Unix & Shell-Programmierung SS21

Vorlesungswoche 2

Helga Karafiat

FH Wedel

Mein erstes Skript revisted

- Neue Anforderungen
  - Nur noch die Anzahl der zu erledigenden TODOs ausgeben
  - Groß- und Kleinschreibung bei TODOs völlig egal: TODO, todo, ToDo, Todo, ...

counttodo.sh (Erste Version)

```
#!/bin/sh
# Zaehlt die vorhandenen TODOs in allen Dateien mit Dateiendung

grep --ignore-case TODO *.* > tmp

wc --lines < tmp

rm tmp
```

- Weiterhin unschön: Zwischenspeicherung in temporäre(r/n) Datei(en)
- Probleme:
  - Ausgabe darf nicht gleichzeitig Eingabe sein (`grep TODO * > tmp`)
  - Dateien sind schnell mal unbedacht überschrieben bzw. gelöscht (`sort tmp > tmp`)
  - Wird schnell unübersichtlich (`tmp`, `tmp2`, `tmp3`, `...`)
  - Speichern von Daten zur Zwischenablage aus dem schnellen Arbeitsspeicher auf die langsame Festplatte
- Idee: Direktes Verbinden von Ausgaben und Eingaben wäre schick :)

Pipe (Anonyme Pipe)

- vom Betriebssystem bereitgestellter unidirektionaler Kommunikationskanal
- verbindet zwei Programme (Filter) miteinander: `stdout -> stdin`
- Designprinzip: "simple & elegant"
- Schreibweise: `cmd1 | cmd2`
- Eigenschaften
  - Pufferspeicher nach dem FIFO-Prinzip (first in, first out)
  - Verwendung des Kernel-Speicherpuffers (Arbeitsspeicher)
  - Ausführung der beiden Programme erfolgt parallel, System kümmert sich um Zeitverteilung und Synchronisation
  - Pipe existiert so lange beide Programme leben:
    - ★ Wenn schreibendes Programm endet, wird Pipe geschlossen und lesendes Programm läuft zu Ende
    - ★ Wenn lesendes vor schreibendem Programm endet, wird schreibendes Programm terminiert (meistens Fehlerfall, ab und an "by design" durch z.B. `head` als lesendes Programm)
  - Programme selbst "wissen nichts" von der Umleitung

Filter

"Any program that reads and writes textual data, one line at a time, reading from standard input and writing to standard output. As a general rule, most filters are designed as tools, to do one thing well." (Harley Hahn's Guide to Unix and Linux)

- Definition: Programm welches Eingabe zeilenweise von `stdin` liest und Ausgabe zeilenweise auf `stdout` schreibt
- Paradigma: nur eine bestimmte Sache machen, die dafür aber besonders gut
- Viele bereits bekannte Unix-Tools sind Filter (`cat`, `grep`, `tr`, `wc`, `head`, `tail`, `more`, `less` ...)
- Einfachster Filter: `cat` (von "catenate - verketten"), neutrales Element bezüglich Pipes: → `echo Foo | cat` liefert das selbe Ergebnis wie `echo Foo`

Pipelines

- Verkettung von beliebig vielen Befehlen (Filtern) durch Pipes
- Start und Ende einer Pipeline muss kein Filter sein (dazwischen schon)
- Einige häufiger verwendete Filter:
  `cat`, `sort`, `grep`, `cut`, `tac`, `rev`, `tr`, `uniq`, `sed`, `perl`, `head`, `tail`, `wc`, `sh`, `bash`, ...
- Einige häufig verwendete Anfangsstücke:
  - Programme, die Ausgaben machen, wie `ls`, `who`, `echo`, `date`, ...
  - Programme, die aus Dateien lesen, z.B. `cat`, `grep`, ...
- Besonderer Filter: `tee`
  - Schreibt in Datei(en) und auf `stdout` (T-Stück)
  - Syntax: *tee [FILE]...*

Benannte Pipes

- eher selten benutztes Konzept
- persistenter FIFO-Speicher (siehe `man fifo`)
- spezielle Form von Datei, die keine Inhalte auf dem Dateisystem hat, sondern vom Kernel verwaltet wird
- zur Kommunikation von zwei beliebigen voneinander unabhängigen Programmen bzw. Kommuniktion zwischen Programmen in verschienden Shells (anonyme Pipe: beide Programme werden in der gerade laufenden Shell ausgeführt)
- Kann erst beschrieben werden, wenn auch gelesen wird (bis dahin wird schreibender Prozess blockiert)
- Verwendung:
  - Anlegen mit `mkfifo`, Syntax: *mkfifo pipe_name*, z.B. `mkfifo mypipe`
  - Löschen mit `rm`
  - Müssen explizit beschrieben und gelesen werden:

    ```
    cmd1 > mypipe
    cmd2 < mypipe
    ```

Mein erstes Skript revisited (again)

- Lösung ohne temporäre Datei?

counttodo.sh (Zweite Version - Pipe statt temporäre Datei)

```
#!/bin/sh

grep --ignore-case TODO * | wc --lines
```

Aufruf `./counttodo.sh`

- Können Skripte auch von `stdin` lesen?

