Programmierkurs 3

Systemnahe Programmierung

Marcel Waldvogel

University of Konstanz

Winter 2016/2017

These slides are based on previous lectures, held by Alexander Holupirek, Roman Byshko, and especially Stefan Klinger.

0

Organisational stuff

0.1 Personnel

Marcel Waldvogel Distributed Systems Laboratory (di.sy)

mail/xmpp marcel.waldvogel@uni-konstanz.de

Klaus Herberth Distributed Systems Laboratory (di.sy)

mail/xmpp klaus.herberth@uni-konstanz.de

Help Channel XMPP

xmpp info3@conference.uni-konstanz.de

0.2 Informatik 3

- Together with **“Betriebssysteme”** (OS, *aka.* “Operating Systems”), this lecture forms the module “Informatik 3”
- You need to gain **50% of the exercise points** in **each** of the exercises be admitted for the final exam (“Three strikes and you’re out“).
- There will be ≈ 30% PK3 questions in the joint exam.
- **Registration**
  - For the exercises, form groups of two.
  - Sign up for OS **and** PK3 via StudIS[1], as usual[2].

[1] https://studis.uni.kn

[2] http://www.informatik.uni.kn/studieren/studium/pruefungen/pruefungsanmeldung/

0.3 Coordinates

Weekly schedule

Lectures	Monday
OS tutorials	Thursday
PK3 tutorials	Friday
Handin	Thursday, 04:00 via **git**
Handout	Thursday, ≈10:00 via **git**

Teaching materials are available via git[3] on GitLab[4]. One member of the group forks the repository[5] into his personal space and

- **grants access** to the other member (at least ‘Developer’),
- **shares** it with the group ‘info3’ (at ‘Master’ level).

You can then browse all files in the repository, which includes instructions on how to clone it into a local repository.

[3] https://git-scm.com/

[4] https://git.uni.kn

[5] https://git.uni.kn/info3/bspk2016

The main command is

```
$ git clone git@git.uni.kn:user-name/bspk2016.git
Cloning into 'bspk2016'...
remote: Zähle Objekte: 20, Fertig.
remote: Komprimiere Objekte: 100% (18/18), Fertig.
remote: Total 20 (delta 3), reused 0 (delta 0)
Receiving objects: 100% (20/20), 1.09 MiB | 0 bytes/s, done.
Resolving deltas: 100% (3/3), done.
Checking connectivity... done.
```

This works only after having set up `ssh` (with keys) and `git`. You will learn how to do this in the tutorials.

0.4 PK3 Assignments

- You need to form groups of two to work on the exercises.
  This is **organised during the BS lecture**, see there.
- One **PK3**-assignment every week
  - Every new assignment is released on **Thursday at 10:00**,
  - due the following Thursday, before 04:00.
  - Discussed on **Thursday/Friday** in the tutorials.
- Submit your exercises via `git`:
  - Each group has r/w access only to their fork.
  - Commit your solutions to: `/ass_pk`, where ass is the 2-digit assignment number.[6]

Especially in systems programming, just because your code works does not mean it is correct!

[6] Put your solution in the directory mentioned before — we will not look in other places.

- We are **quite strict** about compiler errors and warnings:
  - Compile (we will) your code with

```
$ gcc -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast \
> -Wconversion -Wwrite-strings -Wstrict-prototypes source.c
```

  - You will gain **no points at all** for a programming exercise if the compiler stops with an **error**.
  - We will subtract **3 points** for every **compiler warning**.
- **Bonus points** for nice code:
  - Use checkpatch.pl from the Linux kernel project[7] against your source:

```
$ checkpatch.pl --no-tree --no-signoff -f --ignore NEW_TYPEDEFS,\
> AVOID_EXTERNS,GLOBAL_INITIALISERS,BLOCK_COMMENT_STYLE source.c
total: 0 errors, 0 warnings, 44 lines checked
```

  - Your point score for a program will **increase** by 5% if checkpatch generates warnings but **no errors**, and by **10%** if there are neither errors **nor warnings**!
- The Tutors will show you on Friday how to use the compiler and the checkpatch script.

