Not so big Operating System (nsbOS) v1.0

Author: Nikola Kušlaković

Occupation: Student at Faculty of Technical Sciences, University of Novi Sad, Serbia

Date: 8. June 2023.

Source: github.com/nkusla/nsbOS

Introduction

This is my take on making simple yet usable and **not so big Operating system**. This operating system is not meant to be robust and have all advanced components that are present in modern operating systems. The point of this hobby project is to learn and fully understand low-level operating system events which are mostly taken for granted today. Besides that, this journey required understanding how compiler, linker, build system and debugger work together and can be used to develop software.

Bootloader and some small parts of kernel are written in `x86 assembly` and the rest is written in `C`. Everything is compiled and tested on `Intel 80386 (i386)` processor which is a CISC 32-bit architecture processor.

Required tools and installation

This project uses following things for development and building toolchain:

- `nasm` (Netwide assembler) - assembles bootloader and some parts of kernel code
- `gcc` (GNU compiler collection) - used for compiling most of kernel code and device driver code
- `ld` (GNU linker) - combines multiple object files created by assembler and compiler
- `gdb` (GNU debugger) - used for connecting directly to emulator while OS is being executed
- `objdump` - used for examening object and elf file formats
- `make` - used for running scripts for building and automatization
- `qemu` - emulates x86 architecture (Intel i386)

Most of these tools come pre-installed on all Linux distros. Installation on Arch-based distros:

```
sudo pacman -S binutils nasm gcc gdb make qemu-desktop
```

Bootloader

When the computer turns on, it loads and starts executing the BIOS code which is usually located in some kind of read-only memory (ROM). The BIOS then searches for bootable devices. In this case, it looks for a floppy disk that contains a boot sector. The boot sector is the first memory block, which is 512 bytes in size and contains a special value known as the *magic number*.

The magic number serves as a flag for the BIOS, indicating that the sector contains executable bootloader code. It conventionally resides at the end of the sector (last 2 bytes) and is represented by the value `0x55aa`. So byte `0x55aa` should be located at address `0x1fe`.

```
00000000  bd 00 7c 89 ec b4 00 b0  03 cd 10 68 8d 7c e8 0e  |..|........h.|..|
00000010  00 6a 12 e8 20 00 68 da  7c e8 03 00 e8 e0 00 55  |.j.. .h.|......U|
00000020  89 e5 8b 76 04 b4 0e 8a  04 08 c0 74 05 cd 10 46  |...v.......t...F|
00000030  eb f5 89 ec 5d c3 55 89  e5 8b 5e 04 b4 02 88 d8  |....].U...^.....|
00000040  b2 00 b6 00 b5 00 b1 02  bb 00 00 8e c3 bb 00 7e  |...............~|
00000050  cd 13 72 0f 8b 5e 04 38  d8 75 08 68 ac 7c e8 be  |..r..^.8.u.h.|..|
00000060  ff eb 08 68 c5 7c e8 b6  ff eb fe 89 ec 5d c3 00  |...h.|.......]..|
00000070  00 00 00 00 00 00 00 ff  ff 00 00 00 9a cf 00 ff  |................|
00000080  ff 00 00 00 92 cf 00 17  00 6f 7c 00 00 57 65 6c  |.........o|..Wel|
00000090  63 6f 6d 65 20 74 6f 20  6e 73 62 4f 53 20 62 6f  |come to nsbOS bo|
000000a0  6f 74 6c 6f 61 64 65 72  21 0d 0a 00 44 69 73 6b  |otloader!...Disk|
000000b0  20 73 75 63 63 65 73 73  66 75 6c 6c 79 20 72 65  | successfully re|
000000c0  61 64 0d 0a 00 45 72 72  6f 72 20 72 65 61 64 69  |ad...Error readi|
000000d0  6e 67 20 64 69 73 6b 0d  0a 00 53 77 69 74 63 68  |ng disk...Switch|
000000e0  69 6e 67 20 74 6f 20 33  32 2d 62 69 74 20 70 72  |ing to 32-bit pr|
000000f0  6f 74 65 63 74 65 64 20  6d 6f 64 65 0d 0a 00 fa  |otected mode....|
00000100  0f 01 16 87 7c 0f 20 c0  66 83 c8 01 0f 22 c0 ea  |....|. .f...."..|
00000110  14 7d 08 00 66 b8 10 00  8e d8 8e d0 8e c0 8e e0  |.}..f...........|
00000120  8e e8 bd 00 7c 00 00 89  ec eb 00 e9 d0 00 00 00  |....|...........|
00000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200  e8 11 0c 00 00 eb fe 66  90 66 90 66 90 66 90 90  |.......f.f.f.f..|
00000210  60 e8 b4 0a 00 00 61 83  c4 08 cf 6a 00 6a 20 eb  |`.....a....j.j .|
00000220  ef 6a 00 6a 21 eb e9 6a  00 6a 22 eb e3 6a 00 6a  |.j.j!..j.j"..j.j|
00000230  23 eb dd 6a 00 6a 24 eb  d7 6a 00 6a 25 eb d1 6a  |#..j.j$..j.j%..j|
00000240  00 6a 26 eb cb 6a 00 6a  27 eb c5 6a 00 6a 28 eb  |.j&..j.j'..j.j(.|
```

Figure 1: Magic number at the end of bootsector

When BIOS finds boot sector it loads it at address `0x7c00` in memory and passes the execution to the bootloader. Bootloader is compiled so all labeles inside of assembly code are calculated relatively to this address. After that, bootloader needs to read other sector from the disk. These sectors effectively contain the entire operating system and other user programs.

Luckly our good old friend BIOS has some built in routines that can read floppy disk contents. Bootloader reads from floppy disk by putting arguments in registers and executing BIOS system call. This call is invoked with `int 0x13` instruction. Bootloader of `nsbOS` copies contents of floppy disk starting from address `0x7e00` - right behind where the bootloader is placed in memory. See *Figure 2* for better understanding.

Before passing execution to the operating system, the bootloader is responsible for transitioning the CPU from real mode to protected mode. Real mode, operating in 16-bit, exists for compatibility reasons and has limitations such as a 1 MiB memory access limit and direct mapping to physical addresses.

In contrast, protected mode operates with a 32-bit address space and provides enhanced execution capabilities and memory protection. The transition from real to protected mode involves the following steps:

- Setting up the Global Descriptor Table (GDT): The GDT is a specialized data structure used by x86 processors to define memory segments and their access permissions
- Passing the GDT descriptor to the CPU: The CPU stores the location of the GDT to reference it during memory access
- Updating control registers: Control registers within the CPU are modified to activate protected mode and configure system behavior
- Updating segment registers: Segment registers within the CPU are updated to establish memory segmentation, enabling proper memory access and addressing

By completing these steps, the bootloader successfully switches the CPU from real mode to protected mode and passes the execution to the kernel.

NOTE: most of the things mentioned here are legacy things that BIOS does. Today, most operating systems don't use *legacy-BIOS* method in order to boot, instead they rely on UEFI which is newer standard for booting.

