Inhaltsverzeichnis

1. OS: posix

1.1. Ziele

☑ Die **Schnittstellen zum Betriebssystem kennen lernen und benutzen können**.

☑ Wir wollen hier die im **POSIX** (http://de.wikipedia.org/wiki/POSIX)
(**Portable Operating System Interface**) beschriebenen Methoden und Konzepte zum

- ☐ **Multitasking** (Nebenläufigkeit) und der
- ☐ **Prozesskommunikation/-Synchronisation** (IPC=InterProcessCommunication) üben.

☑ Im Detail werden folg. Themen/Begriffe besprochen:

- ☐ Signale (signal(),alarm(), ...)
- ☐ Prozesse (fork(), exec(), wait(), ...)
- ☐ Kommunikation (pipe(), popen(),fdopen(), streams)
- ☐ Synchronisation: Lock-Files

☑ Workshop:

- ☐ http-server mit fork()
- ☐ Parallelles Sortieren (MergeSort)

☑ Quellen:
- ☐ https://computing.llnl.gov/tutorials/pthreads/

1.2. Was bedeutet POSIX

(*http://de.wikipedia.org/wiki/POSIX*)

Das **Portable Operating System Interface** (**POSIX** ['pɒzɪks]) ist ein

gemeinsam von der

1. IEEE und
2. der Open Group

für *Unix* entwickeltes standardisiertes Application Programming Interface, das die Schnittstelle zwischen Applikation und demBetriebssystem darstellt.

Die internationale Norm trägt die Bezeichnung ISO/IEC/IEEE 9945.

1.2.1. POSIX für Windows User:

http://www.dogsbodynet.com/openr/install_cygwin.html

1.3. Signale: alarm(), signal() (4)

Signale werden zur einfachen **Kommunikation zwischen Programmen** verwendet.

Mögliche Anwendungen sind:

☑ auf **Programmabbruch durch Ctrl-C** reagieren können

☑ einem Programm **während seiner Laufzeit mitteilen**, es möge seine Konfiguration neu auslesen

☑ auf einen **Timeout** reagieren können

...

☑ Hinweis:
alarm - set an alarm clock for delivery of a signal
signal - ANSI C signal handling
man 3 alarm
http://www.manpagez.com/man/3/Signal/

1.3.1. Beispiel: selfalarm.c

```
// a.hofmann mar2004
// selfalarm.c

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

#include <unistd.h>

int main(int argc, char * argv[]){
    if (argc==1){
        printf("usage: selfalarm.exe #"); exit(1);
    }

    alarm(10);    //nach 10 sekunden, bricht das Programm ab

    while(1)
        printf("%s", argv[1]);

     return 0;
}

// gcc selfalarm.c -o selfalarm.exe
// starte mit: ./selfalarm.exe 1 & ./selfalarm.exe 0
```

Starten Sie das Programm selfalarm.exe 2× gleichzeitig mit

```
./selfalarm.exe 1 & ./selfalarm.exe 0
```

Die Eingabe "`x & y`" startet die beiden Kommandos x und y gleichzeitig.

☑ Frage:
Können Sie die Wirkung des *preemptive Multitasking* beobachten?

Cooperative Scheduling
Zustand *running*, bis **freiwilliger** Zustandswechsel
Scheduler kann laufenden Prozess nicht unterbrechen

Preemptive Scheduling

Zustand *running* höchstens bis Ablauf des Zeitquantums ("Verdrängung")
Scheduler kontrolliert Zustandswechsel

Abgesehen von SIGSTOP und SIGKILL kann man das Standardverhalten jedes Signals durch Installation einer Signal-Bearbeitungsroutine anpassen.

Eine **Signal-Bearbeitungsroutine** ist eine Funktion, die vom Programmierer implementiert wurde und jedes Mal aufgerufen wird, wenn der Prozess ein entsprechendes Signal empfängt.

Eine Funktion, die als Signal-Bearbeitungsroutine fungieren soll, muss einen einzigen **Parameter vom Typ int und einen void-Ruckgabetyp** definieren. Wenn ein Prozess ein Signal empfängt, wird die Signal-Bearbeitungsroutine mit der Kennnummer des Signals als Argument aufgerufen.

Signale finden Sie in der Header-Datei **/usr/include/bits/signum.h**

Name	Wert	Funktion
SIGHUP	1	Logoff
SIGINT	2	Benutzer-Interrupt (ausgelöst durch [Strg]+[C])
SIGQUIT	3	Benutzeraufforderung zum Beenden (ausgelöst durch [Strg]+[\])
SIGFPE	8	Fließkommafehler, beispielsweise Null-Division
SIGKILL	9	Prozess killen
SIGUSR1	10	Benutzerdefiniertes Signal
SIGSEGV	11	Prozess hat versucht, auf Speicher zuzugreifen, der ihm nicht zugewiesen war
SIGUSR2	12	Weiteres benutzerdefiniertes Signal
SIGALRM	14	Timer (Zeitgeber), der mit der Funktion `alarm()` gesetzt wurde, ist abgelaufen
SIGTERM	15	Aufforderung zum Beenden
SIGCHLD	17	Kindprozess wird aufgefordert, sich zu beenden
SIGCONT	18	Nach einem `SIGSTOP`- oder `SIGTSTP`-Signal fortfahren
SIGSTOP	19	Den Prozess anhalten
SIGTSTP	20	Prozess suspendiert, ausgelöst durch [Strg)+[Z]

Wir wollen nun, wenn das Signal SIGALRM erzeugt wird, das Programm nicht abbrechen lassen, sondern eine eigene C-Funktion aufrufen lassen.

1.3.2. Beispiel: id-code.c

Bringen Sie folgendes Programm zum Laufen und erklären Sie seine Funktion

```
/* Datei: id-code.c Hofmann Anton
 * Demo: Zeiger auf Funktionen und Interprocess Communication
 * Read with timeout
 */
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int toolong;
```

```
int count;

void wakeup(int signum){
    toolong=1;
    count++;
    #ifdef DEBUG
    printf("wakeup was called\n"); fflush(stdout);
    #endif

    alarm(10);
}

int main(){
    int idcode;
    void (*alrm_func)();
     //Pointer,um die Adresse der 'alten' Alarmfunktion zu speichern.

    while(1) {
        printf("Wie lautet Ihr ID-CODE ? ");
        toolong= 0;
           // setze das Alarmsignal auf wakeup() und
           //speichere den alten Zeiger alrm_func.
        alrm_func= signal(?????????, ??????????);

        /* setze den Alarm-Timer */
        alarm(10);

        /* lies ID-CODE */
        scanf("%d", &idcode);

           /* ID-CODE wurde ohne timeout eingeben->verlasse Schleife */
        if (toolong == 0) break;

        if (count == 1) /* Antwort auf ersten timeout */ {
            printf("\nIhren ID-CODE finden Sie auf Ihrer ID-KARTE.\n");
            fflush(stdout);
        }
        else {
            printf("\nFragen Sie im Sekretariat nach einer neuen ");
            printf("ID-KARTE, \nfalls sie verloren gegangen ist.\n");
            exit(1); /* PROGRAMMABBRUCH */
        }
    } /* end_while */

    /* setze das Alarmsignal wieder zurueck*/
    signal(???????????, ?????????????);
    alarm(0);

    printf("\nID-CODE: %d\n", idcode);

     return 0;

}
```

☑ Hinweis:

gcc -DDEBUG .... bewirkt, dass auch die Anweisungen zwischen #define DEBUG .... #endif übersetzt werden

1.3.3. Aufgabe: wecker.c - Wecker mit Text

Wir wollen einen "Wecker mit Text" erstellen. Man startet diesen und übergibt ihm eine Zeit in Sekunden und einen Text. Wenn diese Zeit abgelaufen ist, wird vom Wecker ein Signal (SIGALRM) abgesetzt. Standardmäßig wird dann das laufende Programm beendet. Wir wollen aber, dass das Programm vor dem Ende noch den Text ausgibt.

Aufruf: ./wecker.exe 10 "Kaffeepause mit Bob" &

1.3.4. Beispiel: control-c.c

Schreiben Sie ein Programm, das beim erstmaligen Drücken von Ctrl-C den Text "Control-c" am Bildschirm ausgibt. Beim zweiten Drücken von Ctrl-C soll das Programm beendet werden. (exit(1)).

Bringen Sie folgendes Programm control-c.c zum Laufen.

```
/* hofmann anton
 * control-c.c
 * demo: SIGINT abfangen
 * Ctrl+C fuehrt erst beim 2.mal zu einem Programmabbruch
 * 2004
 */

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

int flagSIGINT = 0;

/* beim 1. Aufruf wird die globale Variable flagSIGINT inkrementiert;
   beim 2. Aufruf erfolgt der Programmabbruch */

void mySIGINT( int signum){
    if (flagSIGINT == 0){
        flagSIGINT++;
    }
    else{
        printf("SIGINT zum 2. Mal\n");
        exit(1);
    }
}


int main (){
    printf("Demo: Erst beim 2. Auftreten von SIGINT erfolgt der
Programmabbruch");
    fflush(stdout);


    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal (SIGINT, mySIGINT);
```

```
    for (;;){
        printf("Hello, world !\n");
    }
}
```

☑ Hinweis:
`if (signal(SIGINT, SIG_IGN) != SIG_IGN)`

☐ SIG_IGN bedeutet, dass das Signal ignoriert werden soll.

