Betriebssysteme

Tutorium 02

Peter Bohner

9. November 2025

1 Über uns

Über uns

Ich

- Peter Bohner (peter.bohner@student.kit.edu)
- armer Student im ~~9~~ **1.** Master Semester
- überraschenderweise Informatik-Student

Ihr

- ~~Name?~~ *Kennt ihr das Spiel* `Portal` *oder habt ihr regelmäßig* `KSP` *gespielt?*
- Studiengang? Ich tippe mal Info :)
- Programmierkenntnisse Richtung C?
- Schon mal was mit Betriebssystemen gemacht?
- Linux Erfahrung? M{1,2,3,4} Mac User?
- Erwartungen?

Orga

Vorlesung

- Übungsschein? Nein. Aber C Teil in der Klausur...
- Klausur: Drei Aufgaben (je 20P) mit gemischter Theorie und Praxis *Anders als (fast) alle Altklausuren!*
- Übungsblätter optional aber sinnvoll!

2 Multi-Programming

Multi-Programming

What is that?
Allow multiple processes to run *concurrently* (in parallel?)

How can they share a CPU? How can you make them take turns?
Two possible broad categories: *Cooperative Multitasking* and *Preemption*

Multi-Programming

How does Cooperative Multitasking work? Do you know any system using it?

- Processes *voluntarily* yield control so another one can take over
- When? Typically when they are blocking on some I/O or have finished their job for now
- Do you know any language or system using that concept? e.g. `await` in JavaScript or Python
- Can you think of any problems? Any advantages?
  - A bad actor or buggy program can bring the whole system to its knees!
  - It can lead to easier programs, as you can just assume you are never interrupted ⇒ Usage on small resource-constrained embedded systems

Multi-Programming

How does Preemption work?

- Processes get periodically interrupted by *Timer-interrupts*
- This transfers control to the OS, which can choose to schedule another process instead

Multi-Programming

But why even bother with this?

- Process A runs while Process B waits for an I/O device to complete its task

⇒ Better CPU and I/O utilization if you have a mixed workload

- What do the I/O devices and CPU need to support for this to be an improvement?

⇒ Interrupts, else the CPU needs to *continuously poll*

Multi-Programming

Are there any drawbacks?

- More complex
- Protection. How do you separate them from each other?
- Scheduler: Fairness and accounting

Multi-Programming

What are CPU and IO-bound Processes?

- CPU-bound: Process that rarely invokes I/O, unlikely to block, likes your CPU very much
- I/O-bound: Often blocks to wait for I/O, performs short CPU-bursts in between

3 C - Basics

Die Sprache C

Geschichte

- Wie auch der Vorgänger von und mit Dennis Ritchie bei Bell Labs um 1972 rum entwickelt
- Ist der Nachfolger von B
- Ist eine der meistgenutzen Sprachen

Eigenschaften

- Imperativ, Prozedural, *nicht* Objektorientiert
- Low level
- Manuelle Speicherverwaltung, Spaß mit Pointern und allen möglichen Implementierungsdetails
- CVE-Factory (*cough* https://kitctf.de/ *cough*) 

Die Sprache C

```c
#include <stdio.h>
#include "World.c"

int computeAnswer();

int main() {
    printf("Hey, your answer is %d\n", computeAnswer());

    return 0;
}

int computeAnswer() {
    return answerMeWorld();
}
```

Die Sprache C - Datentypen

Primitive

Name	Minimale Größe in Bytes	Größe bei mir in Bytes
char	1	1
short	2	2
int	2	4
float		4
double		8
long int	4	8
long long int	8	8
signed int	2	4
unsigned int	2	4

Oder mit bestimmter Größe

```c
#include <inttypes.h> int8_t a; int16_t b; int32_t c; int64_t d;
```

Die Sprache C - Booleans

Wie sieht ein boolean in C aus?
Gibt es nicht. `0` ist `false`, alle anderen Zahlenwerte sind `true`
Was macht man da also? Selbst basteln (oder `0` / `1` nutzen)

```c
// #include <stdbool.h>

typedef unsigned char bool;
#define false 0
#define true 1

int main() {
    bool hey = true;
    return 0;
}
```

Die Sprache C - Arrays

```c
int array[5] = {1, 2, 3, 4, 5};
int array[]  = {1, 2, 3, 4, 5};
int array[5];
```

Was ist das?

