Übung 1 (15.4.1999)

Aufgabe 1:
Melden Sie sich am System an. Auf die Aufforderung „Login:“ geben Sie `dvsp` ein und tippen ↵ und auf die Aufforderung „Password:“ tippen sie nur ein ↵ ein.

Aufgabe 2:
Nach erfolgreicher Anmeldung meldet sich das UNIX-System, genauer gesagt das Shellprogramm. Tippen Sie bitte das Kommando

```
mkdir Nachname_Vorname
```

(belegen Sie das mit Ihrem Nachnamen und Vornamen entsprechend) und schließen Sie das Kommand mit ↵ ab (das wird ab sofort nicht mehr dazugesagt). Dieses Kommando legt ein Unterverzeichnis an in dem Verzeichnis, in dem Sie sich gerade befinden.

Welche Ausgabe erzeugt das Kommando?

Geben Sie das Kommando ein zweites Mal. Ändert sich die Ausgabe?

Kontrollieren Sie mit dem Kommando `ls` (list files), wie sich der Inhalt des aktuellen Verzeichnisses geändert hat. Mit `pwd` (print working directory) können Sie sich den Pfadnamen des aktuellen Verzeichnisses ansehen.

Aufgabe 3:
Wechseln Sie in ihr persönliches Verzeichnis durch den Befehl

```
cd <Nachname>_<Vorname>
```

(belegen Sie das mit Ihrem Nachnamen und Vornamen entsprechend). Geben Sie nochmals das Kommando `pwd`. Was hat sich geändert?

Wie sieht also die Konvention für Pfadnamen in UNIX aus?

Gibt es Laufwerksbuchstaben (wie in DOS/Windows)?

Aufgabe 4:
Wir möchten nun Dateien im Verzeichnis `/bin` ansehen (dort stehen übrigens die Standardprogramme von UNIX). Wie geht das, wenn sie das Verzeichnis wechseln dürfen? Und wenn nicht?

Aufgabe 5:
Die Ausgabe von Aufgabe 4 ist sehr spartanisch. Finden Sie heraus, wie sie die Ausgabe von `ls` etwas ausführlicher gestalten können. (Tip: Zu praktisch jedem UNIX-Kommando gibt es eine Hilfe, die Sie mit dem Kommando `man kommandoname` ansehen können. Die Ausgabe erfolgt bildschirmweise und sie können mit der Leertaste weiterblättern und mit `q` abbrechen.

Probieren Sie `man` an Kommandos aus, die sie bereits kennen, und versuchen Sie eine generelle Struktur in der Darstellung zu erkennen.

Aufgabe 6:
Wir möchten nun die Namen der Dateien im Verzeichnis `/bin` in die Datei `hugo` in Ihrem Verzeichnis speichern. Generell gilt: UNIX-Programme lesen ihre Eingabe vom Terminal und schreiben die Ausgabe dorthin. Man kann der Shell mitteilen, daß sie den Datenfluß umlenken soll auf Dateien (das Programm bekommt davon nichts mit!!). Beispielsweise schreibt

```
ls -l /usr >otto
```

den Inhalt des Verzeichnisses /usr in Langform in die Datei otto im aktuellen Verzeichnis. Das Größerzeichen besagt: „leite Ausgabe um in“.

Stellen Sie Eigenschaften der Datei `hugo` fest (mit Mitteln, die Sie bereits kennen). Führen Sie das Kommando nochmals aus. Überlegen Sie sich vorher, welche Eigenschaften von `hugo` gleich bleiben und welche sich ändern.

Aufgabe 7:
Was passiert, wenn sie statt einem zwei Größerzeichen verwenden (...`>>otto`)? Machen Sie danach `ls -l otto`. Idee? Prüfen Sie Ihre Vermutung, indem Sie die Datei mit `more hugo` betrachten (Blättern und Beenden wie in Aufgabe 5).

Was passiert bei `cat hugo`?

Aufgabe 8:
Stellen Sie fest, wozu das Kommando `wc` gut ist. Starten Sie das Kommando. Was passiert?

Tippen Sie jetzt einige Zeilen mit einfachem Text ein (Sie dürfen auch Zeilenschaltungen verwenden) und schließen Sie Ihre Eingabe mit Strg-D ab. Was passiert und was bedeuten die Zahlen?

Vorsicht: Tippen Sie exakt einmal Strg-D. Wenn Sie dies in der Kommandozeile eingeben, müssen Sie sich neu einloggen (wenn Sie wollen, probieren Sie es aus!). Was könnte dieses Eingabezeichen also bedeuten?

Aufgabe 9:
Geben Sie das Kommando `wc <hugo`. Haben Sie eine Vermutung, warum Sie jetzt nichts zusätzlich eingeben müssen?

Vergleichen Sie das Ergebnis mit der Ausgabe von `ls -l hugo`. Versuchen Sie das Kommando so zu modifizieren, daß nur die Zahl der Zeichen von hugo ausgegeben wird.

Aufgabe 10:
Letztes Experiment mit wc. Geben Sie das Kommando `wc <hugo >otto`. Was steht in `otto`?

Wie kontrollieren Sie Ihre Hypothese?

Was sollte jetzt `wc <otto` liefern? Überlegen Sie vorher, ehe Sie es ausprobieren! Überrascht? Erklärung?

Aufgabe 11:
Geben Sie die Kommandos `mkdir fred >x1` und `mkdir fred >x2`. Was passiert und was steht in x1 und x2?

Erklärung: Fehler werden nicht auf Standardausgabe geschrieben, sondern auf die Standardfehlerausgabe...

Aufgabe 12:
Programme, die eine Eingabe lesen und daraus eine Ausgabe erzeugen heißen *Filter*. Man kann diese Filter kombinieren, um komplexere Transformationen zu machen. Dazu schreibt man die Kommandos in eine Eingabezeile und verbindet sie mit jeweils einem senkrechten Strich (einer „Pipe“) zu einer *Pipeline*.

Probieren Sie `ls -l /bin | wc -l`. Was wird angezeigt?

Pipes sind unidirektionale Pufferbereiche zwischen parallel laufenden Programmen. Es wird also nicht einfach das Ergebnis des ersten Prozesses auf Datei geschrieben und danach vom zweiten Programm abgearbeitet!

Haben Sie eine Erklärung, warum die Ausgaben von UNIX-Programmen so spartanisch sind (keine Kopfzeilen bei Tabellen, kein „50 Dateien im Verzeichnis“ usw.)?

Aufgabe 13:
Finden Sie die Zahl aller Dateien im Verzeichnis `/bin`, die im Namen nacheinander die Buchstaben „ch“ enthalten (also z.B. echo). Nützlich ist dafür das Kommando `grep`. Ihre Lösung soll ohne Zwischendateien auskommen.

Aufgabe 14:
Geben Sie das Kommando `sleep 5`. Was passiert? Konsultieren Sie die Hilfeseite von `sleep`.

Modifizieren Sie das Kommando in `sleep 5 &`. Wie ändert sich das Verhalten von `sleep` oder des Systems?

Ist `&` Ihrer Meinung nach ein Parameter von `sleep`? Testen Sie Ihre Hypothese an anderen Kommandos oder Kommandofolgen.

Aufgabe 15:
Geben Sie das Kommando `sleep 15 &` und unmittelbar anschließend das Kommando `ps`. Was sehen Sie? Haben Sie einen Trick, um nur Informationen über den sleep-Prozeß zu sehen?

