Musterlösung zu Übung 6

1 Aufgabe 11

Die Musterlösung finden Sie in der Datei `spsh-ue06-LOESUNG.c` im Archiv `http://ohm.hgesser.de/sp-ss2014/prakt/sp-ss2014-ue06-loesung.tgz`. Hier finden Sie Erläuterungen zu den neuen Code-Teilen.

Wir können die bis zu neun Pipes (für zehn Prozesse) in einem doppelten Array verwalten:

⟨*spsh mit pipes*⟩≡
```
int cmdno;
int pipe_fds[10][2];  // Pipe-Deskriptor-Paare; 9 reichen auch
```

Aus `args`, `cmd`, `pid` und `noargs` müssen Arrays werden (bzw. bei `args` aus dem eindimensionalen Array ein zweidimensionales):

⟨*spsh mit pipes*⟩+≡
```
char *args[10][10];   // pro Kommando max. 10 Argumente
char *cmds[10];       // max. 10 Kommandos
short no_args[10];    // Zahl der Argumente fuer die Kommandos
int pids[10];         // max. 10 Kind-PIDs
```

Die folgende Funktion `logdup` dient nur dem Debugging; wir geben damit aus, welche `dup2`-Aufrufe wir ausführen:

⟨*spsh mit pipes*⟩+≡
```
void logdup (int pid, int pip, int rw, int fd) {
  // Funktion protokolliert dup2()-Aufrufe
  // ersten Befehl aktivieren, um Ausgabe zu unterdruecken
  printf ("process %d: dup2 (pipe_fds[%d][%d], %d)\n",
    pid, pip, rw, fd);
  return;
};
```

Die separaten Kommandos, die zu einer Pipeline kombiniert werden sollen, könnten wir ähnlich wie bei der Argument-Trennung mit `strtok` zerlegen und dazu erst `"` als Trenner verwenden. Wenn das erledigt ist, würde für jedes Kommando die schon bekannte Aufteilung in Kommandoname und Argumentliste folgen. Wir verwenden eine alternative Lösung, bei der wir direkt die ganze Befehlszeile zerlegen und darauf achten, ob einer der Teile gleich `'|'` ist. Es gibt immer mehrere Lösungswege.

Das Grundprinzip der Lösung für das richtige Erzeugen der Pipe ist folgendes:

- Die Shell erzeugt zunächst die ganzen Pipelines.
- Dann erzeugt sie die bis zu zehn Sohnprozesse und leitet in jedem dieser Prozesse die Standardeingabe, die Standardausgabe oder beide mit `dup2` auf ein lesendes oder schreibendes Ende einer Pipeline um.

- Danach müssen sämtliche Pipeline-Deskriptoren geschlossen werden. (Das schließt nicht die Pipelines, die tatsächlich in Benutzung sind, weil ja die Deskriptoren mit `dup2` auf stdin/stdout umgeleitet wurden.)
- Schließlich starten wir mit `exec` das jeweils gewünschte Programm: Das funktioniert so, weil `exec` die Informationen über geöffnete Dateien nicht verändert, also bleiben auch die Pipes über `exec` hinaus aktiv.

Alle Prozesse, die zur Pipeline gehören, erben die ganzen geöffneten File-Deskriptoren der bis zu neun Pipes. Für das Schließen aller nicht mehr benötigten Deskriptoren stellen wir diese Funktion zur Verfügung, die wir unmittelbar vor dem `exec`-Aufruf verwenden (`cmdno` ist eine globale Variable):

⟨*spsh mit pipes*⟩+≡
```
void close_all_pipes () {
  // schliesst alle Pipes
  int i;
  for (i=0; i<cmdno; i++) {
    close (pipe_fds[i][0]);
    close (pipe_fds[i][1]);
  };
  return;
};
```

In der Schleife, die Kommandos interpretiert und ausführt, sind nun einige Anpassungen nötig. Den Code für die Verwaltung von Foreground- und Background-Prozessen habe ich hier entfernt.

