www.jamesmolloy.co.uk

Home » JamesM's kernel development tutorials

2. Genesis

2.1. The boot code

OK, It's time for some code! Although the brunt of our kernel will be written in C, there are certain things we just *must* use assembly for. One of those things is the initial boot code.

Here we go:

```
;
; boot.s -- Kernel start location. Also defines multiboot header.
; Based on Bran's kernel development tutorial file start.asm
;

MBOOT_PAGE_ALIGN    equ 1<<0    ; Load kernel and modules on a page boundary
MBOOT_MEM_INFO      equ 1<<1    ; Provide your kernel with memory info
MBOOT_HEADER_MAGIC  equ 0x1BADB002 ; Multiboot Magic value
; NOTE: We do not use MBOOT_AOUT_KLUDGE. It means that GRUB does not
; pass us a symbol table.
MBOOT_HEADER_FLAGS  equ MBOOT_PAGE_ALIGN | MBOOT_MEM_INFO
MBOOT_CHECKSUM      equ -(MBOOT_HEADER_MAGIC + MBOOT_HEADER_FLAGS)


[BITS 32]                       ; All instructions should be 32-bit.

[GLOBAL mboot]                  ; Make 'mboot' accessible from C.
[EXTERN code]                   ; Start of the '.text' section.
[EXTERN bss]                    ; Start of the .bss section.
[EXTERN end]                    ; End of the last loadable section.

mboot:
  dd  MBOOT_HEADER_MAGIC        ; GRUB will search for this value on each
                                ; 4-byte boundary in your kernel file
  dd  MBOOT_HEADER_FLAGS        ; How GRUB should load your file / settings
  dd  MBOOT_CHECKSUM            ; To ensure that the above values are correct

  dd  mboot                     ; Location of this descriptor
  dd  code                      ; Start of kernel '.text' (code) section.
```

```
  dd  bss                       ; End of kernel '.data' section.
  dd  end                       ; End of kernel.
  dd  start                     ; Kernel entry point (initial EIP).

[GLOBAL start]                  ; Kernel entry point.
[EXTERN main]                   ; This is the entry point of our C code

start:
  push    ebx                   ; Load multiboot header location

  ; Execute the kernel:
  cli                         ; Disable interrupts.
  call main                   ; call our main() function.
  jmp $                       ; Enter an infinite loop, to stop the processor
                              ; executing whatever rubbish is in the memory
                              ; after our kernel!
```

2.2. Understanding the boot code

There's actually only a few lines of code in that snippet:

```
push ebx
cli
call main
jmp $
```

The rest of it is all to do with the *multiboot header*.

2.2.1. Multiboot

Multiboot is a standard to which GRUB expects a kernel to comply. It is a way for the bootloader to

1. Know exactly what environment the kernel wants/needs when it boots.
2. Allow the kernel to query the environment it is in.

So, for example, if your kernel needs to be loaded in a specific VESA mode (which is a bad idea, by the way), you can inform the bootloader of this, and it can take care of it for you.

To make your kernel multiboot compatible, you need to add a header structure somewhere in your kernel (Actually, the header must be in the first 4KB of the kernel). Usefully, there is a NASM command that lets us embed specific constants in our code - 'dd'. These lines:

```
dd MBOOT_HEADER_MAGIC
dd MBOOT_HEADER_FLAGS
```

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

Do just that. The MBOOT_* constants are defined above.

MBOOT_HEADER_MAGIC
A magic number. This identifies the kernel as multiboot-compatible.

MBOOT_HEADER_FLAGS
A field of flags. We ask for GRUB to page-align all kernel sections (MBOOT_PAGE_ALIGN) and also to give us some memory information (MBOOT_MEM_INFO). Note that some tutorials also use MBOOT_AOUT_KLUDGE. As we are using the ELF file format, this hack is not necessary, and adding it stops GRUB giving you your symbol table when you boot up.