Da Sie praktisch mehrfach eingeloggt sind (eine Kennung!!), können Sie nicht sofort herausfinden, welche Zeilen der Ausgabe von `ps` Sie an Ihrem Terminal betreffen. Schmökern Sie in der Dokumentation zu ps, ob es eine Spalte gibt, die Ihnen weiterhilft (Tip: Sie sitzen an einem bestimmten Terminal...). Erstellen Sie eine Kommandofolge, die nur die Prozesse anzeigt, die Sie angehen.

Aufgabe 16:
Crashkurs für Texteditor `vi` (der VW-Käfer der Unix-Editoren):
Start erfolgt mit `vi dateiname`. Mit der Taste `i` geht man in den Einfügemodus und kann lostippen. Wenn man fertig ist, muß man `ESC` tippen. Cursortasten sollten funktionieren. Löschen von Buchstaben entweder im Einfügemodus mit Rücktaste oder außerhalb davon mit `d` und Leerzeichen.

Um die Datei zu speichern, tippt man nun `:wq` (write and quit) und ist wieder in der Shell.

Schreiben Sie den Text `ls -l /bin | wc -l` in die Textdatei `zaehle`. Wenn Sie wieder in der Shell sind tippen Sie `sh zaehle`. Was passiert?

Aufgabe 17:
Verändern Sie mit `vi` die Datei `zaehle` wie folgt: Statt `/bin` schreiben Sie $1 (also `ls -l $1 | wc -l`). Speichern Sie die Datei und probieren Sie es aus.

Was passiert bei `sh zaehle`? Was passiert bei `sh zaehle /opt`?

Aufgabe 18:
Sie sind fertig (ich auch...)!

Übung 2 (22.4.1999)

Aufgabe 19:
Melden Sie sich am System an. Auf die Aufforderung “Login:” geben Sie `dvsp` ein und tippen ↵.
Wenn Sie auf einer neuen Maschine arbeiten, tippen Sie bitte das Kommando

```
mkdir Nachname_Vorname
```

(belegen Sie das mit Ihrem Nachnamen und Vornamen entsprechend). Wechseln Sie in ihr persönliches Verzeichnis durch den Befehl

```
cd Nachname_Vorname
```

Aufgabe 20:
Man kann der Shell mehrere Kommandos in einer Zeile geben, indem man Sie per Strichpunkt trennt z.B.

```
cd /bin; ls; cd
```

`cd` alleine (oder mit Parameter ~) wechselt in das Verzeichnis, das beim Login aktiv war, das *Heimverzeichnis* (home directory). Was passiert in obiger Zeile?

Aufgabe 21:
Geben Sie das Kommando

```
echo 3 2 hugo
```

und bekommen Sie heraus, was `echo` macht. Machen Sie mehr Leerzeichen oder Tabulatoren zwischen die Parameter und testen Sie, wie sich das auf die Ausgabe auswirkt.
(Leerzeichen sind generell Trenner von Parametern).

Was passiert bei

```
cd /bin; echo *
```

bei

```
cd /bin; echo ch*
```

und bei

```
cd /bin; ls -l ch*
```

Haben Sie eine Idee, was der Stern bedeutet?

Aufgabe 22:
Wer glauben Sie interpretiert den Stern: das Programm (`echo` bzw. `ls`) oder jemand anderes?

Prüfung der Hypothese:

1. Wechseln Sie in Ihr persönliches Verzeichnis und erstellen Sie mit `vi` eine Shell-Kommandodatei namens `printParam1`. Diese Datei erhält lediglich eine Zeile, in der der erste Parameter der Kommandodatei als Parameter an das Kommando `echo` übergeben wird (wie konnte man nochmal auf die Parameter der Kommandozeile zugreifen?!).
2. Was ergibt Ihrer Meinung nach

   ```
   sh ~/printParam1 otto fred hugo
   ```

1. Überprüfen Sie es. Was ergibt Ihrer Meinung nach

```
cd /bin; sh ~/printParam1 *
```

Prüfen Sie es ebenfalls. Was leiten Sie daraus ab (wer interpretiert also den Stern)?

Aufgabe 23:
Kopieren Sie mit dem `cat`-Kommando `printParam1` in `printParam1_2`. Wie geht das?

Modifizieren Sie mit `vi` die Datei `printParam1_2`, sodaß exakt die beiden ersten Parameter ausgegeben werden. Rufen Sie das neue Programm mit zwei, drei, einem und null Parametern auf.

Was passiert? Wie sind nicht belegte Parameter für eine Kommandodatei vorbelegt?

(Sollten Sie das dringende Bedürfnis haben, irgendwo einen Kommentar anzugeben, schreiben Sie ein # an den Anfang einer Zeile in einem Shellskript.)

Aufgabe 24:
Schreiben Sie eine Shell-Kommandodatei `perm312`, die zuerst den dritten Parameter ausgibt, dann den ersten, dann den zweiten, und dann die restlichen in der ursprünglichen Reihenfolge (jeweils in einer Zeile!). Es genügt vorerst, wenn das Programm für maximal neun Parameter funktioniert.

Was passiert, wenn Sie für `perm312` zehn Parameter angeben?

Zur Korrektur von `perm312` benötigen wir noch Konzepte: Kontrollstrukturen der Shell und Parameterumnumerierung

Aufgabe 25:
Das `shift`-Kommando numeriert Parameter einer Shelldatei um. Aus dem Inhalt von `$2` wird der Inhalt von `$1`, aus dem Inhalt von `$3` wird der Inhalt von `$2` usw. Der zehnte Parameter wird jetzt über `$9` angesprochen.

`shift` kann auch mehrfach ausgeführt werden. Nach j Aufrufen von `shift` in einer Shelldatei steht der i+j-te Kommandozeilenparameter in $i, mit $1 \leq i \leq 9$.

Kopieren Sie `perm312` nach `perm312_423` und modifizieren Sie die neue Datei so, daß sowohl die ursprüngliche Ausgabe entsteht und in weiteren Zeilen die Parameter mit Position 4, 2, 3, 5, 6, 7, 8, 9 und 10 ausgegeben werden.

Testen Sie auch, was anfangs in $0 steht. Wozu könnte das gut sein?

Aufgabe 26:
Man kann in Shelldateien Kontrollstrukturen verwenden. Eine Wiederholungsschleife sieht wie folgt aus:

```
while Kommandofolge1
do Kommandofolge2
done
```

Die Syntax ist streng, d.h. Schlüsselworte müssen in der Zeile vor anderen Dingen stehen, die Kommandofolgen können auch in eigenen Zeilen stehen. Kommandofolgen sind durch Strich-

punkte getrennte Kommandos. Auch eine Bedingung wird also durch ein Kommandofolge ausgedrückt. “Wahr” bedeutet, daß das letzte Kommando in `Kommandofolge1` erfolgreich war (liefert einen positiven Rückgabewert), “falsch” bedeutet, daß das letzte Kommando gescheitert ist.

In der Regel möchte man keine Kommandofolge als Bedingung angeben, sondern eine richtige Bedingung. UNIX-Trick: es gibt ein Kommando `test`, das Bedingungen als Parameter hat, sie auswertet und entsprechend erfolgreich ist oder scheitert. Beispielsweise liefert `test $1 = "x"` “wahr”, wenn $1 nach Expansion gleich x ist und “falsch” sonst (Achtung: Leerzeichen sind signifikant, `test` hat drei Parameter...).

Schreiben Sie mit diesem Wissen eine Skriptdatei `seqx`, die die Kommandozeilenparameter nacheinander durchläuft. Solange ein einzelnes x kommt, wird in einer Zeile o.k. ausgegeben, bei einem anderen Parameter beendet sich das Skript.

