Linux Piter 2019

Once upon an API

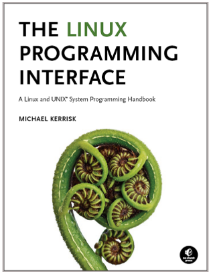
Michael Kerrisk, man7.org © 2019

mtk@man7.org

5 October 2019, St Petersburg, Russia

Who am I?

- Maintainer to Linux *man-pages* project since 2004
  - ≈1050 pages, mainly for system calls & C library functions
    - https://www.kernel.org/doc/man-pages/
    - (I wrote a lot of those pages...)
- Author of a book on the Linux programming interface
  - http://man7.org/tlpi/
- **Trainer**/writer/engineer
  `http://man7.org/training/`
- Email: `mtk@man7.org`
  Twitter: `@mkerrisk`


THE LINUX PROGRAMMING INTERFACE
A Linux and UNIX® System Programming Handbook
MICHAEL KERRISK

Why am I here?

I'd like to see APIs done better

because...

generally, a misdesigned API is unfixable

(we might break some binary that depends on the broken-ness)

and...

large numbers of user-space developers must live with the broken-ness for decades

$\Rightarrow$ we should get APIs right (or at least better) first time

Outline

- I'll focus on first new feature added in a then-new system call, *prctl()* [*]
  - Chosen in part because it *seems* simple
- Aim is not to say that author of this API did anything wrong
- Rather: to show that when it comes to APIs, even something that seems super simple can turn out to be complicated!
- Consider some strategies for reduce frequency of design failures in future APIs

[*] I tested various behaviors of this API using this program:
`http://man7.org/code/procexec/pdeath_signal.c.html`

Outline

Outline

Since nearly the beginning of *time()*, there has been `SIGCHLD`

(Signal sent to parent process when child terminates)

(UNIX 4^{th} Edition, 1973)

One day, someone decided
the converse might be useful

```
Subject:     Patch to deliver signal to children
From:        Richard Gooch
Date:        1997-08-22 0:21:38

Hi, Linus. I've appended a patch (relative to 2.1.51)
which defines a new syscall with the interface:

extern int prctl (int option, ...);

Currently the only option which is supported is
PR_SET_PDEATHSIG [...]

Any child process which does:
prctl (PR_SET_PDEATHSIG, sig);

will have <sig> delivered to it when its parent process
dies. [...]  I've tested this with threaded applications
(i386) and it works like a charm. Just what I wanted.
Eventually I hope to see all kinds of PR_SET_* and
PR_GET_* options :-)
```

Send a signal to child process when its parent terminates

(Of course, there was no explanation of **why** the feature was needed)

Documentation!

- I don't know if the patch author contacted the manual page maintainer of the time
  - Back then, *man-pages* had no mailing list...
- But Andries Brouwer added documentation in early 1998:

```
PR_SET_PDEATHSIG
sets the parent process death signal of the current
process to arg2 (either a signal value in the range
1..maxsig, or 0 to clear). This is the signal that
the current process will get when its parent dies.
This value is cleared upon a fork().
```

  - "current process"??? $\Rightarrow$ "calling process"
    - (Kernel devs sometimes need to get out a little more...)

In summary

- So, now we have converse of `SIGCHLD`:
  - `prctl(PR_SET_PDEATHSIG, sig)`
    - And child gets a signal when parent goes away
- Simple, right?

What could go wrong?

Outline

Missing pieces

- What about discoverability?
- 2.3.15 (Aug 1999): `prctl(PR_GET_PDEATHSIG, &sig)`
  - Return current setting in *sig*
  - (2 years after `PR_SET_PDEATHSIG`)

man7.org

That’s even simpler, no?

(Maybe so, but still it was the start of a small mess)

Missing pieces

```
Subject:    [patch-2.3.44] slight change to prctl(2)
From:       Tigran Aivazian
Date:       2000-02-13 18:07:06

A long time ago I added PR_GET_PDEATHSIG to prctl(2) to
match the existing PR_SET_PDEATHSIG. Now that I noticed
[the subsequently added PR_GET_DUMPABLE] the whole
thing looks inconsistent so I suggest to change
PR_GET_PDEATHSIG so that it is the *return* value of
prctl(PR_GET_PDEATHSIG) instead of [returning the
setting in] the second argument [...]
```

- `PR_GET_DUMPABLE` returns value as function result; `PR_GET_PDEATHSIG` returns value via 2nd argument (**arg2*)
    - ("dumpable" was second *prctl()* operation implemented)
- **arg2* is perhaps better, for cases where value isn't *int*
- But at least we are consistently inconsistent...
    - Of the *prctl()* "get" style operations present in Linux 5.3:
        - 13 use function result, and 9 use **arg2*

Outline

How does some new feature interact with other parts of the Linux API?

Interactions across the interface

- Exploring interactions with these features can often yield surprises:
    - *fork()*
    - *execve()*
    - Signal delivery semantics
    - Threads
    - *exit()* / process termination
- (Of course, surprises can be found in other places too...)

Outline

Back to the manual page

- Back to the (1998) manual page:

```
PR_SET_PDEATHSIG
sets the parent process death signal of the current
process to arg2 (either a signal value in the range
1..maxsig, or 0 to clear) [...]
This value is cleared on fork().
```

- When I see that sentence, I wonder: what about *execve()*?
- Unfortunately, I didn't notice that omission until 2014, but now the manual page tells us:

```
This value is preserved across execve(2).
This value is preserved across execve(2).
```

Maybe, if that detail had been explicitly noted at the start, someone might have noticed a security vulnerability (earlier)...

Signal permissions

- From *kill(2)*:

```
For a process to have permission to send a signal,
it must either [have the CAP_KILL capability] or the
real or effective UID of the sending process must
equal the real or saved set-user-ID of the target
process.
```

  - Sending signals requires privilege or a credential match
- Can we use `PR_SET_PDEATHSIG` to send a signal to a process we could not otherwise signal?
  - That could be interesting for an attacker...

Scenario 0

```
 fork()----------------+
   |                    \
[parent (UID 1000)]   [child (UID 1000)]
   |                        |
   |                      prctl(PR_SET_PDEATHSIG, 1)
   |                        |
 exit()                     |
                            |
                      [child gets signal]
```

(This is what PR_SET_PDEATHSIG does)

Scenario 1

```
 fork()----------------+
   |                    \
[parent (UID 1000)]   [child (UID 1000)]
   |                       |
   |                  prctl(PR_SET_PDEATHSIG, 1)
   |                       |
   |                  execve("setuid-root-binary")
   |                       |
   |                  change all UIDs to 1001
   |                  [makes process unsignalable by parent]
   |                       |
 exit()                    |
                           |
                      [child does not get signal]
```

- Child execs set-UID-*root* binary that subsequently changes its UIDs such that parent can't signal it
- Consequently, parent-death signal **is not sent** to child
  - (Expected and correct behavior)

man7.org

Scenario 2 (a security bug)

```
 fork()-----------------+
   |                     \
[parent (UID 1000)]    [child (UID 1000)]
   |                       |
   |                   prctl(PR_SET_PDEATHSIG, 1)
   |                       |
   |                   execve("setuid-root-binary")
   |                       |
   |                   change all UIDs to 1001
   |                   [makes process unsignalable by parent]
   |                       |
execve("setuid-1001")      |
   |                       |
 exit()                    |
                           |
                       [child does get signal!]
```

- Parent execs a set-UID binary that confers credentials that do allow sending signal to child
- When parent terminates, parent-death signal **is sent** to child

The fix

- In 2007, **10 years after initial implementation**, this got fixed (in Linux 2.6.23):

```
The parent-death signal setting is cleared for the
child  of a fork(2). It is also cleared when
executing a set-user-ID or set-group-ID binary,
or a binary that has associated capabilities;
otherwise, this value is preserved across execve(2).
```

Outline

Threads, part 1

- Let’s go back to the original patch message:

```
Any child process which does:
prctl (PR_SET_PDEATHSIG, sig);

will have <sig> delivered to it when its parent
process dies.
```

- If “process termination” means “termination of last thread”, this turns out not to be true
  - Or at least not by the time we got NPTL in 2003
    - (NPTL is POSIX Threads implementation in GNU C library)

A bug report

- https://bugzilla.kernel.org/show_bug.cgi?id=43300, David Wilcox:

```
I have a process that is forking to [create] a child
process. The child process should not exist if the
parent process [exits]. So, I call
prctl(PR_SET_PDEATHSIG, SIGKILL) in the child
process to kill it if the parent dies. What ends up
happening is the parent thread calls pthread_exit,
and that thread ends up being the catalyst that
kills the child process.
```

- **Signal is sent upon termination of the creating thread**
  - I.e., the thread that actually called *fork()*
    - (Rather than when last thread in parent terminates)

man7.org

That bug report was in May 2012!

(Sometimes, API misdesigns are reported only **much** later)

A bug that we can't fix

- And we can't fix this; Oleg Nesterov in the same bug:

```
And yes, the current behaviour looks just ugly. The
problem is, unlikely we can change it now, this can
obviously break the applications which rely on the
fact that pdeath_signal is per-thread.
```

  - (I.e., some apps might depend on this strange behavior)

- But at least we can document it:

```
Warning: the "parent" in this case is considered to
be the thread that created this process. In other
words, the signal will be sent when that thread
terminates (via, for example, pthread_exit(3)),
rather than after all of the threads in the parent
process terminate.
```

- Actually, the story is more complicated than that; we'll revisit

Outline

Signals and threads

- Back to the original patch message again:

```
Any child process which does:
prctl (PR_SET_PDEATHSIG, sig);

will have <sig> delivered to it when its parent
process dies.
```

- What if the child is multithreaded?

Threads and signals

- From *signal(7)*:

```
A signal may be process-directed or thread-directed.
A process-directed signal is one that is targeted at
(and thus pending for) the process as a whole. [...]
A thread-directed signals is one that is targeted at
a specific thread. A process-directed signal may be
delivered to any one of the threads that does not
currently have the signal blocked.
```

- We better let the user know what they are dealing with:

```
The parent-death signal is process-directed.
```

Does the child get more than a signal?

- So, the child gets a signal; is that all?
- Until recently, you had to read the kernel source to know
- Now the manual page tells us:

```
If the child installs a handler using the
sigaction(2) SA_SIGINFO flag, the si_pid field of
the siginfo_t argument of the handler contains the
PID of the terminating parent process
```

  - Note: it's the **PID** (TGID) of the parent process, **not the TID of the terminating thread**
    - (You wanted consistency in the API misdesigns?)

Another (formerly) undocumented corner case

- If parent has already terminated before child does `PR_SET_PDEATHSIG`, does child get a signal?
- Original patch message and manual page text didn't say
- Now the manual page tells us:

```
If the parent thread [has] already terminated by
the time of the PR_SET_PDEATHSIG operation,
then no parent-death signal is sent to the caller.
```

  - (Yes, there is a race there if trying to detect termination of birth parent...)

man7.org

Outline

What becomes of an orphan?

- Suppose the child continues executing after receiving the parent-death signal
  - Might be perfectly reasonable in some scenarios
- So, what happens to a child after its parent terminates?
  - It gets adopted by *init* (PID 1)
  - Well, that was true once upon a time...

