Vorlesung Linux-Praktikum

2. Dateirechte, Ausgabeumleitung und weitere Befehle

Dirk Frettlöh

Technische Fakultät
Universität Bielefeld

Zusammenfassung heute

`ls -l`	Dateirechte anzeigen
`chmod, chown`	Dateirechte ändern
`umask`	Dateirechte voreinstellen
`bc`	einfacher Taschenrechner
`>, >>, <`	Aus-/Eingabe umleiten
`sort`	Sortieren
`;`	mehrere Befehle in einer Zeile trennen
`|`	"Pipe", Ausgabe des letzten Befehls als Eingabe des nächsten Befehls nehmen
`cat`	Aneinanderhängen
`Strg-c`	Programm abbrechen
`echo`	Argument ausgeben
`head, tail`	Anfang/Ende einer Datei anzeigen
Shellskripte	`skript.sh`

Datei- und Verzeichnisrechte

Übersicht

Dateien verwalten

- vieles kennen wir jetzt schon: `pwd`, `ls`, `cd`, `cp`, `mv`, `rm`

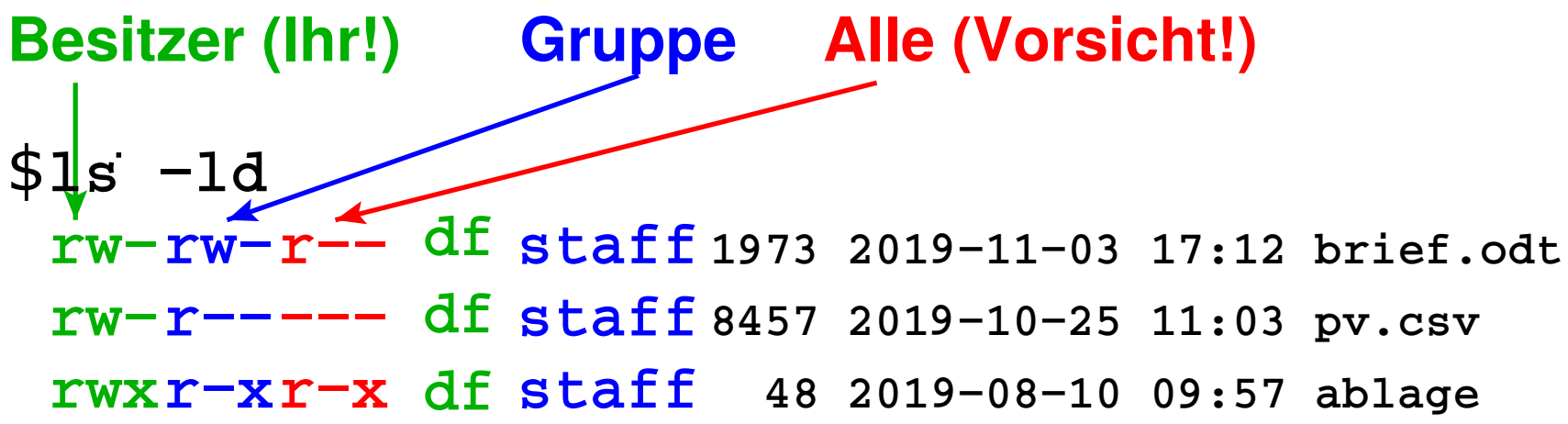
Weitere typische Aufgaben

- Lese- / Schreibrechte verstehen
- ... und verwalten

Datei- und Verzeichnisrechte

Zugriffsrechte

3-stufiges System von Berechtigungen:



Grundlegende Berechtigungen (engl. permissions):

r	read	Öffnen / Lesen erlaubt
w	write	Schreibzugriff / Löschen erlaubt
x	execute	Dateien: Programmausführung erlaubt
		Verzeichnisse: Durchgreifen erlaubt

Datei- und Verzeichnisrechte

Zu welchen Gruppen gehöre ich?

groups

```
$ groups
bmstaff teachlinux vlvkinf tak ...
```

id (identity)

```
$ id
uid=22227(df) gid=12000(bmstaff) groups=...
```

- gid: primäre Gruppe

Wann bekommt man zusätzliche Gruppen? (Im Techfak-System)

- Maschinenbezogen (z.B. audio bei lokalem login an PCs)
- Statuswechsel (HiWi werden, Bachelorarbeit schreiben)

Datei- und Verzeichnisrechte

Berechtigungen ändern

chmod (change file mode)