Testen Sie Ihr Skript mit

```
sh seqx x x x y
```

und

```
sh seqx abc
```

Was passiert für

```
sh seqx x x
```

Gehen Sie dem Problem nach, indem Sie den Trace-Modus der Shell aktivieren. Durch den Parameter “-x” sehen Sie, welche Kommando die Shell ausführt (also `sh -x seqx x x`). Dies ist eine sogenannte Option; solche Parameter fangen in UNIX mit Minus an (als Konvention) und stehen oft vor anderen Parametern.

Gibt es eine Abhilfe? Denken Sie daran, daß Kommandozeilenparameter als Text einkopiert werden. Tip: In der Shell dürfen Anführungszeichen zur Abgrenzung von Parametern verwendet werden.

Aufgabe 27:
Die Notation mit dem Programm `test` in einer Bedingung ist etwas kryptisch. Die vorliegende Shell erlaubt als Abkürzung eckige Klammern um die Bedingung (wieder mit Leerzeichen abgesetzt), also statt `test $1 = "x"` einfacher `[ $1 = "x" ]`

Korrigieren Sie bitte Ihre Skriptdatei entsprechend.

Aufgabe 28:
Jetzt haben wir alles zusammen, um Aufgabe 6 zu korrigieren. Machen Sie das wie folgt: Geben Sie die ersten drei Kommandozeilenparameter in der gewünschten Reihenfolge aus und iterieren Sie über die restlichen, bis Sie auf einen leeren Parameter stoßen (dann kommen keine mehr...).

Aufgabe 29:
Nachdem ich schon mehrfach kritisiert habe, daß manche Kommandos eine Zeile zuviel ausgeben (z.B. ps), sollen Sie das jetzt beheben. Dazu brauchen wir ein weiteres UNIX-Kommando namens `tail`.

Schreiben Sie mit `vi` eine kleine mehrzeilige Textdatei `hugo` (Inhalt egal) und probieren Sie aus was passiert bei

```
tail +2 <hugo
```

und

```
tail -3 <hugo
```

Haben Sie eine Vermutung, was `tail` macht und was die Optionen bedeuten?

Probieren Sie auch das Gegenstück `head` aus mit denselben Optionen. Klappt das alles?

Verifizieren Sie Ihre Beobachtungen an den Online-Manualen.

Aufgabe 30:
Eine Bedingung in einer Shelldatei sieht wie folgt aus:

```
if Kommandofolge1
then Kommandofolge2
else Kommandofolge2
fi
```

Bezüglich Syntax und Bedeutung gilt das bei den Wiederholungsschleifen Gesagte. Der else-Teil ist optional.

Programmieren Sie eine Shelldatei `myps,` die folgendes leistet: Wenn der erste Parameter nicht "`-t`" ist, wird `ps` aufgerufen und sämtliche Parameter an dieses Programm durchgereicht. Anderenfalls wird `ps` mit den restlichen Parametern aufgerufen, aber die Kopfzeile der Ausgabe dieses Programms weggeworfen.

Aufgabe 31:
Eine Verteilungsanweisung in einer Shelldatei sieht wie folgt aus:

```
case wort in
  muster1) Kommandofolge1 ;;
  muster2) Kommandofolge2 ;;
     ...
  musterN) KommandofolgeN ;;
esac
```

Das Wort (z.B. ein Kommandozeilenparameter) wird nacheinander mit allen Mustern verglichen. Beim ersten passenden Muster wird die entsprechende Kommandofolge ausgeführt. Das letzte Muster ist oft "`*`", weil es auf alles paßt.

Schreiben Sie ein Skript namens `mycase`, das einen optionalen Parameter hat. Wenn er –p ist, wird das Skript perm312 auf die Standardausgabe ausgegeben, wenn er mit –s anfängt, wird der Inhalt von `seqx` ausgegeben, ansonsten wird der Inhalt von `mycase` ausgegeben.

Kurzübersicht vi-Editor

Start mit `vi dateiname`

Editiermodus

(* steht für beliebige Zahl, Standardwert ist 1)

Cursor bewegen

* <Cursortaste>	Cursor um * Zeilen oder Spalten bewegen
*<LEER>	um * Zeichen vorwärts
*w	um * Worte vorwärts
*b	um * Worte zurück
^	an Zeilenanfang
$	an Zeilenende

Löschen

*d<LEER>	* Zeichen löschen
*dw	* Worte löschen
*db	* Worte zurück löschen
*dd	* Zeilen löschen
d$	bis Zeilenende löschen

in Einfügemodus wechseln

i	Text einfügen vor Cursor
a	Text einfügen nach Cursor

Einfügemodus (Ende mit ESC)

Tippen von Buchstaben, Rücktaste und Cursortasten möglich; mit ESC kommt man wieder in Editiermodus

Kommandomodus (immer eingeleitet durch ":")

Beenden

`:wq`	Datei speichern (write and quit)
`:q!`	vi abbrechen (quit, really!)

Suchen

`:/muster/`	Suchen vorwärts nach `muster`
`:?muster?`	Suchen rückwärts nach `muster`

Ersetzen

`:s/muster/ersatz/`	einmal Suchen nach `muster`, ersetzen durch `ersatz`
`:s/muster/ersatz/g`	Suchen nach `muster`, ersetzen durch `ersatz` (bis Zeilenende)
`:1,$s/muster/ersatz/`	von erster bis letzter Zeile jeweils einmal Suchen nach `muster`, ersetzen durch `ersatz`
`:1,$s/muster/ersatz/g`	von erster bis letzter Zeile jeweils Suchen nach `muster`, ersetzen durch `ersatz` (bis Zeilenende)

Nach Beenden des Kommandos ist man wieder im Editiermodus.

Übung 3 (6.5.1999)

Melden Sie sich am System an (Benutzername: `dvsp`, kein Paßwort). Kontrollieren Sie, daß ihr Unterverzeichnis immer noch vorhanden ist. Ansonsten erzeugen Sie es mit

```
mkdir Nachname_Vorname
```

(belegen Sie das mit Ihrem Nachnamen und Vornamen entsprechend). Wechseln Sie in ihr persönliches Verzeichnis durch den Befehl `cd Nachname_Vorname`

Aufgabe 32:
Shellvariablen sind Variablen mit Namen und einer Zeichenkette als Inhalt. Ein Name beginnt mit einem Buchstaben und darf zusätzlich Ziffern oder Unterstreichungszeichen enthalten.

Eine Wertdefinition findet durch Zuweisung statt, beispielsweise belegt

```
benutzer=hase
```

die Variable `benutzer` mit dem Wert `hase`.

Benutzt wird eine Variable durch Voranstellung von $ und Angabe des Namens in geschweiften Klammern. Also würde `echo ${benutzer}` als Ergebnis `hase` ausgeben. Anwendung von Shellvariablen sind Expansionen und textuelles Einkopieren.

Vordefinierte Variablen sind beispielsweise:

@	alle Parameter der Kommandozeile
#	Anzahl der Parameter in der Kommandozeile

Schreiben Sie eine Shelldatei namens copy, die wie folgt funktioniert: `sh copy -h` liefert als Ausgabe "Benutzung: copy datei1 datei2", `sh copy dateiname` liefert eine Fehlermeldung und die Benutzungsinformation und `sh copy datei1 datei2` kopiert die erste Datei in die zweite.

Tip: Speichern Sie den Text der Benutzungsinformation in einer eigenen Shellvariable und machen Sie zunächst eine Fallunterscheidung nach der Anzahl der Parameter. Bei einem Parameter müssen Sie dann noch prüfen, ob er "-h" ist (ansonsten geben Sie eine Fehlermeldung aus). Bei zwei Parametern kopieren Sie die Dateien.

