

FOSDEM 2016

How to design a Linux kernel interface

Michael Kerrisk

man7.org Training and Consulting

<http://man7.org/training/>

31 January 2016

Bruxelles / Brussel / Brussels

Who am I?

- Maintainer of Linux *man-pages* project since 2004
 - Documents kernel-user-space and C library APIs
 - 15k commits, 170 releases, author/co-author of 350+ of 990+ pages in project
- Quite a bit of design review of Linux APIs
- Lots of testing, lots of bug reports
- Author of a book on the Linux programming interface
- IOW: looking at Linux APIs a lot and for a long time



Theme is more about process
than technical detail



Outline

- 1 The problem
- 2 Think outside your use case
- 3 Unit tests
- 4 Specification
- 5 The feedback loop
- 6 Write a real application
- 7 A technical checklist
- 8 Concluding thoughts

Outline

- 1 **The problem**
- 2 Think outside your use case
- 3 Unit tests
- 4 Specification
- 5 The feedback loop
- 6 Write a real application
- 7 A technical checklist
- 8 Concluding thoughts

Implementation of APIs is the lesser problem

(Performance can be improved later;
bugs are irritating, but can be fixed)



API design is the big problem



Why is API design a problem?

- Hard to get right
- (Usually) can't be fixed
 - Fix == ABI change
 - User-space will break

- And...



Thousands of user-space
programmers will live with your
(bad) design for *decades*



Many kinds of APIs

- Pseudo-filesystems (`/proc`, `/sys`, `/dev/mqueue`, debugfs, configfs, etc.)
- Netlink
- Auxiliary vector
- Virtual devices
- Signals
- System calls \Leftarrow focus, for purposes of example
- Multiplexor syscalls (`ioctl()`, `prctl()`, `fcntl()`, ...)



Outline

- 1 The problem
- 2 Think outside your use case**
- 3 Unit tests
- 4 Specification
- 5 The feedback loop
- 6 Write a real application
- 7 A technical checklist
- 8 Concluding thoughts

Example: POSIX messages

- POSIX MQs: message-based IPC mechanism, with priorities for messages
 - *mq_open()*, *mq_send()*, *mq_receive()*, ...
 - Linux 2.6.6
- Usual use case: reader consumes messages (nearly) immediately
 - (i.e., queue is usually short)
- Kernel developers coded for usual use case



Example: POSIX messages

- Linux 3.5: a vendor developer raises ceiling on number of messages allowed in MQ
 - Raised from 32,768 to 65,536 to serve a customer request
- I.e., customer wants to queue **masses** of unread messages
- Developer notices problems with algorithm that sorts messages by priority
 - **Approximates to bubble sort(!)**
 - Will not scale well with (say) 50k messages in queue...
- Among a raft of other MQ changes, developer fixes sort algorithm



When designing APIs, remember:

User-space programmers are
endlessly inventive



Moral 1: try to imagine the ways
in which an army of inventive
user-space programmers might
(ab)use your API



Is this such a big deal?

A performance bug got found and fixed. So what?

(but there's more...)



3.5 MQ changes also **broke user space** in at least two places

- Introduced **hard limit of 1024** on `queues_max`, disallowing even superuser to override
 - Fixed by commit `f3713fd9c` in Linux 3.14, and in `-stable`
- Semantics of value exported in `/dev/mqueue QSIZE` field **changed**
 - Count now includes user data **and kernel overhead** bytes
 - <http://thread.gmane.org/gmane.linux.man/7050>
 - Fixed (at last) in Linux 4.2



Moral 2: without unit tests you
will screw up someone's API



Outline

- 1 The problem
- 2 Think outside your use case
- 3 Unit tests**
- 4 Specification
- 5 The feedback loop
- 6 Write a real application
- 7 A technical checklist
- 8 Concluding thoughts

- To state the obvious, unit tests:
 - **Prevent behavior regressions** in face of future refactoring of implementation
 - Provide **checks that API works as expected/advertised**



Regressions happen more often
than you'd expect



Examples of regressions

- Linux 2.6.12 silently changed meaning of *fcntl()* `F_SETOWN`
 - No longer possible to target signals at specific thread in multithreaded process
 - Change discovered many releases later; too late to fix
 - Maybe some new applications depend on new behavior!
 - ⇒ Since Linux 2.6.32, we have `F_SETOWN_EX` to get old semantics



Examples of regressions

- Inotify `IN_ONESHOT` flag
 - (inotify == filesystem event notification API added in Linux 2.6.13)
 - `IN_IGNORED` event informs user when watch is automatically dropped for various reasons
 - By design, `IN_ONESHOT` did **not** cause an `IN_IGNORED` event when watch is dropped after one event
 - Because user **knows** that watch will last for just one events
 - Inotify code was refactored during fanotify implementation (early 2.6.30's)
 - From 2.6.36, `IN_ONESHOT` *does* cause `IN_IGNORED`



Does it do what it says
on the tin?

(Too often, the answer is no)

Does it do what it says on the tin?

- Inotify `IN_ONESHOT` flag (2.6.13)
 - Provide **one notification** event for a monitored object, **then disable monitoring**
 - Tested in 2.6.15; simply did not work (no effect)
 - \Rightarrow zero testing before release...
 - Fixed in 2.6.16
- Inotify event coalescing
 - Successive identical events (same event type on same file) are combined
 - Saves queue space
 - Before Linux 2.6.25, a new event would be coalesced with item at **front** of queue
 - I.e., with oldest event rather than most recent event
 - Clearly: minimal pre-release testing



Does it do what it says on the tin?

- `recvmsg()` system call (linux 2.6.33)
 - Performance: receive multiple datagrams via single syscall
 - *timeout* argument added late in implementation, after reviewer suggestion
- Intention versus implementation:
 - **Apparent** concept: place timeout on receipt of complete set of datagrams
 - **Actual** implementation: timeout *tested only after receipt of each datagram*
 - Renders timeout useless...
- Clearly, no serious testing of implementation
 - Also, confused implementation with respect to use of `EINTR` error after interruption by signal handler
 - <http://thread.gmane.org/gmane.linux.kernel/1711197/focus=6435>



Probably, all of these problems could have been avoided if there were unit tests

Writing a new kernel-user-space
API? \Rightarrow include unit tests

Refactoring code under existing
API that has no unit tests? \Rightarrow
please write some



Where to put your tests?

- Historically, only real home was LTP (Linux Test Project), but:
 - Tests were out of kernel tree
 - Often only added after APIs were released
 - Coverage was only partial
- *kselftest* project (started in 2014) seems to be improving matters:
 - Tests reside in kernel source tree
 - Paid maintainer: Shuah Khan
 - Wiki: <https://kselftest.wiki.kernel.org/>
 - Mailing list: linux-api@vger.kernel.org



But, how do you know what to test if there is no specification?



Outline

- 1 The problem
- 2 Think outside your use case
- 3 Unit tests
- 4 Specification**
- 5 The feedback loop
- 6 Write a real application
- 7 A technical checklist
- 8 Concluding thoughts

“Programming is not just an act of telling a computer what to do: it is also an act of telling other programmers what you wished the computer to do. Both are important, and the latter deserves care.”

Andrew Morton, March 2012



Fundamental problem behind
(e.g.) *recvmsg()* *timeout* bugs:

no one wrote a specification
during development or review



A test needs a specification

`recvmsg()` *timeout* argument needed a specification; something like:

- The *timeout* argument implements three cases:
 - ① *timeout* is `NULL`: the call blocks until *vlen* datagrams are received.
 - ② *timeout* points to `{0, 0}`: the call (immediately) returns up to *vlen* datagrams if they are available. If no datagrams are available, the call returns immediately, with the error `EAGAIN`.
 - ③ *timeout* points to a structure in which at least one of the fields is nonzero. The call blocks until either:
 - (a) the specified timeout expires
 - (b) *vlen* messages are received

In case (a), if one or more messages has been received, the call returns the number of messages received; otherwise, if no messages were received, the call fails with the error `EAGAIN`.

