Betriebssysteme

Tutorium P 13

Peter Bohner

8. November 2024

ITEC - Operating Systems Group

Assigment 1

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Basics

- What is more important? OS or applications

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Basics

- What is more important? OS or applications $\rightarrow$ applications

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Basics

- What is more important? OS or applications $\rightarrow$ applications
- abstraction vs virtualisation

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Basics

- What is more important? OS or applications $\rightarrow$ applications
- abstraction vs virtualisation
- Why do we need protection on a mostly single-user system?

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Basics

- What is more important? OS or applications → applications
- abstraction vs virtualisation
- Why do we need protection on a mostly single-user system? → stability/reliability and hardening against vulnerabilities

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Basics

- What is more important? OS or applications → applications
- abstraction vs virtualisation
- Why do we need protection on a mostly single-user system? → stability/reliability and hardening against vulnerabilities
- What is a zombie process?

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Basics

- What is more important? OS or applications → applications
- abstraction vs virtualisation
- Why do we need protection on a mostly single-user system? → stability/reliability and hardening against vulnerabilities
- What is a zombie process? *rightarrow* stub in process table with exit-code, so it can be returned to *waitpid()*.

Feedback Exercise 1

General

- 16/70 submissions, with average 16/22 points
- Write brief answers, you are time constrained in the exam

Basics

- What is more important? OS or applications → applications
- abstraction vs virtualisation
- Why do we need protection on a mostly single-user system? → stability/reliability and hardening against vulnerabilities
- What is a zombie process? *rightarrow* stub in process table with exit-code, so it can be returned to *waitpid()*.

Feedback Exercise 1

Processes

- program vs process

Feedback Exercise 1

Processes

- program vs process
- What is a zombie process?

Feedback Exercise 1

Processes

- program vs process
- What is a zombie process? *rightarrow* stub in process table with exit-code, so it can be returned to *waitpid()*.

Program Anatomy

- Intel vs AT&T syntax

Feedback Exercise 1

Processes

- program vs process
- What is a zombie process? *rightarrow* stub in process table with exit-code, so it can be returned to *waitpid()*.

Program Anatomy

- Intel vs AT&T syntax
- Notation: parg vs *parg

User-Kernel boundary

- *system call number*

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

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

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

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

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

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

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

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

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

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

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

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
int	2	4

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

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

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

```
#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?

Die Sprache C - Booleans

Wie sieht ein boolean in C aus?

Gibt es erst seit C99. `0` ist `false`, alle anderen Zahlenwerte sind `true`

Was macht man da also?

Die Sprache C - Booleans

Wie sieht ein boolean in C aus?

Gibt es erst seit C99. `0` ist `false`, alle anderen Zahlenwerte sind `true`

Was macht man da also? Selbst basteln (oder `0` / `1` nutzen)

Die Sprache C - Booleans

Wie sieht ein boolean in C aus?

Gibt es erst seit C99. `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;
}
```

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

Was ist das?

```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

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)

```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(array)`?

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(array)`? *Heh.*

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(array)`? *Heh.* GAR NICHT! Nur im Typ präsent.

Die Sprache C - Strings

```
1 char[] myString = "world";
2 char[] myString = { 'w', 'o', 'r', 'l', 'd'};
```

Die Sprache C - Strings

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

Die Sprache C - Strings

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

Was sind strings?

- Ein `char`-Array mit *Null-Terminator* $\Rightarrow$ Ein Byte länger!

Die Sprache C - Strings

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

Was sind strings?

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

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

Die Sprache C - Structs

Was sind Structs?

- Komplexere Datentypen
- Zusammengesetzt aus anderen Datentypen
- KEINE KLASSE. Warum?

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?

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?

