Study-Guide: GNU und Unix Kommandos

Dieses Thema enthält acht definierte Ausbildungsziele, die sich alle um die grundlegenden Unix/ Linux Kommandos drehen. Wichtige Elemente sind hier also sowohl die BASH, als auch die Kommandos zum Kopieren, Verschieben, Löschen von Dateien und Verzeichnissen. Dazu kommen die grundlegenden Unix-Texttools. Neu hinzugekommen ist das Thema vi, das früher im zweiten Teil behandelt wurde. Arbeiten auf der Kommandozeile Texte mittels Filterprogrammen bearbeiten Durchführung eines allgemeinen Datei-Managements Benutzen von Unix Streams, Pipes und Umleitungen Erzeugung, Überwachung und Terminierung von Prozessen Modifizeren von Prozeßprioritäten Durchsuchen von Textdateien mittels regulärer Ausdrücke Allgemeine Dateibearbeitung mit vi

Seite:	-= LinuxLernSystem =- (http://www.lpi-test.de)
Kurs:	LPIC-1 [101]
Buch:	Study-Guide: GNU und Unix Kommandos
Gedruckt von:	André Scholz
Datum:	Dienstag, 1 November 2005, 10:21 Uhr

Inhaltsverzeichnis

- 1.103 - GNU und Unix Kommandos
  - 1.103.1 - Arbeiten auf der Kommandozeile
  - 1.103.2 - Texte mittels Filterprogrammen bearbeiten
  - 1.103.3 - Durchführung eines allgemeinen Datei-Managements
  - 1.103.4 - Benutzen von Unix Streams, Pipes und Umleitungen
  - 1.103.5 - Erzeugung, Überwachung und Terminierung von Prozessen
  - 1.103.6 - Modifizeren von Prozeßprioritäten
  - 1.103.7 - Durchsuchen von Textdateien mittels regulärer Ausdrücke
  - 1.103.8 - Allgemeine Dateibearbeitung mit vi

1.103 - GNU und Unix Kommandos

Dieses Thema enthält acht definierte Ausbildungsziele, die sich alle um die grundlegenden Unix/ Linux Kommandos drehen. Wichtige Elemente sind hier also sowohl die BASH, als auch die Kommandos zum Kopieren, Verschieben, Löschen von Dateien und Verzeichnissen. Dazu kommen die grundlegenden Unix-Texttools. Neu hinzugekommen ist das Thema vi, das früher im zweiten Teil behandelt wurde.

- Arbeiten auf der Kommandozeile
- Texte mittels Filterprogrammen bearbeiten
- Durchführung eines allgemeinen Datei-Managements
- Benutzen von Unix Streams, Pipes und Umleitungen
- Erzeugung, Überwachung und Terminierung von Prozessen
- Modifizeren von Prozeßprioritäten
- Durchsuchen von Textdateien mittels regulärer Ausdrücke
- Allgemeine Dateibearbeitung mit vi

1.103.1 - Arbeiten auf der Kommandozeile

Beschreibung: Prüfungskandidaten sollten in der Lage sein, mit Shell und Kommandos auf der Kommandozeile umzugehen. Das beinhaltet das Schreiben gültiger Kommandos und Kommandoabfolgen, das Definieren, Referenzieren und Exportieren von Umgebungsvariablen, die Benutzung der Kommando-History und Eingabemöglichkeiten, das Aufrufen von Kommandos im und außerhalb des Suchpfades, das Benutzen von Kommandosubstitution, das rekursive Anwenden von Kommandos über einen Verzeichnisbaum und das Verwenden von man, um Informationen über Kommandos zu erhalten.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **.**
- **bash**
- **echo**
- **env**
- **exec**
- **export**
- **man**
- **pwd**
- **set**
- **unset**
- `~/.bash_history`
- ~/.profile

Kommandos

An dieser Stelle werden noch nicht viele Kommandos vorausgesetzt. Trotzdem sollten grundlegende Befehle wie `less, more, cat, echo, ...` klar sein und problemlos verwendet werden können.

Grundsätzlich ist ein Unix-Kommando immer gleich aufgebaut. Zuerst wird das Programm angegeben, dann optionale Parameter und dann die Dateien, auf die das Programm angewandt werden soll.

Parameter liegen meist in zwei verschiedenen Formen vor, einmal als kurze Form, eingeleitet durch einen Bindestrich gefolgt von einem Buchstaben (wie etwa `-r`) und andererseits als lange Form, eingeleitet durch zwei Bindestriche gefolgt von einem Wort (`--recursive`). Kurze Parameter können dabei meist zusammengefasst werden, so daß statt `-z -v -f` einfach `-zvf` geschrieben werden kann. Das erklärt auch, warum lange Parameter mit zwei Bindestrichen beginnen. Sonst könnte ja ein `--recursive` als `-r -e -c -u -r -s -i -v -e` missinterpretiert werden.

Kommandoeingabe

Grundsätzlich ist die Eingabe von Kommandos eine Eingabe an die Shell (bash). Die Shell

interpretiert das eingegebene Kommando, ersetzt bestimmte Muster durch Dateinamen oder Variableninhalte und führt erst dann das Kommando aus.

Linux-Kommandos werden entweder einfach eingegeben, oder zusammen mit dem Pfad, der die genaue Position des aufzurufenden Programmes im Dateisystem festlegt. Normalerweise reicht der Aufruf eines Kommandos ohne Pfad, wenn das Kommando im Suchpfad liegt.

Der Suchpfad wird in einer Shellvariable PATH definiert, die in der Regel in den Konfigurationsdateien der Shell mit vernünftigen Werten belegt wird. Diese Variable enthält eine durch Doppelpunkte getrennte Liste von Verzeichnissen, die von der Shell nach einem Befehl durchsucht werden sollen. Das aktuelle Verzeichnis ist nicht automatisch Teil des Suchpfades (wie bei DOS/Windows) sondern wird nur durchsucht, wenn es explizit durch die Angabe eines einzelnen Punktes im Suchpfad angegeben wurde.

Welches Verzeichnis gerade das aktuelle Arbeitsverzeichnis ist, kann mit dem Befehl **pwd** herausgefunden werden. Dieser Befehl gibt den kompletten absoluten Pfad des aktuellen Arbeitsverzeichnisses auf der Standard-Ausgabe aus.

Die Shell durchsucht den Suchpfad in der angegebenen Reihenfolge. Existieren auf einem System jetzt zwei Programme gleichen Namens, so wird das Programm ausgeführt, das sich in dem zuerst im Suchpfad genannten Verzeichnis befindet. Dieser Mechanismus kann umgangen werden, wenn nicht nur der Programmname, sondern der komplette Pfad als Befehl eingegeben wird. Auch Programme, die sich in Verzeichnissen befinden, die nicht im Suchpfad sind, können auf diese Weise aufgerufen werden.

Die Eingabe gültiger Kommandos und Kommandogruppen beinhaltet natürlich auch das Wissen um die verschiedenen Möglichkeiten, mehrere Befehle in eine Befehlszeile zu schreiben. Zur Erinnerung sind hier nochmal die verschiedenen Möglichkeiten der bash aufgelistet:

`Kommando1 ; Kommando2`
: Kommando2 wird nach Kommando1 ausgeführt.

`Kommando1 & Kommando2`
: Kommando1 und Kommando2 werden gleichzeitig ausgeführt (K1 im Hintergrund, K2 im Vordergrund)

`Kommando1 && Kommando2`
: Kommando2 wird nur dann ausgeführt, wenn Kommando1 fehlerfrei abgeschlossen wurde (Rückgabewert 0)

`Kommando1 || Kommando2`
: Kommando2 wird nur dann ausgeführt, wenn Kommando1 nicht fehlerfrei abgeschlossen wurde (Rückgabewert ungleich 0)

Wenn ein Kommando durch die Enter-Taste abgeschlossen wurde, dann wird es ausgeführt, sofern es syntaktisch fertig war. Fehlt aber noch etwas, für den syntaktischen Abschluß, war also etwa ein Anführungszeichen, das noch nicht durch ein zweites abgeschlossen wurde, so erscheint ein Folgeprompt, der eine weitere Eingabe ermöglicht. Das wird solange fortgesetzt, bis die Eingabe syntaktisch vollständig ist.

Vor der eigentlichen Ausführung des Kommandos wird der gesamte eingegebene Befehl in einen Kommandospeicher abgespeichert, aus dem er jederzeit (durch Navigieren mit den Pfeiltasten nach oben und unten) wiedergeholt und erneut ausgeführt werden kann. Wird die

Shell beendet (etwa durch das Ausloggen), so werden alle im Kommandospeicher gespeicherten Befehle in die Datei `~/.bash_history` im Heimatverzeichnis des jeweiligen Users abgelegt, so daß die Befehle auch beim erneuten Einloggen wieder zur Verfügung stehen.

Ein eingegebenes Kommando erzeugt in der Regel einen neuen Prozeß. Wenn der Prozeß abgeschlossen wurde kehrt die Shell wieder zur Eingabeaufforderung zurück. Ein spezieller Befehl erlaubt es aber, einen Befehl einzugeben, der die Shell ersetzt. Der Befehl **exec *Kommando*** ersetzt den Shellprozeß durch den Prozeß des eingegebenen Kommandos. Das macht Sinn, wenn eine andere Shell aufgerufen werden soll oder der Befehl innerhalb eines Scripts ausgeführt wird. Wird ein durch **exec** ausgeführter Befehl beendet, ist aber logischerweise die Shell nicht mehr da, der Shellprozeß wurde ja durch das Kommando ersetzt.

Umgebungsvariablen

Variablen sind kleine Stücke des Arbeitsspeichers - genauer gesagt des Umgebungsspeichers der jeweiligen Shell - die einen Namen haben und einen Wert speichern können. Jede Instanz einer Shell hat einen eigenen Umgebungsspeicher, so daß verschiedene Shell-Instanzen gleichnamige Variablen mit unterschiedlichen Werten besitzen können.

Umgebungsvariablen werden in einer Shell definiert, indem sie einfach mit der Anweisung

Variablenname=Wert

angelegt werden. Existierte die Variable mit dem entsprechenden Namen noch nicht, so wird sie neu angelegt, gab es sie schon, so wird ihr Wert verändert.

Um auf den Inhalt (Wert) einer Variable zuzugreifen (refernzieren der Variable), wird dem Variablennamen ein Dollarzeichen ($) vorangestellt. Wenn die Shell auf ein Dollarzeichen stößt, versucht sie, den darauf folgenden Begriff als Variablennamen zu interpretieren und das ganze Konstrukt mit dem Wert der entsprechenden Variable zu ersetzen.

```
NAME=Huber
VORNAME=Hans
echo Hallo $VORNAME $NAME

  Hallo Hans Huber
```

Alle definierten Variablen sind mit dem Befehl **set** anzeigbar, mit **unset** können Variablen aus dem Umgebungsspeicher entfernt werden:

```
unset NAME
unset VORNAME
```

Wenn aus einer Shell heraus eine neue Shell (Subshell) aufgerufen wird, dann werden nicht automatisch alle Variablen der aufrufenden Shell für die Subshell übernommen. Um eine bestimmte Variable an spätere Subshells weiterzugeben, muß diese Variable exportiert werden. Das geschieht mit der Shell-Anweisung `export`. Dabei kann entweder eine bestehende Variable exportiert werden wie mit

`export` *`Variablenname`*

oder eine Variable gleich so definiert werden, daß sie später exportiert wird mit

`export` *`Variablenname=Wert`*

Eine exportierte Variable ist eine Kopie der Orginalvariablen. Das heißt, daß die Orginalvariable der aufrufenden Shell unverändert bleibt, wenn die exportierte Variable der Subshell verändert wird!

Variablen, die immer wieder benötigt werden, können in der Datei `~/.profile` im Homeverzeichnis jedes Users definiert werden. Diese Datei wird bei jedem Start einer Login-Shell abgearbeitet. So stehen einmal dort definierte Variablen jederzeit wieder zur Verfügung.

Soll ein Kommando mit bestimmten Variablen zusammen aufgerufen werden (also in einer bestimmten Umgebung), dann kann das mit dem Befehl env erledigt werden. Dabei kann entweder das Programm in einer völlig leeren Umgebung gestartet werden, oder zusätzliche Variablen werden speziell für diesen Programmaufruf definiert.

Kommandosubstitution

Bei der Kommandosubstitution handelt es sich um ein Konstrukt, das eine Subshell öffnet, einen bestimmten Teil einer Kommandozeile in dieser Subshell ausführt und das, was diese Ausführung auf die Standard-Ausgabe schreiben würde, an die Stelle der ursprünglichen Kommandozeile einfügt, an der das Substitutionskonstrukt stand. Erst jetzt wird die gesamte Zeile ausgeführt.

Es gibt zwei Möglichkeiten, die Kommandosubstitution einzuschalten. Zum einen die Klammerung mit dem Grave-Zeichen (`` ` ``) und zum anderen das Konstrukt `$(...)`

Das klingt schlimmer als es ist, ein einfaches Beispiel soll zeigen, worum es geht:

Der Befehl `pwd` gibt an, in welchem Verzeichnis wir uns gerade befinden (Print WorkingDirectory). Wir werden ihn jetzt in einer Kommandosubstitution benutzen. Das Kommando

```
echo Wir befinden uns im Verzeichnis $(pwd)
```

wird eingegeben. Statt dessen wäre auch

```
echo Wir befinden uns im Verzeichnis `pwd`
```

gegangen. Was passiert nun mit dieser Befehlszeile? Die Shell ließt diese Zeile und stößt auf das Konstrukt `$(pwd)`. Sie erkennt dieses Konstrukt als Aufruf einer Kommandosubstitution und startet jetzt eine Subshell. Diese Subshell führt den Befehl `pwd` aus. Dieser Befehl schreibt das aktuelle Arbeitsverzeichnis auf die Standard-Ausgabe. Nehmen wir mal an, wir befänden uns im Verzeichnis `/home/hans`. Die Ausgabe des Befehls wäre also `/home/hans`. Jetzt ersetzt die ursprüngliche Shell das Konstrukt `$(pwd)` mit der Ausgabe der Subshell, also mit `/home/hans`. Die dadurch entstehende Kommandozeile lautet jetzt also

```
echo Wir befinden uns im Verzeichnis /home/hans
```

Und erst jetzt führt die Shell dieses Kommando aus.

Rekursives Anwenden von Kommandos

Viele Unix Kommandos erlauben es, daß sie nicht nur auf eine Datei oder ein Verzeichnis angewandt werden können, sondern auf ganze Verzeichnis-Hierarchien. Dazu steht meist ein Kommandozeilen-Parameter wie -r, -R oder --recursive zur Verfügung. Unglücklicherweise unterscheiden sich die Kommandos dabei ob -r oder -R verstanden wird. Der lange Parameter --recursive funktioniert aber überall, wo Rekursion überhaupt zur Verfügung steht. Ein paar Beispiele:

Befehl	Kurze Form	Lange Form
`ls`	`-R`	`--recursive`
`chown`	`-R`	`--recursive`
`chmod`	`-R`	`--recursive`
`chgrp`	`-R`	`--recursive`
`grep`	`-r`	`--recursive`
`cp`	`-r` und `-R`	`--recursive`
`rm`	`-r` und `-R`	`--recursive`

Informationen über Befehle mit man

Bei der Vielzahl von Linux-Befehlen, kann es sehr schnell passieren, daß man vergisst, welche Parameter von welchem Programm verstanden werden. Linux bietet eine Online-Hilfe an, die diese Informationen bereitstellt. Dabei handelt es sich um ein komplettes Handbuchsystem, das sowohl ausgedruckt, als auch auf dem Bildschirm dargestellt werden kann.

In der Vorbereitung auf die LPI102 Prüfung wird dieses Handbuchsystem noch genauer erläutert. Für den einfachen Umgang damit reichen die folgenden Informationen:

Das Programm man bietet Zugriff auf die Handbuchseiten (manual-pages). Es erwartet mindestens den Namen des Programmes als Parameter, über das Informationen dargestellt werden sollen. Um also Informationen über den Befehl **cp** (copy) zu erhalten, genügt der Befehl

```
man cp
```

Das Programm **man** stellt jetzt eine Handbuchseite zum Thema cp zusammen und gibt sie über den systemeigenen Pager (meist das Programm **less** auf dem Bildschirm aus.

1.103.2 - Texte mittels Filterprogrammen bearbeiten

Beschreibung: Prüfungskandidaten sollten in der Lage sein, Filter auf Textströme anzuwenden. Dieses Lernziel beinhaltet das Senden von Textdateien und -ausgaben durch Textfilterprogramme, um die Ausgabe zu modifizieren, und die Verwendung von Standard-Unix-Kommandos, die im GNU textutils Paket enthalten sind.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **cat**
- **cut**
- **expand**
- **fmt**
- **head**
- **join**
- **nl**
- **od**
- **paste**
- **pr**
- **sed**
- **sort**
- **split**
- **tac**
- **tail**
- **tr**
- **unexpand**
- **uniq**
- wc

Textfilter

Unter Unix werden sehr viele kleine Programme angeboten, die als Filter für Textdateien zur Anwendung kommen und die in der Regel nur eine kleine Aufgabe beherrschen. Durch die Kombination mehrerer solcher Programme mit Pipes und Umleitungen (siehe Abschnitt 1.103.4 - Benutzen von Unix Streams, Pipes und Umleitungen) können dadurch vielfältige Aufgaben gelöst werden. Die in dieser Zielsetzung genannten Textfilter sollten Sie beherrschen. Hier nochmal zu diesen Filtern ein paar Kurzbeschreibungen. Grundsätzlich ist das Studium der jeweiligen Handbuchseiten hier sehr empfehlenswert...

sed

sed ist ein Stream-Editor, ein Editor, der Datenströme oder Dateien nach bestimmten Regeln bearbeitet. sed ist ein sehr komplexes Programm, von dem in der LPI-101-Prüfung nur sehr geringe Kenntnisse verlangt werden. Wichtig ist zu wissen, daß dieses Werkzeug in der Lage ist, immer wiederkehrende Aufgaben mit Hilfe einer Scriptdatei zu lösen.

Ein Beispiel mag das verdeutlichen:

Sie haben 500 Textdateien, in denen immer wieder der Begriff "Synopsis" vorkommt. Sie wollen diesen Begriff nach "Syntax" ändern. Außerdem soll jeweils die 3. Zeile jeder dieser Dateien gelöscht werden. Das wäre unglaublich viel Arbeit, hätten wir nicht das Programm sed. Der Streameditor erlaubt uns verschiedene Aktionen auf einen Datenstrom auszuführen. Dazu erstellen wir eine Datei, die die notwendigen Befehle enthält. Nennen wir sie "befehle":

```
1,$s/Synopsis/Syntax/g
3d
```

Jede Zeile enthält einen Befehl. Jeder Befehl besteht aus einer Adress-Angabe und dem Befehl selbst. Die erste Zeile hat die Adressangabe `1,$`, was soviel heißt wie *von der ersten Zeile bis zum Dateiende*. Der Befehl ist `s` - das steht für *substitute* - also ersetzen. Der Befehl s hat die Form:

s/*Suchbegriff*/*Ersetzungsbegriff*/*Optionen*

Als Option haben wir g benutzt, was heißt, daß eine Zeile global durchsucht werden soll, nicht nur das erste Auftreten, sondern jedes soll ersetzt werden.

Die zweite Zeile unserer Datei ist noch einfacher, ihr Adress-Teil ist einfach die 3 und meint also Zeile 3. Der Befehl ist d und steht für *delete* also löschen. Es ist also die Anweisung, die Zeile 3 zu löschen.

Um diese Befehlsdatei jetzt anzuwenden, schreiben wir nur noch:

```
sed -f befehle Datei > Datei_neu
```

und schon wurde die Datei `Datei` bearbeitet. Das Ergebnis steht jetzt in `Datei_neu`. Eine kleine Schleife konstruiert, und schon könnten wir alle 500 Dateien mit einem Aufwasch bearbeiten...

Wie gesagt, die LPI 101-Prüfung besteht nicht darauf, daß Sie den sed vollständig beherrschen, aber ein paar Übungen dazu sind nicht verkehrt. Genauere Beschreibungen, was dieses Programm für Befehle versteht finden Sie auf der sed-Handbuchseite.

sort

Dieses kleine Programm sortiert einen Eingabedatenstrom oder eine Datei zeilenweise und gibt das Ergebnis wieder auf die Standard-Ausgabe aus.

Normalerweise werden die ganzen Zeilen verglichen, um sie zu sortieren. Es ist aber auch möglich, Zeilen nach einem bestimmten Positionsfeld zu sortieren. Ein Beispiel:

Der Befehl `ls -l` gibt uns eine Liste aller Dateien im aktuellen Verzeichnis aus. Ab Spalte 31 steht die Dateigröße. Würden wir den Befehl

```
ls -l | sort
```

eingeben, dann würden die gesamten Zeilen sortiert. Das macht wenig Sinn, denn die Zeilen beginnen ja immer mit der Angabe der Zugriffsrechte und die zu sortieren ist sicher nicht das, was wir wollen. Um die Ausgabe nach Dateigröße zu sortieren können wir eingeben:

```
ls -l | sort +0.32
```

Dabei bedeutet `0.32` das erste Feld (0) und davon alles, ab dem 31. Buchstaben. (Die Angabe von Feldern bezieht sich auf Dateien, die tatsächlich feldorientiert aufgebaut sind, wie etwa `/etc/passwd`)

Alle Optionen und Möglichkeiten entnehmen Sie wieder der sort-Handbuchseite.

uniq

uniq ist ein Programm, das aus einer Datei (oder einem Eingabedatenstrom) hintereinanderliegende gleiche Zeilen löscht, so daß nur noch eine Zeile übrig bleibt. Die Betonung liegt auf hintereinanderliegend. Am häufigsten wird uniq in Kombination mit **sort** eingesetzt, weil gerade beim sortieren häufig vorhandene Zeilen hintereinander geschrieben werden. Hat eine Datei (oder ein Datenstrom) z.B. viele Leerzeilen, so werden all diese Leerzeilen hintereinander in die sortierte Ausgabe geschrieben. Mit **uniq** können diese Leerzeilen auf eine Zeile reduziert werden.

Über Kommandozeilenparameter kann zudem eingestellt werden, daß nur bestimmte Teile einer Zeile verglichen werden, um die Gleichheit festzustellen. Die genauen Parameter sind der uniq-Handbuchseite zu entnehmen.

cut

Mit cut werden bestimmte Spalten einer Datei ausgeschnitten und auf die Standard-Ausgabe geschrieben. Dabei können Spalten entweder durch absolute Positionen angegeben werden, oder als Felder, die durch bestimmte Trennzeichen abgegrenzt sind.

Um z.B. alle Usernamen aus der Datei /etc/passwd zu schneiden benutzen wir die Feldbegrenzer `:` und wollen nur das erste Feld sehen:

```
cut -d: -f1 /etc/passwd
```

Wenn wir aus der Liste aller laufender Prozesse die PIDs ausschneiden wollen, so müssen wir das mit absoluten Positionsangaben machen. Die Ausgabe von ps huax sieht ja etwa so aus:

```
at         172  0.0  0.4   892   448  ?  S    10:09   0:00 /usr/sbin/atd
bin        128  0.0  0.3   816   360  ?  S    10:09   0:00 /sbin/portmap
...
```

Die ProzeßIDs stehen also von Zeichen 9 bis 14. Um genau diese Spalte auszuschneiden schreiben wir also:

```
ps huax | cut -c9-14
```

So bietet uns cut also die Möglichkeit, jeden beliebigen Teil einer Zeile auszuschneiden und auf die Standard-Ausgabe zu schreiben. Das ist natürlich hervorragend geeignet, um in Pipes weiterverarbeitet zu werden. Die genaue Beschreibung entnehmen Sie wieder der cut-Handbuchseite

expand

Das Programm expand wird benutzt, um Tabulatorzeichen aus einer Textdatei oder einem Eingabedatenstrom durch eine beliebige Menge Leerzeichen im Ausgabedatenstrom zu ersetzen. Das klingt trivial, ist aber sehr häufig gebraucht, um Listen, die durch Tabs getrennt sind in ein druckbares Format zu verwandeln.

Die genaue Beschreibung ist der expand-Handbuchseite zu entnehmen.

unexpand

unexpand arbeitet genau gegenläufig zu expand. Es verwandelt eine bestimmte Menge Leerzeichen aus einer Datei (oder einem Eingabedatenstrom) in Tabulatorschritte. Dabei kann mit Parametern angegeben werden, wieviele Leerzeichen jeweils in einen Tab verwandelt werden sollen.

Die genaue Beschreibung ist der unexpand-Handbuchseite zu entnehmen.

fmt

fmt (steht für format) ist ein Programm, das Textdateien oder Eingabedatenströme absatzweise formatiert. Dabei sind Einstellungen möglich, wie etwa die Breite jedes Absatzes, die Anfangseinrückung oder das Zusammenziehen von mehreren Leerzeichen. So können etwa Textdateien mit Zeilenlängen über 80 für eine Druckausgabe auf eine Zeilenbreite von 75 umgewandelt werden.

Die genaue Beschreibung ist der fmt-Handbuchseite zu entnehmen.

head

Head schreibt die ersten Zeilen einer Datei oder eines Eingabedatenstroms auf die Standard-Ausgabe. Ohne Parameter werden die ersten 10 Zeilen ausgegeben.

Durch Kommandozeilenparameter kann bestimmt werden, wieviele Zeilen oder Bytes vom Anfang der Datei ausgegeben werden sollen. Dazu wird die gewünschte Anzahl Zeilen einfach mit einem Bindestrich versehen an den Befehl angehängt.

Die genaue Beschreibung ist der head-Handbuchseite zu entnehmen.

tail

Tail arbeitet wie head, nur daß nicht die ersten, sondern die letzten Zeilen einer Datei oder eines Eingabedatenstroms ausgegeben werden. Auch hier kann die gewünschte Anzahl der zu zeigenden Zeilen angegeben werden.

In Kombination mit head können damit beliebige Teile einer Datei ausgeschnitten werden. Wollen wir z.B. die Zeilen 7-12 der Datei `versuch.txt`, so können wir zuerst mit head die ersten 12 Zeilen der Datei auschneiden und uns dann daraus die letzten 6 Zeilen mit tail entnehmen. Elegant wird das in einer Pipe:

```
cat versuch.txt | head -12l | tail -6l
```

tail hat noch den speziellen Parameter -f, der es anweist, nicht nach der Ausgabe der letzten 10 Zeilen abzubrechen, sondern zu warten, ob die Datei noch weiter wächst und die dann angehängten Zeilen ebenfalls auszugeben. Wollen wir so etwa die Meldungen des Syslog-Daemon ständig überwachen, so schreiben wir:

```
tail -f /var/log/messages
```

Solange wir tail nicht mit Strg-C abbrechen, wird es nun die Datei /var/log/messages dauernd überwachen und jede neue Zeile auf dem Bildschirm anzeigen. Wir können also der Datei "beim Wachsen zusehen".

Die genaue Beschreibung ist der tail-Handbuchseite zu entnehmen.

join

join verknüpft zwei (alphabetisch) sortierte Dateien, indem je zwei Zeilen mit identischen Schlüsselfeldern zu einer Ausgabezeile verbunden werden.

Die Schlüsselfelder sind - wenn nicht anders angegeben - durch Leerzeichen voneinander getrennt. Führende Leerzeichen werden ignoriert. Wenn nicht anders angegeben, ist das erste Feld einer jeden Zeile Schlüsselfeld. Die Ausgabefelder sind ebenfalls durch Leerzeichen voneinander getrennt. Die Ausgabe besteht aus dem Schlüsselfeld, gefolgt von den übrigen Feldern der Datei1 und schließlich aller Felder der passenden Zeilen von Datei2 ohne das Schlüsselfeld.

Damit können verschiedene Dateien nach inhaltlichen Kriterien zusammengefügt werden. Nehmen wir ein Beispiel:

Die Datei `Adressen.txt` beinhaltet Zeilen wie die folgenden:

```
...
Huber Peter Schillerstr. 23 54321 Musterhausen
Maier Hans Goethestr. 3 12345 Musterstadt
Schmidt Stefan Kleistweg 34 55555 Musterhofen
...
```

Die Datei `Jobs.txt` enthält jetzt etwa

```
Huber Geschäftsführer
Kohler Fensterputzer
Schmidt Aufzugführer
```

Wenn wir jetzt beide Dateien mit dem Befehl

```
join Adressen.txt Jobs.txt
```

miteinander verknüpfen, so werden nur die Zeilen miteinander verbunden, die gemeinsame Schlüsselfelder haben. Voreingestellt ist das erste Feld, also der Name. Das Ergebnis wäre:

```
Huber Peter Schillerstr. 23 54321 Musterhausen Geschäftsführer
Schmidt Stefan Kleistweg 34 55555 Musterhofen Aufzugführer
```

Die genaue Beschreibung ist der join-Handbuchseite zu entnehmen.

nl

nl gibt die Zeilen einer oder mehrerer Dateien (oder der Standardeingabe) mit Zeilennummern auf die Standardausgabe. Es können dabei die Zeilen einer (logischen) Seite in einen Kopf, einen Körper und einen Fuß unterteilt werden, die jeweils einzeln und in unterschiedlichen Stilen nummeriert werden. Jeder Teil kann auch leer sein. Wenn vor dem ersten Kopfteil bereits Zeilen vorhanden sind, werden diese Zeilen wie ein Seitenkörper nummeriert.

Die Nummerierung beginnt auf jeder Seite neu. Mehrere Dateien werden als ein einziges Dokument betrachtet, und die Zeilennummer nicht zurückgesetzt.

Der Kopfteil wird durch eine Zeile eingeleitet, die nur die Zeichenkette \:\:\: enthält. Der Körper wird entsprechend durch \:\: und der Fuß durch \: eingeleitet. In der Ausgabe werden diese Zeilen als Leerzeilen ausgegeben.

Die genaue Beschreibung ist der nl-Handbuchseite zu entnehmen.

od

od (Octal Dump) gibt Dateien oder einen Eingabedatenstrom als dezimalen, oktalen oder hexadezimalen Dump aus. So können Binärdateien in eine lesbare Form gebracht werden. Jede Ausgabezeile enthält neben der eigentlichen Bytedarstellung am Zeilenanfang noch ein siebenstelliges Adressfeld.

Die genaue Beschreibung ist der od-Handbuchseite zu entnehmen.

paste

paste verknüpft zwei oder mehrere Dateien zeilenweise, so daß die ersten Zeilen der Dateien zur ersten Zeile der Ausgabedatei werden.

Damit können also einzelne Dateien, die Spalten enthalten, die etwa von cut erzeugt wurden, wieder zu einer Datei zusammengesetzt werden. Nehmen wir ein simples Beispiel:

Wir wollen eine Datei erstellen, in der in der ersten Spalte die UserID, in der zweiten der Username steht. Die Datei soll alle User des Systems enthalten. Wir können das nicht allein mit

cut erledigen, weil ja in /etc/passwd zuerst der Username und erst im 3. Feld die UserID steht. Zunächst erzeugen wir also zwei Dateien, die jeweils nur eine Spalte der /etc/passwd beinhalten, dann fügen wir diese beiden Dateien umgekehrt wieder zusammen:

```
cut -d: -f1 /etc/passwd > Datei1
cut -d: -f3 /etc/passwd > Datei2
paste -d: Datei2 Datei1 > Datei3
```

Im Gegensatz zu join (siehe oben) werden die Zeilen der angegebenen Dateien ohne inhaltliches Kriterium miteinander verknüpft. Die genaue Beschreibung ist der paste-Handbuchseite zu entnehmen.

pr

pr dient zur Formatierung von Textdateien für die Druckerausgabe. pr wird heute nur noch selten verwendet, es gibt einfach andere, bessere Druckerfilter. Trotzdem kann es manchmal noch gute Dienste leisten.

Immerhin bietet pr auch Features wie Spaltensatz, Seitennummerierung und Kopfzeilen, genaueres siehe bei der pr-Handbuchseite

split

Die Anwendung von split ist einfach und sie wird häufig gebraucht, um z.B. große Dateien in viele kleinere aufzuteilen, damit sie mit Disketten transportiert werden können. Die Größe der anzulegenden Zieldateien kann sowohl in Zeilen, als auch in Bytes angegeben werden.

Die kleineren Zieldateien heißen standardmäßig xaa, xab, xac, ..., wobei - falls die Buchstaben nicht ausreichen - auch mehrere Buchstaben verwendet werden (xaaaa, xaaab,...). Weil das ein wenig unhandlich ist, kann man dem split-Programm ein Präfix mitgeben, das dann das x ersetzt.

Um also z.B. eine Datei netscape.tgz mit ca 13 MByte in Dateien der Größe 1 MByte aufzuteilen, um sie auf Disketten verteilen zu können, schreiben wir:

```
split -b 1m netscape.tgz
```

Als Ergebnis bekommen wir jetzt die Dateien xaa, xab, xac, ... Das ist verwirrend - daher benutzen wir eben das Präfix:

```
split -b 1m netscape.tgz netscape_
```

Jetzt bekommen wir die Dateien netscape_aa, netscape_ab, ...

Um diese Dateien wieder zusammenzubauen benutzen wir einfach das cat-Programm. wir könnten so also schreiben:

```
cat netscape_aa netscape_ab netscape_ac netscape_ad netscape_ae > netscape.tgz
```

Das ist zugegebenerweise etwas unhandlich, aber jetzt kommt der Vorteil ins Spiel, daß die Shell alphabetisch ihre Ausgaben sortiert. Weil die aa, ab, ac ... Aufteilung streng alphabetisch ist, können wir auch viel einfacher schreiben:

```
cat netscape_* > netscape.tgz
```

Die genaue Beschreibung ist der split-Handbuchseite zu entnehmen.

cat

Das Programm cat arbeitet ähnlich wie das DOS-Programm TYPE. Es gibt die Dateien, die als Parameter angegeben wurden auf der Standard-Ausgabe aus. Falls keine Parameter angegeben wurden, so nimmt cat die Standard-Eingabe als Datenstrom an, der auf der Standard-Ausgabe ausgegeben werden soll.

Das klingt auf den ersten Blick trivial, wird aber sehr häufig gebraucht, um Dateien in eine Pipe oder auf ein Gerät zu schicken:

```
cat datei.txt > /dev/tty10
```

Schickt die Datei datei.txt statt auf die Standard-Ausgabe in die Gerätedatei /dev/tty10 und damit auf das Terminal 10.

Außerdem können mit cat mehrere Dateien zu einer zusammengefügt werden, indem die Ausgabe wieder in eine Datei umgeleitet wird:

```
cat teil1.txt teil2.txt teil3.txt > gesamt.txt
```

Mit Optionsschaltern kann cat auch dazu gebracht werden, mehrere aufeinanderfolgende Leerzeilen zu einer Leerzeile zu reduzieren oder Zeilen in der Ausgabe zu nummerieren.

Die genaue Beschreibung ist der cat-Handbuchseite zu entnehmen.

tac

Das Programm tac (cat umgekehrt geschrieben) gibt Dateien in umgekehrter Reihenfolge aus, also die letzte Zeile zuerst, dann die vorletzte usw. Dabei müssen es nicht zwangsläufig Zeilen sein, tac kann auch andere Zeichen als den Zeilentrenner als Trennzeichen benutzen.

Die genaue Beschreibung ist der tac-Handbuchseite zu entnehmen.

tr

tr (translate) löscht oder ändert einzelne Zeichen eines Eingabedatenstroms und schreibt das Ergebnis wieder auf die Standard-Ausgabe.

Die Angabe der zu übersetzenden Zeichen erfolgt dabei in sogenannten Mengen. Eine Menge ist einfach eine Zeichenkette aus Buchstaben. Um z.B. in einer Datei alle vorkommenden a,b und x in A,B und X zu verwandeln, benutzen wir die Mengen "abx" und "ABX".

tr ist ein reiner Filter, d.h., das Programm kann ausschließlich in Pipes verwendet werden. Wenn wir also die Datei `Test.txt` übersetzen wollen, so müssen wir schreiben:

```
cat Test.txt | tr abx ABX > Ergebnis
```

Zugegebenermaßen ein unsinniges Beispiel. tr erlaubt es aber auch, Steuerzeichen und andere undruckbare Zeichen in den Mengen zu verwenden, indem ihr Oktalwert angegeben wird. Dazu muß dem Wert ein Backslash vorangestellt werden. Hier sind dann aber auch Anführungszeichen nötig, sonst würde die Shell uns den Backslash weginterpretieren.

