

**HOW TO
MAKE AN**

**OPERATING
SYSTEM**

FROM SCRATCH

Samy Pessé

Published
with GitBook

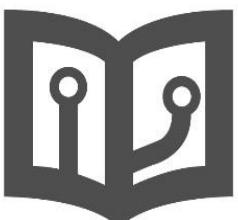


Table of Contents

Introduction	1.1
Introduction about the x86 architecture and about our OS	1.2
Setup the development environment	1.3
First boot with GRUB	1.4
Backbone of the OS and C++ runtime	1.5
Base classes for managing x86 architecture	1.6
GDT	1.7
IDT and interrupts	1.8
Theory: physical and virtual memory	1.9
Memory management: physical and virtual	1.10
Process management and multitasking	1.11
External program execution: ELF files	1.12
Userland and syscalls	1.13
Modular drivers	1.14
Some basics modules: console, keyboard	1.15
IDE Hard disks	1.16
DOS Partitions	1.17
EXT2 read-only filesystems	1.18
Standard C library (libC)	1.19
UNIX basic tools: sh, cat	1.20
Lua interpreter	1.21

How to Make a Computer Operating System

Online book about how to write a computer operating system in C/C++ from scratch.

Caution: This repository is a remake of my old course. It was written several years ago [as one of my first projects when I was in High School](#), I'm still refactoring some parts. The original course was in French and I'm not an English native. I'm going to continue and improve this course in my free-time.

Book: An online version is available at <http://samypesse.gitbooks.io/how-to-create-an-operating-system/> (PDF, Mobi and ePub). It was generated using [GitBook](#).

Source Code: All the system source code will be stored in the `src` directory. Each step will contain links to the different related files.

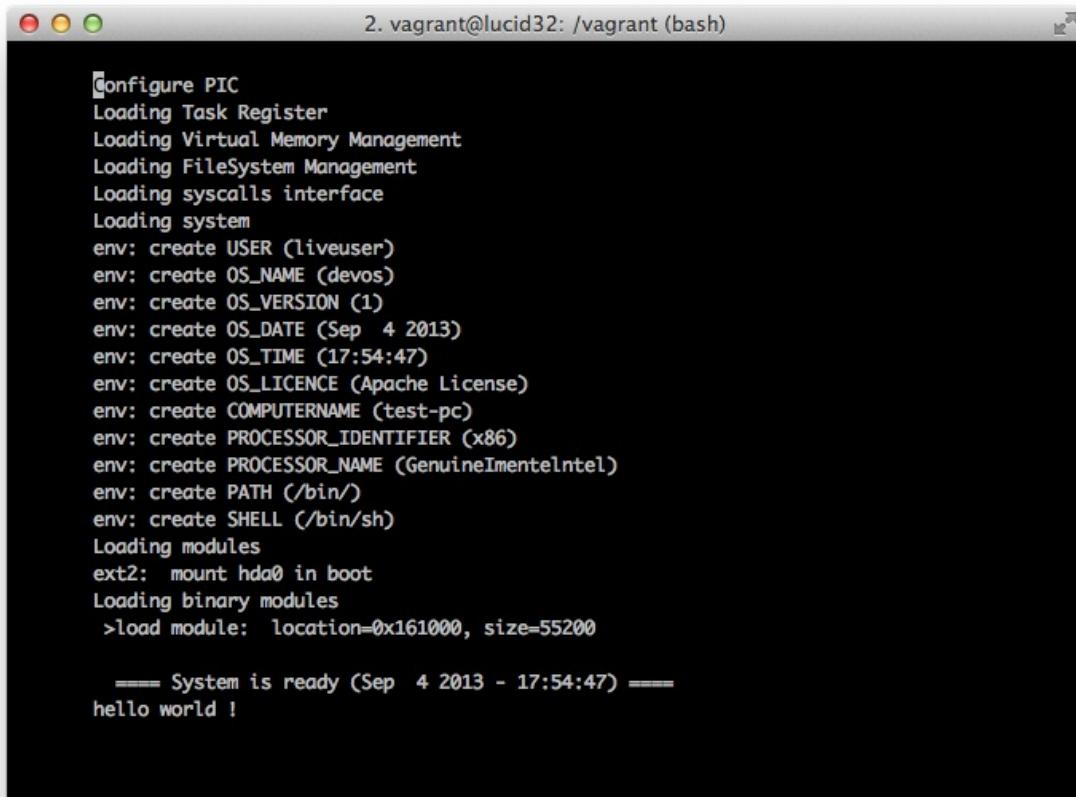
Contributions: This course is open to contributions, feel free to signal errors with issues or directly correct the errors with pull-requests.

Questions: Feel free to ask any questions by adding issues or commenting sections.

You can follow me on Twitter [@SamyPesse](#) or [GitHub](#).

What kind of OS are we building?

The goal is to build a very simple UNIX-based operating system in C++, not just a "proof-of-concept". The OS should be able to boot, start a userland shell, and be extensible.



2. vagrant@lucid32: /vagrant (bash)

```
Configure PIC
Loading Task Register
Loading Virtual Memory Management
Loading FileSystem Management
Loading syscalls interface
Loading system
env: create USER (liveuser)
env: create OS_NAME (devos)
env: create OS_VERSION (1)
env: create OS_DATE (Sep 4 2013)
env: create OS_TIME (17:54:47)
env: create OS_LICENCE (Apache License)
env: create COMPUTERNAME (test-pc)
env: create PROCESSOR_IDENTIFIER (x86)
env: create PROCESSOR_NAME (GenuineIntel)
env: create PATH (/bin/)
env: create SHELL (/bin/sh)
Loading modules
ext2: mount hda0 in boot
Loading binary modules
>load module: location=0x161000, size=55200

==== System is ready (Sep 4 2013 - 17:54:47) ====
hello world !
```

Chapter 1: Introduction to the x86 architecture and about our OS

What is the x86 architecture?

The term x86 denotes a family of backward compatible instruction set architectures based on the Intel 8086 CPU.

The x86 architecture is the most common instruction set architecture since its introduction in 1981 for the IBM PC. A large amount of software, including operating systems (OS's) such as DOS, Windows, Linux, BSD, Solaris and Mac OS X, function with x86-based hardware.

In this course we are not going to design an operating system for the x86-64 architecture but for x86-32, thanks to backward compatibility, our OS will be compatible with our newer PCs (but take caution if you want to test it on your real machine).

Our Operating System

The goal is to build a very simple UNIX-based operating system in C++, but the goal is not to just build a "proof-of-concept". The OS should be able to boot, start a userland shell and be extensible.

The OS will be built for the x86 architecture, running on 32 bits, and compatible with IBM PCs.

Specifications:

- Code in C++
- x86, 32 bit architecture
- Boot with Grub
- Kind of modular system for drivers
- Kind of UNIX style
- Multitasking
- ELF executable in userland
- Modules (accessible in userland using /dev/...) :
 - IDE disks
 - DOS partitions
 - Clock
 - EXT2 (read only)
 - Boch VBE
- Userland :

- API Posix
- LibC
- "Can" run a shell or some executables (e.g., lua)

Chapter 2: Setup the development environment

The first step is to setup a good and viable development environment. Using Vagrant and Virtualbox, you'll be able to compile and test your OS from all the OSs (Linux, Windows or Mac).

Install Vagrant

Vagrant is free and open-source software for creating and configuring virtual development environments. It can be considered a wrapper around VirtualBox.

Vagrant will help us create a clean virtual development environment on whatever system you are using. The first step is to download and install Vagrant for your system at <http://www.vagrantup.com/>.

Install Virtualbox

Oracle VM VirtualBox is a virtualization software package for x86 and AMD64/Intel64-based computers.

Vagrant needs Virtualbox to work, Download and install for your system at <https://www.virtualbox.org/wiki/Downloads>.

Start and test your development environment

Once Vagrant and Virtualbox are installed, you need to download the ubuntu lucid32 image for Vagrant:

```
vagrant box add lucid32 http://files.vagrantup.com/lucid32.box
```

Once the lucid32 image is ready, we need to define our development environment using a *Vagrantfile*, [create a file named Vagrantfile](#). This file defines what prerequisites our environment needs: nasm, make, build-essential, grub and qemu.

Start your box using:

```
vagrant up
```

You can now access your box by using ssh to connect to the virtual box using:

```
vagrant ssh
```

The directory containing the *Vagrantfile* will be mounted by default in the */vagrant* directory of the guest VM (in this case, Ubuntu Lucid32):

```
cd /vagrant
```

Build and test our operating system

The file **Makefile** defines some basics rules for building the kernel, the user libc and some userland programs.

Build:

```
make all
```

Test our operating system with qemu:

```
make run
```

The documentation for qemu is available at [QEMU Emulator Documentation](#).

You can exit the emulator using: Ctrl-a.

Chapter 3: First boot with GRUB

How the boot works?

When an x86-based computer is turned on, it begins a complex path to get to the stage where control is transferred to our kernel's "main" routine (`kmain()`). For this course, we are only going to consider the BIOS boot method and not its successor (UEFI).

The BIOS boot sequence is: RAM detection -> Hardware detection/Initialization -> Boot sequence.

The most important step for us is the "Boot sequence", where the BIOS is done with its initialization and tries to transfer control to the next stage of the bootloader process.

During the "Boot sequence", the BIOS will try to determine a "boot device" (e.g. floppy disk, hard-disk, CD, USB flash memory device or network). Our Operating System will initially boot from the hard-disk (but it will be possible to boot it from a CD or a USB flash memory device in future). A device is considered bootable if the bootsector contains the valid signature bytes `0x55` and `0xAA` at offsets 511 and 512 respectively (called the magic bytes of the Master Boot Record, also known as the MBR). This signature is represented (in binary) as `0b1010101001010101`. The alternating bit pattern was thought to be a protection against certain failures (drive or controller). If this pattern is garbled or `0x00`, the device is not considered bootable.

BIOS physically searches for a boot device by loading the first 512 bytes from the bootsector of each device into physical memory, starting at the address `0x7C00` (1 KiB below the 32 KiB mark). When the valid signature bytes are detected, BIOS transfers control to the `0x7C00` memory address (via a jump instruction) in order to execute the bootsector code.

Throughout this process the CPU has been running in 16-bit Real Mode, which is the default state for x86 CPUs in order to maintain backwards compatibility. To execute the 32-bit instructions within our kernel, a bootloader is required to switch the CPU into Protected Mode.

What is GRUB?

GNU GRUB (short for GNU GRand Unified Bootloader) is a boot loader package from the GNU Project. GRUB is the reference implementation of the Free Software Foundation's Multiboot Specification, which provides a user the choice to boot one of multiple operating systems installed on a computer or select a specific kernel configuration available on a particular operating system's partitions.

To make it simple, GRUB is the first thing booted by the machine (a boot-loader) and will simplify the loading of our kernel stored on the hard-disk.

Why are we using GRUB?

- GRUB is very simple to use
- Make it very simple to load 32bits kernels without needs of 16bits code
- Multiboot with Linux, Windows and others
- Make it easy to load external modules in memory

How to use GRUB?

GRUB uses the Multiboot specification, the executable binary should be 32bits and must contain a special header (multiboot header) in its 8192 first bytes. Our kernel will be a ELF executable file ("Executable and Linkable Format", a common standard file format for executables in most UNIX system).

The first boot sequence of our kernel is written in Assembly: [start.asm](#) and we use a linker file to define our executable structure: [linker.ld](#).

This boot process also initializes some of our C++ runtime, it will be described in the next chapter.

Multiboot header structure:

```
struct multiboot_info {
    u32 flags;
    u32 low_mem;
    u32 high_mem;
    u32 boot_device;
    u32 cmdline;
    u32 mods_count;
    u32 mods_addr;
    struct {
        u32 num;
        u32 size;
        u32 addr;
        u32 shndx;
    } elf_sec;
    unsigned long mmap_length;
    unsigned long mmap_addr;
    unsigned long drives_length;
    unsigned long drives_addr;
    unsigned long config_table;
    unsigned long boot_loader_name;
    unsigned long apm_table;
    unsigned long vbe_control_info;
    unsigned long vbe_mode_info;
    unsigned long vbe_mode;
    unsigned long vbe_interface_seg;
    unsigned long vbe_interface_off;
    unsigned long vbe_interface_len;
};

};
```

You can use the command `mbchk kernel.elf` to validate your kernel.elf file against the multiboot standard. You can also use the command `nm -n kernel.elf` to validate the offset of the different objects in the ELF binary.

Create a disk image for our kernel and grub

The script [diskimage.sh](#) will generate a hard disk image that can be used by QEMU.

The first step is to create a hard-disk image (c.img) using qemu-img:

```
qemu-img create c.img 2M
```

We need now to partition the disk using fdisk:

```
fdisk ./c.img

# Switch to Expert commands
> x

# Change number of cylinders (1-1048576)
> c
> 4

# Change number of heads (1-256, default 16):
> h
> 16

# Change number of sectors/track (1-63, default 63)
> s
> 63

# Return to main menu
> r

# Add a new partition
> n

# Choose primary partition
> p

# Choose partition number
> 1

# Choose first sector (1-4, default 1)
> 1

# Choose last sector, +cylinders or +size{K,M,G} (1-4, default 4)
> 4

# Toggle bootable flag
> a

# Choose first partition for bootable flag
> 1

# Write table to disk and exit
> w
```

We need now to attach the created partition to the loop-device using losetup. This allows a file to be accessed like a block device. The offset of the partition is passed as an argument and calculated using: **offset= start_sector * bytes_by_sector**.

Using `fdisk -l -u c.img`, you get: $63 * 512 = 32256$.

```
losetup -o 32256 /dev/loop1 ./c.img
```

We create a EXT2 filesystem on this new device using:

```
mke2fs /dev/loop1
```

We copy our files on a mounted disk:

```
mount /dev/loop1 /mnt/
cp -R bootdisk/* /mnt/
umount /mnt/
```

Install GRUB on the disk:

```
grub --device-map=/dev/null << EOF
device (hd0) ./c.img
geometry (hd0) 4 16 63
root (hd0,0)
setup (hd0)
quit
EOF
```

And finally we detach the loop device:

```
losetup -d /dev/loop1
```

See Also

- [GNU GRUB on Wikipedia](#)
- [Multiboot specification](#)

Chapter 4: Backbone of the OS and C++ runtime

C++ kernel run-time

A kernel can be written in C++ just as it can be in C, with the exception of a few pitfalls that come with using C++ (runtime support, constructors, etc).

The compiler will assume that all the necessary C++ runtime support is available by default, but as we are not linking `libsupc++` into your C++ kernel, we need to add some basic functions that can be found in the `cxx.cc` file.

Caution: The operators `new` and `delete` cannot be used before virtual memory and pagination have been initialized.

Basic C/C++ functions

The kernel code can't use functions from the standard libraries so we need to add some basic functions for managing memory and strings:

```
void      itoa(char *buf, unsigned long int n, int base);

void *    memset(char *dst, char src, int n);
void *    memcpy(char *dst, char *src, int n);

int      strlen(char *s);
int      strcmp(const char *dst, char *src);
int      strcpy(char *dst, const char *src);
void      strcat(void *dest, const void *src);
char *    strncpy(char *destString, const char *sourceString, int maxLength);
int      strncmp( const char* s1, const char* s2, int c );
```

These functions are defined in [string.cc](#), [memory.cc](#), [itoa.cc](#)

C types

In the next step, we're going to define different types we're going to use in our code. Most of our variable types are going to be unsigned. This means that all the bits are used to store the integer. Signed variables use their first bit to indicate their sign.

```
typedef unsigned char      u8;
typedef unsigned short     u16;
typedef unsigned int       u32;
typedef unsigned long long u64;

typedef signed char       s8;
typedef signed short      s16;
typedef signed int        s32;
typedef signed long long  s64;
```

Compile our kernel

Compiling a kernel is not the same thing as compiling a linux executable, we can't use a standard library and should have no dependencies to the system.

Our [Makefile](#) will define the process to compile and link our kernel.

For x86 architecture, the followings arguments will be used for gcc/g++/ld:

```
# Linker
LD=ld
LDFLAG= -melf_i386 -static -L ./ -T ./arch/$(ARCH)/linker.ld

# C++ compiler
SC=g++
FLAG= $(INCDIR) -g -O2 -w -trigraphs -fno-builtin -fno-exceptions -fno-stack-protecto
r -O0 -m32 -fno-rtti -nostdlib -nodefaultlibs

# Assembly compiler
ASM=nasm
ASMFLAG=-f elf -o
```

Chapter 5: Base classes for managing x86 architecture

Now that we know how to compile our C++ kernel and boot the binary using GRUB, we can start to do some cool things in C/C++.

Printing to the screen console

We are going to use VGA default mode (03h) to display some text to the user. The screen can be directly accessed using the video memory at 0xB8000. The screen resolution is 80x25 and each character on the screen is defined by 2 bytes: one for the character code, and one for the style flag. This means that the total size of the video memory is 4000B (80B25B2B).

In the IO class ([io.cc](#)):

- **x,y**: define the cursor position on the screen
- **real_screen**: define the video memory pointer
- **putc(char c)**: print a unique character on the screen and manage cursor position
- **printf(char* s, ...)**: print a string

We add a method **putc** to the [IO Class](#) to put a character on the screen and update the (x,y) position.

```

/* put a byte on screen */
void Io::putc(char c){
    kattr = 0x07;
    unsigned char *video;
    video = (unsigned char *) (real_screen+ 2 * x + 160 * y);
    // newline
    if (c == '\n') {
        x = 0;
        y++;
    // back space
    } else if (c == '\b') {
        if (x) {
            *(video + 1) = 0x0;
            x--;
        }
    // horizontal tab
    } else if (c == '\t') {
        x = x + 8 - (x % 8);
    // carriage return
    } else if (c == '\r') {
        x = 0;
    } else {
        *video = c;
        *(video + 1) = kattr;

        x++;
        if (x > 79) {
            x = 0;
            y++;
        }
    }
    if (y > 24)
        scrollup(y - 24);
}

```

We also add a useful and very known method: [printf](#)

```

/* put a string in screen */
void Io::print(const char *s, ...){
    va_list ap;

    char buf[16];
    int i, j, size, buflen, neg;

    unsigned char c;
    int ival;
    unsigned int uival;

    va_start(ap, s);

    while ((c = *s++)) {

```

```

size = 0;
neg = 0;

if (c == 0)
    break;
else if (c == '%') {
    c = *s++;
    if (c >= '0' && c <= '9') {
        size = c - '0';
        c = *s++;
    }

    if (c == 'd') {
        ival = va_arg(ap, int);
        if (ival < 0) {
            uival = 0 - ival;
            neg++;
        } else
            uival = ival;
        itoa(buf, uival, 10);

        buflen = strlen(buf);
        if (buflen < size)
            for (i = size, j = buflen; i >= 0;
                 i--, j--)
                buf[i] =
                    (j >=
                     0) ? buf[j] : '0';

        if (neg)
            print("-%s", buf);
        else
            print(buf);
    }
    else if (c == 'u') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 10);

        buflen = strlen(buf);
        if (buflen < size)
            for (i = size, j = buflen; i >= 0;
                 i--, j--)
                buf[i] =
                    (j >=
                     0) ? buf[j] : '0';

        print(buf);
    } else if (c == 'x' || c == 'X') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 16);

        buflen = strlen(buf);
        if (buflen < size)

```

```

        for (i = size, j = buflen; i >= 0;
             i--, j--)
            buf[i] =
                (j >=
                 0) ? buf[j] : '0';

        print("0x%s", buf);
    } else if (c == 'p') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 16);
        size = 8;

        buflen = strlen(buf);
        if (buflen < size)
            for (i = size, j = buflen; i >= 0;
                 i--, j--)
                buf[i] =
                    (j >=
                     0) ? buf[j] : '0';

        print("0x%s", buf);
    } else if (c == 's') {
        print((char *) va_arg(ap, int));
    }
} else
    putc(c);
}

return;
}

```

Assembly interface

A large number of instructions are available in Assembly but there is not equivalent in C (like cli, sti, in and out), so we need an interface to these instructions.

In C, we can include Assembly using the directive "asm()", gcc use gas to compile the assembly.

Caution: gas uses the AT&T syntax.

```
/* output byte */
void Io::outb(u32 ad, u8 v){
    asmv("outb %%al, %%dx" :: "d" (ad), "a" (v));
}

/* output word */
void Io::outw(u32 ad, u16 v){
    asmv("outw %%ax, %%dx" :: "d" (ad), "a" (v));
}

/* output word */
void Io::outl(u32 ad, u32 v){
    asmv("outl %%eax, %%dx" :: "d" (ad), "a" (v));
}

/* input byte */
u8 Io::inb(u32 ad){
    u8 _v;           \
    asmv("inb %%dx, %%al" : "=a" (_v) : "d" (ad)); \
    return _v;
}

/* input word */
u16 Io::inw(u32 ad){
    u16 _v;           \
    asmv("inw %%dx, %%ax" : "=a" (_v) : "d" (ad)); \
    return _v;
}

/* input word */
u32 Io::inl(u32 ad){
    u32 _v;           \
    asmv("inl %%dx, %%eax" : "=a" (_v) : "d" (ad)); \
    return _v;
}
```

Chapter 6: GDT

Thanks to GRUB, your kernel is no longer in real-mode, but already in [protected mode](#), this mode allows us to use all the possibilities of the microprocessor such as virtual memory management, paging and safe multi-tasking.

What is the GDT?

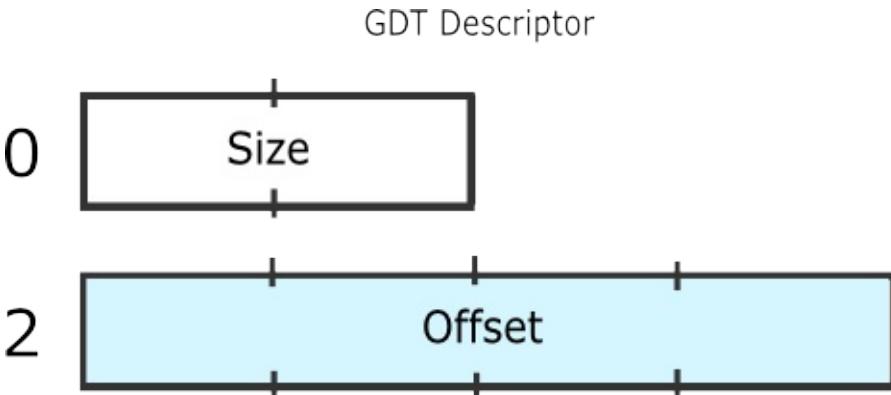
The [GDT](#) ("Global Descriptor Table") is a data structure used to define the different memory areas: the base address, the size and access privileges like execute and write. These memory areas are called "segments".

We are going to use the GDT to define different memory segments:

- "code": kernel code, used to stored the executable binary code
- "data": kernel data
- "stack": kernel stack, used to stored the call stack during kernel execution
- "ucode": user code, used to stored the executable binary code for user program
- "udata": user program data
- "ustack": user stack, used to stored the call stack during execution in userland

How to load our GDT?

GRUB initializes a GDT but this GDT is does not correspond to our kernel. The GDT is loaded using the LGDT assembly instruction. It expects the location of a GDT description structure:



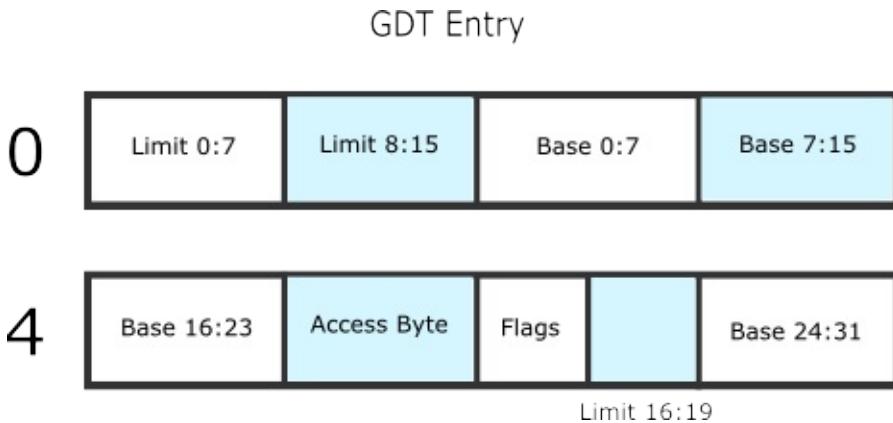
And the C structure:

```
struct gdtr {
    u16 limite;
    u32 base;
} __attribute__ ((packed));
```

Caution: the directive `__attribute__ ((packed))` signal to gcc that the structure should use as little memory as possible. Without this directive, gcc include some bytes to optimize the memory alignment and the access during execution.

Now we need to define our GDT table and then load it using LGDT. The GDT table can be stored wherever we want in memory, its address should just be signaled to the process using the GDTR registry.

The GDT table is composed of segments with the following structure:



And the C structure:

```
struct gdtdesc {
    u16 lim0_15;
    u16 base0_15;
    u8 base16_23;
    u8 acces;
    u8 lim16_19:4;
    u8 other:4;
    u8 base24_31;
} __attribute__ ((packed));
```

How to define our GDT table?

We need now to define our GDT in memory and finally load it using the GDTR registry.

We are going to store our GDT at the address:

```
#define GDTBASE 0x000000800
```

The function **init_gdt_desc** in [x86.cc](#) initialize a gdt segment descriptor.

```
void init_gdt_desc(u32 base, u32 limite, u8 acces, u8 other, struct gdtdesc *desc)
{
    desc->lim0_15 = (limite & 0xffff);
    desc->base0_15 = (base & 0xffff);
    desc->base16_23 = (base & 0xff0000) >> 16;
    desc->acces = acces;
    desc->lim16_19 = (limite & 0xf0000) >> 16;
    desc->other = (other & 0xf);
    desc->base24_31 = (base & 0xff000000) >> 24;
    return;
}
```

And the function **init_gdt** initialize the GDT, some parts of the below function will be explained later and are used for multitasking.

```

void init_gdt(void)
{
    default_tss.debug_flag = 0x00;
    default_tss.io_map = 0x00;
    default_tss.esp0 = 0x1FFF0;
    default_tss.ss0 = 0x18;

    /* initialize gdt segments */
    init_gdt_desc(0x0, 0x0, 0x0, 0x0, &kgdt[0]);
    init_gdt_desc(0x0, 0xFFFFF, 0x9B, 0x0D, &kgdt[1]); /* code */
    init_gdt_desc(0x0, 0xFFFFF, 0x93, 0x0D, &kgdt[2]); /* data */
    init_gdt_desc(0x0, 0x0, 0x97, 0x0D, &kgdt[3]); /* stack */

    init_gdt_desc(0x0, 0xFFFFF, 0xFF, 0x0D, &kgdt[4]); /* ucode */
    init_gdt_desc(0x0, 0xFFFFF, 0xF3, 0x0D, &kgdt[5]); /* udata */
    init_gdt_desc(0x0, 0x0, 0xF7, 0x0D, &kgdt[6]); /* ustack */

    init_gdt_desc((u32) & default_tss, 0x67, 0xE9, 0x00, &kgdt[7]); /* descripteur
de tss */

    /* initialize the gdtr structure */
    kgdtr.limite = GDTSIZE * 8;
    kgdtr.base = GDTBASE;

    /* copy the gdtr to its memory area */
    memcpy((char *) kgdtr.base, (char *) kgdt, kgdtr.limite);

    /* load the gdtr registry */
    asm("lgdtl (kgdtr)");

    /* initiliaz the segments */
    asm("    movw $0x10, %ax    \n \
        movw %ax, %ds    \n \
        movw %ax, %es    \n \
        movw %ax, %fs    \n \
        movw %ax, %gs    \n \
        ljmp $0x08, $next    \n \
        next:        \n");
}

```

Chapter 7: IDT and interrupts

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.

There are 3 types of interrupts:

- **Hardware interrupts:** are sent to the processor from an external device (keyboard, mouse, hard disk, ...). Hardware interrupts were introduced as a way to reduce wasting the processor's valuable time in polling loops, waiting for external events.
- **Software interrupts:** are initiated voluntarily by the software. It's used to manage system calls.
- **Exceptions:** are used for errors or events occurring during program execution that are exceptional enough that they cannot be handled within the program itself (division by zero, page fault, ...)

The keyboard example:

When the user pressed a key on the keyboard, the keyboard controller will signal an interrupt to the Interrupt Controller. If the interrupt is not masked, the controller will signal the interrupt to the processor, the processor will execute a routine to manage the interrupt (key pressed or key released), this routine could, for example, get the pressed key from the keyboard controller and print the key to the screen. Once the character processing routine is completed, the interrupted job can be resumed.

What is the PIC?

The [PIC](#) (Programmable interrupt controller) is a device that is used to combine several sources of interrupt onto one or more CPU lines, while allowing priority levels to be assigned to its interrupt outputs. When the device has multiple interrupt outputs to assert, it asserts them in the order of their relative priority.

The best known PIC is the 8259A, each 8259A can handle 8 devices but most computers have two controllers: one master and one slave, this allows the computer to manage interrupts from 14 devices.

In this chapter, we will need to program this controller to initialize and mask interrupts.

What is the IDT?

The Interrupt Descriptor Table (IDT) is a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions.

Our kernel is going to use the IDT to define the different functions to be executed when an interrupt occurred.

Like the GDT, the IDT is loaded using the LIDTL assembly instruction. It expects the location of a IDT description structure:

```
struct idtr {  
    u16 limite;  
    u32 base;  
} __attribute__ ((packed));
```

The IDT table is composed of IDT segments with the following structure:

```
struct idtdesc {  
    u16 offset0_15;  
    u16 select;  
    u16 type;  
    u16 offset16_31;  
} __attribute__ ((packed));
```

Caution: the directive `__attribute__ ((packed))` signal to gcc that the structure should use as little memory as possible. Without this directive, gcc includes some bytes to optimize the memory alignment and the access during execution.

Now we need to define our IDT table and then load it using LIDTL. The IDT table can be stored wherever we want in memory, its address should just be signaled to the process using the IDTR registry.

Here is a table of common interrupts (Maskable hardware interrupt are called IRQ):

IRQ	Description
0	Programmable Interrupt Timer Interrupt
1	Keyboard Interrupt
2	Cascade (used internally by the two PICs. never raised)
3	COM2 (if enabled)
4	COM1 (if enabled)
5	LPT2 (if enabled)
6	Floppy Disk
7	LPT1
8	CMOS real-time clock (if enabled)
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

How to initialize the interrupts?

This is a simple method to define an IDT segment

```
void init_idt_desc(u16 select, u32 offset, u16 type, struct idtdesc *desc)
{
    desc->offset0_15 = (offset & 0xffff);
    desc->select = select;
    desc->type = type;
    desc->offset16_31 = (offset & 0xffff0000) >> 16;
    return;
}
```

And we can now initialize the interrupts:

```
#define IDTBASE 0x0000000000
#define IDTSIZE 0xFF
idtr kidtr;
```

```

void init_idt(void)
{
    /* Init irq */
    int i;
    for (i = 0; i < IDTSIZE; i++)
        init_idt_desc(0x08, (u32)_asm_schedule, INTGATE, &kidt[i]); //

    /* Vectors 0 -> 31 are for exceptions */
    init_idt_desc(0x08, (u32) _asm_exc_GP, INTGATE, &kidt[13]);           /* #GP */
    init_idt_desc(0x08, (u32) _asm_exc_PF, INTGATE, &kidt[14]);           /* #PF */

    init_idt_desc(0x08, (u32) _asm_schedule, INTGATE, &kidt[32]);
    init_idt_desc(0x08, (u32) _asm_int_1, INTGATE, &kidt[33]);

    init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt[48]);
    init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt[128]); //48

    kidtr.limite = IDTSIZE * 8;
    kidtr.base = IDTBASE;

    /* Copy the IDT to the memory */
    memcpy((char *) kidtr.base, (char *) kidt, kidtr.limite);

    /* Load the IDTR registry */
    asm("lidtl (kidtr)");
}

```

After initialization of our IDT, we need to activate interrupts by configuring the PIC. The following function will configure the two PICs by writing in their internal registries using the output ports of the processor `io.outb`. We configure the PICs using the ports:

- Master PIC: 0x20 and 0x21
- Slave PIC: 0xA0 and 0xA1

For a PIC, there are 2 types of registries:

- ICW (Initialization Command Word): reinit the controller
- OCW (Operation Control Word): configure the controller once initialized (used to mask/unmask the interrupts)

```

void init_pic(void)
{
    /* Initialization of ICW1 */
    io.outb(0x20, 0x11);
    io.outb(0xA0, 0x11);

    /* Initialization of ICW2 */
    io.outb(0x21, 0x20);    /* start vector = 32 */
    io.outb(0xA1, 0x70);    /* start vector = 96 */

    /* Initialization of ICW3 */
    io.outb(0x21, 0x04);
    io.outb(0xA1, 0x02);

    /* Initialization of ICW4 */
    io.outb(0x21, 0x01);
    io.outb(0xA1, 0x01);

    /* mask interrupts */
    io.outb(0x21, 0x0);
    io.outb(0xA1, 0x0);
}

```

PIC ICW configurations details

The registries have to be configured in order.

ICW1 (port 0x20 / port 0xA0)

```

|0|0|0|1|x|0|x|x|
  |   | +--- with ICW4 (1) or without (0)
  |   +---- one controller (1), or cascade (0)
  +----- triggering by level (level) (1) or by edge (edge) (0)

```

ICW2 (port 0x21 / port 0xA1)

```

|x|x|x|x|x|0|0|0|
  | | | | |
  +----- base address for interrupts vectors

```

ICW2 (port 0x21 / port 0xA1)

For the master:

```
|x|x|x|x|x|x|x|x|x|  
| | | | | | | |  
+----- slave controller connected to the port yes (1), or no (0)
```

For the slave:

```
|0|0|0|0|0|x|x|x| pour l'esclave  
| | |  
+----- Slave ID which is equal to the master port
```

ICW4 (port 0x21 / port 0xA1)

It is used to define in which mode the controller should work.

```
|0|0|0|x|x|x|x|1|  
| | | +----- mode "automatic end of interrupt" AEOI (1)  
| | +----- mode buffered slave (0) or master (1)  
| +----- mode buffered (1)  
+----- mode "fully nested" (1)
```

Why do idt segments offset our ASM functions?

You should have noticed that when I'm initializing our IDT segments, I'm using offsets to segment the code in Assembly. The different functions are defined in [x86int.asm](#) and are of the following scheme:

```
%macro SAVE_REGS 0
    pushad
    push ds
    push es
    push fs
    push gs
    push ebx
    mov bx, 0x10
    mov ds, bx
    pop ebx
%endmacro

%macro RESTORE_REGS 0
    pop gs
    pop fs
    pop es
    pop ds
    popad
%endmacro

%macro INTERRUPT 1
    global _asm_int_%1
    _asm_int_%1:
        SAVE_REGS
        push %1
        call isr_default_int
        pop eax    ;;a enlever sinon
        mov al, 0x20
        out 0x20, al
        RESTORE_REGS
        iret
%endmacro
```

These macros will be used to define the interrupt segment that will prevent corruption of the different registries, it will be very useful for multitasking.

Chapter 8: Theory: physical and virtual memory

In the chapter related to the GDT, we saw that using segmentation a physical memory address is calculated using a segment selector and an offset.

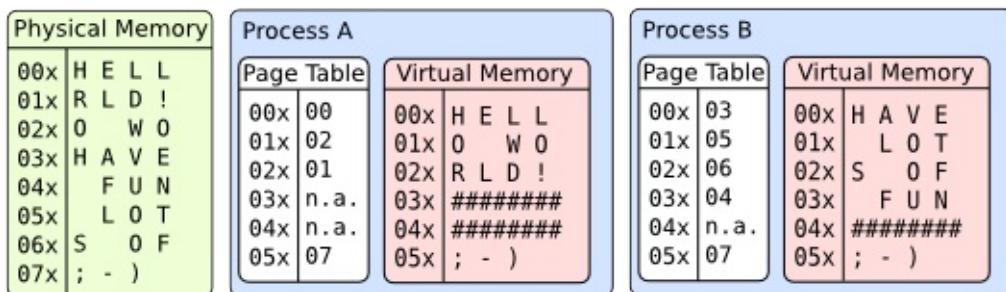
In this chapter, we are going to implement paging, paging will translate a linear address from segmentation into a physical address.

Why do we need paging?

Paging will allow our kernel to:

- use the hard-drive as a memory and not be limited by the machine ram memory limit
- to have a unique memory space for each process
- to allow and unallow memory space in a dynamic way

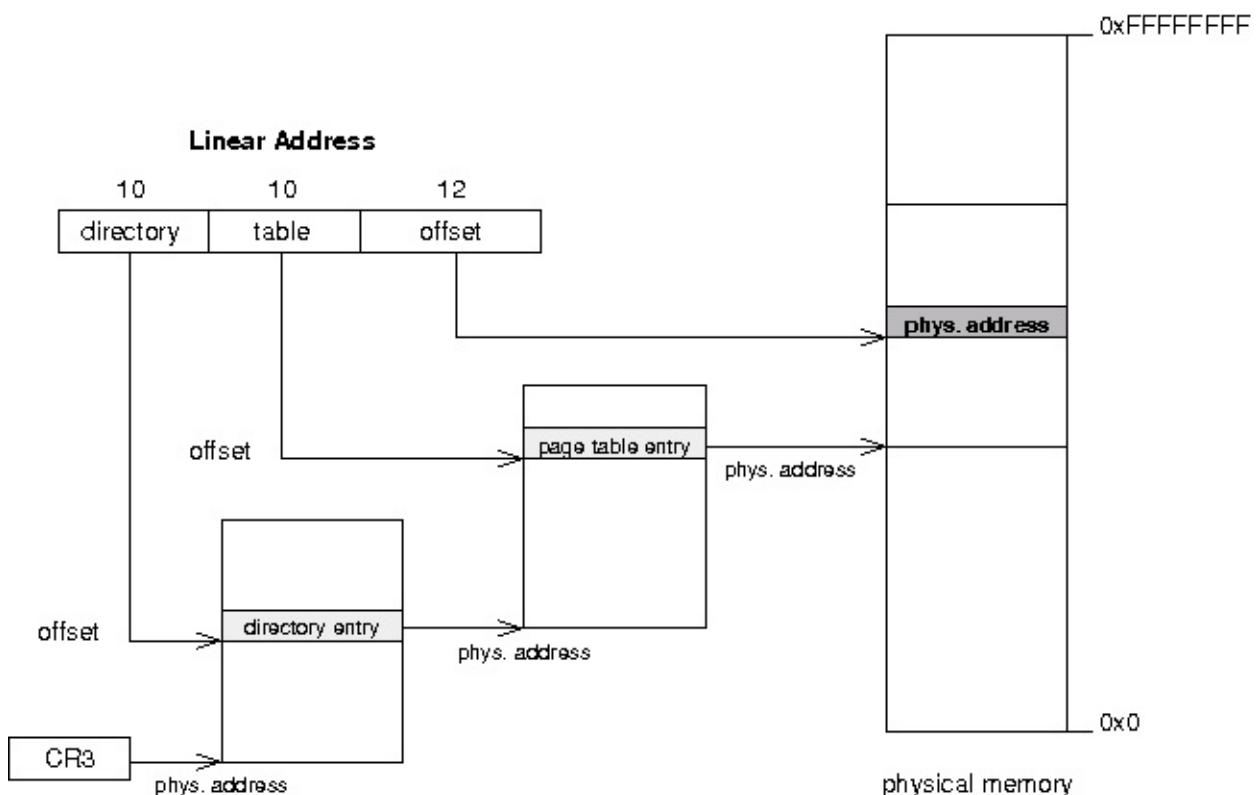
In a paged system, each process may execute in its own 4gb area of memory, without any chance of effecting any other process's memory, or the kernel's. It simplifies multitasking.



How does it work?

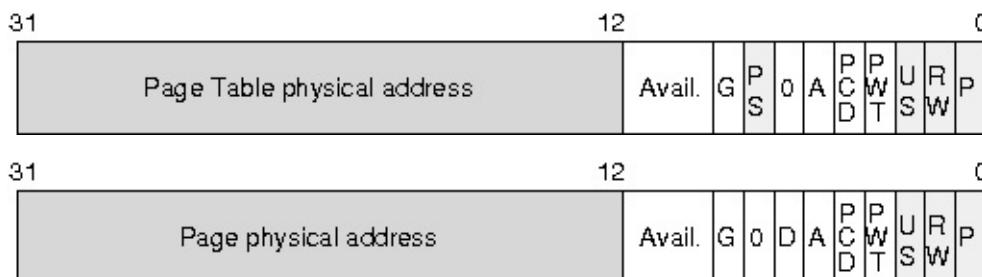
The translation of a linear address to a physical address is done in multiple steps:

1. The processor use the registry `CR3` to know the physical address of the pages directory.
2. The first 10 bits of the linear address represent an offset (between 0 and 1023), pointing to an entry in the pages directory. This entry contains the physical address of a pages table.
3. the next 10 bits of the linear address represent an offset, pointing to an entry in the pages table. This entry is pointing to a 4ko page.
4. The last 12 bits of the linear address represent an offset (between 0 and 4095), which indicates the position in the 4ko page.



Format for pages table and directory

The two types of entries (table and directory) look like the same. Only the field in gray will be used in our OS.



- **P** : indicate if the page or table is in physical memory
- **R/W** : indicate if the page or table is accessible in writing (equals 1)
- **U/S** : equals 1 to allow access to non-preferred tasks
- **A** : indicate if the page or table was accessed
- **D** : (only for pages table) indicate if the page was written
- **PS** (only for pages directory) indicate the size of pages:
 - 0 = 4kb
 - 1 = 4mb

Note: Physical addresses in the pages directory or pages table are written using 20 bits because these addresses are aligned on 4kb, so the last 12 bits should be equal to 0.

- A pages directory or pages table used $1024 \times 4 = 4096$ bytes = 4k

- A pages table can address $1024 * 4k = 4 \text{ Mb}$
- A pages directory can address $1024 (1024 \text{ Mb}) = 4 \text{ Gb}$

How to enable pagination?

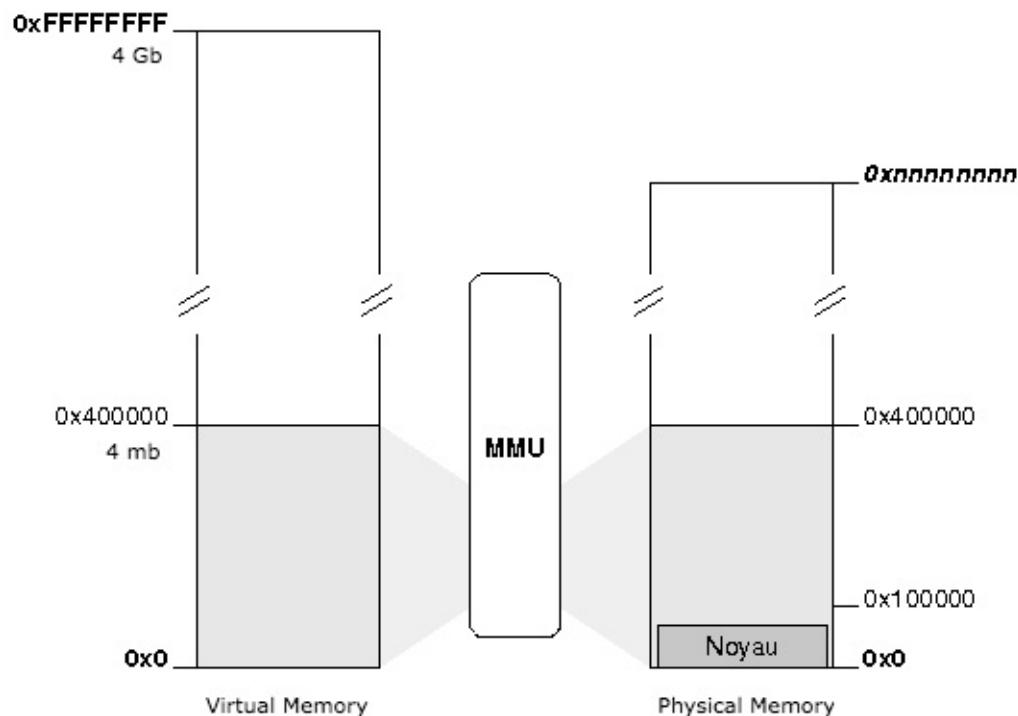
To enable pagination, we just need to set bit 31 of the `CR0` registry to 1:

```
asm("  mov %%cr0, %%eax; \
      or %1, %%eax;      \
      mov %%eax, %%cr0" \
      :: "i"(0x80000000));
```

But before, we need to initialize our pages directory with at least one pages table.

Identity Mapping

With the identity mapping model, the page will apply only to the kernel as the first 4 MB of virtual memory coincide with the first 4 MB of physical memory:



This model is simple: the first virtual memory page coincide to the first page in physical memory, the second page coincide to the second page on physical memory and so on ...

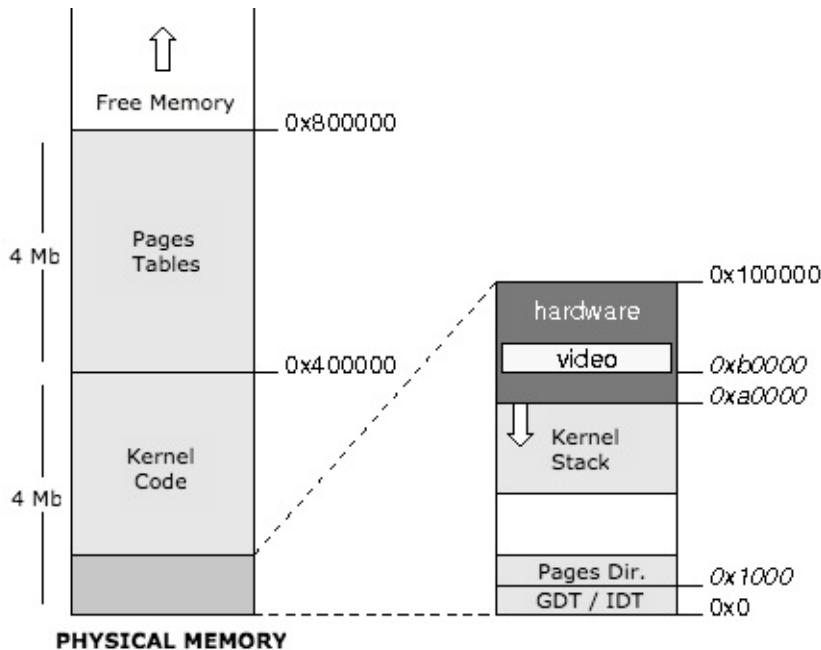
Memory management: physical and virtual

The kernel knows the size of the physical memory available thanks to [GRUB](#).

In our implementation, the first 8 megabytes of physical memory will be reserved for use by the kernel and will contain:

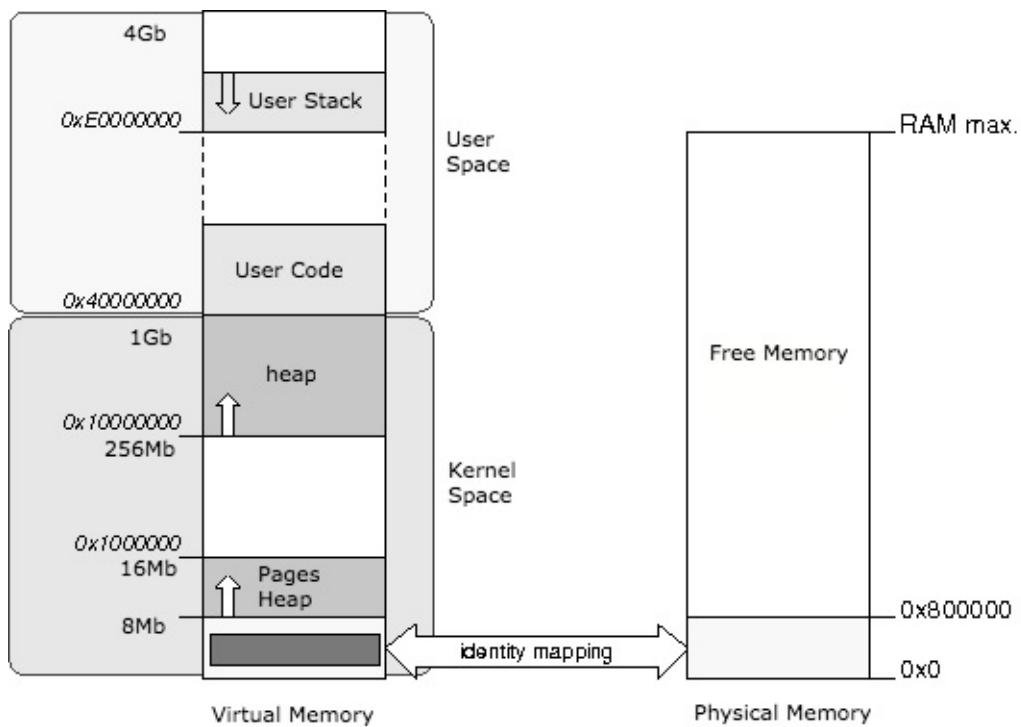
- The kernel
- GDT, IDT et TSS
- Kernel Stack
- Some space reserved to hardware (video memory, ...)
- Page directory and pages table for the kernel

The rest of the physical memory is freely available to the kernel and applications.



Virtual Memory Mapping

The address space between the beginning of memory and `0x40000000` address is the kernel space, while the space between the address `0x40000000` and the end of the memory corresponds to user space:



The kernel space in virtual memory, which is using 1Gb of virtual memory, is common to all tasks (kernel and user).

This is implemented by pointing the first 256 entries of the task page directory to the kernel page directory (In [vmm.cc](#)):

```
/*
 * Kernel Space. v_addr < USER_OFFSET are addressed by the kernel pages table
 */
for (i=0; i<256; i++)
    pdir[i] = pd0[i];
```

Writing a Simple Operating System — from Scratch

by
Nick Blundell

School of Computer Science, University of Birmingham,
UK

Draft: December 2, 2010

Copyright © 2009–2010 Nick Blundell

Contents

Contents	ii	
1	Introduction	1
2	Computer Architecture and the Boot Process	3
2.1	The Boot Process	3
2.2	BIOS, Boot Blocks, and the Magic Number	4
2.3	CPU Emulation	5
2.3.1	Bochs: A x86 CPU Emulator	6
2.3.2	QEmu	6
2.4	The Usefulness of Hexadecimal Notation	6
3	Boot Sector Programming (in 16-bit Real Mode)	8
3.1	Boot Sector Re-visited	8
3.2	16-bit Real Mode	10
3.3	Erm, Hello?	10
3.3.1	Interrupts	11
3.3.2	CPU Registers	11
3.3.3	Putting it all Together	11
3.4	Hello, World!	13
3.4.1	Memory, Addresses, and Labels	13
3.4.2	'X' Marks the Spot	13
	Question 1	16
3.4.3	Defining Strings	16
3.4.4	Using the Stack	17
	Question 2	17
3.4.5	Control Structures	17
	Question 3	19
3.4.6	Calling Functions	19
3.4.7	Include Files	21
3.4.8	Putting it all Together	21
	Question 4	21
3.4.9	Summary	22

3.5	Nurse, Fetch me my Steth-o-scope	22
3.5.1	Question 5 (Advanced)	23
3.6	Reading the Disk	23
3.6.1	Extended Memory Access Using Segments	23
3.6.2	How Disk Drives Work	24
3.6.3	Using BIOS to Read the Disk	27
3.6.4	Putting it all Together	28
4	Entering 32-bit Protected Mode	30
4.1	Adapting to Life Without BIOS	31
4.2	Understanding the Global Descriptor Table	32
4.3	Defining the GDT in Assembly	35
4.4	Making the Switch	36
4.5	Putting it all Together	39
5	Writing, Building, and Loading Your Kernel	41
5.1	Understanding C Compilation	41
5.1.1	Generating Raw Machine Code	41
5.1.2	Local Variables	44
5.1.3	Calling Functions	46
5.1.4	Pointers, Addresses, and Data	47
5.2	Executing our Kernel Code	49
5.2.1	Writing our Kernel	50
5.2.2	Creating a Boot Sector to Bootstrap our Kernel	50
5.2.3	Finding Our Way into the Kernel	53
5.3	Automating Builds with Make	54
5.3.1	Organising Our Operating System's Code Base	57
5.4	C Primer	59
5.4.1	The Pre-processor and Directives	59
5.4.2	Function Declarations and Header Files	60
6	Developing Essential Device Drivers and a Filesystem	62
6.1	Hardware Input/Output	62
6.1.1	I/O Buses	63
6.1.2	I/O Programming	63
6.1.3	Direct Memory Access	65
6.2	Screen Driver	65
6.2.1	Understanding the Display Device	65
6.2.2	Basic Screen Driver Implementation	65
6.2.3	Scrolling the Screen	69
6.3	Handling Interrupts	70
6.4	Keyboard Driver	70
6.5	Hard-disk Driver	70
6.6	File System	70
7	Implementing Processes	71
7.1	Single Processing	71
7.2	Multi-processing	71

CONTENTS

iv

8 Summary **72**

Bibliography **73**

Chapter 1

Introduction

We've all used an operating system (OS) before (e.g. Windows XP, Linux, etc.), and perhaps we have even written some programs to run on one; but what is an OS actually there for? how much of what I see when I use a computer is done by hardware and how much is done by software? and how does the computer actually work?

The late Prof. Doug Shepherd, a lively teacher of mine at Lancaster University, once reminded me amid my grumbling about some annoying programming problem that, back in the day, before he could even *begin* to attempt any research, he had to write his own operating system, from scratch. So it seems that, today, we take a lot for granted about how these wonderful machines actually work underneath all those layers of software that commonly come bundled with them and which are required for their day-to-day usefulness.

Here, concentrating on the widely used x86 architecture CPU, we will strip bare our computer of *all* software and follow in Doug's early footsteps, learning along the way about:

- How a computer boots
- How to write low-level programs in the barren landscape where no operating system yet exists
- How to configure the CPU so that we can begin to use its extended functionality
- How to bootstrap code written in a higher-level language, so that we can really start to make some progress towards our own operating system
- How to create some fundamental operating system services, such as device drivers, file systems, multi-tasking processing.

Note that, in terms of practical operating system functionality, this guide does not aim to be extensive, but instead aims to pool together snippets of information from many sources into a self-contained and coherent document, that will give you a hands-on experience of low-level programming, how operating systems are written, and the kind of problems they must solve. The approach taken by this guide is unique in that the particular languages and tools (e.g. assembly, C, Make, etc.) are not the focus but instead are treated as a means to an end: we will learn what we need to about these things to help us achieve our main goal.

This work is not intended as a replacement but rather as a stepping stone to excellent work such as the Minix project [?] and to operating system development in general.

Chapter 2

Computer Architecture and the Boot Process

2.1 The Boot Process

Now, we begin our journey.

When we reboot our computer, it must start up again, initially without any notion of an operating system. Somehow, it must load the operating system --- whatever variant that may be --- from some permanent storage device that is currently attached to the computer (e.g. a floppy disk, a hard disk, a USB dongle, etc.).

As we will shortly discover, the pre-OS environment of your computer offers little in the way of rich services: at this stage even a simple file system would be a luxury (e.g. read and write logical files to a disk), but we have none of that. Luckily, what we do have is the Basic Input/Output Software (BIOS), a collection of software routines that are initially loaded from a chip into memory and initialised when the computer is switched on. BIOS provides auto-detection and basic control of your computer's essential devices, such as the screen, keyboard, and hard disks.

After BIOS completes some low-level tests of the hardware, particularly whether or not the installed memory is working correctly, it must boot the operating system stored on one of your devices. Here, we are reminded, though, that BIOS cannot simply load a file that represents your operating system from a disk, since BIOS has no notion of a file-system. BIOS must read specific sectors of data (usually 512 bytes in size) from specific physical locations of the disk devices, such as Cylinder 2, Head 3, Sector 5 (details of disk addressing are described later, in Section XXX).

So, the easiest place for BIOS to find our OS is in the first sector of one of the disks (i.e. Cylinder 0, Head 0, Sector 0), known as the *boot sector*. Since some of our disks may not contain an operating systems (they may simply be connected for additional storage), then it is important that BIOS can determine whether the boot sector of a particular disk is boot code that is intended for execution or simply data. Note that the CPU does not differentiate between code and data: both can be interpreted as CPU instructions, where code is simply instructions that have been crafted by a programmer into some useful algorithm.

Again, an unsophisticated means is adopted here by BIOS, whereby the last two bytes of an intended boot sector must be set to the magic number `0xaa55`. So, BIOS loops through each storage device (e.g. floppy drive, hard disk, CD drive, etc.), reads the boot sector into memory, and instructs the CPU to begin executing the first boot sector it finds that ends with the magic number.

This is where we seize control of the computer.

2.2 BIOS, Boot Blocks, and the Magic Number

If we use a binary editor, such as TextPad [?] or GHex [?], that will let us write raw byte values to a file --- rather than a standard text editor that will convert characters such as 'A' into ASCII values --- then we can craft ourselves a simple yet valid boot sector.

```
e9 fd ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
*  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
```

Figure 2.1: A machine code boot sector, with each byte displayed in hexadecimal.

Note that, in Figure 2.1, the three important features are:

- The initial three bytes, in hexadecimal as `0xe9`, `0xfd` and `0xff`, are actually machine code instructions, as defined by the CPU manufacturer, to perform an endless jump.
- The last two bytes, `0x55` and `0xaa`, make up the magic number, which tells BIOS that this is indeed a boot block and not just data that happens to be on a drive's boot sector.
- The file is padded with zeros ('*' indicates zeros omitted for brevity), basically to position the magic BIOS number at the end of the 512 byte disk sector.

An important note on endianness. You might be wondering why the magic BIOS number was earlier described as the 16-bit value `0xaa55` but in our boot sector was written as the consecutive bytes `0x55` and `0xaa`. This is because the x86 architecture handles multi-byte values in *little-endian* format, whereby less significant bytes proceed more significant bytes, which is contrary to our familiar numbering system --- though if our system ever switched and I had £0000005 in my bank account, I would be able to retire now, and perhaps donate a couple of quid to the needy Ex-millionaires Foundation.

Compilers and assemblers can hide many issues of endianness from us by allowing us to define the types of data, such that, say, a 16-bit value is serialised automatically into machine code with its bytes in the correct order. However, it is sometimes useful,

especially when looking for bugs, to know exactly where an individual byte will be stored on a storage device or in memory, so endianness is very important.

This is possibly the smallest program your computer could run, but it is a valid program nonetheless, and we can test this in two ways, the second of which is much safer and better suited to our kind of experiments:

- Using whatever means your current operating system will allow, write this boot block to the first sector of a non-essential storage device (e.g. floppy disk or flash drive), then reboot the computer.
- Use virtual machine software, such as VMWare or VirtualBox, and set the boot block code as a disk image of a virtual machine, then start-up the virtual machine.

You can be sure this code has been loaded and executed if your computer simply hangs after booting, without a message such as “No operating system found”. This is the infinite loop at work, that we put at the start of the code. Without this loop the CPU would tear off, executing every subsequent instruction in memory, most of which will be random, uninitialised bytes, until it throws itself into some invalid state and either reboots or, by chance, stumbles upon and runs a BIOS routine that formats your main disk.

Remember, it is us that program the computer, and the computer follows our instructions blindly, fetching and executing them, until it is switched off; so we need to make sure that it executes our crafted code rather than random bytes of data held somewhere in memory. At this low level, we have a lot of power and responsibility over our computer, so we need to learn how to control it.

2.3 CPU Emulation

There is a *third*, more convenient option for testing these low-level programs without continuously having to reboot a machine or risk scrubbing your important data off a disk, and that is to use a CPU emulator such as Bochs or QEmu. Unlike machine virtualisation (e.g. VMWare, VirtualBox), which tries to optimise for performance and therefore usage of the hosted operating system by running guest instructions directly on the CPU, emulation involves a program that behaves like a specific CPU architecture, using variables to represent CPU registers and high-level control structures to simulate lower level jumps and so on, so is much slower but often better suited for development and debugging such systems.

Note that, in order to do anything useful with an emulator, you need to give it some code to run in the form of a disk image file. An image file simply is the raw data (i.e. machine code and data) that would otherwise have been written to medium of a hard disk, a floppy disk, a CDROM, USB stick, etc. Indeed, some emulators will successfully boot and run a real operating system from an image file downloaded or extracted from an installation CDROM --- though virtualisation is better suited to this kind of use.

The emulators translate low-level display device instructions into pixel rendering on a desktop window, so you can see exactly what would be rendered on a real monitor.

In general, and for the exercises in this document, it follows that any machine code that runs correctly under an emulator will run correctly on the real architecture --- though obviously must faster.

2.3.1 Bochs: A x86 CPU Emulator

Bochs requires that we set up a simple configuration file, `bochsrc`, in the local directory, that describes details of how real devices (e.g. the screen and keyboard) are to be emulated and, importantly, which floppy disk image is to be booted when the emulated computer starts.

Figure 2.2 shows a sample Bochs configuration file that we can use to test the boot sector written in Section XXX and saved as the file `boot_sect.bin`

```
# Tell bochs to use our boot sector code as though it were
# a floppy disk inserted into a computer at boot time.
floppya: 1_44=boot_sect.bin, status=inserted
boot: a
```

Figure 2.2: A simple Bochs configuration file.

To test our boot sector in Bochs, simply type:

```
$bochs
```

As a simple experiment, try changing the BIOS magic number in our boot sector to something invalid then re-running Bochs.

Since Bochs' emulation of a CPU is close to the real thing, after you've tested code in Bochs, you should be able to boot it on a real machine, on which it will run much faster.

2.3.2 QEmu

QEmu is similar to Bochs, though is much more efficient and capable also of emulating architectures other than x86. Though QEmu is less well documented than Bochs, a need for no configuration file means it is easier to get running, as follows:

```
$qemu <your-os-boot-disk-image-file>
```

2.4 The Usefulness of Hexadecimal Notation

We've already seen some examples of *hexadecimal*, so it is important to understand why hexadecimal is often used in lower-level programming.

First it may be helpful to consider why counting in ten seems so natural to us, because when we see hexadecimal for the first time we always ask ourselves: why not simply count to ten? Not being an expert on the matter, I will make the assumption that counting to ten has something to do with most people having a total of ten fingers on their hands, which led to the ideas of numbers being represented as 10 distinct symbols: `0,1,2,...8,9`

Decimal has a base of ten (i.e. has ten distinct digit symbols), but hexadecimal has a base of 16, so we have to invent some new number symbols; and the lazy way is just to use a few letters, giving us: `0,1,2,...8,9,a,b,c,d,e,f`, where the single digit `d`, for example, represents a count of 13.

To distinguish among hexadecimal and other number systems, we often use the prefix `0x`, or sometimes the suffix `h`, which is especially important for hexadecimal digits that happen not to contain any of the letter digits, for example: `0x50` does not equal (decimal) `50` --- `0x50` is actually `80` in decimal.

The thing is, that a computer represent a number as a sequence of *bits* (binary digits), since fundamentally its circuitry can distinguish between only two electrical states: `0` and `1` --- it's like the computer has a total of only two fingers. So, to represent a number larger than `1`, the computer can bunch together a series of bits, just like we may count higher than `9` by having two or more digits (e.g. `456`, `23`, etc.).

Names have been adopted for bit series of certain lengths to make it easier to talk about and agree upon the size of numbers we are dealing with. The instructions of most computers deal with a minimum of 8 bit values, which are named `bytes`. Other groupings are `short`, `int`, and `long`, which usually represent 16-bit, 32-bit, and 64-bit values, respectively. We also see the term `word`, that is used to describe the size of the maximum processing unit of the current mode of the CPU: so in 16-bit real mode, a `word` refers to a 16-bit value; in 32-bit protected mode, a `word` refers to a 32-bit value; and so on.

So, returning to the benefit of hexadecimal: strings of bits are rather long-winded to write out but are much easier to convert to and from the more shorthand hexadecimal notation than to and from our natural decimal system, essentially because we can break the conversion down into smaller, 4-bit segments of the binary number, rather than try to add up all of the component bits into a grand total, which gets much harder for larger bit strings (e.g. 16, 32, 64, etc.). This difficulty with decimal conversion is shown clearly by the example given in Figure 2.3.

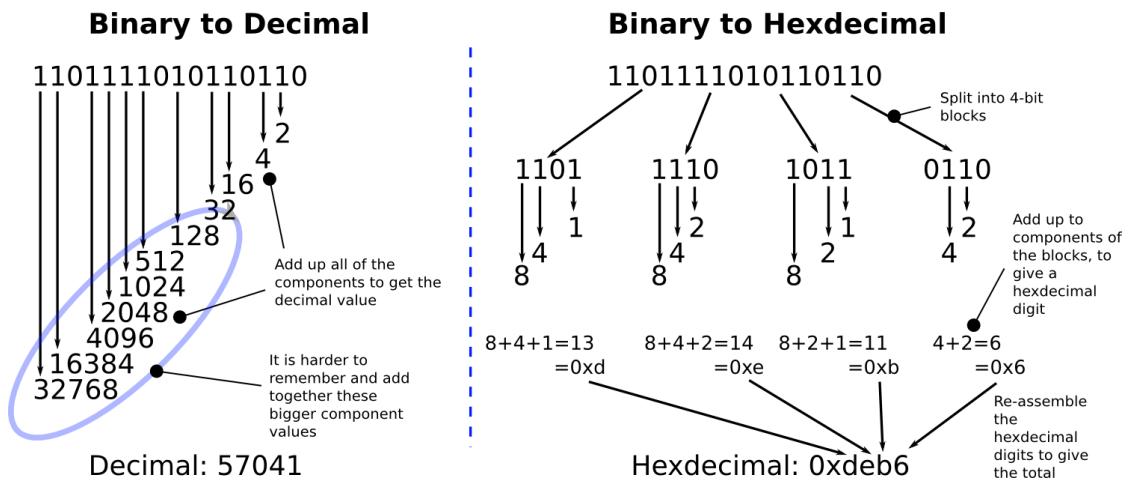


Figure 2.3: Conversion of `1101111010110110` to decimal and hexadecimal

Chapter 3

Boot Sector Programming (in 16-bit Real Mode)

Even with the example code provided, you will no doubt have found it frustrating writing machine code in a binary editor. You'd have to remember, or continuously reference, which of many possible machine codes cause the CPU to do certain functions. Luckily, you are not alone, and so *assemblers* have been written that translate more human friendly instructions into machine code for a particular CPU.

In this chapter we will explore some increasingly sophisticated boot sector programs to familiarise ourselves with assembly and the barren, pre-OS environment in which our programs will run.

3.1 Boot Sector Re-visited

Now, we will re-create the binary-edited boot sector from Section XXX instead using assembly language, so that we can really appreciate the value even of a very low-level language.

We can assemble this into actual machine code (a sequence of bytes that our CPU can interpret as instructions) as follows:

```
$nasm boot_sect.asm -f bin -o boot_sect.bin
```

Where `boot_sect.asm` is the file into which we saved the source code in Figure 3.1 and `boot_sect.bin` is the assembled machine code that we can install as a boot sector on a disk.

Note that we used the `-f bin` option to instruct nasm to produce *raw* machine code, rather than a code package that has additional meta information for linking in other routines that we would expect to use when programming in a more typical operating system environment. We need none of that cruft. Apart from the low-level BIOS routines, we are the only software running on this computer now. We are the operating system now, albeit at this stage with nothing more to offer than an endless loop --- but we will soon build up from this.

```

;
; A simple boot sector program that loops forever.
;

loop:           ; Define a label, "loop", that will allow
; us to jump back to it, forever.

jmp loop       ; Use a simple CPU instruction that jumps
; to a new memory address to continue execution.
; In our case, jump to the address of the current
; instruction.

times 510-($-$$) db 0 ; When compiled, our program must fit into 512 bytes,
; with the last two bytes being the magic number,
; so here, tell our assembly compiler to pad out our
; program with enough zero bytes (db 0) to bring us to the
; 510th byte.

dw 0xaa55      ; Last two bytes (one word) form the magic number,
; so BIOS knows we are a boot sector.

```

Figure 3.1: A simple boot sector written in assembly language.

Rather than saving this to the boot sector of a floppy disk and rebooting our machine, we can conveniently test this program by running Bochs:

`$bochs`

Or, depending on our preference and on availability of an emulator, we could use QEmu, as follows:

`$qemu boot_sect.bin`

Alternatively, you could load the image file into virtualisation software or write it onto some bootable medium and boot it from a real computer. Note that, when you write an image file to some bootable medium, that does not mean you add the file to the medium's file system: you must use an appropriate tool to write directly to the medium in a low-level sense (e.g. directly to the sectors of a disk).

If we'd like to see more easily exactly what bytes the assembler created, we can run the following command, which displays the binary contents of the file in an easy-to-read hexadecimal format:

`$od -t x1 -A n boot_sect.bin`

The output of this command should look familiar.

Congratulations, you just wrote a boot sector in assembly language. As we will see, all operating systems must start this way and then pull themselves up into higher level abstractions (e.g. higher level languages, such as C/C++)

3.2 16-bit Real Mode

CPU manufacturers must go to great lengths to keep their CPUs (i.e. their specific instruction set) compatible with earlier CPUs, so that older software, and in particular older operating systems, can still run on the most modern CPUs.

The solution implemented by Intel and compatible CPUs is to *emulate* the oldest CPU in the family: the Intel *8086*, which had support for 16-bit instructions and no notion of *memory protection*: memory protection is crucial for the stability of modern operating systems, since it allows an operating system to restrict a user's process from accessing, say, kernel memory, which, whether done accidentally or on purpose, could allow such a process to circumvent security mechanisms or even bring down the whole system.

So, for backward compatibility, it is important that CPUs boot initially in *16-bit real mode*, requiring modern operating systems explicitly to switch up into the more advanced 32-bit (or 64-bit) protected mode, but allowing older operating systems to carry on, blissfully unaware that they are running on a modern CPU. Later on, we will look at this important step from 16-bit real mode into 32-bit protected mode in detail.

Generally, when we say that a CPU is 16-bit, we mean that its instructions can work with a maximum of 16-bits at once, for example: a 16-bit CPU will have a particular instruction that can add two 16-bit numbers together in one CPU cycle; if it was necessary for a process to add together two 32-bit numbers, then it would take more cycles, that make use of 16-bit addition.

First we will explore this 16-bit real mode environment, since all operating systems must begin here, then later we will see how to switch into 32-bit protected mode and the main benefits of doing so.

3.3 Erm, Hello?

Now we are going to write a *seemingly* simple boot sector program that prints a short message on the screen. To do this we will have to learn some fundamentals of how the CPU works and how we can use BIOS to help us to manipulate the screen device.

Firstly, let's think about what we are trying to do here. We'd like to print a character on the screen but we do not know exactly how to communicate with the screen device, since there may be many different kinds of screen devices and they may have different interfaces. This is why we need to use BIOS, since BIOS has already done some auto detection of the hardware and, evidently by the fact that BIOS earlier printed information on the screen about self-testing and so on, so can offer us a hand.

So, next, we'd like to ask BIOS to print a character for us, but how do we ask BIOS to do that? There are no Java libraries for printing to the screen --- they are a dream away. We can be sure, however, that somewhere in the memory of the computer there will be some BIOS machine code that knows how to write to the screen. The truth is that we could possibly find the BIOS code in memory and execute it somehow, but this is more trouble than it is worth and will be prone to errors when there are differences between BIOS routine internals on different machines.

Here we can make use of a fundamental mechanism of the computer: *interrupts*.

3.3.1 Interrupts

Interrupts are a mechanism that allow the CPU temporarily to halt what it is doing and run some other, higher-priority instructions before returning to the original task. An interrupt could be raised either by a software instruction (e.g. `int 0x10`) or by some hardware device that requires high-priority action (e.g. to read some incoming data from a network device).

Each interrupt is represented by a unique number that is an index to the interrupt vector, a table initially set up by BIOS at the start of memory (i.e. at physical address `0x0`) that contains address pointers to *interrupt service routines* (ISRs). An ISR is simply a sequence of machine instructions, much like our boot sector code, that deals with a specific interrupt (e.g. perhaps to read new data from a disk drive or from a network card).

So, in a nutshell, BIOS adds some of its own ISRs to the interrupt vector that specialise in certain aspects of the computer, for example: interrupt `0x10` causes the screen-related ISR to be invoked; and interrupt `0x13`, the disk-related I/O ISR.

However, it would be wasteful to allocate an interrupt per BIOS routine, so BIOS multiplexes the ISRs by what we could imagine as a big `switch` statement, based usually on the value set in one of the CPUs general purpose registers, `ax`, prior to raising the interrupt.

3.3.2 CPU Registers

Just as we use variables in a higher level languages, it is useful if we can store data temporarily during a particular routine. All x86 CPUs have four general purpose *registers*, `ax`, `bx`, , and `dx`, for exactly that purpose. Also, these registers, which can each hold a *word* (two bytes, 16 bits) of data, can be read and written by the CPU with negligible delay as compared with accessing main memory. In assembly programs, one of the most common operations is moving (or more accurately, *copying*) data between these registers:

```
mov ax, 1234      ; store the decimal number 1234 in ax
mov cx, 0x234    ; store the hex number 0x234 in cx
mov dx, 't'       ; store the ASCII code for letter 't' in dx
mov bx, ax        ; copy the value of ax into bx, so now bx == 1234
```

Notice that the destination is the first and not second argument of the `mov` operation, but this convention varies with different assemblers.

Sometimes it is more convenient to work with single bytes, so these registers let us set their high and low bytes independently:

```
mov ax, 0          ; ax -> 0x0000, or in binary 0000000000000000
mov ah, 0x56       ; ax -> 0x5600
mov al, 0x23       ; ax -> 0x5623
mov ah, 0x16       ; ax -> 0x1623
```

[?]

3.3.3 Putting it all Together

So, recall that we'd like BIOS to print a character on the screen for us, and that we can invoke a specific BIOS routine by setting `ax` to some BIOS-defined value and then

triggering a specific interrupt. The specific routine we want is the BIOS scrolling teletype routine, which will print a single character on the screen and advance the cursor, ready for the next character. There is a whole list of BIOS routines published that show you which interrupt to use and how to set the registers prior to the interrupt. Here, we need interrupt `0x10` and to set `ah` to `0x0e` (to indicate tele-type mode) and `al` to the ASCII code of the character we wish to print.

```

;
; A simple boot sector that prints a message to the screen using a BIOS routine.
;

    mov ah, 0x0e    ; int 10/ah = 0eh -> scrolling teletype BIOS routine

    mov al, 'H'
    int 0x10
    mov al, 'e'
    int 0x10
    mov al, 'l'
    int 0x10
    mov al, 'l'
    int 0x10
    mov al, 'o'
    int 0x10

    jmp $           ; Jump to the current address (i.e. forever).

;
; Padding and magic BIOS number.
;

    times 510-($-$) db 0 ; Pad the boot sector out with zeros

    dw 0xaa55          ; Last two bytes form the magic number,
                        ; so BIOS knows we are a boot sector.

```

Figure 3.2:

Figure 3.2 shows the whole boot sector program. Notice how, in this case, we only needed to set `ah` once, then just changed `al` for different characters.

```

b4 0e b0 48 cd 10 b0 65 cd 10 b0 6c cd 10 b0 6c
cd 10 b0 6f cd 10 e9 fd ff 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa

```

Figure 3.3:

Just for completeness, Figure 3.3 shows the raw machine code of this boot sector. These are the actual bytes that are telling the CPU exactly what to do. If you are surprised by the amount of effort and understanding that is involved in writing such a barely --- if at all --- useful program, then remember that these instructions map very closely to the CPU's circuitry, so necessarily they are very simple, but also very fast. You are getting to know your computer now, as it really *is*.

3.4 Hello, World!

Now we are going to attempt a slightly more advanced version of the 'hello' program, that introduces a few more CPU fundamentals and an understanding of the landscape of memory into which our boot sector gets plonked by BIOS.

3.4.1 Memory, Addresses, and Labels

We said earlier how the CPU fetches and executes instructions from memory, and how it was BIOS that loaded our 512-byte boot sector into memory and then, having finished its initialisations, told the CPU to jump to the start of our code, whereupon it began executing our first instruction, then the next, then the next, etc.

So our boot sector code is somewhere in memory; but where? We can imagine the main memory as long sequence of bytes that can individually be accessed by an address (i.e. an index), so if we want to find out what is in the 54th byte of memory, then 54 is our address, which is often more convenient to express in hexadecimal: `0x36`.

So the start of our boot-sector code, the very first machine code byte, is at some address in memory, and it was BIOS that put us there. We might assume, unless we knew otherwise, that BIOS loaded our code at the start of memory, at address `0x0`. It's not so straightforward, though, because we know that BIOS has already been doing initialisation work on the computer long before it loaded our code, and will actually continue to service hardware interrupts for the clock, disk drives, and so on. So these BIOS routines (e.g. ISRs, services for screen printing, etc.) themselves must be stored somewhere in memory and must be preserved (i.e. not overwritten) whilst they are still of use. Also, we noted earlier that the interrupt vector is located at the start of memory, and were BIOS to load us there, our code would stomp over the table, and upon the next interrupt occurring, the computer will likely crash and reboot: the mapping between interrupt number and ISR would effectively have been severed.

As it turns out, BIOS likes always to load the boot sector to the address `0x7c00`, where it is sure will not be occupied by important routines. Figure 3.4 gives an example of the typical low memory layout of the computer when our boot sector has just been loaded [?]. So whilst we may instruct the CPU to write data to any address in memory, it may cause bad things to happen, since some memory is being used by other routines, such as the timer interrupt and disk devices.

3.4.2 'X' Marks the Spot

Now we are going to play a game called "find the byte", which will demonstrate memory referencing, the use of labels in assembly code, and the importance of knowing where BIOS loaded us to. We are going to write an assembly program that reserves a byte of

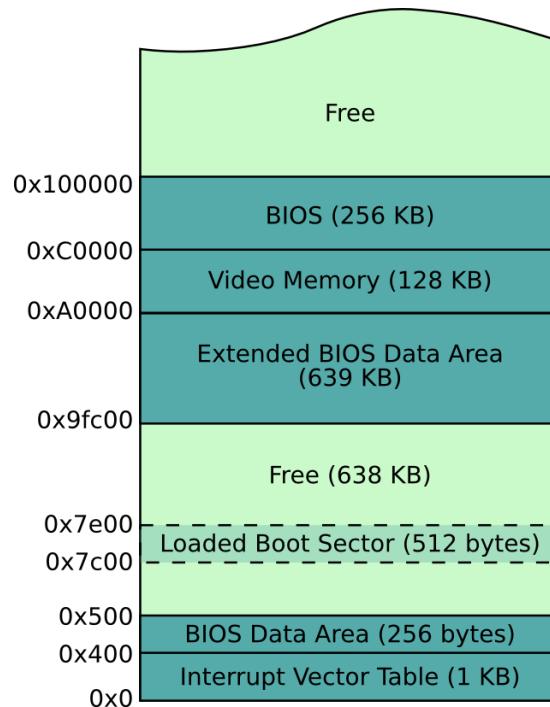


Figure 3.4: Typical lower memory layout after boot.

data for a character, then we will try to print out that character on the screen. To do this we need to figure out its absolute memory address, so we can load it into `al` and get BIOS to print it, as in the last exercise.

```

;
; A simple boot sector program that demonstrates addressing.
;
    mov ah, 0x0e          ; int 10/ah = 0eh -> scrolling teletype BIOS routine

    ; First attempt
    mov al, the_secret
    int 0x10              ; Does this print an X?

    ; Second attempt
    mov al, [the_secret]
    int 0x10              ; Does this print an X?

    ; Third attempt
    mov bx, the_secret
    add bx, 0x7c00
    mov al, [bx]
    int 0x10              ; Does this print an X?

    ; Fourth attempt

```

```

mov al, [0x7c1e]
int 0x10           ; Does this print an X?

jmp $              ; Jump forever.

the_secret:
db "X"

; Padding and magic BIOS number.

times 510-($-$) db 0
dw 0xaa55

```

Firstly, when we declare some data in our program, we prefix it with a label (`the_secret`). We can put labels anywhere in our programs, with their only purpose being to give us a convenient offset from the start of the code to a particular instruction or data.

```

b4 0e b0 1e cd 10 a0 1e 00 cd 10 bb 1e 00 81 c3
00 7c 8a 07 cd 10 a0 1e 7c cd 10 e9 fd ff 58 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa

```

Figure 3.5:

If we look at the assembled machine code in Figure 3.5, we can see that our 'X', which has an hexadecimal ASCII code `0x58`, is at an offset of 30 (`0x1e`) bytes from the start of the code, immediately before we padded the boot sector with zeros.

If we run the program we see that only the second two attempts succeed in printing an 'X'.

The problem with the first attempt is that it tries to load the immediate offset into `al` as the character to print, but actually we wanted to print the character *at* the offset rather than the offset itself, as attempted next, whereby the square brackets instruct the CPU to do this very thing - store the *contents* of an address.

So why does the second attempt fail? The problem is, that the CPU treats the offset as though it was from the start of memory, rather than the start address of our loaded code, which would land it around about in the interrupt vector. In the third attempt, we add the offset `the_secret` to the address that we believe BIOS to have loaded our code, `0x7c00`, using the CPU `add` instruction. We can think of `add` as the higher level language statement `bx = bx + 0x7c00`. We have now calculated the correct memory address of our 'X' and can store the contents of that address in `al`, ready for the BIOS print function, with the instruction `mov al, [bx]`.

In the fourth attempt we try to be a bit clever, by pre-calculating the address of the 'X' after the boot sector is loaded into memory by BIOS. We arrive at the address `0x7c1e` based on our earlier examination of the binary code (See Figure 3.5) which revealed that 'X' was `0x1e` (30) bytes from the start of our boot sector. This last example reminds

us why labels are useful, since without labels we would have to count offsets from the compiled code, and then update these when changes in code cause these offsets to change.

So now we have seen how BIOS does indeed load our boot sector to the address `0x7c00`, and we have also seen how addressing and assembly code labels are related.

It is inconvenient to always have to account for this label--memory offset in your code, so many assemblers will correct label references during assemblege if you include the following instruction at the top of your code, telling it exactly where you expect the code to loaded in memory:

```
[org 0x7c00]
```

Question 1

What do you expect will be printed now, when this `org` directive is added to this boot-sector program? For good marks, explain why this is so.

3.4.3 Defining Strings

Supposing you wanted to print a pre-defined message (e.g. “Booting OS”) to the screen at some point; how would you define such a string in your assembly program? We have to remind ourselves that our computer knows nothing about strings, and that a string is merely a sequence of data units (e.g. bytes, words, etc.) held somewhere in memory.

In the assembler we can define a string as follows:

```
my_string:
    db 'Booting OS'
```

We’ve actually already seen `db`, which translates to “declare byte(s) of data”, which tells the assembler to write the subsequent bytes directly to the binary output file (i.e. do not interpret them as processor instructions). Since we surrounded our data with quotes, the assembler knows to convert each character to its ASCII byte code. Note that, we often use a label (e.g. `my_string`) to mark the start of our data, otherwise we would have no easy way of referencing it within our code.

One thing we have overlooked in this example is that knowing how *long* a string is equally important as to knowing where it is. Since it is us that has to write all the code that handles strings, it is important to have a consistent strategy for knowing how long a string is. There are a few possibilities, but the convention is to declare strings as *null-terminating*, which means we always declare the last byte of the string as `0`, as follows:

```
my_string:
    db 'Booting OS',0
```

When later iterating through a string, perhaps to print each of its characters in turn, we can easily determine when we have reached the end.

3.4.4 Using the Stack

When on the topic of low-level computing, we often hear people talking about the *stack* like it is some special thing. The stack is really just a simple solution to the following inconvenience: the CPU has a limited number of registers for the temporary storage of our routine's local variables, but we often need more temporary storage than will fit into these registers; now, we can obviously make use of main memory, but specifying specific memory addresses when reading and writing is inconvenient, especially since we do not care exactly where the data is to be stored, only that we can retrieve it easily enough. And, as we shall see later, the stack is also useful for argument passing to realise function calls.

So, the CPU offers two instructions `push` and `pop` that allow us, respectively, to store a value and retrieve a value from the top of the stack, and so without worrying exactly where they are stored. Note, however, that we cannot push and pop single bytes onto and off the stack: in 16-bit mode, the stack works only on 16-bit boundaries.

The stack is implemented by two special CPU registers, `bp` and `sp`, which maintain the addresses of the stack base (i.e. the stack bottom) and the stack top respectively. Since the stack expands as we push data onto it, we usually set the stack's base far away from important regions of memory (e.g. such as BIOS code or our code) so there is no danger of overwriting if the stack grows too large. One confusing thing about the stack is that it actually grows *downwards* from the base pointer, so when we issue a `push`, the value actually gets stored below --- and not above --- the address of `bp`, and `sp` is decremented by the value's size.

The following boot sector program in Figure 3.6 demonstrates use of the stack.

Question 2

What will be printed in what order by the code in Figure 3.6? And at what absolute memory address will the ASCII character 'C' be stored? You may find it useful to modify the code to confirm your expectation, but be sure to explain *why* it is this address.

3.4.5 Control Structures

We'd never be comfortable using a programming language if we didn't know how to write some basic control structures, such as `if..then..elseif..else`, `for`, and `while`. These structures allow alternative branches of execution and form the basis of any useful routine.

After compilation, these high-level control structures reduce to simple jump statements. Actually, we've already seen the simplest example of loops:

```
some_label:
    jmp some_label ; jump to address of label
```

Or alternatively, with identical effect:

```
jmp $ ; jump to address of current instruction
```

So this instruction offers us an *unconditional* jump (i.e. it will *always* jump); but we often need to jump based on some condition (e.g. carry on looping *until we have looped ten times*, etc.).

```

;
; A simple boot sector program that demonstrates the stack.
;
    mov ah, 0x0e      ; int 10/ah = 0eh -> scrolling teletype BIOS routine
    mov bp, 0x8000    ; Set the base of the stack a little above where BIOS
    mov sp, bp        ; loads our boot sector - so it won't overwrite us.

    push 'A'          ; Push some characters on the stack for later
    push 'B'          ; retrieval. Note, these are pushed on as
    push 'C'          ; 16-bit values, so the most significant byte
                      ; will be added by our assembler as 0x00.

    pop bx           ; Note, we can only pop 16-bits, so pop to bx
    mov al, bl        ; then copy bl (i.e. 8-bit char) to al
    int 0x10          ; print(al)

    pop bx           ; Pop the next value
    mov al, bl        ; print(al)

    mov al, [0x7ffe] ; To prove our stack grows downwards from bp,
                      ; fetch the char at 0x8000 - 0x2 (i.e. 16-bits)
    int 0x10          ; print(al)

    jmp $             ; Jump forever.

; Padding and magic BIOS number.

times 510-($-$) db 0
dw 0xaa55

```

Figure 3.6: Manipulation of the stack, using `push` and `pop`

Conditional jumps are achieved in assembly language by first running a comparison instruction, then by issuing a specific conditional jump instruction.

```

    cmp ax, 4          ; compare the value in ax to 4
    je then_block      ; jump to then_block if they were equal
    mov bx, 45         ; otherwise, execute this code
    jmp the_end        ; important: jump over the 'then' block,
                      ; so we don't also execute that code.

then_block:
    mov bx, 23
the_end:

```

In a language such as C or Java, this would look like this:

```

if(ax == 4) {
    bx = 23;
} else {
    bx = 45;
}

```

We can see from the assembly example that there is something going on behind the scenes that is relating the `cmp` instruction to the `je` instruction it proceeds. This is an example of where the CPU's special `flags` register is used to capture the outcome of the `cmp` instruction, so that a subsequent conditional jump instruction can determine whether or not to jump to the specified address.

The following jump instructions are available, based on an earlier `cmp x, y` instruction:

```

je  target ; jump if equal          (i.e. x == y)
jne target ; jump if not equal    (i.e. x != y)
jl  target ; jump if less than    (i.e. x < y)
jle target ; jump if less than or equal (i.e. x <= y)
jg  target ; jump if greater than (i.e. x > y)
jge target ; jump if greater than or equal (i.e. x >= y)

```

Question 3

It's always useful to plan your conditional code in terms of a higher level language, then replace it with the assembly instructions. Have a go at converting this pseudo assembly code into full assembly code, using `cmp` and appropriate jump instructions. Test it with different values of `bx`. Fully comment your code, in your own words.

```

mov bx, 30

if (bx <= 4) {
    mov al, 'A'
} else if (bx < 40) {
    mov al, 'B'
} else {
    mov al, 'C'
}

mov ah, 0x0e      ; int=10/ah=0x0e -> BIOS tele-type output
int 0x10          ; print the character in al

jmp $

; Padding and magic number.
times 510-($-$) db 0
dw 0xaa55

```

3.4.6 Calling Functions

In high-level languages, we break big problems down into functions, which essentially are general purpose routines (e.g. print a message, write to a file, etc.) that we use over and over again throughout our program, usually changing parameters that we pass to the function to change the outcome in some way. At the CPU level a function is nothing more than a jump to the address of a useful routine then a jump back again to the instruction immediately following the first jump.

We can kind of simulate a function call like this:

```

...
...
mov al, 'H'          ; Store 'H' in al so our function will print it.

```

```

jmp my_print_function
return_to_here:           ; This label is our life-line so we can get back.
...
...

my_print_function:
    mov ah, 0x0e           ; int=10/ah=0x0e -> BIOS tele-type output
    int 0x10                ; print the character in al
    jmp return_to_here    ; return from the function call.

```

Firstly, note how we used the register `al` as a parameter, by setting it up ready for the function to use. This is how parameter passing is made possible in higher level languages, where the *caller* and *callee* must have some agreement on where and how many parameters will be passed.

Sadly, the main flaw with this approach is that we need to say explicitly where to return to after our function has been called, and so it will not be possible to call this function from arbitrary points in our program --- it will always return the same address, in this case the label `return_to_here`.

Borrowing from the parameter passing idea, the caller code could store the correct return address (i.e. the address immediately after the call) in some well-known location, then the called code could jump back to that stored address. The CPU keeps track of the current instruction being executed in the special register `ip` (instruction pointer), which, sadly, we cannot access directly. However, the CPU provides a pair of instructions, `call` and `ret`, which do exactly what we want: `call` behaves like `jmp` but additionally, before actually jumping, pushes the return address on to the stack; `ret` then pops the return address off the stack and jumps to it, as follows:

```

...
...
mov al, 'H'           ; Store 'H' in al so our function will print it.
call my_print_function
...
...

my_print_function:
    mov ah, 0x0e           ; int=10/ah=0x0e -> BIOS tele-type output
    int 0x10                ; print the character in al
    ret

```

Our functions are almost self-contained now, but there is a still an ugly problem that we will thank ourselves later for if we now take the trouble to consider it. When we call a function, such as a print function, within our assembly program, internally that function may alter the values of several registers to perform its job (indeed, with registers being a scarce resource, it will almost certainly do this), so when our program returns from the function call it may not be safe to assume, say, the value we stored in `dx` will still be there.

It is often sensible (and polite), therefore, for a function immediately to push any registers it plans to alter onto the stack and then pop them off again (i.e. restore the registers' original values) immediately before it returns. Since a function may use many of the general purpose registers, the CPU implements two convenient instructions, `pusha` and `popa`, that conveniently push and pop *all* registers to and from the stack respectively, for example:

```

...

```

```

...
some_function:
    pusha           ; Push all register values to the stack
    mov bx, 10
    add bx, 20
    mov ah, 0x0e    ; int=10/ah=0x0e -> BIOS tele-type output
    int 0x10        ; print the character in al
    popa            ; Restore original register values
    ret

```

3.4.7 Include Files

After slaving away even on the seemingly simplest of assembly routines, you will likely want to reuse your code in multiple programs. nasm allows you to include external files literally as follows:

```

%include "my_print_function.asm" ; this will simply get replaced by
                                ; the contents of the file

...
mov al, 'H'      ; Store 'H' in al so our function will print it.
call my_print_function

```

3.4.8 Putting it all Together

We now have enough knowledge about the CPU and assembly to write a more sophisticated “Hello, World” boot sector program.

Question 4

Put together all of the ideas in this section to make a self-contained function for printing null-terminated strings, that can be used as follows:

```

;
; A boot sector that prints a string using our function.
;
[org 0x7c00] ; Tell the assembler where this code will be loaded

    mov bx, HELLO_MSG ; Use BX as a parameter to our function, so
    call print_string ; we can specify the address of a string.

    mov bx, GOODBYE_MSG
    call print_string

    jmp $             ; Hang

%include "print_string.asm"

; Data
HELLO_MSG:
    db 'Hello, World!', 0 ; -- The zero on the end tells our routine

```

```

        ;      when to stop printing characters.

GOODBYE_MSG:
    db 'Goodbye!', 0

; Padding and magic number.
times 510-($-$$) db 0
dw 0xaa55

```

For good marks, make sure the function is careful when modifying registers and that you fully comment the code to demonstrate your understanding.

3.4.9 Summary

Still, it feels that we have not come very far. That's okay, and that's quite normal, given the primitive environment that we have been working in. If you have understood all up until here, then we are well on our way.

3.5 Nurse, Fetch me my Steth-o-scope

So far we have managed to get the computer to print out characters and strings that we have loaded into memory, but soon we will be trying to load some data from the disk, so it will be very helpful if we can display the hexadecimal values stored at arbitrary memory addresses, to confirm if we have indeed managed to load anything. Remember, we do not have the luxury of a nice development GUI, complete with a debugger that will let us carefully step though and inspect our code, and the best feedback the computer can give us when we make a mistake is visibly to do nothing at all, so we need to look after ourselves.

We have already written a routine to print out a string of characters, so we will now extend that idea into a hexadecimal printing routine --- a routine certainly to be cherished in this unforgiving, low-level world.

Let's think carefully about how we will do this, starting by considering how we'd like to use the routine. In a high-level language, we'd like something like this: `print_hex(0x1fb6)`, which would result in the string `'0x1fb6'` being printed on the screen. We have already seen, in Section XXX, how functions can be called in assembly and how we can use registers as parameters, so let's use the `dx` register as a parameter to hold the value we wish our `print_hex` function to print:

```

    mov dx, 0x1fb6 ; store the value to print in dx
    call print_hex ; call the function

; prints the value of DX as hex.
print_hex:
    ...
    ...
    ret

```

Since we are printing a string to the screen, we might as well re-use our earlier printing function to do the actual printing part, then our main task is to look at how we can build that string from the value in our parameter, `dx`. We definitely don't want to confuse matters more than we need to when working in assembly, so let's consider the following trick to get us started with this function. If we define the complete hexadecimal string as a sort of template variable in our code, as we defined our earlier “Hello, World” messages, we can simply get the string printing function to print it, then the task of our `print_hex` routine is to alter the components of that template string to reflect the hexadecimal value as ASCII codes:

```

mov dx, 0x1fb6 ; store the value to print in dx
call print_hex ; call the function

; prints the value of DX as hex.
print_hex:
; TODO: manipulate chars at HEX_OUT to reflect DX

mov bx, HEX_OUT ; print the string pointed to
call print_string ; by BX
ret

; global variables
HEX_OUT: db '0x0000',0

```

3.5.1 Question 5 (Advanced)

Complete the implementation of the `print_hex` function. You may find the CPU instructions `and` and `shr` to be useful, which you can find information about on the Internet. Make sure to fully explain your code with comments, in your own words.

3.6 Reading the Disk

We have now been introduced to BIOS, and have had a little play in the computer's low-level environment, but we have a little problem that poses to get in the way of our plan to write an operating system: BIOS loaded our boot code from the first sector of the disk, but that is *all* it loaded; what if our operating system code is larger --- and I'm guessing it will be --- than 512 bytes.

Operating systems usually don't fit into a single (512 byte) sector, so one of the first things they must do is bootstrap the rest of their code from the disk into memory and then begin executing that code. Luckily, as was hinted at earlier, BIOS provides routines that allow us to manipulate data on the drives.

3.6.1 Extended Memory Access Using Segments

When the CPU runs in its initial 16-bit real mode, the maximum size of the registers is 16 bits, which means that the highest address we can reference in an instruction is `0xffff`, which amounts by today's standards to a measly 64 KB (65536 bytes). Now, perhaps the likes of our intended simple operating system would not be affected by this limit,

but a day-to-day operating systems would never sit comfortably in such a tight box, so it is important that we understand the solution, of segmentation, to this problem.

To get around this limitation, the CPU designers added a few more special registers, `cs`, `ds`, `ss`, and `es`, called *segment* registers. We can imagine main memory as being divided into *segments* that are indexed by the segment registers, such that, when we specify a 16-bit address, the CPU automatically calculates the absolute address as the appropriate segment's start address offset by our specified address [?]. By *appropriate segment*, I mean that, unless explicitly told otherwise, the CPU will offset our address from the segment register appropriate for the context of our instruction, for example: the address used in the instruction `mov ax, [0x45ef]` would by default be offset from the *data segment*, indexed by `ds`; similarly, the *stack segment*, `ss`, is used to modify the actual location of the stack's base pointer, `bp`.

The most confusing thing about segment addressing is that adjacent segments overlap almost completely but for 16 bytes, so different segment and offset combinations can actually point to the same physical address; but enough of the talk: we won't truly grasp this concept until we've seen some examples.

To calculate the absolute address the CPU multiplies the value in the segment register by 16 and then adds your offset address; and because we are working with hexadecimal, when we multiple a number by 16, we simply shift it a digit to the left (e.g. `0x42 * 16 = 0x420`). So if we set `ds` to `0x4d` and then issue the statement `mov ax, [0x20]`, the value stored in `ax` will actually be loaded from address `0x4d0` ($16 * 0x4d + 0x20$).

Figure 3.7 shows how we can set `ds` to achieve a similar correction of label addressing as when we used the `[org 0x7c00]` directive in Section XXX. Because we do not use the `org` directive, the assmebler does not offset our labels to the correct memory locations when the code is loaded by BIOS to the address `0x7c00`, so the first attempt to print an 'X' will fail. However, if we set the data segment register to `0x7c0`, the CPU will do this offset for us (i.e. `0x7c0 * 16 + the_secret`), and so the second attempt will correctly print the 'X'. In the third and fourth attempts we do the same, and get the same results, but instead explicitly state to the CPU which segment register to use when computing the physical address, using instead the general purpose segment register `es`.

Note that limitations of the CPU's circuitry (at least in 16-bit real mode) reveal themselves here, when seemingly correct instructions like `mov ds, 0x1234` are not actually possible: just because we can store a literal address directly into a general purpose register (e.g. `mov ax, 0x1234` or `mov cx, 0xdf`), it doesn't mean we can do the same with every type of register, such as segment registers; and so, as in Figure 3.7, we must take an additional step to transfer the value via a general purpose register.

So, segment-based addressing allows us to reach further into memory, up to a little over 1 MB (`0xffff * 16 + 0xffff`). Later, we will see how more memory can be accessed, when we switch to 32-bit protected mode, but for now it suffices for us to understand 16-bit real mode segment-based addressing.

3.6.2 How Disk Drives Work

Mechanically, hard disk drives contain one or more stacked platters that spin under a read/write head, much like an old record player, only potentially, to increase capacity, with several records stacked one above the other, where a head moves in and out to get coverage of the whole of a particular spinning platter's surface; and since a particular platter may be readable and writable on both of its surfaces, one read/write head may

```

;
; A simple boot sector program that demonstrates segment offsetting
;
    mov ah, 0x0e          ; int 10/ah = 0eh -> scrolling teletype BIOS routine
    mov al, [the_secret]
    int 0x10              ; Does this print an X?

    mov bx, 0x7c0          ; Can't set ds directly, so set bx
    mov ds, bx              ; then copy bx to ds.
    mov al, [the_secret]
    int 0x10              ; Does this print an X?

    mov al, [es:the_secret] ; Tell the CPU to use the es (not ds) segment.
    int 0x10              ; Does this print an X?

    mov bx, 0x7c0
    mov es, bx
    mov al, [es:the_secret]
    int 0x10              ; Does this print an X?

    jmp $                  ; Jump forever.

the_secret:
    db "X"

; Padding and magic BIOS number.
times 510-($-$$) db 0
dw 0xaa55

```

Figure 3.7: Manipulating the data segment with the `ds` register.

float above and another below it. Figure 3.8 shows the inside of a typical hard disk drive, with the stack of platters and heads exposed. Note that the same idea applies to floppy disk drives, which, instead of several stacked hard platters, usually have a single, two-sided floppy disk medium.

The metallic coating of the platters give them the property that specific areas of their surface can be magnetised or demagnetised by the head, effectively allowing any state to be recorded permanently on them [?]. It is therefore important to be able to describe the exact place on the disk's surface where some state is to be read or written, and so Cylinder-Head-Sector (CHS) addressing is used, which effectively is a 3D coordinate system (see Figure 3.9):

- Cylinder: the cylinder describes the head's discrete distance from the outer edge of the platter and is so named since, when several platters are stacked up, you can visualise that all of the heads select a cylinder through all of the platters
- Head: the head describes which track (i.e. which specific platter surface within the cylinder) we are interested in.
- Sector: the circular track is divided into sectors, usually of capacity 512 bytes, which can be referenced with a sector index.



Figure 3.8: Inside of a hard disk drive

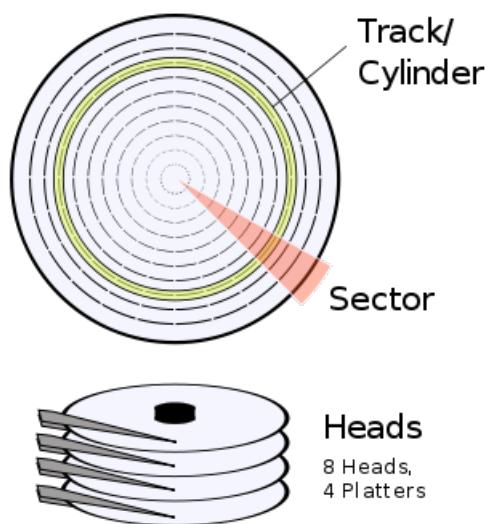


Figure 3.9: Cylinder, Head, Sector structure of a hard disk.

3.6.3 Using BIOS to Read the Disk

As we will see a little later on, specific devices require specific routines to be written to use them, so, for example, a floppy disk device requires us to explicitly turn on and off the motor that spins the disk under the read-and-write head before we can use it, whereas most hard disk devices have more functionality automated on local chips [?], but again the bus technologies with which such devices connect to the CPU (e.g. ATA/IDE, SATA, SCSI, USB, etc.) affect how we access them. Thankfully, BIOS can offer a few disk routines that abstract all of these differences for common disk devices.

The specific BIOS routine we are interested in here is accessed by raising interrupt `0x13` after setting the register `al` to `0x02`. This BIOS routine expects us to set up a few other registers with details of which disk device to use, which blocks we wish to read from the disk, and where to store the blocks in memory. The most difficult part of using this routine is that we must specify the first block to be read using a CHS addressing scheme; otherwise, it is just a case of filling in the expected registers, as detailed in the next code snippet.

```

mov ah, 0x02 ; BIOS read sector function

mov dl, 0      ; Read drive 0 (i.e. first floppy drive)
mov ch, 3      ; Select cylinder 3
mov dh, 1      ; Select the track on 2nd side of floppy
                ; disk, since this count has a base of 0
mov cl, 4      ; Select the 4th sector on the track - not
                ; the 5th, since this has a base of 1.
mov al, 5      ; Read 5 sectors from the start point

; Lastly, set the address that we'd like BIOS to read the
; sectors to, which BIOS expects to find in ES:BX
; (i.e. segment ES with offset BX).
mov bx, 0xa000 ; Indirectly set ES to 0xa000
mov es, bx
mov bx, 0x1234 ; Set BX to 0x1234
; In our case, data will be read to 0xa000:0x1234, which the
; CPU will translate to physical address 0xa1234

int 0x13       ; Now issue the BIOS interrupt to do the actual read.

