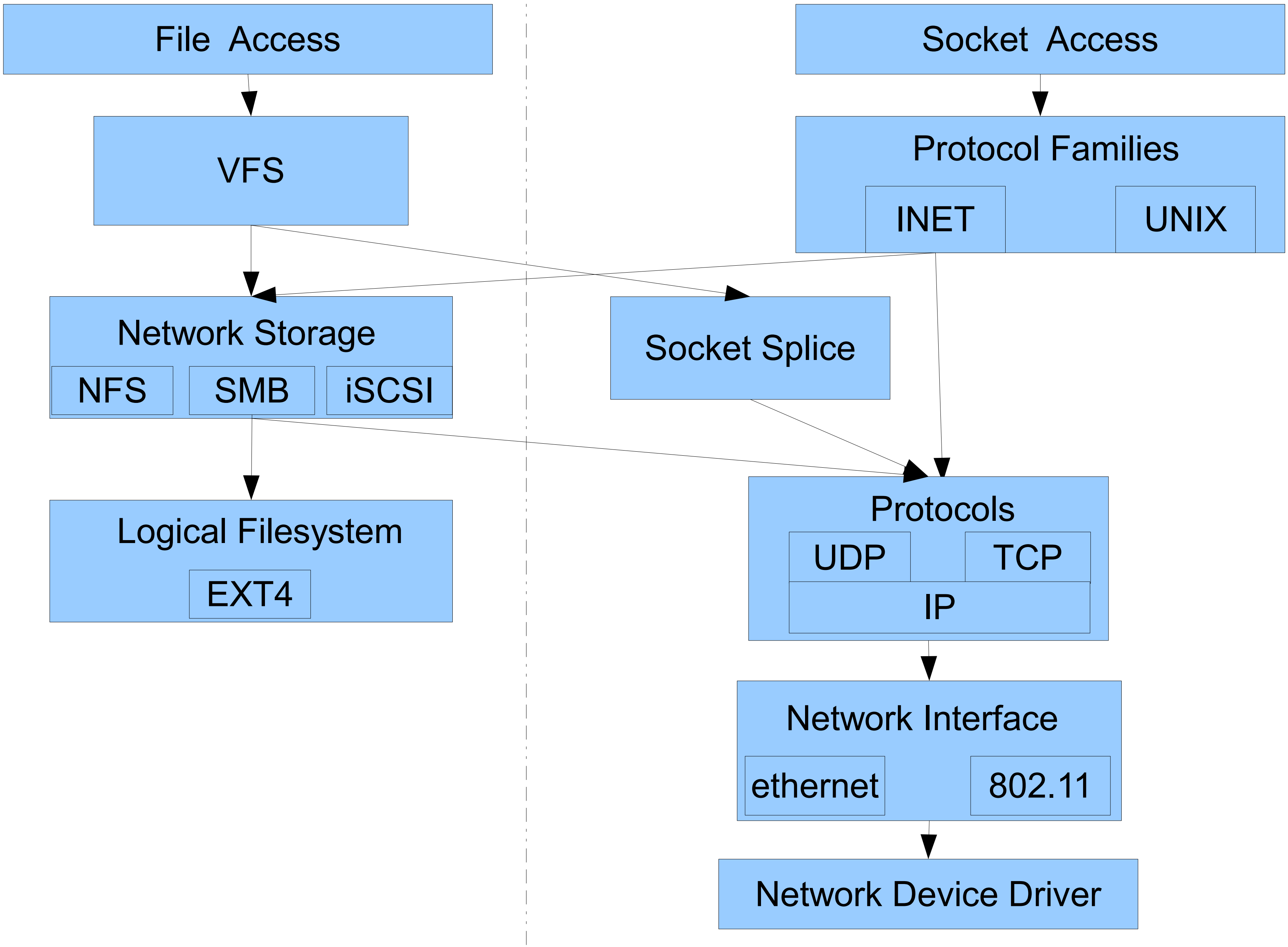
- Kernel Timers are used to create timed events
- They use jiffies to measure time
- Timers are interrupts
  - We can't do much in them!
- Solution: Divide the work in two parts
  - Use the timer handler to signal a thread. (TOP HALF)
  - Let the kernel thread do the real job. (BOTTOM HALF)



