Spawning processes faster and easier with io_uring

Josh Triplett
josh@joshtriplett.org
@josh_triplett

Linux Plumbers Conference 2022

Build systems

Launching a process on UNIX

Redirection / file descriptors

Priority

Affinity

Signal masks

UID/GID

chroot

namespaces

seccomp filters

⋮

Need to do setup before launching

Need code to do setup before launching

Where does that code come from?

Where does that code come from?

The current process!

fork/exec

`fork`

Create a copy-on-write copy
of the current process

Copy-on-write (CoW)

- Doesn't copy all the memory

Copy-on-write (CoW)

- Doesn't copy all the memory
- Does copy all the page metadata

`exec`

Throw away the current process and replace it with a new program

How expensive is fork?

Test and benchmarking setup

Test and benchmarking setup

- Create a pipe

Test and benchmarking setup

- Create a pipe
- Read the start time

Test and benchmarking setup

- Create a pipe
- Read the start time
- Spawn child (using `PATH`) via method to test

Test and benchmarking setup

- Create a pipe
- Read the start time
- Spawn child (using `PATH`) via method to test
- Child writes end time to pipe, then exits

Test and benchmarking setup

- Create a pipe
- Read the start time
- Spawn child (using `PATH`) via method to test
- Child writes end time to pipe, then exits
- Print the fastest (minimum) time from 2000 runs

method	base
`fork`	52.0µs

method	base	1G
`fork`	52.0µs	56.4µs

Linux has clever optimizations

Most programs allocate memory they don't use

“Allocated” memory doesn’t really get allocated until used

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs

Performance isn't the only problem with `fork`

multithreading

locks held by other threads
will remain held (forever) in the child

Calling a library function could deadlock

"async-signal-safe"

`man 7 signal-safety`

chroot

chroot
setpriority

vfork

`vfork`

Create a child that borrows
the current process

Wait until child finishes

`vfork`

Create a child that borrows
the current process

Wait until child finishes

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`			

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs		

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs	31.4µs	

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs	31.4µs	31.9µs

What can you do after `vfork`?

exec

exec
...and _exit

`exec`

`...and _exit`

Also, don't write to any memory

`exec`

`...and _exit`

Also, don't write to any memory

Including local stack (except a PID)

`exec`

`...and _exit`

Also, don't write to any memory

Including local stack (except a PID)

And don't return or call anything

borrows the current process?

`vfork`

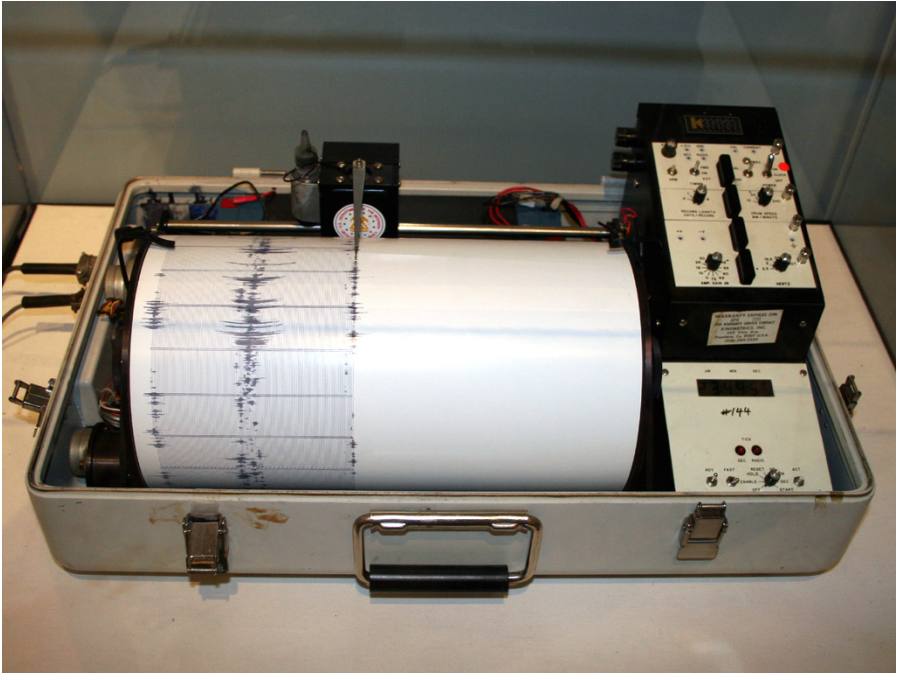
Effectively a thread

`vfork`

Effectively a thread
with no synchronization

`vfork`

Effectively a thread
with no synchronization
running with the same stack as the parent



Hope you didn't actually need setup code...

Hope you didn't actually need setup code...

