POLITECNICO
DI TORINO

Lab 5 - SDP

Jacopo Nasi
Stud. ID: 255320
Politecnico di Torino

25 aprile 2018

1 Genesis

The following text is based on *nix environment. This report is written based on laboratory experience and the *http://www.jamesmolloy.co.uk* tutorial.

1.1 Boot Code

The main part of the boot code is written in C, the assembly part are in the INTEL form. The *boot.s* file is used to call the main of the C program, the other part are used for the multiboot header.
Multiboot is a standard to which GRUB expects a kernel to comply. This multiboot allows to inform the system which kernel must load at the system boot.
The first important command are:

```
1  push EBX
2  cli
3  call main
4  jump $
```

The first line push the content of EBX, the pointer to the multiboot part, to the stack, disable interrupts (CLI), call a 'main' C function, then enter an infinite loop. Since the interface between a C function and the assembly is made by pushing the parameters in the stack, the line 1 is used to passing a paramter to the function main(). The result of the function instead will be stored in the EAX register.
The main code is really simple:

```
1  /* main.c -- Defines the C-code kernel entry point,
2  calls initialisation routines.
3  Made for JamesM's tutorials */
4
5  int main(struct multiboot *mboot_ptr)
6  {
7    // All our initialisation calls will go in here.
8    return 0xDEADBABA;
9  }
```

2 GDT and IDT

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).

2.1 The GDT

The x86 architecture has two methods of memory protection and of providing virtual memory - segmentation and paging.

With segmentation, every memory access is evaluated with respect to a segment. That is, the memory address is added to the segment's base address, and checked against the segment's length. You can think of a segment as a window into the address space - The process does not know it's a window, all it sees is a linear address space starting at zero and going up to the segment length.
With paging, the address space is split into (usually 4KB, but this can change) blocks, called pages. Each page can be mapped into physical memory - mapped onto what is called a 'frame'. Or, it can be unmapped. Like this you can create virtual memory spaces.
Both of these methods have their advantages, but paging is much better. Segmentation is, although still usable, fast becoming obsolete as a method of memory protection and virtual memory. In fact, the x86-64 architecture requires a flat memory model (one segment with a base of 0 and a limit of 0xFFFFFFFF) for some of it's instructions to operate properly.

The structure of the GDT is the following:

```
1  // This structure contains the value of one GDT entry.
2  // We use the attribute 'packed' to tell GCC not to change
3  // any of the alignment in the structure.
4  struct gdt_entry_struct
5  {
6      u16int limit_low;           // The lower 16 bits of the limit
       .
7      u16int base_low;            // The lower 16 bits of the base.
8      u8int  base_middle;         // The next 8 bits of the base.
9      u8int  access;              // Access flags, determine what
       ring this segment can be used in.
10     u8int  granularity;
11     u8int  base_high;           // The last 8 bits of the base.
12 } __attribute__((packed));
13 typedef struct gdt_entry_struct gdt_entry_t;
```

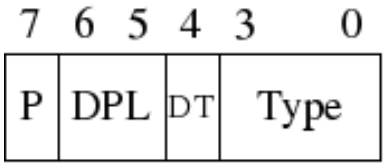

Figura 1: Access byte format GDT

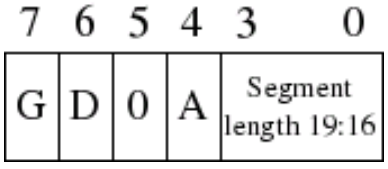

Figura 2: Granularity byte format GDT

The byte are:

- **P**: Is segment present? (1 = Yes)
- **DPL**:Descriptor privilege level - Ring 0 - 3.
- **DT**:Descriptor type
- **Type**:Segment type - code segment / data segment.
- **G**:Granularity (0 = 1 byte, 1 = 1kbyte)
- **D**:Operand size (0 = 16bit, 1 = 32bit)
- **0**:Should always be zero.
- **A**:Available for system use (always zero).

