Fehlertolerante Programmierung in C

— Seminarbericht —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von: Anna Fuchs
Betreuer: Michael Kuhn

Hamburg, den 30.03.2014

Abstract

Die Ausarbeitung ist im Rahmen eines Seminares "Effiziente C-Programmierung" entstanden und behandelt das Thema der Fehlertoleranz der in Programmiersprache C geschriebenen Anwendungen mit Hinblick auf die Laufzeit und weiterreichende Effizienzkriterien.

Contents

1 Begriffsklärung und Motivation

Moderne Computersysteme werden stetig leistungsfähiger und komplexer, die Skalierbarkeit steigt. Diese Entwicklung bleibt nicht ohne Konsequenzen. Die Investitionen sollten sich lohnen und so steigen entsprechend auch die Anforderungen an die Systeme. Verfügbarkeit, Sicherheit, Zuverlässigkeit und Leistung sind nur einige davon. Je komplexer das System und je mehr einzelne Komponenten beteiligt sind, umso höher ist die Ausfallwahrscheinlichkeit dieser. Die Wartbarkeit wird durch die umfassende Vernetzung und Komplexität auch zunehmend erschwert. Die Effizienz der Software muss aber mithalten können. Es ist somit eine durchaus schwierige Aufgabe diese Facetten in Balance zu halten.

Eine nicht unwichtige Konsequenz aus dem Wartungs-, Entwicklungs- und Reparaturaufwand sind die Kosten im weitesten Sinne. Vor allem die Fehlersuche, Reparatur und erneute Inbetriebnahme der Software verursachen oft nicht eingeplante Mehrkosten.

Ein nicht unumstrittener Ansatz der Fehlerbehandlung ist die Fehlertoleranz, von der man sich verspricht, die Kosten und den Aufwand zu reduzieren. Es gibt auch Einsatzgebiete, die einen Fehler nicht nur wegen der Kosten nicht dulden können. Darunter fallen z.B. sicherheitskritische Echtzeitsysteme, dessen Absturz nicht tragbar ist. Auch interaktive Systeme, deren Funktionalität zwar nicht zwingend mit der Sicherheit verknüpft wird, die Ausfallzeiten für den Benutzer jedoch unzumutbar wären, stellen einen möglichen Einsatzgebiet dar. Besonders teuer sind Ausfälle bei massiv parallelen Systemen, da hier die Rechenzeit von Millionen Prozessoren, sowie der Strom besonders kostenintensiv sind.

Der Begriff der Fehlertoleranz lässt einen relativ großen Spielraum für Interpretationen. Darum sollten die zwei Begriffe "Fehler" und "Toleranz" möglichst präzise im gegebenen Kontext definiert werden.

Im Kontext der Programmierung können Fehler auf verschiedene Weisen klassifiziert werden, so z.B. nach ihrer Ursache, Wirkung oder dem Vorkommen in bestimmten Regionen und Abschnitten. Die erste Unterteilung, die hier vorgenommen wird, unterscheidet die Fehler nach ihrer Wirkung in kritische und unkritische Fehler. Unter kritischen Fehlern werden hier solche verstanden, die die Anwendung zur Laufzeit nach dem Eintritt des Fehlers zum Absturz bringen würde. Im Gegenzug bedeutet unkritisch - die Anwendung läuft nach dem Fehler weiter. Diese Klassifizierung mag nicht immer eindeutig und angemessen sein. Zum einen lässt der Compiler einen gewissen Spielraum. So könnten Warnungen einer lauffähigen Anwendung explizit als Fehler verstanden werden, die das Programm gar nicht erst bauen lassen würden. Auf der anderen Seite verleihen die beiden Begriffe dazu, die kritischen Fehler als schlimmere oder gefährlichere anzusehen, was nicht zwingend der Fall sein muss. Fehler, die weder vor noch zur Laufzeit auftreten und somit unentdeckt bleiben, können durchaus einen viel größeren Schaden anrichten und

geben keine Möglichkeit diese rechtzeitig zu beheben, da sie nicht auffallen. Der Kern dieser Differenzierung bezieht sich somit primär auf die Konsequenzen des konkreten Fehlers und an der Stelle weniger auf die Hintergründe.

Die Ursachen des Fehlers können aber nicht komplett ungeachtet bleiben. Eine weitere nötige Unterscheidung muss vorgenommen werden, um das zu erörternde Thema hinreichend abzugrenzen. Die Fehler können ihre Wurzeln ganz allgemein in vielen Abstraktionsebenen haben. Die Logik oder der Algorithmus der Anwendung können fehlerhaft sein. Diese Fehlerquelle ist allerdings anwendungsspezifisch und soll hier nicht weiter betrachtet werden. Die zwei Komponenten, die später einen Programmlauf bestimmen, sind Hard- und Software. Hardwarefehler bleiben in dieser Arbeit unberührt, Softwarefehler dagegen stehen im Fokus ungeachtet dessen, ob der Algorithmus falsch umgesetzt wurde oder es sich um softwareinterne, sprachspezifische Fehler handelt (z.B. Dereferenzierung eines Integers). Die Arbeit behandelt somit Softwarefehler jeglicher Art, die das Programm zum Absturz bringen.

