A report on
Minimalistic Kernel development in C

Prepared by : Amit Tomar

Team

amit.tomar
(MT2013008)

ashutosh.vyas
(MT2013031)

pankajkumar.aggarwal
(MT2013096)

rakesh.rajpurohit
(MT2013122)

@iiitb.org

20 - Dec - 2013 *

*A lot of the material in this document has been taken from several resources available over the internet. It was difficult to give exact citations, thus we have given topic wise references at the end of this document. Permission through e-mail was taken from two authors whose work appears majorly in this document. This document shall be treated more as a collection of tutorial than as an original work of the authors.

1 Problem Statement

This project aims at developing an absolute minimal Kernel, to understand the basic functionality of how Kernels are developed, build, linked and loaded into the memory. We have explored how drivers are written by writing minimal driver for video display manipulation. We have further explored how interrupts are handled and developed a programmable interval timer based on it. Overall work involved was as follows :

1. Development of a Kernel entry point in ASM.
2. Development of a bare minimal custom Kernel in C language.
3. Understanding the build process of this Kernel.
4. Linking this kernel with the startup script.
5. Using GRUB, booting up and loading of this Kernel on a CPU emulator/virtual machine.
6. Write a display driver which is capable of performing the following functionality :
   (a) Writing characters on the screen.
   (b) Clearing screen.
   (c) Scrolling the screen.
   (d) Moving the cursor.
7. Handling of the various interrupts generated in the system by calling the respective Interrupt Service Routines.
8. Development of a programmable interval timer which can be asked to generate timer interrupts at the user defined intervals.

2 The boot process

To boot up an Operating System, another piece of software is required which can 'boot strap' itself and then load the Operating System, further passing control to it. This is called the bootloader. We have used the Bootloader known as GRUB (a Multiboot compliant boot loader) . It puts the system in to the correct state for your kernel to start executing. This includes

enabling the A20 line (to give you access to all available memory addresses) and putting the system in to 32-bit Protected Mode. We have used a kernel executable in ELF format, so that we have control to tell GRUB where to load which part in memory.

Booting the Operating System

The first task we will deal with is how the bootloader starts the kernel. Multiboot Standard for Kernels describes an easy interface between the bootloader and the operating system kernel. It works by putting a few magic values in some global variables (known as a multiboot header), which is searched for by the bootloader. When it sees these values, it recognizes the kernel as multiboot compatible and it knows how to load us, and it can even forward us important information such as memory maps etc to the kernel.

Master Boot Record

A device is said to be "bootable" if it carries a boot sector with the byte sequence 0x55, 0xAA in bytes 511 and 512 respectively. When the BIOS finds such a boot sector, it is loaded into memory at a specific location; this is usually 0x0000:0x7c00 (segment 0, address 0x7c00).

Linking the Kernel

We have to assemble the boot assembly file and compile the main kernel c file. This produces two object files that each contain part of the kernel. To create the full and final kernel we will have to link these object files into the final kernel program, usable by the bootloader.

3 Handling Interrupts

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing, the current thread. The processor responds by suspending its current activities, saving its state, and executing a small program called an interrupt handler (or interrupt service routine, ISR) to deal with the event. This interruption is temporary, and after the interrupt handler finishes, the processor resumes execution of the previous thread.

The Interrupt Descriptor Table

The IDT is a descriptor table which provides information about several interrupt and how they are to be handled. Interrupt means stoping the processor from what it is doing currently, and pass on the control to some other entity, and then return back to the original execution on completion of the generated Interrupt. eg. Timer, Keyboard interrupt requests. The processor return to the code that was running before it stopped to handle an interrupt. Interrupts can be fired externally, via IRQs, or internally, via the 'int n' instruction.

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

Faults, traps and exceptions

An Exception is a special case that the processor encounters when it cannot continue normal execution. This could be something like dividing by zero: The result is an unknown or non-real number, so the processor will cause an exception so that the kernel can stop that process or task from causing any problems. If the processor finds that a program is trying to access a piece of memory that it shouldn't, it will cause a General Protection Fault. For this purpose the first 32 interrupts aew reserved. If a mapping for these is not privided CPU will triple-fault and reset. When the processor receives an interrupt, it saves the contents of the essential registers (instruction pointer, stack pointer, code and data segments, flags register) to the stack. It then finds the interrupt handler location from our IDT and jumps to it.