- Ein zusammenhängender Speicherbereich
- Größe ist Teil des *Typs* (`sizeof` geht, wenn Typ bekannt)
- Wie bekommt man die Größe zur *Laufzeit*? ⇒ `sizeof(a_pointer)`? *Heh. GAR NICHT!* Nur im Typ präsent.

Die Sprache C - Strings

```c
char[] myString = "world";
char[] myString = { 'w', 'o', 'r', 'l', 'd'};, '\0'};
```

Was sind strings?

- Ein `char`-Array mit *Null-Terminator* ⇒ Ein Byte länger!
- Nutzen den ASCII code

`strlen` bauen?
Wie sieht es aus? :)

Die Sprache C - Structs

```c
#include <stdbool.h>
#include <stdio.h>
struct User {
    bool userIsGoat;
    int age;
    char name[20];
};
int main() {
    struct User pete = { .userIsGoat = false, .age = 20, .name = "Pete"};
    printf(
        "I have a user %s (goaty: %s) of age %u\n",
        pete.name,
        pete.userIsGoat ? "true" : "false",
        pete.age
    );
    return 0;
}
```

Die Sprache C - Structs

Was sind Structs?

- Komplexere Datentypen
- Zusammengesetzt aus anderen Datentypen
- *KEINE KLASSE*. Warum? *Keine Vererbung*, keine Methoden

Die Sprache C - Structs

```c
#include <stdbool.h>
#include <stdio.h>
typedef struct User {
    bool userIsGoat;
    int age;
    char name[20];
} User_t;
int main() {
    User_t peter = { .userIsGoat = false, .age = 20, .name = "Peter"};
    printf(
        "I have a user %s (goaty: %s) of age %u\n",
        peter.name,
        peter.userIsGoat ? "true" : "false",
        peter.age
    );
    return 0;
}
```

Die Sprache C - Call-By-Value

```c
#include <stdio.h>
typedef struct User { int age; } User;

void foo(User user) {
    user.age = 200;
    printf("User age in foo is %d\n", user.age);
}

int main() {
    User user = { .age = 20 };
    foo(user);
    printf("User age in main %d\n", user.age);
    return 0;
}
```

Output?

`User age in foo is 200 User age in main 20`

Die Sprache C - Call-By-Reference

```c
#include <stdio.h>
typedef struct User { int age; } User;

void foo(User* user) {
    user->age = 200;
    printf("User age in foo is %d\n", user->age);
}

int main() {
    User user = { .age = 20 };
    foo(&user);
    printf("User age in main %d\n", user.age);
    return 0;
}
```

Output?

`User age in foo is 200 User age in main 200`

Die Sprache C - Pointer

```c
#include <stdio.h>
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main() {
    int first  = 20;
    int second = 5;
    printf("Before swap - First: %d, Second: %d\n", first, second);
    swap(&first, &second);
    printf("After swap  - First: %d, Second: %d\n", first, second);
    return 0;
}
```

Output?

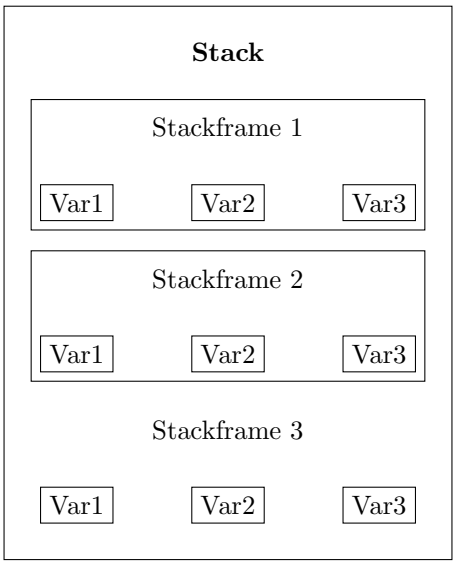
```
Before swap - First: 20, Second: 5 After swap - First: 5, Second: 20
```

Die Sprache C - Speicherlayout für Variablen

Programmbereiche für Variablen

Data	Globale Variablen
Stack	Lokale Variablen
Heap	Explizit zur Laufzeit angeforderter Speicher
...	

