1 Introduction

Writing a kernel from scratch is not an easy task. There is a lot of information available on writing kernels, but it is either scattered, not detailed enough or too technical (like the intel manuals). In this paper we have collected the most important information on writing a kernel.

The first three sections explain how to set up a minimal kernel, that is a kernel capable booting up. In the following sections extra functionality is added to the minimal kernel. First a video driver is added, followed by software/hardware interrupt support, keyboard support, memory management (paging) and process support.

We implemented the video driver first, because we needed a way to display debug messages. After this we implemented interrupt support, because all other components in an operating system on the Intel architecture rely on interrupts.

2 Tool Used

For this project we picked the most widely used and free programs/tools to create the kernel. Our main reason for this was the availability of documentation and tutorials.

The easiest way to write a kernel is to use a high level programming language. We chose C, because we were both familiar with it and it is the most used language for writing kernels. The most obvious tool for this is **gcc**. It is of course not possible the write an entire kernel with a high level language. For low level functionality we use **nasm**.

The kernel has to be continuously tested. Instead of rebooting a real computer, we use **bochs** to emulate an 386 PC. For booting purposes we use the **GRUB** bootloader.

3 Minimal Kernel

The first step in creating a kernel is to set up a minimal kernel that does absolutely nothing. For this three things need to be done:

1. a kernel entry file (`start.asm`) has to be set up,

2. a linker script (`link.ld`) should be created and

3. a Makefile should be created

It would be ideal if we could implement the entire kernel in C, but unfortunately this is not possible. Things like loading the IDT or GDT (discussed later) can only be done in assembly. In our kernel all of assembly stuff is put in a single kernel entry file called `start.asm`. This file initializes the kernel and calls main function in C. See Fig. 1 for a minimal implementation. Note that the file does not contain a booting option yet. Booting is discussed in section 4.

The function `main`, called by `start.asm`, is implemented in file `kernel.c`, which can contain something really simple like the code in Fig. 2. Afterwards the kernel starts looping infinitely.

The linker makes sure assembly code and C code are put together in a single binary file. The linker needs a linker script to know how the files are linked together. Fig. 3 contains the linker script we used to link the files. The script defines the following three sections:

- text: contains the executable itself
- data: used for hardcoded values
- bss: consists of uninitialized data

Everything is put together with the Makefile in Fig. 4.

4 Booting Process

Now that we have a basic kernel, we need a way to boot it up. Our first option is to make a custom bootloader. There are two important rules for writing a bootloader:

1. The bootimage has to be exactly 512 bytes, because the BIOS automatically loads the fist sector, which is 512 bytes long, of the bootdisk. The first 512 bytes of a disk is also called the master boot record(MBR). The BIOS loads the bootimage into memory location 0x7c00.
2. The bootimage must end with 0x55 in byte 511 and 0xAA in byte 512.

The biggest problem we had with a custom bootloader was that we didn't know how to link the bootloader to the kernel. We tried to compile the

```
;=============
SECTION .text
;=============
[BITS 32]
[GLOBAL start]
[EXTERN main]

EXTERN code, bss, end

start:
    ; start C code
    mov esp, stack
    call    main
    ; shouldn’t return from main; if we do, just do an idle loop
    sti
.idle:
    hlt
    jmp .idle

;=============
SECTION .data
;=============

; set up things like IDT, GDT or ISR/IRQ definitions

;=============
SECTION .bss
;=============

; add other stacks like timerstack

stack_start:
        resd 1024
stack:
```

Figure 1: start.asm

```
int main (void) {
    // do some work here, like initialize timer or paging
    return 0;
}
```

Figure 2: kernel.c

```
OUTPUT_FORMAT("elf32-i386")
ENTRY(start)
phys = 0x00100000; SECTIONS {
  .text phys : AT(phys) {
    code = .;
    *(.text)
    *(.rodata*)
    . = ALIGN(4096);
  }
  .data : AT(phys + (data - code))
  {
    data = .;
    *(.data)
    . = ALIGN(4096);
  }
  .bss : AT(phys + (bss - code))
  {
    bss = .;
    *(.bss)
    . = ALIGN(4096);
  }
  end = .;
}
```

Figure 3: link.ld

```
CC=gcc

ifndef CFLAGS
CFLAGS=-g -ggdb -Wall -ansi -pedantic -ffreestanding
endif

all:    kernel.elf

kernel.elf: start.o kernel.o
    ld -T link.ld -o $@ $^

start.o:    start.asm
    nasm -f elf start.asm -o start.o
```

Figure 4: Makefile

bootloader with the kernel, but the kernel was too big to fit in the bootsector. We decided not to create a custom bootloader and use an existing one instead. We chose GRUB as our bootloader.

4.1 GRUB

An advantage of GRUB is that it supports multiple filesystems: ext2/ext3, JFS, ISO 9660, Minix file system, ReiserFS, XFS, UFS/UFS2, VFAT. We used a FAT12 formatted floppy disk as our main disk. GRUB is to big to completely fit into the MBR, so it uses two stages to load the kernel. Stage 1 is 512 bytes big and is located in the MBR. Stage 1 is used to load Stage 1.5 or Stage 2 directly. Stage 1.5 is located in the first 30 kilobytes after the MBR. Stage 1.5 loads Stage 2. Stage 2 displays the boot menu and loads the kernel after a selection by the user.

4.2 Installing GRUB

Two floppy disks are needed to install GRUB. The first one contains GRUB without a file system. It is used to boot into an environment, where GRUB can be installed on the second floppy, which does contain a file system. The first floppy can be created in UNIX with the following commands:

- cat stage1 stage2 > boot
- cat boot > /dev/fd0

Before installing GRUB on the second floppy, it should be formatted with one of the supported file systems. Stage1 and stage2 should be copied to subdirctory "/boot/grub" (of the second floppy) using the UNIX commands below:

- mount /dev/fd0 /mnt
- mkdir /mnt/boot
- mkdir /mnt/boot/grub
- cp stage1 /mnt/boot/grub
- cp stage2 /mnt/boot/grub

Boot the computer using the first floppy. At the GRUB prompt, eject this floppy and insert the second one. Install GRUB on the second floppy using the `setup(fd0)` command. The first floppy is only used to install GRUB on the second floppy. The second floppy will be used as the main disk, which contains the kernel. After installation stage2 may not be modified in any way.

4.3 Multiboot Kernel

GRUB only boots kernels that have a multiboot header. The multiboot header has to be put at the beginning of the kernel. See Fig. 5 for the mulitboot header. The `code`, `bss` and `end` sections are defined in the linker file (link.ld).

5 Video Output

First we need to write a video driver. Without it we wont be able to see what it is doing. Our kernel only supports basic character printing. Advanced features such as graphics are not supported.

The kernel uses the most basic video output, by the VGA controller. This VGA makes controller writing a video driver is very easy. Every VGA video cart has a piece of memory. If this piece of memory is modified the VGA

```
MULTIBOOT_PAGE_ALIGN   equ 1<<0
MULTIBOOT_MEMORY_INFO  equ 1<<1
MULTIBOOT_AOUT_KLUDGE  equ 1<<16
MULTIBOOT_HEADER_MAGIC equ 0x1BADB002
MULTIBOOT_HEADER_FLAGS equ MULTIBOOT_PAGE_ALIGN |
    MULTIBOOT_MEMORY_INFO | MULTIBOOT_AOUT_KLUDGE
MULTIBOOT_CHECKSUM     equ -(MULTIBOOT_HEADER_MAGIC  +
    MULTIBOOT_HEADER_FLAGS)

EXTERN code, bss, end

ALIGN 4 mboot:
    dd MULTIBOOT_HEADER_MAGIC
    dd MULTIBOOT_HEADER_FLAGS
    dd MULTIBOOT_CHECKSUM

    dd mboot  ; these are PHYSICAL addresses
    dd code   ; start of kernel .text (code) section
    dd bss    ; start of kernel .bss section
    dd end    ; end of kernel BSS
    dd start  ; kernel entry point
```

Figure 5: Multiboot header

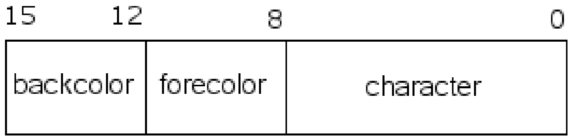


Figure 6: VGA text memory structure

Value	Color	Value	Color
0	Black	8	Dark Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Light Brown
7	Light Gray	15	White

Table 1: VGA colors.

controller will automatically draw the characters and update the screen. Basic functions that are implemented in our kernel are: print, clear screen, scroll screen and move hardware cursor. The hardware cursor is hardware support for displaying the blinking underscore, which represents the current cursor position.