Aufgabe 33:
Es ist unpraktisch, daß man immer das `sh` beim Aufruf einer Shelldatei angeben muß. Dafür gibt es aber Abhilfe: Nach Erstellen einer Shelldatei kann man sie für die Shell als "ausführbare Datei" kennzeichnen. Das passiert mit dem Befehl `chmod` (change file mode). Man kann sämtliche Zugriffsrechte auf einer oder mehreren Dateien ändern für Eigentümer (user), Gruppe (group) und alle Anderen (others). Rechte sind Lesen (r), Schreiben (w) und Ausführen (x). Beispielsweise macht

```
chmod u+x copy
```

die Datei copy ausführbar für den Eigentümer und

```
chmod o-rwx copy
```

die Datei copy weder les-, schreib- noch ausführbar für Personen außerhalb der Gruppe.

Bei als ausführbar markierten Textdateien gibt es eine Konvention: in deren erster Zeile steht eine spezieller Kommentar, der das Programm angibt, für das die Datei gedacht ist. Bei Shelldateien ist das die Shell (mit komplettem Namen `/bin/sh`). Man schreibt als Kommentar in eine Shelldatei

```
#!/bin/sh
```

und stellt damit klar, daß es sich um eine Shelldatei handelt.

Kontrollieren Sie die Konvention bei `/usr/bin/filesize` und bei `/usr/bin/c2ph` (z.B. mit `more`).

Korrigieren Sie `copy` entsprechend und übernehmen Sie das Vorgehen für zukünftige Skriptdateien.

Aufgabe 34:
Schreiben Sie eine Shell-Kommandodatei `mydir`, die maximal zwei Parameter hat: es kann eine Option vorkommen (durch "-" eingeleitet) und auch der Name eines Verzeichnisses (auch beides oder keines).

`mydir` erzeugt in jedem Fall den Inhalt des Verzeichnisses in Langform (wie bei `ls -l`), aber ohne die Zeile "total XXX". Die Option bestimmt, wie die Dateien sortiert sind: "-s" bedeutet aufsteigend nach Größe und "-t" aufsteigend nach Uhrzeit.

```
mydir
```

gibt alle Dateien im aktuellen Verzeichnis in Langform alphabetisch sortiert aus,

```
mydir -t
```

gibt alle Dateien im aktuellen Verzeichnis in Langform sortiert nach Uhrzeit aus und

```
mydir -s /bin
```

gibt alle Dateien im Verzeichnis /bin in Langform sortiert nach Größe aus.

Vorschlag zum Vorgehen: Finden Sie in der Shelldatei zunächst heraus, welche Kommandozeilenparameter angegeben wurden und merken Sie diese in Shellvariablen OPTION und DIRECTORY (diese können auch leer sein!). Speichern Sie dann die mit `tail` um die überflüssige Zeile entsprechend gekürzte Ausgabe von `ls -l` in einer Hilfsdatei. Je nach Option müssen Sie diese Datei noch mit dem Kommando `sort` sortieren. Die Spalte, nach der sortiert wird, können Sie als Parameter bei `sort` angeben (lesen Sie das in der Manualseite nach! Tip: Option "+").

Aufgabe 35:
Probieren Sie aus, was das Kommando

```
tr abc 321
```

mit einer Eingabe macht (dabei sollten die Buchstaben "a", "b" und "c" vorkommen). tr steht übrigens für "translate".

Was passiert bei

```
tr f-j M-Q
```

bei

```
tr -s 0-9 %
```

bei

```
tr -c -s A-Za-z !
```

und bei

```
tr -c -s A-Za-z \\n
```

(\\n steht in der Shell für einen Zeilenvorschub.). Beschreiben Sie die Bedeutung der Parameter von tr, und die Bedeutung der Optionen -s beziehungsweise -c.

Aufgabe 36:
Basierend auf diesem Wissen schreiben Sie eine Pipeline, die Ihnen die Anzahl der Zahlen in einem Text (nicht die Anzahl der Ziffern!) ausgibt. Eine Zahl ist eine ununterbrochene Folge der Ziffern 0 bis 9.

Aufgabe 37:
Schreiben Sie (mit vi) einen sinnvollen deutschen oder englischen Text (ohne deutsche Sonderzeichen und möglichst ohne Rechtschreibfehler) in eine Datei `meinText`. Orientierung: circa 20 bis 50 Wörter.

Was passiert, wenn Sie folgende Pipeline starten?

```
cat meinText | tr A-Z a-z | tr -c -s a-z \\n | sort | uniq
```

Erklären Sie das anhand der Bedeutung der einzelnen Kommandos. Schreiben Sie eine Shelldatei `macheWortliste` mit einem Parameter, die obige Pipeline für eine beliebige Datei ausführt.

Speichern Sie das Ergebnis von

```
macheWortliste meinText
```

in der Datei `wortliste`.

Bauen Sie in `meinText` ein paar Rechtschreibfehler ein und ein paar exotische Wörter. Erzeugen Sie für diese Datei mit dem Skript `macheWortliste` eine neue Wortliste `neueListe`. Sehen Sie sich die Manualseite von `comm` an und überlegen Sie, wie Sie die fehlerhaften oder verdächtigen Worte von `meinText` auf Basis der Datei `wortliste` herausfinden können.

Wenn Sie das für beliebige deutsche Texte durchführen wollen, was sollte dann in `wortliste` stehen? Wie müssen Sie die Shelldatei `macheWortliste` anpassen?

Erste Liste von nützlichen UNIX-Kommandos

cat datei ...	kopiert alle angegebenen Dateien auf die Standardausgabe
cd verzeichnis	wechselt in vorgegebenes relatives oder absolutes Verzeichnis
chmod ugo+rwx datei ...	ändert Zugriffsrechte auf Dateien (+ für Hinzufügen, - für Wegnehmen)
echo arg ...	schreibt die Parameter auf der Kommandozeile in die Standardausgabe
head -n	gibt die ersten n Zeilen einer Datei aus
ls verzeichnis	listet Dateien im Verzeichnis (Standardwert: .)
man befehl	zeigt Handbuchseite für befehl an (sofern vorhanden)
mkdir verzeichnis	legt neues Verzeichnis an
more datei ...	erlaubt Sichtung von Dateien mit Blätterfunktion

sleep zeit	legt Prozeß für gegebene Zeit schlafen
sort	alphabetisches Sortieren der Standardeingabe in die Standardausgabe
tail -n	gibt die letzten n Zeilen einer Datei aus (+n: ab Zeile n)
tr string1 string2	ersetzt Buchstaben aus string1 durch entsprechende in string2
uniq	unterdrückt benachbarte Duplikatzeilen aus der Standardeingabe
vi datei	ermöglicht Editieren von Dateien
wc	zählt Zeilen, Wörter und Zeichen der Standardeingabe

Übung 4 (13.5.1999)

Melden Sie sich am System an (Benutzername: `dvsp`, kein Paßwort). Kontrollieren Sie, daß ihr Unterverzeichnis immer noch vorhanden ist. Ansonsten erzeugen Sie es und wechseln Sie dorthin.

Aufgabe 38:
Man kann die Standardausgabe eines Kommandos direkt als Text in der Shell verwenden, wenn man dieses Kommando in Rückwärtsapostrophe einbettet. `echo `pwd`` führt zunächst pwd aus und übergibt das aktuelle Verzeichnis als einzigen Parameter an echo, der es dann ausgibt. Ebenso legt man mit

```
verzeichnis=`pwd`
```

in der Shellvariable `verzeichnis` den Wert des aktuellen Verzeichnisses ab. Der gesamte String zwischen den Rückwärtsapostrophen wird als Kommando ausgewertet und durch seine Standardausgabe ersetzt.

Erstellen Sie ein Skript `power`, das eine positive ganze Zahl `i` als Parameter hat und 2^i berechnet und ausgibt. Sie können davon ausgehen, daß der einzige Parameter wirklich eine positive ganze Zahl ist.

Tip: Mit dem Kommando `expr` können Sie arithmetische Ausdrücke auswerten (z.B. `expr $a + 1`). Speichern Sie den Parameter in einer Shellvariablen, die Sie in einer Schleife bis auf 0 herunterzählen und multiplizieren Sie sukzessive den Inhalt einer anderen Variablen mit 2. Worauf müssen Sie bei einer Multiplikation mit `expr` achten (bezüglich der Parameter)?

Aufgabe 39:
Auf Übungsblatt 2 haben Sie das Kommando `perm312` erstellt, das die ersten drei Parameter permutiert und die übrigen in derselben Reihenfolge ausgibt. Problem damals war, daß in der allgemeinsten Variante jeder Parameter nur in einer eigenen Zeile ausgegeben werden konnte.

Modifizieren Sie Ihre damalige Lösung (oder erstellen Sie eine neue!), die alle Parameter in einer Zeile ausgibt (und natürlich die ersten drei permutiert hat). Speichern Sie dazu die Zwischenergebnisse in einer Shellvariable. Hinweis: Konkatenation zweier Shellvariablen in eine dritte kann mittels `gerda="$hugo $otto"` erfolgen.

Aufgabe 40:
Eine FOR-Schleife hat folgende Syntax:

```
for name in wort1 wort2 ...
do Kommandofolge
done
```

Die Shellvariable `name` wird nacheinander mit `wort1, wort2` usw. belegt und damit die Kommandofolge durchgeführt.

Häufig iteriert man mit for über das Ergebnis eines Kommandos oder über Parameter der Kommandozeile. In letzterem Fall gibt man diese Parameter mit der Shellvariable @ an, die sinnvollerweise in Anführungszeichen geschrieben werden sollte, also `for i in "$@"`.

Schreiben Sie mit einer for-Schleife ein Skript `checkForBlank`, das prüft, ob in der Kommandozeile ein Parameter vorkommt, der aus einem Leerzeichen besteht. Ausgabe soll sein, daß der i-te Parameter ein Leerzeichen ist.

Wie geben Sie diesen Parameter in der Kommandozeile ein?

Modifizieren Sie Ihr Skript und verwenden Sie statt "$@" die in Shellskripten gängige Variable $*. Funktioniert Ihr Skript noch und wenn nein, warum nicht?

Aufgabe 41:
Um Information in die Shell zeilenweise zu bekommen (meist zur Interaktion mit dem Benutzer) gibt es in der Shell das Kommando read. Es liest eine Zeile von Standardeingabe (also z.B. vom Terminal) und weist einzelne Worte an die Shellvariablen zu die als Parameter angegeben wurden. Die letzte Shellvariable erhält alle Information, die keinem vorhergehenden Parameter zugewiesen wurden.

Wenn die Zeile `read hugo otto karl` als Eingabe

```
Der Fuchs springt auf das Dach.
```

vorfindet, werden implizit folgende Zuweisungen gemacht:

```
hugo="Der", otto="Fuchs", karl="springt auf das Dach."
```

read ist dann nicht erfolgreich, wenn die Standardeingabe erschöpft ist.

Schreiben Sie ein Skript `interactiveWC` mit einem Dateinamen als Parameter, das den Benutzer (mit echo und read) fragt, ob die Zahl der Zeichen oder Zeilen dieser Datei auszugeben ist und dies dann tut (mit wc).

Aufgabe 42:
Generell gilt: Alle Ablaufkonstrukte werden von der Shell als ein Kommando angesehen. Damit können Sie E/A-Umleitung oder Pipelines verwenden. Diese Umleitung gilt dann für alle Kommandos, die innerhalb des Ablaufkonstruktes ausgeführt werden.

Erstellen Sie ein Skript `findUserCode`, das die Datei /etc/passwd analysiert. Diese Datei enthält Information über sämtliche Benutzer. In jeder Zeile steht ein Benutzer: die erste Spalte enthält die Kennung, die zweite das verschlüsselte Paßwort und die dritte die Benutzernummer im System. Ihr Skript hat einen Parameter (die zu suchende Benutzerkennung) und soll die Benutzernummer ausgeben.

Tip: Innerhalb des Skripts müssen Sie die Konvention zur Trennung von Spalten in /etc/passwd (Trennzeichen ist Doppelpunkt) an die Konvention von read anpassen (Trennung von Worten durch Leerzeichen). Benutzen Sie dazu das tr-Kommando. Danach lesen Sie das Ergebnis in einer Schleife zeilenweise mittels read ein und prüfen das erste Wort.

Aufgabe 43:
Die Shell sucht ihre Kommandos in mehreren Verzeichnissen. Diese sind in der Variable PATH abgelegt.

Schreiben Sie eine Shelldatei namens `lookupCommand` mit einem Parameter (dem zu findenden Kommandonamen). Die Datei sucht den Namen in den Verzeichnissen von PATH und gibt den vollständigen Pfadnamen aus (wenn das Kommando irgendwo vorhanden war).

Tip: Speichern Sie PATH in einer lokalen Variable und passen Sie diese mit `tr` so an, daß Sie in einer for-Schleife über deren Inhalt iterieren können (bei jedem Schleifendurchlauf ein Verzeichnis). Ob es eine Datei mit vorgegebenem Namen wirklich gibt, können Sie mit `test` feststellen (Manualseite konsultieren).

Übung 5 (27.5.1999)

Melden Sie sich am System an (Benutzername: `dvsp`, kein Paßwort). Kontrollieren Sie, daß ihr Unterverzeichnis immer noch vorhanden ist. Ansonsten erzeugen Sie es und wechseln Sie dorthin.

Aufgabe 44:

```
/* Implementation des Echo-Kommandos
#include "stdio.h"

int main (int argc, char *argv[])
{
  for (i=0; i<=argc;  i++) {
    printf("%s", argv[i]);
  }
}
```

In `~tensi/dvsp/myecho.c` finden Sie obiges fehlerhaftes Gerüst eines Programms, das die Parameter der Kommandozeile in einer Zeile ausgeben soll. In der Ausgabe soll zwischen den Parametern ein Leerzeichen stehen und nach dem letzten kein (!) Leerzeichen, sondern eine neue Zeile.

Kopieren Sie sich diese Datei in Ihr eigenes Verzeichnis, übersetzen Sie sie mit `gcc myecho.c -o myecho` und korrigieren Sie eventuelle Fehler. Vergleichen Sie visuell die Ausgabe des Kommandos `myecho /bin/*` mit der von `echo /bin/*`

Hinweis: Das Programm benutzt als wesentliche Funktion `printf` aus der `stdio`-Bibliothek Sie können alle gängigen C-Funktionen über das man-Kommando abfragen, indem Sie die Kapitelnummer 3 angeben (also `man 3 printf`). In diesem Kapitel der UNIX-Dokumentaion stehen die C-Funktionen (Kapitel 1 (Standard) enthält die Kommandos, Kapitel 2 die Systemaufrufe und Kapitel 4 die Dateiformate). Finden Sie heraus, wie der erste Parameter (der Formatstring) von `printf` sinnvoll zu benutzen ist.

