1. UNIX-Tools

In diesem abschliessenden Kapitel sollen einige Anregungen aus der Praxis vermittelt werden. Der Kurs wird schliesslich mit der Betrachtung einiger UNIX-Tools wie *grep*, *sed*, *tr*, *find* und der Vorstellung einiger Public Domain Tools abgerundet. Ein kurzer Ausblick in die UNIX-Zukunft soll verdeutlichen, dass UNIX kein «totes» Betriebssystem ist, sondern von verschiedenen Firmen, Gremien, Universitäten und Konsortien weiterentwickelt wird.

1.1 Oft benutzte Utilities

1.1.1 *find* (find files by name, or by other characteristics)

Das Kommando find(1) ist eines der unter UNIX am häufigsten benutzten Utilities. Es durchsucht das Filesystem, ausgehend von einem Startpunkt, rekursiv nach Filenamen, welche nach bestimmten Kriterien ausgewählt werden. Die häufigsten Anwendungen von *find* sind:

- Generieren einer Liste von Files zur Datensicherung
- Durchforsten des Filesystems nach einem «verlorenen» File
- Durchsuchen des Fileinhalts nach einem bestimmten Wort oder Pattern

Directories in Filesystem auflisten

```
$ find . -type d -print
```

Alle Files mit der Endung (.o) in Filesystem löschen

```
$ find . -name "*.o" -exec rm -i {} \;
```

Suchen aller Files die jünger sind als ein Vergleichsfile

```
$ find . -newer <vergleichsfile> -print
```

Löschen aller "core" und "*.BAK" Files

```
$ find . (-name core -o -name '*.BAK') -exec rm -f {} \;
```

Suchen aller Files die in den letzten 7 Tagen verändert wurden

```
$ find . ! -mtime +7 -print
```

Suchen aller Files die in den letzten 7 Tagen NICHT verändert wurden

```
$ find . -mtime +7 -print
```

Suchen aller Files welche "pattern" enthalten

```
$ find . -type f -exec grep -l "stdio.h" {} \;
  ./Xm/examples/dogs/Dog.c
  ./Xm/examples/dogs/Square.c
  .........
```

Suchen aller Files welche "pattern" enthalten mit Angabe der Zeile

```
$ find . -type f -exec grep "stdio.h" /dev/null {} \;
  ./Xm/examples/dogs/Dog.c:#include <stdio.h>
  ./Xm/examples/dogs/Square.c:#include <stdio.h>
```

1.1.2 *sed* (Stream Editor)

Sed ist ein sehr mächtiges Utility um den Inhalt eines Files «im Hintergrund» zu verändern. Das heisst die Datei wird nicht interaktiv bearbeitet, wie dies bei jedem Editor wie *vi, emacs* etc. der Fall ist. *Sed* ist sehr eng mit dem Gebrauch von regulären Ausdrücken (Regular Expressions) gekoppelt. *Regular Expressions* basieren auf dem sogenannten *Pattern-Matching* Mechanismus. Das File wird auf ein "Pattern" durchsucht, wird ein *Match* (Übereinstimmung) gefunden, so wird die Zeile gemäss den sed-Angaben verändert. Da *sed* ein äusserst mächtiges Tool ist kann im Rahmen dieses Kurses nur ein kurzer Abriss vermittelt werden.

Übersicht der Verarbeitung

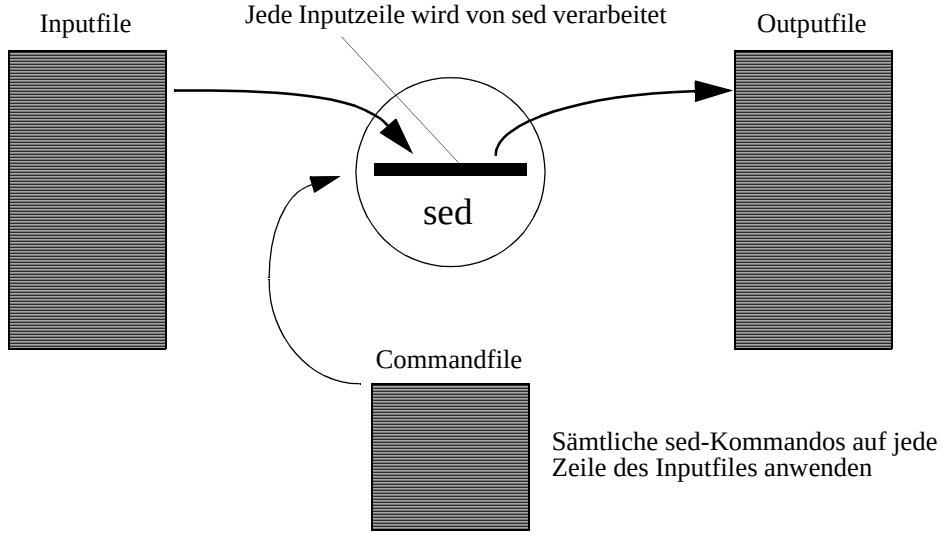


Abb. 6-1 Funktionsweise von sed

Syntax von *sed*

```
sed [-n] [-e command] [-f commandfile] [inputfile]
```

Optionen:

-n Schreibt nur die effektiv veränderten Linien auf stdout
-e Nächstes Argument ist ein sed-Kommando. Damit können mehrere sed-Kommandos auf der Eingabezeile spezifiziert werden.
-f Name des Files mit sed-Kommandos

Aufbau von command

```
[address [, address] ] function [argument] [flag]
```

```
$ sed -e "1,30s/old/new/g" inputfile
```

Aufbau von address

20,30	Von Zeile 20 bis 30
1,$	Von Zeile 1 bis ans Ende
keine Zeilennummer	Auf gesamtes File anwenden

Sed-Functions

`d`	Zeile löschen
`s/old/new/`	Zeile mit "old" durch "new" ersetzen
`p`	Zeile ausgeben

Beispiele

Erste Zeile einer Datei herausschneiden

```
$ sed -n "1,1p" /etc/passwd     (p = print)
```

Im File /etc/passwd Einträge */bin/csh* durch */bin/ksh* ersetzen

```
$ sed "s/\/bin\/csh/\/bin\/ksh/g /etc/passwd
```

Herausschneiden der Login-Namen aus /etc/passwd

```
$ sed -e "1,\$s/:.*//" /etc/passwd
```

Im ganzen File (1,\$) alles (.*) was rechts von : steht durch nichts (//) ersetzen. Die Adresse (1,\$) könnte auch weggelassen werden.

Mit ls-Kommando nur reguläre Files anzeigen

```
$ ls -l | sed -n "/^-/p"
```

Mit ls-Kommando nur Subdirectories anzeigen

```
$  ls -l | sed -n "/^d/p"
```

Sämtliche "kurs.." User aus /etc/passwd entfernen

```
$ sed "/^kurs.*/d" /etc/passwd
```

Output von *banner* in Terminal-Mitte placieren

```
$ banner "Hello" | sed -e "s/^/<TAB><TAB>/"

        #     #
        #     #  ######  #       #        ####
        #     #  #       #       #       #    #
        #######  #####   #       #       #    #
        #     #  #       #       #       #    #
        #     #  #       #       #       #    #
        #     #  ######  ######  ######   ####
```

Script "killbyname"

Damit können Processe bequem mittels Process-Namen gestoppt werden.

Beispiel: *killbyname xterm*

Damit werden alle xterm’s gestoppt

```
#!/bin/sh
kill -9 `ps -ax | grep $1 | sed -e "s/^ *//" | sed -e "s/ .*//"`
```

Zuerst werden am Anfang jeder Zeile alle Leerzeichen entfernt (s/^ *), dann wird alles nach dem ersten Leerzeichen gelöscht (s/ .*//). Damit verbleiben die PID’s welche dem Kommando *kill* übergeben werden.

Alle Leerzeilen in File löschen

```
$ sed -e '/^ *$/d' file1 > file2
$ sed -e 's/^$//' file1 > file2
```

Blanks am Ende einer Zeile löschen

```
$ sed -e 's/ *$//' file1 > file2
```

1.1.3 *expr* (Evaluate arguments as a logical, arithmetic, or string expression)

Expr(1) wird oft im Zusammenhang mit der Kommandosubstitution verwendet. Argumente zu *expr* werden als Ausdruck verwendet und von *expr* ausgewertet. Expr verwendet die folgenden Operatoren:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo-Funktion
:	Vergleich von Strings

Beispiele

Hochzählen einer Variablen

```
$ index=‘expr $index + 1‘
```

Filename aus Path extrahieren

```
$ a=$HOME/filex
$ expr //$a : '.*/\(.*\)'
  filex
```

The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

Länge eines Strings bestimmen

```
$ expr $HOME : '.*'
```

1.1.4 *comm* (Display common lines in two sorted Files)

comm wird benutzt um gemeinsame Zeilen in zwei verschiedenen Files aufzufinden

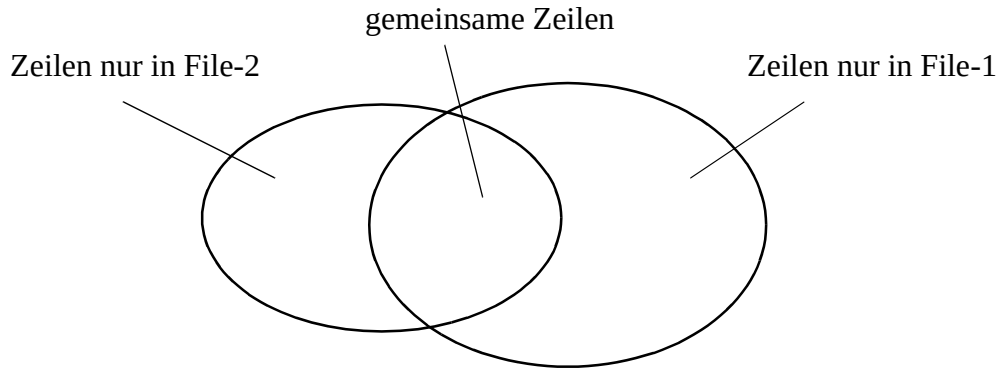


Abb. 6-2 Funktionsweise von *comm*

```
$ comm file-1 file-2
  Zeile nur in file-1
            Zeile in beiden Files
                      Zeile nur in file-2
```

