Init for Docker and Kubernetes.

- Stuff good to know.
- How to do it with Python.

Agenda

- Linux boot and init
- Signals
- Signals: init is different
- Processes: system vs. exec
- Processes: exit value
- Orphan processes
- Init: the orphanage

Agenda

Let's get started...

- **Linux boot and init**
- Signals
- Signals: init is different
- Processes: system vs. exec
- Processes: exit value
- Orphan processes
- Init: the orphanage

Linux boot
(simplified)

- BIOS or UEFI loads boot loader, e.g., grub
- grub loads kernel + initrd ("initial RAM disk")
- kernel starts
- kernel mounts initrd as file system /
- kernel starts some init program
- that init program further initializes
  file systems, hardware, and processes as wanted

Linux shutdown
(simplified)

When the init process stops running,
the kernel switches off the light.

"init" is just a conventional name

- The name of the program does not matter.
- Whichever process has process id 1 is init.
- That process id 1 is originally owned by the process first started by the kernel.
- A process can retire and ordain a successor that inherits the process id. More on that later.

Experiment (2 Terminals)

- `docker pull python:3.10-bullseye`

  ```
  time docker run -ti --rm --name=nn \
  python:3.10-bullseye \
  python3 -c 'import time; time.sleep(30)'
  ```

- `docker exec nn ps -fax`

The initial process started in a Docker container
is init inside that container
(has process id 1).

A Kubernetes POD is typically
just another Docker container (or several).

What we try out here for Docker applies directly to Kubernetes.

Agenda

- Linux boot and init ✔
- **Signals**
- Signals: init is different
- Processes: system vs. exec
- Processes: exit value
- Orphan processes
- Init: the orphanage

Signals

- A mechanism the kernel uses to communicate with programs.
- Different signals exist.
- The same signal can be referred to by name or number.
- Examples:
  - You are writing to a pipe nobody reads (SIGPIPE, 7)
  - Your terminal has disconnected (SIGHUP, 1)
  - You access memory illegally (SIGSEGV, 11)
  - **Please shut down (SIGTERM, 15)**
  - You are dead (SIGKILL, 9)

Numbers dependent on CPU architecture.

- For the complete list (there are dozends of signals):

```
man 7 signal
```

Handle signals

- The program registers a "signal handler" with the kernel.
- That is a function to be called by the kernel if the kernel wants to hand over a signal to the program.
- This function calls happens **in parallel**, whenever the kernel sees fit, independent of what else the program is doing.
- Language: The program "handles" or "has caught" the signal.
- Exception: Signal sigkill 9 "you are dead"
  cannot be caught or handled.

Signals and Python

parallel is never easy

- Give your handler function to `signal.signal`
- When the signal arrives, your function will be called on the "main thread",
- but still in the middle of other stuff running on the main thread, interrupting that other stuff!
- You can use `threading.Lock` to synchronize data exchange.
- Doing so naively can give rise to dead locks:
  The main thread cannot give up the lock while being blocked by the handler trying to acquire that same lock.
- Possible solution: Let your handler function start a different thread and acquire locks only on that thread.

Example: handle `sigterm`

```
import signal
import threading
import queue

lock = threading.Lock()
threads_to_be_joined = queue.Queue() # Joining not shown.

def terminate(): ... # Access here, you have the lock.

def sigterm_handler(_signo, _stackframe):
  def handle_in_thread():
    threads_to_be_joined.put(threading.current_thread())
    with lock:
      terminate()
  threading.Thread(target = handle_in_thread).start()


signal.signal(signal.SIGTERM, sigterm_handler)
```

Default action

Kernel wants to send a certain signal,
but the program hasn't registered a handler for that signal.

Now, what?

There's a default action defined in the kernel for that case.

What actually happens depends on the signal, but most signals' default actions terminate the program.

Agenda

- Linux boot and init ✔
- Signals ✔
- **Signals: init is different**
- Processes: system vs. exec
- Processes: exit value
- Orphan processes
- Init: the orphanage

Init is different

When init terminates,
the entire Linux machine stops
(or the entire Docker container terminates).

For that reason, the kernel default action code
is special-cased for init.

If the kernel wants to send a signal to process 1
and there's no signal handler for that signal,
the default action is to ignore the signal.

Experiment

At some Linux shell prompt (maybe acquired via
`docker run --name nn -ti --rm python:3.10-bullseye /bin/bash  )`

- `python -c 'import time; time.sleep(60)'`
- `ps ax | grep -P '\d python3.+import time'`
  `kill -9 NNNNN`

Now let's make the first the init process
in a Docker container:

prepend the first line with
`docker run --name nn -ti --rm python:3.10-bullseye`

and the other lines with
`docker exec -ti nn /bin/bash`

Experiment

- This program has no handler for sigterm:

```
docker run --name nn -ti --rm python:3.10-bullseye \
  python3 -c 'import time; time.sleep(120)'
```

If this program is your init, Docker (or Kubernetes)
will not be able to shut it down cleanly.

- `time docker stop nn`

After (typically) 10 seconds after sending the signal,
the system becomes impatient
and crashes your program.

