FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Stefan Bader

Semi-Extended Tasks: Application-Specific Fine-Grained Task-Stack Sharing in OSEK Systems (dOSEK-SemiExtended)

Bachelorarbeit im Fach Informatik

29. Februar 2016

Please cite as:
Stefan Bader, “Semi-Extended Tasks: Application-Specific Fine-Grained Task-Stack Sharing in OSEK Systems (dOSEK-SemiExtended)” Bachelor’s Thesis, University of Erlangen, Dept. of Computer Science, Februar 2016.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Semi-Extended Tasks: Application-Specific Fine-Grained Task-Stack Sharing in OSEK Systems (dOSEK-SemiExtended)

Bachelorarbeit im Fach Informatik

vorgelegt von

Stefan Bader

geb. am 19. Mai 1994
in Nürnberg

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**PD Dr.-Ing. habil. Daniel Lohmann** **Christian Dietrich, M.Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. September 2015**
Abgabe der Arbeit:	**29. Februar 2016**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Stefan Bader)
Erlangen, 29. Februar 2016

ABSTRACT

Current real-time operating systems, as being used in the car industry, have to support more and more safety and comfort features.

The target of a car manufacturer is therefore to supply those features whilst keeping the manufacturing costs at a minimum, such as using cheaper microelectronic components. But cheap micro controllers only have limited system resources, so the used operating system should have a very low memory footprint. The target of this paper is to further reduce the memory footprint of OSEK tasks via stack sharing.

OSEK is a well used real-time operating system specification by the car industry. OSEK already supports stack sharing for *some* of its tasks, but not for all of them, as some tasks require a separate stack at certain points of time. Therefore the introduced method of stack sharing in this paper only switches to the shared stack when possible.

In order to find out where the stack can be switched, modifications on the static analysis have been implemented. Further modifications have been made on the compiler, so as to support customized code which switches the stack, and on the task switching mechanism of the operating system, as every task may now run on the shared stack.

At the end the results of this modified implementation have been evaluated by the change in runtime speed and modified memory footprint. The evaluation shows that there is barely any additional runtime as only 5 additional IA32 instructions are needed per stack switch, which, on an AMD64 processor from 2010, results in an additional runtime of 1.7 nano seconds per stack switch.

The change in memory footprint is highly application specific. Not every time one can switch the stack, automatically results in less memory usage. Quite the opposite might happen and the memory footprint might increase. Therefore an additional static analysis would be needed in order to decide whether a stack switch is not only possible but useful.

Should this additional static analysis be implemented, then the memory footprint could be, depending on the application, reduced while keeping additional runtime cost at a minimum.

KURZFASSUNG

Aktuelle Echtzeitbetriebssysteme, wie sie unter anderem in Automobilen vorkommen, sind immer größeren Anforderungen gegenübergestellt, was Sicherheits- und Komfortfunktionen angeht.

Dabei ist es Ziel der Automobilhersteller, diese anzubieten und gleichzeitig die Kosten pro Fahrzeug gering zu halten um so den Profit zu steigern. Eine Stelle an der Kosten gespart werden können ist somit auch die verbaute Mikroelektronik. Dabei stellt der benötigte Speicherbedarf des verwendeten Echtzeitbetriebssystems auf die Minimalvorraussetzung bei der Wahl von Mirkoprozessoren eine wichtige Rolle.

Dieser Speicherbedarf soll in dieser Arbeit weiter gesenkt werden, indem sich *alle* Ausführungseinheiten eines OSEK Betriebssystems den selben Stapel teilen können. OSEK ist dabei eine etablierte Echtzeitbetriebssystemspezifikation aus der Automobilbranche. OSEK unterstützt bereits geteilte Stapel für *manche* Ausführungseinheiten, aber noch nicht für alle. Da jedoch manche Ausführungseinheiten an bestimmten Stellen einen eigenen Stapel zwingend benötigen, wird in dieser Arbeit lediglich an den Stellen an denen es möglich ist, auf den geteilten Stapel gewechselt.

Um diesen Stapelwechsel zu realisieren, waren Modifizierungen an der statischen Analyse, um herauszufinden, wann der Stapel gewechselt werden kann, am Übersetzer, um den Stapelwechsel selbst auszuführen, und am Wechsel der Ausführungseinheiten des Betriebssystems, da nun potentiell jede Ausführungseinheit auf dem geteilten Stapel laufen kann, notwendig.

Am Ende wurden diese Modizierungen auf die veränderte Laufzeit und den veränderten Stapelverbrauch evaluiert. Da die Modifizierungen am Übersetzer sehr gering ausgefallen sind, sind die zusätzlichen Laufzeitkosten ebenfalls sehr gering. So werden für einen Stapelwechsel lediglich 5 zusätzliche IA32 Instruktionen benötigt, was, bei einem im Jahr 2010 hergestellten AMD64 Prozessor, zu einer um ca. 1.7 Nanosekunden gestiegenen Laufzeit beim Stapelwechsel führt.

Die Auswirkungen auf den Speicherverbrauch sind dabei leider nicht so trivial, wie ursprünglich angenommen. So gibt es durchaus Stellen bei den Ausführungseinheiten die zwar auf dem geteilten Stapel laufen können, dies aber zu einem erhöhten Speicherverbrauch führt.

Deswegen sollte in Zukunft eine zusätzliche Analysephase entscheiden, welche aus den gewählten Umschaltstellen auf den geteilten Stapel wirklich auf dem geteilten Stapel laufen sollen.

Mit dieser zusätzlichen Analysephase bietet es sich dann an einen, je nach Anwendung, gesunkenen Speicherverbrauch mit kaum existenten Mehrkosten bei der Laufzeit in Kauf zu nehmen.

INHALTSVERZEICHNIS

Abstract **v**

Kurzfassung **vii**

1 Einleitung **1**

2 Grundlagen **3**

2.1 Konventionen beim Funktionsaufruf 3

2.1.1 Generelles Konzept 3

2.1.2 Konkretes Beispiel bei Verwendung des gcc für die IA32 Architektur 5

2.2 OSEK 9

2.2.1 Leichtgewichtige Prozesse 9

2.2.2 Tasks in OSEK 11

2.2.2.1 Extended Tasks (ETs) 11

2.2.2.2 Basic Tasks (BTs) 12

2.2.3 Statische Konfigurierbarkeit in OSEK 12

2.2.4 Verwendete OSEK Implementierung 12

2.3 Zusammenfassung 13

3 Architektur **17**

3.1 Das Semi Extended Task Konzept 17

3.2 Die Stellen der Umschaltpunkte 18

3.2.1 Implementierung auf aufgerufene Funktion (callee) Seite 19

3.2.2 Implementierung auf aufrufende Funktion (caller) Seite 21

3.3 Auswahl der Umschaltpunkte 22

3.3.1 Funktionen die den Stapel nie wechseln können 23

3.3.2 Auswahl der Funktionen für eine callee seitige Implementierung 24

3.3.3 Auswahl der Funktionsaufrufe für eine caller seitige Implementierung 25

3.4 Nötige Modifizierungen am Betriebssystem 26

4 Analyse **29**
4.1 Veränderter Stapelspeicherverbrauch 29
4.1.1 Testmethodik 29
4.1.2 Fall 1: Stapelwechsel kann nicht zu einem verringerten Speicherverbrauch führen 30
4.1.3 Fall 2: Stapelwechsel führt immer zu verringertem Speicherverbrauch 31
4.1.4 Fall 3: Stapelwechsel führt manchmal zu verringertem Speicherverbrauch . . 32
4.2 Auswirkungen auf die Laufzeit 33
4.3 Zusammenfassung 35

5 Fazit **37**

Verzeichnisse **39**
Abkürzungsverzeichnis 39
Abbildungsverzeichnis 40
Tabellenverzeichnis 46
Quellcodeverzeichnis 48
Literatur 50

1 EINLEITUNG

Computer sind aus dem heutigen gesellschaftlichen Leben nicht mehr wegzudenken. Wir kommen mit ihnen in jedem Aspekt unseres Lebens in Berührung, sei es im Büro, zu Hause oder Unterwegs. Gerade Unterwegs, sei es in einem Automobil, Flugzeug oder einem Schiff, ist es wichtig, dass diese nicht versagen sondern uns vor möglichen Gefahrensituationen erfolgreich beschützen und/oder vorwarnen. Dabei steigen die Anforderungen der Kunden an die verbauten Computer Systeme nicht nur im Bezug auf sicherheitskritische Funktionen sondern auch auf Komfortfunktionen.

Um zwischen wichtigen und unwichtigen Ereignissen unterscheiden zu können, werden dabei Echtzeitbetriebssysteme eingesetzt. Diese Echtzeitbetriebssysteme sollen dafür sorgen, dass einerseits richtig evaluiert wird, wann bestimmte Aufgaben ausgeführt werden und dies möglichst Ressourcen schonend zu erledigen.

Dies wird erreicht, indem möglichst viele Informationen über den Ablauf der Anwendungen des Betriebssystems bereits zur Übersetzungszeit vorliegt und somit das Betriebssystem genauestens auf die benutzten Anwendungen angepasst werden kann. Eine Betriebssystemspezifikation, die vorraussetzt, dass alle Systemobjekte, wie die Ausführungseinheiten oder die möglicherweise auftretenden Ereignisse, bereits zur Übersetzungszeit spezifiziert sein müssen ist OSEK.

In OSEK müssen alle Systemobjekte, was nicht nur die Ausführungseinheiten sind, sondern auch die Ereignisse oder Unterbrechungen, statisch vorkonfiguriert werden. Es gibt in OSEK bisher zwei Arten an Ausführungseinheiten, Basic Tasks und Extended Tasks. Dabei können Basic Tasks in bisherigen modernen OSEK Implementierungen den selben Stapel verwenden um so den Speicherverbrauch zu minimieren. Ziel dieser Arbeit ist es, dass nun möglichst alle Extended Tasks zumindest teilweise auf diesem geteilten Stapel laufen können, also laufen manche Extended Tasks erst auf ihrem eigenen Stapel und können dann zu bestimmten Zeitpunkten auf den geteilten Stapel wechseln, um so den Speicherverbrauch weiter zu minimieren. Dabei soll die Laufzeit nicht stark negativ beeinflusst werden und die OSEK Spezifikation nicht angepasst werden. Die Tasks, die somit manchmal auf ihrem eigenen Stapel laufen und manchmal auf dem geteilten Stapel werden Semi Extended Tasks (SET) genannt.

Um somit herauszufinden wann Semi Extended Tasks (SETs) auf dem geteilten Stapel laufen können, ist eine statische Analysephase notwending. Mit dieser Analysephase wird als erstes festgestellt welche Stellen den Stapel wechseln können. Anschließend werden diese Stellen einem

modifizierten Übersetzer mitgeteilt, welcher den eigentlichen Stapelwechsel in den Instruktionscode SET setzt. Schließlich waren noch einige Modizierungen am Wechsel der Ausführungseinheiten notwendig, da nun alle Ausführungseinheiten potentiell auf dem geteilten Stapel laufen können.

Im Folgenden werden somit zu erst die notwendigen Grundlagen erklärt, die später für die Implementierung notwendig sind. Dies umfasst Funktionsaufrufe und die Auswirkungen derer auf den Stapel, sowie einen generellen Überblick über OSEK und warum bisher lediglich Basic Tasks auf dem selben Stapel laufen und nicht auch Extended Tasks.

Anschließend wird die eigentliche Implementierung von SETs vorgestellt, welche in dOSEK, einem OSEK Betriebssystem, vorgenommen wurde. *dOSEK* bietet dabei bereits die nötigen Modifizierungen um einen geteilten Stapelwechsel für Basic Tasks zu unterstützen.

2 GRUNDLAGEN

Im Folgenden erkläre ich die Grundlagen, welche zum Verständnis dieser Arbeit benötigt werden. Dies umfasst den Ablauf eines Funktionsaufrufs auf einer IA32-Maschine, da dieser später verändert wird um SETs zu implementieren. Des weiteren werden die Arten der OSEK Tasks sowie deren Zustände erklärt.

2.1 Konventionen beim Funktionsaufruf

Im folgenden Abschnitt soll es um Funktionsaufrufe und die dazugehörigen Aufrufkonventionen gehen. Diese werde ich erst in einem allgemeinen abstrakten Fall erklären. Anschließend wird das Konzept an einem konkreten Beispiel an der IA32 Architektur noch einmal vorgeführt.

2.1.1 Generelles Konzept

Jedes Programm hat eine Menge an Funktionen. Diese Funktionen sind, solange das Programm nicht läuft, zuerst nur statische Instruktionsblöcke innerhalb eines Programms. Diese Instruktionsblöcke legen den Ablaufplan der Funktion fest. Sollte nun das Programm gestartet werden, ist es Aufgabe einer Konvention, in diesem Fall des *Application Binary Interface* (ABI), festzulegen, welche Funktion als Erstes startet. Eine ABI spezifiziert wie Maschinencode auszusehen hat, damit verschiedene Programme und Funktionen von unterschiedlichen Übersetzern sich gegenseitig aufrufen können. (Siehe [Gcc] Kapitel 9) Da ich hier lediglich das allgemeine Konzept von Funktionsaufrufen erkläre, gehe ich auf diese spezielle ABI in diesem Abschnitt nicht weiter ein. Wenn nun eine Funktion läuft, so nenne ich dies im Folgenden eine *Funktionsinstanz*. Eine Funktionsinstanz führt den Instruktionsblock der Funktion aus. Dabei wird auf Systemresourcen zugegriffen. Dabei gehe ich davon aus, dass es grundsätzlich immer zwei Arten an Instruktionen gibt:

1. arithmetische Operationen auf Registern
2. Lade-/Speicheroperationen mit Zugriff auf den Speicher

Dies hat den Hintergrund, dass die meisten Prozessoren nur eine relative geringe Menge an Allzweckregistern hat. Jeder IA32 Prozessor hat beispielsweise 8 (Siehe [Ia3] Volume 1, Kapitel 3.4.1),

```
1  Programm P={
2    _start = f;
3    Funktion f={
4      var a = 42;
5      g();
6      return a;
7    }
8    Funktion g={
9      return 1337;
10   }
11 }
```

(a) Pseudocode Beispiel für ein Programm mit zwei Funktionen f und g. Das Symbol '_start' legt fest, welche Funktion beim Starten des Programms ausgeführt werden soll. Funktion f hat drei Variablen. Von diesen Variablen darf Variable 'a' während die von g erzeugte Instanz <g> läuft, nicht verloren gehen. Funktion g gibt eine Konstante zurück.

Programmzeile	Funktionsinstanz	verwendete Register	verwendeter Speicher	Speicherinhalt
3	<f>	-	mem(<f>)	mem(<f>)={}; mem(<g>)={};
4	<f>	reg1:a	mem(<f>)	mem(<f>)={}; mem(<g>)={};
5	<f>	-	mem(<f>)	mem(<f>)={reg1:a, PZ:6}; mem(<g>)={};
8	<g>	-	mem(<g>)	mem(<f>)={reg1:a, PZ:6}; mem(<g>)={};
9	<g>	reg1:1337	mem(<g>)	mem(<f>)={reg1:a, PZ:6}; mem(<g>)={};
6	<f>	reg1:a	mem(<f>)	mem(<f>)={}; mem(<g>)={};

(b) Ausführung des Programms aus (a) auf einer abstrakten Maschine. Die in diesem Beispiel verwendete Konvention sieht vor, dass Rückgabewerte einer Funktion immer in Register 'reg1' liegen. Bei einem Funktionsaufruf wird die als nächste auszuführende Programmzeile in den eigenen Speicher geschrieben wird. Die Konvention sieht außerdem vor, dass jeder Funktion ein eigener Speicherbereich zugewiesen wird. Zu sehen sind in jeder Zeile:

- Die aktuelle Programmzeile aus dem Beispielprogramm aus a.
- Die aktuell laufende Funktionsinstanz <x> der Funktion x.
- Die verwendeten Register und Speicher, welche zusammen den Funktionskontext bilden. Da jede Funktion ihren eigenen Speicherbereich hat, muss der Inhalt dessen hier nicht gesichert werden.
- Zuletzt noch der gesamte Speicherinhalt.

Folgendes passiert in den einzelnen Zeilen der Tabelle:

1. In Programmzeile drei wird die Funktion bisher nur gestartet. In diesem einfachen Beispiel steht ihr somit bisher nur ein Speicherbereich zur Verfügung und mehr nicht.
2. Anschließend wird 'reg1' der Wert der Variablen 'a' zugewiesen.
3. Da in Programmzeile fünf die Funktion 'g' aufgerufen wird muss nun der Kontext der Funktionsinstanz gesichert werden, indem der Wert des Registers reg1 in den Speicher geschrieben wird, und zusätzlich die Adresse der nächsten Programmzeile in den Speicher gelegt werden.
4. Nun startet Funktionsinstanz <g>
5. und legt ihren konstanten Rückgabewert in das Register 'reg1'. Beim beenden der Funktionsinstanz <g> weiß die Maschine welche Instanz vorher lief, nähmlich <f>, schaut in den Speicher der Instanz und weiß somit welche Programmzeile als nächstes kommt.
6. Die Instanz <f> läuft nun wieder und kann ihren Rückgabewert in Register 'reg1' legen

Abbildung 2.1 – Beispiel für Funktionsaufrufkonventionen

und hardwarebedingt meist nur auf diesen Registern arithmetische Operationen ausgeführt werden können. (Siehe beispielsweise [Ia3] Volume 2, Kapitel 3.2: Bei der "ADD"-Instruktion muss mindestens einer der Operanden ein Register oder eine Zahl sein. Es dürfen aber nicht beide Operanden aus dem Speicher kommen.) Die Werte, die während einer Funktionsinstanz in den Registern und im *verwendetem* Speicher stehen bezeichne ich als *Funktionskontext*.

