Operating systems and their parts

Qunix

01.01.1970

Contents

Part I
Intro

Operating systems are like fractals. The deeper you go, the more you are forced to know. Anyway, there's nothing more pleasant than reaching the real end and realizing you are at the very bottom of the unimaginable abyss.....

However, we follow KISS! It means no Windows, no exotic, no challenges. Just make things work and go ahead.

Part II
A Little OS

When we try to do something, we always hope that it'll be simple to achieve the result. But for some awful reasons authors of most guides make monstrous algorithms that are ultimately completely unusable.
To escape these problems, you should remember one thing - `OS is a program`. And the program needs the proper environment to be executed in. No environment - no execution!
Please, do not touch your real computer for now. Our first task is to simulate the execution, not to play with disks. It has too much theory behind. For now. We will return back to this ASAP.
Starting with sandbox - guested isolated RE - we choose QEMU. A bit tricky to play with it freely, but fits excellent to copy & paste or change certain parameters.

0 Adventure begins

The command `man qemu` tells us:

```
qemu-system-x86_64 [options] [disk_image]
```

OK, we need disk image to be started. And here comes the great and terrible truth - the main feature of OS, distinguishing them from other programs, is their ability to be booted. BOOOOOOOtable image. It is achieved relatively easily - your program executable should end up with magic number 0x55aa and have 512 bytes in size.
At first, we make OS, which only hangs. Then, change register values and observe it with debugger. After this we discover the interrupts.
As you could notice, we work with x86-64 architecture. It means reversed byte order so don't be surprised.

1 Assembly

We use nasm. `$$` - the beginning of our code, `$` - the address of current line.

```
jmp $                           ;  infinite loop
times 512 - 2 - ($-$$) db 0     ;  zeroing everything between
dw 0xaa55                       ;  w - word = 2 bytes
```

Let's build it up to executable.
`nasm <input file> [-f <format>] [-o <output file>]`
You can write just `nasm myOS.asm` and you'll here. `hexdump myOS` says:

```
0000000 feeb 0000 0000 0000 0000 0000 0000 0000
0000010 0000 0000 0000 0000 0000 0000 0000 0000
*
00001f0 0000 0000 0000 0000 0000 0000 0000 aa55
0000200
```

Little-endian, as always, makes shit. You can see magic number, the total executable size ($0x200 = 2 * 16^2 = 512$), and also EB FE. According to Intel x86 opcodes, EB is a short (within a byte) jump and its argument, FE, is -2. Jump two bytes back is to start over. Infinite loop as it is.

The next step is to clutter up the registers. Just put up cluttering instruction (e.g. `mov eax, 555`) BEFORE loop, and you're done. `info registers` on qemu monitor should give you what you want to see. Another option is to use GDB. Add option `-s` to QEMU:
`-s Shorthand for -gdb tcp::1234, i.e.  open a gdbserver on TCP port 1234`
And when QEMU starts, start gdb and type
`target remote localhost:1234`
Now debugger and your OS are connected, so you can, for example, clutter up registers in runtime. Use `set $eax=0xdead` and `print /x $eax`.

2 Acquire hardware

Guess you've played enough with registers. Doesn't make any sense, really? Well, to do and see real things we need to know some more. BIOS interrupts allow us to do some basic things, but they are too non-visual while relatively hard to tell about. Skipping this step, we come to modifying peripherals - maybe video memory device. Don't you like colourful text?

3 Getting an upgrade

Part III
The Parts

4 Filesystems

This history is prosaic and too applicative. FS is not OS, and therefore there is a simple mechanism of adding new FS in Linux kernel. Read code, write code, execute code. No "AAAAAAAAAARGH MY HARDDISK HAS BEEN UNEXPECTEDLY REFORMATTED NOOOOO". Don't write OS, write FS.

How to write FS? We interpret it as "how to register new fs in linux kernel and not to cry (as possible)". There are two chairs, and we use dynamic linking because most likely your laptop is a trash already and we don't want to clutter it up any further. So, the first part of question is "how to write a kernel module".
To write a kernel module, we just need a simple Makefile, which specifies our sources and then transfer control to a some of kernel Makefiles written exactly for making modules. Google this step yourself.

As for the sources, there are also some rules of writing FS. You need to specify a chain of functions doing at least mounting and unmounting. Then, you can check kernel log and see if it works. The working (for me) example is in my repo.