Linux Kernel Map

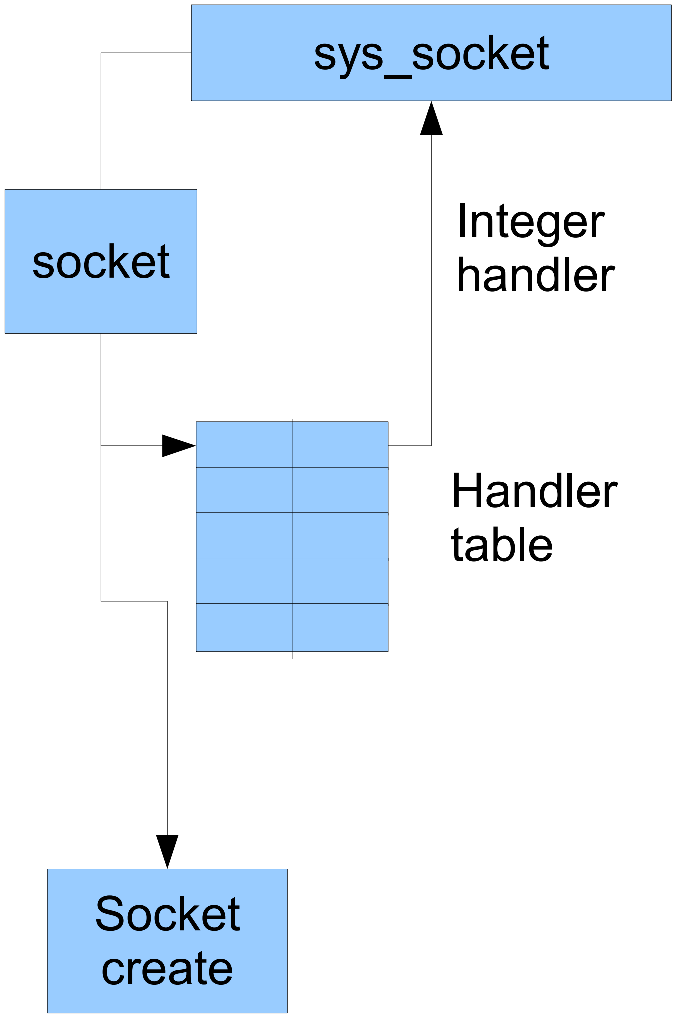


Linux Network Architecture



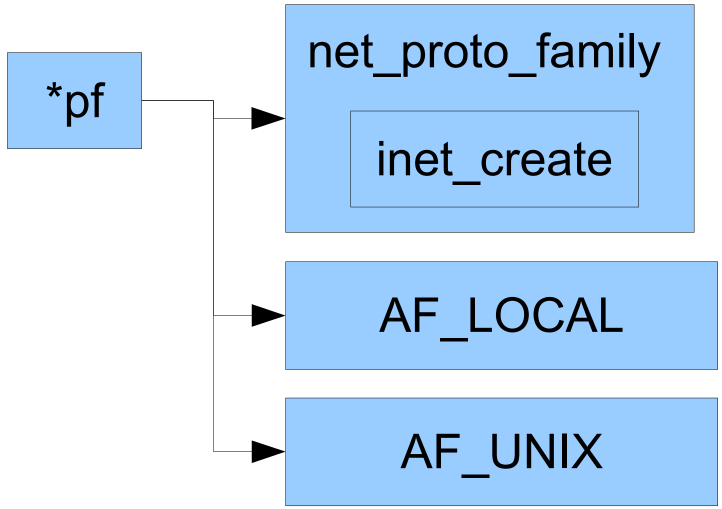
Socket Access

- Contains the system call functions like socket, connect, accept, bind
- Implements the POSIX socket interface
- Independent of protocols or socket types
- Responsible of mapping socket data structures to integer handlers
- Calls the underlying layer functions
  - sys_socket()→sock_create



Protocol Families

- Implements different socket families INET, UNIX
- Extensible through the use of pointers to functions and modules.
- Allocates memory for the socket
- Calls net_proto_familiy → create for familiy specific initilization

