

Systemprogrammierung

Foliensatz A

- Einleitung
- Grundlagen C / Bash

Prof. Dr. Hans-Georg Eßer

Wintersemester 2023/24

v1.0 – 11.10.2023

Hilfreiche Vorkenntnisse:

- C – Grundlagen der Programmierung in C (oder C++, C#, Java)
- Betriebssysteme 1 (Theorie) und
- Betriebssysteme 2 (Shell-Programmierung)

Prüfung:

- keine Vorleistung nötig
- Prüfung besteht aus
 - Implementierungsarbeit mit
 - Dokumentation

1. Einführung / Motivation

Warum Systemprogrammierung?

- verstehen, wie Programme, Bibliotheken und Betriebssystem zusammen arbeiten
- Hauptspeicher effizient nutzen
- systemnahe Programmierung z. B. für Embedded-Systeme wichtig
- interessante Vertiefung zur Vorlesung *Betriebssysteme 1*
- Hier: BS = Linux / Unix, Programmiersprache C mit langer Tradition (Unix: 1969; C: 1972)

Gliederung

Gliederung (1)

1. Einleitung
2. Wiederholung: Grundlagen C und Linux-Shell Bash
3. Software und Betriebssystem; System Calls
4. Prozesse: fork, exec, wait
5. Dateien: open, read, write, close
6. Fortgeschrittene I/O
7. Speicherverwaltung: malloc, free, memcpy
8. Nebenläufigkeit mit POSIX-Threads
9. Netzwerk-Programmierung: TCP/IP, Sockets, HTTP
10. Einführung ins Projekt

Gliederung (2)

- Zu einigen Themen erst Grundlagen (Theorie der Betriebssysteme)
- Vorstellung der Features, die Linux hier bietet
- Programmieraufgaben dazu

Gegen Ende des Semesters:
größeres Programmierprojekt

2. Wiederholung C und Bash

Gliederung

1. Einleitung
2. C und Bash
3. Software/BS,
System Calls
4. Prozesse
5. Dateien
6. Fortgeschr. I/O
7. Speicher
8. Threads
9. Netzwerk

- *nicht* objektorientierte Programmierung
- Variablen, C-Structs
- Funktionen
- Pointer und Arrays
- Linux-Shell Bash (→ *Betriebssysteme 2*)

3. Software / BS / System Calls

Gliederung

1. Einleitung
2. C und Bash
3. Software/BS,
System Calls
4. Prozesse
5. Dateien
6. Fortgeschr. I/O
7. Speicher
8. Threads
9. Netzwerk

- System Calls (Syscalls), Beispiele
- Syscalls in Assembler und C aufrufen
- Bibliotheksfunktionen
- Beispiel: Dateizugriff mit `fread()`
- Syscall-Tracer: `strace`

4. Prozesse

Gliederung

1. Einleitung
2. C und Bash
3. Software/BS,
System Calls
4. Prozesse
5. Dateien
6. Fortgeschr. I/O
7. Speicher
8. Threads
9. Netzwerk

- Prozesskonzept unter Unix / Linux
 - Prozesskontrollblock
 - PID, Vater/Sohn, Baumstruktur
- Prozessverwaltung in der Shell
- Neue Prozesse erzeugen, `fork()`
- Programm in Prozess laden, `exec*()`
- Warten auf Prozess, `wait()`
- Signalisierung, `kill()`;
Signal-Handler, `signal()`
- Priorisierung, `setpriority()`, `nice()`

5. Dateien

Gliederung

1. Einleitung
2. C und Bash
3. Software/BS,
System Calls
4. Prozesse
5. Dateien
6. Fortgeschr. I/O
7. Speicher
8. Threads
9. Netzwerk

- Dateisysteme unter Unix/Linux
 - Datei, Inode, Verzeichnis
 - Link, Symlink, Named Pipe
- Datei öffnen und schließen, file descriptor
- lesen, schreiben, Position
- fork und offene Dateien
- Low-Level- und Bibliotheksfunktionen:
open(), read(), ... vs. fopen(), fread(), ...
- Verzeichnisse

6. Fortgeschrittene I/O

Gliederung

1. Einleitung
2. C und Bash
3. Software/BS,
System Calls
4. Prozesse
5. Dateien
6. Fortgeschr. I/O
7. Speicher
8. Threads
9. Netzwerk

