Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Testen

Florian Schmaus, Simon Schuster

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
https://www4.cs.fau.de

28. Mai 2018

Entwickeln – EAN Aufgabe

- Auf leistungsfähigen Rechnern arbeiten, bspw. `faui00a-y` oder `faui02a-y`

↝ Rechnerliste:
`https://wwwcip.cs.fau.de/cipPools/roomIndex.en.html`

- Entfernt arbeiten per ssh:
  - Per ssh anmelden: `ssh nutzer@fauiXXx.cs.fau.de`
  - Auf andere Nutzer prüfen: `who`
  - Im Hintergrund arbeiten lassen: TMUX
    1. Starten: `tmux`

    ↝ Injektion wie gewohnt starten
    2. Abkoppeln: `Strg + b` dann d
    3. Wieder verbinden: tmux attach
    4. Mehr Informationen:
    + `man tmux`
    + `https://robots.thoughtbot.com/a-tmux-crash-course`
    + `https://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/`

Überblick

1. Abfangen von Integer-Fehlern
2. Testen
3. Übungsaufgabe

Überblick

1 Abfangen von Integer-Fehlern

2 Testen

3 Übungsaufgabe

Umgang mit Ganzzahlfehlern

- C bietet viele subtile Fehlermöglichkeiten
- Im C-Quiz haben wir einige kennengelernt
- Was uns noch fehlt:
  - *Wie verhält man sich als Programmierer richtig?*

  ⇝ Heute ein paar Beispiele

Addition

Was soll da schon schiefgehen...

```
1  unsigned int func(unsigned int a, unsigned int b) {
2    return a + b;
3  }
4
```

Vorbedingungstest

```
1  #include <limits.h>
2  unsigned int func(unsigned int a, unsigned int b) {
3    if (UINT_MAX - a < b) { raise("wraparound"); }
4    return a + b;
5  }
6
```

Nachbedingungstest

```
1  unsigned int func(unsigned int a, unsigned int b) {
2    unsigned int ret = a + b;
3    if (ret < a) { raise("wraparound"); }
4    return ret;
5  }
6
```

Subtraktion

Was soll da schon schiefgehen...

```
1 unsigned int func(unsigned int a, unsigned int b) {
2   return a - b;
3 }
4
```

Vorbedingungstest

```
1 unsigned int func(unsigned int a, unsigned int b) {
2   if (a < b) { raise("wraparound"); }
3   return a - b;
4 }
5
```

Nachbedingungstest

```
1 unsigned int func(unsigned int a, unsigned int b) {
2   unsigned int ret = a - b;
3   if (ret > a) { raise("wraparound"); }
4   return ret;
5 }
6
```

Multiplikation

Was soll da schon schiefgehen...

```
1 unsigned int func(unsigned int a, unsigned int b) {
2   return a * b;
3 }
4
```

Vorbedingungstest

```
1 #include <limits.h>
2 unsigned int func(unsigned int a, unsigned int b) {
3   if (a == 0 or b == 0) { return 0; }
4   if (UINT_MAX / a < b) { raise("wraparound"); }
5   return a * b;
6 }
7
```

Explizite Typumwandlung

Was soll da schon schiefgehen...

```
1 unsigned int func(signed int a) {
2   return (unsigned int) a; /* keine Compilerwarnung wg. Cast */
3 }
4
```

Vorbedingungstest

```
1 unsigned int func(signed int a) {
2   if (a < 0) { raise("wraparound"); }
3   return (unsigned int) a;
4 }
5
```

Explizite Typumwandlung

Was soll da schon schiefgehen...

```
1  unsigned char func(unsigned long int a) {
2    return (unsigned char) a; /* keine Compilerwarnung wg. Cast */
3  }
4
```

Vorbedingungstest

```
1  unsigned char func(unsigned long int a) {
2    if (a > UCHAR_MAX) { raise("overflow"); }
3    return (unsigned char) a; /* keine Compilerwarnung wg. Cast */
4  }
5
```

Explizite Typumwandlung

Was soll da schon schiefgehen...

```
1 signed char func(unsigned long int a) {
2   return (signed char) a; /* keine Compilerwarnung wg. Cast */
3 }
4
```

