Joining the herd of cats: how to work with the kernel development process

Jonathan Corbet
corbet@lwn.net

I:
Introduction

Why?

-
  - “There are a number of very good Linux kernel developers, but they tend to get outshouted by a large crowd of arrogant fools. Trying to communicate user requirements to these people is a waste of time. They are much too 'intelligent' to listen to lesser mortals.”
  -
    - -- Jack O'Quin, Linux audio developer

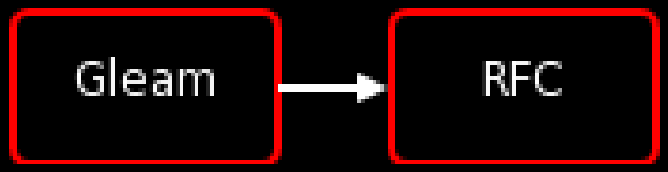
II:
Process
issues

Do: understand the patch lifecycle

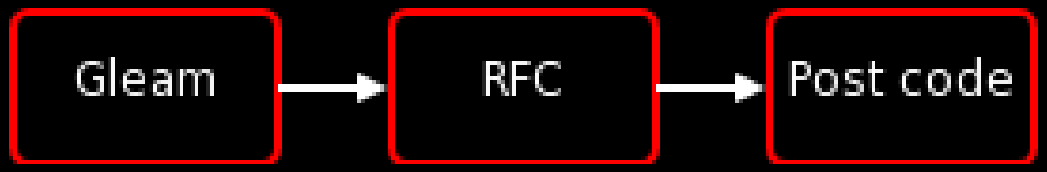
Much developer pain results from a failure to understand how code gets into the kernel.

Patch lifecycle: the beginning

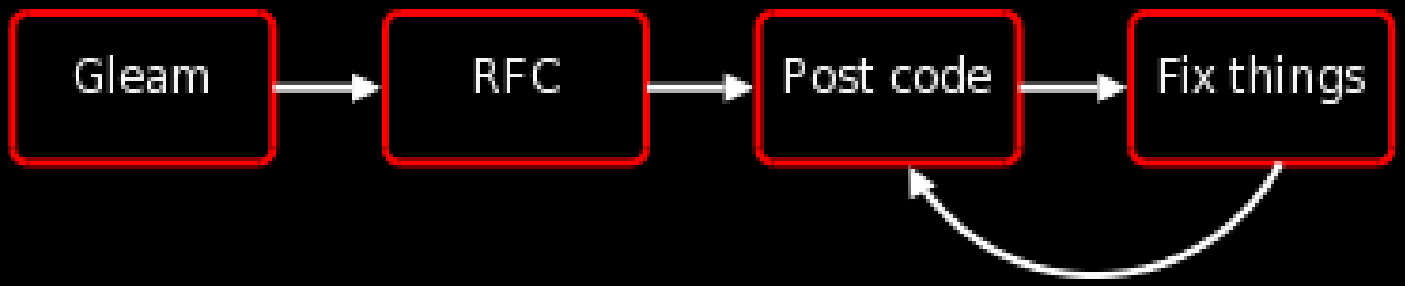
Patch Lifecycle: the RFC



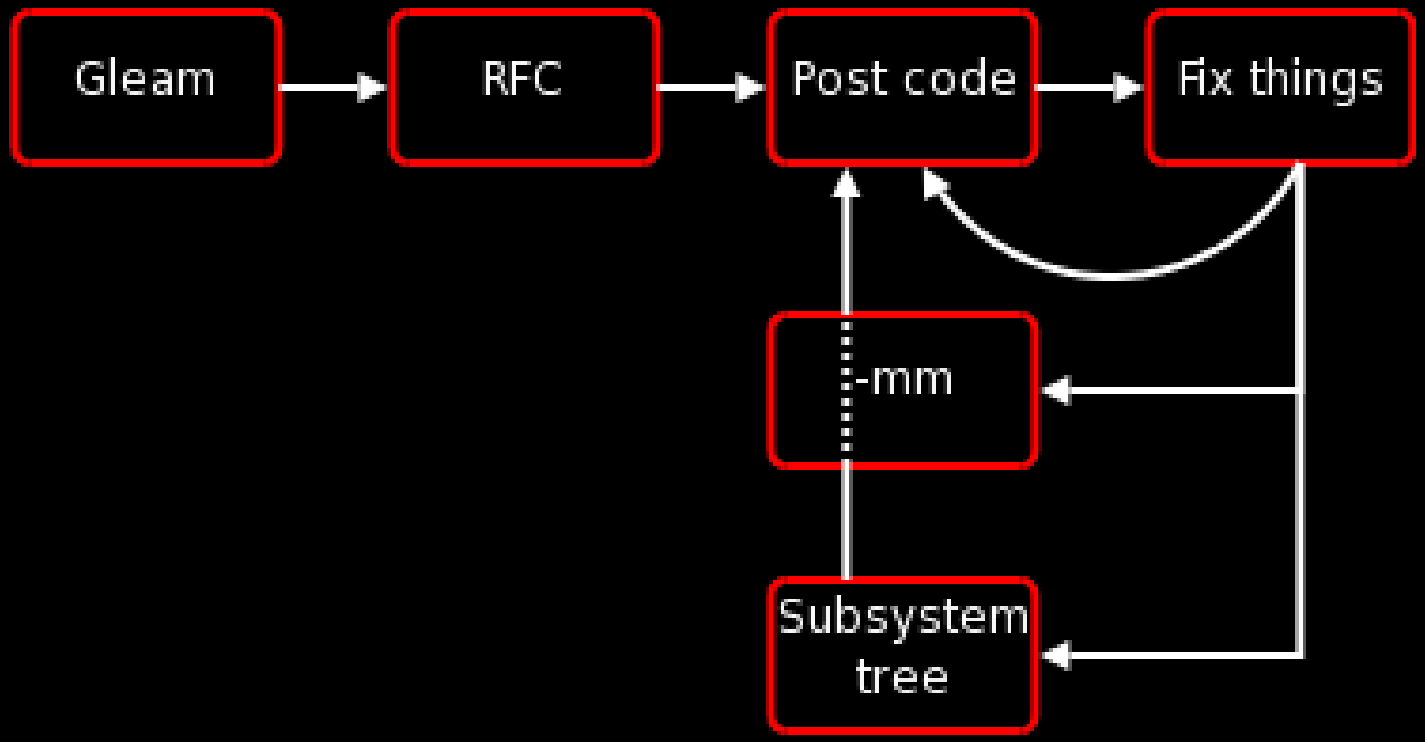
Patch lifecycle: first code



