

Linux

Modul 2: Grundlagen II

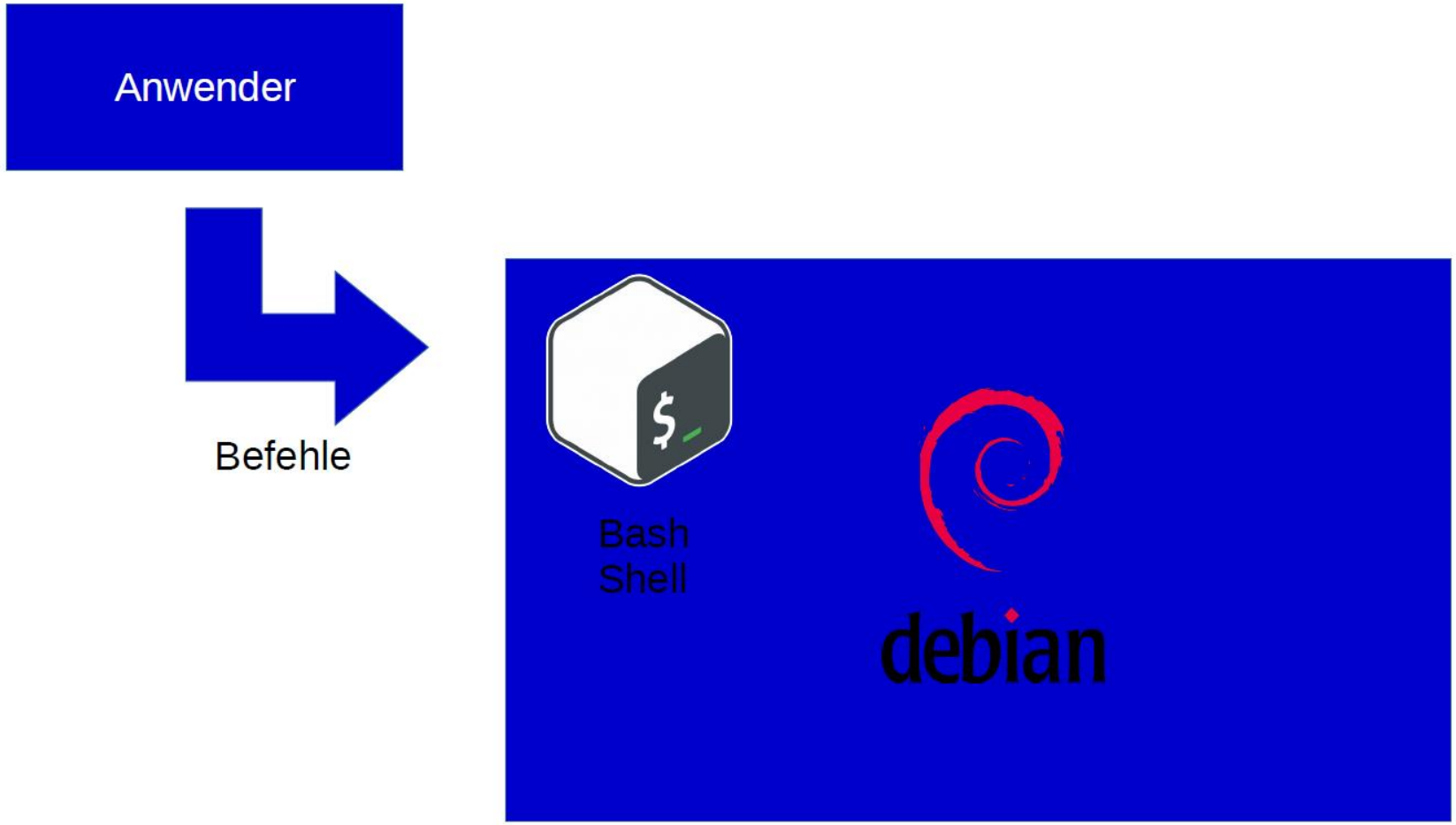
Übersicht

- Die Shell
- Shell-Befehle
- Tipps für die Arbeit mit der Bash-Shell
- Texteditoren
- SSH
- **Lernfolien Modul 2:** Das Wichtigste auf einen Blick
- **Praxis:** Gelerntes direkt anwenden – wir arbeiten live in der Shell
- **Praxis:** SSH in Aktion – Praxis-Experiment

Die Shell

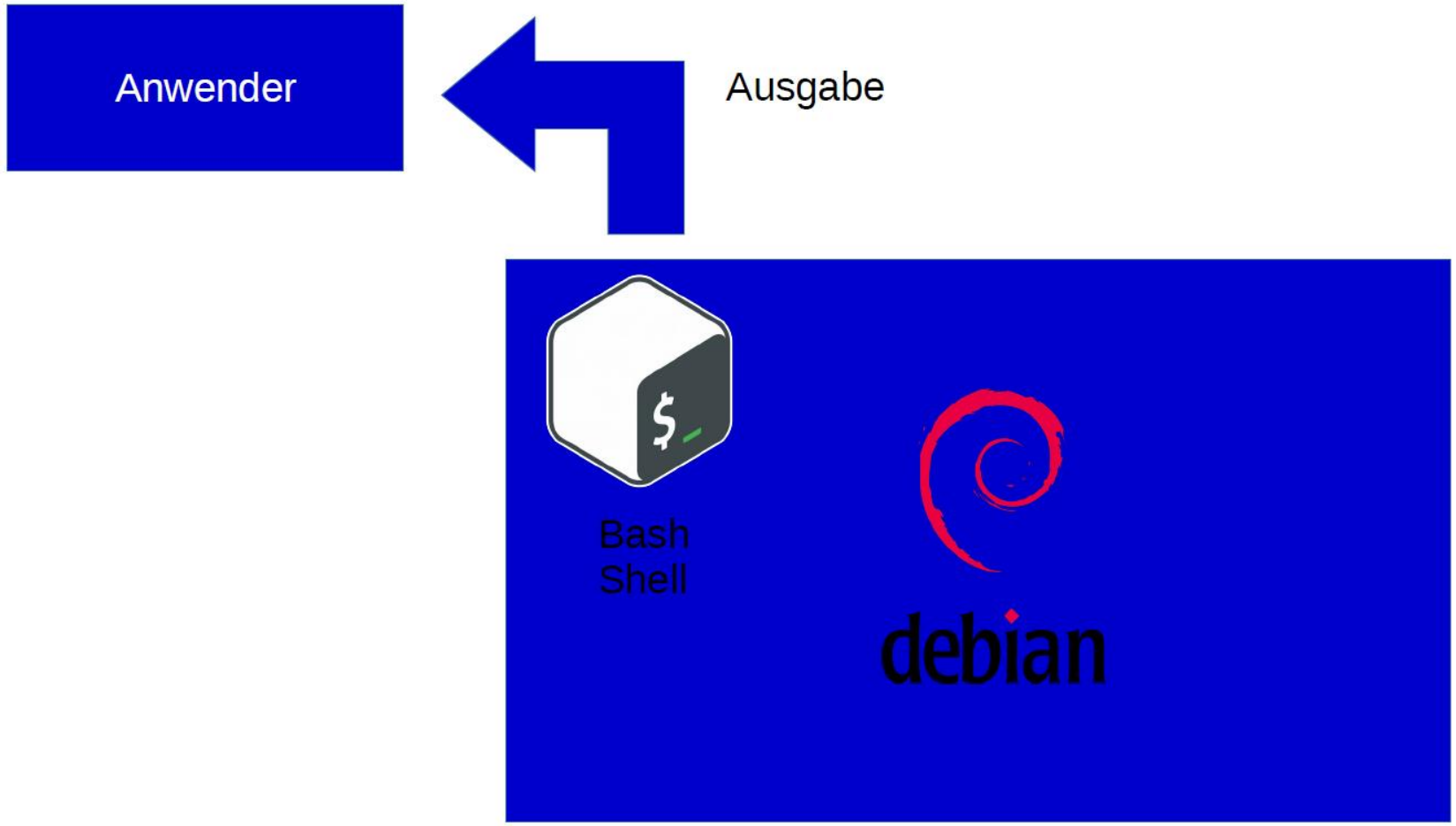
Schnittstelle zwischen Benutzer und Betriebssystem