Sollte nun eine Funktionsinstanz die eigene Funktion oder eine andere Funktion aufrufen, so darf der Funktionskontext der aktuell laufenden Instanz nicht verloren gehen. Damit das nicht passiert, wird jeder Funktionsinstanz ein bestimmter Speicherbereich zugeordnet und nur in diesen Speicherbereich darf die Funktionsinstanz schreiben. Damit können vor einem Funktionsaufruf die Register in den Speicher gesichert werden und nachdem die aufgerufene Funktion zurückgekehrt ist diese wiederhergestellt werden. Für ein komplettes Beispiel siehe Abbildung 2.1 (a) und (b). So würden sich auch Parameterübergaben und Rückgabewertübergaben realisieren lassen, dies sieht allerdings auf der IA32 Architektur unter Verwendung eines gcc Übersetzers anders aus. Wie sich das hier vorgestellte Konzept bei Funktionsaufrufen von einer echten Implementierung unterscheidet, zeige ich im nächsten Abschnitt.

2.1.2 Konkretes Beispiel bei Verwendung des gcc für die IA32 Architektur

Da wir nun generell verstehen wie der Kontext einer Funktionsinstanz gesichert werden kann, zeige ich im Folgenden wie dies bei Verwendung des gcc für IA32 Prozessoren geschieht.

Das Application Binary Interface (ABI) spezifiziert Laufzeitkonventionen. Wenn ein Programm diese Laufzeitkonventionen erfüllt, dann hält es sich an diese ABI. Ziel der ABI ist es unter anderem Funktionsaufrufkonventionen zu beschreiben, um Kompatibilität zu gewährleisten. Diese legen fest, welcher Teil des Funktionskontext wann gesichert werden muss, sowie Parameter und Rückgabewerte übergeben werden. [Gcc] Dadurch wird sichergestellt, dass eine Anwendung A auf eine Bibliothek B zugreifen kann obwohl diese mit verschiedenen Übersetzern übersetzt worden sind. Natürlich müssen A und B für die selbe Hardware-Architektur übersetzt worden sein.

Da später die Implementierung für Prozessoren der IA32 Architektur erfolgt unter Verwendung des LLVM Kompilierers, welcher sich an die selbe ABI und Aufrufkonventionen wie der GCC Kompilierer hält, nenne ich die verwendete Kombination aus IA32 Architektur, ABI und zugehöriger Aufrufkonvention kurzerhand GNU-ABI. Diese benutzt die *C declaration* Aufrufkonvention zusammen mit der System V ABI. Siehe [Fog15] Seite 17 und Seite 56.

Damit eine Funktionsinstanz ihre Register sichern kann, benötigt sie zuerst ihren eigenen Speicherbereich. Dieser sollte dynamisch zur Laufzeit festgelegt werden, da nur so sichergestellt werden kann, dass zwei Funktionsinstanzen der selben Funktion nicht den selben Speicherbereich haben.

Die GNU-ABI benutzt hierfür die Datenstruktur *Stapel*. Auf die Datenstruktur Stapel können Werte abgelegt werden, wodurch der Stapel wächst. Das zuletzt abgelegte Element eines Stapels kann wiederum eingelesen und herunter genommen werden. Innerhalb dieses Dokuments wächst ein Stapel von *oben* nach *unten*. Ein Stapel wird direkt von der IA32 Architektur unterstützt. Dort ist er ein bestimmter Speicherbereich. Das Ende des Stapels, also der zuletzt abgelegte Wert, wird durch

einen Stapelzeiger markiert. Initial befindet sich der Stapelzeiger auf der größten Speicheradresse des Speicherbereichs und wandert beim ablegen von neuen Werten von der größten Speicheradresse zu kleineren Speicheradressen. Dieser Stapelzeiger ist auf der IA32 Architektur eines der acht Allzweckregister, nämlich das Register *%esp*. ([Sys] Seite 36/37.)

Damit nun eine Funktion ihren Kontext sichern kann, legt sie einfach ihre in Verwendung befindlichen Register auf den Stapel. Danach kann eine andere Funktion aufgerufen werden. Diese darf selber natürlich auch Werte auf den Stapel legen, allerdings nicht die Werte der vorherigen Funktion herunternehmen oder verändern.

Dies weicht allerdings noch leicht vom Kontextsichern unter Berücksichtigung der GNU-ABI ab. So wird nur ein Teil der in Verwendung befindlichen Register von der aufrufenden Funktion gesichert, die *caller save* Register. Die andern Register müssen bei Bedarf von der aufgerufenen Funktion gesichert werden. Im englischen sind dies die *callee save* Register. Siehe [Fog15] Seite 10/11. Ein solch bisher noch nicht fertiger Stack ist in Abbildung 2.2 zu sehen.

Ich habe oben bereits erwähnt, dass zu den Funktionsaufrufkonventionen auch gehört, wie Parameter und Rückgabewerte übergeben werden, dies geschieht ebenfalls auf dem Stapel. Dabei sieht die GNU-ABI vor, dass zuerst die *caller save* Register auf den Stapel gelegt werden und anschließend die Parameter.

Erwähnenswert ist noch, dass eine Funktion im allgemeinen nicht weiß, wer sie aufgerufen hat, dadurch legt der Aufrufer automatisch bei einem Funktionsaufruf die nächste Instruktionsadresse auf

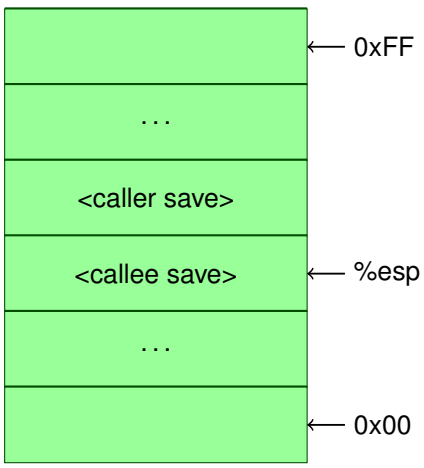


Abbildung 2.2 – Ein bisher nicht fertiger Stack nach einem Funktionsaufruf der GNU-ABI.
Bisher liegen lediglich gesicherte Register auf dem Stack, es fehlen also noch etwaige Paramter und/oder lokale Variablen einer Funktion.
Der %esp ist der Stapelzeiger in der IA32 Architektur und zeigt auf den zuletzt eingetragenen Wert auf dem Stapel. In diesem Fall also eines der *callee save* Register.
Wenn nun ein weiteres Element auf den Stack gelegt werden würde, würde der %esp nach unten in Richtung kleinster Speicheradresse wandern und das Element würde unterhalb von den *callee save* Registern liegen.

den Stapel. Wodurch nun die Funktionsinstanz weiß, wer sie aufgerufen hat. Die Instruktionsadresse liegt in einem Register, somit zählt das Abspeichern der Instruktionsadresse zu einem Spezialfall der *caller save* Register. So legt die *call* Instruktion der IA32 Architektur automatisch die nächste Instruktionsadresse der Funktion auf den Stapel. Siehe Abbildung 2.3

Sollte nun die aufgerufene Funktion selber Werte auf den Stapel legen, so muss sichergestellt werden, dass am Ende der Funktion alle eigenen Werte herunter genommen wurden oder der Stapelzeiger auf die richtige Adresse zeigt. Da beispielsweise die Programmiersprache “C” auch funktionslokale Variablen unterstützt, die nur innerhalb einer Funktionsinstanz zur Verfügung stehen, bietet es sich an diese ebenfalls auf den Stapel zu legen. Nun muss allerdings die Position des Stapelzeigers zu Beginn einer Funktion mit der Position des Stapelzeigers am Ende dieser Funktion übereinstimmen, damit der Funktionskontext des Aufrufers wieder korrekt hergestellt werden kann. Dies wird auf der IA32 Architektur dadurch realisiert, dass zu Beginn einer Funktion der aktuelle Stapelzeiger in ein dafür vorgesehenes Register geschoben wird, nämlich in das %ebp Register, siehe [Sys, Seite 37] Seite 37. Damit kann am Ende der Funktion der Stapelzeiger auf den Wert den er am

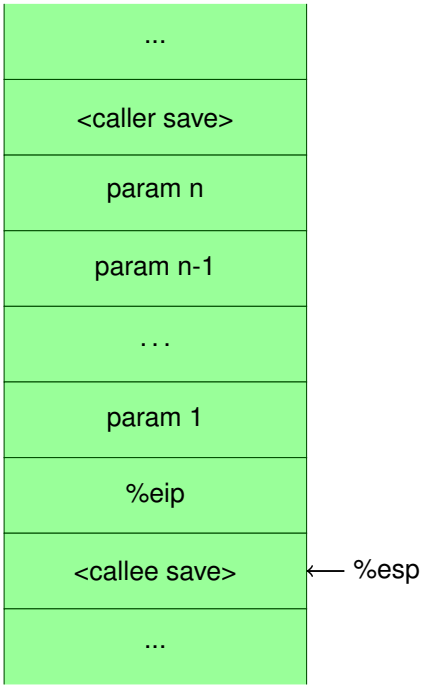


Abbildung 2.3 – Ein im Vergleich zu Abbildung 2.2 leicht erweiterter Stapel nach einem Funktionsaufruf der GNU-ABI.
Hinzugekommen sind Funktionsparameter und der Instruktionszeiger des Aufrufers %eip.
Wobei das %eip Register streng genommen auch einfach nur zu den *caller save* Registern zählt, muss beachtet werden, dass zwischen der Sicherung des %eip und den restlichen *caller save* Registern die Paramter der aufzurufenden Funktion liegen.
Wie hier außerdem zu sehen ist werden die Parameter in rückwärtiger Reihenfolge auf den Stapel gelegt, dies liegt an der *C declaration* Aufrufkonvention und darauf wird in eingegangen. Abbildung 2.4

Anfang der Funktion hatte zurückgesetzt werden. Mit diesem Register können nun außerdem alle Parameter als auch funktionslokalen Variablen unabhängig von der aktuellen Stapelzeigerposition angesprochen werden. Dieser Bereich auf dem Stapel, der zu einer bestimmten Funktionsinstanz gehört wird als Stapelrahmen bezeichnet. Siehe Abbildung 2.4

Nun liegen unterhalb dieser Adresse die Werte der *caller save* Register, danach die Parameter für die Funktion sowie die Instruktionsadresse.

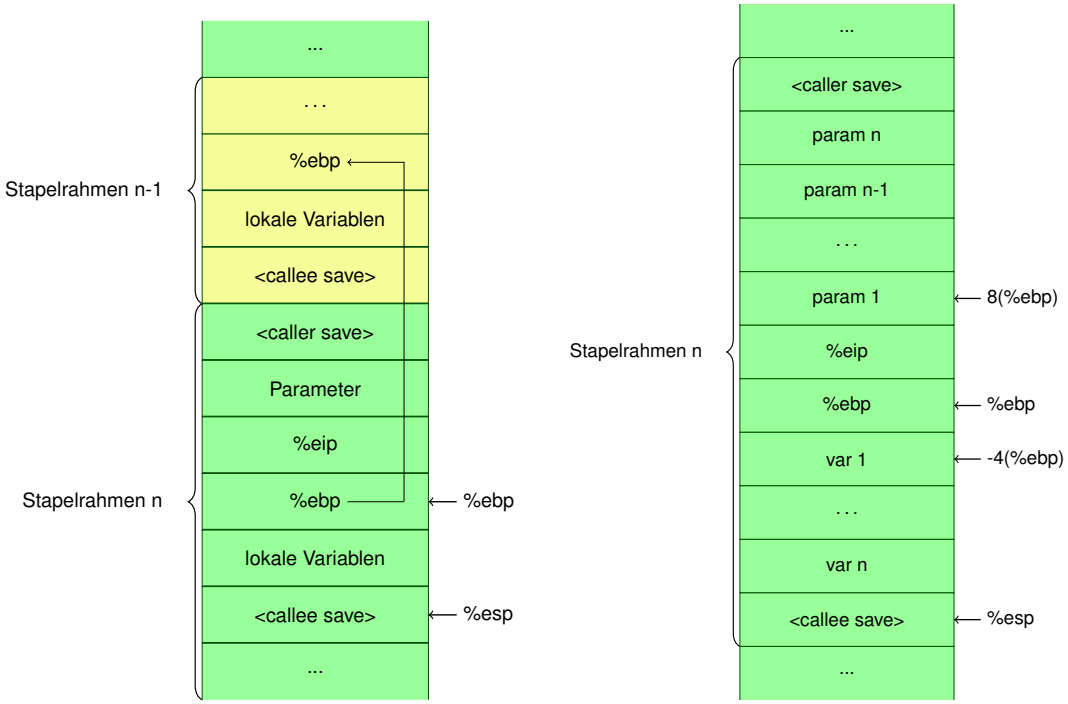


Abbildung 2.4 – Ein Stapelrahmen nach der GNU-ABI.
In grün der aktuelle Stapelrahmen, in gelb der Stapelrahmen der vorherigen Funktion.
Im Vergleich zu Abbildung 2.3 sind die lokalen Variablen der Funktion dazugekommen und ein Zeiger auf den aktuellen Stapelrahmen, wofür das Register %ebp verwendet wird.
An exakt der Stelle des aktuellen Stapelrahmenzeigers liegt der Stapelrahmenzeiger der vorhergehenden Funktion.
Mit Hilfe des %ebp und einem bekannten konstanten Offset kann nun auf die Parameter und die lokalen Variablen zugegriffen werden.
Somit liegt die erste Variable 'var 1' an der Stelle im Speicher %ebp-4, bzw. in AT&T Assembler Syntax '-4(%ebp)'. Bei Parametern geschieht dies äquivalent, aber mit positivem Offset und unter Berücksichtigung der Tatsache, dass zwischen den Parametern und der Stelle auf die der %ebp zeigt noch die gesicherte Rücksprungadresse liegt.
Dadurch, dass die Parameter in rückwärtiger Reihenfolge auf den Stapel gelegt worden sind, kann nun der erste Parameter der Funktion mit dem kleinsten Offset im Vergleich zum %ebp angesprochen werden. Dies hat den Vorteil, dass bei variabler Parameterlänge die Anzahl der Parameter dem ersten Parameter entnommen werden kann, siehe [Fog15] Seite 18.

Wir wissen nun was bei einem Funktionsaufruf alles auf Aufruferseite und auf der Seite des Aufgerufenen getan werden muss, damit diese von Funktionen die die GNU-ABI erfüllen aufgerufen werden können, als auch selbst solche Funktionen aufrufen zu können. Im nächsten Unterkapitel erkläre ich was die OSEK-Spezifikation ist und welche Zustände und Zustandsübergänge OSEK Tasks haben.

2.2 OSEK

Bisher haben wir gesehen, dass Programme aus Funktionen bestehen und wie der Zustand einer gerade laufenden Funktion gesichert wird, wenn diese eine andere Funktion aufruft. Nun wird gezeigt, wie mehrere Programme innerhalb eines Betriebssystems laufen, welches nach der OSEK-Spezifikation entwickelt wurde.

OSEK/VDX, im Folgenden kurz OSEK genannt, ist ein gemeinsames Projekt der deutschen als auch der französischen Automobilindustrie. Es ist eine Spezifikation für Einprozessor-Echtzeitbetriebssysteme, siehe [Ose] Seite 2. Ziel dieser Spezifikation ist es unter anderem, dass bestehender Quellcode möglichst leicht von einer OSEK Implementierung zu einer anderen Implementierung portiert werden kann. Dazu werden verschiedene Betriebssystemobjekte spezifiziert, wie leichtgewichtige Prozesse, Events und Interruptbehandlungen, und eine Application Programming Interface (API) mit entsprechenden API Funktionen für die Systemobjekte. Bevor wir uns jedoch weiter mit der OSEK-Spezifikation befassen, werde ich erst erklären was leichtgewichtige Prozesse sind. Wie diese in OSEK spezifiziert sind wird anschließend geklärt.

2.2.1 Leichtgewichtige Prozesse

Leichtgewichtige Prozesse, im folgenden kurz 'Faden' genannt, sind Programme, die sich in Ausführung befinden. Sollte ein Programm mehrfach ausgeführt werden so gibt es mehrere Fäden zu diesem Programm. Leichtgewichtige Prozesse teilen sich, im Vergleich zu normalen Prozessen, alle den selben Adressraum. Dies führt dazu, dass es keinen besonderen Schutz gibt um zu verhindern, dass ein Faden den Speicher eines anderen Faden modifiziert. Dafür lassen sich Fadenwechsel schneller realisieren, da der Adressraum nicht umgeschaltet werden muss.

