Migration of a Modular Operating System to a Intel Atom Processor

Ladislav Kobza, Martin Vojtko, Tibor Krajčovič
Institute of Computer Systems and Networks
Faculty of Informatics and Information Technologies of STU in Bratislava
Bratislava, Slovakia
e-mail: xkobzal@gmail.com, {martin.vojtko, tibor.krajcovic}@stuba.sk

***Abstract*—This paper summarizes a process of operating system adaptation to an Intel Atom processor. The main objectives of this project was to adapt a simple micro kernel embedded operating system to a more complicated processor family, without destroying the original modules of system or changing their functionality. Our motivation was the lack of information or techniques regarding operating system migration. Based on the analysis we chose to implement a micro kernel for the Intel processor family capable of module loading, user segment isolation and preemptive task switching using Intel specific mechanisms. We also implemented a new module which provides standard output using legacy BIOS functionality, and could be expanded to a fully functional CLI (Command Line Interface). The result of this work is a bootable, fully functional micro kernel which loads all given modules after its initialization phase. This micro kernel is separated from the platform independent parts and easily manageable as it is written mainly in C. This work is a simple guide into developing an own micro kernel for the x86 processor family and using its basic functionality.**

Keywords-operating system; micro kernel; operating system development; operating system migration

I. INTRODUCTION

This work is based on the task of porting a chosen simple modular operating system written for an ARM based processor to an x86 based platform, while keeping the most of the original implementation.

This paper will guide the reader through the process of analyzing the original operating system, mentioning the critical parts, we had to change and implement from scratch, mainly focusing on the task switching procedure.

Based on the analysis of the original Modular Operating System [1] (in short MOS) code, we had specified the platform dependent parts, as the initialization procedure, task switching method and task management. Not all of these mentioned parts could be easily ported, the initialization procedure of MOS was replaced by using GRUB (GRand Unified Bootloader), task switching was reimplemented using Intel specific methods – the TSS (Task Switch Segment – discussed later in this article), the memory and task manager were modified so that they can work with a linear memory mapping and modified task headers. We had to write platform dependent parts of a micro kernel from scratch, starting at the GDT (Global Descriptor Table), which is used for memory access and holds the TSS, also creating the IDT (Interrupt Descriptor Table) responsible for interrupt and exception mapping and handling and we used it also for setting up the user protected mode.

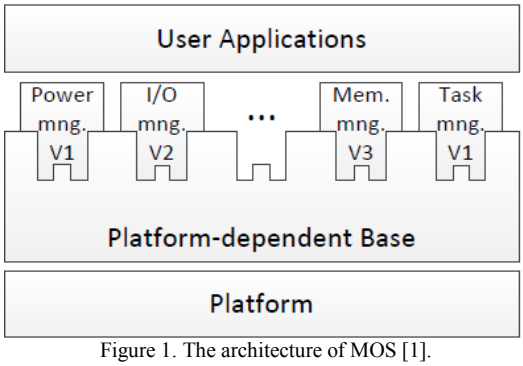

Figure 1. The architecture of MOS [1].

The result of this work is a prototype of the ported operating system micro kernel on an Intel Atom processor with some of the original modules adapted and some new added.

At the end of this article, we would like to mention some proposals for implementation improvements and new modules if anyone would like to pick up where we left.

II. ANALYZING THE ORIGINAL MOS

The MOS is a simple case study of operating system concepts. It is a micro kernel OS managing and planning tasks and memory of an embedded system. Architecture of this operating system allows its simple adaptation and extension.

The Kernel of the MOS consists of two layers (Figure 1). The first one is a platform-dependent base layer (in short PDBL) and consists of pieces of assembly code which encapsulates hardware into a higher abstraction layer [1].

This was the main part we had to rewrite and implement from scratch. The second one is platform-independent modular layer (in short PIML) and encapsulates the PDBL into a presentation layer (see Figure 1). The PIML consists of modules which are connected to PDBL by customizable connectors.

The compulsory modules of the MOS are the Process (Task) management and the Memory management. These two modules are the core of MOS and they provide the basic functions of the whole operating system. Other modules of operating system can be implemented with respect to platform devices and system needs.

III. THE MIGRATION PROCESS

We have identified most of the critical steps based on those facts and the Intel specifications [2].

