Daniel Gruß

Multi-platform Operating System Kernels

Master's Thesis

Graz University of Technology

Institute for Applied Information Processing and Communications
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch

Advisor: Ass.Prof. Dipl.-Ing. Dr.techn. Peter Lipp

Graz, May 2014

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

In this thesis we discuss the multi-platform compatibility of operating system kernels. For this purpose we ported the SWEB operating system kernel from the x86 architecture to the x86-64 architecture and to the ARM-v5 architecture. The reader will find a detailed comparison of the x86 and x86-64 architecture from an operating system development point of view. Afterwards we describe how we ported SWEB from x86 to x86-64 in detail. We compare the ARM-v5 architecture to the x86 architecture and describe how we ported SWEB from x86 to ARM-v5. We ported SWEB to three different ARM boards: the ARM Integrator C/P, the Gumstix Verdex and the Raspberry Pi. To make SWEB a multi-platform compatible operating system, we analyze the Linux and the Minix kernel and identify relevant design principles. Finally we will describe how we applied these design principles to the SWEB kernel while merging four different architecture branches.

Keywords: operating systems, multi-platform compatibility, Intel, x86, x86-64, IA-32, IA-32e, ARM, ARM-v5, ARM-v7, Xen, ARM Integrator C/P, Gumstix Verdex, Raspberry Pi

Kurzfassung

Diese Arbeit behandelt die Multi-Plattform-Kompatibilität von Betriebssystemkerneln. Zu diesem Zweck beschreiben wir, wie wir den SWEB-Betriebssystemkernel von der x86-Architektur auf die x86-64-Architektur und die ARM-v5-Architektur portiert haben. Dem Leser wird dazu ein detaillierter Vergleich der x86-Architektur und der x86-64-Architektur, aus der Sicht der Betriebssystementwicklung, geboten. Daran anschließend beschreiben wir im Detail wie SWEB portiert wurde. Wir vergleichen die x86-Architektur außerdem mit der ARM-v5-Architektur und erklären wie wir SWEB auf ARM-v5 portiert haben. Wir haben SWEB auf 3 verschiedene ARM Platinen portiert: das ARM Integrator C/P, das Gumstix Verdex und das Raspberry Pi. Um SWEB Multiplattform-kompatibel zu machen, analysieren wir den Linux-Kernel und den Minix-Kernel und identifizieren wichtige Entwurfsprinzipien. Abschließend beschreiben wir wie wir diese Entwurfsprinzipien auf den SWEB-Kernel während der Zusammenführung der verschiedenen Architekturentwicklungszweige angewandt haben.

Stichwörter: Betriebssysteme, Multi-Plattform-Kompatibilität, Intel, x86, x86-64, IA-32, IA-32e, ARM, ARM-v5, ARM-v7, Xen, ARM Integrator C/P, Gumstix Verdex, Raspberry Pi

Acknowledgements

I owe several contributors to this thesis my thanks.

First, to my advisor Peter Lipp, for his patience and guidance, and for his reviews of draft versions of this thesis. Thank you very much for your support over the past years.

Second, to my parents Britta and Dieter. Thank you for supporting me and my passion for computer science since my childhood.

Third, to my other half, Maria, the reason why I came to Graz. Thank you for your support during the last year and for reviewing draft versions of my thesis. I also want to thank Maria's parents, Eva and Helmut, for their support.

Finally, I want to thank my friends and fellow students, Manuel, Markus, Herwig, and Alen, for discussing many ideas with me, and Matthias, Florian, and Jakob. Thank you all for reading and reviewing parts of my work.

Daniel Gruß

Contents

List of Figures

Chapter 1

Introduction

In this master thesis we describe characteristics of multi-platform operating systems and how they can be used to make the SWEB kernel multi-platform compatible. Andrew S. Tanenbaum states that operating systems are "huge, complex and long-lived" [Tan09]. An easy way to avoid writing such code several times is to write it once and run it on various platforms. Of course, this influences the design of the kernel.

Linux started as an operating system for the Intel 386. Today it supports far more architectures than any other operating system. In this thesis we explain why the Linux kernel runs on so many platforms.

The Minix kernel originally was an operating system for the Intel 8086. The current version of Minix supports the Intel 386 architecture and the ARM-v7 architecture. Both are 32-bit platforms. We will figure out how the Minix kernel is abstracted to support different platforms.

The family of all architectures based on the Intel 8086 is called the x86 architecture family. The Intel 386 architecture (also called i386 or IA-32) is the first 32-bit extension in the x86 architecture family. It allows switching from the 16-bit mode called "real mode" to 32-bit mode called "protected mode". The name "protected mode" comes from the memory protection it provides. In 2003 AMD introduced AMD64 as the first 64-bit extension to the x86 architecture family. Intel decided to provide this extension in their CPUs as well and released a very similar instruction set called Intel 64 (also known as IA-32e or EM64T). As there are only very few differences between AMD64 and Intel 64 both architectures are commonly referred to as x86-64 or x64. In this thesis we will use the term x86 for the x86 architecture family, and the terms x86-32 and x86-64 when talking specifically about 32-bit x86 architectures respectively 64-bit x86 architectures.

The SWEB kernel is used at Graz University of Technology. Similar to the Minix kernel it runs primarily on the x86 architecture. Some years ago SWEB had support for the 32-bit Xen paravirtualization as well, but as this code has not been maintained it is currently not working. Recently the SWEB kernel has been ported to the ARM-v7 architecture. As a part of this master thesis the SWEB kernel has been ported to the x86-64 architecture and the ARM-v5 architecture. In this master thesis we will describe the differences between the x86 and the x86-64 architectures as well as the x86 and the ARM-v5 architecture. Furthermore, we will describe what changes in the SWEB kernel were necessary to make SWEB a multi-platform compatible operating system by merging the different architecture implementations together.

This master thesis is also intended to give the reader a good overview in porting operating systems to a new platform, especially the architectures we discuss in this work. Therefore, we explained in detail how the operating system was ported to new architectures. We mention each step necessary to avoid typical pitfall scenarios and how to solve them.

1.1 Motivation

SWEB is an operating system for educational purposes used at Graz University of Technology. Its initial development started in 2005. SWEB initially supported the x86 architecture only. Support for the Xen paravirtualization has been added in 2005 by Andreas Weinberger [Wei05] and has been maintained until 2006 [Wei06]. Today the code in the x86 branch and the Xen branch diverged and the Xen branch is not working anymore.

In 2013 Alen Harbas started working on an ARM-v7 (Cortex-M4) port of SWEB called aOS [Har13]. This port runs on a STM32f4Discovery board. It is the third architecture SWEB runs on. Right now we are working on emulation of this port using `qemu-system-arm`. The ARM Cortex-M4 has no memory management unit and thus aOS does not provide support for a memory management unit. Based on the experience gained from this port we wrote a new ARM-v5 port for the ARM Integrator/CP board. This port utilizes the ARM memory management unit and some of the hardware the board provides.

As a part of this master thesis we implemented PAE on x86 and ported SWEB to x86-64. x86-64 has 64-bit addresses and, therefore, brings new aspects to the hardware abstraction in the operating system kernel. With SWEB virtually running on four different architectures (x86, x86-64, ARM-v5/v7, Xen), maintaining all architecture branches separately is significantly more effort than maintaining only one branch. Therefore, we want to group the code all kernels share or could share, so that all architecture branches differ only in an architecture dependent part of the code. Furthermore, we want to minimize the code footprint of the architecture dependent code while preserving its readability.

In order to accomplish this task we will take a look at other operating systems. The Linux kernel, a monolithic kernel, runs on almost any architecture that has a memory management unit, far more platforms than any other operating system. This is interesting, as one would expect that a micro-kernel operating system should be easier to port than a monolithic kernel and thus support more architectures. In addition, Linus Torvalds stated that the Linux kernel “isn’t written to be portable” [Tor99]. He rather describes his approach as “trying to define a sane common architecture”. We will examine how this approach reflects in the Linux source code, and find out whether we can apply similar design strategies to SWEB making it more platform independent.

The Minix kernel is a micro-kernel for educational purposes used at the VU University Amsterdam. While Minix 1 and 2 supported a few different architectures, Minix 3 does not yet. The Minix 3 kernel is a complete redesign in order to make Minix 3 not only usable for educational purposes but as a serious operating system on embedded platforms as well [HBG+06]. It currently supports the x86 architecture and the ARM-v7 architecture. The ARM-v7 support is still under development. We will find out how Minix implements means of hardware abstraction and see if we can apply them to the SWEB kernel.

1.2 Structure of this document

Chapter 2 analyzes and compares the multi-platform compatibility of the Linux kernel, the Minix kernel and the SWEB kernel. Chapter 3 gives a comparison of the x86-32 and the x86-64 with a focus on differences which matter when porting an operating system to 64 bits: virtual memory, interrupt and descriptor tables and the binary format. Chapter 4 describes porting SWEB to x86-64 in detail. We will then compare x86 and ARM-v5 in Chapter 5, again from an operating system development point of view. In Chapter 6 we describe how we ported SWEB to ARM-v5 including the three different ARM boards. Chapter 7 explains improvements on the SWEB kernel to enhance its multi-platform compatibility. Chapter 8 provides a conclusion and summarizes this work.

Chapter 2

Multi-platform compatibility of operating system kernels

In this chapter we examine the micro-kernel based operating system Minix 3.0 and the monolithic Linux kernel in order to find out how they provide fundamentals for multi-platform compatibility. On the one hand we aim for a small code footprint of the architecture dependent code, on the other hand we want little redundant source code. However, `#ifdef` constructs hinder the maintainability of the source code as architectural changes spread over the whole project instead of a small part only. More over source code readability is very important for an educational operating system and `#ifdef` constructs worsen readability.

We analyze how the kernel design splits architecture dependent and architecture independent code, how these parts interact and how this interface changes the way the rest of the kernel is written. We analyze how data type usage influences multi-platform compatibility in the operating system kernels. We will take a look at the boot procedures on different architectures and how much redundant source code exists in the project.

From the design of the Minix and the Linux kernel we extract design characteristics responsible for multi-platform compatibility. In the last section of this chapter we examine which of the design characteristics we found are implemented in SWEB already and which are not.

2.1 The Minix kernel

The Minix kernel is a micro-kernel and as one would expect it has a low code footprint. The code is divided into a main part residing in the root folder of the kernel, the `arch` folder, which contains the architecture dependent part, and the `system` folder, which contains implementations for the system calls. The architecture dependent code makes about 60% of the kernel code, which splits up in 60% x86 code and 40% ARM-v7 code. The folder structure of the Minix kernel is shown in Figure 2.1.

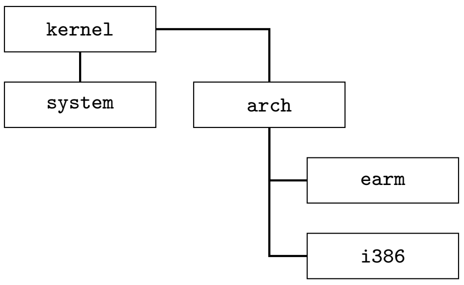


Figure 2.1: Folder structure of the Minix kernel [Min14]

The `proto.h` header contains various function prototypes, some of them implemented in the `arch` folder. That is, any architecture has to implement these functions. So the design characteristic we found here is the separation of architecture dependent code into an `arch` folder. All architectures have to provide functions which are prototyped in a common interface. Then this common interface provides a header which is included by the architecture independent source files.

In some details Minix deviates from this design. For instance, it contains a few `#ifdef` constructs depending on the architecture defines `__arm__` and `__i386__`. They are small enough not to degrade the readability in case of the implementation of two architectures, but it might when supporting a lot more different architectures, because the programmer has to see through the architecture independent code for architecture dependent `#ifdef` constructs.

The Minix kernel schedules upon timer interrupts using the `switch_to_user` function. If no runnable process is found the system idles.

Throughout the kernel some C library functions are used. They are implemented in a `common` folder the kernel shares with the user space. There are different implementations for different architectures, but we do not count that as kernel code, as it is abstracted code which provides the C library interface and the kernel design has no influence on it.

Data type usage can constrain multi-platform compatibility if bad assumptions are made. For instance assuming that a pointer has 32 bits is a bad assumption, if you want to support platforms with different pointer sizes. There are only a few lines of code in Minix where such assumptions are made.

We will take a look at a line from the implementation of the exec syscall in the file `do_exec.c`:

```
arch_proc_init(rp, (u32_t) m_ptr->PR_IP_PTR, (u32_t) m_ptr->
    PR_STACK_PTR, name);
```

The `arch_proc_init` function is implemented in the `arch` folder. `rp` is the address of the calling process. `name` is the name of the program to execute. Both, the new instruction pointer `m_ptr->PR_IP_PTR` and the new stack pointer `m_ptr->PR_STACK_PTR` are casted to `u32_t` in order to match the signature of the `arch_proc_init` function:

```
void arch_proc_init(struct proc *pr, u32_t, u32_t, char *);
```

This might bring problems when porting Minix to a system with a different pointer size. An interface change will be necessary and then all architecture implementations have to be adapted to the new interface.

The boot procedure for ARM-v7 and x86 starts in both cases in the `head.s` file, which of course differ, as they are written in different assembly languages. In both cases the `pre_init` function is called. It parses the multiboot headers and sets up paging. Minix uses an identity mapping while booting. It is set up in the `pg_identity` and maps the upper half of the address space to the physical address space starting from zero. The two files have about 50 lines in common, for instance the multiboot header parsing. Afterwards the `kmain` function in the architecture independent code is called.

The function `kmain` contains the rest of the boot procedure, using functions from the `proto.h` header, implemented in the architecture folders. It initializes the page directory, the interrupt table and the syscall handlers. The assumption of having a page directory leads to the fact that Minix won't work on a platform without a paging mechanism without a number of modifications in the architecture independent code.

Overall, the architecture folders `i386` and `earm` have about 1200 common lines of code, that is about 18% of the code in the `i386` folder respectively about 35% of the code in the `earm` folder.

2.2 The Linux kernel

Although the Linux kernel started as a Minix compatible kernel it design differs significantly. The Linux kernel is a monolithic kernel. Today it comprises more than 12 million lines of code. The code is divided into several directories. An overview of the Linux kernel folder structure is shown in Figure 2.2.

The `arch` folder contains architecture dependent code in terms of the CPU architecture. Following the same design idea, driver code is placed in a `drivers` directory, implemented independently from the CPU architecture. This way it is possible to implement the driver only once and use it on all different architectures. The CPU architecture dependent code makes about 17% of the kernel code, that is about 2 million lines of code. These split over 29 CPU architecture folders. The x86 folder does not only contain the source code for the Intel 386 but for the whole x86 family including the x86-64. Even the xen-x86 support is implemented in that folder, because it runs on the same CPU architecture.

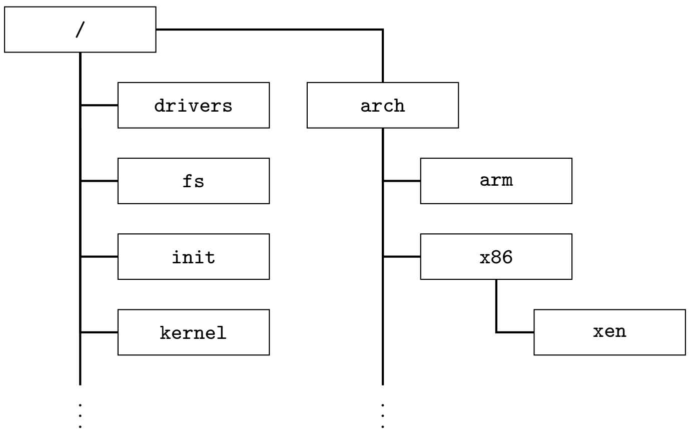


Figure 2.2: Excerpt of the folder structure of the Linux kernel

If you compare the architecture folders you see that they have almost no source code in common with any other architecture. This is great in terms of source code maintenance, as with less duplicate code it is more unlikely to have to change code in several places simultaneously. Inside the architecture folders there are sub folders for different platforms based on this CPU architecture. Only device drivers for devices which are specific to this single CPU can be found in the architecture folders or the platform sub folder. The sub folder structure can be seen as a tree where a leaf is a specific platform and it includes all code in its own branch. This is not true for all leaf folders but it gives a good image of how code replication between platforms derived from the same architecture is avoided.

Linus Torvalds stated that it is a basic design rule of the Linux kernel to avoid adding new interfaces [Tor99]. This way the architecture independent code won't be extended by calls against the interface of a single architecture, which would cause problems for the other architectures. System drivers are not part of the architecture dependent code and thus can be added with less problems. Code that does not comply to this interface will not be added to the Linux kernel. Furthermore, the Linux kernel has a kernel module system to load additional kernel modules at runtime. This allows extending the kernel without changing the kernel source code itself. Kernel modules have to implement the interface defined by the kernel.

Many `#ifdef` constructs can be found in the Linux kernel, depending on architecture defines like `CONFIG_X86_PAE`. This is a fast solution, which keeps the duplicate code between different architectures minimal, but it also might weaken the readability, in particular for students new to operating system kernels. The used data types are not bit size explicit, except for where the interface requires it. This is important as it enables to use the same code on architectures with different pointer sizes.

Linux has a few different schedulers, which are implemented in the `kernel/sched` folder. The boot procedure for x86 starts in the `header.S`, which contains the initialization until calling the first `main()` function from the `boot/main.c` file. The `main()` function initializes the basic hardware, virtual memory and finally uses the function `go_to_protected_mode()` to turn on paging and jump to the protected mode entry point read from the ELF header of the kernel binary.

Many files exist in different versions for different bit sizes or CPU instruction sets. In case of the x86 architecture there are `_32` files for the x86-32 (and PAE) implementation and `_64` files for the x86-64 implementation. PAE code is separated from the x86-32 code using `#ifdef` sections. The sub folder solution Linux uses in similar situations keeps a better overview.

Just like Minix, Linux makes the basic assumption that an architecture has something like the x86 page directory. However, there are forks of the Linux kernel which work on architectures without a paging mechanism. They are maintained separately.

2.3 The SWEB kernel

The SWEB kernel is divided into two parts, the architecture dependent `arch` folder and the architecture independent `common` folder. We already saw this design characteristic in both the Minix kernel and the Linux kernel. The SWEB kernel is smaller than the Linux kernel by several orders of magnitude, measured in lines of code. But compared to the Minix kernel it has significantly more lines of code. The folder structure of the SWEB kernel is shown in Figure 2.3.

The `common` folder contains about 90% of the total source code and is written in C++. The SWEB kernel uses the `ustl` standard library. 40% of the `common` source code belong to the `ustl` library. The `ustl` library uses C++ type abstractions like `size_t` to provide architecture independent code.

Several locking mechanisms are implemented in the kernel and provide a convenient object oriented interface. The `Syscall` class provides an abstraction for syscall handling. Independent from how the syscall is implemented and whether it is implemented in the architecture independent or in the architecture dependent code, it is passed through to this method, which provides a common interface.

SWEB has an ELF32 binary loader inside the kernel. The idea of placing it in the architecture independent code, is that loading ELF32 binaries should not depend on which CPU architecture the kernel runs on. When the binary loader or a syscall require loading data from the disk, SWEB uses the virtual file system layer which abstracts

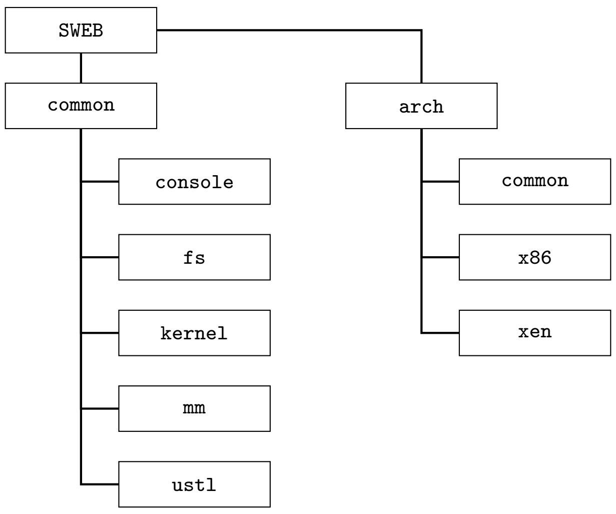


Figure 2.3: Folder structure overview of the SWEB kernel

the different file systems implemented. Both, the virtual file system layer and the file system implementations are part of the architecture independent code, as the file system format does not depend on the actual hardware. The file system implementations use the interfaces of the architecture dependent hard disk drivers.

The SWEB `Scheduler` is currently called on every interrupt 0. The interrupt handler is implemented in the architecture dependent code, whereas the `Scheduler` provides only abstracted methods to manage and schedule threads.