Following are the special interrupt for CPU

1. Division by zero exception
2. Debug exception
3. Non maskable interrupt
4. Breakpoint exception
5. Into detected overflow
6. Out of bounds exception

7. Invalid opcode exception
8. No coprocessor exception
9. Double fault (pushes an error code)
10. Coprocessor segment overrun
11. Bad TSS (pushes an error code)
12. Segment not present (pushes an error code)
13. Stack fault (pushes an error code)
14. General protection fault (pushes an error code)
15. Page fault (pushes an error code)
16. Unknown interrupt exception
17. Coprocessor fault
18. Alignment check exception
19. Machine check exception
20. 19-31 - Reserved

Interrupt Service Routines

Interrupt Service Routines, or ISRs, are used to save the current processor state and set up the appropriate segment registers needed for kernel mode before the kernel's C-level interrupt handler is called. The first 32 entries in the IDT correspond to Exceptions that can possibly be generated by the processor, and therefore need to be handled. Some exceptions will push another value onto the stack: an Error Code value which is specific to the exception caused.

4 The Screen

Monitors can display both text and graphics and have different techniques and memory requirements for each. Consequently, video adapters have two display modes: text and graphics. In text mode, the screen is divided into columns and rows, typically 80 columns by 25 rows, and a character is

displayed at each screen position (character cell). In graphics mode, the screen is again divided into columns and rows, and each screen position is called a pixel (short for picture element). A picture can be displayed by specifying the color of each pixel on the screen.

Kernel gets booted by GRUB in text mode. That is, it has available to it a framebuffer that controls a screen of characters (**not pixels**) 80 wide by 25 high. This area of memory known as the framebuffer is accessible just like normal RAM, at address 0xB8000. It is important to note, however, that it is not actually normal RAM.

Text Mode Programming

Positions on the screen are referenced using (column, row) coordinates. The upper left corner has coordinates (0,0). For an 80 x 25 display, the rows are 0-24 and the columns are 0-79.

Table of some 80 x 25 screen positions

Position	Decimal	Hexadecimal
Upper Left Corner	(0,0)	(0,0)
Lower Left Corner	(0,24)	(0,18)
Upper Left Corner	(79,0)	(4F,0)
Lower Left Corner	(79,24)	(4F,18)
Center Screen	(39,12)	(27,C)

The character displayed at a screen position is specified by the contents of a WORD in the display memory. The low byte of the word contains the character's ASCII code; the high byte contains its attribute, which tells how the character will be displayed (its color, whether it's blinking, underlined, and so on).

The Attribute Byte

The high byte of the word that specifies a display character is called the attribute byte. It describes the color and intensity of the character, the background color, and whether the character is blinking and/or underlined.

16-Color Display Attribute Byte:

Bit# 7 6 5 4 3 2 1 0

Attr BL R G B IN R G B

Attributes:

Bit # Attribute
0-2 character color (foreground color)
3 intensity
4-6 background color
7 blinking

E.g., to display a red character on a blue background, the attribute byte would be: 0001 0100 = 14h

If the attribute byte is: 0011 0101 = 35h, it uses blue + green (cyan) in the background and red+blue (magenta) in the foreground, so the character displayed would be magenta on a cyan background. If the intensity bit (bit 3) is 1, the foreground color is lightened (brightened). If the blinking bit (bit 7) is 1, the character turns on and off.

5 The Keyboard

A scancode is simply a key number. The keyboard assigns a number to each key on the keyboard; this is your scancode. The scancodes are numbered generally from top to bottom and left to right, with some minor exceptions to keep layouts backwards compatible with older keyboards. You must use a lookup table (an array of values) and use the scancode as the index into this table. The lookup table is called a keymap, and will be used to translate scancodes into ASCII values rather quickly and painlessly. If bit 7 is set (test with 'scancode & 0x80'), then this is the keyboard's way of telling us that a key was just released.

The keyboard is attached to the computer through a special microcontroller chip on your mainboard. This keyboard controller chip has 2 channels: one for the keyboard, and one for the mouse. Also note that it is through this keyboard controller chip that you would enable the A20 address line on the processor to allow you to access memory past the 1MByte mark (GRUB enables this, you don't need to worry about it). The keyboard controller, being a device accessible by the system, has an address on the I/O bus that we can use for access and control. The keyboard controller has 2 main registers: a Data register at 0x60, and a Control register at 0x64. Anything that the keyboard wants to send the computer is stored into the Data register. The keyboard will raise IRQ1 whenever it has data for us to read. Observe:

6 Interrupt requests

Interrupt Requests or IRQs are interrupts that are raised by hardware devices. Some devices generate an IRQ when they have data ready to be read, or when they finish a command like writing a buffer to disk, for example. It's safe to say that a device will generate an IRQ whenever it wants the processor's attention.

