**CSE222**

**OPERATING SYSTEMS**

**WINTER SEMESTER 2014-15 PROJECT:**

CREATION OF A KERNEL

Arman Mann 13BCE0073

Riya Daniel 13BCE0278

Anirudh Dahiya 13BCE0606

**PROJECT ID: C2P\_16**

**SLOT: C2**

**FACULTY: PROF. MANOOV R.**

**Project Abstract**

The project deals with the creation of a basic kernel from scratch, using C programming and assembly language programming as the primary tools. A kernel is the basic fundamental unit of an operating system, and the kernel which this project will focus on can be used to execute a few simple commands.

**Motivation**

A course on operating systems cannot be fully appreciated without exploring all aspects of an operating system, and most importantly, its roots. This brings one to the system kernel, and how it is coded and created to hold together the entire operating system. We therefore decided to look at a very basic kernel in order to learn how it works and how it is created.

**Project Description**

The project involves the creation of a simple kernel that can take input from the keyboard. The kernel can be loaded with a bootloader on an x86 system or run on an emulator such as QEMU. The kernel can be used for a rudimentary purpose like displaying a message, and then taking in a response from the user. This can then can be extended to other functions. The tools used in developing the project are:

* x86 computer
* VirtualBox
* Linux
* NASM assembler
* GCC Compiler
* LD (GNU Linker)
* QEMU Emulator

**Project Implementation**

**VirtualBox and Ubuntu**

To create our kernel we needed to install a VirtualBox on our laptop. VirtualBox allows us to run multiple operating systems on one device. Then we downloaded and installed Ubuntu 14.0.4 on Virtual Box. This was done to prevent the project kernel replacing the kernel on our own computer upon reboot. To install all the tools needed for the creation of our kernel we run the following commands on the Ubuntu terminal:

sudo apt-get update

sudo apt-get install qemu nasm build-essential

**Writing the kernel**

First we needed an assembly file that serves as the starting point for our kernel. This file then invokes an external function written in C. To make sure that the assembly code serves as the starting point we use a linker script that links the object files to produce the final kernel executable. In this linker script, we will explicitly specify that we want our binary to be loaded at the address [0x100000]. This address is where the kernel is expected to be. Thus, the bootloader/emulator will take care of firing the kernel’s entry point.

Here’s the assembly code:

;;kernel.asm

bits 32

section .text

;multiboot spec

align 4

dd 0x1BADB002 ;magic

dd 0x00 ;flags

dd - (0x1BADB002 + 0x00) ;checksum. m+f+c should be zero

global start

global keyboard\_handler

global read\_port

global write\_port

global load\_idt

extern kmain ;this is defined in the c file

extern keyboard\_handler\_main

read\_port:

mov edx, [esp + 4]

;al is the lower 8 bits of eax

in al, dx ;dx is the lower 16 bits of edx

ret

write\_port:

mov edx, [esp + 4]

mov al, [esp + 4 + 4]

out dx, al

ret

load\_idt:

mov edx, [esp + 4]

lidt [edx]

sti ;turn on interrupts

ret

keyboard\_handler:

call keyboard\_handler\_main

iretd

start:

cli ;block interrupts

mov esp, stack\_space

call kmain

hlt ;halt the CPU

section .bss

resb 8192; 8KB for stack

stack\_space:

**The assembly file**

The first instruction bits 32 is not an x86 assembly instruction. It’s a directive to the NASM assembler that specifies it should generate code to run on a processor operating in 32 bit mode.

The second line begins the text section. This is where we put all our code.

global is another NASM directive to set symbols from source code as global. By doing so, the linker knows where the symbol start is; which happens to be our entry point.

kmain is our function that will be defined in our kernel.c file. extern declares that the function is declared elsewhere.

We communicate with I/O devices using I/O ports. These ports are just specific address on the x86’s I/O bus, nothing more. The read/write operations from these ports are accomplished using specific instructions built into the processor.

**Reading from and Writing to ports**

I/O ports are accessed using the in and out instructions that are part of the x86 instruction set.