How sure are you that your compiled code didn't use the stack?

Hope you didn't actually need setup code...

How sure are you that your compiled code didn't use the stack?

What if your child process receives a signal?

posix_spawn

One call to spawn a process

Created for systems that couldn't `fork`

Hand the problem to the C library

No setup code, many configuration options

posix_spawn_file_actions_t

posix_spawn_file_actions_t

posix_spawnattr_t

glibc uses a safer version of `vfork`

glibc uses a safer version of `vfork`

Separate stack
Blocking all signals

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs	31.4µs	31.9µs
`posix_spawn`			

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs	31.4µs	31.9µs
`posix_spawn`	44.5µs		

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs	31.4µs	31.9µs
`posix_spawn`	44.5µs	44.0µs	

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs	31.4µs	31.9µs
`posix_spawn`	44.5µs	44.0µs	44.9µs

Why do we need a copy of the process?

Need setup code for a new process

Need setup code for a new process

- `fork` lets setup be in the existing process's code

Need setup code for a new process

- `fork` lets setup be in the existing process's code
- `vfork` doesn't support setup code

Need setup code for a new process

- `fork` lets setup be in the existing process's code
- `vfork` doesn't support setup code
- `posix_spawn` provides specific setup operations

io_uring

io_uring

- Shared-memory communication with the kernel

io_uring

- Shared-memory communication with the kernel
- Submission Queue (SQ) and Completion Queue (CQ) ringbuffers

io_uring

- Shared-memory communication with the kernel
- Submission Queue (SQ) and Completion Queue (CQ) ringbuffers
- Similar to NVMe and virtio protocols

io_uring

- Shared-memory communication with the kernel
- Submission Queue (SQ) and Completion Queue (CQ) ringbuffers
- Similar to NVMe and virtio protocols
- Avoids kernel entry/exit overhead

io_uring

- Shared-memory communication with the kernel
- Submission Queue (SQ) and Completion Queue (CQ) ringbuffers
- Similar to NVMe and virtio protocols
- Avoids kernel entry/exit overhead
- Supports linked operations

What if we specified process setup and launch using a ring of linked operations?

A kernel task doesn't need userspace

New io_uring operations

- `IORING_OP_CLONE` — Capture linked operations and run them in a new task

New io_uring operations

- `IORING_OP_CLONE` — Capture linked operations and run them in a new task
- `IORING_OP_EXEC` — Exec a new program in the task, skipping remaining operations if successful

If a `IORING_OP_CLONE` task runs out of linked operations, it gets `SIGKILL`ed without returning to (non-existent) userspace.

A successful `IORING_OP_EXEC` skips further ring operations.

A successful `IORING_OP_EXEC` skips further ring operations.

A failed `IORING_OP_EXEC` allows more ring operations if not `HARDLINK`ed.

Path search

Bypassing libc wrappers

Works in multithreaded programs

```
struct io_uring_sqe *sqe;
sqe = io_uring_get_sqe(&ring);
io_uring_prep_clone(sqe);
io_uring_sqe_set_flags(sqe, IOSQE_IO_LINK);
sqe = io_uring_get_sqe(&ring);
io_uring_prep_exec(sqe, "./t", argv, envp);
io_uring_submit(&ring);
```

Useful for reasons other than performance...

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs	31.4µs	31.9µs
`posix_spawn`	44.5µs	44.0µs	44.9µs
`io_uring_spawn`			

method	base	1G	1G init
`fork`	52.0µs	56.4µs	7581.8µs
`vfork`	31.5µs	31.4µs	31.9µs
`posix_spawn`	44.5µs	44.0µs	44.9µs
`io_uring_spawn`	29.5µs	30.2µs	28.6µs

6-10% faster than `vfork`
safer and more flexible than `vfork`

6-10% faster than `vfork`
safer and more flexible than `vfork`

31-36% faster than `posix_spawn`

Just getting started

Next steps

Next steps

- Implement `posix_spawn`

Next steps

- Implement `posix_spawn`
- Support pre-spawned process pool

Next steps

- Implement `posix_spawn`
- Support pre-spawned process pool
- Optimize clone further

Next steps

- Implement `posix_spawn`
- Support pre-spawned process pool
- Optimize clone further
- Set up process "from scratch"

Next steps

- Implement `posix_spawn`
- Support pre-spawned process pool
- Optimize clone further
- Set up process "from scratch"
- Use pre-registered file descriptors

Aside: `CLONE_VM`

Acknowledgements

Jens Axboe

@josh_triplett

https://github.com/sponsors/joshtriplett

https://buildit.dev

@josh_triplett

https://github.com/sponsors/joshtriplett

https://buildit.dev

Questions?

Image credits:
Seismograph: By Yamaguchi, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1089235