First, we had to create a micro kernel, which needs to have the following functionality:

1) to initialize our system,
2) pass the memory boundaries to the memory manager,
3) switch tasks in a preemptive manner,
4) divide the tasks to user and kernel specific, and
5) be able to interact with the user.

The first step (after booting up the computer) is to load the processor into 32 bit protected mode and boot our new kernel. We outsourced this procedure using GRUB, which makes this instead of us.

In order to initialize a kernel after booting, one has to set up a GDT (Global Descriptor Table) which is responsible for memory access, an IDT (Interrupt Descriptor Table) responsible for interrupt, exception and (in our case) user system calls, and a TSS (Task Switch Segment) which is the Intel specific structure for task switching and will be discussed later.

To pass the available memory to the memory manager, we have to map it first, in our advantage GRUB does this for us using specific flags in his magic fields and a structure, which we can scan for a large enough piece of memory to pass [3].

As we need preemptive task switching, we want to initialize the PIT (Programmable Interrupt Timer) which will fire an interrupt in an interval which is specified in OS configuration file, and an interrupt handler which will call the task switching procedure from the original MOS.

IV. TASK SWITCHING

The task switching procedure is completely written in assembly code and so is platform specific. It consists of pushing the context of the current task (Task header) to the stack, setting the segment registers to kernel and calling an external function *switchContext*. This function takes the stack top as an argument and returns the new stack top, which we then move to the *esp* register. All we need to do then is to load the context from the new task stack and jump to the address of the last instruction.

As we needed to implement user-protected calls, we used the *TSS* to hold the *esp0/eip0* (kernel stack and instruction pointers) and *esp3/eip3* (user stack and instruction pointers). These registers are filled during the switch context procedure and the user mode is started using a set of pre-prepared functions, which trigger a specific interrupt that calls a user code handler. This handler sets the segment register to the value of the User data *GDT* descriptor, prepares the stack to jump into user mode and jumps. After the user mode jump is finished, the original function arguments are passed to their kernel equivalents, processed and after that the user mode ends returning to the last executed kernel procedure.

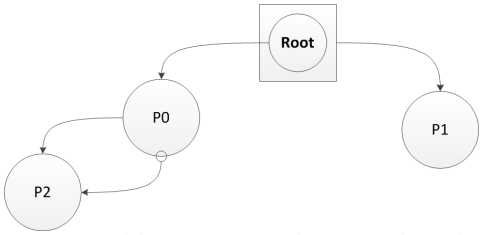

Figure 2. Structure of the process tree. Where P0 sends synchronizing messages to P2 and P2 waits until the message will come.

V. TESTING

The testing itself was performed on an Intel Desktop Board D945GSEJT with an Intel Atom N270 CPU.

The test performed, was a priority planning test, which consisted of spawning two processes P0 and P1 from the MOS root process. In the next step, the P0 process spawned a child process P2. The MOS root process was only used for statistical purposes (see Figure 2). The testing processes consisted of a simple mathematical operation and in the case of P0 and P2 synchronized inter-process communication was included.

VI. CONCLUSIONS

This work brought us many new experiences in the field of kernel development and it´s hardest part was information gathering on how to get started. We had to define steps for operating system adaptation which allowed us successfully complete migration.

In conclusion, we have successfully migrated the given MOS to the x86 platform including some changes in the implementation specific modules. It is a good example of usage extremely simple operating system on complicated processor architecture in comparison with operating system such is Linux. The solution can be used on projects where the operating system can be simple and the overhead of the OS has to be small.

As mentioned before, there is much more work to do, the next logical steps would include implementing other device modules of the platform. The comparison to other operating systems has to be done but only to comparable operating systems such are freeRTOS or AVRx.

ACKNOWLEDGMENT

This work was supported by the Slovak Research and Development Agency under the contract No. VEGA 1/1008/12.

REFERENCES

[1] Vojtko, M.; Krajčovič, T., "Prototype of Modular Operating System for embedded applications," *Applied Electronics (AE), 2013 International Conference on* , vol., no., pp.1,4, 10-12 Sept. 2013.

[2] Intel, “Intel 64 and IA-32 Architectures Software Developer´s Manual – Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C”, 2013.

[3] Free Software Foundation, Inc, “Multiboot Specification”, 2009.