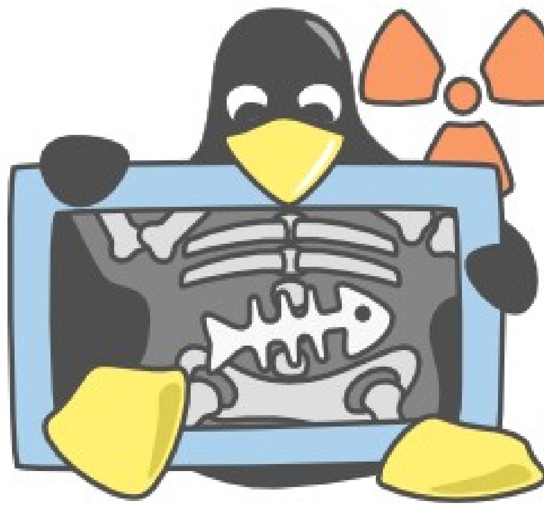


Linux Kernel Programming

Raoul Rivas

Kernel vs Application Programming

KERNEL

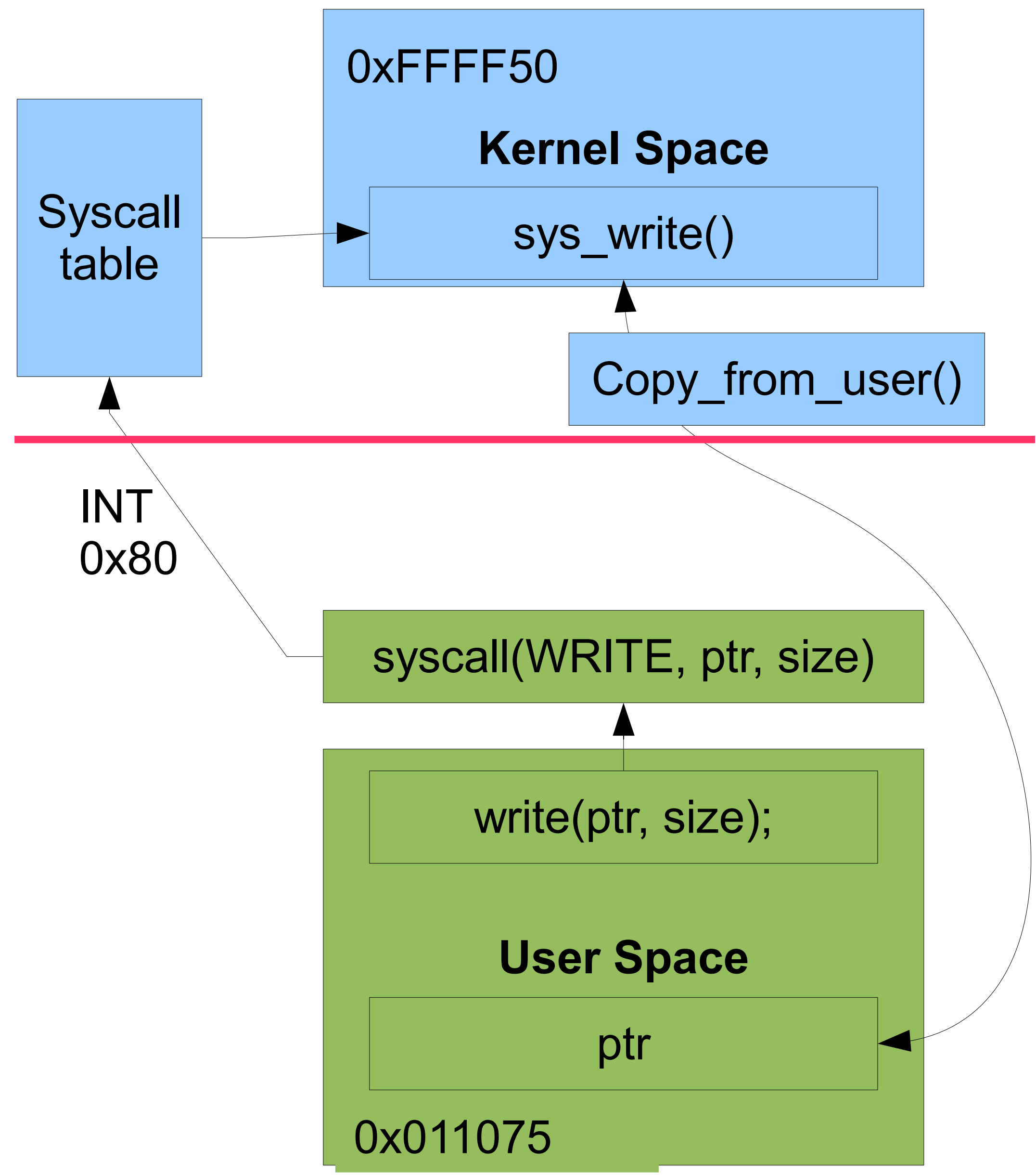
- No memory protection
  - We share memory with devices, scheduler
- Sometimes no preemption
  - Can hog the CPU
  - Concurrency is difficult
- No libraries
  - No Printf(), fopen()
- No security descriptors
- In Linux no access to files
- Direct access to hardware