```

Note that, for one reason or another (e.g. we indexed a sector beyond the limit of the disk, an attempt was made to read a faulty sector, the floppy disk was not inserted into the drive, etc.), BIOS may fail to read the disk for us, so it is important to know how to detect this; otherwise, we may *think* we have read some data but in fact the target address simply contains the same random bytes it did before we issued the read command. Fortunately for us, BIOS updates some registers to let us know what happened: the carry flag (`CF`) of the special `flags` register is set to signal a general fault, and `al` is set to the number of sectors *actually* read, as opposed to the number requested. After issuing the interrupt for the BIOS disk read, we can perform a simple check as follows:

```

...
...
int 0x13       ; Issue the BIOS interrupt to do the actual read.

jc disk_error ; jc is another jumping instruction, that jumps
                ; only if the carry flag was set.

; This jumps if what BIOS reported as the number of sectors

```

```

; actually read in AL is not equal to the number we expected.
cmp al, <no. sectors expected>
jne disk_error

disk_error :
    mov bx, DISK_ERROR_MSG
    call print_string
    jmp $

; Global variables
DISK_ERROR_MSG: db "Disk read error!", 0

```

3.6.4 Putting it all Together

As explained earlier, being able to read more data from the disk will be essential for bootstrapping our operating system, so here we will put all of the ideas from this section into a helpful routine that will simply read the first n sectors following the boot sector from a specified disk device.

```

; load DH sectors to ES:BX from drive DL
disk_load:
    push dx          ; Store DX on stack so later we can recall
                      ; how many sectors were request to be read,
                      ; even if it is altered in the meantime
    mov ah, 0x02      ; BIOS read sector function
    mov al, dh        ; Read DH sectors
    mov ch, 0x00      ; Select cylinder 0
    mov dh, 0x00      ; Select head 0
    mov cl, 0x02      ; Start reading from second sector (i.e.
                      ; after the boot sector)
    int 0x13          ; BIOS interrupt

    jc disk_error     ; Jump if error (i.e. carry flag set)

    pop dx            ; Restore DX from the stack
    cmp dh, al        ; if AL (sectors read) != DH (sectors expected)
    jne disk_error    ; display error message
    ret

disk_error :

    mov bx, DISK_ERROR_MSG
    call print_string
    jmp $

; Variables
DISK_ERROR_MSG  db "Disk read error!", 0

```

And to test this routine, we can write a boot sector program as follows:

```
; Read some sectors from the boot disk using our disk_read function
[org 0x7c00]

    mov [BOOT_DRIVE], dl ; BIOS stores our boot drive in DL, so it's
                          ; best to remember this for later.

    mov bp, 0x8000        ; Here we set our stack safely out of the
    mov sp, bp             ; way, at 0x8000

    mov bx, 0x9000        ; Load 5 sectors to 0x0000(ES):0x9000(BX)
    mov dh, 5              ; from the boot disk.
    mov dl, [BOOT_DRIVE]
    call disk_load

    mov dx, [0x9000]       ; Print out the first loaded word, which
    call print_hex          ; we expect to be 0xdada, stored
                           ; at address 0x9000

    mov dx, [0x9000 + 512] ; Also, print the first word from the
    call print_hex          ; 2nd loaded sector: should be 0xface

    jmp $

%include "../print/print_string.asm" ; Re-use our print_string function
%include "../hex/print_hex.asm"      ; Re-use our print_hex function
%include "disk_load.asm"
; Include our new disk_load function

; Global variables
BOOT_DRIVE: db 0

; Bootsector padding
times 510-($-$$) db 0
dw 0xaa55

; We know that BIOS will load only the first 512-byte sector from the disk,
; so if we purposely add a few more sectors to our code by repeating some
; familiar numbers, we can prove to ourselves that we actually loaded those
; additional two sectors from the disk we booted from.
times 256 dw 0xdada
times 256 dw 0xface
```

Chapter 4

Entering 32-bit Protected Mode

It would be nice to continue working in the 16-bit real mode with which we have now become much better acquainted, but in order to make fuller use of the CPU, and to better understand how developments of CPU architectures can benefit modern operating systems, namely memory protection in hardware, then we must press on into 32-bit protected mode.

The main differences in 32-bit protected mode are:

- Registers are extended to 32 bits, with their full capacity being accessed by pre-fixing an `e` to the register name, for example: `mov ebx, 0x274fe8fe`
- For convenience, there are two additional general purpose segment registers, `fs` and `gs`.
- 32-bit memory offsets are available, so an offset can reference a whopping 4 GB of memory (`0xffffffff`).
- The CPU supports a more sophisticated --- though slightly more complex --- means of memory segmentation, which offers two big advantages:
 - Code in one segment can be prohibited from executing code in a more privileged segment, so you can protect your kernel code from user applications
 - The CPU can implement *virtual memory* for user processes, such that *pages* (i.e. fixed-sized chunks) of a process's memory can be swapped transparently between the disk and memory on an as-needed basis. This ensures main memory is used efficiently, in that code or data that is rarely executed needn't hog valuable memory.
- Interrupt handling is also more sophisticated.

[?]

The most difficult part about switching the CPU from 16-bit real mode into 32-bit protected mode is that we must prepare a complex data structure in memory called the *global descriptor table* (GDT), which defines memory segments and their protected-mode attributes. Once we have defined the GDT, we can use a special instruction to load it

into the CPU, before setting a single bit in a special CPU control register to make the actual switch.

This process would be easy enough if we didn't have to define the GDT in assembly language, but sadly this low-level switch-over is unavoidable if we later wish to load a kernel that has been compiled from a higher-level language such as C, which usually will be compiled to 32-bit instructions rather than the less-efficient 16-bit instructions.

Oh, there is one shocker that I nearly forgot to mention: we can no longer use BIOS once switched into 32-bit protected mode. If you thought making BIOS calls was low-level. This is like one step backwards, two steps forwards.

4.1 Adapting to Life Without BIOS

It is true: in our quest to make full use of the CPU, we must abandon all of those helpful routines provided by BIOS. As we will see when we look in more detail at the 32-bit protected mode switch-over, BIOS routines, having been coded to work only in 16-bit real mode, are no longer valid in 32-bit protected mode; indeed, attempting to use them would likely crash the machine.

So what this means is that a 32-bit operating system must provide its own drivers for all hardware of the machine (e.g. the keyboard, screen, disk drives, mouse, etc). Actually, it is possible for a 32-bit protected mode operating system to switch temporarily back into 16-bit mode whereupon it may utilise BIOS, but this technique can be more trouble than it is worth, especially in terms of performance.

The first problem we will encounter in switching to protected mode is knowing how to print a message on the screen, so we can see what is happening. Previously we have asked BIOS to print an ASCII character on the screen, but how did that result in the appropriate pixels being highlighted at the appropriate position of our computer's screen? For now, it suffices to know that the display device can be configured into one of several resolutions in one of two modes, *text mode* and *graphics mode*; and that what is displayed on the screen is a visual representation of a specific range of memory. So, in order to manipulate the screen, we must manipulate the specific memory range that it is using in its current mode. The display device is an example of memory-mapped hardware because it works in this way.

When most computers boot, despite that they may in fact have more advanced graphics hardware, they begin in a simple Video Graphics Array (VGA) colour text mode with dimensions 80x25 characters. In text mode, the programmer does not need to render individual pixels to describe specific characters, since a simple font is already defined in the internal memory of the VGA display device. Instead, each character cell of the screen is represented by two bytes in memory: the first byte is the ASCII code of the character to be displayed, and the second byte encodes the character's attributes, such as the foreground and background colour and if the character should be blinking.

So, if we'd like to display a character on the screen, then we need to set its ASCII code and attributes at the correct memory address for the current VGA mode, which is usually at address `0xb8000`. If we slightly modify our original (16-bit real mode) `print_string` routine so that it no longer uses the BIOS routine, we can create a 32-bit protected mode routine that writes directly to video memory, as in Figure 4.1.

Note that, although the screen is displayed as columns and rows, the video memory is simply sequential. For example, the address of the column 5 on row 3 can be calculated as follows: `0xb8000 + 2 * (row * 80 + col)`

```

[bits 32]
; Define some constants
VIDEO_MEMORY equ 0xb8000
WHITE_ON_BLACK equ 0x0f

; prints a null-terminated string pointed to by EDX
print_string_pm:
    pusha
    mov edx, VIDEO_MEMORY ; Set edx to the start of vid mem.

print_string_pm_loop:
    mov al, [ebx]           ; Store the char at EBX in AL
    mov ah, WHITE_ON_BLACK ; Store the attributes in AH

    cmp al, 0               ; if (al == 0), at end of string, so
    je done                 ; jump to done

    mov [edx], ax           ; Store char and attributes at current
                           ; character cell.
    add ebx, 1               ; Increment EBX to the next char in string.
    add edx, 2               ; Move to next character cell in vid mem.

    jmp print_string_pm_loop ; loop around to print the next char.

print_string_pm_done :
    popa
    ret                    ; Return from the function

```

Figure 4.1: A routine for printing a string directly to video memory (i.e. without using BIOS).

The downside to our routine is that it always prints the string to the top-left of the screen, and so will overwrite previous messages rather than scrolling. We could spend time adding to the sophistication of this assembly routine, but let's not make things too hard for ourselves, since after we master the switch to protected mode, we will soon be booting code written in a higher level language, where we can make much lighter work of these things.

4.2 Understanding the Global Descriptor Table

It is important to understand the main point of this GDT, that is so fundamental to the operation of protected mode, before we delve into the details. Recall from Section XXX that the design rationale of segment-based addressing in the classical 16-bit real mode was to allow the programmer to access (albeit slightly, by today's standards) more memory than a 16-bit offset would allow. As an example of this, suppose that the programmer wanted to store the value of `ax` at the address `0x4fe56`. Without segment-

based addressing, the best the programmer could do is this:

```
mov [0xffff], ax
```

which falls way short of the intended address. Whereby, using a segment register, the task could be achieved as follows:

```
mov bx, 0x4000
mov es, bx
mov [es:0xfe56], ax
```

Although the general idea of segmenting memory and using offsets to reach into those segments has remained the same, the way that it is implemented in protected mode has completely changed, primarily to afford more flexibility. Once the CPU has been switched into 32-bit protected mode, the process by which it translates logical addresses (i.e. the combination of a segment register and an offset) to physical address is completely different: rather than multiply the value of a segment register by 16 and then add to it the offset, a segment register becomes an index to a particular *segment descriptor* (SD) in the GDT.

A segment descriptor is an 8-byte structure that defines the following properties of a protected-mode segment:

- Base address (32 bits), which defines where the segment begins in physical memory
- Segment Limit (20 bits), which defines the size of the segment
- Various flags, which affect how the CPU interprets the segment, such as the privilege level of code that runs within it or whether it is read- or write-only.

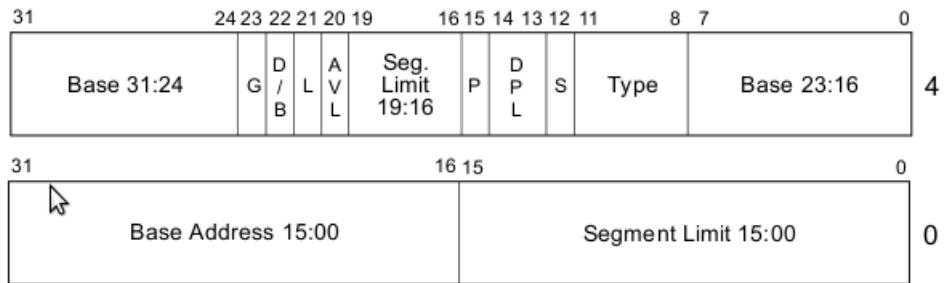
Figure 4.2 shows the actual structure of the segment descriptor. Notice how, just to add to the confusion, the structure fragments the base address and segment limit throughout the structure, so, for example, the lower 16 bits of the segment limit are in the first two bytes of the structure but the higher 4-bits are at the start of the seventh byte of the structure. Perhaps this was done as some kind of joke, or more likely it has historic roots or was influenced by the CPU's hardware design.

We will not concern ourselves with details of all of the possible configurations of segment descriptors, a full explanation of which is given in Intel's Developer Manual [?], but we will learn what we have to in order to get our code running in 32-bit protected mode.

The simplest workable configuration of segment registers is described by Intel as the *basic flat model*, whereby two overlapping segments are defined that cover the full 4 GB of addressable memory, one for *code* and the other for *data*. The fact that in this model these two segments overlap means that there is no attempt to protect one segment from the other, nor is there any attempt to use the paging features for virtual memory. It pays to keep things simple early on, especially since later we may alter the segment descriptors more easily once we have booted into a higher-level language.

In addition to the *code* and *data* segments, the CPU requires that the first entry in the GDT purposely be an invalid *null descriptor* (i.e. a structure of 8 zero bytes). The null descriptor is a simple mechanism to catch mistakes where we forget to set a particular segment register before accessing an address, which is easily done if we had some segment registers set to `0x0` and forgot to update them to the appropriate segment descriptors after switching to protected mode. If an addressing attempt is made with the null descriptor, then the CPU will raise an exception, which essentially is an interrupt ---

and which, although not too dissimilar as a concept, is not to be confused with exceptions in higher level languages such as Java.



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 4.2: Segment descriptor structure.

Our *code* segment will have the following configuration:

- Base: 0x0
- Limit: 0xfffff
- Present: 1, since segment is present in memory - used for virtual memory
- Privilege: 0, ring 0 is the highest privilege
- Descriptor type: 1 for code or data segment, 0 is used for traps
- Type:
 - Code: 1 for code, since this is a code segment
 - Conforming: 0, by not setting it means code in a segment with a lower privilege may not call code in this segment - this is a key to memory protection
 - Readable: 1, 1 if readable, 0 if execute-only. Readable allows us to read constants defined in the code.
 - Accessed: 0 This is often used for debugging and virtual memory techniques, since the CPU sets the bit when it accesses the segment
- Other flags

- Granularity: 1, if set, this multiplies our limit by 4 K (i.e. $16*16*16$), so our 0xffff would become 0xfffff000 (i.e. shift 3 hex digits to the left), allowing our segment to span 4 Gb of memory
- 32-bit default: 1, since our segment will hold 32-bit code, otherwise we'd use 0 for 16-bit code. This actually sets the default data unit size for operations (e.g. push 0x4 would expand to a 32-bit number ,etc.)
- 64-bit code segment: 0, unused on 32-bit processor
- AVL: 0, We can set this for our own uses (e.g. debugging) but we will not use it

Since we are using a simple flat model, with two overlapping code and data segments, the *data* segment will be identical but for the type flags:

- Code: 0 for data
- Expand down: 0 . This allows the segment to expand down - TODO explain this
- Writable: 1. This allows the data segment to be written to, otherwise it would be read only
- Accessed: 0 This is often used for debugging and virtual memory techniques, since the CPU sets the bit when it accesses the segment

Now that we have seen an actual configuration of two segments, exploring most of the possible segment descriptor settings, it should be clearer how protected mode offers much more flexibility in the partitioning of memory than real mode.

4.3 Defining the GDT in Assembly

Now that we understand what segment descriptors to include in our GDT for the basic flat model, let us look at how we might actually represent the GDT in assembly, a task that requires more patience than anything else. Whilst you're experiencing the shear tediousness of this, keep in mind the significance of it: what we do here will allow us soon to boot our operating system kernel, which we will write in a higher level language, then --- for want of a better quote --- our small steps will turn into giant leaps.

We have already seen examples of how to define data within our assembly code, using the `db`, `dw`, and `dd` assembly directives, and these are exactly what we must use to put in place the appropriate bytes in the segment descriptor entries of our GDT.

Actually, for the simple reason that the CPU needs to know how long our GDT is, we don't actually directly give the CPU the start address of our GDT but instead give it the address of a much simpler structure called the GDT descriptor (i.e. something that describes the GDT). The GDT is a 6-byte structure containing:

- GDT size (16 bits)
- GDT address (32 bits)

Note, when working in such a low-level language with complex data structures like these, we cannot add *enough* comments. The following code defines our GDT and the GDT descriptor; in the code, notice how we use `db`, `dw`, etc. to fill out parts of the

structure and how the flags are more conveniently defined using literal binary numbers, that are suffixed with `b`:

```

; GDT
gdt_start:

gdt_null: ; the mandatory null descriptor
    dd 0x0 ; 'dd' means define double word (i.e. 4 bytes)
    dd 0x0

gdt_code: ; the code segment descriptor
    ; base=0x0, limit=0xfffff,
    ; 1st flags: (present)1 (privilege)00 (descriptor type)1 -> 1001b
    ; type flags: (code)1 (conforming)0 (readable)1 (accessed)0 -> 1010b
    ; 2nd flags: (granularity)1 (32-bit default)1 (64-bit seg)0 (AVL)0 -> 1100b
    dw 0xffff ; Limit (bits 0-15)
    dw 0x0 ; Base (bits 0-15)
    db 0x0 ; Base (bits 16-23)
    db 10011010b ; 1st flags, type flags
    db 11001111b ; 2nd flags, Limit (bits 16-19)
    db 0x0 ; Base (bits 24-31)

gdt_data: ;the data segment descriptor
    ; Same as code segment except for the type flags:
    ; type flags: (code)0 (expand down)0 (writable)1 (accessed)0 -> 0010b
    dw 0xffff ; Limit (bits 0-15)
    dw 0x0 ; Base (bits 0-15)
    db 0x0 ; Base (bits 16-23)
    db 10010010b ; 1st flags, type flags
    db 11001111b ; 2nd flags, Limit (bits 16-19)
    db 0x0 ; Base (bits 24-31)

gdt_end: ; The reason for putting a label at the end of the
          ; GDT is so we can have the assembler calculate
          ; the size of the GDT for the GDT descriptor (below)

; GDT description
gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; Size of our GDT, always less one
                                ; of the true size
    dd gdt_start ; Start address of our GDT

; Define some handy constants for the GDT segment descriptor offsets, which
; are what segment registers must contain when in protected mode. For example,
; when we set DS = 0x10 in PM, the CPU knows that we mean it to use the
; segment described at offset 0x10 (i.e. 16 bytes) in our GDT, which in our
; case is the DATA segment (0x0 -> NULL; 0x08 -> CODE; 0x10 -> DATA)
CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start

```

4.4 Making the Switch

Once both the GDT and the GDT descriptor have been prepared within our boot sector, we are ready to instruct the CPU to switch from 16-bit real mode into 32-bit protected mode.

Like I said before, the actual switchover is fairly straight forward to code, but it is important to understand the significance of the steps involved.

The first thing we have to do is disable interrupts using the `cli` (clear interrupt) instruction, which means the CPU will simply ignore any future interrupts that may happen, at least until interrupts are later enabled. This is very important, because, like segment based addressing, interrupt handling is implemented completely differently in protected mode than in real mode, making the current IVT that BIOS set up at the start of memory completely meaningless; and even if the CPU could map interrupt signals to their correct BIOS routines (e.g. when the user pressed a key, store its value in a buffer), the BIOS routines would be executing 16-bit code, which will have no concept of the 32-bit segments we defined in our GDT and so will ultimately crash the CPU by having segment register values that assume the 16-bit real mode segmenting scheme.

The next step is to tell the CPU about the GDT that we just prepared --- with great pain. We use a single instruction to do this, to which we pass the GDT descriptor:

```
lgdt [gdt_descriptor]
```

Now that all is in-place, we make the actual switch over, by setting the first bit of a special CPU control register, `cr0`. Now, we cannot set that bit directly on the register, so we must load it into a general purpose register, set the bit, then store it back into `cr0`. Similarly to how we used the `and` instruction in Section XXX to exclude bits from a value, we can use the `or` instruction to include certain bits into a value (i.e. without disturbing any other bits that, for some important reason, may have been set already in the control register) [?].

```
mov eax, cr0      ; To make the switch to protected mode, we set
or eax, 0x1       ; the first bit of CR0, a control register
mov cr0, eax      ; Update the control register
```

After `cr0` has been updated, the CPU is in 32-bit protected mode [?].

That last statement is not entirely true, since modern processors use a technique called *pipelining*, that allows them to process different stages of an instruction's execution in parallel (and I am talking about single CPUs as opposed to parallel CPUs), and therefore in less time. For example, each instruction might be *fetched* from memory, *decoded* into microcode instructions, *executed*, then perhaps the result is *stored* back to memory; and since these stages are semi-independent, they could all be done within the same CPU cycle but within different circuitry (e.g. the previous instruction could be *decoded* whilst the next is *fetched*) [?].

We do not normally need to worry about CPU internals such as pipelining when programming the CPU, but switching CPU modes is a special case, since there is a risk that the CPU may process some stages of an instruction's execution in the wrong mode. So what we need to do, immediately after instructing the CPU to switch mode, is to force the CPU to finish any jobs in its pipeline, so that we can be confident that all future instructions will be executed in the correct mode.

Now, pipelining works very well when the CPU knows about the next few instructions that will be coming over the horizon, since it can pre-fetch them, but it doesn't like instructions such as `jmp` or `call`, because until those instructions have been executed fully the CPU can have no idea about the instructions that will follow them, especially if we use a *far* jump or call, which means that we jump to another segment. So immediately after instructing the CPU to switch mode, we can issue a far jump, which will force the

CPU to flush the pipeline (i.e. complete all of instructions currently in different stages of the pipeline).

To issue a far jump, as opposed to a near (i.e. standard) jump, we additionally provide the target segment, as follows:

```
jmp <segment>:<address offset>
```

For this jump, we need to think carefully about where we wish to land. Suppose we set up a label in our code such as `start_protected_mode` that marks the beginning of our 32-bit code. As we have just discussed, a *near* jump, such as `jmp start_protected_mode` may not be sufficient to flush the pipeline, and, besides we are now in some strange limbo, since our current code segment (i.e. `cs`) will not be valid in protected mode. So, we must update our `cs` register to the offset of the code segment descriptor of our GDT. Since the segment descriptors are each 8 bytes long, and since our code descriptor was the second descriptor in our GDT (the null descriptor was the first), its offset will be `0x8`, and so this value is what we must now set our code segment register to. Note that, by the very definition of a far jump, it will automatically cause the CPU to update our `cs` register to the target segment. Making handy use of labels, we got our assembler to calculate these segment descriptor offsets and store them as the constants `CODE_SEG` and `DATA_SEG`, so now we arrive at our jump instruction:

```
jmp CODE_SEG:start_protected_mode
[bits 32]
start_protected_mode:
... ; By now we are assuredly in 32-bit protected mode.
...
```

Note that, in fact, we don't need to jump very far at all in terms of the physical distance between the where we jumped from and where we landed, but the importance was in how we jump.

Note also that we need to use the `[bits 32]` directive to tell our assembler that, from that point onwards, it should encode in 32-bit mode instructions. Note, though, that this does not mean we cannot use 32-bit instructions in 16-bit real mode, just that the assembler must encode those instructions slightly differently than in 32-bit protected mode [?]. Indeed, when switching to protected mode, we made use of the 32-bit register `eax` to set the control bit.

Now we are in 32-bit protected mode. A good thing to do once we have entered 32-bit mode proper is to update all of the other segment registers so they now point to our 32-bit data segment (rather than the now-invalid real mode segments) and update the position of the stack.

We can combine the whole process into a re-usable routine, as in Figure XXX.

```
[bits 16]
; Switch to protected mode
switch_to_pm:
    cli
    ; We must switch off interrupts until we have
    ; set-up the protected mode interrupt vector
    ; otherwise interrupts will run riot.
```

```

lgdt [gdt_descriptor] ; Load our global descriptor table, which defines
                      ; the protected mode segments (e.g. for code and data)

mov eax, cr0          ; To make the switch to protected mode, we set
or eax, 0x1           ; the first bit of CR0, a control register
mov cr0, eax

jmp CODE_SEG:init_pm ; Make a far jump (i.e. to a new segment) to our 32-bit
                      ; code. This also forces the CPU to flush its cache of
                      ; pre-fetched and real-mode decoded instructions, which can
                      ; cause problems.

[bits 32]
; Initialise registers and the stack once in PM.
init_pm:

mov ax, DATA_SEG      ; Now in PM, our old segments are meaningless,
mov ds, ax            ; so we point our segment registers to the
mov ss, ax            ; data selector we defined in our GDT
mov es, ax
mov fs, ax
mov gs, ax

mov ebp, 0x90000      ; Update our stack position so it is right
mov esp, ebp          ; at the top of the free space.

call BEGIN_PM         ; Finally, call some well-known label

```

4.5 Putting it all Together

Finally, we can include all of our routines into a boot sector that demonstrates the switch from 16-bit real mode into 32-bit protected mode.

```

; A boot sector that enters 32-bit protected mode.
[org 0x7c00]

mov bp, 0x9000          ; Set the stack.
mov sp, bp

mov bx, MSG_REAL_MODE
call print_string

call switch_to_pm        ; Note that we never return from here.

jmp $

%include "../print/print_string.asm"
%include "gdt.asm"
%include "print_string_pm.asm"
%include "switch_to_pm.asm"

[bits 32]

```

```
; This is where we arrive after switching to and initialising protected mode.
BEGIN_PM:

    mov ebx, MSG_PROT_MODE
    call print_string_pm    ; Use our 32-bit print routine.

    jmp $                  ; Hang.

; Global variables
MSG_REAL_MODE  db "Started in 16-bit Real Mode", 0
MSG_PROT_MODE  db "Successfully landed in 32-bit Protected Mode", 0

; Bootsector padding
times 510-($-$$) db 0
dw 0xaa55
```

Chapter 5

Writing, Building, and Loading Your Kernel

So far, we have learnt a lot about how the computer really works, by communicating with it in the low-level assembly language, but we've also seen how it can be very slow to progress in such a language: we need to think very carefully even about the simplest of control structures (e.g. `if (<something>) <do this> else <do that>`), and we have to worry about how best to make use of the limited number of registers, and juggle with the stack. Another drawback of us continuing in assembly language is that it is closely tied to the specific CPU architecture, and so it would be harder for us to port our operating system to another CPU architecture (e.g. ARM, RISC, PowerPC).

Luckily, other programmers also got fed up of writing in assembly, so decided to write higher-level language compilers (e.g. FORTRAN, C, Pascal, BASIC, etc.), that would transform more intuitive code into assembly language. The idea of these compilers is to map higher level constructs, such as control structures and function calls onto assembly template code, and so the downside --- and there usually always is a downside --- is that the generic templates may not always be optimal for specific functionality. Without further ado, let us look at how C code is transformed into assembly to demystify the role of the compiler.

5.1 Understanding C Compilation

Let's write some small code snippets in C and see what sort of assembly code they generate. This is a great way of learning about how C works.

5.1.1 Generating Raw Machine Code

```
// Define an empty function that returns an integer
```

```
int my_function() {
    return 0xbaba;
}
```

Save the code in Figure XXXX into a file called `basic.c`, and compile it as follows:

```
$gcc -ffreestanding -c basic.c -o basic.o
```

This will produce an *object file*, which, being completely unrelated, is not to be confused with the concept of object-oriented programming. Rather than compiling directly into machine code, the compiler outputs *annotated* machine code, where meta information, such as textual labels, that are redundant for execution, remain present to enable more flexibility in how the code is eventually put together. One big advantage of this intermediary format is that the code may be easily relocated into a larger binary file when linked in with routines from other routines in other libraries, since code in the object file uses relative rather than absolute internal memory references. We can get a good view of an object file's contents with the following command:

```
$objdump -d basic.o
```

The output of this command will give something like that in Figure XXX. Note that we can see some assembly instructions and some additional details about the code. Note that the syntax of the assembly is very slightly different to that used by nasm, so simply ignore this part, since we will soon see it in a more familiar format.

```
basic.o:      file format elf32-i386

Disassembly of section .text:
00000000 <my_function>:
 0: 55                      push   %ebp
 1: 89 e5                   mov    %esp,%ebp
 3: b8 ba ba 00 00          mov    $0xbaba,%eax
 8: 5d                      pop    %ebp
 9: c3                      ret
```

In order to create the actual executable code (i.e. that will run on our CPU), we have to use a *linker*, whose role is to link together all of the routines described in the input object files into one executable binary file, effectively stitching them together and converting those relative addresses into absolute addresses within the aggregated machine code, for example: `call <function_X_label>` will become `call 0x12345`, where `0x12345` is the offset within the output file that the linker decided to place the code for the routine denoted by `function_X_label`.

In our case, though, we do not want to link with any routines from any other object files --- we will look at this shortly --- but nevertheless the linker will convert our annotated machine code file into a raw machine code file. To output raw machine code into a file `basic.bin`, we can use the following command:

```
$ld -o basic.bin -Ttext 0x0 --oformat binary basic.o
```

Note that, like the compiler, the linker can output executable files in various formats, some of which may retain meta data from the input object files. This is useful for executables that are hosted by an operating system, such as the majority of programs we will write on a platform such as Linux or Windows, since meta data can be retained to describe how those applications are to be loaded into memory; and for debugging purposes, for example: the information that a process crashed at instruction address **0x12345678** is far less useful to a programmer than information presented using redundant, non-executable meta-data that a process crashed in function **my_function**, file **basic.c**, on line **3**.

Anyhow, since we are interested in writing an operating system, it would be no good trying to run machine code intermingled with meta data on our CPU, since unaware the CPU will execute every byte as machine code. This is why we specify an output format of (raw) **binary**.

The other option we used was **-Ttext 0x0**, which works in the same way as the **org** directive we used in our earlier assembly routines, by allowing us to tell the compiler to offset label addresses in our code (e.g. for any data we specify in the code, such as strings like **“Hello, World”**) to their absolute memory addresses when later loaded to a specific origin in memory. For now this is not important, but when we come to load kernel code into memory, it is important that we set this to the address we plan to load to.

Now we have successfully compiled the C code into a raw machine code file, that we could (once we have figured out how to load it) run on our CPU, so let's see what it looks like. Luckily, since assembly maps very closely to machine code instructions, if you are ever given a file containing only machine code, you can easily disassemble it to view it in assembly. Ah, yes; this is another benefit of understanding a little of assembly, because you can potentially reverse-engineer any software that lands on you lap minus the original source code, even more successfully if the developer left in some meta data for you --- which they nearly always do. The only problem with disassembling machine code is that some of those bytes may have been reserved as data but will show up as assembly instructions, though in our simple C program we didn't declare any data. To see what machine code the compiler actually generated from our C source code, run the following command:

```
$ndisasm -b 32 basic.bin > basic.dis
```

The **-b 32** simply tells the disassembler to decode to 32-bit assembly instructions, which is what our compiler generates. Figure XXX shows the assembly code generated by gcc for our simple C program.

00000000	55	push ebp
00000001	89E5	mov ebp,esp
00000003	B8BABA0000	mov eax,0xbaba
00000008	5D	pop ebp
00000009	C3	ret

So here it is: gcc generated some assembly code not too dissimilar to that which we have been writing ourselves already. The three columns output from the disassembler,

from left to right, show the file offsets of the instructions, the machine code, and the equivalent assembly instructions. Although our function does a very simple thing, there is some additional code in there that seems to be manipulating the stack's base and top registers, `ebp` and `esp`. C makes heavy use of the stack for storing variables that are local to a function (i.e. variables that are no-longer needed when the function returns), so upon entering a function, the stack's base pointer (`ebp`) is increased to the current top of the stack (`mov ebp, esp`), effectively creating a local, initially empty stack above the stack of the function that called our function. This process is often referred to as the function setting up its *stack frame*, in which it will allocate any local variables. However, if prior to returning from our function we failed to restore the stack frame to that originally set up by our caller, the calling function would get in a real mess when trying to access its local variables; so before updating the base pointer for our stack frame, we must store it, and there is no better place to store it than the top of the stack (`push ebp`).

After preparing our stack frame, which, sadly, doesn't actually get used in our simple function, we see how the compiler handles the line `return 0xbaba;`: the value `0xbaba` is stored in the 32-bit register `eax`, which is where the calling function (if there were one) would expect to find the returned value, similarly to how we had our own convention of using certain registers to pass arguments to our earlier assembly routines, for example: our `print_string` routine expected to find the address of the string to be printed in the register `bx`.

Finally, before issuing `ret` to return to the caller, the function pops the original stack base pointer off the stack (`pop ebp`), so the calling function will be unaware that its own stack frame was ever changed by the called function. Note that we didn't actually change the top of the stack (`esp`), since in this case our stack frame was used to store nothing, so the untouched `esp` register did not require restoring.

Now we have a good idea about how C code translates into assembly, so let's prod the compiler a little further until we have sufficient understanding to write a simple kernel in C.

[?]

5.1.2 Local Variables

Now write the code in Figure XXX into a file called `local_var.c` and compile, link, and disassemble it as before.

```
// Declare a local variable.
int my_function() {
    int my_var = 0xbaba;
    return my_var;
}
```

Now the compiler generates the assembly code in Figure XXX.

```

00000000 55          push ebp
00000001 89E5        mov ebp,esp
00000003 83EC10      sub esp,byte +0x10
00000006 C745FCBABA0000  mov dword [ebp-0x4],0xbaba
0000000D 8B45FC      mov eax,[ebp-0x4]
00000010 C9          leave
00000011 C3          ret

```

The only difference now is that we actually allocate a local variable, `my_var`, but this provokes an interesting response from the compiler. As before, the stack frame is established, but then we see `sub esp, byte +0x10`, which is subtracting 16 (`0x10`) bytes from the top of the stack. Firstly, we have to (constantly) remind ourselves that the stack grows downwards in terms of memory addresses, so in simpler terms this instruction means, 'allocate another 16 bytes on the top of stack'. We are storing an `int`, which is a 4-byte (32-bit) data type, so why have 16 bytes been allocated on the stack for this variable, and why not use `push`, which allocates new stack space automatically? The reason the compiler manipulates the stack in this way is one of optimisation, since CPUs often operate less efficiently on a datatype that is not aligned on memory boundaries that are multiples of the datatype's size [?]. Since C would like all variables to be properly aligned, it uses the maximum datatype width (i.e. 16 bytes) for all stack elements, at the cost of wasting some memory.

The next instruction, `mov dword [ebp-0x4],0xbaba`, actually stores our variable's value in the newly allocated space on the stack, but without using `push`, for the previously given reason of stack efficiency (i.e. the size of the datatype stored is less than the stack space reserved). We understand the general use of the `mov` instruction, but two things that deserve some explanation here are the use of `dword` and `[ebp-0x4]`:

- `dword` states explicitly that we are storing a *double word* (i.e. 4 bytes) on the stack, which is the size of our `int` datatype. So the actual bytes stored would be `0x0000baba`, but without being explicit could easily be `0xbaba` (i.e. 2 bytes) or `0x000000000000baba` (i.e. 8 bytes), which, although the same value, have different ranges.
- `[ebp-0x4]` is an example of a modern CPU shortcut called *effective address computation* [?], which is more impressive than the assembly code seems to reflect. This part of the instruction references an address that is calculated *on-the-fly* by the CPU, based on the current address of register `ebp`. At a glance, we might think our assembler is manipulating a constant value, as it would if we wrote something like this `mov ax, 0x5000 + 0x20`, where our assembler would simply pre-process this into `mov ax, 0x5020`. But only once the code is run would the value of any register be known, so this definitely is not pre-processing; it forms a part of the CPU instruction. With this form of addressing the CPU is allowing us to do more per instruction cycle, and is a good example of how CPU hardware has adapted to better suit programmers. We could write the equivalent, without such address manipulation, less efficiently in the following three lines of code:

```

mov eax, ebp      ; EAX = EBP
sub eax, 0x4      ; EAX = EAX - 0x4
mov [eax], 0xbaba ; store 0xbaba at address EAX

```

So the value `0xbaba` is stored directly to the appropriate position of the stack, such that it will occupy the first 4 bytes above (though physically below, since the stack grows downwards) the base pointer.

Now, being a computer program, our compiler can distinguish different numbers as easily as we can distinguish different variable names, so what we think of as the variable `my_var`, the compiler will think of as the address `ebp-0x4` (i.e. the first 4 bytes of the stack). We see this in the next instruction, `mov eax, [ebp-0x4]`, which basically means, 'store the contents of `my_var` in `eax`', again using efficiently address computation; and we know from the previous function that `eax` is used to return a variable to the caller of our function.

Now, before the `ret` instruction, we see something new: the `leave` instruction. Actually, the `leave` instruction is an alternative to the following steps, that restore the original stack of the caller, reciprocal of the first two instruction of the function:

```
mov esp, ebp ; Put the stack back to as we found it.
pop ebp
```

Though only a single instruction, it is not always the case that `leave` is more efficient than the separate instructions [?]. Since our compiler chose to use this instruction, we will leave that particular discussion to other people.

5.1.3 Calling Functions

Not let's look at the C code in Figure XXX, which has two functions, where one function, `caller_function`, calls the other, `callee_function`, passing it an integer argument. The called function simply returns the argument it was passed.

```
void caller_function() {
    callee_function(0xdede);
}

int callee_function(int my_arg) {
    return my_arg;
}
```

If we compile and disassemble the C code, we will get something similar to that in Figure XXX.

```
00000000 55          push ebp
00000001 89E5        mov ebp,esp
00000003 83EC08      sub esp,byte +0x8
00000006 C70424DEDE0000  mov dword [esp],0xdede
0000000D E802000000  call dword 0x14
00000012 C9          leave
00000013 C3          ret
00000014 55          push ebp
00000015 89E5        mov ebp,esp
00000017 8B4508      mov eax,[ebp+0x8]
```

0000001A 5D	pop ebp
0000001B C3	ret

Firstly, notice how we can differentiate between assembly code of the two functions by looking for the tell-tale `ret` instruction that always appears as the last instruction of a function. Next, notice how the upper function uses the assembly instruction `call`, which we know is used to jump to another routine from which usually we expect to return. This must be our `caller_function`, that is calling `callee_function` at offset `0x14` of the machine code. The most interesting lines here are those immediately before the call, since they are somehow ensuring that the argument `my_arg` is passed to `callee_function`. After establishing its stack frame, as we have seen to be common to all functions, `caller_function` allocates 8 bytes on the top of the stack (`sub esp, byte +0x8`), then stores our passed value, `0xdede`, into that stack space (`mov dword [esp], 0xdede`).

So let's see how `callee_function` accesses that argument. From offset `0x14`, we see that `callee_function` establishes its stack frame as usual, but then look at what it stores in the `eax` register, a register that we know from our earlier analysis is used to hold a function's return value: it stores the contents of address `[ebp + 0x8]`. Here we have to remind ourselves again of that confusing fact that the stack grows downwards in memory, so in terms of logically-more-sensible upward growing stack, `ebp + 0x8` is 8 bytes *below* our stack's base, so we are actually reaching into the stack frame of the function that called us to get the argument's value. This is what we'd expect, of course, because the caller put that value onto the top of their stack, then we put our stack base at the top of their stack to establish our stack frame.

It is very useful to know the calling convention used by any high-level language compiler when interfacing its code in assembly. For example, the default calling convention of C is to push arguments onto the stack in reverse order, so the first argument is on the top of the stack. To mix up the order of arguments would certainly cause the program to perform incorrectly, and likely crash.

5.1.4 Pointers, Addresses, and Data

When working in a high-level language we can easily find ourselves forgetting about the fact that variables are simply references to allocated memory addresses, where sufficient space has been reserved to accomodate their particular data type. This is because, in most cases when we are dealing with variables, we are really only interested in the values that they hold, rather than where they reside in memory. Consider the following snippet of C code:

```
int a = 3;
int b = 4;
int total = a + b;
```

Now that we have more of an awareness about how the computer will actually perform these simple C instructions, we could make a well informed assumption that the instruction `int a = 3;` will involve two main steps: firstly, at least 4 bytes (32 bits) will be reserved, perhaps on the stack, to hold the value; then, the value `3` will be stored at the reserved address. The same would be the case for the second line. And in the line `int`

`total = a + b;`, some more space will be reserved for the variable `total`, and in it will be stored the addition of the *contents of addresses* pointed to by the labels `a` and `b`.

Now, suppose that we'd like to store a value at a specific address of memory; for example, like we have done in assembly, to write characters directly to the video memory at address `0xb8000` when BIOS was no longer available. How would we do that in C, when it seems that any value we wish to store must be in an address that has been determined by the compiler? Indeed, some high-level languages do not allow us to access memory in this way at all, which essentially is breaking the fluffy abstraction of the language. Luckily, C allows us to use *pointer* variables, that are datatypes used for storing addresses (rather than values), and which we can *dereference* to read or write data to wherever they point.

Now, technically, all pointer variables are the same datatype (e.g. a 32-bit memory address), but usually we plan to read and write specific datatypes from and to the address pointed to by a pointer, so we tell the compiler that, say, *this* is a pointer to a `char` and *that* is a pointer to an `int`. This is really a convenience, so that we do not always have to tell the compiler how many bytes it should read and write from the address held in a certain pointer. The syntax for defining and using pointers is shown in Figure XXX.

```
// Here, the star following the type means that this is not a variable to hold
// a char (i.e. a single byte) but a pointer to the ADDRESS of a char,
// which, being an address, will actually require the allocation of at least
// 32 bits.
char* video_address = 0xb8000;

// If we'd like to store a character at the address pointed to, we make the
// assignment with a star-prefixed pointer variable. This is known as
// dereferencing a pointer, because we are not changing the address held by
// the pointer variable but the contents of that address.
*video_address = 'X';

// Just to emphasise the purpose of the star, an omission of it, such as:
video_address = 'X';
// would erroneously store the ASCII code of 'X' in the pointer variable,
// such that it may later be interpreted as an address.
```

In C code we often see `char*` variables used for strings. Let's think about why this is. If we'd like to store a single `int` or `char`, then we know that they are both fixed sized datatypes (i.e. we know how many bytes they will use), but a string is an *array* of datatypes (usually of `char`), which may be of any length. So, since a single datatype cannot hold an entire string, only one element of it, we can use a pointer to a `char`, and set it to the memory address of the first character of the string. This is actually what we did in our assembly routines, such as `print_string`, where we allocated a string of characters (e.g. `“Hello, World”`) somewhere within our code, then, to print a particular string, we passed the first character's address via the `bx` register.

Let's look at an example of what the compiler does when we set up a string variable. In Figure XXX, we define a simple function that does nothing else other than allocate a string to a variable.

```
void my_function() {
    char* my_string = "Hello";
}
```

As before, we can disassemble to give something like that in Figure XXX.

```
00000000 55          push ebp
00000001 89E5        mov ebp,esp
00000003 83EC10      sub esp,byte +0x10
00000006 C745FA48656C6C  mov dword [ebp-0x6],0x6c6c6548
0000000D 66C745FE6FOO  mov word [ebp-0x2],0x6f
00000013 C9          leave
00000014 C3          ret
```

Firstly, to get our bearings we look for the `ret` instruction, that marks the end of the function. We see that the first two instructions of the function set the stack frame up, as usual. The next instruction, which we have also seen before, `sub esp,byte +0x10`, allocates 16 bytes on the stack to store our local variable. Now, the next instruction, `mov dword [ebp-0x4],0xf`, should have a familiar form, since it stores a value in our variable; but why does it store the number `0xf` --- we didn't tell it to do that, did we? After storing this suspicious value, we see the function politely revert the stack to the callers stack frame (`leave`) then return (`ret`). But look, there are five more instructions after the end of the function! What do you think the instruction `dec eax` is doing? Perhaps it decreases the value of `eax` by 1, but why? And what about the rest of the instructions?

At times like this we need to do a sanity check, and remember that: the disassembler cannot distinguish between code and data; and somewhere in that code must be data for the string we defined. Now, we know that our function consists of the first half of the code, since these instructions made sense to us, and they ended with `ret`. If we now assume that the rest of the code is in fact our data, then the suspicious value, `0xf`, that was stored in our variable makes sense, because it is the offset from the start of the code to where the data begins: our pointer variable is being set the the address of the data. To reassure our instincts, if we looked up in an ASCII table the character values of our string `'Hello'`, we would find them to be `0x48`, `0x65`, `0x6c`, `0x6c`, and `0x6f`. Now it is becoming clear, because if we look at the middle column of the disassembler output we see that these are the machine code bytes for those strange instructions that didn't seem to make sense; we see also that the very last byte is `0x0`, which C adds automatically to the end of strings, so that, like in our assembly routine `print_string`, during processing we can easily determine when we reach the end of the string.

5.2 Executing our Kernel Code

Enough of the theory, let's boot and execute the simplest of kernels written in C. This step will use all we have learnt so far, and will pave the way to faster progress in developing our operating system's features.

The involved steps are as follows:

- Write and compile the kernel code.
- Write and assemble the boot sector code
- Create a kernel image that includes not only our boot sector but our compiled kernel code
- Load our kernel code into memory
- Switch to 32-bit protected mode
- Begin executing our kernel code

5.2.1 Writing our Kernel

This will not take long, since, for the moment, the main function of our kernel merely is to let us know it has been successfully loaded and executed. We can elaborate on the kernel later, so it is important initially to keep things simple. Save the code in Figure XXX into a file called `kernel.c`.

```
void main() {
    // Create a pointer to a char, and point it to the first text cell of
    // video memory (i.e. the top-left of the screen)
    char* video_memory = (char*) 0xb8000;
    // At the address pointed to by video_memory, store the character 'X'
    // (i.e. display 'X' in the top-left of the screen).
    *video_memory = 'X';
}
```

Compile this to raw binary as follows:

```
$gcc -ffreestanding -c kernel.c -o kernel.o
$ld -o kernel.bin -Ttext 0x1000 kernel.o --oformat binary
```

Note that, now, we tell the linker that the origin of our code once we load it into memory will be `0x1000`, so it knows to offset local address references from this origin, just like we use `[org 0x7c00]` in our boot sector, because that is where BIOS loads and then begins to execute it.

5.2.2 Creating a Boot Sector to Bootstrap our Kernel

We are going to write a boot sector now, that must bootstrap (i.e. load and begin executing) our kernel from the disk. Since the kernel was compiled as 32-bit instructions, we are going to have to switch into 32-bit protected mode before executing the kernel code. We know that BIOS will load only our boot sector (i.e. the first 512 bytes of the disk), and so not our kernel, when the computer boots, but in Section XXX we have seen how we can use the BIOS disk routines to have our boot sector load additional sectors from a disk, and we are vaguely aware that, after we switch into protected mode, the

lack of BIOS will make it hard for us to use the disk: we would have to write a floppy or hard disk driver ourselves!

To simplify the problem of which disk and from which sectors to load the kernel code, the boot sector and kernel of an operating system can be grafted together into a *kernel image*, which can be written to the initial sectors of the boot disk, such that the boot sector code is always at the head of the kernel image. Once we have compiled the boot sector described in this section, we can create our kernel image with the following file concatenation command:

```
cat boot_sect.bin kernel.bin > os-image
```

Figure XXX shows a boot sector that will bootstrap our kernel from a disk containing our kernel image, [os-image](#).

```
; A boot sector that boots a C kernel in 32-bit protected mode
[org 0x7c00]
KERNEL_OFFSET equ 0x1000 ; This is the memory offset to which we will load our kernel

mov [BOOT_DRIVE], dl ; BIOS stores our boot drive in DL, so it's
                     ; best to remember this for later.

mov bp, 0x9000      ; Set-up the stack.
mov sp, bp

mov bx, MSG_REAL_MODE ; Announce that we are starting
call print_string      ; booting from 16-bit real mode

call load_kernel      ; Load our kernel

call switch_to_pm      ; Switch to protected mode, from which
                     ; we will not return

jmp $

; Include our useful, hard-earned routines
%include "print/print_string.asm"
%include "disk/disk_load.asm"
%include "pm/gdt.asm"
%include "pm/print_string_pm.asm"
%include "pm/switch_to_pm.asm"

[bits 16]

; load_kernel
load_kernel:
    mov bx, MSG_LOAD_KERNEL ; Print a message to say we are loading the kernel
    call print_string

    mov bx, KERNEL_OFFSET   ; Set-up parameters for our disk_load routine, so
    mov dh, 15               ; that we load the first 15 sectors (excluding
    mov dl, [BOOT_DRIVE]     ; the boot sector) from the boot disk (i.e. our
    call disk_load          ; kernel code) to address KERNEL_OFFSET

    ret

[bits 32]
; This is where we arrive after switching to and initialising protected mode.
```

```

BEGIN_PM:

    mov ebx, MSG_PROT_MODE ; Use our 32-bit print routine to
    call print_string_pm    ; announce we are in protected mode

    call KERNEL_OFFSET       ; Now jump to the address of our loaded
                            ; kernel code, assume the brace position,
                            ; and cross your fingers. Here we go!

    jmp $                  ; Hang.

; Global variables
BOOT_DRIVE      db 0
MSG_REAL_MODE   db "Started in 16-bit Real Mode", 0
MSG_PROT_MODE   db "Successfully landed in 32-bit Protected Mode", 0
MSG_LOAD_KERNEL db "Loading kernel into memory.", 0

; Bootsector padding
times 510-($-$$) db 0
dw 0xa55

```

Before running this command in Bochs, ensure that the Bochs configuration file has the boot disk set to your kernel image file, as in Figure XXX.

```

floppya: 1_44=os-image, status=inserted
boot: a

```

One question you might be wondering is why we loaded as many as 15 segments (i.e. $512 * 15$ bytes) from the boot disk, when our kernel image was much smaller than this; actually it was less than one sector in size, so to load 1 sector would have done the job. The reason is simply that it does not hurt to read those additional sectors from the disk, even if they have not been initialised with data, but it may hurt when trying to detect that we didn't read enough sectors at this stage when we later add to, and therefore increase the memory footprint size of, our kernel code: the computer would hang without warning, perhaps halfway though a routine that was split across an unloaded sector boundary --- an ugly bug.

Congratulations if an 'X' was displayed in the top-left corner of the screen, since though appearing pointless to the average computer user this signifies a huge step from where we started: we have now boot-strapped into a higher-level language, and can start to worry less about assembly coding and concern ourselves more with how we would like to develop our operating system, and learning a little more about C, of course; but this is the best way to learn C: looking up to it as a higher level language rather than looking down upon it from the perspective of an even higher abstraction, such as Java or a scripting language (e.g. Python, PHP, etc.).

5.2.3 Finding Our Way into the Kernel

It was definitely a good idea to start with a very simple kernel, but by doing so we overlooked a potential problem: when we boot the kernel, recklessly we jumped to, and therefore began execution from, the first instruction of the kernel code; but we saw in Section XXX how the C compiler can decide to place code and data wherever it chooses in the output file. Since our simple kernel had a single function, and based on our previous observations of how the compiler generates machine code, we might assume that the first machine code instruction is the first instruction of kernel's entry function, `main`, but suppose our kernel code look like that in Figure XXX.

```
void some_function() {
}

void main() {
    char* video_memory = 0xb8000;
    *video_memory = 'X';
    // Call some function
    some_function();
}
```

Now, the compiler will likely precede the instructions of the intended entry function `main` by those of `some_function`, and since our boot-strapping code will begin execution blindly from the first instruction, it will hit the first `ret` instruction of `some_function` and return to the boot sector code without ever having entered `main`. The problem is, that entering our kernel in the correct place is too dependant upon the ordering of elements (e.g. functions) in our kernel's source code and upon the whims of the compiler and linker, so we need to make this more robust.

A trick that many operating systems use to enter the kernel correctly is to write a very simple assembly routine that is always attached to the start of the kernel machine code, and whose sole purpose is to call the entry function of the kernel. The reason assembly is used is because we know exactly how it will be translated in machine code, and so we can make sure that the first instruction will eventually result in the kernel's entry function being reached.

This is a good example of how the linker works, since we haven't really exploited this important tool yet. The linker takes object files as inputs, then joins them together, but resolves any labels to their correct addresses. For example, if one object file has a piece of code that has a call to a function, `some_function`, defined in another object file, then after the object file's code has been physically linked together into one file, the label `:code:'some.function'` will be resolved to the offset of wherever that particular routine ended up in the combined code.

Figure XXX shows a simple assembly routine for entering the kernel.

```
; Ensures that we jump straight into the kernel's entry function.
[bits 32]           ; We're in protected mode by now, so use 32-bit instructions.
[extern main]       ; Declare that we will be referencing the external symbol 'main',
                   ; so the linker can substitute the final address
```

```
call main      ; invoke main() in our C kernel
jmp $         ; Hang forever when we return from the kernel
```

You can see from the line `call main` that the code simply calls a function that goes by the name of `main`. But `main` does not exist as a label within this code, since it is expected to exist within one of the other object files, such that it will be resolved to the correct address at link time; this expectation is expressed by the directive `[extern main]`, at the top of the file, and the linker will fail if it doesn't find such a label.

Previously we have compiled assembly into a raw binary format, because we wanted to run it as boot sector code on the CPU, but for this piece of code cannot stand alone, without having that label resolved, so we must compile it as follows as an object file, therefore preserving meta information about the labels it must resolve:

```
$nasm kernel_entry.asm -f elf -o kernel_entry.o
```

The option `-f elf` tells the assembler to output an object file of the particular format Executable and Linking Format (ELF), which is the default format output by out C compiler.

Now, rather than simply linking the `kernel.o` file with itself to create `kernel.bin`, we can link it with `kernel_entry.o`, as follows:

```
$ld -o kernel.bin -Ttext 0x1000 kernel_entry.o kernel.o --oformat binary
```

The linker respects the order of the files we gave to it on the command line, such that the previous command will ensure our `kernel_entry.o` will precede the code in `kernel.o`.

As before, we can reconstruct our kernel image file with the following command:

```
cat boot_sect.bin kernel.bin > os-image
```

Now we can test this in Bochs, but with more reassurance that our boot-block will find its way into the correct entry point of our kernel.

5.3 Automating Builds with Make

By now you should be fed up of having to re-type lots of commands, every time you change a piece of code, to get some feedback on a correction or a new idea you tried. Again, programmers have been here before, and have developed a multitude of tools for automating the build process of software. Here we will consider `make`, which is the predecessor of many of these other build tools, and which is used for building, amongst other operating systems and applications, Linux and Minix. The basic principle of `make` is that we specify in a configuration file (usually called `Makefile`) how to convert one file into another, such that the generation of one file may be described to depend on the existence of one or more other file. For example, we could write the following rule in a `Makefile`, that would tell `make` exactly how to compile a C file into an object file:

```
kernel.o : kernel.c
gcc -ffreestanding -c kernel.c -o kernel.o
```

The beauty of this is that, in the same directory as the `Makefile`, we can now type:

```
$make kernel.o
```

which will re-compile our C source file only if `kernel.o` does not exist or has an older file modification time than `kernel.c`. But it is only when we add several inter-dependent rules that we see how `make` can really help us to save time and unnecessary command executions.

```
# Build the kernel binary
kernel.bin: kernel_entry.o kernel.o
    ld -o kernel.bin -Ttext 0x1000 kernel_entry.o kernel.o --oformat binary

# Build the kernel object file
kernel.o : kernel.c
    gcc -ffreestanding -c kernel.c -o kernel.o

# Build the kernel entry object file.
kernel_entry.o : kernel_entry.asm
    nasm kernel_entry.asm -f elf -o kernel_entry.o
```

If we run `make kernel.bin` with the Makefile in Figure XXX, `make` will know that, before it can run the command to generate `kernel.bin`, it must build its two dependencies, `kernel.o` and `kernel_entry.o`, from their source files, `kernel.c` and `kernel_entry.asm`, yielding the following output of the commands it ran:

```
nasm kernel_entry.asm -f elf -o kernel_entry.o
gcc -ffreestanding -c kernel.c -o kernel.o
ld -o kernel.bin -Ttext 0x1000 kernel_entry.o kernel.o --oformat binary
```

Then, if we run `make` again, we will see that `make` reports that the build target `kernel.bin` is up to date. However, if we modify, say, `kernel.c`, save it, then run `make kernel.bin`, we will see that only the necessary commands are run by `make`, as follows:

```
gcc -ffreestanding -c kernel.c -o kernel.o
ld -o kernel.bin -Ttext 0x1000 kernel_entry.o kernel.o --oformat binary
```

To reduce repetition in, and therefore improve ease of maintenance of, our Makefile, we can use the special makefile variables `$<`, `$@`, and `$^` as in Figure XXX.

```
# $^ is substituted with all of the target's dependency files
kernel.bin: kernel_entry.o kernel.o
    ld -o kernel.bin -Ttext 0x1000 $^ --oformat binary

# $< is the first dependency and $@ is the target file
kernel.o : kernel.c
    gcc -ffreestanding -c $< -o $@

# Same as the above rule.
kernel_entry.o : kernel_entry.asm
    nasm $< -f elf -o $@
```

It is often useful to specify targets that are not actually real targets, in that they do not generate files. A common use of such phoney targets is to make a `clean` target, so that when we run `make clean`, all of the generated files are deleted from the directory, leaving only the source files, as in Figure XXX.

```
clean:
    rm *.bin *.o
```

Cleaning your directory in this way is useful if you'd like to distribute only the source files to a friend, put the directory under version control, or if you'd like to test that modifications of your makefile will correctly build all targets from scratch.

If `make` is run without a target, the first target in the main file is taken to be the default, so you often see a phoney target such as `all` at the top of Makefile as in Figure XXX.

```
# Default make target.
all: kernel.bin

# $^ is substituted with all of the target's dependency files
kernel.bin: kernel_entry.o kernel.o
    ld -o kernel.bin -Ttext 0x1000 $^ --oformat binary

# $< is the first dependency and $@ is the target file
kernel.o : kernel.c
    gcc -ffreestanding -c $< -o $@

# Same as the above rule.
kernel_entry.o : kernel_entry.asm
    nasm $< -f elf -o $@
```

Note that, by giving `kernel.bin` as a dependency to the `all` target, we ensure that `kernel.bin` and all of its dependencies are built for the default target.

We can now put all of the commands for building our kernel and the loadable kernel image into a useful makefile (see Figure XXX), that will allow us to test changes or corrections to our code in Bochs simply by typing `make run`.

```
all: os-image

# Run bochs to simulate booting of our code.
run: all
    bochs

# This is the actual disk image that the computer loads,
# which is the combination of our compiled bootsector and kernel
os-image: boot_sect.bin kernel.bin
    cat $^ > os-image
```

```

# This builds the binary of our kernel from two object files:
# - the kernel_entry, which jumps to main() in our kernel
# - the compiled C kernel
kernel.bin: kernel_entry.o kernel.o
    ld -o kernel.bin -Ttext 0x1000 $^ --oformat binary

# Build our kernel object file.
kernel.o : kernel.c
    gcc -ffreestanding -c $< -o $@

# Build our kernel entry object file.
kernel_entry.o : kernel_entry.asm
    nasm $< -f elf -o $@

# Assemble the boot sector to raw machine code
#   The -I options tells nasm where to find our useful assembly
#   routines that we include in boot_sect.asm
boot_sect.bin : boot_sect.asm
    nasm $< -f bin -I '../..../16bit/' -o $@

# Clear away all generated files.
clean:
    rm -fr *.bin *.dis *.o os-image *.map

# Disassemble our kernel - might be useful for debugging.
kernel.dis : kernel.bin
    ndisasm -b 32 $< > $@

```

5.3.1 Organising Our Operating System's Code Base

We have now arrived at a very simple C kernel, that prints out an 'X' in the corner of the screen. The very fact that the kernel was compiled into 32-bit instructions and has successfully been executed by the CPU means that we have come far; but it is now time to prepare ourselves for the work ahead. We need to establish a suitable structure for our code, accompanied by a makefile that will allow us easily to add new source files with new features to our operating system, and to check those additions incrementally with an emulator such as Bochs.

Similarly to kernels such as Linux and Minix, we can organise our code base into the following folders:

- **boot**: anything related to booting and the boot sector can go in here, such as `boot_sect.asm` and our boot sector assembly routines (e.g. `print_string.asm`, `gdt.asm`, `switch_to_pm.asm`, etc.).
- **kernel**: the kernel's main file, `kernel.c`, and other kernel related code that is not device driver specific will go in here.
- **drivers**: any hardware specific driver code will go in here.

Now, within our makefile, rather than having to specify every single object file that we would like to build (e.g. `kernel/kernel.o`, `drivers/screen.o`, `drivers/keyboard.o`, etc.), we can use a special *wildcard* declaration as follows:

```
# Automatically expand to a list of existing files that
# match the patterns
C_SOURCES = $(wildcard kernel/*.c drivers/*.c)
```

Then we can convert the source filenames into object filenames using another `make` declaration, as follows:

```
# Create a list of object files to build, simple by replacing
# the '.c' extension of filenames in C_SOURCES with '.o'
OBJ = ${C_SOURCES:.c=.o}
```

Now we can link the kernel object files together, to build the kernel binary, as follows:

```
# Link kernel object files into one binary, making sure the
# entry code is right at the start of the binary.
kernel.bin: kernel/kernel_entry.o ${OBJ}
    ld -o $@ -Ttext 0x1000 $^ --oformat binary
```

A feature of `make` that will go hand-in-hand with our dynamic inclusion of object files is *pattern rules*, which tell `make` how to build one file type from another based on simple pattern machine of the filename, as follows:

```
# Generic rule for building 'somefile.o' from 'somefile.c'
%.o : %.c
    gcc -ffreestanding -c $< -o $@
```

The alternative to this would be much repetition, as follows:

```
kernel/kernel.o : kernel/kernel.c
    gcc -ffreestanding -c $< -o $@

drivers/screen.o : drivers/screen.c
    gcc -ffreestanding -c $< -o $@

drivers/keyboard.o : drivers/keyboard.c
    gcc -ffreestanding -c $< -o $@

...
```

Great, now that we understand `make` sufficiently, we can progress to develop our kernel, without having to re-type lots of commands, over and over, to check if something is working correctly. Figure XXX shows a complete makefile that will be suitable for progressing with our kernel.

```
# Automatically generate lists of sources using wildcards.
C_SOURCES = $(wildcard kernel/*.c drivers/*.c)
HEADERS = $(wildcard kernel/*.h drivers/*.h)

# TODO: Make sources dep on all header files.

# Convert the *.c filenames to *.o to give a list of object files to build
OBJ = ${C_SOURCES:.c=.o}

# Default build target
all: os-image
```

```
# Run bochs to simulate booting of our code.
run: all
    bochs

# This is the actual disk image that the computer loads
# which is the combination of our compiled bootsector and kernel
os-image: boot/boot_sect.bin kernel.bin
    cat $^ > os-image

# This builds the binary of our kernel from two object files:
# - the kernel_entry, which jumps to main() in our kernel
# - the compiled C kernel
kernel.bin: kernel/kernel_entry.o ${OBJ}
    ld -o $@ -Ttext 0x1000 $^ --oformat binary

# Generic rule for compiling C code to an object file
# For simplicity, we C files depend on all header files.
%.o : %.c ${HEADERS}
    gcc -ffreestanding -c $< -o $@

# Assemble the kernel_entry.
%.o : %.asm
    nasm $< -f elf -o $@

%.bin : %.asm
    nasm $< -f bin -I '.../.../16bit/' -o $@

clean:
    rm -fr *.bin *.dis *.o os-image
    rm -fr kernel/*.o boot/*.bin drivers/*.o
```

5.4 C Primer

C has a few quirks that can unsettle a new programmer of the language.

5.4.1 The Pre-processor and Directives

Before a C file is compiled into an object file, a pre-processor scans it for pre-processor directives and variables, and then usually substitutes them with code, such as macros and values of constants, or with nothing at all. The pre-processor is not essential for compiling C code, but serves rather to offer some convenience that makes the code more manageable.

```
#define PI 3.141592
...
float radius = 3.0;
float circumference = 2 * radius * PI;
...
```

The pre-processor would output the following code, ready for compilation:

```
...
float radius = 3.0;
float circumference = 2 * radius * 3.141592;
...
```

The pre-processor is also useful for outputting conditional code, but not conditional in the sense that a decision is made at run-time, like with an `if` statement, rather in the sense of compile-time. For example, consider the following use of pre-processor directives for the inclusion or exclusion of debugging code:

```
...
#ifndef DEBUG
print("Some debug message\n");
#endif
...
```

Now, if the pre-processor variable `DEBUG` has been defined, such debugging code will be included; otherwise, not. A variable may be defined on the command line when compiling the C file as follows:

```
$gcc -DDEBUG -c some_file.c -o some_file.o
```

Such command line variable declarations are often used for compile-time configuration of applications, and especially operating systems, which may include or exclude whole sections of code, perhaps to reduce the memory footprint of the kernel on a small embedded device.

5.4.2 Function Declarations and Header Files

When the compiler encounters a call to a function, that may or may not be defined in the file being compiled, it may make incorrect assumptions and produce incorrect machine code instructions if it has not yet encountered a description of the function's return type and arguments. Recall from Section XXX that the compiler must prepare the stack for variables that it passes to a function, but if the stack is not what the function expects, then the stack may become corrupted. For this reason, it is important that at least a declaration of the function's interface, if not the entire function definition, is given before it is used. This declaration is known as a function's *prototype*.

```
int add(int a, int b) {
    return a + b;
}

void main() {
    // This is okay, because our compiler has seen the full
    // definition of add.
    int result = add(5, 3);

    // This is not okay, since compiler does not know the return
    // type or anything about the arguments.
    result = divide(34.3, 12.76);

    // This is not okay, because our compiler knows nothing about
}
```

```

// this function's interface.
int output = external_function(5, "Hello", 4.5);
}

float divide(float a, float b) {
    return a / b;
}

```

This can be fixed as follows:

```

// These function prototypes inform the compiler about
// the function interfaces.
float divide(float a, float b); // <-- note the semi-colon
int external_function(int a, char* message, float b);

int add(int a, int b) {
    return a + b;
}

void main() {
    // This is okay, because our compiler has seen the full
    // definition of add.
    int result = add(5, 3);

    // This is okay now: compiler knows the interface.
    result = divide(34.3, 12.76);

    // This is okay now: compiler knows the interface.
    int output = external_function(5, "Hello", 4.5);
}

float divide(float a, float b) {
    return a / b;
}

```

Now, since some functions will be called from code compiled into other object files, they will also need to declare identical prototypes of those functions, which would lead to a lot of duplicated prototype declarations, which is difficult to maintain. For this reason, many C programs use the `#include` pre-processor directive to insert common code that contains the required prototypes prior to compilation. This common code is known as a *header file*, which we can think of as the interface to the compiled object file, and which is used as follows.

Sometimes, one header file may include another, so it is important not to re-define the same code...

- Casting types

Chapter 6

Developing Essential Device Drivers and a Filesystem

INTRO PART.

6.1 Hardware Input/Output

By writing to the screen we have actually already encountered a friendlier form of hardware I/O, known as memory-mapped I/O, whereby data written directly to a certain address range in main memory is written to the device's internal memory buffer, but now it is time to understand more about this interaction between CPU and hardware.

Let's take the now-popular TFT monitor as an example. The screen's surface is divided up into a matrix of backlit cells. By containing a layer of liquid crystals sandwiched between polarised film, the amount of light passing through each cell can be varied by the application of an electric field, since liquid crystals have the property that, when subjected to an electrical field, their orientation may be altered in a consistent manner; as the orientation of the crystals changes, they alter the light wave's direction of vibration, such that some of the light will be blocked by the polarised film at the screen's surface. For a colour display, each cell is further divided into three areas that are overlaid with filters for red, blue, and green [?].

So it is the hardware's job to ensure that the appropriate cells, or sub-cell colour areas, get subjected to appropriate electrical field to reconstruct the desired image on the screen. This side of hardware is best left to specialist electronic engineers, but there will be a *controller* chip, ideally with well defined functionality that is described in the chip's datasheet, on the device or motherboard with which the CPU can interact to direct the hardware. In reality, for reasons of backward compatibility, TFT monitors usually emulate older CRT monitors, and so can be driven by the motherboard's standard VGA controller, which generates a complex analog signal that directs an electron beam to scan across the phosphor-coated screen, and since there isn't really a CRT beam to direct, the TFT monitor cleverly interprets this signal as a digital image.

Internally, controller chips usually have several registers that can be read, written or both by the CPU, and it is the state of these registers that tell the controller what to do (e.g. what pins to set high or low to drive the hardware, or what internal function to perform). As an example, from the datasheet of Intel's widely used [82077AA](#) single-chip floppy disk controller [?], we see there is a pin (pin 57, labelled [MEO](#)) that drives the motor of the first floppy disk device (since a single controller can drive several such devices): when the pin is on, the motor spins; when off, the motor does not spin. The state of this particular pin is directly linked to a particular bit of the controller's internal register named the Digital Output Register ([DOR](#)). The state of that register can then be set by setting a value, with the appropriate bit set (bit 4, in this case), across the chip's data pins, labelled [DB0--DB7](#), and using the chip's register selection pins, [A0--A2](#), to select the [DOR](#) register by its internal address [0x2](#).

6.1.1 I/O Buses

Although historically the CPU would talk directly to device controllers, with ever increasing CPU speeds, that would require the CPU artificially to slow down to the same speed as the slowest device, so it is more practical for the CPU to issue I/O instructions directly to the controller chip of a high-speed, top-level *bus*. The bus controller is then responsible for relaying, at a compatible rate, the instructions to a particular device's controller. Then to avoid the top-level bus having to slow down for slower devices, the controller of a another bus technology may be added as a device, such that we arrive at the hierarchy of buses found in modern computers [?].

6.1.2 I/O Programming

So the question is, how do we read and write the registers of our device controllers (i.e. tell our devices what to do) programatically? In Intel architecture systems the registers of device controllers are mapped into an I/O address space, that is seperate from the main memory address space, then varients of the I/O instructions [in](#) and [out](#) are used to read and write data to the I/O addresses that are mapped to specific controller registers. For example, the floppy disk controller described earlier usually has its [DOR](#) register mapped to I/O address [0x3F2](#), so we could switch on the motor of the first drive with the following instructions:

```
mov dx, 0x3f2      ; Must use DX to store port address
in al, dx          ; Read contents of port (i.e. DOR) to AL
or al, 00001000b  ; Switch on the motor bit
out dx, al         ; Update DOR of the device.
```

In older systems, such as the Industry Standard Architecture (ISA) bus, the port addresses would be statically assigned to the devices, but with modern plug-and-play buses, such as Peripheral Component Interconnect (PCI), BIOS can dynamically allocate I/O address to most devices before booting the operating system. Such dynamic allocation requires devices to communicate configuration information over the bus to describe the hardware such as: how many I/O ports are required to be reserved for the registers; how much memory-mapped space is required; and a unique ID of the hardware type, to allow appropriate drivers to be found later by the operating system [?].

A problem with port I/O is that we cannot express these low-level instructions in C language, so we have to learn a little bit about *inline* assembly: most compilers allow you to inject snippets of assembly code into the body of a function, with gcc implementing this as follows:

```
unsigned char port_byte_in(unsigned short port) {
    // A handy C wrapper function that reads a byte from the specified port
    // "=a" (result) means: put AL register in variable RESULT when finished
    // "d" (port) means: load EDX with port
    unsigned char result;
    __asm__("in %%dx, %%al" : "=a" (result) : "d" (port));
    return result;
}
```

Note that the actual assembly instruction, `in %%dx, %%al`, looks a little strange to us, since gcc adopts a different assembly syntax (known as GAS), where the target and destination operands are reversed with respect to the syntax of our more familiar nasm syntax; also, `%` is used to denote registers, and this requires an ugly `%%`, since `%` is an escape character of the C compiler, and so `%%` means: escape the escape character, so that it will appear literally in the string.

Since these low-level port I/O functions will be used by most hardware drivers in our kernel, let's collect them together into the file `kernel/low_level.c`, which we can define as in Figure XXX.

```
unsigned char port_byte_in(unsigned short port) {
    // A handy C wrapper function that reads a byte from the specified port
    // "=a" (result) means: put AL register in variable RESULT when finished
    // "d" (port) means: load EDX with port
    unsigned char result;
    __asm__("in %%dx, %%al" : "=a" (result) : "d" (port));
    return result;
}

void port_byte_out(unsigned short port, unsigned char data) {
    // "a" (data) means: load EAX with data
    // "d" (port) means: load EDX with port
    __asm__("out %%al, %%dx" : : "a" (data), "d" (port));
}

unsigned short port_word_in(unsigned short port) {
    unsigned short result;
    __asm__("in %%dx, %%ax" : "=a" (result) : "d" (port));
    return result;
}

void port_word_out(unsigned short port, unsigned short data) {
    __asm__("out %%ax, %%dx" : : "a" (data), "d" (port));
}
```

6.1.3 Direct Memory Access

Since port I/O involves reading or writing individual bytes or words, the transfer of large amounts of data between a disk device and memory could potentially take up a great deal of better-spent CPU time. This issue has necessitated a means for the CPU to pass over this tedious task to someone else, a direct memory access (DMA) controller [?].

A good analogy of DMA is that of an architect wanting to move a wall from one place to another. The architect knows exactly what is to be done but has other important things to consider other than shifting each brick, and so instructs a builder to move the bricks, one by one, and to alert (i.e. interrupt) him when either the wall is finished or if there was some error that is stopping the wall from being finished.

6.2 Screen Driver

So far, our kernel is capable of printing an 'X' in the corner of the screen, which, whilst is sufficient to let us know our kernel has been successfully loaded and executed, doesn't tell us much about what is happening on the computer.

We know that we can have characters displayed on the screen by writing them somewhere within the display buffer at address `0xb8000`, but we don't want to keep having to worry about that sort of low-level manipulation throughout our kernel. It would be much nicer if we could create an abstraction of the screen that would allow us to write `print("Hello")`, and perhaps `clear_screen()`; and if it could scroll when we printed beyond the last display line, that would be icing on the cake. Not only would this sort of abstraction make it easier to display information within other code of our kernel, but it would allow us to easily substitute one display driver for another at a later date, perhaps if a certain computer could not support the colour VGA text mode that we currently assume.

6.2.1 Understanding the Display Device

Compared with some of the other hardware that we will soon look at, the display device is fairly straightforward, since, as a memory-mapped device, we can get by without understanding anything about control messages and hardware I/O. However, a useful device of the screen that requires I/O control (i.e. via I/O ports) to manipulate is the *cursor*, that flashes to mark the next position that a character will be written to on the screen. This is useful for a user, since it can draw their attention to a prompt to enter some text, but we will also use it as an internal marker, whether the cursor is visible or not, so that a programmer does not always have to specify the coordinates of where on the screen a string is to be displayed, for example: if we write `print("hello")`, each character will be written to.

6.2.2 Basic Screen Driver Implementation

Although we could write all of this code in `kernel.c`, that contains the kernel's entry function, `main()`, it is good to organise such functionality-specific code into its own file, which can be compiled and linked to our kernel code, ultimately with the same effect as putting it all into one file. Let's create a new driver implementation file, `screen.c`, and

a driver interface file, `screen.h`, in our `drivers` folder. Due to our use of *wildcard* file inclusion in our makefile, `screen.c` will (as will any other C files placed in that folder) be automatically compiled and linked to our kernel.

Firstly, let's define the following constants in `screen.h`, to make our code more readable:

```
#define VIDEO_ADDRESS 0xb8000
#define MAX_ROWS 25
#define MAX_COLS 80
// Attribute byte for our default colour scheme.
#define WHITE_ON_BLACK 0x0f

// Screen device I/O ports
#define REG_SCREEN_CTRL 0x3D4
#define REG_SCREEN_DATA 0x3D5
```

Then, let's consider how we would write a function, `print_char(...)`, that displays a single character at a specific column and row of the screen. We will use this function internally (i.e. privately), within our driver, such that our driver's public interface functions (i.e. the functions that we would like external code to use) will build upon it. We now know that video memory is simply a specific range of memory addresses, where each character cell is represented by two bytes, the first byte is the ASCII code of the character, and the second byte is an attribute byte, that allows us to set a colourscheme of the character cell. Figure XXX shows how we could define such a function, by making use of some other functions that we will define: `get_cursor()`, `set_cursor()`, `get_screen_offset()`, and `handle_scrolling()`.

```
/* Print a char on the screen at col, row, or at cursor position */
void print_char(char character, int col, int row, char attribute_byte) {
    /* Create a byte (char) pointer to the start of video memory */
    unsigned char *vidmem = (unsigned char *) VIDEO_ADDRESS;

    /* If attribute byte is zero, assume the default style. */
    if (!attribute_byte) {
        attribute_byte = WHITE_ON_BLACK;
    }

    /* Get the video memory offset for the screen location */
    int offset;
    /* If col and row are non-negative, use them for offset. */
    if (col >= 0 && row >= 0) {
        offset = get_screen_offset(col, row);
    } else {
        offset = get_cursor();
    }

    /* If we see a newline character, set offset to the end of
     * current row, so it will be advanced to the first col
     * of the next row.
    if (character == '\n') {
```

```

int rows = offset / (2*MAX_COLS);
offset = get_screen_offset(79, rows);
// Otherwise, write the character and its attribute byte to
// video memory at our calculated offset.
} else {
    vidmem[offset] = character;
    vidmem[offset+1] = attribute_byte;
}

// Update the offset to the next character cell, which is
// two bytes ahead of the current cell.
offset += 2;
// Make scrolling adjustment, for when we reach the bottom
// of the screen.
offset = handle_scrolling(offset);
// Update the cursor position on the screen device.
set_cursor(offset);
}

```

Let's tackle the easiest of these functions first: `get_screen_offset`. This function will map row and column coordinates to the memory offset of a particular display character cell from the start of video memory. The mapping is straightforward, but we must remember that each cell holds two bytes. For example, if I want to set a character at row 3, column 4 of the display, then the character cell of that will be at a (decimal) offset of 488 ($(3 * 80 \text{ (i.e. the row width)} + 4) * 2 = 488$) from the start of video memory. So our `get_screen_offset` function will look something like that in Figure XXX.

```

// This is similar to get_cursor, only now we write
// bytes to those internal device registers.
port_byte_out(REG_SCREEN_CTRL, 14);
port_byte_out(REG_SCREEN_DATA, (unsigned char)(offset >> 8));
port_byte_out(REG_SCREEN_CTRL, 15);

```

Now let's look at the cursor control functions, `get_cursor()` and `set_cursor()`, which will manipulate the display controller's registers via a set of I/O ports. Using the specific video devices I/O ports to read and write its internal cursor-related registers, the implementation of these functions will look something like that in Figure XXX.

```

cursor_offset -= 2*MAX_COLS;

// Return the updated cursor position.
return cursor_offset;
}

int get_cursor() {
    // The device uses its control register as an index
    // to select its internal registers, of which we are
    // interested in:
}

```

```

// reg 14: which is the high byte of the cursor's offset
// reg 15: which is the low byte of the cursor's offset
// Once the internal register has been selected, we may read or
// write a byte on the data register.
port_byte_out(REG_SCREEN_CTRL, 14);
int offset = port_byte_in(REG_SCREEN_DATA) << 8;
port_byte_out(REG_SCREEN_CTRL, 15);
offset += port_byte_in(REG_SCREEN_DATA);
// Since the cursor offset reported by the VGA hardware is the
// number of characters, we multiply by two to convert it to
// a character cell offset.
return offset*2;
}

void set_cursor(int offset) {
    offset /= 2; // Convert from cell offset to char offset.
    // This is similar to get_cursor, only now we write
    // bytes to those internal device registers.
}

```

So now we have a function that will allow us to print a character at a specific location of the screen, and that function encapsulates all of the messy hardware specific stuff. Usually, we will not want to print each character to the screen, but rather a whole string of characters, so let's create a friendlier function, `print_at(...)`, that takes a pointer to the first character of a string (i.e. a `char *`) and prints each subsequent character, one after the other, from the given coordinates. If the coordinates `(-1, -1)` are passed to the function, then it will start printing from the current cursor location. Our `print_at(...)` function will look something like that in Figure XXX.

```

void print_at(char* message, int col, int row) {
    // Update the cursor if col and row not negative.
    if (col >= 0 && row >= 0) {
        set_cursor(get_screen_offset(col, row));
    }
    // Loop through each char of the message and print it.
    int i = 0;
    while(message[i] != 0) {
        print_char(message[i++], col, row, WHITE_ON_BLACK);
    }
}

```

And purely for convenience, to save us from having to type `print_at('hello', -1, -1)`, we can define a function, `print`, that takes only one argument as in Figure XXX.

```

void print(char* message) {
    print_at(message, -1, -1);
}

```

Another useful, but not too difficult function, is `clear_screen(...)`, which will allow us to tidy up our screen by writing blank characters at every position. Figure XXX shows how we might implement such a function.

```
void clear_screen() {
    int row = 0;
    int col = 0;

    /* Loop through video memory and write blank characters. */
    for (row=0; row<MAX_ROWS; row++) {
        for (col=0; col<MAX_COLS; col++) {
            print_char(' ', col, row, WHITE_ON_BLACK);
        }
    }

    // Move the cursor back to the top left.
    set_cursor(get_screen_offset(0, 0));
}
```

6.2.3 Scrolling the Screen

If you expected the screen to scroll automatically when your cursor reached the bottom of the screen, then your brain must have lapsed back into higher-level computer land. This can be forgiven, because screen scrolling seems like such a natural thing that we simply take for granted; but working at this level, we have complete control over the hardware, and so must implement this feature ourselves.

In order to make the screen appear to scroll when we reach the bottom, we must move each character cell upwards by one row, and then clear the last row, ready for writing the new row (i.e. the row that would otherwise have been written beyond the end of the screen). This means the the top row will be overwritten by the second row, and so the top row will be lost forever, which we will not concern ourselves with, since our aim is to allow the user to see the most recent log of activity on their computer.

A nice way to implement scrolling is to call a function, which we will define as `handle_scrolling`, immediately after incrementing the cursors position in our `print_char`. The the roll of `handle_scrolling`, then, is to ensure that, whenever the cursor's video memory offset is incremented beyond the last row of the screen, the rows are scrolled and the cursor is repositioned within the last visible row (i.e. the new row).

Shifting a row equates to copying all of its bytes --- two bytes for each of the 80 character cells in a row --- to the address of the previous row. This is a perfect opportunity for adding a general purpose `memory_copy` function to our operating system. Since we are likely to use such a function in other areas of our OS, let's add it to the file `kernel/util.c`. Our `memory_copy` function will take addresses of the source and destination and the number of bytes to copy, then, with a loop, will copy each byte as in Figure XXX.

```
/* Copy bytes from one place to another. */
void memory_copy(char* source, char* dest, int no_bytes) {
```

```
    int i;
    for (i=0; i<no_bytes; i++) {
        *(dest + i) = *(source + i);
    }
}
```

Now we can use `memory_copy`, as in Figure XXX, to scroll our screen.

```
/* Advance the text cursor, scrolling the video buffer if necessary. */
int handle_scrolling(int cursor_offset) {

    // If the cursor is within the screen, return it unmodified.
    if (cursor_offset < MAX_ROWS*MAX_COLS*2) {
        return cursor_offset;
    }

    /* Shuffle the rows back one. */
    int i;
    for (i=1; i<MAX_ROWS; i++) {
        memory_copy(get_screen_offset(0,i) + VIDEO_ADDRESS,
                    get_screen_offset(0,i-1) + VIDEO_ADDRESS,
                    MAX_COLS*2
        );
    }

    /* Blank the last line by setting all bytes to 0 */
    char* last_line = get_screen_offset(0,MAX_ROWS-1) + VIDEO_ADDRESS;
    for (i=0; i < MAX_COLS*2; i++) {
        last_line[i] = 0;
    }

    // Move the offset back one row, such that it is now on the last
    // row, rather than off the edge of the screen.
    cursor_offset -= 2*MAX_COLS;

    // Return the updated cursor position.
    return cursor_offset;
}
```

6.3 Handling Interrupts

6.4 Keyboard Driver

6.5 Hard-disk Driver

6.6 File System

Chapter 7

Implementing Processes

7.1 Single Processing

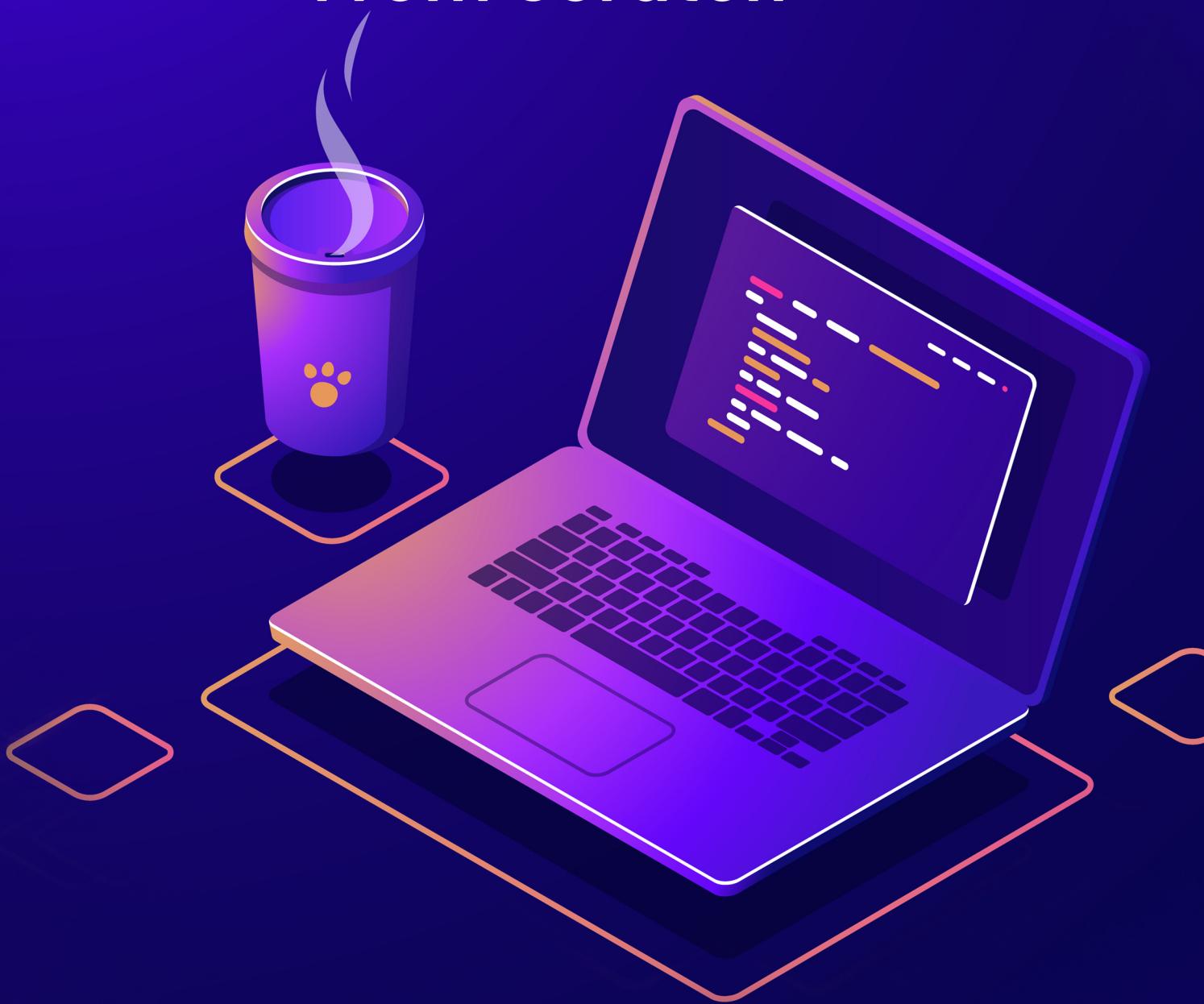
7.2 Multi-processing

Chapter 8

Summary

Bibliography

Developing A Computer Operating System From Scratch



An Attempt To Introduce OS Development At Beginner Level

TINU TOM

Developing A Computer Operating System From Scratch

An Attempt To Introduce OS Development At Beginner
Level

TINU TOM

This book is for sale at <http://leanpub.com/OS-DEV>

This version was published on 2021-07-28



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 TINU TOM

Contents

Preface	1
Getting Started	2
Installing QEMU	2
Installing A Hex Editor	3
Installing Notepad++	4
Installing NASM	5
Installing SASM	5
Installing MinGw For Compiling C Programs	7
Adding The Downloaded Softwares To Environment Path	9
Programming In C	14
Introduction	14
Hello , World	14
Data Types	16
Basic Data Types	16
int Data Type	16
char Data type	17
void Data Type	18
Derived Data Types	19
Pointers	19
Arrays	23
Functions	26
Structures	31
Branching	33
Looping	39
Type Casting	41
Arithmetic Operators	45
Increment , Decrement Operators	48
Bitwise Operators	49
Bitwise AND and Bitwise OR Operators	49
Left shift and Right shift Operators	50
Macros	51
Hexadecimal Notations	53

CONTENTS

Comments	55
Let's Have A Game	56
Programming in Assembly Language	59
Introduction	59
What is an Assembly Language	59
What is a Compiler actually?	60
x86 Processor data sizes	62
Assembly Hello , World	62
Registers	63
General Registers	64
Data Registers	64
Pointer Registers	66
Index Registers	67
Control Registers	67
Segment Registers	68
x86 Processor Endianess	68
Commands For Register Operations	69
mov Command	69
add Command	72
sub Command	73
push And pop Commands	75
Working With Stack	75
Practical Implementation Of Stack	77
pushAll And popAll Commands	81
inc And dec Commands	84
Extra Commands	85
jmp Command	85
call Command	87
cmp Command	91
Variables	94
Memory Addressing	95
Comments	97
Conclusion	98
Beginning Operating System Development	99
Introduction	99
Writing Programs For Boot Sector	100
Printing To Screen (Hello , World OS)	106
Filling The Screen With Characters(For Fun)!!	108
Filling The Screen With Colours	109
Other Bios Display Related Routines	111
Running Programs Written In C	112

CONTENTS

Switching To Protected Mode	112
Defining The GDT	112
Life Without Bios	112
Implementing The GDT	112
Making The Switch	113
Making Way For Running C Code	118
Making A Boot Loader	119
Calling Our C Kernel	121
Video Graphics	125
Introduction	125
Poking Video Memory	125
Displaying Text and Colours To Screen	125
Examples	128
Alpha	128
Beta	129
Gamma	130
Delta	131
Implementing Graphics Driver	132
Developing a Simple Video Player	136
Theory	136
Practical Implementation	137
Implementing Keyboard Driver	139
Introduction	139
Scan Codes	139
Implementing Keyboard Driver	140
The PIC Chip	140
Practical Implementation	140
External References	151
Making Our First Prototype : OS0	152
Introduction	152
Developing the First Prototype	152
Explanation	160
Accessing Hard Disks	161
Introduction	161
Working With Hard Disks	161
Types of Hard Disk	161
HDD	161
SSD	161
How Hard Disk is Divided	162
Implementing a Hard Disk Driver	162

CONTENTS

How it Works?	168
External References	169
Creating a Simple File System	170
Introduction	170
The Implementation	170
Formatting	170
File Allocation Table And Storage Space	171
Create	171
Save	171
Retrieve	172
Conclusion	172
External References	172
Graphics Mode GUI Creation	173
Introduction	173
Drawing In Graphics Mode	173
Modes	173
Choosing A Mode	173
Making Switch To The Selected Mode	174
Video Memory and Drawing	178
Sample User Interface Using Graphics Mode	181
External References	184
Implementing a Mouse Driver	185
Introduction	185
How The Mouse Work	185
IRQ12	185
The Mouse Events And Packets	185
Double Clicks	185
Practical Implementation	186
External References	188
Audio	189
Introduction	189
Generating Sound : First Try	189
Generating Sound : Second Try Integrating With OS0	191
External References	194
Going Advanced	196
CD-ROM	196
ATAPI	196
External Reference	196
USB	197

CONTENTS

Universal Serial Bus	197
External Reference	197
Networking	198
Networking	198
External Reference	198
Paging	199
Paging	199
External Reference	199
GDT	200
Global Descriptor Table	200
External Reference	200
IDT	201
Interrupt Descriptor Table	201
External Reference	201
Timers	202
Programmable Interval Timer	202
External Reference	202
GRUB	203
UEFI	204
How To Move Further?	205
The Thank You Summary	206

Preface

We all have used an operating system, whether you are using a mobile phone or a computer or any electronic device. All of these devices have software that works close to the hardware.

Most of the operating systems are split into two main components which are the kernel and the shell. Kernel of an operating system is what communicates with the hardware, manages memory etc and does all other main stuff etc... The main aim of the kernel is to provide an abstraction level to the shell of that operating system which is the software that the user directly communicates with. Shell of an operating system is what you basically see on the screen and what you directly interact with. When you command the shell to do a specific thing such as creating a file, the shell requests that service to the kernel and the kernel does all of the things needed to create a file. So basically the shell is only a middle stander between you and your device.

Let's learn these basic theories later. Let me say the reason which led me to write this book. OS development is a topic which is seen at an angle so that most people think it can be done only by a small section of geeks. The reason they say is that "Low level stuff is hard to learn and only the brightest ones could do that". And that argument is totally wrong as there are many people including children and teenagers who do the electronic stuff (Including programming). These people will only get the smallest level of abstraction when developing their projects and all other low level stuff should be done their own.

But when coming to the side of creating an operating system (Specifically on x86 architecture), People say that it's too hard, but it's not.

Your question now may be "Then why does only a small section of people do that", and the answer of that question is that "There is only a small level of documentation for beginners to get started". According to the current situation, People who are not too old in software development area could not do this things as all of the documentation are hard to understand for beginners.

Also currently there is no book giving a complete guide to this area apart from some websites which is meant for the "Too old people".

I will promise you that, even if you are starting in this field or know only some part of this area, you could read this book. We will start learning from programming in C (Which is a basic need for OS development), and assembly, and then forward, we will dive into real OS development on the Intel's x86 architecture which is where operating systems such as Windows and Linux mainly run on.

So for now, you need to take away all things in your mind about OS development and focus mainly on learning. I will explain each and every section deeply and also explain how the code given in this book works fully.

Getting Started

Learning os development is fun, But before starting, we need some tools which are essential for development and testing. I will explain you fully how to set everything to get started working on the project. We will use Windows as operating system to develop our project, But the methods discussed in this book can also be implemented in Linux. There is nothing stopping you from doing that.

Installing QEMU

To run the operating system we develop, we have a total of three options.

1. Use an emulator
2. Use a virtual machine
3. And finally, Running it on the the real hardware(By booting from a cd or usb)

Using an emulator or virtual machine works somewhat alike(From an outside perspective). But they are not the same(From an internal perspective) which is why they are called different as emulators and virtual machines.

Virtual machine softwares are used to run software which the processor could directly execute to a great extend. But the purpose of emulators is so that it could enable us running software which is intended to run on different architectures, but also it could emulate the architecture which it is currently running on.

The final option to run the os we develop is to run in the real hardware, by burning the raw binary to a cd or usb and booting it on the real hardware. If you want to run the os we develop on the real hardware, please ensure to disable Secure Boot option provided in latest computers from bios. We don't really encourage you running the os in real hardware unless you clearly know what you are doing. This is because of many reasons one of which is that you could corrupt your hard disk if you program the os incorrectly, So what you need to do is, First try running it on an emulator or virtual machine and study it's behaviour and when you confirm it works safe, You could try running on real hardware.

Please note that i will not be responsible for any damages you do to your system by using unsafe approaches in any manner. If you want to run the os on real hardware, you could try running it on a secondary system which you won't mind if it gets harmed.

For now, We will try running the os that we are going to develop on an Emulator named QEMU. You could find in on <https://www.qemu.org/>



The installation of qemu is straight forward and you could carry it your own.

Installing A Hex Editor

You could install any Hex Editor in your machine if You know how to use it.

But i will follow with a hex editor named Hex Editor Neo found at <https://www.hhdsoftware.com/free-hex-editor>. If you don't know what a hex editor is, we will discuss it in later chapters but for now, you could download it.

Free Hex Editor Neo

Free Hex Editor Neo is the fastest large files optimized binary file editor for Windows platform developed by HHD Software Ltd. It's distributed under "Freemium" model and provides you with all basic editing features for free.



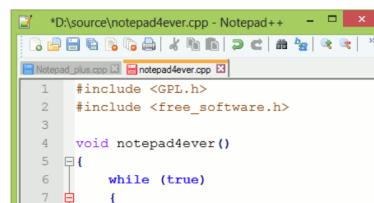
Installing Notepad++

We will use Notepad++ as our ide to develop the project. You could find in on <https://notepad-plus-plus.org/>

What is Notepad++

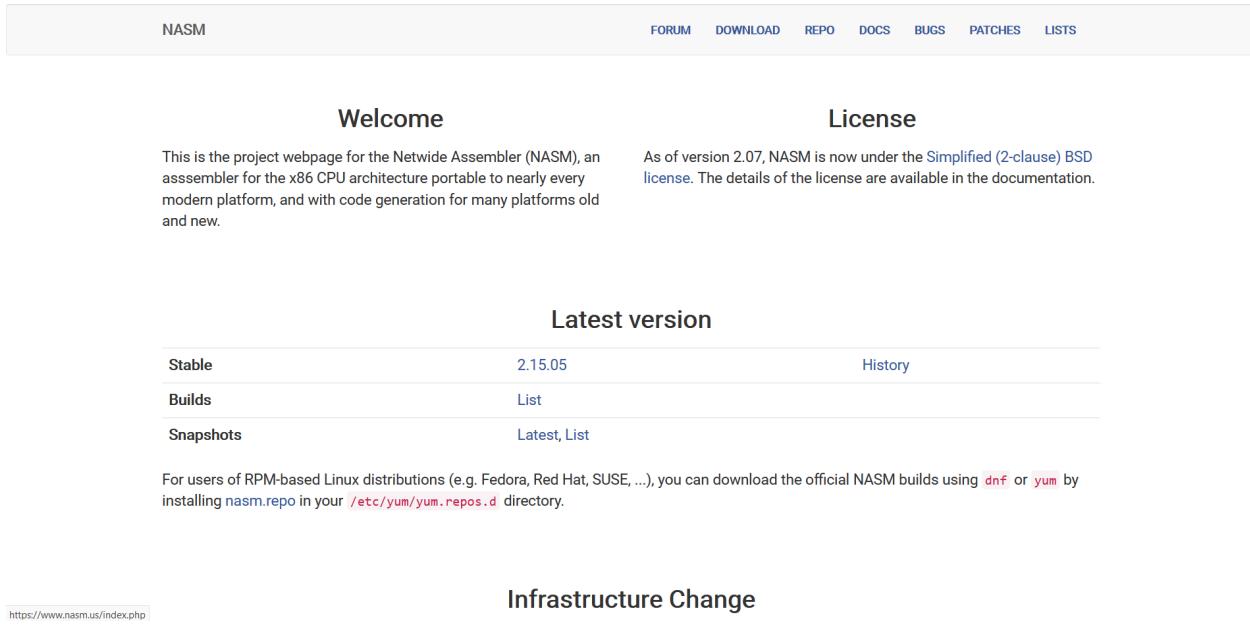
Notepad++ is a free (as in "free speech" and also as in "free beer") source code editor and Notepad replacement that supports several languages. Running in the MS Windows environment, its use is governed by [GNU General Public License](#).

Based on the powerful editing component Scintilla, Notepad++ is written in C++ and uses pure Win32 API and STL which ensures a higher execution speed and smaller program size. By optimizing as many routines as possible without losing user friendliness, Notepad++ is trying to reduce the world carbon dioxide emissions. When using less CPU power, the PC can throttle down and reduce power consumption, resulting in a greener environment.



Installing NASM

Nasm is a widely used assembler. We will learn what an assembler is, Why and how to use it and every point you need to know to get started later. You could find it on <https://www.nasm.us/>



The screenshot shows the official NASM project webpage. At the top, there is a navigation bar with links for FORUM, DOWNLOAD, REPO, DOCS, BUGS, PATCHES, and LISTS. The main content area is divided into three sections: 'Welcome', 'License', and 'Latest version'.

Welcome
This is the project webpage for the Netwide Assembler (NASM), an assembler for the x86 CPU architecture portable to nearly every modern platform, and with code generation for many platforms old and new.

License
As of version 2.07, NASM is now under the [Simplified \(2-clause\) BSD license](#). The details of the license are available in the documentation.

Latest version
A table showing the latest version information:

	2.15.05	History
Stable		
Builds	List	
Snapshots	Latest , List	

For users of RPM-based Linux distributions (e.g. Fedora, Red Hat, SUSE, ...), you can download the official NASM builds using `dnf` or `yum` by installing `nasm.repo` in your `/etc/yum/yum.repos.d` directory.

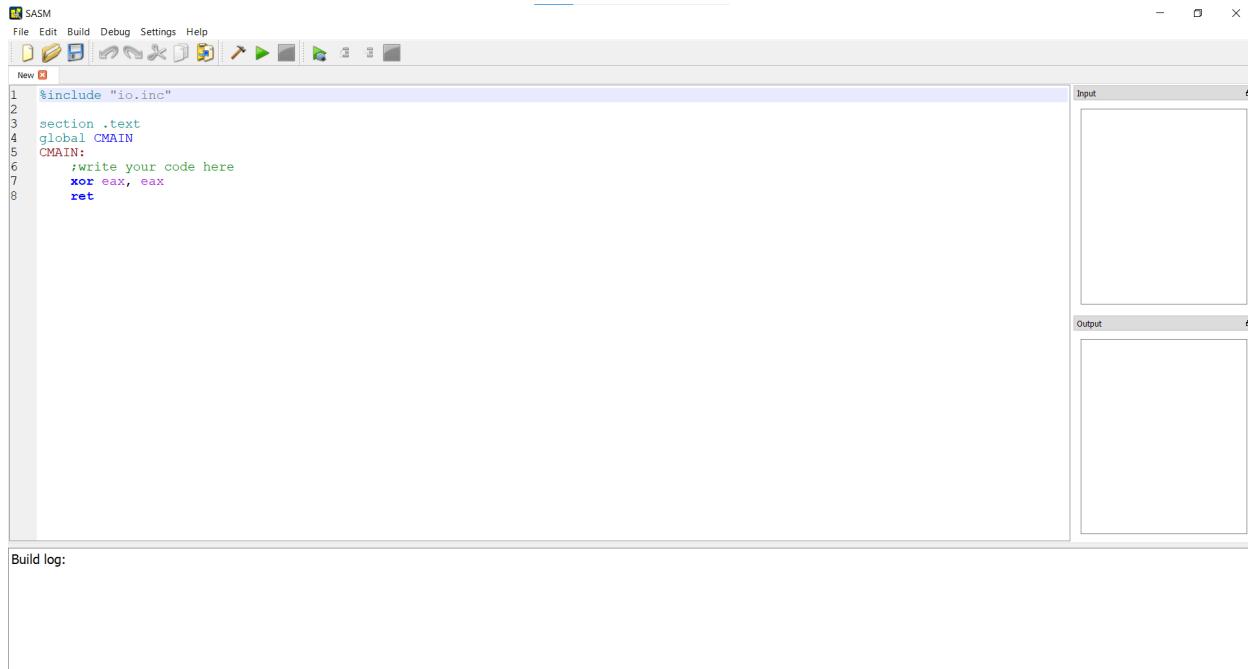
Infrastructure Change
<https://www.nasm.us/index.php>

Installing SASM

SASM is an ide which helps build assembly programs easily. There is no special need to download sasm for programming in assembly as we have already discussed installing nasm as an assembler, But for now for the sake of simplicity in learning developing applications in assembly, we could use this. If you currently know developing assembly programs, You could skip this step and also skip the chapter teaching programming in assembly.

By using sasm, You can avoid learning the concept of linking for some time and it also provides functions to print strings and numbers easily, Else you need to write your own routine or download special ones to print to screen. And the main part is that you could run and test your programs in one windows. You could download it from <https://sasm.software.informer.com/download/>

The image shows a composite screenshot. The top half is a screenshot of the software.informer website. It features a dark blue header with the site's logo and a search bar. Below the header, a breadcrumb navigation shows 'Windows > Developer Tools > IDE > SASM > Download'. A 'Share' button is in the top right. The main content area has a title 'SASM download' with a logo, followed by the subtext 'Develop projects with multiple assembly languages'. An 'Advertisement' for 'HERIOT WATT UNIVERSITY' is displayed with a 'FIND OUT MORE' button. Below this, a navigation bar includes 'Download' (underlined), 'Review', 'Comments (1)', and 'Questions & Answers'. A note says 'We do not have a download file for the latest version (3.11.1), but you can try downloading it from the developer's site'. A section for 'Download version 3.2 from Software Informer' follows, with a note about virus scanning and a 'DOWNLOAD NOW' button. To the right, there's another 'Advertisement' for 'upstox' showing a mobile app interface. The bottom half of the image is the SASM Integrated Development Environment (IDE). It has a menu bar with 'File', 'Edit', 'Build', 'Debug', 'Settings', and 'Help'. Below the menu is a toolbar with various icons. The main workspace displays the text 'Welcome to the SASM!' next to the SASM logo. A sidebar on the left lists options: 'Create new project', 'Open project', and 'Restore last session'.



```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6     ;write your code here
7     xor eax, eax
8     ret
```

Installing MinGw For Compiling C Programs

MinGw is a collection of tools which could be used to develop application in windows platform. We will use the c compiler provided by MinGw to develop our os. C is a mid level language best suited to speed up os development. C was initially developed to develop the unix operating system, We could use great features provided by c to get our os running. If you are new to Programming in c, we will learn necessary thing in upcoming chapters. I will only use the methods that you could easily follow to develop the os in c. If you currently know how to program in c and/or assembly, You could catch os development faster, but others don't need to worry, i will carry you along with the journey.

To install MinGw, you need to go to <https://sourceforge.net/projects/mingw/>

MinGW - Minimalist GNU for Windows

A native Windows port of the GNU Compiler Collection (GCC)

Brought to you by: cstraus, earnie, gressett, keithmarshall

★★★★★ 180 Reviews

Downloads: 2,396,490 This Week

Last Update: 2018-07-13

Download Get Updates Share This

Windows

Summary Files Reviews Support News Wiki Mailing Lists Tickets Git

This project is in the process of moving to osdn.net/projects/mingw, you can continue to follow us there.

MinGW: A native Windows port of the GNU Compiler Collection (GCC), with freely distributable import libraries and header files for building native Windows applications; includes extensions to the MSVC runtime to support C99 functionality. All of MinGW's software will execute on the 64bit Windows platforms.

Project Samples

Get started with a \$200 credit. Create a free account →

Let's put smart to work. IBM.

Until happy customers grow on trees, there's commun

Get latest updates about Open Source Projects, Conferences and News.

Sign Up No, Thank you

Check the Following option after starting the installer:

Package	Class	Installed Version	Repository Version	Description
mingw32-base	bin	2013072300	2013072200	An MSYS Installation for MinGW Developers (meta)
mingw32-gcc-ada	bin	6.3.0-1	6.3.0-1	The GNU Ada Compiler
mingw32-gcc-fortran	bin	6.3.0-1	6.3.0-1	The GNU FORTRAN Compiler
mingw32-gcc-g++	bin	6.3.0-1	6.3.0-1	The GNU C++ Compiler
mingw32-gcc-objc	bin	6.3.0-1	6.3.0-1	The GNU Objective-C Compiler
msys-base	bin	2013072300	2013072300	A Basic MSYS Installation (meta)

General Description Dependencies Installed Files Versions

The GNU Objective-C Compiler

This package provides the MinGW implementation of the GNU Objective-C language compiler.

