Shebang – All der Kram

von Markus Schnalke

Dieser Artikel ist nicht nur eine Einführung in den Shebang-Mechanismus unter Unix. Er soll neben Beispielen und Empfehlungen für die Nutzung auch Hintergrundinformationen und Verständnis der internen Abläufe vermitteln.

Was ist Shebang?

So gut wie jeder, der schon mal ein Shell-Skript geschrieben hat, hat dabei auch eine Shebang-Zeile verwendet. Beispielsweise so:

```
#!/bin/sh
# Dieses Skript gibt das
# Datum aus
date
```

Die erste Zeile ist entscheidend, also die, die mit **#!** beginnt und dann den Pfad zum Interpreter aufführt. In diesem Fall wird das Skript von **/bin/sh** interpretiert.

Name und Namensherkunft

Was genau den Namen „Shebang“ trägt, ist nicht so recht definiert. Die erste Zeile in Skripten wird so genannt, aber auch der ganze Mechanismus oder auch nur die zwei Zeichen **#!**.

Auch die Herkunft des Namens scheint nicht ganz klar zu sein, zumindest kann man an verschiedenen Stellen Unterschiedliches lesen. Im amerikanischen Slang bedeutet „Shebang“ so etwas wie „Kram“ oder „Zeug“. Davon könnte das Wort übernommen worden sein. „Hash-Bang“ (die Namen der Zeichen „#“ und „!“ im Computerjargon) oder „Shell-Bang“ können aber ebenso zu „Shebang“ geführt haben [1].

Vermutlich ist diese Schwammigkeit daran schuld, dass die Shebang-Funktionalität zwar recht bekannt ist, ihr Name aber kaum.

Funktionsweise aus Nutzersicht

Dem Anwender bringt Shebang den Vorteil, dass es ihn nicht kümmern muss, in welcher Programmiersprache ein Programm geschrieben wurde. Auch wenn das Programm in einer anderen Sprache reimplementiert wird, bekommt er davon nichts mit. Dank Shebang kann er Skripte genau so starten, wie kompilierte Programme, nämlich direkt mit ihrem Namen.

Gäbe es kein Shebang, so müsste er je nach verwendeter Programmiersprache Unterschiedliches eingeben. Beispielsweise **sh skript** oder **perl skript**. Oder falls es ein kompiliertes Programm ist, eben nur **skript**.

Für den Skriptentwickler ist Shebang ebenfalls eine Erleichterung, weil es nur einer Zeile am Skriptbeginn bedarf, um den Interpreter festzulegen. Danach muss er sich nicht mehr darum kümmern, es funktioniert einfach.

Allerdings gilt das nur, solange die Shebang-Zeilen einfach sind. Will man etwas komplizierte Dinge tun, dann stößt man schnell an die Grenzen des Mechanismus. Glücklicherweise kommt das selten vor. Und wenn, dann kann der Programmierer ja auch wieder auf seinen herkömmlichen Kommandoaufruf zurückgreifen und Shebang einfach außen vor lassen.

Shebang für den Shell-Skript-Schreiber ist vergleichbar mit Makefiles für einen C-Programmierer: Es ist meist komfortabler mit, aber manchmal will man doch lieber ohne.

Die Betriebssystemsicht

Es ist immer gut, wenn man weiß, wie eine Funktion intern arbeitet. Wie also – oder besser wo – die Shebang-Zeile interpretiert wird.

Diese Aufgabe übernimmt der Kernel. In ihm muss die Unterstützung für Shebang eingebaut sein. Alle aktuellen Unix-Kernel können mit Shebang-Skripten umgehen. Das liegt vor allem daran, dass es Shebang schon ziemlich lange gibt. Eingeführt wurde es von Dennis Ritchie in den 80er-Jahren für Version 8 Unix [2].

Da Skripte mit Shebang so aufgerufen werden, als wären es kompilierte Programme, werden sie von der Shell auch ebenso behandelt. Das heißt, sie kümmert sich gar nicht darum, ob es sich um ein Skript mit Shebang oder um ein kompiliertes Programm oder um irgendetwas anderes handelt, sie übergibt es nur dem Kernel, damit dieser es ausführt.

Der Kernel muss nun entscheiden, wie er den Aufruf behandelt. Dazu liest er die ersten paar By-

tes der Datei ein; dort befinden sich sogenannte „Magic Numbers“ [3] die den Dateiinhalt charakterisieren. Findet er dort „\177ELF“, dann ist es ein für GNU/Linux kompiliertes Programm, das er direkt ausführen kann. **#!** zeigt dagegen ein Shebang-Skript an. Dateien, die er nicht erkennt, lehnt er mit einem Fehler ab.

Bei Shebang-Skripten wird nun der eigentliche Programmaufruf durch den Inhalt der Shebang-Zeile ersetzt, wobei noch der Dateiname angehängt wird. Für diesen neuen Befehl entscheidet der Kernel erneut, ob und wie er ihn verarbeitet.

Für das anfangs gezeigte Beispielskript läuft es folgendermaßen ab:

1. Nach dem Aufruf von **./beispielskript.sh** fordert die Shell die Ausführung dieses Befehls beim Kernel an.
2. Der Kernel schaut sich die ersten Bytes der Datei an und findet dort **#!**. Somit generiert er eine neue Ausführungsanforderung an sich selbst. Er nimmt das erste Wort (nach **#!**) als Befehl und den Rest der Zeile als erstes Argument. Den Dateinamen und die Kommandozeilenargumente hängt er hinten an.

Der neue Aufruf ist somit

```
/bin/sh ./beispielskript.sh
```

Interessant wird die Situation aber, wenn ein Shebang-Skript ein weiteres aufruft – oder gar sich selbst (rekursive Ausführung). In den meisten Unix-Kerneln, so auch bei Linux bis Version 2.6.27.8, wurde die Ausführung von Shebang-Skripten, die von Shebang-Skripten aufgerufen wurden, verweigert. Seit Linux-2.6.27.9 geht auch ein mehrstufiger Aufruf.

Es lohnt sich diesbezüglich unbedingt einmal in den Kernelquellcode zu spicken, so kompliziert ist das nämlich nicht. Die relevante Datei im Linux-Quellcode ist **fs/binfmt_script.c** [4]. Der Commit, der mehrstufige Aufrufe in Linux eingeführt hat, lohnt auch einen Blick [5]. Bei NetBSD findet man die Shebang- Behandlung in **sys/kern/exec_script.c** [6].

Stolpersteine

Shebang ist nicht sehr kompliziert, doch es gibt Dinge zu beachten, jedenfalls wenn das Skript portabel sein soll.

Den Interpreter-Pfad zum Beispiel. Dieser sollte absolut sein. Manche Kernel können zwar mit relativen Pfaden umgehen, aber nicht alle.

Absolute Pfade sind kein Problem solange sie **/bin/sh** lauten, denn eine Bourne-kompatible Shell ist auf allen Unix-Systemen an diesem Ort zu finden. Schreibt man aber Python- oder Perl-Skripte, so hat man das Problem, dass diese Interpreter nicht auf allen Systemen am gleichen Ort installiert sind. Oft wird **/usr/bin/python** zwar zutreffen, aber manchmal ist es halt auch **/usr/local/bin/python** oder etwas ganz anderes.

Um dieses Problem zu lösen, kann man sich des Programms **env** bedienen. Unter der Annahme, dass env auf sehr vielen Systemen am gleichen Ort (**/usr/bin/env**) liegt, überlässt man es ihm, den eigentlichen Interpreter aufzurufen. Zum Beispiel wie in folgendem Skript:

```
#!/usr/bin/env python
print "foo"
```

Leider hilft auch env nicht immer. AWK-Skripte zum Beispiel kann man damit nicht aufrufen. Ein Shebang-Skript für AWK sieht so aus:

```
#!/usr/bin/awk -f
BEGIN { print "foo" }
```

Nach der Auswertung des Shebang führt der Kernel **/usr/bin/awk -f ./foo.awk** aus. Das Argument **-f** sorgt dafür, dass AWK das nachfolgende Argument als Skriptdatei behandelt und nicht als Liste von AWK-Befehlen.