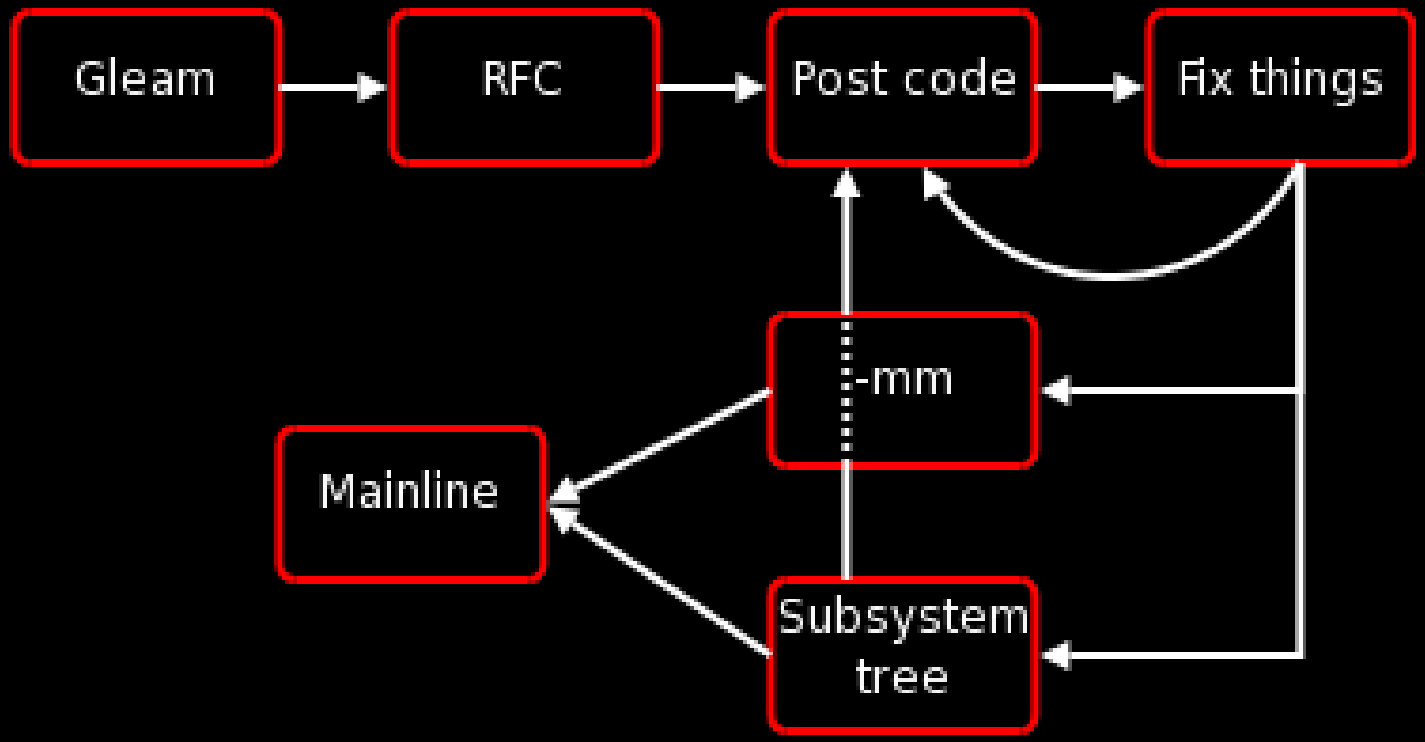
Patch lifecycle: repairs



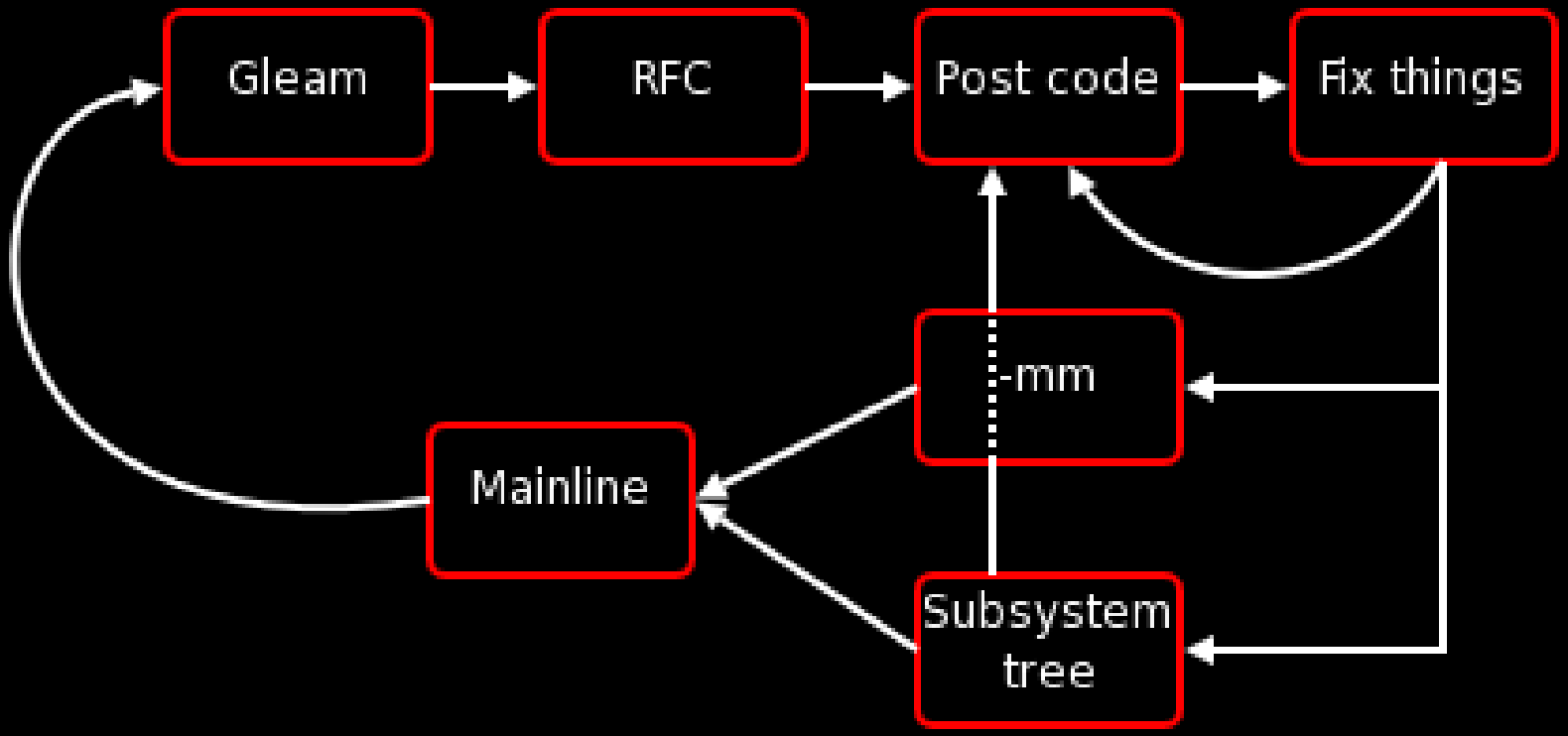
Patch lifecycle: wider testing



