

# Kernel Hacking 101

Deepak Saxena  
MontaVista Software, Inc.  
Presented @ ASULUG  
10/21/2002

## Goals

- Introduction to kernel programming concepts
  - Driver/kernel/user space interfacing
  - Kernel internals
  - Kernel memory management
  - Concurrent execution

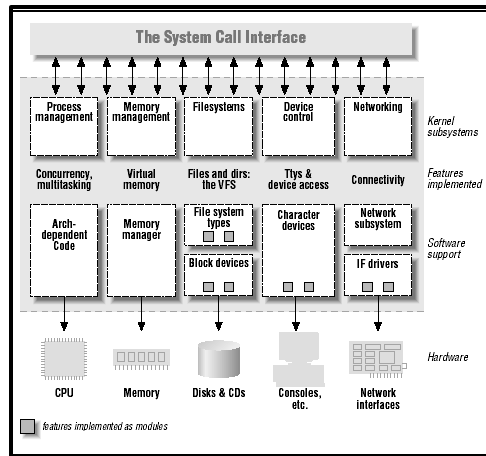
## So you want to be a kernel hacker?

- Fluent 'C' programming
  - Extremely comfortable working with pointers and ptr math
  - Understand how to map OO concepts to C
- Theoretical knowledge of OS concepts
- Understanding of HW concepts
  - Caches
  - Translation lookaside buffers
  - Virtual vs physical vs bus address spaces
  - Multi-Processing issues
  - ...

## What is a driver?

- A driver is a set of kernel code that:
  - Provides access to a HW or virtual resource
  - Abstracts HW interfaces from user applications
- Driver does not:
  - Provide policy on how that device is used

# The Bigger Picture



10/21/2002

Kernel Hacking 101

5

# Classes of Drivers

- **Character devices**
  - Can be accessed as a stream of bytes
    - Real Time Clocks
    - Watch Dog Timers
    - Console (serial)
    - Keyboard/mouse/touch screen
- **Block Devices**
  - Accessed with block size (512K+) granularity
  - Can host a file system
    - Disks
- **Network**
- **Virtual**

10/21/2002

Kernel Hacking 101

6

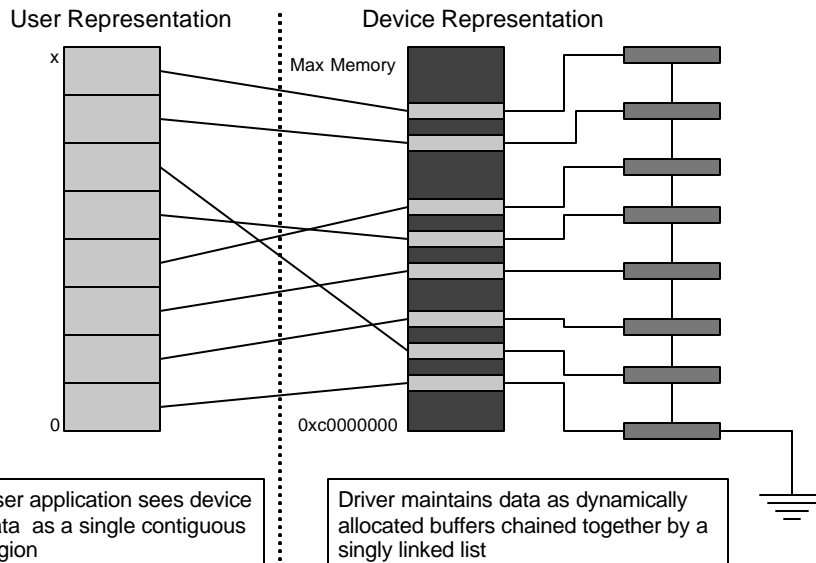
## Accessing Devices from User Space

- Multiple Methods Exist:
  - /proc interface
    - Being killed
  - /dev device nodes
    - Block and Character devices
  - Device class specific syscalls
    - Network interfaces

