A `fork()` in the Road

Jeremy Mates

June 24, 2021

In "A `fork()` in the road"Baumann et al. [2019] two programs are mentioned as being slow with with `fork(2)`–Chrome, and `node`. These are atypical choices, as we shall see. Chrome uses abnormal amounts of memory, and a good security policy might well deny Chrome the right to fork if, hypothetically, Chrome were to suffer from some security issue. What should Chrome do if it cannot fork? Perhaps it should, at launch, fork a small supervisor process–maybe called `init`–which would then be responsible for starting the other processes that Chrome needs–perhaps via some "system daemon" framework. This way, processes with heavy memory use or multithreading needs would only do that, and would not ever be calling fork or exec. Did I just describe an operating system? Probably. Is Chrome a not very good operating system? Probably.

`node` used to crash (in 2019, but no longer does in `v12.16.1`) if you printed to the console in a loop via

```
$ node -e 'while (1) { process.stdout.write("a") }' > /dev/null
Bus error (core dumped)
```

so it's not surprising to me that it would have other problems such as taking a long time to fork. Perhaps as skill with heavily multithreaded programs goes up such issues will get ironed out? Assuming the process even needs to be multithreaded and to fork.

Anyways, how atypical is Chrome? Let's look at memory use on the 2009 MacBook.

```
PID   COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM
0     kernel_task  0.7  12:14.54 114/2 0    2     380M
380   QuickTime Pl 0.0  50:45.86 15    2    342   57M
232   WindowServer 0.3  47:39.50 5     2    311   56M
281   iTerm2       0.2  03:04.65 6     0    252   28M
287   Finder       0.0  00:07.63 3     0    271   26M
285   Dock         0.0  00:01.76 3     0    225   24M
463   Sudoku       0.0  00:09.48 6     1    275   22M
482   Dictionary   0.0  00:00.79 5     0    200   19M
298   Spotlight    0.0  00:00.73 4     0    171   10M
```

Do those memory intensive processes fork? At all? What does fork a lot? The shells, which are using. . . oh there they are, 1888K, 2268K, etc. Would `posix_spawn` even help for processes that small? How much memory does Chrome use? I'd have to install it to find out; Safari.app under Mac OS X 10.11 uses around 128M doing nothing, or two orders of magnitude more memory than the shells. I've heard stories of Chrome eating into another order of magnitude, or two? Chrome having problems with `fork()` is less about `fork()` and more about Chrome being, well, bloaty.

Cherry Picking

I too can cherry pick things favorable to my position. Here are my OpenBSD processes that fork, a lot:

```
$ ps axo rss,vsz,args | grep s[h]
  968  1008 /bin/sh /etc/X11/xenodm/Xsession
 1148  1220 /bin/ksh -l
 1052  1020 /bin/ksh -l
 1068  1152 /bin/ksh -l
 1036  1016 /bin/ksh -l
```

And here we sort on the RES column in `top(1)` with the output cleaned up a bit:

```
  PID USERNAME  SIZE   RES STATE   TIME   CPU COMMAND
64699 _x11       42M   52M sleep/0 0:19 0.49% Xorg
54110 jmates     19M   24M sleep/0 0:02 0.00% mupdf-x11
49777 jmates     12M   18M sleep/0 0:04 0.00% irssi
 2134 root     2100K 8080K idle    0:01 0.00% xenodm
 3606 jmates   1148K 6748K sleep/1 0:00 0.00% cwm
45572 jmates   1884K 5784K sleep/1 0:05 0.00% xterm
14369 jmates   1884K 5756K sleep/1 0:07 0.00% xterm
86394 jmates   1868K 5740K sleep/1 0:01 0.00% xterm
46500 jmates   1864K 5740K sleep/0 0:01 0.00% xterm
76013 jmates   4812K 5344K sleep/1 0:01 0.00% vim
```

`cwm(1)` and `vim(1)` are the only where notable fork/exec would happen, and not ever in any sort of performance critical path; in the others fork/exec would be minimal to zero; in the case of Xorg and irssi `execve(2)` is not even allowed. And the multi-threaded programs, all of them:

```
49777 irssi
64699 /usr/X11R6/bin/X
```