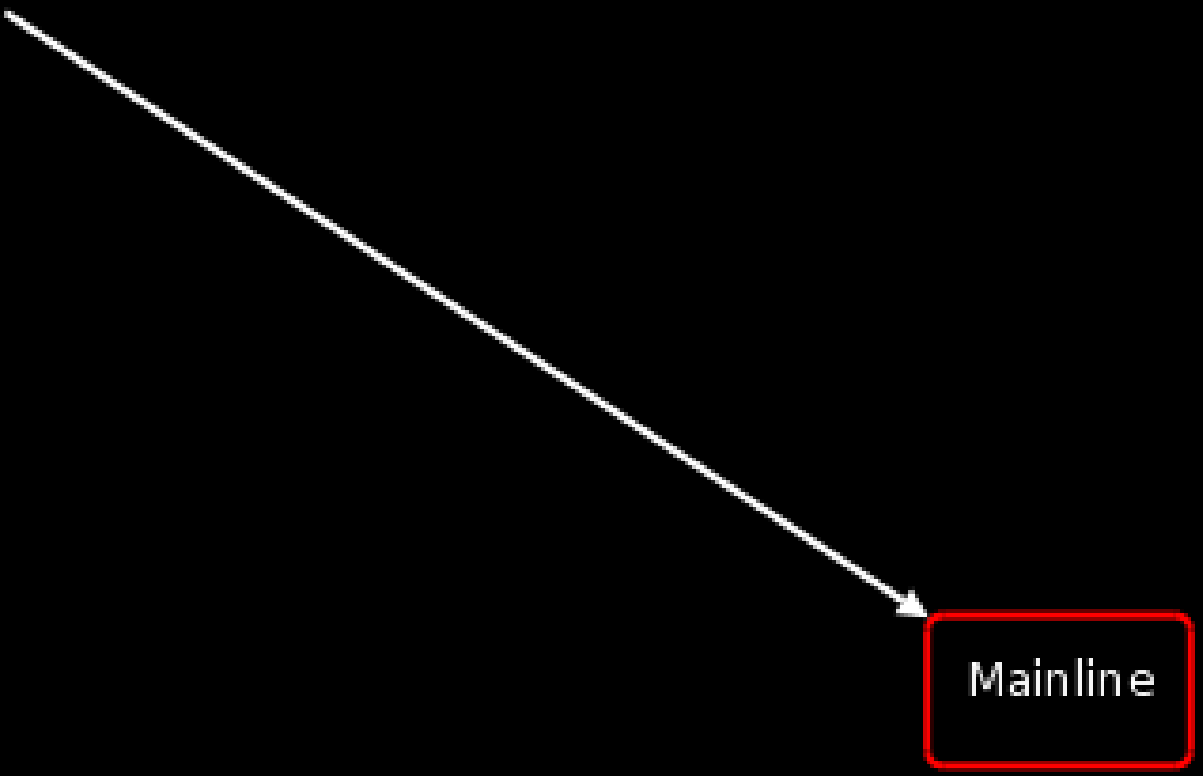
Finally: into the mainline



Patch lifecycle: repeat



Lifecycle: the corporate view



Do: communicate early

Let the community know what you are doing

Avoid duplication

Avoid mistakes

Do: release early

Big vendor mistake:

> “We'll release the code after it passes internal QA”

