

Linux

Modul 3: Grundlagen III

Übersicht – Modul 3

- Der Root-User
- Dateirechte
- Verzeichnisse
- Relative und absolute Pfade
- Weitere nützliche Linux-Tools: pipe, find & locate
- **Lernfolien Modul 3:** Das Wichtigste auf einen Blick
- **Praxis:** Gelerntes direkt anwenden – wir arbeiten live in der Shell
- **Praxis:** Übungsblätter

Der Root-User

Der mächtigste Benutzer unter Linux

Der Root-User I

Root – der Superuser unter Linux

Root darf alles → komplette Kontrolle über das System

Aber: Mit großer Macht kommt große Verantwortung!

Beispiele für Root-Rechte:

- Löschen der gesamten Festplatte
- Zugriff auf private Dateien anderer Nutzer

Der Root-User II

Root ist vergleichbar mit dem eingebauten Administrator-Konto unter Windows, nicht mit einem normalen Benutzer, der nur **administrative Rechte** erhalten kann.

Unter Linux können normale Benutzer über sudo berechtigt werden, einzelne Befehle mit **Root-Rechten** auszuführen.

So können Administratoren administrative Aufgaben erledigen, ohne sich direkt als **Root** anzumelden.

Der Root-User III



Root ist das Konto mit den **höchsten Rechten**.

sudo ist ein Werkzeug, um gezielt Befehle mit diesen Rechten auszuführen.

Der Root-User IV

Wie arbeitet man mit Root-Rechten?

Es gibt mehrere Möglichkeiten, Befehle mit administrativen Rechten auszuführen:

- **Direkt als Root anmelden**
  Nicht auf jedem System erlaubt oder empfohlen
- `su` **oder** `su -`
  Wechselt in eine Root-Shell. Dafür wird das **Root-Passwort** benötigt
- `sudo <befehl>`
  Führt einen einzelnen Befehl mit Root-Rechten aus. Dafür muss der Benutzer zur Nutzung von **sudo** berechtigt sein
- `sudo -i`
  Startet eine Root-Shell über einen normalen Benutzer mit sudo-Berechtigung

Der Root-User V

💡 Warum nicht ständig als Root arbeiten?

Root darf alles im System ändern oder löschen. Deshalb ist es sicherer, im Alltag mit einem **normalen Benutzerkonto** zu arbeiten und **Root-Rechte nur dann** zu verwenden, wenn sie wirklich gebraucht werden.

Vorteile von sudo

- **Feine Benutzerkontrolle**
  Es kann gezielt festgelegt werden, welche Benutzer oder Gruppen welche Befehle mit Root-Rechten ausführen dürfen
- **Nachverfolgbarkeit**
  sudo protokolliert, welcher Benutzer welchen Befehl mit erhöhten Rechten ausgeführt hat
- **Mehr Sicherheit im Alltag**
  Benutzer arbeiten normalerweise ohne Root-Rechte und verwenden diese nur bei Bedarf für einzelne administrative Aufgaben


Bei **su** wird in eine Root-Shell gewechselt. Danach ist oft weniger klar, welcher ursprüngliche Benutzer welche einzelnen Befehle ausgeführt hat.

Praxis: Arbeiten mit sudo

Hinweis:
`sudo` ist auf manchen Systemen nicht standardmäßig installiert.

sudo installieren:

```
$ su -
# apt install sudo
```

Syntax: `$ sudo <befehl>`

Beispiel: `$ sudo nano /etc/network/interfaces`

Einschub: su vs. sudo

Viele merken sich **su** als *„super user“*. Das passt oft, weil man damit häufig zu **root** wechselt.

Eigentlich bedeutet der Befehl aber: zu einem anderen Benutzer wechseln.



su = ***s****ubstitute* ***u****ser* oder ***s****witch* ***u****ser*

sudo = ***s****ubstitute* ***u****ser* ***d****o* → *„als anderer Benutzer ausführen“*

oder

sudo = ***s****uperuser* ***d****o* → *„als Superuser ausführen“*

sudo-Rechte vergeben I

Variante 1: Benutzer für sudo freischalten

Auf Debian/Ubuntu ist die Gruppe **sudo** dafür vorgesehen, Benutzern sudo-Rechte zu geben:

```
# usermod -aG sudo <benutzer>
```

Nach der Änderung muss sich der Benutzer **neu anmelden**, damit die neue Gruppenzugehörigkeit aktiv wird.

Beispiel:

```
# usermod -aG sudo max
```

Danach kann der Benutzer Befehle mit Root-Rechten ausführen:

```
$ sudo apt update
```

sudo-Rechte vergeben II

Benutzer aus der sudo-Gruppe entfernen:

Die sudo-Berechtigung kann wieder entzogen werden, indem der Benutzer aus der Gruppe **sudo** entfernt wird:

```
# usermod -rG sudo <benutzer>
```

Beispiel:

```
# usermod -rG sudo max
```

Danach muss sich der Benutzer neu anmelden, damit die Änderung wirksam wird.

sudo-Rechte vergeben III

Variante 2: Nur einzelne Befehle erlauben

Soll ein Benutzer **nicht alle sudo-Rechte**, sondern nur bestimmte Befehle ausführen dürfen, wird dies in der **sudoers-Konfiguration** geregelt.

Sicher bearbeiten mit:

```
# visudo -f /etc/sudoers.d/<benutzer>
```

Beispiel:

```
max ALL=(root) /usr/bin/systemctl restart apache2
```

Der Benutzer max darf diesen Befehl mit Root-Rechten ausführen:

```
$ sudo systemctl restart apache2
```

Andere administrative Befehle sind dadurch nicht automatisch erlaubt.

sudo-Rechte vergeben IV

💡 **/etc/sudoers.d/** ist ein Verzeichnis für zusätzliche sudoers-Regeln.
Das **.d** steht für *directory*.

So können einzelne Regeln getrennt von der Hauptdatei **/etc/sudoers** verwaltet werden.

sudo-Rechte vergeben V

Wichtig:
Diese Einschränkung funktioniert nur, wenn der Benutzer nicht zusätzlich Mitglied der Gruppe **sudo** ist.

Andernfalls hätte er über die Gruppe **sudo** weiterhin umfassende administrative Rechte.

💡 **sudoers-Regeln sind additiv.**
Eine zusätzliche Einzelregel begrenzt einen Benutzer nicht, wenn er über eine andere Regel – z. B. über die Gruppe **sudo** – bereits alle Rechte hat.

sudo ohne Passwortabfrage I

Normalerweise muss ein Benutzer bei **sudo** sein eigenes Passwort eingeben.

In der sudoers-Konfiguration kann festgelegt werden, dass dies **nicht erforderlich** ist.

Wichtiger Hinweis

NOPASSWD sollte sparsam verwendet werden.
Sicherer ist es, die Passwortfreiheit nur für **einzelne, genau definierte Befehle** zu erlauben — nicht pauschal für alle Root-Befehle.

sudo ohne Passwortabfrage II

Beispiel: Nur ein bestimmter Befehl ohne Passwort

```
# visudo -f /etc/sudoers.d/max
```

Eintrag:

```
max ALL=(root) NOPASSWD: /usr/bin/systemctl restart
apache2
```

Bedeutung:

- **max** → Benutzer
- **ALL** → Regel gilt auf allen Hosts
- **(root)** → Ausführung als Root
- **NOPASSWD:** → keine Passwortabfrage
- **/usr/bin/systemctl restart apache2** → erlaubter Befehl

sudo allgemein ohne Passwortabfrage I

Soll ein Benutzer bei **sudo** grundsätzlich kein Passwort eingeben müssen, kann dies in der sudoers-Konfiguration festgelegt werden:

```
# visudo -f /etc/sudoers.d/max
```

Eintrag:

```
max ALL=(ALL:ALL) NOPASSWD: ALL
```

Bedeutung:

- **(ALL:ALL)** → Darf als beliebiger Benutzer und beliebige Gruppe ausführen. Das ist die umfassendere Variante und wird oft für vollständige sudo-Rechte verwendet.
- **ALL** → für alle sudo-Befehle

sudo allgemein ohne Passwortabfrage II

Wichtig: Diese Einstellung gibt dem Benutzer sehr weitreichende Rechte.

