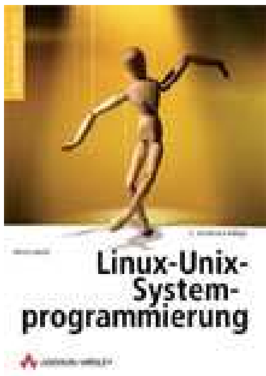
Linux/Unix Systemprogrammierung

Pablo Yánez Trujillo

Poolmanager Team
Universität Freiburg

August 14, 2006

Einführung



- Wer mehr über sein Unix/Linux wissen will, kann in die System Programmierung einsteigen
- Dieser Kurs wurde nach dem Buch "Linux/Unix Systemprogrammierung" von Helmut Herold gestaltet
- Dieser Kurs soll den Einstieg in die Systemprogrammierung erleichtern

Warum gerade C?

- Die Meinungen über C gehen auseinander. Viele halten C für veraltet und unbrauchbar
- C wird nicht in kürzer Zeit aussterben (trotz Vormarsch von C++, Java, C#)
- System- und Hardwareprogrammierung findet fast ausschließlich in C statt
- C ermöglich sehr nah am Speicher zu arbeiten $\Longrightarrow$ viel Macht verknüpft mit viel Verantwortung

Ziel dieses Kurses

- Eine Einführung der Systemprogrammierung unter Unix (mit Beispielen für GNU/Linux)
- Verständnis für einige Vorgänge der Arbeitsweise von Unix ähnliche Systeme
- Am Ende des Kureses wollen wir einen ganz einfachen Cron-Job Daemon schreiben

Überblick über die Unix Systemprogrammierung

- Die Systemprogrammierung ist umfangreich und man muss ein wenig über das System wissen, bevor man startet
- Ganz überflüssig:
  - Anmeldung unter Unix
  - Dateienarten
  - Ein- und Ausgabe
  - Ausgabe von System-Fehlermeldungen
  - Unterschied zwischen System- und Bibliotheksfunktionen

Anmeldung unter Unix

- Damit Prozesse (Programme) gestartet werden können, muss ein Benutzer sich einloggen
- Es muss auf jeden Fall einen Benutzer existieren, denn Prozesse müssen durch einen Benutzer gestartet werden
- Es gibt 3 Datein, die die Benutzerinformation beinhalten: `/etc/passwd`, `/etc/shadow`, `/etc/group`

/etc/passwd

/etc/passwd

```
supertux:x:1000:100:Pablo Yánez Trujillo:/home/supertux:/bin/bash
mysql:x:60:60:added by portage for mysql:/dev/null:/bin/false
ldap:x:439:439:added by portage for openldap:/usr/lib/openldap:/bin/false
```

- `supertux`: Benutzername
- `x`: Passwort
- `1000`: UID (User id)
- `100`: GID (Group id)
- `Pablo Yánez Trujillo`: Real name
- `/home/supertux`: HOME-Verzeichnis
- `/bin/bash`: Login Shell

/etc/shadow

/etc/shadow

```
supertux:$1$Zrp.8O1.$wgCDX8UfnFzsq47QsO0:13212:0:99999:7:::
mysql:!:13212:0:99999:7:::
ldap:!:13213:0:99999:7:::
```

- supertux: Benutzername
- 1Zrp...: Verschlüsseltes Passwort
- 13212: DOC (Day of last change), Tag ab dem 1.1.1970, an dem das Passwort zuletzt geändert wurde
- 0: Minimale Anzahl Tage, die das Passwort gültig ist
- 99999: Maximale Anzahl Tage, die das Passwort gültig ist
- 7: Anzahl der Tage vor Ablauf der Lebensdauer des Passwortes, ab der vor dem Verfall zu warnen ist
- Expire, wieviele Tage gilt das Passwort trotz Ablauf der MaxD?
- Bis zu diesem Tag (gezählt ab 1.1.1970) ist dieser Account gesperrt
- Reserve, Feld wird derzeit nicht ausgewertet

/etc/group

/etc/group

```
smmsp:x:209:smmsp
portage:x:250:portage,supertux
utmp:x:406:
```

- `portage`: Gruppenname
- `x`: Gruppenpasswort
- 250: GID (Group id)
- `portage,supertux,...`: Benutzer der Gruppe portage

