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 build for the x86 32-bit architecture and include a shell and a persistent storage. As you will see we were not able to compleate all of our initial goals for reasons that will become evident later. Let us now breifely 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.

As you know all programs need to run on some physical hardvare. They use RAM to store there variables and code, use CPU cycles to execute it and may or may not use other parts of our comuter like the screen or a keyboard. The kernels job is to facilitate this hardware-software interaction while ensuring secure encapsulation of separate programs to protect user data.

The kernel has compleate ownership of all the mamory availible on the system and divides it between programs, it can decie what programs will run and when, and provides a accesible way of communication with periferials.

The way the kernel does it is highly dependent on the CPU architecture the kernel runs on, however the user program should be more or less unchanged, only being compiled in different instruction, which is the job of the compiler, and maby have different function availible by the kernel. Basicaly the programmer should not really care what hardware it runs on as it communtcates with the kernel, not the CPU itself.

1.2 Our kernel

x86 is an interrupt driven architecture with support of paging for process separation. Thess term will be explain it there proper chapter as they are quite compex. To emulate it we decided to use Bochs a light weight virtual machine. It load our iso into memory, wich contains GRUB - the GNU bootloader wich will load our kernel and other programs we specified into memory and pass the command 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 changing of handling things like errors without the process knowing that anything happend. When an interrupt is called the CPU jumps to a a before specified instuction. This function saves the registers of the CPU, handle the error (or the thing it is supposed to handle), restore the registers and jump back to the code executed before.

Because the CPU registers were svaed by the interrupt the process does not really know that anything has happend.

There are more or less 3 types of interrupts. Exception are generated by the CPU when a program executes an invalid instruction, divides by zero, tries to acess memory it does no own or many other thing.

Next are hardware interrupts, when you press a key on a PS2 keyboard in send a signal trought a PIC controller, wich raises an interrupt. The kernel can now communicate with the PIC on wich key was pressed and handle it. The same goes for the system clock and other devices.

Lastly ther are system calls. Usualy a user process cant acces parts of memory that handele things like the communication with the display. To do so it calls an interrupt wich gives the command to the kernel that can now handle the rquest it has made. Also this is important for returning from a user process back to the kernel as the process does not know wich instruction to jump to and usualy does not have the privilidge anyway. This is the only system call we have implementd.

To use systemcalls, first yuou have to set up an interrupt desctiption table. This table maps an interript vector to the function that handels it and says some information about it, like if a user can call it or not.

3 Paging

The main job of a kerenl is to devide memory between processes. We are using paging for it but other methods like segmantation can be used in x86.

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

To achive this we need to create two structures known as the page directory and page table. The page directory is a list of 1024 structs wich contain the adress of one page table and some information obout it, lie whether it is writable or not. Next the page table contains 1024 simmilar structures containing the physical memory and some information about it.

When translating a virtual memory the CPU looks at the first 10 bits of the adress 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 adress finally the final adress is composed of the 20 bits specified in the page table and the lowest 12 bits in the virtual adress.

TO enable paging we need to load the physical adress of the page directory to itst corresponding register and enable the paging bit in the control register cr0. We decided to identity map the kernel to the adress 0xc000000 - the 3GB mark. Since the kernel is loaded at the adress 0x0 all the kerenls adress can be easily calculated.

Since the kernel is linked to begin at 0xc000000 but is loaded at 0x0 we first need to create the paging, enable it and jump to the code before doing any relative jumps as that would break the system.

3.1 Page Fault

If a process tries to acess a memory marked as not present in the page table it triggers tha page fault interrupt. This interrupt simply crated a new entry in the page tabe with a adress that has not yet been given to another process.

There is currently not any way to mark the adress given to another process as not used once the process is over, this is a thing that would need to be implemented on a real kernel.

4 Processes

The only way we are curently able to load external programs to the system is troughth GRUB modules. Here our bootloader load

5 IO

l

6 Interesting Parts

7 Summary