

# Prozessverwaltung unter Linux



17. Juli 2002

Roman Käppeler  
Universität Ulm

- Linux Proseminar Sommersemester 2002 -

## Inhaltsverzeichnis

<b>0</b>	<b>Einleitung</b> .....	<b>3</b>
<b>1</b>	<b>Begriffe aus der Prozessverwaltung</b> .....	<b>3</b>
<b>2</b>	<b>Prozesse</b> .....	<b>4</b>
	2.1 Organisation von Prozessen .....	4
	2.2 Aufbau eines Prozesstabelleneintrages.....	4
	2.3 Lebenszyklus von Prozessen .....	5
<b>3</b>	<b>Der Scheduler</b> .....	<b>6</b>
	3.1 Scheduling-Strategien.....	7
<b>4</b>	<b>Prozesserzeugung</b> .....	<b>7</b>
	4.1 Anlegen neuer Prozesse .....	7
	4.2 Prozesshierarchie .....	8
	4.3 Daemon-Prozesse .....	8
<b>5</b>	<b>Signale an Prozesse</b> .....	<b>9</b>
<b>6</b>	<b>Informationen über Prozesse</b> .....	<b>10</b>
<b>7</b>	<b>Befehle zur Prozessverwaltung</b> .....	<b>11</b>
	7.1 Prozessstatistik mit <code>ps</code> .....	11
	7.2 Laufende Übersicht mit <code>top</code> .....	11
	7.3 Prozesshierarchie mit <code>ps tree</code> .....	12
	7.4 Prozesse beenden mittels <code>kill</code> .....	12
	7.5 Benutzerdefinierte Priorität mit <code>nice</code> .....	13
	7.6 Prozesse unabhängig von der Shell mit <code>nohup</code> und <code>&amp;</code> .....	13

## 0 Einleitung

Auf einem Prozessor kann zu einem bestimmten Zeitpunkt immer nur ein Programm abgearbeitet werden. Die Aufgaben, die ein Betriebssystem zu bewältigen hat, fordern aber in den meisten Fällen, dass gewisse Aufgaben parallel ausgeführt werden. Dazu wären mehrere Rechner oder zumindest mehrere Prozessoren erforderlich. Dass aber zumindest der Eindruck entsteht, dass die Programme quasi gleichzeitig ausgeführt werden, wird jedem Programm der Reihe nach jeweils ein wenig Rechenzeit (Bruchteile von Sekunden) zugeordnet. Das Betriebssystem arbeitet also im sogenannten *Multiplexbetrieb*. Damit läuft jedes Programm ein wenig langsamer, aber immer noch ohne spürbare Unterbrechungen.

Um das wichtige Konzept der Parallelität umsetzen zu können, müssen vom Betriebssystem gewisse Vorkehrungen getroffen werden. Damit, mit unterschiedlichen Strategien, ein quasi nebenläufiges Ausführen von Programmen auf einem Prozessor erreicht wird, muss ein gerade ablaufendes Programm beliebig angehalten und wieder fortgeführt werden können. Dabei ist es sehr wichtig, dass beim Anhalten eines Programms seine Umgebung gesichert wird und diese beim Fortsetzen auch wieder hergestellt werden kann. Verantwortlich für die Verwaltung dieser Mechanismen zum Anhalten, Sichern und Fortsetzen der (*Prozess-*) *Umgebung* ist der (*Prozess-*) *Scheduler*.

Bevor aber der Prozess-Scheduler mit seinen Scheduling-Strategien im Detail beschrieben wird, müssen noch einige allgemeine Begriffe aus der Prozessverwaltung eingeführt werden. Insbesondere muss zwischen den Begriffen *Prozess* und *Programm* unterschieden werden.

## 1 Begriffe aus der Prozessverwaltung

Die Begriffe *Prozess* und *Programm* werden oft gleichgesetzt, was aber nicht korrekt ist.

In [1] wird der Begriff *Programm* als "eine zur automatischen Verarbeitung in einem Rechensystem geeignete Darstellung eines Algorithmus, d. h. eine Vorschrift zur Lösung einer Aufgabe" definiert, also eine Folge von Befehlen für den Prozessor, die dann typischerweise in Form einer Datei auf der Festplatte liegt. Um ein solches Programm ausführen zu können benötigt man ein "*aktives Organ*", sowie eine *Umgebung* in der das Programm ausgeführt werden kann. Das aktive Organ ist natürlich der Prozessor, der die binären Befehle eines Programms verarbeiten kann. „Die Umgebung ist die Menge der Dinge, die bei der Ausführung manipuliert werden“. Dazu gehören z. B. die Register des Prozessors, die Daten, die sich während der Ausführung im Hauptspeicher befinden, Daten im allgemeinen Sinn (z. B. der Drucker und geöffnete Dateien) und das Programm selbst [1].