2.2 Utilization

The following line report some of the function used over the defined structure:

```
1  // Initialisation routine - zeroes all the interrupt service
     routines,
2  // initialises the GDT and IDT.
3  void init_descriptor_tables()
4  {
5   // Initialise the global descriptor table.
6   init_gdt();
7  }
8
9  static void init_gdt()
10 {
11  gdt_ptr.limit = (sizeof(gdt_entry_t) * 5) - 1;
12  gdt_ptr.base  = (u32int)&gdt_entries;
13
14  gdt_set_gate(0, 0, 0, 0, 0);                // Null segment
15  gdt_set_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // Code segment
16  gdt_set_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // Data segment
17  gdt_set_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // User mode code
     segment
18  gdt_set_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // User mode data
     segment
19
20  gdt_flush((u32int)&gdt_ptr);
21 }
22
23 // Set the value of one GDT entry.
24 static void gdt_set_gate(s32int num, u32int base, u32int limit,
     u8int access, u8int gran)
25 {
26  gdt_entries[num].base_low    = (base & 0xFFFF);
27  gdt_entries[num].base_middle = (base >> 16) & 0xFF;
```

```
28    gdt_entries[num].base_high    = (base >> 24) & 0xFF;
29
30    gdt_entries[num].limit_low    = (limit & 0xFFFF);
31    gdt_entries[num].granularity = (limit >> 16) & 0x0F;
32
33    gdt_entries[num].granularity |= gran & 0xF0;
34    gdt_entries[num].access       = access;
35  }
```

2.3 The IDT

There are times when you want to interrupt the processor. You want to stop it doing what it is doing, and force it to do something different. An example of this is when an timer or keyboard interrupt request (IRQ) fires. An interrupt is like a POSIX signal - it tells you that something of interest has happened. The processor can register 'signal handlers' (interrupt handlers) that deal with the interrupt, then return to the code that was running before it fired. Interrupts can be fired externally, via IRQs, or internally, via the 'int n' instruction. There are very useful reasons for wanting to do fire interrupts from software, but that's for another chapter!

The Interrupt Descriptor Table tells the processor where to find handlers for each interrupt. It is very similar to the GDT. It is just an array of entries, each one corresponding to an interrupt number. There are 256 possible interrupt numbers, so 256 must be defined. If an interrupt occurs and there is no entry for it (even a NULL entry is fine), the processor will panic and reset.

2.4 Faults, traps and exceptions

The processor will sometimes need to signal your kernel. Something major may have happened, such as a divide-by-zero, or a page fault. To do this, it uses the first 32 interrupts. It is therefore doubly important that all of these are mapped and non-NULL - else the CPU will triple-fault and reset (bochs will panic with an 'unhandled exception' error).

The special, CPU-dedicated interrupts are shown below.

- 0 - Division by zero exception
- 1 - Debug exception
- 2 - Non maskable interrupt
- 3 - Breakpoint exception
- 4 - 'Into detected overflow'
- 5 - Out of bounds exception
- 6 - Invalid opcode exception
- 7 - No coprocessor exception
- 8 - Double fault (pushes an error code)
- 9 - Coprocessor segment overrun
- 10 - Bad TSS (pushes an error code)
- 11 - Segment not present (pushes an error code)

- 12 - Stack fault (pushes an error code)
- 13 - General protection fault (pushes an error code)
- 14 - Page fault (pushes an error code)
- 15 - Unknown interrupt exception
- 16 - Coprocessor fault
- 17 - Alignment check exception
- 18 - Machine check exception
- 19-31 - Reserved

3 IRQs and the PIT

3.1 Interrupt requests

There are several methods for communicating with external devices. Two of the most useful and popular are polling and interrupting.

- Polling: Spin in a loop, occasionally checking if the device is ready.
- Interrupts: Do lots of useful stuff. When the device is ready it will cause a CPU interrupt, causing your handler to be run.

As can probably be gleaned from my biased descriptions, interrupting is considered better for many situations. Polling has lots of uses - some CPUs may not have an interrupt mechanism, or you may have many devices, or maybe you just need to check so infrequently that it's not worth the hassle of interrupts. Any rate, interrupts are a very useful method of hardware communication. They are used by the keyboard when keys are pressed, and also by the programmable interval timer (PIT).
The low-level concepts behind external interrupts are not very complex. All devices that are interrupt-capable have a line connecting them to the PIC (programmable interrupt controller). The PIC is the only device that is directly connected to the CPU's interrupt pin. It is used as a multiplexer, and has the ability to prioritise between interrupting devices. It is, essentially, a glorified 8-1 multiplexer. At some point, someone somewhere realised that 8 IRQ lines just wasn't enough, and they daisy-chained another 8-1 PIC beside the original. So in all modern PCs, you have 2 PICs, the master and the slave, serving a total of 15 interruptable devices (one line is used to signal the slave PIC).

