Testat 2

Einführung in die Systemprogrammierung SS 2014

11. Juni 2014

Abgabedatum: 20:00, 08.07.2014

In diesem Testat entwickeln wir eine eigene Kommando-Shell. Dazu behandeln wir Grundlagen der UNIX-Prozeßverwaltung in einer Multi-Benutzer und Multi-Prozeß-Umgebung. Beachten Sie, daß die hier verwendeten Techniken UNIX-spezifisch sind.

Abgabemodalitäten

Das Ergebnisprogramm für dieses Testat muß per e-Mail an den jeweils betreuenden Tutor UND an den Dozenten eingesendet werden. Es gelten die gleichen Regeln zur Bearbeitung, Abgabe und Bewertung wie im ersten Testat:

a. Sie können die Aufgabe alleine oder zu zweit bearbeiten und einsenden.

b. Sie müssen bei der Abgabe Ihre Namen, Matrikelnummern, und E-Mailadressen angeben (als Kommentare im Quellcode).

c. Der eingesendete Code muß vollständig von Ihrem Team entwickelt worden sein.

d. Jedes Teammitglied muß in der Lage sein, den Code eigenständig zu erklären.

e. Wir behalten uns vor, Punktvergabe von persönlichen Einzelgesprächen abhängig zu machen.

f. Sie dürfen sich mit anderen Studierenden austauschen. Vermeiden Sie dabei das direkte Zeigen von Codefragmenten.

1 Aufgabenstellung

Eine Kommando-Shell (bzw. auch Befehls-Shell von engl. '*shell*', Schale) ist eine Kommunikationsplattform zwischen Systemnutzern und Programmen. Eine umgibt dabei die auszuführenden Anwendungen 'schalenartig'. In modernen Systemen finden sich zwei Arten von Kommando-Shells:

- Graphische Kommando-Shells, die meist auf Fenstersystemen oder gitterbasierten Auswahlsystemen aufbauen. Wesentlicher Vorteil dieser Systeme ist ihre leichte Erlernbarkeit und intuitive Verwendbarkeit auch für Benutzer mit wenig Informatik-Wissen.

- Text-basierte Kommando-Shells, wie die von Ihnen bisher verwendeten Kommandozeilensysteme (`bash`, `zsh`, `tcsh` etc.). Wesentliche Vorteile dieser Systeme sind, daß sie Befehle auf komplexe Arten und Weisen miteinander verknüpfen und leicht über Netzwerkverbindungen betrieben werden können.

Text-basierte Kommando-Shells sind einfacher zu implementieren. Wir werden daher in diesem Testat eine solche Shell entwickeln, und dabei einige Techniken der UNIX-Systemprogrammierung einsetzen.

Zur Vereinfachung Ihrer Implementierung sind mehrere Appendices angegeben, die UNIX-Befehle oder auch String-Operationen (Appendix I) beschreiben.

Prüfen Sie jeden Ihrer Schritte per `valgrind` auf Speicherfehler.

1. Schreiben Sie ein Programm, das die `libreadline`-Bibliothek aus Aufgabenblatt 6 verwendet, um Benutzereingaben zu lesen.

2. Zerlegen Sie die Eingabe-Zeichenkette in einzelne Lexeme, ähnlich wie der Lexer eines Übersetzers. Sie können dazu ein einfaches Verfahren verwenden, das die Grenzen von Lexemen als Übergang zwischen Leerzeichen und anderen Zeichen sieht. Dabei sollten die Zeichen „`!`“, „`?`“, „`;`“, und „`|`“ separat erkannt werden.

   Beispielsweise sollten Sie den folgenden String

   ```
   "Einfuehrung in      die Systemprogrammierung!17  "
   ```

   zerlegen in

   ```
   { "Einfuehrung", "in", "die", "Systemprogrammierung", "!", "17" }
   ```

   Sie können folgende Operation zur Hilfe verwenden:

   ```
   #include <ctype.h>

   // liefert 1 gdw 'c' ein Leerzeichen, Tabulatorzeichen etc. ist, und
   // sonst 0:
   int isblank(int c);
   ```

3. Schreiben Sie eine Funktion `execute`, die ein wie oben zerlegtes Zeichenketten-Array ‘ausführt’. Die Funktion soll zwei Parameter nehmen: `char **` (Zeiger auf ein Array von Zeichenketten) und `int` (Anzahl der relevanten Elemente in dem gegebenen Array).

   Implementieren Sie zunächst folgendes Verhalten: Wenn das Array keine Einträge hat, passiert nichts. Ansonsten passiert je nach erstem Eintrag folgendes:

   - ‘`echo`’: Alle folgenden Einträge werden ausgegeben, von einem Leerzeichen getrennt, und gefolgt von einem Zeilenumbruch ("\n").
   - ‘`exit`’: Die Kommandoshell wird beendet. Sie können dafür den Befehl `exit(int rc)` aus `<stdlib.h>` verwenden, der gleichwertig zu ‘`return rc;`’ in der Funktion `main()` ist und also Ihr Programm mit dem Rückgabewert `rc` beendet.
   - Ansonsten: Eine Fehlermeldung wird ausgegeben, um zu signalisieren, daß der angegebene Befehl unbekannt ist.

   Um Stringgleichheit zu prüfen, können Sie die Funktion `strcmp()` (Arbeitsblatt 6 bzw. Appendix I) verwenden.

4. Implementieren Sie zwei weitere fest eingebaute Befehle ‘`cd`’ und ‘`pwd`’, ähnlich ‘`echo`’:

   - ‘`cd` *d*’: Wechselt das aktuelle Arbeitsverzeichnis nach *d*. Wenn der Wechsel nicht erfolgreich war, soll eine entsprechende Fehlermeldung ausgegeben werden.
   - ‘`pwd`’: Druckt das aktuelle Arbeitsverzeichnis aus.

