1. Übung:

Aufgabe 7: Zeigen Sie den Pfad Ihres HOME-Verzeichnisses an. Anschließend zeigen Sie die Verzeichniseinträge aller Dateien (auch versteckter Punkt-Einträge) im HOME-Verzeichnis an.

- HOME-Verzeichnis: `echo $HOME` oder cd home (cd ~) und dann pwd
- Alle Dateien anzeigen: `ls -a`

Aufgabe 8: Zeigen Sie für folgende Fälle die Verzeichniseinträge der Dateien im Verzeichnis /usr/include/sys an, wobei deren Namen

1. ein 'a' enthält;
2. mit 'sa', 'se', 'si', 'so' oder 'su' beginnt;
3. auf 'a.h', 'b.h', 'c.h' oder 'd.h' endet;
4. auf 'a.h', 'b.h', 'c.h' oder 'd.h' endet und an einer beliebigen Stelle ein r enthält.

1. `find -name '*a*' -print`
2. `find -name 's[aeiou]*' -print`
3. `find -name '*[a-d].h' -print`
4. `find -name '*r*[a-d].h' -print`

Aufgabe 9: Listen Sie die Verzeichniseinträge des Verzeichnisses /bin mit dem Befehl "ls" und speichern Sie die Ausgabe in Ihrem HOME-Verzeichnis in einer Datei namens "lsbin".

- `ls > $HOME/lsbin`

Aufgabe 10: Listen Sie die Verzeichnisinformationen aller Dateien und Unterzeichnisse unterhalb von /usr/spool rekursiv (Manual Page von ls nachschauen!) an. Verwenden Sie dabei nacheinander folgende Ausgabeumlenkungen in Ihr HOME-Verzeichnis. Wecher Unterschied ist zwischen der erzeugten Datei out.se und out?

1. Leiten Sie die Standardausgabe in out.s und die Standardfehlerausgabe in out.e.
2. Leiten Sie die Standardausgabe in die Datei out.se und fügen Sie die Standardfehlerausgabe an die Datei out.se an.
3. Leiten Sie beide Standardausgabe und Standardfehlerausgabe in die Datei out.

1. `ls -R > out.s 2> out.e`
2. `ls -R > out.se 2>> out.se`
3. `ls -R > out 2> out`

Unterschied: Reihenfolge der Fehlermeldungen.

2. Übung:

Aufgabe 1: Informieren Sie sich über Prozesse mit dem Befehl "ps" oder "pstree" mit passenden Optionen und/oder mit dem Tool "kpm". Finden Sie dabei folgende Detailinformationen heraus:

1. die Kommandos mit den Namen ausführender Benutzer und den ProzessID's der momentan laufenden Prozesse (Zustand: running);
2. den durch Ihre aktuell benutzte Shell belegten und den maximal erforderlichen Arbeitsspeicher;
3. die Hierarchie der Elternprozesse der akutellen Shell.

1. `ps -ur`
2. `ps -l` (Size = maximal erforderlicher Speicher, Resident Set Size (RSS) = aktuell belegter Speicher
3. Baumansicht in kpm

Aufgabe 2: Erstellen Sie ein Shell-Skript mit dem Namen "finfo", das Informationen über eine Datei in folgender Form (Beispiel) ausgibt:

```
Aufruf: finfo /bin/ls
Ausgabe:
-----------------------------------------------
Datei            : /bin/ls
Rechte           : -rwxr-xr-x
Groesse          : 68524 byte
Besitzer         : root
Gruppe           : root
letzte Aenderung: 09. 09. 02
-----------------------------------------------
```

Das Skript bearbeitet folgende Fehlerfälle:

- Bei einem fehlerhaften Aufruf mit 0 oder mehr als einem Parameter (test) wird eine Fehlermeldung der Form "Aufruf: finfo <Pfad/Datei>" ausgegeben und der Prozess endet mit dem Exit-Code 1.
- Falls <Pfad/Datei> nicht existiert (test -e) wird eine passende Fehlermeldung ausgegeben und der Prozess endet mit dem Exit-Code 2.
- Falls <Pfad/Datei> keine gewöhnliche Datei ist (test -f), wird ebenfalls eine Fehlermeldung ausgegeben und der Prozess endet mit dem Exit-Code 3.

Hinweise: Benutzen Sie den Befehl "ls" mit passenden Optionen (z.B. in der Manual Page suchen), um die gewünschten Grundinformationen zu erhalten. Leiten Sie die Ausgabe dieses "ls"-Befehls in eine temporäre Datei mit ">" um. Lesen Sie diese Datei in die auszugebenden Variablen mit dem Befehl "read" per Eingabeumleitung "<" ein. Geben Sie dann die Variablen mit "echo" aus und löschen Sie die temporäre Datei.

```
#finfo von Patrick Lipinski
#!/bin/bash

#Mehr als ein Parameter?
if test $# -ne 1
then
    echo "Aufruf: finfo <Pfad/Datei> (Genau ein
Parameter)"
    exit 1

#Existiert die Datei?
elif ! test -e $1
then
    echo "Datei existiert nicht"
```

```
    exit 2

#Ist es eine gewöhnliche Datei?
elif ! test -f $1
then
    echo "keine gewöhnliche Datei"
    exit 3
fi

#Informationsverarbeitung und Ausgabe der Informationen,
wenn
#es vorher keinen Fehler gab.
ls -l --time-style="+%d.%m.%y" $1 > finfo_tmp 2>
/dev/null
read rec z1 bes gru gro let_d let_m let_y dat < finfo_tmp
echo "----------------------------------------"
echo "Datei: $dat"
echo "Rechte: $rec"
echo "Groesse: $gro byte"
echo "Besitzer: $bes"
echo "Gruppe: $gru"
echo "letzte Änderung: $let_d $let_m $let_y"
echo "----------------------------------------"
rm finfo_tmp
exit 0
```

3. Übung:

Aufgabe 1: Definieren Sie Alias-Namen, um anstelle der Windows-Befehle "DIR", "COPY", "DEL", "MD", "RD" und "MOVE" die zugehörigen UNIX-Kommandos "ls", "cp", "rm", "mkdir", "rmdir", "mv" mit passenden Optionen auszuführen. Speichern Sie die Definitionen in einer Datei doshabbits und laden Sie die Definitionen mit . doshabbits in Ihre aktive Shell oder schreiben Sie die Befehle in die Datei .bashrc und starten Sie eine neue Shell.

```
alias dir='ls -la'
alias copy=cp
alias del='rm -i'
alias MD=mkdir
alias rd=rmdir
alias move=mv
```

Speicherung der Befehle in der .bashrc und neustarten der Shell oder Speichern in doshabbits und die Datei mit . doshabbits aufrufen.

Aufgabe 2: Steuerung des Zugriffsschutz: (Für diese Aufgabe brauchen Sie mindestens 2 Benutzerkennungen. Arbeiten Sie also in Gruppen zusammen.)

1. Schauen Sie sich die mit "umask" gesetzten Default-Rechte für neu erzeugte Dateien an. Welche Wirkung hat die "umask"-Einstellung ?
2. Definieren Sie die umask ggf. so um, dass innerhalb der Gruppe und anderer Benutzer kein lesender und schreibender Zugriff auf Dateien möglich ist. Erzeugen Sie eine neue Datei (z.B. mit einem Editor oder mit "touch") und probieren Sie den Zugriff auf diese neue Datei durch einen anderen Benutzer aus.
3. Geben Sie den Benutzern innerhalb Ihrer primären Gruppe mit dem Kommando "chmod" das Leserecht auf die neu erzeugte Datei. Probieren Sie erneut den Zugriff durch einen anderen Benutzer aus.
4. Versuchen Sie einem anderen Benutzer diese Datei mit dem Kommando "chown" zu schenken.

1. Voreinstellung der Zugriffsrechte einer neuen Datei. Dabei sind die übergebenen Parameter genau umgekehrt, wie man es erwarten würde: `umask 077` gibt dem User volle Rechte, der Gruppe und den anderen aber nicht. Nur `umask` gibt die aktuelle Einstellung aus.
   Kurze Erläuterung zu der Zahl: Das ist eine Oktalzahl, die binär umgeschrieben die Rechte einer Datei angibt. Eine Datei kann die Rechte rwx rwx rwx haben. Die ersten drei Zeichen stehen für den User, die zweiten drei Zeichen für die Gruppe des Users und die dritten drei Zeichen für alle anderen. 111 111 111 bedeutet also volle Rechte für alle, in oktaler Schreibweise ist das 777.

   umask 000 ➔ rw- rw- rw-
   umask 077 ➔ rw- --- ---
   umask 022 ➔ rw- r-- r--
   umask 777 ➔ --- --- ---

2. `umask 077, touch neue_testdatei`
3. `chmod g+r neue_testdatei`
   Der Gruppe wird das Recht r(ead) gegeben.
4. `chown` kann nur der root ausführen. `chown` steht für „Change Owner".