`$ chmod g-w datei`	kein Schreibzugriff für Gruppe
`$ chmod u+w datei`	erlaube Schreibzugriff für sich selbst
`$ chmod o=r datei`	erlaube nur Lesezugriff für alle (w,x werden gelöscht)
`$ chmod go-rwx *.txt`	für *.txt-Dateien alle Zugriffe für Gruppe und alle wegnehmen
`$ chmod g=rw,o= datei`	Gruppe darf lesen und schreiben, andere haben keinen Zugriff

u : Berechtigung für Besitzer (user; erster rwx-Block)
g : Berechtigung für Gruppe (group; zweiter rwx-Block)
o : Berechtigung für Alle (other; dritter rwx-Block)

Datei- und Verzeichnisrechte

Berechtigungen ändern

chown (change owner)

Im Prinzip so:

```
$ chown juser scan0003.pdf
chown:  changing ownership of 'scan0003.pdf':
Operation not permitted
```

Das darf nur der Superuser (su)

```
$ sudo chown juser datei.txt
```

Datei- und Verzeichnisrechte

Prioritäten auf Dateiberechtigungen

Die speziellste anwendbare Berechtigung gilt:
(am Beispiel jeweils aus Sicht des Nutzers `df`)

Berechtigung	Nutzer	Gruppe	df darf lesen
`-r--------`	df	staff	ja
`----r--r--`	df	staff	nein
`----r--r--`	juser	staff	ja
`-------r--`	juser	staff	nein
`-------r--`	nn	nn	ja

(df sei Mitglied der Gruppe `staff`, aber nicht in `nn`)

Datei- und Verzeichnisrechte

Berechtigungen auf Verzeichnissen

'w'-Berechtigung auf Verzeichnis

- Anlegen von Dateien / Unterverzeichnissen
- Löschen von Dateien / Unterverzeichnissen

Zusammenspiel von Datei- und Verzeichnisberechtigungen

```
$ ls -ld
 dr-xr-xr-x  4  df    staff  ...  .
 drwxrwxr--  3  root  root   ...  ..
 -rw-r--r--  1  df    staff  ...  brief.txt
```

- `brief.txt` kann verändert werden (Dateiberechtigung)
- `brief.txt` kann nicht gelöscht werden (Verzeichnisber.)
- Neue Dateien können nicht angelegt werden (Verzeichnisberechtigung)

Datei- und Verzeichnisrechte

Berechtigungen auf Verzeichnissen

'w' auf Gruppenverzeichnis hebelt Datei-Schreibschutz aus

```
drwxrwxr-x  4  df    projekt  ...  .
drwxrwxr--  3  root  root     ...  ..
-rw-r--r--  1  df    projekt  ...  brief.txt
```

Nutzer `nn` sei ebenfalls in der Gruppe `projekt`:

- nn kann `brief.txt` nicht editieren, aber
- nn kann `brief.txt` löschen und neu anlegen

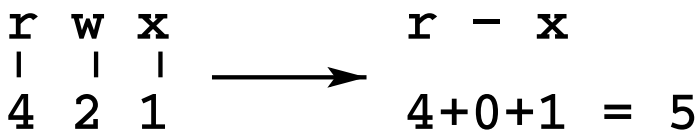
Folgerung:

- Niemals das Home-Verzeichnis gruppen-/weltschreibbar machen!

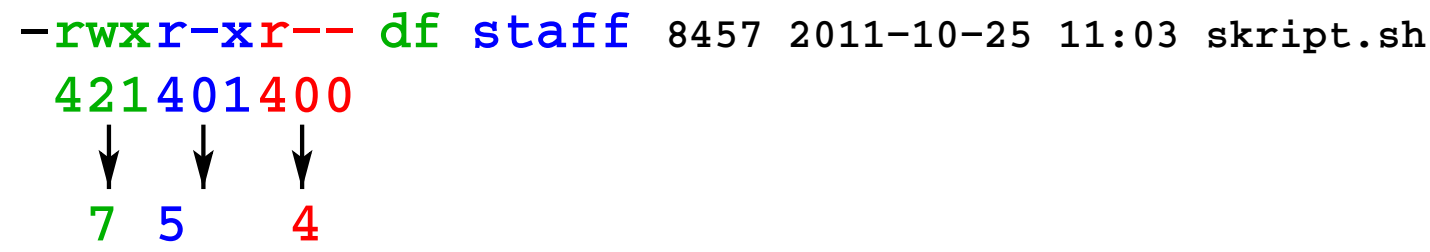
Datei- und Verzeichnisrechte

Nerd-Variante

Bitweise Kodierung: Zahlen für gesetzte Berechtigungen addieren



Beispiel:




```
$ chmod 754 skript.sh
```

Erste Ziffer für User, zweite Ziffer für Group, dritte Ziffer für Others

Datei- und Verzeichnisrechte

Berechtigungen auf Verzeichnissen

'r'-Berechtigung auf Verzeichnis

- erlaubt Dateinamen zu lesen (und sonst nichts!)