The text memory is a part of memory that is located at address 0xB8000. The video controller presents this text memory as a 80x25 matrix of 16-bit values. As seen in Fig. 6, each of the 16-bit values is divided into tree pieces. The lower 8 bits is called the 'character byte', which contains the character itself. The upper 8 bits is the 'attribute byte', which is further divided into two parts: Bits 8 to 11 contain the color of the character and bits 13 to 15 contain the background color. Because there are only 4 bits available for each color code, only 16 colors can be defined. All possible colors are listed in Table 1.

5.1 Printing Characters

The video buffer is a long linear array of 16-bit values. The video controller makes it appear to be a 80x25 grid (starting with column 0 and row 0). So if

```
using short *memlocation = (unsigned short*)0xB8000 + index;
*memlocation = character | (attribute << 8);
```

Figure 7: print function

we want to print something on a particular location on the screen, we have to translate that position into the index of the memory array. This can be done using the following equation:

$$index = (rowposition * 80) + columnposition \tag{1}$$

The exact memory location can be calculated by adding the index to memory address 0xB8000. In our kernel we used chars instead of shorts, but the column position in Equation 1 is expressed in shorts, so we need to use a factor 2 to correct for this.

Example We want to display a light gray `c` character with black background in the bottom right of the screen.

First we have to determine the memory location of the bottom right of the screen by using the equation mentioned above. The index is $(24 * 80) + 79 = 1999$ ($0x7CF$ in hexadecimal), so the memory location is 0x7CF+0xB8000 $=$ 0xB87Cf.

The color code of black background and light gray foreground is 00000111 (0x07 in hex). This value has to be concatenated with the character and written to the memory location. The C code in Fig. 7 is a rough implementation of the print function. `memlocation` points to the display location, where the character has to be written.

6 Global Descriptor Table

The Global Descriptor Table (GDT) is a structure which defines base access privileges for certain parts of the memory. The CPU uses the GDT to perform safety and access control checks to see if an operation may be performed on a certain piece of memory. The GDT consists of a list of descriptors.

GRUB already install a GDT, but because its contents is undefined, we must set up our own GDT.

The GDT contains descriptors which define what rights are assigned to a part of the memory. For instance, when a descriptor is used defining ring 0,

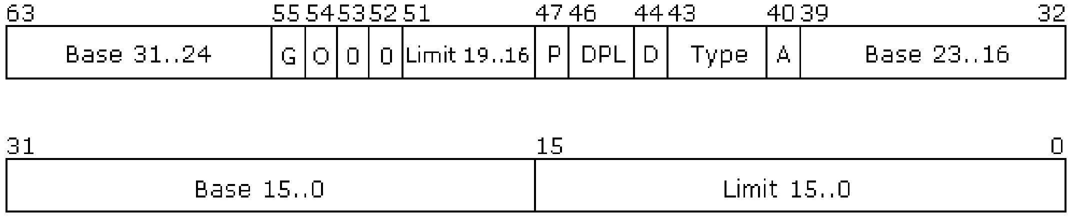


Figure 8: GDT descriptor

defining the most privileged mode, no user-mode program may read or write it.

We don't want to use this system of GDT descriptors, as it needs a lot of manual memory handling (to setup every descriptor right, for whatever use) and there is no need, as the same kind of protection can be created by the paging mechanism.

There is no way to really disable the use of the GDT, but there is a way to setup descriptors such that we don't have to worry about the descriptors anymore. It is setting up a maximum descriptor of 4 gigabyte, both for kernel mode and user mode, and both for data memory and for code memory.

6.1 GDT Descriptor

Each descriptor is 64 bits long. The format is shown in Fig. 8. A descriptor contains:

- Bits 63-56: Bits 32-24 of the base address.
- Bit 55: Granularity bit (0 = 1 byte, 1 = 4 kbyte). If set to 1 then the limit is multiplied by 4k.
- Bit 54: Operand size (0 = 16-bit, 1 = 32-bit). Set 16/32-bit
- Bit 53: Reserved, always zero.
- Bit 52: Reserved for OS. Set to zero.
- Bits 51-48: Bits 19-16 of the limit (max size)
- Bit 47: Present bit. The segment is present in memory. Should be set to 1.

- Bit 46-45: Descriptor privilege level (0 = highest, 3 = lowest). Sets the ring level. Ring 0 is used for kernel mode and ring 3 is used for user mode. Ring 2 is used in our kernel.
- Bit 44: Descriptor bit (0 = system descriptor, 1 = code/data descriptor).
- Bits 43-41: The descriptor type contains:
  - Bit 43: executable (0 = data segment, 1= code segment)
  - Bit 42: expansion direction (for data segments), conforming for code segments
  - Bit 41: read/write. For data segments: 0 = read only, 1 = read & write. For code segments: 0 = execute only, 1 = read & execute.
- Bit 40: Accessed bit (for use with virtual memory).
- Bits 39-16: Bits 23-0 of the base address.
- Bits 15-0: Bits 15-0 of the segment limit.

6.2 Descriptors Used

Fig. 9 shows the contents of the GDT. Our GDT has four types of descriptors:

- `The NULL descriptor`: The first entry of the GDT always has to be a NULL descriptor. The NULL descriptor is never referenced by the processor. If this does happen, than a General Protection fault will be raised. The assembly code is shown in Fig. 10. The descriptor is called a NULL descripter, because it consists only of zeros.
- `Code segment descriptors`: Our kernel has two code segment descripters: one for kernel mode (Fig. 11) and one for user mode (Fig. 12). These descripters are used for read privileges. Write operations are blocked.
- `Data segment descriptors`: Again there are two descriptors: one for kernel mode (Fig. 13) and one for user mode (Fig. 14). These descriptors are used for write privileges. Both read and write operations are allowed.

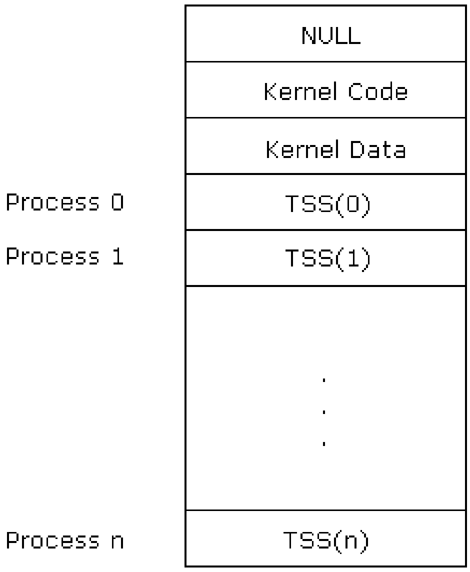


Figure 9: GDT layout

```
NULL_SEL        equ     $-gdt
    dw 0            ; limit 15:0
    dw 0            ; base 15:0
    db 0            ; base 23:16
    db 0            ; type
    db 0            ; limit 19:16, flags
    db 0            ; base 31:24
```

Figure 10: NULL_SEL definition

```
SYS_CODE_SEL    equ $-gdt
    dw 0FFFFh
    dw 0
    db 0
    db 10011010b    ; present,ring 0,code,non-conforming,readable
    db 11001111b    ; page-granular (4 gig limit), 32-bit
    db 0
```

Figure 11: SYS_CODE_SEL definition

```
USER_CODE_SEL   equ ($-gdt)|0x3
    dw 0FFFFh
    dw 0
    db 0
    db 11111010b    ; present,ring 3,code,non-conforming,readable
    db 11001111b    ; page-granular (4 gig limit), 32-bit
    db 0
```

Figure 12: USER_CODE_SEL definition

```
SYS_DATA_SEL    equ $-gdt
    dw 0FFFFh
    dw 0
    db 0
    db 10010010b    ; present, ring 0, data, expand-up, writable
    db 11001111b    ; page-granular (4 gig limit), 32-bit
    db 0
```

Figure 13: SYS_DATA_SEL definition

```
USER_DATA_SEL   equ ($-gdt)|0x3
    dw 0FFFFh
    dw 0
    db 0
    db 11110010b    ; present, ring 3, data, expand-up, writable
    db 11001111b    ; page-granular (4 gig limit), 32-bit
    db 0
```

Figure 14: USER_DATA_SEL definition

- `TSS segment descriptors`: The TSS is used in hardware-based multitasking. This is discussed in chapter 11.2

7 Interrupt Handling

7.1 Interrupts

Interrupts are used to temporarily suspend the program that the CPU is currently running and run another program. After termination of the program, the original program that was suspended will continue running. There are three types of interrupts:

Software interrupts Software interrupts are caused by the software. Every time the program gets to a point where there is an interrupt instruction, an interrupt is issued. Software interrupts can happen for example when a program calls an operating system function for opening a file. These interrupts are user generated.

Hardware interrupts Hardware interrupts come from the hardware itself. Hardware devices use these interrupts to communicate with the processor. An example is the keyboard interrupt. Every time a key is pressed a keyboard interrupt is send.

Exceptions Exceptions come from the processor itself. An exceptions occurs when the processor doesn't know how to solve an internal error caused by software. All possible exceptions are listed in Table 2 (except for `32` to `255`, which can be used for soft- and hardware interrupts). The last column indicates whether the exception automatically pushes an error code onto the stack. If no error code is pushed, a dummy code (for example zero) should be manually pushed. The error code gives more information on the cause of the error.

7.2 Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is used to tell the CPU what to do when a particular error/interrupt occurs. It contains IDT descriptors, which associate interrupts and exceptions with Interrupt Service Routines (ISR). The IDT contains 256 descriptors, allowing a total of 256 different interrupts.

Exception #	Description	Error Code
0	division by zero	no
1	debug	no
2	non maskable interrupt	no
3	breakpoint	no
4	overflow	no
5	out of bounds	no
6	invalid opcode	no
7	coprocessor (FPU) unavailable	no
8	double fault	yes
9	math unit protection fault	no
10	invalid TSS	yes
11	segment not present	yes
12	stack fault	yes
13	general protection fault	yes
14	page fault	yes
15	unknown/reserved	no
16	floating point error	no
17	alignment check (486+)	no
18	machine check (Pentium/568+)	no
19	extreming SIMD extensions	no
20-31	reserved exceptions	no
32-255	available for soft- and hardware interrupts	no

Table 2: exception list

If the IDT has more than 256 descriptors, all entries above 256 are ignored. If there are less descriptors and one of the missing interrupts is raised, then a General Protection Fault will be raised.

In our kernel the macro in Fig. 16, which is used to create an IDT descriptor, is called 256 times (see Fig. 17) to construct the IDT. The layout of the IDT descriptor is described in the following section (section 7.2.1).

During the creation of the IDT the addresses of the handlers are unknown, because the handlers do not exist yet. The address fields of the IDT descriptors are filled with dummy values during creation and after creating the handlers they are filled with the correct handler addresses. The creation of the handlers is covered in section 7.3.

After filling the address fields, the IDT is loaded using the command: `lidt [idt_ptr]`.

7.2.1 IDT Descriptor

The IDT may contain any of three kinds of descriptors:

- Task Gates (described in section 11.2.3)
- Interrupt Gates
- Trap Gates (not used in our kernel)

The layout of the IDT descriptor is shown in Fig. 15. It contains the following fields:

- Gate Type:
  - 00101 for Task Gate
  - 01110 for Interrupt Gate
  - 01111 for Trap Gate
- P: Segment present flag.
- DPL: Descriptor Privilege Level.
- Offset: Address of the interrupt handling routine. For backward compatibility reasons the offset is split in two parts.
  All offsets are zeroed out during creation. The correct address of the handler is inserted afterwards.

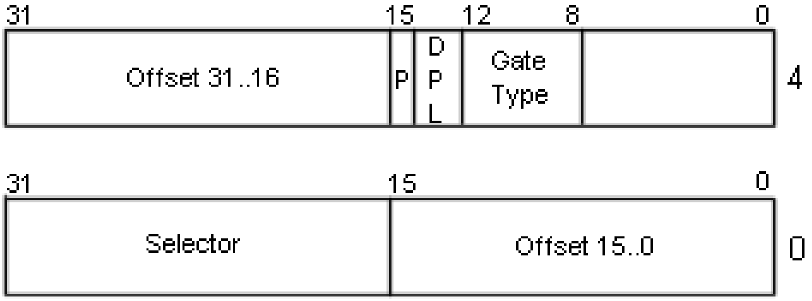


Figure 15: IDT descriptor

- Selector: Code segment (see code/data selectors in GDT) to run the interrupt handler in.
- All other bits are reserved for future use.

Descriptors 0x80 and 0x20 are Task Gates. All other descriptors are Interrupt gates (see Fig. 16). Trap Gates are not used in our kernel.

7.3 Interrupt Service Routines

In the previous two sections it was explained how to build an IDT table. During the creation of the IDT descriptors all address fields where zeroed out, because they were unknown at the time. This section explains how to set up the handlers (ISRs) and fill the IDT with the correct ISR address.

Fig. 18 shows the global overview of the interrupt handler setup. The layout is divided into an Assembly section and a C section. The C section contains the ”real” handlers. There are three different handlers:

- `interrupt(regs_t *regs)`: this interrupt handler is used most often. It is used to handle exceptions and hardware interrupts.
- `timer_handler()`: used to handle the timer.
- `syscall_handler()`: used to handle systemcalls from usermode.

See section 7.5 for more information on the implementation of these handlers.

The ISRs in the Assembly section are used to call the correct hander in C and restore the register states afterwards. In order to construct the ISRs, the

```
%macro SETIDT 1
idt%1:
%if %1 = 0x80
        dw 0                    ; unused
        dw INT_TSS_SEL          ; TSS segment selector
        db 0                    ; unused
        db 11100101b            ; present, ring 3, task gate
        dw 0                    ; unused
%elif %1 = 0x20
        dw 0                    ; unused
        dw TIMER_TSS_SEL        ; TSS segment selector
        db 0                    ; unused
        db 10000101b            ; present, ring 0, task gate
        dw 0                    ; unused
%else
        dw 0x0                  ; not known yet; filled in at execution
        dw SYS_CODE_SEL         ; code selector that will be used
        db 0                    ; 0 for interrupt gates
        db 10001110b            ; 1-00-01110
        dw 0x0                  ; not known yet; filled in at execution
%endif
%endmacro
```

Figure 16: IDT descriptor macro

```
idt:
%assign i 0
%rep 256
        SETIDT i
%assign i (i + 1)
%endrep
idt_end:
```

Figure 17: IDT

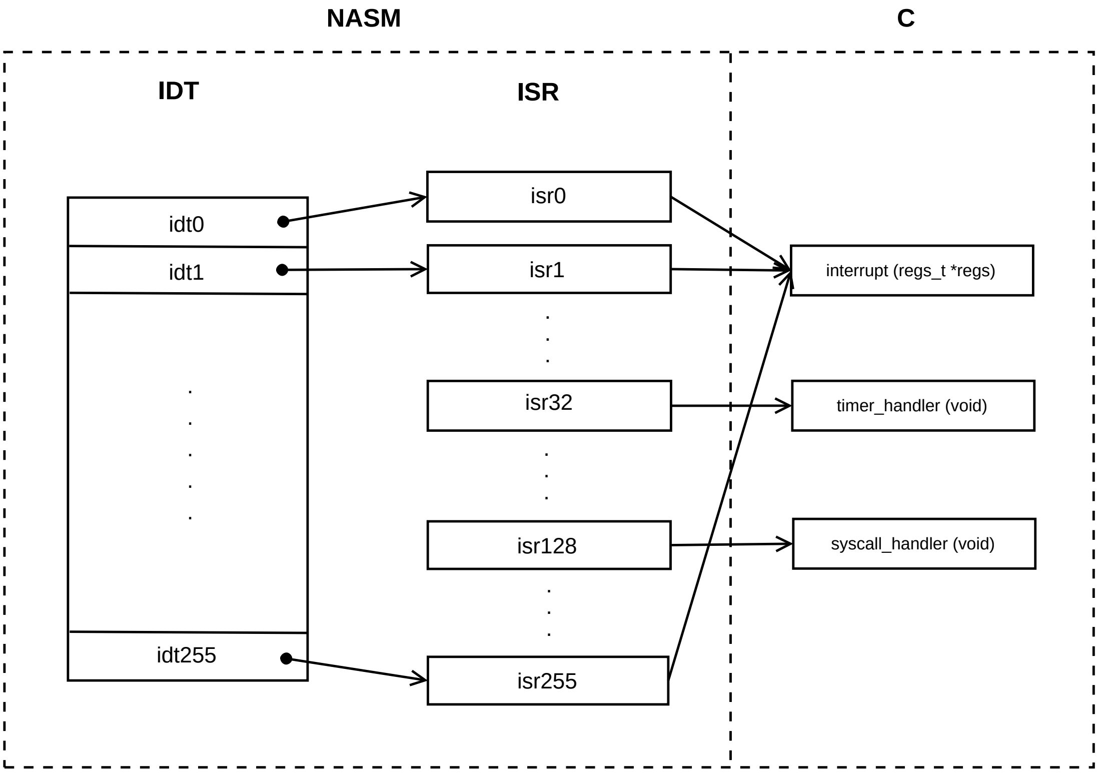


Figure 18: interrupt handling layout

```
%macro TSS_BODY 2
%define label %1
%define handler %2
        call handler
    add esp,4           ; drop error code
%if label = 0x20
        mov al, 0x20
        mov dx, 0x20
        out dx, al
%endif
    iret
; as a task gate rather than an interrupt gate, the line after the iret
; is stored in the eip; solve the problem by jumping back into the isr
        jmp isr %+ label
%endmacro
```

Figure 19: TSS_BODY

code in Fig. 20 is called 256 times. (see Fig. 21). The macro takes two parameters: The ISR number and whether the interrupt pushes its own error code (see `Error Code` field in Table 2). If no error code is automatically pushed, then a dummy error code (zero) is pushed manually. The first parameter, the ISR number, is used to determine which handler to call. All ISRs call `interrupt(regs_t *regs)`, except for `isr32`, which calls `timer_handler()` (using Fig. 19), and `isr128`, which calls `syscall_handler()`.

If handler `interrupt(regs_t *regs)`, which requires a `struct regs` pointer as parameter, is used then the a number of registers should be pushed on the stack before calling the handler. The struct definition is given in Fig. 22. Please read the comments in the code to see which registers have to be put on the stack. Note that a pointer to the stack should be passed as parameter, to enable the handler to modify the stack content. The `regs_t` structure exactly matches the contents of top of the stack.

After execution of the handler code all registers are restored by popping the values off the stack.

Everything is now in place to install the ISRs in the IDT descriptors. In Fig. 23 the macro `SETISR` is used to install the ISRs in their corresponding IDT descriptor. Interrupts `0x20` and `0x80` are skipped, because

```
; implementation of one ISR, depending on the ISR number (argument 1)
and ; whether the given ISR needs a dummy error code (argument 2)
%macro INTBODY 2
isr%1:
%if %2 = 0
        push dword 0                     ; "fake error code"
%endif
%if %1 = 0x20                            ; isr32
        TSS_BODY %1, timer_handler
%elif %1 = 0x80                          ; isr128
        TSS_BODY %1, syscall_handler
%else
        push dword %1                   ; exception number
    push gs             ; push segment registers
    push fs             ;
    push es             ;
    push ds             ;
    pusha               ; push GP registers
    mov ax,SYS_DATA_SEL         ; put known-good values...
    mov ds,ax               ; ...in segment registers
    mov es,ax               ;
    mov fs,ax               ;
        mov gs,ax               ;
    mov eax,esp             ;
    push eax                ; push pointer to regs_t
        call interrupt
    pop eax
    popa                ; pop GP registers
    pop ds              ; pop segment registers
    pop es
    pop fs
    pop gs
    add esp,8           ; drop exception number and error code
    iret
%endif
%endmacro
```

Figure 20: INTBODY

```
;; now declare isr’s
        INTBODY  0, 0
        INTBODY  1, 0
        INTBODY  2, 0
        INTBODY  3, 0
        INTBODY  4, 0
        INTBODY  5, 0
        INTBODY  6, 0
        INTBODY  7, 0
        INTBODY  8, 1
        INTBODY  9, 0
        INTBODY 10, 1
        INTBODY 11, 1
        INTBODY 12, 1
        INTBODY 13, 1
        INTBODY 14, 1
        INTBODY 15, 0
        INTBODY 16, 0
        INTBODY 17, 1
        INTBODY 18, 0
        INTBODY 19, 0

%assign i 014h
%rep 0FFh - 013h
        INTBODY i, 0
%assign i (i + 1)
%endrep
```

Figure 21: ISR declarations

```
/* the layout of this structure must match the order of registers
   pushed and popped by the exception handlers in START.ASM */
typedef struct regs {
  /* pushed by pusha */
  unsigned edi, esi, ebp, esp, ebx, edx, ecx, eax;
  /* pushed separately */
  unsigned ds, es, fs, gs;
  unsigned which_int, err_code;
  /* pushed by exception. Exception may also push err_code.
     user_esp and user_ss are pushed only if a privilege change occurs. */
  unsigned eip, cs, eflags, user_esp, user_ss;
} regs_t;
```

Figure 22: regs struct

they are `task gates` instead of `interrupt gates`. Only the fields of an `interrupt gate` has to be filled, because the address in a `task gate` is already known in the TSS. A `task gate` descriptor refers to a TSS descriptor, which refers to a TSS.

After installing the ISR addresses in the IDT, the IDT can be loaded with: `lidt [idt_ptr]`.

7.4 IRQ

Interrupt Requests (IRQs) are interrupts that are raised by hardware devices. Instead of continuously polling the device, the device itself generates an IRQ to get the CPU's attention. A complete list of IRQs is listed in Table 3.

IRQs 0 to 7 are normally mapped to interrupts 8 to 15. In the previous section, we installed the exception handlers on interrupts 0 to 31. This will cause problems, because now whenever an IRQ is fired the kernel will think a completely different exception has occurred. For example, every time IRQ 0 fires a Double Fault Exception will be raised. To prevent the collision between IRQs and exceptions, IRQs 0 to 15 have to be remapped to IDT entries 32 to 47 (since entry 32 is directly after the last exception).

Remapping of IRQs can be done by sending commands to the Programmable Interrupt Controller (PIC). There are two PICs on a system: a master and a slave PIC. The slave is connected to IRQ 2 on the master

```
; store isr addresses in idt fields and load idt
%macro SETISR 1
%if %1 != 0x80 && %1 != 0x20
        mov     eax, isr%1
        mov     [idt%1], ax
        shr     eax, 16
        mov     [idt%1 + 6], ax
%endif
%endmacro

%assign i 0
%rep 256
        SETISR i
%assign i (i + 1)
%endrep
```

Figure 23: install ISR addresses

controller. The master is connected directly to the CPU. Each PIC handles 8 IRQs. The master handles IRQs 0 to 7 and the slave IRQs 8 to 15. Every time an IRQ from the slave PIC fires, IRQ 2 on the master PIC also fires, because the slave is connected to the master through that IRQ. All possible IRQs are listed in Table 3.

The remapping of the IRQs is done with the function `void remap_pics()` (Fig. 27). It first initializes and remaps the PICs by sending the correct values (Fig. 24) to the controllers using the function `static void init_pics(int pic1, int pic2))` (Fig. 26), and then enables the IRQ with the function `void enableIRQ(void)` (Fig. 25).

7.5 Handlers

7.5.1 interrupt(regs_t *regs)

The implementation of `interrupt(regs_t *regs)`, which handles the exceptions given in Table 2, is as follows.

The function first determines whether the interrupt is an exception or an IRQ by checking the interrupt number in `which_int`. If `which_int` is

IRQ #	Description
0	System timer
1	Keyboard
2	Second IRQ controller
3	COM 2(Default) COM 4(User)
4	COM 1(Default) COM 3(User)
5	Sound card (Sound Blaster Pro or later) or LPT2(User)
6	Floppy disk controller
7	LPT1(Parallel port) or sound card (8-bit Sound Blaster and compatibles)
8	Real time clock
9	ACPI SCI or ISA MPU-401
10	Free / Open interrupt / Available
11	Free / Open interrupt / Available
12	PS/2 Mouse
13	Math co-processor.
14	Primary IDE
15	Secondary IDE

Table 3: IRQ list

```
#define PIC1 0x20
#define PIC2 0xA0
#define PIC1e (PIC1 + 1)
#define PIC2e (PIC2 + 1)
#define ICW1 0x11
#define ICW4 0x01
#define PICenable  0x00
#define PICdisable 0xff
```

Figure 24: PIC remap definitions

```
/* enable all possible IRQ interrupts */
void enableIRQ(void) {
    outportb(PIC1e,PICenable);
    outportb(PIC2e,PICenable);
    sti();
}

/* disable all IRQs */
void disableIRQ(void) {
    outb(PIC1e,PICdisable);
}
```

Figure 25: enable/disable IRQs

```
/* init_pics()
 * init the PICs and remap them
 */
static void init_pics(int pic1, int pic2) {
    /* send ICW1 */
    outb(PIC1, ICW1);
    outb(PIC2, ICW1);

    /* send ICW2 */
    outb(PIC1e, pic1);  /* remap */
    outb(PIC2e, pic2);  /*  pics */

    /* send ICW3 */
    outb(PIC1e, 4); /* IRQ2 -> connection to slave */
    outb(PIC2e, 2);

    /* send ICW4 */
    outb(PIC1e, ICW4);
    outb(PIC2e, ICW4);
}
```

Figure 26: initialize PICs

```
/* remap IRQs to 0x20-0x2f */
void remap_pics(void) {
    init_pics(0x20,0x28);
    enableIRQ();
}
```

Figure 27: PIC remap

between (and including) 32 and 47, then the interrupt was caused by an IRQ and the IRQ handler is called.

Both the exception and the IRQ handler use a `switch` on `which_int` to select the correct code for handling the interrupt. In the current implementation only exceptions 0, 13 and 14 are handled and only IRQ 1 and 7 are handled. Handlers for other exceptions or IRQs can be added easily by adding a `case` statement with the correct interrupt number followed by the handler code.

The exception handlers are implemented as follows:

0 We try to handle exception 0, the division by 0 exception, by skipping the division instruction. There are two problems with this:

   1. We currently are not sure how many bytes to skip.
   2. It may in fact be better to abort the program when a division by 0 exception happens

13 Because of some issues with scheduling code we have had, there is one particular case that is being handled by the general protection exception handler. It is the case that:

   1. A hardware-related event causes the exception;
   2. The TSS where the event happened is the timer TSS;
   3. The timer TSS descriptor is marked busy;

   In short, that means the timer TSS is busy and the hardware timer causes a new timer call. This was necessary as output of the kernel in the bochs environment was slowing things down too much, causing this problem.

```
/* Handles the timer (irq0). In this case, it's very simple: We
 *  increment the 'timer_ticks' variable every time the
 *  timer fires. By default, the timer fires 18.222 times
 *  per second. */
void timer_handler(void) {
  /* Increment our 'tick count' */
  timer_ticks++;

  /* Every 18 clocks (approximately 1 second), we will
   * do something
  if (timer_ticks % 18 == 0)
  {
    /* do whatever you want */
  }
}
```

Figure 28: timer_handler()

14 The page handler. It is very basic, and it is described in Subsection 10.5

7.5.2 timer_handler()

Creating a timer handler is very easy. An implementation of the timer handler is shown in Fig. 28. The most important part of the code is the `timer_ticks++` statement. `timer_ticks` is a global variable, which is incremented every time a timer interrupt is fired. Timer interrupts are generated by the Programmable Interval Timer, also known as the System Clock. More information on the System Clock can be found in section 8.

7.5.3 syscall_handler()

The following system calls have been implemented:

1: exit The current process is removed from available processes in the scheduler and another process gets scheduled. Details depend on the scheduler used

2: fork A new process is created. First the process fields are initiated such that it is a copy of the current process (in another part of memory), then a paging directory is setup for the new process, and finally the process gets added to the available processes in the scheduler. If necessary rescheduling happens, typically if the newly created process has to get started immediately. A value indicating the process number is returned. The new process returns 0.

The fork system call takes a number, which is period information used by a real-time scheduler.

3: read This handler fakes reading from a file. It will fill in the given buffer with random data. It will also call `restart_process` to tell the scheduler to (re-)activate the process; It is best to think of it as a start-cycle call. The file number must be one given be syscall open (number 5).

It simulates an EOF if a read counter has become zero.

4: write This handler fakes writing to a file. It does the following, depending on the file given:

"/tmp/out" file It ignores the given buffer, instead it just calls `halt_cycle_process` to tell the scheduler to (re-)activate another process; It is best to think of it as a stop-cycle call.

"/tmp/wout" file It ignores the given buffer, instead it just calls `block_process` to tell the scheduler to delay further execution of the current process.

1 It prints the given buffer to the video output. This simulates writing to the "stdout" file under Linux.

5: open This handler must be given file name "/tmp/sensor" for reading or "/tmp/out" or "/tmp/wout" for writing. Other file names will always result in failing to open. Note that these aren't real files, as the kernel does not implement a file system. Instead, it is just simulated in a case-specific manner in reading/writing calls.

6: close This handler just accepts any file number given by syscall open (number 5) and rejects any other.

8: creat This is a shortcut for an "open" syscall, with options to create a new file (and to open it)

11: execve This handler will execute the program of which the name is given. The given name must be one of “read2”, “read3” and “read4”. This is because our kernel has no file system support, and to solve that, these files have been compiled directly into the kernel image

45: brk This is the handler to request more memory to be given to a process. It assumes the process is held in `currentProcess`.

Note that the semantics are as close to those of the corresponding Linux calls as possible. The reason is that we wanted to enable running Linux programs on our kernel. However the kernel currently lacks enough `ELF` support to actually let it run unchanged Linux executables, because the modern `GCC` compiler uses shared libraries, even for a “hello world” program.

8 Timer

In the previous section the system timer was set to interrupt 32 (0x20). But in order for the timer to work correctly, the Programmable Interrupt Timer (the PIT) has to be initialized with the correct timer interval. The PIT, also called the System Clock, is a chip that generates timer interrupts at regular time intervals. The chip has 3 channels: Channel 0 is tied to IRQ 0 (mapped to interrupt 0x20), Channel 1 is system specific (once used for refreshing the DRAM), and Channel 2 is connected to the system speaker (make the system beep). Our kernel only supports Channel 0.

By default the PIT generates an IRQ 0 18.222 times per second. The code for setting the timer interval is given in Fig. 29. First the command byte 0x36 is send to the command register 0x43. The second step is to send the low and high bytes of the divisor to I/O port 0x40, which is the data port for Channel 0. The divisor is the value by which the input clock, running at a frequency of 1193180Hz (for historical reasons), will be divided to determine how many times per second an interrupt has to be fired.

9 Keyboard

The easiest way to get input from the user is by using a keyboard. Therefore it is important to write a keyboard driver. This is easily done by using the keyboard controller and interrupts. The keyboard controller has 2 main

```
/* set timer phase */
void timer_phase(int hz) {
  int divisor = 1193180 / hz;       /* Calculate our divisor */
  outportb(0x43, 0x36);             /* Set our command byte 0x36 */
  outportb(0x40, divisor & 0xFF);   /* Set low byte of divisor */
  outportb(0x40, divisor >> 8);     /* Set high byte of divisor */
}
```

Figure 29: set timer interval

registers: a Data register at 0x60, and a Control register at 0x64. When a key is pressed the scancode of that key is put in the data register and IRQ 1 (mapped to interrupt 0x21) is raised.

Each key has a unique scancode. Scancodes are numbered from left to right on each row (from top to bottom). The scancode is read from the data register, address 0x64, with `inportb(0x60)`. To convert the scancode to the correct ASCII value we use a lookup table. The lookup table, also called the keymap, is an `unsigned char` array containing the ASCII values ordered according to the scancodes. For simplicity we have assumed that the keyboard has a standard US layout.

We have implemented the driver with two different keymaps: one for normal keys (`keyboardMap[0x80]`) and one for when SHIFT is pressed (`shiftKeyboardMap[]`). If bit 7 of the scancode is set then the key was just released, otherwise the key is still being pressed. Holding a key down will generate repeated key press interrupts. Bit 7 can be tested with `scancode & 0x80` in C.

Converting scancode to ASCII can be done with:
`key = keyboardMap[scancode]`. If the SHIFT key is pressed and held, keymap `shiftKeyboardMap` is used, otherwise `keyboardMap` is used. `keyboardMap` contains capital characters. For the complete implementation of the keyboard driver please refer to the code in file `keyboard.c`.

10 Paging

10.1 Introduction

For a kernel, memory management is something really important. Because modern compilers simply assume the kernel will handle all memory details, even a basic kernel needs to do some memory management.

We wanted to have as little to do with segments in our kernel. Segments are basically outdated, as they are not supported well by compilers; How we got rid of dealing with segments as much as possible, is dealt with in the section on the GDT.

We found we needed to use paging to be able to run programs. Our kernel supports the ELF format, the format most Unix/Linux programs are compiled in. The ELF format tells the loader (in our case, the kernel) at which virtual address some data is to be loaded, and at which virtual address program execution begins. Therefore we need to be able to deal with virtual addresses, and that's how we started using the paging mechanism.

As the kernel has been set up, segments do not play a significant rule, as each segment is the size of the complete memory space (4 GB). This effectively "hides" them, from the point of view of the paging system.

If this were not the case, page tables do not give physical addresses but "lineair addresses". This could be used to enhance security, but it also adds complexity.

10.2 An Explanation of Paging

Paging basically works as a lookup table: a virtual address is given and the table is used to find the corresponding physical (real) address.

In the Intel architecture, pages are always of a fixed size. Each page is 4 KB long. The size of the virtual address space is 4 GB, this means a table is needed of 4 MB, as each entry of the table is a long integer, which has 4 bytes.

To avoid the requirement of having a table this large in memory, the table is split into two tables. The first table takes the left most ten bytes to select the second table. This second table takes the next ten bytes to select the physical address, and the last twelve bytes are used as an offset in this "page" of physical memory.

The first table is called page directory (it is similar to a directory in a filesystem giving access to an individual file) and the second table is called page table. Both page directory and page table contain physical addresses.

As you can see, the way paging works isn't really so straight forward. (though it isn't that difficult in practice). Fortunately, most of this is hidden to the kernel, as it is handled by the hardware. However there are some things the kernel will have to handle, which will be discussed in the next subsections.

10.3 Page Table Entry Description

The page directory entries and page table entries have the same format, as shown in Fig. 30.

The following bits are set:

P The present bit indicates whether the entry is present. If the bit is zero, none of the other bits have a meaning, and the OS may use or set those bits when needed. The entry is seen as indicated in Fig. 31.

R/W The read/write bit indicates whether the page is read only (set to 0) or read/write (set to 1). There is no way to indicate whether a page is executable so that each page is executable (unless using extentions, which we do not use in our kernel)

U/S The user/superuser bit indicates whether the page is used by a user process or by the kernel. If it is used by the kernel, (the bit is set) the R/W bit is in fact ignored, as if it is always set to 1 (unless, again, one uses extensions)

A The access bit indicates whether the page/page table has been accessed (either read or written).

D The dirty bit indicates whether the page/page table has been written to.

avail Bits 9, 10 and 11 are available to the OS for use.

rest A page is always aligned to 4K, such that exactly every page can be selected through page directory and page table. Therefore the lower 12 bits of its start address are always zero. These bits are the control bits for the entry.

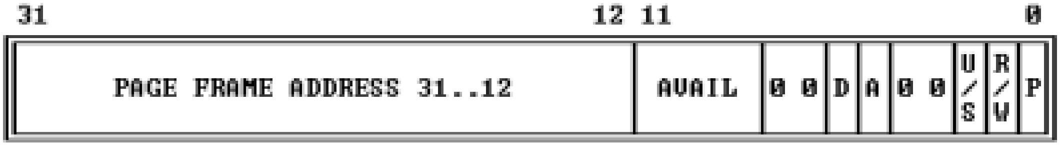


Figure 30: page table entry

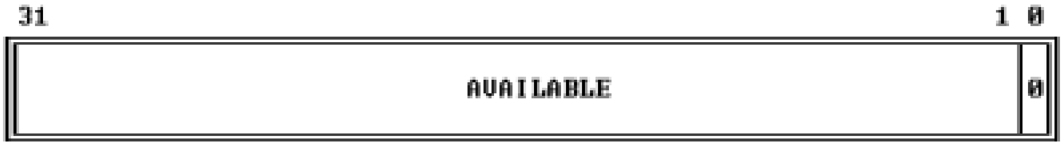


Figure 31: page table entry for unavailable page

The other bits indicate the physical address referenced by the table entry.

10.4 Paging Initialisation

When the kernel starts, paging is not enabled yet; there is no page directory and therefor there are no page tables either.

Before enabling paging, we need to setup a correct page directory, such that the kernel memory is correctly mapped into the page directory. To keep things simple, the first 4 MB of the virtual memory are mapped to the first 4 MB of the physical memory, which includes the memory used for the kernel program and its private data.

We setup the `CR3` register, which indicates the (physical) start address of the page directory to the processor, and setup the page directory such that the first page is not mapped (to catch any `NULL` dereferencing), and the other pages, up to $0x400000$, are setup such that the virtual addresses simply map to physical addresses. Then paging is enabled, by setting the left-most bit of register `CR0` to 1.

10.5 Page Fault Handling

When a virtual address gets referenced (by the kernel or by a process) for which no mapping exists, that is, the relevant page directory entry or the page table entry have the present bit set to zero, exception 14 occurs, the `#PF` (page fault) exception.

In order to handle the page fault, the address which was used, which caused the page fault, is given in `CR2`. Also an error code is provided.

10.5.1 Error Code Format

The processor will provide an error code whenever a page fault occurs. The rightmost bits are:

0. P:

   0 if the fault was caused by using a non-present page,

   1 if the fault was a result of page-level protection violation

1. R/W:

   0 if the fault was caused by a read

   1 if the fault was caused by a write

2. U/S:

   0 if the fault was caused while running in kernel mode

   1 if the fault was caused while running in user mode

3. reversed

4. I/D:

   0 if the fault was not caused during instruction fetch

   1 if the fault was caused during instruction fetch

The rest of the bits are reserved.

10.5.2 Our Way of Page Fault Handling

First, the faulting address is read from CR2.

The fork system call uses the copy-on-write mechanism, which means it just allows the newly created process to read memory of the original process, but it prevents overwriting memory visible to another process, by disabling write access and setting the 10th bit (which indicates this page is secured for copy-on-write). This 10th bit is one of the bits available to the operating system to use for maintanance, see Fig. 30.

If this bit has been set for the page table corresponding to the faulting address, a copy of the 4 KB of physical memory needs to be made, to avoid overwriting memory that is used by another process.

If the page table needs to be copied itself, it has the 10th bit set in the page directory. In that case, the complete page table contents is copied into a new page table location, and the original entry in the page directory is replaced with this new page table. Then a copy of the page itself is made, and the original entry in the page table is replaced with the new page.

If the 10th bit was not set on either the page directory or the page table, we have another, very simplistic way to (try to) handle page faults. We get the CR2 value and just try to map the virtual address to the physical address, available to both user and kernel, and with write access set. The error code is not used, though it's bits are printed to the screen as debugging help.

The function which handles all the virtual-to-physical mappings, mapPages, won't allow any caller to map the NULL page, nor will it allow a process to make kernel memory available, but there's no security beyond that.

It should be obvious that the page fault handler needs more tweaking in order to ensure security; for example, it wouldn't be too hard to mess with memory from another process in this setup (by programmer error or as an "attack"). For our needs, it's sufficient as it is; in our kernel, no or very few, page faults should occur.

10.6 Page Use in Our Kernel

10.6.1 Mapping Addresses

In our kernel we have one function to do all mappings from virtual to physical addresses, named mapPages. It takes the virtual and physical address, the amount of memory to map, and whether it's user mode or kernel mode, which is mapped.

It marks all required page directories and page tables as present, also allowing write access. Finaly it sets `CR3` to clear paging caches (which is necessary to ensure the changes get stored).

10.6.2 Allocating an Amount of Process Memory

As mentioned before, the need for paging support became apparent as an requirement of the ELF format. As a result, the ELF loader manipulates the page table of the process an ELF file will be executing in. This happens during an `execve` functionality, which is part of the general system call, executed by `int 0x80`.

The `user_page_alloc` function allocates an amount of memory, mapped starting at a given virtual address. The physical address really depends on the address of the process itself. (It is appended to the end of the process memory) When it is finished it returns the starting physical memory address to the caller.

10.6.3 Copying Process Strings

Each process gets its own page table, which is required by the ELF loading functionality. Because of this, whenever a string has to be read from process memory, the process page directory is loaded into `CR3`, the string is copied into kernel memory, and the original page directory is loaded again (this is done by the function `u2kstrcpy`). Likewise, if there would be a need to copy kernel strings to process memory, the same trick would be necessary.

A small but important detail is that, together with setting the `CR3` register, also the `cr3` field of the current TSS is set. This is necessary as otherwise TSS switching, which happens because of timer interrupts, would mess up the functionality. We actually saw this happening, causing hangs as “random” memory got copied. (Another solution would be to disable interrupts temporarily, but that would be a more expensive solution).

11 Processes

11.1 Processes Overview

We have basic process support. Our kernel has multi-tasking support and we chose to make it similar to the way Linux processes work; The stages of a

stage	name	EAX value
process creation	fork	2
process termination	exit	1
program execution	execve	11

Table 4: process states

process are shown in Table 4.

The EAX value has to do with the way a system call in unix-like systems is executed. First a number of registers is set, of which EAX indicates which system call has to be executed. Then a software interrupt 128 (0x80) is executed. In our kernel this is implemented using a task switch.

As our kernel has multi-tasking support, there are also scheduling events. This means a process may be stopped to start or restart another process. This scheduling is independent of the running program. That is, the program doesn't have to be written such that it would give hints to the scheduler.

In the following sections we first describe the hardware support as-is. After that we describe how we used the different parts.

11.2 Hardware Support

We tried to use hardware support for processes as appropiate. In the Intel architecture, there is a special 'task-state segment' (TSS) which can be used to hold register values used while a process exists in memory; There are a number of different structures that pertain to the TSS.

11.2.1 TSS Descriptor

A TSS descriptor is an entry in the GDT. It's purpose is to reference a TSS, which is required by some Intel instructions (see below). The format is shown in Fig. 32. It has the following fields:

Limit The limit indicates how big the TSS is. It must be at least 103. This size requirement is because of the way the TSS structure looks like. Using a smaller amount generates an invalid-TSS exception (exception 10).

Like other GDT entries, it is split up in parts.

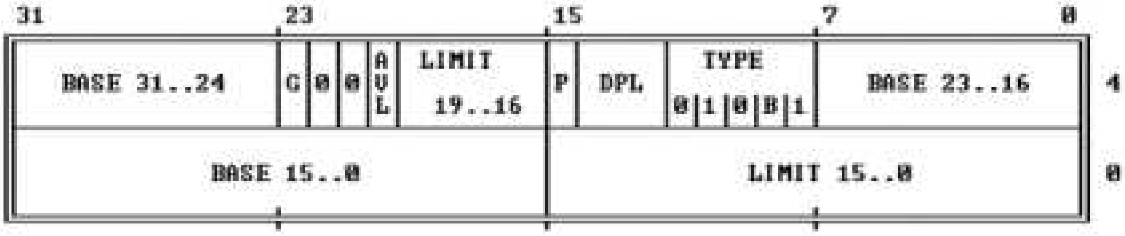


Figure 32: TSS descriptor

Base The base indicates the start address of the TSS; like other GDT entries, it is split up in parts.

Type The type field has to be binary 0-1001. It is set to 0-1011 by the CPU when the TSS becomes active, this is a busy flag. A TSS cannot be (re-)activated if it is marked as busy.

DPL The DPL field works as one would expect. Only a program whose CPL is numerically equal to or less than the DPL of the TSS descriptor can activate the task.

P The P bit (present bit) should always be set to 1, otherwise a segment-not-present exception is generated (exception 11).

AVL As usual the AVL bit may be used freely by software

G The granularity bit (as found in all GDT entries). We always set this bit to 0 for a TSS descriptor

11.2.2 TR

The Task Register (TR) holds the TSS descriptor segment indicating the TSS that is currently used. This register has to be setup correctly in order to be using the TSS (using the LTR instruction), however there is no reason of setting the TR in case the OS is not using any TSS.

11.2.3 Task gate descriptor

A task gate descriptor is an entry in the GDT. It's purpose is to reference a TSS descriptor, which is used to enable programs to reference the TSS that

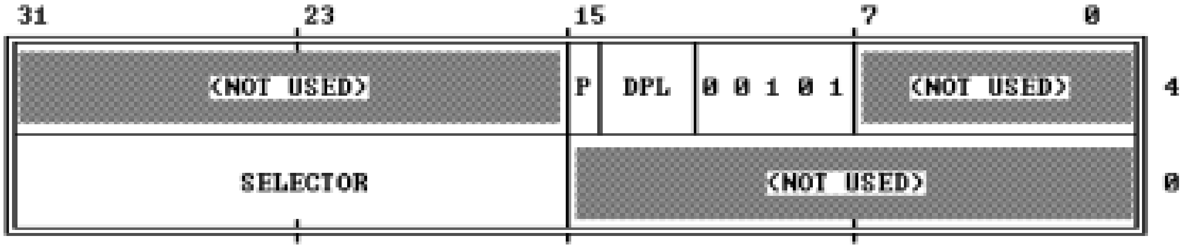


Figure 33: Task gate descriptor

otherwise may not reference the TSS directly. We use it one for the handler of interrupt 128 (0x80) and for the timer handler, interrupt 32 (0x20). The format is shown in Fig. 33. It has the following fields:

Selector The TSS selector that is referenced

Type The type field has to have the individual bits set to 00101.

DPL The DPL field overrides that one of the TSS segment. Only a program whose CPL is numerically equal to or less than the DPL of the task gate descriptor can activate the task.

P The P bit (present bit) should always be set to 1, otherwise a segment-not-present exception is generated (exception 11).

11.2.4 TSS

A task-state segment, a TSS, is a structure designed by Intel to have hardware support for tasks. It is a set of registers that define the state of the CPU. Fig. 34 shows a TSS. The Intel manual defines a task as a unit of work, that a processor can dispatch, execute, and suspend. It is easy to see that processes can be built on Intel tasks.

A TSS contains a lot of information. The following fields are changed during a task switch:

- General purpose registers:
    - EAX
    - ECX

  - EDX
  - EBX
  - ESP
  - EBP
  - ESI
  - EDI

- Segment selector fields:
  - ES
  - CS
  - SS
  - DS
  - FS
  - GS

- EFLAGS The EFLAGS register, which has bits for various CPU-state properties

- EIP The instruction pointer (or "program counter") which indicates the next instruction to execute

- Previous task link The TSS segment selector of the previous task. This field is set on activation of the TSS, and it is used when executing an IRET instruction

The following fields will never be changed by the CPU. They should be setup correctly when the task is created:

- LDT The segment selector for an LDT associated with the task (if any) We always set this field to 0.

- CR3 The physical address of the start of the page directory used by the task.

- Stack pointers for privilege levels 0, 1 and 2:
  - stack segments

    - ∗ SS0
    - ∗ SS1
    - ∗ SS2
  - stack offsets
    - ∗ ESP0
    - ∗ ESP1
    - ∗ ESP2

- T flag If the T flag is set, it causes a debug exception (exception 1) to be raised when a task switch to this task occurs.

- I/O map base address When using an I/O permission bit map, this is an offset to the map. We always set this field to 0.

In our kernel, task switches occur as one of the following ways:

- An interrupt occurs, with a task-gate in the IDT for the interrupt. This sets the back link of the TSS structure.

- The current task executes an IRET and the NT flag of the EFLAGS is set. The TSS referenced by the back link of the TSS structure is (re)activated, and it must be marked as busy.

The back link is used when an IRET is executed, and the NT flag is set. Then the TSS linked to is reactivated.

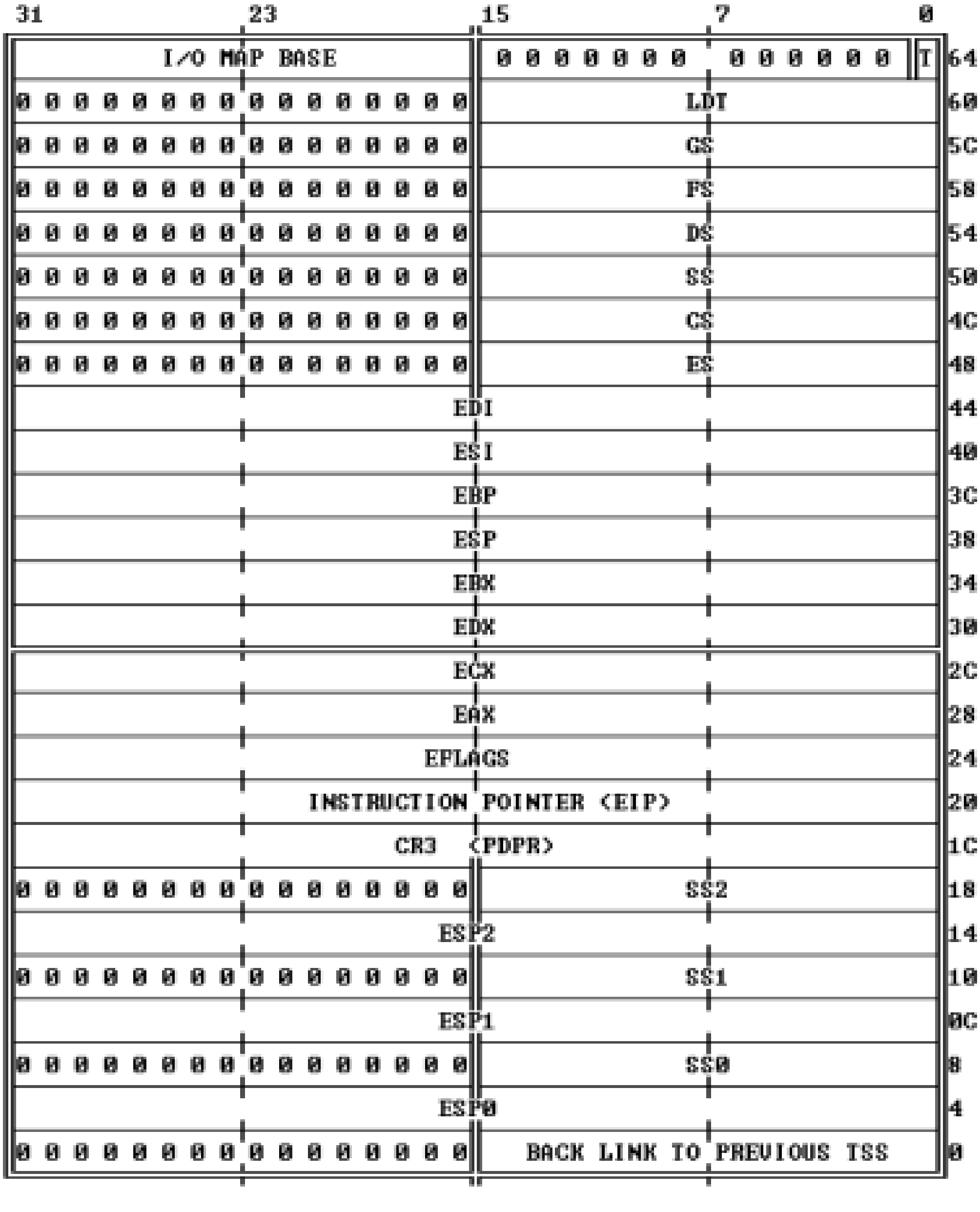
11.3 Structures in use

11.3.1 TSS structures in Use

We have two kernel-specific TSS structures, and one for each task.

1. sys_tss The TSS which the kernel starts with - after initialisation this becomes the idle task. Here `intstack` is the address just after the stack used for the 0x80 interrupt. The fields are setup as indicated in Table 5.

2. int_tss The TSS which is used especially for software interrupt 128 (0x80). This interrupt is used as system call. Here `intstack` is the address just after the stack used for the 0x80 interrupt. This should be read like Table 6.