[7] Included in your repository under /pk_code. Source: https://github.com/torvalds/linux/blob/c5595fa/scripts/checkpatch.pl

- It makes sense to set up some aliases in you `~/.bashrc`:

```
# a Very Picky C Compiler
alias pk-cc=’gcc -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast \
-Wconversion -Wwrite-strings -Wstrict-prototypes’;

# using checkpatch from the Linux kernel
alias pk-chk=’checkpatch.pl --no-tree --no-signoff -f […]’;
```

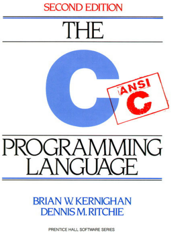
- and then use

```
$ pk-cc source.c                                                  # compiles
$ pk-chk source.c
total: 0 errors, 0 warnings, 44 lines checked               # and looks nice

source.c has no obvious style problems and is ready for submission.
```

0.5 Literature

- Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language*. 1978, Prentice Hall Software Series. Uni-KN kid 248 k27.



- Peter van der Linden. *Expert C Programming — Deep C Secrets*. 1994, Sunsoft Press, Prentice Hall. ISBN 0-13-177429-8, Uni-KN kid 248 v16.
- Randal E. Bryant, David O'Hallaron. *Computer Systems — A Programmer's Perspective*. 2003, Pearson Education International, Prentice Hall. ISBN 0-13-178456-0, Uni-KN kid 100n b79.

0.6 What is this course about?

System Programming

- With **system** we mean *operating system*.
- With **programming** we mean *using the interface* an operating system (OS) provides.
- With OS we mean UNIX-like OSs, *i.e.*, Linux.

Operating System

- Layer of software on top of bare hardware
- Shields programmers from the complexity of the hardware
- Presents an interface (of a virtual machine) that is easier to understand and program

Systems vs. Kernel programming

- Black Box Modell is suitable for systems programming.
- Knowledge about the system's internals, however, is beneficial to use the system properly and to not work against it.
- Providing the system services is (mostly) kernel programming.

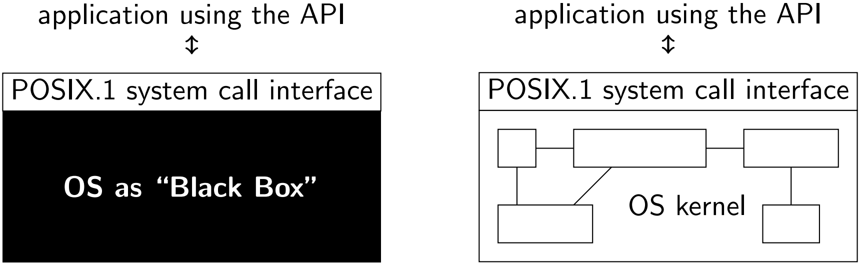

application using the API
POSIX.1 system call interface
OS as "Black Box"
application using the API
POSIX.1 system call interface
OS kernel

Figure: Black Box vs. White Box View of a UNIX System

0.7 Our working environment

- Get GNU/Linux up and running on your machine.
  - For PK3 you will work *under* Linux.
  - For BS you will work *on* Minix.
- Use the virtual machine set up as part of the tutorials.

You will need

- An editor, *e.g.*, `vim`, `emacs`, `nano`, `geany`, … Eclipse is not recommended!
- `gcc` — the GNU project C (and C++) compiler
- …

1

Gentle introduction to C

1.1 C standardization

- ISO/IEC 9899:1990 Programming Language C, (C89 or C90)
- ISO/IEC 9899:1999 Programming Language C, (C99)
- ISO/IEC 9899:2011 Programming Language C, (C11)

Note We will focus on C99, *i.e.*, use `-std=c99` as compiler flag.

