Chapter 1

A basic kernel

In this chapter, we will show how to build and run the most basic of kernels[1]. In keeping with programming tradition, we will call the kernel *HelloWorld*, although, as the world in which our code operates gets destroyed almost as soon as it starts running, a more appropriate name might have been *GoodbyeWorld*, cruel or not.

All source files corresponding to this chapter can be found atxs `http://www.cs.vu.nl/~herbertb/misc/writingkernels/src/basickernel_helloworld.tgz`.

1.1 The bare basics: running on the (virtual) metal

Our aim is to build a real kernel that you can run on a normal x86 machine. Doing so has two implications. First, *we will not have any support if we do not add it to our kernel ourselves.* Specifically, none of the helpful libraries and services are available to help us write, run, and debug our programs. Even printing on the screen, for instance, is not trivial. The `C printf` function does not exist yet, and neither do any of the library function.

As an aside, while testing and coding, you probably do not want to reboot your machine every single time you have made a small change to your kernel. It is better if we do all our development on a virtual machine. To our kernel, these machines provide an illusion of hardware, but in reality they are simply programs that run on our normal operating system. In this text, we will use Qemu [1] as our virtual machine. Qemu is convenient and runs on Linux, BSD, Mac OS X, and even Windows. It is important to emphasise, however, that the virtual machine is for convenience only. All that we develop works equally well on real hardware.

You may wonder how this works. After all, a physical machine typically boots from a hard drive partition[2]. The answer is deceptively simple. We will make a 'boot image', a file that pretends to be a partition, and that contains the kernel, some code to load and execute that kernel, and every thing else we need. Qemu can use this image as if it is a real partition and boot from it. So, if we copy our kernel on a real partition, we boot from a real drive and if we copy it to our generated boot image for Qemu, we can boot it in Qemu.

This brings us to the second implication: *bootstrapping*. We need a way to tell our machine to pick up our kernel from disk, load it in memory with all bits at the right addresses and execute it. While it is entirely possible to write such a bootloader from scratch, it is tedious and, certainly on x86, it is not needed.

On the x86 platform we can avail ourselves of GNU's Grand Unified Bootloader (GRUB), familiar to most Linux users and even users of Solaris on x86. Grub implements the *multiboot specification*, an open standard originally created in 1995, that describes how compliant (multiboot) kernels can be loaded. What it means exactly, to be multiboot compliant, is not very

[1] This document is inspired by Brandon Friesen's Kernel Development Tutorial [2].

[2] PCs can also boot from floppy disks, CD ROMS, and these days even from USB sticks

important at this stage. We will just have to make sure that we are, so we can load our kernel using Grub.

In this first implementation chapter, we will just try to get something up quickly, without thinking about the hows and whys too much. Later, we will look at certain bits of the process in more detail. In later chapters, we will add more functionality to our trivial HelloWorld kernel. The idea is that after a few iterations, we will arrive at a kernel that is 'useful' in the sense that it supports processes, virtual memory, keyboard input and console output, as well as a rudimentary file system.

1.2 The kernel coder's tools: what do we need?

In this text, we assume that we build our kernel on Linux, or cygwin[3] on Windows, so that we can use a uniform and convenient set Unix tools. Cygwin needs to be installed and setup to include a set of development tools, as described below. For Linux, I am using distributions like Debian or Ubuntu, but there is no reason why you could not use some other distribution. The only real difference is in the installation of software packages. On Debian/Ubuntu we use `apt-get` for this purpose. While Red Hat, Slackware, and other distributions may not support this particular command, they will have equivalent ways of installing software.

The things that you really need for this course are:

1. PC: just your normal working machine will do
2. qemu: a whole system emulator (`www.qemu.org/`)
3. gcc: the GNU C compiler (`gcc.gnu.org/`)
4. make: a utility for automatically building executable programs from source code (www.gnu.org/software/make/)
5. gas: the GNU assembler and other binary utilities (`www.gnu.org/software/binutils/`)
6. grub: the Grand Unified Bootloader (`www.gnu.org/software/grub/`)
7. mtools: open source collection of tools to allow Unix Operating System to manipulate files on an MS-DOS filesystem (`www.gnu.org/software/mtools`)

On Debian/Ubuntu you simply install software that is not yet available by means of `apt-get`. For instance, to install all of the above, programs you may type:

```
sudo apt-get install qemu gcc-4.0 make binutils grub mtools
```

Depending on your current configuration, you may have some of these packages installed already. However, all of these programs are standard development tools that you can just install using whatever your local equivalent of `apt-get` may be (except for the PC, which you obtain with your local equivalent of euros). .