Was heißt es nun in diesem Kontext, die kritischen Fehler zu tolerieren? Man könnte darunter synonym das Ignorieren der Fehler oder angemessenes Reagieren auf solche verstehen. Das Konzept der Fehlertoleranz in diesem Sinne meint hier das Verhindern des Programmabbruchs, sodass kritische Fehler nicht ohne weiteres ignoriert werden dürften. In welchem Ausmaß die Reaktion verstanden werden und wann diese als angemessen angesehen werden kann, wird sich beim näheren Betrachten der Problematik herauskristallisieren. Bereits an der Stelle sollte man im Hinterkopf behalten, ob und wie die Anwendung nach dem Fehler weiterführt werden soll. Die Fragen, ob der Originalzustand verlustfrei wiederhergestellt werden kann oder eine Annäherung hinnehmbar wäre, stellen sich in jedem Fall hinter der Problematik des Programmabbruches.

Zusammenfassend lässt sich sagen, dass dem Ansatz der Fehlertoleranz überall dort eine Chance gegeben werden kann, wo sich der Aufwand gegenüber anderen Verfahren auszahlt. Unter andere Verfahren fallen meist ein Neustart oder die Redundanz in unterschiedlichstem Ausmaß.

2 Strategien

Im Fokus dieses Kapitels steht der Programmabbruch, der verhindert werden soll. Dazu werden zwei grundlegend verschiedene Strategien untersucht - die präventiven Maßnahmen, die einen kritischen Fehler und seine Auswirkungen verhindern sollen und die Reaktionsmaßnahmen, die den Fehler passieren lassen und eine angemessene Reaktion einleiten. Je nach Kontext ist die weitere Fragestelldung, wie und ob das Programm weiterlaufen soll.

2.1 Prävention

Um einen Fehler zu verhindern, muss dieser hinreichend genau identifiziert werden. Die erste Vorbeugungsmaßnahme kann bereits statisch getroffen werden, indem der Code noch vor der Laufzeit auf mögliche Fehler und kritische Stellen überprüft wird. Dabei erhält der Entwickler bereits von Editoren unterstützt, die simple Fehlerquellen wie die Syntax aufzeigen können. Viele potentielle Fehler findet auch der Compiler. Logische und semantische Fehler kann jedoch nach wie vor niemand besser lokalisieren, als der Mensch selbst. Code-Reviewing bleibt daher unverzichtbar. Zur Unterstützung werden Werkzeuge wie Static Code Analyzer entwickelt, die ggf. auch grafisch helfen, dem Verlauf des Programms besser folgen zu können und somit auch semantische Fehler deutlicher werden lassen.

Statisch lassen sich jedoch nicht alle Fehlerquellen finden, vor allem wenn der Ablauf des Programms von dynamischen Eingaben abhängt. Zu den statisch ggf. nicht entdeckten Fehlerquellen wie z.B. fehlende Initialisierung kommen zur Laufzeit diverse andere Gefahren

- Dynamischer Speicher
  - Hängende, doppelte, ungültige Zeiger
  - Stack-Überlauf
  - Heap-Überlauf
  - Fehlende, doppelte, ungültige Speicherfreigabe
- Nebenläufigkeit
- Semantische Inhalte
- Undefiniertes Verhalten

Als Werkzeug zur Vorbeugung der Fehler zur Laufzeit bietet C einige Sprachkonzepte an.

`if`

Das überall bekannte Konstrukt der Verzweigungen wird zur Steuerkontrolle benutzt. Damit können dynamisch Abfragen durchgeführt und bestimmte Ausdrücke auf Ungültigkeit überprüft werden.

`assert`

`assert`-Makro ist definiert in `assert.h` und bietet die Möglichkeit Testpunkte im Programm zu setzen, an den bestimmte Behauptungen überprüft werden. Der zu überprüfende Ausdruck muss einen Integer als Rückgabewert liefern.

Listing 2.1: `assert` Syntax

```
1 void assert (int expression);
```

Das Konzept ist als Zusicherung bekannt. Wird der Ausdruck zu 0 ausgewertet, wird *asertion failure* ausgelöst und das Programm terminiert mit einer vom Compiler abhängigen Fehlermeldung. Die Ausdrücke, die ausgewertet werden, sollten den Programmzustand nicht verändern. Einen Vorteil gegenüber Abfragen im `if`-Block stellt die Möglichkeit dar, die Zusicherungen zu ignorieren, indem ein Makro `NDEBUG` definiert wird.