C popularity

- Requirements that make C mandatory:
  - embedded systems (close to hardware, scarce resources)
  - extreme performance (better usage of resources)
  - the world is built on C and C++ (with C++ being a superset of C)

    — Herb Sutter. C++ and Beyond.[8]
  - C is simple & powerful

    — Damien Katz (CouchDB). The Unreasonable Effectiveness of C.[9]
- Programming Languages Rankings
  - 2nd place in TIOBE[10] (October 2015)
  - 9th place in RedMonk[11], with C++ ranking 5th (June 2015)

[8] https://www.youtube.com/watch?v=xcwxGzbTyms

[9] http://damienkatz.net/2013/01/the_unreasonable_effectiveness_of_c.html

[10] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[11] http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/

1.2 First C Program

Print the sentence: “Hello world!”

- Create the program text
- Compile it successfully
- Run it
- Get the output

```
#include <stdio.h>

main()
{
        printf("Hello world!\n");
}
```

Compilation on a UNIX-like OS

```
$ gcc hello.c
hello.c:3:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^
$ ls
a.out   hello.c
$ ./a.out
Hello world!
```

engine	filename	description
	hello.c	source code
preprocessor	hello.i	source w/ preproc. directives expanded
compiler	hello.s	assembler code
assembler	hello.o	object code ready to be linked
linker	a.out	executable

(Use `-save-temps` to preserve these files)

1.3 Program structure

Basic building blocks

- *functions* contain *statements*
- *statements* specify computing operations to be done
- *variables* store values used during computation
- *arguments* (one way to) communicate data between functions

Building blocks of our example

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello world!\n");
6     return 0;
7 }
```

- line 1: include the standard input/output library
- line 3–7: define a function `main`
  - `main` is special, a program begins execution here
  - `main` will usually call other functions to help perform its job
  - You can define your own functions, but don't name them `main`
  - `main` returns an `int`. (It has to!)
- line 3: Parentheses after the function name surround the argument list, `void` means no arguments.
- line 5: `main` calls library function `printf`, which prints this sequences of characters; `\n` represents the newline character.

Arithmetics

Fahrenheit-Celsius: °C = (5/9)(°F-32)

```c
/* print fahrenheit-celsius table for fahrenheit = 0, 20, ..., 300 */

#include <stdio.h>

int main(void)
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;    /* lower limit */
    upper = 300;  /* upper limit */
    step = 20;    /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr - 32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
    return 0;
}
```

Running:

```
$ ./a.out
0       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```

Declarations and assignment statements

- A *declaration* announces the properties of variables.
  Consists of *type name* and a *list of variables*, such as:

```c
    int fahr, celsius;
    int lower, upper, step;
```

  Range/ size of data types depends on machine

- *Assignment statements* set the variables to their initial values.

```c
    lower = 0;    /* lower limit */
    upper = 300;  /* upper limit */
    step = 20;    /* step size */
```

Basic data types

`char` a single byte. By definition, this is the unit of measurement for memory size.

`int` an integer, typically reflecting the natural size of integers on the host machine

`float` single-precision floating point

`double` double-precision floating point

`short` and `long` are *qualifiers* that can be applied to integers:

```
short int i;
long int f;
unsigned long d;
```

The qualifiers `signed` and `unsigned` can be applied to `char` and any integer.

The `while` loop

Each line in the result table is computed the same way:

```
    while (fahr <= upper) {
        celsius = 5 * (fahr - 32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
```

Note that $^\circ C = (5/9)(^\circ F - 32)$ is computed as

```
        celsius = 5 * (fahr - 32) / 9;
```

- **Integer division truncates**, i.e., any fractional part is discarded. Since 5 and 9 are integers, 5/9 would be truncated to zero and so all the Celsius temperature would be reported as zero.

$\Rightarrow$ Be careful with integer divisions.

printf revisited

```
#include <stdio.h>
int printf(const char *format, ...);
```

printf(3) is a general-purpose output formatting function.[12]

- 1st argument is the string of characters to be printed.
  - Each **%** indicates **where** one of the other arguments
  - and **in what form** it is to be printed.
- Each % in the 1st arg is paired with the 2nd, 3rd arg etc.

```
        printf("%d\t%d\n", fahr, celsius);
```

- %d, for instance, specifies an integer argument, so fahr and celsius are printed with a tab (\t) between them.

[12] Not part of the C language, but defined in ANSI X3.159-1989 (“ANSI C”)