- *Regular Files*: Normale Dateien: Sammlung von Zeichen
- *Special Files*: Gerätedateien: logische Beschreibung von physikalischen Geräten
  - Zeichenorientiert (Datentransfer zeichenweise)
  - blockorientiert (Datentransfer in Blöcken)
- *Directory* Verzeichnisse
- *FIFO* (Named Pipes) dienen zur Komunikation und Synchronisation von Prozessen
- *Sockets* dienen zur Komunikation von Prozessen in einem Netzwerk
- *Links* (hard- & symbolische): Dateien, die auf andere zeigen

- Der Kernel weist jeder geöffneten Datei eine positiven Zahl zu
- Diese Zahl heißt *Filedesciptor* (File-Deskiptor)
- Mit Hilfe von Systemroutinen kann man in eine Datei durch ihren Deskriptor schreiben und lesen
- Jedes Programm hat 3 standardmäßige Filedeskriptoren (`<unistd.h>`)
  - *Standard input* (Standardeingabe) **STDIN_FILENO**
  - *Standard output* (Standardausgabe) **STDOUT_FILENO**
  - `Standard error` (Standardfehlerausgabe) **STDERR_FILENO**

- In `<stdio.h>` werden die Standard I/O-Funktionen definiert, die nicht über Deskriptoren arbeiten, sondern durch Zeiger auf `FILE` Strukturen
- Die 3 Standard Filedeskriptoren sind ebenfalls als Zeiger auf `FILE` Strukturen zugänglich
  - `STDIN_FILENO` ⟶ **stdin**
  - `STDOUT_FILENO` ⟶ **stdout**
  - `STDERR_FILENO` ⟶ **stderr**

Beispiel: copy

```c
#include <stdio.h>


int main(void)
{
    int c;


    while( (c = getc(stdin)) != EOF)
        if(putc(c, stdout) == EOF) {
            fprintf(stderr, "error using putc\n");
            return 1;
        }


    return 0;
}
```

Ausführung

```
$ gcc copy.c -ocopy
$ ./copy < copy.c
$ cat copy.c | ./copy
$ ./copy > a.c
$ ./copy < copy.c > a.c
```

perror

```
#include <stdio.h>

void perror(const char *s);
```

strerror

```
#include <string.h>

char *strerror(interrnum);
```

Äquivalente Aufrufe

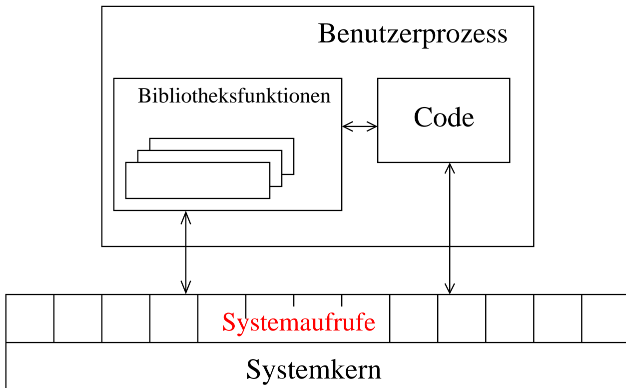
```
#include <stdio.h>
#include <string.h>
#include <errno.h> /* globale Variable errno */

perror("testausgabe");
fprintf(stderr, "testausgabe:  %s\n", strerror(errno));
```

Siehe Beispiel `tag01/error`

Unterschied zwischen System- und Bibliotheksfunktionen

- Es gibt im Wesentlichen 2 Arten von Funktionen
  - Systemaufrufe
  - Bibliotheksfunktionen
- Systemaufrufe sind Schnittstellen zum Systemkern. Sie sind in der 2. Sektion des Unix Programmrs's Manual (man 2 ...) beschrieben
- Systemaufrufe sind keine Schnittstellen zum Systemkern, wenn auch sie als Wrapper benutzt werden oder Systemaufrufe erledigen. Sie sind in der 3. Sektion des Unix Programmrs's Manual (man 3 ...) beschrieben
- `printf` ruft `write` auf, `strlen` oder `sqrt` kommen ohne Systemaufrufe zurecht

Unterschied zwischen System- und Bibliotheksfunktionen



Auszug: Linux/Unix Systemprogrammierung (Helmut Herold)

- ANSI C ist die Spezifikation der Sprache C
- ANSI C ist eine vom ANSI Komitee entworfene Norm der Sprache C
- Jeder C Compiler **sollte** sich daran halten und ANSI C Code übersetzen
- Im Kurs verwendeter C Compiler: GNU GCC

- Die C Syntax ist einfach (C++/Java ähnlich)
- Die Sprache enthält:
  - Data structures (primitive, Felder, Zeiger, zusammengesetzte)
  - Funktionen
  - Operatoren (+,-,*,/,%)
  - Kontrollstrukturen (`for`,`if`,`while`,`switch`,`do`)