1.3 Preparing a Qemu boot image

At some point during this chapter, we will have a bootable kernel. Let us think about what to do with it when we do. As mentioned, we will use Grub as a bootloader to load and start our kernels. That means that we still need an image with grub installed on it. We also have to make sure that our kernel is on the image. In addition, we will have to add a menu for grub that lists all the kernels it can boot. Grub is quite powerful, you see, and it can boot a machine with any of a set of kernels listed in the Grub menu. The menu is typically called the `menu.lst` file. In our

[3] www.cygwin.com

case, we need to add an entry that points to our kernel. Summarising, we need to create an image file and we need to store Grub, a grub menu, and a kernel on this image file.

The way we create a bootable image is by means of the `dd` tool and the programs that come with `mtools`. The `dd` tool is a common Unix program whose primary purpose is the low-level copying and conversion of raw data. We use `dd` to create an zero-filled image called `core.img` with 088704 blocks of 512 bytes each.

```
dd if=/dev/zero of=core.img count=088704 bs=512
```

Now, we have to transform this raw image into something that carries a file system layout. We use `mtools` for this purpose. First we make sure that all mtools' commands work on our image `core.img`.

```
echo "drive c: file=\"`pwd`/core.img\" partition=1" > ~/.mtoolsrc
```

The tools will automatically pick up the file `.mtoolsrc` from our home directory now. So we no longer have to specify that we want to execute the following commands on this image.

First we create a `c:` partition. The '-I' option means that the parition table will be initialised and all existing partitions, if any, destroyed. In our case, we do not have partitions yet, but we do need a new partition table.

```
mpartition -I c:
```

Next, we create (option `-c`) a new partition, with 88 cylinders (`-t`), 16 heads (`-h`), and 63 sectors per track. Note that $88 * 16 * 63 = 88704$. In other words, it matches the number of sectors we made on our image.

```
mpartition -c -t 88 -h 16 -s 63 c:
```

Next, we have to format our newly created partition. We format it in MS DOS format (FAT16), as this is nice and simple, and good enough for our purpose:

```
mformat c:
```

Given this file system image, we can add directories and copy files. For instance, let us create the following two directories:

```
mmd c:/boot
mmd c:/boot/grub
```

We are getting there, but we are not done yet. We also have to make Grub available on the image. It may well be that you are already using Grub on your real machine. If not, you need to install it somewhere on your hard drive. I will assume that Grub lives in `/boot/grub/` on your real machine, which is where it will typically live if you are using Grub yourself. We need to copy a few files to our new image. The precise use of each of these Grub files is not very important. Suffice to say that Grub needs them all. We will copy them into the directory `c:/boot/grub`

```
mcopy /boot/grub/stage1 c:/boot/grub
mcopy /boot/grub/stage2 c:/boot/grub
mcopy /boot/grub/fat_stage1_5 c:/boot/grub
```

Presently, we need a bit of Grub magic to provide a 'device map'. It is not very important, but basically, the story is as follows. Grub has its own way of naming hard disks and partitions. Not surprisingly, its conventions differ from the Linux device names. Linux uses such names as `/dev/hda1`. The first hard disk on Linux is called `/dev/hd0`, while the floppy drive is called `/dev/fd0`. The four primary partitions allowed per disk are numbered from 0 to 3. Logical partitions are counted beginning with 4. So we end up with something like:

```
(hd0,0)   first primary partition on first hard disk
(hd0,1)   second primary partition
(hd0,2)   third primary partition
(hd0,3)   fourth primary partition (usually an extended partition)
(hd0,4)   first logical partition
(hd0,5)   second logical partition ...
```

All hard disks detected by the BIOS or other disk controllers are simpply counted according to the boot sequence in the BIOS itself. As the BIOS device names do no match up with Linux device names, we need a mapping between the two. Grub stores this mapping in a file called the device map.