☐ SIG_DFL bedeutet, dass die Standardbearbeitung fur das Signal eingestellt werden soll.

1.3.5. Beispiel: kill-usr1-usr2.c

Mit dem Kommando kill (man kill) kann man von der Shell aus Programmen Signale schicken. Oft werden hier die Signale USR1 bzw. USR2 verwendet. Hier ein Beispiel:

```
 * aufruf: ./kill-usr1-usr2.exe &
 * [1]     4720
 * $ kill -USR1  4720
 * $ kill -USR2  4720
 * $ kill  4720
```

Bringen Sie das folgende Programm kill-usr1-usr2.c zum Laufen:

```
/* Datei: kill-usr1-usr2.c       Hofmann Anton
 * demo signal()
 * gcc kill-usr1-usr2.c -o kill-usr1-usr2.exe
 * aufruf: ./kill-usr1-usr2.exe &
 * [1]     4720
 * $ kill -USR1  4720
 * $ kill -USR2  4720
 * $ kill  4720
 */

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

static void sig_usr (int); /* one handler for both signals */

int main(){
      if (signal (SIGUSR1, sig_usr) == SIG_ERR)
            perror ("Can't catch SIGUSR1");
      if (signal (SIGUSR2, sig_usr) == SIG_ERR)
            perror ("Can't catch SIGUSR2");

      for (;;)
            pause();
            // pause() forces a process to pause until a signal is received

      return 0;
}
```

```
/* ------------------------------------------------ */
static void
sig_usr( int signo)    /* argument is signal number */
{

      if (signo == SIGUSR1)
            printf("received SIGUSR1\n");
      else if (signo == SIGUSR2)
            printf("received SIGUSR2\n");
      else
            printf ("received signal %d\n", signo);
}
```

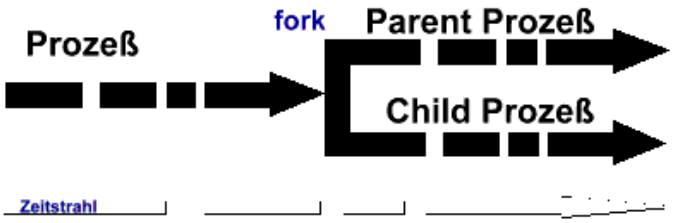
☑ Hinweis*:*
siehe auch man 2 kill

1.3.6. Aufgabe: t_minish.c

Schreiben Sie das Programm *t_minish.c (o_strlist.o, o_strlist.h)* derart um, dass bei Auftreten des Signals USR1 die zur Zeit aktive Liste von Befehlen verworfen wird und falls vorhanden die Datei default.msh eingelesen wird.

1.4. Prozesse (4)

1.4.1. fork, vfork - erzeuge einen Kindprozess



Vater- und Kindprozess haben einen **getrennten Adressraum**. Nach einem fork() wird ein sogenannter Child-Prozess erzeugt, der einen eigenen (echte Kopie des Parent-Prozesses) Speicher besitzt. Auch der IP (Instruction Pointer) wird kopiert. Deshalb ist eine Verzweigung nach dem fork() wichtig. (s.u.)

Anmerkung:
Bei **Threads** ist dies nicht der Fall. Hier **teilen sich die Threads den Adressraum**. Jeder Thread besitzt allerdings einen eigenen Stack und Statusinformationen. Dazu aber weiter unten genaueres.

1.4.2. Beispiel: fork-demo.c

Bringen Sie das folgende Programm zum Laufen:

```
// a.hofmann 2012
// fork-demo.c

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

void childs_code();
void parents_code();

int main() {
     int i;
     int pid= getpid();

     printf("\nJust one process till now: PID= %i", pid);
     printf("\nCalling fork() in 5 sec ...\n");

     for(i=1; i<=5; i++){
          printf("%i sec, \n", i);
          sleep(1);
     }

     switch(fork()){
          case  -1: exit(1); // error
          case   0: childs_code(); break;
          default: parents_code(); break;
     }

     printf("\n\n");
     return 0;
}

void childs_code(){
     int pid= getpid();
     int i;

     for (i=1; i<=10; i++){
          printf("\n                    CHILD: PID= %i i=%i\n", pid, i);
          sleep(1);
     }

     exit(0);
}

void parents_code(){
     int pid= getpid();
     int i=1;

     for (i=1; i<=10; i++){
          printf("\nPARENT: PID= %i i=%i\n", pid, i);
          sleep(1);
     }
}

// gcc fork-demo.c -o fork-demo.exe; ./fork-demo.exe
```

☑ Anmerkung:
Erst bei schreibendem Zugriff auf Variablen, werden diese kopiert.
D.h. parent und child haben dann eigene Variablen
☑ Anmerkung:
oft wird statt des switch ein if...else if ... else verwendet;
...
state= fork();
if (state==0) childs_code();
else if(state>0) parents_code();
else exit(1);
...

1.4.3. Aufgabe: selfalarm-fork.c

Schreiben Sie das obige Programm selfalarm.c derart um, dass das Verhalten gleich bleibt, aber nur folgendermaßen aus der Shell aufgerufen werden muss.

./selfalarm-fork.exe 1 0

1.4.4. Aufgabe: file-server-fork.c

Schreiben Sie das Programm **t_fileserver.c** derart um, dass das Filehandling durch einen Child-Prozesse realisiert wird. Dadurch kann der Server bereits den nächsten Request eines Client-Programmes **t_fileclient.c** annehmen.

Frage:
Beim Testen des Programmes finden Sie in der Prozessliste (ps -aux) Einträge mit ....<defunc>. Was ist damit gemeint? Wie kann man dies verhindern?

1.4.5. +Aufgabe: file-server-base64.c

Schreiben Sie das Programm t_fileserver.c, t_fileclient.c derart um, dass der Client bei Eingabe von "get filename" die Datei namens filename öffnet, zum Server schickt. Dieser verschüsselt die empfangenen Daten mit dem base64 Algorithmus und sendet diese an den Client zurück. Der Client zeigt die kodierten Daten auf dem Bildschirm an.

Hinweis:
Bei www.zotteljedi.de finden Sie Hinweise zur Lösung.

1.4.6. +Aufgabe: file-server-infix2postfix.c

Wie oben, aber ein arithmetischer Infix-Ausdruck soll vom Server in einen Postix-Ausdruck umgewandelt werden.

Hinweis: convert.cpp
Das Programm convert.cpp wandelt einen Infix-Ausdruck in einen Postfix-Ausdruck um. Gelesen wird von der Standardeingabe und geschrieben wird in die Standardausgabe.

```
/* Filename:  convert.cpp
   Author:  Br. David Carlson
   Date:  July 9, 1999
```

```
  Last Revised:  December 23, 2001

   This program repeatedly prompts the user to enter an infix expression
   and converts it to postfix, which is printed on the screen.

   Reference:  The basic algorithm and background material can be found
   in Data Structures Using Pascal, 2nd ed. Tenenbaum and Augenstein.
   Prentice-Hall (1986).  See chapter 2.

   Tested with:
      Microsoft Visual C++ 6.0
      Microsoft Visual C++ .NET
      g++ under Linux
*/

#include <iostream>
#include <string>
#include <stack>
#include <cctype>   // to use the tolower function

using namespace std;


void Convert(const string & Infix, string & Postfix);

bool IsOperand(char ch);

bool TakesPrecedence(char OperatorA, char OperatorB);

int main(void) {
     char Reply;

     string Infix, Postfix;   // local to this loop

     cout << "Enter infix expression (e.g. (a+b)/c^2, with no spaces):"
         << endl;
     cin >> Infix;

     Convert(Infix, Postfix);
     cout << Postfix << endl;
     cout << "quit\n"<< endl ;

     return 0;
}


/* Given:  ch   A character.
Task:To determine whether ch represents an operand (here understood
     to be a single letter or digit).
Return: In the function name: true, if ch is an operand,
     false otherwise.
*/
bool IsOperand(char ch) {
   if (((ch >= 'a') && (ch <= 'z')) ||
      ((ch >= 'A') && (ch <= 'Z')) ||
      ((ch >= '0') && (ch <= '9')))
      return true;
```

```
   else
      return false;
   }


/* Given:
OperatorA    A character representing an operator or parenthesis.
OperatorB    A character representing an operator or parenthesis.
Task:   To determine whether OperatorA takes precedence over OperatorB.
Return: In the function name: true, if OperatorA takes precedence over
           OperatorB.
*/
bool TakesPrecedence(char OperatorA, char OperatorB)
   {
   if (OperatorA == '(')
      return false;
   else if (OperatorB == '(')
      return false;
   else if (OperatorB == ')')
      return true;
   else if ((OperatorA == '^') && (OperatorB == '^'))
      return false;
   else if (OperatorA == '^')
      return true;
   else if (OperatorB == '^')
      return false;
   else if ((OperatorA == '*') || (OperatorA == '/'))
      return true;
   else if ((OperatorB == '*') || (OperatorB == '/'))
      return false;
   else
      return true;

   }


/*
Given:  Infix    A string representing an infix expression (no spaces).
Task:   To find the postfix equivalent of this expression.
Return: Postfix  A string holding this postfix equivalent.
*/
void Convert(const string & Infix, string & Postfix)
   {
   stack<char> OperatorStack;
   char TopSymbol, Symbol;
   int k;

   for (k = 0; k < Infix.size(); k++)
      {
      Symbol = Infix[k];
      if (IsOperand(Symbol)){
        Postfix = Postfix + "push ";
     Postfix = Postfix + Symbol;
     k++;
          while (isdigit(Infix[k])){
          Postfix = Postfix  + Infix[k];
          k++;
```

```
    }
    k--;
       Postfix = Postfix + "\n";
     }
     else
        {
        while ((! OperatorStack.empty()) &&
           (TakesPrecedence(OperatorStack.top(), Symbol)))
           {
           TopSymbol = OperatorStack.top();
           OperatorStack.pop();
        if (TopSymbol=='+') Postfix = Postfix + "add\n";
        else if (TopSymbol=='-') Postfix = Postfix + "sub\n";
        else if (TopSymbol=='*') Postfix = Postfix + "mul\n";
        else if (TopSymbol=='/') Postfix = Postfix + "div\n";
           }
        if ((! OperatorStack.empty()) && (Symbol == ')'))
           OperatorStack.pop();   // discard matching (
        else
           OperatorStack.push(Symbol);
        }
    }

  while (! OperatorStack.empty())
     {
     TopSymbol = OperatorStack.top();
     OperatorStack.pop();
     if (TopSymbol=='+') Postfix = Postfix + "add\n";
     else if (TopSymbol=='-') Postfix = Postfix + "sub\n";
     else if (TopSymbol=='*') Postfix = Postfix + "mul\n";
     else if (TopSymbol=='/') Postfix = Postfix + "div\n";
     }
  }
```