- primitive: char,int, long,double, float
- Felder: Variable, die mehrere Werte speichern kann
- Zeiger: Variable, deren Inhalt eine Adresse im Speicher ist
- zusammengesetzte: structs

Deklaration von Variablen

```
int anzahl; /* Variable vom Typ Integer */
int feld[3]; /* Feld vom Typ Integer */
int *ptr1; /* Zeiger auf eine int Variable */
int *ptr2 = &anzahl; /* Zeiger auf anzahl */
struct map { /* struct mit int und char */
    char *name;
    int wert;
};
struct map a,b; /* mehrere Variablen von Typ struct
map */
```

- Sie helfen, den Code zu strukturieren und in kurzen Abschnitten zu teilen
- Programmier können davon profitieren, z.B. in Bibliotheken
- Der Code ist einfacher zu lesen und es ist einfacher nach Fehlern zu suchen
- Wenn man sieht, dass ein Teilcode immer wieder vorkommt, dann kann man es in eine Funktion packen.

type **function-name**([Parameter...]) *code*

Beispiel 1: Additionsfunktion

```
int addition(int a, int b)
{
    return a+b;
}
⋮
void foo()
{
    int a,b,c;
    a = 8; b = 10;
    c = addition(a,b);
}
```

IF Abfrage

if(*Bedingung*) *code*

IF Abfrage

```
int abs(int n)
{
    if(n<0)
        return -n;
    else
        return n;
}
```

SWITCH Abfrage

switch(*variable*) **case** *value*: *code* **break;**

SWITCH Abfrage

```
void print_name(int val)
{
    switch(val)
    {
        case 0:  printf("Pablo\n");
                 break;
        case 1:
        case 2:
        case 3:  printf("Michael\n");
                 break;
        default:  printf("Unbekannt\n");
    }
}
```

FOR Schleife

for(*Startzuweisung*; *Bedingung* ; *Aktion*) *code*

FOR Schleife

```
int n;
for(n=0; n<10; ++n)
    printf("%d ", n);

printf("\n");

/* Ausgabe:  0 1 2 3 4 5 6 7 8 9 */
```

WHILE Schleife

while(*Bedingung*) *code*

WHILE Schleife

```
#include <stdio.h>
#include <time.h>
⋮
int zufall = 8;
srand(time(NULL)); /* initialisiere Zufallsgenerator */

while(zufall != 1048)
{
    printf("Mist....  zufall ist %d und nicht 1048\n", zufall);
    zufall = rand() % 2000;
}

printf("Endlich raus aus dieser Schleife!\n");
```

- Die Funktion `printf` ist die wichtigste Funktion, wenn man am Bildschirm etwas ausgeben will
- Sie wird in der Datei `stdio.h` definiert und muss stets eingebunden sein
- Konstrukte von C++ oder Java wie `"a = " + a + "\n"` sind in C nicht möglich
- Bei `printf` muss man zuerst das Format eingeben und dann alle Variablen hinten anhängen

printf Beispiele

```
#include <stdio.h>

printf("Nur Text ausgeben\n");
printf("Inhalt eines Integers ausgeben, d=%d\n", d);
printf("Inhalt eines Floats ausgeben, d=%f\n", d);
printf("Inhalt eines Strings ausgeben:  %s\n", "Hallo, Welt!");
printf("Nur ein einzelnes Zeichen ausgeben:  %c\n", 'a');
printf("Gemischte Ausgabe:  3+5=%d, string=%s\n", 3+5, "Hallo, Welt!");
```

complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H


struct complex {
    float real;
    float imaginary;
};


struct complex complex_addition(struct complex a, struct complex b);
float get_real(struct complex x);
float get_imaginary(struct complex x);


#endif
```

complex.c

```
#include "complex.h"


struct complex complex_addition(struct complex a, struct complex b)
{
    /* Code comes here */
}
...
```

main.c

```
#include <stdio.h>
#include "complex.h"

int main(void)
{
    struct complex a,b,c;
    a = assign(4, 1);
    b = assign(6, -9.7);

    c = complex_addition(a,b);

    printf("c = (%f, %f)\n", c.real, c.imaginary);

    return 0;
}
```

Die `main` Funktion

- Damit ein kompilierter Code lauffähig wird, muss der Code die `main` Funktion haben
- Es sollte nur eine Datei geben, die die `main` Funktion implementiert, sonst kann es zu Linker-Fehlern kommen
- Man braucht kein Prototyp für die `main` Funktion
- Aber ... es gibt mehrere (gültige) Möglichkeiten, die `main` Funktion zu implementieren

Die main Funktion