Die Sprache C - Stack



Die Sprache C - malloc

Speicher zur Laufzeit allokieren

```c
int main(int argc, char** argv) {
 char** stringsCopy[??];
 return 0;
}
```

Die Sprache C - malloc

Speicher zur Laufzeit allokieren

```c
#include <stdlib.h>

int main(int argc, char** argv) {
    char** stringsCopy = malloc(sizeof(char*) * argc);
    for(int i = 0; i < argc; i++) {
        stringsCopy[i] = argv[i]; // shallow copy
    }
    free(stringsCopy);
    return 0;
}
```

Die Sprache C - malloc

Speicher zur Laufzeit allokieren

```c
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
struct User { bool userIsGoat; int age; char name[20]; };
int main() {
    struct User* pete = malloc(sizeof(struct User));

    (*pete).userIsGoat = false;
    pete->age = 20;
    strcpy((*pete).name, "Pete");

    free(pete);
    return 0;
}
```

Die Sprache C - Forward declarations

Warum?

- Funktionen müssen *vor dem Aufruf deklariert* sein
- *Implementierung* kann auch irgendwann später erfolgen
- Implementierung *zur Entwicklung* nicht notwendig!

```c
int halfTruths(); // Deklaration

int main() {
   return halfTruths();
}

int halfTruths() { // Implementierung
    return 21;
}
```

Die Sprache C - Header files

Implementierung *zur Entwicklung* nicht notwendig!

- $\Rightarrow$ Header files
- Aufsplitten in Deklarationen und Definitionen
- Nutzer bindet Header zum Programmieren ein, *linkt* dann gegen eine Implementierung

Listing 1: World.h

```c
int answerMeWorld();
```

Listing 2: World.c

```c
int answerMeWorld() {
    return 42;
}
```

Die Sprache C - Mehrere Dateien

```c
#include <stdio.h>
#include "World.c"

int computeAnswer();

int main() {
    printf("Hey, your answer is %d\n", computeAnswer());

    return 0;
}

int computeAnswer() {
    return answerMeWorld();
}
```

„Hello, world!“

But now in C

We follow the arduous journey of Aeneas

- ? Start with ILIAS
- ? Obtain a text editor
- ? Obtain a compiler (we will probably use `gcc`)
- ? Draw the rest of the owl

4 Address Spaces

Address Space

Why can’t a process read another process’s memory?

- Let’s have a look at two processes...
- Address `0xFF` in Process A is *not* the same as Address `0xFF` in Process B