This is an optional component of the MinGW Compiler Suite; you require it only if you wish to compile programs written in the Objective-C language.

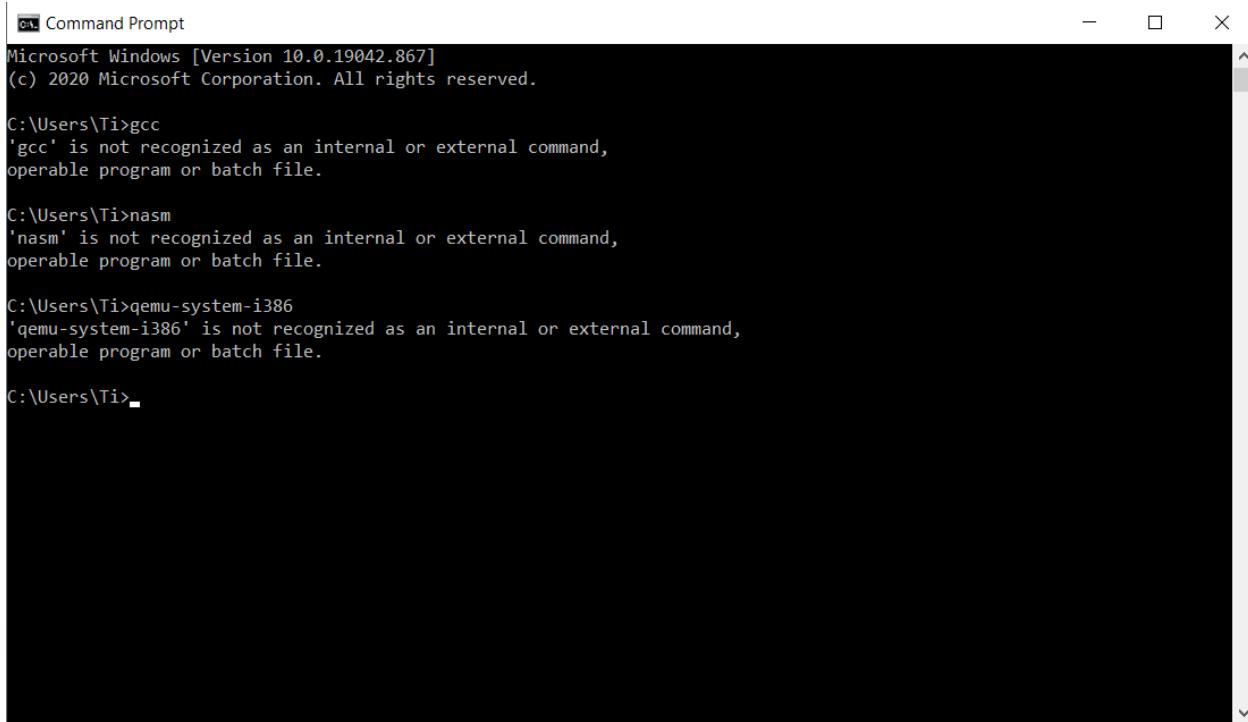
NOTE: For the gcc-4.8.1 release in order to install objc you need to install the gcc-objc-%-mingw32-dev package instead of the -bin package. This change was made to be consistent with what -bin usually contains compared to -dev. In reality, this release simply created the -dev package as a hybrid of content which would normally be distributed as separate -bin and -dev packages, whereas earlier releases provided the -bin package as such a hybrid. This change added an unnecessary level of package complexity; it has been reverted for the gcc-4.9.3 release.

And finally, Install it.

Adding The Downloaded Softwares To Environment Path

Finally, We have downloaded all tools that we need to get started in this learning process. But we need to access three of these softwares from the command line(cmd.exe).

For that, go to command prompt by typing cmd in the search bar and type gcc and hit enter. You will get an error message. Also type nasm and qemu-system-i386 in it. It Also will show an error message.



```
Command Prompt
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

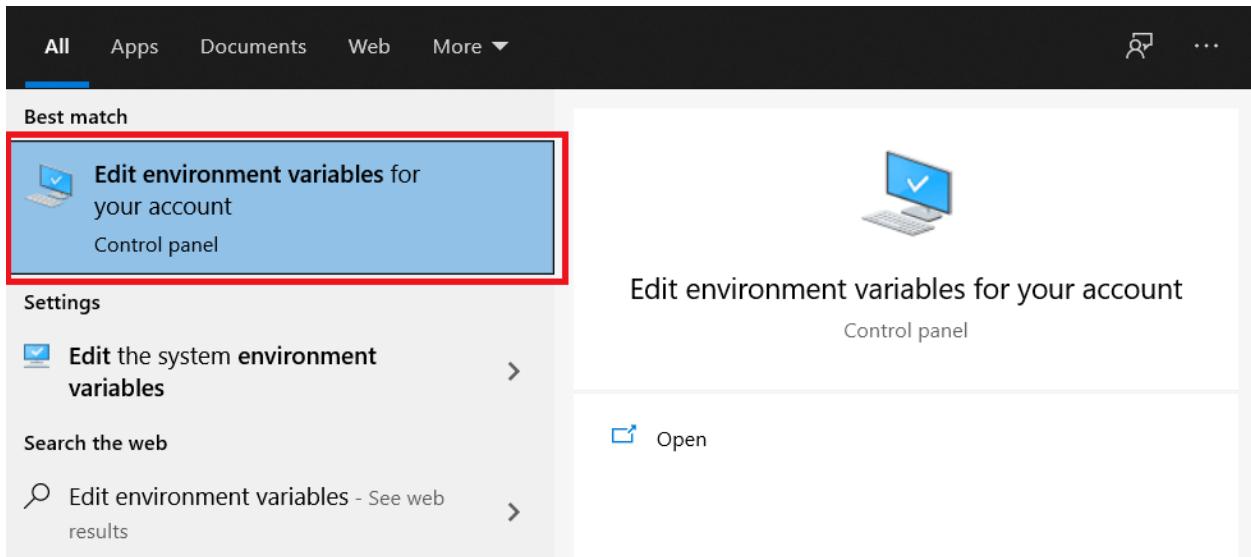
C:\Users\Ti>gcc
'gcc' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Ti>nasm
'nasm' is not recognized as an internal or external command,
operable program or batch file.

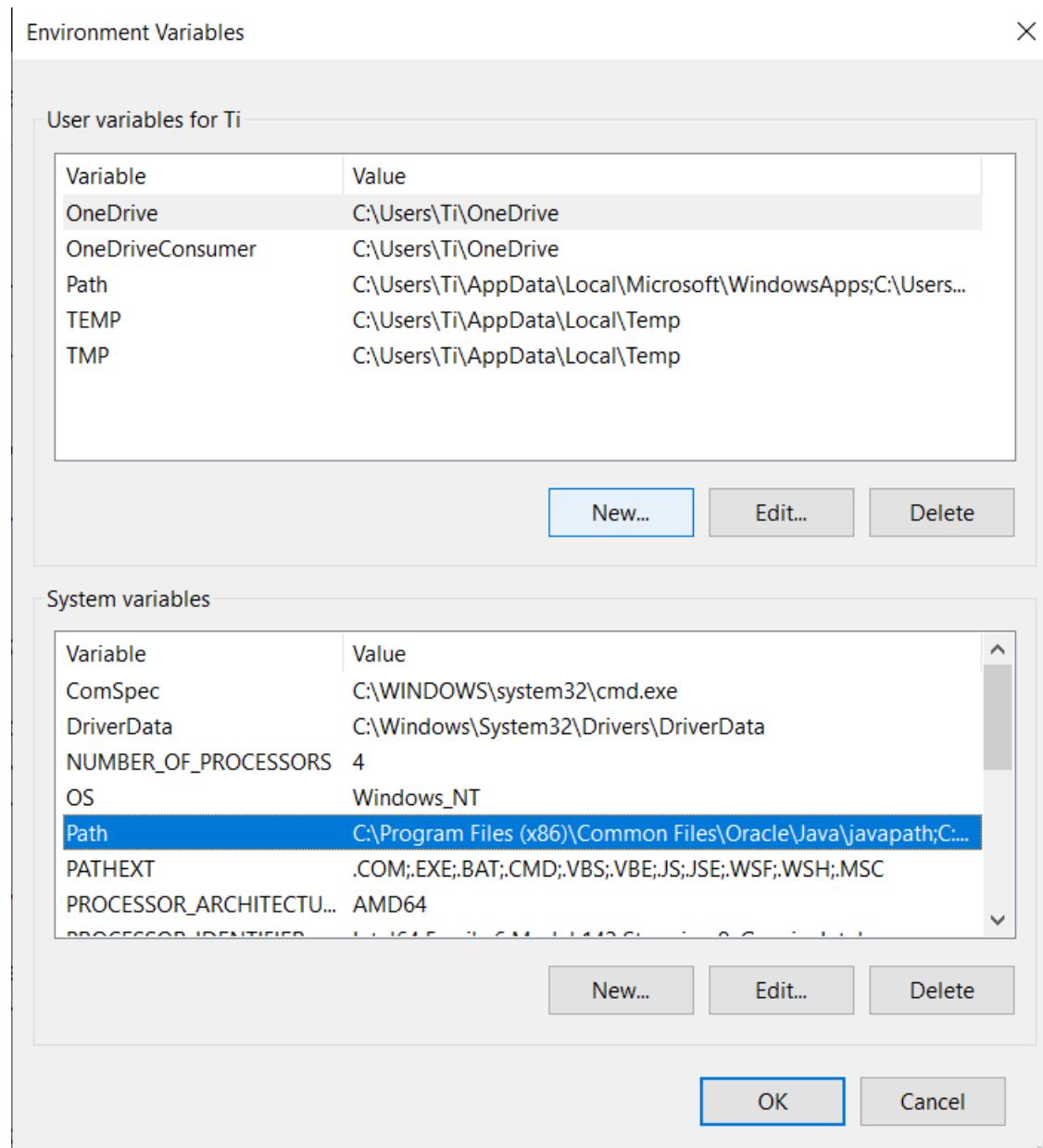
C:\Users\Ti>qemu-system-i386
'qemu-system-i386' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Ti>
```

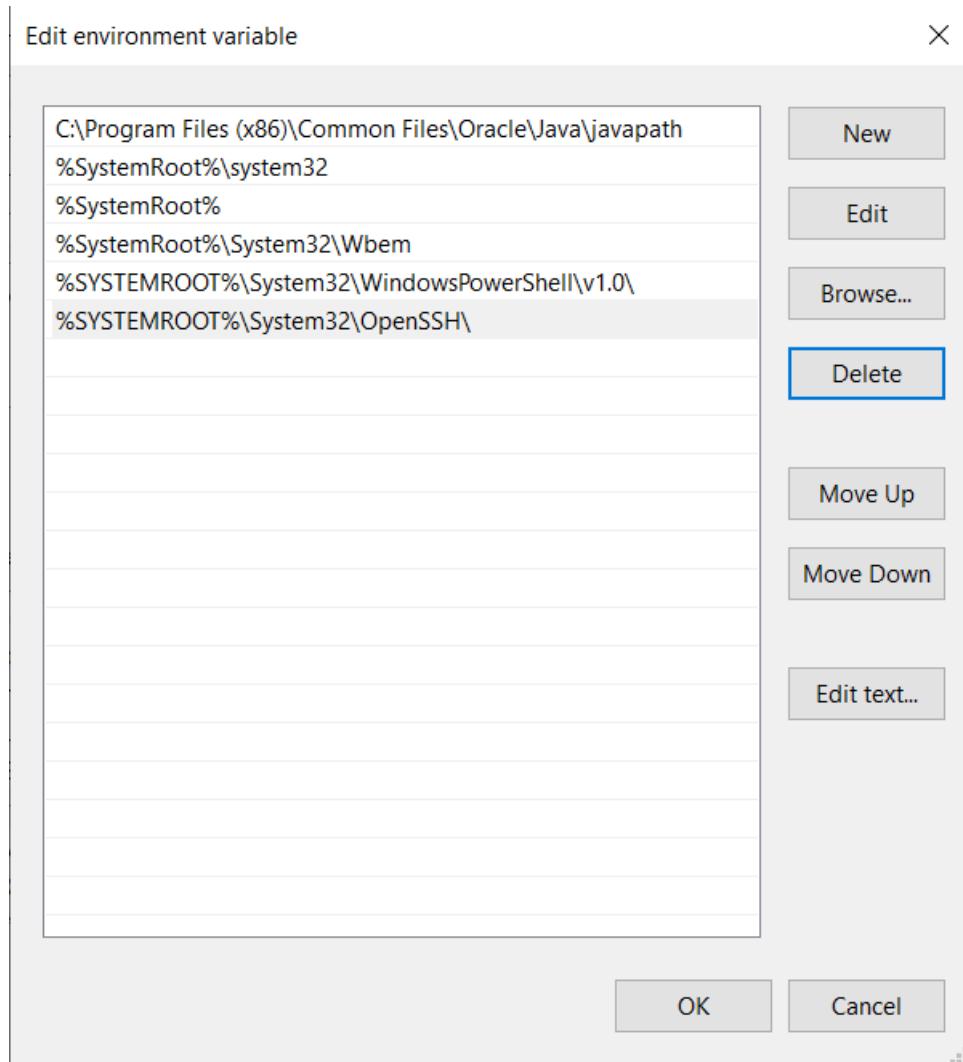
We need to access these softwares by typing it's name in the command prompt, to do that, search 'Edit environment variables' in the search box and select the first option



Now select 'Environment Variables...' button in the 'Advanced' tab,
In the new window, click on Path in the System variables section and click edit



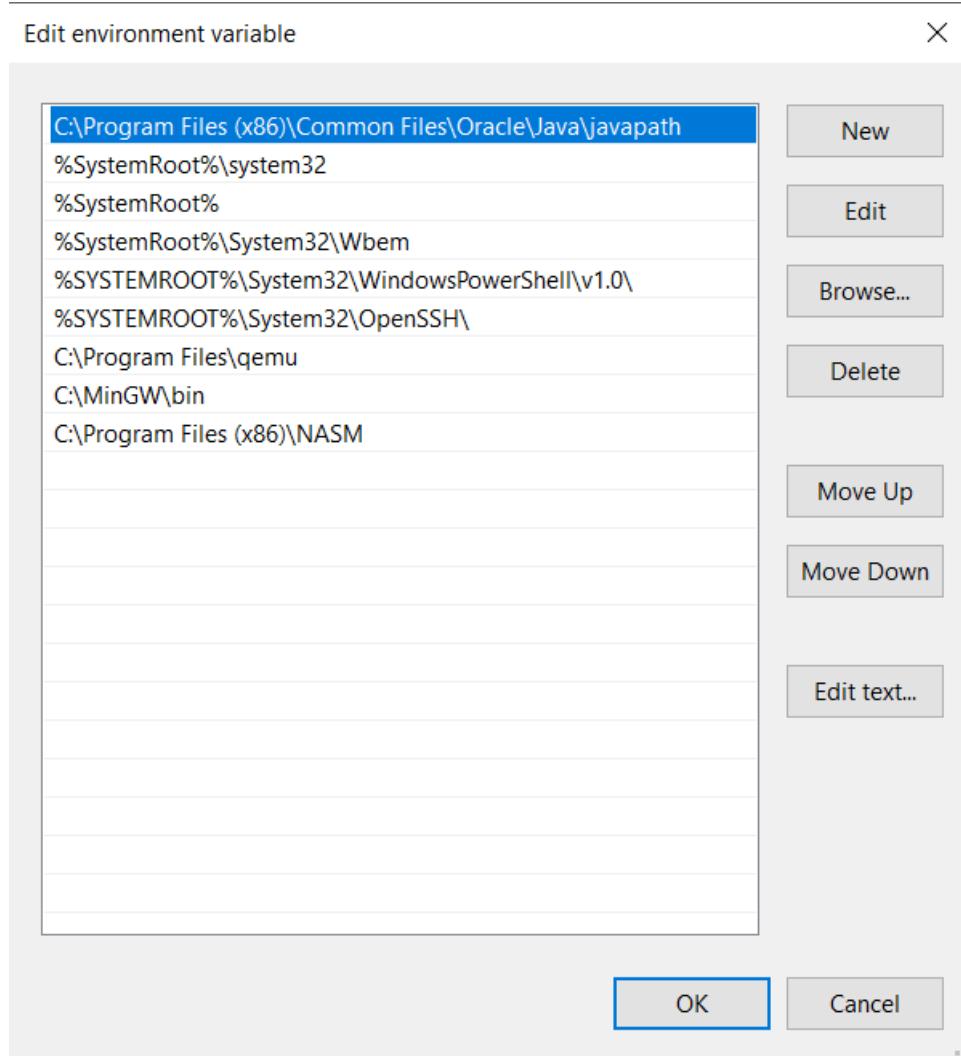
You will now get a window like the one below(Please note that all system may not be same like this as different systems could have different settings, but it will look somewhat like this):



Now you need to add the location to the bin folder of three of the softwares that we downloaded by clicking New button and pasting it there. You need to set the location of Qemu , Nasm and MinGw. In My system the following is the location to those folders.

- 1)Qemu : C:\Program Files\qemu
- 2)MinGw : C:\MinGW\bin
- 3)Nasm : C:\Program Files (x86)\NASM

Now, After adding all these locations, the window looks somewhat like this(Please note that, the location of these application i show here might not be the case for you, It will depend on the settings you adjusted when installing those softwares):



Now close all of the existing windows only by pressing OK.

Now you could type `gcc`, `nasm` and `qemu-system-i386` in the cmd and you will get some messages(Probably like 'Input file missing' , but that's ok)

We are now ready to start getting our hands dirty by doing , LETS GOOOO!!!!!!

Programming In C

Introduction

C is generally called as a high level language, but some of them consider it as a mid level language. The reason being that it does not provide as lot of abstraction which is generally given by languages such as java, python and other new trendy languages.

There is a great chance for your application to get crashed probably because of memory corruption or wrong system calls etc when using c. If you are new to c, don't be scared as everything can be learned by doing. Learning advanced c will make you more knowledgeable about the working of computers. In this book, we will only use very basic concepts of c, We will not use any advanced concepts and will try to keep it the simplest as possible. This is so that, by using this simple methods , there is a great chance for you to not get demotivated.

A common misconception about c is that, it is a program which converts human readable source code to binary. But c is only a programming standard which defines how the compiler should accept instructions. What c does is only translate your source code to it's assembly equivalent of a specific processor architecture. We will discuss it and more in later chapters.

If you know basic c concepts, You could skip this chapter, But if you are not being in touch with c for a long time and needs a way to refresh the concepts, you could read this chapter, But i will try to keep it as simple as possible so that even beginners could start.

If you are interested, you should also master the c programming language along with the development of your own os. This is the technique i used to learn c. I learned c by making a small 2d game. Learning by doing and asking questions is a great way to practice. Asking question is not a bad thing , everyone who knows programming or any skill used to ask questions during their learning days.

Note that this chapter won't give you full coverage of c programming, but we will discuss everything necessary which can't be avoided when reading this book.

So lets dive into THE C PROGRAMMING LANGUAGE!!!!.

Hello , World

Hello , World is the first program used in most tutorials to introduce programming. What this does is only print some text to the screen(Commonly "Hello , World" itself).

Lets see how we could print "Hello , World" to the terminal using c.

First, open Notepad++ and type or copy paste the following code and save it:

```
1 #include <stdio.h>
2
3 int main(){
4     printf("Hello , World");
5 }
```

Please note that you need to give .c as file extension.

Compile the code by opening cmd, going to the directory where you saved the code using the cd command and finally compile it using this command:

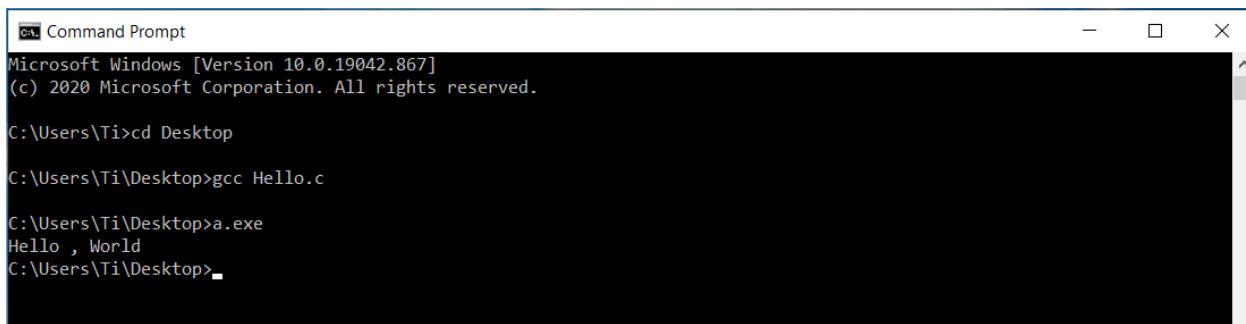
```
gcc code.c
```

You need to replace code.c with the file name you saved.

Now you will see an executable file named a.exe

Now type a.exe in that command prompt itself to start it.

You will see something like this in your command prompt:



The screenshot shows a Microsoft Windows Command Prompt window. The title bar says "Command Prompt". The window content is as follows:

```
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Ti>cd Desktop
C:\Users\Ti\Desktop>gcc Hello.c
C:\Users\Ti\Desktop>a.exe
Hello , World
C:\Users\Ti\Desktop>
```

This is a simple program to display a message to the screen, You could replace content in the printf function inside quotes for example if you put that line like:

```
printf("Hello David");
```

It will print Hello David to screen.

Here the computer tries to print whatever is present in the “ “ Quotes.

printf is a function which is predefined in the stdio.h header file which we included at the very top. We will learn what a header file is later.

A thing to note is that after every statement in a line, we need to put a ; at the end.

You could also see a portion:

```
int main(){
}
```

This is where the computer starts execution, So every program needs this main function to start execution.

Please note that you cannot call the printf function to display to the screen when developing the os as what this printf function does requires an operating system to display it. And as the only operating system in our virtual environment when testing the os is our os, We need to build some code so that when it is called, The text would be printed. We will see that later.

Lets now start learning the basics of c.

Data Types

Data types are a concept introduced to hold data. different data types have different storage capacities. Every data type have its own use.

Basic Data Types

Basic data types include `int` , `char` , `float` , `double` etc. These data types are generally used to hold arithmetic values. We can prefix `signed` and `unsigned` keywords before these keyword to alter its data holding limit.

We will learn about `int` and `char` data types.

int Data Type

`int` is the data type specifically used to store larger numbers. The size of `int` data type is mostly the bit length of the cpu it is running on.

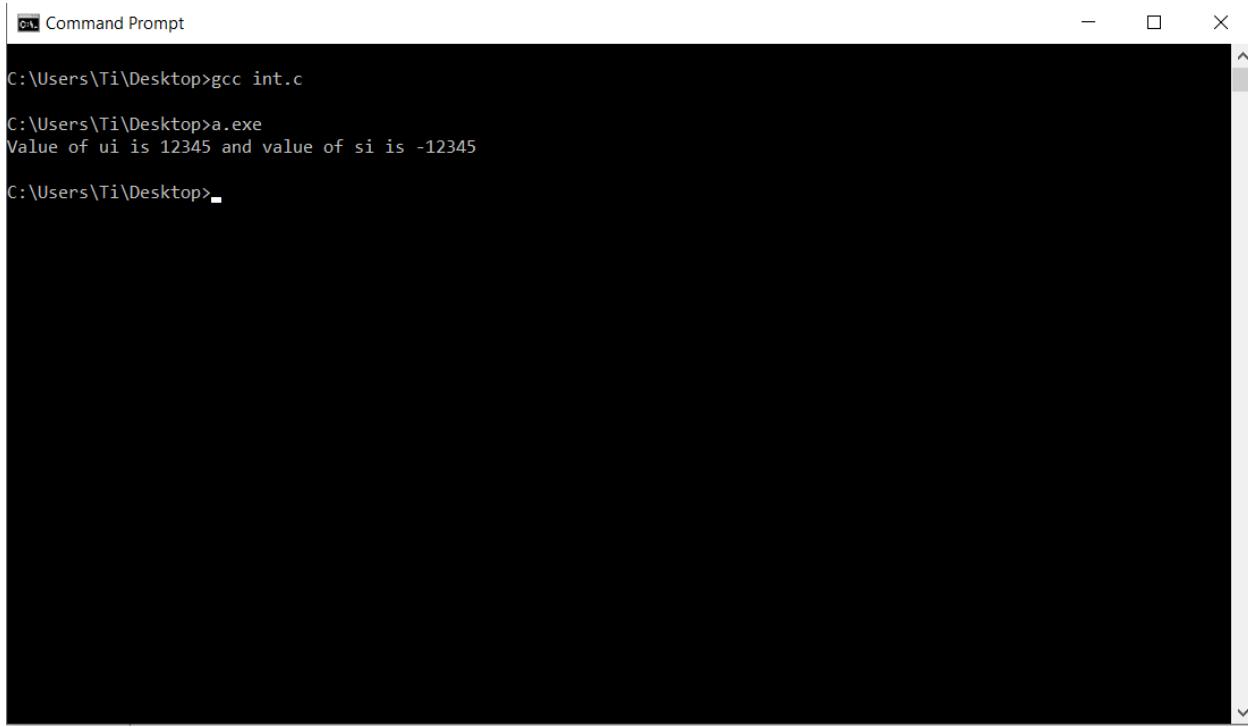
`unsigned int` can hold a value from 0 to 65,535 or 0 to 4,294,967,295

`signed int` can hold a value from -32,768 to 32,767 or -2,147,483,648 to 2,147,483,64. By default integers are signed so, You do not need to put `signed` keyword before `int` keyword to make it signed.

Lets see how we can assign and print some value from both `signed` and `unsigned int`.

```
1 #include <stdio.h>
2
3 int main(){
4
5     unsigned int ui = 12345;
6     signed int si = -12345;
7
8     printf("Value of ui is %d and value of si is %d\n" , ui , si);
9
10 }
```

compile and run the program, we will see something like this:



```
Command Prompt

C:\Users\Ti\Desktop>gcc int.c
C:\Users\Ti\Desktop>a.exe
Value of ui is 12345 and value of si is -12345
C:\Users\Ti\Desktop>
```

You can see that both the negative and positive numbers get printed. But the main effect of signed and unsigned keywords work during branching operations. We will see that later.

Here, We first assigned some values to the variables with the = operator.

Then you can see the characters %d in printf function two times. When executing the program, computer will replace the first %d with the value of ui and the second %d with the value of si. %d helps us printing integer values which is passed after ; token.

The characters \n is put so that it puts a line break in the output.

Please try assigning values to both signed and unsigned int which crosses its limit and try printing it. Look what the result will be and try studying the reason.

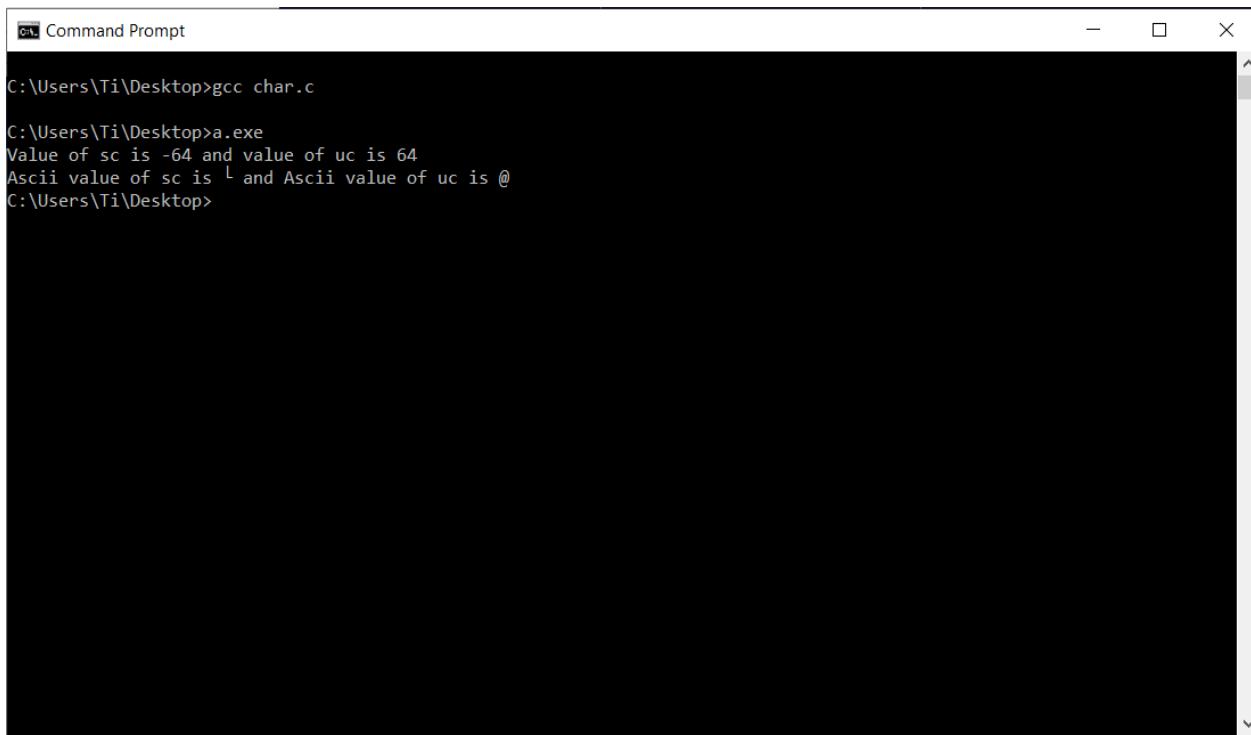
char Data type

char data type is specifically used to store ascii characters.

it is a one byte data type. signed char could store value from -128 to 127. unsigned char could store value from 0 to 255

```
1 #include <stdio.h>
2
3 int main(){
4
5     signed char sc = -64;
6     unsigned char uc = 64;
7
8     printf("Value of sc is %d and value of uc is %d\n" , sc , uc);
9     printf("Ascii value of sc is %c and Ascii value of uc is %c" , sc , uc);
10
11 }
```

Output will be like this when compiled and executed



```
Command Prompt
C:\Users\Ti\Desktop>gcc char.c
C:\Users\Ti\Desktop>a.exe
Value of sc is -64 and value of uc is 64
Ascii value of sc is L and Ascii value of uc is @
C:\Users\Ti\Desktop>
```

You could see in the source code that we used characters %c in it. %c is used to print the ascii character of value specified to it after the ‘;’ token

void Data Type

void is a data type used to represent absense of value. We will see the use of it later.

Derived Data Types

Derived data types are data types which are derived from fundamental data types like `int` , `char` etc...

Derived data types include:

- 1) Pointers
- 2) Arrays
- 3) Structures
- 4) Unions
- 5) Functions

Pointers

A pointer is a data type which holds memory addresses. This could be used to access data in a specific location. Every data in computers memory have an address, If we know the address of some specific data that we want, we could use pointers to access it.

Pointer data type do not stand on its own. We need some other keywords to work on pointers

We could first try declaring a `char` pointer. Have a look at the following code:

```
1 #include <stdio.h>
2
3 int main(){
4
5     char *Pointer = "Pointers are powerfull\n";
6     printf("%s" , Pointer);
7
8 }
```

Have a look at the output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc pointer.c
C:\Users\Ti\Desktop>a.exe
Pointers are powerfull
C:\Users\Ti\Desktop>
```

We have declared a pointer by prefixing a `*` before a variable name.

The one we declared is a character pointer. If you want to declare a `char` pointer with `addr` as pointer name, we could declare it as:

```
char *addr
```

To assign it with the address of a string such as "Pointers are useful", Just declare it like this:

```
char *addr = "Pointers are useful";
```

You could see that we used the character %s in the code. %s accepts an address to a string. We passed the address by typing the name Pointer after ' ';

We could now study pointers deeply. Look at the code up above. Here, what it does is, it declared a character pointer named addr and assigned it with the address of the string "Pointers are useful". With that code we are only assigning the address of that string, not the string itself.

I will say why we used the keyword char to declare the pointer, Have a look at the following code:

```
1 #include <stdio.h>
2
3 int main(){
4
5     char *Pointer = "COMPUTERS are powerfull\n";
6     printf("%c" , *Pointer);
7
8     Pointer++;
9     printf("%c" , *Pointer);
10
11    Pointer++;
12    printf("%c" , *Pointer);
13
14    Pointer++;
15    printf("%c" , *Pointer);
16
17    Pointer++;
18    printf("%c" , *Pointer);
19
20    Pointer++;
21    printf("%c" , *Pointer);
22
23    Pointer++;
24    printf("%c" , *Pointer);
25
26    Pointer++;
27    printf("%c" , *Pointer);
28
29    Pointer++;
```

```
30     printf("%c" , *Pointer);
31
32 }
```

Lets see the output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc char_pointer.c
C:\Users\Ti\Desktop>a.exe
COMPUTERS
C:\Users\Ti\Desktop>
```

Here, we have printed the string "COMPUTERS" on screen. At the very top, we have declared a variable and assigned it with the address of a string. Then we have printed a character like this:

```
printf("%c" , *Pointer);
```

Here, we included a * before the variable name. What this does is to get whatever is present in the address specified in the variable Pointer.

If we didn't put the star before it, it will print the data contained in the variable Pointer which is an address, but as we put a star before it, it will take the data in the variable named Pointer as an address and returns whatever is located at that address.

By default the variable Pointer points to the first character of the string. So after printing the first character, We issued the command:

```
Pointer++;
```

What it does is only increment the address in Pointer by one byte(which is the size of data type char defined for the pointer). Now if we print the data pointed by the Pointer variable, We will print the second character of the string and this process continues with each increment and printing to the screen.

We could see how to get the address of a variable, for example the address of an integer and print whatever is in it:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int num = 2021;
6     int *pnum = &num;
7     printf("address at pnum is %d and value at pnum is %d" , pnum , *pnum);
8
9 }
```

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc int_pointer.c
C:\Users\Ti\Desktop>a.exe
address at pnum is 6422296 and value at pnum is 2021
C:\Users\Ti\Desktop>
```

Here, we first declared an integer named `num` and assigned it with `2021`. Then we issued the following command:

```
int *pnum = &num;
```

This will declare an integer pointer named `pnum` and assign it with the address of value in `num` with the `&` symbol. The `&` symbol will return the address of value in variable `num`. Then we issued the following command:

```
printf("address at pnum is %d and value at pnum is %d" , pnum , *pnum);
```

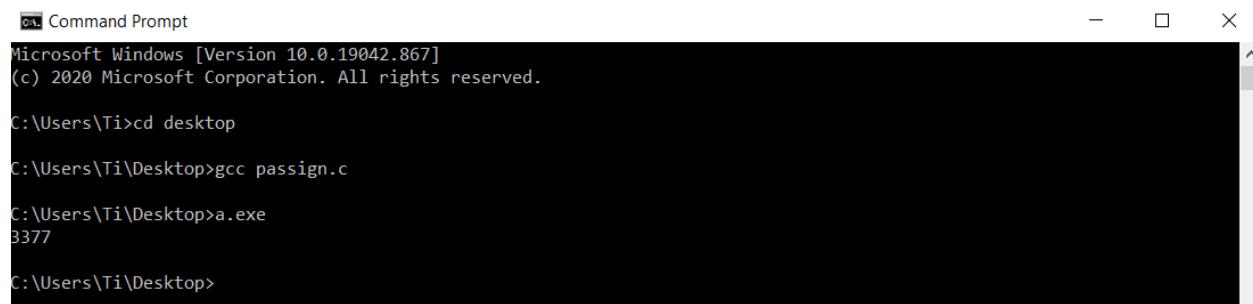
First we printed whatever is present in the variable `pnum` (As we assigned it with the address of `num`, this will print the memory address of `num`). Then the command `*pnum` will take whatever is present in the variable `pnum` as an address and tries to print the data pointed by that address which in this case is the value `2021`.

Let's now see how we could assign values to a specific memory address using pointers:

```

1 #include <stdio.h>
2
3 int main(){
4
5     int iv = 0;
6
7     int *address = &iv;
8
9     *address = 3377;
10
11    printf("%d\n" , iv);
12
13 }
```

Output



```

Command Prompt
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Ti>cd desktop
C:\Users\Ti\Desktop>gcc passign.c
C:\Users\Ti\Desktop>a.exe
3377
C:\Users\Ti\Desktop>
```

First, We created an integer named `iv` and assigned it with `0`. And in the second line, we created an integer pointer named `address` and assigned it with the address of variable `iv`.

The next line is as follows : `*address = 3377;`. Here what it does is that, it first takes the value in variable `address` as a memory address and assigns the value `3377` to that address.

Here, as we previously assigned the address of variable `iv` to the variable named `address` , The line : `*address = 3377;` will assign `3377` to the memory location of the variable `iv`.

We confirmed that the value got assigned by printing the value in `iv`.

So, This way we could use pointers to also **ASSIGN** values.

Arrays

Arrays are a collection of specific data type. An `integer array` means a collection of `integers` and `char array` means a collection of `characters`. Arrays are always one after other. We can access each element of array with its `index`. For example if we want to access the `first` element of an array we pass `0` as the index, and to get the `second` element, we pass `1` as the index and this process goes on.....

Lets see how we could declare and access an array of integers:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int num[5];
6
7     num[0] = 12;
8     num[1] = 23;
9     num[2] = 34;
10    num[3] = 2020;
11    num[4] = 8281;
12
13    printf("%d %d %d %d %d" , num[0] , num[1] , num[2] , num[3] , num[4]);
14
15 }
```

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc int_arr.c
C:\Users\Ti\Desktop>a.exe
12 23 34 2020 8281
C:\Users\Ti\Desktop>
```

Here, we declared an `integer` `array` of 5 elements with this code:

```
int num[5];
```

We can access or assign all of the elements with `num[0]` to `num[4]`. Please note that the index of an array starts from 0 to `size_of_array - 1`. Here, it is from 0 to 4.

Please look how we assigned values to each element and printed it.

Now let me reveal a secret. The arrays work somewhat as same as pointers. The only difference is that pointers use `*` and arrays use `[]`.

Let me show how we could access the array with pointers, Have a look at the code below:

```

1 #include <stdio.h>
2
3 int main(){
4
5     int num[5];
6
7     num[0] = 5;
8     num[1] = 4;
9     num[2] = 3;
10    num[3] = 2;
11    num[4] = 1;
12
13    printf("%d %d %d %d %d" ,*(num + 0),*(num + 1),*(num + 2),*(num + 3),*(num + 4));
14
15 }
```

Output:



```

C:\Users\Ti\Desktop>gcc arr_pont.c
C:\Users\Ti\Desktop>a.exe
5 4 3 2 1
C:\Users\Ti\Desktop>
```

Here, We declared the array and assigned the value using the technique we learned just before this. But have a look at how we printed it, It is different from what we have previously done. We've used the following technique:

`*(num + 1)`

What this does is like this : It first does thing which is inside the brackets, It increments 4 to the `num` variable(We have actually incremented 1, but as the size of datatype of the variable `num` is 4 bytes , incrementing one will practically increments four bytes, incrementing two will increase eight bytes and so on), So now the `num` variable points to the second integer which is just after `*(num + 0)`.

And as we have put a * before `(num + 1)`, it will take `(num + 1)` as an address and take whatever is present in that address. This will practically access the second integer, likewise `*(num + 2)` will access the third integer.

I will request you to play with these concepts and do some more research so that you could learn more about this.

We will use pointers extensively when developing our operating system such as when we make the display driver. In 32 bit protected mode(We will explain that later), we need to put the data we

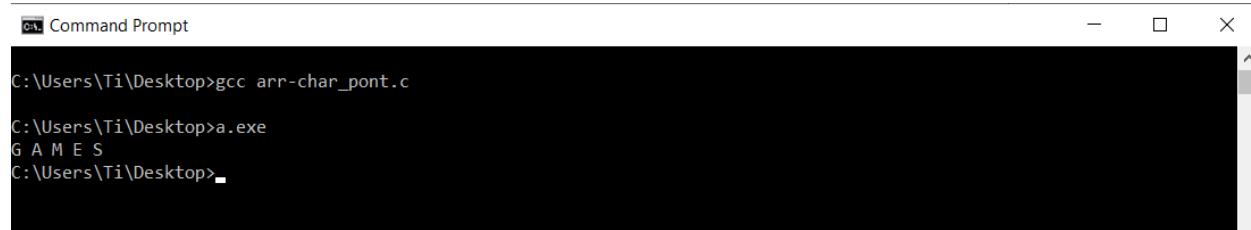
want to print to the screen to a specific address in memory, Which the video card will render to the screen, But for now, as i have requested , please work with pointers for some time to get the full idea of pointers

Lets now look at character array and work with pointers on it.

```

1 #include <stdio.h>
2
3 int main(){
4
5     char ca[5];
6
7     ca[0] = 'G';
8     ca[1] = 'A';
9     ca[2] = 'M';
10    ca[3] = 'E';
11    ca[4] = 'S';
12
13    printf("%c %c %c %c %c" ,*(ca + 0), *(ca + 1), *(ca + 2), *(ca + 3), *(ca + 4));
14
15 }
```

Output:



```

Command Prompt
C:\Users\Ti\Desktop>gcc arr-char_pont.c
C:\Users\Ti\Desktop>a.exe
G A M E S
C:\Users\Ti\Desktop>
```

Here, the only difference is that, Instead of printing numbers, We just printed characters. You can try studying this code and move further.

I believe you are getting clear about the concept of pointers. You should further study pointers your own, there are many more concepts relating to pointers such as pointer to pointer etc....

Functions

Functions are a block of code which could be called and does a specific task. You have already used a function which is the main function:

```
int main(){
}
```

This is an example of a function and it is named `main`. We could create our own functions if we need. Look at the following example to get to know about functions:

```

1 #include <stdio.h>
2
3 int add(int a , int b);
4 int sub(int a , int b);
5
6 int main(){
7
8     int num = add(150 , 50);
9     printf("%d\n" , num);
10
11    int num2 = sub(200 , 50);
12    printf("%d" , num2);
13 }
14
15 int add(int a , int b){
16     int c = a + b;
17     return c;
18 }
19
20 int sub(int a , int b){
21     int c = a - b;
22     return c;
23 }
```

Output:



```

Command Prompt
C:\Users\Ti\Desktop>gcc functions.c
C:\Users\Ti\Desktop>a.exe
200
150
C:\Users\Ti\Desktop>
```

Let me explain it, Take a look at the `add` function that we have just created:

```

int add(int a , int b){
    int c = a + b;
    return c;
}
```

We have defined three integer variables here : `a` , `b` and `c`.

`a` and `b` are in the round brackets `()`. This means that the value of `a` and `b` will be given when calling it.

we called this function from the main function like this:

```
int num = add(150 , 50);
```

Here, what the computer will do at runtime is that it passes the number 150 to the integer variable named `a` in `add` function and 50 to the integer variable named `b` in the `add` function.

After passing these arguments the computer starts executing the code in `add` function. In that function, the code adds the values in `a` and `b` and stores it in `c`. Then it returns the value in `c` to the variable `num` in main function. Then the main function prints it. We have also seen another function named `sub`.

```
int sub(int a , int b){  
  
    int c = a - b;  
    return c;  
  
}
```

There is no special difference in the working of the code when this function is called with:

```
int num2 = sub(200 , 50);
```

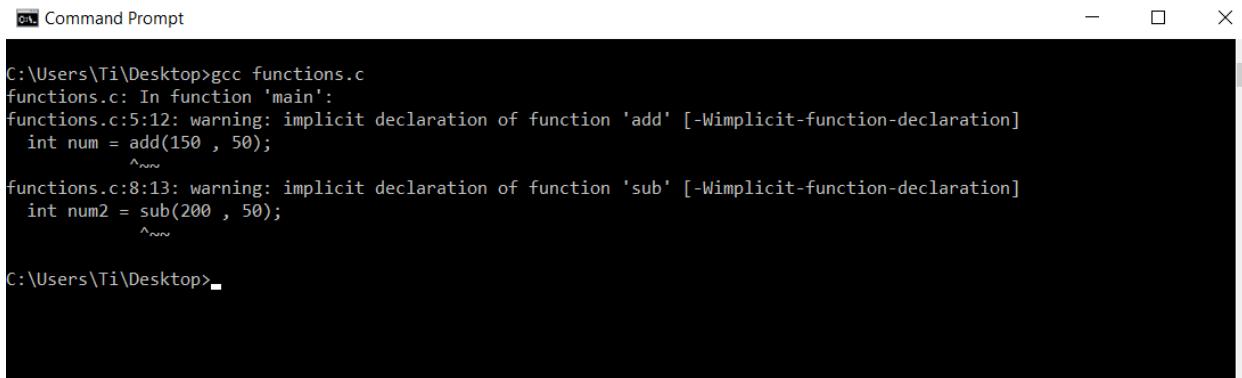
The only difference is that, the two numbers which got passed will go under a subtraction operation and it returns the result to the `num2` variable in main function.

At the very top of the code you could see the following code:

```
int add(int a , int b);  
int sub(int a , int b);
```

What this does is only telling the compiler that we will include code for those function later under the main function. If you do not do that before the main function, the compiler would stuck at what it is doing when it encounters the function call to `add` and `sub` functions because it haven't seen a function like that to jump to it.

You could try removing that two lines from the top and it will print some error message like this:



```
Command Prompt

C:\Users\Ti\Desktop>gcc functions.c
functions.c: In function 'main':
functions.c:5:12: warning: implicit declaration of function 'add' [-Wimplicit-function-declaration]
    int num = add(150 , 50);
               ^~~
functions.c:8:13: warning: implicit declaration of function 'sub' [-Wimplicit-function-declaration]
    int num2 = sub(200 , 50);
               ^~~
C:\Users\Ti\Desktop>
```

Lets learn something about the `return` type of a function. You can see that i typed the `add` function like this:

```
int add(int a , int b){

    int c = a + b;
    return c;

}
```

Not like this:

```
add(int a , int b){

    int c = a + b;
    return c;

}
```

The `int` keyword we gave before the function name `add` is used so that the compiler could get an idea about the data type of variable which that function will return using the `return` keyword. Here we have returned a variable named `c` which is of `int` type, This is the reason why we prefix the `int` keyword before the function name `add`.

The same rule applies also to the `sub` function.

You cannot define a certain data type as the return type and return variable of another data type. Like this:

```
1 #include <stdio.h>
2
3 char add(int a , int b);
4
5 int main(){
6
7     int num = add(150 , 50);
8     printf("%d\n" , num);
9
10 }
11
12 char add(int a , int b){
13     int c = a + b;
14     return c;
15 }
```

Some compilers will show error when doing something like this but our compiler haven't shown any error , but you could see when executing that program that the output will be wrong

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc functions2.c
C:\Users\Ti\Desktop>a.exe
-56
C:\Users\Ti\Desktop>
```

It is possible so that you can make a function which do not return any value, for that, you could use the **void** keyword.

```
1 #include <stdio.h>
2
3 void printHello();
4
5 int main(){
6     printHello();
7 }
8
9 void printHello(){
10
11     printf("Hello!!");
12 }
```

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc void.c
C:\Users\Ti\Desktop>a.exe
Hello!!
C:\Users\Ti\Desktop>
```

Here, we have returned no value. We just jumped to a function and printed some characters to the screen from there. Then the control goes back to the main function and when at the end of the main function, the program itself terminates.

MORE INFO

You could return from any function(with or without return type) from anywhere in the code with the `return` keyword. You only need to type `return;` and if that code gets executed , the program control will go back to the function which called it.

You could make a function(with or without return type) With many , less or no arguments Like this:

```
void print();
int add();
void add(int a , int b);
int add (int a , int b);
```

You could call any type of function(with or without arguments) Like this:

```
PrintMessage();
Add(a , b);
```

Please note that the name of the function and/or variable is of your choice. I used these names just for example

I will suggest you to play with the concept of functions and also the overall concepts that we have covered this far before jumping to the next section.

Structures

Structures help us to combine a collection of data type into one packet. These data types may or may not be of the same type, Let's see an example:

```

1 #include <stdio.h>
2
3 struct car{
4     int numberOfTyres;
5     int price;
6     char topSpeed;
7 };
8
9 int main(){
10     struct car goldenCar;
11     struct car yellowCar;
12
13     goldenCar.numberOfTyres = 4;
14     goldenCar.price = 2000;
15     goldenCar.topSpeed = 100;
16
17     yellowCar.numberOfTyres = 2;
18     yellowCar.price = 4000;
19     yellowCar.topSpeed = 110;
20
21     printf("GOLDEN CAR : %d , %d , %d\n", goldenCar.numberOfTyres, goldenCar.price, gol\
22 denCar.topSpeed);
23     printf("YELLOOW CAR : %d , %d , %d\n", yellowCar.numberOfTyres, yellowCar.price, ye\
24 llowCar.topSpeed);
25 }
```

Output



```

C:\Users\Ti\Desktop>gcc struct.c
C:\Users\Ti\Desktop>a.exe
GOLDEN CAR : 4 , 2000 , 100
YELLOOW CAR : 2 , 4000 , 110
C:\Users\Ti\Desktop>
```

Here, We first created a structure named `car` with the `struct` keyword.

Then we included three variables named `numberOfTyres` , `price` and `topSpeed`. After closing the structure with `}`, we put a `;` denoting the end of that structure.

In the `main` function, we created two instances of the structure named `car` with it's name as `goldenCar` and `yellowCar`.

We did it with the command `struct car goldenCar;` and `struct car yellowCar;`

This command will allocate memory space to hold values in the structure named `car` and we could

access those areas with its name (`goldenCar` and `yellowCar`).

Both `goldenCar` and `yellowCar` will have separate space to hold all of the three variables in the structure `car`.

Then we assigned values to variables in both of the instances of structure `car` by combining the instance name and variable name with a dot(.) .

In the command : `goldenCar.numberOfTyres = 4;`, we assigned the value 4 to the variable `numberOfTyres` of instance `goldenCar` of `car` structure.

The next two lines assigned values to the rest of the variables in the instance of structure `car`.

Later, we assigned values to variables in the instance named `yellowCar` of the `car` structure.

At last we printed all of the variables in both `goldenCar` and `yellowCar` instance of `car` structure.

Branching

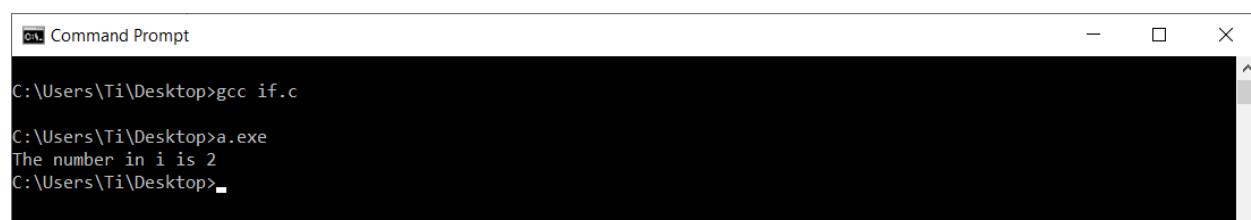
Branching is a concept which is used so that you could execute some piece of code based on some conditions. The keyword used for this is the 'if' Keyword.

Lets look at an example:

```

1 #include <stdio.h>
2
3 int main(){
4
5     int i = 2;
6
7     if(i == 2){
8         printf("The number in i is 2");
9     }
10
11 }
```

Output:



```

C:\Users\Ti\Desktop>gcc if.c
C:\Users\Ti\Desktop>a.exe
The number in i is 2
C:\Users\Ti\Desktop>
```

Here, We first assigned the value 2 to the integer variable named `i`. Then we checked in the `if` condition whether the value in `i` is 2 or not with the `==` operator.

As the value in the variable `i` is 2, The program enters inside the condition and prints the message. There are different operator which we could use in the `if` condition which are:

- 1) `==` : Equals to
- 2) `!=` : Not equal to
- 3) `>=` : Greater than or equal to
- 4) `<=` : Less than or equal to
- 5) `>` : Greater than
- 6) `<` : Less than

You could use any of these operators in the `if` condition. Lets see another example:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int i = 2;
6
7     if(i != 2){
8         printf("The number in i is not 2");
9     }
10 }
```

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc ifn.c
C:\Users\Ti\Desktop>a.exe
C:\Users\Ti\Desktop>
```

You could see that, no message is printed, This is because we initialized `i` with 2, Then we checked in the condition if the value in `i` is not 2. But actually as the value in it is 2, The `printf` function will not execute and the program will exit doing nothing.

Lets now see how to make that program running by making only a small change:

```

1 #include <stdio.h>
2
3 int main(){
4
5     int i = 4;
6
7     if(i != 2){
8         printf("The number in i is not 2");
9     }
10
11 }

```

Output:



```

Command Prompt

C:\Users\Ti\Desktop>gcc ifn4.c
C:\Users\Ti\Desktop>a.exe
The number in i is not 2
C:\Users\Ti\Desktop>

```

Here, We initialized *i* with 4. Then we checked if value in *i* is not 2 and as the value in *i* is not 2, The `printf` function got executed.

We also have another keyword to work with branching which is the `else` keyword. Code in this `else` part will get executed if the condition checked with the `if` keyword is not `true`. Have a look at the following code:

```

1 #include <stdio.h>
2
3 int main(){
4
5     int i = 2;
6
7     if(i != 2){
8         printf("The number in i is not 2");
9     }
10    else{
11        printf("The number is 2");
12    }
13
14 }

```

Output:



```
C:\Users\Ti\Desktop>gcc else.c
C:\Users\Ti\Desktop>a.exe
The number is 2
C:\Users\Ti\Desktop>
```

Here we initialized `i` with 2. Then we checked in the `if` condition if the value in `i` is not 2. But as the value in `i` is 2, The `else` part gets executed and the message got printed.

Now we will learn about '`else if`'. '`else if`' helps us check another condition if the condition we checked in the previous `if` keyword is false.

Have a look at the following code:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int i = 2;
6
7     if(i >= 3){
8         printf("Value in i is greater than or equal to 3");
9     }
10    else if(i == 2){
11        printf("Value in i is 2");
12    }
13 }
```

Output:



```
C:\Users\Ti\Desktop>gcc elseif.c
C:\Users\Ti\Desktop>a.exe
Value in i is 2
C:\Users\Ti\Desktop>
```

At here, We first initialized `i` with 2. Then we checked if the value in `i` is greater than or equal to 3. But as the condition is false , the program will start evaluating the condition in '`else if`' part. Now as the '`else if`' condition is true, It will execute the code inside that '`else if`' part and will print the message.

[MORE INFO](#)

You could use the `&&` operator to check whether two or more condition in the `if` keyword are true or not. Like this:

```
if(i >= 2 && i <= 5){  
}
```

You could use `||` as an operator to check if any of two or more conditions specified are true or not. Like this:

```
if(i == 2 || i == 3){  
}
```

In this case, the computer will start execution of code inside the `if` statement if any of the condition is true(If `i` is equal to 2 and/or `i` is equal to 3)

You could use nested `if` cases where one or more `if` cases are inside other `if` case Like this:

```
if(i == 2){  
    if(y < 4){  
        printf("Both conditions are true");  
    }  
}
```

We will now learn about the keyword '`switch`':

`switch` is also a keyword which allows you to check conditions. This is used instead of the '`if`' keyword when we have to check a lot of conditions. Have a look at the following:

```
1 #include <stdio.h>  
2  
3 int main(){  
4  
5     int i = 3;  
6  
7     switch(i){  
8         case 1:  
9             printf("Value is 1");  
10            break;  
11         case 2:  
12             printf("Value is 2");  
13            break;  
14         case 3:
```

```

15         printf("Value is 3");
16         break;
17     case 4:
18         printf("Value is 4");
19         break;
20     case 5:
21         printf("Value is 5");
22         break;
23 }
24 }
25 }
```

Output:



```

Command Prompt
C:\Users\Ti\Desktop>gcc switch.c
C:\Users\Ti\Desktop>a.exe
Value is 3
C:\Users\Ti\Desktop>
```

Here, We first declared a variable named `i` and initialized it with 3.

The next line: `'switch(i)'` will say to compiler to check the value in variable named `i`.

Inside the switch statement after `{` , we put the `'case'` statement like this:

```
case 1:
```

This line will check if the value in `i` is 1 or not. If its 1 , then it will execute every code until a `break;` keyword.

In our case, it will only execute the following line:

```
printf("Value is 1");
```

But , in the program we developed , we initialized variable `i` with 3. So the compiler will jump directly to the this line:

```
case 3:
```

And will execute every code inside this until it see a `break;`

And the only code between `'case 3:'` and `'break'` is `'printf("Value is 3");'` , the processor will only print the string "Value is 3".

There is no limit that you could only call the `printf` function there. You could do any operation you want for eg : calling a function , doing some other branching or anything you want.

After the cpu run any of the case inside the switch statement and after the `break;` keyword, cpu will jump to the code after the switch statement(code after '}').

Here, You could also try changing the value in `i` to get different results.

Please take the time to play with switch keyword.

You should now try playing with all of the concepts that we covered till here. This will help you attain more experience.

Looping

Looping is a concept which makes it practically possible to run same code repeatedly until a condition turns to be false.

To do that, we will use the keyword '`while`'. Have a look at the following code:

```

1 #include <stdio.h>
2
3 int main(){
4
5     int i = 0;
6
7     while(i <= 5){
8         printf("*");
9         i = i + 1;
10    }
11 }
12 }
```

Output:



```

C:\Users\Ti\Desktop>gcc while.c
C:\Users\Ti\Desktop>a.exe
*****
C:\Users\Ti\Desktop>
```

Here, We have defined a variable named `i` and initialized it with `0`. We then declared the `while` keyword and gave the following as condition:

```
i <= 5
```

This line checks whether the value in `i` is less than or equal to 5.

Computer will first check this condition when entering the loop and if its true, it will execute whatever is inside that loop. At the end of the loop at `}`, The computer will go to the top of that while loop and checks the condition and if it's true again, it will continue this work until the condition becomes false.

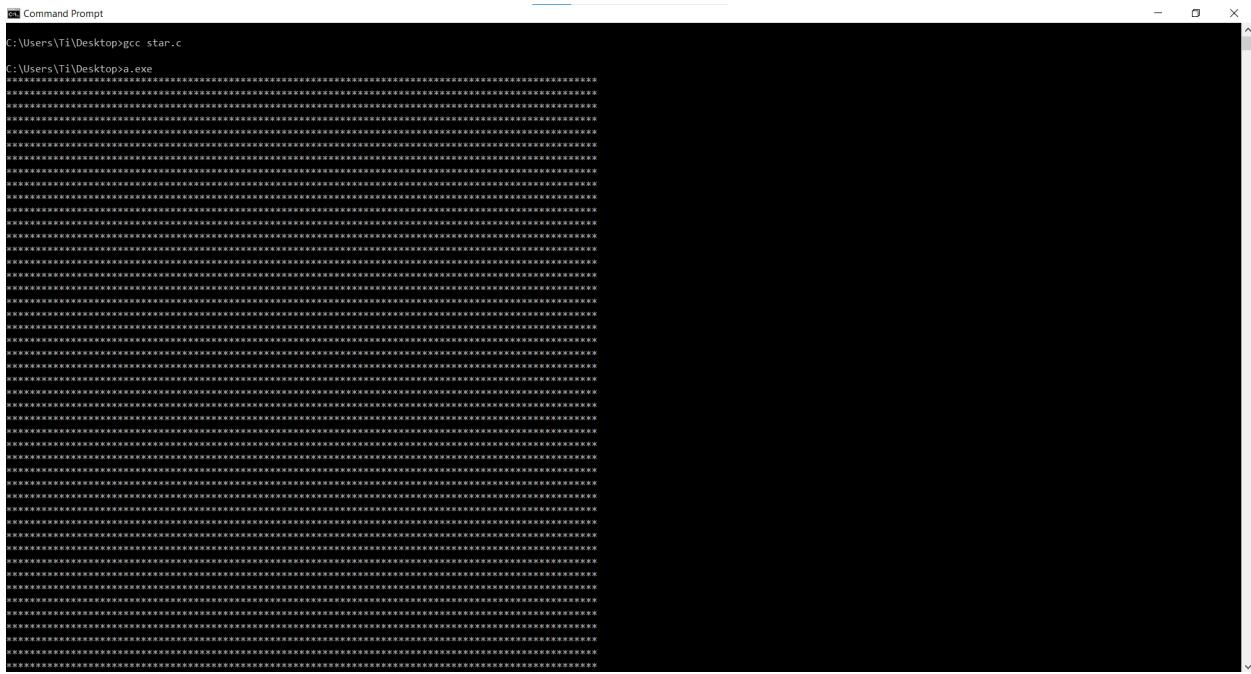
You could use every conditional operator that works in `if` keyword also at the `while` keyword. That conditional operators include `==` , `!=` , `>=` , `<=` , `>` and `<`.

You could use the `&&` and `||` operators which we discussed for '`if`' keyword here.

You could also use nested loops which is a loop inside another loop : There is nothing preventing you to implement something like that. Lets see an example:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int i = 0;
6     int y;
7     while(i <= 150){
8         y = 0;
9         while(y <= 100){
10             printf("*");
11             y++;
12         }
13         printf("\n");
14         i++;
15     }
16
17 }
```

Output:



```
Command Prompt
C:\Users\T1\Desktop>gcc star.c
C:\Users\T1\Desktop>star.exe
*****
```

Here `y++;` is same as `y = y + 1;` and `i++;` is same as `i = i + 1;`

Look at how fast your computer executes this program, This is the POWER and the USE of computers. You should now try to implement programs to make patterns like these. This is a very interesting job and it will help you a lot in this journey.

MORE INFO

You could use `break;` as a keyword so that when it executes, the cpu will exit from the loop it is currently running on

You could also use `continue;` as a keyword so that, when it executes, cpu will jump to the top of the current loop, which practically checks if the condition specified in that loop is true and if it's true, it will continue execution of code inside that loop

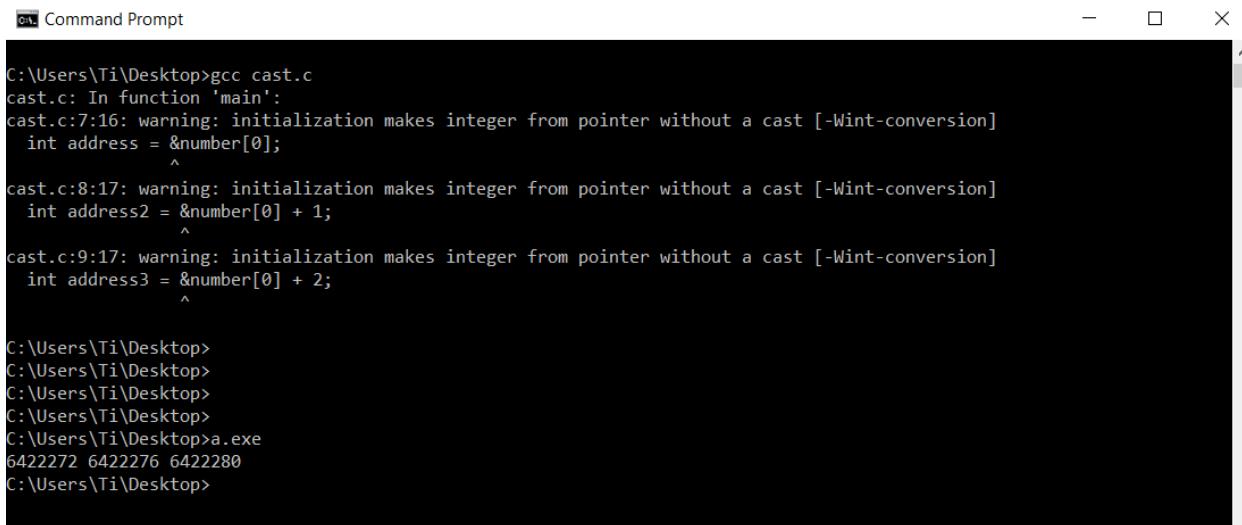
Type Casting

Type casting is a feature which allows us changing one data type to other. Also we could use this to change the behaviour of a special data type to the behaviour of another datatype. We do not have any special keyword to do this but we could do it. Have a look at the following code:

```

1 #include <stdio.h>
2
3 int main(){
4
5     int number[5];
6
7     int address = &number[0];
8     int address2 = &number[0] + 1;
9     int address3 = &number[0] + 2;
10
11    printf("%d %d %d" , address , address2 , address3);
12
13 }
```

Output:



```

Command Prompt
C:\Users\Ti\Desktop>gcc cast.c
cast.c: In function 'main':
cast.c:7:16: warning: initialization makes integer from pointer without a cast [-Wint-conversion]
  int address = &number[0];
               ^
cast.c:8:17: warning: initialization makes integer from pointer without a cast [-Wint-conversion]
  int address2 = &number[0] + 1;
                   ^
cast.c:9:17: warning: initialization makes integer from pointer without a cast [-Wint-conversion]
  int address3 = &number[0] + 2;
                   ^
C:\Users\Ti\Desktop>
C:\Users\Ti\Desktop>
C:\Users\Ti\Desktop>
C:\Users\Ti\Desktop>
C:\Users\Ti\Desktop>a.exe
6422272 6422276 6422280
C:\Users\Ti\Desktop>
```

Here you could see some warnings, You could forget about it for now. The output we got is the following "6422272 6422276 6422280". You could see that each of this address is incremented with 4. This means that address of number [0] is 6422272,

Address of number [0] + 1(or number [1]) is 6422276 and address of number [0] + 2(or number [2]) is 6422280. This is a valid output as the size of an int is 4 bytes, incrementing it with one will always add four.

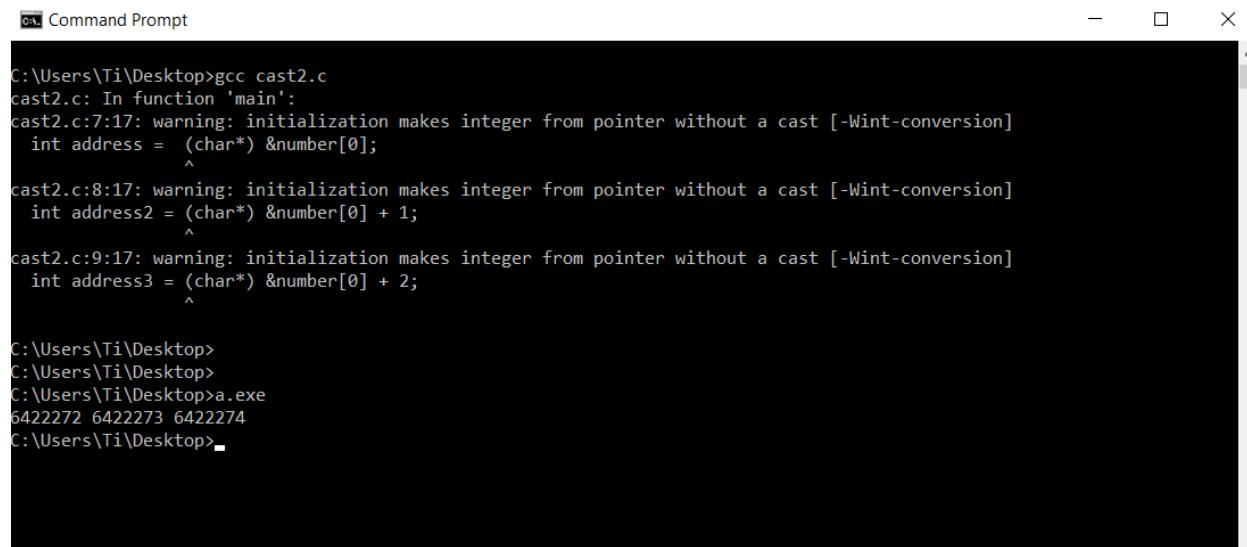
The & symbol here helps us get address of the variable.

But what if you want to increment the address by only one byte which helps you obtain each bytes of a number instead of getting the whole number. You could use casting like this:

```

1 #include <stdio.h>
2
3 int main(){
4
5     int number[5];
6
7     int address = (char*) &number[0];
8     int address2 = (char*) &number[0] + 1;
9     int address3 = (char*) &number[0] + 2;
10
11    printf("%d %d %d" , address , address2 , address3);
12
13 }
```

Output:



```

C:\Users\Ti\Desktop>gcc cast2.c
cast2.c: In function 'main':
cast2.c:7:17: warning: initialization makes integer from pointer without a cast [-Wint-conversion]
  int address = (char*) &number[0];
                  ^
cast2.c:8:17: warning: initialization makes integer from pointer without a cast [-Wint-conversion]
  int address2 = (char*) &number[0] + 1;
                  ^
cast2.c:9:17: warning: initialization makes integer from pointer without a cast [-Wint-conversion]
  int address3 = (char*) &number[0] + 2;
                  ^
C:\Users\Ti\Desktop>
C:\Users\Ti\Desktop>
C:\Users\Ti\Desktop>a.exe
6422272 6422273 6422274
C:\Users\Ti\Desktop>
```

Here, You could see in the output that each address is incremented by only one , not 4.

What we does special in this code is that we added `(char*)`. This says to the compiler that we should only increment the value with the size of `char`(which is one byte), and this will result so that only 1 is added to the address for each increment.

We added `*` to the `(char)` so that it tells the compiler that what we do is an operation on memory address so it looks like this now: `(char*)`.

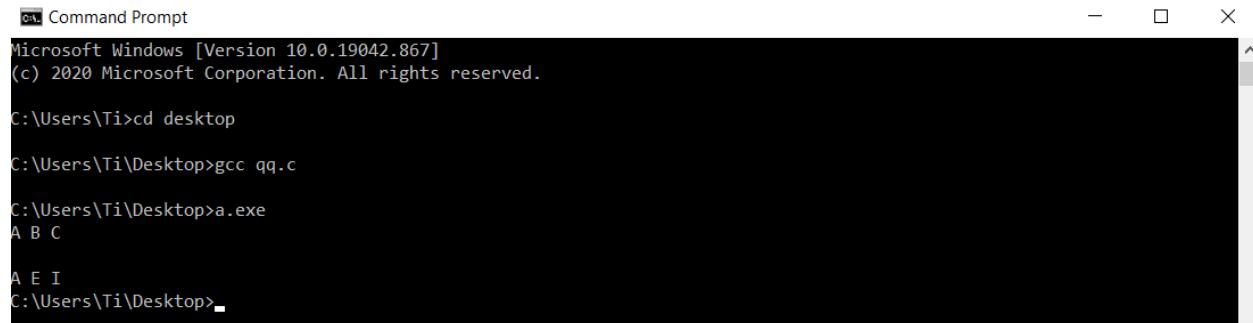
If you did not understand this, Please try reading again, This is not a big deal.

I will show another example:

```

1 #include <stdio.h>
2
3 int main(){
4
5     char *string = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
6
7     char a = *(string + 0);
8     char b = *(string + 1);
9     char c = *(string + 2);
10
11    printf("%c %c %c\n\n" , a , b , c);
12
13    char x = *((int*) string + 0);
14    char y = *((int*) string + 1);
15    char z = *((int*) string + 2);
16
17    printf("%c %c %c" , x , y , z);
18 }
```

Output:



```

Command Prompt
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Ti>cd desktop
C:\Users\Ti\Desktop>gcc qq.c
C:\Users\Ti\Desktop>a.exe
A B C
A E I
C:\Users\Ti\Desktop>
```

Lets study the code now:

The first line is as follows : `char *string = "ABCDEFGHIJKLMNPQRSTUVWXYZ";`

This line will declare a char pointer named `string` and assign it with the address of : `"ABCDEFGHIJKLMNPQRSTUVWXYZ"` in memory.

The next three lines are as follows :

```

char a = *(string + 0);
char b = *(string + 1);
char c = *(string + 2);
```

Here the line: `char a = *(string + 0);` will add 0 to the address specified in the variable `string`. The * before the variable named `string` will result in returning the data pointed by the address in `string` variable to the character variable `a`.

This assigns the first character pointed by the variable `string` to the character `a`;

The only difference in the second line is that instead of adding 0 to the `string` variable , we added 1 to it. This will result in returning the second character pointed by `string` variable to the character variable `b`.

The same follows also in the third line , adding two will result in returning the third character to the variable named `c`.

In three of the cases , You saw that incrementing the value in `string` variable by one result in returning the consecutive characters.

Then we printed the values in `a` , `b` and `c` with: `printf("%c %c %c\n\n", a , b , c);`

This prints the consecutive characters "A B C" to the terminal.

Then we does another approach like this:

```
char x = *((int*) string + 0);
char y = *((int*) string + 1);
char z = *((int*) string + 2);
```

Here we applied an integer casting with `(int*)`.

The first line result in returning the character 'A' To the variable `x`.

Then , in the second line we added 1 to the variable `string`.

Here as we casted it with `(int*)` , adding 1 will actually result in adding 4 which is the size of integer(`int`). And the outer most * in `*((int*) string + 1)` will obtain the character 'E' pointed by address in `string + 1` and return it to the variable `y`.

The same applies to the third line and it obtains the character 'I'. As we added 2 to the variable `string`, it will return the character at `2 * size of int` which is `2 * 4` which is 8.

There are more details we could add to the topic Type casting, But for this tutorial, This much is enough.

NOTE

It is a good idea to learn more on this topic which will help you in advanced implementations

Arithmetic Operators

We have already covered one of the Arithmetic Operations which is the addition operation. We used this to implement examples programs. Here you could learn more:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int base = 4;
6
7     int a = base + 2;
8     int b = base - 2;
9     int c = base * 2;
10    int d = base / 2;
11
12    printf("%d %d %d %d %d" , base , a , b , c , d);
13
14
15 }
```

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc ao.c
C:\Users\Ti\Desktop>a.exe
4 6 2 8 2
C:\Users\Ti\Desktop>
```

The code and the output says it all, But let me explain it.

Addition could be done as:

```
a = x + y;
```

Substraction could be done as:

```
a = x - y;
```

Multiplication could be done as:

```
a = x * y;
```

Division could be done as:

```
a = x / y;
```

Now, Have a look at the following code:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int a = 2;
6     int b = 4;
7     int c = 6;
8     int d = 8;
9
10    a += 2;
11    b -= 2;
12    c *= 2;
13    d /= 2;
14
15    printf("%d %d %d %d" , a , b , c , d);
16
17
18 }
```

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc ao2.c
C:\Users\Ti\Desktop>a.exe
4 2 12 4
C:\Users\Ti\Desktop>
```

I assume that you have already figured out what this code does. But let me explain it:

Here, $a += 2$ is same as $a = a + 2$

$b -= 2$ is same as $b = b - 2$

$c *= 2$ is same as $c = c * 2$

And finally $d /= 2$ is same as $d = d / 2$

We finally have one more operator, Which is the modulus operator. Modulus operator returns a remainder after integer division. Have a look at the following code:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int a = 12;
6
7     int b = a % 10;
8     a %= 10;
9
10    printf("%d %d" , b , a);
11
12 }
```

Output:

```
Command Prompt
C:\Users\Ti\Desktop>gcc ao3.c
C:\Users\Ti\Desktop>a.exe
2 2
C:\Users\Ti\Desktop>
```

Here, `int b = a % 10;` first divides the value in `a` with `10` and returns the remainder to `b`. $12 / 10$ returns `2` as remainder. so value in `b` will be `2`.

`a %= 10;` is same as `a = a % 10;` This will return `2` as remainder and it will be assigned to `a`.

Increment , Decrement Operators

Look at the following code:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int a = 4;
6     int b = 8;
7
8     a++;
9     b--;
10
11    printf("%d %d" , a , b);
12
13 }
```

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc incdec.c
C:\Users\Ti\Desktop>a.exe
5 7
C:\Users\Ti\Desktop>
```

Here, `a++` is the increment operation and it is same as `a = a + 1` and `a += 1`.
`b--` is the decrement operation and it is same as `b = b - 1` and `b -= 1`.

You can try comparing this with the output.

Bitwise Operators

Bitwise AND and Bitwise OR Operators

Both the AND and OR operators do pure binary operations. Let's first learn Bitwise AND operation.

Take the binary value `11010` and `11011` as two variables. Doing an AND operation on these values looks like this:

```
11010
11011
-----
11010
```

Doing AND operation on `11010` and `11011` gives us `11010` as result.

In an AND operation, The cpu takes each bit in both of the variables and generates a corresponding binary value `1` if both of the bits are `1` and generates `0` if both or any of the bit is `0`.

The Bitwise OR operation looks like this:

```
11010
11011
-----
11011
```

Here, The cpu takes each bit in both of the variables and generates a corresponding binary value `1` if any of the bit is `1` and generates `0` if both of the bits are `0`.

In c, The symbol `&` could be used to perform AND operation and the symbol `|` could be used to perform OR operation.

Lets See an example:

```
1 #include <stdio.h>
2
3 int main(){
4
5     char aa = 3;
6     char ab = 5;
7
8     char AND = aa & ab;
9     char OR  = aa | ab;
10
11    printf("%d %d" , AND , OR);
12
13 }
```

Output



```
Command Prompt
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Ti>cd desktop

C:\Users\Ti\Desktop>gcc aobit.c

C:\Users\Ti\Desktop>a.exe
1 7
C:\Users\Ti\Desktop>
```

Left shift and Right shift Operators

Both `left shift` and `right shift` operators shifts bits in a binary value to the specified side, the specified number of times.

Take binary value `00010` for example. Left shifting it by one will change that value to `00100`, Right shifting the value `00010` it by one will change it to `00001`.

You could shift the bits to any number of position, But as an example here , we will shift by one.

In c, The symbol `<<` could be used as left shift operator and `>>` could be used as right shift operator.

Lets look at an example:

```

1 #include <stdio.h>
2
3 int main(){
4
5     char left = 3;
6     char right = 3;
7
8     left = left << 1;
9     right = right >> 1;
10
11    printf("%d %d" , left , right);
12
13 }
```

Output

```

Command Prompt
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Ti>cd desktop
C:\Users\Ti\Desktop>gcc leftright.c
C:\Users\Ti\Desktop>a.exe
6 1
C:\Users\Ti\Desktop>
```

Macros

Macros or pre-processors are a feature provided by the C compiler to do things at the compile time.

From the start of this book, we read about Data types, branching, looping etc. These things will take effect only when we run the executable.

Macros provide us the ability to do things at the compile time (when we compile the code with commands like : `gcc code.c`).

We will learn about two Macros here which are the `#include` and `#define` directives.

#include

The `#include` directive is used to include other C source files into the program we develop so that we could call function in that file and more.

All of the examples we have covered this far have `#include <stdio.h>` in the very top.

This will include the file named `stdio.h` into our source file. The `<>` brackets tell the compiler that to look for the `stdio.h` file in the C standard library folder.

When developing our operating system, it is advised not to include any of these standard libraries as if any of the functions we call in this files contains platform specific calls(Specifically calls to the operating system), It will result in run time errors.

But we can include our own files into the source files we create. Imagine if you want to create a file named `expl.h` and call a function we made in that file. To do this , create a file with that name and put the following code into it so that we could call it:

```
1 void out(){
2     printf("Hello");
3 }
```

Now create a file named `func.c`(You can name it anything you want) and put the following code:

```
1 #include <stdio.h>
2 #include "expl.h"
3
4 int main(){
5     out();
6 }
```

Now compile and run `func.c` file to get the output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc func.c
C:\Users\Ti\Desktop>a.exe
Hello
C:\Users\Ti\Desktop>
```

You can see that the message got printed. We called a function named `out`, in `expl.h` file to print something to the screen. This way you could include any file you need.

For that we used the `#include` directive with filename in "" at the top of the file.

As what we does is printing to the screen, don't forget to include the `stdio.h` file

Also note that two of this files should be in the same folder to get compiled.

Now, Look at the following case:

The main file `func.c` is in a folder named `src` and you want to include the file `expl.h` which is contained in a sub folder named `header`(which is also inside the `src` folder), To do that, include the following line in the `func.c` file.

```
#include "header/expl.h"
```

You should now go to the `src` folder using command prompt using `cd` command and compile the `func.c` file

Please note that you should put `.h` as extension for the files included by the `#include` directive.

`#define`

The best way to explain `#define` directive is to show an example, Have a look at the following:

```

1 #include <stdio.h>
2
3 #define Message "WE ALMOST COMPLETED OUR C PROGRAMMING TUTORIAL"
4
5 int main(){
6     printf(Message);
7 }
```

Output:



```

C:\Users\Ti\Desktop>gcc define.c
C:\Users\Ti\Desktop>a.exe
WE ALMOST COMPLETED OUR C PROGRAMMING TUTORIAL
C:\Users\Ti\Desktop>
```

Here, what the line:

```
#define Message "WE ALMOST COMPLETED OUR C PROGRAMMING TUTORIAL"
```

Does is only make the compiler to copy paste where ever it see the word 'Message' with the string "WE ALMOST COMPLETED OUR C PROGRAMMING TUTORIAL"

In the `printf` function We passed 'Message' inside `()`. What this does is only replace the string 'Message' with the string "WE ALMOST COMPLETED OUR C PROGRAMMING TUTORIAL" at the compile time.

So it will look like this when compiling(But we can't see this):

```
printf("WE ALMOST COMPLETED OUR C PROGRAMMING TUTORIAL");
```

Hexadecimal Notations

Hexadecimal is just a number system which allows us with ease of work with computers. We Humans generally use decimal as the number system which have a base of 10. Base of 10 means we have a total of 10 number ranging from 0 to 9.

The base 10 of decimal number system is said to be derived from the fact that humans have a total of 10 fingers. But anyway, when we work with computers, the efficient way is to use hexadecimal number system.

This system have a base of 16 which comprises of values from 0 - 9 and a - f

Hexadecimal value of 1 to 9 is same as that of decimal.

For eg: Hex of decimal value 1 is 1,

Hex of decimal value 9 is 9.

But the hex value of 10 is a.

11 = b

12 = c

13 = d

14 = e

15 = f

Now the hex of decimal value 16 is 10

17 = 11

18 = 12

19 = 13

20 = 14

21 = 15

22 = 16

23 = 17

24 = 18

25 = 19

26 = 1a

27 = 1b

28 = 1c

29 = 1d

30 = 1e

31 = 1f

32 = 20

And this process goes on.

Hex of 255 is ff.

255 or ff is the highest value that a byte could have.

A byte means 8 bits. so in the value ff , The first f is took from the first four bits of the byte and the next f is took from the remaining four bits

So, The main advantage of the hexadecimal format allows us to get a single hex digit from four bits.

Each group of 4 bits in binary is a single digit in the hexadecimal system. This makes it really easy to convert binary to hexadecimal numbers.

We could use hexadecimal number system in c by prefixing `0x` on a hex value, for eg: if we define it as `0xa`, this will get converted to the decimal value 10.

`0xff` will be converted to decimal value 255.

Two hexadecimal characters make one byte, Four of them make 2 bytes, six of them make 3 bytes. eg: `0xff` is one byte, `0xffaa` is two bytes and `0xffaa11` is three bytes.

Have a look at the following code:

```
1 #include <stdio.h>
2
3 int main(){
4
5     int a = 0x4;
6     int b = 0xa;
7     int c = 0xff;
8     int d = 0xff12;
9
10    printf("%d %d %d %d" , a , b , c , d);
11 }
```

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc hex.c
C:\Users\Ti\Desktop>a.exe
4 10 255 65298
C:\Users\Ti\Desktop>
```

You could now do further studies relating to the hexadecimal number system if you want.

Comments

Commenting is a feature provided by almost every language which let us include documentation or other helpful information along with the code.

Comments would be skipped at compile time.

We have two ways to comment inside our code in c. We could comment on a single line with `//` and comment on multiple lines with `/*` and `*/`, Have a look at the following:

```
1 #include <stdio.h>
2
3 int main(){
4
5     /*
6         This is an example of multi line comment.
7         This allows us include multiple lines.
8         The main aim of comments is to document working of the code
9     */
10
11     printf("Hello"); // This is an example of single line comment
12
13 }
```

When compiled and executed, We won't have any errors, and the code would work fine.

Output:



```
Command Prompt
C:\Users\Ti\Desktop>gcc comment.c
C:\Users\Ti\Desktop>a.exe
Hello
C:\Users\Ti\Desktop>
```

Let's Have A Game

This section will help you to put what you have learned this far and implement it.

This is the challenge for you:

You have a variable named `i` , You can assign it with any value , but first assign it with 5 and try to print one star on the first line , two stars in the second line and , three stars on the third line and continue until the number of lines reaches the value of `i`.

If you assign `i` with 5, This will be the output:



```
C:\Users\Ti\Desktop>gcc c-Game.c
C:\Users\Ti\Desktop>a.exe
*
**
***
****
*****
C:\Users\Ti\Desktop>
```

Now, Try to find a solution to this problem.

Solution

```
1 #include <stdio.h>
2
3 int main(){
4
5     int i = 5;
6
7     int y = 1;
8     int z;
9
10    while(y <= i){
11        z = 0;
12        while(z < y){
13            printf("*");
14            z++;
15        }
16        printf("\n");
17        y++;
18    }
19 }
```

Programming in Assembly Language

Introduction

Every computers have a common way of working which follows: Input → Processing → Output. Processing is the powerfull and resource intensive area.

The Hardware which processes instruction in normal Desktop or Laptop computers is commonly called as a Micro Processor. Micro processors does three thing in common which are : Fetch → Decode → Execute.

The processor first takes an instruction from the memory, Then it will decodes the instruction which makes the internal circuitry ready for the execution of that single instruction and finally, it will do the operation specified by the fetched instruction.

I am not willing to make you mad by explaining theories, But i will explain only the necessary things which you can't skip.

What is an Assembly Language

Keeping it simple, Assembly language is a low level language which helps us teach computers do a task by explaining it in a step by step manner.

An assembler is a software which converts the program's we wrote in assembly language to pure binary which the processor could directly execute.

We have special instructions in assembly language to do the task we need. The addition operation will look something like this in assembly:

```
add x , 5
```

This instruction will simply add 5 to x. Copying values will look something like this:

```
mov numb , x
```

This will simply copy the value in x to numb.

Now, Lets see what the main role of an assembler is:

The processor won't be able to do things that we write in english or other human readable language.

Take the addition instruction

```
add x , 5
```

as an example: The assembler will convert this human readable code to binary. For this example, Please take the following hex values into consideration(Please don't take it as exact , this is only used to convey the idea):

```
0x83 0x45ff 0x5
```

The assembler will convert the add instruction to the hex value `0x83`, Then to represent the address of `x`, it converts it to the hex value `0x45ff`(Please note that this is not the exact value everytime, it will depend on the address of variable in focus). Now, to denote the addition of value `5`, it will be converted to `0x5`

During the execution, The processor will first take the byte `0x83`.

It is predefined in the processor circuitry so that when it see the hex value `0x83`, to prepare the processor to fetch some of the next binary values to perform the addition operation.

Now, The processor fetches these values `0x45ff 0x5`

Finally, The processor adds `0x5`(`5` in decimal) to the value located at address `0x45ff`.

Likewise, the processor have a table of binary values like these which represent each operation.

These values are called machine code.

There are different micro processor architectures in the market. Each processor architecture will have its own unique table of values(machine code). This means that programs written for a special processor architecture won't work in another processor architecture.

The Processor architecture we are going to work with is intel's x86 architecture.

This is the mostly used architecture currently and it is where Windows and Linux Mainly run on. Even though x86 is developed by intel, Companies like AMD also manufactures the same.

So at the conclusion, Assembly language programs are a processor architecture specific, low-level, Human readable routine which would be translated to a non-human-readable(binary) form also known as machine language, by a software commonly named as assembler.

What is a Compiler actually?

The main purpose of a compiler is not to convert the source code to binary.

We have discussed about the `add` and `mov` commands in x86 assembly language. There are many instructions that we can use to make our program in assembly.

To know what a compiler is, Please save the following c code to a file, name it `code.c` :

```

1 #include <stdio.h>
2
3 int main(){
4     printf("Hello , World");
5 }
```

Now, Go to where you saved the file with command prompt and type the following command

```
gcc -S code.c
```

You will now get a new file in the same directory. In my system, I got a file named code.s

Have a look at its content:

```

1      .file      "code.c"
2      .def      __main;      .scl      2;      .type      32;      .edef
3      .section .rdata,"dr"
4 LC0:
5      .ascii "Hello , World\0"
6      .text
7      .globl      _main
8      .def      _main;      .scl      2;      .type      32;      .edef
9 _main:
10 LFB10:
11      .cfi_startproc
12      pushl      %ebp
13      .cfi_def_cfa_offset 8
14      .cfi_offset 5, -8
15      movl      %esp, %ebp
16      .cfi_def_cfa_register 5
17      andl      $-16, %esp
18      subl      $16, %esp
19      call      __main
20      movl      $LC0, (%esp)
21      call      _printf
22      movl      $0, %eax
23      leave
24      .cfi_restore 5
25      .cfi_def_cfa 4, 4
26      ret
27      .cfi_endproc
28 LFE10:
29      .ident      "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"
30      .def      _printf;      .scl      2;      .type      32;      .edef
```

This is an assembly code equivalent to the c code we wrote. So this is the point, a compiler is only a program which translates the code we write in a higher level language like c, to a lower level language like assembly. Then the c compiler will call its own assembler to assemble the assembly code which the compiler generated.

You don't need to worry about the complexity of assembly code you saw here. We will not write assembly programs like this. We will only use a neat style to learn this. And mainly we are not going to use the assembler which comes with the compiler. This is why we have downloaded nasm as our assembler.

You just saw the amount of lines created just for our hello world program. This is why no one these days write code in assembly and it is also the reason behind the development of languages like c.

Assembly language is processor specific, and if we write code in assembly for a specific processor, we wont be able to make it work on other processor architectures. If you want to do that, You may need to write that program in assembly language of the processor you want to port to from scratch. This is where a compiler come. If we write our os in c or similar language, we will be able to port the program to other processor architectures if the processor vendor provide a c compiler for their processor. But we wont be able to port assembly programs like this.

This is why we write our os in c, But we surely need to write some part of the code in assembly. When porting your os to other processor architecture, You may need to change every assembly code you wrote for your os to the equivalent code for the processor architecture you want to port to.

x86 Processor data sizes

The x86 processor defines different data sizes with its name and its size. Here is the list of it:

Word: a 2-byte data item

Doubleword: a 4-byte (32 bit) data item

Quadword: an 8-byte (64 bit) data item

Paragraph: a 16-byte (128 bit) area

Kilobyte: 1024 bytes

Megabyte: 1,048,576 bytes

Assembly Hello , World

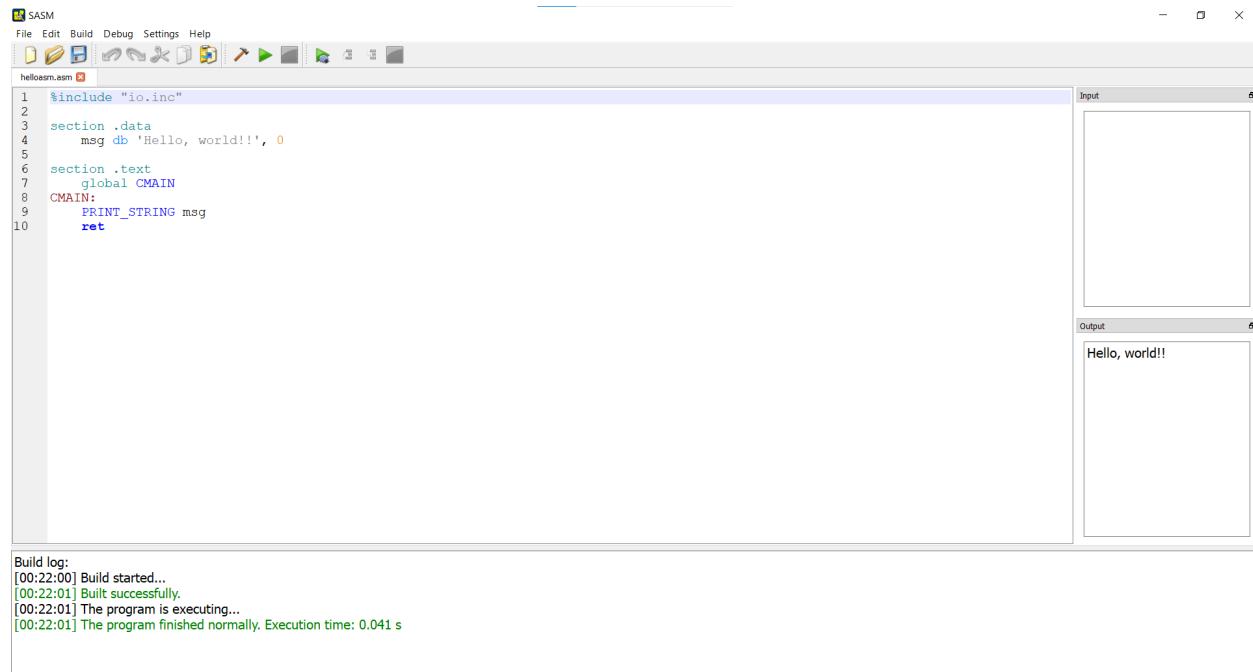
We are now going to use the software named sasm that we have downloaded. First open it and create a new project and include the following code into it:

```

1 %include "io.inc"
2
3 section .data
4     msg db 'Hello, world!!', 0
5
6 section .text
7     global CMAIN
8 CMAIN:
9     PRINT_STRING msg
10    ret

```

Run it by pressing the play button and we will get the following output:



We will explain the working later. For now, We could jump to the next section.

Registers

Now, we will learn some theories.

The main function of a processor is to process data. These data's are mainly stored in main memory(also known as RAM or Random Access Memory). As the job of a processor is to process data, It may need to access memory often.

Here, We have to think about a factor, which is speed. The processor is very much faster than RAM and accessing it frequently will decrease it's processing speed to a greater extend. It is also an inefficient way as accessing it frequently will put a lot of traffic on the system bus.

This is where the concept of register's come. Registers are the memory inside the processor. The technology used to implement registers are so advanced so that the processor could execute at a maximum speed.

But we can't use the technology used to implement registers to build a main memory. If this is practical, We could think of an architecture with no registers, With only a main memory. But this idea will only be in theories. The registers in the processor these days are either 32 bit or 64 bit in size. But when talking about main memory, We will atleast need 2 or 4 GB.

If we use the technology used to implement registers to build main memory, It will cost you as much as money which only a few could handle. This is the reason why we implement computers with RAM as a slow memory -> cache as a faster memory than ram -> And finally registers as an even faster memory than cache.

The following is the list of devices in computer with increasing amount of speed and decreasing amount of capacity:

- 1)Hard disks – Permanent storage
- 2)Ram – Data goes off when the computer is off
- 3)Cache – Data goes off when the computer is off
- 4)Registers – Data goes off when the computer is off

The registers are grouped into three categories:

- 1)General registers
- 2)Control registers
- 3)Segment registers

The general registers are further divided into the following groups:

- 1)Data registers
- 2)Pointer registers
- 3)Index registers

General Registers

Data Registers

There are four data registers used for arithmetic and other operations. These are the 64 bit data registers:

- 1)RAX
- 2)RBX
- 3)RCX
- 4>RDX

Lower halves of the 64 bit data registers can be used as 32 bit data registers which are:

- 1)EAX
- 2)EBX

- 3)ECX
- 4)EDX

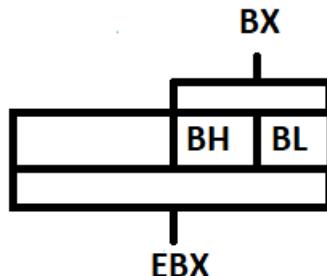
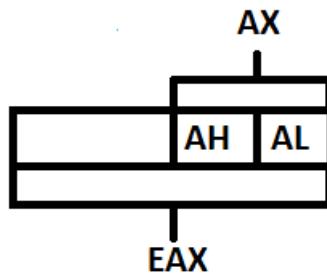
Lower halves of the 32 bit data registers can be used as 16 bit data registers which are:

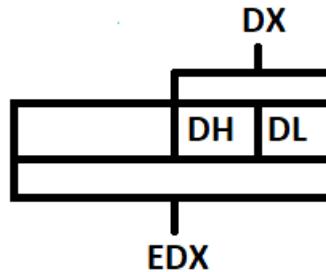
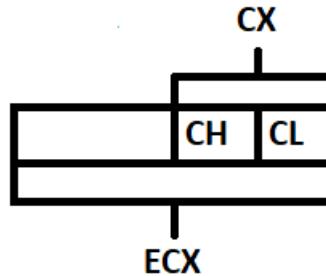
- 1)AX
- 2)BX
- 3)CX
- 4)DX

Higher and Lower halves of 16 bit data registers can be used as 8 bit data registers which are:

- 1)AH , AL
- 2)BH , BL
- 3)CH , CL
- 4)DH , DL

Have a look at the following images which shows division of 32 bit data registers:





The AX is the primary accumulator: It is used in input/output and most arithmetic operations.

The BX is known as base register: It could be used in indexed addressing(We will see this later).

The CX is known as the count register: It is mainly used to store loop count.

The DX is known as data register: It is also used in input/output operations. It is also used along with the AX register for arithmetic operations.

Pointer Registers

There are three pointer registers. 64 bit ones are named:

- 1)RIP
- 2)RSP
- 3)RBP

32 bit ones are named:

- 1)EIP
- 2)ESP
- 3)EBP

16 bit ones are named:

- 1)IP
- 2)SP
- 3)BP

Instruction Pointer (IP) - This register stores the address of next instruction to be executed. x86 architecture use segmented addressing. IP is associated with the CS register. CS : IP gives the complete address of the next instruction to be executed(We will see this later).

Stack Pointer (SP) - Mainly used to point to the top of stack. SS : SP Gives the complete address in segmented addressing(We will see this later).

Base Pointer (BP) - This is mainly used to access memory relatively. BP - A special offset gives access to variables in memory. SS : BP gives the complete address in segmented addressing(We will see this later).

Index Registers

There are 2 index registers. 64 bit ones are named:

- 1)RSI
- 2)RDI

32 bit ones are named:

- 1)ESI
- 2)EDI

16 bit ones are named:

- 1)SI
- 2)DI

Source Index (SI) - This is mainly used as source index for string operations.

Destination Index (DI) - This is mainly used as destination index for string operations.

Control Registers

Many instructions in x86 architecture involves comparisions and mathematical operations. This will change the status of some flags and some other conditional instructions test the values of these flags to influence the control flow.

The common flag bits include:

- 1)Overflow Flag (OF)
- 2)Direction Flag (DF)
- 3)Interrupt Flag (IF)
- 4)Trap Flag (TF)
- 5)Sign Flag (SF)
- 5)Zero Flag (ZF)
- 6)Auxiliary Carry Flag (AF)
- 7)Parity Flag (PF)
- 8)Carry Flag (CF)

I am not going to explain and overwhelm you with lot of theories about this. If you need more information about this, You could google about x86 control registers.

Not learning this deeply wont affect you for starting in this field. But when you are planning to go advanced, It's a good idea to learn about this.

Segment Registers

Segments are specific areas in a program for containing Data , Code and Stack. x86 initially was a 16 bit architecture. So we could only access data in memory with 16 bit address. This will limit the size of ram that could be implemented. Segmented memory is used to allow us access more memory. Segment registers are the solution used to solve this problem.

The Registers include:

- 1)**Code Segment (CS):** Used to point to executable code. This register stores the starting address of the code segment.
- 2)**Data Segment (DS):** Used to point to Data. This stores the starting address of Data Segment.
- 3)**Stack Segment (SS):** Used to point to the starting of the stack segment. Stack segment is where variables which are local to functions are stored.

x86 Processor Endianess

Processor Endianess defines how a multi-byte data type is stored in memory.

The early computers implemented a 4 bit or 8 bit processor architecture.

x86 architecture was a 16 bit architecture and data items in it's registers qualify to be named as multi-byte data type.

The two types of endianess which exists are named as Little Endian and Big Endian. x86 uses Little Endian architecture. Little Endian architecture stores multi byte data in a reverse order in memory.

By reversing, it doesn't mean the first bit is stored last and last bit is stored first. The reversing get affected at the byte level which means that first byte of a multi-byte data item get stored last, Second byte get stored second last and goes until the last byte gets stored first.

When copying the data from memory to any register, another byte reversing will be done which practically makes the number in the real form.

For Eg: Consider 0xaabb as a multi-byte data. In memory, It will be stored as 0xbbaa. After copying it to any register, It will be back in normal form as 0xaabb.

Consider the following 32 bit data :

0xaabbccdd

This will be in memory as 0xddccbbaa.

It will be in normal form like this:

0xaabbccdd

When in register.

When talking about the C Programming language in x86 architecture, The keyword `char` won't be affected by Endianess as `char` is a single byte data type, But the `int` keyword will always be affected by Endianess as `int` keyword always uses 4 - 8 bytes.

The other architecture which exists is the Big Endian architecture. This architecture uses the normal way of storing which stores the first byte first and continues until the last byte get stored last.

Commands For Register Operations

Most of the operations we do involves registers. So, We have different commands to work with registers. We will discuss about the important ones.

mov Command

The `mov` command is used to copy values from one location to other. Following is the basic syntax:

```
mov destination , source
```

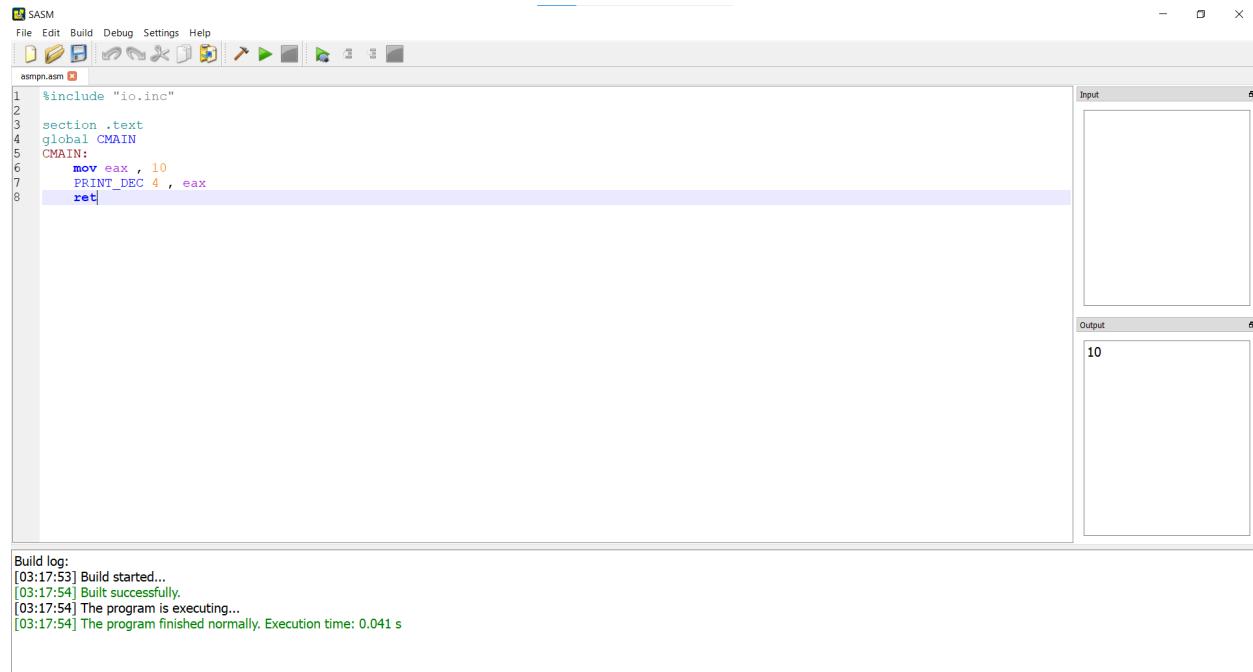
Consider a situation where you want to copy the value 10 to the `eax` register. The code will look like this:

```
mov eax , 10
```

Here's how we could print it in sasm.

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6     mov eax , 10
7     PRINT_DEC 4 , eax
8     ret
```

Output



The screenshot shows the SASM (Savannah Assembler) interface. The assembly code in the editor window is:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6     mov eax, 10
7     PRINT_DEC 4, eax
8     ret
```

The 'Input' window is empty. The 'Output' window displays the value '10'.

Build log:

- [03:17:53] Build started...
- [03:17:54] Built successfully.
- [03:17:54] The program is executing...
- [03:17:54] The program finished normally. Execution time: 0.041 s

You only need to look at the code after CMAIN:

The command `mov eax, 10` only copies the value 10 to the eax register.

In `PRINT_DEC 4, eax`, `PRINT_DEC` is not a command. It is only a routine defined in the `io.inc` file included at the top of the code. Here, It helps us convert binary value in eax register to decimal form and print it as an output.

`4` in `PRINT_DEC 4, eax` tells the routine that the operation is done in a four byte data item (Here it is the eax register).

You do not need to focus on this printing function as we wont have this routine when developing our os. We will build our own routine for that.

The very last command `ret` will practically terminate that program. The function of `ret` command is not to terminate a process. We will discuss that later.

We could do this operation in more register, Look at the following:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 10
8     PRINT_DEC 4, eax
9     NEWLINE
10
11    mov ebx, 20
12    PRINT_DEC 4, ebx
13    NEWLINE
14
15    mov ecx, 30
16    PRINT_DEC 4, ecx
17    NEWLINE
18
19    mov edx, 40
20    PRINT_DEC 4, edx
21    NEWLINE
22
23    ret
```

Output

```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 10
8     PRINT_DEC 4, eax
9     NEWLINE
10
11    mov ebx, 20
12    PRINT_DEC 4, ebx
13    NEWLINE
14
15    mov ecx, 30
16    PRINT_DEC 4, ecx
17    NEWLINE
18
19    mov edx, 40
20    PRINT_DEC 4, edx
21    NEWLINE
22
23    ret

```

Build log:

- [03:52:59] Build started...
- [03:52:59] Built successfully.
- [03:52:59] The program is executing...
- [03:52:59] The program finished normally. Execution time: 0.042 s

Here, We tested the `mov` command with different registers and it worked!

