ADMIN:DOCS

Linux Administration #3: Prozesse und Ressourcenverwaltung

In diesem dritten Teil unserer Linux-Administrations-Serie lernst du alles über Prozesse und Ressourcenverwaltung. Von grundlegenden Konzepten bis zur praktischen Anwendung.

Kategorie: Linux-Administration **Veröffentlicht:** 31.10.2024 **Zuletzt geprüft:** 02.11.2026

Willkommen zum dritten Teil unserer technischen Wiki-Serie über Linux-Administration!

Nachdem wir in den vorherigen Artikeln die Grundlagen der Systemadministration und die fortgeschrittene Benutzerverwaltung behandelt haben, widmen wir uns heute einem weiteren fundamentalen Aspekt:

Der Verwaltung von Prozessen und Systemressourcen.

Als Linux-Administrator ist es wichtig zu verstehen, wie Prozesse funktionieren und wie du Systemressourcen effektiv verwaltest. Stell dir dein Linux-System wie ein großes Unternehmen vor, in dem viele verschiedene Mitarbeiter (Prozesse) unterschiedliche Aufgaben erledigen und Ressourcen (CPU, RAM, etc.) teilen müssen.

> ⚠ **Hinweis:** In diesem Artikel verwenden wir Ubuntu/Debian als Beispiel-Distribution. Die grundlegenden Konzepte sind auf allen Linux-Systemen gleich, aber die Installation von Paketen und einige Konfigurationspfade können sich je nach Distribution unterscheiden. Wenn du eine andere Distribution verwendest, konsultiere bitte die entsprechende Dokumentation für die spezifischen Installationsbefehle und Pfade.

Quelle: https://admindocs.de/de/linux-administration/linux-administration-prozesse-und-ressourcenverwaltung

Was ist ein Prozess?

Ein Prozess ist ein laufendes Programm in deinem System. Um das besser zu verstehen, hier eine Analogie:

- Ein Programm ist wie ein Mitarbeiterhandbuch mit Arbeitsanweisungen
- Ein Prozess ist wie ein aktiver Mitarbeiter, der diese Anweisungen ausführt
- Jeder Prozess hat eine eindeutige ID (PID), wie eine Personalnummer
- Prozesse können miteinander kommunizieren, wie Mitarbeiter in Teams

Arten von Prozessen

In Linux gibt es verschiedene Arten von Prozessen, ähnlich wie in einem Unternehmen verschiedene Mitarbeitertypen existieren. Lass uns die wichtigsten Prozesstypen kennenlernen:

Wichtige Prozess-Kategorien im Überblick:

- **Systemdienste (Daemons):** Laufen im Hintergrund ohne direktes Terminal, starten meist beim Booten via systemd (z. B. `sshd`, `cron`, `systemd-journald`).
- **Benutzerprozesse:** Werden von interaktiven Benutzern gestartet, laufen unter deren UID und sind an ein Terminal oder eine GUI gebunden (z. B. `bash`, `firefox`, `vim`).
- **Kernel-Threads:** Werden direkt vom Kernel erzeugt und verwaltet, besitzen keine Userspace-Speicherbereiche und steuern Hardware-Interrupts, I/O und Paging (z. B. `kthreadd`, `kswapd`, `ksoftirqd`).

System-Prozesse (Daemons)

Merkmale von System-Prozessen:

- Starten automatisch beim Booten über den Init-Prozess (`systemd`)
- Laufen dauerhaft im Hintergrund und warten auf Anfragen oder Timer
- Besitzen in der Regel kein steuerndes Terminal (`TTY = ?`)
- **Typische Beispiele:** Webserver (`nginx`, `apache2`), Datenbanken (`mariadb`, `postgresql`), SSH-Daemon (`sshd`)

Benutzerprozesse

Merkmale von Benutzerprozessen:

- Werden interaktiv durch angemeldete Benutzer initiiert
- Laufen mit den effektiven Rechten des jeweiligen Benutzers
- Sind typischerweise an eine Pseudo-Terminal-Session (`pts/X`) oder eine grafische Sitzung gebunden
- Werden standardmäßig beim Schließen des Terminals beendet (sofern nicht via `nohup`, `disown` oder Multiplexer entkoppelt)
- **Typische Beispiele:** Texteditoren (`nano`, `vim`), Shells (`bash`, `zsh`), Compiler (`gcc`), Webbrowser

Kernel-Prozesse

Merkmale von Kernel-Prozessen:

- Werden direkt vom Kernel initialisiert (Elternprozess ist `kthreadd`, PID 2)
- Laufen ausschließlich im Kernel-Space mit höchsten Rechten
- Erkennbar in der Prozessliste an eckigen Klammern: `[kworker/...]`, `[kswapd0]`
- **Typische Beispiele:** `kswapd` (Speicherauslagerung und Paging), `ksoftirqd` (Software-Interrupt-Verarbeitung)

Wie ein Prozess entsteht und arbeitet

Wenn du ein Programm startest, passieren im Hintergrund viele wichtige Schritte. Lass uns verstehen, wie ein Prozess von der Entstehung bis zum Ende funktioniert.

Der Lebenszyklus eines Prozesses

1. **Programm laden** Binary wird von Speichermedium in RAM geladen, PID zugewiesen
2. **Prozess initialisieren** Virtueller Adressraum und Filedeskriptoren eingerichtet

Quelle: https://admindocs.de/de/linux-administration/linux-administration-prozesse-und-ressourcenverwaltung

3. **Ausführung (Running/Sleeping)** CPU-Zuteilung durch Scheduler, I/O-Verarbeitung
4. **Terminierung & Exit** Exit-Status an System übergeben, Speicher freigegeben
5. **Zombie & Cleanup** Elternprozess liest Exit-Code (wait), Prozessdeskriptor entfernt

Praktisches Beispiel: Firefox starten

```
# Firefox im Vordergrund starten
firefox

# Firefox im Hintergrund starten
firefox &
```

Was passiert dabei?