APPLICATION

- 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

System Calls

- A system call involves 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. We can sleep in them
- Interrupts cannot sleep and should be small!
- All these entities are 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;
```

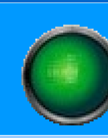
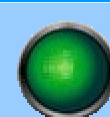
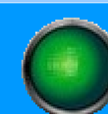
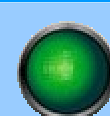
We can't sleep while the spinlock is locked! → DEADLOCK

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

- Mutex

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

//critical region

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

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

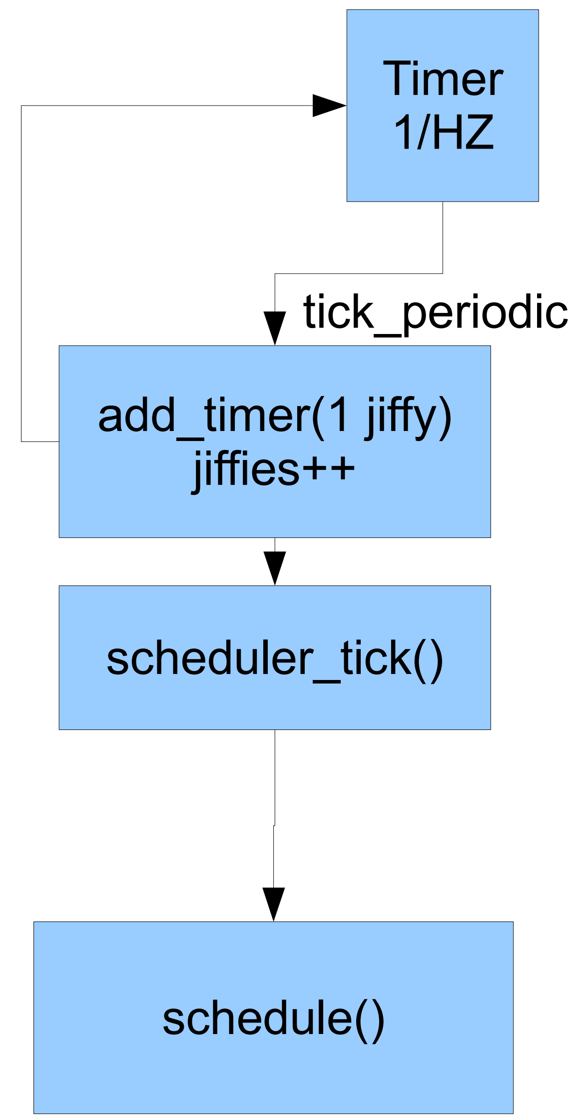
Linux Kernel Modules

- 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”);
```

