- 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.

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.

```
doolin$ gcc -Wall -std=c99 openmax.c
doolin$ a.out
maximal number of open file descriptors: 512
doolin$
```

- 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
```

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);
}
```

copy.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);
   }
   /* ... */
}
```

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

copy.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 können zahlreiche Werte mit einem bitweisen Oder verknüpft werden, wobei nicht jede Kombination sinnvoll ist. Eine Auswahl:

O_RDONLY	Nur zum Lesen eröffnen
O_WRONLY	Nur zum Schreiben eröffnen
O_RDWR	Zum Lesen und Schreiben eröffnen
O_CREAT	Datei neu anlegen, falls noch nicht existent
O_TRUNC	Datei auf Länge 0 kürzen, falls existent

- 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.

copy.c

```
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);
```

- Die Systemaufrufe *read* und *write* erhalten jeweils als Parameter einen Dateideskriptor, einen Zeiger auf einen Puffer und eine Angabe, wieviel Bytes maximal zu transferieren sind.
- Grundsätzlich haben *read* und *write* die Freiheit, weniger Bytes zu übertragen als angegeben. Das sollte nicht als Fehler missverstanden werden.
- Der Rückgabewert gibt die Zahl der übertragenen Bytes im Erfolgsfalle (immer positiv) oder ist gleich 0 (bei *read* steht dies für das Eingabeende) oder -1 bei Fehlern.

copy.c

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

- Mit *close* können Dateideskriptoren geschlossen werden.
- Bei einem zuvor zum Schreiben geöffneten Dateideskriptor ist es sinnvoll, den Erfolg zu überprüfen, weil so noch am Ende aufgetretene Fehler erkannt werden können – auch wenn dies eher selten der Fall sein dürfte.

copy.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* liefert die Fehlermeldung passend zu *errno*. Die bislang bekannte Funktion *perror* basiert auf *strerror*.

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.

```
thales$ mkfile 10m 10m
thales$ time scopy 10m out && rm out

real    0m0.099s
user    0m0.084s
sys     0m0.011s
thales$ time copy 10m out && rm out

real    0m0.020s
user    0m0.001s
sys     0m0.016s
thales$ time mcopy 10m out && rm out

real    0m0.019s
user    0m0.000s
sys     0m0.014s
thales$
```

- Prinzipiell erlaubt Unix den konkurrierenden Zugriff mehrerer Prozesse auf die gleiche Datei.
- Das u.U. notwendige gegenseitige Ausschließen und die Atomizität von Änderungen ergeben sich dabei nicht von selbst, sondern sind Aufgabe der parallel zugreifenden Anwendungen.
- Es gibt aber einige Systemaufrufe, die hier eine Hilfestellung leisten können.

Es ist ein kleines Werkzeug *unique* zu entwickeln, das einen Dateinamen als Parameter erhält und folgende Anforderungen erfüllt:

- Die Zahl in der gegebenen Datei ist auszulesen, um eins zu erhöhen, wieder in die Datei zu schreiben und auf der Standardausgabe auszugeben.
- Gegenseitiger Ausschluss: Jeder Wert darf höchstens einmal ausgegeben werden, egal wieviele Instanzen des Programms gleichzeitig auf die Datei zugreifen.
- Atomizität: Die Datei muss immer einen gültigen Inhalt haben, selbst wenn inmitten einer Operation der Strom ausfällt.

Wenn mehrere gleichzeitig zugreifende Prozesse sich gegenseitig ausschließen möchten, kommen folgende auf dem Dateisystem basierende Techniken in Frage, die alle ohne Interprozess-Kommunikation auskommen:

- Option *O_EXCL* zusammen mit *O_CREAT* bei *open* setzen. Dann ist *open* nur erfolgreich, wenn die Datei vorher noch nicht existiert.
- Mit *link* zu einer existierende Datei einen weiteren Namen hinzufügen. Dies ist nur erfolgreich, wenn der neue Name noch nicht existiert.
- Mit *lockf* können bei einem gegebenen Deskriptor einzelne Bereiche reserviert werden. Jedoch wird *lockf* nicht überall unterstützt oder ist (wie bei NFS) nicht ausreichend zuverlässig.

- Wenn das Ergebnis einer Schreib-Operation abgesichert werden soll, dann empfiehlt sich *fsync*, das einen Dateideskriptor erhält und im Erfolgsfalle wartet, bis der aktuelle Stand auf die Platte gesichert ist.
- Datenbanken und andere Anwendungen arbeiten bei Transaktionen mit mehreren Versionen (der alten und der neuen). Erst wenn die neue Version mit *fsync* abgesichert worden ist, wird ein Versionszeiger in der Datei so aktualisiert, dass er auf die neue Fassung verweist.

- Im einfachen Falle empfiehlt sich die Verwendung des Systemaufrufs *rename*.
- Hier wird zunächst eine vollständig neue Version der Daten in einer temporären Datei erstellt.
- Dann wird *rename* aufgerufen mit der temporären Datei und der eigentlichen Datei als Ziel.
- Das ist auch zulässig, wenn das Ziel existiert. In diesem Falle wird implizit zuvor der alte Verweis gekappt.
- Diese Operation ist atomar und alle anderen Prozesse sehen entweder den alten oder den neuen Inhalt, vermissen aber nie die Datei und sehen unter keinen Umständen eine nur teilweise beschriebene Datei.

Beispiel: *unique*

unique.c

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

char* cmdname;
stralloc tmpfile = {0}; bool tmpfile_created = false;

/* print an out of memory message to standard error and exit */
void memerr() { /* ... */ }

/* print a error message to standard error and exit;
   include "message" in the output message, if not 0,
   otherwise strerror(errno) is being used
*/
void die(char* filename, char* message) { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

Beispiel: *unique*

unique.c