Seit dem Standard C11 gibt es auch statische Zusicherungen, die zur Kompilierzeit ausgewertet werden und das Fehlerrisiko somit noch vor der Laufzeit minimieren sollen.

Das Konzept wird im Kontext dieser Arbeit allerdings keine Anwendung finden, da das Programm im Falle einer nicht erfolgreichen Zusicherung terminiert werden würde. Dies ist sicherlich eine schönere Art, das Programm zu terminieren, als den Fehler tatsächlich passieren und größeren Schaden anrichten zu lassen. Das primäre Ziel bleibt jedoch bestehen - der Programmabbruch soll verhindert werden.

`goto`

Mit der `goto`-Anweisung lassen sich Kontextsprünge innerhalb eines Funktionsblocks ermöglichen. Dies ist eine der umstrittensten Funktionen, die relativ mächtig, damit aber auch gefährlich ist.

Die Syntax von der Anweisung sieht wie folgt aus

Listing 2.2: `assert` Syntax

```
1 goto LABEL;
2 LABEL : expression;
```

Die Beispiele 2.3 und 2.4 zeigen eine mögliche Benutzung von `goto`. Besonders Verschachtlungen, deren Ausführungstiefe zur Laufzeit bestimmt wird, können mit

`goto` übersichtlicher gestaltet werden. Würde das Beispiel, realisiert mit Abfragen, noch zusätzliche `else`-Blöcke oder Schleifen enthalten, wäre das Verlassen des jeweiligen Blocks sehr umständlich. Man sollte allerdings beachten, dass auch das `goto` die Lesbarkeit erheblich beeinträchtigen kann.

Die Laufzeit dieses Beispiels bleibt ungeachtet der Realisierung mit oder ohne der Sprünge gleich.

Listing 2.3: `goto` Anwednung

```
1  int stuff(N)
2  {
3    int ret = -1;
4    int* p1 = space(N);
5    if(check(p1,N)==0)
6      goto error1;
7
8    int* p2 = space(N);
9    if(check(p2,N) == 0)
10     goto error2;
11
12   int* p3 = space(N);
13   if(check(p3,N) == 0)
14     goto error3;
15
16   ret = todo(p1,p2,p3);
17
18 error3:
19   cleanup(p3);
20 error2:
21   cleanup(p2);
22 error1:
23   cleanup(p1);
24
25   return ret;
26 }
```

Listing 2.4: Verschachtelte Anwednung

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

Solche Sprungbefehle können gefährlich sein, vor allem wenn in lokale Blöcke gesprungen und auf Werte zugegriffen wird, deren Deklaration oder Initialisierung übersprungen wurde. Das Verhalten in diesen Fällen ist betriebssystemunabhängig oder undefiniert.

`try-catch`

Ausdrücke, die einen Fehler verursachen würden, können in `if-else`-Blöcken nicht direkt ausgewertet werden, da dies zum Absturz führen würden. Es wäre nützlich, wenn man solche Ausdrücke dennoch in einer sicheren Umgebung "ausprobieren" könnte. Das Konzept `try-catch` ist vor allem aus der Programmiersprache Java bekannt. Eine Analogie gibt es in C nicht. Es lässt sich mit Hilfe anderer Konstrukte bedingt simulieren. Unerlässlich sind dabei die Funktionen `setjmp` und `longjmp`

2.2 `setjmp/longjmp`

Diese zwei wenig bekannten Funktionen ermöglichen Kontextsprünge über die Funktionsgrenzen hinaus.

Listing 2.5: `assert` Syntax

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

`setjmp` hat dabei einen Rückgabewert vom Typ `int` und nimmt `jmp_buf env` als Parameter. Es wird 0 zurückgegeben, wenn die Funktion zum ersten Mal und direkt im normalen Programmfluss aufgerufen wird. `setjmp` setzt eine Art Marker, ähnlich dem Label beim `goto`, zu dem in Zukunft gesprungen werden kann. Die Funktion `longjmp`, die keinen Rückgabewert hat, hat als Parameter denselben Buffer und einen Wert `val`, der von `setjmp` zurückgegeben wird, nachdem ein Sprung erfolgt ist. Das Konzept funktioniert ähnlich dem invertierten `goto`, da hier erst der Marker gesetzt wird und somit dieser Programmteil auf jedem Fall schon mindestens einmal abgearbeitet worden sein muss. Der Wert `val` darf dabei nicht 0 und der Buffer muss identisch sein, damit in einem Programm mehrere verschiedene Sprünge möglich bleiben. `jmpbuf` ist der Datentyp der Speichers, der den aktuellen Programmzustand hält. Zum gespeicherten Inhalt gehört der Kontext bzw. die Programmumgebung. Darunter versteht man das *calling environment* und ggf. die Signalmaske. Calling environment hält den Inhalt lokaler Variablen, den *Stack*- und *Framepointer*, den *Programcounter* und ggf. einige anderen Werte abhängig von der Architektur.