While the `ustl` uses abstracted data types, SWEB in general does not. This is much like Minix. In SWEB development, it has been considered a good practice to always use data types containing the number of bits of the variable, like `uint32` or `uint8`, since it makes the programmer aware of how many bits a variable has. While this may be tolerable on a 32-bit only operating system it produces a number of problems when porting to 64-bit operating systems. In particular, one should never cast pointers to a fixed size integer type like `uint32`, as this makes it impossible to use the code without changes on a platform with a different pointer size.

The `arch` source code contains a common interface for all architectures. Each implementation of an architecture is placed inside a different sub folder. At this point we will only discuss the implementation of the x86 architecture, as it is the most comprehensive. Moreover, the xen architecture implementation is not working at the moment and has been repaired partially while writing this thesis. The architecture dependent code contains about 10% of the source code. It is written in C and in x86-asm partially. The x86-asm code comprises the boot code, the code for interrupt handling and context switching. In each case according C++ methods are called.

The `InterruptUtils` class contains methods for interrupt handling. The `ArchMemory` class provides an interface for the architecture independent code to manage virtual memory, for instance to initialize the page directory or to map or unmap pages. Currently it contains only static methods. The `ArchThreads` class provides an interface for an abstracted handling of a threads CPU registers.

Furthermore, the x86 architecture folder contains the driver implementations for an ATA driver, an IDE driver, a serial port driver, a driver for the interrupt controller and more. But there are also classes which are not really architecture dependent, for instance the `ArchCommon` class. So, SWEB basically provides architecture abstraction in its design, but it is only implemented rudimentarily. In Chapter 7 we present changes in the SWEB kernel to enhance its architecture independency.

Chapter 3

Comparison of x86 and x86-64

In this chapter we describe the x86 and x86-64 architectures from an operating system developer's point of view. The x86-64 architecture implements the whole x86 instruction set and extends it by some new instructions. In 64-bit mode, however, there are a lot of differences as most registers and instruction operands have been changed in their size. x86-64 introduces the new long mode paging, which we will describe in detail. We will take a closer look at the setup of the global descriptor table and the interrupt descriptor table. Furthermore, we will explain what changed in the calling convention, in interrupt handling and in the binary format.

3.1 32-bit paging

In 32-bit paging mode the memory management unit does address translation using one or two levels of lookup tables. One level paging means having 4 MiB pages, whereas two level paging means having 4 KiB pages.

The first lookup table is the page directory. Its 20-bit physical page number is stored in upper 20 bits of the CR3 register. The lower 12 bits are used for changing the memory management units' behavior. The page directory contains 1024 entries of 4 bytes each. That is, the page directory fits on a single 4 KiB page. The page directory maps the first 10 bits of the virtual address either to a 4 MiB physical page or to a page table, depending on the size bit in the page directory entry. These first 10 bits are called the page directory index (PDI).

The page table is the second lookup table. It contains 1024 entries of 4 bytes each, just as the page directory. And it has to be page aligned as well, as only a 20-bit physical page number is stored in the page directory entry. The second 10 bits of the virtual address are used as the page table index (PTI).

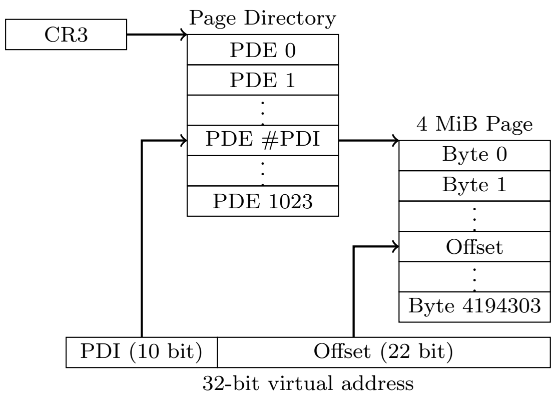


Figure 3.1: Virtual address resolution in 32-bit 4 MiB paging mode according to page 4-12 of the Intel manual [Int12]

When addressing a 4 MiB page the CPU takes the page directory physical page number from the CR3 register and the page directory index from the virtual address to find the page directory entry to this virtual address. The page directory entry contains the 10-bit physical page number of the 4 MiB page. Combined with the lower 22 bits of the virtual address this forms the physical address. The virtual address resolution for 4 MiB pages is also shown in Figure 3.1.

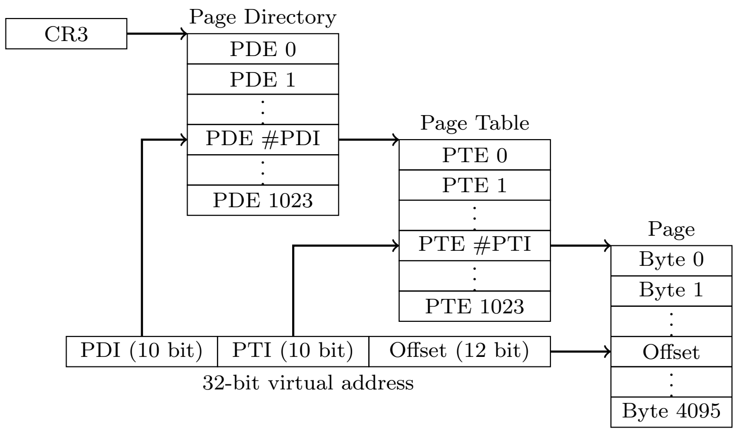


Figure 3.2: Virtual address resolution in 32-bit 4 KiB paging mode according to page 4-12 of the Intel manual [Int12]

In the case of 4 KiB pages the page directory entry is found the same way, but the page directory now contains a 20-bit physical page number of a page table. The physical page number can be extended with zeros to get the physical address, because the page table is page aligned. The page table index leads to the page table entry. The page table entry maps a 4 KiB virtual page to a 4 KiB physical page. Therefore, the page table entry contains a 20-bit physical page number of a 4 KiB page. Combined with the lower 12 bits of the virtual address this forms the physical address. The virtual address resolution for 4 KiB pages is also shown in Figure 3.2.

The page directory entries and page table entries contain information on the memory area they map. The present bit defines whether there is any mapping in the memory area of this entry. The size bit only exists in the page directory and defines whether this entry maps directly to a 4 MiB page or to a page table. The user space bit and the writeable bit specify access rights on the page.

3.2 PAE paging

With physical address extension enabled the behavior of the memory management unit differs from 32-bit paging mode in some details.

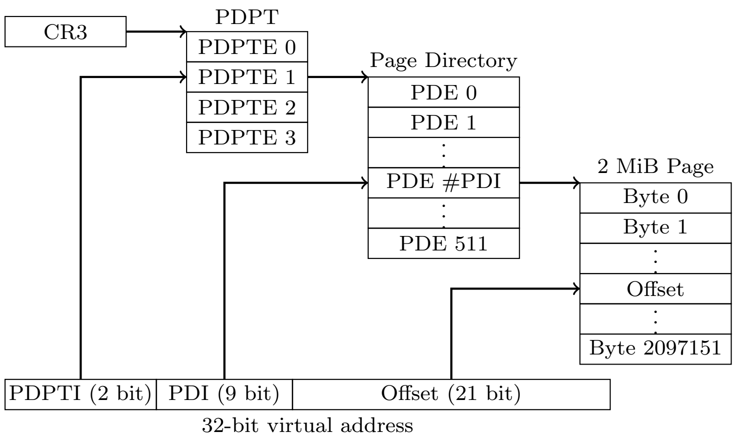


Figure 3.3: Virtual address resolution in PAE 2 MiB paging mode with PDPTI=1 according to page 4-21 of the Intel manual [Int12]

To extend the usable physical address space, the physical page number fields in page tables and page directories are extended by 4 bits. CPUs with physical address extension allow at least 36-bit physical addresses. Depending on the CPU longer physical addresses may be possible. The actual maximum value can be retrieved from the CPU. It is at most 52 bits. We will further on only describe the 36-bit case, as it does only differ from other cases in the lengths of some fields in the paging structs. The new layer in PAE paging mode is the page directory pointer table.

The physical page number field of the page table entry had only 20 bits in 32-bit paging mode, because the 12-bit offset from the virtual address was used, resulting in a 32-bit physical address space. In PAE paging mode virtual addresses still have 32 bits, but the physical address space has at least 36 bits. Therefore, the page table entry has to contain

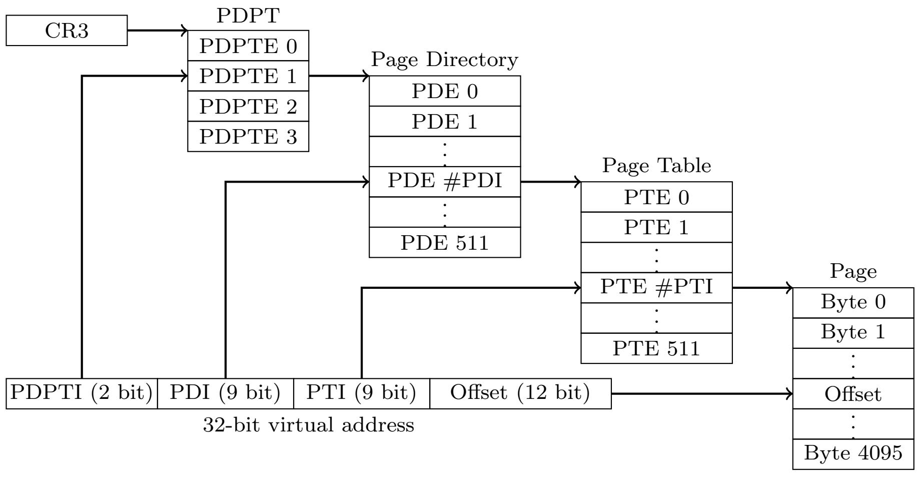


Figure 3.4: Virtual address resolution in PAE 4 KiB paging mode with PDPTI=1 according to page 4-20 of the Intel manual [Int12]

a 24-bit physical page number. For this reason the page table entries are extended to 64 bits using 27 padding bits after the physical page number field. A page table should still fit on a single page, thus the number of page table entries per page table are reduced to 512. Thus one page table controls only 512 pages, that is 2 MiB. To address any page table entry you need 9 bits. Hence the page table index part in the virtual address is only 9 bits long. Having a 24-bit physical page number and 12 bits offset from the virtual address we can construct any 36-bit physical address.

The same changes apply to the page directory. The page table page number field is extended from 20 bits to 24 bits. In the case of large page mappings, a large page is only 2 MiB long in PAE paging mode. This is because the page table index and the offset part of the virtual address together comprise only 21 bits and 2^{21} bytes are 2 MiB.

In 32-bit paging mode the physical page number field in the page directory had 10 bits. We now need to add 4 bits for the physical address extension and 1 bit as each large page is half as long as in 32-bit paging mode and therefore, there are twice as many large pages. Hence, the physical page number field is 15 bits long. Together with the 21 bits from the virtual address as offset we can again construct any 36-bit physical address. The virtual address resolution for 2 MiB pages in PAE paging mode is also shown in Figure 3.3. As in the page tables, padding bits are added after the page number field.

As each page directory entry doubled in size, there are only 512 instead of 1024 page directory entries. Each entry controls a region of 2 MiB size. Therefore, one page directory controls 1 GiB of virtual address space.

Now we have 12 bits offset, 9 bits page table index (respectively 21 bits offset, in the case of 2 MiB pages), 9 bits page directory index and 2 bits remain for a new layer in address translation.

The page directory pointer table is added as another layer to the virtual address translation. It has four entries, each 64 bits long. These four entries divide the virtual address space into four 1 GiB regions. Each entry holds the physical address of the page directory and additional information, like whether the entry is valid (present bit).

Figure 3.4 shows virtual address resolution in 4 KiB PAE paging mode.

3.3 x86-64 paging

x86-64 paging mode is an extension to PAE paging. PAE paging mode allowed to translate 32-bit virtual addresses to physical addresses with 36 to 52 bits length, depending on the hardware used. In x86-64 paging mode we can translate 48-bit virtual addresses to physical addresses with 40 to 52 bits length. The actual maximum value can be retrieved from the CPU. We will further on only describe the 40-bit case, as it does only differ from other cases in the lengths of some fields in the paging structs.

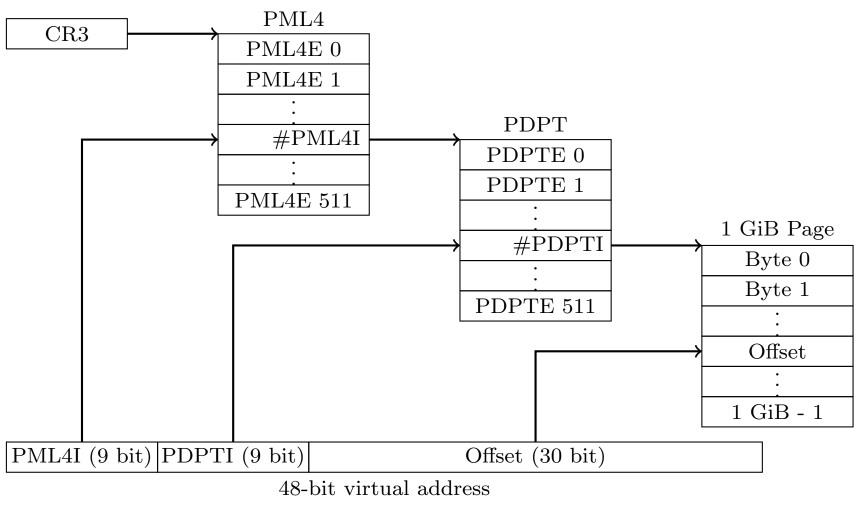


Figure 3.5: Virtual address resolution in x86-64 1 GiB paging mode according to page 4-30 of the Intel manual [Int12]

All page number fields are extended by 4 bits, compared to the 36-bit case. The reserved bits after the address are reduced accordingly. As the structure sizes do not differ from PAE paging mode, we still have the following structure in virtual addresses: The last 12 bits are used as an offset inside the 4 KiB page. The next 9 bits are used as the page table index. The next 9 bits are used as the page directory index. If the page directory entry has the page size bit set, the 12 bits offset and the 9 bits page table index together form the 21 bits offset for a 2 MiB page. Figure 3.6 shows virtual address resolution in x86-64 paging mode with 2 MiB pages and Figure 3.7 show address resolution with 4 KiB pages.

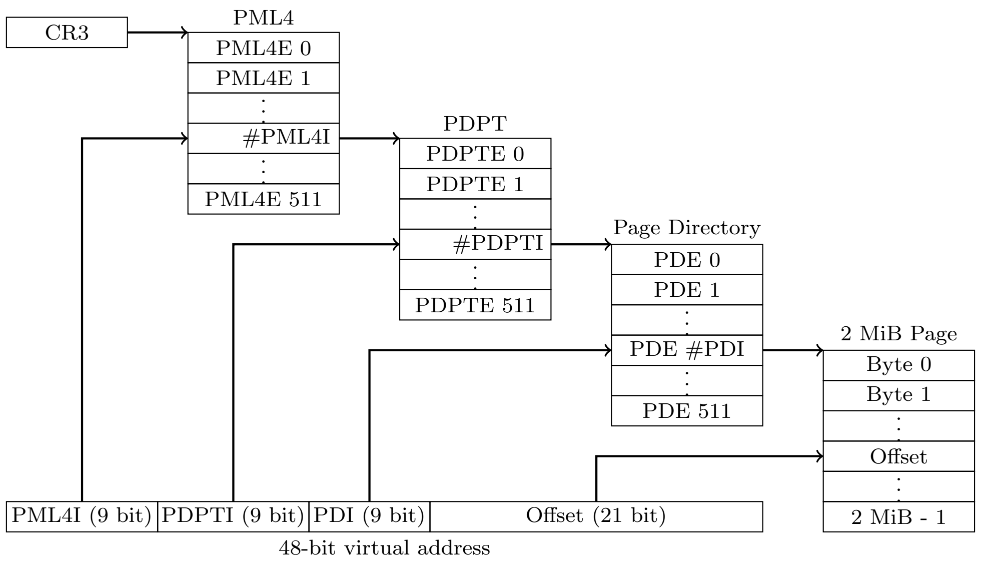


Figure 3.6: Virtual address resolution in x86-64 2 MiB paging mode according to page 4-29 of the Intel manual [Int12]

In either case, the next 9 bits form the page directory pointer table index. Thus, the page directory pointer table is extended from 4 entries to 512 entries. Each page directory pointer table entry now has a size bit too. If it is set, the 12 bits offset, the 9 bits page table index and the 9 bits page directory index together form a 30-bit offset for a 1 GiB page, provided that the CPU supports 1 GiB pages. Figure 3.5 shows virtual address resolution in x86-64 paging mode with 1 GiB pages. The last 9 bits are used as an index for the page map level 4, which is the newly introduced address translation layer. It holds 512 page map level 4 entries, each pointing to a page directory page table, managing a 512 GiB region.

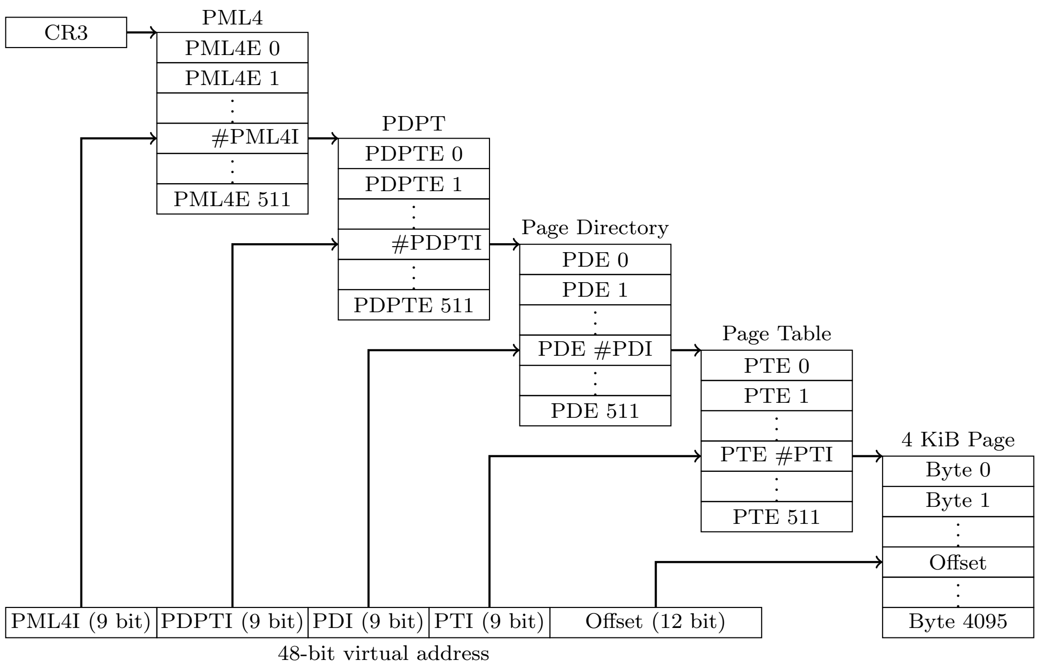


Figure 3.7: Virtual address resolution in x86-64 4 KiB paging mode according to page 4-28 of the Intel manual [Int12]

3.4 Global descriptor table

Besides paging there is segmentation as another address translation mechanism. It is a mapping from process and the type of memory access to a physical base address which is added to the accessed address. In protected and long mode segmentation allows access protection, for example a read only code segment or a segment which is not accessible for user processes. An invalid access to a segment will produce a CPU fault, i.e. a segmentation fault. Segments are defined using segment descriptors. Figure 3.8 shows the structure of a segment descriptor as described in the Intel manual in section 3.4.5 [Int12].

You can see that a segment descriptor has 64 bits. The lowest 16 bits form the lowest 16 bits of the segment limit. They are combined with the second segment limit field from bits 48 to 51, resulting in a 20-bit segment limit. That is, the maximum segment limit is 1 MiB. The segment limit defines where the segment ends. If the granularity bit is set, the segment limit is read as multiples of 2^{12}, that is 4 KiB. Thus, the segment limit can be defined in 4 KiB steps and the maximum segment limit is 4 GiB, which is the full 32-bit address space. Memory accesses above the segment limit are not allowed and produce a segmentation fault.

Bits 16 to 39 together with bits 56 to 63 form the 32-bit base address. Whenever accessing an address this base address will be added as an offset. The segment type defines whether the segment is a code or a data segment, its expansion direction and whether it is writeable. The descriptor type field defines whether this descriptor is a system descriptor, for example an interrupt descriptor or a task state segment descriptor. The descriptor privilege level defines from which ring this segment may be accessed. A kernel segment descriptor has always privilege level 0, whereas a user space segment has always privilege level 3. The segment present field is set if the segment is present. Bit 52 has no effect and can be used by the operating system. Bit 53 must be set to make a 32-bit code segment descriptor a 64-bit code segment descriptor. The operation size bit selects whether 16-bit addresses or 32-bit addresses are used.

