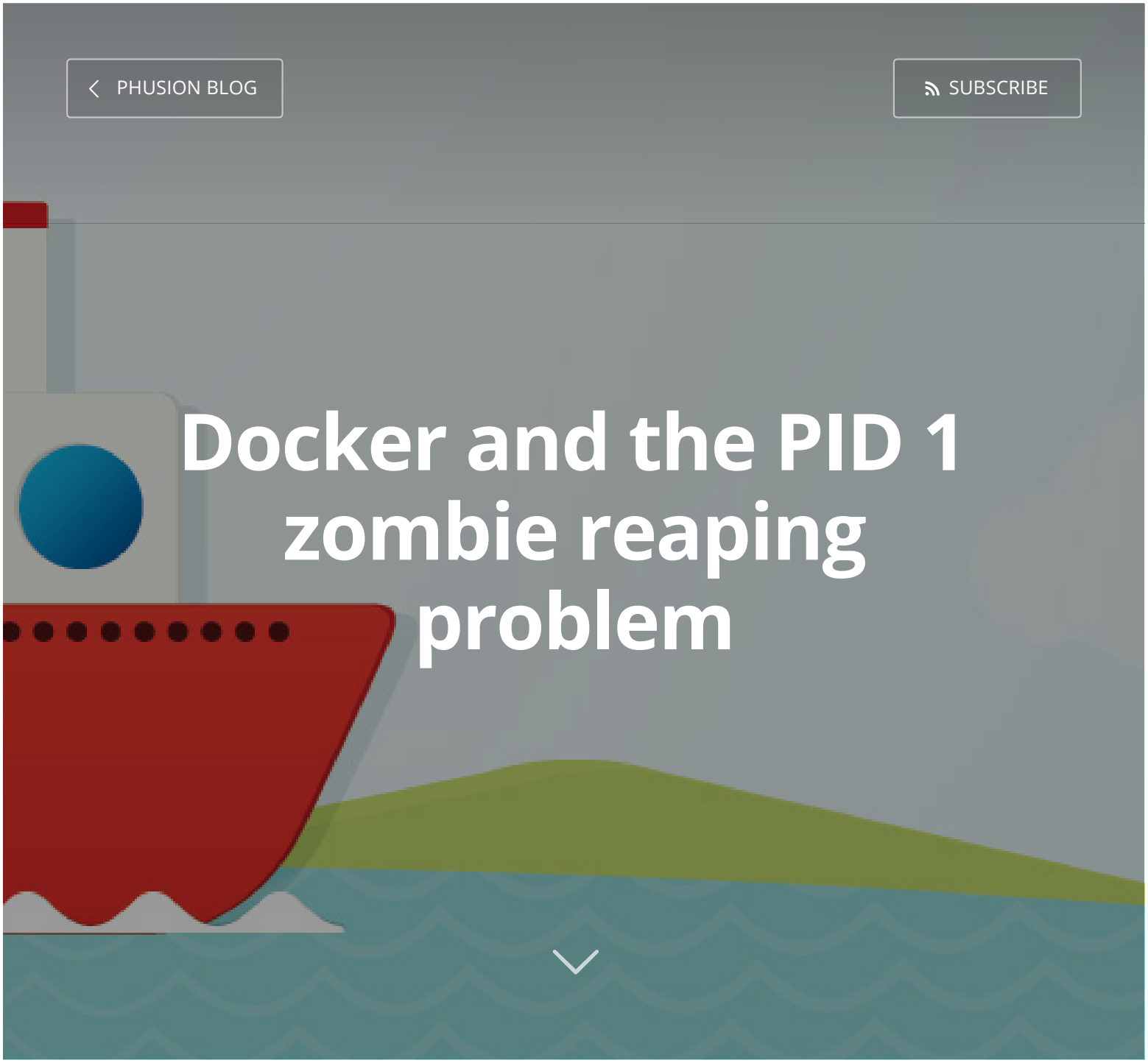


Docker and the PID 1 zombie reaping problem

20 JANUARY 2015 on baseimage-docker, docker, featured

When building Docker containers, you should be aware of the PID 1 zombie reaping problem. That problem can cause unexpected and obscure-looking issues when you least expect it. This article explains the PID 1 problem, explains how you can solve it, and presents a pre-built solution that you can use: Baseimage-docker.

When done, you may want to read part 2: Baseimage-docker, fat containers and "treating containers as VMs".

