Inhaltsverzeichnis

1. POSIX–IPC: Inter Process Communication (4)

1.1. Basics – Shell-Pipes,Named-Pipes, low-level IO

- **Pipes** sind **unidirektionale** Datenkanäle zwischen zwei verwandten Prozessen (Parent und Child).
- **Pipes** nutzen das **FIFO**-Prinzip.
  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).
- Pipes können im **RAM-Speicher** oder als **Dateien (Named-Pipes)** realisiert sein.
- Lebensdauer in der Regel solange beide Prozesse existieren.

1.1.1. Beispiel: shell-pipes

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

| … bedeutet:

- stdout des Vorgänger-Prozesses wird nach stdin des Nachfolger-Prozesses umgeleitet.

Weitere Beispiele:

```
curl -s -L http://www.zamg.ac.at/ogd | grep Salzburg | awk -F";" '{print $2 "
hat " $6 " celsius"}'
```

https://www.howtogeek.com/438882/how-to-use-pipes-on-linux/

1.1.2. Beispiel: Named-Pipes

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
```

1.1.3. open,read,write,close - low-level 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 2 open

☑ man read write close

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)
```

1.1.4. Beispiel: demo_open.c

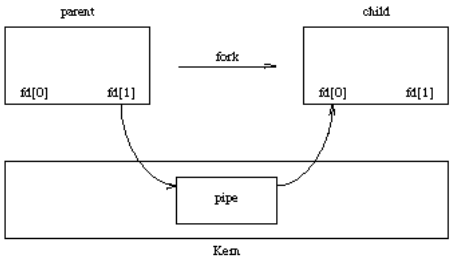
```
#include <stdlib.h>
#include <stdio.h>

#include <unistd.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);
      }

      printf("fd=%i\n", fd);


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

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

      close(fd);

      return 0;
}
```

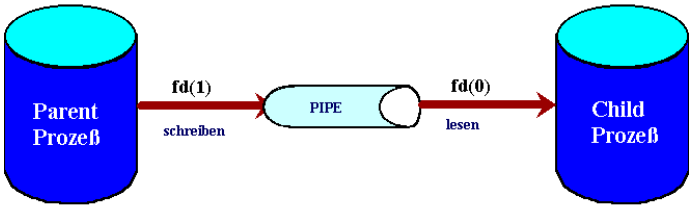
1.2. pipe() - IPC zwischen Parent und Child

- Die C-Funktion pipe() liefert 2 Kommunikationskanäle (=Filedeskriptoren).
- Einer zum Lesen und einer zum Schreiben.
- Eine Pipe wird mit dem Systemaufruf **int fd[2]; pipe(fd)** 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: verwende Makros zur besseren Lesbarkeit

```
#define child_in fd[0]
#define parent_out fd[1]

…Parent
close(child_in);
write (parent_out,“hallo“, 6); // inkl. EOS

…Child
close(parent_out
int anzahl= read (child_in, buf, sizeof(buf));
```

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.

1.2.1. Beispiel: demo_pipe_fork.c – IPC

Das folgende Beispiel zeigt:

1. Das Erstellen einer uni-direktionalen Pipe
2. Kommunikation zwischen Parent- und Child Prozess
3. parent und child verwenden die gleiche MSG_SIZE
4. #define Makros vereinfachen die Verwendung der Pipe fd.

```
// demo_pipe_fork.c
```

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

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

#define pipe_in fd[0]
#define pipe_out fd[1]

#define MSG_SIZE 128

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

int main()
{
   char buf[MSG_SIZE];
   int fd[2]; // pipe
   pid_t pid;

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

   // FORK
   if ((pid = fork()) < 0)
   {
      perror("Fehler bei fork");
      exit(1);
   }

   // PARENT ====================
   //  1. fd[0] schliessen und
   //  2. in fd[1] schreiben
   else if (pid > 0)
   {

      close(pipe_in);

      write(pipe_out, mes1, MSG_SIZE);
      write(pipe_out, mes2, MSG_SIZE);
      write(pipe_out, mes3, MSG_SIZE);

      close(pipe_out);

      wait(NULL);
   }

   //CHILD ==================================
   //  1. fd[1] schliessen und
   //  2. von fd[0] lesen
   else if (pid == 0)
   {

      int len;
```

```
      close(pipe_out);

      len = read(pipe_in, buf, MSG_SIZE);
      printf("%i Bytes gelesen: <%s>\n", len, buf);

      len = read(pipe_in, buf, MSG_SIZE);
      printf("%i Bytes gelesen: <%s>\n", len, buf);

      len = read(pipe_in, buf, MSG_SIZE);
      printf("%i Bytes gelesen: <%s>\n", len, buf);

      close(pipe_in);

      exit(EXIT_SUCCESS);
   }

   return 0;
}

// gcc demo_pipe_fork.c -o demo_pipe_fork.exe; ./demo_pipe_fork.exe
/* output:
128 Bytes gelesen: <Hallo, Welt Nr.1>
128 Bytes gelesen: <Hallo, Welt Nr.2>
128 Bytes gelesen: <Hallo, Welt Nr.3>
*/
```

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**.

1.2.2. fdopen – 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");
```

```
while (fgets(buf, sizeof(buf), reading) ){
   // …..
}
```

Geben sie Ihrem Dateizeiger einfach einen aussagekräftigen Namen um Verwechslungen auszuschliessen.