- Standard-Eingabe, -Ausgabe und -Fehlerausgabe, inkl. file descriptors
- Bibliotheksfunktionen printf() und scanf()
- Pipes, dup()
- I/O-Multiplexing mit select()
- Memory-mapped files
- Locking
- Sparse Files

7. Speicher

Gliederung

1. Einleitung
2. C und Bash
3. Software/BS,
System Calls
4. Prozesse
5. Dateien
6. Fortgeschr. I/O
7. Speicher
8. Threads
9. Netzwerk

- Speicherverwaltung in C
- Organisation des Prozess-Speichers (Code, Daten, Stack, Heap)
- Speicher reservieren, `sbrk()`, `malloc()`
- Speicher freigeben, `free()`
- Eigene `malloc`-Version

8. Threads

Gliederung

1. Einleitung
2. C und Bash
3. Software/BS,
System Calls
4. Prozesse
5. Dateien
6. Fortgeschr. I/O
7. Speicher
8. Threads
9. Netzwerk

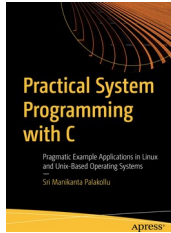
- Nebenläufigkeit innerhalb einer Anwendung
- Speichermodell Threads
- POSIX-Threads
 - erzeugen, `pthread_create()`
 - warten, `pthread_join()`
- Vergleich Threads / Prozesse

9. Netzwerk

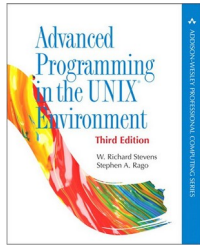
Gliederung

1. Einleitung
2. C und Bash
3. Software/BS,
System Calls
4. Prozesse
5. Dateien
6. Fortgeschr. I/O
7. Speicher
8. Threads
9. Netzwerk

- Grundlagen TCP/IP
- Sockets
- einfache Client-/Server-Programmierung
- HTTP



Practical System Programming with C
Sri Manikanta Palakollu; 2020
→ KAI, PDF



Advanced Programming in the UNIX Environment
W. Richard Stevens, Stephen A. Rago; 2013 (3rd edition)



Linux/Unix Systemprogrammierung
Helmut Herold; 2004 (3. Auflage)

A Tutorial on Pointers and Arrays in C
Ted Jensen, 50 S., 2003
<https://github.com/jflaherty/ptrtut13/tree/master/pdf>

2. Wiederholung C und Bash

C-Grundlagen (1)

- Vorab das wichtigste:
 - keine Klassen / Objekte
 - statt Objekten: „structs“ (zusammengesetzte Datentypen)
 - statt Methoden nur Funktionen
 - zu bearbeitende Variablen immer als Argument übergeben
 - kein String-Datentyp (sondern Zeichen-Arrays)
 - häufiger Einsatz von Zeigern
 - `int main () { }` ist immer Hauptprogramm

C-Grundlagen (2)

- Ausführlichere Informationen fürs Selbststudium:
<http://www.c-howto.de/>
→ auf der Webseite: ausführlichere Version mit erklärenden Kommentaren (Download: Zip-Archiv)
- auch als Buch für ca. 20 € erhältlich



C-Grundlagen (3)

- Im Anschluss an diese Vorlesung:
erstes Übungsblatt mit C-Aufgaben
- Vorbereitend ein paar Informationen zu
 - **Structs** (Strukturen, zusammengesetzte Typen)
 - **Pointern**
 - **Quellcode- und Header-Dateien** (Prototypen)

Structs

- Mehrere Möglichkeiten der Deklaration

```
struct {  
    int i;  
    char c;  
    float f;  
} variable;
```

```
variable.i = 9;  
variable.c = 'a';  
variable.f = 0.123;
```

```
struct mystruct {  
    int i;  
    char c;  
    float f;  
};
```

```
struct mystruct variable;  
  
variable.i = 9;  
variable.c = 'a';  
variable.f = 0.123;
```

```
typedef struct {  
    int i;  
    char c;  
    float f;  
} mystruct;
```

```
mystruct variable;  
  
variable.i = 9;  
variable.c = 'a';  
variable.f = 0.123;
```

Pointer

- Deklaration mit *: `char *ch_ptr;`
- verwalten Speicheradressen (an welchem Ort befindet sich die Variable?)