```
/* print an out of memory message to standard error and exit */
void memerr() {
   static char memerrmsg[] = "out of memory error\n";
   write(2, memerrmsg, sizeof(memerrmsg) - 1);
   if (tmpfile_created) unlink(tmpfile.s);
   exit(1);
}
```

- Sollte tatsächlich der Speicher ausgehen, dann sollte die Ausgabe der zugehörigen Fehlermeldung ohne dynamische Speicheranforderungen auskommen.
- Von **sizeof**(*memerrmsg*) wird 1 abgezogen, weil das Nullbyte nicht auszugeben ist.
- Wenn die Ausführung abgebrochen wird, sollten ggf. temporäre Dateien aufgeräumt werden. Mit *unlink* kann eine Verweis aus einem Verzeichnis auf eine Datei entfernt werden.

Beispiel: *unique*

unique.c

```
/* print a error message to standard error and exit;
   include "message" in the output message, if not 0,
   otherwise strerror(errno) is being used
*/
void die(char* filename, char* message) {
   stralloc msg = {0};
   if (stralloc_copys(&msg, cmdname) &&
         stralloc_cats(&msg, ": ") && (
            message?
               stralloc_cats(&msg, message)
            :
               stralloc_cats(&msg, strerror(errno))
         ) && stralloc_cats(&msg, ": ") &&
         stralloc_cats(&msg, filename) &&
         stralloc_cats(&msg, "\n")) {
      write(2, msg.s, msg.len);
   } else {
      memerr();
   }
   if (tmpfile_created) unlink(tmpfile.s);
   exit(1);
}
```

Beispiel: *unique*

unique.c

```
int main(int argc, char* argv[]) {
   /* process command line arguments */

   /* try to open the temporary file which also serves as a lock */

   /* determine current value of the counter */

   /* increment the counter and write it to the tmpfile */

   /* update counter file atomically by a rename */

   /* write counter value to stdout */
}
```

- Vorgehensweise: Wir erhalten einen Dateinamen als Argument, leiten daraus den Namen einer temporären Datei ab, eröffnen diese exklusiv zum Schreiben, lesen den alten Zählerwert aus, erhöhen diesen um eins, schreiben den neuen Zählerwert in die temporäre Datei, taufen diese in den gegebenen Dateinamen um und geben am Ende den neuen Zählerwert aus.

unique.c

```
/* process command line arguments */
cmdname = argv[0];
if (argc != 2) {
   stralloc usage = {0};
   if (stralloc_copys(&usage, "Usage: ") &&
         stralloc_cats(&usage, cmdname) &&
         stralloc_cats(&usage, " counter\n")) {
      write(2, usage.s, usage.len);
   } else {
      memerr();
   }
   exit(1);
}
char* counter_file = argv[1];
```

- Genau ein Dateiname wird als Argument erwartet. In dieser Datei wird der Zähler verwaltet.

Beispiel: *unique*

unique.c

```
/* try to open the temporary file which also serves as a lock */
if (!stralloc_copys(&tmpfile, counter_file) ||
      !stralloc_cats(&tmpfile, ".tmp") ||
      !stralloc_0(&tmpfile)) {
   memerr();
}
int outfd;
for (int tries = 0; tries < 10; ++tries) {
   outfd = open(tmpfile.s, O_WRONLY|O_CREAT|O_EXCL, 0666);
   if (outfd >= 0) break;
   if (errno != EEXIST) break;
   sleep(1);
}
if (outfd < 0) die(tmpfile.s, 0);
tmpfile_created = true;
```

- Den Namen der temporären Datei gewinnen wir durch ein Anhängen der Endung ».tmp« an den übergebenen Dateinamen.
- Damit liegt die temporäre Datei im gleichen Verzeichnis wie die angegebene Datei und damit auch auf dem gleichen Dateisystem.
- Das Nullbyte am Ende der Zeichenkette *tmpfile* wird für *open* benötigt.

unique.c