Sie sollte nur verwendet werden, wenn es wirklich notwendig ist, zum Beispiel in Laborumgebungen, Automatisierungen oder klar kontrollierten Administrationsszenarien.

Sicherer ist es meist, **NOPASSWD** nur für einzelne genau definierte Befehle zu erlauben.

visudo I

Wichtig:

sudoers-Dateien sollten **immer mit visudo** bearbeitet werden.

Das Programm prüft die Syntax und verhindert, dass fehlerhafte Einträge die sudo-Nutzung blockieren.

visudo II



Das „**vi**“ in **visudo** kommt nur historisch vom Editor **vi**.

visudo bedeutet nicht, dass zwingend der Editor **vi** verwendet wird.

Auf Debian öffnet sich häufig **nano**, je nach System und Konfiguration auch **vi** oder **vim**.

visudo III

Der Zweck von visudo ist:

visudo bearbeitet die sudoers-Konfiguration **sicher**.

Dabei macht es vor allem zwei Dinge:

- sperrt die Datei, damit sie nicht gleichzeitig mehrfach bearbeitet wird
- prüft die Syntax, bevor die Änderung übernommen wird

Zur Erinnerung: Shell-Prompt

Der **Shell-Prompt** zeigt an, dass die Shell bereit für einen Befehl ist.

Beispiele:

```
user@hostname:~$
root@hostname:~#
```

Das letzte Zeichen zeigt den Rechtekontext an:

$ → normaler Benutzer

→ Root-Benutzer / Root-Rechte

Prompt-Zeichen in Befehlsbeispielen richtig lesen

In Anleitungen steht vor Befehlen oft ein Prompt-Zeichen:

```
$ ls -l /etc/network
# cat /etc/shadow
```

Bedeutung:

- `$` → Befehl kann mit normalen Benutzerrechten ausgeführt werden
- `#` → Befehl benötigt Root-Rechte

Hinweis zum Shell-Prompt

💡 Der **Shell-Prompt** kann je nach Shell und Konfiguration unterschiedlich aussehen.

Manche Shells verwenden andere Zeichen, zum Beispiel **% statt $** bei **zsh**.

In vielen Linux-Anleitungen steht **$** trotzdem als allgemeines Zeichen für normale Benutzerrechte.

Der Fortsetzungsprompt

Das Zeichen **>** kann erscheinen, wenn die Shell auf eine **Fortsetzung der Eingabe** wartet, zum Beispiel weil ein Anführungszeichen noch nicht geschlossen wurde.

Beispiel:

```
$ echo "Hallo
>
```

Das bedeutet: Die Shell wartet noch auf das schließende Anführungszeichen.

Dateirechte

Wer darf was? Zugriffsrechte unter Linux

Rechte unter Linux I

Linux ist ein **Multiuser-Betriebssystem.**

Nicht jeder darf alles → Dateirechte regeln den Zugriff

Beispiel: Benutzer Alice darf Bob‘s Dateien nicht lesen.

Rechte unter Linux II

Standardmäßig drei Rechte:

- **Read** (**r**) → Datei lesen
- **Write** (**w**) → Datei ändern / schreiben
- **Execute** (**x**) → Datei ausführen

Rechte unter Linux III

Dateirechte mit `ls -l` anzeigen:

```
$ ls -l
-rw-rw-r-- 1 sp sp 0 19. Mär 01:38 datei
```

→ **ls** listet Dateien und Verzeichnisse auf

→ Die Option **-l** steht für ***long format*** und zeigt eine ausführliche Listenansicht

Rechte unter Linux IV

Beispielzeile erklärt:

```
-rw-rw-r--  1  sp  sp  0  19. Mär 01:38  datei
```

- `-rw-rw-r--` → **erstes Zeichen:** Dateityp, **danach:** Zugriffsrechte
- `1` → Anzahl der Hardlinks
- `sp` → Besitzer
- `sp` → Gruppe
- `0` → Dateigröße in Byte
- `19. Mär 01:38` → letztes Änderungsdatum
- `datei` → Dateiname

Dateityp	Rechte Eigentümer	Rechte Gruppe	Rechte Andere
-	rw-	rw-	r--

Dateityp

Dateitypen in Linux (erste Stelle von `ls -l`)

Zeichen	Typ
`-`	normale Datei
`d`	Verzeichnis (*directory*)
`l`	symbolischer Link
`c`	Zeichen-Gerätedatei (*character device*, z. B. **/dev/tty**)
`b`	Block-Gerätedatei (*block device*, z. B. Festplatten)
`s`	Socket
`p`	Named Pipe (auch FIFO, *First In First Out*); Spezialdatei, über die zwei Prozesse Daten austauschen können

Hardlinks I

Hardlinks sind weitere Verweise auf dieselbe Datei.

Die Zahl **1** bedeutet: Es gibt genau **einen Verweis** auf diese Datei, nämlich den Dateinamen.



Bei normalen Dateien steht hier oft **1**, wenn es nur einen Dateinamen gibt, der auf diese Datei verweist.

Hardlinks II

Beispiel: Zahl größer als 1

```
$ echo "Hallo" > datei
$ ln datei zweiter_name
$ ls -l datei zweiter_name
```

→ `ln` = *link*; erstellt einen weiteren Verweis auf eine Datei

→ Beide Namen zeigen nun auf **dieselbe Datei**. Deshalb steht in der Ausgabe bei beiden Einträgen die Zahl **2**.

→ `datei` und `zweiter_name` sind dann keine Kopien, sondern zwei Namen für denselben Dateiinhalt



Die Zahl zeigt, wie viele Dateinamen auf denselben Dateiinhalt zeigen.

Symbolische Links vs. Hardlinks I

Symbolischer Link	**Hardlink**
Verweist auf den **Pfad** einer Datei	Verweist direkt auf dieselben **Daten**
Wie eine Windows-Verknüpfung	Wie ein zweiter echter Dateiname
Kann kaputtgehen, wenn das Ziel gelöscht/verschoben wird	Bleibt gültig, auch wenn ein anderer Name gelöscht wird
Kann auch auf Verzeichnisse zeigen	Normalerweise nicht auf Verzeichnisse
Kann über Dateisystemgrenzen hinweg zeigen	Nur im selben Dateisystem

Symbolische Links vs. Hardlinks II

Stellen Sie sich eine Datei wie ein Dokument in einem Archiv vor.

Ein **symbolischer Link** ist ein Zettel mit der Aufschrift: „Das Dokument liegt im Schrank A, Fach 3.“ Wenn das Dokument verschoben wird, stimmt der Zettel nicht mehr.

Ein **Hardlink** ist dagegen ein zusätzlicher **Registereintrag** für dasselbe Dokument. Egal welcher Registereintrag entfernt wird, das Dokument bleibt erhalten (unter Linux allerdings nur, solange noch mindestens ein „Registereintrag“ darauf verweist 🙂).

Rechte unter Linux V

Rechte modifizieren:

- `chmod` → **Datei- oder Verzeichnisrechte** ändern
- `chown` → **Eigentümer** einer Datei oder eines Verzeichnisses ändern
- `chgrp` → **Gruppe** einer Datei oder eines Verzeichnisses ändern

chmod I

chmod (*change mode*) **– Rechte setzen**

Rechte können auf verschiedene Arten vergeben werden:

- **Oktale Schreibweise:** Rechte mit Zahlen setzen
- **Symbolische Schreibweise:** Rechte mit Buchstaben und Operatoren ändern oder setzen



Oktal: immer neu setzen

Symbolisch: gezielt ändern oder neu setzen

chmod II

Oktale Schreibweise:

7: `rwx` **3:** `-wx`

6: `rw-` **2:** `-w-`

5: `r-x` **1:** `--x`

4: `r--` **0:** `---`

Zahlenwert	Berechtigung
4	Lesen (read, r)
2	Schreiben (write, w)
1	Ausführen (execute, x)

💡 Oktalrechte merken (super einfach!):

`r` = **4**, `w` = **2**, `x` = **1**

Summe = Zahl

Beispiel: `rwx` = 4+2+1 = **7**, `rw-` = 4+2 = **6**, `r-x` = **5**, `-wx` = **3**

chmod III

Bei der **Oktalen Schreibweise** stehen die drei Ziffern immer in derselben Reihenfolge: **1. Ziffer = Eigentümer, 2. Ziffer = Gruppe, 3. Ziffer = Andere**

Beispiele:

```
$ chmod 777 datei
```

→ Jeder darf alles: Eigentümer, Gruppe und Andere dürfen lesen, schreiben und ausführen.

```
$ chmod 770 datei
```

→ Eigentümer und Gruppe dürfen alles, Andere haben keinen Zugriff.

```
$ chmod 660 datei
```

→ Eigentümer und Gruppe dürfen lesen und schreiben, Andere haben keinen Zugriff.

chmod IV

Symbolische Schreibweise:

Rechte werden für u (*user*/Eigentümer), g (*group*/Gruppe), o (*others*/Andere) oder a (*all*/alle, u+g+o) gesetzt.