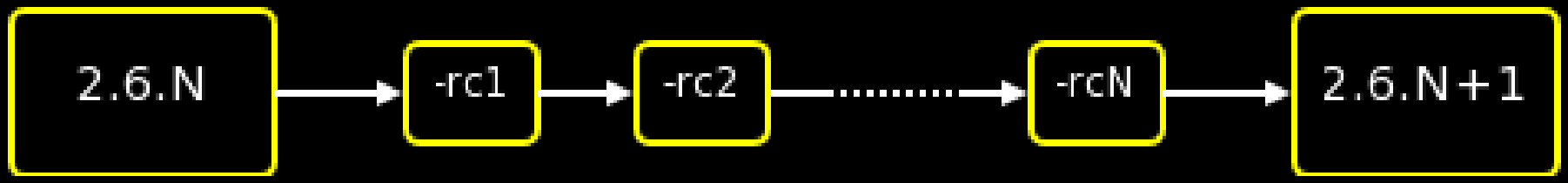
By then it is too late

Do: expect to make changes

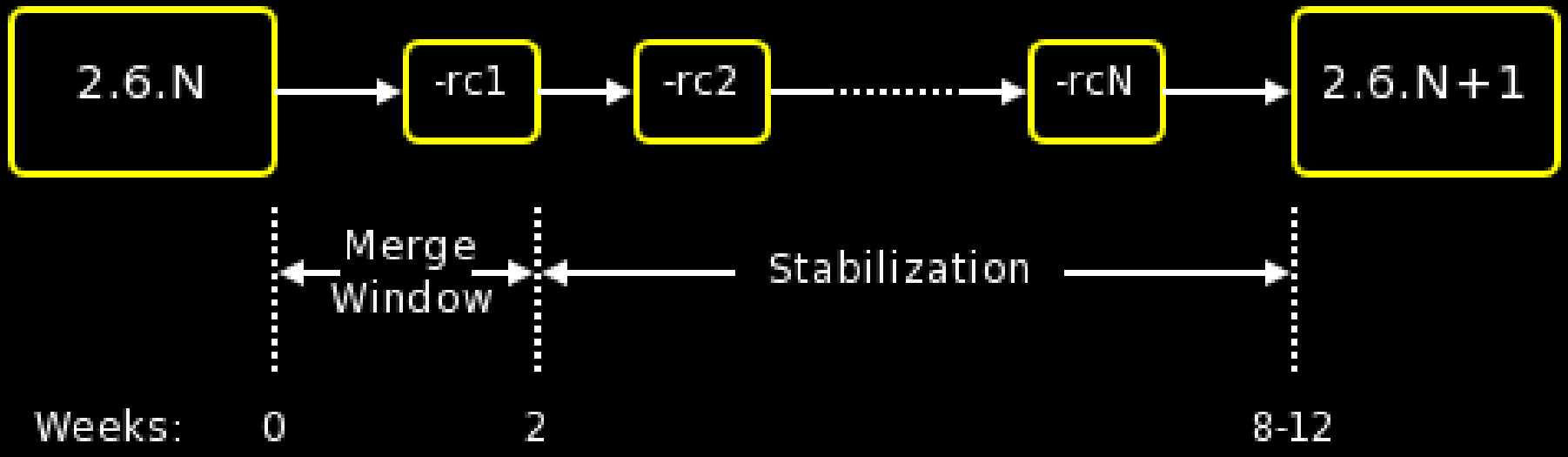
No initial code submission is perfect

Kernel developers have different goals

The kernel release cycle



The kernel release cycle



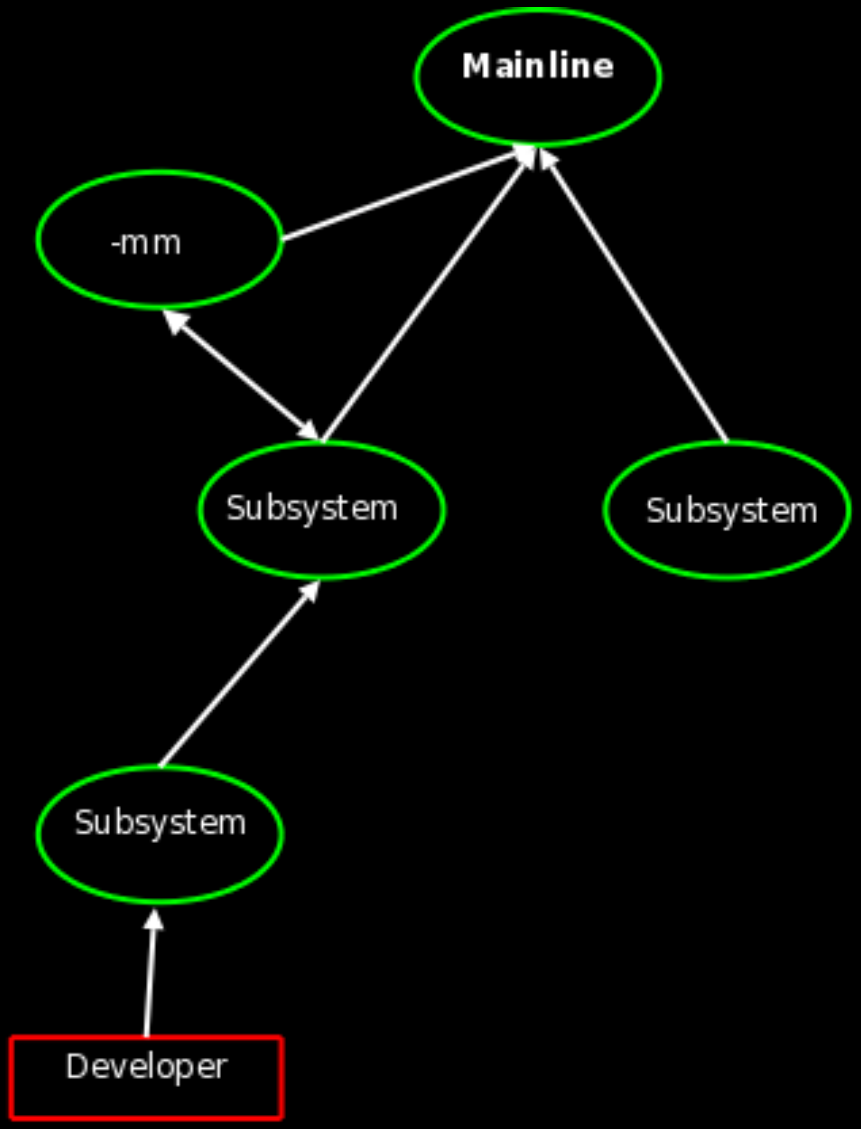
Do: observe the merge window

“I'm really fed up with having to pull big changes after the merge window, because it just doesn't seem to let up. I'm going to go postal on the next maintainer who doesn't understand what 'merge window' and 'fixes only' means”