In read\_port, the port number is taken as argument. When compiler calls your function, it pushes all its arguments onto the stack. The argument is copied to the register edx using the stack pointer. The register dx is the lower 16 bits of edx. The in instruction here reads the port whose number is given by dx and puts the result in al. Register al is the lower 8 bits of eax. Function return values are received through the eax register. Thus read\_port lets us read I/O ports.

write\_port is very similar. Here we take 2 arguments: port number and the data to be written. The out instruction writes the data to the port.

read\_port:

mov edx, [esp + 4]

;al is the lower 8 bits of eax

in al, dx ;dx is the lower 16 bits of edx

ret

write\_port:

mov edx, [esp + 4]

mov al, [esp + 4 + 4]

out dx, al

ret

Then, we have the start function, which calls the kmain function and halts the CPU using the hlt instruction. Interrupts can awake the CPU from an hlt instruction. So we disable interrupts beforehand using cli instruction. cli is short for clear-interrupts.

We should ideally set aside some memory for the stack and point the stack pointer (esp) to it. So, we allocate some space in the BSS section and point the stack pointer to the beginning of the allocated memory. We use the resb instruction which reserves memory given in bytes. After it, a label is left which will point to the edge of the reserved piece of memory. Just before the kmain is called, the stack pointer (esp) is made to point to this space using the mov instruction.

**The C file**

In kernel.asm, we made a call to the function kmain(). So our C code will start executing at kmain():

// kernel.c

#define LINES 25

#define COLUMNS\_IN\_LINE 80

#define BYTES\_FOR\_EACH\_ELEMENT 2

#define SCREENSIZE BYTES\_FOR\_EACH\_ELEMENT \* COLUMNS\_IN\_LINE \* LINES

#define KEYBOARD\_DATA\_PORT 0x60

#define KEYBOARD\_STATUS\_PORT 0x64

#define IDT\_SIZE 256

#define INTERRUPT\_GATE 0x8e

#define KERNEL\_CODE\_SEGMENT\_OFFSET 0x08

#define ENTER\_KEY\_CODE 0x1C

extern unsigned char keyboard\_map[128];

extern void keyboard\_handler(void);

extern char read\_port(unsigned short port);

extern void write\_port(unsigned short port, unsigned char data);

extern void load\_idt(unsigned long \*idt\_ptr);

/\* current cursor location \*/

unsigned int current\_loc = 0;

/\* video memory begins at address 0xb8000 \*/

char \*vidptr = (char\*)0xb8000;

struct IDT\_entry{

unsigned short int offset\_lowerbits;

unsigned short int selector;

unsigned char zero;

unsigned char type\_attr;

unsigned short int offset\_higherbits;

};

struct IDT\_entry IDT[IDT\_SIZE];

void idt\_init(void)

{

unsigned long keyboard\_address;

unsigned long idt\_address;

unsigned long idt\_ptr[2];

/\* populate IDT entry of keyboard's interrupt \*/

keyboard\_address = (unsigned long)keyboard\_handler;

IDT[0x21].offset\_lowerbits = keyboard\_address & 0xffff;

IDT[0x21].selector = KERNEL\_CODE\_SEGMENT\_OFFSET;

IDT[0x21].zero = 0;

IDT[0x21].type\_attr = INTERRUPT\_GATE;

IDT[0x21].offset\_higherbits = (keyboard\_address & 0xffff0000) >> 16;

/\* Ports

\* PIC1 PIC2

\*Command 0x20 0xA0

\*Data 0x21 0xA1

ICW1 - begin initialization \*/

write\_port(0x20 , 0x11);

write\_port(0xA0 , 0x11);

/\* ICW2 - remap offset address of IDT

\* In x86 protected mode, we have to remap the PICs beyond 0x20 because

\* Intel have designated the first 32 interrupts as "reserved" for cpu exceptions

\*/

write\_port(0x21 , 0x20);

write\_port(0xA1 , 0x28);

/\* ICW3 - setup cascading \*/

write\_port(0x21 , 0x00);

write\_port(0xA1 , 0x00);

/\* ICW4 - environment info \*/

write\_port(0x21 , 0x01);

write\_port(0xA1 , 0x01);

/\* Initialization finished \*/

/\* mask interrupts \*/

write\_port(0x21 , 0xff);

write\_port(0xA1 , 0xff);

/\* fill the IDT descriptor \*/

idt\_address = (unsigned long)IDT ;

idt\_ptr[0] = (sizeof (struct IDT\_entry) \* IDT\_SIZE) + ((idt\_address & 0xffff) << 16);

idt\_ptr[1] = idt\_address >> 16 ;

load\_idt(idt\_ptr);