Normally PCs have 2 chips that are used to manage IRQs. These 2 chips are known as the Programmable Interrupt Controllers or PICs. One acts as the 'Master' IRQ controller, and one is the 'Slave' IRQ controller. The slave is connected to IRQ2 on the master controller. The master IRQ controller is connected directly to the processor itself, to send signals. Each PIC can handle 8 IRQs. The master PIC handles IRQs 0 to 7, and the slave PIC handles IRQs 8 to 15. Remember that the slave controller is connected to the primary controller through IRQ2: This means that every time an IRQ from 8 to 15 occurs, IRQ2 fires at exactly the same time.

When a device signals an IRQ, an interrupt is generated, and the CPU pauses whatever it's doing to call the ISR to handle the corresponding IRQ. The CPU then performs whatever necessary action (like reading from the keyboard, for example), and then it must tell the PIC that the interrupt came from that the CPU has finished executing the correct routine. The CPU tells the right PIC that the interrupt is complete by writing the command byte 0x20 in hex to the command register for that PIC. The master PIC's command register exists at I/O port 0x20, while the slave PIC's command register exists at I/O port 0xA0.

IRQ0 to IRQ7 are originally mapped to IDT entries 8 through 15. IRQ8 to IRQ15 are mapped to IDT entries 0x70 through 0x78. IDT entries 0 through 31 are reserved for exceptions. Interrupt Controllers are however 'programmable': You can change what IDT entries that their IRQs are mapped to. For this project, we have mapped IRQ0 through IRQ15 to IDT entries 32 through 47.

7 The programmable Interval timer

The Programmable Interval Timer also called the System Clock, is a very useful chip for accurately generating interrupts at regular time intervals. The chip itself has 3 channels: Channel 0 is tied to is tied to IRQ0, to interrupt the CPU at predictable and regular times, Channel 1 is system

specific, and Channel 2 is connected to the system speaker. By default, channel 0 of the timer is set to generate an IRQ0 18.222 times per second.

To set the rate at which channel 0 of the timer fires off an IRQ0, we must use our outportb function to write to I/O ports. There is a Data register for each of the timer's 3 channels at 0x40, 0x41, and 0x42 respectively, and a Command register at 0x43. The data rate is actually a 'divisor' register for this device. The timer will divide it's input clock of 1.19MHz (1193180Hz) by the number you give it in the data register to figure out how many times per second to fire the signal for that channel. You must first select the channel that we want to update using the command register before writing to the data/divisor register.

8 Installing BOCHS

Go to the official sourceforge page of BOCHS and download the latest rpm package available there :

```
http://sourceforge.net/projects/bochs/files/bochs/
```

Ubuntu distro doesnt support rpm package installation directly, you can use the utility called alien to convert it into deb package :

```
# sudo apt-get install alien
# sudo alien -k your_rpm_package.rpm
```

Then right click and open the obtained deb package using the software centre provided by ubuntu to install the package.

9 Limitations

1. All the keys on keyboard are not handled. Keyboard lights, right hand side keys etc are not handled.

2. When an exception is raised, system halts completely. Not able to recover from the exception.

10 References

Following sources were referred before writing this documents to get the basic understanding of Kernel development process and approximate time it might require:

General references about operating systems and LaTeX

1. http://wiki.osdev.org/Bare_Bones
2. http://wiki.osdev.org/Category:Babystep
3. http://www.osdever.net/bkerndev/index.php
4. http://www.osdever.net/tutorials/pdf/getting_started.pdf
5. http://www.osdever.net/tutorials/pdf/comparison.pdf
6. http://www.jamesmolloy.co.uk/index.html
7. http://www.nondot.org/sabre/os/articles
8. http://www.wikihow.com/Make-a-Computer-Operating-System
9. http://forums.devshed.com/c-programming-42/i-ve-written-a-small-os-kernel-and-i-have-388986.html
10. http://gusc.lv/2012/11/im-writing-my-own-os/
11. http://www.brokenthorn.com/Resources/OSDev2.html
12. http://joelgompert.com/OS/TableOfContents.htm
13. http://stackoverflow.com/questions/2195908/latex-multiple-linebreaks
14. http://techfreaks4u.com/blog/posts/kernel-development-from-scratch/
15. http://wiki.osdev.org/Tutorials
16. http://tex.stackexchange.com/questions/81834/whats-the-best-way-to-typeset-c-codes-in-latex
17. http://stackoverflow.com/questions/741985/latex-source-code-listing-like-in-professional-books
18. http://www.personal.ceu.hu/tex/breaking.htm