Die Shell als Schnittstelle zum Betriebssystem I



Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Die Shell als Schnittstelle zum Betriebssystem II



Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Die Shell I

- Die **Shell** ist ein Programm, das **Befehle entgegennimmt, interpretiert und ausführt**.
- Sie bildet die **Schnittstelle zwischen Benutzer und Betriebssystem**.
- Sie ist ein **zentrales Arbeitswerkzeug** unter Linux.



„Shell“ bedeutet **„Hülle“** – der Begriff beschreibt bildlich eine Schicht um den Betriebssystemkern (**Kernel**), über die Benutzer mit dem System interagieren.

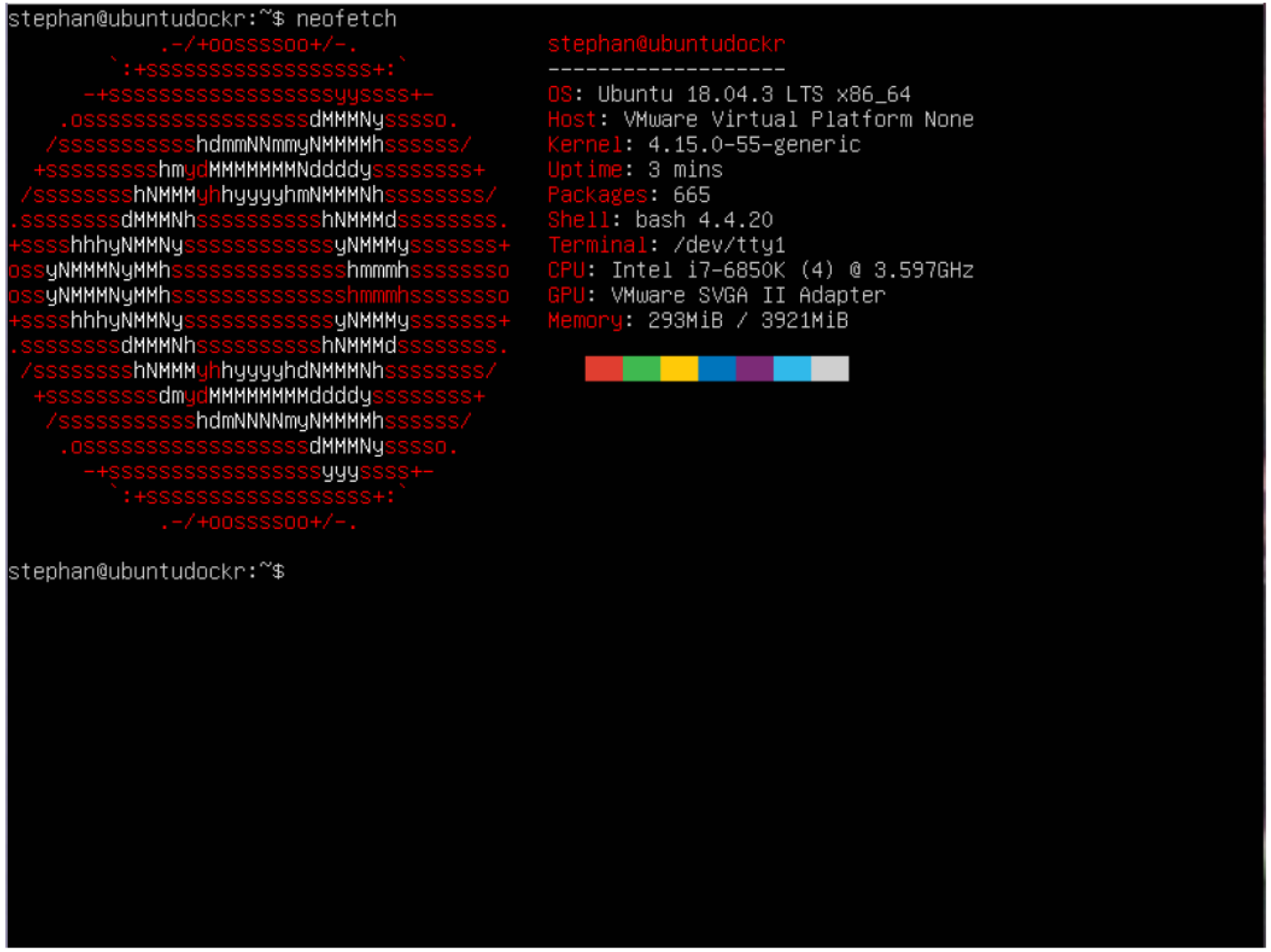
Die Shell II

Die **Shell** wird über **Befehle** bedient.

Keine Maus, keine grafische Oberfläche – nur Text.

Auch die **Ausgabe** der Shell erfolgt in **Textform**.

Die Shell III

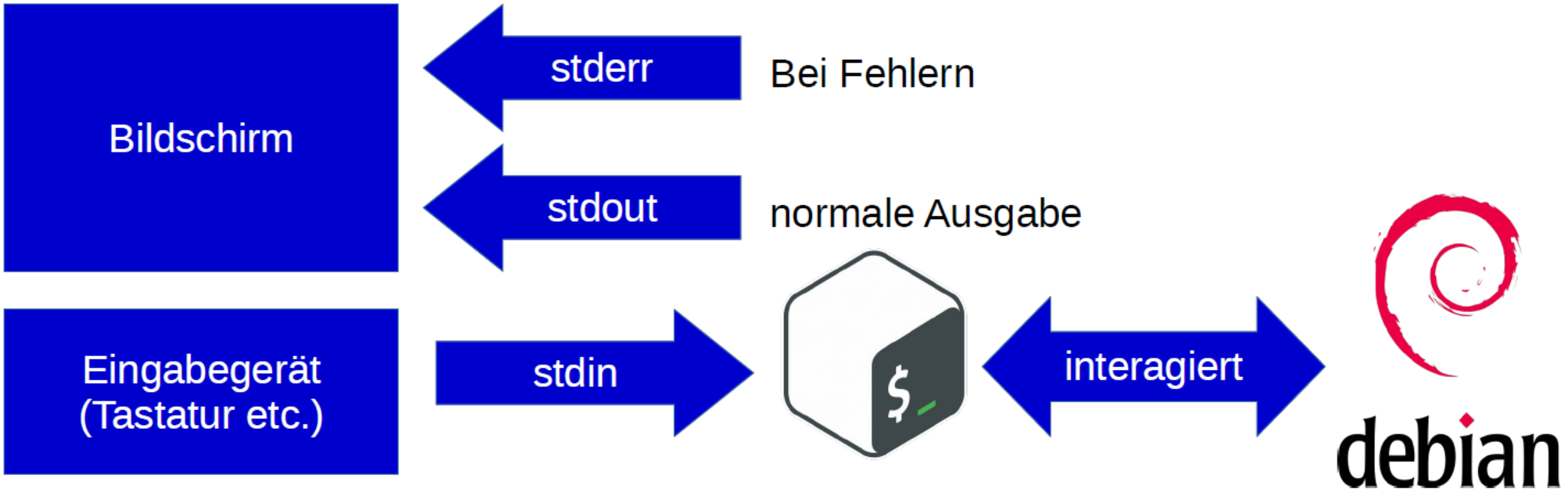


Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Die **Bash-Shell** unter Ubuntu Linux (Ausgabe des Befehls **neofetch**)

💡 **neofetch** zeigt Systeminformationen an. Seit 2024 wird das Programm nicht mehr weiterentwickelt – ein aktiv weiterentwickelter Nachfolger ist **fastfetch**.

