Design of a OSEK/VDX-compatible System API for Linux

Study Thesis

by

Johannes Bauer

born December 6th, 1983 in Lichtenfels

Department of Computer Sciences 4
Distributed Systems and Operating Systems
University of Erlangen-Nuremberg

Tutors:

Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
Dipl. Inf. Michael Stilkerich
Dipl. Inf. Christian Wawersich

Begin: 1. Januar 2007
Submission: 30. September 2007

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.

.....

Erlangen, den 30. September 2007

Ich bin damit einverstanden, dass die Arbeit durch Dritte eingesehen und unter Wahrung urheberrechtlicher Grundsätze zitiert werden darf. Im Falle eine Aufbewahrung meiner Arbeit im Staatsarchiv erkläre ich mein Einverständnis, dass die Arbeit Benutzern zugänglich gemacht wird.

.....

Erlangen, den 30. September 2007

Abstract

Embedded real time systems often need to be optimized for high availability and deterministic runtime- and scheduling behavior. The OSEK-OS operating system standard is quite fit for this purpose: by means of the priority ceiling protocol many actions and properties of the system are already known before runtime allowing for a customized generation of the actual operating system code. For testing of functional properties of an OSEK-OS-conform operating system it is useful to test on a platform which has sophisticated debugging utilities available. A Linux system is suitable as most Linux distributions already innately include versatile debugging tools. This study thesis will evaluate the possibility of simulation of an OSEK-OS-conform operating system and it's mapping onto a UNIX-process.

After a brief explanation of how a OSEK operating system works the developed code generator called *josek* will be introduced. The method of operation and particularities *josek* will be discussed, paying special attention to the scheduler – the integral component of any OSEK operating system. It will be explained how a specially crafted stack is used in order to perform task switching and how this stack can be protected with userland means provided to any Linux-process. Problem cases which will appear during development of such an operating system are be illuminated and their solution is presented. This includes cases where special compiler optimizations might cause dysfunction of the generated code. After the study thesis has shown that and how it is possible to have functional components of an OSEK operating system emulated by a UNIX-process, the study thesis will be completed by a detailed performance review. Not only will the code generated by different configurations of *josek* be compared against itself, but it will also compare against *Trampoline*, another open source implementation of an OSEK operating system.

Zusammenfassung

Eingebettete Echtzeitsysteme müssen *per definitionem* darauf optimiert werden, eine hohe Verfügbarkeit zuzusichern und oft auch deterministisches Laufzeit- und Schedulingverhalten zeigen. Hierfür ist der Betriebssystemstandard OSEK-OS wie geschaffen: In OSEK-OS sind durch das Priority-Ceiling-Protokoll viele Eigenschaften und Vorgänge zur Zeit der Erzeugung des Betriebssystem bereits bekannt und können daher individuell optimiert generiert werden. Zum Testen funktionaler Eigenschaften eines OSEK-OS-konformen Betriebssystems ist es für den Entwickler von unschätzbarem Wert, das System auf einer mächtigeren Plattform als dem Zielsystem auszuführen, um Debugging-Vorgänge effizient ausführen zu können. Hierfür eignet sich ein Linux-System, da die allermeisten Distributionen bereits vielfältige und hoch entwickelte Debugging-Werkzeuge mitbringen. Diese Arbeit beschäftigt sich mit der Simulation eines OSEK-OS-konformen Betriebssystem und dessen Abbildung auf einen UNIX-Prozess.

Nach einer grundlegenden Beschreibung, wie ein OSEK-konformes Betriebssystem funktioniert wird detailliert der im Laufe der Arbeit entwickelte Codegenerator *josek* vorgestellt. Besondere Aufmerksamkeit gilt dem Scheduler und Optimierungen an diesem, da er den integralen Bestandteil eines jeden OSEK-Systems darstellt. Es wird erklärt, wie ein Stackaufbau und -schutz mit Mitteln, die einem Linux-Prozess zur Verfügung stehen, realisieren kann. Probleme, die es während der Entwicklung gibt, werden vorgestellt, diskutiert und eine Lösung präsentiert. Hierzu zählen beispielsweise Compileroptimierungen, die die Funktionsweise des erzeugten Codes beeinträchtigen können. Nachdem die Arbeit gezeigt hat, dass und wie ein OSEK-OS auf einen UNIX-Prozess abgebildet werden kann, wird die Arbeit durch Messungen abgerundet. Hierbei werden entstehende Performancedifferenzen zwischen verschiedenen *josek* Betriebssystemen ebenso erörtert wie ein Vergleich mit dem OpenSouce-Projekt *Trampoline*.

Contents

Chapter 1

Introduction

Many embedded operating systems rely on a lower layer which handles the communication with the underlying hardware. In the operating system layer terms like "stack pointer" or "instruction pointer" are unknown – only the far higher abstraction of tasks is relevant there. Tasks are basically functions which have a certain priority assigned and which can be scheduled by the operating system. The *means* of scheduling tasks is of course changing internal CPU-registers, but this is completely hidden to the top layer.

1.1 Motivation

One of the many alternatives in choosing such a low, hardware-dependent layer for an embedded operating system is OSEK. OSEK-OS has many advantages: it is a fully static, real time capable operating system with a freely available specification and also has been standardized by ISO 17356. Its small overhead in terms of memory usage and code footprint make it predestined for use in highly embedded systems. The actual problem is to find a free implementation of such an OSEK conform operating system – which is exactly the point where this study thesis comes into place. It shall provide the necessary academic background for understanding the OSEK operating system and will implement the OSEK-OS system API. Step by step the interesting points in development will be explained and reasons of challenging design decisions will be balanced. The final goal is developing a portable, open-source OSEK operating system generator, which can be used to interconnect an operating system like *KESO* with either the Linux API or any other operating system that the OSEK implementation supports.

1.2 Notation

- `Monospaced font` is used for the following purposes:
  - Code snippets: "[...] achieved by the `ret` assembly command."
  - Filenames: "This is described in the `RULES` file."
  - Commands: "We will use `sed` for patching the code."
  - OSEK system calls and OSEK hooks: "The `ActivateTask` system call may invoke the `ErrorHook`."
- Parts which will be printed *emphasized* are:
  - Names of software packages: "*josek* is written in *Java*."
  - Operating system task states: "The task changed from the *running*-state to *ready*."
- Citations are written with square braces: "Scheduling behavior is defined in [OSE05]."
- References to a certain OSEK task or resource are printed in French quotation marks: "Task »A« requires resource »X«."
- *Technical terms* are explained in the terminology section 1.3.

1.3 Terminology

- The time at which the actual OSEK operating system is being constituted by generating C-code will be referred to as *system generation time* or short *generation time*.
- The Intel x86 compatible machine will be called x86, or more specifically x86-32 throughout this document (called IA-32 by Intel). The AMD64 extensions to the x86 which are called "Intel 64" by Intel (formerly known as EM64T) will be referred to as x86-64. They are not to be confused with the completely different IA-64 architecture.
- Switches which can be defined in *josek* during system generation time in order to affect the generated code will be called *generation defines* or short *defines*, if unambiguous.
- Although this study thesis refers to an OSEK/VDX compatible operating system, it will be called OSEK for simplicity throughout this document. This is common practice and also done by the OSEK Group itself.[OSE04]

Chapter 2

OSEK

2.1 Description

The OSEK operating system specification describes an operating system interface which was designed for use in the automotive area. As many different vehicle distributors including Daimler-Benz, BMW, Volkswagen and Siemens Automotive called for one common, slim operating system interface, OSEK was developed to fulfill these needs. An OSEK operating system is intended for use in highly embedded systems – systems which typically provide only meager resources for the developer. As the whole system is limited in terms of RAM availability and/or text segment size, OSEK aims towards omitting unnecessary features which would be bloating the code.

OSEK introduces four so-called conformity classes. These are four different classes of operating systems (many of which are subsets of each other) specifying the minimum requirements the system needs to provide so it can be called OSEK compliant. They are provided for one sole purpose: optimization. For example, if it is known that there won't be any events in the system (or no *extended tasks* to speak in OSEK-terms) there is no need to have a *waiting*-state for tasks. Also, there is no need for memory allocation for event masks or delivered events, no need for the code resembling the event API and so on.

These highly optimized OSEK systems can most easily be built when using a code generator. Such a generator will be developed and explained in the course of this paper.

2.2 Priority Ceiling Protocol

The priority ceiling protocol is *the* integral part of an OSEK operating system. It is the method task priorities are changed during acquirement of resources so no

deadlocking can occur. For this to be achieved the OSEK system generator has to know in advance which task can occupy which resource or resources. This mapping is one piece of information present in the OIL file. Each single resource is then assigned a priority – its so-called *ceiling priority*. This is the same priority as the top priority task which can occupy this resource. To make it clearer, a small example:

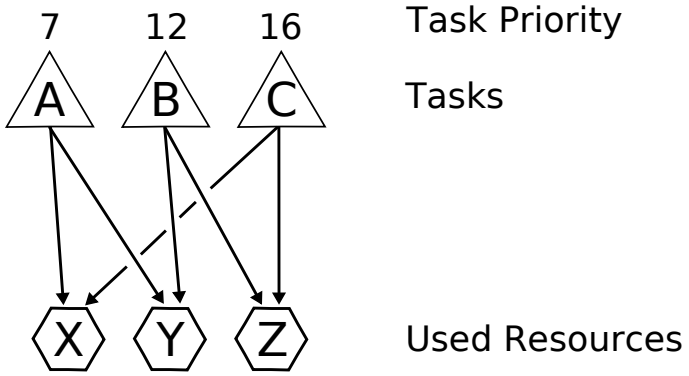

Figure 2.1: Tasks depending on resources

Figure 2.1 shows three tasks and three resources, arrows resembling the relationship "can occupy". Each task has its default priority, which also is defined in the OIL file. The code generator analyzes this graph and determines the ceiling priority of resources »X«, »Y« and »Z«. They are 16, 12 and 16 respectively.