Introduction

About a year ago -- back in the Docker 0.6 days -- we first introduced Baseimage-docker. This is a minimal Ubuntu base image that is modified for Docker-friendliness. Other people can pull Baseimage-docker from the Docker Registry and use it as a base image for their own images.

We were early adopters of Docker, using Docker for continuous integration and for building development environments way before Docker hit 1.0. We developed Baseimage-docker in order to solve some problems with the way Docker works. For example, Docker does not run processes under a special init process that properly reaps child processes, so that it is possible for the container to end up with zombie processes that cause all sorts of trouble. Docker also does not do anything with syslog so that it's possible for important messages to get silently swallowed, etcetera.

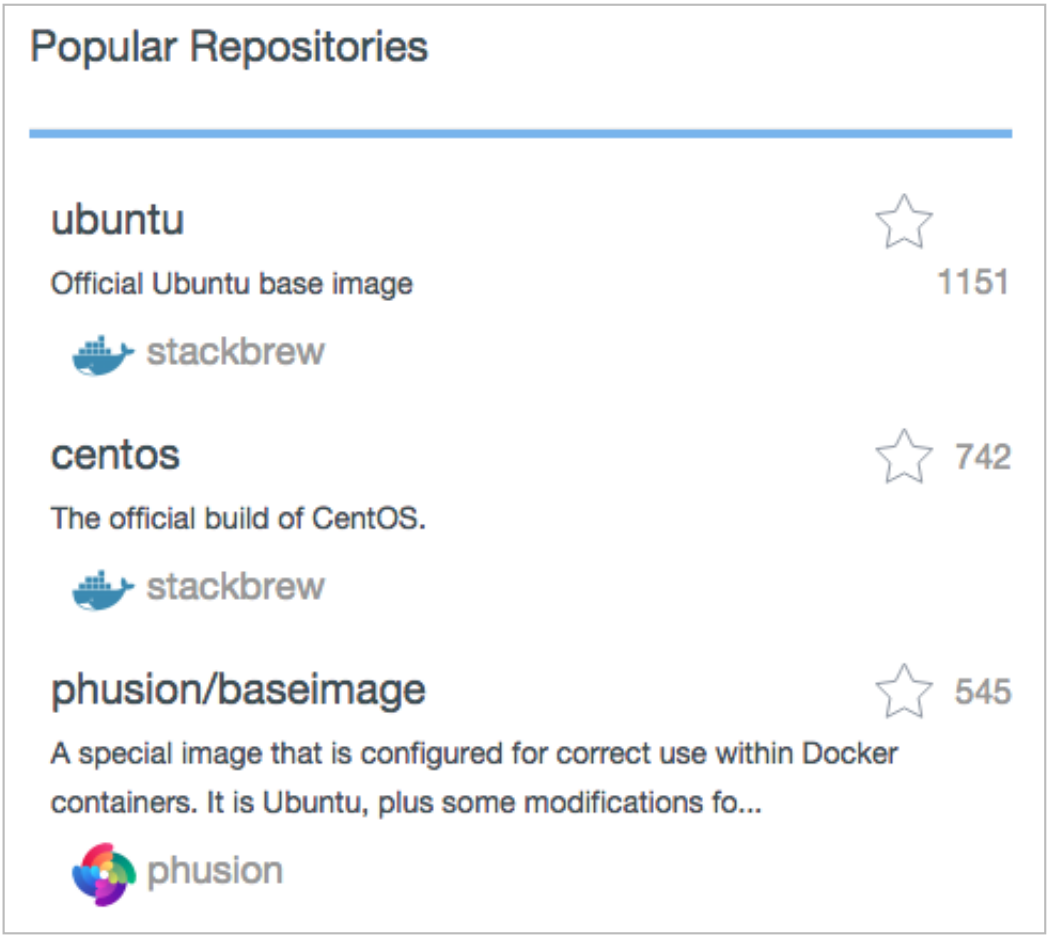
However, we've found that a lot of people have problems understanding the problems that we're solving. Granted, these are low-level Unix operating system-level mechanisms that few people know about or understand. So in this blog article we will describe the most important problem that we're solving -- the PID 1 problem zombie reaping problem -- in detail.

We figured that:

1. The problems that we solved are applicable to *a lot* of people.
2. Most people are not even aware of these problems, so things can break in unexpected ways (Murphy's law).
3. It's inefficient if everybody has to solve these problems over and over.

So in our spare time we extracted our solution into a reusable base image that everyone can use: Baseimage-docker. This image also adds a bunch of useful tools that we believe most Docker image developers would need. We use Baseimage-docker as a base image for all our Docker images.