Eingabe und Ausgabe



Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Die Shell IV

Es gibt verschiedene Shells, zum Beispiel **Bash**, **ksh**, **zsh** oder **tcsh**.

Die am häufigsten genutzte ist die **Bash (Bourne Again Shell)**, die auch standardmäßig in Debian verwendet wird.



https://bashlogo.com/

Die Bash I

Bash – Bourne Again Shell:

- **Name:** Wortspiel zur ursprünglichen **Bourne Shell** (**sh**) von Stephen Bourne
- **„Bourne Again“** spielt darauf an, dass **Bash** eine erweiterte Version der alten Bourne Shell ist
- **Besonderheit:** Standard-Shell in vielen Linux-Distributionen (z. B. Debian), unterstützt Skripte der Bourne Shell und hat viele neue Features, zum Beispiel:
  - Merkt sich die zuletzt eingegebenen Befehle (**Pfeil-hoch-Taste**)
  - Kann Dateinamen, Verzeichnisse oder Befehle automatisch vervollständigen (**Tab**)

Die Bash II

💡 **Stephen Richard Bourne** ist ein britischer Informatiker, der die ursprüngliche **Bourne Shell** (**sh**) entwickelt hat – die frühe und einflussreiche Kommandozeilen-shell für Unix.



Bildquelle: Dmitry Azovtsev, *Steve Bourne at SDWest2005.hires.jpg*, Wikimedia Commons, CC BY-SA 3.0.

Korn Shell

ksh – Korn Shell:

- **Name:** Benannt nach ihrem Entwickler **David Korn**, der die Shell 1983 bei den Bell Labs entwickelte
- Entwickelt, um die Bourne Shell zu verbessern, z. B. durch erweiterte Skript-Funktionen und interaktive Nutzung
- **Besonderheit:** Kombination aus Benutzerfreundlichkeit für interaktive Nutzung und mächtiger Skriptbarkeit

Z Shell

zsh – Z Shell:

- **Name: Paul** Ein Student der Princeton University entwickelte die Shell 1990 und benannte sie nach dem Login-Namen **zsh** des dortigen wissenschaftlichen Assistenten **Zhong Shao**
- **Besonderheit:** Sehr flexibel, viele Erweiterungen, mächtige Autovervollständigung, Themes und Plugins – besonders beliebt bei Entwicklern

TENEX C Shell

tcsh – TENEX C Shell:

- **Name:** Erweiterung der **C Shell** (**csh**), deren Syntax teilweise an die Programmiersprache C angelehnt ist
- Das **„T“** steht für TENEX, ein früheres Betriebssystem, das einige Features (z. B. Command-Line Completion) inspirierte
- **Besonderheit:** Interaktive Klassiker-Shell – heute selten, aber prägend für viele Komfortfunktionen



Eine Shell kann auch als **Skriptsprache** verwendet werden. **tcsh/csh** nutzen dafür eine teilweise an **C** angelehnte Syntax, beispielsweise für Bedingungen.

Shell-Befehle

Die wichtigsten Befehle für den Einstieg

Shell-Prompt und Befehle – Syntax I

→ Die folgenden Folien enthalten Befehle zur direkten Eingabe in der **Bash**.

Jeder dieser Befehle beginnt mit einem „$“, erkennbar auch an der `speziellen Schriftart`.

💡 **Shell-Prompt (Prompt-Symbol):**

- Das $ ist **kein Teil des Befehls**
- Es zeigt an, dass die **Shell bereit für Eingaben** ist
- $ steht für einen **normalen Benutzer**
- # steht für den **Root-Benutzer**

Shell-Prompt und Befehle – Syntax II

```
$ Hier steht ein Befehl
```

Hier nicht

Text ausgeben & umleiten I

`$ echo` gibt eine **Zeile Text** auf dem Bildschirm aus.

💡 **Hintergrund:**
echo gehört zu den ältesten Unix-Befehlen und wurde bereits in den **1970er-Jahren** verwendet, um **Texte und Meldungen auszugeben** – besonders praktisch für Hinweise und die Fehlersuche in Shell-Skripten.

Text ausgeben & umleiten II

Mit den **Umleitungsoperatoren** `>` und `>>` kann die **Standardausgabe** eines Befehls **umgeleitet** werden (z. B. in eine Datei):

- `>` **überschreibt** den Inhalt der Zieldatei
- `>>` **hängt** die Ausgabe an das Dateiende **an**

Beispiel:

```
$ echo "Hallo" > ausgabe.txt
$ echo "Noch eine Zeile" >> ausgabe.txt
```



Mit `>` oder `>>` wird die Standardausgabe umgeleitet, sie erscheint dann **nicht mehr auf dem Bildschirm**, sondern wird in das angegebene Ziel geschrieben.

ls – Inhalt eines Verzeichnisses anzeigen I

`ls` listet die Dateien und Ordner im **aktuellen Verzeichnis** auf.

Der Befehl kann noch viel mehr – dazu später Details.

Syntax: `$ ls [pfad_zum_verzeichnis]`



`[…]` **eckige Klammern = optional** – kann weggelassen werden

`<…>` **spitze Klammern = notwendig** – Platzhalter muss ersetzt werden durch echten Wert

ls – Inhalt eines Verzeichnisses anzeigen II

```
$ ls
$ ls .
```

→ Listet den Inhalt des **aktuellen Verzeichnisses** auf



`ls` und `ls .` sind **gleichbedeutend**. Der **Punkt** `.` steht in Linux für das **aktuelle Verzeichnis**. `ls` ist nur eine **Abkürzung**.

ls – Inhalt eines Verzeichnisses anzeigen III

```
$ ls /etc
```

→ Zeigt **Systemkonfigurationsdateien und -verzeichnisse** an

```
$ ls /usr/bin
```

→ Listet **ausführbare Programme** auf, darunter **viele klassische Linux-Befehle**

```
$ ls /proc
```

→ Listet **laufende Prozesse und Einträge zu Systeminformationen** auf

ls – Inhalt eines Verzeichnisses anzeigen IV



Hintergrund:

- Der Befehl ls (kurz für *list*) gehört zu den **ältesten Unix-Befehlen** und wurde bereits **1971** im ersten Unix-Handbuch dokumentiert
- Ein wichtiger Grundgedanke der **Unix-Philosophie**: *„Mach ein kleines Programm, das **eine Sache gut** macht.“*
- `ls` sollte genau das tun:
  Dateien und Verzeichnisse auflisten – einfach und ohne unnötige Zusatzfunktionen

cat – Inhalt von Dateien anzeigen I

`cat` zeigt den **Inhalt einer Textdatei** an.

