OSP - An Operating System Project

Antoine Fraiteur, Matouš Topor

April 2025

1 Introduction

The broad goal of OSP was to build a semi-working kernel. The kernel would be built for the x86 32-bit architecture and include a shell and persistent storage. As you will see, we were not able to complete all of our initial goals for reasons that will become evident later. Let us now briefly introduce the current state of our kernel.

1.1 What is a Kernel

If you are already fluent in bare-metal programming, you can skip this part. However, we find it important to first explain the basic concept.

All programs need to run on some physical hardware. They use RAM to store their variables and code, use CPU cycles to execute them, and may or may not use other parts of the computer like the screen or keyboard. The kernel's job is to facilitate this hardware-software interaction while ensuring secure encapsulation of separate programs to protect user data.

The kernel has complete ownership of all the memory available on the system and divides it between programs. It can decide what programs will run and when and provides an accessible way of communicating with peripherals.

How the kernel does this is highly dependent on the CPU architecture the kernel runs on. However, the user program should be more or less unchanged, only compiled into different instructions (which is the job of the compiler), and may have different functions available provided by the kernel. Basically, the programmer should not really care what hardware it runs on, as it communicates with the kernel, not the CPU itself.

1.2 Our Kernel

x86 is an interrupt-driven architecture with support for paging for process separation. These terms will be explained in their proper chapters as they are quite complex. To emulate it, we decided to use Bochs, a lightweight virtual machine. It loads our ISO into memory, which contains GRUB - the GNU bootloader, which will load our kernel and other specified programs into memory and pass control to the kernel.

For ease of use, we made a Makefile that compiles our ISO and starts Bochs with all the settings we need.

2 Interrupts

An interrupt is a way of handling things like errors without the process knowing that anything happened. When an interrupt is called, the CPU jumps to a previously specified instruction. This function saves the CPU registers, handles the error (or the thing it is supposed to handle), restores the registers, and jumps back to the previously executing code.

Because the CPU registers were saved by the interrupt, the process does not know that anything has happened.

There are more or less three types of interrupts. Exceptions are generated by the CPU when a program executes an invalid instruction, divides by zero, tries to access memory it does not own, or many other things.

Next are hardware interrupts. When you press a key on a PS/2 keyboard, it sends a signal through a PIC controller, which raises an interrupt. The kernel can now communicate with the PIC about which key was pressed and handle it. The same goes for the system clock and other devices.

Lastly, there are system calls. Usually, a user process can't access parts of memory that handle things like communication with the display. To do so, it calls an interrupt, which gives the command to the kernel that can now handle the request. Also, this is important for returning from a user process back to the kernel, as the process does not know which instruction to jump to and usually does not have the privilege anyway. This is the only system call we have implemented.

To use system calls, you first have to set up an Interrupt Descriptor Table. This table maps an interrupt vector to the function that handles it and specifies some information about it, like whether a user can call it or not.

3 Paging

The main job of a kernel is to divide memory between processes. We are using paging for it, but other methods like segmentation can also be used in x86.

The basic concept of paging is to map the virtual memory a process uses to some physical memory. Now every program can think it starts at 0x0 and ends at the 3GB limit, as is common in 32-bit programs, while actually being on completely different physical memory.

To achieve this, we need to create two structures known as the page directory and page table. The page directory is a list of 1024 structs which contain the address of one page table and some information about it, like whether it is writable. The page table contains 1024 similar structures containing the physical memory and some information about it.

When translating virtual memory, the CPU looks at the first 10 bits of the address and finds the corresponding entry in the page directory. In the resulting page table, it finds the corresponding entry with the next 10 bits of the address. Finally, the final address is composed of the 20 bits specified in the page table and the lowest 12 bits in the virtual address.

To enable paging, we need to load the physical address of the page directory into its corresponding register and enable the paging bit in the control register cr0. We decided to identity map the kernel to the address 0xC0000000 - the 3GB mark. Since the kernel is loaded at address 0x0, all the kernel's addresses can be easily calculated.

Since the kernel is linked to begin at 0xC0000000 but is loaded at 0x0, we first need to create the paging, enable it, and jump to the correct code before doing any relative jumps as that would break the system.

3.1 Page Fault

If a process tries to access memory marked as not present in the page table, it triggers the page fault interrupt. This interrupt simply creates a new entry in the page table with an address that has not yet been given to another process.

There is currently no way to mark the address given to another process as unused once the process is over. This would need to be implemented in a real kernel.

4 Processes

The only way we are currently able to load external programs to the system is through GRUB modules. Here our bootloader loads our modules to memory and passes to the kernel a pointer to a multiboot structure, which contains information about how the system was loaded. It includes a linked list of addresses of loaded programs and their sizes.

When executing a process, we first create its page directory and two page tables: one for the stack and one for the program code. The program is then copied from its initial location to the second table at virtual address 0x0. In the process's directory, we also need to add the kernel page tables as they are important for interrupts. The kernel is therefore left at address 0xC0000000 and all higher addresses are reserved for it. The stack table is placed one table lower so it can grow downward.

This stack growth will, however, create a problem since, to execute an interrupt, the CPU needs to push some values on the stack. The stack bottom is, however, outside of available memory, and so this will create another page fault. When unresolved, this will cause a system reboot. To solve this, we would need to implement a TSS (Task State Segment), which would allow us to store the kernel's stack and push the necessary values on it instead of on the process's stack.

A TSS is also needed to switch from ring 3 to ring 0 (change privilege levels). Unfortunately, we did not have time to implement this, so all programs run in ring 0.

When a process is called, the kernel saves important registers to a structure of known location, loads the process's page directory, switches the stack to the top of the stack table, pushes the arguments onto it, and finally jumps to 0x0 to start the process.

