Introduction to Linux Device Drivers

Recreating Life One Driver At a Time

Muli Ben-Yehuda

mulix@mulix.org

IBM Haifa Research Labs, Haifa Linux Club

why write Linux device drivers

- For fun
- For profit (Linux is hot right now, especially embedded Linux)
- To scratch an itch
- Because you can

OK, but why Linux drivers?

- Because the source is available
- Because of the community's cooperation and involvement
- Have I mentioned it's fun yet?

klife - Linux kernel game of life

klife is a Linux kernel Game of Life implementation. It is a software device driver, developed specifically for this talk.

- the game of life is played on a square grid, where some of the cells are alive and the rest are dead
- each generation, based on each cell's neighbours, we mark the cell as alive or dead
- with time, amazing patterns develop
- the only reason to implement the game of life inside the kernel is for demonstration purposes

Software device drivers are very common on Unix systems and provide many services to the user. Think about /dev/null, /dev/zero, /dev/random, /dev/kmem...

anatomy of a device driver

- a device driver has two sides. One side talks to the rest of the kernel and to the hardware, and the other side talks to the user
- to talk to the kernel, the driver registers with subsystems to respond to events. Such an event might be the opening of a file, a page fault, the plugging in of a new USB device, etc
- since Linux is UNIX and in UNIX everything is a file, users talk with device drivers through device files
- klife is a character device, and thus the user talks to it through a character device file
- the other common kind of device file is a block device file. We will only discuss character device files today

anatomy of a device driver - cont

- the user talks with klife through the /dev/klife device file
- when the user opens /dev/klife, the kernel calls klife's open routine
- when the user closes /dev/klife, the kernel calls klife's release routine
- when the user reads or writes from or to /dev/klife - you get the idea. . .
- klife talks to the kernel through its initialization function
- . . . and through register_chardev
- . . . and through hooking into the timer interrupt
- we will elaborate on all of these later

driver initialization code

```
static int __init klife_module_init(void)
{
        int ret;

        pr_debug("klife module init called\n");

        ret = register_chrdev(KLIFE_MAJOR_NUM, "klife", &klife_fops);
        if (ret < 0)
                printk(KERN_ERR "register_chrdev: %d\n", ret);

        return ret;
}

static void __exit klife_module_cleanup(void)
{
        unregister_chrdev(KLIFE_MAJOR_NUM, "klife");
}
```

driver initialization

- one function (init) called on the driver's initialization
- one function (exit) called when the driver is removed from the system
- question: what happens if the driver is compiled into the kernel, rather than as a module?
- the init function will register hooks that will get the driver's code called when the appropriate event happens
- question: what if the init function doesn't register any hooks?
- there are various hooks that can be registered: file operations, pci operations, USB operations, network operations - it all depends on what kind of device this is.

registering chardev hooks

```
struct file_operations klife_fops = {
        .owner = THIS_MODULE,
        .open = klife_open,
        .release = klife_release,
        .read = klife_read,
        .write = klife_write,
        .mmap = klife_mmap,
        .ioctl = klife_ioctl
};
...
if ((ret = register_chrdev(KLIFE_MAJOR_NUM, "klife", &klife_fops)) < 0)
        printk(KERN_ERR "register_chrdev: %d\n", ret);
```

user space access to the driver

We saw that the driver registers a character device tied to a given major number, but how does the user create such a file?

```
# mknod /dev/klife c 250 0
```

And how does the user open it?

```
if ((kfd = open("/dev/klife", O_RDWR)) < 0) {
        perror("open /dev/klife");
        exit(EXIT_FAILURE);
}
```

And then what?

file operations

. . . and then you start talking to the device. klife makes use of the following device file operations:

- open for starting a game (allocating resources)
- release for finishing a game (releasing resources)
- write for initializing the game (setting the starting positions on the grid)
- read for generating and then reading the next state of the game's grid
- ioctl for querying the current generation number, and for enabling or disabling hooking into the timer interrupt (more on this later)
- mmap for potentially faster but more complex direct access to the game's grid

klife_open - 1

```
static int klife_open(struct inode *inode, struct file *filp)
{
        struct klife* k;

        k = alloc_klife();
        if (!k)
                return -ENOMEM;

        filp->private_data = k;

        return 0;
}
```

klife_open - 2

```
static struct klife* alloc_klife(void)
{
        struct klife* k = kmalloc(sizeof(*k), GFP_KERNEL);

        if (!k)
                return NULL;

        init_klife(k);

        return k;
}
```

