Kapitel 7: Standard-I/O-Funktionen

7: Standard-I/O-Funktionen

- Standard-I/O-Funktionen, definiert in `<stdio.h>`, erlauben das `Öffnen`, `Lesen`, `Schreiben` und `Schließen` von Dateien.
- Die STDIO-Funktionen sind Teil des POSIX.1-Standards.
- **Frage:** Zu welcher Bibliothek gehört die Header-Datei `<stdio.h>`?

Datenströme

- Datenstrom: Verknüpfung von Dateideskriptor mit Lese- und Schreibpuffer.
- Die Standard-I/O-Funktionen benutzen Datenströme (streams).
- In C-Repräsentiert der Datentyp `FILE` einen Datenstrom.
- Die Verwendung von Datenströme bietet mehr Komfort als die Nutzung von elementarem I/O-Funktionen welche direkt auf einem Dateideskriptor operieren.

Der Datentyp FILE

Bei dem Datentyp **FILE** handelt es sich um ein `struct` das von der Standard-I/O-Funktion **fopen()** zurückgegeben wird.

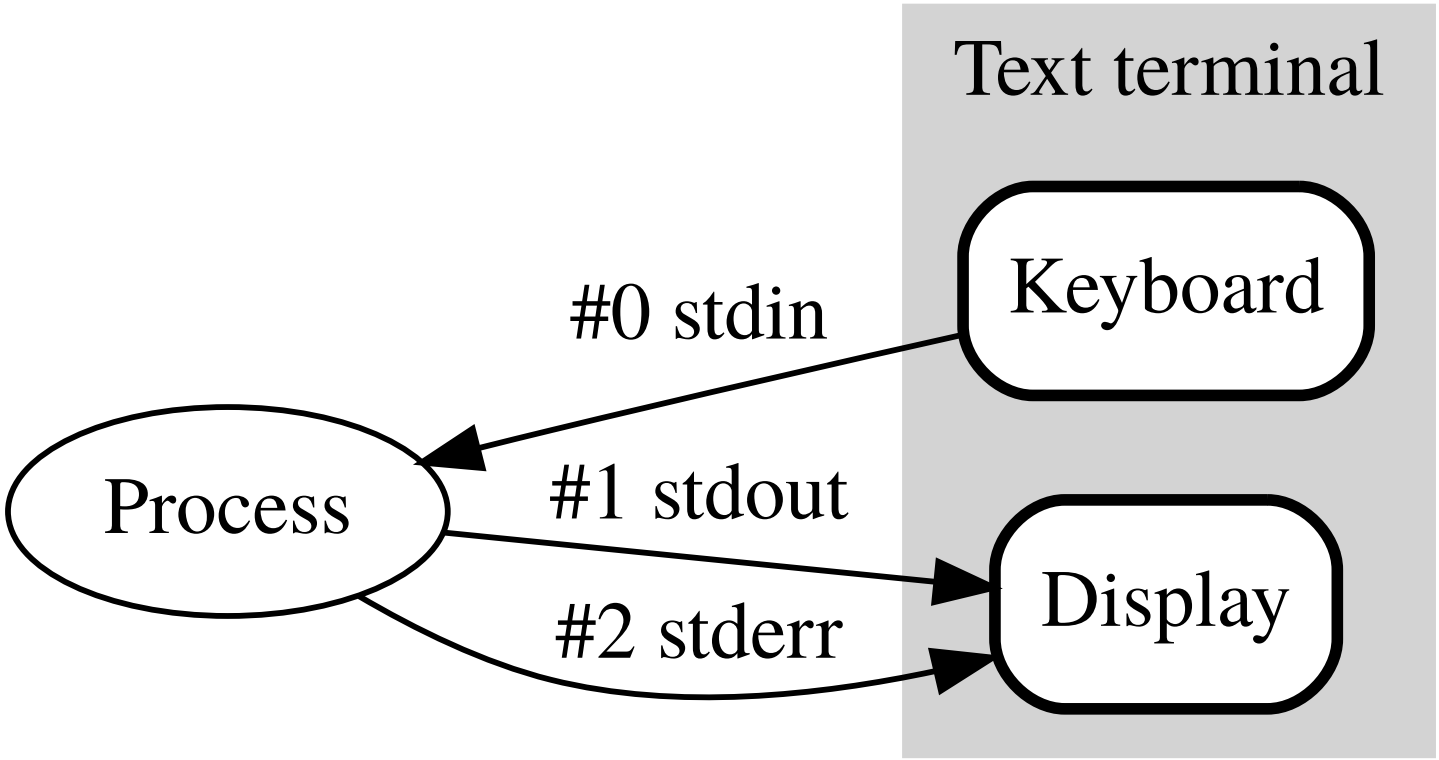
```
1  typedef struct _IO_FILE FILE;
2  ...
3  struct _IO_FILE {
4    char* _IO_buf_base;   /* Anfangadresse des Puffers */
5    __off64_t _offset;    /* Datei-Offset */
6    int _fileno;         /* Filedeskriptor */
7    ...
8  };
```

- Wissen über die Interna der **FILE**-Struktur wird meist nicht benötigt.
- Der von **fopen()** erhaltene **FILE**-Zeiger wird als Argument bei den entsprechenden IO-Funktionen übergeben

Standardstreams

- Prozesse haben durch die Dateien `/dev/stdin`, `/dev/stdout` und `/dev/stderr` Zugriff auf Keyboard und Terminal.
- Beim Starten eines Prozesses werden diese immer geöffnet.
- Zugriff erhält die Entwicklerin durch in `stdio.h` vordefinierte `FILE`-Pointer.
  - Standardeingabe: `stdin`
  - Standardausgabe: `stdout`
  - Standardfehlerausgabe: `stderr`

Standardstreams: Illustration



By Danielpr85 based on Graphviz source of TuukkaH - Own work, Public Domain

7.1: Öffnen und Schließen von Dateien

Wir betrachten die folgenden Standard-IO-Funktionen zum Öffnen und Schließen von Dateien:

- `fopen()`: Öffnet eine Datei und erzeugt einen damit verbundenen Datenstrom.
- `fdopen()`: Generiert aus einem existierenden Dateideskriptor ein `FILE`-Objekt inklusive Datenstrom.
- `freopen()`: Verbindet einen existierenden Datenstrom mit einem neuen Datenstrom.
- `fclose()`: Leert den Datenstrom und schließt den zugrundeliegenden Dateideskriptor.

Öffnen einer Datei

```
1 #include <stdio.h>
2
3 FILE *fopen(const char *pfad, const char *modus);
```

- **pfad**: Name der zu öffnenden Datei
- **fd**: Existierender Filedeskriptor
- **modus**: Zugriffsart

Beispiel: Öffnen der Datei `test.txt`

```
1 FILE *pfile = fopen("test.txt", "r");
```

Zugriffsarten

- **`"r"`**: Lesen (*read*)
- **`"w"`**: Schreiben (*write*).
  Achtung: Bereits vorhandene Dateiinhalte werden **gelöscht**.
- **`"a"`**: Anhängen (*append*).
- **`"r+"`**: Lesen und Schreiben.
- **`"w+"`**: Lesen und Schreiben.
  Achtung: Bereits vorhandene Dateiinhalte werden **gelöscht**.
- **`"a+"`**: Lesen und Schreiben. Datei wird am Dateiende geöffnet.

Auswirkungen und Einschränkungen von Zugriffsarten

Einschränkung bzw. Auswirkung	`r`	`w`	`a`	`r+`	`w+`	`a+`
Datei muss zuvor existieren	✓			✓		
Alter Dateiinhalt wird gelöscht		✓			✓	
Aus Datei kann gelesen werden	✓			✓	✓	✓
In Datei kann geschrieben werden		✓	✓	✓	✓	✓

Fehler

- Das Öffnen von Dateien kann fehlschlagen weil:
  - die Datei nicht vorhanden ist.
  - die Berechtigungen nicht ausreichen.
  - Modus oder Dateiname ungültig sind.
  - ...
- In Fehlerfall liefern `fopen()` und `fdopen()` den Wert `NULL` zurück und setzen die globale Integer-Variable `errno`
- `errno` ist im Headerfile `<errno.h>` definiert
- Die `errno`-Manpage enthält eine Liste an Fehlercodes.
  Beispiel: `EACCES`: Keine Berechtigung (POSIX.1)
- `errno == 0`: Es ist noch kein Fehler aufgetreten

Exkurs: Systemfehlermeldungen

```
1 #include <string.h>
2
3 char *strerror(int errnum);
```

- **`sys_errlist[]`** ist eine systemweite Fehlerliste mit allen Systemfehlermeldungen
- Die Methode gibt die **`sys_errlist[errnum]`** oder **`unknown error`** zurück
- Lokalisierung von Systemfehlermeldungen
  → **`strerror`**-Manpage