Ein *Prozess* (*Task*<sup>1</sup>) hingegen ist "ein in der Ausführung befindliches Programm mit seiner Ausführungsumgebung" [2]. An manchen Stellen in der Literatur spricht man auch von einem Image. Ein Image enthält ein momentanes Abbild aller notwendigen Informationen, die zur Laufzeit eines Programms und für seine Ausführung benötigt werden. Ein Prozess ist dann die eigentliche Ausführung eines solchen Images.

Linux unterscheidet also streng zwischen dem eigentlichen Programm und der Umgebung, in der dieses Programm abläuft. Die einzelnen Prozesse stehen in einer hierarchischen Ordnung zueinander, d.h. jeder Prozess führt eine eindeutige Nummer (*Prozessnummer*, *PID*) und die Nummer seines *Elternprozesses* (*Erzeugerprozess*) mit sich. Jeder Prozess kennt also die Nummer des Elternprozesses, aber der Elternprozess weiß nicht direkt, welche Prozesse er im einzelnen erzeugt hat. Diese Verbindung ist wichtig im Hinblick auf den Informationsfluss zwischen Prozessen.

Der in der Einleitung erwähnte *Multiplexbetrieb* wird oft auch *Mehrprogrammbetrieb*, *Multitasking* oder auch *Multiprogramming* genannt, was aber nicht zu verwechseln ist mit dem Begriff *Multiprocessing*. Darunter versteht man den Betrieb mehrerer Prozessoren in einem System. Diese Art des Multiprocessing findet man in der Literatur oft unter dem Namen *SMP* (*symmetric multiprocessing*).

---

<sup>1</sup> engl. Aufgabe

## 2 Prozesse

### 2.1 Organisation von Prozessen

Um Prozesse verwalten zu können, muss der Kernel eine klare Vorstellung davon haben, was jeder einzelne Prozess gerade macht. Er muss z. B. wissen, welche Priorität der Prozess hat, ob er gerade auf der CPU ausgeführt wird oder ob er blockiert ist, welche Rechte ein Prozess hat usw.

Diese Informationen werden für jeden einzelnen Prozess in einem Eintrag in einer Prozesstabelle abgespeichert. Die Prozesstabelle ist in Linux statisch angelegt und in der Größe auf `NR_TASKS = 512` festgelegt. Festgelegt wird diese Größe in `/include/linux/tasks.h`.

### 2.2 Aufbau eines Prozesstabelleneintrages

Die Definition eines solchen Prozesstabelleneintrags kann man entlang der Definition in der Datei `/usr/include/linux/sched.h` nachvollziehen. Dort wird eine Struktur `struct task_struct` definiert, die den Aufbau eines solchen Prozesstabelleneintrags festlegt. Diese Struktur ist sehr komplex. Sie enthält nicht nur viele Einträge, sondern auch Pointer zu anderen Datenstrukturen, die selbst auch wieder Pointer zu anderen Strukturen haben.

Eine ausführliche Beschreibung der `task_struct` findet man im Buch *Linux Kernelprogrammierung* [4].

Ich will hier nur auf die wichtigsten Bereiche dieser `task_struct` eingehen:

- *Allgemeine Zustandsinformationen:*  
Hier ist der aktuelle Status des Prozesses (siehe nächster Abschnitt) definiert. Außerdem findet man noch Flags und Verweise auf die Speicherbereiche, die für diesen Prozess reserviert sind.
- *Elemente der Signalbehandlung:*  
Signale sind numerische Kennwerte, die Prozessen zugestellt werden können. Hier werden diese für einen Prozess registriert und verwaltet. Signale werden im Abschnitt 5 behandelt
- *Verknüpfungen von Prozessen:*  
Prozesse sind auf viele verschiedene Arten miteinander verkettet. Es existieren z. B. Verweise auf alle Prozesse, die im Zustand `runnable` sind, Verweise auf Prozesse, die den gleichen Elternprozess haben etc.
- *Prozessidentifikation:*  
Dieser Teilbereich beinhaltet Daten darüber, welche Rechte der Prozess hat. Diese ergeben sich aus der Benutzernummer (UID) und der Gruppennummer (GID). Außerdem muss jeder Prozess eine eindeutige Prozessnummer (PID) haben.
- *Prozesspriorität:*  
Damit ein einzelner Prozess nicht die CPU auf lange Sicht für sich belegen kann, arbeitet Linux die Prozesse der Reihe nach ab und gibt jedem Prozess immer nur einen bestimmten Zeitraum für seine Arbeit (siehe Abschnitt 3.1).
- *Accounting-Informationen:*  
Linux speichert auch Informationen zu den Speicherzugriffen im virtuellen Speicher. Hier kommt es häufig vor, dass ein Prozess innerhalb seines gültigen Prozessadressraums auf eine Speicherseite zugreifen will, die noch nicht geladen wurde. Die Hardware meldet dann einen Seitenzugriffsfehler, woraufhin der Kernel die gewünschte Speicherseite nachlädt.
- *Zeitmessung:*  
Dieser Teil enthält u. a. Informationen zur Startzeit des Prozesses und zur verbrauchten Rechenzeit

- *Interprozesskommunikation:*  
Dient zum Speichern von prozessspezifischen Semaphoren und den Wiederherstellungsstrukturen.
- *Dateisystem:*  
Enthält eine Informationsstruktur zum aktuellen Dateisystem.