Beispiel:

```
$ chmod ugo+rwx datei
```

→ Eigentümer, Gruppe und andere dürfen die Datei lesen, schreiben und ausführen

Syntax: `$ chmod[ugoa][+-=][rwx] <datei>`

chmod V

Beispiele für die Symbolische Schreibweise:

`$ chmod o+r datei` → **Andere** dürfen die Datei jetzt **lesen**

`$ chmod ug+w datei` → **Eigentümer** und **Gruppe** dürfen die Datei **schreiben**

`$ chmod o-rwx datei` → **Andere** haben **keine Rechte** mehr

chmod VI

Rechte hinzufügen oder entfernen: + / -

Mit **+** und **-** werden einzelne Rechte gezielt geändert.
Alle anderen Rechte bleiben unverändert.

Beispiel:

```
$ chmod a-x datei
```

Bedeutung:

Allen Benutzern wird das Ausführungsrecht entzogen.
Lese- und Schreibrechte bleiben so, wie sie vorher waren.

chmod VII

Rechte neu setzen:

Mit „= “ werden die Rechte für die angegebene Benutzerklasse neu gesetzt.
Vorherige Rechte dieser Benutzerklasse werden überschrieben.

Beispiel:

```
$ chmod u=rwx,g=r,o= datei
```

Bedeutung:

- Der Besitzer darf lesen, schreiben und ausführen
- Die Gruppe darf nur lesen
- Andere Benutzer erhalten keine Rechte

chmod VIII

Merksatz:

- **+ / -** → Rechte gezielt hinzufügen oder entfernen, Rest bleibt erhalten
- **=** → Rechte neu setzen (bisherige Rechte werden ersetzt)



chmod kann der Eigentümer einer Datei selbst ausführen. Für fremde Dateien oder Systemdateien sind Root-Rechte erforderlich.

chmod: Oktal oder symbolisch? I

Linux-Dateirechte werden intern als **Bitmuster** gespeichert. Deshalb passt die **oktale Schreibweise** gut zur technischen Darstellung:

```
$ chmod 755 script.sh
```

→ Die Zahl beschreibt den gewünschten Endzustand der Rechte:

7 = rwx **5** = r-x **5** = r-x

Die **symbolische Schreibweise** macht Rechteänderungen besser lesbar und ist näher an der Alltagssprache:

```
$ chmod u+x script.sh
$ chmod g+w projektordner
```

→ Sie beschreibt gezielte Änderungen: Wer soll welche Rechte bekommen, verlieren oder exakt gesetzt bekommen?

chmod: Oktal oder symbolisch? II

Oktal, wenn der komplette Rechtezustand eindeutig gesetzt werden soll:

```
$ chmod 644 datei.txt
$ chmod 755 script.sh
```

→ Gut geeignet für Standards, Dokumentationen, Skripte und klare Vorgaben.

Symbolisch, wenn nur einzelne Rechte geändert werden sollen:

```
$ chmod u+x script.sh
$ chmod o-r geheim.txt
```

→ Gut geeignet, wenn bestehende Rechte erhalten bleiben sollen und nur gezielt etwas ergänzt oder entfernt wird. Praktisch für einzelne Anpassungen.

chmod: Oktal oder symbolisch? III

Merksatz:

Oktal = gesamten Zustand setzen
Symbolisch = einzelne Rechte gezielt ändern

💡 **Historisch war die oktale Schreibweise zuerst da**. In der 1. Unix-Manual-Ausgabe von 1971 stand bei **chmod** noch ausdrücklich die Syntax:

```
$ chmod octal file
```

Die symbolische Schreibweise kam später dazu. In der Unix-Version 7 war bereits dokumentiert, dass der Modus *„absolute or symbolic“* sein kann.

chown I

`chown` (*change owner*) **– Eigentümer ändern**

Funktion: Ändert den **Eigentümer** einer Datei oder eines Verzeichnisses

Syntax:

```
# chown <neuer_eigentümer> datei
# chown <neuer_eigentümer> verzeichnis
```

chown II

`chown` kann **Eigentümer und Gruppe in einem Schritt** festlegen.

Syntax:

```
# chown <neuer_eigentümer>:<neue_gruppe> datei
```

Beispiel:

```
# chown sebastian:admins dump.sql
```



Das Ändern des Eigentümers ist normalerweise eine administrative Aufgabe und erfordert daher **Root-Rechte**.

chgrp

`chgrp` (*change group*) **– Gruppenzugehörigkeit einer Datei oder eines Verzeichnisses ändern**

Syntax:

```
$ chgrp [neue_Gruppe] datei
$ chgrp [neue_Gruppe] verzeichnis
```



Ein normaler Benutzer kann **chgrp** nur für eigene Dateien verwenden und nur auf Gruppen ändern, in denen er selbst Mitglied ist.
Für fremde Dateien oder Systemdateien sind Root-Rechte erforderlich.

Rechte-Maske zum Entfernen von Rechten – umask I

umask (*user mask*) ist die **Benutzer-Maske für Standardrechte**:

Legt fest, welche Rechte bei neu erstellten Dateien und Verzeichnissen standardmäßig **entfernt** werden.



Die **umask** wird häufig als ***user file creation mode mask*** erklärt, also sinngemäß:

Benutzer-Maske für die Rechte neu erstellter Dateien und Verzeichnisse

Standardrechte neuer Dateien und Verzeichnisse

Wenn ein Programm eine neue Datei oder ein neues Verzeichnis anlegt, werden zunächst **typische Ausgangsrechte** verwendet:

- Dateien: 666 → rw-rw-rw-
- Verzeichnisse: 777 → rwxrwxrwx

Warum haben neue Dateien kein x?
Eine normale Datei soll nicht automatisch ausführbar sein. Das Ausführungsrecht wird nur gezielt gesetzt, z. B. bei Skripten oder Programmen.

Warum haben Verzeichnisse x?
Bei Verzeichnissen bedeutet **x**: Das Verzeichnis darf betreten bzw. durchsucht werden.

umask II

Die **umask** entfernt anschließend bestimmte Rechte, damit neue Dateien und Verzeichnisse nicht zu offen zugänglich sind.

Beipsiel:

```
$ umask 022
```

Ergebnis:

- **Neue Dateien:** 666 **-** 022 **=** 644 → rw-r--r--
- **Neue Verzeichnisse:** 777 **-** 022 **=** 755 → rwxr-xr-x



Die Ziffern der **umask** gelten von links nach rechts für **Eigentümer**, **Gruppe** und **Andere**.
Die **2** steht für: Schreibrecht entfernen

umask III

Häufige umask-Werte mit Wirkung:

umask	In Worten	Neue Dateien	Neue Verzeichnisse
000	nichts entfernen	**666** → rw-rw-rw-	**777** → rwxrwxrwx
002	Schreibrecht für **andere** entfernen	**664** → rw-rw-r--	**775** → rwxrwxr-x
022	Schreibrecht für **Gruppe und andere** entfernen	**644** → rw-r--r--	**755** → rwxr-xr-x
027	Schreibrecht für Gruppe entfernen, bei anderen alles entfernen	**640** → rw-r-----	**750** → rwxr-x---
077	Für Gruppe und andere alles entfernen	**600** → rw-------	**700** → rwx------

umask IV

Syntax:

`$ umask` → aktuelle Maske anzeigen

`$ umask [wert]` → neue Maske setzen

Beispiele:

`$ umask 022` → Dateien: `644`, Verzeichnisse: `755`

`$ umask 027` → Dateien: `640`, Verzeichnisse: `750`

`$ umask 077` → Dateien: `600`, Verzeichnisse: `700`

umask V

`$ umask [wert]` setzt die **umask nur für die aktuelle Shell-Sitzung**. Nach dem Logout bzw. nach dem Schließen des Terminals ist diese Änderung normalerweise wieder weg.