The line `NEWLINE` will put a line break in the output. It is not a command in x86 architecture. `NEWLINE` is pre defined in the `io.inc` file we included at the top.

add Command

As the name say's, add command is a command to perform the addition operation.

The syntax is as follows:

`add destination , source`

Here's the implementation:

```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 10
8     PRINT_DEC 4, eax
9     NEWLINE
10
11    add eax, 10
12    PRINT_DEC 4, eax

```

```

13     NEWLINE
14
15     ret

```

Output

```

1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax, 10
8      PRINT_DEC 4, eax
9      NEWLINE
10
11     add eax, 10
12     PRINT_DEC 4, eax
13     NEWLINE
14
15     ret

```

Input

Output

Build log:

[22:09:12] Build started...

[22:09:12] Built successfully.

[22:09:12] The program is executing...

[22:09:12] The program finished normally. Execution time: 0.047 s

Here, we first copied the value 10 to the eax register and printed it. Then we issued the command `add eax , 10`. This command adds 10 to the value in eax register and stores the result in the eax register itself.

$10 + 10 = 20$, so the next string printed to screen is 20.

sub Command

As you have guessed, the `sub` command is used to perform the subtraction operation.

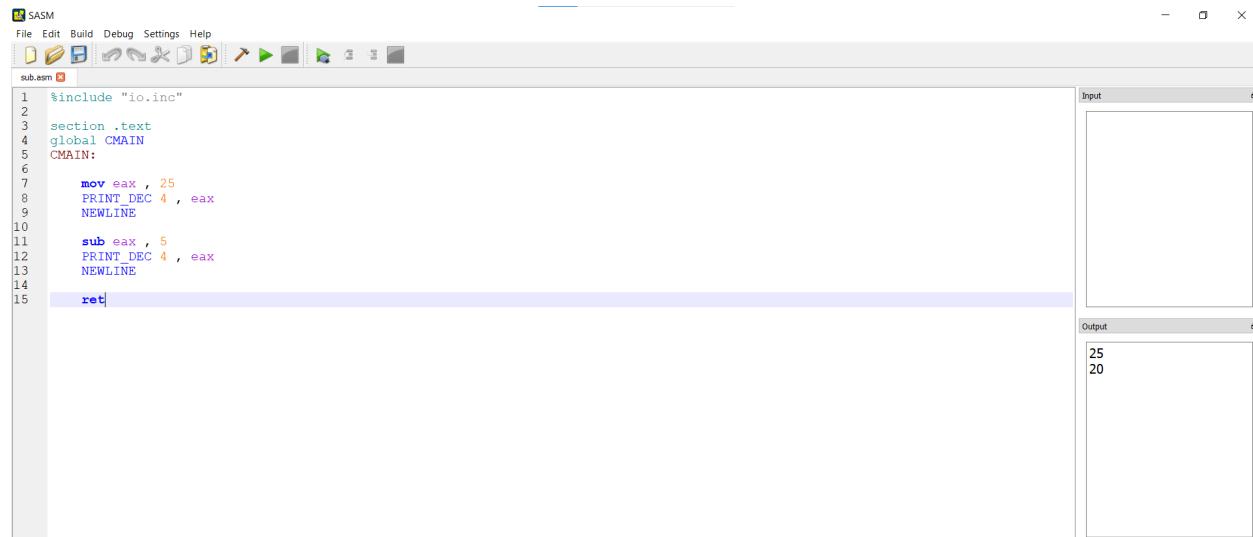
The syntax is as follows:

`sub destination , source`

Here's the implementation:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 25
8     PRINT_DEC 4, eax
9     NEWLINE
10
11    sub eax, 5
12    PRINT_DEC 4, eax
13    NEWLINE
14
15    ret
```

Output



The screenshot shows the SASM debugger interface. The assembly code pane displays the provided assembly code. The input pane is empty. The output pane shows the results of the program execution: '25' on the first line and '20' on the second line. The build log at the bottom shows the following messages:

```
Build log:
[22:20:26] Build started...
[22:20:27] Built successfully.
[22:20:27] The program is executing...
[22:20:27] The program finished normally. Execution time: 0.031 s
```

The `sub` instruction here will subtract 5 from 25 and the value 20 is printed.

push And pop Commands

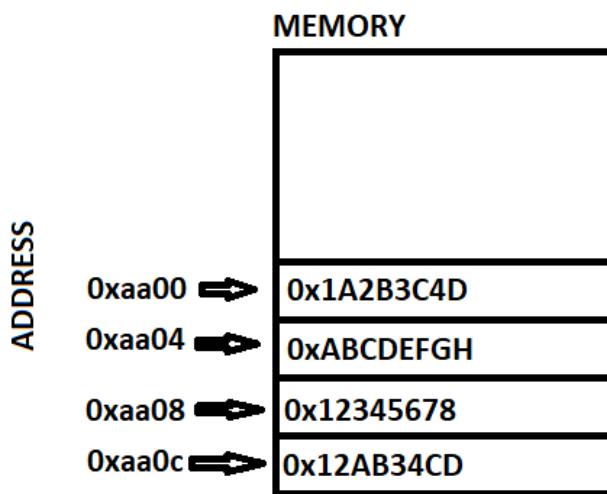
Working With Stack

Stack is an area in the main memory(RAM) where we could store variables and other data items. The two operations that we could do on stack memory is the push and pop operations. Push basically means to store a value in memory and Pop means to take a value from memory.

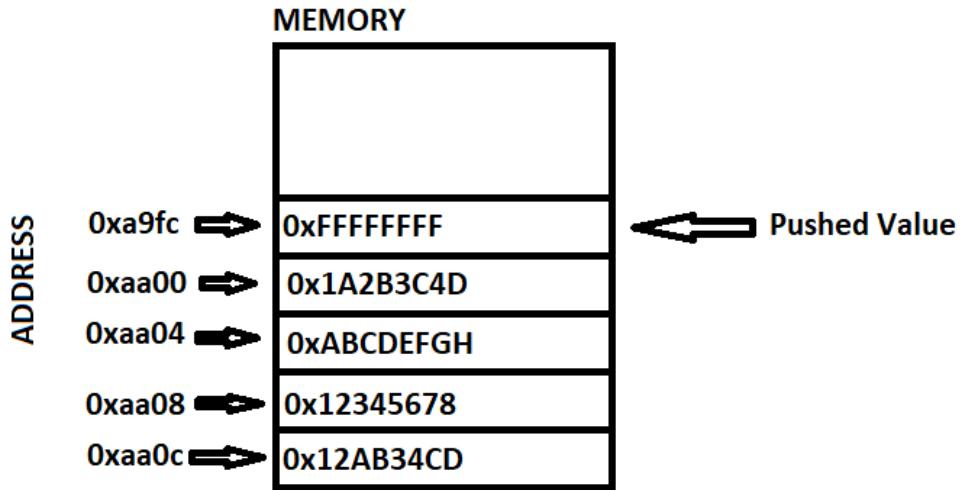
On the x86 architecture, pushing a value to stack means to put a value in lower memory address. When we attempt another push operation, The processor will put the value specified to it in memory just BEFORE the previous one. This means that the stack grows to lower memory addresses.

Popping values(Or taking value) from memory means taking value from the lowest address of the stack. Another pop attempt will take the value just AFTER the previous one. This means that pop instruction goes to higher memory addresses.

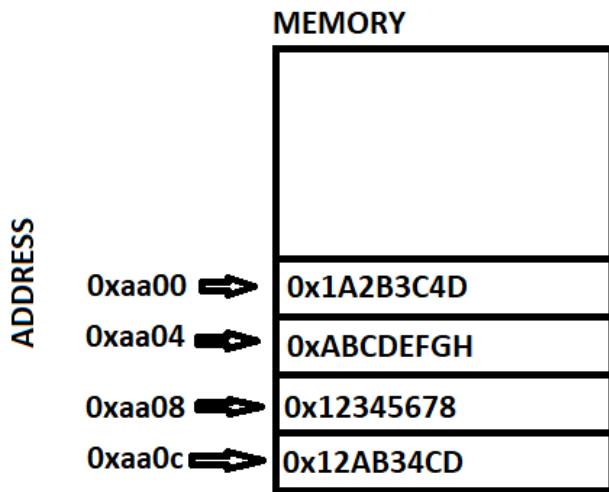
We will see an example image, Here's an image of the stack with some initial values in it:



Here is the image of the stack memory when a value 0xFFFFFFFF is pushed onto it:



At this stage, If we issue a pop command, The stack memory will look like this:



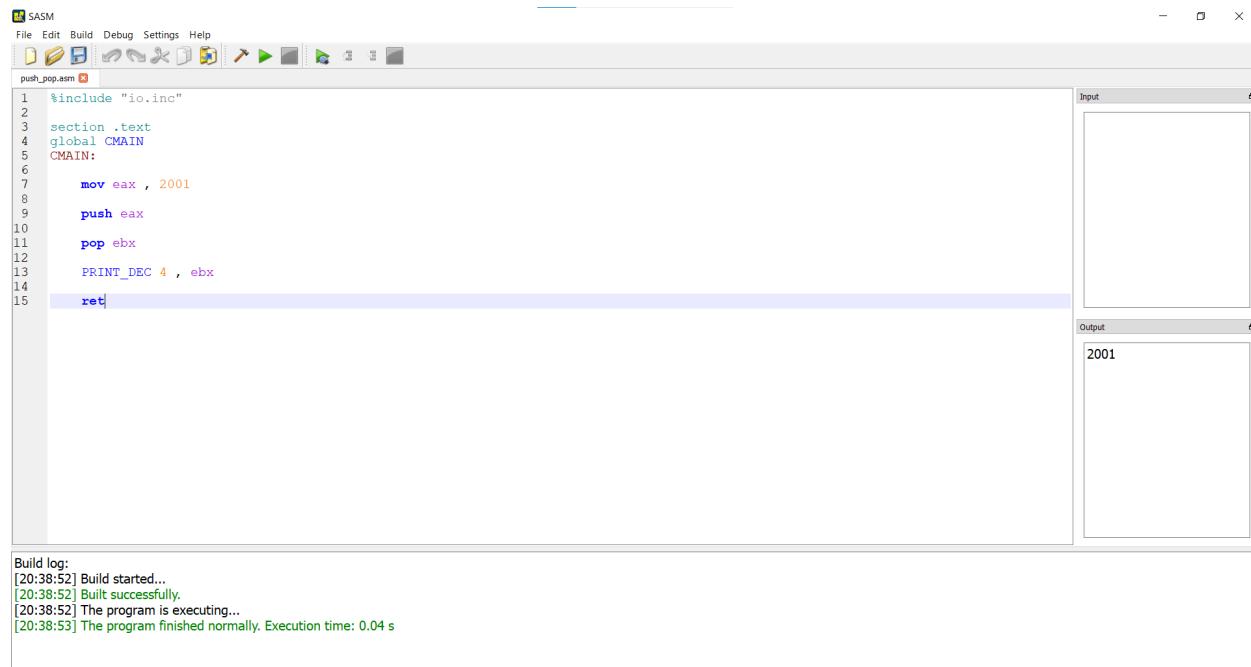
Practical Implementation Of Stack

```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax , 2001
8
9     push eax
10
11    pop ebx
12
13    PRINT_DEC 4 , ebx
14
15    ret

```

Output



```

SASM
File Edit Build Debug Settings Help
push_pop.asm
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax , 2001
8
9     push eax
10
11    pop ebx
12
13    PRINT_DEC 4 , ebx
14
15    ret

```

Build log:
[20:38:52] Build started...
[20:38:52] Built successfully.
[20:38:52] The program is executing...
[20:38:53] The program finished normally. Execution time: 0.04 s

Here , We first initialized eax with 2001. Then we pushed the value in eax to the stack with the command push eax. This will place the value 2001 to the stack memory.

Then we issued the following command, pop ebx. This will take the value that we last pushed to the stack to the ebx register. So now, the ebx register have the value 2001.

Then we printed the value in ebx. This will print the value 2001 to the screen.

This is what the push and pop command basically does.

But how does the processor know what the last item pushed to the stack is?

This is where the esp register comes. The esp register holds address to the last item in stack memory.

For eg: If we push a value in eax register to the stack, What the processor does is first decrement the value in esp by four bytes(Which is the size of the eax register), Then it pushes the value in eax register to the address pointed by the esp register.

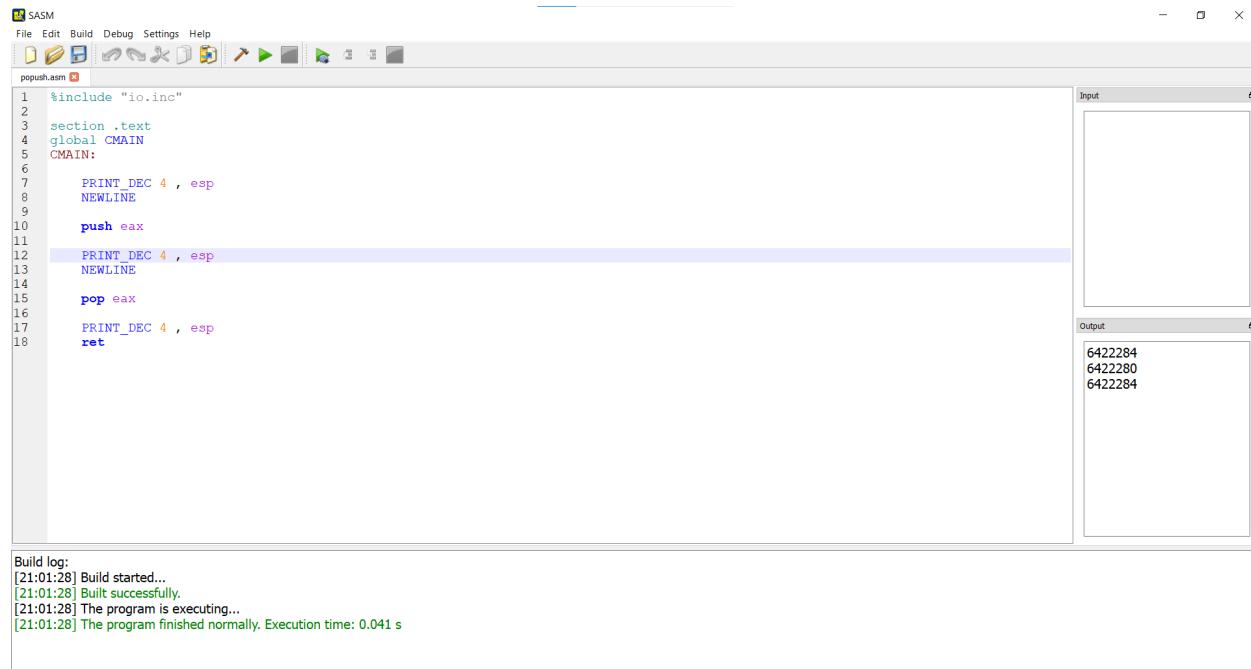
Likewise, when we pop a value to a register for eg: to the ebx register, The processor first takes the value in memory pointed by the esp register and copy it to register ebx and increases the value in esp by four bytes(Which is the size of ebx register).

So at conclusion, The push command decreases the value in esp register and the pop command increases the value in esp register.

Lets see it practically:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     PRINT_DEC 4 , esp
8     NEWLINE
9
10    push eax
11
12    PRINT_DEC 4 , esp
13    NEWLINE
14
15    pop eax
16
17    PRINT_DEC 4 , esp
18    ret
```

Output



```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     PRINT_DEC 4 , esp
8     NEWLINE
9
10    push eax
11
12    PRINT_DEC 4 , esp
13    NEWLINE
14
15    pop eax
16
17    PRINT_DEC 4 , esp
18    ret

```

Build log:

- [21:01:28] Build started...
- [21:01:28] Built successfully.
- [21:01:28] The program is executing...
- [21:01:28] The program finished normally. Execution time: 0.041 s

Here, We First printed the initial value of the esp register and we got the value 6422284. This is the address of the data which is at the top of stack.

Then we pushed the value in eax register to the stack with the `push eax` command.

This will practically decrease the value in esp register by four bytes. To confirm this, We printed the value in esp register after that push instruction.

And we got this value : 6422280.

We can clearly see that the value decreased by four bytes. This value is the address of data at top of the stack.

Finally we issued the pop command like this: `pop eax`. This will copy the value at top of the stack to the eax register and also increases the value in esp register by four bytes and we printed it. The value of esp register now is 6422284. This is how the stack and commands to work with stack works.

Let's look at another example:

```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax , 100
8     mov ebx , 200
9
10    push eax

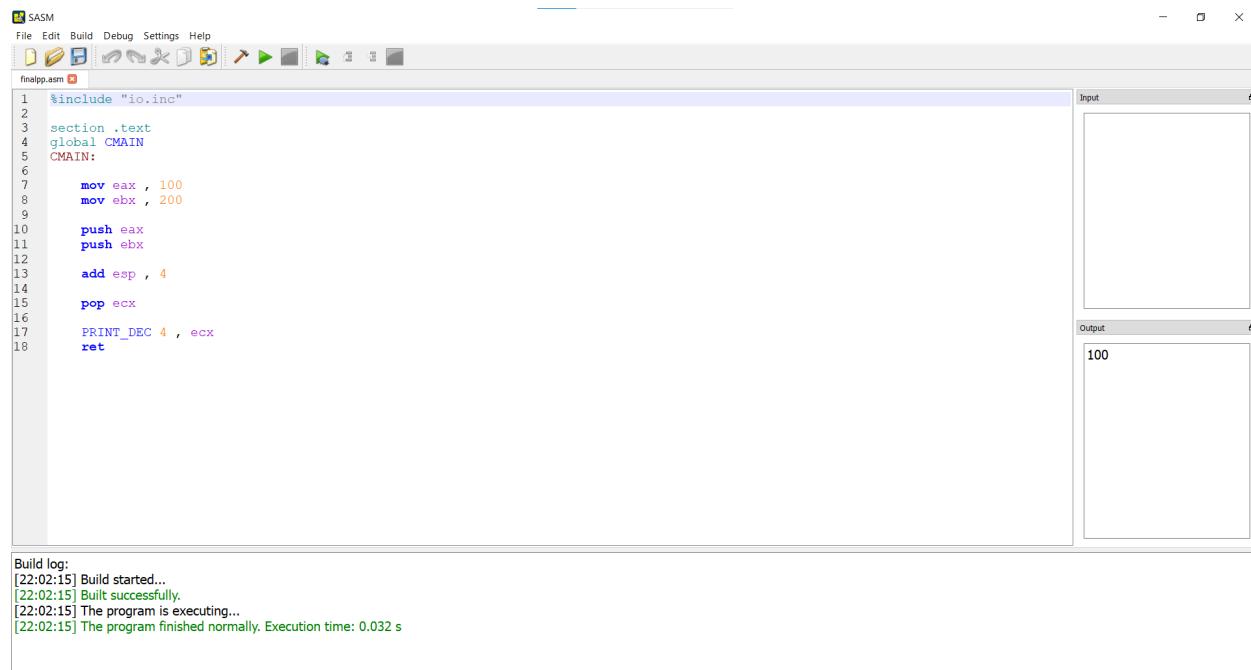
```

```

11      push ebx
12
13      add esp , 4
14
15      pop ecx
16
17      PRINT_DEC 4 , ecx
18      ret

```

Output



```

1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax , 100
8      mov ebx , 200
9
10     push eax
11     push ebx
12
13     add esp , 4
14
15     pop ecx
16
17     PRINT_DEC 4 , ecx
18     ret

```

Build log:
[22:02:15] Build started...
[22:02:15] Built successfully.
[22:02:15] The program is executing...
[22:02:15] The program finished normally. Execution time: 0.032 s

Here, We first moved the number 100 to the eax register and then moved the value 200 to the ebx register.

Then we pushed the value in eax and after that we pushed the value in ebx register to the stack.

At this stage, The value at the top of the stack is 200(Value of ebx register). And if we issue a pop command and print the value, We will get 200 as output.

But before popping and printing the value, We Increased the value in esp register by four bytes with the command : add esp , 4. Now, As you have guessed, The esp register points to the value 100 which we first pushed.

And if we pop the value at top of stack with the pop command and print it, We will get 100 as output, not 200.

You could try removing the add esp , 4 line and differentiate the results.

I showed this example so that you could understand how all of the things incorporate.

pushAll And popAll Commands

What if you want to store values in all register to stack temporarily so that you could load new values to those register and later assign the previous values from stack back to registers?

We have some special commands to do that in x86 assembly.

To temporarily store the values in 32 bit general purpose registers to the stack, You could use the command `pushad` and to retrieve the value back to registers, You could use the `popad` command.

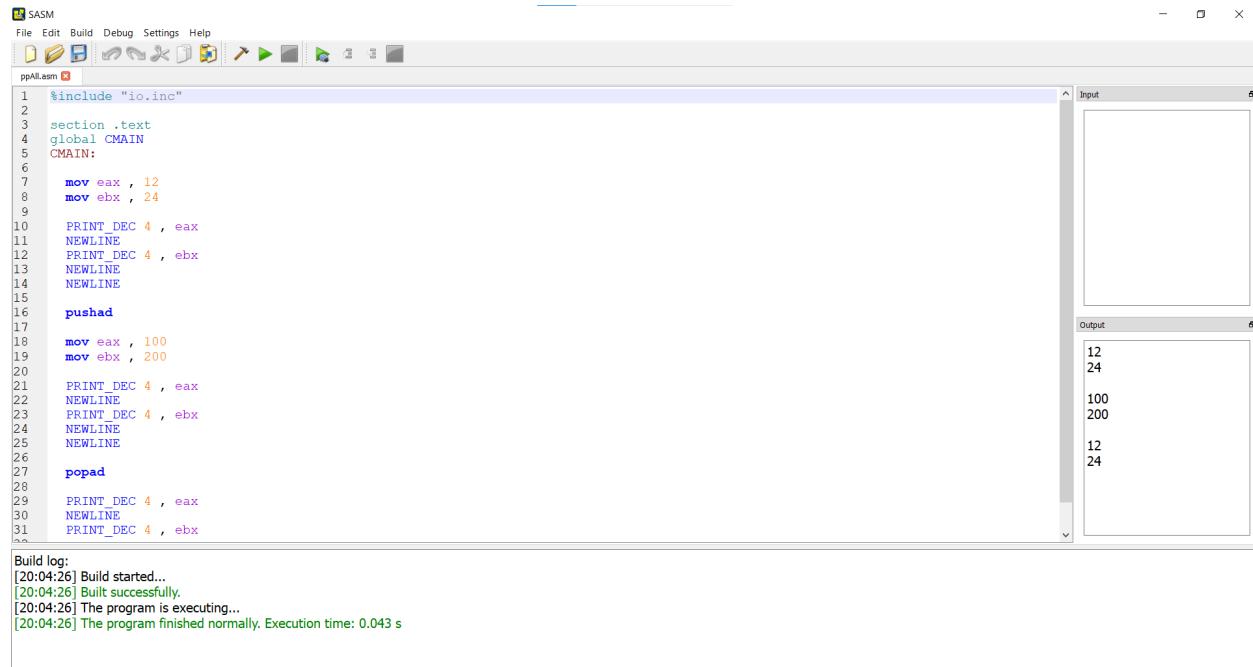
When working on 16 bit register, You could use `pusha` and `popa` commands for these operations.

Have a look at an example:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 12
8     mov ebx, 24
9
10    PRINT_DEC 4, eax
11    NEWLINE
12    PRINT_DEC 4, ebx
13    NEWLINE
14    NEWLINE
15
16    pushad
17
18    mov eax, 100
19    mov ebx, 200
20
21    PRINT_DEC 4, eax
22    NEWLINE
23    PRINT_DEC 4, ebx
24    NEWLINE
25    NEWLINE
26
27    popad
28
29    PRINT_DEC 4, eax
30    NEWLINE
31    PRINT_DEC 4, ebx
```

```
32
33     ret
```

Output



The screenshot shows the SASM (Savannah Assembler) interface. The assembly code in the editor pane is as follows:

```

1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax, 12
8      mov ebx, 24
9
10     PRINT_DEC 4, eax
11     NEWLINE
12     PRINT_DEC 4, ebx
13     NEWLINE
14     NEWLINE
15
16     pushad
17
18     mov eax, 100
19     mov ebx, 200
20
21     PRINT_DEC 4, eax
22     NEWLINE
23     PRINT_DEC 4, ebx
24     NEWLINE
25     NEWLINE
26
27     popad
28
29     PRINT_DEC 4, eax
30     NEWLINE
31     PRINT_DEC 4, ebx
32

```

The 'Input' pane is empty. The 'Output' pane shows the following text:

```

12
24
100
200
12
24

```

Build log:

```

[20:04:26] Build started...
[20:04:26] Built successfully.
[20:04:26] The program is executing...
[20:04:26] The program finished normally. Execution time: 0.043 s

```

I believe you could describe it yourself. But i will say how it works:

Here, We first moved some values to both the `eax` and `ebx` register and then we printed it. Now comes our command in focus: We put the `pushad` command there, This will push the values in all general purpose registers to the stack. Practically the values in `eax` and `ebx` registers also will be pushed.

Then we moved the value 100 to the `eax` register and 200 to `ebx` register and printed it. Here, the previous values in `eax` and `ebx` register got overwritten by new values.

Then we issued the `popad` command. This command will copy the previous values we pushed to the stack with `pushad` command, Back to registers. So the values we copied to `eax` and `ebx` registers before the `pushad` command also will be back to those registers.

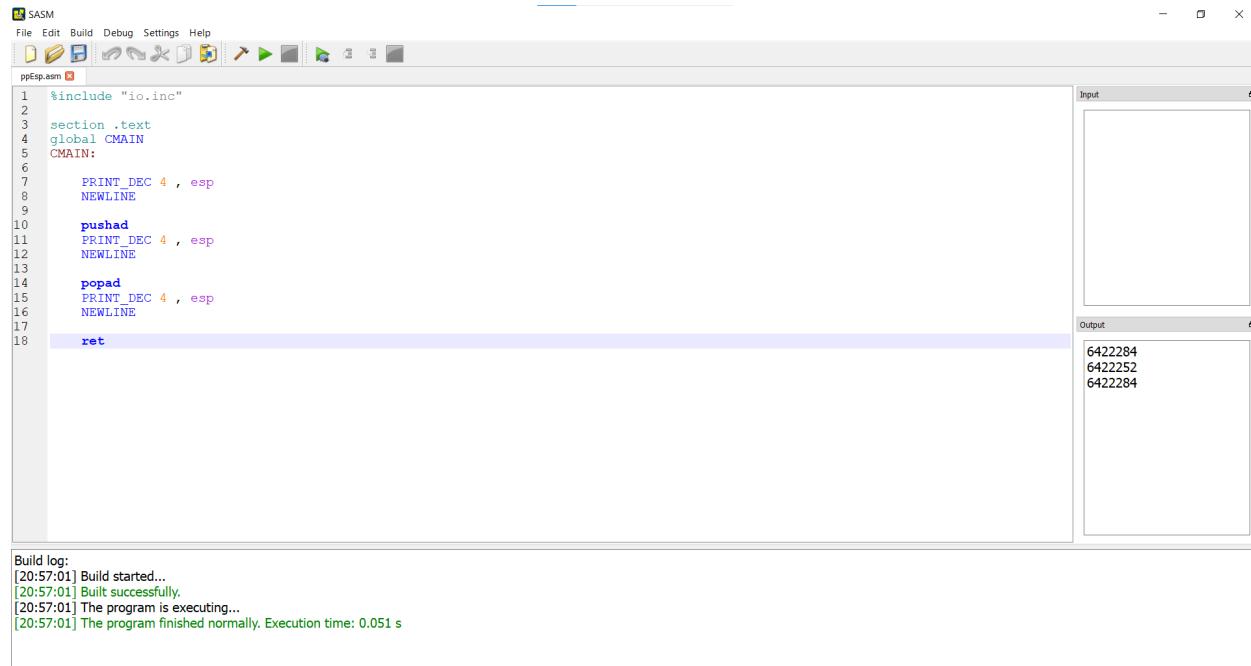
See the following code:

```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     PRINT_DEC 4 , esp
8     NEWLINE
9
10    pushad
11    PRINT_DEC 4 , esp
12    NEWLINE
13
14    popad
15    PRINT_DEC 4 , esp
16    NEWLINE
17
18    ret

```

Output



The screenshot shows the SASM (Savannah Assembler) interface. The assembly code is displayed in the main window, and the output window shows the results of the program execution.

```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     PRINT_DEC 4 , esp
8     NEWLINE
9
10    pushad
11    PRINT_DEC 4 , esp
12    NEWLINE
13
14    popad
15    PRINT_DEC 4 , esp
16    NEWLINE
17
18    ret

```

The output window displays the following text:

```

Input
Output
6422284
6422252
6422284

```

Build log:

```

[20:57:01] Build started...
[20:57:01] Built successfully.
[20:57:01] The program is executing...
[20:57:01] The program finished normally. Execution time: 0.051 s

```

Here, We first printed the initial value of esp register. Then we issued the pushad command. This will push whatever data is present in the general registers to the stack. We confirmed it by printing the value of esp after the pushad operation, We saw that the value of esp decreased from that of the initial value.

Then, We issued the `popad` command, This will pop the values pushed to the stack back to registers, We confirmed it by printing the value in `esp` and we saw that the value came back to the initial value.

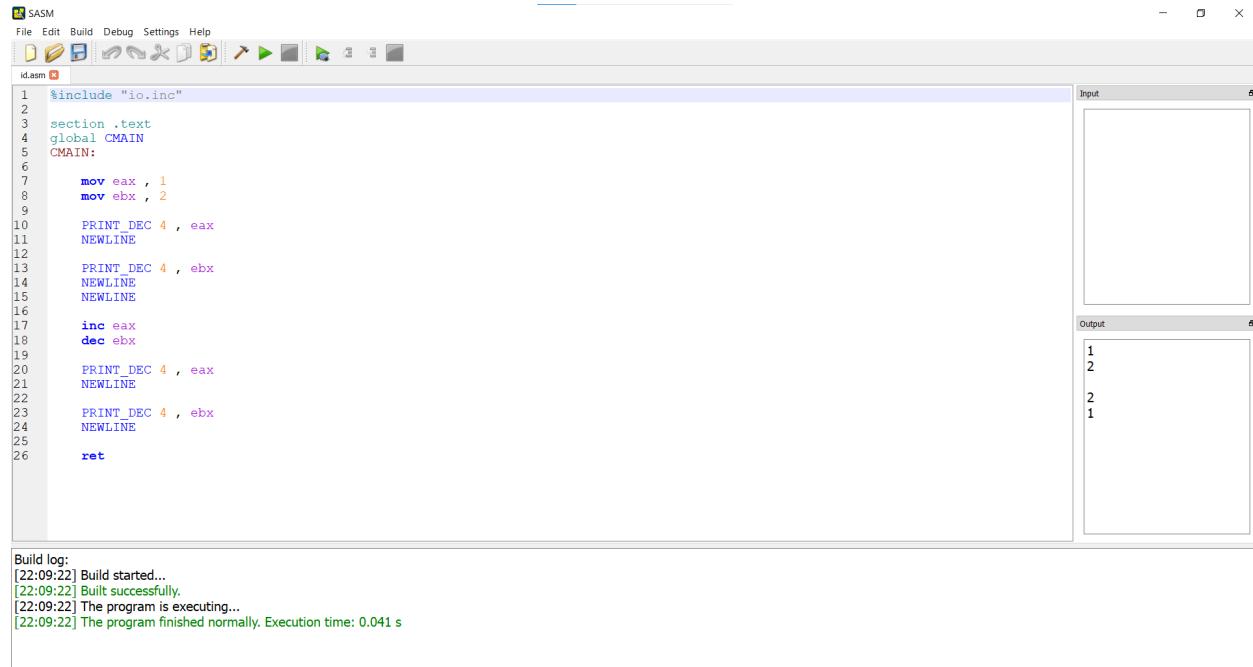
inc And dec Commands

Both `inc` and `dec` commands are simple. The `inc` command increments the value in a register by 1 and the `dec` command decrements the value in a register by 1

Have a look at the following code:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 1
8     mov ebx, 2
9
10    PRINT_DEC 4, eax
11    NEWLINE
12
13    PRINT_DEC 4, ebx
14    NEWLINE
15    NEWLINE
16
17    inc eax
18    dec ebx
19
20    PRINT_DEC 4, eax
21    NEWLINE
22
23    PRINT_DEC 4, ebx
24    NEWLINE
25
26    ret
```

Output



```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 1
8     mov ebx, 2
9
10    PRINT_DEC 4, eax
11    NEWLINE
12
13    PRINT_DEC 4, ebx
14    NEWLINE
15    NEWLINE
16
17    inc eax
18    dec ebx
19
20    PRINT_DEC 4, eax
21    NEWLINE
22
23    PRINT_DEC 4, ebx
24    NEWLINE
25
26    ret

```

Build log:
[22:09:22] Build started...
[22:09:22] Built successfully.
[22:09:22] The program is executing...
[22:09:22] The program finished normally. Execution time: 0.041 s

The output itself explains all!!

Extra Commands

jmp Command

jmp is just a command used to jump to a specified section of code, Let's see an example:

```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 100
8     PRINT_DEC 4, eax
9     NEWLINE
10
11    jmp sec
12
13    mov eax, 200
14    PRINT_DEC 4, eax
15    NEWLINE

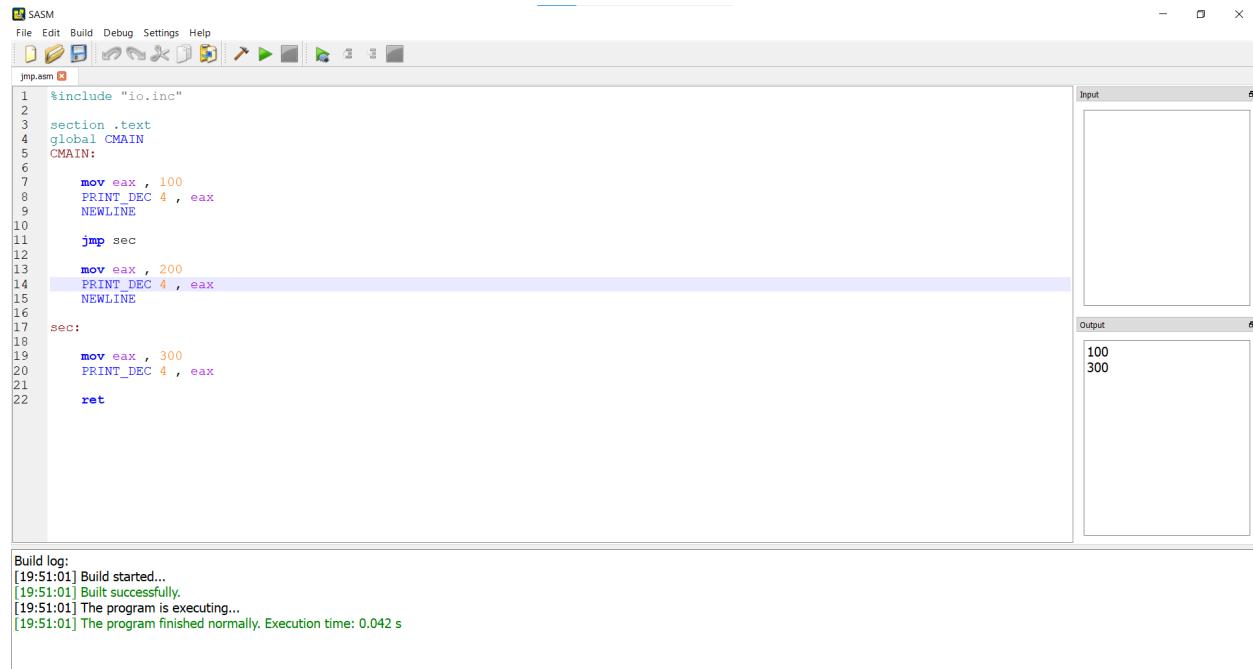
```

```

16
17 sec:
18
19     mov eax , 300
20     PRINT_DEC 4 , eax
21
22     ret

```

Output



The screenshot shows the SASM (Savannah Assembler) interface. The assembly code in the editor pane is:

```

1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax , 100
8      PRINT_DEC 4 , eax
9      NEWLINE
10
11     jmp sec
12
13     mov eax , 200
14     PRINT_DEC 4 , eax
15     NEWLINE
16
17 sec:
18
19     mov eax , 300
20     PRINT_DEC 4 , eax
21
22     ret

```

The 'Input' pane is empty. The 'Output' pane shows the results of the assembly execution:

```

100
300

```

The 'Build log' pane at the bottom shows the following messages:

```

Build log:
[19:51:01] Build started...
[19:51:01] Built successfully.
[19:51:01] The program is executing...
[19:51:01] The program finished normally. Execution time: 0.042 s

```

Here, We first moved the value 100 to the eax register and printed it to the screen along with a newline. Then we issued the command `jmp sec` which transfers execution control to the `sec` section.

The `sec` section is defined with the line `sec:` , When cpu executes the command `jmp sec`, It jumps to the first code in the `sec` section. This results in the execution of code which moves the value 300 to the eax register and prints it.

You also saw a part of code which tries to moves the value 200 to the eax register and print it, But practically, That code won't be executed as we jumped to another section of code before reaching that line.

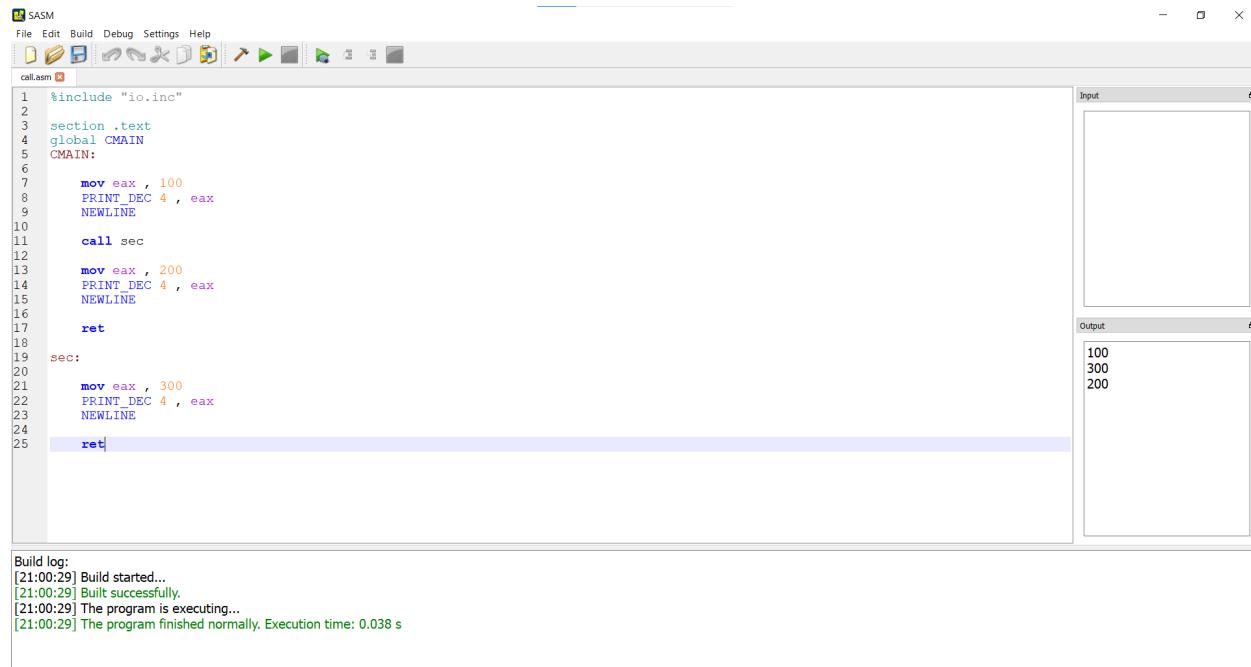
The thing to note here is that, if you want to create a new section, you need to suffix the section name with `:` , which means that if you want to create a section named `ADD` , You need to define it as `ADD:` . And to jump to that section, you need to call it like `jmp ADD`.

call Command

call is also a command which could be used to jump to other section but with an added capability. Let's make a small modification to the code we wrote for the jmp command :

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 100
8     PRINT_DEC 4, eax
9     NEWLINE
10
11    call sec
12
13    mov eax, 200
14    PRINT_DEC 4, eax
15    NEWLINE
16
17    ret
18
19 sec:
20
21    mov eax, 300
22    PRINT_DEC 4, eax
23    NEWLINE
24
25    ret
```

Output



The screenshot shows the SASM (Savannah Assembler) interface. The assembly code in the editor window is as follows:

```

1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax, 100
8      PRINT_DEC 4, eax
9      NEWLINE
10
11     call sec
12
13     mov eax, 200
14     PRINT_DEC 4, eax
15     NEWLINE
16
17     ret
18
19 sec:
20
21     mov eax, 300
22     PRINT_DEC 4, eax
23     NEWLINE
24
25     ret

```

The 'Input' window is empty. The 'Output' window shows the following text:

```

100
300
200

```

Build log:

```

[21:00:29] Build started...
[21:00:29] Built successfully.
[21:00:29] The program is executing...
[21:00:29] The program finished normally. Execution time: 0.038 s

```

In this code, We first moved the value 100 to eax register and printed it.

Then, we took another approach to jump to the sec section using the `call` instruction.

Here, the difference from the `jmp` instruction is that, Before jumping to the section named sec, The cpu pushes the address of instruction after the `call` instruction to the stack before jumping to the sec section.

In our code, When the cpu executes the `call` instruction, It first pushes the address of the instruction `mov eax, 200` to the stack and jumps to the sec section.

In that section, We printed the number 300 to the screen.

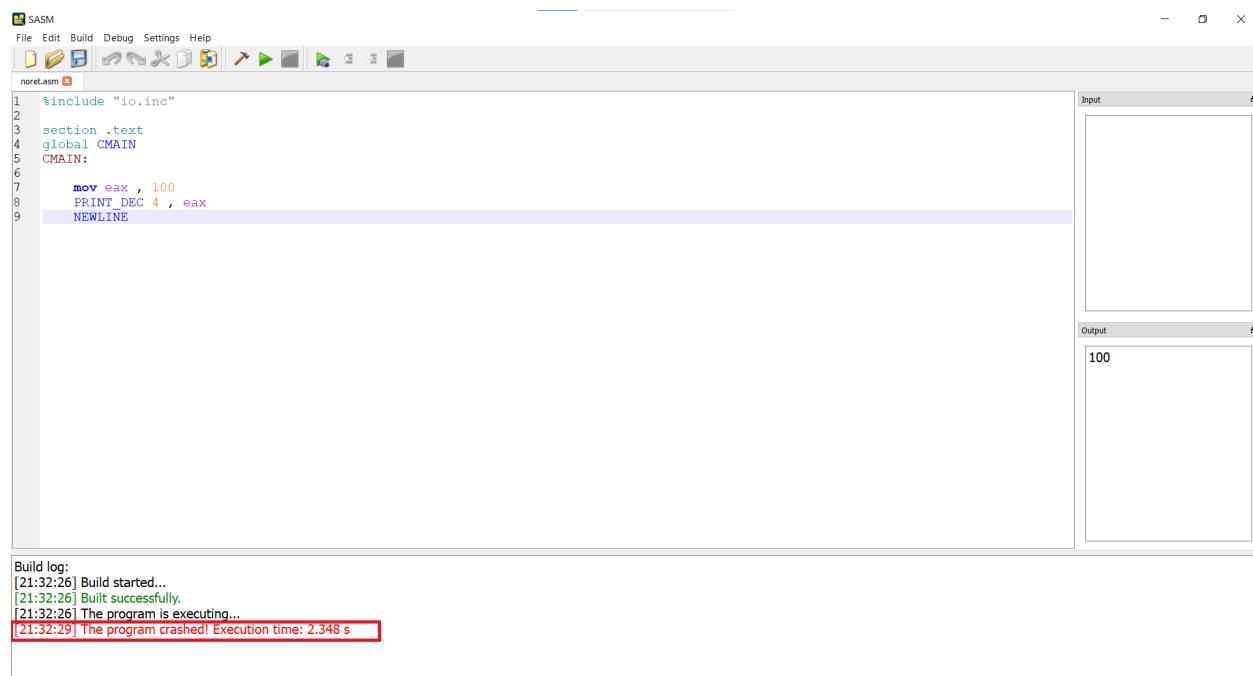
The last code in that section is `ret`, The `ret` instruction always pops the value at the top of the stack and stores it in the `eip` register (This is the register where the location of next instruction to be executed is present).

As the `call` instruction executed before, pushed an address to the stack, That pushed address itself will be popped to the `eip` register when the `ret` instruction is met. This results in the execution of the instruction `mov eax, 200` and then the printing of that number along with a newline. The next instruction to be executed is also `ret`. When this code is executed, The cpu will jump to code section which called our code which we defined as `CMAIN`. This will result in execution of code to terminate the process.

Have a look at the following code with no `ret` instruction and you can see that the program crashed:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 100
8     PRINT_DEC 4, eax
9     NEWLINE
```

Output



The screenshot shows the NASM assembly editor interface. The assembly code in the editor window is:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 100
8     PRINT_DEC 4, eax
9     NEWLINE
```

The output window on the right shows the result of the program execution:

```
100
```

Build log:

```
[21:32:26] Build started...
[21:32:26] Built successfully.
[21:32:26] The program is executing...
[21:32:29] The program crashed! Execution time: 2.348 s
```

The actual reason for the crash is that as we haven't included `ret` instruction, The processor will start executing every data in memory after our code.

The processor will practically take everything in memory as executable code and the chance is that it might run code which it doesn't know or do unnecessary things.

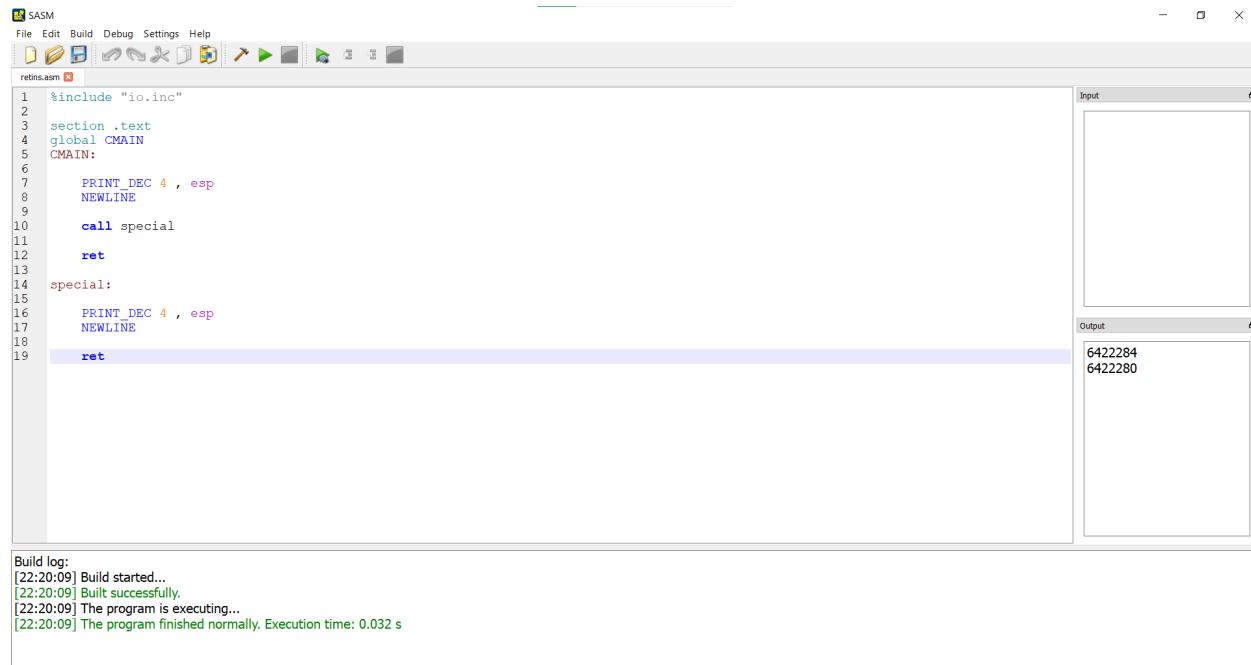
Now, Look at the following code:

```

1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     PRINT_DEC 4 , esp
8     NEWLINE
9
10    call special
11
12    ret
13
14 special:
15
16    PRINT_DEC 4 , esp
17    NEWLINE
18
19    ret

```

Output



The screenshot shows the SASM (Savannah Assembler) interface. The assembly code is in the main editor window. The input window (Input) is empty. The output window (Output) shows the values 6422284 and 6422280. The build log at the bottom shows the following messages:

```

Build log:
[22:20:09] Build started...
[22:20:09] Built successfully.
[22:20:09] The program is executing...
[22:20:09] The program finished normally. Execution time: 0.032 s

```

In this code, We first printed the value in esp register. This is the address of the data which is at the top of the stack.

Then, We called a section of code named `special` with the `call` instruction.

This will push the address of the instruction after the `call` instruction to the stack and jumps to the code section named `special`.

In the `special` section, we printed the value in `esp` and you can see that the value changed. This is because the `call` instruction pushed an address to the stack. This changed value is the address of value at the top of the stack, which here is the address to instruction after the `call` instruction. After returning from the section `special`, The value in `esp` will be back to normal : You could try printing it.

I believe you have learned a lot till this. We will learn some more concepts before learning OS-DEVELOPMENT. Don't forget to have fun learning!!

cmp Command

The `cmp` command is mostly used to compare two values to do a conditional execution.

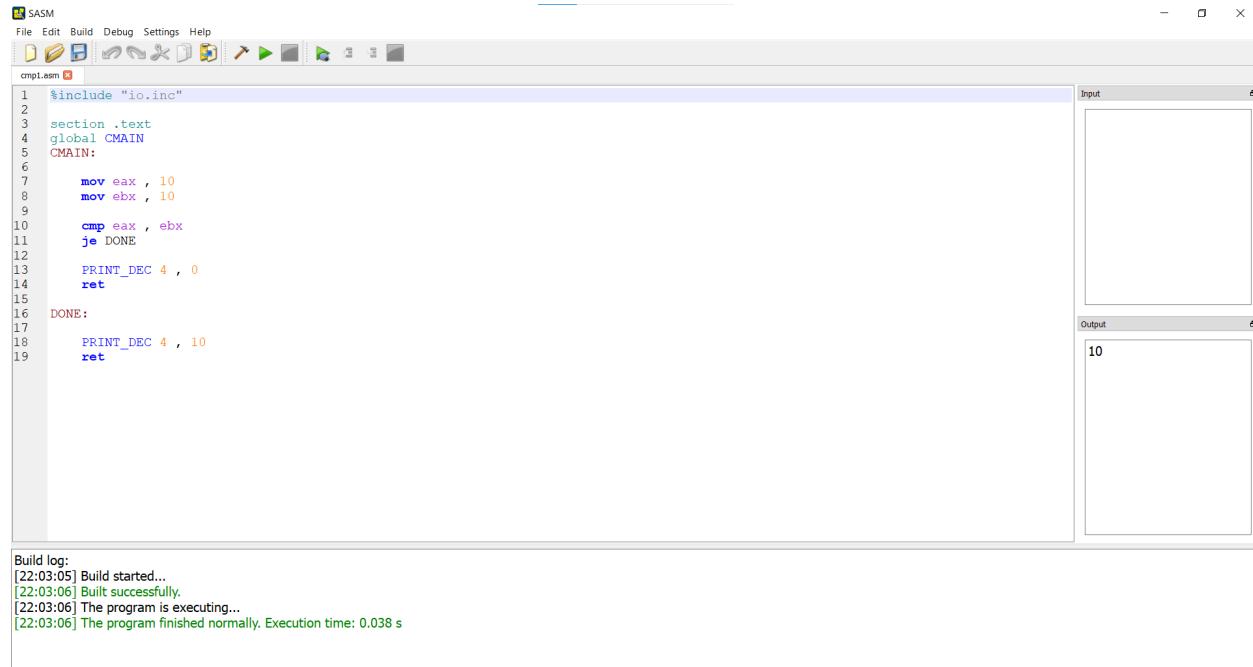
The `if` command we used in C makes use of the `cmp` instruction.

The `cmp` instruction makes changes to some Flag registers so that a logical jump can be made with other instructions.

See the following code:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax, 10
8     mov ebx, 10
9
10    cmp eax, ebx
11    je DONE
12
13    PRINT_DEC 4, 0
14    ret
15
16 DONE:
17
18    PRINT_DEC 4, 10
19    ret
```

Output



The screenshot shows the NASM assembler interface. The assembly code in the editor is:

```

1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax, 10
8      mov ebx, 10
9
10     cmp eax, ebx
11     je DONE
12
13     PRINT_DEC 4, 0
14     ret
15
16 DONE:
17
18     PRINT_DEC 4, 10
19     ret

```

The build log at the bottom shows:

```

Build log:
[22:03:05] Build started...
[22:03:06] Built successfully.
[22:03:06] The program is executing...
[22:03:06] The program finished normally. Execution time: 0.038 s

```

The output window shows the value 10.

Here, We first copied the value 10 to both eax and ebx registers.

Then, With the command `cmp eax , ebx`, We compared the values in both the eax and ebx register.

The next command we issued is the `je` command. It stands for `Jump Equal` or `Jump If Equal`.

When executing the `je DONE` command, The processor checks the comparison we done using the `cmp` command and if the values in both eax and ebx registers are equal, Then it will jump to the code section named `DONE`.

If the values are not equal, It will continue execution of command from `PRINT_DEC 4 , 0`. This means that the cpu will not jump to the section `DONE` if the values are not equal.

Now look at the following code:

```

1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax, 10
8      mov ebx, 12
9
10     cmp eax, ebx
11     je DONE
12
13     PRINT_DEC 4, 0

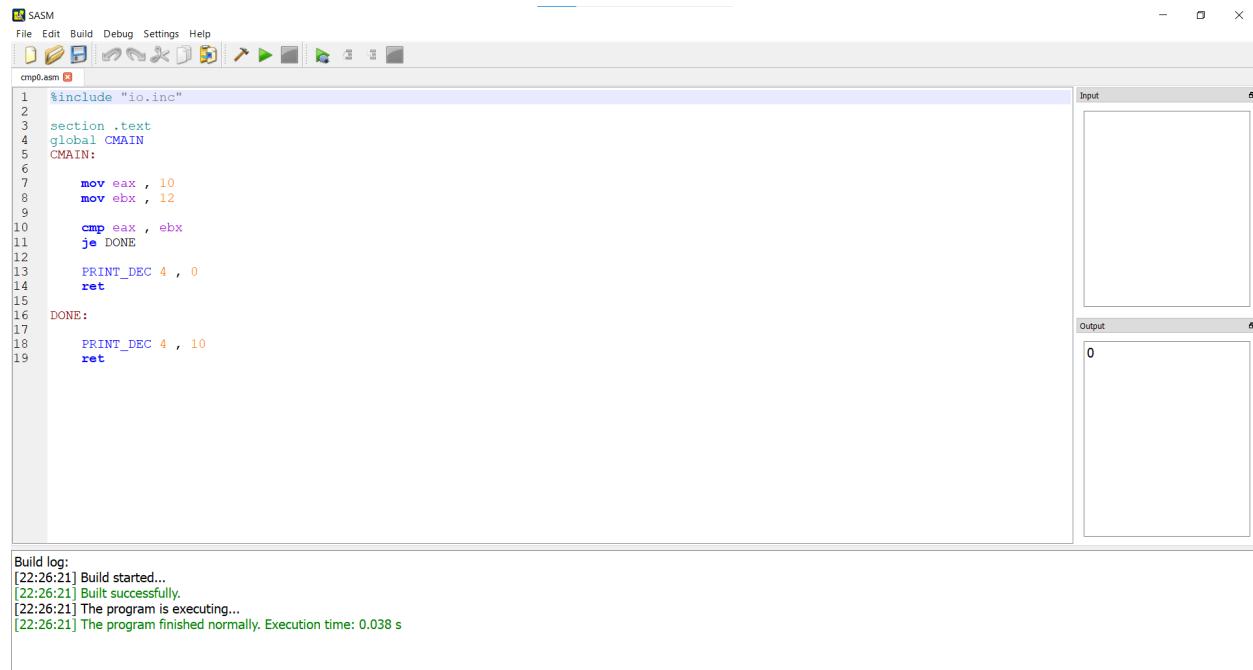
```

```

14     ret
15
16 DONE:
17
18     PRINT_DEC 4 , 10
19     ret

```

Output



```

1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax , 10
8      mov ebx , 12
9
10     cmp eax , ebx
11     je DONE
12
13     PRINT_DEC 4 , 0
14     ret
15
16 DONE:
17
18     PRINT_DEC 4 , 10
19     ret

```

Build log:
[22:26:21] Build started...
[22:26:21] Built successfully.
[22:26:21] The program is executing...
[22:26:21] The program finished normally. Execution time: 0.038 s

Here, We applied only a small change, We moved the number 12 to the ebx register. So now, The values in eax and ebx registers are completely different.

So when executing the `je DONE` command, The cpu realises that values in those registers are completely different, So it will not execute the instructions in the DONE section and executes the instruction `PRINT_DEC 4 , 0` and finally terminates.

Here, We have used `je` as conditional operator, but we have more commands that can be used after the `cmp` command. They are:

- 1)JE : Jump Equal
- 2)JNE : Jump Not Equal
- 3)JG : Jump Greater
- 4)JGE : Jump Greater Than Or Equal
- 5)JL : Jump Less
- 6)JLE : Jump Less Than Or Equal

There are more branching commands to use with the `cmp` command, You could refer to other sources to learn more, But for this book, This much is enough.

I also recommend you to master x86 assembly. You could do it after completing this book or along with this book. It's all upto you.

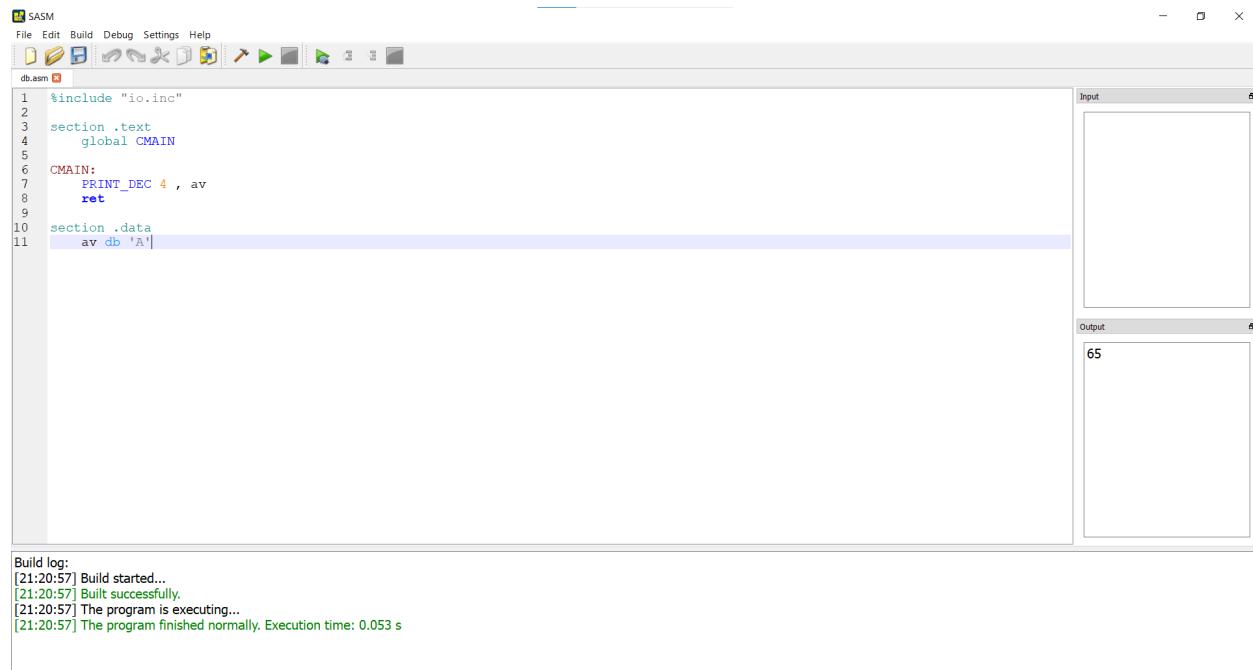
Variables

Nasm provides us with features to store variables in areas other than in the stack segment. **define** directives help us allocate these locations.

Have a look at the following:

```
1 %include "io.inc"
2
3 section .text
4     global CMAIN
5
6 CMAIN:
7     PRINT_DEC 4 , av
8     ret
9
10 section .data
11     av db 'A'
```

Output



The screenshot shows the NASM assembly editor interface. The left pane displays the assembly source code. The right pane shows the output window with the character 'A' and a build log at the bottom.

Build log:

- [21:20:57] Build started...
- [21:20:57] Built successfully.
- [21:20:57] The program is executing...
- [21:20:57] The program finished normally. Execution time: 0.053 s

Here you could see a section code starting with `section .data` and in that section, We declared a variable with `av db 'A'`.

Here `av` is the name of the variable, `db` says to allocate a byte and '`A`' will be assigned to the location reserved for `av`.

The output you saw on the screen is the ascii value of the character assigned to that variable.

You could also create an array of characters(String) with the `db` command itself.

Eg: `msg db 'Hello, world!', 0`

Here, some memory will be allocated to store the string `Hello, world!` ending with a null character(`0`). And it can be accessed by referencing with the variable name `msg`.

We have more keywords to allocate variables of different length. The full options are given below:

- 1) `db` : Allocates 1 byte
- 2) `dw` : Allocates 2 bytes(Word)
- 3) `dd` : Allocates 4 bytes(Doubleword)
- 4) `dq` : Allocates 8 bytes(Quadword)
- 5) `dt` : Allocates 10 bytes

Memory Addressing

In the c programming language, We used pointers to access memory directly. Accessing memory directly is also possible in assembly and there need not be a doubt for that.

Lets see an example:

```

1  %include "io.inc"
2
3  section .text
4      global CMAIN
5
6  CMAIN:
7      mov eax, msg
8      mov dl, [eax]
9
10     PRINT_DEC 1, dl
11     NEWLINE
12
13     inc eax
14     mov dl, [eax]
15
16     PRINT_DEC 1, dl

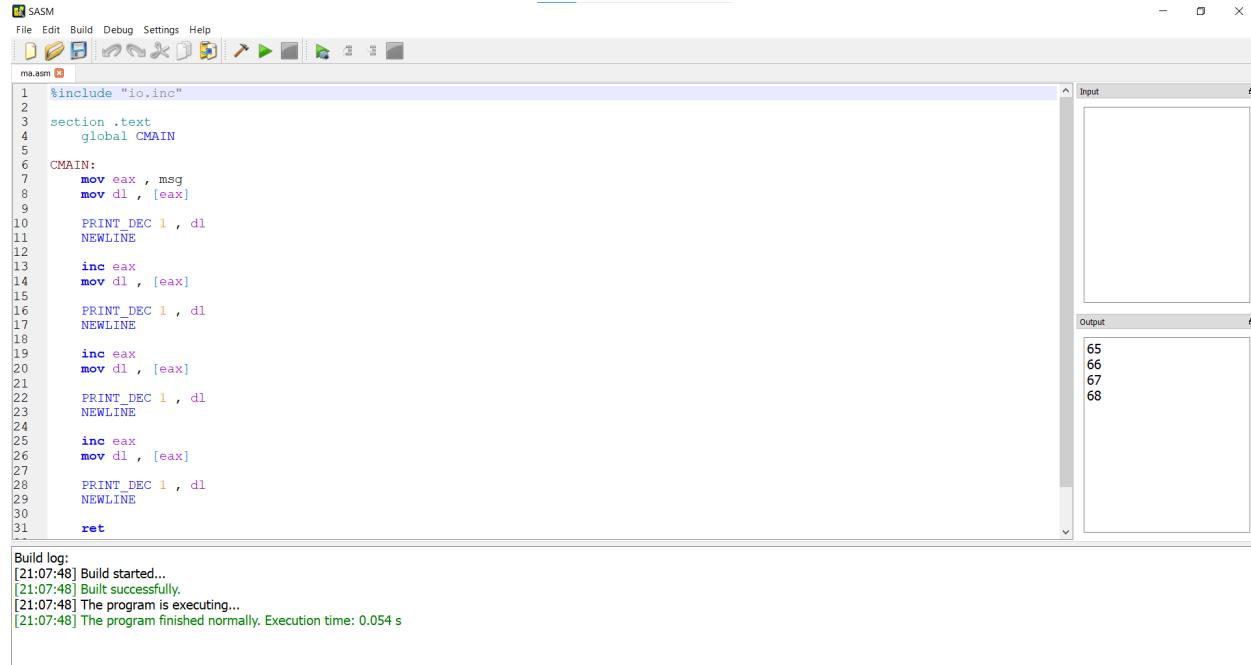
```

```

17     NEWLINE
18
19     inc eax
20     mov dl , [eax]
21
22     PRINT_DEC 1 , dl
23     NEWLINE
24
25     inc eax
26     mov dl , [eax]
27
28     PRINT_DEC 1 , dl
29     NEWLINE
30
31     ret
32
33 section .data
34     msg db 'ABCDEFGH'

```

Output



```

File Edit Build Debug Settings Help
ma.asm
1 %include "io.inc"
2
3 section .text
4     global CMAIN
5
6 CMAIN:
7     mov eax , msg
8     mov dl , [eax]
9
10    PRINT_DEC 1 , dl
11    NEWLINE
12
13    inc eax
14    mov dl , [eax]
15
16    PRINT_DEC 1 , dl
17    NEWLINE
18
19    inc eax
20    mov dl , [eax]
21
22    PRINT_DEC 1 , dl
23    NEWLINE
24
25    inc eax
26    mov dl , [eax]
27
28    PRINT_DEC 1 , dl
29    NEWLINE
30
31    ret

```

Input

Output

65
66
67
68

Build log:
[21:07:48] Build started...
[21:07:48] Built successfully.
[21:07:48] The program is executing...
[21:07:48] The program finished normally. Execution time: 0.054 s

Here, At the very last of the program, We declared a variable named `msg` as:

```
msg db 'ABCDEFGH'
```

Whenever we reference something with the name `msg`, We will get the address of the string ABCDEFGH.

The very first line in the code is:

```
mov eax , msg
```

This line moves the address of the string ABCDEFGH to the eax register.

The next line is as follows:

```
mov d1 , [eax]
```

Here, the Square Brackets in [eax] is used so that computer when at execution, will take the value in eax register as an address and take the data in that address to the d1 register. So Here, Square Brackets are used as a sign to access memory.

The processor will only fetch one byte to the d1 register as the size of d1 register is only one byte.

Then we printed the value in d1 register with the command PRINT_DEC 1 , d1.

This will print the ascii value of character A, as the first character in the variable msg is A itself.

Then we incremented address and printed each character's ascii values.

Here, We used the command `mov d1 , [eax]` to copy the first character's ascii. We accessed the next value after incrementing the address with `inc` command. if we want to access the second character's ascii without using the `inc` command, You could use something like `mov d1 , [eax + 1]`. This will give us the second character's ascii, likewise the command `mov d1 , [eax + 2]` will give us third character's ascii. Further increment will give us further values in the msg variable.

Think of another case where we want to access the data in the address 61ff0c specifically. Then, We could use the command `mov eax , [0x61ff0c]`. This will always move the value in address 61ff0c to the eax register. But accessing a specific address like this wont work every time as when we are working on a pre made operating system(Windows here), the operating system may deny access to that memory which might crash the process. This is because in windows and most other current operating systems, some memory parts are protected. But if the address we specify is not protected, We will get the data.

But when running the os WE made, We will get access to every locations as there is nothing preventing us from doing that. The kernel of the operating system have access to all memory locations.

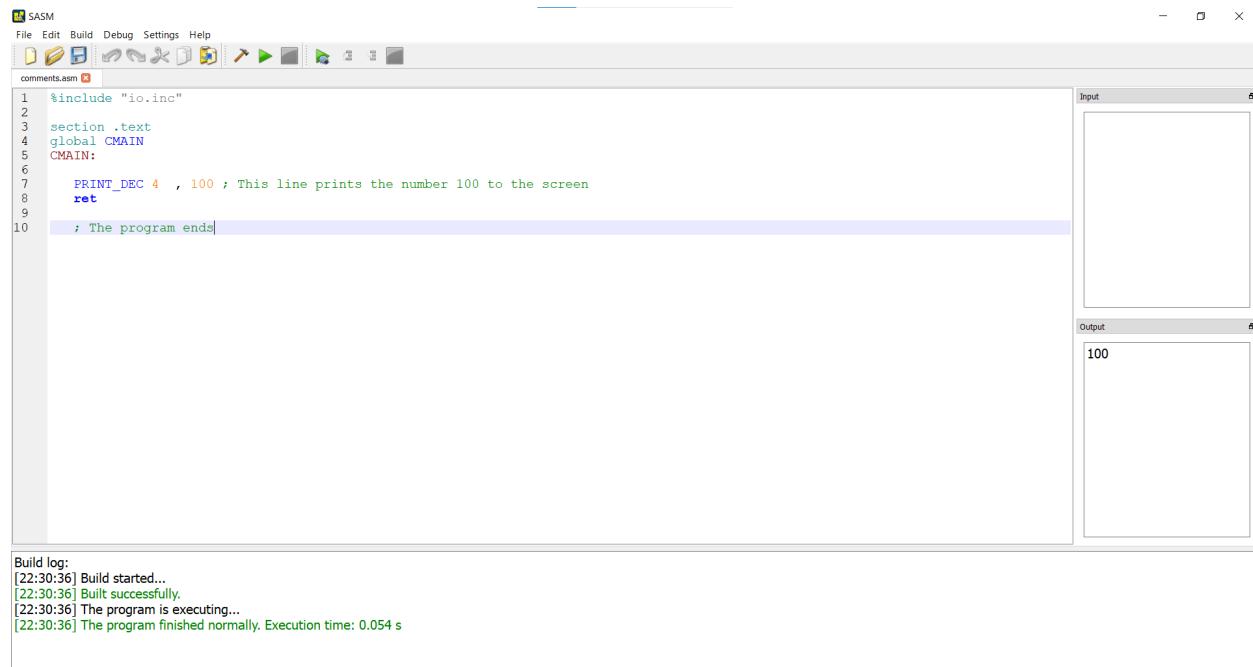
Comments

We could use the commenting feature in nasm to document our code, We could do it using ' ; '

Example:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7 PRINT_DEC 4 , 100 ; This line prints the number 100 to the screen
8 ret
9
10 ; The program ends
```

Output



The screenshot shows the SASM debugger interface. The assembly code is in the main editor window, and the output is displayed in the 'Output' window. The assembly code is as follows:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7 PRINT_DEC 4 , 100 ; This line prints the number 100 to the screen
8 ret
9
10 ; The program ends
```

The 'Output' window shows the number 100, indicating the program's output. The 'Build log' at the bottom of the interface shows the following messages:

```
Build log:
[22:30:36] Build started...
[22:30:36] Built successfully.
[22:30:36] The program is executing...
[22:30:36] The program finished normally. Execution time: 0.054 s
```

Conclusion

Finally, You have completed everything to get started into developing your own os.

Now we will start learning operating system development, which you was eagerly waiting for. I think you have enjoyed this far, but the upcoming chapter are more interesting. Let's GOOOOOO!!!

Beginning Operating System Development

Introduction

We have travelled a long road till here. And we are now at the Main and Interesting topic. Before beginning, We have to know what an operating system is.

The early computers(At the very begining of computing), didn't have any operating system. In those days, computers are mainly used for mathematical calculation and for research purposes. As there is no operating system those days, The developers who want to solve a mathematical or other problems need to include extra code to handle memory, input/output devices etc along with the code to solve the problem they intend to solve.

As this is a repetitive job to include extra code along with the code to solve thier problem every time, They started the idea of Operating Systems. So, they made some code which manages memory , io devices etc and put it into the computers they ran. Later, If they need to solve a particular problem, They only need to focus on the code to solve the problem they want to solve as the Operating System they previously made will do all other tasks.

We all are lucky these days, We have plenty of operating systems to use and we have even writen programs for one. When writing and running your own operating system, It's you, who have the supreme power on your computer. You will manage all io devices and everything will be in your control. You could even write a GAME and run it on your computer with no operating system. You will get direct access to your video memory which lets you create your own graphics, Control it with keyboard and mouse and MAINLY with all of the processor power for your program only.

Anyway, We are going to start it, Lets Go!!

Writing Programs For Boot Sector

Boot Sector is the location in Hard Disk where we place code that we want to run. Most of the times, It contains code to load the operating system installed in that machine. But the code in boot sector is not the first code to get executed.

When we power on our computer, It starts copying some piece of code named Bios from ROM(Read Only memory) to main memory(RAM).

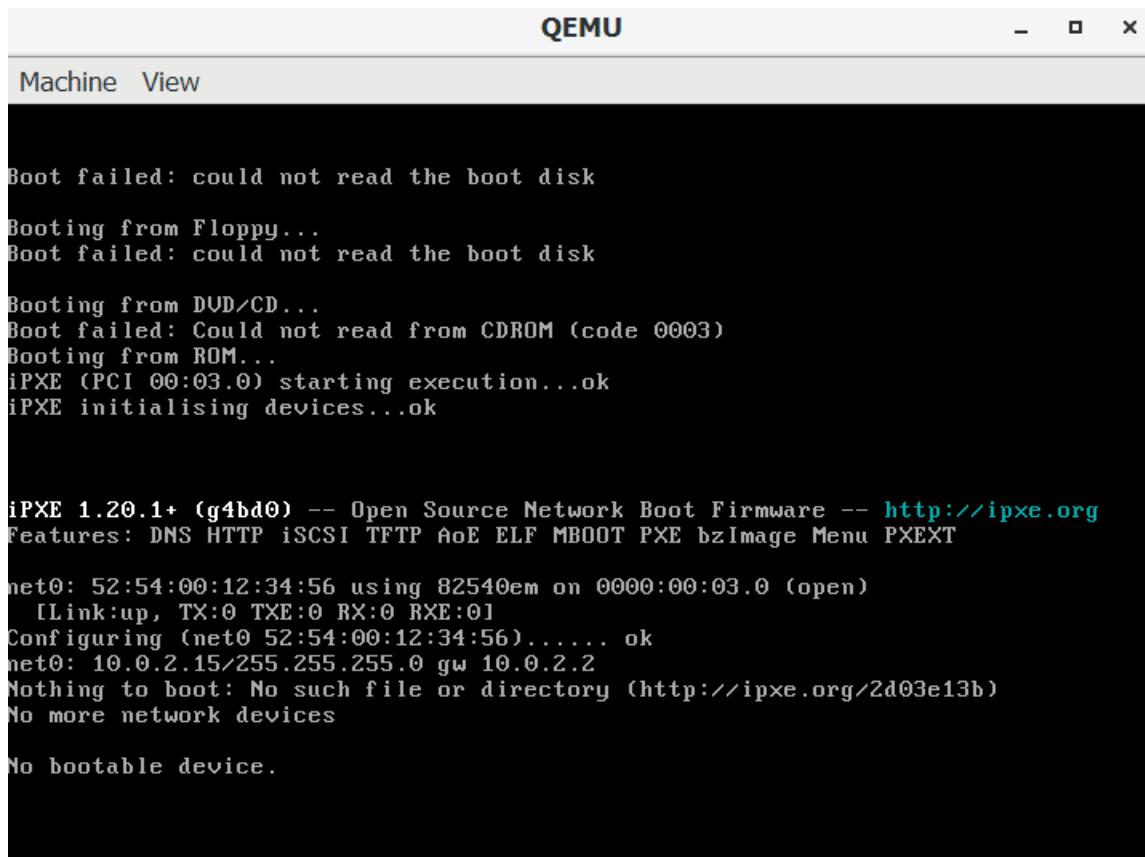
Then the cpu executes it. The main function of bios is to check if all connected devices are working properly and finally Load some code from the first 512 bytes in the hard disk(Or #1 Bootable device) to Memory and execute it.

The location of this first 512 bytes is named as Boot Sector. We could write some programs and place it into the boot sector to execute it.

We are going to run our operating system in QEMU for now. Type the following command in cmd and hit enter:

```
qemu-system-i386
```

You will get a window like the following:



```

QEMU

Machine View

Boot failed: could not read the boot disk
Booting from Floppy...
Boot failed: could not read the boot disk
Booting from DVD/CD...
Boot failed: Could not read from CDROM (code 0003)
Booting from ROM...
iPXE (PCI 00:03.0) starting execution...ok
iPXE initialising devices...ok

iPXE 1.20.1+ (g4bd0) -- Open Source Network Boot Firmware -- http://ipxe.org
Features: DNS HTTP iSCSI TFTP AoE ELF MBOOT PXE bzImage Menu PXEXT

net0: 52:54:00:12:34:56 using B2540em on 0000:00:03.0 (open)
  [Link:up, TX:0 RX:0 RXE:0]
Configuring (net0 52:54:00:12:34:56)..... ok
net0: 10.0.2.15/255.255.255.0 gw 10.0.2.2
Nothing to boot: No such file or directory (http://ipxe.org/2d03e13b)
No more network devices

No bootable device.

```

It says no bootable device found, But that's OK as we didn't give it with a file to boot, It will keep saying that.

Let's create a 512 byte file using `nasm` and give it to QEMU so that it could execute it.

First create a file named `boot-sect0.asm` and include the following code:

```
1 times 512 db 0
```

Here, what the code does is create a 512 byte long file and initialize all of the bytes with `0`. You already know this command: The `db` command(Or `define byte`).

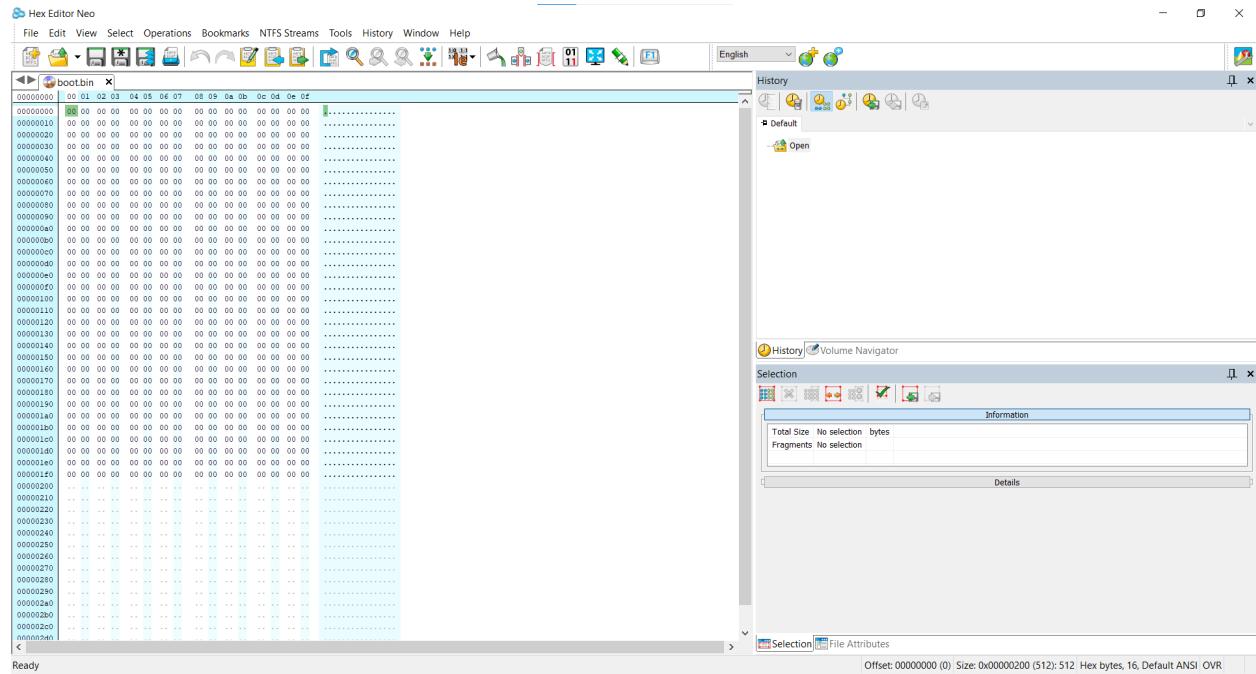
The `db` command defines a byte, But as we included the command `times 512` in `times 512 db 0`, It will define 512 bytes, not 1 byte. The `0` at the end is used to initializes all of the bytes with `0`.

assemble it using the following command:

```
nasm boot-sect0.asm -f bin -o boot.bin
```

In this command, `boot.bin` will be the name of the 512 byte file which the assembler will create.

Open this file with the `Hex Editor` that we previously downloaded, You will see the following output:



You could see that the file is 512 bytes long and all of the bytes are initialized with 0.

Now, we could boot this file using QEMU. For that, Type the following command in cmd:

```
qemu-system-i386 -drive format=raw,file=boot.bin
```

Here, `boot.bin` is the name of file to be booted.

Output

```

QEMU

Machine View
Boot failed: not a bootable disk

Booting from Floppy...
Boot failed: could not read the boot disk

Booting from DVD/CD...
Boot failed: Could not read from CDROM (code 0003)
Booting from ROM...
iPXE (PCI 00:03.0) starting execution...ok
iPXE initialising devices...ok

iPXE 1.20.1+ (g4bd0) -- Open Source Network Boot Firmware -- http://ipxe.org
Features: DNS HTTP iSCSI TFTP AoE ELF MBOOT PXE bzImage Menu PXEXT

net0: 52:54:00:12:34:56 using 82540em on 0000:00:03.0 (open)
  [Link:up, TX:0 RX:0 RXE:0]
Configuring (net0 52:54:00:12:34:56)..... ok
net0: 10.0.2.15/255.255.255.0 gw 10.0.2.2
Nothing to boot: No such file or directory (http://ipxe.org/2d03e13b)
No more network devices

No bootable device.

```

It is showing that no bootable device is found. There is nothing wrong here, It won't boot everything which is fed into it. For the computer to boot our operating system, It need to know whether the file is actually an executable. This is implemented so that, The computer will not try to take any piece of data as executable until we say that it is an executable.

We need to make small changes to the code so that the computer can confirm that it is an executable.

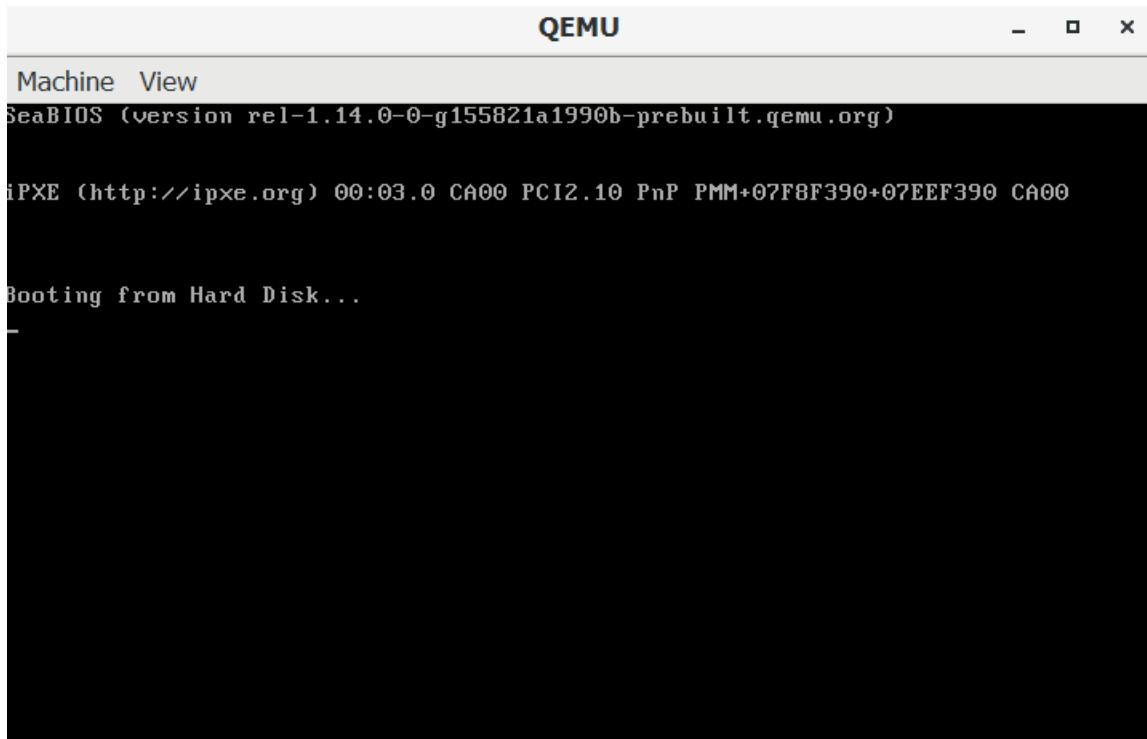
For that, put the following code into the source file and assemble it from cmd using nasm(Use the previous command which we used to assemble the source code using nasm):

```

1  loop:
2      jmp  loop
3
4  times 510-($-$$) db 0
5  dw 0xaa55

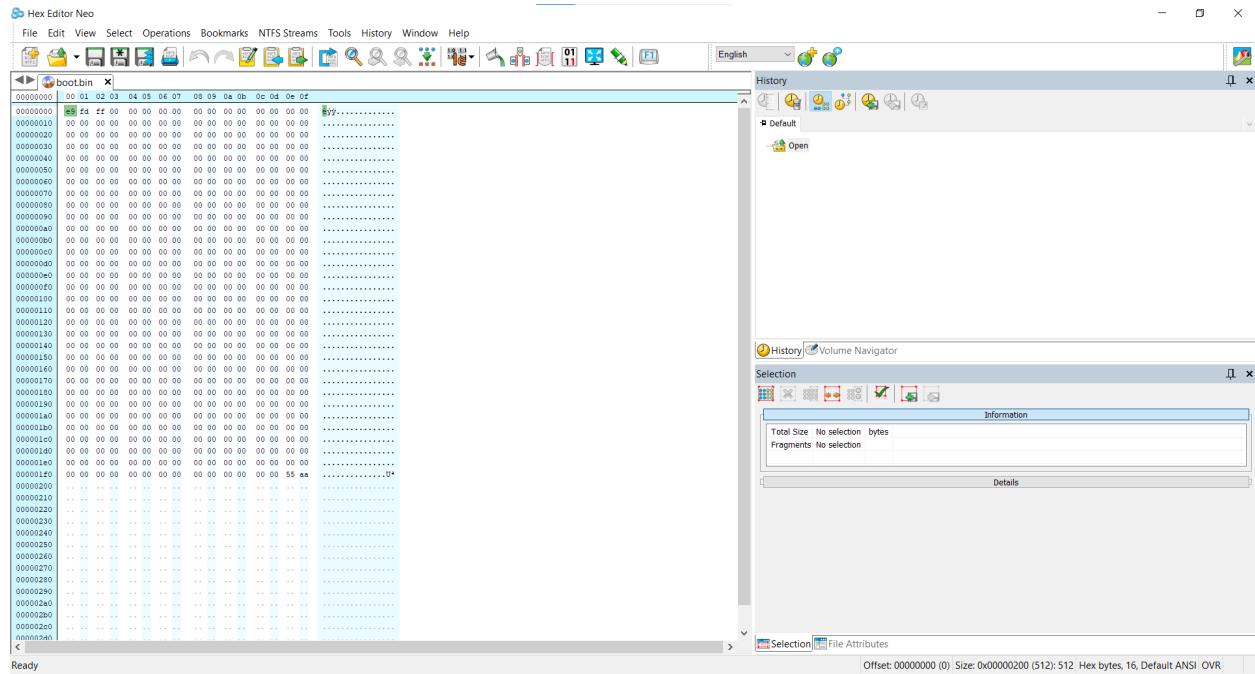
```

Now, Boot the file with qemu, We will see the following:



Congratulations!!!!, You have successfully created something that the computer could directly execute!!!

Lets learn how this works, Open the binary file we just booted, using the Hex Editor and look at its content.



Look at the very last two bytes here:

These bytes are called **Magic Number**, when our computer tries to boot from hard disk, It first checks if the last two bytes of the 512 byte boot sector is the this magic number or not, When it finds that the last two bytes are this magic number, It will start to execute the code from the 0'th location. In our case it executes the `jmp` instruction. This is an endless jump as it jumps every time to this `jmp` instruction itself so this program will not terminate.

We applied this `jmp` instruction here so that the computer will not execute the remaining bytes which we initialized it with 0. There is a chance for the computer to crash as it executes the data that is not intended to be executed.

We applied the magic number with the command `dw 0xaa55`. This command will attach the bytes `aa` and `55` to the end of the file generated.

But when looking at the Hex Editor, we can see that the bytes are `55` and `aa`, not `aa` and `55`.

This is because of the **endianess** property we discussed in the assembly programming section. The x86 compilers and assemblers will put multi-byte data type in reverse order. This is what that happened here.

You can also see a section in our code : `times 510-($-$$) db 0.`

The job of this command is to fill the content between the bytes representing the `jmp` instruction and `dw` instruction with 0.

It fills the content after the byte representation of the `jmp` instruction to the 510'th byte with 0.

The final `dw 0xaa55` command generates those bytes and places it to the very last of the file.

So finally, The size of the file becomes 512 bytes.

When creating further programs, We could replace the `jmp` instruction with whatever instruction we want, But we shouldn't take away the `dw` command which places the magic number at the last two byte locations. if we do that, The program won't be executed by the computer.

Printing To Screen (Hello , World OS)

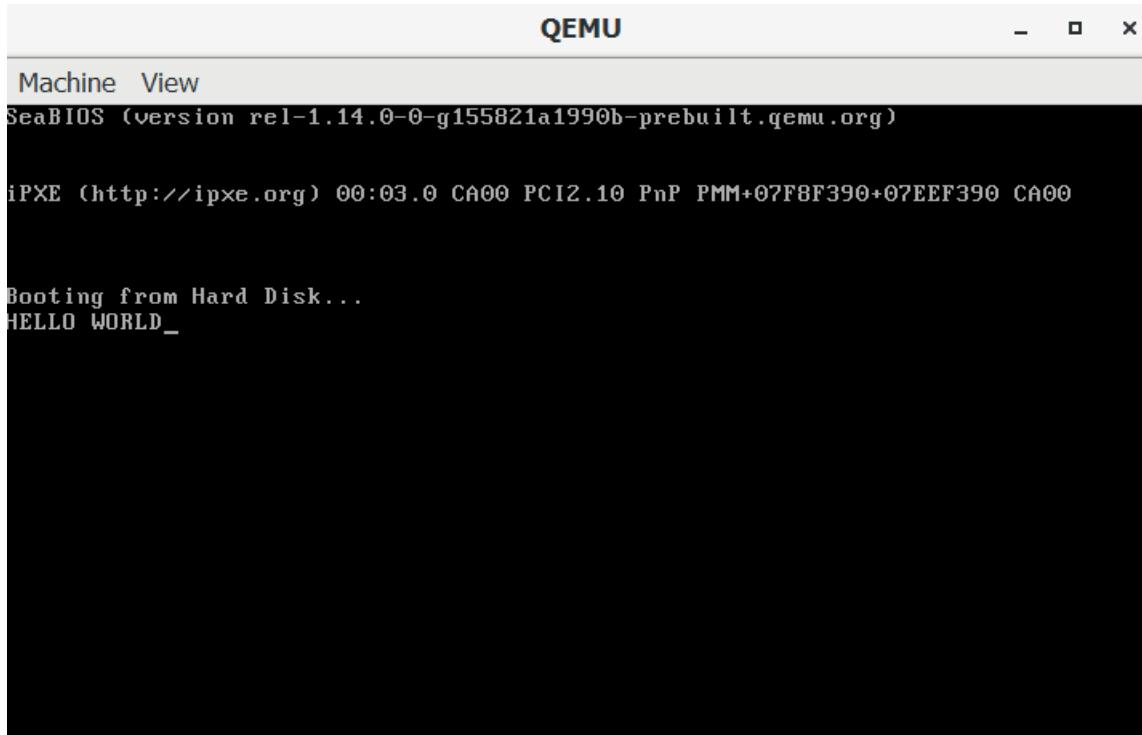
So now, Lets try printing something to the screen.

Save and assemble the following code:

```
1  mov ah, 0x0e
2
3  mov al , 'H'
4  int 0x10
5
6  mov al , 'E'
7  int 0x10
8
9  mov al , 'L'
10 int 0x10
11
12 mov al , 'L'
13 int 0x10
14
15 mov al , 'O'
16 int 0x10
17
18 mov al , ' '
19 int 0x10
20
21 mov al , 'W'
22 int 0x10
23
24 mov al , 'O'
25 int 0x10
26
27 mov al , 'R'
28 int 0x10
29
30 mov al , 'L'
31 int 0x10
32
33 mov al , 'D'
```

```
34 int 0x10
35
36 jmp $
37
38 times 510-($-$) db 0
39 dw 0xaa55
```

Boot the assembled file and we could see this:



We have successfully printed HELLO WORLD to the screen. Let's study the code:

Please note that, the computer will be initially in 16 bit mode(16 Bit Real Mode) after boot-up. So, We could only use 16 bit registers such as ax , bx or lower parts of it such as ah , al , bh , dl etc..

We will later learn how to switch to 32 bit mode so that we could use 32 bit registers such as eax , ebx and also lower parts of it such as ax , bx etc..

Here, we printed these characters by calling BIOS with int 0x10. The bios will do neccessary thing to print the characters to the screen.

We first copied the hex value 0e to the ah register with the command : `mov ah, 0x0e`. This value is named as The Scrolling Teletype. Then we copied the character H to the al register using the command `mov al, 'H'`. Finally we called Bios using `int 0x10`

`int` is a command in assembly known as Interrupt. This command is used to pause the current

execution and execute another code which is defined in a section named as Interrupt Descriptor Table.

`int 0x10` jumps to code section which have a list of display related routines. Bios selects which routine among them to be executed by looking at the values we passed to the registers.

When the Bios gets this interrupt, It first checks the value in `ah` register.

Here, as we copied the value `0e` to the `ah` register before the interrupt command, The bios realises that we want to implement a scrolling teletype. A scrolling teletype advances the cursor position to the next column in the screen after printing a character.

The Bios knows which character to print to the screen by referring to the value in `al` register. As we previously passed the character `H` to `al` register, The bios prints the character `H` to the screen and advances the cursor to the next column in screen.

After Bios prints that character, It passes the control of execution to the code we created. So practically, In our code, It will execute code which moves the character `E` to the `al` register and continue printing the character. With this method, we have printed the string `HELLO WORLD` fully to the screen.

The last command(`"jmp $"`) is an endless jump which jumps to that line itself.

This is so that the computer(As explained earlier), Will not try to execute data which is not intended to be executed.

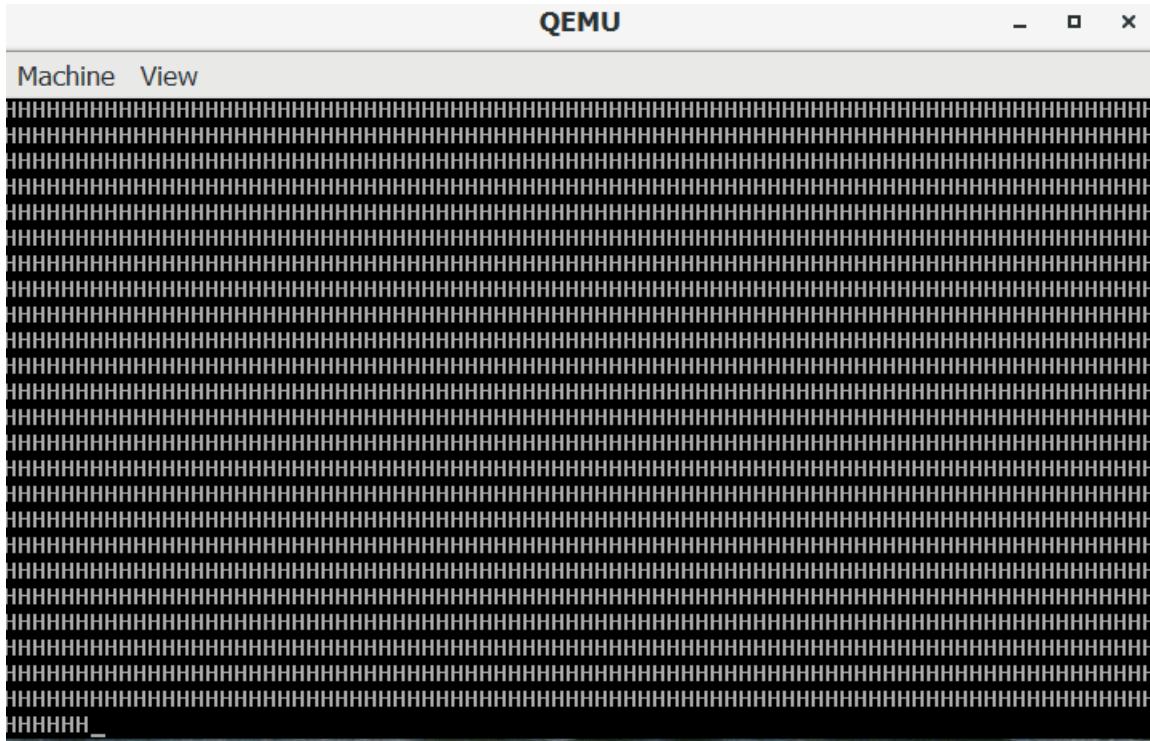
Filling The Screen With Characters(For Fun)!!

This section is just for FUN!!!, Lets fill the screen with some data.

I implemented it with the following code:

```
1 mov ah, 0x0e
2
3 Noend:
4
5     mov al, 'H'
6     int 0x10
7     jmp Noend
8
9 times 510-($-$$) db 0
10 dw 0xaa55
```

Output



Filling The Screen With Colours

Bios also allows us to fill the screen with colours, Take a look at the following code:

```
1 mov ah , 0x0b
2 mov bh , 0x0
3 mov bl , 0xff
4 int 0x10
5
6 mov ah, 0x0e
7
8 mov al , 'H'
9 int 0x10
10
11 mov al , 'E'
12 int 0x10
13
14 mov al , 'L'
15 int 0x10
16
17 mov al , 'L'
18 int 0x10
```

```
19
20 mov al , '0'
21 int 0x10
22
23 mov al , ' '
24 int 0x10
25
26 mov al , 'W'
27 int 0x10
28
29 mov al , '0'
30 int 0x10
31
32 mov al , 'R'
33 int 0x10
34
35 mov al , 'L'
36 int 0x10
37
38 mov al , 'D'
39 int 0x10
40
41 jmp $
42
43 times 510-($-$$) db 0
44 dw 0xaa55
```

Output



Look at the first few lines of the above code:

```
mov ah , 0x0b
mov bh , 0x0
mov bl , 0xff
int 0x10
```

This few lines is the code which changed the color of screen. Try changing the value copied to bl register and you could see different colours.

You can goto the following link to get values for different colours.

https://en.wikipedia.org/wiki/BIOS_color_attributes

Other Bios Display Related Routines

This is an optional study. If you are interested, Please follow the following link to get information about other display related bios routines.

https://en.wikipedia.org/wiki/INT_10H

Running Programs Written In C

Switching To Protected Mode

Till here, We wrote programs in assembly language to interact with computer. Writing programs in assembly is both risky and time consuming. Writing the os in c is a good option(But some part of it always need assembly).

Before writing and running programs in C, We will switch our processor to 32 bit mode(Or 32 bit protected mode).

Switching to 32 bit mode allows us protect some memory locations from other user mode programs trying to access those location. If a user mode program could access these special locations where the Kernel of the os resides, Those programs could take control of the cpu so our OS would not have any control.

We can define which part of the memory area to be protected by defining it in a table named GDT or Global Descriptor Table. It is necessary to Set The GDT Before switching to 32 bit protected mode. After defining and loading the GDT, We could switch the cpu to 32 bit mode.

Defining The GDT

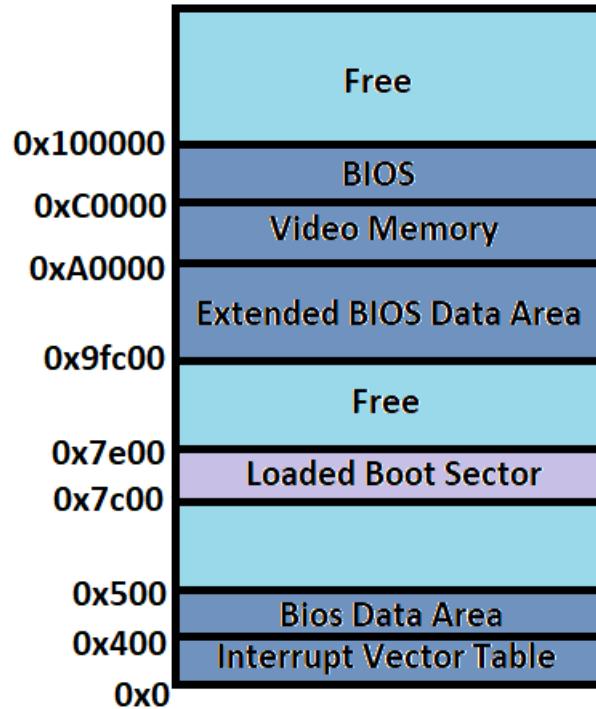
Life Without Bios

Before beginning this, Please know that we won't be able to use bios to do anything such as printing to screen after we define the GDT, Load it and finally Make the switch to Protected mode. We will make our own display related functions Later in this book.

Bios also gives other necessary functions such as for accessing i/o devices , Hard disks etc.... These too also will be unavailable after the switch.

Implementing The GDT

Lets see where our code and other resources will be present in memory:



From this image, You can see that the location where our Boot sector is present, is at 0x7c00.

This means that Bios will load our Boot sector to 0x7c00 in memory.

Before switching the processor to 32 bit mode, We need to define our Global Descriptor Table(GDT).

GDT is a special data Structure which the processor directly validates.

It is little complex to understand for the first time. But this this can't be avoided when trying to switch to protected mode. Let's First define GDT and switch to 32 bit mode:

Making The Switch

The Following code will define the gdt and Make the switch to 32 bit mode:

(Alternatively, You can go to <https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/GDT.asm> to get a fine copy of this code.)

```
1 [org 0x7c00]
2
3 [bits 16]
4
5 ;Switch To Protected Mode
6 cli ; Turns Interrupts off
7 lgdt [GDT_DESC] ; Loads Our GDT
8
9 mov eax , cr0
10 or eax , 0x1
11 mov cr0 , eax ; Switch To Protected Mode
12
13 jmp CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
14 ;Forces the cpu to flush out contents in cache memory
15
16 [bits 32]
17
18 INIT_PM:
19 mov ax , DATA_SEG
20 mov ds , ax
21 mov ss , ax
22 mov es , ax
23 mov fs , ax
24 mov gs , ax
25
26 mov ebp , 0x90000
27 mov esp , ebp ; Updates Stack Segment
28
29
30
31
32
33 jmp $ ;Hang , Remove This To Start Doing Things In 32 bit Mode
34
35
36
37
38
39 GDT_BEGIN:
40
41 GDT_NULL_DESC: ;The Mandatory Null Descriptor
42 dd 0x0
43 dd 0x0
```

```

44
45 GDT_CODE_SEG:
46     dw 0xffff          ;Limit
47     dw 0x0             ;Base
48     db 0x0             ;Base
49     db 10011010b      ;Flags
50     db 11001111b      ;Flags
51     db 0x0             ;Base
52
53 GDT_DATA_SEG:
54     dw 0xffff          ;Limit
55     dw 0x0             ;Base
56     db 0x0             ;Base
57     db 10010010b      ;Flags
58     db 11001111b      ;Flags
59     db 0x0             ;Base
60
61 GDT_END:
62
63 GDT_DESC:
64     dw GDT_END - GDT_BEGIN - 1
65     dd GDT_BEGIN
66
67 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
68 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
69
70 times 510-($-$$) db 0
71 dw 0xaa55

```

You can try assembling and running this, But it won't show any special output as what we have done is only a switch to 32 bit mode.

Let's now understand what this code does:

The very first line [`org 0x7c00`] Says to the assembler to make adjustments to the program so that it could calculate `0x7c00` as the address which it will be loaded to.

The second line [`bits 16`] says to produce 16 bit opcodes for instructions given below it.