### 2.3 Lebenszyklus von Prozessen

Prozesse durchlaufen eine Reihe von Zuständen, bevor sie letztendlich ihre Arbeit getan haben und terminieren. Prozesse haben demnach einen bestimmten Lebenszyklus. Eine Übersicht über die Zustände, die ein Prozess von seiner Entstehung bis zur Terminierung durchläuft, gibt folgende Abbildung:

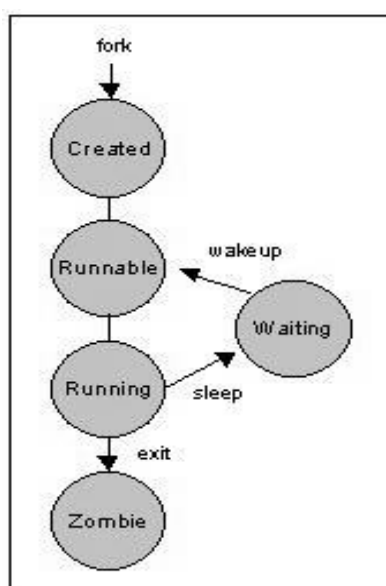


Abbildung 1: Prozesszustandsmodell

Wie in der Abbildung zu sehen ist, wird ein neuer Prozess nach seiner Erzeugung in den Zustand `runnable` gebracht. Sobald ein Prozess durch den Scheduler auf der CPU ausgeführt werden kann, wird der Zustand des Prozesses in `running` umgeschaltet und der Prozess dann tatsächlich auf der CPU ausgeführt. In den meisten Fällen pendelt ein Prozess zwischen den Zuständen `runnable` und `waiting` mehrfach hin und her.

Diese Zustände werden in der `task_struct` in der Variablen `state` für jeden Prozess festgelegt. Die folgende Auflistung liefert eine Übersicht über die Konstanten der Variable `state`:

#### TASK\_RUNNING

Ein Prozess im Zustand `running`, wird gerade auf der CPU ausgeführt oder wartet darauf, dass er ausgeführt wird. Wartende Prozesse und solche, die gerade ausgeführt werden haben also den gleichen Zustand. Einen Zustand `TASK_READY` für Prozesse, die auf die Zuteilung zur CPU warten, gibt es unter Linux nicht.

#### TASK\_INTERRUPTIBLE

Im Normalfall versucht ein Prozess irgendwann aus einer Datei zu lesen oder auch auf den Bildschirm zu schreiben (zwei Beispiele für Systemaufrufe). In so einer Situation wartet dann der Prozess auf die Rückkehr aus dem Systemaufruf und kann von daher unterbrochen werden. Nach der Rückkehr aus dem Systemaufruf wird der Prozess wieder in den Zustand `TASK_RUNNING` versetzt.

#### TASK\_UNINTERRUPTIBLE

Gegenüber dem Zustand `TASK_INTERRUPTIBLE` kann ein Prozess der den Zustand `TASK_UNINTERRUPTIBLE` erreicht hat nicht durch das Eintreffen eines Signals unterbrochen werden. Ein Prozess in diesem Zustand ist aber ebenfalls grundsätzlich wartend. Dieser Zustand wird aber selten benutzt.

#### TASK\_ZOMBIE

Das Beenden eines Prozesses und die Freigabe seines Prozesstabelleneintrags in der Prozess-tabelle gehen nicht in einem Schritt von statten. Der Elternprozess des terminierten Prozesses (Kindprozess) muss noch dessen Rückgabewert abgreifen. Solange das noch nicht geschehen ist, hat ein Prozess den Zustand `TASK_ZOMBIE`, d.h. er hat seine Arbeit getan, muss aber noch ein wenig in der Prozesstabelle herumliegen.

#### TASK\_STOPPED

Ein Prozess kann durch verschiedene Signale angehalten. Seine Bearbeitung kann dann zu einem späteren Zeitpunkt wieder aufgenommen werden. Der Zustand eines angehaltenen Prozesses ist dann `TASK_STOPPED`.