Damit eine umask dauerhaft gilt, muss sie in einer Start-/Konfigurationsdatei gesetzt werden:

- Auf Debian kann die systemweite Standard-umask über **/etc/login.defs** vorgegeben werden
- Einzelne Benutzer können sie in ihrer eigenen Shell-Konfiguration, z. B. in **~/.profile**, überschreiben

umask VI

`umask` hat nur wenige Optionen und wird meist ohne Optionen verwendet:

`$ umask -S` → zeigt die Maske in symbolischer Form an

Beispiel:

```
$ umask -S
u=rwx,g=rx,o=rx
```

Spezialrechte unter Linux – Sticky Bit I

Sticky Bit (t) – Schutz in gemeinsamen Verzeichnissen

Das **Sticky Bit** wird bei Verzeichnissen verwendet, die von mehreren Benutzern gemeinsam genutzt werden.

Das **Sticky Bit** erlaubt gemeinsames Schreiben in ein Verzeichnis, schützt aber fremde Dateien vor dem Löschen oder Umbenennen.

Spezialrechte unter Linux – Sticky Bit II

In einem Verzeichnis mit **Sticky Bit** dürfen Dateien nur gelöscht oder umbenannt werden von:

- dem **Dateibesitzer**
- dem **Besitzer des Verzeichnisses**
- **Root**

Klassisches Beispiel:

- **/tmp**
- Dort dürfen viele Benutzer Dateien anlegen, aber nicht einfach fremde Dateien löschen.

Spezialrechte unter Linux – Sticky Bit III

Syntax:

Symbolische Schreibweise: `# chmod +t <verzeichnis>`

Oktale Schreibweise: `# chmod 1<_ _ _> <verzeichnis>`

Beispielanzeige mit ls -ld:

```
$ ls -ld /tmp
drwxrwxrwt ...
```

→ Das **t** am Ende zeigt das gesetzte Sticky Bit an

Spezialrechte unter Linux – Sticky Bit IV

Warum heißt das Sticky Bit eigentlich „sticky“?

💡 **Ursprünglich** sorgte das **Sticky Bit** dafür, dass ein Programm nach der Ausführung im Speicher bzw. Swap erhalten blieb. Es „klebte“ also gewissermaßen dort, um beim nächsten Start schneller verfügbar zu sein.

Heute sorgt das **Sticky Bit** dafür, dass Benutzer in gemeinsamen Verzeichnissen zwar **Dateien anlegen** können, aber nicht einfach **fremde Dateien löschen oder umbenennen** dürfen.

(Ob eine Datei bearbeitet werden darf, entscheiden weiterhin die normalen Dateirechte.)

Spezialrechte unter Linux – SetGID I

SetGID (s) – gemeinsame Gruppe beibehalten

Das **SetGID-Bit** wird häufig bei gemeinsam genutzten Projektverzeichnissen eingesetzt.

Ist **SetGID** auf einem **Verzeichnis** gesetzt, dann erben neu angelegte Dateien und Unterverzeichnisse die **Gruppe des Verzeichnisses**.

Nicht entscheidend ist dann die primäre Gruppe des Benutzers.

Spezialrechte unter Linux – SetGID II

Warum ist das wichtig?

Spezialrechte unter Linux – SetGID III

Syntax:

Symbolische Schreibweise: `# chmod g+s <verzeichnis>`

Oktale Schreibweise: `# chmod 2<_ _ _> <verzeichnis>`

Beispiel: `# chmod 2775 /projekt`

→ `2` → SetGID-Bit setzen
→ `775` → normale Rechte für Eigentümer, Gruppe und Andere

Anzeige mit ls -ld: `drwxrwsr-x`

→ Das `s` bei den Gruppenrechten zeigt das gesetzte **SetGID-Bit** an.

Spezialrechte unter Linux – SetGID IV



Bei **ausführbaren Programmen** bewirkt **SetGID**, dass das Programm mit **der Gruppe der Programmdatei** ausgeführt wird.

Das wird nur in **speziellen Fällen** genutzt, wenn ein Programm kontrolliert auf **gruppengeschützte Ressourcen** zugreifen soll.

SetGID auf Programmen sollte nur **sehr gezielt** gesetzt werden, weil dadurch **zusätzliche Rechte** vergeben werden.

Spezialrechte unter Linux – SetUID I

SetUID (s) – Programm mit Besitzerrechten ausführen

Das **SetUID-Bit** kann bei **ausführbaren Programmen** gesetzt werden.

Dann läuft das Programm nicht mit den Rechten des aufrufenden Benutzers, sondern mit den **Rechten des Dateibesitzers**.

Spezialrechte unter Linux – SetUID II

Typischer Anwendungsfall: passwd

Normale Benutzer dürfen ihr Passwort ändern. Dafür muss das Programm kontrolliert in geschützte Systemdateien wie **/etc/shadow** schreiben.

Das Programm **passwd** gehört deshalb normalerweise **root** und besitzt das **SetUID-Bit**.

Spezialrechte unter Linux – SetUID III

Beispielanzeige:

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x ...
```

→ Das `s` bei den Besitzerrechten zeigt das gesetzte **SetUID-Bit** an

Spezialrechte unter Linux – SetUID IV

Syntax:

Symbolische Schreibweise: `# chmod u+s <programm>`

Oktale Schreibweise: `# chmod 4<_ _ _> <programm>`

Beispiel:

```
# chmod 4755 <programm>
```

→ 4 → **SetUID-Bit** setzen
→ 755 → normale Rechte

Spezialrechte unter Linux – SetUID V

Wichtig:

SetUID sollte nur **sehr gezielt** verwendet werden, weil Programme dadurch mit zusätzlichen Rechten laufen können.

Spezialrechte unter Linux erkennen



Mit **ls -ld** lassen sich **alle drei Spezialrechte erkennen.**

`ls -ld` zeigt die Rechte des Verzeichnisses an (`d` für *directory*), ohne die Dateien darin aufzulisten.

Man muss nur wissen, **an welcher Stelle** sie im Rechteblock stehen:

- **Sticky Bit: 10. Zeichen** (dritte Dreiergruppe)
- **SetGID: 7. Zeichen** (zweite Dreiergruppe)
- **SetUID: 4. Zeichen** (erste Dreiergruppe)

Spezialrechte unter Linux erkennen – Sticky Bit

Beispiel (Sticky Bit):

```
$ ls -ld /tmp
drwxrwxrwt  10 root root  4096  /tmp
```

→ Das letzte Zeichen `t` zeigt: **Sticky Bit aktiv**

→ Ohne **Sticky Bit** stünde dort ein `x`

Spezialrechte unter Linux erkennen – SetGID

Beispiel (SetGID):

```
$ ls -ld /projekt
drwxr-sr-x   5 user gruppe   4096  /projekt
```

→ Das `s` an der **7. Stelle** (Gruppen-x-Position) zeigt: **SetGID aktiv**

→ Ohne **SetGID** stünde dort ein `x`

Spezialrechte unter Linux erkennen – SetUID

Beispiel (SetUID):

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x   1 root root   54256  /usr/bin/passwd
```

→ Das `s` an der **4. Stelle** (Owner-x-Position) zeigt: **SetUID aktiv**

→ Ohne **SetUID** stünde dort ein `x`

Minimale Rechte (Principle of Least Privilege) I

Benutzer und Prozesse erhalten **nur die Rechte, die sie wirklich benötigen.**

→ **Weniger Rechte = geringeres Sicherheitsrisiko**

Praxis unter Linux:

- **Admin-Zugriff nur bei Bedarf über sudo** (keine dauerhafte Root-Sitzung)
- **Keine unnötigen Gruppenmitgliedschaften** (z. B. docker, sudo, adm)
- **Restriktive Standardrechte** bei Dateien & Verzeichnissen (z. B. sinnvolle umask)
- **Ausführrechte (x) nur dort setzen, wo sie erforderlich sind**

Minimale Rechte (Principle of Least Privilege) II

Warum ist das wichtig?

