Chapter 3

Getting to C

This chapter will show you how to use C instead of assembly code as the programming language for the OS. Assembly is very good for interacting with the CPU and enables maximum control over every aspect of the code. However, at least for the authors, C is a much more convenient language to use. Therefore, we would like to use C as much as possible and use assembly code only where it make sense.

Setting Up a Stack

One prerequisite for using C is a stack, since all non-trivial C programs use a stack. Setting up a stack is not harder than to make the `esp` register point to the end of an area of free memory (remember that the stack grows towards lower addresses on the x86) that is correctly aligned (alignment on 4 bytes is recommended from a performance perspective).

We could point `esp` to a random area in memory since, so far, the only thing in the memory is GRUB, BIOS, the OS kernel and some memory-mapped I/O. This is not a good idea - we don't know how much memory is available or if the area `esp` would point to is used by something else. A better idea is to reserve a piece of uninitialized memory in the `bss` section in the ELF file of the kernel. It is better to use the `bss` section instead of the `data` section to reduce the size of the OS executable. Since GRUB understands ELF, GRUB will allocate any memory reserved in the `bss` section when loading the OS.

The NASM pseudo-instruction `resb` [24] can be used to declare uninitialized data:

```
KERNEL_STACK_SIZE equ 4096                  ; size of stack in bytes

section .bss
align 4                                     ; align at 4 bytes
kernel_stack:                               ; label points to beginning of memory
    resb KERNEL_STACK_SIZE                  ; reserve stack for the kernel
```

There is no need to worry about the use of uninitialized memory for the stack, since it is not possible to read a stack location that has not been written (without manual pointer fiddling). A (correct) program can not pop an element from the stack without having pushed an element onto the stack first. Therefore, the memory locations of the stack will always be written to before they are being read.

The stack pointer is then set up by pointing `esp` to the end of the `kernel_stack` memory:

```
mov esp, kernel_stack + KERNEL_STACK_SIZE   ; point esp to the start of the
                                            ; stack (end of memory area)
```

Calling C Code From Assembly

The next step is to call a C function from assembly code. There are many different conventions for how to call C code from assembly code [25]. This book uses the *cdecl* calling convention, since that is the one used by GCC. The cdecl calling convention states that arguments to a function should be passed via the stack (on x86). The arguments of the function should be pushed on the stack in a right-to-left order, that is, you push the rightmost argument first. The return value of the function is placed in the `eax` register. The following code shows an example:

```
/* The C function */
int sum_of_three(int arg1, int arg2, int arg3)
{
    return arg1 + arg2 + arg3;
}

; The assembly code
external sum_of_three   ; the function sum_of_three is defined elsewhere

push dword 3            ; arg3
push dword 2            ; arg2
push dword 1            ; arg1
call sum_of_three       ; call the function, the result will be in eax
```

Packing Structs

In the rest of this book, you will often come across "configuration bytes" that are a collection of bits in a very specific order. Below follows an example with 32 bits:

```
Bit:     | 31     24 | 23          8 | 7     0 |
Content: | index     | address       | config  |
```

Instead of using an unsigned integer, `unsigned int`, for handling such configurations, it is much more convenient to use "packed structures":

```
struct example {
    unsigned char config;   /* bit 0 - 7   */
    unsigned short address; /* bit 8 - 23  */
    unsigned char index;    /* bit 24 - 31 */
};
```

When using the `struct` in the previous example there is no guarantee that the size of the `struct` will be exactly 32 bits - the compiler can add some padding between elements for various reasons, for example to speed up element access or due to requirements set by the hardware and/or compiler. When using a `struct` to represent configuration bytes, it is very important that the compiler does *not* add any padding, because the `struct` will eventually be treated as a 32 bit unsigned integer by the hardware. The attribute `packed` can be used to force GCC to *not* add any padding:

```
struct example {
    unsigned char config;   /* bit 0 - 7   */
```

```
    unsigned short address; /* bit 8 - 23  */
    unsigned char index;    /* bit 24 - 31 */
} __attribute__((packed));
```

Note that `__attribute__((packed))` is not part of the C standard - it might not work with all C compilers.

Compiling C Code

When compiling the C code for the OS, a lot of flags to GCC need to be used. This is because the C code should *not* assume the presence of a standard library, since there is no standard library available for our OS. For more information about the flags, see the GCC manual.

The flags used for compiling the C code are:

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

As always when writing C programs we recommend turning on all warnings and treat warnings as errors:

```
-Wall -Wextra -Werror
```

You can now create a function `kmain` in a file called `kmain.c` that you call from `loader.s`. At this point, `kmain` probably won't need any arguments (but in later chapters it will).

Build Tools

Now is also probably a good time to set up some build tools to make it easier to compile and test-run the OS. We recommend using `make` [13], but there are plenty of other build systems available. A simple Makefile for the OS could look like the following example:

```
OBJECTS = loader.o kmain.o
CC = gcc
CFLAGS = -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector \
         -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c
LDFLAGS = -T link.ld -melf_i386
AS = nasm
ASFLAGS = -f elf

all: kernel.elf

kernel.elf: $(OBJECTS)
    ld $(LDFLAGS) $(OBJECTS) -o kernel.elf

os.iso: kernel.elf
    cp kernel.elf iso/boot/kernel.elf
    genisoimage -R                              \
                -b boot/grub/stage2_eltorito    \
                -no-emul-boot                   \
```

```
                -boot-load-size 4               \
                -A os                           \
                -input-charset utf8             \
                -quiet                          \
                -boot-info-table                \
                -o os.iso                       \
                iso

run: os.iso
    bochs -f bochsrc.txt -q

%.o: %.c
    $(CC) $(CFLAGS)  $< -o $@

%.o: %.s
    $(AS) $(ASFLAGS) $< -o $@

clean:
    rm -rf *.o kernel.elf os.iso
```

The contents of your working directory should now look like the following figure:

```
.
|-- bochsrc.txt
|-- iso
|   |-- boot
|     |-- grub
|       |-- menu.lst
|       |-- stage2_eltorito
|-- kmain.c
|-- loader.s
|-- Makefile
```

You should now be able to start the OS with the simple command `make run`, which will compile the kernel and boot it up in Bochs (as defined in the Makefile above).

Further Reading

- Kernigan & Richie's book, *The C Programming Language, Second Edition*, [8] is great for learning about all the aspects of C.