Zeilen, welche nur im *file-1* enthalten sind, stehen in der ersten Spalte, Zeilen welche in beiden Files vorhanden sind stehen in der mittleren Spalte und Zeilen welche nur im *file-2* vorhanden stehen in der dritten Spalte.

1.1.5 *diff* (Display differences in two files)

Oft interessiert nicht der gemeinsame Fileinhalt, sonden der unterschiedliche. Mit *diff* können alle Arten von Files verglichen werden, mit Ausnahme von Directories. Unterschiede werden mit einem «<» bzw. «>» gekennzeichnet.

< kennzeichnet das erste File (linkes Argument)

> kennzeichnet das zweite File (rechtes Argument)

Um eine Übereinstimmung zu erzielen werden zur Hilfe drei Flags angegeben:

a: Addiere die folgende Zeile(n) zum File

c: Ändere die folgenden Zeile(n) im File

d: Lösche die folgenden Zeilen(n) im File

Beispiel:

```
$ diff old.c new.c
  10c10
  < getenv()
  ---
  > getpwnam()
```

Ändere in *old.c* die Linie 10 von "getenv()" auf "getpwnam()" um eine Übereinstimmung mit *new.c* zu erhalten.

```
  27a28
  > /* -------- */
```

Füge in *old.c* auf der Linie 27, die Zeile /* -------- */ ein, welche sich in *new.c* auf der Linie 28 befindet, um eine Übereinstimmung zu erreichen.

Versionsaktualisierung mit *diff* und *ed*

Das Utility *diff* wird sehr oft dazu verwendet eine automatische Versionsaktualisierung beim Versenden von Software zu ermöglichen. Mit der Option -e kann ein *ed* Script generiert werden, mit dessen Hilfe die bereits versendeten Dateien auf den neusten Stand zu bringen. Damit müssen nicht gesamte, geänderte Softwarepakete versendet werden, sondern nur die *ed*-Scripts um die bereits versendeten Sourcen zu aktualisieren.

Beispiel: Das C-Programm *old.c* wurde an Kunden versendet. Man stellt später Fehler an diesem Programm fest und generiert das neue Programm *new.c.* Dieses Programm ist sehr gross, so dass die Telephon-Kosten (Modem) sehr teuer zu stehen kommen. Anstelle des gesamten Programms *new.c* versendet man ein durch *diff -e* generiertes *ed*-Script, das *old.c* in *new.c* überführt.

```
$ diff -e old.c new.c > new.ed
$ cat new.ed
  27a
  /* ---------- */
  .
  10c
  getpwnam()
  .
```

Im generierten ed-Script müssen noch die ed-Kommandos *w* zum speichern und *q* zum verlassen des ed Editors eingefügt werden

```
$ vi new.ed
```

Das ed-Script hat nun folgenden Inhalt:

```
$ cat new.ed
  27a
  /* ---------- */
  .
  10c
  getpwnam()
  .
  w
  q
```

Nun wird *old.c* in *new.c* überführt:

```
$ ed old.c < new.ed
  3774
  3791
```

Es wurden 3774 Zeichen gelesen und 3791 gespeichert. Ein erneuter Vergleich zeigt, dass die Files nun übereinstimmen:

```
$ diff old.c new.c
```

1.2 Filter-Kommandos

Filter-Kommandos stehen praktisch immer in Pipes. Sie verändern den Datenfluss von Kommando zu Kommando in einer bestimmten Weise; sie wirken als «Filter». Typische Filter sind *sort, grep, cut, tee, uniq, tr* und weitere.

1.2.1 *sort* (sort and collate lines)

Sort sortiert eine Datei alphanumerisch oder numerisch unter Berücksichtigung des länderspezifischen Zeichensatzes (Vergleich mit Zeichensatz = collate). Standardmässig wird der ASCII-Zeichensatz verwendet. Wird das UNIX-System in der länderspezifischen Version betrieben (NLS-Zusatz), so wird der entsprechende Zeichensatz gewählt.

Sort wird häufig auf tabellenartige Files angewendet. Die Sortierung erfolgt dabei grundsätzlich auf der ersten Spalte.