## Our “Device”

- We don't want to deal with real hardware
  - Too much of a chance of crashing system
  - Too many other variables to learn about
- Creating a virtual character device
  - Blocks of memory that user can access as stream of bytes
  - Provide following system call interfaces:
    - open()
    - close()
    - read()
    - write()
    - ioctl()
    - fasync()

## A graphical view



10/21/2002

Kernel Hacking 101

9

## Step1: Telling the kernel about ourselves

- Must provide a way for user to choose the driver!
- Driver must tell the kernel it exists
- Driver must tell kernel about user space interface

10/21/2002

Kernel Hacking 101

10

## Step 1.a: User Choice

- First thing to do is edit drivers/char/config.in:
  - `tristate "Support Deepak's Test Driver" CONFIG_MY_DRIVER`
  - This allows three selections:
    - Y – Yes, build driver into kernel
    - N – No, do not build driver
    - M – Build driver as a loadable module
  - Sets the `CONFIG_MY_DRIVER` variable in the `.config` file to one of above
- Edit drivers/char/Makefile:
  - `obj-$(CONFIG_MY_DRIVER) += my_driver.o`
  - Build system now know this driver exists

## Step 1.b: Telling kernel about us

- Following lines @ end of source code:
  - `MODULE_LICENSE("GPL");`
  - `MODULE_AUTHOR("Deepak Saxena <dsaxena@mvista.com>");`
  - `MODULE_DESCRIPTION("A test driver");`
  - `module_init(my_device_init);`
  - `module_exit(my_device_exit);`
- Used by kernel for module execution:
  - `module_init()` is initialization function
  - `module_exit()` is exit function
- Used for reporting purposes:
  - Licensing and author info **required**
  - Description is not needed

## Step 1.a: Details

```
static int
my_device_init(void)
{
    int ret, i;
    printk(KERN_INFO "MyDriver Version %s (built %s:%s)\n",
           DRIVER_VERSION, __DATE__, __TIME__);
    if((ret = devfs_register_chrdev(240, "mydev", &my_device_fops))){
        printk(KERN_ERR "Could not register device with kernel\n");
        return ret;
    }
    for(i = 0; i < MAX_DEVICES; i++) {
        /*
         * allocate device, allocate device data
         */
        my_devices[i].dev_data =
            (struct device_data*)kmalloc(sizeof(struct device_data), GFP_KERNEL);
        if(!my_devices[i].dev_data) {
            printk (KERN_WARNING "Could not allocate memory\n");
            return -ENOMEM;
        }
        my_devices[i].dev_data->nnext = NULL;
        my_devices[i].end_data = my_devices[i].dev_data;
        my_devices[i].dev_data->ptr = vmalloc(BUFFER_SIZE);
        if(!my_devices[i].dev_data->ptr) {
            printk (KERN_WARNING "Could not allocate memory\n");
            return -ENOMEM;
        }
        memset(my_devices[i].dev_data->ptr, BUFFER_SIZE);
        my_devices[i].memsize = BUFFER_SIZE;
        my_devices[i].needed_mem = BUFFER_SIZE;
        init_timer(&my_devices[i].timer);
        my_devices[i].timer.data = (unsigned long)&my_devices[i];
        my_devices[i].timer.function = my_device_timer;
        init_waitqueue_head(&my_devices[i].waitq);
        init_rwsem(&my_devices[i].sem);
    }
    return 0;
}
```

## Let's do this one step at a time

```
printk(KERN_INFO "MyDriver Version %s (built %s:%s)\n",
        DRIVER_VERSION, __DATE__, __TIME__);
```

- `printk()` is your friend
  - Kernel version of `printf()`
    - Supports all standard args plus a logging level
      - `KERN_INFO`, `KERN_DEBUG`, `KERN_CRIT`, `KERN_ERR`,  
`KERN_WARNING`, `KERN_EMERGENCY`
      - Level which you wish to see depends on syslog configuration

## Telling the kernel about our device

```
if((ret = devfs_register_chrdev(240, "mydev", &my_device_fops)) {
    printk(KERN_ERR "Could not register device with kernel\n");
    return ret;
}
```

- `devfs_register_chrdev()`:
  - Telling kernel we are a character device at major 240
  - User needs to create a device to access us:
    - `mknod /dev/mynod0 c 240 0`
    - `mknod /dev/mynod1 c 240 1`
    - ...
- Why would it fail?
  - Someone else might have that device node already

## struct file\_operations

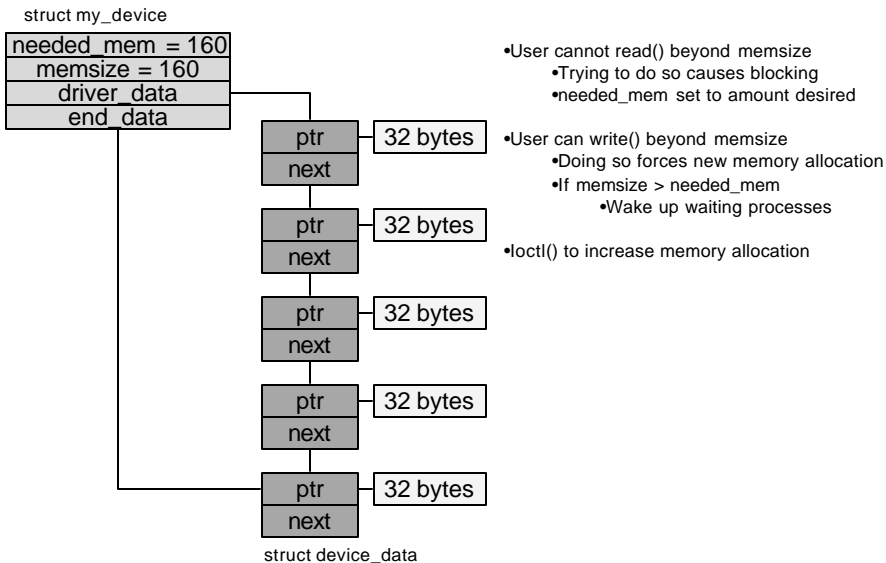
- Third parameter to `devfs_register_chrdev()`
- Tells kernel what system call interfaces we support:

```
static struct file_operations my_device_fops =
{
    .owner          = THIS_MODULE,
    .read           = my_device_read,
    .write          = my_device_write,
    .ioctl          = my_device_ioctl,
    .fasync         = my_device_fasync,
    .open           = my_device_open,
    .release        = my_device_release,
};
```

- `THIS_MODULE` is a macro that points to ourselves
- Strange syntax is new kernel coding guideline...



## Device Details



10/21/2002

Kernel Hacking 101

19

## Allocating Memory

- First thing we do is go allocate some memory for device

```

my_devices[i].dev_data =
    (struct device_data*)kmalloc(sizeof(struct device_data), GFP_KERNEL);
if(!my_devices[i].dev_data) {
    printk (KERN_WARNING "Could not allocate memory\n");
    return -ENOMEM;
}
my_devices[i].dev_data->next = NULL;
my_devices[i].end_data = my_devices[i].dev_data;
my_devices[i].dev_data->ptr = vmalloc(BUFFER_SIZE);

if(!my_devices[i].dev_data->ptr) {
    printk (KERN_WARNING "Could not allocate memory\n");
    return -ENOMEM;
}
    
```