19. http://en.wikibooks.org/wiki/LaTeX/Source-Code-Listings

20. http://tex.stackexchange.com/questions/38829/remove-paragraph-indentation

References about the operating system boot process

21. http://www.cs.rutgers.edu/ pxk/416/notes/02-boot.html

22. http://en.wikipedia.org/wiki/Booting

23. http://www.tldp.org/LDP/intro-linux/html/sect_04_02.html

24. http://www.thegeekstuff.com/2011/02/linux-boot-process/

25. http://sansbound.com/2012/08/understanding-the-boot-process-charles-joseph/

26. http://kevinboone.net/boot.html

27. http://www.osdever.net/tutorials/pdf/bootsector_hd.pdf

28. http://www.osdever.net/tutorials/brunmar/tutorial_01.php

References about BOCHS

29. http://bochs.sourceforge.net/doc/docbook/user/bochsrc.html

30. http://bochs.sourceforge.net/

31. http://manpages.ubuntu.com/manpages/dapper/man5/bochsrc.5.html

32. http://en.wikipedia.org/wiki/Bochs

33. http://sourceforge.net/projects/bochs/files/bochs/2.6.2/

34. http://joeyh.name/code/alien/

References about Makefiles and linker scripts

35. http://www.cs.duke.edu/courses/cps108/doc/makefileinfo/sample.html

36. http://stackoverflow.com/questions/3220277/what-do-the-makefile-symbols-and-mean

37. http://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

38. https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html

39. http://www.delorie.com/gnu/docs/binutils/ld_6.html

Mount and loop devices

40. http://en.wikipedia.org/wiki/Loop_device

41. http://www.linuxcommand.org/man_pages/losetup8.html

Bootloaders, custom boot and GRUB in particular

42. http://wiki.osdev.org/Boot_Sequence

43. http://wiki.osdev.org/Diskless_Booting

44. http://www.osdever.net/tutorials/grub.php

45. http://wiki.osdev.org/PS2_Keyboard

46. http://www.gnu.org/software/grub/manual/grub.html#Embedded-data

47. http://www.linuxquestions.org/questions/linux-general-1/how-to-export-customized-linux-os-to-usb-bootable-pendrive-4175443832/

48. http://rudd-o.com/linux-and-free-software/a-better-way-to-create-a-customized-ubuntu-live-usb-drive

49. https://help.ubuntu.com/community/BootFromUSB

50. http://workingdirectory.net/posts/2009/grub-on-usb/

Writing to the Video device

51. http://wiki.osdev.org/Printing_to_Screen

52. http://www.osdev.org/howtos/2/#related

53. http://bytes.com/topic/c/answers/621381-plz-explain-outportb-function

54. http://wiki.osdev.org/Text_Mode_Cursor

55. http://wiki.osdev.org/Memory_Map_(x86)#BIO_Data_Area_.28BDA.29

56. http://www.codeproject.com/Articles/39069/Beginning-Operating-System-Development-Part-Two

57. http://fleder44.net/312/notes/18Graphics/index.html

58. http://en.wikipedia.org/wiki/Text_mode

Keyboard

59. http://www.brokenthorn.com/Resources/OSDev19.html

Miscellaneous topics

60. http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/bitshift.html

61. http://wiki.osdev.org/A20_Line

62. http://www.muppetlabs.com/ breadbox/software/tiny/teensy.html

63. http://stackoverflow.com/questions/2427011/what-is-the-difference-between-elf-files-and-bin-files

64. http://wiki.osdev.org/8259_PIC

65. http://forum.osdev.org/viewtopic.php?f=1&t=23473

66. http://stackoverflow.com/questions/4584089/assembler-push-pop-registers

67. http://wiki.answers.com/Q/What_are_segment_registers

68. http://reverseengineering.stackexchange.com/questions/2006/how-are-the-segment-registers-fs-gs-cs-ss-ds-es-used-in-i386-and-amd64

69. http://stackoverflow.com/questions/9580383/gcc-fno-stack-protector-option

70. http://stackoverflow.com/questions/16879434/why-do-these-c-struct-definitions-give-warnings-and-errors

71. http://www.eecg.toronto.edu/ amza/www.mindsec.com/files/x86regs.html

72. http://wiki.osdev.org/PIC

73. http://reverseengineering.stackexchange.com/questions/2006/how-are-the-segment-registers-fs-gs-cs-ss-ds-es-used-in-i386-and-amd64

74. http://stackoverflow.com/questions/10810203/what-is-the-fs-gs-register-intended-for

75. http://wiki.osdev.org/Stack

Interrupt Descriptor table

76. http://wiki.osdev.org/Interrupt_Descriptor_Table

Assembly

77. http://www.cs.virginia.edu/ evans/cs216/guides/x86.html

78. http://en.wikipedia.org/wiki/JMP_(x86_instruction)

79. http://tigcc.ticalc.org/doc/gnuasm.html

80. http://www.tldp.org/HOWTO/html_single/Assembly-HOWTO/

81. http://docs.cs.up.ac.za/programming/asm/derick_tut/

82. http://www.plantation-productions.com/Webster/

Intel i386 Reference Manuals

83. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf