Porting of an Operating System kernel

Allan Juhl Petersen

Kongens Lyngby 2010
IMM-B.ENG-2010-21

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.ENG: ISSN

Abstract

When it comes to the domain of embedded technologies, the ARM processor architecture is very popular. According to Wikipedia[8] around 90% of all 32 bit embedded CPU's are based on the ARM processor architecture as of 2009. Its usage is growing in cell phones, PDA's, GPS devices, and netbooks. The fact that the ARM architecture has entered the netbook marked has made it a very interesting architecture for operating system development. Systems based on the ARM processor architecture, can become very complex machines since these are meant to support varied tasks such as memory management and process management. This means that an operating system has to be ported to these ARM processor architectures. An operating system redesigned to extract the maximum performance out of the hardware and still be stable and secure. Often such operating system porting is the work of specialized third party vendors having expertise on this domain. This Thesis describes the details of porting the FenixOS the research operating system to the ARM Cortex A8 processor architecture. It describes; the Kernel boot sequences utilizing an embedded bootloader, the utilization of a software ARM emulator, early kernel initialization where virtual memory is setup, IRQ/FIQ setup and finally contexts switches are also described.

Resumé

Når det kommer til indlejrede teknologier, såer ARM processor arkitekturen meget populært.Ifølge Wikipedia[8] er - fra 2009 - omkring 90% af alle 32 bit indlejrede CPU'er baseret påARM processor arkitekturen. Dens brug er voksende i mobiltelefoner, PDA'er, GPS enheder og netbooks. Det faktum, at ARM-arkitekturen har indtaget scenen med netbook markedet, har gjort det til en meget interessant arkitektur for udvikling af styresystemer. Systemer baseret påARM processor arkitektur, kan blive til højt udviklede maskiner, da disse er beregnet til at stætte forskellige opgaver såsom hukommelse og processtyring. Det betyder, at det er nødvendigt at portere et styresystem til disse ARM processor arkitekturer. Et styresystem som er designet fra bunden til at opnåden maksimale ydelse ud af hardware og stadig være stabil og sikker. Ofte er sådanne operativsystem porteringer et arbejde for specialiserede tredjeparts virksomheder med ekspertise pånetop dette område. Denne afhandling beskriver detaljer, porteringen af FenixOS - et forsknings operativsystem - til ARM Cortex A8 processor arkitektur. Den beskriver, kernens boot sekvens ved at bruge en indlejret bootloader, brugen af en software ARM emulator, tidlig kerne initialisering hvor virtuel hukommelse er sat op, IRQ / FIQ opsætning og endelig bliver context switches ogsåbeskrevet.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the B.Eng. degree in engineering. The thesis deals with different aspects of the porting of an operating system to a processor architecture using knowledge about the structure of the processor architecture and the knowledge of operating systems. The main focus is on the ARM Cortex A8 processor architecture and the operating system FenixOS, but the techniques can be applied when porting various operating systems to various processor architectures . The thesis consists of this report written during the period February 2010 - June 2010.

Lyngby, June 2010

Allan Juhl Petersen

Dedication

I would like to dedicate this thesis to my wife and kids, who stood by me throughout this education. It's been a long hard road, but now we're finally able to harvest the fruit.

Acknowledgements

First, thanks to my freinds and fellow students at DTU who participated in the research project of developing the operating system FenixOS. They all in some way contributed by supporting with review and conversations regarding glitches and problems in the project.

To Sven Karlsson for being a great supervisor on my project, and always being able to motivate me to strive further.

To Poul Bondo for reviewing this report, and for many insightful discussions on the project.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In this thesis I would like to describe the necessary steps to be able to port a new operating system to the ARM processor architecture. The recent years have shown that the need for a redesign of operating systems is necessary. An operating system which can utilize multi-core processors (multi-cores here means 2 - many) much better, more reliable and safer than the operating systems of today. And here FenixOS enters the picture. Recent years has also shown that the ARM processor, best known for its usability in embedded devices, has begun its entry on the netbook marked. This gives the advantage of an operating system could easily run on netbooks and embedded devices such as PDA's and SmartPhones (ARM has announced the work of a dual (multi) core processor for smartphones and netbooks).

My motivation for this thesis is first my belief in that ARM processors will become a major player in the netbook/smartphone marked (just as ARM recently is a major player on the embedded marked) and therefore it's essential to have an operating system ported and able to utilize the architecture of the ARM processor. Second, the operating system being ported should be one that has been re-designed with attention on the optimal utilization of processing, performance, stability, power consumption and security. This is because this will become essential parameters for smartphones and netbooks in the near future. While this is all good and well, no operating system will survive without supporting software already running on the currently developed operating systems.

It will simply loose its value on the marked. Imagine however an operating system able to support the software running on Linux (and variations), MacOS and Windows! I agree that the idea is not new, but it has yet to be developed to a working state. This is one of the end goals of FenixOS, but it is beyond the scope of this report to explain how this is achieved. It is mentioned here as a reason of my motivation to port FenixOS to the ARM processor. I will also argue that over time the raw processing power at the client end will become reduced by the introduction of cloud computing. The ability to move the processing power into the cloud, will make the processing power on client smaller as everything will be computed on the cloud server.

Due to the current state of the operating system FenixOS, it will not be possible to install and run a full-blown version of FenixOS on a live ARM processor. For this FenixOS is not mature enough yet. I will therefore run the port on QEMU processor emulator, which is an emulator that can emulate entire machines as well as processor types. I have used QEMU to emulate the Versitile development board running the ARM Cortex A8 processor architecture, running with a minimum of memory, namely 128 Megabytes.

1.1 An Overview of this Report

Chapter 2 - Problem Definition: In this chapter I try to carve out the problem at hand, giving an overview of the task I'm trying to solve throughout this report.

Chapter 3 - Background: In this chapter I build a the knowledge base needed to understand the task that I'm working on. I'll take you through subjects such as:

- What Is an Operating system?
  - Explaining the fundamental terms of the operating system
- Iterrupts and Exceptions
  - Explaining the basics of interrupts and exceptions, and the fundamental differences between them.
- Process
  - An introduction to the term of processes which is a fundamental thing of an operating system.
- Memory management
  - Explaining the fundamentals of memory management. This is a more in depth introduction, since this part is the concept hardest to remember throughout an IT-university eductaion.
- CPU
  - What is an CPU, and also the very important differences among Single-core, Dual-core and Multi-core CPU's
- ARM Architecture
  - a brief overview of the ARM processor architecture.
- FenixOS
  - A brief overview of what FenixOS is all about.
- Bootloader
  - Explaining what a bootloader is, and how it works on the PC and embedded platforms.

- ELF-Image
  - A short overview of the ELF binary format.

Chapter 4 - Theory: This chapter is in part a discussion on why I've made the choices that I have, and a design and implementation overview of how I implementetd the choices I made into this project.

- Theoretical Discussion
  - An overview and a discussion on the choices I made for this project, and why I chose as I have.
- Kernel Initialization
  - An overview of the design choices and implementation of the kernel initialization code.
- Analysis of Further Steps
  - In this section I explain about the analysis I have made based on the future implementation of the Kernel code.

Chapter 5 - Conclusion: In this chapter I present my conclusion on the project as a whole.

Chapter 6 - Future Work: In this chapter I mention steps for the further and complete development the code.

Chapter 7 - Project Planning: This chapter is all about the project management of my project. Here I explain the thoughts and choices I've made along the way, to get the project delivered in due time.

Chapter 2

Problem Definition

Due to the future will be dominated by notebooks and embedded devices possibly with many multi core processors and the fact that ARM processor architecture is a strong leading competitor in the embedded marked and now moving into the notebook marked, the need for a new and redesigned operating system is essential. An operating system build from bottom with focus on multi cores, performance, security, stability and power consumption and this operating system is FenixOS. FenixOS has until now only been running on x86 processor architecture and now I will try to port It the ARM processor architecture.

2.1 Defining the problems

The first part of running the operating system is to power on. During the power on process we want to boot the hardware and set up a default usable hardware configuration to load the kernel.

In order to speed up the development and ensure a proper quality of the development of the code for the project we need a proper setup that facilitates easy and reliable debugging. It should also have a quick turn around and finally it should be low cost, i.e. within reasonable cost for a thesis project.

To be able to ensure that the project will survive the future, a processor with future prospect must be selected. It must be widely used in the present, and it must be focused on performance and power consumption.

For the future in hardware, a new operating system must be found, one that is designed for the future with stability, performance, security and power consumption in mind. An OS designed for the future in hardware.

In order for the kernel to run in virtual memory we have to setup the CPU to run in a mode that provides us access to the registers needed. We need to setup the MMU with caches / buffers and page tables. In the end we need to enable the MMU and setup a stack for the C environment to run in.

For protection, we need to have the kernel running in kernel space. The kernel must also be able to handle interrupts, whether its exceptions or interrupts. We need to setup handling for software interrupts in a way that system calls can be implemented, and we need to have the ability to switch process contexts.

Chapter 3

Background

In this chapter I'll try to outline what an operating systems is and explain some of the things that lies under the hood and makes it tick. Some parts are more detailed than others, and I have done so deliberately to detail out what I feel are important things to know when reading this report. I could have left out some sections, but I felt it would leave the report as incomplete. I will go through what an operating system is, and then dive into some of the things an operating system consists of, that users normally know nothing about, such as the design of operating systems, processes and memory management. I'll end this section off by going through what an CPU is, some design aspects of the ARM CPU in general and at last I will outlining what FenixOS is all about.

3.1 What is an operating system?

An operating system[21] (OS) is the software that communicates with computer hardware on the most basic level also called an abstraction between software and hardware. Without an operating system, running software will be hard to do. The operating system is in charge of allocation and assignment of system resources such as I/O devices, software (Processes), central processing unit (CPU), memory, and serves as an interface towards the user and therefore an

operating system can be viewed as a resource manager. In general an operating system has two more types of functionality, namely scheduling; coordinating resources and jobs by following certain given priority patterns, and monitoring; keeps track of the activities in the computer system. It maintains logs of job operation, notifies users of any abnormal terminations or error conditions. An operating system is divided into layers called user space and kernel space where the kernel (the core of the operating system) is running in kernel space, and out of reach from user space. The programs (applications) - which reside in user space - can only communicate with the kernel by issuing a system call, and thereby requesting a service that the kernel provides e.g. I/O.

3.1.1 Types of Operating Systems

There are several types of operating systems, with Windows, Linux and Macintosh being the most widely used. Here is an overview on each system:

Windows: Windows is the popular Microsoft brand preferred by most personal users. This system has come a long way from version 1.0 all the way up to the new Windows 7. This new version of Windows (Win 7) is also used as an embedded system, it is used as operating system on smart phones, information boards at train stations etc. Before Windows 7, a special designed Windows XP was used on the information boards, and yet another special designed operating system was used on smart phones. Now Microsoft has build Windows 7 so that developers are able to remove unwanted features all the way down to removing only the frame from Internet Explorer, or the entire Internet Explorer itself.

UNIX/Linux: The UNIX operating system has been around for years, and it is well known for its stability. UNIX is often used more as a server than a workstation. Linux was based on the UNIX system, with the source code being a part of GNU open-source project. Linux is well know on the embedded marked. Today Linux is running on things such as smart phones to payment solutions on petrol stations.

Macintosh: Recent versions of the Macintosh operating system, including the Mac OS X, follow the architecture of UNIX. Systems developed by Apple are efficient and easy to use, but can only function on Apple branded hardware. Embedded software running Mac OS X are IPhone and IPods.

3.1.2 Different design models of Operating Systems

Monolithic: The most common Operation System design, in quoting Andrew S. Tanenbaum in his book - Operating Systems Design and Implementation, Third Edition - "this approach might well be subtitled The Big Mess." The structure is that the Operating System all resides in the kernel space along with device drivers and file systems. Leaving applications to run in user space and communicating with the Operating System through System calls, this meaning that when an application (in user space) needs to request resources such as memory it must make a system call to the kernel to get the resources. This design model is used in UNIX, LINUX and Windows Operating Systems.

