Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tine Šubic

Implementacija minimalnega operacijskega sistema

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2017

University of Ljubljana
Faculty of Computer and Information Science

Tine Šubic

Minimal operating system implementation

DIPLOMA THESIS

UNIVERSITY STUDY PROGRAMME
UNDERGRADUATE DEGREE
COMPUTER AND INFORMATION SCIENCE

Supervisor: doc. dr. Tomaž Dobravec

Ljubljana, 2017

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tine Šubic

Implementacija minimalnega operacijskega sistema

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2017

Faculty of Computer and Information is publishing the following diploma thesis:

Thesis subject:

In your diploma thesis survey the field of operating systems and implement a minimal version of operating system. Your implementation should include basic kernel functions for handling hardware interrupts, dynamic memory management, filesystem support, drivers for peripheral devices (e.g. keyboard) and a user interface with a basic command set.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomskem delu se osredotočite na področje operacijskih sistemov in predstavite različne tipe, ki se pojavljajo v praksi. V okviru dela implementirajte minimalni operacijski sistem. Vaša implementacija naj vsebuje osnovne jedrne storitve za upravljanje strojnih prekinitev, dinamično upravljanje s pomnilnikom, podporo datotečnemu sistemu, gonilnike za tipkovnico in osnoven uporabniški vmesnik z nekaj ukazi.

I would like to thank my mentor, doc. dr. Tomaž Dobravec for his quick responses and guidance during writing of this thesis. I am especially thankful to my family and friends support during my studies, and Luka Bradeško at Jozef Stefan Institute, for his indulgence of my absences at work.

Contents

List of acronyms

acronym	English	Slovene
ASCII	American Standard Code for Information Exchange	ameriški standardni nabor za zapis znakov
GCC	GNU Compiler Collection	prevajalniška zbirka GNU
NASM	Netwide Assembler	program za prevajanje zbirnika x86
OS	Operating System	operacijski sistem
BIOS	Basic Input/Output System	programska koda ki se izvede ob zagonu sistema
VGA	Video Graphics Array	standard za grafični prikaz podatkov na zaslonu
GNU	GNU's Not Unix	GNU ni Unix
ELF	Executable and Linkable Format	Datoteka pripravljena za izvajanje ali povezovanje
COM	Communication Port	komunikacijski vhod
IDT	Interrupt Descriptor Table	tabela prekinitvenih deskriptorjev
GDT	Global Descriptor Table	tabela globalnih deskriptorjev
ISR	Interrupt Service Register	prekinitveno-servisni register
IRQ	Interrupt Request	prekinitvnea zahteva
CPU	Central Processing Unit	centralna procesna enota
REPL	Read-Evaluate-Print Loop	interaktivni besedilni vmesnik
ACPI	Advanced Configuration and Power Interface	napredni vmesnik za konfiguracijo in napajanje

Abstract

Title: Minimal operating system implementation

Author: Tine Šubic

The following thesis describes an implementation of a minimal operating system which supports user interaction through keyboard and a terminal window. The system design is inspired by GNU/Linux operating system and is at its core monolithic kernel with support for memory management, kernel modules and various peripheral drivers, like serial ports, VGA terminal, timers and interrupts. The second part of the system is the user mode using a REPL terminal with some inbuilt tools for basic computer tasks and demonstration on system capabilities. The operating system uses C programming language, as well as x86 architecture assembly for some critical parts. Bochs, the virtual machine software was used to emulate hardware while in development, however, the system can be booted on compatible physical hardware.

Keywords: assembly, C, interrupts, operating system, kernel, Linux.

Povzetek

Naslov: Implementacija minimalnega operacijskega sistema

Avtor: Tine Šubic

V diplomskem delu je predstavljena implementacija minimalnega operacijskega sistema, ki podpira interakcijo z uporabnikom skozi terminalsko okno. Sam sistem se zgleduje po operacijskem sistemu GNU/Linux in je v osrčju sestavljen iz monolitnega jedra in podpira dinamično upravljanje s pomnilnikom, podporo jedrnim modulom in gonilnike za tipkovnico, VGA terminal, prekinitve in serijske vhode. Drugi del sistema predstavlja uporabniški del v obliki REPL terminala z nekaj vgrajenimi osnovnimi orodji za uporabo sistema in prikaz osnovnih funkcionalnosti. Projekt je napisan v programskih jezikih C in zbirniku za arhitekturo x86. Za simulacijo strojne opreme med razvojem je bil uporabljen virtualni stroj Bochs, sistem pa je možno naložiti tudi na zgoščenko in ga zagnati na kompatibilni fizični strojni opremi.

Ključne besede: zbirnik, C, prekinitve, operacijski sistem, jedro, Linux.

Razširjeni povzetek

Naslov: Implementacija minimalnega operacijskega sistema

Avtor: Tine Šubic

Trg modernih računalniških operacijskih sistemov že vrsto let zasedajo veliki igralci, ki razvijajo in prodajajo operacijske sisteme, namenjene širši javnosti. Tako na trgu operacijskih sistemov osebnih računalnikov prevladuje Microsoft 90-odstotnim deležem [4], medtem ko si operacijski sistemi podjetja Apple, GNU/Linux in različni nišni sistemi delijo preostalih 10 odstotkov. Na strani javno dostopnih strežniških sistemov (poštni in DNS strežniki, strežniki za strežbo spletnih strani) je stanje ravno nasprotno, saj tu prevladujejo različne distribucije GNU/Linux s 66% deležem [9], preostalih 34% pa skorajda v celoti zajemajo različne izdaje operacijskega sistem Microsoft Windows Server. Mac OS X se v tem sektorju pojavlja z manj kot 0.1% deležem. V podatkih sicer niso vključeni zasebni strežniki in interni strežniki podjetij, saj zanje v večini primerov ni informacij. Iz podatkov je razvidno, da imajo najbolj razširjeni sistemi za sabo podporo močnih podjetij, ali pa zelo zavzeto razvijalsko skupnost, saj je sicer nemogoče razumeti in urejati razvoj tako velikega števila komponent z več milijoni vrstic izvorne kode. Tudi v GNU/Linux skupnosti, ki slovi po velikem številu distribucij operacijskega sistema, gre pri raznolikosti pogosto le za manjše nadgradnje ali zgolj razlike v uporabniških aplikacijah oziroma uporabniškem vmesniku. Nišni in specializirani sistemi se v veliki meri nahajajo le na napravah zanesenjakov in namensko izdelanih platformah.

Kljub vsemu pa je razvoj manjših sistemov v današnjih časih še vedno popolnoma dosegljiv cilj, saj je na voljo mnogo virov in odlične dokumentacije - že Intel za svojo arhitekturo IA-32 ponuja več kot 3000 strani podrobnih navodil o strojnih ukazih, procesorskih mehanizmih in funkcionalnosti platforme. Manjši, specializirani sistemi lahko v nekaj tisoč vrsticah kode podpirajo osnovne funkcionalnosti kot so periferni gonilniki, sistemske prekinive, dinamično upravljanje s pomnilnikom in uporabniški vmesniki. Seveda je pri razvoju sistema potrebno najprej ugotoviti za kakšno platformo želimo napisati operacijski sistem. Odločitev je lahko odvisna od različnih dejavnikov, na primer kompleksnosti platforme, željenih lastnosti sistema in obsegu dokumentacije ki je na voljo za to platformo. Če so naši končni uporabniki ljudje, ki so navajenih modernih operacijskih sistemov, je ena od ključnih funkcionalnosti zagotovo grafični vmesnik, kot tudi možnost zagona večih programov hkrati in mrežna povezljivost.

Pred letom 2000 letih je na področju osebnih računalnikov prevladovala arhitektura IA-32/x86 (32 bitna zasnova podjetja Intel), po prelomu tisočletja pa sta se pojavili 64 bitni arhitekturi IA-64 (Intel) in AMD64 (AMD). Kljub temu da gre pri obeh za moderno 64-bitno arhitekturo sta se podjetji lotili zasnove na različne načine. Intel je IA-64 (Itanium Architecture) proizvedel kot popolnoma novo platformo, medtem ko je AMD razvil zgolj nadgradnjo obstoječega 32-bitnega modela z dodatnimi funkcionalnostmi in strojnimi ukazi. AMD svojo arhitekturo še vedno uporablja v svoji zadnji generaciji procesorjev (Ryzen) z nekaj nadgradnjami, medtem ko je Intelovo Itanium arhitekturo na trgu osebnih računalnikov zamenjala arhitektura Intel64. AMD, Intel in VIA so v letu 2000 izdali dokument, ki specificira lastnosti generične arhitekture 64-bitne platforme pod imenom x86_64. Zaradi te odločitve je uporaba modernih generacij procesorjev podjetij AMD in Intel skoraj identična z vidika platforme, čeprav imata nekaj razlik pri naborih strojnih ukazov in izvajanju le teh. Predstavljeni operacijski sistem v tem diplomskem delu uporablja arhitekturo IA-32 zaradi enostavnosti in lažjega uporabljanja s pomnilnikom.

V sledečem diplomskem delu je predstavljena implementacija operacijskega sistema, ki se zgleduje po operacijskem sistemu GNU/Linux in je v osrčju sestavljena iz monolitnega jedra, ki v tesno povezanem kosu kode združuje jedrne storitve in periferne gonilnike. Gre za eno-uporabniški sistem, ki podpira eno aplikacijo naenkrat - tok kode je, z izjemo prekinitev, linearen. V samem jedru je podprto dinamično upravljanje s pomnilnikom preko funkcij `free` in `malloc` ki uporavljata s kopico, podporo jedrnim modulom in upravljanje s strojnimi prekinitvami. Implementirani so tudi gonilniki za nekatere periferne naprave. Sistemski čas je podpt s prekinitvenim časovnikom, uporabnik lahko uporablja tipkovniko z razporeditvijo tipk UK, izpis je viden na VGA terminalu velikosti 25x80 znakov, možno pa je tudi dostopati do datotek na statičnem podatkovnem sistemu.

