

Übungen zu Systemnahe Programmierung in C (SPiC)

Peter Wägemann, Heiko Janker, Moritz Strübe, Rainer Müller
(Lehrstuhl Informatik 4)



Wintersemester 2014/2015



Inhalt

Abgaben

Linux

Fehlerbehandlung



Inhalt

Abgaben

Linux

Terminal

Arbeiten unter Linux

Arbeitsumgebung

Manual Pages

Fehlerbehandlung

Bibliotheksfunktionen



Abgegebene Aufgaben

- sb_timer_cancelAlarm() nicht notwendig single-shot Timer
- Fehlerfall müssen bei Unit-Tests beachtet werden
- Überprüfung durch Ausgabe auf Anzeige



Inhalt

Abgaben

Linux

Terminal

Arbeiten unter Linux

Arbeitsumgebung

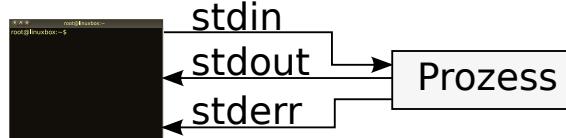
Manual Pages

Fehlerbehandlung



Terminal - Funktionsweise

- Drei Kanäle:



- stdin Eingaben
- stdout Ausgaben
- stderr Fehlermeldungen



Terminal - historisches (etwas vereinfacht)

- Als die Computer noch größer waren:



- Als das Internet noch langsam war:



- Farben, Positionssprünge, etc. werden durch spezielle Zeichenfolgen ermöglicht

¹Televideo 925

Terminal - stdio und stderr

- Beispiel stdio und stderr

- Ausgabe in eine Datei schreiben

1 `find . > ordner.txt`

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

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

- Vorteil von stderr:

⇒ Fehlermeldungen werden weiterhin am Terminal ausgegeben



Umleitungen

- > die Standardausgabe stdout umleiten
- >> Ausgabe an existierende Dateien anhängen
- 2> Stderr umleiten
- < Umleitung der Eingabe
- | Ausgabe eines Befehls direkt an einen anderen Befehl weiterleiten



Programme beenden

- Per Signal: **CTRL-C** (Kann vom Programm ignoriert werden)
- Von einer anderen Konsole aus: **killall cworld** beendet alle Programme mit dem Namen "cworld"
- Von der selben Konsole aus:
 - **CTRL-Z** hält den aktuell laufenden Prozess an
 - **killall cworld** beendet alle Programme mit dem Namen cworld
⇒ Programme anderer Benutzer dürfen nicht beendet werden
 - **fg** setzt den angehaltenen Prozess fort
- Wenn nichts mehr hilft: **killall -9 cworld**



Die Shell

- Wichtige Kommandos
 - **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 .**
 - **rm** (remove) Löschen
 - 1 **rm test1.c**
 - 2 # Ordner mit allen Dateien löschen
 - 3 **rm -r aufgabe1**



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:
 - ⇒ 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 (erlaubt das Editieren der Dateien im CIP)
 - Wichtig: Auf jeden Fall auch (per SSH) im CIP testen!



Übersetzen

- Wir Testen die Abgaben mit:

```
1 gcc -pedantic -Wall -Werror -O2 -std=c99 -D_XOPEN_SOURCE=500 -o ↵  
    ↵ printdir printdir.c
```

- spezielle Aufrufoptionen des Compilers

- -pedantic liefert Warnungen in allen Fällen, die nicht 100% dem verwendeten ANSI-C-Standard entsprechen
- -Wall Warnt vor möglichen Fehlern (z.B.: `if(x = 7)`)
- -Werror wandelt Warnungen in Fehler um
- diese Optionen führen zwar oft zu nervenden Warnungen, helfen aber auch dabei, Fehler schnell zu erkennen.
- -O2 führt zu Optimierungen des Programms
- -D_XOPEN_SOURCE=500 Fügt unter anderem die POSIX Erweiterungen hinzu die in C99 nicht enthalten sind
- -std=c99 Setzt verwendeten Standard auf C99
- -o print Die Ausgabe wird in die Datei print geschrieben.

Standardwert: a.out



Inhalt

Abgaben

Linux

Fehlerbehandlung

Bibliotheksfunktionen



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 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



- Gute Software erkennt Fehler:
 - führt eine angebrachte Behandlung durch
 - gibt aussagekräftige Fehlermeldungen aus
- Kann das Programm trotz des Fehlers sinnvoll weiterlaufen?
- Beispiel 1: Ermittlung des Hostnamens zu einer IP-Adresse für Log
⇒ Fehlerbehandlung: IP-Adresse im Log eintragen, Programm läuft weiter
- Beispiel 2: Öffnen einer zu kopierenden Datei schlägt fehl
 - ⇒ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden
 - ⇒ Oder den Kopiervorgang bei der nächsten Datei fortsetzen
 - ⇒ Entscheidung liegt beim Softwareentwickler



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 perror(3) und feof(3)

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



- 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 }
```



Debuggen

- ```
1 gcc -g -pedantic -Wall -Werror -O2 -std=c99 -D_XOPEN_SOURCE=500
```
- -g: aktiviert das Einfügen von Debug-Symbolen
  - gdb: Standard-Debugger
  - cgdb: "schönerer" Debugger
  - gdb ./a.out
  - cgdb --args ./a.out arg0 arg1 ...



- Strings:

"hallo" ≡ ↗ h a l l o \0

- `char* s = "hallo";`
- `strlen(hallo) == sizeof(s);`

- malloc()

```
1 void *malloc(size_t size);
2 void free(void *ptr);
```

- `malloc()` reserviert mindestens `size` Byte Speicher
- Der Speicher muss mit `free()` wieder freigegeben werden

- Was ist ein Segfault?

⇒ Zugriff auf Speicher der dem Prozess nicht zugeordnet ist  
≠ Speicher der reserviert ist