When any of these tasks does acquire a declared resource, an atomic priority change happens: should the priority of the resource be higher than the current priority of the task, the task's priority will temporarily be raised to the priority of the resource.

This has one important implication which is essential to the functionality of the priority ceiling protocol: the priority of the task is higher than usual during the occupation of a resource. Therefore this task cannot be preempted by other tasks which might require this resource. Hence a possible deadlock-situation is avoided.

This is illustrated by figure 2.2. The task »A« is activated while the operating system is in "Idle" state. It is immediately scheduled with its default priority (7) as there are no other tasks in state "ready". It first acquires the resource »Y« and afterwards »X« which raises the task's current ceiling priority to 12 and 16, respectively. Then the system is interrupted by an interrupt service routine, which activates task »C«. Task »C« has default priority 16, but as currently task »A« runs with this same priority also, »C« cannot preempt »A«. Only after »A« releases resource »X« its ceiling priority is decreased back to 12. The scheduler immediately switches to task »C«, as it now has top priority. After »C« is finished

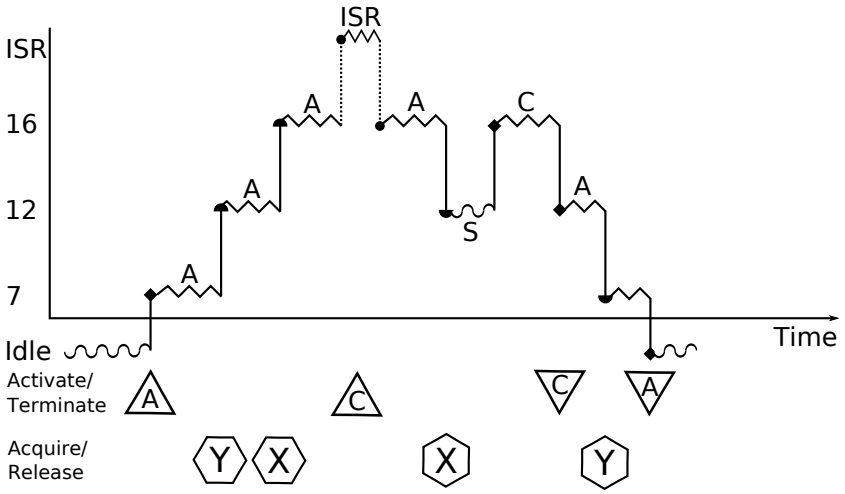

Figure 2.2: Acquirement of Resources using the Priority Ceiling Protocol

through the `TerminateTask` system call, »A« is scheduled again which releases its remaining resource »Y« and also terminates.

The priority ceiling protocol correlates with an OSEK operating system in two dimensions. On the one hand, special optimizations are possible when it is known in advance the priority ceiling protocol is in use – it will, for example, never occur that a task is preempted by another task of same priority. On the other hand the priority ceiling protocol can only be used when certain things like the priority of each single task is already known in advance (i.e. at system generation time). It is helpful to keep these two causal connections in mind during the development of an OSEK operating system.

2.3 Related Projects

2.3.1 openOSEK

openOSEK is one of the first projects which can be found when searching the Internet for a free OSEK implementation. It aims towards a slim implementation of a real-time-capable operating system for automotive applications.[ope07] The project seems to have one major problem, though: obviously, a top-down approach to the problem of creating an OSEK compliant operating system was used. This means that coding standards and preparations for OSEK-OS, OSEK-CAN, Time-Triggered-OSEK, OSEK-COM and OSEK-NM have already been taken. However, most of the code consists of stubs, no stable release has been made up to today. The SVN repository reveals that important components of *openOSEK*

have not been touched in a period of half a year, although obviously dysfunctional. The project has been bogged down in details: the developers provide commercial support, project donations, a Wiki, mailing lists, project forums, IRC channels and blogs – but no usable code.

2.3.2 Trampoline

The *Trampoline* project makes a solid impression upon its reviewer. It has the advantage of caring developers, a cleanly designed concept and frequent updates. Trampoline is split up into three major parts:

- *goil*, the OIL parser. It takes an OIL file as input and outputs a C-file and two header-files which resemble the configuration of the OSEK system. The *goil* parser also needs information about the destination platform. Currently the Infineon C167, PowerPC and Linux (through use of the `libpcl`[1]) are supported.[BBFT06]

- *viper*, the Virtual Processor Emulator. On a UNIX system it is implemented as an own process for target abstraction purposes. *viper* takes care of timers, interrupts and alarms. It communicates with the actual OSEK process via shared memory and asynchronous POSIX signals.

- *Trampoline* provides the OSEK-Kernel. The *Trampoline* code is not being generated (in contrast to *josek*), still it uses code-specific optimizations. For this purpose many preprocessor-statement are used.

The intermediate *viper* layer is introduced as it realatively hard to keep the actual OSEK operating system code portable among different architectures. Code which interferes with the processor (like controlling the sleep states), switches context or programms the interrupt controller is highly hardware dependent. Therefore these pieces of code are replaced by *viper* code when compiled for UNIX emulation mode. This *viper* code is then interpreted by the *viper* daemon and appropriate results are performed. The tasks code can therefore be interchanged between different architectures without any adaptions.

The minor disadvantages of Trampoline are its rather large memory footprint (e.g. any alarm needs 17 bytes, any counter needs 14 bytes on a 32-bit PPC [BBFT06]), the code readability and system speed. Because of many preprocessor-statements in kernel-code the reader of the code gets easily distracted from the integral parts. The system is relatively slow as it uses the *viper* hardware abstraction layer.

[1] Portable Coroutine Library

Chapter 3

josek

All the required considerations mentioned in 1.1 taken into account it is evident that one solution to the problem of creating a highly optimized OSEK operating system is writing an OSEK/VDX code generator. Thus, *josek* (short for "*Java* OSEK", pronounced /ˈdʒoʊ-sɛk/) was developed. Its characteristics will be illustrated in the next few sections.

3.1 Overview

josek is a *Java* application which basically does only two things: parse an OIL[1] file, which serves as input and then generate ANSI-C-code (with minor parts of assembly) which resembles the actual operating system. The resulting C-Code can then be compiled with, for example, the `gcc` compiler and then be linked against the actual application. Generating the code via a separate program has many advantages compared to a static solution:

- The code can be highly optimized to the actual problem. It is, for example, possible to have systems which work completely without using timers or alarms. Not only will in these special cases the text-segment become smaller (as the dead code is removed by the generating engine), but the variables which would be held in RAM can also be discarded, hence saving valuable resources.
- Carefully generated code is much more readable than code which has been statically written and which relies on many `#ifdef` preprocessor statements. These are easily processed by any preprocessor, but for humans they are, especially when nested, hard to understand.

[1] OSEK Interface Language

- Special features which have to be implemented in many parts of the resulting operating system (like a debugging feature or memory protection facilities for example) can easily be implemented when using a code generator. The resulting code is left completely unaffected when these features are deactivated. This kind of aspect-oriented approach makes the *generating* code (i.e. the *josek* source itself) more readable as coherent parts are combined in one single location.

3.2 Modular Concept

josek uses a modular concept to generate OSEK code. There are two ways of enhancing the features *josek* provides:

1. Placing code into the `arch/` directory and adding references to these added code files in the global `RULES` file. The `RULES` file will be parsed during generation time and the referenced code will be added at the appropriate locations. Parsing of code in the `RULES` file allows only for minor and relatively static adaptions or conditions (like the definition of a certain preprocessor symbol).

2. Adding a code hook: Code hooks are classes which are derived from interface `CodeHook`. They provide much more comfortable means of altering the generating code as all internal data structures at the time of generation are known to the hook and every imaginable *Java* construct can be used to influence the result.

3.3 Scheduler

The essential and integral part of an OSEK operating system is the scheduler. Therefore careful attention has to be drawn to implement the scheduler not only correctly, but also efficiently. As OSEK uses task priorities and the *Priority Ceiling Protocol* (explained in section 2.2 and also at [OSE05], chapter 8.5) to avoid priority inversion there also exist more than one scheduling queue with different priorities.

3.3.1 Work Principle

What the priority ceiling scheduler essentialy does after its invokation is:

- Determine if there are tasks in the *ready*-state (include one task which might currently be in the *running*-state).

- If there are, determine the most important of these tasks according to their priority.
- Schedule or dispatch the task with highest priority. Change the context from the currently running task to the new task, if applicable.