Subreapers

- `prctl(PR_SET_CHILD_SUBREAPER, 1)` marks a process as a “subreaper” for any orphaned descendants

```
A subreaper fulfills the role of init(1) for its
descendant processes.  When a process becomes
orphaned, then that process will be reparented
to the nearest still living ancestor subreaper.
```

  - If any of my descendants become orphaned, reparent them to me (not to *init*)
  - Linux 3.4, 2012
  - (Most well-known user of this feature is *systemd*)
- How do subreapers interact with `PR_SET_PDEATHSIG`?

man7.org

Subreapers

- Thanks to subreaper mechanism, a child process can have a series of parents
  - (When a subreaper terminates, it's children are adopted by next ancestor subreaper)
- So, a child may get a series of parent-death signals!

```
The parent-death signal is sent upon subsequent
termination of the parent thread and also upon
termination of each subreaper process to which
the caller is subsequently reparented.
```

man7.org

Outline

Threads caused quite a bit of grief for traditional UNIX (and Linux) APIs...

Threads, part 2

- Suppose one of those subreaper processes is multithreaded... when does child get the parent-death signal?
- What's your guess?
  - When **first** thread in parent terminates
  - When **last** thread in parent terminates
  - When **thread group leader** in parent terminates
  - Upon termination of **each thread** in parent

“none of the above”

Threads, part 2

- From looking at code and comments in *find_new_reaper()*, here's what I think happens:
    - Child processes are parented by individual threads
    - When a thread terminates, its children are reparented to another thread (if one exists) in same process
        - And child gets pdeath signal, if it requested it
    - Search for new parent is in order of thread creation, so that search starts with thread group leader
        - Or possibly, search is on numerical order of TID
- Same behavior for original parent of process that used `PR_SET_PDEATHSIG`

man7.org

Do I dare to document this?

(So far, I did not)

(Given enough time, users will invent every possible use case for an API, or write programs that accidentally depend on details of API behavior)

(Even if you don't document it...)

Outline

What we wanted:

Child should get a signal when parent terminates

What we actually got...

- If parent is multithreaded, child gets signal when creating **thread** terminates
- Child may get multiple signals if parent is multithreaded
  - # of signals depends on order of thread creation in parent!
  - Each signal has same *si_pid* value
  - Accidental exposure of details of kernel's implementation of process management
- Child gets multiple signals if there are ancestor subreapers
  - And if those subreapers are multithreaded, see above...
- A race, if trying to detect termination of birth parent
- A security bug (signal a process owned by another UID)
  - Now fixed
- The start of an API inconsistency (*prctl()* "get" operations)

Clearly, many of these behaviors were unintended

What went wrong?

- No one person/group owns the interface
- No/insufficient documentation
- Insufficient consideration of interaction with other parts of interface
- Behavior evolved with the addition of other interfaces / kernel features
  - Some of these behaviors almost certainly changed over time
- Decentralized design often fails us

Outline

Who owns the interface?

IOW: who gets to say what the interface contract is?

The answer isn't simple, and that's part of the problem

(http://man7.org/conf/lpc2008/who_owns_the_interface.pdf)

Is it the kernel developers?

- It seems “obvious” that it must be the kernel developers
  - They write the code that implements the interfaces!
- But, various points contradict:
  - What if implementation deviates from intention? (A bug)
  - What about unforeseen uses of interface?
  - Glibc wrappers mediate between kernel and user space

Is it glibc developers?

- Glibc provides wrappers for most system calls
  - Sometimes wrappers change or add behavior
- But...
  - In many cases, wrapper is trivial (no behavior change)
  - In some cases, it's a long time, or never, before wrapper lands in glibc
    - It was 18 years before *gettid()* got a glibc wrapper; *https://sourceware.org/bugzilla/show_bug.cgi?id=6399*
  - Various APIs are not (conventional) syscalls; glibc is not involved
    - `/proc`, `/sys`, *ioctl()*, netlink

Is it me?

- *man-pages* documents kernel APIs
  - Goal: document what kernel guarantees to user space
  - Documentation can act as specification, describing developer's intention
    - Allows testing for difference between implementation and intention
- But...
  - Many things remain undocumented
  - Sometimes implementation is right and docs are wrong :-(

Is it the users?