⇒ „If you can’t name it, you can’t touch it“

- You will learn about this in way more detail in later lectures

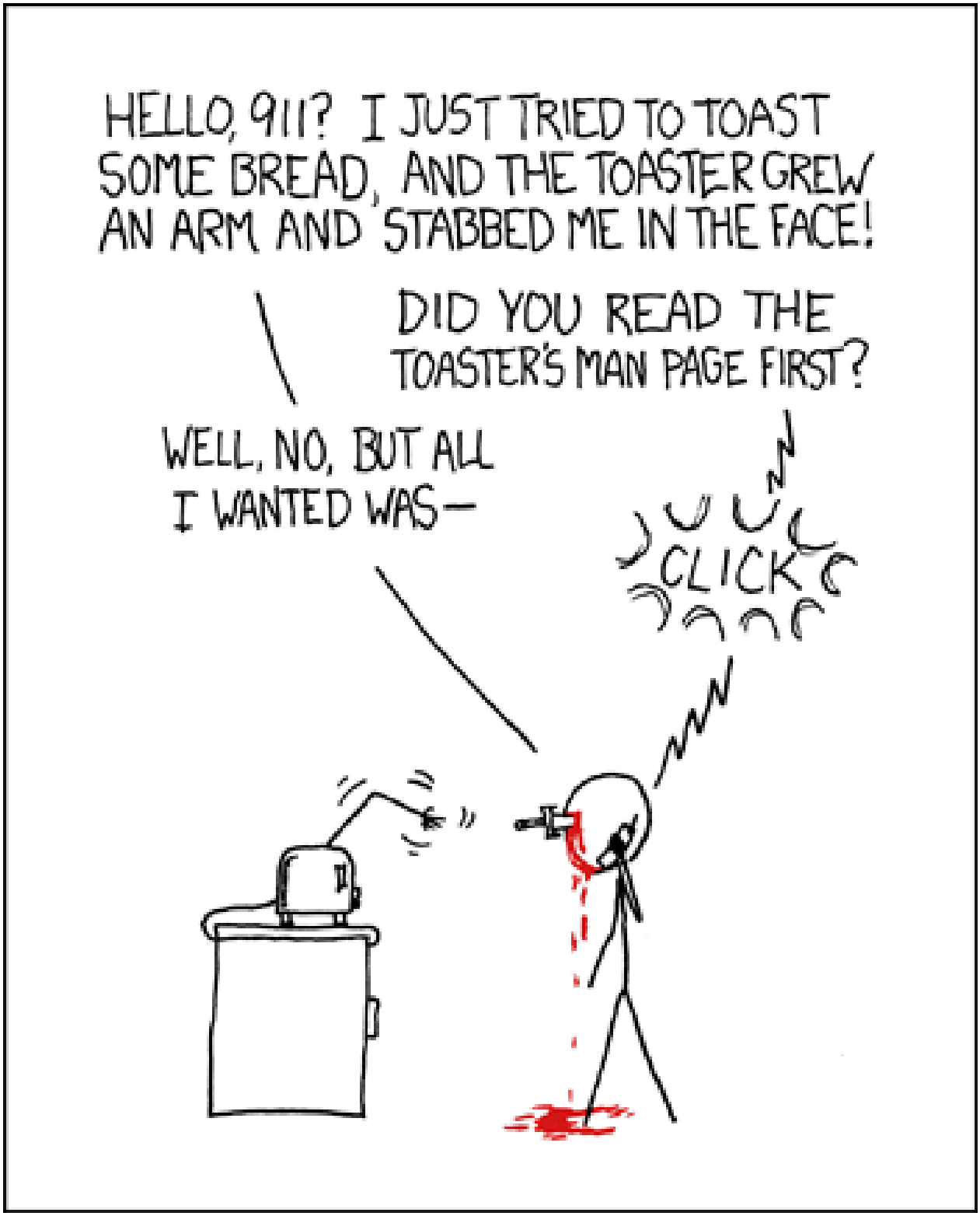
5 Fork

Fork

What does `fork` do?

- Creates a mostly identical child process
  - Same address space layout
  - Same data (as if it was all copied over)
  - Share open file descriptors
  - But the child has its own PID and address space
  - ... ⇒ `man fork`

~~Fork~~ *Knife*



Life is too short for man pages, and occasionally much too short without them. XKCD 293 - RTFM

Fork

Let's fork!

- Print `Fork failed` if the fork failed
- Print `I am the child` in the child
- Print `I am the parent of <child PID>` in the parent
- Wait for the child to die peacefully and print when it is dead in the parent

Fork

Let's fork! (More official solution)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    switch( (pid = fork()) ) {
        case -1:
            printf( "Error. Fork failed\n" );
            break;
        case 0:
            printf( "I am the child!\n" );
            break;
        default: // pid > 0
            printf( "I am the parent!\nChild PID is %d\n", pid );
            wait( NULL ); // wait and ignore the result
            printf( "Child terminated\n" );
    }
    return 0;
}
```

Fork

Is `fork` sufficient to create a small shell?

- No, you also need to *load the other program* in the child

$\Rightarrow$ `exec(vp|v|...)`