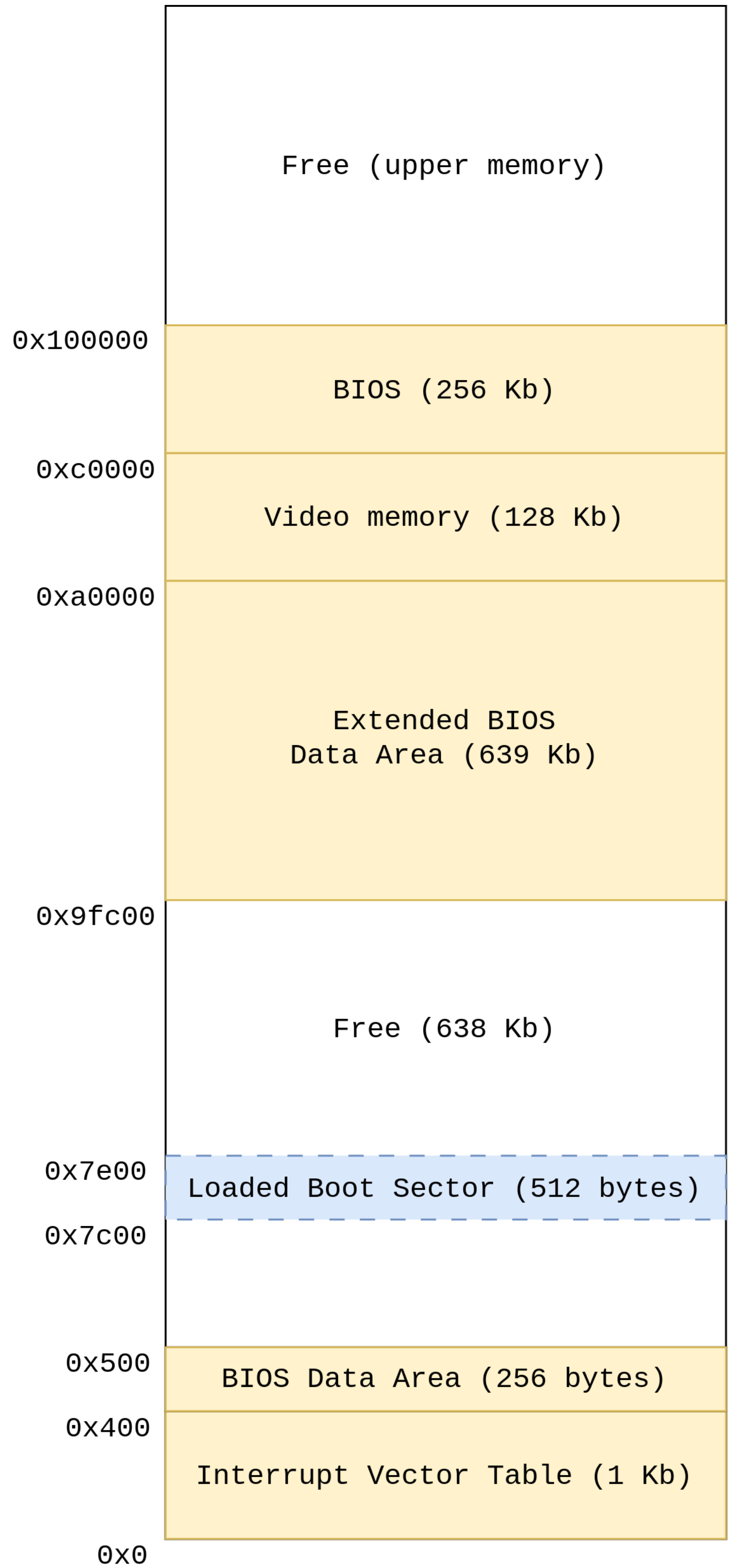

Figure 2: Lower memory layout

Drivers

To interact with physical devices such as the monitor, keyboard, and mouse, they need to be mapped to specific addresses in the I/O address space. The `x86` instruction set provides two useful functions, namely `in` and `out`, which enable the CPU to read data from I/O-mapped addresses and store that data in registers. These low-level functions are used for direct communication with the hardware.

Video driver

The video memory is memory-mapped starting from address `0xb8000`. For simplicity, the screen resolution is set to 25x80, which corresponds to the basic VGA mode called `text mode 0`. In this mode, we can only write colored text to the screen. Each character displayed on the screen is represented by 2 bytes. The first byte represents the ASCII value of the character, while the second byte represents the background and foreground colors. Following table shows how background and foreground colors are encoded in color byte:

Bit	7	654	3	210
Color	background bright bit	background	foreground bright bit	foreground

Following table shows how each color is encoded with 3 bits:

Value	Color
000	Black
001	Blue
010	Green
011	Cyan
100	Red
101	Meganta
110	Brown
111	Light Gray

Keyboard driver

The keyboard driver implemented in this project is designed for keyboards that use the PS/2 port for communication. The keyboard data register is mapped to the I/O address `0x60`. When a key is pressed on the keyboard, the corresponding scancode is stored in this data register. The driver reads the scancode from the address, parses it, stores it in the keyboard buffer, and displays it on the screen.