Vorbedingungstest

```
1 #include <limits.h>
2 signed char func(unsigned long int a) {
3   if (a > SCHAR_MAX) { raise("overflow"); }
4   return (signed char) a;
5 }
6
```

Explizite Typumwandlung

Was soll da schon schiefgehen...

```
1 signed char func(signed long int a) {
2   return (signed char) a; /* keine Compilerwarnung wg. Cast */
3 }
4
```

Vorbedingungstest

```
1 #include <iso646.h>
2 #include <limits.h>
3 signed char func(signed long int a) {
4   if (a < SCHAR_MIN or SCHAR_MAX < a) { raise("overflow"); }
5   return (signed char) a; /* keine Compilerwarnung wg. Cast */
6 }
7
```

Addition

Was soll da schon schiefgehen...

```
1  signed int func(signed int a, signed int b) {
2    return a + b;
3  }
4
```

Vorbedingungstest

```
1  #include <iso646.h>
2  #include <limits.h>
3  signed int func(signed int a, signed int b) {
4    if ((b > 0 and a > INT_MAX - b)
5     or (b < 0 and a < (INT_MIN - b))) { raise("overflow"); }
6    return a + b;
7  }
8
```

Subtraktion

Was soll da schon schiefgehen...

```
1 signed int func(signed int a, signed int b) {
2   return a - b;
3 }
4
```

Vorbedingungstest

```
1 #include <iso646.h>
2 #include <limits.h>
3 signed int func(signed int a, signed int b) {
4   if ((b > 0 and a < INT_MIN + b)
5    or (b < 0 and a > INT_MAX + b)) { raise("overflow"); }
6   return a - b;
7 }
8
```

Division

Was soll da schon schiefgehen...

```
1 signed long func(signed long a, signed long b) {
2   return a / b;
3 }
4
```

Vorbedingungstest

```
1 #include <iso646.h>
2 #include <limits.h>
3 signed long func(signed long a, signed long b) {
4   if (b == 0) { raise("division by 0"); }
5   return a / b;
6 }
7
```

Division

- Reicht das schon?

Was soll da schon schiefgehen...

```
1  signed long func(signed long a, signed long b) {
2    if (b == 0) { raise("division by 0"); }
3    return a / b;
4  }
5
```

Vorbedingungstest

```
1  #include <iso646.h>
2  #include <limits.h>
3  signed long func(signed long a, signed long b) {
4    if (b == 0) { raise("division by zero"); }
5    if (a == LONG_MIN and b == -1) { raise("overflow"); }
6    return a / b;
7  }
8
```

Modulo

Was soll da schon schiefgehen...

```
1 signed long func(signed long a, signed long b) {
2   return a % b;
3 }
4
```

Vorbedingungstest

```
1 #include <iso646.h>
2 #include <limits.h>
3 signed long func(signed long a, signed long b) {
4   if (b == 0) { raise("division by zero"); }
5   if (a == LONG_MIN and b == -1) { raise("overflow"); }
6   return a % b;
7 }
8
```

Negation

Was soll da schon schiefgehen...

```
1 signed long func(signed long a) {
2   return -a;
3 }
4
```

Vorbedingungstest

```
1 #include <limits.h>
2 signed long func(signed long a) {
3   if (a == LONG_MIN) { raise("overflow"); }
4   return -a;
5 }
6
```

Multiplikation

Was soll da schon schiefgehen...

```
1  signed int func(signed int a, signed int b) {
2    return a * b;
3  }
4
```

Vorbedingungstest

```
1  #include <iso646.h>
2  #include <limits.h>
3  signed int func(signed int a, signed int b) {
4    if (a == 0 or b == 0) { return 0; }
5    if (a > 0 and b > 0 and a > INT_MAX / b) { raise("overflow"); }
6    if (a > 0 and b < 0 and b < INT_MIN / a) { raise("overflow"); }
7    if (a < 0 and b > 0 and a < INT_MIN / b) { raise("overflow"); }
8    if (a < 0 and b < 0 and b < INT_MAX / a) { raise("overflow"); }
9    return a * b;
10 }
11
```

Überblick

1 Abfangen von Integer-Fehlern

2 Testen

3 Übungsaufgabe

Testen

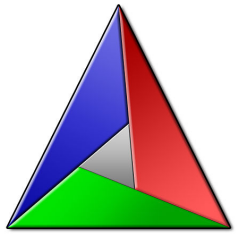
- Erste Grundregeln:
  - Testbarkeit von vornherein einplanen
    - ⇝ Feingranulare Testfälle
    - ⇝ *Ein Testfall für jede einzelne Funktion!*
  - Teste Datentypen an ihren Wertebereichsgrenzen
    - → `INT16_MAX, INT16_MIN, ...`
  - *Minimale Testabdeckung*: erreichbarer Code/Zeilenüberdeckung
- Hilfsmittel:
  - Automatisierte Testinfrastruktur
  - Code-Coverage-Analysewerkzeug

Vorsicht!

- Testfälle können nur die Anwesenheit von Fehlern zeigen
- Nicht deren Abwesenheit! (→ vgl. formale Verifikation)

⇝ Alle *Randfälle* erkennen und abdecken

Testfallintegration mit CMake



- Integration von Tests im Softwareprojekt
- Automatisierte Ausführung und Auswertung von Testläufen
- Konfigurationsdatei: `tests/CMakeLists.txt`
  - Ausführbares Target:

    ```
    add_executable(plus_test plus_test.c)
    ```
  - Hinzubinden der zu testenden Bibliothek:

    ```
    target_link_libraries(plus_test mathe)
    ```
  - Bekanntmachen als Testfall:

    ```
    add_test(MatheTest_PLUS plus_test)
    ```
- Ausführung der Tests: `make && make test`
- Automatische Testauswertung:
  - Anhand Rückgabewert (0 → OK, -1 → Fehler)
  - Notfalls auch Parsen von Ausgaben

Codeüberdeckung: gcov/lcov

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	**Lines:**	49	114	43.0 %
Test:	coverage.lcov	**Functions:**	6	14	42.9 %
Date:	2018-05-26 00:38:08	**Branches:**	19	72	26.4 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

Directory	Line Coverage		Functions		Branches	
src		43.0 %	49 / 114	42.9 %	6 / 14	26.4 %	19 / 72

Generated by: LCOV version 1.13-14-ga5dd952

- Werkzeug aus der gcc-Toolchain
- Instrumentierung des Binärcodes $\rightsquigarrow$ *Laufzeitkosten*
- Protokollieren der Programmausführung
  - Wie oft wird jede Codezeile ausgeführt?
  - Welche Zeilen werden überhaupt ausgeführt?
  - Welche Verzweigungen wurden genommen?
- HTML Ausgabe: lcov

$\rightarrow$ Tests solange erweitern, bis *vollständige Verzweigungsüberdeckung* erreicht!

Aufdecken von Laufzeitfehlern – AddressSanitizer

- *„Im besten Fall kracht es bei Speicherzugriffsfehlern!"*
- In Übungen: Verwendung von Clang AddressSanitizer [1][1]
- Checks zur Laufzeit
  - falsche Verwendung von Zeigern
  - nicht-definierte Integer-Operationen
  - Lesen uninitialisierten Speichers
  - Integer-Überlauf
  - ...

Entdeckt Fehler . . .

. . . nur, wenn die verwendeten Testfälle diese auslösen.

☞ **zur Laufzeit**

- Laufzeitkosten: ≈ 2 x

[1] `http://clang.llvm.org/docs/AddressSanitizer.html`

Clang AddressSanitizer – Verwendung

```
1  // program.cpp
2  int main(int argc, char **argv) {
3    int *array = new int[100];
4    delete[] array;
5    return array[argc];  // BOOM
6  }
```

```
$ clang++ -O1 -g -fsanitize=address program.cpp
```

```
$ ./a.out
```

```
ERROR: AddressSanitizer: heap-use-after-free on address 0x602e0001fc64 at pc ...
```

- Wird von cmake-Skripten automatisch verwendet, wenn
  - Debugging aktiviert ist
  - und `clang` als Compiler verwendet wird
  - siehe `cmake/sanitizer.cmake`
- Aufruf von cmake