```
int outfd;
for (int tries = 0; tries < 10; ++tries) {
   outfd = open(tmpfile.s, O_WRONLY|O_CREAT|O_EXCL, 0666);
   if (outfd >= 0) break;
   if (errno != EEXIST) break;
   sleep(1);
}
```

- Die Option *O_EXCL* lässt den Aufruf von *open* scheitern, wenn die Datei bereits existiert. In diesem Falle hat *errno* den Wert *EEXIST*.
- Wenn *open* aus diesem Grunde schiefgeht, wird die Operation mit Zeitverzögerung wiederholt. *sleep* erlaubt ein sekundengenaues Suspendieren des eigenen Prozesses.
- Sobald der Aufruf von *open* erfolgreich ist, schließen wir alle Konkurrenten aus.

Beispiel: *unique*

unique.c

```
/* determine current value of the counter */
int current_value;
int infd = open(counter_file, O_RDONLY);
if (infd >= 0) {
   char buf[512];
   ssize_t nbytes = read(infd, buf, sizeof buf);
   if (nbytes <= 0) die(counter_file, 0);
   current_value = 0;
   for (char* cp = buf; cp < buf + nbytes; ++cp) {
      if (!isdigit(*cp)) die(counter_file, "decimal digits expected");
      current_value = current_value * 10 + *cp - '0';
   }
} else if (errno != ENOENT) {
   die(counter_file, 0);
} else {
   /* start a new counter */
   current_value = 0;
}
```

- Sobald wir einen exklusiven Zugriff haben, lohnt es sich, den bisherigen Zählerstand auszulesen.
- Falls die Datei noch nicht existiert, gehen wir von einem bisherigen Zählerwert von 0 aus.

unique.c

```
/* increment the counter and write it to the tmpfile */
++current_value;
stralloc outbuf = {0};
if (!stralloc_copys(&outbuf, "") ||
      !stralloc_catint(&outbuf, current_value)) {
   memerr();
}
int nbytes = write(outfd, outbuf.s, outbuf.len);
if (nbytes < outbuf.len) die(tmpfile.s, 0);
if (fsync(outfd) < 0) die(tmpfile.s, 0);
if (close(outfd) < 0) die(tmpfile.s, 0);
```

- Der um eins erhöhte Zählerwert wird in die temporäre Datei geschrieben.
- Mit *fsync* wird der Inhalt der temporären Datei mit der Festplatte synchronisiert.

unique.c

```
/* update counter file atomically by a rename */
if (rename(tmpfile.s, counter_file) < 0) die(counter_file, 0);
tmpfile_created = false;
```

- Mit *rename* wird der Verweis auf die Zieldatei, falls dieser zuvor existierte, implizit mit *unlink* entfernt und danach die temporäre Datei in die Zieldatei umgetauft.
- IEEE Std 1003.1 verlangt ausdrücklich, dass *rename* atomar ist. Dies ist eine Präzisierung im Vergleich zu ISO 9989-2011 (C11-Standard), das den Fall, dass die Zieldatei existiert, ausdrücklich offen lässt.

unique.c

```
/* write counter value to stdout */
if (!stralloc_cats(&outbuf, "\n")) memerr();
nbytes = write(1, outbuf.s, outbuf.len);
if (nbytes < outbuf.len) die("stdout", 0);
```

- Am Ende wird hier, falls alles soweit erfolgreich war, der neue Zählerwert auf der Standard-Ausgabe ausgegeben.

Folgende Nachteile sind mit dem vorgestellten Beispiel verbunden:

- Sollte das Programm gewaltsam terminiert werden, während die temporäre Datei noch existiert, kommt keine weitere Instanz mehr zum Zuge, da alle darauf warten, dass diese irgendwann verschwindet. Das Problem kann dahingehend angegangen werden, dass die anderen Instanzen überprüfen, ob derjenige, der dem die Datei gehört, noch lebt. Dies ist möglich, wenn die Prozess-ID bekannt ist und der Prozess auf dem gleichen Rechner läuft. Andernfalls läuft es nur über Netzwerkprotokolle oder über Heuristiken, die eine zeitliche Beschränkung einführen.
- Eine Wartezeit von einer Sekunde ist recht grob. Kleinere Wartezeiten sind mit Hilfe des Systemaufrufs *poll* möglich.

unique2.c