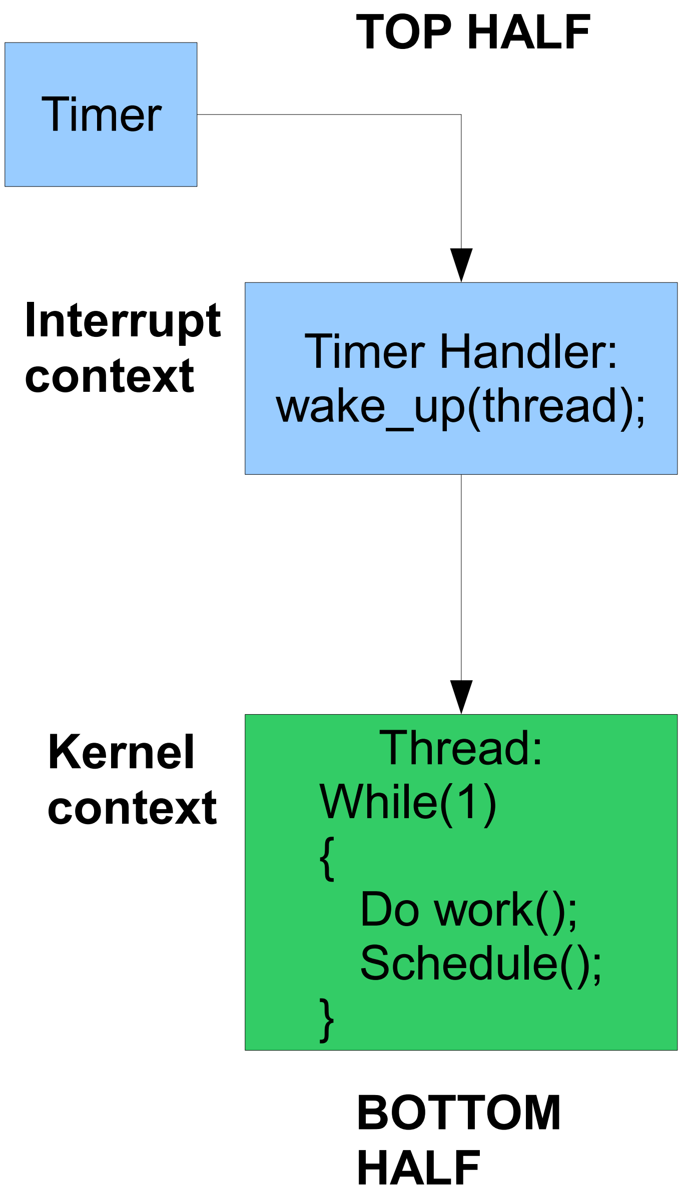
Jiffies and The Kernel Loop

- 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

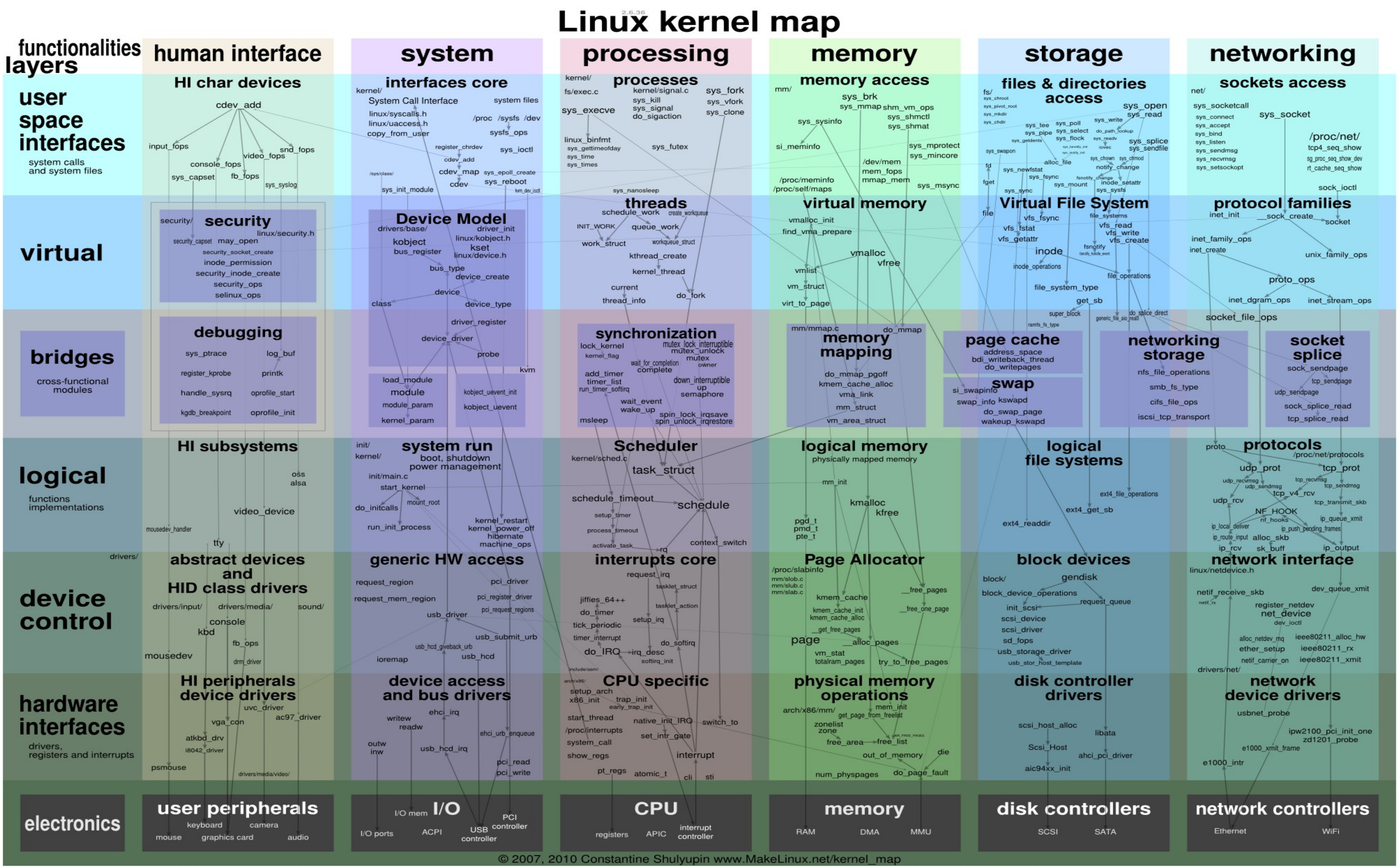


Deferring Work / Two Halves

- Kernel Timers are used to create timed events
- They use jiffies to measure time
- Timers are interrupts
  - We can't sleep or hog CPU 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



Optimizing Performance

- Minimize copying
- Use good data structures
- Optimize the common case
  - Branch optimization: likely(), unlikely()
- 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
  - e.g: Checksum + data copying

Optimizing Performance

- Cache/Reuse as much as you can
  - Cache Headers, SLAB allocator
- Hierarchical Design + Information Hiding
  - Data encapsulation
- Separation of concerns
- Interrupt Moderation/Mitigation
  - Group Timers if possible

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

References

- Linux Kernel Map http://www.makelinux.net/kernel_map
- Linux Kernel Cross Reference Source
- R. Love, Linux Kernel Development , 2nd Edition, Novell Press, 2006