Übungen zu Systemnahe Programmierung in C (SPiC)

Moritz Strübe, Rainer Müller
(Lehrstuhl Informatik 4)

Sommersemester 2013

Inhalt

Linux
- Terminal
- Arbeitsumgebung
- Manual Pages

Fehlerbehandlung
- Bibliotheksfunktionen

Verzeichnisschnittstelle
- opendir, closedir, readdir
- Fehlerbehandlung bei readdir
- Verwendung von stat

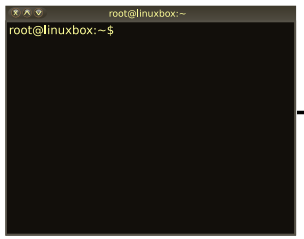
Terminal - historisches (etwas vereinfacht)

- Als die Computer noch größer waren:

Seriell — Computer

1

- Als das Internet noch langsam war:



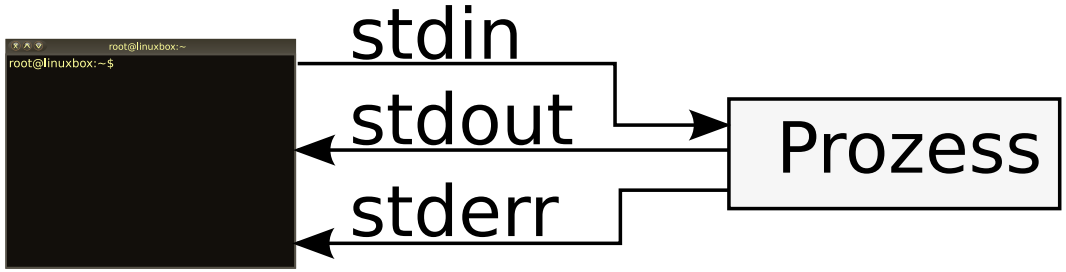
Netzwerk / DFÜ — Server

- Farben, Positionssprünge, etc werden durch spezielle Zeichenfolgen ermöglicht

[1] Televideo 925

Terminal - Funktionsweise

- Drei Kanäle:



- `stdin` Eingaben
- `stdout` Ausgaben
- `stderr` Fehlermeldungen

Terminal - stdout und stderr

- Beispiel `stdout` und `stderr`
  - Ausgabe in eine Datei schreiben

```
1 find . > ordner.txt
```

  - Vor allem unter Linux wird `stdout` häufig direkt mit `stdin` anderer Programme verbunden

```
1 cat ordner.txt | grep tmp | wc -l
```

- Vorteil von `stderr`:

⇒ Fehlermeldungen werden weiterhin am Terminal ausgegeben

Die Shell

- Wichtige Komandos
  - `cd` (change directory) Wechseln in ein Verzeichnis

```
1 cd /proj/i4spic/<login>/aufgabeX
```

  - `ls` (list directory) Verzeichnisinhalt auflisten

```
1 ls
```

  - `cp` (copy) Datei kopieren

```
1 cp /proj/i4spic/pub/aufgabeX/vorgabe.h /proj/i4spic/<login>/
    aufgabeX
2 # oder
3 cd /proj/i4spic/<login>/aufgabeX
4 cp /proj/i4spic/pub/aufgabeX/vorgabe.h .
```

Arbeitsumgebung

- Unter Linux:
  - Kate, gedit, Eclipse cdt, Vim, Emacs, ...
- Zugriff aus der Windows-Umgebung über SSH (nur Terminalfenster)
  - Editor unter Linux via SSH:
    - mcedit, nano, emacs, vim
  - Editor unter Windows:

    $\Rightarrow$ Dateizugriff über das Netzwerk
    - AVR-Studio ohne Projekt
    - Notepad++
  - Übersetzen und Test unter Linux (z.B. via Putty)
- Emulation der Linux-Umgebung unter Windows für daheim:
  - Cygwin/MinGW
  - Code::Blocks: IDE mit vorkonfiguriertem MinGW/gcc (Support im Forum)
  - Notepad++ und NppFTP
  - Wichtig: Auf jeden Fall auch (per SSH) im CIP testen!