}

void kb\_init(void)

{

/\* 0xFD is 11111101 - enables only IRQ1 (keyboard)\*/

write\_port(0x21 , 0xFD);

}

void kprint(const char \*str)

{

unsigned int i = 0;

while (str[i] != '\0') {

vidptr[current\_loc++] = str[i++];

vidptr[current\_loc++] = 0x07;

}

}

void kprint\_newline(void)

{

unsigned int line\_size = BYTES\_FOR\_EACH\_ELEMENT \* COLUMNS\_IN\_LINE;

current\_loc = current\_loc + (line\_size - current\_loc % (line\_size));

}

void clear\_screen(void)

{

unsigned int i = 0;

while (i < SCREENSIZE) {

vidptr[i++] = ' ';

vidptr[i++] = 0x07;

}

}

void keyboard\_handler\_main(void) {

unsigned char status;

char keycode;

/\* write EOI \*/

write\_port(0x20, 0x20);

status = read\_port(KEYBOARD\_STATUS\_PORT);

/\* Lowest bit of status will be set if buffer is not empty \*/

if (status & 0x01) {

keycode = read\_port(KEYBOARD\_DATA\_PORT);

if(keycode < 0)

return;

if(keycode == ENTER\_KEY\_CODE) {

kprint\_newline();

return;

}

vidptr[current\_loc++] = keyboard\_map[(unsigned char) keycode];

vidptr[current\_loc++] = 0x07;

}

}

void kmain(void)

{

const char \*str = “This is a kernel with keyboard input. Type something.”;

clear\_screen();

kprint(str);

kprint\_newline();

kprint\_newline();

idt\_init();

kb\_init();

while(1);

}

First we make a pointer vidptr that points to the address [0xb8000]. This address is the start of video memory in protected mode. The screen’s text memory is simply a chunk of memory in our address space. The memory mapped input/output for the screen starts at [0xb8000] and supports 25 lines, each line contain 80 ascii characters.

Each character element in this text memory is represented by 16 bits (2 bytes), rather than 8 bits (1 byte) which we are used to. The first byte should have the representation of the character as in ASCII. The second byte is the attribute-byte. This describes the formatting of the character including attributes such as color.

To print the character s in green color on black background, we will store the character s in the first byte of the video memory address and the value [0x02] in the second byte.   
0 represents black background and 2 represents green foreground.  
Have a look at table below for different colors:

0 - Black, 1 - Blue, 2 - Green, 3 - Cyan, 4 - Red, 5 - Magenta, 6 - Brown, 7 - Light Grey, 8 - Dark Grey, 9 - Light Blue, 10/a - Light Green, 11/b - Light Cyan, 12/c - Light Red, 13/d - Light Magenta, 14/e - Light Brown, 15/f – White.

In our kernel, we will use light grey character on a black background. So our attribute-byte must have the value [0x07]. In the first while loop, the program writes the blank character with [0x07] attribute all over the 80 columns of the 25 lines. This thus clears the screen.

**Interrupts**

Now, before we go ahead with writing any device driver; we need to understand how the processor gets to know that the device has performed an event.

The easiest solution is polling - to keep checking the status of the device forever. This, for obvious reasons is not efficient and practical. This is where interrupts come into the picture. An interrupt is a signal sent to the processor by the hardware or software indicating an event. With interrupts, we can avoid polling and act only when the specific interrupt we are interested in is triggered.

A device or a chip called Programmable Interrupt Controller (PIC) is responsible for x86 being an interrupt driven architecture. It manages hardware interrupts and sends them to the appropriate system interrupt.

When certain actions are performed on a hardware device, it sends a pulse called Interrupt Request (IRQ) along its specific interrupt line to the PIC chip. The PIC then translates the received IRQ into a system interrupt, and sends a message to interrupt the CPU from whatever it is doing. It is then the kernel’s job to handle these interrupts.

Without a PIC, we would have to poll all the devices in the system to see if an event has occurred in any of them.

Let’s take the case of a keyboard. The keyboard works through the I/O ports 0x60 and 0x64. Port 0x60 gives the data (pressed key) and port 0x64 gives the status. However, you have to know exactly when to read these ports.

