Unix & Shell-Programmierung SS21

Vorlesungswoche 4

Helga Karafiat

FH Wedel

Prozesse und Subshells

- Prozess
  - Vereinfachte Definition:
    Programm, welches im Speicher zur Ausführung geladen ist
  - Besitzt Kontext, z.B. Variablen, Zustandsinformationen, Informationen fürs System (pid) etc.
  - Prozesse können Unterprozesse erzeugen (Elternprozess, Kindprozess)
  - Elternprozess wartet auf die Beedingung des Kindprozesses
- Shells / Subshells
  - Jede Shell ist ein eigener Prozess
  - Kindprozesse von Shells werden Subshells genannt
  - Subshells verfügen über den gleichen globalen Kontext wie Elternshell
  - Subshells können den Kontext (Variablen, Zustand etc.) der Elternshell nicht ändern

Befehlsverknüpfungen

- `cmd1 | cmd2`
  - verbindet zwei Befehle (`stdout -> stdin`)
  - Beispiel `ls *.mp3 | wc -l`
- `cmd1 ; cmd2`
  - führt Befehle innerhalb einer Zeile nacheinander aus (ersetzt den Zeilenumbruch)
  - z.B. `echo "Verzeichnisinhalte:" ; ls -1`
- `{ list ; }`
  - Fasst die Befehle aus `list` zusammen (Priorisierungsklammern)
  - Befehle werden weiterhin innerhalb der selben Shell ausgeführt
  - z.B. `{ ls *.mp3 ; ls new/*.mp3 ; } | wc -l`
  - Vorsicht: Priorisierungsklammern werden nur nach Zeilenumbruch, Semikolon oder Schlüsselwort erkannt

- `( list )`
  - führt die Befehle aus `list` in einer Subshell aus (Subshellklammern)
  - Keine Seiteneffekte: Kontext (Zustand, Variablen) der Elternshell wird nicht verändert
  - Nützlich bei Verzeichniswechsel oder anderer Zustandsveränderung und um Variablen ohne Seiteneffekt neu zu belegen
  - z.B. `( cd new ; ls *.mp3 ) ; ls *.mp3`
  - z.B. `{ ls *.mp3 ; ( cd new ; ls *.mp3 ) } | sort | uniq -c`
  - Vorsicht: Subshells sind relativ teuer

Befehlsersetzung

- Shell bietet die Möglichkeit Programmaufrufe durch ihren Wert zu ersetzen, z.B.
  - um Befehle dynamisch auszuwerten
  - einer Variablen die Ausgabe eines Programms zuzuweisen
  - um Programme, die nicht von stdin lesen, mit generierten Eingaben zu versorgen
- Syntax: `` `cmd1` `` (Backticks) oder `$(cmd1)`
- Einfaches Beispiel:
  `echo Ein` `` `echo Hallo` `` bzw.
  `echo Ein $(echo Hallo)`
  - zunächst wird `echo Hallo` ausgeführt
  - dann der Aufruf von `echo Hallo` durch sein Ergebnis, also "Hallo", ersetzt
  - dann wird der zusammengesetzte Befehl `echo Ein Hallo` ausgeführt

- Vorteil der `$(...)` Schreibweise:
  - Schachtelung und Quoting gut lesbar möglich
  - z.B.
    `echo outer $(echo inner1 $(echo inner2) inner1)`
    im Vergleich zu
    `echo "outer `echo inner1 \`echo inner2\` inner1`"`
- Anwendungsbeispiel: Liste mit Beschreibung aller installierten Programme
  - Kombination aus `whatis` und `ls /bin`
  - Problem: `whatis` liest Eingaben nur von der Kommandozeile (nicht von `stdin`)
  - Lösung: `whatis `ls /bin``
  - Vorsicht bei sehr langen Eingaben:
    die Länge der Kommandozeile ist beschränkt und wird einfach abgeschnitten
    Abhilfe schafft die Verwendung von `xargs`,
    z.B.: `ls /bin /usr/bin | xargs whatis`

Datenströme Teil 2 - Umleitungen

- Häufiger ist es sinnvoll oder notwendig die Standardausgabe und die Standardfehlerausgabe getrennt zu behandeln, einzelne zu verwerfen oder zusammenzufassen
- Dazu notwendig: Filedeskriptoren (siehe Skript 1):
  - 0: `stdin`
  - 1: `stdout`
  - 2: `stderr`
- Verwerfen eines Datenstroms
  - Umleiten nach `/dev/null` (spezielles Device, dass "ins Nichts" schreibt)
  - z.B. Fehlerausgabe verwerfen `ls nichtvorhanden 2> /dev/null`
- Ausgabe und Fehler in getrennte Dateien schreiben
  - Mehrere Umleitungen pro Aufruf sind möglich
  - z.B. `ls da* nichtda* 1> vorhanden.txt 2> nichtvorhanden.txt`
  - Erinnerung: bei `1>` ist die 1 nicht zwingend notwendig, erhöht aber ggfs die Lesbarkeit