'x'-Berechtigung auf Verzeichnis

- erlaubt Inhalt von Dateien und Unterverzeichnissen zu lesen

Typischerweise: `rx` zusammen setzen oder wegnehmen

Datei- und Verzeichnisrechte

umask

Jeweils von Hand Rechte ändern ist lästig.

Rechte für neu angelegte Dateien voreinstellen mit `umask`.

Logik genau andersum:

$$\begin{array}{r} 777 \\ -\,023 \\ \hline 754 \end{array}$$

`umask 023` bewirkt also für alle neuen Dateien 754:

`rwxr-xr--`

Datei- und Verzeichnisrechte

umask

Jeweils von Hand Rechte ändern ist lästig.

Rechte für neu angelegte Dateien voreinstellen mit `umask`.

Logik genau andersum:

$$\begin{array}{r} 777 \\ -\,023 \\ \hline 754 \end{array}$$

`umask 023` bewirkt also für alle neuen Dateien 754:
`rwxr-xr--`

(Lies "Maske" als Filter.)

Die meisten Programme erzeugen neue Dateien ohne das x-bit.
(Ausnahme z.B. Compiler, die sollen ja eine ausführbare Datei liefern)

Datei- und Verzeichnisrechte

setuid und setgid

Auf aktuellen Linux-Rechnern sieht man oft so was:

```
drwsrwSr--
```

- Das User s: `setuid` führe solche Dateien mit den Rechten seines Besitzers aus (nicht mit denen des Users, der sie startet.)
- Das Groups s: `setgid`
  - Führe solche Dateien mit den Rechten seiner Gruppe aus (nicht mit der des Users, der sie startet.)
  - Neu angelegte Dateien in solchen Ordnern erben die Rechte dieses Ordners (nicht die der `umask` des Users, der sie erzeugt)

Datei- und Verzeichnisrechte

setuid und setgid

Auf aktuellen Linux-Rechnern sieht man oft so was:

```
drwsrwSr--
```

- Das User s: `setuid` führe solche Dateien mit den Rechten seines Besitzers aus (nicht mit denen des Users, der sie startet.)
- Das Groups s: `setgid`
  - Führe solche Dateien mit den Rechten seiner Gruppe aus (nicht mit der des Users, der sie startet.)
  - Neu angelegte Dateien in solchen Ordnern erben die Rechte dieses Ordners (nicht die der `umask` des Users, der sie erzeugt)

Großes S: das x-Bit ist nicht gesetzt, aber das setuid/setgid Bit schon. (mehr: 'sticky bit' suchen)

Etwas sinnlose Einstellung, da die Gruppe nicht in den Ordner kann.

Ein-/Ausgabe-Umleitung

Ein-/Ausgabeumleitung

bc (basic calculator)

- ein Kommandozeilen - Taschenrechner

```
$ bc
4 + 7
11
9 * 3
27
quit
```

Ein-/Ausgabeumleitung

Grundidee

Ein- und Ausgabe sind hier Text

Der Standard ist:

- `stdin`: (Standard Input) Tastatur
- `stdout`: (Standard Output) Monitor
- `stderr`: (Standard Error, Fehlermeldung) Monitor

Ein-/Ausgabeumleitung

Grundidee

Ein- und Ausgabe sind hier Text

Der Standard ist:

- `stdin`: (Standard Input) Tastatur
- `stdout`: (Standard Output) Monitor
- `stderr`: (Standard Error, Fehlermeldung) Monitor

Man kann Tastatur und Monitor durch Textdateien ersetzen

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

Der Befehl `echo` gibt einfach seine Argumente an die Ausgabe.

```
$ echo Hallo Welt
Hallo Welt
```

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

Der Befehl `echo` gibt einfach seine Argumente an die Ausgabe.

```
$ echo Hallo Welt
Hallo Welt
```

Also:

```
$ echo Hallo Welt > ausgabe.txt
```

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

Der Befehl `echo` gibt einfach seine Argumente an die Ausgabe.

```
$ echo Hallo Welt
Hallo Welt
```

Also:

```
$ echo Hallo Welt > ausgabe.txt
$ more ausgabe.txt
Hallo Welt
```

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

Der Befehl `echo` gibt einfach seine Argumente an die Ausgabe.

```
$ echo Hallo Welt
Hallo Welt
```

Also:

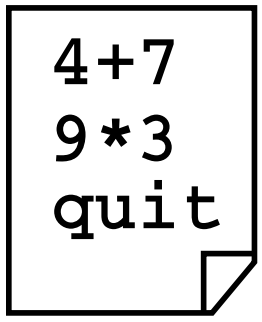
```
$ echo Hallo Welt > ausgabe.txt
$ more ausgabe.txt
Hallo Welt
```

Falls in ausgabe.txt schon etwas steht:

> löscht alten Inhalt
>> hängt an alten Inhalt an

Ein-/Ausgabeumleitung

Beispiel: Eingabeumleitung

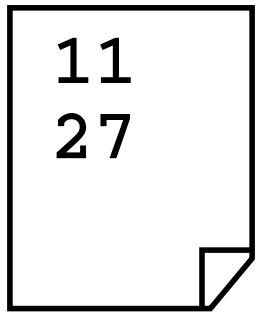



```
$ bc < eingabe.txt
11
27
```

Zeichen für Eingabeumleitung!