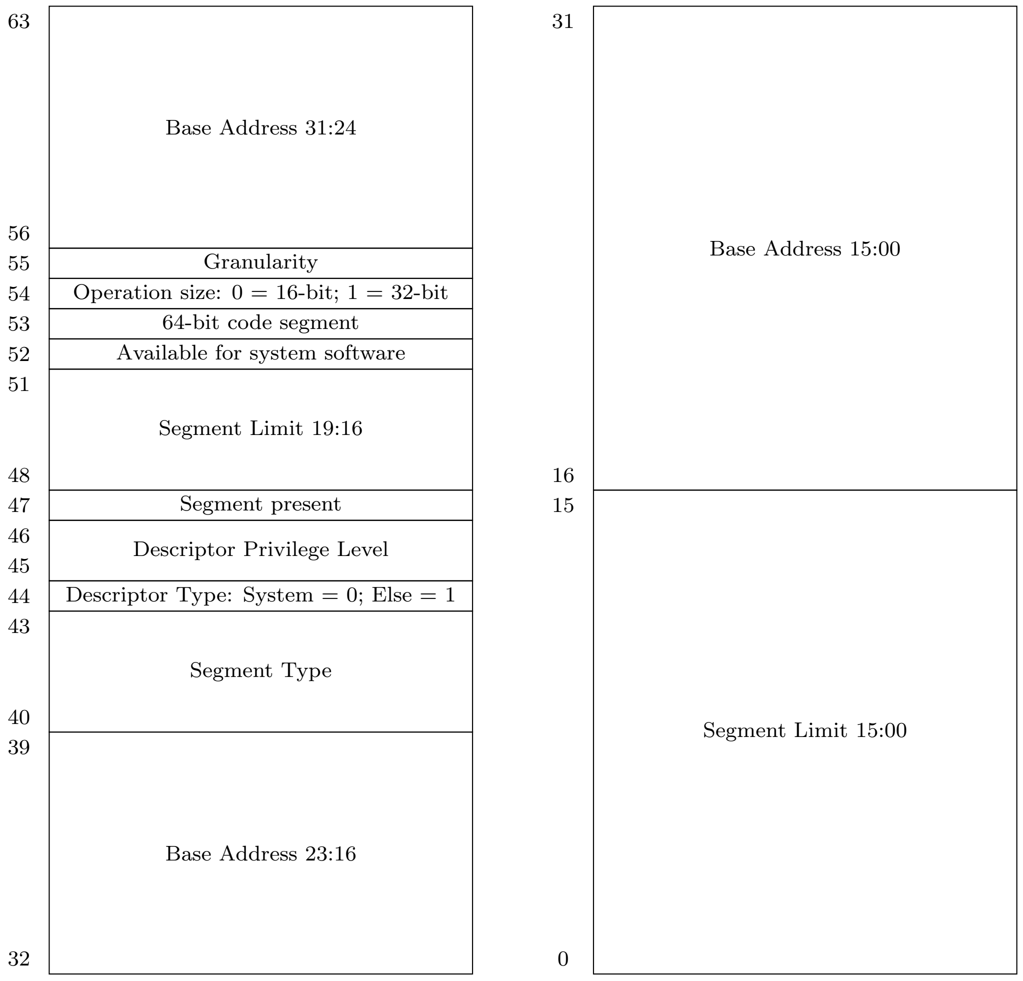


Figure 3.8: Structure of a segment descriptor

The global descriptor table contains all segment descriptors. The first segment descriptor in the global descriptor table is always the null descriptor, a descriptor with all bits set to zero. Trying to access memory through this descriptor always throws a CPU fault. This idea is similar to the idea of leaving the address range around `0x0` unused to detect invalid accesses. A typical x86 global descriptor table can be seen in Figure 3.9.

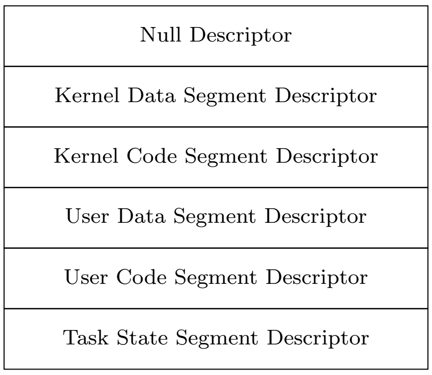

Null Descriptor
Kernel Data Segment Descriptor
Kernel Code Segment Descriptor
User Data Segment Descriptor
User Code Segment Descriptor
Task State Segment Descriptor

Figure 3.9: Typical global descriptor table in an x86 operating system

The CPU has several registers for memory accesses. The code segment (CS) is used for instruction fetches, the stack segment (SS) is used together with the stack pointer or the base pointer and the data segment (DS) is used for most other accesses and the extra segment (ES) which may be used for a compiler or operating system defined purpose. Additional segments FS and GS were introduced later to provide more general purpose segments. The operating system may use them if it needs a general purpose segment.

The segment descriptor is selected through the segment registers in the CPU. These are 16-bit registers, consisting of the privilege level in the lowest two bits, an indicator whether it references a local descriptor table or not and the descriptor table index in the remaining upper 13 bits. That is, you can have up to 8192 segment descriptors in the global descriptor table. The indicator bit will always be zero in case you use only the global descriptor table. The privilege level will be 3 if the descriptor describes a user level (ring 3) segment and it will be 0 if the descriptor describes a kernel level (ring 0) segment.

In x86-64 mode the global descriptor table stays almost the same, but system descriptors are expanded from 8 to 16 bytes. Although it is possible to mix both 8 byte and 16 byte descriptors, there are hardly any use cases for it. If you expand each 8 byte descriptor by an 8 byte null descriptor you can work with all descriptors as if they would have 16 bytes. Furthermore, x86-64 mode treats all segments except FS and GS as if they would have base address zero and the maximum limit. For FS and GS no limit checks are performed either. That is, all segments allow to address the whole physical address space (see Intel manual section 3.2.4 [Int12]).While segments have been used on 16-bit and even on 32-bit x86 systems to offset or limit memory accesses this is not possible in x86-64 long mode anymore.

The task state segment (TSS) is a special segment which can be used for hardware task switching in x86 mode. It contains fields for most of the CPU registers defining the task state. The CPU updates these fields by itself upon task switches. As each task has its own task state segment they can switch between each other either completely without the operating system or make task switching for the operating system much easier. Storing most of the CPU registers but not all is a significant drawback as thread data might get corrupted. Furthermore, there is a maximum of 8192 entries in the global descriptor table. If using hardware task switching you will not be able to have more tasks than that.

Most modern operating systems do not use hardware task switching. Linux removed hardware task switching support, because you can not change what happens during a hardware context switch and because hardware and software switching took the same amount of time. [BC05]

Hardware task switching is not supported anymore in x86-64 mode. However, you still have to define at least one task state segment descriptor in x86-64 mode, as the CPU expects to find one. And in legacy stack switching mode this task segment is still used. The format of the task state segment changed completely in x86-64 mode. Instead of the CPU registers defining the task state it now contains stack pointers for the privilege levels 0, 1 and 2 and 7 interrupt stack table pointers. Interrupt handlers can be set up to always work on one of the interrupt stacks from this table. This way the operating system can switch to a known working stack in case of a CPU fault to execute the CPU fault handling properly. If an interrupt occurs and a thread uses the according interrupt stack, then the operating system has to make sure that no interrupt using the same interrupt stack may occur.

When using the x86 legacy stack switching mechanism the interrupt handler works on the current stack or if there was a privilege change on the according stack from the task state segment. This allows us to enable interrupts while handling interrupts. This can be used for example in the case of syscalls or page faults.

3.5 Interrupt handling

The interrupt descriptor table defines what the CPU executes upon receiving an interrupt. There are three different kinds of descriptors the interrupt descriptor table may contain: the task gate descriptor which can be used for hardware task switching, the interrupt gate descriptor and the trap gate descriptor which can be used for instance for debugging. We will only discuss the interrupt gate descriptor in detail. The structure of an interrupt gate descriptor can be found in Figure 3.10.

Upon interrupt i the CPU uses the descriptor defined in the i-th entry of the interrupt descriptor table. The descriptor provides a segment selector for a code segment and an interrupt handler address so that the CPU can execute code starting from that address.

In x86-64 mode the interrupt gate descriptors are extended to 16 bytes as shown in Figure 3.11. The upper 32 bits of the interrupt handler address are added as bits 64 to 95. Bits 96 to 127 of the interrupt gate descriptor are reserved. The previously reserved bits 32 to 34 are now used as the index to the interrupt stack table.

When an interrupt occurs in x86 mode the stack pointer is only pushed to the stack if the interrupt changes the privilege level. In either case the registers `EFLAGS`, `CS`, `EIP` and the error code are pushed to the stack in this order. In x86-64 mode the stack segment selector and the stack pointer are always pushed to the stack. Furthermore, the stack segment register is set to `NULL` if the interrupt changes the privilege level. In x86 mode the stack pointer for the interrupt handler is taken from the TSS segment. In x86-64 mode it is either taken from the TSS segment or the interrupt stack table is used, depending on the interrupt stack table field in the interrupt gate descriptor.

The interrupt stack table allows to provide certain interrupts always with a clean and working stack even if the threads stack is not usable. However, nested interrupts will lead to problems if two or more threads operate on the same stack.

As the stack segment and stack pointer (`RSP`) are pushed onto the stack upon an interrupt in x86-64 mode, they are always popped from the stack upon interrupt return (`IRET`). Another important change is the stack alignment. While in x86 mode the interrupt stack frame was 4 byte aligned it is 8 byte aligned in x86-64 mode.

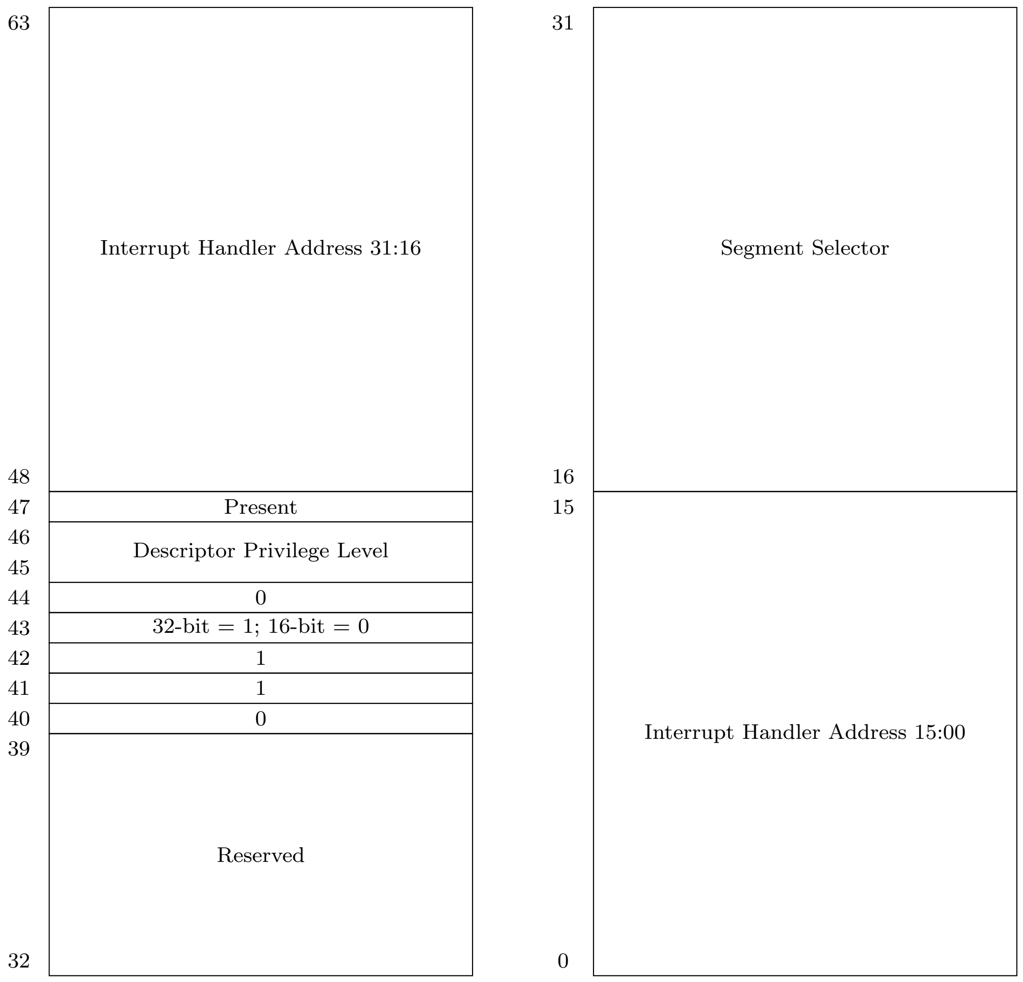

63
48
47
46
45
44
43
42
41
40
39
32
Interrupt Handler Address 31:16
Present
Descriptor Privilege Level
0
32-bit = 1; 16-bit = 0
1
1
0
Reserved
31
16
15
0
Segment Selector
Interrupt Handler Address 15:00

Figure 3.10: Structure of an interrupt gate descriptor in x86 mode according to page 6-15 of the Intel manual [Int12]. Some fields are required to be always 1 respectively always 0, in order to avoid mixing interrupt descriptors and other descriptors.

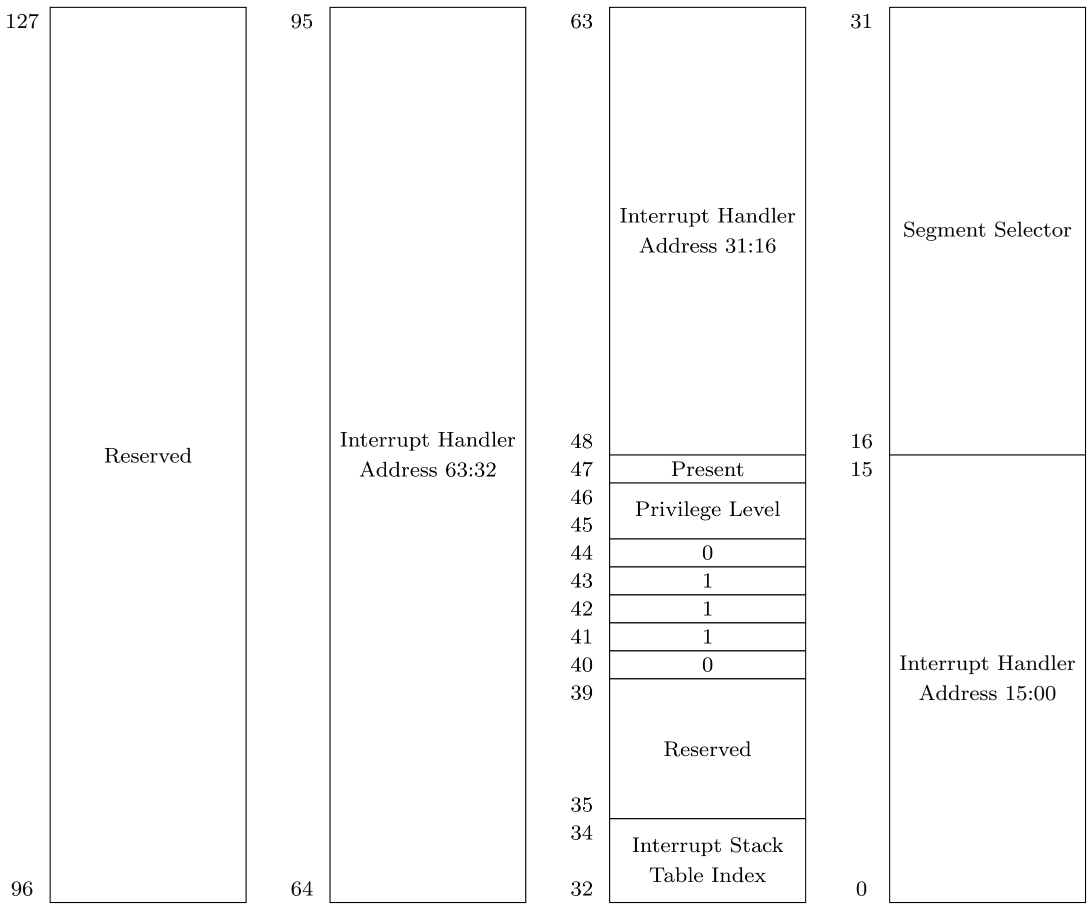

127
96
Reserved
95
64
Interrupt Handler Address 63:32
63
Interrupt Handler Address 31:16
48
47
Present
46
45
Privilege Level
44
0
43
1
42
1
41
1
40
0
39
Reserved
35
34
Interrupt Stack Table Index
32
31
Segment Selector
16
15
Interrupt Handler Address 15:00
0

Figure 3.11: Structure of an interrupt gate descriptor in x86-64 mode according to page 6-23 of the Intel manual [Int12]

3.6 Calling convention

If we have a function call in C the compiler has to translate this function call into assembly language. The x86 `call` instruction takes only one parameter, the new instruction pointer. The `call` instruction pushes the old instruction pointer to the stack. So the CPU has no mechanism to pass arguments to a function. Therefore, we need a calling convention, a standard way to pass arguments before executing the `call` instruction.

In this section we will describe the cdecl calling convention, which is used by many C compilers when building x86-32 binaries. The x86-64 introduced two new calling conventions, first the AMD64 ABI calling convention, second the Microsoft x64 calling convention which is used by Microsoft only. We will only describe the AMD64 ABI calling convention.

The ARM64 ABI calling convention defines how arguments of different sizes are passed. As the compiler implements the calling convention we only have to implement it in assembly language parts of the kernel. This includes parts where C and assembly language parts of the kernel interact. In most cases we only want to pass values to general purpose registers or retrieve general purpose register values. Therefore, we will only talk about passing variables which fit into a general purpose register.

The cdecl calling convention has a simple approach. First it starts a new stack frame, by pushing the `ebp` to the stack and setting the `ebp` to the current value of the `esp` afterwards. Then the function arguments are pushed to the stack in opposite order, that is from right to left. Now the `call` instruction is executed. The function finds the arguments on the stack. If the function pushes variables to the stack it has to pop these variables from the stack before returning. [Fog14]

When the functions returns using the `ret` instruction the return value should already be stored in the `eax` register. The CPU then pops the instruction pointer from the stack. Finally the caller has to clean up the stack. The stack frame now contains the function arguments and the old `ebp` value. The function arguments are then popped from the stack and the old `ebp` value is restored. Finally, the execution of the calling function is continued.

Stack operations are slow compared to register operations. Therefore, the AMD64 ABI calling convention uses a significantly different approach using registers to improve the performance. The first six arguments are copied into the registers `rdi`, `rsi`, `rdx`, `r8` and `r9`. The remaining arguments are pushed in opposite order to the stack, just as in the cdecl calling convention [MJAM13]. Then the `call` instruction is executed. The called functions stores its return value in the `rax` register. The caller has to clean up the stack, if any arguments have been pushed to the stack before.

The syscall convention used on x86 Linux or SWEB can also be used on x86-64. The change in the calling convention even leads to a simplification. As the arguments are not passed over the stack anymore, they do not have to be copied from the stack to the registers for the syscall. One can simply exchange the register values as expected by the kernel or make the kernel syscall interface expect the arguments as defined in the AMD64 ABI calling convention.

3.7 ELF Binary format comparison

The ELF binary format is the default format used by the `gcc`. Between the ELF32 and ELF64 format only a few small differences exist. Therefore, we will first describe the ELF32 binary format and afterwards point out where the ELF64 format differs.

