Trapping signals in Docker containers

Grigorii Chudnov

Apr 7, 2015 · 5 min read

Have you ever stopped a Docker container?

There are essentially two commands: `docker stop` and `docker kill` that can be used to stop it. Behind the scenes, `docker stop` stops a running container by sending it SIGTERM signal, let the main process process it, and after a grace period uses SIGKILL to terminate the application.

Running applications in Docker, you could use signals to communicate with an application from the host machine, reload configuration, do some cleanup on program termination or coordinate more than one executable.

Let's see the way signals can be used in the Docker environment.

Signals

Signals represent a form of inter-process communication. A signal is a message to a process from the kernel to notify that some condition has

occurred [1].

When a signal is issued to a process, the process is interrupted and a signal handler is executed. If there is no signal handler, the default handler is called instead.

The process tell the kernel the types of signals it is interested in to handle in a non-default way by providing handler functions (callbacks). To register a signal handler, use one of the signal API functions[2].

When you run the `kill` command in a terminal, you're asking the kernel to send a signal to another process.

One common signal is SIGTERM that tells the process to shut down and terminate. It could be used to close down sockets, database connections, or remove any temporary files. Many daemons accept SIGHUP to reload configuration files. SIGUSR1 and SIGUSR2 are the user-defined signals and can be handled in an application-specific way.

For example, to set up SIGTERM signal in node.js:

```
process.on('SIGTERM', function() {
  console.log('shutting down...');
});
```

When SIGTERM is handled by a process, the normal sequence of execution is temporarily interrupted by the provided signal handler.

The function is executed, and the process continues to run from the point it was interrupted.

Here is the list of common signals. See [2] for the full reference.

Name	Default action	Description
SIGHUP	Terminate process	Terminal line hangup
SIGINT	Terminate process	Interrupt program
SIGQUIT	Create core image	Quit program
SIGABRT	Create core image	Abort program
SIGKILL	Terminate process	Kill program
SIGTERM	Terminate process	Software termination signal
SIGUSR1	Terminate process	User defined signal 1
SIGUSR2	Terminate process	User defined signal 2

All signals except for SIGKILL and SIGSTOP can be *intercepted* by the process.

. . .

Signals in Docker

A docker command `docker kill` used to send a signal to the main process inside a container.

```
Usage: docker kill [OPTIONS] CONTAINER [CONTAINER...]

Kill a running container using SIGKILL or a specified signal

  -s, --signal="KILL"    Signal to send to the container
```

The signal sent to container is handled by the main process that is running (PID 1).

The process could ignore the signal, let the default action occur or provide a callback function for that signal.

As an example, let's run the following application *(program.js)* inside a container and examine signal handlers.

```
'use strict';

var http = require('http');

var server = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(3000, '0.0.0.0');

console.log('server started');

var signals = {
  'SIGINT': 2,
  'SIGTERM': 15
};
```

```
function shutdown(signal, value) {
  server.close(function () {
    console.log('server stopped by ' + signal);
    process.exit(128 + value);
  });
}

Object.keys(signals).forEach(function (signal) {
  process.on(signal, function () {
    shutdown(signal, signals[signal]);
  });
});
```

Here we create an HTTP-server that listens on port 3000 and setup two signal handlers for SIGINT and SIGTERM. When handled, the following message will be printed to *stdout*: **`server stopped by [SIGNAL]`**.

. . .

Application is the foreground process (PID1)

When the application is the main process in a container (PID1), it could handle signals directly.

Here is the Dockerfile to build an image based on io.js:

```
FROM iojs:onbuild

COPY ./program.js ./program.js
COPY ./package.json ./package.json
```

```
EXPOSE 3000

ENTRYPOINT ["node", "program"]
```

> *When creating a Dockerfile, make sure you use exec form of ENTRYPOINT or CMD commands. Otherwise the application will be started as a subcommand of /bin/sh -c, which does not pass signals. The container's PID1 will be the shell, your application will not receive any signals from the `docker kill` command.*

Build the image:

```
$ docker build -t signal-fg-app .
```

Run the container:

```
$ docker run -it --rm -p 3000:3000 --name="signal-fg-app"
signal-fg-app
```

Visit *http://localhost:3000* to verify the application is running.

Open a new terminal and issue the `docker kill` command:

```
$ docker kill --signal="SIGTERM" signal-fg-app
```

or

```
$ docker stop signal-fg-app
```

Both commands can be used to issue SIGTERM signal and stop the application.

Application handles the signal and you should see the following message in the terminal you run the container in:

```
server stopped by SIGTERM
```

. . .

Application is the background process (not PID1)

The process to be signaled could be the background one and you cannot send any signals directly. In this case one solution is to set up a shell-script as the entrypoint and orchestrate all signal processing in that script.

An updated Dockerfile:

```
FROM iojs:onbuild

COPY ./program.js ./program.js
COPY ./program.sh ./program.sh
COPY ./package.json ./package.json

RUN  chmod +x ./program.sh

EXPOSE 3000

ENTRYPOINT ["./program.sh"]
```

The new entrypoint is a bash-script *program.sh* that orchestrates signal processing:

```
#!/usr/bin/env bash
set -x

pid=0

# SIGUSR1-handler
my_handler() {
  echo "my_handler"
}

# SIGTERM-handler
term_handler() {
  if [ $pid -ne 0 ]; then
    kill -SIGTERM "$pid"
    wait "$pid"
  fi
  exit 143; # 128 + 15 -- SIGTERM
```

```
}

# setup handlers
# on callback, kill the last background process, which is
`tail -f /dev/null` and execute the specified handler
trap 'kill ${!}; my_handler' SIGUSR1
trap 'kill ${!}; term_handler' SIGTERM

# run application
node program &
pid="$!"

# wait forever
while true
do
  tail -f /dev/null & wait ${!}
done
```

Here we set up two signal *handlers*: one for the user-defined signal, SIGUSR1 and the second one—SIGTERM to gracefully shut down the application.

> *The Single UNIX Specification specifies the functions that are guaranteed to be safe to call from within a signal handler. These functions are reentrant and are called async-signal safe by the Single UNIX Specification.*

In the script, we put our application in the *background* with the ampersand (&): `node program &`

Finally, we use `wait` to suspend execution until a child process exits. `wait` and `waitpid` are the functions that always interrupted when a

signal is caught. After the signal kicks-in, we process it in the specified handler and wait for a signal again.

> *According to the Docker docs, SIGCHLD, SIGKILL, and SIGSTOP are not proxied.*

To run the code, open a new terminal window and build the image:

```
docker build -t signal-bg-app .
```

Run the container:

```
docker run -it --rm -p 3000:3000 --name="signal-bg-app"
signal-bg-app
```

Open a new terminal and issue SIGUSR1:

```
docker kill --signal="SIGUSR1" signal-bg-app
```

Finally, to stop the application:

```
docker kill --signal="SIGTERM" signal-bg-app
```

The application should terminate gracefully with the corresponding message printed to *stdout*.

. . .

Conclusion

Signals provide a way of handling asynchronous events and can be used by applications running in Docker containers. Use signals to communicate with an application from the host machine, reload configuration, do some cleanup or coordinate multiple processes.

> *Source code can be found in the GitHub repo.*

The people over at Codeship liked my article so much that they asked me to republish it on their blog

. . .

References

1. Overview of signals http://man7.org/linux/man-pages/man7/signal.7.html

2. Michael Kerrisk (2010), The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press