3.2 Programmable Interval Timer

The programmable interval timer is a chip connected to IRQ0. It can interrupt the CPU at a user-defined rate (between 18.2Hz and 1.1931 MHz). The PIT is the primary method used for implementing a system clock and the only method available for implementing multitasking (switch processes on interrupt).

The PIT has an internal clock which oscillates at approximately 1.1931MHz. This clock signal is fed through a frequency divider, to modulate the final output frequency. It has 3 channels, each with it's own frequency divider.

- Channel 0: is the most useful. It's output is connected to IRQ0.
- Channel 1: is very un-useful and on modern hardware is no longer implemented. It used to control refresh rates for DRAM.
- Channel 2: controls the PC speaker.

The timer code is the following:

```
// timer.c -- Initialises the PIT, and handles clock updates.
// Written for JamesM's kernel development tutorials.

#include "timer.h"
#include "isr.h"
#include "monitor.h"

u32int tick = 0;

static void timer_callback(registers_t regs)
{
   tick++;
   monitor_write("Tick: ");
   monitor_write_dec(tick);
   monitor_write("\n");
}

void init_timer(u32int frequency)
{
   // Firstly, register our timer callback.
   register_interrupt_handler(IRQ0, &timer_callback);

   // The value we send to the PIT is the value to divide it's input clock
   // (1193180 Hz) by, to get our required frequency. Important to note is
   // that the divisor must be small enough to fit into 16-bits.
   u32int divisor = 1193180 / frequency;

   // Send the command byte.
   outb(0x43, 0x36);

   // Divisor has to be sent byte-wise, so split here into upper/lower bytes.
   u8int l = (u8int)(divisor & 0xFF);
   u8int h = (u8int)( (divisor>>8) & 0xFF );

   // Send the frequency divisor.
   outb(0x40, l);
   outb(0x40, h);
}
```

4 Paging

4.1 Paging as a concretion of virtual memory

Virtual memory is an abstract principle. As such it requires concretion through some system/algorithm. Both segmentation and paging are valid methods for implementing virtual memory. As mentioned before however, segmentation is becoming obsolete. Paging is the newer, better alternative for the x86 architecture.
Paging works by splitting the virtual address space into blocks called pages, which are usually 4KB in size. Pages can then be mapped on to frames - equally sized blocks of physical memory.

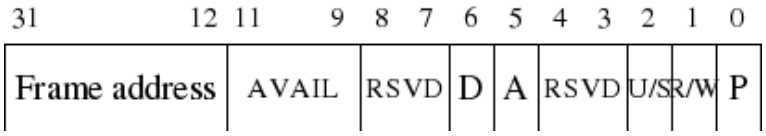

Figura 3: Page Table entry format

Each process normally has a different set of page mappings, so that virtual memory spaces are independent of each other. In the x86 architecture (32-bit) pages are fixed at 4KB in size. Each page has a corresponding descriptor word, which tells the processor which frame it is mapped to. Note that because pages and frames must be aligned on 4KB boundaries (4KB being 0x1000 bytes), the least significant 12 bits of the 32-bit word are always zero. The architecture takes advantage of this by using them to store information about the page, such as whether it is present, whether it is kernel-mode or user-mode etc. The layout of this word is in the picture on the right.

The fields in that picture are pretty simple, so let's quickly go through them.

- **P**: Set if the page is present in memory.
- **R/W**: If set, that page is writeable. If unset, the page is read-only. This does not apply when code is running in kernel-mode (unless a flag in CR0 is set).
- **U/S**: If set, this is a user-mode page. Else it is a supervisor (kernel)-mode page. User-mode code cannot write to or read from kernel-mode pages.
- **Reserved**: These are used by the CPU internally and cannot be trampled.
- **A**: Set if the page has been accessed (Gets set by the CPU).
- **D**: Set if the page has been written to (dirty).

- **AVAIL**: These 3 bits are unused and available for kernel-use.
- **Page frame address**: The high 20 bits of the frame address in physical memory.

4.2 Page Faults

When a process does something the memory-management unit doesn't like, a page fault interrupt is thrown. Situations that can cause this are (not complete):

- Reading from or writing to an area of memory that is not mapped (page entry/table's 'present' flag is not set)
- The process is in user-mode and tries to write to a read-only page.
- The process is in user-mode and tries to access a kernel-only page.
- The page table entry is corrupted - the reserved bits have been overwritten.

The page fault interrupt is number 14, and looking at chapter 3 we can see that this throws an error code. This error code gives us quite a bit of information about what happened.

- **Bit 0**: If set, the fault was not because the page wasn't present. If unset, the page wasn't present.
- **Bit 1**: If set, the operation that caused the fault was a write, else it was a read.
- **Bit 2**: If set, the processor was running in user-mode when it was interrupted. Else, it was running in kernel-mode.
- **Bit 3**: If set, the fault was caused by reserved bits being overwritten.
- **Bit 4**: If set, the fault occurred during an instruction fetch.

The processor also gives us another piece of information - the address that caused the fault. This is located in the CR2 register. Beware that if your page fault hander itself causes another page fault exception this register will be overwritten - so save it early!

5 The VFS and the initrd

5.1 The virtual filesystem

A VFS is intended to abstract away details of the filesystem and location that files are stored, and to give access to them in a uniform manner. They are

usually implemented as a graph of nodes; Each node representing either a file, directory, symbolic link, device, socket or pipe. Each node should know what filesystem it belongs to and have enough information such that the relavent open/close/etc functions in its driver can be found and executed. A common way to accomplish this is to have the node store function pointers which can be called by the kernel. We'll need a few function pointers:

- **Open** - Called when a node is opened as a file descriptor.
- **Close** - Called when the node is closed.
- **Read**
- **Write**
- **Readdir** - If the current node is a directory, we need a way of enumerating it's contents. Readdir should return the n'th child node of a directory or NULL otherwise. It returns a 'struct dirent', which is compatible with the UNIX readdir function.
- **Finddir** - We also need a way of finding a child node, given a name in string form. This will be used when following absolute pathnames.

So far then our node structure looks something like:

```
1  typedef struct fs_node
2  {
3     char name[128];     // The filename.
4     u32int mask;        // The permissions mask.
5     u32int uid;         // The owning user.
6     u32int gid;         // The owning group.
7     u32int flags;       // Includes the node type. See #defines
    above.
8     u32int inode;       // This is device-specific - provides a
    way for a filesystem to identify files.
9     u32int length;      // Size of the file, in bytes.
10    u32int impl;        // An implementation-defined number.
11    read_type_t read;
12    write_type_t write;
13    open_type_t open;
14    close_type_t close;
15    readdir_type_t readdir;
16    finddir_type_t finddir;
17    struct fs_node *ptr; // Used by mountpoints and symlinks.
18 } fs_node_t;
```

5.2 Mountpoints

Mountpoints are the UNIX way of accessing different filesystems. A filesystem is mounted on a directory - any subsequent access to that directory will actually access the root directory of the new filesystem. So essentially the directory is told that it is a mountpoint and given a pointer to the root node

of the new filesystem. We can actually reuse the ptr member of fs_node_t for this purpose (as it is currently only used for symlinks and they can never be mountpoints).

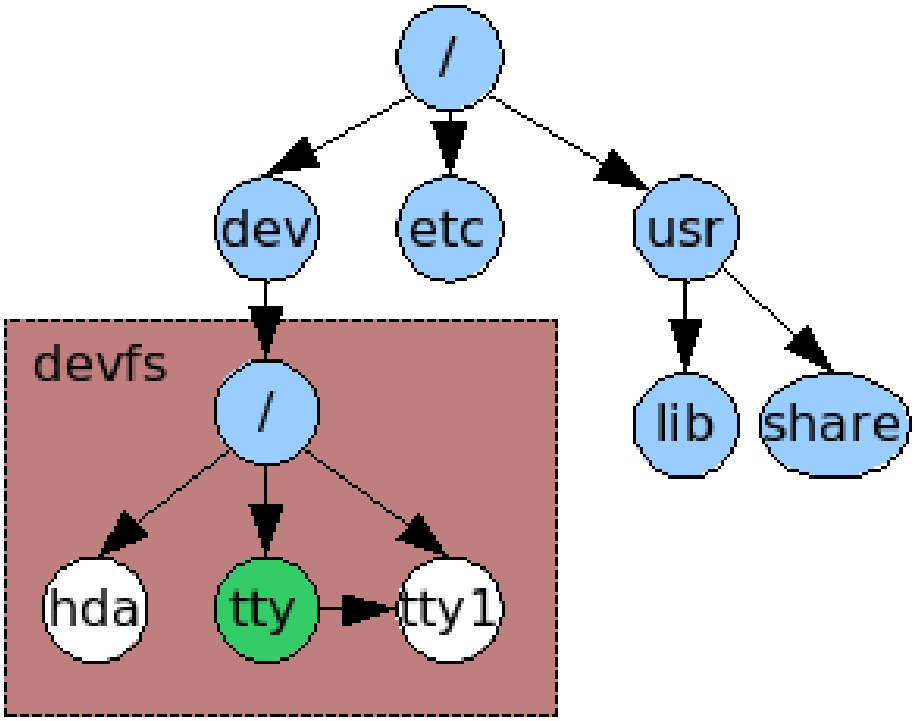

Figura 4: devfs mounted on /dev

5.3 The initial ramdisk

An initial ramdisk is just a filesystem that is loaded into memory when the kernel boots. It is useful for storing drivers and configuration files that are needed before the kernel can access the root filesystem (indeed, it usually contains the driver to access that root filesystem!).
An initrd, as they are known, usually uses a propriatary filesystem format. The reason for this is that the most complex thing a filesystem has to handle, deletion of files and reclaimation of space, isn't necessary. The kernel should try to get the root filesystem up and running as quick as possible. The initrd must load as a multiboot module.

6 User mode

Kernel mode makes available certain instructions that would usually be denied a user program - like being able to disable interrupts, or halt the processor.
Once you start running user programs, you'll want to make the jump from kernel mode to user mode, to restrict what instructions are available. You can

also restrict read or write access to areas of memory. This is often used to 'hide' the kernel's code and data from user programs.

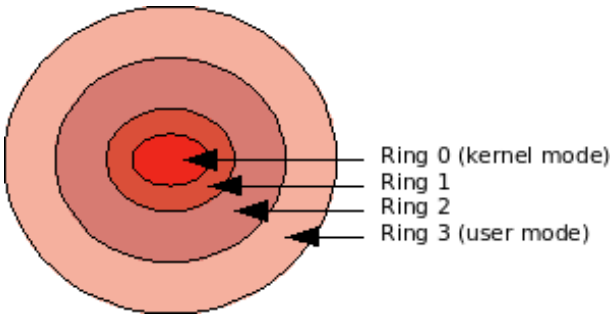

Figura 5: devfs mounted on /dev

The x86 is strange in that there is no direct way to switch to user mode. The only way one can reach user mode is to return from an exception that began in user mode. The only method of getting there in the first place is to set up the stack as if an exception in user mode had occurred, then executing an exception return instruction (IRET).
The IRET instruction expects, when executed, the stack to have the following contents (starting from the stack pointer - the lowermost address upwards):

- The instruction to continue execution at - the value of EIP.
- The code segment selector to change to.
- The value of the EFLAGS register to load.
- The stack pointer to load.
- The stack segment selector to change to.

The EIP, EFLAGS and ESP register values should be easy to work out, but the CS and SS values are slightly more difficult.
When we set up our GDT we set up 5 selectors - the NULL selector, a code segment selector for kernel mode, a data segment selector for kernel mode, a code segment selector for user mode, and a data segment selector for user mode.

They are all 8 bytes in size, so the selector indices are:

- **0x00**: Null descriptor
- **0x08**: Kernel code segment
- **0x10**: Kernel data segment
- **0x18**: User code segment
- **0x20**: User data segment

We're currently using selectors 0x08 and 0x10 - for user mode we want to use selectors 0x18 and 0x20. However, it's not quite that straightforward. Because the selectors are all 8 bytes in size, the two least significant bits of the selector will always be zero. Intel use these two bits to represent the RPL - the Requested Privilege Level. These have currently been zero because we were operating in ring 0, but now that we want to move to ring three we must set them to '3'. If you wish to know more about the RPL and segmentation in general, you should read the intel manuals. There is far too much information for me to explain everything here.

The following snippet is to switch beetween the different mode:

```
void switch_to_user_mode()
{
   // Set up a stack structure for switching to user mode.
   asm volatile("  \
     cli; \
     mov $0x23, %ax; \
     mov %ax, %ds; \
     mov %ax, %es; \
     mov %ax, %fs; \
     mov %ax, %gs; \
                   \
     mov %esp, %eax; \
     pushl $0x23; \
     pushl %eax; \
     pushf; \
     pushl $0x1B; \
     push $1f; \
     iret; \
   1: \
     ");
}
```