Übersetzen

- spezielle Aufrufoptionen des Compilers
  - `-pedantic` liefert Warnungen in allen Fällen, die nicht 100% dem ANSI-C-Standard entsprechen
  - `-Wall` Warnt vor möglichen Fehlern (z.B.: `if(x = 7)`)
- diese Optionen führen zwar oft zu nervenden Warnungen, helfen aber auch dabei, Fehler schnell zu erkennen.
- Wir Testen die Abgaben mit:

```
1 gcc -pedantic -Wall -Werror -std=c99 -D_BSD_SOURCE -o print
      printdir.c
```

- `-Werror` wandelt Warnungen in Fehler um
- `-std=c99` Setzt verwendeten Standard auf C99
- `-D_BSD_SOURCE` Fügt unter anderem die POSIX Erweiterungen hinzu die in C99 nicht enthalten sind
- `-o print` Die Ausgabe wird in die Datei print geschrieben. Standardwert: `a.out`

Manual Pages

- Das Linux-Hilfesystem
- aufgeteilt nach verschiedenen Sections
  1. Kommandos
  2. Systemaufrufe
  3. Bibliotheksfunktionen
  5. Dateiformate (spezielle Datenstrukturen, etc.)
  7. verschiedenes (z.B. Terminaltreiber, IP, ...)
- man-Pages werden normalerweise mit der Section zitiert: `printf(3)`

```
1 # man [section] Begriff
2 man 3 printf
```

- Suche nach Sections: `man -f Begriff`
- Suche von man-Pages zu einem Stichwort: `man -k Stichwort`
- Alternativ: Webseiten, z.B. `http://die.net`

Fehlerursachen

- Fehler können aus unterschiedlichsten Gründen im Programm auftreten
  - Systemressourcen erschöpft

    ⇒ `malloc(3)` schlägt fehl
  - Fehlerhafte Benutzereingaben (z.B. nicht existierende Datei)

    ⇒ `fopen(3)` schlägt fehl
  - Transiente Fehler (z.B. nicht erreichbarer Server)

    ⇒ `connect(2)` schlägt fehl

Fehlerbehandlung

- Gute Software erkennt Fehler, führt eine angebrachte Behandlung durch und gibt eine aussagekräftige Fehlermeldung aus
- Kann das Programm trotz des Fehlers sinnvoll weiterlaufen?
- Beispiel 1: Ermittlung des Hostnamens zu einer IP-Adresse für Log

  $\Rightarrow$ Fehlerbehandlung: IP-Adresse im Log eintragen, Programm läuft weiter
- Beispiel 2: Öffnen einer zu kopierenden Datei schlägt fehl

  $\Rightarrow$ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden

  $\Rightarrow$ Oder den Kopiervorgang bei der nächsten Datei fortsetzen

  $\Rightarrow$ Entscheidung liegt beim Softwareentwickler

Fehler in Bibliotheksfunktionen

- Fehler treten häufig in Funktionen der C-Bibliothek auf
  - erkennbar i.d.R. am Rückgabewert (Manpage!)
- Die Fehlerursache wird meist über die globale Variable `errno` übermittelt
  - Bekanntmachung im Programm durch Einbinden von `errno.h`
  - Bibliotheksfunktionen setzen errno nur im Fehlerfall
  - Fehlercodes sind immer >0
  - Fehlercode für jeden möglichen Fehler (siehe `errno(3)`)
- Fehlercodes können mit `perror(3)` und `strerror(3)` ausgegeben bzw. in lesbare Strings umgewandelt werden

```
1  char *mem = malloc(...); /* malloc gibt im Fehlerfall */
2  if(NULL == mem) {        /* NULL zurück */
3          fprintf(stderr, "%s:%d: malloc failed with reason: %s\n",
4                          __FILE__, __LINE__-3, strerror(errno));
5          perror("malloc"); /* Alternative zu strerror + fprintf */
6          exit(EXIT_FAILURE); /* Programm mit Fehlercode beenden */
7  }
```

Erweiterte Fehlerbehandlung

- Signalisierung von Fehlern normalerweise durch Rückgabewert
- Nicht bei allen Funktionen möglich, z.B. `getchar(3)`

```
1 int c;
2 while ((c=getchar()) != EOF) { ... }
3 /* EOF oder Fehler? */
```

- Rückgabewert `EOF` sowohl im Fehlerfall als auch bei `End-of-File`
- Erkennung im Fall von I/O-Streams mit `ferror(3)` und `feof(3)`

```
1 int c;
2 while ((c=getchar()) != EOF) { ... }
3 /* EOF oder Fehler? */
4 if(ferror(stdin)) {
5         /* Fehler */
6         ...
7 }
```

opendir, closedir, readdir

- Funktions-Prototypen (details siehe Vorlesung)

```
1 #include <sys/types.h>
2 #include <dirent.h>
3 DIR *opendir(const char *dirname);
4 int closedir(DIR *dirp);
5 struct dirent *readdir(DIR *dirp);
```