Außerdem bietet tr mehrere bereits vordefinierte Mengen, wie etwa alle Großbuchstaben und alle Kleinbuchstaben. So können wir eine Datei z.B. in Großbuchstaben verwandeln indem wir schreiben:

```
cat Test.txt | tr [:lower:] [:upper:] > Ergebnis
```

Die genaue Beschreibung ist der tr-Handbuchseite zu entnehmen.

wc

Dieses Programm (word count) zählt Zeichen, Wörter und Zeilen einer Datei oder eines Eingabedatenstroms. Es ist in der LPI 101 Prüfung relativ häufig gefragt, aber auch wirklich sehr einfach anzuwenden.

Ohne Parameter aufgerufen gibt wc alle drei Angaben aus, in der Reihenfolge Zeilen, Wörter, Zeichen. wc kennt aber auch Parameter, die es veranlassen nur die Zeilen, Wörter oder Zeichen zu zählen. Es gibt diese Parameter jeweils in einer langen und kurzen Form, aber auch jeweils leicht zu merken und zu verstehen:

- `-l` oder `--lines` zählt nur die Zeilen
- `-w` oder `--words` zählt nur die Wörter
- `-c` oder `--chars` zählt nur die Zeichen

Die genaue Beschreibung ist der wc-Handbuchseite zu entnehmen.

1.103.3 - Durchführung eines allgemeinen Datei-Managements

Beschreibung: Prüfungskandidaten sollten in der Lage sein, die grundlegenden Unix Kommandos zum Kopieren und Verschieben von Dateien und Verzeichnissen zu benutzen. Dieses Lernziel beinhaltet das Durchführen fortgeschrittener Dateiverwaltungs-Operationen wie dem rekursiven Kopieren mehrerer Dateien, das rekursive Löschen von Verzeichnissen und das Verschieben von Dateien, deren Namen einem bestimmten Muster entsprechen. Dies beinhaltet das Benutzen einfacher und komplexerer Wildcard-Muster für das Bestimmen von Dateien sowie die Verwendung von **find**, um Dateien aufgrund von Typ, Größe oder Zeitstempel ausfindig zu machen und Operationen an ihnen durchzuführen.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **cp**
- **find**
- **mkdir**
- **mv**
- **ls**
- **rm**
- **rmdir**
- **touch**
- *File-Globbing* (Suchmuster mit Wildcards)

Kommandos zum Umgang mit Dateien

Grundsätzlich stehen uns unter Linux die gleichen Programme zur Verfügung, wie unter jedem anderen textorientierten System. Wir können Dateien kopieren, verschieben (umbenennen) und löschen. Die jeweils notwendigen Kommandos sind cp (copy), mv (move) und rm (remove).

Im Gegensatz zu MSDOS Befehlen haben diese Linux/Unix-Befehle aber eine Eigenschaft, die nennenswert ist; sie können beliebig viele Dateien bearbeiten. Das heißt, es ist möglich mehrere Dateinamen anzugeben, die kopiert, verschoben oder gelöscht werden sollen.

Das bringt zumindestens beim kopieren und verschieben ein Problem mit sich, das so unter DOS nicht entsteht. DOS-Befehle zum kopieren und verschieben erlauben es, das Ziel der Operation wegzulassen. Fehlt das Ziel, so wird das aktuelle Verzeichnis als Ziel angenommen. Das ist bei Unix Befehlen **nicht** möglich, und zwar eben genau weil diese Befehle mehrere Dateien bearbeiten können. Dürften wir das Ziel weglassen, so wäre der Befehl, sofern ihm mehrere Dateien mitgegeben wurden, nicht in der Lage zu ermitteln, ob der letzte Parameter eine zu kopierende Datei sein soll oder das Zielverzeichnis... Trotzdem ist das leicht zu verschmerzen, heißt das aktuelle Verzeichnis doch einfach **.** Es reicht also, als letzten Parameter einen Punkt anzugeben, das ist nicht viel Mehraufwand.

Alle drei Befehle sind in der Lage, auch Verzeichnisse zu bearbeiten, cp und rm jedoch nur,

wenn ihnen das mittels Kommandozeilenparameter mitgegeben wurde. Näheres dazu siehe unten unter dem Punkt "Rekursive Bearbeitung".

Manchmal ist es einfach nur nötig, eine leere Datei neu anzulegen. Diese Aufgabe kann auf unterschiedliche Weise gelöst werden, hier drei Beispiele:

- **Kopieren der Gerätedatei /dev/null in eine Datei**
  Die Gerätedatei /dev/null ist eigentlich dazu gedacht, unerwünschte Ausgaben ins Nirwana zu schicken, indem sie dorthin umgeleitet werden. Man kann diese Datei aber dazu nutzen, eine neue, leere Datei anzulegen, indem man schreibt:

  ```
  cp /dev/null Neudatei
  ```

  Das Ergebnis ist eine leere Datei mit Namen Neudatei.

- **Einfache Umleitung**
  Mit einem Umleitungssymbol (>) kann auch eine leere Datei angelegt werden. Das ist der einzige Fall, in dem in einer Unix-Kommandozeile kein Befehl am Anfang steht. Das Kommando

  ```
  >Neudatei
  ```

  legt eine leere neue Datei mit Namen Neudatei an.

- **Das Programm touch**
  Das Programm touch ist eigentlich dazu gedacht, die Zeitmarkierung einer Datei auf das aktuelle Datum und die aktuelle Uhrzeit zu ändern. Die Datei wird sozusagen "berührt" und damit werden die Zeitmarken verändert. Wenn die angegebene Datei jedoch noch nicht existiert, so wird sie angelegt. Der Befehl

  ```
  touch Neudatei
  ```

  legt also auch eine leere neue Datei mit Namen Neudatei an.

Alle drei Methoden sind im Ergebnis gleich. In jedem Fall haben wir eine leere, neue Datei der Größe 0 Byte.

Kommandos zum Umgang mit Verzeichnissen

Neue Verzeichnisse werden unter Linux mit dem Befehl mkdir (make directory) erstellt. Mit dem Parameter `-p` oder `--parents` kann dieser Befehl sogar einen ganzen Verzeichnisbaum mit beliebig vielen Unterverzeichnissen anlegen.

Leere Verzeichnisse können mit dem Befehl rmdir (remove directory) gelöscht werden. Verzeichnisse können so aber nicht gelöscht werden, wenn sie noch Dateien enthalten. Eine leere Verzeichnishierarchie kann wieder mit dem Parameter `-p` oder `--parents` komplett gelöscht werden.

Das aktuelle Arbeitsverzeichnis kann mit dem Befehl cd gewechselt werden. Dieser Befehl

funktioniert wie unter DOS auch, allerdings beharrt er auch beim Wechsel in das Mutterverzeichnis (`..`) auf ein Leerzeichen zwischen `cd` und `..`

Wird der Befehl cd ohne Parameter eingegeben, so wechselt man in sein Homeverzeichnis.

Wird der Befehl in der Form

cd ~*username*

verwendet, so wechselt man ins Homeverzeichnis des angegebenen Users.

Um sich den Inhalt eines Verzeichnisses anzusehen, gibt es den Befehl ls (list). Er kann auf vielfältige Weise angewandt werden, wichtig sind die folgenden Parameter:

Befehl	Bedeutung
`ls`	Nur die Namen der Dateien und Verzeichnisse werden angegeben
`ls -l`	Langes Listing. Zugriffsrechte, Anzahl der Hardlinks, Eigentümer, Gruppe, Datum, Größe und Namen werden angegeben.
`ls -i`	Die Inode Nummern der Dateien werden mit angegeben.
`ls -a`	Auch die "versteckten" Dateien (deren Namen mit einem Punkt beginnt) werden angezeigt.

Werden ls keine Dateinamen mitgegeben, so zeigt der Befehl den Inhalt des aktuellen Verzeichnisses. Ansonsten wird ls die Dateien eines angegebenen Verzeichnisses oder die Informationen über die angegebenen Dateien zeigen.

Rekursive Bearbeitung

Die genannten Befehle zum Kopieren, Verschieben und Löschen von Dateien sind alle auch in der Lage, ganze Unterverzeichnisäste zu bearbeiten. Dazu muß folgendes erwähnt sein:

- Der Befehl cp (copy) kann Unterverzeichnisäste nur dann kopieren, wenn ihm als Parameter ein `-r, -R` oder `--recursive` mitgegeben wurde.
- Der Befehl rm (remove) kann Unterverzeichnisse nur dann löschen, wenn ihm als Parameter ein `-r, -R` oder `--recursive` mitgegeben wurde.
- Der Befehl mv (move) verschiebt grundsätzlich auch Unterverzeichnisse, allerdings nur innerhalb der Grenzen einer Partition (eines Dateisystems). Normale Dateien können auch von Partition zu Partition verschoben werden, Verzeichnisse nicht. mv kennt keinen Parameter `-r`
- Der Befehl ls (list) zeigt alle Dateien auch die der Unterverzeichnisse, wenn ihm der Parameter `-R` oder `--recursive` mitgegeben wurde.

Wildcards

Wildcards (oder auch Jokerzeichen) werden unter Linux nicht von den einzelnen Programmen, sondern von der Shell interpretiert. Alle der genannten Programme können ja - wie oben schon erwähnt - mehrere Dateien als Parameter mitbekommen. Die Shell ersetzt die Wildcard Konstruktion durch eine Liste von passenden Dateinamen - der Befehl selbst bekommt also das Wildcard Konstrukt selbst nie zu sehen.

An Wildcards gibt es unter Linux folgende Möglichkeiten:

Wildcard	Bedeutung
*	Das Sternchen steht für eine beliebige Folge beliebiger Zeichen, auch die leere Folge. Ein einzelnes Sternchen meint also alle Dateien mit Ausnahme der Dateinamen, die mit einem Punkt beginnen. Das MSDOS-Konstrukt `*.*` würde nur alle Dateien ansprechen, die irgendwo (auch am Anfang oder Ende) einen Punkt im Namen vorweisen.
?	Das Fragezeichen steht für genau ein beliebiges Zeichen.
`[...]`	Eine Menge von Zeichen in eckigen Klammern steht für genau ein Zeichen aus dieser Menge.
`[...-...]`	Weiterentwicklung des Mengenprinzips. So ist es auch möglich, Bereiche anzugeben, ohne alle Zeichen anzugeben. Beispiel `[0-9]` oder `[a-zA-Z]`. Wieder ist genau ein Zeichen aus der Menge gemeint.
`[!...]`	Steht am Anfang der Menge ein Ausrufungszeichen, so wird der Wahrheitswert der Menge umgedreht (NOT) - das Konstrukt steht also für genau ein Zeichen, aber dieses Zeichen darf **nicht** in der angegebenen Menge stehen.

All diese Konstrukte dürfen beliebig miteinander kombiniert werden, jede einzelne Wildcard darf auch mehrfach vorkommen. So würde etwa die Konstruktion

```
[A-Z]*.[0-9][0-9][0-9]_[!A]
```

jede Datei ansprechen, deren Name mit einem Großbuchstaben (`[A-Z]`) beginnt. Danach darf irgendwas (`*`) folgen, beliebig lang. Dann muß ein Punkt kommen, gefolgt von drei Ziffern (`[0-9]`). Dem folgt ein Unterstrich und dann ein beliebiges Zeichen - nur eben kein A.

Dateien finden mit find

Ein universelles Werkzeug für den Umgang mit Dateien ist das Programm find. Es ermöglicht das Suchen von Dateien anhand aller denkbaren Kriterien wie

- Namensmuster

- Zugriffs-, Modifikations- und Statusveränderungsdatum
- Eigentümer und Gruppenzugehörigkeit
- Dateiart und Zugriffsrecht
- Größe
- ...

find ermöglicht es darüber hinaus, beliebige Befehle auf die gefundenen Dateien anzuwenden, sie also zu löschen, kopieren, verschieben oder was auch immer. Im Abschnitt 1.104.8 - Auffinden von Systemdateien und Platzieren von Dateien an den korrekten Ort wird der Befehl genauer besprochen.