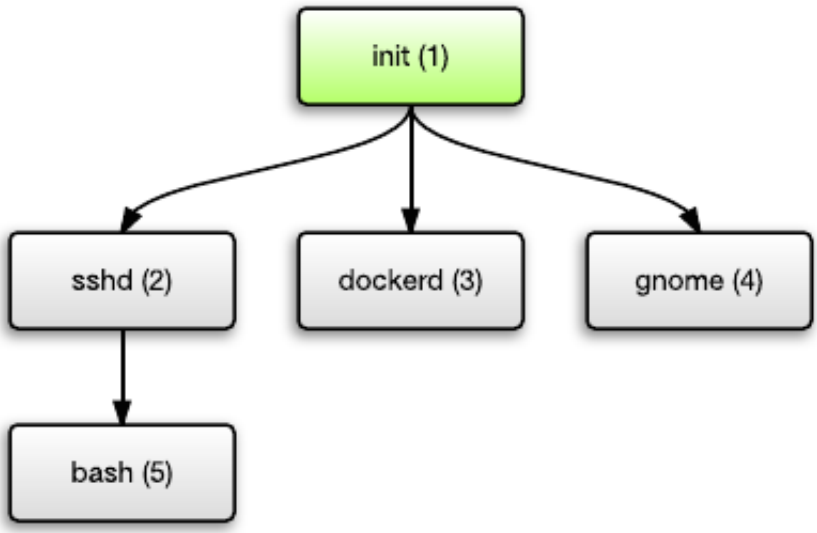
The community seemed to like what we did: we are the most popular third party image on the Docker Registry, only ranking below the official Ubuntu and CentOS images.



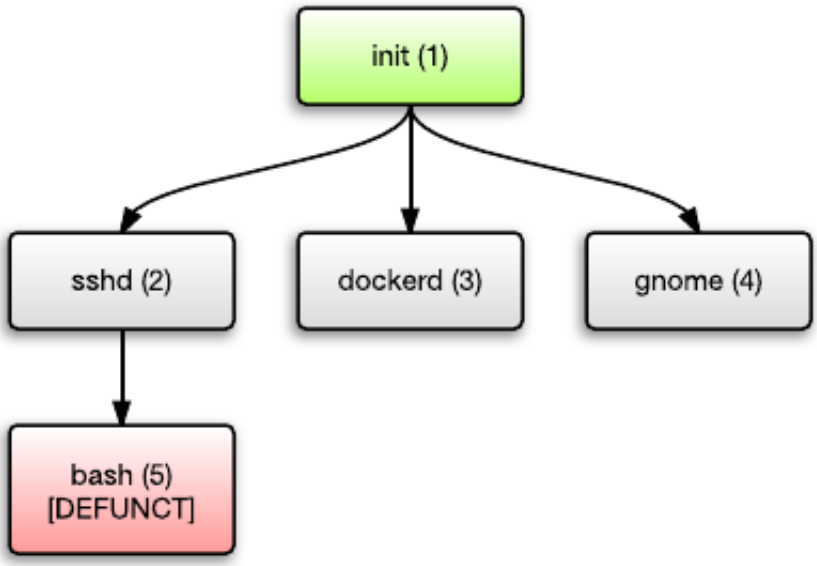
The PID 1 problem: reaping zombies

Recall that Unix processes are ordered in a tree. Each process can spawn child processes, and each process has a parent except for the top-most process.

This top-most process is the init process. It is started by the kernel when you boot your system. This init process is responsible for starting the rest of the system, such as starting the SSH daemon, starting the Docker daemon, starting Apache/Nginx, starting your GUI desktop environment, etc. Each of them may in turn spawn further child processes.



Nothing special so far. But consider what happens if a process terminates. Let's say that the bash (PID 5) process terminates. It turns into a so-called "defunct process", also known as a "zombie process".