Folgendes Beispiel zeigt die Wirkung der beiden Funktionen

```
1  jmp_buf jbuf;
2  volatile int glob;
3
4  void func()
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     func();
24     printf("Das wird nie ausgegeben. \n");
25   }
26
27   printf("loc = %d, glob = %d \n", loc, glob);
28   return 0;
29 }
```

Es werden zwei globale Variablen `jbuf` und `glob` angelegt. Das Schlüsselwort `volatile` bedeutet für den Compiler, dass der Wert von außerhalb verändert werden kann und somit bei jedem Zugriff aus dem Hauptspeicher gelesen werden muss. Die Optimierung, bei der der Compiler einige Werte in Registern zwischenspeichert, entfällt somit. In der folgenden Tabelle ist der Ablauf des Beispiels chronologisch kommentiert, um die Funktionalität besser zu verstehen.

Zeile	Kommentar	Ausgabe
13	`loc` auf 10	
14	`glob` auf 20	
16	Marker zum Sprung setzen, Programmzustand merken	
17		`setjmp`
18	Den Block betreten, da `rc` = 0	
19		rc=0
20	Überschreiben von `loc` mit 100	
21	`func` aufrufen	
22	Diese Zeile wird nie ausgeführt, weil nie aus `func` zurückgekehrt wird	
4	`func` aufrufen	
6		`lognjmp`
7	`glob` mit 200 überschreiben	
8	Springen zum gesetzten Marker von `jbuf` mit dem Wert 1	
16	Hierhin gesprungen, der Wert von `jbuf` gelesen und geladen	
17		`setjmp`
19	Der Block wird nicht mehr betreten, weil der Rückgabewert von `setjmp val` war, und `val` auf 1 gesetzt wurde	
27	Der lokale Wert, wenn auch früher überschrieben, wurde im `jbuf` mit 10 gespeichert; spätere Zuweisungen werden vergessen. Globale Variablen sind allerdings nicht teil vom `jbuf` und werden beim Setzen des Markers nicht gemerkt	`loc = 10, glob = 200`

Kritisch sind Sprünge in die zukünftigen Funktionen. Die Funktion musste einmal besucht worden sein, damit dieser Marker gesetzt werden konnte. Hätte dieselbe Funktion in Zukunft mit Parameterübergabe aufgerufen werden sollen, mit `longjmp` aber dorthin gesprungen wird, so kennt diese Funktion noch keine Parameter. Denn frühere Parameter existieren ggf. nicht mehr, zukünftige wurden nicht übergeben. Ebenfalls ist es gefährlich in vergangene Funktonen zu springen, da die Werte überschrieben oder gelöscht worden sein könnten.

Mit diesen beiden Funktionen und der Hinzunahme vieler weiteren Sprachkonstrukte ist es möglich das `try-catch` Prinzip zu simulieren. Es ist allerdings nicht möglich, eine Art Versuchslabor für alle Fehlerarten zu implementieren, da das Konzept nur eine bequemere eingeschränkte Handhabung der Fehler darstellt und keine neue Funktionalität anbietet. Daher ist die Implementierung an der stelle nicht mehr zielführend und wird hier nicht vorgeführt. Die Funktionalität von `setjmp` und `longjmp` kann allerdings später noch nützlich sein.

3 Reaktion

Prävention alleine kann nicht alle Probleme lösen. Es können nur bestimmte Teilausdrücke auf ihren Wert hin untersucht oder vorhersehbare Gefahren abgegriffen werden. Jedoch gibt es auch sprachspezifische Einschränkungen, die nicht ohne weiteres einkalkuliert werden können. Typabfragen oder Grenzenüberprüfung vom Stack und Heap sind in C nicht vorgesehen. Aber auch Abfragen von wohldefinierten Werten bringen Kosten mit sich und können nicht beliebig betrieben werden.

Murphys Gesetz besagt: "Whatever can go wrong will go wrong."[1]. Man muss sowohl als Entwickler, als auch als Benutzer immer mit einem Fehler rechnen, sei er noch so unwahrscheinlich. Im besten Fall wäre es nicht zu lauern und darauf zu warten, sondern den Fehlerfall eintreten zu lassen und zu reagieren.

Es wird ein Fall betrachtet, bei dem es eine Möglichkeit gäbe, auch präventiv den Fehlerfall zu behandeln, es aber ggf. günstiger wäre, eine andere Strategie zu wählen.