Kernel

Once the boot phase is complete, the kernel requires a mechanism to handle interrupts. Interrupts are special signals that indicate to the CPU that an urgent task needs to be processed, such as hardware events, exceptions, or other software-related events. These interrupts are managed by specific functions known as *interrupt routines*.

When an interrupt occurs, the CPU temporarily suspends its current execution and shifts to executing the interrupt routine assigned to that specific interrupt. The interrupt routine is responsible for handling the specific task associated with the interrupt.

Following things in order are done by the kernel:

- Initialization of the Interrupt Descriptor Table (IDT): The kernel sets up the IDT, which is a data structure that maps specific interrupt numbers to their corresponding Interrupt Service Routines (ISRs).
- Setting up the first 32 ISR entries: The kernel initializes the IDT with the necessary code addresses to handle the first 32 interrupt events, which include critical system exceptions and processor-defined interrupts.
- Remapping the primary and secondary Programmable Interrupt Controller (PIC) chips: The kernel configures the PIC chips to properly manage and prioritize hardware interrupt. PIC chisp are resposnible for passing hardware requests to the CPU directly, so the CPU doesn't need to ask every hardware device if it needs servicing.
- Setting up 15 IRQ entries: The kernel populates the IDT with the required ISRs to handle the remaining 15 interrupt requests (IRQs) generated by various hardware devices.
- Setting up software interrupt routines: The kernel includes additional entries in the IDT to accommodate software-generated interrupts, allowing software components to trigger specific system functions or services.
- Passing the IDT descriptor to the CPU: Once the IDT is fully configured, the kernel passes the descriptor, containing the IDT's memory address, to the CPU. This enables the CPU to efficiently access and execute the appropriate ISR when an interrupt occurs.

By performing these tasks, the kernel establishes the necessary infrastructure for interrupt handling.

Interrupt handling steps

Let's say software interrupt happened which was called using `int 0x80` assembly instruction. This instruction interrupts the CPU and the following steps describe what happens in order for this interrupt to get handeled:

- The CPU uses the interrupt number (in this case, `0x80`) to locate the corresponding entry in the Interrupt Descriptor Table (IDT), which contains the address of the interrupt handler.
- The interrupt routine associated with the interrupt number is a small piece of assembly code that performs specific actions. Depending on the type of interrupt, it may push the interrupt number and error code (if applicable) onto the stack.
- Before executing the interrupt routine, the CPU saves the state of the general-purpose registers and segment registers onto the stack. This creates an interrupt frame, which allows the interrupt routine to access any data passed through registers.

- Once the interrupt frame is established, the CPU transfers control to the interrupt routine. The routine executes the necessary tasks to handle the specific interrupt.
- Upon completion of the interrupt routine's execution, the interrupt frame is restored, freeing up the stack space that was allocated.
- The CPU returns from the interrupt and resumes the execution of the interrupted task.

Example of inner exception

Here is one interesting example that happend during development phase. Debugging showed that descriptors inside of GDT were wrongly configured, which caused `General Protection Fault` exception to happen. This was displayed on the screen when the OS booted:

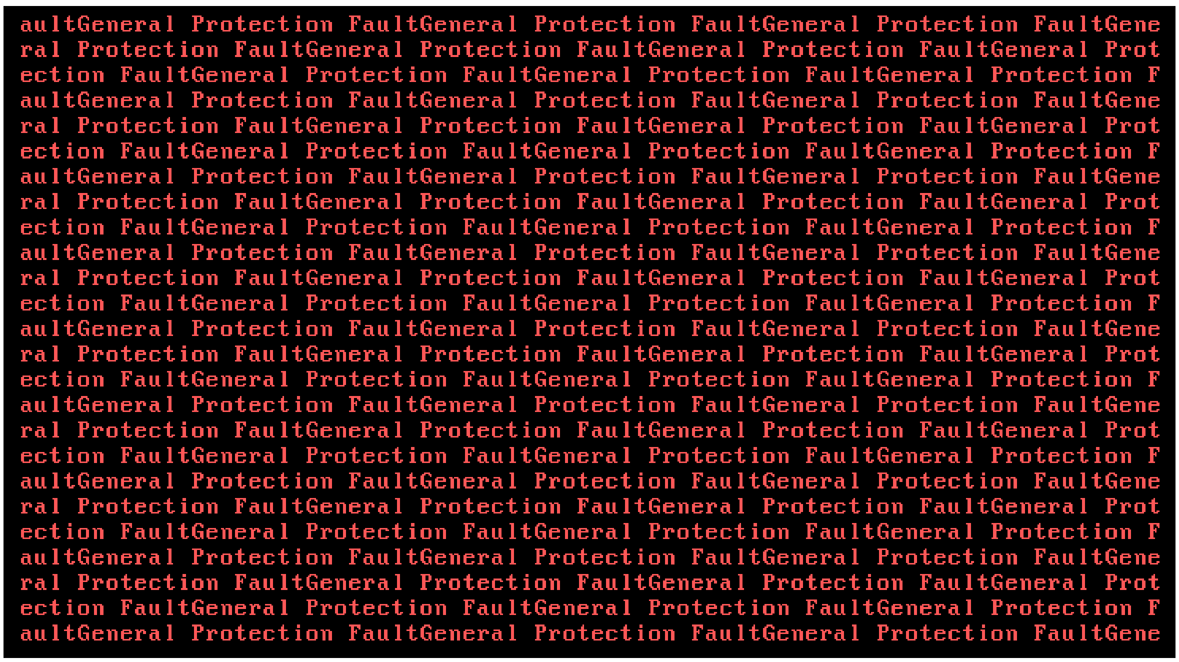

Figure 3: General Protection Fault

User program

User program is pretty simple at the moment it can only calculate sum of operands. User passes number of operand, then passes all operands and the total sum is displayed. In the background user program calls `printf` and `scanf` functions which parse string to integer (and vice versa) and execute system calls for writing to the screen and reading user input from keyboard. Kernel is the one who handles these system calls.

This component still needs some testing and developing. . .

Building

In root nsbOS directory there is a file called `Makefile`. This file is run with `make` and tells it how to integrate, build, compile and assemble everything. This file calls other `Makefiles` located in subdirectories where every subdirectory corresponds to one component of this operating system. This means that every component can be build and tested separatly.

In order to build the entire operating system, you can simply run `make` from root direcotry of nsbOS project. What happens in the background is as follows:

- Bootloader gets built
- Kernel gets built
- Shell gets built

After every of these steps, multiple `bin` files get created. Finally, bootloader, kernel and shell binary files are added together to produce final `disk.img` file. This file is a raw binary file and it basically represents our floppy disk which will get loaded into virtual machine. Besides disk image file, there are other object and elf files that get produced during build process. These files are saved in `build/` directory in separate subdirectories.

Building stage will also produce debug files which get stored inside of `debug/` directory.

Additonally, components can separatly be built by calling `make` and passing component name. For example this command will only build shell:

```
make shell
```

Running

Operating system is intended to run in `qemu` virtual machine to emulate entire environment. Running the OS in virtaul machine can be done by executing following command:

```
make run
```

This will load `disk.img` as a floppy disk and run virtual machine with some other flags.

Debugging

As mentioned above, building stage will produce some files that are useful for debugging, so these files are passed to the `gdb` debugger. In order to start debugging you should execute following commands:

```
make run_debug
make debug
```

The first command will pass OS image to the `qemu` virtual machine and it will wait for the debugger to connect to it. The second command will launch `gdb` and it will connect to previously launched virtual machine.

Debugger configurations are stored in file `.gdbinit`. This file defines how debugger should display information, which debug files to load and some other configurations.

References and future work

The development of an operating system requires a deep understanding of various concepts and techniques. This project produced simple operating system, so there are still opportunities for future development and work. Here are some potential areas of focus for future work:

- Virtual memory
- Processes and threads
- Better security and permissions
- File system
- Better shell for user-space

The following references have been really helpful in the creation of this project:

- OSDev Wiki - *(https://wiki.osdev.org/)*
- Writing a simple operating system from scratch by Nick Blundell - *(https://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf)*
- Operating systems: from 0 to 1 - *(https://github.com/tuhdo/os01)*
- The little book about OS development - *(https://littleosbook.github.io/)*
- Writing my own operating system - *(https://dev.to/frosnerd/series/9585)*