⟨*spsh mit pipes*⟩+≡
```
int main () {

  ...

  while (1) {
    printf ("spsh$ ");
    fgets (command, sizeof(command), stdin);
    // Aus Eingabe \n abschneiden
    command[strlen(command)-1] = (char) 0;

    cmdno = 0;
    no_args[0] = 0;

    part = strtok (command, seps);
    if (part == NULL) continue;  // kein fork/exec

    while ( part != NULL ) {
      if (!strcmp(part,"|")) {
        // | gefunden
        cmdno++;
        no_args[cmdno] = 0;
      } else {
```

```
    // aktuelles Kommando weiter bearbeiten
    args[cmdno][no_args[cmdno]] = part;
    no_args[cmdno]++;
  }
  part = strtok (NULL, seps);
};

if (!strcmp(args[0][0],"exit"))   exit(0);  // Ende?

// Pipes erzeugen, n Prozesse brauchen n-1 Pipes
int i;
for (i=0; i<cmdno; i++) {
  printf ("create pipe %d\n", i);
  pipe (pipe_fds[i]);
};

// Argumentlisten der Kommandos terminieren
for (i=0; i<cmdno+1; i++)
  args[i][no_args[i]] = NULL;
```

Jetzt folgt die entscheidende Fallunterscheidung, die prüft, ob es sich um den ersten, einen mittleren oder den letzten Prozess der Pipe handelt. Im ersten Fall wird nur `stdout` umgebogen, im letzten Fall nur `stdin` und ansonsten beide. Jeweils nach dem oder den nötigen `dup2`-Aufruf(en) werden mit `close_all_pipes` alle Pipe-Dateideskriptoren geschlossen. Nach der Fallunterscheidung wird das jeweilige Kommando mit `exec` ausgeführt.

⟨*spsh mit pipes*⟩+≡

```
for (i=0; i<cmdno+1; i++) {
  pids[i] = fork();
  if ( pids[i] == 0 ) {
    // Kindprozess
    // Pipes bearbeiten
    if ((i==0) && (cmdno>0)) {
      // Fall 1: erster Prozess
      logdup (i, 0,1,1);
      dup2 (pipe_fds[0][1],1);  // Write-Ende von pipe 0
      close_all_pipes ();
    };
    if ((i>0) && (i<cmdno)) {
      // Fall 2: Prozess "in der Mitte"
      logdup (i, i-1,0,0);
      logdup (i, i,1,1);
      dup2 (pipe_fds[i-1][0],0);  // Read-Ende von pipe i-1
      dup2 (pipe_fds[i]  [1],1);  // Write-Ende von pipe i
      close_all_pipes ();
    };
    if ((i==cmdno) && (cmdno>0)) {
      // Fall 3: letzter Prozess
      logdup (i, i-1,0,0);
      dup2 (pipe_fds[i-1][0],0);  // Read-Ende von pipe i-1
      close_all_pipes ();
```

```
        };

        execvp (args[i][0], args[i]);
        // exec fehlgeschlagen?
        printf ("%s not found\n", args[i][0]);
        exit(0);
      };
    }; // ends for loop
```

Auch im Shell-Prozess müssen nun die Pipes alle geschlossen werden:

⟨*spsh mit pipes*⟩+≡

```
    // folgender Code nur im Originalprozess
    close_all_pipes();
    printf ("Processes launched: ");
    for (i=0; i<cmdno+1; i++) {
      printf ("%d, ", pids[i]);
    };
  }
}
```

2 Aufgabe 12

Die Umstellung auf der kleinen Datenbankanwendung auf `mmap` sollte aus dem folgenden Code weitestgehend selbsterklärend sein. Die ursprünglichen Befehle aus der älteren Programmversion stehen auskommentiert weiter im Text, so dass Sie sehen können, wie aus Blocklese- und Blockschreib-Operationen Speicherzugriffe geworden sind.

Wichtig ist, dass die Typdefinition (`typedef struct { ... } record;`) für den Pointer verwendet wird (`record *database`), der im `mmap`-Aufruf auf die Startadresse des gemappten Bereichs gesetzt wird.