1.4.7. Aufgabe: file-server-qrcode.c

Wie oben, aber ein Text soll vom Server in ein QR-Bild umgewandelt werden.

Hinweis: QR-Code
1. qrencode installieren:
   sudo dpkg -i libqrencode3_3.1.1-1_i386.deb
   sudo dpkg -i qrencode_3.1.1-1_i386.deb

2. QR-Bild erzeugen: qrencode -o htlurl.png "http://www.htl-salzburg.ac.at"

1.5. Prozesse überlagern: exec(), execlp()

Wir wollen nun die sogenannte Prozessüberlagerung kennenlernen. Ein Prozess kann sich selbst in einen anderen Prozess "verwandeln".

1.5.1. Beispiel: execlp-demo.c

Bringen Sie das folgende Programm zum Laufen:

```
//Prozessüberlagerung mit execlp(), ....
// a.hofmann 2012
// execlp-demo.c

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

int main(){
    int rv;

    printf("Demo: execlp(): Ausfuehrung von sort:\n");
    printf("Bitte geben sie einzelne Textzeilen ein.\n");
    printf("Diese werden nach <Ctrl-D> sortiert ausgegeben\n");

    rv = execlp("sort", "sort", NULL);

    /* Fehlerfall!!: Dürfte eigentlich nicht mehr ausgeführt werden */
    printf("Fehler bei execlp");
    exit(1);
}

// gcc -o execlp-demo.exe execlp-demo.c; ./execlp-demo.exe
```

Hinweis:
rv = execlp("/bin/ls", "ls", "-l", NULL);

1.5.2. Warten auf Prozese: wait()

Oft bzw. meist ist es praktisch/notwendig, dass der Vaterprozess auf den Kindprozess wartet. Dabei spricht man von einer Art Synchronisation.

Die Funktion **wait**() kann dazu verwendet werden. Man kann dadurch auch den Rückgabewert (return bzw. exit()) des Kindprozesses erhalten. Also eine einfache Art der Kommunikation erreichen.

Hinweis:
```
#include <sys/types.h>
#include <sys/wait.h>
```

3 Formen sind möglich:

1. `pid_of_child= wait (&status);`
   `//status= der vom child mittels exit() zurückgeg. Wert`

2. `pid_of_child= wait ((int*)0);`
   `//status ist für parent nicht interessant`

3. `pid_t waitpid(pid_t pid, int *status, int options);`

   man waitpid
   … The waitpid() system call suspends execution of the calling process until a child specified by pid argument has changed state. **By default, waitpid() waits only for terminated children**, but this behavior is modifiable via the options argument, as described below...

Hinweis zum Exit-Status eines Prozesses:

This status can be evaluated with the following macros (these macros take the stat buffer (an int) as an argument — not a pointer to the buffer!):

WIFEXITED(status)
returns true if the child terminated normally, that is, by calling exit() or _exit(), or by returning from main().

WEXITSTATUS(status)
evaluates to the least significant eight bits of the return code of the child which terminated, which may have been set as the argument to a call to exit() or _exit() or as the argument for a return statement in the main program. This macro can only be evaluated if WIFEXITED returned true.

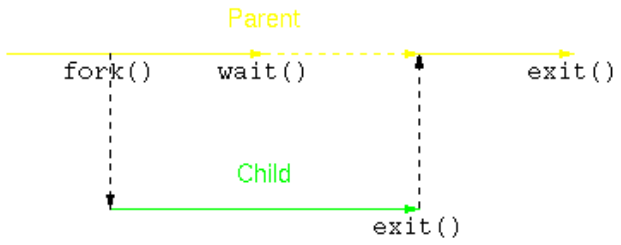
1.5.3. Beispiel: wait.c

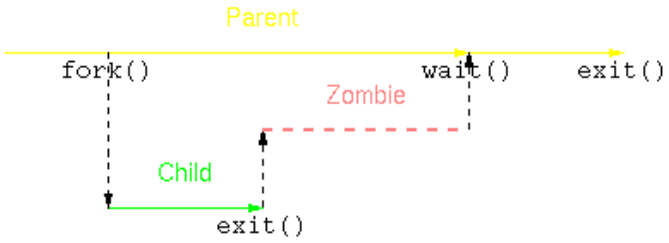
Der laufende Prozess teilt sich durch **fork** auf in Parent und Child, Parent bleibt unverändert. Child lädt durch **exec** ein anderes Programm und verwandelt sich in dieses.

Von da an laufen beide Prozesse konkurrent zueinander ab und verrichten verschiedene Aufgaben. Durch **wait** kann Parent sich mit dem Child **synchronisieren** in dem Sinne, dass er wartet, bis dieser fertig ist. Durch exit() kann das Child zusätzlich eine Erfolgsmeldung an den Parent-Prozess geben.

```
//a.hofmann 2012
// wait.c
//2 unabh. prozesse starten: fork-> execlp | wait

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

int main() {
   int pid;
   printf("Demo: fork(), execlp(), wait(): Ausfuehrung von sort\n");
   printf("Bitte geben sie einzelne Textzeilen ein.\n");
   printf("Diese werden nach <Ctrl-D> sortiert ausgegeben\n");

   pid = fork();

   /* Parent-Prozeß ----------------------  */
   if (pid > 0) {
      int status;

      wait(&status); /* wartet, bis Child fertig */

      printf("PARENT-Process: CHILD is ready & returned: %i\n", status);
      exit(0);
   }

   /* Child-Process ----------------------- */
   if (pid == 0) {

      execlp("sort", "sort", NULL);
```

```
      /* Fehlerfall */
      printf("Fehler bei execlp");
      exit(1);
   }

}
// gcc -o wait.exe wait.c ; ./wait.exe
```





Anmerkung:

Wenn der Parentprozess nicht auf den Childprozess warten soll, aber kein Zombie erzeugt werden soll, verwendet man:

```
waitpid(-1, NULL, WNOHANG);
```

In diesem Fall werden die Zombies vom init-Prozess verwaltet, sobald der Parent beendet.

1.6. IPC: Inter Process Communication (4h)

1.6.1. Beispiel: Named Pipes sind FIFOs

Prozesse kommunizieren mittels **gemeinsamer Dateien (named pipes).**

Prozesse schreiben in Dateien, die von anderen Prozessen gelesen werden.

☑ **Beispiel:**

☐ Öffne 2 Terminals und positioniere sie untereinander

☐ Im Terminal 1:

```
mkfifo /tmp/myFIFO;
ls -l /tmp/myFIFO
cat /etc/passwd > /tmp/myFIFO
```

☐ Im Terminal 2:

```
cat /tmp/myFIFO
rm /tmp/myFIFO
```

1.7. Pipes verbinden verwandte Prozesse

Pipes sind unidirektionale Datenkanäle zwischen zwei Prozessen.

Ein Prozess schreibt Daten in den Kanal (Anfügen am Ende) und ein anderer Prozess liest die Daten in der gleichen Reihenfolge wieder aus (Entnahme am Anfang). Realisierung im Speicher oder als Dateien. Lebensdauer in der Regel solange beide Prozesse existieren.

Beispiel: |

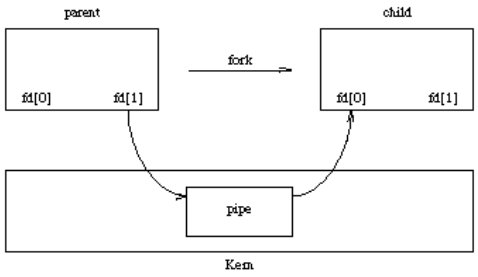
```
cat /etc/passwd | cut -d":" -f1 | sort > usernames.txt
```

Die C-Funktion pipe() liefert 2 Kommunikationskanäle (=Filedeskriptoren). Einer zum Lesen und einer zum Schreiben.

Eine Pipe wird mit dem Systemaufruf **int pipe(int fd[2])** eingerichtet.

Zwei Filedeskriptoren werden als Resultat geliefert, wobei

- **fd[0] zum Lesen** und
- **fd[1] zum Schreiben** geöffnet ist.