   Geben Sie eine Fehlermeldung aus, wenn die Anzahl der Parameter nicht mit den Erwartungen übereinstimmt.

   Appendix B führt die notwendigen Systemaufrufe auf, um diese Operationen durchzuführen.

5. Das Detailverhalten von Programmen wird oft durch *Umgebungsvariablen* bestimmt. Wenn Sie beispielsweis `/bin/ls /z` ausführen (angenommen, daß die Datei `/z` nicht existiert), erhalten Sie unterschiedliche Fehlermeldungen abhängig davon, ob Ihre Umgebungsvariable `LANG` auf ‘`de_DE`’ oder ‘`en_US`’ gesetzt ist. Diese spezielle Variable wird verwendet, um die Sprachpräferenzen der Nutzer zu speichern.

   Fügen Sie neue eingebaute Operationen hinzu, um die aktuellen Umgebungsvariablen zu manipulieren:

- ‘`getenv` *v*’: Gibt die Zuweisung der Variable *v* aus, falls vorhanden.
- ‘`setenv` *v* *w*’: Ändert die Zuweisung der Variable *v* auf *w*, bzw. legt eine neue solche Zuweisung an, falls v noch nicht belegt war.
- ‘`setenv` *v*’: Löscht jegliche existierende Zuweisung an die Variable *v*.
- ‘`env`’: Gibt eine Liste aller Variablenzuweisungen aus, mit einer Zuweisung pro Zeile.

Appendix C beschreibt Operationen, um Umgebungsvariablen zu manipulieren.

Da Sie nun schon einen größeren Katalog an eingebauten Operationen haben, kann es unübersichtlich werden, wenn Sie alle diese Operationen in einem großen **`if`** / **`else`** – Block implementieren. Verwenden Sie daher eine Tabelle, in Form eines Arrays von **`struct`**-Elementen, so daß jedes **`struct`**-Element den Namen einer eingebauten Operation und einen Funktionszeiger auf eine Funktion zum Behandeln der Operation beinhaltet, z.B.

```
struct eingebaute_operationen { ... };
struct eingebaute_operationen eingebaute_ops[] = {
  { "echo", &meine_echo_funktion },
    ...
  { "setenv", &meine_setenv_funktion }
};
```

Sie können dem **`struct`** auch noch weitere Informationen mitgeben.

6. Erweitern Sie Ihr Programm so, daß es Kindprozesse starten kann (Appendix D). Wenn Ihre interne Funktion `execute` einen Befehl nicht als eingebauten Befehl erkennt, soll Ihre Shell nun statt Ausgabe einer Fehlermeldung versuchen, einen Kindprozeß mit dem betreffenden Namen und eventuellen Parametern zu starten.

   Sie können zur Vereinfachung davon ausgehen, daß der vollständige Name jedes Kindprozesses angegeben werden muß, also z.B. ‘`/bin/ls`’, um die Inhalte des aktuellen Verzeichnisses auszugeben. `execve` kann ebenfalls mit relativen Pfaden umgehen, so daß Sie z.B. auch ‘`./matrix-multiply`’ angeben könnten, um das betreffende Programm vom 7. Arbeitsblatt auszuführen.

   Beispielsweise sollte die Eingabe ‘`/usr/bin/ldd /bin/cat`’ dazu führen, daß `ldd` alle dynamischen Abhängikeiten von `/bin/cat` ausgibt (also dies als seinen Parameter erkennt).

   Ihr Programm soll auf das Ende des Kindprozesses warten, bevor es die Kontrolle zurück an die Benutzereingabe gibt.

7. Eine der Stärken von Kommandoshells ist die Fähigkeit, die Eingabe eines Prozesses an einen anderen Prozeß weiterzuleiten.

   So kann beispielsweise folgende Kommandozeile aufsteigend die Häufigkeit aller Worte in einer Textdatei ‘`text`’ berechnen:

   cat text | tr "␣" "\n" | tr −d " ,.()[]; " | sort | uniq −c | sort −n

   Dabei gibt ‘cat text’ die Datei `text` an die Standardausgabe aus. Die Standardausgabe von `cat` wird dann an das nächste Programm weitergeleitet. Dort wird die Ausgabe des vorherigen Programms zur Eingabe, und das Folgeprogramm produziert eine neue Ausgabe usw.:

   - ‘tr "␣" "\n"’ ersetzt alle Leerzeichen durch Zeilenumbrüche, so daß nun jedes Wort in einer eigenen Zeile steht
   - ‘tr −d " ,.()[]; "’ entfernt alle Klammern, Kommas, Punkte, und Semikolons
   - ‘`sort`’ sortiert die eingegangenen Worte alphabetisch
   - ‘`uniq -c`’ entfernt unmittelbar aufeinanderfolgende Duplikate und gibt zu jedem Wort die Anzahl der Vorkommnisse aus

- '`sort -n`' sortiert das Resultat numerisch aufsteigend.

Das Weiterreichen wird dabei durch as '`|`'-Symbol ('pipe' bzw. 'Rohrleitung') signalisiert.

Appendix E beschreibt die 'Standardeingabe' und 'Standardausgabe', und Appendix F beschreibt, wie Sie diese nutzen können, um Informationen zwischen mehreren Kindprozessen weiterzureichen.

Erweitern Sie Ihr Programm so, daß es mindestens zwischen zwei Programmen auf die oben genannte Art und Weise Text zwischen Standardeingabe und Standardausgabe weiterreichen kann; Programme sollten sich in dieser Hinsicht genauso verhalten wie auf der normalen Kommandozeile.

