

Übungen zu Systemnahe Programmierung in C (SPiC)

Sebastian Maier, Heiko Janker
(Lehrstuhl Informatik 4)

Übung 5



Wintersemester 2015/2016



Inhalt

Linux

Fehlerbehandlung

Aufgabe: cworld

Anhang



Linux

Terminal

Arbeiten unter Linux

Arbeitsumgebung

Manual Pages

Fehlerbehandlung

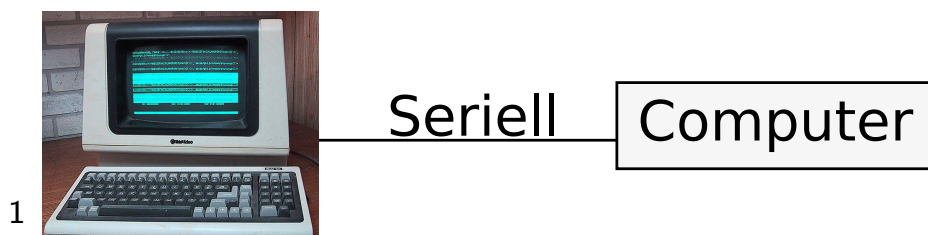
Aufgabe: cworld

Anhang



Terminal - historisches (etwas vereinfacht)

- Als die Computer noch größer waren:



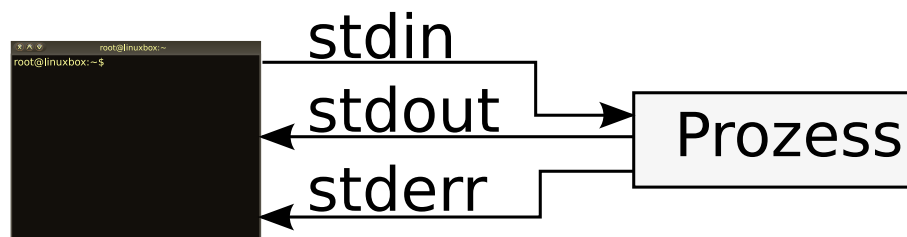
- Als das Internet noch langsam war:



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



- Drei Standardkanäle für Ein- und Ausgaben



`stdin` Eingaben

`stdout` Ausgaben

`stderr` Fehlermeldungen

- Standardverhalten

- Eingaben kommen von der Tastatur
- Ausgaben & Fehlermeldungen erscheinen auf dem Bildschirm



Terminal - Standardkanäle umleiten

- `stdout` Ausgabe in eine Datei schreiben

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

- `stdout` wird häufig direkt mit `stdin` anderer Programme verbunden

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

- Vorteil von `stderr`

⇒ Fehlermeldungen werden weiterhin am Terminal ausgegeben

- Übersicht

> Standardausgabe `stdout` in Datei schreiben

>> Standardausgabe `stdout` an existierende Dateien anhängen

2> Fehlerausgabe `stderr` in Datei schreiben

< Standardeingabe `stdin` aus Datei einlesen

| Ausgabe eines Befehls direkt an einen anderen Befehl weiterleiten



Shell - Wichtige Kommandos

- Wechseln in ein Verzeichnis mit `cd` (change directory)

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

- Verzeichnisinhalt auflisten mit `ls` (list directory)

```
1 ls
```

- Datei oder Ordner kopieren mit `cp` (copy)

```
1 cp /proj/i4spic/pub/aufgabeX/vorgabe.h /proj/i4spic/<login>/↵
   ↳ aufgabeX
```

- Datei oder Ordner löschen mit `rm` (remove)

```
1 rm test1.c
2 # Ordner mit allen Dateien löschen
3 rm -r aufgabe1
```



Shell - 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`



- 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:
 - AVR-Studio ohne Projekt
 - Notepad++
 - Dateizugriff über das Netzwerk
 - Übersetzen und Test unter Linux (z.B. via Putty)



Übersetzen & Ausführen

- Programm mit dem GCC übersetzen ²

1 `gcc -pedantic -Wall -Werror -O2 -std=c99 -D_XOPEN_SOURCE=500 -o ↗
 ↘ cworld cworld.c`

- Aufrufoptionen des Compilers, um Fehler schnell zu erkennen
 - -pedantic liefert Warnungen in allen Fällen, die nicht 100% dem verwendeten C-Standard entsprechen
 - -Wall warnt vor möglichen Fehlern (z.B.: `if(x = 7)`)
 - -Werror wandelt Warnungen in Fehler um
 - -O2 führt zu Optimierungen des Programms
 - -std=c99 setzt verwendeten Standard auf C99
 - -D_XOPEN_SOURCE=500 fügt unter anderem die POSIX Erweiterungen hinzu, die in C99 nicht enthalten sind
 - -o cworld legt Namen der Ausgabedatei fest (Standard: a.out)
- Ausführen des Programms mit `./cworld`



