How Not to Get Your Code Accepted Into the Kernel: Social and Technical Lessons

- Deepak Saxena
- MontaVista Software, Inc.
- dsaxena@plexity.net

- January 19th, 2007

Why? What? Who? How?

- Lots of people getting involved with kernels
  - University and Industry Research projects
  - HW vendors: CPUs, SOCs, I/O devices, embedded, etc
- See same mistakes made over and over
  - Social
  - Technical
  - Point out some of the more common ones
    - Specially from the embedded folks
    - Include examples

Release Late, Release Rarely

- You have a great idea
    - New exciting research area, new feature, new HW tech, etc
    - Go hide in a dungeon for 12 months, working furiously!
    - Achieve marvelous results!
    - Release code to community after it is “done”
        - Asked to rewrite large parts of it!
        - “What?! I need to completely rethink my core idea!”
- Kernel developers get final say
    - You may have a great idea but implementation..
        - may make changes to kernel that have large ramifications
        - may not work cleanly across all arches
        - may be full of the issues I'm going to point out (and more)
    - Better to find issues early then to wait until you are “done”

The Cross-OS Abstraction Layer

- You have written some code for a new device
  - You want to share code across multiple OSes with not changes
  - You say to yourself “I need an abstraction layer!”
  - You create a sexy abstraction layer, hiding the OS specifics from your driver core. Your CS professor would be proud!
  - You submit code upstream
  - Your code gets rejected
- Cross-OS abstraction:
  - Makes it harder for upstream maintainers
    - Code is not calling same kernel APIs as everyone else
    - Abstraction layer might have bugs

Don't Create A Proper Abstraction Layer

- Your driver needs something not currently supported
  - New HW capability such as checksum offload, RAID offload, etc
  - You code capabilities, ways to enable/disable, etc directly into your driver
  - Or, you make changes directly to network, VFS, etc layer
  - Your code will not be accepted
    - Chances are others need these capabilities too,
    - Need an approach that is generic across HW implementations
- Work with community to add new features

#define my_custom_macro_because_I_can()

- `#define my_debug_warn(dev, ...) printk(KERN_WARN“%s”, ..., dev->name)`
    - Use `dev_warn()`
- `#define ASSERT(x) if (!x) printk(..., __FILE__, __LINE__, __FUNCTION__)`
    - Use `WARN_ON()`
- `#define ms_delay(x) asm(“magic assembly code to delay xms”)`
    - Use `mdelay()`
- Custom macros are an unneeded abstractions
    - Kernel maintainers know what existing macros do
    - Custom macros will be missed in search/replace

Don't Do Your Homework

- Large HW vendor developed new SOC
  - Needed I2C support to read MAC address configuration
  - Wrote custom chardev driver to access this information
    - drivers/i2c already defines a clean interface between I2C and users
- Different HW vendor has various crypto offload engines
  - Has written custom drivers in arch/$arch/security/ with custom ioctls()
    - drivers/crypto already exists
- Yet another HW vendor with a network device
  - Wrote custom MII handling code instead of using existing API
- Do your research before you start:
  - Read docs
  - Ask on mailing list
  - Use the source

You're Confusing Me!!

- You said:
  - Creating an abstraction layer is bad
  - Not abstracting things is bad
  - Abstracting things with custom macros is bad
  - I should work with community to create abstraction layer
  - **Which one is it?**
    - **It depends on the type of abstraction**
      - Abstracting HW capabilities into common interfaces is good
        - (Up to a point....)
      - Abstracting away kernel interface with custom interface is bad

Genius: Let's Implement Userspace in the Kernel!

- HW vendor's reference platform port:
  - Needed to load device firmware from flash
  - Flash is formatted using FAT
  - Kernel driver:
    - Mounts flash
    - Opens configuration file
    - Loads MAC address
    - Loads firmware
  - “We need to initialize HW before it can be used”
  - !!This is what initramfs is for!!
- **Thou shalt not access file system contents from kernel**

The “I Am Smarter Than You” Strategy

- “This is all part of what responsible release management is about. I was the junior whiz kid in professional release management teams before starting $company. I listened to my elders and learned from them. My standards for professional conduct in this arena are higher than yours as a result of that. You are a bunch of young kids who lack professional experience in release management.”