- Mischen von Datenströmen
  - Syntax: `m>&n`
  - z.B. `stderr` mit auf `stdout` schreiben `ls da* nichtda* 2>&1`
- Mischen und gleichzeitiges Umleiten ist möglich
  - Umleiten von `stdout` und `stderr` in eine Datei
    `ls da* nichtda* 1> ausgabe 2>&1`
    - `1> ausgabe`: Dateideskriptor 1 (`stdout`) wird zu Datei `ausgabe`
    - `2>`: leitet Dateideskriptor 2 (`stderr`) um
    - `&1`: Notation für "wo auch immer Dateideskriptor 1 ist"
    - Vorsicht: Zwischen `2>&1` dürfen keine Leerzeichen stehen
  - Reihenfolge der Angaben ist entscheidend, Auswertung von rechts nach links
    - `ls da* nichtda*  2>&1 1> ausgabe`
    - `2>&1`: leitet die Ausgabe von `stderr` aufs Terminal (wo 1 zu dem Zeitpunkt noch ist)
    - `1> ausgabe`: Leitet `stdout` in die Datei `ausgabe` um

Exitcodes

- Analog zu `stdout` und `stderr` Möglichkeit den Programmzustand beim Beenden über einen Exitcode zu signalisieren
- Exitcode des zuletzt aufgerufenen Programms wird in `$?` (Shell-Variable) gespeichert, Abfrage über `echo $?`
- Semantik
  - = 0: Alles ist gut
  - > 0: Fehler
  - Hintergrund: Es gibt nur einen Zustand ok, aber viele mögliche Fehler.
  - Verschiedene Fehler werden über unterschiedliche Exitcodes signalisiert
- Skript hat als Exitcode den Exitcode des letzten ausgeführten Befehls
- Explizites Setzen des Exitcodes mit `exit EXITCODE` möglich

- Exitcodes (Konvention):
  - 1–125: zur freien Verwendung / Definition
  - 126: Befehl gefunden, Datei war nicht ausführbar
  - 127: Befehl nicht gefunden
  - > 128: Befehl ist aufgrund eines empfangenen Signals “gestorben”

Bedingte Ausführung von Befehlen

- Bedingte Ausführung von Befehlen anhand von Exitcodes
- Shells kennen verkürzte boolesche Auswertung auch bei Verknüpfungen von Befehlen
  - `cmd1 && cmd2`
    `cmd2` wird nur ausgeführt, wenn `cmd1` erfolgreich war
    (Exitcode `cmd1` $= 0$)
  - `cmd1 || cmd2`
    `cmd2` wird nur ausgeführt, wenn `cmd1` nicht erfolgreich war
    (Exitcode `cmd1` > 0)
- Beispiele:
  - `cd new && ls *.mp3`
  - `cd new || echo "Could not change directory"`
- Tiefere Schachtelung mit `if - elif - else - fi` möglich

- Schreibweise bedingte Anweisungen:

```
if AUSDRUCK1
then
  ...
elif AUSDRUCK2
then
  ...
else
  ...
fi
```

- Mehrere `elif`-Abschnitte möglich
- `elif`- und `else`-Abschnitt sind optional (müssen nicht angegeben werden)
- Zeilenumbrüche sind signifikant, können aber durch `;` ersetzt werden

- Beispiel:
  Erst versuchen die Namen aller mp3s in new auszugeben, falls new nicht existiert, die aus other und falls beide nicht exisitieren eine entsprechende eigene Fehlermeldung

```
if cd new 2>/dev/null
then
  ls *.mp3
elif cd other 2>/dev/null
then
  ls *.mp3
else
  echo "Sorry, neither new nor other exist!"
fi
```

Formulieren von Bedingungen mit `test`

- spezielles Tool zum Überprüfen von Bedingungen (`/usr/bin/test`)
- erhält seine Eingaben als Argumente auf Parameterposition
- signalisiert das Ergebnis der angeforderten Auswertung über seinen Exitcode (0 wahr, 1 falsch)
- keine Ausgaben auf `stdout` oder `stderr`
- drei Aufgabenbereiche
  - Vergleiche von Zeichenketten
  - Ganzzahlvergleiche
  - Überprüfung von Dateieigenschaften

