Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Testen

Tobias Klaus, Florian Schmaus, Peter Wägemann

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

6. Juni 2016

Überblick

Annahmen

- C99
- x86 bzw. x86-64, d. h.
  - vorzeichenbehaftete Integer als Zweierkomplement implementiert
  - `char` hat 8 Bit
  - `short` hat 16 Bit
  - `int` hat 32 Bit
  - `long` hat 32 Bit auf x86 und 64 Bit auf x86-64

Frage 13

Angenommen x hat Typ **`short`**. Ist `x << 29` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x?

Erklärung

- Vor der Verschiebeoperation wird nach **`int`** umgewandelt
- Verschiebung um mehr als die Bitbreite ist also kein Problem

Frage 14

Angenommen x hat Typ **unsigned**. Ist `x << 31` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x?

Erklärung

- jeder Wert, dessen *promoted type* **unsigned** ist kann um nichtnegativen Wert verschoben werden
- solange die Bitbreite nicht erreicht wird

Frage 15

Angenommen x hat Typ **unsigned short**. Ist `x << 31` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x?

Erklärung

- **unsigned short** wird nach **int** umgewandelt
- eine 1 darf nicht in das Vorzeichenbit hineinverschoben werden
- Verschiebung um bis zu 15 wäre immer in Ordnung

Frage 16

Angenommen x hat Typ **int**. Ist `x + 1` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x?

Erklärung

- nicht definiert genau dann, wenn `INT_MAX`

Quelle des C-Quiz

`http://blog.regehr.org/archives/721`

Überblick

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 }
```

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 }
```

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 }
```

Subtraktion

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

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 }
```

Multiplikation

Was soll da schon schiefgehen...

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

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 }
```

Explizite Typumwandlung

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

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 }
```

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 }
```

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 }
```

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 }
```

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 }
```

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 }
```

Addition

Was soll da schon schiefgehen...

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

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 }
```

Subtraktion

Was soll da schon schiefgehen...

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

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 }
```

Division

Was soll da schon schiefgehen...

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

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 }
```

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 }
```

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 }
```

Modulo

Was soll da schon schiefgehen...

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

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 }
```

Negation

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Multiplikation

Was soll da schon schiefgehen...

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

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 }
```

Überblick

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!

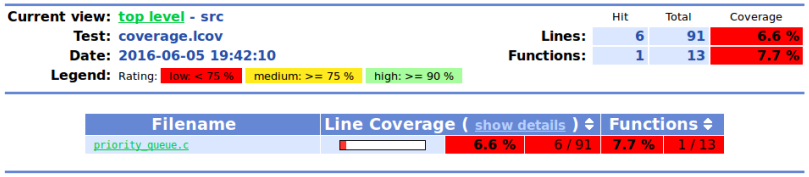
- Testfälle können nur die Anwesenheit von Fehlern zeigen
- Nicht deren Abwesenheit! (→ vgl. Verifikation)

⇝ 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 test`
- Automatische Testauswertung:
  - Anhand Rückgabewert (0 → OK, -1 → Fehler)
  - Notfalls auch Parsen von Ausgaben

Codeüberdeckung: gcov/lcov



		Hit	Total	Coverage
Current view:	top level - src			
Test:	coverage.lcov	Lines:	6	91	6.6 %
Date:	2016-06-05 19:42:10	Functions:	1	13	7.7 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %			

Filename	Line Coverage (show details)		Functions	
priority_queue.c	6.6 %	6 / 91	7.7 %	1 / 13

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

→ Tests solange erweitern, bis *vollständige Zeilenüberdeckung* 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`
- Aktivierung im CIP-Pool (im Terminal ausführen):
- *Im CIP-Pool erst* `addpackage clang` *im Terminal ausführen*
- 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 ↝ „statische Analyse"
- Eingabewerte als *symbolisch* angenommen → *Symbolische Ausführung*
- 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)
  - Falsch positive Befunde mögliche (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 -DCMAKE_BUILD_TYPE=Debug ..`

- 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

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]); }
```

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

  ⇝ Verpflichtung ggü. Benutzer

Iterator – 2. Versuch

- Iterator als Teil des Objekts
- Header:

```
typedef struct Elements Elements;
void El_reset_iterator(Elements *self);
void El_next(Elements *self);
bool El_isAtEnd(Elements *self);
int64_t El_iterator_value(Elements *self);
```

- Verwendung:

```
El_reset_iterator(dings);
while(!El_isAtEnd(dings)) {
  use(El_iterator_value(dings));
  El_next(dings);
}
```

Iterator – 2. Versuch

- Implementierung:

```
typedef struct Element { int64_t value; } Element;
struct Elements {
  Element elements[ELEMENTS_SIZE];
  Element *it;
};
void El_reset_iterator(Elements *self)
  { self->it = &self->elements }
void El_next(Elements *self)
  { self->it = self->it + 1; }
bool El_isAtEnd(Elements *self)
  { return self->it
      == &(self->elements[ELEMENTS_SIZE]); }
int64_t El_iterator_value(Elements *self)
  { return self->it->value; }
```

- *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:

```
typedef struct Elements Elements;
typedef struct El_Iterator El_Iterator;

El_Iterator *El_begin(Elements *self);
void El_Iterator_destroy(El_Iterator *self);
void El_Iterator_next(El_Iterator *self);
bool El_Iterator_isAtEnd(El_Iterator *self);
int64_t El_Iterator_value(El_Iterator *self);
```

- Verwendung:

```
El_Iterator *it;
for (it = El_begin(dings);
     not El_Iterator_isAtEnd(it);
     El_Iterator_next(it)) {
  use(El_Iterator_value(it))
}
El_Iterator_destroy(it);
```

Iterator – 3. Versuch

- Implementierung:

```
typedef struct Element { int64_t value; } Element;
struct Elements { Element elements[ELEMENTS_SIZE]; };
struct El_Iterator {
  Element *position;
  Element *end;
};

El_Iterator *El_begin(Elements *self) {
  El_Iterator *ret = malloc(sizeof(El_Iterator));
  if (ret == NULL) { return NULL; }
  ret->position = self->elements;
  ret->end = &self->elements[ELEMENTS_SIZE];
  return ret;
}

void El_Iterator_next(El_Iterator *self)
  { self->position += 1; }
bool El_Iterator_isAtEnd(El_Iterator *self) { ... }
int64_t El_Iterator_value(El_Iterator *self) { ... }
void El_Iterator_destroy(El_Iterator *self) { ... }
```

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.