- Begrenzt Schäden bei Fehlbedienung oder Kompromittierung
- Erhöht Nachvollziehbarkeit und Kontrolle
- Entspricht gängigen Sicherheitsgrundsätzen (Best Practice)



So wenig Rechte wie möglich – so viele wie nötig.

Beispiel:

Ein normaler Benutzer benötigt kein Schreibrecht in **/etc**.

Zero Trust und Principle of Least Privilege

Zero Trust – Idee:

Vertraue nichts automatisch – **jeder Zugriff** muss **immer wieder geprüft** werden.

Principle of Least Privilege (PoLP) ist ein Baustein von Zero Trust:

Zero Trust ist „größer“, es umfasst **PoLP**, aber zusätzlich das Prinzip **„ständig verifizieren“**.



PoLP = minimale Rechte

Zero Trust = minimales Vertrauen + ständige Überprüfung

Verzeichnisse

Wo liegt was? Orientierung im Linux-Dateisystem

Verzeichnisstruktur unter Linux I

In der Informatik werden Verzeichnisstrukturen häufig als **Baum** dargestellt, der üblicherweise **nach unten wächst**.

Unter Linux hat dieser Baum genau einen gemeinsamen Startpunkt:

das Root-Verzeichnis /

Darunter befinden sich weitere Verzeichnisse, die sich wiederum in Unterverzeichnisse verzweigen können.

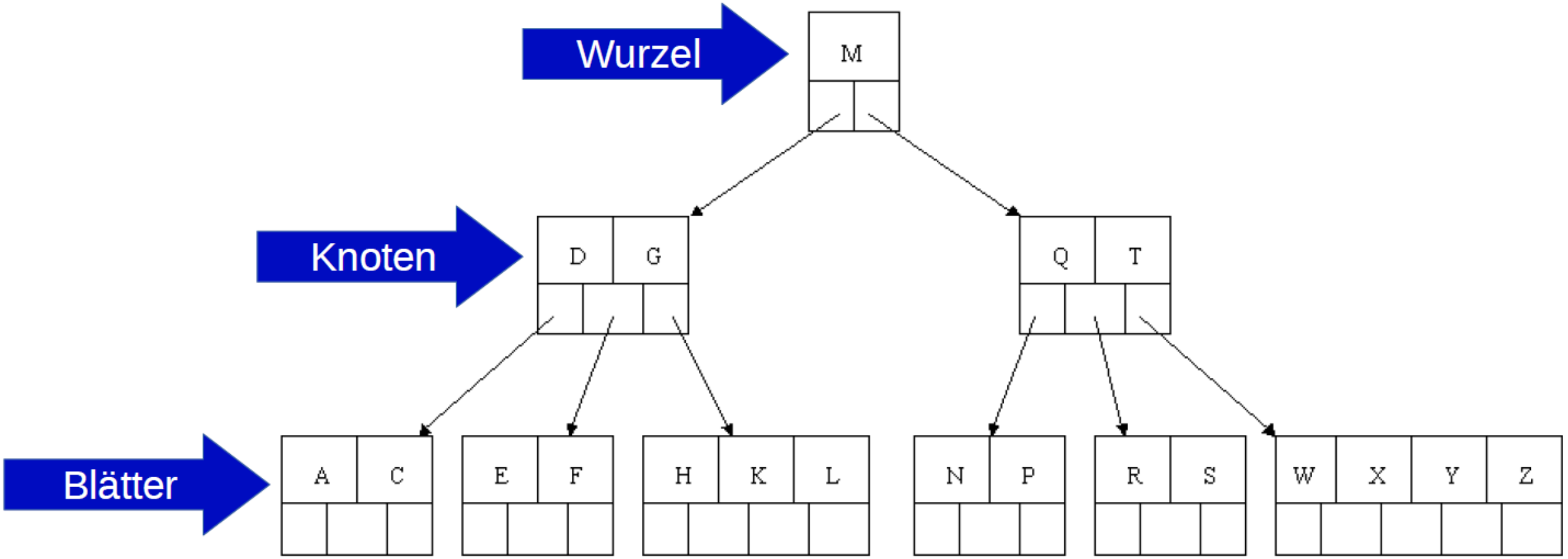
B-Bäume in Dateisystemen I

Ein **B-Baum** ist eine **Baumstruktur**, die **große Datenmengen effizient** verwaltet.

B-Bäume halten die Suchwege kurz: Alle Zweige sind **gleich lang**, und jeder Knoten enthält **mehrere sortierte Einträge** – ideal für Festplattenzugriffe.

💡 Viele **Linux-Dateisysteme** nutzen Varianten von B-Bäumen, z. B. ext4

B-Bäume in Dateisystemen II



Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Einschub: B-Baum



Der **B-Baum** wurde Anfang der 1970er-Jahre von **Rudolf Bayer** und **Edward M. McCreight** bei **Boeing Research Labs** entwickelt.

Die ursprüngliche Bedeutung des **„B“** ist nicht eindeutig überliefert. Deshalb kursieren mehrere Deutungen:

- Bayer → einer der Entwickler
- Balanced → der Baum bleibt ausgeglichen
- Boeing → dort wurde die Struktur entwickelt
- Bushy/Broad → viele Verzweigungen pro Knoten

Verzeichnisstruktur unter Linux II

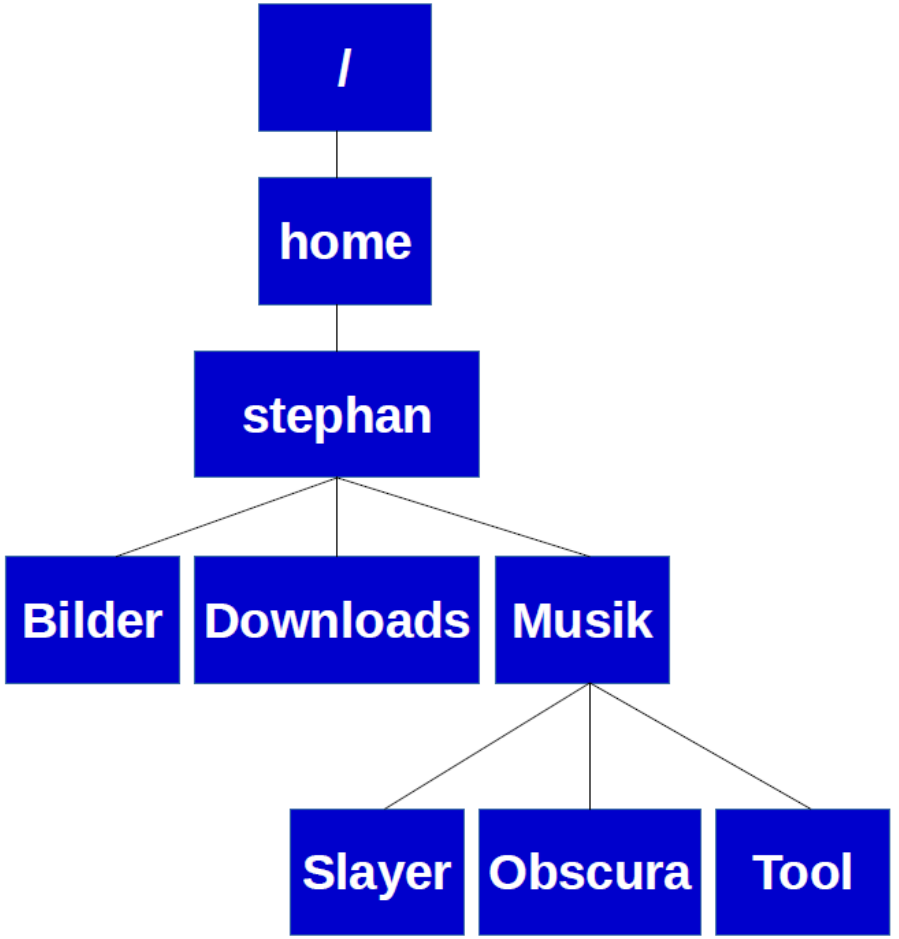
Linux vs. Windows: Wurzel des Dateisystems

- **Linux** besitzt **nur eine einzige Wurzel**, das Verzeichnis „**/**“ (gesprochen: *root* oder *slash*):
  Alle Dateien und Ordner hängen unter dieser gemeinsamen Wurzel.
- **Windows** hat **mehrere Wurzeln**:
  Jedes Laufwerk wie „**C:**“, „**D:**“ usw. bildet eine **eigene** Wurzel des Dateisystems.