- Einige nützliche Operatoren (Rest: `man test`):

Operator	Wahr falls, ...
S1 = S2	die Zeichenketten S1 und S2 gleich sind
S1 != S2	die Zeichenketten S1 und S2 nicht gleich sind
N1 -eq N2	die Ganzzahlen N1 und N2 gleich sind
N1 -ne N2	die Ganzzahlen N1 und N2 nicht gleich sind
N1 -lt N2	N1 kleiner als N2 ist
N1 -gt N2	N1 größer als N2 ist
N1 -le N2	N1 kleiner gleich N2 ist
N1 -ge N2	N1 größer gleich N2 ist
-d DATEI	DATEI ein Verzeichnis ist
-e DATEI	DATEI existiert
-r DATEI	DATEI existiert und lesbar ist
-w DATEI	DATEI existiert und beschreibbar ist
-x DATEI	DATEI existiert und ausführbar oder durchsuchbares Verzeichnis ist
! AUSDRUCK	AUSDRUCK zu falsch auswertet (! negiert den dahinter stehenden Testausdruck)

- POSIX-Algorithmus für `test`

Args	Argumentwerte	Ergebnis
0		falsch (1)
1	Falls $1 nicht leer ist Falls $1 leer ist	wahr (0) falsch (1)
2	Falls $1 ein ! ist Falls $1 ein unärer Operator ist Alles andere	Negiert das Ergebnis des Tests des einzelnen Arguments $2 Ergebnis des Tests des Operators Nicht spezifiziert
3	Falls $2 ein binärer Operator ist Falls $1 ein ! ist Alles andere	Ergebnis des Tests des Operators Negiert das Ergebnis des Tests des doppelten Arguments $2 $3 Nicht spezifiziert
4	Falls $1 ein ! ist Alles andere	Negiert das Ergebnis des Tests des dreifachen Arguments $2 $3 $4 Nicht spezifiziert

- Beispiele
  - `test vier != sechs && echo "vier ist nicht sechs!"`
  - `test 4 -ne 6 && echo "4 ist nicht 6!"`
  - `test -x new && ( cd new ; ls *.mp3 )`
- Elegantere Kurzschreibweise `[ ... ]` statt `test ...`
  - `test` wird hierbei durch `[` ersetzt
  - `[` erwartet als letzten Parameter immer `]`
  - Daraus ergibt sich: Nach `[` und vor `]` muss jeweils Whitespace sein!
  - Beispiele:
    - `[ vier != sechs ] && echo "vier ist nicht sechs!"`
    - `[ 4 -ne 6 ] && echo "4 ist nicht 6!"`
    - `[ -x new ] && ( cd new ; ls *.mp3 )`

- Verknüpfungen von mehreren Tests
  - unschön:
    - `test` bietet die Möglichkeit mehrere Tests intern zu verknüpfen: `-a` für `&&` und `-o` für `||`
    - z.B. `if [ -e new -a -d new ] ...` (Wenn `new` existiert und ein Verzeichnis ist ...)
    - POSIX empfiehlt diese Option aus Gründen der Portabilität zu vermeiden
  - besser:
    - Verknüpfung auf Shell-Ebene
    - z.B. `[ -e new ] && [ -d new ]`
    - weiterer Vorteil: bessere Lesbarkeit
- Schwächen von `test`
  - Zahlvergleiche sind nur auf Ganzzahlen möglich
  - Eingabe auf Parameterposition ist anfällig für Fehler (z.B. fehlender Whitespace)
  - Vergleiche von Zeichenketten können `test` durcheinander bringen, z.B. wenn die Zeichenkette leer ist oder mit - beginnt
    Lösungskonvention: Voranstellen von `X`, also z.B.
    `if [ "X$FOO" = "XInhalt" ] ...`

- Besonderheiten
  - die meisten Shells implementieren das `test`-Kommando aus Effizienzgründen intern
  - die `bash` verfügt über eine erweiterte built-in Variante von `test`
    - wird aufgerufen mit `[[ .... ]]`
    - bietet z.B. erweiterte Stringvergleiche, wie
      =~, welches eine Zeichenkette auf einen regulären Ausdruck "matcht"
      oder < und >, die Zeichenketten lexikographisch vergleichen
    - in der Manpage unter `[[` bzw. `CONDITIONAL EXPRESSIONS`
    - Vorsicht: nicht POSIX-konform!