### 3 Der Scheduler

Der (Prozess-)Scheduler hat die Verwaltung aller Prozesse unter seiner Kontrolle. Es handelt sich dabei um einen der wichtigsten Teile des gesamten Kernels. Er sorgt dafür, dass jedem Prozess die gleiche Chance zukommt gerechnet zu werden. Dazu müssen ihm bestimmte Verfahren, sogenannte *Schedulingstrategien* (siehe Abschnitt 3.1) bekannt sein, nach denen einzelne Prozesse der CPU zugeteilt werden oder auch wieder angehalten werden können. Er sorgt also für das Umschalten zwischen den verschiedenen Prozesszuständen, die im vorigen Kapitel vorgestellt wurden. In der Taskliste taucht der Scheduler meist unter dem Begriff `swapper` mit der PID 0 auf. Der Scheduler selbst ist in den Kernel-Quellen unter `kernel/sched.c` zu finden.

Grundsätzlich durchläuft der Scheduler in regelmäßigen Abständen die gesamte Prozesstabelle. Unter anderem wird nach Prozessen gesucht, die gerechnet werden wollen, was an dem Prozesszustand `TASK_RUNNING` zu erkennen ist. Damit aber nicht sinnlos alle 512 Einträge (`NR_TASKS`) untersucht werden müssen, sind die existierenden Prozesse über die Komponenten `next_task` und `prev_task` miteinander verkettet. Von daher wird immer ab dem ersten Startprozess die Prozesstabelle entlang der Verkettung durchlaufen, bis man wieder am Anfang ist. Dieser Durchlauf durch die Prozesstabelle findet in regelmäßigen Abständen statt. Er wird durch einen Timer-Interrupt ausgelöst, der 100mal pro Sekunde ausgelöst wird. Dies bedeutet aber nicht, dass der Durchlauf durch die Prozesstabelle so häufig durchgeführt wird, aber dies ist die kleinste Zeiteinheit, die unter Linux bekannt ist.

### 3.1 Scheduling-Strategien

Linux verwendet verschiedenen Scheduling-Strategien um zwischen Prozessen mit dem Status `runnable` umzuschalten. Diese Strategien sind allesamt *präemptive Schedulingstrategien*, was so viel bedeutet, dass generell alle Prozesse vom Scheduler verdrängt werden können.

Die zur Verfügung stehende Rechenzeit wird in Linux in sogenannte Zeitscheiben aufgeteilt. Anhand diesem Zeitscheiben-Verfahren teilt der Scheduler den Prozessen die Rechenzeit nach drei unterschiedlichen Strategien zu:

#### SCHED\_FIFO

Bei diesem Verfahren existiert für jede Priorität von Prozessen eine Warteschlange, aus der die Prozesse dann nacheinander abgearbeitet werden. Wird der Prozessor frei, wählt der Scheduler den vordersten Prozess aus der Schlange aus. Sobald dieser dann die CPU wieder frei gibt, wird er wieder hinten in die Warteschlange eingefügt. Ein Prozess mit niedriger Priorität kann aber erst dann gerechnet werden, wenn in keiner Warteschlange ein Prozess mit höherer Priorität bereit ist ausgeführt zu werden.

#### SCHED\_RR

Dieses Verfahren realisiert das sogenannte *Round-Robin-Verfahren*. Das Verfahren ist ähnlich zu dem von `SCHED_FIFO`. Allen Prozessen wird in regelmäßigen Zeitspannen eine bestimmte Zeit auf dem Prozessor garantiert. Die Prozesse haben somit eine Maximallaufzeit, nach der sie dann vom Prozessor entfernt werden. Diese Laufzeit nennt man Zeitscheibe<sup>2</sup>.

#### SCHED\_OTHER

Dies ist der klassische UNIX-Schedulingalgorithmus und ist die voreingestellte Schedulingstrategie für neu erstellte Prozesse unter Linux. Das Verfahren arbeitet rein mit Prioritäten. Jeder Prozess erhält eine Basispriorität (*nice level*), die normalerweise 0 ist. Es gibt einen Wertebereich von -20 bis +19, wobei eine niedrigere Zahl eine höhere Priorität bedeutet. Ist ein Prozess in der Warteschlange, wird der *nice level* in bestimmten Zeitabständen inkrementiert (nach [5] jeweils in 210ms). Nach diesem Zeitintervall wird ein *reschedule* durchgeführt und der Prozess der jeweils höchsten Priorität dem Prozessor zugewiesen.

Bleibt noch zu sagen, dass `SCHED_FIFO` und `SCHED_RR` nur für Superuser-Prozesse vorgesehen sind.

In welche der oben genannten Strategien ein Prozess eingeteilt ist, ist für jeden Prozess in der `task_struct` in der Variablen `policy` festgehalten.

## 4 Prozesserzeugung

### 4.1 Anlegen neuer Prozesse

Neue Prozesse können in Linux durch den Systemruf `fork()` angelegt werden. Dabei kann jeder Prozess auch wieder einen neuen Prozess anlegen. Der Prozess, der den Aufruf zum Erzeugen eines Prozesses gegeben hat, der also `fork()` aufgerufen hat, wird *Elternprozess (parent process)* genannt. Der Prozess der daraus erzeugt worden ist, heißt *Kindprozess (child process)*.