```
void randsleep() {
   static int invocations = 0;
   if (invocations == 0) {
      srand(getpid());
   }
   ++invocations;
   /* determine timeout value (in milliseconds) */
   int timeout = rand() % (10 * invocations + 100);
   if (poll(0, 0, timeout) < 0) die("poll", 0);
}
```

- *poll* blockiert den aufrufenden Prozess bis zum Eintreffen eines Ereignisses (aus einer Menge gegebener Ereignisse im Kontext von Dateideskriptoren) oder wenn ein Zeitlimit abgelaufen ist.
- Das Zeitlimit wird in Millisekunden als ganze Zahl spezifiziert.
- Im einfachsten Falle kann *poll* wie hier auch als reines Suspendierungs-Werkzeug verwendet werden, das im Gegensatz zu *sleep* Zeitangaben in Millisekunden akzeptiert.
- Wie genau das jedoch aufgelöst wird, hängt vom Betriebssystem ab.

- Grundsätzlich können beliebig viele Prozesse gleichzeitig auf die gleiche Datei zugreifen.
- Eine Synchronisierung oder Koordinierung bleibt grundsätzlich den Anwendungen überlassen.
- Es gibt aber einen entscheidenden Punkt: Arbeiten die konkurrierenden Prozesse mit unabhängig voneinander geöffneten Dateideskriptoren oder sind die Dateideskriptoren gemeinsamen Ursprungs?
- Dateideskriptoren können vererbt werden. Bei der Shell wird dies intensiv ausgenutzt, um beispielsweise die Standard-Kanäle im gewünschten Sinne vorzubereiten.
- Zu jedem Dateideskriptor gibt es eine aktuelle Position. Wenn Dateideskriptoren vererbt werden, arbeiten alle Erben mit der gleichen Position.

write10.c

```
int main(int argc, char* argv[]) {
   cmdname = argv[0];
   for (int i = 1; i <= 10; ++i) {
      stralloc text = {0};
      if (!stralloc_copys(&text, "")) memerr();
      if (!stralloc_catint(&text, getpid())) memerr();
      if (!stralloc_cats(&text, ": ")) memerr();
      if (!stralloc_catint(&text, i)) memerr();
      if (!stralloc_cats(&text, "\n")) memerr();
      ssize_t nbytes = write(1, text.s, text.len);
      if (nbytes < text.len) die("stdout", 0);
   }
}
```

- Dieses Programm ruft 10 mal *write* auf, um die eigene Prozess-ID zusammen mit einer laufenden Nummer auf der Standardausgabe auszugeben.
- Dies dient im folgenden als Testkandidat.

Testfall 1

test1

```
#!/bin/sh

rm -f out

./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
```

- Hier wird das Testprogramm 30 mal aufgerufen und dabei jeweils individuell die Ausgabedatei zum Schreiben eröffnet.
- Das Eröffnen erfolgt durch die Shell mit den Optionen *O_WRONLY*, *O_CREAT* und *O_TRUNC*.
- Jedes Programm arbeitet mit einem eigenen unabhängigen Dateideskriptor, der jeweils ab Position 0 beginnt.

Testfall 2

test2

