Who owns the interface?

Linux Plumbers Conference
Portland, OR, USA; 18 Sep 2008

Michael Kerrisk
Fellow, Linux Foundation
mtk.manpages@gmail.com

http://www.kernel.org/pub/linux/docs/manpages
git://git.kernel.org/pub/scm/docs/man-pages/man-pages.git
http://linux-man-pages.blogspot.com
Overview

- What does it mean to own the interface?
- Who could claim to own the interface?
- Arguments for *(pros)* and against *(cons)* each claim
- Problems resulting from the Linux ownership model
- Conclusion/Discussion
What does it mean to “own the interface”?

• Who determines what gets included in the interface, and what doesn’t?

• How is “in the interface” even defined?
  – By the implementation (code below the interface)?
  – By the use cases (code above the interface)?
  – By a specification?
Traditional Interface Ownership Model

- (Strong) degree of centralization
- Interface owned by vendor
  - e.g., AT&T's SVID
- Or by a standards body
  - e.g., POSIX.1
But, who does own the Linux interface?
Candidates

• Austin
• Kernel developers
• Users
• glibc
• LSB
• LTP
• man-pages
Is it the Austin Group?

Pros

- They define standard (POSIX.1/SUS) that describes much of the Unix API
- Linux kernel (and glibc) strive to conform to standard

Cons

- Surprisingly long list!
The Austin Group - Cons

• Linux doesn’t implement all POSIX.1 interfaces
• Many Unix/Linux interfaces not specified in POSIX.1
  - e.g., `setresuid()`, `mincore()`, `brk()`, `flock()`, `settimeofday()`
• Linux provides extensions
  - e.g., `inotify`, `epoll`, `/proc`, capabilities, extended attributes, NUMA, scheduler affinity, various process resource limits (`setrlimit()`), `sendfile()`, etc.
The Austin Group - Cons (2)

- Linux may deliberately violate the standard
  - e.g., `link()` to a symlink
- Sometimes, an implementation accidentally violates the standard
  - e.g., successful `sched_setscheduler()` should return previous scheduling policy, not 0
  - Define non-conformant details as being “part of the Linux interface”? 
The Austin Group - Cons (3)

• SUS/POSIX.1 leaves many things unspecified
  – Examples:
    • Does `select()` modify its `timeout` argument?
    • Does `signal()` provide reliable semantics?
    • Do `setitimer()` and `alarm()` interact?
    • What is maximum size of a Unix domain datagram?
    • What are precise semantics of `vfork()`?
    • What is resolution of system clock?
    • Does `longjmp()` restore signal mask to state at time of `setjmp()`?

• Left to implementation to define (or not care about) them
The Austin Group?

- Many parts of Linux interface clearly not owned by Austin
Is it the Kernel Developers?

Pros

- *Surely* it is the kernel developers?
- After all, they implement the interface!
The Kernel Developers - Cons

• What about difference between implementation and intention?
  – What happens if the kernel-user interface has bugs?
    • Kernel developers write a lot of bugs in interfaces
    – Unforeseen uses of interface (see “Users” later)

• glibc mediates between kernel and user
  – e.g., syscall wrapper may provide different behavior from raw syscall
  – More on glibc later
The Kernel Developers – Cons (2)

- If implementation defines interface, how can user know what definition is?
- Read the source???
- Wrong answer for many reasons...
  - Takes too long
  - Doesn't tell us whether a feature is intended or a bug
  - More on reading source later...
Is it the Users?

Cons:
• How could it possibly be the users? They didn’t write the interface!

Can there be any pros?
The Users - Pros

Bugs!

- Suppose an interface contains a bug
- What is “correct” definition of interface?
- The intention? or the implementation?
- Should we fix the bug?
- What if users already program around the bug?
  - Maybe better not fix...
  - Unless we can tell users they suck...
    - (i.e., intended behavior was documented)
  - Then, arguably, users have defined part of interface
The Users – Pros (2)

Reading the source, revisited

- Creates a “tight” specification
- Suppose implementer has some detail of interface behavior (“x”) that shouldn't be fixed in stone
  - maybe want to change it in future
- Don't want users to rely on detail “x” in their code
- Making users read the source:
  - Reveals all details of interface
  - Provides no warning against relying on detail “x”
  - Some users will write code that uses “x”
  - Again, users have defined part of interface
Is it the glibc Developers?