Interrupts come quite handy here. When a key is pressed, the keyboard gives a signal to the PIC along its interrupt line IRQ1. The PIC has an offset value stored during initialization of the PIC. It adds the input line number to this offset to form the Interrupt number. Then the processor looks up a certain data structure called the Interrupt Descriptor Table (IDT) to give the interrupt handler address corresponding to the interrupt number.

**Setting up the IDT**

struct IDT\_entry{

unsigned short int offset\_lowerbits;

unsigned short int selector;

unsigned char zero;

unsigned char type\_attr;

unsigned short int offset\_higherbits;

};

struct IDT\_entry IDT[IDT\_SIZE];

void idt\_init(void)

{

unsigned long keyboard\_address;

unsigned long idt\_address;

unsigned long idt\_ptr[2];

/\* populate IDT entry of keyboard's interrupt \*/

keyboard\_address = (unsigned long)keyboard\_handler;

IDT[0x21].offset\_lowerbits = keyboard\_address & 0xffff;

IDT[0x21].selector = 0x08; /\* KERNEL\_CODE\_SEGMENT\_OFFSET \*/

IDT[0x21].zero = 0;

IDT[0x21].type\_attr = 0x8e; /\* INTERRUPT\_GATE \*/

IDT[0x21].offset\_higherbits = (keyboard\_address & 0xffff0000) >> 16;

/\* Ports

\* PIC1 PIC2

\*Command 0x20 0xA0

\*Data 0x21 0xA1

\*/

/\* ICW1 - begin initialization \*/

write\_port(0x20 , 0x11);

write\_port(0xA0 , 0x11);

/\* ICW2 - remap offset address of IDT \*/

/\*

\* In x86 protected mode, we have to remap the PICs beyond 0x20 because

\* Intel have designated the first 32 interrupts as "reserved" for cpu exceptions

\*/

write\_port(0x21 , 0x20);

write\_port(0xA1 , 0x28);

/\* ICW3 - setup cascading \*/

write\_port(0x21 , 0x00);

write\_port(0xA1 , 0x00);

/\* ICW4 - environment info \*/

write\_port(0x21 , 0x01);

write\_port(0xA1 , 0x01);

/\* Initialization finished \*/

/\* mask interrupts \*/

write\_port(0x21 , 0xff);

write\_port(0xA1 , 0xff);

/\* fill the IDT descriptor \*/

idt\_address = (unsigned long)IDT ;

idt\_ptr[0] = (sizeof (struct IDT\_entry) \* IDT\_SIZE) + ((idt\_address & 0xffff) << 16);

idt\_ptr[1] = idt\_address >> 16 ;

load\_idt(idt\_ptr);

}

We implement IDT as an array comprising structures IDT\_entry. We’ll discuss how the keyboard interrupt is mapped to its handler later in the project. First, let’s see how the PICs work.

Modern x86 systems have 2 PIC chips each having 8 input lines. Let’s call them PIC1 and PIC2. PIC1 receives IRQ0 to IRQ7 and PIC2 receives IRQ8 to IRQ15. PIC1 uses port 0x20 for Command and 0x21 for Data. PIC2 uses port 0xA0 for Command and 0xA1 for Data.

The PICs are initialized using 8-bit command words known as Initialization command words (ICW). See [this link](http://stanislavs.org/helppc/8259.html) for the exact bit-by-bit syntax of these commands.

In protected mode, the first command you will need to give the two PICs is the initialize command ICW1 (0x11). This command makes the PIC wait for 3 more initialization words on the data port.

These commands tell the PICs about:

* Its vector offset. (ICW2).
* How the PICs wired as master/slaves. (ICW3)
* Gives additional information about the environment. (ICW4)

The second initialization command is the ICW2, written to the data ports of each PIC. It sets the PIC’s offset value. This is the value to which we add the input line number to form the Interrupt number.

PICs allow cascading of their outputs to inputs between each other. This is setup using ICW3 and each bit represents cascading status for the corresponding IRQ. For now, we won’t use cascading and set all to zeroes.

ICW4 sets the additional enviromental parameters. We will just set the lower most bit to tell the PICs we are running in the 80x86 mode.

PICs are now initialized.