1.103.4 - Benutzen von Unix Streams, Pipes und Umleitungen

Beschreibung: Prüfungskandidaten sollten in der Lage sein, Ein- und Ausgabeströme umzuleiten und sie zu verknüpfen, um Textdaten effizient zu verarbeiten. Dieses Lernziel beinhaltet das Umleiten von Standardeingabe, Standardausgabe und Standardfehlerausgabe, das Umleiten der Ausgabe eines Kommandos in die Eingabe eines anderen Kommandos, das Verwenden der Ausgabe eines Kommandos als Argument für ein anderes Kommando und das gleichzeitige Senden einer Ausgabe sowohl an die Standardausgabe als auch in eine Datei.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **tee**
- **xargs**
- <
- <<
- >
- >>
- |
- ``

Grundlegendes Prinzip

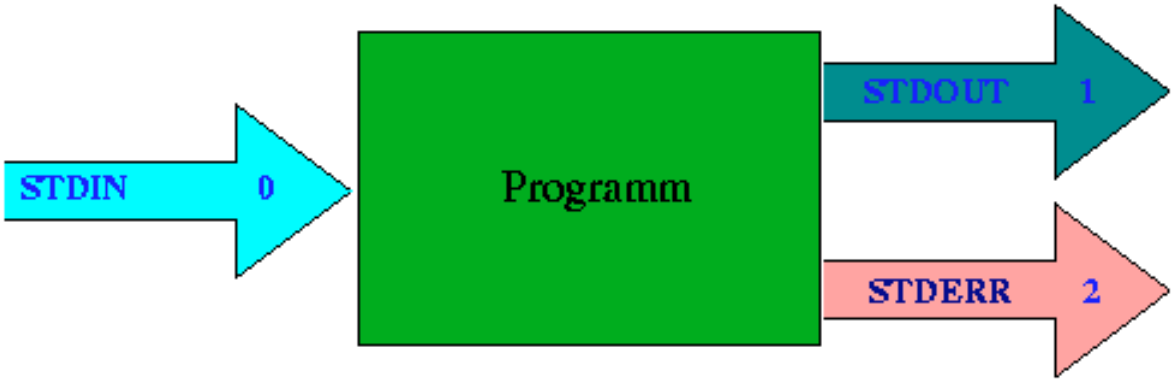
Jedes Programm unter Linux besitzt drei Standardkanäle für die Datenein/ausgabe. Diese drei Kanäle sind

Standard-Eingabe (STDIN)
: Der Kanal, von dem das Programm seine Eingabe erwartet. Dieser Kanal besitzt die Kenn-Nummer 0 und ist normalerweise mit der Tastatur verbunden.

Standard-Ausgabe (STDOUT)
: Der Kanal, auf den das Programm seine Ausgaben schreibt. Dieser Kanal besitzt die Kenn-Nummer 1 und ist normalerweise mit dem Monitor verbunden.

Standard-Fehlerausgabe (STDERR)
: Der Kanal, auf den das Programm seine Fehlerausgaben schreibt. Dieser Kanal besitzt die Kenn-Nummer 2 und ist wie STDOUT normalerweise mit dem Monitor verbunden.

Schematisch könnten wir das etwa folgendermaßen darstellen:



Der Grund für die Auftrennung von Standard Ausgabe und Standard Fehlerausgabe ist einfach der, daß wir diese Kanäle oftmals umleiten. Und eine Umleitung sorgt natürlich auch dafür, daß die Ausgabe jetzt nicht mehr auf dem Schirm erscheint. Käme es jetzt zu einem Fehler, so würden wir es nicht mitbekommen, weil die Ausgabe nicht sichtbar ist. Aus diesem Grund gibt es die separate Fehlerausgabe, die dafür sorgt, daß Fehlerausgaben immer noch auf dem Bildschirm zu lesen sind, auch wenn die Ausgabe umgeleitet wurde.

Umleitungen

Die Shell verwaltet die Ein- und Ausgabekanäle. Sie ist es auch, die diese Kanäle umleiten kann. Das Programm selbst merkt von dieser Umleitung nichts, es schreibt weiterhin auf die Standard Ausgabe und liesst von der Standard Eingabe. Die Shell bietet uns vier Möglichkeiten der Umleitungen. Zwei für die Eingabe und zwei für die Ausgabe:

Umleitung	Bedeutung
Programm > Datei	Das Programm leitet seine Standard Ausgabe in die Datei um, statt sie auf den Bildschirm zu schreiben. Existiert die Datei schon, so wird sie überschrieben, existiert sie noch nicht, so wird sie neu angelegt.
Programm >> Datei	Das Programm leitet seine Standard Ausgabe in die Datei um, statt sie auf den Bildschirm zu schreiben. Existiert die Datei schon, so wird die Ausgabe hinten an die bestehende Datei angehängt, existiert sie noch nicht, so wird sie neu angelegt.
Programm < Datei	Das Programm ließt seine Eingabe aus der Datei statt von der Tastatur.
Programm << EOM ... EOM	Das Programm ließt seine Eingaben statt von der Tastatur aus dem Block zwischen den beiden EOM Marken. Dabei dürfen diese Marken beliebige Worte sein, es gilt immer vom ersten bis zum zweiten Auftreten der Marke. Diese Konstruktion wird "here-document" genannt und findet Anwendung fast ausschließlich in der Scriptprogrammierung.

Manchmal ist es auch sinnvoll oder gewünscht, den Fehlerausgabekanal umzuleiten. Das ist über die Nennung seiner Kenn-Nummer vor dem Umleitungssymbol möglich. So würde die Anweisung

```
Programm 2> Fehlerprotokoll
```

die Fehlerausgabe des Programms in die Datei Fehlerprotokoll umleiten. Selbstverständlich ist das auch mit 2 Umleitungssymbolen (anhängend) möglich.

Und in einigen Fällen sollen beide Ausgabekanäle (STDOUT und STDERR) doch in eine Datei umgeleitet werden. Dazu kann man beide Kanäle zu einem zusammenfassen, der dann umgeleitet wird. Das geschieht mit der kryptischen Anweisung 2>&1. Aber vorsicht, um wirklich beide Kanäle in eine Datei umzuleiten muß die Bündelung nach der eigentlichen Umleitung vorgenommen werden:

```
Programm > Datei 2>&1
```

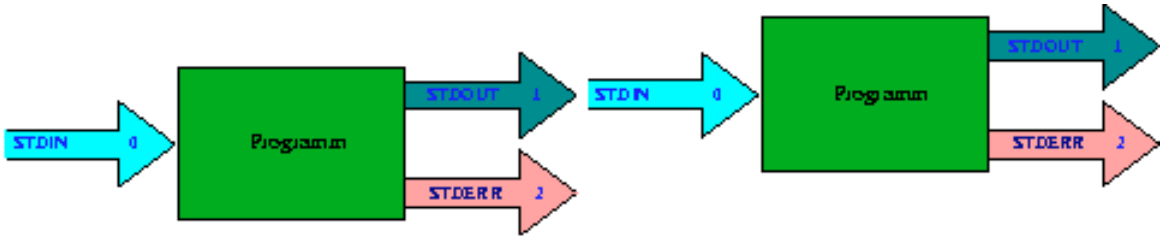
Diese Anweisung bündelt die Kanäle 2 und 1 (STDERR und STDOUT) und leitet beide in die Datei um.

Verkettung (piping)

Wenn wir jetzt zwei Programme haben, von denen das zweite die Ausgabe des ersten weiterverarbeiten sollte, so könnten wir jetzt die Ausgabe des ersten Programms in eine Datei umleiten und dann das zweite Programm aufrufen und seine Eingabe aus der Datei lesen lassen:

```
Programm1 > Datei
Programm2 < Datei
```

Das können wir auch ohne den Umweg über die Datei haben, indem wir zwei Programme direkt miteinander verbinden. Genauer gesagt verbinden wir die Standard-Ausgabe des ersten Programms mit der Standard-Eingabe des zweiten.



Dazu benutzen wir das Symbol | so daß wir einfach nur schreiben müssen:

```
Programm1 | Programm2
```

Das läßt sich beliebig fortsetzen, es ist also möglich, auch mehrere Programme hintereinanderzuhängen.

xargs

Das Programm xargs bietet eine Möglichkeit, die stark an die Kommandosubstitution erinnert. Hauptsächlich wird dieses Programm in Shells benötigt, die keine Substitution bieten, aber es gibt auch Fälle, in denen mit xargs elegantere Lösungen möglich sind, als mit der Kommandosubstitution.

xargs führt ein beliebiges Kommando aus, und gibt diesem Kommando als Parameter das mit, was xargs von der Standard-Eingabe gelesen hat. In einer Pipe ist es also möglich, daß ein Programm seine Ausgaben an xargs weiterleitet und xargs führt dann ein anderes Programm

aus, das mit eben der Ausgabe des ersten Programms als Parameter bestückt wird. Ein simples Beispiel:

Wir haben eine Datei mit Namen `Liste`, die folgenden Inhalt hat:

```
datei1
datei2
datei3 datei4
datei5 /home/hans
```

Jetzt rufen wir folgenden Befehl auf:

```
cat Liste | xargs cp
```

Der Befehl `cat Liste` gibt den Inhalt der Datei `Liste` auf die Standard-Ausgabe aus. Diese ist aber mit der Standard-Eingabe von `xargs cp` verbunden. xargs ruft also den Befehl `cp` auf und gibt ihm das mit, was es selbst von der Standard-Eingabe gelesen hat - also in unserem Fall den Inhalt der Liste. Zeilentrenner werden dabei wie Leerzeichen interpretiert. xargs ruft also folgenden Befehl auf:

```
cp datei1 datei2 datei3 datei4 datei5 /home/hans
```

Den selben Effekt hätten wir mit Kommandosubstitution mit folgender Zeile erreichen können:

```
cp `cat Liste`
```

Der Vorteil von xargs liegt darin, daß beliebig lange Ergebnisse mitgegeben werden können, weil xargs die Parameterkette aufteilt und das Programm entsprechend oft hintereinander aufruft. Die Kommandosubstitution kann das nicht und stößt irgendwann an die Grenzen der erlaubten Kommandolänge.

xargs kennt noch viele verschiedenen Kommandozeilenparameter, die aber für die LPI-101 Prüfung eher irrelevant sind.

Zwischensicherung

Die Ausnutzung komplexer Pipekonstruktionen erfordert es oft, daß einzelne Zwischenschritte während des Ablaufs in eine Datei gespeichert werden. Dazu gibt es das Programm tee, das einfach seine Standard-Eingabe auf seine Standard-Ausgabe weitergibt und daneben aber auch den Datenstrom in eine Datei zwischenspeichert. Es handelt sich also um eine Art T-Stück in einer Pipe (Rohrleitung) - daher stammt auch der Name.

Das folgende Konstrukt gibt ein Beispiel für die Anwendung:

```
Programm1 | tee Datei1 | Programm2 | tee Datei2 | Programm3 > Datei3
```

Das Programm1 schickt seine Ausgabe an das Programm tee, das sie dann in die Datei1 schreibt, aber gleichzeitig auch wieder an das Programm2 weitergibt. Dieses Programm gibt wiederum seine Ausgabe auf eine weitere Instanz von tee weiter, welches sie wieder in die

Datei2 schreibt und an das Programm3 weiterleitet. Das dritte Programm schließlich speichert sein Ergebnis in der Datei3. Wir haben also alle Zwischenergebnisse einzeln gespeichert, also das gleiche Ergebnis, als hätten wir geschrieben:

```
Programm1 > Datei1
Programm2 < Datei1 > Datei2
Programm3 < Datei2 > Datei3
```

1.103.5 - Erzeugung, Überwachung und Terminierung von Prozessen

Beschreibung: Prüfungskandidaten sollten in der Lage sein, Prozesse zu verwalten. Dieses Lernziel beinhaltet das Ausführen von Prozessen in Vorder- und Hintergrund, das Bringen eines Jobs vom Hintergrund in den Vordergrund und umgekehrt, das Starten eines Prozesses, der ohne Verbindung zu einem Terminal laufen soll, und die Mitteilung an ein Programm, daß es nach Abmelden weiterlaufen soll. Ebenfalls enthalten ist die Überwachung aktiver Prozesse, die Auswahl und das Sortieren von Prozessen für die Ausgabe, das Senden von Signalen an Prozesse, das Terminieren von Prozessen sowie das Erkennen und Terminieren von X-Anwendungen, die nach dem Schließen der X-Sitzung nicht ordnungsgemäß beendet wurden.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- &
- **bg**
- **fg**
- **jobs**
- **kill**
- **nohup**
- **ps**
- **top**

Ausführen von Prozessen im Vorder- und Hintergrund

Prozesse werden von der Shell grundsätzlich im Vordergrund gestartet, indem einfach eine Kommandozeile eingegeben wird. Die Shell wartet dann bis der Prozess beendet ist und kehrt erst danach zur Eingabeaufforderung zurück.

Damit die Eingabeaufforderung sofort wieder zur Verfügung steht, kann ein Prozess im Hintergrund gestartet werden, indem der Kommandozeile einfach ein Ampersand (&) angehängt wird.