- How could it possibly be the users?
- Given enough time, users collectively discover every possible detail of the API
  - They read the source code
  - They look at exported symbols
  - They just try stuff
  - They (eventually) discover all the misdesigns/antifeatures

Is it the users?

- Discoveries can be both deliberate and accidental
  - Deliberate: user discovers API behaviors and explicitly makes use of them
  - Accidental: user writes code that implicitly depends on an API behavior
    - Users may even write code that depends on **buggy** interface behavior
- User code may depend on behaviors that implementer hadn't considered/was unaware of
  - Ancient example: oddball uses cases for files with permissions such as `rw----r--`!
    - (Users in one group have less permission than world)

man7.org

Outline

The original PR_SET_PDEATHSIG documentation (1998)

```
PR_SET_PDEATHSIG sets the parent process death signal of the current
process to arg2 (either a signal value in the range 1..maxsig, or 0
to clear). This is the signal that the current process will get
when its parent dies. This value is cleared upon a fork().
```

The current PR_SET_PDEATHSIG documentation (2019)

```
PR_SET_PDEATHSIG (since Linux 2.1.57)
Set the parent-death signal of the calling process to arg2 (either
a signal value in the range 1..maxsig, or 0 to clear).  This is
the signal that the calling process will get when its parent dies.

Warning: the "parent" in this case is considered to be the thread
that created this process. In other words, the signal will be sent
when that thread terminates (via, for example, pthread_exit(3)),
rather than after all of the threads in the parent process terminate.

The parent-death signal is sent upon subsequent termination of the
parent thread and also upon termination of each subreaper process
(see the description of PR_SET_CHILD_SUBREAPER above) to which the
caller is subsequently reparented. If the parent thread and all
ancestor subreapers have already terminated by the time of the
PR_SET_PDEATHSIG operation, then no parent-death signal is sent
to the caller.

The parent-death signal is process-directed (see signal(7)) and, if
the child installs a handler using the sigaction(2) SA_SIGINFO flag,
the si_pid field of the siginfo_t argument of the handler contains
the PID of the terminating parent process.

The parent-death signal setting is cleared for the child of a
fork(2).  It is also (since Linux 2.4.36 / 2.6.23) cleared when
executing a set-user-ID or set-group-ID binary, or a binary that
has associated capabilities (see capabilities(7)); otherwise, this
value is preserved across execve(2).
```

Some of the now-documented behaviors didn't exist in 1997, but several did

(and there's still that piece that is undocumented)

The problems resulting from lack of documentation

- If *execve()* semantics had been documented, people might have noticed a security vulnerability earlier
- If kernel semantics for reparenting child when parent thread terminates had been documented, we might have:
  - Noticed those semantics are slightly insane
    - Children should be children of parent **process**, not a thread
  - And realized implications for `PR_SET_PDEATHSIG`

The problems resulting from lack of documentation

- If **documentation** was written as API was implemented, it might have helped reviewers spot these problems
  - **Document lowers the bar for code review**
- Documentation provides a specification for testing
  - How do we write a (correct) test without a spec?

I'll just ignore the many problems that insufficient documentation creates for API consumers

Outline

Interactions across the interface

- `PR_SET_PDEATHSIG` is a case study in how overlooking interactions with other interfaces can cause later surprises:
  - Threads
  - Process termination
  - *execve()*
- Other features that should always be considered:
  - *fork()*
    - What should/does child inherit from/share with parent?
  - Signals
    - Process / thread directed? What *siginfo* arrives with signal?
  - If file descriptors are in play: FDs vs open file descriptions
    - Multiple FDs may refer to same OFD (*dup()*, *fork()*, etc)
    - This mismatch has led to ugly corner cases in, e.g., *signalfd(2)* and *epoll(7)* (and, further back, *fcntl()* locking)
- The above lists are far from exhaustive...