Das obige Bild zeigt, dass der PARENT mit fd[1] schreibt und das CHILD mit fd[0] liest. Die jeweils anderen Filedeskriptoren wurden mit close() geschlossen.

Es können die Funktionen **write()** und **read()** bzw. **fgets(),fputs(),** ... verwendet werden.

Hinweis:

```
write(1, buf, 128);
      schreibt 128 Bytes (buf[0] bis buf[127])
      auf den Standardausgabekanal (Bildschirm.)

read(0, buf, 128);
      liest 128 Bytes vom Standardeingabekanal (Tastatur)
```

1.7.1. Beispiel: demo-open.c - unix-like filehandling

Das folgende Beispiel zeigt, wie man in C die low-level Filehandling Funktionen: open(),reade(),write() close() verwendet.

Dadurch soll gezeigt werden, dass ein Grossteil der Ein/Ausgabe mittels Filedeskriptoren

organisiert ist.

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

☑ **int open(const char *pathname, int flags);**
The open() system call is used to convert a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process. This call creates a new open file, not shared with any other process. (But shared open files may arise via the fork(2) system call.) The new file descriptor is set to remain open across exec functions (see fcntl(2)). The file offset is set to the beginning of the file.

The parameter flags is one of O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively, bitwise-or'd.

☑ **size_t read(int fd, void *buf, size_t count);**
read wird solange **blockiert,** bis sich wieder genügend Daten in der Pipe befinden.
Schreibt kein Prozeß mehr in die Pipe bleibt read solange stecken bis der **schreibende** Prozeß den Systemaufruf **close** verwendet hat. Dieses steckenbleiben von read eignet sich prima zum Syncronisieren von Prozessen.

☑ **size_t write(int fd, const void *buf, size_t count);**
write schreibt die Daten in der richtigen Reihenfolge in die Pipe.
Ist die Pipe **voll,** wird der schreibende Prozess solange **angehalten** bis wieder genügend Platz vorhanden ist. Diese Verhalten könnten sie abschalten in dem sie das Flag O_NONBLOCK mit z.B. der Funktion fcntl setzen.
In diesem Fall liefert der schreibende Prozess 0 zurück.

☑ **int close(int fd);**
Schliesst den Filedeskriptor

☑ man pipe

☑ man 2 open

☑ man read write close

1.7.2. Beispiel: demo-open.c

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

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(){
     int fd;
     char buf[128];
     int anz;

     fd=open("demo-open.c", O_RDONLY);
     if (-1==fd){
```

```
            fprintf(stderr, "Error: opening demo-open.c");
            exit(1);
      }

      anz= read(fd, buf, 127);
      while (anz > 0){
            write (1, buf, anz);  // 1...Console

            anz= read(fd, buf, 127);
      }

      close(fd);

      return 0;
}
```

1.7.3. Beispiel: pipe-fork.c – IPC

Das folgende Beispiel zeigt:

1. Das Erstellen einer bidirektionalen Pipe
2. Kommunikation zwischen Parent- und Child Prozess

```
// a.hofmann 2004
// pipe-fork.c

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

#define LESE_KANAL 0
#define SCHREIB_KANAL 1

// der Text soll vom Parent ->zum-> Child geschickt werden.
char *mes1 = "Hallo, Welt Nr.1";
char *mes2 = "Hallo, Welt Nr.2";
char *mes3 = "Hallo, Welt Nr.3";

int main(){
   char buf[128];
   int fd[2], k, pid;

/* Pipe oeffnen */
   if (pipe(fd) < 0) {
      perror("Fehler bei pipe");
      exit(1);
   }

   /* Child-Prozess erzeugen */
   if ((pid = fork()) < 0) {
      perror("Fehler bei fork");
      exit(1);
   }
```

```
   /* Parent-Prozess:
      1. Leseseite der Pipe schliessen und
2. in die Pipe schreiben */
 else if (pid > 0) { /* PARENT ==================== */
 int status;

 close(fd[LESE_KANAL]);

 write(fd[SCHREIB_KANAL], mes1, strlen(mes1)+1);    //inkl EOS
 write(fd[SCHREIB_KANAL], mes2, strlen(mes2)+1);
 write(fd[SCHREIB_KANAL], mes3, strlen(mes3)+1);

 close(fd[SCHREIB_KANAL]);

 wait(&status);

 }

   /* Child-Prozess:
      1. Schreibseite der Pipe schliessen und
      2. von der Pipe lesen */
   else if (pid == 0) { /* CHILD ==================================*/

      int len;
      close(fd[SCHREIB_KANAL]);

      len=read(fd[LESE_KANAL], buf, sizeof(buf));

      printf("%s\n%d Bytes gelesen!\nFrage: Warum wird nur die erste
Zeile angezeigt?\n\n", buf, len);

      close(fd[SCHREIB_KANAL]);

      exit(1);
   }

   return 0;
}

// gcc pipe-fork.c -o pipe-fork.exe; ./pipe-fork.exe
```

Antwort:

Weil im Lesebuffer buf auch die EOS Zeichen stehen und printf bei EOS aufhört Daten auszugeben.

Anmerkung:
anzahl=read(kanal, buf, len) ist blockiernd, d.h. **read() beendet, wenn**

- so viele Daten in den Kommunikationskanal geschrieben wurden, wie in len angegeben sind.
  Dann steht in **anzahl** der gleiche Wert **wie in len**.(s. pipe-demo.c)

  Oder, wenn

- der **Schreibkanal geschlossen** wird. close(fd[1]);
  Dann steht in anzahl die **Anzahl der gesendeten Bytes**.

1.7.4. Beispiel: pipe2.c – FILE* E/A Funktionen verwenden

Wir wollen nun zwischen Prozessen mittels FILE* E/A-Funktionen kommunizieren. Dazu folgendes Beispiel:

Hinweis: FILE* - E/A-Funktionen mit pipe()
Natürlich ist es auch möglich auf Pipes mit Standard Stream E/A - Funktionen zuzugreifen. Dazu müssen sie nur die mit dem pipe() - Aufruf erhaltenen Filedeskriptoren mit der Funktion fdopen () einen Dateizeiger (FILE *) zuteilen. Natürlich müssen sie fdopen mit dem richtigen Modus verwenden. Denn es ist nicht möglich.....

```
FILE *f;
f=fdopen(fd[0], "w"); /*falsch*/
```

...zu verwenden da fd[0] für das Lesen aus einer Pipe steht. Richtig ist dagegen.....

```
FILE *reading, *writing;

reading= fdopen(fd[0], "r");
writing =fdopen(fd[1], "w");
```

Geben sie Ihrem Dateizeiger einfach einen aussagekräftigen Namen um Verwechslungen auszuschliessen. Sehen wir uns dazu wieder ein Beispiel an.....

☑ PARENT:
  ☐ liest von stdin
  ☐ schreibt zum CHILD
☑ CHILD:
  ☐ liest vom PARENT
  ☐ schreibt in eine Datei.

Folgende Streamnamen werden verwendet:
stdin
parent_out
child_in
newfile_out

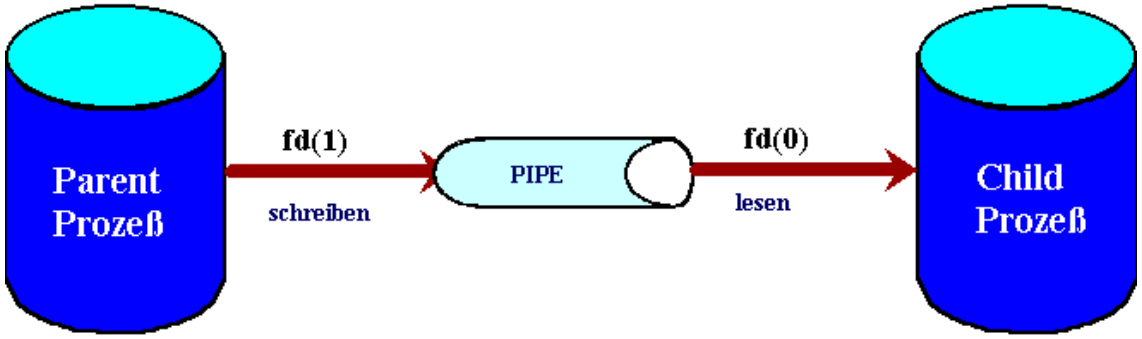
```
/*
 * pipe2.c: Demo: pipe, fork, fdopen
 * PARENT:
 *    fgets(stdin)
 *    fputs(parent_out) →
 * CHILD:
 *    fgets(child_in)
 *    fputs(newfile_out)
 * gcc pipe2.c -o pipe2 ; ./pipe2 pipe2-out.txt
 */

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

```
#include <sys/types.h>
#include <fcntl.h>

#define USAGE printf("usage : %s dateiname.txt\n",argv[0]);
#define MAX 4096

int main(int argc, char *argv[]){
     int fd[2],i, n;
     pid_t pid;
     char puffer[MAX];
     FILE *child_in, *parent_out, *newfile_out;

     if(argc !=2) { USAGE; exit(0); }

     /*Wir erstellen eine pipe*/
     if(pipe(fd) <0) { perror("pipe2: "); exit(0); }


     /*Wir erzeugen einen neuen Prozess*/
     if((pid=fork()) < 0) { perror("pipe : "); exit(0); }

     /* PARENT ================================== */
     else if(pid > 0) {
          close(fd[0]); /*Leseseite schliessen*/

          parent_out=fdopen(fd[1], "w");
          if(parent_out==NULL){ perror(" fdopen : "); exit(0); }

          fputs("\nBitte einen Text: ", stdout);
          fgets(puffer, MAX, stdin); /*Wir lesen von stdin */
          fputs(puffer, parent_out); /*Wir schreiben in die Pipe*/

          fclose(parent_out);

          wait(NULL);
     }
     /* CHILD =================================== */
     else {
          close(fd[1]); /*Schreibseite schliessen*/

          child_in= fdopen(fd[0], "r");
          if(child_in == NULL) { perror(" fdopen : "); exit(0); }

          newfile_out=fopen(argv[1], "a+");
          if (newfile_out == NULL){ perror("fopen : "); exit(0); }

          fgets(puffer, MAX, child_in); /*Wir lesen aus der Pipe*/
          fputs(puffer, newfile_out);   /*und schreiben in die Datei*/


          fclose(newfile_out);
          fclose(child_in);

          exit(0);
     }

     return 0;
```

```
}
```

In der Praxis werden Pipes zur Datenübermittlung zwischen verwandten Prozessen, z.B. zwischen Parent-Prozess und Child-Prozess herangezogen.