3.1 Kosten

Als Beispiel soll ein System mit permanentem Input, der über Sensoren wahrgenommen wird, betrachtet werden. Angenommen es seien Sensoren, die die Entfernung zum nächsten Objekt messen, wobei diese Werte in spätere Berechnungen einfließen. Unter anderem soll eine Division durch den errechneten Wert stattfinden, sodass eine 0 nicht erlaubt wäre. Die präventiven Maßnahmen in diesem Fall wären inhaltliche Abfragen auf Ungültigkeit von jedem Wert. Einige Sensoren können bis zu Tausend Werte pro Sekunde weiterleiten, sodass die Rechenlast durch das Abfragen nicht unerheblich wird[1]. Solch eine Verzögerung wirkt sich bei interaktiven Reaktionen besonders negativ aus. Sei es ein Robotersystem, das auf die Entfernung mit bestimmtem Verhalten reagieren muss, so könnte die Verzögerung bereits zu lang sein und die Reaktion nicht mehr angemessen. Das Problem ließe sich in diesem konstruierten Fall auch auf physischer ebene lösen, indem die Konstruktion es gar nicht erst zulässt, dass die Entfernung zu einem bestimmten Objekt null wird. So unwahrscheinliche Situationen, wie die Wahrnehmung von Staub oder z.B. eines Insekten, der sich unmittelbar auf dem Sensor befinden würde, könnten einen solcher Fehlerfall verursachen. Sollte die Wahrscheinlichkeit für eine solche Situation als relativ gering abgeschätzt werden, so lohnt es sich umso mehr zu überlegen, ob permanente

[1] US-amerikanischen Ingenieur Edward A. Murphy

Überprüfungen rentabel sind. Die bessere Strategie lautet hier, unwahrscheinliche Fehler passieren lassen und ggf. zu reagieren und zu reparieren.

Bevor man diesen mit Sicherheit komplizierteren Ansatz verfolgt, sollten die Gesamtkosten nochmal überschlagen werden. Eine fiktive Beispielrechnung könnte bereits aufschlussreich sein.

Es werden n-viele Abfragen betrachtet, die jeweils eine konstante zeit m $O(1)$ dauern. Hinzu kommt die Wahrscheinlichkeit P, dass ein Fehler passiert. Die Fehlerbehandlungskosten A mittels Prävention seien konstant x $O(1)$ teuer, Kosten für ein Reaktion B seien y $O(1)$ teuer. Generell lässt sich sagen, dass $A << B$ gilt, da beim Letzteren ohne jegliches Vorwissen auf eine de facto Situation reagiert werden muss, während die Präventiven Maßnahmen den Fehlerfall viel genauer spezifizieren müssen und daher eher in der Summe aller möglichen Fälle teurer werden. Die Hochrechnung $n \cdot m + P \cdot A >? P \cdot B$ lässt überblicken, welches der Verfahren effizienter sein würde. Eine gesonderte Schwierigkeit ist sicherlich im Vorfeld die Wahrscheinlichkeit P abzuschätzen.

3.2 Error

Nachdem die komplexere Strategie gewählt wurde, stellt sich die Frage, worauf genau reagiert werden muss. Wer oder was bricht das Programm ab? Aus der Sicht des Benutzers liegt die Verantwortung beim "error", intern entscheidet aber das Betriebssystem über die Terminierung eines Programms oder Prozesses. Die Ursache für die Terminierung ist der kritische Fehler.

Es ist nötig sich an der Stelle den Fehler genauer anzuschauen. Es gibt Aufrufe, die im Fehlerfall einen Fehlerwert liefern. Dies sind im Allgemeinen Aufrufe, die in einer Bibliothek oder einem Funktionssatz definiert sind. Der Fehlerwert kann abgefangen, umgeleitet, ignoriert, etc. werden. An der Stelle ist es nicht weiter schwer dem Betriebssystem die Verantwortung für einen solchen Fehler und die Konsequenzen daraus abzunehmen. Eine andere Art von Fehler generiert ein Signal auf Betriebssystemebene (hier werden nur Linux bzw. UNIX-artige Systeme betrachtet). Im allgemeinen sind dies Fehler, die unabhängig von der Definition in einer Bibliothek überall passieren können. Dazu gehören z.b. Division durch null, ungültige Speicherzugriffe, fehlerhafte Ein- und Ausgabe. Die Signale werden in dieser Arbeit in drei Gruppen aufgeteilt - fatale, kritische und unkritische. Unter den fatalen Signalen werden solche verstanden, die man nicht beeinflussen kann, d.h. nicht umleiten, abfangen, ignorieren oder aufhalten. Kritische Signale können in dieser Hinsicht manipuliert werden, blieben sie unberührt, würden sie jedoch das Programm abbrechen. Und zuletzt die unkritischen Signale, wie z.B. `SIGCHLD`, die keinen direkten Einfluss auf den Programmablauf haben und lediglich Statusinformationen beinhalten (z.B. über den Zustand des Kindprozesses).

Zu den fatalen Signalen gehören `SIGKILL` und `SIGSTOP`. Im folgenden wird es nur um kritische oder ggf. unkritische Signale gehen, die fatalen sind ausgenommen.

Ein (kritisches) Signal kann in einem Signalhandler blockiert werden. Dazu biete C zwei Funktionen an (siehe dazu mehr in der Manpage):

Listing 3.1: Signalhandler

```
1 typedef void (*sighandler_t)(int);
2 sighandler_t signal(int signum, sighandler_t handler);
3
4 int sigaction(int signum, const struct sigaction *act,
5                         struct sigaction *oldact);
```

Die Signale werden innerhalb des Funktionsblocks blockiert. Nach dem Verlassen des Signalhanlders ist das Verhalten weiter undefiniert. Innerhalb des Signalhandlers können z.B. temporäre Dateien gelöscht oder der Speicher aufgeräumt werden. Mit Hilfe der oben erwähnten Funktionen können Signale auch ignoriert werden. Es sollten nur unkritische Signale ignoriert werden, da das Ignorieren anderer Signale zu undefiniertem Verhalten führt[2].

3.3 Beispiel

Am folgenden Beispiel wird die Funktionsweise der Signalhandler gezeigt. Als Auslöser des Signals `SIGFPE` soll die Division durch Null durchgeführt (Hinweise zur Division durch null im Abschnitt 3.5).

Das Programm soll in einer Schleife Berechnungen durchführen, die auch eine Division enthalten. Dividiert wird dabei durch einen Wert aus dem globalen Array, das die Eingabedaten darstellt. Die Schleifenindizes sind global, damit sie beim Zurücksetzen ihren Wert beibehalten und die Schleife nicht neu abgearbeitet werden muss. Die ungewöhnliche Form der Schleifenindizes `n=n` und `i=i` dient der Möglichkeit des Wiedereintritts in die Schleife. Der Signalhandler sollte nach jedem Gebrauch wieder neu initialisiert werden, daher steht der Aufruf unmittelbar nach dem `setjmp`. Schleifenindex i wird in jedem äußeren Schleifendurchlauf resettet. Der Gebrauch von globale Indizes ist in diesem Fall natürlich nicht optimal, da das Beispiel aber recht künstlich konstruiert ist soll es nur der Veranschaulichung dienen. Das Design eines echten Programms mit dynamischem Input würde vermutlich anders aussehen. Ebenso ist der Eingabedatensatz ein vorbelegtes Array, welches in einem realen Beispiel zur Laufzeit generiert werden würde und das Design, sowie der Ablauf anders als hier dargestellt wären.

Wichtig ist die Zuweisung innerhalb des Signalhandlers. Einmal in den Signalhandler eingetreten, muss die Fehlerursache repariert werden, da das BS den Fehler immer wieder triggern würde. Innerhalb des Signalhandlers gibt es einen eigenen Stack, der sich vom Programmstack unterscheidet.

```
1  jmp_buf jbuf;                  // environment buffer
2  volatile int stuff[5] = {11, 0, 13, 14, 15};  // input 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=99; 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 }
```

Die folgende Tabelle stellt den chronologischen Ablauf des Programms dar.

Zeile	Ausführliche Kommentare
14	Lokale Variable anlegen und initialisieren
15	Globale Variablen Initialisieren
16	Rücksprungmarker setzen
17	Signalhandler initialisieren
19	Erster Schleifendurchlauf der äußeren Schleife für $n = 99$
21	Erster Schleifendurchlauf der inneren Schleife für $i = 0$
23	Berechnung von a: $99/11 = 0$
24	Berechnung von c
21	Zweiter Schleifendurchlauf der inneren Schleife für $i = 1$
23	Berechnung von a: $99/0$ führt zum Fehler
6	Signalhandler wird aufgerufen
8	Der den Fehler verursachende Wert an der Stelle i wird durch einen zulässigen ersetzt
9	Sprung zum Marker mit dem Wert 1
16	Hierhin gesprungen
17	Signalhandler wird neu initialisiert
19	Noch immer der erste Durchlauf der äußeren Schleife für $n = 99$
20	Da i eine globale Variable ist, wurde der Wert nicht zurückgesetzt, sodass bei $i = 1$ weiter gerechnet wird
23	Berechnung von a: $99/1$ führt nicht mehr zum Fehler
...	

3.4 Laufzeit

Laufzeitmessungen des Programms zeigen ein kurioses Bild. Es wurden 500.000.000 äußere Iterationen auf einem Intel Xeon Westmere 5650 @2,67 GHz, 2 Gbyte DDR3 / PC1333 Hauptspeicher und dem 64-bit Betriebssystem Ubuntu 12.04.3 LTS gemessen. Das Schlüsselwort `volatile` hat offenbar das meiste Optimierungspotenzial ausgeschaltet und somit wirkt sich die Optimierung auf `O3` nicht anders aus als die Optimierungsstufe `O0`.