31 23	15 7 0	
I/O MAP BASE	0 0 0 0 0 0 0 0 0 0 0 0 0 T	64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	LDT	60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34
EDX		30
ECX		2C
EAX		28
EFLAGS		24
INSTRUCTION POINTER (EIP)		20
CR3 (PDPR)		1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS2	18
ESP2		14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS1	10
ESP1		0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS0	8
ESP0		4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	BACK LINK TO PREVIOUS TSS	0

Figure 34: TSS

field	value
back link	0
ESP0	`intstack - 4`
SS0	`SYS_DATA_SEL`
ESP1	0
SS1	0
ESP2	0
SS2	0
CR3	`end + 0x1000`
EIP	0
EFLAGS	0
EAX	0
ECX	0
EDX	0
EBX	0
ESP	0
EBP	0
ESI	0
EDI	0
ES	`SYS_DATA_SEL`
CS	`SYS_CODE_SEL`
SS	`SYS_DATA_SEL`
DS	`SYS_DATA_SEL`
FS	`SYS_DATA_SEL`
GS	`SYS_DATA_SEL`
LDT	0
T-bit	0
I/O map offset	0

Table 5: sys_tss data

field	value
back link	0
ESP0	0
SS0	`SYS_DATA_SEL`
ESP1	0
SS1	0
ESP2	0
SS2	0
CR3	`end + 0x1000`
EIP	isr128
EFLAGS	0x3202
EAX	0
ECX	0
EDX	0
EBX	0
ESP	`intstack - 4`
EBP	0
ESI	0
EDI	0
ES	`SYS_DATA_SEL`
CS	`SYS_CODE_SEL`
SS	`SYS_DATA_SEL`
DS	`SYS_DATA_SEL`
FS	`SYS_DATA_SEL`
GS	`SYS_DATA_SEL`
LDT	0
T-bit	0
I/O map offset	0

Table 6: int_tss data

```
dw 104                        ; limit 15.. 0
dw 0                    ; base  15.. 0, to be filled
db 0                    ; base  23..16, to be filled
db 10001001b            ; present, ring 0, not busy TSS
db 0                    ; limit 19..16, not granular
db 0                    ; base  31..24, to be filled
```

Figure 35: A TSS SEL definition

3. timer_tss The TSS which is used especially for software interrupt 128 (0x80). This interrupt is used as system call. Here `intstack` is the address just after the stack used for the 0x80 interrupt. This should be read like Table 7.

11.3.2 TSS descriptors in use

There are five TSS descriptors in use:

1. `SYS_TSS_SEL` is the TSS descriptor of the kernel task. The definition is in Fig. 35. The base address is filled immediately after control was transfered to the kernel. After initialisation, this is the idle task.

2. `USER_TSS_SEL1` is the first TSS descriptor for a user-level process. It is selected and filled in during scheduling.

3. `USER_TSS_SEL2` is the second TSS descriptor for a user-level process. It is selected and filled in during scheduling. Because of the fact that it is possible to fill in the descriptor during scheduling, there is no need to have more than two TSS descriptors for user-level processes. The way this works is described later on.

4. `INT_TSS_SEL` is the TSS descriptor of the interrupt handler (for interrupt 0x80, the system call interrupt). The base address is filled immediately after control is transfered to the kernel.

5. `TIMER_TSS_SEL` is the TSS descriptor of the timer handler (for interrupt 0x20, the timer interrupt). The base address is filled immediately after control is transfered to the kernel.

field	value
back link	0
ESP0	0
SS0	`SYS_DATA_SEL`
ESP1	0
SS1	0
ESP2	0
SS2	0
CR3	`end + 0x1000`
EIP	isr32
EFLAGS	0x3202
EAX	0
ECX	0
EDX	0
EBX	0
ESP	`intstack - 4`
EBP	0
ESI	0
EDI	0
ES	`SYS_DATA_SEL`
CS	`SYS_CODE_SEL`
SS	`SYS_DATA_SEL`
DS	`SYS_DATA_SEL`
FS	`SYS_DATA_SEL`
GS	`SYS_DATA_SEL`
LDT	0
T-bit	0
I/O map offset	0

Table 7: timer_tss data

```
typedef struct TSSsel {
  word lowLimit, lowBase;
  byte middleBase, type, highLimit, highBase;
} TSSsel;
```

Figure 36: TSSsel type

11.3.3 TR Initialisation

After setting up the GDT and interrupt handlers, the TR register is set to `SYS_TSS_SEL` using the `ltr` (load TR) instruction. After that, the TR register is never changed directly, but it is changed through task switch operations.

11.4 Processes and Tasks

11.4.1 TSSsel structure

Sometimes it is necessary to write into a TSS descriptor. For this a TSSsel type is defined, as showed in Fig. 36. It matches the memory layout of a TSS descriptor exactly.

11.4.2 Tasksel Structure

For a process a bit more information is needed than a TSS stores, so a custom-defined structure has been defined, as shown in Fig. 37. It contains the following fields:

sel A pointer to a TSSsel, to associate a TSS descriptor with a process.

tss A pointer to a TSS structure, indicating this TSS is part of the process.

stack A pointer to an array, which is the stack of this process.

addressBase A pointer indicating the start address of memory taken by this process.

endAddress A pointer indicating the end address of memory taken by this process.

memoryLimit The total amount of memory (RAM) this process may take.

state One of the possible `process_state` values

```
typedef enum {running = 0, died = 1, waiting_io = 2,
    waiting_sys = 3, waiting_next} process_state;
typedef dword processStack[STACKSIZE]; typedef struct process {
  TSSsel *sel;
  TSS *tss;
  processStack *stack;

  char *addressBase;
  char *endAddress;
  unsigned long int memoryLimit;
  process_state state;

  int period;
  int startTick;
  int periodCount;
  int *missCounter;
} process;
```

Figure 37: process type

period An integer indicating the period of a cycle of the real-time run

startTick The tick at which the current period started

periodCount The number of the current period (a counter)

missCounter A pointer to an integer that should hold the number of misses of this process. This is a pointer as the value should be available even after the process is removed from the process list.

11.4.3 Process creation

A new process is created when a fork() is executed. That is, interrupt 0x80 is executed with EAX set to 2. Our fork implementation wants to have a value in EBX which is the period of a cycle in the cricical part of the process to be used. The creation of the new process happens in setupTask(period). It does some initialisation, it copies the stack of the current task, and copies the TSS of the current task. It sets the ESP and EBP pointers to point into the newly created stacks, at the corresponding position of the old process; At last it initiates the cr3 field to funtion as a real paging directory and tells the scheduler a new process is to get scheduled.

11.4.4 Process termination

When a program exits, it executes interrupt 0x80 with register EAX set to 1. The currentTask state is set to 1 (exited) and the scheduler is called to start the next process.

11.4.5 Program execution

When a process needs to start a program, it needs to execute an 'execve' call, that is, it executes interrupt 0x80 with register EAX set to 11. If successfull, the program will get started, and the process will stop existing when the program exits. When failing, the system call will return.

Our execve implementation only supports executable ELF-files, which are statically linked. This means most ELF-files built using gcc cannot be executed, as they depend on a standard shared library to start.

12 Conclusion

We have now implemented most of the kernel. Our kernel now has video support, interrupt support, keyboard input support and has multi-tasking support. One thing missing is disc I/O support.

Kernel development is difficult as many different parts depend on each other. Text-based video output doesn't require much, but most other components are tricky.

1. We found switching from kernel-mode to user-mode to be a lot harder than one would think. The problems occured with changing the mode the code is running in.

2. Any hardware drivers, even simple ones like the hardware timer and the keyboard, need an exception handling mechanism to be in place.

3. Program execution needs a program loader. In case of the ELF format, this in turn needs paging support.

Information on kernel development can be found on the internet, but most of the pages assume component X already is in place.

Also debugging problems can be quite hard.

13 Reference

For gaining understanding, and writing the kernel code, the following pages were very helpfull:

- http://osdever.net/bkerndev/index.php?the_id=90
- http://www.mega-tokyo.com/osfaq2/
- http://www.intel.com/design/pentium4/manuals/index_new.htm
- http://www.openbg.net/sto/os/xml/grub.html
- http://www.osdever.net/FreeVGA/vga/crtcreg.htm
- http://stakface.com/nuggets/index.php?id=11&replyTo=0
- http://www.osdever.net/tutorials/brunmar/tutorial_02.php
- http://www.logix.cz/michal/doc/i386/chp09-04.htm
- http://www.delorie.com/djgpp/doc/ug/interrupts/inthandlers1.html
- http://www.osdever.net/tutorials/interrupts.3.php?the_id=41
- http://www.fordax.com/os/publish_files/image085.gif
- http://www.internals.com/articles/protmode/interrupts.htm
- http://www.intel.com/design/itanium/manuals/iiasdmanual.htm
- http://en.wikipedia.org/wiki/Interrupt_request
- http://www.osdev.org/osfaq2/index.php/Can%20I%20remap%20the%20PIC%3F
- http://www.osdev.org/osfaq2/index.php/PIT
- http://pdos.csail.mit.edu/6.828/2005/readings/i386/toc.htm

- http://www.logix.cz/michal/doc/i386/
- IA-32 Intel® Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1