Der Grund: die Vererbung der Filedeskriptoren macht es erst möglich, die Pipe durch zwei Prozesse gleichzeitig zu benutzen.



1.7.5. Aufgabe: pipe-toupper.c - toupper

Erstellen Sie das Programm pipe-toupper.c, wobei
der Vaterprozess

- schliesst Lesekanal und
- liest von der Tastaur Text ein (Ende mit "quit") und schreibt diesen in den Schreibekanal (inkl.EOS)

der Child-Prozess

- schliesst den Schreibkanal und
- liest aus dem Lesekanal und
- wandelt das Gelesene in Grossbuchstaben um (toupper()) und
- schreibt den Text auf den Bildschirm

```
printf("                                        CHILD: %s\n", buf);
fflush(stdout);
```

Der Kommunikationsbuffer soll 128 Bytes sein. D.h. während der Vaterprozess die Benutzereingabe liest und an den Childprozess weiter reicht, soll der Childprozesse die gelesenen Daten in Grossbuchstaben umwandeln und ausgeben.

Hinweis:
Verwenden Sie read(), write() zur Kommunikation

1.7.6. Aufgabe: pipe-rate.c

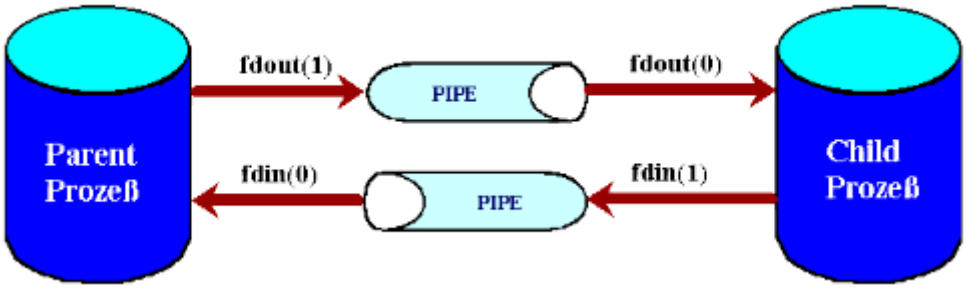
Zwei Prozesse spielen "Zahlenraten"

Parentprozess denkt sich eine Zahl zwischen 1 und 100 aus
Dann wiederholt bis zum Treffer...

1. Childprozess macht einen Rateversuch und schreibt die Zahl in die up-Pipe
2. Parentprozess liest die geratene Zahl aus der up-Pipe
3. Parentprozess bewertet die geratene Zahl mit -1 (zu tief), 0 (getroffen) oder +1 (zu hoch)
4. Parentprozess schreibt die Bewertung in die down-Pipe
5. Childprozess liest die Bewertung aus der down-Pipe und leitet daraus einen neuen Rateversuch ab

Hinweis:
Verwenden Sie fdopen(), fprintf(), fscanf() zur Kommunikation

1.8. Pipes zur bidirektionalen Kommunikation



Eine weitere gängige Aufgabenstellung im Zusammenhang mit Pipes ist die, dass zwei Prozesse (Parent und Child) durch zwei Pipes verbunden sind, die in verschiedene Richtungen wirken. Also ein bidirektionaler Kanal.

1.8.1. Beispiel: pipe2-execlp-sort.c

Verwendung von: fork,close,dup,exec sort
Im folgenden Programm wird die bidirektionale Kommunikation zwischen Prozessen getestet.

Zusätzlich wird der Child-Prozess mit exec durch ein neues Programm überlagert. In unserem Spezialfall soll dieses die UNIX-Utility sort sein. Dabei ergibt sich ein für Pipes charakteristisches Problem: sort kennt wie viele andere UNIX-Dienstprogramme nur die standardmäßigen Filedeskriptoren 0, 1 und 2. sort liest also von stdin und schreibt die sortierten Daten auf stdout.

Anmerkung:
0 ... stdin, 1 ... stdout, 2 ...stderr

Zur Hilfe kommt uns der Systemaufruf dup(). (Dupliziere Filedeskriptoren). Dadurch kann der mit execlp() gestartete Prozess auf die pipes zugreifen.

Dadurch kann also ein externes Programm aufgerufen werden und mit ihm über pipes, wie oben, kommuniziert werden.

Im folgenden Programm kommt dieser Trick öfters vor. Finden Sie heraus, wo! Man muß sich für das Verständnis vor allem die Reihenfolge **fork, close, dup, exec** merken.

Siehe:
dup, dup2 - duplicate a file descriptor
close - close a file descriptor

Anmerkung:
Wenn zB. ein Webserver mit einem externen Programm kommuniziert (zB: SSI= Server Side Include), wird diese Technik verwendet.

```
// a.hofmann 2004
// pipe2-execlp-sort.c
// Beispiel fuer pipe, dup, fork, exec

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

#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

#define BUFLEN 512
#define SCHREIB 1
#define LIES 0

//PARENT: ---> fdparent_out[SCHREIB] ------> CHILD: fdparent_out[LIES]
//PARENT: <--- fdparent_in[LIES]     <------ CHILD: fdparent_in[SCHREIB]


int sortieren(char* fname){
     int fdparent_out[2], fdparent_in[2];
     int fd, nread;
     char buf[BUFLEN];

     //pipes erzeugen
     if (pipe(fdparent_in) < 0 || pipe(fdparent_out) < 0) {
          perror("Fehler bei pipe");
          exit(1);
     }

     switch (fork()) {
          case -1: /* Fehler */
               perror("Fehler bei fork");
               exit(1);

     /* CHILD: richtet eine pipe zw. parent und unix utility sort ein*/
          case  0:
          // schliesst stdin, stdout
          //
          // dadurch werden die filedeskriptoren 0 und 1 frei
          //
          // beim nächsten dup()
          // werden diese freien filedeskriptoren vom System
          // wieder verwendet.
          //
```

```
        // Dadurch werden (s.u.), die mit dup() duplizierten
        //      fdparent_out[LIES] an filedeskriptor 0 und
        //      fdparent_in[SCHREIB]an filedeskriptor 1 gebunden
        //
        // dies ist ideal für den mit exec() aufgerufenen
        // sort-Befehl, da dieser
        // mit stdin und stdout arbeitet.
        //
        // D.h.
        // Die Ausgabe des PARENT wird an die stdin des Sort-Befehls
        // und
        // Die Eingabe des PARENT wrid an die stdout des Sort-Befehls
        // gebunden
        //
        // die standardkanäle des sort-prozesses sind mit den zuvor
        // duplizierten Kanälen verbunden. d.h.

        // ein read(0,buf,len) des sort-prozesses liest
        // tatsächlich vom
        //     fdparent_out[LIES]

        // ein write(1,buf,len) des sort-prozesses schreibt
        // tatsächlich nach
        //     fdparent_in[SCHREIB]
        //
        // nun kommuniziert d. Parent direkt mit d. externen Programm


        close(fdparent_out[SCHREIB]);
        close(fdparent_in[LIES]);

        //std-eingabe schliessen
        if (close(0) < 0) {
             perror("Fehler bei close");
             exit(1);
        }
        if (dup(fdparent_out[LIES]) != 0) {
             perror("Fehler bei dup");
             exit(1);
        }

        //std-ausgabe schliessen
        if (close(1) < 0) {
             perror("Fehler bei close");
             exit(1);
        }
        if (dup(fdparent_in[SCHREIB]) != 1) {
             perror("Fehler bei dup");
             exit(1);
        }

        //werden nicht mehr gebraucht, da sort den child überlagert
        close(fdparent_out[LIES]);
        close(fdparent_in[SCHREIB]);

        execlp("sort", "sort", NULL);
```

```
     /* PARENT: */
          default:
          close(fdparent_out[LIES]);
          close(fdparent_in[SCHREIB]);

          // FILE
          if ((fd = open(fname, O_RDONLY)) < 0) {
               perror("Fehler bei open");
               exit(1);
          }

          while ((nread = read(fd, buf, sizeof(buf))) != 0) {
               if (nread == -1) {
                    perror("Fehler bei read");
                    exit(1);
               }

               if (write(fdparent_out[SCHREIB], buf, nread) == -1) {
                    perror("Fehler bei write auf Pipe");
                    exit(1);
               }
          }//while

          close(fd);
          close(fdparent_out[SCHREIB]);


          // sortierte Daten lesen
          while((nread=read(fdparent_in[LIES], buf, sizeof(buf))) != 0){
               if (nread == -1) {
                    perror("Fehler bei read von Pipe");
                    exit(1);
               }

               if (write(1, buf, nread) == -1) {
                    perror("Fehler bei write");
                    exit(1);
               }
          }

          close(fdparent_in[LIES]);
     }//switch

     return 0;
}


int main(int argc, char* argv[]) {
     if (argc < 2) {
          fprintf(stderr, "Aufruf: %s file\n", argv[0]);
          exit(1);
     }

     sortieren(argv[1]);

     return 0;
```

```
}

// Wir lassen das Programm seinen eigenen Quelltext sortieren.
// gcc -o pipe2-execlp-sort.exe pipe2-execlp-sort.c;
// ./pipe2-execlp-sort.exe pipe2-execlp-sort.c
```