Outline

Inconsistencies and surprises

- Inconsistencies: `PR_GET_PDEATHSIG` vs `PR_GET_DUMPABLE`
  - Return info via function result or via an argument?
- 2003 addition of NPTL almost certainly changed the behavior of `PR_SET_PDEATHSIG`
  - Child gets signal when creating **thread** terminates
  - Child may get multiple signals as threads in parent terminate one by one
- 2012 addition of `PR_SET_CHILD_SUBREAPER` magnified the previous point
- In each case, problem was failure to see the bigger picture

API problems are often noticed only much later

- Some problems in `PR_SET_PDEATHSIG` were noticed only after years:
    - 2007: *execve()* security issue was found and fixed
    - 2012: the bizarre behavior when parent process is multithreaded was reported
        - But can't be fixed!
- These are not isolated cases
    - E.g., by the time an accidental behavior change in *fcntl()* `F_SETOWN` in Linux 2.6.12 was noticed, it was too late to fix
        - So now we have `F_SETOWN_EX` (Linux 2.6.32) to do what `F_SETOWN` used to do
- Often users don't know how or where to report such issues
    - A POSIX MQ API breakage in Linux 3.5 was notified to me a year later as a bug report against manual page!
        - (Instead, the kernel breakage got fixed)

`PR_SET_CHILD_SUBREAPER` is
a small lesson in the school of
“we don’t do decentralized design well”

We’ve had much harsher lessons

Control groups v1,
overloaded `CAP_SYS_ADMIN` capability, ...

We have too few eyes looking at the big picture

Not enough people with motivation, time, and knowledge to consider things such as API consistency and interactions across the interface

Why isn't there a paid kernel user-space API maintainer(s)?

Outline

In my ideal world, here's what would happen

(We can't eliminate the problems, but we can reduce them)

linux-api@vger.kernel.org

- Every patch that changed the interface would CC linux-api@vger.kernel.org
  - (And every kernel dev would **remind those who forget**)
- List intended to advertise proposed API changes
  - Part of answering question: **how do we even know if interface changed?**
- Subscribers do/may include:
  - (GNU) C library developers
  - *strace* developers
  - Testing / fuzzing project maintainers
  - Various people who just care about the interface
    - Kernel and user-space developers
  - Me

man7.org

Commit messages

- Every commit message, but especially those that change interfaces would
  - Explain **why** the (interface) change was being made
  - Include explanations of why features **are** included
  - Include explanations of why features **are not** included
  - Include **URLs referring to mailing list discussions**
  - Include a **version history** that explains how patch evolved over time
    - (That often helps with two preceding points)
- It's not **so** hard...
  - If you want to see how it's done, take a lesson from Christian Brauner
    - **3eb39f47934f and 7f192e3cd316 fill me with gratitude**

Engaging with glibc

- Kernel developers adding new user-space APIs would work with (g)libc developers
    - To ensure that glibc support is added in parallel with kernel changes
- The glibc developers are good at spotting API problems that will make user-space unhappy

Tests

- Naturally, every new API feature would have multiple tests in *kselftest*
- Things have gotten better, but they could be better still

Features should have real users

- No new API would be merged without a real-world app that provides a first test of the design (and implementation)
- Numerous times, real users started using API only after it was merged into kernel
    - And then we discovered the design problems
        - Which by then are usually unfixable
    - Example sad story: *inotify*
        - https://lwn.net/Articles/605128/

Documentation

- A man-pages patch would be written in parallel with development of new API
    - Not as an after-thought
- Documenting an API:
    - Is a great trigger for developer to reconsider their design concept
    - Lowers the bar for reviewers to understand (and therefore comment) on your patch
- And of course, end users will thank you for that documentation
    - (So will I)

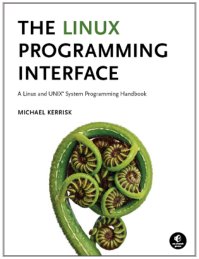
man7.org

Thanks!

Michael Kerrisk, Trainer and Consultant
http://man7.org/training/

mtk@man7.org @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/


THE LINUX PROGRAMMING INTERFACE
A Linux and UNIX® System Programming Handbook
MICHAEL KERRISK