Aufgabe 3: Erstellen Sie ein Shell-Skript, das einen Taschenrechner mit den vier Grundrechenarten +, -, *, / als ganzzahlige Operationen simuliert. Der Rechner nimmt in einer Schleife Ausdrücke der Form 3 + 7 oder 4 * 3 oder 7 / 5 entgegen und berechnet deren ganzzahlige Ergebnisse (Befehl let). Ein ungültiger Operator, z.B. bei 7 ? 5, wird mit einer Fehlermeldung quittiert und der Rechner wird verlassen. Bei Eingabe der Zeichenkette "ende", wird die Schleife und das Skript beendet.

Hinweis: Benutzen Sie eine while-Schleife und ein case-Anweisung. Achten Sie darauf, dass * ein Spezialzeichen in der case-Anweisung darstellt. Die Sonderbedeutung von * kann man durch Maskierung * aufheben.

```
#!/bin/bash

# solange die Eingabe nicht ende ist, weitermachen.
while [ "$a" != "ende" ] ;
do
     echo "Bitte Kommando eingeben: "
     # b ist das Rechenzeichen
     read a b c

     case "$b" in
     + )       (( "result=a + c" ))
               echo "$a + $c = $result"
               continue
               ;;

     - )  (( "result=a - c" ))
               echo "$a - $c = $result"
               continue
               ;;

     # * gilt eigentlich als Ersatzzeichen für alle anderen
     # Zeichen, so dass es hier mit dem \ maskiert werden muss.
     \* )      (( "result=a * c" ))
               echo "$a * $c = $result"
               continue
               ;;

     / )  (( "result=a/c" ))
               echo "$a / $c = $result"
               continue
               ;;

     "" ) if [ "$a" = "ende" ]
               then
                    exit 0
               fi
               ;;
     * )       echo "falsche Eingabe"
               exit 1
               ;;
     esac
done
```

Alternative Möglichkeit mit „let“:

```
#!/bin/bash
```

```
clear

while [ "$a" != "ende" ] ;
do

echo -n "Bitte Term eingeben: "
read a x b

case "$x" in
+ )   let "erg=a + b"
      echo "$a + $b = $erg"
      continue
      ;;

- )   let "erg=a - b"
      echo "$a - $b = $erg"
      continue
      ;;

\* )  let "erg=a * b"
      echo "$a * $b = $erg"
      continue
      ;;

/ )   let "erg=a/b"
      echo "$a / $b = $erg"
      continue
      ;;

"" )  if [ "$a" = "ende" ]
      then
      continue
      fi
      ;;
* )   echo "falsche Eingabe"
      exit 1
      ;;

esac
done
echo "Taschenrechner beendet"
clear
exit 0
```

Übung 4:

Aufgabe 1: Erstellen Sie ein Skript "sdel" zum sicheren Löschen einer Datei, d.h. nach versehentlichem Löschen kann die Datei wiederhergestellt werden. Verwenden Sie dazu in "sdel" den "mv"-Befehl, um die Datei beim Aufruf "sdel Pfad/Datei" in ein Verzeichnis "$HOME/.papierkorb" zu verlagern. Das Papierkorb-Verzeichnis soll von "sdel" automatisch angelegt werden, falls es noch nicht existiert. Datei wird dabei in folgenden Fällen nicht gelöscht:

- Datei ist keine reguläre Datei.
- Verzeichnis von Datei ist nicht schreibbar.
- Datei ist für den Aufrufer des Skripts nicht schreibbar.
- Datei ist für den Aufrufer des Skripts nicht lesbar.
- Eine Datei gleichen Namens existiert schon im Papierkorb-Verzeichnis.

```
#!/bin/bash
clear

#Papierkorb schon vorhanden?
if ! test -e ~/papierkorb
then
     mkdir ~/papierkorb
     #alternativ auch moeglich ist $HOME/.papierkorb
fi

#Datei zum Loeschen vorhanden?
if ! test -e $1
     then echo "Es existiert keine Datei"
     exit 0
fi

#Ordner?
if ! test -f $1
then echo "Diese Datei ist keine ASCII Datei"
     exit 1
fi

#Schreibrechte?
if ! test -w $1
     then echo "Diese Datei ist nicht schreibbar"
     exit 3
fi

#Leserechte?
```

```
if ! test -r $1
     then echo "Diese Datei ist nicht lesbar"
     exit 4
fi

#liefert Verzeichnisstruktur und speichert in temp-Datei
dirname $1 > temp1

#liefert Dateinamen und speichert in temp-Datei
basename $1 > temp2

read verzeichnis < temp1
read datei < temp2

#Abfrage, ob gleichnamige Datei schon im Papierkorb vorhanden
if test -e ~/papierkorb/$datei
     then echo "Diese Datei besteht bereits in Papierkorb"
     exit 5
fi

#Datei wird in den Papierkorb verschoben
mv $1 ~/papierkorb/$datei
rm temp1; rm temp2

echo "Datei $1 wurde in den Papierkorb verschoben!"

exit 0
```

Aufgabe 2: Das Skript undel soll eine Datei ohne Pfadangabe aus dem Papierkorb im aktuellen Verzeichnis wiederherstellen. Existiert dort eine Datei gleichen Namens, so muss der Benutzer das Überschreiben quittieren.

Hinweis zu Aufgaben 1 und 2: Um das Verzeichnis und den Namen der übergebenen Datei mit Pfad zu trennen, können Sie die Befehle dirname und basename verwenden:

Der Aufruf:

```
dirname /home/meier/datei1
```

liefert als Ausgabe:

```
/home/meier;
```

Der Aufruf:

```
basename /home/meier/datei1
```

liefert als Ausgabe:

```
datei1
```

```
#!/bin/bash
clear

# Ueberprueft, ob Datei im Papierkorb ueberhaupt vorhanden
ist
if ! test -e ~/papierkorb/$1
then
     echo "Datei besteht nicht!"
     exit 1
fi

# -i fragt nach, ob Datei ueberschrieben werden soll
# -v zeigt an, welche Aktion durchgefuehrt wurde
#mv -iv ~/papierkorb/$1 ~/
mv -iv ~/papierkorb/$1 ./
exit 0
```

Übung 5:

Aufgabe 1: Um einen Hintergrundprozess (Dämon) für das Sichern von Dateien zu realisieren, soll ein Skript "bkdaemon" entworfen werden. Dies soll wie folgt arbeiten: Augerufen wird das Skript mit dem Verzeichnis, in dem die zu sichernden Dateien stehen "bkdaemon Verzeichnis". Die Dateien werden in dem angegebenen Verzeichnis automatisch z.B. alle 120 Sekunden (Befehl "sleep") in Kopien gesichert. Hierbei sollen die Sicherungskopien eine Erweiterung, z.B. .BAK tragen. Folgende Randbedingungen sind zu beachten:

- Wenn das Verzeichnis nicht existiert, gibt es eine Fehlermeldung.
- Wenn das Verzeichnis weitere Unterverzeichnisse enthält, so werden diese nicht berücksichtigt.
- Wenn die Sicherung schon existiert, d.h. die Datei nicht verändert wurde, wird keine neue Sicherung erstellt (Befehle "cmp" und "continue").
- Berücksichtigen Sie, dass keine Sicherungen von Sicherungen entstehen, z.B. ist file123.BAK.BAK unerwünscht.

```
#!/bin/bash
#Test auf Existenz des Verzeichnisses
if ! test -e $1
then
     echo "Verzeichnis existiert nicht!"
     exit 1
fi

#Endlosschleife
while true
do
     #finde alle Dateien und speichere ihre Namen in $Datei
     #(Ein Namen pro Schleifendurchlauf)
     for Datei in `find $1 -name "*" -print 2>/dev/null`
          do
          echo "$Datei" > bkdaemon_tmp

          #Wenn es schon ein Backup gibt oder die Datei
selbst
          #ein Backup ist, gehe zur naechsten Datei
          if grep .bak bkdaemon_tmp 2>/dev/null >/dev/null
          || cmp -s $Datei $Datei.bak

          then
               continue
          fi
```

```
            #Existenz wird ueberprueft
            if [ -f $Datei ]
                 then
                 cp $Datei $Datei.bak
            fi
     done
rm bkdaemon_tmp
sleep 5
#Ist die Sleepzeit abgelaufen, so beginnt die Endlosschleife
von vorne.
done
exit 0
```