Sehen wir uns dazu wieder ein Beispiel an.....

1.2.3. Beispiel: demo_pipe_fdopen.c – shell-commands

Aufgabenbeschreibung:
☑ PARENT:
- ☐ liest einen Befehl von stdin
- ☐ schreibt cden Befehl zum CHILD (um diesen dort zu protokollieren)
- ☐ führt den Befehl aus (s. system())

☑ CHILD:
- ☐ liest vom PARENT den Befehl
- ☐ schreibt den Befehl in eine Datei. (zB. Default.msh)

Folgende Streamnamen werden verwendet:
stdin
parent_out
child_in
newfile_out

```
/*
 * Demo: pipe, fork, fdopen
 *
 * Parent
 * - liest Shell-Befehle von tastatur
 * - sendet diese zum Logging an den Child-Prozess
 * - führt den Befehl mit system() aus
 * - ende mit exit als Eingabe
 *
 * PARENT:
 *    fgets(stdin)
 *    fputs(pipe_out)
 * CHILD:
 *    fgets(pipe_in)
 *    fputs(newfile_out)
 *
 * gcc demo_pipe_fdopen.c -o demo_pipe_fdopen.exe
 * ./demo_pipe_fdopen.exe default.msh
 */

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

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

#define USAGE printf("usage : %s default.msh\n", argv[0]);
#define BUF_SIZE 4096
```

```
int main(int argc, char *argv[])
{
	int fd[2], i, n;
	pid_t pid;
	char buf[BUF_SIZE];

	FILE *child_in, *parent_out, *newfile_out;

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

	// PIPE
	if (pipe(fd) < 0)
	{
		perror("pipe()");
		exit(0);
	}

	// FORK
	if ((pid = fork()) < 0)
	{
		perror("pipe : ");
		exit(0);
	}

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

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

		// user eingabe und senden an client (zum Logging)
		do
		{
			fputs("\nBitte einen Befehl (or exit) > ", stdout);
			fgets(buf, BUF_SIZE, stdin);
			//buf[strlen(buf) - 1] = '\0'; // del EOL

			// zum Child
			fputs(buf, parent_out);

			// execute command
			system(buf);
		} while (strncmp(buf, "exit" , 4) !=0);

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

```
            int line = 0;

            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);
            }

            do
            {
                  // read command from parent
                  fgets(buf, BUF_SIZE, child_in);

                  // log command to file
                  line++;
                  fprintf(newfile_out, "%6d\t%s", line, buf);
            } while (strncmp(buf, "exit" , 4) != 0);

            fclose(newfile_out);
            fclose(child_in);

            exit(EXIT_SUCCESS);
      }
      return 0;
}
```

1.2.4. 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

1.2.5. Aufgabe: pipe_toupper.c (mab)

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

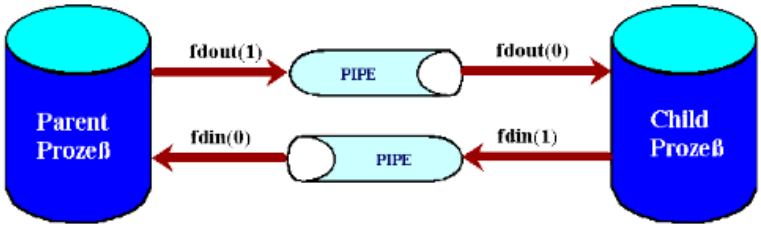
- wandelt das Gelesene in Grossbuchstaben um (toupper()) und
- schreibt den Text auf den Bildschirm

```
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

1.3. 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.

Tipps:

```
2 pipes verwenden:
  int fdout[2];
  int fdin[2];

use macros for better understanding
  #define child_in fdout[0]
  #define child_out fdin[1]

  #define parent_in fdin[0]
  #define parent_out fdout[1]


  PARENT:      ---> fdout[1] --- fdout[0] --->        CHILD:
  =======      <--- fdin[0]  --- fdin[1] <---         =====


  write (parent_out, ...)       ---->          read (child_in, ...)
  read (parent_in, ...)         <----          write (child_out, ...)
```

1.3.1. Beispiel: demo_pipe_dup_exec_sort.c

Verwendung von: fork,close,dup,exec, sort

Im folgenden Programm wird die **bidirektionale** Kommunikation zwischen Prozessen getestet.

- Der Child-Prozess wird mit execlp 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 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

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

```
// demo_pipe_dup_exec_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 MSG_SIZE 512
/*
2 pipes verwenden:

  int fdout[2];
  int fdin[2];

use macros for better understanding
*/
  #define child_in fdout[0]
  #define child_out fdin[1]

  #define parent_in fdin[0]
  #define parent_out fdout[1]

/*
  PARENT:      ---> fdout[1] --- fdout[0] --->        CHILD:
  =======      <--- fdin[0]  --- fdin[1] <---         =====


  write (parent_out, ...)       ---->          read (child_in, ...)
  read (parent_in, ...)         <----          write (child_out, …)
*/


int sortieren(char* fname){
    int fdout[2];
```

```
    int fdin[2];

     int fd, nread;
     char buf[MSG_SIZE];

     // PIPES
     if (pipe(fdout) < 0 || pipe(fdin) < 0) {
           perror("Fehler bei pipe");
           exit(1);
     }

   // FORK
     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
           //      child_in an filedeskriptor 0 und
           //      child_out an filedeskriptor 1 gebunden
           //
           // dies ist ideal für den mit execlp() 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 wird 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
           //     parent2child[0]

           // ein write(1,buf,len) des sort-prozesses schreibt
           // tatsächlich nach
           //     child2parent[1]
           //
           // nun kommuniziert d. Parent direkt mit d. externen Programm sort



           //stdin schliessen
           close(0);
           dup(child_in);

           //stdout schliessen
           close(1);
```

```
            dup(child_out);

            //werden nicht mehr gebraucht, da sort den child überlagert
            close(parent_out);
            close(parent_in);
            close(child_in);
            close(child_out);

            execlp("/usr/bin/sort", "sort", NULL);


      // PARENT:
            default:
        // werden hier nicht gebraucht
            close(child_in);
            close(child_out);

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

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

             // write to child/sort
                  if (write(parent_out, buf, nread) == -1) {
                        perror("Fehler bei write auf Pipe");
                        exit(1);
                  }
            }//while

            close(fd);
            close(parent_out);


            // sortierte Daten lesen
            while((nread=read(parent_in, 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(parent_in);
      }//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]);

    // auch das geht
    // sortieren("/etc/passwd");

    return 0;
}

// Wir lassen das Programm seinen eigenen Quelltext sortieren.
// gcc -o demo_pipe_dup_exec_sort.exe demo_pipe_dup_exec_sort.c
// ./demo_pipe_dup_exec_sort.exe demo_pipe_dup_exec_sort.c
```

1.4. Projekt: parallel_sort.c - Paralleles Sortieren

Siehe Ordner: parallel_sort

Eine große Datei ("daten.txt") mit positiven 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.

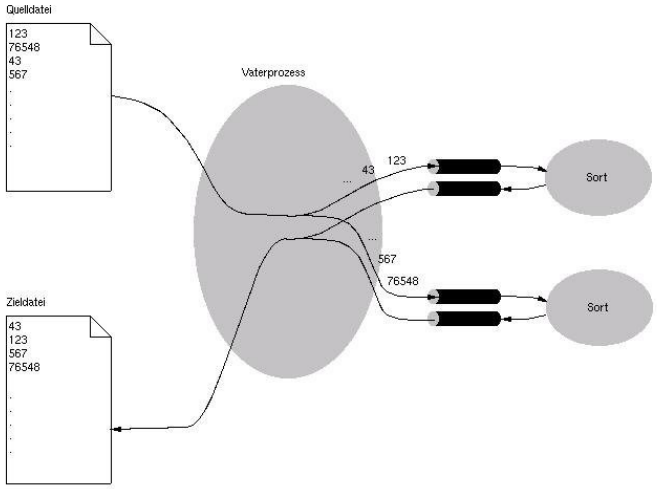
Das Grundprinzip lautet wie folgt:

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 sortierten Zahlenfolgen wird das Gesamtergebnis erzeugt

Daraus resultiert die folgende Implementation (Beispiel):

- Ein C-Programm "parallel_sort" 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 von 2 sortierent Arrays)
// 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

```
// generiert eine datei mit zufallszahlen
// gcc create_daten.c -ocreate_daten.exe
// ./create_daten.exe daten.txt 500000
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <sys/unistd.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 %i integers ...\n",argv[1], anzahl);

    for (i=0; i< anzahl; i++)
        fprintf(fp,"%i\n", rand()%100000);

    fclose(fp);
    return 0;
}
```

Hinweis: Laufzeitmessung: (www.pronix.de)

```
// laufzeit.c
#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 der ausführbare Code: Laufzeitmessung*/
 /*Wir verwenden einfach ein Schleife*/
 for(i=0; i<200000000; i++)
          ;

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

 /*Ergebnis der Laufzeitmessung in Sekunden*/

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

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

 return 0;
}
```