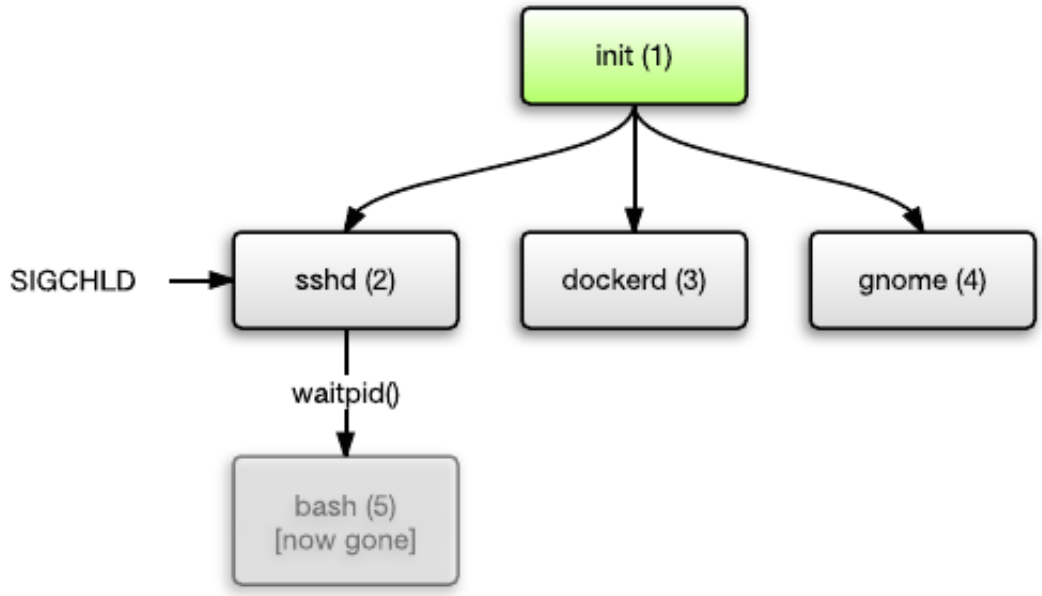


Why does this happen? It's because Unix is designed in such a way that parent processes must explicitly "wait" for child process termination, in order to collect its exit status. The zombie process exists until the parent process has performed this action, using the waitpid() family of system calls. I quote from the man page:

> "A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child."

In every day language, people consider "zombie processes" to be simply runaway processes that cause havoc. But formally speaking -- from a Unix operating system point of view -- zombie processes have a very specific definition. They are processes that have terminated but have not (yet) been waited for by their parent processes.

Most of the time this is not a problem. The action of calling `waitpid()` on a child process in order to eliminate its zombie, is called "reaping". Many applications reap their child processes correctly. In the above example with sshd, if bash terminates then the operating system will send a SIGCHLD signal to sshd to wake it up. Sshd notices this and reaps the child process.

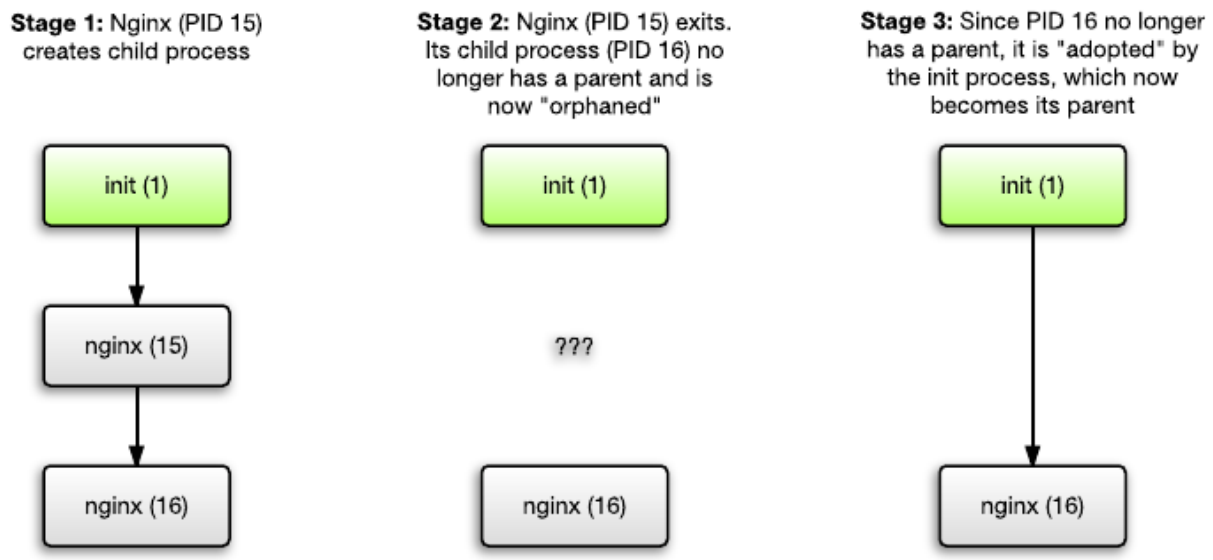


But there is a special case. Suppose the parent process

terminates, either intentionally (because the program logic has determined that it should exit), or caused by a user action (e.g. the user killed the process). What happens then to its children? They no longer have a parent process, so they become "orphaned" (this is the actual technical term).