Syntax:

```
$ cat <pfad_zur_datei>
```



cat – Inhalt von Dateien anzeigen II

```
$ cat /proc/cpuinfo
```

→ Zeigt Informationen über den **Prozessor (CPU)** an

```
$ cat /etc/debian_version
```

→ Zeigt die installierte **Debian-Version** an

```
$ cat /etc/passwd
```

→ Zeigt Informationen über **Benutzerkonten** an

```
$ cat /proc/meminfo
```

→ Zeigt Informationen über den **Arbeitsspeicher (RAM)** und **Swap** an

cat – Inhalt von Dateien anzeigen III



Hintergrund:

- `cat` = *concatenate* (engl. „hintereinanderhängen“)
- Schon im **Unix der 1970er-Jahre** gab es `cat`, weil man oft **Textdateien zusammenfügen oder ausgeben** wollte
- Viele kennen `cat` nur zum Anzeigen (`cat file.txt`), aber ursprünglich war das **Zusammenfügen von Dateien** die Hauptidee

Beispiel:

```
$ cat datei1.txt datei2.txt > zusammen.txt
```

touch – Datei anlegen I

`touch` legt eine **leere Datei** an oder aktualisiert die **Zeitstempel** (letzter Zugriff und letzte Änderung) einer vorhandenen Datei.

Syntax:

```
$ touch <dateipfad>
```

touch – Datei anlegen II



Hintergrund:

- Viele denken, `touch` erstellt nur Dateien – der **ursprüngliche Zweck** war aber das „Berühren“ von Dateien, um deren **Zeitstempel zu aktualisieren**
- Das Wort „*touch*“ ist also **wörtlich zu nehmen**: man berührt die Datei, **ohne ihren Inhalt zu verändern**

Quiz – Zeitstempel ändern, aber wozu? I

Auf den ersten Blick erscheint es wenig sinnvoll, den **Zeitstempel einer Datei zu ändern, ohne ihren Inhalt zu verändern**.

Dafür gibt es aber praktische Anwendungsfälle.

Fallen Ihnen welche ein? 🤔

touch – Datei anlegen III

Syntax:

```
$ touch <datei>
$ touch <datei1> <datei2> <datei3> …
```

Beispiel:

```
$ touch hello world foo bar
```

→ Man **kann mehrere Dateien mit einem einzigen Befehl** erstellen oder deren Zeitstempel ändern.

💡 Unter Linux wird bei Dateinamen normalerweise zwischen **Groß- und Kleinschreibung** unterschieden (**case-sensitive**). Das bedeutet: **Datei.txt**, **datei.txt** und **DATEI.TXT** sind drei unterschiedliche Dateien!

Erste Befehle in der Bash I

```
$ echo Hallo

$ echo Man kann auch Leerzeichen haben

$ touch datei

$ ls

$ cat datei
```

Erste Befehle in der Bash II

```
$ echo Hallo > datei

$ cat datei

$ echo Welt > datei

$ cat datei

$ echo Ich bin ein Text >> datei

$ cat datei
```

cp – Dateien und Verzeichnisse kopieren

`cp` steht für *copy*.

`cp` **kopiert eine oder mehrere Dateien**.

Die **Originaldatei bleibt unverändert**.

Syntax:

```
$ cp <quelle> <ziel>
```

💡 Mit der Option `-r` (*recursive*) können auch Verzeichnisse inklusive ihres Inhalts kopiert werden.

mv – Dateien und Verzeichnisse verschieben

`mv` steht für *move*.

`mv` **verschiebt Dateien und Verzeichnisse**.

Anders als bei `cp` wird **keine Kopie** erstellt.

Syntax:

```
$ mv <quelle> <ziel>
```

Kann auch zum **Umbenennen** verwendet werden:

```
$ mv alt.txt neu.txt
```

rm – Dateien löschen

`rm` steht für *remove*.

Löscht Dateien direkt – ohne Papierkorb!

Syntax:

```
$ rm <pfad_zur_datei>
```

Quiz – cp, mv und rm I

Welche der drei Dateien (`datei`, `dateineu`, `dateiganzneu`) werden jeweils nach `ls` angezeigt?

```
$ cp datei dateineu
$ ls
$ mv dateineu dateiganzneu
$ ls
$ rm dateiganzneu
$ ls
```

Grundlegende Befehle – Übersicht I

- `echo` → ?
- `ls` → ?
- `cat` → ?
- `touch` → ?
- `cp` → ?
- `mv` → ?
- `rm` → ?

Grundlegende Befehle – Übersicht I

- `echo` → Text ausgeben
- `ls` → ?
- `cat` → ?
- `touch` → ?
- `cp` → ?
- `mv` → ?
- `rm` → ?

Grundlegende Befehle – Übersicht I

- `echo` → Text ausgeben
- `ls` → Verzeichnisinhalt auflisten
- `cat` → ?
- `touch` → ?
- `cp` → ?
- `mv` → ?
- `rm` → ?

Grundlegende Befehle – Übersicht I

- `echo` → Text ausgeben
- `ls` → Verzeichnisinhalt auflisten
- `cat` → Dateiinhalt anzeigen
- `touch` → ?
- `cp` → ?
- `mv` → ?
- `rm` → ?

Grundlegende Befehle – Übersicht I

- `echo` → Text ausgeben
- `ls` → Verzeichnisinhalt auflisten
- `cat` → Dateiinhalt anzeigen
- `touch` → Datei erstellen / Zeitstempel aktualisieren
- `cp` → ?
- `mv` → ?
- `rm` → ?

Grundlegende Befehle – Übersicht I

- `echo` → Text ausgeben
- `ls` → Verzeichnisinhalt auflisten
- `cat` → Dateiinhalt anzeigen
- `touch` → Datei erstellen / Zeitstempel aktualisieren
- `cp` → Datei oder Verzeichnis kopieren
- `mv` → ?
- `rm` → ?



Zum **Kopieren ganzer Verzeichnisse** wird die Option `-r` benötigt.

Grundlegende Befehle – Übersicht I

- `echo` → Text ausgeben
- `ls` → Verzeichnisinhalt auflisten
- `cat` → Dateiinhalt anzeigen
- `touch` → Datei erstellen / Zeitstempel aktualisieren
- `cp` → Datei oder Verzeichnis kopieren
- `mv` → Datei oder Verzeichnis verschieben / umbenennen
- `rm` → ?



Zum **Kopieren ganzer Verzeichnisse** wird die Option `-r` benötigt.

Grundlegende Befehle – Übersicht I

- `echo` → Text ausgeben
- `ls` → Verzeichnisinhalt auflisten
- `cat` → Dateiinhalt anzeigen
- `touch` → Datei erstellen / Zeitstempel aktualisieren
- `cp` → Datei oder Verzeichnis kopieren
- `mv` → Datei oder Verzeichnis verschieben / umbenennen
- `rm` → Datei löschen



Zum **Kopieren ganzer Verzeichnisse** wird die Option `-r` benötigt.

mkdir – Verzeichnisse erstellen

`mkdir` steht für *make directory*.

Erstellt ein Verzeichnis.

Syntax:

```
$ mkdir <pfad_zum_verzeichnis>
```

Mit **mkdir** lassen sich auch mehrere Verzeichnisse gleichzeitig erstellen:

```
$ mkdir ordner1 ordner2 ordner3
```

rmdir – Verzeichnisse löschen I

`rmdir` steht für *remove directory*.