Verzeichnisstruktur unter Linux III

Ein Verzeichnis besitzt **immer genau ein übergeordnetes Verzeichnis**, kann aber **beliebig viele Unterverzeichnisse** haben.

Der Weg **nach oben** (zum Elternverzeichnis) ist daher **immer eindeutig**.



Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Verzeichnisse unter Linux I

- **/bin, /sbin** – Zentrale Systembefehle und essenzielle ausführbare Programme (**/bin** → Benutzerbefehle, **/sbin** → Systemprogramme/ Administratorbefehle)
- **/boot** – Linux-Kernel sowie Dateien des Bootloaders
- **/dev** – Virtuelles Verzeichnis für Geräte-Dateien, die Hardware und Geräte im Dateisystem abbilden
- **/etc** – Systemweite Konfigurationsdateien
- **/home** – Benutzerverzeichnisse mit persönlichen Dateien und Einstellungen
- **/lib** – Wichtige Bibliotheken für Systemprogramme und Kernel-Module

/etc

💡 **/etc** steht in Linux/Unix historisch für **„et cetera“** (= *„und so weiter“*).

Gemeint war ursprünglich: *„alles, was sonst noch systemweit gebraucht wird“* – heute ist **/etc** ganz klar der Ort für **systemweite Konfigurationsdateien** (host-spezifische Einstellungen), **nicht** für Programme oder Nutzerdaten.

Verzeichnisse unter Linux II

- **/media** – Standard-Einhängepunkt für externe Datenträger (z. B. USB-Sticks, CD/DVD)
- **/mnt** – Temporärer Mountpunkt für manuell eingehängte Datenträger
- **/opt** – Optional installierte zusätzliche Softwarepakete
- **/root** – Persönliches Home-Verzeichnis des Systemadministrators
- **/tmp** – Temporäre Dateien; Inhalte werden beim Neustart in der Regel gelöscht
- **/run** – Temporäres Verzeichnis für Laufzeitdaten seit dem letzten Start; enthält Informationen von Diensten (z. B. PIDs, Sockets)
- **/srv** – Daten, die von Diensten bereitgestellt oder gehostet werden (z. B. Web- oder FTP-Daten)

Verzeichnisse unter Linux III

- **/usr** – Systemweit installierte Benutzerprogramme, Bibliotheken und Ressourcen
  - **/usr/bin** – Nicht-essenzielle ausführbare Programme
  - **/usr/lib** – Zugehörige Bibliotheken für Programme unter **/usr**
- **/var** – Speicherort für dynamische, sich im Betrieb ändernde Daten
  - **/var/cache** – Zwischenspeicher (Cache) für Anwendungen
  - **/var/log** – Logdateien
- **/proc** – Virtuelles Dateisystem mit Informationen zu laufenden Prozessen und Systemzustand
- **/sys** – Virtuelles Dateisystem mit Kernel- und Hardware-Schnittstellen; dient dazu, Geräte und Kernelfunktionen zur Laufzeit auszulesen oder zu konfigurieren

/usr

/usr steht historisch für **„user“** – aber **nicht** im Sinne von „hier liegen die User-Dateien (Home-Verzeichnisse)“.

Der Name kommt daher, dass **/usr** in frühen Unix-Systemen ursprünglich oft als **separates Dateisystem für Benutzerverzeichnisse** genutzt wurde (quasi „*user space*“ auf einer eigenen Platte).

Später wanderte das meiste davon nach **/home**, und **/usr** wurde zum Ort für **zusätzliche Systemprogramme und -ressourcen**.



„*Unix System Resources*“ (oder „*Universal System Resources*“) ist ein nachträgliches **Backronym.**

/usr vs. /bin & /sbin I

/usr – systemweite Software für den Alltag

- Enthält **Programme, Bibliotheken und gemeinsame Ressourcen** (z. B. **/usr/bin**, **/usr/lib**, **/usr/share**)
- Meist **durch die Distribution / Paketverwaltung installiert** (z. B. apt, dnf, pacman)
- Wichtig: **Nicht „pro Benutzer“**, sondern **systemweit** („*Userland*“)

/bin & /sbin – historisch: das Nötigste zum Starten

- Dienten früher als Orte für **essenzielle Befehle**, die man **früh beim Booten** oder im **Rettungsmodus** braucht
  Beispiele: sh, mount, ip, fsck
- **/sbin** war dabei oft für **Admin-/Systembefehle** (Root) gedacht

/usr vs. /bin & /sbin II

usrmerge (Zusammenführung nach /usr):

- Auf vielen modernen Linux-Systemen sind **/bin** → **/usr/bin** und **/sbin** → **/usr/sbin** nur noch **Symlinks** (symbolische Links)
- Folge: Inhaltlich gibt es oft **keinen echten Unterschied mehr**, weil alles unter **/usr/** zusammengeführt ist

/usr vs. /bin & /sbin III

→ **/usr = *Userland*** (normale System-Software)

→ **/bin & /sbin = historisch** essenzielle Tools fürs Booten / Rettung

💡 **Schnelltest im Terminal:**

`$ ls -ld /bin /sbin` → zeigt, ob die Pfade symbolische Links sind

Verzeichnisse unter Linux IV



Filesystem Hierarchy Standard (FHS):

- Die Linux-Verzeichnisstruktur orientiert sich am **FHS** (von den meisten Distributionen umgesetzt)
- Der FHS legt fest, welche Standardverzeichnisse es gibt und wofür sie vorgesehen sind
- **Vorteil:** einheitlicher Aufbau → Systeme sind leichter zu administrieren und zu verstehen

Virtuelle Verzeichnisse unter Linux I

Die Verzeichnisse **/dev**, **/proc** und **/sys** werden beim Systemstart **vom Linux-Kernel dynamisch erzeugt**.

Sie sind **virtuelle Verzeichnisse** – das heißt, sie liegen **nicht auf der Festplatte**, sondern werden **im Arbeitsspeicher** bereitgestellt und spiegeln aktuelle System- und Hardwareinformationen wider.

Virtuelle Verzeichnisse unter Linux II

Unter Linux gilt: Fast alles wird wie eine Datei behandelt.

Deshalb kann man auch auf **virtuelle Verzeichnisse** mit üblichen Befehlen wie z. B. **cat** oder **ls** zugreifen.

💡 **Fast alles sieht wie eine Datei aus** – aber dynamische Systeminformationen, z. B. zu Prozessen, Geräten oder Netzwerkverbindungen, sind **keine echten Dateien auf der Festplatte**.

Virtuelle Verzeichnisse unter Linux III

Auch die „Dateien“ in virtuellen Verzeichnissen wie **/proc** lassen sich mit normalen Linux-Werkzeugen anzeigen oder bearbeiten – obwohl sie **nicht als echte Dateien auf der Festplatte gespeichert** sind.

Sie werden vom **Kernel dynamisch bereitgestellt** und zeigen aktuelle Systeminformationen oder Kernel-Einstellungen.

Linux stellt sie dateiähnlich dar. Deshalb funktionieren bekannte Werkzeuge wie **cat**, **less** oder **ls**.

💡 Nicht alle virtuellen Dateien können verändert werden. Viele sind nur lesbar; Änderungen benötigen häufig Administratorrechte und gelten oft nur bis zum nächsten Neustart.

Virtuelle Verzeichnisse unter Linux IV

Beispiele:

- `$ cat /proc/cpuinfo` – zeigt Informationen zur CPU an
- `$ cat /proc/sys/net/ipv4/ip_forward` – zeigt, ob IPv4-Forwarding aktiviert ist
- `$ echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward` – aktiviert **IPv4-Forwarding** temporär



tee ist ein Linux-Befehl, der Eingaben **gleichzeitig ausgibt und in eine Datei schreibt**. Der Name kommt sinngemäß vom **T-Stück** aus der Rohrleitungstechnik: Eine Eingabe wird in zwei Richtungen aufgeteilt.

Virtuelle Verzeichnisse unter Linux V

→ Ein **virtuelles Verzeichnis** enthält keine normalen gespeicherten Dateien. Stattdessen stellt das System aktuelle Informationen **dynamisch als Dateisystem sichtbar** dar.