Aufgabe 2: Erstellen Sie ein Shell-Skript "bstatus" zur Anzeige des Benutzerstatus für alle in /etc/passwd registrierten Benutzer. Dabei soll ausgegeben werden, ob ein Benutzer momentan angemeldet ist und wieviele Prozesse gerade von ihm laufen. Die Ausgabe soll wie folgt aussehen:

```
1 adabas  ist abgemelde 20 Prozesse
2 at Batch jobs daemon ist abgemeldet 59 Prozesse
3 bin bin ist abgemeldet 72 Prozesse
4 daemon Daemon ist abgemeldet 1 Prozesse
5 ftp FTP account ist abgemeldet 0 Prozesse
6 games Games account ist abgemeldet 0 Prozesse
7 kremer karim kremer ist abgemeldet 0 Prozesse
8 lp Printing daemon ist abgemeldet 0 Prozesse
9 mail Mailer daemon ist abgemeldet 0 Prozesse
usw.
```

Hinweise: Leiten Sie die Ausgabe des Befehls "sort /etc/passwd" über eine Pipeline zum Einlesen mit "read" in eine "while"-Schleife. Benutzen Sie den Befehl "ps aux | grep $Kennung | wc -l" zur Ermittlung der Prozessanzahl.

```
#!/bin/bash

#Die Userdatei wird zeilenweise Über eine pipe in eine
while-#Schleife geleitet
sort /etc/passwd | while read Kennung REST
do
    #Nur der Username wird aus der Passwd genommen
    #das set trennt die Variable $Kennung an jedem
    #Doppelpunkt, die einzelnen Elemente stehen dann in
    $1, #$2 usw.
    IFS=:
    set $Kennung

    #Anzahl aller Prozesse: Ausgabe aller Prozesse, suche
    #nach jeder Zeile mit dem aktuellen User und Zählung
    der #Zeilen
```

```
    prozesse=`ps aux|grep $1 2>/dev/null|wc -l`
    ((count += 1))
    #Alternativ let count = count + 1
    #Unterscheidung zwischen an- und abgemeldeten Usern.
    Zu #beachten ist, dass der angemeldete Username nur 8
    #Zeichen lang sein darf.
    if who | grep $1 >/dev/null 2>/dev/null
    then
    echo $count\) $1 ist angemeldet, $prozesse
    Prozess\(e\)
    else
    echo $count\) $1 ist abgemeldet, $prozesse
    Prozess\(e\)
    fi
done
exit 0
```

Übung 6:

Aufgabe 1: Schreiben Sie ein C-Programm,

1. das eine beliebige Umgebungsvariable einliest und deren Inhalt auf dem Bildschirm ausgibt.

```
/* umgebungsvariable.c - Übung 6, Aufgabe 1a) */
/* Ausgabe des Inhalts einer Umgebungsvariablen */

#include <stdio.h>
#include <stdlib.h>

main (int argc, char *argv[], char *envp[]){

    char *inhalt;

    /* *vname ist der Zeiger auf PWD, vname dann
    der Inhalt von PWD*/
    char *vname="PWD";

    /* getenv liefert den Wert der Variable */
    inhalt = getenv(vname);
    if (inhalt != NULL) printf("%s\n", inhalt);
}
```

2. das sämtliche Aufrufparameter und Umgebungsvariablen auf dem Bildschirm ausgibt.

```
/* argumentenliste.c - Übung 6, Aufg. 1b) */
/* Ausgabe aller Aufrufparameter und Umgebungsvariablen
*/

#include <stdio.h>
```

```
main (int argc, char *argv[], char *envp[]){

    int i;
    int x=0;

    /* Schleife die alle Aufrufparameter durchläuft */
    printf("Argumenten-Liste (Anfang)\n");
    for (i=0; i < argc; i++) {
        printf("%i. Aufrufparameter: %s\n", i, argv[i]);
    }
    printf("Argumenten-Liste (Ende)\n\n");

    /* Schleife die alle Umgebungsparameter durchläuft */
    printf("Umgebungsvariablen-Liste (Anfang) \n");
    while(*envp != NULL) {
        printf("%i. Umgebungsvariable: %s\n", x++
    ,*envp++);
    }
    printf("Umgebungsvariablen-Liste (Ende) \n");

}
```

3. das ein beliebiges Shell-Kommando ausführt.

```
/* shell_kommando.c - Übung 6, Aufg. 1c) */
/* Ausführung eines beliebigen Shell-Kommandos */

#include <stdio.h>
#include <stdlib.h>

main (void){

    int rc;
    char command[1024];

    printf("Kommando eingeben: ");

    /* Eingabe einlesen */
    scanf("%s", command);

    /* system(...) für Kommandoausführung */
    rc=system(command);

    if (rc != 0){
        printf("\nKommando fehlgeschlagen! rc = %d\n",
    rc);
    }
}
```

4. das das Arbeitsverzeichnis anzeigt und nach Eingabe eines Zielpfades in ein anderes Verzeichnis verzweigt und dann das Arbeitsverzeichnis erneut anzeigt.

```
/* arbeitsverzeichnis.c - Übung 6, Aufg. 1d) */
/* Wechsel des aktuellen Arbeitsverzeichnis */

#include <stdio.h>
#include <sys/param.h>
#include <unistd.h>

main (void){

    char dir[MAXPATHLEN], subdir[MAXPATHLEN];

    /* getcwd legt eine Kopie von PWD an
    Returncode NULL bedeutet Fehler */
    if (getcwd(dir, MAXPATHLEN) == NULL){
        perror("getcwd error");
        exit(1);
    }
    else{
        printf("Aktuelles Arbeitsverzeichnis: %s\n",
    dir);
    }

    printf("Verzeichnis wechseln: ");

    /* Unterverzeichnis einlesen */
    scanf("%s", subdir);

    /* Verzeichnis wechseln mit "chdir" */
    if(chdir(subdir) == -1){
        perror("Falsches Verzeichnis!!");
    }
    if(getcwd(dir, MAXPATHLEN) == NULL){
        perror("getcwd error");
        exit(1);
    }
    else{
        printf("Aktuelles Arbeitsverzeichnis: %s\n",
    dir);
    }
}
```

Aufgabe 2: Schreiben Sie ein C-Programm, das mit fork() einen Kind-Prozess erzeugt und anschließend endet, während der Kind-Prozess weiterläuft und von init adoptiert wird. Zeigen Sie diese Adoption mit dem Kommando ps.

```
/* waise.c - Übung 6, Aufg.2 */
/* Erzeugung eines Waisen-Prozesses, der
von init adoptiert wird */

#include <stdio.h>

int main(void) {
    int pid;

    /* Beim ersten Aufruf wird nur der erste Zweig
    der if-Bedingung ausgeführt, dabei wird der
    Kindprozess gestartet, der nur den zweiten Teil
    des Zweiges ausführt (Da Returncode von fork im
    Kindprozess = 0)
    Der Elternprozess beendet sich, der Kindprozess
    steckt in der while-Schleife fest und wird zur
    Waise. */

    if ((pid=fork())!=0) {
        printf ("Elternprozess erzeugte PID %d\n", pid);
    }
    else {
        while (1) {
            sleep (4);
            printf("Elternprozess hat PID %d \n",
    getppid());
        }
    }
}
```

Aufgabe 3: Schreiben Sie ein C-Programm, das mit fork() einen Kind-Prozess erzeugt, der sofort endet, während der Eltern-Prozess weiterläuft aber keinen wait()-Aufruf auf das Ende des Kindes macht. Zeigen Sie den entstandene Zombie-Prozess mit ps an und versuchen Sie, ihn oder den Elternprozess mit kill zu beenden.

```
/* zombie.c - Ãœbung 6, Aufg. 3 */
/* Erzeugung eines Zombie-Prozesses */

#include <stdio.h>

int main(void) {
    int pid;

    /* Beim ersten Aufruf wird nur der erste Zweig
    der if-Bedingung ausgeführt, dabei wird der
    Kindprozess gestartet, der nur den zweiten Teil
    des Zweiges ausführt (Da Returncode von fork
    im Kindprozess = 0)
    Der Kindprozess beendet sich, der Elternprozess
    steckt in der while-Schleife fest und fragt nie
```

```
        den exit-code des Kindprozesses ab. Der Kindprozess
        wird zum Zombie in der Prozesstabelle, der sich mit
        kill nicht beenden lässt. */

        if ((pid=fork())!=0) { /*Elternprozess */
            printf ("Kindprozess: PID %d\n", pid);
            while (1) sleep(10);
        }
        else { /*Kindprozess*/
            printf("Elternprozess: PPID %d\n", getppid());
        }
    }
```