-- Linus Torvalds

III:
Patch submission

The patch submission path



The gatekeepers

Non-author signoffs in 2.6.22

Andrew Morton	19%
Linus Torvalds	18%
David Miller	9%
Paul Mackerras	5%
Jeff Garzik	5%
Greg Kroah-Hartman	3%
Mauro Carvalho Chehab	3%
Andi Kleen	3%
James Bottomley	2%
Jaroslav Kysela	2%
Russell King	2%
Stefan Richter	2%
Len Brown	2%
John Linville	1%

Do: send patches to the maintainer

Look in MAINTAINERS if need be

Cc anybody else who might have an interest

Find the correct mailing list

linux-kernel is not the right place for all patches

Example: networking patches go to netdev

See:

- MAINTAINERS
- vger.kernel.org/vger-lists.html

On mailing lists

Ask whether you really need to read linux-kernel
- Many of us do

Consider a subsystem list instead

If so, read it sparingly
- Look for interesting topics and contributors

Don't: send multipurpose patches

Patches should:

- Be small (if possible)
- Do exactly one thing

If you have a big change:

- Split it into independent pieces

Do: send bisectable patches

“git bisect” is a great tool for finding regressions
- Binary search on the patch stream

To support bisect:
- Your patch series must work after every patch

Do: take care in submitting patches

- Use diff -u
- No MIME attachments
- Describe them properly
  - A one-line summary at top
  - Longer description below (if needed)
  - Justify the patch
- Include a Signed-off-by: line
- Avoid word-wrapping
  - Thunderbird is especially bad here

See:

- Documentation/SubmittingPatches

Do: listen to reviewers

Reviewing patches is hard, thankless work

When a reviewer makes a comment

- Say “thanks”
- Respond politely
- Fix the problem (or justify the current code)

Don't: attack reviewers

...even if they are rude

Don't: take criticism personally

Patch reviewers do not hate you

They do not hate your company

They do not hate your employees

Requests for major changes

Reviewers may ask for big changes
- Push functionality into higher layers
- Reimplement major functionality

Their goals are different than yours
- Long-term maintainability is key

Try to accommodate these requests
- They usually make sense in the long term

IV: Coding issues

Do: follow the coding style

Documentation/CodingStyle

Do: avoid unnecessary abstractions

Things to avoid:

- HAL layers
- Unused parameters “just in case”
- Single-line functions

API stability

There is no stable internal kernel API
- Get used to it

Ways to cope
- Get your code into the mainline
- http://lwn.net/Articles/2.6-kernel-api

Don't: add multi-version code

Support the current mainline kernel

...and no others

Don't add regressions

...even to fix something else

Don't: assume all the world is a PC

Linux runs on all kinds of systems
- handhelds to supercomputers
- 32/64 bit
- Single processor through thousands of processors
- A few dozen architectures

Your code should build and work everywhere

In particular:
- Get your locking right from the beginning

Do: clean up your messes

Breaking an internal API is OK

...if there is a good reason for it

But:

you have the responsibility to fix in-tree code

Do: avoid silly mistakes

Use the tools:

- gcc -W
- lockdep
- fault injection framework
- slab poisoning
- sparse
- ...

IV:
Final
notes

Don't: submit tainted code

Read the Developer's Certification of Origin
- Be sure you mean it

Be very careful with reverse engineering
- Chinese wall approach should be used

Don't: ship binary-only modules

Legality of these modules is dubious

The benefit is even more dubious

Respect your customers: give them the source

Do: use the resources available

There is information and help out there

- kernelnewbies.org
- Kernel mentors
- Documentation/HOWTO
- LWN

Do: join in and have fun

Questions....?