↝ `CC=clang CXX=clang++ cmake -DCMAKE_BUILD_TYPE=Debug ..`

Statische Programmanalyse – Clang Static Analyzer

- Analyse des Quellcodes (C, C++, Objective-C)
- Keine Ausführung des Codes auf Hardware $\rightsquigarrow$ „statische Analyse”
- Eingabewerte als *symbolisch* angenommen
  $\rightarrow$ *symbolische Ausführung/Erreichbarkeitsanalyse*
- Verfügbare Checks[2]
  - Wertebereichsanalysen: Division mit Null
  - Verwendung uninitialisierter Variablen
  - ...
- Analyse ist *nicht fehlerfrei* (engl. sound)
  - Nicht möglich alle Fehler zu finden (engl. false negatives)
- Analyse ist *nicht präzise* (engl. precise)
  - Falsche positive Befunde sind möglich (engl. false positives)

[2] `http://clang-analyzer.llvm.org/available_checks.html`

Clang Static Analyzer – Verwendung

```
1  void test() {
2    int i, a[10];
       1 'i' declared without an initial value →
3    int x = a[i]; // warn: array subscript is undefined
       2 ← Array subscript is undefined
4  }
```

- Einzelne Datei überprüfen: `scan-build clang -c program.c`
- Übung: Aufruf von `scan-build` mit `cmake` als Argument

  ⇝ `CC=clang CXX=clang++ scan-build cmake ..`

  ⇝ `scan-build make`
- Fehler/Warnungen gefunden → Ausgabe von HTML Dateien
- Aufruf von `scan-view` wie in Ausgabe beschrieben

Verzeichnisstruktur

Quellverzeichnis

```
% tree ~/source
~/source
|-- CMakeLists.txt
|-- include
|   `-- mathe.h
|-- src
|   |-- CMakeLists.txt
|   |-- abs.c
|   `-- plusminus.c
`-- tests
    |-- CMakeLists.txt
    |-- abs_test.c
    `-- plus_test.c
```

Binärverzeichnis

```
% cd ~/binary
% cmake ../
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Checking whether C compiler has -isysroot
...
-- Configuring done
-- Generating done
-- Build files have been written to: ~/build
% make
[ 20%] Building C object src/CMakeFiles/mathe.dir/plusminus.c.o
[ 40%] Building C object src/CMakeFiles/mathe.dir/abs.c.o
Linking C static library libmathe.a
[ 60%] Built target mathe
Scanning dependencies of target abs_test
[ 80%] Building C object tests/CMakeFiles/abs_test.dir/abs_test.c.o
Linking C executable abs_test
[ 80%] Built target abs_test
Scanning dependencies of target plus_test
[100%] Building C object tests/CMakeFiles/plus_test.dir/plus_test.c.o
Linking C executable plus_test
[100%] Built target plus_test
% make test
Running tests...
Test project ~/build
    Start 1: MatheTest_PLUS
1/2 Test #1: MatheTest_PLUS .......   Passed    0.00 sec
    Start 2: MatheTest_ABS
2/2 Test #2: MatheTest_ABS  ........***Failed    0.00 sec

50% tests passed, 1 tests failed out of 2

Total Test time (real) =   0.02 sec

The following tests FAILED:
     2 - MatheTest_ABS (Failed)
Errors while running CTest
```

Überblick

1 Abfangen von Integer-Fehlern

2 Testen

3 Übungsaufgabe

Aufgabe 4 – Testen

- Verwendung von GNU/Linux (kein eCos mehr)
- Ziele
  1. Objektbasierter Softwareentwurf
  2. Testfallentwurf
     - Vollständige Pfadeüberdeckung
     - Abdecken aller Randfälle
  3. Implementierung von Software und Testfällen
     - Getrennte Implementierung von Software und Testfällen
     - Möglichst durch verschiedene Übungsteilnehmer
- Implementiert werden soll eine *Prioritätswarteschlange*
- Einfügen, Entfernen, *Iterieren*

↝ `for (... x = ...; x = ...; ++x) ...!`

- Implementierung?

Iterator – 1. Versuch

- *Datenstruktur als Array* im Header vereinbaren
- Zugriff durch Zeigerarithmetik

```
1 typedef struct Element { ... } Element;
2 Element elements[ELEMENTS_SIZE];
3 ...
4   for (size_t i = 0; i < ELEMENTS_SIZE; ++i)
5     { use(elements[i]); }
6
```

- *Vorteile:*
  - Einfache Implementierung
  - Für den Compiler leicht zu optimieren
- *Nachteil:* Implementierung offen gelegt

$\rightsquigarrow$ Verpflichtung ggü. Benutzer

Iterator – 2. Versuch

- *Iterator als Teil des Objekts*
- Header:

```
1 typedef struct Elements Elements;
2 void El_reset_iterator(Elements *self);
3 void El_next(Elements *self);
4 bool El_isAtEnd(Elements *self);
5 int64_t El_iterator_value(Elements *self);
6
```

- Verwendung:

```
1 El_reset_iterator(dings);
2 while(!El_isAtEnd(dings)) {
3   use(El_iterator_value(dings));
4   El_next(dings);
5 }
6
```

Iterator – 2. Versuch

- Implementierung:

```
1  typedef struct Element { int64_t value; } Element;
2  struct Elements {
3    Element elements[ELEMENTS_SIZE];
4    Element *it;
5  };
6  void El_reset_iterator(Elements *self)
7    { self->it = &self->elements }
8  void El_next(Elements *self)
9    { self->it = self->it + 1; }
10 bool El_isAtEnd(Elements *self)
11   { return self->it
12       == &(self->elements[ELEMENTS_SIZE]); }
13 int64_t El_iterator_value(Elements *self)
14   { return self->it->value; }
15
```

- *Vorteil:* Kapselung sehr gut
- *Nachteile:*
  - Für den Compiler evtl. nicht mehr optimierbar (Schleife ausrollen)
  - So nur ein Iterator gleichzeitig möglich

Iterator – 3. Versuch

- *Iterator als eigenes Objekt*
- Header:

```
1  typedef struct Elements Elements;
2  typedef struct El_Iterator El_Iterator;
3
4  El_Iterator *El_begin(Elements *self);
5  void El_Iterator_destroy(El_Iterator *self);
6  void El_Iterator_next(El_Iterator *self);
7  bool El_Iterator_isAtEnd(El_Iterator *self);
8  int64_t El_Iterator_value(El_Iterator *self);
9
```

- Verwendung:

```
1  El_Iterator *it;
2  for (it = El_begin(dings);
3       not El_Iterator_isAtEnd(it);
4       El_Iterator_next(it)) {
5    use(El_Iterator_value(it))
6  }
7  El_Iterator_destroy(it);
8
```

Iterator – 3. Versuch

- Implementierung:

```
1  typedef struct Element { int64_t value; } Element;
2  struct Elements { Element elements[ELEMENTS_SIZE]; };
3  struct El_Iterator {
4    Element *position;
5    Element *end;
6  };
7
8  El_Iterator *El_begin(Elements *self) {
9    El_Iterator *ret = malloc(sizeof(El_Iterator));
10   if (ret == NULL) { return NULL; }
11   ret->position = self->elements;
12   ret->end = &self->elements[ELEMENTS_SIZE];
13   return ret;
14 }
15
16 void El_Iterator_next(El_Iterator *self)
17   { self->position += 1; }
18 bool El_Iterator_isAtEnd(El_Iterator *self) { ... }
19 int64_t El_Iterator_value(El_Iterator *self) { ... }
20 void El_Iterator_destroy(El_Iterator *self) { ... }
21
```

Iterator – 3. Versuch

- *Vorteile:*
  - Vollständige Kapselung
  - Beliebig viele Iteratoren möglich
- *Nachteil:*
  - Iterator muss nach Gebrauch beseitigt werden
  - Compiler hat evtl. Probleme zu optimieren

Literatur I

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov.
AddressSanitizer: A fast address sanity checker.
In *Proceedings of the USENIX Annual Technical Conference*, pages 309–318, 2012.