The line `c1i` turns the interrupt services off. We previously made some interrupts to bios with the `int` command. from now onwards, we will not be able to use that feature. Please note, not to avoid this instruction when switching to 32 bit mode.

Next, We loaded the GDT we defined at the bottom of the code with: `lgdt [GDT_DESC]`

Now, focus on the following line in the code we made:

```

GDT_BEGIN:

GDT_NULL_DESC: ;The Mandatory Null Descriptor

dd 0x0
dd 0x0

```

Both GDT_BEGIN and GDT_NULL_DESC gives us the address to that location which we could later refer. The next two lines starting with dd defines two null pointer(0x0). This is a mandatory null descriptor when defining the gdt. These locations will be used by the processor when working on things with the gdt.

THE GDT IS A COMPLEX CONCEPT TO UNDERSTAND FOR THE FIRST TIME. AS THE MOTIVE OF THIS BOOK IS ONLY TO INTRODUCE OS DEVELOPMENT, WE WONT DELVE MORE INTO THIS TOPIC. THIS TOPIC MAY POSSIBLY DEMOTIVATE YOU.

THIS TOPIC IS INCLUDED ONLY BECAUSE IT CAN'T BE AVOIDED WHEN SWITCHING TO PROTECTED MODE.

YOU DON'T NEED TO WORRY ABOUT THE GDT WHEN YOU ARE STARTING THE JOURNEY. YOU CAN JUST COPY PASTE THIS CODE, WHAT WE MAINLY FOCUS IS AT THE PRACTICAL SIDE OF OS DEVELOPMENT SUCH AS ACCESSING HARD DISK, KEYBOARD, MOUSE, CREATING GRAPHICAL USER INTERFACE ETC AND MORE. WE WILL GO INTO THAT IN LATER CHAPTERS.

WHEN YOU THINK YOU WANT TO MASTER GDT, THE FOLLOWING ARTICLES WILL HELP:

```

https://wiki.osdev.org/Global\_Descriptor\_Table
https://wiki.osdev.org/GDT\_Tutorial

```

Look at the following lines:

GDT_CODE_SEG:

```

dw 0xffff ;Limit
dw 0x0 ;Base
db 0x0 ;Base
db 10011010b ;Flags
db 11001111b ;Flags
db 0x0 ;Base

```

These lines tells to the cpu about the code segment. We defined the base , limit and some flags. These values could be referenced using the name GDT_CODE_SEG.

Now lets see how we defined the data segment:

GDT_DATA_SEG:

```

dw 0xffff ;Limit
dw 0x0 ;Base
db 0x0 ;Base
db 10010010b ;Flags
db 11001111b ;Flags
db 0x0 ;Base

```

We could refer to this content in memory with the name GDT_DATA_SEG.

The line GDT_END: helps us refer to that name later to get the ending location of this GDT Entry.

The next line is the "Informer" of the GDT Entry we defined:

GDT_DESC:

```

dw GDT_END - GDT_BEGIN - 1
dd GDT_BEGIN

```

The line dw GDT_END - GDT_BEGIN - 1 stores the size of our GDT Entry. We have reduced 1 from GDT_END - GDT_BEGIN, This is because the size of GDT(According to the x86 cpu), is always one less than the real size.

The next line dd GDT_BEGIN Stores the beginning address of our gdt entry.

We passed the address of GDT_DESC to the lgdt command at top so that it could access the size and beginning location of our GDT.

The next two lines:

```

CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN

```

defines the CODE SEGMENT and DATA SEGMENT locations so that we could later refer to them. In the first line, the equ command assigns the Reslut of GDT_CODE_SEG - GDT_BEGIN(Substraction operation) to CODE_SEG. The same applies to the second line also.

Now, go to the top of the code at:

```

mov eax, cr0
or eax, 0x1
mov cr0, eax

```

This code sets the cr0 bit. This will switch the processor to 32 bit protected mode.

We finally need to implement a `far jump` to fully utilize 32 bit mode. We implemented it with `jmp CODE_SEG:INIT_PM`. This will clear the contents in cache memory and jump to The section named `INIT_PM`. Here is the `INIT_PM` section:

```
[bits 32]
INIT_PM:

    mov ax , DATA_SEG
    mov ds , ax
    mov ss , ax
    mov es , ax
    mov fs , ax
    mov gs , ax

    mov ebp , 0x90000
    mov esp , ebp
```

We started it with the line `[bits 32]`. This tells to the assembler to generate 32 bit opcodes for the instructions following it. This is necessary as we previously switched to 32 bit mode by setting the `cr0` bit.

later we used the `mov` command to move the address of data segment to all of the Segment Registers.

Finally we set the `esp` and `ebp` registers. This also is very necessary as the C Programs we later make will surely push values to the stack.

The final line `jmp $` is used to stop processor from executing further code. This is technically a jump to that `jmp` instruction itself.

IT'S FINE IF YOU ARE NOT VERY CLEAR ABOUT THIS TOPIC. YOU WILL BE ABLE TO UNDERSTAND THESE LATER.

Making Way For Running C Code

Now, as we switched to 32 bit mode, We could start writing our code in C, But first we need a way to call it from assembly. initially, Only 512 bytes are loaded to memory by bios. When we make a C program and compile it with the boot sector, It's size will always be greater that 512 bytes.

So we need to add code to the boot sector program which loads the remaining part of the code to memory. A `BOOT LOADER` is a piece of code in Boot Sector which loads the remaining part of the os to memory.

Let's make a Boot loader:

Making A Boot Loader

You could download the full code for this section and the next section(Calling Our C Kernel) from here :

<https://github.com/TINU-2000/OS-DEV/tree/main/Beginning-OS-DEV/Entering%20c>

ALSO NOTE TO INCLUDE ALL OF THE SOURCE FILES IN SAME FOLDER

Boot Loader is only a small program. So, we will be able to include the boot loader program with the GDT defining program we just made before.

Lets add some code to it:

Alternatively , you could go to the following link to get the specific code we are going to discuss in this section :

<https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/Entering%20c/Boot.asm>

```
1 [org 0x7c00]
2 [bits 16]
3
4 ;Boot Loader
5 mov bx , 0x1000 ; Memory offset to which kernel will be loaded
6 mov ah , 0x02 ; Bios Read Sector Function
7 mov al , 30 ; No. of sectors to read(If your kernel won't fit into 30 sectors , \
8 you may need to provide the correct no. of sectors to read)
9 mov ch , 0x00 ; Select Cylinder 0 from harddisk
10 mov dh , 0x00 ; Select head 0 from hard disk
11 mov cl , 0x02 ; Start Reading from Second sector(Sector just after boot sector)
12
13 int 0x13 ; Bios Interrupt Relating to Disk functions
14
15
16 ;Switch To Protected Mode
17 cli ; Turns Interrupts off
18 lgdt [GDT_DESC] ; Loads Our GDT
19
20 mov eax , cr0
21 or eax , 0x1
22 mov cr0 , eax ; Switch To Protected Mode
23
24 jmp CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
25 ;Forces the cpu to flush out contents in cache memory
26
27 [bits 32]
```

```
28
29 INIT_PM:
30 mov ax, DATA_SEG
31 mov ds, ax
32 mov ss, ax
33 mov es, ax
34 mov fs, ax
35 mov gs, ax
36
37 mov ebp, 0x90000
38 mov esp, ebp ; Updates Stack Segment
39
40
41 call 0x1000
42 jmp $
43
44
45
46
47
48 GDT_BEGIN:
49
50 GDT_NULL_DESC: ;The Mandatory Null Descriptor
51     dd 0x0
52     dd 0x0
53
54 GDT_CODE_SEG:
55     dw 0xffff          ;Limit
56     dw 0x0             ;Base
57     db 0x0             ;Base
58     db 10011010b      ;Flags
59     db 11001111b      ;Flags
60     db 0x0             ;Base
61
62 GDT_DATA_SEG:
63     dw 0xffff          ;Limit
64     dw 0x0             ;Base
65     db 0x0             ;Base
66     db 10010010b      ;Flags
67     db 11001111b      ;Flags
68     db 0x0             ;Base
69
70 GDT_END:
```

```

71
72 GDT_DESC:
73     dw GDT_END - GDT_BEGIN - 1
74     dd GDT_BEGIN
75
76 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
77 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
78
79
80 times 510-($-$) db 0
81 dw 0xaa55

```

Here, We added the bootloader at the very begining of the code:

```

mov bx , 0x1000 ; Memory offset to which kernel will be loaded
mov ah , 0x02 ; Bios Read Sector Function
mov al , 30 ; No. of sectors to read (If your kernel won't fit into 30 sectors , you may need
to provide the correct no. of sectors to read)
mov ch , 0x00 ; Select Cylinder 0 from harddisk
mov dh , 0x00 ; Select head 0 from hard disk
mov cl , 0x02 ; Start Reading from Second sector (Sector just after boot sector)
int 0x13 ; Bios Interrupt Relating to Disk functions

```

This code will load 30 sectors($512 * 30$ bytes) to the memory location `0x1000` which we copied to the `bx` register. This 30 sectors are excluding the boot sector(which bios loaded).

We requested for this service to the bios with the `int 0x13` interrupt.

Then, after the code where we switched to 32 bit mode, We called the code at location `0x1000` with the command `call 0x1000`.

This will result in the execution of code we just loaded to memory using our bootloader. We will see how to compile a C program and place it into the location `0x1000` in order to execute it.

YOU SHOULD SAVE THIS CODE TO A FILE, LETS NAME IT: `Boot.asm`

Calling Our C Kernel

In order to execute the c program we make, We must know where our entry c function is in memory. This may be hard to find as in the future, if we add more code to our c program, the offset where the entry function will be present might change.

We could solve this problem by using the linker to calculate automatically where the entry function will be.

NOTE: WE WON'T DISCUSS ANYTHING ABOUT LINKER SCRIPTS, WE WILL WORK ONLY ON THE COMMAND LINE

Open a new file and add the following code:

Alternate link :

https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/Entering%20c/Kernel_Entry.asm

```

1 START:
2 [bits 32]
3 [extern _start]
4     call _start
5     jmp $
```

Save this file and name it Kernel_Entry.asm

Here, we declared an extern function named `_start` with the command `[extern _start]`. When assembling this code, the assembler will leave a hint to the linker that there will be a function named `start`, in an outside source file which we will call with the code `call _start`.

There is an interesting matter here. We declared the function as `_start` and expect the linker to link with `start`.

This is very true in this case. for eg: If you want to call a function named `enter`, you should declare it as `_enter`.

As the name of our entry c function is `start`, we should call it as `_start`.

Now, we will make our c program, Open a new file and add the following code:

Alternate link :

<https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/Entering%20c/main.c>

```

1 int start(){
2     char* video_memory = (char*) 0xb8000;
3     *video_memory = 'K';
4 }
```

Name this file as `main.c`

Now, We have a lot of job to assemble , compile and link all of these files.

Typing lot of commands for every build is boring and time consuming. So we will include all of the commands to assemble , compile and link together in a `.bat` file.

After including all of the commands and saving it, We could double click on it and the build process will be done automatically.

For this, please create a folder named `bin` in the directory where our codes exist.

Now, include the following code in a new file named `compile.bat` (Don't save it in the `bin` folder, save it where our other programs are present):

Alternate link :

<https://github.com/TINU-2000/OS-DEV/blob/main/BEGINNING-OS-DEV/Entering%20c/compile.bat>

```
1 nasm Boot.asm -f bin -o bin\bootsect.bin
2 nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
3
4 gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
5
6 ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o
7
8 objcopy -O binary -j .text bin\kernel.img bin\kernel.bin
9 copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
10
11 qemu-system-i386 -drive format=raw,file=bin\os-image
```

Executing this file will Do the Build process and launch the final executable.

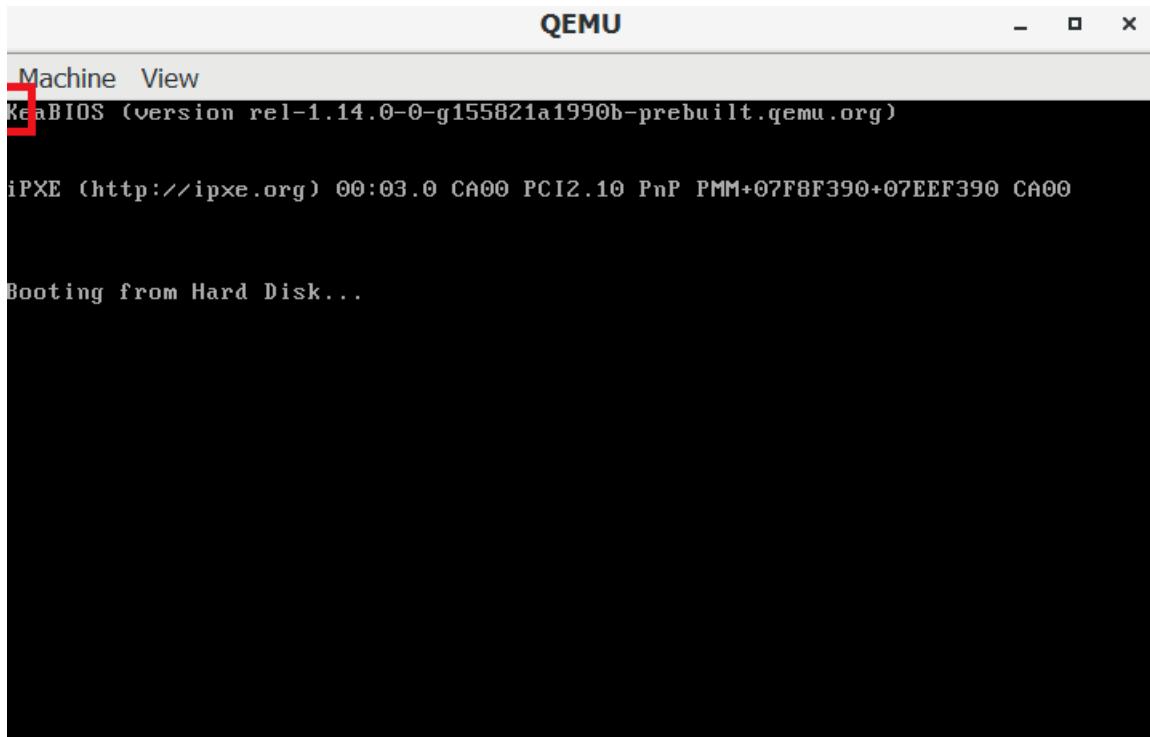
You could reuse this program every time. we will now be able to include more code to the c file, save it and finally click on the `.bat` file to automatically build and execute it.

All of the binary files will be generated to the `bin` folder

At the very last of our `.bat` file, we called `qemu` with: `os-image` as file to be booted.

Here, `os-image` is the final executable that we could distibute to others for use.

Run the `.bat` file and we will get the following output in `qemu`



The K at the top left of the screen ensures us that, the c program we made is working.

You can try changing the c program to print other characters.

The Fun Begins from the next chapter!!!!

NOTE: IT'S A GOOD IDEA TO CHECK IF THE .bat FILE WHEN LAUNCHED PRODUCES ANY ERROR MESSAGES TO THE TERMINAL. NO ERROR WILL BE PRESENT WHEN COMPILING THE ABOVE PROGRAM. BUT MAKE SURE TO CHECK WHEN SOME MODIFICATIONS ARE MADE.

Video Graphics

Introduction

Displaying something to screen is the crucial part when developing an operating system. As 90% of a computers output uses the Display system. Even during development, We may need to print the values in variables to debug any issue arising.

In this section, We are going to discuss about Text Mode User Interface. But this mode also allows printing colours to the screen.

Normally after the boot up, The computers video card will be in Text Mode. We could Print something to the screen by Poking memory location starting from 0xb8000.

We could write directly to this address in memory to print characters and colours.

The Text Mode interface allows 80 characters wide and 25 characters lines per screen. This means that the computer will be in 80 x 25 mode.

Let's learn how this works!!

NOTE: IT'S A GOOD IDEA TO USE A CROSS COMPILER WHEN DEVELOPING YOUR OS, AS OUR C COMPILER TOOLSET MINGW LACKS SUPPORT FOR RAW BINARY GENERATION HELPFUL IN WRITING OS. WINDOWS LACKS A GOOD NATIVE CROSS COMPILER. SO IT WILL BE PRACTICAL TO USE LINUX WHEN DEVELOPING THIS. LINUX GIVES ACCESS TO GOOD CROSS COMPILERS. YOU COULD INSTALL ALL OF THE MAIN TOOLS DESCRIBED IN THIS BOOK ON LINUX AND CONTINUE DEVELOPMENT.

SOME PROBLEMS WILL OCCUR WHEN USING GOLOBAL VARIABLES ETC... BUT IN THIS BOOK, WE WILL ONLY USE METHODS THAT YOU COULD FOLLOW IN WINDOWS. BUT IN FUTURE, WHEN TRYING TO DO SERIOUS PROJECTS, CONSIDER USING CROSS COMPILERS ON LINUX SPECIFICALLY i386 cross compilers OR YOU COULD EVEN TRY "WINDOWS SUBSYSTEM FOR LINUX"

Poking Video Memory

Displaying Text and Colours To Screen

We saw that the computer will be initially in 80 x 25 mode. The video memory in text mode starts from 0xb8000. Consider a situation where you want to print the character J to the Top Left of the screen, What you should do to implement this is to write the Ascii value of J To the location 0xb8000.

In c, It will look like this:

```
char* aa = (char*) 0xb8000;
*aa = 'J';
```

Here we declared a character pointer named `aa` and assigned it with the address `0xb8000`. Then in the next line, We took the value in `aa` as an address and assigned it with the Ascii value of `J`. This will print the character `J` to the Top Left of screen. Now, if you want to add background and foreground colour to that character, You could poke the next memory location(`0xb8001`) to give that character a colour.

For representing the colour, The first four bits in the byte represents background colour and next four bits represent foreground colour.

To give the first character a colour, You can use the following c program:

```
char* aa = (char*) 0xb8001;
*aa = 0xb5;
```

Here, we copied the hex value `b5`. `b` will be the background colour and `5` will be the foreground colour.

This means that the video card uses two bytes for representing a single character.

Suppose If you only want to Print the string `AS` to the screen with no colour. You could copy the character `A` to `0xb8000` and `S` to `0xb8002`.

So the total size of Text Mode Video Memory is $2 * 80 * 25$ bytes.

Lets see how to fill the screen with a special colour:

```
1 int start(){
2     char* cell = (char*) 0xb8000;
3     int i = 1;
4     while(i < (2 * 80 * 25)){
5         *(cell + i) = 0xd5;
6         i += 2;
7     }
8 }
```

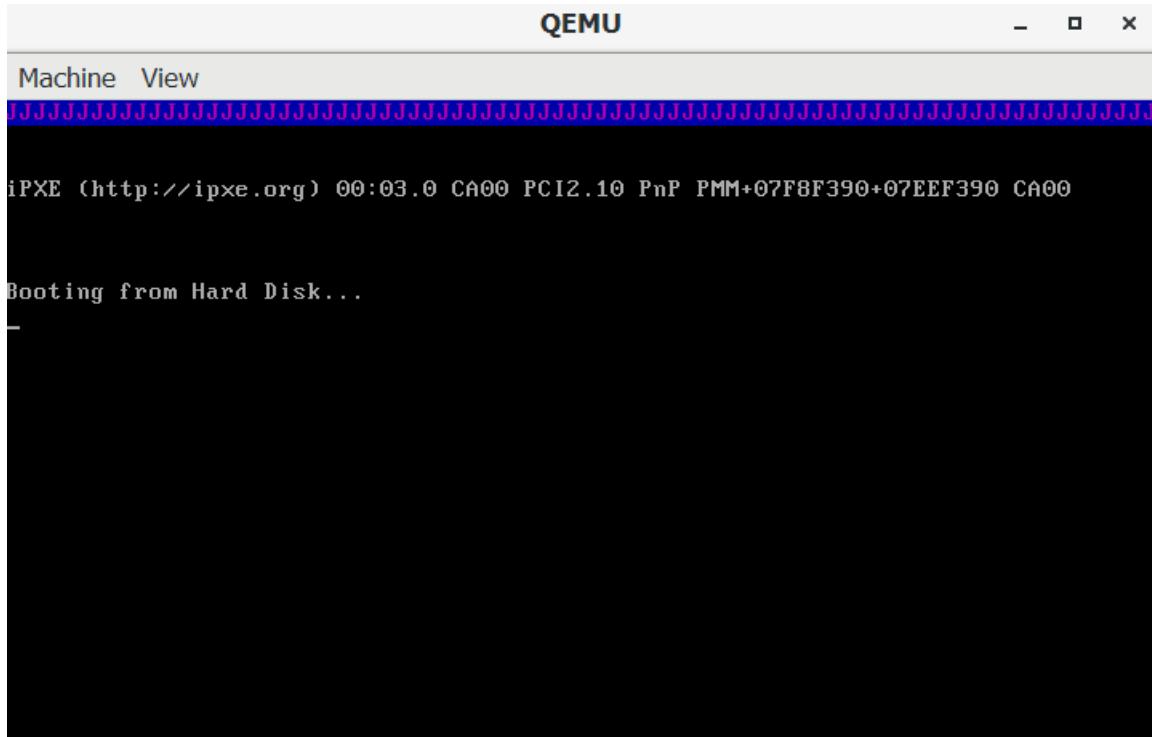
Save this program in the C Source file we created in the previous chapter. It will give the following output when executed:



Let's now see an example where the first row of the screen fills with a character:

```
1 int start(){
2     char* cell = (char*) 0xb8000;
3     int i = 0;
4     while(i < (2 * 80)){
5         *(cell + i) = 'J';
6         *(cell + i + 1) = 0x15;
7         i += 2;
8     }
9 }
```

Output



You could even make a GAME or SMALL SCALE GUI with this mode. In later chapters, we will learn how to switch to Graphics mode to display content to screen pixel by pixel.

Examples

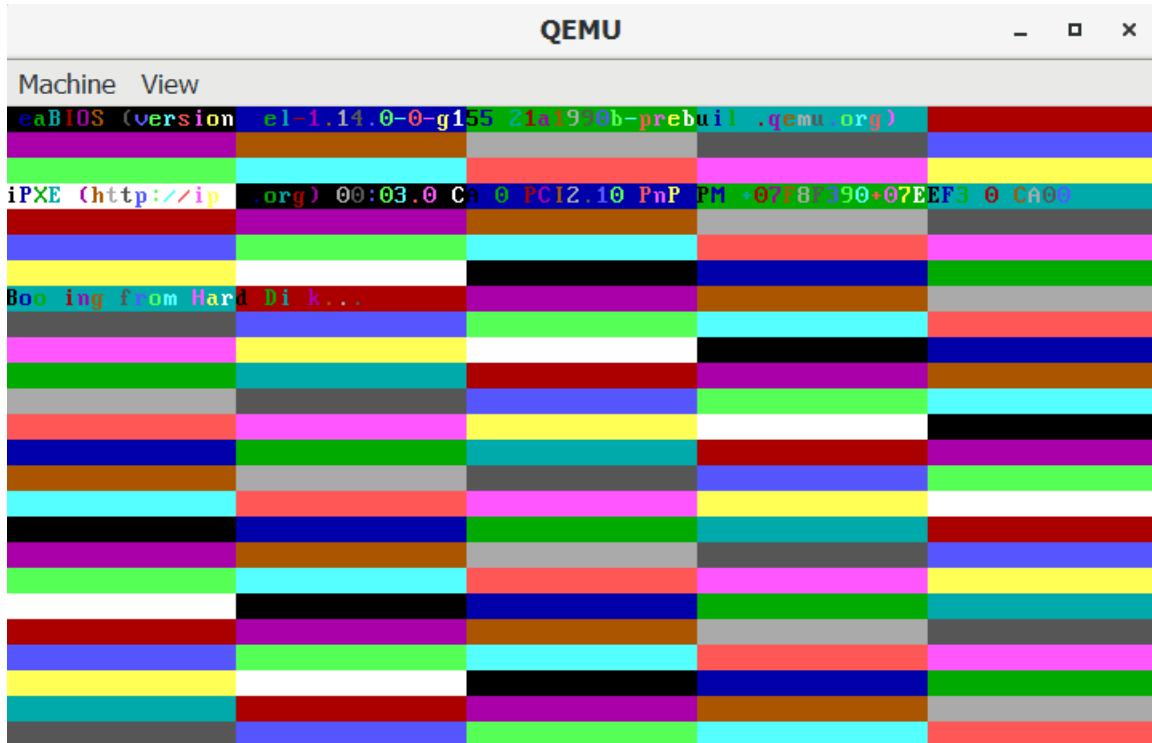
You could skip this section if you want, This is just for fun!!!!

Alpha

Program:

```
1 int start(){
2     char* TM_START = (char*) 0xb8000;
3     int i = 1;
4     char co = 0;
5     while(i < 2 * 80 * 25){
6         *(TM_START + i) = co;
7         i += 2;
8         co++;
9     }
10 }
```

Output:

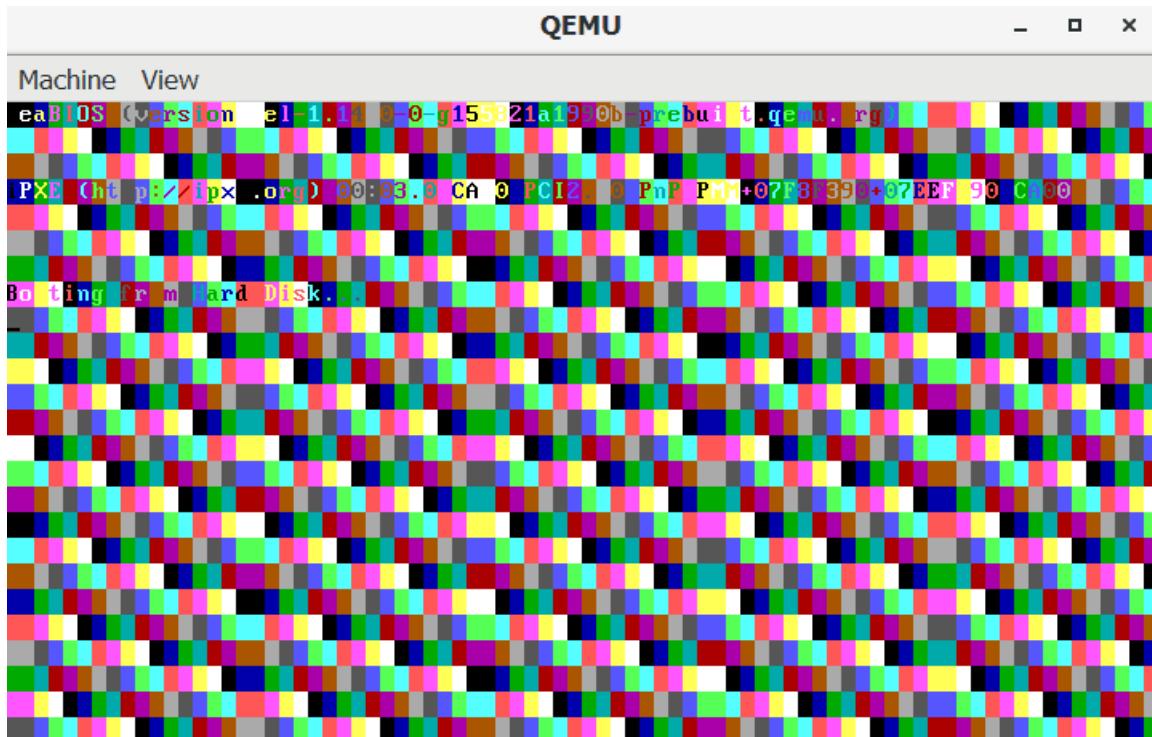


Beta

Program:

```
1 int start(){
2     char* TM_START = (char*) 0xb8000;
3     int i = 1;
4     char obj = 0;
5     while(i < 2 * 80 * 25){
6         *(TM_START + i) = obj;
7         i += 2;
8         obj += 15;
9     }
10 }
```

Output:

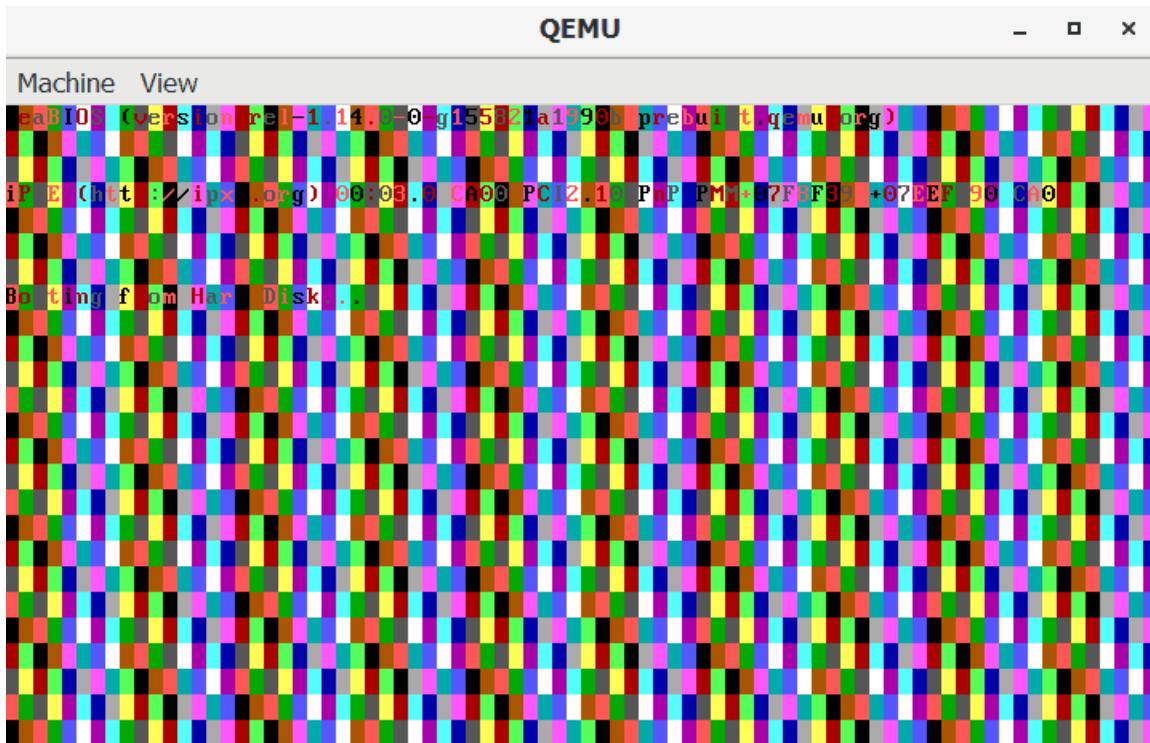


Gamma

Program:

```
1 int start(){
2     char* TM_START = (char*) 0xb8000;
3     int i = 1;
4     char obj = 0;
5     while(i < 2 * 80 * 25){
6         *(TM_START + i) = obj;
7         i += 2;
8         obj += 100;
9     }
10 }
```

Output:



Delta

Program:

```
1 int start(){
2     char* TM_START = (char*) 0xb8000;
3     int i = 0;
4     char obj = 0;
5     while(i < 2 * 80 * 25){
6         *(TM_START + i) = obj;
7         i++;
8         obj++;
9     }
10 }
```

Output:



Implementing Graphics Driver

In future, it's hard for us to print strings to the screen by looking at every details of Text Mode Video Graphics. So, we are going to implement our own "printf" function that we could later call to do the print operation.

Link to project files for this section :

<https://github.com/TINU-2000/OS-DEV/tree/main/Video%20Graphics0>

```

1 void cls();
2 void setMonitorColor(char);
3
4 void printString(char* );
5 void printChar(char);
6
7 void scroll();
8
9 void printColorString(char* , char);
10 void printColorChar(char , char);
11
12 void getDecAscii(int);
13

```

```
14 char* TM_START;
15 char NumberAscii[10];
16 int CELL;
17
18 int start(){
19     TM_START = (char*) 0xb8000;
20     CELL = 0;
21
22     cls();
23     setMonitorColor(0xa5);
24
25     char Welcome[] = "Welcome To OS0 : Copyright 2021\n";
26     char Welcome2[] = "Command Line Version 1.0.0.0\n\n";
27     char OSM[] = "OS0 > ";
28
29     printString(Welcome);
30     printString(Welcome2);
31     printColorString(OSM , 0xa8);
32 }
33
34
35 void cls(){
36     int i = 0;
37     CELL = 0;
38     while(i < (2 * 80 * 25)){
39         *(TM_START + i) = ' ' ; // Clear screen
40         i += 2;
41     }
42 }
43
44 void setMonitorColor(char Color){
45     int i = 1;
46     while(i < (2 * 80 * 25)){
47         *(TM_START + i) = Color;
48         i += 2;
49     }
50 }
51
52 void printString(char* cA){
53     int i = 0;
54     while(*(cA + i) != '\0'){
55         printChar(*(cA + i));
56         i++;
57     }
58 }
```

```
57         }
58     }
59
60     void printChar(char c){
61         if(CELL == 2 * 80 * 25)
62             scroll();
63         if(c == '\n'){
64             CELL = ((CELL + 160) - (CELL % 160));
65             return;
66         }
67         *(TM_START + CELL) = c;
68         CELL += 2;
69     }
70
71     void scroll(){
72         int i = 160 , y = 0;
73         while(i < 2 * 80 * 25){
74             *(TM_START + y) = *(TM_START + i);
75             i += 2;
76             y += 2;
77         }
78         CELL = 2 * 80 * 24;
79         i = 0;
80         while(i < 160){
81             *(TM_START + CELL + i) = ' ';
82             i += 2;
83         }
84     }
85
86     void printColorString(char* c , char co){
87         int i = 0;
88         while(*(c + i) != '\0'){
89             printColorChar(*(c + i) , co);
90             i++;
91         }
92     }
93
94     void printColorChar(char c , char co){
95         if(CELL == 2 * 80 * 25)
96             scroll();
97         if(c == '\n'){
98             CELL = ((CELL + 160) - (CELL % 160));
99             return;
```

```
100      }
101      *(TM_START + CELL) = c;
102      *(TM_START + CELL + 1) = co;
103      CELL += 2;
104  }
105
106 void getDecAscii(int num){
107     if(num == 0){
108         NumberAscii[0] = '0';
109         return;
110     }
111     char NUM[10];
112     int i = 0 , j = 0;
113     while(num > 0){
114         NUM[i] = num % 10;
115         num /= 10;
116         i++;
117     }
118     i--;
119     while(i >= 0){
120         NumberAscii[j] = NUM[i];
121         i--;
122         j++;
123     }
124     NumberAscii[j] = 'J';
125     j = 0;
126     while(NumberAscii[j] != 'J'){
127         NumberAscii[j] = '0' + NumberAscii[j];
128         j++;
129     }
130     NumberAscii[j] = 0;
131 }
```

Here, you could call the `printString` function by passing it with an address of null terminated character array. This will print the string to the screen.

`printColorString` can be called with the address of null terminated character array as first argument and a byte representing background and foreground colour as second argument.

`printChar` can be called with the character to be printed as the first argument.

`printColorChar` can be called with the first argument as the character to be printed and a byte representing background and foreground color as second argument.

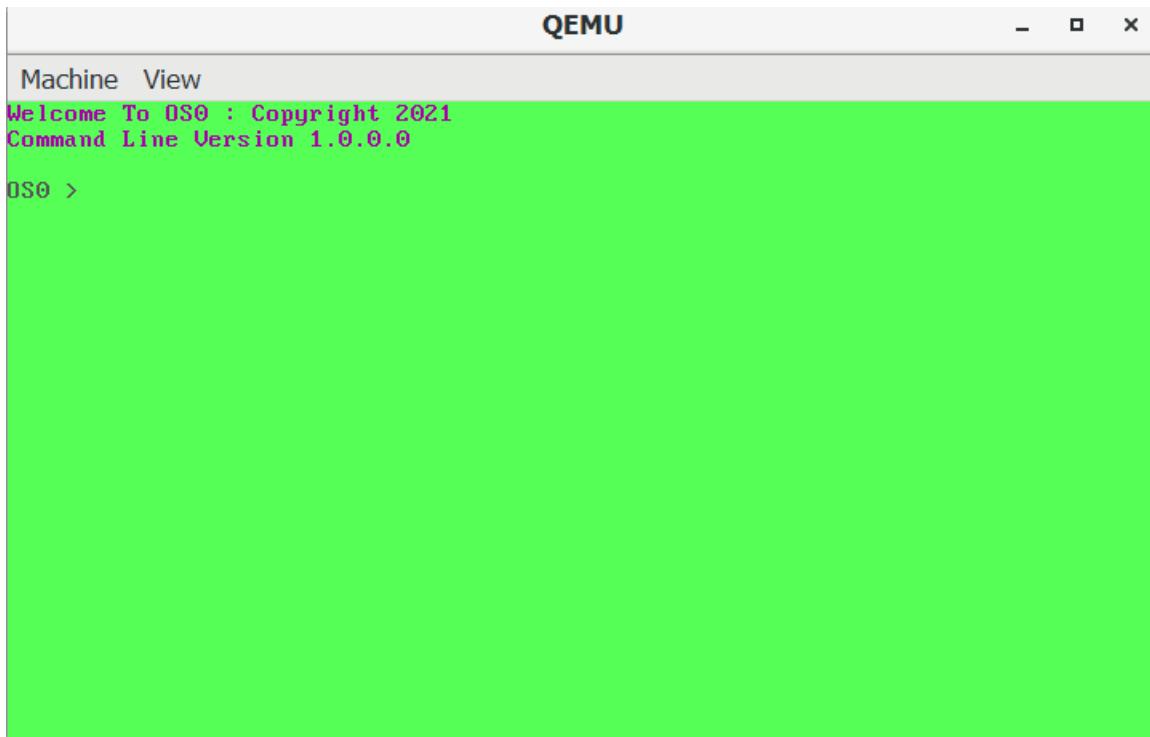
These functions will also take care of the scrolling activity by calling the `scroll()` function.

We do not need to call the `scroll()` function from our side. These things will be handled by the printing functions.

if you need to turn an integer variable to a string which could later be used to print using the `printString` function, You could use the `getDecAscii` function. Pass it with the integer to be turned to string and later refer to the array `NumberAscii` to print it. Call the `printString` function with `NumberAscii` as argument to print the number generated by `getDecAscii` function.

Try printing some coloured and non coloured string!!

Output of the Program we developed for this section:



Developing a Simple Video Player

In this section, We are not going to discuss or implement modern type video players, But we will cover the basic concept of video playing.

Theory

Videos are actually moving pictures. When we see multiple pictures during a small amount of time, Our brain takes it as moving or gives us a sense of The so called "Video".

What the computer do to present us with a video is by poking the video memory. Changing the data contained in the video memory continuously gives us this feel.

But the processor is very faster than our video card and the monitor. So it won't be able to display every frame the processor generate very fastly. So the solution is to minimize the frame updating procedure depending on the refresh rate of the monitor.

But the computer must also be fast enough to play about 60 frames during a second. Human eyes see upto 60 frames per second. So theres no point in generating video with frames more than 60 per second.

Let's make a program which gives us with a VIDEO feel.

Practical Implementation

```
1 int start(){
2     char* TM_START = (char*) 0xb8000;
3     int i;
4     char obj = 0;
5     while(1){
6         i = 0;
7         while(i < (2 * 80 * 25)){
8             *(TM_START + i) = obj;
9             i++;
10            obj++;
11        }
12    }
13 }
```

One Frame Of Output During Video Playing:



The program is in an infinite loop with `while(1)`.

Implementing Keyboard Driver

Introduction

Keyboard is the primary input device for a computer. Working with keyboard is a non avoidable factor in os development.

How does the computer know when a key is being pressed?

This is an interesting question. We could try to implement it by asking keyboard everytime if there is any input or not. But this is not a good idea.

Keyboard is very slower than the processor. Asking it everytime will affect the processing speed badly. And it also will put lot of traffic in the system bus.

We need to avoid that. So the solution implemented by the x86 chip manufacturer uses interrupts.

When ever a key is being pressed, The cpu will call a fuction that we pre define to it. The processor won't do the keyboard logic, but it will let us execute some code whenever a key is being pressed.

We need to make some code to handle the keyboard input, and pass its address to the interrupt descriptor table and say to the processor to load it.

After loading all the parameters and the location of our keyboard handling code, It will call that code whenever a key is being pressed.

When we get this interrupt, We could try reading from the keyboard which gives us the key being pressed.

What keyboard gives as the value for pressed key is not ascii. It is named as `scan codes`. We need to write our own program to convert it to ascii or other text encoding schemes.

Scan Codes

As said earlier, we need to know the values given by keyboard(scan codes) in order to convert it to ascii.

The following link lists the scan codes:

https://wiki.osdev.org/PS/2_Keyboard

You can also take this link as an alternate way to know more about keyboard working

Implementing Keyboard Driver

Modern keyboards use `usb` interface to communicate to the computer. This section covers working of `PS/2` keyboards. But `usb` keyboards will also work here as `usb` keyboard emulates the older `PS/2` keyboards.

The PIC Chip

`PIC` or `Programmable Interrupt Controller` is a chip in the computer whose main job is to generate interrupts. When a key in keyboard is pressed, The Chip inside keyboard tells to the `pic` chip inside our computer to generate a `#1` Interrupt. The `pic` chip will then decide the time to notify the `cpu` about the interrupt. When the `cpu` gets the message which says a key is being pressed, It executes a set of code which we told earlier to the `cpu` to execute when a key is pressed.

We can make use of the `in` assembly instruction to read the key being pressed.

It is very necessary to do this read as a failure in reading from keyboard will prevent the keyboard from giving further keyboard press events.

Practical Implementation

The full source code for this section could be downloaded from:

<https://github.com/TINU-2000/OS-DEV/tree/main/Keyboard>

Have a quick look at the code here:

```
main.c

1 #define PIC1_C 0x20
2 #define PIC1_D 0x21
3 #define PIC2_C 0xa0
4 #define PIC2_D 0xa1
5
6 #define ICW1_DEF 0x10
7 #define ICW1_ICW4 0x01
8 #define ICW4_x86 0x01
9
10 void cls();
11 void setMonitorColor(char);
12
13 void printString(char*);
14 void printChar(char);
15
16 void scroll();
```

```
17
18 void printColorString(char* , char);
19 void printColorChar(char , char);
20
21 void getDecAscii(int);
22
23 void initIDT();
24 extern void loadIdt();
25 extern void isr1_Handler();
26 void handleKeyPress(int);
27 void pressed(char);
28 void picRemap();
29
30 unsigned char inportb(unsigned short);
31 void outportb(unsigned short , unsigned char);
32
33 char* TM_START;
34 char NumberAscii[10];
35 int CELL;
36
37 struct IDT_ENTRY{
38     unsigned short base_Lower;
39     unsigned short selector;
40     unsigned char zero;
41     unsigned char flags;
42     unsigned short base_Higher;
43 };
44
45 struct IDT_ENTRY idt[256];
46 extern unsigned int isr1;
47 unsigned int base;
48
49 int start(){
50     TM_START = (char*) 0xb8000;
51     CELL = 0;
52     base = (unsigned int)&isr1;
53
54     cls();
55     setMonitorColor(0xa5);
56
57     char Welcome[] = "Welcome To OS0 : Copyright 2021\n";
58     char Welcome2[] = "Command Line Version 1.0.0.0\n\n";
59     char OSM[] = "OS0 > ";
```

```
60
61     printString(Welcome);
62     printString(Welcome2);
63     printColorString(OSM , 0xa8);
64
65     initIDT();
66 }
67
68
69 void cls(){
70     int i = 0;
71     CELL = 0;
72     while(i < (2 * 80 * 25)){
73         *(TM_START + i) = ' ' ; // Clear screen
74         i += 2;
75     }
76 }
77
78 void setMonitorColor(char Color){
79     int i = 1;
80     while(i < (2 * 80 * 25)){
81         *(TM_START + i) = Color;
82         i += 2;
83     }
84 }
85
86 void printString(char* cA){
87     int i = 0;
88     while(*(cA + i) != '\0'){
89         printChar(*(cA + i));
90         i++;
91     }
92 }
93
94 void printChar(char c){
95     if(CELL == 2 * 80 * 25)
96         scroll();
97     if(c == '\n'){
98         CELL = ((CELL + 160) - (CELL % 160));
99         return;
100    }
101    *(TM_START + CELL) = c;
102    CELL += 2;
```

```
103 }
104
105 void scroll(){
106     int i = 160 , y = 0;
107     while(i < 2 * 80 * 25){
108         *(TM_START + y) = *(TM_START + i);
109         i += 2;
110         y += 2;
111     }
112     CELL = 2 * 80 * 24;
113     i = 0;
114     while(i < 160){
115         *(TM_START + CELL + i) = ' ';
116         i += 2;
117     }
118 }
119
120 void printColorString(char* c , char co){
121     int i = 0;
122     while(*(c + i) != '\0'){
123         printColorChar(*(c + i) , co);
124         i++;
125     }
126 }
127
128 void printColorChar(char c , char co){
129     if(CELL == 2 * 80 * 25)
130         scroll();
131     if(c == '\n'){
132         CELL = ((CELL + 160) - (CELL % 160));
133         return;
134     }
135     *(TM_START + CELL) = c;
136     *(TM_START + CELL + 1) = co;
137     CELL += 2;
138 }
139
140 void getDecAscii(int num){
141     if(num == 0){
142         NumberAscii[0] = '0';
143         return;
144     }
145     char NUM[10];
```

```
146     int i = 0 , j = 0;
147     while(num > 0){
148         NUM[i] = num % 10;
149         num /= 10;
150         i++;
151     }
152     i--;
153     while(i >= 0){
154         NumberAscii[j] = NUM[i];
155         i--;
156         j++;
157     }
158     NumberAscii[j] = 'J';
159     j = 0;
160     while(NumberAscii[j] != 'J'){
161         NumberAscii[j] = '0' + NumberAscii[j];
162         j++;
163     }
164     NumberAscii[j] = 0;
165 }
166
167 void initIDT(){
168     idt[1].base_Lower = (base & 0xFFFF);
169     idt[1].base_Higher = (base >> 16) & 0xFFFF;
170     idt[1].selector = 0x08;
171     idt[1].zero = 0;
172     idt[1].flags = 0x8e;
173
174     picRemap();
175
176     outportb(0x21 , 0xfd);
177     outportb(0xa1 , 0xff);
178
179     loadIdt();
180 }
181
182 unsigned char inportb(unsigned short _port){
183     unsigned char rv;
184     __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
185     return rv;
186 }
187
188 void outportb(unsigned short _port, unsigned char _data){
```

```
189     __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
190 }
191
192 extern void isr1_Handler(){
193     handleKeypress(inportb(0x60));
194     outportb(0x20, 0x20);
195     outportb(0xa0, 0x20);
196 }
197
198 void handleKeypress(int code){
199     char Scancode[] = {
200         0, 0, '1', '2',
201         '3', '4', '5', '6',
202         '7', '8', '9', '0',
203         '-', '=', 0, 0, 'Q',
204         'W', 'E', 'R', 'T', 'Y',
205         'U', 'I', 'O', 'P', '[',
206         ']', 0, 'A', 'S', 'D', 'F', 'G',
207         'H', 'J', 'K', 'L', ';', '\'', ``,
208         0, '\\', 'Z', 'X', 'C', 'V', 'B', 'N', 'M',
209         ',', '.', '/', 0, '*', 0, ' '
210     };
211
212     if(code == 0x1c)
213         printChar('\n');
214     else if(code < 0x3a)
215         pressed(Scancode[code]);
216 }
217
218 void pressed(char key){
219     printChar(key);
220 }
221
222 void picRemap(){
223     unsigned char a, b;
224     a = inportb(PIC1_D);
225     b = inportb(PIC2_D);
226
227     outportb(PIC1_C, ICW1_DEF | ICW1_ICW4);
228     outportb(PIC2_C, ICW1_DEF | ICW1_ICW4);
229
230     outportb(PIC1_D, 0);
231     outportb(PIC2_D, 8);
```

```
232
233     outportb(PIC1_D , 4);
234     outportb(PIC2_D , 2);
235
236     outportb(PIC1_D , ICW4_x86);
237     outportb(PIC2_D , ICW4_x86);
238
239     outportb(PIC1_D , a);
240     outportb(PIC2_D , b);
241 }
```

Kernel_Entry.asm

```
1 START:
2 [bits 32]
3 [extern _start]
4     call _start
5     jmp $
```

IDT.asm

```
1 extern _idt
2 extern _isr1_Handler
3 global _isr1
4 global _loadIdt
5
6 idtDesc:
7     dw 2048
8     dd _idt
9
10 _isr1:
11     pushad
12     call _isr1_Handler
13     popad
14     iretd
15
16 _loadIdt:
17     lidt[idtDesc]
18     sti
19     ret
```

compile.bat

```

1 nasm Boot.asm -f bin -o bin\bootsect.bin
2 nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
3 nasm IDT.asm -f elf -o bin\IDT.bin
4
5 gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
6
7 ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o bin\IDT.bin
8
9 objcopy -O binary -j .text bin\kernel.img bin\kernel.bin
10 copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
11
12 qemu-system-i386 -drive format=raw,file=bin\os-image

```

Boot.asm

```

1 [org 0x7c00]
2 [bits 16]
3
4 ;Boot Loader
5 mov bx, 0x1000 ; Memory offset to which kernel will be loaded
6 mov ah, 0x02 ; Bios Read Sector Function
7 mov al, 30 ; No. of sectors to read (If your kernel won't fit into 30 sectors, \
8 you may need to provide the correct no. of sectors to read)
9 mov ch, 0x00 ; Select Cylinder 0 from harddisk
10 mov dh, 0x00 ; Select head 0 from hard disk
11 mov cl, 0x02 ; Start Reading from Second sector (Sector just after boot sector)
12
13 int 0x13 ; Bios Interrupt Relating to Disk functions
14
15
16 ;Switch To Protected Mode
17 cli ; Turns Interrupts off
18 lgdt [GDT_DESC] ; Loads Our GDT
19
20 mov eax, cr0
21 or eax, 0x1
22 mov cr0, eax ; Switch To Protected Mode
23
24 jmp CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
25 ;Forces the cpu to flush out contents in cache memory
26
27 [bits 32]
28

```

```
29 INIT_PM:  
30     mov ax, DATA_SEG  
31     mov ds, ax  
32     mov ss, ax  
33     mov es, ax  
34     mov fs, ax  
35     mov gs, ax  
36  
37     mov ebp, 0x90000  
38     mov esp, ebp ; Updates Stack Segment  
39  
40  
41     call 0x1000  
42     jmp $  
43  
44  
45  
46  
47  
48 GDT_BEGIN:  
49  
50 GDT_NULL_DESC: ;The Mandatory Null Descriptor  
51     dd 0x0  
52     dd 0x0  
53  
54 GDT_CODE_SEG:  
55     dw 0xffff ;Limit  
56     dw 0x0 ;Base  
57     db 0x0 ;Base  
58     db 10011010b ;Flags  
59     db 11001111b ;Flags  
60     db 0x0 ;Base  
61  
62 GDT_DATA_SEG:  
63     dw 0xffff ;Limit  
64     dw 0x0 ;Base  
65     db 0x0 ;Base  
66     db 10010010b ;Flags  
67     db 11001111b ;Flags  
68     db 0x0 ;Base  
69  
70 GDT_END:  
71
```

```

72 GDT_DESC:
73     dw GDT_END - GDT_BEGIN - 1
74     dd GDT_BEGIN
75
76 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
77 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
78
79
80 times 510-($-$$) db 0
81 dw 0xaa55

```

Also note to create a folder named `bin` in the same directory.

Lets look at how this works:

In the `main.c` file, in the `start()` function, Look at the last line: We called the `initIDT()` function.

The job of this function is to set the Interrupt Descriptor Table so that when a key is pressed, It will call the `isr1_Handler()` function.

Look at the very top of this file;

```

struct IDT_ENTRY{

    unsigned short base_Lower;
    unsigned short selector;
    unsigned char zero;
    unsigned char flags;
    unsigned short base_Higher;

};

struct IDT_ENTRY idt[256];
extern unsigned int isr1;
unsigned int base;

```

The `struct IDT_ENTRY` could be used to define the `idt`. There are a total of 256 possible idt entries. So we created 256 of them with `struct IDT_ENTRY idt[256];`

The next command `extern unsigned int isr1;` says to the compiler to implement compilation so that it knows that there will be a section named `isr1` in an outside source file. We could find this section in the `IDT.asm` file.

In the `start()` function in `main.c`, we copied the address of the `isr1` section to the variable `base` with `base = (unsigned int)&isr1;` (We used the & Symbol to get the address).

In the `initIDT()` function, we mapped the #1 interrupt(Keyboard interrupt(IRQ 1)) with:

```

idt[1].base_Lower = (base & 0xFFFF);
idt[1].base_Higher = (base >> 16) & 0xFFFF;
idt[1].selector = 0x08;
idt[1].zero = 0;
idt[1].flags = 0x8e;

```

Here we set the location of `isr1` section in the first two lines using the value in `base` variable.

The next three variables sets the arguments necessary for the `idt` entry.

Then we prepared the pic chip to handle interrupts and finally, we called the `loadIdt()` function. At the very top of the code, we said to the compiler that the `loadIdt()` function will be outside the c source file with `extern void loadIdt();`

When the `loadIdt()` code is executed, it jumps to that section we defined in the `IDT.asm` file.

`_loadIdt:`

```

lidt[idtDesc]
sti
ret

```

`lidt` is a x86 command used to make the processor load the Interrupt Descriptor Table. Here, it loads the `idt` entry Which the `idtDesc` section points to.

The command `sti` is used to enable the interrupts. From now onwards , we will get every interrupts.

From now onwards, the `_isr1` section in `IDT.asm` file will be called when ever a keyboard key is being pressed.

The `_isr1` section calls the `_isr1_Handler` function we defined in the `main.c` file.

When ever a key is pressed, the fuction `isr1_Handler()` will be called. We could implement what ever to do in that function.

According to our implementation we first obtained the scan code of key being pressed with `inportb(0x60)` and passed it to the `handleKeypress()` function with `handleKeypress(inportb(0x60));`

In the `handleKeypress()` function, we have an array of values named `Scancode[]`.

We could take the scan code as an index to point to the `Scancode[]` array.

The arrangement of values in `Scancode[]` gives us the ascii representation of the scan code.

Finally we called a function to print the character being pressed.

THE `inportb` AND `outportb` FUNCTIONS ARE FUNCTIONS THAT HELP US COMMUNICATE WITH EXTERNAL DEVICES. `in` COMMAND ACCEPTS INPUT FROM EXTERNAL DEVICES AND `out` COMMAND OUTPUTS COMMANDS AND DATA TO EXTERNAL DEVICES.

Compile and run the code by executing the `compile.bat` file and try pressing any character on keyboard, The program simply prints it to the screen.

External References

https://wiki.osdev.org/8259_PIC

<https://wiki.osdev.org/Interrupts>

<https://www.osdev.org/howtos/2/>

https://wiki.osdev.org/Interrupt_Descriptor_Table

Making Our First Prototype : OS0

Introduction

In this section, We will try to implement everything we have learned this far.

We will make a program which changes the colour of screen depending on the command it receives , play videos , print string etc.... This chapter just gives you an idea about practical implementation.

Developing the First Prototype

Use the following link to get the full source code for this section:

<https://github.com/TINU-2000/OS-DEV/tree/main/First%20Prototype>

main.c

```
1 #include "extra.h"
2
3 #define PIC1_C 0x20
4 #define PIC1_D 0x21
5 #define PIC2_C 0xa0
6 #define PIC2_D 0xa1
7
8 #define ICW1_DEF 0x10
9 #define ICW1_ICW4 0x01
10 #define ICW4_x86 0x01
11
12 void cls();
13 void setMonitorColor(char);
14
15 void printString(char*);
16 void printChar(char);
17
18 void scroll();
19
20 void printColorString(char* , char);
21 void printColorChar(char , char);
22
```

```
23 void getDecAscii(int);
24
25 void initIDT();
26 extern void loadIdt();
27 extern void isr1_Handler();
28 void handleKeypress(int);
29 void pressed(char);
30 void picRemap();
31
32 unsigned char inportb(unsigned short);
33 void outportb(unsigned short , unsigned char);
34
35
36 char NumberAscii[10];
37 int CELL;
38
39
40 char COMMAND[21];
41 int i = 0;
42
43 struct IDT_ENTRY{
44     unsigned short base_Lower;
45     unsigned short selector;
46     unsigned char zero;
47     unsigned char flags;
48     unsigned short base_Higher;
49 };
50
51 struct IDT_ENTRY idt[256];
52 extern unsigned int isr1;
53 unsigned int base;
54
55 int start(){
56     TM_START = (char*) 0xb8000;
57     CELL = 0;
58     base = (unsigned int)&isr1;
59
60     cls();
61     setMonitorColor(0xa5);
62
63     char Welcome[] = "Welcome To OS0 : Copyright 2021\n";
64     char Welcome2[] = "Command Line Version 1.0.0.0\n\n";
65     char OSM[] = "OS0 > ";
```

```
66
67     printString(Welcome);
68     printString(Welcome2);
69     printColorString(OSM , 0xa8);
70
71     initIDT();
72 }
73
74
75 void cls(){
76     int i = 0;
77     CELL = 0;
78     while(i < (2 * 80 * 25)){
79         *(TM_START + i) = ' ';// Clear screen
80         i += 2;
81     }
82 }
83
84 void setMonitorColor(char Color){
85     int i = 1;
86     while(i < (2 * 80 * 25)){
87         *(TM_START + i) = Color;
88         i += 2;
89     }
90 }
91
92 void printString(char* cA){
93     int i = 0;
94     while(*(cA + i) != '\0'){
95         printChar(*(cA + i));
96         i++;
97     }
98 }
99
100 void printChar(char c){
101     if(CELL == 2 * 80 * 25)
102         scroll();
103     if(c == '\n'){
104         CELL = ((CELL + 160) - (CELL % 160));
105         return;
106     }
107     *(TM_START + CELL) = c;
108     CELL += 2;
```

```
109 }
110
111 void scroll(){
112     int i = 160 , y = 0;
113     while(i < 2 * 80 * 25){
114         *(TM_START + y) = *(TM_START + i);
115         i += 2;
116         y += 2;
117     }
118     CELL = 2 * 80 * 24;
119     i = 0;
120     while(i < 160){
121         *(TM_START + CELL + i) = ' ';
122         i += 2;
123     }
124 }
125
126 void printColorString(char* c , char co){
127     int i = 0;
128     while(*(c + i) != '\0'){
129         printColorChar(*(c + i) , co);
130         i++;
131     }
132 }
133
134 void printColorChar(char c , char co){
135     if(CELL == 2 * 80 * 25)
136         scroll();
137     if(c == '\n'){
138         CELL = ((CELL + 160) - (CELL % 160));
139         return;
140     }
141     *(TM_START + CELL) = c;
142     *(TM_START + CELL + 1) = co;
143     CELL += 2;
144 }
145
146 void getDecAscii(int num){
147     if(num == 0){
148         NumberAscii[0] = '0';
149         return;
150     }
151     char NUM[10];
```

```

152     int i = 0 , j = 0;
153     while(num > 0){
154         NUM[i] = num % 10;
155         num /= 10;
156         i++;
157     }
158     i--;
159     while(i >= 0){
160         NumberAscii[j] = NUM[i];
161         i--;
162         j++;
163     }
164     NumberAscii[j] = 'J';
165     j = 0;
166     while(NumberAscii[j] != 'J'){
167         NumberAscii[j] = '0' + NumberAscii[j];
168         j++;
169     }
170     NumberAscii[j] = 0;
171 }
172
173 void initIDT(){
174     idt[1].base_Lower = (base & 0xFFFF);
175     idt[1].base_Higher = (base >> 16) & 0xFFFF;
176     idt[1].selector = 0x08;
177     idt[1].zero = 0;
178     idt[1].flags = 0x8e;
179
180     picRemap();
181
182     outportb(0x21 , 0xfd);
183     outportb(0xa1 , 0xff);
184
185     loadIdt();
186 }
187
188 unsigned char inportb(unsigned short _port){
189     unsigned char rv;
190     __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
191     return rv;
192 }
193
194 void outportb(unsigned short _port, unsigned char _data){

```

```

195     __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
196 }
197
198 extern void isr1_Handler(){
199     handleKeypress(inportb(0x60));
200     outportb(0x20, 0x20);
201     outportb(0xa0, 0x20);
202 }
203
204 void handleKeypress(int code){
205     char OSM[] = "\nOS0 > ";
206     char Scancode[] = {
207         0, 0, '1', '2',
208         '3', '4', '5', '6',
209         '7', '8', '9', '0',
210         '-', '=', 0, 0, 'Q',
211         'W', 'E', 'R', 'T', 'Y',
212         'U', 'I', 'O', 'P', '[' , ']' ,
213         0, 0, 'A', 'S', 'D', 'F', 'G',
214         'H', 'J', 'K', 'L', ';', '\'', ``',
215         0, '\\', 'Z', 'X', 'C', 'V', 'B', 'N', 'M',
216         ',', '.', '/', 0, '*', 0, ' '
217     };
218
219     if(code == 0x1c){
220         COMMAND[i] = '\0';
221         i = 0;
222         strEval(COMMAND);
223         printString(OSM);
224     }
225     else if(code < 0x3a)
226         pressed(Scancode[code]);
227 }
228
229 void pressed(char key){
230     if(i != 20){
231         COMMAND[i] = key;
232         i++;
233         printChar(key);
234     }
235     else{
236         blink();
237     }

```

```

238 }
239
240 void picRemap(){
241     unsigned char a , b;
242     a = inportb(PIC1_D);
243     b = inportb(PIC2_D);
244
245     outportb(PIC1_C , ICW1_DEF | ICW1_ICW4);
246     outportb(PIC2_C , ICW1_DEF | ICW1_ICW4);
247
248     outportb(PIC1_D , 0);
249     outportb(PIC2_D , 8);
250
251     outportb(PIC1_D , 4);
252     outportb(PIC2_D , 2);
253
254     outportb(PIC1_D , ICW4_x86);
255     outportb(PIC2_D , ICW4_x86);
256
257     outportb(PIC1_D , a);
258     outportb(PIC2_D , b);
259 }
```

extra.h

```

1 void setMonitorColor(char);
2 void cls();
3 void printString(char* );
4 void vid();
5
6 char* TM_START;
7
8 void blink(){
9     setMonitorColor(0x59);
10    int TIME_OUT = 0x10ffff;
11    while(--TIME_OUT);
12    setMonitorColor(0xa5);
13 }
14
15 char strcmp(char* sou , char* dest){
16     int i = 0;
17     while(*(sou + i) == *(dest + i)){
18         if(*(sou + i) == 0 && *(dest + i) == 0)
```

```
19             return 1;
20         i++;
21     }
22     return 0;
23 }
24
25 void strEval(char* CMD){
26     char cmd1[] = "CLS";
27     char cmd2[] = "COLORA";
28     char cmd3[] = "COLORB";
29     char cmd4[] = "COLORC";
30     char cmd5[] = "COLORDEF";
31     char cmd6[] = "VID";
32     char cmd7[] = "HI";
33
34     char msg1[] = "\nHELLO , HAVE A GOOD JOURNEY LEARNING\n";
35
36     if(strcmp(CMD , cmd1))
37         cls();
38
39     else if(strcmp(CMD , cmd2))
40         setMonitorColor(0x3c);
41
42     else if(strcmp(CMD , cmd3))
43         setMonitorColor(0x5a);
44
45     else if(strcmp(CMD , cmd4))
46         setMonitorColor(0x2a);
47
48     else if(strcmp(CMD , cmd5))
49         setMonitorColor(0xa5);
50
51     else if(strcmp(CMD , cmd6))
52         vid();
53     else if(strcmp(CMD , cmd7))
54         printString(msg1);
55 }
56
57 void vid(){
58     char clr = 'A';
59     while(1){
60         int i = 0;
61         while(i < 2 * 80 * 25){
```

```

62             *(TM_START + i) = clr;
63             clr++;
64             i++;
65         }
66     }
67 }
```

Explanation

We have implemented a small amount of application in this code. Here is the list:

After compiling and executing the code, We will be able to type to the screen. If the number of characters in a line becomes 20, Further key presses will generate a blinking in the screen. We did this by setting the colour of the screen for a small amount of time and changing it back to the orginal color.

hitting enter key will allow us to write to the next line.

Typing `HI` and hitting enter will display a pre defined text to the screen.

Typing `COLORA` , `COLORB` , `COLORC` and `COLORDEF` will change the color of the screen.

Typing `CLS` will clear the screen.

Typing `VID` will generate a video feeling look. Here you need to relaunch the executable to do further experiments. This is because as the computer is fully working on video playing, it will not have time to exit from it when we say to it for eg: when pressing a key.

We could easily solve this problem by using Threading, But currently, we are not at that topic.

The function `strEval()` in `extra.h` is the function which evaluates the commands and performs necessary operation.

We made the call to this function from the function named `handleKeypress()` in `main.c` inside an `if` condition :

```

if(code == 0x1c){
}
```

This is so that `strEval()` will only be called when the enter key is pressed.

TRY MAKING YOUR OWN IMPLEMENTATION OF THIS PROGRAM.

OR YOU CAN TRY MAKING SOME 2D GAME WHICH YOU CAN CONTROL WITH THE KEYBOARD.

YOU WILL BE VERY CONFIDENT IN THIS TOPIC AFTER YOU INITIATE YOUR OWN PROJECTS.

In later chapters, we will discuss about Creating pixel by pixel drawing. But This Text Mode graphics can also be used to implement a gui as we have the option to draw column based colors. Earlier computers even used text mode graphics capabilities to make a GUI.

Accessing Hard Disks

Introduction

Hard Disk or the Primary "Secondary Storage Device" is also one of the important part of a computer. Along with storage uses, it's abilities are also used to maximize the use of Primary storage device or RAM. Concepts such as paging make use of this.

When implementing a hard disk driver , we usually use the `in` and `out` assembly commands. Before developing a hard disk driver, we need to know the types of hard disks and how the it is organized.

Working With Hard Disks

Types of Hard Disk

The two main types of hard disk available today are `HDD` and `SSD`. `HDD` Stands for Hard Disk Drive and `SSD` stands for Solid State Drive.

HDD

Hard Disk Drive or `HDD` is a mechanical hard disk with a rotating platter and a moving head. Data is stored into it in magnetic form. The Platter(Where data is stored) , rotates during read or write operations and the Head(The part of hard disk which do the read/write operation) performs the desired operation on the platter.

Even though the RPM of modern hard disks are pretty huge, It is little slower when comparing with the speed of Main Memory or `RAM`, as every read or write operation needs the head to move to desired locations.

SSD

`SSD` or Solid State Drive is a non mechanical type of hard disk where the data is stored typically using flash memory technology. As it is non mechanical , `SSD`'s are much faster(But costly too). `SSD`'s will give us a faster boot time , load time and even increase the process execution speed. This is because the paging operation will be much faster. Paging is a concept used to utilize main memory to its fullest by putting "less often used" data temporarily to hard Hard disk and later copy it back to memory when needed.

How Hard Disk is Divided

Let's now learn how the HDD is organized to store data.

We said that the data is stored in magnetic form in platters. Typically hard disks have one to four platters stacked together. All of this platters will have separate Heads to read/write data.

All of this platters are further divided into tracks and sectors. There are number of tracks in a hard disk. All of this tracks will have a set of sectors in it. Usually the size of one sector is 512 bytes.

Another term associated with the hard disk is *cylinder*.

We already said that hard disks have number of platters and each platters are further divided into tracks. A cylinder is formed by joining the same tracks in all of the platters. For eg: Track 1 of Platter 1 + Track 1 of Platter 2 + Track 1 of Platter 3 Forms one cylinder.

Implementing a Hard Disk Driver

The full source code for this section is at:

<https://github.com/TINU-2000/OS-DEV/tree/main/Hard%20Disk%20-%20ATA>

ata.asm

```
1 [bits 32]
2
3 extern _blockAddr
4 extern _At
5 global _read
6 global _write
7
8 _read:
9
10         pushfd
11         and eax , 0x0FFFFFFF
12         push eax
13         push ebx
14         push ecx
15         push edx
16         push edi
17
18         mov eax , [_blockAddr]
19         mov cl , 1
20         mov edi , _At
21
22         mov ebx , eax
```

```
23
24      mov edx, 0x01F6
25      shr eax, 24
26      or al, 11100000b      ; Sets bit 6 in al for LBA mode
27      out dx, al
28
29      mov edx, 0x01F2      ; Port to send number of sectors
30      mov al, cl          ; Get number of sectors from CL
31      out dx, al
32
33      mov edx, 0x1F3      ; Port to send bit 0 - 7 of LBA
34      mov eax, ebx          ; Get LBA from EBX
35      out dx, al
36
37      mov edx, 0x1F4      ; Port to send bit 8 - 15 of LBA
38      mov eax, ebx          ; Get LBA from EBX
39      shr eax, 8          ; Get bit 8 - 15 in AL
40      out dx, al
41
42
43      mov edx, 0x1F5      ; Port to send bit 16 - 23 of LBA
44      mov eax, ebx          ; Get LBA from EBX
45      shr eax, 16          ; Get bit 16 - 23 in AL
46      out dx, al
47
48      mov edx, 0x1F7      ; Command port
49      mov al, 0x20          ; Read with retry.
50      out dx, al
51
52 .loop1:
53      in al, dx
54      test al, 8
55      jz .loop1
56
57      mov eax, 256          ; Read 256 words , 1 sector
58      xor bx, bx
59      mov bl, cl          ; read CL sectors
60      mul bx
61      mov ecx, eax
62      mov edx, 0x1F0
63      rep insw            ; copy to [RDI]
64
65      pop edi
```

```
66          pop edx
67          pop ecx
68          pop ebx
69          pop eax
70          popfd
71          ret
72
73
74 _write:
75          pushfd
76          and eax , 0x0FFFFFFF
77          push eax
78          push ebx
79          push ecx
80          push edx
81          push edi
82
83          mov eax , [_blockAddr]
84          mov cl , 1
85          mov edi , _At
86
87          mov ebx , eax
88
89          mov edx , 0x01F6
90          shr eax , 24
91          or al , 11100000b      ; Set bit 6 in al for LBA mode
92          out dx , al
93
94          mov edx , 0x01F2      ; Port to send number of sectors
95          mov al , cl          ; Get number of sectors from CL
96          out dx , al
97
98          mov edx , 0x1F3      ; Port to send bit 0 - 7 of LBA
99          mov eax , ebx          ; Get LBA from EBX
100         out dx , al
101
102         mov edx , 0x1F4      ; Port to send bit 8 - 15 of LBA
103         mov eax , ebx          ; Get LBA from EBX
104         shr eax , 8          ; Get bit 8 - 15 in AL
105         out dx , al
106
107
108         mov edx , 0x1F5      ; Port to send bit 16 - 23 of LBA
```

```
109          mov eax , ebx          ; Get LBA from EBX
110          shr eax , 16         ; Get bit 16 - 23 in AL
111          out dx , al
112
113          mov edx , 0x1F7        ; Command port
114          mov al , 0x30          ; Write with retry.
115          out dx , al
116
117 .loop2:
118          in al , dx
119          test al , 8
120          jz .loop2
121
122          mov eax , 256          ; Read 256 words , 1 sector
123          xor bx , bx
124          mov bl , cl            ; write CL sectors
125          mul bx
126          mov ecx , eax
127          mov edx , 0x1F0
128          mov esi , edi
129          rep outsw             ; go out
130
131          pop edi
132          pop edx
133          pop ecx
134          pop ebx
135          pop eax
136          popfd
137          ret
```

extra.h

```
1 void setMonitorColor(char);
2 void cls();
3 void printString(char*);
4 void vid();
5 void put();
6 void get();
7
8 extern void read();
9 extern void write();
10
11
```

```
12 int blockAddr;
13 char At[1024];
14
15 char* TM_START;
16
17 void blink(){
18     setMonitorColor(0x59);
19     int TIME_OUT = 0x10ffff;
20     while(--TIME_OUT);
21     setMonitorColor(0xa5);
22 }
23
24 char strcmp(char* sou , char* dest){
25     int i = 0;
26     while(*(sou + i) == *(dest + i)){
27         if(*(sou + i) == 0 && *(dest + i) == 0)
28             return 1;
29         i++;
30     }
31     return 0;
32 }
33
34 void strEval(char* CMD){
35     char cmd1[] = "CLS";
36     char cmd2[] = "COLORA";
37     char cmd3[] = "COLORB";
38     char cmd4[] = "COLORC";
39     char cmd5[] = "COLORDEF";
40     char cmd6[] = "VID";
41     char cmd7[] = "HI";
42     char cmd8[] = "PUT";
43     char cmd9[] = "GET";
44
45     char msg1[] = "\nHELLO , HAVE A GOOD JOURNEY LEARNING\n";
46
47     if(strcmp(CMD , cmd1))
48         cls();
49
50     else if(strcmp(CMD , cmd2))
51         setMonitorColor(0x3c);
52
53     else if(strcmp(CMD , cmd3))
54         setMonitorColor(0x5a);
```

```
55
56     else if(strcmp(CMD , cmd4))
57         setMonitorColor(0x2a);
58
59     else if(strcmp(CMD , cmd5))
60         setMonitorColor(0xa5);
61
62     else if(strcmp(CMD , cmd6))
63         vid();
64     else if(strcmp(CMD , cmd7))
65         printString(msg1);
66     else if(strcmp(CMD , cmd8)){
67         blockAddr = 0;
68         int i = 0;
69
70         while(i < 511){
71             At[i] = 'J'; // Fill with J
72             i++;
73         }
74         At[i] = 0; // Null character
75
76         put(); // Writes to Hard disk
77
78         i = 0;
79         while(i < 511){
80             At[i] = 0; // Clears the content
81             i++;
82         }
83     }
84     else if(strcmp(CMD , cmd9)){
85         blockAddr = 0;
86         get();
87         printString(At);
88     }
89
90 }
91
92 void vid(){
93     char clr = 'A';
94     while(1){
95         int i = 0;
96         while(i < 2 * 80 * 25){
97             *(TM_START + i) = clr;
```

```

98             clr++;
99             i++;
100        }
101    }
102 }
103
104 void put(){
105     write();
106 }
107
108 void get(){
109     read();
110 }
```

compile.bat

```

1 nasm Boot.asm -f bin -o bin\bootsect.bin
2 nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
3 nasm IDT.asm -f elf -o bin\IDT.bin
4 nasm ata.asm -f elf -o bin\ata.bin
5
6 gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
7
8 ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o bin\IDT.bin bin\
9 \ata.bin
10
11 objcopy -O binary -j .text bin\kernel.img bin\kernel.bin
12 copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
13
14 qemu-system-i386 -drive format=raw,file=bin\os-image
```

How it Works?

Here, we implemented a hard disk driver for the ATA technology. ATA or Advanced Technology Attachment allows hard disks and CD-ROMs to be internally connected to the motherboard and perform input/output functions.

There are different ways to read/write to hard disk. What we have used is LBA mode.

This is the easiest way to read/write to hard disk , all we need to do is pass the Block address of sector. Passing 0 will give us access to the first sector(Boot sector).

Please note not to write to the 0'th sector as this can make your computer non bootable, but you could always copy a boot loader to that sector.

Let's now understand the code:

The two new commands we added to the `strEval` function are `GET` and `PUT`.

When we type and enter the `PUT` command, it first copies the number `0` to `blockAddr` and then proceeds to initialize every cell in `At` character array with `'J'` and finally adds a null character. Then it calls the `put` function which calls another function named `write`.

The `write` function is defined in `ata.asm` file. In that function, look at the following section:

```
mov eax, [_blockAddr]
mov cl, 1
mov edi, _At
```

Here, the program copies the value we copied to the `blockAddr` variable in `strEval` function to the `eax` register. This value represents which sector to write to hard disk. Then we copied the value `1` to `cl` register, this represents the number of sectors to write. As we give `1` to that register, it writes `1` sector(512 bytes). After that, we copied the address of the array `At` we defined in `extra.h` file. This is so that the processor will write the data in that array to hard disk. The rest of the code communicates with the hard disk to write to it, and after the operation ends, it returns to our code in `C`.

The same thing happens when we give the `GET` command. But instead of the writing process, it does the reading process. In the code, we first copied the value `0` to `blockAddr` variable, Then we called the `get` function which calls the `read` function in `ata.asm` file. That function reads the `0`'th sector to the `At` array and returns to the code in `C`. and finally prints the content in that array.

Now, when running it, try giving the `GET` command first. We will get some random characters as output.

Now, try giving the `PUT` command and later call the `GET` command. This will print a lot of `J` to the screen.

This is because, when we give the `PUT` command, it copies the string of `J`'s to `At` array and writes it to the hard disk. After that, when we call the `GET` command, it reads from the hard disk and outputs the string of `J`'s we previously copied to it.

We done all of these operations to the `0`'th sector. We could change the value in `blockAddr` variable to read/write to other sectors.

Now, as we developed the Hard disk driver, lets create a File System so that we could do some file operations.

External References

<https://wiki.osdev.org/Category:ATA>
https://wiki.osdev.org/ATAPIO_Mode

Creating a Simple File System

Introduction

A **File System** is a system which lets us use the secondary storage device to store files , edit , rename , delete and do all other file operations. An ideal file system should be able to use the capacity of hard disk to the most. Fat , ntfs , ext etc... are examples of file systems.

We need to work on a good implementation so that it enables all of the file operations. The file system should be ideal enough so that even a single byte shouldn't be unused.

This is technically impossible to make use of every byte, But what it mean is that We should be able to use our hard disk to the most

Here , we will discuss how to implement a simple file system.

File system implementation is an advanced concept as it involves use of Data Structures.

We will discuss about implementing a simple file system which will be able to only save and retrieve some data. It won't be able to delete , rename or do other complicated stuff.

This is because as this is a beginner level book , we don't want the readers to be overwhelmed with big details.

The Implementation

Formatting

We are going to build a file system which allows us to give names to a file and store upto 512 byte data in that file.

Heres how we are going to implement it:

After our os is launched , before trying to create a file , The user needs to format the hard disk. This is usually done when the user enters the command **FORMAT** and presses enter key.

The formatting operation first stores four random bytes(Which the developer can choose) in the beginning of sector 0.

Then , the developer needs to code so that the remaining bytes in sector 0 and the rest of sectors initialize with 0. This is because , that we don't want the default data in hard disk to combine with the data we will store in it in the future. So it's a good idea to clear it. This is not so necessary but you may end up with problems depending upon your implementation.

Please note that the sector 0 is always used to store the boot loader. Writing other data to sector 0 will make the system un-bootable. If you are storing and booting the OS from real hardware, please choose other sector.

File Allocation Table And Storage Space

Create

The Sector 0 Where we stored the random bytes will be called as the file allocation table here. This sector could be used to store names of file that we will create in the future.

Now we will enable the users to type a command to create a file (You could choose any name for the command) and after obtaining the name for the file we do the following operations:

1. We check the first four bytes stored in sector 0 and if it is same as what we stored in the formatting section, we do the next operation.

We do this check to see if we have previously formatted the disk. If the check fails, say to the user to enter the command to format it.

2. Now we store the name of file in sector 0 just after the name of last file we created. Name of the very first file can be stored just after the first four bytes.

Please note that you need to manage the implementation so that we could differentiate between different files

Save

To save some data to a file, the user first needs to create a file using the command you chose at the previous section. After creating the file, the user could attempt the SAVE operation we will discuss here.

Lets use the command SAVE to do the save operation. What the save operation does here is copying a specified string of text to hard disk.

After the user enters the save command, we will also obtain a file name and a string of text to save to hard disk. Now we will do a check to see whether the user previously created that file by looking to the 0'th sector and if it succeeds, we do the following operation

1. We obtains the index of the file specified. We will get 1 as index if the file specified is the first file defined at sector 0. Like that, we will get 2 as index if the file specified is the second file defined at sector 0.
2. Now we take this index as the sector count where the string should be saved to.

if the index is 1 , we save the string of text to sector 1 , and if the index is 2 , we save the string to sector 2. and this process goes on.

Please note that we will only be able to store string of upto 512 bytes as the size of a sector is always 512 bytes. Storing a large file needs different implementation.

I assume you already know how to access the sectors in hard disk. Heres how to do it (Assuming you already implemented the hard disk driver said in the previous chapter):

1. Store the index of sector to read/write to the variable `blockAddr` defined at `extra.h`.
Copying 0 will give access to first sector(Boot sector) copying 1 will give access to 2'nd sector , copying 2 will give access to 3'rd sector.....
2. Now to do the save operation , call the `put()` function and to retrieve the content from hard disk , call the `get()` function in `extra.h` header file.

Retrieve

To read the data in saved file , We will do things similar to what we done at Save Section. We first obtains the name of file , then searches the sector 0 to get index of that file and use that index to point to the sector and finally do the read operation.

Conclusion

The file system we discussed here is actually worth nothing. But i think it gives some ideas about how it should be done. An ideal file system should always support creating file of variable size , rename , delete , edit and other advanced things such as using algorithms to find best space to store a file. Finding best place to store a file is so crucial so that we could use the hard disk to the most.

Every file allocation methods have it's own pros and cons. But it's our responsibility to implement a system which best suit for our needs.

External References

https://wiki.osdev.org/File_Systems

<https://wiki.osdev.org/FAT>

Graphics Mode GUI Creation

Introduction

We this far have used Text Mode graphics to output to the screen. Even if it allows to print colours , we are not able to do it in a pixel by pixel manner. After switching to graphics mode , we will be able to draw pixel. We will now choose the screen resolution we want and do the video operations.

The Drawback of switching to Graphics mode is that , we need to design special programs to render text to the screen. We should first create a font and use it to render to screen whenever we want. Most Hobbyist operating systems use Text Mode graphics as it is the best solution to learn it. But in order to get the most of the systems abilities such as displaying icons , pictures etc... needs this switch.

Not all modes allow rendering full colours , we will choose from a list of modes which have different abilities.

Drawing In Graphics Mode

Modes

Choosing A Mode

When talking about switching to Graphics mode , we have different video modes to choose from. These different modes have its own abilities with its own supported resolution and colours.

The following table shows the supported modes and its representation hex value:

This is taken from:

https://wiki.osdev.org/Drawing_In_Protected_Mode

```
00 text 40*25 16 color (mono)
01 text 40*25 16 color
02 text 80*25 16 color (mono)
03 text 80*25 16 color
04 CGA 320*200 4 color
05 CGA 320*200 4 color (m)
06 CGA 640*200 2 color
07 MDA monochrome text 80*25
08 PCjr
```

```

09 PCjr
0A PCjr
0B reserved
0C reserved
0D EGA 320*200 16 color
0E EGA 640*200 16 color
0F EGA 640*350 mono
10 EGA 640*350 16 color
11 VGA 640*480 mono
12 VGA 640*480 16 color
13 VGA 320*200 256 color

```

We need to choose one from this list and note its associated hex value.

For now we will take **VGA 320*200 256 color** so its representation number hex value is 13.

Making Switch To The Selected Mode

We will now switch the video mode from our current Text Mode to Graphics Mode.

Switching to graphics mode is done using bios. But as we are in 32 bit Protected Mode , we won't be able to use bios. Bios is only available in 16 bit mode. So we will leave our current source codes for now and start a new project from scratch to show the development in graphics mode. After we switch to Graphics mode , we could switch to 32 bit protected mode.

Please note that we will use our current source codes in later chapters. But for this chapter , we will create a sample from scratch.

The following assembly statements switches to **VGA 320*200 256 color** Mode:

```

mov ah , 0x00
mov al , 0x13
int 0x10

```

Here , we first copied the value `0x00` to the `ah` register. This value represents the option in bios to switch video modes.

We then copied the value `0x13` to the `al` register, And finally called the bios with `int 0x10`.

Bios will now switch to **VGA 320*200 256 color** mode which we represented with `0x13`

We Could place this assembly statements in `Boot.asm` file just after the the Boot loader code.

We wont be able to switch to graphics mode in 32 bit protected mode. So the best place to put code to switch to Graphics mode in our case is just after the Boot Loader in `Boot.asm` file.

Please obtain the source code from:

<https://github.com/TINU-2000/OS-DEV/tree/main/Graphics%20Mode0>

`Boot.asm`

```
1 [org 0x7c00]
2 [bits 16]
3
4 ;Boot Loader
5 mov bx , 0x1000 ; Memory offset to which kernel will be loaded
6 mov ah , 0x02 ; Bios Read Sector Function
7 mov al , 30 ; No. of sectors to read(If your kernel won't fit into 30 sectors , \
8 you may need to provide the correct no. of sectors to read)
9 mov ch , 0x00 ; Select Cylinder 0 from harddisk
10 mov dh , 0x00 ; Select head 0 from hard disk
11 mov cl , 0x02 ; Start Reading from Second sector(Sector just after boot sector)
12
13 int 0x13 ; Bios Interrupt Relating to Disk functions
14
15 ;Switch to Graphics Mode
16 mov ah , 0x00
17 mov al , 0x13
18 int 0x10
19
20 ;Switch To Protected Mode
21 cli ; Turns Interrupts off
22 lgdt [GDT_DESC] ; Loads Our GDT
23
24 mov eax , cr0
25 or eax , 0x1
26 mov cr0 , eax ; Switch To Protected Mode
27
28 jmp CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
29 ;Forces the cpu to flush out contents in cache memory
30
31 [bits 32]
32
33 INIT_PM:
34 mov ax , DATA_SEG
35 mov ds , ax
36 mov ss , ax
37 mov es , ax
38 mov fs , ax
39 mov gs , ax
40
41 mov ebp , 0x900000
42 mov esp , ebp ; Updates Stack Segment
43
```

```
44
45 call 0x1000
46 jmp $
47
48
49
50
51
52 GDT_BEGIN:
53
54 GDT_NULL_DESC: ;The Mandatory Null Descriptor
55     dd 0x0
56     dd 0x0
57
58 GDT_CODE_SEG:
59     dw 0xffff ;Limit
60     dw 0x0 ;Base
61     db 0x0 ;Base
62     db 10011010b ;Flags
63     db 11001111b ;Flags
64     db 0x0 ;Base
65
66 GDT_DATA_SEG:
67     dw 0xffff ;Limit
68     dw 0x0 ;Base
69     db 0x0 ;Base
70     db 10010010b ;Flags
71     db 11001111b ;Flags
72     db 0x0 ;Base
73
74 GDT_END:
75
76 GDT_DESC:
77     dw GDT_END - GDT_BEGIN - 1
78     dd GDT_BEGIN
79
80 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
81 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
82
83
84 times 510-($-$$) db 0
85 dw 0xaa55
```

Kernel_Entry.asm

```

1 START:
2 [bits 32]
3 [extern _start]
4     call _start
5     jmp $

```

main.c

```

1 int start(){
2
3 }

```

compile.bat

```

1 nasm Boot.asm -f bin -o bin\bootsect.bin
2 nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
3
4 gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
5
6 ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o
7
8 objcopy -O binary -j .text bin\kernel.img bin\kernel.bin
9 copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
10
11 qemu-system-i386 -drive format=raw,file=bin\os-image

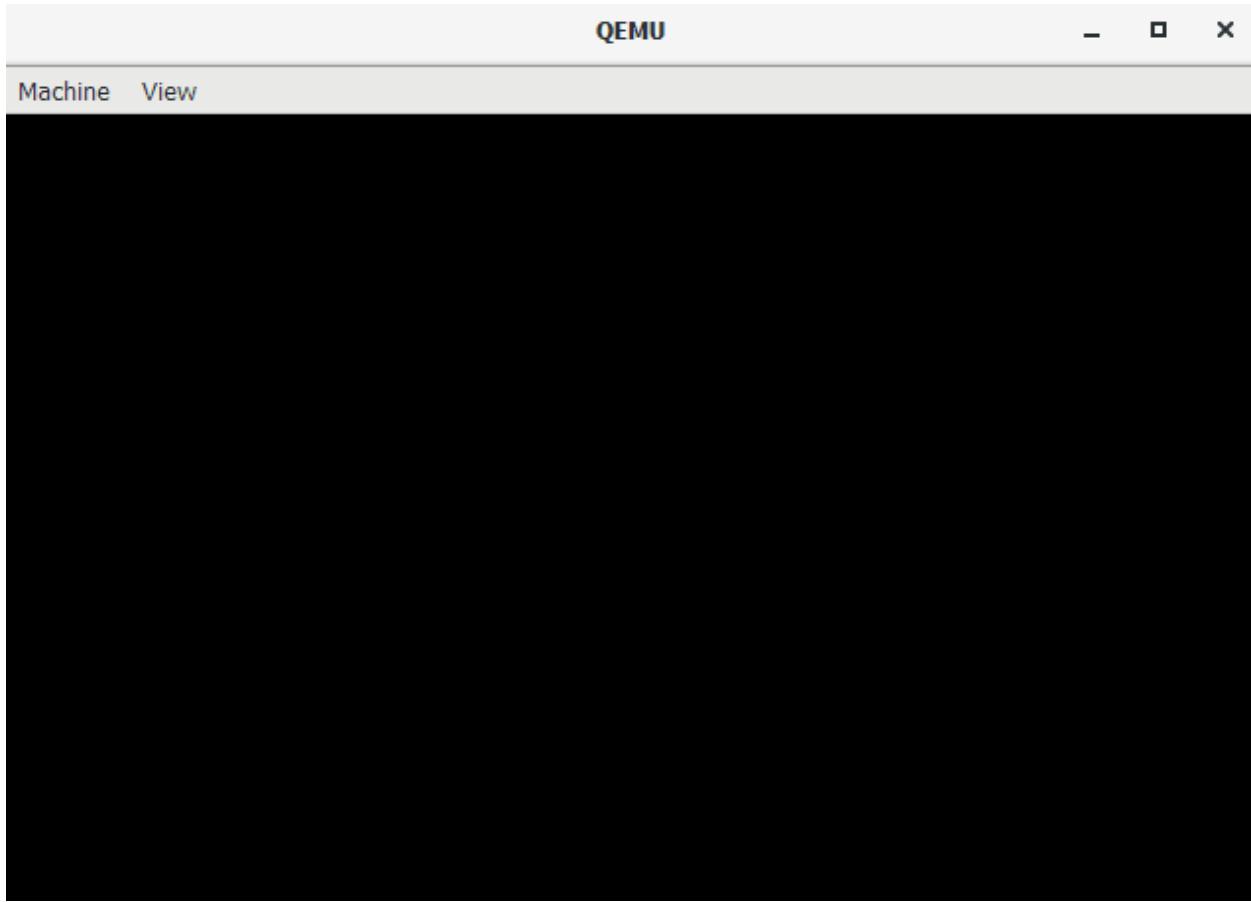
```

Please note to create a folder named `bin` in the same folder as the `compile.bat` file saves the compiled files to it.

Look at the `Boot.asm` file to see changes.

Compile and run the program and you will get a black screen with no content. This is the desired output at this stage. We will now see how to draw to that screen.

Please also note that the `start()` function in `main.c` will be blank, we will now create code to render to the screen in that function.



Video Memory and Drawing

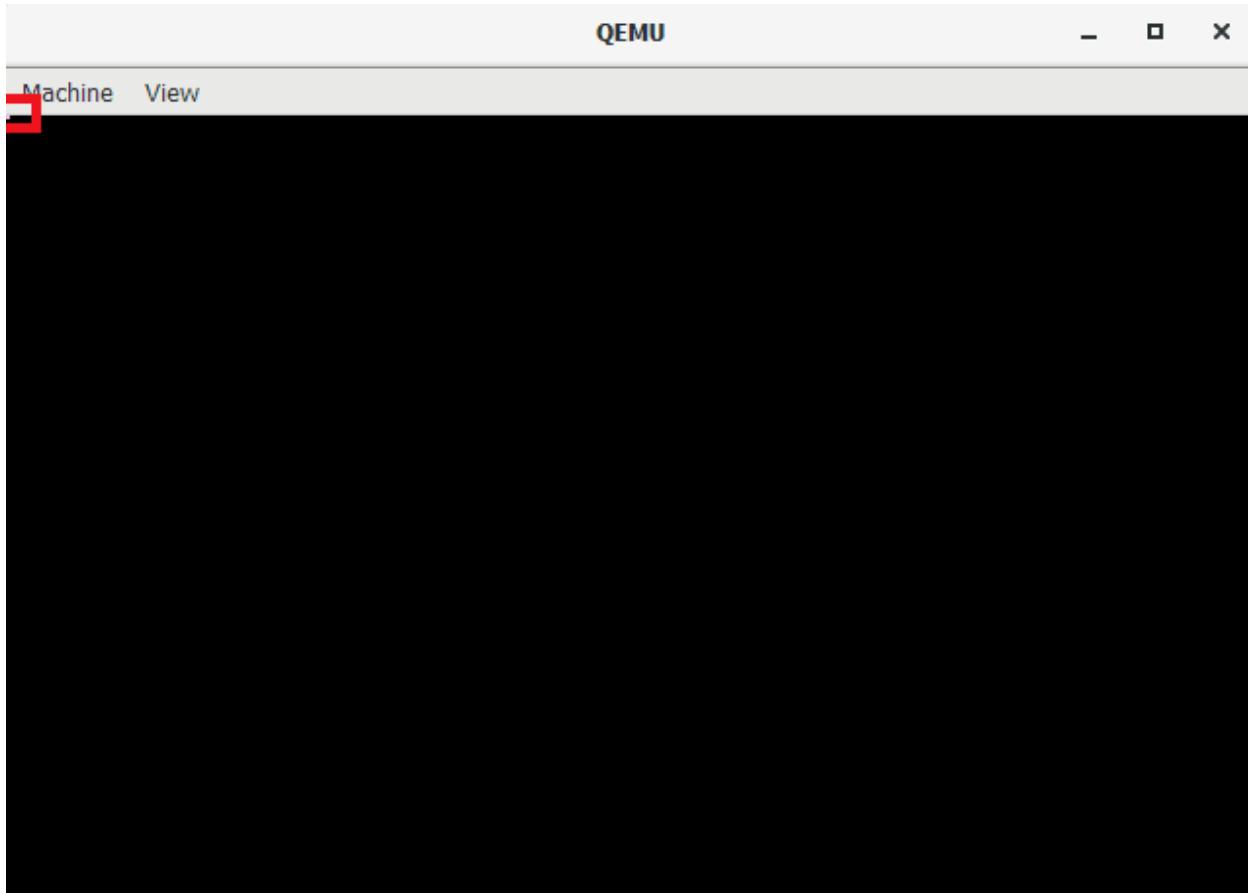
In Text Mode graphics , we wrote to a special location in ram to make things display to the screen. Graphics Mode Graphical operations also make use of memory to draw to the screen. But in this case , we will use `0xA0000` as address.

The following program renders one pixel to the Top Left of the screen by poking the very first video memory buffer byte:

`main.c`

```
1 int start(){
2     char* vbuff = (char*) 0xA0000;
3     *(vbuff) = 0x55;
4 }
```

Output



The pixel displayed here represents the hex value 0x55 which we copied to the video memory.

The supported values that can be copied to one cell is from 0x00 to 0xFF or 0 to 255

The rest of the pixels can be rendered by poking the rest of the video memory.

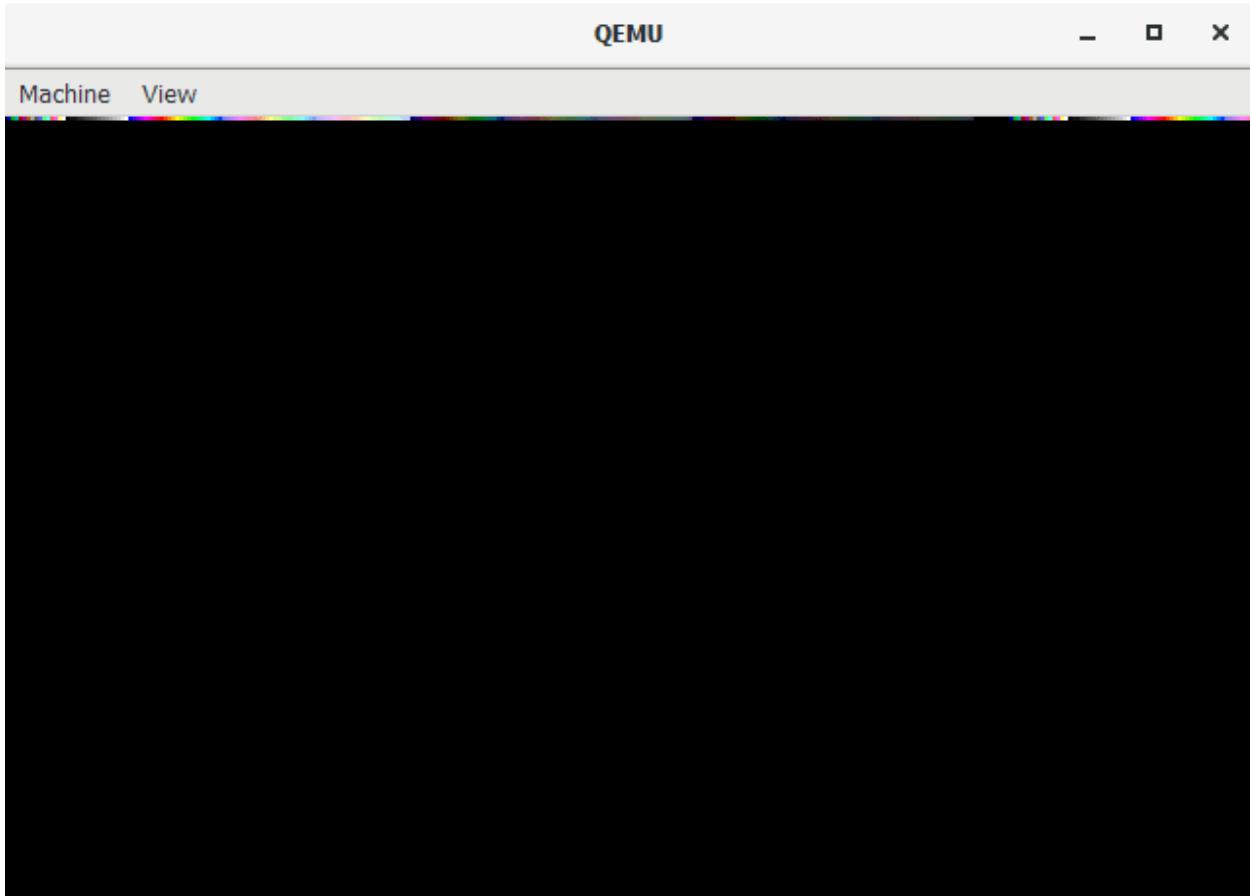
For eg: poking 0xA0000 + 1 renders the second pixel , poking 0xA0000 + 2 renders third pixel and this process goes on.

Look at the following program to poke every cell in first row:

main.c

```
1 int start(){
2     char* vbuff = (char*) 0xA0000;
3     char colr = 0x00;
4     int i = 0;
5     while(i < 320){
6         *(vbuff + i) = colr;
7         colr++;
8         i++;
9     }
10 }
```

Output

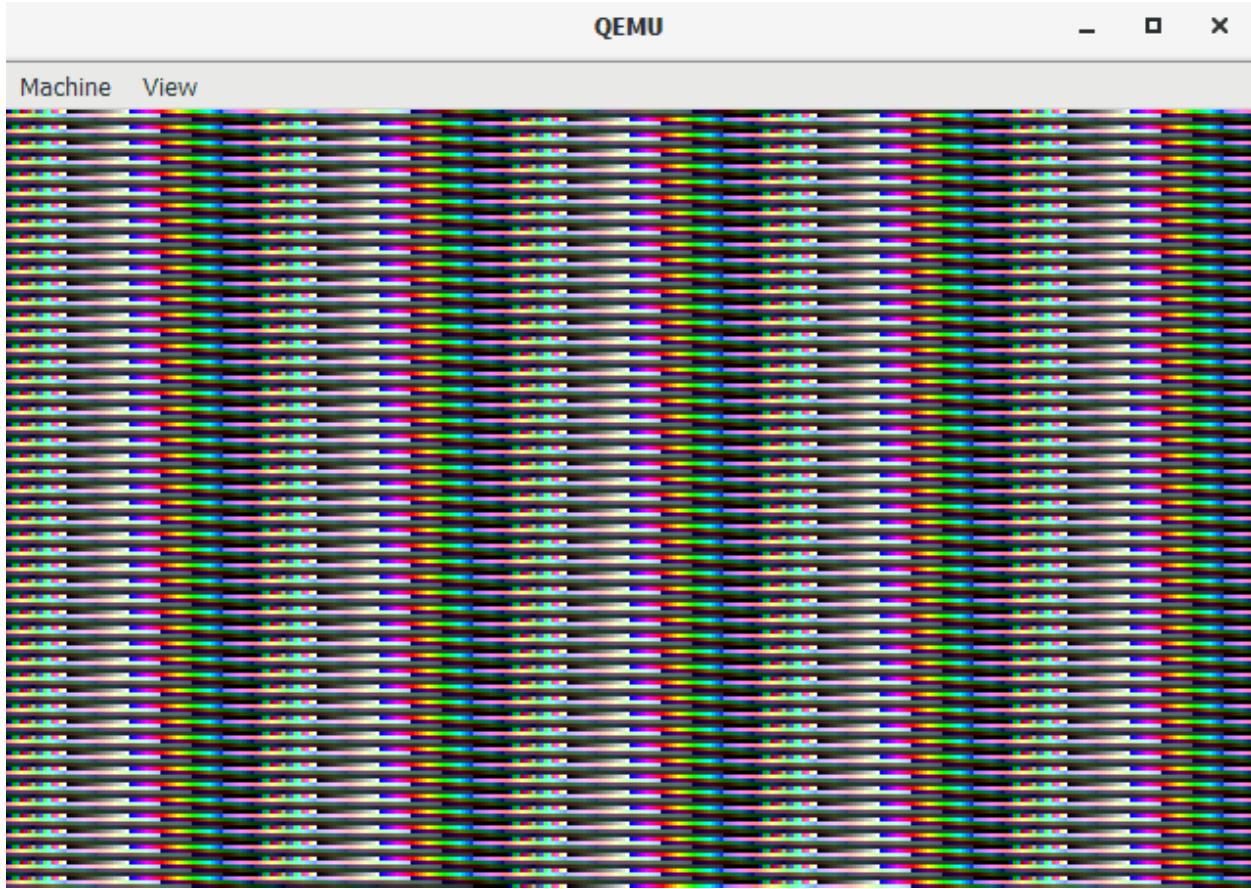


Now lets fill the screen:

main.c

```
1 int start(){
2     char* vbuff = (char*) 0xA0000;
3     char colr = 0x00;
4     int i = 0;
5     while(i < 320 * 200){
6         *(vbuff + i) = colr;
7         colr++;
8         i++;
9     }
10 }
```

Output



Sample User Interface Using Graphics Mode

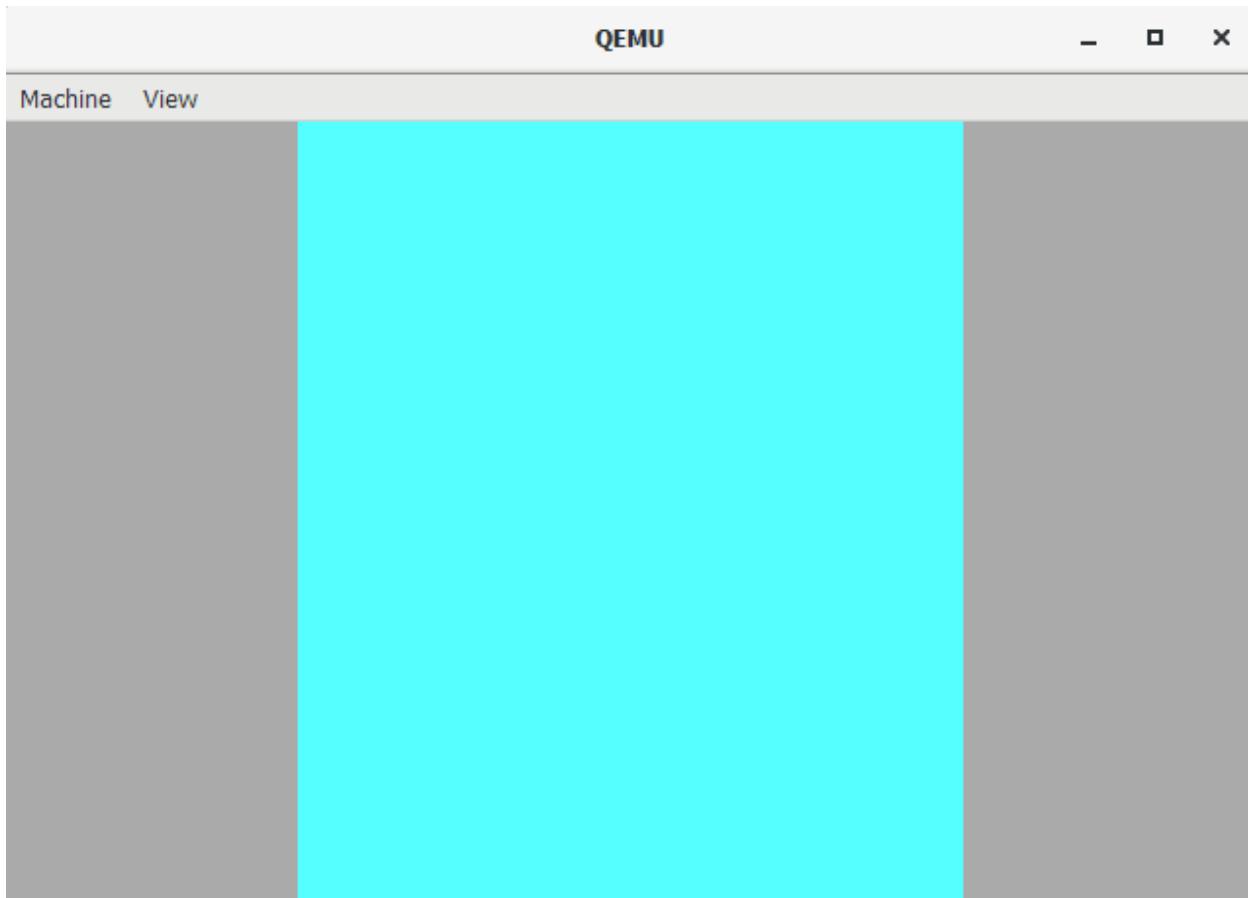
We have discussed about the base of creating a pixel by pixel drawing. Creating a good user interface needs deep knowledge in different colours and its values. Let me give a sample user interface , for that i will create a user interface looking like a text editor:

main.c

```
1 int start(){
2     char* vbuff = (char*) 0xA0000;
3     int y = 0;
4     while(y < 200){
5         int x = 0;
6         while(x < 75){
7             *(vbuff + (y * 320) + x) = 0x07;
8             x++;
9         }
10        while(x < 245){
11            *(vbuff + (y * 320) + x) = 0x0B;
```

```
12             x++;
13         }
14         while(x < 320){
15             *(vbuff + (y * 320) + x) = 0x07;
16             x++;
17         }
18         y++;
19     }
20 }
```

Output

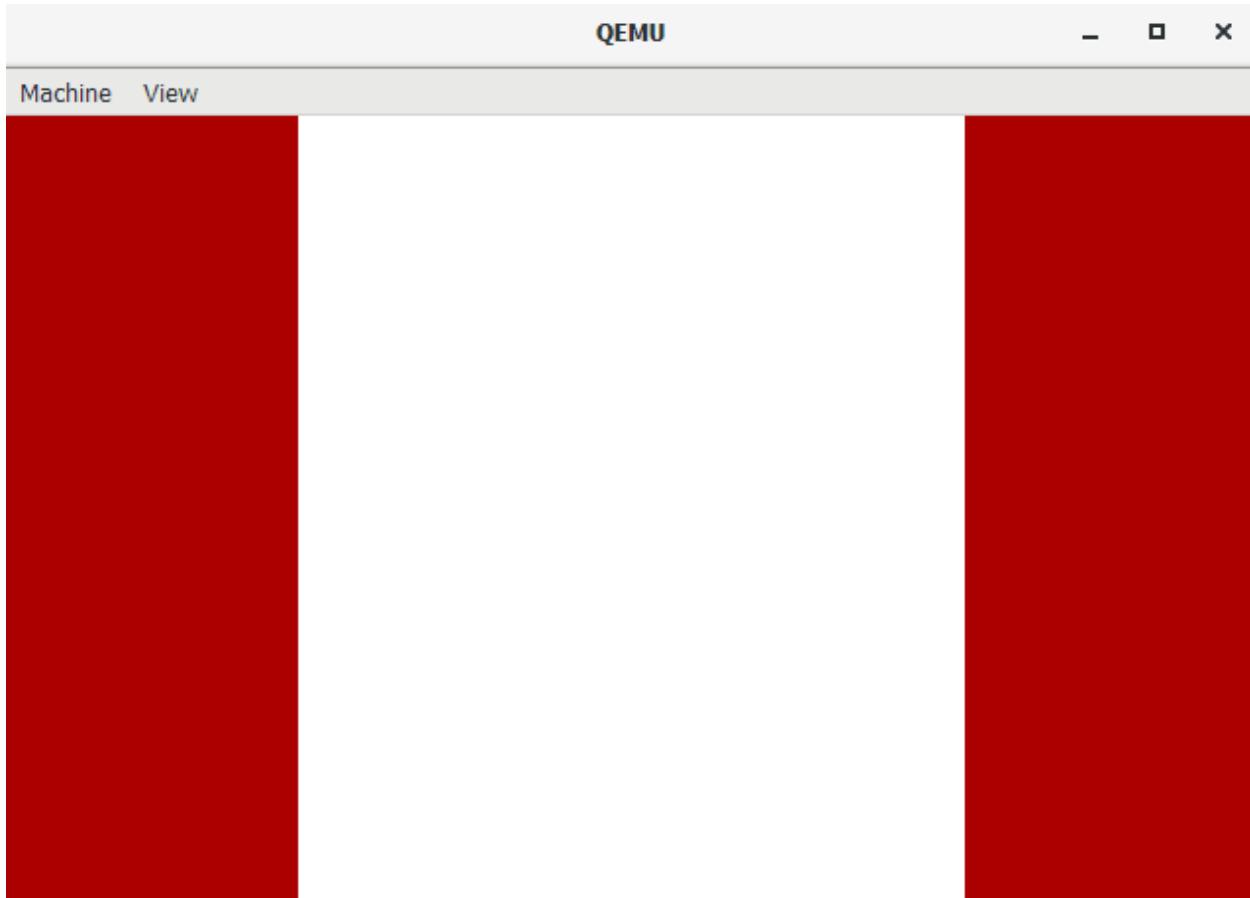


Lets see another one:

main.c

```
1 int start(){
2     char* vbuff = (char*) 0xA0000;
3     int y = 0;
4     while(y < 200){
5         int x = 0;
6         while(x < 75){
7             *(vbuff + (y * 320) + x) = 0x04;
8             x++;
9         }
10        while(x < 245){
11            *(vbuff + (y * 320) + x) = 0x0F;
12            x++;
13        }
14        while(x < 320){
15            *(vbuff + (y * 320) + x) = 0x04;
16            x++;
17        }
18        y++;
19    }
20 }
```

Output



Here, we could take the center part as the Text Editing field. This is just a sample, we won't be able to render text to the screen with any trick. We need to create our own font and make some program to render it to the screen. This may take some time, but it's a good idea to keep working on this mode and develop an amazing UI.