Beachten Sie, daß die Anführungszeichen (") im obigen Beispiel eine besondere Bedeutung in den existierenden Shell-Implementierungen haben; so erlaubt das "␣" von 'tr "␣" "\n"' beispielsweise, eine Zeichenkette, die nur aus einem Leerzeichen besteht, als Parameter an den Kindprozeß zu übergeben. Sie müssen diese Fähigkeit *nicht* implementieren.

Sie können als Test 'cat /proc/cpuinfo | grep model' verwenden. Dies wird Ihnen den Prozessortyp Ihres Systems anzeigen. Bei Multikernsystemen erhalten Sie dabei mehrere Einträge.

8. Unterstützen Sie nun das Semikolon (`;`) als Separator. Ein Benutzer soll mehrere Befehle hintereinander eingeben können, durch ein Semikolon getrennt. Die Befehle sollen in der angegebenen Reihenfolge ausgeführt werden.

   Beispielsweise soll '`echo Hallo` ; `echo Welt`' hintereinander `Hallo` und `Welt` ausgeben, jeweils in einer neuen Zeile.

9. Entwickler verwenden oft mehrere Shell-Prozesse gleichzeitig. Somit kann z.B. eine Shell verwendet werden, um eine längere Übersetzung durchlaufen zu lassen, während eine zweite Shell zum Arbeiten an einem anderen Projekt verwendet wird. Da diese Shells aber oft nicht alle gleichzeitig sichtbar sind, wäre es nützlich, Nachrichten zwischen den Shells austauschen zu können.

   In dieser Teilaufgabe erweitern Sie unsere Kommandoshell um genau diese Fähigkeit:

   - Fügen Sie eine neue Operation '`!`' hinzu, die genau einen Parameter nimmt. Dieser Parameter wird als *Nachricht* an alle auf dem gleichen Rechner laufenden Shells des gleichen Benutzers versendet. Die Nachricht darf maximal 255 Zeichen lang sein; wenn sie länger ist, soll sie automatisch gekürzt werden.
   - Jede der laufenden Shells soll die Nachricht sobald möglich im `readline`-Prompt anzeigen.
   - Mehrere Nachrichten können angezeigt werden. Diese sollen jeweils in einem *Nachrichtenpuffer* gespeichert und durch ein Komma getrennt werden. Jede Nachricht wird im Nachrichtenpuffer nur ein Mal gespeichert.
   - Fügen Sie eine neue Operation '`ack`' hinzu, die den Nachrichtenpuffer löscht. Diese Operation beeinflußt nur die aktuelle Shell; die Nachrichtenpuffer anderer Shells werden nicht beeinflußt.

   Appendix G beschreibt, wie Sie solche Kommunikation über UNIX-Sockets implementieren können. Verwenden Sie als Bezeichner für die Sockets einen Dateinamen im Heimatverzeichnis des Benutzers, statt '`/tmp/server`' wie im Appendix. Sie können über `getenv("HOME")` (Appendix C) das Heimatverzeichnis bestimmen und dann einen Dateinamen anhängen, z.B. über die Funktion `strcat` (Appendix I).

   Appendix H beschreibt, wie Sie diese Kommunikation mit `readline` integrieren können.

   Führen Sie bei allen verwendeten Operationen eine angemessene Fehlerbehandlung durch, die zumindest unerwartete Fehler ausgibt und das Programm abbricht.

Wichtig: Um die Nachrichten jederzeit empfangen und verteilen zu können, muß ein separater, nur dafür zuständiger Prozeß laufen. Dieser *Verteilerprozeß* muß bei Bedarf automatisch gestartet werden[1].

10. Unsere neugewonnene Fähigkeit, Nachrichten zu versenden, wollen wir nun auch in Shell-Befehlen nutzen können. Erweitern Sie die Shell um die Fähigkeit, auf Signale zu warten, indem Sie einen eingebauten Befehl '`?`' hinzufügen. Der Befehl nimmt einen Parameter n und hält die Ausführung der Shell solange an, bis eine Nachricht n im Nachrichtenpuffer liegt. Dann wird die Nachricht aus dem Puffer entfernt und die Ausführung fortgesetzt. Beispielsweise soll

    '`echo A`; ? foo; `B`'

    zunächst `A` ausgeben, und dann warten. Wenn wir in einer anderen Shell nun '`! foo`' ausführen, soll in der ersten Shell die Ausführung fortgesetzt werden: `B` wird ausgegeben und die Befehlseingabe wieder aktiviert.

    Sie dürfen beim Warten auf Nachrichten optional eine kurze Information ausgeben, um anzukündigen, daß Sie auf eine Nachricht warten bzw. auf welche Nachricht Sie warten.

    Alle Nachrichten *außer* der Nachricht, auf die wir explizit warten, sollen dabei regulär dem Nachrichtenpuffer hinzugefügt werden.

In den folgenden Anhängen erhalten Sie technische Informationen zu einigen Operationen, die zum Lösen der Aufgaben nötig sind. Sie können die meisten hier angegebenen Operationen im UNIX-Nutzerhandbuch nachschlagen, indem Sie auf der Kommandozeile '`man` ***befehlsname***' ausführen. `man` druckt daraufhin Hilfe zu dem Befehl aus. Sie können mit `q` die Anzeige beenden.

Einige Befehle, wie `printf`, existieren sowohl als C-Befehle als auch als Kommandozeilen-Programme. Wenn Sie mit `man` nachschlagen, finden Sie meist das Kommandozeilen-Programm und nicht den C-Befehl. Sie müssen in diesem Fall `man` explizit das Kapitel mitteilen, in dem Sie nachschlagen wollen:

- `man 2` *befehlsname* für Systemaufrufe, und
- `man 3` *befehlsname* für C-Bibliotheksaufrufe.