Aufgabe 4: Schreiben Sie ein C-Programm, das mit fork() einen Kind-Prozess erzeugt und anschließend auf dessen Ende mit wait() wartet. Hierbei soll der exit-Kode des Kindprozesses vom Eltern-Prozess ausgegeben werden.

```
/* warten.c - Übung 6, Aufg. 4 */
/* Erzeugung eines Elternprozesses, der auf den
Kindprozess wartet */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {

    int i, pid, exit_status;

    if ((pid=fork())!=0) { /*Elternteil*/

        /* Arbeit des Elternteils */
        printf("Elternteil arbeitet...\n");
        sleep(2);
        printf("Elternteil fertig, warten auf
        Kind...\n");

        /* Warten auf Ende des Kindteils */
        if (wait(&exit_status)==pid)
            printf("Elternprozess sagt: Kindprozess mit %d
                        beendet\n",
            WEXITSTATUS(exit_status));
        }
    else { /*Kindprozess*/

        /* Arbeit des Kindteils */
        sleep(1);
        printf("Kindteil arbeitet...\n");
        sleep(3);
        printf("Kindteil fertig!\n");
```

```
            /* Ende des Kindteils */
            exit(0);
        }
    }
```

Aufgabe 5: Schreiben Sie ein C-Programm (ohne fork()), das sich mit einem Aufruf aus der exec-Familie mit einem anderen Programm überlagert. Dabei soll das Programm versuchen bei Eingabe von 0 who, 1 ls, 2 date und von 3 xxx ein Programm, das es nicht gibt, zu starten. Bei 3 soll eine Fehlermeldung Ihres Programms erscheinen.

```
/* exec_menu.c - Übung 6, Aufg. 5 */
/* Einfaches Menü mit exec */

#include <stdio.h>

int main(void) {
    char *kommando[] = {"who", "ls", "date", "xxx"};
    int i;

    printf("0=who, 1=ls, 2=date, 3=xxx\n");
    scanf("%d", &i);

    /* Programm wird bei Falscheingabe verlassen */
    if (i > 3 || i < 0) {
        printf("Falsche Eingabe\n");
        exit(1);
    }

    /* Überlagerung des Prozesses "exec_menu" mit dem
    Shell-kommando */
    execlp(kommando[i], kommando[i], 0);

    /* Falls es den Shell-Befehl nicht gibt, wird der
    Prozess nicht überlagert und die Fehlermeldung kann
    ausgegeben werden*/
    printf("Kommando nicht gefunden.\n");
}
```

Aufgabe 6: Benutzen Sie das Programm der vorigen Aufgabe, um mit fork(), exec() und wait() ein kleine Menü-Shell zu erzeugen, wobei der Elternprozess in einer Schleife immer wieder die Auswahl 0 who, 1 ls, 2 date, 3 xxx und 4 Ende anbietet. Bei einer falschen Eingabe endet die Menü-Shell mit dem Exit-Kode 1. Der Elternprozess gibt jeweils den Exit-Status des Kind-Prozesses aus.

```
/* efw_menu.c - Übung 6, Aufg. 6 */
/* Erzeugung einer kleinen Menü-Shell */
```

```
#include <stdio.h>
#include <sys/wait.h>

int main(void) {
    char *kommando[] = {"who", "ls", "date", "xxx"};
    int i, rc;

    while(1) {
        printf("0=who, 1=ls, 2=date, 3=xxx, 4=Ende\n");
        scanf("%d", &i);

        if (i==4){
            printf("Ende.\n");
            exit(0);
        } else if (i > 4 || i < 0) {
            /* Menü-Shell wird bei Falscheingabe verlassen
            */
            printf("Falsche Eingabe\n");
            exit(1);
        }

        if (fork()==0) { /*Kindprozess */

            /* Kindprozess führt Kommando aus */

            /* Überlagerung des Prozesses "exec_menu" mit
            dem
            Shell-kommando */
            execlp(kommando[i], kommando[i], 0);

            /* Falls es den Shell-Befehl nicht gibt, wird
            der
            Prozess nicht überlagert und die Fehlermeldung
            kann ausgegeben werden*/
            printf("Kommando nicht gefunden.\n");
            exit(1);

        } else { /* Elternprozess */

            /* Elternprozess wartet auf Kindprozess */
            wait(&rc);
            printf ("Kind mit %d beendet \n",

            WEXITSTATUS(rc));
        }
    }
}
```

Aufgabe 7: Schreiben Sie ein C-Programm mit einem Interrupt-Handler für die Signale SIGINT und SIGQUIT mit folgendem Verhalten: Das erste Senden von SIGINT (bzw. SIGQUIT) wird vom Interrupt-Handler abgefangen (Ausgabe mit printf()). Weitere SIGINT (bzw. SIGQUIT) Signale werden ignoriert. Ein Aufruf von SIGQUIT aktiviert den Interrupt-Handler für SIGINT wieder und umgekehrt.

```
/* signale.c - Übung 6, Aufg. 7 */
/* Interrupthandler für SIGQUIT und SIGINT */

#include <stdio.h>
#include <sys/signal.h>

void sig_handler(int sig) {
    printf("Signal %d empfangen\n", sig);
    if (sig == SIGQUIT){
        /* SIG_IGN = ignorieren */
        signal(SIGQUIT, SIG_IGN);
        signal (SIGINT, sig_handler);
    } else if (sig == SIGINT) {
        signal(SIGINT, SIG_IGN);
        signal (SIGQUIT, sig_handler);
    }
}

int main(void) {
    int i=0;
    signal (SIGQUIT, sig_handler);
    signal (SIGINT, sig_handler);
    while(1) {
        printf("working... %d\n",i++);
        sleep(3);
    }
}
```

Übung 7:

In dieser Übung soll eine einfache Shell, die als C-Programm shell.c realisiert ist, übersetzt und ausgeführt werden. Nachfolgend soll de shell mit einer neuen Funktion ausgestattet werden.

Aufgabe 1: Übersetzen Sie shell.c und machen Sie sich mit der Funktionsweise des Programms vertraut. Studieren Sie dazu die Manual-Seiten zu den benutzen Bibliotheksfunktionen und Systemaufrufen.

Aufgabe 2: Benutzen Sie das Programm strace um sich die ausgeführten Systemaufrufe anzeigen zu lassen.

```
strace -t -f -p <PID>
```

Zuerst muss die Shell gestartet werden, dann wird der obige Befehl in einem anderen Bash-Fenster ausgeführt. `-t` gibt die Zeit mit aus, `-f` folgt Kindprozessen und `-p <PID>` gibt die PID an, die `strace` überwachen soll.

Aufgabe 3: Erweitern Sie shell.c um die Fähigkeit, Programme im Hintergrund zu starten, die parallel zur shell arbeiten. Programme, die im Hintergrund arbeiten sollen, werden dadurch erkannt, dass das letzte Wort in der Kommandozeile ein & Zeichen ist.

Hinweis zu Aufgabe 3: Man kann nicht auf verschiedene Prozesse synchron und auf andere Prozesse asynchron vom Shell-Prozess aus gleichzeitig warten, da die Aktivierung des asynchronen Wartens im Elternprozess für alle Kindprozesse gilt. Es kann aber folgender Trick angewendet werden: Der Hintergrundprozess wird nicht direkt aus dem Shell-Prozess gestartet, sondern indirekt aus einem Kindprozess der Shell. Dieser Kindprozess wird sofort terminiert, wodurch der Elternprozess des Hintergrundprozess der Prozess init mit der PID 1 wird. Beim Ende des Hintergrundprozesses ruft init ein wait() auf, wodurch der Hintergrundprozess ordnungsgemäss terminiert wird und kein Zombie-Prozess entsteht.

```
/* shell.c - Übung 7, Aufg. 3 */
/* Shell mit Hintergrundprozessen */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define BUFLEN 512

static const char *progname = "shell";

static void write_prompt() {
```

```
    fprintf(stdout, "%s> ", progname);
}

static char* next_token(char **s) {

    char *token, *p;
    /* isspace prüft auf Leerzeichen */
    for (p = *s; *p && isspace(*p); p++) {
        *p = '\0';
    }
    token = p;
    for (; *p && !isspace(*p); p++) ;
    for (; *p && isspace(*p); p++) {
        *p = '\0';
    }
    *s = p;
    return token;
}

static void read_command(char **cmd, char ***args) {
    static char line[BUFLEN];
    static char* argv[BUFLEN];
    char *p;
    int i;

    memset((char *) argv, 0, sizeof(argv));
    p = fgets(line, sizeof(line), stdin);

    for (i = 0; p && *p; i++) {

        /* bei der 1. Iteration wird &line an next_token
        übergeben */
        /* p enthält nach dem Aufruf einen Zeiger auf den
        Anfang des nächsten token oder *p = NULL. */
        argv[i] = next_token(&p);
    }
    *cmd = argv[0];
    *args = argv;
}

int main(int argc, char **argv) {

    pid_t pid, bg_pid;
    int status, cnt, i,j;
    char *cmd; // String
    char **args; // String-Array

    while (1) {
        write_prompt();
        read_command(&cmd, &args);
```

```
        if (cmd == NULL || strcmp(cmd, "exit") == 0) {
            break;
        }

        if (strlen(cmd) == 0) {
            continue;
        }
        pid = fork();
        if (pid == -1) {
            perror(progname);
            continue;
        }
        if (pid == 0) {
            cnt=0;
            /* Ermittlung der Länge des Argumenten-Arrays
        */
            while (args[cnt]!=(NULL)){cnt++;}
           /* Wenn das letzte Zeichen ein & ist ->
           Hintergrundprozess */
           if (strcmp(args[cnt-1], "&") == 0) {

               /* Kopie des aktuellen Prozesses */
               bg_pid = fork();

               if (bg_pid!=0){
                   /*Elternprozess endet sofort */
                   exit(0);
               } else {
                   /* Kindprozess löscht das & und
                   startet,
                    * da der Elternprozess schon beendet
                   ist,
                    * wird das Kind zur Waise und wird
                   von
                    * init adoptiert. Somit läuft das
                    * Programm unabhängig von der Shell
                    */
                   args[cnt-1] = NULL;
                   execvp(cmd, args);
                   perror(progname);
                   exit(1);
               }
           }
           execvp(cmd, args);
           perror(progname);
           exit(1);
        } else {
           waitpid(pid, &status, 0);
        }
    }
```

```
    return 0;
}
```

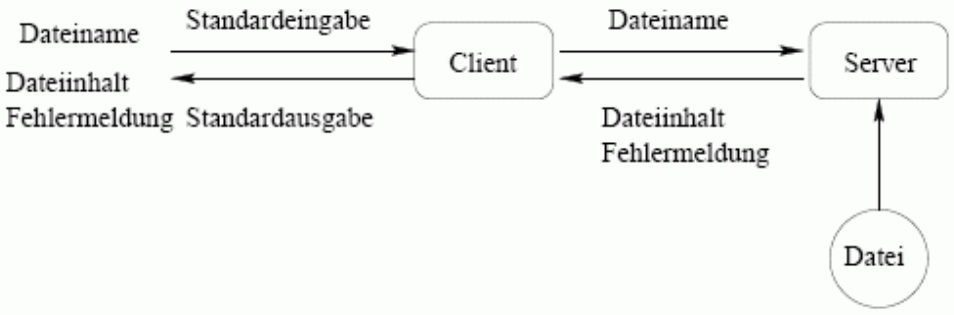
Endlosschleife zum Testen:

```
#include <stdio.h>

int main(void) {
    int i=0;
    while(1) {
        printf("working... %d\n",i++);
        sleep(2);
    }
}
```

Übung 8:

Aufgabe 1: Nachfolgend ist ein Beispiel für ein Client-Server-Modell dargestellt. Realisieren Sie Client und Server, wobei die Prozesse über eine Pipe kommnunizieren. Der Client liest einen Dateinamen von der Standardeingabe und sendet den Namen an den Server. Der Server liest den Dateinamen und versucht, die Datei zu öffnen. Wenn dies gelingt übermittelt er den Dateiinhalt zum Client. Kann der Server die Datei nicht öffnen, so gibt er eine Fehlermeldung an den Client zurück. Der Client gibt den Dateiinhalt bzw. die Fehlermeldung auf der Standardausgabe aus.




```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <fcntl.h>

#define MAXBUFF 1024

void client(readfd, writefd) {
```

```
    char buff[MAXBUFF];
    int n;

    /* Dateiname einlesen */

    printf("Dateiname eingeben: ");
    if (fgets(buff, MAXBUFF, stdin) == NULL) {
        fprintf(stderr, "client: Fehler beim Einlesen");
        return;
    }
    n = strlen(buff);
    if (buff[n-1] == '\n') n--;

    /* Dateiname an den Server schicken */

    write(writefd, buff, n);

    /* Daten vom Server empfangen und auf die
    Standardausgabe senden */

    while ((n = read(readfd, buff, MAXBUFF)) > 0)
        write(1, buff, n);
        if (n < 0) {
            fprintf(stderr, "client: Datenlesefehler");
        }
}

void server(int readfd, int writefd) {
    char buff[MAXBUFF];
    char errmsg[256];
    int n, fd;
    extern int errno;

    /* Dateiname lesen */

    if ((n = read(readfd, buff, MAXBUFF)) <= 0) {
        fprintf(stderr, "Server: Dateiname Lesefehler");
        return;
    }
    buff[n]='\0';

    /* Datei öffnen */

    if ((fd = open(buff, 0)) < 0) {

        /* Öffnen fehlgeschlagen, Fehlermeldung an Client
        schicken */

        sprintf(errmsg, "Server: Öffnen fehlgeschlagen:
        %s\n",
```

```
                buff);
            write(writefd, errmsg, strlen(errmsg));
        } else {

            /* Öffnen erfolgreich, Dateiinhalt an den client
            schicken */

            while ((n = read(fd, buff, n)) > 0)
                write(writefd, buff, n);
            if (n < 0)
                fprintf(stderr, "Server: Lesefehler");
        }
    }

    int main(void) {

        int childpid, pipe1[2], pipe2[2];

        /* Pipes anlegen */

        if (pipe(pipe1) < 0 || pipe(pipe2) < 0) {
            fprintf(stderr, "Pipes können nicht geöffnet
                                                  werden.");
            exit(1);
        }
        if ((childpid = fork()) < 0) {
            fprintf(stderr, "fork() fehlgeschlagen");
            exit(2);
        } else if (childpid > 0) { /* Elternteil */
            close(pipe1[0]);
            close(pipe2[1]);
            client(pipe2[0], pipe1[1]);
            while (wait(0) != childpid);
            close(pipe1[1]);
            close(pipe2[0]);
            exit(0);
        } else { /* Kindteil */
            close(pipe1[1]);
            close(pipe2[0]);
            server(pipe1[0], pipe2[1]);
            close(pipe1[0]);
            close(pipe2[1]);
            exit(0);
        }
    }
```

Aufgabe 2: Realisieren Sie Client und Server aus Aufgabe 1, wobei die Prozesse über named pipes (FIFO) kommunizieren.

Header:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <fcntl.h>
extern int errno;

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS 0666
#define MAXBUFF 1024
```

Server:

```
#include "header.h"

void server(int readfd, int writefd) {
    char buff[MAXBUFF];
    char errmsg[256];
    int n, fd;
    extern int errno;

    /* Dateiname lesen */

    if ((n = read(readfd, buff, MAXBUFF)) <= 0) {
        fprintf(stderr, "Server: Dateiname Lesefehler");
        return;
    }
    buff[n]='\0';

    /* Datei Öffnen */

    if ((fd = open(buff, 0)) < 0) {

        /* Datei öffnen fehlgeschlagen, Fehlermeldung
        ausgeben */

        sprintf(errmsg, "Server: Kann Datei nicht öffnen:
                                                %s\n",
        buff);
        write(writefd, errmsg, strlen(errmsg));
    } else {

        /* Datei öffnen erfolgreich, Dateiinhalt an den
        Client schicken */

        while ((n = read(fd, buff, MAXBUFF)) > 0)
            write(writefd, buff, n);
        if (n < 0)
            fprintf(stderr, "Server: Lesefehler");
```

```
        }
    }

    int main() {
        int readfd, writefd;

        if ((mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno
        !=

            EEXIST))
            perror("Kann FIFO1 nicht erstellen");
        if ((mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno
        !=

            EEXIST)) {
            unlink(FIFO1);
            perror("Kann FIFO2 nicht erstellen");
        }
        if ((readfd = open(FIFO1, O_RDONLY)) < 0)
            perror("Kann Lesefifo nicht Ã¶ffnen");
        if ((writefd = open(FIFO2, O_WRONLY)) < 0)
            perror("Kann Schreibfifo nicht Ã¶ffnen");
        server(readfd, writefd);
        close(readfd);
        close(writefd);
        exit(0);
    }
```

Client:

```
#include "header.h"

void client(readfd, writefd) {
    char buff[MAXBUFF];
    int n;

    /* Dateiname von Standardeingabe lesen */

    printf("Dateinamen eingeben: ");
    if (fgets(buff, MAXBUFF, stdin) == NULL) {
        fprintf(stderr, "Client: Dateiname Lesefehler");
        return;
    }
    n = strlen(buff);
    if (buff[n-1]=='\n') n--;