The ELF32 binary format defines the unsigned 32-bit data types `Elf32_Addr` for addresses, `Elf32_Off` for file offsets and `Elf32_Word` as well as the signed 32-bit data type `Elf32_Sword` and the unsigned 16-bit data type `Elf32_Half`. An ELF binary always begins with the ELF header as shown in Listing 3.1. It starts with an ELF identification string which consists of a magic number, the ELF class (ELF32 or ELF64) and further information on how the binary should be parsed (i.e. parse integer values as MSB or as LSB). The ELF header also stores the address of the execution entry point and the location of the section headers and program headers in the binary. [Too93]

```
struct sELF32_Ehdr
{
  uint8          e_ident[EI_NIDENT];
  Elf32_Half     e_type;
  Elf32_Half     e_machine;
  Elf32_Word     e_version;
  Elf32_Addr     e_entry;      // execution entry point
  Elf32_Off      e_phoff;      // program header offset
  Elf32_Off      e_shoff;      // section header offset
  Elf32_Word     e_flags;
  Elf32_Half     e_ehsize;
  Elf32_Half     e_phentsize;
  Elf32_Half     e_phnum;      // program header count
  Elf32_Half     e_shentsize;
  Elf32_Half     e_shnum;      // section header count
  Elf32_Half     e_shstrndx;
};
```

Listing 3.1: ELF32 header as defined in SWEB

A program header as shown in Listing 3.2 defines where the binary code is located in the binary and where in the virtual memory it has to be placed for execution. For virtual memory areas where there is no data in the binary to load the size in memory `p_memsz` will be greater than the size in file `p_filesz`.

```
struct sELF32_Phdr
{
  Elf32_Word   p_type;
  Elf32_Off    p_offset; // offset in file
  Elf32_Addr   p_vaddr;  // virtual address
  Elf32_Addr   p_paddr;
  Elf32_Word   p_filesz; // size in file
  Elf32_Word   p_memsz;  // size in memory
  Elf32_Word   p_flags;
  Elf32_Word   p_align;
};
```

Listing 3.2: ELF32 program header as defined in SWEB

In order to load a byte for a certain virtual address `x` from the binary into virtual memory the operating system has to search through all program headers. If a program header has a virtual address smaller than `x` and `x` is in the range defined by the size in memory, then this is the right program header. If it moreover is in the range defined by the size in file, then the data can be loaded from the file. The position of the searched byte is `p_offset + x - p_vaddr`. If the byte is not in the range defined by the size in file then it is in the data segment or heap and it just has to be mapped without any data being loaded into memory.

The ELF32 binary format can contain sections with additional information, like debug information. A simple debugger format is the stabs format which provides symbol tables to look up virtual addresses. It allows to tell what symbol belongs to a virtual address. The Dwarf-2 debugger format is the default format used by `gcc`. It is more sophisticated and provides better abstraction and thus can be used on various platforms. "DWARF is also designed to be extensible to describe virtually any procedural programming language on any machine architecture". [Eag07]

The ELF64 binary format extends the types `Elf64_Addr` and `Elf64_Off` to 64 bits and adds new data types for 64-bit unsigned and signed words. In order to avoid padding some structs are reordered. The general structure and the way the data is loaded stays the same. Unfortunately the stabs debugger format is not available for the ELF64 binary format. Thus you have to switch to a different debugger format like Dwarf-2. [HI98]

Now that we have learned the differences between the x86-32 and the x86-64 architecture we are able to port an operating system kernel to the x86-64 architecture. In the next chapter we describe how we ported the SWEB kernel to the x86-64 architecture.

Chapter 4

Porting SWEB from x86 to x86-64

In the previous chapter we explained the differences between x86 and x86-64. In this chapter we will go through the changes in the SWEB source code to port it to x86-64 step by step. Therefore, we will first implement the x86 physical address extension. x86 with physical address extension enabled (x86-PAE) maps 32-bit virtual addresses to 36-bit physical addresses. The physical address extension leaves the boot up procedure almost unchanged.

From x86 to x86-64 the boot up procedure changes significantly. We will first describe how we changed the boot up procedure until the point where x86-64 paging is setup and then how we extended PAE paging to x86-64 paging. x86-64 paging maps 48-bit virtual addresses to 52-bit physical addresses. Finally we describe what we changed in the more abstracted code after calling the C entry function.

Comparing the source code of the x86-64 port to the source code of the x86 version of SWEB, we see that only small code changes were necessary. About 1500 lines of code were removed and 1550 lines of code either added or modified. These 1500 lines are divided into 450 lines of assembler code and 1050 lines of C++ code.

4.1 Implementing x86 physical address extension

The x86 physical address extension doubles the size of each page directory and page table entry, allowing much longer physical addresses. This way each page directory and page table may only contain 512 instead of 1024 entries in order to fit on a single page. The first two bits of the virtual address are then used as an index to the new page directory pointer table which provides up to 4 page directories, one for each gigabyte of virtual address space.

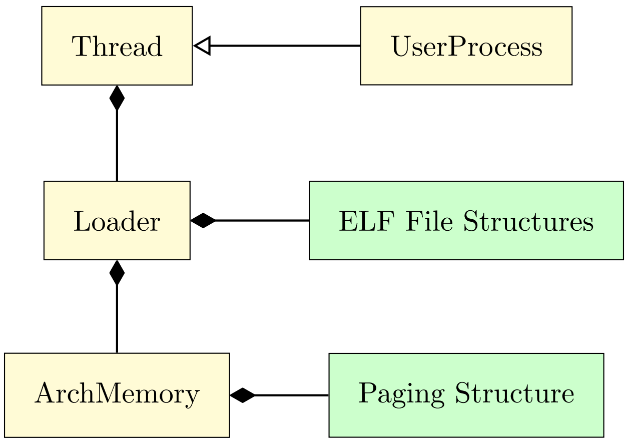


Figure 4.1: Class diagram: Relation between `UserProcess` and paging structure. Initially the paging structure of a process was stored in the `Loader` class, now it is stored in the `ArchMemory` class.

Until now, the page directory page number of a users process was stored inside a member variable in the `Loader` object of the `UserProcess`. The `Loader` lies in the common folder which should not contain any architecture dependent code. The page directory page number could be abstracted in this place and the abstracted code then moved to the arch folder, where the architecture dependent code is. The class diagram in Figure 4.1 shows the relation between `UserProcess` class and the paging structure of the process after this improvement. We will describe this in detail in Chapter 7.

In the 32-bit version of SWEB, the page number of the page directory page was stored in a `uint32` member variable. Now it is replaced by two variables: a pointer to the page directory pointer table (`page_directory_pointer_table_`) and an array containing this table. Just as page directories, page tables and pages have to be memory aligned to their own size, the page directory pointer table has to be aligned to its size (32 bytes) too. The `KernelMemoryManager` class implements the dynamic memory functions in the SWEB kernel. Its implementation does not care about the alignment of an object. It may occur that the object containing the page directory pointer table is not 32 byte aligned.

The `g++` compiler provides variable attributes for memory alignment, but that has no useful effect in this case. The variable will then be aligned relative to the beginning of the object (or struct). But, if the object itself is not 32 byte aligned the member variable will not be 32 byte aligned either. Looking at the next step towards x86-64 paging the page directory pointer table will be extended from 32 bytes (4 pointers) to a full 4 KiB page. Pages are always page aligned, so this problem will just disappear anyway.

Doubling the array size to 64 bytes is a quick and simple solution to this temporary problem. Therefore, we need the second variable to point to a 32-byte aligned memory location within the 64 bytes array. The memory location is calculated by a bit mask operation `(page_dir_pointer_table_space_ + 0x20) & ~0x1F`. The bitwise and-operation with `~0x1F` sets the last 5 bits of the address to zero. This is a round down to the next 32 byte aligned address. Therefore, we add 32 bytes to stay within the 64 byte array.

The paging data structures are basically just extended by 64 bits per entry in order to provide space for the longer physical addresses. Page tables and page directories now contain only 512 entries in order to fit on a single 4 KiB page.

Until now, upon process creation a page directory page was allocated and the second half of the kernel page directory page copied to it. This way the (not user space accessible) virtual address space from 2 GiB upwards was the same for all processes and the kernel. Now this changed as one page directory manages only 1 GiB instead of the full 4 GiB virtual address space. Thus with physical address extension enabled, we just set the first two page directory pointer table entries to zero and copy the other two from the kernel page directory pointer table.

The `ArchMemory` class provides methods for mapping pages and for resolving mappings. In all cases we had to add the new layer and check whether a page directory is present. The method `mapPage` maps a virtual page to a given physical page. In order to do so it resolves the virtual address and checks whether a page directory is present, if it is not, we allocate a physical page and map it in the page directory pointer table. Afterwards the process is unchanged: If `mapPage` should map a large page, we just map it into the page directory. Otherwise we check whether the page table is present. If it is not we allocate a physical page and map it in the page directory. Afterwards the mapping is set up in the according page table entry.

A little change in some thread functions as well as in the `pageFaultHandler` was necessary. With 32-bit paging the CR3 register held the physical address of the page directory. With PAE paging the CR3 register holds the physical address of the page directory pointer table. If we change something in the mapping it might occur that the according translation look aside buffer (TLB) entry is not invalidated properly. As SWEB has no focus on optimization we simply invalidate the whole TLB. This helps keeping the number of pitfalls for developers small. In order to invalidate the TLB we just copy the value from the CR3 register to EAX and back to CR3 before the end of a page fault. There might be some situations where this is not necessary, but in almost all situation it is. Upon an update of the CR3 register the memory management unit updates the memory mappings and flushes the TLB.

The space for the kernel page directory and the kernel page table is allocated in the bss section of the kernel binary. Instead of the one kernel page directory we already have, we need four to manage the same virtual address space and instead of four page table pages we need eight page table pages. Additionally we need a 32 byte aligned 32 byte array for the kernel page directory pointer table. The initialization of these data structures is implemented in C functions in the file `init_boottime_pagetables.cpp`. First the four page directory pointer table entries are set to present and the physical page numbers of the four page directory pages we just allocated in the bss section.

We set up a 1:1 mapping which will be used while booting and removed before the kernel finished booting. As this was done by using 4 MiB pages we have to double the number of mapped pages here in order to map the same amount of memory and maintain the old behavior of SWEB. The kernel is also mapped to the virtual address 2 GiB. That is what we need the eight page tables for, as this is not done using 4 MiB pages but 4 KiB pages instead. eight page tables allow a kernel size of 16 MiB.

Afterwards the frame buffer is set up and the whole GiB from 3 GiB upwards in virtual address space is mapped to the first GiB in physical memory. This will also be a temporary thing. We will use 1 GiB pages for this when switching to x86-64 paging.

In 32-bit paging mode we already had the PSE bit (bit 4) of the CR4 enabled. The PSE bit determines whether the page size bits of the page directory entries are enabled. The PSE bit is ignored in PAE and x86-64 paging modes, so we can safely just ignore it and not set it to any value. To enable PAE paging mode we have to set the PAE bit (bit 5) of the CR4 register. The next time we modify the CR3 register the memory management unit will switch to PAE paging mode.

So this is the final thing we do: Setting the CR3 register to the physical address of the kernel page directory pointer table and enable PAE paging.

4.2 Booting a 64-bit kernel

To enter 64-bit long mode we have to first look at the boot up procedure. There are several ways to enter 64-bit long mode. The first is to build a 64-bit kernel and use a boot loader which supports booting directly into long mode. SWEB currently uses GRUB Legacy 0.94 as its boot loader, which does not support this. Oracle developed a modified version of GRUB [Ora13], capable of loading 64-bit kernels. To the best of our knowledge, Solaris is the only operating system using this modified GRUB. Therefore, we decided to stay with the old boot loader.

Two other variants require having an ELF32 part and an ELF64 part. Many elements of those two variants overlap so we will explain them only once. The first variant is using a 32-bit loader which loads the 64-bit kernel binary and then jumps to the 64-bit kernel entry point. The second variant loads a 32-bit kernel binary which contains a 64-bit kernel binary so we do not have to parse the 64-bit kernel binary.

4.2.1 32-bit loader - 64-bit kernel

The idea of this variant is to build a tiny 32-bit kernel which does nothing but parse and load the ELF64 binary in order to jump to the 64-bit long mode [OsD14].

Both the loader binary and the kernel binary are still compiled using `gcc` and `g++`. We added some compiler flags which will build the kernel binary in a way more suited to our needs. First is `-ffreestanding`. It tells the compiler that this binary will run in an environment where "the standard library may not exist" [GCC13b]. It should always be added when compiling an operating system kernel. Second is `-mcmodel=kernel` [GCC13a].

The default model is `-mcmodel=small`. This model tries to produce small 16-bit or 32-bit jumps. This allows executing code within the address space between 0 GiB and 2 GiB. Using the kernel code model, the compiler links the binary in the negative 2 GiB of the address space. That is the 2 GiB from the upper end of the address space downwards. On 32-bit architectures the negative 2 GiB address space is equivalent to the upper 2 GiB address space.

There is also the medium code model which links the binary in the address space between 0 GiB and 2 GiB and the large model, both of which are even more cautious with optimizing jumps. The large model is not implemented by `gcc` at all. Either way, using a less restricted model slows down the code and makes it larger. Finally, we added some flags which disable the red zone (a temporary data storage on the stack) as well as the MMX, SSE2, SSE3 and 3D-Now extensions. They would all need some kind of initialization, which is not implemented in SWEB.

Until now, SWEB was linked using the ld linker tool. We changed that to linking using `gcc`. The `gcc` compiler provides more linking options, like the `-mcmodel` flag which we already used for code compilation. When porting x86 code to x86-64 you might stumble over the error message `relocation truncated to fit: R_X86_64_PC32 against symbol` when linking the kernel binary. When linking a 32-bit kernel binary in the negative 2 GiB address space, the kernel will start at `0x8000 0000`. This address and all addresses up to 0GiB will fit into 32 bits. When linking a 64-bit kernel binary in the negative 2 GiB address space, the kernel will start at `0xFFFF FFFF 8000 0000`. However, for this address we need 64 bits. The error message above tells us that with the relocation to the negative 2 GiB address space some operation loses the first 32 bits of the 64-bit address of the symbol. This should never occur in any code your compiler produces. However, it may still happen. For example, if you have an instruction of the form `mov eax,dword[somesymbol]`, which should be `mov rax,somesymbol` instead or a section in your linker script in which you reference a symbol from a different section which resides too far away in memory but does not support 64-bit addressing. However, if you compile the kernel in a way such that all sections lie within the first 32 bits, you should not get relocation errors either.

The kernel loader is a significantly stripped down version of the old 32-bit kernel. It parses the multiboot data structures provided by GRUB, the ELF64 kernel binary, sets up paging and long mode and finally jumps into long mode. The first thing that changed in the boot procedure is the x86-64 check. To check whether this mode is available we first have to get the maximum input value for extended function CPUID information. This is done by a `CPUID.8000 0000h` instruction. If the CPU supports long mode, the eax register will have a value greater than `0x8000 0000` after execution of the CPUID instruction. Subsequently we check whether x86-64 long mode is available. This is done by a `CPUID.8000 0001h` instruction. It will write whether x86-64 long mode is available or not into the 29th bit of the edx register. If everything is alright, we will continue booting the loader just as in the x86 version of SWEB.

The kernel loader is very simple. It checks the ELF64 headers for errors. If there are no errors, it returns the lower 2 GiB address of the kernel entry point shifted by the offset of the multiboot header. This is the physical address of the entry point. Before doing a far jump to the entry point we still have to set up paging. We will leave that out for now and explain it in the next section.

As the loader binary is 32-bit code you will probably not be able to compile a `call far` or `jmp far` into it. So you cannot jump to a long mode 32-bit address provided in a register. As a workaround you can generate the op code yourself and execute it. A more beautiful workaround is to use `retf`. `retf` is executed when a function returns which was called by a `call far`. It pops the 32-bit return address and the segment selector from the stack and far jumps to that location. So we can just push the segment selector to the stack and the 32-bit entry point address of the 64-bit kernel and far jump into long mode by executing `retf`.

Until this point we have not yet parsed the multiboot remainder of the kernel binary. Parsing it requires long mode as we want to place our kernel at -2 GiB, which is close to the upper end of the 64-bit address space. After that, we will initialize paging properly and boot the higher level parts of the kernel.

4.2.2 64-bit kernel linked into a 32-bit binary

The second approach we tried is more convenient when booting a 64-bit kernel: Linking an ELF64 kernel binary into an ELF32 kernel binary using `objcpy -O elf32-i386`. First we compile the ELF64 kernel as usual, but add a 32-bit code section to it. Our entry point will be in this section. We will boot up as in the other variant until setting up paging.

While booting we will set up a very simple memory model using one page map level 4, which maps its first entry to a page directory pointer table. This page directory pointer table maps its first entry to a kernel page directory, which maps its first entry to the first two MiB of physical memory. We will cover how x86-64 paging is implemented in the SWEB kernel in the next section.

To enable 64-bit paging mode we enable PAE paging mode first, and directly afterwards set the long mode enable (LME) bit of the extended feature enables register (EFER) to 1. This is done by reading the model-specific registers, putting `0xC000 0080` into the edx register and using the instruction `rdmsr`. The LME bit is the 7th bit. We set it to 1 and write it back to the model-specific registers using the instruction `wrmsr`.

Before switching to long mode we have to set up a TSS segment and the new 64-bit global descriptor table. We actually do not use the TSS segment, but x86-64 requires to have one. On x86-64 each global descriptor table entry doubled in size. All base addresses are required to be zero and all limits to maximum, that is all segments span over the whole address space.

This approach allows us to jump directly from the 32-bit code to the 64-bit label, without any workarounds: `jmp LINEAR_CODE_SEL:(entry64-BASE)`.

4.3 x86-64 paging in SWEB

x86-64 paging is an extension to the PAE paging we already implemented. Physical page numbers are generally extended from a minimum of 36 bits to at least 40 bits (and maximum 52 bits) while the reserved bit fields behind them are shortened by 4 bits. Apart from this, the old data structures did not change. 64-bit paging, as we implement it, translates 48-bit virtual addresses to physical addresses[1]. The address resolution mechanism has to be extended to handle 16 bits more than before. This is done by first increasing the number of page directory pointer table entries from 4 to 512. It now uses 9 bits of the virtual address as an index instead of 2 bits as before. The remaining 9 bits are mapped by a new mapping layer, the page map level 4. It is structured very similar to the page directory pointer table. Page directory pointer table entries get the size bit in turn, allowing to map a 1 GiB page, if the hardware supports it.

The two variables we introduced with PAE paging, the actual page directory pointer table and the pointer to it, are now replaced by a single one, the page map level 4 physical page number.

In PAE mode, we set the first two page directory pointer table entries to zero and copy the other two from the kernel page directory pointer table. As we have now introduced the page map level 4, this moved to a higher level. One page map level 4 entry manages a 512 GiB memory region. We only need 2 of these entries, one for user space starting from `0x0` upwards and one for kernel space lying in the last 512 GiB memory region of the virtual address space. This last entry is set to the same page directory pointer table for all processes.

The data structures allocated in `arch/loader/source/boot.s` now have to be changed too. We reserve memory for two page directories, two page directory pointer tables and one page map level 4.

When setting up paging, we first set up the page map level 4. The first entry will point to the first page directory pointer table, the last entry will point to the second page directory pointer table.

[1]Please note that the x86-64 standard is designed to allow 64-bit virtual addresses and more recent CPUs support virtual addresses longer than 48 bits already.

Furthermore, we want a identity mapping of physical memory. An identity mapping linearly maps a number of pages in virtual memory to the same number of pages in physical memory. In 32-bit SWEB this mapping started at `0xC000 0000`. Each virtual page starting from `0xC0000` was mapped linearly to a physical page starting from `0x0`. In 64-bit SWEB it will start at `0xFFFF F000 0000 0000`, that is page map level 4 entry 480. We can map this entry to the same page directory pointer table as entry 0 which is only used while booting.

Second the page directory pointer tables are set up. We map the first entry, that is the region from 0 GiB to 1 GiB of virtual memory, to the first page directory. We will use it for an identity mapping the first GiB of physical memory to the first GiB of virtual memory while booting. After booting, this identity mapping will only be available starting from `0xFFFF F000 0000 0000`.

In the second page directory pointer table we map index 510, that is the region between -2 GiB and -1 GiB (`0xFFFF FFFF 8000 0000`) to the first GiB of physical memory. This is done by mapping this entry to the first page directory.

Figure 4.2 shows the layout of the 64-bit virtual address space. Figure 4.3 shows how the lower half is used and Figure 4.4 shows how the upper half is used in the SWEB kernel.

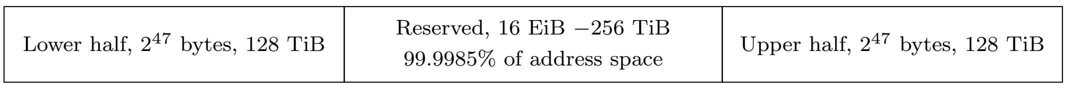

Lower half, 2^{47} bytes, 128 TiB
Reserved, 16 EiB −256 TiB
99.9985% of address space
Upper half, 2^{47} bytes, 128 TiB

Figure 4.2: Virtual memory: Lower half and upper half

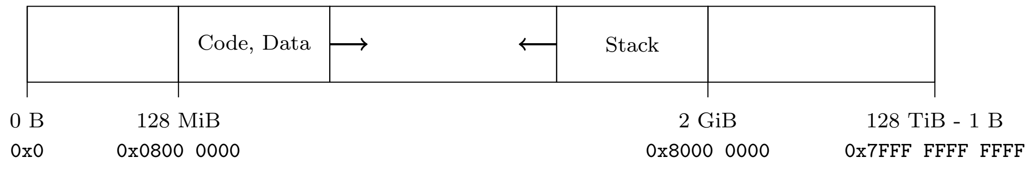


Figure 4.3: Virtual memory: Lower half layout in SWEB for an example binary

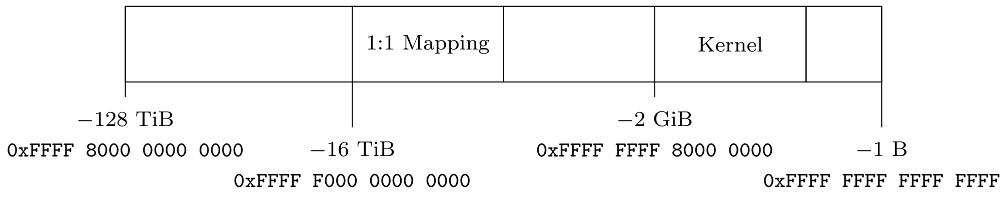


Figure 4.4: Virtual memory: Upper half layout in SWEB

In the first page directory all entries are 2 MiB pages which are identity mapped from virtual to physical memory. The second page directory, which is mapped starting from

-2 GiB, has only its first two entries set up. These two entries map the first 4 MiB of physical memory, where GRUB loaded the kernel image into memory.

