Introduction to Linux Device Drivers

Recreating Life One Driver At a Time

Muli Ben-Yehuda

`mulix at mulix.org`

IBM Haifa Research Labs and Haifux - 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

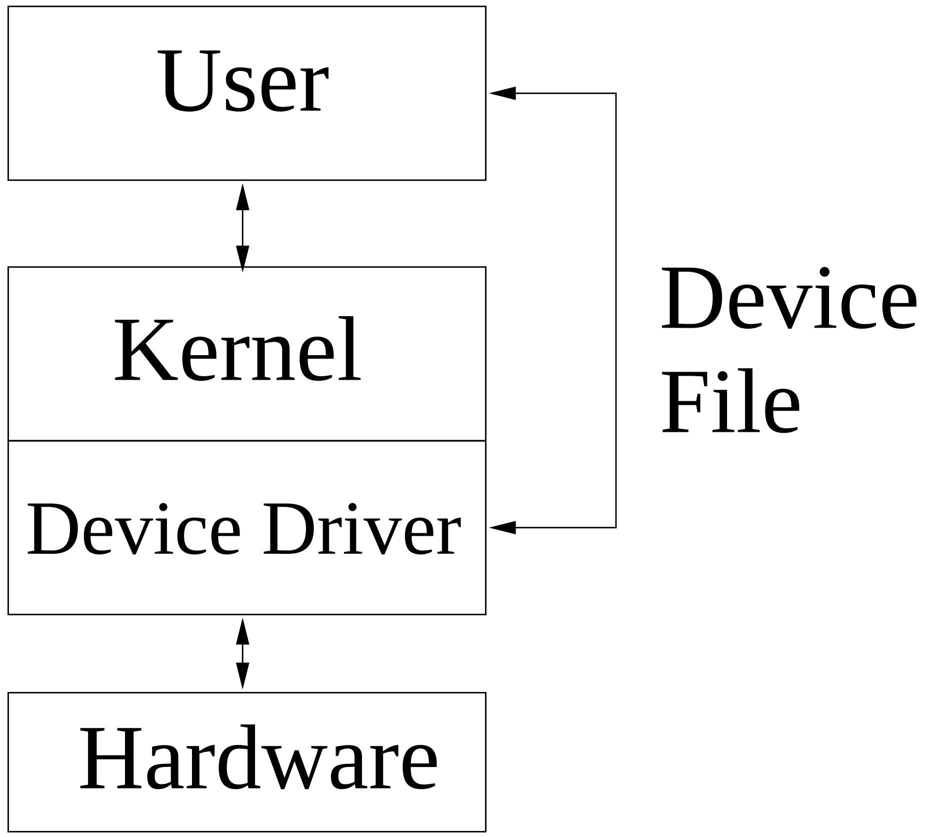
klife is a Linux kernel Game of Life implementation. It is a software device driver, developed specifically for this talk.

- 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 neighbors, 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...

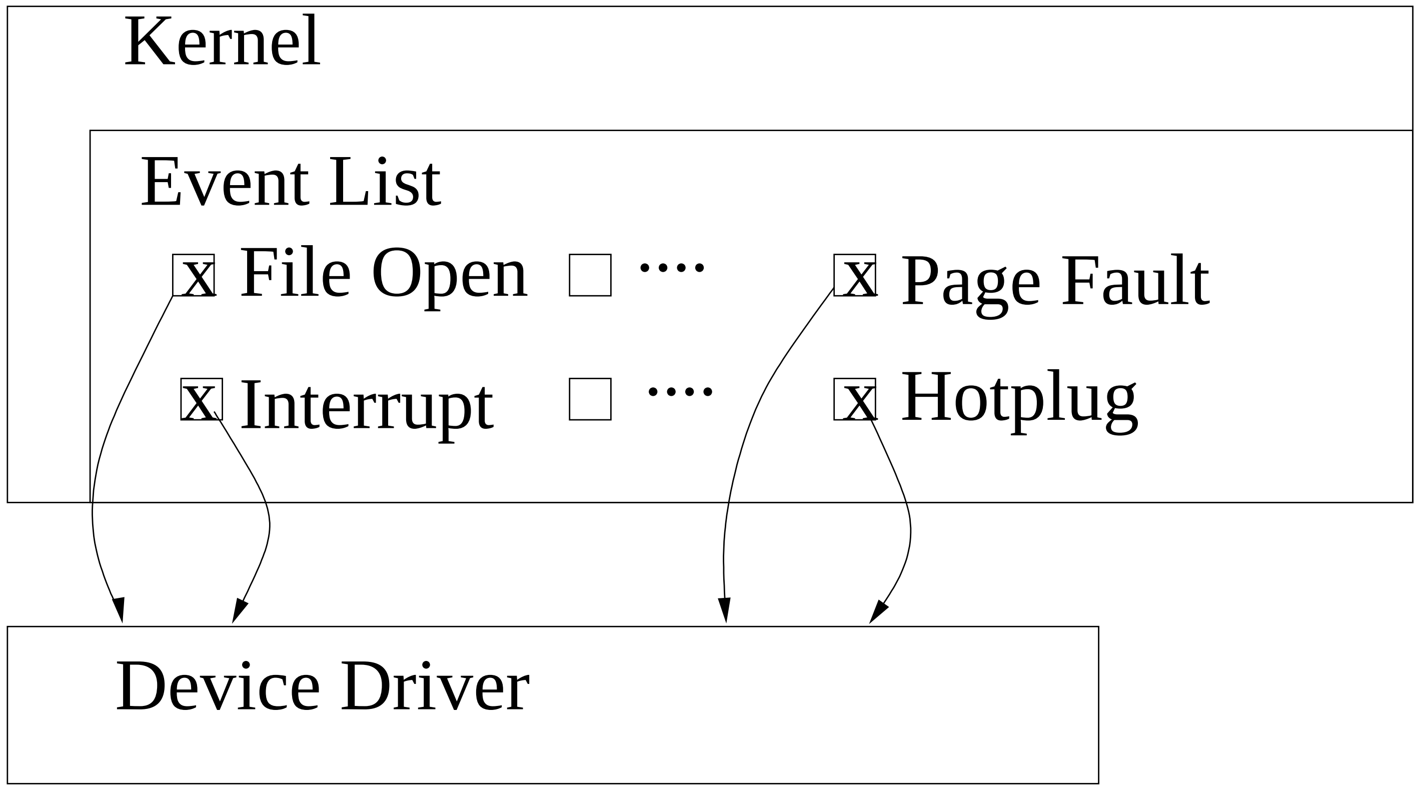
Anatomy of a Device Driver

- A device driver has three sides: one side talks to the rest of the kernel, one talks to the hardware, and one talks to the user:



Kernel Interface of a Device Driver

- In order 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.



User Interface of a Device driver

- Since Linux follows the UNIX model, and in UNIX everything is a file, users talk with device drivers through device files.
- Device files are a mechanism, supplied by the kernel, precisely for this direct User-Driver interface.
- 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 klife device driver

- 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_chrdev
  - . . . 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");

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

        return ret;
}
```

Driver Initialization

- One function (init) is called on the driver's initialization.
- One function (exit) is 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 uses 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.

The open and release Routines

open and release are where you perform any setup not done in initialization time and any cleanup not done in module unload time.

klife_open

klife's open routine allocates the klife structure which holds all of the state for this game (the grid, starting positions, current generation, etc).

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

        ret = alloc_klife(&k);
        if (ret)
                return ret;

        filp->private_data = k;

        return 0;
}
```

klife_open - alloc_klife

```
static int alloc_klife(struct klife** pk)
{
        int ret;
        struct klife* k;

        k = kmalloc(sizeof(*k), GFP_KERNEL);
        if (!k)
                return -ENOMEM;

        ret = init_klife(k);
        if (ret) {
                kfree(k);
                k = NULL;
        }

        *pk = k;
        return ret;
}
```

klife_open - init_klife

```
static int init_klife(struct klife* k)
{
        int ret;

        memset(k, 0, sizeof(*k));

        spin_lock_init(&k->lock);

        ret = -ENOMEM;
        /* one page to be exported to userspace */
        k->grid = (void*)get_zeroed_page(GFP_KERNEL);
        if (!k->grid)
                goto done;

        k->tmpgrid = kmalloc(sizeof(*k->tmpgrid), GFP_KERNEL);
        if (!k->tmpgrid)
                goto free_grid;
```

klife_open - init_klife cont'

```
        k->timer_hook.func = klife_timer_irq_handler;
        k->timer_hook.data = k;
        return 0;

free_grid:
        free_page((unsigned long)k->grid);
done:
        return ret;
}
```

klife_release

klife's release routine frees the resource allocated during open time.

```
static int klife_release(struct inode *inode, struct file *filp)
{
        struct klife* k = filp->private_data;
        if (k->timer)
                klife_timer_unregister(k);
        if (k->mapped) {
                /* undo setting the grid page to be reserved */
                ClearPageReserved(virt_to_page(k->grid));
        }
        free_klife(k);
        return 0;
}
```

Commentary on open and release

- 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?