Dadurch kehrt die Shell sofort wieder zur Eingabeaufforderung zurück, der gestartete Prozess läuft jetzt im Hintergrund.

Diese Ausführungsart eignet sich jedoch nur für Prozesse, die nichts auf die Standard-Ausgabe schreiben (oder deren Ausgaben umgeleitet werden), weil auch die Hintergrundausführung die Ausgaben auf der Standard-Ausgabe belässt. Ein Hintergrundprozeß, der permanent auf den Bildschirm Meldungen ausgibt, ist wohl eher störend.

Häufig wird diese Fähigkeit benutzt, um in der graphischen Oberfläche aus einem XTerm heraus graphische Programme aufzurufen. Durch die Hintergrundausführung kehrt die Shell sofort zur Eingabeaufforderung zurück, das XTerm ist also nicht solange gesperrt, solange das graphische Programm läuft.

Wechsel zwischen Vorder- und Hintergrundausführung

Die Shell bietet auch Möglichkeiten, Prozesse aus dem Vordergrund in den Hintergrund zu schieben und umgekehrt. Dazu sind fünf Funktionen erforderlich:

Strg-Z
: Hält einen Vordergrundprozess an. Dadurch wird dem Job jegliche Rechenzeit entzogen und die Shell kehrt zur Eingabeaufforderung zurück.

Strg-C
: Killt einen Vordergrundprozess. Dadurch wird der Job beendet und die Shell kehrt zur Eingabeaufforderung zurück.

`bg` [*Jobnummer*]
: Startet einen angehaltenen Job im Hintergrund. Der Job bekommt wieder Rechenzeit, die Shell kehrt aber zur Eingabeaufforderung zurück.

`fg` [*Jobnummer*]
: Startet einen angehaltenen Job oder einen im Hintergrund laufenden Job im Vordergrund. Die Shell kehrt nicht mehr zur Eingabeaufforderung zurück.

`jobs`
: Zeigt eine Liste aller Jobs, Vordergrund- Hintergrund und gestoppte Jobs samt ihren JobIDs.

Zusammengefasst kann gesagt werden, daß jede Shell beliebig viele Hintergrundjobs und maximal einen Vordergrundprozess besitzen kann. Mit den oben genannten Funktionen ist es möglich jede beliebige Kombination herzustellen. Die Befehle `fg`, `bg` und `jobs` sind eingebaute Shellfunktionen. Wichtig ist die Unterscheidung zwischen JobID und ProzessID. Die JobID bezieht sich auf die jeweilige Shell, die ProzessID ist von überall her gültig und im ganzen System eindeutig.

Prozesse unabhängig von der Terminalsitzung starten

Normalerweise wird ein Programm von einer Shell aufgerufen. Die Shell startet dann einen weiteren Prozeß, dessen Elternprozeß sie selbst darstellt. Wenn die Shell jetzt beendet wird, etwa durch ein **exit**, durch ein **Strg-d** oder den Befehl **logout**, wird der neu gestartete Befehl dadurch automatisch auch beendet.

In manchen Fällen kann es jedoch auch gewünscht sein, daß ein Prozeß auch nach dem Abmelden weiterläuft. Diese Möglichkeit bietet uns das Programm **nohup**. Es startet einen Befehl, und ignoriert dann das Hangup-Signal (SIGHUP) für diesen Prozeß. Dieses Signal wird gewöhnlich beim Abmelden an alle Prozesse gesendet, um ihnen mitzuteilen, daß die Sitzung jetzt geschlossen wird. (Hangup steht für Auflegen, also das Beenden einer Modemverbindung)

Die Aufrufform von **nohup** ist einfach, außer den Kommandozeilenparametern `--help` und `--version` existieren keine weiteren Schalter. Dem Programmnamen wird einfach der Name des zu startenden Befehls mitgegeben:

```
nohup Befehl
```

Dadurch wird der angegebene Befehl ausgeführt, und bleibt auch nach dem Abmelden weiter aktiv im Speicher. Die Ausgaben des gestarteten Prozesses werden nicht mehr auf die

Standard-Ausgabe geschrieben (weil die ja nach dem Abmelden gar nicht mehr zur Verfügung steht), sondern an die Datei `nohup.out` angehängt. Der Prozeß hat also seine Verbindung zum Terminal verloren, er kann nur noch über seine ProzeßID angesprochen werden.

Überwachung von Prozessen

Laufende Prozesse können überwacht werden mit den Programmen ps und top.

Das Programm ps bietet einen Schnappschuß der gerade laufenden Prozesse. Es zeigt verschiedene Daten dieser Prozesse an, das wichtigste ist dabei die ProzessID (PID) des jeweiligen Prozesses. Ohne Parameter aufgerufen zeigt ps nur die eigenen Prozesse. Häufige Kombinationen von Parametern sind

`ps uax`
: Zeigt eine Liste aller (a) Prozesse des Systems inclusive der Angabe der User (u) denen die Prozesse gehören, es werden auch daemon-Prozesse ohne eigenes Terminal (x) angezeigt.

`ps fax`
: Zeigt eine Liste aller (a) Prozesse des Systems wieder mit den daemon-Prozessen (x) an. Die Prozesse werden als Baumstruktur (f) angezeigt, die klar macht, welche Prozesse von welchen Eltern-Prozessen abstammen.

Im Gegensatz zu ps zeigt top eine ständig aktualisierte Liste der Prozesse mit der höchsten CPU-Auslastung. Top ist ein interaktives Programm, das auch Modifikationen der Prozesse zulässt. top läuft solange, bis es mit q beendet wird.

Ansprechen und Beenden von Prozessen

Unter Linux können laufende Prozesse angesprochen werden, genauer gesagt können ihnen Signale geschickt werden. Wie die jeweiligen Prozesse auf Signale reagieren hängt davon ab, was der Programmierer des jeweiligen Programms vorgesehen hat.

Die wichtigste Aufgabe von Signalen ist es, Prozesse zu beenden. Das Programm zum Versenden von Signalen heißt daher folgerichtig kill. Es gibt aber auch viele andere Aufgaben für Signale, so kann etwa ein Prozess durch ein Signal dazu aufgefordert werden, seine Konfigurationsdatei neu einzulesen. Damit kann ein laufender Prozess verändert werden, ohne ihn neu starten zu müssen.

Das Programm kill erwartet eine oder mehrere ProzessIDs als Parameter, die die Prozesse beschreiben, denen das Signal geschickt werden soll. Wenn kein spezielles Signal angegeben wird, so schickt kill den Prozessen das Signal Nr. 15 (SIGTERM), das einen Prozess auffordert, sich selbst zu beenden. Es gibt ein Signal, das immer "tödlich" ist, es hat die Nummer 9 (SIGKILL).

Die Syntax von kill ist einfach:

kill [-*Signal*] PID [PID ...]

Ein Parameter, mit führendem Bindestrich wird als Signal interpretiert, alle anderen als PIDs. Signale können entweder in ihrer numerischen Form oder über ihern Namen angegeben werden

- also z.B. entweder `-9` oder `-KILL`.

Eine Liste aller unterstützten Signale bekommen Sie mit `kill -l`

1.103.6 - Modifizeren von Prozeßprioritäten

Beschreibung: Prüfungskandidaten sollten in der Lage sein, Prozeßprioritäten zu verwalten. Dieses Lernziel beinhaltet das Ausführen von Prozessen mit höherer oder niedrigerer Priorität, das Bestimmen der Priorität eines Prozesses und das Ändern der Priorität eines laufenden Prozesses.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **nice**
- **ps**
- **renice**
- **top**

Ausführen von Programmen mit verschiedenen Prioritäten

Prozesse, die ohne spezielle Handhabe gestartet werden, haben eine "normale" Priorität, das heißt, sie sind mit einem sogenannten Nice-Wert von 0 ausgestattet. Ein Prozess ist dann "nice" (engl. für nett), wenn er anderen Prozessen mehr Rechenzeit übriglässt, also selber weniger verbraucht.

Sowohl das Programm ps, als auch top sind in der Lage, die NICE-Werte eines Prozesses darzustellen.

Ein normaler User kann Programme mit einem höheren Nice-Wert starten, indem er das Programm nice benutzt. Dieses Programm startet Prozesse mit einem veränderten Nice-Wert, also mit mehr oder weniger Priorität.

Hoher Nice-Wert heißt geringere Priorität

Der Superuser (root) kann Prozesse auch mit negativen Nice-Werten starten, also ihre Priorität erhöhen.

Der maximale Nice-Wert ist 19, der minimale -20.

Das Programm `nice` selbst ist einfach anzuwenden, es wird aufgerufen mit dem gewünschten Nice-Wert und der Angabe, welches Programm gestartet werden soll. Wollen wir also etwa als Normaluser das Programm `foo` mit einem Nicewert von 15 starten, dann schreiben wir:

```
nice -n15 foo
```

oder einfach

```
nice -15 foo
```

Vorsicht: im letzten Beispiel geht es nicht um -15 sondern um 15. Der Bindestrich hat nur die Aufgabe, dem nice-Programm klar zu machen, daß es sich nicht um das Programm 15 handelt! Damit der Superuser auch negative Werte angeben kann, muß er den Parameter -n benutzen. Damit könnte er also schreiben:

```
nice -n-15 foo
```

und der Nice-Wert wäre auf -15 gesetzt.

Das Programm nice ist nur gemacht, um Programme mit unterschiedlichen Prioritäten zu starten. Ein nachträgliches Ändern der Priorität ist damit nicht möglich.

Festlegen und Verändern der Prioritäten laufender Prozesse

Um die Priorität bzw. den Nice-Wert bereits laufender Prozesse zu verändern, gibt es zwei Möglichkeiten. Zum einen das speziell zu diesem Zweck entwickelte Programm renice und zum anderen das Programm top, das wir schon bei der Prozessbeobachtung kennengelernt haben.

Beide Programme erlauben es, daß ein laufender Prozess seinen Nice-Wert verändert bekommt. Normaluser dürfen den Nice-Wert bei beiden Programmen nur vergrößern. Das heißt, das auch ein Normaluser, der selbst seinem eigenen Prozess einen Nicewert von beispielsweise 10 gegeben hat, diesen Wert nicht mehr auf 5 runtersetzen kann.

Das Programm renice ist im Gegensatz zu top auch in der Lage, alle Prozesse eines Users oder einer Gruppe gleichzeitig zu verändern. top kann dagegen nur einzelnen PIDs einen neuen Nice-Wert zuweisen.

1.103.7 - Durchsuchen von Textdateien mittels regulärer Ausdrücke

Beschreibung: Prüfungskandidaten sollten in der Lage sein, Dateien und Textdaten mittels regulärer Ausdrücke zu bearbeiten. Dieses Lernziel beinhaltet das Erzeugen einfacherer regulärer Ausdrücke mit verschiedenen Elementen. Ebenfalls enthalten ist die Verwendung von Regex-Werkzeugen zum Durchführen von Suchen über ein Dateisystem oder Dateiinhalte.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **grep**
- **regexp**
- **sed**

Das Grundprinzip regulärer Ausdrücke

Reguläre Ausdrücke (regular expressions - regexp) sind nah verwandt mit den Wildcards, aber nicht das Selbe. Es handelt sich um Suchmuster, die auch eine Art Jokerzeichen beinhalten, jedoch wesentlich mächtiger sind, als die Wildcards der Shell.

Die unterschiedlichsten Programme unter Linux können mit regulären Ausdrücken umgehen, allerdings unterscheiden sie sich manchmal im Detail, in der Interpretation. Es gibt aber eine gemeinsame Grundmenge von Ausdrücken, die von allen Programmen verstanden werden.

Die LPI-101-Prüfung verlangt nur das Wissen um einfache reguläre Ausdrücke, die die folgenden Elemente enthalten dürfen:

Ausdruck	Bedeutung
c	Ein einzelner Buchstabe passt auf sich selbst.
.	Ein Punkt steht für genau einen beliebigen Buchstaben, außer das Zeilenende.
\?	Das dem Operator \? voranstehende Zeichen null oder einmal.
*****	>Das dem Operator * voranstehende Zeichen null mal oder öfter.
\+	Das dem Operator \+ voranstehende Zeichen ein mal oder öfter.
^	Das Caret (^) bedeutet Zeilenanfang.
$	Das Dollarzeichen ($) bedeutet Zeilenende.
\<	Bedeutet Wortanfang.
\>	Bedeutet Wortende.
[*Buchstaben***]**	Ein Buchstabe aus der Menge.

