Verlässliche Echtzeitsysteme - Übungen

Testen

Wintersemester 2024

Eva Dengler, Peter Wägemann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Systemsoftware)
https://sys.cs.fau.de

Überblick

1 Abfangen von Integer-Fehlern

2 Testen

- Vollständig Testen
- Peer-Review

3 Übungsaufgabe

Defensive Programmieren

- C bietet viele subtile Fehlermöglichkeiten
- Wie verhält man sich als Programmierer richtig?
- *Defensives Programmieren*

⇝ beispielhaft anhand von Ganzzahloperationen

Addition

Was soll da schon schiefgehen…

```
unsigned int func(unsigned int a, unsigned int b) {
  return a + b;
}
```

Vorbedingungstest

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

Nachbedingungstest

```
unsigned int func(unsigned int a, unsigned int b) {
  unsigned int ret = a + b;
  if (ret < a) { raise("wraparound"); }
  return ret;
}
```

Subtraktion

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Nachbedingungstest

```
unsigned int func(unsigned int a, unsigned int b) {
  unsigned int ret = a - b;
  if (ret > a) { raise("wraparound"); }
  return ret;
}
```

Multiplikation

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Explizite Typumwandlung

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Explizite Typumwandlung

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Explizite Typumwandlung

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Explizite Typumwandlung

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Addition

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Subtraktion

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Division

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Division

- Reicht das schon?

Was soll da schon schiefgehen...

```
signed long func(signed long a, signed long b) {
  if (b == 0) { raise("division by o"); }
  return a / b;
}
```

Vorbedingungstest

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

Modulo

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Negation

Was soll da schon schiefgehen...

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

Vorbedingungstest

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

Multiplikation

Was soll da schon schiefgehen...

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

Vorbedingungstest

```
#include <iso646.h>
#include <limits.h>
signed int func(signed int a, signed int b) {
  if (a == 0 or b == 0) { return 0; }
  if (a > 0 and b > 0 and a > INT_MAX / b) { raise("overflow"); }
  if (a > 0 and b < 0 and b < INT_MIN / a) { raise("overflow"); }
  if (a < 0 and b > 0 and a < INT_MIN / b) { raise("overflow"); }
  if (a < 0 and b < 0 and b < INT_MAX / a) { raise("overflow"); }
  return a * b;
}
```

Weitere Maßnahmen (I)

Konstruktiver Ausschluss

- Einhalten der Grenzbereiche durch Verifikation sichergestellt
- *beweisbare* Sicherheit

Garantiertes Ausnahmeverhalten

- auf Sprachebene
  - Rust: Operationen mit Überprüfung (bspw. `checked_add`)
  - D: Operationen mit Überprüfung: `checkedint`
  - Ada: Constraint_Error bei Überläufen
- durch die Hardware $\rightsquigarrow$ MIPS

Weitere Maßnahmen (II)

Softwareseitige Maßnahmen

- compilergestützt
  - gcc built-in functions
    - `__builtin_{add,sub,mul}_overflow`
  - spezielle Warnungen nutzen
    - `-W-sign-compare`, `-W-sign-conversion`
    - `-W-strict-overflow`, `-W-shift-overflow`
- mittels Bibliotheken (bspw. *Safe Numerics* von boost.org)

Keine Patentlösung

- abhängig von Anwendung und System
- muss beim *Systementwurf* bedacht werden
- zieht sich durch die *gesamte Systementwicklung*
- C macht es einem hier nicht einfach

Überblick

1 Abfangen von Integer-Fehlern

2 Testen

- Vollständig Testen
- Peer-Review

3 Übungsaufgabe

Spezifikation einer Warteschlange (1)

Die Warteschlange soll folgenden Anforderungen genügen: Es soll möglich sein, eine beliebige Anzahl von Warteschlangen zu erstellen, und die maximale Anzahl der Elemente einer solchen Warteschlange soll nur durch den verfügbaren freien Speicher und die Zahl der Prioritätsebenen begrenzt sein. Jede Prioritätsebene soll innerhalb einer Warteschlange maximal einem Element gleichzeitig zugewiesen sein. 0 entspricht hierbei der höchsten Priorität. Die Warteschlange soll nicht-vorzeichenbehaftete 64 Bit-Ganzzahlen verwalten.

Versucht der Benutzer ein Element mit einer Priorität einzufügen, die in der Warteschlange bereits vergeben ist, so ist dies ein Fehler. Dieser soll nicht durch eine Ausgabe auf dem Bildschirm, sondern durch einen Rückgabewert angezeigt werden. Geht während des Aufrufs einer Warteschlangenoperation der Hauptspeicher aus, so ist dies ein Fehler, über den der Benutzer über die API benachrichtigt wird. Die Reaktion auf den Fehler ist dem Aufrufer überlassen.