Eine Verwendung von `man` sollte allerdings nicht nötig sein.

A Fehler unter UNIX

Viele der im Appendix angegebenen Operationen können *Fehlercodes* zurückgeben, um zu signalisieren, daß eine Operation fehlgeschlagen ist. Um bei Erhalt eines Fehlercodes eine verständliche Fehlerausgabe zu erzeugen, können Sie die folgende Operation verwenden:

```
#include <stdio.h>

void perror(const char *s);
```

'`perror(s)`' gibt zunächst die Zeichenkette `s` aus, gefolgt von einer Beschreibung des letzten beobachteten Fehlers. Sie müssen `perror` also unmittelbar nach dem Fehler ausführen, da sonst der Fehlerzustand verloren geht.

B Pfadoperationen

Um den Pfad zu insipizieren und zu manipulieren, können Sie folgende Operationen verwenden:

[1]Zur korrekten Operation wäre es dazu nötig, einen *locking*-Mechanismus zu verwenden, um Wettlaufsituationen zwischen mehreren Shells, die gleichzeitig versuchen, ihren eigenen Verteilerprozeß zu starten, zu verhindern. Wir verzichten hier auf dieses Detail.

```c
#include <unistd.h>

// definiert PATH_MAX ... : Maximale Länge (Anzahl Zeichen) des Pfades

int chdir(const char *path); // gibt 0 bei Erfolg zurück
char *getcwd(char *buf, size_t size); // gibt 'buf' bei Erfolg zurück
```

- '`chdir(s)`' wechselt in den als Parameter angegebenen Pfad *s* (relative Pfade sind dabei erlaubt). Der Rückgabewert ist entweder 0 (Erfolg) oder −1 (Fehler, siehe Appendix A).
- '`getcwd(b, size)`' schreibt das aktuelle Arbeitsverzeichnis in den Speicherbereich ab Adresse *b*, bis zu *size* Bytes. Falls der Pfad länger als *size* ist oder ein anderer Fehler auftritt, liefert die Funktion `NULL` zurück, sonst zeigt sie Erfolg, indem Sie den Zeiger b zurückliefert.

C Umgebungsvariablen

```c
#include <stdlib.h>

char *getenv(const char *name);
int setenv(const char *name, const char *value, int overwrite);

extern char **environ; // Alle Umgebungsvariablen
```

- '`getenv(name)`' liest die Umgebungsvariable *name* aus. Liefert `NULL`, wenn die Variable nicht gesetzt ist.
- '`setenv(name, value, 1)`' setzt die Umgebungsvariable *name* auf den Wert *value*. Rückgabewert ist 0 (Erfolg) oder −1 (Fehler, siehe Appendix A).
- '`environ`' ist das Array aller Umgebungsvariablen und deren Werte. Der letzte Eintrag in diesem Array ist `NULL`, die davor stehenden Einträge sind Zeichenketten der Form '*n*=*v*', wobei *n* der Name der Umgebungsvariable und *v* ihr Wert ist.

D Neue Prozesse starten

Wie in der Vorlesung besprochen, werden unter UNIX Kind-Prozesse gestartet, indem zunächst ein neuer Prozeß erzeugt wird (`fork`). Der erzeugte Prozeß ist fast identisch mit dem Elternprozeß und ersetzt sich nun durch ein anderes Programm (`execve`).

```c
#include <unistd.h>

// Der Typ 'pid_t' ist ein 'int'-artiger Typ

pid_t fork(void);
int execve(const char *filename, char *const argv[],
           char *const envp[]);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- '`fork()`' spaltet den aktuellen Prozeß in zwei Prozesse. Eltern- und Kindprozeß erhalten dabei separate (aber zunächst identische) Speicherinhalte; Schreiboperationen in den Speicher sind also für den jeweils anderen Prozeß unsichtbar.

Der Rückgabewert unterscheidet sich bei beiden Prozessen: Beim Kindprozeß wird 0 zurückgeliefert, beim Elternprozeß die *Prozeßnummer* des Kindprozesses. Diese Nummer können wir z.B. verwenden, um auf das Ende des Kindprozesses zu warten (s.u.).

Wenn '`fork`' fehlschlägt, wird -1 zurückgeliefert (siehe Appendix A).

- '`execve(filename, argv, envp)`' ersetzt den Inhalt des aktuellen Prozesses durch das Programm in Datei `filename`. Zusätzliche Kommandozeilenparameter werden in `argv` angegeben. Dabei ist argv[0] der Name des Programmes selbst, argv[1] der erste Parameter usw. Der letzte Eintrag in `argv` muß `NULL` sein. `envp` speichert die Umgebungsvariablen ähnlich, im gleichen Format wie `environ` (Appendix C).

  `execvp` kehrt nur dann zum aufrufenden Programm zurück, wenn ein Fehler aufgetreten ist (Rückgabewert -1).

- '`wait(status)`' wartet, bis sich irgendein beliebiger Kindprozeß beendet hat. Diese Operation kann nach einem `fork()` verwendet werden, um den Elternprozeß schlafen zu legen, bis der Kindprozeß fertig ist.

  Rückgabewert ist die Prozeßnummer des beendeten Prozesses, oder -1 bei einem Fehler (Appendix A).

  Zusätzlich werden in die Speicherstelle, auf die `status` zeigt, Rückgabestatusinformationen abgelegt:

```
  int resultat;
  pid_t pid = wait(&resultat);
  if (pid < 0) {
    perror("wait");
  } else {
    printf("Kindprozess %d mit Rueckgabe %d beendet\n",
           pid, WEXITSTATUS(resultat));
    // Der WEXITSTATUS ist der Rückgabewert der main()-Funktion des
    // Kindprozesses bzw. der Parameter 'rc' in 'exit(rc)'.
}
```

  `status` darf aber auch NULL sein, falls diese Informationen nicht benötigt werden.

- '`waitpid(pid, status, options)`' ist eine spezialisiertere Variante von `wait`, die nur auf eine bestimmte Prozeßnummer wartet. Dabei ist `pid` die Nummer der Prozeßnummer, auf die gewartet werden soll, und `status` wie bei `wait`. Der Parameter `options` kann 0 oder `WNOHANG` sein. Mit 0 verhält es sich wie `wait`. Mit `WNOHANG` kehrt die Funktion sofort zurück, auch dann, wenn der betreffende Kindprozeß noch nicht beendet wurde.

  Rückgabewerte sind wie für `wait`, außer, daß mit `WNOHANG` auch 0 zurückgegeben werden kann, um anzugeben, daß kein Kindprozeß beendet wurde.

E Dateideskriptoren, Standard-Eingabe und Standard-Ausgabe

In UNIX werden Dateizugriffe durch sogenannte *Dateideskriptoren* realisiert. Diese sind `int`-artige nichtnegative Zahlen, die bei der Kommunikation mit dem Betriebssystemkern verwendet werden, um zu sagen, in welche Datei geschrieben bzw. von welcher Datei gelesen werden soll.

Diese Deskriptoren zeigen nicht notwendigerweise auf echte 'Dateien'. Sie werden auch für Netzwerkkommunikation und Kommunikation zwischen Prozessen verwendet.

Drei Dateideskriptoren sind vordefiniert und schon zu Beginn des Programmes offen:

```
#include <stdio.h>

// definiert:
```

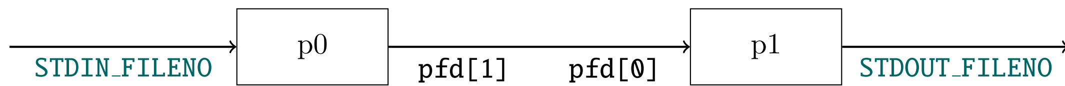

Abbildung 1: Datenleitungen zwischen zwei Prozessen p0 und p1. Hierbei wurde `pfd` durch `pipe(pfd)` erzeugt und entsprechend präpariert.

```
//    STDIN_FILENO:  Standardeingabe
//    STDOUT_FILENO:  Standardausgabe
//    STDERR_FILENO:  Standard-Fehlerausgabe
```

- `STDIN_FILENO` wird verwendet, um Benutzereingaben einzulesen, z.B. per `scanf`.
- `STDOUT_FILENO` wird von `printf` und ähnlichen Befehlen zur Ausgabe verwendet.
- `STDERR_FILENO` wird zur Ausgabe von Fehlermeldungen verwendet, die nicht zur regulären Ausgabe gehören sollen, z.B. für `perror`.

Beim Verwenden von `fork()` erben Kindprozesse die offenen Dateideskriptoren. Kindprozesse können also z.B. in die Ausgabe des Elternprozesses hineinschreiben.

F Datenleitungen zwischen Kindprozessen

Um Daten zwischen Prozessen hindurchzureichen, bietet UNIX das Konzept der '*pipes*' (Datenleitungen) an. Diese sind Paare von Dateideskriptoren, wobei einer davon nur zum Lesen und einer nur zum Schreiben verwendet wird.

In einem Prozeß ersetzt das eine Ende der Leitung die Standardausgabe, und im anderen Prozeß ersetzt das andere Ende der Leitung die Standardeingabe. Dies ist für die Prozesse selbst unsichtbar: sofern ein Programm von `STDIN_FILENO` (Appendix E) liest (z.B. mit `scanf()`) und nach `STDOUT_FILENO` schreibt (z.B. mit `printf()`), kann dessen Ein- und Ausgabe auf diese Weise beeinflußt werden.

Abbildung 1 zeigt, wie die betreffenden Dateideskriptoren eingesetzt werden.

Datenleitungen erfordern das Vorhandensein mehrerer Prozesse, die Dateideskriptoren untereinander austauschen können. Appendix D erklärt, wie wir dies per `fork()` erreichen können: Ein mit `fork()` erzeugter Kindprozeß erbt alle Dateideskriptoren des Elternprozesses.

Die nötigen Operationen für Datenleitungen sind folgende:

```
#include <unistd.h>

int pipe(int pipefd[2]);
int close(int fd);
int dup2(int oldfd, int newfd);
```

- '`pipe(`*`pfd`*`)`': Diese Operation erzeugt ein Paar von Datenleitungen. *`pfd`*`[0]` ist zum Lesen, und *`pfd`*`[1]` zum Schreiben. Die Leitungen funktionieren nur dann korrekt, wenn:
  - Je ein Ende einem Prozeß gehört und dort nur gelesen bzw. nur beschrieben wird (je nach Ende)
  - Das andere Ende im anderen Prozeß entsprechend geschlossen ist (s.u.).
- '`close(`*`fd`*`)`': Diese Operation schließt einen Dateideskriptor *`fd`*.
- '`dup2(`*`oldfd`*`, `*`newfd`*`)`': Diese Operation überschreibt einen existierenden Dateideskriptor *`oldfd`* mit einem anderen Dateideskriptor *`newfd`*. Auf diese Weise können insbesondere vor einem `execve` die Dateideskriptoren `STDIN_FILENO` oder `STDOUT_FILENO` mit dem Dateideskriptor einer Datenleitung überschrieben werden. Jegliche spätere Schreib- oder Leseoperationen auf *`oldfd`* werden vom Betriebssystemkern transparent nach *`newfd`* weitergeleitet.

`pipe` und `close` liefern bei Erfolg 0, sonst −1 (Fehlerausgabe: siehe Appendix A). `dup2` liefert im Erfolgsfall die Nummer des neuen Dateideskriptors, hat aber sonst identische Rückgabesemantik.

G Kommunikation über UNIX-Sockets

Sockets ('Steckdosen') sind ein allgemeinerer Kommunikationsmechanismus als Datenleitungen; sie werden unter anderem auch zur Netzwerkkommunikation verwendet. Wir verwenden Sie hier allerdings nur zur Kommunikation auf einem einzelnen Rechner, in der Form der sogenannten UNIX-Sockets.

Hier zunächst die relevanten Befehle zur Kommunikation mit UNIX-Sockets, gefolgt von einer Beschreibung, wie diese zusammen eingesetzt werden.

G.1 Socket-Erzeugung

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Ein Socket kann mit dem Befehl `socket` erzeugt weden. Das Betriebssystem unterstützt Sockets in verschiedenen *Domänen*; die wichtigsten davon sind `AF_INET` und `AF_INET6` (für das Inernet-Protokoll IP) sowie `AF_UNIX` für lokale Kommunikation auf UNIX-Rechnern (ohne Netzwerk). Internet-Sockets haben eine Socketnummer (*port*), die explizit beantragt oder vom Betriebssystem zugewiesen werden kann. Der Name des Sockets ergibt sich dann aus der IP-Adresse des Rechners zusammen mit dem gewählten Port.

UNIX-Sockets (Domäne `AF_UNIX`) werden stattdessen durch Namen im Dateisystem bezeichnet. Diese haben also das Erscheinungsbild von normalen Dateien, verhalten sich aber etwas anders. Sie können zum Testen zunächst den Namen `/tmp/server` verwenden.

Beim Aufruf von `socket` zum Erzeugen eines Sockets wird die Domäne als erster Parameter angegeben:

```
int socket(int domain, int type, int protocol);
```

`protocol` ist immer 0, und `type` gibt an, welche Art von Verbindung in dieser Domäne hergestellt werden soll. Die beiden wichtigsten Optionen sind `SOCK_STREAM` (geordnete, sichere Übertragung) und `SOCK_DGRAM` (ungeordnet, Pakete können verloren gehen). Wir verwenden `SOCK_STREAM` (`SOCK_DGRAM` ist primär für Internet-Verbindungen gedacht).

Falls die Erzeugung des Sockets erfolgreich war, liefert `socket` einen Dateideskriptor größer/gleich 0 zurück, sonst −1 (Appendix A).

G.2 Server: Socket-Benennung

Ein Socket selbst ist zunächst ein rein prozeßlokales Konzept. Um den Socket nach außen sichtbar zu machen, verwenden wir den Befehl `bind(2)`, der dem Socket einen globalen Namen verleiht. `bind` hat folgenden Prototypen:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

1. *`sockfd`* ist der zu benennende Socket.
2. *`addr`* zeigt auf eine Struktur, die den Namen des Sockets kodiert. Die genaue Struktur variiert je nach verwendeter Domäne. Bei `AF_UNIX` wird folgende Struktur verwendet:

   ```
   #include <sys/socket.h>
   #include <sys/un.h>

   struct sockaddr_un {
   ```

```
    sa_family_t  sun_family;                /* AF_UNIX */
    char         sun_path[UNIX_PATH_MAX];   /* pathname */
};
```

Dabei muß `sun_family` immer `AF_UNIX` sein, und `sun_path` enthält eine Zeichenkette mit einem Dateinamen, der dem Socket im Dateisystem zugewiesen wird (z.B. `/tmp/server`).

3. `addrlen` ist die Größe des Speicherbereiches von `addr`. Da verschiedene Socket-Arten verschiedene Socket-Strukturen verwenden, kann diese Größe variieren. Wir könnnen wie üblich **`sizeof`** verwenden, um die Größe der Struktur zu berechnen.

Im Erfolgsfall liefert die Funktion 0; andere Werte signalisieren einen Fehler.

G.3 Server: Socket zum Horchen konfigurieren

Um unseren Socket serverseitig verwenden zu können, müssen wir ihn noch mit dem Aufruf `listen` als einen Server-Socket konfigurieren:

```
int listen(int sockfd, int backlog);
```

Der Parameter `backlog` gibt dabei an, wieviel Platz das Betriebssystem als Puffer für eingehende Verbindungsanfragen reservieren soll, also wieviele eingehende Verbindungen in eine Warteschleife gesteckt werden können, während der Server arbeitet. Desto größer die Zahl, desto mehr Speicher wird benötigt. Kleine Zahlen sorgen hingegen dafür, daß Verbindungsgesuche abgelehnt werden können, wenn der Server gerade stark belastet ist.

Im Erfolgsfall liefert die Funktion 0; andere Werte signalisieren Fehler.

G.4 Server: Warten auf Verbindungen

Um nun auf eingehende Verbindungen zu warten, müssen wir warten, bis es von unserem Socket etwas zu lesen gibt. Dazu verwenden wir die Funktion `select` mit ihren Helferfunktionen:

```
#include <sys/select.h>

void FD_ZERO(fd_set *set);

void FD_SET(int fd, fd_set *set);
void FD_CLR(int fd, fd_set *set);

int  FD_ISSET(int fd, fd_set *set);

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

`select` ist eine Funktion, die auf mehrere Dateideskriptoren gleichzeitig warten kann. Das heißt, `select` nimmt eine Menge von Dateideskriptoren und unterbricht die Ausführungen des aktuellen Prozesses so lange, bis mindestens einer der Dateideskriptoren gelesen oder beschrieben werden kann, oder einen Ausnahmezustand angenommen hat. Um diese drei Fälle zu unterscheiden, werden `select` drei verschiedene Dateideskriptormengen übergeben, wobei wir hier nur den ersten davon, ***`readfds`*** benötigen und ***`writefds`*** und ***`exceptfds`*** auf `NULL` halten können.