Das Konzept eines Fadens lässt sich mit Zuständen beschreiben:
Sollte der Faden ausgeführt werden so befindet er sich im Zustand *laufend*. Wenn er nicht ausgeführt wird, also nur der gesicherte Maschinencode[1] des Programms ist, so befindet er sich im Zustand *beendet*. Sollte der Faden sich im Zustand 'laufend' befinden, so hat dieser einen Kontext, bestehend aus den verwendeten Registern und dem Speicher. Dieser Kontext wird, äquivalent zu Funktionen, beim Fadenwechsel gesichert.

Anders als bei den Funktionen bekommt jeder Faden seinen eigenen Stapel. Durch diesen eigenen Stapel können Fäden pseudoparallel ablaufen, indem sich ein Faden erst im Zustand 'laufend' befindet, dann seinen Kontext auf dem eigenen Stapel sichert und anschließend ein anderer Faden läuft (Siehe

[1] Maschinencode sind die Instruktionen, die von einem Prozessor ausgeführt werden können

Abbildung 2.5). Der Faden, der so seinen Kontext auf seinem Stapel gesichert hat befindet sich somit weder im Zustand 'laufend' noch im Zustand 'beendet', sondern im Zustand *bereit*.

Da es nun mehrere Fäden gibt, muss das Betriebssystem eine Entscheidung treffen, welcher dieser Fäden läuft. Diese Entscheidung kann dabei auf Basis von Prioritäten getroffen werden. Die Priorität kann beispielsweise beim Starten eines Faden gesetzt werden und das Betriebssystem lässt solange den Faden mit der höchsten Priorität laufen bis dieser beendet ist oder ein anderer Faden eine höhere Priorität hat. In klassischen Betriebssystem wird öfters mehreren Prozessen dieselbe Priorität zugeordnet. Prozesse der selben Priorität können sich abwechselnd verfügbare CPU-Zeit teilen, wie in Abbildung 2.5 gesehen werden kann. Die später verwendete Implementierung des OSEK Betriebssystems erlaubt allerdings, dass jede Prioritätsstufe nur einmal oder gar nicht vorkommen darf, weswegen dieser Fall nicht weiter betrachtet werden muss.

Eine weitere Eigenschaft von Fäden ist, dass sie auf Betriebssystemresourcen warten können. Dies geschieht indem ein spezieller Betriebssystemaufruf ausgeführt wird. Bei diesem Aufruf wird dem Betriebssystem signalisiert auf welche Resource gewartet wird, gleichzeitig wird in den Zustand *wartend* gewechselt. Äquivalent zum Zustand 'bereit' wird dabei der Kontext auf dem Stapel gesichert. Wenn die Resource verfügbar ist wird der Faden vom Zustand 'wartend' in den Zustand 'bereit' überführt und kann fortgesetzt werden, sobald der Faden die höchste Priorität hat. Ein Zustandsdiagramm eines Fadens wie er bis hier erklärt wurde ist in Abbildung 2.6 zu sehen.

Da wir nun das Konzept eines Faden verstehen, werde ich im nächsten Abschnitt die Modifizierung und Einschränkungen vorstellen, die für einen OSEK Tasks nötig sind.

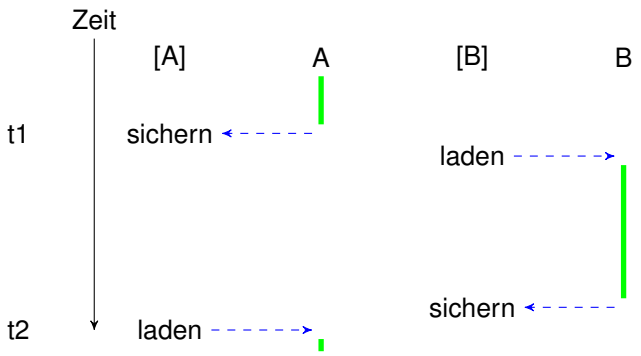


Abbildung 2.5 – Beispielsequenzdiagramm für die Quasiparallelität von zwei Fäden.
Zu sehen sind die Fäden A und B, sowie deren zugesicherter Speicher [A] und [B], als auch die beiden Zeitpunkte t1 und t2.
Als erstes läuft der Faden A. Zum Zeitpunkt t1 soll Task B laufen, dadurch sichert als erstes Faden A seinen Kontext auf seinem Speicher [A]. Anschließend kann Faden B seinen gesicherten Kontext von [B] laden und laufen.
Zum Zeitpunkt t2 wird wieder zum Faden A gewechselt, dadurch muss Task B seinen Kontext sichern und Faden A seinen Kontext laden.
Das Sequenzdiagramm könnte nun wieder von vorne beginnen und dadurch können die beiden Fäden durchgängig laufen.

2.2.2 Tasks in OSEK

Tasks in OSEK sind Fäden wie sie bisher vorgestellt wurden mit einigen Einschränkungen, dazu gehört, dass OSEK Tasks nur von sich selbst beendet werden können und nicht durch das Betriebssystem oder einen anderen Task.

Um diese Einschränkungen fein granular aufzustellen gibt es vier sogenannte *conformance classes*.

1. Basic Conformance Class 1 (BCC1)
2. Basic Conformance Class 2 (BCC2)
3. Extended Conformance Class 1 (ECC1)
4. Extended Conformance Class 2 (ECC2)

Die später verwendete Implementierung des OSEK Betriebssystems unterstützt die *conformance classes BCC1* und *ECC1*, weshalb nur diese im Folgenden berücksichtigt werden. Der größte Unterschied zwischen BCC1 und ECC1 ist, dass es in BCC1 keine Extended Tasks gibt, in ECC1 dementsprechend schon. Einer der Unterschiede zwischen BCC1/ECC1 und BCC2/ECC2 ist, dass es in BCC2/ECC2 mehrere Tasks pro Priorität geben kann, wohingegen es in BCC1/ECC1 nur einen Task pro Priorität gibt. In allen Fällen wird diese Priorität statisch, also zur Übersetzerzeit, festgelegt. (Für eine genauere Einstufung der einzelnen *conformance classes* siehe [Ose, Seite 14])

2.2.2.1 Extended Tasks (ETs)

Das Zustandsdiagramm eines OSEK Extended Task ist äquivalent zu dem eines Fadens, wie in Abbildung 2.6(a) vorgestellt. Die einzige Art an Betriebssystemresourcen auf die Extended Tasks warten können sind OSEK Event Objekte. Ein einzelnes Event gehört dabei *immer* einem oder mehreren Extended Tasks und wird vom Betriebssystem verwaltet (Siehe [Ose] Seite 27). Events können gesetzt (API Funktion: `SetEvent`) und gelöscht (`ClearEvent`) werden. Sowohl Basic als auch Extended Tasks können ein Event setzen, aber nur ein Extended Task, dem das Event gehört, kann das Event klären. Auch nur dieser Extended Task darf auf das Event warten (`WaitEvent`). In ECC1 darf ein ET bis zu 8 Events zugeordnet haben (Siehe [Ose, Seite 14]). Nur wenn das Event geklärt ist kann darauf gewartet werden, ansonsten wird der Task nicht in den Wartezustand überführt, sondern setzt seine Ausführung fort.
Events können zusätzlich von *Interrupt Service Routinen* (ISR) gesetzt werden. ISRs sind dabei die Interruptbehandlungen von OSEK. Ein Interrupt ist ein von außerhalb des System kommendes Signal, wie ein Taster oder ein Sensor. ISRs werden in OSEK in Kategorie 1 und 2 unterteilt. Nur ISRs der Kategorie 2 dürfen dabei Systemaufrufe vornehmen, wie das setzen eines Events oder das Aktivieren eines Task. Dies führt dazu, dass auch nur nach ISRs der Kategorie 2 das Betriebssystem entscheiden muss ob der aktuell laufende Task durch einen höherprioren ersetzt werden muss. ISRs der Kategorie 1 hingegen dürfen nur einfache Funktionen sein. Im Gegensatz zu Tasks muss sich die Hardware um die Prioritäten der ISRs kümmern.

Die zweite Kategorie an Tasks sind Basic Tasks, welche nicht den vollen Funktionsumfang wie Extended Tasks liefern, dafür aber Speicherplatz zur Laufzeit eingespart werden kann.

2.2.2.2 Basic Tasks (BTs)

Basic Tasks besitzen, im Vergleich zu Extended Tasks, keinen Wartezustand. (Siehe [Ose, Seite 18]) Dies ist auch der Grund weshalb sie nicht auf Events warten können. Von den Zustandsübergängen die einen Wartezustand voraussetzen abgesehen, sind die Zustandsübergänge von Basic Tasks identisch mit denen von Fäden, mit den Einschränkungen wie sie bei Extended Tasks vorgestellt wurden. Für ein Zustandsdiagramm eines Basic Task siehe Abbildung 2.7. Einer der Vorteile, die sich daraus ergeben, dass Basic Tasks keinen Wartezustand haben, ist, dass sie einen verringerten Speicher Verbrauch, im Vergleich zu Extended Tasks, haben. (Siehe [Ose, Seite 18, Abschnitt 4.2.3])

2.2.3 Statische Konfigurierbarkeit in OSEK

Die *OSEK Implementation Language* (OIL) ist eine Sprache um eine OSEK Betriebssystemimplementierung für eine bestimmte CPU statisch zu konfigurieren. (Siehe [Oil] Seite 7 Abs. 1)

Dies geschieht indem in einer OIL Datei alle OSEK System Objekte, wie Tasks, Events und ISRs, aufgelistet und spezifiziert werden. Die Spezifikation eines Tasks wäre beispielsweise die Priorität oder zugeordnete Events. Für einen beispielhaften Inhalt einer solchen OIL Datei siehe Listing 2.1.

```
1    TASK T1 {
2        PRIORITY = 4;
3        AUTOSTART = TRUE;
4        EVENT = E1;
5    };
6    EVENT E1 {
7        MASK = AUTO;
8    };
```

Listing 2.1 – Beispiel für den Inhalt einer OIL Datei:
Es gibt einen Task T1 und ein Event E1.
Der Task hat eine statische Priorität der Größe 4 und wird direkt beim Betriebssystemstart in den Zustand 'bereit' versetzt. Da dem Task ein Event gehört ist es ein Extended Task.
Das Event hat eine Maske bestehend aus Eventname und Eventeigentümer T1, da diese Informationen aus dem Rest der OIL Datei eindeutig sind kann diese automatisch gesetzt werden.

2.2.4 Verwendete OSEK Implementierung

Die in dieser Arbeit verwendete Implementierung eines OSEK Betriebssystems ist *dOSEK*. Bisherige Ziele der dOSEK Implementierung bestanden auf Zuverlässigkeit [HDL13] und Robustheit gegenüber Bitfehler die zur Laufzeit auftreten können [Hof+15].

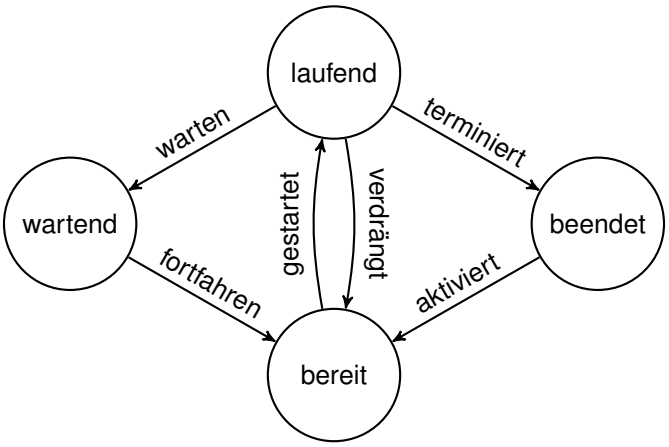
Um Speicherplatz einzusparen wurde bisher ein Stapel für *alle* Basic Tasks erstellt und diese haben sich den Stapel geteilt. Dies ist möglich, da Basic Tasks nicht warten können, es pro Priorität nur einen Task gibt und Tasks nur durch sich selbst beendet werden können. Dadurch gibt es eine strikte, nur durch die Prioritäten gegebene, Abarbeitungsreihenfolge für Basic Tasks. Somit muss sich nur noch die aktuelle Stapelposition vermerkt werden, damit Werte nicht überschrieben werden, für einen solchen beispielhaften Ablauf mehrerer Basic Tasks siehe Abbildung 2.8.

Bei einem Taskwechsel werden alle Register bis auf den Stapelzeiger auf den Stapel des Tasks gelegt. Der Stapelzeiger wird in einer Task lokalen Variable gesichert. Diese Taskvariable wird statisch für jeden Task angelegt. Die Stapelposition des als nächstes laufenden Tasks kann dann aus dieser Task lokalen Variable eingelesen werden und die gesicherten Registerwerte vom Stapel geladen werden.

2.3 Zusammenfassung

Application Binary Interfaces spezifizieren Aufrufkonventionen. Diese dienen dazu, dass der generierte Binärcode verschiedener Übersetzer sich gegenseitig aufrufen kann. Dabei muss primär auf den richtigen Auf- und Abbau des Stapels geachtet werden.
Die OSEK Betriebssystemspezifikation spezifiziert verschiedene Objekte, dazu gehören Tasks, Events und ISRs. Solange nur die *conformance classes* BCC1 oder ECC1 betrachtet werden, können alle Basic Tasks auf dem selben Stapel laufen. Extended Tasks können auf Events warten und Events wiederum von ISRs der Kategorie 2 oder Tasks ausgelöst werden.

Im folgenden widme ich mich nun der Implementierung von *Semi Extended Tasks* in dOSEK mit Modifizierung des LLVM Übersetzers.



(a) Zustände:
laufend: Der aktuelle Faden ist der einzige auf dem Prozessor laufende Faden.
beendet: Der Faden ist beendet weil er bis zu seiner Beendigung komplett durchgelaufen ist.
bereit: Der Faden kann laufen, sobald er die höchste Priorität aller bereiten Fäden hat. Dann wird er vom Betriebssystem gestartet.
wartend: Der Faden wartet auf bestimmte Betriebssystemresourcen und wird vom Betriebssystem in den Zustand 'bereit' überführt sobald diese verfügbar sind.

Zustandsübergang	Vorheriger Zustand	Folgender Zustand	Beschreibung
aktiviert	beendet	bereit	Der Faden wurde durch einen Systemaufruf in den Zustand 'bereit' versetzt
gestartet	bereit	laufend	Der Faden hat die höchste Priorität aller lauffähigen Fäden und wurde deswegen vom Betriebssystem in den Zustand 'laufend' versetzt.
verdrängt	laufend	bereit	Ein anderer Faden hat eine höhere Priorität als der aktuelle Task gehabt. Deswegen wurde der Kontext des aktuellen Task auf dem Stapel gesichert.
terminiert	laufend	beendet	Der aktuelle Faden hat sich beendet oder wurde beendet.
warten	laufend	wartend	Aufgrund eines Systemaufrufs wartet der Faden auf eine Betriebssystemresource.
fortfahren	wartend	bereit	Die Resource auf die gewartet wurde ist frei.

(b) Die Zustandsübergänge eines Faden.

Abbildung 2.6 – Zustandsdiagramm und Zustandsübergänge eines Faden, zu Teilen entnommen aus [Ose] Seite 17
In (a) sind die Zustände abgebildet und beschrieben, während in (b) die Zustandsübergänge beschrieben werden.

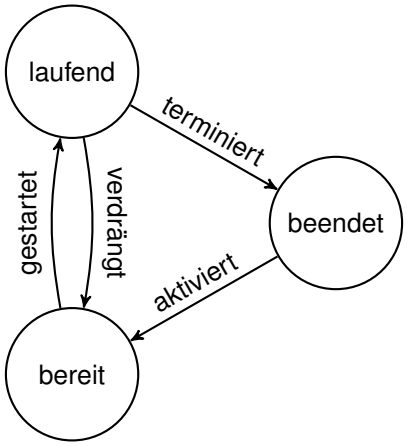


Abbildung 2.7 – Zustandsdiagramm eines Basic Task, entnommen aus [Ose] Seite 18
Zustände:
laufend: Der aktuelle Task ist der einzige auf dem Prozessor laufende Task.
beendet: Der Task ist beendet weil er bis zu seiner Beendigung komplett durchgelaufen ist.
bereit: Der Task kann laufen, sobald er die höchste Priorität aller laufbereiten Tasks hat. Dann wird er vom Betriebssystem gestartet.