- Shell erstellt einen neuen Kindprozess via `fork()`
- Das Firefox-Binary wird via `execve()` in den Adressraum geladen
- Eine freie PID wird zugewiesen
- Der Scheduler teilt dem Prozess CPU-Zeit zu

Prozess-Status überprüfen

Ein Prozess durchläuft während seiner Lebensdauer verschiedene Zustände im Linux-Kernel:

- **Running / Runnable (R):** Der Prozess wird gerade auf einer CPU ausgeführt oder wartet in der Run-Queue auf Rechenzeit.
- **Interruptible Sleep (S):** Der Prozess schläft und wartet auf ein Ereignis (z. B. Tastatureingabe, Netzwerkpaket oder Timer). Er reagiert sofort auf Signale.
- **Uninterruptible Sleep (D):** Der Prozess wartet blockierend auf I/O (meist Festplatte/NFS) und kann nicht durch Signale (selbst `SIGKILL`) unterbrochen werden.

- **Stopped (T):** Der Prozess wurde manuell angehalten (z. B. durch `Ctrl+Z` oder das Signal `SIGSTOP`).
- **Zombie (Z):** Der Prozess hat seine Ausführung beendet, sein Eintrag verbleibt jedoch in der Prozesstabelle, bis der Elternprozess den Exit-Status mittels `wait()` ausliest.

> **Kernel-Update (Linux 6.6 / 6.12 LTS): Vom CFS zum EEVDF-Scheduler** Über mehr als 15 Jahre war der *Completely Fair Scheduler* (CFS) der Standard-CPU-Scheduler des Linux-Kernels. Mit Linux 6.6 und gefestigt im aktuellen LTS-Kernel 6.12 wurde CFS vollständig durch **EEVDF** (*Earliest Eligible Virtual Deadline First*) abgelöst. EEVDF eliminiert historische Latenzprobleme bei interaktiven Tasks, indem es Prozesse nicht mehr nur nach aufgelaufener CPU-Laufzeit (`vruntime`) steuert, sondern virtuelle Deadlines und „Lag“ berechnet. Ergänzend ermöglicht das mit Kernel 6.12 eingeführte Framework `sched_ext` die Ausführung flexibler, via eBPF programmierter Scheduler direkt im Kernel.

Detaillierte Prozessinformationen mit `ps -l`:

```
# Status eines Prozesses im Langformat anzeigen
ps -l
```

Beispielausgabe:

```
F S   UID   PID  PPID  C PRI   NI ADDR SZ WCHAN  TTY          TIME     CMD
0 S  1000  1234  1000  0  80   0  -    234 do_sys pts/0    00:00:01 bash
```

Bedeutung der wichtigsten Spalten:

- `S`: Prozess-Status (`R`, `S`, `D`, `T`, `Z`)
- `UID`: Benutzer-ID des Prozesseigentümers
- `PID`: Eindeutige Prozess-ID
- `PPID`: Prozess-ID des übergeordneten Elternprozesses
- `PRI`: Dynamische Kernel-Priorität
- `NI`: Nice-Wert (-20 bis +19) zur Feinsteuerung
- `SZ`: Virtuelle Speichergröße
- `WCHAN`: Kernel-Funktion, in der der Prozess schläft (z. B. beim Warten auf I/O)
- `TIME`: Akkumulierte CPU-Rechenzeit
- `CMD`: Aufgerufener Befehl / Programmname

Prozess-Kommunikation und Signale

Prozesse in Linux müssen häufig miteinander kommunizieren und gesteuert werden. Das Betriebssystem stellt hierfür POSIX-Signale und verschiedene Inter-Prozess-Kommunikationswege (IPC) bereit.

1. Prozess-Signale verstehen

Signale sind asynchrone Benachrichtigungen des Kernels an einen Prozess, um ihn über Systemereignisse zu informieren oder sein Verhalten zu steuern:

- **SIGTERM (15):** Standard-Terminierungssignal. Fordert den Prozess auf, sich sauber zu beenden, offene Dateien zu schließen und Puffer zu leeren.
- **SIGKILL (9):** Harter, sofortiger Abschuss durch den Kernel. Kann vom Prozess weder abgefangen noch ignoriert werden; kein sauberes Aufräumen möglich.
- **SIGHUP (1):** Hangup-Signal. Wird gesendet, wenn das steuernde Terminal schließt. Von Daemons genutzt, um Konfigurationsdateien im laufenden Betrieb neu einzulesen.
- **SIGINT (2):** Unterbrechungssignal (Keyboard Interrupt, `Ctrl+C`).
- **SIGSTOP (19):** Hält den Prozess sofort an (nicht abfangbar).
- **SIGCONT (18):** Setzt einen angehaltenen Prozess fort.

- **SIGUSR1 (10) / SIGUSR2 (12):** Benutzerdefinierte Signale für anwendungsspezifische Aktionen (z. B. Log-Rotation).

2. Signale in der Praxis

Signale werden über das Kommando `kill` oder Hilfswerkzeuge wie `pkill` und `killall` übermittelt:

```
# Prozess geordnet beenden (Standard; SIGTERM)
kill -15 1234
# Alternativ: kill -TERM 1234

# Prozess sofort und unweigerlich beenden (Notfall; SIGKILL)
kill -9 1234
# Alternativ: kill -KILL 1234

# Prozess anhalten (pausieren)
kill -STOP 1234

# Angehaltenen Prozess fortsetzen
kill -CONT 1234

# Konfiguration eines Daemons neu laden (ohne Neustart)
sudo kill -HUP $(pgrep nginx)
```

Inter-Prozess-Kommunikation (IPC)

Wenn Prozesse Daten miteinander austauschen oder Aktivitäten koordinieren müssen, kommen verschiedene IPC-Mechanismen zum Einsatz:

- **Pipes (`|`) & FIFOs (Named Pipes):** Unidirektionale Datenströme zwischen verwandten Prozessen oder über Dateisystem-Puffer.
- **Signale:** Übertragung kurzer Status- und Steuerbefehle.
- **Shared Memory:** Gemeinsam genutzter physikalischer RAM-Bereich für extrem schnellen Datenaustausch ohne Kernel-Kopieraufwand.