Löscht ein leeres Verzeichnis.

Syntax:

```
$ rmdir <pfad_zum_verzeichnis>
```

rmdir – Verzeichnisse löschen II

Ein Verzeichnis inklusive aller enthaltenen Dateien und Unterverzeichnisse kann mit `rm -r <verzeichnis>` gelöscht werden (`-r` für *recursive*).

Hinweis:

`rm -r` löscht auch den gesamten Inhalt eines Verzeichnisses – **ohne Papierkorb!**

cd – Verzeichnis wechseln I

`cd` steht für *change directory*.

Wechselt das aktuelle Arbeitsverzeichnis.

Syntax:

```
$ cd [pfad_zum_verzeichnis]
```

💡 `cd` ohne Pfadangabe wechselt in das **Home-Verzeichnis** des Benutzers.

cd – Verzeichnis wechseln II

Zwei wichtige Grundkonzepte:

- `.` steht für das aktuelle Verzeichnis.
- `..` steht für das übergeordnete Verzeichnis.

Beispiele:

`$ cd .` → bleibt im aktuellen Verzeichnis

`$ cd ..` → wechselt eine Verzeichnisebene nach oben

💡 `cd .` ist im Grunde ein Verzeichniswechsel ohne tatsächlichen Wechsel. Das ist normalerweise überflüssig, aber technisch vollkommen gültig. 😄

Weiter geht‘s: Quiz – mkdir und cd I

Was zeigt `ls` jeweils im aktuellen Verzeichnis an?

```
$ mkdir verzeichnis1
$ ls
$ cd verzeichnis1
$ ls
$ cd ..
$ ls
```

Grundlegende Befehle – Übersicht II

- `mkdir` → neues Verzeichnis erstellen
- `rmdir` → leeres Verzeichnis löschen
- `cd` → Verzeichnis wechseln

Befehlsoptionen I

Syntax:

```
<befehl> [optionen] [argumente]
```

Beispiel:

```
$ cp [optionen] <quelle> <ziel>
$ cp -r -v verzeichnis1 verzeichnis2
```

Bedeutung der Optionen:

- `-r` (*recursive*) kopiert Verzeichnisse rekursiv, also inklusive aller Unterverzeichnisse und Dateien
- `-v` (*verbose*) zeigt an, was gerade kopiert wird



Zur Erinnerung: Eckige Klammern `[ ]` kennzeichnen optionale Angaben, spitze Klammern `< >` stehen für Platzhalter.

Befehlsoptionen II

```
$ cp -r -v verzeichnis1 verzeichnis2
$ cp -rv verzeichnis1 verzeichnis2
$ cp --recursive --verbose verzeichnis1 verzeichnis2
```

→ **Alle drei Befehle bewirken dasselbe:** Sie kopieren `verzeichnis1` inklusive aller Unterverzeichnisse und Dateien nach `verzeichnis2` und zeigen dabei die Kopiervorgänge an.

Befehlsoptionen III

- **Optionen:** erweitern oder verändern das Verhalten eines Befehls
- **Kurzoptionen:** ein Bindestrich, z. B. `-l`, `-a`
- **Kombinierte Kurzoptionen:** `-la`
- **Langoptionen:** zwei Bindestriche, z. B. `--verbose`, `--recursive`



`-l` und `-a` sind typische Optionen für den Befehl `ls`:

- `-l` (*long format*) zeigt Dateien detailliert an (Rechte, Besitzer, Größe, Datum usw.)
- `-a` (*all*) zeigt alle Dateien und Verzeichnisse, einschließlich versteckter Einträge (die mit **.** beginnen)

man und --help I

Um sich unter Linux selbst zu helfen, kann man Informationen zu Befehlen direkt im Terminal nachschlagen!

`man` und `--help` geben Auskunft über Aufbau, Verwendung und Optionen eines Befehls:

- `man` (*manual*) → ausführliches **Handbuch**
- `--help` → kurze **Übersicht** über Verwendung und Optionen

Syntax:

```
$ man <befehl>
$ <befehl> --help
```

man und --help II

`man`: Ideal, um einen Befehl genauer zu verstehen oder ausführlichere Informationen nachzulesen

`--help`: Ideal, um schnell die Syntax oder eine bestimmte Option nachzuschlagen

man und --help III

```
CP(1)                                    Dienstprogramme für Benutzer

BEZEICHNUNG
       cp - Dateien und Verzeichnisse kopieren

ÜBERSICHT
       cp [OPTION]… [-T] QUELLE ZIEL
       cp [OPTION]… QUELLE… VERZEICHNIS
       cp [OPTION]… -t VERZEICHNIS QUELLE…

BESCHREIBUNG
       Kopieren von QUELLE nach ZIEL oder mehrere QUELLE(n) in VERZEICHNIS.

       Die obligatorischen Argumente für Optionen sind für deren Kurz- und Langform gleich.

       -a, --archive
              Genau wie -dR --preserve=all

       --attributes-only
              Die Daten der Datei nicht kopieren, nur die Attribute

       --backup[=STEUERUNG]
              Eine Sicherungskopie von jeder vorhandenen Zieldatei erstellen

       -b     Wie --backup, akzeptiert aber kein Argument

       --copy-contents
              Wenn rekursiv, Inhalt von Spezialdateien kopieren

Manual page cp(1) line 1 (press h for help or q to quit)
```

Beispiel:
Die Handbuchseite des Befehls **cp** (`man cp`).

Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

man und --help IV

Beispiel: `$ man cp`

- Öffnet das Handbuch zum Befehl `cp`
- Beenden mit **q**
- Navigation mit **Pfeiltasten** oder **Bild ↑ / Bild ↓**

💡 **Suche innerhalb einer Handbuchseite:**

- Vorwärts suchen: **/** + Suchbegriff + **Enter**
- Rückwärts suchen: **?** + Suchbegriff + **Enter**
- Zum nächsten Suchtreffer springen: **n**
- Zum vorherigen Suchtreffer springen: **N**

Miniübung – man und --help

Beispiel:

```
$ ls -l
$ ls -la
$ ls -lisah
```

→ Unterschiedliche Ausgabe – **was bedeuten die Optionen?**

→ Erst mal `man ls` oder `ls --help` im Terminal aufrufen und nachschauen!