Eine Dateideskriptormenge hat den Typ `fd_set`. Wir operieren auf diesen Typen mit den vier Helferfunktionen:

- `FD_ZERO(fd_set *set)` setzt `*set` auf die leere Menge.
- `FD_SET(int fd, fd_set *set)` fügt *`fd`* `*set` hinzu.
- `FD_CLR(int fd, fd_set *set)` entfernt *`fd`* aus `*set`.

- `FD_ISSET(int fd, fd_set *set)` überprüft, ob *fd* in **set* ist.

Wir werden in unserem Server nur auf die Lesbarkeit von Dateideskriptoren warten. Wenn wir passende Deskriptormengen erzeugt haben, können wir `select` aufrufen:

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Die Parameter sind wie folgt:

1. `ndfs` ist der höchste Dateideskriptor aus den drei folgenden Menge, plus eins.
2. `readfds` ist die Menge aller Dateideskriptoren, auf deren Lesebereitschaft wir warten.
3. `writefds` ist die Menge aller Dateideskriptoren, auf deren Schreibbereitschaft wir warten (`NULL` hier).
4. `exceptfds` ist die Menge aller Dateideskriptoren, auf deren Ausnahmezustände wir warten (`NULL` hier).
5. `timeout` erlaubt uns, eine Zeitspanne anzugeben, nach der `select` automatisch zu warten aufhören sollte. Wir können diese Option deaktivieren, indem wir `NULL` übergeben.

Die Rückgabe von `select` ist die Anzahl der Dateideskriptoren, die lesbar/schreibbar/in Ausnahmezustand sind. Die genauen betroffenen Deskriptoren werden in `readfds` `writefds` und `exceptfds` gesetzt und können mit `FD_ISSET` geprüft werden. `select` kann aber auch einen negativen Wert liefern, um einen Fehler oder eine Unterbrechung anzuzeigen.

G.5 Klient: Verbinden mit dem Server

Wir wissen nun genug, um den Server dazu zu bringen, auf Klienten zu warten. Um den Klienten allerdings dazu zu bringen, den Server zu kontaktieren, müssen wir Sockets etwas anders verwenden. Wir erzeugen dazu (wie zuvor) einen Socket per `socket` und rufen dann `connect` auf:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Die Parameter entsprechen denen von `bind` (Abschnitt G.2).

`connect` wartet, bis der Server die Verbindung akzeptiert hat, und liefert bei Erfolg 0 zurück (sonst -1 mit der üblichen Fehlerbehandlung). Aus Sicht des Klienten können wir nun an den Socket schreiben und von ihm lesen.

G.6 Server: Akzeptieren einer Verbindung

Auf der Server-Seite müssen wir also nun eingehende Verbindungen akzeptieren, sobald `select` die Kontrolle an das Programm zurückgibt und signalisiert, daß eine eingehende Verbindung vorliegt. Dazu erzeugen wir aus dem Socket, den wir auf die Adresse im Dateisystem `/tmp/server` gebunden haben, einen neuen Socket, der uns die Kommunkation mit dem Klienten erlaubt. Die dafür zuständige Funktion ist `accept(2)`:

```
int accept(int sockfd, const struct sockaddr *addr, socklen_t *addrlen);
```

Die Parameter entsprechen wieder denen von `bind` (Abschnitt G.2). `addr` und `addrlen` beziehen sich aber hier auf die Verbindungsdaten des Klienten, die in lokale Strukturen eingetragen werden. Auf diese Weise kann man bei einer `AF_INET`-Verbindung zum Beispiel die IP-Adresse des Klienten bestimmen. Wenn der Server sich nicht für diese Details interessiert, können die beiden Zeiger auf NULL gesetzt werden.

Im Erfolgsfall liefert `accept` eine nichtnegative Zahl, die ein Socket-Dateideskriptor ist, von dem wir lesen und auf den wir schreiben können.

Auf der Server-Seite erzeugen wir für jede eingehende Verbindungsanfrage einen solchen Socket per `accept`. Jeder derart akzeptierte Socket ist mit dem korrespondierenden Socket des Klienten direkt verbunden. Auf diese Weise kann sich eine große Menge von Sockets ansammeln, auf deren Eingänge wir per `select` warten.

G.7 Daten Senden

Zum Lesen und Schreiben von Daten auf Dateideskriptoren stehen folgende Befehle zur Verfügung:

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

- '`write(`*`fd`*`, `*`buf`*`, `*`count`*`)`' schreibt *`count`* aus dem Puffer *`buf`* in den Dateideskriptor *`fd`*.
- '`read(`*`fd`*`, `*`buf`*`, `*`count`*`)`' liest bis zu *`count`* Zeichen in den Puffer *`buf`*, aus dem Dateideskriptor *`fd`*. Die Funktion liefert zurück, wieviele Bytes tatsächlich gelesen wurden. Die Zahl kann gleich 0 sein, um auszudrücken, daß der Dateideskriptor *`fd`* am anderen Ende geschlossen wurde (z.B. wenn der Server eine Verbindung an den Klienten geschlossen hat und der Klient zu lesen versucht).

Beide Befehle geben eine negative Zahl zurück, wenn ein Fehler auftegreten ist.