[^*Buchstaben***]**	Ein Buchstabe, der NICHT in der Menge steht.
\(...\)	Zusammenfassung eines Ausdrucks. Wichtig zum Ersetzen.
|	Ein logisches ODER mit dem verschiedene Ausdrücke verknüpft werden können.
****	Jedes Zeichen nach dem Backslash verliert seine Sonderbedeutung.

Mit Hilfe dieser Ausdrücke sind sehr komplexe Suchmuster möglich, nehmen wir ein Beispiel:

Wir haben eine Datei mit Namen und wir wollen alle Formen von Maier (Maier, Mayer, Meier, Meyer, Mayr) ansprechen. Der reguläre Ausdruck würde nun lauten:

```
M[ae][iy]e\?r
```

Wir suchen also ein Wort, das mit einem `M` beginnt, danach entweder ein a oder ein e vorweist. Dem folgt entweder ein `i` oder ein `y`. Dann kommt ein oder kein `e`, gefolgt vom Buchstaben `r`.

Wichtig ist zu wissen, daß die einzelnen Ausdrücke auch kombinierbar sind. So bedeutet etwa `.*` *eine Folge von beliebigen Zeichen, auch die leere Folge.* Warum? Ganz einfach. Der Punkt meint *ein beliebiges Zeichen*, das Sternchen meint *das dem Sternchen vorangehende Zeichen belibig oft, auch Null mal*. In Kombination wird daraus eben die beliebige Folge beliebiger Zeichen.

Zu beachten ist, daß diese Suchmuster viele Zeichen enthalten, die von der Shell interpretiert würden. Wenn wir also ein solches Muster auf der Kommandozeile eingeben, dann müssen wir es in Anführungszeichen setzen.

Aber genug der Theorie, jetzt folgen die zwei Anwendungen, mit denen wir diese Ausdrücke nutzen können:

Dateien durchsuchen mit grep

Das Programm grep durchsucht entweder Dateien oder einen Datenstrom (seine Standard-Eingabe) nach einem regulären Ausdruck. Dabei wird standardmäßig jede Zeile, die den Ausdruck enthält auf die Standard-Ausgabe geschrieben.

Damit ist es sowohl möglich, bestimmte Dateien zu durchsuchen, als auch die Ausgabe anderer Programme. Im einfachsten Fall durchsucht grep die Dateien nur nach einem festen Suchbegriff. Wenn Sie z.B. alle Useraccounts sehen wollen, die als Startshell die bash haben, könnten Sie das mit grep ganz leicht erledigen:

```
grep bash /etc/passwd
```

Das heißt also, die Datei /etc/passwd wird nach dem Auftreten des Begriffs "bash" durchsucht und alle Zeilen, die den Begriff enthalten werden ausgegeben. Wenn mehrere Dateien durchsucht werden sollen, dann gibt grep in der Ausgabe auch noch den Dateinamen der Datei aus, in der der Suchbegriff gefunden wurde.

Wenn grep einen Datenstrom durchsuchen soll, so wird ihm einfach kein Dateiname mitgegeben. Ein Beispiel: Wir wollen alle Prozesse des Users "hans" aufgelistet bekommen. Der Befehl "ps uax" listet uns alle laufenden Prozesse samt ihrer Eigentümer auf. Der Befehl

```
ps uax | grep hans
```

filtert die Ausgabe des ps uax Befehls durch grep und gibt nur die Zeilen aus, die den Begriff "hans" enthalten. Allerdings werden wir hier auch noch die Zeile angezeigt bekommen, die den Prozess "grep hans" beschreibt. Um das zu vermeiden, bedienen wir uns der regulären Ausdrücke:

```
ps uax | grep "^hans"
```

Das "^" Zeichen steht in einem regulären Ausdruck für den Zeilenanfang. Also suchen wir jetzt nach dem Auftauchen des Wortes "hans" am Zeilenanfang. Jetzt haben wir tatsächlich nur die Prozesse des Users hans.

Ein wichtiger Parameter von grep ist das `-v`. Mit dieser Kommandozeilenoption gibt grep nur die Zeilen aus, die **nicht** den Suchbegriff enthalten.

Anstatt eines einfachen Suchbegriffs können wir natürlich auch mit komplexen regulären Ausdrücken suchen. Um etwa aus einer Datei mit Adressangaben alle Adressen aller Formen des Namens Maier zu suchen, könnten wir schreiben:

```
grep "[Mm][ae][iy]e\?r" Adressen.txt
```

Bitte beachten Sie die Anführungszeichen. Sie verhindern, daß die Shell selbst versucht, die eckigen Klammerm und das Fragezeichen als Wildcard zu interpretieren. Grundsätzlich ist es von Vorteil, wenn Sie den Suchbegriff in solche Anführungszeichen setzen, sobald er mehr als nur Buchstaben und Zahlen beinhaltet.

Suchen und Ersetzen mit sed

Mit grep haben wir Dateien nach Suchbegriffen durchsucht, wir konnten diese Begriffe aber nicht ändern. Dazu sind Editoren notwendig. Alle Unix-Editoren wie etwa vi, ex, ed oder emacs können reguläre Ausdrücke verwenden, um ein "Suchen und Ersetzen" durchzuführen. Aber eine kommandozeilenorientierte Funktion, die dieses Ersetzen bietet, wird mit einem ganz speziellen Editor durchgeführt, dem Stream-Editor oder kurz sed.

Dieser Editor ist dazu entworfen, einen Datenstrom oder eine Datei automatisch zu bearbeiten. Dazu werden bestimmte Befehle, die auf diese Datei oder diesen Datenstrom anzuwenden sind, entweder in einer Datei formuliert oder gleich auf der Kommandozeile miteingetragen. Der Stream-Editor bearbeitet dann die Datei oder den Datenstrom zeilenweise und führt die jeweiligen Befehle darauf aus. Das Ergebnis wird dann wiederum auf die Standard-Ausgabe geschrieben. Wir haben das schon auf der Seite über die Textfilter besprochen.

Die mit Abstand häufigste Anwendung dieses Stream-Editors ist das Suchen und Ersetzen mit regulären Ausdrücken. Diese Vorgehensweise soll hier eingehend beschrieben werden.

Ein sed-Kommando ist immer in einen Adressbereich und einen Befehl aufgeteilt. Das Adressfeld kann dabei folgende Werte aufnehmen:

Eine Zahl *n*
: Eine einfache Zahl *n* beschreibt die Zeile *n*. Der folgende Befehl wird nur auf diese Zeile angewandt.

Ein Zahlenbereich *n,m*
: Ein Bereich meint die Zeilen *n* bis *m*. Start- und Endzeile werden durch Komma getrennt. Ein Dollarzeichen ($) meint die letzte Zeile. So bedeutet also `1,$` alle Zeilen der Datei (von der Zeile 1 bis zur letzten Zeile). Der folgende Befehl wird auf jede Zeile angewandt, die innerhalb des genannten Bereiches liegt.

Ein regulärer Ausdruck */Ausdruck/*
: Wenn als Adresse ein regulärer Ausdruck steht, dann ist jede Zeile gemeint, die ein Element enthält, das auf den Ausdruck passt. Damit sed das Konstrukt als Ausdruck erkennt, muß der Ausdruck in Slash-Zeichen (/) geklammert sein.
Wenn ein regulärer Ausdruck innerhalb einer Adressbereichsangabe steht, dann ist immer die erste Zeile gemeint, auf die der Ausdruck passt.
Damit ein Ausdruck auch nach dem Auftreten von Slashes suchen kann, erlaubt sed auch die Verwendung einer anderen Klammerung, in der Form `\#`, wobei für # jedes beliebige Zeichen stehen kann. Also ist auch die Konstruktion `\+Ausdruck\+` eine legale Angabe einer Adresse.

Wird bei sed der Adressteil weggelassen, so wird jede Zeile des Datenstroms bearbeitet, es wird also die Adresse `1,$` angenommen.

Der Befehl zum Suchen und Ersetzen für sed ist das `s`, das wie folgt angewandt werden muß:

```
s/Suchbegriff/Ersetzung/[Optionen]
```

Die Angabe der Optionen ist optional (daher die eckigen Klammern), der abschließende Slash nach dem Ersetzungsteil ist aber zwingend, auch wenn keine Optionen angegeben werden sollen.

Als Optionen stehen zur Verfügung:

Eine Zahl *n*
: Eine Zahl von 1 bis 512 ersetzt nur das *n*-te Auftreten des Musters.

g
: Global - jedes Auftreten des Begriffes innerhalb einer Zeile wird bearbeitet. Ohne diese Option wird immer nur das erste Auftreten des Begriffs bearbeitet.

p
: Print -wenn eine Ersetzung stattgefunden hat, wird der Inhalt des Arbeitsspeichers von sed in die Standardausgabe geschrieben

Im einfachsten Fall können wir also z.B. das Auftreten des Wortes "Huber" in "Herr Huber" mit dem folgenden Befehl ersetzen:

```
s/Huber/Herr Huber/g
```

Wie wird das nun angewandt? Entweder, wir geben diesen Befehl gleich auf der

Kommandozeile ein (mit der Option `-e`, dann würde das etwa folgendermaßen aussehen:

```
cat Datei.txt | sed -e "s/Huber/Herr Huber/g" > Datei2.txt
```

oder einfacher

```
sed -e "s/Huber/Herr Huber/g" Datei.txt > Datei2.txt
```

Der Editorbefehl wurde hier in Anführungszeichen gesetzt, weil er ein Leerzeichen enthält. Wie schon bei grep ist es grundsätzlich zu empfehlen, diese Konstruktion in Anführungszeichen zu setzen um die Shell davon abzuhalten, Sonderzeichen zu interpretieren.

Was hat unser Befehl jetzt bewirkt? Jedes Vorkommen des Wortes "Huber" in der Datei `Datei.txt` wurde in "Herr Huber" verwandelt. Die gesamte Datei mit den Änderungen wurde in die Datei `Datei2.txt` geschrieben.

Wir hätten den Befehl aber auch in eine separate Datei (nennen wir sie mal `Befehle`) schreiben können. Das macht insbesondere dann Sinn, wenn es sich um mehrere Befehle handelt, oder wir die Ersetzung immer wieder brauchen werden. Dann hätte der sed-Aufruf einfach statt der Option `-e` ein `-f` enthalten, gefolgt vom Namen der Befehlsdatei:

```
sed -f Befehle Datei.txt > Datei2.txt
```

Die eigentliche Stärke dieses Suchen und Ersetzen liegt aber natürlich wieder in der Angabe von regulären Ausdrücken. Es können hier beliebige Ausdrücke verwendet werden, wie sie oben beschrieben sind. Als besondere Möglichkeit sei hier noch auf die Klammerung mit der Konstruktion `\(...\)` verwiesen. Auf jedes Element, das im Suchen-Teil derart geklammert ist, kann im Ersetzen-Teil wieder zugegriffen werden. Dazu muß nur eine Nummer *n* mit vorgestelltem Backslash (\) angegeben werden. Das meint die *n*-te Klammer. Ein Beispiel:

Wir haben eine Datei `Namen.txt` mit folgendem Inhalt:

```
Hans Müller
Peter Schmidt
Michael Huber
Gabi Maier
Thomas Gruber
```

Wir wollen jetzt die Namen in der Form Nachname, Vorname haben. Kein Problem mit sed. Wir schreiben

```
sed -e "s/\(.*\) \(.*\)/\2, \1/" Namen.txt > Namen2.txt
```

Schon steht in der Datei `Namen2.txt`

```
Müller, Hans
Schmidt, Peter
Huber, Michael
Maier, Gabi
Gruber, Thomas
```

Was ist passiert? Schauen wir uns den sed-Befehl einmal genauer an. Der reguläre Ausdruck `.*` steht für eine beliebige Zeichenfolge beliebiger Länge. Also können wir die zwei Namen (Vor- und Nachname) auch als `.* .*` angeben. Also zwei beliebige Zeichenfolgen, getrennt durch ein Leerzeichen. Jetzt klammern wir die beiden Konstrukte jeweils mit `\(...\)` ein. So entsteht zweimal der Ausdruck `\(.*\)`. Auf jeden dieser geklammerten Ausdrücke können wir jetzt im Ersetzen-Teil des sed-Befehls wieder zugreifen, mit `\1` für den ersten gefundenen Ausdruck und `\2` für den zweiten.

