- Die Systemschnittstelle für Ein- und Ausgabe dient primär zwei Zielen:
  - Sie sollte möglichst gut abstrahieren und somit Anwendungen befreien von Hardware-Abhängigkeiten und bis zu einem gewissen Umfange auch von den Besonderheiten eines Dateisystems.
  - Sie sollte eine höchstmögliche Effizienz erlauben bis hin zum Verzicht auf jegliche zusätzliche Kopieraktionen zwischen dem Betriebssystem und dem Adressraum des Prozesses (*zero copy*).

- Dateideskriptoren sind ganzzahlige Werte aus dem Bereich $[0, N-1]$, wobei N typischerweise eine Zweierpotenz ist (etwa 512 oder 1024).
- Dateideskriptoren werden innerhalb des Betriebssystems als Indizes für Verwaltungstabellen verwendet.
- Dateideskriptoren referenzieren somit vom Betriebssystem verwaltete Objekte.
- Für jeden Prozess verwaltet das System eine eigene Tabelle. Entsprechend kann beispielsweise der Dateideskriptor 2 bei zwei Prozessen mit völlig unterschiedlichen Objekten verbunden sein.
- Die so referenzierten Objekte sind typischerweise Dateien, können aber auch Netzwerkverbindungen, Verbindungen zu anderen Prozessen, Geräte und Speicherbereiche sein.
- In C wird für Dateideskriptoren der Datentyp **int** verwendet.

Wieviel Dateideskriptoren gibt es?

openmax.c

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

int main() {
   long maxfds = sysconf(_SC_OPEN_MAX);
   printf("maximal number of open file descriptors: %ld\n", maxfds);
}
```

- Der Systemaufruf *sysconf* erlaubt die Abfrage zahlreicher Größen, von denen auch einige erst zur Laufzeit festliegen.
- Der Parameter *_SC_OPEN_MAX* liefert die maximale Zahl offener Dateien und die damit die Größe der systeminternen Tabelle der Objekte für diesen Prozess.

```
theon$ gcc -Wall -std=c11 -o openmax openmax.c
theon$ ./openmax
maximal number of open file descriptors: 512
theon$
```

- Wenn ein neuer Prozess erzeugt wird, dann wird die Tabelle mit den Dateideskriptoren kopiert.
- Entsprechend kann die Shell einige Dateideskriptoren für ein Programm, das sie startet, vorbereiten.
- Wenn nichts anderes spezifiziert wird, sind dies folgende Dateideskriptoren:
  - 0 Standard-Eingabe
  - 1 Standard-Ausgabe
  - 2 Standard-Fehlerausgabe
- Die Bourne-Shell und die von ihr abgeleiteten Shells erlauben das Öffnen und Schließen beliebiger Dateideskriptoren. Folgendes Beispiel ruft a.out auf, wobei 0 geschlossen wird, 7 zum Schreiben geöffnet wird auf die Datei *out* und 10 zum Lesen für die Datei *in* eröffnet wird:

```
a.out 0<&- 7>out 10<in
```

int *open*(**const char**∗ *path*, **int** *oflag*, . . .)

Eröffnet die Datei *path* mit den unter *oflag* angegebenen Modi. Ein weiterer Parameter ist nur bei neu anzulegenden Dateien notwendig. Der Systemaufruf liefert im Erfolgsfalle (d.h. ≥ 0) einen Dateideskriptor zurück.

Für *oflag* wird zunächst (normalerweise) eines der folgenden Flags ausgewählt:

O_RDONLY	nur zum Lesen eröffnen
O_WRONLY	nur zum Schreiben eröffnen
O_RDWR	zum Lesen und Schreiben eröffnen

int *open*(**const char**∗ *path*, **int** *oflag*, . . .)

oflag kann mit einem oder mehreren der folgenden Flags kombiniert werden:

O_APPEND	vor jeder Schreiboperation wird die Dateiposition implizit an das Ende gesetzt
O_CREAT	lege die Datei neu an, falls sie noch nicht existiert
O_EXCL	in Kombination mit *O_CREAT* kommt es zu einem Fehler, falls die Datei bereits existiert
O_TRUNC	die Datei wird, falls sie bereits existiert, auf die Länge 0 gekürzt

Optional gibt es noch einen dritten Parameter für den Fall, dass es eine Datei neu angelegt wird. Dieser gibt dann die initialen Zugriffsrechte an, z.B. 0666 für „rw-rw-rw-“.

ssize_t read(**int** fd, **void**∗ *buf*, *size_t nbytes*)

Liest bis zu *nbytes* Bytes von der Dateiverbindung *fd* und legt diese bei *buf* ab. Die Zahl der gelesenen Bytes wird zurückgeliefert (falls 0: Eingabeende, ≤ 0: Fehler).

ssize_t write(**int** fd, **const void**∗ *buf*, *size_t nbytes*)

Schreibt bis zu *nbytes* Bytes bei *buf* in die mit *fd* verbundene Datei. Die Zahl der geschriebenen Bytes wird zurückgeliefert (kann niedriger sein als *nbytes*, -1 bei Fehlern).

int *close*(**int** *fd*)

Schließt eine Dateiverbindung.

off_t lseek(**int** *filedes*, *off_t offset*, **int** *whence*)

Verändert die Dateiposition *pos* in Abhängigkeit von *whence* und ggf. von der Länge der Datei *len*:

SEEK_SET	$pos = offset$
SEEK_CUR	$pos \mathrel{+}= offset$
SEEK_END	$pos = len + offset$

Der Returnwert liefert die neue Dateiposition bzw. -1 bei einem Fehler.

int *ftruncate*(**int** *filedes*, *off_t length*)

Verändert die Länge der Datei, die sowohl verkleinert als auch vergrößert werden kann. Bei einer Vergrößerung entstehen potentiell „Löcher“, die als Nullbytes ausgelesen werden.

Die beiden ganzzahligen Datentypen *size_t* und *ssize_t* haben die gleiche Größe, die an die jeweilige Plattform angepasst ist:

size_t ist Teil des C-Standards (Rückgabetyp des **sizeof**-Operators!) und steht in Standard-Headern wie beispielsweise **#include** <stdlib.h> zur Verfügung. Der Datentyp ist ohne Vorzeichen.

ssize_t ist Teil des POSIX-Standards (**#include** <unistd.h>) als Rückgabetyp von u.a. *read* und *write*. Der Datentyp ist mit Vorzeichen und erlaubt so die Rückgabe von -1.

off_t ist ebenfalls Teil des POSIX-Standards (**#include** <unistd.h>) und ist ein vorzeichenbehafteter ganzzahliger Datentyp, der für Dateipositionen verwendet wird. (*off_t* kann auch größer sein als *ssize_t*, wenn z.B. eine 32-Bit-Anwendung auf einem 64-Bit-System zur Ausführung kommt.)

syshello.c

```
#include <unistd.h>

int main() {
   const char hello[] = "Hello, world!\n";
   write(1, hello, sizeof hello - 1);
}
```

- Der Systemaufruf *write* gibt im Normalfall die angegebene Zahl von Bytes aus (möglicherweise auch weniger). Dies schließt auch die Ausgabe beliebiger binärer Inhalte einschließlich des Nullbytes ein. Bei der Ausgabe einer Zeichenkette ist daher immer die Länge ohne das Nullbyte anzugeben.
- Bei Zeichenkettenkonstanten ist implizit immer ein abschließendes Nullbyte enthalten, das in der Größenangabe von **sizeof** natürlich einberechnet wird.
- Da das Nullbyte nicht auszugeben ist, ist das von der auszugebenden Länge abzuziehen.

scopy.c

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

int main(int argc, char* argv[]) {
   char* cmdname = argv[0];
   if (argc != 3) {
      fprintf(stderr, "Usage: %s infile outfile\n", cmdname);
      exit(1);
   }
   char* infile = argv[1]; char* outfile = argv[2];
   FILE* in = fopen(infile, "r"); if (!in) perror(infile), exit(1);
   FILE* out = fopen(outfile, "w"); if (!out) perror(outfile), exit(1);
   int ch;
   while ((ch = getc(in)) != EOF) {
      if (putc(ch, out) == EOF) perror(outfile), exit(1);
   }
   fclose(in);
   if (fclose(out) == EOF) perror(outfile), exit(1);
}
```

bcopy.c

```
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stralloc.h>
#include <string.h>
#include <unistd.h>

char* cmdname;
/* ... */

int main(int argc, char* argv[]) {
   cmdname = argv[0];
   if (argc != 3) {
      stralloc usage = {0};
      if (stralloc_copys(&usage, "Usage: ") &&
            stralloc_cats(&usage, cmdname) &&
            stralloc_cats(&usage, " infile outfile\n")) {
         write(2, usage.s, usage.len);
      }
      exit(1);
   }
   /* ... */
}
```

bcopy.c

```
char* infile = argv[1]; char* outfile = argv[2];

int infd = open(infile, O_RDONLY);
if (infd < 0) die(infile);
int outfd = open(outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
if (outfd < 0) die(outfile);

for(;;) {
   char ch;
   ssize_t res = read(infd, &ch, 1);
   if (res < 0) die(infile);
   if (res == 0) break;
   res = write(outfd, &ch, 1);
   if (res <= 0) die(outfile);
}

close(infd);
if (close(outfd) < 0) die(outfile);
```

bcopy.c

```
int infd = open(infile, O_RDONLY);
if (infd < 0) die(infile);
int outfd = open(outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
if (outfd < 0) die(outfile);
```

- Mit dem Systemaufruf *open* kann eine Datei eröffnet werden. Im Erfolgsfalle wird ein (zuvor unbenutzter) Dateideskriptor zurückgeliefert.
- Der zweite Parameter gibt an, wie die Datei zu eröffnen ist. Hier wird die Datei nur zum Schreiben eröffnet. Wenn die Datei noch nicht existiert wird sie angelegt, andernfalls wird sie auf die Länge 0 gekürzt. Das entspricht genau der Semantik des Parameters "w" bei *fopen* oder einer normalen Ausgabe-Umlenkung in der Shell.
- Der optionale dritte Parameter wird nur hinzugefügt, falls bei dem zweiten Parameter *O_CREAT* mit angegeben wurde. Er legt die Zugriffsrechte fest. 0666 steht für rw-rw-rw.

