Chapter 11

User Mode

User mode is now almost within our reach, there are just a few more steps required to get there. Although these steps might seem easy they way they are presented in this chapter, they can be tricky to implement, since there are a lot of places where small errors will cause bugs that are hard to find.

Segments for User Mode

To enable user mode we need to add two more segments to the GDT. They are very similar to the kernel segments we added when we set up the GDT in the chapter about segmentation:

Index	Offset	Name	Address range	Type	DPL
3	`0x18`	user code segment	`0x00000000 - 0xFFFFFFFF`	RX	PL3
4	`0x20`	user data segment	`0x00000000 - 0xFFFFFFFF`	RW	PL3

Table 11.1: The segment descriptors needed for user mode.

The difference is the DPL, which now allows code to execute in PL3. The segments can still be used to address the entire address space, just using these segments for user mode code will not protect the kernel. For that we need paging.

Setting Up For User Mode

There are a few things every user mode process needs:

- Page frames for code, data and stack. At the moment it suffices to allocate one page frame for the stack and enough page frames to fit the program's code. Don't worry about setting up a stack that can be grow and shrink at this point in time, focus on getting a basic implementation work first.
- The binary from the GRUB module has to be copied to the page frames used for the programs code.
- A page directory and page tables are needed to map the page frames described above into memory. At least two page tables are needed, because the code and data should be mapped in at `0x00000000`

and increasing, and the stack should start just below the kernel, at `0xBFFFFFFB`, growing towards lower addresses. The U/S flag has to be set to allow PL3 access.

It might be convenient to store this information in a `struct` representing a process. This process `struct` can be dynamically allocated with the kernel's `malloc` function.

Entering User Mode

The only way to execute code with a lower privilege level than the current privilege level (CPL) is to execute an `iret` or `lret` instruction - interrupt return or long return, respectively.

To enter user mode we set up the stack as if the processor had raised an inter-privilege level interrupt. The stack should look like the following:

```
[esp + 16]  ss      ; the stack segment selector we want for user mode
[esp + 12]  esp     ; the user mode stack pointer
[esp +  8]  eflags  ; the control flags we want to use in user mode
[esp +  4]  cs      ; the code segment selector
[esp +  0]  eip     ; the instruction pointer of user mode code to execute
```

See the Intel manual [33], section 6.2.1, figure 6-4 for more information.

The instruction `iret` will then read these values from the stack and fill in the corresponding registers. Before we execute `iret` we need to change to the page directory we setup for the user mode process. It is important to remember that to continue executing kernel code after we've switched PDT, the kernel needs to be mapped in. One way to accomplish this is to have a separate PDT for the kernel, which maps all data at `0xC0000000` and above, and merge it with the user PDT (which only maps below `0xC0000000`) when performing the switch. Remember that physical address of the PDT has to be used when setting the register `cr3`.

The register `eflags` contains a set of different flags, specified in section 2.3 of the Intel manual [33]. Most important for us is the interrupt enable (IF) flag. The assembly code instruction `sti` can't be used in privilege level 3 for enabling interrupts. If interrupts are disabled when entering user mode, then interrupts can't enabled once user mode is entered. Setting the IF flag in the `eflags` entry on the stack will enable interrupts in user mode, since the assembly code instruction `iret` will set the register `eflags` to the corresponding value on the stack.

For now, we should have interrupts disabled, as it requires a little more work to get inter-privilege level interrupts to work properly (see the section "System calls").

The value `eip` on the stack should point to the entry point for the user code - `0x00000000` in our case. The value `esp` on the stack should be where the stack starts - `0xBFFFFFFB` (`0xC0000000 - 4`).

The values `cs` and `ss` on the stack should be the segment selectors for the user code and user data segments, respectively. As we saw in the segmentation chapter, the lowest two bits of a segment selector is the RPL - the Requested Privilege Level. When using `iret` to enter PL3, the RPL of `cs` and `ss` should be `0x3`. The following code shows an example:

```
USER_MODE_CODE_SEGMENT_SELECTOR equ 0x18
USER_MODE_DATA_SEGMENT_SELECTOR equ 0x20
mov cs, USER_MODE_CODE_SEGMENT_SELECTOR | 0x3
mov ss, USER_MODE_DATA_SEGMENT_SELECTOR | 0x3
```

The register `ds`, and the other data segment registers, should be set to the same segment selector as `ss`. They can be set the ordinary way, with the `mov` assembly code instruction.

We are now ready to execute `iret`. If everything has been set up right, we should now have a kernel that can enter user mode.

Using C for User Mode Programs

When C is used as the programming language for user mode programs, it is important to think about the structure of the file that will be the result of the compilation.

The reason we can use ELF [18] as the file format for for the kernel executable is because GRUB knows how to parse and interpret the ELF file format. If we implemented an ELF parser, we could compile the user mode programs into ELF binaries as well. We leave this as an exercise for the reader.

One thing we can do to make it easier to develop user mode programs is to allow the programs to be written in C, but compile them to flat binaries instead of ELF binaries. In C the layout of the generated code is more unpredictable and the entry point, `main`, might not be at offset 0 in the binary. One common way to work around this is to add a few assembly code lines placed at offset 0 which calls `main`:

```
extern main

section .text
    ; push argv
    ; push argc
    call main
    ; main has returned, eax is return value
    jmp  $    ; loop forever
```

If this code is saved in a file called `start.s`, then the following code show an example of a linker script that places these instructions first in executable (remember that `start.s` gets compiled to `start.o`):

```
OUTPUT_FORMAT("binary")    /* output flat binary */

SECTIONS
{
    . = 0;                 /* relocate to address 0 */

    .text ALIGN(4):
    {
        start.o(.text)     /* include the .text section of start.o */
        *(.text)           /* include all other .text sections */
    }

    .data ALIGN(4):
    {
        *(.data)
    }

    .rodata ALIGN(4):
    {
```

```
        *(.rodata*)
    }
}
```

Note: `*(.text)` will not include the `.text` section of `start.o` again.

With this script we can write programs in C or assembler (or any other language that compiles to object files linkable with `ld`), and it is easy to load and map for the kernel (`.rodata` will be mapped in as writeable, though).

When we compile user programs we want the following GCC flags:

```
-m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles
-nodefaultlibs
```

For linking, the followings flags should be used:

```
-T link.ld -melf_i386  # emulate 32 bits ELF, the binary output is specified
                       # in the linker script
```

The option `-T` instructs the linker to use the linker script `link.ld`.

A C Library

It might now be interesting to start thinking about writing a small "standard library" for your programs. Some of the functionality requires system calls to work, but some, such as the functions in `string.h`, does not.

Further Reading

- Gustavo Duarte has an article on privilege levels: http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection