Seminar Effiziente C-Programmierung

Fehlertoleranz

Anna Fuchs
Betreuer: Michael Kuhn

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

23.01.2014

Outline

Begriffe

Fehler

Toleranz

Begriffe

Fehler

- Kritische und unkritische

Toleranz

Begriffe

Fehler

- Kritische und unkritische

Toleranz

Begriffe

Fehler

- Kritische und unkritische
  - Algorithmus / Logik
  - Software
    - Algorithmus falsch umgesetzt
    - Softwareintern
  - Hardware
    - CPU
    - Speicher
    - Netzwerk

Toleranz

Begriffe

Fehler

- Kritische und unkritische
  - Algorithmus / Logik
  - Software
    - Algorithmus falsch umgesetzt
    - Softwareintern
  - Hardware
    - CPU
    - Speicher
    - Netzwerk

Toleranz

Begriffe

Fehler

- Kritische und unkritische
  - Algorithmus / Logik
  - Software
    - Algorithmus falsch umgesetzt
    - Softwareintern
  - Hardware
    - CPU
    - Speicher
    - Netzwerk

Toleranz

- Eine der Möglichkeiten der Fehlerbehandlung
- Ignorieren oder reagieren

Begriffe

Fehler

- Kritische und unkritische
  - Algorithmus / Logik
  - Software
    - Algorithmus falsch umgesetzt
    - Softwareintern
  - Hardware
    - CPU
    - Speicher
    - Netzwerk

Toleranz

- Eine der Möglichkeiten der Fehlerbehandlung
- Ignorieren oder reagieren
- Programmabbruch verhindern
- Originalen Zustand verlustfrei wiederherstellen
  - Selten möglich oder teuer
  - Setzt oft Redundanz voraus

Begriffe

Fehler

- Kritische und unkritische
  - Algorithmus / Logik
  - Software
    - Algorithmus falsch umgesetzt
    - Softwareintern
  - Hardware
    - CPU
    - Speicher
    - Netzwerk

Toleranz

- Eine der Möglichkeiten der Fehlerbehandlung
- Ignorieren oder reagieren
- Programmabbruch verhindern
- Originalen Zustand verlustfrei wiederherstellen
  - Selten möglich oder teuer
  - Setzt oft Redundanz voraus
- Sich an den originalen Zustand annähern
  - Verändert das Endergebnis

Motivation

- Fakten

Motivation

- Fakten
- Steigerung von
  - Verbreitung der Computersysteme (Ubiquitous computing)
  - Leistungsfähigkeit
  - Skalierbarkeit
  - Komplexität

Motivation

- Fakten
- Steigerung von
  - Verbreitung der Computersysteme (Ubiquitous computing)
  - Leistungsfähigkeit
  - Skalierbarkeit
  - Komplexität
- Konsequenzen
  - Anforderungen an Verfügbarkeit, Sicherheit, Zuverlässigkeit und Leistung steigen ebenfalls
  - Ausfallwahrscheinlichkeit $\propto$ Anzahl Einheiten
  - Wartbarkeit immer schwieriger
  - Effizienz der Software muss mithalten

Motivation

- Fakten
- Steigerung von
  - Verbreitung der Computersysteme (Ubiquitous computing)
  - Leistungsfähigkeit
  - Skalierbarkeit
  - Komplexität
- Konsequenzen
  - Anforderungen an Verfügbarkeit, Sicherheit, Zuverlässigkeit und Leistung steigen ebenfalls
  - Ausfallwahrscheinlichkeit $\propto$ Anzahl Einheiten
  - Wartbarkeit immer schwieriger
  - Effizienz der Software muss mithalten
- Fehleranalyse und -behandlung unvermeidbar

Einsatzgebiet

- Echtzeitsysteme
  - Sicherheitskritisch
  - Dürfen nicht abstürzen

Einsatzgebiet

- Echtzeitsysteme
  - Sicherheitskritisch
  - Dürfen nicht abstürzen
- Interaktiv
  - Reaktionszeit während / nach dem Fehler unzumutbar

Einsatzgebiet

- Echtzeitsysteme
  - Sicherheitskritisch
  - Dürfen nicht abstürzen
- Interaktiv
  - Reaktionszeit während / nach dem Fehler unzumutbar
- Massiv Parallele Systeme
  - Neustart oder Redundanz zu teuer

Einsatzgebiet

- Echtzeitsysteme
  - Sicherheitskritisch
  - Dürfen nicht abstürzen
- Interaktiv
  - Reaktionszeit während / nach dem Fehler unzumutbar
- Massiv Parallele Systeme
  - Neustart oder Redundanz zu teuer

- Aufwand der Fehlertoleranz $<$ Kosten für andere Verfahren

Strategie

crash

Strategie

- Prävention
  - Fehler verhindern
  - Code-Ausführung in sicherer Umgebung

crash

Strategie

- Prävention
  - Fehler verhindern
  - Code-Ausführung in sicherer Umgebung

crash

- Reaktion
  - Fehler abfangen, umleiten, ausgeben
  - Fehler analysieren, beheben
  - Daten retten / wiederherstellen / sichern

Strategie

- Prävention
  - Fehler verhindern
  - Code-Ausführung in sicherer Umgebung

crash

- Reaktion
  - Fehler abfangen, umleiten, ausgeben
  - Fehler analysieren, beheben
  - Daten retten / wiederherstellen / sichern
- Weiterlaufen?

Statische Vorsorge

Versuchen, den Fehler zu verhindern - vorbeugen

- Statisch überprüfen
  - Vor der Laufzeit
  - Editor, Compiler
  - Reviewer
  - Static Code Analyzer

Statische Vorsorge

Versuchen, den Fehler zu verhindern - vorbeugen

- Statisch überprüfen
  - Vor der Laufzeit
  - Editor, Compiler
  - Reviewer
  - Static Code Analyzer
    - Clang Static Analyzer