Don't Directly Participate

- Hire team of people to work on Linux drivers, subsystems, etc
- Filter all upstream contribution through one person
  - Who cannot answer all the questions because he/she did not write code
  - Who must go back and forth between original developer and community
- “My $customer sent me this patch to solve problem X”
  - Release patch but don't explain how problem found
    - Developer's can't reproduce
    - Maybe original assumptions are wrong
    - We can't guess...so we'll probably ignore you

The Other OS Does it Strategy

- $other_os provides $feature
- $other_os has larger market share
- Here's a patch implementing $feature for Linux
- Who cares if it makes sense to have $feature in kernel?
- “How about having a simple Game API like SDL included in the Kernel and officially announce the promise to change it only once every couple of years?”

Tie Code to Reference Platform

- Common mistake by embedded chipset vendors
  - Linux support done for HW validation purposes
    - Code written specifically for reference platform to get it done quickly
    - Hard coded addresses, IRQ routing, etc
    - No differentiation between CPU features and platform features
- Drivers that assume only one device per system
  - Might seem realistic, but you never know what end users might
- Code needs to be portable/extensible to new platforms

It Works on X86, so It Must be OK!

- Bad:

```
virt = ioremap(HW_ADDRESS);
...
irq_status = *(virt + IRQ_STATUS_REG_OFFSET)
```

  - It will work on x86 (most of the time)
  - There may be architecture or platforms workarounds
    - I/O operation may be series of accesses across special registers

- Good:
  - Your code:

```
virt = ioremap(HW_ADDRESS);
...
irq_status = readl(virt + IRQ_STATUS_REG_OFFSET);
```

  - Kernel API:

```
#define readl(address) do {
    if (requires_special_fixup(address))
        do_special_hw_fixup(address);
        return special_hw_read(address);
    }
```

The Hypothetical System

- Note: Following are paraphrased:
  - “Our customers are going to be running on systems with 1000s of disks. Boot up and discover time will take too long b/c udev is calling fork() and this unacceptable to our customers. The CGL spec requires such and such timing. We've rewritten hotplug handling and replaced udev.”
  - “Show us the numbers”
  - “We don't have any”
  - “Go away”
  - Repeat
- In the end, udev got rewritten to deal with forking issues
- Idea was right, but..
  - We're not theorists. We want real applications, real data
  - Reality trumps assumptions and specifications

TRUE != b_win32CodeIsSoMuchFunToRead

```
int nNIRLP_open (struct inode *inode, struct file *filep)
{
    struct nNIRLP_tDriverContext *context = NULL;

    int minor = MINOR(inode->i_rdev);

    tStatus status;
    tStatus_set (status, 0);

    nNIRLP_printDebug("nNIRLP_open(inode (%p), file (%p))\n", inode, filep);
    nNIRLP_printDebug("minor %i\n", minor);

    if (0 != minor)
        return -ENODEV;

    context = nNIRLP_tDriverContext_create (&status);
    if ( tStatus_isNotFatal(status) )
    {
        filep->f_op = &nNIRLP_fops;
        filep->private_data = (void *)context;
    }
        return status;
}
```

Summary

- Release Early, Release Often
  - If it boots, ship it!
- Understand that there is more than just your HW/device/stack
  - Your code may have ramifications you can't see
- Follow existing APIs and coding standards
- Treat the community as an extension of your team
  - Listen to feedback
  - Work with them to add changes you need to kernel
  - Provide data so they can make decisions
  - Ask questions to the right people: kernelnewbies.org
  - Act courteously
  - Let your engineers interact with the community
    - Send them to LCA, OLS, etc