In our case, we have called the mapping file `bmap`, and we add an entry in it to say that (hd0) is the `core.img` file. Plus we specify what sort of partition this is and that our boot partition hd0 is the first partition on disk. Finally, with `setup` we write the boot sector on the first disk:

```
echo "(hd0) core.img" > bmap
printf "geometry (hd0) 88 16 63 \n root (hd0,0) \n setup (hd0)\n" | /usr/sbin/grub \
      --device-map=bmap  --batch
```

All we need now is a menu and a working kernel. The latter is the topic of the remainder of this chapter, but the former is easy. Let us assume that when we finally do have a working kernel (HelloWorld or otherwise), we will call it `kernel.bin` and store it in `c:/boot/grub/`. In anticipation of this kernel, we can make a Grub menu that contains an entry for it. To do so, create a file called `menu.lst` with the following content:

```
      serial --unit=0 --stop=1 --speed=115200 --parity=no --word=8
      terminal --timeout=0 serial console

      default 0
      timeout = 0
      title = mykernel
      kernel=/boot/grub/kernel.bin
      #       module=/boot/grub/additional_modules
```

The first two lines specify that we can control our computer using a serial line. We specify the speed of the connection (115200 bps), parity, and a few other things.

The next few lines specify that kernel 0 is called `mkernel` and that it is the default kernel that will be immediately booted and that the kernel that goes with it is the one specified.

We now copy this file to `core.img`

```
mcopy menu.lst c:/boot/grub/
```

And we are done. That is, we have an image eagerly anticipating a kernel. As soon as we have one, we can copy it to the image too:

```
mcopy  kernel.bin c:/boot/grub/
```

Assuming there are no bugs in the kernel it will be booted as soon as we turn on the machine. The equivalent to turning on the machine for Qemu is to start the virtual machine:

```
qemu -hda core.img
```

At this point, of course, this will not do anything yet. We first need the kernel. The remainder of this chapter is devoted to creating a (trivial) kernel that can be booted either in the virtual machine, or on real hardware. It will be equally unimpressive in either case. However, the first HelloWorld in kernel terms is a giant leap forward. Once we have *any* code running on our machine, we can also add *any* code we want.

1.4 A World kernel (without Hellos)

One thing that is easy to minimise but hard to avoid entirely in writing kernels is some assembly language. The very first instructions that are executed are written in assembly (the "entry point") and so is some of the code that deals with interrupts, the code that turns on paging, etc. Assembly language, even with an instruction set as ugly as that of the x86, should not frighten us. Most of it is straightforward and where it is not, we will furnish ample explanation.

We want to use Grub. This implies, as mentioned earlier, that we should make our kernel multiboot compliant. In the old days, every OS came with its own set of boot mechanims. If the boot mechanism that came with your OS was not exactly what you wanted, tough. Making multiple OSs (such as your HelloWorld kernel and Windows or Linux) coexist together peacefully, was a real challenge. The multiboot specification addresses this problem. It specifies an interface

between the boot loader and the OS, so that any bootloader that is multiboot compliant should be able to boot any OS that is also multiboot compliant. Grub certainly is multiboot compliant. How do we make our kernel compliant too?

A minimum requirement to be multiboot compliant is to have a header with the following three 32 bit words somewhere in the first 8 KB of your file:

1. The multboot magic number: 0xbadboo2

2. The multiboot header flags. The flags specify features that the OS requests or requires of the bootloader (e.g., information about available memory that should be provided by grub to our kernel). For now, we will set the flags to 0x03. We will look at the multiboot header flags in a bit more detail later.

3. A checksum, which can simply be set to multibootHeaderFlags - multibootMagicNumber

Finally, it is time to have a look at some code. We will start with the most trivial kernel written in C that we can imagine. Without printing anything, it will just add up two numbers and die:

Listing 1.1: main.c

```
void cmain ()
{
   int sum = 1+1; // even in this world 1 and 1 makes 2
}
```

As this trivial C program does not print anything to screen, we will not be able even to check whether it runs, and if so, whether it runs correctly. Let us not worry about that now. We will add basic I/O functions soon enough. First, we have to make sure that the C code gets executed. Grub will not do this for us, but we can make it execute some bootstrap code in assembly that subsequently calls into our C code.

Listing 1.2 shows very basic bootstrap code that allows us to call into our kernel. At first sight, it looks like a sizeable program. However, the file contains hardly any assembly. Most of it is filled with definitions (and comments!). In a nutshell, the following takes place:

1. Grub starts executing our kernel by means of a jump to the (assembly) entry point for our kernel, which is the instruction at the address indicated by the global label `start` (line 26). Now we are out of Grub's clutches and executing our own code.