Aufgabe 45:
Mit `#include "stdio.h"` wird die Schnittstelle der Standard-Ein-/Ausgabebibliothek von UNIX textuell einkopiert. Elementare Funktionen sind **`int`** `getchar()` und **`void`** `putchar(`**`char`**`)`. `getchar` liest ein Zeichen von der Standardeingabe und liefert es als Resultatwert zurück, `putchar` schreibt ein Zeichen auf die Standardausgabe. Wenn die Eingabe erschöpft ist, wird von `getchar` der spezielle int-Wert `EOF` zurückgeliefert, der niemals ein Zeichen repräsentiert (in C gilt: **char** $\subset$ **int**).

Schreiben Sie ein C-Programm `mycat`, das keine Parameter hat und die gesamte Standardeingabe auf die Standardausgabe kopiert. Kopieren Sie mit diesem Programm probeweise eine Datei in eine andere und prüfen Sie, daß beide Dateien gleich lang sind.

Tip: Im Programm lesen Sie einen Buchstaben von der Standardeingabe und merken Sie ihn sich in einer Variablen. Wenn er ungleich EOF ist, geben Sie ihn auf der Standardausgabe aus und wiederholen Sie den Vorgang.

Aufgabe 46:
Modifizieren Sie `mycat` wie folgt: Wenn ein Parameter angegeben ist (egal wie er aussieht), werden alle Zeichen der Eingabe mit Zeichencode > 127 als "#XXX" auf die Standardausgabe geschrieben (wobei XXX für die dreistellige Dezimaldarstellung des Zeichenkodes steht). Bedenken Sie, daß Zeichen in C wie Ganzzahlen behandelt werden können, weil **char** ⊂ **int**. Beispielsweise hat 'ü' den Code 252.

Testen Sie das neue Kommando durch direkte Eingaben am Terminal.

Aufgabe 47:
Bisher wurde nur von Standardeingabe gelesen und auf Standardausgabe geschrieben. Man kann beliebige Dateien mit einem Dateizeiger (file pointer) assoziieren und auf ihnen arbeiten. Deklaration einer Dateizeigervariablen erfolgt beispielsweise mit `FILE *fp`. Die Zuordnung einer Datei zum Dateizeiger passiert durch Aufruf der Funktion `FILE *fopen(name, modus)`, z.B: durch

```
fp = fopen("hugo", "r");
```

Der Name ist ein Pfadname der Datei als Zeichenkette, der Modus ist "r" für Lesen, "w" für (Neu-)Schreiben und "a" für anhängend-Schreiben. Wenn eine Datei für Schreiben nicht vorhanden ist, wird sie erzeugt. Wenn es einen Fehler gibt, dann wird der NULL-Zeiger als Ergebnis von fopen zurückgeliefert. Mit `int getc(FILE *)` wird ein Zeichen von der Datei gelesen (ergibt EOF am Ende der Datei) und mit `int putc(char, FILE *)` geschrieben. Am Ende sollte jede geöffnete Datei mit `fclose(FILE *)` für den entsprechenden Dateizeiger geschlossen werden.

Drei konstante (!) Dateizeiger `stdin`, `stdout` und `stderr` sind beim Start eines Programms bereits vordefiniert.

Schreiben Sie einen einfachen Klon von `wc` namens `mywc`, der wie folgt funktioniert: mywc zählt Zeichen und Zeilen der Eingabe (keine Wörter!). Wenn keine Parameter angegeben sind, wird die Standardeingabe benutzt, ansonsten wird jeder Parameter als Dateiname interpretiert, dessen Inhalt gezählt und auf die Standardausgabe mit Dateinamen geschrieben. Sehen Sie sich die Ausgabe von `wc` bei Angabe von Parametern an; allerdings müssen Sie keine summarische Ausgabe für die Totalwerte bei `mywc` vorsehen.

Aufgabe 48:
Machen Sie das Programm `mywc` robuster, indem Sie prüfen, ob ein in der Kommandozeile angegebener Namen wirklich der einer Datei ist (Rückgabewert von `fopen`!). Geben Sie auf `stderr` mit `fprintf` eine Fehlermeldung aus (man 3 fprintf ansehen...).

Erhöhen Sie die Wartbarkeit Ihres Programms dadurch, daß Sie die Zeichenkonstante für neue Zeile durch einen symbolischen Namen ersetzen, der mit einer #define-Anweisung vor der main-Funktion Ihres Programms eingeführt wird.

Aufgabe 49:
Die stdio-Bibliothek ist eine Abstraktion der einfachen Systemaufrufe auf und ergänzt sie um Dienste wie beispielsweise Pufferung. Die Systemaufrufe arbeiten nicht mit Dateizeigern, sondern mit *Dateideskriptoren*. Das sind kleine ganze Zahlen und Indizes in interne Tabellen. Beispielsweise stehen die Deskriptoren 0, 1 und 2 für die Standardeingabe, -ausgabe und -fehlerausgabe.

Aus Effizienzgründen sollten bei einem Systemaufruf größere Blöcke gelesen oder geschrieben werden (z.B. 128 Zeichen). Diese Information steht in einem Feld von Zeichen. Die Systemaufrufe sind `int read(int, char *, int)` und `int write(int, char *, int)` mit folgender Benutzung

```
bytesRead = read(fileDescriptor, characterArray, numberOfBytes);
bytesWritten=write(fileDescriptor, characterArray, numberOfBytes);
```

read liefert immer die Zahl der gelesenen Zeichen zurück. Dies weicht von der vorgegebenen Zahl dann ab, wenn die Eingabe fast erschöpft ist. Ein Wert kleiner 0 ist bei beiden Funktionen ein Fehler.

Kopieren Sie `mycat.c` aus Aufgabe 2 (!!) in `mycatsimple.c` und modifizieren die Datei so, daß die stdio-Bibliothek nicht mehr benutzt wird, sondern die primitiven Systemaufrufe `read` und `write`. Den Code für die Sonderbehandlung von Zeichen aus Aufgabe 3 dürfen Sie in `mycatsimple.c` weglassen.

Aus Wartbarkeitsgründen sollten Sie drei symbolische Konstanten (mittels #define) einführen: die Anzahl der zu lesenden Bytes (und damit die Länge des Zeichenfelds), `stdinFD` für den Dateideskriptor für die Standardeingabe und `stdoutFD` für den Dateideskriptor für die Standardausgabe.

Aufgabe 50:
Für die Dateibehandlung gibt es als Systemaufrufe `int open(int, int)`, `int creat(int, int)` und `void close(int)`, die wie folgt benutzt werden:

```
fileDescriptor = open(fileName, rwMode);
fileDescriptor = creat(fileName, protectionMode);
close(fileDescriptor);
```

open öffnet eine existierende Datei (als Modus steht 0 für Lesen und 1 für Schreiben), creat legt eine neue Datei zum Schreiben an mit entsprechenden Zugriffsrechten (z.B. 0755 für Lesen und Ausführen für alle und Schreiben für den Eigentümer). close schließt die dem Dateideskriptor zugeordnete Datei.

Erstellen Sie ein C-Programm, das genau einen Parameter hat. Dieser Parameter gibt eine Datei an, deren Tabs durch jeweils ein Leerzeichen ersetzt werden sollen. Benutzen Sie nur Systemaufrufe und keine Funktionen aus der stdio-Bibliothek. Wenn die Datei nicht existiert, passiert nichts.

Hinweis: Verwenden Sie wiederum einen symbolischen Namen für die Anzahl der zu lesenden Bytes.

Übung 6 (10.6.1999)

Melden Sie sich am System an (Benutzername: `dvsp`, kein Paßwort). Kontrollieren Sie, daß ihr Unterverzeichnis immer noch vorhanden ist. Ansonsten erzeugen Sie es und wechseln Sie dorthin.

Aufgabe 51:
In Aufgabe 49 der 5. Übung haben Sie gesehen, daß die Systemaufrufe für Ein-/Ausgabe bevorzugt große Blöcke einlesen lassen.