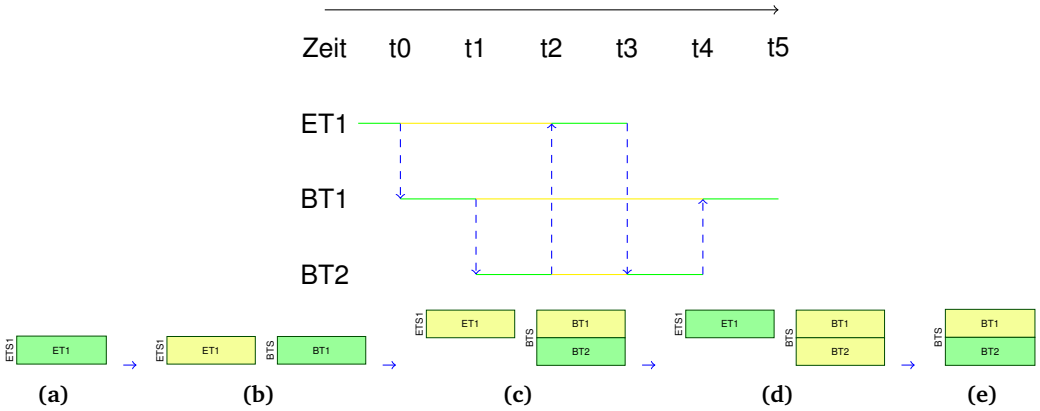


Abbildung 2.8 – Beispielhafter Ablauf für mehrere Basic Tasks, die auf dem selben Stapel laufen können
Zu sehen ist oben das Sequenzdiagramm für die Ausführung der Tasks Extended Task 1 (ET1), Basic Task 1 (BT1) und Basic Task 2 (BT2). Zusätzlich die Umschaltpunkte t0 - t4, an denen zwischen verschiedenen Tasks gewechselt wird, so wie das Ende des BT1 zum Zeitpunkt t5.
Unten sind die jeweiligen Stapel der Tasks zu sehen: ETS1 für den Extended Task ET1 und der BTS für die Basic Tasks BT1 und BT2. Grün bedeutet, dass ein Task gerade läuft, bzw. der zugehörige Speicherbereich des Stapels gerade in Verwendung ist. Gelb bedeutet, dass ein Task verdrängt wurde oder auf ein Event wartet und dementsprechend ist der Kontext des Tasks auf seinem zugehörigen Speicherbereich des Stapels gesichert.
Die Übergänge, die mit blauen Pfeilen markiert sind, sind Taskumschaltpunkte des Betriebssystems.
Für die Prioritäten, der Tasks gilt:
Priorität(BT1) < Priorit(BT2) < Priorität(ET1)
Nicht in der Abbildung abgebildet ist das Event E1, welches dem Extended Task ET1 zugeordnet ist und zu Beginn nicht gesetzt ist.
(a) ET1 läuft auf seinem Stapel, dem ETS1
(a) → (b) bzw. t0 ET1 aktiviert den Task BT1 und wartet anschließend auf das Event E1. Daraufhin läuft BT1 auf dem BTS.
(b) → (c) bzw. t1 BT1 aktiviert BT2. BT2 hat eine höhere Priorität als BT1 und läuft somit als nächstes. Da außerdem Basic Tasks nicht warten können, kann BT2 auf dem selben Stapel laufen wie BT1, da BT2 bis zu seiner selbständigen Beendigung durchläuft.
(c) → (d) bzw. t2 BT2 setzt das Event E1, womit ET1 nun in den Zustand *bereit* wechselt. Da die Priorität von ET1 höher ist, als die von BT2, läuft somit ET1.
(d) → (e) bzw. t3 ET1 beendet sich, womit wieder BT2 läuft.
t4 BT2 beendet sich, womit BT1 läuft.
t5 BT1 beendet sich.

3 ARCHITEKTUR

Bisher haben wir die Grundlagen über einen Funktionsaufruf auf der GNU-ABI gelernt. Dazu gehört wie der Kontext einer Funktionsinstanz gesichert und wiederhergestellt wird, wenn diese eine andere Funktion oder sich selbst aufruft. Zudem wissen wir nun was ein OSEK Task ist, was für Unterschiede es zwischen Basic und Extended Tasks gibt sowie die entsprechende Zustände und Zustandsübergänge.

Nun zeige ich was für Optimierungen möglich sind um den Speicherverbrauch eines Extended Task zu minimieren. Dabei werde ich als erstes das generelle Konzept vorstellen, welches vorsieht, dass es bestimmte Stellen eines Extended Task gibt, die auf dem gemeinsam genutzten Basic Task Stapel (BTS) laufen können. Anschließend wird gezeigt, wie bei einem OSEK Extended Task statisch ermittelt werden kann welche Stellen dies sind.

3.1 Das Semi Extended Task Konzept

SETs sind ETs, die, mit dem Ziel den Speicherverbrauch des gesamten Systems zu minimieren, an gewissen Umschaltpunkten von ihrem eigenen Stapel auf den Basic Task Stapel (*BTS*) wechseln. Das heißt, es müssen nun genau die Stellen gefunden werden an denen dieser Wechsel möglich ist.

Wie wir vorhin gesehen haben, ist dOSEK ein rein Prioritäten gesteuertes Betriebssystem, wodurch immer der höchstpriore Task läuft, *außer* dieser wartet gerade auf ein Event. Dieses Warten ist auch primär der Grund warum ein ET seinen eigenen Stapel benötigt.

Sollten ETs auch auf dem BTS laufen und beispielsweise ein BT niedriger Priorität einen ET höherer Priorität starten, so läuft der ET auf dem BTS auf einer niedrigeren Speicheradresse als der BT. Somit verändert der ET den gesicherten Kontext des vorherlaufenden BT nicht. Wenn nun allerdings der ET wartet, so sichert dieser seinen Kontext auf dem BTS und der BT läuft. Dadurch kann der BT bereits durch geringen Stapelspeicherverbrauch in den gesicherten Kontext des ET schreiben, da lediglich die Allzweckregister beim Fadenwechsel auf dem Stapel gesichert werden und somit der Abstand zum gesicherten Kontext des Extended Task sehr gering ist (siehe auch Abbildung 3.1).

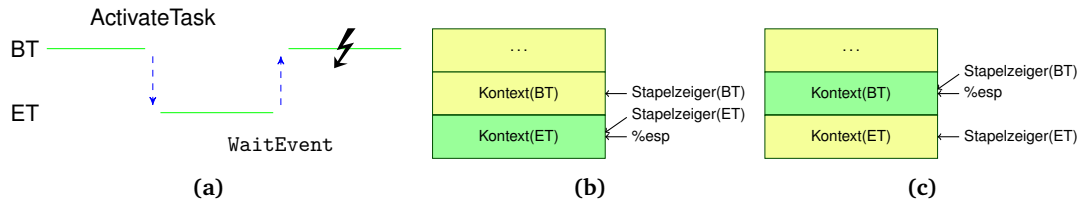


Abbildung 3.1 – Probleme, wenn Extended Tasks auf dem BTS laufen:
(a) Zu sehen ist ein Sequenzdiagramm. Ein BT startet einen ET höherer Priorität. Daraufhin läuft der ET bis dieser einen `WaitEvent` Aufruf macht und somit wartet. Anschließend läuft wieder der BT. Da der ET jedoch auf dem selben Stapel lief, überschreibt er nun den gesicherten Kontext des ET (siehe (c)).
(b) Zustand des Stapel nachdem der ET aktiviert wurde. Noch geht alles gut.
(c) Nun wartet der ET und der BT läuft wieder. Der %esp wächst nach unten ⇒ der BT schreibt in den gesicherten Kontext des ET.

Also können ETs an den Stellen an denen sie einen `WaitEvent` Aufruf haben nicht auf dem BTS laufen, da sie sonst Gefahr laufen, dass ihr gesicherter Kontext verändert wird. Vor dem `WaitEvent` Aufruf kann der ET auf dem BTS laufen, solange sichergestellt ist, dass unmittelbar vor dem `WaitEvent` Aufruf der gesamte Kontext nur auf dem eigenen Stapel und nicht auf dem BTS liegt. Nach dem `WaitEvent` Aufruf kann wieder auf den BTS gewechselt werden, solange vor dem nächsten `WaitEvent` Aufruf wieder der gesammte Kontext auf dem eigenen Stapel liegt (siehe Abbildung 3.2).

Da Funktionsaufrufe die Eigenschaft haben, dass sie beim zurückkehren den Stapel auf den Zustand vor dem Aufruf zurücksetzen (siehe Abbildung 4.2) bietet es sich an, diese so zu modifizieren, dass beim Funktionsaufruf der Stapel gewechselt wird. Wie genau dies möglich ist wird im Folgenden erklärt.

3.2 Die Stellen der Umschaltpunkte

Um bei einem Funktionsaufruf den Stapel zu wechseln gibt es grundsätzlich die Möglichkeit dies direkt auf callee Seite zu tun oder caller Seite.

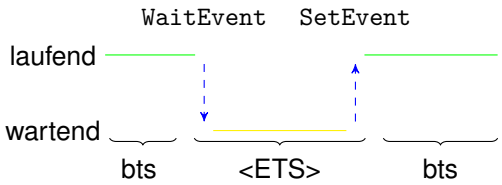


Abbildung 3.2 – Mögliche Stellen für den Stapelwechsel.
Wie in Abbildung 3.1 gesehen werden kann, existiert immer dann ein Problem, wenn ein ET auf dem BTS wartet. Darum muss kann ein ET vor dem `WaitEvent` Aufruf auf dem BTS laufen, muss aber vor dem `WaitEvent` Aufruf seinen gesamten Kontext auf seine ETS Instanz, <ETS> legen. Nach dem `WaitEvent` Aufruf kann der ET wieder auf dem BTS laufen.

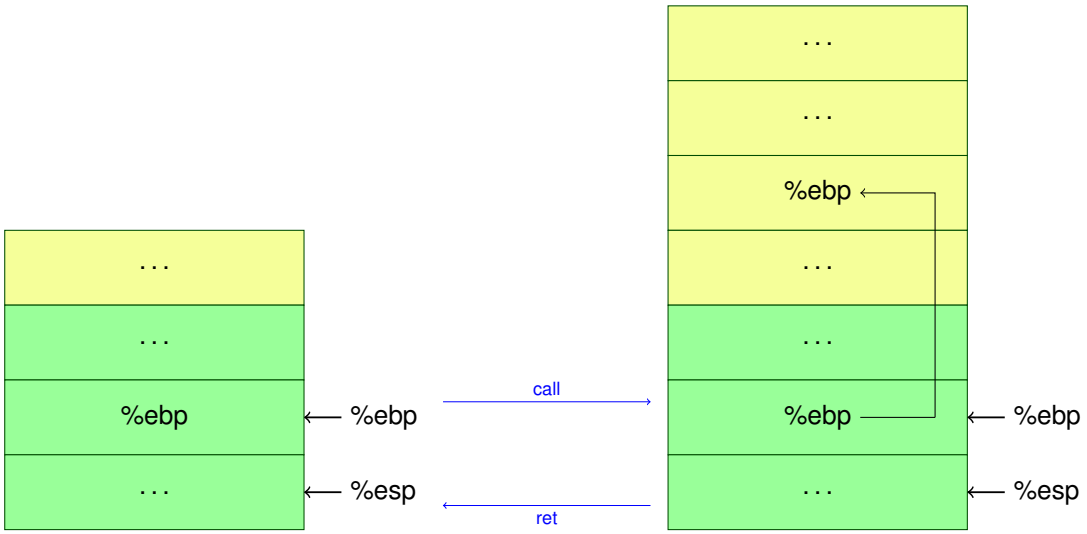


Abbildung 3.3 – Auf- und Abbau des Stapels bei Funktionsaufruf und Funktionsende

Wenn die callee Seite verändert wird bedeutet dies, dass die Funktion die auf dem BTS laufen soll selbstständig den Stapel wechselt, somit wird die Funktion selbst modifiziert. Dadurch sind keinerlei Modifizierungen beim caller nötig. Auf caller Seite hingegen muss jeder einzelne Aufruf einer Funktion, die auf dem BTS laufen soll, angepasst werden. Wenn man die callee Seite implementiert bietet dies den Vorteil, dass nur einmal pro Funktion der Instruktionscode verändert werden muss, anstatt bei jedem Funktionsaufruf der Funktion.

Die Möglichkeit den Stapel zu wechseln ist in erster Linie davon abhängig ob eine Funktion einen `WaitEvent` Aufruf hat oder nicht. Dies ist somit statisch bestimmbar und dadurch für eine Funktion konstant. Dadurch, und da der, in dieser Arbeit modifizierte, Übersetzer lediglich die callee Seite implementiert, werden im Folgenden als erstes die Übersetzermodifizierungen für die callee Seite erklärt.

3.2.1 Implementierung auf callee Seite

Wie in Abschnitt 2.1.2 gezeigt wurde, werden insbesondere die Funktionsparameter und die Rücksprungadresse immer vom caller auf den Stapel gelegt. Sollte nun der callee den Stapel wechseln, so liegt das Problem vor, dass dieser Teil des Kontext immer noch auf dem alten Stapel liegt. Dadurch wird bei der Implementierung auf callee Seite weiterhin der Stapelrahmenzeiger auf seinen normalen Wert zeigen, also den alten Stapelrahmenzeiger, um so die Funktionsparameter und die Rücksprungadresse lesen zu können.

Zusätzlich kann so am Funktionsende der modifizierte Stapelzeiger wiederhergestellt werden. Damit muss bisher nur der Stapelzeiger am Anfang der Funktion auf einen neuen Wert gesetzt werden. Dieser Wert muss vom Betriebssystem in einem spezielle globale Variable geschrieben werden, damit zur Laufzeit der Wert der globalen Variable in den Stapelzeiger geschrieben werden

kann. Mit dem Stapelzeiger würden sich dann die lokalen Variablen ansprechen lassen. Damit könnte bisher dieser Stapelwechsel mit dem Zusatz einer Maschineninstruktion am Funktionsanfang realisiert werden (siehe Abbildung 4.3).

Nun müssen allerdings alle lokalen Variablen über den Stapelzeiger angesprochen werden, da der Stapelrahmenzeiger auf den alten Stapel zeigt. Dies ist insbesondere dann ein Problem, wenn innerhalb der callee Funktion Felder variabler Länge verwendet werden. Felder variabler Länge werden seit dem C99 Standard unterstützt [2] und sind Felder welche als Länge keine statischen Ausdrücke benötigen sondern auch dynamische, also zur Übersetzerzeit nicht auswertbare Ausdrücke, erlauben (siehe [Fou]). Nun sind die genauen Anfangs- und Endpositionen des Feldes nicht mehr statisch bestimmbar, sondern von der zur Übersetzungszeit nicht bekannten Länge des Feldes und von der Stapelzeigerposition abhängig. Dadurch lassen sich solche Felder schlecht über den Stapelzeiger ansprechen.

Das gleiche Problem besteht, wenn bestimmte Maschineninstruktionen ausgeführt werden, die bestimmte Speicherausrichtungen benötigen, wie SSE-Instruktionen, die einen 16 Byte ausgerichteten Speicherplatz benötigen (siehe [Ia3, Volume 1 Kapitel 10.4.1.1]). Da bei Neuausrichtung des Stapels, auf die geforderte Speicherausrichtung der Maschineninstruktion, eine Lücke unbekannter Länge zwischen dem Stapelrahmenzeiger und den lokalen Variablen entsteht (siehe Abbildung 4.4). Dort wurde dieses Problem gelöst indem ein weiteres Register als Basiszeiger verwendet wird. Dieser Basiszeiger zeigt auf die erste lokale Variable.

Hier wurde das Problem der Felder variabler Länge auf die selbe Weise, unter leichter Modifizierung des bestehende Übersetzer Quellcodes, gelöst.

Zusammenfassend sind also folgende Modifizierungen am Übersetzer nötig um einen Stapelwechsel auf callee Seite zu unterstützen:

1. Sicherstellung, dass der Stapelrahmenzeiger verwendet wird und auf seine normale Stelle zeigt um so die Funktionsparameter lesen zu können und am Funktionsende wieder auf den alten Stapel zu wechseln

[2] Obwohl Felder variabler Länge nicht Teil des C++ Standards sind, so unterstützen einige C++ Übersetzer, wie der g++, diese Erweiterung trotzdem

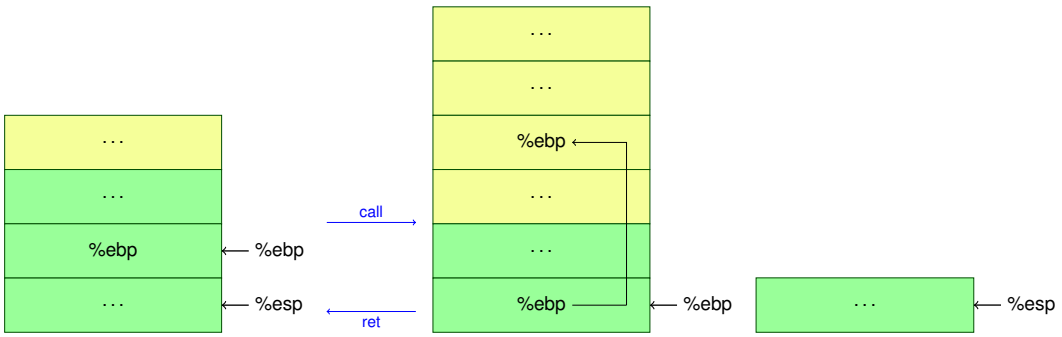


Abbildung 3.4 – Stapelwechsel auf callee Seite indem lediglich der %esp auf einen neuen Stapel gesetzt wird

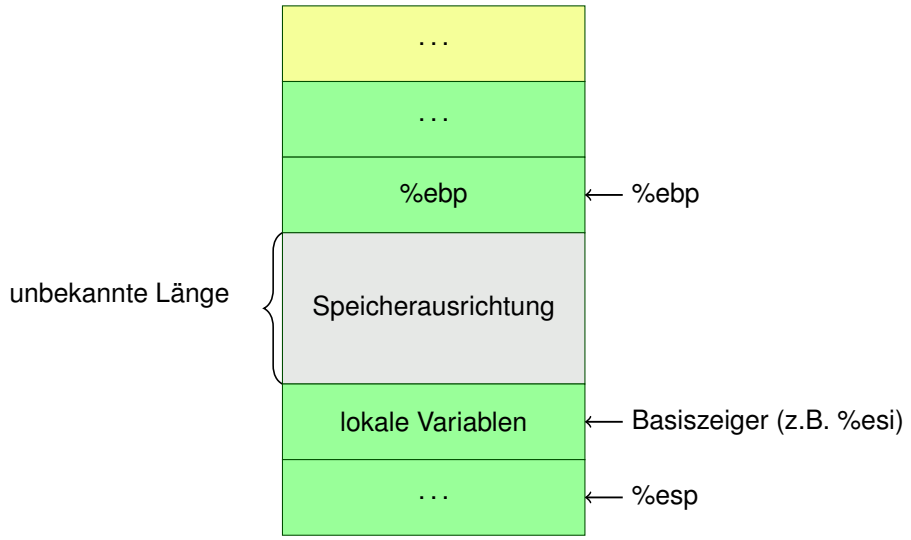


Abbildung 3.5 – Verwendung eines Basiszeigers um Lücken unbekannter Länge zu umgehen. Bei einem Stapelwechsel ist die Differenz zwischen neuer und alter Stapeladresse an sich auch nur eine Lücke unbekannter Länge.