Der eigentliche Ablauf, der hinter dem Systemruf `fork()` steckt, ist im Prinzip sehr simpel. Es werden alle Speicherseiten (Datensegment, Stacksegment und Heap) des Elternprozesses kopiert.

---

<sup>2</sup> engl. time slice

Der Kindprozess bekommt jetzt noch eine neue PID und einen Eintrag in der Prozessliste. Zu diesem Zeitpunkt laufen dann zwei Ausführungen des gleichen Programms in jeweils unterschiedlichen Prozessen. Jetzt ist aber der neu erzeugte Prozess dann in der Lage, anstelle des alten Programms, mittels des Systemaufrufs `exec()`, ein neues Programm von der Festplatte zu laden und dieses dann auszuführen.

So wird bei der Erzeugung eines neuen Prozesses meist nacheinander ein `fork()` und dann kurz danach ein `exec()` verwendet. Dies wäre ja reine Zeitverschwendung, wenn zuerst mehrere Megabyte kopiert werden müssten, nur dass kurz danach die Speicherseiten des neuen Prozesses mit neuen Inhalten geladen würden. Aus diesem Grund verwendet Linux das sogenannte *Copy-On-Write-Verfahren*. Bei diesem Verfahren wird für den Kindprozess zunächst keine Kopie des Datensegments, Stacksegments und Heaps erstellt, sondern die Originale des Elternprozesses, durch Verweise auf die selben Speicherbereiche, genutzt. Erst wenn einer der beiden Prozesse versucht, in einem der entsprechenden Speicherbereiche zu schreiben, erzeugt der Kern für diesen Speicherbereich wirklich eine Kopie.

Das Besondere an der Funktion `fork()` ist, dass sie nur einmal aufgerufen wird, aber zweimal zurückkehrt. Die Rückkehr zum Kindprozess wird durch den Rückgabewert 0 angezeigt, die Rückkehr zum Elternprozess zeigt sie durch die Rückgabe der Prozess-ID (PID) des neuen Kindprozesses an. Der Grund für diese Rückgabewert-Regelung ist, dass ein Elternprozess mehrere Kindprozesse, aber jeder Kindprozess nur einen Elternprozess haben kann. Somit kann ein Elternprozess die Prozess-ID seiner Kinder nur zum Zeitpunkt ihrer Erzeugung mit `fork()` erfahren.

Wenn oben vom Kopieren die Rede ist, stellt sich nun die Frage, in welchen Punkten sich Elternprozess und Kindprozess unterscheiden. Der erste Unterschied ist, wie oben schon erwähnt, der Rückgabewert (0 beim Kindprozess; PID des Kindprozesses beim Elternprozess). Weiter unterscheiden sie sich in ihrer Prozess-ID (PID) und in Ihrer Parent-Prozess-ID (PPID). Außerdem werden Dateisperrern des Elternprozesses nicht an den Kindprozess vererbt.

## 4.2 Prozesshierarchie

Wenn alle Prozesse immer kopiert werden, ist die Frage berechtigt, wo denn eigentlich der erste Prozess herkommt.

Beim Start des Systems werden üblicherweise spezielle Prozesse eingerichtet. So zum Beispiel der *Scheduler-Prozess* (oder auch `swapper`) mit der PID 0. Dieser Systemprozess ist Teil des Kernels.

Ein weiterer, wichtiger Prozess, ist der sogenannte `init`-Prozess, der vom `swapper` kreiert wird. Dieser Prozess erhält die PID 1 und ist damit der Vater aller Prozesse. Die zu diesem Prozess gehörige Programmdatei befindet sich in dem Verzeichnis `/sbin/init`. Der `init`-Prozess ist für die systemspezifische Initialisierung zuständig. Das Programm `init` liest dazu die Datei `/etc/inittab` und konfiguriert auf dieser Basis das System. Es führt sozusagen den Bootvorgang durch, indem es spezielle Skripte abarbeitet, Daemonen lädt, Dateisysteme mountet und schließlich die sogenannten `getty's` („get tty“) startet. Hier wird eine Konsole bereitgestellt, um einen Benutzer einloggen zu lassen. Der Login wird dann vom `login`-Prozess übernommen. Dem Benutzer wird eine Shell zur Verfügung gestellt, von der aus weitere Programme gestartet werden können. Die Shell ist damit der Elternprozess für jedes Programm, das aus ihr gestartet wird.

