Review of Current Principalled Approaches to System Programming

Olivier Lischer
OST Eastern Switzerland University of Applied Sciences
MSE Seminar "Programming Languages"
Supervisor: Farhad Mehta
Spring 2024

Abstract

Computers are an essential part of our everyday life. In this paper, I described based on the Intel's CPU which hardware initialization an operating system has to perform and how to interact with the hardware. Starting from how a computer boots to memory management and interrupts to multitasking and finishing with kernel API. The second focus is the language in which the operating system is written. As most general purpose operating systems are written in C/C++ the question arose, if it is possible to write one using functional programming languages and techniques. I described why functional programming languages are not always suited for systems programming, and how one still could use techniques and concepts from such languages.

Keywords: Systems Programming, Operating Systems, Boot sequence, Interrupts, Paging, Multitasking, Haskell

1 Introduction

Operating systems are an important part in our every day life. We are often not even aware that we are interacting with a kind of operating system. The operating system is the fundamental software to control our hardware. If the operating system is faulty, such errors can affect the correctness of other applications running on top of the operating system.

In this paper, I want to have a look at how a computer boots up and what tasks the kernel has to perform until a developer can write its own application for the operating system. To show how the discussed problem is solved in most modern operating system, I will mainly use Linux and FreeBSD as examples. These operating systems, as well as most other general purpose operating systems, are written in either C or C++. As C/C++ are often considered "unsafe" programming languages, it may be interesting to consider alternatives. This leads to the questions, is it possible to write certain parts of an operating system in another language? To answer these questions, I will show if and how the same problem could be solved using methods from functional programming languages, such as Haskell.

Since many details are CPU dependent, I will focus on CPUs of the Intel 64 and IA-32 architecture family, as they are among the most commonly used CPU architectures today.

2 The boot sequence and the bootloader

Each computer has to go thought the boot sequence before it is ready for use. These steps can be roughly broken down into the following steps:

1. Motherboard initialization
2. BIOS / UEFI initialization, including Power-On Self Test
3. Master Boot Record
4. Bootloader
5. Kernel Initialization
6. First User-Mode Process

In this section the steps from the motherboard initialization up to the bootloader are discussed. The last two steps are discussed in the other sections.

2.1 Hardware initialization

When the computer is powered on, the motherboard starts with its own initialization using its firmware. At this point, how and what happens is very dependent on the manufacturer. In case everything worked, the motherboard tries to get the CPU running.

When the CPU starts, the registers in are set to some predefined values. On a modern Intel CPU the instruction pointer is set to `0xFFFFFFF0`, the location where the Basic Input/Output System (BIOS) / Unified Extensible Firmware Interface (UEFI) should be located . The CPU is running in *Real Address Mode* where only 16-bit mode is available, and starts the execution at this location and runs the BIOS / UEFI code.

The BIOS' / UEFI's responsibility is to perform a *Power-on self-test (POST)* to verify that all minimum required components are available and working properly. If something is wrong, the BIOS notifies the user of the error using manufacturer-dependent methods. For example, the UEFI on the "ROG Strix X570-E Gaming" motherboard displays an error code on a seven-segment display, while other motherboards notify the user via a specific beep pattern[ROG, HPD]. However, as most BIOS / UEFI implementations are closed source, it is difficult to say, what exactly happens here.

The next important task of the BIOS / UEFI is to determine the boot device. Most modern BIOS / UEFI implementations permit a selection of possible boot devices.

The very last task in the POST is the `INT 0x19` instruction. This instruction will read the first 512 bytes from the first sector of the chosen boot device and loads it to the memory location `0x7c00`. After this, the CPU jumps to this location and starts executing the so called *Master Boot Record (MBR)*.

2.2 Loading and executing the bootloader

As the MBR is only 512 bytes long, the whole bootloader cannot be placed there. Therefore, the MBR has the task to load the remaining part of the bootloader from the boot device.

The remaining task of a bootloader is to load the operating system. Older operating systems usually have their own bootloader. It was therefore difficult or even impossible to install several operating systems on the same machine. The *Multiboot Specification* solves this problem by defining an interface between the bootloader and the operating system. This means that any multiboot compliant bootloader can load any mulitboot compliant operating system[Mul].

The *OS image*, a regular `a.out` binary file that can be loaded by any multiboot bootloader, must contain a so-called *Multiboot header*. The C code for such a multiboot header can be seen in Listing 1.

```
1  typedef unsigned int    multiboot_uint32_t;
2  struct multiboot_header
3  {
4    /* Must be MULTIBOOT_MAGIC - 0x1BADB002 */
5    multiboot_uint32_t magic;
6    /* Feature flags.  */
7    multiboot_uint32_t flags;
8    multiboot_uint32_t checksum;
9    /* These are only valid if MULTIBOOT_AOUT_KLUDGE
         is set.  */
10   multiboot_uint32_t header_addr;
11   multiboot_uint32_t load_addr;
12   multiboot_uint32_t load_end_addr;
13   multiboot_uint32_t bss_end_addr;
14   multiboot_uint32_t entry_addr;
15   /* These are only valid if MULTIBOOT_VIDEO_MODE is
          set.  */
16   multiboot_uint32_t mode_type;
17   multiboot_uint32_t width;
18   multiboot_uint32_t height;
19   multiboot_uint32_t depth;
20 };
```

Listing 1. Multiboot header in Grub

The `entry_addr` field contains the address to which the bootloader should jump after initialization in order to start the operating system. The other fields are required for the bootloader to load the `a.out` file and later to initialize the file. This information is stored in the *Multiboot information structure*[Mul].

2.3 Functional programming in bootloaders

The MBR will likely never be written in a language like Haskell because of the extreme size constraints. Another problem is, that Haskell (more specific GHC) produces an executable binary that includes the Haskell Runtime System, which depends on an existing operating system. The fact that it is possible to modify GHC so that it can run directly on bare metal was demonstrated in the development of House [HJLT05].

Another approach is to write an "executable specification" in a functional programming language and verify it. Such an approach was demonstrated using the *riotboot* bootloader in [YT21]. They formalized the ARM Instruction set architecture (ISA) and then implemented *riotboot* in F*, a general-purpose proof-oriented programming language and Low*. With Low*, the F* implementation could be compiled into verified C code[PZR+17].

3 Interrupts

After the bootloader has loaded the kernel, further initialization is required. To allow external peripheral devices such as keyboards and mice to interact with the operating system, the CPU most somehow react to their events. Also, the CPU must somehow be able to react to faulty behaviour. In modern CPUs, this is achieved using interrupts.

The CPU differentiates between two types of interrupts: *CPU exceptions* and *hardware interrupts*. The hardware interrupts that are normally triggered asynchronously by an I/O device such as a keyboard or mouse while exceptions are synchronous events that are generated when the CPU detects one or more predefined conditions[Int].

In the following, two types of interrupts are discussed, first the CPU exceptions and second the hardware interrupts.

3.1 Handling CPU exceptions

CPU exceptions are similar to exceptions in languages like Java or C++. If an invalid action is performed, for example division by zero, the programming language will throw an exception, and the programmer may catch it. If the CPU itself tries to perform a division by zero, it will call the *division by zero* interrupt handler[Int].

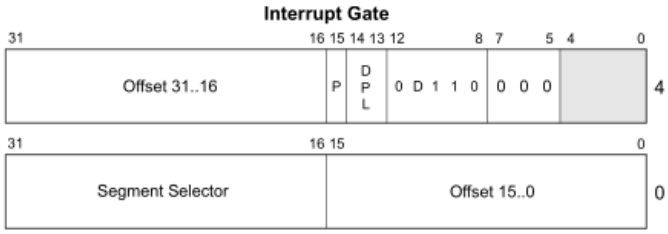
These handlers are not hard coded into the CPU but can be configured using the *interrupt descriptor table (IDT)*[Int]. The base address to the interrupt descriptor table is stored inside the `IDTR` register.

The structure for the IDT is CPU dependent and can therefore vary. The IDT can store up to 256 entries. Depending on the architecture the offset is either 8 bytes for a 32 bit architecture or 16 bytes for a 64 bit architecture and is lay outed in memory as seen in Figure 1[Int].

Each exception corresponds to a specific offset in the interrupt descriptor table. For example, the *division by zero* exception has the offset 0. The interrupts with offset 0 up to 21 are well-defined. A selection is given in Table 1. Interrupts from 22 to 31 are reserved for usage by Intel, and from 32 to 255 are used for hardware interrupts.

Interrupts can be categorized into four types: faults, traps, interrupts and aborts[Int]. A fault is triggered, if instructions

Figure 1. IDT Gate Descriptors[Int]



Offset	Description	Type
0	Divide Error	Fault
3	Break Point	Trap
8	Double Fault	Abort
14	Page Fault	Fault
32 - 255	User Defined Interrupts	Interrupt

Table 1. Selection of CPU exceptions

are executed that lead to errors on the CPU itself, for example a division by zero. Traps are exceptions that are reported immediately following the execution of the trapping instruction. An example for a trap is the breakpoint instruction `INT3`. After the instruction is executed, the handler is called. After the handler is finished, it will return after the causing instruction. A so called *Double Fault*, two unhandled faults, is a non-recoverable error and forces the CPU to shut down. This is called an abort. The interrupts are the exception that are triggered by peripheral devices such as keyboards. They are explained in subsection 3.2.

3.2 Handling hardware interrupts

Interrupts are an important thing in today's CPU. They allow stopping the current work on the CPU, perform some arbitrary other work, and the switch back to where it was previously. For example, the keyboard controller can send an interrupt when a character key was pressed. The CPU will receive the interrupt, stop the current execution, and for instance print the character on screen. Afterwards, the CPU will return to what it was doing during before the interrupt. Interrupts are technically similar to CPU exceptions, except that they can be configured by the kernel.

Modern Intel CPUs consist of an *Advanced Programmable Interrupt Controller (APIC)*. The APIC receives interrupts from the interrupt pins of the processor, from internal sources and from an external I/O APIC or any other external interrupt controller. The local APIC then sends these interrupts to the processor core for handling. The primary task of the external I/O interrupt controller is to receive interrupts events from the system and the associated I/O devices and then forward them to the local APIC. To handle interrupts send by the local APIC, the processor uses the interrupt and exception handling mechanism described in subsection 3.1. The APIC interrupts are received by the processor as interrupts with an offset starting at 32, while the others are used by exceptions, as already described in subsection 3.1[Int]. These interrupts are also referred to as *interrupt request (IRQ)*.

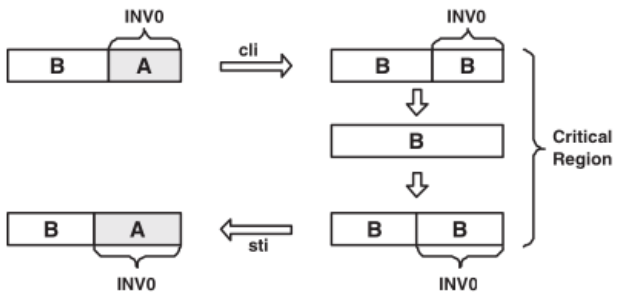
In today's computer, it can be neglected between the kernel and the firmware of the hardware which interrupt number / IRQ the I/O device should trigger. Depending on the assigned IRQ number, the interrupt has a higher or lower priority. The bits 7:0 in the IRQ indicate the interrupt priority, whereby a lower number means a higher priority.[Int].

3.3 Reasoning about interrupts

Interrupts are extremely hard to reason about, as an external source can alter the control flow and state. This makes the invariants in low-level concurrent code complicated. In [FSDG08] they describe how it is possible to reason about interrupts using ownership-transfer semantics. The developed an *Abstract Interrupt Machine (AIM)*, a theoretical representation of a computer, and assigned an operational semantics to each assembler instruction.

Informally, the idea is the following. In a single threaded system the memory would be divided into two blocks. One block (called block A) of the global memory is reserved for interrupt handlers, and the other block is freely usable by the thread. As an interrupt (handler) can pre-empt a running task, the block A must be well-formed and hold its invariants. When the operation `cli` to disable interrupts is executed, the semantic is that the block A becomes also part of block B. Before the interrupts are activated again using `sti` the memory block that was part A, must restore its invariant. The idea can be seen in Figure 2.

Figure 2. Memory Partition for Handler and Non-Handler[FSDG08]



This is reminiscent of *Resource acquisition is initialization (RAII)*, a common pattern in C++. If operations are to be executed while interrupts are disabled, such an approach would be possible. The idea is that a resource is acquired and initialized when an object is created. In this case, the

constructor would disable interrupts by calling `cli` and saving the current state (registers). If the object goes out of scope (is destroyed), the destructor would restore the previous state to reactivate interrupts by calling `sti`. In Haskell one could write something similar by writing a function `withoutInterrupts` that disables interrupts and saves the state, executes the provided function and then restores the state and re-enables the interrupts. This approach does not prevent a developer from writing invalid code, but by using the provided constructs, the developer does not think about how to preserve the invariants.

4 Memory management

After all interrupts have been set up and the kernel can react to them, the memory should be set up correctly

Normally, the CPU runs multiple applications simultaneously. However, an application developer should not care if there is another application running at the same time. The application developer should write his own application, as it is the only one that runs on the CPU. To achieve this, modern operating systems utilize the paging system from CPUs, as one can see in the Linux Kernel as well as the documentation for FreeBSD[Bai].