Leider werden von den meisten Kerneln alle in der Shebang- Zeile übergebenen Parameter zu einem zusammengefasst. (Die einzige bedeutende Ausnahme ist FreeBSD.) Folgendes ist also nicht möglich:

```
#!/usr/bin/awk -F: -f
{ print $1 }
```

Denn der Kernel würde **/usr/bin/awk "-F: -f" firstfield.awk** starten. Doch mit der einen Option **-F: -f** kann AWK halt nichts anfangen.

Die Bedeutung der Option **-F:** ist hierbei unwichtig, entscheidend ist, dass ein weiterer Komman-

dozeilenparameter vorhanden ist. Wer sich dennoch für die Bedeutung interessiert, der möge in der Manpage von AWK nachlesen.

Dies trifft auch die Verwendung von env mit awk. Die Shebang-Zeile wäre **#!/usr/bin/env awk -f** und der Kernel würde **/usr/bin/env "awk -f" skript.awk** ausführen. Das Ergebnis ist die Fehlermeldung **/usr/bin/env: awk -f: No such file or directory**. Das am Ende dieses Artikels vorgestellte Programm printargs kann dieses Verhalten wunderbar veranschaulichen.

Die meisten Systeme verbieten bei Skripten das Setzen des setuid- und setgid-Bits. Auf diese Weise wird Sicherheitslöchern vorgebeugt, denn sie entstehen bei solchen Skripten einfach viel zu schnell.

Dann wäre noch die Längenbegrenzung der Shebang-Zeile zu erwähnen. Früher waren das mal 14 Zeichen, doch diese Zeiten sind lange vorbei. Heutzutage dürfte man, da man ja sowieso nur einen Parameter übergeben kann, kaum jemals an das Limit stoßen.

Eigene Skripte

Nach dem bisherigen Text und seinen Beispielen sollte der Leser nun in der Lage sein, sein eigenes Shell-Skript zu erstellen. Hier soll dieser Vorgang jedoch noch einmal exemplarisch gezeigt werden.

Man erstellt im Editor eine Textdatei **foo.sh** mit folgendem Inhalt:

```
#!/bin/sh
whoami
echo "http://freiesmagazin.de"
date
```

Diese macht man mit

```
$ chmod +x foo.sh
```

ausführbar. Nun lässt sich das Skript mit

```
$ ./foo.sh
```

starten und führt zu einer Ausgabe ähnlich der folgenden:

```
meillo
http://freiesmagazin.de
Di 15. Sep 10:49:18 CEST 2009
```

Die häufigsten Shebang-Skripte dürften für die Shell sein, gerade bei diesen ist leider eine Unart sehr verbreitet. Es geht um Bashism [7]. Das bedeutet, dass Bash-spezifische Funktionen in ein Skript für **/bin/sh** enthalten sind. Bei GNU/Linux-Systemen wirkt sich das oft nicht aus, weil **/bin/sh** ein Symlink auf **/bin/bash** ist und damit das Skript doch von der Bash ausgeführt wird. Das ist aber nicht überall so. POSIX verlangt nur, dass **/bin/sh** Bourne-Shell-kompatibel ist. Es kann sich somit auch zum Beispiel dash oder die ksh dahinter verbergen. Diese Shells können aber nichts anfangen mit Konstrukten wie:

```
while (( ++i < 5 )) ; do
  ...
done
```

Deshalb: Wer Bash-spezifische Funktionen in ein Skript einbaut, der möge in der Shebang-Zeile auch **#!/bin/bash** oder **#!/usr/bin/env bash** schreiben.

Der Leser sollte nicht glauben, dass nur „exotische“ Systeme **/bin/sh** nicht auf die Bash verlinken. Ubuntu ist ein gutes Gegenbeispiel. In neueren Version zeigt dort **/bin/sh** nämlich auf **dash** (mit „d“!). Auch andere GNU/Linux-Distributionen sind dabei, diese Umstellung zu vollziehen.

Aufgaben

Zum Schluss noch zwei Verständnisfragen, um das vermittelte Wissen zu festigen und um zum eigenen Nachforschen anzuregen.