In case you want to use a frame buffer you have to map it at this point as well.

As we have changed memory mappings and segment descriptors to ones that use the -2 GiB offset we have to reload them before jumping to a label with offset -2 GiB.

After we have done that we will remove the first two page map level 4 entries (the entire lower half of virtual memory). Now we can call the `startup` C-function.

4.4 Boot up procedure after C entry

The 32-bit version of SWEB uses the stabs debugger format for debug information. While booting the stabs symbol table is parsed providing the kernel developers a simple stack trace function. When porting to x86-64 we had to switch from this debugger format to the newer Dwarf debugger format. Therefore, the stack trace function has to be rewritten. SWEB currently provides no stack trace function in 64-bit mode. However, debugging the 64-bit version with GDB is possible.

Many addresses had to be changed from the old 32-bit address to the new 64-bit addresses, especially the kernel location and the address of the identity mapping.

As we described in Section 3.5 the interrupt descriptor table changed its format and thus we had to change its initialization. The interrupt descriptor table is initialized in the `InterruptUtils` constructor. The interrupt gate descriptor struct was extended by the new fields as can be seen in Listing 4.1.

```
struct GateDesc
{
  uint16 offset_ld_lw     : 16;
  uint16 segment_selector : 16;
  uint8 ist               :  3;
  uint8 zeros             :  5;
  uint8 type              :  4;
  uint8 zero_1            :  1;
  uint8 dpl               :  2;
  uint8 present           :  1;
  uint16 offset_ld_hw     : 16;
  uint32 offset_hd        : 32;
  uint32 reserved         : 32;
}__attribute__((__packed__));
```

Listing 4.1: Interrupt gate descriptor struct

The behavior when handling interrupts changed as well, as described in Section 3.5: registers SS and RSP are always pushed onto the stack and the stack is 8-byte-aligned. If there was a privilege change, which is the case upon switching from or to user processes, the new SS register value is set to zero. Thus we have to set the SS to the according data segment descriptor right after a context switch.

Although there is the new interrupt stack switching mechanism using the interrupt stack table, we do not use it for any interrupt right now. The interrupt stack switching mechanism could be utilized to provide a more stable environment upon CPU faults, because the operating system does not have to rely on the stack of the thread. Figure 4.5 compares how the stack looks upon interrupt entry after a non maskable interrupt (a CPU fault) and a normal interrupt.

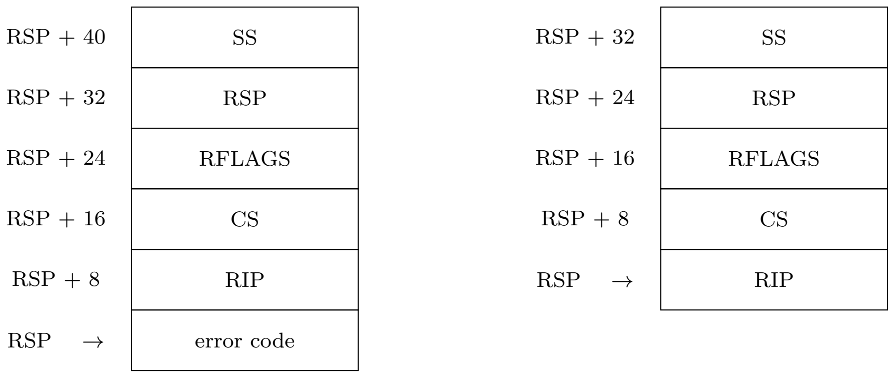


Figure 4.5: Stack when handling a CPU fault (left) vs. when handling an interrupt (right), both in case of a privilege change

Interrupt handlers have assembly language entry points. Before calling the corresponding interrupt handler implemented in C, all CPU registers are saved in a struct in the current thread. In 32-bit SWEB the instructions pushad and popad where used in this context. The pushad instruction pushes all registers on the stack, whereas the popad instruction pops all registers from the stack. Both instructions have been removed from the instruction set. Therefore, we implemented these two instructions using a nasm macro. By saving all CPU registers in a struct in the current thread, we can execute kernel code and restore the previous state of the CPU afterwards. As both, the instructions to push the registers on the stack and the struct containing the register values have changed, the according assembler routines had to be rewritten.

After these changes, the kernel is set up completely and will run in 64-bit mode.

4.5 64-bit user processes

In order to load and execute user processes, SWEB has the Minix file system layer implemented in the kernel. User binaries can be stored on a Minix partition which will be mounted by the kernel. The 32-bit version of SWEB uses the ELF32 binary format for both the kernel binary and user space binaries. The kernel binary is loaded by the boot loader. For the user space binaries the 32-bit version of SWEB provides an ELF32 binary loader.

When porting SWEB to 64 bits the Minix file system layer as well as the underlying IDE and ATA drivers worked with minimal changes.

In the 64-bit version of SWEB we use the ELF64 binary format. The ELF binary format and the few differences between ELF32 and ELF64 are described in Section 3.7. The 32-bit `Loader` class contained the methods and structs for the ELF32 binary format. Only a few source code changes were necessary to make the `Loader` work with the new format.

As the calling convention changed, the interface between kernel space and user space changed. First, the passing of arguments to a user space program is not done through the stack anymore but through registers. Second, syscalls now take the arguments in the registers according to the AMD64 ABI and pass them as is to the kernel.

With regard to multi-platform compatibility a `types.h` was added to provide one file which contains all data type definitions.

4.6 Debugging 64-bit SWEB

As explained in the previous sections, the binary format of the kernel has changed from ELF32 to ELF64 which is linked into an ELF32 binary. The debugger format of the kernel has changed from stabs to Dwarf.

There are two convenient debugging methods for the 32-bit version of SWEB. First, there is debugging using GDB. This works by starting SWEB with Qemu or Bochs with GDB-stub enabled and then starting GDB, connecting it to the GDB-stub. Developers may then debug the kernel with GDB. The second method is the stack trace which allows printing the stack trace of a thread by calling the `printStackTrace` method of the `Thread` object.

The stack trace method is used very often, as it does not influence timing as much as a debugger does. Therefore, it allows searching for bugs which only occur with a certain timing. Debugging gets a lot harder without this method. In order to provide a stack trace functionality the kernel has to parse the `gstabs2` symbol table from the kernel binary during the boot process.

SWEB stores a sorted table in memory, which maps function pointers to function names. In order to retrieve a function name for an instruction pointer p the table can be searched for the first entry smaller than or equal to p. This entry maps to the name of the function p points into.

A stack trace is generated using the instruction pointer (`eip/rip`) and the base pointer (`ebp/rbp`). The instruction pointer is used to find out which function is currently being executed. The base pointer points to the current stack frame. Each stack frame contains the return address and the previous stack frame (the previous `ebp/rbp` value). The stack trace is generated by iterating through this linked list of stack frames and storing all return addresses. When the stack trace is printed the return addresses are translated to function names using the symbol table we parsed previously.

The stabs debugger format is not available for ELF64 binaries. Therefore, parts of this method have to be implemented from scratch in the 64-bit version of SWEB.

Debugging using GDB works in the 64-bit version of SWEB, but it has one important limitation. As we linked the ELF64 binary named `kernel64.x` into an ELF32 binary named `kernel.x` you have to decide what you want to debug. The `kernel.x` binary only contains address and debug information which can be used before enabling long mode. The `kernel64.x` binary only contains addresses and debug information for the kernel in the higher half, starting at -2 GiB. Before long mode is enabled you won't be able to connect GDB using the `kernel64.x` binary to the Qemu or Bochs GDB stub. In the 32-bit version of SWEB Qemu was started in a stopped emulation mode for debugging. The emulation was then continued through the GDB after it connected. Now this is not possible anymore, because GDB may not be connected before long mode is enabled.

This makes this debugging method much less convenient as you have to connect GDB while SWEB is already in the middle of the booting. If you want to debug a situation which occurs while booting, you can easily miss the moment when to connect GDB. Furthermore, breakpoints do not work until GDB is connected, that is you can't set a breakpoint right after long mode is enabled because you can't connect GDB before that.

We will have to work on both methods to make them as convenient as in the 32-bit version of SWEB.

4.7 Expected impact on education using SWEB 64-bit

SWEB is an operating system designed to be used in education. In context of the course "Operating Systems" at the Graz University of Technology, students extend this operating system by small components. While designing and developing these components design decisions have to be made. Until now there has only been the 32-bit variant. In this chapter we talk about the design decisions and problems the students faced while working with the 32-bit variant and how that changes when working with the 64-bit variant. We also try to list expected new questions and problems. One major drawback is that the page directory, page table structure was already hard to understand for undergraduates and the four level paging of the x86-64 makes this even harder to understand.

4.7.1 Multi threading

Part of the first assignment is to implement multi threading for user processes. The students are required to implement the user threads in the kernel. The interface is the POSIX `pthread_create` function. It starts a thread executing a given function `start_routine` and an argument. As we have described in Section 3.6 the calling convention changed. Instead of pushing the argument on the stack, students now have to put it in the `rdi` register, which is easier as no stack offset calculation has to be done.

One important design decision is the placement of the stack. Currently stacks are only one page in SWEB, but some students like to implement growing stacks. If a user program accesses the next lower stack page this page should be mapped by the operating system. On the other hand, new threads may start and need an address for their stack. A common solution is to limit the stack to a certain maximum size and let it only grow to that maximum. With a large 48-bit address space there is far more space for stacks than before. Therefore, the maximum size can be significantly higher, resulting in less headache about that design decision for the students.

Another design decision is to decide how to generate the thread ids. A simple approach is to use a counter giving each thread a new thread id, incrementing this counter. Unfortunately, a 32-bit thread id overflows after a while, so you have to deal with thread id reuse and handle thread ids that are still in use. A 64-bit thread id is very unlikely to overflow - you would need to create 2^{38} threads per second for one year in order to get an overflow. Therefore, this can definitely neglected now and students could implement the simple approach without any overflow handling. However, they still have to implement stack reuse, as they would reach a stack position overflow much earlier.

In section 7.1, we will discuss how a platform independent implementation could abstract the user process stack allocation, such that the 48-bit virtual address space is used on the x86-64 architecture, while providing a uniform interface for user process stack allocation on all architectures.

4.7.2 Fork

The fork syscall is another task of the first assignment. Part of this task is to iterate over the page directory and copy it or mark it for copy on write. As the paging mechanism was extended by the page directory pointer table and the page map level 4, students would have to implement these two layers as well. Iterating over more additional layers is conceptually harder. Programming errors in this code are very common. Students will need significantly more time to solve and debug this task.

On the contrary, process id generation changes just like the thread id generation: a simple counter without any overflow protection will suffice.

4.7.3 Clock/Sleep

When implementing the `clock` or `sleep` syscalls many students use the time stamp counter, which is a 64-bit value. When processing this value into a unit they can work with, many students needed a 64-bit division which is not available on 32-bit x86 architectures. A common workaround was to implement the division oneself or use 64-bit division functions from open source software projects. The x86-64 supports 64-bit division natively, so this task might get more straightforward as well.

4.7.4 Virtual memory

The main task of the second assignment is to implement swapping. In an environment with only a few megabytes of memory physical pages are swapped out to the hard disk. Part of this task is the implementation of a page replacement algorithm. While the two new levels in the paging mechanism make the task more complex, there are now up to 14 ignored bits in the page table struct instead of 3. These ignored bits can be used for information of the page replacement algorithm or for information regarding the page (swapped out, copy on write, shared, etc.).

To find the swapped out pages on the hard disk many groups store the position on the hard disk in the page base address field. As this field has more bits now, this simple solution can no support much larger swap devices. Swapping of page tables might get be more relevant now as each page table manages only 2 MiB instead of 4 MiB.

4.7.5 Shared memory

Shared memory is an additional task of the second assignment. On x86-32 it is regarded as good practice to place the shared memory segments in the same memory area as the stacks. Using a first fit strategy a virtual memory area can be reserved for a new stack or a new shared memory segment. This is possible on x86-64 as well, but determining a fixed region of the virtual memory as the shared memory region is also more tolerable than in a 32-bit address space. Besides that, the aforementioned page table bits may be used for information regarding shared memory as well.

Chapter 5

Comparison of x86-32 and ARM-v5

In this chapter we describe the ARM-v5 architecture compared to the x86 architecture from an operating-system developer's point of view. While the most popular CISC (Complex Instruction Set Computer) architectures belong to the x86 architecture family, the most popular RISC (Reduced Instruction Set Computer) architectures belong to the ARM architecture family. The ARM-v5 architecture was developed in in the late 1990s. Newer architectures like the ARM-v6 or the ARM-v7 provide backward compatibility to the ARM-v5.

CPUs based on the same ARM architecture vary widely in their features. We initially ported SWEB to the ARM Integrator/CP base board with an ARM926EJ-S CPU which is an ARM-v5 CPU. The ARM926EJ-S CPU has a memory management unit, whereas other ARM-v5 CPUs may have none. We will learn how ARM-v5 paging works, compared to the x86 paging mechanism. We will point out differences in the most similar paging scenarios, which is using 1 MiB on ARM and 4 MiB pages x86-32 respectively using 4 KiB pages on both ARM and x86-32.

In C and C++ there is very little difference to programming on a CISC architecture like the x86. Finally, we will take a look at differences regarding the calling convention and interrupt handling.

In the following chapter we will use the knowledge we acquire in this chapter, on the differences between the two architectures, to implement an ARM-v5 port of SWEB. This ARM-v5 port will run emulated using `qemu` and on real hardware using a Raspberry Pi.

5.1 ARM-v5 execution environment

The ARM architecture has 31 general purpose registers of which 16 registers are accessible at any time, depending on the execution mode. The ARM Architecture Reference Manual DDI 0100E [ARM00] lists seven different execution modes in chapter 2.2:

- User mode: normal program execution,
- system mode: a privileged mode for operating system tasks,
- supervisor exception mode: a protected mode for the operating system,
- abort exception mode: used for virtual memory (protection) faults,
- undefined exception mode: used for instructions not supported by the CPU[1],
- interrupt mode and
- fast interrupt mode.

All modes have access to registers `r0` to `r15`, which are mapped differently depending on the execution mode. Registers `r0` to `r7` are always the same physical registers in all modes. Registers `r8` to `r12` are the same physical registers in all modes, except for the fast interrupt mode. In fast interrupt mode, these register identifiers are mapped to separate physical registers. `r13` and `r14` are the same physical registers in user mode and system mode, but all other modes have their own physical registers `r13` and `r14`. Finally, the register `r15` is the same physical register in all modes.

The register `r15` is the program counter register and can also be accessed in assembly code using the alias `pc`. The `pc` register is the equivalent to the `eip` register of the x86-32 architecture. The register `r14` respectively `lr` is used as the link register. When calling a function this register holds the return address. The register `r13` respectively `sp` is the stack pointer register, equivalent to the `esp` register on the x86.

The current program status register (`cpsr`) is available in all execution modes. The saved program status register (`spsr`) is available in all interrupt modes. The `cpsr` register is equivalent to the `eflags` register on the x86 architecture. A comparison of the two registers can be found in Figure 5.1. Both registers contain bits for arithmetic operations and both registers contain an interrupt enable/disable bit. Both registers contain bits to specify the execution mode: The `eflags` register contains bits to specify the current privilege level, the `cpsr` register contains mode bits to specify the execution mode.

[1]Undefined instruction exceptions facilitate execution of software which uses instructions not supported by the processor. The operating system can emulate the instruction in software and return from the fault afterwards.

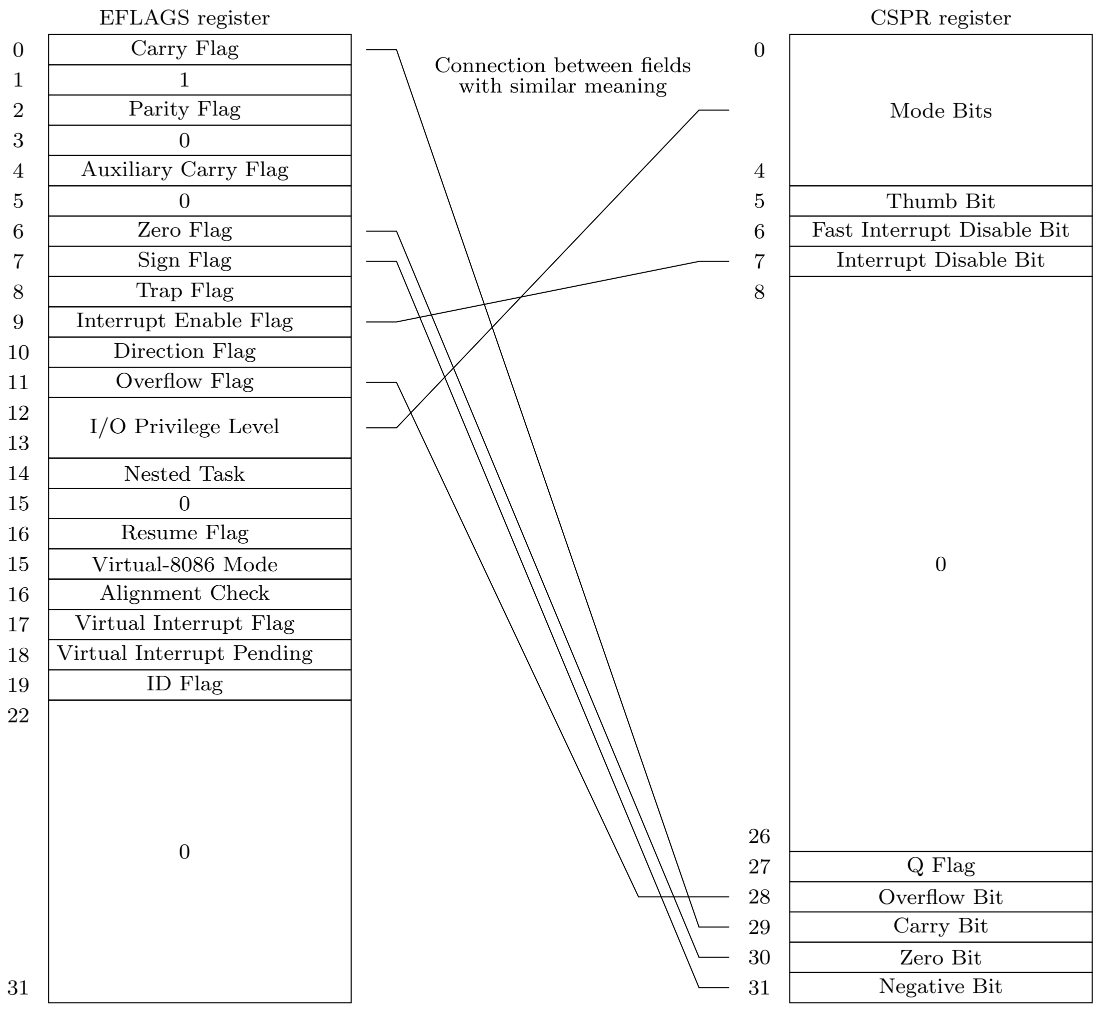


Figure 5.1: Comparison of the x86 `eflags` and the ARMv5 `cpsr` register

5.2 Paging on ARM-v5

Chapter B3.3 of the ARM Architecture Reference Manual DDI 0100E [ARM00] explains how the ARM memory management unit translates virtual to physical addresses. Similar to x86 paging the ARM-v5 memory management unit supports address translation using one or two levels of lookup tables. The ARM manual uses the terms first-level table and second-level table, but in order to compare the two architectures we will try to use the terms from Chapter 3, that is page directory and page table.

One level paging means having 1 MiB pages (called “sections” in the ARM manual), whereas two level paging means having 1 KiB “tiny” pages, 4 KiB “small” pages or 64 KiB “large” pages. We will only discuss paging with 1 MiB pages and 4 KiB pages to point out differences to the x86-32 architecture apart from the additional features.

The page directory contains 4096 entries of each four byte. Thus the page directory has a size of 16 KiB. The physical address of the page directory is stored in the translation table base register (TTBR). The last 14 bits of the physical address are required to be zero, that is the page directory has to be 16 KiB aligned. The first 18 bits allow to address all 16 KiB frames in the 32-bit physical address space.

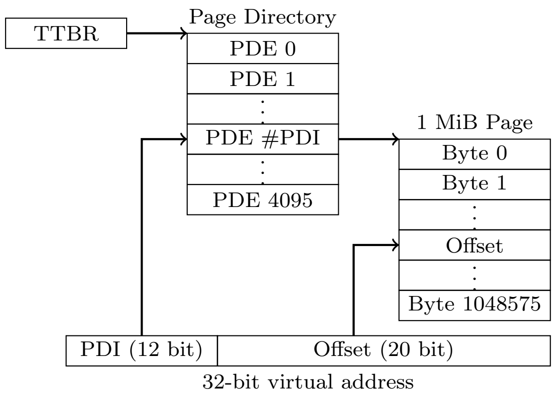


Figure 5.2: Virtual address resolution in 1 MiB paging mode according to page B3-8 of the ARM manual [ARM00]

The page directory maps the first 12 bits of the virtual address either to a 1 MiB physical page or to a page table, depending on the two type bits of the page directory entry. These first 12 bits are called the page directory index (PDI).

The page table is the second lookup table. It contains 256 entries of each 4 bytes, that is a page table has a size of 1 KiB. The page directory entry stores the 22-bit physical base address of the page table. 22 bits allow to address all 1 KiB frames in the 32-bit physical address space. Therefore, the page tables have to be 1 KiB aligned. The second 10 bits of the virtual address are used as the page table index (PTI).

When addressing a 1 MiB page the CPU takes the page directory physical page number from the TTBR0 register and the page directory index from the virtual address, to find the page directory entry to this virtual address. The page directory entry contains the 12-bit physical page number of the 1 MiB page. These 12 bits, combined with the lower 20 bits of the virtual address, form the physical address. The virtual address resolution for 1 MiB pages is also shown in Figure 5.2.

In 4 KiB paging mode the page directory contains the 22-bit physical base address of a page table instead of a 20-bit physical page base address for a 1 MiB page. The physical base address can be extended with zeros, because the page table is 1 KiB aligned. Using the 8-bit page table index we get the page table entry to the given virtual address. The page table entry contains the 20-bit physical page number of the 4 KiB page. Together with the lower 12 bits of the virtual address this forms the physical address. The virtual address resolution for 4 KiB pages is also shown in Figure 5.3.

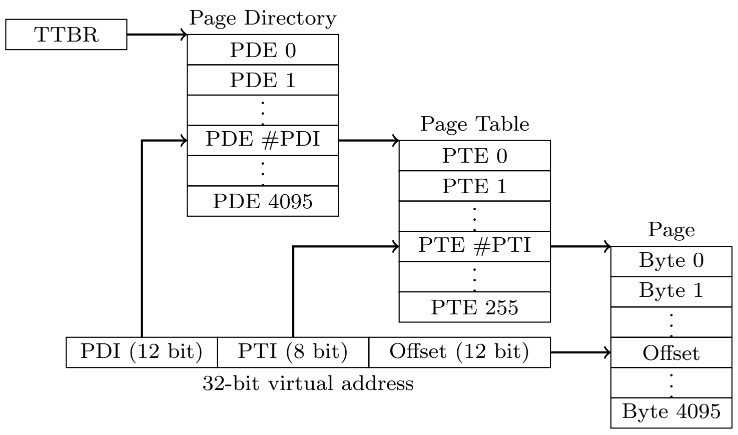


Figure 5.3: Virtual address resolution in 4 KiB paging mode according to page B3-14 of the ARM manual [ARM00]

Similar to the x86 page tables the ARM page tables contain bits for managing access permissions on the according virtual memory area. Page directory and page table entries have a 2-bit size field which specifies the page size. Size 0 means that this entry is not present, other values select different sizes. A permission field allows to specify permissions for privileged modes and user mode.

5.3 Interrupts

Similar to the interrupt descriptor table (IDT) of the x86 architecture the ARM-v5 architecture defines an interrupt vector table. The interrupt vector table is located at address `0x0` and has a format similar to the x86 real mode interrupt vector table[2], which is a much simpler format than the x86 IDT.

The basic interrupt vector table contains 8 entries: Reset exception, undefined instruction exception, software interrupt, prefetch abort exception, data abort exception, interrupt and fast interrupt. Depending on the CPU model and base board this can be extended by more interrupts. The base board often comes with an interrupt controller which enables devices to invoke certain interrupts from this table. Each entry of the interrupt vector table is a 4 byte instruction. This instruction usually is a jump instruction (`b`, `bl` or `blx`) or a manipulation of the program counter in order to execute the corresponding interrupt handler for the interrupt.

When an interrupt occurs the address of the next instruction (i.e. the return address) is stored in the link register (`lr`) of the new execution mode. The mode bits in the

[2]The x86 real mode interrupt vector table lies at address `0x0`, has 256 entries. Each entry is the 32-bit address of the corresponding interrupt handler.

cpsr are changed to the new execution mode. In the case of a reset exception or a fast interrupt the CPU disables fast interrupts in the cpsr register. Normal interrupts are disabled in the cpsr any case. The cpsr value is then stored in the saved flags register (spsr) of the new execution mode. The program counter is then set to the according address in the interrupt vector table.

To return from an interrupt handler, the old cpsr value has to be restored from the spsr register and the program counter set to the return address stored in the lr register.

5.4 Calling convention

Similar to the x86 call instruction the bl instruction takes only one parameter: the new instruction pointer. Therefore, we need a convention how to pass arguments to a function and how to get a functions return value. In this section we will describe the ARM calling convention. As before, we will only talk about passing integer variables as this is the only case we need for porting a simple operating system kernel from the x86 architecture to the ARM-v5 architecture.

Similar to the AMD64 ABI calling convention, the arguments are passed through the registers. The ARM calling convention uses the registers r0 to r3 for the first four integer arguments [ARM12]. The remaining integer arguments are passed to the called function on the stack.

Next, the bl instruction is executed to jump to the called function. The CPU stores the return address in the lr register and the address of the called function in the pc register. The function may then read the function arguments from the registers r0 to r3 and from the stack. The ARM calling convention defines the registers r4 to r11 to be used for local variables. Thus the called function has to preserve these registers or restore them before returning.

Returning from a function is typically done using the bx lr instruction. This instruction branches to the return address stored in the lr register. The gcc uses the register r11 as the frame pointer (fp) register. It basically provides the same functionality as the ebp on x86. A function generated by the gcc will typically push the registers pc, lr, sp and fp onto the stack. The fp register is then set to the address where the pc is stored on the stack. As a means of optimization the compiler may omit pushing all registers onto the stack.

The syscall convention used on x86 Linux or Sweb can be used in a slightly modified version on the ARM-v5 architecture. It uses six registers to pass six arguments to the syscall. We can use registers r0 to r3 as in the ARM calling convention to pass four arguments. If we want to pass another two arguments we can use registers r4 and r5, but the user space syscall wrapper function has to prepare the registers for that. Therefore, we have to store the current values of r4 and r5 on the stack and store the additional two arguments in these two registers. After the operating system handled the syscall, we have to restore registers `r4` and `r5`, before the user space syscall wrapper function returns.

We now discussed the most important differences between the x86-32 architecture and the ARM-v5 architecture from an operating-system developer's point of view. In the next chapter we describe how we ported the SWEB kernel to three ARM-v5 boards.

Chapter 6

Porting SWEB from x86 to ARM-v5

In the previous chapter we pointed out the differences between the x86 architecture and the ARM-v5 architecture. Now we will describe the process how the SWEB source code has been ported from x86 to ARM-v5.

Before that we want to compare the source code of the ARM-v5 port to the source code of the x86 version of SWEB. About 2350 lines of code were removed and 400 lines of code either added or modified. These 400 lines are divided into 60 lines of assembler code and 340 lines of C and C++ code. 400 lines of code are about 1% of the SWEB source code.

6.1 Compiling for ARM-v5

Building the SWEB kernel for the ARM-v5 architecture requires some changes in process of the compilation and linking, which we will describe in this section.

SWEB uses CMake for managing the build process. It was necessary to extend the CMake files slightly. We introduced a new `CMakeLists.compiler` file which is included at the beginning of the `CMakeLists.txt` in the root folder in order to force CMake to use the ARM cross compiler. The assembler files are not compiled using `nasm` as in the x86 version of SWEB, but using GNU `as` instead.

The x86 variant of the SWEB kernel uses no libraries except the ones integrated in its source code. On ARM, it was necessary to link the `libgcc` into the kernel binary because of the reduced instruction set of the ARM-v5 architecture. Some instructions the SWEB kernel uses are not implemented in hardware on the ARM-v5 architecture. For instance, the x86 architecture provides a 32-bit integer division instruction, whereas the ARM-v5 has no such instruction. In this case, the `libgcc` provides an implementation for a 32-bit integer division.

When linking with the `libgcc` the compiler may throw an error because the implementation for `raise` is missing. `raise` is a function used in the implementation of C++ exceptions. We can simply fix that by adding an endless loop or a function causing a CPU fault with that name, because there will be no C++ exceptions in the kernel.

If the kernel uses static or global objects, the function `__aeabi_atexit` might be needed as well, as the `libgcc` may use `__aeabi_atexit` for clean up on exit. In this case it matters even less than in the case of `raise` what that symbol `__aeabi_atexit` will point to. The kernel does not exit, thus `atexit` functions would not be called anyway. Implementing `__aeabi_atexit` as an endless loop might help finding accidental calls to that function.

It is much easier to debug an operating system kernel if you can be sure what a stack frame looks like. `gcc` optimizes function calls by omitting values from the stack frame which are not necessary for returning from the function. In order to force `gcc` not to alter the stack frame layout, we use the `-mapcs` compiler flag.

6.2 SWEB on the ARM Integrator/CP

On x86, SWEB boots using Grub. Grub loads the kernel binary into memory. The ARM-v5 port expects to be loaded into memory as well. This can be done using a boot loader. The first ARM port of SWEB runs on the ARM Integrator/CP board emulated with `qemu`. `qemu` allows loading the kernel binary directly into memory just as Grub would do.

The x86 version of SWEB uses a special debug port which allows printing debug messages on the host console. This debug port is not available in the ARM version of `qemu`. Therefore, we use one of the serial ports for that. This can also be used as a debug console when working on real hardware. A serial cable connects the ARM board to another computer. This other computer is then used to display the debug output of SWEB.

The ARM Integrator/CP board has a number of memory-mapped device registers. Figure 6.1 shows an overview of memory-mapped devices in the physical address space (see section 3.9.1 of the Integrator/CP manual [ARM02] and the `qemu info mtree` command). It is important to keep an eye on these addresses when activating paging. The kernel should still be able to access these mapped registers.

On other ARM boards the addresses and devices differ, but they basically follow the same structure: devices are mapped into physical address space and accessible through memory-mapped I/O.

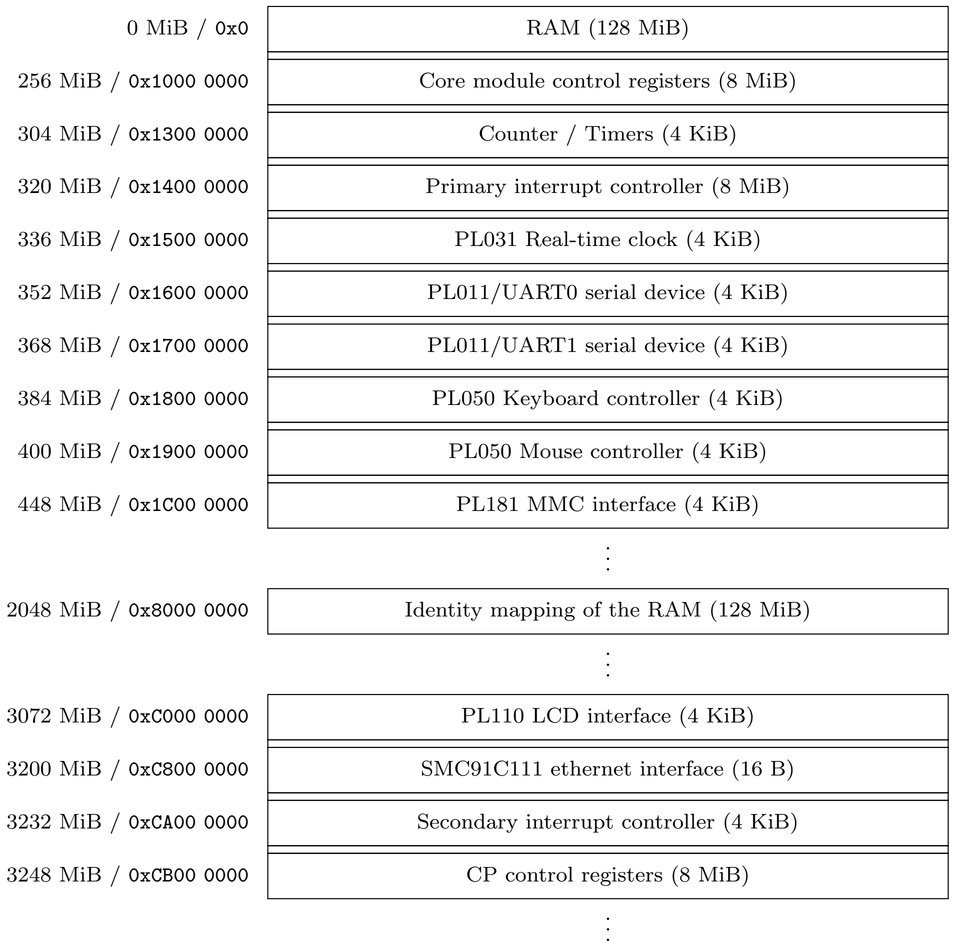

0 MiB / 0x0
RAM (128 MiB)
256 MiB / 0x1000 0000
Core module control registers (8 MiB)
304 MiB / 0x1300 0000
Counter / Timers (4 KiB)
320 MiB / 0x1400 0000
Primary interrupt controller (8 MiB)
336 MiB / 0x1500 0000
PL031 Real-time clock (4 KiB)
352 MiB / 0x1600 0000
PL011/UART0 serial device (4 KiB)
368 MiB / 0x1700 0000
PL011/UART1 serial device (4 KiB)
384 MiB / 0x1800 0000
PL050 Keyboard controller (4 KiB)
400 MiB / 0x1900 0000
PL050 Mouse controller (4 KiB)
448 MiB / 0x1C00 0000
PL181 MMC interface (4 KiB)
2048 MiB / 0x8000 0000
Identity mapping of the RAM (128 MiB)
3072 MiB / 0xC000 0000
PL110 LCD interface (4 KiB)
3200 MiB / 0xC800 0000
SMC91C111 ethernet interface (16 B)
3232 MiB / 0xCA00 0000
Secondary interrupt controller (4 KiB)
3248 MiB / 0xCB00 0000
CP control registers (8 MiB)

Figure 6.1: Memory-mapped devices in the physical address space of the ARM Integrator/CP board in `qemu`.

Upon booting, the program counter is set to the address of the `entry` symbol. This function is implemented in the `boot.s` file. The `entry` function first sets up the stack and calls the function `initialiseBootTimePaging` to initialise paging.

The `initialiseBootTimePaging` function initializes the page directory by creating an identity mapping using 1 MiB pages. We map the first 8 MiB of virtual address space to the first 8 MiB of physical address space. This mapping is used by the interrupt handlers and the frame buffer and therefore has to exist in user and kernel mode, although it is not accessible in user mode. The CPU switches to interrupt mode for interrupt handling and then jumps to one of the addresses in the interrupt vector table while the virtual address space is unchanged. We will explain this in the next section in detail.

The memory above `0xC000 0000` is identity mapped to the memory starting from `0x0`. For the kernel execution, we map the first 4 MiB of physical memory to `0x8000 0000`.

We mapped most of the devices shown in Figure 6.1 with an offset of `0x7000 0000`, resulting in virtual addresses after kernel end (2 GiB + 4 MiB). Thus the lower 2 GiB of address space can be used for user programs except for the first 8 MiB. The virtual address space layout for a typical SWEB binary on ARM is shown in Figure 6.2.

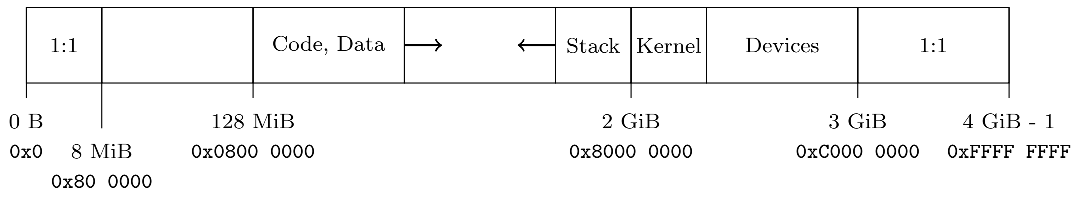


Figure 6.2: Virtual memory layout of SWEB on ARM for a typical SWEB binary

After the page directory is initialized we enable buffering and paging through the CP15 control registers. Finally, we call the function `PagingMode`. The jump moves execution to the higher half. In `PagingMode`, the stack is set to the new address above 2 GiB and the SWEB `startup()` C-function is called. The `startup()` function is implemented in the architecture independent part and therefore remains unchanged.

In the `ArchCommon` class we introduced a `createConsole` method when merging the SWEB x86 implementation with the SWEB Xen architecture implementation. We use this method to initialize the PL110 LCD device and setup a `FrameBufferConsole` using the frame buffer of that LCD device. This works surprisingly well, as the frame buffer can be configured to work with the same pixel format as the x86 VESA frame buffer. The CLCD device allows to configure where the frame buffer memory lies. We put the frame buffer memory at the end of the physical memory. Having a resolution of 640 × 480 pixels with 16 bits per pixel, about 600 KiB of physical memory are used as frame buffer memory.

The ARM-v5 architecture lacks an atomic add operation. Therefore, we implemented the `atomic_add` function in the ARM-v5 version of SWEB on a higher level, protected by a global spin lock. Instead of the x86 `xchg` instruction, we use the `swp` instruction which does basically the same. However, the `swp` instruction is deprecated in ARM-v6. ARM-v6 introduces a performance-enhanced locking mechanism (load exclusive, store exclusive) which should be used instead.

The `backtrace()` function using the `gstabs2` debug information works on ARM exactly the same way as on x86 after it has been adapted to the ARM calling convention. In order to add functions implemented in assembly language to the `gstabs2` symbol table, it is possible to annotate the function with the `.stabs` directive.

As the hardware configuration of the ARM Integrator/CP board differs significantly from common x86 hardware, we had to replace some drivers to provide the same functionality. The x86 version of SWEB uses an ATA driver to load data like user programs from a hard disk. The ARM port replaces the ATA driver by an MMC card driver. An SD card containing the same data as the hard disk is attached to the MMC slot. The MMC controller can be accessed through memory-mapped I/O. The implemented driver provides the same interface as the ATA driver and thus the existing IDE driver can use the MMC driver in the same way. Using SD cards with an ARM board is far more common than using an IDE controller attached to an ARM board. The SD card is initialized using the initialization sequence specified in the SD specifications [SD 13]. After the initialization the SD card stays in a mode where read and write operations are always possible.

User space binaries use the ELF32 format just as on x86. To load the ARM user space binaries no changes to ELF32 loader source code were necessary at all.

6.3 SWEB on the Gumstix Verdex

The Gumstix Verdex is a more recent ARM-v5 board developed by Gumstix. The Gumstix Verdex has an on-board flash memory for the boot loader and the kernel. For booting SWEB on the Gumstix Verdex, we use the "Das U-Boot" boot loader which is a very common boot loader on embedded platforms. The compiled kernel has to be prepared for this boot loader using the `mkimage` tool. This tool builds a file containing the kernel and meta data for the boot loader. In order to copy the boot loader and the kernel file to the onboard flash memory, they need to be written into an image file first. This image file can then be copied to real hardware or used to run SWEB in `qemu` emulating the Gumstix Verdex board.