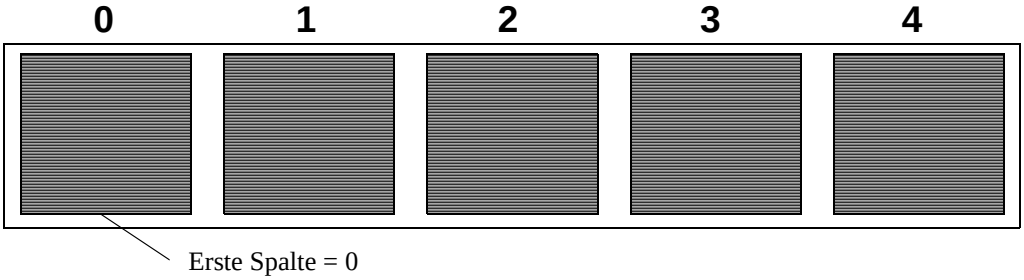


Abb. 6-3 Spalten-Numerierung von *sort*

Beispiele

`$ sort /etc/passwd`	Alphanumerisch nach Login-Namen sortieren
`$ sort -r /etc/passwd`	Reverse sortieren nach Login-Namen
`$ ls -l | sort -n +3`	*ls -l* numerisch nach der Filegrösse sortieren, dies ist unter SUNOS-4.1 das Feld 3
`$ sort -t: +2n -3 /etc/passwd`	Sortiere /etc/passwd nach den UID's. +2n heisst: Sortiere die Spalte Nr. 2 numerisch und stoppe in der Spalte Nr 3. Damit wird nur nach den UID's sortiert.

Datenströme mischen und sortieren

Sort ist in der Lage mehrere Files zu mischen und zu sortieren (sort *file1 file2*). Etwas schwieriger wird es, wenn *sort* seinen Input aus einer Pipe liest, und dieser mit einem File gemischt und sortiert werden soll, wie das folgende Beispiel zeigt.

Beispiel: Der Output von *ls -1* soll mit der Datei *file* gemischt und sortiert werden.

```
$ ls -l          $ cat file       ls -1 | sort - file
  adm              bin              adm
  boot             cnode            bin
  demo             etb              boot
  diag                              cnode
  dict                              demo
  etc                               diag
  export                            dict
  games                             etb
                                    etc
                                    export
                                    games
```

Sort liest *ls -1* via Pipe und speichert intern die gelesenen Daten. Anschliessend wird der Standard-Input mittels "-" explizit für das Argument *files* geöffnet und der Inhalt eingelesen. Dann erfolgt intern die Sortierung der zwei Datenkanäle.

1.2.2 *cut* (remove selected fields from each line of a file)

Mit *cut* können Spalten aus einem tabellenorientierten File herausgeschnitten werden. Im Gegensatz zu sort beginnt die erste Spalte mit "1"

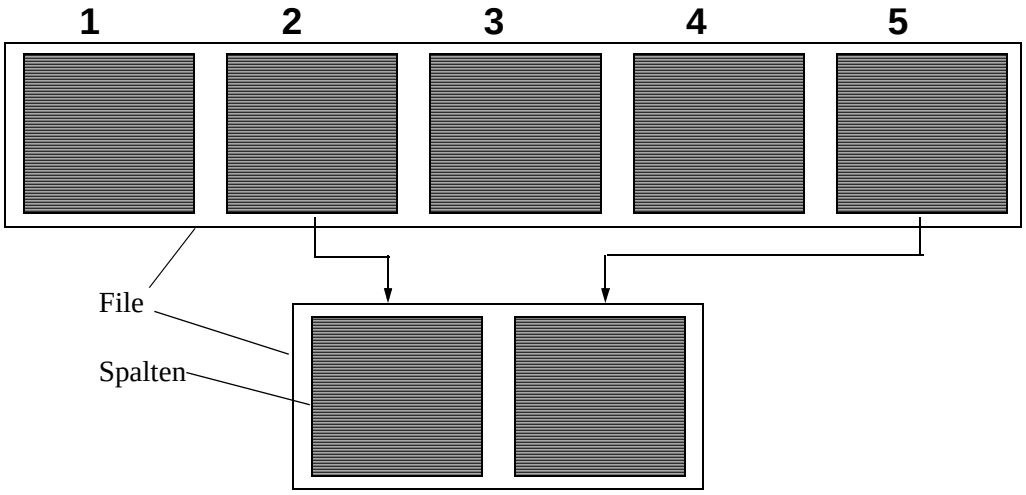


Abb. 6-4 Funktionsweise von *cut*

```
$ cut -d: -f1,5 /etc/passwd
```
Login-Name und wirklicher User Name einander gegenüberstellen

```
$ ls -l | cut -c45-
```
Ab Zeichenposition 45 herausschneiden

1.2.3 *paste* (join corresponding lines of several files)

Paste stellt das Gegenteil von *cut* dar. Einzelne tabellenorientierte Files können zusammengesetzt werden:

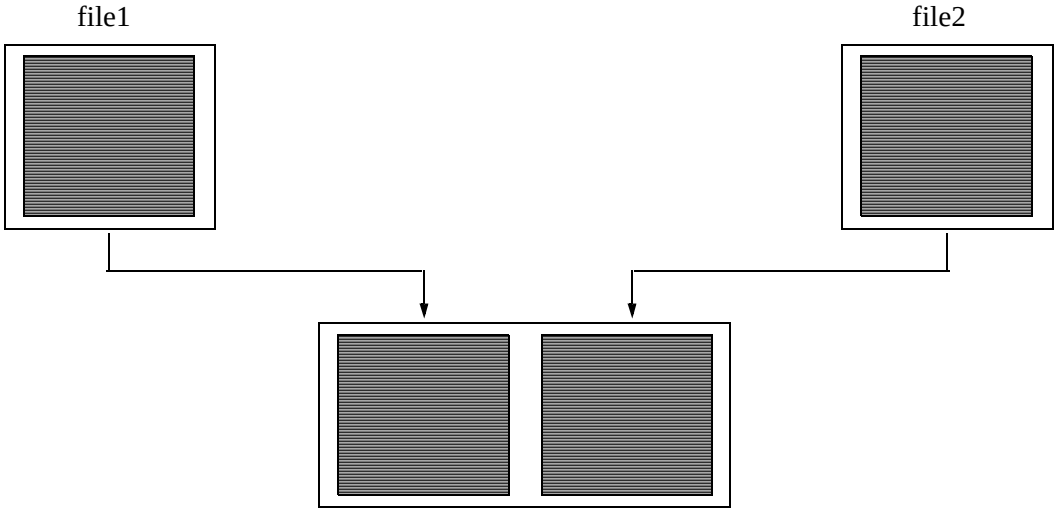


Abb. 6-5 Funktionsweise von *paste*

```
$ paste file1 file2
$ ls /bin | past - - -
```
Output von *ls /bin* 3-Spaltig darstellen. Die drei "-" bedeuten «read from standard input»

1.2.4 *grep* (search a file for a string or regular expression)

Das Utility *grep* durchsucht ein File auf einen String. Wird der String gefunden, so wird die gesamte Zeile des Files auf den Standard Output ausgegeben. Grep wird sehr oft mit regulären Ausdrücken (*regular Expressions*) verwendet. Grep verfügt über einige sehr nützliche Optionen:

-v Display all lines expect those containing pattern.

-c Report only the number of matching lines.

-l List only the names of files containing pattern.

-n Precede each line by the line number in the source file.

Beispiele

Bestimmte Linien aus File entfernen

```
$ ps -ax | grep -v "xterm"
```
Alle Processe anzeigen ausser xterm's

Suchen aller Files welche "pattern" enthalten

```
$ find . -type f -exec grep -l "stdio.h" {} \;
```

Zeilen mit Nummern versehen

```
$ cat source.c | grep -n ".*"
```

Suchen eines Files im gesamten Filesystem

```
$ ls -lR | grep "file"
```

1.2.5 *tee* (replicate the standard output)

Tee ermöglicht es eine Pipe an einem bestimmten Ort "anzuzapfen" und den Standard Output in ein File zu lenken. Damit kann beispielsweise eine Liste der gesicherten Files bei der Datensicherung mittels *cpio* erstellt werden.

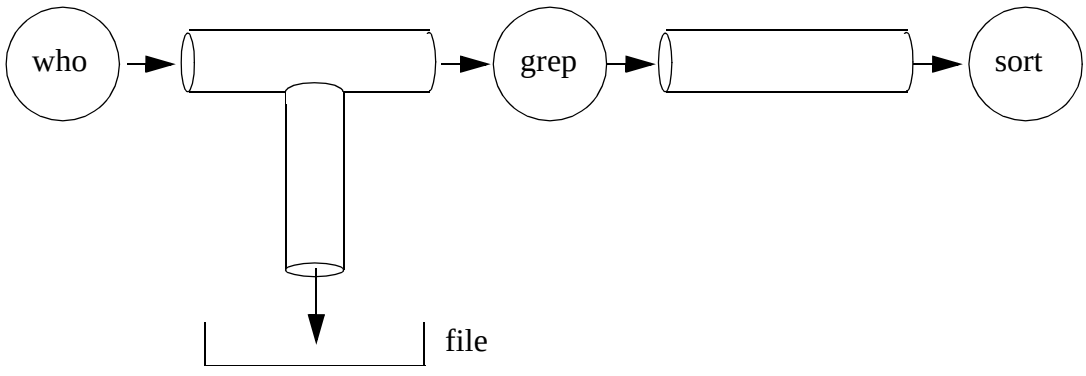


Abb. 6-6 Funktionsweise von *tee*

```
$ who | tee file | grep "pattern" | sort
$ find . -print | tee backuplist | cpio -ocB > /dev/rst0
```

1.3 Regular Expressions

Regular Expressions (reguläre Ausdrücke) werden in verschiedenen Utilities wie *grep, ex, sed* für das *Pattern-Matching* benötigt. Die Regular Expressions sind jedoch nicht in allen Utilities gleich definiert. Die folgende Tabelle stellt die wichtigsten Regular Expressions für die am meisten verwendeten Utilities zusammen.