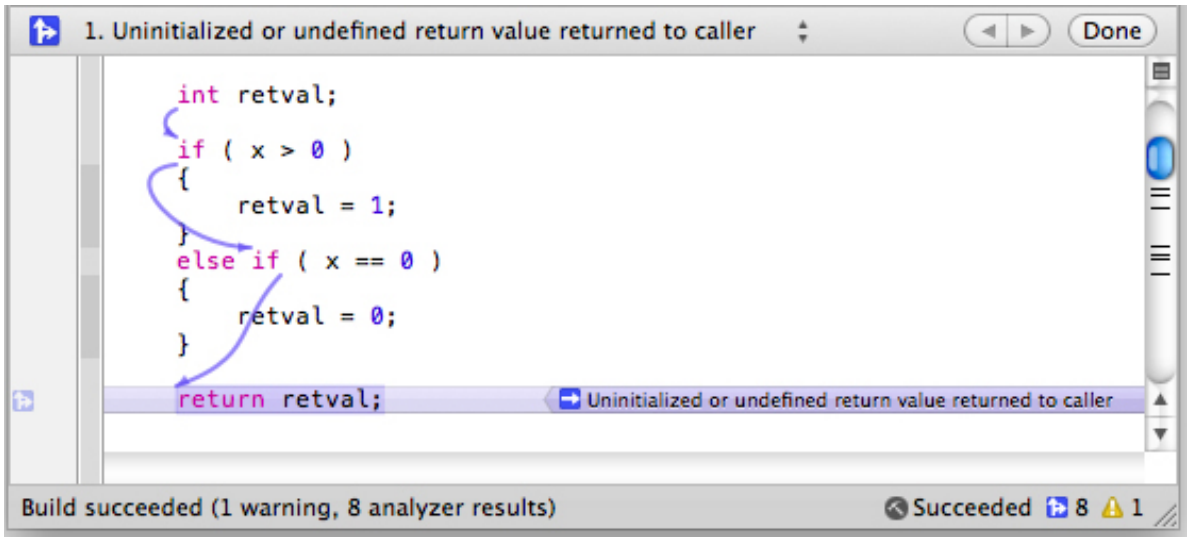


Figure : http://clang-analyzer.llvm.org/images/analyzer_xcode.png

Dynamische Sorge

- Dynamisch überprüfen
  - Typumwandlungen
  - Dynamischer Speicher
    - Hängende, doppelte, ungültige Zeiger
    - Stack-Überlauf
    - Heap-Überlauf
    - Fehlende, doppelte, ungültige Speicherfreigabe
  - (Fehlende Initialisierung)
  - Datenstrukturen
    - Threadsafety
  - Semantische Inhalte
  - Undefiniertes Verhalten?

Werkzeuge

- `assert`
  - Statisch - Überprüfung des zur Compile-Zeit Bekannten
  - Dynamisch - erzwingt ggf. einen kontrollierten Programmabbruch

Werkzeuge

- `assert`
  - Statisch - Überprüfung des zur Compile-Zeit Bekannten
  - Dynamisch - erzwingt ggf. einen kontrollierten Programmabbruch
- `if`
  - Dynamische Abfragen

Werkzeuge

- `assert`
  - Statisch - Überprüfung des zur Compile-Zeit Bekannten
  - Dynamisch - erzwingt ggf. einen kontrollierten Programmabbruch
- `if`
  - Dynamische Abfragen
- `goto`
  - Kontextsprünge innerhalb einer Funktion
  - Mächtig und gefährlich
  - Vermeidbar

Werkzeuge

- `assert`
  - Statisch - Überprüfung des zur Compile-Zeit Bekannten
  - Dynamisch - erzwingt ggf. einen kontrollierten Programmabbruch
- `if`
  - Dynamische Abfragen
- `goto`
  - Kontextsprünge innerhalb einer Funktion
  - Mächtig und gefährlich
  - Vermeidbar
- `try-catch-block`
  - Gibt es nicht in C
  - Simulieren mit `setjmp` und `longjmp`

goto

if{if{if}}

```
1   int zeug(int N)
2   {
3     int* p1 = get_space(N);
4     if(check(p1, N) == 0)
5     {
6
7
8
9
10
11
12
13
14
15
16
17     }
18    cleanup(p1);
19
20    return ret;
21  }
22
23
24
25
```

goto

if{if{if}}

```
1   int zeug(int N)
2   {
3     int* p1 = get_space(N);
4     if(check(p1, N) == 0)
5     {
6       int* p2 = get_space(N);
7       if(check(p2, N) == 0)
8       {
9
10
11
12
13
14
15      }
16      cleanup(p2);
17    }
18    cleanup(p1);
19
20    return ret;
21  }
22
23
24
25
```

goto

if{if{if}}

```
1   int zeug(int N)
2   {
3     int* p1 = get_space(N);
4     if(check(p1, N) == 0)
5     {
6       int* p2 = get_space(N);
7       if(check(p2, N) == 0)
8       {
9         int* p3 = get_space(N);
10        if(check(p3, N) == 0)
11        {
12          int ret = do_it(p1, p2, p3);
13        }
14        cleanup(p3);
15      }
16      cleanup(p2);
17    }
18    cleanup(p1);
19
20    return ret;
21  }
22
23
24
25
```

goto

if{if{if}}

```
1   int zeug(int N)
2   {
3     int* p1 = get_space(N);
4     if(check(p1, N) == 0)
5     {
6       int* p2 = get_space(N);
7       if(check(p2, N) == 0)
8       {
9         int* p3 = get_space(N);
10        if(check(p3, N) == 0)
11        {
12          int ret = do_it(p1, p2, p3);
13        }
14        cleanup(p3);
15      }
16      cleanup(p2);
17    }
18    cleanup(p1);
19
20    return ret;
21  }
22
23
24
25
```

goto

```
1   int zeug(int N)
2   {
3     int* p1 = get_space(N);
4     if(check(p1, N) != 0)
5       goto error1;
6
7     int* p2 = get_space(N);
8     if(check(p2, N) != 0)
9       goto error2;
10
11    int* p3 = get_space(N);
12    if(check(p3, N) != 0)
13      goto error3;
14
15    int ret = do_it(p1, p2, p3);
16
17    error3:
18      cleanup(p3);
19    error2:
20      cleanup(p2);
21    error1:
22      cleanup(p1);
23
24    return ret;
25  }
```

goto - 2

- Laufzeit - gleich

goto - 2

- Laufzeit - gleich
- Sprünge im Programm zurück - gefährlich
  - `while`-Schleife als Alternative

goto - 2

- Laufzeit - gleich
- Sprünge im Programm zurück - gefährlich
  - `while`-Schleife als Alternative
- Sprünge in lokale Blöcke gefährlich
  - Verwendung von lokalen nicht deklarierten/initialisierten Variablen - systemabhängig oder undefiniert

```
1       ...
2       goto step;
3
4       for (int i=0; i<10; i++)
5       {
6     step: printf("i = %d \n", i);
7       }
```

setjmp/longjmp

- C89, C99, and POSIX.1-2001
- Kontextsprünge (aus der Funktion) mit `setjmp` und `longjmp`

```
1  int setjmp(jmp_buf env);
2  void longjmp(jmp_buf env, int val);
```

setjmp/longjmp

- C89, C99, and POSIX.1-2001
- Kontextsprünge (aus der Funktion) mit `setjmp` und `longjmp`

```
1    int setjmp(jmp_buf env);
2    void longjmp(jmp_buf env, int val);
```