2. We make sure that the magic values for multiboot are stored almost immediately after the entry point (lines 37–40).

3. Since these are values and not executable code, we jump over them (the jump in line 32 will continue execution at line 42).

4. We want to leave assembly as quickly as possible. All we want is to set up a stack and then start executing C code. By convention, stacks grow from high to low, so in line 47 we load the stack pointer (esp) with the address of the top of the stack (space for the stack is defined in line 64).

5. All we need to do now is call our C function (line 50); assume that we will call the function to call '`cmain`'.

6. When we return from our C code we enter an infinite loop in line 53. The `hlt` instruction is meant to halt the CPU when no immediate work needs to be done. It is run in Windows in the System Idle Process.

Listing 1.2: boot.S: bootstrap code with kernel entry point

```
1. /* boot.S - bootstrap the kernel */
2. /* first we give some definitions, to make our code more readable later */
3.
4. #define ASM     1
5.
6. #define MULTIBOOT_HEADER_MAGIC  0x1BADB002 /* magic no. for multiboot header */
7. #define MULTIBOOT_HEADER_FLAGS  0x00000003 /* flags for multiboot header */
8. #define STACK_SIZE 0x4000                  /* size of our stack (16KB) */
9.
10. /* On some systems we have to jump from assembly to C by referring to
11.    the C name prefixed by an underscore. This is all defined during
12.    configuration. We do not worry about it here and make sure we can
13.    handle either (HAVE_ASM_USCORE is defined by configure). */
14.
15. #ifdef HAVE_ASM_USCORE
16. # define EXT_C(sym)                    _ ## sym
17. #else
18. # define EXT_C(sym)                    sym
19. #endif
20.
21.         /* The text segment will contain code, the data segment data
22.         and the bss segment zero-filled static variables */
23.
24.         .text
25.         .globl  start /* start is the global entry point into kernel */
26. start:
27.
28.         /* As we need the multiboot magic values in the beginning of
29.        our file, we will add them presently. The real code
30.        continues right after those 3 long words, so jump over them
31.      */.
32.         jmp     multiboot_entry
33.
34.         /* Align the multiboot header at a 32 bits boundary.  */
35.         .align 4
36.
37. multiboot_header: /* Now comes the multiboot header.  */
38.         .long   MULTIBOOT_HEADER_MAGIC
39.         .long   MULTIBOOT_HEADER_FLAGS
40.         .long   -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)
41.
42. multiboot_entry:
43.         /* We do not like assembly. All we want is to create a
44.            a stack and start executing C code */
45.
46.         /* Initialize the stack pointer (definition of stack follows) */
47.         movl    $(stack + STACK_SIZE), %esp
48.
49.         /* Now enter the C main function (EXT_C will handle the underscore, if needed)
*/
50.         call    EXT_C(cmain)
51.
52.         /* Halt.  When we return from C, we end up here. */
53. loop:   hlt
54.         jmp     loop
55.
56. .section ".bss"
57.         /* We define our stack area. The .comm pseudo op declares
58.            a common symbol (common means that if a symbol with the
59.            same name is defined in another object file, it can be
60.            they can be merged (after all, we only need one stack).
61.            If the loader (ld) does not find an existing definition,
62.            it will allocate STACK_SIZE bytes of uninitialised
63.            memory. */
64.         .comm   stack, STACK_SIZE
```

In the listing, we see that a program consists of three main segments. The text segment contains the executable code, the data segment contains data, and the bss segment contains zero-filled static variables. Given this, there is one more thing that we need to specify before we can start compiling and running our code: the locations in memory where we want the linker to place our segments. For instance, should the text segment start at physical address 0x100000, physical address 0x200000, or somewhere else entirely? The same should be answered for the data and bss segments. In addition, we must specify how the segments should be aligned and we must inform the linker that 'start' is the entry point of our code.

The *linker* is responsible for combining different output files (such as `boot.S` and `main.c`). By

means of a *linker script*, we can specify for each of the segments address, size, alignment, etc. Without much explanation, we now give a linker script for our example kernel (see Figure 1.4). The entry point is indicated by the `start` symbol. We define our (physical) start address to be 0x00100000 (1M). Thus, the text segment starts at physical address 0x00100000 which is aligned at page size (4KB). A bit above the text segment starts our data segment[4] and a bit above that we find our bss segment.

Listing 1.3: link.ld: Linker script for simple kernel

```
ENTRY(start)
phys = 0x00100000;
SECTIONS
{
  .text phys : AT(phys) {
    code = .;
    *(.text)
    *(.rodata)
    . = ALIGN(4096);
  }
  .data : AT(phys + (data - code))
  {
    data = .;
    *(.data)
    . = ALIGN(4096);
  }
  .bss : AT(phys + (bss - code))
  {
    bss = .;
    *(.bss)
    . = ALIGN(4096);
  }
  end = .;
}
```

With the linker script we have all the pieces of our puzzle. Compiling the code is straightforward and so is running our shiny new kernel under Qemu. Since we will be doing a lot of compiling in the course of this text, we will use a Makefile, to capture the compilation commands. For our code, the Makefile below will do.

Listing 1.4: Makefile

```
CFLAGS  := -fno-stack-protector -fno-builtin -nostdinc -O -g -Wall -I.

all:    kernel.bin

kernel.bin:     boot.o  main.o
        ld -T link.ld -o kernel.bin boot.o main.o
        @echo Done!

clean:
      rm -f *.o *.bin
```

You may wonder what all the options are that are specified in CFLAGS. As you probably know, CFLAGS are the options that are passed to the C compiler. In this case, we specify that we do not want gcc to muck around with our addresses much, that we are not interested in all sorts of libraries and that we want to compile it with some (but not much) optimisation and with all debugging symbols present. Table 1.1 briefly explains the exact meaning of the various options.

At this point, we are solely interested in building the kernel and running it. Assuming all code and the `core.img` file are in the same directory, we simply execute the following:

```
make
mcopy kernel.bin c:/boot/grub/
qemu -hda core.img
```

You may not be very impressed as you do not actually see anything. But really, we just made the most important step of all. We are executing our new kernel's code that we programmed in C! All we need now is some I/O functions to allow our kernel to send its greetings to the world.

[4] `data - code` means the offset of the data segment relative to the text segment, as code is defined as pointing to the start of the text segment.

option	meaning
-fno-stack-protector	Do not emit extra code to check for buffer overflows, such as stack smashing attacks.
-fno-builtin	Don't recognize built-in functions that do not begin with __builtin_ as prefix. Gcc normally treats certain built-in functions (like memcpy, and printf) differently to make the resulting code smaller and faster. This option makes it easier to set breakpoints.
-nostdinc	Do not search the standard system directories for header files. Only the directories you have specified with -I options (plus that of current file).
-O	Optimise compilation (makes code run faster)
-g	Produce debugging information in the OS's native format
-Wall	Enables all warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
-I.	Also look for header (include) files in the current directory.

Table 1.1: gcc options explained

1.5 Hello World!

The problem with I/O is that functions like `printf` are not supported in our trivial kernel. If we want to print things on the screen, we will have to write all the required support ourselves. For simple, console-based IO with VGA, this is not very hard to do.

In this section we will add minimal support functions to make a programmer's life easier and to allow for printing on screen. Since we are now in C, adding functions itself is straightforward. For instance, let us start by adding a few memory manipulation functions in a separate file `mem.c`. These functions are so simple that we simply give them without any explanation. We will see shortly why we need these functions. At any rate, they are good functions to have.

Listing 1.5: mem.c: simple memory functions

```
/* Convenient functions for manipulating memory. We do not have
   standard libc functions, so we must implement everything ourselves
 */
unsigned char *memcpy(unsigned char *dest, const unsigned char *src, int count)
{
  int i;
  for (i=0; i<count;i++) dest[i]=src[i];
  return dest;
}
unsigned char *memset(unsigned char *dest, unsigned char val, int count)
{
  int i;
  for (i=0; i<count;i++) dest[i]=val;
  return dest;
}
```

1.5.1 Basic I/O

Memory copies and initialisations are all well and good, but it is I/O we are aiming for. Our basic strategy is to provide two fundamental I/O functions (`inportb()` and `outportb()`) that allow us to read and write I/O ports. For instance, we may use `inportb` to read a single byte from the keyboard, and `outportb` to write a single byte to the screen.

These fundamental functions map exactly on two fundamental instructions in the x86 instruction set for performing I/O: `inb` and `outb`. Unfortunately, these instructions are only available from assembly, but the assembly is really simple (one instruction) and can be easily wrapped in a C function. Listing 1.5.2 shows how this is implemented in C using a (tiny) bit of inline assembly.

Listing 1.6: basicio.c: the primitive I/O functions `inbyte` and `outbyte`

```
/* We use inbyte for reading from the I/O ports to get data from *
   devices such as the keyboard. To do so, we need the 'inb'
```

```
   instruction, which is only accessible from assemby. So the C
   function is simply a wrapper around a single assembly
   instruction.
*/
uint8_t inbyte (uint16_t port)
{
    uint8_t ret;
    __asm__ __volatile__ ("inb %1, %0" : "=a" (ret) : "d" (port));
    return ret;
}

/* We use outbyte to write to I/O ports, i.e., to send bytes to
*  devices. Again, we use inline assembly for the stuff that cannot be
*  done in C. */

void outbyte (uint16_t  port, uint8_t data)
{
    __asm__ __volatile__ ("outb %1, %0" : : "d" (port), "a" (data));
}
```

Inline assembly is easily worth a book in its own right[5], but in this chapter, we limit ourself to fairly mundane usage. In the first function, we see that the instruction `inb` is called with two parameters. The second argument (right after the first colon) lists all the output registers. In this case, the return value `ret` will be returned in the `a` register (the `=a` 'constraint' means that the return valie will be in `eax`). As input (specified after the second colon) it takes an operand in the `d` (which indicates `edx`) register and which represents an I/O port. The `outbyte` function works similarly. It does not have output operands, but takes two inputs: the I/O port (in register `edx`) and a data byte (in the `eax` register).

1.5.2 Writing to screen

We will show how the above two functions, `inbyte` and `outbyte` help us print characters on the screen. We limit ourselves to fairly simple VGA-based output. VGA stands for Video Graphics Array and represents an interface between a computer and its corresponding monitor. The VGA card is easily the most common video card, supported by almost any video card. Better still, it is really easy to program VGA in (colour) text mode.

In a nutshell, the card offers an area of memory that starts at address 0xB8000[6] to which we can simply write characters consisting of two byte values: the character itself is in the lower byte, while an attribute byte is the highest byte. The attribute byte represents the background colour in its most significant four bits and the foreground colour in its least significant four bits. You can play with these colours yourself. For example: black is 0x0, and white is 0xF, so we can print a black on white 'B' by storing the value `(0xF0<<8)|'B'` at the appropriate position in the memory area.

The memory area itself is flat but represents an 80x25 matrix of these 16-bit characters laid out as 25 consecutive lines. The origin $(0, 0)$ represents the top left corner. The first 80 characters (of 16b each)represent the top line. The next 80 characters represent the next line, and so on. This means that if we want to print a character at position (x, y), we have to calculate the offset from the base address as follows: $offset = y * 80 + x$. We can print a black 'B' on a white background at location $(1, 2)$ as follows:

```
#define VGA_START 0xB8000
uint16_t addr = VGA_START + 2*80 + 1;
*addr = (0xF0<<8)|'B'
```

Rather than printing each character individually, we will write a few functions to facilitate the printing of strings and other values. The code is extremey straightforward and we give it without further explanation.

[5] In fact, the web hosts several long tutorials on the topic, such as `http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html` and `http://www.ibm.com/developerworks/linux/library/l-ia.html`

[6] VGA offers other memory areas as well. For instance, if you want to use graphics, you want to use the area that starts at 0xA0000 (and set the card to mode 0x13),