Wichtig ist im Kindprozeß das Duplizieren des Filedeskriptors. Mit Hilfe der dup()-Funktion bleibt bei einem exec-Aufruf das close-on-exec-Flag gelöscht. Wir verbinden praktisch mit der Funktion dup die Standardaus/eingabekanäle mit unseren Pipekanälen. Wichtig ist in einem solchen Fall, wenn wir unseren Kindprozeß überlagern die Reihenfolge: fork, close, dup, exec. Somit kommuniziert unser aktuell laufender Elternprozeß mit dem Programm sort.

1.8.2. Aufgabe: pipe-execlp-befehl.c

Schreiben Sie auf der Grundlage des obigen Programmes, ein Programm, das als Argument einen einfachen Unixbefehl (ls oder ps oder ...) erhält. Der Befehl soll vom PARENT an den CHILD geschickt werden und dort mit einer entsprechenden exec-Funktion ausgeführt werden. Die Ausgabe des Befehls soll direkt an den PARENT geschickt werden.

1.8.3. Aufgabe: einfacher-telnetd.c

Schreiben Sie das Programm t_fileserver.c derart um, dass vom Client-Programm einfache Unixbefehle geschickt werden. Diese sollen analog zur vorherigen Aufgabe mittels child-prozesse ausgeführt werden und das Ergebnis an den Client geschickt werden. Client-Programm t_fileclient.c .

1.9. Bidirektionale Kommunikation mit Komfort: popen()

Die Subroutine popen() führt das im Parameter (String) angegebene Kommando mit Hilfe von system() aus. Dabei wird zusätzlich eine Pipe angelegt, die von popen zurückgegeben wird.Auf diese pipe kann dann geschrieben (mode = "w") bzw. gelesen (mode = "r") werden. Die Subroutine pclose schließt den als Argument anzugebenden Stream wieder und liefert den Status der Shell als Resultat. Falls sich der Stream nicht schließen läßt, ist das Resultat gleich EOF.

Zum Lesen bzw. Schreiben des Streams dienen die alltäglichen Standard-I/O-Funktionen wie fgets() oder fputs().

popen, pclose - process I/O

1.9.1. Beispiel: popen-demo.c

```
// a.hofmann 2004
// popen-demo.c

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

#define LEN 500

char* getDIR(){
   FILE *pstream;
   char *ptr;
```

```
   pstream = popen("pwd", "r");

   ptr = (char *) malloc(LEN);

   if (fgets(ptr, LEN, pstream) == NULL) {
      pclose(pstream);
      return(NULL);
   }
   else {
      pclose(pstream);
      return(ptr);
   }
}

int main() {
   char* name;
   name = getDIR();

   if (name != NULL)
      printf("\ncurrent dir= %s\n", name);
   else
      printf("Fehler bei gcwdtst\n");

   free(name);
   return 0;
}

// gcc popen-demo.c -o popen-demo.exe; ./popen-demo.exe
```

1.9.2. Aufgabe: popen-starter.c

Schreiben Sie ein Programm, das aus der Kommandozeile als 1. Argument einen Dateinamen und als 2. Argument ein Shellkommando erhält. Führen Sie dieses Kommando mit popen() aus und schreiben Sie das Ergebnis in die Datei.

Beispiel:
./popen-starter.exe erg.txt "find /etc -type f | grep -i localhost"

Hinweis:
Da der Befehl spaces enthalten kann, muss er mit " " geklammert werden.

1.10. Zeitmessung-Pipes: Bytes/Sekunde

Fragenstellung:

1. Wieviele Bytes pro Sekunde können gelesen/geschrieben werden und
2. hängt dies von der Anzahl Bytes, die mit einem read() / write()-Aufruf geschrieben/gelesen werden, ab.

Um den Resourcen-Verbrauch von Prozessen zu ermitteln, verwendet man folg.:

1. **int wait3(int statusp, options, struct rusage *rusage)**
   man 2 wait3

liefert dem aufrufenden Prozess, wieviel an Ressourcen der terminierte Kind-Prozess verbraucht hat.

2. **int getrusage(int who, struct rusage *rusage)**
   man 2 getrusage
   liefert, wieviel Ressourcen ein Prozess bis zum Zeitpunkt des Aufrufs von getrusage() verbraucht hat, bzw. wieviel Ressourcen alle seine Kinder verbraucht haben.

Eine Variable vom Typ struct rusage enthält folgende Informationen:

```
struct    rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    int  ru_maxrss;          /* maximum resident set size */
    int  ru_ixrss;           /* currently 0 */
    int  ru_idrss;           /* integral resident set size */
    int  ru_isrss;           /* currently 0 */
    int  ru_minflt;          /* page faults not requiring physical I/O*/
    int  ru_majflt;          /* page faults requiring physical I/O */
    int  ru_nswap;           /* swaps */
    int  ru_inblock;         /* block input operations */
    int  ru_oublock;         /* block output operations */
    int  ru_msgsnd;          /* messages sent */
    int  ru_msgrcv;          /* messages received */
    int  ru_nsignals;        /* signals received */
    int  ru_nvcsw;           /* voluntary context switches */
    int  ru_nivcsw;          /* involuntary context switches */
};
```

Hier nun eine mögliche Verwendung:

```
#define PRINT_R(who)                        \
    printf("%s!user: %d.%.6d s\n", who,   \
              rusage.ru_utime.tv_sec,     \
              rusage.ru_utime.tv_usec );  \
    printf("%s!sys:  %d.%.6d s\n",  who,  \
              rusage.ru_stime.tv_sec,     \
              rusage.ru_stime.tv_usec );

void used_ressource() {
     struct rusage rusage;

     getrusage(1, &rusage); /* 1 == der eigene Prozess */
     PRINT_R("writer");

     wait3(NULL, WNOHANG | WUNTRACED, &rusage);
     PRINT_R("reader");
}
```

1.10.1. Aufgabe: pipe-zeitmessung.c

Erstellen Sie unter Verwendung von fork-demo.c das Programm pipe-zeitmessung.c, das den Datendurchsatz beim Lesen/Schreiben einer Pipe messen soll.

Ausgabe:

Zeitmessung: PIPE_BUF= 4096
Number of Bytes: 409600000

PARENT as writer!user: 0.016001 s
PARENT as writer!sys: 2.180136 s

CHILD as reader!user: 0.000000 s
CHILD as reader!sys: 2.172135 s

Wie man sieht ist das Schreiben etwas aufwendiger.

Hinweise:

PIPE_BUF ist die Standardgrösse für den Pipe-Buffer. (s. limits.h)

```
void childs_code(){
     int i;

     close(fd[1]);

     for (i = 0 ; i < MAX; i++){
          if ( read(fd[0], buf, PIPE_BUF ) < 0 ) {
               perror("child!read"); exit(3);
          }
     }
     exit(0);
}


void parents_code(){
     int i;

     close(fd[0]);

     for (i = 0 ; i < MAX; i++){
          if ( write(fd[1], buf, PIPE_BUF ) < 0 ) {
               perror("parent!write"); exit(4);
          }
     }

    used_ressource();
}
```

1.11. Named Pipes bei nicht verwandten Prozessen: mkfifo

Nicht miteinander verwandte Prozesse können durch gewöhnliche Pipes nicht kommunizieren. Bei diesen werden zB. Named Pipes (FIFOs) verwendet.

Eine FIFO ist letztlich eine Pipe, die aber über einen Namen im Dateisystem zu erreichen ist.

Eine FIFO wird mittels des Systemaufrufs (man 3 mkfifo)

```
NAME
       mkfifo - make a FIFO special file (a named pipe)

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>

       int mkfifo(const char *pathname, mode_t mode);
```

angelegt. Die FIFOs werden im Dateisystem abgelegt und müssen dort wieder entfernt werden.

Wurde eine FIFO erzeugt, kann diese mit den Standard-I/O-Funktionen **open(),close(), read(), write() und unlink()** benutzt werden.

1.11.1. Beachte im Umgang mit FIFOs

☑ Mehrere Schreiber und 1 Reader: (vgl. Druckerspooler)
Haben alle Schreiber einer FIFO ihre Verbindung abgebaut und wurden alle Daten aus der FIFO gelesen, liefert der nächste read(2)-Aufruf 0 Bytes gelesen, ohne zu blockieren.