## 4.3 Daemon-Prozesse

Wie im vorigen Abschnitt schon erwähnt, startet der `init`-Prozess unter anderem auch Daemon-Prozesse (oder auch Dämonen). Dies sind spezielle Prozesse, die ständig im Hintergrund ablaufen. Sie werden beim Booten des Systems gestartet und erst beim Herunterfahren oder Absturz des Systems beendet. Die Daemon-Prozesse werden für ständig anfallende Aufgaben, wie z. B. Überprüfen auf neu angekommene Mails etc. verwendet. In der Taskliste sind diese Prozesse meist am Namensanhang `d` zu erkennen. Einige wichtige Daemon-Prozesse sind:

- `syslogd`  
Dieser Prozess erlaubt es jedem Programm Systemmeldungen auf der Konsole auszugeben. Diese Meldungen werden zusätzlich in einer Log-Datei eingetragen.
- `sendmail` (Mail-Dämon)  
Er ist der Standard-Mail-Dämon und ist dafür verantwortlich Mails zu empfangen oder zu senden.
- `cron`  
Der `cron`-Daemon ist für das regelmäßige Ausführen von Kommandos zuständig. Er prüft in bestimmten Zeitabständen den Inhalt der `crontab`-Dateien. Diese legen die Zeitpunkte fest, zu denen entsprechende Kommandos automatisch ablaufen sollen.
- `lpd`  
Er steuert das Drucksystem und kann dazu verwendet werden, Druckaufträge entgegennehmen oder auf Anforderung auch wieder zu löschen.
- `inetd`  
Der `inetd`-Daemon (Netz-Dämon) überprüft ständig die Netzwerkanschlüsse auf das Ankommen neuer Netzanforderungen.

Die Besonderheit dieser Prozesse ist, dass sie alle mit Superuser-Rechten (`UID=0`) laufen. Ihr Elternprozess ist der `init`-Prozess. Ein Daemon-Prozess ist nicht an ein Terminal gebunden, er muss daher andere Wege haben Fehlermeldungen auszugeben. Dazu benutzt ein Daemon-Prozess den oben vorgestellten Prozess `syslogd` Daemon-Prozess.

## 5 Signale an Prozesse

Signale sind Hardware- oder Softwareinterrupts, die in eigenen Programmen zur Interprozesskommunikation oder zum Ansprechen von Prozessen durch den Benutzer verwendet werden. Einem Prozess muss z. B. mitgeteilt werden können, dass ein Benutzer eine Programmabbruchtaste (meist `Strg + C` oder `DELETE`) gedrückt hat, oder dass eine Ausnahmesituation, wie z. B. Division durch 0 aufgetreten ist. Damit ein Prozess auch auf Signale reagieren kann, werden bestimmte Signalmechanismen benötigt. Die Signalwerte kommen in Form von ganzzahligen Werten zwischen 1 und 32 vor. Im Prozesstabelleneintrag gibt es in der Komponente `signal` einen Merker für angekommene Signale. Dies ist eine vorzeichenlose 32 Bit Ganzzahl, wobei jedes Bit für ein Signal vorgesehen ist. Außerdem existiert eine Tabelle mit Funktionsadressen, die bestimmen, welche Funktion bei welchem Signal ausgeführt wird. Ich möchte nur einige Signale im Folgenden vorstellen. Eine ausführlichere Auflistung bietet z. B. [5].

- `SIGHUP`  
"Hang-Up"-Signal des Terminals (führt zur Beendigung des Prozesses, da Kindprozess der Shell).
- `SIGKILL`  
Führt zum sofortigen Beenden des Prozesses
- `SIGTERM`  
Prozess beendet sich daraufhin selbst
- `SIGSTOP`  
Stoppt die Ausführung des Prozesses
- `SIGFPE`  
Floating Point Exception (z. B. bei Division durch 0 oder Überlauf)

## 6 Informationen über Prozesse

Die Aufgabe des `proc`-Dateisystems ist es, Informationen über den Kern und die Prozesse auf einfache Art bereitzustellen. Hierbei sollen teilweise kryptischen Aufrufe umgangen und die Informationen auf möglichst lesbare Weise angeboten werden. Einige der System-Utilities, wie z. B. `ps` verlassen sich darauf. Das `proc`-Dateisystem wird normalerweise unter `/proc` gemountet. Unter diesem Verzeichnis existieren dann virtuelle Verzeichnisse und Dateien, die die einzelnen Prozesse repräsentieren. Der Name eines solchen Verzeichnisses entspricht der PID eines Prozesses. Eine gekürzte Ausgabe des `proc`-Dateisystems gibt Abbildung 2:

```
roman@linux:~ > ls -l /proc
insgesamt 1
dr-xr-xr-x   3 root      root          0 Jul  2 09:41 1
dr-xr-xr-x   3 roman     users        0 Jul  2 09:41 1006
dr-xr-xr-x   3 roman     users        0 Jul  2 09:41 1054
dr-xr-xr-x   3 roman     users        0 Jul  2 09:41 1057
dr-xr-xr-x   3 roman     users        0 Jul  2 09:41 1060
dr-xr-xr-x   3 roman     users        0 Jul  2 09:41 1062
dr-xr-xr-x   3 root      root          0 Jul  2 09:41 2
dr-xr-xr-x   3 root      root          0 Jul  2 09:41 3
dr-xr-xr-x   3 mail      mail         0 Jul  2 09:41 789
...
-r--r--r--   1 root      root          0 Jul  2 09:41 cpuinfo
-r--r--r--   1 root      root          0 Jul  2 09:41 filesystems
-r--r--r--   1 root      root          0 Jul  2 09:41 loadavg
-r--r--r--   1 root      root          0 Jul  2 09:41 meminfo
-r--r--r--   1 root      root          0 Jul  2 09:41 partitions
```

In einem Verzeichnis eines Prozesses befinden sich jeweils wieder weitere Dateien, die sämtliche Informationen zum diesem Prozess bieten.

Unter `/proc` selbst findet man auch noch nützliche Dateien, wie z. B.:

<b>cpuinfo</b>	Informationen zur CPU, auf der das System läuft.
<b>filesystems</b>	Liste der bekannten Dateisystemtypen.
<b>loadavg</b>	durchschnittliche Systemauslastung für die letzten 1, 5 und 15 Minuten
<b>meminfo</b>	Anzahl sämtlicher, benutzter und freier Bytes des Hauptspeichers und des Swapbereichs.
<b>partitions</b>	Angaben zu den Partitionen

Eine vollständige Auflistung der Programme aus dem `/proc`-Verzeichnis findet man in [3] ab Seite 418 ff.

## 7 Befehle zur Prozessverwaltung

Linux kennt einige Befehle, die Informationen über Prozesse liefern und die es erlauben Prozesse zu beenden oder im Hintergrund abzuarbeiten. Einige davon sollen im Folgenden näher beschrieben werden.

### 7.1 Prozessstatistik mit `ps`

Das Kommando `ps` (process status) liefert Statusinformationen zu laufenden Prozessen. Die Informationen werden dem `proc`-Dateisystem entnommen und sind somit immer aktuell.

```
roman@linux:~ > ps
  PID TTY          TIME CMD
 3633 pts/1        00:00:00 bash
 3661 pts/2        00:00:00 bash
 3667 pts/2        00:00:25 soffice.bin
 3690 pts/2        00:00:00 soffice.bin
 3691 pts/2        00:00:00 soffice.bin
 3692 pts/2        00:00:00 soffice.bin
 3693 pts/2        00:00:00 soffice.bin
 3694 pts/2        00:00:00 soffice.bin
 3757 pts/2        00:00:00 ps
```

Ohne die Angabe von Parametern zeigt das Kommando `ps` die Daten zu den Prozessen an, die vom Bildschirm (bzw. vom Terminal) des Benutzers gestartet wurden. Weitere Optionen zeigt die folgende Tabelle:

Option	Bedeutung
-a	Anzeige von Informationen zu allen Prozessen (aller Benutzer)
-f	Ausgabe mit Hinweisen zu Abhängigkeiten zwischen Prozessen
-j	PPID, PGID, TPGID, SID und Status anzeigen
-l	Erzeugen eines Langformats für Prozessinformationen
-m	Anzeige von zusätzlichen Speicherinformationen
-u	Anzeige des Benutzernamens und der Startzeit für die Prozesse
-x	Auch Anzeige von Prozessen ohne Kontrollterminal

### 7.2 Laufende Übersicht mit `top`

Das Programm `top` liefert eine Übersicht aller momentan laufenden Prozesse inklusive Systemauslastung. Diese Übersicht wird laufend aktualisiert. So kann man das laufende Prozessgeschehen ständig beobachten.

### 7.3 Prozesshierarchie mit *pstree*

Das Kommando `pstree` gibt einen Baum mit allen Prozessen auf dem Bildschirm aus. Der Baum zeigt, welcher Prozess von welchem anderen Prozess gestartet wurde. Wenn eine Prozessnummer angegeben wird, beginnt der Baum an dieser Stelle, ansonsten bei `init`, also dem ersten Prozess der beim Systemstart ausgeführt wird. Abbildung 4 liefert eine stark gekürzte Ausgabe des mit dem Kommando `pstree` erzeugten Baums:

```
roman@linux:~ > pstree
init(1)--atd
    |-bonobo-moniker-
    |-cron
    |-syslogd(375)
    |-klogd(378)
    |-kapmd
    |-kdeinit--artsd
    |       |-7*[kdeinit]
    |       `--kdeinit---bash---pstree
    |-11*[kdeinit]
    |-kdm--X
    |       `--kdm---kde---kwrapper
    |-keventd
    |-6*[mingetty]
    |-nscd---nscd---5*[nscd]
    |-sshd
    |-syslogd
    `--wombat
```

### 7.4 Prozesse beenden mittels *kill*

Das Kommando `kill` sendet ein Signal an die angegebenen Prozesse. In den meisten Fällen reagieren Prozesse auf das Eintreffen eines Signals mit der vorzeitigen Beendigung. Der allgemeine Aufbau des Kommandos ist `kill [-Signal] [-l] Prozessnummer(n)`.