Listing 1.7: scrn.c: Functions for printing to screen

```
#include "types.h"   // uint16_t, etc.
#include "mem.h"     // memcpy and memset
#include "basicio.h" // inbyte and outbyte

#define COLOURS 0xF0
#define COLS 80
#define ROWS 25
#define VGA_START 0xB8000
#define PRINTABLE(c) (c>=' ') /* is this a printable character? */

uint16_t *Scrn; // screen area
int Curx, Cury = 0; // current cursor coordinates
uint16_t EmptySpace = COLOURS << 8 | 0x20; /* 0x20 is ascii value of space */

// scroll the screen (a 'copy and blank' operation)
void scroll(void)
{
  int dist = Cury - ROWS + 1;

  if (dist > 0) {
    uint8_t *newstart = ((uint8_t*) Scrn) + dist * COLS *2;
    int bytesToCopy = (ROWS-dist)*COLS*2;
    uint16_t *newblankstart = Scrn + (ROWS-dist) * COLS;
    int bytesToBlank = dist*COLS*2;
    memcpy ((uint8_t*)Scrn, newstart, bytesToCopy);
    memset ((uint8_t*) newblankstart, EmptySpace, bytesToBlank);
  }
}

// Print a character on the screen
void putchar (uint8_t c)
{
    uint16_t *addr;

    // first handle a few special characters

    // tab -> move cursor in steps of 4
    if(c == '\t') Curx = ((Curx + 4)/4)*4;
    // carriage return -> reset x pos
    else if(c == '\r') Curx = 0;
    // newline: reset x pos and go to newline
    else if(c == '\n')
    {
        Curx = 0;
        Cury++;
    }
    // backspace -> cursor moves left
    else if(c == 0x08 && Curx != 0) Curx--;
    // finally, if a normal character, print it
    else if(PRINTABLE(c))
    {
        addr = Scrn + (Cury * COLS + Curx);
        *addr = (COLOURS<<8) | c;
        Curx++;
    }


    // if we have reached the end of the line, move to the next
    if (Curx >= COLS)
    {
        Curx = 0;
        Cury++;
    }

    // also scroll if needed
    scroll();
}

// print a longer string
void puts(unsigned char *str)
{
  while (*str) { putchar(*str); str++;}
}

void itoa (char *buf, int base, int d)
{
  char *p = buf;
  char *p1, *p2;
  unsigned long ud = d;
```

```
  int divisor = 10;

  /* If %d is specified and D is minus, put `-' in the head.  */
  if (base == 'd' && d < 0)
    {
      *p++ = '-';
      buf++;
      ud = -d;
    }
  else if (base == 'x')
    divisor = 16;

  /* Divide UD by DIVISOR until UD == 0.  */
  do
    {
      int remainder = ud % divisor;

      *p++ = (remainder < 10) ? remainder + '0' : remainder + 'a' - 10;
    }
  while (ud /= divisor);

  /* Terminate BUF.  */
  *p = 0;

  /* Reverse BUF.  */
  p1 = buf;
  p2 = p - 1;
  while (p1 < p2)
    {
      char tmp = *p1;
      *p1 = *p2;
      *p2 = tmp;
      p1++;
      p2--;
    }
}


// Format a string and print it on the screen, just like the libc
// function printf.
void
printf (const char *format, ...)
{
  char **arg = (char **) &format;
  int c;
  char buf[20];

  arg++;

  while ((c = *format++) != 0)
    {
      if (c != '%')
        putchar (c);
      else
        {
          char *p;

          c = *format++;
          switch (c)
            {
            case 'd':
            case 'u':
            case 'x':
              itoa (buf, c, *((int *) arg++));
              p = buf;
              goto string;
              break;

            case 's':
              p = *arg++;
              if (p == NULL)
                p = "(null)";

            string:
              while (*p)
                putchar (*p++);
              break;

            default:
              putchar (*((int *) arg++));
```

```
                break;
            }
        }
    }
}

// Clear the screen
void clear()
{
  int i;
  for(i = 0; i < ROWS*COLS; i++) putchar (' ');
  Curx = Cury = 0;
  //Scrn[i] = EmptySpace;

}

// init and clear the screen
void vga_init(void)
{
    Scrn = (unsigned short *)VGA_START;
    clear();
}
```

We wrap up our very basic kernel with a simple main function. It prints its greeting and starts an endless loop. It still is not much, but we do have a kernel with primitive I/O. In the next few chapters we will add (more interesting) drivers and other interesting features to our basic kernel.

Listing 1.8: main.c: simple main function

```
  void cmain (unsigned long magic, unsigned long addr)
{

  vga_init();
  puts ((uint8_t*)"hello world!");
  for(;;); // start infinite loop
}
```

We are done with the first chapter. Assuming that we saved the main function in main.c, memset and memcopy in mem.c, inbyte and outbyte in basicio.c, vga functions in scrn.c, and bootstrap code in boot.S and that we included the appropriate header files everywhere, we can now simply build the kernel and boot it. Here is the makefile:

Listing 1.9: Makefile: hello world with VGA output

```
CFLAGS  := -fno-stack-protector -fno-builtin -nostdinc -O -g -Wall -I.

all:    kernel.bin
kernel.bin:     boot.o  main.o mem.o basicio.o scrn.o
        ld -T link.ld -o kernel.bin boot.o main.o mem.o basicio.o scrn.o
        @echo Done!
```

Bibliography

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX ATEC '05*, 2005.

[2] B. Friesen. Bran's kernel development tutorial. `http://www.osdever.net/bkerndev/index.php`.