- Rückgabewert von `setjmp`
  - 0 beim direkten Aufruf
  - `val` beim Aufruf nach `longjmp`

setjmp/longjmp

- C89, C99, and POSIX.1-2001
- Kontextsprünge (aus der Funktion) mit `setjmp` und `longjmp`

```
1  int setjmp(jmp_buf env);
2  void longjmp(jmp_buf env, int val);
```

- Rückgabewert von `setjmp`
  - 0 beim direkten Aufruf
  - `val` beim Aufruf nach `longjmp`
- `jmp_buf env`
  - manpage: stack context/environment
  - *calling environment* (und *signal mask*)

setjmp/longjmp

- C89, C99, and POSIX.1-2001
- Kontextsprünge (aus der Funktion) mit `setjmp` und `longjmp`

```
1   int setjmp(jmp_buf env);
2   void longjmp(jmp_buf env, int val);
```

- Rückgabewert von `setjmp`
  - 0 beim direkten Aufruf
  - `val` beim Aufruf nach `longjmp`
- `jmp_buf env`
  - manpage: stack context/environment
  - *calling environment* (und *signal mask*)
- calling environment
  - Inhalt lokaler Variablen
  - **s***tack***p***ointer*, **f***rame***p***ointer*, **p***rogram***c***ounter*

setjmp/longjmp

- C89, C99, and POSIX.1-2001
- Kontextsprünge (aus der Funktion) mit `setjmp` und `longjmp`

```
1   int setjmp(jmp_buf env);
2   void longjmp(jmp_buf env, int val);
```

- Rückgabewert von `setjmp`
  - 0 beim direkten Aufruf
  - `val` beim Aufruf nach `longjmp`
- `jmp_buf env`
  - manpage: stack context/environment
  - *calling environment* (und *signal mask*)
- calling environment
  - Inhalt lokaler Variablen
  - **s***tack***p***ointer*, **f***rame***p***ointer*, **p***rogram***c***ounter*
- Beim `longjump`-Aufruf wird `env` gelesen

Beipsiel setjmp/longjmp

```
1   jmp_buf jbuf;
2   volatile int glob;
3
4   void zeug()
5   {
6     printf("longjmp \n");
7     glob = 200;
8     longjmp(jbuf, 1);
9   }
10
11  int main()
12  {
13    int loc = 10;
14    glob = 20;
15
16    int rc = setjmp(jbuf);
17    printf("setjmp \n");
18
19    if (rc == 0)
20    {
21      printf("rc = 0 \n");
22      loc = 100;
23      zeug();
24      printf("Das wird nie ausgegeben. \n");
25    }
26
27    printf("loc = %d, glob = %d \n", loc, glob);
28    return 0;
29  }
```

Beipsiel setjmp/longjmp

```
1   jmp_buf jbuf;
2   volatile int glob;
3
4   void zeug()
5   {
6     printf("longjmp \n");
7     glob = 200;
8     longjmp(jbuf, 1);
9   }
10
11  int main()
12  {
13    int loc = 10;
14    glob = 20;
15
16    int rc = setjmp(jbuf);
17    printf("setjmp \n");
18
19    if (rc == 0)
20    {
21      printf("rc = 0 \n");
22      loc = 100;
23      zeug();
24      printf("Das wird nie ausgegeben. \n");
25    }
26
27    printf("loc = %d, glob = %d \n", loc, glob);
28    return 0;
29  }
```

```
1       setjmp
```

Beipsiel setjmp/longjmp

```
1   jmp_buf jbuf;
2   volatile int glob;
3
4   void zeug()
5   {
6     printf("longjmp \n");
7     glob = 200;
8     longjmp(jbuf, 1);
9   }
10
11  int main()
12  {
13    int loc = 10;
14    glob = 20;
15
16    int rc = setjmp(jbuf);
17    printf("setjmp \n");
18
19    if (rc == 0)
20    {
21      printf("rc = 0 \n");
22      loc = 100;
23      zeug();
24      printf("Das wird nie ausgegeben. \n");
25    }
26
27    printf("loc = %d, glob = %d \n", loc, glob);
28    return 0;
29  }
```

```
1   setjmp
2   rc = 0
```

Beipsiel setjmp/longjmp

```
1   jmp_buf jbuf;
2   volatile int glob;
3
4   void zeug()
5   {
6     printf("longjmp \n");
7     glob = 200;
8     longjmp(jbuf, 1);
9   }
10
11  int main()
12  {
13    int loc = 10;
14    glob = 20;
15
16    int rc = setjmp(jbuf);
17    printf("setjmp \n");
18
19    if (rc == 0)
20    {
21      printf("rc = 0 \n");
22      loc = 100;
23      zeug();
24      printf("Das wird nie ausgegeben. \n");
25    }
26
27    printf("loc = %d, glob = %d \n", loc, glob);
28    return 0;
29  }
```

```
1   setjmp
2   rc = 0
3   longjmp
```

Beipsiel setjmp/longjmp

```
1   jmp_buf jbuf;
2   volatile int glob;
3
4   void zeug()
5   {
6     printf("longjmp \n");
7     glob = 200;
8     longjmp(jbuf, 1);
9   }
10
11  int main()
12  {
13    int loc = 10;
14    glob = 20;
15
16    int rc = setjmp(jbuf);
17    printf("setjmp \n");
18
19    if (rc == 0)
20    {
21      printf("rc = 0 \n");
22      loc = 100;
23      zeug();
24      printf("Das wird nie ausgegeben. \n");
25    }
26
27    printf("loc = %d, glob = %d \n", loc, glob);
28    return 0;
29  }
```

```
1   setjmp
2   rc = 0
3   longjmp
4   setjmp
```

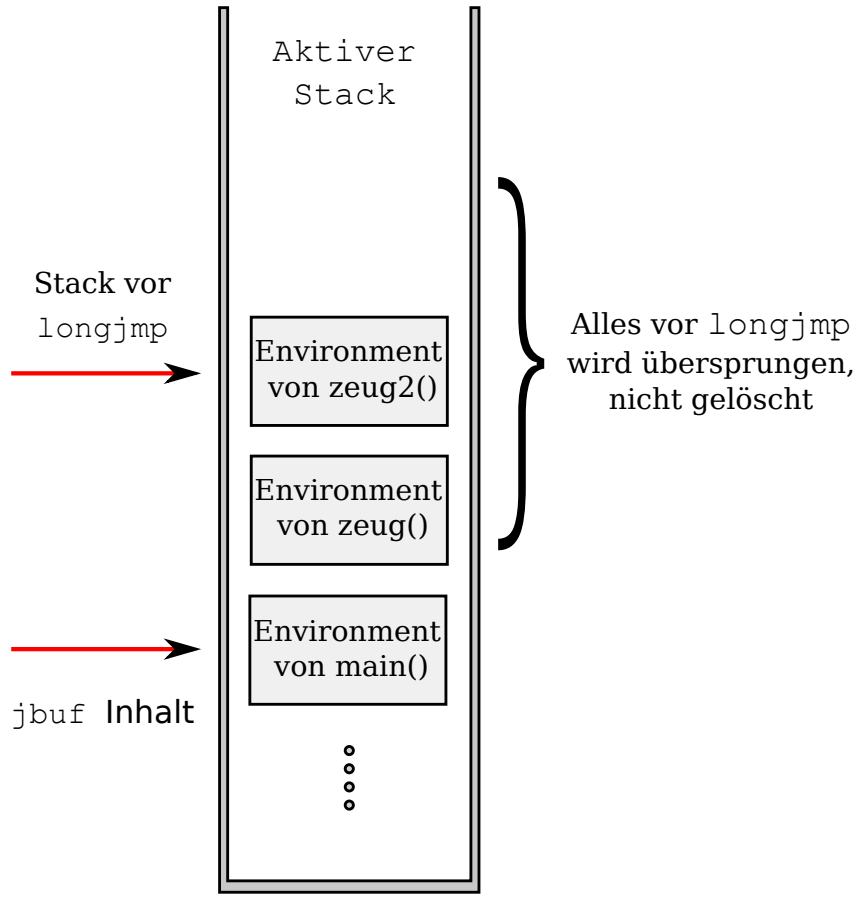
Beipsiel setjmp/longjmp

```
1   jmp_buf jbuf;
2   volatile int glob;
3
4   void zeug()
5   {
6     printf("longjmp \n");
7     glob = 200;
8     longjmp(jbuf, 1);
9   }
10
11  int main()
12  {
13    int loc = 10;
14    glob = 20;
15
16    int rc = setjmp(jbuf);
17    printf("setjmp \n");
18
19    if (rc == 0)
20    {
21      printf("rc = 0 \n");
22      loc = 100;
23      zeug();
24      printf("Das wird nie ausgegeben. \n");
25    }
26
27    printf("loc = %d, glob = %d \n", loc, glob);
28    return 0;
29  }
```

```
1   setjmp
2   rc = 0
3   longjmp
4   setjmp
5   loc = 10, glob = 200
```