⟨*records mit mmap*⟩≡

```
// record-mmap.c
// Simple Adress-Datenbank, Variante mit mmap()
// Hans-Georg Esser, 07.05.2013

#include <string.h>  // strcpy, strcmp
#include <fcntl.h>   // open
#include <stdio.h>   // printf, scanf
#include <unistd.h>  // lseek
#include <sys/stat.h>    // S_IWUSR etc.
#include <sys/mman.h>    // fuer mmap

typedef struct {
  int  id : 15;      // 15 Bits
  int  deleted : 1;  // 1 Bit
  char name[30];
  char vorname[30];
```

```
  char telefon[30];
  char gebdatum[11];
} record;

int recsize = sizeof(record);

record *database;    // Darueber sprechen wir die DB im Speicher an
record *database_start;

int main () {
  record r;
  int fd;
  char filename[]="adressen.dat";
  char action[100];

  strncpy (r.name, "Mueller", 29);
  strncpy (r.vorname, "Anton", 29);
  strncpy (r.telefon, "0172/12345", 29);
  strncpy (r.gebdatum, "25.12.1980", 10);
  r.deleted = 0;

  int i;
  fd = open (filename, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);

  if ( lseek(fd,0,SEEK_END) == 0 ) {
    // Datei ist leer; Standardinhalte erzeugen
    printf ("open: Initialisiere Daten\n");
    for (i=0; i<100; i++) {
      r.id = i;
      write (fd, &r, recsize);
    }
  } else printf ("open: Daten existieren schon\n");
```

Jetzt wird das Mapping eingerichtet:

⟨*records mit mmap*⟩+≡

```
  // ab hier Aenderungen (mit mmap)
  database_start = mmap (0, 100*sizeof(record), PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, 0);
  close (fd);   // brauchen offene Datei nach Mapping nicht mehr
  database = database_start;
```

und ab jetzt können wir über `database[i]` auf den `i`-ten Eintrag in der Datei zugreifen.

⟨*records mit mmap*⟩+≡

```
  printf ("Aktionen: show n, edit n, del n, new 0, search 0, quit 0\n");

  while (1) {
    printf ("Aktion Nummer: ");
    scanf ("%s %d", (char*)&action, &i);
```

```
// action: show
if (!strcmp((char*)&action, "show")) {
  // lseek (fd, i*recsize, SEEK_SET);
  // read (fd, &r, recsize);
  r = database[i];
  if (!r.deleted) {
    printf ("ID:   %d\n", r.id);
    printf ("Name: %s %s\n", r.vorname, r.name);
    printf ("Tel.: %s\n", r.telefon);
    printf ("Geb.: %s\n", r.gebdatum);
  } else printf ("Record %d deleted\n", i);
};

// action: edit
if (!strcmp((char*)&action, "edit")) {
  // lseek (fd, i*recsize, SEEK_SET);
  // read (fd, &r, recsize);
  r = database[i];
  if (!r.deleted) {
    printf ("ID: %d, Name: %s %s\n", r.id, r.vorname, r.name);
    printf ("Eingabe: Name Vorname Telefon Geburtstag: ");
    scanf ("%s %s %s %s", r.name, r.vorname, r.telefon, r.gebdatum);
    // lseek (fd, i*recsize, SEEK_SET);
    // write (fd, &r, recsize);
    database[i] = r;
  } else printf ("Record %d deleted\n", i);
};

// action: delete
if (!strcmp((char*)&action, "del")) {
  // lseek (fd, i*recsize, SEEK_SET);
  // read (fd, &r, recsize);
  r = database[i];
  r.deleted = 1;
  // lseek (fd, i*recsize, SEEK_SET);
  // write (fd, &r, recsize);
  database[i] = r;
};
```

Auf eine Implementierung des `new`-Features haben wir hier verzichtet, denn mit `mmap` eingebundene Dateien lassen sich nicht so einfach vergrößern, was für das Anlegen eines neuen Records nötig wäre.

⟨*records mit mmap*⟩+≡

```
// action: search (Aufgabe 7 c)
if (!strcmp((char*)&action, "search")) {
  char search[100];
  int i;
  printf ("Suchbegriff: ");
  scanf ("%s", (char*)&search);
  // lseek (fd, 0, SEEK_SET);
  // while (read (fd, &r, recsize)) {
```

```
      for (i=0; i<100; i++) {
        r = database[i];
        if ((r.deleted == 0) &&
            ((strcmp(r.vorname, search) == 0) ||
             (strcmp(r.name, search) == 0)))
          printf ("Treffer in record %d, %s %s\n", r.id, r.vorname, r.name);
      }
    }

    // action: quit
    if (!strcmp((char*)&action, "quit")) {
      munmap (database, 100*sizeof(record));
      printf ("Programmende.\n");
      break;
    };

  };

};
```

Wie Sie sehen, wird im Wesentlichen aus `lseek(fd, index*recsize, recsize)`- und `read`- bzw. `write`-Zugriffen ein direkter Speicherzugriff auf `database[index]`.