`counttodo.sh` (Dritte Version - Lesen von `stdin`)

```
#!/bin/sh

grep --ignore-case TODO | wc --lines
```

Aufruf z.B. `cat * | ./counttodo.sh`

Variablen in der Shell - Grundlagen

- Name-Wert-Paare, Werte sind immer Zeichenketten
- Bennenung `[a-z,A-Z,_][a-z,A-Z,0-9,_]*`
  Konvention: nur Großschrift, also `[A-Z,_][A-Z,0-9,_]*`
- Erstellung implizit durch Deklaration / Zuweisung
  - Syntax: *name*=*wert*
  - z.B. `SOMETHING=foo`
- Adressierung des Inhaltes über `$VARNAME` bzw. `${VARNAME}`
  ({} optional, aber manchmal notwendig)
  - z.B. Ausgabe des Inhaltes mit `echo $SOMETHING`
  - oder z.B. Ausgabe des Inhaltes als Teilausdruck `echo ${SOMETHING}_hallo`
- Löschen über *unset name*, z.B. `unset SOMETHING`
- Anzeige aller in der aktuellen Shell vorhandenen Variablen mit `set` möglich
  (Beschränkung auf Variablen: `set -o posix`, Zurücksetzen mit `set +o posix`)
- Vorsicht: Zugriff auf nichtvorhandene Variable ist kein Fehler,
  sondern Inhalt ist einfach leer (böse Falle!)

Variablen in der Shell - Sichtbarkeitsbereiche

- Unterscheidung von Shell-Variaben (“lokal” verfügbar) und Umgebungs-Variablen (“global” verfügbar)
- Shell-Variablen
  - enstehen durch Deklaration (default)
  - nur in der Shell verfügbar, in der sie deklariert wurden
- Umgebungs-Variablen
  - durch explizite Anpassung des Sichtbarkeitsbereiches einer Variablen per `export`
  - werden an Kindprozesse (Subshells) vererbt
  - Anzeige aller Umgebungsvariablen per `printenv`
  - Einige interessante Umgebungsvariablen: `$HOME`, `$PWD`, `$EDITOR`, `$PATH`, ...

Besondere Variablen (Verwendung in Shell-Skripten)

Einige implizit von der Shell beim Aufruf gesetzte Variablen:

- `$1 .. $n` Erstes bis n-tes Kommandozeilenargument
  (mehrstellige Zahlen müssen mit `{}` geklammert werden, z.B. `${10}`)
- `$#` Anzahl der Kommandozeilenargumente
- `$*` Alle Kommandozeilenargumente als eine Zeichenkette,
  z.B. geeignet um alle Parameter auf einmal auszugeben
- `$@` Alle Kommandozeilenargumente als separate Zeichenketten,
  z.B. geeignet zur Weitergabe an andere Programme
- `$0` Name des aktuellen Shellprogramms
- `$$` Prozessnummer

Quoting

- Shell kennt viele Metazeichen, also Zeichen, die eine besondere Bedeutung haben: Leerzeichen, `*`, `?`, `[`, `]`, `>`, `>>`, `<`, `|`, `&`, `$`, `(`, `)`, `;`, `{`, `}`, `'`, `"`, `\`, `` ` ``
- Metazeichen werden im Normalfall von der Shell ausgewertet:
  - z.B. `echo *`
  - oder `echo $FOO`
  - oder `echo Hallo  Welt` (viele Leerzeichen)
- Was tun wenn man Metazeichen nicht auswerten, also z.B. einfach ausgeben will
- Lösung: man teilt es der Shell mit (aka Quoting)

Drei Möglichkeiten für Quoting:

- Backslash
  - maskiert ein einzelnes Zeichen, aber dafür auch wirklich jedes
  - z.B. `echo \*` oder `echo Hallo\ \ \ \ Welt`
  - Kann auch Zeilenumbrüche maskieren, z.B. um Pipelines mehrzeilig zu schreiben
  - Nachteil: bei vielen zu maskierenden Zeichen schnell unübersichtlich
- Einfache Anführungszeichen (strong quotes)
  - Alles dazwischen ist gequotet (wird genau so genommen wie es da steht)
    z.B. `echo 'ganz viele Zeichen: [*, ?, >>, |, $, $FOO ]'`
  - Einfache Anführungszeichen können nicht in einfachen Anführungszeichen stehen
- Doppelte Anführungszeichen (weak quotes)
  - Dazwischen ist alles gequotet, außer: `$`, `\`, `` ` ``
  - Maskierung mit `\` möglich
  - Auswertung von Variablen findet statt, z.B. `echo "<\$WELT>: <$WELT>"`

Mein erstes Skript revisited (again and again)

- Suchwort über Kommandozeilenparameter angegeben
- Vorsicht: Quotingzeichen werden beim Auswerten durch die Shell konsumiert
- Lösung: Variablen in Skripten immer in doppelte Anführungszeichen schreiben, außer man will explizit etwas anderes

countphrase.sh

```
#!/bin/sh

grep "$1" * |  wc --lines
```

Aufruf z.B. `./countphrase.sh "Hallo Welt"`