Ausgabe von Fehlermeldungen

```
1 #include <stdio.h>
2
3 void perror(const char *s);
```

- Gibt zunächst den String **`s`** auf **`stderr`** aus
- `s` sollte den Namen der Funktion enthalten, welche den Fehler ausgelöst hat, sowie deren Argumente
- Anschließend wird auf **`stderr`** nach einem Doppelpunkt **`strerror(errno)`** ausgegeben

Systemfehlermeldungen – Beispiel

```
1  #include <errno.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(int args, char *argv[]) {
6    FILE *pfile;
7
8    if(args>1) {
9      pfile = fopen(argv[1], "r");
10     if(!pfile) {
11       perror(argv[1]);
12     } else {
13       fclose(pfile);
14     }
15   }
16
17   puts(strerror(EAGAIN));
18   return errno;
19 }
```

Anmerkung: Die `usage()`-Methode fehlt.

Umlenken von Datenströmen

```
1 #include <stdio.h>
2
3 FILE *freopen(const char *pfad, const char *modus,
4                FILE *datenstrom);
```

- Rückgabe: `FILE`-Zeiger bei Erfolg, ansonsten `NULL`
- Funktionsweise:
  - Die Datei, die mit dem `datenstrom` verbunden ist, wird geschlossen
  - Die Datei `pfad` wird geöffnet und mit `datenstrom` verbunden
- Normalerweise wird `freopen()` benutzt um `stdin`, `stdout` oder `stderr` umzulenken

Umlenken von Datenströmen – Beispiel

```
1  #include <errno.h>
2  #include <stdio.h>
3
4  int main() {
5    char *file = "test.txt";
6    FILE *pfile = NULL;
7    int c;
8
9    while((c=getc(stdin)) != EOF) {
10     if(!(pfile = freopen(file, "a", stdout))) {
11       perror(file);
12       break;
13     }
14     putchar(c);
15   }
16
17   if(pfile) {
18     fclose(pfile);
19   }
20
21   return errno;
22 }
```

Schließen von Dateien

```
1 #include <stdio.h>
2
3 int fclose(FILE *stream);
```

- Bei Erfolg wird `0` zurückgegben, ansonsten **`EOF`**
- Bei invalider Eingabe (z. B. **`NULL`**) wird ein Segmentation Fault ausgelöst und das Programm *stürzt ab*
- Funktionsweise:
  - Die Standard-IO-Puffer werden geleert
  - Die Verbindung zum Datenstrom wird gekappt
  - Die Datei wird geschlossen
- Falls der Prozess normal endet, wird **`close()`** explizit für alle noch offenen Datenströme aufgerufen

7.2: Lesen und Schreiben von Dateien

Im Folgenden betrachten wir u. A. die folgenden Standard-IO-Funktionen zum Öffnen und Schließen von Dateien

- `getc()`, `fgetc()`: Lesen eines Zeichens
- `putc()`, `fputc()`: Schreiben eines Zeichens
- `fgets()`, `fputs()`: Lesen und Schreiben von Zeilen
- `fread()`, `fwrite()`: Lesen und Schreiben von Blöcken

EOF- und Fehler-Flags

```
1 #include <stdio.h>
2
3 int feof(FILE *stream);
4 int ferror(FILE *stream);
```

- Die FILE-Struktur verfügt über ein EOF- und ein Fehler-Flag
- Die meisten Funktionen geben bei einem Fehler oder dem Dateieinde den Code **`EOF`** zurück
- Wird das Dateiende erreicht, wird das EOF-Flag gesetzt; bei einem Fehler das Fehler-Flag
- Ist das jeweilige Flag gesetzt, wird ein Wert ungleich `0` zurückgegeben; ansonsten `0`
- Bei einem ungültigen Stream wird `-1` zurückgegeben
- Die Funktion **`int clearerr(FILE *stream)`** löscht das EOF- und das Fehler-Flag

EOF- und Fehler-Flags – Beispiel

```
1  #include <stdio.h>
2
3  int main() {
4    FILE *pfile = fopen("test.txt","w+");
5    int c = getc(pfile);
6    c = c;
7
8    printf("%d\n",feof(pfile));
9    clearerr(pfile);
10   printf("%d\n",feof(pfile));
11
12   fclose(pfile);
13   return 0;
14 }
```

Frage: Was ist die Ausgabe des Programms?

