Systemnahe Software
WS 2006/2007

Andreas F. Borchert
Universität Ulm

12. Februar 2007

Eine kleine Anwendung: shuffle

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.

Eine kleine Anwendung: shuffle

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.

Die Datenstruktur für die Zeilen

lposlist.h

```
#ifndef LPOSLIST_H
#define LPOSLIST_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;
   int length;
   lpos* line;
} lposlist;

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

Die Datenstruktur für die Zeilen

lposlist.c

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

int 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 0;
      list->line = new; list->allocated = allocated;
   }
   list->line[list->length++] = (lpos) {pos, len};
   return 1;
}
```

- 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.

Zufallszahlen aus dem Betriebssystem

rval.h

```
#ifndef RGEN_H
#define RGEN_H

int 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.

Zufallszahlen aus dem Betriebssystem

rval.c

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

int get_rval(int* rval) {
   static int fd = 0;
   if (fd == 0) {
      fd = open("/dev/urandom", O_RDONLY);
      if (fd < 0) return 0;
   }
   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.

Das Anlegen temporärer Dateien

tmpfile.h

```
#ifndef TMPFILE_H
#define TMPFILE_H

int get_tmpfile();
#endif
```

- Es ist möglich, eine Datei zu kreieren, 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.

Das Anlegen temporärer Dateien

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 0;
      int rval;
      if (!get_rval(&rval)) return 0;
      if (!stralloc_catint(&tmpfile, rval)) return 0;
      if (!stralloc_0(&tmpfile)) return 0;
      int outfd = open(tmpfile.s, O_RDWR|O_CREAT|O_EXCL, 0);
      if (outfd >= 0) {
         if (unlink(tmpfile.s) < 0) return 0;
         return outfd;
      }
   }
   return -1;
}
```

Das Anlegen temporärer Dateien

tmpfile.c

```
for (int attempt = 0; attempt < 10; ++attempt) {
   if (!stralloc_copys(&tmpfile, "/tmp/tmp.")) return 0;
   int rval;
   if (!get_rval(&rval)) return 0;
   if (!stralloc_catint(&tmpfile, rval)) return 0;
   if (!stralloc_0(&tmpfile)) return 0;
   int outfd = open(tmpfile.s, O_RDWR|O_CREAT|O_EXCL, 0);
   if (outfd >= 0) {
      if (unlink(tmpfile.s) < 0) return 0;
      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 müssen aber in jedem Falle einkalkuliert werden.

Der erste Durchgang

lscan.h

```
#ifndef LSCAN_H
#define LSCAN_H

#include "lposlist.h"

int 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.

Der erste Durchgang

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;
}

int 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 0;

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

Der erste Durchgang

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 bevorzugte Blockgrösse ermitteln.
- Diese liegt typischerweise bei 4096 oder 8192 Bytes.

Der erste Durchgang

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 0;
   }
   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 0;
if (llen) add_lpos(list, pos, llen);
return 1;
```

Das Hauptprogramm

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[]) { /* ... */ }
```

Das Hauptprogramm

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;
   }
}
```

Das Hauptprogramm

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.

Das Hauptprogramm

shuffle.c

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

Das Positionieren in einer Datei

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.