SelfLinux-0.13.1

# Prozessverwaltung

Autor: Ferdinand Hahmann (*FerdinandHahmann@gmx.net*)
Formatierung: Matthias Hagedorn (*matthias.hagedorn@selflinux.org*)
Lizenz: GPL

# Inhaltsverzeichnis

**1 Einleitung**

**2 ps**

**3 pstree**

**4 top**

**5 tload**

**6 fuser**

**7 kill**

**8 killall**

**9 nice**

# 1 Einleitung

Linux ist ein Multitasking-Betriebssystem. Das heißt, es können mehrere Programme zur selben Zeit ablaufen. Üblicherweise benutzt jeder Anwender ein Programm im **Vordergrund**, während mögliche weitere im **Hintergrund** ablaufen. Es ist ein Mechanismus vorhanden, um Hintergrundprozesse zu beeinflussen, sie beispielsweise anzuhalten, fortzusetzen oder zu beenden. Ebenso unterstützen die meisten Shells Job-Control, also die Möglichkeit, Vordergrundprozesse in den Hintergrund zu schicken und umgekehrt.

# 2 ps

In einem Multitasking-Betriebssystem wie Linux läuft zu jedem Zeitpunkt eine Vielzahl von Prozessen, über die es interessante Dinge herauszufinden gibt. Dazu gibt es das Kommando ps:

```
user@linux / $ ps

  PID TTY      TIME CMD
18301 pts/1    0:00 bash
18901 pts/1    0:00 ps
```

Die Ausgabe bedeutet, dass unser Pinguin momentan zwei Prozesse betreibt: Eine Shell und ps selbst.

Die einzelnen Spalten bedeuten folgendes:

* Die **PID** (Prozess ID) ist eine Nummer, die jeden Prozess eindeutig identifiziert.
* Unter **TTY** wird der Name des Terminals angegeben, das den Prozess kontrolliert. Meist bedeutet dieser Name, dass der Prozess von diesem Terminal aus gestartet worden ist, und für die Ein- und Ausgabe benutzt wird.
* **TIME** zeigt die von jedem dieser Prozesse bisher genutzte Rechenzeit an.
* **CMD** (oder COMMAND) zeigt den Befehl an, mit dem der Prozess gestartet worden ist.

ps hat eine Unmenge von Optionen, die alle dazu dienen, die Berge von Daten, die sich über Prozesse gewinnen lassen, zu filtern. Die einfachste Möglichkeit der Auflistung aller laufenden Prozesse ist ps -ax. Es werden alle Prozesse aufgelistet, auch diejenigen, die nicht vom aktuellen Benutzer gestartet wurden und die, die nicht mit einem Terminal verbunden sind (also im Hintergrund laufen). Diese Liste ist normalerweise sehr lang, daher sind nur die ersten paar Zeilen wiedergegeben.

```
user@linux / $ ps -ax

PID     TTY             STATE   TIME    COMMAND
1       ?               S       0:04    init [2]
2       ?               SW      0:00    [keventd]
3       ?               SWN     0:00    [ksoftirqd_CPU0]
...
```

Bei ps -ax wird auch noch der **Status** (STATE) des jeweiligen Prozesses ausgegeben. Dabei werden folgen Abkürzungen verwendet:

* **D** (Deep sleeping)
  Der Prozess befindet sich in einem ununterbrechbaren Schlaf.
* **R** (Running)
  Der Prozess verarbeitet im Moment gerade Daten.
* **S** (Sleeping)
  Der Prozess wartet auf irgendein Ereignis um Daten zu verarbeiten.
* **T** (Traced)
  Der Prozess wurde angehalten.
* **Z** (Zombie)
  Zombies sind Prozesse, die eigentlich schon beendet sein sollten, aber immer noch auf irgendetwas warten.

Sie können immer mal wieder vorkommen und stellen keinen Grund zur Besorgnis dar, solange sie nicht in Mengen auftreten und es sich nicht um viele Zombies mit gleichem Namen handelt.

Stehen neben diesen Statusmeldungen noch ein oder mehrere Zeichen bedeuten diese folgendes:

* **W**
  bedeutet, dass der Prozess in den Swap-Speicher ausgelagert wurde und somit keine Speicherseiten belegt werden.
* **<**
  bedeutet, dass der Prozess mit einer höheren Priorität läuft.
* **N**
  bedeutet genau das Gegenteil. Der Prozess läuft mit einer niedrigeren Priorität.
* **L** (Locked)
  bedeutet, dass der Prozess im Speicher geladen wurde und auch dort gehalten wird.