Each PIC has an internal 8 bit register named Interrupt Mask Register (IMR). This register stores a bitmap of the IRQ lines going into the PIC. When a bit is set, the PIC ignores the request. This means we can enable and disable the nth IRQ line by making the value of the nth bit in the IMR as 0 and 1 respectively. Reading from the data port returns value in the IMR register, and writing to it sets the register. Here in our code, after initializing the PICs; we set all bits to 1 thereby disabling all IRQ lines. We will later enable the line corresponding to keyboard interrupt. Now if IRQ lines are enabled, our PICs can receive signals via IRQ lines and convert them to interrupt number by adding with the offset. Now, we need to populate the IDT such that the interrupt number for the keyboard is mapped to the address of the keyboard handler function we will write.

Now we need to know which interrupt number the keyboard handler address should be mapped against in the IDT.

The keyboard uses IRQ1. This is the input line 1 of PIC1. We have initialized PIC1 to an offset 0x20 (see ICW2). To find interrupt number, add 1 + 0x20 ie. 0x21. So, keyboard handler address has to be mapped against interrupt 0x21 in the IDT.

So, the next task is to populate the IDT for the interrupt 0x21. We will map this interrupt to a function keyboard\_handler which we will write in our assembly file.

Each IDT entry consist of 64 bits. In the IDT entry for the interrupt, we do not store the entire address of the handler function together. We split it into 2 parts of 16 bits. The lower bits are stored in the first 16 bits of the IDT entry and the higher 16 bits are stored in the last 16 bits of the IDT entry. This is done to maintain compatibility with the 286.

In the IDT entry, we also have to set the type - that this is done to trap an interrupt. We also need to give the kernel code segment offset. Each GDT entry is 8 bytes long, and the kernel code descriptor is the second segment; so its offset is 0x08 (More on this would be too much for this article). Interrupt gate is represented by 0x8e. The remaining 8 bits in the middle has to be filled with all zeroes. In this way, we have filled the IDT entry corresponding to the keyboard’s interrupt.

Once the required mappings are done in the IDT, we got to tell the CPU where the IDT is located. This is done via the lidt assembly instruction. lidt take one operand. The operand must be a pointer to a descriptor structure that describes the IDT.

The descriptor is quite straight forward. It contains the size of IDT in bytes and its address. I have used an array to pack the values. You may also populate it using a struct.

We have the pointer in the variable idt\_ptr and then pass it on to lidt using the function load\_idt().

load\_idt:

mov edx, [esp + 4]

lidt [edx]

sti

ret

Additionally, load\_idt() function turns the interrupts on using sti instruction.

Once the IDT is set up and loaded, we can turn on keyboard’s IRQ line using the interrupt mask we discussed earlier.

void kb\_init(void)

{

/\* 0xFD is 11111101 - enables only IRQ1 (keyboard)\*/

write\_port(0x21 , 0xFD);

}

**Keyboard interrupt handling function**

Well, now we have successfully mapped keyboard interrupts to the function keyboard\_handler via IDT entry for interrupt 0x21. So, everytime you press a key on your keyboard you can be sure this function is called.

keyboard\_handler:

call keyboard\_handler\_main

iretd

This function just calls another function written in C and returns using the iret class of instructions. We could have written our entire interrupt handling process here, however it’s much easier to write code in C than in assembly - so we take it there.  iret/iretd should be used instead of ret when returning control from an interrupt handler to a program that was interrupted by an interrupt. These class of instructions pop the flags register that was pushed into the stack when the interrupt call was made.

void keyboard\_handler\_main(void) {

unsigned char status;

char keycode;

/\* write EOI \*/

write\_port(0x20, 0x20);

status = read\_port(KEYBOARD\_STATUS\_PORT);

/\* Lowest bit of status will be set if buffer is not empty \*/

if (status & 0x01) {

keycode = read\_port(KEYBOARD\_DATA\_PORT);

if(keycode < 0)

return;

vidptr[current\_loc++] = keyboard\_map[keycode];

vidptr[current\_loc++] = 0x07;

}

}

We first signal EOI (End Of Interrput acknowlegment) by writing it to the PIC’s command port. Only after this; will the PIC allow further interrupt requests. We have to read 2 ports here - the data port 0x60 and the command/status port 0x64.

We first read port 0x64 to get the status. If the lowest bit of the status is 0, it means the buffer is empty and there is no data to read. In other cases, we can read the data port 0x60. This port will give us a keycode of the key pressed. Each keycode corresponds to each key on the keyboard. We use a simple character array defined in the file keyboard\_map.h to map the keycode to the corresponding character. This character is then printed on to the screen using the same technique we used in the previous article.