Special requirements that are necessary in an OSEK compliant operating system are:

- Honor the priority ceiling protocol. This includes calculation of resource priorities (which can be done in advance, i.e. at system generation time) and change of task priorities according to the resources held by that task.
- Handle multiple activation of tasks correctly. Should a task be activated more than once, it will also be run more than once (according to the required OSEK conformity class). The order in which tasks are run has to be the same as the order of activation. Imagine a system which consists of two tasks »A« and »B« of same priority. If the activation order is »A«, »B«, »A« then the tasks have to be executed in exactly that order. Rearrangement is, according to the OSEK specification, illegal.[OSE05]

To fulfill these needs, tasks are usually stored in a so-called scheduling queue. There are multiple queues, one for each priority present in the system. Only tasks in the *ready*-, *running*- or *waiting* state are enqueued, *suspended* tasks are not. In an operating system with no currently running tasks, all queues are empty:

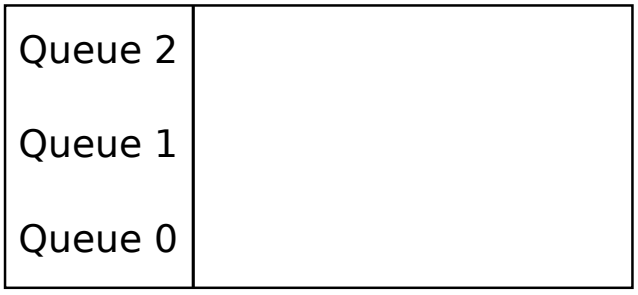

Figure 3.1: Scheduling Queue when Empty

When in this state a task is activated, it is inserted in the queue according to the task's default priority (figure 3.2.

Activation order is preserved because when other tasks are activated, they are inserted in the back of the priority queue as shown in figure 3.3.

Should a task then gain prirority by the acquirement of resources, it will additionally be inserted in the queue according to ceiling priority. In figure 3.4 task

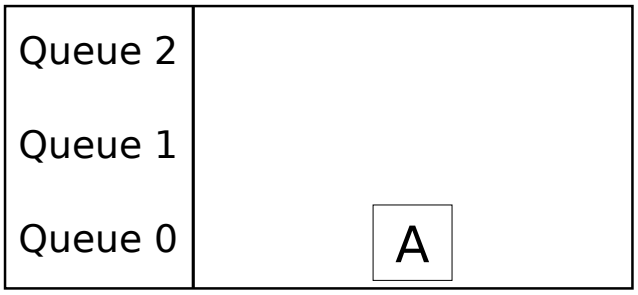

Figure 3.2: Single Task Enqueued with Priority 0

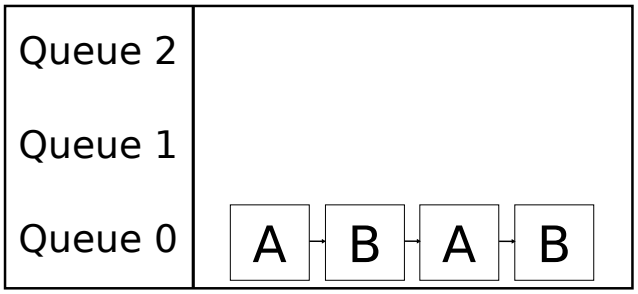

Figure 3.3: Multiple Activation

»A« acquires two resources and therefore first gains priority level 1 and afterwards priority 2. Tasks which gain priority due to resource acquiration are always entered in front of the queue. Conseuqently, when the held resources are released, the queue entry at the front of the queue is removed.

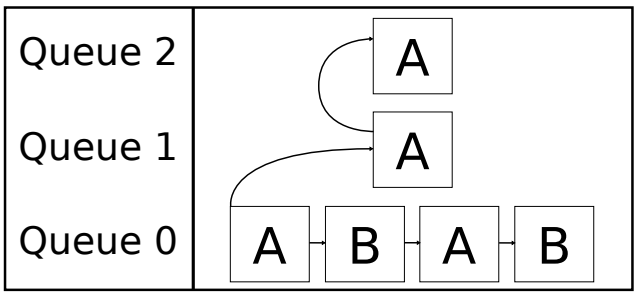

Figure 3.4: Priority Ceiling Protocol in Effect

In figure 3.5 can be seen what happens when a task of high priority is activated (task »C«) while another task is on its ceiling priority (task »A«): the recently activated task is inserted in the back of the queue and will not be scheduled until task »A« releases its resource. Through the release, the priority of »A« would decrease to 1 resulting in »C« with its default priority of 2 being the most important task in

the system. It would therefore preempt »A«.

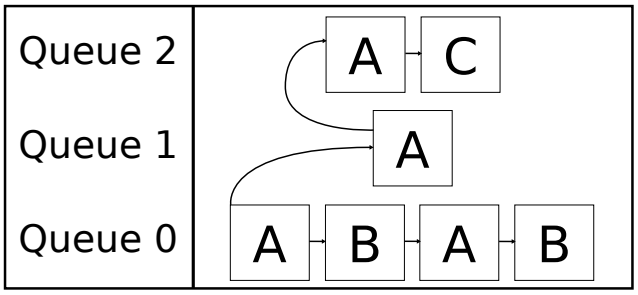

Figure 3.5: Activation after Priority Gain

More graphically speaking the currently running task can be determined by traversing the queues from top to bottom and in each queue from left to right. The running task is the first task encountered not in the *waiting* state.

3.3.2 Optimization of the Queue Length

An important optimization is to predict the exact maximum length of each of these queues. It is determined by two factors:

1. The number of multiple activations of a task in its base queue (the queue with the task's default priority)

2. The number of different resources the tasks can acquire to gain a higher priority

As the number of multiple activations are limited due to the OSEK conformance classes and the resources which any given task uses are known at the time of code-generation, an exactly fitting queue length can be constructed.

Consider the following example with tasks »A«, »B« and »C« and resources »X«, »Y« and »Z«:

The scheduler has its maximal load when all schedulable tasks have been activated the number of times permitted by the used conformance class of the OSEK operating system. Additionally all priority gains which are possible by the acquirement of all resources have to be calculated in this worst-case scenario.

However, it becomes soon evident that there are numerous optimizations with this kind of scheme. The first, obvious optimizations is possible because resources which would lead to a priority gain of tasks can only be occupied once:

Furthermore when it is the case that there is only one task per priority level, the correct but circuitous implementation of the scheduler can be deskilled. Multiple

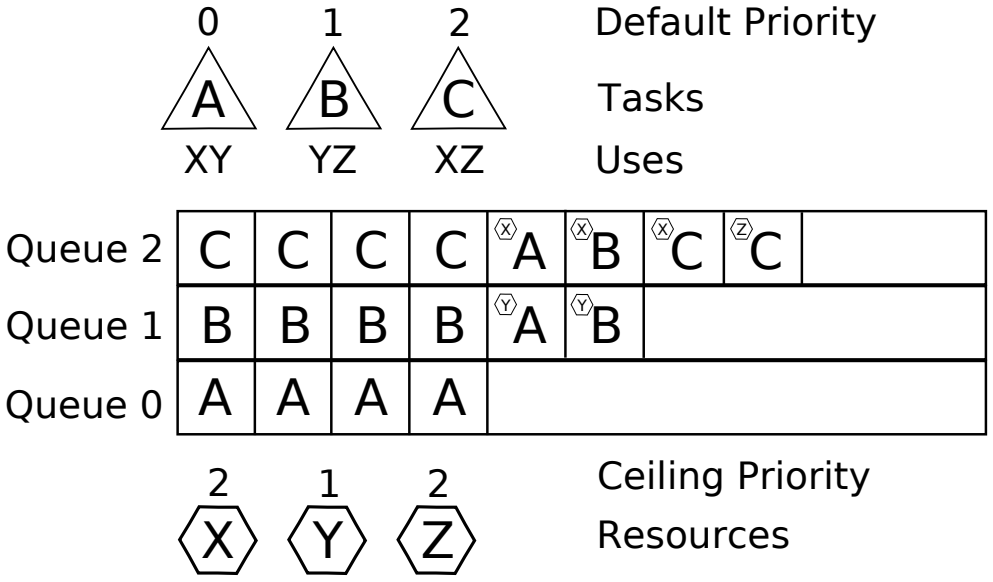

Figure 3.6: Simple task/resource example

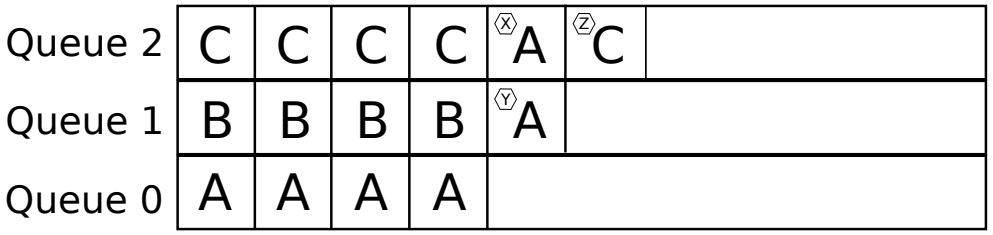

Figure 3.7: Resources occupied only once

activation of different tasks is not possible in this scenario – the tasks need not to be explicitly scheduled by putting an entry into the scheduling queue, but instead an activation counter can be used:

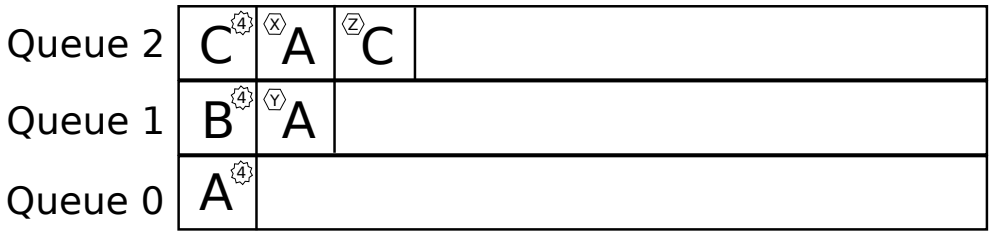

Figure 3.8: Resources occupied only once

Through these optimizations the scheduling queue size can be reduced drastically – in this small example from 18 entries of a naive implementation to a total of 6 entries.