And this is where the init process kicks in. The init process -- PID 1 -- has a special task. Its task is to "adopt" orphaned child processes (again, this is the actual technical term). This means that the init process becomes the parent of such processes, even though those processes were never created directly by the init process.

Consider Nginx as an example, which daemonizes into the background by default. This works as follows. First, Nginx creates a child process. Second, the original Nginx process exits. Third, the Nginx child process is adopted by the init process.

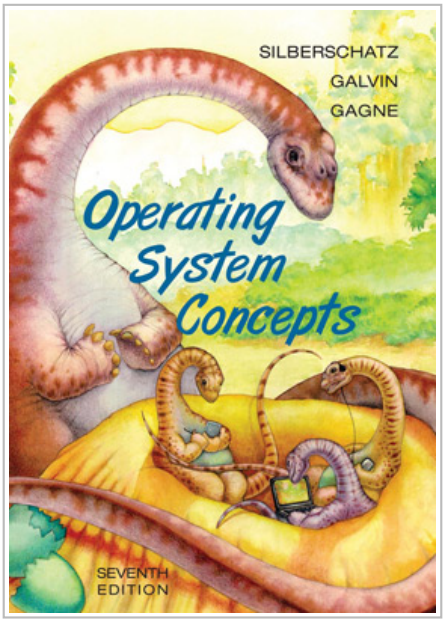


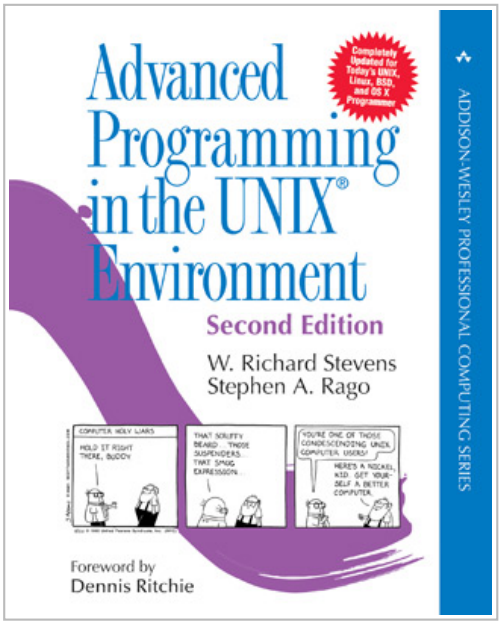
You may see where I am going. The operating system kernel automatically handles adoption, so this means that the kernel expects the init process to have a special responsibility: **the operating system expects the init process to reap adopted**

children too.

This is a very important responsibility in Unix systems. It is such a fundamental responsibility that many many pieces of software are written to make use of this. Pretty much all daemon software expect that daemonized child processes are adopted and reaped by init.

Although I used daemons as an example, this is in no way limited to just daemons. Every time a process exits even though it has child processes, it's expecting the init process to perform the cleanup later on. This is described in detail in two very good books: Operating System Concepts by Silberschatz et al, and Advanced Programming in the UNIX Environment by Stevens et al.





Why zombie processes are harmful

Why are zombie processes a bad thing, even though they're terminated processes? Surely the original application memory has already been freed, right? Is it anything more than just an entry that you see in `ps`?

You're right, the original application memory has been freed. But the fact that you still see it in `ps` means that it's still taking up some kernel resources. I quote the Linux waitpid man page:

> "As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes."

Relationship with Docker

So how does this relate to Docker? Well, we see that a lot of people run only one process in their container, and they think that when they run this single process, they're done. But most likely, this process is not written to behave like a proper init process. That is, instead of properly reaping adopted processes, it's probably expecting another init process to do that job, and rightly so.

Let's look at a concrete example. Suppose that your container contains a web server that runs a CGI script that's written in bash. The CGI script calls grep. Then the web server decides that the CGI script is taking too long and kills the script, but grep is not affected and keeps running. When grep finishes, it becomes a zombie and is adopted by the PID 1 (the web server). The web server doesn't know about grep, so it doesn't reap it, and the grep zombie stays in the system.