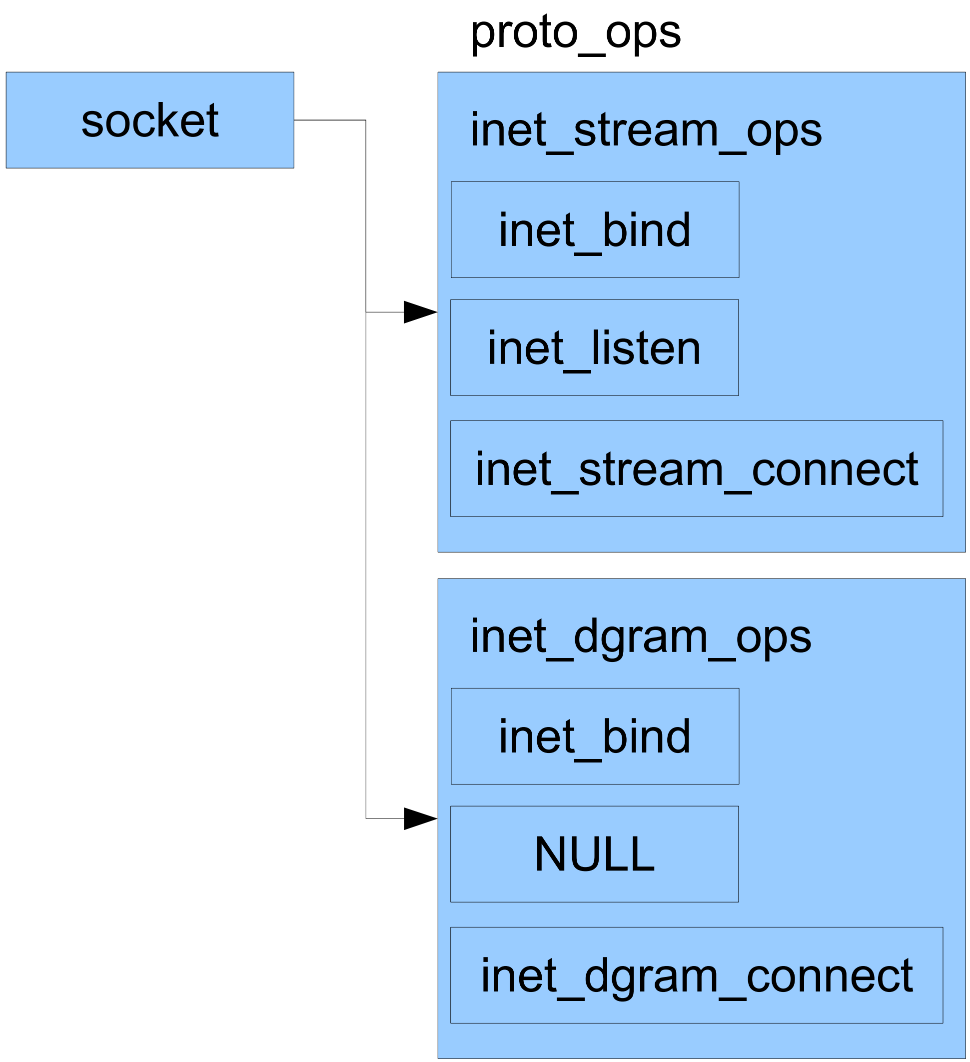


Socket Splice

- Unix uses the abstraction of Files as first class objects
- Linux supports to send entire files between file descriptors.
    - A descriptor can be a socket
- Also Unix supports Network File Systems
    - NFS, Samba, Coda, Andrew
- The socket splice is responsible of handling these abstractions