Now that we know in advance how much memory the entries need to consume in the worst case it has to be determined how to store these entries: as a linked list or an array. Any naive implementation would probably immediately choose the linked list as a solution, as the linked list is the standard implementation of queues in which many queue/dequeue operations may occur. However there is a memory space penalty associated with this approach: taking a look at, for example, the AVR architecture it is clear that pointers are quite memory consuming – they're "expensive". Any pointer on an AVR consumes two bytes of RAM – one for each entry in the list. Considering the minimal size of 6 entries in the previous example still 12 bytes of memory would be wasted as data storage overhead. This is no acceptable implementation for an operating system which has highly embedded systems as its primary target platform.

There is a possibility of reducing the problem of memory consumption. Consider the following data structure:

```
1 struct bbqueue {
2     unsigned char write, size;
3     TaskType tids[QUEUE_LENGTH];
4 };
```

In this data type resembling a queue are no pointers used. Instead there is an index `write` denoting the next element in the `tids` array which can be written to. Additionally there is a counter `size` which yields the number of elements currently stored in this queue. Using this type of bounded buffer ensures that the overhead stays small (two bytes in total), yet it has little cost of inserting elements at the queue's ends:

```
1  void insert_front(struct bbqueue &q, TaskType new) {
2      q->tids[(q->write - q->size - 1 + QUEUE_SIZE)
3                  % QUEUE_SIZE] = new;
4      (q->size)++;
5  }

7  void insert_back(struct bbqueue &q, TaskType new) {
8      q->tids[(q->write)++] = new;
9      (q->size)++;
10 }
```

Removing elements from the queue's back performs even better:

```
1  void remove_front(struct bbqueue &q) {
2      (q->size)--;
3  }

5  void remove_back(struct bbqueue &q) {
6      (q->write)--;
7      (q->size)--;
8  }
```

The only operation which would be quite costly is inserting or removing elements from the middle of the queue. This will, however, because of the nature of the scheduling queues in an OSEK conforming operating system, never happen. The data structure is therefore very well suited for use as a OSEK scheduler queue.

3.3.3 $O(1)$ Scheduler

The currently presented and optimized scheduler has a performance of $O(n)$ with n being the number of scheduling queues. This can be further reduced on some machines to have a constant cost of $O(1)$. As it is somewhat machine specific (although many architectures provide the necessary armamentarium) this optimization has not made it into the final *josek* code. However as it would be a nice improvement it will be quickly discussed.

The following loop, which triggers the $O(n)$ cost, is embedded in the generated code:

```
1  for (priority = NUMBER_PRIORITIES - 1; priority >= 0;
       priority--) {
2      /* Check if queue is nonempty */
3      [...]
4  }
```

This can be simplified in the following way: for each enqueuing process, do a bitwise `OR` with a global variable and the bitvalue of the queue number. After each dequeuing which clears the queue, do a bitwise `AND` with the same global variable and the complement of the bitvalue of the queue number. To make it clearer:

```
1 /* Enqueue*/
2 active_queues |= (1 << queue_nr);

4 /* Dequeue */
5 if (!queuesize[queue_nr]) active_queues &= ~(1 <<
      queue_nr);
```

When this is done, the queue selection algorithm can be replaced by a variant showing far greater performance:

```
1 if (!queue_nr) return;          /* All queues empty */
2 asm("bsr %1, %0\n" : "=r"(priority) : "r"(queue_nr));

4 /* Highest priority queue is now held by "priority" */
5 [...]
```

So by using the special `bsr` (Bit Search Reverse) operation [Int07a] the performance while searching all queues can be significantly reduced. When the number of available priority queues exceeds 32 (or 64 on x86-64, respectively) however, an additional check is necessary and could be implemented like:

```
1 highqueue = 3;
2 for (i = 2; i >= 0; i--) {
3     if (queue_nr[i]) highqueue = i;
4     break;
5 }
6 if (highqueue == 3) return;     /* All queues empty */
7 asm("bsr %1, %0\n" : "=r"(priority) : "r"(queue_nr[
      highqueue]));
```

The presented code would allow for up to 96 different priority levels. Although it contains a tiny loop (which will get inlined by any good compiler) it still is a great win over the naive solution.

3.4 Alarms and Counters

The lowest conformance class that OSEK requires states that any OSEK operating systems needs to provide at least one alarm. Alarms are based on counters. The

implementation of these counters is very platform-specific. More concretely, a Linux process has only one possibility of mapping counters to operating system calls: the `SIGALRM` signal delivered by the kernel after a `alarm()` or `setitimer()` system call. Any quick look into the man page will make the reader realize that `alarm()` disqualifies for the purpose of serving counter ticks as it's granularity is much too coarse: one second. Therefore only `setitimer()` can be used.

The man page of the `setitimer()` system call explains that any Linux process is provided three Timers which vary in their functionality: one counts real time (`ITIMER_REAL`), another counts virtual time (time on which the process is scheduled in user space, `ITIMER_VIRTUAL`) and a last one which counts the time in which the process is scheduled plus the time in which the operating system performs system calls for that process (`ITIMER_PROF`).

This again means that there is only one useful timer available: `ITIMER_REAL`. Therefore for any OSEK task which needs more than one alarm, a mapping of many internal alarms to this system alarm has to be performed. It is realized by the *josek* counters.

Counters are handled in a very simple way: they're represented by a global variable counting every tick. The tick interval is known at system generation time and is specified in the OIL file. During generation of the OSEK operating system, the greatest common denominator over all counter intervals is calculated. This will later on during runtime pose the interval for the `setitimer()` system call – we will refer to is as t_{min}. Each counter also has a property called `CntrTickMaxValue` which essentially is $\frac{t_{counter}}{t_{min}}$. This value will not change during runtime and can therefore be stored either in RAM or in the text-segment. Refer to section 4.1 for further reference on how this is done.

During the execution of the `setitimer()` handler, a tick count is increased for every counter available in the system. When the tick count reaches the `CntrTickMaxValue`, a "real" counter tick is triggered. This will cause the `CntrCurrentValue` variable to be increased, which wraps if it reaches `CntrMaxValue` – a variable also specified during generation time. Should the `CntrCurrentValue` wrap it will trigger possible alarms dependent on it.

When an alarm is finally triggered, it does what it is supposed to to – one of three possible actions:

- *Callback function*: Probably the most common action after an alarm was triggered is calling a callback function. This function has to be defined in the operating system using the `ALARMCALLBACK` keyword.

- *Setting an event*: The alarm can set an event for a certain task. For this, the alarm simply calls `SetEvent`.

- *Activating a task*: If requested, the alarm will use the system call `ActivateTask`

in order to activate a certain task.

3.5 Event Handling

In order to be able to deliver and receive events the `taskdesc` structure of any generated *josek* operating system can be extended by two values: the event *mask* and *wait* values. *josek* automatically decides whether these values are needed and how large they need to be according to how many events are used in the system – if they are used at all.

Handling of events then is straightforward. A task clearing its event mask through the `ClearEvent` system call will directly write to the `mask` value:

```
1 StatusType ClearEvent(EventMaskType m) {
2     taskdesc[current_task].mask &= ~m;
3 }
```

When a task delivers an event, an analogous piece of code is executed, except that the bitwise `OR` is being used with the mask and a rescheduling point is introduced should the destination task have been woken up through delivery of this event.

Should a task want to wait for an event through usage of the `WaitEvent` system call, the *wait* value is set and the state is set to *waiting* if this event has not yet been delivered, i.e. if

```
1 taskdesc[current_task].wait & taskdesc[current_task].
    mask
```

yields the value 0. If the task's state is changed to the *waiting*-state again a rescheduling point is activated afterwards. The scheduler will then switch to a *ready* task (or idle, if none is available).

3.6 Interrupt Mapping

When mapping hardware IRQs onto a UNIX compatible system there is basically only one choice for asynchronous, userland process interruptions: POSIX signals.

This is what *josek* does: Simulating interrupts by a signal handler for `SIGUSR1` and `SIGUSR2`. As the introduction of asynchronous program interruptions always brings along concurrency close attention has to be paid that all data structures are locked before the ISR-code is invoked. As the OSEK specification does not permit many system calls to be invoked from within an ISR this is no big problem. In particular the following system calls have to be interlocked against an ISR interruption:

- `SetEvent` and `ClearEvent` have to be locked, as atomic access to the `taskdesc[n].mask` and `taskdesc[n].wait` variables has to be guaranteed.
- `Schedule` has to lock its access to the task queue as execution of the `ActivateTask` system call from within the ISR could corrupt this data structure.

Apart from these difficulties, the signal handler code simply executes the appropriate ISR handler upon decision on which signal was received.