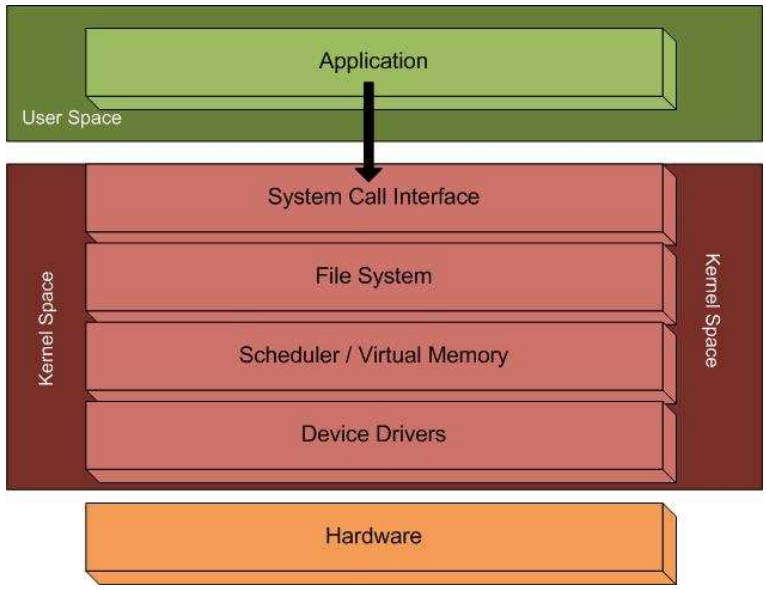

Figure 3.1: An overview of an operating system based on the monolithic design model.

Microkernel: This Operating System design is very much different than that of the Monolithic, in the way that the structure has moved as much as possible into user space. In the Microkernel structure things such as File systems, device drivers and etc. is moved into user space. Scheduling, Virtual Memory and basic IPC resides in the kernel space. Microkernel works with the notion of Servers, which essential is your well known daemon (as daemons on Linux systems), just with special privileges granted from the kernel to interact with parts of the physical memory (which is off limits for to most programs). Microkernel design also works with the notion of IPC (Inter Process Communication), which are processes that communicate among one another by message passing. In this fashion processes can invoke services on the system to fulfill requests. The Microkernel design is by some argued to be the most stable design model, due to the use of servers (file servers, device drivers etc.) and that these are located in user space. If a device driver should crash the kernel would not be affected, and the service can be restarted without crashing the kernel. This design model is used in Minix, L4, QNX and Mac OS X.

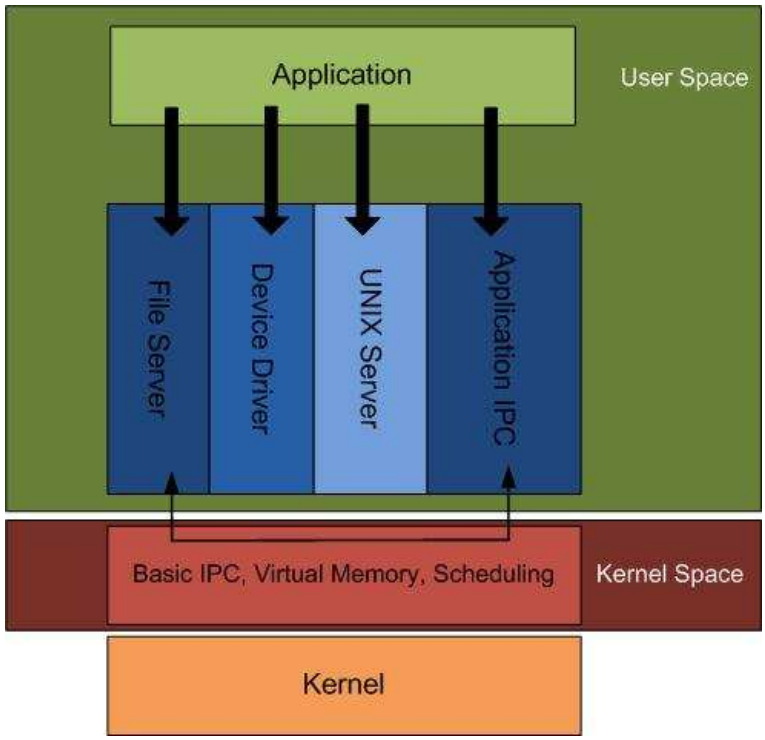

Figure 3.2: An overview of an operating system based on the microkernel design model.

3.2 Interrupts and Exceptions

Interrupts are usually defined as signals that change the order of instructions that currently are executed by the processor. Interrupts are often divided into synchronous (exceptions) and asynchronous (interrupts) signals:

- Exceptions are produced by the CPU control unit while executing instructions and are called synchronous because the control unit only issues them after the execution of an instruction has finished.
- Interrupts are generated by other hardware devices at random times.

Interrupts are issued by interval timers and I/O devices; for instance, the push of a key on a PS2 keyboard from a user sets off an interrupt. The effect of this behavior is that the CPU is told to immediately stop the current execution, save the current process' state to registers, and jump to the register containing the information regarding the interrupt.

Exceptions are caused by two things; either by programming errors or by abnormal conditions that must be handled by the kernel.

If more than one interrupt is waiting to be handled the operating system usually has instructions called an interrupt handler. The interrupt handler prioritizes the interrupts and saves them in a queue, to be executed in the prioritized order. There are 256 interrupts in all but only 16 are used by devices, and these are called IRQs or hardware interrupts. Non-IRQ interrupts are often used for APIs, or functions that can be called by applications and these are called software interrupts. Note that the first 32 interrupts are reserved and used by the CPU, as exceptions.

3.3 Process

The notion of a process is fundamental to an operating system. The general definition of a process is that a process is an instance of a program currently executing, including the current values of the program counter (PC), registers and variables. In multiprogramming operating systems (Linux, Windows, Mac OS X etc.) the process thinks it has its own CPU, where in fact the real CPU just rapidly switches the processes back and forth to give the illusion that the many programs (processes) can run i parallel at the same time. A modern operating system makes use of foreground processes and background processes, the latter called daemons. An example of a background process could be a webserver waiting for service requests. The difference between a process and a daemon is that a daemon runs in until explicitly terminated or until the system is shut down and, whereas a process often lives a shorter life. A daemon often lives its life without the interference from the user. The kernel itself is neither process or daemon, but more a process manager, and its the job of the kernel to create, terminate and synchronize the processes of the system. Operating systems operate in different modes; user-mode in user pace, kernel-mode (privileged mode) in kernel space (These modes are actually modes that the CPU run in, and often there are several more modes for a CPU to run in e.g. IRQ-mode, system-mode etc.). This means that a program executed in user-mode cannot directly access kernel data or kernel programs. When a program executes in kernel mode, these restrictions no longer apply. Each CPU manufacturer provides special instructions to switch between user-mode and kernel-mode. Normally a program executes in user-mode, and only switches to kernel mode when requesting a service that the kernel provides. It does so through the use of a special programming construct called a **system call**. See figure 3.3 for information on how a system call is done.

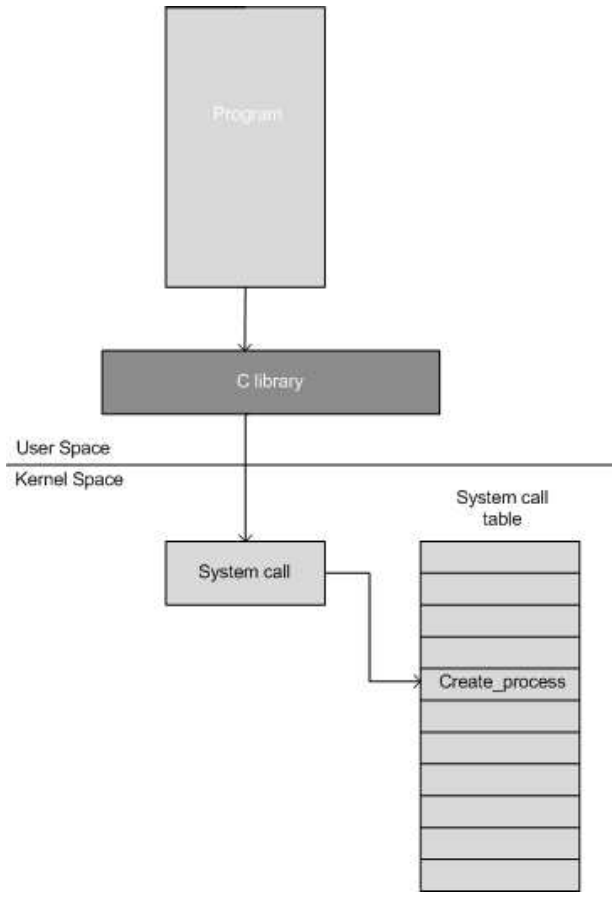

Figure 3.3: A program from user space makes a system call

Each process runs in its own private address space. A process running in user-mode refers to a private stack, data and code areas, and when running in kernel-mode the process addresses the kernel data and code areas, but uses another private stack.

In the figure 3.4 I illustrate how a process is organized internally.

The TEXT segment (code segment) is where the binary image of a process is stored. In other words, it is here the compiled code of the program itself is stored. The DATA segment is where Static variables that have been initialized by the programmer is stored. The BSS segment is where uninitialized static variables

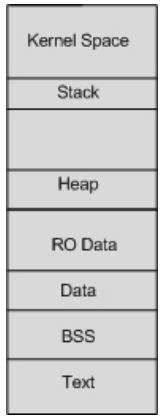

Figure 3.4: An overview of a process.

are stored. The stack, is where the local variables and function parameters are stored. Calling a method/function pushes a new stack-frame onto the stack. This stack frame is destroyed when the function returns. The data put onto the stack obey the rules of LIFO (Last-In-First-out), and all that is needed to track the contents of the stack is a pointer at the top of the stack. The heap - like the stack - provides runtime memory allocation (also called dynamic runtime memory allocation), but unlike the stack, this memory allocations is meant to outlive the function doing the memory allocation. The fact that the memory allocation outlive the function that called it, means that the memory requests are a shared effort between the language runtime (e.g. C# garbage-collection) and the kernel. Figure 3.5 Illustrates the variables stored in memory.

```
static in BSS-Stored; // This is stored in the BSS segment.

static int DATA-Stored = 2; // This is stored in the DATA segment.
```

Although:

```
static *prg = "This isn't stored in the DATA segment";
```

In this case, the contents of the pointer (the memory address) is stored in the DATA segment, but the actual string it points to lives in the TEXT segment.

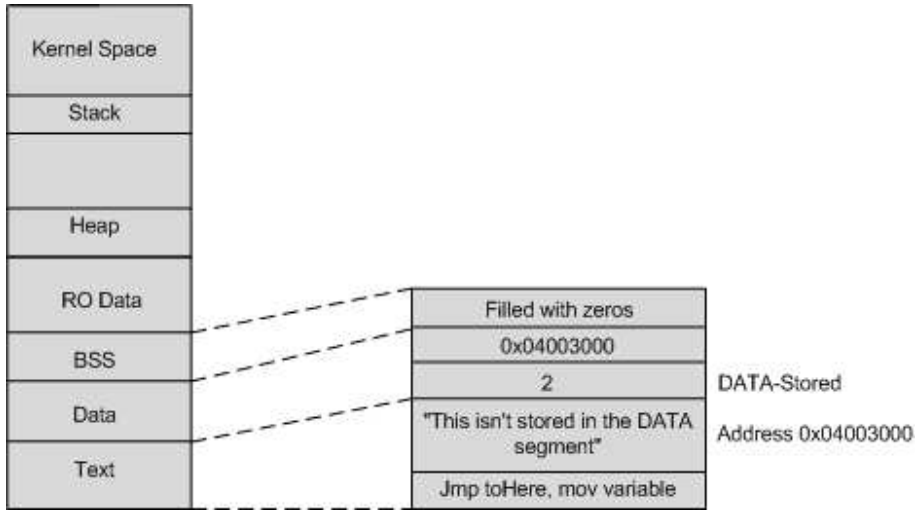

Figure 3.5: An overview of a process with contents in DATA, BSS and TEXT segments.

3.4 Memory management

Almost all operating systems today makes use of virtual memory[3][20], this allows for the all-over size of a program, data, and stack to exceed the amount of physical memory provided by the hardware. This means that a 256MB program can run on a machine that only provides 128MB of physical memory. It does this by carefully loading the bits program it needs into physical memory and swapping the bits in and out as they are needed. The approach often taken is to make use of a technique called **Paging**, which uses a term called a **Page** for a virtual memory address, a term called **Page-frames** for physical memory addresses and a term called **Page-tables** that allows the CPU to look up physical addresses. And in the middle of it all we have the MMU (Memory Management Unit)[2] which is in charge of the actual translation from virtual memory addresses to physical memory addresses, and the TLB (Translation Lookaside Buffer) which helps by speeding up the translation process. Do note that the memory know not of any MMU, it only sees the physical memory addresses.

The virtual address space is divided into small units called **Pages**, this contrasts to the physical address space where units are called **Page-frames**. A page and a page-frame are always identical in size, and often enough that size resides to 4KB. This means that if a program has got he size of 32KB, it takes up 8 pages.

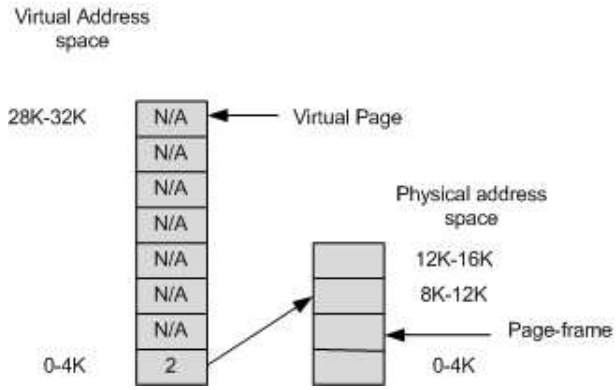

Figure 3.6: Relation between virtual memory addresses and physical memory addresses.

Imagine a system as shown in figure 3.6 which have 32KB of virtual memory and 16KB of physical memory, this gives us 8 pages and 4 page-frames (pages are virtual memory and page-frames are physical memory). When the address 0 is send to the MMU, it maps the address to page-frame 2 and outputs the physical address 8192, and thereby the MMU has mapped all virtual addresses from 0-4095 to physical addresses 8192-12287.

When eventually all the page-frames are full (and in this case it will be due to the fact that virtual memory is twice as big as the physical memory and therefore only half of the virtual memory[4] can be mapped at any given time) the hardware keeps track of the pages present in memory by using a present/absent bit. When a program tries to use a page that is not mapped, a **Page-fault** occurs and the operating system choses a little used page-frame, writes it back to disk, gets the page just referenced and stores it into the newly freed page-frame, corrects the mapping and restarts the instruction.

The specific internal workings of the MMU is processor specific, but usually it's in the ballpark of; The page number is being used as an index into the page-table, and depending on the size of the address (32 bit / 16 bit) the page-frame number located in the page-table is copied to the high-order bits of the output register along with a specified offset (the offset is copied as is). See figure 3.7

The sole purpose of the page-table is to map virtual pages on to physical page-frames, and every process has its own set of page-tables. Due to the fact that most systems today use 32 bit virtual addresses (and with a page size of 4KB, that gives an address space of 1 million pages!) the page-table becomes very

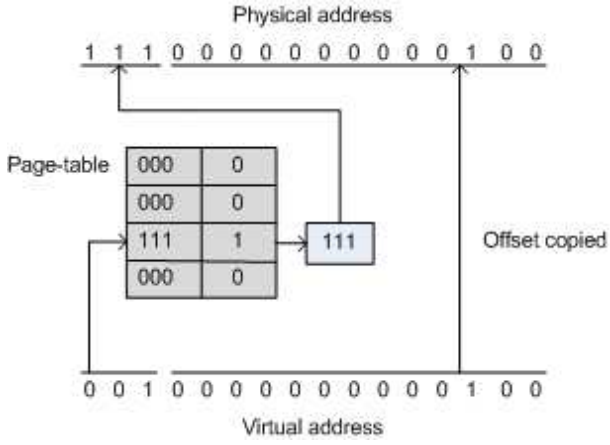

Figure 3.7: Internal working of a MMU.

large, and (as stated earlier) every process has its own set of page-tables (because it has its own virtual address space). And the fact that the virtual to physical address mapping is done on every memory reference (It is often necessary to make two or more memory references pr. instruction), the usage of multi-level page-tables and TLB (Translation Lookaside Buffer) has become almost standard.

When using Multi-level page-tables, you avoid the need of keeping all the page-tables in memory at all times. Those page-tables not needed at current time is simply not mapped.

As can be seen on figure 3.8 the 32 bit address field is split into two index fields and one offset field e.g. 10 bit index into first-level (FL) page-table, 10 bit index into second-level (SL) page-table and a 12 bit offset. When a virtual address is send to the MMU, it extracts the FL field and uses the value as index into the first-level page-table. The entry found at that index holds the address or page-frame number of the second-level page-table and now the SL field is used as index into the selected second-level page-table to find the page-frame number for the page itself. This gives the interesting thing that only the page-table needed will be mapped e.g. the first-level page-table, and the second-level page-tables that are indexed.

To aid in the fight for performance on memory references, computers (or CPU's) have been equiped with a small hardware device called the TLB, that rapidly maps virtual to physical addresses without using the page-table. The TLB

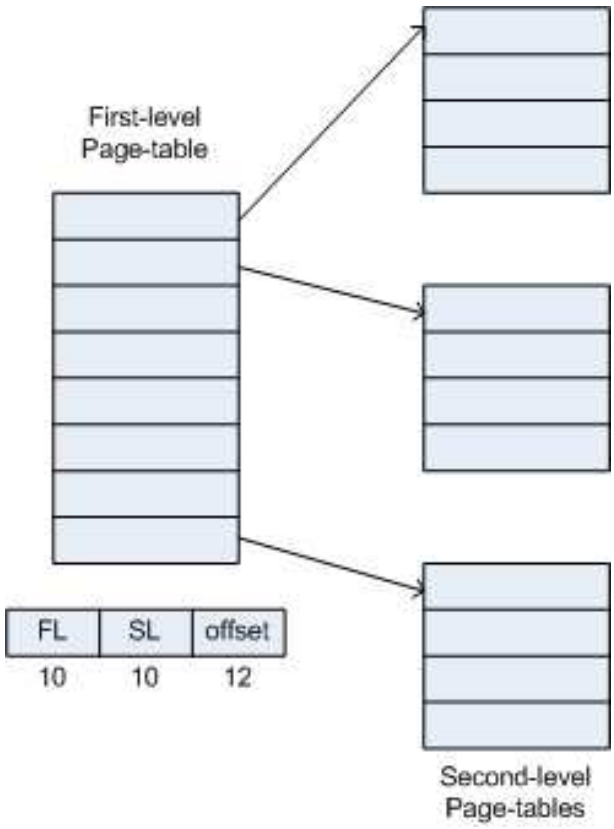

Figure 3.8: Overview of a two level page-table

consists of a small amount of entries, where each entry hold information of one page (virtual page number, modified bit, Read/write/execute permissions, physical page-frame and valid bit). When an address is send to the MMU, it now looks if the virtual page number is present in the TLB, by comparing the number to all the entries in the TLB, and it looks at all the entries simultaneously (as, in parallel). If it exists in the TLB, the page-frame is taken directly and without using the page-tables. If the virtual page number does not exist in the TLB, a page-table lookup is completed, and an entry in the TLB is removed and replaced with the page-table entry just found.

3.5 CPU

When I look up the question; What is a CPU?, almost everywhere the same answer appears: *"The Central Processing Unit (CPU) is responsible for interpreting and executing most of the commands from the computer's hardware and software. It is often called the "brains" of the computer."* [5]. And I believe this to be true and that this sounds reasonable.

3.5.1 Single-core CPU

On a single-core processor, the CPU can only handle one operation at a time. It works by receiving a string of instructions that it must order, execute and then store in its cache (a cache is a step between the processor and main memory (RAM) named Level 1 cache, Level 2 cache etc. This functionality makes it quicker to retrieve data stored in the faster caches than retrieve data from main memory, but these caches are very expensive, and therefore often very small in sizes e.g. 512KB - 32MB). Accessing data from outside the cache e.g. RAM or Hard drives slows down performance, and this situation is aggravated when multi-tasking, due to the CPU switching back and forth between processes.

3.5.2 Dual-core CPU

On a dual core processor each core handles strings of instructions simultaneously to improve efficiency. While one core is executing code the other can be executing its own code. For an average user the difference in performance will be most noticeable in multi-tasking until more software is SMT* (simultaneous multi-threading) aware. Servers running multiple dual core processors will see an appreciable increase in performance. This has the significance that operating systems must be able to recognize multi-threading, and that the software must be written with SMT in mind. SMT enables parallel multi-threading where the cores receives multi-threaded instructions in parallel. See [6] . Without SMT the software will only recognize one core.

3.5.3 Multi-processor

Multi-processor systems are different from dual core processor systems. In the former there are two physically separate cores with their own resources. In

the latter, resources are shared among processors and the cores reside on the same chip. There is nothing preventing a multi-processors system from running CPU's consisting of Dual/Multi-cores.

3.6 ARM Architecture

The Cortex A8 processor[7] is a 32 bit RISC (reduced instruction set computer), a high-performance (Influenced by multi-tasking OS system requirements), low-power, application processor with full virtual memory capabilities. The features of the processor include:

- Full implementation of the ARM architecture v7-A instruction set.
- 14 stage pipeline for executing ARM integer instructions.
- Dynamic branch prediction with branch target address cache, global history buffer, and 8-entry return stack.
- Memory Management Unit (MMU) and separate instruction and data Translation Look-aside Buffers (TLBs) of 32 entries each.
- Level 1 instruction and data caches of 16KB or 32KB configurable size.
- Level 2 cache of 0KB, 64KB through 2MB configurable size.
- Seven basic operating modes, where each mode has access to own stack and a different subset of registers (Some operations can only be carried out in a privileged mode). The seven modes are:
  - Supervisor - Entered on reset and when a Software Interrupt instruction is executed - Runs in privileged mode.
  - System - Privileged mode utilizing the same registers as User mode.
  - FIQ - Entered when a high priority (fast) interrupt is FIQ raised - Runs in privileged mode.
  - IRQ - Entered when a "normal" interrupt IRQ is raised - Runs in privileged mode.
  - Abort - Entered when memory access violations need to be handled - Runs in privileged mode.
  - Undef - Entered when undefined instructions need to be handled - Runs in privileged mode.
  - User - Mode under which most Applications / Operating System tasks run - Runs in un-privileged mode.

- 37 registers, a subset of these registers is accessible in each mode.
- ARM architecture supports 16 co-processors (a co-processor is an external chip connected to the ARM data and control buses), the system control co-processor is a set of registers that you can write to and read from, where some of the registers allows more than one type of operation.

3.7 FenixOS

FenixOS is a research operating system, which is aimed at the hardware specifications of computers today, but even more at fact that computer systems of the future will contain even more CPU's than currently computers. This fact and the fact that current operating systems all have a creation date ranging from the 70's through the 90's where uni-core systems where common, has made it due time to refactor the architecture of operating systems. FenixOS supports an arbitrary number of CPU's and focuses on stability, security and the need for less power usage. It leans towards the microkernel architecture to enhance stability issues, and enforces security through access control lists - among others. The reason that FenixOS "leans" towards a microkernel structure, is that FenixOS tries to utilize the same notion of servers, and move parts out of kernel-space. FenixOS does not implement everything in user-space just because it is possible (like say Minix does). FenixOS currently holds device drivers and file systems in user-space. A complete rewrite of the underlying system structure is made in C++ and assembly, to clean up the disorderly structure seen in many current operating systems. Although everything is new and cleaned up, FenixOS is still set to be compatible with current applications running on current operating systems (FenixOX aims to be POSIX compliant), this meaning that applications that run on Linux, MacOS and Windows is aimed to be able to run on FenixOS.

3.8 Bootloader

In this section I will describe the what a boot loaders is on the different platforms - PC (x86) and embedded devices (ARM). And try to explain the differences in how an operating system is loaded on the different architectures.

3.8.1 Embedded platform

When a computer is first powered on, it does not have an operating system in ROM or RAM. The computer must initially execute a small program stored in ROM (A CPU can only execute code found in Read-Only Memory (ROM) and Random Access Memory (RAM)) along with a minimum of data needed to access the nonvolatile devices (Today's operating systems and application's store their data on devices, such as hard disk drives, CD, DVD and USB flash drive - called nonvolatile devices) from which the operating system programs and data are loaded into RAM.

The small program that starts this sequence of loading into memory is known as a bootstrap loader, bootstrap or bootstrap environment or boot monitor or boot loader. These programs reside in a special region of flash memory on the embedded hardware, and provide the necessary routines to load in an operating system to flash memory and execute it. Often, multiple-stage boot loaders are used during which several programs of increasing complexity sequentially load one after the other in a process of chain loading. In embedded systems these boot monitors often cover the first- and second-stage boot loaders and they - as with PC systems - often perform some level of system test and hardware initialization.

An overview of the boot process in an embedded system can be seen on figure 3.9.

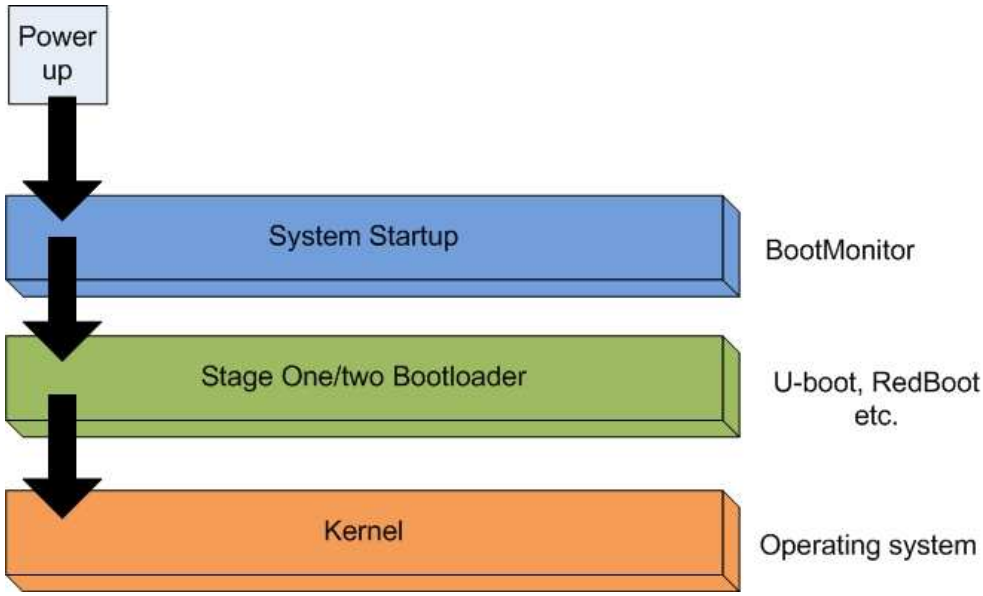

Figure 3.9: Booting process for an Emebedded Systems. 1) System Startup - BootMonitor, the program runs from a fixed location in Flash memory. 2) stage1/2 bootloader - U-boot,Redboot etc. 3) kernel - Operating system

Known embedded bootloaders

- U-Boot - Used by a big part of the opensource community as bootloader for ARM processor architecture and on embedded devices such as mobile/smart phones and PDA's
  - U-Boot can display boot menu on the serial port and after that boot from a lot of different media, AFAIK, including internal flash, network, FC and SD card.
  - **Supported Architectures**
    - PPC, ARM, AVR32, Blackfin, Coldfire, IXP, Leon2, m68k, MicroBlaze, MIPS, NIOS, NIOS2, PXA, x86, StrongARM, SH2, SH3, SH4
  - **Supported Filesystems**
    - FAT, VFAT, ext2, ext3, jffs2, cramfs, reiserfs, yaffs2, ubifs, nfs
  - **Supported Executables**
    - ELF, U-Boot image format

  - **Supported Decompressions**
    * bzip2, GZIP
  - **Supported Protocols**
    * TFTP, NFS, serial (S-Record, Y-Modem, Kermit binary protocol)

- QI - Developed specifically for the OpenMoko Neo FreeRunner Mobile phone (An Open Source Phone, in both Soft as well as hardware).
- Smart-QI - Further development of QI bootloader. Currently no support for Cortex A8 processor.
- RedBoot - Supports a vast amount of processor architectures, including ARM, PowerPC, x86, MIPS and Sparc.

3.8.2 PC platform

When a PC is powered on the BIOS launches the Power-on- Self-Test (POST) which tests various components in the computer. The POST does a mixture of testing, initialization, and the sorting out of resources such as; interrupts, memory ranges and I/O ports. After the POST the BIOS wants to boot up an operating system, which resides on any one of the following; hard drives, CD-ROM drives, floppy disks, etc. The BIOS now reads the first 512-byte sector (sector zero) of the hard disk (see figure 3.10). This first sector is called the Master Boot Record and it normally contains two vital components; a tiny OS-specific bootstrapping program at the start of the MBR followed by a partition table for the disk. The BIOS however does not care about any of this it simply loads the contents of the MBR into a specific memory location and jumps to that location in order to start executing the code that is in the MBR.

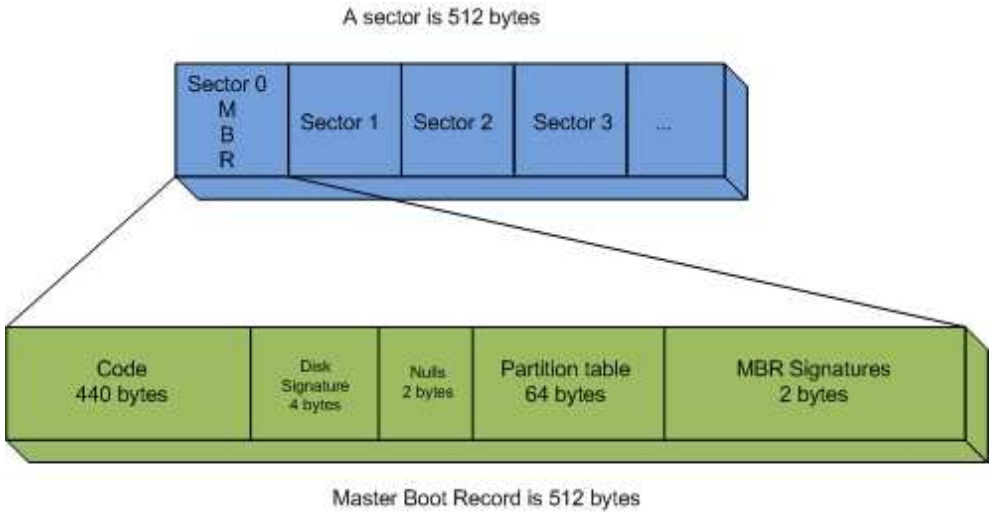

Figure 3.10: Overview of the Master Boot record.

The specific code in the MBR could be the Windows MBR loader, the code from Linux loaders such as LILO or GRUB, or even a virus. The partition table is a 64-byte area with four 16-byte entries describing how the disk has been divided up (so you can run multiple operating systems or have separate volumes in the same disk). This leaves us at the place where a jump from "Second stage Bootloader" to the "Kernel Initialization" is done and the operating system kernel now starts to be initialized.

An overview of the boot process in modern PC's can be seen on figure 3.11.

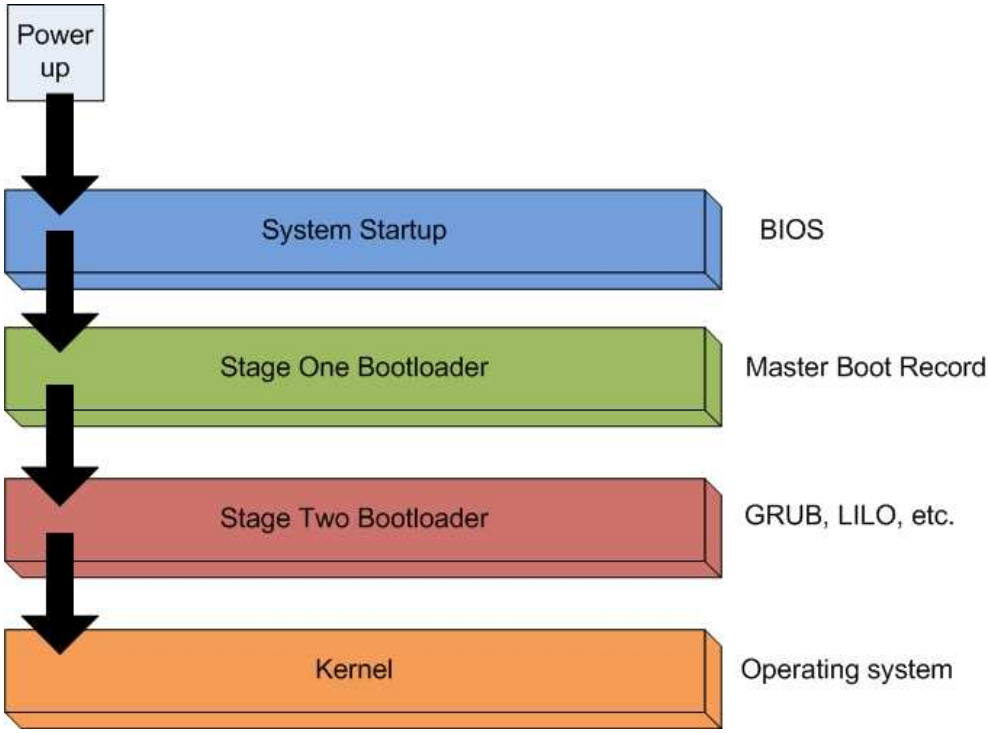

Figure 3.11: Booting process for a PC. 1) System Startup - PC-BIOS/BootMonitor. 2) Stage1 bootloader - MBR. 3) stage2 bootloader - LILO, GRUB etc. 4) kernel - Operating system

Known PC bootloaders

- GNU Grub legacy / 2
- LILO - Only works with x86 Processor architecture
- NTLDR - NT-Loader
- U-Boot - Less know on x86 processor architecture, but it does support this architecture to same extend as GRUB

3.9 ELF image

ELF-image (see figure 3.12) is a executable binary format designed to support dynamical objects and shared libraries. It is also used as an executable format for binary images used with embedded processors like ARM. The ELF-image is organized in segments and sections. Sections contains; TEXT (executable), Read-Only DATA (Read Only) and DATA (Read/Write), all organized in the Section header table. Segments are are organized in a Program header table, which contain zero or more sections. Segments are used to show information needed for run-time execution of the file, where sections are used for linking.

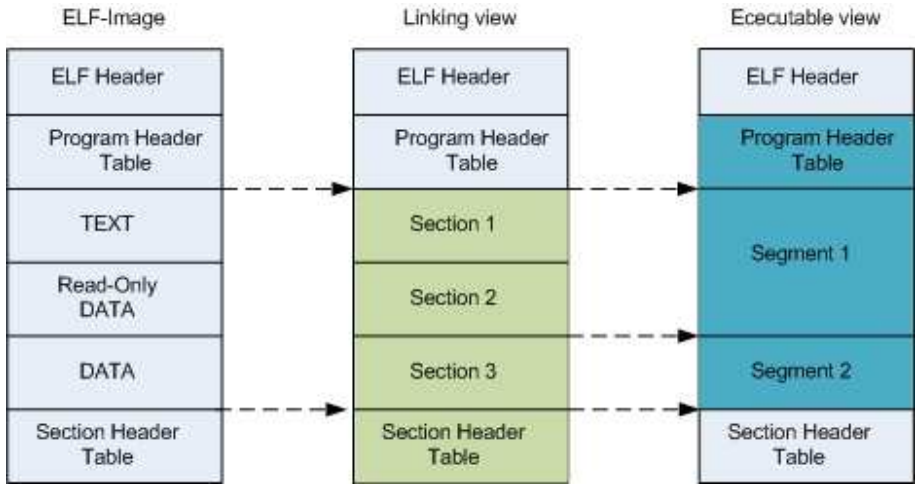

Figure 3.12: Overview of the ELF-Image format in linking view and executable view.

Chapter 4

Theory

Due to the future will be dominated by notebooks and embedded devices with multi core processors (multi as in 2 - many) and the fact that ARM processor architecture is a strong leading competitor in the embedded marked and now moving into the notebook marked, the need for a new and redesigned operating system is essential. An operating system build from bottom with focus on multi cores, performance, security, stability and power consumption and this operating system is FenixOS. FenixOS has until now only been running on x86 processor architecture and now I will try to port It the ARM processor architecture.

4.1 Theoretical Discussion

In this section I will set the stage for the choices I've made during this thesis. I'll make a theoretical discussion back and forth to explain why I've chosen; the processor I have, the operating system I have, the bootloader I have and finally the environment settings I have.

4.1.1 ARM

When choosing the ARM processor rather than Intel it comes down to a showdown between titans. First of all, the biggest participant on the embedded marked today is ARM, but they have certainly moved up in regards to the netbook marked also when comparing ARM's arsenal of processors against Intel's. Intel produce their ATOM processor for the netbook marked, which is a quite impressive processor line with the leading processors being the Atom 300 (Diamondville) from 2008 which is a 64 bit dual-core CPU with hyperthreading (45 nm) 1.6 GHz and the Atom D510 (Pineview) from January 2010 which is a 64 bit dual-core CPU with hyperthreading and on-die GPU (45 nm) 1.66 GHz, with 1 MB of L2 cache. But looking at what Intel has instore for the future lines of processors, they seem to "fall back to their roots" and flood the marked with processors of high clock rates based off course on the new and faster DDR3 RAM technology. Intel is in my opinion not mature enough on the embedded/netbook marked, not when considering the history of ARM processors on the same marked. ARM has proven their processor architecture in regards to perfomance, stability and last but definitively not least power consumption wise. Intel does lack severely in performance and power consumption when facing ARM processors. When looking at the netbook marked, ARM hasn't been a big participant here. But with their experience on the embedded marked, their new processors will most likely swoop the marked due to a few simple reasons; the future is not based on a few processors able to run 5 GHz in a device, but rather a lot of really small processors working in coherence with one another to get the job done. I'm thinking 500 maybe even 1000 processors running on a desktop/laptop/netbook making the systems faster (thinking in throughput) and more reliable. But even so, in the present ARM still has a line of Cortex A9[10] MPCore^{TM}processors coming out running as dual-cores, and quadro-cores with up to 8MB of L2 cache through the optional L2 cache controller, still implementing the well supported ARMv7 architecture. Second of all, the ARM processor architecture is a well tested platform when it comes to operating systems making the process of porting a bit easier, although not well documented. This is not intended to be clash between open source and proprietary operating systems, but in the nature of porting an operating system, the support and documentation is easier to get when leaning on the open source community. Thirdly, when using one ARM processor architecture it is in my opinion fairly easy to port to newer ARM processor architectures due to the nature of the nature of RISC (Reduced Instruction Set Computing). Forth, when testing the ported operating system on actual hardware is currently well supported and cheaper when utilizing the ARM processor architecture.

When looking at how I see the future in computer/device design (thousands of processor cores on a computer/device), the way processors are today, who's

leading the netbook/device processor marked, the fact that I believe that the need for a new operating system design to accommodate these many cores (and stability, performance, security and power consumption), and the fact that ARM processors now run at least Linux and Windows 7 operating systems (and many more) makes my choice go to the ARM processor architecture when porting an oprating system based for the future.

looking into the near future, I wanted to find a ARM processor architecture that is widely used. This would make the port more successful to target a processor with the potential to be here in the near future, and this means that I have to target an architecture that at current point in time is used in most successful products, also with the likelihood of being here in a near future. To this extend my choice fell on the ARM Cortex A8^{TM}architecture. This architecture is used in products such as[8]: Apple iPhone 3GS, Apple iPod touch (3rd Generation), Apple iPad (Apple A4 processor), Apple iPhone 4 (Apple A4 processor), Motorola Droid, Palm Pre, Touch Book, Nokia N900 and many more. And whether or not people would like to admit it, when an Apple product is launched with a processor, it means that the performance must be from the top shelf. Based on all this, the Cortex A8 processor architecture seems like a sensible choice.

4.1.2 FenixOS

The common operating systems of today are based on technology developed many years ago when topics as scalability, security and power consumption was not a major concern in computing, like it is today. This is the primary motivation for developing a new operating system from the bottom with the aforementioned issues taken into account from the start. This operating system is called FenixOS.

The future of embedded and PC platform operating systems must be one of many qualities. As stated, the design of operating systems today, are based on old ideas, and technologies as single-core CPU's. These ideas are - in a near future - not applicable to operating systems of the future. This is due to the fact that even embedded systems will soon be running on multi-core CPU's. Sure the operating systems have been trying to pursue the changes in technology that has arrived throughout time, but they have been doing so in a manner that had bloated the system design to try to fit these new changes into the designs at that time. There is a growing tendency in the world of operating systems to start re-thinking the original ideas of how an operating system design should look like[9], and how would the hardware of the future look like. All this is necessary to create stable, secure operating system that performs to the hardware's fullest

potential and taking great notice of the power consumption on netbooks and embedded systems.

One such operating system is FenixOS, which in design should adhere to the standards of today, while still being designed for the future. FenixOS takes the best the Micro kernel structure, and combines it with re-designed security plans, stability plans and performance. It is here that the sense of porting FenixOS to ARM comes into the picture. As stated earlier, I firmly believe that the ARM processor architecture will prevail on the embedded and netbook marked of the future. This belief combined with the beliefs of the FenixOS operating system, makes it a pair of the future in regards to; security performance, stability and compatibility.

4.1.3 Bootloader

When in the marked for a bootloader I took some time reading up on which kind of bootloaders would best suit the needs. There are a few things to bare in mind when trying to melt together worlds by bringing ARM to the netbook marked. I looked at the well know and widely used GNU GRUB[15] (both legacy and 2). The problem with this bootloader is the obvious reasons that it is designed for the x86 processor architecture, and it does support it well, but ARM has for many years been boxed as a processor for embedded devices, and therefore GRUB does not support ARM or the Cortex A8 processor architecture. This is also the case with the LILO[16] bootloader, which also excels with its lack of scripting abilities. My attention quickly turned towards bootloaders primarily targeting embedded devices. And here I found a few; QI[12], smart-QI[13] and RedBoot[14] which all support ARM, but does not support the Cortex A8 processor architecture. I was not interested in starting my own hack on one of these bootloaders to make it work with the ARM Cortex A8 processor architecture. I needed a bootloader which could work on x86 processor architecture if necessary (not a strict demand), would work on both a software emulator and hardware development board, and last but very significant, it should support the ARM Cortex A8 processor architecture. Luckily such a bootloader does exist! Das U-boot[11] (the universal bootloader) has for a long time been a well-supported bootloader, on both embedded devices (e.g. OpenMoko Neo-freerunner touch phone) using ARM (and a wide variety of other processors) , x86 processor architecture, build in support from hardware emulators and well tested support on hardware development boards. With this bootloader, switching between hardware and software testing should be a matter of minor tweaks. The specifications of the u-boot bootloader can be seen in 3.8.1.

4.1.4 Runtime Environment

I started out with an idea of running the port directly on a hardware device. And there is a solution on the marked called a Beagle Board, which is a hardware development board based on the Omap3 solution from Texas Instruments ™. The Omap3 utilizes the ARM Cortex A8 processor, runs with 256MB of RAM, and uses u-boot as a bootloader. Testing a port directly on hardware is not a good idea though. There are issues with the speed of the constant rebooting of the hardware and the constant transferring of software to the board. Therefore I also needed an emulator to run the port on.

Based on the close ties between the ARM processor and embedded devices you cannot get around that the support is greater on emulators, bootloaders etc. that target embedded devices. So to get a solution that quickly can be configured (and still support the x86 processor architecture) and booted up I will focus on what the embedded community has to offer. Here I found several emulators, but remembering that I chose U-boot as bootloader I need to respect this while choosing the emulator. And with all this in mind, only one emulator really fits the bill. I found QEMU[17], which supports u-boot and the ARM Cortex A8 processor architecture. There is only experimental support for the emulation of the hardware development board called Beagle board. This did not discourage me, and I chose to run with QEMU due to the fact of the huge support for u-boot and the Cortex A8 processor.

Due to the lack of proper support on the QEMU Beagle Board version, I decided to chose the configuration of the Versatile development board, with a Cortex A8 processor architecture. The only hurdles I experienced with this configuration was that the memory area are different from that of the Beagle board. This is manageable in code, and beyond the time used to read up on this configuration, it has not posed any problem.

When switching from Beagle development board to the Versatile development board running on QEMU, the idea of getting the port tested on the purchased hardware Beagle development board is postponed until a complete version of the port is able to run on the emulated Versatile development board.

4.2 Kernel initialization

When starting this process I switched between configurations of the Beagle development board and the Versatile development board on QEMU. This resulted in different startup addresses in each configuration. As I discovered that I was not going to test the port on hardware just yet, I chose to utilize the address space in the Versatile board. I start by loading the kernel in at 64MB (PA). I do this to be sure that I don't overwrite any kernel parameters stored by u-boot. As the port matures, this address will be set to fit more appropriate. Thereafter I have chosen to test the port by utilizing the smallest amount of main memory possible for the Versatile development board, namely 128MB of memory.

Now it's time to start initializing the CPU. When starting up the processor you want to make sure that things happen in a certain order, otherwise everything might fail to run. The steps to go through when initializing the CPU:

- Make sure that processor is in Supervisor mode
- Disable Interrupts
- Disable MMU, Caches and Buffers
- Flush TLBs and invalidate data cache
- Setup address space
- Set domain access to full access
- Set the Stack Pointer
- Call C code to create the page tables and setup addresses
- Enable MMU

We start out by making sure that the processor is in supervisor mode. We need to make sure that the CPU is running in a privilege mode because if it started in user mode, there wouldn't be a way to enter supervisor mode (because that's exactly what the code running in user mode is not allowed to do), and there will be registers that we cannot reach. The next step is to disable the interrupt routines to make sure that the CPU doesn't react to interrupts. When starting up the CPU we haven't implemented any functionality to catch and react to interrupts yet, and therefore we disable them at first boot up. The next step is to disable the MMU, buffers and caches. It goes without saying that it is necessary to disable the MMU etc. before you can setup the translation system

that utilizes it. The next step is to flush and invalidate caches. I do this because I believe that the TLB entries are invalid the first time the CPU starts up, and I deal with this by completely flushing the TLB.

But before you can go and realize the next steps (setup the address space) in code, there are some 1138 pages of Technical Reference[18] to look through, and I urge you to look at it in order to get all the bits and bytes up on working. It is necessary to make some design decisions regarding how the access permissions should be distributed on the page-tables. In the following I will sketch up the design I decided to have.

First, I need to figure out how the address space should be organized, and on the Cortex A8 architecture this is controlled by the CP15 register 2 - The translation Table Base, which consists of two registers called TTBR0 and TTBR1, and a control register called TTBC. In my case I decide to have a segmentation of 0-1GB the lower address space should go to user space (TTBR0), and 1GB - 4GB, the upper address space should go to the kernel (TTBR1). This is done by setting the TTBC register to the value 2 (N). This effectively says that we are looking at the upper two bit of an address to determine if it belongs to user space or kernel space, see figure 4.1.

```
N = 2

0b00xxxxxx 00000000 00000000 00000000 = 0GB => TTBR0 User space

0b01xxxxxx 00000000 00000000 00000000 = 1GB => TTBR1 Kernel space

0b10xxxxxx 00000000 00000000 00000000 = 2GB => TTBR1 Kernel space

0b11xxxxxx 00000000 00000000 00000000 = 3GB => TTBR1 Kernel space
```

Figure 4.1: Looking at the two upper bits to determine the address space

If we had chosen to set TTBC to the value 1, we would have had the situation shown in figure 4.2 where we are looking at the upper bit to determine the address is user space or kernel space.

```
N=1

0b0xxxxxxx 00000000 00000000 00000000 = 0GB – 2GB => TTBR0 User space
0b1xxxxxxx 00000000 00000000 00000000 = 2GB – 4GB => TTBR1 Kernel space
```

Figure 4.2: Looking at the upper bit to determine the address space

Then it's time to look at how the page table should be organized, and I have quite a few options to go through. To get things up and running I will make

the configuration easy and clear by sticking to the absolute necessary only.

In general:

- I don't implement sections
  - Sections are "pages" of the size 1MB, and they are supported to allow the mappings of large regions of memory while using only a single entry in the TLB.
- I implement small pages (4KB)
- I use one domain, namely domain 0
  - A domain is a collection of memory regions. There are 16 domains. Each page table- and TLB-entry has a field specifying what domain it resides in.
- Memory region type is; Normal, Outer and inner non-cacheable and non-shareable.

For first-level page-tables:

- Since I don't implement sections, the first-level page-table´s function is to act as a pointer to the corresponding second-level page-table.
- The first-level page-table must be aligned on a 16KB boundary.

Second-level page-tables:

- Use small page translations (4KB)
- The second-level page-table´s must be aligned on a 1KB boundary.

To give a better overview, I will illustrate the bit pattern to a first-level page-table in figure 4.3.

Bits [1:0] indicates that the first-level page is a pointer to the second-level page-table, and I set this value to be 0x01.

Bits [8:5] These bits are all set to zero to utilize domain 0 and this requires that you set the co-processor register 3: domain access control - to allow access to domain 0.

Figure 4.3: The bit pattern of the first-level page-table

```
MCR P15, 0, #0x11, C3, C0, 0
```

The value #0x11 gives manager rights to the domain, meaning that access is not checked against access permission bits in the TLB entry, and as a result, no permission faults can be generated.

These tweaks gives us the bit pattern 0x01 which we set on every relevant (not zero) addresses in our first-level page-table. There is no difference in regards to if the first-level page-table is located in kernel space or user space; the bit pattern remains the same.

Now on figure 4.4. I will illustrate the bit pattern for a second-level page-table.

Figure 4.4: The bit pattern of the second-level page-table

Bits [1:0] Indicates the type of table descriptor, and here I use small pages. The XN bit indicate if the region is able to execute code (XN = 0) or not (XN =1).

Bits [3:2] indicate whether the page- table is cacheable and bufferable. Page-table formats use five bits to encode the memory region type. These are TEX[8:6](Type extension field) and the C and B bits.

Bits [5:4] Sets the access permissions, and in my case I chose AP = 01 for kernel space (It runs in privileged mode). When AP = 01 it gives access to Read/write for privilege modes. I chose to set AP = 11 for user space which gives full access for read/writes (for both user and kernel). These bits don't control the right to execute, as this is done by the XN bit [0].

Bits [11:6] Is used for additional access control functions.

Bits[8:6] The TEX bits (along with the C and B bits) are used to set the memory region type to; Normal (TEX = 001), Outer and inner non-cacheable (C and B = 0) and non-shareable (S bit = 0).

Bits [9] The APX bits are also used as access permissions bits, and work in conjunction with the AP bits. This meaning that when APX = 0 and AP = 01 it gives access to Read/write for privilege modes etc. (for further information see[18]).

Bit [10] bit S determines if the memory region is shared (1), or not-shared (0). In my case S=0

Bit [11] The nG bit (not-global) determines whether the translation should be marked as global (0), or process specific (1) in the TLB. In my case nG=0

This leaves me with different patterns depending on the situation:

- For kernel space page-tables with code - [31:12] = xxxxxxxxxxxxxxxxxxxx [11:0] = 000001010010
  - Allow executable, privilege Read/Write.
- For kernel space page-tables with data - [31:12]= xxxxxxxxxxxxxxxxxxxx [11:0] 000001010011
  - Disallow executable, privilege Read/Write.
- For User space page-tables with code - [31:12]= xxxxxxxxxxxxxxxxxxxx [11:0] 000001110010
  - Allow executable, full access Read/write.
- For User space page-tables with data - [31:12]= xxxxxxxxxxxxxxxxxxxx [11:0] 000001110011
  - Disallow executable, full access Read/write.

After this "design" consideration, it is time to set up the stack pointer. The reason for doing this, is that I have chosen to implement the page tables I C code. Before being able to utilize the C language you need a stack (in fact, all C really needs is the stack pointer and nothing more) to store variables. Even if you don't use variables in your code, C still needs the stack pointer (stack) to

function. Hereafter the C code sets up the first- and second-level page-tables. The setup is done by mapping the entire 128MB of physical memory into the 0GB - 1GB user space virtual memory area, and into the 1GB - 4GB kernel space virtual memory area. This setup is shown on figure 4.5. Please keep in mind that I load the kernel into physical memory address 64MB and upwards, when looking at figure 4.5.

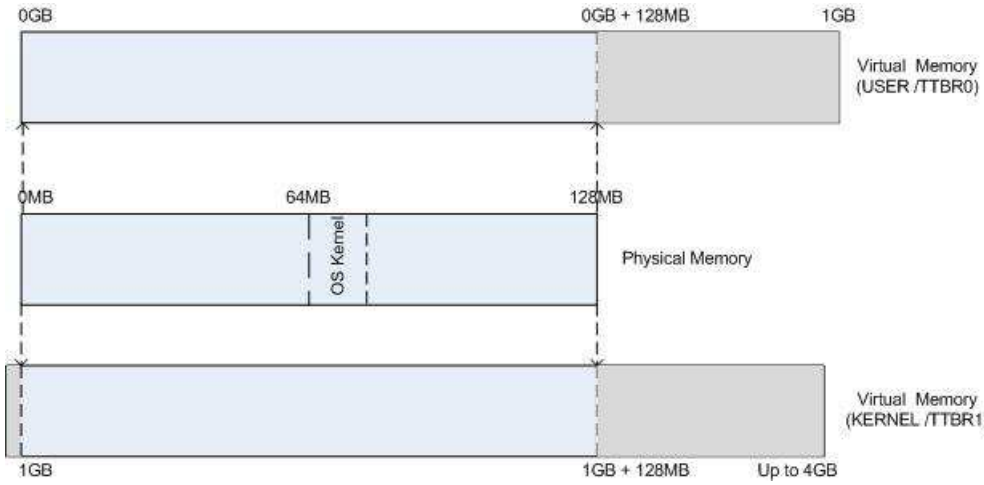

Figure 4.5: The mapping of physical memory to virtual memory

When setting up the page-tables there are some things to notice; First-level page-tables for kernel space has to be aligned on a 16KB boundary, First-level page-tables for users pace has to be aligned on a 4KB boundary, and second-level page-tables has to be aligned on a 1KB boundary.

After the mapping is established between physical memory and virtual memory, it's time to enable the MMU, to get the virtual memory system up and running. The instant that the MMU is enabled, it is important that the code has identical virtual and physical addresses. Now that the MMU is enabled, you could think that the kernel is running in the kernel space region, but it is not. No addresses have been changed, and therefore the kernel still runs in user space, see figure 4.6.

On the figure 4.7 I give an overview of how the virtual memory system is build up from our design.

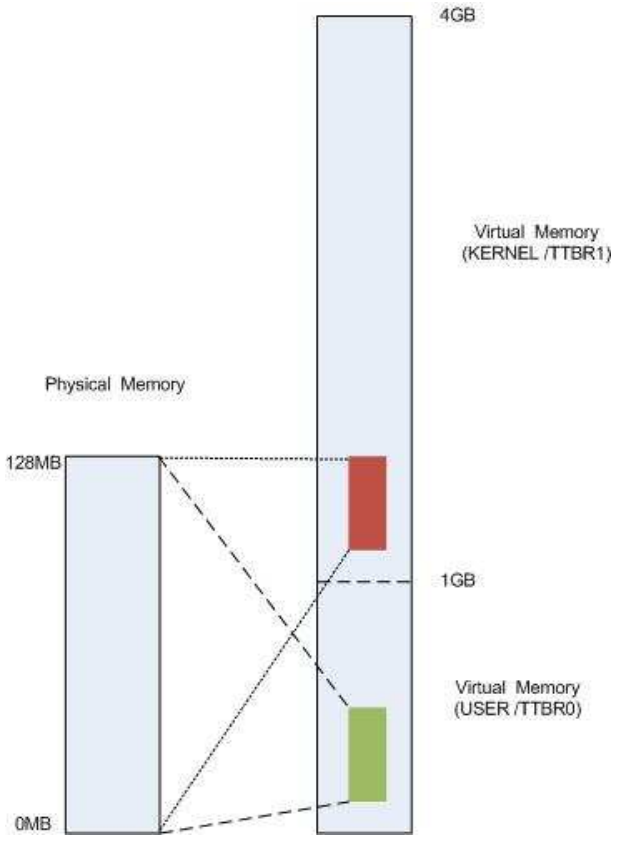

Figure 4.6: Kernel still executes in userspace (green area) after the MMU is enabled.

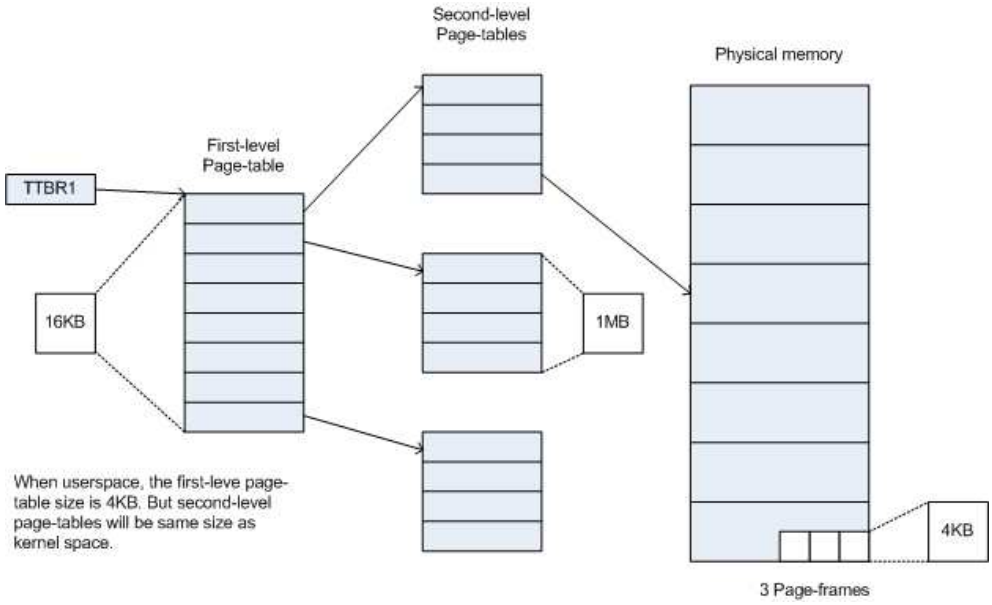

Figure 4.7: An overview of the virtual memory system; kernel page-tables.

4.3 Analysis of Further Steps

In this section I give an analysis of the further steps to get the kernel up and running. I have not implemented this part in code due to reasons explained in section 7.5.

First of all, I need to switch from running in user space to start running in kernel space. At the current time, my kernel is mapped at 64B in user space, and at 1GB+64MB in kernel space, and I would like to make a jump to the kernel space. I would choose to place the kernel where I mapped the corresponding physical address, see figure 4.8. There are multiple ways to do this mapping, but I would chose to do it, so that the current image I have (my ELF binary image), would contain this new image (meaning I would embed an ELF image into another ELF image). I would then extract the ELF-Headers from this new image, and jump to the address which is specified as an offset address in the ELF-headers, see figure 4.9

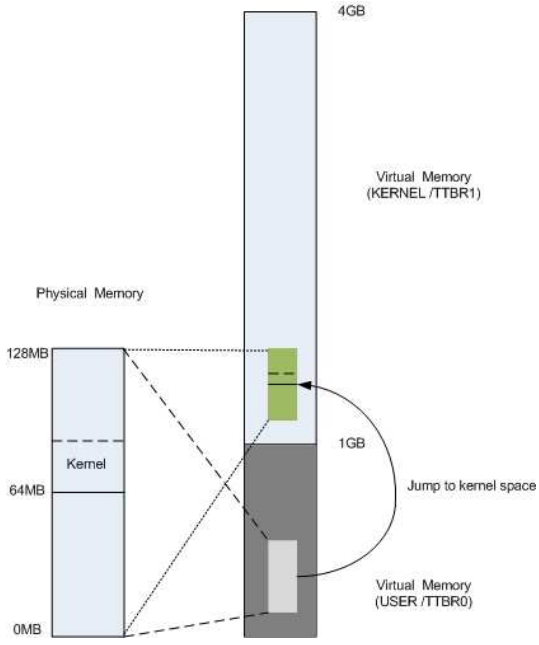

Figure 4.8: Memory layout of a jump from user space to kernel space.

When jumping from one ELF image to another, you must be aware that the old ELF image knows nothing of any possible labels or values of the new ELF image being jumped to. The new ELF image is nothing more than a piece of data to the old ELF image. In the old ELF image (ELF image one on figure 4.9) I have set up a stack for the C environment to use, but this cannot be used in the new ELF image (ELF image two on figure 4.9), and to avoid that the system crashes, I would create a new assembly code. I would then jump from the old assembly code (kernel initialization) which resides in user space, to a new assembly code located in kernel space, and setup a new stack here for the environment to utilize.

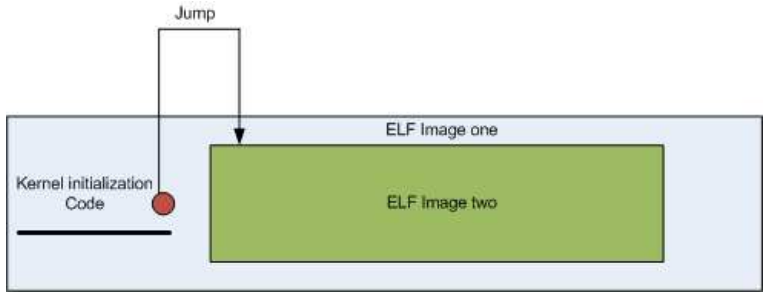

Figure 4.9: ELF image embedded in another ELF image, and the jump from initialization code in ELF image one to the start of ELF image two.

After the kernel is executing in its final location, it would be time to update the page tables to reflect that the instructions (TEXT segment) in the kernel would only be readable and executable. We would never want to change anything in the TEXT segment of the kernel. The page tables should also reflect that the RODATA (read only data) segment, would only be read-only. Everything else in should be regarded as being read and writable (This applies only to supervisor mode, as nothing should be accessible from users pace, at least not pr. Default). Hereafter I would discard the user space mapping, as this is without relevance until user space processes are run.

Now that we have the kernel running in kernel space, it's time to look at setting up the interrupt/exception routines. On the ARM processor architecture, there are different exceptions each with a priority stating who can trump and who cannot. The exceptions, where there are located in memory and in order of priority:

- Reset

  - Located at physical memory address 0x00000000
  - FIQ and IRQ are disabled

- Data abort
  - The software is trying to read/write an illegal memory location
  - Located at physical memory address 0x00000010
  - IRQ are disabled

- FIQ
  - If developers wants to handle certain interrupts faster
  - Located at physical memory address 0x0000001C
  - IRQ and FIQ are disabled (By default ARM cores do not handle nested interrupts)

- IRQ
  - Also called "Normal" interrupts. Triggered by hardware through the interrupt controller
  - Located at physical memory address 0x00000018
  - Disabled IRQ

- Prefetch abort
  - If processor reads instructions from undefined memory. Occurs when instruction reaches execution stage of pipeline
  - Located at physical memory address 0x0000000C
  - Disabled IRQ

- Undefined instruction
  - ARM waits for co-processor to acknowledge that it can execute instruction, when executing co-processor instructions
  - Located at physical memory address 0x00000004

- Software interrupt (SWI)
  - SWI interrupt is used to enter supervisor mode, to execute a privileged kernel function
  - Execution of any system call will cause a SWI to change to supervisor mode.
  - Located at physical memory address 0x00000008

On the ARM Cortex A8 processor architecture there is a way to utilize other addresses then the ones stated above. These addresses are called high vector addresses and start with 0xFFFFxxxx where the xxxx are the same low bits as in the addresses stated above e.g. SWI is located at 0xFFFF0008. The choice of using high vector addresses or normal addresses are determined / set by bit [13] of coprocessor 15 register 1, where 0 = Normal exception vectors selected and 1 = High exception vectors selected. There is support to define your own mechanism, to determine interrupt vectors by setting bit [24] of co-processor 15 register 1 = 1. I will not cover this here. By setting bit [24] of co-processor 15 register 1 = 0, Interrupt vectors are fixed.

In general what needs to be done to get interrupts up and running is to create a stack for all the seven exception modes mentioned earlier. Create interrupt handlers in the kernel code to handle the exceptions thrown by the system (or to handle the system calls). Setup the addresses for the kernel interrupt handlers in the respective vector addresses. This meaning that the vector addresses should point to their equivalent interrupt handlers residing in the kernel code, see figure 4.10.

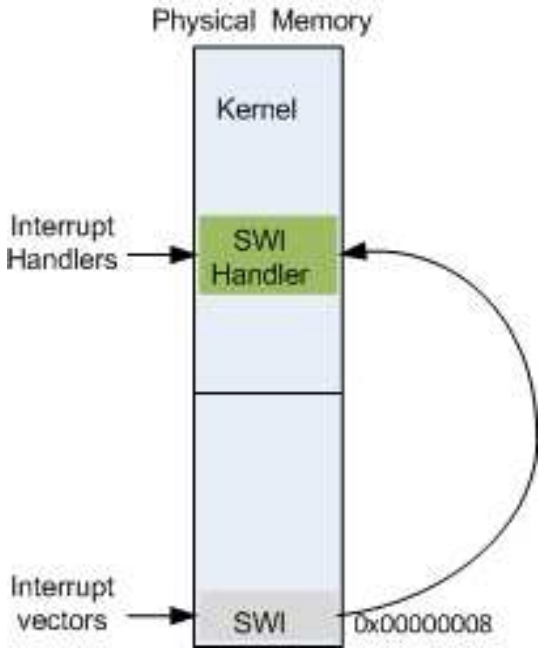

Figure 4.10: SWI interrupt vector pointing to the SWI interrupt handler in the kernel.

Each interrupt handler must on its own implement the steps to change to the corresponding stack. This meaning that the IRQ handler must implement the step to change from user process stack to the IRQ_stack.

The action conducted on a Software interrupt:

- R14_svc is set to the address of the next instruction to execute after the SWI
- SPSR_irq is set to the Current Program Status Register (CPSR)
  - Each exception mode has a Saved Program Status Register (SPSR) which holds the CPSR of the task immediately before the exception occurred
- CPSR bit [4:0] is set to 0b10011 which indicate a switch to supervisor mode
- CPRR bit [5] is set to 0 - to make sure that we execute in ARM mode and not (Thumb mode)
- CPSR bit [6] is unchanged - the FIQ bit
- CPSR bit [7] is set to 1 - this disables further normal interrupts (IRQ)
- PC (program counter) is set to point to 0x00000008

In the case of the IRQ interrupt handler, it must implement a list of Interrupt Request numbers (from 0 -15) that points to functions further up in the FenixOS kernel code, to give notice that an interrupt has occurred (a thing here to consider is that the device drivers reside in user space in FenixOS, and normally an Interrupt makes a call switching to supervisor mode. This must be dealt with).

When a system call is made, it is essentially a software interrupt that is conducted. The basic steps to conduct for a system call can be seen on figure 4.11.

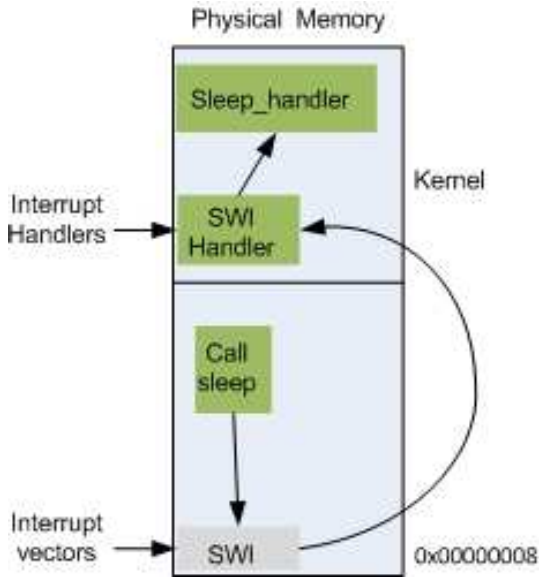

Figure 4.11: The steps for conducting a system call.

Basically the user space process calls into libc/EGlibc/GLibc or equivalent API, and the API then calls the SWI at address 0x00000008 with a value E.g. 12 (let's say that 12 = the system call sleep) . The SWI jumps to the kernel address of the SWI_Handler function, which changes to SWI stack, stores the state of the process and starts executing the kernel code for the handler (this could be a call further up in FenixOS to sleep_handler that executes the code for the sleep system call).

For implementing processes and threads in FenixOS the context switch must be implemented. This will reality mean at least preparing a new process context and switching between two existing process contexts.

For preparing a new process context the stack must be allocated and initialized with proper values to be loaded into the registers. The layout can have the form as can be seen on figure 4.12

The idea during initialization is to ensure that the stack pointer is pointing to the top of the stack and the PC will be set to run at the process initialization point, e.g. the main function.

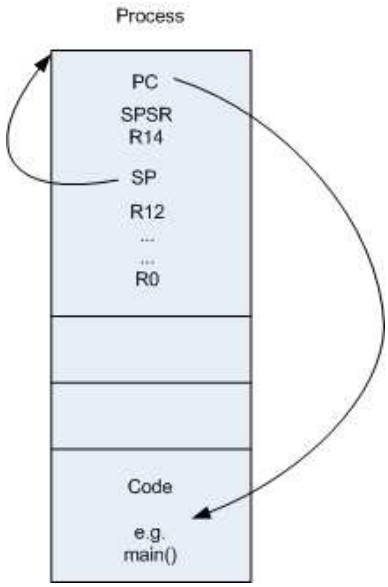

Figure 4.12: Initializing the process stack.

In order to perform the process context switch a new (or existing) process context must be available.

The process context switcher performs the action of storing all registers from user space onto the user space stack available for the user process. It will then load all the registers from the next process user space stack.

The actual switching function is typically performed due to some event, e.g. timer interrupt. At that instant the old process 1 is frozen and we enter privileged mode. Normally when the interrupt is finished the old user space context is restored.

However in this example the kernel decides to change user space process, i.e. Perform a process context switch. The old and new process are determined and the idea now is to exchange the return process context when we return from privileged mode to user space. Thus the actual process context switch will happen when we return from privileged mode to user space see figure 4.13.

In order to exchange the process context we store the original process context in a reserved space. Typically the user space stack is used because it is available and uniquely identified with the process. All registers are stored to the stack.

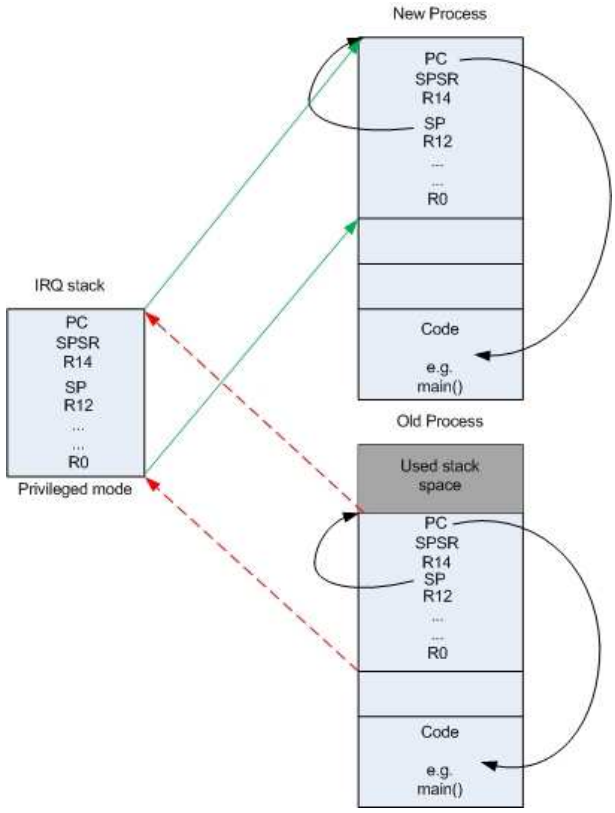

Figure 4.13: The switch of context between to processes.

The the next process context is found and loaded from the next user space stack into the registers. Finally the return to user space will activate the next process.

Chapter 5

Conclusion

Porting a research operating system is not an easy task. Throughout this thesis I have learned much about operating systems that I did not know before. I have successfully analyzed which processor was suited for the future in embedded and importantly in the netbook domain. I have successfully analyzed the necessity of a new operating system design, and found this design to fit the research operating system FenixOS. I have completed the analysis and implementation of; initialization of the kernel (pre-steps after first boot), getting the virtual manager up and running (enabled MMU), and a full analysis of the further needed steps to implement IRQ/FIQ, SWI/System calls, and context switches (process and processor mode). The latter parts are not complete. Some of it is implemented but does not yet work properly, and therefore I chose to state that it is not implemented. I have confidence that the work I have done is a splendid piece of work, and I hope to be able to continue this work henceforth.

When I started this project, I quickly learned new things about myself. I found that I needed to learn ho to search for resources properly. I also found that I have gained an enormous increase in knowledge regarding the subjects at hand, and that I'm not despondent when it comes to facing tough times. All in all this project have given me a lot to take further into the working life, that I hopefully soon will be part of. With regards to the project management part I have learned the benifit of a proper milestone plan combined with an iterative management model.

Chapter 6

Future Work

The implementation I have completed is only a draft design for how the initialization code of the kernel (memory system etc.) should be. A fully working kernel running user processes in user space and the kernel in kernel space, switching context, having system calls and a complete working interrupt system routine is not something you analyze and implement over the course of 20 weeks. The projects I've searched on the Internet are stating "man-years" on such project, and not "man-hours". The next steps should be along the lines of getting a firm base kernel up and running, and then re-visit the boot process, to make sure that the addresses chosen for startup are the ones wanted in the future. For the FenixOS memory manager the interface needs to be clearly defined and the hardware abstraction layer must be implemented along the same lines as used in this report for initializing the kernel virtual memory manager.

The next phases of the project should indeed try to focus on getting the kernel up and running on a hardware platform such as the Beagle Development board. This does also enforce the notion of generalize the code to have multiple choices of startup addresses.

Chapter 7

Project planning

In this chapter I give an insight to how I planned my project from the beginning (February 25, 2010) and till the end (June 23, 2010). Besides The Timeplan, Milestone plan and Risk analysis I'll give my thoughts and experiences on the planning of the project from the various iterations throughout the duration of my thesis. I've been trying to follow materials on project management [19]

7.1 Timeplan

I completed this thesis with a timeplan shared among my thesis, a course in assembly programming and a course in Network Security.

Timebox:

- There is 117 days devoted to this project. This is not negotiable.
- 2 days a week goes to other courses taken in parallel with this thesis.
- This gives 85 working days for the thesis project.

All in all, I have planned the thesis to last for 81 days, when accounting for 4 sickdays.