- kcalloc vs. vmalloc?
- Need to understand how kernel manages memory

10/21/2002

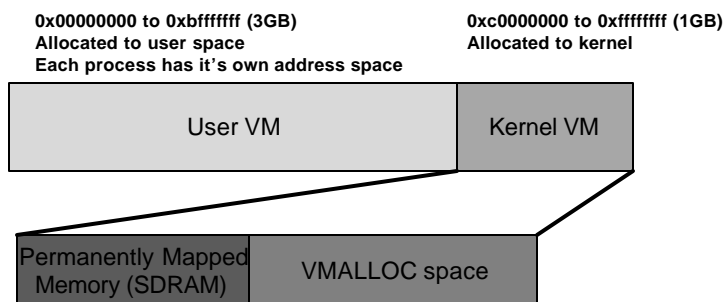
Kernel Hacking 101

20

## Acquiring Memory

- Linux always runs with MMU turned on
  - ALL address references are virtual
- `kmalloc()` allocates physically contiguous memory
  - Very important when the memory will be accessed by HW
  - Kernel provides macros to convert to phys addr and back
- `vmalloc()` allocates virtual contiguous memory
  - Useful for internal use only buffers
  - Reduces strain on VM to keep blocks contiguous
  - However: can hurt TLB performance if memory is accessed often

## Linux Memory map



## kmalloc() in detail

- kmalloc(size, flags)
  - Size is up to 128K due to VM limitations
  - Flags:
    - GFP\_KERNEL
      - Allows execute process to be put to sleep while allocation completes
    - GFP\_ATOMIC
      - Does not allow sleeping...used in interrupts where there is no context
    - GFP\_DMA & GFP\_HIGHMEM
      - Request memory from a specific zone
      - Linux divides phys memory into different zones
      - See next slide.

## Memory Zones

0x00000000 to 0x00FFFFFF (16MB) ZONE_DMA	0x00100000 to 0x3FFFFFFF (1GB) ZONE_NORMAL	0x3FFFFFFF to 0xFFFFFFFF+(4GB+) ZONE_HIGH
--	--	---

- Kernel maps ZONE\_DMA and ZONE\_NORMAL starting @ virtual memory 0xc0000000. This memory is permanently mapped and cannot be paged.
- ZONE\_DMA is leftover from ISA days since ISA devices cannot access beyond 16MB of memory. On non-x86 platforms, it's treated very differently
- ZONE\_HIGH is memory beyond ~1G (default) that is not permanently mapped.

## Step 2: Doing something

- my\_device\_open:

```
374 struct my_device *dev = NULL;
375 int minor = MINOR(ip->i_dev);
383 fp->private_data = (void*)&my_devices[minor];
```

- private\_data allows us to keep driver specific information across system calls

## Let's write some data

- User Code:

```
int fd = open("/dev/mynod0", O_RDWR);
write(fd, buffer, 4096); // Write a big buffer
```

- Kernel code takes 4 parameters:

- struct file \*fp // Pointer to file
- char \*data // User space pointer to data buffer
- ssize\_t len // Length of buffer
- loff\_t \*off // Offset into data buffer

## Concurrent Writes

- Only one person can write data to device at time
- Writes must be atomic or data will be corrupted

```
237 down_write(&dev->sem);
238 if(signal_pending(current))
239 return -ERESTARTSYS;
```

- struct my\_device.sem is of type rw\_semaphore
- down\_write() causes semaphore to be taken
  - Any other writers will go to sleep when code gets to this point
- If interrupted by another signal (ctrl-c for example), we return
  - current is very often used variable that provides process context

- Later in code:

```
323 up_write(&dev->sem);
```

- **Wakes up next sleeping writer in queue**

## Concurrent Access part 2

- If we need more memory, call allocate\_mem()

```
247 if(*f_pos + len > dev->memsize)
248     allocate_mem(dev, *f_pos + len);
```

- allocate\_memory() is pretty simple
  - Allocate a couple of device\_data structures and memory areas
  - Increase memsize parameter
  - But...what's this:

```
115 spin_lock_irqsave(&dev->lock, flags);
116 dev->memsize += BUFFER_SIZE;
117 spin_unlock_irqrestore(&dev->lock, flags);
```

- dev->lock is of type spinlock\_t
  - Much finer grain locking
  - Low overhead
  - Just spins

## Concurrency Continued

- struct semaphore
  - Allow processes to be put to sleep
  - High setup/tear down cost
  - Used to protect large sections of code from concurrent execution
- spinlock\_t
  - Just spins on lock until lock is released
  - No setup/teardown costs
  - Used for protecting a piece of data from being changed/accessed concurrently
  - Translates to IRQ disable on single processor systems

## Accessing user data

- Unlike kernel memory user memory can be paged
  - This means that you could try to access a ptr, but that ptr could not be in physical memory and page fault
  - That pointer could be an invalid user address
- ```
255 if(copy_from_user(buffer, data, len)) {  
256   up_write(&dev->sem)  
257   return -EFAULT;  
258 }
```
- copy\_from\_user() returns number of bytes not copied
    - If we failed to copy data, we return `-EFAULT` which means that an address fault occurred.

## Done writing data

- Finished writing data and possibly increased buffer size
  - Check to see if any readers are waiting on data

```
307 spin_lock_irqsave(&dev->lock, flags);
308 if(dev->memsize >= dev->needed_mem) {
309     dprintk("WRITE: Waking up waiting readers: %d bytes\n", dev->needed_mem);
310     dev->needed_mem = dev->memsize;
311     wake_up_interruptible(&dev->waitq);
312 }
313 spin_unlock_irqrestore(&dev->lock, flags);
```

- Acquire spinlock since we are touching memsize
- Call wake\_up\_interruptible() to wake up sleeping readers
- Hmm...let's look at my\_device\_read code

## my\_device\_read()

- First thing we do, is check to see if there's enough data

```
142 spin_lock_irqsave(&dev->lock, flags);
143 if(*f_pos + len >= dev->memsize) {
144     if(*f_pos + len > dev->needed_mem) 144
145         dev->needed_mem = *f_pos + len;
146     spin_unlock_irqrestore(&dev->lock, flags); 146
147     dprintk("READ: Sleeping...need data: %d bytes\n", dev->needed_mem); 147
148     interruptible_sleep_on(&dev->waitq); 148
149     dprintk("READ: Woken up!\n"); 149
150     if(signal_pending(current)) 150
151         return -ERESTARTSYS; 151
152 }
153 spin_unlock_irqrestore(&dev->lock, flags); 153
```

- Again, acquire spinlock since we're touching memory sizes
- Check to see if we need more memory than is currently queued
- If so, we go to sleep on the &dev->waitq wait queue
- If signal pending when done, we exit

## Wait Queues

- Wait Queues integral to many portions of kernel code
- Initialized by:

```
466 init_waitqueue_head(&my_devices[i].waitq);
```
- Provide consumer/produce synchronization
  - Consuming device goes to sleep on a waitq:
    - `sleep_on()/interruptible_sleep_on()`
      - Sleep on a waitq. Using `sleep_on()` means that the process is locked in sleeping state until woken and cannot receive signals. Can be dangerous! (This is 'D' state in ps)
    - `sleep_on_timeout()/sleep_on_interruptible_timeout()`
      - Timesout after number of 'jiffies' have elapsed
  - Producing device wakes up consumer when ready:
    - `wake_up()/wake_up_interruptible()`
  - Or just wait for an event
    - `wait_event(&wq, test_condition)`

## Concurrency, pt 3

- We allow multiple readers, but only one writer
- Use same waitq
- Instead of `down_write()`, use `down_read()`

```
155 dprintk("READ: Trying to acquire read semaphore\n");
156 down_read(&dev->sem);
157 if(signal_pending(current))
158     return -ERESTARTSYS;
159 dprintk("READ: Read semaphore acquired\n");
```
- Multiple readers can be accessing at same time
- `down_write()` causes readers and writers to sleep

## Asynchronous Access

- User wants to asynchronously know when data available

- Registers application to receive signals

```
151     fcntl(fd, F_SETOWN, getpid()); 151
152     oflags = fcntl(STDIN_FILENO, F_GETFL);
153     fcntl(fd, F_SETFL, oflags | FASYNC);
154     if(sigaction(SIGIO, &action, &old_action))
155         perror("Could not set signal handler: ");
156     else
157         printf("\nAsync notification now ON\n\n");
```

- Kernel responds by just putting the process on a queue

```
332 static int
333 my_device_fasync(int fd, struct file *fp, int on)
334 {
335     struct my_device *dev = (struct my_device *)fp->private_data;
336
337     return fasync_helper (fd, fp, on, &dev->async_queue);
338 }
```

## Back to write() syscall

- When data is available, the kernel notifies all waiting processes

```
316if(dev->async_queue) {
317     dprintk("WRITE: Sending async notification\n");
318     kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
319 }
```

- All real work is done by kernel helper functions

## Device Specific Control

- We can now read/write and register for data notification
- But devices can do a lot more than that
  - Reset
  - Eject
  - Link up/down
  - Query for information
- We need hardware specific controls
  - Don't want to add a new system call per device command
- I/O Controls to the rescue
  - ioctl(fd, command, arg)
    - Command and arg are device spec

## Our I/O Controls

- IOMYDEV\_ALLOC\_MEM
  - #define IOMYDEV\_ALLOC\_MEM 0x6b01
  - Argument is an unsigned long
  - Allocates new memory
- IOMYDEV\_MEM\_INFO
  - #define IOMYDEV\_MEM\_INFO 0x6b02
  - Argument is an unsigned long\*
  - Fills \*ptr with current value of dev->memsize

## I/O Control Code is generally simple

```
343 static int
344 my_device_ioctl(struct inode *ip, struct file *fp, unsigned int cmd,
                 unsigned long arg)
345 {
346     struct my_device *dev = (struct my_device *)fp->private_data;
347     unsigned long flags;
348
349     if(cmd == IOMYDEV_ALLOC_MEM) {
350         spin_lock_irqsave(&dev->lock, flags);
351         if(dev->memsize >= arg) {
352             spin_unlock_irqrestore(&dev->lock, flags);
353             return 0;
354         }
355         spin_unlock_irqrestore(&dev->lock, flags);
356         allocate_mem(dev, arg);
357     } else if (cmd == IOMYDEV_MEM_INFO) {
358         unsigned long *addr = (unsigned long *)arg;
359         if(!put_user(dev->memsize, addr))
360             return -EFAULT;
361     } else return -EINVAL;
362
363     return 0;
364 }
```

10/21/2002

Kernel Hacking 101

39

## Our driver is complete

- Provides basic functionality:
  - read
  - write
  - ioctl
  - fasync
  - open
  - release

10/21/2002

Kernel Hacking 101

40

## Additional Topics

- Some more memory management
- Flow of time
- Scheduling & asynchronous tasks

## More Memory Management

- We talked about allocating memory, but how do we free?
  - `kfree()` to free `kmalloc`'d memory
  - `vfree()` to free `vmalloc`'d memory
- Other ways to allocate memory
  - What if we're going to be creating and destroying lots and lots of the same object type (ex: `struct file *`, network buffers)
  - We can `kmalloc/kfree` everytime we need object
    - High overhead
    - VM system does not like you if you do this
  - Kernel provides an object cache allocator (slab allocator)

# Slab Allocator

```
kmem_cache_t *my_cache = kmem_cache_create("my cache", offset,  
sizeof(my_object), flags, my_constructor, my_destructor);
```

- Offset – set to 0...offset into page frame...
- Flags: Can force cache line alignment on object allocations
- Constructor/destructor
  - Think C++. Functions called to initialize and destroy objects
- To get an object:

```
struct my_object *ptr = (struct my_object*)kmem_cache_alloc(my_cache, flags);
```

- To delete an object:

```
kmem_cache_alloc(my_cache, (void*)ptr);
```

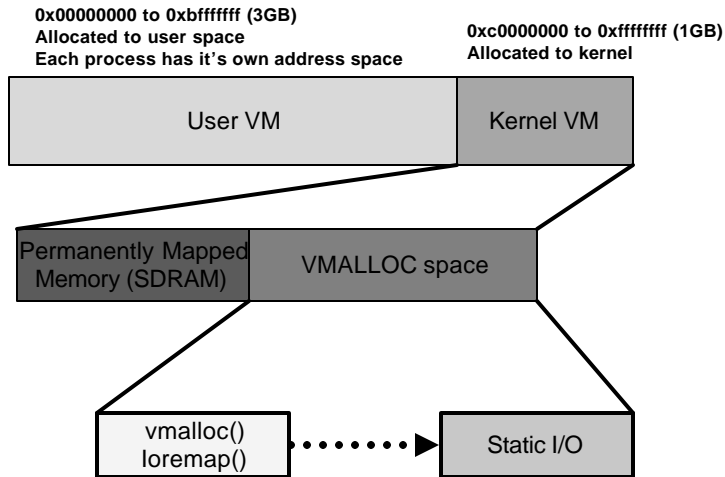
- To destroy cache:

```
kmem_cache_destroy(my_cache);
```

# Accessing Device Memory

- Devices live on physical bus
- Need to be mapped into virtual address space to use
  - void \*ptr = ioremap(physaddr, length)
    - Maps in length bytes starting at physaddr into VM
    - Cannot access directly
      - readl/writel(ptr) – 32bit
      - reads/writes(ptr) – 16bit
      - readb/writeb(ptr) – 8 bit
    - When done:
      - iounmap(ptr);

## Memory Map Again



10/21/2002

Kernel Hacking 101

45

## What if we run out of VM?!

- What if more > 1GB of RAM
  - All of VM taken up
- What if lots of devices on system
  - Need large amount of VM space
- This is where high memory comes in
  - Reduces amount of permanently mapped SDRAM
  - Allows for more VM growth
  - Reduces amount of memory accessible by devices
- Advanced options allow sliding of user/kernel VM split

10/21/2002

Kernel Hacking 101

46

## Flow of Time

- Linux keeps time internally in terms of jiffies
- One jiffie = 1 clock tick
- Default is 100 ticks/per second
  - 10ms between ticks
- To timeout for x seconds:  $x * \text{HZ}$

## Scheduling

- Talked about sleeping/waking up a lot
- When does Linux schedule things?
- Linux scheduler is in kernel/sched.c:schedule()
- Scheduler is called when one of following occurs:
  - Clock tick
  - After interrupts
  - User forced scheduling
    - schedule()
    - schedule\_timeout(jiffies)
    - Placing process in a waitq

## Tasks/tasklets/processes/threads

- Linux almost always executing in a process context
  - current points to struct task\_struct
  - Thread in Linux are just processes that share VM
- Things that execute out of process context:
  - IRQ handlers
  - Tasklets and task queues
- When executing outside of user context:
  - Cannot sleep as it makes no sense
    - Causes kernel to OOPs
  - All memory allocation must be GFP\_ATOMIC

## Linux kernel non-preempt by Default

- Process A running, makes a system call
  - System call executing
  - IRQ comes in, gets handled
    - Scheduler gets called
      - Non-process tasks get executed
  - Returns to interrupted system call
- Pre-K Patch (default in 2.5)
  - Allows kernel space to be pre-empted
  - Decreases latency for time critical operations

## What Next?

- Go read code
- Run test code
- Try to add functionality
  - Non-Blocking I/O
  - More ioctls to do interesting things
- Read sample drivers:
  - drivers/char/wdt.c (watch dog timer) is good simple example
- Pick up following:
  - Linux Device Drivers
  - Understanding the Linux Kernel
- Start skimming lkml
  - Kernel Traffic and lwn.net are good summaries to read