Lesen eines Zeichens

```
1 #include <stdio.h>
2
3 int getc(FILE *stream);
4 int getchar(void);
```

- Die **`getc()`**-Funktion liest das nächste Zeichen aus dem Datenstrom **`stream`** und gibt es zurück
- Im Fehlerfall wird **`EOF`** zurückgegeben
- **`int getchar()`** entspricht **`int getc(stdin)`**

Schreiben eines Zeichens

```
1 #include <stdio.h>
2
3 int putc(int c, FILE *stream);
4 int putchar(int c);
```

- Die **`putc()`**-Funktion schreibt das Zeichen **`c`** in den Datenstrom **`stream`** und gibt es wieder zurück
- Im Fehlerfall wird **`EOF`** zurückgegeben
- **`int putchar(c)`** entspricht **`int putc(c,stdout)`**

Beispiel: getc

```
1  #include<stdlib.h>
2  #include<stdio.h>
3
4  int main(int argc, char * argv[]) {
5    FILE *fp;
6    int i;
7    unsigned long long ctr, total=0;
8
9    if(argc < 2) {
10     fputs("Usage: bytecount <file1> <file2> ...\n",stderr);
11     return EXIT_FAILURE;
12   }
13
14   for(i=1;i < argc; i++){
15     if(!(fp = fopen(argv[i], "r"))) perror(argv[i]);
16     else {
17       ctr = 0;
18       while( fgetc(fp) != EOF) ctr+=1;
19       total += ctr;
20       fclose(fp);
21       printf("%s: %llu\n",argv[i], ctr);
22     }
23   }
24   printf("total: %llu\n",total);
25   return EXIT_SUCCESS;
26 }
```

Lesen von ganzen Zeilen

```
1 #include <stdio.h>
2
3 char *fgets(char *buf, int size, FILE *stream);
```

- Der Buffer **buf** wird mit maximal **n-1** Zeichen gefüllt, die aus dem Datenstrom **stream** gelesen werden
- Im Erfolgsfall wird **buf** zurückgegeben, ansonsten **NULL**
- Das Lesen endet vorzeitig bei einem Newline-Zeichen **\n** oder bei Erreichen des Dateiendes
- Am Ende des Lesevorgangs wird ein Stringterminator **\0** angefügt

Schreiben von ganzen Zeilen

```
1 #include <stdio.h>
2
3 int fputs(const char *s, FILE *stream);
4 int puts(const char *s);
```

- Der String **s** und **\n** wird in den Datenstrom **stream** geschrieben
- Im Fehlerfall wird **EOF** zurückgegeben; ansonsten eine nicht-negative Zahl
- **putchar(c)** entspricht **putc(c, stdout)**

Beispiel: `fgetc`

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <limits.h>
4
5  int main(int argc, char * argv[]) {
6    FILE *fp;
7    char buf[LINE_MAX+1];
8    int i;
9    unsigned long long ctr, total=0;
10
11   if(argc < 2) {
12     fputs("Usage: linecount <file1> <file2> ...\n",stderr);
13     return EXIT_FAILURE;
14   }
15
16   for(i=1;i < argc; i++){
17     if(!(fp = fopen(argv[i], "r"))) perror(argv[i]);
18     else {
19       ctr = 0;
20       while( fgets(buf,LINE_MAX,fp) ) ctr+=1;
21       total += ctr;
22       fclose(fp);
23       printf("%s: %llu\n",argv[i], ctr);
24     }
25   }
26   printf("total: %llu\n",total);
27   return EXIT_SUCCESS;
28 }
```

Lesen von Blöcken

```
1 #include <stdio.h>
2
3 size_t fread(void *buf, size_t size,
4 size_t blockzahl, FILE *stream);
```

- Liest bis zu **`blockzahl`** Objekte mit **`size`** Bytes von dem Datenstrom **`stream`**.
- Die Blöcke werden nach **`buf`** geschrieben.
- Rückgabewert: Anzahl gelesener Blöcke.
- Merke: Die gelesenen Blöcke können auch **`\n`** oder **`\0`** enthalten.
- Bei **`size_t`** handelt es sich um eine vorzeichenlose Ganzzahl.

Schreiben von Blöcken