2. Den Stapelzeiger auf den neuen Stapel zeigen lassen (in diesem Fall also das Ende des BTS)

3. Sollten lokale Variablen dynamischer Größe verwendet werden oder sollte es bevorzugt sein die lokalen Variablen im Allgemeinen nicht über den Stapelzeiger anzusprechen, so wird ein Basiszeigerregister benötigt, welches auf die lokalen Variablen zeigt

Diese Modifizierungen wurden am Backend des LLVM Übersetzers eingefügt, welcher von dOSEK für die Übersetung des Quellcodes verwendet wird. Andere Modifizierungen sind hingegen für einen Stapelwechsel auf caller Seite nötig, worauf ich nun weiter eingehen werde.

3.2.2 Implementierung auf caller Seite

Ich habe in Abschnitt 3.2 erwähnt, dass die Auswahl einer Funktion zum Stapelwechsel in erster Linie davon abhängt ob diese Funktion einen `WaitEvent` Aufruf hat. Allerdings gibt es Funktionen die keinen `WaitEvent` Aufruf haben und trotzdem den Stapel nicht wechseln können. Dazu gehören rekursive Funktionen, da diese ansonsten beim Aufruf auf sich selbst, auf den BTS wechseln würden obwohl sie sich schon auf diesem befinden. Genauso können Bibliotheksfunktionen für die kein Quelltext verfügbar ist im nachhinein eher schlecht angepasst werden. Darum ist es sinnvoll auch über einen Stapelwechsel auf caller Seite nachzudenken.

Auf caller Seite muss nun der Stapelwechsel so implementiert werden, dass die aufgerufene Funktion nicht davon beeinflusst wird, dass sie auf einem anderen Stapel läuft als der caller. Das bedeutet, dass bereits die Parameter der aufzurufenden Funktion auf dem BTS liegen müssen. Eine mögliche Stelle den Stapelzeiger auf den BTS zu setzen wäre direkt nachdem die *caller save* Register auf den Stapel gelegt worden sind. Da somit lediglich die Parameter des callee auf dem BTS liegen

bevor die eigentliche *call* Instruktion ausgeführt wird. Der callee kann nun ausgeführt werden ohne, dass dieser beeinflusst wird, da alle Parameter und der restliche Kontext bereits auf dem BTS liegen. Nach dem Beenden des callee sollte der Stapelzeiger wieder auf den eigenen Stapel zurückgesetzt werden, *bevor* die eigenen *caller save* Register vom Stapel heruntergenommen werden.
Um den Stapelzeiger zurückzusetzen sollte ein gesicherter Wert des alten Stapelzeigers auf den neuen Stapel gelegt werden. Dies kann ermöglicht werden, indem eines der *caller save* Register nach dem Sichern der *caller save* Register, dazu verwendet wird den alten Wert des Stapelzeigers so lange zwischenzuspeichern, bis der Stapel gewechselt wurde und nun der gesicherte Wert des Stapelzeigers auf den neuen Stapel gelegt werden kann. Für einen wie hier beschriebenen Stapelaufbau siehe Abbildung 3.6.

Da nun geklärt ist wie ein Stapelwechsel auf callee Seite oder auf caller Seite vorgenommen werden kann, wird im Folgenden gezeigt, welche Funktionen genau für einen Stapelwechsel auf callee bzw. auf caller Seite in Frage kommen.

3.3 Auswahl der Umschaltpunkte

Im vorherigen Abschnitt habe ich gezeigt, wie ein Übersetzer angepasst wurde, so dass bei einem Funktionsaufruf der Stapel gewechselt werden kann. Diesem modifizierten Übersetzer muss nun mitgeteilt werden, welche Funktionen den Stapel wechseln sollen.Dabei gibt es Funktionen, die sowohl auf callee als auch auf caller Seite umgeschaltet werden können. Wenn so ein Fall auftritt sollte immer die callee Implementierung genommen werden, da diese nur einmal pro Funktion zusätzlichen Quellcode einfügt, anstatt bei jedem Funktionsaufruf einer Funktion.

Um herauszufinden, welche Funktionen den Stapel wechseln dürfen, betrachten wir alle Funktionen eines dOSEK Programmes und entfernen dann die Funktionen, die nie den Stapel wechseln dürfen. Anschließend zeige ich, welche Funktionen zusätzlich bei einer callee Implementierung nicht gewählt werden dürfen. Schließlich können noch einige Funktionsaufrufe ausgewählt werden, die für eine etwaige caller Implementierung in Frage kommen würden.

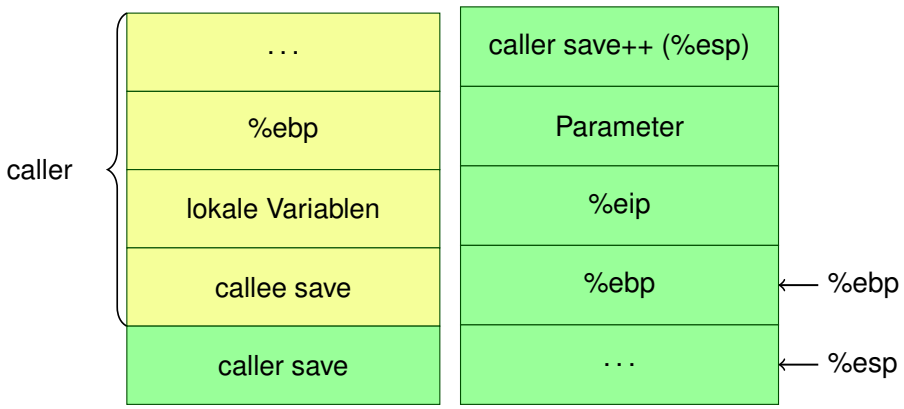


Abbildung 3.6 – Stapelaufbau bei einem möglichen Stapelwechsel auf caller Seite.

3.3.1 Funktionen die den Stapel nie wechseln können

Zu den Funktionen, die nie den Stapel wechseln, gehören zum einen Funktionen, die *nur* von BTs aufgerufen werden, da sich diese bereits auf dem BTS befinden und es somit keinen Sinn macht den Stapel zu wechseln. Des weiteren dürfen einige Betriebssystem spezifische Funktionen den Stapel nicht wechseln, da diese unter anderem selbst den Stapel wechseln um so den Kontextwechsel von einem ET zu einem anderen Task zu realisieren.

Wie in Abschnitt 3.1 gesehen werden konnte, dürfen Funktionen, die einen `WaitEvent` Aufruf beinhalten nicht den Stapel wechseln, da ansonsten in den Kontext des wartenden Extended Task geschrieben werden könnte (siehe Abbildung 3.1). Dies sind Funktionen, die einen *direkten* `WaitEvent` Aufruf haben. Funktionen die einen *indirekten*[3] `WaitEvent` Aufruf haben dürfen ebenfalls nicht den Stapel wechseln, da dort das selbe Problem auftreten kann (siehe Abbildung 3.7).

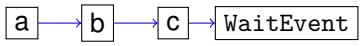


Abbildung 3.7 – Direkter und indirekter `WaitEvent` Aufruf:
Zu sehen sind die Funktionen a,b,c und `WaitEvent`. Die Kanten stellen die Funktionsaufrufe dar. Funktion c ruf direkt `WaitEvent` auf, Funktionen a und b lediglich indirekt. Keine dieser Funktionen darf den Stapel wechseln.

Funktionen, die wiederum direkt oder indirekt eine Funktion aufrufen, die als variabler Zeiger übergeben wurde, dürfen ebenfalls den Stapel nicht wechseln, da zur Übersetzungszeit nicht bestimmt werden kann, was das Ziel des Zeigers ist.

Zu guter letzt ist noch zu beachten, dass ein Stapelwechsel *nie* mehrfach hintereinander vorkommen darf, da der verwendete Übersetzer als neuen Wert für den Stapelzeiger eine globale Variable erwartet. Diese globale Variable wird lediglich beim Taskwechsel aktualisiert. Also wenn das Betriebssystem von einem Task, der auf dem BTS lief auf einen anderen Task umschaltet. Sollte also der Stapel mit Hilfe des modifizierten Übersetzers mehrfach gewechselt werden, so wird beim zweiten Stapelwechsel lediglich der Stapel auf den Wert beim ersten Stapelwechsel zurückgesetzt.

Für eine wie hier modifizierte Funktion in AT&T IA32 Assembler Syntax siehe Listing 3.1.

```
1  push %ebp
2  mov %esp, %ebp
3  push %esi #Alten Wert von %esi sichern.
4  mov BTS, %esp #%esp auf die unterste Stelle des BTS setzen
5  sub $zahl, %esp # 'zahl' Byte fuer lokale Variablen
6  mov %esp, %esi #%esi auf das Ende der lokalen Variablen setzen
7  <Funktionsrumpf>
8  lea -4(%ebp), %esp #Am Ende der Funktion den Stapelzeiger zuruecksetzen,
9  pop %esi #den gesicherten Wert vom %esi laden
10 pop %ebp
11 ret
```

[3] also Funktionen sind, die eine Funktion aufruft, die einen direkten `WaitEvent` Aufruf hat oder Funktionen, die eine Funktion aufruft, die einen indirekten `WaitEvent` Aufruf hat

Listing 3.1 – Modifizierungen auf callee Seite um einen Stapelwechse zu unterstützen.
Die Zeilen 3,4,6 und 9 sind neu im Vergleich zu einem normalen Funktionsaufruf der GNU-ABI.
Das Register %esi steht fortan nicht mehr für allgemeine Operationen zur Verfügung, da damit die lokalen Variablen der Funktion angesprochen werden.
Zeile 8 ist lediglich modifiziert: Normalerweise liegen alle caller save Register vor dem %ebp, hier liegt aber der gesicherte Wert vom %esi nach dem %ebp.
Für IA32 gibt es für das standardmäßige Beenden einer Funktion die `leave` Instruktion, da diese hier nicht verwendet werden kann, werden diese aufgeteilt in Zeilen 8 und 10.
Somit gibt es 4 zusätzliche Instruktionen für den Stapelwechsel selbst und 1 zusätzliche, weil eine Standardinstruktion nicht verwendet werden kann. Also 5 zusätzliche Instruktionen insgesamt.

3.3.2 Auswahl der Funktionen für eine callee seitige Implementierung

Da die Stapelposition des BTS lediglich beim Taskwechsel aktualisiert wird, müssen aus der bestehenden Menge an möglichen Funktionen für den Stapelwechsel, zusätzlich die Funktionen entfernt werden, die sowohl von BTs als auch von ETs aufgerufen werden. Ansonsten würden diese Funktionenen, wenn sie von einem BT aufgerufen werden ein zweites Mal den Stapel wechseln, da sie sich bereits auf dem BTS befinden, aber der Übersetzer einen zweiten Stapelwechsel in den Quellcode eingefügt hätte.

Aus allen Funktionen kann nun ein gerichteter Graph aufgestellt werden. Die Knoten des Graphen sind dabei die Funktionen und die gerichteten Kanten sind die Funktionsaufrufe zu den jeweiligen Funktionen. Alle bisherigen Funktionen, die *nicht* für den Stapelwechsel in Frage kommen, werden markiert. Wenn in diesem Graphen nun ein Zyklus vorhanden ist, so bedeutet dies, dass jede Funktion in dem Zyklus direkt oder indirekt sich selbst aufruft. Dadurch darf keine Funktion, die in einem Zyklus auftaucht den Stapel wechseln, da ansonsten die Gefahr besteht, dass diese Funktion mehrfach aufgerufen wird. Diese werden ebenfalls markiert.

Die restlichen Funktionen F sind alle mögliche Funktionen die für den Stapelwechsel auf callee Seite in Frage kommen. Es kann nun eine Teilmenge f von F bestimmt werden, für diese Teilmenge muss gelten, dass bei 2 verschiedenen Elementen $a \neq b$ der Teilmenge $a, b \in f$ es keinen Weg von a nach b im Graphen gibt. Bzw. die Funktion a ruft weder direkt noch indirekt die Funktion b auf, mit $a \neq b$ und $a, b \in f$. Ansonsten würde wieder beim zweiten Ausführen des Stapelwechsels lediglich der Stapelzeiger auf den Wert beim ersten Stapelwechsel zurückgesetzt werden.

In der verwendeten Implementierung wurde dieser Analyseschritt so ausgeführt, dass für jede Funktion b, dessen Aufrufer a noch in der Menge F vorhanden ist, die Funktion b entfernt wurde. Dadurch bleiben nur Funktionen übrig, die nicht eine Funktion aufrufen, die selbst den Stapel wechselt.

3.3.3 Auswahl der Funktionsaufrufe für eine caller seitige Implementierung

Auf caller Seite darf nun nicht mehr lediglich jede Funktion für sich betrachtet werden, sondern es muss jeder Funktionsaufruf betrachtet werden. Dabei muss wieder verhindert werden, dass der Stapel mehrfach gewechselt wird. Da nun allerdings die Implementierung auf caller Seite vorgenommen wird, kann an einigen Stellen der Stapel gewechselt werden, an denen dies vorher nicht ging. Beispielsweise können rekursive Funktionen selbst nicht den Stapel wechseln, der Aufruf zur Rekursiven Funktion hingegen schon.

Es wird nun ein Algorithmus forgestellt, welcher so viele Stapelwechsel wie möglich einbaut. Dabei ist zu beachten, dass dies nicht direkt den geringsten Speicherverbrauch erwirtschaft, sondern lediglich so viel wie möglich auf dem BTS laufen lässt. Zu erst muss jeder Funktionsaufruf für jeden Extended Task betrachtet werden. Sollte die Implementierung der Funktion bereits einen direkten callee seitigen Stapelwechsel haben, so ist trivialer Weise mit der nächsten Funktion fortzuführen. Sollte die Funktion hingegen einen indirekten callee seitigen Stapelwechsel haben, so kann es noch Funktionen in der selben Verschachtelungstiefe geben, die keine callee seitige Implementierung haben. Dies sind somit Kandidaten für eine caller seitige Implementierung und werden ebenfalls in den hier vorgestellten Algorithmus gesteckt (siehe Abbildung 3.8).

Sollte die Funktion einen direkten `WaitEvent` Aufruf haben, so kommen die Funktionen vor und nach dem `WaitEvent` Aufruf in Frage und werden in den Algorithmus gesteckt. Bei einem indirekten `WaitEvent` Aufruf muss weiter analysiert werden welche Funktion genau den `WaitEvent` Aufruf hat, damit wieder die Funktionen vor und nach dem Aufruf den Stapel wechseln können. Also werden alle direkt aufgerufenen Funktionen in den Algorithmus gesteckt (siehe Abbildung 3.9).

Sollte eine Funktion bis hier gekommen sein, so hat sie weder einen direkten noch einen indirekten Stapelwechsel und auch keinen direkten oder indirekten `WaitEvent` Aufruf. Also kann sie auf caller Seite den Stapel wechseln.

Nach dem ausführen dieses Algorithmus können dann die entsprechenden Funktionsaufrufe für eine caller seitige Implementierung markiert werden.

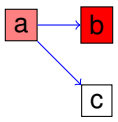


Abbildung 3.8 – Indirekter `WaitEvent` Aufruf:
Zu sehen sind die Funktionen a,b,c und `WaitEvent`. Die Kanten stellen die Funktionsaufrufe dar. a sei in diesem Beispiel eine Funktion, die direkt von einem ET aufgerufen wird. Die Implementierung von der Funktion b sieht bereits einen Stapelwechsel vor. Weder Funktion a noch Funktion b können auf caller Seite den Stapel wechseln, Funktion c hingegen schon.