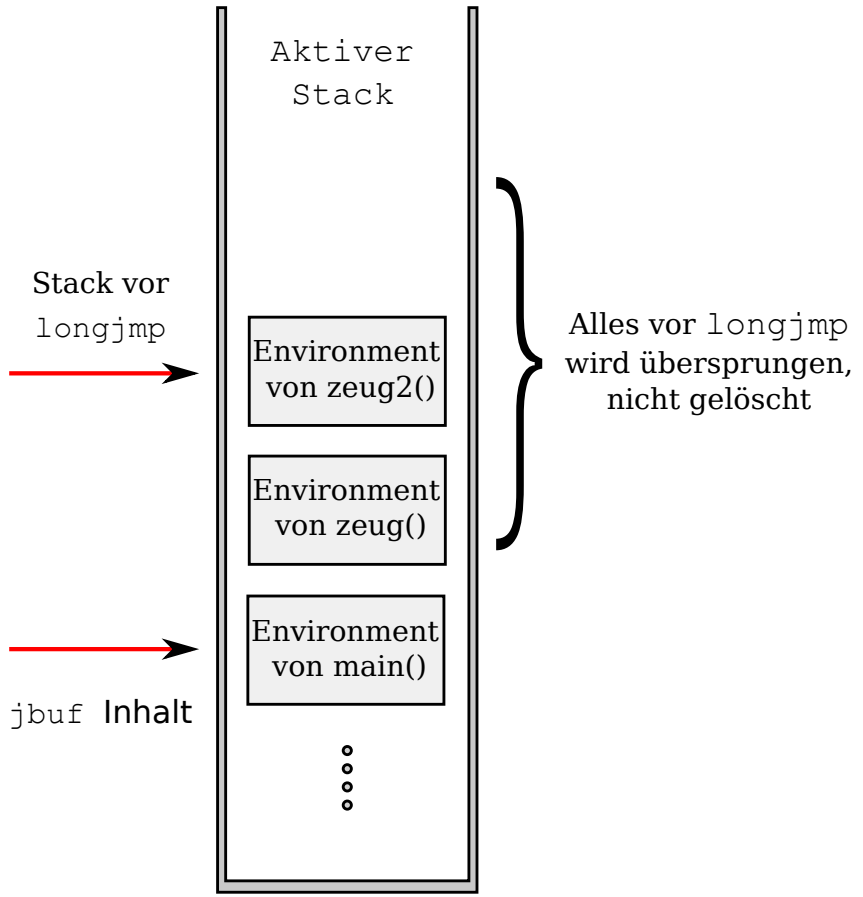
Stack

- Kein Stack-unwinding



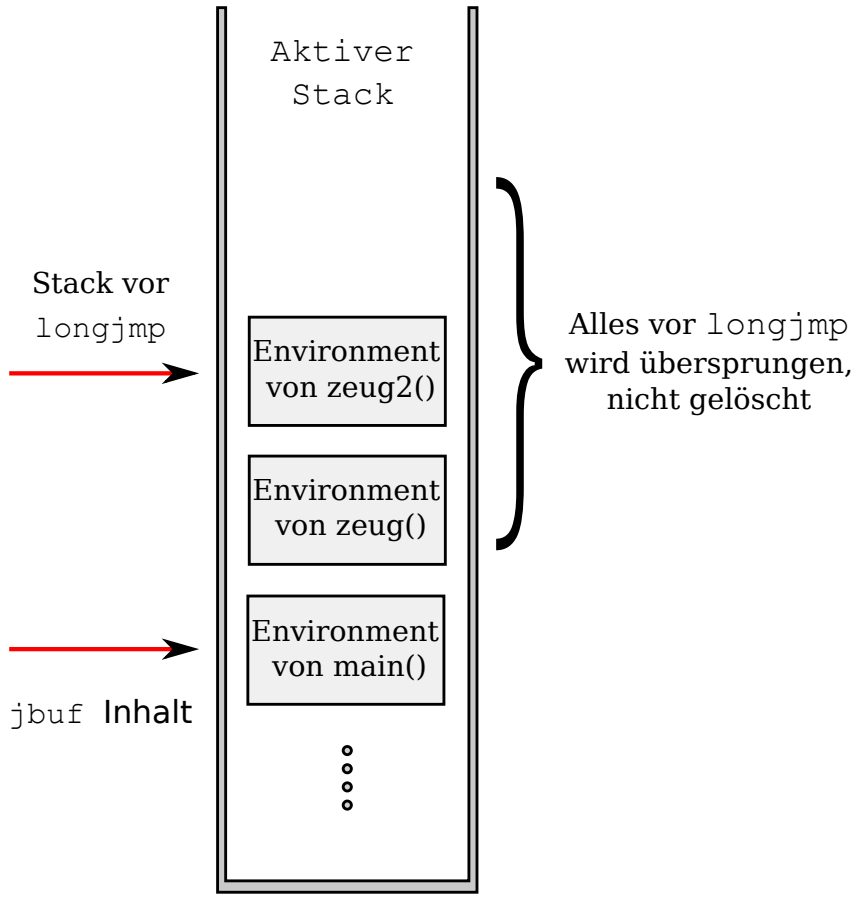
Stack

- Kein Stack-unwinding
- Sprünge in ”zukünftige” Funktionen kritisch
  - Parameterübergabe?



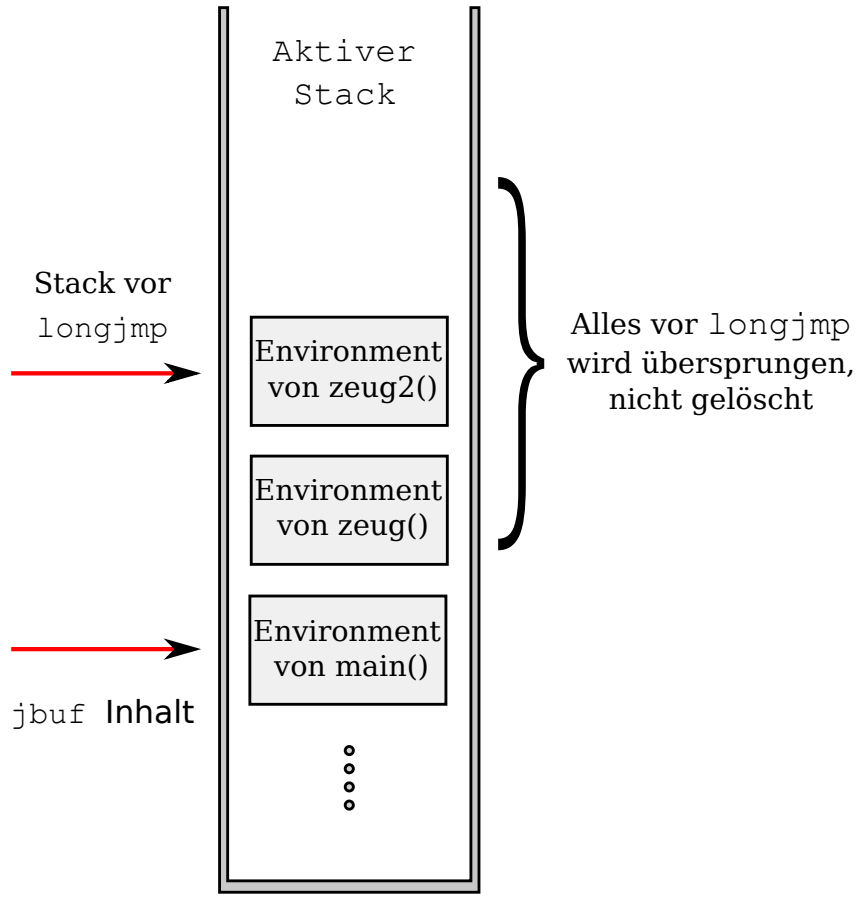
Stack

- Kein Stack-unwinding
- Sprünge in ”zukünftige” Funktionen kritisch
  - Parameterübergabe?
- Sprünge in ”vergangene” Funktionen kritisch
  - Stackinhalt bereits gelöscht



Stack

- Kein Stack-unwinding
- Sprünge in "zukünftige" Funktionen kritisch
  - Parameterübergabe?
- Sprünge in "vergangene" Funktionen kritisch
  - Stackinhalt bereits gelöscht
- `try-catch` simulieren
  - Kombination aus Makros, if-s, Signal-handler, `setjmp/longjmp`



- Prävention kann nicht alle Probleme lösen
- Absolute Prävention nicht möglich
  - Typabfragen nicht möglich
  - Stack-, Heapgrenzen nicht möglich
  - Alle Grenzwerte abfragen - teuer
  - Hardware

- Prävention kann nicht alle Probleme lösen
- Absolute Prävention nicht möglich
  - Typabfragen nicht möglich
  - Stack-, Heapgrenzen nicht möglich
  - Alle Grenzwerte abfragen - teuer
  - Hardware

Murphys Gesetz

Whatever can go wrong will go wrong.

- Prävention kann nicht alle Probleme lösen
- Absolute Prävention nicht möglich
  - Typabfragen nicht möglich
  - Stack-, Heapgrenzen nicht möglich
  - Alle Grenzwerte abfragen - teuer
  - Hardware

Murphys Gesetz

Whatever can go wrong will go wrong.

- Man muss mit einem Fehler rechnen
- Im besten Fall reagieren

Effizienz

Fehlertoleranz als Abhilfe? Beispiel

- Sensoren mit permanentem Input
- Inhaltliche Abfragen auf Ungültigkeit teuer
- Verzögern interaktive Reaktion

http://www.tripletsandus.com/disney/wall-e.htm

Effizienz

Fehlertoleranz als Abhilfe? Beispiel

- Sensoren mit permanentem Input
- Inhaltliche Abfragen auf Ungültigkeit teuer
- Verzögern interaktive Reaktion
- Unwahrscheinliche Fehler passieren lassen
- Reagieren und reparieren

http://www.tripletsandus.com/disney/wall-e.htm
http://www.insektenfallen.com/fliege.gif

Kosten

Fiktive Beispielrechnung

- n-viele Abfragen je $O(m)$ Zeit

Kosten

Fiktive Beispielrechnung

- n-viele Abfragen je $O(m)$ Zeit
- Fehlerwahrscheinlichkeit P

Kosten

Fiktive Beispielrechnung

- n-viele Abfragen je $O(m)$ Zeit
- Fehlerwahrscheinlichkeit P
- Fehlerbehandlung A mit $O(x)$ mit Prävention
- Fehlerbehandlung B mit $O(y)$ ohne Prävention
- $A << B$

Kosten

Fiktive Beispielrechnung

- n-viele Abfragen je $O(m)$ Zeit
- Fehlerwahrscheinlichkeit P
- Fehlerbehandlung A mit $O(x)$ mit Prävention
- Fehlerbehandlung B mit $O(y)$ ohne Prävention
- $A << B$
- $n \cdot m + P \cdot A > P \cdot B$?