**The linker script**

We will assemble kernel.asm with NASM to an object file; and then using GCC we will compile kernel.c to another object file. Now, our job is to get these objects linked to an executable bootable kernel.  
For that, we use an explicit linker script, which can be passed as an argument to ld (our linker).

// link.ld

OUTPUT\_FORMAT(elf32-i386)

ENTRY(start)

SECTIONS

{

. = 0x100000;

.text : { \*(.text) }

.data : { \*(.data) }

.bss : { \*(.bss) }

}

First, we set the output format of our output executable to be 32 bit [Executable and Linkable Format](http://elinux.org/Executable_and_Linkable_Format_(ELF)" \t "_blank)(ELF). ELF is the standard binary file format for Unix-like systems on x86 architecture.

**ENTRY** takes one argument. It specifies the symbol name that should be the entry point of our executable.

**SECTIONS**is the most important part for us. Here, we define the layout of our executable. We could specify how the different sections are to be merged and at what location each of these is to be placed.

Within the braces that follow the SECTIONS statement, the period character (.) represents the location counter.  
The location counter is always initialized to [0x0] at beginning of the SECTIONS block. It can be modified by assigning a new value to it.

Remember, the kernel’s code should start at the address [0x100000]. So, we set the location counter to [0x100000].

Have look at the next line .text : { \*(.text) }

The asterisk (\*) is a wildcard character that matches any file name. The expression \*(.text) thus means all .text input sections from all input files.  
  
So, the linker merges all text sections of the object files to the executable’s text section, at the address stored in the location counter. Thus, the code section of our executable begins at [0x100000].

After the linker places the text output section, the value of the location counter will become   
0x1000000 + the size of the text output section.

Similarly, the data and bss sections are merged and placed at the then values of location-counter.

The dd defines a double word of size 4 bytes.

dd 0x1BADB002 ;magic dd 0x00 ;flags dd - (0x1BADB002 + 0x00) ;checksum. m+f+c should be zero

**Building the kernel:** We will now create object files from kernel.asm and kernel.c and then link it using our linker script.

nasm -f elf32 kernel.asm -o kasm.o

will run the assembler to create the object file kasm.o in ELF-32 bit format.

gcc -m32 -c kernel.c -o kc.o

The ’-c ’ option makes sure that after compiling, linking doesn’t implicitly happen.

ld -m elf\_i386 -T link.ld -o kernel kasm.o kc.o

will run the linker with our linker script and generate the executable named kernel.

To run the kernel on the qemu emulator, we execute the following command:

qemu-system-i386 -kernel kernel

**Challenges**

These were the following challenges we faced during the course of our project: -

1. Originally we used an Ubuntu basebox called precise32 to build our kernel from scratch, it didn’t run because the tools that we needed for creating our own bootloader could not be installed and the packages we needed for linux had bugs that kept them from running on the basebox.
2. Since our custom bootloader didn’t work, we decided to use a pre existing bootloader off the Internet. Firstly, we tried GRUB and then secondly we tried mikeOS bootloader. The GRUB bootloader didn’t work either because there was an error in mounting it onto the floppy image drive. MikeOS didn’t work as well because a bin file wasn’t being created out of the c and asm file, which was needed to get it to run on MikeOS.
3. Our limited knowledge on assembly language programming made it hard to create our assembly file. A lot of research had to be done to link our C file with the assembly file.
4. Finally we decided upon using the qemu emulator instead of a bootloader to run our code. Qemu worked with our base kernel but again gave problems while building the kernel with added features and functionality. But we figured it out, in the end and came up with the final kernel for our project. ☺

**Summary**

Our final kernel can print output to the screen as well as take input from the keyboard. We are only handling lowercase a-z and digits 0-9, but with ease this can be extended to include special characters, ALT, SHIFT, CAPS LOCK. We can also get to know if the key was pressed or released from the status port output and perform desired action. A combination of keys may also be mapped for special functions such as shutdown etc.

**References**

1. linuxfromscratch.org
2. mikeos.sourceforge.net
3. gnu.org
4. GNU – GRUB Manual 2.0
5. wiki.osdev.org
6. joelgompert.com
7. debian-handbook.info
8. <http://resources.infosecinstitute.com>
9. How to create an operating system from scratch by Samy Pesse