klife_open - 3

```
static void init_klife(struct klife* k)
{
        memset(k, 0, sizeof(*k));

        spin_lock_init(&k->lock);

        k->timer_hook.func = klife_timer_irq_handler;
        k->timer_hook.data = k;
}
```

klife_release

```
static int klife_release(struct inode *inode, struct file *filp)
{
        struct klife* k = filp->private_data;

        klife_timer_unregister(k);

        free_klife(k);

        return 0;
}

static void free_klife(struct klife* k)
{
        kfree(k);
}
```

commentary on open and release

- open and release are where you perform any setup not done in initialization time and any cleanup not done in module unload time
- klife's open routine allocates the klife structure which holds all of the state for this game (the grid, starting positions, current generation, etc)
- klife's release routine frees the resource allocated during open time
- beware of races if you have any global data . . . many a driver author stumble on this point
- note also that release can fail, but almost no one checks errors from close(), so it's better if it doesn't . . .
- question: what happens if the userspace program crashes while holding your device file open?

klife_write - 1

```
static ssize_t klife_write(struct file* filp, const char __user * ubuf,
                           size_t count, loff_t *f_pos)
{
        size_t sz;
        char* kbuf;
        struct klife* k = filp->private_data;
        ssize_t ret;

        sz = count > PAGE_SIZE ? PAGE_SIZE : count;

        kbuf = kmalloc(sz, GFP_KERNEL);
        if (!kbuf)
                return -ENOMEM;
```

klife_write - 2

```
        ret = -EFAULT;
        if (copy_from_user(kbuf, ubuf, sz))
                goto free_buf;

        ret = klife_add_position(k, kbuf, sz);
        if (ret == 0)
                ret = sz;

free_buf:
        kfree(kbuf);
        return ret;
}
```

commantary on write

- for klife, I “hijacked” write to mean “please initialize the grid to these starting positions. There are no hard and fast rules to what write has to mean, but KISS is a good principle to follow.
- note that even for such a simple function, care must be exercise when dealing with untrusted users (users are always untrusted).
- check the size of the user’s buffer
- use copy_from_user in case the user is passing a bad pointer
- always be prepared to handle errors!

klife_read - 1

```
static ssize_t
klife_read(struct file *filp, char *ubuf, size_t count, loff_t *f_pos)
{
        struct klife* klife;
        char* page;
        ssize_t len;
        ssize_t ret;
        unsigned long flags;

        klife = filp->private_data;

        /* special handling for mmap */
        if (klife->mapped)
                return klife_read_mapped(filp, ubuf, count, f_pos);

        if (!(page = kmalloc(PAGE_SIZE, GFP_KERNEL)))
                return -ENOMEM;
```

klife_read - 2

```
spin_lock_irqsave(&klife->lock, flags);

klife_next_generation(klife);

len = klife_draw(klife, page);

spin_unlock_irqrestore(&klife->lock, flags);

if (len < 0) {
        ret = len;
        goto free_page;
}

len = min(count, (size_t)len); /* len can't be negative */
```

klife_read - 3

```
        if (copy_to_user(ubuf, page, len)) {
                ret = -EFAULT;
                goto free_page;
        }

        *f_pos += len;
        ret = len;

free_page:
        kfree(page);
        return ret;
}
```

klife_read - 4

```
static ssize_t
klife_read_mapped(struct file *filp, char *ubuf, size_t count,
        loff_t *f_pos)
{
        struct klife* klife;
        unsigned long flags;

        klife = filp->private_data;

        spin_lock_irqsave(&klife->lock, flags);

        klife_next_generation(klife);

        spin_unlock_irqrestore(&klife->lock, flags);

        return 0;
}
```

commentary on read

- for klife, read means “please calculate and give me the next generation”
- the bulk of the work is done in klife_next_generation and klife_draw. klife_next_generation calculate the next generation based on the current one according to the rules of the game of life, and klife_draw takes a grid and “draws” it as a single string in a page of memory
- we will see later what the lock is protecting us against
- note that the lock is held for the smallest possible time
- again, copy_to_user in case the user is passing us a bad page
- there’s plenty of room for optimization in this code . . . can you see where?

klife_ioctl - 1

```
static int
klife_ioctl(struct inode* inode, struct file* file,
        unsigned int cmd,  unsigned long data)
{
        struct klife* klife = file->private_data;
        unsigned long gen;
        int enable;
        int ret;
        unsigned long flags;

        ret = 0;
        switch (cmd) {
        case KLIFE_GET_GENERATION:
                spin_lock_irqsave(&klife->lock, flags);
                gen = klife->gen;
                spin_unlock_irqrestore(&klife->lock, flags);
                if (copy_to_user((void*)data, &gen, sizeof(gen))) {
                        ret = -EFAULT;
                        goto done;
                }
```

klife_ioctl - 2

```
                break;
        case KLIFE_SET_TIMER_MODE:
                if (copy_from_user(&enable, (void*)data, sizeof(enable))
                        ret = -EFAULT;
                        goto done;
                }
                pr_debug("user request to %s timer mode\n",
                         enable ? "enable" : "disable");

                if (klife->timer && !enable)
                        klife_timer_unregister(klife);
                else if (!klife->timer && enable)
                        klife_timer_register(klife);
                break;
        }
done:
        return ret;
}
```

commentary on ioctl

- ioctl is a “special access” mechanism, for operations that do not cleanly map anywhere else
- it is considered exteremly bad taste to use ioctls in Linux where not absolutely necessary
- new drivers should use either sysfs (a /proc like virtual file system) or a driver specific file system (you can write a linux file system in less than a 100 lines of code)
- in klife, we use ioctl to get the current generation number, for demonstration purposes only . . .

klife_mmap

```
static int klife_mmap(struct file* file, struct vm_area_struct* vma)
{
        int ret;
        struct klife* klife = file->private_data;

        pr_debug("inside klife_mmap, file %p, klife %p, vma %p\n",
                 file, klife, vma);

        if (vma->vm_flags & VM_SHARED)
                return -EINVAL; /* we don't support MAP_SHARED */

        pr_debug("vma %p, vma->vm_start %ld, vma->vm_end %ld "
                 "gridsize %d\n", vma, vma->vm_start, vma->vm_end,
                 sizeof(klife->grid));
```

klife_mmap

```
        SetPageReserved(virt_to_page(&klife->grid));
        ret = remap_page_range(vma, vma->vm_start,
                               virt_to_phys(&klife->grid),
                               sizeof(klife->grid),
                               vma->vm_page_prot);

        pr_debug("remap_page_range returned %d\n", ret);

        if (ret == 0)
                klife->mapped = 1;

        return ret;
}
```

commentary on mmap

- mmap is used to map pages of a file into memory
- programs can access the memory directly, instead of having to use read and write (saves up the overhead of a system call and related context switching, and memory copying overhead)
- . . . but fast synchronization between kernel space and user space is a pain, and Linux read and write are really quite fast
- again, implemented in klife for demonstration purposes, with read() calls used for synchronization and triggering a generation update (why do we need synchronization between kernel space and userspace?)

klife interrupt handler

What if we want a new generation on every raised interrupt? since we don't have a hardware device to raise interrupts for us, let's hook into the one hardware every pc has - the clock - and steal its interrupt! Usually, interrupts are requested using request_irq():

```
/* claim our irq */
rc = -ENODEV;
if (request_irq(card->irq, &trident_interrupt, SA_SHIRQ,
                card_names[pci_id->driver_data], card)) {
        printk(KERN_ERR "trident: unable to allocate irq %d\n", card->ir
        goto out_proc_fs;
}
```

It is not possible to request the timer interrupt. Instead, we will directly modify the kernel code to call our interrupt handler, if it's it's registered. We can do this, because the code is open…

aren't timers good enough for you?

- "does every driver that wishes to get periodic notifications need to hook the timer interrupt?". Nope.
- Linux provides an excellent timer mecahnism which can be used for periodic notifications.
- The reason for hooking into the timer interrupt in klife is because we wish to be called from hard interrupt context, also known as (top half context) . . .
- . . . where as timer functions are called in softirq (bottom half context).
- why insist on getting called from hard interrupt context? so we can demonstrate deferring work

hook into the timer interrupt routine - 1

```
+struct timer_interrupt_hook* timer_hook;
+
+static void call_timer_hook(struct pt_regs *regs)
+{
+       struct timer_interrupt_hook* hook = timer_hook;
+
+       if (hook && hook->func)
+               hook->func(hook->data);
+}
+
@@ -851,6 +862,8 @@ void do_timer(struct pt_regs *regs)
        update_process_times(user_mode(regs));
 #endif
        update_times();
+
+       call_timer_hook(regs);
 }
```