Bei der Verwendung von FIFOs ist es üblich, dass mehrere Schreiber in eine FIFO schreiben. Damit die Daten der einzelnen Schreiboperationen nicht vermischt werden, müssen atomare Schreiboperationen verwendet werden. Also müssen bei jeder Schreiboperation weniger als PIPE_BUF Bytes geschrieben werden.

Wenn man eine FIFO zum Schreiben mit close oder fclose schließt, bedeutet dies für die FIFO zum Lesen ein EOF.

Hier nun einige Programmfragmente:

FIFO erzeugen:

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

int main(){

     unlink("/tmp/myFIFO"); // if already here

     /* create FIFO*/
     if((mkfifo("/tmp/myFIFO", I_SRUSR|I_SWUSR)) == -1){
          fprintf(stderr, "Error creating /tmp/myFIFO.......\n");
          exit(0);
     }
     return 0;
}
```

1.11.2. Beispiel: fifo-demo-echo.c – 1 Reader und N Schreiber

Terminal 1: ./fifo-demo-echo.exe

Terminal 2: cat > /tmp/myFIFO-echo

Terminal 3: cat > /tmp/myFIFO-echo

```
// a.hofmann, 2012
// fifo-demo-echo.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>

#define FIFO_NAME "/tmp/myFIFO-echo"

int main(){
     int fd, ret;
     char c;

     ret=mkfifo(FIFO_NAME, S_IRUSR | S_IWUSR);
     if (ret==-1){ perror(FIFO_NAME); exit(0); }

     fd=open(FIFO_NAME,O_RDONLY);
     if (fd<0 ) { perror(FIFO_NAME); exit(0); }

     while (read(fd,&c,1)==1)
          fputc(c, stdout);

     close(fd);

     unlink(FIFO_NAME);
     return 0;
 }
 // gcc fifo-demo-echo.c -o fifo-demo-echo.exe
 // ./fifo-demo-echo.exe
 //
 // in 2 anderen Terminals
 // cat > /tmp/myFIFO-echo
```

1.11.3. Beispiel: fifo-zeitmessung.c

Wir wollen nun wie beim pipe-zeitmessung.c (s. o.) das Zeitverhalten von FIFOS ermitteln.

Ausgabe: Kommunikation via FIFO

Zeitmessung: PIPE_BUF= 4096
Number of Bytes: 409600000

PARENT as writer!user: 0.016001 s
PARENT as writer!sys: 2.284142 s

CHILD as reader!user: 0.004000 s
CHILD as reader!sys: 2.296143 s

Hier nochmal die pipe()-Kommunikations zum Vergleich.

Ausgabe: Kommunikation via pipe()

Zeitmessung: PIPE_BUF= 4096
Number of Bytes: 409600000

PARENT as writer!user: 0.016001 s
PARENT as writer!sys: 2.180136 s

CHILD as reader!user: 0.000000 s
CHILD as reader!sys: 2.172135 s

Man sieht, dass kaum ein Unterschied ist.

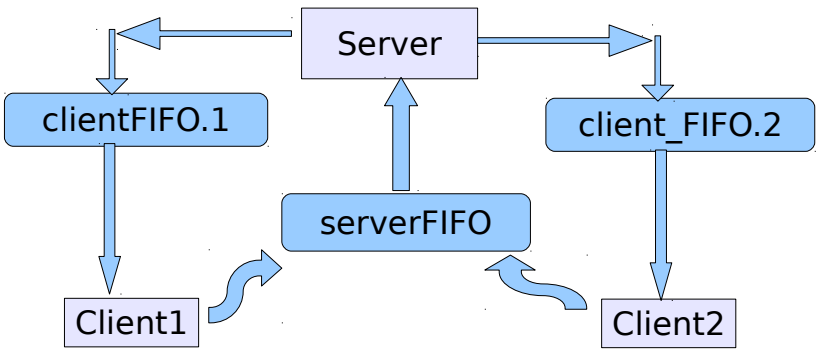
Frage:

Wie sieht es aus, wenn geringere Datenmengen (< 4096) verschickt werden?

1.11.4. Beispiel: fifo-client-server

Mit FIFOs lassen sich zB. mittels Shellprogrammierung sehr leicht Client/Server-Anwendungen implementieren. Als Beispiel implementieren wir einen Server, der ein Telefonbuch verwaltet. Die zugehörigen Clienten, von denen es viele geben kann, können vom Server Telefondaten abrufen und ihn auffordern, neue Telefondaten in die Datenbank einzutragen. Der Kommunikationsfluss:

Wenn der Server den verschiedenen Clients Antworten schicken soll, dann muss für jeden Client ein eigener FIFO angelegt werden.



Jeder Client nimmt über eine bekannte FIFO Kontakt mit dem Server auf, die Antwort liefert der Server über einen nur ihm und dem Klient bekannt FIFO.

Datei: fifo-client.sh

```
#!/bin/sh
# fifo-client.sh
# $$ ... liefert die PID
#
CLIENT_FIFO=/tmp/clientFIFO.$$
SERVER_FIFO=/tmp/serverFIFO

#
mkfifo $CLIENT_FIFO
```

```
# SIGTERM SIGINT abfangen und CLIENT_FIFO löschen und ende
trap "trap '' 0 1 2 3 15;
     rm -f $CLIENT_FIFO; exit 0;" 0 1 2 3 15

# infos
echo " Kommandos:"
echo "          add   Name  Number"
echo "          get   Namer"
echo "          get   Number"

# Anfrage lesen und an Server schicken
while read question
do
     echo $question $CLIENT_FIFO > $SERVER_FIFO
     read answer < $CLIENT_FIFO
     echo $answer > /dev/tty
done
```

Der Client

1. legt eine nur ihm bekannte CLIENT_FIFO an und
2. liest von der Standardeingabe die Anfragen.
3. Die Anfrage (question) und der Name der CLIENT_FIFO -welche der Server für die Antwort verwendet- werden über die SERVER_FIFO an der Server geschickt.
4. Die Antworten werden aus der CLIENT_FIFO gelesen und aufs Terminal ausgegeben.
5. Falls der Client terminiert, wird die CLIENT_FIFO durch die trap aus dem Dateisystem entfernt.

Datei: fifo-server.sh

```
#!/bin/sh
# fifo-server.sh
#    Telefonserver
#               $1    $2         $3         $4
#    Kommandos:
#               add   Name  Number     FIFO
#               get Name    FIFO
#               get Number  FIFO

SERVER_FIFO=/tmp/serverFIFO
DATA=fifo-server.db         # Name Number

mkfifo $SERVER_FIFO
#
trap 'trap "" 0 1 2 3 15;
     rm -f $SERVER_FIFO; exit ' 0 1 2 3 15

while true
do
     (  while read question < $SERVER_FIFO
          do
               set $question
               case  $1
                    in a*) echo "$2 $3" >>  $DATA
                              sort -u -o $DATA $DATA
                              echo added > $4
```

```
                    ;; g*) found="`grep "$2" $DATA`"
                                echo "$found" > $3
                    ;; *)  echo "What?" > /dev/console
                esac
            done
        )
done
```

1.12. Synchronisation mit Lockfiles

Wir betrachten in diesem Abschnitt eine einfache und altbewährte Methode zur Erreichung des **gegenseitigen Ausschlusses bei kritischen Abschnitten**.

Mit Hilfe der Eigenschaften des Systemaufrufs open() implementieren sogenannte Lock-Files eine Art von **Semaphoren**.

☑ P-Operation: in den kritischen Abschnitt eintreten
Als Simulation der P-Operation wird wiederholt versucht, das Lock-File mit open() zu kreieren.

☑ Die V-Operation: den kritischen Abschnitt verlassen
wird auf das Löschen des Lock-Files mit unlink zurückgeführt.

☑ Die beiden Operationen P und V heißen in unserem Zusammenhang lock und unlock.

Die wesentliche Idee bei dieser Vorgehensweise ist die gemeinsame Verwendung der Zugriffs-Flags O_CREAT und O_EXCL bei open (man open). Wenn das File schon existiert, kann ein Prozess nicht erfolgreich damit sein. Unter der Voraussetzung, dass der Systemaufruf open atomar ist, also nicht unterbrochen werden kann, ist damit die Semaphor-Eigenschaft sichergestellt.

Wir wollen in der Folge zwei Funktionen zum Betreten und Verlassen des kritischen Abschnittes implementieren:

☑ **int lock(char* name);**

☑ **void unlock(char* name);**

Die Funktionen haben beide das Argument name, einen String-Pointer, der den Namen des Lock-Files im Directory /tmp angibt. Das Resultat von lock ist im Erfolgsfalls die Anzahl der Versuche, die zur Inbesitznahme des Lock-Files nötig waren, minimal 1. Im Fehlerfall ist das Resultat negativ, absolut gleich der Anzahl der Fehlversuche.

1.12.1. Beispiel: lock-demo.c

```
// A.Hofmann, 2012
// lock-demo.c
// implement critical section using lock-files

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#define LOCKDIR "/tmp/"
#define MAX_LOCKNAME 5+64+1
#define MAX_ATTEMPTS 10
#define SLPTIME (unsigned int)1


/* Lock-File in Besitz bringen */
int lock(char* name){
     int fd;
     int nAttempts;
     char full_lockname[MAX_LOCKNAME];

     // absolute path of the lock-file:
     // /tmp/..... erstellen
     strcpy(full_lockname, LOCKDIR);
     strncat(full_lockname, name, 64);