One job of init in Docker: sigterm

In a Docker container, init should handle sigterm.

- If your init is your application,
  your application should cleanly and swiftly shut down
  (saving what may need to be saved).
- If your init starts the real application,
  init should forward sigterm to the real application
  and wait for it to shut down.

If crashing is all you want your app to do,
why wait 10 seconds?

Agenda

- Linux boot and init ✔
- Signals ✔
- Signals: init is different ✔
- **Processes: system vs. exec**
- Processes: exit value
- Orphan processes
- Init: the orphanage

System vs. exec

- "system"
  - Python: `subprocess.run` or `subprocess.popen` (old style: `os.system`).
  - This is also what the shell normally does.
- starts a new "child" program
- the "parent" program can
  - interact
  - find out what became of it

We'll get to that.

System vs. exec

- "exec"
  - in Python: `os.exec*` functions
  - in Shell: `exec`
- replaces the program that calls "exec"
  with a successor.
- The successor inherits the process id.
- So the successor of init still has process id 1,
  so it inherits the init role with all peculiarities and duties.

Try out system and exec

- `docker run --name n1 -ti --rm python:3.10-bullseye \`
  `/bin/sh -c 'sleep 90'`
- `docker exec -ti n1 ps fax`
  `docker stop n1`
- `docker run --name n2 -ti --rm python:3.10-bullseye \`
  `/bin/sh -c 'exec sleep 90'`
- `docker exec -ti n2 ps fax`
  `docker stop n2`

Agenda

- Linux boot and init ✔
- Signals ✔
- Signals: init is different ✔
- Processes: system vs. exec ✔
- **Processes: exit value**
- Orphan processes
- Init: the orphanage

The final verdict: The exit value.

UNIX / Linux – Convention:

- Exit value 0 means: All's well
- Any exit value != 0 (1 to 255) means: failure
- In Python: `subprocess.run( …, check=true, …)` translates the exit value of the program called into an error that's raised, if not 0.
- The shell provides the exit value of the last program it ran via `$?`

Experiment:

```
true; echo $?
false; echo $?
python -c 'import sys; sys.exit(7)'; echo $?
```

Job von Docker - init

Be honest to your runtime environment
and tell it whether all is well or not.

Beginner's mistake:

The application cannot reach its database
but does not immediately fail (with an exit value != 0),
but keeps on going.

In a Kubernetes world, this will lead to the previous version
(that may still have been able to reach the database)
to be replaced by the new broken one (that no longer is).

Parents, watch your kids!

- Processes are in a hirarchy.
- A process X start another process Y:
  X becomes the parent of Y, has a child to take care of.
- Parent processes typically outlive their children.
- A parent has some moral obligation to process the exit value of all its children.
- Under the hood:
  The kernel tells you a child terminated via the SIGCHLD-signal.

Agenda

- Linux boot and init ✔
- Signals ✔
- Signals: init is different ✔
- Processes: system vs. exec ✔
- Processes: exit value ✔
- **Orphan processes**
- Init: the orphanage

Orphan processes

- Each running process uses up quite a few resources, in particular: One slot in the kernel's process table. (The process id is the index into that table.)
- After termination, the process still occupies one slot in the kernel's process table.
- This is where the kernel keeps the exit value until someone comes and asks for it.
- A terminated process with nobody asking for the exit value is called "orphan" or "zombie process" or "defunct".

Let's create a zombie

- `docker run -ti --name=nn --rm python:3.10-bullseye \`
  `sh -c 'exec sleep 600'`
- `docker exec -ti nn sh -c 'sleep 2 & ps fax'`
  `docker exec -ti nn ps fax`

A zombie process in a container occupies a process table slot of the Docker host or Kubernetes node!

- `ps ax | grep defunct # On Linux, you see the zombie`

Agenda

- Linux boot and init ✔
- Signals ✔
- Signals: init is different ✔
- Processes: system vs. exec ✔
- Processes: exit value ✔
- Orphan processes ✔
- **Init: the orphanage**

Init to the rescue, init the orphanage

- The kernel reports zombie processes to init.
- Init can and should ask the kernel for the exit value.
- If init fails to do this, a zombie will remain.
- In extreme cases, no new processes can be started inside the container. That means: No shell script can run.
- In even more extreme cases, no new processes can be run on the Docker host or Kubernetes node.
- This is a appalling state to be in.
  You don't want to go there.

We're done with the agenda.

- Linux boot and init ✔
- Signals ✔
- Signals: init is different ✔
- Processes: system vs. exec ✔
- Processes: exit value ✔
- Orphan processes ✔
- Init: the orphanage ✔

But…

Shameless plug / bonus material

(this and one more slide)

yasinit

"Yet another simple init"

https://github.com/aknrdureegaesr/yasinit

Python3, Apache License.

Has this talk in `branch_talk` .

Design decisions yasinit

- Implement using Python.
- Can start one or several programs.
  - One possible via command line interface,
  - any number via simple config file.
- If one of the programs terminates, give up. Restarting left to Kubernetes, systemd, …
- Log via stderr.

Questions? Remarks?