Dadurch entsteht folgende Zuordnung, am Beispiel der ersten Zeile:

`\(.*\)`	\(.*\)
Hans	Müller
`\1`	`\2`

Im Ersetzen-Teil schreiben wir einfach `\2, \1`, das wird dann eben ersetzt durch den Inhalt der zweiten Klammer, gefolgt von einem Komma, einem Leerzeichen und dem Inhalt der ersten Klammer. Also in unserem Beispiel durch `Müller, Hans`.

1.103.8 - Allgemeine Dateibearbeitung mit vi

Beschreibung: Prüfungskandidaten müssen in der Lage sein, Textdateien mit **vi** zu bearbeiten. Dieses Lernziel beinhaltet Navigation im vi, die grundlegenden vi Modi, Einfügen, Bearbeiten, Löschen, Kopieren und Auffinden von Text.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **vi**
- `/, ?`
- `h,j,k,l`
- `G, H, L`
- `i, c, d, dd, p, o, a`
- `ZZ, :w!, :q!, :e!`
- `:!`

Der Unix-Standard-Editor ist - unabhängig vom verwendeten System - immer das Programm **vi**. Dieser Editor ist auf allen Terminals lauffähig, das bedeutet aber auch, daß er sich nicht auf die Existenz bestimmter Funktionstasten verlassen kann, um bestimmte Befehle auszuführen. Einfache Terminals haben nicht einmal Pfeiltasten, der Editor **vi** muß also auch die Möglichkeit haben, mit anderen Befehlen Aktionen wie die Bewegungen des Cursors vorzunehmen.

Aus diesem Grund kennt der Editor **vi** verschiedene Modi. Die beiden wichtigsten sind der Kommandomodus und der Eingabemodus. Im Kommandomodus werden alle eingegebenen Zeichen als Befehle interpretiert, statt sie als in den Text einzugebende Zeichen zu interpretieren. Im Eingabemodus werden alle Zeichen an der Cursorposition in den Text eingefügt. Der Wechsel vom Eingabemodus in den Kommandomodus erfolgt immer durch die **Esc**-Taste. Beim Aufruf von **vi** befinden wir uns grundsätzlich im Kommandomodus.

Der Editor **vi** wird in der Regel zusammen mit einem Dateinamen aufgerufen.

```
vi Datei
```

Dadurch wird **vi** angewiesen, die angegebene Datei zu editieren. Existiert die Datei noch nicht, so startet **vi** mit einem leeren Puffer, beim Speichern wird dann aber der angegebene Dateiname verwendet.

Befehle zum Bewegen des Cursors im Kommandomodus

Nachdem **vi** nicht davon ausgehen kann, daß dem Terminal Pfeiltasten zur Verfügung stehen, muß er Befehle kennen, die die Aufgabe dieser Pfeiltasten übernehmen. Dazu dienen die Tasten `h j k l`, die im Kommandomodus folgende Bedeutung haben:

h	Bewegt den Cursor nach links
j	Bewegt den Cursor nach unten

- k Bewegt den Cursor nach oben
- l Bewegt den Cursor nach rechts

Jedem dieser Befehlszeichen kann eine Zahl vorangestellt werden, die angibt, wie oft der Befehl ausgeführt werden soll. Der Befehl `23j` bewegt den Cursor also 23 Zeilen nach unten.

Neben diesen grundlegenden Befehlen gibt es noch weitere Möglichkeiten den Cursor zu bewegen:

w	Bewegt den Cursor auf den Beginn des nächsten Wortes
b	Bewegt den Cursor auf den Beginn des aktuellen Wortes
0	Bewegt den Cursor auf den Beginn der aktuellen Zeile
$	Bewegt den Cursor auf das Ende der aktuellen Zeile
Strg-f	(Forward) Eine Bildschirmseite vorwärts (PgDn)
Strg-b	(Back) Eine Bildschirmseite zurück (PgUp)
G	Bewegt den Cursor zum Dateiende (letzte Zeile)
*n*G	Bewegt den Cursor in die *n*te Zeile der Datei
H	Bewegt den Cursor in die erste Zeile des aktuellen Bildschirms
*n*H	Bewegt den Cursor in die *n*te Zeile des aktuellen Bildschirms
L	Bewegt den Cursor in die letzte Zeile des aktuellen Bildschirms
*n*L	Bewegt den Cursor in die *n*te Zeile des aktuellen Bildschirms von unten

Auch diese Befehle (bis auf 0) können alle nach einer Zahl auftreten, die angibt, wie oft der Befehl ausgeführt werden soll. Mit `3w` bewegt sich der Cursor also auf den Beginn des dritten Wortes nach dem aktuellen Wort. Bei den Befehlen `G, H` und `L` ist angegeben, was die Zahl jeweils bedeutet.

Die modernen Versionen von **vi**, die unter Linux zum Einsatz kommen, kennen natürlich auch dei entsprechenden Funktionstasten der PC-Tastatur. Es ist also sehr wohl möglich, die Pfeiltasten, Home, End, PgUp und PgDn zu verwenden. Trotzdem sollten die hier besprochenen Befehle bekannt sein, zumindestens die vier grundlegenden Cursorbewegungen `h j k l`.

Befehle zum Wechsel in den Eingabemodus

Ein Editor ist dazu gedacht, Text in eine Datei zu schreiben, die wichtigsten Befehle sind also die, wie vom Kommandomodus in den Eingabemodus gewechselt werden kann. **vi** bietet hierfür viele verschiedenen Möglichkeiten an, die sich auch wieder aus der alten terminalbasierten Arbeitsweise erklären. Heute - mit Verwendung der Pfeiltasten - können wir bedenkenlos im Eingabemodus den Cursor bewegen. Früher - ohne Pfeiltasten - musste für jede Bewegung des Cursors in den Kommandomodus gewechselt werden. Aus diesem Grund gibt es so viele verschiedene Möglichkeiten, in den Eingabemodus zu wechseln:

- i Einfügen vor der aktuellen Cursorposition
- I Einfügen am Anfang der aktuellen Zeile
- a Einfügen nach der aktuellen Cursorposition

- A Einfügen am Ende der aktuellen Zeile
- o Einfügen nach der aktuellen Zeile
- O Einfügen vor der aktuellen Zeile
- R Überschreiben des Textes (Replace)
- C Ersetzen der aktuellen Zeile ab Cursorposition

Achtung: Auch diesen Befehlen kann eine Zahl vorangestellt werden. Der Befehl `23i` führt zwar auch nur einmal in den Eingabemodus, die geschriebenen Zeilen werden aber 23 mal eingefügt.

Suchen im Text

Um ein bestimmtes Wort - bzw. einen regulären Ausdruck - zu suchen, bietet **vi** zwei Methoden:

- /*Suchmuster* Sucht nach dem *Suchmuster* von der Cursorposition in Richtung Dateiende und bewegt den Cursor auf das gefundene Ergebnis.
- ?*Suchmuster* Sucht nach dem *Suchmuster* von der Cursorposition in Richtung Dateianfang und bewegt den Cursor auf das gefundene Ergebnis.

Beiden Methoden kann wiederum eine Zahl vorausgehen, die dann den Cursor nicht zum ersten, sondern zum *n*ten Auftreten des Suchmusters bewegt.

Löschen von Text

Um aus dem Kommandomodus heraus Text aus der Datei zu löschen, existieren viele verschiedenen Möglichkeiten. die wichtigsten sind

- x Löscht das Zeichen, auf dem der Cursor gerade steht.
- dl Löscht das Zeichen, auf dem der Cursor gerade steht (wie x).
- dd löscht die aktuelle Zeile
- dw löscht von der Cursorposition bis zum Anfang des folgenden Wortes
- db löscht von der Cursorposition nach links bis zum Anfang des aktuellen Wortes
- d$ löscht von der Cursorposition bis zum Zeilenende
- d0 löscht von der Cursorposition nach links bis zum Anfang der aktuellen Zeile

Die Löschbefehle speichern den gelöschten Text in einem Zwischenpuffer. Es ist möglich, diesen Zwischenpuffer wieder an einer beliebigen Stelle des Textes auszugeben. Somit ist auch eine Art Kopieren oder Verschieben von Textblöcken möglich. Die dazu nötigen Befehle sind:

- p Fügt den Inhalt des Zwischenpuffers nach der aktuellen Cursorposition ein.
- P Fügt den Inhalt des Zwischenpuffers vor der aktuellen Cursorposition ein.
- yy Liesst eine Zeile in den Zwischenpuffer, ohne sie zu löschen.

Es gibt auch noch eine raffinierte Methode, Text zu löschen oder zu ersetzen. Dazu wird ein Befehl zusammen mit einem der Befehle zur Cursorbewegung benutzt. Das Löschen bzw.

Ersetzen bezieht sich dann auf den Bereich zwischen der aktuellen Cursorposition und dem Punkt, an den der Befehl zur Cursorbewegung den Cursor bewegt hätte:

d*Cursor-Bewegung*	Löscht den Bereich zwischen aktueller Cursorposition und dem Punkt, an den die *Cursor-Bewegung* den Cursor geführt hätte.
c*Cursor-Bewegung*	Ersetzt den Bereich zwischen aktueller Cursorposition und dem Punkt, an den die *Cursor-Bewegung* den Cursor geführt hätte durch den Eingabetext.

Der Befehl d3k würde also den Text zwischen der aktuellen Cursorposition bis 3 Zeilen weiter nach oben (3k bewegt den Cursor 3 Zeilen nach oben) löschen. Als *Cursor-Bewegung* kann jeder beliebige Befehl angegeben werden, der den Cursor bewegt, also auch ein Suchbefehl.

Beenden und Sichern

Die meisten Befehle des **vi** zum Beenden und Sichern sind sogenannte Ex-Befehle, die durch einen führenden Doppelpunkt eingeleitet werden. Diese Befehle verweigern unter bestimmten Umständen ihren Dienst, wenn ihnen kein Ausrufungszeichen angefügt wird. So kann der **vi** nicht mit :q beendet werden, wenn der Text, den der Editor bearbeitet hat, verändert wurde, aber noch nicht gesichert ist. Durch das Anhängen eines Ausrufungszeichen an den Befehl (:q!) wird diese Einschränkung aufgehoben. Folgende Befehle sind in diesem Zusammenhang interessant:

ZZ	Sichert die Datei und beendet den Editor
:w *Dateiname*	Sichert die Datei. Wurde beim Aufruf von **vi** schon ein Dateiname angegeben, kann er hier weggelassen werden.
:w! *Dateiname*	Wie :w. Speichert aber auch, wenn es Gründe gibt, die das Speichern normalerweise verunmöglichen (Datei existiert schon)
:wq *Dateiname*	Speichern der Datei und Beenden des Editors (auch hier kann ein ! angegeben werden, wenn ansonsten das Speichern unmöglich wäre). Der Dateiname kann weggelassen werden, wenn er schon bekannt ist.
:x *Dateiname*	Wie :wq. Datei wird aber nur gespeichert, wenn es Veränderungen gab.
:q	Beendet den Editor
:q!	Beendet den Editor auch wenn vorher nicht gesichert wurde.

Sonstige wichtige Kommandos

Neben den bereits vorgestellten Befehlen gibt es noch hunterte anderer. Hier seien nur noch ein paar genannt, die wirklich wichtig sind:

:e *Dateiname*	Läd die angegebene Datei in den Editor. Nur ausführbar, wenn die aktuelle Datei schon gesichert wurde.
:e! *Dateiname*	Läd die angegebene Datei in den Editor. Auch ausführbar, wenn die aktuelle Datei noch nicht gesichert wurde.
:r *Dateiname*	Läd die angegebene Datei an der aktuellen Cursorposition in den Text.

:! *Shellbefehl*	Führt den angegebenen Shellbefehl aus und kehrt dann zum Editor zurück.
:*Bereich*! *Shellbefehl*	Filtert den angegebenen Bereich durch das angegebene Shellkommando und ersetzt ihn durch die Ausgabe des Shellkommandos. Bereiche werden durch ihre Zeilennummern angegeben in der Form *erste_Zeile,letzte_Zeile*. Ein Dollarzeichen bezeichnet die letzte Zeile der Datei. Soll die ganze Datei gefiltert werden, kann also die Angabe `1,$` benutzt werden. Die aktuelle Zeile wird durch einen Punkt (.) repräsentiert.
u	Undo der letzten Aktion.