7.2 Milestones

Milestones:

- Read and understand resource material (After 1 month)
- Write Chapter 2 - Problem statement (After 1 month)
- Setup Build environment - QEMU, Compiler etc. (After 2 months)
- Write Chapter 3 - Related Work / Background (After 2 months)
- Bootloader - U-boot (After $2\frac{1}{2}$ months)
- Write Chapter 4 - Design and Implementation (After 3 months)
- Kernel initialization code (After $3\frac{1}{2}$ months)
- Stakeholder perspective on the kernel part - Be able to jump to C-code from the kernel (until 15/6-2010)
- Write missing parts of the report (Until 20/6-2010)

7.3 Risk analysis

The risk analysis shown below contain two tables; table 7.1 and table 7.2

ID:	Description:	Probability (1-5):	Impact (1-5):	Impact reduction:	If impact occurs:
1	Family (Sickness from children and family)	2	3	N/A	Get help from friends
2	Reading resource material	3	4	Keep regular contact with my supervisor	Redefine scope
3	Hardware development board	3	1	Utilize "known" and well tested hardware solutions	Make use of an emulator to emulate the hardware needed to run software
4	Third party software (QEMU)	3	3	Utilize configurations that are well tested	Find new emulation software
5	Third party software (U-boot)	2	4	Utilize well known configurations that are tested	Build own bootloader / redefine scope

Table 7.1: Risk analysis created February 25, 2010

ID:	Description:	Probability (1-5):	Impact (1-5):	Impact reduction:	If impact occurs:
1	Family (Sickness from children and family)	2	3	N/A	Get help from friends
2	Reading resource material (DONE)	(1)	(1)	Keep regular contact with my supervisor	Redefine scope
3	Hardware development board (DONE)	(3)	(1)	Utilize "known" and well tested hardware solutions	Make use of an emulator to emulate the hardware needed to run software
4	Third party software (QEMU) (DONE)	(2)	(3)	Utilize configurations that are well tested	Find new emulation software
5	Third party software (U-boot) (DONE)	(2)	(4)	Utilize well known configurations that are tested	Build own bootloader / redefine scope
6	Latex report writing tool	3	3	Utilize templates delivered from DTU	Write report in MS Word or Open Office

Table 7.2: Risk analysis created June 1, 2010

7.4 Project iterations

Project iterations (timeboxed, both theory and praxis) Each iteration delivers a product independent of what has taken place in previous iterations.

- Read and understand resource material (After 1 month) + Write Chapter 2 - Problem statement (After 1 month)
- Setup Build environment - QEMU, Compiler etc. (After 2 months) + Write Chapter 3 - Related Work / Background (After 2 months) + Bootloader - U-boot (After $2\frac{1}{2}$ months)
- Write Chapter 4 - Design and Implementation (After 3 months) + Kernel initialization code (After $3\frac{1}{2}$ months)
- Stakeholder perspective on the kernel part - Be able to jump to C-code

from the kernel (until 15/6-2010) + Write missing parts of the report (Until 20/6-2010)

7.5 Process Report

The process report is divided into two sections; a Chronological part, and a Themed part. This is done to give a better overview of the result orientated sections of the Chronological part, from the more soft and highly reflecting of the Themed part.

7.5.1 Chronological part

In this part of the process report I setup lists indicating what goals I had for the iteration, and what I actually achieved in this iteration. If the outcome should be two identical lists in the expected list and in the actual list, a miracle would have happened. These list are seldom in perfect harmony, and therefore they are good indicators of what went wrong if a project exceeds its time limits, budget or a product is not ready for delivery. I end the sections of by reflecting over the progress in the iteration.

7.5.1.1 Establishment phase

Plan - Expected course of events In this phase I need to:

- Identify the scope of the project
- Identify the risk of the project and write a risk analysis
- Identify type of processor needed
- Identify type of bootloader needed
- Identify the resources materials needed to understand the project
- Read and understand the resources discovered for this project
- Start writing the problem statement
- Identify milestones and make project plan
- Create skeleton for the report

Process - Actual course of events After this phase what I actually achieved was:

- Identified some risks of the project
- Wrote a small risk analysis
- Identified the processor to be used
- Identified the bootloader to be used
- Identified some materials to be read
- Read the resource materials discovered
- Created a skeleton for the report

Reflection All in all I have not quite achieved to reach all the goals as I would have wanted to. I simply feel that I have not identified the scope of the project to its fullest. I struggle a bit to search for information on this subject, as it has surprised me to be much more complex than first assumed. This complexity has thrown me of guard, and I tend to seek away from the planning in order to keep up with the large amount of information that has to be uncovered in order to get the project started. The focus of this iteration quickly turned to finding the right processor and bootloader, and searching for information surrounding these two subjects. What I did not plan for was the amount of "information-catch-up" needed to be done. This meaning that although courses on parts of the subject have been completed, the need for recapping this information was great and took a long time.

7.5.1.2 Environment setup phase

Plan - Expected course of events In this phase I need to:

- Identify startup address space on environment
  - On hardware boards and emulation software
  - Bootloader environment
- Identify and decide on pros and cons regarding coding directly on hardware or on an emulator
- Identify a qualified emulator for running the port code

- Chose and setup compiler environment
- Create and compile the environment for the first boot-up
  - In regards to building the bootloader and a small test program to get the ball rolling
  - In regards to get this environment to run on a emulator
- Write the chapter regarding background knowledge for the report
- Identify the resources materials needed to complete this iteration
- Read and understand the resources discovered for this iteration
- Update milestones, risks and project plan

Process - Actual course of events After this phase what I actually achieved was:

- Identified startup address space on environment(s)
- Identified pros and cons regarding emulator and hardware
- Identified a qualified emulator for running the port code
- Chose a compiler for the project solution
- Created the environment to compile and boot up a very small test program on the emulator
- Identified some materials to be read
- Read the resource materials identified

Reflection This phase composed a lot of problems for me. There were many problems trying to get the u-boot environment to work with QEMU. The reason is that the newest versions of u-boot will not work with the "home-patched" version of QEMU, when trying to target the hardware Beagle development board environment. It took too long time to get this environment up and working. I had to target a different hardware development board, in order to get the test code up and running, which completely occupied this iteration. This left me with only sporadic updating the project plan, and not enough time to consume the newly identified reading materials.

7.5.1.3 Construction phase

Plan - Expected course of events In this phase I need to:

- Read and understand materials on ARM assembly language programming
- Read and understand the information given in the ARM Architecture Reference Manual
- Identify necessary steps to startup an ARM processor to initialize the memory environment
- Identify and analyze the necessary design (e.g. in assembly or C code etc.) of the memory system in regards to page-tables etc.
- Identify and analyze the necessary steps to get the kernel initialized and up and running in kernel space (virtual memory)
- Identify and analyze how to get interrupts, system calls, I/O and context switch up and running (after the kernel has reached kernel space)
- Implement the setup code
- Implement the virtual memory system
- Make the kernel code run on QEMU and debug/test that the MMU is able to be enabled
- Write the chapter regarding Design and Implementation for the report
- Identify the resources materials needed to complete this iteration
- Read and understand the resources discovered for this iteration
- Update milestones, risks and project plan

Process - Actual course of events After this phase what I actually achieved was:

- Read the materials on ARM assembly language programming
- Read the virtual memory section and the section on Coprocessor registers in the ARM Architecture Reference Manual
- Identified the necessary steps to startup the ARM processor to initialize the memory environment

- Identified and analyzed the design of the memory system in regards to page-tables etc.
- Identified and analyzed the necessary steps to get the kernel initialized and up and running in kernel space (virtual memory)
- Implemented the setup code
- Implemented parts of the virtual memory system

Reflection I am behind! This is a very clear fact when looking at what should have been, and what has actually been done. This phase was composed of a lot of activities, which depended on things working right away. Things seldom do, but I have come a pretty long way. I have (again) neglected the project management part in favor of doing code. The choices of not updating the project plan properly has forced me to take make a "here-and-now" assessment of the situation, and I have decided to cut the scope of the implementation. At this point I don't have the time to implement; the jump from user space to kernel space in virtual memory, the interrupt routines, system call routines and so on. I will turn my attention to finishing the implementation of the virtual memory setup, and then focus on writing the report, which I have so far neglected to do. The analysis of the virtual memory, the switch to kernel space, the interrupt routines and so forth has taken too long for me to get in place.

7.5.1.4 Documentation phase

Plan - Expected course of events In this phase I need to:

- Implement the jump to kernel space, and setup interrupt routines etc.
- Write missing parts of the report

Process - Actual course of events After this phase what I actually achieved was:

- Implemented parts of the virtual memory system
- Made the kernel code run on QEMU and debug/test that the MMU is able to be enabled
- Updated the project plan, the milestones, and risk analysis

- Wrote the entire report
  - Abstract (and Resumé), Preface, Dedication, Acknowledgement, Introduction, Problem Definition, Background, Theory, Conclusion and Future Work
- Completed and organized the Project Plan, Milestones, Risk Analysis and Process into the report

Reflection This phase should have been all about doing the last implementations and writing missing sections of the report. This was not the case, due to the decisions taken in the preceding iteration. This iteration was all about finishing the implementation of the virtual memory system, and writing the report. The decision of cutting parts of the implementation (but not the analysis) and focusing has left me with time enough to implement the virtual memory system and write the report. The aim from the beginning was to write the report simultaneously with the implementation, but ended up being a long report writing phase.

7.5.2 Themed part

In this section I'll reflect on; my ability as a project manager in this project, the expected plan of the project, the actual plan of the project, and on project as a whole.

7.5.2.1 Project management

I started out with good intentions of running my project using the agile unified process methods. This unfortunately did not turn out as I would have hopped, not blaming the methods used, but more my lack of perspective on some of the aspects, and bad decision makings.

Plan - Expected course of events From the beginning the overall plan was to make a plan that would accommodate the project at hand. I started out by making a short term plan, with the intention of building it out as I identified the scope of the project. I would use four iterations with combined read/write/implement in short and manageable iterations. The aim was to make sure that I would finish the project in due time, with a component ready at the end of each iteration. The aim was also to make sure that I would not sit in the weeks before the end, and have writing nothing.

Process - Actual course of events The course of events did not work out as I had thought in the beginning of the project. I followed the plan I had created, but it quickly turned out that this was not good enough. Therefore I lost valuable time trying figure out what to do next. This fact, and the fact that too late in the project started to update the project plan, milestones etc. did that I lacked implementation of the rest of the kernel.

Reflection When looking in the huge mirror of time, it is easy for me to see, that I fairly quickly lost track of the project. I did not, to a great enough extend, uncover the scope of the project. This left me without enough knowledge to plan correctly further in the project. I also did not in due time consult the project plan to discover what I was missing in the iterations. My planning was not in-depth enough, and I lost perspective in regards as to what should be finished at what time. My iterations were not detailed enough, which would have made it clear to me that things would take longer time than first acknowledged. I did however make a good decision (albeit a little late) to cut off some of the

implementation, so that I would be able to meet my deadline with a product not completely implemented, but somewhat implemented and fully analyzed.

Bibliography

[1] Linux.org - *Information about Linux* - http://www.linux.org/.

[2] YouTube.com - *Lecture 13 Virtual Memory and Memory Management Unit* - http://www.youtube.com/watch?v=TF_KP4SMIfM.

[3] YouTube.com - *Lecture 20 Virtual Memory* - http://www.youtube.com/watch?v=3ye2OXj32DM&feature=channel.

[4] YouTube.com - *Lecture 31 Memory Hierarchy : Virtual Memory* - http://www.youtube.com/watch?v=cIlKSD8ptAk&feature=related .

[5] ABOUT.COM - *What is a CPU* - http://pcsupport.about.com/od/componentprofiles/p/p_cpu.htm .

[6] Wikipedia.org - *Simultaneous multithreading* - http://en.wikipedia.org/wiki/Simultaneous_multithreading.

[7] ARM.com - *ARM Cortex A8 Processor* - http://www.arm.com/products/processors/cortex-a/cortex-a8.php.

[8] Wikipedia.org - *ARM architecture* - http://en.wikipedia.org/wiki/ARM_architecture.

[9] Microsoft Reasearch - *Operating Systems Review, Singularity* - http://research.microsoft.com/en-us/projects/singularity/.

[10] ARM.com - *ARM Cortex A9 Processor* - http://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[11] denx.de - *Home of the U-Boot* - http://www.denx.de/wiki/U-Boot.

[12] wiki.openmoko.org - *The QI bootloader* - `http://wiki.openmoko.org/wiki/Qi`.

[13] zenvoid.org - *Smart-QI* - `http://alone-in-the-light.zenvoid.org/2009/11/smartqi-bootloader.html`.

[14] sourceware.org - *Home of the RedBoot* - `http://sourceware.org/redboot/`.

[15] GNU.org - *GNU GRUB* - `http://www.gnu.org/software/grub/`.

[16] Wikipedia.org - *LILO bootloader* - `http://en.wikipedia.org/wiki/LILO_(boot_loader)`.

[17] QEMU.wiki.org - *Open Source Processor Emulator* - `http://wiki.qemu.org/Main_Page`.

[18] ARM DDI 0100I - *ARM Architecture Reference Manual*, ARM Limited, July 2005.

[19] M.L. Attrup, J.R. Olsson - *Power i projecter & portføljer*, DJØF Forlag, 2008.

[20] David A. Patterson, John L. Hennessey - *Computer Organization and Design 3rd.*, Morgan Kaufmann, 2007.

[21] Andrew S. Tanenbaum, Albert S. Woodhull - *Operating System - Design and Implementation 3rd.*, Pearson Prentice Hall, 2006.