Also note to work with other video modes.

External References

https://wiki.osdev.org/Drawing_In_Protected_Mode

https://wiki.osdev.org/VGA_Resources

Implementing a Mouse Driver

Introduction

Mouse is very essential when working with a Graphical Operating system. Although most hobbyist operating systems do not use a mouse as they work on text modes. But we need to implement it for those who are trying for a graphical OS.

For that , We will implement a driver for PS2 mouse.

How The Mouse Work

IRQ12

A PS2 mouse generates IRQ12. Once we initialize a mouse , it sends 3 or 4 byte packets to communicate mouse movement, and mouse button press/release events.

We used IRQ1 to get data from the keyboard. likewise we will be able to get mouse input by defining IRQ12

We could give an address of a function we create so that it will be called during every mouse input.

The Mouse Events And Packets

The first 3 bytes which the mouse give always have the same format. The second byte which it generate signifies movement in X axis and the third byte signifies mouse movement in Y axis.

When talking about the first byte , it is a combination of 8 bits where every bit signifies different things.

The 6th bit of first byte is set if the middle button is clicked. 7th bit represents the click of Right button and 8th bit represents the click of Left button.

You can use the bit shift and logical and , or operators in c to evaluate each of this bits.

Double Clicks

The PS2 mouse won't give you special data to represent double clicks. The best way to check if a double click occur is by looking at the time difference between first and second click. If the time difference is less(According to your need) , you could treat it as a double click.

Practical Implementation

Please obtain the full source code from:

<https://github.com/TINU-2000/OS-DEV/tree/main/Mouse>

```
1 unsigned char inportb(unsigned short _port){
2     unsigned char rv;
3     __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
4     return rv;
5 }
6
7 void outportb(unsigned short _port, unsigned char _data){
8     __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
9 }
10
11 void irq_install_handler(){
12
13     // Initialize IRQ12 in IDT and load it
14
15 }
16
17 void mouse_wait(unsigned char a_type){
18     int _time_out=100000;
19     if(a_type==0){
20         while(_time_out--){
21             if((inportb(0x64) & 1) == 1){
22                 return;
23             }
24         }
25         return;
26     }
27     else{
28         while(_time_out--){
29             if((inportb(0x64) & 2) == 0){
30                 return;
31             }
32         }
33         return;
34     }
35 }
```

```
36
37 void ProcessMouse(){
38
39 }
40
41 void mouse_write(unsigned char a_write){
42     mouse_wait(1);
43     outportb(0x64, 0xD4);
44
45     mouse_wait(1);
46     outportb(0x60, a_write);
47 }
48
49 unsigned char mouse_read(){
50     mouse_wait(0);
51     return inportb(0x60);
52 }
53
54 void initMouse(){
55     unsigned char status;
56     mouse_wait(1);
57     outportb(0x64, 0xA8);
58
59     mouse_wait(1);
60     outportb(0x64, 0x20);
62
63     mouse_wait(0);
64     status = (inportb(0x60) | 2);
65     mouse_wait(1);
66     outportb(0x64, 0x60);
67     mouse_wait(1);
68     outportb(0x60, status);
69
70     // Use default settings
71     mouse_write(0xF6);
72     mouse_read();
73
74     mouse_write(0xF4);
75     mouse_read();
76
77     irq_install_handler(12 , &ProcessMouse);
78 }
```

```
79
80 int start(){
81     initMouse();
82 }
```

Here , the `start()` function calls a function to initialize the mouse , and in that initialize function , We communicated with the mouse to make it start working considering the PS2 specifications. At the end of that function , we called the `irq_install_handler` function which initializes IRQ12 and load it. This function will be same as the one we used to initialize the keyboard and the only difference is the IRQ number.

After all of this is complete , whenever the mouse generates an interrupt , the cpu will start executing the `ProcessMouse()` function.

We will be able to use the `inportb` function to read from mouse and do the operation we want. So it is left to you for your own implementation!!!

External References

https://wiki.osdev.org/Mouse_Input
<https://forum.osdev.org/viewtopic.php?t=10247>

Audio

Introduction

Generating audio is an interesting part when doing your project. We actually give some integer value(Representing the frequency) which will turn to an audible effect.

We directly communicate with the sound card using out command and the sound card work on further stuff.

Let's directly start working on it!!

Generating Sound : First Try

Let's Generate a small beep :

main.c

```
1 unsigned char inportb(unsigned short _port){
2     unsigned char rv;
3     __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
4     return rv;
5 }
6
7 void outportb(unsigned short _port, unsigned char _data){
8     __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
9 }
10
11 void play(unsigned int fr) {
12     unsigned int Div;
13     unsigned char tmp;
14
15     Div = 1193180 / fr;
16     outportb(0x43, 0xb6);
17     outportb(0x42, (unsigned char) (Div) );
18     outportb(0x42, (unsigned char) (Div >> 8));
19
20     tmp = inportb(0x61);
21     if (tmp != (tmp | 3)) {
```

```

22             outportb(0x61, tmp | 3);
23         }
24     }
25
26     void stop() {
27         unsigned char fr = inportb(0x61) & 0xFC;
28         outportb(0x61, fr);
29     }
30
31     void scream(int fr) {
32         play(fr);
33         int Delay = 10000;
34         while(Delay--);
35         stop();
36     }
37
38     int start(){
39         scream(2020);
40     }

```

Before compiling , add the following argument when calling qemu in compile.bat:

-soundhw pcspk

compile.bat

```

1 nasm Boot.asm -f bin -o bin\bootsect.bin
2 nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
3
4 gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
5
6 ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o
7
8 objcopy -O binary -j .text bin\kernel.img bin\kernel.bin
9 copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
10
11 qemu-system-i386 -soundhw pcspk -drive format=raw,file=bin\os-image

```

Here , we called the `scream` function with `scream(2020)`. We could try giving different value as an argument to the `scream` function to generate different sound.

Generating Sound : Second Try | Integrating With OS0

Please obtain the full source code for this section from:

<https://github.com/TINU-2000/OS-DEV/tree/main/Audio>

Let's include the sound generating program in our previous code. This lets us generate sound when entering a command.

This time we will feel it like a music , lets see:

extra.h

```
1 void setMonitorColor(char);
2 void cls();
3 void printString(char* );
4 void vid();
5 void put();
6 void get();
7
8 void scream(int);
9
10 extern void read();
11 extern void write();
12
13 unsigned char inportb(unsigned short);
14 void outportb(unsigned short , unsigned char);
15
16
17
18 int blockAddr;
19 char At[1024];
20
21 char* TM_START;
22
23 void blink(){
24     setMonitorColor(0x59);
25     int TIME_OUT = 0x10ffff;
26     while(--TIME_OUT);
27     setMonitorColor(0xa5);
28 }
29
30 char strcmp(char* sou , char* dest){
```

```
31     int i = 0;
32     while(*sou + i) == *(dest + i)){
33         if(*sou + i) == 0 && *(dest + i) == 0)
34             return 1;
35         i++;
36     }
37     return 0;
38 }
39
40 void strEval(char* CMD){
41     char cmd1[] = "CLS";
42     char cmd2[] = "COLORA";
43     char cmd3[] = "COLORB";
44     char cmd4[] = "COLORC";
45     char cmd5[] = "COLORDEF";
46     char cmd6[] = "VID";
47     char cmd7[] = "HI";
48     char cmd8[] = "PUT";
49     char cmd9[] = "GET";
50     char cmd10[] = "PLAY";
51
52     char msg1[] = "\nHELLO , HAVE A GOOD JOURNEY LEARNING\n";
53
54     if(strcmp(CMD , cmd1))
55         cls();
56
57     else if(strcmp(CMD , cmd2))
58         setMonitorColor(0x3c);
59
60     else if(strcmp(CMD , cmd3))
61         setMonitorColor(0x5a);
62
63     else if(strcmp(CMD , cmd4))
64         setMonitorColor(0x2a);
65
66     else if(strcmp(CMD , cmd5))
67         setMonitorColor(0xa5);
68
69     else if(strcmp(CMD , cmd6))
70         vid();
71     else if(strcmp(CMD , cmd7))
72         printString(msg1);
73     else if(strcmp(CMD , cmd8)){
```

```
74         blockAddr = 0;
75         int i = 0;
76
77         while(i < 511){
78             At[i] = 'J'; // Fill with J
79             i++;
80         }
81         At[i] = 0; // Null character
82
83         put(); // Writes to Hard disk
84
85         i = 0;
86         while(i < 511){
87             At[i] = 0; // Clears the content
88             i++;
89         }
90     }
91     else if(strcmp(CMD , cmd9)){
92         blockAddr = 0;
93         get();
94         printString(At);
95     }
96     else if(strcmp(CMD , cmd10)){
97         int stone = 15000;
98         while(stone--){
99             scream(stone);
100        }
101    }
102 }
103 }
104
105 void vid(){
106     char clr = 'A';
107     while(1){
108         int i = 0;
109         while(i < 2 * 80 * 25){
110             *(TM_START + i) = clr;
111             clr++;
112             i++;
113         }
114     }
115 }
116 }
```

```
117 void put(){
118     write();
119 }
120
121 void get(){
122     read();
123 }
124
125 void play(unsigned int fr) {
126     unsigned int Div;
127     unsigned char tmp;
128
129     Div = 1193180 / fr;
130     outportb(0x43, 0xb6);
131     outportb(0x42, (unsigned char) (Div) );
132     outportb(0x42, (unsigned char) (Div >> 8));
133
134     tmp = inportb(0x61);
135     if (tmp != (tmp | 3)) {
136         outportb(0x61, tmp | 3);
137     }
138 }
139
140 void stop() {
141     unsigned char fr = inportb(0x61) & 0xFC;
142     outportb(0x61, fr);
143 }
144
145 void scream(int fr) {
146     play(fr);
147     int stone = 0xffff;
148     while(stone--);
149     stop();
150 }
```

compile and run the program. Then type PLAY to play the sound.

Please note that you need to include the previous `compile.bat` file for the audio to generate. Audio will not be played if qemu is not called with `-soundhw pcspk`.

External References

https://wiki.osdev.org/PC_Speaker

<https://wiki.osdev.org/Sound>

https://wiki.osdev.org/Sound_Blasters_16

Going Advanced

This section is dedicated for advanced development. We will give some small descriptions for each topic and relevant reference links. Please note that this won't give you a complete idea as each topic usually requires an entire Book.

CD-ROM

ATAPI

The technology used to communicate with CD-ROM is ATAPI. It uses the Packet Interface of the ATA6 or higher standard command set. ATAPI give commands needed for controlling CD-ROM drive and Tape drive.

The command set for ATAPI includes one for Reading and Writing Sectors , Additional features like setting CD SPEED , Read Track Information etc and more....

External Reference

<https://wiki.osdev.org/ATAPI>

USB

Universal Serial Bus

USB was introduced with the intention of replacing custom Hardware interfaces, and to simplify the configuration of these devices. The official USB specification might be hard to read and may demotivate you from implementing a driver for that. Anyway it is a Must-To-Implement feature if you have the plan to support USB Drives or the “Pen Drives”.

USB uses serial data transfer methods (And so the name `Universal Serial Bus`) and it's different version support different transfer speeds.

External Reference

<https://en.wikipedia.org/wiki/USB>

<https://wiki.osdev.org/USB>

Networking

Networking

Networking also is a large topic as it involves communicating between Network interfaces(We need to implement a lot of drivers for different Interfaces including Wired and Wireless).

When a driver for these Network cards are developed , we need to implement some Program for the communication protocol like TCP , UDP etc....

External Reference

https://wiki.osdev.org/Intel_Ethernet_i217

https://en.wikipedia.org/wiki/Transmission_Control_Protocol

https://en.wikipedia.org/wiki/User_Datagram_Protocol

Paging

Paging

Paging is a feature provided by x86 cpu which allows use of hard disk to temporarily store Data intended to be stored in ram and when needed take it back. This practically allows a program to utilize memory more than what a system have.

But this operation is little slower as a hard disk actually works at low speed. A solution for this is to use an SSD.

External Reference

https://en.wikipedia.org/wiki/Memory_paging

<https://wiki.osdev.org/Paging>

GDT

Global Descriptor Table

Global Descriptor Table or GDT is a table which contains information about memory areas and who can access those. This is very essential as an absence of this feature could allow user mode programs to do things like editing the kernel routines which results in complete system takedown.

We have already implemented gdt with the lgdt assembly instruction.

External Reference

https://en.wikipedia.org/wiki/Global_Descriptor_Table

https://wiki.osdev.org/Global_Descriptor_Table

https://wiki.osdev.org/GDT_Tutorial

IDT

Interrupt Descriptor Table

Interrupt Descriptor Table or IDT is a mechanism implemented in x86 cpu which allows an operating system to capture events and do necessary operations. The events include Keyboard Input , Mouse Input etc... and also includes software defined interrupts which in x86 cpu occurs during things like when executing the int command.

An absense of feature like Interrupt Descriptor Table will make the system inefficient at working. This is because in this case , our only solution will be to ask for events every time to these devices.

External Reference

<https://wiki.osdev.org/Interrupts>

https://wiki.osdev.org/Interrupt_Descriptor_Table

https://wiki.osdev.org/Interrupt_Service_Routines

Timers

Programmable Interval Timer

Programmable Interval Timer or PIT is a counter that generates an interrupt when it reaches a programmed count. Things such as a Delay Operations could be implemented using this.

External Reference

https://en.wikipedia.org/wiki/Programmable_interval_timer

https://wiki.osdev.org/Programmable_Interval_Timer

GRUB

https://en.wikipedia.org/wiki/GNU_GRUB

<https://wiki.osdev.org/GRUB>

UEFI

<https://www.howtogeek.com/56958/htg-explains-how-uefi-will-replace-the-bios/>

<https://wiki.osdev.org/UEFI>

https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface

How To Move Further?

There are many resources online to move further. You now have an idea how things works , You can utilize it to move further. Here's some resources

<https://littleosbook.github.io/book.pdf>

<http://www.brokenthorn.com/Resources/OSDevIndex.html>

<http://wyoos.org/Sources/index.php>

The Thank You Summary

We have travelled a small road till here. This book aims for beginners to get started in os development and to give an idea about how things work basically. Getting good fluency in this topic need doing some projects.

Everyone have to put effort when learning a new topic. Not understanding for the first time is not a bad thing. Consistency and Patience will surely lead you to success.

At the end of time , i want every one to be successful in the Journey.
Wishing You Good Luck!!!

Thank You!!

Create Your Own Operating System

Build, deploy, and test your very own operating systems for the Internet of Things and other devices

LUCUS DARNELL

Create Your Own Operating Systems for the Internet of Things

By Lucas Darnell

Table of Contents

[Disclaimer](#)

[Introduction](#)

[0x01 OS Basics](#)

[0x02 Intro to Machine Code](#)

[0x03 Intro to the Assembly Programming Language](#)

[0x04 Intro to the C Programming Language](#)

[0x05 Getting Started](#)

[Installing Virtualbox](#)

[Installing Linux](#)

[Installing GNOME](#)

[Preparing CentOS and the VM](#)

[Troubleshooting VirtualBox Guest Additions](#)

[Preparing the Development Environment](#)

[0x06 Bootstrapping with the Bootloader](#)

[Creating the Entry Point](#)

[GNU GRUB](#)

[Compiling the Entry Point](#)

[0x07 Welcome to the Kernel](#)

[0x08 Putting it all Together](#)

[0x09 Testing Your Operating System](#)

[0x0A Starting Your Architecture Library](#)

[Expanding the Console](#)

[0x0B Expanding Your OS](#)

[0x0C Cross-Compiling for Other Architectures](#)

[Create a Custom Cross-Compiler](#)

[Porting for the Raspberry Pi](#)

[Testing on Physical Hardware](#)

[Conclusion](#)

[Acknowledgements](#)

[Appendix](#)

[Compiling the “Print Lucas” Example](#)

[Complete x86 Operating System Source](#)

[boot.asm](#)

[kernel.c](#)

[linker.ld](#)

[grub.cfg](#)

[Makefile](#)

[Index](#)

Disclaimer

I have spent a lot of extremely late nights writing this book. So, if you find any errors within this book, please feel free to send me an email at the address below so that I can correct those errors and help you resolve any problems you might have.

With that said, the information in this book is for educational purposes only. Neither the author nor the publisher are responsible or liable for any direct, indirect, consequential, or incidental loss, damage, injury, or other issues that may result or arise from the use, misuse, or abuse of the information provided in this book.

No part of this book may be reproduced in any manner whatsoever without the written permission of the publisher.

Copyright © 2016 Lucus Darnell – lucus@prodigyproductionsllc.com

All rights reserved.

Introduction

Take a look around you right now and count how many electronic devices you can see from your current point of view. If you are like most people, you probably have numerous devices just within your current eye sight. In order for many of those devices to work, they have some sort of operating system (OS) that allows them to boot up and provide whatever experiences it is they have to offer. In fact, if you are reading the digital version of this book, the very device you are now holding has an operating system of its own.

However, not all electronic devices have an operating system. For example, most microwave ovens do not have an OS since they only have a small set of tasks to perform with very little input (in the form of a keypad) to process and no chance of the hardware changing. Operating systems would typically be overkill for devices like this as they would increase development and manufacturing costs and add complexity where none is required.

With that said, technologies are getting smaller, cheaper, and faster which is providing manufacturers the ability to leverage full-blown operating systems for handling more processing and adding additional functionality in their devices. Take the microwave oven from before as an example. Even devices such as these are beginning to be connected to the *Internet of Things*¹ (IoT) so that manufacturers can track the usage (and issues) with their devices even after they have been purchased and installed in our homes. Plus, IoT-enabled devices such as the “connected microwave” allows the end user - the consumer in this case - to monitor and manage their own devices remotely and autonomously. Devices with this capability require an operating system to handle the connectivity to the Internet along with whatever base functionality the devices already provide.

¹ If you would like to learn more about the *Internet of Things* and explore various real-world use cases, please check out my book on Amazon aptly titled, “*The Internet of Things: A Look at Real-World Use Cases and Concerns*” which you can find at <http://www.amazon.com/gp/product/B017M7S0A0>

Recently I started working on a new project for the *Internet of Things* that involves the need for an entirely new operating system. Unfortunately, my C programming skills were a bit rusty and it had been more than fifteen years since I had written anything in Assembly. So, in order for me to brush up on my C and Assembly skills I needed to build a new operating system, I did like any good programmer would do and I turned to the Internet. The first place I began my search was amazon.com. I was hoping that I would find a decent book that could help me along with my project. However, to my surprise, I could only find one book that is dedicated to developing an operating system from scratch and it has some pretty nasty reviews. That was when I decided to write my own book that will hopefully help the next generation of programmers get back to the roots of developing software by explaining the fundamentals of how computers actually work all while building toward the end goal of having your own operating system.

Despite what the forum trolls will tell you, writing your own operating system is actually quite easy. The tough part comes when it is time to expand your OS. Since you are building a completely new operating system, you will not have the luxury of utilizing readily available libraries that existing operating systems provide out-of-the-box. That means that for every little piece of extra functionality to even the most basic of system calls, you will have to write every last bit from scratch. If you continue working on your OS, over time you will develop a good size library that rivals some of the operating systems that exist today.

For the purposes of this book, I will walk you through the step-by-step process of building and running your very own operating system for the x86 processor. Why the x86 processor? Well, this decision is made based on the fact that there are already plenty of x86 based computers available in which you can get your feet wet. Plus, there are lots of websites out there to help you should you get stuck with anything further down the road. Even though the operating system you will learn to develop in this book is intended for the x86 processor, you can use these same techniques for building an OS for other types of processors as well, which I will explain in a later chapter.

One assumption I will make throughout this book is that you already have a fair amount of understanding for how computers work. For readers that do, a few of the chapters in this book will basically be refreshers. For readers that are completely new to programming and/or how computers work, you will most likely find some of the things mentioned herein a little difficult to follow. However, I will still do my best to (re)introduce certain concepts to you to help illustrate what we are trying to achieve in this book. Regardless of your knowledge and skill level, I hope that you will find the following text educational, informative, and useful. With that said, let's dive right in.

0x01 OS Basics

Before we jump into building the next Windows, Linux, Android, OS X, iOS, or other fancy operating system, we need to begin by examining what an operating system is and how it works in the first place. To begin with, the operating system (OS) is the primary piece of software that sits between the physical hardware and the applications used by the end user (such as Word and Excel).

The most popular operating systems are those mentioned above where Windows, Linux, and OS X are the most common operating systems for desktop & laptop computers. Variations of Linux (along with Unix) are also responsible for powering the majority of high-end servers such as those that power the Internet while Android and iOS are [currently] the two most popular operating systems for powering mobile devices such as smartphones and tablets.

The primary responsibility of the operating system is to provide a common way for applications to work with hardware regardless of who manufactures that hardware and what it includes. For example, a laptop computer contains different hardware than that found in a desktop computer. Both types of computers can also have different amounts of memory, processor speeds, and peripherals. Instead of requiring software developers to build their applications to support all of these various pieces of hardware, the operating system abstracts that away by providing an interface that the applications can interact with in a common way, regardless of what the underlying hardware looks like.

Since the primary responsibility of the operating system is to provide a common interface for applications to interact with the physical hardware (and with each other), it is also the OS's job to make sure that all applications are treated equally so that they can each utilize the hardware when necessary.

In order for any software (including the operating system) to run, it has to first be loaded into memory where the Central Processing Unit (CPU) can access and execute it. At first, the software starts out residing on some sort of storage device - be it a hard disk drive (HDD), solid state drive (SSD), CD or DVD, SD card, thumb drive, or other type of flash memory device or ROM - but is moved into memory when the system is ready to operate with the relevant piece of software.

In general, the operating system is responsible for several tasks. The first task that the operating system is responsible for is managing the usage of the processor. It works by dividing the processor's work into manageable chunks. Next, it prioritizes these chunks before sending them to the CPU for processing and execution.

The second task the OS is responsible for is memory management. Depending on the purpose and needs of the operating system, memory management can be handled in different ways. The first is what is known as “single contiguous allocation”. This is where just enough memory for running the operating system is reserved for the OS while the rest is made available to a single application (such as MS-DOS and embedded systems).

The second type of memory management is known as “partitioned allocation”. This is where the memory is divided into multiple partitions and each partition is allocated to a specific job or task and is unallocated (freed) when the job is complete and the memory can be re-allocated to other tasks.

The third type of memory management is known as “paged management”. This is where the system divides all of its memory into equal fixed-sized units called “page frames”. As applications are loaded, their virtual address spaces are then divided into pages of the same size.

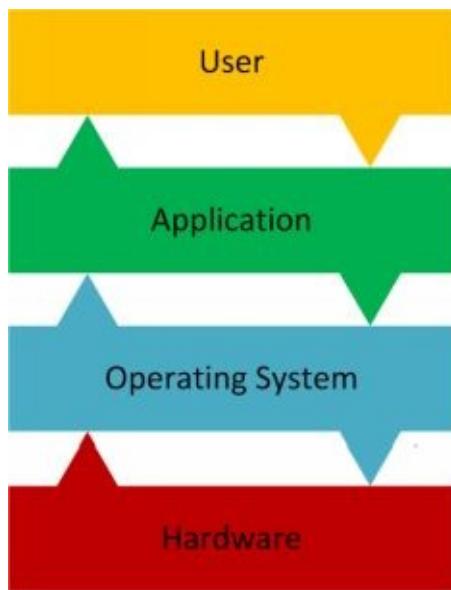
The fourth type of memory management is known as “segmented management”. This is where the system takes a “chunk” of an application (such as a function) and loads it into a memory segment by itself. This memory segment is allocated based on the size of the “chunk” being loaded and prevents other applications from accessing the same memory space.

After the operating system has setup the memory to accept code to be executed, it then moves its attention to managing the rest of the hardware. For example, the OS is responsible for interpreting input from peripherals such as the mouse and keyboard so that the applications that depend on these devices can interact with them in a uniform fashion. It is also responsible for sending output to the monitor at the correct resolution as well as configuring the network interface if available.

Next, the operating system takes care of managing devices for storing data. Persisting data to long term storage is absolutely necessary for user-based operating systems such as Windows, Linux, and OS X, but isn’t necessarily required for other operating systems such as those that run on embedded devices.

Aside from managing the physical hardware and providing an interface for applications to work with said hardware, the operating system is also responsible for providing a user interface (UI) which is what allows the user to interact with the system. The user interface that the operating system provides is also [typically] what defines how the applications running on the OS look as well.

To better understand where the operating system sits in relation to the rest of the system and to better understand its responsibilities, the following image should help.



As you can see (looking at the image from the bottom up), all computer systems begin with the bare metal hardware. The operating system sits on top of the hardware and acts as a manager of the hardware. As mentioned before, it is also responsible for interfacing applications (the software we use) with said hardware. At the top of it all is the user. Whenever a task needs to be performed by the system, the user feeds input into the system via hardware peripherals such as a mouse and keyboard. This input is then read in by the operating system and passed up to the application which provides the code for performing whatever task is expected by the user. After the application has completed its tasks, it passes its output back to the operating system where it is passed back to the hardware in the form of feedback to the user. This feedback can come in various forms such as text and images on a monitor, sounds from the speakers, or other output via tactile devices.

0x02 Intro to Machine Code

At its most basic level, a computer works by pushing a string of numbers into memory where the CPU can process those numbers. Depending on what those numbers are and what sequence they are ordered in, the computer will read those numbers and perform different tasks. When these numbers, are passed into memory, they take on the form of higher and lower voltage. When these numbers are passed into the CPU, they tell tiny transistors to change their state to be either on or off (again, higher or lower voltage). This shifting in state is what controls something called a “logic gate”, which can be either open or closed.

If you noticed, the examples mentioned in the previous paragraph can only take two states: high or low; on or off; open or closed. In order for a computer to understand these states, they are represented as ones and zeros which we call “binary” or “machine code”: 10110110, for example. When strings of these zeros and ones are read by the computer, they can cause it to do great things.

Each zero and one found in a binary string represents what is called a “bit”. When you group eight bits together, you get what is known as a “byte”. One thousand bytes is equal to one kilobyte. One thousand kilobytes is equal to one megabyte. One thousand megabytes is equal to one gigabyte. And so on. In the world of computers that we will be designing our operating system for, a bit can only be a one or a zero as explained above. However, in the world of quantum computing, a “quantum bit” (also known as a “qubit” for short) can be a one or a zero or both at the same time. But, I will leave that for another book.

Originally, bits were grouped together in threes with each bit being on or off and representing eight numbers from 0 to 7: 000 = 0; 001 = 1; 010 = 2; 011 = 3; 100 = 4; 101 = 5; 110 = 6; 111 = 7. This is what is known as “octal” (meaning 8). As computers became more powerful, engineers found that by grouping bits together four at a time instead, they could double the number of possibilities which gives us the number 16 and where we get hexadecimal: “hex” meaning 6 and “decimal” meaning 10. We will learn about decimal and hexadecimal shortly.

To the untrained eye, a string of zeros and ones looks like ... well ... a string of zeros and ones. But, these zeros and ones can be crunched at incredible speeds by a computer where endless calculations can be performed. However, since humans can't crunch the zeros and ones as fast as a computer can, we choose to use other ways to represent these numbers. Plus, by using other representations for machine code and by using the same simple mathematics that we learned as children, we too can perform the same calculations as the computer, just not as fast.

To keep things simple for us humans, we like to use the base-10 numerical system when counting. This numerical system is known as the “decimal” numeral system because “dec” in “decimal” actually means “10”. The “base” part, also known as “radix”, represents the number of different digits or combinations of digits and letters a system uses to represent letters. For example, in the decimal counting system, everything begins with a derivative of 10, with 0 being the lowest possible value, and counting upward 10 places (i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Once we have counted 10 places, the numerical system starts over again and increments by 10 each time it starts over and so on (i.e. **10**, 11, 12, 13, 14, 15, 16, 17, 18, 19, **20** 96, 97, 98, 99, **100**, 101).

The base-10 numerical system provides us with the simple numbers that we use in everyday life. At the same time, every number we can possibly imagine also has a binary representation. For example, the current year, 2016, can be represented in binary as 00110010001100000011000100110110. However, the use of binary digits (zeros and ones) is also an abstraction. The number “2016” could just as well be an instruction that tells the computer what to do, such as load a value from memory into a CPU register. The zeros and ones are simply a way for us to distinguish between two different values which is what we call “boolean logic”. It is also what is known as a base-2 system. But, as mentioned before, using a base-10 (decimal) system makes things easier for us humans. Take a look at the following sequence of numbers for example.

```
0 17791 17996 258 1 0 0 0 0 2 62 1 0 176 64 0 0 64 0 0 0 240 0 0 0 0 0 64 56 2 64 4 3 1 0 5 0 0 0 0 0 0 64 0 0 0 64 0 0
205 0 0 0 205 0 0 0 0 32 0 0 1 0 6 0 208 0 0 0 208 96 0 0 208 96 0 0 6 0 0 0 6 0 0 0 0 32 0 0 1722 0 47360 208 96 443 0
47104 4 0 32973 440 0 52480 128 0 30028 30051 2675 11776 26739 29811 29810 25185 11776 25972 29816 11776
24932 24948 0 0 0 0 0 0 0 0 0 11 0 1 0 6 0 0 0 176 64 0 0 176 0 0 0 29 0 0 0 0 0 0 0 16 0 0 0 0 0 0 0 17 0 1 0 3 0 0 0
208 96 0 0 208 0 0 0 6 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 1 0 3 0 0 0 0 0 0 0 0 214 0 0 0 23 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
```

To you and me, these numbers don’t mean a thing. But, to a machine that can speak this language, the numbers are a set of instructions. In this particular case, the numbers above tell the computer to print my name, “Lucus”, to a console window. When these numbers are converted from decimal to machine code, that is when they become binary digits in the form of 1’s and 0’s as mentioned above.

Although humans like to use the base-10 (decimal) system, computers prefer to use a base-16 system instead called “hexadecimal” or “hex” for short (mentioned earlier). Unfortunately, counting sixteen digits and starting over with an additional increment of sixteen like we do with the base-10 system proved to be a hassle when only working with numerical digits. So, the letters A, B, C, D, E, and F were added to the decimal counting system to account for the remaining six digits. Therefore, after counting from 0 to 9, you can substitute the letter A for the number 10, B for the number 11, all the way up to the letter F for 15, giving us a grand total of 16 digits (remember, we start counting with zero).

As per Wikipedia, four bits is known as a “nibble” and a nibble is one hexadecimal digit

(i.e. 0-9 or A-F). Two nibbles, or eight bits, make up a byte (i.e. 00 - FF). When you total up all possibilities of 00 - FF combinations, 00 - FF in hexadecimal is equal to 0 - 255 in decimal. Again, this demonstrates how using our base-10 counting system is easier for us humans.

To give you an idea of what programs look like in hexadecimal format, let's take the same number sequence from above and convert all of the decimal values to hexadecimal. When you do the conversion, you will end up with the following.

Even though this mess isn't any easier to read than the decimal version above, it should at least give you an idea of how hexadecimal works. If you were to take those same hexadecimal values and convert them to binary so that the computer can understand them, you would end up with the following.



The purpose of showing you all of this is to give you an idea of what the computer needs from us in order to operate. However, as you can imagine, writing software by typing in a bunch of zeros and ones (binary) or even a bunch of numbers and letters (hexadecimal) can be excruciatingly painful. Luckily, some very smart guys came up with a programming language called “Assembly” which allows us to write applications using human words such as add, call, and push. These words (also known as “mnemonics”) are then translated back into machine code by programs called “assemblers”, hence the name “Assembly”.

0x03 Intro to the Assembly Programming Language

If you know anything about programming - which I am hoping you do since you are choosing to read this book, you will know that Assembly language programming is as close to machine language programming as you can get. Because of this, software written in Assembly can be developed more efficiently and better optimized than in any other language. However, Assembly-written applications do not have the high-level conveniences such as functions and variables that other programming languages provide which makes developing in Assembly a little more difficult than in other languages (until you know what you are doing). But, it is still much easier and faster than writing a bunch of 1's and 0's.

Even though it is possible to create our own Assembly language syntax, there are two main syntaxes that people tend to use: Intel syntax and AT&T syntax. Identifying one of these syntaxes from the other is quite easy as the AT&T syntax prefixes the \$ and % symbols to almost everything. If you have ever done any program disassembly on Linux, you will know that most disassemblers like to use the AT&T syntax by default. But, in my opinion, the Intel syntax is a little cleaner and will therefore be the syntax I use for the examples in this book.

So, how does Assembly work? The Assembly programming language works by using symbols, also called “mnemonics”, to represent binary code in a way that humans can easily read and understand it. For example, the following code is a mnemonic way of telling the computer to move the data “0x1234” into the **EAX** register. Depending on the lines that follow this instruction, the computer will do something different with that data from the **BX** register.

`MOV EAX, 0x1234`

Now, isn't that a lot easier to read than 1011 1000 0011 0100 0001 0010 0000 0000 0000 0000? By the way, Assembly instructions are not case-sensitive. For example, the same line above can also be illustrated as:

`mov eax, 0x1234`

Also, instead of using hexadecimal values like above, you can also use decimal and even ascii values like this:

`mov eax, 4660`

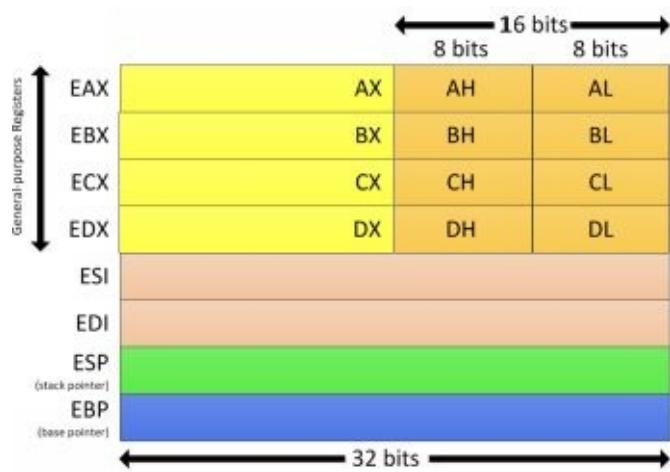
With only a few exceptions, Assembly commands come in the form of **MNEMONIC [space] DESTINATION [comma] SOURCE**. If you can remember this simple rule, following Assembly code can actually be quite simple. The only thing you really have left to do from there is learn what each of the mnemonics are and what they do.

Note: We can also add comments (non-instructional messages) to our code by pre-fixing them with a semi-colon like this:

```
mov eax, 4660 ; Move the value 4660 into the eax register
```

Since we will be designing our super-cool operating system for the x86 architecture, we can find a full list of x86 mnemonics and instructions at https://en.wikipedia.org/wiki/X86_instruction_listings. For now, I will go ahead and mention some of the most common instructions – at least those that we will be using in our operating system. But before I do that, let's first take a look at how the x86 registers are laid out so that we can have a better idea of what the data looks like as it moves around in the CPU.

As you can see in the diagram below, x86 processors consist of 32-bit general purpose registers. You can think of registers as variables, but in the processor instead of in memory. Since registers are already in the processor and that is where the work actually takes place, using registers instead of memory to store values makes the entire process faster and cleaner.



As shown in the diagram above, the x86 architecture consists of four general purpose registers (EAX, EBX, ECX, and EDX) which are 32-bits. These can then be broken down further into 16-bits (AX, BX, CX, & DX) and even further into 8-bits (AH, AL, BH, BL, CH, CL, DH, & DL). The “H” and “L” suffix on the 8-bit registers represent high and low byte values. (Remember earlier how high and low meant on and off, respectively?)

Everything that includes an “A” (EAX, AX, AH, & AL) are referred to as “accumulator registers”. These are used for I/O (input/output), arithmetic, interrupt calls, etc. Everything that includes a “B” (EBX, BX, BH, & BL) are known as “base registers” which are used as base pointers for memory access. Everything with a “C” in it (ECX, CX, CH, & CL) are known as “counter registers”. These are used as loop counters and for shifts. Everything with a “D” in it (EDX, DX, DH, & DL) are called “data registers”. Similar to the accumulator register, these are used for I/O port access, arithmetic, and some interrupt calls.

The bottom four rows in the diagram above represent indexes (ESI & EDI) and pointers (ESP & EBP). These registers are responsible for pointing to memory addresses where other code is stored and referenced.

Going back to the **mov eax, 0x1234** (**mov eax, 4660**) example presented earlier, the “**mov**” instruction simply copies data from one location to another. In this case, it moves the value contained in location “4660” into the “**eax**” register. You can also use the “**mov**” instruction to move data from one register to another like so:

```
mov eax, ebx
```

In this case, the four bytes of data that are currently at the address contained in the **ebx** register will be moved into the **eax** register. Likewise, you can also use 32-bit constant variables in place of registers. For example, if you wanted to move the contents of **ebx** into the four bytes at memory address “**var**”, you would use:

```
mov [var], ebx
```

Before we can use our “**var**” (aka “**variable**”), we will first need to define it. We can do this in a static data region rightfully named “**.data**”. Any static data should be declared in the **.data** region which in effect will make it globally available to the rest of our Assembly code. So, before using “**var**” from the example above, we will begin by declaring our **.data** region followed by declaring our variable like so:

```
.data
var db 32
```

In this example, we are declaring a variable called “**var**” which will be initialized with the

value “32”. However, there will be a lot of times where we will not have or know the value to initialize our variable with. In these instances, we can declare an uninitialized byte with the following:

```
.data  
var db ?
```

Depending on the size required for our variable, we will need to swap out “**db**” with the size that matches our needs. Here is a simple cheat-sheet for identifying which size type to use.

DB = Define Byte: 1 byte = 8 bits

DW = Define Word: 2 bytes = 16 bits

DD = Define Double Word: 4 bytes = 32 bits

Along with the “**mov**” instruction, the next instruction we need to know about is the “**int**” instruction. This instruction is an **interrupt** which tells the hardware to stop what it is doing and do whatever comes next in our list of instructions.

```
int 0x80
```

Next up is the “**push**” instruction. Basically, the push instruction tells the system to place its operand on top of the hardware stack in memory. It begins by first decrementing **esp** by four followed by placing its operand into the contents of the 32-bit location at address **esp** (the stack pointer). Since the **esp** instruction is decremented every time the push instruction is performed, the stack grows from high addresses to lower addresses.

Note: “Stack” refers to the simple data structure used in computing for allocating and aligning code so that it can be accessed in a uniform way. As new objects are introduced to the stack, they are placed one on top of the other where the most recent piece of code placed (pushed) onto the stack is the first to be removed (popped) from the stack once it is no longer needed. This is known as Last-In-First-Off (LIFO) since objects further down the stack cannot be accessed or removed from the stack until higher up objects have first been removed.

```
push eax
```

If the “**push**” instruction pushes operands onto the stack, how do we get them back off? Well, that is where the “**pop**” instruction comes into play. It works by removing the four bytes data element from the top of the hardware stack (**esp**), moving it into the memory location at **sp**, and then increments **sp** by four.

```
pop ebx
```

Aside from the instructions already mentioned, you should also go ahead and learn arithmetic and logic instructions for **add** (**addition**), **sub** (**subtraction**), **inc** (**increment**), **dec** (**decrement**), **and**, **or**, **xor**, and **not**. Also, you should probably go ahead and learn the **jmp** and **call** instructions while you are at it too. Basically, the **jmp** instruction transfers program control flow to the instruction at the memory location indicated by the operand while the **call** instruction calls out to another subroutine (such as in the C code of our kernel which we will learn about in the next chapter).

As mentioned earlier, any time you want to declare global variables, you will need to define them in a section called “**.data**”. Similarly, any time you want to perform operations, you will need to define them in a section called “**.text**”. However, unlike the **.data** section which can be written to as well as read, the **.text** section is read-only. Since the **.text** section is read-only and will never change, this section is loaded into memory only once which reduces the usage and launch time of the application. Even though values in the **.data** section are intended to be changed, you can also use this section to declare constant values that do not change such as file names and buffer sizes.

With that said, there is also a third section called the “**.bss**” section which is used for declaring variables. Wait a minute! I thought the **.data** section was used for declaring variables? Even though you declare variables in the **.data** section, in a “formal” application, the **.data** section will be used for declaring initialized data and constants while the **.bss** section is used for declaring variables that will change during runtime. However, as you will see in the following example, you do not have to include the **.bss** section if you don’t need/want it. Just to complicate things even further, when we finally move into building our operating system, we will see that we can also use the **.bss** and not the **.data** section. Although they can be interchangeable, both have their time and place and should be used accordingly for best performance and practice.

To give us our first taste of an Assembly-written application, let’s take a look at the series of numbers from the [last chapter](#) to see how those decimal, hexadecimal, and binary commands would appear in Assembly.

The instructions for writing, compiling, and executing this code can be found at the [end of this book](#). But for now, I want to briefly explain it as it can help us get a better idea of how

the Assembly programming language looks.

print_lucus.asm

```
section .text
```

```
global _start
```

```
_start:
```

```
    mov edx,len
```

```
    mov ecx,msg
```

```
    mov ebx,1
```

```
    mov eax,4
```

```
    int 0x80
```

```
    mov eax,1
```

```
    int 0x80
```

```
section .data
```

```
msg db 'Lucus', 0xa
```

```
len equ $ - msg
```

In the above example, you will see that the **.text** section begins with two lines: “**global _start**” and “**_start:**”. This is what tells the system where the program execution begins. The “**msg**” part at the beginning of the **.data** section is where we define the text (“Lucus” in this case) that we want to print to the screen and that we want to store this text in the “**msg**” variable as mentioned earlier. Also in the **.data** section, we define another variable called “**len**” which stores the length of the text found in our **msg** variable.

Although we will use the **_start:** symbol in the examples in this book, this definition can actually be named something else as it is only specific to the NASM compiler which we will learn about later. The purpose of using **_start:** (or whatever name you choose) is to tell NASM where the entry point to our application is. The linker application (that we will also learn about later on) will then read this symbol and tell the system that this is where it should look for its first instruction. By adding “**global _start**” at the beginning of our app, the “**global**” directive simply tells NASM to load the “**_start**” symbol into the object code so that it can be found later on in this location.

Back in the `.text` section after “`_start:`” (where our program execution begins), we tell the processor to move the value from our `len` variable into the `edx` register followed by moving the actual text itself into the `ecx` register. Basically, this is telling the system to allocate `[len]` bytes so that we can store `[msg]` text in that allocated space.

After that, the “`mov ebx, 1`” command indicates “standard output” (aka “print to terminal”) while “`move eax, 4`” is a system call to “`sys_write`”. Immediately after those commands you will see “`int 0x80`” (again, “`int`” is short for “`interrupt`”) which passes control over to the kernel, allowing for any system calls to be made (“write to standard output” in this case). The “`move eax, 1`” command is a system call to “`sys_exit`” which tells the system that it has completed and that the following “`int 0x80`” should once again return control back over to the kernel for whatever processing is next on the stack (or to exit since there are no more instructions).



At this point, you should have a very simple Assembly application that prints “Lucus” (or whatever text you chose to use) to the terminal. Based on this simple application, you should also have a very basic understanding of how the Assembly programming language works. It isn’t much, but it is enough to get you started with creating your own operating system. With that said, if you found the Assembly programming language a little difficult to follow, don’t worry. The [next chapter](#) will show you something that makes programming a computer a little easier than if it was all done in Assembly.

0x04 Intro to the C Programming Language

There are plenty of books and other resources for learning the C programming language and I encourage you to look into those to further advance your skills. But, I still want to take some time to introduce you to the basics required to get started developing in the C programming language since it is what we will be using to develop the majority of our operating system.

As we learned in the [last chapter](#), the Assembly programming language is just an abstraction from machine code. Even though Assembly can make programming easier, it is still a little difficult to learn and follow. To help with that, between 1969 and 1973, a man at AT&T Bell Labs named Dennis M. Ritchie brought us yet another abstraction layer on top of machine instructions called the C programming language.

According to Wikipedia, “*C is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. By design, C provides constructs that map efficiently to typical machine instructions, and therefore it has a found lasting use in applications that had formerly been coded in Assembly language, including operating systems, as well as various application software for computers ranging from supercomputers to embedded systems.*”

I know that is a mouthful, but let me break it down for you. Basically, C is a high-level programming language (where machine code / binary is the low-level language) that allows you to read and write code for a large number of platforms. Not only does C allow you to develop applications for pretty much any kind of system you can imagine, but it also allows you to develop applications that are optimized to perform at the fastest speeds possible. This is because applications written in C get compiled straight to machine instructions, just like with applications written in Assembly but not quite as fast. If application speed is what matters, you should stick with developing everything in Assembly. But, if you can afford to trade-off a slight performance hit for easier development, then you will not be disappointed with C.

It is also worth noting that C was originally invented as a way to develop the UNIX operating system. Since we too are wanting to create our own operating system, you can see why C is an obvious choice for us to program in. Not to mention, once you get the hang of the C programming language, you can continue to use it to further develop your operating system as well as have a nice employable skill in your toolbox.

So, how does the C programming language work? Well, to begin with, the C programming language is based on the premise of providing a list of procedures - also known as routines, subroutines, or functions - that the computer must follow. Inside each procedure

is a list of instructions, similar to those found in Assembly, that when followed enable the computer to do the many things a computer is capable of. In Assembly, these procedures are listed in what we learned in the [last chapter](#) as “sections”. Sections in Assembly are declared by the word “section” followed by a period prefixed to its name.

```
section .text
```

In C, these sections are called “procedures” which is why developing applications in the C programming language is known as “procedural programming” which is derived from “structured programming”. Each procedure consists of the procedure’s return type followed by the procedure name followed by a list of parameters that are expected to be passed into the procedure which are all contained within parenthesis.

```
int say_hello (char *to)
```

The most common procedure, that also happens to be found in all C applications, is the “main” procedure. This procedure is required in all C applications because it is the entry point for all applications. In fact, the *main* procedure has a one-to-one mapping to the *_start:* symbol found in Assembly like we learned about in the [last chapter](#). The purpose of the *main* procedure is to kick off the list of instructions that the system must follow and, in return, notify the system whether or not the application started successfully.

In order to do this in a common fashion, all *main* procedures are required to return a zero or a non-zero integer: zero meaning success, non-zero meaning failure where each non-zero response can mean something different. As mentioned before, all procedures are declared first with its return type followed by its name and a list of parameters contained within parenthesis. The return type simply tells the system what type of value to expect it to send back upon completion. After the parenthesis, procedures include beginning and ending curly braces (i.e. { and }) which wrap the procedure’s list of instructions.

The best way to explain the “main” procedure would be to see an example of it. The following example is the C representation of the example from the [last chapter](#) that prints my name, “Lucus”, to the screen.

print_lucus.c

```
int main() {  
    puts("Lucus");  
    return 0;
```

}

Since the *main* procedure is required to return either a zero or a non-zero integer to indicate success or failure respectively, we define the procedure as “**int**” (short for “**integer**”). As you will notice in the example above, the very last thing we do is return a zero which indicates everything has executed as expected. If we wanted to indicate that something went awry, we could simply return something such as one or negative one.

You will also notice in the example above that the parentheses do not include any parameters as stated earlier. This is because, in the *main* procedure, it is implied that it will always contain “**int argc**” and “**char *argv[]**” as its parameters. The purpose of these parameters is to allow users to pass in parameters to our application from the command line. For example, if instead of hardcoding my name, “Lucus”, in the example above, we choose to allow users to pass in a name when running the app from the command line, we would need to include these parameters like so:

```
int main(int argc, char *argv[]) {  
    ....  
}
```

Note: If we do not use those parameters within our application, there is no need to define them in the procedure.

The first parameter, “**int argc**”, is an integer that indicates the **argument count**. This tells the application how many parameters it should be expecting. The second parameter, “**char *argv[]**” is a string representation of the parameters you are passing into the application. The “**argv**” name is short for **argument vector**. As you can see, the **[]** brackets after “**argv**” indicate that this is an array. This array will always contain the application name as its first value and all remaining arguments will make up the remainder of the array.

The “**puts**” instruction in the example above is what tells the system to output “Lucus” to the console. Later on we will use the “**puts**” function in our operating system as well as a similar function called “**printf**”. One difference between **puts** and **printf** is that the former always appends a new-line character (“**\n**”) to the end of whatever is to be printed. If we want the **printf** function to print a new-line, we will need to include “**\n**” along with our string as shown below.

```
printf("Lucus\n");
```

Even though the line above shows how to use the **printf** function with a new-line, I would highly recommend that you do not use this function in the manner demonstrated above when only printing a variable (more on that in a minute). For now, the second difference between **puts** and **printf** is that the latter is used to interpret arguments as formatted strings. Whereas the **puts** function only accepts one argument and outputs that straight to the screen, the **printf** function can accept a string-literal as its first argument followed by as many arguments as needed to match the number of format parameters found within the first string-literal argument.

Format parameters are simply placeholders that get replaced by their counterparts in the subsequent arguments that get passed to the **printf** function. Format parameters are denoted by the percentage sign (%) followed by a character that tells the **printf** function how to interpret its argument. The format parameters that we will be using in our operating system are %c (single character) and %s (string of characters). Here is a list of the most common format parameters supported by the **printf** function.

%c	single character
%d or %i	signed decimal integer
%e	scientific notation (mantissa/exponent) using e character
%E	scientific notation (mantissa/exponent) using E character
%f	decimal floating point
%g	uses the shorter of %e or %f
%G	uses the shorter of %E or %f
%o	signed octal
%s	string of characters
%u	unsigned decimal integer
%x	unsigned hexadecimal integer
%X	unsigned hexadecimal integer (capital letters)
%p	pointer address
%n	nothing printed

Here are a couple of examples of using the **printf** function.

```
printf(string);  
printf("Lucus\n");
```

```
printf("Name: %s", "Lucus");
printf("Name: %s", string);
```

In order to use the **printf** procedure, you will first need to add a reference to a file that knows about this procedure. To do that, you will add a line that includes a “.h” file called “**stdio.h**” so that you can use any functions that **stdio.h** knows about. These “.h” files are known as “**header**” files which are nothing more than files that contain definitions of functions/procedures found in other “.c” files. Here is what our code would look like if we were to replace the **puts** call with a call to **printf**.

print_lucus.c

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("Number of arguments: %d, Name: %s\n", argc, argv[1]);
    return 0;
}
```

If passing in “Lucus” as your only argument, the above example will output the following:

output

Number of arguments: 2, Name: Lucus

Again, we will get two as the number of arguments because the application name itself will always be stored as the first value in the **argv** array. If we want to print the application name instead of the second argument being passed in, we can simply replace **argv[1]** with **argv[0]** in the example above.

When using the **#include** statement, you will notice that sometimes the file names are wrapped with < and > symbols while other times file names are wrapped with double-quotes. Any time you see an include wrapped with < and >, it means that the compiler should look for the library version of the file first. If it cannot locate the library version of the file, the compiler will then look in the local directory for the file. Any time you see an include wrapped with double-quotes, it means the compiler should look in the local directory first. If it cannot locate the file in the local directory, it should then look for it in the library location instead.

Since our applications can grow quite large, especially is the case when developing our

own operating system, it is recommended that we move procedures into their own .c files and list their definitions (procedure return type, name, and parameters list) in their own .h files and reference those files from other files. Not only does this help minimize the amount of code in any given file which makes it easier to follow, but it also helps us to develop reusable code that can be called from any other application later down the road as well. Plus, every time we have code in our application and run said application, every bit of code included in our application will get loaded into memory whether that code gets executed or not. So, by having our procedures in their own .h and .c files and only calling those procedures only when/if needed, we can reduce the amount of memory required for executing our application which can also provide our app with better performance in regards to its speed of execution.

As mentioned earlier, it is bad practice to use the first example above to print values from variables using the **printf** procedure and not including a format string (i.e. **printf(string);**). Even though it will work functionally, it can lead to security vulnerabilities in our application. You can find lots of information online about this vulnerability by searching for “*format string vulnerability in printf*”. You can also contact me directly if you would like to learn more about it as I have personally found and exploited this vulnerability in several applications I have been responsible for throughout my career.

Once we have written our C code, we will need to compile it before we can run it. To do that with the examples and operating system illustrated in this book, we will use the “gcc” compiler that we will learn to install in the [next chapter](#). For now, the only thing we really need to know is that we compile applications by going to the command prompt and typing “gcc” followed by the name of your .c file like so:

```
# gcc print_lucus.c
```

If we run the command above, gcc will compile our code and output a file called “**a.out**” which we can execute with the following command:

```
# ./a.out
```

If you prefer to use a different name for your compiled application output, you can declare it by using the “**-o**” flag followed by the name you want to use. For example, if I want to compile the print_lucus.c example from above and output it to an executable file called “**runme**”, I would use the following command:

```
# gcc print_lucus.c -o runme
```

Then, I could execute the application with the following:

```
# ./runme
```

As explained before, the **main** procedure is expected to return an integer. Therefore, we begin our procedure declaration with “**int**” (short for **integer**). Other procedures will have the need to return other types such as a character, strings of characters, and so on. For those, we will need to declare our procedure with the appropriate return type. However, there will also be lots of times where we want to execute a procedure that isn’t expected to return anything at all. For these, we will use “**void**” at the beginning of our procedure declaration. For example, if we want to create a procedure that uses the **printf** function and can be reused from multiple locations throughout our operating system, we could create a new procedure like so:

```
void print_name(char *name) {  
    printf("Name: %s\n", name);  
}
```

We can then call this procedure from any other procedure. For example, if we want to call this procedure from our **main** procedure, our code would look like the following.

print_lucus.c

```
#include <stdio.h>  
  
void print_name(char *name) {  
    printf("Name: %s\n", name);  
}  
  
int main(int argc, char *argv[]) {  
    print_name(argv[1]);  
    return 0;  
}
```

Whenever the system loads our application into memory and the CPU so that it can be executed, procedures (and everything else) are loaded onto what is called the “**stack**”. As

things get loaded, whatever is encountered first in the code gets placed onto the stack first. The next piece of code that is encountered gets loaded on top of that and so on. Because of this and because the **main** procedure depends on the **print_name** procedure in the example above, we have to make sure that **print_name** is already available on the stack so that it can be called from **main**. To make sure that happens, we will need to make sure that **print_name** is listed in our code above the place it gets called later on (i.e. in the **main** procedure). Basically, the rule of thumb is that any time we plan on calling another piece of code, that other piece of code needs to be defined in our application above the line of code that calls it. This is why you will always find **#include** statements at the top of any application. Other programming languages such as Java and C# take care of arranging code on the stack for you.

Back to our “**void**” example, many programmers believe that **void** has no meaning in applications. Unfortunately for them, they are wrong. The “**void**” keyword is a pointer and can be used as a pointer either to procedures or to variables. The **void** pointer’s main purpose is to simply hold a memory address. Otherwise, if “**void**” had no meaning, the system would not know about the code found in that memory location and therefore would not do anything with it. For the examples and purposes found in this book, we only need to know that **void** procedures are not expected to return anything. But, I still thought I would mention the part about **void** pointers holding memory addresses to help with your knowledge of how computers work.



In this chapter I introduced you to the C programming language where we learned the basic structure of C code, how to reference code in external files, and how to pass parameters to applications and other procedures. By now you should have a pretty good grasp of what it takes to write and compile an application in C. Now it is time to put together an environment that will allow us to do all of this and to move us one step closer to having our own operating system.

0x05 Getting Started

In order for us to develop an operating system, we will first need an operating system that provides all of the tools necessary for building our new operating system. Lucky for us, there are already plenty of options for us to choose from. Even though OS development can be done in Windows, we will use Linux for our development and for the exercises in this book. To be specific, we will be using **CentOS** (Community ENTerprise Operating System) which you can get from <https://www.centos.org>. However, instead of replacing our current installation of Windows with Linux, we will instead use a virtualized environment where we will not only run Linux, but where we will also test our new operating system.

One of the best things about running a VM (Virtual Machine) is that whenever we need to shut down our computer, we don't have to also power down the operating system running in the VM. Instead, we can simply save the current state of our VM and close it so that the next time we open our VM, everything is exactly where we left it.

Another great thing about running a VM is that should we do anything that damages the operating system in the VM, we do not run the same risk of damaging our host operating system (the one that is running the physical computer). If we do run into any issues, we can always start from scratch and rebuild everything again. Or, more preferably, we can simply delete the current VM and replace it with a backup or "snapshot" that we make periodically when working within our VM environment.

Installing Virtualbox

For our virtualization environment, we will be using a product called **VirtualBox**. According to their website (<https://www.virtualbox.org>), “VirtualBox is a powerful x86 and AMD64/Intel64 virtualization product for enterprise as well as home use. Not only is VirtualBox an extremely feature rich, high performance product for enterprise customers, it is also the only professional solution that is freely available as Open Source Software under the terms of the GNU General Public License (GPL) version 2.”

If you have never worked with a virtualized environment before, the basic idea is that you can run multiple operating systems on your computer from within your current operating system and without replacing or interfering with your current OS or file system. For example, throughout this book I will be assuming that you are running Windows as the primary operating system on your computer. But, instead of replacing Windows with Linux or doing anything that will mess up our Windows installation, we will run Linux from inside of Windows as if it is just another application such as Word or Excel.

To get started, we will open our browser and navigate to <https://www.virtualbox.org/wiki/Downloads>. There we will find links to download VirtualBox for Windows hosts, OS X hosts, Linux hosts, and Solaris hosts. Since I am assuming you are running Windows as mentioned before, we will need to download the version for “Windows hosts” which is (as of the writing of this book) version 5.0.10.

After we have downloaded VirtualBox, we will run it to begin the installation process. Unless you have other reasons not to, you should stick with the default options that the installer suggests. Toward the end of the installation, you will be presented with a warning message that reads, “*Installing the Oracle VM VirtualBox 5.0.10 Networking feature will reset your network connection and temporarily disconnect you from the network.*” This is OK. Just click the “Yes” button followed by clicking the “Install” button on the next screen to install VirtualBox. Depending on your system settings, you may be asked throughout the installation process to approve the installation of the various components.

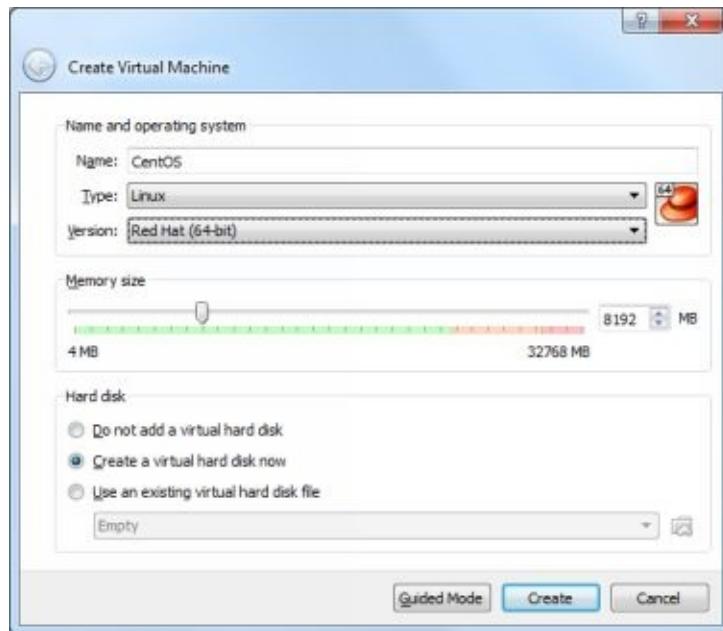
Installing Linux

Now that we have a place to run Linux, we will need to download Linux itself. To do that, we will open our browser and navigate to <https://www.centos.org/download/>. There we will find links to download a “DVD ISO”, “Everything ISO”, or a “Minimal ISO”. You can choose any of the ISO’s you want. But for the purposes of this book, you should choose to download the “Minimal ISO”. If you would prefer to boot your computer into Linux and not run Linux from inside of Windows using VirtualBox as explained in this book, you can choose to download the “DVD ISO” and burn it to a DVD (or thumbdrive), but you will need to locate instructions for doing that on your own. For now, just stick with downloading the “Minimal ISO”.

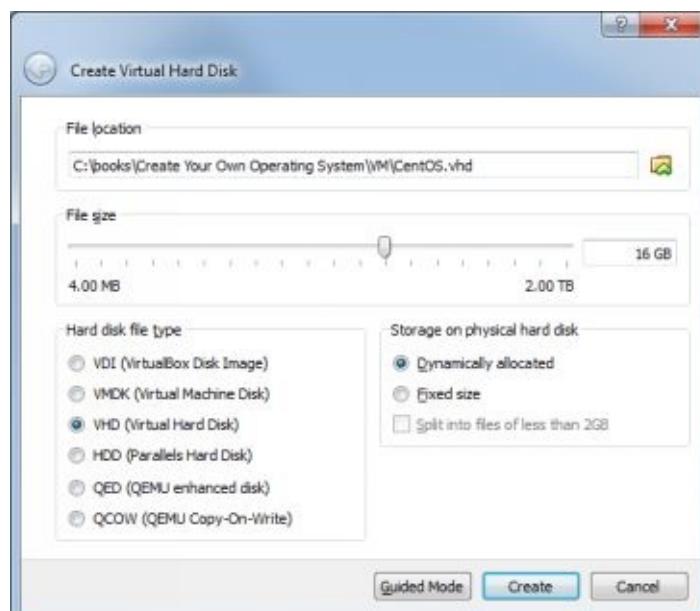
Once CentOS has finished downloading, open VirtualBox and click the “New” button in the upper-left corner. This will open a modal window that will ask you about the virtual machine you are attempting to create. For the “Name” field, type in something meaningful such as “CentOS” or similar so that you can easily identify your development environment OS from other virtualized operating systems, including the OS you are about to build.

Next, click on the “Type” dropdown and select “Linux” for the type followed by “Red Hat (64-bit)” for the Version. Now, I’m sure you are probably asking yourself, “Why am I selecting Red Hat when I downloaded CentOS?” Well, that is because CentOS and Red Hat are almost identical with a few exceptions that we don’t need to concern ourselves with for the purposes of this book. And, to be honest, I don’t think it really matters what you select for the Version since we will be loading CentOS anyways.

Since this will be our development environment, it is recommended to select as much memory for our Virtual Machine (VM) as possible. Just remember that we will be running Linux and Windows (and our new OS) at the same time. So, we will need to make sure that all systems are allocated enough memory so that the computer doesn’t begin performing slowly. But, your memory size selection will all be dependent on how much physical memory your computer has. To give you a starting idea, the laptop I am using has 32GB of RAM. So, selecting 8GB (8192MB) is plenty for running my CentOS development environment without compromising the Windows host. For the Hard Disk, choose the second option - “Create a virtual hard disk now” - and click the “Create” button.



On the next screen, we will need to choose a location on our file system where we want to store the VM we are currently creating. Before we select a File Location, we will need to select “VHD (Virtual Hard disk)” for the “Hard disk file type”. Then, we will click the icon to the right of the File Location field to select where we want to create and save this file. Since most computers now come with a fairly decent amount of disk space, you can set the File Size to whatever makes sense for you. If you aren’t planning on doing a whole lot more with your CentOS development environment aside from creating your new / cool operating system, you can stick with the proposed 8GB. However, if you plan on cross-compiling your OS to run on other architectures (such as on the Raspberry Pi as shown in [Chapter 12](#)), you will need to specify a larger file size. As shown below, I chose to go with 16GB since the cross-compiler from [Chapter 12](#) requires an additional 3.5GB. Plus, if I decide to create other cross-compilers, those too will require more space as well. When you are done, click the “Create” button to create the VM and close the modal window.

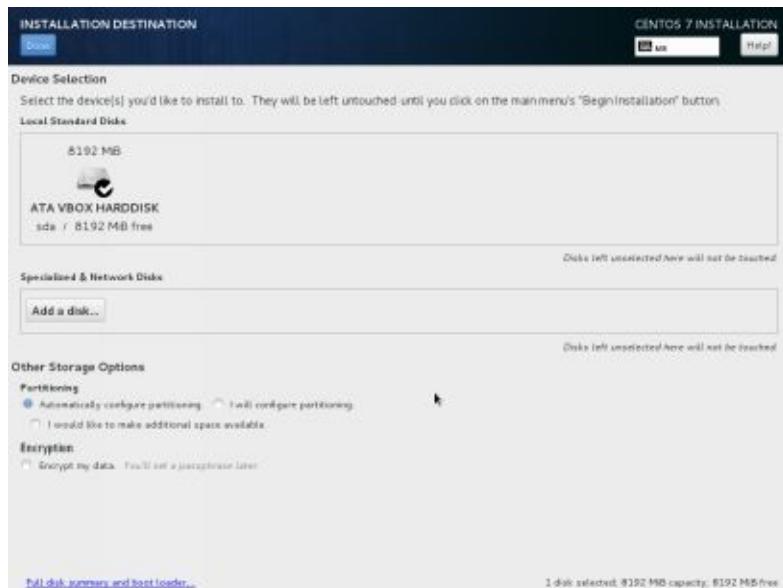


At this point we should see “CentOS” (or whatever we named our development environment) in the VirtualBox list on the left. Go ahead and click to select CentOS from the list on the left and click the “Start” button from the toolbar. This will launch the VM where the first thing we will be asked is to select a start-up disk. To do that, click the folder icon with the green arrow, change directories to the location where you downloaded the CentOS ISO and select it.



With our ISO selected, we will click the “Start” button to proceed.

As soon as our VM is started, we will click inside the VM to take control of it. Using your arrow keys, move up to select “Install CentOS” and press the enter key. At this point, you will be asked to provide input in the form of a wizard for selecting things such as your language and dialect of choice, date & time, keyboard, etc... You will also be asked to select an installation destination. To do that, click on the “INSTALLATION DESTINATION” option under “SYSTEM”. VirtualBox will select the first and only drive in the list. If you have a need for an additional disk, go for it. Otherwise, just click the “Done” button in the upper-left corner to select the only drive you have for your VM and return to the previous screen.



When you are ready to move on, click the “Begin Installation” button in the lower-right corner and you will be taken to a screen that not only begins the installation process, but it also asks you to create a root password and a secondary user. You will need to provide at least a password for root before completing the installation process. Just make sure you provide something you will remember. If you use something weak for your password (such as “root”), you will need to press the “Done” button in the upper-left corner twice to confirm it as indicated by the yellow bar at the bottom of your screen. Make sure you also create a secondary user at this time as we will be using that user for the exercises remaining in this book.

While CentOS is being installed, press the Ctrl key to the right of your spacebar to release the mouse from the VM and give control back to Windows until the installation has completed. When the installation has finished, click back inside your VM to move your mouse control back to it and press the “Finish configuration” button in the bottom-right corner. Afterwards, you will need to click the “Reboot” button to restart CentOS at which point you will be dropped into a login prompt where you will enter the username and whatever password you provided during installation. Do not log in as “root” at this time.

Installing GNOME

In order to do some development, there are several tools that we will need to install. Before we can do that, we will first need to configure our network connection since the tools we will be using will need to be fetched directly from the web.

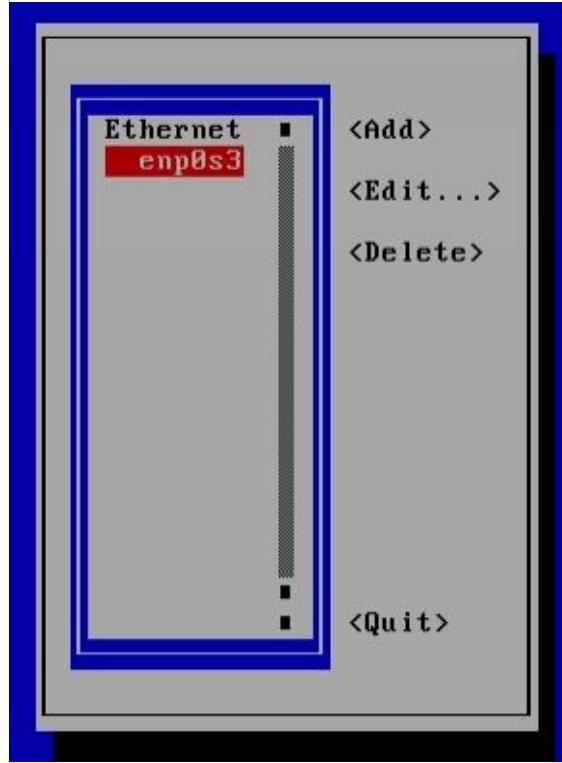
There are a few ways to do this. But, thankfully CentOS provides a very simple mechanism for configuring our network connection. At the command prompt, type the following command to open the Network Manager in GUI Mode.

```
# sudo nmtui
```

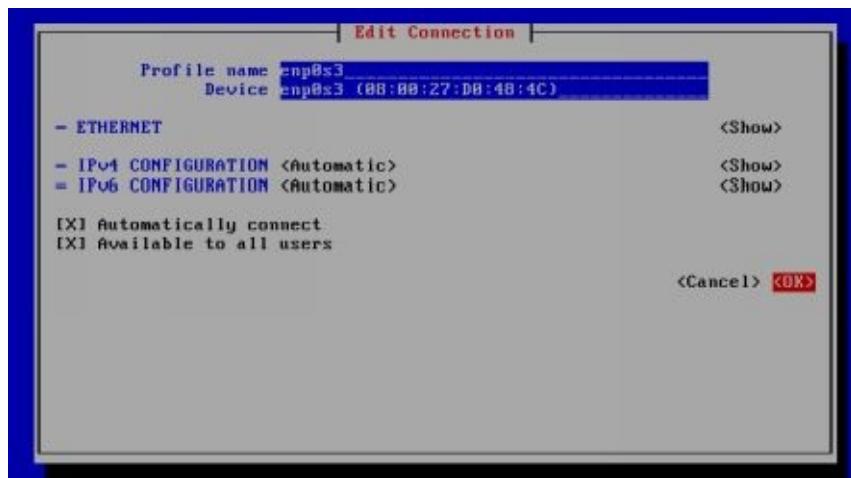
Note: Do not type the leading # sign. It is only there to indicate that this command is to be entered at the console prompt.



Next, press the enter key to select “Edit a connection”. More than likely you should only have one network interface available in the list on the next screen. Regardless, use your arrow keys to select the network interface you will be using and press the enter key to choose it.



Since you probably don't care how VirtualBox connects to your host computer for accessing the Internet, you can leave "IPv4 CONFIGURATION" set to "Automatic" and will need to check "Automatically connect" in order to enable DHCP. To check the box next to "Automatically connect", use your arrow keys to navigate down until it turns red. Then, press the spacebar to select it. After that, use the arrow keys again to navigate to "OK" and press the enter key to return to the network interface selection screen. Move the cursor down to "Quit" so that it is red. Then, simply press the enter key to quit the Network Manager.

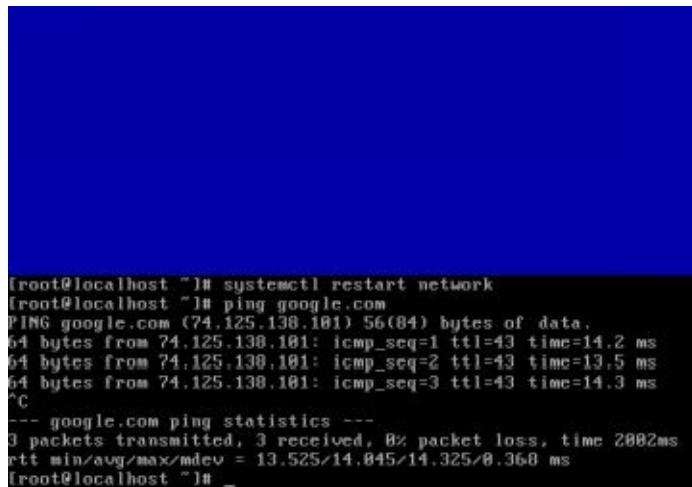


Now that your network connection has been configured, it is time to make CentOS aware of it. You can do that by restarting the network service using the following command:

```
# sudo systemctl restart network
```

You can verify that your network connection is working by executing the following command which should result in some results and no errors.

```
# ping google.com
```



```
(root@localhost ~) # systemctl restart network
(root@localhost ~) # ping google.com
PING google.com (74.125.130.101) 56(84) bytes of data.
64 bytes from 74.125.130.101: icmp_seq=1 ttl=43 time=14.2 ms
64 bytes from 74.125.130.101: icmp_seq=2 ttl=43 time=13.5 ms
64 bytes from 74.125.130.101: icmp_seq=3 ttl=43 time=14.3 ms
^C
-- google.com ping statistics --
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 13.525/14.045/14.325/0.368 ms
[root@localhost ~] #
```

Press Ctrl+C to stop pinging the Google servers and to drop you back to the prompt.

Even though it is possible to do all of your OS development from the console, I find it easier to do development from a GUI (Graphical User Interface) environment. There are a few options to choose from when selecting a GUI. But, for the exercises in this book, I will be working with and referring to the GNOME Desktop Environment which you can install by executing the following command:

```
# yum -y groupinstall "GNOME Desktop"
```

Depending on your Internet connection speed, it will probably take a little while for GNOME to install. For me, it took about three minutes to download everything and about an hour to install it all. So, go have yourself a coffee & a sandwich and check back later. When you return, your screen will probably be black. If it is, simply click inside the VM and press one of the arrow keys. You should now see your screen again. If everything is still installing, you can press the Ctrl key to the right of your spacebar just like before to return control back to Windows where you can do other things while the installation finishes.

After the installation has completed and you are back to a command prompt, type the following command to start the GNOME desktop.

```
# startx
```

Every time you reboot CentOS, you will need to run “startx” to get back into the GNOME desktop. However, if you want to have GNOME automatically load every time you restart, you can do that using “targets” (instead of “runlevels” from previous versions). The following commands will tell CentOS to load the GNOME GUI on system start from now on.

```
# sudo rm /etc/systemd/system/default.target
# sudo ln -sf /lib/systemd/system/graphical.target /etc/systemd/system/default.target
```

When GNOME loads up for the very first time, you will be presented with the Gnome-initial-setup where you will be asked to select your language of choice, input sources, and whether you want to “connect to your existing data in the cloud”. After you click the “Start using CentOS Linux” button at the end of the setup, you will be ready to load the tools needed for building your own operating system.

Preparing CentOS and the VM

Before we move on, it is recommended that we update CentOS so that everything from this point on works as expected. To do that from inside of GNOME, click on **Applications > Utilities > Terminal**, then run the following command to update your entire system.

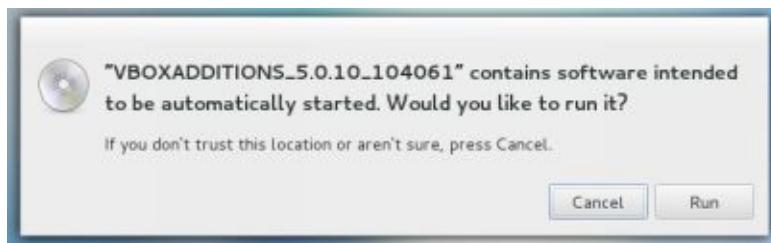
```
# sudo yum -y update
```

Note: This will probably take several minutes to run. For me, it took around thirty minutes to complete.

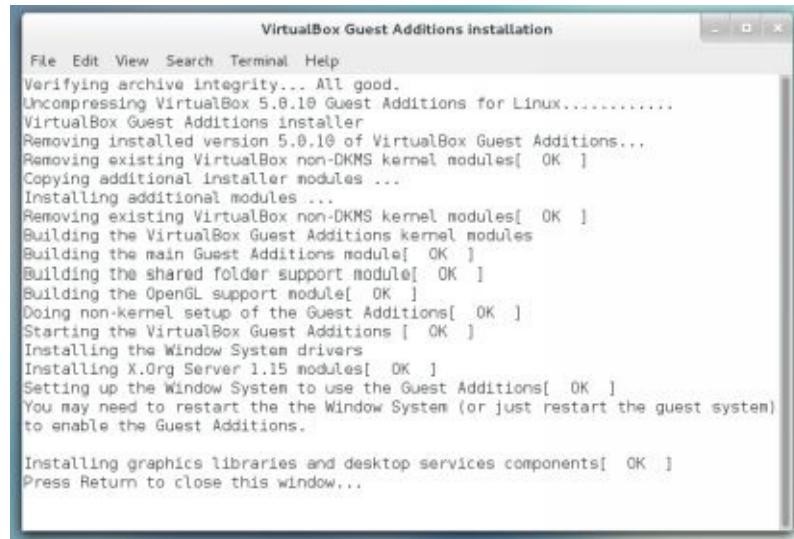
Next, we will need to install the VirtualBox Guest Additions which will allow us to do things later on such as dragging and dropping files between our Windows host and CentOS guest as well as copying and pasting text between the two. To do that, the first thing we will need to do is execute the following commands to install the kernel development tools that will be needed by the guest additions installer.

```
# sudo yum -y install kernel-devel
# echo export KERN_DIR=/usr/src/kernels/`uname -r` >> ~/.bashrc
# reboot
```

After CentOS has restarted, we will need to install the VirtualBox Guest Additions by clicking Devices on the VM menu bar and selecting “Insert Guest Additions CD image...” Inside the VM, you should see a message confirming that you trust the VBOXADDITIONS CD. Go ahead and click the “Run” button to confirm this.



Once Guest Additions have been installed, you will see a message telling you everything was successful and that you now need to restart the guest system (CentOS). It is a good idea that you also restart Windows at this point too (just in case).



Troubleshooting VirtualBox Guest Additions

For some reason, you might experience times when you can no longer drag/drop or copy/paste between your CentOS guest machine and your Windows host machine. When this happens, you can usually redo the steps above to re-install the VirtualBox Guest Additions and everything will start working again. However, I have experienced times when CentOS will complain about the Guest Additions with a message along the lines of “mount: unknown filesystem type ‘iso9660’”. If you encounter this, don’t worry. There is a manual way of remounting the VBOXADDITIONS CD in CentOS so that you can re-install the Guest Additions. Below are those steps.

```
# sudo mkdir /media/VirtualBoxGuestAdditions/  
# sudo mount /dev/cdrom /media/VirtualBoxGuestAdditions
```

You should get the following error at this point:

```
> mount: unknown filesystem type ‘iso9660’
```

```
# ls /lib/modules/
```

Make note of the “generic” folder with the highest version number here.

```
# insmod /lib/modules/[replace me with the version number above]-  
generic/kernel/fs/isofs/isofs.ko  
# sudo mount /dev/cdrom /media/VirtualBoxGuestAdditions  
> mount: block device /dev/sr0 is write-protected, mounting read-only  
# sudo /media/VirtualBoxGuestAdditions/VBoxLinuxAdditions.run
```

Preparing the Development Environment

At the beginning of this chapter, we downloaded and installed the minimal version of the CentOS ISO. Because of that, there are a few development tools missing that we will now need to install.

The first is the GNU Compiler Collection also known as “**gcc**”. This is the tool that we will be using to compile our C code into machine code.

Since we are focusing (for now) on compiling our OS to run on the x86 architecture, the next tool we will need to install is the **Netwide Assembler** also known as “**nasm**”. This is the tool we will use to compile our Assembly code into machine code.

The third tool we will need to install is called “**tk**” which is a graphical toolkit for the Tcl scripting language. This is needed by one of the dependencies needed for building our ISO.

Even though we can install gcc, nasm, and tk separately, we can also knock them all out with a single line. To install them all at the same time, run the following command:

```
# sudo yum -y install gcc nasm tk
```

When it comes time for us to [cross-compile our operating system for other architectures](#), there are a few other dependencies we will need. So, let's go ahead and install them now.

```
# sudo yum -y install gcc-c++ glibc-devel glibc-static libstdc++* glibc.i686
```

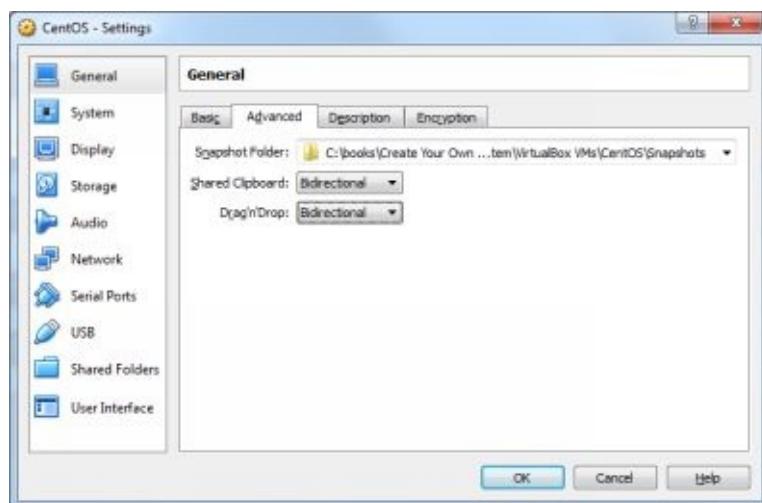
Next, we will need to manually download and install **xorriso** which is needed by the utility we will be using to create our ISO as mentioned above. You can find the latest version of xorriso on the gnu website at <http://www.gnu.org/software/xorriso/#download>. Run the following commands to install xorriso. If you download a different version of xorriso, just be sure to swap out “1.4.2” with the appropriate version number found on the gnu website.

```
# wget http://www.gnu.org/software/xorriso/xorriso-1.4.2.tar.gz
# tar -zvxf xorriso-1.4.2.tar.gz
# cd xorriso-1.4.2/
```

```
# make  
# sudo make install
```

That's it! We now have all the tools we will need for developing and compiling our very own operating system. But before moving on, there is still one more thing that we can do to make our development a little easier. Later on we will have the need to copy the ISO we will create for our OS from the CentOS VirtualBox VM to our Windows computer where we will create a separate VirtualBox VM for testing it. Plus, there will be times where we will want to copy other files and text between our VirtualBox VM and our host machine.

In order to enable copying and pasting between our VirtualBox VM guest and Windows host, we will first need to power off our CentOS VM. To do that, simply click the X in the upper-right corner and tell it to "Power Off". Once the VM window has closed, select CentOS from the list in the VirtualBox Manager and click Settings on the toolbar. Under the General section, click the Advanced tab and change both "Shared Clipboard" and "Drag 'n Drop" to "Bidirectional" as shown below.



After you have made those changes, click the OK button to return to the VirtualBox Manager. From there, you can restart your CentOS VM by selecting the VM from the list on the left and clicking the Start button. If you ran the "ln -sf" command earlier in the "[Installing GNOME](#)" section, CentOS should load you back into the GNOME desktop. If it doesn't, you can return by simply executing the "startx" command.

Congratulations! We now have an environment setup that will allow us to develop our very own operating system.

0x06 Bootstrapping with the Bootloader

When you power on a computer, the first thing to run is what is known as the “Basic Input Output System” (BIOS). The BIOS is responsible for verifying that all of the hardware has powered on successfully and is ready for use. This test is known as the “Power-On Self-Test” (POST). In most systems that do not pass the POST, a series of beeps are played that – depending on their tone and frequency – indicate what has gone wrong (or right) and why the system has halted (or started). You can do a Google search to find a list of beep-codes that are specific to your platform or you can use the list of codes on Wikipedia found at https://en.wikipedia.org/wiki/Power-on_self-test.

On an x86 system, the CPU is pre-configured to look for the last sixteen bytes of the 32-bit address space (0xFFFFFFFF0) for a “jump” instruction to the address in memory where the BIOS has copied itself. When found, the system will kick off the BIOS which will initiate the POST test.

Once all hardware has powered on and the POST has passed successfully, the BIOS will look for the operating system and will turn control over to it if found. The first piece of the OS that gets loaded into memory is known as the “Master Boot Record” (MBR). The MBR is the first sector found on a bootable device and is only 512 bytes in size: 446 bytes for the primary bootloader and 64 bytes for the partition table that describes the primary and extended partitions.

When the BIOS has successfully located the MBR that contains bootable code, it will take this code and load it into memory starting from physical address “0x07c0” at which point the CPU will jump to and execute the code. The code that gets loaded and executed is known as the “bootloader” (short for “bootstrap loader”). If all goes according to plan, the bootloader will then load the kernel into physical address 0x10000000 which loads the rest of the operating system. We will learn about the kernel in the [next chapter](#). The main thing to know here is that the bootloader contains the entry point into our system.

To help us better understand how the bootloader works, we will now take a quick look at a simple bootloader used on x86 architectures. Even though it is possible to use this for bootstrapping our own operating system, we will not be using this bootloader going forward. Instead, we will opt for using already provided bootloaders which we will learn about shortly. For now, let’s take a look at the bootloader code.

With your CentOS environment running in VirtualBox and booted into the desktop, click on **Applications > Accessories > gedit** to open the text editor we will be using for writing our code. Then, click on **File > New** (or press Ctrl + N or click the white-paper icon on the toolbar) to open a new / blank file. Type the following code into that new file and save the file as “**bootloader.asm**”. You can save the file where ever you want. But for the purposes

of this book, we will save our file into a new folder on the Desktop called “**bootloader**”.

The following code will boot the system and print a simple “Hello world!” message to the screen one character at a time and then halt the system.

bootloader.asm

bits 16

start:

```
mov ax, 0x07c0
add ax, 0x20
mov ss, ax
mov sp, 4096
```

```
mov ax, 0x07c0
mov ds, ax
```

```
mov si, msg
call print
cli
hlt
```

data:

```
msg db 'Hello world!', 0
```

print:

```
mov ah, 0x0e
```

.printchar:

```
lodsb
cmp al, 0
je .done
int 0x10
```

```
jmp .printchar

.done:
    ret

times 510 - ($ - $$) db 0
dw 0xaa55
```

Since [Chapter 3](#) gave us an introduction to the Assembly programming language and because we will cover it quite extensively in the upcoming section about creating our entry point, I will not go into the details about what the code in this bootloader does. However, one thing I believe is worth mentioning in this bootloader code is the fact that it begins with “**bits 16**” which indicates the code is intended to be executed in 16-bit mode as opposed to the 32-bit mode the rest of our operating system will be running in.

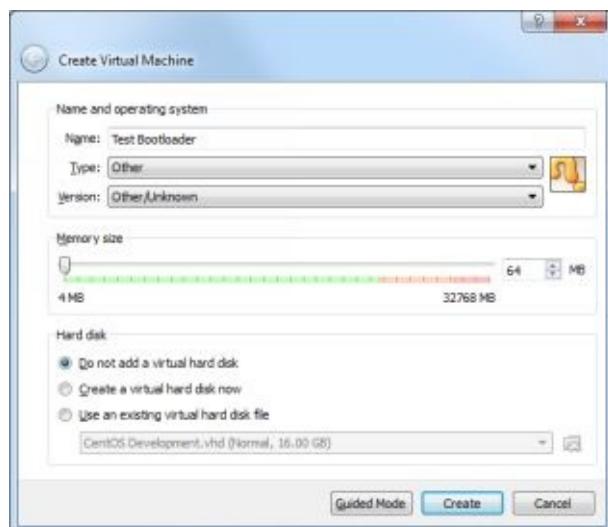
The reason for this is because when all x86 architectures boot up, they start out running in what is called “real mode” (where the bootloader runs) and eventually get switched to “protected mode” where everything else gets ran. The main difference between real mode and protected mode is that in real mode, all applications run within the same memory at the same time. Therefore, any running applications can access the memory space of all of the other running applications. As you can imagine, this would create a security nightmare. Therefore, after the bootloader has finished doing its work (loading the system and turning over execution to the kernel), the system then switches into protected mode where applications are physically isolated from each other due to the nature of running in different address spaces.

Even though we will not be using this particular bootloader for our operating system, I at least wanted to provide you with the code so that you would have it in the event you want to create your own bootloader for other projects. And, since we already have the code, we should also take a look at how to compile and test it.

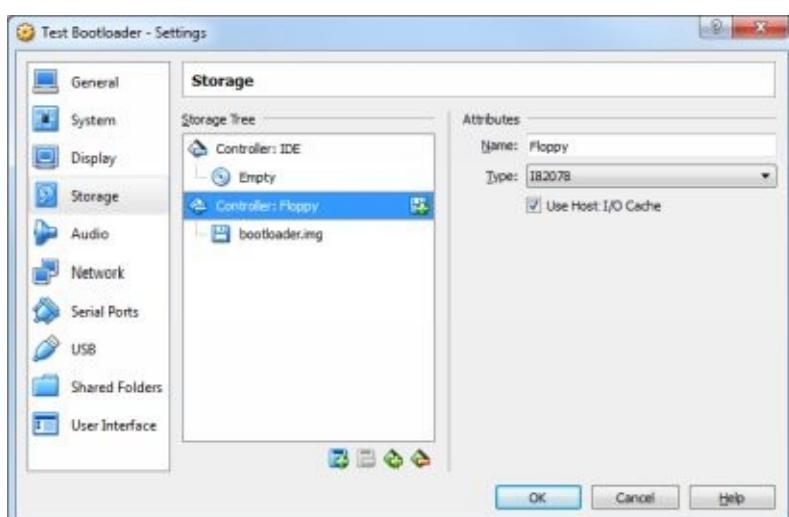
In order for us to test the above bootloader, we will first need to compile it. To do that, open a terminal in your CentOS environment, use the “cd” command to change directory into the location where you saved the **bootloader.asm** file, and execute the following command.

```
# nasm -f bin -o bootloader.img bootloader.asm
```

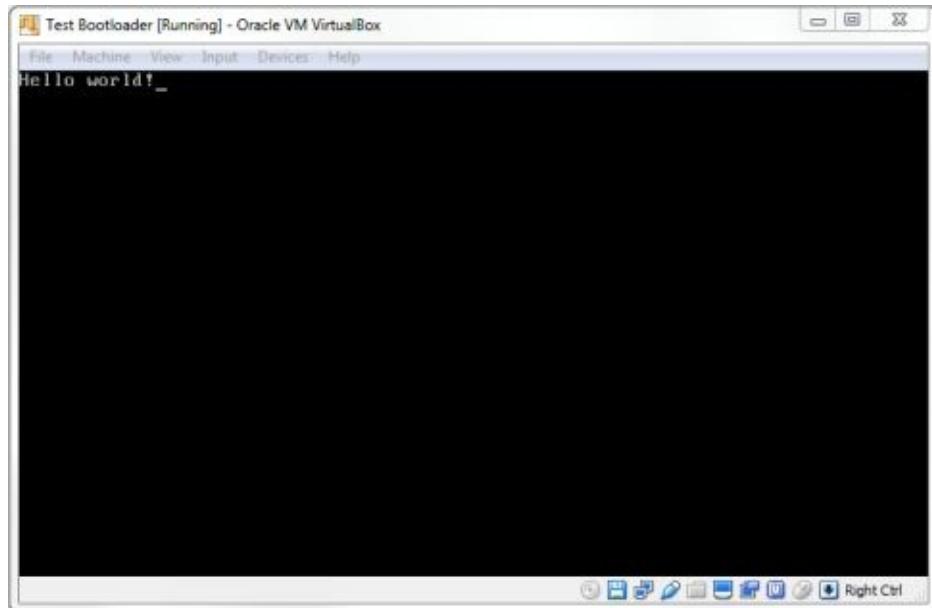
Within the same folder that contains your **bootloader.asm** file, you should now also see an additional file called “**bootloader.img**”. Next, copy the newly created **bootloader.img** file from your CentOS environment over to your Windows environment. Then, inside VirtualBox, go to **Machine** > **New...** to open the “Create Virtual Machine” dialog. From there, enter something such as “Test Bootloader” for the “Name” field and select “Other” and “Other/Unknown” for the “Type” and “Version”. Select “Do not add a virtual hard disk” and click the “Create” button.



After you click the “Create” button, you should now see a new virtual machine in the list on the left-side of VirtualBox. At this time, we will need to create a bootable drive to run our bootloader from. To do that, select the new VM from the list and click the “Settings” button on the toolbar. Once the settings dialog has appeared, select “Storage” from the list on the left. Next, right-click in the white space under the word “Empty” in the “Storage Tree” and select “Add Floppy Controller”. Then, right-click on “Controller: Floppy” from the list and select “Add Floppy Drive”. This will pop up a dialog window where you will need to click the “Choose disk” button, navigate to, and select the **bootloader.img** file you copied over from your CentOS environment.



Once you have everything selected (and matching the above screenshot), click the “OK” button to apply your settings and return to VirtualBox. From there, double-click on your “Test Bootloader” VM to start it. Once your VM starts up, you should see the words “Hello world!” printed on the screen.



Creating the Entry Point

At the beginning of this chapter, we learned that once the bootloader has executed, its final job is to load the kernel into physical memory address 0x10000000 which will load the rest of the operating system. But, in order for our bootloader to know where the entry point into the kernel is, we will have to tell it by building out the actual kernel.

Even though the entire kernel can either be written completely in Assembly or completely in C, we will go with a mixed method here by separating the actual entry point from the rest of the kernel. For that, we will write the entry point in Assembly and leave the rest for the C programming language which we will drill into in the [next chapter](#). The reason for separating the entry point from the rest of the kernel and including the entry point in this chapter as opposed to the next is so that we can expand on what we have learned so far about where each piece of our operating system gets loaded into memory. With that said, let's jump right into building the entry point of our kernel.

Just as we did in the previous section when creating the file for our bootloader, make sure you are in the CentOS development environment running in VirtualBox and click on **Applications > Accessories > gedit** to open the text editor we will be using for writing our code. Then, click on **File > New** (or press Ctrl + N or click the white-paper icon on the toolbar) to open a new / blank file. Type the following code into that new file and save the file as “**start.asm**”. Whereas we created our bootloader in a folder on our Desktop called “bootloader”, we will create the remaining files in a new folder on the Desktop called “**my_os**”. This will be the location where we save all of our files, compile our operating system, and build the ISO which we will use later for testing our operating system. Since we will not be using the bootloader we created in the previous section, there is no need to have it in the same folder as the rest of our code. But, it is OK if you do choose to put them all in the same folder for easy reference.

start.asm

```
bits 32
global _start
extern main

section .text
_start:
    cli
    mov esp, stack
    call main
```

```
hlt
```

```
section .bss
resb 8192
stack:
```

As mentioned before, the code above will act as the entry point into our operating system. It begins by notifying the NASM compiler that it should compile this code for 32-bit processors as indicated by the “**bits 32**” directive on the first line (as opposed to the “**bits 16**” directive in our bootloader code from before). Although this line isn’t required – the compiler will figure it out on its own, it is still best practice to explicitly declare it here. This can make it easy to identify places that need to be changed later on when compiling the operating system to run on different architectures.

The second line (“**global _start**”) should already look familiar. As mentioned in [Chapter 3](#), this line tells the linker (which we will learn about in [Chapter 8](#)) where the program execution begins. From [Chapter 3](#), you will recall that we placed this directive just after the “**section .text**” line and just before the “**_start:**” line. We could have done the same here, but I wanted to demonstrate that we can actually place the **global** directive in different locations since the linker will locate it regardless of its location.

The third line (“**extern main**”) in the code above indicates that the “**main**” procedure can be found in an “**external**” file. As learned earlier in this book, the **main** function is the entry point into all applications. However, since we already have an entry point into our operating system (which we declared in our Assembly code using “**_start**”), we can actually rename the **main** function here to anything we want. But, to keep with what we have already learned in this book (and for following best practice), we will leave our procedure named “**main**”. Whatever you name this, though, keep it in mind as we will see it again a few lines down in our entry code.

The next line (“**section .text**” - which we also learned about in [Chapter 3](#)) tells the compiler that the following lines will be the actual code that handles the processing. The first thing we do in this section (aka “function” or “procedure”) is to disable (or **clear**) interrupts by using the “**cli**” command. This is done because other instructions such as the “**hlt**” (**halt**) instruction can awake the CPU, telling it to do other things – sometimes unintended.

After we have disabled interrupts, we then allocate some memory for the stack and point the stack pointer (**esp**) to it. The **stack** itself is defined as the last function in the file (indicated by “**stack:**”). As you will notice, there is a colon after **stack** which indicates it

is a function. But, since there are no other instructions after it, this indicates that the function is currently empty. That is because we haven't added anything to the stack as of yet. We are just allocating memory for it and mapping it to the stack pointer.

The next line ("call **main**") is where we tell the system that it should now execute the instructions found in our "**main**" function. Again, this function will be provided by our kernel's C code which we will define in a separate file in the [next chapter](#). Because the function is in a different file, we had to alert the compiler to this by using the "**extern**" instruction earlier. That is why I asked you to make note of whatever you chose to name this function. Plus, you will be using this same name in your **kernel.c** file in the [next chapter](#). So, again, it is best to just stick with calling it "**main**" for now.

After we have told the system to call our **main** procedure, the only instruction we have left is to tell the system to "**halt**" as defined by the "**hlt**" instruction. The reason we want our system to halt for now is because we have nothing left for the CPU to process. Later on we will omit this instruction because we will want the CPU to continue processing further commands (such as user input). So, there will be no need for the system to halt at that time.

After the **.text** section, you will find the **.bss** section. As explained in [Chapter 3](#), this section is where we define our variables. In this particular example, we use the "**resb**" instruction which reserves 8KB of memory for our stack. The "**res**" part of the instruction represents "**reserve**" while the "**b**" part of the instruction represents "**bytes**". You could have just as easily used "**resw**" to reserve a "**word**", "**resd**" to reserve a "**double**", or "**resq**" to reserve an array of ten reals. But, "**resb**" is all we need for our purposes.

GNU GRUB

That's it! We now have ourselves a nice shiny bootloader. However, since we want our operating system to boot on various devices using a specific kernel configuration, we will be using a bootloader from the GNU Project called “**GNU GRUB**” (short for “**GNU GRand Unified Bootloader**”) instead of the bootloader we learned to build earlier in the chapter. Not only will this bootloader allow us to bootstrap the kernel, but it will also allow us to boot one of multiple operating systems should we decide later on to add multiple / different OSes.

In order for us to use GRUB, we will need to make a few additions to our kernel’s entry code so that it follows the multiboot specification that GRUB requires. To do that, we will only need to add four additional lines to the beginning of our **.text** section which are indicated in bold below.

boot.asm

```
bits 32
global _start
extern kernel_early
extern main

section .text
    align 4
    dd 0x1BADB002          ; magic
    dd 0x00                  ; flags
    dd - (0x1BADB002 + 0x00) ; checksum
```

_start:

```
    cli
    mov esp, stack
    call kernel_early
    call main
    hlt
```

```
section .bss
resb 8192
```

stack:

The first of these additional lines (“**align 4**”) is a directive that allows you to enforce alignment of the instruction or data immediately after the directive. Whatever number follows the **align** directive must be a power of 2 (i.e. 2, 4, 8, 16, 32, ...) and must not be greater than the default alignment of the segment which we defined by saying that this should compile for a 32-bit system (i.e. “**bits 32**”). Since the system is expected to be 32 bits (i.e. 4 bytes), we must use alignment values that are less than or equal to 4.

The next line we added for the header (“**dd 0x1BADB002**”) is a double-word (i.e. 4 bytes) that indicates a “magic” field that is required by GRUB. The next line (“**dd 0x00**”) is where we define “flags” that are needed by GRUB. Since we will not be needing any additional flags at this time, we will simply set this field to zero. The third line (“**dd - (0x1BADB002 + 0x00)**”) is a checksum that, when added to the values of the **magic** and **flags** fields, should always be equal to zero.

Aside from the additional header code, everything else in the code above will be the same as before except for one last thing. As you will see near the top of the code, I have added the line “***extern kernel_early***” as indicated by italics. Since it is an **extern** directive, you already know that the function will be located in an **external** file. Also, just before our **call main** line, you will see another line that calls the **kernel_early** function which we will define in our **kernel.c** file in the [next chapter](#).

Although these lines aren’t necessarily required for our operating system, I wanted to include them anyways as a way to demonstrate how we can name our functions differently as well as include multiple external calls. The reason for naming this procedure “**kernel_early**” and calling it before our call to “**main**” is so that we will have a way to execute instructions that need to be run prior to our kernel actually doing its thing. In this case, we will use the **kernel_early** function for initializing our terminal which we will use from within the kernel’s **main** function. Other times we may also want to include other instructions in our Assembly file between the two **call** instructions for doing things such as re-enabling interrupts (using the “**sti**” instruction – short for “**start interrupts**”), allocating more memory, or switching between real mode and protected mode as explained earlier.

Compiling the Entry Point

Now that we have our kernel's entry point, it is time to compile it. This is where we will use the NASM Assembly compiler we learned about in [Chapter 5](#). To compile our entry point, click on **Assembly** > **Utilities** > **Terminal** which will open a prompt where the following commands need to be executed.

