Nano-Kernel : A Bare Metal OS

## Part 2 - Multiboot Kernel - A first kernel

The “multiboot” standard has become the standard executable format for modern PC’s. The typical boot-up sequence is that the computer’s on-board EFI BIOS can directly boot a multiboot executable, or can load a boot utility, such as the “Grand Unified Boot Loader” (a.k.a GRUB). Our first step is to write a multiboot image that can print the letters “hello world” on a PC console. This requires some understanding of the way the PC video system is arranged.

The multiboot standard expects an ELF file (Executable and Linker File)[[1]](#footnote-1) with a specifically named “multiboot” section that contains a multiboot header. The following assembly file will set this up:

|  |
| --- |
| /\* Declare constants for the multiboot header. \*/ .set ALIGN, 1<<0 /\* align loaded modules on page boundaries \*/ .set MEMINFO, 1<<1 /\* provide memory map \*/ .set FLAGS, ALIGN | MEMINFO /\* this is the Multiboot 'flag' field \*/ .set MAGIC, 0x1BADB002 /\* 'magic number' lets bootloader find the header \*/ .set CHECKSUM, -(MAGIC + FLAGS) /\* checksum of above, to prove we are multiboot \*/  /\*  Declare a multiboot header that marks the program as a kernel. These are magic values that are documented in the multiboot standard. The bootloader will search for this signature in the first 8 KiB of the kernel file, aligned at a 32-bit boundary. The signature is in its own section so the header can be forced to be within the first 8 KiB of the kernel file. \*/ .section .multiboot .align 4 .long MAGIC .long FLAGS .long CHECKSUM |

<https://wiki.osdev.org/Bare_Bones>

When an ELF file contains this header then a multiboot loader, such as EFI or GRUB will then further inspect the file to determine whether it can be executed. According to the multiboot rules, this header must be located within the first 8KB of memory. Later we will see how to tell the linker to respect this rule.

### PC Video Memory

One of the early goals of this project is to print out “Hello World”, but it’s a little more complicated than usual. Because we are writing code on a “bare metal” machine, we need to handle writing characters ourselves. In fact, we even need to make the cursor move as we write characters! The good news is that there is some hardware support from the video controller.

The PC’s video memory is 32KB starting at address 0xB8000. Each page is, by default, 80 columns by 25 rows. Each text position uses 16-bits:

|  |  |  |
| --- | --- | --- |
| Bits 15:12 | Bits 11:8 | Bits 7:0 |
| Background Color | Foreground Color | Character |

But, in Little Endian format:

|  |  |  |
| --- | --- | --- |
| Bits 7:0 | Bits 15:12 | Bits 11:8 |
| Character | Background Color | Foreground Color |

One display full of memory requires 80 x 25 x 2 = 4,000 bytes of memory, thus there are :

\lfloor 32768 {B} \div 4000 \textrm{B per page}\rfloor = 8 \textrm{ pages}

The memory is organized in a row-major order, starting at row 0, column zero in the upper-left hand corner of the screen, and row 24, column 79 in the bottom right corner of the screen. Using the standard integer definitions found in <stdint.h> we can determine the address of a page, or of a character in the video array:

volatile uint16\_t \* get\_page(int page\_num) {  
 return (volatile uint16\_t \*)(0xb8000 + (4096 \* page\_num));  
}

// offset expressed in terms of 16-bit pointers!

uint16\_t get\_offset(int row, int col) {  
 return (row \* 80 + col);  
}

The colors are made from combining a foreground and a background color from the following table:

[[2]](#footnote-2)

For example, we could write a function to put a character onto a location in memory:

void write\_character(uint8\_t page\_num,   
 uint8\_t row, uint8\_t col,   
 uint8\_t bgcolor, uint8\_t fgcolor, uint8\_t ch)  
{

volatile uint16\_t \*page = get\_page(page\_num);  
 uint16\_t offset = get\_offset(row, col);  
 page[offset] = bgcolor << 12 | fgcolor << 8 | ch;  
}

Of course, we could refine this a little further and create a console driver that will make this a more useful tool.

### Creating a Kernel - version 1

So, if we put all of these pieces together, we are ready to create a basic kernel that can display the letter “A” on the screen in bright white text on a blue background.

Create a file “boot.s” that concatenates the multiboot header with the test code that were both given earlier. After the “\_start” symbol, you will need to insert code to:

* Move the “stack\_top” symbol to the 32-bit stack pointer register
* Call your “kernel\_main” function (that will be defined in a C file we haven’t written yet)

Your “boot.s” file will be compiled into an object file using the Assembler, it will not be runnable on its own - because it depends on the C file that contains kernel\_main, and the segments listed in the “.s” file need to be assigned to a location in memory.