    /* Dateiname an Server schicken */

    write(writefd, buff, n);
```

```
    /* Daten vom Server lesen und auf die Standardausgabe
    schreiben */

    while ((n = read(readfd, buff, MAXBUFF)) > 0)
        write(1, buff, n);
    if (n < 0)
        fprintf(stderr, "Client: Daten-Lesefehler");
}

int main(void) {
    int readfd, writefd;

    if ((writefd = open(FIFO1, O_WRONLY)) < 0)
        perror("Kann Schreibfifo nicht öffnen");
    if ((readfd = open(FIFO2, O_RDONLY)) < 0)
        perror("Kann Schreibfifo nicht öffnen");
    client(readfd, writefd);
    close(readfd);
    close(writefd);
    unlink(FIFO1);
    unlink(FIFO2);
    exit(0);
}
```

Aufgabe 3: Realisieren Sie Client und Server aus Aufgabe 1, wobei die Prozesse über eine Message Queue miteinander kommunizieren.

Server:

```
    #include <stdio.h>
    #include <sys/msg.h>

    #define KEY ((key_t) 21)
    #define BUFFERSIZE 1024

    int main()
    {
        struct my_msg {
            long mtype;
            char c[];
        };

        struct my_msg message, send;

        char puffer[BUFFERSIZE], temp[BUFFERSIZE];
        int count, mq, msg_id, msg_type = 7688;
        FILE *f;

        /* Anlegen der Message-Queue */

        if(mq=msgget(KEY,0666|IPC_CREAT) == -1){
```

```
    printf("Fehler beim Erzeugen der Message-
    Queue\n");
    exit(1);
} else {
    printf("Message-Queue erfolgreich angelegt\n");
}

/* Verbindung zur Message Queue erstellen */

if ((msg_id = msgget(KEY,0))<0) {
    printf("Message-Queue existiert nicht!\n");
    exit(1);
} else {
    printf("Verbindung zur Message-Queue
erstellt.\n");
}

/* Nachricht von Client empfangen */

if
((count=msgrcv(msg_id,&message,BUFFERSIZE,msg_type,0)
)

    == -1)
{
    printf("Nachrichtenabruf fehlgeschlagen\n");
    exit(3);
} else {
    printf("Nachricht empfangen: ");
}

/* Verarbeitung der Nachricht */

memcpy(puffer, message.c, count);
printf("%s\n",puffer);

/* Datei öffnen & Inhalt lesen */

if((f = fopen(puffer,"r")) == 0){
    count=sprintf(temp,"Fehler beim Öffnen der Datei

        %s!\n",puffer);
    strcpy(puffer,temp);
} else {
    count=fread(puffer,1,BUFFERSIZE,f);
    fclose(f);
}

/* Dateiinhalt bzw. Fehlermeldung zurück an Client */

memcpy(message.c, puffer, strlen(puffer));
```

```
        message.mtype = msg_type;

        /* Message senden */

        if (msgsnd(msg_id, &message, strlen(puffer), 0) == -
        1)
        {
            printf("Fehler beim Senden\n");
            exit(1);
        } else {
            printf("Nachricht zurückgesendet\n");
        }
        exit(0);
    }
```

Client:

```
    #include <stdio.h>
    #include <sys/msg.h>

    #define KEY ((key_t) 21)
    #define BUFFERSIZE 1024

    int main()
    {
        struct my_msg {
            long mtype;
            char c[];
        };

        struct my_msg message, rec;

        char puffer[BUFFERSIZE], lesen[BUFFERSIZE];
        int count, msg_id;
        int msg_type = 7688;

        /* Verbindung zur Message Queue erstellen */

        if ((msg_id = msgget(KEY,0))<0) {
            printf("Message-Queue existiert nicht!\n");
            exit(1);
        } else {
            printf("Verbindung zur Message-Queue
        erstellt.\n");
        }

        /* Dateinamen einlesen */

        printf("Geben Sie den Namen der Datei an: ");
        scanf("%s",puffer);
        fflush(stdout);
```

```
        /* Message erstellen */

        if (puffer[strlen(puffer) - 1] == '\n')
            puffer[strlen(puffer) - 1] = '\0';
        memcpy(message.c, puffer, strlen(puffer));
        message.mtype = msg_type;

        /* Message senden */

        if (msgsnd(msg_id, &message, strlen(puffer), 0) == -
        1)
        {
            printf("Fehler beim Senden\n");
            exit(1);
        } else {
            printf("Nachricht gesendet\n");
        }

        /* Nachricht von Server empfangen */

        if
        ((count=msgrcv(msg_id,&message,BUFFERSIZE,msg_type,0)
        )

            == -1)
        {
            printf("Nachrichtenabruf fehlgeschlagen\n");
            exit(3);
        } else {
            printf("Nachricht empfangen:\n\n");
        }

        /* Verarbeitung der Nachricht */

        memcpy(puffer, message.c, count);
        puffer[count-1] = '\0';
        printf("%s\n",puffer);

        /* Message-Queue schließen */

        if (msgctl(msg_id, IPC_RMID, 0) == -1){
            printf("Message-Queue nicht geschlossen\n");
        } else {
            printf("Message-Queue geschlossen\n");
        }
        exit(0);
    }
```

Übung 9:

Realisieren Sie Client und Server aus Übung 8 Aufgabe 1, wobei die Prozesse über ein shared memory (gemeinsamen Speicherbereich) kommunizieren. Gehen Sie dabei folgendermaßen vor:

- Benutzen Sie folgende System-Header:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
```

- Benutzen Sie z.B. folgende Struktur zur Kommunikation im shared memory:

```
#define MAXMESGDATA 128
#define MESGHDRSIZE (sizeof(Mesg) - MAXMESGDATA)
typedef struct {
        int mesg_len;
        char mesg_data[MAXMESGDATA];
} Mesg;
```

- Schreiben Sie das Server-Programm mit folgendem Ablauf:

Es erzeugt shared memory (shmget) und blendet ihn in den Adressraum des Programms ein (shmat). Ferner erzeugt (semget) und initialisiert (semctl) der Server einen Client- und einen Server-Semaphor. Wegen dieser Initialisierungen muss der Server vor dem Client gestartet werden. (Zur Programmierung des Initialisierens eines Semaphors lesen Sie bitte aufmerksam die Manual Page zu semctl.) Dann liest der Server den Dateinamen vom Client und versucht, die zugehörige Datei zu öffnen. Falls das Öffnen fehlschlägt, wird eine Fehlermeldung

an den Client gesendet. Ansonsten wird der Inhalt der Datei stückweise in den gemeinsamen Speicher übertragen. Zum Schluss wird der shared memory aus dem Adressraum entfernt (shmdt).

- Schreiben Sie das Client-Programm mit folgendem Ablauf:

Es öffnet den vom Server erzeugten shared memory (shmget) und blendet ihn in den Adressraum des Programms ein (shmat). Ferner öffnet es die Semaphore (semget). Dann liest es den Dateinamen von der Tastatur ein und überträgt ihn in den shared memory. Anschließend liest es in einer Schleife die Fehlermeldung oder den aktuellen Teil der Datei aus dem shared memory vom Server ein und gibt ihn auf der Standardausgabe aus. Die Schleife endet, wenn die Länge der Nachricht 0 ist. Zum Schluss wird der shared memory aus dem Adressraum entfernt (shmdt) und gelöscht (shmctl) und die Semaphore werden ebenfalls gelöscht (semctl).

Die Semaphore werden zur Sperrung und Freigabe des shared memory durch Client und Server verwendet. Dabei ist der Ablauf wie folgt:

- Am Anfang wird der Client-Semaphor als frei und der Server-Semaphor als gesperrt initialisiert.
- Der Client beginnt mit der Arbeit, indem er den Client-Semaphor sperrt und den Dateinamen einliest. Der Server wartet auf die Freigabe des Server-Semaphors durch den Client, bevor er mit seiner Arbeit beginnt. Nachdem der Dateiname in den shared memory übertragen ist, befreit der Client den Server-Semaphor und deaktiviert sich selbst durch Ausführung einer erneuten Sperr-Operation auf dem Client-Semaphor.
- Hierdurch wird der Server aktiviert, der seinerseits den Server-Semaphor sperrt und den Dateinamen aus dem shared memory einliest. Dann versucht der Server, die Datei zu öffnen und schreibt eine Fehlermeldung oder einen Teil der Datei in den shared memory. Danach gibt der Server den Client-Semaphor frei und deaktiviert sich selbst durch eine Sperr-Operation auf dem Server-Semaphor.
- Nun sperrt der Client wieder den Client-Semaphor und liest die Nachricht des Servers, die er dann auf der Standardausgabe angezeigt. Danach übergibt er die Kontrolle wieder an den Server durch Freigabe des Server-Semaphor und Sperrung des Client-Semaphor zur Übermittlung des nächsten Teils der Datei.
- Die Aktionen des Client und Server wiederholen sich nun, bis das Dateiende erreicht ist.

