Exercise 1: Syscall conventions

An interesting question we left unanswered in the introduction is, how does your program jump over to the kernel and get it to execute your syscalls? In the diagram from the intro pdf, there's an arrow in the interface between the kernel and user program that says "INT 0x80". This is an assembly instruction that exists in most CPU's (including x86, which is what you're likely using right now) that sends a "software interrupt". We'll talk about interrupts, but suffice it to say that these are events, usually sent by hardware devices, but in this case also software, to signal the CPU of an important operation that needs to be handled. This includes things like a network card telling the CPU that it got a new packet, or a keyboard telling the CPU that it got a keystroke. These are handled and acted on by the kernel, and the CPU will execute them in a privileged mode.

Now, notice that I only mentioned the instruction "INT 0x80", which is called regardless of the syscall used or the parameters. How does the kernel know which one was called? Let's use as an example the `write()` call. This is used to write to files but, because everything in Linux is a file, you can also write to standard output (aka your terminal) using the write syscall.

For this, we'll use a sample hello world program written in assembly. For this to work, we'll briefly go over the binary build process and what is assembly.

What are binaries?

Most of the executable files on your system are binary files. These binary files contain and encoded set of instructions for your CPU. Every CPU architecture has a manual containing a full list of every instruction you can use on their CPU's. The vast majority of modern servers and personal computers (including laptops) use either x86, or one of the 64-bit equivalents, like IA-64, or x86-64 (both of which are, for most day-to-day uses, compatible). Intel provides its manual here. It's a massive document so I advise against reading the whole thing, but keep it bookmarked in case we reference it later.

When you write a program in a compiled language like C, you run the code through a compiler (like gcc, which is what we'll be using here). This program turns your program into this binary-formatted machine code. Compilation is a complicated procedure, as the C program and the compiled result look nothing alike. However, machine code is extremely difficult and tedious to write, so people generally don't write those binary files themselves.

That all said, there are times when you might want to write something like machine code to get specific behaviours that a compiler would not generate. This is where assembly comes in. Assembly is a language that lets you write readable machine code. Basically, it's a text file containing names of instructions and the arguments, which an assembler can turn into machine code very easily.

To understand what's going on in a machine, you can think of a CPU as a complex machine with main memory, registers, outside devices, and the computation core. Main memory is your RAM, and it's not the fastest to access, but can hold a lot of data, and is easy to fetch from the CPU by referencing it as a memory address (or a number representing how many bytes off of the begining of memory something is). Registers are the data you can actively work on from the CPU. It's a set of small chunks of memory (usually each one can hold 4 or 8 bytes of memory, depending on whether you're using a 32-bit or 64-bit architecture) which are quick to access and modify. These are all named, and can usually be used to store anything, though there's conventions around what they should be used for. On a standard x86_64 architecture, these general purpose registers are:

```
%rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp, and %r8-r15
```

There's two of these, `%rsp` and `%rbp`, that have a special purpose related to keeping track of the stack memory. We'll come back to those later in an aside.

There's a few different syntaxes for assembly that are used in the wild, but for this we'll be using *AT&T syntax*. The main thing to keep in mind here is that instructions with two parameters are written with the source first and destination second. For example, the instruction:

```
movl %eax, %ebx
```

Will copy a long (4 bytes) from the register `%eax` to the register `%ebx`.

Lauching our example

Now, let's actually take a look at the sample code for this exercise. Run this from wherever you're launching docker:

```
docker run -it pementorship/w2_ex1
```

You should see the following files:

```
[me@2e219c7525b1 ~]$ ls
Makefile  write_test.s
```

`write_test.s` is an assembly file which we're gonna compile and run. Run the following and let's see what happens:

```
[me@083afbb3b899 ~]$ make
gcc write_test.s -o write_test
[me@083afbb3b899 ~]$ ./write_test
Hello, world!
```

So it's a hello world program! Notice we used `make` to compile the program, which picked up the configuration from the `Makefile` file on that directory, and ran the command you can see printed out bellow make.

Let's take a look at how this hello world is implemented. Open `write_test.s` with `less` or your editor of choice. You should see this:

```
.global main

.text
main:
        # write(1, message, 14)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file descriptor 1 is stdout
        mov     $message, %rsi          # address of string to output
        mov     $14, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax               # system call 60 is exit
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world!\n"
```

Source: https://cs.lmu.edu/~ray/

This is a lot to take in, but the important thing to note is everything “inside” the `main:`. This is the function we’re calling. There’s two things happening here, first we’re calling the `write()` syscall, to print out our message, and then we’re caling the `exit()` syscall, to finish the program with a certain exit code. Let’s focus on the `write()`:

The `write()` syscall

If you run and read `man 2 write`, you can find the signature of the syscall:

```
ssize_t write(int fd, const void *buf, size_t count);
```

(the 2 in `man 2 write` tells us to look at section 2 of the manual, where you’ll find all the syscalls. If you don’t specify a section, it’ll look for the closest match, which for system calls will usually result in the wrong section)

Notice it has 3 arguments, `fd`, or the file descriptor we’re writing to, `buf`, the pointer to the data to write out, and `count`, the number of bytes to write. We have to put these in this order in their expected location, and then we can call the syscall with the `syscall` instruction (which is the x86_64 version of `int 0x80`).

How do we choose the ordering of the arguments? Simple. There’s a man page that tells you how your specific system handles argument ordering. If you do `man syscall`, you can read the “Architecture calling conventions” section, which contains the following convensions for x86-64:

```
arch/ABI    instruction           syscall #  retval  error    Notes
--------------------------------------------------------------------
[...]
x86-64      syscall               rax        rax     -        [5]
[...]


arch/ABI      arg1  arg2  arg3  arg4  arg5  arg6  arg7  Notes
--------------------------------------------------------------
[...]
x86-64        rdi   rsi   rdx   r10   r8    r9    -
[...]
```

As you can see, we store the syscall # in %rax (here, all we care is `write()` has number 1), and the three arguments in %rdi, %rsi and %rdx respectively. Let’s see how we do that:

```
mov     $1, %rax                # system call 1 is write
mov     $1, %rdi                # file descriptor 1 is stdout
mov     $message, %rsi          # address of string to output
mov     $14, %rdx               # number of bytes
syscall                         # invoke operating system to do the write
```

So first, we do `mov $1, %rax`, which puts the literal value 1 into `%rax`, representing the syscall number for write.

Then we do `mov $1, %rdi`, which passes in a 1 for the `fd` paramenter. Why 1? Well, usually, this number comes from the output of `open()`, which opens files and creates new file descriptors for them, which it then returns. However, standard output will always have a fixed file descriptor of 1, so we can just hard-code it.

Next, we do `mov $message, %rsi`, which will put the memory address of $message in `%rsi`. Where does `message` come from? Where it’s a static value that you can see defined at the `message` segment (everything after the `message:`).

Almost done now, we do a `mov $14, %rdx`, which puts a 14 for the final `count` argument, which is just a constant for the size of number of bytes, which we've basically hard-coded.

Finally, we just call `syscall`, which sends the interrupt that eventually lets the OS act on this syscall, and use the parameters we've saved in the registers.