Überlegen und begründen Sie, warum es gute Argumente gibt, bei einem Systemaufruf große Informationsmengen zu übertragen!

Viele Anwendungen wollen aber nur ein Zeichen lesen und ein Zeichen schreiben (z.B. via `getchar` und `putchar`). Das sollen Sie jetzt nachbilden.

Kopieren Sie `mycat.c` in `mygetchar.c` und modifizieren Sie dieses Programm. Es soll prinzipiell das Gleiche tun, aber Sie sollen statt `getchar` aus der Stdio-Bibliothek eine eigene Routine **`int`** `mygetchar()` erstellen und verwenden. Diese stützt sich nur auf `read` ab, aber verhält sich genauso wie `getchar` (liefert immer ein Zeichen zurück und schließlich `EOF`, wenn kein Zeichen mehr vorhanden ist). Bei jedem read-Aufruf müssen Sie 100 Zeichen auf einmal lesen (wenn Sie nur eines lesen, ist die Aufgabe trivial). Aus der Stdio-Bibliothek dürfen Sie `putchar` und `EOF` verwenden, aber sonst nichts!

Hinweis zur Lösung: Legen Sie jedesmal die per `read` gelesenen Zeichen in einem Zeichenfeld ab und geben bei jedem Aufruf von `mygetchar` das nächste Zeichen zurück, bis das Zeichenfeld abgearbeitet ist. Dann rufen Sie wieder `read` auf usw. bis die Eingabe leer ist. Um die Information im Zeichenfeld und die Position des nächsten Zeichens zwischen den Aufrufen von `mygetchar` zu erhalten, können Sie globale Variablen oder besser statische Variablen in der Routine verwenden.

Erkenntnis für Sie: Sämtliche Routinen der Stdio-Bibliothek von UNIX sind Abstraktionen der primitiven Systemaufrufe (Sie haben gerade `getchar` nachprogrammiert!). Der Gewinn an Abstraktion wiegt meist die geringe Leistungseinbuße auf.

Aufgabe 52:
Schreiben Sie ein Programm hurra_x.c, das die Standardeingabe zeichenweise liest und jedesmal, wenn ein 'x' kommt "Hurra, ein x!" (und Zeilenvorschub ausgibt). Sie können die stdio-Bibliothek benutzen.

Um die Sache etwas schwerer zu machen, sollen Sie das Programm jetzt modifizieren: Der Suchprozeß wird in zwei Prozesse aufgespalten. Jeder Prozeß liest ein Zeichen, begutachtet es und schläft dann für eine Zehntelsekunde (Sie sollte erreichen, daß beide Prozesse immer abwechselnd lesen). Die Ausgabe wird etwas variiert: Prozeß 1 schreibt "Hurra, ein x bei 1!" und Prozeß 2 schreibt Analoges.

Tip: Benutzen Sie zum Schlafenlegen den Systemaufruf `usleep`, der als Parameter eine Pausenzeit in Mikrosekunden erwartet (man 3 usleep). Zum Aufspalten verwenden Sie `fork`.

Was schließen Sie daraus für die Dateibehandlung in voneinander via fork abgeleiteten Prozessen?

Aufgabe 53:
Das Kopieren von Eingabe nach Ausgabe in `mycat.c` könnte schneller gehen, wenn zwei Prozesse (via `fork` gebildet) parallel arbeiten.Angeregt durch die vorige Aufgabe möchten Sie also `mycat.c` aus Aufgabe 46/Übung 5 nochmals modifizieren (ein letztes Mal!!).

Testen Sie das neue Kommando auf einer größeren Textdatei.

Überlegen Sie genau: Kann es passieren, daß jetzt die Ausgabe nicht identisch zur Eingabe ist? Tip: Warum kann es sein, daß ein Prozeß den anderen überholt?

Aufgabe 54:
Erstellen Sie ein Programm `unsterblich.c`, das für jedes Signal eine Signalbehandlung vorsieht. Dabei wird die Signalnummer ausgegeben und das Programm beendet sich. Wenn kein Signal kommt, schläft das Programm 10 Sekunden lang.

Tip: Prüfen Sie in der Datei `/usr/include/signum.h` nach, in welchem Bereich die Signalnummern liegen. Installieren Sie für jedes Signal dieselbe Signalbehandlungsroutine (über eine Schleife). Diese Routine hat einen int-Parameter: die Nummer des Signals.

Testen Sie das Programm mit Eingaben wie ^C (Interrupt), ^Z (Programmstop), ^\(Programmende) und vergleichen Sie die Ausgabe mit Information aus `/usr/include/signum.h`.

Aufgabe 55:
`unsterblich.c` soll jetzt wirklich unsterblich werden, d.h. alle Signale ignorieren. Wie müssen Sie es ändern?

Testen Sie das Programm wiederum mit Eingaben wie ^C (Interrupt), ^Z (Programmstop), ^\ (Programmende).

Verlängern Sie die Schlafenszeit auf 150s und starten das Programm im Hintergrund (mit &). Loggen Sie sich sofort aus und wieder ein und sehen Sie sich alle Ihre Prozesse mit

```
ps ux
```

an. Was fällt Ihnen auf? Erklärung?

Haben Sie - aufgrund der Vorlesung - eine Idee, wie man `unsterblich` beendet (Tip: Tabelle der Signale und Standardbehandlungen dafür ansehen!)?

Übung 7 (24.6.1999)

Melden Sie sich am System an (Benutzername: `dvsp`, kein Paßwort). Kontrollieren Sie, daß ihr Unterverzeichnis immer noch vorhanden ist. Ansonsten erzeugen Sie es und wechseln Sie dorthin.

Aufgabe 56:
Der Systemaufruf **`int`** `alarm(`**`int`** `t)` schaltet einen internen Zähler scharf, sodaß er nach t Sekunden ein Signal `SIGALRM` auslöst. Wenn t=0 ist, wird dieser Zähler abgeschaltet.

Schreiben Sie ein kleines Programm `alarmtest.c`, das 10 Sekunden lang aufsteigend Zahlen auf dem Bildschirm ausgibt und dann abbricht.

Aufgabe 57:
Ein Alarm unterbricht sogenannte *langsame Systemaufrufe*, d.h. Systemaufrufe, die blockieren können. Dazu gehört beispielsweise die Eingabe am Terminal.

Modifizieren Sie das Programm `alarmtest.c` folgendermaßen: Es liest via read von Standardeingabe ein. Wenn das nicht innerhalb von 10s erfolgt ist, wird diese Eingabe und das Programm abgebrochen. Ansonsten wird die eingelesene Zeile auf die Standardausgabe ausgegeben.

Denken Sie daran, daß Sie den Zeitalarm abschalten sollten, wenn `read` erfolgreich zurückkehrt, weil er sonst im restlichen Programm zuschlagen könnte.

Leider können wir nicht nur die Leseoperation abbrechen, weil LINUX nach dem Alarm versucht, den durch das Signal unterbrochenen Systemaufruf `read` wieder neu aufzusetzen. Um das Lesen abzubrechen, brauchen wir Funktionen aus der folgenden Aufgabe.

Aufgabe 58:
In C gibt es (Gottseidank!) keine Möglichkeit via **`goto`** eine Funktion zu verlassen. Manchmal braucht man das aber. Beispielsweise ist es praktisch in einem Editor zu sagen, "das aktuelle Kommando ist komplett schiefgelaufen" (ich merke es aber in einer tief geschachtelten Funktion) "bitte gehe zu einem Punkt in der Hauptkommandoschleife des Editors zurück".