# 3 pstree

Linux merkt sich, wenn ein Programm ein anderes startet. Wenn der **Elternprozess** beendet wird, wird auch für das Ende aller **Kindprozesse** gesorgt, damit diese nicht für immer herumliegen und Platz wegnehmen. Die **Baumstruktur** der laufenden Prozesse wird mit `pstree` oder mit ps und der Option `-f` angezeigt:

```
user@linux / $ pstree

init-+-atd
     |-automount
     |-cron
     |-httpd---httpd
     |-inetd
     |-kbgndwm
     |-kblankscrn.kss
     |-kdm-+-X
     |     `-kdm---kwm
     |-kflushd

     |-kfm-+-Eterm---bash---pstree
     |     |-netscape---netscape
     |     `-xemacs
...
```

Hier ist ein umfangreicher Ast. Man sieht, wie der Autor an diesem Text mit XEmacs unter KDE schreibt. Gleichzeitig laufen noch ein Terminalfenster (Eterm), in dem gerade `pstree` ausprobiert wird und `netscape`.

# 4 top

Der Systemverwalter kann seine Augen und Ohren nicht überall haben, darum braucht er ein Programm, das ihm die wichtigsten Prozesse anzeigt, die auf dem System laufen. Die wichtigsten sind die, die am meisten Rechenzeit oder am meisten Speicher belegen. Nach diesen Kriterien entscheidet `top`.

```
fboerner@gentoo:~
top - 14:14:21 up 1 day, 23:24,  5 users,  load average: 0.62, 0.60, 0.55
Tasks: 111 total,   4 running, 107 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.7% us, 42.9% sy, 29.7% ni, 25.7% id,  0.0% wa,  0.0% hi,  1.0% si
Mem:   1035632k total,  1015124k used,    20508k free,   188636k buffers
Swap:  1999832k total,     1704k used,  1998128k free,   256280k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
28134 fboerner  35  19 56332  24m  48m R 72.8  2.4 105:49.49 kfiresaver.kss
 1999 root      16   0  154m  24m 138m R  1.0  2.4  19:32.86 X
29915 fboerner  16   0  1972 1092 1752 R  0.3  0.1   0:00.21 top
    1 root      16   0  1396  512 1236 S  0.0  0.0   0:00.79 init
    2 root      34  19     0    0    0 S  0.0  0.0   0:00.12 ksoftirqd/0
    3 root       5 -10     0    0    0 S  0.0  0.0   0:00.07 events/0
    4 root       6 -10     0    0    0 S  0.0  0.0   0:00.00 khelper
    5 root      15 -10     0    0    0 S  0.0  0.0   0:00.00 kacpid
   33 root       5 -10     0    0    0 S  0.0  0.0   0:02.55 kblockd/0
   34 root      15   0     0    0    0 S  0.0  0.0   0:00.00 khubd
   44 root      15   0     0    0    0 S  0.0  0.0   0:02.40 pdflush
   45 root      15   0     0    0    0 S  0.0  0.0   0:05.66 pdflush
   47 root      15 -10     0    0    0 S  0.0  0.0   0:00.00 aio/0
   46 root      15   0     0    0    0 S  0.0  0.0   0:04.72 kswapd0
   51 root      25   0     0    0    0 S  0.0  0.0   0:00.00 kseriod
  160 root      15   0     0    0    0 S  0.0  0.0   0:11.28 kjournald
  328 root      16   0  1760  924 1448 S  0.0  0.1   0:00.10 devfsd
```

top

`top` fasst alle bisher genannten Kommandos zusammen und fügt noch einiges hinzu.

* Ganz oben stehen die von `uptime` bekannten Werte (Systemzeit, Uptime, angemeldete Benutzer und Auslastung).
* In der nächsten Zeile kommt die Anzahl der laufenden Prozesse und ihre Zustände. Die Zustände wurden bereits bei ps erklärt.
* Die dritte Zeile zeigt den prozentualen Anteil an, womit sich der (oder die) Prozessor(en) in den letzten Sekunden beschäftigt hat (haben).
* **user:**
  Mit user sind Prozesse gemeint, die von den Benutzern des Systems gestartet worden sind.
* **system:**
  Hierbei sind sämtliche Prozesse gemeint, die vom System gestartet worden sind.
* **nice:**
  Damit sind all die Prozesse gemeint, die nicht mit einem nice von 0 (null) laufen.
* **idle:**
  Mit idle wird die noch zur Verfügung stehende Prozessorlast bezeichnet.
* Nun kommen noch die von free bekannten Speicherangaben.