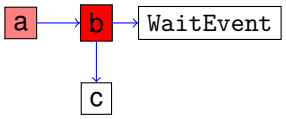


Abbildung 3.9 – Indirekter WaitEvent Aufruf:
Zu sehen sind die Funktionen a,b,c und `WaitEvent`. Die Kanten stellen die Funktionsaufrufe dar. a sei in diesem Beispiel eine Funktion, die direkt von einem ET aufgerufen wird. Die Implementierung von der Funktion b sieht einen `WaitEvent` Aufruf vor. Weder Funktion a noch Funktion b können auf caller Seite den Stapel wechseln, Funktion c hingegen schon.

3.4 Nötige Modifizierungen am Betriebssystem

Zusätzlich zu der Analyse über alle Funktionen und die Modifizierungen am Übersetzer sind noch Modifizierungen am Taskwechsel des Betriebssystems notwendig.

Dabei lässt sich der Taskwechsel als Tabelle darstellen, in der von einem Taskart zu einer anderen Taskart gewechselt wird. Bisher gab es in dieser Tabelle lediglich je eine Zeile und Spalte für sowohl BTs und ETs. Nun kommt noch jeweils eine Zeile und Spalte für SETs hinzu (siehe Tabelle 3.1), wobei in diesem Fall ein SET ein ET ist, der momentan auf dem BTS läuft.

Innerhalb von dOSEK wurden die einzelnen Stapelzeigerpositionen beim Taskwechsel immer separat im Speicher abgelegt, wo diese den einzelnen Tasks zugeordnet werden können. Sollte nun von einem ET zu einem anderen Task gewechselt werden, so genügte es erst den Kontext auf dem aktuellen Stapel zu sichern, dann die aktuelle Stapelzeigerposition in der dem ET zugehörigen Variable für den Stapelzeiger abzulegen um schließlich den neuen Stapelzeiger aus der Variable die dem nächsten Task angehört zu laden und von dem neuen Stapel den Kontext zu laden.

Bei dem Taskwechsel von einem BT zu einem BT kann äquivalent verfahren werden.

Da allerdings alle BTs auf dem selben Stapel laufen, dem BTS, teilen sich alle BTs die Variable für die gesicherte Stapelzeigerposition. Um nun bei einem Wechsel von einem BT zu einem anderen BT die aktuelle Stapelposition, und damit den Zeiger auf den gesicherten Kontext des aktuell laufenden BT, nicht zu verlieren, wurde zwischen dem *starten* eines neuen BT und dem *beenden* des aktuell laufenden BT unterschieden:

1. Beim *starten* eines neuen BT wird die aktuelle Stapelzeigerposition als letztes auf den BTS gelegt und die Variable für die Position des BTS zeigt auf diesen Wert, ähnlich wie beim Stapelrahmenzeiger beim Funktionsaufruf

2. Beim *beenden* des aktuell laufenden BT wird die Stapelzeigerposition auf die beim starten gesicherte Position zurückgesetzt, womit der alte Kontext geladen werden kann. Ähnlich dem `ret` am Funktionsende

So wurde bisher der Taskwechsel in dOSEK realisiert, wenn nun von einem SET auf einen anderen Task gewechselt wird, so wird einerseits die Variable für den BTS aktualisiert und andererseits die Variable für den ETS des SET.

nach \ von	BT	ET	SET
BT	✓	✓	✗
ET	✓	✓	✗
SET	✗	✗	!!!

Tabelle 3.1 – Taskwechseltabelle bei der Verwendung von SETs
Mit ✓markierte Stellen werden bereits durch die Implementierung ohne SETs abgedeckt, mit ✗markierte Stellen wurden dem Betriebssystem für die Benutzung von SETs hinzugefügt.
Da ein SET nur ein ET ist, der momentan auf dem BTS läuft, ist der Wechsel von einem SET zu einem *SET* äquivalent zu dem Wechsel von einem SET zu einem *ET*.

Wenn wie hier beide Variablen immer aktualisiert werden, so sind keine weiteren Modifizierungen beim Wechsel von ET bzw. BT nach SET nötig, da somit der Kontext eines SET genauso Wiederherstellen werden kann wie der Kontext eines ET.

4 ANALYSE

Wir haben nun eine funktionierende Implementierung auf callee Seite für SETs. Diese Implementierung werde ich auf den veränderten Stapelspeicherverbrauch als auch die Ausführzeit evaluieren. Bei der Evaluation werde ich jeweils zu erst auf die Randfälle eingehen, die betrachtet werden müssen und anschließend auf die Meßmethodik und die Messung an sich.

4.1 Veränderter Stapelspeicherverbrauch

Ein generelles Problem bei der bisherigen Implementierung von SETs ist, dass Funktionen, wenn sie den Stapel wechseln können, immer den Stapel wechseln.

Dabei muss nicht immer Speicher auf dem ETS eingespart werden, wenn eine Funktion f auf den BTS wechselt. Es wird genau dann kein Speicher auf dem ETS eingespart, wenn die Funktion f nicht für den maximalen Stapelspeicherverbrauch auf dem ETS verantwortlich ist, sondern eine andere Funktion g, welche nicht den Stapel wechselt. Wenn aber kein Speicher auf dem ETS eingespart wird, dann bleibt der gesamte Stapelspeicherverbrauch des Systems im besten Fall gleich, da die Funktion f nun auf dem BTS läuft und diesen unter Umständen vergrößert.

Sollte f einen *größeren* Stapelspeicherverbrauch haben als g, so bedeutet dies nicht automatisch, dass Speicherplatz eingespart wird, wenn f auf dem BTS läuft. Dafür muss der eingesparte Speicher auf dem ETS größer sein, als der zusätzlich verbrauchte Speicher auf dem BTS, wenn f auf dem BTS läuft.

Ich werde nun auf die verwendete Testmethodik eingehen und anschließend auf die beide gerade vorgestellten Fälle.

4.1.1 Testmethodik

Um den Stapelspeicherverbrauch zu messen wurde beim Start des Betriebssystems jedes Byte des BTS mit einer zufällig ausgewählten Zahl beschrieben. Beim erstmaligen Ausführen eines ETs wurde der jeweilige Stapel des ET so weit wie möglich mit der selben Zahl beschrieben.

Da zum Zeitpunkt des Startens eines ET bereits ein initialer Kontext auf dem jeweiligen ETS liegt, kann dieser nicht vollständig beschrieben werden, da ansonsten dieser initiale Kontext verloren geht. Deswegen hat jeder ET einen minimalen Stapelspeicherverbrauch. Da dieser initiale Kontext immer gleich groß ist, im Fall der POSIX Implementierung sind dies 6 Registerwerte also die obersten 24 Byte.

Wenn sich nun ein ET terminiert hat, so wurde, bevor ein anderer Task läuft, ausgewertet wie viele von diesen Zahlen noch auf dem Stapel liegen. Die Differenz zwischen Größe des angelegten Stapels und den gezählten Zahlen ist der verwendete Speicherverbrauch des ET.

Für den BTS wurde der Speicherverbrauch genauso gemessen, allerdings erst beim Beenden des Betriebssystems, da mehrere Tasks auf den BTS zugreifen können.

Da nun allerdings diese zufällig ausgewählte Zahl genau der Wert sein kann, der als letztes auf einen der Stapel gelegt worden ist, muss ein zweites Mal mit einer zweiten Zahl gemessen werden. Dies muss solange wiederholt werden bis für zwei unterschiedliche Zahlen der selbe Stapelspeicherverbrauch auf jedem Stapel gemessen worden ist.

4.1.2 Fall 1: Stapelwechsel kann nicht zu einem verringerten Speicherverbrauch führen

Im ersten Fall wechselt eine Funktion f den Stapel ohne, dass der Speicherverbrauch des entsprechenden ETS kleiner wird.

Dies ist genau dann der Fall, wenn es eine andere Funktion g gibt, die nicht auf den BTS ausgelagert werden kann und diese für den maximalen Speicherverbrauch des ETS verantwortlich ist. Da f nun auf dem BTS läuft, kann dieser unter Umständen wachsen.

Da der ETS durch den Stapelwechsel von f nicht kleiner wird, aber unter Umständen der BTS wächst, da f nun zusätzlich auch diesen Stapel verwendet, bleibt der Stapelspeicherverbrauch (SSV) des gesamten Systems in einem solchen Fall bestens gleich dem SSV ohne Stapelwechsel.
Für diesen Fall habe ich eine Testanwendung geschrieben, in der die Funktion f eine lokale Variable der Größe von 512 Byte verwendet. Diese Funktion f wechselt bei der SET Implementierung den Stapel (siehe Abbildung 4.1).

Da der SSV des ETS in dem hier vorgestellten Fall nie kleiner wird und somit der gesamte SSV des Systems nicht kleiner werden kann, sollte in diesem ersten Fall *nie* der Stapel gewechselt werden. Im zweiten und dritten Fall, werde ich mich nun damit beschäftigen, wie sich der Stapelspeicherver-

beobachteter Stapel	mit SETs	ohne SETs
ETS	1056	1056
BTS	1668	1052
Gesamt	2724	2108

Tabelle 4.1 – Gemessene SSV Werte im Fall 1.
Die Werte sind in Byte angegeben.

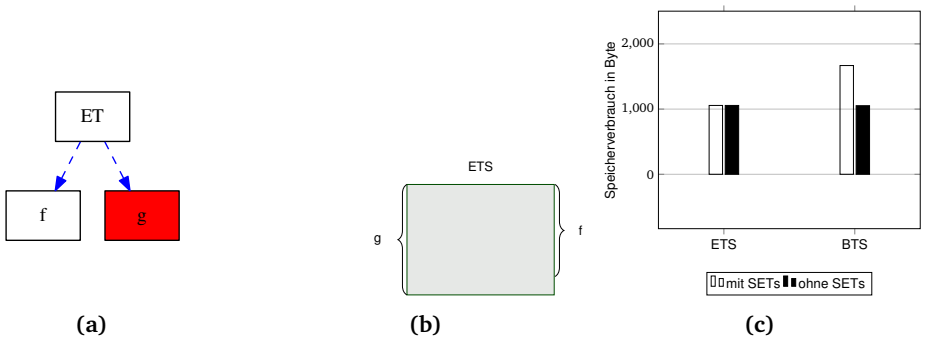


Abbildung 4.1 – Testanwendung für den Fall 1:
(a) Aufrufgraph des ET:
g kann den Stapel nicht wechseln, da g einen WaitEvent Aufruf hat.
(b) SSV der beiden aufgerufenen Funktionen.
(c) Gemessener Speicherverbrauch der Testanwendung im Fall 1.
Es gibt einen Extended Task, der 2 Funktionen f und g aufruft (siehe (a)). Lediglich die Funktion f kann den Stapel wechseln, da g einen WaitEvent Aufruf hat. Allerdings benötigt die Funktion g mehr Speicher als die Funktion f (siehe (b)), weswegen sich der Speicherverbrauch des ETS nicht ändert. Dies ist auch in (c) zu sehen.
Der erhöhte Speicherverbrauch des BTS ist damit zu erklären, dass zum Zeitpunkt zu dem f aufgerufen wird, sich der BTS an seiner maximalen Speicherbelegung befindet und deswegen der SSV von f zusätzlich auf dem BTS liegt.
Die genau gemessenen Werte sind in Tabelle 4.1 zu sehen.

brauch des Systems ändert, wenn eine Funktion den Stapel wechselt, welche am maximalen SSV des ETS beteiligt ist.

4.1.3 Fall 2: Stapelwechsel führt immer zu verringertem Speicherverbrauch

Im zweiten Fall ist die Funktion f mit am maximalen SSV des ETS beteiligt. Wenn die Funktion f auf den BTS wächselt, so wird der ETS kleiner. In diesem zweiten Fall ist die Lücke zwischen der Stapelposition des BTS zum Zeitpunkt des Aufrufs von f und dem maximalen SSV des BTS groß genug, damit die gesamte Funktion f auf dem BTS laufen kann, ohne dass sich der maximale SSV des BTS ändert (siehe Abbildung 4.2).

beobachteter Stapel	mit SETs	ohne SETs
ETS	124	1072
BTS	1052	1052
Gesamt	1176	2124

Tabelle 4.2 – Gemessene SSV Werte im Fall 1.
Die Werte sind in Byte angegeben.

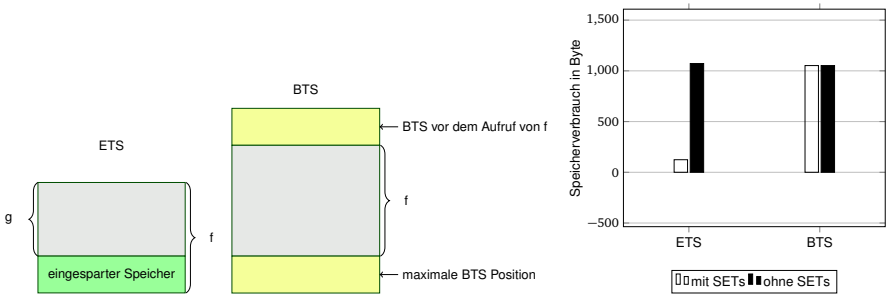


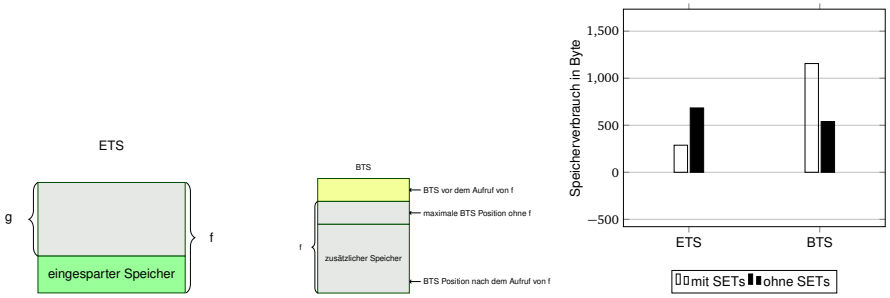
Abbildung 4.2 – Testanwendung für den Fall 2:
Der Aufrufgraph ist äquivalent zu dem von Abbildung 4.1.
(a) SSV der Funktionen f und g auf dem ETS
(b) SSV der Funktion f, wenn diese auf dem BTS läuft. Da die Differenz zwischen maximaler Stapelposition des BTS und Stapelposition des BTS beim Aufruf von f größer ist als der SSV der Funktion f, verändert sich der SSV des BTS nicht.
(c) Gemessener Speicherverbrauch der Testanwendung im Fall 2.
Es gibt, wie im 1. Fall, einen Extended Task, der 2 Funktionen f und g aufruft. Lediglich die Funktion f kann den Stapel wechseln, da g einen WaitEvent Aufruf hat.
Diesmal benötigt die Funktion f mehr Speicher als die Funktion g (siehe (a)), deswegen verringert sich der SSV des ETS, wenn die Funktion f auf dem BTS läuft, siehe (b).
Da die Funktion f, zum Zeitpunkt an dem sie aufgerufen wird, vollständig in die Lücke zwischen der aktuellen BTS Position und der maximalen BTS Position passt, verändert sich der SSV des BTS nicht.
Insgesamt verringert sich somit der Speicherverbrauch des Systems. Die genau gemessenen Werte sind in Tabelle 4.2 zu sehen.

4.1.4 Fall 3: Stapelwechsel führt manchmal zu verringertem Speicherverbrauch

Im dritten Fall ist, äquivalent zum zweiten Fall, wieder die Funktion f am maximalen SSV des ETS beteiligt. Deswegen wird der ETS kleiner wenn die Funktion f auf dem BTS läuft. Allerdings ist nun zum Zeitpunkt an dem f aufgerufen wird *nicht* mehr genug Plats auf dem BTS vorhanden, so dass ein verringerter Speicherverbrauch entstehen kann. Es wird lediglich der Speicherverbrauch verschoben: Anstatt, dass es 2 Stapel gibt mit einem großen Speicherverbrauch gibt es nun den ETS mit einem relativ geringen SSV und den BTS, der einen relativ großen SSV hat (siehe fig:fig5).

beobachteter Stapel	mit SETs	ohne SETs
ETS	288	684
BTS	1156	540
Gesamt	1444	1224

Tabelle 4.3 – Gemessene SSV Werte im Fall 3.
Die Werte sind in Byte angegeben.



(a) **(b)** **(c)**

Abbildung 4.3 – Testanwendung für den Fall 3:
Der Aufrufgraph ist äquivalent zu dem von Abbildung 4.1.
(a) SSV der Funktionen f und g auf dem ETS
(b) SSV der Funktion f, wenn diese auf dem BTS läuft. Da die Differenz zwischen maximaler Stapelposition des BTS und Stapelposition des BTS beim Aufruf von f *kleiner* ist als der SSV der Funktion f, wächst der SSV des BTS.
(c) Gemessener Speicherverbrauch der Testanwendung im Fall 3.
Es gibt, wie im ersten und zweiten Fall, einen ET, der 2 Funktionen f und g aufruft. Lediglich die Funktion f kann den Stapel wechseln, da g einen WaitEvent Aufruf hat.
Aus den selben Gründen wie im zweiten Fall verringert sich der SSV des ETS. Da die Funktion f, zum Zeitpunkt an dem sie aufgerufen wird, nicht vollständig in die Lücke zwischen der aktuellen BTS Position und der maximalen BTS Position passt, wächst der SSV des BTS.
Insgesamt vergrößert sich somit der Speicherverbrauch des Systems. Die genau gemessenen Werte sind in Tabelle 4.3 zu sehen.