- **Message Queues:** Strukturierte Nachrichten-Warteschlangen für asynchronen Nachrichtenaustausch.
- **Unix Domain Sockets:** Bidirektionale, dateisystembasierte Kommunikation auf demselben Host (z. B. `/run/systemd/...`).
- **Semaphoren:** Zählergestützte Synchronisationsmechanismen zur Vermeidung von Race Conditions bei gleichzeitigem Zugriff auf gemeinsame Ressourcen.

2. Pipes in der Praxis

Pipes sind wie Rohre, durch die Daten von einem Prozess zum anderen fließen:

Einfache Pipe-Beispiele

```
ls -l | grep ".txt"

# Listet nur .txt Dateien
```

Was passiert hier?

- `ls -l` erzeugt eine Liste aller Dateien
- Die Pipe (`|`) leitet diese Liste weiter
- `grep` filtert nach „.txt“

Mehrfache Pipes

```
cat logfile.txt | grep "Error" | sort | uniq

# 1. Liest die Logdatei
# 2. Filtert nach "Error"
```

```
# 3. Sortiert die Zeilen
# 4. Entfernt Duplikate
```

Weitere Inter-Prozess-Kommunikationsmethoden

Neben Pipes gibt es noch weitere wichtige Methoden, wie Prozesse miteinander kommunizieren können. Lass uns diese im Detail anschauen.

1. Shared Memory (Gemeinsamer Speicher)

Shared Memory ist die schnellste Form der Inter-Prozess-Kommunikation. Statt Daten über Pipes durch den Kernel zu kopieren, mappen zwei oder mehr Prozesse denselben physischen Speicherbereich direkt in ihren jeweiligen virtuellen Adressraum:

- **Prozess A & B:** Greifen parallel auf denselben RAM-Bereich zu
- **Kernel-Bypass:** Nach dem Einrichten keine Syscall-Overheads bei Lese-/Schreibzugriffen
- **Synchronisationsbedarf:** Erfordert zwingend Semaphoren oder Mutexe, um gleichzeitiges Überschreiben zu verhindern

Shared Memory Segmente anzeigen:

```
ipcs -m

# Zeigt alle System-V Shared Memory Segmente
```

Beispielausgabe:

```
------ Shared Memory Segments --------
```

```
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          root       644        16384      2          dest
```

Shared Memory Status im Detail überprüfen:

```
ipcs -m -i 0

# Zeigt detaillierte Segment-Informationen (PID des Erstellers, Zugriffszeiten)
```

2. Message Queues (Nachrichtenwarteschlangen)

Message Queues speichern diskrete Nachrichtenblöcke in einer vom Kernel verwalteten Warteschlange. Ein oder mehrere Prozesse können Nachrichten in die Queue schreiben, während andere sie asynchron abholen:

- **Entkopplung:** Sender und Empfänger müssen nicht gleichzeitig laufen
- **Strukturierte Daten:** Nachrichten können Typen und Prioritäten tragen
- **Verwaltung:** Anzeige aktiver Queues via `ipcs -q`

Semaphoren und Inter-Prozess-Synchronisation

Semaphoren fungieren als Zähler und Ampeln im Betriebssystem, um den gleichzeitigen Zugriff auf geteilte Ressourcen zu regeln.

1. Semaphoren verstehen

- **Binäre Semaphoren (Mutexe):** Kennen nur zwei Zustände (0 oder 1; gesperrt oder frei). Nur ein Prozess darf den geschützten Bereich betreten.
- **Zählende Semaphoren:** Zählen verfügbare Instanzen einer Ressource (z. B. maximal 5 parallele Datenbank-Verbindungsslots). Übersteigt der Bedarf die Grenze, wird der anfragende Prozess schlafen gelegt (`Sleeping`), bis ein anderer Slot freigegeben wird.

2. Praktische Anwendung

Semaphoren im System anzeigen:

```
ipcs -s

# Zeigt alle aktiven Semaphoren-Arrays
```

Beispielausgabe:

```
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 0          root       600        1
```

Detaillierte Informationen zu einem Semaphore-Set abrufen:

```
ipcs -s -i 0
```

Beispiel: Gemeinsame Ressourcennutzung mit Semaphoren

```
# Python-Beispiel für Semaphoren-Nutzung
```

```
from multiprocessing import Semaphore

# Semaphore für maximal 3 parallele Zugriffe
sem = Semaphore(3)

def geschuetzter_bereich():
    sem.acquire()    # "Ampel prüfen / Zähler dekrementieren"
    try:
        # Hier kommt der geschützte Code
        print("Zugriff auf gemeinsame Ressource gewährt")
    finally:
        sem.release()    # "Ressource freigeben / Zähler inkrementieren"
```

Praktische Anwendungsfälle

Prozess- und Ressourcenverwaltung im Serverbetrieb

Lass uns anhand von realen Szenarien lernen, wie du Prozesse und Ressourcen in typischen Produktivumgebungen verwaltest. Typische Ziele sind:

- **Performance & Stabilität:** Schnelle Antwortzeiten auch bei Lastspitzen
- **Ressourcentrennung:** Verhindern, dass Hintergrundjobs oder unbändige Prozesse Webserver lahmlegen
- **Gezielte Steuerung:** Priorisierung via Nice-Werte, CPU-Pinning (`taskset`) und I/O-Scheduling (`ionice`)