Zur Simulation von nichtlokalen Sprüngen gibt es in C und C++ die Funktionen `int setjmp(jmp_buf)` und `void longjmp(jmp_buf, int)`. `setjmp` merkt sich eine Stelle im Programm (und den Kellerpegel sowie Prozessorregister) in einer Variable vom Typ `jmp_buf`, `longjmp` springt an die durch die Sprungpuffervariable beschriebene Stelle zurück. Um zu unterscheiden, ob `setjmp` definitorisch verwendet oder über `longjmp` angesprungen wurde, wird der Rückgabewert benutzt. Bei Definition liefert `setjmp` 0 zurück, bei Ansprung den 2. Parameter der `longjmp`-Funktion. Sie werden wie folgt benutzt:

```
#include <setjmp.h>
jmp_buf sprungpuffer;

int tiefVerschachtelteFunktion()
{
  ...
  /* Fehlersituation */
  longjmp(sprungpuffer, 1);
}
```

```
int main()
{
  ...
  if (setjmp(sprungpuffer)==0)
    /* Definition */;
  else
    /* Ansprung */;
}
```

Modifizieren Sie mit `setjmp` und `longjmp` die Lösung aus Aufgabe 2, sodaß die Eingabe unterbrochen wird, aber das Programm regulär weitermacht, d.h. beispielsweise eine Fehlermeldung ausgibt. Es ist dabei nicht zulässig, daß sie in der Signalbehandlung irgendwelche Ausgaben machen!

Aufgabe 59:
Pipes sind ein sehr einfacher Mechanismus zur Prozeßkommunikation unter UNIX. Pipes sind Kanäle, wobei sowohl das Lese- als auch das Schreibende über einen Dateideskriptor im Prozeß verfügbar ist.Eine Pipe wird durch den Systemaufruf `int pipe(int fileDesc[2])` erzeugt. `pipe` liefert zurück, ob eine Pipe erzeugt werden konnte (<0 heißt: erfolglos) und gibt zwei Dateideskriptoren in `fileDesc` zurück: `fileDesc[0]` enthält das Lese-Ende der Pipe, `fileDesc[1]` das Schreib-Ende.

Pipes können nur zwischen Prozessen benutzt werden mit gemeinsamen Vorfahrprozeß (alle abgeleiteten Prozesse haben ja Zugriff auf dieselben Dateideskriptoren!). Schreiben Sie ein Programm, das sich in zwei Prozesse aufspaltet und bei dem der Vaterprozeß dem Kindprozeß über eine Pipe mitteilt, ob dessen Prozeßnummer gerade oder ungerade ist. Das Kind gibt diese Information via write auf der Standardausgabe aus. Sie dürfen übrigens im Kind nicht die Funktion getpid aufrufen!!

Aufgabe 60:
Für bidirektionale Kommunikation benötigen Sie zwei unterschiedliche Pipes, da eine Pipe immer nur in einer Richtung verwendet werden kann. Schreiben Sie ein Programm `buchstabenraten.c`, in dem der Vaterprozeß einen Buchstaben einliest und der Kindprozeß versucht, diesen Buchstaben zu raten. Der Vater sagt dem Kind, ob der Buchstabe paßt oder ob er zu groß oder zu klein ist (bezüglich des Interncodes). Jeder Rateversuch wird vom Vater protokolliert. Sobald der Buchstabe geraten wurde, beendet sich das Programm.

Für die Prozeßkommunikation benutzen Sie bitte zwei Pipes. Der Kindprozeß muß nicht sehr intelligent raten (obwohl sich binäre Suche anbieten würde...), sondern Schwerpunkt dieser Aufgabe liegt auf der Kommunikation!

Hinweise zur Realisierung:

- Es ist guter Stil, das Ende einer Pipe in einem Prozeß mittels `close` zu schließen, das von ihm nicht benutzt wird.
- Schreiben Sie für die Aufgabe in jedem Prozeß jeweils eine Funktion (z.B. `rateBuchstabe` und `bewerteVersuche`). Beide haben die jeweilig benutzten Enden der Pipe als Parameter und sie verwenden sie zur Eingabe und Ausgabe. Jeder Prozeß benutzt natürlich nur genau eine der beiden Funktionen. Trotzdem wird Ihr Programm etwas übersichtlicher.

Übung 8 (1.7.1999)

Melden Sie sich am System an (Benutzername: `dvsp`, kein Paßwort). Kontrollieren Sie, daß ihr Unterverzeichnis immer noch vorhanden ist. Ansonsten erzeugen Sie es und wechseln Sie dorthin.

Aufgabe 61:
Sie haben aus der STDIO-Bibliothek unter anderem bereits die Funktionen `printf/fprintf` kennengelernt. Wesentliche Idee war, daß es einen Formatparameter gab (eine Zeichenkette), die die Ausgabe der folgenden Parameter gesteuert hat. Dieser Formatstring enthält Angaben wie `%s`, `%d` usw., die sagen, daß der korrespondierende Parameter als Ganzzahl oder als Zeichenkette ausgegeben werden soll.

Analog gibt es für die Eingabe eine mächtige Funktion namens `scanf bzw. fscanf` mit Signatur `int fscanf(FILE *fp, const char* format, ...)`. Beispielsweise liest `n=fscanf(stdin, "%d %s", &i, &s)` eine Ganzzahl und einen String ein. Der Rückgabewert gibt die Anzahl der korrekt eingelesenen Werte an.

Schreiben Sie ein C-Programm `stueckliste.c`, das eine Datei verarbeitet. Die Datei enthält in jeder Zeile eine ganzzahlige Stückzahl c_i, ein Leerzeichen, einen reellwertigen Stückpreis p_i und einen Artikelnamen als Zeichenkette.

Lesen Sie die Datei zeilenweise ein und geben Sie die Informationen auf der Standardausgabe aus, ergänzt um den Postenpreis $c_i p_i$. Am Schluß geben Sie den Gesamtpreis $\Sigma c_i p_i$ aus.

Wenn Ihr System nicht deutsch konfiguriert ist, müssen Sie bei den Preisen statt Dezimalkomma einen Dezimalpunkt eingeben.

Aufgabe 62:
In C und C++ gibt es die Konvention, daß Zeichenketten mit dem Zeichen '\0' (mit Interncode 0) abgeschlossen werden. Leider ist es nicht ohne weiteres möglich, eine Zeichenkette an ein Zeichenfeld zuzuweisen oder die Länge eines 0-terminierten Zeichenfelds herauszufinden.

Dazu dienen die Funktionen aus der STRING-Bibliothek (`string.h`). Beispielsweise bestimmt `int strlen(const char* st)`, wieviele Zeichen in `st` vorkommen,
`void strcpy(char *dest, const char *src)` kopiert die Zeichenkette `src` in `dest`.

Implementieren Sie Ihre eigene Version von `strcpy` und testen Sie sie anhand des folgenden Programms:

```
#include <stdio.h>
... Ihre Implementierung von strcpy ...
int main(int argc, char* argv[])
{
  char buffer[200];
  strcpy(buffer, argv[1]);
  printf("Parameter 1: '%s', Kopie '%s'\n", argv[1], buffer);
}
```

Aufgabe 63:
Modifizieren Sie Ihr Programm `stueckliste.c` aus Aufgabe 1. Öffnen Sie eine Pipe mit `popen` (`FILE *popen(const char* command, mode_t mode)`) zum wordcount-Programm. Geben Sie alle Eingabezeilen auch an "wc -l" weiter und schließen Sie am Ende der Eingabe die Pipe mit `pclose`. wc sollte Ihnen die Zahl der Eingabezeilen ausgeben.