💡 Ein guter Administrator muss nicht jede Option auswendig kennen, sondern wissen, wie er sich selbst helfen kann.

man und --help V

Beispiel: `$ man cp`

- Öffnet das Handbuch zum Befehl `cp`
- Beenden mit **q**
- Navigation mit **Pfeiltasten** oder **Bild ↑ / Bild ↓**

💡 **Suche innerhalb einer Handbuchseite:**

- Vorwärts suchen: **/** + Suchbegriff + **Enter**
- Rückwärts suchen: **?** + Suchbegriff + **Enter**
- Zum nächsten Suchtreffer springen: **n**
- Zum vorherigen Suchtreffer springen: **N**

Tipps für die Arbeit mit der Shell



Tipps für die Arbeit mit der Bash-Shell I

Neben **man** und **--help** gibt es unter Linux noch weitere Möglichkeiten, sich direkt im Terminal Hilfe zu holen:

- **help** – Hilfe zu integrierten Bash-Befehlen (Hilfe der Shell selbst)
- **apropos** – passende Handbuchseiten anhand eines Stichworts finden (damit können Sie sogar Befehle entdecken, deren Namen Sie noch gar nicht kennen 🙂)



Tipps für die Arbeit mit der Bash-Shell II

Hilfe zu Befehlen unter Linux:

Form	Bedeutung	Typischer Einsatz	Beispiel
help	Zeigt Hilfe zu Shell-internen Befehlen an	Für eingebaute Bash-Befehle wie cd, history, export	`help history`
--help	Zeigt eine kurze Übersicht zu einem Programm oder Befehl an	Für einen schnellen Überblick über Optionen	`ls --help`
man	Öffnet das ausführliche Handbuch	Für detaillierte Beschreibungen und Optionen	`man ls`
apropos	Sucht nach passenden Handbuchseiten zu einem Stichwort	Wenn der genaue Befehl noch nicht bekannt ist	`apropos kernel`



Tipps für die Arbeit mit der Bash-Shell III

Pfeiltasten (↑ / ↓): Einfaches Durchblättern

- Mit den **Pfeiltasten ↑ und ↓** durch zuvor eingegebene Befehle blättern
- Die **Befehlshistorie** lässt sich mit `history` ausgeben



Tipps für die Arbeit mit der Bash-Shell IV

Strg + R: Intelligente Rückwärtssuche

- Mit **Strg + R** die **interaktive Rückwärtssuche** in der Befehlshistorie öffnen
- Durch erneutes Drücken von **Strg + R** werden **weitere Treffer** basierend auf dem eingegebenen Suchtext angezeigt
- Sobald der gewünschte Befehl erscheint, kann er mit **Enter ausgeführt** oder mit der **rechten Pfeiltaste (→) zur Bearbeitung übernommen** werden



Tipps für die Arbeit mit der Bash-Shell V

Autovervollständigung mit der Tab-Taste:

- Die Bash kann Tipparbeit abnehmen, indem sie Pfade und Befehle automatisch ergänzt
- **Tab** vervollständigt automatisch, wenn es nur eine eindeutige Möglichkeit gibt
- **Tab Tab** (zweimal drücken) zeigt die möglichen Vervollständigungen an

Beispiel:

Statt den gesamten Pfad zu tippen, einfach den Anfang eingeben und **Tab** drücken.



Tipps für die Arbeit mit der Bash-Shell VI

Mehr Übersicht im GNOME-Terminal:

Wenn die Ausgabe länger ist als das sichtbare Terminalfenster, kann im Ausgabeverlauf nach oben oder unten gescrollt werden.

- **Shift + Bild↑/ Bild↓** → eine Seite nach oben oder unten scrollen
- **Shift + Strg + ↑ / ↓** → zeilenweise nach oben oder unten scrollen
- **Shift + Pos1 / Ende** → ganz nach oben bzw. ganz nach unten springen

Tipps für die Arbeit mit der Bash-Shell VII

Das **GNOME-Terminal** ist ein verbreitetes Terminalprogramm für die **GNOME-Desktopumgebung**. Es ist also das Fensterprogramm für die Arbeit mit der Shell, nicht die Shell selbst. Die Bash läuft typischerweise **im** GNOME-Terminal.

Der wichtige Unterschied:

- **GNOME-Terminal** = Das Programm, das das Terminalfenster bereitstellt
- **Bash** = Die Shell, die darin Befehle interpretiert und ausführt

Quiz – GNOME

Wofür stand **GNOME** ursprünglich?

Quiz – GNU

GNU = ? 🙂 Wer weiß es noch?



Tipps für die Arbeit mit der Bash-Shell VIII

Lange Ausgaben besser lesen:

Bei sehr langen Textdateien empfiehlt sich `less` statt `cat`:

- `less` zeigt den Inhalt **seitenweise** an
- Darin kann **gescrollt**, **gesucht** und bequem **navigiert** werden

Beispiel:

```
$ less datei.txt
```

Zur Erinnerung: Weitere Funktionen mit `man less` nachschlagen.

 Tipps für die Arbeit mit der Bash-Shell – Alias I

Alias: Lange Befehle abkürzen

Manche Befehle sind lang, mühsam zu tippen und werden trotzdem ständig benötigt.

Beispiel:

```
$ ls -l --color=auto
```

→ liefert eine übersichtliche, farbige Ausgabe – aber der Befehl ist ziemlich lang

Da **Admins bekanntermaßen faul** sind 😉, gibt es eine praktische Lösung!



Tipps für die Arbeit mit der Bash-Shell – Alias II

→ Der Befehl `alias`

Mit `alias` lassen sich:

- Lange Befehle **abkürzen**
- Eigene **Kurzbefehle** erstellen
- Häufig genutzte Optionen in einem Kurzbefehl verpacken

Syntax:

```
$ alias kurzerName='langerBefehl'
```

Beispiel:

```
$ alias ll='ls -l --color=auto'
```



Tipps für die Arbeit mit der Bash-Shell – Alias III

Alias dauerhaft speichern

Aliase, die direkt im Terminal erstellt werden, gelten **nur für die aktuelle Shell-Sitzung**.

Damit sie **dauerhaft verfügbar** sind, müssen sie in einer Konfigurationsdatei hinterlegt werden.



Tipps für die Arbeit mit der Bash-Shell – Alias IV

1. Datei öffnen:

```
$ nano ~/.bashrc
```

2. Alias eintragen:

```
alias ll='ls -l --color=auto'
```

3. Speichern & neu laden:

```
$ source ~/.bashrc
```

(oder ein neues Terminal öffnen)



Kurzhilfe für nano:

- **Speichern: Strg + O**, anschließend **Enter**
- **Beenden: Strg + X**

Tipps für die Arbeit mit der Bash-Shell – Alias V

```
[root@pxesrv srv]# ls
tftp
[root@pxesrv srv]# ls -l
insgesamt 0
drwxr-xr-x. 6 stephan stephan 178 26. Jul 2020  tftp
[root@pxesrv srv]# alias ls='ls -l --color=auto'
[root@pxesrv srv]# ls
insgesamt 0
drwxr-xr-x. 6 stephan stephan 178 26. Jul 2020  tftp
[root@pxesrv srv]#
```

Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Anders als bei unserem vorherigen Beispiel mit `ll` wird hier der bereits vorhandene Befehlsname `ls` als Alias verwendet.

Tipps für die Arbeit mit der Bash-Shell – Alias VI

```
[root@pxesrv srv]# schicki
bash: schicki: Kommando nicht gefunden.
[root@pxesrv srv]# alias schicki='ls -lh --color=auto'
[root@pxesrv srv]# schicki /etc
insgesamt 1,1M
-rw-r--r--.  1 root root      16  9. Mai 2020  adjtime
-rw-r--r--.  1 root root   1,5K 15. Mai 2020  aliases
drwxr-xr-x.  2 root root   4,0K 30. Okt 23:12 alternatives
-rw-r--r--.  1 root root    541 16. Okt 07:07 anacrontab
drwxr-xr-x.  4 root root   4,0K  9. Mai 2020  asciidoc
drwxr-x---.  4 root root    100 24. Apr 2020  audit
drwxr-xr-x.  3 root root     46 17. Jul 07:09 authselect
drwxr-xr-x.  2 root root     80 15. Aug 15:08 bash_completion.d
-rw-r--r--.  1 root root   3,0K 15. Mai 2020  bashrc
-rw-r--r--.  1 root root    535 21. Apr 2021  bindresvport.blacklist
drwxr-xr-x.  2 root root      6  5. Okt 17:08 binfmt.d
-rw-r--r--.  1 root root     24 14. Sep 06:32 centos-release
drwxr-xr-x.  2 root root      6 28. Jul 07:05 chkconfig.d
-rw-r--r--.  1 root root   1,1K 24. Jun 16:21 chrony.conf
-rw-r-----.  1 root chrony  540 12. Mai 13:06 chrony.keys
drwxr-xr-x.  2 root root     26  9. Aug 21:17 cifs-utils
drwxr-xr-x.  4 root root     42 16. Okt 07:07 cockpit
drwxr-xr-x.  2 root root     21 16. Okt 07:07 cron.d
drwxr-xr-x.  2 root root     23 16. Mai 2020  cron.daily
-rw-r--r--.  1 root root      0 16. Okt 07:07 cron.deny
drwxr-xr-x.  2 root root     22 30. Okt 23:12 cron.hourly
drwxr-xr-x.  2 root root      6 12. Jan 2021  cron.monthly
```