1. Was ist der Unterschied zwischen den folgenden zwei Skripten?

```
#!/bin/sh
echo $0 $@
```

und

```
#!/bin/echo
```

Wann verhalten sich die Skripte nicht gleich?

2. Wie sehen die einzelnen Verarbeitungsschritte bei einem zweifach indirekten Shebang-Aufruf aus? Das Szenario lässt sich so generieren:

```
$ cd /tmp
$ echo '#!/tmp/b' >a
$ echo '#!/tmp/c' >b
$ cp printargs c
$ chmod +x a b
```

Was passiert beim Aufruf von **./a 1 2 3** in einem Kernel, der die Verschachtelung unterstützt (zum Beispiel Linux >= 2.6.27.9)? Was bei anderen? Was wird letztendlich wirklich ausgeführt? Und welche Ausgabe bekommt man?

Das Hilfsprogramm **printargs** gibt einfach seine Argumente aus und ist damit sehr nützlich um Shebang-Skripte zu testen. Sein Quellcode ist trivial und wurde von Sven Maschecks umfassender Seite zu Shebang [8] übernommen:

```
#include <stdio.h>

int
main (int argc, char* argv[])
{
  int i;
  for (i = 0; i < argc; i++) {
    printf(%d: %s\n", i, argv[i]);
  }
  return 0;
}
```

Listing 1: printargs.gcc

Der Quellcode muss abgespeichert und mit einem C- Compiler übersetzt werden:

```
$ gcc -o printargs printargs.gcc
```

Redaktioneller Hinweis: *Die Antworten und Kommentare zum Artikel können mit dem Betreff „shebang“ an die Redaktion unter* redaktion@freiesMagazin.de *gesendet werden. Wir leiten die E-Mails an den Autor weiter.*

LINKS

[1] http://de.wikipedia.org/wiki/Shebang

[2] http://www.in-ulm.de/~mascheck/various/shebang/sys1.c.html

[3] http://de.wikipedia.org/wiki/Magische_Zahl_(Informatik)

[4] http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.27.y.git;a=blob;f=fs/binfmt_script.c

[5] http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.27.y.git;a=commitdiff;h=09932109

[6] http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/kern/exec_script.c?rev=1.63

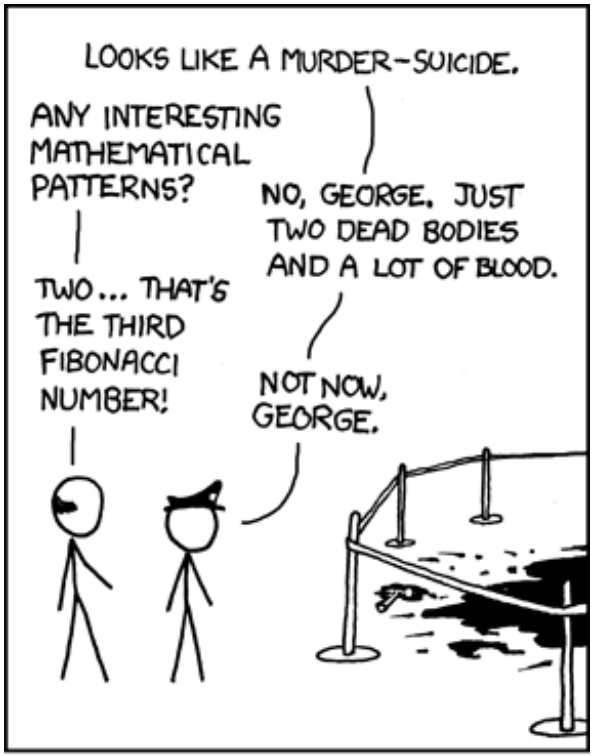
[7] http://wiki.ubuntu.com/DashAsBinSh

[8] http://www.in-ulm.de/~mascheck/various/shebang.html

Autoreninformation

Markus Schnalke liebt Unix und dessen Skripting-Möglichkeiten. So auch den Shebang-Mechanismus, der die Ausführung von Skripten deutlich erleichtert.

Diesen Artikel kommentieren



„Crime Scene“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/587