Webserver-Prozesse optimieren

1. Höhere Priorität setzen

```
sudo renice -n -5 $(pgrep apache2)
```

2. Dedizierte CPU-Kerne zuweisen

```
sudo taskset -pc 4-7 $(pgrep mysqld)
```

3. I/O-Priorität setzen

```
sudo ionice -c 1 -n 0 -p $(pgrep mysqld)
```

Webserver-Optimierung in der Praxis

Stell dir vor, du betreibst einen Apache-Webserver, der unter hoher Last steht:

- **Schnelle Antwortzeiten:** Webserver-Threads priorisieren
- **Effiziente Ressourcennutzung:** CPU-Affinität für parallele Worker festlegen
- **Stabile Verfügbarkeit:** Hintergrunddienste drosseln

1. Apache-Prozesse identifizieren

```
ps aux | grep apache2

# Zeigt alle Apache-Prozesse und ihre PIDs
```

Beispielausgabe:

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
www-data  1234  2.5  1.2 245300 48200 ?        S    10:00   0:15 /usr/sbin/apache2 -k
start
www-data  1235  1.8  1.1 243200 45100 ?        S    10:00   0:11 /usr/sbin/apache2 -k
start
```

2. Prozess-Priorität optimieren

```
sudo renice -n -5 $(pgrep apache2)

# Gibt Apache eine höhere Scheduling-Priorität
```

3. CPU-Affinität setzen

```
sudo taskset -pc 0-2 $(pgrep apache2)

# Bindet Apache-Worker gezielt an die ersten drei CPU-Kerne (0, 1, 2)
```

Datenbank-Server optimieren

Bei einer MySQL/MariaDB-Datenbank ist die I/O- und CPU-Performance besonders kritisch:

1. MySQL-Prozess identifizieren

```
ps aux | grep mysql

# Zeigt den MySQL-Daemon und seine PID
```

2. Ressourcen in Echtzeit überwachen

```
top -p $(pgrep mysqld)

# Zeigt die gezielte Echtzeit-Ressourcennutzung der Datenbank
```

3. I/O-Priorität setzen

```
sudo ionice -c 1 -n 0 -p $(pgrep mysqld)

# Weist der Datenbank Realtime-I/O-Priorität (Klasse 1) zu
```

Ressourcenverwaltung

Als Linux-Administrator ist es deine Aufgabe, Systemressourcen fair und stabil zu verteilen, damit kein einzelner Dienst das Gesamtsystem zum Stillstand bringt.

Systemressourcen verstehen

Die drei zentralen Säulen der Ressourcenverwaltung:

- **Prozessor (CPU):** Rechenzeit, Anzahl der Kerne, Scheduling-Warteschlangen und Kontextwechsel.
- **Arbeitsspeicher (RAM & Swap):** Physischer RAM, virtueller Adressraum, Page Cache und Swap-Auslagerung.
- **Ein-/Ausgabe (I/O):** Blockgerät-Durchsatz (NVMe/SSD/HDD), IOPS, Latenz und Netzwerkbandbreite.

CPU-Ressourcen überwachen

Eine verlässliche Überwachung hilft dir, Performance-Engpässe und CPU-Sättigung frühzeitig zu diagnostizieren.

Wichtige CPU-Metriken im Detail:

- **`%user` (`us`):** Rechenzeit in Userspace-Anwendungen (z. B. Webserver, Datenbanken, Python-Skripte).
- **`%system` (`sy`):** Rechenzeit im Linux-Kernel (z. B. Systemaufrufe, Interrupts, I/O-Verarbeitung).
- **`%iowait` (`wa`):** Zeit, in der die CPU unbeschäftigt auf das Beenden von Festplatten- oder Dateisystem-I/O wartet. Ein hoher Wert deutet auf ein I/O-Bottleneck hin, nicht auf eine überlastete CPU!
- **`%idle` (`id`):** Unausgelastete CPU-Zeit im Leerlauf.
- **`%steal` (`st`):** In virtuellen Maschinen (VMs): Zeit, die der Hypervisor der VM entzieht, um andere VMs auf demselben Host zu bedienen.

CPU-Überwachung in der Praxis

```
# Grundlegende CPU- und Hardware-Informationen abfragen
lscpu
```

```
# CPU-Auslastung in Echtzeit interaktiv beobachten
top

# Auslastung pro CPU-Kern im Sekundentakt messen (aus Paket sysstat)
mpstat -P ALL 1
```

CPU-Load Average richtig interpretieren

Die **Load Average** gibt die durchschnittliche Anzahl der Prozesse an, die im Zustand `R` (Running / Runnable) oder `D` (Uninterruptible Sleep / waiting on disk I/O) auf Ausführung warten. Sie wird für 1, 5 und 15 Minuten berechnet:

- **Wert < Kernanzahl:** Das System hat freie Rechenkapazitäten.
- **Wert = Kernanzahl:** Das System ist exakt zu 100 % ausgelastet (alle Kerne voll beschäftigt, keine Warteschlange).
- **Wert > Kernanzahl:** CPU-Sättigung – Prozesse müssen in der Run-Queue warten. Bei einer 4-Kern-CPU bedeutet eine Load von 8.0, dass im Schnitt 4 Prozesse gleichzeitig auf freie Rechenzeit warten.

Load Average mit `uptime` abfragen:

```
uptime
```

Beispielausgabe:

```
14:30:05 up 5 days, 3:12,  2 users,  load average: 0.15, 0.25, 0.30
                                                   |     |     └── 15 Minuten
                                                   |     └──────  5 Minuten
```