Paging is the process of translating linear addresses or virtual addresses, addresses used by the application, to real physical addresses. These physical addresses are then used to access the data in the memory or I/O devices. At the same time, the system verifies that the currently running process is also allowed to access the given memory location. This mechanism also has the advantage that it is possible to provide more virtual memory than the available physical memory by offloading unused memory to secondary storage such as hard disks or SSDs.

The mechanism used for paging differs slightly, depending on the CPU architecture and the used address size. An Intel CPU has four paging modes:

- 32-bit paging
- PAE paging
- 4-level paging
- 5-level paging

[Int]

In this section, only the 32-bit paging will be discussed, as the basic concept is the same for all modes.

The CPU has, beside *General Purpose Registers*, also other kinds of register. One of them are the *Control Registers*. Those are used to enable and disable features on the CPU itself. To enable the 32-bit paging, mode the register `CR0.PG` must set to 1 and the `CR4.PAE` to 0. `PG` and `PAE` corresponds to specific bits' in the register[Int].

4.1 The MMU

The *Memory Management Unit (MMU)* is a part of the CPU that performs the memory translation. The MMU maps the memory through two tables: the *page directory* and the *page table*. The page directory is essentially an array of page directory entries (PDE). Every page directory entry has the layout as described in Figure 3.

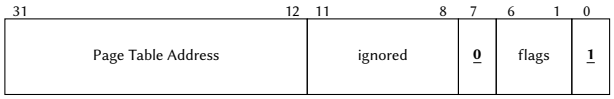


Figure 3. The page directory entry layout

The bits 31:12 refer to the next structure, the page table. The bit 7 must be 0, otherwise the CPU would work with another mode (4-MByte pages instead of 4-KByte pages). The bit 0 indicates, that the page directory entry is available when set to 1.

The *page table* is the second paging structure and is similar to the page directory. The primary difference between them is that the upper part of the page table entry points to the *page frame*, a 4-KByte area of the physical memory, instead of another paging structure. In Figure 4 one can see the layout of a page table entry. The bit 0 indicates, again, that the page frame is available in memory when set to 1. The bits 8:1 store various flags, such as if the page frame was changed while loaded.

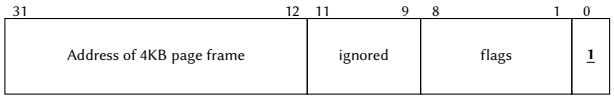


Figure 4. The page table entry layout

To tell the CPU where the page directory lies in memory, one has to write the base address to the register `CR3` The address to the page directory must be placed into the bits' from 12 to 31 as shown in Figure 5.

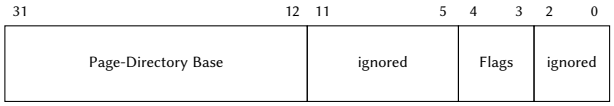


Figure 5. The CR3 register

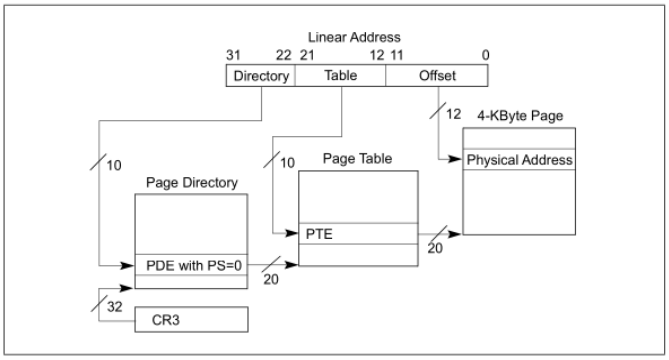
The MMU calculates the following three parts from a given virtual address[Int]:

- the index of the page directory entry (PDE): the most significant 10 bits (22 - 31)
- the index of the page table entry (PTE): the next 10 bits (12 - 21)
- the page offset: the least significant 12 bits (0 - 11)

In Figure 6 the translation process from the linear address to the physical address is depicted. Using the index into the page directory, it will find the corresponding entry. This

entry allows the MMU to find the correct page directory entry and from there the right page frame. The page offset is then used to find the physical address inside the page frame.

Figure 6. Linear-Address Translation to a 4-KByte page using 32-Bit Paging[Int]



If during the translation an error occurs, the processor issues a page fault. For why page faults can happen and how they are handled is discussed in the next section.

4.2 Page fault

A page fault happens, when a process tries to read or write from a linear memory that is not mapped to physical memory, or when the process does not have the required access permissions. A page fault caused because the linear memory is not mapped to the physical memory is called a *pure* page fault and is not considered an error. Pure page faults should be resolved by the installed page fault handler, as describe in subsection 3.2. The linear address, that causes the page fault, is available in the `CR2` register[Int], the handler can resolve the interrupt by loading the appropriated paging structure and page from the secondary storage and map it into the memory. As the page is now available, the interrupt can finish here, and the execution of the interrupted process can go on.

In all other cases, the kernel can not do anything to resolve the interrupt. In this case the kernel can not do anything any more must terminate the process.

4.3 Paging using functional programming

In [HJLT05] the authors describing the memory management of the *House*, an operating system written in Haskell. House utilizes the *Foreign Function Interface (FFI)* of Haskell to perform low level operations. All these low level operations are abstracted by the `H` monad. In Listing 2 one can see the public interface for working with pages and virtual memory.

```
1  -- Page interface --
2  type Page a     = Ptr a
3
4  allocPage :: H (Maybe (Page a))
5  freePage :: Page a -> H () -- caller must ensure arg
       is valid
6  registerPage :: Page a -> b -> (Page a -> H()) -> H
       ()
7  zeroPage :: Page a -> H()
8  validPage :: Page a -> Bool
9
10 -- Virtual memory interface --
11 type VAddr = Word32
12 minVAddr, maxVAddr :: VAddr
13 minVAddr = 0x10000000
14 maxVAddr = 0xffffffff
15
16 type PTE        = Word32
17 type PTable     = P.Page PTE
18
19 type PDE        = Word32
20 type PDir       = P.Page PDE
21 data PageMap    = PageMap {fromPageMap::PDir}
22   deriving (Show, Eq, Ord)
23
24 allocPageMap :: H(Maybe PageMap)
25
26 data PageInfo
27   = PageInfo {
28       physPage :: PhysPage,
29       writable :: Bool,
30       dirty :: Bool,
31       accessed :: Bool
32     }
33     deriving (Eq,Show)
34
35 setPage :: PageMap -> VAddr -> Maybe PageInfo -> H
       Bool
36 getPage :: PageMap -> VAddr -> H(Maybe PageInfo)
```

Listing 2. Houses paging interface

`PageMap` is the data type that provides the entry point to the translation of virtual addresses to physical addresses. One can see clearly the representation of the concepts described in subsection 4.1 in the Listing 2, with the `PageMap` as the representation of the `CR3` registers with a pointer to the page directory entry. Using the `getPage` one can retrieve the current state of the page, e.g. is writeable or the physical address. In the public interface, a function for freeing a `PageMap` is missing. As House is written in Haskell and therefore uses a garbage collector (GC) such a function is not required. The `PageMap` will be freed as soon, as no other part of the system refers to the `PageMap`.

One could write a virtual memory management system in any programming language. One difficulty when writing such a memory management system is, that one has to work with the raw memory as this happens in most cases using pointers. In C/C++ the type system can not help as much as in Haskell with the right pointer type because C/C++ does perform many implicit casts. The solution proposed in House may be even extended to not only use type alias but use new types, to prevent usage of the raw `Ptr` type.

5 Multitasking

On modern computers, it seems that many applications are run in parallel. To enable a such a parallel experience, the

CPU switches between two process / tasks at speeds such that it seems that two or more tasks are running in parallel. This is called multitasking.

In the following I will focus again on the 32-bit architecture as the 64-bit architecture supports hardware based context switches only while working in 32-bit mode (protected mode).

5.1 Context switching on the CPU level

A *task* can be described as a unit of work that a processor can dispatch, execute and suspend. More or less anything can be executed in a task, from the execution of a program to an operating system service to an interrupt/exception handler. The CPU itself provides a mechanism to save the state of a task and to switch from one task to another[Int].

A task consists of two parts: the task execution space and a *task-state segment (TSS)*. The task execution space consists of a code segment, a stack segment and one or more data segments. In addition to the task execution space, the task state segment also saves the task state. If paging is implemented for a specific task, the base address of the page directory is loaded into the `CR3` register. The segments, blocks of memory, are managed using the *global descriptor table (GDT)*. The GDT is an array of segment descriptors, a so-called segment descriptor table. There are two types of such tables: exactly one global descriptor table and many local descriptor tables, called *local descriptor table (LDT)*. As the name suggests, the GDT is used globally by all tasks, while LDTs are task-specific structures. The segments are not saved directly in the GDT, but are referenced by segment descriptors within the table. A segment descriptor contains information about the segment, e.g. the required privilege level or the storage location of the segment.

Before a task is executed, the current task state must be saved inside the TSS. The task state consist of the following items:

- Current execution space, defined by the segment selectors in the segment registers
- State of various registers and flags
- instruction and stack pointer
- link to the previously executed task

After that, the CPU will load the TSS from the new task and will populate the registers with the values from the TSS.

According to the Intel CPU Manual[Int], the processor transfers execution to another task in one of four cases:

- the current execution performs a `JMP` or `CALL` to a TSS descriptor in the GDT
- the current execution performs a `JMP` or `CALL` to a task-gate descriptor in the GDT or LDT
- an interrupt or exception vector points to a task-gate descriptor in the IDT
- the current task executes an `IRET` when the NT (Nested task) flag in the EFLAGS register is set

While hardware based context switches seems tempting to use it has major drawbacks. It stores and performs more checks than often required. As most modern operating systems do not use segments any more, it is not required to save them and perform any checks related to them. However, the CPU saves them and this requires more operations and is therefore not as fast as the software base context switch.

In the next section will be explained how the Linux kernel performs task switches.

5.2 Context switching using software (in the Linux kernel)

For a simple software based context switching, only a few steps must be implemented:

1. Save the current state on the stack
2. Save the current stack pointer and reload a new stack pointer
3. Restore the state of the other context

The current state consists of the stack pointer, instruction pointer and the EFLAGS, general register and any data segment registers. If the paging structure should be changed during the context switch, the `CR3` register must be reloaded as well.

Next, I want to show how Linux implemented the context switch on a conceptual level. After the scheduler has decided which task should run next, the function `context_switch` is called to actually perform the switch. In Listing 3 a simplified version of the `context_switch` from the Linux Kernel can be seen. The function operates on the running queue (`rq`), the previously running task, `prev` and the task that should run as next. First, various preparations have to be done, generic ones as well as architecture dependent ones. Depending on the mode of the next and previously running task different configurations must be happened. The field `mm` keeps track of the current configured memory paging setup. For example, if the next task is a kernel task, the paging system is not required to be updated as the kernel always works with the real physical memory addresses. However, the kernel must keep track of the current configured paging setup as it could be a required information in the next task. After all preparations, the switch happens inside `switch_to`. Inside the `switch_to` function is architecture specific code executed.

```
1  static __always_inline struct rq *
2  context_switch(struct rq *rq, struct task_struct *
       prev,
3                 struct task_struct *next, struct
                      rq_flags *rf)
4  {
5    prepare_task_switch(rq, prev, next);
6    arch_start_context_switch(prev);
7
8    if (!next->mm) { // to kernel
9      enter_lazy_tlb(prev->active_mm, next);
10
11     next->active_mm = prev->active_mm;
```

```
12      if (prev->mm) // from user
13        mmgrab_lazy_tlb(prev->active_mm);
14      else
15        prev->active_mm = NULL;
16    } else { // to user
17      membarrier_switch_mm(rq, prev->active_mm, next->
            mm);
18      switch_mm_irqs_off(prev->active_mm, next->mm,
            next);
19      lru_gen_use_mm(next->mm);
20
21      if (!prev->mm) { // from kernel
22        rq->prev_mm = prev->active_mm;
23        prev->active_mm = NULL;
24      }
25    }
26
27    switch_mm_cid(rq, prev, next);
28    switch_to(prev, next, prev);
29    return finish_task_switch(prev);
30  }
```

Listing 3. Context switch in Linux

For the 32-bit architecture of an Intel 86 CPU the following happens:

- pushing various registers to the stack
- pushing the EFLAGS to the stack
- switch the stack by switching the stack pointer
- popping the EFLAGS from the stack
- popping the registers from the stack

A simplified version of the switch on an 32-bit Intel CPU can be seen in Listing 4.

```
1   /*
2    * %eax: prev task
3    * %edx: next task
4    */
5   .pushsection .text, "ax"
6   SYM_CODE_START(__switch_to_asm)
7     /*
8      * Save callee-saved registers
9      * This must match the order in struct
           inactive_task_frame
10     */
11    pushl %ebp
12    pushl %ebx
13    pushl %edi
14    pushl %esi
15    pushfl
16
17    /* switch stack */
18    movl  %esp, TASK_threadsp(%eax)
19    movl  TASK_threadsp(%edx), %esp
20
21    popfl
22    /* restore callee-saved registers */
23    popl  %esi
24    popl  %edi
25    popl  %ebx
26    popl  %ebp
27
28    jmp __switch_to
29  SYM_CODE_END(__switch_to_asm)
30  .popsection
```

Listing 4. Context Switch using software

After the context switch happened clean up work must be done, such as releasing locks and the new task takes over.

5.3 Task switching using continuations

In [KS07] the authors describe how continuations could be used to perform system calls.

```
1   data Req r = Exit -- destruction of the process
2              | Read (Char -> CC r (Req r))
3              | Write Char (() -> CC r (Req r))
4
5   interpret world pcb Exit = do liftIO (sClose (
        psocket pcb))
6                                 return world
7   interprt world pcb (Read k) = return world {jobQueue
        = jobQueue world ++ [JQBlockedOnRead pcb k]}
8
9   service p req = shiftP p (\k -> return (req k))
10
11  cat p = do input <- service p Read
12             service p (Write input)
13             cat p
```

Listing 5. Simplified system calls structure

The `Read` request from Listing 5 contains a continuation that should be executed after the successful read. The type of the continuation shows that it should consume the read character and the yielding a new continuation as an answer. This is normally the `Exit` request to terminate the process. The `Write` request works similarly but writes a character and executes a continuation that does not have any input arguments.

Using the function `service` a user application can perform a system call and therefore a context switch. Using this simple interface, one can write a really basic user application `cat`.

The system calls are handled in the `interpret` function that is invoked by the scheduler. The scheduler passes an old `world` and expect a new `world` from `interpret`. The `world` data structure would be the whole state of the operating system, such as job queue. Additionally, to the `world` the `interpret` takes the *process control block (PCB)* that describes various resources that are allocated to the current process as well as the request from the user application.

In the House operating system, they also use a continuation like system.[HJLT05] While continuations can be a good abstraction, in the end one has to resemble to C / assembly to change the register on the CPU itself. The alternative would be to use the hardware context switch mechanism provided by the Intel CPUs. However, this would limit the written software to Intel CPUs and can not be run anywhere else.

6 Kernel API and user applications

An application developer should never bother how the kernel performs memory management or do I/O. For example, if the application develop wants to read a file from the file system, he will instruct the kernel to read the file and returning its

content. This is usually done using so called *system calls*, the API to the kernel.

6.1 UNIX / POSIX API

Most operating systems provide a standard C library called `libc`. Many system calls are hidden behind such a library / API.

To open a file using the `libc` under Linux, one could use the open function.

```
int open(const char *pathname, int flags, ...
                    /* mode_t mode */ );
```

Listing 6. Open a file using libc

From the man page:

> On success, `open()`, `openat()`, and `creat()` return the new file descriptor (a non-negative integer). On error, -1 is returned and `errno` is set to indicate the error.

This function has two problems: the return value is a `int` that acts as a file descriptor and the error code at the same time. Also, nothing prevents the developer to use the error code, a negative integer, to use as a file descriptor.

In the next section, I want to show how one could design an API that is more expressive and forces the developer to check the return value.

6.2 Alternative kernel API

In subsection 5.3 I already discussed how [KS07] modelled a minimal system call API using continuations in Haskell. Such an API could be easily extended to support all kinds of system calls. Because Haskell is a strongly typed language and its type system is more expressive than C's, failures could also be modelled within the type system level.

To indicate that a read could fail, one could use the type `Either`.

```
type Result a = Either Error a
data Error
data FileDescriptor
data Req r = Exit -- destruction of the process
           | Read (Result Char -> CC r (Req r))
           | Write Char (Result () -> CC r (Req r))
           | Open String (Result FileDescriptor ->
                CC r (Req r))
```

Listing 7. Model failure on type system level

As one can see in Listing 7, in the case of a successful read, the kernel would call the continuation using a `Right` and a character. In the error case `Left` and an error type could be used.

The system call to open a file can be modeled in the same way. In contrast to the Linux version one is forced to look at the result and can not access the file descriptor without prior checks, something that is not possible in standard C.

While it is not feasible to change the fundamental library written in C to a library written in Haskell, one could still use some design principles from Haskell. The function signature for open could be changed to something in like in Listing 8. While the compiler does still not enforce checking the return value for success, the return value is not a plain integer anymore but capsuled inside a result structure together with the information for success or failure.

```
struct Result {
  int success; // 0 = false, otherwise true
  int errno;
  int fileDescriptor;
}

Result open(const char * pathname, int flags, ...
          /* mode_t mode */ ) ;
```

Listing 8. Alternative open in C

7 Relevancy

Computer Systems are a fundamental part of our daily lives (Windows, Linux, OS X, ...) as they power our everyday computers and smartphones. It is therefore helpful and useful to have a basic understanding of how our devices work. This is one part I have tried to achieve in this article.

I have also tried to show how certain parts of an operating system can be implemented in a functional programming language or using techniques that are common using functional programming. As the user expected that the devices to function reliable in every situation and at all times, the system must work as smoothly as possible. In certain use cases, it is even necessary for the entire software stack to be verified as correct. Such applications can be in the aerospace or medical sectors. In such cases, the seL4 kernel could be a possible option, as it is already verified[KEH+09].

However, proving a system written in an imperative language can be a difficult task. Since many imperative languages like C allow side effects during the execution of a function, it is not guaranteed that the same function will produce exactly the same system as an output for a given input. In a pure programming language like Haskell, one tries to avoid site effects wherever possible. If it is nevertheless necessary, one can only perform site effects explicitly and note this in type system. In addition, the side effects are limited to all input and output operations. All other functions can be proven with the help of proof assistants such as *Isabelle/HOL* or *Coq*.

8 Conclusion

It is not possible today to write the complete operating system in a language like Haskell. The direct interactions with the hardware make it impossible to use Haskell alone for such a task. However, it is possible to write operating systems mostly in another language than C/C++, and fallback to C/C++ where required, as seen in the following selection of non C/C++ based operating systems:

- Redox using Rust
- MirageOS using OCaml

- House using Haskell (abandoned)

But where pure languages like Haskell are a good fit is to implement a *executable specification* for verification purposes.

The *abstract specification* describes *what* a given system should do, without making any assumption how it should do it. Using Haskell, one could then write a *executable specification*. Its purpose is to fill in the detail that are left open in the abstract specification. The last step is to write an *implementation*. Such an implementation could be written using C. The crucial task here is the translation from C to a proof assistant as Isabelle/HOL. The developer of the seL4 kernel used this approach.

Another strategy could be to keep the low-level interactions in C, but all hardware independent task could be written in a more high-level language. In this case, the high-level language would call the C functions through a foreign function interface. This way, only a small portion of code must be written in C. This approach was used in the development of the House operating system.

A third possible approach is to write the code in any high-level language and compile it to C.

In section 5 I described how multitasking and context switches works. One can build great abstractions with the help of continuations. However, the essential work for the switching still has to be done with assembler, as the CPU registers have to be changed manually. Unless, one uses the hardware mechanism, which is only provided by Intel CPUs, to switch context, as the update of the registers is done automatically by the CPU.

In subsection 6.2 I showed how a more expressive API could be designed, that forces the API user to check the return values. However, such an API is only possible if the functions to handle system calls are also written in Haskell. Otherwise, one would only write a thin abstraction layer around the real system calls.

In general, one can say, it is not required to write a complete operating system using C/C++. While one will always must write some minimal assembler instructions, these can always abstracted away using FFI. A promising alternative to C/C++ is Rust. As Rust allows one to directly interact with the hardware, it also has an expressive type system and many useful features from the ML programming language family such as pattern matching that allows to write simple and expressive code.

References

[Bai] Danilo Baio. Chapter 7. Virtual Memory System. https://docs.freebsd.org/en/books/arch-handbook/vm/.

[FSDG08] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 170–182, New York, NY, USA, June 2008. Association for Computing Machinery.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 116–128, New York, NY, USA, September 2005. Association for Computing Machinery.

[HPD] HP Desktop PCs - Computer beeps or a light blinks during startup | HP® Support. https://support.hp.com/us-en/document/ish_1997210-1528385-16.

[Int] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220, Big Sky Montana USA, October 2009. ACM.

[KS07] Oleg Kiselyov and Chung-chieh Shan. Delimited Continuations in Operating Systems. In Boicho Kokinov, Daniel C. Richardson, Thomas R. Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context*, volume 4635, pages 291–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[Mul] Multiboot Specification version 0.6.96. https://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[PZR+17] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified Low-Level Programming Embedded in F*. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, August 2017.

[ROG] ROG Strix X570-E Gaming | Motherboards | ROG Global. https://rog.asus.com/motherboards/rog-strix/rog-strix-x570-e-gaming-model/helpdesk_manual/motherboards/rog-strix/rog-strix-x570-e-gaming-model/.

[YT21] Shenghao Yuan and Jean-Pierre Talpin. Verified functional programming of an IoT operating system's bootloader. In *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '21, pages 89–97, New York, NY, USA, December 2021. Association for Computing Machinery.