Introducing dumb-init, an init system for Docker containers

Chris K., Software Engineer
Jan 6, 2016

At Yelp we use Docker containers everywhere: we run tests in them, build tools around them, and even deploy them into production. In this post we introduce dumb-init, a simple init system written in C which we use inside our containers.

Lightweight containers have made running a single process without normal init systems like systemd or sysvinit practical. However, omitting an init system often leads to incorrect handling of processes and signals, and can result in problems such as containers which can't be gracefully stopped, or leaking containers which should have been destroyed.

dumb-init is simple to use and solves many of these problems: you can just add it to the front of any container's command, and it will take on the role of PID 1 for itself. It immediately spawns your process as PID ~2, and then proxies on any signals it receives. This helps to avoid special kernel behavior applied to PID 1, while also handling regular responsibilities of the init system (like reaping orphaned zombie processes).

The motivation: modeling Docker containers as regular processes

What we really want is to be able to treat Docker containers just like regular processes, so that we can slowly migrate our tools and infrastructure toward Docker. Instead of forcing developers to unlearn their existing workflow, we can move individual commands into containers without developers even realizing they're spawning a Docker container on each invocation.

It also lets us take a practical approach to Docker in development: rather than require that everything live in a container, we can choose to use containers when it makes sense from a business or technical perspective.

To achieve this goal, we want processes to behave just as if they weren't running inside a container. That means handling user input, responding the same way to signals, and dying when we expect them to. In particular, when we signal the docker run command, we want that same signal to be received by the process inside.

Our quest to model Docker containers as regular processes led us to discovering more than we ever wanted to know about how the Linux kernel handles processes, sessions, and signals.

Process behavior inside Docker containers

Containers are unique in that they often run just a single process, unlike traditional servers where even a minimal install usually runs at least a complex init system, cron, syslog, and an SSH daemon.

While single-process containers are quick to start and light on resources, it's important to remember that, for most intents and purposes, these containers are full Linux systems. Inside your container, the process running as PID 1 has special rules and responsibilities as the init system.

What is PID 1 inside a container? There are two common scenarios.

Scenario 1: A shell as PID 1

A quirk of Dockerfiles is that if you specify your container's command without using the recommended JSON syntax, it will feed your command into a shell for execution.

That results in a process tree that looks like:

- `docker run` (on the host machine)
    - `/bin/sh` (PID 1, inside container)
        - `python my_server.py` (PID ~2, inside container)

Having a shell as PID 1 actually makes signaling your process almost impossible. Signals sent to the shell won't be forwarded to the subprocess, and the shell won't exit until your process does. The only way to kill your container is by sending it `SIGKILL` (or if your process happens to die).

For this reason, you should always try to avoid spawning a shell. If you can't easily avoid that (for example, if you want to spawn two processes), you should `exec` your last process so that it replaces the shell.

Scenario 2: Your process as PID 1

When you use the recommended syntax in your Dockerfile, your process is started immediately and acts as the init system for your container, resulting in a process tree that looks like:

- `docker run` (on the host machine)
    - `python my_server.py` (PID 1, inside container)

This is better than the first scenario; your process will now actually receive signals you send it. However, being PID 1, it might not respond to them quite as you expect it to.

Trouble signaling PID 1

The Linux kernel treats PID 1 as a special case, and applies different rules for how it handles signals. This special handling often breaks the assumptions that programs or engineers make.

First, some background. Any process can register its own handlers for `TERM` and use them to perform cleanup before exiting. If a process hasn't registered a custom signal handler, the kernel will normally fall back to the default behavior for a `TERM` signal: killing the process.

For PID 1, though, the kernel won't fall back to any default behavior when forwarding `TERM`. If your process hasn't registered its own handlers (which most processes don't), `TERM` will have no effect on the process.

Since we're modeling containers as processes, we'd like to just send `SIGTERM` to the docker run command and have the container stop. Unfortunately, this usually doesn't work.

When docker run receives `SIGTERM`, it forwards the signal to the container and then exits, even if the container itself never dies. In fact, the `TERM` signal will often bounce right off of your process without stopping it because of the PID 1 special case.

Even using the command docker stop won't do what you want; it sends `TERM` (which the Python process won't notice), waits ten seconds, and then sends KILL when the process still hasn't stopped, immediately stopping it without any chance to do cleanup.

Not being able to properly signal services running inside your Docker container has lots of implications, both in development and in production. For example, when deploying a new version of your app, it might have to kill the previous service version without letting it clean up (potentially dying in the middle of serving a request, or leaving connections open to your database). It also leads to a common problem in CI systems (such as Jenkins) where aborted tests leave Docker containers still running in the background.

The same problem applies to other signals. The most notable case is `SIGINT`, the signal generated when you press `^C` in a terminal. Since this signal is caught even less frequently than `SIGTERM`, it can be especially troublesome trying to manually kill servers running in your development environment.

dumb-init to the rescue

To address this need, we created dumb-init, a minimal init system intended to be used in Linux containers. Instead of executing your server process directly, you instead prefix it with dumb-init in your Dockerfile, such as `CMD ["dumb-init", "python", "my_server.py"]`. This creates a process tree that looks like:

- `docker run` (on the host machine)
    - `dumb-init` (PID 1, inside container)
        - `python my_server.py` (PID ~2, inside container)

dumb-init registers signal handlers for every signal that can be caught, and forwards those signals on to a session rooted at your process. Since your Python process is no longer running as PID 1, when dumb-init forwards it a signal like `TERM`, the kernel will still apply the default behavior (killing your process) if it hasn't registered any other handlers.

Using a regular init system also solves these problems, but at the expense of increased complexity and resource usage. dumb-init is a simpler way to do things properly: it spawns your process as its only child, and proxies signals to it. dumb-init won't actually die until your process dies, allowing you to do proper cleanup.

dumb-init is deployed as a statically-linked binary with no extra dependencies; it's ideal to serve as a simple init system, and can typically be added to any container. We recommend using it in basically any Docker container; not only does dumb-init improve signal handling, but it also takes care of other functions of an init system, such as reaping orphaned zombie processes.

You can find much more information about the role of dumb-init and how to start using it on its GitHub page.

Back to blog