```
                                                └──────────────  1 Minute
```

Speicherverwaltung und Ressourcenlimits

Die verlässliche Verwaltung des Arbeitsspeichers schützt dein System vor Instabilität und Deadlocks.

Speichertypen verstehen

- **Physischer RAM:** Schneller primärer Arbeitsspeicher für laufenden Code und aktive Datenstrukturen.
- **Swap-Speicher:** Auslagerungsbereich auf SSD oder Festplatte (als Swap-Partition oder Swap-Datei). Verhindert Abstürze bei Speicherspitzen, ist aber um Größenordnungen langsamer als physischer RAM.
- **Buffer & Cache:** Ungenutzter RAM wird vom Kernel dynamisch als Page Cache für Dateizugriffe verwendet. Benötigen Anwendungen mehr Arbeitsspeicher, wird der Cache sofort automatisch freigegeben.

Speichernutzung überwachen

Gesamten Speicherstatus anzeigen mit `free`:

```
free -h
```

Beispielausgabe:

```
               total        used        free      shared  buff/cache   available
```

```
Mem:            15Gi       5.2Gi       6.8Gi       1.2Gi       3.8Gi       9.1Gi
Swap:          4.0Gi       128Mi       3.9Gi
```

Bedeutung der Kennzahlen:

- `total`: Gesamter physisch verbauter Arbeitsspeicher
- `used`: Von Prozessen belegter Speicher (ohne Cache)
- `free`: Völlig ungenutzter Speicher
- `buff/cache`: Für I/O-Puffer und Dateisystem-Caching genutzter Speicher
- `available`: Die entscheidende Metrik: Wie viel RAM für neue Anwendungen zur Verfügung steht, ohne in den Swap auszulagern (inkl. freigebbarem Cache)

Speicher-Limits und Ressourcenverwaltung

Die Kontrolle des Speicherverbrauchs verhindert, dass speicherhungrige Prozesse (wie Memory Leaks) den Server zum Erliegen bringen.

Speicherlimits: Traditionell vs. Modern

- **ulimit (POSIX-Ebene):** Begrenzt Ressourcen innerhalb der aktuellen Shell-Session (z. B. maximale Dateigröße, virtueller Speicher, Core-Dumps).
- **`/etc/security/limits.conf`:** Setzt statische Benutzer- und Gruppengrenzen beim Login via PAM.
- **systemd cgroups v2 (Moderner Standard):** Zuverlässige, hierarchische Ressourcensteuerung auf Ebene von Diensten, Slices und Benutzer-Scopes.

> **Praxis-Tipp: cgroups v2 und `systemd-run` statt `limits.conf`** Klassische `limits.conf`-Grenzen gelten nur pro einzelnem Prozess – forkt ein Prozess 100 Kindprozesse, kann er das System dennoch lahmlegen. Moderne Linux-Systeme nutzen **cgroups v2**. Mit `systemd-run` kannst du beliebige Skripte oder Befehle ad hoc in einer isolierten CGroup mit festen Speicher- und CPU-Grenzen ausführen:

```
systemd-run --user --scope -p MemoryMax=1G -p CPUQuota=50% ./build.sh
```

Speicherlimits setzen (Legacy POSIX)

Systemweite Shell-Limits anzeigen:

```
ulimit -a
```

Prozess-spezifische Limits über PAM (`limits.conf`):

□□ Praktisches Beispiel: Limits für MySQL/MariaDB:

In der Datei `/etc/security/limits.conf` eintragen:

```
# <domain>      <type>  <item>         <value>
mysql           soft    memlock        524288
mysql           hard    memlock        524288
mysql           soft    nofile         65536
mysql           hard    nofile         65536
```

OOM-Killer und proaktives Speichermanagement

Wenn der physische RAM und der Swap erschöpft sind, tritt eine *Out of Memory* (OOM) Situation ein.

> **Modernes OOM-Management: `systemd-oomd` & PSI (Pressure Stall Information)**
> Frühere Linux-Systeme verließen sich ausschließlich auf den reaktiven Kernel-OOM-Killer, der oft erst eingreift, wenn das System durch extremes Paging ("Thrashing") bereits minutenlang blockiert ist. Aktuelle Distributionen (wie Ubuntu 24.04/26.04 oder Fedora) nutzen stattdessen **`systemd-oomd`**. Dieser Userspace-Dienst überwacht kontinuierlich die Kernel-Schnittstelle **PSI** (`/proc/pressure/memory`). Steigt der Speicherdruck über definierte Schwellenwerte, beendet `systemd-oomd` proaktiv ganze betroffene Cgroup-Slices, bevor der Linux-Kernel einfriert.

Kernel-OOM-Score verstehen und anpassen

Der Kernel vergibt für jeden Prozess einen `oom_score`, der anhand von Speichergröße, Laufzeit und Privilegien berechnet wird. Über `oom_score_adj` (-1000 bis +1000) lässt sich dieser Score justieren:

OOM-Score eines Prozesses prüfen:

```
cat /proc/1234/oom_score
```

Wichtigen Dienst vor dem OOM-Killer schützen (-1000):

```
echo -1000 | sudo tee /proc/$(pgrep mysqld)/oom_score_adj

# Wert -1000 schließt den Prozess vollständig vom Kernel-OOM-Killer aus
```

Unkritischen Prozess gezielt zur Beendigung vormerken (+500):

