

Übungen zu Systemprogrammierung 1

Ü6 – Besprechung A3 & Testen

Sommersemester 2019

Simon Ruderich, Dustin Nguyen, Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



Agenda



- 13.1 Besprechung Aufgabe 3: halde
- 13.2 Datentypen nach C11
- 13.3 Zeiger und undefiniertes Verhalten

Aufgabe 3: halde



Aufgabenstellung

- Eigene **Freispeicherverwaltung**
- `malloc()`, `realloc()`, `calloc()` & `free()`
- Makefile
- Testfälle

Vorstellen der Lösung von...

Standard Datentypen



- Größe der Datentypen ist von der Ausführungsumgebung abhängig
⇒ Hardware oder Betriebssystem
- Datentypen können mit `unsigned` als vorzeichenlos deklariert werden
- Typen und vorgegebene Wortbreiten ¹ in Bytes:

Typ	relative Größe	C11	64-Bit Linux
char		1	1
short int	≥ char	≥ 1	2
int	≥ short	≥ 2	4
long int	≥ int	≥ 4	8
long long int	≥ long int	≥ 8	8

¹Eigentlich ist der Wertebereich und nicht die Wortbreite definiert

Exakte Wortbreiten



- „exact-width integer“ geben Garantie über die tatsächliche Wortbreiten
- Die Datentypen lauten: `{u}int{8,16,32,64}_t`
- Limit-Makros definiert in `stdint.h`: `UINT{8,16,32,64}_MAX`
`INT{8,16,32,64}_{MAX,MIN}`
- Benutzt Makros für Format-Strings ² (siehe `inttypes.h(7p)`)

```
uint8_t c = 0x11;          int8_t c = -128;
uint16_t s = 0x1122;       int16_t s = -32768;
uint32_t i = 0x11223344;   int32_t i = -2147483648;
uint64_t l = 0x1122334455667788; int64_t l = -9223372036854775808;
printf("%PRIu8\n", c);     printf("%PRIi8\n", c);
printf("%PRIu16\n", s);   printf("%PRIi16\n", s);
printf("%PRIu32\n", i);   printf("%PRIi32\n", i);
printf("%PRIu64\n", l);   printf("%PRIi64\n", l);
```

²Konkatenation von String Literalen, wenn sie direkt aufeinander folgen.

```
char *x = "Hallo" "Welt"; ⇔ char *x = "HalloWelt";
```



- `{u}intptr_t`
 - Ein Ganzzahl-Typ, der den Zahlenwert eines Zeigers speichern kann
 - Kann zur Arithmetik auf Adressen verwendet werden
- `{s}size_t`
 - `size_t` ist der Ergebnis-Typ des `sizeof`-Operators
 - Beschreibt die Größe von Datentypen in Byte
 - Das Makro `SIZE_MAX` beinhaltet den maximal Wert
 - `ssize_t` ist vorzeichenbehaftet und wurde mit POSIX eingeführt
 - Der max. Wert ist `SSIZE_MAX`
 - Formatstring mit “%uz” bzw. “%z” für `ssize_t`



```
#include <stdio.h>
#include <stdint.h>
#include <stddef.h>

#define SIZE 10

int main(void) {
    int i_arr[SIZE] = {1};
    intptr_t start = (intptr_t) &i_arr[0];
    intptr_t end = (intptr_t) &i_arr[SIZE];

    intptr_t b_diff = end - start;
    printf("byte diff: %ld\n", b_diff);
}

> gcc intptr.c && ./a.out
byte diff: 40
```



- Fehler und Ergebnis werden mit dem selben Mechanismus mitgeteilt
- Zur Unterscheidung wird der Wertebereich des Ergebnis aufgeteilt
- `void *malloc(size_t size);`
 - Nutzt den Wert `NULL` als Rückgabewert im Fehlerfall
 - Alle anderen Werte sind gültige Zeiger auf allokierten Speicher
- `ssize_t write(int fd, const void *b, size_t nbytes)`
 - `ssize_t` ist das vorzeichenbehaftete Pendant zu `size_t`
 - Der Rückgabewert beschreibt die Anzahl der geschriebenen Bytes
 - Negativer Rückgabewerte beschreibt Fehler
 - $\Rightarrow$ `nbytes` darf nicht größer als `SSIZE_MAX` sein
 - $\Rightarrow$ Der Wertebereich halbiert sich



```
#include <stdio.h>
#include <limits.h>

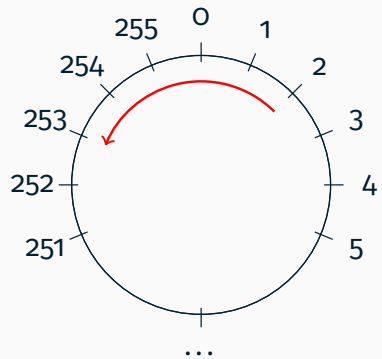
int main(void) {
    int i = INT_MAX + 1;
    unsigned u = UINT_MAX + 1;
    printf("%d %d\n", i, u);
}
```