MBOOT_CHECKSUM
This field is defined such that when the magic number, the flags and this are added together, the total must be zero. It is for error checking.

mboot
The address of the structure that we are currently writing. GRUB uses this to tell if we are expecting to be relocated.

code,bss,end,start
These symbols are all defined by the linker. We use them to tell GRUB where the different sections of our kernel can be located.

On bootup, GRUB will load a pointer to another information structure into the EBX register. This can be used to query the environment GRUB set up for us.

2.2.2. Back to the code again...

So, immediately on bootup, the asm snippet tells the CPU to push the contents of EBX onto the stack (remember that EBX now contains a pointer to the multiboot information structure), disable interrupts (CLI), call our 'main' C function (which we haven't defined yet), then enter an infinite loop.

All is good, but the code won't link yet. We haven't defined main()!

2.3. Adding some C code

Interfacing C code and assembly is dead easy. You just have to know the calling convention used. GCC on x86 uses the __cdecl calling convention:

- All parameters to a function are passed on the stack.
- The parameters are pushed *right-to-left*.
- The return value of a function is returned in EAX.

...so the function call:

```
d = func(a, b, c);
```

Becomes:

```
push [c]
push [b]
push [a]
call func
mov [d], eax
```

See? nothing to it! So, you can see that in our asm snippet above, that 'push ebx' is actually passing the value of ebx as a parameter to the function main().

2.3.1. The C code

```
// main.c -- Defines the C-code kernel entry point, calls initialisation
routines.
// Made for JamesM's tutorials

int main(struct multiboot *mboot_ptr)
{
  // All our initialisation calls will go in here.
  return 0xDEADBABA;
}
```

Here's our first incarnation of the main() function. As you can see, we've made it take one parameter - a pointer to a multiboot struct. We'll define that later (we don't actually need to define it for the code to compile!).

All the function does is return a constant - 0xDEADBABA. That constant is unusual enough that it should stand out at you when we run the program in a second.

2.4. Compiling, linking and running!

Now that we've added a new file to our project, we have to add it to the

makefile also. Edit these lines:

```
SOURCES=boot.o
CFLAGS=
```

To become:

```
SOURCES=boot.o main.o
CFLAGS=-nostdlib -nostdinc -fno-builtin -fno-stack-protector
```

We must stop GCC trying to link your linux C library with our kernel - it won't work at all (yet). That's what those CFLAGS are for.

OK, you should now be able to compile, link and run your kernel!

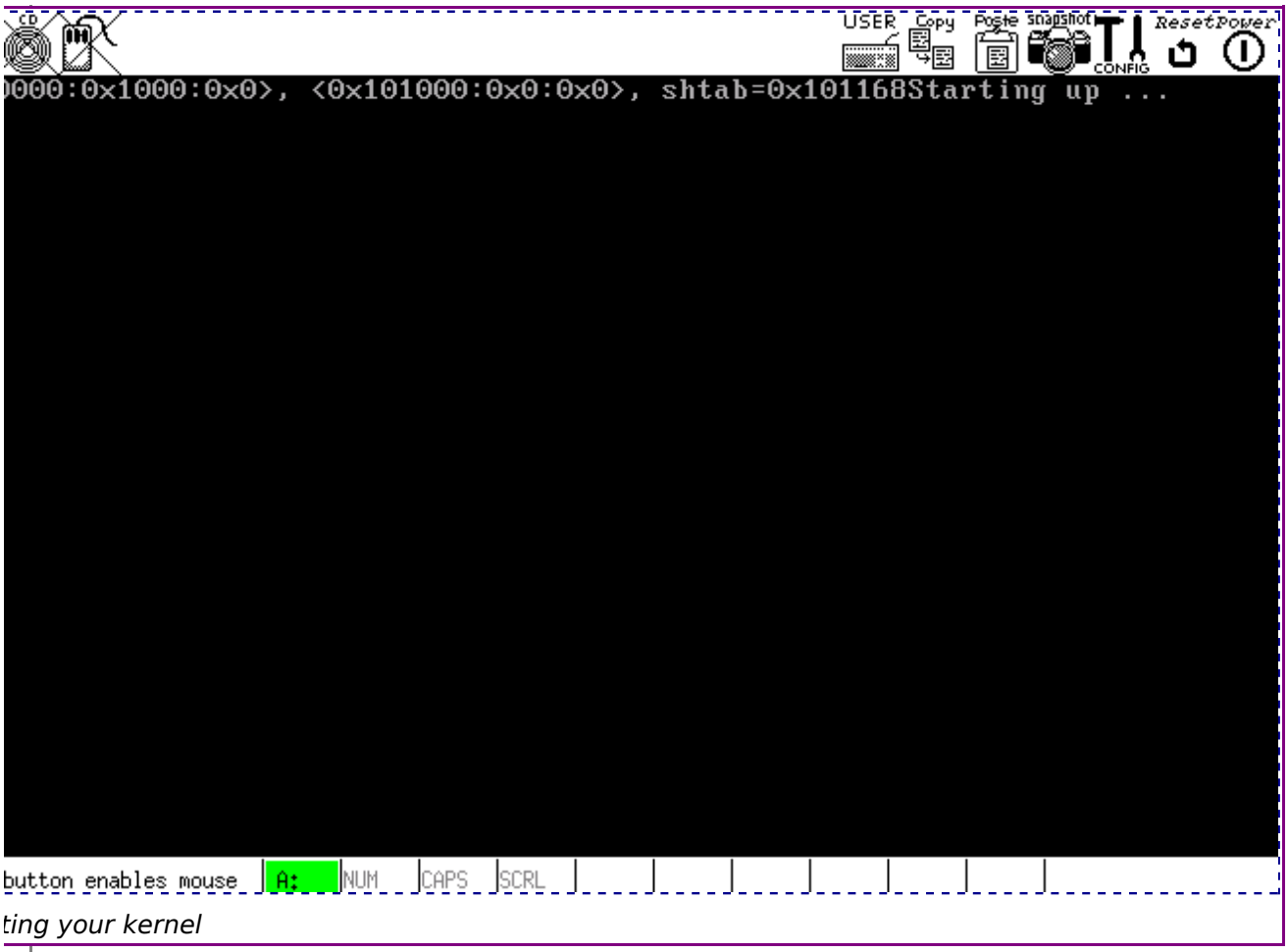
```
cd src
make clean  # Ignore any errors here.
make
cd ..
./update_image.sh
./run_bochs.sh  # This may ask your for your root password.
```

That should cause bochs to boot, you'll see GRUB for a few seconds then the kernel will run. It doesn't actually *do* anything, so it'll just freeze, saying 'starting up...'.

If you open bochsout.txt, at the bottom you should see something like:

```
00074621500i[CPU  ] | EAX=deadbaba  EBX=0002d000  ECX=0001edd0 EDX=00000001
00074621500i[CPU  ] | ESP=00067ec8  EBP=00067ee0  ESI=00053c76 EDI=00053c77
00074621500i[CPU  ] | IOPL=0 id vip vif ac vm rf nt of df if tf sf zf af pf cf
00074621500i[CPU  ] | SEG selector     base    limit G D
00074621500i[CPU  ] | SEG sltr(index|ti|rpl)     base    limit G D
00074621500i[CPU  ] |  CS:0008( 0001| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  DS:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  SS:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  ES:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  FS:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  GS:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] | EIP=00100027 (00100027)
00074621500i[CPU  ] | CR0=0x00000011 CR1=0 CR2=0x00000000
00074621500i[CPU  ] | CR3=0x00000000 CR4=0x00000000
00074621500i[CPU  ] >> jmp .+0xfffffffe (0x00100027) : EBFE
```

Notice what the value of EAX is? 0xDEADBABA - the return value of main(). Congratulations, you now have a multiboot compatible assembly trampoline,



ting your kernel

and you're ready to start printing to the screen!

Sample code for this tutorial can be found here