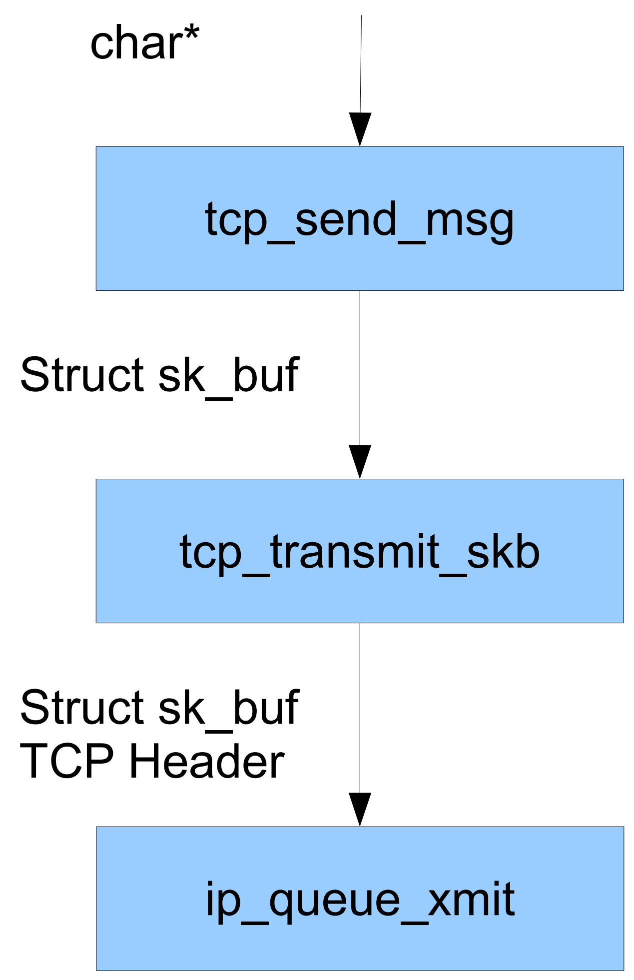
Protocols

- Families have multiple protocols
  - INET: TCP, UDP
- Protocol functions are stored in proto_ops
- Some functions are not used in that protocol so they point to dummies
- Some functions are the same across many protocols and can be shared



