Ein Python-init für Docker (und Kubernetes)

Inhalt

- Linux-Bootprozess und init
- Signale
- Signale: init ist anders
- Prozesse: system vs. Exec
- Prozesse: Returnwert
- Verwaiste Prozesse
- Init: Das Waisenhaus

Inhalt

- **Linux-Bootprozess und init**
- Signale
- Signale: init ist anders
- Prozesse: system vs. Exec
- Prozesse: Returnwert
- Verwaiste Prozesse
- Init: Das Waisenhaus

Linux-Bootprozess
(vereinfacht)

- BIOS oder UEFI lädt Bootloader, z.B. grub
- grub lädt Kernel + Initrd ("initial RAM disk")
- Kernel startet
- Kernel mount-et Initrd als Dateisystem /
- Kernel startet das init-Programm
- Das init-Programm initialisiert Dateisysteme und Prozesse nach Wunsch

Linux-Shutdown
(vereinfacht)

Wenn das init – Programm aufhört, zu laufen macht der Kernel das Licht aus.

"init" ist nur ein Name

- Für die Sonderrollen von init ist es egal, wie das Programm heißt.
- Wichtig ist nur, dass es die Prozessnummer 1 hat.
- Die Prozessnummer 1 hat der erste Prozess, den der Kernel gestartet hat.
- (Ein Prozess kann abdanken und einen Nachfolger bestimmen, der die Prozessnummer übernimmt. Das kommt noch.)

Experiment (2 Terminals)

- `docker pull python:3.8-buster`

  ```
  time docker run -ti --rm --name=nn \
  python:3.8-buster \
  python3 -c 'import time; time.sleep(30)'
  ```

- `docker exec nn ps -fax`

Der in einem Dockercontainer gestartete Prozess ist dort "init" (hat Prozessnummer 1).

Ein Kubernetes POD ist normalerweise auch nur ein (oder mehrere) Dockercontainer.

Inhalt

- Linux-Bootprozess und init ✔
- **Signale**
- Signale: init ist anders
- Prozesse: system vs. Exec
- Prozesse: Returnwert
- Verwaiste Prozesse
- Init: Das Waisenhaus

Signale

- Mechanismus für den Kernel, dem Programm etwas mitzuteilen
- Verschiedene Signale, Namen oder Zahl synonym
- Beispiele:
  - Du schreibst in eine Pipe, die niemand liest (SIGPIPE, 7)
  - Dein Terminal ist weg (SIGHUP, 1)
  - Du greifst illegal auf Speicher zu (SIGSEGV, 11)
  - **Mach bitte Feierabend (SIGTERM, 15)**
  - Du bist tot (SIGKILL, 9)
- Komplette Liste (mehrere Dutzend Signale):
  `man 7 signal`

Signale entgegennehmen

- Das Programm registriert beim Kernel einen "Signal Handler".
- Das ist eine Funktion, die der Kernel aufruft, wenn er das Signal abgeben will.
- Das passiert mittendrin, **nebenläufig,** wenn der Kernel will, unabhängig vom Programmfluss.
- Man sagt auch, man habe das Signal "gefangen".
- Ausnahme: Signal sigkill 9 "du bist tot"
  kann nicht gefangen werden.

Signale mit Python

Parallelität wird manchmal etwas unangenehm

- Handlerfunktion an `signal.signal` übergeben.
- Handleraufruf kommt auf dem "main Thread"
- … aber trotzdem nebenläufig!
- Synchronisation mit `threading.Lock`
- Deadlockgefahr: Main kann das Lock nicht abgeben, während Main (im Handler) auf das Lock wartet und Main deshalb blockiert.
- Mögliche Lösung: Handler startet neuen Thread.

Beispiel: `sigterm` fangen

```
import signal
import threading
import queue

lock = threading.Lock()
threads_to_be_joined = queue.Queue()


def terminate(): ...

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)
```

Standardverhalten

Kernel will Signal abgeben, aber
Programm hat keinen Handler dafür registriert?
Dann greift (der Kernel zu) Standardverhalten.

Was genau passiert, hängt vom konkreten Signal ab, das zugestellt werden soll.

In der Regel wird das Programm abgeräumt.

Inhalt

- Linux-Bootprozess und init ✔
- Signale ✔
- **Signale: init ist anders**
- Prozesse: system vs. Exec
- Prozesse: Returnwert
- Verwaiste Prozesse
- Init: Das Waisenhaus

Init ist anders

Wenn `init` fertig ist, falten Linux-Maschine oder Dockercontainer zusammen.

Daher ist das Standardverhalten für init anders.

Wenn der Kernel ein Signal an Prozess 1 nicht zustellen kann, ignoriert er es.

Experiment

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

Anschließend dasselbe im Dockercontainer

erste Zeile mit vorgestelltem

`docker run --name nn -ti --rm python:3.8-buster`

die anderen beiden in der Shell

`docker exec -ti nn /bin/bash`

Experiment

- Dieses init hat keinen Handler für Sigterm:

```
docker run --name nn -ti --rm python:3.8-buster \
  python3 -c 'import time; time.sleep(120)'
```

Das kann Docker (Kubernetes)
nicht sauber runterfahren.

- `time docker stop nn`

Nach 10 Sekunden
verliert das System die Geduld
und "schaltet den Strom ab" (Crash!).

Ein Job von init in Docker: Sigterm

Im Dockercontainer sollte init Sigterm behandeln!

- Wenn init selbst die Anwendung ist,
  sich selbst zügig sauber beenden
  (noch schnell Status retten).
- Wenn init die eigentliche Anwendung startet,
  Sigterm an die Anwendung weitergeben
  und warten, bis die Anwendung sich beendet hat.