```
1 #include <stdio.h>
2
3 size_t fwrite(void *buf, size_t size, size_t
      blockzahl, FILE *stream);
```

- Schreibt bis zu **`blockzahl`** Objekte mit **`size`** Bytes von der Adresse **`buf`** in den Datenstrom **`stream`**.
- Rückgabewert: Anzahl geschriebener Blöcke.
- Die zu schreibenen Blöcke können auch **`\n`** oder **`\0`** enthalten.

Schreiben von Blöcken – Beispiel

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "person.h"
4
5  int main() {
6    FILE *fp = fopen ("test.data", "a+");
7    struct person alice = {
8      "Alice\0" , "Smith\0", 23
9    };
10   struct person bob =   {
11     "Bob\0" , "Jonson\0", 27
12   };
13   size_t size = sizeof(struct person);
14
15   if (!fp) return EXIT_FAILURE;
16
17   fwrite(&alice, size, 1, fp);
18   fwrite(&bob, size, 1, fp);
19
20   fclose(fp);
21   return EXIT_SUCCESS;
22 }
```

Lesen von Blöcken – Beispiel

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "person.h"
4
5  int main() {
6    FILE *fp = fopen ("test.data", "r");
7    struct person p;
8
9    if(!fp) {
10     return EXIT_FAILURE;
11   }
12
13   while(fread(&p, sizeof(struct person), 1, fp)) {
14     printf("%s␣%s␣(%u)\n", p.firstname, p.lastname, p.age)
       ;
15   }
16
17   fclose(fp);
18   return EXIT_SUCCESS;
19 }
```

Unterschiedliches Zeitverhalten von I/O-Operationen

Frage: Wie performant sind die vorgestellten Funktionen?

- Das Lesen und Schreiben großer Dateien mit `getc()` und `putc()` ist enorm langsam, da für jedes Zeichen ein Funktionsaufruf benötigt wird.
- Das Lesen und Schreiben großer Text-Dateien mit `fgets()` und `fputs()` ist schon etwas schneller.
- Das Lesen und Schreiben von Dateien geht am schnellsten mit `fread()` und `fputs()`. Als Puffergröße wird ein Vielfaches von 4096 Bytes (4 KiB) empfohlen.
- Im Folgenden messen wir die Performance von drei Programmen: `copybyte`, `copyline` und `copyblock`.

`copybyte` – Byteweises Kopieren von Dateien

```
1  #include<stdlib.h>
2  #include<stdio.h>
3
4  int main(int argc, char * argv[]) {
5    FILE *fpr;
6    FILE *fpw;
7    int c;
8
9    if(argc != 3) {
10     fputs("Usage: copybyte <source> <target>\n",stderr);
11     return EXIT_FAILURE;
12   }
13   fpr = fopen(argv[1], "r");
14   fpw = fopen(argv[2], "w");
15
16   if(!fpr) { perror(argv[1]); return EXIT_FAILURE; }
17   if(!fpw) { perror(argv[2]); return EXIT_FAILURE; }
18
19   while((c = fgetc(fpr)) != EOF) {
20     fputc(c,fpw);
21   }
22   return EXIT_SUCCESS;
23 }
```

`copyline` – Zeilenweises Kopieren von Dateien

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <limits.h>
4
5  int main(int argc, char * argv[]) {
6    FILE *fpr, *fpw;
7    char buf[LINE_MAX+1];
8
9    if(argc != 3) {
10     fputs("Usage: copyline <source> <target>\n",stderr);
11     return EXIT_FAILURE;
12   }
13
14   fpr = fopen(argv[1], "r");
15   fpw = fopen(argv[2], "w");
16
17   if(!fpr) { perror(argv[1]); return EXIT_FAILURE; }
18   if(!fpw) { perror(argv[2]); return EXIT_FAILURE; }
19
20   while(fgets(buf,LINE_MAX,fpr)) {
21     fputs(buf, fpw);
22   }
23   return EXIT_SUCCESS;
24 }
```

