

Besprechung der halde¹

Tafelübung Systemprogrammierung

Andreas Schärfl <andreas.schaertl@fau.de>

SS 2021

¹<https://www.schaertl.me/sp1>

Besprechung

- Jetzt Besprechung einer beispielhaften Aufgabe.
- Be excellent to each other.

Overflow

- Selbst `size_t` mit seinen 64 Bit (auf amd64) kann überlaufen.
- Zum Beispiel passiert das in `calloc`!

Behandeln²

```
void *calloc(size_t m, size_t n) {  
    void *p;  
    size_t *z;  
    if (n && m > (size_t)-1/n) {  
        errno = ENOMEM;  
        return 0;  
    }  
}
```

[...]

²Aus: musl libc: <https://git.musl-libc.org/cgit/musl/tree/src/malloc/calloc.c?h=rs-1.0>

Fehlerbehandlung mit `errno`

Fehlerbehandlung mit `errno`

- Viele Funktion geben ihren genauen Fehler nicht genau zurück.
 - Zum Beispiel `fgets` mit `NULL` oder `chdir` mit `-1`.
- Was genau passiert ist steht in der globalen³ Variable `errno`. Der Typ von `errno` ist `int`.
- Viele vordefinierte Fehlercodes wie `ENOMEM`, `EINVAL` oder `ENOENT`
 - Nachschlagbar zum Beispiel mit `man 3 errno`.
- Der Ansatz mit der globalen Variable ist heute nicht bei allen beliebt⁴ aber auch nicht unüblich.

³ und `thread-local`

⁴ <https://stackoverflow.com/a/42622258>

Fehlerbehandlung mit errno

```
void perror(const char *prefix) {  
    int my_errno = errno; // read global variable  
    const char *error_string = strerror(my_errno);  
    fprintf(stderr, "%s: %s\n", prefix, error_string);  
}
```

- Funktionen wie `perror` lesen `errno` selber auch nur aus.
- Fehlerbehandlung so früh wie nur möglich! Aufruf von Bibliotheksfunktionen könnte die `errno` ändern.

Fehlerbehandlung mit errno

- Viele Funktionen garantieren nur, dass sie *im Fehlerfall* die `errno` setzen. Was im Erfolgsfall passiert ist gerne mal nicht definiert. Au weia!
- Im Zweifelsfall vorher `errno = 0` setzen.

```
const struct passwd *get_user(uid_t uid) {
    errno = 0; // required!
    const struct passwd *user_info = getpwuid(uid);
    if (!user_info) {
        if (errno == 0) {
            fprintf(stderr, "user with uid=%d not found\n", uid);
            exit(EXIT_FAILURE);
        } else {
            perror("getpwuid");
            exit(EXIT_FAILURE);
        }
    }
    return user_info;
}
```

Coole Typen: bool

Cooler Typen: `bool`

- Sehr alte Versionen von C hatten keinen Typ `bool`. Man nahm stattdessen einfach `int` o.Ä.
- Seit C99 gibt es aber den Typ `bool` mit den bekannten Werten `true` und `false`.
- Header `stdbool.h` muss inkludiert werden.

Coole Typen: `size_t`

Cooler Typen: `size_t`

- Es gibt mehr als nur einen schönen `int`.
- Besonders Sinnvoll: `size_t` ist ein `unsigned` Typ für Größenangaben jeder Art.
 - Funktionen wie `strlen` oder `strcpy` arbeiten alle mit Typ `size_t`.

Das ist wirklich wichtig!

- `INT_MAX` auf `amd64` ist 2.147.483.647! Das reicht gerade mal um 2 GiB zu adressieren. Der `int` ist also total ungeeignet für Speicheroperationen in portablen Programmen.