Kopierschleife mit *read* und *write*

bcopy.c

```
for(;;) {
   char ch;
   ssize_t res = read(infd, &ch, 1);
   if (res < 0) die(infile);
   if (res == 0) break;
   res = write(outfd, &ch, 1);
   if (res <= 0) die(outfile);
}
```

- Sowohl bei *read* als auch *write* ist hier als Puffer nur eine **char**-Variable angegeben.
- Somit wird hier zeichenweise kopiert.

bcopy.c

```
void die(char* filename) {
   stralloc msg = {0};
   if (stralloc_copys(&msg, cmdname) &&
         stralloc_cats(&msg, ": ") &&
         stralloc_cats(&msg, strerror(errno)) &&
         stralloc_cats(&msg, ": ") &&
         stralloc_cats(&msg, filename) &&
         stralloc_cats(&msg, "\n")) {
      write(2, msg.s, msg.len);
   }
   exit(1);
}
```

- *strerror* aus **#include** <string.h> liefert die Fehlermeldung passend zu *errno*. Die bislang bekannte Funktion *perror* basiert auf *strerror*.

```
theon$ mkfile 10m 10m
theon$ time bcopy 10m out && rm -f out

real 1m8.829s
user 0m2.738s
sys 1m6.105s
theon$ time scopy 10m out && rm -f out

real 0m0.137s
user 0m0.130s
sys 0m0.007s
theon$
```

- Das Kopierprogramm mit Systemaufrufen schneidet katastrophal schlecht ab im Vergleich zu der Fassung auf Basis der *stdio*. Warum?

copy.c

```
char buf[8192]; ssize_t nbytes;
while ((nbytes = read(infd, buf, sizeof buf)) > 0) {
   ssize_t count;
   for (ssize_t written = 0; written < nbytes; written += count) {
      count = write(outfd, buf + written, nbytes - written);
      if (count <= 0) die(outfile);
   }
}
if (nbytes < 0) die(infile);
```

- Durch die Pufferung reduzieren wir deutlich die Zahl der benötigten Systemaufrufe.
- Bei *write* akzeptieren wir die Möglichkeit, dass weniger Bytes geschrieben werden als verlangt. Das ist immer möglich – etwa durch eine Unterbrechung. Entsprechend ist eine Schleife notwendig, die darauf Rücksicht nimmt.

```
theon$ time scopy 10m out && rm -f out

real 0m0.137s
user 0m0.130s
sys 0m0.007s
theon$ time copy 10m out && rm -f out

real 0m0.027s
user 0m0.002s
sys 0m0.025s
theon$
```

- Jetzt schneidet die auf Systemaufrufen basierende Fassung deutlich besser ab.
- Das liegt insbesondere daran, dass die *stdio*-Fassung zeichenweise operiert. Zwar gibt es ebenfalls wenige Systemaufrufe dank der Pufferung der *stdio*, aber der CPU-Aufwand für das zeichenweise Kopieren bleibt dennoch.

mcopy.c

```
struct stat statbuf; if (fstat(infd, &statbuf) < 0) die(infile);
off_t nbytes = statbuf.st_size;
char* buf = (char*) mmap(0, nbytes, PROT_READ, MAP_SHARED, infd, 0);
if (buf == MAP_FAILED) die(infile);
ssize_t count;
for (ssize_t written = 0; written < nbytes; written += count) {
   count = write(outfd, buf + written, nbytes - written);
   if (count <= 0) die(outfile);
}
```

- Der Systemaufruf *mmap* (*memory map*) erlaubt es, den Inhalt des Puffer-Cache, der zu einer Datei gehört, direkt in den eigenen Adressraum zu legen.
- Auf diese Weise entfällt das Kopieren des Inhalts der zu kopierenden Datei in den Adressraum des Kopierprogramms.
- Allerdings sind auch die Anpassungen des virtuellen Adressraums und die laufenden *page faults* auch nicht ohne Kosten.

```
theon$ time scopy 10m out && rm -f out

real 0m0.137s
user 0m0.130s
sys 0m0.007s
theon$ time copy 10m out && rm -f out

real 0m0.027s
user 0m0.002s
sys 0m0.025s
theon$ time mcopy 10m out && rm -f out

real 0m0.019s
user 0m0.001s
sys 0m0.017s
theon$
```