Dies hat in dieser Messung einen negativen Einfluss auf den SSV des Systems, allerdings kann es bei mehreren ETs durchaus sinnvoll sein, diese Funktion trotzdem auf den BTS auszulagern. Sollte es mehrere dieser ETs geben und jeder der ETs hat eine wie hier beschriebene Funktion f. Dann kann unter Umständen, der SSV eines *jeden* ETS leicht verringert werden, während der SSV des BTS nur *ein mal* erhöht wird (siehe Abbildung 4.4).

Bis hier habe ich nur die Auswirkungen auf den veränderten SSV bei der Verwendung von SETs gezeigt. Für Echtzeitbetriebssystem ist außerdem die Ausführzeit von Anwendungen eine wichtiges Kriterium, weswegen die Veränderung auf die Ausführzeit bei der Verwendung von SETs im Folgenden evaluiert wird.

4.2 Auswirkungen auf die Laufzeit

Um die Auswirkungen auf die Laufzeit zu evaluieren werden nicht die Modifizierungen am Taskwechsel des Betriebssystems evaluiert, da ein Taskwechsel im Vergleich zu einem Funktionsaufruf verhältnismäßig selten vorkommt, sondern die Auswirkungen des Stapelwechsels auf callee Seite. Wie bereits in Kapitel 3 gesehen werden konnte ist die zusätzliche Anzahl der Instruktionen für einen Stapelwechsel auf callee Seite konstant. Für die Ausführzeit ist wichtig, dass es zum einen

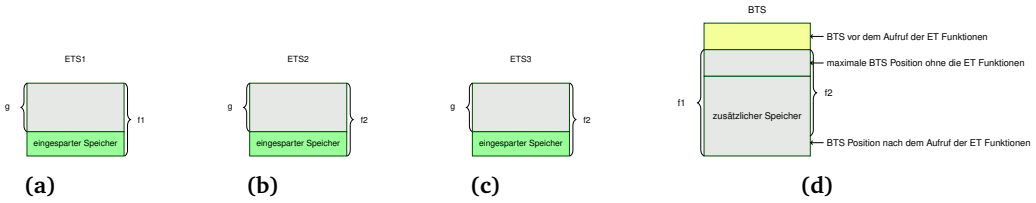


Abbildung 4.4 – Beispielhafter Aufbau für eine komplexe Evaluation ob ein Stapelwechsel sinnvoll ist:
(a) SSV der Funktionen f1 und g auf dem ETS1
(b) SSV der Funktionen f2 und g auf dem ETS2
(c) SSV der Funktionen f2 und g auf dem ETS3
(d) SSV der Funktionen f1 und f2, wenn diese auf dem BTS läuft. Da die Differenz zwischen maximaler Stapelposition des BTS und Stapelposition des BTS beim Aufruf von f1 bzw. f2 *kleiner* ist als der SSV der Funktion f1 bzw. f2, wächst der SSV des BTS.
Bei diesem beispielhaften Versuchsaufbau ist nicht trivial ob der SSV des Systems sinkt oder steigt. Um dies zu bestimmen sollte eine weitere Analysephase entscheiden ob f2 den Stapel wechseln sollte. Dabei muss gelten, dass der eingesparte Speicher von ETS2 und ETS3 zusammen größer ist, als der zusätzlich benötigte Speicher von f2, wenn diese auf dem BTS läuft.
Sollte f2 auf dem BTS laufen, so kann es auch sinnvoll sein f1 auf dem BTS laufen zu lassen, da es nun einen größeren maximalen SSV auf dem BTS gibt.

mehr Instruktionen sind und, dass die Instruktion zum Wiederherstellen des Stapelzeigers nun keine Addition mehr ist, sondern eine Ladeoperation vom alten Stapel. Im Normalfall sind Ladeoperationen vom Hauptspeicher langsamer als arithmetische Operationen auf Registern, allerdings lässt sich dies schlecht evaluieren, da sowohl Cache Größe, als auch Assoziativitätsgrad des Caches eine Rolle spielen. Deswegen wird im folgenden angenommen, dass bei der Messung der gesicherte Wert, des alten Stapelzeigers im L1 Cache vorhanden war und diese Ladeoperation im Vergleich zur arithmetischen Operationen keinen Unterschied auf die Laufzeit verursacht.
Für die eigentliche Messung der zusätzlichen Laufzeit wurde eine weitere Testanwendung für dOSEK erstellt. In dieser Testanwendung wird in einer iterativen Schleife 10000 mal die Funktion aufgerufen, welche auf den BTS wechselt. In der ohne SETs konfigurierten Version von dOSEK wechselt die Funktion nicht den Stapel.

Direkt vor und nach der Schleife wird jeweils eine Zeitmessung mit Hilfe der Standard C Bibliotheksfunktion `clock`, siehe [Koe], vorgenommen. Durch das Abziehen der beiden gemessenen Zeiten wurde bestimmt wie lange die Schleife läuft. Wenn nun für beide dOSEK Versionen die Differenz der Laufzeit der Schleife gemessen wird, so ist bekannt wie groß die Laufzeitdifferenz beim 10000-fachen Stapelwechsel ist. Da die Laufzeitdifferenz nicht schwankt, weil die Anzahl der zusätzlichen Instruktionen konstant ist, kann angenommen werden, dass somit die Laufzeitdifferenz für den einfachen Stapelwechsel direkt proportional mit der Differenz für den 10000-fachen Stapelwechsel ist.
Um weitere Schwankungen während der Messung zu minimieren wird 100 mal die Laufzeit für den 10000-fachen Stapelwechsel gemessen und mit den gemessenen Werten das arithmetische Mittel so wie weitere statistische Werte ermittelt, siehe Abbildung 4.5. Da die Laufzeit und Laufzeitschwan-

kungen auch stark vom verwendeten Rechner abhängen, gehe ich hier kurz auf diesen ein.
Verwendet wurde ein AMD Phenom II X6 1055t Prozessor, welcher mit 2.8 GHz taktet. Die Taktfrequenz wurde für die Ausführung des Tests festgesetzt, indem etwaige Stromspachermechanismen, welche im Leerlauf den Takt des Prozessors senken, abgestellt wurden um gleichmäßige Testergebnisse zu erhalten. Aus dem selben Grund wurde auch der Turbomodus, welcher bei kurzzeitigen Lastspitzen den Prozessor schneller taktet, abgestellt. Als verwendetes Betriebssystem kommt eine Linux Variante zum Einsatz.
Um möglichst nah an eine direkt auf der Maschine laufende Laufzeit zu kommen, wurde die Priorität des Prozesses mit Hilfe des Linux Kommandozeilen Befehls `chrt` auf die Echtzeitpriorität 99 gesetzt, damit dieser zur Laufzeit nicht von einem anderen Prozess verdrängt wird. Zusätzlich wurde mit dem Befehl `taskset` der Prozess auf einen bestimmten Prozessorkern festgesetzt, damit der Prozess zur Laufzeit nicht auf einen anderen Prozessorkern verdrängt wird.

Wie die Messwerte zeigen sind die Auswirkungen auf die Laufzeit sehr gering, weshalb immer wenn Speicher eingespart werden kann bei einem Stapelwechsel der Stapelwechsel auch realisiert werden sollte.

4.3 Zusammenfassung

Wie im ersten Teil dieses Kapitels gesehen werden konnte ist im allgemeinen eine zusätzliche Analysephase notwendig um zu evaluieren ob eine Funktion nicht nur den Stapel wechseln *kann* sondern ob dies auch sinnvoll ist. So kann es vorkommen, dass ein Stapelwechsel einer Funktion nicht zu einem verringerten SSV des ETS führt und somit ein Stapelwechsel dieser Funktion niemals sinnvoll ist.

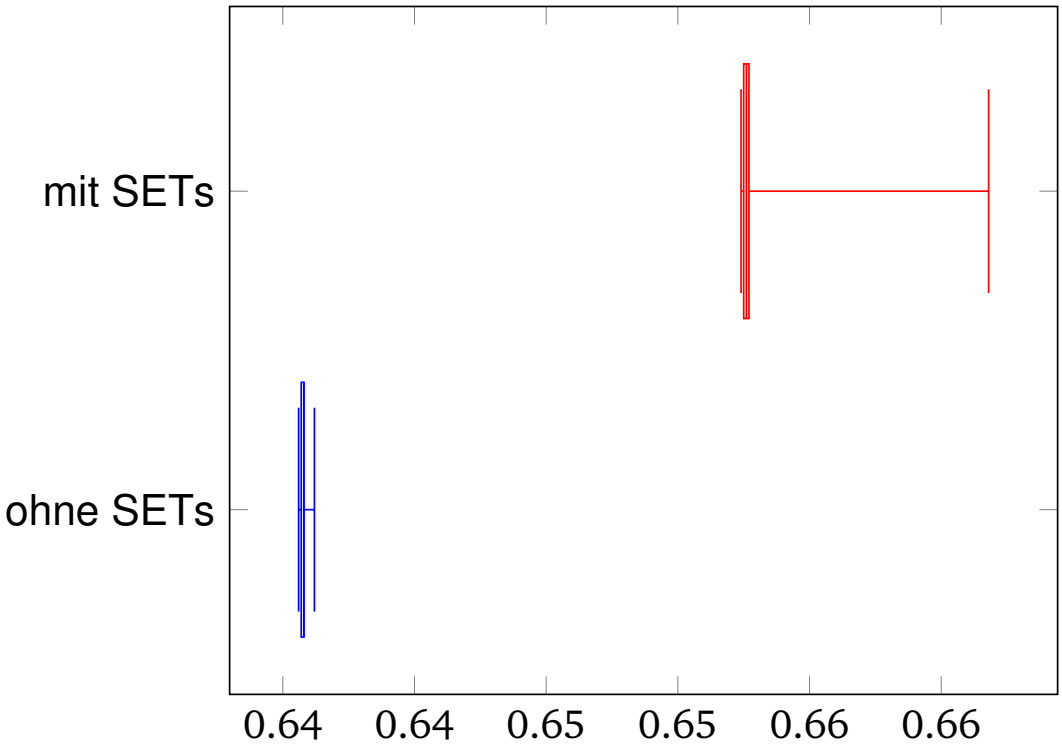
Sollte der Stapelwechsel einer Funktion zu einem verringerten SSV des ETS führen, so muss zusätzlich ausgewertet werden, wie viel Speicherplatz auf dem ETS eingespart wird im Vergleich zu dem zusätzlichen SSV, die auf dem BTS entstehen.

Im Allgemeinen ist die aktuelle Implementierung in Bezug auf den Speicherverbrauch somit nicht optimal, da auch zusätzlicher Speicherverbrauch entstehen kann. Wenn beim erstellen einer dOSEK Anwendung dem Programmierer bewusst ist, nach welchen Kriterien bisher der Stapel gewechselt wird, so kann zwar ein verringerter Speicherverbrauch erzielt werden, jedoch ist es nicht immer trivial ob der Stapelwechsel einer bzw. mehrerer Funktionen zu einem verringertem Speicherverbrauch führt.

Im Bezug zur Laufzeit wurde festgestellt, dass diese zwar zugenommen hat, allerdings ist die Zunahme zur Laufzeit linear abhängig im Vergleich zu der Anzahl der Stapelwechsel und lediglich bei einer sehr großen Anzahl an Stapelwechseln überhaupt messbar, weswegen dies kein größeres Problem ist.

Art des Wertes	ohne SETs	mit SETs
Minimum	0.6406	0.6574
1. Quartil	0.6407	0.6575
Median	0.6408	0.6576
Mittelwert	0.6408	0.6579
3. Quartil	0.6408	0.6577
Maximum	0.6412	0.6668
Standardabweichung	0.000090	0.001298

(a) Gemittelte Messwerte aus 100 Messungen für die Laufzeit des 10000 fachen Stapelwechsels ohne SETs und mit SETs
Alle Angaben in Sekunden. Die Differenz des Mittelwerts, und damit die gemittelte Laufzeitdifferenz für den 10000-fachen Stapelwechsel auf callee Seite, ergibt sich zu 0.0171 Sekunden. Für den einfachen Stapelwechsel erhält man somit eine Laufzeitdifferenz von ungefähr 1.7 Nanosekunden



(b) Boxplot aus den gemittelten Messwerten. Auf der x-Achse sind die Sekunden angegeben, auf der y-Achse die verwendete dOSEK Konfiguration.

Abbildung 4.5

5 FAZIT

Die hier vorgestellte Methode um Speicherplatz bei einem OSEK Betriebssystem einzusparen ist somit durchaus ein guter Ansatz. Allerdings wird eine weitere Analysephase benötigt um aus den möglichen Stapelwechsel Funktionen, diejenigen zu finden, die effektiv zu einer Speicherplatzminimierung führen. Ohne die Analysephase kann nicht garantiert werden, dass nicht weniger Speicher benötigt wird, sondern der Speicherverbrauch durchaus steigen kann, gerade wegen dem Stapelwechsel.

Sollte eine solche zusätzliche Analysephase entwickelt werden, könnte direkt entschieden werden, welche Funktionen den Stapel wechseln sollen, da die Auswirkungen auf die Laufzeit sehr geringfügig sind.

Zusätzlich sollte der Stapelwechsel auch auf caller Seite implementiert werden um so mehr Flexibilität bei der Auswahl der Stelle des Stapelwechsels zu erhalten. So könnten, wie in Kapitel 3 beschrieben, auch rekursive Funktionen direkt auf den BTS wechseln.

Vor allem mit dieser zusätzlichen statischen Analyse würde sich somit durchgängig ein verringerter oder gleicher Speicherverbrauch für jedes mit dOSEK erstelltem Betriebssystem garantieren ohne dass dabei der entsprechende Quellcode verändert werden muss, da jegliche Optimierungen von den entsprechenden Entwicklungswerkzeugen, wie den statischen Analysen, so wie dem Übersetzer vorgenommen werden.

Mit einem so gesunkenen Speicherverbrauch würden bisherige OSEK Echtzeitbetriebssysteme auch auf Mikrocontrollern mit einem kleinerem Hauptspeichervorrat laufen lassen oder zu den bisherigen Echtzeitbetriebssystemen weitere Funktionen hinzufügen lassen ohne dass die entsprechende Hardware ersetzt werden muss.

Damit können dann bessere Sicherheits- bzw. Komfortfunktionen mit den selben Hardwarekosten realisiert werden oder bei gleichen Sicherheits- und Komfortfunktionen günstigere Hardware verbaut werden.