- Raw kernel syscall interface is like a newly built house: you want someone make it livable before you move in
- glibc mediates between kernel and user, providing syscall wrappers
Glibc - Pros

• Some wrappers do significant work on top of system calls
  - `stat()`, `readv()`, `writev()`, `pselect()`, `mq_getattr() / mq_setattr()`

• glibc implements many functions not based on syscalls
  - To users, these are part of Linux interface
Glibc - Cons

• See kernel “cons” -- bugs, unintended uses of API, etc
• For many syscalls, wrapper is trivial
  – glibc is transparently exposing kernel interface
• glibc doesn't provide wrappers for every system call
• What gets wrapped or not is a little arbitrary:
  – Did glibc folk notice new kernel interface?
  – Did they think it was worth wrapping?
• Sometimes, glibc developers deliberately choose not to wrap a syscall
  – \texttt{gettid()} (though thread IDs are needed by some syscalls)
Glibc – Cons (2)

- Many kernel interfaces other than syscalls
- Those interface are *not* mediated by glibc, e.g.:
  - `/proc`
  - `sysfs`
  - `ioctl()`
  - `netlink`
Is it LSB?

Pros

• LSB defines an ABI standard for Linux

Cons

• Many interfaces not specified (sometimes deliberately)
• Parts of some interfaces are deliberately unspecified
• LSB is largely just standardizing the implementation provided by kernel and glibc developers
Linux Test Project (LTP)?

Pros

• Tests embody a specification, in code

Cons

• Test coverage is not complete, and excludes glibc
• Tests may themselves be buggy
• LTP tests usually only added (well) after syscall is added to kernel
Is it me?

- The *man-pages* project documents kernel and glibc interfaces
- Documentation is contract between kernel (/ glibc) and user(?)
man-pages - Pros

• Documentation can describe the developer's *intention*
• Provides reference for determining where implementation deviates from intention
  - i.e., is this a bug?
• Documentation can loosen the specification:
  - Can say things like: you *can* do “x”, but if you do, the results are undefined
  - (cf. “Tight” specification that results from reading source)
man-pages - Cons

- Many things remain undocumented.
  - Does that mean that they are not part of the interface?
- Sometimes implementation is right and documentation is wrong :-(


Summary of Ownership

• Many groups have claims to ownership
• Some validity in each claim
• No group can claim exclusive ownership
• Distributed ownership is source of some problems
Problems resulting from distributed ownership

Many of the problems arise from a single point...

- How do we even know when an interface has been added or changed?
  - We == kernel developers, glibc developers, userland programmers, testers, LSB, man-pages
Problems resulting from distributed ownership

Consequences for documentation

- Documentation may be late (i.e., after implementation)
- Poor documentation
  - (esp. if implementer was not involved)
- No documentation
Problems resulting from distributed ownership

Consequences for testing

• Late and insufficient testing
• Insufficient pre-release testing →
• many bugs in released interfaces
  – `epoll`, `timerfd`, `utimensat()`, `signalfd()`
Problems resulting from distributed ownership

Consequences for interface design

- Insufficient design review before release
- Inconsistent interfaces
  - Rounding of args for `mlock()` and `remap_file_pages()`
- Poorly designed interfaces
  - Dnotify
- Design mistakes
  - `epoll_create()` nowadays ignores `size` arg
Problems resulting from distributed ownership

- Design mistakes
- Capabilities
- Divide *root* into many distinct pieces

```
CAP_AUDIT_CONTROL CAP_AUDIT_WRITE CAP_CHOWN CAP_DAC_OVERRIDE
CAP_DAC_READ_SEARCH CAP_FOWNER CAP_FSETID CAP_IPC_LOCK CAP_IPC_OWNER
CAP_KILL CAP_LEASE CAP_LINUX_IMMUTABLE CAP_MAC_ADMIN CAP_MAC_OVERRIDE
CAP_MKNOD CAP_NET_ADMIN CAP_NET_BIND_SERVICE CAP_NET_BROADCAST
CAP_NET_RAW CAP_SETFCAP CAP_SETGID CAP_SETPCAP CAP_SETUID CAP_SYS_ADMIN
CAP_SYS_BOOT CAP_SYS_CHROOT CAP_SYS_MODULE CAP_SYS_NICE CAP_SYS_PACCT
CAP_SYS_PTRACE CAP_SYS_RAWIO CAP_SYS_RESOURCE CAP_SYS_TIME
CAP_SYS_TTY_CONFIG
```

- Great! But which one do I (an implementer) use?
- Ahh! I know!
- **CAP_SYS_ADMIN**, the new *root*, 180 uses in 2.6.27-rc
Concluding thoughts

• Interfaces are contracts
• Cast in stone
• We live with them “forever”
• So: need to get them right, at the beginning
• Getting things right:
  – Requires some degree of planning and coordination
  – Probably more than we currently do.
• Linux may be evolution, but intelligent design might sometimes get us there better and faster
Discussion / Questions

- How do we know when an interface has been changed or added?