Create a file “kernel.c” that contains a function: void kernel\_main(). This will be a C function that kicks off your kernel. You are now, basically, programming like you did in microcontrollers.

Edit your “kernel” to display the letter A as required. Since its done, it can just return - back to the \_start code in the boot.s file, which will then loop forever.

### Linker Script

The following is a “linker script” - a file that tells the GNU linker how to structure your ELF file. This file should be called “linker.ld” (or else you’ll need to edit the Makefile in the next step)

|  |
| --- |
| /\* what symbol is the first instruction \*/  ENTRY(\_start)  /\* Describe the sections of the program \*/ SECTIONS {  /\* Begin putting sections at 1 MiB,  loaded at by the bootloader. \*/  . = 1M;  /\* First put the multiboot header, as it is required to be put very early  early in the image or the bootloader won't recognize the file format.  Next we'll put the .text section. \*/  .text BLOCK(4K) : ALIGN(4K)  {  \*(.multiboot)  \*(.text)  }   /\* Read-only data. \*/  .rodata BLOCK(4K) : ALIGN(4K)  {  \*(.rodata)  }   /\* Read-write data (initialized) \*/  .data BLOCK(4K) : ALIGN(4K)  {  \*(.data)  }   /\* Read-write data (uninitialized) and stack \*/  .bss BLOCK(4K) : ALIGN(4K)  {  \*(COMMON)  \*(.bss)  }   /\* The compiler may produce other sections and will put them in  a segment with the same name. Simply add stuff here as needed. \*/ } |

Finally, we need to compile, assemble, and link all of these different parts into one executable. This is the job of the make utility.

|  |
| --- |
| TARGET=i686-elf  AS:=$(TARGET)-as  CC:=$(TARGET)-gcc  LD:=$(TARGET)-ld  CFLAGS := -ffreestanding -O2 -Wall -Wextra -nostdlib -nostartfiles -nodefaultlibs -Wl,--build-id=none  LIBS:=-L/opt/cross/lib/gcc/i686-elf/6.4.0 -lgcc  OBJS:=\  boot.o \  kernel.o \  all: myos.bin  .PHONEY: all clean  myos.bin: $(OBJS) linker.ld  $(LD) -T linker.ld -o $@ $(LDFLAGS) $(OBJS) $(LIBS)  %.o: %.c  $(CC) -c $< -o $@ -std=gnu99 $(CFLAGS)  %.o: %.S  $(GCC) $(CFLAGS) $< -o $@  clean:  rm -rf isodir  rm -f myos.bin myos.iso $(OBJS) |

The “%.o: %c” rule instructs Make how to convert a C file to a “.o” file. The “%o: %s” rule instructions how to assemble an assembler file to a .o. The “.PHONEY” rule tells Make that the targets “all” and “clean” don’t have any dependencies, so they should always be run.

So, you should have: linker.ld, kernel.c, and boot.s. This Makefile will create a file called “myos.bin” from these sources.

You should now be able to do a “make all” and verify that the file is successfully created.

**Note: if you are being a lazy sot, you will find that copy-paste doesn’t preserve tabs. This is not a new problem - Emacs has the “tabify” command - but then, that implies you took the time to learn how to use a real text editor!**

### QEMU Image

QEMU is a system emulator (different from virtualization). To make this a runnable system for QEMU we need to wrap the executable into a disk image. For now, we will use the standard ISO format (used for CD roms and other bootable devices).

You will need a default “grub.cfg” file:

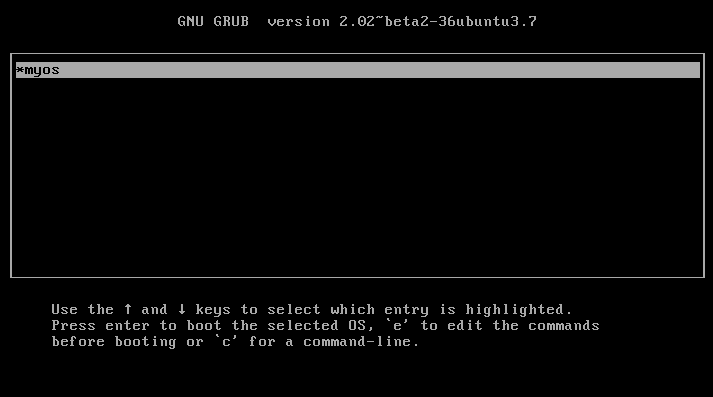
|  |
| --- |
| menuentry "myos" {  multiboot /boot/myos.bin  } |

An addendum to the “makefile” is below:

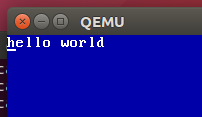
|  |
| --- |
| iso: myos.iso  isodir isodir/boot isodir/boot/grub:  mkdir -p $@  isodir/boot/myos.bin: myos.bin isodir/boot  cp $< $@  isodir/boot/grub/grub.cfg: grub.cfg isodir/boot/grub  cp $< $@  myos.iso: isodir/boot/myos.bin isodir/boot/grub/grub.cfg  grub-mkrescue -o $@ isodir  run-qemu: myos.iso  qemu-system-i386 -cdrom myos.iso |

Now, you can simply type “make run-qemu” which depends on “myos.iso”, which depends on myos.bin, which depends the source code from above.