```
# cd ~/Desktop/my_os/  
# nasm -f elf32 start.asm -o start.o
```

The first command above will change directories into the “**my_os**” folder we created on our Desktop that contains the source code for our operating system. The second line tells **nasm** to compile our Assembly code into machine code. The “**-f**” flag tells **nasm** what type of architecture it should compile the code for. Since we are building an operating system intended to be run on a 32-bit architecture, we will pass “**elf32**” as the architecture output format. The “**elf**” part stands for “**E**xecutable and **l**inkable **F**ormat” and the “**32**” part should be pretty evident at this point. With that said, you can also just pass “**elf**” as the output type and **nasm** will assume you meant “**elf32**”.

For a list of other output formats we can compile Assembly code into using **nasm**, we can execute “**nasm -hf**” as described in the [Appendix](#) at the end of this book.

If everything worked correctly, you should be dropped back to the command prompt with nothing output to the screen. To verify that your Assembly file did compile correctly, run the “**ls**” command and you should see a file called “**start.o**”.



At this point we now have a multi-boot loader that will boot our kernel which we will build in the [next chapter](#). But before moving on, there is one last thing I would like to mention. Since different processors utilize different architectures, Assembly code can very much be architecture specific. That means, what works for one architecture might not / most likely does not work for another architecture. This is especially true when making system calls. Keep this in mind when developing operating systems for embedded devices, especially those devices built for the *Internet of Things*.

0x07 Welcome to the Kernel

If the operating system is the heart of a computer, you can think of the kernel as the brain. It is the central piece of the operating system that loads first, always remains in memory, and is responsible for managing low level hardware such as the storage device, memory, processor, and network card. Any time software further up in the stack wishes to communicate with low level hardware, it does so by sending requests to the kernel which in turn get passed on to the hardware. Basically, it is also the job of the kernel to make sure that every application that gets ran is allocated sufficient memory and CPU time.

When dealing with OS kernels, there are two types available. The first type is what is known as a “monolithic kernel”. This type of kernel contains the majority of the operating system, including drivers to low level hardware as mentioned above. Since monolithic kernels contain most / all of the underlying operating system, these types of kernels are typically loaded into a protected area of memory so that other applications cannot tamper with them while in use. This protected - see privileged - area is known as the “kernel space”. This space is separated from the space where applications are run, which is known as the “user space”.

The second type of kernel is what is known as a “microkernel”. This type of kernel is typically only responsible for two things: memory management and CPU management. All other functions get moved outside of the kernel which allows them to live and die (i.e. “crash”) without bringing down the entire system (such as what happens when a device driver or other piece of software fails in a monolithic kernel). Where monolithic kernels run the entire operating system within the kernel space, microkernel systems only run the most critical activities in the kernel space and everything else is run in the user space.

Unlike monolithic kernels, microkernels are quite difficult to develop and debug. Operating systems such as Linux utilize monolithic kernels because they are easier to develop. Therefore, the kernel we will be building will be monolithic as well. By the way, do not get “monolithic” in this context confused with “non-modular”. Monolithic kernels *do* support modularity such as the case of device drivers. In this case, modules are attached to the monolithic kernel which in turn increases the overall size of the kernel, yet it still remains monolithic since it (the kernel) and the modules are still run in the kernel space.

For an easy way of understanding how the kernel works, the kernel we will be building in this chapter will only have three functions: 1) initialize and clear the screen 2) print “Hello world” 3) return control to the bootloader which will halt the CPU. In [Chapter 11](#), we will build on this functionality by extending our kernel to handle other things such as accepting user input via the keyboard.

Before we can start printing stuff to the screen, the first thing we need to do is declare a few variables. Since we will be referencing these variables from multiple functions, we will need to define them globally which we can do by adding them directly to the top of our code (i.e. outside of any functions – identified by curly braces { and }).

In order to print text to the screen like we want to do in our operating system, we will need to begin by gaining access to the physical memory that has been allocated for our video buffer. To do that, we will need to create a pointer to the address space which is always located at address **0xb8000** in protected mode for color monitors and **0xb0000** for monochrome monitors. Since this is a text buffer, our variable will need to be a pointer of type **char** like so:

```
static char* const VGA_MEMORY = (char*)0xb8000;
```

As you can see above, the allocated memory space for video in our operating system will begin at **0xb8000**. This buffer is capable of holding twenty five lines with each line consisting of eighty ascii characters. Therefore, the next two variables we will need to define will contain the width and height of our terminal.

```
static const int VGA_WIDTH = 80;  
static const int VGA_HEIGHT = 25;
```

To understand how the video memory buffer is used, it is worth noting that the characters in our video buffer are represented by two bytes aligned as a 16-bit word. The lower byte is the actual character itself while the upper byte contains the format of the character such as its color, blinking, character set, and so on.

That's it for our global variable declarations. In [Chapter 10](#) we will add more global variables when it comes time to start the architecture library for our operating system. But for now, these are all the variables we will need to define in order for our operating system to function. In fact, since we are only using these variables within one function at the moment, we could have just as easily placed them inside that single function. But, since later on we will be using these variables from multiple functions, it is wise to define them globally now.

As you will recall in the [previous chapter](#), we added a line in our assembly code for **call kernel_early**. The idea there was to provide us with a place to do things (such as initializing and clearing the screen) that we want to happen before jumping into the functionality of the kernel itself. Even though we have already initialized our screen when we created a pointer to the video memory address space, we could have done other things

inside of this function such as painting the background a different color or drawing a logo. Regardless, since we added that line to our Assembly code, we will now need to create that function here. For now, it will just be an empty function as follows:

```
void kernel_early(void) {  
    // do something here  
}
```

With that out of the way, the next thing we will need to do is define our **main** function which we learned in a previous chapter is the entry point into all applications. Since our “real” entry point was defined in our Assembly file (remember “**global_start**”?), we could have named our **main** function anything we wanted. But, for illustration purposes, we chose to stick with naming it “**main**” which is what we will also use in our kernel code like so:

```
int main(void *) {  
    return 0;  
}
```

As you can see, our **main** function has been defined as type “**int**” which means it is expected to return an **integer**, hence the “**return 0;**” inside the function. Since the function isn’t expecting any incoming parameters, the “**void**” statement between the parentheses indicates an empty argument list. The **void** could have been omitted, but I chose to include it here as best practice. Also, instead of setting the return type as **int** and returning **0** (zero) at the end, we could have also omitted the **return 0;** and substituted **int main(void *) {** with **void main(void *) {**. But, again, I went with this approach as it is best practice. Plus, using **void main(void *) {** instead of **int main(void *) {** doesn’t work with some compilers. Also, the **return 0;** line indicates that the system terminated without any abnormalities. If the system terminates unexpectedly for any other reason, you can instead return a non-zero value where the value can be mapped to a list of error codes that describe what went wrong with the system.

Moving on. The first piece of code we will need to define at the top of our **main** function will be the string of text that we plan on printing to the screen – “**Hello world**” in this case. Since this variable will never change, we can define it as a **constant** as shown below. This line (and all following lines) will need to be inserted inside the **main** function before the **return**.

```
const char *str = "Hello world";
```

In order for us to set each character in our video memory, we will need to define a new variable that will be the place holder for our current position within the video memory buffer. Likewise, we will also need to define a second variable that will act as the place holder for the current character position when iterating over the text string that we will be printing to the screen one character at a time.

```
unsigned int i = 0; // place holder for text string position  
unsigned int j = 0; // place holder for video buffer position
```

As I just mentioned, we will be iterating through our text string and printing the string to the screen one character at a time. To iterate over that string, we will use a “**while**” loop that will run until it has discovered its first null byte. Even though we never explicitly defined a null byte in our text string, a null byte does exist at the very end since there is nothing left in the variable after our text. This can easily be seen if you were to examine the code when it has been loaded into memory and dumped to the console in hexadecimal values.

In the C programming language, null character values in ascii are indicated by ‘\0’ (backslash zero). Knowing this, we can define our **while** loop like so:

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

Inside of that **while** loop is where we will pass each character of our text string to the video buffer. Since we learned earlier that each character is expected to be passed as two bytes (one byte for the character and a second byte for the format of that character), we will need to add two updates to the video buffer. Since we are iterating through each character of our text string one character at a time, we will need to increment our **i** variable (place holder for our text string position) by one each time we pass through our **while** loop. We will also need to increment our **j** variable (place holder for our video buffer position) while we are at it. However, where our **i** variable gets incremented by *one* each time we iterate (because we want *one* character at a time), our **j** variable will be incremented by *two* each time we iterate (because we are setting values in our video buffer *two* bytes at a time).

```
VGA_MEMORY[j] = str[i];  
VGA_MEMORY[j + 1] = 0x07;  
i++;
```

```
j = j + 2;
```

As mentioned before, the upper byte of each character in the video memory buffer is a format flag. In the code above, you will see that we have used “**0x07**” which tells the system that this particular character should be formatted with a light grey color. If you would like to use a different color for your text (or you would like to use a different color for every character), here is a list of colors available to you.

VGA Color Codes

Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
Light Grey	7
Dark Grey	8
Light Blue	9
Light Green	10
Light Cyan	11
Light Red	12
Light Magenta	13
Light Brown	14
White	15

So that you don’t have to attempt to create the entire kernel using the code snippets above, you can find the kernel code in its entirety below.

kernel.c

```
static char* const VGA_MEMORY = (char*)0xb8000;
```

```
static const int VGA_WIDTH = 80;  
static const int VGA_HEIGHT = 25;
```

```

void kernel_early(void) {
    // do some early work here
}

int main(void) {
    const char *str = "Hello world";
    unsigned int i = 0; // place holder for text string position
    unsigned int j = 0; // place holder for video buffer position

    while (str[i] != '\0') {
        VGA_MEMORY[j] = str[i];
        VGA_MEMORY[j + 1] = 0x07;
        i++;
        j = j + 2;
    }
    return 0;
}

```

That's it. You now have yourself a working kernel. But, before we get too excited, we need to make sure it will compile successfully. To do that, open a terminal window and change directory to where your kernel.c file is located. If you are using the same naming conventions as I have been throughout this book, your kernel.c file should be located in a folder on your Desktop called "**my_os**". If so, you can use the following commands to get to that location and compile the kernel.

```

# cd ~/Desktop/my_os/
# gcc -c kernel.c -o kernel -ffreestanding -m32

```

The arguments for the **gcc** (**GNU C Compiler**) are fairly self-explanatory. But, I will run through them real quick anyways. The **-c** flag tells the compiler that **kernel.c** is the input file that it will be compiling. The **-o** flag tells the compiler that the output / compiled file will be **kernel.o**. Unlike the first two flags, the **-ffreestanding** flag might be new to you. Basically, it tells the compiler that the standard C library may not exist and that the entry point may not necessarily be located at **main**. Remember, since we are building an entirely new operating system from scratch, we do not have the luxury of utilizing pre-

existing C libraries like we do when developing other applications. Because of this, every function we utilize we will need to write ourselves. Otherwise, when we deploy our app / OS, the code we write may or may not work. The **-m32** flag simply tells the compiler that it should compile our code for a 32-bit architecture.



You now have yourself a new and shiny kernel. By combining your new kernel with the entry point you built in the [last chapter](#), you are only a few steps away from having your very own operating system. In the [next chapter](#), we will bring these pieces together to build that operating system and test it out for the very first time.

0x08 Putting it all Together

Before we can test our operating system, we will need to link the object files we created for the entry point and kernel. To do that, we will create a linker file that **gcc** will use to link those object files into a single executable kernel that we can boot with. Once we are finished with that, we will then create a **Makefile** that will take care of compiling the Assembly entry point and C kernel along with using our linker file to create that executable / bootable kernel. This file will also take care of building an **ISO** file which we will use to create the virtual machine (VM) where we can test our operating system and admire it in all its glory. Now, I know that all sounds like a lot of work, but it really isn't. So, we will just jump right in.

In your CentOS environment, open **gedit** as before. Then, create a new file and save it as “**linker.ld**” in the same folder that you created your **start.asm** and **kernel.c** files (i.e. in the “**my_os**” folder on your Desktop). At the top of that file, add the following line which will, just like in our Assembly code in [Chapter 6](#), tell the system where the entry point (i.e. “**_start**”) to our code can be found.

```
ENTRY (_start)
```

Below that, we will add a place to define the sections that we listed in our Assembly code (i.e. “**.text**” and “**.bss**”). If we had also included a “**.data**” section as mentioned in an earlier chapter, we would also declare that section here as well. At that point, **gcc** would merge the **.data** and **.bss** sections into one and place them after the “**.text**” section that contains all of our executable code. Remember, the **.bss** section is where we define changeable variables and the **.data** section is where we store global variables such as initialized data and constants.

SECTIONS

```
{  
    . = 0x10000000;  
    .text : { *(.text) }  
    .bss : { *(.bss) }  
}
```

In [Chapter 6](#), we learned that the kernel is always loaded into physical address **0x10000000**. Well, as you can see in the code above, this is where we declare that address and tell the system where our executable code can be found once it is loaded into memory. The period (.) on the left side of the equals sign represents the location counter which is always initialized to **0x0**.

The asterisk (*) before each section name within the curly braces indicates a wildcard. This tells the compiler to locate any executable code (**.text**) from all input files (*), which we will define shortly, and merge it all into the **.text** section that we have defined here. The same is true for the **.bss** section as well. But, instead of executable code (**.text**), it tells the compiler to look for all global variables (**.bss**) from all input files (*) and to compile those into the **.bss** section we have defined here.

That's it. That is everything you need for your **linker.ld** file. Below is that file in its entirety to prove it.

linker.ld

```
ENTRY (_start)
SECTIONS
{
    . = 0x100000;
    .text : { *(.text) }
    .bss : { *(.bss) }
}
```

Now that we have our linker file, the only thing we have left to do is to compile everything. But, instead of manually typing out every compile command into a terminal, we are going to ease our pains by building a reusable **Makefile**. Since you are already familiar with the commands for compiling the entry point and kernel, and because you can easily learn how **Makefiles** work with a quick Internet search, I am going to skip most of the details here and just show you the code.

In your CentOS development environment, open **gedit** once more and add the following code to a new file. Then, save that file as “**Makefile**” (with no file extension) into the same location as your **start.asm**, **kernel.c**, and **linker.ld** files. This file will be read by the “**make**” tool.

As per Wikipedia, “**make** is a utility that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program.”

Makefile

```
CC=gcc
TARGET=myos
C_FILES=./kernel.c
OBJS=$(C_FILES:.c=.o)
```

```
all compile: $(TARGET)
all: finale
.PHONY: all compile clean finale
```

```
%o:
$(CC) -c $(@:.o=.c) -o $@ -ffreestanding -fno-exceptions -m32
```

```
$(TARGET): $(OBJS)
$(shell nasm -f elf start.asm -o start.o)
$(CC) -m32 -nostdlib -nodefaultlibs -lgcc start.o $? -T linker.ld -o $(TARGET)
```

finale:

```
$(shell cd ~/Desktop/my_os/)
$(shell cp $(TARGET) ./iso/boot/$(TARGET))
$(shell grub2-mkrescue iso --output=$(TARGET).iso)
```

clean:

```
rm -f *.o $(TARGET) $(TARGET).iso
find . -name \*.o | xargs --no-run-if-empty rm
```

Before we can check that everything compiles correctly, there is one more thing we need to do to wrap up development. As explained in [Chapter 6](#), we will be using a utility called “**GNU GRUB**” (short for “**GNU GRand Unified Bootloader**”) that will allow us to compile our operating system for various systems using the specific kernel we built in the [last chapter](#). But, before we can use the **GRUB** utility, we first need to create a configuration file that tells **GRUB** what to do.

Back in your CentOS development environment, open a command terminal and run the following command.

```
# mkdir -p ~/Desktop/my_os/iso/boot/grub/
```

This will create a subfolder called “**iso**” in the same folder as your source code. Inside that folder, another folder called “**boot**” will be created which will include a subfolder within it called “**grub**”. This folder structure is required by the **grub2-mkrescue** utility that we will learn about shortly.

Next, use **gedit** to create a new file and save it into the **~/Desktop/my_os/iso/boot/grub/** folder named “**grub.cfg**”. This is the configuration file that will tell **grub** how it should create our **iso** which we will use in the [next chapter](#). Since it is pretty much self-explanatory, here is the code that you will need to add into your **grub.cfg** file.

grub.cfg

```
set timeout=0
set default=0
menuentry "My Cool OS" {
    multiboot /boot/myos
}
```

As mentioned before, you should already be familiar with most of the commands listed in the **Makefile**. The one section I do want to point out, however, is the “**finale**”. As you can see in the **Makefile** code above, this section makes a call out to the **shell** that tells it to change directory into the location where we created our operating system files (i.e. “**~/Desktop/my_os/**” - assuming you followed the naming conventions in this book). Then, it makes another call out to the **shell** telling it to copy the compiled executable kernel into the “**/iso/boot/**” folder we just created. After that, it tells the **grub2-mkrescue** utility to generate an **iso** file that we will use in the [next chapter](#) for creating our VirtualBox VM and testing our operating system.

To verify that our **Makefile** works and that our operating system can compile successfully, click on **Applications > Utilities > Terminal** and run the following command:

```
# make
```

If everything compiled successfully, you should now see some extra files in your OS folder: **start.o**, **kernel.o**, and **myos**. The main file to check for here is the “**myos**” file.

This is the binary file for your executable kernel. If you want to also have your **Makefile** cleanup your OS folder a bit by removing the compiled object files (which is a good idea to do before re-running the **make** utility to prevent compilation using stale files), you can add the word “**clean**” to the end of the “**all: finale**” line inside your **Makefile**. Then, whenever you run the **make** utility from here on, it will call the “**clean**” section located at the bottom of the **Makefile** which will remove all compiled object files from your OS folder (i.e., all files that have the “**.o**” file extension) as well as your compiled kernel binary.



Congratulations! You now have a brand new operating system that you built yourself! And now is the time where you get to enjoy the fruits of your labor. In the [next chapter](#), we will finally get to test our new shiny operating system (and show it off to all of our soon-to-be jealous friends)! So, let’s get right to it.

0x09 Testing Your Operating System

So you've learned how the underlying mechanics of a computer work, got an introduction to the Assembly and C programming languages, and have gone through the [little] work of writing on your own operating system. Now comes the fun part! It is now time to test your shiny new operating system and admire it in all its glory. Thankfully, we already have some experience in doing this. Just like we did in [Chapter 5](#) when we setup our CentOS development environment, we will go through the same process for creating a new virtual machine using VirtualBox.

Before we can create a new VM (Virtual Machine) for testing our new operating system, we will need to copy the ISO that we created in the [last chapter](#) over to our Windows host computer. Luckily for us, we have already taken the necessary steps to make this an easy task (Remember in [Chapter 5](#) when we installed the VirtualBox Guest Additions?).

To copy your compiled ISO from your VirtualBox guest machine over to your Windows host machine, begin by launching Windows Explorer on your Windows host machine and navigate to a location where you would like to copy your ISO to. For now, you can choose your Desktop if you'd like. Just make sure you select a place that you will easily remember in the next step.

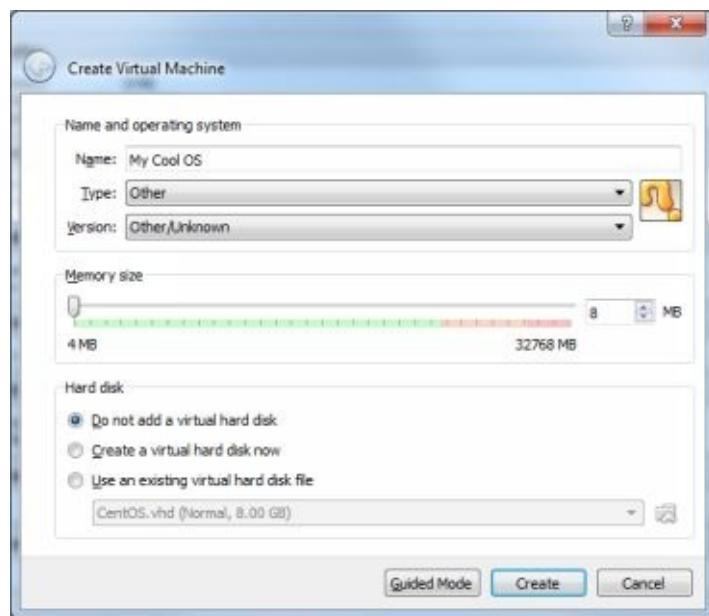
Over in your VirtualBox CentOS guest environment, click on **Places > Home** and navigate to **~/Desktop/my_os/** (or where ever you saved your source code and created your ISO). Next, right-click on the ISO ("myos.iso") in your CentOS environment and select **"Copy"**. Then, right-click on your Windows Explorer in your Windows environment and select **"Paste"**.

Note: If you have any issues copying and pasting from your CentOS guest machine to your Windows machine, you can resize your VirtualBox window so that you can position it on top of Window Explorer where you can drag the ISO from CentOS to Windows Explorer. Also, if you do not want to copy from CentOS to Windows, you can always install VirtualBox in your CentOS environment and create a VM within that VM.

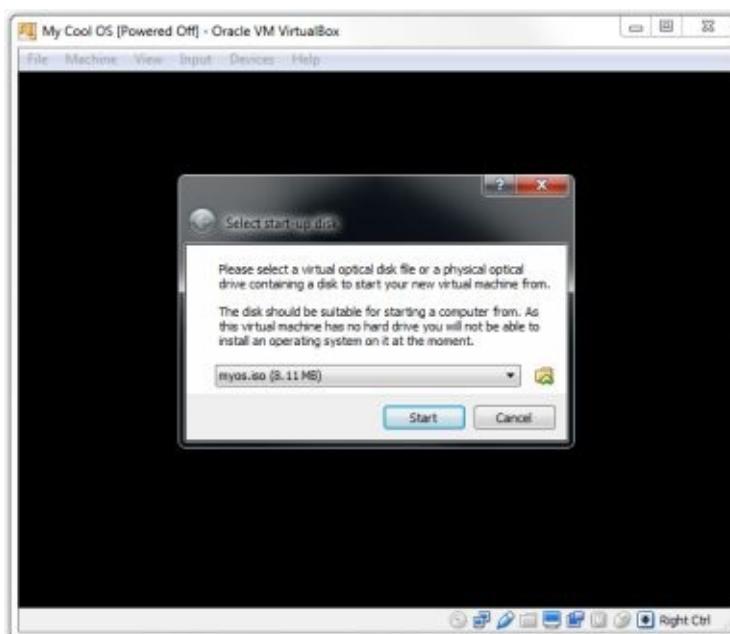
Now that you have your ISO copied to your Windows host machine, it is time to create the VM. To do that, begin by opening the Oracle VM VirtualBox Manager and clicking the blue **"New"** button on the toolbar or by going to **Machine > New**. This will open a modal window that will ask you about the virtual machine you are attempting to create.

For the **"Name"** field, type in something cool such as **"My Cool OS"** or **"My Bad Ass OS"**. Next, select **"Other"** for the **"Type"** field and **"Other/Unknown"** for the **"Version"** field. Since our operating system doesn't do much [yet], we can set the **"Memory size"** to

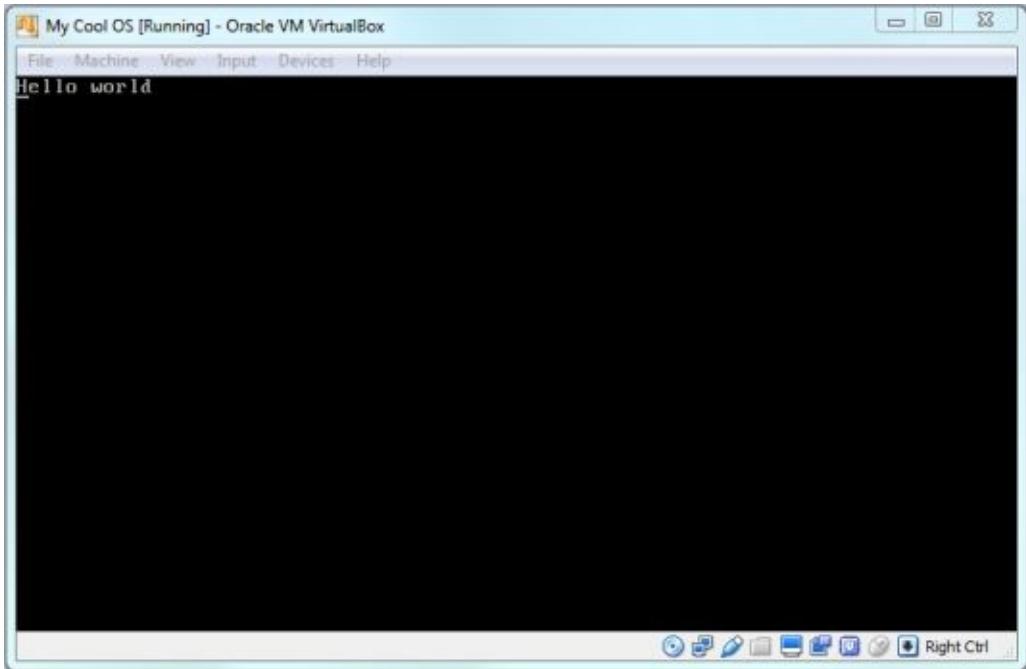
something like “**8 MB**” or less. Since we also do not have anything in our OS for accessing a file system, there is no need for selecting a hard disk. So, under “**Hard disk**”, select the first option for “**Do not add a virtual hard disk**”. After that, click the “**Create**” button.



Back in the Oracle VM VirtualBox Manager, you should now see your new operating system (“**My Cool OS**”) listed on the left below your “**CentOS**” VM. You can now launch that VM by either double-clicking the VM in the list or by single-clicking it and then clicking the green “**Start**” button on the toolbar. Just like when we were creating our CentOS VM, VirtualBox will now display a dialog window asking us to “**Select start-up disk**”. Click the folder-arrow icon at the right of the drop-down, navigate to your Desktop (or where ever you copied your ISO), and select your ISO.



The only thing you have left to do now is click the “**Start**” button. It will only take a second and voila! You should now see “**Hello world**” printed inside a black window. This is your operating system. Congratulations! Give yourself a pat on the back!



I know it doesn’t look like much right now, but this is the beginning of many great things that you can do from here on. And, to help you get started with making your new operating system do those great things, the next couple of chapters will walk you through starting your architecture library and expanding your OS to do even more.

0x0A Starting Your Architecture Library

So we have ourselves a brand new operating system. Now what? Well, now we will begin to grow that operating system into something bigger and better by providing it with more capabilities. We will do that by starting a library that can be used with different system architectures. Since we have already designed our operating system for the x86 architecture, we will continue with that by expanding our console capabilities.

Again, you have to remember that since we are building a completely new operating system from scratch, we do not have the C libraries that other operating systems provide. Therefore, we are solely responsible for building out all of the additional functionality for even the simplest of functions from here on.

Before we can start adding to what our console is capable of, we first need to add in several utility methods. Although not all of these utilities will be used in the architecture code we will look at in this chapter, it is still a nice idea to list them now as they will be used extensively in other parts of our operating system and other applications.

To get started, open the file explorer in your CentOS development environment and navigate to where you have created your operating system. Inside that folder, create two subfolders called “**kernel**” and “**libc**”. The **kernel** folder is where we will be adding our architecture specific code and the **libc** folder is where we will create the following utilities. For now, navigate into the **libc** folder and create two subfolders called “**include**” and “**string**”. Within the **include** folder, create a file called “**string.h**” with the following code:

string.h

```
#ifndef _STRING_H
#define _STRING_H 1

#include "stddef.h"

int memcmp(const void* aptr, const void* bptr, size_t size);size_t strlen(const char*);
char* strcat(char* d, const char* s);
char* strcpy(char* d, const char* s);
int strcmp(const char * s1, const char * s2);
char *strncat(char *dest, const char *src, size_t n);
char *strncpy(char *dest, const char *src, size_t n);
```

```
char *strstr(char *s1, const char *s2);
char *strchr(const char *s, int c);
int strncmp(const char * s1, const char * s2, size_t n);
```

```
#endif
```

This code simply lists out the available functions that we will define in the next utilities.

Next, inside the **string** folder, create individual files using the following file names and code.

ctos.c

```
#include "../include/string.h"

char *ctos(char s[2], const char c)
{
    s[0] = c;
    s[1] = '\0';
    return s;
}
```

This utility converts a single character into a NULL terminated string.

memcmp.c

```
#include "../include/string.h"

int memcmp(const void* aptr, const void* bptr, size_t size)
{
    const unsigned char* a = (const unsigned char*) aptr;
    const unsigned char* b = (const unsigned char*) bptr;
    size_t i;
    for ( i = 0; i < size; i++ )
```

```
    if ( a[i] < b[i] )
        return -1;
    else if ( b[i] < a[i] )
        return 1;
    return 0;
}
```

This utility accepts two memory areas and compares the first n bytes of the two.

memset.c

```
#include “./include/string.h”

void* memset(void* bufptr, int value, size_t size)
{
    unsigned char* buf = (unsigned char*) bufptr;
    size_t i;
    for ( i = 0; i < size; i++ )
        buf[i] = (unsigned char) value;
    return bufptr;
}
```

This utility fills the first $size$ bytes of memory pointed to by $bufptr$ with the constant byte $value$.

strlen.c

```
#include “./include/string.h”

size_t strlen(const char* str)
{
    size_t ret = 0;
    while ( str[ret] != 0 )
        ret++;
}
```

```
return ret;
```

```
}
```

This utility accepts a character array and returns the length (i.e. number of characters) of that string.

strcpy.c

```
#include “..../include/string.h”
```

```
char* strcpy(char* dest, const char* src) {
```

```
    char* tmp = dest;
```

```
    while ((*dest++ = *src++) != 0);
```

```
    return tmp;
```

```
}
```

This utility accepts two string arrays and copies the second string (“**src**”) into the first string (“**dest**”).

strcmp.c

```
#include “..../include/string.h”
```

```
int strcmp(const char* s1, const char* s2)
```

```
{
```

```
    while(*s1 && (*s1 == *s2))
```

```
        s1++, s2++;
```

```
    return *(const unsigned char*)s1 - *(const unsigned char*)s2;
```

```
}
```

This utility accepts two string arrays and compares them. If the return value is less than zero, it indicates that **s1** is less than **s2**. If the return value is greater than zero, it indicates **s2** is less than **s1**. If the return value is equal to zero, it indicates that the two string arrays are the same.

strncmp.c

```
#include “./include/string.h”
#include “stddef.h”

int strncmp(const char* s1, const char* s2, size_t n)
{
    while(n--)
        if (*s1++ != *s2++)
            return *(unsigned char*)(s1 - 1) - *(unsigned char*)(s2 - 1);
    return 0;
}
```

Similar to the **strcmp** utility, this utility compares two string arrays as well. However, this utility accepts a third parameter that tells the utility to compare the first **n** characters of each string.

strchr.c

```
#include “./include/string.h”

char *strchr(const char *s, int c)
{
    while (*s != (char)c)
        if (!*s++)
            return 0;
    return (char *)s;
}
```

This utility accepts a string array and a character (represented as an **int**) and searches the string array for the existence of the character. If the character is found within the string array, a pointer to that character’s location will be returned. Otherwise, it will return NULL.

strstr.c

```
#include “..../include/string.h”

char *strstr(char *s1, const char *s2)
{
    size_t n = strlen(s2);
    while (*s1)
        if (!memcmp(s1++, s2, n))
            return s1 - 1;
    return 0;
}
```

Similar to the previous utility, this utility searches **s1** for the presence of **s2**. If the string array **s2** (i.e. the “needle”) is found in the string array **s1** (i.e. the “haystack”), the utility will return a pointer to the location of **s2**. Otherwise, it will return zero.

strcat.c

```
#include “..../include/string.h”

char* strcat(char* dest, const char* src) {
    char* tmp = dest;

    while (*dest) dest++;
    while ((*dest++ = *src++) != 0);

    return tmp;
}
```

This utility takes the string array **src** and appends it to the end of the **dest** string array and returns that concatenated value.

strutil.c

```
#include “..../include/string.h”
```

```
int isupper(char c)
{
    return (c >= 'A' && c <= 'Z');
}
```

```
int islower(char c)
{
    return (c >= 'a' && c <= 'z');
}
```

```
int isalpha(char c)
{
    return ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'));
}
```

```
int isspace(char c)
{
    return (c == ' ' || c == '\t' || c == '\n' || c == '\12');
}
```

```
int isdigit(char c)
{
    return (c >= '0' && c <= '9');
}
```

```
char *ltrim(char *s)
{
    while (isspace(*s)) s++;
    return s;
}
```

```
char *rtrim(char *s)
```

```
{  
    char* back = s + strlen(s);  
    while (isspace(*—back));  
    *(back + 1) = '\0';  
    return s;  
}
```

```
char *trim(char *s)  
{  
    return rtrim(ltrim(s));  
}
```

These utilities are fairly self-explanatory, especially to those that are familiar with programming. The first two utilities check if the incoming characters are uppercase or lowercase respectively. The third utility checks if the incoming character is a letter (i.e. A-Z). The fourth utility checks if the incoming character is a space. The fifth utility checks if the incoming character is a number (i.e. 0-9). The sixth utility will trim whitespaces from the left side of a string. The seventh utility will trim whitespaces from the right side of a string. And, the eighth utility will trim whitespaces from both the left and right sides of a string.

Expanding the Console

As mentioned before, most of the work we do in our kernel will be specific to the system architecture we are planning to run our operating system on. Because of that, we will need to create files that are also architecture specific. Since earlier I mentioned that we will be expanding our operating system by expanding the capabilities of our console, that is what the next couple of files will focus on.

Earlier inside our “**my_os**” folder, we created two subfolders called “**kernel**” and “**libc**”. Inside the “**kernel**” folder, create a new file called “**tty.h**”. This file is where we will list the different VGA colors as mentioned in [Chapter 7](#). This is also where we will define the methods for “**printf**” that we will use to print strings to the console, “**terminal_initialize**” which we will use for setting up our console, and procedures for getting the current X and Y values of the cursor.

tty.h

```
#ifndef _TTY_H
#define _TTY_H 1

enum vga_color {
    COLOR_BLACK = 0,
    COLOR_BLUE = 1,
    COLOR_GREEN = 2,
    COLOR_CYAN = 3,
    COLOR_RED = 4,
    COLOR_MAGENTA = 5,
    COLOR_BROWN = 6,
    COLOR_LIGHT_GREY = 7,
    COLOR_DARK_GREY = 8,
    COLOR_LIGHT_BLUE = 9,
    COLOR_LIGHT_GREEN = 10,
    COLOR_LIGHT_CYAN = 11,
    COLOR_LIGHT_RED = 12,
    COLOR_LIGHT_MAGENTA = 13,
    COLOR_LIGHT_BROWN = 14,
    COLOR_WHITE = 15,
```

```
};
```

```
void terminal_initialize(void);
int printf(const char* format, ...);

int get_terminal_row(void);
int get_terminal_col(void);

#endif
```

Now that we have defined the procedures we will be calling from inside our kernel, it is now time for us to add the code that actually does the work. For that, create a new file called “**tty.c**” and save it within the same “**kernel**” folder where you created the “**tty.h**” file. Since you should already have a good understanding of how C code works, I won’t bore you with the details of what is going on in this file. Instead, I will just show you the code as it is relatively straight forward and extremely simple to follow.

tty.c

```
#include “stdint.h”
#include “stddef.h”
#include “stdarg.h”
#include “tty.h”
#include “../libc/include/string.h”

static const size_t VGA_WIDTH = 80;
static const size_t VGA_HEIGHT = 25;

static uint16_t* const VGA_MEMORY = (uint16_t*) 0xb8000;

size_t terminal_row;
size_t terminal_column;
uint8_t terminal_color;
uint16_t* terminal_buffer;
```

```

static inline uint8_t make_color(enum vga_color fg, enum vga_color bg) {
    return fg | bg << 4;
}

static inline uint16_t make_vgaentry(char c, uint8_t color) {
    uint16_t c16 = c;
    uint16_t color16 = color;
    return c16 | color16 << 8;
}

void terminal_initialize(void) {
    terminal_row = 0;
    terminal_column = 0;
    terminal_color = make_color(COLOR_LIGHT_GREY, COLOR_BLACK);
    terminal_buffer = VGA_MEMORY;
    size_t y;
    for (y = 0; y < VGA_HEIGHT; y++) {
        size_t x;
        for (x = 0; x < VGA_WIDTH; x++) {
            const size_t index = y * VGA_WIDTH + x;
            terminal_buffer[index] = make_vgaentry(' ', terminal_color);
        }
    }
}

void terminal_scroll()
{
    int i;
    for (i = 0; i < VGA_HEIGHT; i++){
        int m;
        for (m = 0; m < VGA_WIDTH; m++){
            terminal_buffer[i * VGA_WIDTH + m] = terminal_buffer[(i + 1) * VGA_WIDTH + m];
        }
    }
}

```

```

}

terminal_row--;
}

terminal_row = VGA_HEIGHT - 1;

}

void terminal_putchar(char c, uint8_t color, size_t x, size_t y) {
    const size_t index = y * VGA_WIDTH + x;
    terminal_buffer[index] = make_vgaentry(c, color);
}

void terminal_putchar(char c) {
    if (c == '\n' || c == '\r') {
        terminal_column = 0;
        terminal_row++;
        if (terminal_row == VGA_HEIGHT)
            terminal_scroll();
        return;
    } else if (c == '\t') {
        terminal_column += 4;
        return;
    } else if (c == '\b') {
        terminal_putentryat(' ', terminal_color, terminal_column--, terminal_row);
        terminal_putentryat(' ', terminal_color, terminal_column, terminal_row);
        return;
    }
}

terminal_putentryat(c, terminal_color, terminal_column, terminal_row);
if ( ++terminal_column == VGA_WIDTH ) {
    terminal_column = 0;
    if ( ++terminal_row == VGA_HEIGHT ) {
        terminal_row = 0;
    }
}

```

```
        }
```

```
    }
```

```
}
```

```
void terminal_write(const char* data, size_t size) {
```

```
    size_t i;
```

```
    for ( i = 0; i < size; i++ )
```

```
        terminal_putchar(data[i]);
```

```
}
```

```
int putchar(int ic) {
```

```
    char c = (char)ic;
```

```
    terminal_write(&c, sizeof(c));
```

```
    return ic;
```

```
}
```

```
static void print(const char* data, size_t data_length) {
```

```
    size_t i;
```

```
    for ( i = 0; i < data_length; i++ )
```

```
        putchar((int) ((const unsigned char*) data)[i]);
```

```
}
```

```
int printf(const char* format, ...)
```

```
    va_list parameters;
```

```
    va_start(parameters, format);
```

```
    int written = 0;
```

```
    size_t amount;
```

```
    int rejected_bad_specifier = 0;
```

```
    while ( *format != '\0' ) {
```

```
        if ( *format != '%' ) {
```

```

print_c:
    amount = 1;
    while ( format[amount] && format[amount] != '%' )
        amount++;
    print(format, amount);
    format += amount;
    written += amount;
    continue;
}

const char* format_begun_at = format;

if ( *(++format) == '%' )
    goto print_c;

if ( rejected_bad_specifier ) {
    incomprehensible_conversion:
    rejected_bad_specifier = 1;
    format = format_begun_at;
    goto print_c;
}

if ( *format == 'c' ) {
    format++;
    char c = (char) va_arg(parameters, int /* char promotes to int */);
    print(&c, sizeof(c));
} else if ( *format == 's' ) {
    format++;
    const char* s = va_arg(parameters, const char* );
    print(s, strlen(s));
} else {
    goto incomprehensible_conversion;
}

```

```
        }

    va_end(parameters);

    return written;
}

int get_terminal_row(void)
{
    return terminal_row;
}

int get_terminal_col(void)
{
    return terminal_column;
}
```

Now that we have those procedures out of the way, we need to reference the files we just created and make calls to the necessary procedures from inside our kernel. To do that, open up “**kernel.c**” and make the following changes.

kernel.c

```
#include “kernel/tty.h”

void kernel_early(void) {
    terminal_initialize();
}

__attribute__((noreturn))
int main(void) {
    printf(“Hello world\n”);
    while (1) { }
```

```
return 0;
```

```
}
```

As you can see in the above code, the first line is where we reference the file that includes the definitions for “**printf**” and “**terminal_initialize**”. Since those are the only two procedures we call from our kernel, those are the only two procedures we needed to define in our “**tty.h**” file. Next, inside the **kernel_early** function, we make a call to “**terminal_initialize**” which sets up our video memory and buffer. Then, inside our “**main**” function, we utilize the “**printf**” procedure we created before.

Before we can compile all of the code that we just added, we need to update our **Makefile** so that it too is aware of our new files. To do that, simply add the file names for each of the **.c** files to the **C_FILES** variable in your **Makefile**. When finished, it should look like the following where the items in bold are the changes made since we built this file in [Chapter 8](#).

Makefile

```
CC=gcc
```

```
TARGET=myos
```

```
C_FILES= ./libc/string/ctos.c \
          ./libc/string/memcmp.c \
          ./libc/string/memset.c \
          ./libc/string/strcat.c \
          ./libc/string/strchr.c \
          ./libc/string/strcmp.c \
          ./libc/string/strcpy.c \
          ./libc/string/strlen.c \
          ./libc/string/strncmp.c \
          ./libc/string/strstr.c \
          ./libc/string/strutil.c \
          ./kernel/tty.c \
          ./kernel.c
```

```
OBJS=$(C_FILES:.c=.o)
```

all compile: \$(TARGET)

```
all: finale
```

```
.PHONY: all compile clean finale
```

```
% .o:
```

```
$(CC) -c $(@:.o=.c) -o $@ -ffreestanding -fno-exceptions -m32
```

```
$(TARGET): $(OBJS)
```

```
$(shell nasm -f elf start.asm -o start.o)
```

```
$(CC) -m32 -nostdlib -nodefaultlibs -lgcc start.o $? -T linker.ld -o $(TARGET)
```

```
finale:
```

```
$(shell cd ~/Desktop/my_os/)
```

```
$(shell cp $(TARGET) ./iso/boot/$(TARGET))
```

```
$(shell grub2-mkrescue iso --output=$(TARGET).iso)
```

```
clean:
```

```
rm -f *.o $(TARGET) $(TARGET).iso
```

```
find . -name \*.o | xargs --no-run-if-empty rm
```

The only thing you have left to do now is compile everything by executing the “**make**” command again from your terminal. Once you have done that, follow the steps from [Chapter 9](#) again to deploy and test your operating system to make sure you didn’t miss anything.

0x0B Expanding Your OS

Now that we have started building out a working architecture-specific operating system, let's build on what we have done so far to expand our operating system even further. Even though I will be throwing a lot of code at you in this chapter, do not get discouraged. All of the code mentioned in this chapter can be downloaded in its entirety from my website at the link below. In fact, the source code for the entire operating system mentioned in this book is available in the ZIP file at the following address:

http://www.prodigyproductionsllc.com/downloads/my_cool_os.zip

Note: This chapter contains more source code and less text. So, if you are happy with your current operating system, want to simply download the source code from my website, or want to move on to learning how to [cross-compile for other architectures](#), please feel free to skip this chapter and return to it at a later time.

The first thing we will take a look at adding to our existing operating system is the ability to accept user input via the keyboard. In order to do that, we will make use of inline Assembly code which we haven't done yet, but is relatively straight forward. Doing this from C code is just as easy as doing it directly in Assembly. All you have to do is take the code you would normally write in Assembly and wrap it in parenthesis prefixed with “**asm**” or “**__asm__**” as in the case below.

You will also notice that we have also added the keyword “**__volatile__**” to the beginning of our Assembly code as well. The reason for doing this is because calling Assembly code from inside of C can sometimes produce unintentional side effects. This is in part due to the **gcc** (and similar) compilers that attempt to optimize the code during compilation. By prefixing Assembly code with “**__volatile__**” qualifier, it will tell the compiler to disable certain optimizations and basically execute the code “as-is”.

As always, we will begin adding new functionality to our operating system by first defining the procedures that we are adding. For keyboard input, create a new file in your “**kernel**” folder (since this is architecture specific) called “**io.h**” and add the following code.

io.h

```
#ifndef _IO_H
#define _IO_H 1

#include "stdint.h"
```

```

/* The I/O ports */

#define FB_COMMAND_PORT      0x3d4
#define FB_DATA_PORT         0x3d5

/* The I/O port commands */

#define FB_HIGH_BYTE_COMMAND 0x0e
#define FB_LOW_BYTE_COMMAND  0x0f

uint8_t inb(uint16_t port);
void outb(uint16_t port, uint8_t val);
uint8_t scan(void);

void move_cursor(int row, int col);
void printprompt(void);

#endif

```

Next, create another file in the same location and name it “**io.c**” that contains the following code. Basically, all this code does is it takes a 8/16/32-bit value and either receives it in (“**inb**”) from an I/O location or sends it out (“**outb**”) to an I/O location. It also monitors the I/O port **0x60** (which is the keyboard port) for any input and returns incoming value values via the “**scan**” procedure. After that, there are procedures for moving the cursor on the screen and drawing a command prompt similar to the one found in Linux terminals (i.e., “\$>”).

io.c

```

#include "io.h"
#include "tty.h"

uint8_t inb(uint16_t port)
{
    uint8_t ret;
    __asm__ __volatile__("inb %1, %0" : "=a" (ret) : "Nd" (port));

```

```
return ret;
}

void outb(uint16_t port, uint8_t val)
{
    __asm__ __volatile__("outb %0, %1" :: "a" (val), "Nd" (port));
}

uint8_t scan(void)
{
    unsigned char brk;
    static uint8_t key = 0;
    uint8_t scan = inb(0x60);
    brk = scan & 0x80;
    scan = scan & 0x7f;
    if (brk)
        return key = 0;
    else if (scan != key)
        return key = scan;
    else
        return 0;
}

void move_cursor(int row, int col)
{
    unsigned short pos = (row * 80) + col;
    outb(FB_COMMAND_PORT, FB_LOW_BYTE_COMMAND);
    outb(FB_DATA_PORT, (unsigned char)(pos & 0xFF));
    outb(FB_COMMAND_PORT, FB_HIGH_BYTE_COMMAND);
    outb(FB_DATA_PORT, (unsigned char)((pos >> 8) & 0xFF));
}
```

```
void printprompt(void)
{
    printf("\n$> ");
    move_cursor(get_terminal_row(), get_terminal_col());
}
```

As we read data from I/O ports directly, the data that we get needs to be translated into something meaningful, at least from the user's standpoint. In the case of reading keyboard data, we will need to map the input to keys that are actually on the keyboard. To do that, we will create another file in our “**kernel**” folder called “**kbd.h**” which will hold the keyboard map for us. This file will also include a few other definitions (aka. “variables”) that will make keyboard mapping easier.

kbd.h

```
#define KBSTATP      0x64 // kbd controller status port(I)
#define KBS_DIB       0x01 // kbd data in buffer
#define KBDATAP       0x60 // kbd data port(I)

#define NO           0

#define SHIFT        (1<<0)
#define CTL          (1<<1)
#define ALT          (1<<2)

#define CAPSLOCK     (1<<3)
#define NUMLOCK       (1<<4)
#define SCROLLLOCK    (1<<5)

#define E0ESC        (1<<6)

// Special keycodes
#define KEY_HOME     0xE0
#define KEY_END       0xE1
#define KEY_UP        0xE2
```

```
#define KEY_DN      0xE3
#define KEY_LF      0xE4
#define KEY_RT      0xE5
#define KEY_PGUP    0xE6
#define KEY_PGDN    0xE7
#define KEY_INS     0xE8
#define KEY_DEL     0xE9
```

// C('A') == Control-A

```
#define C(x) (x - '@')
```

```
static char shiftcode[256] =
```

```
{  
[0x1D] CTL,  
[0x2A] SHIFT,  
[0x36] SHIFT,  
[0x38] ALT,  
[0x9D] CTL,  
[0xB8] ALT
```

```
};
```

```
static char togglecode[256] =
```

```
{  
[0x3A] CAPSLOCK,  
[0x45] NUMLOCK,  
[0x46] SCROLLLOCK
```

```
};
```

```
static char normalmap[256] =
```

```
{  
NO, 0x1B, '1', '2', '3', '4', '5', '6', // 0x00  
'7', '8', '9', '0', '-', '=', '\b', '\t',
```

```

'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', // 0x10
'o', 'p', '[', ']', '\n', NO, 'a', 's',
'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', // 0x20
'', '^', NO, 'v', 'z', 'x', 'c', 'v',
'b', 'n', 'm', ',', '.', '/', NO, '*', // 0x30
NO, ' ', NO, NO, NO, NO, NO, NO, NO,
NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
'8', '9', '-', '4', '5', '6', '+', '1',
'2', '3', '0', '.', NO, NO, NO, NO, // 0x50
[0x9C] '\n', // KP_Enter
[0xB5] '/', // KP_Div
[0xC8] KEY_UP, [0xD0] KEY_DN,
[0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
[0xCB] KEY_LF, [0xCD] KEY_RT,
[0x97] KEY_HOME, [0xCF] KEY_END,
[0xD2] KEY_INS, [0xD3] KEY_DEL
};


```

```

static char shiftmap[256] =
{
NO, 033, '!', '@', '#', '$', '%', '^', // 0x00
'&', '*', '(', ')', '_', '+', '\b', '\t',
'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', // 0x10
'O', 'P', '{', '}', '\n', NO, 'A', 'S',
'D', 'F', 'G', 'H', 'J', 'K', 'L', ':', // 0x20
'', '~', NO, '|', 'Z', 'X', 'C', 'V',
'B', 'N', 'M', '<', '>', '?', NO, '*', // 0x30
NO, ' ', NO, NO, NO, NO, NO, NO, NO,
NO, NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
'8', '9', '-', '4', '5', '6', '+', '1',
'2', '3', '0', '.', NO, NO, NO, NO, // 0x50
[0x9C] '\n', // KP_Enter

```

```

[0xB5] '/', // KP_Div
[0xC8] KEY_UP, [0xD0] KEY_DN,
[0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
[0xCB] KEY_LF, [0xCD] KEY_RT,
[0x97] KEY_HOME, [0xCF] KEY_END,
[0xD2] KEY_INS, [0xD3] KEY_DEL
};

static char ctlmap[256] =
{
    NO, NO, NO, NO, NO, NO, NO, NO,
    NO, NO, NO, NO, NO, NO, NO, NO,
    C('Q'), C('W'), C('E'), C('R'), C('T'), C('Y'), C('U'), C('I'),
    C('O'), C('P'), NO, NO, '\r', NO, C('A'), C('S'),
    C('D'), C('F'), C('G'), C('H'), C('J'), C('K'), C('L'), NO,
    NO, NO, NO, C('\'), C('Z'), C('X'), C('C'), C('V'),
    C('B'), C('N'), C('M'), NO, NO, C('/'), NO, NO,
    [0x9C] '\r', // KP_Enter
    [0xB5] C('/'), // KP_Div
    [0xC8] KEY_UP, [0xD0] KEY_DN,
    [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
    [0xCB] KEY_LF, [0xCD] KEY_RT,
    [0x97] KEY_HOME, [0xCF] KEY_END,
    [0xD2] KEY_INS, [0xD3] KEY_DEL
};

```

Before we can use the code from this chapter, we first need to compile it. That means we will need to update our **Makefile** to include the files we just finished creating. To do that, open up your **Makefile** and add the following files under your **C_FILES** variable.

```

./kernel/tty.c \
./kernel/io.c \

```

Your overall **C_FILES** variable should now look like this:

C_FILES

```
C_FILES=./libc/string/memcmp.c \
./libc/string/memset.c \
./libc/string/strcat.c \
./libc/string/strchr.c \
./libc/string/strcmp.c \
./libc/string/strcpy.c \
./libc/string/strlen.c \
./libc/string/strncmp.c \
./libc/string/strstr.c \
./libc/string/strutil.c \
./libc/string/ctos.c \
./kernel/tty.c \
./kernel/io.c \
./kernel.c
```

At this point we now have everything we need to accept user input via the keyboard and display that input in the console. So, all we have left to do now is update our **kernel.c** file to account for this user input and send it to the screen. While we are at it, we will also add some code to check incoming text for the presence of specific keyboards and will treat those accordingly. As you can see in the code below, I am checking for the presence of the single word “**exit**” and am printing the text “**Goodbye!**” to the screen if the user types the word “**exit**” and presses the enter key. You will also see that I continuously move the cursor to the new location after every keystroke and print a new line with the prompt (“\$>”) that we defined in our **io.c** file.

kernel.c

```
#include "kernel/tty.h"
#include "kernel/io.h"
#include "kernel/kbd.h"
#include "libc/include/string.h"

void kernel_early(void) {
```

```

terminal_initialize();

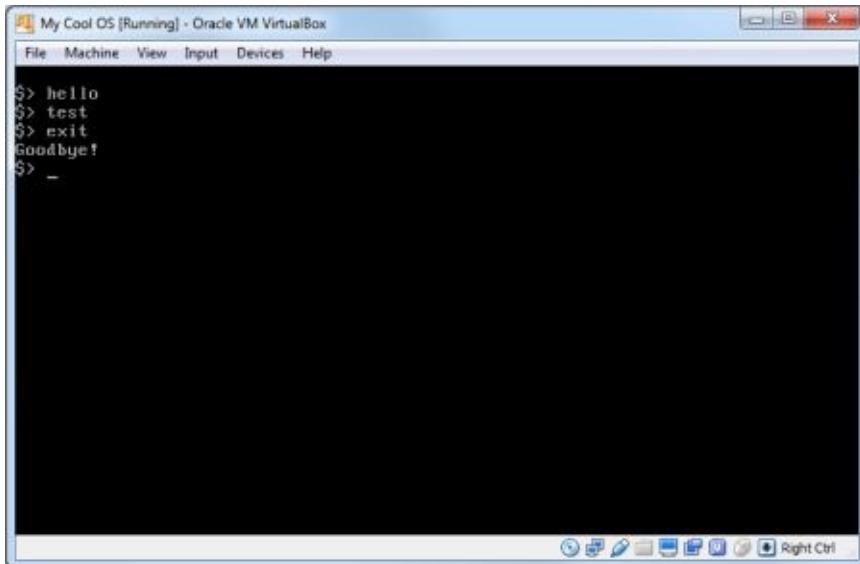
}

int main(void) {
    char *buff;
    strcpy(&buff[strlen(buff)], "");
    printprompt();
    while (1) {
        uint8_t byte;
        while (byte = scan()) {
            if (byte == 0x1c) {
                if (strlen(buff) > 0 && strcmp(buff, "exit") == 0)
                    printf("\nGoodbye!");
                printprompt();
                memset(&buff[0], 0, sizeof(buff));
                break;
            } else {
                char c = normalmap[byte];
                char *s;
                s = ctos(s, c);
                printf("%s", s);
                strcpy(&buff[strlen(buff)], s);
            }
        }
        move_cursor(get_terminal_row(), get_terminal_col());
    }
    return 0;
}

```

That's it. We should now be able to compile our operating system using the “**make**” command that we learned about earlier, copy the newly created ISO to our Windows host machine, and create a new VM (Virtual Machine) just as we did in [Chapter 9](#). If everything went accordingly, we should see something like the following whenever we

test our updated operating system.



I will now leave it with you to expand your operating system to do all of the magical things you want it to do. However, I won't leave it with you blindly. Instead, I will leave you with a quick note to point you in the right direction should you have the need to add networking capabilities to your operating system (which I am sure you will).

When it comes time to add networking to your operating system, whether it is for desktop type systems or embedded systems, there are two TCP/IP stacks that I suggest you look into.

The first is called “**lwIP**” (Light Weight IP) which you can find at <http://savannah.nongnu.org/projects/lwip/>. This is the networking library that I use the most when developing custom operating systems. It is one of the faster libraries (especially when compared to the next library I will mention) and includes DHCP as well as full UDP with multicast which are typically nice to have if not required, especially with embedded systems.

The second is called “**uIP**” which you can find at <https://github.com/adamdunkels/uip>. One of the best things about **uIP** is its small footprint. However, **uIP** uses polling which can constrain high throughput use cases and doesn't include DHCP out-of-the-box. If you need higher throughput than what **uIP** offers out-of-the-box, you can do some substantial tweaking to make it perform better.

If you have a little extra RAM and CPU, I would recommend going with **lwIP**. But, if

resource limitations are a big factor in your project, you should take a look at **uIP**. I have used both libraries and both are capable of getting the job done. It just depends on what your use case is and how much additional coding and tweaking you don't mind doing.

0x0C Cross-Compiling for Other Architectures

Everything we have learned in this book so far has been about developing an operating system that works on the x86 architecture. However, since not all devices utilize an x86 processor (like many of those found in the *Internet of Things*), it is advantageous for us to also know how to develop and compile operating systems for other architectures as well. So, that is what we will address in this chapter.

Not only will we learn how to cross-compile our operating system for other architectures, but we will also learn how to build our own cross-compilers that will allow us to compile for even more architectures as well - the architectures that **gcc** doesn't support natively. As an example, we will learn how to build a **gcc** cross-compiler for the **ARM-CORTEX** processor that the Raspberry Pi uses since it is widely gaining traction in the *Internet of Things*.

But, before we can cross-compile our operating system for the Raspberry Pi, we will begin by getting an existing cross-compiler that has been pre-built and ready for us to use. To do that, open a terminal prompt and run the following commands. These commands will create a directory to download our cross-compiler into, change directory into the newly created directory, download the cross-compiler, extract it, and rename the extracted folder for easier reference.

```
# sudo mkdir /opt
# cd /opt
# wget https://launchpad.net/gcc-arm-embedded/5.0/5-2015-q4-major/+download/gcc-arm-none-eabi-5\_2-2015q4-20151219-linux.tar.bz2
# sudo tar xjf gcc-arm-none-eabi-5_2-2015q4-20151219-linux.tar.bz2
# sudo mv gcc-arm-none-eabi-f_5_2-2015q4 arm-none-eabi
```

As you can see in the above commands, the download tarball includes a version number in the filename. To ensure you have the latest build, point your web browser to <https://launchpad.net/gcc-arm-embedded/+download> and check for the latest version number and latest download link. Just make sure you select the download for “Linux installation tarball”.

Now that you have downloaded and extracted the ARM cross-compiler, all you have left to do is add it to your PATH environment variable so that you can easily reference it. To add the cross-compiler to your PATH variable, run the following command from a terminal prompt in CentOS. This command will ensure that the path to your ARM cross-compiler is always available, even after rebooting CentOS.

```
# echo export PATH=${PATH}:~/opt/arm-none-eabi/bin
```

From now on, whenever you want to compile your operating system for the Raspberry Pi, all you have to do is prefix your gcc, as, ld, and objcopy utilities with “arm-none-eabi” in your Makefile like so.

```
ARCH=arm-none-eabi
CC=${ARCH}-gcc
AS=${ARCH}-as
LD=${ARCH}-ld
OBJCPY=${ARCH}-objcopy
```

Even though we now have a way to cross-compile our operating system to run on a Raspberry Pi, do not get too excited just yet. As we learned in earlier chapters, the code we developed for our bootloader, linker, and parts of our kernel are specific to the x86 architecture. In order to have our operating system run on the Raspberry Pi (or other architectures for that matter), we will need to modify our bootloader, kernel, and linker file to match the architecture we wish to run on.

Toward the end of this chapter, I have provided some sample code to help get you started with porting your operating system to run on the Raspberry Pi. But, before we get into that, I want to first show you how to build your own cross-compilers so that you will be armed with enough knowledge to build operating systems that can run on any architecture. And, since we are already on the path of cross-compiling for the Raspberry Pi, we will stick with this approach and learn how to build a custom cross-compiler from scratch that will be similar to the pre-built cross-compiler we previously downloaded. To do that, we will need to download and install yet another tool.

Create a Custom Cross-Compiler

The tool we will be using to create our own cross-compilers is called “crosstool-ng” (<http://crosstool-ng.org>). Begin by opening a terminal in CentOS and downloading the tool using the following commands.

```
# cd /tmp  
# wget http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-1.22.0.tar.bz2
```

Note: At the time of writing this book, the release version of crosstool-ng is 1.22.0. It is recommended that you check the <http://crosstool-ng.org> website for the latest version and substitute the latest version number into the wget URL above. You will also need to use the same version in the commands below when unpacking the tool.

Next, you will need to run the following command to install a few dependencies that are required by the crosstool-ng installer.

```
# sudo yum -y install gperf bison flex texinfo libtool automake help2man patch ncurses*
```

Next, run the following commands to unpack and install the crosstool-ng tool (“**ct**” hence forth).

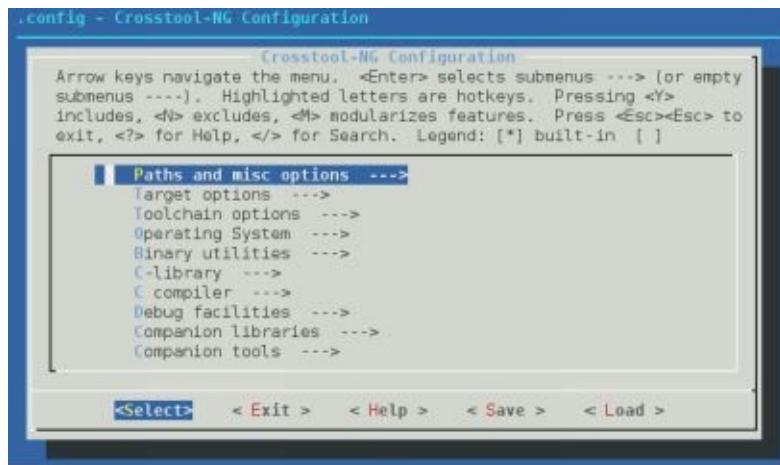
```
# tar xjf crosstool-ng-1.22.0  
# cd crosstool-ng  
# ./configure --prefix=/opt/cross  
# make  
# sudo make install  
# sudo cp ct-ng.comp /etc/bash_completion.d/  
# export PATH="${PATH}:/opt/cross/bin"
```

You can replace the path in the configure prefix above with a location that better suits you and your system, but make sure you also replace the path in the **PATH** environment variable as well.

Now that you have **ct** installed, it is time to use it. To begin with, we will need to create a new directory that will store the configuration files for **ct**. Once we have created that

directory, we will change directory into it so that **ct** knows where to store its configuration files. From there, we can launch the tool.

```
# sudo mkdir /opt/cross/toolchains
# sudo mkdir /opt/cross/config
# cd /opt/cross/config
# sudo /opt/cross/bin/ct-ng menuconfig
```



Using your arrow keys, you will need to traverse the configuration tool to make the following changes. Note: To select (enable/disable) items, press the spacebar. To edit text fields (such as the “Prefix directory”, highlight the item and press enter.

- Paths and misc options
 - Enable “Try features marked as EXPERIMENTAL”
 - Enable “Allow building as root user”
 - Enable “Are you sure?”
 - Change the “\${HOME}/x-tools/\${CT_TARGET}” Prefix directory to “/opt/cross/toolchains/\${CT_TARGET}”
- Target options
 - Change the “Target Architecture” to “arm”
 - Leave “Endianness” as “Little endian”
 - Leave “Bitness” as “32-bit”
 - Set “Architecture level” as “armv6zk”
 - Set “Emit assembly for CPU” as “arm1176jf-s”
 - Set “Tune for CPU” as “arm1176jf-s”
 - Set “Use specific FPU” as “vfp”
 - Change “Floating point” to “hardware (FPU)”
 - Leave “Default instruction set mode” as “arm”
 - Make sure “Use EABI” is selected
- Toolchain options

- Change “Tuple’s vendor string” to “rpi”
- Operating system
 - Change “Target OS” to “linux”
- Binary utilities
 - Leave “Binary format” as “ELF”
 - Change “bintools version” to “2.22”
- C compiler
 - Enable “Show Linaro versions”
 - Change the “gcc version” to “linaro-4.9-2015.06” (or whatever the latest linaro version is)
 - Set “gcc extra config” as “—with-float=hard”
 - Enable “C++”

After you have made your changes, be sure to save your changes before exiting the utility. The next thing we will do is tell **ct** to build our new cross-compiler. A word of warning, this can take quite a while to run. So, I would advise that you do this late at night, just before going to bed, or at another time when you have other things you can be doing while it runs.

If we try to build our cross-compiler without setting the flag “Allow building as root user” in the “Paths and misc options” menu above, it will complain about not having permissions to remove certain files. Even if we attempt to run the build process as sudo, it will complain with the error “You must NOT be root to run crosstool-NG”. To get around this, we will need to either enable “Allow building as root user” in the config tool or change ownership of the folder and files for the configuration we did in the previous step. To do the latter, execute the following command and replace “USER:USER” with the name and group of the user you are currently logged in as.

```
# sudo chown -R USER:USER /opt/cross/config
# sudo chown -R USER:USER /opt/cross/toolchains
```

Next, begin the cross-compiler build while you find something else to do.

```
# cd /opt/cross/config/
# sudo /opt/cross/bin/ct-ng build
```

As mentioned before, this process will take a while to run. As you can see in the screenshot below, it took almost an hour (57 minutes, 58 seconds) to run on my laptop. I am also running a SSD (Solid State Drive) and 32GB of RAM with a 2.7GHz quad-core i7 processor. So, if you are running anything less, it will most likely take longer to complete

on your system.

```
[INFO] =====
[INFO] Extracting and patching toolchain components
[INFO] Extracting and patching toolchain components: done in 204.90s (at 05:45)
[INFO] =====
[INFO] Installing GMP for host
[INFO] Installing GMP for host: done in 116.88s (at 07:42)
[INFO] =====
[INFO] Installing MPFR for host
[INFO] Installing MPFR for host: done in 51.52s (at 08:34)
[INFO] =====
[INFO] Installing ISL for host
[INFO] Installing ISL for host: done in 44.94s (at 09:19)
[INFO] =====
[INFO] Installing CLoog for host
[INFO] Installing CLoog for host: done in 7.89s (at 09:26)
[INFO] =====
[INFO] Installing MPC for host
[INFO] Installing MPC for host: done in 20.63s (at 09:47)
[INFO] =====
[INFO] Installing binutils for host
[INFO] Installing binutils for host: done in 114.58s (at 11:42)
[INFO] =====
[INFO] Installing pass-1 core C gcc compiler
[INFO] Installing pass-1 core C gcc compiler: done in 545.34s (at 20:47)
[INFO] =====
[INFO] Installing kernel headers
[INFO] Installing kernel headers: done in 51.39s (at 21:38)
[INFO] =====
[INFO] Installing C library headers & start files
[INFO] Installing C library headers & start files: done in 15.97s (at 21:54)
[INFO] =====
[INFO] Installing pass-2 core C gcc compiler
[INFO] Installing pass-2 core C gcc compiler: done in 727.44s (at 34:02)
[INFO] =====
[INFO] Installing C library
[INFO] Installing C library: done in 555.73s (at 43:18)
[INFO] =====
[INFO] Installing final gcc compiler
[INFO] Installing final gcc compiler: done in 879.77s (at 57:57)
[INFO] =====
[INFO] Cleaning-up the toolchain's directory
[INFO]   Stripping all toolchain executables
[INFO] Cleaning-up the toolchain's directory: done in 2.20s (at 58:00)
[INFO] Build completed at 20160111.141923
[INFO] (elapsed: 57:58.65)
[INFO] Finishing installation (may take a few seconds)...
```

After the build process has completed, you will now have a compiler that will allow you to compile your operating system for the Raspberry Pi. However, there are a few changes we will need to make to our code before we can use the compiler.

To make it easy to get to our compiler, we will need to add its path to our **PATH** environment variable. So that this path is still available after we reboot, we will go ahead and make it permanent by using the following command.

```
# echo export PATH=${PATH}:~/opt/cross/toolchains/arm-rpi-linux-gnueabi/bin/ >> ~/.bashrc
```

Now, every time we want to call our cross-compiler, we can do so by typing just the compiler name itself which we can test with the following command.

```
# arm-rpi-linux-gnueabi-gcc --version
```

output

arm-rpi-linux-gnueabi-gcc (crosstool-NG crosstool-ng-1.22.0) 4.9.4 20150629
(prerelease)

Copyright (C) 2015 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.

To see what compilers were built by crosstool-ng, you can simply list all the files in the
toolchain directory using the following command.

```
# ls /opt/cross/toolchains/arm-rpi-linux-gnueabi/bin/
```

Porting for the Raspberry Pi

Even though we now have new compilers for compiling our operating system for different architectures, there is still work to be done. As we learned before, Assembly code is written per architecture – meaning, code that is written for one architecture will not run on another. The same is also true for parts of our kernel (mostly the pieces that communicate directly with the hardware such as our graphics card) and the linker file we use to pull it all together. Therefore, in order for us to run our operating system on the Raspberry Pi like we now want to do, we will have to make several changes to our code.

However, since explaining how operating systems work for the Raspberry Pi requires an entire book itself, I will not be explaining it all here nor will I be providing the code here either. Instead, you can download the code directly from my website at http://www.prodigyproductionsllc.com/downloads/pi_os.zip.

Testing on Physical Hardware

Now that we have our operating system compiled to run on the Raspberry Pi, it is time to actually run it on the physical hardware. But, before we can do that, there are a couple of things we need to do to prepare.

First, we need to format our SD card with the correct partition type that our Raspberry Pi can understand. To do that, insert your SD card into your Windows computer and open Windows Explorer. Then, right-click on the drive for your SD card and select “Format...”. In the dialog window that appears, select “FAT (Default)” for the “File system” and leave everything else as the defaults. When you are ready, click the “Start” button to format the card.

Next, open a web browser and navigate to

<http://www.prodigyproductionsllc.com/downloads/rpiloader.zip>. In this ZIP, you will find two files called “bootcode.bin” and “start.elf”. These are the bootloader files required by the Raspberry Pi in order to boot your operating system.

You will need to extract these files from the ZIP and copy them into the root directory of the SD card you just formatted. Once you have done that, take the “kernel.img” file you built in the previous section over to your SD card. When done, your SD card’s file structure should look like the following.



If your files look like the above, then your SD card is now ready to test. To do that, simply remove the SD card from your Windows computer and insert it into your Raspberry Pi. Next, connect the Raspberry Pi to a TV or monitor via the HDMI cable and power up the Pi by connecting a power source to the micro-USB port. At this point, you should now see your very own operating system displaying “Hello world” on your display.



Congratulations! You are now ready to take on the *Internet of Things*!

Conclusion

Throughout this book, we have covered a lot of ground. We started our journey by learning the basics about how operating systems work. Then we were introduced to machine code and the Assembly and C programming languages. The next steps of our journey showed us how to setup a virtualized development environment which we would next use to build our operating system, after which we did just that.

We started building our own operating system by writing a bootloader entry point using the Assembly programming language followed by building our very own kernel using C. Heck, we even followed a road that helped us start our architecture library and even expand our operating system just a bit. Plus, just for good measures, we even learned how to create our own cross-compilers and how to port our operating system to run on other devices such as the Raspberry Pi.

It has taken us quite a while to get here, but we have finally made it. We have now built our very own operating system and from here on, the possibilities are endless. So, I encourage you to now take what you have learned from this book and set out into the world to build some amazing technologies. As you do, I would like to ask that you share your story with me so that I too can follow your journey as you have followed mine. I look forward to using future technologies that you develop by first building your own operating system. The technology world is going to be a better place and you will be its leader. Congratulations and thank you!

Acknowledgements

First and foremost, I would like to thank my fiancée, Rita. This is the fifth book I have written in the last six months and she has stood by me during all five books. I can't count how many times I have stayed up late to work on my books, only to wake her when coming to bed. She is a great woman - especially for putting up with me.

Second, I would like to thank Paul Brown, Chief Enterprise Architect at TIBCO Software Inc., for his feedback throughout the writing and editing process of this book. Paul has written several books of his own about Service Oriented Architecture (SOA) as well as architecting applications using TIBCO products, all of which are well worth the read.

Lastly, I would like to thank my dog, Cheerio. Just like Rita, Cheerio has (literally) been by my side while writing my books this year. As Rita slept in bed and I worked late into the nights, Cheerio slept under my desk and never moved until I moved first. Crazy dog!

Appendix

Compiling the “Print Lucus” Example

In the [Intro to the Assembly Programming Language](#) chapter, I introduced you to a tiny application that prints my name, “Lucus”, to the screen. Now that we have an environment for developing such code, it is time to show you how to write, compile, and execute that application.

To begin with, start up CentOS and click on **Applications > Accessories > gedit** which will launch a text editor where you can enter the below code. Once you have the code in gedit, you can save the file as whatever you want. For the demonstration purposes here, I have chosen to save my file as “**print_lucus.asm**” which we will reference shortly.

print_lucus.asm

```
section .text
global _start
```

```
_start:
```

```
    mov edx,len
    mov ecx,msg
    mov ebx,1
    mov eax,4
    int 0x80
```

```
    mov eax,1
    int 0x80
```

```
section .data
```

```
msg db 'Lucus', 0xa
len equ $ - msg
```

Before you can run the code above, you will first need to compile it. To do that, you can use the **nasm** utility mentioned in [Chapter 5](#). The following command line will use that tool to compile the Assembly code above so that it can run on a 64-bit operating system (which is what our CentOS development environment is). If you want to compile the code to run on a 32-bit operating system, replace “**elf64**” with “**elf32**” or simply “**elf**” (which is

shorthand for **elf32**).

```
# nasm -f elf64 print_lucus.asm
```

It is also possible to compile the code to run on other processors by replacing “**elf64**” with the format that matches your OS. To get a list of available formats, you can run the following command:

```
# nasm -hf
```

For reference purposes, here is that list.

valid output formats for -f are (* denotes default):

* bin	flat-form binary files (e.g. DOS .COM, .SYS)
ith	Intel hex
srec	Motorola S-records
aout	Linux a.out object files
aoutb	NetBSD/FreeBSD a.out object files
coff	COFF (i386) object files (e.g. DJGPP for DOS)
elf32	ELF32 (i386) object files (e.g. Linux)
elf64	ELF64 (x86_64) object files (e.g. Linux)
elfx32	ELFX32 (x86_64) object files (e.g. Linux)
as86	Linux as86 (bin86 version 0.3) object files
obj	MS-DOS 16-bit/32-bit OMF object files
win32	Microsoft Win32 (i386) object files
win64	Microsoft Win64 (x86-64) object files
rdf	Relocatable Dynamic Object File Format v2.0
ieee	IEEE-695 (LADsoft variant) object file format
macho32	NeXTstep/OpenStep/Rhapsody/Darwin/MacOS X (i386) object files
macho64	NeXTstep/OpenStep/Rhapsody/Darwin/MacOS X (x86_64) object files
dbg	Trace of all info passed to output stage
elf	ELF (short name for ELF32)
macho	MACHO (short name for MACHO32)

Once you have your code compiled, the next step will be to link the compiled output to an actual program that we can execute. To do that, you will need to run the following command, replacing “**print_lucus**” with the name of your application.

```
# ld -s -o print_lucus print_lucus.o
```

To execute the program, all you have to do is prepend a `./` before the file name like so:

```
# ./print_lucus
```

After you compile the application as shown above, you can get the hexadecimal numbers from the example in the [Intro to Machine Code](#) chapter by executing the following command:

```
# hexdump print_lucus
```

output

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000  
0000010 0002 003e 0001 0000 00b0 0040 0000 0000  
0000020 0040 0000 0000 0000 00f0 0000 0000 0000  
0000030 0000 0000 0040 0038 0002 0040 0004 0003  
0000040 0001 0000 0005 0000 0000 0000 0000 0000  
0000050 0000 0040 0000 0000 0000 0040 0000 0000  
0000060 00cd 0000 0000 0000 00cd 0000 0000 0000  
0000070 0000 0020 0000 0000 0001 0000 0006 0000  
0000080 00d0 0000 0000 0000 00d0 0060 0000 0000  
0000090 00d0 0060 0000 0000 0006 0000 0000 0000  
00000a0 0006 0000 0000 0000 0000 0020 0000 0000  
00000b0 06ba 0000 b900 00d0 0060 01bb 0000 b800  
00000c0 0004 0000 80cd 01b8 0000 cd00 0080 0000  
00000d0 754c 7563 0a73 2e00 6873 7473 7472 6261
```

```
00000e0 2e00 6574 7478 2e00 6164 6174 0000 0000
00000f0 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 000b 0000 0001 0000 0006 0000 0000 0000
0000140 00b0 0040 0000 0000 00b0 0000 0000 0000
0000150 001d 0000 0000 0000 0000 0000 0000 0000
0000160 0010 0000 0000 0000 0000 0000 0000 0000
0000170 0011 0000 0001 0000 0003 0000 0000 0000
0000180 00d0 0060 0000 0000 00d0 0000 0000 0000
0000190 0006 0000 0000 0000 0000 0000 0000 0000
00001a0 0004 0000 0000 0000 0000 0000 0000 0000
00001b0 0001 0000 0003 0000 0000 0000 0000 0000
00001c0 0000 0000 0000 0000 00d6 0000 0000 0000
00001d0 0017 0000 0000 0000 0000 0000 0000 0000
00001e0 0001 0000 0000 0000 0000 0000 0000 0000
00001f0
```

At this point, you should see a series of numbers laid out in equal rows and columns as shown in the output above. Now, I am sure you are probably thinking to yourself right now that what you are now looking at doesn't exactly match the numbers shown in the earlier chapter exactly, and there is a good reason for that. As mentioned earlier in this book, we learned that all software has to be loaded into memory before it can be executed. We also learned that memory is divided into multiple partitions and equal parts. Here, the code is broken into eight equal parts containing two bytes per part, equaling sixteen bytes per line. This is done because the **hexdump** utility defaults to using base-16 output.

The numbers in the column on the left are the numbers that are missing from the numbers shown earlier. These are the offsets that indicate how and where the application will be broken up when it gets loaded into memory and read by the CPU (i.e. the physical memory addresses). If you convert each of these hexadecimal values into decimals, you will notice that the first row begins at 0 and every row thereafter increments by 16 as shown below.

Another thing you will also notice is that there is an asterisk in the middle that separates two sections, the first being 16 rows and the second being 13 rows. This is done because, instead of repeating lines that contain the same data as the previous line, the **hexdump** utility replaces repeating lines (such as all zeros at the "240" / "00000f0" memory

address) with an asterisk.

Below is the same code as above, but converted and displayed as decimal values in place of hexadecimal values.

output

```
0 17791 17996 258 1 0 0 0 0  
16 2 62 1 0 176 64 0 0  
32 64 0 0 0 240 0 0 0  
48 0 0 64 56 2 64 4 3  
64 1 0 5 0 0 0 0 0  
80 0 64 0 0 0 64 0 0  
96 205 0 0 0 205 0 0 0  
112 0 32 0 0 1 0 6 0  
128 208 0 0 0 208 96 0 0  
144 208 96 0 0 6 0 0 0  
160 6 0 0 0 0 32 0 0  
176 1722 0 47360 208 96 443 0 47104  
192 4 0 32973 440 0 52480 128 0  
208 30028 30051 2675 11776 26739 29811 29810 25185  
224 11776 25972 29816 11776 24932 24948 0 0  
240 0 0 0 0 0 0 0 0  
*  
304 11 0 1 0 6 0 0 0  
320 176 64 0 0 176 0 0 0  
336 29 0 0 0 0 0 0 0  
352 16 0 0 0 0 0 0 0  
368 17 0 1 0 3 0 0 0  
384 208 96 0 0 208 0 0 0  
400 6 0 0 0 0 0 0 0  
416 4 0 0 0 0 0 0 0  
432 1 0 3 0 0 0 0 0  
448 0 0 0 0 214 0 0 0  
464 23 0 0 0 0 0 0 0  
480 1 0 0 0 0 0 0 0  
496
```

Complete x86 Operating System Source

As a refresher, following are the files for the for the x86 operating system in their entirety.

boot.asm

```
bits 32
global _start
extern kernel_early
extern main

section .text
align 4
dd 0x1BADB002          ; magic
dd 0x00                 ; flags
dd - (0x1BADB002 + 0x00) ; checksum
```

_start:

```
cli
mov esp, stack
call kernel_early
call main
hlt
```

section .bss

```
resb 8192
```

stack:

kernel.c

```
static char* const VGA_MEMORY = (char*)0xb8000;
```

```
static const int VGA_WIDTH = 80;
```

```
static const int VGA_HEIGHT = 25;
```

```
void kernel_early(void) {
```

```
// do some early work here
}

int main() {
    const char *str = "Hello world";
    unsigned int i = 0; // place holder for text string position
    unsigned int j = 0; // place holder for video buffer position

    while (str[i] != '\0') {
        VGA_MEMORY[j] = str[i];
        VGA_MEMORY[j + 1] = 0x07;
        i++;
        j = j + 2;
    }
    return 0;
}
```

[linker.ld](#)

```
ENTRY (_start)
SECTIONS
{
    . = 0x100000;
    .text : { *(.text) }
    .bss : { *(.bss) }
}
```

[grub.cfg](#)

```
set timeout=0
set default=0
menuentry "My Cool OS" {
    multiboot /boot/myos
}
```

Makefile

```
CC=gcc
TARGET=myos
C_FILES=./kernel.c
OBJS=$(C_FILES:.c=.o)
```

```
all compile: $(TARGET)
all: finale
.PHONY: all compile clean finale
```

```
%.o:
$(CC) -c $(@:.o=.c) -o $@ -ffreestanding -fno-exceptions -m32

$(TARGET): $(OBJS)
$(shell nasm -f elf boot.asm -o boot.o)
$(CC) -m32 -nostdlib -nodefaultlibs -lgcc boot.o $? -T linker.ld -o $(TARGET)
```

```
finale:
$(shell cd ~/Desktop/my_os/)
$(shell cp $(TARGET) ./iso/boot/$(TARGET))
$(shell grub2-mkrescue iso --output=$(TARGET).iso)
```

```
clean:
rm -f *.o $(TARGET)
find . -name \*.o | xargs --no-run-if-empty rm
```

Index

.bss 16
.data 16
.text 16
16-bits 14
32-bits 14
8-bits 14
accumulator registers 14
ascii 13
assemblers 12
Assembly 12
Assembly programming language 13
base registers 14
base-10 9
base-16 10
base-2 10
basic input output system 35
binary 9
BIOS *See* basic input output system
bit 9
boolean logic 10
bootloader 35
byte 9
C programming language 18
C_FILES 71
CentOS 24
Central Processing Unit 6
counter registers 14
CPU *See* Central Processing Unit
ctos.c 55
data registers 14

decimal	9
DHCP	29
entry point	38
extern main	40
ffreestanding	47
gcc	33
general purpose registers	14
GNOME	30
GNU GRUB	40, 49
Graphical User Interface	30
grub2-mkrescue	50
GUI	<i>See</i> graphical user interface
hex	9
hexadecimal	9, 10
io.c	68
io.h	67
ISO	25, 52
kbd.h	69
kernel	43
kernel space	43
kernel.c	64, 72
linker	48
logic gate	9
lwIP	73
machine code	9
machine language	13
make	49
Makefile	48, 49
Master Boot Record	35
MBR	<i>See</i> Master Boot Record
memcmp.c	56
memset.c	56

microkernel	43
mnemonics	12, 13
monolithic kernel	43
nasm	33
NASM	17
Network Manager	29
nibble	10
octal	9
operand	15
operating system	6
page frames	7
paged management	7
partitioned allocation	6
POST	<i>See</i> Power-On Self-Test
Power-On Self-Test	35
printf	20
procedures	18
protected mode	36
puts	20
quantum bit	9
qubit	<i>See</i> quantum bit
radix	9
real mode	36
segmented management	7
single contiguous allocation	6
strcat.c	58
strchr.c	58
strcmp.c	57
strcpy.c	57
string.h	55
strlen.c	56
strncmp.c	57

strstr.c	58
strutil.c	59
tty.c	61
tty.h	60
UI	<i>See</i> UI
uIP	73
UNIX	18
user interface	7
user space	43
Virtual Machine	24
VirtualBox	24
VirtualBox Guest Additions	31
virtualized environment	24
VM	<i>See</i> Virtual Machine
xorriso	33

Writing Your Own Toy OS

By

Krishnakumar R.

Raghu and Chitkala

Assembled by

Zhao Jiong

gohigh@sh163.net

2002-10-6

Writing Your Own Toy OS (Part I)

By [Krishnakumar R.](#)

This article is a hands-on tutorial for building a small boot sector. The first section provides the theory behind what happens at the time the computer is switched on. It also explains our plan. The second section tells all the things you should have on hand before proceeding further, and the third section deals with the programs. Our little startup program won't actually boot Linux, but it will display something on the screen.

1. Background

1.1 The Fancy Dress

The microprocessor controls the computer. At startup, every microprocessor is just another 8086. Even though you may have a brand new Pentium, it will only have the capabilities of an 8086. From this point, we can use some software and switch processor to the infamous *protected mode*. Only then can we utilize the processor's full power.

1.2 Our Role

Initially, control is in the hands of the *BIOS*. This is nothing but a collection of programs that are stored in ROM. BIOS performs the *POST* (Power On Self Test). This checks the integrity of the computer (whether the peripherals are working properly, whether the keyboard is connected, etc.). This is when you hear those beeps from the computer. If everything is okay, BIOS selects a boot device. It copies the first sector (boot sector) from the device, to address location *0x7C00*. The control is then transferred to this location. The boot device may be a floppy disk, CD-ROM, hard disk or some device of your choice. Here we will take the boot device to be a floppy disk. If we had written some code into the boot sector of the floppy, our code would be executed now. Our role is clear: just write some programs to the boot sector of the floppy.

1.3 The Plan

First write a small program in 8086 assembly (don't be frightened; I will teach you how to write it), and copy it to the boot sector of the floppy. To copy, we will code a C program. Boot the computer with that floppy, and then enjoy.

2. Things You Should Have

as86

This is an assembler. The assembly code we write is converted to an object file by this tool.

ld86

This is the linker. The object code generated by as86 is converted to actual machine language code by this tool. Machine language will be in a form that 8086 understands.

gcc

The C compiler. For now we need to write a C program to transfer our OS to the floppy.

A free floppy

A floppy will be used to store our operating system. This also is our boot device.

Good Old Linux box

You know what this is for.

as86 and ld86 will be in most of the standard distributions. If not, you can always get them from the site <http://www.cix.co.uk/~mayday/>. Both of them are included in single package, bin86. Good documentation is available at www.linux.org/docs/ldp/howto/Assembly-HOWTO/as86.html.

3. 1, 2, 3, Start!

3.1 The Boot Sector

Grab your favourite editor and type in these few lines.

```
entry start
```

```
start:  
    mov ax, #0xb800  
    mov es, ax  
    seg es  
    mov [0], #0x41  
    seg es  
    mov [1], #0x1f  
loop1: jmp loop1
```

This is an assembly language that as86 will understand. The first statement specifies the entry point where the control should enter the program. We are stating that control should initially go to label *start*. The 2nd line depicts the location of the label *start* (don't forget to put ":" after the start). The first statement that will be executed in this program is the statement just after *start*.

0xb800 is the address of the video memory. The # is for representing an immediate value. After the execution of

```
mov ax, #0xb800
```

register ax will contain the value 0xb800, that is, the address of the video memory. Now we move this value to the *es* register. *es* stands for the extra segment register. Remember that 8086 has a segmented architecture. It has segments like code segments, data segments, extra segments, etc. --hence the segment registers *cs*, *ds*, *es*. Actually, we have made the video memory our extra segment, so anything written to extra segment would go to video memory.

To display any character on the screen, you need to write two bytes to the video memory. The first is the ascii value you are going to display. The second is the attribute of the character. Attribute has to do with which colour should be used as the foreground, which for the background, should the char blink and so on. *seg es* is actually a prefix that tells which instruction is to be executed next with reference to *es* segment. So, we move value 0x41, which is the ascii value of character A, into the first byte of the video memory. Next we need to move the attribute of the character to the next byte. Here we enter 0x1f, which is the value for representing a white character on a blue background. So if we execute this program, we get a white A on a blue background. Finally, there is the loop. We need to stop the execution after the display of the character, or we have a loop that loops forever. Save the file as *boot.s*.

The idea of video memory may not be very clear, so let me explain further. Suppose we assume the screen consists of 80 columns and 25 rows. So for each line we need 160 bytes, one for each character and one for each character's attribute. If we need to write some character to column 3 then we need to skip bytes 0 and 1 as they are for the 1st column; 2 and 3 as they are for the 2nd column; and

then write our ascii value to the 4th byte and its attribute to the 5th location in the video memory.

3.2 Writing Boot Sector to Floppy

We have to write a C program that copies our code (OS code) to first sector of the floppy disk. Here it is:

```
#include <sys/types.h> /* unistd.h needs this */
#include <unistd.h>    /* contains read/write */
#include <fcntl.h>

int main()
{
    char boot_buf[512];
    int floppy_desc, file_desc;

    file_desc = open("./boot", O_RDONLY);
    read(file_desc, boot_buf, 510);
    close(file_desc);

    boot_buf[510] = 0x55;
    boot_buf[511] = 0xaa;

    floppy_desc = open("/dev/fd0", O_RDWR);
    lseek(floppy_desc, 0, SEEK_CUR);
    write(floppy_desc, boot_buf, 512);
    close(floppy_desc);
}
```

First, we open the file *boot* in read-only mode, and copy the file descriptor of the opened file to variable *file_desc*. Read from the file 510 characters or until the file ends. Here the code is small, so the latter case occurs. Be decent; close the file.

The last four lines of code open the floppy disk device (which mostly would be */dev/fd0*). It brings the head to the beginning of the file using *lseek*, then writes the 512 bytes from the buffer to floppy.

The man pages of *read*, *write*, *open* and *lseek* (refer to *man 2*) would give you enough information on what the other parameters of those functions are and how

to use them. There are two lines in between, which may be slightly mysterious. The lines:

```
boot_buf[510] = 0x55;  
boot_buf[511] = 0xaa;
```

This information is for BIOS. If BIOS is to recognize a device as a bootable device, then the device should have the values 0x55 and 0xaa at the 510th and 511th location. Now we are done. The program reads the file *boot* to a buffer named *boot_buf*. It makes the required changes to 510th and 511th bytes and then writes *boot_buf* to floppy disk. If we execute the code, the first 512 bytes of the floppy disk will contain our boot code. Save the file as *write.c*.

3.3 Let's Do It All

To make executables out of this file you need to type the following at the Linux bash prompt.

```
as86 boot.s -o boot.o
```

```
ld86 -d boot.o -o boot
```

```
cc write.c -o write
```

First, we assemble the *boot.s* to form an object file *boot.o*. Then we link this file to get the final file *boot*. The *-d* for *ld86* is for removing all headers and producing pure binary. Reading man pages for *as86* and *ld86* will clear any doubts. We then compile the C program to form an executable named *write*.

Insert a blank floppy into the floppy drive and type

```
./write
```

Reset the machine. Enter the BIOS setup and make floppy the first boot device. Put the floppy in the drive and watch the computer boot from your boot floppy.

Then you will see an 'A' (with white foreground color on a blue background). That means that the system has booted from the boot floppy we have made and then executed the boot sector program we wrote. It is now in the infinite loop we had written at the end of our boot sector. We must now reboot the computer and remove the our boot floppy to boot into Linux.

From here, we'll want to insert more code into our boot sector program, to make it do more complex things (like using BIOS interrupts, protected-mode switching, etc). The later parts (PART II, PART III etc.) of this article will guide you on further improvements. Till then GOOD BYE !

Krishnakumar R.

Krishnakumar is a final year B.Tech student at Govt. Engg. College Thrissur, Kerala, India. His journey into the land of Operating systems started with module programming in Linux. He has built a routing operating system by name GROS. (Details available at his home page: www.askus.way.to) His other interests include Networking, Device drivers, Compiler Porting and Embedded systems.

Writing Your Own Toy OS (PART II)

By [Krishnakumar R.](#)

Part I was published in April.

The next thing that any one should know after learning to make a boot sector and before switching to protected mode is, how to use the BIOS interrupts. BIOS interrupts are the low level routines provided by the BIOS to make the work of the Operating System creator easy. This part of the article would deal with BIOS interrupts.

1. Theory

1.1 Why BIOS ?

BIOS does the copying of the boot sector to the RAM and execution of code there. Besides this there are lot of things that the BIOS does. When an operating system boots up it does not have a video driver or a floppy driver or any other driver as such. To include any such driver in the boot sector is next to impossible. So some other way should be there. The BIOS comes to our help here. BIOS contains various routines we can use. For example there are ready made routines available for various purposes like, checking the equipments installed, controlling the printer, finding out memory size etc. These routines are what we call BIOS interrupts.

1.2 How do we invoke BIOS interrupts ?

In ordinary programming languages we invoke a routine by making a call to the routine. For example in a C program, if there is a routine by name display having parameters nofchar - number of characters to be displayed, attr - attribute of characters displayed is to just to call the routine that is just write the name of the routine. Here we make use of interrupts. That is we make use of assembly instruction int.

For example for printing something on the screen we call the C function like this :

```
display(nofchar, attr);
```

Equivalent to this, when we use BIOS, we write :

```
int 0x10
```

1.3 Now, how do we pass the parameters ?

Before calling the BIOS interrupt, we need to load certain values in prespecified format in the registers. Suppose we are using BIOS interrupt 13h, which is for transferring the data from the floppy to the memory. Before calling interrupt 13h we have to specify the segment address to which the data would be copied. Also we need to pass as parameters the drive number, track number, sector number, number of sectors to be transferred etc. This we do by loading the prespecified registers with the needed values. The idea will be clear after you read the explanation on the boot sector we are going to construct.

One important thing is that the same interrupt can be used for a variety of purposes. The purpose for which a particular interrupt is used depends upon the function number selected. The choice of the function is made depending on the value present in the ah register. For example interrupt 13h can be used for displaying a string as well as for getting the cursor position. If we move value 3 to register ah then the function number 3 is selected which is the function used for getting the cursor position. For displaying the string we move 13h to register ah which corresponds to displaying a string on screen.

2. What are we going to do ?

This time our source code consists of two assembly language programs and one C program. First assembly file is the boot sector code. In the boot sector we have written the code to copy the second sector of the floppy to the memory segment 0x500 (the address location is 0x5000). This we do using BIOS interrupt 13h. The code in the boot sector then transfers control to offset 0 of segment 0x500. The code in the second assembly file is for displaying a message on screen using BIOS interrupt 10h. The C program is for transferring the executable code produced from assembly file 1 to boot sector and the executable code produced from the assembly file 2 to the second sector of the floppy.

3. The boot sector

Using interrupt 13h, the boot sector loads the second sector of the floppy into memory location 0x5000 (segment address 0x500). Given below is the source code used for this purpose. Save the code to file bsect.s.

LOC1=0x500

```
entry start
start:
    mov ax, #LOC1
    mov es, ax
    mov bx, #0

    mov dl, #0
    mov dh, #0
    mov ch, #0
    mov cl, #2
    mov al, #1

    mov ah, #2

    int 0x13

    jmpi 0, #LOC1
```

The first line is similar to a macro. The next two statements might be familiar to you by now. Then we load the value 0x500 into the es register. This is the address location to which the code in the second sector of the floppy (the first sector is the boot sector) is moved to. Now we specify the offset within the segment as zero.

Next we load drive number into dl register, head number into dh register, track number into ch register, sector number into cl register and the number of sectors to be transferred to register al. So we are going to load the sector 2, of track number 0, drive number 0 to segment 0x500. All this corresponds to 1.44Mb floppy.

Moving value 2 into register ah is corresponds to choosing a function number. This is to choose from the functions provided by the interrupt 13h. We choose function number 2 which is the function used for transferring data from floppy.

Now we call interrupt 13h and finally jump to 0th offset in the segment 0x500.

4. The second sector

The code in the second sector will be given below :

```
entry start
start:
    mov    ah, #0x03
    xor    bh, bh
    int    0x10

    mov    cx, #26
    mov    bx, #0x0007
    mov    bp, #mymsg
    mov    ax, #0x1301
    int    0x10

loop1: jmp    loop1

mymsg:
    .byte  13, 10
    .ascii "Handling BIOS interrupts"
```

This code will be loaded to segment 0x500 and executed. The code here uses interrupt 10h to get the current cursor position and then to print a message.

The first three lines of code (starting from the 3rd line) are used to get the current cursor position. Here function number 3 of interrupt 13h is selected. Then we clear the value in bh register. We move the number of characters in the string to register ch. To bx we move the page number and the attribute that is to be set while displaying. Here we are planning to display white characters on black background. Then address of the message to be printed is moved to register bp. The message consists of two bytes having values 13 and 10 which correspond to an enter which is the Carriage Return (CR) and the Line Feed (LF) together. Then there is a 24 character string. Then we select the function which corresponds to printing the string and then moving the cursor. Then comes the call to interrupt. At the end comes the usual loop.

5. The C program

The source code of the C program is given below. Save it into file write.c.

```
#include <sys/types.h> /* unistd.h needs this */
#include <unistd.h>    /* contains read/write */
#include <fcntl.h>

int main()
{
```

```
char boot_buf[512];
int floppy_desc, file_desc;

file_desc = open("./bsect", O_RDONLY);

read(file_desc, boot_buf, 510);
close(file_desc);

boot_buf[510] = 0x55;
boot_buf[511] = 0xaa;

floppy_desc = open("/dev/fd0", O_RDWR);

lseek(floppy_desc, 0, SEEK_SET);
write(floppy_desc, boot_buf, 512);

file_desc = open("./sect2", O_RDONLY);
read(file_desc, boot_buf, 512);
close(file_desc);

lseek(floppy_desc, 512, SEEK_SET);
write(floppy_desc, boot_buf, 512);

close(floppy_desc);
}
```

In PART I of this article I had given the description about making the boot floppy. Here there are slight differences. We first copy the file bsect, the executable code produced from bsect.s to the boot sector. Then we copy the sect2 the executable corresponding to sect2.s the second sector of the floppy. Also the changes to be made to make the floppy bootable have also been performed.

6. Downloads

You can download the sources from

1. [bsect.s](#)

LOC1=0x500

```
entry start
start:
    mov ax, #LOC1
    mov es, ax
    mov bx, #0 ;segment offset
```

```
    mov dl, #0 ; drive no.
    mov dh, #0 ; head no.
    mov ch, #0 ; track no.
    mov cl, #2 ; sector no. ( 1..18 )
    mov al, #1 ; no. of sectors transferred
    mov ah, #2 ; function no.
    int 0x13

    jmpi 0, #LOC1
```

2. [sect2.s](#)

```
entry start
start:
    mov ah, #0x03          ; read cursor position.
    xor bh, bh
    int 0x10

    mov cx, #26            ; length of our beautiful string.
    mov bx, #0x0007          ; page 0, attribute 7 (normal)
    mov bp, #mymsg
    mov ax, #0x1301          ; write string, move cursor
    int 0x10
loop1: jmp loop1

mymsg:
    .byte 13, 10
    .ascii "Handling BIOS interrupts"
```

3. [write.c](#)

```
#include /* unistd.h needs this */
#include /* contains read/write */
#include

int main()
{
    char boot_buf[512];
    int floppy_desc, file_desc;
```

```
file_desc = open("./bsect", O_RDONLY);

read(file_desc, boot_buf, 510);
close(file_desc);

boot_buf[510] = 0x55;
boot_buf[511] = 0xaa;

floppy_desc = open("/dev/fd0", O_RDWR);

lseek(floppy_desc, 0, SEEK_SET);
write(floppy_desc, boot_buf, 512);

file_desc = open("./sect2", O_RDONLY);
read(file_desc, boot_buf, 512);
close(file_desc);

lseek(floppy_desc, 512, SEEK_SET);
write(floppy_desc, boot_buf, 512);

close(floppy_desc);
}
```

4. [Makefile](#)

```
all : bsect sect2 write

bsect : bsect.o
        ld86 -d bsect.o -o bsect

sect2 : sect2.o
        ld86 -d sect2.o -o sect2

bsect.o : bsect.s
        as86 bsect.s -o bsect.o

sect2.o : sect2.s
        as86 sect2.s -o sect2.o

write : write.c
        cc write.c -o write
```

clean :

```
rm bsect.o sect2.o bsect sect2 write
```

Remove the txt extension of the files, and type

make

at the shell prompt or you can compile everything separately. Type
as86 bsect.s -o bsect.o

ld86 -d bsect.o -o bsect

and repeat the same for sect2.s giving sect2. Compile write.c and execute it
after putting the boot floppy in to drive by typing :

```
cc write.c -o write
```

```
./write
```

7. What Next?

After booting with the floppy you can see the string being displayed. Thus we
will have used the BIOS interrupts. In the next part of this series I hope to
write about how we can switch the processor to protected mode. Till then, bye !



Krishnakumar R.

Krishnakumar is a final year B. Tech student at Govt. Engg. College Thrissur, Kerala, India. His journey into the land of Operating systems started with module programming in Linux. He has built a routing operating system by name GROS. (Details available at his home page: www.askus.way.to) His other interests include Networking, Device drivers, Compiler Porting and Embedded systems.

Writing your own Toy OS - Part III

By [Raghu and Chitkala](#)

[Krishnakumar is unable to continue this series himself due to other commitments, so he has handed it over to his junior colleagues, Raghu and Chitkala, who have written part III. -Editor.]

In Parts [I](#) and [II](#) of this series, we examined the process of using tools available with Linux to build a simple boot sector and access the system BIOS. Our toy OS will be closely modelled after a 'historic' Linux kernel - so we have to switch to protected mode real soon! This part shows you how it can be done.

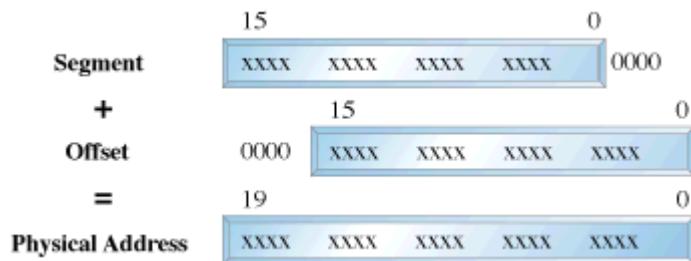
1. What is Protected Mode ?

The 80386+ provides many new features to overcome the deficiencies of 8086 which has almost no support for memory protection, virtual memory, multitasking, or memory above 640K - and still remain compatible with the 8086 family. The 386 has all the features of the 8086 and 286, with many more enhancements. As in the earlier processors, there is the real mode. Like the 286, the 386 can operate in protected mode. However, the protected mode on 386 is vastly different internally. Protected mode on the 386 offers the programmer better protection and more memory than on the 286. The purpose of protected mode is not to protect your program. The purpose is to protect everyone else (including the operating system) from your program.