Drugi del sistema predstavlja uporabniški del v obliki interaktivne besedilne konzole z nekaj vgrajenimi osnovnimi orodji za uporabo sistema in prikaz osnovnih funkcionalnosti. Tako lahko na zaslon izpisujemo nize znakov, pregledamo seznam datotek v datotečnem sistemu, preberemo njihovo vsebino, ali preverimo trenutni čas sistema. Sama konzola podpira tudi nalaganje izvršljivih skript iz podatkovnega sistema in izvajanje le-teh - skripte podobno kot v okolju Bash podpirajo standardne ukaze iz konzole. Projekt je napisan v programskih jezikih C in zbirniku za arhitekturo x86, saj ta jezika zaradi nizkonivojskosti omogočata natančno upravljanje z viri in naslavljanje pomnilnika. Izvorno kodo je mogoče prevesti z orodji ld, GCC in NASM, ki so na voljo v večini GNU/Linux distribucij. Za simulacijo strojne opreme je bil uporabljen virtualni stroj Bochs, ki je sposoben emulirati različno strojno opremo, procesorske frekvence in periferne naprave. Sistem je možno naložiti tudi na zgoščenko in ga zagnati na kompatibilni fizični strojni opremi.

V diplomskem delu je na začetku predstavljenih nekaj osnovnih konceptov operacijskih sistemov in orodja, uporabljena za prevod izvorne kode v izvršljivo obliko. Sledeča poglavja, opisujejo: proces inicializacije sistemskega jedra, kjer vklopimo podporo prekinitvam in dinamičnemu pomnilniku ter naložimo gonilnike, sledijo periferni gonilniki za tipkovnico, serijske vhode in podatkovni sistem, nato pa na kratko opišemo še preskok sistema iz jedrnega načina v neprivilegirani način. Predzadnje poglavje predstavi odločitve pri implementaciji besedilnega uporabniškega vmesnika in predstavi nekaj implementiranih ukazov. Prikazani so tudi njihovi rezultati in zaslonske slike vmesnika. Zaključno poglavje zaokroži opis in predstavi nekaj idej za nadaljnji razvoj - sistemu lahko v bodoče dodamo podporo drugim podatkovnim sistemom, možnost vzporednega zagona programov, ali pa morda podporo mrežni opremi za komunikacijo s spletom.

Chapter 1

Introduction

The current operating systems market is dominated by the big players: On desktop, 90% [4] of the market is taken up by various Microsoft Windows distributions, with remaining 10 percent divided between Mac OS X (4.0%), GNU/Linux (2.4%) and various other specialised OSes (3.6%). The disparity is reversed if one looks at public-facing Internet servers (website, mail and DNS server), where Unix derivatives are used in approximately 66% of servers and Microsoft Windows distributions claim remaining 34%. However, the unknown/other category in server section remains well below 0.1% of market share. These numbers do not include data about privately owned and internal company servers. A look at the numbers above tells us, that homegrown and specialised OSes are not exactly widespread. The most commons operating systems either have a large corporation or a dedicated community backing them and continuing the development.

However, development of an operating system that provides some basic functionality like peripheral drivers, memory management, user interface and interrupt handling need not be done by a large group of specialised experts. Such a system, utilising a monolithic kernel and tightly coupled code can be implemented in just a couple thousand lines of code. The operating system showcased herein was inspired by the GNU/Linux philosophy [1] on a smaller scale, mainly with the use of Virtual File System, shell-like interface and other components.

1.1 Structure and aims of the thesis

The following thesis is divided into 6 separate chapters. First chapters deals with motivations for this thesis and introduction to some general operating systems concepts, as well as reasoning behind specific types of the implemented operating system. The second chapter describes tools used for compiling and preparing operating system and shows relevant scripts and commands for generating the bootable disk image. The next chapters contain details on implementation for system service initialisation, device drivers and user interface. The final chapters sums up the structure and capabilities of shown operating system.

1.2 Types of operating systems

Most commonly, modern operating systems like Windows, Max OS X and various Linux distributions are general-purpose multi- or single-user, multitasking operating systems, which allow one or more users to run more than one program concurrently. However, there do exist other specialised operating systems, built specifically with certain tasks and hardware in mind:

- **distributed** - used to manage computer clusters, commonly in a networked environment like data centres (example: Inferno[1] OS by Bell Labs and Vita Nuvoa Holdings),

- **templated** - used in conjunction with virtualization and cloud computing for virtual guest operating systems, allowing easy deployment of identical environment instances,

- **embedded** - developed for use with embedded computer systems with limited resources and specialized interfaces (example: Windows CE/Embedded[2]),

- **real-time** - used in time-critical environments with deterministic task scheduling. Specialises in minimising operation latency and buffer delays (example: FreeRTOS[3]),

[1] Available at: http://www.vitanuova.com/inferno/

[2] Available at: https://www.microsoft.com/windowsembedded/en-us/purchase.aspx

[3] Available at: http://www.freertos.org/

- **library** - commonly used with unikernels [5], using a conglomeration of libraries that provide common operating system services like memory management, networking, peripheral drivers, etc. (example: used by IncludeOS[4] unikernel).

  The OS implementation presented in this thesis uses a single-user, single-task approach, commonly seen in early version of operating systems before the rise of time-sharing and late true multitasking. The architecture allows the user to run a single program.

1.3 Types of operating systems' kernels

The operating system's core, named kernel, handles the most critical tasks from start-up to I/O requests, peripheral interfaces and memory management. As shown in diagram 1.1, traditional kernel in monolithic systems is the interconnecting layer between hardware devices and user-space applications, using system calls, interrupts and inter-process communication protocols to communicate between layers and interfaces. Some kernel types may differ form this diagram, especially those using external services that bypass kernel when talking to the hardware.

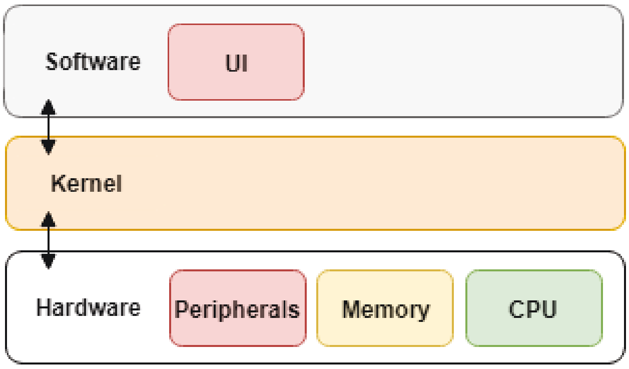


Figure 1.1: System architecture diagram.

There are multiple overarching kernel design approaches, described below:

- **monolithic kernels** - a kernel where all OS services are stored in one common memory space. Most work is done by system calls and

[4] Available at: http://www.includeos.org/

hardware interrupts due to easy hardware access. Due to dynamic loading of subsystems, they can be very small when installed, but can become hard to maintain when the codebase grows (example: Linux),

- **microkernels** - usually designed for a specific platform and implements only the core services multi-tasking, memory management, interrupt handling, inter-process communication, etc. in privileged mode. Other peripheral interfaces are implemented as user-space software. They are easier to maintain, but can be slower due to higher amount of abstraction,
- **hybrid kernels** - used by modern Microsoft Windows and Mac OS X operating systems. They take inspiration from microkernel design, but include some additional services (like networking) in kernel-space to provide a performance boost. They also use modules that provide additional kernel functionalities, but cannot be dynamically loaded as is the case with monolithic kernels,
- **others** - exokernels, nanokernels - rare and used for very specific hardware architectures or tasks.

The implemented operating system uses a monolithic kernel type due to relative ease of implementation, with all essential services running in kernel mode and optional privileged mode switch.

Chapter 2

System initialisation

2.1 Boot code

Every operating system needs a section of code that allows it to be actually called and executed. This implementation uses GRUB's Multiboot specification along with GRUB's second-stage bootloader file, allowing it to boot from a disk image into full environment. Listing 2.1 shows main code for this section. After setting up Multiboot-specific fields, such as header flags, checksum and section addresses, we first clear our flags register and then push 2 registers to the stack, containing the magic number `0x1BADB002` and multiboot structure address (used later for initialising memory management routines). Expected magic numbers, flags and header layout are described in Multiboot documentation [3].

With that done, we disable interrupts to avoid unwanted operations during our setup phase and finally call the `kmain(...)` function, which allows the C code to take over. `kmain` function will initialise all appropriate drivers, peripherals and kernel modules like memory management and interrupt handling. The final instructions merely start an infinite loop to allow the OS to run asynchronous code, such as interrupt handlers and the like.

```
MAGIC_NUMBER equ 0x1BADB002
; Before this: magic numbers, constants, fields requested by
    Multiboot spec
GLOBAL start            ; Kernel entry point.
EXTERN kmain            ; C main addres

start:
    push 0              ; clear EFLAGS
    push    eax         ; Load magic number
    push    ebx         ; Load multiboot info ptr

    cli                 ; Disable interrupts.
    xor     eax, eax
    call    kmain       ; start kernel execution
.loop:                  ; run infinite loop
    jmp     .loop
```

Listing 2.1: Main bootloader assembly code.

The kernel C code entry function called `kmain` starts the initialisation process with serial and VGA terminal devices. These two services allow boot procedure logging to the screen and to an external log file for further analysis. We initialise device drivers for keyboard and initrd filesystem, enable interrupt handling and memory management and finally set up system call facility. Thus equipped, we can make the jump to user-space, where user can interact with the system using inbuilt utilities that utilise system calls to generate expected feedback.

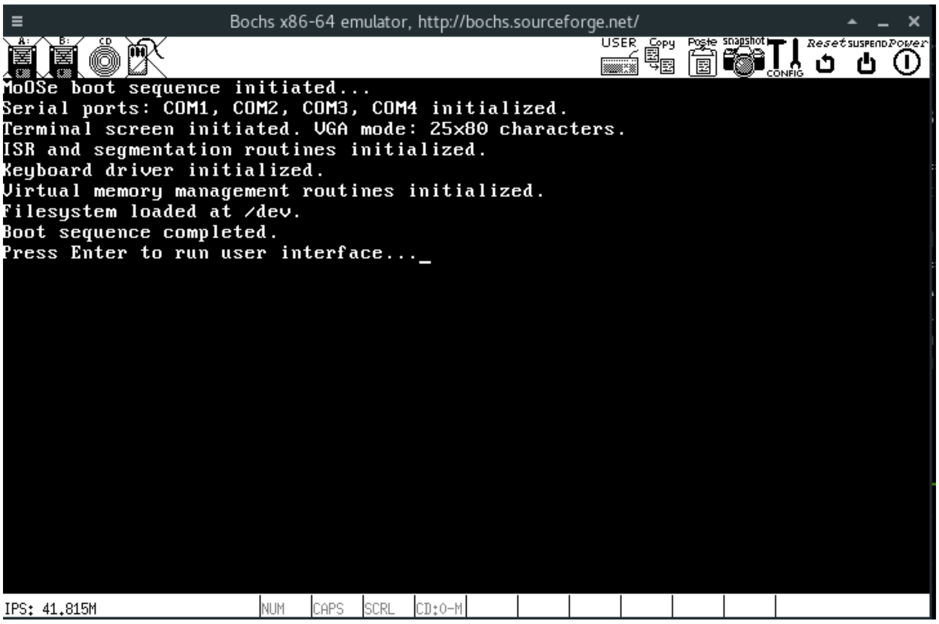
```
int main(struct multiboot *mboot, uint32 stack) {
    init_serial();
    init_terminal();
    print_boot_msg();
    fb_write("Serial ports: COM1, COM2, COM3, COM4
        initialized.\n", 100);
    fb_write("Terminal screen initiated. VGA mode: 25x80
        characters.\n", 100);
    log(KERNEL, INFO, "Serial ports: COM1, COM2, COM3, COM4
        initialized.");
    log(KERNEL, INFO, "Terminal screen initiated. VGA mode:
        25x80 characters.");

    initial_esp = stack;
    init_descriptor_tables();
    fb_write("ISR and segmentation routines initialized.\n",
        100);
    log(KERNEL, INFO, "ISR and segmentation routines
        initialized.");

    interrupt_enable();
    init_timer(100);
    init_kbd();
    fb_write("Keyboard driver initialized.\n", 50);
    log(KERNEL, INFO, "Keyboard driver initialized");

    placement_addr = *(uint32 *) (mboot->mods_addr + WORD_S);

    initialise_paging();
    fb_write("Virtual memory management routines initialized
        .\n", 100);
    log(KERNEL, INFO, "Virtual memory management routines
        initialized.");

    fb_write("Filesystem loaded at /dev.\n", 100);
    log(KERNEL, INFO, "Filesystem loaded at /dev.");
    fs_root = init_initrd(*((uint32 *) mboot->mods_addr));
    initialise_syscalls();
    fb_write("Boot sequence completed.\n", 50);
    fb_write("Press Enter to run user interface...", 100);
    wait_for_keypress();
    log(KERNEL, INFO, "User interface started.");
    ui_run();

    return RET_VAL;
}
```

Listing 2.2: C code entry point.

When the system finalises boot process, we are greeted with screen, shown in screenshot shown in Figure 2.1. Similar output is logged to `kernel.out` log file. The system is now waiting for a key press that will trigger the console user can then interface with.

2.2 Common library functions

Since our source code is compiled without inclusion of the standard library, we need to implement some of the common functions and definitions our-



```
MoOSe boot sequence initiated...
Serial ports: COM1, COM2, COM3, COM4 initialized.
Terminal screen initiated. VGA mode: 25x80 characters.
ISR and segmentation routines initialized.
Keyboard driver initialized.
Virtual memory management routines initialized.
Filesystem loaded at /dev.
Boot sequence completed.
Press Enter to run user interface..._
```

Figure 2.1: Boot process log displayed on screen.

selves, for easier programming and system extension. In the project, they are implemented in `stddef.h` and `stddef.c` source files.

```c
#define NULL ((void *)0)
#define TRUE 1
#define FALSE 0
#define RET_VAL 0xCAFEBABE

typedef unsigned long uint64;
typedef unsigned int uint32;
typedef unsigned short uint16;
typedef unsigned char uint8;

typedef long int64;
typedef int int32;
typedef short int16;
typedef char int8;
typedef int32 bool
```

Listing 2.3: Part of custom standard library header file.

Header file `stddef.h` (Listing 2.3) implements some useful type aliases.

For better visibility in code, it adds explicit short forms for unsigned and signed integers - `uint32`, `uint16`, `int32`..., as well as a type alias for `bool`. We also define a number of constants: TRUE and FALSE values, NULL value and RET_VAL which is the value that will be returned in `eax` register when kernel finishes execution.

`stddef.c` file contains implementations of some standard functions:

1. **strlen** - Returns the length of NULL-terminated string
2. **strcpy** - Copies a NULL-terminated string to a destination address, assuming valid free memory location.
3. **strcat** - Concatenates two strings in memory by appending to destination.
4. **memcpy** - Copies value in memory from source to destination (Used for duplicating structures and arrays)
5. **memset** - Fills memory location with repeated user-defined character (Commonly used for structure and array initialisation)
6. **panic** - Helper function for logging a critical error and triggering an infinite loop.

Since we treat all memory structures and strings as blocks of bytes, the functions are generally simple, iterating over a given memory block until they encounter a NULL byte or reach requested length, while performing a requested operation - swapping or replacing byte values. However, to use these functions efficiently, we need memory management facilities that expose function with which we can allocate memory to use. This is described in Memory management section.

```c
int32 strlen(const char *str) {
    int len = 0;
    while (str[len] != NULL) len++;
    return len;
}

void memcpy(uint8 *dest, uint8 *src, uint32 len) {
    uint8 *sp = (const uint8 *) src;
    uint8 *dp = (uint8 *) dest;
    for (int i = len; len != 0; len--) *dp++ = *sp++;
}

// Write len copies of val into dest.
void memset(uint8 *dest, uint8 val, uint32 len) {
    uint8 *temp = (uint8 *) dest;
    for (int i = len; i != NULL; i--) *temp++ = val;
}

//Copy string from source to destination
char *strcpy(char *dest, const char *src) {
    while (*src != NULL) {
        *dest++ = *src++;
    }
}

extern void panic() {
    int_disable(); // Disable interrupts.
    log(KERNEL, ERR, "PANIC!");
    for (;;); //Loop forever
}
```

Listing 2.4: Custom standard library.

2.3 Interrupt handling

Interrupt request handling is one of the core services of the kernel. Interrupts allow devices and software to asynchronously trigger certain actions without the need for endless spinning or periodic device checks (polling). As such, they are used for device drivers for keyboards, mouses, hard drives, software timers, handling system exceptions (like division by zero), special subroutine calls and others.

In the presented operating system, interrupts are used primarily for keyboard driver, programmable timers and exception handling. However, the implemented interface offers a `register_interrupt_handler(...)` function which allows new kernel modules to register their own interrupt handlers for specific actions. In further development, we could implement interrupt handler that might react on mouse commands, allow hot-plugging for new external devices and other actions.

2.3.1 Interrupt Descriptor Table

While capturing the interrupts signal is relatively easy, we still need to map these interrupt requests to correct handlers (function). To do this, we use an Interrupt Descriptor Table (IDT), that stores architecture-specific flags (Specified by Intel's IA-32 manual [2, p. 2823],) and handler addresses. Intel also provides `LIDT` command, that will load base address of our IDT to a special-purpose register to be used in calculating interrupt vector offsets in IDT, as shown in diagram from Intel's manual in Figure 2.2.

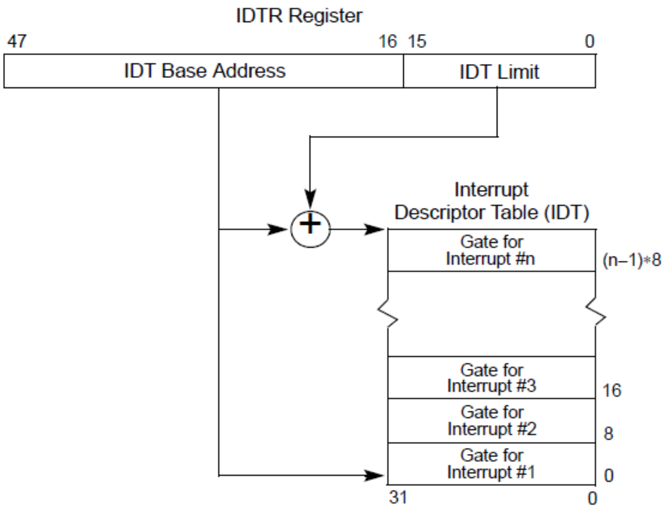


Figure 2.2: Structure of IDT register in IA-32 achitecture, diagram from Intel's IA-32 manual [2].

```
struct idt_descriptor {
   uint16 offset_low;   // low 16b of interrupt address
   uint16 selector;     // kernel segment sel
   uint8  zero;         // must be zero
   uint8  flags;        // x86 flags.
   uint16 offset_high;  // high 16 of interrupt addr
};

#define KERN_SEL 0x08
#define IDT_FLAGS 0x8E
static void initialize_idt() {
    ... //address setup

    //remap PIC numbers with port commands
    outb(PIC1, 0x11);
    outb(PIC2, 0x11);
    ...
    outb(0xA1, 0x0);

   install_gate( 0, (uint32)isr0 , KERN_SEL, IDT_FLAGS);
   ...
   install_gate(31, (uint32)isr32, KERN_SEL, IDT_FLAGS);

   lidt_trigger(&idt_ptr);
}

static void install_gate(uint8 num, uint32 addr, uint16 kss,
   uint8 flags) {
   idt_entries[num].offset_low = addr & OFFSET_MASK;
   idt_entries[num].offset_high = (addr >> 16) & OFFSET_MASK;

   idt_entries[num].selector = kss;
   idt_entries[num].zero = NULL;
   idt_entries[num].flags = flags;
}
```

Listing 2.5: Initialization of Interrupt Descriptor Table.

Listing 2.5 shows IDT entry structure and initialisation code. The IDT entry structure contains the address, as well as specific flags field that consists of magic number 14 and bits that describe privilege level. During initialisation, we prepare the IDT memory block and then add first 32 interrupt handlers. Since they are all working on kernel level, we add a kernel segment

selector parameter. install_gate function takes care of splitting the base address and setting the structure fields correctly.
We also remap interrupt controller numbers, since by default, some interrupts will conflict with CPU's exception and fault handlers. Thus we shift IRQs 0 through 15 to 32 through 47, since 31 is the last interrupt service register used by the CPU. In the end, we call assembly code to reload IDT address in IDTR register.

2.3.2 Interrupt handlers

To facilitate usage of interrupts, we need to implement certain functions. We need a catch-all initial interrupt request (IRQ) code which will handle the registers and and data segment descriptors and call the IRQ handler in C code. This is shown in Listing 2.6.

```
irq_common:
    pusha    ; Push general purpose register to stack
    ...      ; Data segment descriptor loading

    call irq_handler ; Call C code handler
    pop ebx   ; reload the original data segment descriptor
    ...      ;  restore register values

    popa    ; Restore registers from stack
    add esp, 8 ; Clean stack
    sti      ; Reenable interrupts
    iret
```

Listing 2.6: Common IRQ handler.

The code above is called by a specific IRQ# function, identified by a number (#) from 0 to 15. This function disables interrupts (since we are in a interrupt service subroutine already), pushes error code (0) and an interrupt service register (ISR) mapping (32) to the stack and finally jumps to execution of the irq_common code as shown above.

```
GLOBAL irq0
irq0:
  cli ; disable interrupts
  push byte 0 ;store error code
  push byte 32 ; store register mapping
  jmp irq_common ; execute common handler
```

Listing 2.7: IRQ handler for interrupt number 0, mapped to ISR 32.

The final part is the IRQ handler implemented in C code (shown in Listing 2.8). When `irq_common` calls `irq_handler` function, it firstly sends appropriate reset signals, acknowledging the interrupt and then executes the interrupt service routine handler registered for the particular interrupt number.

```
#define RESET 0x20
void irq_handler(registers regs) {
    if (regs.int_no >= 40) {
        outb(0xA0, RESET); //slave rst signal;
    }
    outb(0x20, RESET); //master rst signal

    if (interrupt_handlers[regs.int_no] != NULL) {
        isr_t handler = interrupt_handlers[regs.int_no];
        handler(&regs);
    }
}
```

Listing 2.8: C code for handling interrupt requests.

New handlers can be registered using a function shown in Listing 2.9, which maps the provided handler function to an element in the list of handlers, that will be later accessed by interrupt number.

```
void register_interrupt_handler(uint8 n, void *handler (
   registers_t)) {
  interrupt_handlers[n] = handler;
}
```

Listing 2.9: Function for registering custom IRQ handlers.

2.4 Memory management

IA-32 (Intel x86) architecture provides two common approaches to memory protection and virtual memory utilisation. The currently popular modern x86-64 architecture has in fact largely dropped segmentation support due to superiority of paging method, as well as because some x86-64 instructions require a flat memory model that cannot be provided by the segmentation. However, for our purposes, segmentation is still a perfectly viable (and in certain cases, unavoidable) approach, especially when setting privilege ring levels.

Memory allocation routines like `malloc` are generally provided by standard C library and supported by the host operating system. Since we have neither, implementing memory management routines provides us with a custom analogue of these methods. These routines will later on provide basis of tools available to developer and exposed via special functions (in kernel) or system calls (in user-space).

2.4.1 Segmentation

Memory segmentation approach dictates a division of computer's physical memory into segments. x86 architecture implements multiple physical registers, aimed at holding addresses of different memory segments (up to 6). Most commonly used are `cs` (code segment), `es` (extra segment), `ds` (data segment) and `ss` (stack segment) registers. Addresses within a specific segment are specified by a segment address and an offset. Upon accessing a memory location, the address is checked against segment bounds and accordingly, a hardware exception is triggered is the address is incorrect. One of the common results of this in C programs is the infamous `segmentation fault`.

To start using segmentation in operating system, we must initialise a global descriptor table (GDT). It holds addresses and access permissions for difference segments. The data for each segment is stored in a `gdt_data` structure, referenced in Listing 2.10. `base` field contains our 32-bit address of the segment. `limit`field allows us to define the size of addressable memory block. Access and granularity fields contain special bit fields, which allow setting privilege status, executable status, readability and so forth.

```
struct gdt_data {
    uint16 limit_low;   // low 16 address bits of the limit.
    uint16 base_low;    // low 16 address bits of the base.
    uint8  base_middle; // next 8 address bits of the base.
    uint8  access;      // privilege ring flags
    uint8  granularity;
    uint8  base_high;   // final 8 address bits of the base.
}
```

Listing 2.10: Global description table element structure.

With this, we can set up initial four descriptors (kernel code segment, kernel data segment, user code segment, user data segment). When switching to user mode later, we will need an additional stack segment. Additionally, to satisfy Bochs limitations, we include a null descriptor.

When call data fields are correctly set up, we use `gdt_flush` (shown in Listing 2.11) function to reload GDT data in CPU registers - replacing the base addresses and GDT limit. This is achieved with `LGDT`[1] instruction, used to flush previous selectors.

```
gdt_flush:
    xor eax, eax
    mov eax, [esp+4]   ; load param from stack
    lgdt [eax]         ; load new GDT address

    mov ax, 0x10       ; offset 16 in GDT for data segment
    mov ds, ax         ; Load all data segment selectors
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:.flush    ; far jump, 0x08 = data segment
.flush:
    ret
```

Listing 2.11: gdt_flush function for reloading GDT data.

This concludes segmentation initialisation and sets up the groundwork for virtual memory management that will allow us to dynamically allocate and free arbitrary memory blocks.

[1] Instruction documentation: http://x86.renejeschke.de/html/file_module_x86_id_156.html

2.4.2 Paging

Segmentation can be improved with the addition of paging. Instead of actual data, the segment information holds information about page table, where pages are mapped into physical memory in blocks called frames. These pages are generally 4KB in size and can be mapped/unmapped to physical memory location, thus providing a virtual memory mechanism.

Each page contains a metadata contains frame address, which tells the CPU which frame (block of physical memory) the page is mapped to. Because of this, we also need to maintain a list of which frames are currently in use and which are free to be allocated. To do this, we can define a set of functions that will allow us to set, unset and check which frames are free:

```c
#define PAGE 0x1000

void set(uint32 addr) {
   uint32 frame = addr/PAGE;
   frames[frame/32] |= (1 << (frame%32));
}

void unset(uint32 addr) {
   uint32 frame = addr/PAGE;
   frames[frame/32] &= ~(1 << frame%32);
}

bool is_free(uint32 addr) {
   uint32 frame = addr/PAGE;
   return (frames[frame/32] & (1 << (frame%32));
}
```

Listing 2.12: Memory allocation routine.

These routines are subsequently used by frame allocation and deallocation in heap routines that expand and contract free and used blocks of memory.

Memory allocation and deallocation procedure

The main requirement of memory allocation routine in this system is, that it returns the address of allocated memory block that is page-aligned (4kB page size). Because of this, we first check if the current free block is already page aligned, and if not, we allocate the beginning of memory block to the next page-aligned address:

```
#define PAGE_MASK 0xFFFFF000
#define PAGE_SIZE 0x1000
uint32 kmalloc(uint32 length, uint32 *phys_addr) {
    uint32 retaddr = NULL;
    if(heap_addr != NULL) {
        void *addr = alloc(size, DO_ALIGN, heap);
        page_t *page = get_page_at((uint32)addr, NULL,
           kernel_dir);
        *phys_addr = page->frame * PAGE_SIZE + ((uint32)addr
           & 0xFFF);
        retaddr = addr;
    } else {
        if (placement_addr & PAGE_MASK) {
            placement_addr &= PAGE_MASK;
            placement_addr += PAGE_SIZE;
        }
        retaddr = placement_addr + size;
    }
    return retaddr;

}
```

Listing 2.13: Memory allocation routine.

The `kmalloc` calls the `alloc` function in `kheap.c` file, which performs necessary checks and find the requested free space in page directory. Likewise, `free` function for memory deallocation frees the memory and attempts to merge the resulting fragments if any exist.

2.5 System calls

System calls allow the user's programs to request execution of certain actions from the kernel. More specifically, they allow us to call kernel-only function through a special interface from unprivileged Ring 3. In Linux, these calls[2] can perform many actions, from filesystem management to handling system signals or even rebooting the system.

[2] Linux Syscall Reference: http://syscalls.kernelgrok.com/

Setup

To use system calls, we must register the interrupt handler for interrupt vector 128. The syscall handler will access the table of function mappings, indexed by syscall number (passed via `eax` register to handler) which will be called with appropriate parameters.

The assembly function to trigger a syscall interrupt can now look like this:

```
GLOBAL syscall2

syscall2:
    mov eax, 2      ; syscall number
    mov ebx, param1 ; syscall parameter
    mov ecx, param2 ; syscall parameter
    int 80h         ; interrupt
```

Listing 2.14: Sample syscall code with 2 parameters.

This code will trigger interrupt #2, passing it two parameters (can be read from stack or registers). This will be handler by syscall handler described above, that will access the appropriate handler function and pass it the given parameters.

2.5.1 Implemented system calls

This section contains the list of some implemented system calls. While some calls utilise new code, many of them are merely user-mode interfaces for already implemented kernel-mode functions - like terminal and serial functions. All implemented system calls that reuse existing function can be called by prefixing said function with `syscall_`.

Shutdown

In a proper, feature-complete operating system, the developer generally uses ACPI mechanism to handle power state management. However, Bochs emulator includes an undocumented feature, that allows the operating system to write a string of characters to port `0x8900`, as shown in Listing 2.15, causing a system shutdown.

```
#define HALT_PORT 0x8900

void shutdown() {
    log(KERNEL, INFO, "System shutdown initiated.");
    char shutdown[9] = "Shutdown";
    for (int s = 0; i < 9; i++) {
        outb(HALT_PORT, shutdown[i]);
    }
}
```

Listing 2.15: Shutdown code.

Reboot

As with shutdown, reboot can be caused by multiple means - ACPI reset, system triple fault and most importantly a keyboard controller[3] triggering the CPU reset pin.

As shown in Listing 2.16, do this, we merely need to write a value of `0xFE` to CPU port `0x64` when the keyboard buffer is empty.

```
void reboot() {
    log(KERNEL, INFO, "System reboot initiated.");
    outb(0x64, 0xFE);
    halt_system(); //halt if reboot failed
}
```

Listing 2.16: Reboot code.

Others

We also need some previously used functions to work in user mode:

1. **syscall_read_scan_code()** - allows us to poll the keyboard for pressed keys in user mode when interrupt handler is disabled
2. **syscall_fb_write(...)** - display a user defined string
3. **syscall_fb_write_dec(...)** - display a decimal number
4. **syscall_fb_write_hex(...)** - display a hexadecimal number

[3] 8042 keyboard controller: http://stanislavs.org/helppc/8042.html

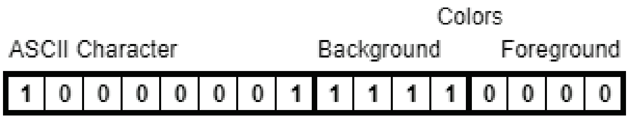
5. **syscall_fb_write_char(...)** - display a character
6. **syscall_log(...)** - log a message from user mode.
7. **syscall_shutdown(...)** - initiate system shutdown.
8. **syscall_reboot(...)** - reboot the system.
9. **syscall_read_file(...)** - reads file contents from filesystem.
10. **syscall_find_file(...)** - finds file in filesystem.
11. **syscall_memset(...)** - sets a memory block to value.

Chapter 3

Peripheral drivers

3.1 Terminal screen

The terminal screen routines implement a VGA text mode video driver using writing to frame buffer located at address `0x000B8000`. This allows the driver to display 25 rows of 80 characters, where each character is represented by a 16 bit value. The highest 8 bits encode ASCII value, bits 7-4 the background colour and 3-0 the foreground text color, as shown in Figure 3.1.



1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0

Figure 3.1: In-memory binary representation of letter A, displayed in white color with black background.

3.1.1 Initialising the terminal screen

To start using the terminal screen, our kernel entry point first calls initialisation function `init_terminal()` shown in Listing 3.1, which resets colours and erases all characters on screen.

```c
void clear_terminal() {
    //Set all characters to ' '
    for (int i = 0; i < FB_COL_HEIGHT; i++) {
        for (int j = 0; j < FB_ROW_LEN; j++) {
            fb_write_colour_char(get_fb_pos(i, j), BLANK, WHITE, BLACK);
        }
    }

    set_fb_pos(0, 0); //set cursor position to start
    fb_move_cursor(tc.fb_pos); //move cursor to position;
}

void init_terminal() {
    set_foreground_color(WHITE);
    set_background_color(BLACK);
    clear_terminal();
}
```

Listing 3.1: Terminal initialisation.

3.1.2 Displaying data

Because we often want to display various kinds of data in our terminal, the driver implements multiple functions, shown in Listing 3.2 for printing single characters, null-terminated strings and numbers in decimal and hexadecimal representations.

```c
/**
 * Write a character c to screen using specified colors
 * @param i Address in framebuffer where we are writing
 * @param c Character to be written
 * @param fg Colour of text
 * @param bg Colour of background
 */
void fb_write_clr_char(uint32 pos, char c, uint8 fg, uint8 bg);

/**
 * Write a string to screen at current position
 * @param buf Array of character to be written
 * @param len Length of text to be written
 */
```

```
void fb_write(int8 *buf, int32 len);

/**
 * Print a number to terminal in base 16
 * @param x Number to be displayed.
 */
void fb_write_hex(uint64 x);

/**
 * Print a number to terminal in base 10
 * @param x Number to be displayed.
 */
void fb_write_dec(int64 x);
```

Listing 3.2: Terminal screen interface functions.

The most basic function (Listing 3.3) merely prints the given character in specified colour at set position by setting ASCII value and colour value at an appropriate offset relative to starting position of the frame buffer.

```
void fb_write_clr_char(uint32 addr, int8 c, uint8 fg, uint8
    bg) {
    fb[addr] = c;
    fb[addr + 1] = (bg << 4) | (fg);
}
```

Listing 3.3: Function for printing a single ASCII character on screen.

Cursor movement is achieved through sending location data to an appropriate port. Since the `outb` function writes a single byte to a given port, we need to make two such calls. We first announce our intent (using command port `0x3D4`) to send the high bits of the location indicator and then send the actual data to a data port `0x3D5`. This is repeated for the lower bits. Such function will produce a blinking cursor at given location.

```
void fb_move_cursor(uint16 pos) {
    pos = pos / 2; //needed due to 16b resize;
    //First request write to upper byte, send upper, then
        send lower
    outb(FB_COMMAND_PORT, FB_HIGH_BYTE_COMMAND);
    outb(FB_DATA_PORT, ((pos >> 8) & 0x00FF));
    outb(FB_COMMAND_PORT, FB_LOW_BYTE_COMMAND);
    outb(FB_DATA_PORT, pos & 0x00FF);
}
```

Listing 3.4: Function for setting cursor position.

3.2 Serial ports and logging

While serial ports and RS-232 protocol have been largely succeeded by USB and other modern interfaces and hardware connections are rare, the software implementations still have some uses, mainly due to ease of programming and direct hardware support. In this project, the serial port has been used as a general method of communicating with host OS and a logger facility.

BIOS generally provides at least two COM ports - shown implementation uses four, to separate various sources of messages and allow easier debugging.

3.2.1 Implementing serial port

Listing 3.5 shows the core of serial port communication implementation. Each COM port needs to be initialised with baud rate to determine its speed and a line configuration. I used a baud rate of 57.600 bauds with line configuration 8N1 - 8 bits of data, no parity bits and one stop bit.

```
uint16 serial_ports[4] = {COM1, COM2, COM3, COM4};

void conf_baud_rate(uint16 port, uint16 divisor) {
    outb(port + 3, 0x80);
    outb(port, (divisor >> BYTE) & LOW_MASK);
    outb(port, divisor & LOW_MASK);
}

void conf_line(uint16 com) {
    outb(com + 3, LINE_CONFIG);
}

void serial_char_write(uint16 port, int8 c) {
    while (!transmit_line_empty(port)) {}
    outb(port, c);
}
```

Listing 3.5: Functions comprising main serial port functionality.

ASCII characters can be written to a serial port by `serial_char_write(port, char)` function, which first checks if transmit buffer is empty before sending requested character. Other implemented functions in serial port library employ multiple such calls to print strings, decimal numbers and hexadecimal numbers to serial port when needed.

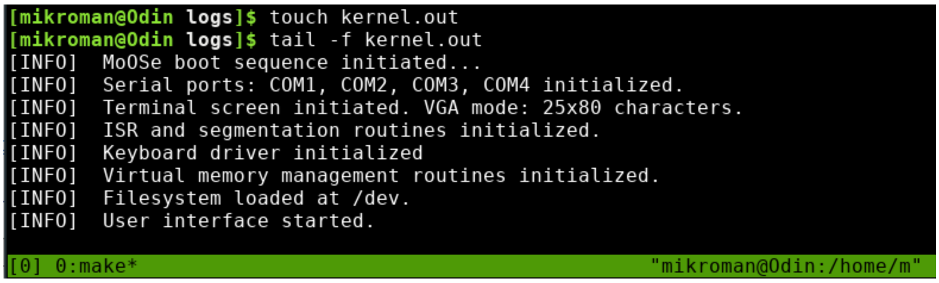
3.2.2 Logger extensions

Logging functionality is extended via `logger.c` file, that provides multiple log function. For this, `logger.h` file defines KERNEL, USER, DEV and OTHER log files, corresponding to COM ports 1 through 4. Thus, a developer can use `klog` (kernel), `ulog` (user), `dlog` (dev) or `olog` (other) function to write to any given log file. For every port, we also allow event severity statuses: ERR, WARN and INFO - this status will be prefixed to the message when writing to log file. Below, function `log(...)` shows the generic log implementation that handles all writing to log files.

```c
void log(log_src facility, sev severity, const char *msg) {
    switch (facility) {
        case KERNEL:
            out = COM1;
            break;
        ...
        default:
            out = COM4;
    }

    char *severity_str;
    switch(severity) {
        case ERR:
            severity_str = "[ERROR] ";
            break;
        ...
        default:
            severity_str = "[NONE]  ";
    }

    serial_write(out, severity_str, 100);
    serial_write(out, msg, 200);
}

void klog(const char *format_str, sev severity) { log(KERNEL,
    severity,format_str); }
```

Listing 3.6: Parts logger functions.

Screenshot 3.2 shows a sample kernel log output during system boot. Not that all messages are prefixes by [INFO] status string to denote that message is non-critical, also allowing easier log searching with `grep` and other tools.



```
[mikroman@Odin logs]$ touch kernel.out
[mikroman@Odin logs]$ tail -f kernel.out
[INFO]  MoOSe boot sequence initiated...
[INFO]  Serial ports: COM1, COM2, COM3, COM4 initialized.
[INFO]  Terminal screen initiated. VGA mode: 25x80 characters.
[INFO]  ISR and segmentation routines initialized.
[INFO]  Keyboard driver initialized
[INFO]  Virtual memory management routines initialized.
[INFO]  Filesystem loaded at /dev.
[INFO]  User interface started.

[0] 0:make*                                         "mikroman@Odin:/home/m"
```

Figure 3.2: Example screenshot of a kernel log output during boot process.

3.3 Keyboard

To allow the user means of interaction with the system, a keyboard driver is implemented which supports standard alphanumerical keys as well as monitoring state of special keys (Caps Lock, Shift, Control, etc.), allowing us to modify application behaviour based on this. The system uses UK layout by default (shown in Listing 3.3), but other layouts can be implemented or loaded via kernel modules.

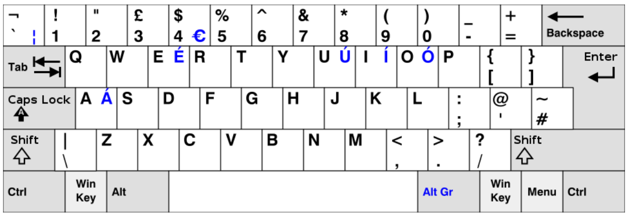


Figure 3.3: UK keyboard layout, from Wikipedia [10].

3.3.1 Implementing keyboard driver

Keyboard driver is initialised by registering an interrupt handler on IRQ1, passing it a pointer to `void kbd_hadler(...)` (shown in Listing 3.7) which is called on interrupt trigger.

```
uint8 read_scan_code() {
    return inb(KBD_DATA_PORT);
}

static void kbd_handler(registers_t regs) {
    //read scancode from port
    uint8 scancode = read_scan_code();

    if (scancode & KEY_RELEASE_MASK) { //check if releases
        //Negate mask to remove this bit from scancode
        reset_special(scancode &~KEY_RELEASE_MASK);
    } else {
        set_special(scancode);
        put_char(kmp[scancode]);
    }
}
```

Listing 3.7: Keyboard reading.

Upon interrupt trigger, we first read a keyboard scancode from keyboard input port `0x60`. Since interrupt is triggered for every press and release of a key, we check which is the current action by using `0x80` key mask. On every press, we simply write an ASCII character representation of a key press to the terminal screen. In both cases, we also check the status of our special keys and whether we need to set/unset any of them, as well as modify the current scan code if necessary (ex. when Shift modifier is pressed, make letter uppercase).

3.3.2 Keymap

By default, the implemented kernel uses a standard UK layout keymap. Key values are stored in an array (shown in Listing 3.8) with their ASCII values. Certain keys (function keys, modifiers, etc.) are stored as value 0 since their ASCII representation are not printable, but rather their press is handled directly by the keyboard interrupt handler function and treated accordingly

- modifier keys, Backspace, Enter key...

```
static uint8 kmp[KEYMAP_LEN] = {
        0x0, //none
        0x1B, //ESC
        '1', '2', '3', '4', '5', '6', '7', '8', '9', '0',
        '-', '=', '\b', '\t',
        'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p',
        '[', ']', '\n', 0x0,
        'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';',
        '\'', '`', 0x0, '\\',
        'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/',
        ...
```

Listing 3.8: Part of a UK keymap implementation.

3.3.3 Special keys

This implementation uses a special structure to hold the state of special keys: `Alt, Ctrl, Left/Right Shift, Caps Lock, Num Lock` and `Scroll Lock`.

With `Ctrl`, `Shift` and `Alt`, every press of the corresponding key sets the value to 1, while the release resets it back to 0. When using any of the `Lock` keys, the first press will set the value to 1, but only releasing and pressing again will reset it. This structure can be used to check modifier key state at any time and modify system behaviour accordingly. Using this, the developer can also easily bind certain actions to specific keys (like system shutdown, or screen clearing).

3.4 Filesystem

3.4.1 Virtual File System Interface

The implemented filesystem interface follows the basic design of Linux's Virtual File System (VFS [8]), which is an abstraction on top of a filesystem implementation. It is meant to provide a common interface for reading and storing files across multiple common file systems.

Thus, every file in filesystem is represented by a structure containing its metadata: file name, length in bytes, identification number and implementation specific fields. Since different filesystems also have different methods of accessing files, these structures define common format of function callbacks via C's function pointers. Thus, any filesystem driver can implement its' own version of file writing or reading functions. One such version of this file entry structure is shown in Listing 3.9

```c
typedef struct {
    char name[255];     // The filename.
    uint32 flags;       // describes file type (dir, file)
    uint32 inode;       // file id
    uint32 length;      // filesize in bytes.
    read_type_t read;   //func callback for reading content
    readdir_clb readdir; //callback for dir listing
    finddir_clb finddir; //callback for searching
    fs_node_t *ptr; // file ptr for mountpoints.
} file_entry;
```

Listing 3.9: Structure containing file metadata.

Since this implementation uses only read-only initial ramdisk filesystem, some structure elements are ommited - generic implementation should also include write function callback, as well as permission mask and others which are note needed in a single user system.

```c
uint32 read_fs(file_entry *node, uint32 offset, uint32 size,
   uint8 *buf) {
    return node->read != NULL ? node->read(node, offset, size
       , buf) : NULL;
}
dir_entry *readdir_fs(file_entry *node, uint32 index) {
    return is_dir(node->flags) && node->readdir != NULL ?
    node->readdir(node, index) : NULL;
}

fs_node *finddir_fs(file_entry *node, int8 *name) {
    return is_dir(node->flags) && node->finddir != NULL ?
    node->finddir(node, name) : NULL;
}
```

Listing 3.10: Generic VFS function handlers.

Listing 3.10 shows implementations of generic VFS interface file handling functions. When a file callback is triggered by a system call or an internal kernel function, the request is first passed through VFS that check for flags and presence of the callback and only then triggers actual filesystem action.

3.4.2 Construction of initrd

Initrd is a static, read-only, filesystem, existing on the bootable ISO image alongside with kernel code. The `initrd` disk image is constructed before bootable ISO image is finalised, and in this step, any file can be added. To achieve this, we use a simple piece of C code, that allows us to specify local files to be added. In its current iteration, it supports only flat filesystem model, without nested directories.

Beacuse `initrd` is loaded as GRUB module, and we currently ony have one module, the location can be easily read from multiboot header structure:

```
uint32 initrd_location = *((uint32 *) mboot->mods_addr);
```

When mounting the initrd filesystem in kernel boot process, we need to first set up our root filesystem node:

```
initrd_root = (file_entry*) malloc (sizeof(file_struct));
strcpy(initrd_root->name, "ramdisk");
initrd_root->inode = NULL;
initrd_root->length = NULL; //just a dir node
initrd_root->flags = IS_DIRECTORY;
initrd_root->read = NULL; //cannot be read
initrd_root->readdir = &initrd_readdir;
initrd_root->finddir = &initrd_finddir;
```

Listing 3.11: Loading data for root node.

During addition of files to `initrd` disk image, the script also sets final file count to filesystem header. Now we can easily iterate over filesystem address space and load the file metadata:

```
for (i = 0; i < initrd_header->num_files; i++) {
    strcpy(root_nodes[i].name, &file_headers[i].name);
    root_nodes[i].length = file_headers[i].length;
    root_nodes[i].inode = i; //simple sequential nums
    root_nodes[i].flags = IS_FILE;
    root_nodes[i].read = &initrd_read;
    ... //Other fields set to NULL
```

Listing 3.12: Loading metadata for files.

3.4.3 Filesystem interaction

The current interface for mounted filesystem exposes two chief function, `find` and `read`. The first allows a user or developer to execute a search for file that will return a file pointer, containing file size, name and other metadata. Read function (implementation shown in Listing 3.13) fetches file contents from disk image to a buffer that can be used for further purpose.

```
uint32 initrd_read(file_entry *node, uint32 off, uint32 size,
    uint8 *buffer) {
    initrd_file_header header = file_headers[node->inode];
    if (off > header.length) return NULL;
    if (off + size > header.length)
        size = header.length - off;
    memcpy(buffer, (uint8*) (header.offset+off), size);
    return size;
}
```

Listing 3.13: Function for reading file contents from ramdisk filesystem.

This functions can now be used in console commands and via syscalls, providing us with ability to list files in filesystem as well as read their contents.

Chapter 4

User mode

The x86 architecture provides a mechanism called privilege rings, which allows fault protection and better security. This is done by a hierarchy of protection levels, where outermost privilege level has the least privileges and is usually exposed to the user. In this mode, we usually implement the user-facing applications that communicate with kernel via system calls. This allows for a level of abstraction, as well as a protection mechanism that disallows indiscriminate memory access.

4.0.1 Privilege levels

The multiple levels of protection were first pioneered by Multics operating system, a predecessor of UNIX - since the underlying hardware, a GE645 mainframe did not have the support for this novel feature, Multics developers implemented ring protection in software. Nowadays however, most CPU architectures (like Intel x86 used here) implement some form of it, although not all operating systems use this protection to the full extent.

The schematic in Figure 4.1 shows the privilege rings available for usage in x86 architecture. In general, the most privileged mode, called Ring 0, is used for kernel facilities, Rings 1 and 2 for device drivers and Ring 3 for running unprivileged code, usually the software users interact with. However, these delineations are mostly guidelines and not hard rules, and different systems will use a different amount of protection rings. In our system, Ring 0, which is the default protection ring we boot into, is used for hosting both kernel code and peripheral drivers, while Ring 3 runs the user interface and

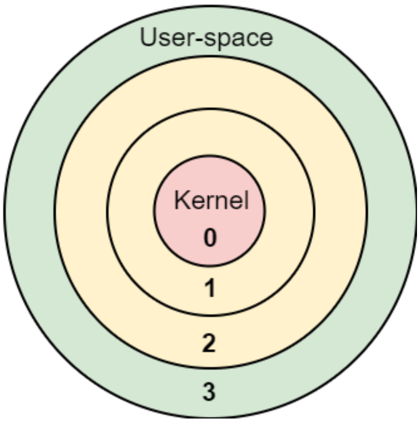


Figure 4.1: Diagram of privilege rings.

inbuilt utilities. Rings 1 and 2 are unused.

The general procedure calls for kernel to be initialised first in Ring 0 and then running a special subroutine to trigger user-mode, continuing the boot of the user-facing part of the operating system.

4.0.2 Switching mechanism

The IA-32/x86 architecture does not have a straightforward mechanism for switching from privileged to user mode. We fake the user mode switch by using a subroutine, shown in Listing 4.1. This subroutine firstly switches kernel mode segment selector with user mode selectors, and then loads the desired destination address (user-mode C code) onto stack.

```
GLOBAL switch_mode
EXTERN user_mode_code

switch_mode:
    cli ;disable interrupts
    mov ax, 0x23 ; set user selector
    mov ds, ax   ; for all segments
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov eax, esp
    push 0x23 ;user data segment selector
    push eax ;stack address
    pushf
    push 0x1B ;user code segment selector
    push user_mode_code ;returning address
    iret
```

Listing 4.1: Assembly function for triggering user mode.

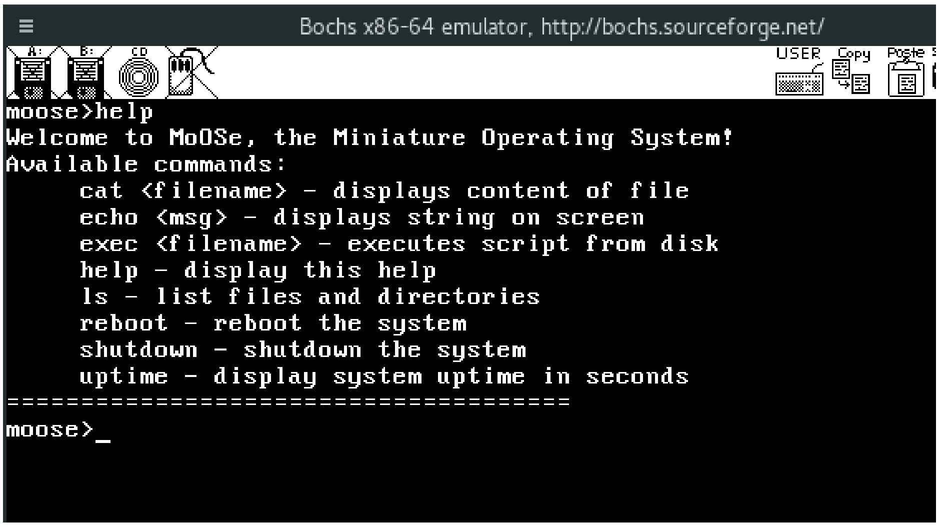
The `IRET` instruction returns from a subroutine by popping the instruction pointer, return code segment selector and `EFLAGS` register value from stack. Since we replaced previous caller's values with our own, we fool the CPU into jumping to our desired function. With this, we can now implement other applications, assuming we initialised system calls accordingly.

Chapter 5

User interaction console

To provide some interactive experience, the implemented operating system exposes a simple shell that starts up after boot process. The shell is inspired by GNU/Linux shells, using REPL (Read-Evaluate-Print Loop) principle.

Screenshot 5.1 shows the console displaying the output of `help` command. At boot, the display is cleared and the prompt is displayed. During user's typing, the presses are buffered and displayed on screen. This implementation is shown in Listing 5.1 and is triggered via keyboard interrupt handler. When press of enter key is detected (a newline character), the buffer is flushed to command decoding and execution.



```
moose>help
Welcome to MoOSe, the Miniature Operating System!
Available commands:
     cat <filename> - displays content of file
     echo <msg> - displays string on screen
     exec <filename> - executes script from disk
     help - display this help
     ls - list files and directories
     reboot - reboot the system
     shutdown - shutdown the system
     uptime - display system uptime in seconds
======================================
moose>_
```

Figure 5.1: Output of help command.

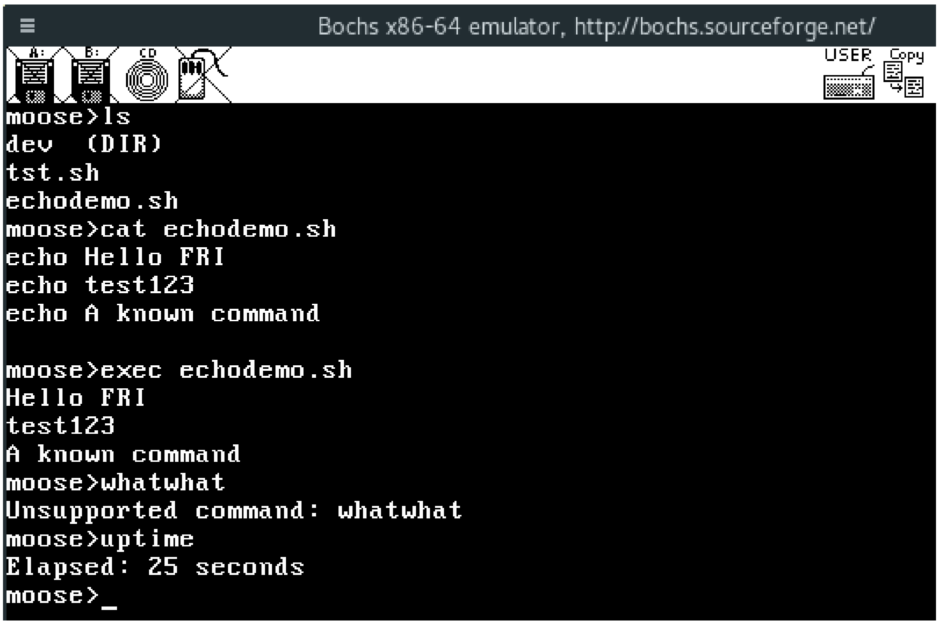
```
void put_char(char c) {
    if (!script) fb_write_char(c);
    if (c == '\b') {
        log(KERNEL, INFO, "Backspace");
        idx = idx > 0 ? idx - 1 : idx;
        command[idx] = '\0';
    } else if (c == '\n') {
        execute();
        print_prompt();
        memset(command, 0, 255);
        idx = 0;
    } else {
        command[idx] = c;
        idx++;
    }
}
```

Listing 5.1: Console command reading.

Driver also implements fully working Backspace key for fixing typing mistakes. During command decoding, the buffer is tokenized and the system extracts specific commands and their parameters. Commands are compared to an array of known string using our own `strcmp(...)` implementation and accordingly, the correct function is executed, be it system reboot, displaying text message or others. If an unrecognised command is encountered, the system will react by printing a string signifying this to the user on screen. This can be seen in screenshot shown in Figure 5.2.

5.0.1 Commands

- **shutdown** - shuts down the operating system
- **reboot** - reboots the operating system via CPU reset pin
- **help** - displays a help text, listing all available commands and their short descriptions
- **uptime** - displays the system uptime in seconds. This command reads the status of Programmable Interrupt Timer ticking at 100Hz and converts this to second count.



```
moose>ls
dev  (DIR)
tst.sh
echodemo.sh
moose>cat echodemo.sh
echo Hello FRI
echo test123
echo A known command

moose>exec echodemo.sh
Hello FRI
test123
A known command
moose>whatwhat
Unsupported command: whatwhat
moose>uptime
Elapsed: 25 seconds
moose>_
```

Figure 5.2: Using exec and uptime commands.

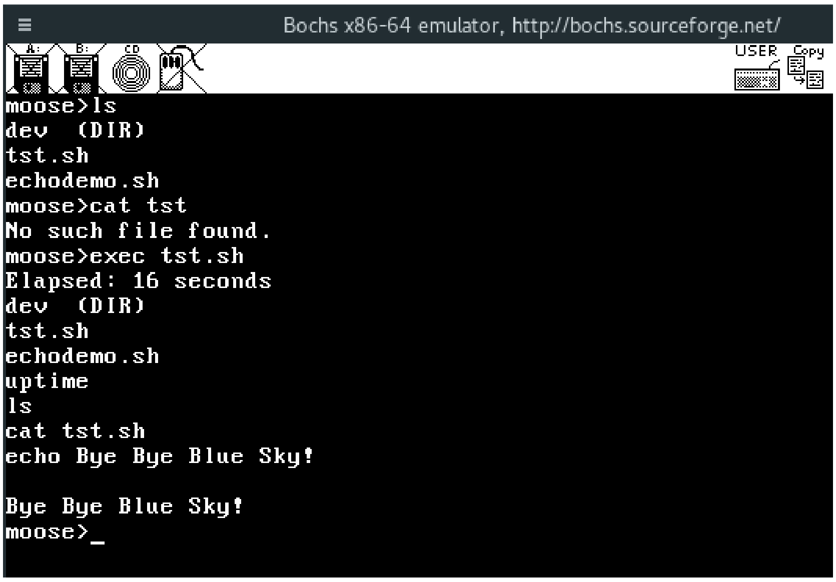
- **name** *string* - used with a single parameters to set string displayed in console prompt
- **ls** - lists files present in `/dev` filesystem
- **exec** *script_name* - executes script containing console commands that is loaded from filesystem
- **cat** *file_name* - display contents of the file
- **echo** *string string*2 - display a string on screen
- **size** *filename* - displays size of file in bytes on screen

Executable scripts (an example in Listing 5.2) are inserted into initial ramdisk image before boot and can include any of the above commands, even executing sub-scripts.

```
echo Hello FRI
uptime
ls
cat tst.sh
echo Bye Bye Blue Sky!
```

Listing 5.2: Example of an executable script.

The result of the short script above is shown on the bottom of screenshot depicted in Figure 5.3.



```
moose>ls
dev  (DIR)
tst.sh
echodemo.sh
moose>cat tst
No such file found.
moose>exec tst.sh
Elapsed: 16 seconds
dev  (DIR)
tst.sh
echodemo.sh
uptime
ls
cat tst.sh
echo Bye Bye Blue Sky!

Bye Bye Blue Sky!
moose>_
```

Figure 5.3: Result of executing tst.sh script loaded from filesystem.

Chapter 6

Development environment and toolchain

Most commonly, operating systems are developed using system programming languages that allow the programmer to closely interact with hardware. Following this, the operating system described in this thesis uses C programming language, augmented by assembly language for x86 architecture, since resources for both are widely available and they allow finely-tuned hardware and memory interaction. All source code was compiled to 32-bit executable binaries.

6.1 Compilation tools

Multiple tools were used to transform and prepare the source code for insertion into a bootable disk image and further execution. `make` build system was used to link these tools together for streamlining the compilation process.

6.1.1 GCC suite

GCC[1] (standing for GNU Compiler Collection) is a suite of tools, first released in May 1987, which includes multiple compiler for a variety of languages like C, C++, Fortran, Java, Ada, etc. GCC runs on a wide set of architectures and is therefore a singularly useful tool for operating systems development. In this thesis, C sources were built using `gcc` cross-compiler

[1] Available at: https://gcc.gnu.org/

[7] v6.3.0, specially compiled to produce i686-elf binaries.

The compilation options (shown in Listing 6.1) ensure that all C sources are compiled without inclusion of standard C library and built-in functions, as well as output all warnings about potential problems.

```
> i686-elf-gcc -std=gnu99 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -c -Wall
```

Listing 6.1: GCC command line configuration.

6.1.2 NASM

NASM[2] is an open-source assembler and disassembler for x86 architecture, originally written by Simon Tatham. It can produce various binary formats, including but not limited to ELF, COFF and Mach-O. It is especially useful because of its ability to create flat binary files often used for bootloaders and the like in OS development. Assembly programs written for NASM assembling use Intel's assembly syntax.

Listing 6.2 shows how Assembly sources are assembled into ELF binaries.

```
> nasm -felf $SRCS
```

Listing 6.2: NASM command line configuration.

6.1.3 GNU linker

GNU linker `ld`[3] is a part of `binutils` package, provided by GNU Project. It supports most input and output formats, like ELF, PE, COFF, etc. It is used to combine object and archive files and resolve symbol references. In this project, linker combines the objects, compiled from source code and produces an ELF binary, that is later inserted into a bootable image.

With command shown in Listing 6.3, the compiled sources are linked and combined into a single ELF binary, which is later embedded into the disk image.

```
> ld -Tlink.ld -melf_i386 $SRCS -o oskernel.elf
```

Listing 6.3: LD linker command line configuration.

[2] Available at: http://www.nasm.us/

[3] Available as part of binutils package: https://www.gnu.org/software/binutils/

6.2 Running the kernel

The source files are put through multiple steps to produce executable kernel. We first compile the C sources with `gcc`, assemble ASM sources using `nasm` and finally link them together using `ld` linker. This produces an ELF file, which is used to generate the final bootable ISO image.

The ISO image is used by `Bochs`[4], an open-source IA-32 and x86-64 emulator. It is able to emulate core components of a computer, as well as common hardware peripherals.

Following Bochs configuration file (Listing 6.4) was used to run the virtual machine. It causes the system to boot from a bootable CD-ROM image `test.iso`, sets Bochs and OS log files and synchronises timers with host OS. Emulation ROM images for BIOS and VGA are provided by Bochs installation in host OS.

```
megs:               32
display_library:    x
romimage:           file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage:        file= /usr/share/bochs/VGABIOS-lgpl-latest
ata0-master:        type=cdrom, path=test.iso, status=inserted
boot:               cdrom
log:                ./logs/bochs.log
clock:              sync=realtime
cpu:                ips=1000000
com1:               enabled=1, mode=file, dev=./logs/kernel.out
com2:               enabled=1, mode=file, dev=./logs/user.out
com3:               enabled=1, mode=file, dev=./logs/debug.out
com4:               enabled=1, mode=file, dev=./logs/err.out
```

Listing 6.4: Bochs virtual environment configuration.

[4] Available at: http://bochs.sourceforge.net/

Chapter 7

Conclusions

Development of a modern operating system is a very open-ended task, since different users and developers imagine different core services and tools that should be included in such a system. Operating systems with large userbases have become so enormous, that no single person can objectively have a handle on everything going on in the codebase. Operating systems built by a small group of enthusiasts have been mostly phased out of mainstream usage, especially with rise of users who demand impressive graphical interfaces, streamlined usage and a multitude of obscure tools. Even GNU/Linux collective has grown to a large size, including a number of supporting companies and acclaimed developers. Many other niches operating systems are merely finely tuned and tweaked versions of each other (as is often the case with Unix distributions).

The main aim of this thesis was to investigate the development process and structure of a minimal operating system with a monolithic kernel, inspired by core GNU/Linux concepts [6], that would allow the developer further extensibility and the user some simple tools to interact with. The final product features complete kernel with memory management facility and several peripheral drivers as well as working user mode, interrupt handling and system calls. Its source code is available in an online repository [11] and can be built with the array of tools described in this thesis. Due to time constraints and some compatibility issues, FAT16 filesystem and multitasking implementations have been dropped for the moment, though they remain on the roadmap and the base support exists in the codebase via Virtual File System (VFS) interface and virtual memory management respectively.

Such additional modules can be integrated relatively easily via initialisation functions in C entry code, interrupt handler registration and system call registration functions. With this, full support for mounting file-systems, multitasking and process forking, network drivers and more can be added in the future. The final step to usability would be recompiling and porting commonly used Unix tools, as well as implementing a proper graphics driver, dropping reliance on inbuilt VGA support and adding a graphical user interface.

Bibliography

[1] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[2] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. Available at: `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`. [Accessed: 10. 7. 2017].

[3] Free Software Foundation. Grub's multiboot specification. Available at: `https://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Header-layout`. [Accessed: 1. 8. 2017].

[4] NetMarketShare.com. Desktop operating system market share. Available at: `https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0`. [Accessed: 29. 8. 2017].

[5] Xen Project. Introduction to unikernels. Available at: `https://wiki.xenproject.org/wiki/Unikernels`. [Accessed: 3. 8. 2017].

[6] Eric S. Raymon. *The Art of UNIX Programming.* Addison-Wesley, 2003.

[7] Crowd sourced documentation. Using a cross compiler. Available at: `http://wiki.osdev.org/GCC_Cross-Compiler`. [Accessed: 6. 4. 2017].

[8] tldp.org. Virtual file system documentation. Available at: `http://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html`. [Accessed: 3. 8. 2017].

[9] W3Techs.com. Website server operating system market share. Available at: `https://w3techs.com/technologies/overview/operating_system/all`. [Accessed: 29. 8. 2017].

[10] Wikipedia. Uk keyboard layout example. Available at: https://en.wikipedia.org/wiki/British_and_American_keyboards#/media/File:KB_United_Kingdom.svg. [Accessed: 29. 8. 2017].

[11] Tine Šubic. Moose source code repository. Available at: https://github.com/MikroMan/moOSe. [Accessed: 1. 8. 2017].