The Gumstix Verdex provides different devices than the ARM Integrator/CP and organizes the physical address space differently. For instance, the physical RAM is located at `0xA000 0000` instead of `0x0`. Therefore, the kernel page directory needs to be constructed somewhat different in order to build a virtual address space equivalent to the virtual address space in the ARM Integrator/CP version of SWEB. The SWEB `PageManager` manages the physical RAM pages. As it starts counting at 0, we have to add the `0xA0000` offset before writing page numbers into page directory or page table structs [Int04]. Currently the Gumstix Verdex is the only architecture SWEB runs on where it is necessary to add this offset. An adaption of the `PageManager` to work with the real page numbers even if there is a physical RAM offset would probably help to make the code easier.

The Gumstix Verdex board has no keyboard controller. Common setups use the serial port for debug output and input. The serial port outgoing line is used for the debug console output. It can be connected to a computer or some other device to display the debug messages. Since SWEB accepts no input on the debug console, we redirected the input to the SWEB internal keyboard buffer. Thus, user programs will perceive inputs from the serial ports as keyboard inputs.

The MMC driver and the LCD initialization from the Integrator/CP had to be rewritten, since the Gumstix Verdex has a different MMC controller and LCD controller than the ARM Integrator/CP. In either case we use an SD card attached to the MMC controller. The communication with the SD card follows the SD specification. Therefore, SD card commands are issued in the same order on both controllers. The LCD controller receives the frame buffer data through Direct Memory Access (DMA). The DMA controller copies data in physical memory with minimal CPU interaction. For this purpose, the DMA controller uses DMA descriptors. A DMA descriptor defines what the DMA controller should do. Usually this is copying data from or to an address in physical memory. Each DMA descriptor contains a pointer to the next DMA descriptor, thus forming a linked list of jobs for the device. As soon as a device has finished one job, it jumps to the pointer of the next DMA descriptor and starts with this new job.

0	Physical address of the next frame descriptor
4	Physical address of the frame buffer
8	Frame ID field, unused by the hardware
12	Length of the frame buffer

Figure 6.3: DMA/frame descriptor for the Gumstix Verdex LCD controller

The structure of a DMA descriptor for the Gumstix Verdex LCD controller is shown in Figure 6.3. We want the LCD controller to continuously load the data from the same frame buffer, therefore we link the frame descriptor to itself.

6.4 SWEB on the Raspberry Pi

In the last two sections we developed SWEB ports that run on emulated ARM boards. Running SWEB on a real ARM board brings up new challenges. First of all does the emulated hardware not behave like real hardware in all situations. Second, timing plays a far more important role on real hardware. The Raspberry Pi is a cheap ARM platform, currently available for about $25. It comes with a ARM1176JZF-S processor which is backward compatible to the ARM-v5 instruction set. Furthermore, SWEB is an operating system for educational purposes. Being able to run SWEB on student-affordable real hardware might also enrich education using SWEB. Therefore, we ported SWEB to the Raspberry Pi.

Implementation for a real board is much easier if you can also emulate the board, like in case of the ARM Integrator/CP and the Gumstix Verdex. `qemu` has no support for the Raspberry Pi board, but there is a `qemu` fork [Est14] that emulates a simplified variant of the Raspberry Pi. However, some devices like the emulated USB controller are not compatible with the corresponding ones on the real Raspberry Pi. The Raspberry Pi port of SWEB was implemented after applying the changes from Chapter 7. Therefore this platform is not listed in the evaluation and comparison in Chapter 7 and it had no influence on the developed enhancements. Instead, the Raspberry Pi port of SWEB benefits from the measures taken, such as the architecture tree.

The Raspberry Pi comes with a number of hardware devices. SWEB only uses a few of these devices: the MMC controller, the interrupt controller, the timer controller and the USB controller [Bro12]. Additionally, it communicates with the GPU for graphics display.

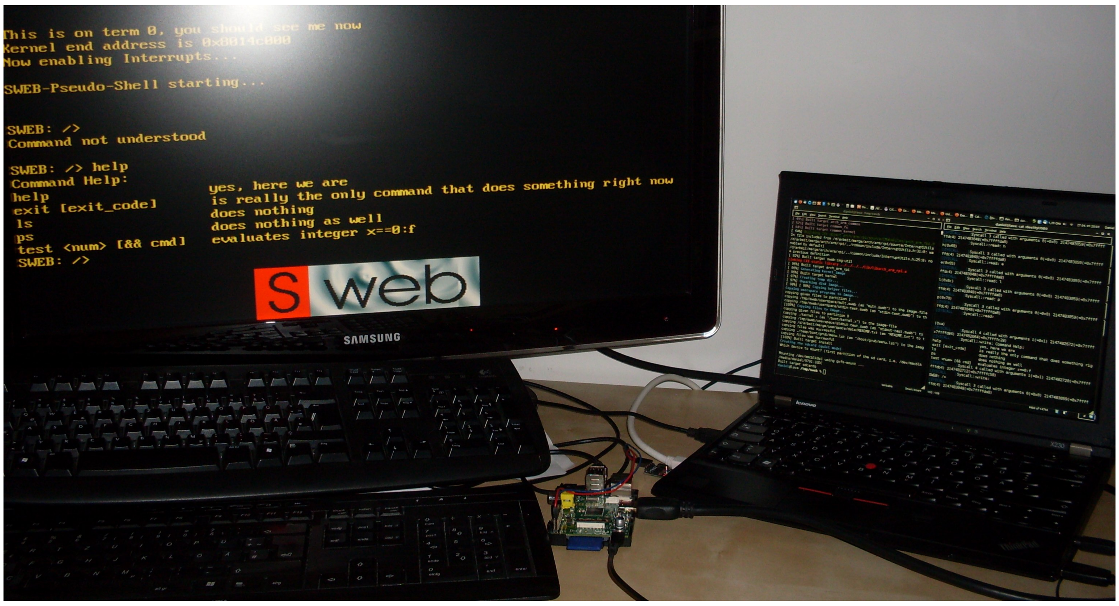
Similar to the ARM Integrator C/P and the Gumstix Verdex, the Raspberry Pi has no ATA controller. Instead, the typical setup uses an SD card containing the kernel as well as operating system and user files. The first partition of the SD card is expected to be formatted as FAT32. The execution begins on the GPU which will load a first-stage boot loader from this partition. The first-stage boot loader powers up the ARM CPU and starts the execution of the second-stage boot loader on the ARM CPU. The second-stage boot loader will then set up the hardware according to the configuration file provided on the FAT32 partition. For instance, some physical memory is reserved for the video adapter. Still, this physical memory is mapped in the physical address space. Finally, it will load and start the execution of the kernel image provided on the FAT32 partition. The x86 variant of SWEB has a first partition in a GRUB-readable format and contains GRUB and the kernel image. On all platforms, SWEB expects the user programs on the second partition.

After building the SWEB kernel, it is necessary to produce a stripped binary file using `objcopy`. This binary file does not contain the ELF headers and is loaded directly into memory. The Raspberry Pi port comes with a script that mounts the partitions of the SD card automatically and copies all files to it. Using the SD card from within SWEB works using the MMC controller. There are a only a few small differences to the MMC controller of the ARM Integrator/CP.

The Raspberry Pi has an on-board HDMI interface and a composite video port. The Raspberry Pi GPU sends output to one of them, depending on the configuration and the attached video device. The GPU has to be initialized before the operating system is able to use the frame buffer. The initialization involves so-called mailboxes. Mailboxes provide a way for different devices to communicate with each other. In this case, we want the CPU to send a message to the GPU which sets the GPU to frame buffer mode. The GPU will then acquire memory for the frame buffer in the physical memory reserved for the video adapter. This buffer along with more information on the external device configuration is sent back to the CPU using another mailbox message. The operating system receives that message and reads the frame buffer address returned by the GPU. This frame buffer can from then on be used as usual. The GPU continuously updates the screen with the data from the frame buffer.

The Raspberry Pi does not have a keyboard controller. Instead you can attach a USB keyboard. We integrated the CSUD [Cha14] USB stack, which is a very small USB stack implementation. It has about 5500 lines of code, which is far more than the complete architecture dependent source code for the Raspberry Pi. It comes with a simple USB keyboard driver and was relatively easy to integrate into SWEB. While integrating the CSUD USB stack in SWEB we submitted patches which provide a more tolerant keyboard detection and easier integration into C++ projects. The CSUD USB stack implements no interrupt mechanisms. Therefore, the USB keyboard is used in polling mode currently. The keyboard state is checked by the `IdleThread`.

As the ARM1176 behavior is undefined in some cases of unaligned memory accesses, the linker script was adapted to load all sections to aligned offsets. For the same reason, the `KernelMemoryManager` was modified to return only 16 byte aligned addresses. The linker script also loads the symbol table into physical memory, just as it does in the x86 version fo SWEB. Therefore, debugging using the `backtrace()` function works on the Raspberry Pi just as well as in the emulator. The stack traces are printed to the debug console running on a serial port as on all ARM versions of SWEB. The serial port can be connected to a computer and the received debug console data can be displayed to the user easily. This setup is also shown in Figure 6.4.


```
This is on term 0, you should see me now
Kernel end address is 0x8014c000
Now enabling Interrupts...

SWEB-Pseudo-Shell starting...


SWEB: />
Command not understood

SWEB: /> help
Command Help:
help                      yes, here we are
exit [exit_code]          is really the only command that does something right now
ls                        does nothing
ps                        does nothing as well
test <num> [&& cmd]       evaluates integer x==0:f
SWEB: />
```

Figure 6.4: Setup for testing SWEB on the Raspberry Pi: a Raspberry Pi running SWEB can be seen in the middle of the picture. The left screen is attached to the Raspberry Pi HDMI port. It shows the SWEB console after some user input. The upper keyboard is attached to one of the Raspberry Pi USB ports and is used for user input. The blue card in the Raspberry Pi is the SD card containing SWEB. The lower keyboard is attached to the laptop on the right. The laptop screen shows two windows, the build window on the left and the debug console with debug output received from the Raspberry Pi on the right. A USB serial adapter is connected to the laptop on the left. The red and a blue wire connect it to the Raspberry Pi.

6.5 Interrupt handling and context switching

We discussed porting SWEB to three specific ARM boards and differences between these three ports. In this section we describe how interrupt handling and context switching works in SWEB on all ARM architectures.

The initialization of the interrupt vector table is implemented in the `ArchInterrupts` class. We initialize the primary interrupt controller, the timer and the keyboard. The timer will produce interrupts in a regular interval, just like the programmable interval timer (IRQ0) on x86. It is used for preemptive scheduling. The keyboard controller works very similar to the keyboard controller on x86. Unlike the x86 architecture all devices are accessed through memory-mapped I/O.

When the CPU starts the execution of an interrupt handler, it works in the corresponding exception mode. All exception modes have their own registers `r13` and `r14`. In SWEB all interrupt modes use the same stack. This stack, however, is used for context switching only. This way, context switching shares more similarities between ARM and x86 and nested interrupts can be handled in the same way. The first thing each interrupt handler does is store all registers in the `currentThreadInfo` struct and switch from the exception mode to the system task mode. In this mode, we may enable interrupts again and therefore allow nested interrupts.

Listing 6.1 shows the interrupt handler entry in the ARM port of SWEB. First, registers `r0` to `r12` are stored to the stack because they may be changed by the following operations. We don not use the stack except for these 13 integer values and switch to a different stack a few instructions later anyway. Therefore, and in order to keep the stack pointer at the same address for the next interrupt, we reset it immediately. At this point, the execution is still in interrupt mode; that is registers `lr` (`r14`) and `sp` (`r13`) are the interrupt mode registers. The code in Listing 6.1 then stores the `lr` register and the `spsr` register in the `currentThreadInfo` struct. The `currentThreadInfo` struct holds all register values of a thread while it is not running. Afterwards, we switch to system task mode with interrupts disabled (`0xdf`) by changing the `cpsr` register. In system task mode, the `sp` and `lr` registers are shared with user space mode. Therefore, we can now store these registers in the struct.

The interrupt handler entry checks whether this interrupt came from user space. We do not want to execute the interrupt handler on a user space stack. Therefore, we replace it by the `sp0` value from the `currentThreadInfo` struct. The `sp0` field does not represent a real register, but the address of the kernel thread stack of this user thread.

In SWEB, each user thread contains a full kernel thread as well. This way, syscalls can be executed by the kernel thread belonging to the user thread. This mechanism is identical to the one on x86 SWEB.

Now we work on the kernel stack, which is used by the running thread only. Therefore, we may call functions and work on the stack as usual. The function `storeRegisters()`

now stores the variables from the other stack into the `currentThreadInfo` struct. Thus, all registers are stored and can be restored when the thread is scheduled the next time.

The ARM port of SWEB knows two different software interrupts: one for syscalls and one for yielding. The `yield` software interrupt tells the `Scheduler` to schedule a new thread. The `Scheduler` will then replace the `currentThread` and `currentThreadInfo` pointers. When the CPU returns from the interrupt, the registers are restored from the `currentThreadInfo` struct. If it has been changed during this interrupt, the new `currentThread` is scheduled. Before returning from the interrupt handler, the translation table base register 0 (`ttbr0`) is updated in the same way as the `cr3` register is updated on x86.

```
asm("push {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12}");\
asm("add sp, sp, #0x34");\
asm("mov %[v], lr" : [v]"=r" (currentThreadInfo->pc));\
asm("mrs r0, spsr"); \
asm("mov %[v], r0" : [v]"=r" (currentThreadInfo->cpsr));\
asm("mrs r0, cpsr \n\
     bic r0, r0, #0xdf \n\
     orr r0, r0, #0xdf \n\
     msr cpsr, r0 \n\
    ");\
asm("mov %[v], sp" : [v]"=r" (currentThreadInfo->sp));\
asm("mov %[v], lr" : [v]"=r" (currentThreadInfo->lr));\
if (currentThreadInfo->sp < 0x80000000) { asm("mov sp, %[v]"
    : : [v]"r" (currentThreadInfo->sp0)); }\
storeRegisters();
```

Listing 6.1: Context switching: Interrupt handler entry on ARM

When switching modes, the ARM processor accesses different `sp`, `lr` and `spsr` registers, except for system task and user mode. Therefore, they have to be restored before returning from the interrupt.

Listing 6.2 shows the interrupt handler exit procedure in the ARM ports of SWEB. The execution begins in system task mode, that is, `lr` and `sp` for the user thread can be restored first. In order to return from the interrupt, we switch from system task mode to service mode, because we need to be in an interrupt mode to execute an interrupt return. Only interrupt modes have an `spsr` register, which is used when executing an interrupt return. When the CPU executes the interrupt return, the `cpsr` register is changed implicitly to the old `cpsr` value restored from the `spsr` register. We set the interrupt mode `spsr` register to the value stored in the `cpsr` field and the interrupt mode `lr` register to the `pc` register from the struct. Then we push registers `r12` to `r0` from the struct on the stack and restore them using a single `pop` instruction. Finally, the `movs pc, lr` instruction causes the interrupt return.

The ARM-v5 architecture defines two different kinds of page faults: page faults on instruction fetches and page faults on data accesses. Both cases are implemented in the `void pageFaultHandler(uint32 address, uint32 type)` method. The `type` defines which of the two types a page fault is. In case of a page fault caused by an instruction fetch, the instruction pointer causing this page fault can be retrieved from the stored `lr` register. In case of a data access page fault, the address causing the page fault can be retrieved from the fault address register. The ELF32 binary loader is almost unchanged from x86, as the binary format is the same. Some constants were replaced in order to allow execution of ARM binaries and forbid the execution of binaries for other architectures.

```
asm("mov lr, %[v]" : : [v]"r" (currentThreadInfo->lr));\
asm("mov sp, %[v]" : : [v]"r" (currentThreadInfo->sp));\
asm("mrs r0, cpsr \n\
     bic r0, r0, #0xdf \n\
     orr r0, r0, #0xd3 \n\
     msr cpsr, r0 \n\
    ");\
asm("mov r0, %[v]" : : [v]"r" (currentThreadInfo->cpsr));\
asm("msr spsr, r0"); \
asm("mov lr, %[v]" : : [v]"r" (currentThreadInfo->pc));\
asm("mov r3, %[v]" : : [v]"r" (currentThreadInfo->r12));\
asm("push {r3}");\
// [... identically for registers r11 to r1 ...]
asm("mov r3, %[v]" : : [v]"r" (currentThreadInfo->r0));\
asm("push {r3}");\
asm("pop {r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12}");\
asm("movs pc, lr")
```

Listing 6.2: Context switching: Interrupt handler exit on ARM

6.6 User processes

SWEB provides each user process with its own address space. This is implemented by giving each user process its own page directory. Upon context switch, the `ttbr0` register is updated to the address of the page directory of the new running process. A page directory uses 16 KiB of space. Therefore, the `PageManager` was extended to allow allocating 4 physically subsequent pages. In a new user process page directory, all entries between 8 MiB and 2 GiB are marked as not present. The remaining entries are copied from the kernel page directory. When loading the binary, the kernel maps a 4 KiB stack page for the user process below 2 GiB.

Although page tables contain only 256 entries and thus need only 1 KiB of space, a 4 KiB page is allocated for each page table currently. To reduce the waste of memory, it would be possible to completely switch the ARM architecture implementation, or at least the `PageManager`, to 1 KiB paging. A different solution would be to create a page table pool where page table space can be acquired. Page tables are then grouped together to fill a 4 KiB page.

When starting a user thread, the thread info structs have to be initialized to the initial register values of the new user thread. The `pc` register is set to the address of the entry point of the program. The `cpsr` register is set to user mode. The stack pointer (`sp`) and the frame pointer (`fp`) are set to the user stack. The `sp0` is set to the kernel thread stack. Some ARM processors will generate a CPU fault if the stack pointers are not 8-byte aligned, thus it is recommended to align stack pointers to 8 byte.

With user processes running in SWEB on ARM, we finished the ARM port of SWEB. In the next chapter we will examine how we enhanced multi-platform compatibility while we ported SWEB.

Chapter 7

Increasing the multi-platform compatibility of SWEB

As a part of this master thesis we developed the x86-PAE port, the x86-64 port, and the ARM-v5 port of SWEB, as well as a Xen port and an ARM-v7 port. With the Xen variant the two folder structure of SWEB was established, with one folder for architecture dependent and one folder for architecture independent code. However, as the Xen variant had only a few users, the code diverged over the years and finally in 2011 the Xen folder was removed from the master branch and moved into a separate branch. Over the next years the code diverged further.

In this chapter we describe how we merged the different architecture branches of SWEB. The initial situation was having each platform implemented in separate branches. We merged all branches, except for the ARM-v7 branch. Although merging the build system of the ARM-v7 branch would not take much effort, it would take a lot to merge the architecture independent folders of the two branches. Just like the Linux kernel, the SWEB kernel follows the design principle that architecture independent code may be accessed from anywhere. However, architecture dependent code must be abstracted behind the existing interface to the architecture dependent code. In the ARM-v7 branch in many architecture independent files C++ has been replaced with C upon porting to the new architecture. Thus, the interface of the architecture independent code changed in many files. The architecture dependent code again relies on that interface. Therefore, we decided not to merge the ARM-v7 port.

We merged the branches in order of difference in the architecture independent code in order to minimize merging efforts. That is, we first merged x86 and x86-PAE. The merge of x86 and x86-PAE introduced the abstraction of the memory model which we describe in detail in Section 7.1. The two platforms are virtually the same, but differ in the way paging is done. If we take a look at the Linux kernel, we see that in the Linux kernel x86-32, x86-PAE and x86-64 are implemented as one architecture with `#ifdef` constructs to provide different source code for the different architectures and files with different postfixes for 32-bit respectively 64-bit implementations. The Minix kernel only implements the x86-32 architecture, neither x86-PAE nor x86-64.

The next branch we merged was the Xen branch. Although it diverged for several years and the old code was not compiling at all, the interface between architecture dependent and independent code was mostly unchanged. This merge lead to a rudimentary hardware abstraction as described in Section 7.2 and abstraction in the build system as described in Section 7.3.

Finally, we merged the x86-64 branch, which was a lot easier, because of the changes introduced by the previous merges. The most significant change the x86-64 merge brought to the architecture independent code was the data type usage. As all data types were size explicit until now (for instance `uint32`) this lead to small changes in many files. The changes are described in detail in Section 7.4.

After the merge we started developing the ARM-v5 port. Almost no adaption in the architecture independent code was necessary. Only about 300 lines of code had to be added and another 300 lines of code were modified, both in the architecture dependent code. This shows that the existing and newly introduced architecture abstraction eases porting SWEB to new architectures. The ARM-v5 port still lacks some features x86 SWEB provides, but most hardware dependent parts are already implemented.

It is important to note that there are big differences between different base boards using the same ARM processor. The ARM-v5 port initially was written for the ARM Integrator/CP. Support for Gumstix Verdex Boards and the Raspberry Pi was added subsequently. We introduced a new folder tree structure for the different architectures in order to reduce code replicated in several architecture folders.