```
#!/bin/sh

rm -f out

./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
```

- Hier wird von der Shell die Ausgabedaei wiederum jeweils individuell zum Schreiben eröffnet.
- Aber diesmal fällt die Option *O_TRUNC* weg.
- Stattdessen positioniert die Shell den Dateideskriptor an das aktuelle Ende.
- Nach wie vor arbeitet jeder der aufgerufenen Prozesse mit einer eigenen Dateiposition.

Testfall 3

test3

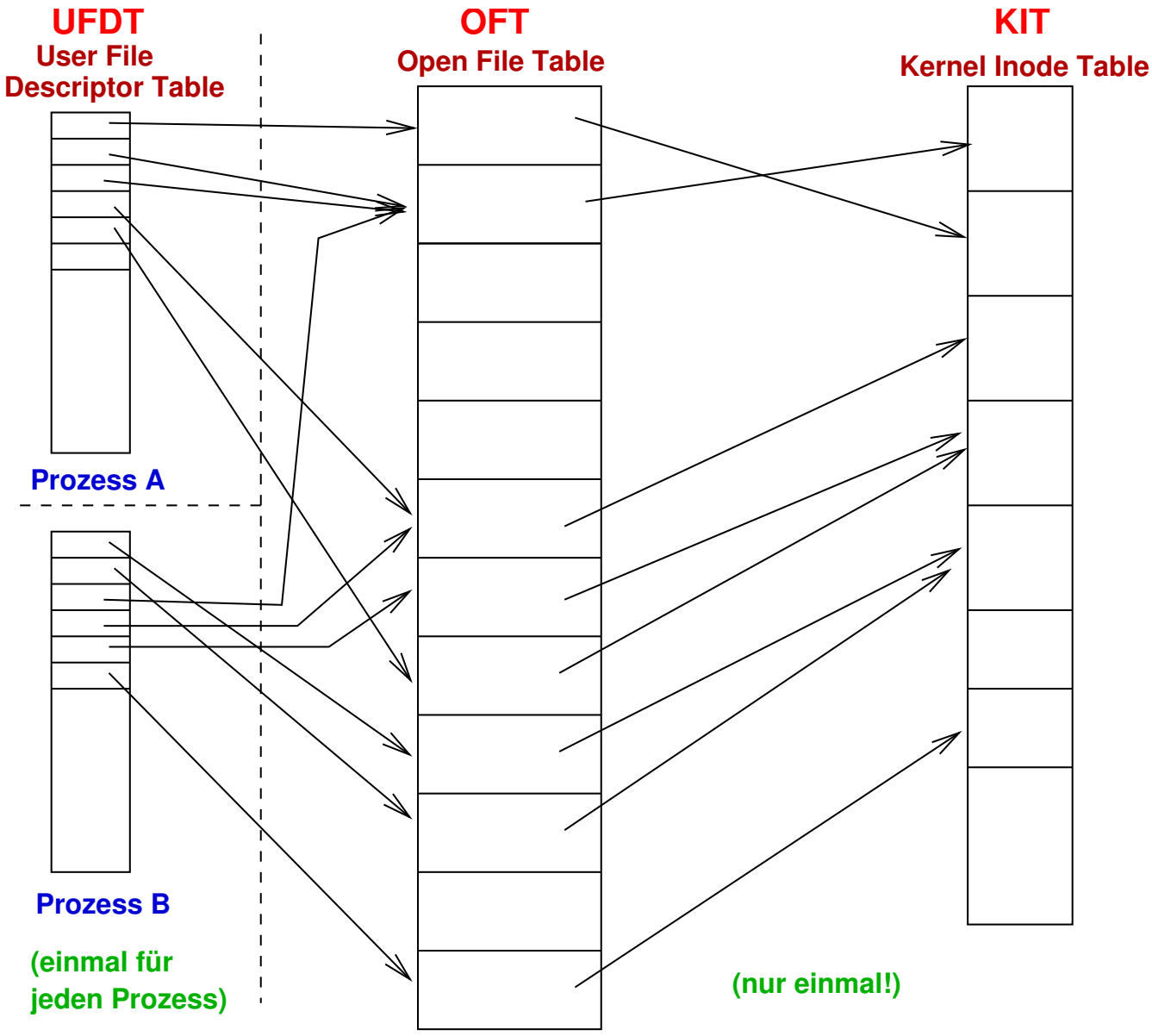
```
#!/bin/sh

rm -f out

exec >out

./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
```

- Hier eröffnet die Shell die Ausgabedatei genau einmal zu Beginn im Rahmen der *exec*-Anweisung.
- Dieser Dateideskriptor wird danach an alle aufgerufenen Prozesse vererbt.
- Entsprechend arbeiten alle Prozesse mit einer gemeinsamen Dateiposition.