Als letztes folgt die Liste mit den ressourcenträchtigesten Prozessen. Im Gegensatz zu `ps` werden bei `top` weitere Informationen zu den einzelnen Prozessen ausgegeben:

* **User**
  gibt den Namen des Users an, unter welchem der Prozess läuft.
* **PR**
  gibt die Priorität des Prozesses an
* **NI**
  gibt den Nice-Wert an. Ein negativer Wert bedeutet eine höhere Priorität, ein positiver Wert eine geringere Priorität.
* **VIRT**
  gibt die gesamte Anzahl an Speicher an, den dieser Prozess benötigt, inklusive den Code, den Daten und den geteilten Bibliotheken.

* **RES**
  gibt die Belegung des physikalischen Arbeitsspeichers an. Auslagerungen im Swap-Speicher werden hier nicht berücksichtigt.
* **SHR**
  gibt die Anzahl des Speichers an Daten an, die auch von anderen Prozessen genutzt werden können.
* **S**
  gibt den Status des Prozesses an. Siehe dazu auch die Erklärung bei ps.
* **%CPU**
  gibt an wieviel Prozent von der gesamten Prozessorleistung der Prozess benötigt.
* **%MEM**
  gibt dasselbe wie **%CPU** an, diesmal aber bezogen auf den physikalischen Arbeitsspeicher.

# 5 tload

Die Aufgabe von `tload` ist es, die durchschnittliche Systemlast in einer einfachen ASCII-Grafik darzustellen. Mit der Option `-d` kann die Zeit in Sekunden angeben werden, nach der die Grafik aktualisiert wird. Mit `-s` wird die Skalierung angegeben, also wie viele Zeilen für eine Einheit verwendet werden. Je größer die Zahl ist, desto gröber die Darstellung.

# 6 fuser

Manchmal kann es ganz nützlich sein, festzustellen welche Prozesse eine bestimmte Datei benutzen. Für diese Aufgabe gibt es den Befehl `fuser` gefolgt vom Dateinamen.

```
user@linux / $ fuser ksycoca

ksycoca:        463 463m 471 471m 539 539m 546 546m
```

Diese Ausgabe sagt aus, dass die Datei ksycoca von den Prozessen mit der **PID** 463, 471, 539 und 546 benutzt wird. Gefolgt von der **PID** wird ein Buchstabe mit ausgegeben (in diesem Fall ist es immer ein **m**). Dieser Buchstabe gibt an, wie der Prozess auf die Datei zugreift. Konkret bedeuten die Buchstaben folgendes:

* **c:**
  Die Datei wird vom Prozess als Verzeichnis behandelt.
* **e:**
  Die Datei ist ausführbar und wird vom Prozess ausgeführt.
* **f:**
  Die Datei wurde vom Prozess geöffnet. Im Standard-Modus wird das **f** weg gelassen. Somit ist auch klar, warum in unserem Beispiel jede **PID** zweimal angegeben wurde.
* **m:**
  Die Datei ist eine Bibliothek, die gemeinsam von den Prozessen genutzt wird.
* **r:**
  wird verwendet, wenn es sich um das Wurzelverzeichnis (root-Verzeichnis) handelt.

Auch für `fuser` gibt es noch eine Reihe weiterer Optionen. Interessant dabei ist die Option `-n`. Diese erwartet noch eine weitere Angabe:

* **file**,
  wenn es sich um eine Datei handelt. Dabei bewirkt `fuser -n file ksycoca` dasselbe wie `fuser`

ksycoca
* **udp**,
  wenn sich die darauf folgende Angabe um einen UDP-Port handelt.
* **tcp**,
  bedeutet das gleiche wie udp mit dem Unterschied, dass ein TCP-Port gemeint ist.

Der oben beschriebe Buchstaben-Code ist auch für die Port-Angaben gültig, da unter Linux Ports auch als Dateien behandelt werden.

Weitere wichtige Optionen sind noch:

* -a:
  Dabei werden auch Dateien ausgegeben, auf die gerade kein Prozess zugreift.
* -u:
  Gibt in Klammern den zur PID gehörenden Benutzernamen aus.
* -v (verbose)
  gibt mehr Informationen aus.

# 7 kill

kill ist - trotz seines Namens - nicht nur zum Beenden von Prozessen geeignet. Es kann alle möglichen Signale an die Prozesse senden, die vom Benutzer gestartet wurden, der kill aufruft. Ist dieser Benutzer der Systemverwalter, dann sind ihm alle Prozesse zugänglich.

Der Aufruf ist:

```
kill <Signal> <Prozess-ID>
```

Die Prozess-ID (PID) können Sie beispielsweise aus der Ausgabe des Programmes ps entnehmen:

```
user@linux / $ ps

[...]
19376 pts/0    00:00:00 ps
```

Hier ist die PID des ps-Prozesses selbst die 19376.