Der Name eines Alias ist frei wählbar – solange er den Namensregeln der Bash entspricht. Auch **schicki** ist also erlaubt! 😉

Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

Texteditoren

Dateien direkt im Terminal erstellen und bearbeiten

Texteditoren

Texteditoren sind Programme zum **Erstellen und Bearbeiten von Textdateien**, z. B. Konfigurationsdateien oder Skripten.

Beliebte Editoren für das Terminal:

- **vi** – der klassische Unix-Editor
- **vim** (*vi improved*) – erweiterte Version von **vi** mit vielen zusätzlichen Funktionen
- **nano / pico** – einfach zu bedienen, ideal für Einsteiger
- **emacs** – sehr leistungsfähig und vielseitig, aber etwas komplexer
- 💡 **Geheimtipp: JOE** – benutzerfreundlich und leicht zu erlernen

Texteditoren – vi

vi – der klassische Texteditor:

- **Vorteil:** Auf vielen Linux-Systemen vorinstalliert
- **Nachteil:** Ungewohnte, nicht intuitive Bedienung
- **Fazit:** Ein Klassiker, dem man immer wieder begegnet – Grundbefehle sollte man kennen

Das Verlassen von **vi** ist für Einsteiger oft eine kleine Herausforderung:





Kurzhilfe für vi:

- **Ohne Speichern beenden: Esc**, dann **:q!** und **Enter**
- **Speichern und beenden: Esc**, dann **:wq** und **Enter**

Texteditoren – vim

vim – *vi improved*:

- **Vorteil:** Mit **vi** weitgehend kompatibel, plus viele Zusatzfunktionen (Syntax-Highlighting, Undo-Historie, Plugins usw.)
- **Nachteil:** Nicht immer vorinstalliert, Einstieg erfordert Übung
- **Fazit:** Lohnt sich, wenn man regelmäßig mit dem Terminal arbeitet
- 💡 **vim** baut auf dem Bedienkonzept von **vi** auf – bietet aber deutlich mehr Komfort

💡 **Kurzhilfe für vim:**
- **Ohne Speichern beenden: Esc**, dann **:q!** und **Enter**
- **Speichern und beenden: Esc**, dann **:wq** und **Enter**

Texteditoren – nano / pico

nano / pico – einsteigerfreundlich:

- **Vorteil:** Sehr einfach zu bedienen, ideal für Einsteiger
- **Nachteil:** Weniger mächtig als **vi/vim**, nicht immer vorinstalliert
- **Fazit:** Praktisch, schnell einsatzbereit – *„nano geht immer“* 😉
- 💡 **nano** wurde als freie Alternative zu **pico** entwickelt – mit nahezu gleicher Bedienung, aber zusätzlichen Funktionen

💡 **Kurzhilfe für nano / pico:**

- **Ohne Speichern beenden: Strg + X**, anschließend ggf. **N** (Änderungen verwerfen)
- **Speichern und beenden: Strg + O**, dann Enter und anschließend **Strg + X**

Texteditoren – emacs

emacs – der mächtige Alleskönner:

- **Vorteil:** Sehr mächtig, kann fast alles, von Textbearbeitung bis zu Programmierung und E-Mail
- **Nachteil:** Extrem komplex – vollständiges Beherrschen dauert sehr lange
- **Fazit:** Nur für Geduldige oder Profis, die viele Funktionen nutzen wollen



Kurzhilfe für emacs:

- **Ohne Speichern beenden: Strg + X**, dann **Strg + C** und das Speichern bei Nachfrage ablehnen (**n**), anschließend das Beenden bestätigen (**yes**)
- **Speichern und beenden: Strg + X**, dann **Strg + S**; anschließend **Strg + X**, dann **Strg + C**

Texteditoren – JOE

JOE (*Joe's Own Editor*) – der Geheimtipp:

- **Vorteil:** Alle wichtigen Funktionen für den Alltag
- **Nachteil:** Meist nicht vorinstalliert
- **Fazit:** Praktisch und einfach zu bedienen



Kurzhilfe für JOE:

- **Ohne Speichern beenden: Strg + C**, bei Rückfrage mit **y** bestätigen
- **Speichern und beenden: Strg + K**, dann **X**

Texteditoren selbst ausprobieren! I

Bei einer **typischen Debian-Installation** sind **vi** und **nano** vorinstalliert.

vi-Tutorial:

https://www.tutorialspoint.com/unix/unix-vi-editor.htm

vim spielerisch lernen:

- **vim installieren:** `$ sudo apt install vim`
- **vimtutor starten:** `$ vimtutor` – interaktives Tutorial direkt im Terminal
- https://vim-adventures.com/ – Browsergame 🎮
- https://www.vim.org/docs.php – vim-Dokumentation

Texteditoren selbst ausprobieren! II

nano ausprobieren:

```
$ nano [pfad/zur/datei]
```

Hilfe zu nano:

- Handbuch: `$ man nano`
- **Hilfeleiste** unten im Editor
- **Ausführliche Hilfe: Strg + G**

SSH

Verschlüsselte Verbindungen zu entfernten Systemen

SSH – Secure Shell I

SSH ist ein weit verbreitetes **Standardprotokoll**, um Systeme **aus der Ferne sicher zu administrieren**.

Es ersetzt unsichere ältere Verfahren wie **Telnet**, weil die **Kommunikation verschlüsselt übertragen** wird (Passwörter, Befehle, Ausgaben).

Ein **SSH-Client** stellt eine **verschlüsselte Verbindung** zu einem Rechner her, auf dem ein **SSH-Server** läuft.

SSH – Secure Shell II

SSH ermöglicht eine **interaktive Remote-Shell**, mit der man ein entferntes System bedienen kann, als säße man direkt davor.

SSH ermöglicht außerdem, **einzelne Befehle auszuführen oder Dateien sicher zu übertragen** – ganz ohne interaktive Remote-Shell.

SSH – Secure Shell III

Voraussetzungen:

- Eine funktionierende **Netzwerkverbindung** zwischen Client und Server
- Der **SSH-Server** auf dem Zielsystem läuft und akzeptiert Verbindungen
- Der **SSH-Port** (standardmäßig **TCP 22**) ist erreichbar und wird nicht durch eine Firewall blockiert

SSH – Secure Shell IV

```
stephan@vmhost2:~$ ssh stephan@10.0.0.16
stephan@10.0.0.16's password:
Activate the web console with: systemctl enable --now cockpit.socket

Last failed login: Tue Nov  2 18:09:43 CET 2021 from 10.0.0.12 on ssh:notty
There were 3 failed login attempts since the last successful login.
[stephan@pxesrv ~]$
```

Quelle: Stephan Rösner, Linux-Unterrichtsskript (Vorgängerversion)

→ Mit `ssh stephan@10.0.0.16` wird eine SSH-Verbindung zum Rechner **10.0.0.16** mit dem Benutzer **stephan** aufgebaut

→ Nach der Anmeldung kann man direkt auf dem entfernten Rechner arbeiten

💡 **Woran erkennt man das?** Der Shell-Prompt wechselt von **stephan@vmhost2** zu **stephan@pxesrv**. Die Befehle werden jetzt auf dem entfernten Rechner ausgeführt.

SSH – Verbindungsaufbau I

Grundsyntax des SSH-Befehls:

```
$ ssh [optionen] [benutzername@]server
```

Beispiel:

```
$ ssh stephan@192.168.117.157
```

Wenn der **lokale Benutzername** und der **Benutzername auf dem Zielsystem** gleich sind, kann der Benutzername weggelassen werden:

```
$ ssh 192.168.117.157
```

Statt einer IP-Adresse kann auch ein **Hostname** genutzt werden:

```
$ ssh server01.schule.local
```

SSH – Verbindungsaufbau II

SSH – Wichtige Optionen:

- `-p <port>` – Verbindung über einen anderen Port als den Standard-Port 22
- `-i <datei>` – verwendet einen bestimmten privaten SSH-Schlüssel, z. B. für verschiedene Server oder Benutzerkonten (`i` für *identity file*)
- `-v` / `-vv` / `-vvv` – zeigt zusätzliche Informationen zur Fehlersuche an (je mehr „v“, desto ausführlicher die Ausgabe)

 **ssh -v** ist sehr hilfreich, wenn eine SSH-Verbindung nicht funktioniert. Damit lässt sich beispielsweise nachvollziehen, an welcher Stelle der Verbindungsaufbau oder die Anmeldung scheitert.

SSH-Client auf verschiedenen Betriebssystemen

SSH nutzen unter Linux, macOS und Windows:

- **Linux:** oft bereits vorhanden, sonst per Paketverwaltung installierbar
- **macOS:** SSH direkt im Terminal nutzbar
- **Windows:** OpenSSH-Client in modernen Versionen verfügbar (ggf. nachinstallieren); Nutzung in PowerShell, Eingabeaufforderung oder Windows Terminal

Komfort-Tools für Windows:

- **PuTTY** – leichtgewichtig und klassisch
- **MobaXterm** – viele Zusatzfunktionen in einem Werkzeug

 **Auch unter Windows sind zusätzliche SSH-Programme nicht zwingend notwendig.** Der OpenSSH-Client ermöglicht SSH-Verbindungen direkt über das Terminal. **PuTTY** und **MobaXterm** bieten lediglich zusätzliche Komfortfunktionen.

SSH in Aktion – Praxis-Experiment I

Ziel:

Von **VM1** per SSH auf **VM2** zugreifen und dort eine Datei finden und anzeigen.

Vorbereitung:

Eine **zweite Debian-VM in VirtualBox installieren** und dabei in der Softwareauswahl die Option **„SSH server“ aktivieren** – wir wollen später eine SSH-Verbindung zwischen diesen beiden VMs herstellen!

SSH in Aktion – Praxis-Experiment II

Optional: Vorbereitung auf VM2 – OpenSSH-Server nachinstallieren

Auf VM2 im Terminal:

```
# apt update
# apt install openssh-server
```

SSH-Dienst starten und aktivieren:

```
# systemctl start ssh
# systemctl enable ssh
```

Status des Dienstes überprüfen:

```
# systemctl status ssh
```

SSH in Aktion – Praxis-Experiment III

1. Vorbereitung auf VM2 – Testdatei erstellen

Auf VM2 im Terminal:

```
$ mkdir ~/vm2_test
$ echo "Ich bin eine Textdatei auf VM2" > ~/vm2_test/hallo_vm2.txt
```

SSH in Aktion – Praxis-Experiment IV

2. IP-Adressen herausfinden

Auf beiden VMs im Terminal:

```
$ ip a
```

→ Die **IP-Adressen** notieren (für unsere Übung verwenden wir **IPv4**). Wichtig ist die Adresse der Netzwerkschnittstelle, über die sich die beiden VMs erreichen können.

💡 Die **IPv4-Adresse** erkennen Sie bei **ip a** an der Kennzeichnung **inet**. **inet6** bezeichnet dagegen eine IPv6-Adresse.

SSH in Aktion – Praxis-Experiment V

3. Netzwerk testen (von VM1 → VM2):

Auf VM1:

```
$ ping -c 3 <IP_von_VM2>
```

→ `-c 3` **(*count*):** Anzahl der Pings, hier 3 Pakete

→ **Wenn Antworten kommen:** VM2 ist über das Netzwerk erreichbar

SSH in Aktion – Praxis-Experiment VI

4. SSH-Verbindung von VM1 zu VM2

Auf VM1:

```
$ ssh benutzername@<IP_von_VM2>
```

→ **Beim ersten Verbindungsaufbau:** Server-Fingerprint prüfen und mit ***yes*** bestätigen (das eine wichtige Sicherheitsregel: Nicht jede SSH-Verbindung einfach bestätigen!)

→ **Passwort des Benutzers auf VM2** eingeben

→ Jetzt sind Sie per **Remote-Shell** auf VM2 angemeldet

SSH in Aktion – Praxis-Experiment VII

5. Verzeichnis & Datei auf VM2 finden

In der SSH-Sitzung (von VM1 aus auf VM2 angemeldet):

```
$ cd ~/vm2_test
$ ls
$ cat hallo_vm2.txt
```

→ Sie arbeiten von **VM1** aus direkt im Dateisystem von **VM2** – das ist **SSH in der Praxis!**

Lernfolien Modul 2

Das Wichtigste auf einen Blick

Modul 2 – Der Überblick zum Schluss

- **Grundlegende Befehle:** `echo`, `ls`, `cat`, `touch`, `cp`, `mv`, `rm`, `mkdir`, `rmdir`, `cd`
- **Befehlsoptionen:** Kurzoptionen, kombinierte Kurzoptionen, Langoptionen
- **Hilfe zu Befehlen:** `man`, `--help`, `help` und `apropos`
- **Praktische Shell-Tipps:** Befehlshistorie, Rückwärtssuche, Tab-Vervollständigung, `less` und Aliase
- **Texteditoren: vi** (Klassiker), **vim** (*vi improved*), **nano** (einsteigerfreundlich), **emacs** (mächtig) und **JOE** (Geheimtipp)
- **SSH (Secure Shell):** Verschlüsselte Verbindungen zu entfernten Systemen

Die wichtigsten Befehle – Übersicht

- `echo` → Text ausgeben
- `ls` → Dateien und Verzeichnisse auflisten
- `cat` → Dateiinhalt anzeigen
- `touch` → leere Datei erstellen oder Zeitstempel aktualisieren
- `cp` → Dateien oder Verzeichnisse kopieren
- `mv` → Dateien oder Verzeichnisse verschieben bzw. umbenennen
- `rm` → Dateien löschen
- `mkdir` → Verzeichnis erstellen
- `rmdir` → leeres Verzeichnis löschen
- `cd` → Verzeichnis wechseln