3.7 Rescheduling Points

When designing an OSEK-OS compliant operating system it is important to investigate exactly at which points in time rescheduling is necessary – and why:

- `Schedule`: A manual rescheduling point obviously occurs when the `Schedule` system call is invoked.
- `ActivateTask`: When a new task has a transition to the *ready*-state, the OSEK operating system may determine that it is more important than the currently running task. The context will have to be changed to the recently activated task.
- `SetEvent`: Upon setting an event a currently blocked task might change from the *waiting*-state to *ready*. If this task is of higher priority than the currently running task, once again the context will change.
- `WaitEvent`: Analogously to `SetEvent` a task may go into the *waiting*-state after executing the `WaitEvent` system call. It won't, of course, if the event mask has already been set. But when it does, another task will have to be scheduled.
- `ReleaseResource`: When releasing a resource the currently running task might lose of priority as the resource's ceiling priority might be higher than the priority of the task prior to its acquirement. It may then happen that a switch occurs to another task.
- `TerminateTask`: After the currently running task has changed to the *suspended*-state the operating system must determine which task to schedule next.
- `ChainTask`: When the current task is exiting and a switch to another task is requested, the scheduler is invoked. However, `ChainTask` requires no ordinary rescheduling point. In fact, the OSEK-OS specification states that

a call of `ChainTask` will have *immediate* effect on the state of the requested task instead of resulting in multiple requests. Consider a currently running task »A« has performed an `ActivateTask` call on a task »B« having the same priority. Afterwards it calls `ChainTask(A)`. The next running task will be »A«, not »B«.

- Initialization: After the system initialization and entering of the idle-loop the scheduler needs to be invoked to bootstrap the whole system.

3.8 Hooks

The OSEK/VDX operating system specification provides a convenient means for the application developer to integrate preparation or clean-up-tasks into the environment: operating system hooks. These hooks are basically functions which are executed at special times. They are valid system wide and the times they "fire" can be one or more of the following:

- `ErrorHook`: Fires after a system call was completed unsuccessfully, but before the switch back to the task level.
- `PreTaskHook`: Fires before a task is scheduled but already in the context of the new task.
- `PostTaskHook`: Fires prior to a task being scheduled away, but still in the old task's context.
- `StartupHook`: Fires before initialization of the operating system, but before the scheduler is running.
- `ShutdownHook`: Fires when `ShutdownOS` is called to shut the operating system down.

These hooks seem pretty straightforward to implement, but the exact circumstances on how and when they are invoked make the implementing code a little bit tricky. The easiest of these calls is probably the `ErrorHook`. The only thing the generator needs to check is if the user wants an `ErrorHook` installed and insert a simple `if`-clause at the finalization points (i.e. before every `return` statement) which conditionally calls the hook.

The `PreTaskHook` and `PostTaskHook` are a little bit more complicated. This is because the specification states that they both need to be executed in the context of the task to be scheduled or of the task which is currently being left, respectively. The first naive approach would probably be inserting `PreTaskHook` calls at the

beginning of each `TASK()` block and inserting `PostTaskHook` calls at the end of these blocks. Thinking about this approach – parsing the C-file – will certainly yield the result that this is far more complicated than one would assume at first. Although insertion of the `PreTaskHook` calls is straightforward, insertion of the `PostTaskHook` calls is not – they have to be inserted *before* any `TerminateTask`, `ChainTask` or `ActivateTask` calls. And what about the user redefining some macros, which is perfectly legal? The hooks also need to be called when a task is scheduled away from or scheduled to – not just the first time at startup and at the `TerminateTask` system call.

Realizing that this approach is a dead-end road, how about calling these hooks from the scheduler? The scheduler is the part of the system which knows if a task change is necessary. This approach works pretty well with the exception of two aspects:

- The user might call an `ActivateTask` in the `PreTaskHook`. This will again case the scheduler to be invoked. Therefore care has to be taken that the scheduler does not call the `PreTaskHook` from an inconsistent state.
- The user might want to access the task ID of the task which is scheduled to or scheduled away from, respectively. This variable has to be properly initialized in the scheduler before calling the hooks.

Using this method works well if attention is paid to these aspects – this is also how it is done in *josek*.

The problem with the implementation of the `StartupHook` isn't evident at first glance, either. It looks as if it would be as simple as inserting a call to the hook upon operating system initialization. And it is, almost. Consider a system which has two tasks called »subsidiary« having priority 1 and »important« having priority 10. Then there is the following `StartupHook`:

```
1 void StartupHook() {
2     ActivateTask(subsidiary);
3     ActivateTask(important);
4 }
```

When this piece of code is executed, the task with low priority, »subsidiary« is scheduled first, leaving »important« waiting for its turn although it has higher priority. It seems acceptable, as when you come to think of it »subsidiary« is activated *before* »important«. A closer look into the OSEK operating system specification will reveal, however, that the scheduler has not yet been activated when the `StartupHook` is called. It will only become active after *both* `ActivateTask` system calls have finished – the scheduler will then of course schedule »important« first and priority inversion is avoided.

The `ShutdownHook` is as straightforward as it seems: a simple call to this hook in the body of the `ShutdownOS` system call is fully sufficient.

Chapter 4

josek internals

4.1 Optimization on Arrays

During development it will occur that a table of static values has to be accessible from inside *josek*. Simply using a global array of values has the advantage of being easy to implement and fast lookup. However the lookup is fast at a price: the values are kept in RAM. This might pose a problem to embedded systems which have limited resources. Especially when these values are never changed during the course of the program, preserving RAM space is one big goal which needs to be achieved when writing an operating system for highly embedded devices. Although of minor importance when testing functional components of the OS on an x86 with hundreds of megabytes of RAM it is a real drawback when using a microcontroller unit which only provides 128 bytes of SRAM.

josek has a solution: during the generation of the operating system's code a generation define (`saveram`) will decide how static arrays of values are stored: without any optimization as a global array or as a lookup-table. The lookup-table basically is a lookup-function which has the sole purpose of running input data through a switch/case statement which determines the appropriate return value.

The penalty is evident: a function call is more costly in terms of CPU time than a memory lookup. Registers have to be saved and the lookup of an arbitrary value in an equally distributed and compiler-optimized switch/case statement is of order $O(\log n)$ as opposed to $O(1)$ when using a global array. What is preferred has to be decided from case to case – it's just one opportunity *josek* provides. The last decision has to be made by the developer who generates the operating system.

4.2 Configuration Parsing and Internal Data Structures

For parsing the input configuration file, the OIL[1], code generated by *JavaCC*[2] is being used. It is based on a `.jj` file resembling the OIL grammar. For this purpose a modified version of the KESO grammar was used – actually the OIL grammar is a bit easier to implement as fewer recursions of objects are permitted.

The parser code generates – granted the OIL file is syntactically correct – a tree according to the input data. This tree is represented by the *Java* class `Configuration`. It can differentiate different attribute types such as boolean, integer, float or string. The OIL configuration file is bijectively mapped onto this configuration-tree. After the parser is done, the raw first pass is complete.

Afterwards, the configuration data needs to refined and a so-called `HighLevelConfiguration` object is created. This object parses the simple configuration tree for *logical* objects. It can, in contrast to the simple configuration, also detect logical errors such as a task referring to an undefined resource. In this parsing step, all logical objects are parsed specifically according to their jobs in the OSEK operating system to be generated. These objects are:

- Alarms
- Counters
- Events
- Hooks
- Resources
- Tasks

Each single of these objects has very specific member variables. Settings which have meaning to the high level configuration (like the `CYCLETIME` of an alarm, for example) are being taken from the simple configuration, values without meaning are simply being discarded without emitting any error or warning.

4.3 Stack Creation

Before changing the stack pointer to a new memory area it has to be assured that this area has been properly initialized. The piece of memory which the stack

[1] OSEK Interface Language
[2] Java Compiler Compiler

pointer will point to after its modification is a global array. It has to be well-prepared so the assembly instructions which are executed after its change find sane values to work with. These instructions are for an x86 machine `popa`, `popf` and `ret` in order of execution.[Int07b] Hence there have to be 32 bytes for the `popa` instruction, 4 bytes for the `popf` instruction which pops the `EFLAGS` register from the stack and 4 bytes for the return address.[Int07a] Therefore the complete context size (accessible through the variable `CTXSIZE`) is 40 bytes.

Note that during this context change the contents of the floating point registers and floating point control values are *not* saved – a context change from code which uses the FPU to another context which uses the FPU is therefore note possible without a race condition.

The reasoning behind this is the following:

1. *josek* has been developed having systems with tiny resources in mind. These embedded systems often do not have any FPU at all – floating point support seems unnecessary for a real-world embedded-system scenario.

2. The context of the FPU on an x86-32 machine has a size of 108 bytes. This is almost three times the size of a "regular" context.

3. As the FPU context contains control registers which have to be set correctly the creation of an initial stack becomes more difficult. This would be a detail that great attention would have to be paid to and it would therefore distract any reader of the code from the important design aspects.

Should an implementation of an FPU stack save be necessary after all this can easily be achieved afterwards by insertion of two assembly statement for each context saving and restoring. The x86-32 architecture provides two assembly instructions: they need pointers to allocated memory as an input and will save or restore the FPU context to or from there, respectively.[Int07b] The code to be inserted would be

```
1 sub $108, %esp            frstor (%esp)
2 fsave (%esp)              add $108, %esp
```

This code will reserve space on the stack and store the FPU registers there ("`pushfpu`") or load the FPU registers from the stack and free this stack space ("`popfpu`").

Moreover, not only the FPU registers are not saved, but none of the extended x86-operation registers are. This includes the registers provided by special CPU features like MMX, MMXExt, SSE, SSE2, 3dNow and 3dNowExt. The reasoning is obvious: these instructions are not only overkill for any embedded application, they're also absolutely machine-dependent. This is not the intended use of *josek*.

4.4 Stack Protection

A problem commonly found in programs which used fixed stack sizes – as this is the case in any OSEK operating system – is stack overrun. Due to the fact that the stack cannot grow dynamically according to the underlying program's needs it will at some point hit the boundary. This can especially occur by a too high nesting level of functions: as each function call uses some stack memory (at least 4 bytes for the return address of the function, usually 8 bytes on an x86 because the old frame pointer `%ebp` is also saved) the problem particularly arises when using recursive functions.

Luckily, this problem can be detected when running the OSEK operating system on Linux quite comfortably using the `mprotect` system call. When generating the OSEK-OS with the generation define `stackprotector` four things are being done:

1. The stack sizes of tasks are being rounded up to multiples of the page size (usually 4096 bytes).

2. In between each of the tasks' stacks a separate protector memory area is inserted which has exactly the size of one page.

3. All stacks and pages are being aligned at page boundaries. This is a compiler-specific parameter. Using the `gcc` compiler it can be achieved using the `__attribute__ ((aligned (4096))` attribute extension.

4. The protector-pages are marked as inaccessible through invocation of the Linux-specific `mprotect` system call. They are marked as `PROT_NONE` which means the page has no read, write or execute privileges.

As soon as these steps are performed any access beyond the tasks' stack memory (but within the size of one page) leads to an immediate synchronous program termination (`SIGSEGV`, segmentation violation). This can be easily traced to determine where the illegal access occurred.

This is, of course, no exhaustive memory protection. Any task can access the whole system memory through use of arbitrary pointers. Hence, such deliberate memory corruptions are not detectable. However, it is very well suited for detection of unintentional stack overflow as is the case with a too high nesting level of functions.

The drawbacks of this type of memory protection are evident:

1. It is highly Linux-specific.

2. It only detects memory corruption within one page size from the top or bottom of the tasks' stack.

3. For a number of n tasks, $n+1$ pages are wasted.

The greatest problem is probably portability to non-Linux architectures. Therefore a "poor man's version" of the memory protection is available by use of the generation define `stackprotector_simple`. When it is in use again memory pages separate tasks' stacks, but this time they're not protected using `mprotect` but initialized with a sequence of pseudo-random numbers. As the sequence is deterministic it can easily be verified at any time through a call of the `check_stackprotector()` function. Although this method is more portable it can not detect reading of invalid memory, only writing. Another drawback is that memory corruption is only perceptible afterwards, not synchronous to the actual memory access. Debugging is therefore slightly more complicated.

4.5 Context Switching

4.5.1 Method of Operation

The most important internal part of a preemptive operating system is the context switching, which is used by the scheduler. There are two possible context switching methods available which differ in functionality:

- `dispatch`: Starts a task for the first time. The stack pointer set to a piece of memory which was specially crafted for stack dispatching (explained in section 4.3). The registers and CPU flags are `pop`ped from the stack – these need to be set to zero to avoid accidentally setting flags which might trigger traps. Afterwards a `ret` is performed which jumps to the address on very top of the stack. This is usually the function pointer of the task which gets dispatched.

- `switchtask`: This will switch from a currently active task to another task. In order to be able to resume the currently running task appropriately, the first action is saving the processor flags and registers by `push`ing them onto the stack. Afterwards the current stack pointer is copied into the global `taskdesc` structure. The stack pointer of the task to be switched to is then restored from the codetaskdesc structure. CPU flags and registers are `pop`ped from the stack and a `ret` is performed, returning to the exact position the task was previously preempted from.

4.5.2 Pitfalls

Manipulating the stack pointer directly is hazardous business: the operating system does not condone any off-by-one-errors; should the stack pointer which is restored be off by a single byte relative to the correct position it is likely the entire system will crash. In such a case the accompanying stackframe is lost implying development tools like debuggers become useless.

Then there are more subtle pitfalls, which can be caused by the compiler being used. Any function call which occurs at the very end of a function can be optimized away and replaced by an unconditional jump. This is called a tail-call optimization.[Sch07] Essentially the following code:

```
1  dostuff:
2      push %rbp          ; Enter
3      mov %rsp, %rbp

5      [...]

7      call finalize

9      mov %rbp, %rsp     ; Leave
10     pop %rbp
11 ret
```

Will be replaced by this optimized version:

```
1  dostuff:
2      push %rbp          ; Enter
3      mov %rsp, %rbp

5      [...]

7      mov %rbp, %rsp     ; Leave
8      pop %rbp
9  jmp finalize
```

The `ret` instruction in the function `finalize` will therefore directly return into `dostuff`'s parent function instead of first returning into `dostuff` and from there on returning to the caller. The `dostuff` stackframe is bypassed this way – CPU time is saved.

It's a horrible optimization scenario for anyone directly manipulating the stack pointer, however. The reason is easy: it may be the case that `dostuff` has no parent, for example when it is executed on a specially crafted stack – as it is exactly the case in *josek*. When the optimization is activated the `ret` will jump

to some undefined address and the whole operating system will in all probability crash.

This is the case when using any recent version of *gcc* using more than `-O1` optimization. As soon as `-O2` the optimization flag `-foptimize-sibling-calls` is implied resulting in the described optimization. The naive solution would be to pass `-fno-optimize-sibling-calls` to the compiler flags of the task source file. This is extremely simple and effective, it has the disadvantage of slowing down other code, however, which is unaffected by the optimization.

josek uses a more tricky approach. First, all fragile functions need to be determined – by that is meant all functions which are broken by the tail-call-optimization. These are all functions which have no parent (i.e. no caller): it solely applies to tasks. Thinking about what functions tasks may call at their tail is the next target to identify. Only few are relevant:

- `ShutdownOS`
- `TerminateTask`
- `ChainTask`

Calling any other function at the end of a task is illegal according to the OSEK-OS system specification and therefore undefined behavior – the operating system crashing – is a perfectly legal consequence. `ShutdownOS` is uncritical as it will never return at all. Any optimization is fine here.

But things are different with `ChainTask` or `TerminateTask`. `ChainTask` also works with the optimization as long as *another* task is chained. Should a task chain itself, a return into the current context is needed and the `ret` will jump into nirvana. `TerminateTask` is also fine as long as the next task to be scheduled is not currently active. Should it be, maybe through multiple activation, it will crash in exactly the same way.

The problem can be avoided by a trick: first, the system calls `ChainTask` and `TerminateTask` are renamed to `__ChainTask` and `__TerminateTask`. Then two macros of the following kind are inserted into the `os.h` file:

```
1 #define TerminateTask()  do { __TerminateTask(); \
2                               __asm__ volatile(""); \
3                             } while(0)
```

This will cause any call to `TerminateTask` to be replaced by its actual call and an (empty) volatile assembly statement afterwards. The compiler detects code after the `TerminateTask` call – out `asm` statement. Actually there is *no* code, but we marked the inline assembly volatile so the compiler must assume the worst case

and is not allowed to move the code away either. It will therefore just do a plain `call` to `TerminateTask` and the problem described vanishes. The exact same approach applies to the `ChainTask` function which will also solve the described problems there.

4.6 Further Optimizations

There are a few tweaks which can be used to make the generated code perform slightly better than usual. Some are compiler specific, however, and will, if so, refer to recent versions of the *gcc*.

4.6.1 Stack Sharing

A straitforward optimization is stack sharing. It is possible as a direct consequence of the usage of the priority ceiling protocol. The protocol ensures, as explained in section 2.2, that always the task with the highes priority is in the *running*-state. As a matter of fact it can safely assumed that a task will never be preempted by another task of same or lower priority. In particular it is not possible that a task is preempted by a second activation of itself. It is a most elemental optimization that only each task is assigned only one piece of memory for use as a stack. But it is also possible that two *different* tasks share the same piece of memory, if they are mutually exclusive. This is always possible with tasks which have the same default priority. This shared stack has to be sized to fit both task's needs, of course. This is why it has to be sized according to the most greedy (in terms of memory usage) task's requirements.

4.6.2 `const` Functions

The *GNU Compiler Collection* can make use of the `const` function attribute, which tells the compiler that the attributed function's return value will solely depend on the parameters of the function.[StGDC07] It will never depend on the state of the program (i.e. not read global variables). If the compiler knows this about a function, it is a safe optimization to call the function less often than actually required by the source code. In practice this means that subsequent calls to the function in question will always yield the same return value as long as they get the same input. This allows the *gcc* to optimize these subsequent calls away. A good example of a function which should definitely be declared `const` are lookup tables which are stored in the text segment as described in section 4.1.

To demonstrate the effect, consider the following piece of C code:

```
1 printf("%d\n", CntrMaxValue(x));
2 printf("%d\n", CntrMaxValue(x));
```

It actually requires two calls to the `CntrMaxValue` function. So without having this function declared `const`, the following code will be generated:

```
1  movzbl %spl, %ebx

3  mov    %ebx, %edi                    ; 1st Parameter
4  callq  400eb1 <CntrMaxValueValues>

6  movzbl %al, %esi                     ; Value
7  mov    $0x40129d, %edi               ; String reference
8  mov    $0x0, %eax
9  callq  4006c0 <printf@plt>

11 mov    %ebx, %edi                    ; 1st Parameter
12 callq  400eb1 <CntrMaxValueValues>

14 movzbl %al, %esi                     ; Value
15 mov    $0x40129d, %edi               ; String reference
16 mov    $0x0, %eax
17 callq  4006c0 <printf@plt>
```

It can be clearly seen that the call to `CntrMaxValueValues`, which is the function that the `CntrMaxValue` macro expands to, is called twice. It takes `%edi` as input and returns its value in register `%al`. This needs to be done twice, as the *gcc* does not the function will return the same value each call for same values of `%edi`. When the optimization is active, however, the following code is generated:

```
1  movzbl %spl, %edi                    ; 1st Parameter
2  callq  400eb1 <CntrMaxValueValues>

4  movzbl %al, %ebx

6  mov    %ebx, %esi                    ; Value
7  mov    $0x40129d, %edi               ; String reference
8  mov    $0x0, %eax
9  callq  4006c0 <printf@plt>

11 mov    %ebx, %esi                    ; Value
12 mov    $0x40129d, %edi               ; String reference
13 mov    $0x0, %eax
14 callq  4006c0 <printf@plt>
```

The function `CntrMaxValueValues` is also called, but this time only once. Its return value `%al` is saved to register `%ebx` from where it is passed on to the two `printf` function calls.