Reguläre Ausdrücke in verschiedenen UNIX-Utilities

Funktion	sh, ksh	sed, grep	egrep	ed	ex, vi
beliebiges Zeichen	`?`	`.`	`.`	`.`	`.`
beliebige Zeichenkette	`*`	`.*`	`.*`	`.*`	`.*`
beliebige Wiederholung	fehlt	`*`	`*`	`*`	`*`
beliebige Wiederholung (mindestens 1)	fehlt	fehlt	`+`	`\{1\}`	fehlt
keine oder eine Wiederholung	fehlt	fehlt	`?`	`\{0,1\}`	fehlt
n-malige Wiederholung	fehlt	fehlt	fehlt	`\{n\}`	fehlt
n bis m-malige Wiederholung	fehlt	fehlt	fehlt	`\{n,m\}`	fehlt
Zeichen aus	`[...]`	`[...]`	`[...]`	`[...]`	`[...]`
Kein Zeichen aus	`[!...]`	`[^...]`	`[^...]`	`[^...]`	`[^...]`
am Zeilenanfang	fehlt	`^pattern`	`^pattern`	`^pattern`	`^pattern`
am Zeilenende	fehlt	`pattern$`	`pattern$`	`pattern$`	`pattern$`
am Wortanfang	`*xyz`	fehlt	fehlt	fehlt	`\<pattern`
am Wortende	`xyz*`	fehlt	fehlt	fehlt	`pattern\>`

Tabelle 6-1 Reguläre Ausdrücke in UNIX Utilities

Reguläre Ausdrücke im Ersetzungsteil

Funktion	sed, ed	ex, vi	Beispiel
Suchen eines Teilausdrucks	`\(...\)`	`\(...\)`	`s/\(Martin\) \(Zahn\)/\2 \1/`
n-ter Teilausdruck	`\n`	`\n`	`s/\(Martin\) \(Zahn\)/\2 \1/`
gefundene Zeichenkette	`&`	`&`	`s/[0-9][0-9]*/&./`
vorhergehende Ersetzung	fehlt	`~`	

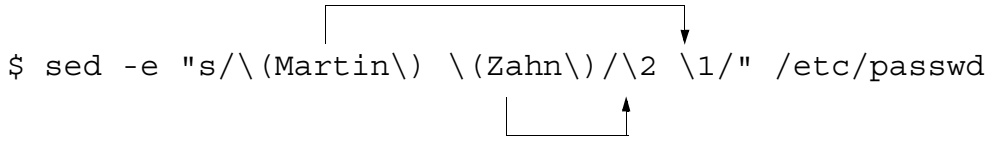
Tabelle 6-2 Reguläre Ausdrücke im Ersetzungsteil

Beispiele

Vertauschen eines Strings

In */etc/passwd* soll "Martin Zahn" vertauscht werden:

```
vorher:     mz:zyvipeOwM3QZ.:101:30:Martin Zahn:/home/mzahn:/bin/sh
nachher     mz:zyvipeOwM3QZ.:101:30:Zahn Martin:/home/mzahn:/bin/sh
```



```
$ sed -e "s/\(Martin\) \(Zahn\)/\2 \1/" /etc/passwd
```

Berechung der vorhandenen Diskkapazität (SUNOS-4.1)

```
#!/bin/sh

echo ""
echo "calculating disk space ..."
echo ""
```

calculate total disk space

```
sum=0
for i in `df | sed "s/.*[a-z][ ]*\([0-9]*[ ]\).*/\1/"`
do
   sum=`expr $sum + $i`
done
echo "total space .......: $sum kbytes"
```

calculate total used space

```
sum=0
for i in `df | sed "s/.*[a-z][ ]*[0-9]*[ ]*\([0-9]*[ ]\).*/\1/"`
do
   sum=`expr $sum + $i`
done
echo "total used ........: $sum kbytes"
```

calculate total available space

```
sum=0
for i in `df | sed "s/.*[a-z][ ]*[0-9]*[ ]*[0-9]*[ ]*\([0-9]*[ ]\).*/\1/"`
do
   sum=`expr $sum + $i`
done
echo "total available ...: $sum kbytes"
```

Ausgabe des Scripts:

```
calculating disk space ...

total space .......: 2055188 kbytes
total used ........: 1000291 kbytes
total available ...: 852292 kbytes
```

Zweispaltige *ls* Darstellung, entfernen von Leerzeilen

```
$ ls | /usr/5bin/pr -2 | grep -v "^$"
```

1.4 Fortgeschrittene Shell-Script Konstruktionen

Der folgende Abschnitt wendet einige interessante Shell-Script Konstruktionen an, welche in der Praxis ab und zu anzutreffen sind.

1.4.1 Pipes und Shell Loops

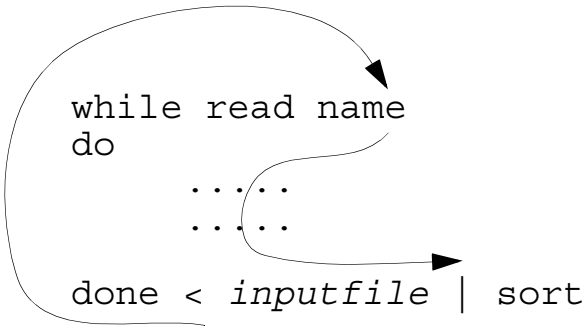
Pipes und Shell-Loops können kombiniert werden wie das folgende Beispiel zeigt.

```
ls /bin |
while read name
do
       echo $name
done
```

While liest direkt aus der Pipe und wertet jeden Filenamen aus.

1.4.2 Shell Loops und I/O-Redirection

While ist in der Lage aus einem File zu lesen und seinen Standard-Output auf eine Pipe zu lenken:



1.4.3 Here-Dokument mit interaktiven Kommandos

In diesem Beispiel werden *ed*-Kommandos in einem *Here-Dokument* gespeichert, welche in den durch *find* bereitgestellten Files «oldstring» durch «newstring» ersetzen.

```
#!/bin/ksh
#
# Replace oldstring with newstring in all files starting from here
#
if [ $# -lt 2 ]
then
     echo "Usage: $0 Oldstring Newstring"
     exit 1
fi

old="$1"
new="$2"

shift 2

find . -exec grep -l "$old" {} \; |
while read name
do
     echo "Replacing >$old< with >$new< in $name"
     ed - $name <<-!
            H
            g/$old/s//$new/g
            w
            q
     !
done
```

1.4.4 Co-Processe (Korn-Shell Erweiterung)

Die Korn-Shell erlaubt es, aus einem Shell-Script einen Background-Process zu starten, mit dem über Standard-Ausgabe und Standard-Eingabe kommuniziert werden kann. Der Hintergrundprocess muss dazu mit «|&» anstelle «&» gestartet werden.

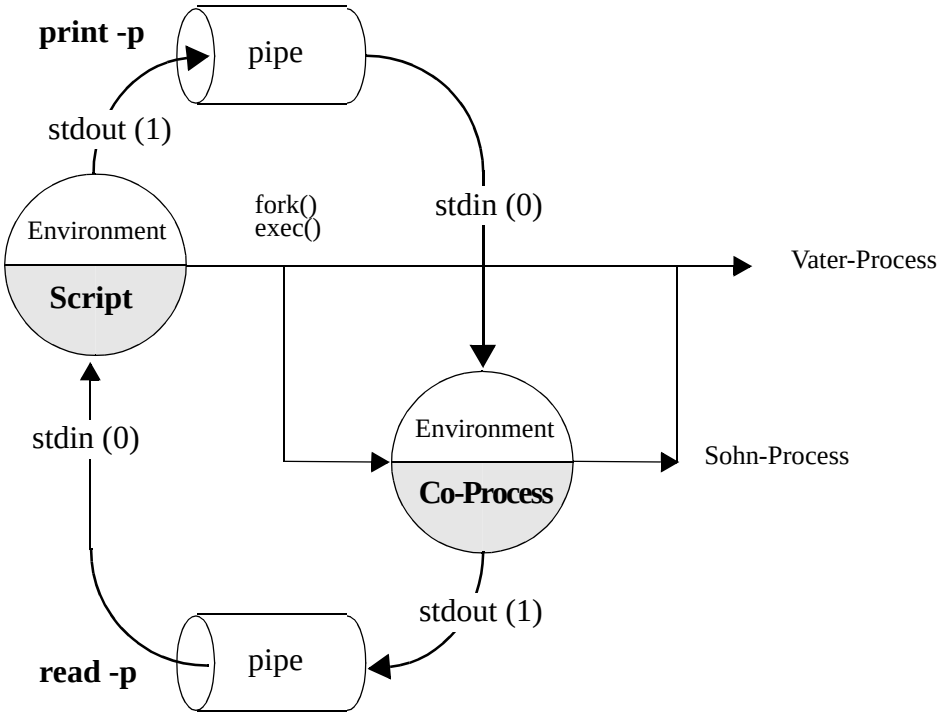


Abb. 6-7 Korn-Shell Co-Processes

Das Shellscript überträgt Daten mittels *print -p* an den Co-Process und liest Daten mittels *read -p* von diesem.