`copyblock` – Blockweises Kopieren von Dateien

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main(int argc, char * argv[]) {
6    FILE *fpr, *fpw;
7    int size = sysconf(_SC_PAGESIZE);
8    char buf[size];
9
10   if(argc != 3) {
11     fputs("Usage: copyblock <source> <target>\n",stderr);
12     return EXIT_FAILURE;
13   }
14
15   fpr = fopen(argv[1], "r");
16   fpw = fopen(argv[2], "w");
17
18   if(!fpr) { perror(argv[1]); return EXIT_FAILURE; }
19   if(!fpw) { perror(argv[2]); return EXIT_FAILURE; }
20
21   while(fread(buf,size,1,fpr)) {
22     fwrite(buf, size, 1, fpw);
23   }
24   return EXIT_SUCCESS;
25 }
```

Performance-Benchmarks

Task: Kopieren einer 34 MB großen Textdatei mit 744.764 Zeilen.

Programm	Zeit
`copybyte`	ca. 750 ms
`copyline`	ca. 131 ms
`copyblock`	ca. 50 ms

7.3: Pufferung

Über die folgenden Konstanten (`<stdio.h>`) können die Pufferungsarten für einen Datenstrom eingestellt werden:

- `_IOFBF` – Vollpufferung (Default)
  Das eigentliche Lesen und Schreiben in einer Datei (Stream) findet immer erst dann statt, wenn der entsprechende Puffer gefüllt ist.
- `_IOLBF` – Zeilenpufferung
  Das eigentliche Lesen einer und Schreiben in eine Datei (Stream) findet immer erst dann statt, wenn ein `\n` gelesen oder geschrieben wird bzw. der entsprechende Puffer voll ist.
- `_IONBF` – Keine Pufferung
  Das eigentliche Lesen einer und Schreiben in eine Datei findet immer direkt statt. Normalerweise ist das Schreiben auf `stderr` ungepuffert.

Einstellen der Pufferungsarten

```
1 #include <stdio.h>
2
3 int setvbuf(FILE *pf, char *buf, int mode, size_t size);
```

- Die Funktion müssen direkt nach dem Öffnen einer Datei aufgerufen werden
- Im Erfolgsfall wird `0` zurückgegeben
- Wird `buf` auf `NULL` gesetzt, ist nur der Modus betroffen
- `setbuffer(stream, NULL,_IONBF,0)` deaktiviert den Puffer
- Standardpuffer der Länge `BUFSIZ` wird durch einen neuen Puffer der Länge `size` ersetzt
- Beispiel: `pf` mit neuem Zeilenpuffer ausstatten: `setvbuf(pf, buf, _IOLBUF, size)`

`setvbuf()` Wrapperfunktionen

Die folgenden Funktionen erleichtern das Programmiererleben:

```
1  #include <stdio.h>
2
3  void setbuf(FILE *pf, char *buf);
4  void setbuffer(FILE *pf, char *buf, size_t size);
5  void setlinebuf(FILE *pf);
```

- **`setbuf(pf, buf)`** entspricht
  `setvbuf(pf, buf, buf ? _IOFBF : _IONBF, BUFSIZ)`
  (Hinweis: Die Länge von `buf` muss mind. `BUFSIZ` sein)
- **`setlinebuf(pf)`** entspricht
  `setvbuf(pf, NULL, _IOLBF, 0)`
- **`setbuffer(pf, buf, size);`** entspricht
  `setvbuf(pf, buf, buf ? _IOFBF : _IONBF, size)`

Typischer Fehler

Die lokale Deklaration eines Arrays welches als Dateipuffer verwendet wird, ist ein typischer Fehler.

Frage: Warum ist dies schlimm, und wie wird es richtig gemacht?

Typischer Fehler – Beispiel

```
1  #include <stdio.h>
2
3  void setlbuf(FILE *pf, const size_t size) {
4    char buf[size];
5    setvbuf(pf, buf, _IOFBF, size);
6  }
7
8  int main() {
9    size_t size = 2 * BUFSIZ;
10   FILE *pfile = fopen("zpg.txt", "r");
11   char line[size];
12
13   setlbuf(pfile, size);
14   while(fgets(line, size, pfile)) puts(line);
15   return 0;
16 }
```

Diese Programm bricht mit einem Segmentation Fault ab.

Aufgabe: Korrigieren Sie die fehlerhafte(n) Zeile(n).

Performance-Unterschiede

Frage: Welche Auswirkungen haben die verschiedenen Pufferungstechniken auf die Lese- bzw. Schreib-Performance?

Antwort: Übung.

Inhalte von Puffern in eine Datei übertragen

```
1 #include <stdio.h>
2
3 fflush(FILE *stream);
```

- Leert alle Puffer, die dem Datenstrom **`stream`** zugeordnet sind.
- **`fflush(NULL)`** leert sämtliche Ausgabepuffer bei denen die letzte Aktion kein Lesen war
- Das Verhalten von **`fflush()`** auf Dateien, von denen zuletzt gelesen wurde, ist undefiniert
- **`fflush()`** soll daher nur beim Schreiben verwendet werden

7.4: Misc

Das letzte Abschnitt des Kapitels beschäftigt sich mit den folgenden drei Themen:

1. Positionieren in einer Datei
2. Temporäre Dateien
3. Löschen und Umbennenen von Dateien

Der Datei-Offset