     // try to create lock-file
     nAttempts = 1;
     fd = open(full_lockname, O_WRONLY|O_CREAT|O_EXCL,0666);
     while (fd==-1 && errno==EEXIST){
          // lockfile exists-> so try again

          if (nAttempts >= MAX_ATTEMPTS)
                return -nAttempts;


          sleep(SLPTIME);
          nAttempts++;
          fd = open(full_lockname, O_WRONLY|O_CREAT|O_EXCL,0666);
     }

     return nAttempts;
}

/* Lock-File freigeben */
void unlock(char* name) {
     char full_lockname[MAX_LOCKNAME];

     // absolute path of the lock-file:
     // /tmp/..... erstellen
     strcpy(full_lockname, LOCKDIR);
     strncat(full_lockname, name, 64);

     if (unlink(full_lockname) == -1) {
          fprintf(stderr, "Error: unlock().\n");
          exit(1);
     }
}


int main(int argc, char* argv[]){
     char lockname[128];
     int  pid, i, l, nLoops;
     int ret_lock;
```

```
     if (argc != 4) {
          fprintf(stderr, "usage: %s processname lock-filename
nLoops\n", argv[0]);
          exit(1);
     }

     strcpy(lockname, argv[2]);
     nLoops = atoi(argv[3]);


     for (i=0; i<nLoops; i++) {
          // TRY LOCK
          printf("\n%s: TRY to ENTER critical section", argv[1]);
          printf(" (nLoop=%2i Lock-File=%s)", i, lockname);
          fflush(stdout);

          ret_lock = lock(lockname);
          if (ret_lock < 0) {
               fprintf(stderr,"%2i attempts failed to create Lock-File
%s\n",
                    ret_lock, lockname);
               exit(1);
          }

          // critical section
          printf("\n%s: IN critical section", argv[1]);
          printf(" (nLoop=%2i locking-attempts=%2i)",  i, ret_lock);
          fflush(stdout);

          //UNLOCK
          unlock(lockname);
          printf("\n%s: LEFT critical section", argv[1]);
          printf(" (nLoop=%2i Lock-File=%s)", i, lockname);
          fflush(stdout);

          sleep(2);
     }/*for*/

     printf("\n\n");
     return 0;
}
```

Wir compilieren locktest.c und starten zwei Versionen zum konkurrenten Ablauf. Dies sieht dann so aus:

gcc lock-demo.c -o lock-demo.exe
./lockt-demo.exe "AAA" LOCK 10 & ./lock-demo.exe "……ZZZ" LOCK 10 &

Typische Beispiele der Anwendung von Lockfiles innerhalb UNIX sind zu finden bei relativen langsamen Vorgängen, wie der Synchronisierung von Druckjobs.

1.13. Aufgabe: Paralleles Sortieren (parallelSort-fork-pipe.c)

Siehe Ordner: AB-parallelSort-fork-pipe

Eine große Datei ("daten.txt") mit positiven, ASCII-codierten Ganzzahlen ist zu sortieren. Dies kann unter Unix/Linux mit dem Kommando

```
sort -g daten.txt
```

geschehen. Bei z.B. 500000 Zahlen dauert dies schon relativ lange. Es soll nun versucht werden, die Sortierzeit durch eine Aufteilung des Sortiervorganges auf 2 parallele Prozesse zu reduzieren.

Das Grundprinzip lautet wie folgt:

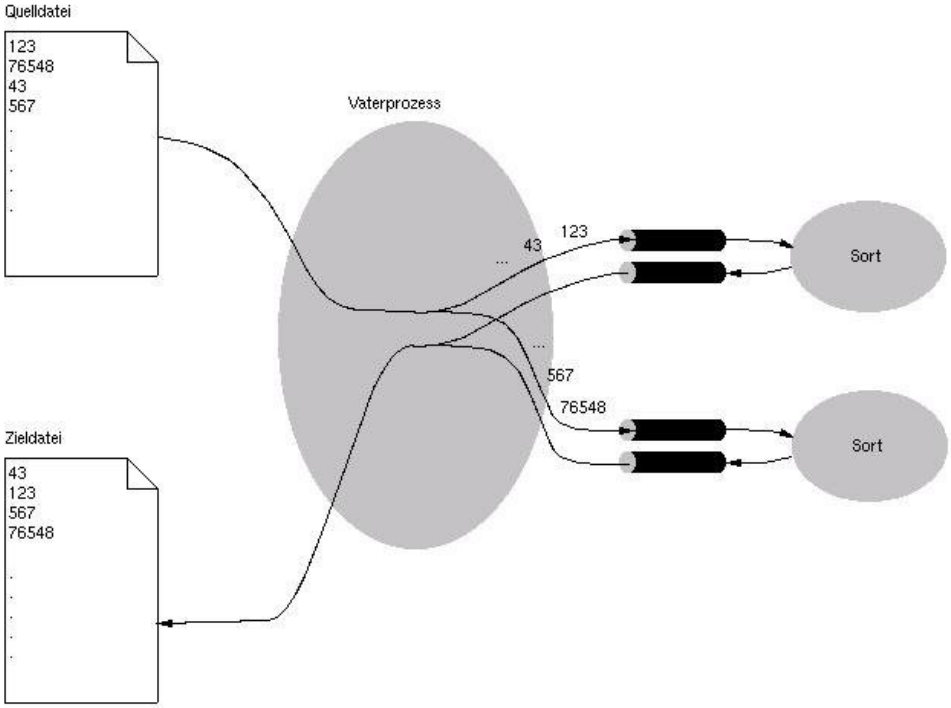
1. jeweils eine Hälfte der Zahlen wird zu den beiden Sortierprozessen geschickt
2. beide Prozesse sortieren parallel
3. das Sortierergebnis wird zurückgesendet
4. durch "Mischen" der beiden Zahlenfolgen wird das Gesamtergebnis erzeugt

Daraus resultiert die folgende Implementation (Beispiel):

Ein C-Programm "mysort" kreiert 4 "Pipes" zur bidirektionalen Kommunikation mit den beiden Sortierprozessen. Es erzeugt danach 2 Kindprozesse mit "fork", die ihrerseits über "execlp" das Programm "sort" starten - und zwar so, dass die unsortierten Zahlen über die Standardeingabe gelesen und die sortierten über die Standardausgabe geschrieben werden.

Der Vaterprozess liest nun die Datei "daten.txt" und sendet die Zahlen im Wechsel zu seinen beiden Kindprozessen.
Danach wartet er auf die Ergebnisdaten. Dabei liest er jeweils eine Zahl von den beiden "Ergebnis"-Pipes und schreibt die jeweils kleinere auf den Bildschirm (dies wird als "Mischen" bezeichnet).

Die folgende Grafik erläutert das Prinzip:



Hinweis: Mischen (Merging)
kann zunächst für 2 Felder demonstriert werden.Eine mögliche Implementation wäre:

```
 // merge(int a[], int b[], int c[], int n, int m)
 // mischt zwei sortierte Felder a[], der Dimension n
 // und b[] der Dimension m in ein Feld c[], der Dimension m+n
 //

void merge(int a[], int b[], int c[], int n, int m){
    int i, j, k;
    i = j = 0;
      for( k=0; k<m+n; k++ ) {
                /* kleineres Element a[i] oder b[j] finden */
           if( i<n && j<m ) {
                  if (a[i] < b[j])
                            c[k]=  a[i++];
                  else
                            c[k]=  b[j++];
          }
          else if( i<n ) {
                        c[k] = a[i++];
          }
          else if( j<m ) {
                        c[k] = b[j++];
          }
       }
```

```
       return;
}
```

Hinweis: Zufallszahlen

```
// a.hofmann
// generiert eine datei mit zufallszahlen
// aufruf: ./create-daten.exe filename anzahl
//         ./create-daten.exe daten.txt 500000
// gcc create-daten.c -ocreate-daten.exe

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

int main(int argc, char * argv[]){
    int anzahl;
    int i;
    FILE* fp;

    if (argc != 3){
        printf("\nAufruf: ./create-daten.exe daten.txt 500000\n");
        exit(1);
    }

    if (isdigit(argv[2][0]))
        anzahl= atoi(argv[2]);
    else {
        printf("\nAufruf: ./create-daten.exe daten.txt 500000\n");
        exit(1);
    }

    fp= fopen(argv[1], "w");
    if (fp==NULL){
        perror(argv[1]);
        exit(1);
    }

    printf("\n....generating file: %s with %d integers ....\n",argv[1], anzahl);

    for (i=0; i< anzahl; i++)
        fprintf(fp,"%d\n", rand()%10000);

    close(fp);
}
```

Hinweis: Laufzeitmessung: (www.pronix.de)

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

int main(){
 long i;
 float zeit;

 clock_t start, ende;

 /*start bekommt die aktuelle CPU-Zeit*/
 start = clock();

 /*Hier sollte der ausführbare Code stehen für die
   Laufzeitmessung*/
```

```
 /*Wir verwenden einfach ein Schleife*/
 for(i=0; i<200000000; i++)
           ;

 /*stop bekommt die aktuelle CPU-Zeit*/
 ende = clock();

 /*Ergebniss der Laufzeitmessung in Sekunden*/

 zeit = (float)(ende-start) / (float)CLOCKS_PER_SEC;

 printf("Die Laufzeitmessung ergab %.2f Sekunden\n",zeit);
 return 0;

}
```

1.14. Weitere Themen

siehe auch
unix-guru (openbook)

1. shared memory und semaphore
2. pthreads