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

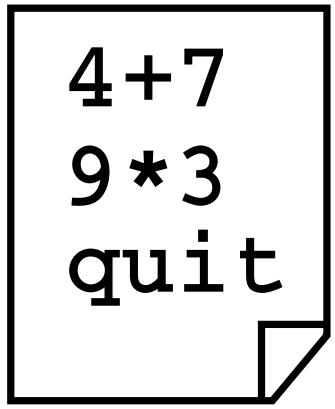


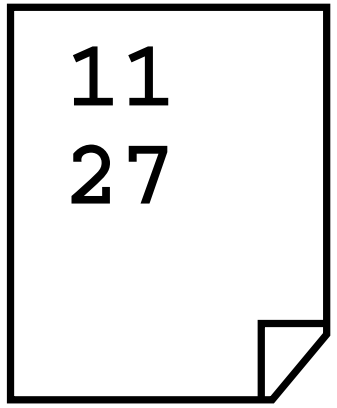

```
$ bc > ausgabe.txt
4+7
9*3
quit
```

Zeichen für Ausgabeumleitung!

Ein-/Ausgabeumleitung

Beispiel: beides gleichzeitig





```
$ bc < eingabe.txt > ausgabe.txt
```

Ein-/Ausgabeumleitung

Textdateien zeilenweise sortieren

sort

```
$ sort planeten.txt
```

- sortiert alphabetisch nach der ersten Spalte

```
$ sort -k 2 planeten.txt
```

- sortiert alphabetisch nach der zweiten Spalte

Ein-/Ausgabeumleitung

Textdateien zeilenweise sortieren

sort

```
$ sort planeten.txt
```

- sortiert alphabetisch nach der ersten Spalte

```
$ sort -k 2 planeten.txt
```

- sortiert alphabetisch nach der zweiten Spalte

```
$ sort -k 2 -n planeten.txt
```

- sortiert numerisch nach der zweiten Spalte

Ein-/Ausgabeumleitung

Verzeichnislisting nach Größe sortieren

```
$ ls -l > zwischen.txt

$ sort -k 5 -n zwischen.txt > sort.txt

$ more sort.txt

$ rm zwischen.txt sort.txt
```

- das Hantieren mit temporären Dateien ist lästig!

Ein-/Ausgabe -Weiterleitung

Ein-/Ausgabeweiterleitung

Grundidee: Verketten von Programmen

Eingabe

Programm 1

Programm 2

...

Programm n

Ausgabe

Ein-/Ausgabeweiterleitung

Anwendung auf das Sortierproblem

Das “Pipe”-Symbol | verbindet die Programme:

```
$ ls -l | sort -k 5 -n | more
```

- Ausgabe des links von | stehenden Programms
- wird Eingabe des rechts von | stehenden Programms

▶ deutlich effizienter als Zwischenspeichern!

Kommandos zum Bearbeiten von Textdateien

Textdateien zusammenfügen

cat (concatenate files)

```
$ cat eins.txt zwei.txt drei.txt
```

- gibt den Inhalt der Dateien nacheinander aus.

```
$ cat eins.txt zwei.txt drei.txt > sammlung.txt
```

- Ergebnis in neuer Datei speichern.

```
$ cat eins.txt
```

- Nützlicher Spezialfall: Eine kurze Datei anschauen

Kommandos zum Bearbeiten von Textdateien

Textdateien zusammenfügen

cat (concatenate files)

```
$ cat eins.txt zwei.txt drei.txt
```

- gibt den Inhalt der Dateien nacheinander aus.

```
$ cat eins.txt zwei.txt drei.txt > sammlung.txt
```

- Ergebnis in neuer Datei speichern.

```
$ cat eins.txt
```

- Nützlicher Spezialfall: Eine kurze Datei anschauen

`cat` ohne Argument liest von der Standardeingabe.

Nützlich & wichtig: Strg-c bricht Programm ab.

Kommandos zum Bearbeiten von Textdateien

Anfang einer Datei ausgeben

head (show head of file)

```
$ head -3 liste.txt
```

- zeigt die ersten 3 Zeilen einer Datei.