This problem applies to other situations too. We see that people often create Docker containers for third party applications -- let's say PostgreSQL -- and run those applications as the sole process inside the container. You're running someone elses code, so can you really be sure that those applications *don't* spawn processes in such a way that they become zombies later?

If you're running your own code, and you've audited all your libraries and all their libraries, then fine. But in the general case you *should* run a proper init system to prevent problems.

But doesn't running a full init system make the container heavyweight and like a VM?

An init system does not have to be heavyweight. You may be thinking about Upstart, Systemd, SysV init etc with all the implications that come with them. You may be thinking that full system needs to be booted inside the container. None of this is true. A "full init system" as we may call it, is neither necessary nor desirable.

The init system that I'm talking about is a small, simple program whose only responsibility is to spawn your application, and to reap adopted child processes. Using such a simple init system is completely in line with the Docker philosophy.

A simple init system

Is there already an existing piece of software that can run another application and that can reap adopted child processes at the same time?

There is **almost** a perfect solution that everybody has -- it's plain old bash. Bash reaps adopted child processes properly. Bash can run anything. So instead having this in your Dockerfile...

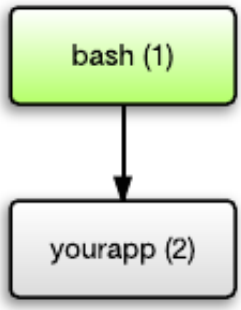
```
CMD ["/path-to-your-app"]
```

...you would be tempted to have this instead:

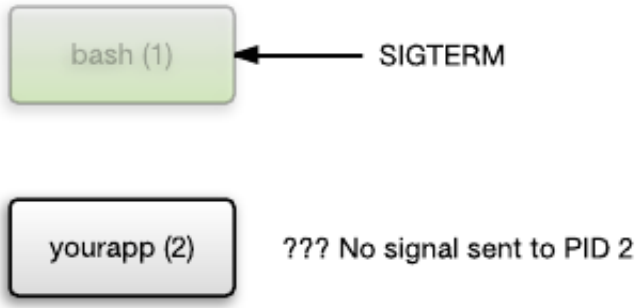
```
CMD ["/bin/bash", "-c", "set -e && /path-to-your-app"]
```

(The -e directive prevents bash from detecting the script as a simple command and `exec()`'ing it directly.)

This would result in the following process hierarchy:



But unfortunately, this approach has a key problem. It doesn't handle signals properly! Suppose that you use `kill` to send a SIGTERM signal to bash. Bash terminates, but does *not* send SIGTERM to its child processes!



When bash terminates, the kernel terminates the entire container with all processes inside. These processes are terminated *uncleanly* through the SIGKILL signal. SIGKILL cannot be trapped, so there is no way for processes to terminate

cleanly. Suppose that the app you're running is busy writing a file; the file could get corrupted if the app is terminated uncleanly in the middle of a write. Unclean terminations are bad. It's almost like pulling the power plug from your server.

But why should you care whether the init process is terminated by SIGTERM? That's because `docker stop` sends SIGTERM to the init process. "docker stop" should stop the container cleanly so that you can start it later with "docker start".

Bash experts would now be tempted to write an EXIT handler that simply sends signals to child processes, like this:

```
#!/bin/bash
function cleanup()
{
    local pids=`jobs -p`
    if [[ "$pids" != "" ]]; then
        kill $pids >/dev/null 2>/dev/null
    fi
}

trap cleanup EXIT
/path-to-your-app
```

Unfortunately, this does not solve the problem. Sending signals to child processes is not enough: the init process must also *wait* for child processes to terminate, before terminating itself. If the init process terminates prematurely then all children are terminated uncleanly by the kernel.

So clearly a more sophisticated solution is required, but a full

init system like Upstart, Systemd and SysV init are overkill for lightweight Docker containers. Luckily, Baseimage-docker has a solution for this. We have written a custom, lightweight init system especially for use within Docker containers. For the lack of a better name, we call this program my_init, a 350 line Python program with minimal resource usage.

Several key features of my_init:

- Reaps adopted child processes.
- Executes subprocesses.
- Waits until all subprocesses are terminated before terminating itself, but with a maximum timeout.
- Logs activity to "docker logs".

Will Docker solve this?

Ideally, the PID 1 problem is solved natively by Docker. It would be great if Docker supplies some builtin init system that properly reaps adopted child processes. But as of January 2015, we are not aware of any effort by the Docker team to address this. This is not a criticism -- Docker is very ambitious, and I'm sure the Docker team has bigger things to worry about, such as further developing their orchestration tools. The PID 1 problem is very much solvable at the user level. So until Docker has officially solved this, we recommend people to solve this issue themselves, by using a proper init system that behaves as described above.