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 it’s good to KISS (Keep It Simple, Silly...)

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;
```

Not trusting users: checking the size of the user's buffer

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;
}
```

Use copy_from_user in case the user is passing a bad pointer.

Commentary on write

- Note that even for such a simple function, care must be exercised when dealing with untrusted users.
- Users are always untrusted.
- Always be prepared to handle errors!

read

- For klife, read means “please calculate and give me the next generation”.
- The bulk of the work is done in two other routines:
  - klife_next_generation calculates the next generation based on the current one, according to the rules of the game of life.
  - klife_draw takes a grid and “draws” it as a single string in a page of memory.

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

Note that the lock is held for the shortest possible time.

We will see later what the lock protects us against.

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;
}
```

copy_to_user in case the user is passing us a bad page.

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;
}
```

Again, mind the short lock holding time.

Commentary on read

- There’s plenty of room for optimization in this code . . . can you see where?

ioctl

- ioctl is a “special access” mechanism, for operations that do not cleanly map anywhere else.
- It is considered extremely 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_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;
}
```

memory mapping

- The read-write mechanism, previously described, involves an overhead of a system call and related context switching and of memory copying.
- mmap maps pages of a file into memory, thus enabling programs to directly access the memory directly and save the overhead, . . . but:
  - fast synchronization between kernel space and user space is a pain (why do we need it?),
  - and Linux read and write are really quite fast.
- mmap is implemented in klife for demonstration purposes, with read() calls used for synchronization and triggering a generation update.

klife_mmap

```
        ...
        SetPageReserved(virt_to_page(klife->grid));
        ret = remap_pfn_range(vma, vma->vm_start,
                              virt_to_phys(klife->grid) >> PAGE_SHIFT,
                              PAGE_SIZE, vma->vm_page_prot);

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

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

        return ret;
}
```

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!

Usual Request For an Interrupt Handler

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->irq);
        goto out_proc_fs;
}
```

klife Interrupt Handler

- It is impossible to request the timer interrupt.
- Instead, we will directly modify the kernel code to call our interrupt handler, if it’s registered.
- We can do this, because the code is open. . .

Aren’t Timers Good Enough For You?

- “Does every driver which wishes to get periodic notifications need to hook the timer interrupt?” - Nope.
- Linux provides an excellent timer mechanism 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 . . .
- . . . whereas timer functions are called in softirq bottom half context.
- Why insist on getting called from hard interrupt context?
  - So we can demonstrate deferring work.

The Timer Interrupt Hook Patch

- The patch adds a hook which a driver can register for, to be called directly from the timer interrupt handler. It also creates two functions:
  - register_timer_interrupt
  - unregister_timer_interrupt

Hook Into The Timer Interrupt Routine 1

'+' marks the lines added to the kernel.

```
+struct timer_interrupt_hook* timer_interrupt_hook;
+
+static void call_timer_hook(struct pt_regs *regs)
+{
+        struct timer_interrupt_hook* hook = timer_interrupt_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);
+}
```

Commentary - The Timer Interrupt Hook

- 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?
- 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 which 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);
}
```

The klife Tasklet

Here's what our klife tasklet does:

- First, it derives the klife structure from the parameter it gets.
- Then, it 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?
- Last, it releases the lock.

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);
}
```

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

- Writing Linux 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)
- Use The Source, Luke!
- Your fellow kernel hackers.

Bibliography

- kernelnewbies - http://www.kernelnewbies.org
- linux-kernel mailing list archives - `http://marc.theaimsgroup.com/?l=linux-kernel&w=2`
- Understanding the Linux Kernel, by Bovet and Cesati
- Linux Device Drivers, 3rd edition, by Rubini et. al.
- Linux Kernel Development, 2nd edition, by Robert Love
- /usr/src/linux-xxx/