header.h:

```
#include <stdio.h>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define SHMKEY ((key_t) 7890)
#define CLIKEY ((key_t) 7891)
#define SERVKEY ((key_t) 7892)
#define PERMS 0666

#define MAXMESGDATA 128
#define MESGHDRSIZE (sizeof(Mesg) - MAXMESGDATA)

typedef struct {
int mesg_len;
char mesg_data[MAXMESGDATA];
} Mesg;
```

Client:

```
#include "header.h"

int shm_id, client_sem, server_sem;
Mesg *mesgptr;

/* sops-Befehlssätze für den semop-Befehl */

static struct sembuf op_lock[2] = {

    /* 1. Stelle: Nummer des Semaphores
       2. Stelle: Operation auf Semaphor
       3. Stelle: Option für diese Operation */

    0,0,0, /* Warten, bis Semaphor=0, also freigegeben */
    0,1,0 /* Semaphor auf 1 setzen, also sperren */
};

static struct sembuf op_unlock[1] = {
    0,-1,0 /* Semaphor wieder auf 0 setzen, also
freigeben */
};

int my_lock(int sem) {

    /* Zuerst wird der Semaphor mit dem 1. Befehlssatz
    aufgerufen:
    Warten, bis Semaphor frei ist
    Wenn das geschehen ist, wird der Semaphor mit dem 2.
    Befehlssatz aufgerufen: Sperren für andere Prozesse
    2 Befehlssätze, weil letzte Option bei semop = 2 */
```

```
        if (semop(sem, &op_lock[0], 2) < 0) {
            perror("Client: Semaphor-Lock-Fehler");
            return(1);
        }
    }

    int my_unlock(int sem) {
        if (semop(sem, &op_unlock[0], 1) < 0) {
          perror("Client: Semaphor-Unlock-Fehler");
          return(1);
        }
    }

    void client(void) {

        int n;

        /* Kontrolle über Shared Memory erlangen */

        my_lock(client_sem);

        /* Dateinamen einlesen */

        fprintf(stdout, "Geben Sie einen Dateinamen ein:\n");
        if (fgets(mesgptr->mesg_data, MAXMESGDATA, stdin) ==

            NULL)
            fprintf(stderr, "Client: Dateinamen-
        Lesefehler\n");
        n = strlen(mesgptr->mesg_data);

        /* Zeilenumbruch entfernen */

        if (mesgptr->mesg_data[n-1]== '\n') n--;
        mesgptr->mesg_len = n;

        /* Server aktivieren */

        my_unlock(server_sem);

        /* Client in Warteposition */

        my_lock(client_sem);

        while ((n = mesgptr->mesg_len) > 0) {
        write(1, mesgptr->mesg_data, n);

        /* Server aktivieren */

        my_unlock(server_sem);
```

```
        /* Client in Warteposition */

        my_lock(client_sem);
        }
    }


    int main(void) {

        /* auf Shared Memory zugreifen */

        if ((shm_id = shmget(SHMKEY, sizeof(Mesg), 0)) < 0) {
            fprintf(stderr, "Client: kann auf Shared Memory
            nicht

                zugreifen\n");
            exit(1);
        }

        /* Shared Memory in den Adressbereich einhängen */

        if ((mesgptr = (Mesg *) shmat(shm_id,0,0))== (Mesg *)
        -1) {
            fprintf(stderr, "Client: Shared Memory kann nicht
            in den Adressbereich eingehängt werden\n");
            exit(2);
        }

        /* auf Client- & Server-Semaphoren zugreifen */

        if ((client_sem = semget(CLIKEY, 1, 0)) == -1) {
            perror("Client: Client-semget fehlgeschlagen");
            exit(3);
        }

        if ((server_sem = semget(SERVKEY, 1, 0)) == -1) {
            perror("Client: Server-semget fehlgeschlagen");
            exit(4);
        }

        client();

        /* Shared Memory aushängen */

        if (shmdt(mesgptr) < 0) {
            fprintf(stderr, "Client: Shared Memory kann nicht
                                        ausgehängt
            werden\n");
            exit(5);
        }
```

```
    /* Shared Memory löschen */

    if (shmctl(shm_id, IPC_RMID,0) < 0) {
        fprintf(stderr, "Client: Shared Memory kann nicht
                                            gelöscht
        werden\n");
        exit(6);
    }

    /* Client-Semaphor löschen */

    if (semctl(client_sem,0,IPC_RMID,0) < 0) {
        fprintf(stderr, "Client: Client-Semaphor kann
        nicht
                                            gelöscht
        werden\n");
        exit(7);
    }

    /* Server-Semaphor löschen */

    if (semctl(server_sem,0,IPC_RMID,0) < 0) {
        fprintf(stderr, "Client: Server-Semaphor kann
        nicht
                                            gelöscht
        werden\n");
        exit(8);
    }
    exit(0);
}
```

Server:

```
#include "header.h"

int shm_id, client_sem, server_sem;
Mesg *mesgptr;

/* Union für die Intialisierung des Semaphors */

union semum {
    int val; /* Wert für SETVAL */
    struct semid_ds *buf; /* Puffer für IPC_STAT, IPC_SET
    */
    unsigned short int *array; /* Array für GETALL,
    SETALL */
    struct seminfo *__buf; /* buffer for IPC_INFO */
} arg;

/* sops-Befehlssätze für den semop-Befehl */
```

```
static struct sembuf op_lock[2] = {

    /* 1. Stelle: Nummer des Semaphores
       2. Stelle: Operation auf Semaphor
       3. Stelle: Option für diese Operation */

    0,0,0, /* Warten, bis Semaphor=0, also freigegeben */
    0,1,0 /* Semaphor auf 1 setzen, also sperren */
};

static struct sembuf op_unlock[1] = {
    0,-1,0 /* Semaphor wieder auf 0 setzen, also
freigeben */
};

int my_lock(int sem) {

    /* Zuerst wird der Semaphor mit dem 1. Befehlssatz
    aufgerufen:
    Warten, bis Semaphor frei ist
    Wenn das geschehen ist, wird der Semaphor mit dem 2.
    Befehlssatz aufgerufen: Sperren für andere Prozesse
    2 Befehlssätze, weil letzte Option bei semop = 2 */

    if (semop(sem, &op_lock[0], 2) < 0) {
        perror("Server: Semaphor-Lock-Fehler");
        return(1);
    }
}

int my_unlock(int sem) {
    if (semop(sem, &op_unlock[0], 1) < 0) {
        perror("Server: Semaphor-Unlock-Fehler");
        return(1);
    }
}

void server(void) {
    int n, filefd;
    char errmesg[256];

    /* Warten auf die Nachricht (Dateinamen) des Client
    */

    my_lock(server_sem);

    mesgptr->mesg_data[mesgptr->mesg_len] = '\0';

    if ((filefd = open(mesgptr->mesg_data, 0)) < 0) {

        /* Fehlermeldung erzeugen */
```

```
            sprintf(errmesg, " kann nicht geöffnet
            werden\n");
            strcat(mesgptr->mesg_data,errmesg);
            mesgptr->mesg_len = strlen(mesgptr->mesg_data);

            /* Client aktivieren */

            my_unlock(client_sem);

            /* Server in Warteposition */

            my_lock(server_sem);
        } else {

            /* Dateiinhalt senden */

            while ((n=read(filefd, mesgptr->mesg_data,
            MAXMESGDATA
                                                    - 1))
            > 0)
            {
                mesgptr->mesg_len = n;

                /* Client aktivieren */

                my_unlock(client_sem);

                /* Server in Warteposition */

                my_lock(server_sem);
            }
            close(filefd);
            if (n<0) fprintf(stderr, "Server: Lesefehler\n");
        }

        /* Nachricht mit der Länge 0 als Endzeichen */

        mesgptr->mesg_len = 0;

        /* Client aktivieren */

        my_unlock(client_sem);
    }


    int main(void) {

        /* Shared Memory erzeugen */

        if ((shm_id = shmget(SHMKEY, sizeof(Mesg),
```

```
                                        PERMS|IPC_CREAT)) <
0)
{
    fprintf(stderr, "Server: Shared Memory kann nicht
                                            erzeugt
    werden\n");
    exit(1);
}

/* Shared Memory in den Adressbereich einhängen */

if ((mesgptr = (Mesg *) shmat(shm_id,0,0)) ==(Mesg *)
-1) {
    fprintf(stderr, "Server: Shared Memory kann nicht
    in
                den Adressbereich eingehängt
    werden\n");
    exit(2);
}

/* Client- & Server-Semaphoren erzeugen */

if ((client_sem = semget(CLIKEY, 1, 0)) == -1) {

    /* Wenn Zugriff auf Semaphor wegen Nichtexistenz
    fehlschlägt, wird Semaphor selbst erzeugt */

    if ((client_sem = semget(CLIKEY, 1, IPC_CREAT |
                                            PERMS)) <
    0)
    {
        perror("Server: Client-semget
        fehlgeschlagen");
        exit(3);
    }
}

if ((server_sem = semget(SERVKEY, 1, 0)) == -1) {

    /* Wenn Zugriff auf Semaphor wegen Nichtexistenz
    fehlschlägt, wird Semaphor selbst erzeugt */

    if ((server_sem = semget(SERVKEY, 1, IPC_CREAT |
                                            PERMS)) <
    0) {
        perror("Server: Server-semget
        fehlgeschlagen");
        exit(4);
    }
}
```

```
    /* Semaphoren initialisieren */

    arg.val=0; /* SETVAL auf 0 setzen */
    if (semctl(client_sem, 0, SETVAL, arg) == -1) {
        perror("Client-Semaphor-Initialisierung

            fehlgeschlagen");
        exit(5);
    }

    arg.val=1; /* SETVAL auf 1 setzen */
    if (semctl(server_sem, 0, SETVAL, arg) == -1) {
        perror("Server-Semaphor-Initialisierung

            fehlgeschlagen");
        exit(6);
    }

    server();

    /* Shared Memory aushängen */

    if (shmdt(mesgptr) < 0) {
        fprintf(stderr, "Server: Shared Memory kann nicht
                                        ausgehängt
        werden\n");
        exit(7);
    }
    exit(0);
}
```