```
int main();

int main(void);

int main(int argc, char *argv[]);

int main(int argc, char **arvg);
```

- Wenn das Programm fehlerlos beendet wurde, sollte `main` die 0 zurückgeben (wichtig für das Betriebsystem)
- Wenn das Programm fehlerhaft beendet wurde, sollte `main` etwas ungleich 0 zurückgeben

- Zeiger sind sehr wichtig in der C Sprache. Sie ermöglichen eine am Speicher nahe Prorgrammierung
- Zeiger sind an sich Integer Variablen, deren gespeicherten Werte den Adressen des Speichers (worauf sie zeigen) entsprechen
- D.h. mit Zeigern kann man auf Stellen des Speichers zugreifen, die wir beispielsweise nicht deklariert haben (siehe `main` Funktion)
- Es gibt Zeiger auf Zeiger (manchmal ist es notwendig)

Annahme: 1 byte pro Datenstruktur

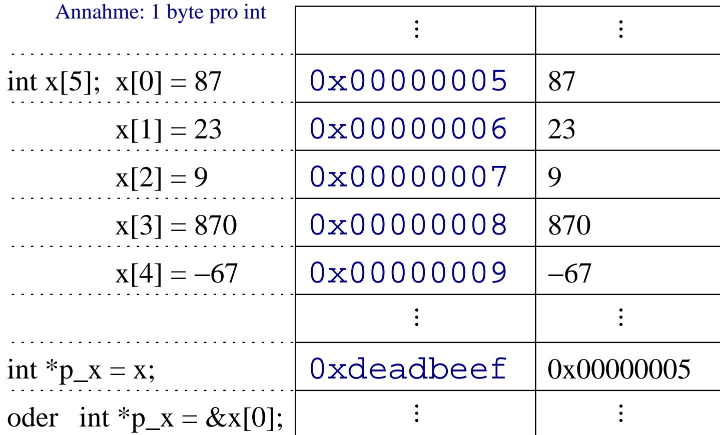
	Speicher Adresse	Inhalt
int a = 6;	0x00000001	6
char c = 'Z';	0x00000002	90
void* p;	0x00000003	–187267182
	⋮	⋮
float f = –25/3;	0xdeadbeef	–0.8333333
int *p_a = &a;	0xdeadbef0	0x00000001
	⋮	⋮
float *p_f = &f;	0xaf5654aa	0xdeadbeef
int* p_so = NULL;	0xaf5654ab	0x00000000

- Da Zeiger im Prinzip Integer Variablen sind, kann man mit Zeigern die Operation auf Integers ausführen.
- Was man damit bewirkt, ist dass man den Inhalt des Zeigers (also die Speicheradresse, auf die der Zeiger zeigt) ändert
- So kann man sich durch den Speicher "bewegen", erst interessant, wenn man mit dynamsichen Speicher arbeitet

Zeiger-Arithmetik

```
int a = 10; int *p_a = &a; /* Zeiger auf a */

printf("Inhalt von p_a:  %x.  Inhalt von a:  %d\n", p_a, *p_a);

*p_a = 18; /* Adresse bleibt gleich, Inhalt von a ändert sich auf 18 */

p_a++; /* Zeige auf nächste Stelle im Speicher */

(*p_a)++; /* Erhöhe Inhalt von a um 1 */

p_a = 0xdeadbeef; /* Möglich, aber wenn 0xdeadbeef nicht im Adressraum
liegt, dann kann nicht drauf zugreifen */
```

- Arrays (Felder) sind wie Tabellen, die mehrere Werte eines gleichen Typs speichern können
- Sie verhalten sich sehr ähnlich wie Zeiger. Wenn eine Funktion ein Array braucht, wird in der Tat einen Zeiger auf das erste Element des Arrays übergegeben
- Die Länge der Arrays muss vor der Kompilierungszeit bekannt sein (ab C99 nicht mehr, aber die wenigsten Compiler halten sich jetzt an C99)


Speicher Adresse
Inhalt
Annahme: 1 byte pro int
int x[5]; x[0] = 87
x[1] = 23
x[2] = 9
x[3] = 870
x[4] = –67
int *p_x = x;
oder int *p_x = &x[0];
0x00000005
87
0x00000006
23
0x00000007
9
0x00000008
870
0x00000009
–67
0xdeadbeef
0x00000005

Arrays