Folgende Signale können verwendet werden:

Nummer	Name
1	SIGHUP
2	SIGINT
3	SIGQUIT
9	SIGKILL
15	SIGTERM

Ohne die Angabe einer Signalnummer oder eines Signalnamens wird das Signal `SIGTERM` an die angegebenen Prozesse gesendet. Das Signal zeigt nur Wirkung, wenn der Zielprozess die gleiche Benutzernummer hat wie der Sendeprozess.

Man kann auch das Kommando `killall` verwenden. Hier muss man aber anstelle der Prozessnummer den Namen des Programms angeben.

### 7.5 Benutzerdefinierte Priorität mit `nice`

In manchen Fällen kann es sinnvoll sein einem Prozess bewusst mehr oder weniger Rechenzeit zuzuteilen. Um ein Programm mit erhöhter oder reduzierter Priorität zu starten, kann man das Programm `nice`, mit der Syntax `nice [Optionen] Programm`, verwenden. Standardmäßig würde ein Programm mit dem `nice`-Wert 0 gestartet werden (ohne eine Angabe von `nice`). Als Option kann man jetzt einen Wert von -20 (höchste Priorität) bis +19 (niederste Priorität) angeben. Werte kleiner als 0 dürfen nur von Programme von `root` angegeben werden. Die "normalen" Anwender können somit nur Programme mit reduzierter Priorität starten. Wird bei der Verwendung von `nice` auf eine Option verzichtet, startet das angegebene Programm mit der Priorität +10.

### 7.6 Prozesse unabhängig von der Shell mit `nohup` und `&`

Startet man aus einer Shell heraus ein Programm, wird die Shell so lange blockiert, bis das ausgeführte Programm beendet ist. Wie wir oben gesehen haben, ist das gestartete Programm ja ein Kindprozess der Shell und diese wartet darauf, dass ihr Kind sich beendet. Will man dies verhindern, dann startet man das Programm einfach durch den Zusatz `&`, also z. B. `netscape &`. So kann die Shell während der Ausführung von Netscape weiterhin Kommandos entgegennehmen und ausführen.

Startet man ein Kommando als Prozess oder als Hintergrundprozess (`&`) in einer Shell und schließt diese noch während das Programm ausgeführt wird, wird das Programm auch automatisch beendet, da die Shell der Vaterprozess des gestarteten Programms ist. Manchmal möchte man aber, dass ein Programm trotzdem weiterläuft, obwohl die Shell beendet wurde. Dies erreicht man mit dem Kommando `nohup` (no hang up). Bei der Verwendung dieses Kommandos ist aber auch die Shell nicht mehr für Textausgaben (z. B. Fehlermeldungen) zuständig. Daher werden Ausgaben gegebenenfalls in einer Datei `nohup.out` im lokalen Verzeichnis ausgegeben.

Oft werden die beiden Kommandos auch zusammen verwendet, z. B. dann `nohup netscape &`.

## Literatur

- [1] Prof. Dr. J. Leslie Keedy  
*Technische Informatik (Betriebssystemkonzepte) Vorlesungsskript, 1998*  
Universität Ulm, Fakultät für Informatik, Abteilung Rechnerstrukturen
- [2] M. Wielsch, J. Prahm, H.-G. Eßer.  
*LINUX intern - Technik, Administration und Programmierung, 1. Auflage 1999*  
Verlag DATA BECKER
- [3] Helmut Herold  
*Linux / Unix Systemprogrammierung, 2., überarbeitete Auflage 1999*  
Verlag Addison-Wesley
- [4] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus,  
Claus Schröter, Dirk Verworner  
*Linux Kernelprogrammierung, 5. Auflage 1999*  
Verlag Addison-Wesley
- [5] Daniel P. Bovet, Marco Cesati  
*Understanding the Linux Kernel, January 2001: First Edition*  
Verlag O'Reilly
- [6] Florian Holeczek  
*Prozeßmanagement unter Linux, 31. August 2001*  
Universität Ulm, Fakultät für Informatik, Abteilung Verteilte Systeme
- [7] Jörg Wendland  
*Multiprogrammbetrieb und Prozessverwaltung unter Linux, Sommersemester 2000*  
Universität Ulm, Fakultät für Informatik, Abteilung Verteilte Systeme
- [8] Michael Kofler  
*Linux – Installation, Konfiguration, Anwendung, 5. Auflage 2000*  
Verlag Addison-Wesley