7.1 Memory model abstraction

In the old SWEB the `Loader` class contained a member variable `page_directory_page_` storing the physical page number of the page directory page. In Section 3.1, 3.2 and 3.3 we described the differences between paging modes on the x86 architecture. For instance we saw that in PAE paging mode there is the page directory pointer table. The page directory pointer table in PAE paging mode does not only differ from the page directory in x86 paging mode in the fact that it is the third level of the paging mechanism, but furthermore, it is only a table containing 4 pointers. It would be wasteful to use a whole page for these 4 pointers. This shows that we need an abstraction of the memory model for the different architectures.

We found that the `ArchMemory` class can be used for this abstraction with only a few source code changes. In the old SWEB the `ArchMemory` class provided several static methods to modify paging data structures. Therefore, the methods took the page number of the page directory page as the first parameter. This already suggests to redesign this class as an object, which is what we did. Thus the `Loader` holds an `ArchMemory` object and does not need to know how the paging data structures work or even if they exist at all. We saw this design idea already in the Linux kernel in Section 2.2, where one of the basic assumptions was that the platform has some address translation mechanism and that it works similarly to the Intel 386 architecture.

As an additional measure the method signatures of the `ArchMemory` class were given simpler and more general names[1]. Furthermore, the C++ constructor and destructor features provide a less error-prone way to initialize and clean up the address space. In the old SWEB the address space of a user process was initialized by calling the `initNewPageDirectory` method. Now this is done by the default constructor of the `ArchMemory` object, so it is much harder to not initialize the address space. When it comes to process termination, the user had to call a `cleanupUserSpaceAddressSpace` method to clean up the paging data structures.

In the old implementation, if the programmer forgot about the cleanup, the previously used paging data structures remain in physical memory and this would be a physical memory leak. Now the cleanup is done by the destructor, making it virtually impossible to forget about the cleanup and leak physical memory this way. Listing 7.1 and 7.2 compare the interface-related method calls in the old and the new `ArchMemory` implementation during the life time of a user process. The `initNewPageDirectory` method call is removed as well as the call to the `cleanupUserspaceAddressSpace` method.

```
User process creation:
new UserProcess(path, fs_working_dir, this);
  ↳ loader_ = new Loader(fd_, this);
    loader_->loadExecutableAndInitProcess();
      ↳ initUserspaceAddressSpace();
          ↳ ArchMemory::initNewPageDirectory(page_dir_page_);
            ArchMemory::mapPage(page_dir_page_, 1024*512-1,
                                stack_page, 1);

User process destruction:
delete destroy_list[i];
  ↳ loader_->cleanupUserspaceAddressSpace();
      ↳ ArchMemory::freePageDirectory(page_dir_page_);
    delete loader_;
```

Listing 7.1: Old `ArchMemory` implementation: interface-related method calls during a user process life time

[1] For instance `setPageDirectory` was renamed to `setAddressSpace`.

```
User process creation:
new UserProcess(path, fs_working_dir, this);
  ↳ loader_ = new Loader(fd_, this); // ArchMemory is member
    loader_->loadExecutableAndInitProcess();
      ↳ initUserspaceAddressSpace();
          ↳ arch_memory_.mapPage(1024*512-1, stack_page, 1);

User process destruction:
delete destroy_list[i];
  ↳ delete loader_; // deconstructs ArchMemory member
```
Listing 7.2: New `ArchMemory` implementation: interface-related method calls during a user process life time

In Section 4.7.1 we described how a 48-bit virtual address space simplifies the stack-placement design decision. The basic SWEB has no dynamic stack allocation. Listing 7.3 shows the user process stack allocation in basic SWEB. A physical page is allocated from the `PageManager` and the physical page number is then mapped to virtual page number $1024 \cdot 512 - 1$, that is the page starting at virtual address 2 GiB − 4 KiB.

```
size_t page_for_stack = PageManager::instance()->
    getFreePhysicalPage();
arch_memory_.mapPage(1024*512-1, page_for_stack, 1);
```
Listing 7.3: User process stack allocation

Students are expected to implement stack allocation for user threads in the Operating Systems exercise class at Graz University of Technology. An architecture-independent multi threading implementation requires additional effort. In case of supporting only one architecture, the new `ArchMemory` implementation is always less code to write and less error-prone. In case of supporting several architectures it gives only small advantages over the old `ArchMemory` implementation.

The new `ArchMemory` class allows to abstract the stack allocation and stack mapping behind the `ArchMemory` class interface. The `ArchMemory` object is specific to one user process. Therefore, it is possible to store additional information in the `ArchMemory` object, for instance the data structures for the stack slot management. For example, a student could implement a method `allocateStack()` in the `ArchMemory` class, which allocates a physical stack page and a stack position in the user process address space. The physical stack page would then be mapped into the user process address space. An implementation of the `allocateStack()` method would be required for each supported architecture.

A similar method could also be implemented in the old `ArchMemory` class, but would require passing more arguments and the separation between architecture dependent and architecture independent code would be less clear, as the old `ArchMemory` class cannot contain any data structures.

7.2 Hardware abstraction

When merging the Xen branch, one of the conflicts was the console. Xen provides a console interface as hypervisor calls (`Xenprintf`, etc.), whereas the x86 architecture works with a frame buffer which can be used as a frame buffer based console or a text based console. The Cortex-M4 board on which SWEB runs has neither of this, the console on the ARM port of SWEB uses one of the serial port for this purpose.

Until now the console object had been created in the `main.cpp` in the architecture independent code. In an old version of the code with x86 and Xen in one branch an `#ifdef` at this point decided which console object to create, depending on the architecture. Because SWEB is an operating system used for education, one of its design principles is to avoid using `#ifdef` constructs to produce different code for the different architectures. Therefore, we abstracted this and defined a new `createConsole` method each architecture has to implement.

When porting SWEB to ARM-v5 we found several x86-specific statements in the code, such as an inline assembler execution of the `hlt` statement. We abstracted these assembler instructions in the common code behind new generic methods of the architecture interface.

7.3 Build system

SWEB uses the CMake build system. The basic structure in SWEB is that every directory contains a file `CMakeLists.txt` defining what to compile in this directory and how to compile it. The CMake file in the root directory of the repository defines some basic rules which apply to all sub folders so that the CMake files in the sub folders are much shorter.

The build process for each architecture differs in the used programs, the compiler flags and more. In order to provide a simple way to build the SWEB kernel differently for each architecture, we introduced two new files which have to be provided in each architecture folder: `CMakeLists.include`, which is included by the root CMake file, and `CMakeLists.userspace`, which is included by the CMake files in the user space folders. This way we are able to specify architecture dependent flags and targets for both, the kernel binary and the user space programs.

A third file `CMakeLists.compiler` had to be introduced in order to facilitate cross compilation in the existing framework more conveniently. The ARM-v5 port uses this file to force CMake to use a specific compiler.

7.4 Data type usage in an operating system kernel

Using platform independent data types is necessary as soon as the kernel has to work with architectures with different pointer sizes. Throughout the SWEB source codes a large number of casts between pointers or indices and size explicit data types like `uint32` can be found. First of all we replaced many occurrences of `uint32` by `size_t` and `int32` by `ssize_t`. `size_t` is an unsigned integer variable defined to be as many bits wide as are necessary to index the whole address space. `ssize_t` is a signed integer variable of the same width as `size_t`.

Anyway, switching to existing platform independent data types does not always work. On 32-bit SWEB we have an ELF32 binary loader. It requires many constants and some parsing methods which are format specific. We split the ELF binary loader into a part which is ELF32 specific and into a part which forms a generic ELF binary loader. The ELF32 specific part is placed in a new `ElfFormat.h` header file inside the architecture dependent folder. The 64-bit variant of SWEB implements this header file differently, providing support to load ELF64 binaries. This solution is similar to the previously introduced `ArchMemory` object.

7.5 Multi-platform compatibility of the new SWEB kernel

In this section we want to analyze how the merged SWEB kernel supports multi-platform compatibility by design. The merged SWEB kernel did not change in its general structure. That is, the architecture dependent code is still in the `arch` folder and the architecture independent code in the `common` folder.

One factor is the amount of code redundancy. Code redundancy interferes with multi-platform compatibility as it requires to apply changes to the redundant code manually. We analyzed the folders of the five architectures and built the code redundancy matrix shown in Figure 7.1.

	x86	x86-PAE	X86-64	Xen	ARM-v5	lines of code
x86		96%	62%	32%	53%	5427
x86-PAE	95%		63%	30%	52%	5481
x86-64	70%	71%		21%	47%	4851
Xen	28%	27%	17%		24%	6174
ARM-v5	87%	86%	69%	44%		3285

Figure 7.1: Architecture code redundancy matrix (before grouping common code)

As you can see x86 and x86-PAE share about 95% of their source code. Even x86 and x86-64 share about 70% of their source code. In the next section we build a structure which allows different architectures to group their common code together. For example driver code could be grouped together in a `driver` folder as in Linux. Furthermore, architecture family folders could provide a source folder structure which allows having code all architectures of a family share in one place only.

Xen and x86 have less than 30% of their source code in common. The architectures that differ the most with respect to their common source code are ARM and Xen. Only 16% of the lines of code in the Xen implementation exist identically in the ARM implementation. Surprisingly the ARM-v5 source code shares the greatest part of its source code with other architectures. This is due to the fact that the the ARM-v5 source code has about 50% fewer lines of code than the other architectures. Therefore the fraction of common source code is higher, although it has fewer lines in common in absolute numbers.

In the case of the different ARM base boards the Linux kernel solves this through sub folders. All base boards have the same processor and therefore share the architecture specific code for this processor. The differences between the base boards are implemented in the sub folders. In SWEB we have a similar folder structure but on a higher level. We have a folder for common architecture dependent code and code which is specific to one architecture only. To demonstrate the potential in moving source code that exists in multiple architecture folders to a common folder, we exemplary moved some source and header files which were very similar or identical in all architecture implementations into a common folder. The resulting difference matrix can be seen in Figure 7.2. All architecture implementations now have significantly less lines of code. For instance the ARM-v5 implementation has 2134 lines of code, which is only about 5 percent of the total 40000 lines of code.

	x86	x86-PAE	X86-64	Xen	ARM-v5	lines of code
x86		96%	66%	26%	27%	4775
x86-PAE	95%		66%	24%	25%	4836
x86-64	70%	71%		21%	25%	4552
Xen	22%	20%	17%		13%	5607
ARM-v5	60%	58%	53%	33%		2134

Figure 7.2: Architecture code redundancy matrix (after grouping common code)

Another important measure is to see how many lines were modified or added. This is shown in Figure 7.3. The ARM-v5 architecture was added with a very small amount of lines of code. This indicates that SWEB can easily be ported to architectures significantly different from the x86 architecture family.

One design principle of SWEB is to avoid platform specific use of `#ifdef`. The merged SWEB contains no such `#ifdef` constructs at all.

	lines of code modified	lines of code added	lines of code
x86-PAE	176	85	4836
x86-64	474	906	4552
Xen	113	4265	5607
ARM-v5	414	445	2134

Figure 7.3: Architecture code compared to the x86 base branch: modified and added lines of code

7.6 Reduction of code replication

We have seen that the x86 and x86-PAE architecture implementations each have about 4800 lines of code. The difference between the two implementations is only about 250 lines of code. Thus, the remaining 4550 lines of code still are identical in both implementations. We enhanced the folder structure introduced in the previous section and implemented a recursive architecture tree in SWEB in order to reduce code replication even further. The root of the tree is the `arch` folder which contains all architecture dependent code. Each node of the tree now has sub folders for further architecture specialization and a sub folder `common` for code shared by all derived architectures. If the node is a leaf it has no sub folder `common`. Instead this code is placed in the folder of the leaf node itself. This makes the tree more shallow and thus simpler. Figure 7.4 shows the architecture tree in SWEB.

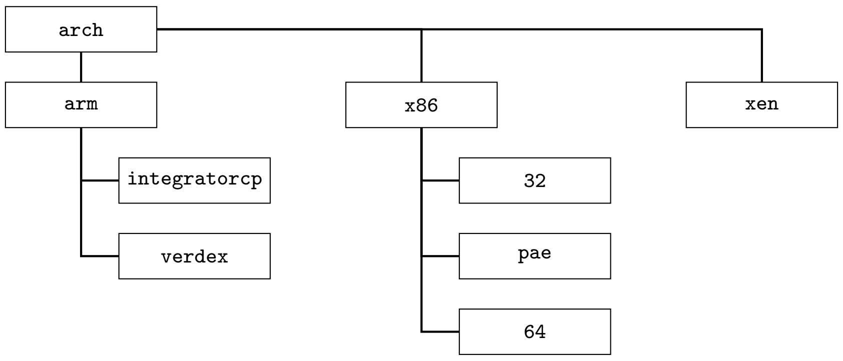


Figure 7.4: Architecture tree of the SWEB kernel

In the x86 architecture family we identified 11 common header files and 12 common source files. These files belong to drivers which do not differ between x86-32, x86-PAE and x86-64, for instance the keyboard driver, the IDE driver, the ATA driver and the serial port driver. This is similar for the ARM architecture family, although more drivers are implemented for a device on one specific base board. Figure 7.5 shows the reduction in lines of code for the different x86 architecture implementations, resulting from the architecture tree. Figure 7.6 shows how much code the architecture families have in common and Figure 7.7 shows how much code the architecture implementations (the second level of the tree structure) have in common. Overall the architecture tree allowed for the removal of 13000 previously replicated lines of code, which is about 22% of the total source code.

	lines of code difference	lines of code
x86-32	-38%	2954
x86-PAE	-38%	3015
x86-64	-41%	2693
x86 (common)		1831

Figure 7.5: Lines of code reduction for x86 architectures after architecture tree implementation

	x86	Xen	arm	lines of code
x86		7%	5%	9815
Xen	12%		9%	5607
arm	17%	17%		3118

Figure 7.6: Architecture family code redundancy matrix

	x86-32	x86-PAE	x86-64	ARM I/CP	ARM Verdex	LoC
x86-32		93%	47%	9%	9%	2954
x86-PAE	91%		47%	9%	9%	3015
x86-64	51%	53%		9%	9%	2693
ARM I/CP	26%	25%	23%		86%	1092
ARM Verdex	27%	26%	24%	90%		1048

Figure 7.7: Architecture code redundancy matrix after architecture tree implementation

The tree structure is easy to understand, but it is not sufficient to prevent future code replication, especially if SWEB gets ported to more architectures or different base boards. The folder tree structure follows the assumption that a driver belongs to exactly one architecture branch. This might be true if you only have architecture branches which have very little in common, but it is certainly not true in the case of closely related architecture branches. If two architectures have the same hardware device but the parent architecture or the architecture family does not, it will be necessary to place the driver source code in both directories.

Instead of replicating the driver source code we could copy another idea from the Linux kernel structure and move all driver code in SWEB to a `driver` folder. This way we separate drivers from processor architectures. Hence, the driver has to be implemented independently from the processor architecture.

Chapter 8

Conclusion

In this thesis we examined how different kernels are designed regarding multi-platform compatibility. We found and identified good design principles in the Linux kernel. Moreover, we described in detail how the SWEB kernel was ported to the x86-64 architecture, the ARM-v5 architecture and how it has been made multi-platform compatible.

With this intention in mind, we compared the x86-32 architecture and the x86-64 architecture in detail. We saw how paging was extended by a third level in PAE and by a fourth level in x86-64 paging mode. Furthermore, we described changes regarding interrupt handling, the calling convention and the small differences in the ELF binary format.

With this knowledge we were able to port SWEB from x86-32 to x86-64. This was achieved by implementing PAE paging first and then implementing x86-64 paging.

The most time consuming part of this work has been debugging the operating system ports on the different platforms. That of course does not show up in this document proportionately, but we tried to give the reader a good overview what steps are necessary to avoid time consuming kernel debugging.

We compared the x86-32 architecture and the ARM-v5 architecture and described how paging works on the ARM-v5 architecture. We described how interrupts work on ARM boards. We compared the ARM EABI calling convention to the AMD64 ABI calling convention.

With this knowledge we were able to port SWEB from x86-32 to the ARM-v5 on an ARM Integrator C/P board. The ARM-v5 port uses the memory management unit and interrupts for the kernel, a MMC driver instead of an ATA driver for block device access.

Having SWEB implemented for four different architectures in different branches, we tried to make SWEB a multi-platform compatible kernel. Therefore, we merged the four architecture branches and applied the design principles we saw in the Minix kernel and in the Linux kernel. It was necessary to abstract the paging data structures into a newly introduced object. We abstracted the initialization of hardware components. We adapted the build system to allow architecture specific building. Finally, we had to replace data type usage from 32-bit specific data types to more abstract data types, allowing to run the same code on systems with different pointer sizes.

In the analysis of the merged SWEB kernel we saw that there is room for improvement regarding code redundancy. We implemented some of the ideas, such as the architecture folder tree, which allows more flexible implementation of new architectures. The enhancements helped achieve a reduction of the existing source code by about 22% measured in lines of code. In the case that SWEB will be ported to more architectures the separation of drivers from the processor architecture dependent code might become necessary. Furthermore, there is still some potential in grouping more identical or similar code to improve the source code maintainability.

After the SWEB kernel structure was improved we ported SWEB to the Gumstix Verdex ARM board and the Raspberry Pi within a few days. This showed how well the structure supports porting SWEB to new architectures.

Bibliography

[ARM00] ARM Limited. *ARM Architecture Reference Manual DDI 0100E, Part B*, 2000.

[ARM02] ARM Limited. *Integrator/CP HBI-0086 User Guide 0159B*, 2002.

[ARM12] ARM Limited. *Procedure Call Standard for the ARM Architecture IHI 0042E*, 2012.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel - Third Edition*. 2005.

[Bro12] Broadcom Corporation. *BCM2835 ARM Peripherals*, 2012.

[Cha14] Alex Chadwick. CSUD Github Repository `https://github.com/Chadderz121/csud/tree/e13b9355d043a9cdd384b335060f1bc0416df61e` (retrieved on April 29, 2014), 2014.

[Eag07] Michael J. Eager. Introduction to the dwarf debugging format. 2007.

[Est14] Gregory Estrade. Torlus Qemu Github Repository `https://github.com/Torlus/qemu/tree/1e7829e34329c2a7ed9cd94084e96cdc3714f1e3` (retrieved on April 29, 2014), 2014.

[Fog14] Agner Fog. *Calling conventions for different C++ compilers and operating systems*. `http://agner.org/optimize/calling_conventions.pdf` (retrieved on May 5, 2014), 2014.

[GCC13a] Gcc manual: Intel 386 and amd x86-64 options. `http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/i386-and-x86_002d64-Options.html` (retrieved on April 29, 2014), 2013.

[GCC13b] Gcc manual: Options controlling c dialect. `http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/C-Dialect-Options.html` (retrieved on April 29, 2014), 2013.

[Har13] Alen Harbas. Bachelor thesis: Nand flash driver, 2013.

[HBG$^+$06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, pages 80–89, 2006.

[HI98] HP and Intel. Elf-64 object file format version 1.5 draft 2. May 1998.

[Int04] Intel Corporation. *Intel PXA27x Processor Family Developer's Manual*, 2004.

[Int12] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A*, 2012.

[Min14] Minix Source Repository `http://git.minix3.org/?p=minix.git;a=tree;f=kernel;h=3a00398f72d925c10dda245969934bf690fe3109;hb=HEAD` (retrieved on April 29, 2014), 2014.

[MJAM13] Matz M., Hubicka J., Jaeger A., and Mitchell M. System v application binary interface, amd64 architecture processor supplement, draft version 0.99.6. 2013.

[Ora13] Oracle. *Oracle® Solaris Administration: Basic Administration.* `http://docs.oracle.com/cd/E26505_01/pdf/E29492.pdf` (retrieved on April 29, 2014), 2013.

[OsD14] Creating a 64-bit kernel. OSDev Wiki `http://wiki.osdev.org/index.php?title=Creating_a_64-bit_kernel&oldid=16274` (retrieved on April 29, 2014), 2014.

[SD 13] SD Card Association Technical Committee. *SD Specifications, Part 1, Physical Layer, Simplified Specification, Version 4.10*, 2013.

[Tan09] Andrew S. Tanenbaum. *Modern Operating Systems - Third Edition.* Pearson Education, Inc., 2009.

[Too93] Tool Interface Standard Committee. *Portable Formats Specification*, 1993.

[Tor99] Linus Torvalds. *Open Sources: Voices from the Open Source Revolution - The Linux Edge.* 1999.

[Wei05] Andreas Weinberger. Sweb Developer Mailing List `http://sourceforge.net/mailarchive/message.php?msg_id=1373257` (retrieved on April 29, 2014), 2005.

[Wei06] Andreas Weinberger. Sweb Source Repository `http://sourceforge.net/p/sweb/sweb/ci/45be1e3576aa952beb04345f36ffcf6576157ddf/` (retrieved on April 29, 2014), 2006.