Beispiel: In **/proc** sieht man Informationen über laufende Prozesse. Diese Dateien liegen nicht wirklich so auf der Festplatte, sondern werden vom System erzeugt, wenn man sie abfragt.

💡 **/run** ist anders: Dort werden **echte temporäre Dateien** von Programmen und Diensten abgelegt, z. B. PID-Dateien oder Sockets. Diese Dateien sind zwar nur seit dem Systemstart gültig und liegen oft im RAM, aber sie sind trotzdem normale Laufzeitdateien.

Virtuelle Verzeichnisse unter Linux VI

„Virtuell“ bedeutet nicht einfach nur: liegt im RAM

Sondern: Das Verzeichnis zeigt Informationen als Dateien an, die so nicht als normale Dateien gespeichert sind.

Bildlich:

- **/proc** und **/sys** sind wie ein Live-Statusfenster des Systems
- **/run** ist eher wie ein temporärer Ablageordner für laufende Programme

Die Verzeichniszeiger . und ..

Jedes Verzeichnis enthält automatisch zwei spezielle Einträge:

. → verweist auf das **aktuelle Verzeichnis**

.. → verweist auf das **übergeordnete Verzeichnis**

Diese Einträge sind **fester Bestandteil der Linux-Verzeichnisstruktur** und werden z. B. für **relative Pfade** genutzt.

Versteckte Dateien und Verzeichnisse I

In Unix/Linux gelten alle Dateien und Ordner als versteckt, deren Name mit einem Punkt **.** beginnt.

Beispiele:

- **.config**
- **.ssh**
- **.bashrc**

Versteckte Dateien anzeigen: `$ ls -la`

→ Zeigt alle Dateien an – inklusive der versteckten

Versteckte Dateien und Verzeichnisse II

```
[root@pxesrv srv]# touch Binda
[root@pxesrv srv]# ls
insgesamt 0
-rw-r--r--. 1 root    root      0  2. Nov 23:10 Binda
drwxr-xr-x. 6 stephan stephan 178 26. Jul 2020  tftp
[root@pxesrv srv]# mv Binda .Binweg
[root@pxesrv srv]# ls
insgesamt 0
drwxr-xr-x. 6 stephan stephan 178 26. Jul 2020  tftp
[root@pxesrv srv]# ls -la
insgesamt 4
drwxr-xr-x.  3 root    root     33  2. Nov 23:10 .
dr-xr-xr-x. 17 root    root   4096 15. Aug 15:09 ..
-rw-r--r--.  1 root    root      0  2. Nov 23:10 .Binweg
drwxr-xr-x.  6 stephan stephan  178 26. Jul 2020  tftp
[root@pxesrv srv]#
```

Quiz I

```
$ echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
```

– warum verwendet man hier nicht einfach:

```
$ sudo echo 1 > /proc/sys/net/ipv4/ip_forward ?
```

Virtuelle Verzeichnisse unter Linux V

```
root@debian-openvpn ~ # cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel Xeon Processor (Skylake, IBRS)
stepping        : 4
microcode       : 0x1
cpu MHz         : 2100.000
cache size      : 16384 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 syscall nx pdpe1
a cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invp
d rtm mpx avx512f rdseed adx smap clwb avx512cd xsaveopt xsavec xgetbv1 arat
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf
bogomips        : 4200.00
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
```

Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Virtuelle Verzeichnisse unter Linux VI

Zum Ausprobieren: Erkunden Sie Ihr System mit einfachen Befehlen

Mit den folgenden Befehlen können Sie testen, welche Informationen Linux über **/proc** und **/dev** bereitstellt.

Dabei sehen Sie, dass sich diese Einträge wie normale Dateien / Verzeichnisse nutzen lassen – obwohl sie **dynamisch vom Kernel** (bzw. vom System) bereitgestellt werden:

`$ cat /proc/version` – Kernel-Version

`$ cat /proc/meminfo` – Speicherinformationen

`$ ls -l /dev` – Gerätedateien

Quiz II

Ordnen Sie die Beschreibung dem passenden Verzeichnis zu:

/home, **/etc**, **/var/log**, **/proc**

a) Systemweite Konfigurationsdateien
b) Persönliche Dateien und Einstellungen der Benutzer
c) Protokolldateien, die im Betrieb fortlaufend geschrieben werden
d) Virtuelles Verzeichnis mit Informationen zu laufenden Prozessen

Quiz III

Stimmt oder stimmt nicht?

„**/dev** enthält echte Dateien, die auf der Festplatte gespeichert sind.“

Quiz IV

Welches Verzeichnis enthält systemkritische Befehle für Administratoren?

a) **/bin**
b) **/sbin**
c) **/opt**
d) **/tmp**

Quiz V

Welches dieser Verzeichnisse wird beim Systemstart vom Kernel erzeugt?

a) **/boot**
b) **/usr/bin**
c) **/proc**
d) **/home**

Quiz VI

„**/var** enthält _____ Daten, die beim Systembetrieb entstehen oder wachsen.“

Quiz VII

„In **/etc** liegen die _____ Konfigurationsdateien des Systems.“

Quiz VIII

„Das Verzeichnis **/proc** enthält keine echten Dateien, sondern zeigt Informationen an, die der Kernel _____ erzeugt.“

Quiz IX

Wofür steht „**bin**“ in **/bin**?

a) Binary
b) Build Information
c) Binder
d) Backup-Index-Node

Quiz X

Was ist der Unterschied zwischen **/bin** und **/sbin** (historisch betrachtet)?

a) **/bin** für Logdateien, **/sbin** für Konfiguration
b) **/bin** für Bibliotheken, **/sbin** für Dokumentation
c) **/bin** nur für Kernel-Module, **/sbin** nur für Treiber
d) **/bin** für Benutzerprogramme, **/sbin** für System-/Admin-Programme

Quiz XI

Welche Aussage trifft auf viele moderne Linux-Systeme zu?

a) „**/bin** und **/sbin** sind eigene physische Verzeichnisse und dürfen nicht unter **/usr** liegen, da sie für den Bootprozess zwingend getrennt sein müssen“

b) „**/bin** und **/sbin** sind auf vielen Systemen **Symlinks** und verweisen auf **/usr/bin** bzw**. /usr/sbin** (Stichwort **usrmerge**)“

c) „**/bin** wird heute hauptsächlich für Kernel-Module genutzt, während **/usr/bin** nur Benutzer-Skripte enthält“

d) „**/bin** ist ein virtuelles Dateisystem ähnlich **/proc**, dessen Inhalte ausschließlich zur Laufzeit vom Kernel erzeugt werden“

Relative und absolute Pfade

Zwei Wege zum Ziel im Linux-Dateisystem

Relative und absolute Pfade I

Relative Pfade:

- Relative Pfade beziehen sich auf das **aktuelle Verzeichnis** (Arbeitsverzeichnis)
- `ls -l` ohne Pfadangabe listet den Inhalt **des aktuellen Verzeichnisses** auf
- Je nach **aktueller Position** kann derselbe relative Pfad **auf ein anderes Ziel zeigen** und dadurch unterschiedliche Ergebnisse liefern

Beispiel:

`$ ls -l ../../` listet den Inhalt des Verzeichnisses **zwei Ebenen oberhalb** der **aktuellen Position** auf

→ Das Ergebnis hängt davon ab, **wo** der Befehl ausgeführt wird

Relative und absolute Pfade II

Absolute Pfade:

- Beginnen immer an der **Wurzel** (**/**) und beschreiben den **vollständigen Pfad** zu einem Ziel im Dateisystem
- Das Ziel ist **unabhängig vom aktuellen Verzeichnis eindeutig**

Beispiel:

```
$ ls -l /home/stephan/Musik/Obscura
```

→ Liefert immer den Inhalt dieses Verzeichnisses – unabhängig davon, von welchem Ort der Befehl ausgeführt wird

Relative und absolute Pfade – pwd

Der Befehl pwd (*print working directory*) gibt den vollständigen (absoluten) Pfad des derzeitigen Arbeitsverzeichnisses aus.

```
root@debian-openvpn /home/stephan/geheim # pwd
/home/stephan/geheim
```

Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Relative und absolute Pfade – Tilde

Die **Tilde** (**~**) ist eine Abkürzung für das **Home-Verzeichnis des aktuell angemeldeten Benutzers.** Sie zeigt also immer auf das persönliche Startverzeichnis.