Is this *really* such a problem?

At this point, the problem might still sound hypothetical. If you've never seen any zombie processes in your container then

you may be inclined to think that everything is all right. But the only way you can be sure that this problem never occurs, is when you have audited all your code, audited all your libraries' code, and audited all the code of the libraries that your libraries depend on. Unless you've done that, there *could* be a piece of code somewhere that spawns processes in such a way that they become zombies later on.

You may be inclined to think, I've never seen it happen, so the chance is small. But Murphy's law states that when things *can* go wrong, they *will* go wrong.

Apart from the fact that zombie processes hold kernel resources, zombie processes that don't go away can also interfere with software that check for the existence of processes. For example, the Phusion Passenger application server manages processes. It restarts processes when they crash. Crash detection is implemented by parsing the output of `ps`, and by sending a 0 signal to the process ID. Zombie processes are displayed in `ps` and respond to the 0 signal, so Phusion Passenger thinks the process is still alive even though it has terminated.

And think about the trade off. To prevent problems with zombie processes from ever happening, all you have to do is to is to spend 5 minutes, either on using Baseimage-docker, or on importing our 350 lines my_init init system into your container. The memory and disk overhead is minimal: only a couple of MB on disk and in memory to prevent Murphy's law.

Conclusion

So the PID 1 problem is something to be aware of. One way to solve it is by using Baseimage-docker.

Is Baseimage-docker the only possible solution? Of course not. What Baseimage-docker aims to do is:

1. To make people aware of several important caveats and pitfalls of Docker containers.
2. To provide pre-created solutions that others can use, so that people do not have to reinvent solutions for these issues.

This means that multiple solutions are possible, as long as they solve the issues that we describe. You are free to reimplement solutions in C, Go, Ruby or whatever. But why should you when we already have a perfectly fine solution?

Maybe you do not want to use Ubuntu as base image. Maybe you use CentOS. But that does not stop Baseimage-docker from being useful to you. For example, our `passenger_rpm_automation` project uses CentOS containers. We simply extracted Baseimage-docker's `my_init` and imported it there.

So even if you do not use, or do not want to use Baseimage-docker, take a good look at the issues we describe, and think about what you can do to solve them.

Happy Dockering.

***There is a part 2**: We will discuss the phenomenon that a lot of people associate Baseimage-docker with "fat containers". Baseimage-docker is not about fat containers at all, so what is it then? See Baseimage-docker, fat containers and "treating containers as VMs"*

Discuss this on Hacker News

Union Station is Phusion's brand new take on Passenger application monitoring and analytics. Union Station aims to help you easily find performance bottlenecks and errors in your application and to help you fix them. **Sign up for a free 1 month trial today!**

Hongli Lai

Read more posts by this author.

Share this post

Maciej Filipiak • 12 days ago

`set -e` is not for exec but for:

`-e Exit immediately if a command exits with a non-zero status.`

therefore you could just:

```
CMD ["sh", "-c", "exec /path-to-your-app"]
```

which will `Replace the shell with the given command` and solve the problem.

Hongli Lai Mod ➚ Maciej Filipiak • 12 days ago

That does not solve the problem. The `set -e` was in there exactly to *prevent* the shell from execing the command. The point was that the app itself does not reap zombies correctly, and that the shell *might* do that. By calling `set -e` and preventing the shell from execing the app, I was showing that the shell does reap zombies correctly but does not handle signals correctly, and thus using the shell is not a viable solution.

Mark Riggins • 22 days ago

You may find my `dockerfy` utility useful for reaping zombies, handling signals properly, starting services, pre-running initialization commands before the primary command starts, editing configuration files via templates, overlaying content and managing secrets.

https://github.com/markriggins...

for example,
dockerfy --secrets-files /secrets/secrets.env
--template /app/nginx.conf.tmpl:/etc/nginx/nginx.conf
--wait 'tcp://{{ .Env.MYSQLSERVER }}:{{ .Env.MYSQLPORT }}' --timeout 60s
--run '/app/bin/migrate_lock' --server='{{ .Env.MYSQLSERVER }}:{{ .Env.MYSQLPORT }}' --password='{{.Secret.MYSQLPASSWORD}}' --
--start /app/bin/cache-cleaner-daemon --
--stderr /var/log/nginx/error.log
--reap
--user nobody --

Reply • Share ›

win • a month ago

if the problem with using bash is only on waiting child process to be stopped, AFAIK, bash has wait built-in command, why not use that?