hook into the timer interrupt routine - 2

```
+int register_timer_interrupt(struct timer_interrupt_hook* hook)
+{
+       printk(KERN_INFO "registering a timer interrupt hook %p "
+              "(func %p, data %p)\n", hook, hook->func,
+              hook->data);
+
+       xchg(&timer_hook, hook);
+       return 0;
+}
+
+void unregister_timer_interrupt(struct timer_interrupt_hook* hook)
+{
+       printk(KERN_INFO "unregistering a timer interrupt hook\n");
+
+       xchg(&timer_hook, NULL);
+}
+
+EXPORT_SYMBOL(register_timer_interrupt);
+EXPORT_SYMBOL(unregister_timer_interrupt);
```

commentary on timer interrupt hook

- the patch adds a hook a driver can register for, to be called directly from the timer interrupt handler. It also creates two functions, register_timer_interrupt and unregister_timer_interrupt, which do the obvious thing.
- note that the register and unregister calls use xchg(), to ensure atomic replacement of the pointer to the handler. Why use xchg() rather than a lock?
- interesing questions: what context (hard interrupt, bottom half, process context) will we be called in? which CPU's timer interrupts would we be called in? what happens on an SMP system?

deferring work

- you were supposed to learn in class about bottom halves, softirqs, tasklets and other such curse words
- the timer interrupt (and every other interrupt) has to happen very quickly (why?)
- the interrupt handler (top half, hard irq) usually just sets a flag that says “there is work to be done”
- the work is then deferred to a bottom half context, where it is done by an (old style) bottom half, softirq, or tasklet
- for klife, we defer the work we wish to do, updating the grid, to a bottom half context by scheduling a tasklet

preparing the tasklet

```
DECLARE_TASKLET_DISABLED(klife_tasklet, klife_tasklet_func, 0);

static void klife_timer_register(struct klife* klife)
{
        unsigned long flags;
        int ret;

        spin_lock_irqsave(&klife->lock, flags);
        /* prime the tasklet with the correct data - ours */
        tasklet_init(&klife_tasklet, klife_tasklet_func,
                (unsigned long)klife);
        ret = register_timer_interrupt(&klife->timer_hook);
        if (!ret)
                klife->timer = 1;
        spin_unlock_irqrestore(&klife->lock, flags);

        pr_debug("register_timer_interrupt returned %d\n", ret);
}
```

deferring work - the klife tasklet

```
static void klife_timer_irq_handler(void* data)
{
        struct klife* klife = data;

        /* 2 times a second */
        if (klife->timer_invocation++ % (HZ / 2) == 0)
                tasklet_schedule(&klife_tasklet);
}

static void klife_tasklet_func(unsigned long data)
{
        struct klife* klife = (void*)data;

        spin_lock(&klife->lock);

        klife_next_generation(klife);

        spin_unlock(&klife->lock);
}
```

commentary on the klife tasklet

Here's what our klife tasklet does:

- first, it derives the klife structure from the parameter it is passed
- and then locks it to prevent concurrent access on another CPU (what are we protecting against?)
- then, it generates the new generation (what must we never do here? hint: can tasklets block?)
- and last, it releases the lock

adding klife to the build system

Building the module in kernel 2.6 is a breeze. All that's required to add klife to the kernel's build system are these tiny patches:

- in drivers/char/Kconfig

```
+config GAME_OF_LIFE
+ tristate "kernel game of life"
+ help
+   Kernel implementation of the Game of Life.
```

- in drivers/char/Makefile

```
+obj-$(CONFIG_GAME_OF_LIFE) += klife.o
```

summary and questions

- writing drivers is easy
- . . . and fun
- most drivers do fairly simple things, which Linux provides APIs for
- (the real fun is when dealing with the hardware's quirks)
- it gets easier with practice
- . . . but it never gets boring

Questions?

where to get help

- google
- community resources: web sites and mailing lists
- distributed documentation (books, articles, magazines)
- the source
- your fellow kernel hackers

bibliography

- kernelnewbies - http://www.kernelnewbies.org
- linux-kernel mailing listarchives -

  `http://marc.theaimsgroup.com/?l=linux-kernel&w=2`
- Understading the Linux Kernel, by Bovet and Cesati
- Linux Device Drivers, 2nd edition, by Rubini et. al.
- Linux Kernel Development, by Robert Love
- /usr/src/linux-xxx/