```
1  #include <stdio.h>
2
3  long ftell(FILE *stream);
4  int fseek(FILE *stream, long offset, int whence);
```

- **`ftell()`** ermittelt den aktuellen Datei-Offset und gibt den Abstand zum Dateianfang in Bytes zurück.
- **`fseek()`** setzt den Datei-Offset von **`stream`** auf eine gewünschte Position, welche sich aus aus der Basis **`whence`** und dem dazugehörigen Byteoffset **`offset`** berechnet.
- Gültige Werte für **`whence`**:
  - **`SEEK_SET`**: Dateianfang
  - **`SEEK_CUR`**: Aktuelle Position
  - **`SEEK_END`**: Dateiende
- **`fseek()`** gibt bei Erfolg **`0`** zurück; ansonsten **`-1`**

Hexdump eines Dateiausschnitts

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4   void usage() {
5     fprintf(stderr,"usage: hexbytes <file> <len> <end>\n");
6     exit(EXIT_FAILURE);
7   }
8
9  int main (int args, char *argv[]) {
10   FILE *fp;
11   size_t size;
12   unsigned char *buf;
13
14   if(args != 4) usage();
15   if( !(fp = fopen(argv[1], "r")) ) usage();
16
17   fseek(fp, atoi(argv[2]), SEEK_SET);
18   size = (size_t) atol(argv[3]);
19   buf = malloc(size);
20   fread(buf, size, 1, fp);
21
22   for(size_t i = 0; i< size; i++) {
23     printf("%02x ", buf[i]);
24     if(!((i+1)%17)) puts("");
25   }
26   puts("");
27   return EXIT_SUCCESS;
28 }
```

Löschen einer Datei

```
1  #include <stdio.h>
2
3  int remove(const char *pathname);
```

- **`remove()`** löscht eine angegeben Datei, Link oder Verzeichnis
- Falls die Datei noch geöffnet ist, bleibt Sie bestehen, bis alle Prozesse welche diese Datei geöffnet haben terminieren.
- Beim erfolgreichen Aufruf wird `0` zurückgegeben, ansonsten `-1`

Umbenennen einer Datei

```
1 #include <stdio.h>
2
3 int rename(const char *oldpath, const char *newpath);
```

- **`rename()`** benennt eine Datei um und verschiebt sie, wenn nötig, in ein anderes Verzeichnis
- Achtung: Falls **`newpath`** existiert, wird er überschrieben
- Wenn **`newpath`** bereits existiert, aber die Umbenennung fehlschlägt, garantiert **`rename()`**, dass **`newpath`** erhalten bleibt
- Bei Erfolg wird `0` zurückgegeben, ansonsten `-1`.

Temporäre Dateien

- Temporäre Dateien werden nur kurzfristig bei einer Programmausführung gebraucht und werden am Programmende nicht mehr benötigt
- Auf Unix werden temporäre Dateien unter `/tmp` bzw. `/var/tmp` angelegt
- Dateien in `/tmp` werden bei Neustart des Systems gelöscht
- Dateien in `/var/tmp` überdauern einen Neustart
- Temporäre Dateien werden oftmals auch Tempfiles gennant

Erzeugen und automatisches Löschen von Tempfiles

```
1 #include <stdio.h>
2
3 FILE *tmpfile(void);
```

- **`tmpfile()`** öffnet eine temporäre Datei unter **`/tmp`** mit eindeutigem Namen zum Lesen und Schreiben
- Das Tempfile wird automatisch gelöscht, sobald sie geschlossen oder das Programm beendet wird. Dies wird erreicht, indem **`tmpfile()`** nach dem Öffnen der Datei diese gleich wieder löscht. Damit wird die Datei durch das Schließen gelöscht ($\rightarrow$ strace Demo)
- Im Fehlerfall gibt die Funktion **`NULL`** zurück

7.5: Dateideskriptoren und Datenströme

Dieser Abschnitt behandelt die folgenden Themen:

- Duplizieren von Dateideskriptoren (`dup()` und `dup2()`)
- Dateideskriptor von einem Datenstrom erfragen
- Datenstrom zu einem Dateideskriptor erzeugen

Duplizieren von Dateideskriptoren I

```
1 #include <unistd.h>
2
3 int dup2(int oldfd, int newfd);
```