G.8 Socket schließen

Sockets können genau wie Dateien auch mit `close()` geschlossen werden. Wenn ein Lese- oder Schreibbefehl auf einem Socket einen Fehler meldet, liegt dies meist daran, daß sich der Klient abgemeldet hat oder abgebrochen wurde; in diesem Fall ist es wichtig, den nicht mehr benötigten Socket freizugeben, damit er in späteren Verbindungen wiederverwertet werden kann– ansonsten gehen dem Prozeß früher oder später die Socketnummern aus.

G.9 Unterbrechungen

Wenn ein verbundener Socket geschlossen wird, markiert dies den Partner-Socket so, als ob er gelesen werden könnte. Wenn ein Klient seinen Socket schließt, wird dies also `select` in der Menge der lesbaren Dateideskriptoren gemeldet. Lese-Operationen auf dem betreffenden socket liefern die Ergebnislänge 0.

Zusammengefaßt könnte der Ablauf einer einfachen Verbindung (ohne Warten und Wiederholung) wie folgt aussehen:

SERVER		KLIENT
`sfd = socket(AF_UNIX, SOCK_STREAM, 0)`		`cfd = socket(AF_UNIX, SOCK_STREAM, 0)`
`bind(sfd, &sockaddr, sizeof(sockaddr))`		
`listen(sfd, 10)`		
`select(max_fd + 1, &readfd_set, NULL, NULL, NULL)`		
		`connect(cfd, &sockaddr, sizeof(sockaddr))`
`cconn = accept(sfd, NULL, NULL)`		
`read(cconn, &puffer, sizeof(puffer))`	⟸	`write(cfd, "Hallo", 6)`
`write(cconn, "Hallo", 6)`	⟹	`read(cfd, &puffer, sizeof(puffer))`
`close(cconn)`		`close(cfd)`
`unlink("/tmp/server")`		

Die temporär erzeugte Verbindungsdatei `/tmp/server` sollte beim Beenden des Servers per `unlink()` gelöscht werden (`unlink("foo")` 'löscht'[2] die Datei mit Namen `foo`).

[2] Genauer gesagt wird der betreffende Eintrag aus dem Dateisystem entfernt. Ob die zugehörigen Daten gelöscht werden, hängt von anderen Faktoren ab. Dies ist aber gleichwertig mit der umgangssprachlichen Bedeutung des „Löschens“ einer Datei.

H `select` und libreadline

Die Funktion `readline` geht davon aus, daß nur die Standardeingabe für die Ausführung des Programmes relevant ist. Wenn wir allerdings gleichzeitig auf Eingaben von anderen Dateideskriptoren warten, ist diese Annahme nicht mehr gültig. Für diesen Fall existiert eine alternative Schnittstelle zur Verwendung von libreadline. Diese haben wir für dieses Testat auf zwei Operationen vereinfacht.

In der Datei `http://web.sepl.cs.uni-frankfurt.de/2014-ss/b-sysp/rl-prompt.c` finden Sie die Helferfunktion `rl_change_prompt`, die Ihnen die Initialisierung von `readline` und das Ändern des Prompts abnimmt:

```c
/**
 * Setzt den Readline-Prompt und installiert 'process_input' als Funktion,
 * die aufgerufen wird, wenn eine readline-Eingabe abgeschlossen ist.
 *
 * @param prompt Der neue Prompt
 * @param preserve_user_input Ob beim Setzen des Prompts partielle
 *        Benutzereingaben erhalten bleiben (1)
 *        oder gelöscht werden (0) sollen
 * @param process_input Die Funktion, die später bei fertiger Eingabe
 *        aufgerufen wird
 */
void
rl_change_prompt(char *prompt, int preserve_user_input,
                 void(*process_input)(char *eingabe));
```

Sie können `rl_change_prompt` beliebig oft aufrufen. Die Parameterfunktion `process_input` wird bei abgeschlossener Eingabe zu einem späteren Zeitpunkt zurückgerufen (daher wird sie im Englischen auch als ein *callback* bezeichnet).

Um nach einer Initialisierung der `readline`-Bibliothek zu erlauben, Eingaben zu bearbeiten, können Sie nun die Funktion '`rl_callback_read_char()`' verwenden. Diese Funktion müssen Sie (ohne Parameter) jedesmal aufrufen, wenn auf `STDIN_FILENO` neue Eingaben verfügbar sind (z.B. gemäß `select`). Die Funktion bearbeitet diese neuen Eingaben, ruft bei Bedarf `process_input` auf (sofern installiert), und kehrt so schnell wie möglich zurück.

Dies erlaubt Ihnen, `select()` zum Warten auf UNIX-Sockets und andere Dateideskriptoren mit `readline` zu kombinieren.

I Zeichenketten-Befehle

```c
#include <string.h>

// Hängt 'src' hinten an 'dest' an und überschreibt dabei den
// Speicher hinter dem Ende von 'dest'.  Das terminierende Null-Byte
// wird entsprechend aktualisiert.
// Prüft nicht, daß genug Speicher verfügbar ist.  Kann also zu
// Speicherkorruption etc. führen.
char *strcat(char *dest, const char *src);

// Berechnet die Länge der Zeichenkette 'str'
size_t strlen(const char *str);

// Vergleicht 's1' mit 's2' und liefert einen Wert kleiner, gleich,
// oder größer 0 sofern 's1' jeweils kleiner, gleich, oder größer
// 's2' ist.
int strcmp(const char *s1, const char *s2);

// Kopiert String 'src' nach Adresse 'dest', inklusive terminierendem
// Null-Byte.
// Führt keine Speicherprüfung durch.
char *strcpy(char *dest, const char *src);
```