```
turing$ ./testit1
turing$ wc -l out
     10 out
turing$ ./testit2
turing$ wc -l out
     50 out
turing$ ./testit2
turing$ wc -l out
     29 out
turing$ ./testit3
turing$ wc -l out
    300 out
turing$
```

- Nur im dritten Falle geht hier keine Ausgabe verloren. Allerdings könnte diese bunt gemischt sein.
- IEEE Std 1003.1 garantiert allerdings hier nicht die Verlustfreiheit, weil das Betriebssystem nicht notwendigerweise entsprechend intern synchronisiert. In der Praxis kann dies allerdings bei *write* dennoch klappen. Bei *read* wird aus Performance-Gründen jedoch weitgehend darauf verzichtet.

Die internen Datenstrukturen für Dateien



- Diese Tabelle gibt es für jeden Prozess.
- Dateideskriptoren dienen als Index zu dieser Tabelle.
- Als Werte hat die Tabelle
  - einen Zeiger in die systemweite *Open File Table* und
  - Optionen, die nur dem Dateideskriptor zugeordnet sind – das ist momentan nur *FD_CLOEXEC*, mit dem Dateideskriptoren automatisiert beim Aufruf des Systemaufrufs *exec* geschlossen werden können. (Diese Option kann mit dem Systemaufruf *fcntl* und dem Parameter *F_GETFD* bzw. *F_SETFD* angesehen bzw. verändert werden).

- Diese Tabelle gibt es nur einmal global im Betriebssystem.
- Zu einem Eintrag gehören folgende Komponenten:
  - Ein Zeiger in die *Kernel Inode Table*.
  - Die Optionen, die bei *open* angegeben wurden und später durch *fcntl* und dem Parameter *F_GETFL* bzw. *F_SETFL* angesehen bzw. verändert werden können.
  - Die aktuelle Dateiposition.
  - Eine ganze Zahl, die die Zahl der Verweise aus der UFDT auf den jeweiligen Eintrag spezifiziert. Geht diese Zahl auf 0 zurück, kann der entsprechende Eintrag freigegeben werden.

- Diese Tabelle gibt es nur einmal global im Betriebssystem.
- Jede geöffnete Datei ist in dieser Tabelle genau einmal vertreten.
- Zu einem Eintrag gehören folgende Komponenten:
  - Eine vollständige Kopie der Inode von der Platte.
  - Eine ganze Zahl, die die Zahl der Verweise aus der OFT auf den jeweiligen Eintrag spezifiziert. Solange diese positiv ist, bleibt die Inode auch auf der Platte enthalten, selbst wenn der Referenzzähler innerhalb der Inode auf 0 ist, weil die Datei aus sämtlichen Verzeichnissen entfernt wurde.

Aufgabenstellung:

- Die Zeilen aus der Standard-Eingabe sind in einer zufälligen Reihenfolge auszugeben.
- Dies sollte möglichst effizient und mit geringem Speicherplatzbedarf geschehen.
- Die Standard-Eingabe muss nicht notwendigerweise eine Datei sein – sie könnte auch beispielsweise aus einer Pipeline kommen.
- Die Zeilen sollen beliebig lange sein können.
- Alle Permutationen sollen mit gleicher Wahrscheinlichkeit ausgewählt werden. Dies ist nicht-trivial, da die Zahl der Permutationen ($n!$ für n Zeilen) rasch die Zahl der möglichen Seed-Werte eines Pseudo-Zufallszahlengenerators übersteigt.

Vorgehensweise:

- Zunächst wird die gesamte Eingabe gelesen und dabei Buch geführt über alle gefundenen Zeilen, jeweils mit Anfangsposition und Zeilenlänge.
- Dies ist die einzige dynamische Datenstruktur, die im Speicher verbleibt.
- Danach werden Zeilen zufällig ausgewählt und ausgegeben.
- Da letzteres nur für Dateien funktioniert, wird bei Bedarf die gesamte Eingabe im ersten Durchgang in eine temporäre Datei kopiert, aus der dann später gelesen wird.

lposlist.h

```
#ifndef LPOSLIST_H
#define LPOSLIST_H

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

typedef struct lpos {
   off_t pos;
   ssize_t len; /* length without line terminator */
} lpos;

typedef struct lposlist {
   int allocated; /* number of allocated entries */
   int length; /* number of actually used entries */
   lpos* line; /* pointer to array of entries */
} lposlist;

bool add_lpos(lposlist* list, off_t pos, ssize_t len);
#endif
```

lposlist.c

```
#include <stdlib.h>
#include <unistd.h>
#include "lposlist.h"

bool add_lpos(lposlist* list, off_t pos, ssize_t len) {
   if (list->length == list->allocated) {
      int allocated = (list->allocated << 1) + 16;
      lpos* new = realloc(list->line, allocated * sizeof(lpos));
      if (!new) return false;
      list->line = new; list->allocated = allocated;
   }
   list->line[list->length++] = (lpos) {pos, len};
   return true;
}
```

- Wenn das erste Argument von *realloc* ein Nullzeiger ist, dann ist der Aufruf äquivalent zu *malloc*.
- Beginnend mit C99 können Strukturen auch innerhalb eines Ausdrucks konstruiert werden. Die Syntax gleicht der Initialisierung. Hinzukommen muss jedoch der Datentyp in Klammern vor den geschweiften Klammern.

rval.h

```
#ifndef RGEN_H
#define RGEN_H

#include <stdbool.h>

bool get_rval(int* rval);
#endif
```

- Die Aufgabe dieser Funktion ist die Generierung von Pseudo-Zufallszahlen, die nicht von einem begrenzten Seed-Wert abhängen.
- Die Funktion liefert 0 zurück, falls es nicht geklappt hat. Ansonsten wird der Zufallswert hinter dem Zeiger abgelegt und 1 zurückgeliefert.

rval.c