```
echo 500 | sudo tee /proc/$(pgrep heavy_worker)/oom_score_adj
```

I/O-Limits und Ressourcenkontrolle

Die Kontrolle der Ein- und Ausgabe (I/O) ist entscheidend für die Systemleistung. Hier lernst du, wie du I/O-Limits setzt und überwachst.

I/O-Scheduling und Limits verstehen

Unkontrollierte Ein-/Ausgabeoperationen (z. B. große Backups oder Datenbank-Dumps) können Speicher- und Festplatten-Queues sättigen und interaktive Dienste blockieren.

- **I/O-Scheduler:** Verwaltet Festplatten-Queues im Kernel (z. B. `none` für NVMe, `mq-deadline` oder `bfq` für SATA-SSDs und HDDs).
- **I/O-Priorisierung via `ionice`:** Erlaubt die Zuweisung von I/O-Klassen für Prozesse.
- **cgroups v2 I/O-Throttling:** Begrenzung von IOPS (`IOReadIOPSMax=` , `IOWriteIOPSMax=`) und Bandbreite (`IOReadBandwidthMax=`).

I/O-Limits in der Praxis

I/O-Priorität für Prozesse mit `ionice` setzen:

```
# Best-Effort Klasse (2) mit niedriger Priorität (7) für Hintergrundprozess setzen
ionice -c 2 -n 7 -p $(pgrep rsync)
```

Die drei I/O-Klassen im Überblick:

- **Klasse 1 (Realtime):** Höchste Priorität. Erhält immer sofortigen Festplattenzugriff vor allen anderen Prozessen (nur als `root` setzbar). Vorsicht: Kann bei Missbrauch das System aushungern.
- **Klasse 2 (Best-Effort):** Standardklasse für alle normalen Prozesse. Einstellbar von Priorität 0 (höchste) bis 7 (niedrigste).
- **Klasse 3 (Idle):** Niedrigste Priorität. Darf nur dann auf die Festplatte zugreifen, wenn kein anderer Prozess I/O anfordert. Ideal für Backups, Indizierungen und Virenscans.

I/O-Priorität für Backup-Prozesse setzen:

```
# Backup-Skript komplett in der Idle-Klasse starten
ionice -c 3 ./backup.sh
```

I/O-Statistiken in Echtzeit überwachen:

```
# I/O-Auslastung pro Blockgerät im Sekundentakt messen (erfordert sysstat)
iostat -x 1
```

Systemüberwachung

Als Linux-Administrator benötigst du kontinuierliche Einblicke in Systemauslastung, Sättigung und Fehlerzustände.

Wichtige Monitoring-Tools: Von `top` bis `btop`

- **`top`:** Der vorinstallierte Standard auf jedem Linux-Server. Zeigt Uptime, Load, Task-Zustände, CPU-Verteilung und Speichernutzung.

- **`htop`:** Interaktiver Prozess-Viewer mit Farbbalken, Mausunterstützung, Baumansicht (`F5`) und intuitiver Prozess-Steuerung.
- **`btop`:** Moderne, ressourcenschonende TUI-Alternative mit detaillierten Graphen für CPU, Speicher, Festplatten und Netzwerk.
- **`vmstat` & `iostat`:** Schnelle Snapshot-Werkzeuge für Kontextwechsel, Run-Queues und Plattenauslastung.

Beispiel für die interaktive Statusanzeige mit `top`:

```
top
```

Typische Ausgabezeilen und Interpretation:

```
top - 19:09:30 up  9:46,  2 users,  load average: 1.39, 1.30, 1.35
Tasks: 282 total,   1 running, 281 sleeping,   0 stopped,   0 zombie
%Cpu(s):  8.3 us,  4.0 sy,  0.0 ni, 87.2 id,  0.0 wa,  0.4 hi,  0.1 si,  0.0 st
MiB Mem :  23930.2 total,   2294.4 free,  11825.2 used,  10645.2 buff/cache
MiB Swap:  50253.9 total,  50253.9 free,      0.0 used.  12105.0 avail Mem
```

- **Zeile 1:** Uptime, eingeloggte Benutzer und Load Average (1, 5, 15 Min).
- **Zeile 2 (Tasks):** Prozesszähler – warnt sofort vor hängenden Zombies (`zombie`).
- **Zeile 3 (%Cpu):** Prozentsätze für User (`us`), System (`sy`), Idle (`id`) und I/O-Wait (`wa`).
- **Zeile 4/5 (Mem/Swap):** Physischer Speicher und Swap; entscheidend ist `avail Mem`.

Erweiterte Systemanalyse mit SAR (System Activity Reporter)

Das Paket `sysstat` protokolliert Metriken im Hintergrund und erlaubt historische Performance-Analysen.

SAR installieren und Dienst aktivieren:

```
sudo apt update && sudo apt install -y sysstat
sudo systemctl enable --now sysstat
```

CPU-Auslastung mit Intervallen messen:

```
sar -u 1 5

# Misst 5-mal im Abstand von jeweils 1 Sekunde
```

Beispielausgabe:

```
Linux 6.12.0-amd64 (server01)    02.11.2026      _x86_64_        (4 CPU)

09:16:28        CPU     %user     %nice   %system   %iowait    %steal     %idle
09:16:29        all      4.00      0.00      2.00      0.00      0.00     94.00
09:16:30        all      4.02      0.00      1.76      0.00      0.00     94.22
09:16:31        all      4.04      0.00      1.77      0.00      0.00     94.19
Durchschnitt:   all      4.02      0.00      1.84      0.00      0.00     94.14
```

Speicher- und Swap-Nutzung abfragen:

```
# RAM-Statistiken abrufen
sar -r

# Swap-Nutzung und Paging analysieren
sar -S
```

Troubleshooting

Wenn Server langsam reagieren, Prozesse hängen oder Alarme schlagen, hilft ein systematisches Vorgehen nach der **USE-Methode** (*Utilization, Saturation, Errors*):

1. **Symptome & Engpässe identifizieren:** Welche Ressource ist gesättigt (CPU, RAM, Swap oder Festplatten-I/O)?
2. **Verursachende Prozesse isolieren:** Welche PIDs beanspruchen die Run-Queue oder den Speicher?
3. **Gezielte Maßnahme ergreifen:** Priorität anpassen (`renice`), I/O drosseln (`ionice`), Dienst neu starten oder limitieren via cgroups v2.

Häufige Probleme und Lösungen

CPU-Probleme erkennen und beheben:

1. CPU-Auslastung prüfen

```
top

# Hohe Werte bei %us oder %sy deuten auf Probleme hin
```

Quelle: https://admindocs.de/de/linux-administration/linux-administration-prozesse-und-ressourcenverwaltung

2. Problematische Prozesse identifizieren

```
ps aux --sort=-%cpu | head -5

# Zeigt die 5 CPU-intensivsten Prozesse
```

3. Prozess-Priorität anpassen

```
renice +10 $(pgrep firefox)

# Gibt Firefox niedrigere Priorität
```

Speicherprobleme diagnostizieren:

1. Speichernutzung überprüfen

```
free -h

# Wichtige Indikatoren:
# - Wenig freier RAM
# - Hohe Swap-Nutzung
```

2. Speicherfresser finden

```
ps aux --sort=-%mem | head -5

# Zeigt die 5 speicherintensivsten Prozesse
```

3. Cache leeren (wenn nötig)

```
sync && echo 3 > /proc/sys/vm/drop_caches
```

I/O-Probleme erkennen und beheben

Typische Indikatoren für I/O-Engpässe:

- Das System reagiert zäh, obwohl die CPU-Auslastung niedrig ist (`%idle` hoch, aber `%iowait` über 20 %)
- Befehle im Terminal pausieren sekundenlang, während Platten-LEDs dauerhaft leuchten
- Prozesse hängen im ununterbrechbaren Schlafzustand `D` (`Uninterruptible Sleep`)

1. I/O-Probleme mit `iostat` diagnostizieren

```
iostat -xz 1

# Wichtige Messwerte:
# %util: Auslastung des Geräts (nahe 100% bedeutet Sättigung)
# await: Durchschnittliche Wartezeit von I/O-Anfragen in Millisekunden
# r/s & w/s: Lese- und Schreib-IOPS
```

2. I/O-Verursacher mit `iotop` in Echtzeit aufspüren

```
sudo iotop -o

# Der Schalter -o filtert und zeigt nur Prozesse an, die aktuell I/O erzeugen
```

3. I/O-Probleme beheben

```
# Multi-Queue I/O-Scheduler für SATA-SSDs anpassen
echo mq-deadline | sudo tee /sys/block/sda/queue/scheduler

# Unwichtige Hintergrundprozesse auf Idle-I/O drosseln
sudo ionice -c 3 -p $(pgrep backup)
```

Übungen

Lass uns das Gelernte durch praktische Übungen vertiefen. Hier sind reale Szenarien, die dir in deinem Alltag als Linux-Administrator begegnen können.

Übung 1: Webserver-Optimierung

Szenario: Du betreust einen Apache-Webserver, der unter hoher Last läuft. Benutzer beschweren sich über langsame Ladezeiten, und das Monitoring zeigt eine hohe CPU- und Speicherauslastung.

Anforderungen:

- Apache-Prozesse identifizieren
- Ressourcenverbrauch analysieren

- Performance optimieren
- Prozess-Priorität anpassen
- Ressourcenlimits setzen

Mögliche Lösung:

```
# 1. Apache-Prozesse identifizieren
ps aux | grep apache2

# Zeigt alle Apache-Prozesse und ihre PIDs
# 2. Ressourcenverbrauch analysieren
top -p $(pgrep -d',' apache2)

# Beobachte CPU und RAM-Nutzung
# 3. Prozess-Priorität optimieren
sudo renice -5 $(pgrep apache2)

# Gibt Apache höhere Priorität
# 4. Ressourcenlimits setzen
sudo nano /etc/security/limits.conf

# Füge hinzu:
www-data soft nproc 150

# Maximale Prozessanzahl
www-data soft nofile 8192

# Maximale offene Dateien
```

Übung 2: Performance-Optimierung eines Datenbank-Servers

Szenario: Du betreust einen MySQL-Datenbankserver, der die Kundendatenbank eines Online-Shops verwaltet. In Spitzenzeiten kommt es zu Performance-Einbrüchen und langsamen Antwortzeiten.

Anforderungen:

- MySQL-Prozesse identifizieren
- Ressourcenverbrauch analysieren
- CPU- und I/O-Priorität optimieren
- Speicherlimits anpassen
- Performance überwachen

Mögliche Lösung:

```
# 1. MySQL-Prozesse identifizieren
ps aux | grep mysql

# Zeigt alle MySQL-Prozesse und ihre PIDs
# 2. Ressourcen analysieren
top -p $(pgrep mysqld)

# Beobachte:
# - CPU-Auslastung
# - Speicherverbrauch
# - Load Average
# 3. CPU-Priorität optimieren
sudo renice -10 $(pgrep mysqld)

# Gibt MySQL höhere Priorität
# 4. I/O-Priorität setzen
sudo ionice -c 1 -n 0 -p $(pgrep mysqld)

# Höchste I/O-Priorität für MySQL
# 5. OOM-Killer-Schutz aktivieren
echo -1000 > /proc/$(pgrep mysqld)/oom_score_adj

# Schützt MySQL vor dem OOM-Killer
```

Übung 3: Prozess-Priorisierung und Ressourcenkontrolle

Szenario: Du betreust ein Linux-System, auf dem mehrere Entwickler gleichzeitig arbeiten. Ein rechenintensiver Kompilierungsprozess (gcc) eines Entwicklers belastet das System so stark,

dass andere Benutzer über langsame Reaktionszeiten klagen.

Anforderungen:

- Kompilierungsprozesse identifizieren
- Prozess-Priorität anpassen
- CPU-Ressourcen fair verteilen
- System-Performance für andere Benutzer sicherstellen
- OOM-Killer richtig konfigurieren

Mögliche Lösung:

```
# 1. Kompilierungsprozesse identifizieren
ps aux | grep gcc

# Zeigt alle gcc-Prozesse und ihre PIDs
# 2. Prozess-Priorität anpassen
renice +10 $(pgrep gcc)

# Gibt Kompilierung niedrigere Priorität
# 3. CPU-Limits setzen
# Prozess auf bestimmte CPU-Kerne beschränken
taskset -cp 2,3 $(pgrep gcc)

# Beschränkt gcc auf Kerne 2 und 3
# 4. OOM-Score anpassen
echo 500 > /proc/$(pgrep gcc)/oom_score_adj

# Prozess wird bei Speichermangel eher beendet
```

Befehlsreferenz (Cheatsheet)

Für den schnellen Zugriff bei der Analyse, Überwachung und Steuerung von Prozessen fasst die folgende Referenz die wichtigsten Linux-Kommandos zusammen:

Prozesse im System auflisten Zeigt alle laufenden Prozesse im System im BSD-Format

$ `ps aux`

Prozesse nach CPU-Last sortieren Sortiert Prozesse absteigend nach CPU-Auslastung

$ `ps aux --sort=-%cpu`

Prozesshierarchie visualisieren Zeigt Eltern-Kind-Beziehungen von Prozessen in einer Baumstruktur

$ `ps -ef --forest`

Interaktives System-Monitoring Echtzeit-Monitoring von CPU, RAM, Load Average und Prozessen

$ `top`

Prozess-IDs nach Name finden Sucht PIDs passend zum angegebenen Prozessnamen

$ `pgrep -l <name>`

Geordnetes Beenden anfordern (SIGTERM) Sendet Standard-Terminierungssignal für sauberen Shutdown

$ `kill -15 <PID>`

Sofortigen Prozessabbruch erzwingen (SIGKILL) Erzwingt sofortigen, nicht abfangbaren Prozessabschuss

$ `kill -9 <PID>`

Konfiguration im laufenden Betrieb neu laden (SIGHUP) Sendet SIGHUP zum Neuladen von Daemon-Konfigurationen

$ `kill -1 <PID>`

Prozesse nach Muster beenden Beendet Prozesse zielgerichtet nach Namensfilter

$ `pkill -15 <name>`

Programm mit reduzierter Priorität starten Startet Programm mit höherem Nice-Wert (niedrigere Priorität)

```
$ nice -n 10 <cmd>
```

Scheduling-Priorität anpassen Ändert den Nice-Wert eines bereits laufenden Prozesses

```
$ renice -n 5 -p <PID>
```

I/O-Priorität auf Idle setzen Begrenzt Festplattenzugriffe auf Phasen ohne sonstige I/O-Aktivität

```
$ ionice -c 3 -p <PID>
```

Hintergrundjobs der Shell anzeigen Listet alle Hintergrund-Jobs der aktuellen Session mit PID auf

```
$ jobs -l
```

Prozess immun gegen Terminalabbruch starten Entkoppelt Befehl von SIGHUP beim Schließen der SSH-Session

```
$ nohup <cmd> &
```

Ressourcenverbrauch pro CGroup überwachen Echtzeit-Monitoring des Ressourcenverbrauchs pro systemd-CGroup

```
$ systemd-cgtop
```

Befehl in isolierter CGroup limitieren Startet Befehl mit striktem CPU- und Speicherlimit über cgroups v2

```
$ systemd-run --scope -p CPUQuota=30% -p MemoryMax=500M <cmd>
```

Weiterführende Ressourcen

Die folgenden Leitfäden, Kernel-Dokumentationen und internen Kursmodule vertiefen das Prozess- und Ressourcenmanagement:

Quelle: https://admindocs.de/de/linux-administration/linux-administration-prozesse-und-ressourcenverwaltung

systemd Resource Control Guide Offizieller Leitfaden zur systemd cgroups v2-Ressourcensteuerung

Linux Kernel Procfs Documentation Dokumentation der virtuellen Kernel-Schnittstellen unter `/proc/[PID]`

Linux Performance Tools (Brendan Gregg) Umfassende Methodik und Werkzeugübersicht zur Linux-Performance-Analyse

Linux Administration #2: Benutzerverwaltung Das vorherige Modul: PAM-Stacks, Quotas, POSIX-ACLs & Sudoers

Linux Administration #4: Netzwerk Das nachfolgende Modul: IP-Routing, Netplan, nmcli, DNS & UFW

Befehlszeilenprozessor in Linux Fundamentales Wissen zu Shells, I/O-Streams und Pipes

Fazit

Die Beherrschung des Prozess- und Ressourcenmanagements ist eine Kernkompetenz jedes Linux-Systemadministrators. Durch das Verständnis des Prozesslebenszyklus, den kontrollierten Einsatz von POSIX-Signalen, gezieltes CPU-Scheduling mit Nice-Werten und moderne Ressourcenlimits über systemd und cgroups v2 sorgst du dafür, dass deine Linux-Server auch unter extremen Lastspitzen stabil, reaktionsschnell und vorhersehbar performen.

> **Praxis-Tipp:** Nutze für ressourcenintensive periodische Hintergrundjobs (wie z. B. nächtliche Datenbank-Dumps oder Archivierungen) immer die Kombination aus `nice -n 19` und `ionice -c 3`. Dadurch läuft der Job mit niedrigster CPU- und Festplatten-Priorität und beeinträchtigt den regulären Produktivbetrieb zu keinem Zeitpunkt.

Im nächsten Modul unseres Kurses verbinden wir unsere Server mit der Außenwelt: **Weiter geht es mit:** Linux Administration #4: Netzwerkkonfiguration und Verwaltung

Zur Kursübersicht: Alle Linux-Administrations-Artikel & Module