The Basic Kernel

In this section of the tutorial, we will delve into a bit of assembler, learn the basics of creating a linker script as well as the reasons for using one, and finally, we will learn how to use a batch file to automate the assembling, compiling, and linking of this most basic protected mode kernel. Please note that at this point, the tutorial assumes that you have NASM and DJGPP installed on a Windows or DOS-based platform. We also assume that you have a a minimal understanding of the x86 Assembly language.

The Kernel Entry

The kernel's entry point is the piece of code that will be executed FIRST when the bootloader calls your kernel. This chunk of code is almost always written in assembly language because some things, such as setting a new stack or loading up a new GDT, IDT, or segment registers, are things that you simply cannot do in your C code. In many beginner kernels as well as several other larger, more professional kernels, will put all of their assembler code in this one file, and put all the rest of the sources in several C source files.

If you know even a small amount of assembler, the actual code in this file should be very straight forward. As far as code goes, all this file does is load up a new 8KByte stack, and then jump into an infinite loop. The stack is a small amount of memory, but it's used to store or pass arguments to functions in C. It's also used to hold local variables that you declare and use inside your functions. Any other global variables are stored in the data and BSS sections. The lines between the 'mboot' and 'stublet' blocks make up a special signature that GRUB uses to verify that the output binary that it's going to load is, infact, a kernel. Don't struggle too hard to understand the multiboot header.

```
; This is the kernel's entry point. We could either call main here,
; or we can use this to setup the stack or other nice stuff, like
; perhaps setting up the GDT and segments. Please note that interrupts
; are disabled at this point: More on interrupts later!
[BITS 32]
global start
start:
    mov esp, _sys_stack     ; This points the stack to our new stack area
    jmp stublet

; This part MUST be 4byte aligned, so we solve that issue using 'ALIGN 4'
ALIGN 4
mboot:
    ; Multiboot macros to make a few lines later more readable
    MULTIBOOT_PAGE_ALIGN      equ 1<<0
    MULTIBOOT_MEMORY_INFO     equ 1<<1
    MULTIBOOT_AOUT_KLUDGE     equ 1<<16
    MULTIBOOT_HEADER_MAGIC    equ 0x1BADB002
    MULTIBOOT_HEADER_FLAGS    equ MULTIBOOT_PAGE_ALIGN | MULTIBOOT_MEMORY_INFO | MULTIBOOT_AOUT_KLUDGE
    MULTIBOOT_CHECKSUM        equ -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)
    EXTERN code, bss, end

    ; This is the GRUB Multiboot header. A boot signature
    dd MULTIBOOT_HEADER_MAGIC
    dd MULTIBOOT_HEADER_FLAGS
    dd MULTIBOOT_CHECKSUM

    ; AOUT kludge - must be physical addresses. Make a note of these:
    ; The linker script fills in the data for these ones!
```

```
    dd mboot
    dd code
    dd bss
    dd end
    dd start

; This is an endless loop here. Make a note of this: Later on, we
; will insert an 'extern _main', followed by 'call _main', right
; before the 'jmp $'.
stublet:
    jmp $


; Shortly we will add code for loading the GDT right here!


; In just a few pages in this tutorial, we will add our Interrupt
; Service Routines (ISRs) right here!



; Here is the definition of our BSS section. Right now, we'll use
; it just to store the stack. Remember that a stack actually grows
; downwards, so we declare the size of the data before declaring
; the identifier '_sys_stack'
SECTION .bss
    resb 8192               ; This reserves 8KBytes of memory here
_sys_stack:
```

The kernel's entry file: 'start.asm'

The Linker Script

The Linker is the tool that takes all of our compiler and assembler output files and links them together into one binary file. A binary file can have several formats: Flat, AOUT, COFF, PE, and ELF are the most common. The linker we have chosen in our toolset, if you can remember, was the LD linker. This is a very good multi-purpose linker with an extensive feature set. There are versions of LD that exist which can output a binary in any format that you wish. Regardless of what format you choose, there will always be 3 'sections' in the output file. 'Text' or 'Code' is the executable itself. The 'Data' section is for hardcoded values in your code, such as when you declare a variable and set it to 5. The value of 5 would get stored in the 'Data' section. The last section is called the 'BSS' section. The 'BSS' consists of uninitialized data; it stores any arrays that you have not set any values to, for example. 'BSS' is a virtual section: It doesn't exist in the binary image, but it exists in memory when your binary is loaded.

What follows is a file called an LD Linker Script. There are 3 major keywords that might pop out in this linker script: OUTPUT_FORMAT will tell LD what kind of binary image we want to create. To keep it simple, we will stick to a plain "binary" image. ENTRY will tell the linker what object file is to be linked as the very first file in the list. We want the compiled version of 'start.asm' called 'start.o' to be the first object file linked, because that's where our kernel's entry point is. The next line is 'phys'. This is not a keyword, but a variable to be used in the linker script. In this case, we use it as a pointer to an address in memory: a pointer to 1MByte, which is where our binary is to be loaded to and run at. The 3rd keyword is SECTIONS. If you study this linker script, you will see that if defines the 3 main sections: '.text', '.data', and '.bss'. There are 3 variables defined also: 'code', 'data', 'bss', and 'end'. Do not get confused by this: the 3 variables that you see are actually variables that

are in our startup file, start.asm. ALIGN(4096) ensures that each section starts on a 4096byte boundary. In this case, that means that each section will start on a separate 'page' in memory.

```
OUTPUT_FORMAT("binary")
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 = .;
}
```

The Linker Script: 'link.ld'

Assemble and Link!

Now, we must assemble 'start.asm' as well as use the linker script, 'link.ld' shown above, to create our kernel's binary for GRUB to load. The simplest way to do this in Unix is to create a makefile script to do the assembling, compiling, and linking for you, however, most of the people here including myself, use a flavour of Windows. Here, we can create a batch file. A batch file is simply a collection of DOS commands that you can execute with one command: the name of the batch file itself. Even simpler: you just need to double-click the batch file in order to compile your kernel under windows.

Shown below is the batch file we will use for this tutorial. 'echo' is a DOS command that will say the following text on the screen. 'nasm' is our assembler that we use: we compile in aout format, because LD needs a known format in order to resolve symbols in the link process. This assembles the file 'start.asm' into 'start.o'. The 'rem' command means 'remark'. This is a comment: it's in the batch file, but it doesn't actually mean anything to the computer. 'ld' is our linker. The '-T' argument tells LD that a linker script follows. '-o' means the output file follows. Any other arguments are understood as files that we need to link together and resolve in order to create kernel.bin. Lastly, the 'pause' command will display "Press a key to continue..." on the screen and wait for us to press a key so that we can see what our assembler or linker gives out onscreen in terms of syntax errors.

```
echo Now assembling, compiling, and linking your kernel:
nasm -f aout -o start.o start.asm
rem Remember this spot here: We will add 'gcc' commands here to compile C sources


rem This links all your files. Remember that as you add *.o files, you need to
```

```
rem add them after start.o. If you don't add them at all, they won't be in your kernel!
ld -T link.ld -o kernel.bin start.o
echo Done!
pause
```

Our builder batch file: 'build.bat'