```
#include <fcntl.h>
#include <unistd.h>
#include "rgen.h"

bool get_rval(int* rval) {
   static int fd = 0;
   if (fd == 0) {
      fd = open("/dev/urandom", O_RDONLY);
      if (fd < 0) return false;
   }
   ssize_t nbytes = read(fd, rval, sizeof(int));
   return nbytes == sizeof(int);
}
```

- Es bietet sich die spezielle Gerätedatei */dev/urandom* an, die sich aus dem Entropie-Pool des Betriebssystems bedient.
- Alternativ gibt es auch */dev/random*, das aber solange blockiert, bis genügend Zufallswerte höchster Qualität (in Bezug auf Unvorhersehbarkeit) zur Verfügung stehen. Im Vergleich dazu blockiert */dev/urandom* nicht und überbrückt stattdessen mit einem gewöhnlichen Pseudo-Zufallszahlengenerator.

tmpfile.h

```
#ifndef TMPFILE_H
#define TMPFILE_H

int get_tmpfile();
#endif
```

- Es ist möglich, eine Datei anzulegen, sie mit *unlink* sofort wieder aus dem Verzeichnis zu entfernen und den Dateideskriptor zu behalten.
- Auch wenn dann die Datei nirgends im Dateisystem zu sehen ist, so bleibt sie dennoch erhalten, bis der letzte auf sie verweisende Dateideskriptor geschlossen wird.
- Die Funktion *get_tmpfile* legt eine entsprechende temporäre Datei an, entfernt sie gleich wieder und liefert den Dateideskriptor zurück. Die Datei ist sinnvollerweise zum Lesen und Schreiben geöffnet.

tmpfile.c

```
#include <fcntl.h>
#include <stralloc.h>
#include <unistd.h>
#include "tmpfile.h"
#include "rgen.h"

int get_tmpfile() {
   stralloc tmpfile = {0};
   for (int attempt = 0; attempt < 10; ++attempt) {
      if (!stralloc_copys(&tmpfile, "/tmp/tmp.")) return -1;
      int rval;
      if (!get_rval(&rval)) return -1;
      if (!stralloc_catint(&tmpfile, rval)) return -1;
      if (!stralloc_0(&tmpfile)) return -1;
      int outfd = open(tmpfile.s, O_RDWR|O_CREAT|O_EXCL, 0);
      if (outfd >= 0) {
         if (unlink(tmpfile.s) < 0) { close(outfd); return -1; }
         return outfd;
      }
   }
   return -1;
}
```

tmpfile.c

```
for (int attempt = 0; attempt < 10; ++attempt) {
   if (!stralloc_copys(&tmpfile, "/tmp/tmp.")) return -1;
   int rval;
   if (!get_rval(&rval)) return -1;
   if (!stralloc_catint(&tmpfile, rval)) return -1;
   if (!stralloc_0(&tmpfile)) return -1;
   int outfd = open(tmpfile.s, O_RDWR|O_CREAT|O_EXCL, 0);
   if (outfd >= 0) {
      if (unlink(tmpfile.s) < 0) { close(outfd); return -1; }
      return outfd;
   }
}
```

- Da */tmp* von vielen gleichzeitig genutzt wird, ist es sinnvoll, möglichst noch nicht gewählte Dateinamen auszuwählen. Dafür bietet sich etwa die Prozess-ID oder eine Zufallszahl an. Mehrere Versuche sollten aber besser einkalkuliert werden.

lscan.h

```
#ifndef LSCAN_H
#define LSCAN_H

#include <stdbool.h>
#include "lposlist.h"

bool scan_lines(int fd, int out, lposlist* list);
#endif
```

- Die Funktion *scan_lines*
  - liest die gesamte Eingabe aus *fd*,
  - kopiert sie nach *out*, falls *out* nicht-negativ ist, und
  - legt die gefundenen Zeilen unter *list* ab.

lscan.c