Currently, the program needs to be compiled into a flat binary as our kernel can't read files like ELF. If we implemented ELF support, we would need to jump to another address specified in the ELF header.

As long as it starts at 0x0, the program can do whatever it wants. When it is finished, it pushes its return value to the stack and calls interrupt 0x30 - the system call. The return value could be a pointer to a structure, but in our example, it is the result of a simple addition. The system call then saves the return value and restores the registers to the values saved before, returning to kernel code.

Since we did not implement any system calls except the return-to-kernel call, the process can't really do much. But implementing them would be as easy as passing a pointer to a struct containing an ID of what syscall we want and the arguments instead of the return value. Inside the system call function, there would be a switch statement deciding what to do with it. This could be wrapped inside a standard library that the program could be compiled with.

5 IO

5.1 Output

There are two ways our kernel can output something to us, excluding the debug functions provided by Bochs. These are the frame buffer and a serial port.

The frame buffer is basically the screen. It is represented by an array of physical memory in which each pair of bytes represents one character on screen and its color. Writing something on screen is then as easy as changing this array. Another function the frame buffer has is a blinking cursor. Here we use an outb function to write to the output register instructions to the frame buffer about the location of the cursor, and the frame buffer then displays it.

The serial port works differently. We communicate with it exclusively using the outb instruction and receive information from it (like whether or not the next character can be passed to outb) using the inb instruction, which reads the value from the input register. We first need to set up Bochs to write the COM1 port to a file and then configure our kernel to write to it. Using this, we have implemented a logging function, which is our primary helper in debugging.

5.2 Input

The only implemented input of our system is the PIC keyboard. The PIC (Programmable Interrupt Controller) sends an interrupt to the kernel when a key is pressed, and the kernel can then communicate using the outb and inb functions to get information about which key was pressed and make this information available to the system. However, initially, these interrupts are assigned to vectors 0x8 to 0xE, as there are only 8 Intel exceptions in real mode, unlike in 32-bit protected mode, where there are 32 reserved interrupt vectors. We need to first remap it to something else.

Whenever a PIC interrupt is called, another will not be called until the kernel has sent a message to the PIC telling it that it was handled. It is important to implement this since the PIC sends us a system clock interrupt every millisecond or so, and we need to acknowledge them to prevent the system from breaking.

6 Shell

In most systems, the shell is a user process and not provided by the kernel. However, we wanted to implement it as it is a good way to see that the kernel actually does anything without needing to implement many system calls.

Our shell can execute predefined functions and programs loaded as GRUB modules. Our problem is that since, as mentioned above, we can't read ELF files, we do not have any way to identify what the programs loaded into memory actually are. There is no string attached to them giving their name. To get around this, we just hardcoded the names into our kernel. A better implementation would need to be made if the kernel were to be actually used. The shell can also accept arguments, but since our program can't read strings without implementing a standard class (which we did not do), we manually translate arguments into two numbers and pass them as integers. In a real-world scenario, we would copy the string to the top of the process's stack and leave the translation to it.

7 Interesting Parts

Our kernel also features a basic standard library containing wrappers for the frame buffer and serial port, keyboard handling, a malloc and free function, and some more handy tools like a string class. Since we can't use any system calls provided by normal systems, we decided not to use any external libraries in our code. Everything in our kernel was written by us or at least copied by us from somewhere else.

This creates interesting problems. For example, our malloc function is hardly very efficient and has problems with growing. It actually implements a bitmap, splitting the heap into predefined chunks and linearly searching it for available space. This is the same system we used for page frame allocation, although it was heavily modified.

8 Missing Features

Overall, our kernel is still far from fully functional. The most obvious limitation is the lack of any persistent file system support, which was one of our original goals. To implement this, we would first need to create a driver for a physical device. The easiest to use would probably be a floppy drive, and on top of it, we would need to implement a file system to serve as an abstraction layer for user programs.

Another major limitation is that every process currently runs in ring 0. This would likely be relatively easy to address, as it mostly requires implementing a Task State Segment (TSS) and a proper mechanism to switch to user mode.

A further unimplemented feature is ELF file parsing. An ELF file includes a header that contains metadata about how the program should be loaded into memory and, most importantly, the name of the program. With this, the kernel could associate program names with shell commands and execute them accordingly. If we had a file system, we could use that instead to store command names. Still, an ELF parser would greatly simplify program execution.

Finally, there are many interrupts that remain unhandled. If a program crashes, it can halt the entire system since exceptions are not implemented, with the exception of the page fault handler. Additionally, many interrupt sources from the PIC are unimplemented. For example, the system clock is unused, though it could be useful for preemptive scheduling or timekeeping. Even the keyboard driver is incomplete: many keys are not registered, the Shift key does not work, key repeats are not supported, and standard terminal behaviors like the Delete key are unavailable.

There is also only one system call currently implemented, so user processes can't meaningfully interact with the system. Only the kernel has access to the frame buffer and other hardware components.

9 Summary

There are many features that remain unimplemented in our kernel. However, there are also many things we did manage to implement—some of which we didn't even consider at the beginning of this project. Either we didn't know about them or didn't expect we would need them.

Without a doubt, the most difficult part was implementing paging. Paging is a complex topic, and getting it to work took months and caused many headaches. When we started, we had no idea how memory isolation worked, and as a result, it was basically impossible to estimate how long this part would take.

There will always be more to implement in a kernel—just look at Linux, which is still being developed and improved today. This is especially challenging for us because we are writing our own standard library. If we wanted to make full use of it, we would effectively have to write it twice: once for the kernel and once for user programs.

That said, this project succeeded in achieving its main goal: we learned a lot about how kernels work. This knowledge is not only essential for OS development, but also valuable for understanding how computers work under the hood. At the beginning of the year, we didn't even know what a pointer was.