Übungen zu Systemnahe Programmierung in C (SPiC) – Wintersemester 2018/19

Übung 5

Rebecca Felsheim
Benedict Herzog

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Linux

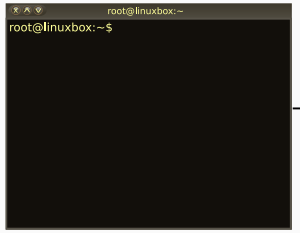
Terminal - historisches (etwas vereinfacht)

- Als die Computer noch größer waren:

1

Seriell — Computer

- Als das Internet noch langsam war:

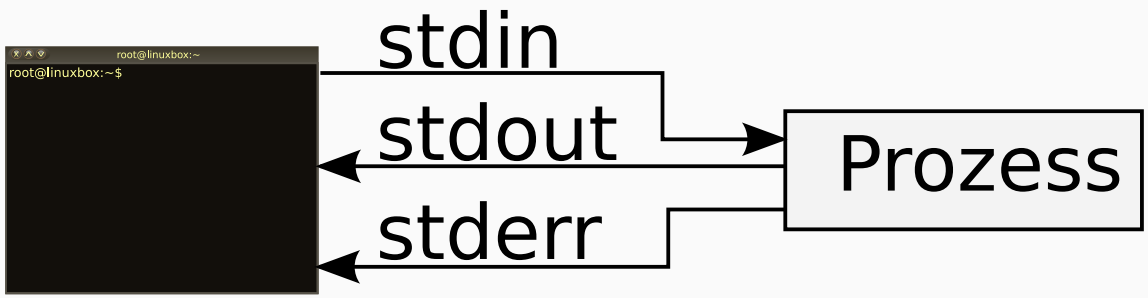


Netzwerk / DFÜ — Server

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

Terminal - Funktionsweise

- 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

```
01 find . > ordner.txt
```

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

```
01 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)

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

- Verzeichnisinhalt auflisten mit `ls` (list directory)

```
01 ls
```

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

```
01 cp /proj/i4spic/pub/aufgabeX/vorgabe.h /proj/i4spic/<login>/
       ↪ aufgabeX
```

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

```
01 rm test1.c
02 # Ordner mit allen Dateien löschen
03 rm -r aufgabe1
```

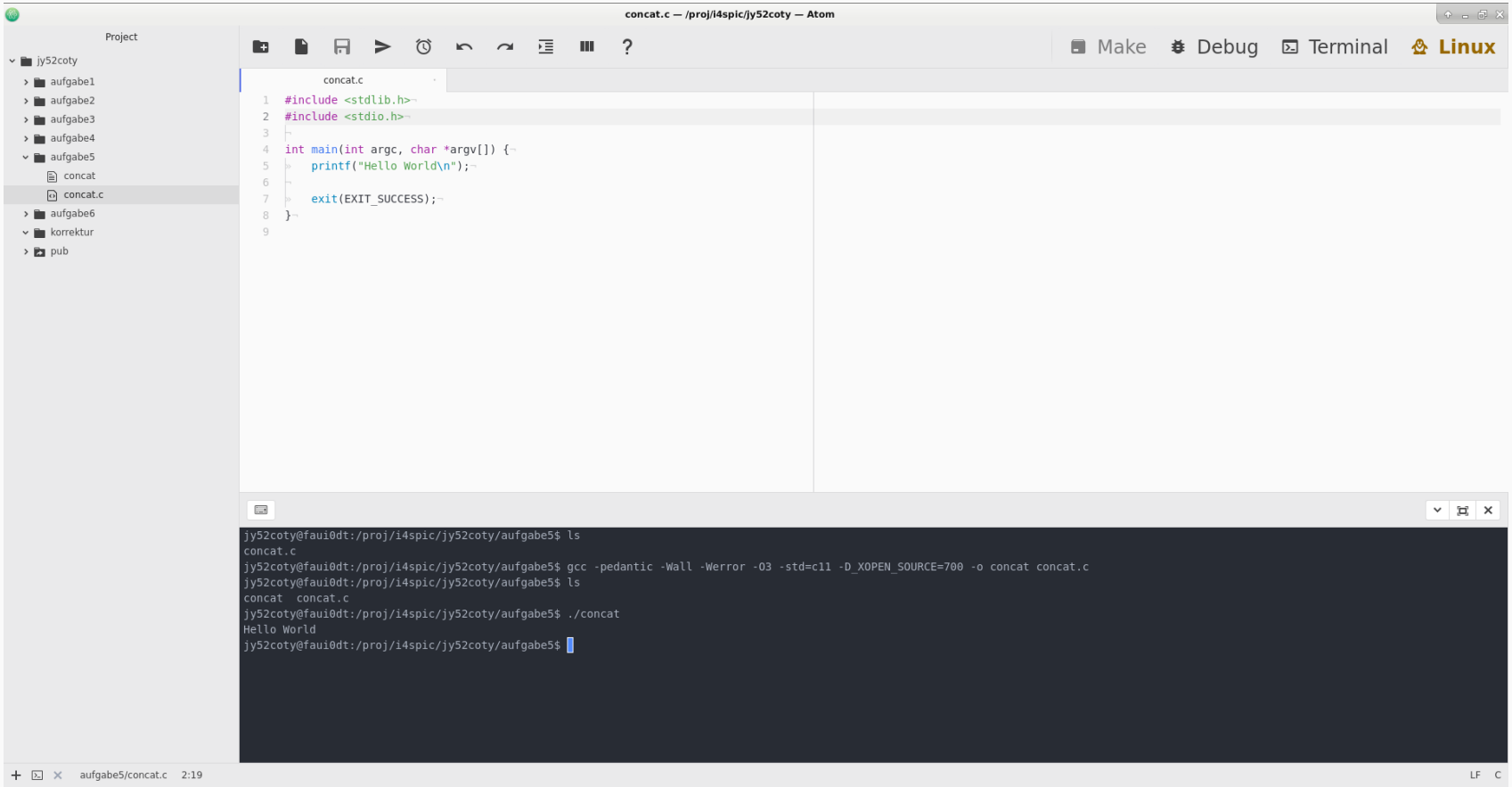
Shell - Programme beenden

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

Arbeitsumgebung

- Im CIP:
  - SPiC IDE
- Allgemein unter Linux:
  - beliebiger Editor (Kate, gedit, geany, Vim, Emacs)
  - Dateizugriff über das Netzwerk
- Unter Windows:
  - Notepad++, Sublime
  - Dateizugriff über das Netzwerk
  - Übersetzen und Test unter Linux (z.B. via Putty)

SPiC IDE (Linux)



- **Terminal**: öffnet ein Terminal und startet eine Shell
  - effiziente Interaktion mit dem (Betriebs-)system
  - optional Vollbild
- **Debug**: startet den Debugmodus
- **Make**: siehe nächste Woche

Übersetzen & Ausführen

- Programm mit dem GCC übersetzen[2]