- `gcc -Wall -Werror` warnt vor „overflow“ für `INT8_MAX + 1`
- Das Ergebnis ist zu groß für `int` und Überläufe sind undefiniert
- Bei vorzeichenlosen Typen kann jedoch kein Überlauf geschehen
- Stattdessen folgt nach dem Größten Wert die 0, bzw. `UINT_MAX` vor der 0

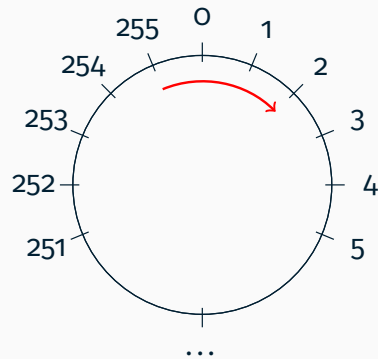
Überläufe mit Umbruch



```
uint8_t i = 2;  
i -= 5;  
/* i = 253 */
```



```
uint8_t i = 255;  
i += 3;  
/* i = 2 */
```



10

calloc(3): Umbruch vorzeichenloser Ganzzahlen



`calloc(3)` allokiert Speicher für ein Array mit `nmemb` Elementen, die jeweils `size` Byte groß sind.

```
static void *calloc(size_t nmemb, size_t size) {  
    size_t total = nmemb * size;  
    void *p = malloc(total);  
    if (p) {memset(p, 0, total);}  
    return p;  
}  
  
int main(void) {  
    size_t s = 16;  
    void *p = calloc((SIZE_MAX/16)+3, s);  
    if (p) {memset(p, 0xff, 5 * s);}  
}
```

- `size_t` ist groß genug jeden der Operanden zu speichern
- Das Produkt der Multiplikation ist jedoch zu groß
- `total` wird der Wert 32 zugewiesen (bei 8 Byte Wortbreite)
- `memset` in `main` schreibt über den zugewiesenen Buffer hinaus

11

calloc(3): Von den Großen lernen



```
#define __glibc_unlikely(cond) \
    __builtin_expect((cond), 0)

void *
__libc_calloc(size_t n, size_t elem_size)
{
    /* ... */
    INTERNAL_SIZE_T sz;
    ptrdiff_t bytes;
    if (__glibc_unlikely(
        __builtin_mul_overflow (
            n, elem_size, &bytes)))
    {
        __set_errno (ENOMEM);
        return NULL;
    }
    sz = bytes;
    /* ... */
}
```

Listing 1: glibc malloc/malloc.c:3365 (version 2.29)

- Arbeitet mit übersetzerabhängigen Erweiterungen
- Übersetzer ist verantwortlich die Multiplikation mit Befehlen zu ersetzen, die für die jeweilige Architektur Überläufe erkennen kann

Undefiniertes Verhalten



When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose.

- Nicht alles ist im C-Standard definiert
- Z.B. Vergleiche von Zeigern, die auf versch. Arrays zeigen
- In solchen Fällen darf der Übersetzer „frei gestalten“
- Der übersetzte Code kann der Erwartung des Programmierers widersprechen

Beispiel für undefiniertes Verhalten



```
#include <stdio.h>
int main(void) {
    long x[4], y[4];
    void *p = &x[0], *q = &y[4];
    printf("%p %p %d\n", p, q, p == q);
}
```

```
> gcc -O0 ./ptr_comparison.c && ./a.out
0x7ffd4c6eae20 0x7ffd4c6eae20 1
> gcc -O2 ./ptr_comparison.c && ./a.out
0x7ffec72d95d0 0x7ffec72d95d0 0
```

- Ohne Optimierung: Vergleich der Zeiger während der Laufzeit
- Mit -O2: Auswertung des Vergleichs `p==q` während der Übersetzung
- Vergleich ist undefiniert, da `p` und `q` auf versch. Objekte zeigen
- Der Übersetzer darf annehmen, dass beide Zeiger verschieden sind³

³Details im Appendix

Appendix

Details zum undefinierten Zeigervergleich (1)



```
#include <stdio.h>
int main(void) {
long x[4], y[4];
void *p = &x[0], *q = &y[4];
printf("%p %p %d\n", p, q, p==q);
}

> gcc -O0 ./ptr_comparison_01.c
> ./a.out
0x7ffd4c6eae20 0x7ffd4c6eae20 1
```

```
#include <stdio.h>
int main(void) {
long x[4], y[4];
void *p = &x[4], *q = &y[0];
printf("%p %p %d\n", p, q, p==q);
}

> gcc -O2 ./ptr_comparison_02.c
> ./a.out
0x7ffec72d95d0 0x7ffec72d95d0 0
```

- Verwendet wurde der gcc in der Version „8.3.0 (Debian 8.3.0-6)“ aus dem Debian Package-Repository „buster“
- Der Übersetzer ordnet je nach Optimierungsstufe die Arrays auf dem Stack um