Kosten

Fiktive Beispielrechnung

- n-viele Abfragen je $O(m)$ Zeit
- Fehlerwahrscheinlichkeit P
- Fehlerbehandlung A mit $O(x)$ mit Prävention
- Fehlerbehandlung B mit $O(y)$ ohne Prävention
- $A << B$
- $n \cdot m + P \cdot A > P \cdot B$?

- P abschätzen?

Keep going

- Wer bricht das Programm ab? $\rightarrow$ Betriebssystem

Keep going

- Wer bricht das Programm ab? → Betriebssystem
- Was ist die Ursache des Absturzes? → **Error**

Keep going

- Wer bricht das Programm ab? → Betriebssystem
- Was ist die Ursache des Absturzes? → **Error**
- Einige Fehler liefern einen Fehlerwert
  - Im Allgemeinen Aufrufe, die in einer Bibliothek definiert sind
  - Der Wert kann abgefangen, umgeleitet, ignoriert etc. werden

Keep going

- Wer bricht das Programm ab? → Betriebssystem
- Was ist die Ursache des Absturzes? → **Error**
- Einige Fehler liefern einen Fehlerwert
  - Im Allgemeinen Aufrufe, die in einer Bibliothek definiert sind
  - Der Wert kann abgefangen, umgeleitet, ignoriert etc. werden
- Andere generieren ein *Signal* auf der BS-Ebene (Linux)
  - Im Allgemeinen die Fehler, die überall passieren können (unabhängig von einer bestimmten Definition oder Bibliothek)
  - div-by-zero, ungültige Speicherzugriffe, I/O, Ctrl+C

Keep going

- Wer bricht das Programm ab? → Betriebssystem
- Was ist die Ursache des Absturzes? → **Error**
- Einige Fehler liefern einen Fehlerwert
  - Im Allgemeinen Aufrufe, die in einer Bibliothek definiert sind
  - Der Wert kann abgefangen, umgeleitet, ignoriert etc. werden
- Andere generieren ein *Signal* auf der BS-Ebene (Linux)
  - Im Allgemeinen die Fehler, die überall passieren können (unabhängig von einer bestimmten Definition oder Bibliothek)
  - div-by-zero, ungültige Speicherzugriffe, I/O, Ctrl+C
- Signale

fatal	`SIGKILL, SIGSTOP`
kritisch	`SIGFPE, SIGILL, SIGINT, SIGSEGV ...`
unkritisch	`SIGCHLD`

Signale

- Signal blockieren
  - Signal-Handler
  - `signal(SIGSEGV, my_handler);`*
  - Signal wird innerhalb des Funktionsblocks blockiert
  - Beim "normalen" Verlassen des Blocks - Verhalten *undefiniert*

Signale

- Signal blockieren
  - Signal-Handler
  - `signal(SIGSEGV, my_handler);`*
  - Signal wird innerhalb des Funktionsblocks blockiert
  - Beim "normalen" Verlassen des Blocks - Verhalten *undefiniert*
- `SIGTERM` ermöglicht "anständiges" Terminieren
  - Dateien schließen
  - Temporäre Dateien löschen
  - Kindprozesse terminieren

Signale

- Signal blockieren
  - Signal-Handler
  - `signal(SIGSEGV, my_handler);`*
  - Signal wird innerhalb des Funktionsblocks blockiert
  - Beim "normalen" Verlassen des Blocks - Verhalten *undefiniert*
- `SIGTERM` ermöglicht "anständiges" Terminieren
  - Dateien schließen
  - Temporäre Dateien löschen
  - Kindprozesse terminieren
- Division durch 0 **kann** zu einem `SIGFPE` führen
  - Könnte abgefangen und blockiert werden

Signale

- Signal blockieren
  - Signal-Handler
  - `signal(SIGSEGV, my_handler);`*
  - Signal wird innerhalb des Funktionsblocks blockiert
  - Beim "normalen" Verlassen des Blocks - Verhalten *undefiniert*
- `SIGTERM` ermöglicht "anständiges" Terminieren
  - Dateien schließen
  - Temporäre Dateien löschen
  - Kindprozesse terminieren
- Division durch 0 **kann** zu einem `SIGFPE` führen
  - Könnte abgefangen und blockiert werden
- Signal ignorieren
  - `signal (SIGCHLD, SIG_IGN);`*
  - Bei "kritischen" Signalen - Verhalten *undefiniert*

Beispiel

```
1
2     volatile int stuff[5] = {1, 2, 3, 0, 5};          // random data
3     volatile int i, c;                                // global vars
4     volatile long long n;
5
6
7
8
9
10
11
12    int main()
13    {
14      int a=0;                                        // local var
15      c=0; n=0; i=0;                                  // init global vars
16
17
18
19
20
21
22
23
24
25
26
27
28
29      return 0;
30    }
```

Beispiel

```
1
2     volatile int stuff[5] = {1, 2, 3, 0, 5};          // random data
3     volatile int i, c;                                // global vars
4     volatile long long n;
5
6
7
8
9
10
11
12    int main()
13    {
14      int a=0;                                        // local var
15      c=0; n=0; i=0;                                  // init global vars
16
17
18
19      for(n=n; n<500000000; n++)                      // outer loop
20      {
21        for(i=i; i<5; i++)
22        {
23          a = n / stuff[i];                           // global data request
24          c = c+a+i;                                  // dummy compute
25        }
26        i=0;                                          // reset inner loop index
27      }
28      printf("c=%d\n",c);
29      return 0;
30    }
```

Beispiel

```
1
2     volatile int stuff[5] = {1, 2, 3, 0, 5};          // random data
3     volatile int i, c;                                // global vars
4     volatile long long n;
5
6     void handler(int sig)                             // my handler
7     {
8       stuff[i] = 1;                                   // repair stuff occured signal
9
10    }
11
12    int main()
13    {
14      int a=0;                                        // local var
15      c=0; n=0; i=0;                                  // init global vars
16
17      signal(SIGFPE, handler);                        // handler for SIGFPE
18
19      for(n=n; n<500000000; n++)                      // outer loop
20      {
21        for(i=i; i<5; i++)
22        {
23          a = n / stuff[i];                           // global data request
24          c = c+a+i;                                  // dummy compute
25        }
26        i=0;                                          // reset inner loop index
27      }
28      printf("c=%d\n",c);
29      return 0;
30    }
```

Beispiel

```
1   jmp_buf jbuf;                                  // environment buffer
2   volatile int stuff[5] = {1, 2, 3, 0, 5};       // random data
3   volatile int i, c;                             // global vars
4   volatile long long n;
5
6   void handler(int sig)                          // my handler
7   {
8     stuff[i] = 1;                                // repair stuff occured signal
9     longjmp(jbuf, 1);                            // go back to setjmp
10  }
11
12  int main()
13  {
14    int a=0;                                     // local var
15    c=0; n=0; i=0;                               // init global vars
16    setjmp(jbuf);                                // save env BEFORE handler
17    signal(SIGFPE, handler);                     // handler for SIGFPE
18
19    for(n=n; n<500000000; n++)                   // outer loop
20    {
21      for(i=i; i<5; i++)
22      {
23        a = n / stuff[i];                        // global data request
24        c = c+a+i;                               // dummy compute
25      }
26      i=0;                                       // reset inner loop index
27    }
28    printf("c=%d\n",c);
29    return 0;
30  }
```

Leistungsmessung

500000000 Iterationen, O3, O0 west2

Testfall	Sekunden
`setjmp` 0-3x errors	33,8
`if`	38,4
`ifififif`	41,3
`if #unlikely`	38,5
`if #likely`	39,7
ohne `if` ohne error	34,3

Leistungsmessung

500000000 Iterationen, O3, O0 west2

Testfall	Sekunden	Assembler
`setjmp` 0-3x errors	33,8	`cmpq $2, %rax`
`if`	38,4	
`ifififif`	41,3	
`if #unlikely`	38,5	
`if #likely`	39,7	
ohne `if` ohne error	34,3	`cmpq $499999999, %rax`

Kleine Probleme

- * `signal` nicht benutzen - `sigaction`

Kleine Probleme

- * `signal` nicht benutzen - `sigaction`
- `setjmp/longjmp` hier nicht benutzen - `sigsetjmp/siglongjmp` (POSIX.1-2001)
- `longjmp` async-signal-unsafe
  - Signal vor / innerhalb des Handlers
  - Speicherinkonsistenz

Kleine Probleme

- * `signal` nicht benutzen - `sigaction`
- `setjmp/longjmp` hier nicht benutzen - `sigsetjmp/siglongjmp` (POSIX.1-2001)
- `longjmp` async-signal-unsafe
  - Signal vor / innerhalb des Handlers
  - Speicherinkonsistenz
- O3 optimiert globale Variablen - `volatile`

Kleine Probleme

- * `signal` nicht benutzen - `sigaction`
- `setjmp/longjmp` hier nicht benutzen - `sigsetjmp/siglongjmp` (POSIX.1-2001)
- `longjmp` async-signal-unsafe
  - Signal vor / innerhalb des Handlers
  - Speicherinkonsistenz
- O3 optimiert globale Variablen - `volatile`
- `warning: variable c might be clobbered by longjmp or vfork [-Wclobbered]`

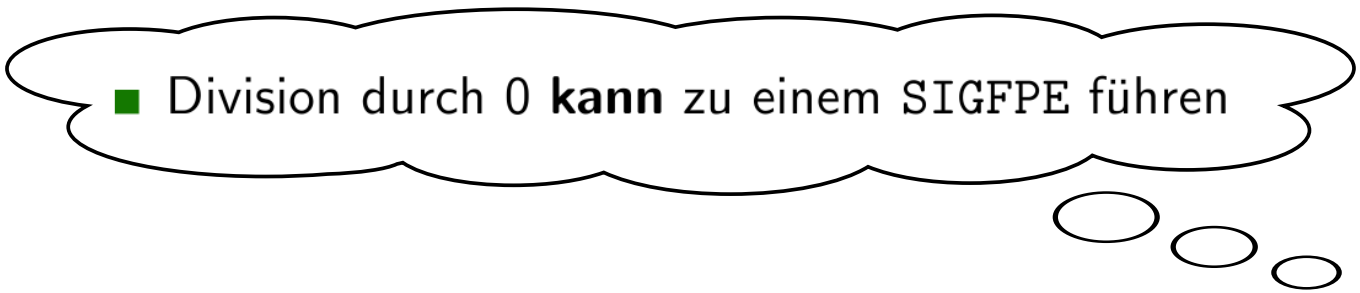
Kleine Probleme

- * `signal` nicht benutzen - `sigaction`
- `setjmp/longjmp` hier nicht benutzen - `sigsetjmp/siglongjmp` (POSIX.1-2001)
- `longjmp` async-signal-unsafe
  - Signal vor / innerhalb des Handlers
  - Speicherinkonsistenz
- O3 optimiert globale Variablen - `volatile`
- `warning: variable c might be clobbered by longjmp or vfork [-Wclobbered]`
- Kein Stack-unwinding

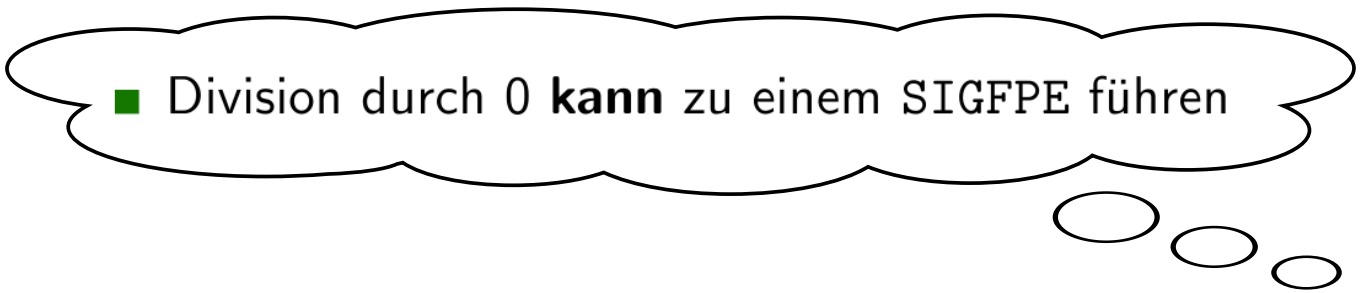
Kleine Probleme

- * `signal` nicht benutzen - `sigaction`
- `setjmp/longjmp` hier nicht benutzen - `sigsetjmp/siglongjmp` (POSIX.1-2001)
- `longjmp` async-signal-unsafe
  - Signal vor / innerhalb des Handlers
  - Speicherinkonsistenz
- O3 optimiert globale Variablen - `volatile`
- `warning: variable c might be clobbered by longjmp or vfork [-Wclobbered]`
- Kein Stack-unwinding
- Eintreten in den Handler "sofort"

Große Probleme

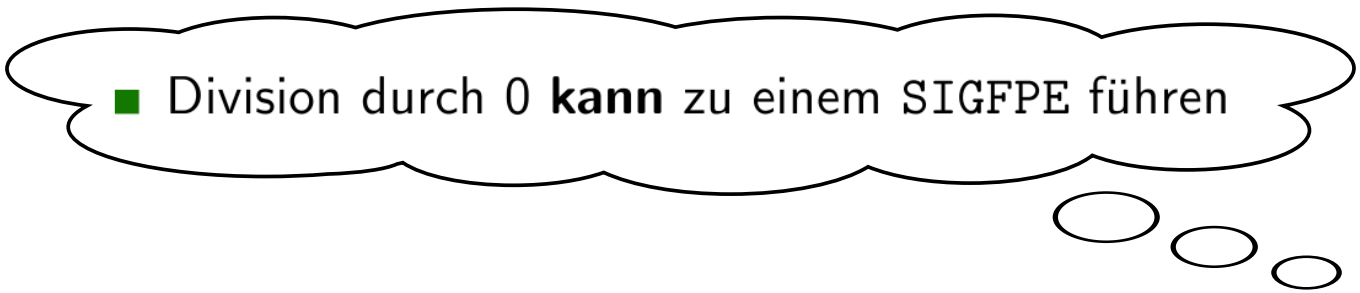


Große Probleme



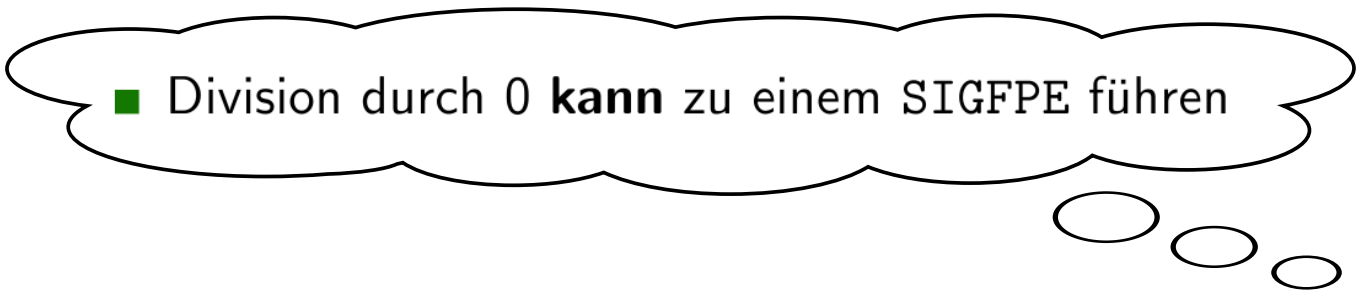
- Division durch 0 - *undefiniert*

Große Probleme



- Division durch 0 - *undefiniert*
- Zurückkehren aus einem Error-handler - *undefiniert*
- BS bemerkt nicht, dass Error-handler verlassen wurde

Große Probleme



- Division durch 0 - *undefiniert*
- Zurückkehren aus einem Error-handler - *undefiniert*
- BS bemerkt nicht, dass Error-handler verlassen wurde
- Undefined Behaviour - ?

Andere Strategien

- Neustart
  - Teuer
  - Erst nach der Reparatur sinnvoll

Andere Strategien

- Neustart
  - Teuer
  - Erst nach der Reparatur sinnvoll
- Redundanz
  - Checkpoints - regelmäßig Daten / Systemzustand sichern
  - RAID - Speicherredundanz

Andere Strategien

- Neustart
  - Teuer
  - Erst nach der Reparatur sinnvoll
- Redundanz
  - Checkpoints - regelmäßig Daten / Systemzustand sichern
  - RAID - Speicherredundanz
- Immunität ?
  - Im Fehlerfall die Ursache zurückverfolgen
  - Z.B. `core dump` Datei automatisch analysieren
  - Programm in kritische und unkritische Blöcke unterteilen
  - Wenn Ursache im unkritischen Block, könnte toleriert werden

Fazit

- Komplexe Aufgabe
  - Vielschichtig - Algorithmik, Software, Hardware
  - Vielseitig - Kontrollverlust, Datenverlust, irreversibler Schaden

Fazit

- Komplexe Aufgabe
  - Vielschichtig - Algorithmik, Software, Hardware
  - Vielseitig - Kontrollverlust, Datenverlust, irreversibler Schaden
- Prävention
- Reaktion

Fazit

- Komplexe Aufgabe
  - Vielschichtig - Algorithmik, Software, Hardware
  - Vielseitig - Kontrollverlust, Datenverlust, irreversibler Schaden
- Prävention
  - Nicht immer möglich
  - Wie weit betreiben?
- Reaktion

Fazit

- Komplexe Aufgabe
  - Vielschichtig - Algorithmik, Software, Hardware
  - Vielseitig - Kontrollverlust, Datenverlust, irreversibler Schaden
- Prävention
  - Nicht immer möglich
  - Wie weit betreiben?
- Reaktion
  - Durchaus Bedarf
  - Einziger Weg - `setjmp/longjmp`
  - An unzählige Einschränkungen gebunden
  - Gefahrenquelle
  - Keine allgemeine Lösung
  - Nicht trivialer Effizienzgewinn
  - Zu anwendungsspezifische Entscheidungen

Fazit

- Komplexe Aufgabe
  - Vielschichtig - Algorithmik, Software, Hardware
  - Vielseitig - Kontrollverlust, Datenverlust, irreversibler Schaden
- Prävention
  - Nicht immer möglich
  - Wie weit betreiben?
- Reaktion
  - Durchaus Bedarf
  - Einziger Weg - `setjmp/longjmp`
  - An unzählige Einschränkungen gebunden
  - Gefahrenquelle
  - Keine allgemeine Lösung
  - Nicht trivialer Effizienzgewinn
  - Zu anwendungsspezifische Entscheidungen
- Undefined Behaviour
  -