```
01 gcc -pedantic -Wall -Werror -O3 -std=c11 -D_XOPEN_SOURCE=700 -o
     ↪ concat concat.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
  - `-O3` führt zu Optimierungen des Programms
  - `-std=c11` setzt verwendeten Standard auf C11
  - `-D_XOPEN_SOURCE=700` fügt unter anderem die POSIX Erweiterungen hinzu, die in C11 nicht enthalten sind
  - `-o concat` legt Namen der Ausgabedatei fest (Standard: `a.out`)

- Ausführen des Programms mit `./concat`

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)`

```
01 # man [section] Begriff
02 man 3 printf
```

- Suche nach Sections: `man -f Begriff`
- Suche von man-Pages zu einem Stichwort: `man -k Stichwort`

Fehlerbehandlung

Einschub: Dynamische Speicherverwaltung

```
01 void *malloc(size_t size);
02 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

```
01 char* s = (char *) malloc(strlen(...) + 1);
02 if(s == NULL){
03   perror("malloc");
04   exit(EXIT_FAILURE);
05 }
```

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

```
01 free(s);
```

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

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, um beides in eine Logdatei einzutragen
  - ⇒ 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 in Bibliotheksfunktionen

- 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

```
01 char *mem = malloc(...);    /* malloc gibt im Fehlerfall */
02 if(NULL == mem) {           /* NULL zurück */
03   fprintf(stderr, "%s:%d: malloc failed with reason: %s\n",
04       __FILE__, __LINE__-3, strerror(errno));
05   perror("malloc");   /* Alternative zu strerror + fprintf */
06   exit(EXIT_FAILURE); /* Programm mit Fehlercode beenden */
07 }
```

Erweiterte Fehlerbehandlung

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

```
01 int c;
02 while ((c=getchar()) != EOF) { ... }
03 /* 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)`

```
01 int c;
02 while ((c=getchar()) != EOF) { ... }
03 /* EOF oder Fehler? */
04 if(ferror(stdin)) {
05   /* Fehler */
06   ...
07 }
```

Kommandozeilenparameter

Kommandozeilenparameter

```
01 ...
02 int main(int argc, char *argv[]){
03   strcmp(argv[argc - 1], ... )
04   ...
05   return EXIT_SUCCESS;
06 }
```

- Übergabeparameter:
  - `main()` bekommt vom Betriebssystem Argumente
  - `argc`: Anzahl der Argumente
  - `argv`: Array aus Zeigern auf Argumente (Indizes von 0 bis argc-1)
- Rückgabeparameter:
  - Rückgabe eines Wertes an das Betriebssystem
  - Zum Beispiel Fehler des Programms: `return EXIT_FAILURE;`

Aufgabe: concat

Aufgabe: concat

- Zusammensetzen der übergebenen Kommandozeilenparameter zu einer Gesamtzeichenfolge und anschließende Ausgabe
  - Bestimmung der Gesamtlänge
  - Dynamische Allokation eines Buffers
  - Schrittweises Befüllen des Buffers
  - Ausgabe der Zeichenfolge auf dem Standardausgabekanal
  - Freigabe von dynamisch allokiertem Speicher
- Implementierung eigener Hilfsfunktionen:

```
01  size_t str_len(const char *s)
02  char *str_cpy(char *dest, const char *src)
03  char *str_cat(char *dest, const char *src)
```

- Wichtig: Korrekte Behandlung von Fehlern (!)

Umgang mit Strings (1)

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

  ⇒ `strlen(s)` != Speicherbedarf
- `printf(3)` Formatierungsstrings

  `%s` String
  `%d` Dezimalzahl
  `%c` Character
  `%p` Pointer
  ...

Umgang mit Strings (2)

- `size_t strlen(const char *s)`
  - Bestimmung der Länge einer Zeichenkette `s` (ohne abschließendes Null-Zeichen)
  - Rückgabewert: Länge
  - Dokumentation: `strlen(3)`
- `char *strcpy(char *dest, const char *src)`
  - Kopieren einer Zeichenkette `src` in einen Buffer `dest` (inkl. Null-Zeichen)
  - Rückgabewert: `dest`
  - Dokumentation: `strcpy(3)`
  - Gefahr: Buffer Overflow (⇒ `strncpy(3)`)
- `char *strcat(char *dest, const char *src)`
  - Anhängen einer Zeichenkette `src` an eine existierende Zeichenkette im Buffer `dest` (inkl. Null-Zeichen)
  - Rückgabewert: `dest`
  - Dokumentation: `strcat(3)`
  - Gefahr: Buffer Overflow (⇒ `strncat(3)`)

Anhang

Arbeiten im Terminal

- Navigieren & Kopieren:

```
01 cp /proj/i4spic/pub/aufgabeX/vorgabe.h /proj/i4spic/<login>/
       ↪ aufgabeX
02 # oder
03 cd /proj/i4spic/<login>/aufgabeX
04 cp /proj/i4spic/pub/aufgabeX/vorgabe.h .
```

- Kompilieren:

```
01 gcc -pedantic -Wall -Werror -O3 -std=c11 -D_XOPEN_SOURCE=700 -o
       ↪ concat concat.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

```
01 gcc -g -pedantic -Wall -Werror -O0 -std=c11 -D_XOPEN_SOURCE=700 -
     ↪ o concat concat.c
```

- `-g` aktiviert das Einfügen von Debug-Symbolen
- `-O0` deaktiviert Optimierungen
- Standard-Debugger: `gdb`

```
01 gdb ./concat
```

- „schönerer” Debugger: `cgdb`

```
01 cgdb --args ./concat 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!**

Valgrind

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

Hands-on: Buffer Overflow

Hands-on: Buffer Overflow

- Passwortgeschütztes Programm

```
# Usage: ./print_exam <password>
./print_exam spic
Correct Password
Printing exam...
```

- Ungeprüfte Verwendung von Benutzereingaben ⇒ Buffer Overflow

```
long check_password(const char *password){
  char buff[8];
  long pass = 0;

  strcpy(buff, password); // buffer overflow
  if(strcmp(buff, "spic") == 0){
    pass = 1;
  }
  return pass;
}
```

Hands-on: Buffer Overflow

```
01 long check_password(const char *password){
02   char buff[8];
03   long pass = 0;
04
05   strcpy(buff, password); // buffer overflow
06   if(strcmp(buff, "spic") == 0){
07     pass = 1;
08   }
09   return pass;
10 }
```

- Mögliche Lösungen
  - Prüfen der Benutzereingabe und/oder dynamische Allokation des Buffers
  - Sichere Bibliotheksfunktionen verwendeten ⇒ z. B. `strncpy(3)`