```
int arr1[5] = { 3, 4, 1, -2, 97 };
int arr1[] = { 3, 4, 1, -2, 97 };
int arr1[5]; arr1[0] = 3; arr1[2] = 4; ...
int *p = arr1; p[0] = 3; p[1] = 4; ...
int *p = &arr1[0]; p[0] = 3; p[1] = 4; ...
```

- Wir haben einige Funktionen gesehen, die Zeichen lesen und ausgeben
- Jetzt wollen wir einige der Standard Funktionen zur Ein- und Ausgabe von Zeichen

- Die meisten Standard Funktion haben als Parameter einen Zeiger auf eine `FILE` Struktur
- Die Struktur enthält folgenden Attributen:
  - Anfangsadresse des Puffers
  - aktueller Pufferzeiger
  - Puffergröße
  - Filedeskriptor
  - Position des Schreib-/Lesezeigers in einer Datei
  - Fehler-Flag
  - EOF-Flag
- Man sollte niemals auf diese Elemente einzeln zugreifen, dafür existieren vordefinierte Funktion, die diese Parameter korrekt bearbeiten

Dateien öffnen

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);

FILE *freopen(const char *path, const char *mode, FILE *fp);
```

Datein Schließen

```
#include <stdio.h>

int fclose(FILE *fp);
```

Öffnungsmodi

modus-Argument	*Bedeutung*
`r` oder `rb`	(read) zum Lesen öffnen
`w` oder `wb`	(write) Zum Schreiben öffnen Datei wird **immer** leer angelegt
`a` oder `ab`	(append) Zum Schreiben öffnen Schreibzeiger am Ende des Puffers
`r+`, `r+b` oder `rb+`	Zum Lesen und Schreiben öffnen
`w+`, `w+b` oder `wb+`	Zum Lesen und Schreiben öffnen Datei wird **immer** leer angelegt
`a+`, `a+b` oder `ab+`	Zum Lesen und Schreiben öffnen Schreibzeiger am Ende des Puffers

Übung

Erweitere das Programm `tag01/error` so, dass die Ausgabe auf `stderr` in eine Datei geschrieben wird, die über die Parameter gelesen wird

Schreiben in Dateien

```
#include <stdio.h>

int fputc(int c, FILE *fp);

int fputs(const char *buffer, FILE *fp);

int fprintf(FILE *fp, const char *format, ...);
```

Lesen aus Dateien

```
#include <stdio.h>

int fgetc(FILE *fp);

char *fgets(char *s, int size, FILE *stream);

int fscanf(FILE *fp, const char *format, ...);
```

- Wenn man die Funktionen `fget*` benutzt, sollte man überprüfen:
  - Lesefehler
  - EOF (end of file)
- Ungepufferte Daten sollten geschrieben werden (`fflush`)

Dateistatus

```
#include <stdio.h>

int feof(FILE *fp);

int ferror(FILE *fp);

int fflush(FILE *stream);
```

- Textdateien zu lesen ist einfach, weil Strings einach zu lesen sind
- Wenn man aber binäre Daten oder das Format unbekannt ist, muss man ganze Blöcke von Daten lesen

Blöcke Lesen und Schreiben

```
#include <stdio.h>

size_t fread(void *puffer, size_t bgroesse, size_t banzahl, FILE *fp);

size_t fwrite(const void *puffer, size_t bgroesse, size_t banzahl,
              FILE *fp);
```

Siehe Beispiel `tag01/readwrite`

Positionierung in einer Datei

```
#include <stdio.h>

int fseek(FILE *fp, long offset, int modus);

long ftell(FILE *fp);
```

- `fseek` liefert bei Erfolg 0 zurück, sonst -1
- `ftell` liefert bei Erfolg die momentane Position des Schreib-/Lesezeigers zurück, sonst -1L

`modus`-Angabe	*Wirkung*
SEEK_SET	Schreib-/Lesezeiger vom Dateianfang um `offset` Bytes versetzt
SEEK_CUR	Schreib-/Lesezeiger von momentaner Position um `offset` Bytes versetzt
SEEK_END	Schreib-/Lesezeiger vom Dateiende um `offset` Bytes versetzt

Übungen

- Schreibe ein Programm, welches nur ein Teil der Datei ausgibt (Benutzer muss Anfang und Ende eingeben)

- Was macht folgender Code?

```
FILE *f = fopen("file.dat", "r+");
if(f){
    fseek(f, 4, SEEK_SET);
    fputc('A', f);
    fclose(f);
}
```

- Erweitere das Programm `tag01/readwrite` so, dass nach der Angabe einer Position, die Daten in der Datei geändert werden, ohne sie vollständig neu zu schreiben