Wenn die PID negativ ist, wird **<Signal>** nicht an einen bestimmten Prozess gesendet, sondern an alle Prozesse mit der angegebenen PGID. Der Befehl lautet also dann folgendermaßen:

```
kill <Signal> -<PGID>
```

Anstelle der Prozess-ID kann auch -1 angegeben werden. Dabei wird **<Signal>** an alle Prozesse gesendet, ausser dem kill-Prozess selbst und init. Zudem zeigt dieser Befehl nur bei den Prozessen Wirkung, bei welchen man die nötigen Rechte hat.

Wenn Sie kein Signal angeben, sendet kill das **Signal 15** (**SIGTERM**), das die meisten Programme dazu veranlasst, hinter sich aufzuräumen, um sich dann zu beenden.

Weitere wichtige Signale sind **SIGHUP** (**1**) und **SIGKILL** (**9**). **SIGHUP** steht für **hang-up**. Wenn ein Benutzer sich über das Netzwerk angemeldet hat und die Verbindung abbricht, z. B. weil ein Modem auflegt (eben ein

**hang-up**), wird dieses Signal an alle Prozesse gesendet, die während dieser Anmeldesitzung gestartet wurden. Sie werden dann beendet.

**SIGKILL** tut genau das, was sein Name andeutet: Es unternimmt alles, um einem Prozess den Garaus zu machen.

```
user@linux / $ kill -SIGKILL 12345
```

Eine Liste mit allen Signalen finden Sie entweder in der Manpage von `kill` (`man kill`) oder mit dem Befehl `kill -l`.

Die Option `-p` sendet kein Signal an den Prozess, sondern gibt nur den Namen des zur PID passenden Prozesses aus.

Die bash-Shell enthält einen eingebauten Befehl `kill`, der im Prinzip dieselbe Auswirkung wie `/bin/kill` hat, jedoch zwei Vorteile bietet:

Er erlaubt die Verwendung von Job-IDs statt PIDs, und wenn Sie die maximale Anzahl von Prozessen gestartet haben, die Ihr Systemverwalter ihnen zubilligt, können Sie einen davon beenden, ohne dazu einen weiteren Prozess mit `/bin/kill` starten zu müssen. Die Eingabe von `kill` startet den in die bash eingebauten `kill`-Befehl. `/bin/kill` verwendet den externen `kill`-Befehl.

# 8 killall

`killall` bewirkt im Prinzip dasselbe wie kill. Es sendet Signale an Prozesse. Der Unterschied zwischen `kill` und `killall` ist, an welchen Prozesse ein Signal gesendet wird. Bei `killall` wird nicht die PID des Prozesses angegeben, sondern dessen Name. Da allerdings der Name, im Gegensatz zur PID, nicht eindeutig ist, wird das Signal an alle Prozesse mit diesem Namen gesendet. Der genaue Aufruf lautet:

```
killall [Option] [Signal] <Name>
```

Wenn **<Name>** einen / enthält so ist damit nicht der Name eines Prozesses gemeint, sondern der Name einer auszuführenden Datei. Dabei wird ein Signal an alle Prozesse gesendet, die diese Datei ausführen.

Die Signale sind dieselben wie bei kill. Ohne Angabe von Signal wird auch ein **SIGTERM** (**15**) gesendet.

Eine wichtige Option ist `-e` (`--exact`). Normalerweise wertet `killall` nur die ersten 15 Zeichen von **<Name>** aus. Haben nun zwei Prozesse unterschiedliche Namen stimmen aber trotzdem in den ersten 15 Zeichen überein, so sind dennoch beide Prozesse von `killall` betroffen. Die Option `-e` erzwingt dass der volle Name ausgewertet wird.

Eventuell will man nicht blind alle Prozesse mit gleichen Namen beenden, sondern vorher noch eine Nachfrage haben. Dafür gibt es die Option `-i` (`--interactive`).

# 9 nice

Unter Linux ist es möglich, dass mehrere Befehle, auch Jobs genannt, zur gleichen Zeit ausgeführt werden. Da allerdings nicht alle Ressourcen unbegrenzt zur Verfügung stehen, muss eine Auswahl getroffen werden, welche Priorität ein Job hat. Diese Aufgabe erledigt der Befehl `nice`:

```
nice <Priorität> <Befehl> [Argumente]
```

**<Priorität>** ist für normale User eine Zahl zwischen 0 und 19, wobei 19 die niedrigste Priorität ist. Der Superuser hat die Möglichkeit Prioritäten von -20 bis 19 zu vergeben. Dabei ist -20 die höchste Priorität.

**<Befehl>** ist der eigentliche Befehl, der ausgeführt werden soll. **[Argumente]** sind dabei Optionen, die an **<Befehl>** übergeben werden.