```
#include <sys/stat.h>
#include "lposlist.h"
#include "lscan.h"

static off_t get_blocksize(int fd) {
   struct stat statbuf;
   if (fstat(fd, &statbuf) < 0) return 0;
   return statbuf.st_blksize;
}

bool scan_lines(int fd, int out, lposlist* list) {
   off_t blocksize;
   if (out >= 0) {
      blocksize = get_blocksize(out);
   } else {
      blocksize = get_blocksize(fd);
   }
   if (!blocksize) return false;

   char buf[blocksize];
   // ...
}
```

lscan.c

```
static off_t get_blocksize(int fd) {
   struct stat statbuf;
   if (fstat(fd, &statbuf) < 0) return 0;
   return statbuf.st_blksize;
}
```

- Bei regulären Dateien lässt sich über das Feld *st_blksize* die vom zugehörigen Dateisystem für I/O-Operationen bevorzugte Blockgröße ermitteln.
- Diese liegt traditionellerweise bei 4096 oder 8192 Bytes.
- Bei neueren Dateisystemen wie etwa ZFS kann diese Blockgröße bei jeder Datei unterschiedlich (und sehr viel größer) sein.

lscan.c

```
off_t pos = 0;
ssize_t llen = 0; /* length of current line */
off_t blockpos = pos; /* keep track of current position */
ssize_t nbytes;
while ((nbytes = read(fd, buf, blocksize)) > 0) {
   if (out >= 0) {
      ssize_t written = write(out, buf, nbytes);
      if (written < nbytes) return false;
   }
   for (char* cp = buf; cp < buf + nbytes; ++cp) {
      if (*cp == '\n') {
         add_lpos(list, pos, llen);
         pos = blockpos + cp - buf + 1;
         llen = 0;
      } else {
         ++llen;
      }
   }
   blockpos += nbytes;
}
if (nbytes < 0) return false;
if (llen) add_lpos(list, pos, llen);
return true;
```

shuffle.c

```
#include <errno.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stralloc.h>
#include <string.h>
#include "lscan.h"
#include "lposlist.h"
#include "rgen.h"
#include "tmpfile.h"

char* cmdname;
static void memerr() { /* ... */ }
static void die(char* filename, char* message) { /* ... */ }
static int select_line(int noflines) { /* ... */ }
static void print_line(int fd, off_t pos, ssize_t len) { /* ... */ }
static bool seekable(int fd) { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

shuffle.c

```
static bool seekable(int fd) {
   return lseek(fd, 0, SEEK_CUR) >= 0;
}

int main(int argc, char* argv[]) {
   cmdname = argv[0];
   int fd = 0;
   int out = -1;
   if (!seekable(fd)) {
      out = get_tmpfile();
      if (out < 0) die("tmpfile", 0);
   }
   lposlist list = {0};
   if (!scan_lines(fd, out, &list)) die("scan_lines", 0);
   if (out >= 0) fd = out;
   while (list.length > 0) {
      int i = select_line(list.length);
      print_line(fd, list.line[i].pos, list.line[i].len);
      int j = list.length - 1;
      if (i != j) list.line[i] = list.line[j];
      --list.length;
   }
}
```

shuffle.c

```
static int select_line(int noflines) {
   int selected;
   if (!get_rval(&selected)) die("get_rval", 0);
   if (selected < 0) {
      selected = -(selected+1);
   }
   selected %= noflines;
   return selected;
}
```

- Zu beachten ist hier, dass die Werte von *get_rval* negativ sein können und der Modulo-Operator in C nicht genügt, um den Wert in den Bereich $[0, \text{noflines} - 1)$ zu bringen.

shuffle.c

```
static void print_line(int fd, off_t pos, ssize_t len) {
   char buf[len+1];
   ssize_t copied = 0;
   ssize_t nbytes;
   if (len > 0) {
      if (lseek(fd, pos, SEEK_SET) < 0) die("lseek", 0);
      while (copied < len &&
            (nbytes = read(fd, buf + copied, len - copied)) > 0) {
         copied += nbytes;
      }
      if (nbytes < 0) die("read", 0);
      if (nbytes == 0) die("read", "unexpected end of file");
   }
   buf[len++] = '\n';
   copied = 0;
   while (copied < len &&
         (nbytes = write(1, buf + copied, len - copied)) > 0) {
      copied += nbytes;
   }
   if (nbytes < 0) die("write", 0);
}
```

shuffle.c

```
if (lseek(fd, pos, SEEK_SET) < 0) die("lseek", 0);
```

- Der Systemaufruf *lseek* hat drei Parameter: Den Dateideskriptor, eine relative Position und die Angabe wozu die Position relativ ist.
- Für den dritten Parameter gibt es folgende Varianten:

  SEEK_SET: Die Positionsangabe wird relativ zur Position 0, also absolut interpretiert.

  SEEK_CUR: Die Positionsangabe wird relativ zur aktuellen Dateiposition interpretiert.

  SEEK_END: Die Positionsangabe wird relativ zum Ende der Datei interpretiert.

- Die Funktion *lseek* liefert entweder einen Fehler zurück (-1) oder die aktuelle Position nach der durchgeführten Operation.