4.6.3 `noreturn` Functions

There are functions in any OSEK operating system which will never return. These are:

- `ShutdownOS`
- `ChainTask`
- `TerminateTask`

When the compiler knows a function will never return, it can optimize away any cleanup-work which would be usually be done. This means, for example, the following code:

```
1 00000000004007c0 <JOSEK_TASK_job5>:
2 4007c0: 48 83 ec 08     sub    $0x8, %rsp
3 4007c4: 31 c0           xor    %eax, %eax
4 4007c6: e8 55 07 00 00  callq  400f20 <__TerminateTask>

6                         ; Free stack and return:
7 4007cb: 48 83 c4 08     add    $0x8, %rsp
8 4007cf: c3              retq
```

Will become optimized in a fashion that the two instructions after the call to `TerminateTask` are simply dropped.

Chapter 5

Running KESO on josek

5.1 Integration into a UNIX conform operating system

In order to have the possibility to access UNIX device drivers from within *KESO* it is necessary to map some UNIX system calls onto a *KESO*-interface. This is why the *PosixIO* component was added to *KESO*. It provides the following functions, of which *Java* prototypes are shown:

- `int open(String path, int how)`
- `int read(int fd, char[] buf, int size)`
- `int write(int fd, char[] buf, int size)`
- `void close(int fd)`

Making use of these functions which are directly mapped onto the corresponding UNIX system calls makes it possible to open a UNIX device and write or read data from it. This was tested using a Linux-compatible CAN controller. The CAN controller provides a character device (`/dev/can0`) for bus communication. When any *KESO* program wants to send data over the bus, it simply writes a specially crafted structure to this character device. Once the write is complete, the CAN bus packet is already sent over the bus. Reading works by a polling `read` system call on the driver device. If there is no data available, the read call will not block, but return unsuccessfully immediately, indicating by its return code no data was available.

Chapter 6

Performance Analysis

6.1 Test Conditions

For testing purposes two systems were used, a Pentium M 1400 MHz for all x86 measurements and an Athlon64 3700+ for x86-64 measurements. As both systems are hardly comparable, there is an additional variable which scales the timings down. This variable has been determined by a small assembly routine essentially executing a lot of no operation (`nop`) operations in a row and counting how many can be executed per second. This value is $2.517 \cdot 10^9$ for the x86-32 and $5.973 \cdot 10^9$ for the x86-64 architecture. The reference architecture will, in any cases, be the x86-64. So when any x86-32 timings are normalized, they will be divided by the factor of $f = \frac{5.973 \cdot 10^9}{2.517 \cdot 10^9} \approx 2.373$, when the throughput (MOps per second) of any function will be normalized, they will be multiplied by f.

To measure the throughput all functions which were evaluated included a counter to a global variable. This counter was incremented every time a task unit was completed (e.g. for every `ChainTask` call or for every `ActivateTask`/`TerminateTask` combination). After a period of at least 100 seconds the process was interrupted via a asynchronous POSIX signal. The signal handler did a output of the counter and terminated the whole task.

In order to also be able to measure single functions as is the case in the `GetResource`/`ReleaseResource` scenario the following piece of assembly code was used:

```
1 xor %eax, %eax
2 cpuid

4 rdtsc

6 mov %rax, %0
7 shl $32, %rdx
```

```
8  or %rdx, %0

10 xor %eax, %eax
11 cpuid
```

This function, which was always inlined by use of `#define` statements and `__volatile__ __asm__` inline assembly, does the following:

- Clear the CPU instruction pipeline by a call of `cpuid`
- Read the time stamp counter the CPU provides, store it in the given variable
- Clear the instruction pipeline again

The instruction pipeline has to be specifically cleared as the position of the `rdtsc` opcode may be arbitrarily rearranged witin the pipeline. It therefore leads to inconsistent and strange result if not done. The whole function – when executed twice to determine the evaluated function's runtime – took a total of 109 CPU cycles on the reference machine. These 109 cycles were always subtracted.

6.2 Scheduling Overhead

In order to schedule between tasks, their specific contexts have to be saved and restored. We will now compare the size of these contexts in detail across different architectures.

Types	Machine Words			Bytes		
	x86-32	x86-64	AVR	x86-32	x86-64	AVR
General Purpose Registers	6	6	27	24	48	27
Extended Registers	0	8	0	8	64	0
Pointer Registers	2	1	0	8	8	0
Processor Flags	1	1	1	4	8	1
Return Address	1	1	2	4	8	2
Sum	10	17	30	40	136	30

Table 6.1: Scheduling Overhead in Terms of RAM Size

Note that the only architecture where the pointer size is unequal the machine word size is the AVR, where a pointer (like the return address) is two bytes compared to a one byte machine word.

It becomes clear the AVR is not the ideal architecture to do multitasking on, especially since it has huge overhead (30 bytes) compared to the usual SRAM size

(usually ranging from 128 bytes up to 1024 bytes).[Atm07]

6.3 Measurements

One of the first things of interest is how long it takes to schedule and terminate tasks. In this test it has been measured how long it takes for a task to chain itself repeatedly:

	x86-32		x86-64	
	Time	*n*/sec	Time	*n*/sec
`ChainTask(self)`	104 ns	9.60 M	48.4 ns	20.7 M
`ActivateTask(self)` `TerminateTask()`	161 ns	6.22 M	67.9 ns	14.7 M
`GetResource()` `ReleaseResource()`	56.8 ns	17.6 M	29.3 ns	34.1 M

`WaitEvent()`
`ClearEvent()`
`SetEvent()`
	360 ns	2.78 M	228 ns	4.39 M

`ActivateTask(HP)` `ChainTask(self)`	330 ns	3.03 M	175 ns	5.71 M

`ActivateTask(MP)`
`ActivateTask(HP)`
`ChainTask(self)`
	553 ns	1.81 M	323 ns	3.10 M

`ActivateTask(MP1)`
`ActivateTask(MP2)`
`ActivateTask(HP)`
`ChainTask(self)`
	755 ns	1.32 M	502 ns	1.99 M

`Schedule()`	48.1 ns	20.8 M	17.4 ns	57.5 M

Table 6.2: *josek* x86-32 against x86-64

These values can be normalized according to the explanation in 6.1:

	Norm. x86-32		x86-64		Ratio
	Time	*n*/sec	Time	*n*/sec	
`ChainTask(self)`	43.9 ns	22.8 M	48.4 ns	20.7 M	0.91
`ActivateTask(self)` `TerminateTask()`	67.7 ns	14.8 M	67.9 ns	14.7 M	1.00
`GetResource()` `ReleaseResource()`	23.9 ns	41.8 M	29.3 ns	34.1 M	0.82

`WaitEvent()`
`ClearEvent()`
`SetEvent()`
	152 ns	6.59 M	228 ns	4.39 M	0.67

`ActivateTask(HP)` `ChainTask(self)`	139 ns	7.20 M	175 ns	5.71 M	0.79

`ActivateTask(MP)`
`ActivateTask(HP)`
`ChainTask(self)`
	233 ns	4.29 M	323 ns	3.10 M	0.72

`ActivateTask(MP1)`
`ActivateTask(MP2)`
`ActivateTask(HP)`
`ChainTask(self)`
	318 ns	3.14 M	502 ns	1.99 M	0.63

`Schedule()`	20.3 ns	49.3 M	17.4 ns	57.5 M	1.16

Table 6.3: *josek* x86-32 (normalized) against x86-64

Comparing these valued yields an interesting result: *josek* performs better (at least relatively) on the x86-32 architecture. The difference is not to be dismissed as beside the point; it's up to 61% faster[1] when heavy scheduling occurs. This is not really surprising, as the context change is much more expensive on the x86-64 as it is on the x86-32 – the reasoning behind this is explained in section 6.2.