- Operatoren

- & (Adresse von)
- * (Dereferenzieren)

```
char ch, ch2;  
char *ch_ptr; char *ch_ptr2;  
  
ch_ptr = &ch; // Adresse von ch?  
ch2 = *ch_ptr; // Inhalt  
  
ch_ptr2 = ch_ptr;  
// kopiert nur Adresse
```

Strukturen und Pointer (3)

- Struct und Pointer kombiniert
- Oft bei verketteten Listen

```
struct liste {  
    struct liste *next;  
    struct liste *prev;  
    int inhalt;  
};
```

```
struct liste *anfang;  
struct liste *p;
```

```
for (p=anfang; p != NULL; p=p->next) {  
    use (p->inhalt);  
}
```


Strukturen und Pointer (4)

- Pointer-Typen
 - `typ *ptr;`
→ `ptr` ist ein Zeiger auf etwas vom Typ `typ`
 - `typ **pptr;`
→ `pptr` ist ein Zeiger auf einen Zeiger vom Typ `typ`
 - `ptr` bzw. `pptr` sind Speicheradressen
 - `*ptr` gibt den Wert zurück, der an der Speicherstelle abgelegt ist, auf die `ptr` zeigt
 - analog: `**pptr` ist ein Wert, aber `*pptr` ein Zeiger

Strukturen und Pointer (5)

- Pointer-Typen
 - &-Operator erzeugt zu Variable einen Pointer
 - Beispiele:

```
int i;  
int *ip;  
int **ipp;
```

```
i    = 42;  
ip   = &i;           // ip = Adresse von i  
ipp  = &ip;         // ipp = Adresse von ip
```

```
print (*ip);        // -> 42  
print (**ipp);     // -> auch 42
```

Strukturen und Pointer (6)

- Nicht-initialisierte Pointer: schlecht
 - Beispiel:

```
int *ip;  
int **ipp;
```

```
print (ip);    // nicht-init. Adresse (0)  
print (*ip);  // illegal -> Abbruch
```

```
*ip = 42;     // auch illegal, schreibt an  
              // nicht def. Adresse
```

Strukturen und Pointer (7)

- Vorsicht bei `char* a, b, c; etc.`

```
$ cat t2.c
```

```
int main () {  
    char* a, b;  
    printf ("|a| = %d \n", sizeof(a));  
    printf ("|b| = %d \n", sizeof(b));  
}
```

```
$ gcc t2.c; ./a.out
```

```
|a| = 8
```

```
|b| = 1
```

- besser: `char *a, *b, *c;`

Prototypen (1)

- Programm- und Header-Dateien
 - Header-Dateien (*.h) enthalten Funktionsprototypen und Makrodefinitionen (aber keinen normalen Code)
 - Programmdateien (*.c) enthalten den Code, können aber ebenfalls Prototypen und Makros enthalten (kein Zwang, eine .h-Datei zu erzeugen)

Prototypen (2)

- Funktionsprototypen (in Header-Dateien)
 - erlauben die Verwendung von Funktionen, deren Implementierung weiter unten im Programm (oder in einer anderen Datei) steht
 - Prototyp enthält nur Rückgabebetyp, Name und Argumente, z. B.
`int summe (int x, int y);`

Prototypen (3)

- Wie findet der Compiler die Header-Dateien?
→ Zwei Varianten:
 - `#include "pfad/zu/datei.h"`
Dateiname ist Pfad (relativ zu Verzeichnis mit der .c-Datei)
 - `#include <name.h>`
name.h wird in den Standard-Include-Verzeichnissen gesucht.
Welche sind das? Beim Bauen des gcc festgelegt...

Include-Verzeichnisse

- Standard-Include-Verzeichnisse

Debian 6.0 ↓

Ubuntu 22.04 →

```
deb:~$ cpp -v
Using built-in specs.
Target: i486-linux-gnu
[...]
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/gcc/i486-linux-gnu/4.4.5/include
 /usr/lib/gcc/i486-linux-gnu/4.4.5/include-fixed
 /usr/include
End of search list.
```

```
ubu:~$ cpp -v
Using built-in specs.
Target: x86_64-linux-gnu
[...]
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/11/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
```

... und deren
Unterordner

- Vorkenntnisse aus BS2:

ls

ls -l

touch

cp

cp -r

mv

rm

rm -r

rm -f

cd

mkdir

rmdir

- Wildcards (*, ?)
- relative / absolute Pfade
- Kurz- und Langoptionen
(- vs. --)
- Manpages
- Shell-Variablen; export
- Filter: cat, cut, head, sed,
sort, split, tail, tr,
uniq, wc