- Systemcall **`dup()`** ist eine atomare Operation
- **`dup()`** erzeugt eine Kopie des Dateideskriptors **`oldfd`**
- Systemcall **`dup2()`** erzeugt eine Kopie des Dateideskriptors **`oldfd`** mit dem Wert **`newfd`**
- Bei Erfolg wird der neue Dateideskriptor zurückgegeben, ansonsten **`-1`**
- Der neue und alte Dateideskriptor verweisen auf den gleichen Eintrag in der Dateitabelle

Duplizieren von Dateideskriptoren II

```
1 #include <unistd.h>
2
3 int dup2(int oldfd, int newfd);
```

- Systemcall **`dup2()`** ist eine atomare Operation und verhält sich analog zu **`dup()`**
- Es wird eine Kopie des Dateideskriptors **`oldfd`** mit dem Wert **`newfd`** erzeugt
- Ist **`oldfd`** valide und ungleich **`newfd`**, wird **`oldfd`** geschlossen, bevor er erneut benutzt wird
- Ist **`newfd`** gleich **`oldfd`** wird einfach **`oldfd`** zurückgegeben, falls er gültig ist
- Ist **`oldfd`** kein gültiger Dateideskriptor, schlägt der Aufruf fehl und **`newfd`** wird nicht geschlossen

Beispiel – Umleitung von `stderr` nach `stdout`

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main() {
6    dup2(STDOUT_FILENO, STDERR_FILENO);
7    perror("");
8    return EXIT_SUCCESS;
9  }
```

```
$ ./dupdup2
Success
$./dupdup2 > /dev/null
$./dupdup2 2> /dev/null
Success
```

Dateideskriptoren zu einem Datenstrom erfragen

```
1 #include <stdio.h>
2
3 int fileno(FILE *stream);
```

- **`fileno()`** liefert zu einem Datenstrom den dazugehörigen Dateideskriptor
- Im Fehlerfall wird **`-1`** zurückgegeben
- **`fileno()`** wird benötigt, wenn eine Datei mit **`fopen()`** geöffnet wurde und ein Systemcall welcher der dazugehörigen Dateideskriptor benötigt, ausgeführt werden soll (z. B. **`dup()`**)

Datenstrom für einen Dateideskriptor erzeugen

```
1 #include <stdio.h>
2
3 FILE *fdopen(int fd, const char *modus);
```

- **`fdopen()`** erzeugt den Dateideskriptor **`fd`**
- **`modus`**: Zugriffsart. Die folgenden Werte sind valide: `"r"`, `"w"`, `"a"`, `"r+"`, `"w+"`, `"a+"` (siehe Kapitel 7.1)
- Im Fehlerfall wird **`NULL`** zurückgegeben
- **`fdopen()`** wird oft in der Netzwerkprogrammierung eingesetzt, weil das Öffnen eines Netzwerksockets einen Dateideskriptor zurückliefert.

Beispiel – `fileno` und `fdopen` I

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  void error(char *msg) {
6    perror(msg);
7    exit(EXIT_FAILURE);
8   }
9
10 int main () {
11   char *filename = "yz.txt";
12   FILE *fp, *fp2;
13   int fd;
14
15   printf("stdout␣(%d)\n", fileno(stdout));
16   printf("stdin␣␣(%d)\n", fileno(stdin));
17   printf("stderr␣(%d)\n", fileno(stderr));
18
19   fp = fopen(filename, "w+");
20   fd = fileno(fp);
21   if(fd < 0) error(filename);
22   printf("%s␣(%d)\n", filename, fd);
```

Beispiel – `fileno` und `fdopen` II

```
23
24    fp2 = fdopen( dup2(fd, 9), "r" );
25    if(!fp2) error(filename);
26    printf("%s (%d)\n", filename, fileno(fp2));
27
28    fclose(fp2);
29    fclose(fp);
30    return EXIT_SUCCESS;
31  }
```

Zusammenfassung

Sie sollten . . .

- . . . wissen was ein Filedeskriptor ist.
- . . . die Default-Filedeskriptoren kennen.
- . . . in der Lage sein Dateien zu öffnen und wieder zu schließen.
- . . . die verschiedenen Datei-Zugriffsarten kennen.
- . . . in der Lage sein Systemfehlermeldungen auszugeben.
- . . . Datenströme umlenken können.
- . . . das Lesen und Schreiben von Dateien beherrschen.
- . . . das unterschiedliche Zeitverhalten von I/O-Operationen kennen.
- . . . wissen welche Pufferungsarten es gibt, wozu diese benötigt werden sowie den Umgang mit ihnen beherschen.