Stefano Franzoni

mat: 253061

REPORT LAB 5

EXERCISE 2

The GDT and the IDT are descriptor tables. They are arrays of flags and bit values describing the operation of either the segmentation system (in the case of the GDT), or the interrupt vector table (IDT).

We will be using GRUB to load our kernel. To do this, we need a floppy disk image with GRUB preloaded onto it

GRUB sets the GDT. In the x86 system we have 6 segmentation registers. Each holds an offset into the GDT. They are cs (code segment), ds (data segment), es (extra segment), fs, gs, ss (stack segment)

A GDT entry:

```
// This structure contains the value of one GDT entry.
// We use the attribute 'packed' to tell GCC not to change
// any of the alignment in the structure.
struct gdt_entry_struct
{
   u16int limit_low;           // The lower 16 bits of the limit.
   u16int base_low;            // The lower 16 bits of the base.
   u8int  base_middle;         // The next 8 bits of the base.
   u8int  access;              // Access flags, determine what ring this segment
can be used in.
   u8int  granularity;
   u8int  base_high;           // The last 8 bits of the base.
} __attribute__((packed));
```

To tell the processor where to find our GDT, we have to give it the address of a special pointer structure:

```
struct gdt_ptr_struct
{
   u16int limit;               // The upper 16 bits of all selector limits.
   u32int base;                // The address of the first gdt_entry_t struct.
}
 __attribute__((packed));
```

Use function **static void init_gdt()** to set up values of the GDT entry(struct gdt entry_struct) calling function **gdt_set_gate(...)**

Finally we have the ASM function that will write and load the GDT pointer.

Like the GDT for the IDT we have more data structures

We use **struct idt_entry_struct** to store and describe an interrupt gate

And **struct idt_ptr_struct** to describe a pointer to an an array of interrupt handlers

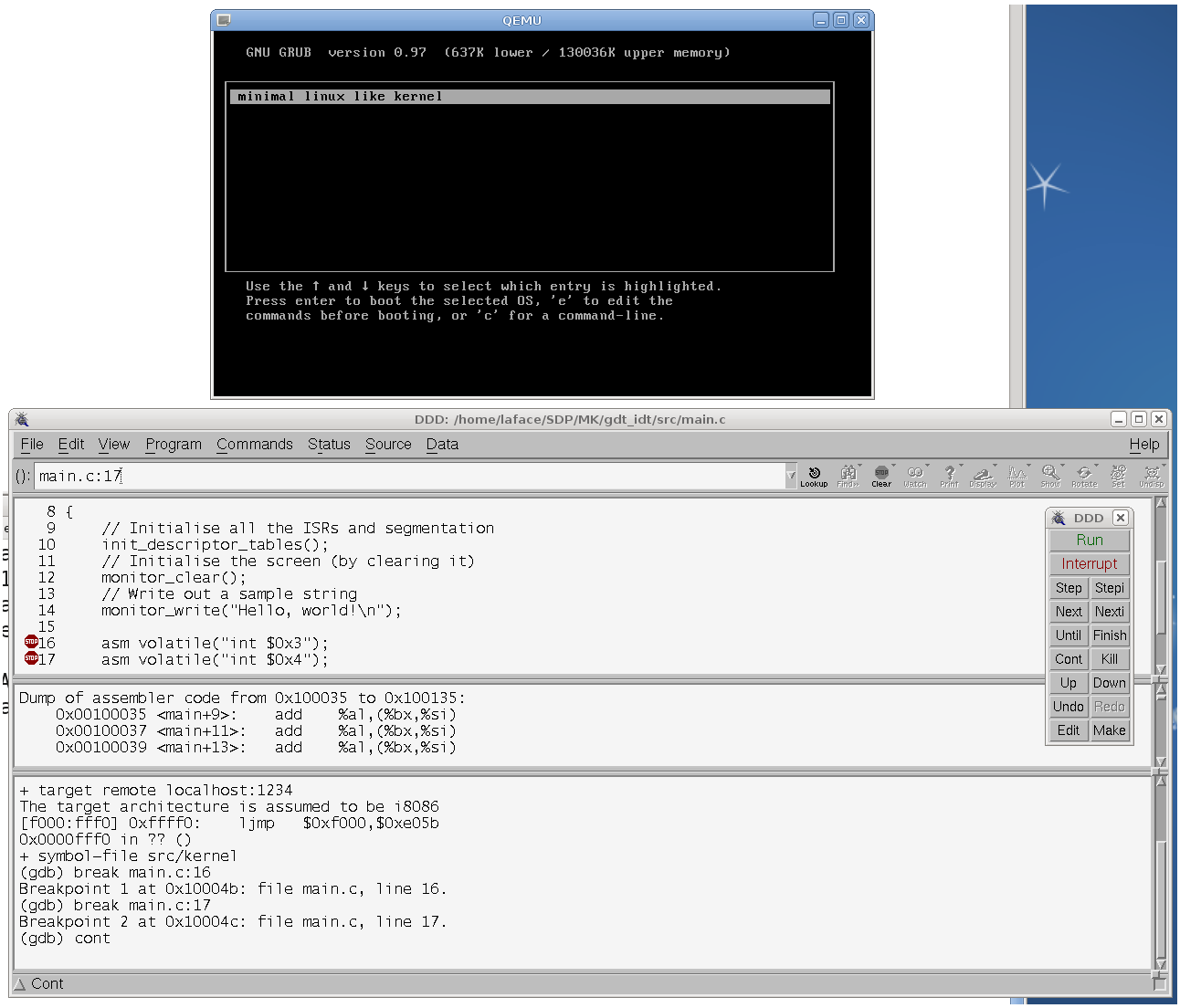
```
// A struct describing an interrupt gate.
struct idt_entry_struct
{
   u16int base_lo;             // The lower 16 bits of the address to jump to when
this interrupt fires.
   u16int sel;                 // Kernel segment selector.
   u8int  always0;             // This must always be zero.
   u8int  flags;               // More flags. See documentation.
   u16int base_hi;             // The upper 16 bits of the address to jump to.
} __attribute__((packed));
typedef struct idt_entry_struct idt_entry_t;

// A struct describing a pointer to an array of interrupt handlers.
// This is in a format suitable for giving to 'lidt'.
struct idt_ptr_struct
{
   u16int limit;
   u32int base;                // The address of the first element in our
idt_entry_t array.
} __attribute__((packed));
typedef struct idt_ptr_struct idt_ptr_t;
```

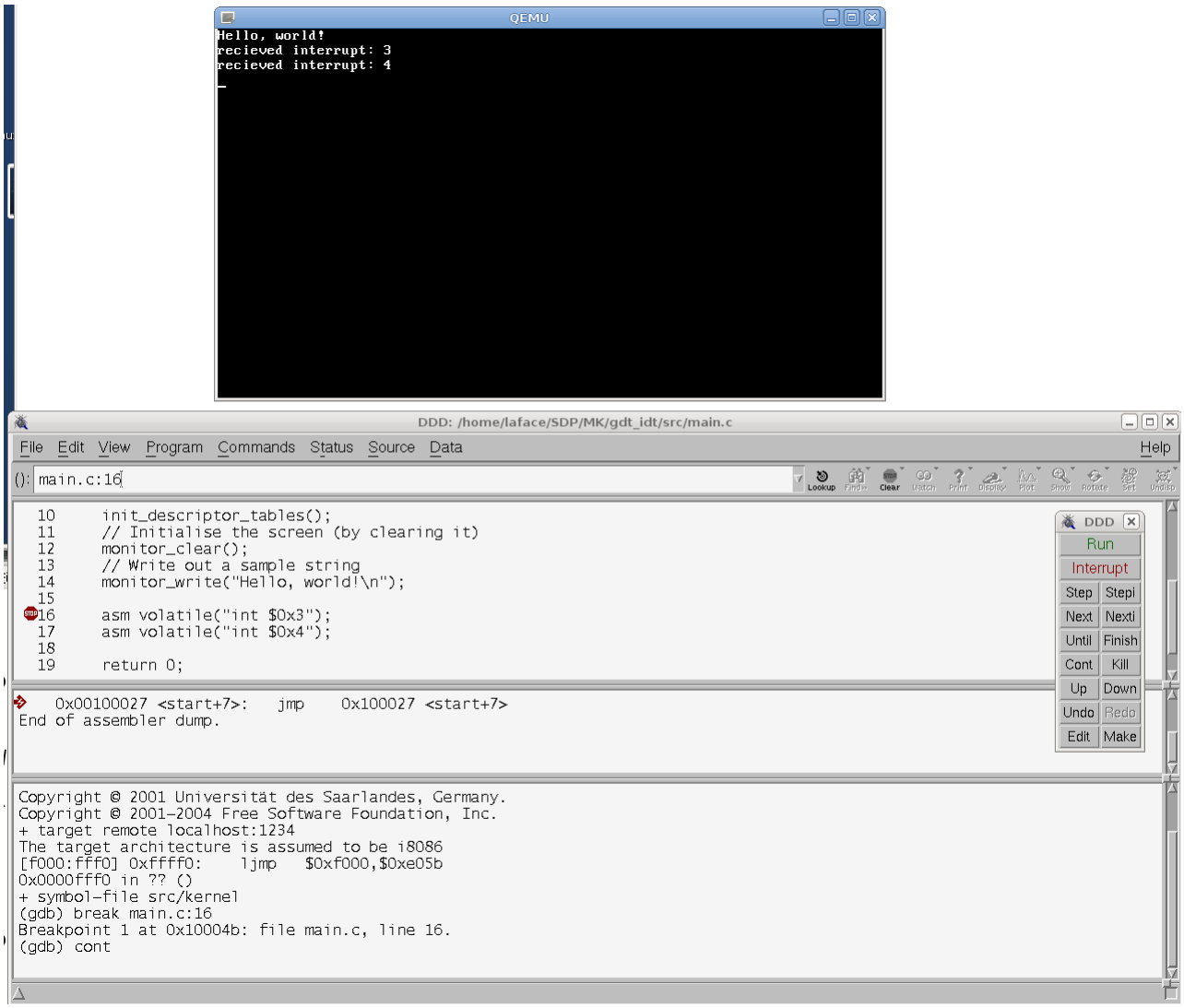
Calling the function **static void init_idt()** will be set up all the parameters of the data structures, so we initialize also our Interrupt Vector Table

```
[GLOBAL idt_flush]; Allows the C code to call idt_flush().

idt_flush:
   mov eax, [esp+4]; Get the pointer to the IDT, passed as a parameter.
   lidt[eax]; Load the IDT pointer.

   ret
```

Through this piece of code we get the pointer of the inizialized IDT and load IDT pointer

After that we can use NASM's macro to describe how manage the stack frame when an interrupt was called and than we can create the ASM handler function(Interrupt service routine)





Running ./qemu.sh in the gdt_idt directory and running in an other terminal the ddd command we can see the minimal kernel running. Setting the breakpoint at line 16 and 17 on the interrupt command(to test that kernel can handle interrupts and set up its own segmentation table) we can see the memory field that has been runned and than the main return.