The Replacements

Assuming, hypothetically, that we are on some sort of unix that runs memory-intensive processes with heavy multithreading that also need to fork, what are we to replace fork with? The paper indicates `posix_spawn`, which is "not a complete replacement for fork and exec... it... lacks an effective error reporting mechanism"[Baumann et al., 2019, p.5]. Perhaps someone could spend time (isn't programmer time expensive?) to catalog what existing uses of fork could actually be rewritten–and then debugged, and tests written for any regressions (wait, are there even tests?), the documentation updated, etc.–to use `posix_spawn` and then process startup might be a bit faster? Is this even a problem? `vim(1)` is slow to come up cold off of disk, but that's mostly due to the spinning metal hard drive and the 0.8 GHz CPU running OpenBSD, an OS well noted for its speed. Here are some hot cache forks and execs:

```
$ repeat 3 time vim -c quit
        0.11 real         0.08 user         0.03 sys
        0.11 real         0.09 user         0.02 sys
        0.11 real         0.08 user         0.03 sys
$ highcpu
$ repeat 3 time vim -c quit
        0.04 real         0.04 user         0.01 sys
        0.04 real         0.03 user         0.01 sys
        0.05 real         0.04 user         0.01 sys
```

How would `posix_spawn` help here? A migration is a non-starter if you need that error reporting, or otherwise looks to be of moderate effort and low reward. Perhaps if `vim(1)` were being spawned often from software with excessive memory use? But why run such? Why not launch `vim(1)` from a small, fast shell?

An Aside with `nmap`

I did once have a problem with launching many `nmap(1)` processes; I was testing a LDAP server, where lots of connections (or traffic, or something) would (sometimes) wedge the server–obviously only in production, and never in the test environment. The test script probably looked something like:

```
for i in ...; do nmap ... & done
```

Except it was much less pretty, maybe with a `wait` and a second loop to keep spawning more nmaps. Did I rewrite this slow thing to use `posix_spawn`? Nope. I wrote a new script–two actually–that performed a test inside a very tight loop in a single process. The shell forked and execed these test programs, and LDAP fell over. Success! Where would `posix_spawn` help? Maybe to shave off some script startup time? Noise,

compared to converting a quick "I have an idea. . . " mess of a shell pipeline (the prototype) into a pair of very fast test scripts (a first implementation).

Back to the so-called Alternatives

vfork(2) is mentioned as an alternative BSD thing. Let's read the fine manual.

```
BUGS
     This system call will be eliminated when proper system sharing
     mechanisms are implemented. Users should not depend on the memory
     sharing semantics of vfork as it will, in that case, be made
     synonymous to fork.
```

Based on that I'd agree with the article that "in most cases it is better avoided"[Baumann et al., 2019, p.5]. The documentation on Mac OS X 10.11 is not known for being updated or up-to-date; let's try OpenBSD.

```
DESCRIPTION
     vfork() was originally used to create new processes without fully
     copying the address space of the old process, which is
     horrendously inefficient in a paged environment. It was useful
     when the purpose of fork(2) would have been to create a new
     system context for an execve(2). Since fork(2) is now efficient,
     even in the above case, the need for vfork() has diminished.
...
HISTORY
     The vfork() function call appeared in 3.0BSD with the additional
     semantics that the child process ran in the memory of the parent
     until it called execve(2) or exited. That sharing of memory was
     removed in 4.4BSD, leaving just the semantics of blocking the
     parent until the child calls execve(2) or exits.
```

Not seeing anything new nor viable about `vfork(2)`. Why is it even listed as an alternative?

The article then mentions some low-level something from elsehwere, and concludes that it "seems at first glance challenging, but may also be productive for future research"[Baumann et al., 2019, p.6]. Good luck with that?

How about `clone()`?

```
% man clone
No manual entry for clone

$ man clone
man: No entry for clone in the manual.
```

So not portable, plus "clone suffers most of the same problems as fork"[Baumann et al., 2019, p.6].

To conclude, there are simply not any viable alternatives; the call to "strongly discourage" the use of fork in new code is both laughable and absurd. Why not simply use `fork()` in a small and fast single threaded processes. . . oh wait, the authors did realize that. In hindsight.

References

Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. Hotos 2019: Proceedings of the workshop on hot topics in operating systems. https://dl.acm.org/doi/10.1145/3317550.3321435, 2019.