- If, while blocking, the call is interrupted by a signal handler, then:
 - if 1 or more datagrams have been received, then those datagrams are returned (and interruption by a signal handler is not (directly) reported by this or any subsequent call to `recvmsg()`).
 - if no datagrams have so far been received, then the call fails with the error `EINTR`.



Specifications have numerous benefits:

- Provides target for implementer
- Without specification, how can we differentiate implementer's *intention* from actual *implementation*
 - IOW: how do we know what is a bug?
- Allow us to write unit tests
- Allow reviewers to more easily understand and critique API
 - \Rightarrow will likely increase number of reviewers



Where to put your specification?

- At a minimum: in the commit message
- To gain good karma: a *man-pages* patch
 - <https://www.kernel.org/doc/man-pages/patches.html>



Man pages as a test specification

A well written man page often suffices as a test specification for finding real bugs:

- *utimensat()*:
http://linux-man-pages.blogspot.com/2008/06/whats-wrong-with-kernel-userland__30.html
- *timerfd*:
<http://thread.gmane.org/gmane.linux.kernel/613442>



Outline

- 1 The problem
- 2 Think outside your use case
- 3 Unit tests
- 4 Specification
- 5 The feedback loop**
- 6 Write a real application
- 7 A technical checklist
- 8 Concluding thoughts

The problem

- Probably 6+ months before your API appears in distributions and starts getting used in real world
- Worst case: only then will bugs be reported and design faults become clear
- But that's too late...
 - (Probably can't change ABI...)
- Need as much feedback as possible **before** API is released



Strive to shorten worst-case
feedback loop



Publicize API design
as widely + early as possible



Shortening the feedback loop

Ideally, do all of the following before API release:

- Write a detailed **specification**
- Write **example programs** that fully demonstrate API
- Email relevant mailing lists and, especially, relevant people
- CC *linux-api@vger.kernel.org*
 - As per `Documentation/SubmitChecklist...`
 - Alerts interested parties of API changes:
 - C library projects, *man-pages*, LTP, trinity, kselftest, LSB, tracing projects, and user-space programmers
 - <https://www.kernel.org/doc/man-pages/linux-api-ml.html>
- For good karma + more publicity: write an LWN.net article
 - Good way of **reaching end users** of your API
 - Ask readers for feedback
 - <http://lwn.net/op/AuthorGuide.lwn>



Of course

- Of course, you'd only do all of this if you wanted review and cared about long-term health of the API, right?
 - My inner cynic: in some cases, implementers actively avoid these steps, to minimize patch resistance
- Subsystem maintainers: watch out for developers who avoid these steps



Outline

- 1 The problem
- 2 Think outside your use case
- 3 Unit tests
- 4 Specification
- 5 The feedback loop
- 6 Write a real application**
- 7 A technical checklist
- 8 Concluding thoughts

Example: inotify

- Filesystem event notification API
 - Detect file opens, closes, writes, renames, deletions, etc.
- *A Good Thing*TM...
 - Improves on predecessor (*dnotify*)
 - Better than polling filesystems using *readdir()* and *stat()*
- But it should have been *A Better Thing*TM



Writing a “real” inotify application

- Back story: I thought I understood inotify
- Then I tried to write a “real” application...
 - Mirror state of a directory tree in application data structure
 - 1500 lines of C with (lots of) comments
 - http://man7.org/tlpi/code/online/dist/inotify/inotify_dtree.c.html
 - Written up on LWN (<https://lwn.net/Articles/605128/>)
- And understood all the work that inotify still leaves you to do
- **And what inotify could perhaps have done better**