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

Gemessen wurde zum einen die Variante mit eingebautem fehlertoleranten Mechanismus bei 0 bis 3 eingetretenen Fehlern. Die Laufzeit blieb dabei konstant bei 33,8 Sekunden, sodass der Aufwand der Reparatur minimal zu sein scheint. Alternativ dazu wurde dasselbe Beispiel mit denselben Voraussetzungen (dieselben Variablen global und `volatile`) mit präventiven Maßnahmen in Form von `if`-Abfragen von jedem Wert implementiert (in der Tabelle unter Testfall "`if`"). Die Laufzeit dieser Variante ist rund 5 Sekunden länger. Zusätzlich wurden zwei weitere Abfragen ergänzt (Testfall "`ifififif`"), um den Fall zu untersuchen, dass nicht nur ein, sondern mehrere Wert der Eingabe unzulässig sind. Das Resultat ist eine um 3 Sekunden längere Laufzeit. Denkbar ist, dass bei komplexeren Ausdrücken, die in den Abfragen ausgewertet werden, entsprechend mit einem deutlichen Mehraufwand zu rechnen ist. Die Built-In Funktion des Compilers `__builtin_expect ((x),0)`, bekannt unter *unlikely* (oder *likely* mit `((x),1)`) wirkt sich nicht positiv auf die Laufzeit aus. Die Funktion hilft dem Compiler bei der Erwartungshaltung, ob der abgefragte Fall eintreten wird oder nicht. Eine positive Haltung (likely) bremst das Programm um eine zusätzliche Sekunde aus, während unlikely kaum von den Zeiten abweicht, die ohne jegliche Compiler-Hints gemessen wurden. Die Ursache kann darin liegen, dass der Datensatz statisch aus nur 5 Einträgen besteht, die Milliarden male gelesen werden. Das Programm kann durchaus zur Laufzeit bedingt lernen und erkennen, dass die Bedingung nahezu nie erfüllt wird und die Erwartungshaltung selbst entsprechend wählen.

Das überraschende Ergebnis ist allerdings die Messung von einem Programm, das gar keine Fehlerbehandlung vorsieht (Testfall "ohne `if` ohne error"), weder durch Fehlertoleranz noch durch Abfragen. In diesem Fall wurden alle ungültigen Einträge aus dem Eingabedatensatz eliminiert. Diese Laufzeit ist jedoch eine halbe Sekunde länger, als der fehlertolerante Fall. Dies scheint unlogisch, da auch hier mindestens genauso viel Arbeit verrichtet werden muss. Die Messungen wurden ca. zehn mal in isolierter Umgebung durchgeführt, wodurch eine Laufzeitschwankung oder äußere Einflüsse nahezu ausgeschlossen werden können. Der Assembler Code der beiden Codeabschnitte unterscheidet sich lediglich in einer Zeile:

Listing 3.2: Signalhandler

```
1 setjmp: cmpq    $\$$2, $\%$rax
2 ohne if ohne error: \cmpq   $\$$499999999, $\%$rax
```

Dies Zeilen hängen mit dem Vergleich des Schleifenindex zusammen. Genauere Gründe konnten im Rahmen dieser Arbeit nicht herausgefunden werden und müssen näher untersucht werden.

3.5 Offene Fragen

Der vorgestellte Ansatz beinhaltet nicht wenige kleinere und größere Schwierigkeiten. Zum einen sollte die Funktion `signal` laut dem Standard aus Portabilitätsgründen nicht mehr benutzt werden. Alternativ dazu steht nun die Funktion `sigaction` zur Verfügung, die eine komplexeres Interface hat, in der Funktionalität jedoch mindestens der von `signal` entspricht. Ebenfalls im Kontext dieses Beispiels sollten die Funktionen `setjmp` und `longjmp` nicht benutzt werden. Möchte man diese zusammen mit Signalen benutzen, so stehen hierfür sicherere `sigsetjmp` und `siglongjmp` Funktionen zur Verfügung, die erweiterte Funktionalität für die Signalmaske. Die hier benutzten Funktionen sind außerdem *async-signal-unsafe*, d.h. beim auslösen mehrerer Signale während oder vor dem aktiven Handlers diese nicht korrekt abgearbeitet werden könnten. `volatile` erlischt die Vorteile der Optmierungen, ohne das könnte der Compiler allerdings einen fehlerhaften Code produzieren, der zusammen mit `longjmp` zu unerwartetem Verhalten frühen könnte. Diese Probleme lassen sich jedoch auf mehr oder weniger aufwändigem Weg lösen.