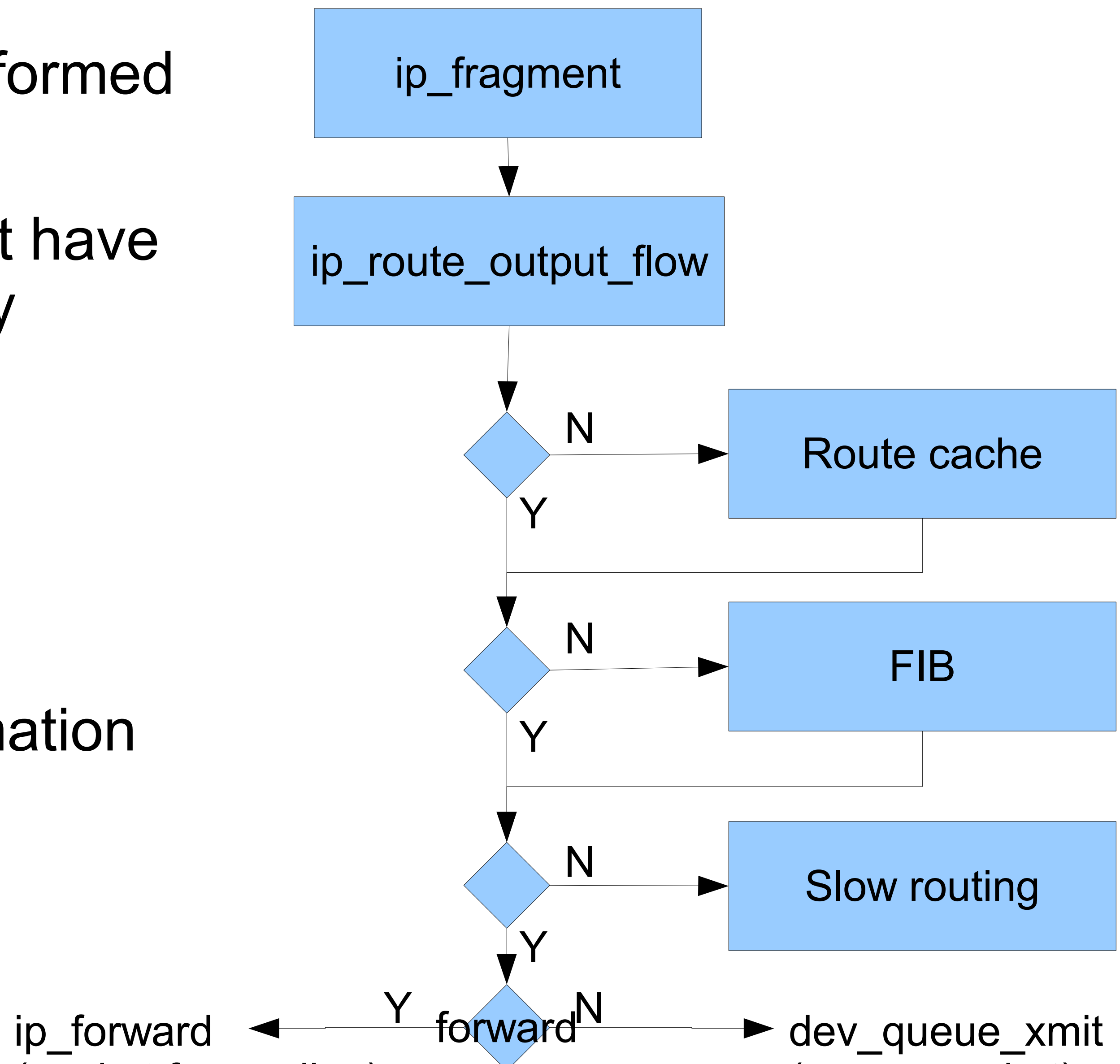
Packet Creation

- At the sending function, the buffer is packetized.
- Packets are represented by the sk_buff data structure
- Contains pointers the:
  - transport layer header
  - Link-layer header
  - Received Timestamp
  - Device we received it
- Some fields can be NULL



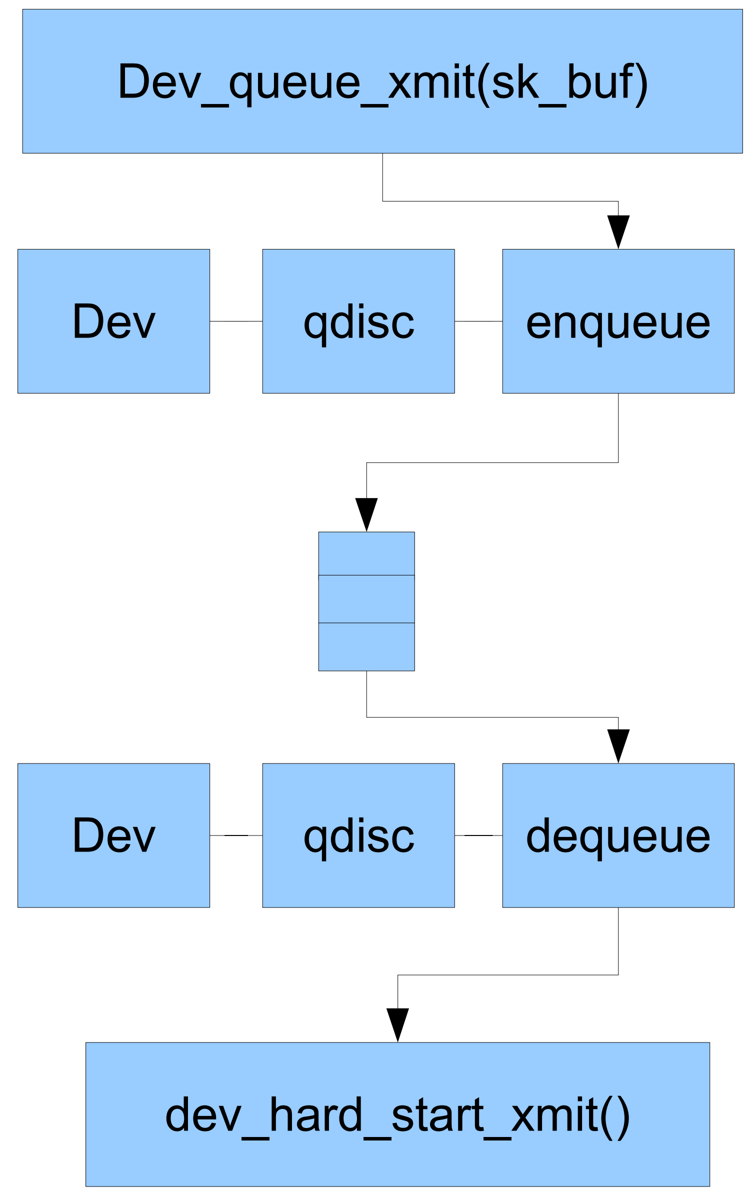
Fragmentation and Routing

- Fragmentation is performed inside ip_fragment
- If the packet does not have a route it is filled in by ip_route_output_flow
- There are routing mechanisms used
  - Route Cache
  - Forwarding Information Base
  - Slow Routing