Übung 10:

Für ein einfaches zweidimensionales Kalkulationsblatt soll gelten, dass die letzte Zahl jeder Zeile gleich der Summe der anderen Zahlen der Zeile ist und dass die letzte Zahl jeder Spalte gleich der Summe der anderen Zahlen in dieser Spalte ist.

Beispiel:

7	4	15	9	12	47
3	21	5	2	6	37
8	18	6	1	13	46
20	10	17	6	5	58
1	1	1	1	1	5
39	54	44	19	37	

Auf dieses Kalkulationsblatt greifen zwei Prozesse updater und checker zu. Der updater verändert periodisch eine zufällig ausgewählte Zelle in einen zufälligen Wert und berichtigt die Summe der Zeilen und Spalten. Der checker gibt das Kalkulationsblatt aus und überprüft, ob die Summen korrekt sind.

Das Kalkulationsblatt wird in einem gemeinsamen Speicher untergebracht. Ohne die Anwendung von Semaphoren zur Synchronisation der Prozesse findet der checker Fehler.

Realisieren Sie also einen Sperrmechanismus mit Semaphoren. Der einfachste Ansatz wäre eine Semaphore zum Sperren der gesamten Matrix zu verwenden. Dies schränkt den checker aber zu sehr ein, da er nur die Zeilensumme und Spaltensumme der geänderten Zelle nicht überprüfen und drucken darf. Also definieren wir zwei Semaphorensätze einen für Spalten und einen für Zeilen.

Implementieren Sie den updater als Kind des checkers. Hierdurch werden die Semaphorsätze und der gemeinsame Speicher nur im checker angelegt und an den updater vererbt. Das Kind hat hierdurch die Systemaufrufe zum Öffnen des gemeinsamen Speichers und der Semaphorsätze eingespart.

```
/* Kalkulationsblatt */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

/* nolock =1 zur Simualtion von Fehlern ohne Semaphoren,
sonst noclock = 0 */

int nolock = 0;
```

```
/* Index für Zeilen und Spalten der Summen */

int totalrow, totalcol;

/* Index fr Zeilen und Spalten der Semaphor-Arrays */

int row_semas, col_semas;

/* Makro zum Zugriff der Zelle auf Zeile r und Spalte c
das Kalk.blattes s */

#define CELL(s,r,c) (*((s)+((r)*NCOLS)+(c)))

/* Dimension der Matrix */

#define NROWS 8
#define NCOLS 8

/* Sperren des Semaphors n aus dem Satz sem */

void lock(int sem, int n){
    struct sembuf sop;

    if (nolock) return;

    sop.sem_num = n;
    sop.sem_op = -1;
    sop.sem_flg = 0;

    /* Warten bis Semaphor frei ist */

    semop(sem, &sop, 1);
}

/* Freigabe des Semaphors n aus dem Satz sem */

void unlock(int sem, int n){
    struct sembuf sop;

    if (nolock) return;

    sop.sem_num = n;
    sop.sem_op = 1;
    sop.sem_flg = 0;

    /* Semaphor freigeben */

    semop(sem, &sop, 1);

}
```

```
/* Anlegen des Semaphor-Satzes mit Schlüssel k mit n
Semaphoren */

int make_semas(key_t k, int n){
    int semid, i;

    if (nolock) return 0;

    /* Ein existierender Semaphor-Satzes mit Schlssel k
    wird gelöscht */

    if ((semid = semget(k,n,0)) != -1)
    semctl(semid,0,IPC_RMID);

    if ((semid = semget(k, n, IPC_CREAT | 0600)) != -1){

        /* Alle Semaphore freigeben */

        for (i = 0; i < n; i++)
            unlock(semid, i);
        }
        return semid;
}

/* Schlssel für shared memory und Zeilen Semaphor-Array.
Das Spalten-Semaphor-Array hat den Schlssel SHEET_KEY +1
*/

#define SHEET_KEY 1841

/* make_random_entry selektiert zufi¿½lig eine Zelle, fgt
einen zufälligen Wert ein und berechnet die Zeilen- und
Spaltennummern neu. */

void make_random_entry(int *s){
    int row, col, old, new;

    /* Berechnen der Zellen-Indices und des neuen Wertes
    */

    row = rand() % (NROWS-1);
    col = rand() % (NCOLS-1);
    new = rand() % 100;

    /* Sperren der Zeile und Spalte */

    lock(row_semas, row);
    lock(col_semas, col);

    /* Neuen Zellenwert eintragen */
```

```
        old = CELL(s, row, col);
        CELL(s, row, col) = new;

        sleep(1);

        /* Neuberechnung der Summen */

        CELL(s, row, totalcol) += (new-old);
        CELL(s, totalrow, col) += (new-old);

        /* Ende des kritischen Bereichs */

        unlock(col_semas, col);
        unlock(row_semas, row);

    }

    void print_and_check(int *s){
        int row, col, sum, totalbad;

        static int scount = 0;

        totalbad = 0;
        scount++;
        for (row = 0; row < NROWS; row++){

            sum = 0;

            /*Start des kritschen Bereichs*/

            lock(row_semas, row);

            for (col = 0; col < NCOLS; col++){
                if (col != totalcol)
                sum += CELL(s, row, col);
                printf("%5d", CELL(s, row, col));
            }
            if (row != totalrow)
                totalbad += (sum != CELL(s, row, totalcol));

            /* Ende des kritischen Berichs*/

            unlock(row_semas,row);
            printf("\n");

        }

        for (col = 0; col < totalcol; col++){

            sum = 0;
```

```
            /* Start des kritischen Bereichs */

            lock(col_semas,col);

            for (row = 0; row < totalrow; row++)
                sum += CELL(s, row, col);
            totalbad += (sum != CELL(s, totalrow, col));

            /* Ende des kritischen Bereichs */

            unlock(col_semas, col);
        }

        if (totalbad)
            fprintf(stderr, "Kalkulationsblatt %d falsch\n",

            scount);
        else
            fprintf(stderr, "Kalkualtionsblatt %d korrekt\n",

            scount);

        if ((scount % 100) == 0)
            fprintf(stderr, "Kalkulationsblatt %d
        berechnet\n",

            scount);

        printf("-----------------------------------------
        \n");

        sleep(1);

    }

    int main(void){

        int id, row, col, *sheet;

        setbuf(stdout, NULL);
        setbuf(stderr, NULL);

        totalrow = NROWS - 1;
        totalcol = NCOLS - 1;


        id = shmget(SHEET_KEY, NROWS*NCOLS*sizeof(int),
        IPC_CREAT

        0600);
        if (id < 0){
```

```
        perror("shmget failed:");
        exit(1);
    }

    sheet = (int *)shmat(id, 0, 0);
    if (sheet <= (int *)(0)){
        perror("shmat failed:");
        exit(2);
    }

    /* Zellen zu Null setzen */

    for (row = 0; row < NROWS; row++)
        for (col = 0; col < NCOLS; col++)
            CELL(sheet, row, col) = 0;

    /* Anlegen der Semaphore */

    row_semas = make_semas(SHEET_KEY, NROWS);
    col_semas = make_semas(SHEET_KEY + 1, NCOLS);

    if ((row_semas < 0) || (col_semas < 0)){
        perror("shmget failed:");
        exit(3);
    }

    if (fork()){ /* Elterprozess */
        while (1)
            print_and_check(sheet);
    }else {/* Kindprozess */
        while (1)
            make_random_entry(sheet);
    }
}
```