The limitations of inotify

A few among several tricky problems when using inotify:

- Event notifications don't include PID or UID
 - Can't determine who/what triggered event
 - It might even be you
 - *Why not supply PID / UID, at least for privileged programs?*
- Monitoring of directories is not recursive
 - Must add new watches for each subdirectory
 - (But, probably an unavoidable limitation of API)
 - Can be expensive for large directory tree ⇒ see next point



The limitations of inotify

File renames generate `MOVED_FROM+MOVED_TO` event pair

- Useful: provides old and new name of file
- But two details combine to create a problem:
 - `MOVED_FROM+MOVED_TO` not guaranteed to be consecutive
 - No `MOVED_TO` if target directory is not monitored
 - Can't be sure if `MOVED_FROM` will be followed by `MOVED_TO`
- \Rightarrow matching `MOVED_FROM+MOVED_TO` must be done heuristically
 - Unavoidably racey, leading to possible matching failures
- Matching failures \Rightarrow treated as tree delete + tree re-create (expensive!)
- *User-space handling would have been much simpler, and deterministic, if `MOVED_FROM+MOVED_TO` had been guaranteed consecutive by kernel*



Only way to discover design problems in a new nontrivial API is by writing complete, real-world application(s)

(before the API is released in mainline kernel...)

API limitations should be rectified, or at least clearly documented, before API release...



Outline

- 1 The problem
- 2 Think outside your use case
- 3 Unit tests
- 4 Specification
- 5 The feedback loop
- 6 Write a real application
- 7 A technical checklist**
- 8 Concluding thoughts

A few technical points that frequently come up in Linux API design



New system calls should allow for extensibility

- Allow for future extensibility
- Possibility 1: *flags* bit-mask argument
 - Examples of past failures, and their fixes:
 - *futimesat()* ⇒ *utimensat()*
 - *epoll_create()* ⇒ *epoll_create1()*
 - *renameat()* ⇒ *renameat2()*
 - And **many** more
 - <https://lwn.net/Articles/585415/>
- Possibility 2: package arguments in extensible structure
 - Additional *size* argument allows kernel to determine “version” of structure
 - [Documentation/adding-syscalls.txt](#) (since Linux 4.2)



Undefined arguments and flags must be zero

- APIs should ensure that reserved/unused arguments and undefined bit flags are zero
 - `EINVAL` error
 - Allows user-space to test if feature is supported
- Failing to do this, allows applications to pass random values to args/masks
 - **Many** historical syscalls failed to do this check
- Those applications may fail when future kernels define meanings for those arguments/bits
- Conversely: you may not be able to define meanings, because user-space gets broken
 - (This has happened)
 - <https://lwn.net/Articles/588444/>



File descriptors syscall should support O_CLOEXEC

- Causes file descriptor (privileged resource) to be closed during *exec()* of new program
- Historical pattern

```
fd = open(pathname, ...);
flags = fcntl(fd, F_GETFD);
flags |= O_CLOEXEC;
fcntl(fd, F_SETFD, flags);
```

- Multithreaded programs have a race...
 - If another thread does *fork()* + *exec()* in middle of above steps, FD leaks to new program
- 2.6.27, + 2.6.28 added raft of replacements for existing syscalls to allow O_CLOEXEC to be set at FD creation time
 - E.g., *epoll_create1()*, *inotify_init1()*, *dup3()*, *pipe2()*
- New system calls that create FDs should support O_CLOEXEC



Syscalls with timeouts should allow absolute timeouts

- Some blocking system calls allow setting of timeout to limit blocking period
- In many cases, syscalls support **relative** timeouts
 - Specify timeout relative to present time (e.g., wait up to 10s)
 - Simple and convenient, often what we want
- But... subject to creep on restart after interruption by signal handler
 - (Because each restart can oversleep)
- ⇒ also include support for absolute timeouts measured on `CLOCK_MONOTONIC` clock
 - E.g., `clock_nanosleep()` `TIMER_ABSTIME` flag
 - (Added precisely to fix creeping sleep problem of `nanosleep()`)