Reply • Share ›

Hongli Lai Mod → win • a month ago

That has the same problem. It doesn't handle signals properly.

Reply • Share ›

Matthew Fernandez • 2 months ago

This is a legit problem, but why didn't you just use sinit (http://core.suckless.org/sinit... You say "Is there already an existing piece of software that can run another application and that can reap adopted child processes at the same time?" and the answer is "yes, sinit." Even if you had to modify it a little to add Docker logging, you're still better off than writing your own from scratch. Am I missing something?

Reply • Share ›

Hongli Lai Mod → Matthew Fernandez • 2 months ago

I didn't know about sinit. It's not exactly a well-known tool. I just had a look at the source code, and decided that it doesn't satisfy my requirements. For example sinit hardcodes the initializer command and the poweroff command, instead of accepting them via command line parameters. sinit does not properly shutdown when the initializer command shuts down. sinit also does not handle SIGTERM, which Docker uses during shutdown. This does not surprise my because sinit is not written specifically for Docker environments. `my_init` *is* specifically written for Docker environments.

Reply • Share ›

David Wragg • 2 months ago

tini (https://github.com/krallin/tin... is a small C program that solves the same problem. It's about 10KB dynamically linked, or <1MB statically linked.

Reply • Share ›

Vitor Mattos • 6 months ago

Thank you for this excellent article! It really helped me solve my problem!

Reply • Share ›

James Coulter • 7 months ago

Thanks for the excellent article. How much disk space does the Python installation (required to run my_init) take in the container? Python can be heavy on disk usage, unless it happens to be needed for the application anyway. Otherwise, a C or C++ -based solution would be preferable, since it would be light both in execution overhead and in disk overhead.

Reply • Share ›

mmuurr • 7 months ago

In your simple bash script example, would a `wait [PID ...]` command not work after the SIGKILLs are sent to the children (i.e. inside the EXIT trap)? (I ask fully realizing the other features of baseimage-docker wouldn't be there, but just out of curiosity regarding the waiting-on-children issue.)

Reply • Share ›

Hongli Lai Mod mmuurr • 7 months ago

No, because not only must the init process wait for its own children, it must also wait for adopted children and the children's children.

Reply • Share ›

mmuurr Hongli Lai • 7 months ago

Fair point. But if the basic bash script is run as PID 1, it'd be the session leader, right? Then would it be possible to find all other PIDs under that session leader (possibly using recursion), enumerate them, then call kill/wait on each? I might be confused about the role of session leaders in containers, but my (naive) guess is that unless a new session is explicitly created (in which case all bets are off), all forking off the PID 1 process will use that PID 1 process as the session leader.

Not trolling here, just thinking aloud :-)

Reply • Share ›

Geo • 8 months ago

Great post! You explained this simply and clearly, thank you!

Reply • Share ›

Romuald Brunet • 9 months ago

Not sure I had the same exact problem (and maybe that's a stupid solution), but solved my zombie problems with this simple bash script as "init":

```
#!/bin/bash -me
$@
```

Reply • Share ›

Rafael Capucho • 9 months ago

Hello, thank you for your article. I'm suffering from the problem of being unable to graceful stop a process inside a container. I'm using simple Python3.5 (https://hub.docker.com/r/nexus...) and using a simple cmd (in marathon: https://paste.ee/r/w5wFP) how/where I add your init script? It was not clear for me how to perform. Thank you very much.

Reply • Share ›

pickaname • 9 months ago

Wow, you calling upstart&systemd an overkill, but implement your own

solution in python!? You got to be kidding me.

Reply • Share ›

Hongli Lai Mod ↗ pickaname • 9 months ago

Our solution is less than 300 lines. Can you say that about Upstart and Systemd?

Our solution also does not manage hardware. *That* is overkill in the context of Docker, but that's also exactly what Upstart and Systemd try to do.

Our init system is designed from the ground up to adhere to Docker best practices, like ensuring everything logs to stdout instead of log files, handling Unix signals that the Docker daemon sends, not assuming that we have too many privileges in a container, etc. You cannot say that about Upstart and Systemd.

2 Reply • Share ›

ALSO ON **PHUSION BLOG**

What's new in Passenger 5 part 2: better logging, better restarting,

3 comments • a year ago•

Heroku and Passenger: focus on the app-performance

1 comment • 9 months ago•