²Abgaben werden von uns mit diesen Optionen getestet

- 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`



## Inhalt

---

Linux

Fehlerbehandlung

    Bibliotheksfunktionen

    Kommandozeilenparameter

Aufgabe: cworld

Anhang



- 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
  - ⇒ 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



- Fehler treten häufig in Funktionen der C-Bibliothek auf
  - erkennbar i.d.R. am Rückgabewert (Manpage!)
- 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 }
```





```
1 ...
2 int main(int argc, char *argv[]){
3 strcmp(argv[argc - 1], ...)
4 ...
5 return EXIT_SUCCESS;
6 }
```

## ■ Übergabeparameter:

- `main()` bekommt vom Betriebssystem Argumente
- `argc`: Anzahl der Argumente
- `argv`: Vektor aus Strings der Argumente (Indices von 0 bis `argc-1`)

## ■ Rückgabeparameter:

- Rückgabe eines Wertes an das Betriebssystem
- Zum Beispiel Fehler des Programms: `return EXIT_FAILURE;`



## Inhalt

Linux

Fehlerbehandlung

Aufgabe: cworld

    Aufgabenstellung

    Dynamische Speicherverwaltung

    Umgang mit Strings

Anhang



- Ausgabe von "Hello World" unter Verwendung von malloc und strcpy
  - Allokieren von Speicher für Zeichenfolge (malloc(3), strlen(3))
  - Kopieren der Zeichenfolge in den allokierten Speicher (strcpy(3))
  - Ausgabe (printf(3)):  
Der Inhalt von pCH lautet: "<Inhalt von pCH>". pH zeigt auf <Adresse von pH>; pCH zeigt auf <Adresse von pCH>;
  - Freigabe des allokierten Speichers (free(3))
- Behandlung von Fehlern (Ausgabe über stderr)



## Dynamische Speicherverwaltung

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

- malloc(3) allokiert Speicher auf dem Heap
  - reserviert mindestens size Byte Speicher
  - liefert Zeiger auf diesen Speicher zurück
- malloc(3) kann fehlschlagen ⇒ Fehlerüberprüfung notwendig

```
1 char* s = (char *) malloc(strlen(...) + 1);
2 if(s == NULL){
3 perror("malloc");
4 exit(EXIT_FAILURE);
5 }
```

- Speicher muss später mit free(3) wieder freigegeben werden

```
1 free(s);
```

- Was ist ein Segfault?  
⇒ Zugriff auf Speicher der dem Prozess nicht zugeordnet ist  
  ≠ Speicher der reserviert ist



"hallo"  $\equiv$  

## ■ Repräsentation von Strings

- Zeiger auf erstes Zeichen der Zeichenfolge
- null-terminiert: Null-Zeichen kennzeichnet Ende

$\Rightarrow$  `strlen(s)` != Speicherbedarf

## ■ `printf(3)` Formatierungsstrings

`%s` String

`%d` Dezimalzahl

`%c` Character

`%p` Pointer

...



## Inhalt

Linux

Fehlerbehandlung

Aufgabe: cworld

Anhang

Arbeiten im Terminal

Debuggen

Valgrind



## ■ Navigieren & 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 .
```

## ■ Kompilieren:

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

## ■ Bereits eingegebene Befehle: Pfeiltaste nach oben

## ■ Besondere Pfadangaben:

- aktuelles Verzeichnis
- .. übergeordnetes Verzeichnis
- ~ Home-Verzeichnis des aktuellen Benutzers

⇒ Eine Verzeichnisebene nach oben wechseln: `cd ..`



# Debuggen

```
1 gcc -g -pedantic -Wall -Werror -O0 -std=c99 -D_XOPEN_SOURCE=500 -o ↵
 ↳ o cworld cworld.c
```

## ■ -g aktiviert das Einfügen von Debug-Symbolen

## ■ -O0 deaktiviert Optimierungen

## ■ Standard-Debugger: gdb

```
1 gdb ./cworld
```

## ■ „schönerer“ Debugger: cgdb

```
1 cgdb --args ./cworld arg0 arg1 ...
```

## ■ Kommandos

- b(reak): Breakpoint setzen
- r(un): Programm bei main() starten
- n(ext): nächste Anweisung (nicht in Unterprogramme springen)
- s(tep): nächste Anweisung (in Unterprogramme springen)
- p(rint) <var>: Wert der Variablen var ausgeben

⇒ **Debuggen ist (fast immer) effizienter als Trial-and-Error!**



- Informationen über:
  - Speicherlecks (malloc/free)
  - Zugriffe auf nicht gültigen Speicher
- Ideal zum Lokalisieren von Segmentation Faults (SIGSEGV)
- Aufrufe:
  - `valgrind ./cworld`
  - `valgrind --leak-check=full --show-reachable=yes --track-origins↵`  
↳ `=yes ./cworld`