Beispiele

Kommunikation mit interaktivem Filetransferprogramm *ftp*

```
#!/bin/ksh
#
# Anwendung von Co-Processes
#
# Background Filetransfer mit interaktivem ftp
#
ftp -n |&
print -p open mzsun
print -p user loginname passwort
print -p binary
print -p cd /etc
print -p get hosts
print -p quit
```

Kommunikation mit interaktivem Editor *ed*

In diesem Beispiel erfolgt die Process Kommunikation zwischen dem Shell-Script und dem Editor *ed*. Als Beispiel wird *ed* dazu veranlasst aus */etc/passwd* die Zeile mit dem eigenen Loginnamen zu lesen und diesen dann auf einen neuen Namen zu ändern. Wird der Loginname nicht gefunden so schreibt *ed* das Zeichen «?» an das Shellscript, welches dort ausgewertet werden kann (if ["$line" = "?"]).

```
#!/bin/ksh
#
# Anwendung von Co-Processes
#
# Interaktion mit Editor ed aus Korn-Shell Script
#

if [ $# -lt 1 ]
then
      echo "Usage: $0 loginname"
      exit 1
fi

loginname="$1"

ed - /etc/passwd |&      # start co-process
print -p /$LOGNAME/      # search for LOGNAME in /etc/passwd
read -p line             # read stdout from ed-editor

if [ "$line" = "?" ]     # LOGNAME found?
then
      print -p e         # write to co-process
      print -p q
      print - "line with pattern \"$LOGNAME\" not found"
      exit 1
else
      print - "$line"
      print -p s/$LOGNAME/$loginname/
      print -p /$loginname/
      read -p line
      print - "$line"
      print -p w
      print -p q
fi
```

1.4.5 Shellscript Eingaben ohne [Return]-Tasten Bestätigung

Bei Benutzer-Abfragen auf Ja/Nein ist es für den Benutzer oft mühsam jedesmal «y» <RETURN» zu drücken. Einfacher wäre es, man könnte nur «y» eingeben und das Script würde sofort weiter ausgeführt ohne auf die Bestätigung der Return-Taset zu warten. Dies ist möglich indem der Terminal-Treiber aus dem Shell-Script in den «Raw-Mode» versetzt wird. Mit einem kleinen C-Programm wird genau ein Zeichen von der Standard-Eingabe gelesen.

C-Programm (input_nowait.c)

```
/*
 * Character von stdin lesen (Non-canonical Terminal I/O)
 */

main()
{
      char c;

      /* Read Character from stdin */

      read(0,&c,1);

      /* Write Character to stdout */

      write(1,&c,1);
}
```

Shell-Script das C-Programm benutzt

```
#!/bin/ksh
#
# Non-canonical Terminal I/O innerhalb Shell-Script
#
# Terminal I/O auf Raw-Input stellen

stty raw

/usr/5bin/echo "Geben [y/n]: \c"

# Character lesen (input_nowait.c)

answer=`input_nowait`

# Terminal wieder auf canonical Input-Processing stellen

stty -raw

echo ' '
if [ "$answer" = y -o "$answer" = Y ]
then
      echo " "
      echo $answer
      echo " "
else
      echo " "
      echo "Input muss [y/n] sein"
      echo " "
fi
```

1.5 Tips aus der Praxis

Komprimiertes tar-File eines Directories erstellen

Directory komprimieren:

```
$ tar cvf - directory | compress > directory.tar.Z
```

Komprimiertes Directory «auspacken»

```
$ uncompress < directory.tar.Z | tar xvf -
```

Directories, Filesysteme über Partitions hinweg kopieren

Mit tar:

```
$ cd /directory
$ tar cf - . | (cd /newdirectory; tar xvlpf -)
```

Mit find/cpio:

```
$ cd /directory
$ find . -print | cpio -pdvmu /newdirectory
```

Directories, Filesysteme netzwerkweit kopieren

Local --> Remote

```
$ cd <fromdir>; tar cf - <files> | rsh <machine> '(cd <todir>; tar xBfp -)'
```

Local <-- Remote

```
$ rsh <machine> '(cd <todir>; tar cf -)' | tar xBfp -
```

Datensicherung mit «compress»

Daten sichern:

```
$ cd /directory
$ find . -print | cpio -ocB | compress > /dev/rst0
$ tar cvfb - 20 . | compress | dd of=/dev/rst0 obs=20b
```

Daten zurücklesen:

```
$ cd /newdirectory
$ uncompress < /dev/rst0 | cpio -idcvmB 'pattern'
$ dd if=/dev/rst0 ibs=20b | uncompress | tar xvfb - 20 'pattern'
```

Remote Datensicherung

Daten sichern:

```
$ cd /directory
$ tar cvfb - 20 directory | rsh host dd of=/dev/rst0 obs=20b
```

Daten zurücklesen:

```
$ cd /newdirectory
$ rsh -n host dd if=/dev/rst0 ibs=20b | tar xvBfb - 20 'pattern'
$
```

Stdout, stderr in Shell-Script permanent in Logfile umlenken

Oft möchte man, dass ein Shellscript, das im Hintergrund abläuft, mögliche Fehlermeldungen in ein Logfile protokolliert. Am einfachsten wird dazu der Standard-Output und Standard-Error permanent in das Logfile umgelenkt. Dazu verwendet man die folgende Zeile am Script-Anfang:

```
exec 1>$HOME/log 2>$HOME/log
```