Es gibt eine Reihe anderer Hindernisse, die den Ansatz ggf. diskreditieren können. Der Ansatz macht einige Annahmen, die unzutreffend sein können. So wurde in dem Beispiel angenommen, dass die Division durch null zu einem `SIGFPE` führen kann. Das Verhalten in diesem Fall ist aber streng genommen undefiniert. Ebenso wie das Zurückkehren aus dem Handler undefiniert ist, jedoch nicht garantiert wird, dass der Sprung aus dem Handler nicht als gewöhnlichen Zurückkehren verstanden wird. Unter Umständen ist es möglich, dass das BS trotz des korrekten Ablauf nicht erkennt, das der Signalhandler verlassen wurde. In diesem Fall existiert aus der Sicht des Betriebssystems nur der separate Stack, der für den Handler aufgebaut wurde.

Alle oben erwähnten Fälle müssen zwingend tiefer untersucht werden, da das undefinierte Verhalten nicht behandelt werden kann. Im Vorfeld wurde vorausgesetzt, dass der Fehlerfall identifizierbar ist. Undefiniertes Verhalten ist jedoch nicht greifbar. Weder der Programmierer, noch die Anwendung oder das Betriebssystem dürfen im Falle des undefinierten Verhaltens Annahmen darüber machen. Man sollte auch keine Spekulation anstellen, die leider im falle der Division durch null weit verbreitet sind. Auch wenn es zufälligerweise auf der gegeben Architektur der Fall ist, dass `SIGFPE` Signal ausgelöst wurde, so ist das Verhalten dennoch undefiniert und kann nicht behandelt werden.

Die Problematik der kritischen Signale als solche ist also nicht das primäre Kriterium. Solange der Kontrollfluss an einer beliebigen stelle undefiniertes Verhalten zulässt, kann der Fehler nicht behandelt werden und vor allem kann keine Rede vom Weiterlaufen sein.

Andere Strategien können sich als effizienter und verlässlicher erweisen. So ist ein Neustart des Systems nach den Reparaturmaßnahmen vermutlich die sicherste Variante, die jedoch auch sehr kostenintensiv oder nicht immer möglich ist. Die Redundanz ist die gängige Praxis. Es werden Daten redundant gespeichert (Stichwort RAID) oder auch der Systemzustand durch redundante Speicherung eines stabilen Zustandes regelmäßig gesichert.

Eine eher futuristische Alternative könnte eine "selbstheilende" Software sein, die nach dem Vorbild der menschlichen Immunität ihre Defizite selbst ausgleichen kann. Hier wäre es denkbar, das Programm in kritische und unkritische Blöcke zu unterteilen. Tritt der Fehler in einem unkritischen Block, so könnte dieser toleriert werden. Im Idealfall würde die Ursache möglichst genau analysiert und repariert werden. Dies setzt aber einen Ansatz der Intelligenz voraus, der über das Konzept des maschinellen Lernens hinaus zum heutigen Zeitpunkt unrealistisch scheint.

4 Fazit

Eine Fehlertolerante Software zu entwickeln ist keine triviale Aufgabe. Die Problematik ist sehr vielschichtig - Algorithmen, Programmiersprache, Hardware. Man trifft sehr schnell auf erhebliche Schwierigkeiten, die sich nicht immer mit präventiven Maßnahmen lösen lassen. Hier stellt sich zusätzlich die Frage, wie weit solche Maßnahmen noch zweckmäßig betreiben werden sollten. Die Strategie, auf bereits geschehene Fehler zu reagieren, mag vielversprechend klingen, lässt sich jedoch im Rahmen existenter Systeme nicht optimal einpflegen. Es gibt keine allgemeine Lösung die einen definierten Effizienzgewinn oder vor allem Sicherheit und Korrektheit garantiert. Da man bei diesem Verfahren an unzählige Einschränkungen gebunden ist, sind alle zu treffenden Entscheidungen sehr anwendungsspezifisch. Letztendlich droht die Gefahrenquelle des fertigen Produktes ein höheres Risiko darzustellen, als die ursprüngliche Problematik. Dabei gibt es durchaus Anwendungsbereiche für fehlertolerante Software, allerdings muss das gesamte Konzept inkl.dem Betriebssystem und der Umgebung auch eine solide Basis dafür bieten. Bei der Betrachtung von gängigen Linux Distributionen und der Programmiersprache C muss die Funktionsfähigkeit des Konzeptes kritisch noch tiefer untersucht werden.

Bibliography

[1] Analog Devices Digital Accelerator. `http://www.analog.com/static/imported-files/data_sheets/ADXL345-EP.pdf`. [Online; accessed 30.04.2013].

[2] Robert Love. *Linux System Programming, Second Edition.* 2013.