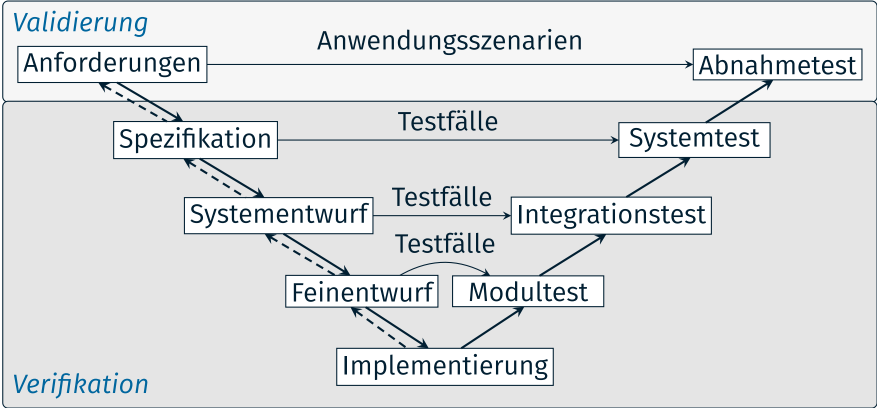
Spezifikation einer Warteschlange (2)

Die Warteschlange soll die Möglichkeit bieten, das höchstpriore Element zu entfernen und hierbei den Wert dieses Elements dem Benutzer zugänglich zu machen. Ferner soll es möglich sein, die Anzahl der Elemente, die sich derzeit in der Warteschlange befinden, zu erfragen.

Im Fehlerfall soll sich die Warteschlange allgemein gutmütig verhalten, d. h. ein Speicherzugriffsfehler oder nicht definiertes Verhalten sind beispielsweise nicht akzeptabel. Sofern das Verhalten der Warteschlange vom Laufzeitsystem abhängig ist, soll dies dokumentiert werden.

Nebenläufige Operationen auf der Warteschlange müssen **nicht** unterstützt werden. Es ist also keine Synchronisation erforderlich.

Einordnung in den Entwicklungsprozess



Testen

- Erste Grundregeln:
  - Testbarkeit von vornherein einplanen
    - ⇝ Feingranulare Testfälle
    - ⇝ *Separate Testfälle für jede einzelne Funktion!*
  - Teste Datentypen an ihren Wertebereichsgrenzen
  - Fehlerfälle explizit testen
  - *Minimale Testabdeckung* erreichbarer Codeüberdeckung
- Hilfsmittel:
  - Überdeckungsmetriken
  - Automatisierte Testinfrastruktur

Vorsicht!

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

⇝ Alle *Randfälle* erkennen und abdecken

Peer-Review und Quality-Assurance

1. ✉ Betreff: “Euer QA-Team wartet”
2. `make doxy`: Dokumentation lesen und implementieren
3. `make pack-review`

To: i4ezsmux+projectX-dev@i4.cs.fau.de
Subject: Mögliche Fehler

Hallo liebes Dev-Team,

bei uns schlagen die folgenden Tests mit euerer Implementierung fehl:
- unordered_insert:
  Der Test fügt in inverser Reihenfolge ein und prüft den folgenden Satz der Spezifikation “...”
  Ausgabe: ...
- ..
Sag

`make pack-review`, ALIEN_TEST, … im CIP ausführen
⇝ Referenzumgebung für alle

Überblick

1 Abfangen von Integer-Fehlern

2 Testen

- Vollständig Testen
- Peer-Review

3 Übungsaufgabe

Aufgabe 5 – Testen

- Verwendung von GNU/Linux (kein eCos mehr)
- Ziele
  1. Testfokusierter Softwareentwurf
  2. Testfallentwurf
     - Vollständige Pfadüberdeckung
     - Abdecken aller Randfälle
  3. Implementierung von Software und Testfällen
     - Getrennte Implementierung von Software und Testfällen
     - Möglichst durch verschiedene Übungsteilnehmer

     ⇝ Peer-Review
- 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

```
typedef struct Element { ... } Element;
Element elements[ELEMENTS_SIZE];
...
  for (size_t i = 0; i < ELEMENTS_SIZE; ++i)
    { use(elements[i]); }
```

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

  $\leadsto$ Verpflichtung gegenüber 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

$\Rightarrow$ Verwendung in der Übung