On the x86-64 platform these tests were also performed against *Trampoline*:

	Trampoline		*josek*		Ratio
	Time	*n*/sec	Time	*n*/sec	
`ChainTask(self)`	911 ns	1.10 M	48.4 ns	20.7 M	18.8
`ActivateTask(self)` `TerminateTask()`	1.15 μs	870 k	67.9 ns	14.7 M	16.9
`GetResource()` `ReleaseResource()`	492 ns	2.03 M	29.3 ns	34.1 M	16.8

`WaitEvent()`
`ClearEvent()`
`SetEvent()`
	2.53 μs	395 k	228 ns	4.39 M	11.1

`ActivateTask(HP)` `ChainTask(self)`	2.05 μs	488 k	175 ns	5.71 M	11.7

`ActivateTask(MP)`
`ActivateTask(HP)`
`ChainTask(self)`
	3.17 μs	315 k	323 ns	3.10 M	9.8

[1] $\frac{100 \cdot 3.20}{1.99} - 100 \approx 61\%$

`ActivateTask(MP1)`
`ActivateTask(MP2)`
`ActivateTask(HP)`
`ChainTask(self)`
	4.36 μs	229 k	502 ns	1.99 M	8.7

`Schedule()`	247 ns	4.05 M	17.4 ns	57.5 M	14.2

Table 6.4: *Trampoline* compared to *josek* on x86-64

When comparing *Trampoline* to a equal *josek* system, the *Trampoline* OSEK always seems to be at least one magnitude behind in terms of CPU time. This is primarily because of the *viper* hardware abstraction layer which *Trampoline* provides. The additional overhead added by usage of the `libpcl` (Portable Coroutine Library) is relatively small.

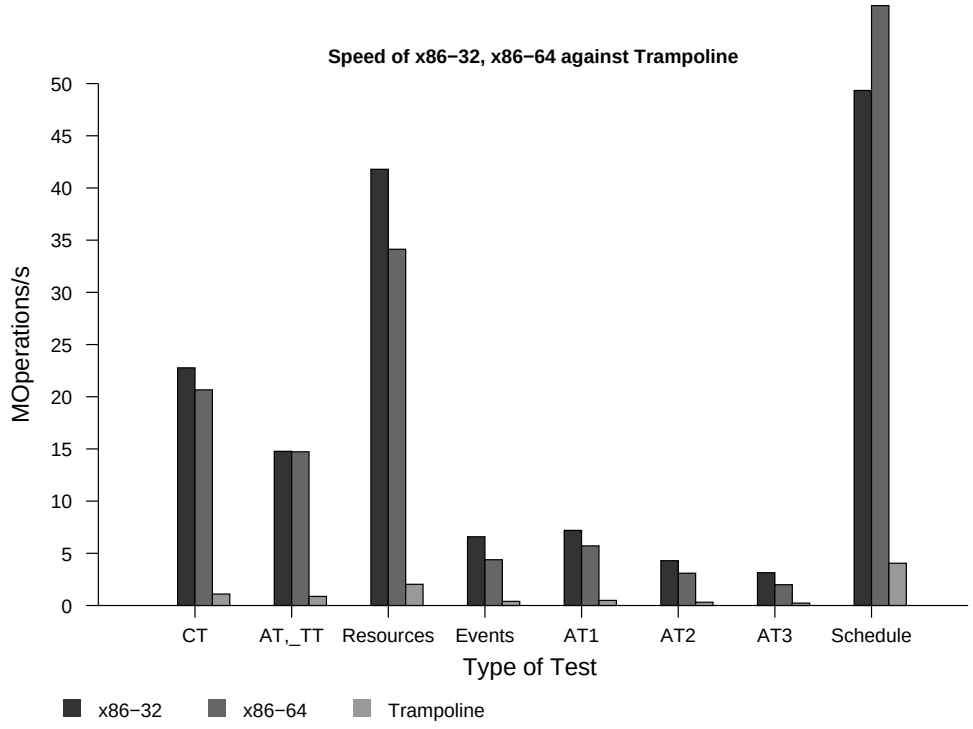

Figure 6.1: Comparison of Architectures

The particular reason the example using `GetResource` and `ReleaseResource` is so fast compared to the other examples is that it doesn't need any context switches. As there is always only one task active which repeatedly acquires and releases the

resource, a context change to another task simply never becomes necessary. Although the scheduler is called in every instance of `ReleaseResource`, it decides every single time that nothing has to be changed and the program continues right where it left of. How costly in terms of CPU-times context switches are illustrates figure 6.2. A test system performing solely the `ChainTask` example was executed many times, each time with a bigger context. It was artificially inflated by inserting `push` and `pop` operations at the appropriate places in `switchtask` and `dispatch`. The figure shows the time of each `ChainTask` cycle over the increase in context size (measured in machine words, i.e. 8 bytes each):

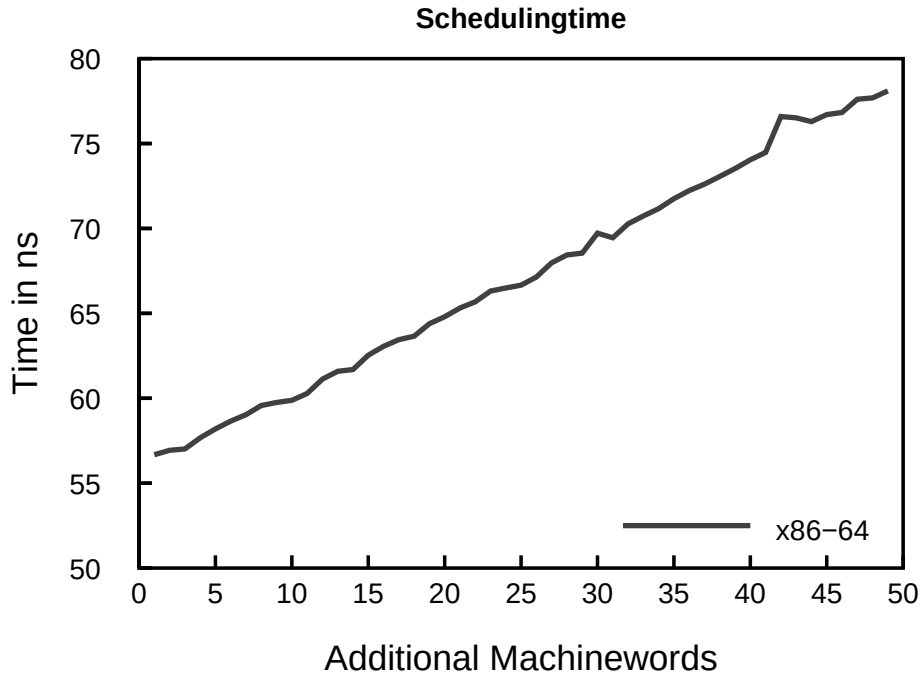

Figure 6.2: Artificially Inflating Context Size

The figure clearly shows that the runtime (which is almost equal to the context switch time in this example) grows linearly with the size of the context.

A particularity of the code using `GetResource`/`ReleaseResource` becomes not evident through the above numbers, however. It's the difference in runtime of the two resource functions. For this purpose, both have been sperately analyzed using the method described above.

	`GetResource`	`ReleaseResource`

Number of executions:	369312353	
Total execution time:	77.8 sec	
Operations/Second:	4.75 M	
Number of CPU cycles:	9928252581	22505181787
Cycles/Execution:	27	61
Theoretical Ops/Second:	15.5 M	6.84 M

Table 6.5: `GetResource` compared to `ReleaseResource`

It is not surprising that the `ReleaseResource` system call takes more than twice the time `GetResource` needs. The reason for that is simple: `GetResource` will only increase the task priority or not alter it at all. The currently running task will therefore never be preempted by a `GetResource` system call – no rescheduling point is necessary. When calling `ReleaseResource`, however, it is very well possible the task which is currently in the *running*-state decreases in priority and is therefore preemted by a more important task. `ReleaseResource` thus calls `Schedule`, which takes time to come to a scheduling decision.

Chapter 7

Conclusion

TODO-assessment Writing an OSEK conform The goal of writing a OSEK conform implementation was for one thing to show how it can be done and for another to point out optimization possibilities. Through usage of *josek* a very clean OSEK conform operating system can be generated within the matter of minutes. Attention was paid that the resulting code is cleanly readable – the trade-off between highly optimized code and good code readability was taken in favor of code concinnity. Testing functional components of an OSEK compatible operating system is easy, extending the code generator via the use of code hooks or additional files is not much more difficult. Thus, the *josek* project has to be considered a success. It shows clearly what can be done with considerable effort and has the potential of being used as a starting point for other developers.

Bibliography

[Atm07] Atmel Corporation. *8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash – ATmega128*. ATMEL, August 2007. http://www.atmel.com/atmel/acrobat/doc2467.pdf.

[BBFT06] Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, and Yvon Trinquet. *Trampoline – An OpenSource Implementation of the OSEK/VDX RTOS Specification*. ETFA Proceedings pg. 62-69, 2006. http://trampoline.rts-software.org/IMG/pdf/trampoline.pdf.

[Int07a] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 2A: Instruction Set Reference, A-M*. May 2007. http://www.intel.com/design/processor/manuals/253666.pdf.

[Int07b] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 2B: Instruction Set Reference, N-Z*. May 2007. http://www.intel.com/design/processor/manuals/253667.pdf.

[ope07] openOSEK.org. *The most common questions answered*. March 2007. http://www.openosek.org/tikiwiki/tiki-index.php.

[OSE04] The OSEK Group. *OSEK/VDK Binding Specification 1.4.2*. July 2004. http://portal.osek-vdx.org/files/pdf/specs/binding142.pdf.

[OSE05] The OSEK Group. *OSEK/VDK Operating System Specification 2.2.3*. February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[Sch07] Michel Schinz. *Advanced Compiler Construction - Tail Call Elimination*. March 2007. http://lamp.epfl.ch/teaching/

archive/advanced_compiler/2007/resources/slides/act-2007-07-tail-calls_6.pdf.

[StGDC07] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection – Function Attributes*. GNU Press, July 2007. http://gcc.gnu.org/onlinedocs/gcc-4.2.1/gcc/Function-Attributes.html.