If you run: $ make run-qemu you should see window :



Pressing the enter key should boot your OS and then display the desired result:



### Debugging QEMU Kernel

The venerable GDB command line debugger was introduced in 1986 and has evolved into the Swiss army knife of the programmer / embedded systems developer. In this section we will use GDB to debug the compiled kernel.

You may already be familiar with using GDB to debug an executable, where GDB will launch the program and control its debugging. GDB can also connect to an existing executable or in this case, an executable running on a remote machine. GDB will allow us to debug the process remotely.

#### CFLAGS for debugging

To make debugging useful we need to tell the C compiler to preserve the debugging information - specifically the information that ties our source code to the compiled executable code. This way GDB can show us source code and variable names to go along with assembly instructions and memory. The standard flag is “-g” or even “-ggdb”. This is typically added to the CFLAGS environment in any Makefile. **Do so now.**

#### QEMU Enable Debugging

QEMU will launch a GDB listener by using the “-s” command line option. Add this to the “qemu-system-i386” line in the Makefile. This will cause QEMU to listen on TCP port #1234 for incoming GDB requests.

#### Connect GDB and QEMU

1. Boot the QEMU system through to the GRUB menu.
2. Run GDB:  
   $ **i686-elf-gdb**
3. Target the QEMU remote system:  
   (gdb) **target remote localhost:1234**  
   Remote debugging using localhost:1234  
   warning: No executable has been specified and target does not support  
   determining executable automatically. Try using the "file" command.  
   0x000090fa in ?? ()
4. Load the symbols from the executable:  
   (gdb) **symbol-file myos.bin**   
   Reading symbols from myos.bin...done.  
   (gdb)
5. Set a break-point where you are interested in stopping the program:

(gdb) **b \_start**

Breakpoint 1 at 0x10000c: file boot.S, line 28.

(gdb)

1. Release QEMU to run:  
   (gdb) **continue**

Continuing.

1. Press “enter” on the GRUB menu and the boot will start and then hit the breakpoint set in GDB:  
   Continuing.

Breakpoint 1, \_start () at boot.S:28

28 lgdt gdtdesc

(gdb)

You can place these commands into a text file, like “**gdb-start**” and then tell GDB to run the command at startup: **i686-elf-gdb -x gdb-start.**

At this point, you can interact with GDB the way you normally would. There are many, many guides on using GDB. Importantly, you can debug at the assembly or the C language level. When debugging the assembly code it is useful to single step instructions and watch the registers values. When debugging C code it useful to step over lines and inspect memory. You can do both.

# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. I want to see your code – it should be well documented, including doxygen style comments (or similar standard), and should be well-organized into separate files by subject, with .h and .c files as appropriate.
2. I want to see that you are using const and volatile keyword wherever appropriate – it would help if you are ready to show me where you think you’ve used them and why.
3. Show me your “hello world” program running in your VM
4. Connect with GDB, set a break-point to your kernel’s main, and also to your write\_character or equivalent and then show me how you are writing out each character.

Deliver to me answers to the following questions (1 answer per group):

1. What companies were involved in designing Multiboot? Why? What did it replace in the PC ecosystem? Is it still the standard for PCs, if not, what replaced it?
2. The video memory is “mapped” into the PC’s main memory – in the early PCs, are there any other devices that use this technique?
3. Do a little research into the VESA VGA standard – would it be possible to pick different background colors and change fonts using more modern hardware? If so describe what is necessary – you don’t have to actually do it, just talk about it.
4. In the linker script, what does the “.” mean?
5. Using the internet (for something other than cat videos), show the syntax that would create a struct, containing three 8-bit integers, that would occupy exactly 24-bits of memory space and would be placed into a linker section named “question5”.
6. Using the templates above, show the syntax for how you would create the section for the section in the previous question.

Points: \_\_\_\_\_\_\_\_\_ / 40

1. DWARF files are also common underneath the operation of a system - they contain the debugging and symbolic information of the ELF file. [↑](#footnote-ref-1)
2. https://en.wikipedia.org/wiki/Enhanced\_Graphics\_Adapter [↑](#footnote-ref-2)