ABKÜRZUNGSVERZEICHNIS

BT	Basic Task
ETS	Extended Task Stapel
BTS	Basic Task Stapel
ET	Extended Task
SET	Semi Extended Task
API	Application Programming Interface
caller	aufrufende Funktion
callee	aufgerufene Funktion
SSV	Stapelspeicherverbrauch

ABBILDUNGSVERZEICHNIS

2.1 Beispiel für Funktionsaufrufkonventionen 4

2.2 Ein bisher nicht fertiger Stack nach einem Funktionsaufruf der GNU-ABI. Bisher liegen lediglich gesicherte Register auf dem Stack, es fehlen also noch etwaige Paramter und/oder lokale Variablen einer Funktion. Der %esp ist der Stapelzeiger in der IA32 Architektur und zeigt auf den zuletzt eingetragenen Wert auf dem Stapel. In diesem Fall also eines der *callee save* Register. Wenn nun ein weiteres Element auf den Stack gelegt werden würde, würde der %esp nach unten in Richtung kleinster Speicheradresse wandern und das Element würde unterhalb von den *callee save* Registern liegen. 6

2.3 Ein im Vergleich zu Abbildung 2.2 leicht erweiterter Stapel nach einem Funktionsaufruf der GNU-ABI. Hinzugekommen sind Funktionsparameter und der Instruktionszeiger des Aufrufers %eip. Wobei das %eip Register streng genommen auch einfach nur zu den *caller save* Registern zählt, muss beachtet werden, dass zwischen der Sicherung des %eip und den restlichen *caller save* Registern die Paramter der aufzurufenden Funktion liegen. Wie hier außerdem zu sehen ist werden die Parameter in rückwärtiger Reihenfolge auf den Stapel gelegt, dies liegt an der *C declaration* Aufrufkonvention und darauf wird in eingegangen. Abbildung 2.4 7

2.4 Ein Stapelrahmen nach der GNU-ABI. In grün der aktuelle Stapelrahmen, in gelb der Stapelrahmen der vorherigen Funktion. Im Vergleich zu Abbildung 2.3 sind die lokalen Variablen der Funktion dazugekommen und ein Zeiger auf den aktuellen Stapelrahmen, wofür das Register %ebp verwendet wird. An exakt der Stelle des aktuellen Stapelrahmenzeigers liegt der Stapelrahmenzeiger der vorhergehenden Funktion. Mit Hilfe des %ebp und einem bekannten konstanten Offset kann nun auf die Parameter und die lokalen Variablen zugegriffen werden. Somit liegt die erste Variable 'var 1' an der Stelle im Speicher %ebp-4, bzw. in AT&T Assembler Syntax '-4(%ebp)'. Bei Parametern geschieht dies äquivalent, aber mit positivem Offset und unter Berücksichtigung der Tatsache, dass zwischen den Parametern und der Stelle auf die der %ebp zeigt noch die gesicherte Rücksprungadresse liegt. Dadurch, dass die Parameter in rückwärtiger Reihenfolge auf den Stapel gelegt worden sind, kann nun der erste Parameter der Funktion mit dem kleinsten Offset im Vergleich zum %ebp angesprochen werden. Dies hat den Vorteil, dass bei variabler Parameterlänge die Anzahl der Parameter dem ersten Parameter entnommen werden kann, siehe [Fog15] Seite 18. 8

2.5 Beispielsequenzdiagramm für die Quasiparallelität von zwei Fäden. Zu sehen sind die Fäden A und B, sowie deren zugesicherter Speicher [A] und [B], als auch die beiden Zeitpunkte t1 und t2. Als erstes läuft der Faden A. Zum Zeitpunkt t1 soll Task B laufen, dadurch sichert als erstes Faden A seinen Kontext auf seinem Speicher [A]. Anschließend kann Faden B seinen gesicherten Kontext von [B] laden und laufen. Zum Zeitpunkt t2 wird wieder zum Faden A gewechselt, dadurch muss Task B seinen Kontext sichern und Faden A seinen Kontext laden. Das Sequenzdiagramm könnte nun wieder von vorne beginnen und dadurch können die beiden Fäden durchgängig laufen. 10

2.6 Zustandsdiagramm und Zustandsübergänge eines Faden, zu Teilen entnommen aus [Ose] Seite 17 In (a) sind die Zustände abgebildet und beschrieben, während in (b) die Zustandsübergänge beschrieben werden. 14

2.7 Zustandsdiagramm eines Basic Task, entnommen aus [Ose] Seite 18 Zustände: **laufend**: Der aktuelle Task ist der einzige auf dem Prozessor laufende Task. **beendet**: Der Task ist beendet weil er bis zu seiner Beendigung komplett durchgelaufen ist. **bereit**: Der Task kann laufen, sobald er die höchste Priorität aller laufbereiten Tasks hat. Dann wird er vom Betriebssystem gestartet. 15

2.8 Beispielhafter Ablauf für mehrere Basic Tasks, die auf dem selben Stapel laufen können Zu sehen ist oben das Sequenzdiagramm für die Ausführung der Tasks Extended Task 1 (ET1), Basic Task 1 (BT1) und Basic Task 2 (BT2). Zusätzlich die Umschaltpunkte t0 - t4, an denen zwischen verschiedenen Tasks gewechselt wird, so wie das Ende des BT1 zum Zeitpunkt t5. Unten sind die jeweiligen Stapel der Tasks zu sehen: ETS1 für den Extended Task ET1 und der BTS für die Basic Tasks BT1 und BT2. Grün bedeutet, dass ein Task gerade läuft, bzw. der zugehörige Speicherbereich des Stapels gerade in Verwendung ist. Gelb bedeutet, dass ein Task verdrängt wurde oder auf ein Event wartet und dementsprechend ist der Kontext des Tasks auf seinem zugehörigen Speicherbereich des Stapels gesichert. Die Übergänge, die mit blauen Pfeilen markiert sind, sind Taskumschaltpunkte des Betriebssystems. Für die Prioritäten, der Tasks gilt: Priorität(BT1) < Priorit(BT2) < Priorität(ET1) Nicht in der Abbildung abgebildet ist das Event E1, welches dem Extended Task ET1 zugeordnet ist und zu Beginn nicht gesetzt ist. **(a)** ET1 läuft auf seinem Stapel, dem ETS1 **(a) → (b) bzw. t0** ET1 aktiviert den Task BT1 und wartet anschließend auf das Event E1. Daraufhin läuft BT1 auf dem BTS. **(b) → (c) bzw. t1** BT1 aktiviert BT2. BT2 hat eine höhere Priorität als BT1 und läuft somit als nächstes. Da außerdem Basic Tasks nicht warten können, kann BT2 auf dem selben Stapel laufen wie BT1, da BT2 bis zu seiner selbständigen Beendigung durchläuft. **(c) → (d) bzw. t2** BT2 setzt das Event E1, womit ET1 nun in den Zustand *bereit* wechselt. Da die Priorität von ET1 höher ist, als die von BT2, läuft somit ET1. **(d) → (e) bzw. t3** ET1 beendet sich, womit wieder BT2 läuft. **t4** BT2 beendet sich, womit BT1 läuft. **t5** BT1 beendet sich. 16

3.1 Probleme, wenn Extended Tasks auf dem BTS laufen: **(a)** Zu sehen ist ein Sequenzdiagramm. Ein BT startet einen ET höherer Priorität. Daraufhin läuft der ET bis dieser einen `WaitEvent` Aufruf macht und somit wartet. Anschließend läuft wieder der BT. Da der ET jedoch auf dem selben Stapel lief, überschreibt er nun den gesicherten Kontext des ET (siehe (c)). **(b)** Zustand des Stapel nachdem der ET aktiviert wurde. Noch geht alles gut. **(c)** Nun wartet der ET und der BT läuft wieder. Der %esp wächst nach unten ⇒ der BT schreibt in den gesicherten Kontext des ET. 18

3.2 Mögliche Stellen für den Stapelwechsel. Wie in Abbildung 3.1 gesehen werden kann, existiert immer dann ein Problem, wenn ein ET auf dem BTS wartet. Darum muss kann ein ET vor dem `WaitEvent` Aufruf auf dem BTS laufen, muss aber vor dem `WaitEvent` Aufruf seinen gesamten Kontext auf seine ETS Instanz, <ETS> legen. Nach dem `WaitEvent` Aufruf kann der ET wieder auf dem BTS laufen. 18

3.3 Auf- und Abbau des Stapels bei Funktionsaufruf und Funktionsende 19

3.4 Stapelwechsel auf callee Seite indem lediglich der %esp auf einen neuen Stapel gesetzt wird 20

3.5 Verwendung eines Basiszeigers um Lücken unbekannter Länge zu umgehen. Bei einem Stapelwechsel ist die Differenz zwischen neuer und alter Stapeladresse an sich auch nur eine Lücke unbekannter Länge. 21

3.6 Stapelaufbau bei einem möglichen Stapelwechsel auf caller Seite. 22

3.7 Direkter und indirekter `WaitEvent` Aufruf: Zu sehen sind die Funktionen a,b,c und `WaitEvent`. Die Kanten stellen die Funktionsaufrufe dar. Funktion c ruf direkt `WaitEvent` auf, Funktionen a und b lediglich indirekt. Keine dieser Funktionen darf den Stapel wechseln. 23

3.8 Indirekter `WaitEvent` Aufruf: Zu sehen sind die Funktionen a,b,c und `WaitEvent`. Die Kanten stellen die Funktionsaufrufe dar. a sei in diesem Beispiel eine Funktion, die direkt von einem ET aufgerufen wird. Die Implementierung von der Funktion b sieht bereits einen Stapelwechsel vor. Weder Funktion a noch Funktion b können auf caller Seite den Stapel wechseln, Funktion c hingegen schon. 25

3.9 Indirekter WaitEvent Aufruf: Zu sehen sind die Funktionen a,b,c und `WaitEvent`. Die Kanten stellen die Funktionsaufrufe dar. a sei in diesem Beispiel eine Funktion, die direkt von einem ET aufgerufen wird. Die Implementierung von der Funktion b sieht einen `WaitEvent` Aufruf vor. Weder Funktion a noch Funktion b können auf caller Seite den Stapel wechseln, Funktion c hingegen schon. 26

4.1 Testanwendung für den Fall 1: **(a)** Aufrufgraph des ET: g kann den Stapel nicht wechseln, da g einen WaitEvent Aufruf hat. **(b)** SSV der beiden aufgerufenen Funktionen. **(c)** Gemessener Speicherverbrauch der Testanwendung im Fall 1. Es gibt einen Extended Task, der 2 Funktionen f und g aufruft (siehe (a)). Lediglich die Funktion f kann den Stapel wechseln, da g einen WaitEvent Aufruf hat. Allerdings benötigt die Funktion g mehr Speicher als die Funktion f (siehe (b)), weswegen sich der Speicherverbrauch des ETS nicht ändert. Dies ist auch in (c) zu sehen. Der erhöhte Speicherverbrauch des BTS ist damit zu erklären, dass zum Zeitpunkt zu dem f aufgerufen wird, sich der BTS an seiner maximalen Speicherbelegung befindet und deswegen der SSV von f zusätzlich auf dem BTS liegt. Die genau gemessenen Werte sind in Tabelle 4.1 zu sehen. 31

4.2 Testanwendung für den Fall 2: Der Aufrufgraph ist äquivalent zu dem von Abbildung 4.1. **(a)** SSV der Funktionen f und g auf dem ETS **(b)** SSV der Funktion f, wenn diese auf dem BTS läuft. Da die Differenz zwischen maximaler Stapelposition des BTS und Stapelposition des BTS beim Aufruf von f größer ist als der SSV der Funktion f, verändert sich der SSV des BTS nicht. **(c)** Gemessener Speicherverbrauch der Testanwendung im Fall 2. Es gibt, wie im 1. Fall, einen Extended Task, der 2 Funktionen f und g aufruft. Lediglich die Funktion f kann den Stapel wechseln, da g einen WaitEvent Aufruf hat. Diesmal benötigt die Funktion f mehr Speicher als die Funktion g (siehe (a)), deswegen verringert sich der SSV des ETS, wenn die Funktion f auf dem BTS läuft, siehe (b). Da die Funktion f, zum Zeitpunkt an dem sie aufgerufen wird, vollständig in die Lücke zwischen der aktuellen BTS Position und der maximalen BTS Position passt, verändert sich der SSV des BTS nicht. Insgesamt verringert sich somit der Speicherverbrauch des Systems. Die genau gemessenen Werte sind in Tabelle 4.2 zu sehen. 32

4.3 Testanwendung für den Fall 3: Der Aufrufgraph ist äquivalent zu dem von Abbildung 4.1. **(a)** SSV der Funktionen f und g auf dem ETS **(b)** SSV der Funktion f, wenn diese auf dem BTS läuft. Da die Differenz zwischen maximaler Stapelposition des BTS und Stapelposition des BTS beim Aufruf von f *kleiner* ist als der SSV der Funktion f, wächst der SSV des BTS. **(c)** Gemessener Speicherverbrauch der Testanwendung im Fall 3. Es gibt, wie im ersten und zweiten Fall, einen ET, der 2 Funktionen f und g aufruft. Lediglich die Funktion f kann den Stapel wechseln, da g einen WaitEvent Aufruf hat. Aus den selben Gründen wie im zweiten Fall verringert sich der SSV des ETS. Da die Funktion f, zum Zeitpunkt an dem sie aufgerufen wird, nicht vollständig in die Lücke zwischen der aktuellen BTS Position und der maximalen BTS Position passt, wächst der SSV des BTS. Insgesamt vergrößert sich somit der Speicherverbrauch des Systems. Die genau gemessenen Werte sind in Tabelle 4.3 zu sehen. 33

4.4 Beispielhafter Aufbau für eine komplexe Evaluation ob ein Stapelwechsel sinnvoll ist: **(a)** SSV der Funktionen f1 und g auf dem ETS1 **(b)** SSV der Funktionen f2 und g auf dem ETS2 **(c)** SSV der Funktionen f2 und g auf dem ETS3 **(d)** SSV der Funktionen f1 und f2, wenn diese auf dem BTS läuft. Da die Differenz zwischen maximaler Stapelposition des BTS und Stapelposition des BTS beim Aufruf von f1 bzw. f2 *kleiner* ist als der SSV der Funktion f1 bzw. f2, wächst der SSV des BTS. Bei diesem beispielhaften Versuchsaufbau ist nicht trivial ob der SSV des Systems sinkt oder steigt. Um dies zu bestimmen sollte eine weitere Analysephase entscheiden ob f2 den Stapel wechseln sollte. Dabei muss gelten, dass der eingesparte Speicher von ETS2 und ETS3 zusammen größer ist, als der zusätzlich benötigte Speicher von f2, wenn diese auf dem BTS läuft. Sollte f2 auf dem BTS laufen, so kann es auch sinnvoll sein f1 auf dem BTS laufen zu lassen, da es nun einen größeren maximalen SSV auf dem BTS gibt. 34

4.5 36

TABELLENVERZEICHNIS

3.1 Taskwechseltabelle bei der Verwendung von SETs Mit ✓markierte Stellen werden bereits durch die Implementierung ohne SETs abgedeckt, mit ✗markierte Stellen wurden dem Betriebssystem für die Benutzung von SETs hinzugefügt. Da ein SET nur ein ET ist, der momentan auf dem BTS läuft, ist der Wechsel von einem SET zu einem *SET* äquivalent zu dem Wechsel von einem SET zu einem *ET*. 27

4.1 Gemessene SSV Werte im Fall 1. Die Werte sind in Byte angegeben. 30
4.2 Gemessene SSV Werte im Fall 1. Die Werte sind in Byte angegeben. 31
4.3 Gemessene SSV Werte im Fall 3. Die Werte sind in Byte angegeben. 32

QUELLCODEVERZEICHNIS

2.1 Beispiel für den Inhalt einer OIL Datei:
Es gibt einen Task T1 und ein Event E1.
Der Task hat eine statische Priorität der Größe 4 und wird direkt beim Betriebssystemstart in den Zustand 'bereit' versetzt. Da dem Task ein Event gehört ist es ein Extended Task.
Das Event hat eine Maske bestehend aus Eventname und Eventeigentümer T1, da diese Informationen aus dem Rest der OIL Datei eindeutig sind kann diese automatisch gesetzt werden. 12

3.1 Modifizierungen auf callee Seite um einen Stapelwechse zu unterstützen.
Die Zeilen 3,4,6 und 9 sind neu im Vergleich zu einem normalen Funktionsaufruf der GNU-ABI.
Das Register %esi steht fortan nicht mehr für allgemeine Operationen zur Verfügung, da damit die lokalen Variablen der Funktion angesprochen werden.
Zeile 8 ist lediglich modifiziert: Normalerweise liegen alle caller save Register vor dem %ebp, hier liegt aber der gesicherte Wert vom %esi nach dem %ebp.
Für IA32 gibt es für das standardmäßige Beenden einer Funktion die `leave` Instruktion, da diese hier nicht verwendet werden kann, werden diese aufgeteilt in Zeilen 8 und 10.
Somit gibt es 4 zusätzliche Instruktionen für den Stapelwechsel selbst und 1 zusätzliche, weil eine Standardinstruktion nicht verwendet werden kann. Also 5 zusätzliche Instruktionen insgesamt. 23

LITERATUR

[Fog15] Agner Fog. “Calling conventions for different C++ compilers and operating systems”. In: (2015), S. 17.

[Fou] Free Software Foundation. *Arrays of Variable Length*. URL: https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html (besucht am 17. 02. 2016).

[Gcc] “Using the GNU Compiler Collection For gcc version 4.9.3”. In: (2015), S. 739–742.

[HDL13] M. Hoffmann, C. Dietrich und D. Lohmann. “dOSEK: A Dependable RTOS for Automotive Applications”. In: *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*. 2013, S. 120–121. DOI: 10.1109/PRDC.2013.22.

[Hof+15] M. Hoffmann u. a. “dOSEK: the design and implementation of a dependability-oriented static embedded kernel”. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*. 2015, S. 259–270. DOI: 10.1109/RTAS.2015.7108449.

[Ia3] “Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D”. In: (2015), S. 73.

[Koe] Thomas Koenig. *CLOCK(3) Linux Programmer’s Manual*. URL: http://man7.org/linux/man-pages/man3/clock.3.html (besucht am 23. 02. 2016).

[Oil] “OSEK/VDX System Generation OIL: OSEK Implementation Language Version 2.5”. In: (2004). DOI: http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf.

[Ose] “OSEK/VDX Operating System Specification 2.2.3”. In: (2003). DOI: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[Sys] “SYSTEM V APPLICATION BINARY INTERFACE Intel386 Architecture Processor Supplement Fourth Edition”. In: (1997), S. 35–40.

LEBENSLAUF

Von: September 2000, Bis: Juli 2003 - Grundschule: “Siedlerschule” in Nürnberg
Von: September 2003, Bis: Juli 2004 - Grundschule: “Theodor-Billroth” in Nürnberg
Von: September 2005, Bis: Juni 2012 - Gymnasium: Hans-Sachs-Gymnasium in Nürnberg mit Abschluss der allgemeinen Hochschulreife
Seit: Oktober 2012 - Bachelorstudium: Universität Erlangen-Nürnberg, Bachelor of Science Informatik