Kommandos zum Bearbeiten von Textdateien

Ende einer Datei ausgeben

tail (show tail of file)

```
$ tail -4 liste.txt
```

- zeigt die letzten 4 Zeilen einer Datei.

```
$ tail +7 liste.txt
```

- zeigt alle Zeilen ab der 7ten Zeile
  (bzw. unterdrückt die Zeilen 1 bis 6)

Kommandos zum Bearbeiten von Textdateien

Zusammenfassendes komplexes Beispiel

Aufgabe: Planeten-Tabelle mit Überschrift sortieren

```
$ sort planeten2.txt
```

- klappt nicht wegen der Überschrift

Ansatz: Überschrift mit tail abschneiden

```
$ tail +3 planeten2.txt | sort
```

- besser, aber Überschrift fehlt jetzt

Kommandos zum Bearbeiten von Textdateien

Zusammenfassendes komplexes Beispiel

Überschrift erhält man mit head:

```
$ head -2 planeten2.txt
```

Alles zusammenfügen:

```
$ head -2 planeten2.txt > teil1.txt
$ tail -n +3 planeten2.txt | sort > teil2.txt
$ cat teil1.txt teil2.txt > sortiert.txt
$ rm teil1.txt teil2.txt
```

- aber es entstehen wieder die unschönen Zwischendateien!

Kommandos zum Bearbeiten von Textdateien

Zusammenfassendes komplexes Beispiel

Es geht auch ohne Zwischendateien:

```
$ head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
```

Semikolon trennt Aufrufe

- man kann mehr als ein Programm pro Zeile ausführen
- Ausführung von links nach rechts
- Ausgaben werden aneinandergehängt

Kommandos zum Bearbeiten von Textdateien

Ausgabeumleitung des Ergebnisses

```
$ head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
                                                   > ergebnis.txt
```

- liefert nicht das Gewünschte:
  nur die Ausgabe von `tail` wird umgeleitet

Lösung:

```
$ (head -2 planeten2.txt; tail -n +3 planeten2.txt | sort)
                                                   > ergebnis.txt
```

- gesamten Ausdruck in runden Klammern ausführen, dessen Ausgabe umgeleitet werden soll

Kommandos zum Bearbeiten von Textdateien

Ausgabeumleitung des Ergebnisses

```
$ head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
                                                   > ergebnis.txt
```

- liefert nicht das Gewünschte:
  nur die Ausgabe von `tail` wird umgeleitet

Lösung:

```
$ (head -2 planeten2.txt; tail -n +3 planeten2.txt | sort)
                                                   > ergebnis.txt
```

- gesamten Ausdruck in runden Klammern ausführen, dessen Ausgabe umgeleitet werden soll

Aber es gibt eine bessere Lösung.

Shellskripte

Shellskripte

Exkurs: verschiedene Shells

Es gibt verschiedene Kommandozeileninterpreter, also shells:

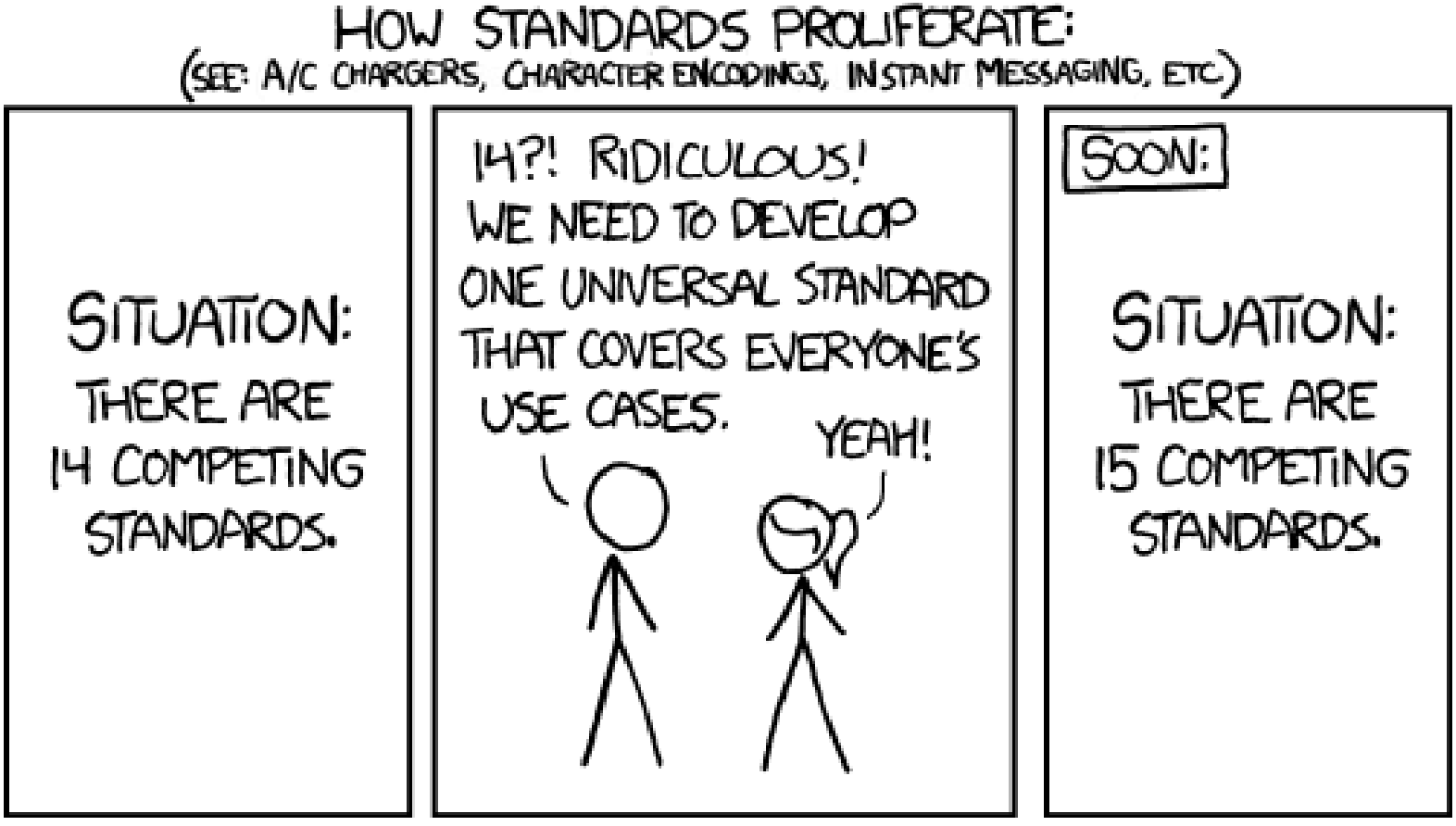
- bash: Bourne-again shell heute DIE shell, auf fast allen Linux- und MacOS-Systemen. Von GNU, seit 1989

- sh: Thompson shell (1971), Bourne shell (1975)
- ksh: Korn shell, pdksh: public domain ksh
- ash: Almquist shell, dash: Debian ash
- zsh: Z shell, csh: C shell,... moderne Erweiterungen

Anzeigen mit `$ echo $SHELL`

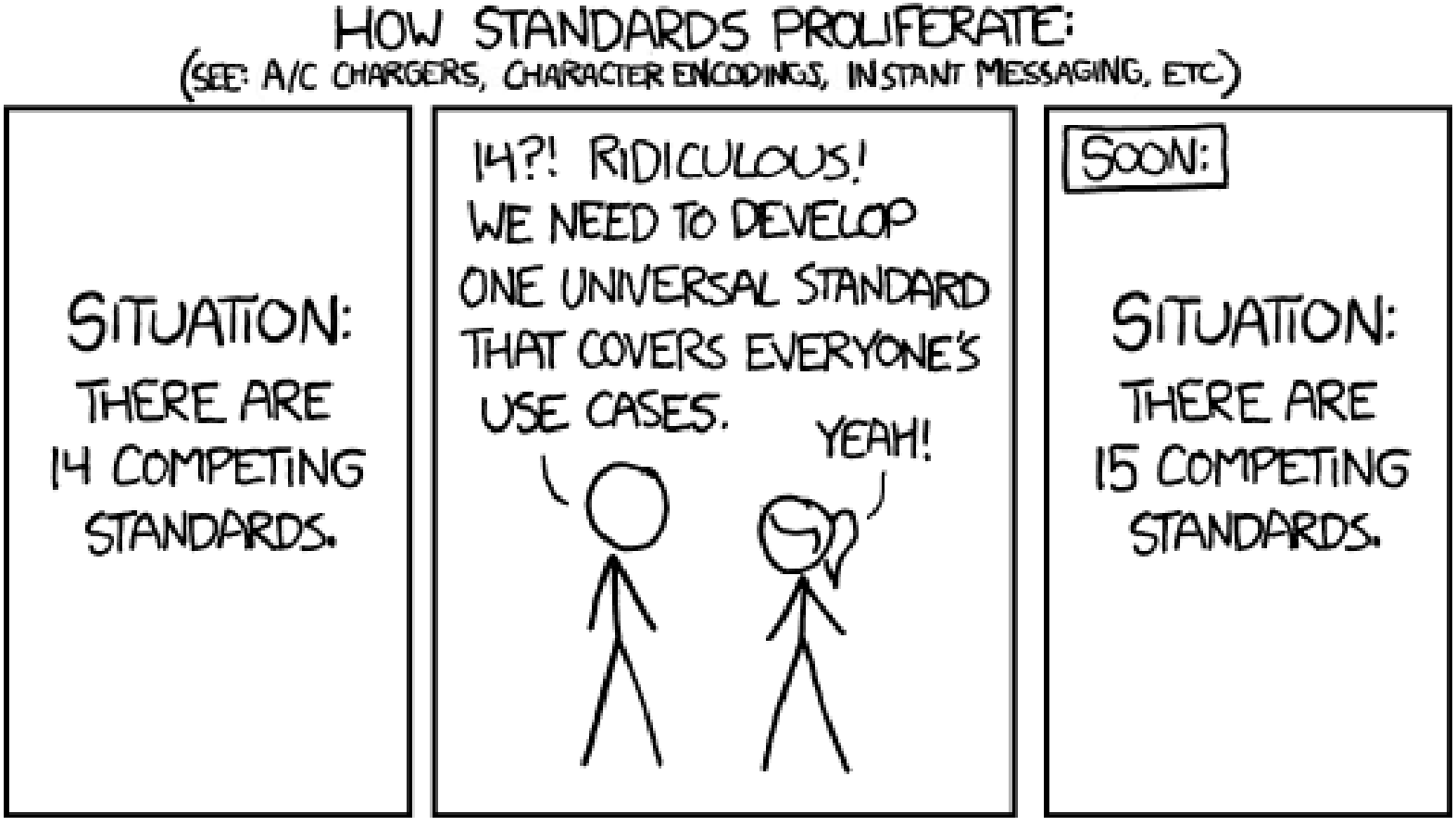
Shellskripte

Exkurs: verschiedene Shells



Shellskripte

Exkurs: verschiedene Shells



...bei shells zum Glück gerade nicht so: bash!

Umgebungsvariablen

Später sehen wir Variablen (z.B. `i=1`)

Falls `i` eine Variable ist, liefert `echo i` einfach `i`.

Umgebungsvariablen

Später sehen wir Variablen (z.B. `i=1`)

Falls `i` eine Variable ist, liefert `echo i` einfach `i`.

`$ echo $i` liefert den Inhalt von `i`.

Umgebungsvariablen

Später sehen wir Variablen (z.B. `i=1`)

Falls `i` eine Variable ist, liefert `echo i` einfach `i`.

`$ echo $i` liefert den Inhalt von `i`.

Es gibt in der bash ganz spezielle Variablen, die **Umgebungsvariablen**. Z.B.

```
$ echo $SHELL
bash
$ echo $USER
frettloe
$ echo $PWD
/homes/frettloe      (vgl pwd!)
```

Umgebungsvariablen

```
$ echo $LANG
de_DE.UTF-8
```

```
$ echo $PS1
\u@\h:\w$
```

`PS1` regelt die Gestalt der Eingabeaufforderung, hier:
`frettloe@hopf57:~$` (user, host, working directory)

...und viele mehr.

Umgebungsvariablen

```
$ echo $LANG
de_DE.UTF-8
$ echo $PS1
\u@\h:\w$
```

PS1 regelt die Gestalt der Eingabeaufforderung, hier:
`frettloe@hopf57:~$` (user, host, working directory)

...und viele mehr.

Alle Anzeigen mit **env**, bzw besser

```
~ $ env | more
TERM_PROGRAM=Apple_Terminal
SHELL=/bin/bash
TERM=xterm-256color
TERM_PROGRAM_VERSION=421.2
USER=frettloe
....
```

Umgebungsvariablen

Suchpfade

which Welchen Pfad hat Programm...

Programme wie `ls`, `grep`, `find`... liegen irgendwo als Dateien auf dem Rechner. Anzeige wo mittels:

```
$ which find
/usr/bin/find

$ which ls
/bin/ls
```

Umgebungsvariablen

Suchpfade

which Welchen Pfad hat Programm...

Programme wie `ls`, `grep`, `find`... liegen irgendwo als Dateien auf dem Rechner. Anzeige wo mittels:

```
$ which find
/usr/bin/find

$ which ls
/bin/ls
```

PATH sagt Linux, wo es `ls` suchen soll.

```
$ echo $PATH
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/homes/frettlo
```

Befehle direkt eingeben

Grenzen dessen

- nur für einzelne Befehle/Pipes praktikabel
- nur ein variabler Eingabewert/-datei pro Alias

  Gegenbeispiel:

```
head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
```

- Eingabewert/-datei muss am Ende stehen
- Programmiermöglichkeiten sehr eingeschränkt
  (Nutzung von Variablen, Fallunterscheidungen, Schleifen)

Shellskripte

Wdh: Aufgaben der Kommandozeile

1. Programm ausführen
2. Programme zu mächtigeren Werkzeugen kombinieren (siehe Pipes!)
3. Kommandozeilen-Skripte
   - (1) und (2) abstrahieren und in Datei speichern
   - wiederverwenden statt erneut eintippen

Shellskripte

Aufbau

Prinzipieller Aufbau eines Shellskriptes

- Textdatei mit folgendem Inhalt:

```
#!/bin/bash
```

Shell zum Ausführen des Skriptes

```
echo Hallo
echo ich bin ein
echo Shellskript
```

Aufrufe, wie Ihr sie auch direkt eintippen würdet

Shellskripte

Aufrufen

Ausführbarkeits-Bit setzen; aufrufen (mit `./`):

```
$ chmod u+x skript.sh
$ ./skript.sh
```

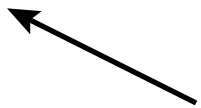
Shellskripte

Suchpfade erweitern

Vorsicht: vermurkster Suchpfad → alle Programme “weg”

(Programme sind noch da, aber die Shell findet sie nicht mehr)

```
> PATH=$PATH:~/shell-skripte
```



nicht vergessen (beliebte Falle ;–)

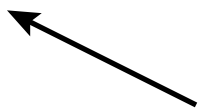
Shellskripte

Suchpfade erweitern

Vorsicht: vermurkster Suchpfad → alle Programme "weg"

(Programme sind noch da, aber die Shell findet sie nicht mehr)

```
> PATH=$PATH:~/shell-skripte
```



nicht vergessen (beliebte Falle ;–)

Erste Hilfe: absolute Pfade benutzen, z.B.

```
$ /bin/ls
$ /usr/bin/emacs ~/.bash_aliases
```

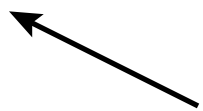
Shellskripte

Suchpfade erweitern

Vorsicht: vermurkster Suchpfad → alle Programme "weg"

(Programme sind noch da, aber die Shell findet sie nicht mehr)

```
> PATH=$PATH:~/shell-skripte
```



nicht vergessen (beliebte Falle ;–)

Erste Hilfe: absolute Pfade benutzen, z.B.

```
$ /bin/ls
$ /usr/bin/emacs ~/.bash_aliases

$ which ls
/bin/ls
```

Shellskripte

Keine Sicherheitslücken aufmachen!

Bitte **nicht** nachmachen:

~~den Punkt . in den Suchpfad aufnehmen, also~~
~~`PATH=.:$PATH` oder `PATH=$PATH:.`~~

Im Verzeichnis `/tmp` gebe es folgendes Skript:

```
#!/bin/bash
rm -rf ~/* # löscht das Benutzerverzeichnis
```

und zwar mit dem Namen "`ls`".

Würdet Ihr dort `./ls` aufrufen? Nein?
Dann nehmt auch `.` nicht in Euren Suchpfad auf!

Shellskripte

Parameterübergabe

Beispiel zur Übergabe von Parametern:

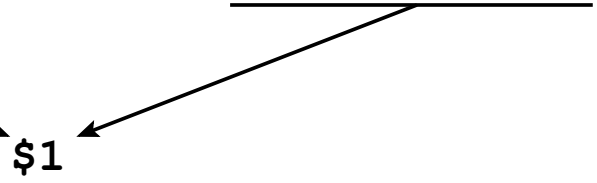
```
#!/bin/bash

echo "Erstes : $1"
echo "Zweites: $2"
echo "Drittes: $3"

echo "Anzahl : $#"
echo "Alle   : $*"
```

Shellskripte

Beispiel

Erinnerung:

```
> head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
```



Abstrahieren und zusammenfassen:

```
#!/bin/bash
head -2 $1 ; tail -n +3 $1 | sort
```

```
$ ./hsort.sh planeten2.txt
```

Shellskripte

zu sortierende Spalte mit angeben

```
#!/bin/bash
head -2 $1 ; tail -n +3 $1 | sort -k $2 -n
```

```
$ ./hsort2.sh planeten2.txt 2
```

Zusammenfassung heute

`ls -l`	Dateirechte anzeigen
`chmod, chown`	Dateirechte ändern
`umask`	Dateirechte voreinstellen
`bc`	einfacher Taschenrechner
`>, >>, <`	Aus-/Eingabe umleiten
`sort`	Sortieren
`;`	mehrere Befehle in einer Zeile trennen
`|`	"Pipe", Ausgabe des letzten Befehls als Eingabe des nächsten Befehls nehmen
`cat`	Aneinanderhängen
`Strg-c`	Programm abbrechen
`echo`	Argument ausgeben
`head, tail`	Anfang/Ende einer Datei anzeigen
Shellskripte	`skript.sh`

Ende der heutigen Vorlesung

Sie probieren das alles in den Tutorien aus.

Viel Spaß!