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?

```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?

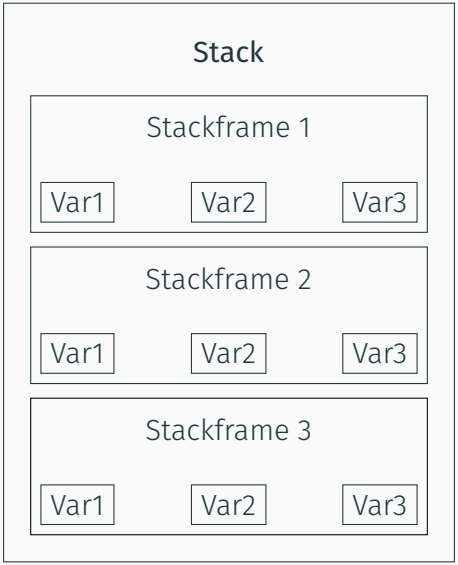
```
Before swap - First: 20, Second: 5
After swap - First: 5, Second: 20
```

Die Sprache C - Speicherlayout für Variablen

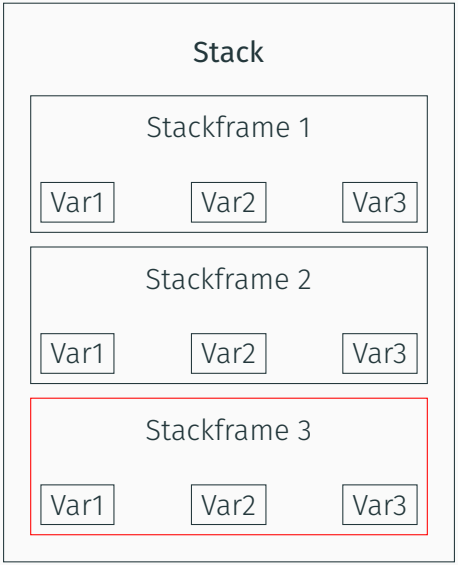
Programmbereiche für Variablen

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

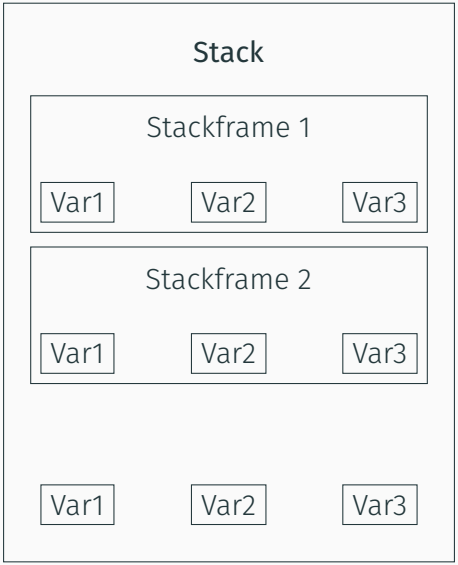
Die Sprache C - Stack



Die Sprache C - Stack



Die Sprache C - Stack



Speicher zur Laufzeit allokieren

```
1 int main(int argc, char** argv) {
2  char** stringsCopy[??];
3  return 0;
4 }
```

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;
}
```

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!

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

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 3: World.h

```
1 int answerMeWorld();
```

Listing 4: World.c

```
1 int answerMeWorld() {
2     return 42;
3 }
```

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();
}
```

Write an insert function

Task

```
1  void insert(int *a, size_t *len, int z);
2
3  void main() {
4      int a[10] = {0, 1, 2, 3, 5, 6, 7, 8, 9};
5      size_t len = 9;
6      insert(a, &len, 4);
7      assert(a[4] == 4);
8      assert(len == 10);
9  }
```

Write an insert function

Task

```
1 void insert(int *a, size_t *len, int z);
2
3 void main() {
4     int a[10] = {0, 1, 2, 3, 5, 6, 7, 8, 9};
5     size_t len = 9;
6     insert(a, &len, 4);
7     assert(a[4] == 4);
8     assert(len == 10);
9 }
```

What happens when we insert another?

Write an insert function

Task

```
1 void insert(int *a, size_t *len, int z);
2
3 void main() {
4     int a[10] = {0, 1, 2, 3, 5, 6, 7, 8, 9};
5     size_t len = 9;
6     insert(a, &len, 4);
7     assert(a[4] == 4);
8     assert(len == 10);
9 }
```

What happens when we insert another?

We write over the array boundary!

What methods does C offer to copy memory?

`memcpy`

What methods does C offer to copy memory?

`memcpy`

- \+ Potentially more efficient
- \- Can not handle overlaps

What methods does C offer to copy memory?

`memcpy`

+ Potentially more efficient
- Can not handle overlaps

`memmove`

What methods does C offer to copy memory?

`memcpy`

+ Potentially more efficient
- Can not handle overlaps

`memmove`

+ *Can* handle overlaps
- Potentially less efficient

Macros

Here will be demo!

Write `max(a, b)` as a function and as a macro.

Macros

Here will be demo!

Write `max(a, b)` as a function and as a macro.

- What differences in usage do you notice?

Macros

Here will be demo!

Write `max(a, b)` as a function and as a macro.

- What differences in usage do you notice?
- What is more efficient?

Macros

Here will be demo!

Write `max(a, b)` as a function and as a macro.

- What differences in usage do you notice?
- What is more efficient?
- Any pitfalls you might see?

Here will be demo!

Write `max(a, b)` as a function and as a macro.

- What differences in usage do you notice?
- What is more efficient?
- Any pitfalls you might see?

What happens with `max(a++, b)`?

Ende

XKCD 676 - Abstraction

F R A G E N?

Bis nächste Woche :)