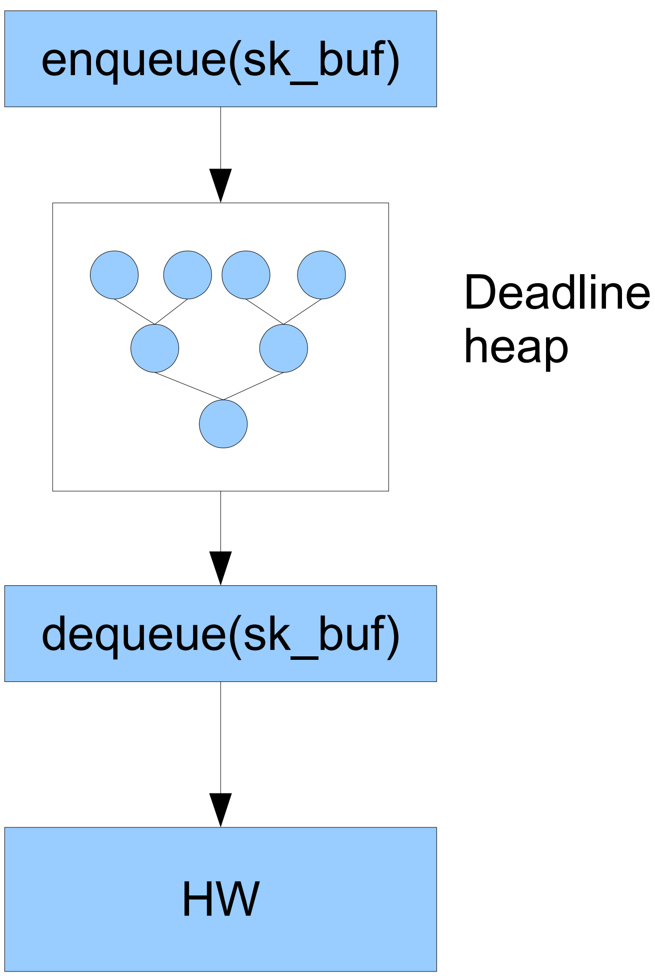
Data Link Layer

- The Data Link Layer is responsible of packet scheduling
- The dev_queue_xmit is responsible of enqueing packets for transmission in the qdisc of the device
- Then in process context it is tried to send
- If the device is busy we schedule the send for a later time
- The dev_hard_start_xmit is responsible for sending to the device



Case Study: iNET

- INET is an EDF (Earliest Deadline First) packet scheduler
- Each Packet has a deadline specified in the TOS field
- We implemented it as a Linux Kernel Module
- We implement a packet scheduler at the qdisc level.
- Replace qdisc enqueue and dequeue functions
- Enqueued packets are put in a heap sorted by deadline



High-Performance Network Stacks

- Minimize copying
  - Zero copy technique
  - Page remapping
- Use good data structures
  - Inet v0.1 used a list instead of a heap
- Optimize the common case
  - Branch optimization
- Avoid process migration or cache misses
  - Avoid dynamic assignment of interrupts to different CPUs
- Combine Operations within the same layer to minimize passes to the data
  - Checksum + data copying

High-Performance Network Stacks

- Cache/Reuse as much as you can
  - Headers, SLAB allocator
- Hierarchical Design + Information Hiding
  - Data encapsulation
- Separation of concerns
- Interrupt Moderation/Mitigation
  - Receive packets in timed intervals only (e.g. ATM)
- Packet Mitigation
  - Similar but at the packet level

Conclusion

- The Linux kernel has 3 main contexts: Kernel, Process and Interrupt.
- Use spinlock for interrupt context and mutexes if you plan to sleep holding the lock
- Implement a module avoid patching the kernel main tree
- To defer work implement two halves. Timers + Threads
- Socket families are implemented through pointers to functions (net_proto_family and proto_ops)
- Packets are represented by the sk_buf structure
- Packet scheduling is done at the qdisc level in the Link Layer

References

- Linux Kernel Map http://www.makelinux.net/kernel_map
- A. Chimata, Path of a Packet in the Linux Kernel Stack, University of Kansas, 2005
- Linux Kernel Cross Reference Source
- R. Love, Linux Kernel Development , 2^{nd} Edition, Novell Press, 2006
- H. Nguyen, R. Rivas, iDSRT: Integrated Dynamic Soft Realtime Architecture for Critical Infrastructure Data Delivery over WAN, Qshine 2009
- M. Hassan and R. Jain, High Performance TCP/IP Networking: Concepts, Issues, and Solutions, Prentice-Hall, 2003
- K. Ilhwan, Timer-Based Interrupt Mitigation for High Performance Packet Processing, HPC, 2001
- Anand V., TCPIP Network Stack Performance in Linux Kernel 2.4 and 2.5, IBM Linux Technology Center