Lokale static Variablen

Lokale static Variablen

```
int hashmap_get(keytype k) {  
    static bool is_initialized = false;  
    if (!is_initialized) {  
        initialize();  
        is_initialized = true;  
    }  
  
    return get(k);  
}
```

- Lokale Variablen können static sein.
- Sie verhalten sich dann wie globale Variablen, sind aber im Scope auf eine Funktion begrenzt.

make und die Flags

make und die Flags

- Im Makefile beschreibt man die einzelnen Schritte, die zum Bauen eines Projektes notwendig sind.
 - Typischerweise teilt man das ganze in zwei Kategorien ein.
 - *.c-Datei → *.o-Datei (*Übersetzen*)
 - *.o-Dateien → Programm, auf Windows *.exe (*Linken*)
- Für das Übersetzen sind die CFLAGS zuständig.
- Für das Linken die LDFLAGS.

Aufgepasst

- CFLAGS braucht es also beim Linken nicht.
- LDFLAGS braucht es beim Übersetzen nicht.

make und die Flags

- Wichtig sind die Abhängigkeiten. Header nicht vergessen!
- `make` baut dann nur das neu, was neu gebaut werden muss.
- Sind alle Abhängigkeiten richtig, könnt ihr mit `make -jN` euer Projekt parallel mit `N` Threads ausführen. Cool!

make und die Flags

```
1 [|||||] [100.0%] 17 [|||||] [100.0%]
2 [|||||] [100.0%] 18 [|||||] [100.0%]
3 [|||||] [100.0%] 19 [|||||] [100.0%]
4 [|||||] [100.0%] 20 [|||||] [100.0%]
5 [|||||] [100.0%] 21 [|||||] [100.0%]
6 [|||||] [100.0%] 22 [|||||] [100.0%]
7 [|||||] [100.0%] 23 [|||||] [100.0%]
8 [|||||] [100.0%] 24 [|||||] [100.0%]
9 [|||||] [100.0%] 25 [|||||] [100.0%]
10 [|||||] [100.0%] 26 [|||||] [98.0%]
11 [|||||] [100.0%] 27 [|||||] [100.0%]
12 [|||||] [100.0%] 28 [|||||] [100.0%]
13 [|||||] [100.0%] 29 [|||||] [100.0%]
14 [|||||] [98.7%] 30 [|||||] [100.0%]
15 [|||||] [100.0%] 31 [|||||] [100.0%]
16 [|||||] [100.0%] 32 [|||||] [100.0%]
Mem| 9.92G/126G Tasks: 103, 1 thr; 32 running
Swp| 0K/0K Load average: 6.86 3.93 1.64
Uptime: 276 days(!), 02:02:50

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
5118 ru64tiji 20 0 289M 70268 13772 S 0.0 0.1 4:25.59 emacs --daemon
5119 ru64tiji 20 0 289M 70268 13772 S 0.0 0.1 0:00.00 | gmain
20712 ru64tiji 20 0 78000 63408 5024 S 0.0 0.0 0:00.63 -zsh
27579 ru64tiji 20 0 78004 63484 5072 S 0.0 0.0 0:00.84 -zsh
30819 ru64tiji 20 0 5720 2088 1872 S 0.0 0.0 0:00.00 | /usr/bin/make -j32
30822 ru64tiji 20 0 6100 2892 2128 S 0.0 0.0 0:00.01 | /usr/bin/make -s -f CMakeFiles/Makef
31463 ru64tiji 20 0 5588 2104 1892 S 0.0 0.0 0:00.00 | | /usr/bin/make -s -f tests/CMakeFi
31466 ru64tiji 20 0 2388 768 696 S 0.0 0.0 0:00.00 | | | /bin/sh -c cd /home/cip/2014/r
31467 ru64tiji 20 0 158M 89476 55692 R 35.8 0.1 0:00.54 | | | | /proj/ciptmp/ru64tiji/ma/cl
F1Help F2Setup F3Search F4Filter F5Sorted F6Collap F7Nice F8Nice F9Kill F10Quit
```

→ make und vergleichbare Tools sind clever und sparen viel Zeit.
Man muss sie aber richtig konfigurieren.

Fragen?