Avoid extending multiplexor system calls

- Disfavor adding new commands to existing multiplexor syscalls
 - *prctl()*, *fcntl()*, *ioctl()*
- No type checking of arguments
- Becomes messy when you later decide to extend feature with new options
 - *seccomp*: (`/proc` API \Rightarrow) *prctl()* \Rightarrow *seccomp()* system call



- General concept:
 - Divide power of root into small pieces
 - Replace set-UID-root programs with programs that have capabilities attached
 - Less harm can be inflicted if program is compromised

- The problem for kernel developers: what capability should I use for my new privileged operation?
 - Read *capabilities(7)*
 - Choose a capability that governs similar operations
 - Or, if necessary, devise a new capability
 - Don't choose `CAP_SYS_ADMIN`
 - “The new root”
 - 1/3 of all capability checks in kernel are `CAP_SYS_ADMIN`
 - <https://lwn.net/Articles/486306/>
- Send in a *man-pages* patch for *capabilities(7)*



64-bit arguments and structure fields

- Take care when dealing with 64-bit arguments and structure fields
 - Daniel Vetter, “Botching up ioctls”,
<http://blog.ffwll.ch/2013/11/botching-up-ioctls.html>
 - Jake Edge, “System calls and 64-bit architectures”
<http://lwn.net/Articles/311630/>



Test, test, test

- “show me a newly released kernel interface, and I’ll show you a bug”
- Yes, bugs are fixable, but...
- Bug fixes **are** ABI changes
 - (Fixed) bad bugs may require user-space to special-case based on kernel version
 - Worst case: cost of keeping buggy ABI < cost of breaking existing ABI



Outline

- 1 The problem
- 2 Think outside your use case
- 3 Unit tests
- 4 Specification
- 5 The feedback loop
- 6 Write a real application
- 7 A technical checklist
- 8 Concluding thoughts**

Jeff Layton, OFD locks, Linux 3.15 (commit 5d50ffd7c31):

- “Open file description locks”
- Fix serious design problems with POSIX record locks
 - (POSIX record locks are essentially unreliable in the presence of any library that works with files)
- Did everything nearly perfectly, in terms of developing feature



Jeff Layton, OFD locks, Linux 3.15 (commit 5d50ffd7c31):

- Clearly explained **rationale** and changes in commit message
- Provided example programs
- Publicized the API
 - Mailing lists
 - LWN.net article (<http://lwn.net/Articles/586904/>)
- Wrote a man pages patch
 - (Feedback led to renaming of constants and feature)
- Engaged with glibc developers (patches for glibc headers + manual)
 - Refined patches in face of review
 - Maintainers were unresponsive \Rightarrow resubmitted *many* times
- Made it all look simple



Getting involved in kernel development

- Want to get involved in kernel development?
- Review / testing / documenting kernel-userpace APIs is one of the easier paths
- There's a **lot** of low-hanging fruit...
 - Design errors
 - Finding bugs
 - Fixing API bugs / extending APIs



Getting involved in kernel development

- Make developer(s) explain API and its use cases
 - Kernel developers are often quite bad at:
 - Explaining...
 - Explaining from a user-space perspective
- Asking naive questions often uncovers interesting info
 - And leads to ideas for improvements...
- Documenting an API is a good way of finding bugs
 - Can't write good documentation without testing (i.e., understanding) API
- Finding bugs gives you a chance hack to fix them
 - E.g., Heinrich Schuchardt cowrote *fanotify(7)* man page
 - Found a good six bugs while doing so
 - Wrote patches to fix most of them



Thanks!

mtk@man7.org

Slides at <http://man7.org/conf/>

Linux/UNIX system programming training (and more)
<http://man7.org/training/>

The Linux Programming Interface, <http://man7.org/tlpi/>

