Chapter 7

The Road to User Mode

Now that the kernel boots, prints to screen and reads from keyboard - what do we do? Usually, a kernel is not supposed to do the application logic itself, but leave that for applications. The kernel creates the proper abstractions (for memory, files, devices) to make application development easier, performs tasks on behalf of applications (system calls) and schedules processes.

User mode, in contrast with kernel mode, is the environment in which the user's programs execute. This environment is less privileged than the kernel, and will prevent (badly written) user programs from messing with other programs or the kernel. Badly written kernels are free to mess up what they want.

There's quite a way to go until the OS created in this book can execute programs in user mode, but this chapter will show how to easily execute a small program in kernel mode.

Loading an External Program

Where do we get the external program from? Somehow we need to load the code we want to execute into memory. More feature-complete operating systems usually have drivers and file systems that enable them to load the software from a CD-ROM drive, a hard disk or other persistent media.

Instead of creating all these drivers and file systems we will use a feature in GRUB called modules to load the program.

GRUB Modules

GRUB can load arbitrary files into memory from the ISO image, and these files are usually referred to as *modules*. To make GRUB load a module, edit the file `iso/boot/grub/menu.lst` and add the following line at the end of the file:

```
module /modules/program
```

Now create the folder `iso/modules`:

```
mkdir -p iso/modules
```

The application `program` will be created later in this chapter.

The code that calls `kmain` must be updated to pass information to `kmain` about where it can find the modules. We also want to tell GRUB that it should align all the modules on page boundaries when loading them (see the chapter "Paging" for details about page alignment).

To instruct GRUB how to load our modules, the "multiboot header" - the first bytes of the kernel - must be updated as follows:

```
; in file ‘loader.s‘


MAGIC_NUMBER    equ 0x1BADB002      ; define the magic number constant
ALIGN_MODULES   equ 0x00000001      ; tell GRUB to align modules

; calculate the checksum (all options + checksum should equal 0)
CHECKSUM        equ -(MAGIC_NUMBER + ALIGN_MODULES)

section .text:                      ; start of the text (code) section
align 4                             ; the code must be 4 byte aligned
    dd MAGIC_NUMBER                 ; write the magic number
    dd ALIGN_MODULES                ; write the align modules instruction
    dd CHECKSUM                     ; write the checksum
```

GRUB will also store a pointer to a `struct` in the register `ebx` that, among other things, describes at which addresses the modules are loaded. Therefore, you probably want to push `ebx` on the stack before calling `kmain` to make it an argument for `kmain`.

Executing a Program

A Very Simple Program

A program written at this stage can only perform a few actions. Therefore, a very short program that writes a value to a register suffices as a test program. Halting Bochs after a while and then check that register contains the correct number by looking in the Bochs log will verify that the program has run. This is an example of such a short program:

```
; set eax to some distinguishable number, to read from the log afterwards
mov eax, 0xDEADBEEF

; enter infinite loop, nothing more to do
; $ means "beginning of line", ie. the same instruction
jmp $
```

Compiling

Since our kernel cannot parse advanced executable formats we need to compile the code into a flat binary. NASM can do this with the flag `-f`:

```
nasm -f bin program.s -o program
```

This is all we need. You must now move the file `program` to the folder `iso/modules`.

Finding the Program in Memory

Before jumping to the program we must find where it resides in memory. Assuming that the contents of `ebx` is passed as an argument to `kmain`, we can do this entirely from C.

The pointer in `ebx` points to a *multiboot* structure [19]. Download the `multiboot.h` file from http://www.gnu.org/software/grub/manual/multiboot/html_node/multiboot.h.html, which describes the structure.

The pointer passed to `kmain` in the `ebx` register can be cast to a `multiboot_info_t` pointer. The address of the first module is in the field `mods_addr`. The following code shows an example:

```
int kmain(/* additional arguments */ unsigned int ebx)
{
    multiboot_info_t *mbinfo = (multiboot_info_t *) ebx;
    unsigned int address_of_module = mbinfo->mods_addr;
}
```

However, before just blindly following the pointer, you should check that the module got loaded correctly by GRUB. This can be done by checking the `flags` field of the `multiboot_info_t` structure. You should also check the field `mods_count` to make sure it is exactly 1. For more details about the multiboot structure, see the multiboot documentation [19].

Jumping to the Code

The only thing left to do is to jump to the code loaded by GRUB. Since it is easier to parse the multiboot structure in C than assembly code, calling the code from C is more convenient (it can of course be done with `jmp` or `call` in assembly code as well). The C code could look like this:

```
typedef void (*call_module_t)(void);
/* ... */
call_module_t start_program = (call_module_t) address_of_module;
start_program();
/* we'll never get here, unless the module code returns */
```

If we start the kernel, wait until it has run and entered the infinite loop in the program, and then halt Bochs, we should see `0xDEADBEEF` in the register `eax` via the Bochs log. We have successfully started a program in our OS!

The Beginning of User Mode

The program we've written now runs at the same privilege level as the kernel - we've just entered it in a somewhat peculiar way. To enable applications to execute at a different privilege level we'll need to, beside *segmentation*, do *paging* and *page frame allocation*.

It's quite a lot of work and technical details to go through, but in a few chapters you'll have working user mode programs.