Internal Report 2012–21 August 2012

Universiteit Leiden

Opleiding Informatica

Porting S.M.A.C.K to the x86 architecture

Matthijs van Drunen

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Porting S.M.A.C.K to the x86 architecture

Matthijs van Drunen

August 30, 2012

Supervisors:
Prof. Dr. H.A.G. Wijshoff
Mattias Holm, MSc
Kristian F. D. Rietveld, MSc

Universiteit Leiden
The Netherlands

Abstract

The S.M.A.C.K operating system is used in the Operating Systems course at LIACS. Running natively on the BeagleBoard, it is used by students to familiarize themselves with different aspects of operating systems. This thesis examines the portability of S.M.A.C.K by discussing the process of porting S.M.A.C.K to the x86 architecture. It introduces the x86 architecture, shows that most parts of S.M.A.C.K are portable and provides recommendations for increasing the portability of the parts that could not be shared between platforms.

Contents

1 Introduction

S.M.A.C.K is an operating system that is used in the Operating Systems course at LIACS. Students are supposed to implement parts of a kernel that are usually found in an operating system, such as process scheduling and physical memory management. The kernel runs natively on the BeagleBoard[2], which contains an ARM CPU.

Porting is the process of adapting software to run in a different environment than the one it was originally written for. Generic user applications can be adapted to run using a different set of libraries, or on another platform. Portability is a measure for how much effort it takes to port software. An operating system kernel needs to have knowledge about how the hardware that it runs on works and how it should configure the hardware, because user applications depend on it to provide an abstraction layer for different services, such as I/O. This means that a kernel cannot simply be recompiled for another platform without changing the source code to include support for this platform. A kernel also contains code that is not specific to one platform, such as code to manage processes. For operating systems, portability is a measure for how much of the code can be re-used or shared between platforms the kernel runs on.

Portability is also a software quality factor ([1]). Software that is portable usually depends on a separate layer to provide an abstraction for the platform it is running on. This makes code easier to understand and maintain. Other advantages of having an operating system that runs on different platforms include the ability to study different architectures and make the kernel more accessible to people who do not have access to a BeagleBoard.

The goal of this project was to port S.M.A.C.K to another architecture. One of S.M.A.C.K's requirements was to be portable. This project investigates whether it is. The x86 architecture was chosen because hardware based on this architecture is easily available to students, the 32-bit version was chosen because it was easier to implement support for virtual memory for the 32-bit version. The aim was to get the kernel and a subset of the user applications running on an emulator and on at least one physical machine. The kernel was supposed to boot successfully and start the shell program. It was also supposed to be able to run the `mmaptest` program that is used to test the implementation of the third assignment ([3]) of the Operating Systems course at LIACS. The `mmaptest` program can be used as a test case for the virtual memory system. To limit the scope, the kernel did not have to be able to draw graphics to the screen or run the `video-ps` program that draws a visual representation of running processes.

In this thesis, the process of porting the kernel is discussed, as well as some of the problems that have been encountered and how they have been resolved. This thesis is structured as follows. Section 2 provides an introduction to the x86 architecture. Section 3 describes the general approach that was used to port the kernel. Section 4 describes the implementation of specific tasks or modules of the kernel. The final section discusses things that can be improved in future versions.

2 Introduction to the x86 architecture

X86, formally known as IA-32, is the instruction set architecture (ISA) that was first implemented by the Intel 80386 processor. It is a 32-bit extension of the original x86 architecture. It was later extended to 64-bit, known as x86-64. S.M.A.C.K only uses the default IA-32 architecture, without any extensions. Every Intel/AMD CPU that implements x86 is backwards compatible with previously released CPUs. An x86 CPU will generally start in 16-bit mode, and should be able to run legacy software that was written for earlier CPUs. The x86 architecture supports features that modern operating systems rely on to function, such as paging and a flat memory model.

The CPU can operate in the following modes:

- **Protected mode**. The native state of the CPU. The CPU can execute 16-bit or 32-bit instructions in this mode, depending on the selected code segment and the instruction. Paging is an optional feature. S.M.A.C.K uses protected mode with paging enabled.
- **Real-address mode**. Legacy 16-bit mode. This is the initial mode of the CPU. The boot loader is responsible for switching to protected mode.
- **System management mode (SMM)**. This mode can be used for implementing platform-specific functions, such as power management. This mode is activated by the external SMM interrupt pin or when an SMI[1] is received from the interrupt controller.

The x86 architecture offers the following registers:

- General-purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.
- Segment selector registers: CS, DS, SS, ES, FS, GS.
- Program status and control register: EFLAGS.
- Instruction pointer: EIP.

2.1 Memory model

To ensure that multiple applications can run simultaneously, each must have its own view of the memory, so that it does not interfere with the memory of other applications. This technique is called virtual memory. Segmentation and paging are two ways to implement virtual memory. The x86 architecture uses segmentation by default, but it can also use paging. In real-address mode, a 64 kilobyte segment is selected by one of the segment selector registers. Each time a memory location is accessed, a segment is selected by one of the segment selectors and the memory address is used as an offset into this segment. In protected mode, the segment selector registers hold an index into the GDT[2]. This table contains the 32-bit base address of a segment. When memory is accessed, the offset is added to this address and the memory location at the resulting linear address is accessed. The table also contains bits to set permissions, such as marking segments as read-only or restricting access to privilege level 0 (kernel mode).

Paging allows each application to have what appears to be a large contiguous address space. Memory is divided in pages, which usually have a size of 4 kilobytes. Each page can be mapped to a frame in physical memory, or it can be stored on a backing store, such as a hard drive. Pages can have access restrictions, such as read-only, no-execute or supervisor-only rights. The translation of pages to frames, or virtual addresses to physical addresses, is performed by the MMU[3]. The MMU uses special data structures, known as page tables, to look up a virtual memory address and find the corresponding physical address each time memory is accessed. See [4, p. 328–336] for a more detailed description. Because a memory location is often referenced multiple times in a short timespan, the result of these translations is cached in the TLB[4]. The operating system is responsible for setting up the page tables and controlling the TLB.

[1]System-management interrupt
[2]Global Descriptor Table
[3]Memory-management unit
[4]Translation lookaside buffer

Intel x86 implements three paging modes[7, p. 4-2]: 32-bit paging; PAE paging; and IA-32e paging. S.M.A.C.K on x86 uses 32-bit paging, since it is the easiest mode to implement. 32-bit paging uses two-level page tables. The `CR3` register determines the start of the first-level page table (called page directory). A virtual address is organized as follows: the first 10 bits determine the offset in the page directory. The entry found at this offset points to a 4 MB physical page, or a second-level page table (called page table). The second 10 bits determine the offset in this page table. This entry points to a 4 KB physical page. This is shown in Figure 1. The entries in the page tables have bits to control caching and access restrictions. The meaning of these bits is described in Table 1. 32-bit paging does not have support for an execute disable flag.

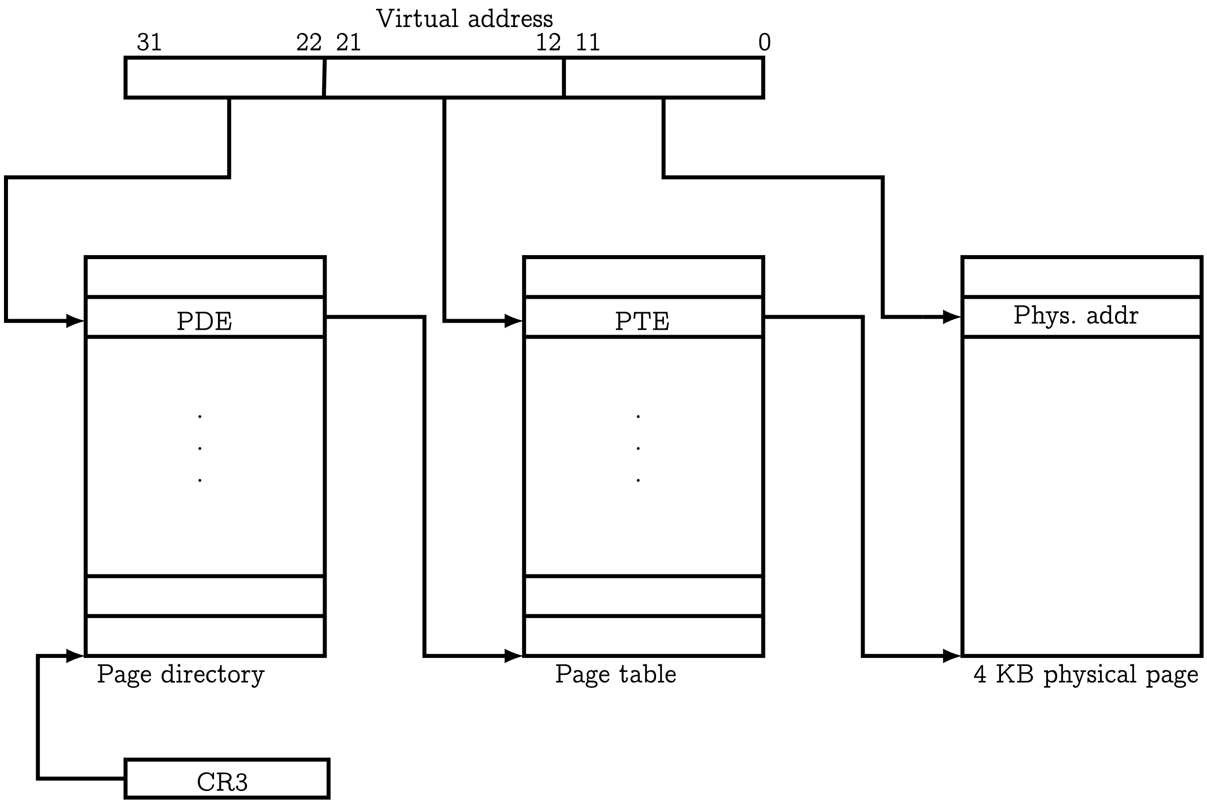


Figure 1: The translation of a virtual address to a physical address using 32-bit paging

In 32-bit paging, each paging structure consists of 1024 entries of 32 bits. The page directory can hold entries that refer to a 4 MB page or to a page table. A page table can hold entries that refer to a 4 KB page. Both structures can also hold a record that marks the entry as not present. All paging structures are aligned on a 4 KB page. S.M.A.C.K does not use 4 MB pages, so the structure of such an entry is omitted. The layout of the entries is shown in Figures 2 and 3.

31 … 12	11 … 8	7	6	5	4	3	2	1	0
Physical address of page table	Ignored	**0**	Ign	A	PCD	PWT	U/S	R/W	**1**

Figure 2: Layout of a PDE pointing to a page table

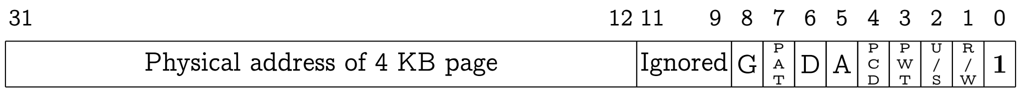


Figure 3: Layout of a PTE pointing to a 4 KB page

Bit	Description
G	Global. Determines whether the translation is global. Used for cache control.
PAT	Paging and memory typing. Indirectly determines the memory type used to access the referenced memory.
D	Dirty bit. Determines whether something was written to the referenced memory.
A	Accessed. Determines whether the entry has been used or not.
PCD	Page-level cache disable. Indirectly determines which type of caching is used to access the page table.
PWT	See above.
U/S	User/supervisor. 0 means that user mode (CPL=3) does not have access to the referenced memory.
R/W	Read/write. 0 means that writes are not allowed to the referenced memory location.

Table 1: The meaning of the different bits in the paging structures

2.2 Boot process

When a PC is powered on, the first thing that is loaded into memory is the BIOS[5] (typically loaded from a flash chip on the motherboard). The BIOS is responsible for doing basic initialization of the hardware, loading the first sector of the hard drive (known as the MBR[6]) to physical address 0x7C00 and transferring control to the boot loader by jumping to this address. At this point, usually a boot loader (such as GRUB) takes over, loading a user-specified kernel image from disk into memory and transferring control to this kernel. On newer PCs, the BIOS has been replaced by UEFI[5]. UEFI can load images directly from a partition on the hard drive instead of using a dedicated boot loader and also provides runtime services that can be accessed when the operating system is running. UEFI is currently not supported in S.M.A.C.K.

3 General approach to porting S.M.A.C.K

Porting an operating system is not a trivial task. First, a small “Hello world” kernel was written to become familiar with the boot process and the data structures that have to be initialized. This kernel would boot and print the text `Hello World` on the screen.

The architecture-specific code of S.M.A.C.K is located in subdirectories (e.g. `arch/arm/` and `arch/x86/`). When the kernel was ported to x86, the `arch/arm/` subdirectory was excluded from compilation and the required functions were replaced with empty stubs. The boot code was disabled by commenting out each line, and un-commenting these one at a time. The empty stubs were replaced by real code until all functionality was implemented.

[5]Basic Input Output System
[6]Master Boot Record

The following tools were used during development:

- Compilers: The kernel was tested with multiple compilers.
  - gcc 4.6.3
  - clang 3.1
- Linker: Used to form the final kernel image.
  - GNU ld (part of GNU binutils) 2.22
- Emulators: Bochs provides an environment that is closer to real hardware than the environment provided by QEMU, but attaching a source level debugger to QEMU is easier, since QEMU supports GDB by default.
  - Bochs 2.4.5
  - QEMU 1.0.1
- Boot loader: The boot loader is used to load and boot the kernel on real hardware.
  - GRUB 0.97
- Debugger: A debugger is used to trace problems and confirm that data structures are filled correctly.
  - GNU gdb 7.4.1

Hardware vendors often provide documentation about the hardware. Because an operating system needs to be able to control the hardware, it must know how to do this. The following documentation contains information about the CPU and the PC platform in general was used during this project:

- Intel(R) 64 and IA-32 Architecture Software Developer's Manual (Volume 2: Instruction Set Reference)[6].
- Intel(R) 64 and IA-32 Architecture Software Developer's Manual (Volume 3A: System Programming Guide)[7].
- IBM PS/2 Reference manuals[8].

4 Implementation

4.1 Memory layout and data structures

For S.M.A.C.K, it is currently assumed that the machine it runs on has at least 256 MB of memory available. The virtual memory layout is shown in Figure 4. The kernel is always mapped in, starting from address 0xC0000000. The kernel is always in memory, even when a user application is running. This is done so that the kernel does not have to do a full context switch which involves clearing caches such as the TLB on systems calls or interrupts that do not swap out the currently running process. The kernel has a total of 1 GB of memory available, which is likely to be enough for future changes.

Both memory segmentation and paging can be used on x86. Since S.M.A.C.K on ARM uses paging, like most modern operating systems, it will also use paging on x86. Memory segmentation cannot be turned off on x86, but all memory can be described in one big segment. These segments are set up using the GDT.

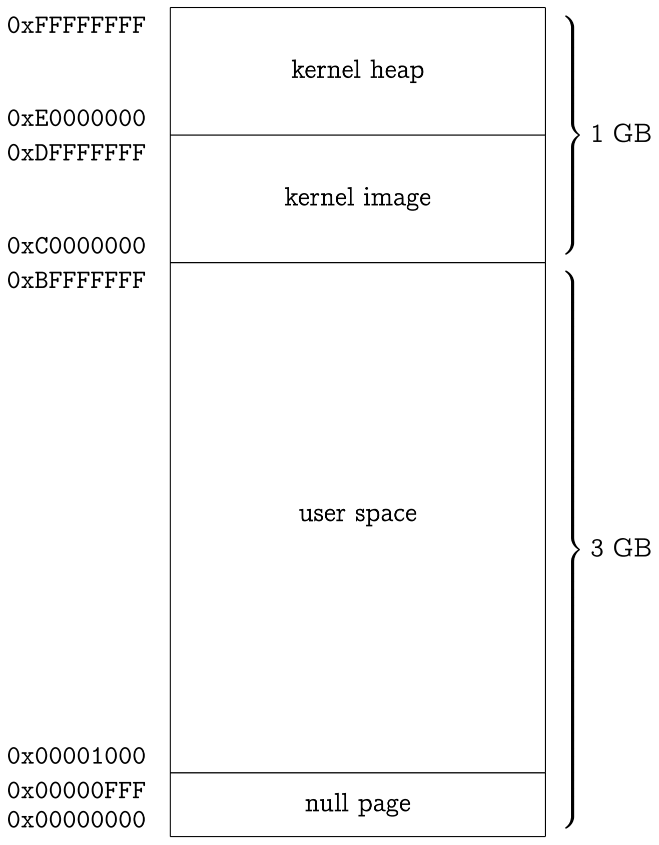


Figure 4: Virtual memory layout for S.M.A.C.K on x86

The GDT contains different types of descriptors. The segment registers select a code or data segment descriptor from the GDT for memory access. The GDT can also contain a TSS[7] descriptor, which is used for task switching.

The GDT used by S.M.A.C.K contains six entries. The first entry should always be `NULL`. There are two segments for kernel mode (code and data segment) and two for user mode. These entries describe all memory in one large segment, with a total size of 4 GB. The last entry is used to store a pointer to the TSS. The TSS contains the values that are to be loaded in various registers (i.e. the segment selectors, `EIP` and `ESP`) on a switch to privilege level 0 (kernel mode).

4.2 Booting

The kernel image conforms to the Multiboot Specification[9]. This specification describes a set of rules that must be adhered to by the kernel image that is to be loaded and the boot loader. The boot loader uses this information to load the image at the correct memory address. The S.M.A.C.K image is loaded by the boot loader at physical address 0x100000.

On the BeagleBoard, the kernel is always loaded at physical address 0x80000000 (the start of physical RAM) by the boot loader. On x86, RAM starts at 0x0. The kernel is mapped in at a high memory address (above 0xC0000000), so the MMU has to be enabled and set up to map the kernel to this address before the actual kernel can be started. To accomplish this,

[7]Task-state segment

the kernel has a `.header` section that is mapped 1-to-1, which contains code to initialize the MMU and paging structures, and set up the correct mappings. When this is done, the kernel jumps to the C function `boot` (which resides in the `.text` section). The `boot` function calls the `platform_init` function, which sets up the GDT and TSS (as described in Section 4.1) The 1-to-1 mapping of the `.header` section is removed, because it is only required during the early stages of the boot process. Details about the MMU and paging structures can be found in Section 2.1.

In order to form the final kernel image that can be loaded by a boot loader, the object files are linked together to form an executable. In contrast to the ARM version, the ramdisk is included in the final kernel image. This prevents the ramdisk from interfering with physical memory management. A side-effect of this is that it also prevents the kernel from releasing the memory occupied by the ramdisk should it become no longer necessary. The linker links all the sections together and sets the addresses of the symbols to the correct values. The fields in the Multiboot Header specify where the sections are to be loaded and where the entry point can be found. The different sections are shown in Table 2.

Section	**Virtual address**	**Size**
.header	0x00100000	0x2FE
.text	0xC0101000	0x9F9B
.rodata	0xC010B000	0xFED
.data	0xC010C000	0x1548
.ramdisk	0xC010E000	0x809000
.bss	0xC0917000	0x2CD0

Table 2: The different sections and their starting addresses in the final kernel image when compiled using clang 3.1

4.3 Context switching

S.M.A.C.K on ARM supports multiple processes. The same is supported on x86. This is implemented using virtual memory and paging. Every process has its own address space. The addresses starting from 0xC0000000 upwards are reserved for the kernel, whereas the memory range 0x00001000–0xBFFFFFFF is used for user applications. Processes should not be able to access memory of other processes. To ensure this, every process has its own set of page tables. The kernel page tables are shared and thus the same for each process. If the active process is changed, the page tables for the user application memory must be changed.

A task switch can happen in two cases:

1. Explicitly by means of a system call.
2. A timer interrupt (automatically, either because the kernel has finished blocking I/O for a task, or the scheduler decides to switch tasks).

The first case is implemented by the `cswitch` function, which gets called from `proc_switch`. It is called if a process voluntarily releases the CPU, e.g. using the `sleep` system call, or a blocking I/O call. The `cswitch` function stores the context of the current process in a data structure. This context consists of the values of the general purpose registers, the value of the instruction pointer, the value of the stack pointer and the current CPU state (`EFLAGS` register). Then, all aforementioned values are re-loaded to their respective registers for the process that is about to be resumed. Finally, the execution of the new process is resumed using an interrupt return instruction (IRET). In case of a switch to privilege level 3 (i.e. a switch from kernel to user mode), the IRET instruction performs the stack switch automatically. In case of a switch to privilege level 0 (i.e. to kernel mode), the stack switch is done manually by pushing the EFLAGS, CS and EIP values on the destination stack and executing IRET.

The second case is implemented by changing the CPU context that is stored by the interrupt handling routine. This is described in more detail in Section 4.5. The two methods of context switching should be compatible with each other, since they share the process data structures.

4.4 MMU

The virtual memory system in S.M.A.C.K consists of three parts: virtual memory management, physical memory management and hardware support functions. In the source code, the functions belonging to each part can be identified by looking at the prefix of the function name. The three prefixes are respectively vm_, pmap_ and hw_. All the hardware support functions were re-written for x86 32-bit paging. The other functions did not have to be changed, except for pmap_init to compensate for the difference between the virtual and physical addresses of the kernel symbols (an offset of 0xC0000000). Various ARM-specific assertions were also disabled for x86.

The hw_-prefixed functions set up the paging structures and perform operations on them, such as setting up a mapping for a new page or a manual lookup. The format of the paging structures is described in Section 2.1.

Every running process has its own page directory. The entries in this directory that describe the kernel range (0xC0000000 and up) point to the page tables of the kernel. This means that the kernel shares its page tables with every process, but not its page directory. It is currently not necessary to update the kernel part of the page directory after it has been created, because all page tables are statically created and always available. If this should become necessary in the future, the page directory of every process would have to be updated separately.

4.5 Interrupt handling

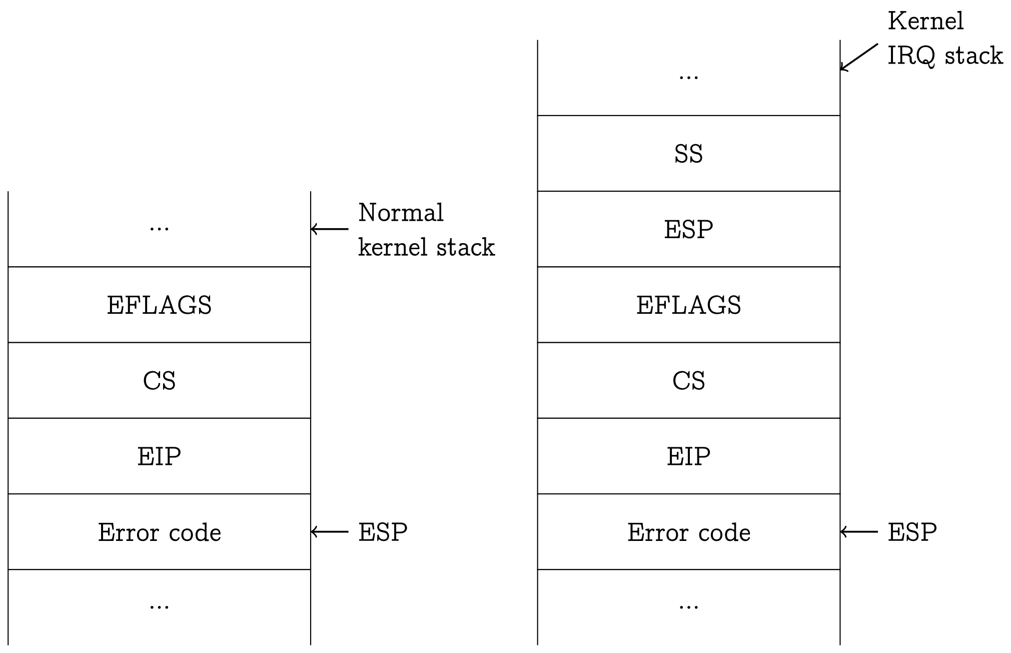
Interrupts are used to interrupt the CPU from normal operation when an external event occurs. This can originate from an external source (such as keyboard input, or a timer interrupt) or from an internal source (such as a divide by zero error). It is also possible to generate interrupts from software (known as a software interrupt). The CPU can respond to an interrupt by running a specific routine and resuming normal operation afterwards.

When an interrupt happens, the CPU looks up the routine that should be executed in the Interrupt Descriptor Table (IDT). This table contains pointers to functions. When the kernel boots, this table is initialized with pointers to interrupt service routines (ISRs). By default, the CPU interrupts (internal) overlap with the system interrupts (external). Interrupts numbers 0–31 are reserved for internal use by the CPU. To make sure there is no confusion about which interrupt was fired, the system interrupts are re-mapped to interrupt numbers 32–47 by programming the PIC[8] to give these interrupts an offset of 32. The PIC combines interrupt lines and forwards interrupts to the CPU. The PIC that S.M.A.C.K uses is an

[8]Programmable interrupt controller

Intel 8259. It is supported by all 8086 microprocessors, though newer models also support the APIC[9].

When an interrupt fires, the CPU can either be in user mode or in kernel mode. If the CPU is in kernel mode, it automatically pushes the EFLAGS, CS and EIP registers on the current stack. If the CPU is in user mode, the CPU does a stack and mode switch (switching to the kernel stack and kernel mode), and pushes the old SS and ESP registers to save the location of the old stack. The CPU then pushes the EFLAGS, CS and EIP registers on the stack. In either case, if the interrupt has an error code, this is also pushed on the stack. Finally, the corresponding ISR is called. The layout of the stack for both cases is shown in figure 5.



(a) Stack layout without privilege level change (b) Stack layout after privilege level change

Figure 5: Stack layout after an interrupt fired

Every type of interrupt has its own ISR. The ISRs for all interrupts are similar. If the interrupt did not push an error code, a dummy error code (0) is pushed on the stack. Finally, the interrupt number is pushed on the stack and the isr_common routine is called.

The isr_common routine is responsible for calling the C function interrupt. To make sure that the normal operation of the CPU can be restored when interrupt handling is ready, the current CPU state is saved on the stack. This state (including the values pushed by the ISRs) is saved in a interrupt_state_t data structure. The layout of this structure is shown in Figure 6. If the CPU did not perform a stack switch, this is done manually by copying the stack contents to the kernel interrupt stack. This ensures that all interrupt code can handle the state in a uniform way.

At some point during development, the kernel would crash when returning from a context switch. The problem turned out to be related to the fact that the CPU does not perform a stack switch if there is no privilege level change. If the kernel interrupted a process that was running in kernel mode and resumed a process that was running in user mode, the kernel would set the SS and ESP fields of the interrupt stack (see Figure 5), but since these fields are not available if there is no privilege level change, it would overwrite part of the stack of

[9] Advanced programmable interrupt controller

the interrupted process. The problem was corrected by doing a stack switch explicitly if the CPU did not do one, with dummy values for the SS and the ESP fields.

When the kernel switches processes (pre-emption), the contents from the interrupt_state_t data structure are copied to the cpu_ctxt_t data structure of the old process, and the contents from the new process are copied back the the interrupt_state_t data structure. This way, when the CPU resumes normal operation, it will resume executing a different process.

The interrupt function forwards interrupts to other systems, such as the timer module or the keyboard driver. Page faults and general protection exceptions are also processed. Currently, other exceptions (such as divide by zero) are not handled.

```
typedef struct {
    uint32_t ds;
    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
    uint32_t int_no;
    uint32_t error_code;
    uint32_t eip, cs, eflags;
    uint32_t useresp, userss;
} interrupt_state_t;
```

Figure 6: The interrupt_state_t data structure

4.5.1 System calls

System calls are used by user applications to call on services provided by the kernel. This can for instance be to read from a file on the file system, or to get the current time. The x86 architecture provides multiple ways to implement system calls. S.M.A.C.K uses the oldest (and most widely supported) one of them, software interrupts.

When an application wants to perform a system call, it has to perform the following steps:

1. Store the system call number in the EAX register.
2. Store the argument for the system call in the EBX register.
3. Generate software interrupt 0x80 (using the INT instruction).
4. Read the return value from the EAX register.

When interrupt 0x80 fires, the kernel will handle it differently from the other interrupts. It will save the current CPU context, so the user application can be resumed when the system call is ready. It will then copy the stack contents (that were pushed on the IRQ stack, see section 4.5) to the application's syscall stack. It will then call the hw_syscall function, which will call the platform-independent syscall function, which will call the specific system call handler corresponding to the system call number (retrieved from the EAX register). When the system call is finished, the return value is saved in the EAX register and the user process is resumed.

4.6 Hardware drivers

S.M.A.C.K on x86 uses the screen for output and the keyboard for input. The input and output is done using the console driver, which relies on two low level drivers to interact with the hardware. These two drivers are the keyboard driver (described in section 4.6.1) for input and the VGA[10] console driver (described in section 4.6.2) for output. User applications can use this console driver (using the virtual device `/dev/serial`) to perform basic I/O.

4.6.1 Keyboard driver

The keyboard driver is responsible for translating scancodes (generated by the keyboard) to keycodes. These keycodes are forwarded to the console driver, which will buffer them until a user application is ready to read them.

The keyboard driver itself is implemented as a table-driven state machine. A scancode corresponds with a certain action performed by the user. This action can be the pressing or release of a key. When the keyboard controller has data available for the driver, it generates an interrupt and the driver can read one byte from the controller. Because a scancode can consist of multiple bytes, the keyboard driver maintains an internal state to remember which bytes have been received. The following example will illustrate this. The table provides a mapping from state, optional modifier (e.g. shift, caps lock), scancode to new state, action (e.g. key up/down, set or clear modifiers) and a parameter for that action.

If the user releases the escape key on the keyboard, the keyboard generates a scancode (0x81) and the keyboard controller generates an interrupt. The keyboard driver reads one byte of data from the keyboard controller. Based on this value, it looks in its table and finds the corresponding action (`A_KEYUP` with keycode `KEY_ESC`). It notifies the console driver of this action. The scancode 0xE0, 0x47 corresponds with the pressing of the home key. If the driver reads the 0xE0 byte, it remembers that it has seen 0xE0 by moving to a different state. The driver then receives 0x47 while it is in this state, and it performs a look-up in its table to find the correct action for this scancode.

The table also contains a field for modifier keys. The $ key (Shift+4 on a US Qwerty keyboard) generates the same scancode as the 4 key. The modifier field for the $ key is set to `MOD_SHIFT`, so the corresponding action is only activated if the Shift key is pressed while the 4 key is pressed.

4.6.2 VGA console driver

VGA hardware can operate in a number of modes. By default, it operates in a text mode. This means that the VGA hardware will display characters in a 80x25 grid. The operating system can place text on the screen by writing character data to the framebuffer (address 0xB8000). Every character is represented by a 16-bit word[10]. The layout is shown in Figure 7.

15	14	8	7 0
Blink	Background color	Foreground color	Character

Figure 7: VGA attribute / data

To display a character at position (x, y), the kernel uses the following formula to find the memory address to write the data ($base$ = 0xB8000):

$$addr = base + y \cdot 80 + x$$

[10]Video Graphics Array

The kernel computes the data that is used to write a character c using foreground color f and background color b:

$$data = ((b \ll 4) \mid (f \mathbin{\&} \texttt{0x0F})) \ll 8 \mid c$$

If the screen is full, the driver scrolls the console by copying every line to the line above it, discarding the first row. The last row is emptied before data gets written to it.

5 Conclusions

This thesis discussed the process of porting S.M.A.C.K to the x86 architecture. The resulting kernel can run on the QEMU emulator and on physical hardware. The `mmaptest` program runs successfully. The kernel, in its current state, is portable, although the shared code needed some minor changes. This code has been surrounded with `#ifdef` statements to include such blocks only in the compilation process for one specific platform. It is likely that more code that can be shared between platforms will be identified as the development of the kernel continues.

The shared code that had to be changed related mostly to the virtual memory system. The shared code contained hard-coded assertions that were specific to the memory layout of the BeagleBoard. It was also assumed that the physical and virtual memory addresses would be the same for the kernel image. On x86, these are offset by a pre-defined constant. The portability could be increased by always applying this offset, but with a value of 0 for platforms on which the kernel is loaded with the same virtual and physical addresses. The virtual memory range for user processes was also hard-coded in the shared process creation code. This memory layout does not have to be the same for all platforms, so this code should be moved to the platform-specific subdirectories.

The kernel did not contain a platform-specific initialization function, which had to be added to initialize data structures that are specific to x86 (e.g. the GDT). Platforms that do not need a specialized initialization routine could simply provide an empty function. The ramdisk driver was also changed for the x86 platform, because the ramdisk is included in the final kernel image. Although not technically necessary, this does make the handling of the ramdisk easier. In order to be portable, the ramdisk driver should provide support for ramdisks that are loaded by the bootloader and for ramdisks that have been included in the kernel image.

There are interesting opportunities for future work. The current x86 port provides the same functionality as the ARM version, except for the exceptions mentioned in Section 1. Possible future work includes:

- The x86 caching mechanism is currently not enabled. Although the kernel seems to work fine when caching is enabled, certain areas of memory should not be cached (such as memory-mapped devices). Caching should, where possible, be enabled and configured in such a way that memory access to the memory area used by devices is not cached.
- Memory on x86 does not have to be contiguous. Currently, it is assumed that there will be 256 MB of memory available, starting at the lowest physical address. In reality, this does not have to be the case. The kernel should be able to detect the memory layout of the system it is running on and configure the memory regions in the virtual memory subsystem accordingly. In its current state, the virtual memory system does not support multiple regions of physical memory and a variable amount of memory. This should be addressed before correct handling of the memory layout is possible.

- The platform-dependent code that is not in the platform-specific subdirectory but surrounded with `#ifdef` statements should ultimately be moved to files in the platform-specific subdirectories.
- It would be possible to write support for virtual consoles on framebuffer devices. Most graphics adapters support the VESA VBE standard ([11]), which could be used to access the framebuffer without depending on a vendor-specific driver. In order to keep the kernel portable, the virtual console support should be designed in such a way that it does not depend on one specific framebuffer implementation.
- Parts of the console driver on x86 and the serial driver on ARM could be merged into a platform-independent high-level driver. The logic for communicating with user applications and buffering can be shared.

References

[1] Jim A. McCall, Paul K. Richards and Gene F. Walters. *Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality.* Final technical rept. Aug 1976-Jul 1977, GENERAL ELECTRIC CO SUNNYVALE CA

[2] beagleboard.org. `http://www.beagleboard.org`

[3] Assignment 3: Virtual Memory (Operating Systems, spring 2012), `http://www.liacs.nl/~krietvel/courses/os2012/os-2012-assignment3-virtual-memory.pdf`

[4] Abraham Silberschatz. *Operating System Concepts.* John Wiley & Sons Software, 8th edition, 2009.

[5] The UEFI homepage. `http://www.uefi.org/home/`

[6] Intel Corporation. *Intel(R) 64 and IA-32 Architecture Software Developer's Manual, Volume 2.* December 2011.

[7] Intel Corporation. *Intel(R) 64 and IA-32 Architecture Software Developer's Manual, Volume 3A.* December 2011.

[8] International Business Machines Corporation, *PS/2 Reference manuals.* `http://www.mcamafia.de/pdf/pdfref.htm`. Retrieved Jul 7 2012.

[9] *Multiboot Specification version 0.6.96*, `http://www.gnu.org/software/grub/manual/multiboot/multiboot.html`. Retrieved June 28 2012.

[10] OSDev Wiki, *Text UI*, `http://wiki.osdev.org/Text_UI`. Retrieved July 19 2012.

[11] Video Electronics Standards Association, *VESA BIOS EXTENSION (VBE) version 3.0*, `http://www.petesqbsite.com/sections/tutorials/tuts/vbe3.pdf`