- Rückgabewert von `readdir`
  - Zeiger auf Datenstruktur vom Typ `struct dirent`
  - `NULL`, wenn `EOF` erreicht wurde **oder** im Fehlerfall

  ⇝ bei `EOF` bleibt errno unverändert (auch wenn `errno != 0`), im Fehlerfall wird `errno` entsprechend gesetzt

Fehlerbehandlung bei readdir

- Fehlerprüfung durch Setzen und Prüfen von `errno`

```
1  #include <errno.h>
2  ...
3      struct dirent *ent;
4      while(1) {
5          errno = 0;
6          ent = readdir(...);
7          if(ent == NULL) break;
8          ... /* keine weiteren break-Statements in der Schleife */
9      }
10     /* EOF oder Fehler? */
11     if(errno != 0) {
12         /* Fehler */
13         ...
14     }
```

- `errno=0` unmittelbar vor Aufruf der problematischen Funktion

⇒ `errno` wird nur im Fehlerfall gesetzt und bleibt sonst evtl. unverändert

- Abfrage der errno unmittelbar nach Rückgabe des pot. Fehlerwerts

⇒ `errno` könnte sonst durch andere Funktion verändert werden

Fehlerbehandlung bei readdir

- Fehlerprüfung durch Setzen und Prüfen von `errno`

```
1  #include <errno.h>
2  ...
3      struct dirent *ent;
4      while(errno=0, (ent=readdir()) != NULL) {
5
6
7
8          ... /* keine weiteren break-Statements in der Schleife */
9      }
10     /* EOF oder Fehler? */
11     if(errno != 0) {
12         /* Fehler */
13         ...
14     }
```

- `errno=0` unmittelbar vor Aufruf der problematischen Funktion

⇒ `errno` wird nur im Fehlerfall gesetzt und bleibt sonst evtl. unverändert

- Abfrage der errno unmittelbar nach Rückgabe des pot. Fehlerwerts

⇒ `errno` könnte sonst durch andere Funktion verändert werden

Datei-Attribute ermitteln: stat

- `readdir(3)` liefert nur Name und Typ eines Verzeichniseintrags
- Weitere Attribute stehen im Inode
- `stat(2)` Funktions-Prototyp:

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 int stat(const char *path, struct stat *buf);
```

- Argumente:
  - `path`: Dateiname
  - `buf`: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert: `0` wenn OK, `-1` wenn Fehler
- Beispiel:

```
1 struct stat buf;
2 stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
3 printf("Inode-Nummer: %ld\n", buf.st_ino);
```

Das struct stat

- Ausgewählte Elemente
  - `dev_t st_dev` Gerätenummer (des Dateisystems) = Partitions-Id
  - `ino_t st_ino` Inodenummer (Tupel `st_dev`, `st_ino` eindeutig im System)
  - `mode_t st_mode` Dateimode, u.a. Zugriffs-Bits und Dateityp
  - `nlink_t st_nlink` Anzahl der (Hard-) Links auf den Inode
  - `uid_t st_uid` UID des Besitzers
  - `gid_t st_gid` GID der Dateigruppe
  - `dev_t st_rdev` DeviceID, nur für Character oder Blockdevices
  - `off_t st_size` Dateigröße in Bytes
  - `time_t st_atime` Zeit des letzten Zugriffs (in Sekunden seit 1.1.1970)
  - `time_t st_mtime` Zeit der letzten Veränderung (in Sekunden ...)
  - `time_t st_ctime` Zeit der letzten Änderung der Inode-Information (...)
  - `unsigned long st_blksize` Blockgröße des Dateisystems
  - `unsigned long st_blocks` Anzahl der von der Datei belegten Blöcke

stat / lstat: st_mode

- `st_mode` enthält Informationen über den Typ des Eintrags:
  - `S_IFMT` `0170000` bitmask for the file type bitfields
  - `S_IFSOCK` `0140000` socket
  - `S_IFLNK` `0120000` symbolic link
  - `S_IFREG` `0100000` regular file
  - `S_IFBLK` `0060000` block device
  - `S_IFDIR` `0040000` directory
  - `S_IFCHR` `0020000` character device
  - `S_IFIFO` `0010000` FIFO
- Zur einfacheren Auswertung werden Makros zur Verfügung gestellt:
  - `S_ISREG(m)` - is it a regular file?
  - `S_ISDIR(m)` - directory?
  - `S_ISCHR(m)` - character device?
  - `S_ISLNK(m)` - symbolic link?