Inhalt

- Linux-Bootprozess und init ✔
- Signale ✔
- Signale: init ist anders ✔
- **Prozesse: system vs. Exec**
- Prozesse: Returnwert
- Verwaiste Prozesse
- Init: Das Waisenhaus

System vs. exec

- "system"
  - in Python: `os.system` oder moderner `subprocess.run` bzw. `subprocess.Popen`
  - in Shell: Normalfall.
- startet ein neues Programm
- und man kann aus dem laufenden Programm
  - damit interagieren
  - später schauen, was daraus geworden ist (kommt gleich).

System vs. exec

- "exec"
  - in Python: `os.exec*` Funktionen
  - in Shell: `exec`
- ersetzt das laufende Programm durch einen Nachfolger
- der Nachfolger erbt die Prozessid (ist also ggf. immer noch Prozess 1, also init mit allen Rechten und Pflichten)

Experiment

- `docker run --name n1 -ti --rm python:3.8-buster \`
  `/bin/sh -c 'sleep 90'`
- `docker exec -ti n1 ps fax`
  `docker stop n1`
- `docker run --name n2 -ti --rm python:3.8-buster \`
  `/bin/sh -c 'exec sleep 90'`
- `docker exec -ti n2 ps fax`
  `docker stop n2`

Inhalt

- Linux-Bootprozess und init ✔
- Signale ✔
- Signale: init ist anders ✔
- Prozesse: system vs. Exec ✔
- **Prozesse: Returnwert**
- Verwaiste Prozesse
- Init: Das Waisenhaus

Das Endergebnis: Der Exitwert

UNIX / Linux – Konvention:

- Exitwert 0 bedeutet: Alles gut
- Exitwert != 0 (1 bis 255) bedeutet: Programm gescheitert.
- `subprocess.run( …, check=true, …)`
  übersetzt den Exitwert des aufgerufenen Programms
  in eine Python-Exception.

Experiment:

```
true; echo $?
false; echo $?
python3 -c 'import sys; sys.exit(7)'; echo $?
```

Job von Docker - init

Der Laufzeitumgebung ehrlich mitteilen,
ob alles gut war oder nicht.

Anfängerfehler:

Die Anwendung kommt nicht an die Datenbank,
aber scheitert nicht sofort (mit Exitwert != 0)
sondern macht weiter.

Das kann dazu führen, das Kubernetes die vorige
Version, die noch an die Datenbank kam, abräumt
und durch die neue, kaputte ersetzt.

Eltern, hütet Eure Kinder!

- Prozesse sind hierarchisch geordnet.
- Wer einen Prozess startet, ist der Elternprozess und hat nun einen Kindprozess.
- Elternprozesse überleben in der Regel ihre Kindprozesse.
- Der Elternprozess hat die moralische Verpflichtung, den Exitwert der Kinder zu verarbeiten.
- Unter der Motorhaube: SIGCHLD-Signal

Inhalt

- Linux-Bootprozess und init ✔
- Signale ✔
- Signale: init ist anders ✔
- Prozesse: system vs. Exec ✔
- Prozesse: Returnwert ✔
- **Verwaiste Prozesse**
- Init: Das Waisenhaus

Waisenprozesse

- Ein laufender Prozess belegt viele Resourcen, insbesondere einen Slot in der Prozesstabelle des Kernels (die PID ist der Index des Slots).
- Ein beendeter Prozess belegt immer noch einen Slot in der Prozesstabelle des Kernels.
- Dort verwahrt der Kernel den Exitwert auf, bis jemand danach gefragt hat.
- Ein beendeter Prozess, von dem niemand den Exitwert wissen will, heißt "Zombie" oder ist "verwaist" oder "defunct".

Experiment

- ```
  docker run -ti --name=nn --rm python:3.8-buster \
  sh -c 'exec sleep 600'
  ```
- ```
  docker exec -ti nn \
    sh -c 'sleep 2 & ps fax'
  docker exec -ti nn ps fax
  ps ax | grep defunct
  ```

Zombieprozesse im Container verbrauchen Prozesstabellenslots auf dem Host!

Inhalt

- Linux-Bootprozess und init ✔
- Signale ✔
- Signale: init ist anders ✔
- Prozesse: system vs. Exec ✔
- Prozesse: Returnwert ✔
- Verwaiste Prozesse ✔
- **Init: Das Waisenhaus**

Init ist das Waisenhaus

- Zombieprozesse werden vom Kernel an `init` gemeldet.
- `Init` kann und soll den Exitwert verarbeiten.
- Wenn `init` das nicht tut, bleiben die Zombies.
- Im Extremfall können im Container (oder gar auf dem Host) keine neuen Prozesse gestartet werden. Was u.a. bedeutet, dass kein Shellskript mehr läuft.
- Dieser Zustand ist unangenehm.
  Das möchte man nicht.

Wir sind durch!

- Linux-Bootprozess und init ✔
- Signale ✔
- Signale: init ist anders ✔
- Prozesse: system vs. Exec ✔
- Prozesse: Returnwert ✔
- Verwaiste Prozesse ✔
- Init: Das Waisenhaus ✔

Shameless plug

https://github.com/aknrdureegaesr/yasinit

"Yet Another Simple Init"

Python3

Designentscheidungen yasinit

- Implementierung in Python.
- Ein oder mehrere Programme können gestartet werden.
  - Ein Programm via Kommandozeile,
  - mehrere via Configdatei.
- Ein Programm scheitert: Aufgeben Restart Job von Kubernetes, `systemd`, ...
- Log auf Stderr.

Fragen? Bemerkungen?