Beispiel:

`$ ls -l ~` entspricht `$ ls -l /home/stephan`

Für den Benutzer **Root**:

`# ls -l ~` entspricht `# ls -l /root`



Das Home-Verzeichnis von **Root** ist standardmäßig **/root** (nicht /home/root). Root nutzt ein eigenes Home-Verzeichnis außerhalb von **/home**, da es sich um das **System-Administratorkonto** handelt.

Weitere nützliche Linux-Tools

pipe, find & locate

Weitere nützliche Tools für die Bash – pipe I

Die Zeichen **>** und **>>** kennen wir bereits als **Umleitungsoperatoren**:
Sie leiten die Ausgabe eines Befehls in eine Datei um.

Das Pipe-Symbol (|) funktioniert anders:
Es leitet die Ausgabe eines Befehls als Eingabe an den nächsten Befehl weiter.
Beispiel:

```
$ befehl1 | befehl2
$ ls -l | less
```

Deutscher IT-Begriff: **Verkettungszeichen** / **Pipe**

💡 Bei **>** und **>>** geht es um **Dateiumleitung**, bei **|** um die **Verkettung von Befehlen**.

pipe II

Beispiel: `$ dmesg | less`

→ Die Ausgabe von **dmesg** wird seitenweise in **less** angezeigt und lässt sich bequem durchsuchen und scrollen.

Ohne Pipe wäre derselbe Ablauf umständlicher:

```
$ dmesg > tempfile
$ less tempfile
```

Da `dmesg` **etwas ausgibt** und `less` **Eingaben liest**, kann die Ausgabe von `dmesg` direkt an `less` weitergegeben werden – genau das erledigt die Pipe.



dmesg steht für *display messages*: Der Befehl zeigt Kernel-Meldungen aus dem Kernel-Ringpuffer (*kernel log buffer*) an.

pipe III

Mit der **Pipe** lassen sich **mehrere Befehle** zu einer **Verarbeitungskette** verbinden.

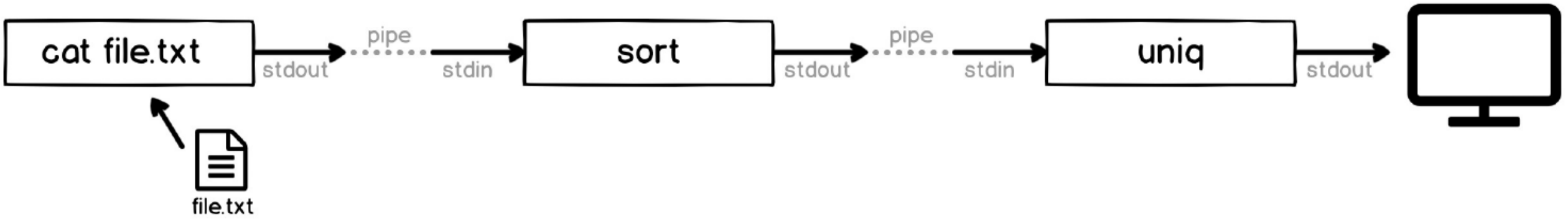
Die Ausgabe eines Befehls wird dabei direkt als Eingabe an den nächsten Befehl weitergegeben:

```
$ befehl1 | befehl2 | befehl3
```

So entsteht eine **Befehlskette**, bei der jeder Befehl das Ergebnis des vorherigen Befehls weiterverarbeitet.

pipe IV

→ **Pipes** verbinden die Ausgabe eines Befehls mit der Eingabe des nächsten – dadurch entstehen **flexible Kommando-Ketten.**



Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

pipe V – Effizientes Arbeiten durch Befehlsverkettung

Durch **Pipes** lassen sich die Features verschiedener Befehle kombinieren – das spart Tipparbeit und macht Abläufe deutlich effizienter.

Beispiele:

Anzahl der Dateien im aktuellen Verzeichnis anzeigen:

```
$ ls | wc -l
```

Eine große Datei alphabetisch sortiert anzeigen und bequem scrollen:

```
$ cat datei | sort -f | less
```

Nur Zeilen mit einem bestimmten Begriff anzeigen:

```
$ cat /etc/passwd | grep sp
```

Suchen und Finden mit find I

Der Befehl **find** durchsucht das Dateisystem ab einem definierten **Startpunkt** nach Dateien oder Verzeichnissen.

Syntax: `find <startpunkt> <optionen/kriterien>`

Beispiel:

```
$ find /etc -name interfaces
```

→ Datei namens **interfaces** in **/etc** suchen

```
$ find /etc -type f -name "*.conf“
```

→ Sucht ab **/etc** nach regulären Dateien mit der Endung **.conf**

💡 Eine **reguläre Datei** ist eine „normale“ Datei, die Inhalte speichert, z. B. Text, Konfigurationsdaten, Bilder oder Programme — im Gegensatz zu Verzeichnissen oder speziellen Gerätedateien.

Suchen und Finden mit find II

find kann nicht nur nach Dateinamen suchen, sondern auch nach **anderen Eigenschaften**, z. B.:

- Dateigröße
- Dateityp (Datei, Verzeichnis, Symlink)
- Besitzer / Gruppe
- Änderungszeitpunkte
- Berechtigungen

Wichtige Suchkriterien für find I

Kriterium	Bedeutung	Beispiel
`-name <muster>`	Sucht nach Dateien oder Verzeichnissen mit einem bestimmten Namen. Groß-/Kleinschreibung wird beachtet.	`find /etc -name interfaces`
`-iname <muster>`	Wie **-name**, aber Groß-/Kleinschreibung wird ignoriert (**i** für *ignore case*).	`find /etc -iname InTeRfAcEs`
`-type f`	Sucht nach regulären Dateien.	`find /etc -type f`
`-type d`	Sucht nach Verzeichnissen.	`find /home -type d`
`-type l`	Sucht nach symbolischen Links.	`find /usr/bin -type l`

Wichtige Suchkriterien für find II

Kriterium	Bedeutung	Beispiel
`-size <größe>`	Sucht nach Dateien mit bestimmter Größe (+ **größer als**, - **kleiner als** und ohne Vorzeichen **ungefähr bzw. genau**).	`find /var -size +100M` `find /var -size 1G`
`-user <benutzer>`	Sucht nach Dateien, die einem bestimmten Benutzer gehören.	`find /home -user max`
`-group <gruppe>`	Sucht nach Dateien, die einer bestimmten Gruppe gehören.	`find /srv -group projekt`
`-mtime <tage>` `-atime <tage>`	Sucht nach Dateien anhand des Änderungszeitpunkts / letzten Zugriffszeitpunkts (**m** für *modification*, **a** für *access*).	`find /var/log -mtime -7` `find /home -atime +30`
`-perm <rechte>`	Sucht nach Dateien mit bestimmten Berechtigungen.	`find /home -perm 644`

Schneller suchen mit locate I

Der Befehl **locate** ist deutlich bequemer und schneller zu benutzen als **find** – allerdings sucht er **ausschließlich nach Dateinamen**.

locate arbeitet extrem schnell, weil die Suche über eine **vorgenerierte Datenbank** erfolgt.

Syntax: `$ locate [optionen] <suchstring>`

Beispiel: `$ locate resolv.conf`

locate II

locate sucht nicht live im Dateisystem, sondern nutzt eine **vorbereitete Datenbank**.

Damit die Suchergebnisse aktuell bleiben, muss diese Datenbank regelmäßig aktualisiert werden.

Hinweise:

- Das Aktualisieren erfordert **Root-Rechte**
- Viele Systeme aktualisieren die Datenbank **automatisch per Cron-Job** (z. B. einmal täglich)
- Nach dem Aktualisieren findet **locate** auch neu angelegte oder verschobene Dateien

Einschub: Cron-Job

Ein **Cron-Job** ist ein zeitbasierter Dienst, der Aufgaben automatisch zu festgelegten Zeiten ausführt.

💡 **Cron** steht nicht für eine technische Abkürzung, sondern wird von ***Chronos*** abgeleitet — also dem griechischen Begriff bzw. der Figur für **Zeit**.

locate III

Wenn locate nichts findet …

kommen dafür also zwei Ursachen infrage:

- **?**
- **?**

Lernfolien Modul 3

Das Wichtigste auf einen Blick