1.1 Protected Mode vs Real Mode

Superficially protected mode and real mode don't seem to be very different. Both use memory segmentation, interrupts and device drivers to handle the hardware. But there are differences which justify the existence of two separate modes. In real mode, we can view memory as 64k segments at least 16bytes apart. Segmentation is handled through the use of an internal mechanism in conjunction with segment registers. The contents of these segment registers (CS, DS, SS...) form part of the physical address that the CPU places on the address bus. The physical address is generated by multiplying the segment register by 16 and then adding a 16 bit offset. It is this 16 bit offset that limits us to 64k segments.

fig 1 : Real Mode Addressing



In protected mode, segmentation is defined via a set of tables called descriptor tables. The segment registers contain pointers into these tables. There are two types of tables used to define memory segmentation : The Global Descriptor Table and The Local Descriptor Table. The GDT contains the basic descriptors that all applications can access. In real mode one segment is 64k big followed by the next in a 16 byte distance. In protected mode we can have a segment as big as 4Gb and we can put it wherever we want. The LDT contains segmentation information specific to a task or program. An OS for instance could set up a GDT with its system descriptors and for each task an LDT with appropriate descriptors. Each descriptor is 8 bytes long. The format is given below (fig 3). Each time a segment register is loaded, the base address is fetched from the appropriate table entry. The contents of the descriptor is stored in a programmer invisible register called shadow registers so that future references to the same segment can use this information instead of referencing the table each time. The physical address is formed by adding the 16 or 32 bit offset to the base address in the shadow register. These differences are made clear in figures 1 and 2.

fig 2 : Protected Mode Addressing

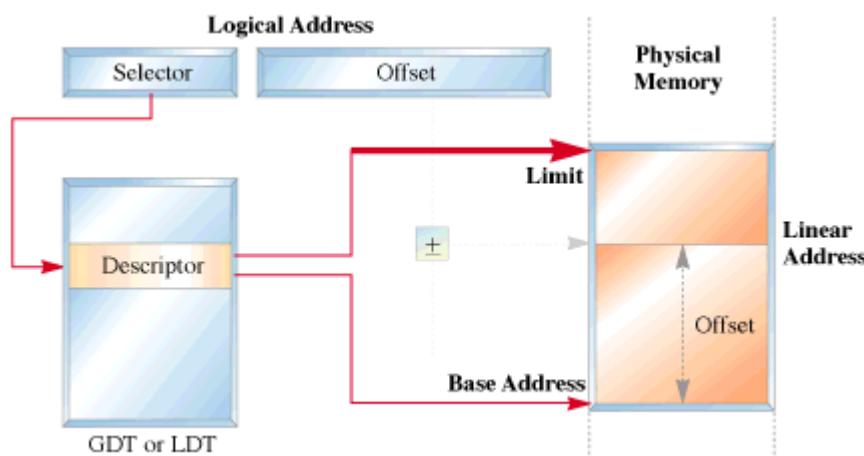
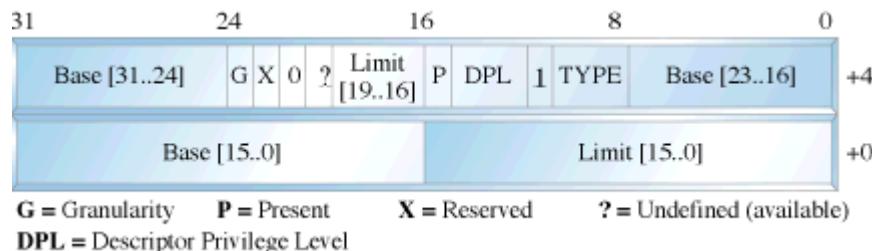


fig 3 : Segment Descriptor Format



We have yet another table called the interrupt descriptor table or the IDT. The IDT contains the interrupt descriptors. These are used to tell the processor where to find the interrupt handlers. It contains one entry per interrupt, just like in Real Mode, but the format of these entries is totally different. We are not using the IDT in our code to switch to the protected mode so further details are not given.

2. Entering Protected Mode

The 386 has four 32 bit control registers named CR0, CR1, CR2 and CR3. CR1 is reserved for future processors, and is undefined for the 386. CR0 contains bits that enable and disable paging and protection and bits that control the operation of the floating point coprocessor. CR2 and CR3 are used by the paging mechanism. We are concerned with bit 0 of the CR0 register or the PE bit or the protection enable bit. When PE = 1, the processor is said to be operating in protected mode with the segmentation mechanism we described earlier. If PE = 0, the processor operates in real mode. The 386 also has the segmentation table base registers like GDTR, LDTR and IDTR. These registers address segments that contain the descriptor tables. The GDTR points to the GDT. The 48 bit GDTR defines the base and the limit of the GDT directly with a 32 bit linear address and a 16 bit limit.

Switching to protected mode essentially implies that we set the PE bit. But there are a few other things that we must do. The program must initialise the system segments and control registers. Immediately after setting the PE bit to 1 we have to execute a jump instruction to flush the execution pipeline of any instructions that may have been fetched in the real mode. This jump is typically to the next instruction. The steps to switch to protected mode then reduces to the following :

1. Build the GDT.
2. Enable protected mode by setting the PE bit in CR0.
3. Jump to clear the prefetch queue.

We'll now give the code to perform this switching.

3. What we need

- a blank floppy
- NASM assembler

[Click here](#) to download the code.

```
org 0x07c00          ; Start address 0000:7c00
jmp short begin_boot ; Jump to start of boot routine & skip other data

bootmsg db "Our OS boot sector loading ....."
pm_mesg db "Switching to protected mode ....."

dw 512              ; Bytes per sector
db 1                ; Sectors per cluster
dw 1                ; Number of reserved sectors
db 2                ; Number of FATs
dw 0x00e0            ; Number of dirs in root
dw 0x0b40            ; Number of sectors in volume
db 0x0f0              ; Media descriptor
dw 9                ; Number of sectors per FAT
dw 18               ; Number of sectors per track
dw 2                ; Number of read/write sectors
dw 0                ; Number of hidden sectors

print_mesg :
    mov ah,0x13          ; Fn 13h of int 10h writes a whole string on screen
    mov al,0x00            ; bit 0 determines cursor pos,0->point to start after
    mov bx,0x0007          ; bh -> screen page ie 0,bl = 07 ie white on black
    mov cx,0x20            ; Length of string here 32
    mov dx,0x0000          ; dh->start cursor row,dl->start cursor column
    int 0x10              ; call bios interrupt 10h
    ret                  ; Return to calling routine

get_key :
    mov ah,0x00
    int 0x16              ; Get_Key Fn 00h of 16h, read next character
    ret

clrscr :
    mov ax,0x0600          ; Fn 06 of int 10h, scroll window up, if al = 0 clrscr
    mov cx,0x0000          ; Clear window from 0,0
    mov dx,0x174f          ; to 23,79
    mov bh,0                ; fill with colour 0
```

```
int 0x10          ; call bios interrupt 10h
ret

begin_boot :
    call clrscr      ; Clear the screen first
    mov bp,bootmesg ; Set the string ptr to message location
    call print_mesg ; Print the message
    call get_key     ; Wait till a key is pressed

bits 16
    call clrscr      ; Clear the screen
    mov ax,0xb800    ; Load gs to point to video memory
    mov gs,ax        ; We intend to display a brown A in real mode
    mov word [gs:0],0x641 ; display
    call get_key     ; Get_key again, ie display till key is pressed
    mov bp,pm_mesg ; Set string pointer
    call print_mesg ; Call print_mesg subroutine
    call get_key     ; Wait till key is pressed
    call clrscr      ; Clear the screen
    cli             ; Clear or disable interrupts
    lgdt[gdtr]       ; Load GDT
    mov eax,cr0      ; The lsb of cr0 is the protected mode bit
    or al,0x01       ; Set protected mode bit
    mov cr0,eax      ; Mov modified word to the control register
    jmp codesel:go_pm

bits 32
go_pm :
    mov ax,data sel
    mov ds,ax        ; Initialise ds & es to data segment
    mov es,ax
    mov ax,video sel ; Initialise gs to video memory
    mov gs,ax
    mov word [gs:0],0x741 ; Display white A in protected mode
    spin : jmp spin   ; Loop

bits 16
gdtr :
    dw gdt_end-gdt-1 ; Length of the gdt
    dd gdt           ; physical address of gdt

gdt
nullsel equ $-gdt      ; $->current location, so nullsel = 0h
gdt0
    dd 0             ; Null descriptor, as per convention gdt0 is 0
    dd 0             ; Each gdt entry is 8 bytes, so at 08h it is CS
    dd 0             ; In all the segment descriptor is 64 bits
```

```
codesel equ $-gdt          ; This is 8h, ie 2nd descriptor in gdt
code_gdt
    dw 0xffff          ; Code descriptor 4Gb flat segment at 0000:0000h
    dw 0x0000          ; Limit 4Gb bits 0-15 of segment descriptor
    dw 0x0000          ; Base 0h bits 16-31 of segment descriptor (sd)
    db 0x00            ; Base addr of seg 16-23 of 32bit addr, 32-39 of sd
    db 0x09a           ; P, DPL(2), S, TYPE(3), A->Present bit 1, Descriptor
                        ; privilege level 0-3, Segment descriptor 1 ie code
    db 0x0cf           ; Upper 4 bits G, D, 0, AVL ->1 segment len is page
                        ; granular, 1 default operation size is 32bit seg
                        ; Lower nibble bits 16-19 of segment limit
    db 0x00            ; Base addr of seg 24-31 of 32bit addr, 56-63 of sd
dataset equ $-gdt          ; ie 10h, beginning of next 8 bytes for data sd
data_gdt
    dw 0xffff          ; Data descriptor 4Gb flat seg at 0000:0000h
    dw 0x0000          ; Limit 4Gb
    dw 0x0000          ; Base 0000:0000h
    db 0x00            ; Descriptor format same as above
    db 0x092
    db 0x0cf
    db 0x00
videosel equ $-gdt          ; ie 18h, next gdt entry
    dw 3999            ; Limit 80*25*2-1
    dw 0x8000          ; Base 0xb8000
    db 0x0b
    db 0x92            ; present, ring 0, data, expand-up, writable
    db 0x00            ; byte granularity 16 bit
    db 0x00
gdt_end

times 510-($-$)  db 0  ; Fill bytes from present loc to 510 with 0s
                    dw 0xaa55 ; Write aa55 in bytes 511,512 to indicate that
                        ; it is a bootable sector.
```

Type in the code to a file by name abc.asm. Assemble it by typing the command **nasm abc.asm**. This will produce a file called abc. Then insert the floppy and type the following command **dd if=abc of=/dev/fd0**. This command will write the file abc to the first sector of the floppy. Then reboot the system. You should see the following sequence of messages.

- Our os booting.....
- A (Brown colour)
- Switching to protected mode....
- A (White colour)

4. The Code that does everything !

We'll first give the code to perform the switching. It is followed by a detailed explanation.

As mentioned in the previous article (Part 1) the BIOS selects the boot device and places the first sector into the address 0x7c00. We thus start writing our code at 0x7c00. This is what is implied by the `org` directive.

FUNCTIONS USED

print_msg: This routine uses the subfunction 13h of BIOS interrupt 10h to write a string to the screen. The attributes are specified by placing appropriate values in various registers. Interrupt 10h is used for various string manipulations. We store the subfn number 13h in ah which specifies that we wish to print a string. Bit 0 of the al register determines the next cursor position; if it is 0 we return to the beginning of the next line after the function call, if it is 1 the cursor is placed immediately following the last character printed.

The video memory is split into several pages called video display pages. Only one page can be displayed at a time (For further details on video memory refer Part 1). The contents of bh indicates the page number, bl specifies the colour of the character to be printed. cx holds the length of the string to be printed. Register dx specifies the cursor position. Once all the attributes have been initialised we call BIOS interrupt 10h.

get_key: We use BIOS interrupt 16h whose sub function 00h is used to get the next character from the screen. Register ah holds the subfn number.

clrscr: This function uses yet another subfn of int 10h i.e 06h to clear the screen before printing a string. To indicate this we initialise al to 0. Registers cx and dx specify the window size to be cleared; in this case it is the entire screen. Register bh indicates the colour with which the screen has to be filled; in this case it is black.

Where everything begins !!

The first assembly language statement is a short jump to the `begin_boot` code. We intend to print a brown 'A' in real-mode, set up a GDT, switch to protected mode and print a white 'A'. Both these modes use their own addressing methods.

In Real -Mode :

We use segment register gs to point to video memory. We use a CGA adapter (default base address 0xb8000). But hey we have a missing 0 in the code. Well the Real-mode segmentation unit provides the additional 0. This is a matter of convenience, as the 8086 usually does a 20bit address manipulation. This has been carried over in the real-mode addressing of the 386. The ascii value for A is 0x41; 0x06 specifies that we need a brown coloured character. The display stays till we press a key. Next let us display a message on the screen saying we are going to the world of protected mode. So let us point the bp(base pointer register to the message to be printed).

Launchpad to the protected mode :

We don't need any interrupts bothering us, while in protected mode do we? So let's disable them (interrupts that is). That is what cli does. We will enable them later. So let's start by setting up the GDT. We initialise 4 descriptors in our attempt to switch to protected mode. These descriptors initialise our code segment (code_gdt), data and stack segments (data_gdt) and the video segment in order to access the video memory. A dummy descriptor is also initialised although it's never used except if you want to triple fault of course. This is a null descriptor. Let us probe into some of the segment descriptor fields.

- The first word holds the limit of the segment, which for simplicity is assigned the maximum of FFFF (4G). For the video segment we set a predefined value of 3999 (80 cols * 25 rows * 2bytes - 1).
- The base address of the code and data segments is set to 0x0000. For the video segment it is 0xb8000 (Video Memory base address).

The GDT base address has to be loaded into GDTR system register. The gdtr segment is loaded with the size of the GDT in the first word and the base address in the next dword. The lgdt instruction then loads the gdt segment into the GDTR register. Now we are ready to actually switch to pmode. We start by setting the least significant bit of CR0 to 1 (ie the PE bit). We are not yet in full protected mode!

Section 10.3 of the INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986 states : Immediately after setting the PE flag, the initialization code must flush the processor's instruction prefetch queue by executing a JMP instruction. The 80386 fetches and decodes instructions and addresses before they are used; however, after a change into protected mode, the prefetched instruction information (which pertains to real-address mode) is no longer valid. A JMP forces the processor to discard the invalid information.

We are in protected mode now. Want to check it out? Let's get our A printed in white. For this we initialise the data and extra segments with the data segment selector (datasel). Initialise gs with the video segment selector (videosel).

To display a white 'A' move a word containing the ascii value and attribute to location [gs:0000] ie b8000 : 0000. The spin loop preserves the text on the screen until the system is rebooted.

The times instruction is used to fill in 0s in the remaining unused bytes of the sector. To indicate that this is a bootable sector we write AA55 in bytes 511, 512. That's about all.



Raghu and Chitkala

Raghu and Chitkala are seventh-semester students at the Government Engineering College, Thrissur, India. Their final-year project is porting User Mode Linux to BSD. Their interests include Operating Systems, Networking and Microcontrollers.

Writing Your Own Toy OS

By

Krishnakumar R.

Raghu and Chitkala

Assembled by

Zhao Jiong

gohigh@sh163.net

2002-10-6

Writing Your Own Toy OS (Part I)

By [Krishnakumar R.](#)

This article is a hands-on tutorial for building a small boot sector. The first section provides the theory behind what happens at the time the computer is switched on. It also explains our plan. The second section tells all the things you should have on hand before proceeding further, and the third section deals with the programs. Our little startup program won't actually boot Linux, but it will display something on the screen.

1. Background

1.1 The Fancy Dress

The microprocessor controls the computer. At startup, every microprocessor is just another 8086. Even though you may have a brand new Pentium, it will only have the capabilities of an 8086. From this point, we can use some software and switch processor to the infamous *protected mode*. Only then can we utilize the processor's full power.

1.2 Our Role

Initially, control is in the hands of the *BIOS*. This is nothing but a collection of programs that are stored in ROM. BIOS performs the *POST* (Power On Self Test). This checks the integrity of the computer (whether the peripherals are working properly, whether the keyboard is connected, etc.). This is when you hear those beeps from the computer. If everything is okay, BIOS selects a boot device. It copies the first sector (boot sector) from the device, to address location *0x7C00*. The control is then transferred to this location. The boot device may be a floppy disk, CD-ROM, hard disk or some device of your choice. Here we will take the boot device to be a floppy disk. If we had written some code into the boot sector of the floppy, our code would be executed now. Our role is clear: just write some programs to the boot sector of the floppy.

1.3 The Plan

First write a small program in 8086 assembly (don't be frightened; I will teach you how to write it), and copy it to the boot sector of the floppy. To copy, we will code a C program. Boot the computer with that floppy, and then enjoy.

2. Things You Should Have

as86

This is an assembler. The assembly code we write is converted to an object file by this tool.

ld86

This is the linker. The object code generated by as86 is converted to actual machine language code by this tool. Machine language will be in a form that 8086 understands.

gcc

The C compiler. For now we need to write a C program to transfer our OS to the floppy.

A free floppy

A floppy will be used to store our operating system. This also is our boot device.

Good Old Linux box

You know what this is for.

as86 and ld86 will be in most of the standard distributions. If not, you can always get them from the site <http://www.cix.co.uk/~mayday/>. Both of them are included in single package, bin86. Good documentation is available at www.linux.org/docs/ldp/howto/Assembly-HOWTO/as86.html.

3. 1, 2, 3, Start!

3.1 The Boot Sector

Grab your favourite editor and type in these few lines.

```
entry start
```

```
start:  
    mov ax, #0xb800  
    mov es, ax  
    seg es  
    mov [0], #0x41  
    seg es  
    mov [1], #0x1f  
loop1: jmp loop1
```

This is an assembly language that as86 will understand. The first statement specifies the entry point where the control should enter the program. We are stating that control should initially go to label *start*. The 2nd line depicts the location of the label *start* (don't forget to put ":" after the start). The first statement that will be executed in this program is the statement just after *start*.

0xb800 is the address of the video memory. The # is for representing an immediate value. After the execution of

```
mov ax, #0xb800
```

register ax will contain the value 0xb800, that is, the address of the video memory. Now we move this value to the *es* register. *es* stands for the extra segment register. Remember that 8086 has a segmented architecture. It has segments like code segments, data segments, extra segments, etc. --hence the segment registers *cs*, *ds*, *es*. Actually, we have made the video memory our extra segment, so anything written to extra segment would go to video memory.

To display any character on the screen, you need to write two bytes to the video memory. The first is the ascii value you are going to display. The second is the attribute of the character. Attribute has to do with which colour should be used as the foreground, which for the background, should the char blink and so on. *seg es* is actually a prefix that tells which instruction is to be executed next with reference to *es* segment. So, we move value 0x41, which is the ascii value of character A, into the first byte of the video memory. Next we need to move the attribute of the character to the next byte. Here we enter 0x1f, which is the value for representing a white character on a blue background. So if we execute this program, we get a white A on a blue background. Finally, there is the loop. We need to stop the execution after the display of the character, or we have a loop that loops forever. Save the file as *boot.s*.

The idea of video memory may not be very clear, so let me explain further. Suppose we assume the screen consists of 80 columns and 25 rows. So for each line we need 160 bytes, one for each character and one for each character's attribute. If we need to write some character to column 3 then we need to skip bytes 0 and 1 as they are for the 1st column; 2 and 3 as they are for the 2nd column; and

then write our ascii value to the 4th byte and its attribute to the 5th location in the video memory.

3.2 Writing Boot Sector to Floppy

We have to write a C program that copies our code (OS code) to first sector of the floppy disk. Here it is:

```
#include <sys/types.h> /* unistd.h needs this */
#include <unistd.h>    /* contains read/write */
#include <fcntl.h>

int main()
{
    char boot_buf[512];
    int floppy_desc, file_desc;

    file_desc = open("./boot", O_RDONLY);
    read(file_desc, boot_buf, 510);
    close(file_desc);

    boot_buf[510] = 0x55;
    boot_buf[511] = 0xaa;

    floppy_desc = open("/dev/fd0", O_RDWR);
    lseek(floppy_desc, 0, SEEK_CUR);
    write(floppy_desc, boot_buf, 512);
    close(floppy_desc);
}
```

First, we open the file *boot* in read-only mode, and copy the file descriptor of the opened file to variable *file_desc*. Read from the file 510 characters or until the file ends. Here the code is small, so the latter case occurs. Be decent; close the file.

The last four lines of code open the floppy disk device (which mostly would be */dev/fd0*). It brings the head to the beginning of the file using *lseek*, then writes the 512 bytes from the buffer to floppy.

The man pages of *read*, *write*, *open* and *lseek* (refer to *man 2*) would give you enough information on what the other parameters of those functions are and how

to use them. There are two lines in between, which may be slightly mysterious. The lines:

```
boot_buf[510] = 0x55;  
boot_buf[511] = 0xaa;
```

This information is for BIOS. If BIOS is to recognize a device as a bootable device, then the device should have the values 0x55 and 0xaa at the 510th and 511th location. Now we are done. The program reads the file *boot* to a buffer named *boot_buf*. It makes the required changes to 510th and 511th bytes and then writes *boot_buf* to floppy disk. If we execute the code, the first 512 bytes of the floppy disk will contain our boot code. Save the file as *write.c*.

3.3 Let's Do It All

To make executables out of this file you need to type the following at the Linux bash prompt.

```
as86 boot.s -o boot.o
```

```
ld86 -d boot.o -o boot
```

```
cc write.c -o write
```

First, we assemble the *boot.s* to form an object file *boot.o*. Then we link this file to get the final file *boot*. The *-d* for *ld86* is for removing all headers and producing pure binary. Reading man pages for *as86* and *ld86* will clear any doubts. We then compile the C program to form an executable named *write*.

Insert a blank floppy into the floppy drive and type

```
./write
```

Reset the machine. Enter the BIOS setup and make floppy the first boot device. Put the floppy in the drive and watch the computer boot from your boot floppy.

Then you will see an 'A' (with white foreground color on a blue background). That means that the system has booted from the boot floppy we have made and then executed the boot sector program we wrote. It is now in the infinite loop we had written at the end of our boot sector. We must now reboot the computer and remove the our boot floppy to boot into Linux.

From here, we'll want to insert more code into our boot sector program, to make it do more complex things (like using BIOS interrupts, protected-mode switching, etc). The later parts (PART II, PART III etc.) of this article will guide you on further improvements. Till then GOOD BYE !

Krishnakumar R.

Krishnakumar is a final year B.Tech student at Govt. Engg. College Thrissur, Kerala, India. His journey into the land of Operating systems started with module programming in Linux. He has built a routing operating system by name GROS. (Details available at his home page: www.askus.way.to) His other interests include Networking, Device drivers, Compiler Porting and Embedded systems.

Writing Your Own Toy OS (PART II)

By [Krishnakumar R.](#)

Part I was published in April.

The next thing that any one should know after learning to make a boot sector and before switching to protected mode is, how to use the BIOS interrupts. BIOS interrupts are the low level routines provided by the BIOS to make the work of the Operating System creator easy. This part of the article would deal with BIOS interrupts.

1. Theory

1.1 Why BIOS ?

BIOS does the copying of the boot sector to the RAM and execution of code there. Besides this there are lot of things that the BIOS does. When an operating system boots up it does not have a video driver or a floppy driver or any other driver as such. To include any such driver in the boot sector is next to impossible. So some other way should be there. The BIOS comes to our help here. BIOS contains various routines we can use. For example there are ready made routines available for various purposes like, checking the equipments installed, controlling the printer, finding out memory size etc. These routines are what we call BIOS interrupts.

1.2 How do we invoke BIOS interrupts ?

In ordinary programming languages we invoke a routine by making a call to the routine. For example in a C program, if there is a routine by name display having parameters nofchar - number of characters to be displayed, attr - attribute of characters displayed is to just to call the routine that is just write the name of the routine. Here we make use of interrupts. That is we make use of assembly instruction int.

For example for printing something on the screen we call the C function like this :

```
display(noofchar, attr);
```

Equivalent to this, when we use BIOS, we write :

```
int 0x10
```

1.3 Now, how do we pass the parameters ?

Before calling the BIOS interrupt, we need to load certain values in prespecified format in the registers. Suppose we are using BIOS interrupt 13h, which is for transferring the data from the floppy to the memory. Before calling interrupt 13h we have to specify the segment address to which the data would be copied. Also we need to pass as parameters the drive number, track number, sector number, number of sectors to be transferred etc. This we do by loading the prespecified registers with the needed values. The idea will be clear after you read the explanation on the boot sector we are going to construct.

One important thing is that the same interrupt can be used for a variety of purposes. The purpose for which a particular interrupt is used depends upon the function number selected. The choice of the function is made depending on the value present in the ah register. For example interrupt 13h can be used for displaying a string as well as for getting the cursor position. If we move value 3 to register ah then the function number 3 is selected which is the function used for getting the cursor position. For displaying the string we move 13h to register ah which corresponds to displaying a string on screen.

2. What are we going to do ?

This time our source code consists of two assembly language programs and one C program. First assembly file is the boot sector code. In the boot sector we have written the code to copy the second sector of the floppy to the memory segment 0x500 (the address location is 0x5000). This we do using BIOS interrupt 13h. The code in the boot sector then transfers control to offset 0 of segment 0x500. The code in the second assembly file is for displaying a message on screen using BIOS interrupt 10h. The C program is for transferring the executable code produced from assembly file 1 to boot sector and the executable code produced from the assembly file 2 to the second sector of the floppy.

3. The boot sector

Using interrupt 13h, the boot sector loads the second sector of the floppy into memory location 0x5000 (segment address 0x500). Given below is the source code used for this purpose. Save the code to file bsect.s.

LOC1=0x500

```
entry start
start:
    mov ax, #LOC1
    mov es, ax
    mov bx, #0

    mov dl, #0
    mov dh, #0
    mov ch, #0
    mov cl, #2
    mov al, #1

    mov ah, #2

    int 0x13

    jmpi 0, #LOC1
```

The first line is similar to a macro. The next two statements might be familiar to you by now. Then we load the value 0x500 into the es register. This is the address location to which the code in the second sector of the floppy (the first sector is the boot sector) is moved to. Now we specify the offset within the segment as zero.

Next we load drive number into dl register, head number into dh register, track number into ch register, sector number into cl register and the number of sectors to be transferred to register al. So we are going to load the sector 2, of track number 0, drive number 0 to segment 0x500. All this corresponds to 1.44Mb floppy.

Moving value 2 into register ah is corresponds to choosing a function number. This is to choose from the functions provided by the interrupt 13h. We choose function number 2 which is the function used for transferring data from floppy.

Now we call interrupt 13h and finally jump to 0th offset in the segment 0x500.

4. The second sector

The code in the second sector will be given below :

```
entry start
start:
    mov    ah, #0x03
    xor    bh, bh
    int    0x10

    mov    cx, #26
    mov    bx, #0x0007
    mov    bp, #mymsg
    mov    ax, #0x1301
    int    0x10

loop1: jmp    loop1

mymsg:
    .byte 13, 10
    .ascii "Handling BIOS interrupts"
```

This code will be loaded to segment 0x500 and executed. The code here uses interrupt 10h to get the current cursor position and then to print a message.

The first three lines of code (starting from the 3rd line) are used to get the current cursor position. Here function number 3 of interrupt 13h is selected. Then we clear the value in bh register. We move the number of characters in the string to register ch. To bx we move the page number and the attribute that is to be set while displaying. Here we are planning to display white characters on black background. Then address of the message to be printed is moved to register bp. The message consists of two bytes having values 13 and 10 which correspond to an enter which is the Carriage Return (CR) and the Line Feed (LF) together. Then there is a 24 character string. Then we select the function which corresponds to printing the string and then moving the cursor. Then comes the call to interrupt. At the end comes the usual loop.

5. The C program

The source code of the C program is given below. Save it into file write.c.

```
#include <sys/types.h> /* unistd.h needs this */
#include <unistd.h>    /* contains read/write */
#include <fcntl.h>

int main()
{
```

```
char boot_buf[512];
int floppy_desc, file_desc;

file_desc = open("./bsect", O_RDONLY);

read(file_desc, boot_buf, 510);
close(file_desc);

boot_buf[510] = 0x55;
boot_buf[511] = 0xaa;

floppy_desc = open("/dev/fd0", O_RDWR);

lseek(floppy_desc, 0, SEEK_SET);
write(floppy_desc, boot_buf, 512);

file_desc = open("./sect2", O_RDONLY);
read(file_desc, boot_buf, 512);
close(file_desc);

lseek(floppy_desc, 512, SEEK_SET);
write(floppy_desc, boot_buf, 512);

close(floppy_desc);
}
```

In PART I of this article I had given the description about making the boot floppy. Here there are slight differences. We first copy the file bsect, the executable code produced from bsect.s to the boot sector. Then we copy the sect2 the executable corresponding to sect2.s the second sector of the floppy. Also the changes to be made to make the floppy bootable have also been performed.

6. Downloads

You can download the sources from

1. [bsect.s](#)

LOC1=0x500

```
entry start
start:
    mov ax, #LOC1
    mov es, ax
    mov bx, #0 ;segment offset
```

```
    mov dl, #0 ; drive no.
    mov dh, #0 ; head no.
    mov ch, #0 ; track no.
    mov cl, #2 ; sector no. ( 1..18 )
    mov al, #1 ; no. of sectors transferred
    mov ah, #2 ; function no.
    int 0x13

    jmpi 0, #LOC1
```

2. [sect2.s](#)

```
entry start
start:
    mov ah, #0x03          ; read cursor position.
    xor bh, bh
    int 0x10

    mov cx, #26            ; length of our beautiful string.
    mov bx, #0x0007          ; page 0, attribute 7 (normal)
    mov bp, #mymsg
    mov ax, #0x1301          ; write string, move cursor
    int 0x10
loop1: jmp loop1

mymsg:
    .byte 13, 10
    .ascii "Handling BIOS interrupts"
```

3. [write.c](#)

```
#include /* unistd.h needs this */
#include /* contains read/write */
#include

int main()
{
    char boot_buf[512];
    int floppy_desc, file_desc;
```

```
file_desc = open("./bsect", O_RDONLY);

read(file_desc, boot_buf, 510);
close(file_desc);

boot_buf[510] = 0x55;
boot_buf[511] = 0xaa;

floppy_desc = open("/dev/fd0", O_RDWR);

lseek(floppy_desc, 0, SEEK_SET);
write(floppy_desc, boot_buf, 512);

file_desc = open("./sect2", O_RDONLY);
read(file_desc, boot_buf, 512);
close(file_desc);

lseek(floppy_desc, 512, SEEK_SET);
write(floppy_desc, boot_buf, 512);

close(floppy_desc);
}
```

4. [Makefile](#)

```
all : bsect sect2 write

bsect : bsect.o
        ld86 -d bsect.o -o bsect

sect2 : sect2.o
        ld86 -d sect2.o -o sect2

bsect.o : bsect.s
        as86 bsect.s -o bsect.o

sect2.o : sect2.s
        as86 sect2.s -o sect2.o

write : write.c
        cc write.c -o write
```

clean :

```
rm bsect.o sect2.o bsect sect2 write
```

Remove the txt extension of the files, and type

make

at the shell prompt or you can compile everything separately. Type
as86 bsect.s -o bsect.o

ld86 -d bsect.o -o bsect

and repeat the same for sect2.s giving sect2. Compile write.c and execute it
after putting the boot floppy in to drive by typing :

```
cc write.c -o write
```

```
./write
```

7. What Next?

After booting with the floppy you can see the string being displayed. Thus we
will have used the BIOS interrupts. In the next part of this series I hope to
write about how we can switch the processor to protected mode. Till then, bye !



Krishnakumar R.

Krishnakumar is a final year B. Tech student at Govt. Engg. College Thrissur, Kerala, India. His journey into the land of Operating systems started with module programming in Linux. He has built a routing operating system by name GROS. (Details available at his home page: www.askus.way.to) His other interests include Networking, Device drivers, Compiler Porting and Embedded systems.

Writing your own Toy OS - Part III

By [Raghu and Chitkala](#)

[Krishnakumar is unable to continue this series himself due to other commitments, so he has handed it over to his junior colleagues, Raghu and Chitkala, who have written part III. -Editor.]

In Parts [I](#) and [II](#) of this series, we examined the process of using tools available with Linux to build a simple boot sector and access the system BIOS. Our toy OS will be closely modelled after a 'historic' Linux kernel - so we have to switch to protected mode real soon! This part shows you how it can be done.

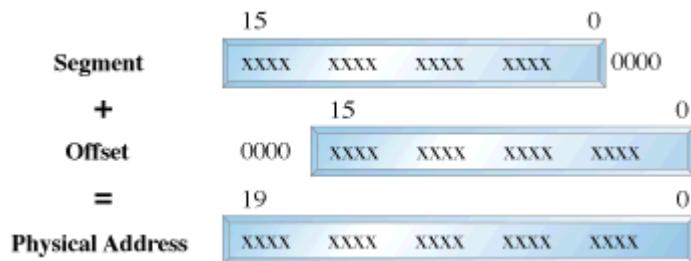
1. What is Protected Mode ?

The 80386+ provides many new features to overcome the deficiencies of 8086 which has almost no support for memory protection, virtual memory, multitasking, or memory above 640K - and still remain compatible with the 8086 family. The 386 has all the features of the 8086 and 286, with many more enhancements. As in the earlier processors, there is the real mode. Like the 286, the 386 can operate in protected mode. However, the protected mode on 386 is vastly different internally. Protected mode on the 386 offers the programmer better protection and more memory than on the 286. The purpose of protected mode is not to protect your program. The purpose is to protect everyone else (including the operating system) from your program.

1.1 Protected Mode vs Real Mode

Superficially protected mode and real mode don't seem to be very different. Both use memory segmentation, interrupts and device drivers to handle the hardware. But there are differences which justify the existence of two separate modes. In real mode, we can view memory as 64k segments at least 16bytes apart. Segmentation is handled through the use of an internal mechanism in conjunction with segment registers. The contents of these segment registers (CS, DS, SS...) form part of the physical address that the CPU places on the address bus. The physical address is generated by multiplying the segment register by 16 and then adding a 16 bit offset. It is this 16 bit offset that limits us to 64k segments.

fig 1 : Real Mode Addressing



In protected mode, segmentation is defined via a set of tables called descriptor tables. The segment registers contain pointers into these tables. There are two types of tables used to define memory segmentation : The Global Descriptor Table and The Local Descriptor Table. The GDT contains the basic descriptors that all applications can access. In real mode one segment is 64k big followed by the next in a 16 byte distance. In protected mode we can have a segment as big as 4Gb and we can put it wherever we want. The LDT contains segmentation information specific to a task or program. An OS for instance could set up a GDT with its system descriptors and for each task an LDT with appropriate descriptors. Each descriptor is 8 bytes long. The format is given below (fig 3). Each time a segment register is loaded, the base address is fetched from the appropriate table entry. The contents of the descriptor is stored in a programmer invisible register called shadow registers so that future references to the same segment can use this information instead of referencing the table each time. The physical address is formed by adding the 16 or 32 bit offset to the base address in the shadow register. These differences are made clear in figures 1 and 2.

fig 2 : Protected Mode Addressing

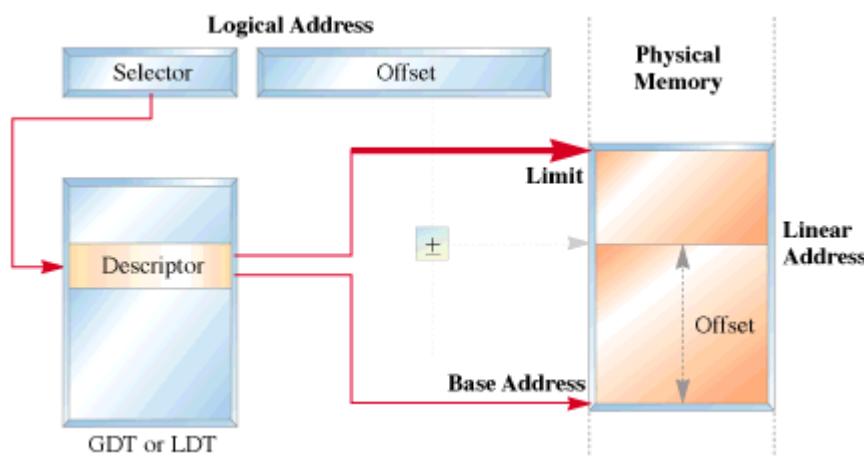
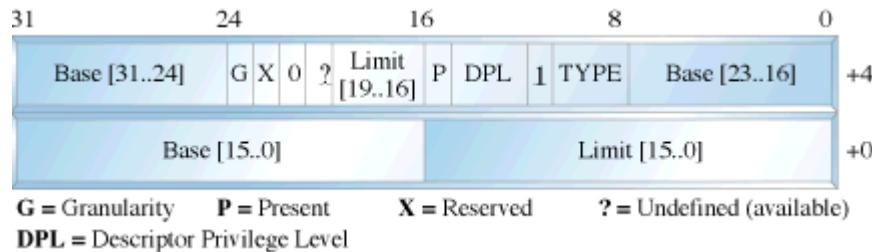


fig 3 : Segment Descriptor Format



We have yet another table called the interrupt descriptor table or the IDT. The IDT contains the interrupt descriptors. These are used to tell the processor where to find the interrupt handlers. It contains one entry per interrupt, just like in Real Mode, but the format of these entries is totally different. We are not using the IDT in our code to switch to the protected mode so further details are not given.

2. Entering Protected Mode

The 386 has four 32 bit control registers named CR0, CR1, CR2 and CR3. CR1 is reserved for future processors, and is undefined for the 386. CR0 contains bits that enable and disable paging and protection and bits that control the operation of the floating point coprocessor. CR2 and CR3 are used by the paging mechanism. We are concerned with bit 0 of the CR0 register or the PE bit or the protection enable bit. When PE = 1, the processor is said to be operating in protected mode with the segmentation mechanism we described earlier. If PE = 0, the processor operates in real mode. The 386 also has the segmentation table base registers like GDTR, LDTR and IDTR. These registers address segments that contain the descriptor tables. The GDTR points to the GDT. The 48 bit GDTR defines the base and the limit of the GDT directly with a 32 bit linear address and a 16 bit limit.

Switching to protected mode essentially implies that we set the PE bit. But there are a few other things that we must do. The program must initialise the system segments and control registers. Immediately after setting the PE bit to 1 we have to execute a jump instruction to flush the execution pipeline of any instructions that may have been fetched in the real mode. This jump is typically to the next instruction. The steps to switch to protected mode then reduces to the following :

1. Build the GDT.
2. Enable protected mode by setting the PE bit in CR0.
3. Jump to clear the prefetch queue.

We'll now give the code to perform this switching.

3. What we need

- a blank floppy
- NASM assembler

[Click here](#) to download the code.

```
org 0x07c00          ; Start address 0000:7c00
jmp short begin_boot ; Jump to start of boot routine & skip other data

bootmsg db "Our OS boot sector loading ....."
pm_mesg db "Switching to protected mode ....."

dw 512              ; Bytes per sector
db 1                ; Sectors per cluster
dw 1                ; Number of reserved sectors
db 2                ; Number of FATs
dw 0x00e0            ; Number of dirs in root
dw 0x0b40            ; Number of sectors in volume
db 0x0f0              ; Media descriptor
dw 9                ; Number of sectors per FAT
dw 18               ; Number of sectors per track
dw 2                ; Number of read/write sectors
dw 0                ; Number of hidden sectors

print_mesg :
    mov ah,0x13          ; Fn 13h of int 10h writes a whole string on screen
    mov al,0x00            ; bit 0 determines cursor pos,0->point to start after
    mov bx,0x0007          ; bh -> screen page ie 0,bl = 07 ie white on black
    mov cx,0x20            ; Length of string here 32
    mov dx,0x0000          ; dh->start cursor row,dl->start cursor column
    int 0x10              ; call bios interrupt 10h
    ret                  ; Return to calling routine

get_key :
    mov ah,0x00
    int 0x16              ; Get_Key Fn 00h of 16h, read next character
    ret

clrscr :
    mov ax,0x0600          ; Fn 06 of int 10h, scroll window up, if al = 0 clrscr
    mov cx,0x0000          ; Clear window from 0,0
    mov dx,0x174f          ; to 23,79
    mov bh,0                ; fill with colour 0
```

```
int 0x10          ; call bios interrupt 10h
ret

begin_boot :
    call clrscr      ; Clear the screen first
    mov bp,bootmesg ; Set the string ptr to message location
    call print_mesg ; Print the message
    call get_key     ; Wait till a key is pressed

bits 16
    call clrscr      ; Clear the screen
    mov ax,0xb800    ; Load gs to point to video memory
    mov gs,ax        ; We intend to display a brown A in real mode
    mov word [gs:0],0x641 ; display
    call get_key     ; Get_key again, ie display till key is pressed
    mov bp,pm_mesg ; Set string pointer
    call print_mesg ; Call print_mesg subroutine
    call get_key     ; Wait till key is pressed
    call clrscr      ; Clear the screen
    cli             ; Clear or disable interrupts
    lgdt[gdtr]       ; Load GDT
    mov eax,cr0      ; The lsb of cr0 is the protected mode bit
    or al,0x01       ; Set protected mode bit
    mov cr0,eax      ; Mov modified word to the control register
    jmp codesel:go_pm

bits 32
go_pm :
    mov ax,data sel
    mov ds,ax        ; Initialise ds & es to data segment
    mov es,ax
    mov ax,video sel ; Initialise gs to video memory
    mov gs,ax
    mov word [gs:0],0x741 ; Display white A in protected mode
    spin : jmp spin   ; Loop

bits 16
gdtr :
    dw gdt_end-gdt-1 ; Length of the gdt
    dd gdt           ; physical address of gdt

gdt
nullsel equ $-gdt      ; $->current location, so nullsel = 0h
gdt0
    dd 0             ; Null descriptor, as per convention gdt0 is 0
    dd 0             ; Each gdt entry is 8 bytes, so at 08h it is CS
    dd 0             ; In all the segment descriptor is 64 bits
```

```
codesel equ $-gdt          ; This is 8h, ie 2nd descriptor in gdt
code_gdt
    dw 0xffff          ; Code descriptor 4Gb flat segment at 0000:0000h
    dw 0x0000          ; Limit 4Gb bits 0-15 of segment descriptor
    dw 0x0000          ; Base 0h bits 16-31 of segment descriptor (sd)
    db 0x00            ; Base addr of seg 16-23 of 32bit addr, 32-39 of sd
    db 0x09a           ; P, DPL(2), S, TYPE(3), A->Present bit 1, Descriptor
                        ; privilege level 0-3, Segment descriptor 1 ie code
    db 0x0cf           ; Upper 4 bits G, D, 0, AVL ->1 segment len is page
                        ; granular, 1 default operation size is 32bit seg
                        ; Lower nibble bits 16-19 of segment limit
    db 0x00            ; Base addr of seg 24-31 of 32bit addr, 56-63 of sd
dataset equ $-gdt          ; ie 10h, beginning of next 8 bytes for data sd
data_gdt
    dw 0xffff          ; Data descriptor 4Gb flat seg at 0000:0000h
    dw 0x0000          ; Limit 4Gb
    dw 0x0000          ; Base 0000:0000h
    db 0x00            ; Descriptor format same as above
    db 0x092
    db 0x0cf
    db 0x00
videosel equ $-gdt          ; ie 18h, next gdt entry
    dw 3999            ; Limit 80*25*2-1
    dw 0x8000          ; Base 0xb8000
    db 0x0b
    db 0x92            ; present, ring 0, data, expand-up, writable
    db 0x00            ; byte granularity 16 bit
    db 0x00
gdt_end

times 510-($-$)  db 0  ; Fill bytes from present loc to 510 with 0s
                    dw 0xaa55 ; Write aa55 in bytes 511,512 to indicate that
                        ; it is a bootable sector.
```

Type in the code to a file by name abc.asm. Assemble it by typing the command **nasm abc.asm**. This will produce a file called abc. Then insert the floppy and type the following command **dd if=abc of=/dev/fd0**. This command will write the file abc to the first sector of the floppy. Then reboot the system. You should see the following sequence of messages.

- Our os booting.....
- A (Brown colour)
- Switching to protected mode....
- A (White colour)

4. The Code that does everything !

We'll first give the code to perform the switching. It is followed by a detailed explanation.

As mentioned in the previous article (Part 1) the BIOS selects the boot device and places the first sector into the address 0x7c00. We thus start writing our code at 0x7c00. This is what is implied by the `org` directive.

FUNCTIONS USED

print_msg: This routine uses the subfunction 13h of BIOS interrupt 10h to write a string to the screen. The attributes are specified by placing appropriate values in various registers. Interrupt 10h is used for various string manipulations. We store the subfn number 13h in ah which specifies that we wish to print a string. Bit 0 of the al register determines the next cursor position; if it is 0 we return to the beginning of the next line after the function call, if it is 1 the cursor is placed immediately following the last character printed.

The video memory is split into several pages called video display pages. Only one page can be displayed at a time (For further details on video memory refer Part 1). The contents of bh indicates the page number, bl specifies the colour of the character to be printed. cx holds the length of the string to be printed. Register dx specifies the cursor position. Once all the attributes have been initialised we call BIOS interrupt 10h.

get_key: We use BIOS interrupt 16h whose sub function 00h is used to get the next character from the screen. Register ah holds the subfn number.

clrscr: This function uses yet another subfn of int 10h i.e 06h to clear the screen before printing a string. To indicate this we initialise al to 0. Registers cx and dx specify the window size to be cleared; in this case it is the entire screen. Register bh indicates the colour with which the screen has to be filled; in this case it is black.

Where everything begins !!

The first assembly language statement is a short jump to the `begin_boot` code. We intend to print a brown 'A' in real-mode, set up a GDT, switch to protected mode and print a white 'A'. Both these modes use their own addressing methods.

In Real -Mode :

We use segment register gs to point to video memory. We use a CGA adapter (default base address 0xb8000). But hey we have a missing 0 in the code. Well the Real-mode segmentation unit provides the additional 0. This is a matter of convenience, as the 8086 usually does a 20bit address manipulation. This has been carried over in the real-mode addressing of the 386. The ascii value for A is 0x41; 0x06 specifies that we need a brown coloured character. The display stays till we press a key. Next let us display a message on the screen saying we are going to the world of protected mode. So let us point the bp(base pointer register to the message to be printed).

Launchpad to the protected mode :

We don't need any interrupts bothering us, while in protected mode do we? So let's disable them (interrupts that is). That is what cli does. We will enable them later. So let's start by setting up the GDT. We initialise 4 descriptors in our attempt to switch to protected mode. These descriptors initialise our code segment (code_gdt), data and stack segments (data_gdt) and the video segment in order to access the video memory. A dummy descriptor is also initialised although it's never used except if you want to triple fault of course. This is a null descriptor. Let us probe into some of the segment descriptor fields.

- The first word holds the limit of the segment, which for simplicity is assigned the maximum of FFFF (4G). For the video segment we set a predefined value of 3999 (80 cols * 25 rows * 2bytes - 1).
- The base address of the code and data segments is set to 0x0000. For the video segment it is 0xb8000 (Video Memory base address).

The GDT base address has to be loaded into GDTR system register. The gdtr segment is loaded with the size of the GDT in the first word and the base address in the next dword. The lgdt instruction then loads the gdt segment into the GDTR register. Now we are ready to actually switch to pmode. We start by setting the least significant bit of CR0 to 1 (ie the PE bit). We are not yet in full protected mode!

Section 10.3 of the INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986 states : Immediately after setting the PE flag, the initialization code must flush the processor's instruction prefetch queue by executing a JMP instruction. The 80386 fetches and decodes instructions and addresses before they are used; however, after a change into protected mode, the prefetched instruction information (which pertains to real-address mode) is no longer valid. A JMP forces the processor to discard the invalid information.

We are in protected mode now. Want to check it out? Let's get our A printed in white. For this we initialise the data and extra segments with the data segment selector (datasel). Initialise gs with the video segment selector (videosel).

To display a white 'A' move a word containing the ascii value and attribute to location [gs:0000] ie b8000 : 0000. The spin loop preserves the text on the screen until the system is rebooted.

The times instruction is used to fill in 0s in the remaining unused bytes of the sector. To indicate that this is a bootable sector we write AA55 in bytes 511, 512. That's about all.



Raghu and Chitkala

Raghu and Chitkala are seventh-semester students at the Government Engineering College, Thrissur, India. Their final-year project is porting User Mode Linux to BSD. Their interests include Operating Systems, Networking and Microcontrollers.

**HOW TO
MAKE AN**

**OPERATING
SYSTEM**

FROM SCRATCH

Samy Pessé

Published
with GitBook

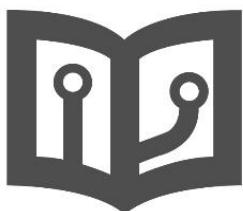


Table of Contents

1. [Introduction](#)
2. [Introduction about the x86 architecture and about our OS](#)
3. [Setup the development environment](#)
4. [First boot with GRUB](#)
5. [Backbone of the OS and C++ runtime](#)
6. [Base classes for managing x86 architecture](#)
7. [GDT](#)
8. [IDT and interrupts](#)
9. [Memory management: physical and virtual](#)
10. Process management and multitasking
11. External program execution: ELF files
12. Userland and syscalls
13. Modular drivers
14. Some basics modules: console, keyboard
15. IDE Hard disks
16. DOS Partitions
17. EXT2 read-only filesystems
18. Standard C library (libC)
19. UNIX basic tools: sh, cat
20. Lua interpreter

How to Make a Computer Operating System

Online book about how to write a computer operating system in C/C++ from scratch.

Caution: This repository is a remake of my old course. It was written several years ago [as one of my first projects when I was in High School](#), I'm still refactoring some parts. The original course was in French and I'm not an English native. I'm going to continue and improve this course in my free-time.

Book: An online version is available at <http://samypesse.gitbooks.io/how-to-create-an-operating-system/> (PDF, Mobi and ePub). It was been generated using [GitBook](#).

Source Code: All the system source code will be stored in the [src](#) directory. Each step will contain links to the different related files.

Contributions: This course is open to contributions, feel free to signal errors with issues or directly correct the errors with pull-requests.

Questions: Feel free to ask any questions by adding issues. Please don't email me.

You can follow me on Twitter [@SamyPesse](#) or support me on [Flattr](#) or [Gittip](#).

What kind of OS are we building?

The goal is to build a very simple UNIX-based operating system in C++, not just a "proof-of-concept". The OS should be able to boot, start a userland shell, and be extensible.

```
2. vagrant@lucid32: /vagrant (bash)

Configure PIC
Loading Task Register
Loading Virtual Memory Management
Loading FileSystem Management
Loading syscalls interface
Loading system
env: create USER (liveuser)
env: create OS_NAME (devos)
env: create OS_VERSION (1)
env: create OS_DATE (Sep 4 2013)
env: create OS_TIME (17:54:47)
env: create OS_LICENCE (Apache License)
env: create COMPUTERNAME (test-pc)
env: create PROCESSOR_IDENTIFIER (x86)
env: create PROCESSOR_NAME (GenuineIntel)
env: create PATH (/bin/)
env: create SHELL (/bin/sh)
Loading modules
ext2: mount hda0 in boot
Loading binary modules
>load module: location=0x161000, size=55200

==== System is ready (Sep 4 2013 - 17:54:47) ====
hello world !
```

Chapter 1: Introduction to the x86 architecture and about our OS

What is the x86 architecture?

The term x86 denotes a family of backward compatible instruction set architectures based on the Intel 8086 CPU.

The x86 architecture is the most common instruction set architecture since its introduction in 1981 for the IBM PC. A large amount of software, including operating systems (OS's) such as DOS, Windows, Linux, BSD, Solaris and Mac OS X, function with x86-based hardware.

In this course we are not going to design an operating system for the x86-64 architecture but for x86-32, thanks to backward compatibility, our OS will be compatible with our newer PCs (but take caution if you want to test it on your real machine).

Our Operating System

The goal is to build a very simple UNIX-based operating system in C++, but the goal is not to just build a "proof-of-concept". The OS should be able to boot, start a userland shell and be extensible.

The OS will be built for the x86 architecture, running on 32 bits, and compatible with IBM PCs.

Specifications:

- Code in C++
- x86, 32 bit architecture
- Boot with Grub
- Kind of modular system for drivers

- Kind of UNIX style
- Multitasking
- ELF executable in userland
- Modules (accessible in userland using `/dev/...`) :
 - IDE disks
 - DOS partitions
 - Clock
 - EXT2 (read only)
 - Boch VBE
- Userland :
 - API Posix
 - LibC
 - "Can" run a shell or some executables like Lua, ...

Chapter 2: Setup the development environment

The first step is to setup a good and viable development environment. Using Vagrant and Virtualbox, you'll be able to compile and test your OS from all the OSs (Linux, Windows or Mac).

Install Vagrant

Vagrant is free and open-source software for creating and configuring virtual development environments. It can be considered a wrapper around VirtualBox.

Vagrant will help us create a clean virtual development environment on whatever system you are using. The first step is to download and install Vagrant for your system at <http://www.vagrantup.com/>.

Install Virtualbox

Oracle VM VirtualBox is a virtualization software package for x86 and AMD64/Intel64-based computers.

Vagrant needs Virtualbox to work, Download and install for your system at <https://www.virtualbox.org/wiki/Downloads>.

Start and test your development environment

Once Vagrant and Virtualbox are installed, you need to download the ubuntu lucid32 image for Vagrant:

```
vagrant box add lucid32 http://files.vagrantup.com/lucid32.box
```

Once the lucid32 image is ready, we need to define our development environment using a *Vagrantfile*, [create a file named *Vagrantfile*](#). This file defines what prerequisites our environment needs: nasm, make, build-essential, grub and qemu.

Start your box using:

```
vagrant up
```

You can now access your box by using ssh to connect to the virtual box using:

```
vagrant ssh
```

Put any file into the same directory of Vagrantfile, it will be available in the */vagrant* directory of VM (in this case, Ubuntu Lucid32):

```
cd /vagrant
```

Build and test our operating system

The file [**Makefile**](#) defines some basics rules for building the kernel, the user libc and some userland programs.

Build:

```
make all
```

Test our operating system with qemu:

```
make run
```

The documentation for qemu is available at [QEMU Emulator Documentation](#).

You can exit the emulator using: Ctrl-a.

Chapter 3: First boot with GRUB

How the boot works?

When an x86-based computer is turned on, it begins a complex path to get to the stage where control is transferred to our kernel's "main" routine (`kmain()`). For this course, we are only going to consider the BIOS boot method and not it's successor (UEFI).

The BIOS boot sequence is: RAM detection -> Hardware detection/Initialization -> Boot sequence.

The most important step for us is the "Boot sequence", where the BIOS is done with its initialization and tries to transfer control to the next stage of the bootloader process.

During the "Boot sequence", the BIOS will try to determine a "boot device" (e.g. floppy disk, hard-disk, CD, USB flash memory device or network). Our Operating System will initially boot from the hard-disk (but it will be possible to boot it from a CD or a USB flash memory device in future). A device is considered bootable if the bootsector contains the valid signature bytes `0x55` and `0xAA` at offsets 511 and 512 respectively (called the magic bytes of Master Boot Record (MBR), This signature is represented (in binary) as `0b1010101001010101`. The alternating bit pattern was thought to be a protection against certain failures (drive or controller). If this pattern is garbled or `0x00`, the device is not considered bootable)

BIOS physically searches for a boot device by loading the first 512 bytes from the bootsector of each device into physical memory, starting at the address `0x7C00` (1 KiB below the 32 KiB mark). When the valid signature bytes are detected, BIOS transfers control to the `0x7C00` memory address (via a jump instruction) in order to execute the bootsector code.

Throughout this process the CPU has been running in 16-bit Real Mode (the default state for x86 CPUs in order to maintain backwards compatibility). To

execute the 32-bit instructions within our kernel, a bootloader is required to switch the CPU into Protected Mode.

What is GRUB?

GNU GRUB (short for GNU GRand Unified Bootloader) is a boot loader package from the GNU Project. GRUB is the reference implementation of the Free Software Foundation's Multiboot Specification, which provides a user the choice to boot one of multiple operating systems installed on a computer or select a specific kernel configuration available on a particular operating system's partitions.

To make it simple, GRUB is the first thing booted by the machine (a bootloader) and will simplify the loading of our kernel stored on the hard-disk.

Why are we using GRUB?

- GRUB is very simple to use
- Make it very simple to load 32bits kernels without needs of 16bits code
- Multiboot with Linux, Windows and others
- Make it easy to load external modules in memory

How to use GRUB?

GRUB uses the Multiboot specification, the executable binary should be 32bits and must contain a special header (multiboot header) in its 8192 first bytes. Our kernel will be a ELF executable file ("Executable and Linkable Format", a common standard file format for executables in most UNIX system).

The first boot sequence of our kernel is written in Assembly: [start.asm](#) and we use a linker file to define our executable structure: [linker.ld](#).

This boot process also initializes some of our C++ runtime, it will be described in the next chapter.

Multiboot header structure:

```
struct multiboot_info {
    u32 flags;
    u32 low_mem;
    u32 high_mem;
    u32 boot_device;
    u32 cmdline;
    u32 mods_count;
    u32 mods_addr;
    struct {
        u32 num;
        u32 size;
        u32 addr;
        u32 shndx;
    } elf_sec;
    unsigned long mmap_length;
    unsigned long mmap_addr;
    unsigned long drives_length;
    unsigned long drives_addr;
    unsigned long config_table;
    unsigned long boot_loader_name;
    unsigned long apm_table;
    unsigned long vbe_control_info;
    unsigned long vbe_mode_info;
    unsigned long vbe_mode;
    unsigned long vbe_interface_seg;
    unsigned long vbe_interface_off;
    unsigned long vbe_interface_len;
};
```

You can use the command `mbchk kernel.elf` to validate your `kernel.elf` file against the multiboot standard. You can also use the command `nm -n kernel.elf` to validate the offset of the different objects in the ELF binary.

Create a disk image for our kernel and grub

The script [diskimage.sh](#) will generate a hard disk image that can be used by QEMU.

The first step is to create a hard-disk image (c.img) using qemu-img:

```
qemu-img create c.img 2M
```

We need now to partition the disk using fdisk:

```
fdisk ./c.img

# Switch to Expert commands
> x

# Change number of cylinders (1-1048576)
> c
> 4

# Change number of heads (1-256, default 16):
> h
> 16

# Change number of sectors/track (1-63, default 63)
> s
> 63

# Return to main menu
> r

# Add a new partition
> n

# Choose primary partition
> p
```

```
# Choose partition number
> 1

# Choose first cylinder (1-4, default 1)
> 1

# Choose last cylinder, +cylinders or +size{K,M,G} (1-4, default 4)
> 4

# Toggle bootable flag
> a

# Choose first partition for bootable flag
> 1

# Write table to disk and exit
> w
```

We need now to attach the created partition to the loop-device (which allows a file to be accessed like a block device) using losetup. The offset of the partition is passed as an argument and calculated using: **offset= start_sector * bytes_by_sector**.

Using `fdisk -l -u c.img`, you get: $63 * 512 = 32256$.

```
losetup -o 32256 /dev/loop1 ./c.img
```

We create a EXT2 filesystem on this new device using:

```
mke2fs /dev/loop1
```

We copy our files on a mounted disk:

```
mount /dev/loop1 /mnt/
cp -R bootdisk/* /mnt/
umount /mnt/
```

Install GRUB on the disk:

```
grub --device-map=/dev/null << EOF
device (hd0) ./c.img
geometry (hd0) 4 16 63
root (hd0,0)
setup (hd0)
quit
EOF
```

And finally we detach the loop device:

```
losetup -d /dev/loop1
```

See Also

- [GNU GRUB on Wikipedia](#)
- [Multiboot specification](#)

Chapter 4: Backbone of the OS and C++ runtime

C++ kernel run-time

A kernel can be programmed in C++, it is very similar to making a kernel in C, except that there are a few pitfalls you must take into account (runtime support, constructors, ...)

The compiler will assume that all the necessary C++ runtime support is available by default, but as we are not linking in libsupc++ into your C++ kernel, we need to add some basic functions that can be found in the [cxx.cc](#) file.

Caution: The operators `new` and `delete` cannot be used before virtual memory and pagination have been initialized.

Basic C/C++ functions

The kernel code can't use functions from the standard libraries so we need to add some basic functions for managing memory and strings:

```
void      itoa(char *buf, unsigned long int n, int base);

void *    memset(char *dst, char src, int n);
void *    memcpy(char *dst, char *src, int n);

int      strlen(char *s);
int      strcmp(const char *dst, char *src);
int      strcpy(char *dst, const char *src);
void      strcat(void *dest, const void *src);
char *    strncpy(char *destString, const char *sourceString, int maxLength);
int      strncmp( const char* s1, const char* s2, int c );
```

These functions are defined in [string.cc](#), [memory.cc](#), [itoa.cc](#)

C types

During the next step, we are going to use different types in our code, most of the types we are going to use unsigned types (all the bits are used to stored the integer, in signed types one bit is used to signal the sign):

```
typedef unsigned char      u8;
typedef unsigned short     u16;
typedef unsigned int       u32;
typedef unsigned long long u64;

typedef signed char        s8;
typedef signed short       s16;
typedef signed int         s32;
typedef signed long long   s64;
```

Compile our kernel

Compiling a kernel is not the same thing as compiling a linux executable, we can't use a standard library and should have no dependencies to the system.

Our [Makefile](#) will define the process to compile and link our kernel.

For x86 architecture, the followings arguments will be used for gcc/g++/ld:

```
# Linker
LD=ld
LDFLAG= -melf_i386 -static -L ./ -T ./arch/$(ARCH)/linker.ld

# C++ compiler
SC=g++
FLAG= $(INCDIR) -g -O2 -w -trigraphs -fno-builtin -fno-exceptions -fno-stack-
```

```
protector -O0 -m32 -fno-rtti -nostdlib -nodefaultlibs
```

```
# Assembly compiler
```

```
ASM=nasm
```

```
ASMFLAG=-f elf -o
```

Chapter 5: Base classes for managing x86 architecture

Now that we know how to compile our C++ kernel and boot the binary using GRUB, we can start to do some cool things in C/C++.

Printing to the screen console

We are going to use VGA default mode (03h) to display some text to the user. The screen can be directly accessed using the video memory at 0xB8000. The screen resolution is 80x25 and each character on the screen is defined by 2 bytes: one for the character code, and one for the style flag. This means that the total size of the video memory is 4000B (80B25B2B).

In the IO class ([io.cc](#)):

- **x,y**: define the cursor position on the screen
- **real_screen**: define the video memory pointer
- **putc(char c)**: print a unique character on the screen and manage cursor position
- **printf(char* s, ...)**: print a string

We add a method **putc** to the [IO Class](#) to put a character on the screen and update the (x,y) position.

```
/* put a byte on screen */
void Io::putc(char c){
    kattr = 0x07;
    unsigned char *video;
    video = (unsigned char *) (real_screen + 2 * x + 160 * y);
    // newline
    if (c == '\n') {
        x = 0;
```

```

        y++;
        // back space
    } else if (c == '\b') {
        if (x) {
            *(video + 1) = 0x0;
            x--;
        }
        // horizontal tab
    } else if (c == '\t') {
        x = x + 8 - (x % 8);
        // carriage return
    } else if (c == '\r') {
        x = 0;
    } else {
        *video = c;
        *(video + 1) = kattr;

        x++;
        if (x > 79) {
            x = 0;
            y++;
        }
    }
    if (y > 24)
        scrollup(y - 24);
}

```

We also add a useful and very known method: [printf](#)

```

/* put a string in screen */
void Io::print(const char *s, ...){
    va_list ap;

    char buf[16];
    int i, j, size, buflen, neg;

```

```

unsigned char c;
int ival;
unsigned int uival;

va_start(ap, s);

while ((c = *s++)) {
    size = 0;
    neg = 0;

    if (c == 0)
        break;
    else if (c == '%') {
        c = *s++;
        if (c >= '0' && c <= '9') {
            size = c - '0';
            c = *s++;
        }

        if (c == 'd') {
            ival = va_arg(ap, int);
            if (ival < 0) {
                uival = 0 - ival;
                neg++;
            } else
                uival = ival;
            itoa(buf, uival, 10);

            buflen = strlen(buf);
            if (buflen < size)
                for (i = size, j = buflen; i >= 0;
                     i--, j--)
                    buf[i] =
                        (j >=
                         0) ? buf[j] : '0';

            if (neg)

```

```

        print("-%s", buf);
    else
        print(buf);
    }

    else if (c == 'u') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 10);

        buflen = strlen(buf);
        if (buflen < size)
            for (i = size, j = buflen; i >= 0;
                 i--, j--)
                buf[i] =
                    (j >=
                     0) ? buf[j] : '0';

        print(buf);
    } else if (c == 'x' || c == 'X') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 16);

        buflen = strlen(buf);
        if (buflen < size)
            for (i = size, j = buflen; i >= 0;
                 i--, j--)
                buf[i] =
                    (j >=
                     0) ? buf[j] : '0';

        print("0x%s", buf);
    } else if (c == 'p') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 16);
        size = 8;

        buflen = strlen(buf);
        if (buflen < size)

```

```

        for (i = size, j = buflen; i >= 0;
             i--, j--)
            buf[i] =
                (j >=
                 0) ? buf[j] : '0';

        print("0x%s", buf);
    } else if (c == 's') {
        print((char *) va_arg(ap, int));
    }
} else
    putc(c);
}

return;
}

```

Assembly interface

A large number of instructions are available in Assembly but there is not equivalent in C (like cli, sti, in and out), so we need an interface to these instructions.

In C, we can include Assembly using the directive "asm()", gcc use gas to compile the assembly.

Caution: gas uses the AT&T syntax.

```

/* output byte */
void Io::outb(u32 ad, u8 v){
    asmv("outb %%al, %%dx" :: "d" (ad), "a" (v));;
}

/* output word */
void Io::outw(u32 ad, u16 v){
    asmv("outw %%ax, %%dx" :: "d" (ad), "a" (v));;
}

```

```
}

/* output word */

void Io::outl(u32 ad, u32 v){
    asmv("outl %%eax, %%dx" : : "d" (ad), "a" (v));
}

/* input byte */

u8 Io::inb(u32 ad){
    u8 _v;           \
    asmv("inb %%dx, %%al" : "=a" (_v) : "d" (ad)); \
    return _v;
}

/* input word */

u16 Io::inw(u32 ad){
    u16 _v;           \
    asmv("inw %%dx, %%ax" : "=a" (_v) : "d" (ad)); \
    return _v;
}

/* input word */

u32 Io::inl(u32 ad){
    u32 _v;           \
    asmv("inl %%dx, %%eax" : "=a" (_v) : "d" (ad)); \
    return _v;
}
```

Chapter 6: GDT

Thanks to GRUB, your kernel is no longer in real-mode, but already in protected mode, this mode allows us to use all the possibilities of the microprocessor such as virtual memory management, paging and safe multi-tasking.

What is the GDT?

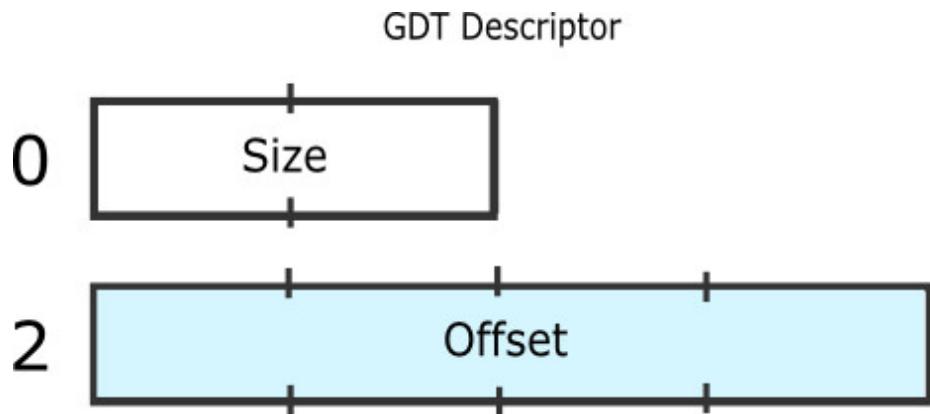
The GDT ("Global Descriptor Table") is a data structure used to define the different memory areas: the base address, the size and access privileges like execute and write. These memory areas are called "segments".

We are going to use the GDT to define different memory segments:

- "*code*": kernel code, used to stored the executable binary code
- "*data*": kernel data
- "*stack*": kernel stack, used to stored the call stack during kernel execution
- "*uode*": user code, used to stored the executable binary code for user program
- "*udata*": user program data
- "*ustack*": user stack, used to stored the call stack during execution in userland

How to load our GDT?

GRUB initializes a GDT but this GDT is does not correspond to our kernel. The GDT is loaded using the LGDT assembly instruction. It expects the location of a GDT description structure:



And the C structure:

```

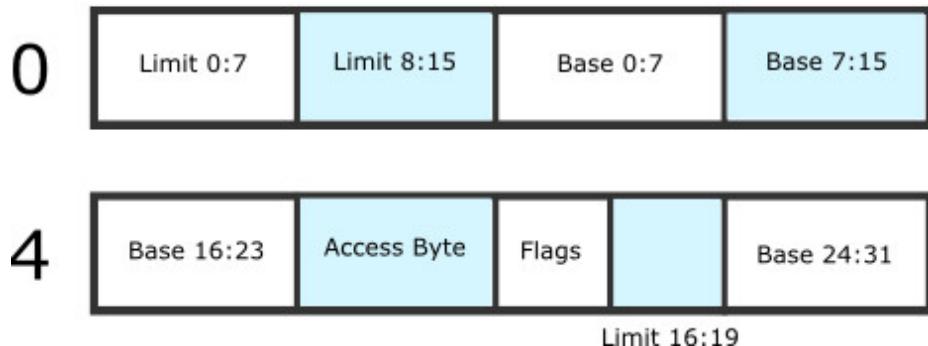
struct gdtr {
    u16 limite;
    u32 base;
} __attribute__ ((packed));
  
```

Caution: the directive `__attribute__ ((packed))` signal to gcc that the structure should use as little memory as possible. Without this directive, gcc include some bytes to optimize the memory alignment and the access during execution.

Now we need to define our GDT table and then load it using LGDT. The GDT table can be stored wherever we want in memory, its address should just be signaled to the process using the GDTR registry.

The GDT table is composed of segments with the following structure:

GDT Entry



And the C structure:

```
struct gdtdesc {  
    u16 lim0_15;  
    u16 base0_15;  
    u8 base16_23;  
    u8 acces;  
    u8 lim16_19:4;  
    u8 other:4;  
    u8 base24_31;  
} __attribute__ ((packed));
```

How to define our GDT table?

We need now to define our GDT in memory and finally load it using the GDTR registry.

We are going to store our GDT at the address:

```
#define GDTBASE 0x000000800
```

The function **init_gdt_desc** in [x86.cc](#) initialize a gdt segment descriptor.

```

void init_gdt_desc(u32 base, u32 limite, u8 acces, u8 other, struct gdtdesc *desc)
{
    desc->lim0_15 = (limite & 0xffff);
    desc->base0_15 = (base & 0xffff);
    desc->base16_23 = (base & 0xff0000) >> 16;
    desc->acces = acces;
    desc->lim16_19 = (limite & 0xf0000) >> 16;
    desc->other = (other & 0xf);
    desc->base24_31 = (base & 0xff000000) >> 24;
    return;
}

```

And the function **init_gdt** initialize the GDT, some parts of the below function will be explained later and are used for multitasking.

```

void init_gdt(void)
{
    default_tss.debug_flag = 0x00;
    default_tss.io_map = 0x00;
    default_tss.esp0 = 0x1FFF0;
    default_tss.ss0 = 0x18;

    /* initialize gdt segments */
    init_gdt_desc(0x0, 0x0, 0x0, 0x0, &kgdt[0]);
    init_gdt_desc(0x0, 0xFFFF, 0x9B, 0x0D, &kgdt[1]); /* code */
    init_gdt_desc(0x0, 0xFFFF, 0x93, 0x0D, &kgdt[2]); /* data */
    init_gdt_desc(0x0, 0x0, 0x97, 0x0D, &kgdt[3]); /* stack */

    init_gdt_desc(0x0, 0xFFFF, 0xFF, 0x0D, &kgdt[4]); /* ucode */
    init_gdt_desc(0x0, 0xFFFF, 0xF3, 0x0D, &kgdt[5]); /* udata */
    init_gdt_desc(0x0, 0x0, 0xF7, 0x0D, &kgdt[6]); /* ustack */

    init_gdt_desc((u32) & default_tss, 0x67, 0xE9, 0x00, &kgdt[7]); /* descripteur de tss */
}

```

```
/* initialize the gdtr structure */
kgdtr.limite = GDTSIZE * 8;
kgdtr.base = GDTBASE;

/* copy the gdtr to its memory area */
memcpy((char *) kgdtr.base, (char *) kgdt, kgdtr.limite);

/* load the gdtr registry */
asm("lgdtl (kgdtr)");

/* initiliaz the segments */
asm("    movw $0x10, %ax    \n \
     movw %ax, %ds    \n \
     movw %ax, %es    \n \
     movw %ax, %fs    \n \
     movw %ax, %gs    \n \
     ljmp $0x08, $next    \n \
next:        \n");
}
```

Chapter 7: IDT and interrupts

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.

There are 3 types of interrupts:

- **Hardware interrupts:** are sent to the processor from an external device (keyboard, mouse, hard disk, ...). Hardware interrupts were introduced as a way to reduce wasting the processor's valuable time in polling loops, waiting for external events.
- **Software interrupts:** are initiated voluntarily by the software. It's used to manage system calls.
- **Exceptions:** are used for errors or events occurring during program execution that are exceptional enough that they cannot be handled within the program itself (division by zero, page fault, ...)

The keyboard example:

When the user pressed a key on the keyboard, the keyboard controller will signal an interrupt to the Interrupt Controller. If the interrupt is not masked, the controller will signal the interrupt to the processor, the processor will execute a routine to manage the interrupt (key pressed or key released), this routine could, for example, get the pressed key from the keyboard controller and print the key to the screen. Once the character processing routine is completed, the interrupted job can be resumed.

What is the PIC?

The [PIC](#) (Programmable interrupt controller) is a device that is used to combine several sources of interrupt onto one or more CPU lines, while allowing priority levels to be assigned to its interrupt outputs. When the device has multiple interrupt outputs to assert, it asserts them in the order of their relative priority.

The best known PIC is the 8259A, each 8259A can handle 8 devices but most computers have two controllers: one master and one slave, this allows the computer to manage interrupts from 14 devices.

In this chapter, we will need to program this controller to initialize and mask interrupts.

What is the IDT?

The Interrupt Descriptor Table (IDT) is a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions.

Our kernel is going to use the IDT to define the different functions to be executed when an interrupt occurred.

Like the GDT, the IDT is loaded using the LIDTL assembly instruction. It expects the location of a IDT description structure:

```
struct idtr {  
    u16 limite;  
    u32 base;  
} __attribute__((packed));
```

The IDT table is composed of IDT segments with the following structure:

```
struct idtdesc {  
    u16 offset0_15;  
    u16 select;  
    u16 type;  
    u16 offset16_31;  
} __attribute__((packed));
```

Caution: the directive `__attribute__ ((packed))` signal to gcc that the structure should use as little memory as possible. Without this directive, gcc includes some bytes to optimize the memory alignment and the access during execution.

Now we need to define our IDT table and then load it using LIDTL. The IDT table can be stored wherever we want in memory, its address should just be signaled to the process using the IDTR registry.

Here is a table of common interrupts (Maskable hardware interrupt are called IRQ):

IRQ	Description
0	Programmable Interrupt Timer Interrupt
1	Keyboard Interrupt
2	Cascade (used internally by the two PICs. never raised)
3	COM2 (if enabled)
4	COM1 (if enabled)
5	LPT2 (if enabled)
6	Floppy Disk
7	LPT1
8	CMOS real-time clock (if enabled)
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

How to initialize the interrupts?

This is a simple method to define an IDT segment

```

void init_idt_desc(u16 select, u32 offset, u16 type, struct idtdesc *desc)
{
    desc->offset0_15 = (offset & 0xffff);
    desc->select = select;
    desc->type = type;
    desc->offset16_31 = (offset & 0xffff0000) >> 16;
    return;
}

```

And we can now initialize the interrupts:

```

#define IDTBASE    0x00000000
#define IDTSIZE 0xFF
idtr kidtr;

void init_idt(void)
{
    /* Init irq */
    int i;
    for (i = 0; i < IDTSIZE; i++)
        init_idt_desc(0x08, (u32)_asm_schedule, INTGATE, &kidt[i]); // */

    /* Vectors 0 -> 31 are for exceptions */
    init_idt_desc(0x08, (u32) _asm_exc_GP, INTGATE, &kidt[13]);      /* #GP */
    init_idt_desc(0x08, (u32) _asm_exc_PF, INTGATE, &kidt[14]);      /* #PF */

    init_idt_desc(0x08, (u32) _asm_schedule, INTGATE, &kidt[32]);
    init_idt_desc(0x08, (u32) _asm_int_1, INTGATE, &kidt[33]);

    init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt[48]);
    init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt[128]); //48

    kidtr.limite = IDTSIZE * 8;
    kidtr.base = IDTBASE;
}

```

```

/* Copy the IDT to the memory */
memcpy((char *) kidtr.base, (char *) kidt, kidtr.limite);

/* Load the IDTR registry */
asm("lidtl (kidtr)");

}

```

After initialization of our IDT, we need to activate interrupts by configuring the PIC. The following function will configure the two PICs by writing in their internal registries using the output ports of the processor `io.outb`. We configure the PICs using the ports:

- Master PIC: 0x20 and 0x21
- Slave PIC: 0xA0 and 0xA1

For a PIC, there are 2 types of registries:

- ICW (Initialization Command Word): reinit the controller
- OCW (Operation Control Word): configure the controller once initialized (used to mask/unmask the interrupts)

```

void init_pic(void)
{
    /* Initialization of ICW1 */
    io.outb(0x20, 0x11);
    io.outb(0xA0, 0x11);

    /* Initialization of ICW2 */
    io.outb(0x21, 0x20);    /* start vector = 32 */
    io.outb(0xA1, 0x70);    /* start vector = 96 */

    /* Initialization of ICW3 */
    io.outb(0x21, 0x04);
}

```

```

        io.outb(0xA1, 0x02);

        /* Initialization of ICW4 */
        io.outb(0x21, 0x01);
        io.outb(0xA1, 0x01);

        /* mask interrupts */
        io.outb(0x21, 0x0);
        io.outb(0xA1, 0x0);

    }

```

PIC ICW configurations details

The registries have to be configured in order.

ICW1 (port 0x20 / port 0xA0)

```

|0|0|0|1|x|0|x|x|
      | +--- with ICW4 (1) or without (0)
      | +---- one controller (1), or cascade (0)
      +----- triggering by level (level) (1) or by edge (edge) (0)

```

ICW2 (port 0x21 / port 0xA1)

```

|x|x|x|x|x|0|0|0|
      | | | | |
      +----- base address for interrupts vectors

```

ICW2 (port 0x21 / port 0xA1)

For the master:

```
|x|x|x|x|x|x|x|x|x|
| | | | | | |
+----- slave controller connected to the port yes (1), or no (0)
```

For the slave:

```
|0|0|0|0|0|x|x|x|  pour l'esclave
| | |
+----- Slave ID which is equal to the master port
```

ICW4 (port 0x21 / port 0xA1)

It is used to define in which mode the controller should work.

```
|0|0|0|x|x|x|x|1|
| | | +----- mode "automatic end of interrupt" AEOI (1)
| | +----- mode buffered slave (0) or master (1)
| +----- mode buffered (1)
+----- mode "fully nested" (1)
```

Why do idt segments offset our ASM functions?

You should have noticed that when I'm initializing our IDT segments, I'm using offsets to segment the code in Assembly. The different functions are defined in [x86int.asm](#) and are of the following scheme:

```
%macro  SAVE_REGS 0
    pushad
    push ds
    push es
    push fs
    push gs
```

```

push ebx
mov bx, 0x10
mov ds, bx
pop ebx
%endmacro

%macro RESTORE_REGS 0
pop gs
pop fs
pop es
pop ds
popad
%endmacro

%macro INTERRUPT 1
global _asm_int_%1
_asm_int_%1:
    SAVE_REGS
    push %1
    call isr_default_int
    pop eax    ;a enlever sinon
    mov al, 0x20
    out 0x20, al
    RESTORE_REGS
    iret
%endmacro

```

These macros will be used to define the interrupt segment that will prevent corruption of the different registries, it will be very useful for multitasking.

Chapter 8: Memory management: physical and virtual

In the chapter related to the GDT, we saw that using segmentation a physical memory address is calculated using a segment selector and an offset.

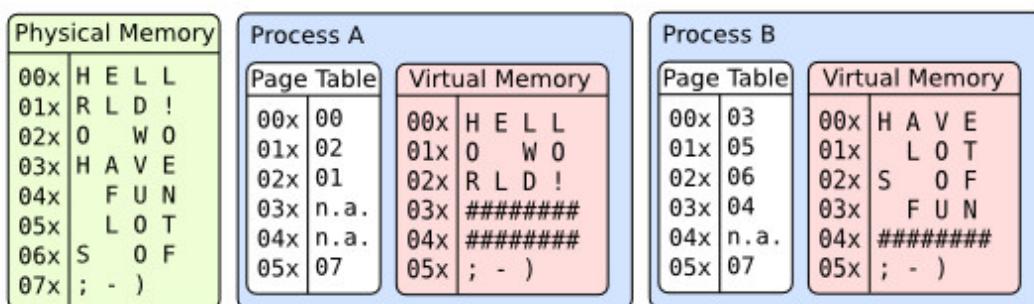
In this chapter, we are going to implement paging, paging will translate a linear address from segmentation into a physical address.

Why do we need paging?

Paging will allow our kernel to:

- use the hard-drive as a memory and not be limited by the machine ram memory limit
- to have a unique memory space for each process
- to allow and unallow memory space in a dynamic way

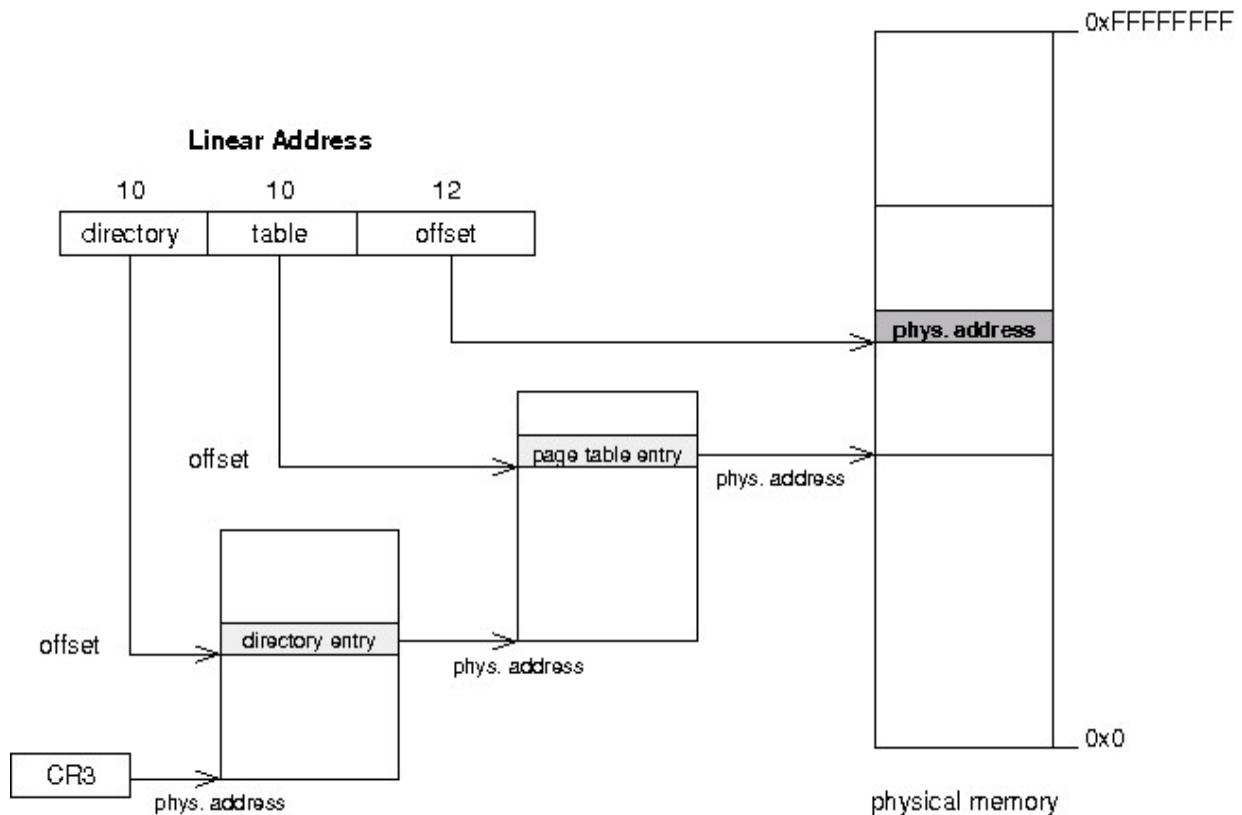
In a paged system, each process may execute in its own 4gb area of memory, without any chance of effecting any other process's memory, or the kernel's. It simplifies multitasking.



How does it work?

The translation of a linear address to a physical address is done in multiple steps:

1. The processor use the registry `CR3` to know the physical address of the pages directory.
2. The first 10 bits of the linear address represent an offset (between 0 and 1023), pointing to an entry in the pages directory. This entry contains the physical address of a pages table.
3. the next 10 bits of the linear address represent an offset, pointing to an entry in the pages table. This entry is pointing to a 4ko page.
4. The last 12 bits of the linear address represent an offset (between 0 and 4095), which indicates the position in the 4ko page.



Format for pages table and directory

The two types of entries (table and directory) look like the same. Only the field in gray will be used in our OS.

31	12	0
Page Table physical address	Avail. G P S 0 A P C W U D T S R P P W P	

31	12	0
Page physical address	Avail. G 0 D A P C W U R S W P	

- P: indicate if the page or table is in physical memory
- R/W: indicate if the page or table is accessible in writing (equals 1)
- U/S: equals 1 to allow access to non-preferred tasks
- A: indicate if the page or table was accessed
- D: (only for pages table) indicate if the page was written
- PS (only for pages directory) indicate the size of pages:
 - 0 = 4ko
 - 1 = 4mo

Note: Physical addresses in the pages directory or pages table are written using 20 bits because these addresses are aligned on 4ko, so the last 12 bits should be equal to 0.

- A pages directory or pages table used $1024 * 4 = 4096$ bytes = 4k
- A pages table can address $1024 * 4k = 4$ Mo
- A pages directory can address $1024 (1024 4k) = 4$ Go

How to enable pagination?

To enable pagination, we just need to set bit 31 of the CR0 register to 1:

```
asm("  mov %%cr0, %%eax; \
      or %1, %%eax;      \
      mov %%eax, %%cr0" \
      :: "i"(0x80000000));
```

But before, we need to initialize our pages directory with at least one pages table.

Table of Contents

[Introduction](#)

[Introduction about the x86 architecture and about our OS](#)

[Setup the development environment](#)

[First boot with GRUB](#)

[Backbone of the OS and C++ runtime](#)

[Base classes for managing x86 architecture](#)

[GDT](#)

[IDT and interrupts](#)

[Memory management: physical and virtual](#)



Your gateway to knowledge and culture. Accessible for everyone.



z-library.se singlelogin.re go-to-zlibrary.se single-login.ru



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>

**HOW TO
MAKE AN**

**OPERATING
SYSTEM**

FROM SCRATCH

Samy Pessé

Published
with GitBook

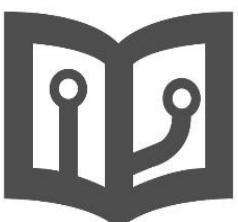


Table of Contents

Introduction	1.1
Introduction about the x86 architecture and about our OS	1.2
Setup the development environment	1.3
First boot with GRUB	1.4
Backbone of the OS and C++ runtime	1.5
Base classes for managing x86 architecture	1.6
GDT	1.7
IDT and interrupts	1.8
Theory: physical and virtual memory	1.9
Memory management: physical and virtual	1.10
Process management and multitasking	1.11
External program execution: ELF files	1.12
Userland and syscalls	1.13
Modular drivers	1.14
Some basics modules: console, keyboard	1.15
IDE Hard disks	1.16
DOS Partitions	1.17
EXT2 read-only filesystems	1.18
Standard C library (libC)	1.19
UNIX basic tools: sh, cat	1.20
Lua interpreter	1.21

How to Make a Computer Operating System

Online book about how to write a computer operating system in C/C++ from scratch.

Caution: This repository is a remake of my old course. It was written several years ago [as one of my first projects when I was in High School](#), I'm still refactoring some parts. The original course was in French and I'm not an English native. I'm going to continue and improve this course in my free-time.

Book: An online version is available at <http://samypesse.gitbooks.io/how-to-create-an-operating-system/> (PDF, Mobi and ePub). It was generated using [GitBook](#).

Source Code: All the system source code will be stored in the `src` directory. Each step will contain links to the different related files.

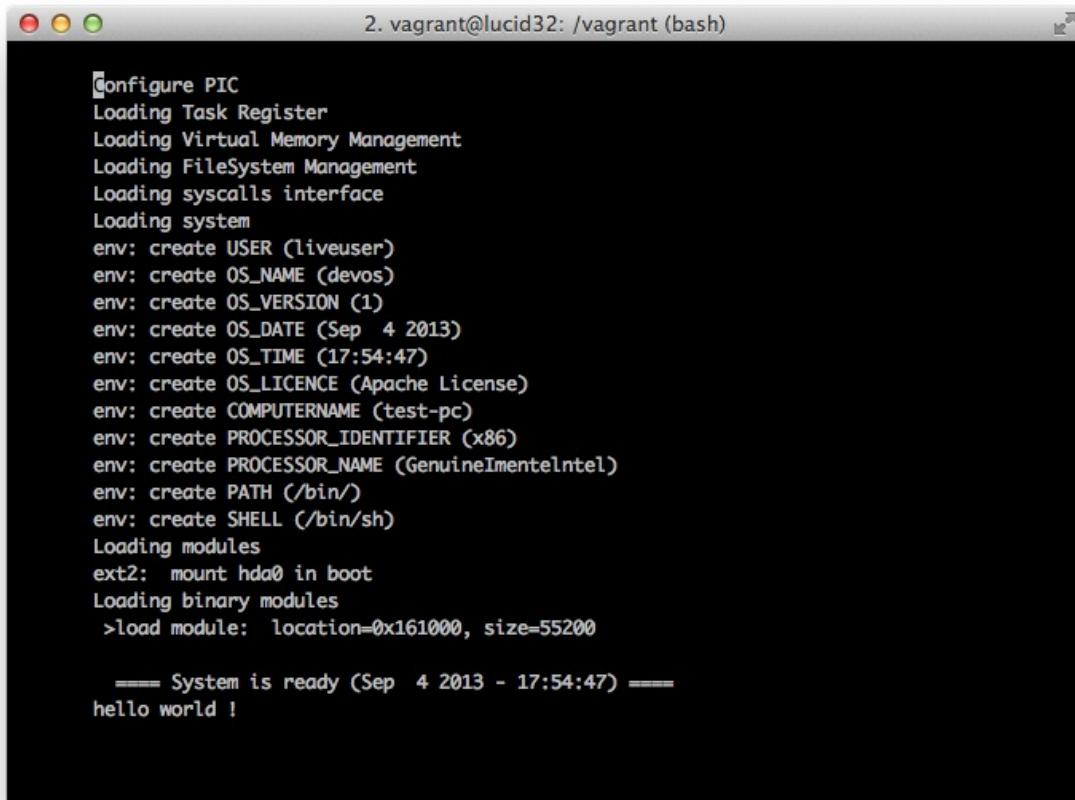
Contributions: This course is open to contributions, feel free to signal errors with issues or directly correct the errors with pull-requests.

Questions: Feel free to ask any questions by adding issues or commenting sections.

You can follow me on Twitter [@SamyPesse](#) or [GitHub](#).

What kind of OS are we building?

The goal is to build a very simple UNIX-based operating system in C++, not just a "proof-of-concept". The OS should be able to boot, start a userland shell, and be extensible.



2. vagrant@lucid32: /vagrant (bash)

```
Configure PIC
Loading Task Register
Loading Virtual Memory Management
Loading FileSystem Management
Loading syscalls interface
Loading system
env: create USER (liveuser)
env: create OS_NAME (devos)
env: create OS_VERSION (1)
env: create OS_DATE (Sep 4 2013)
env: create OS_TIME (17:54:47)
env: create OS_LICENCE (Apache License)
env: create COMPUTERNAME (test-pc)
env: create PROCESSOR_IDENTIFIER (x86)
env: create PROCESSOR_NAME (GenuineIntel)
env: create PATH (/bin/)
env: create SHELL (/bin/sh)
Loading modules
ext2: mount hda0 in boot
Loading binary modules
>load module: location=0x161000, size=55200

==== System is ready (Sep 4 2013 - 17:54:47) ====
hello world !
```

Chapter 1: Introduction to the x86 architecture and about our OS

What is the x86 architecture?

The term x86 denotes a family of backward compatible instruction set architectures based on the Intel 8086 CPU.

The x86 architecture is the most common instruction set architecture since its introduction in 1981 for the IBM PC. A large amount of software, including operating systems (OS's) such as DOS, Windows, Linux, BSD, Solaris and Mac OS X, function with x86-based hardware.

In this course we are not going to design an operating system for the x86-64 architecture but for x86-32, thanks to backward compatibility, our OS will be compatible with our newer PCs (but take caution if you want to test it on your real machine).

Our Operating System

The goal is to build a very simple UNIX-based operating system in C++, but the goal is not to just build a "proof-of-concept". The OS should be able to boot, start a userland shell and be extensible.

The OS will be built for the x86 architecture, running on 32 bits, and compatible with IBM PCs.

Specifications:

- Code in C++
- x86, 32 bit architecture
- Boot with Grub
- Kind of modular system for drivers
- Kind of UNIX style
- Multitasking
- ELF executable in userland
- Modules (accessible in userland using /dev/...) :
 - IDE disks
 - DOS partitions
 - Clock
 - EXT2 (read only)
 - Boch VBE
- Userland :

- API Posix
- LibC
- "Can" run a shell or some executables (e.g., lua)

Chapter 2: Setup the development environment

The first step is to setup a good and viable development environment. Using Vagrant and Virtualbox, you'll be able to compile and test your OS from all the OSs (Linux, Windows or Mac).

Install Vagrant

Vagrant is free and open-source software for creating and configuring virtual development environments. It can be considered a wrapper around VirtualBox.

Vagrant will help us create a clean virtual development environment on whatever system you are using. The first step is to download and install Vagrant for your system at <http://www.vagrantup.com/>.

Install Virtualbox

Oracle VM VirtualBox is a virtualization software package for x86 and AMD64/Intel64-based computers.

Vagrant needs Virtualbox to work, Download and install for your system at <https://www.virtualbox.org/wiki/Downloads>.

Start and test your development environment

Once Vagrant and Virtualbox are installed, you need to download the ubuntu lucid32 image for Vagrant:

```
vagrant box add lucid32 http://files.vagrantup.com/lucid32.box
```

Once the lucid32 image is ready, we need to define our development environment using a *Vagrantfile*, [create a file named Vagrantfile](#). This file defines what prerequisites our environment needs: nasm, make, build-essential, grub and qemu.

Start your box using:

```
vagrant up
```

You can now access your box by using ssh to connect to the virtual box using:

```
vagrant ssh
```

The directory containing the *Vagrantfile* will be mounted by default in the */vagrant* directory of the guest VM (in this case, Ubuntu Lucid32):

```
cd /vagrant
```

Build and test our operating system

The file **Makefile** defines some basics rules for building the kernel, the user libc and some userland programs.

Build:

```
make all
```

Test our operating system with qemu:

```
make run
```

The documentation for qemu is available at [QEMU Emulator Documentation](#).

You can exit the emulator using: Ctrl-a.

Chapter 3: First boot with GRUB

How the boot works?

When an x86-based computer is turned on, it begins a complex path to get to the stage where control is transferred to our kernel's "main" routine (`kmain()`). For this course, we are only going to consider the BIOS boot method and not its successor (UEFI).

The BIOS boot sequence is: RAM detection -> Hardware detection/Initialization -> Boot sequence.

The most important step for us is the "Boot sequence", where the BIOS is done with its initialization and tries to transfer control to the next stage of the bootloader process.

During the "Boot sequence", the BIOS will try to determine a "boot device" (e.g. floppy disk, hard-disk, CD, USB flash memory device or network). Our Operating System will initially boot from the hard-disk (but it will be possible to boot it from a CD or a USB flash memory device in future). A device is considered bootable if the bootsector contains the valid signature bytes `0x55` and `0xAA` at offsets 511 and 512 respectively (called the magic bytes of the Master Boot Record, also known as the MBR). This signature is represented (in binary) as `0b1010101001010101`. The alternating bit pattern was thought to be a protection against certain failures (drive or controller). If this pattern is garbled or `0x00`, the device is not considered bootable.

BIOS physically searches for a boot device by loading the first 512 bytes from the bootsector of each device into physical memory, starting at the address `0x7C00` (1 KiB below the 32 KiB mark). When the valid signature bytes are detected, BIOS transfers control to the `0x7C00` memory address (via a jump instruction) in order to execute the bootsector code.

Throughout this process the CPU has been running in 16-bit Real Mode, which is the default state for x86 CPUs in order to maintain backwards compatibility. To execute the 32-bit instructions within our kernel, a bootloader is required to switch the CPU into Protected Mode.

What is GRUB?

GNU GRUB (short for GNU GRand Unified Bootloader) is a boot loader package from the GNU Project. GRUB is the reference implementation of the Free Software Foundation's Multiboot Specification, which provides a user the choice to boot one of multiple operating systems installed on a computer or select a specific kernel configuration available on a particular operating system's partitions.

To make it simple, GRUB is the first thing booted by the machine (a boot-loader) and will simplify the loading of our kernel stored on the hard-disk.

Why are we using GRUB?

- GRUB is very simple to use
- Make it very simple to load 32bits kernels without needs of 16bits code
- Multiboot with Linux, Windows and others
- Make it easy to load external modules in memory

How to use GRUB?

GRUB uses the Multiboot specification, the executable binary should be 32bits and must contain a special header (multiboot header) in its 8192 first bytes. Our kernel will be a ELF executable file ("Executable and Linkable Format", a common standard file format for executables in most UNIX system).

The first boot sequence of our kernel is written in Assembly: [start.asm](#) and we use a linker file to define our executable structure: [linker.ld](#).

This boot process also initializes some of our C++ runtime, it will be described in the next chapter.

Multiboot header structure:

```
struct multiboot_info {
    u32 flags;
    u32 low_mem;
    u32 high_mem;
    u32 boot_device;
    u32 cmdline;
    u32 mods_count;
    u32 mods_addr;
    struct {
        u32 num;
        u32 size;
        u32 addr;
        u32 shndx;
    } elf_sec;
    unsigned long mmap_length;
    unsigned long mmap_addr;
    unsigned long drives_length;
    unsigned long drives_addr;
    unsigned long config_table;
    unsigned long boot_loader_name;
    unsigned long apm_table;
    unsigned long vbe_control_info;
    unsigned long vbe_mode_info;
    unsigned long vbe_mode;
    unsigned long vbe_interface_seg;
    unsigned long vbe_interface_off;
    unsigned long vbe_interface_len;
};

};
```

You can use the command `mbchk kernel.elf` to validate your kernel.elf file against the multiboot standard. You can also use the command `nm -n kernel.elf` to validate the offset of the different objects in the ELF binary.

Create a disk image for our kernel and grub

The script [diskimage.sh](#) will generate a hard disk image that can be used by QEMU.

The first step is to create a hard-disk image (c.img) using qemu-img:

```
qemu-img create c.img 2M
```

We need now to partition the disk using fdisk:

```
fdisk ./c.img

# Switch to Expert commands
> x

# Change number of cylinders (1-1048576)
> c
> 4

# Change number of heads (1-256, default 16):
> h
> 16

# Change number of sectors/track (1-63, default 63)
> s
> 63

# Return to main menu
> r

# Add a new partition
> n

# Choose primary partition
> p

# Choose partition number
> 1

# Choose first sector (1-4, default 1)
> 1

# Choose last sector, +cylinders or +size{K,M,G} (1-4, default 4)
> 4

# Toggle bootable flag
> a

# Choose first partition for bootable flag
> 1

# Write table to disk and exit
> w
```

We need now to attach the created partition to the loop-device using losetup. This allows a file to be accessed like a block device. The offset of the partition is passed as an argument and calculated using: **offset= start_sector * bytes_by_sector**.

Using `fdisk -l -u c.img`, you get: $63 * 512 = 32256$.

```
losetup -o 32256 /dev/loop1 ./c.img
```

We create a EXT2 filesystem on this new device using:

```
mke2fs /dev/loop1
```

We copy our files on a mounted disk:

```
mount /dev/loop1 /mnt/
cp -R bootdisk/* /mnt/
umount /mnt/
```

Install GRUB on the disk:

```
grub --device-map=/dev/null << EOF
device (hd0) ./c.img
geometry (hd0) 4 16 63
root (hd0,0)
setup (hd0)
quit
EOF
```

And finally we detach the loop device:

```
losetup -d /dev/loop1
```

See Also

- [GNU GRUB on Wikipedia](#)
- [Multiboot specification](#)

Chapter 4: Backbone of the OS and C++ runtime

C++ kernel run-time

A kernel can be written in C++ just as it can be in C, with the exception of a few pitfalls that come with using C++ (runtime support, constructors, etc).

The compiler will assume that all the necessary C++ runtime support is available by default, but as we are not linking `libsupc++` into your C++ kernel, we need to add some basic functions that can be found in the `cxx.cc` file.

Caution: The operators `new` and `delete` cannot be used before virtual memory and pagination have been initialized.

Basic C/C++ functions

The kernel code can't use functions from the standard libraries so we need to add some basic functions for managing memory and strings:

```
void      itoa(char *buf, unsigned long int n, int base);

void *    memset(char *dst, char src, int n);
void *    memcpy(char *dst, char *src, int n);

int      strlen(char *s);
int      strcmp(const char *dst, char *src);
int      strcpy(char *dst, const char *src);
void      strcat(void *dest, const void *src);
char *    strncpy(char *destString, const char *sourceString, int maxLength);
int      strncmp( const char* s1, const char* s2, int c );
```

These functions are defined in [string.cc](#), [memory.cc](#), [itoa.cc](#)

C types

In the next step, we're going to define different types we're going to use in our code. Most of our variable types are going to be unsigned. This means that all the bits are used to store the integer. Signed variables use their first bit to indicate their sign.

```
typedef unsigned char      u8;
typedef unsigned short     u16;
typedef unsigned int       u32;
typedef unsigned long long u64;

typedef signed char       s8;
typedef signed short      s16;
typedef signed int        s32;
typedef signed long long  s64;
```

Compile our kernel

Compiling a kernel is not the same thing as compiling a linux executable, we can't use a standard library and should have no dependencies to the system.

Our [Makefile](#) will define the process to compile and link our kernel.

For x86 architecture, the followings arguments will be used for gcc/g++/ld:

```
# Linker
LD=ld
LDFLAG= -melf_i386 -static -L ./ -T ./arch/$(ARCH)/linker.ld

# C++ compiler
SC=g++
FLAG= $(INCDIR) -g -O2 -w -trigraphs -fno-builtin -fno-exceptions -fno-stack-protecto
r -O0 -m32 -fno-rtti -nostdlib -nodefaultlibs

# Assembly compiler
ASM=nasm
ASMFLAG=-f elf -o
```

Chapter 5: Base classes for managing x86 architecture

Now that we know how to compile our C++ kernel and boot the binary using GRUB, we can start to do some cool things in C/C++.

Printing to the screen console

We are going to use VGA default mode (03h) to display some text to the user. The screen can be directly accessed using the video memory at 0xB8000. The screen resolution is 80x25 and each character on the screen is defined by 2 bytes: one for the character code, and one for the style flag. This means that the total size of the video memory is 4000B (80B25B2B).

In the IO class ([io.cc](#)):

- **x,y**: define the cursor position on the screen
- **real_screen**: define the video memory pointer
- **putc(char c)**: print a unique character on the screen and manage cursor position
- **printf(char* s, ...)**: print a string

We add a method **putc** to the [IO Class](#) to put a character on the screen and update the (x,y) position.

```

/* put a byte on screen */
void Io::putc(char c){
    kattr = 0x07;
    unsigned char *video;
    video = (unsigned char *) (real_screen+ 2 * x + 160 * y);
    // newline
    if (c == '\n') {
        x = 0;
        y++;
    // back space
    } else if (c == '\b') {
        if (x) {
            *(video + 1) = 0x0;
            x--;
        }
    // horizontal tab
    } else if (c == '\t') {
        x = x + 8 - (x % 8);
    // carriage return
    } else if (c == '\r') {
        x = 0;
    } else {
        *video = c;
        *(video + 1) = kattr;

        x++;
        if (x > 79) {
            x = 0;
            y++;
        }
    }
    if (y > 24)
        scrollup(y - 24);
}

```

We also add a useful and very known method: [printf](#)

```

/* put a string in screen */
void Io::print(const char *s, ...){
    va_list ap;

    char buf[16];
    int i, j, size, buflen, neg;

    unsigned char c;
    int ival;
    unsigned int uival;

    va_start(ap, s);

    while ((c = *s++)) {

```

```

size = 0;
neg = 0;

if (c == 0)
    break;
else if (c == '%') {
    c = *s++;
    if (c >= '0' && c <= '9') {
        size = c - '0';
        c = *s++;
    }

    if (c == 'd') {
        ival = va_arg(ap, int);
        if (ival < 0) {
            uival = 0 - ival;
            neg++;
        } else
            uival = ival;
        itoa(buf, uival, 10);

        buflen = strlen(buf);
        if (buflen < size)
            for (i = size, j = buflen; i >= 0;
                 i--, j--)
                buf[i] =
                    (j >=
                     0) ? buf[j] : '0';

        if (neg)
            print("-%s", buf);
        else
            print(buf);
    }
    else if (c == 'u') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 10);

        buflen = strlen(buf);
        if (buflen < size)
            for (i = size, j = buflen; i >= 0;
                 i--, j--)
                buf[i] =
                    (j >=
                     0) ? buf[j] : '0';

        print(buf);
    } else if (c == 'x' || c == 'X') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 16);

        buflen = strlen(buf);
        if (buflen < size)
    }
}

```

```

        for (i = size, j = buflen; i >= 0;
             i--, j--)
            buf[i] =
                (j >=
                 0) ? buf[j] : '0';

        print("0x%s", buf);
    } else if (c == 'p') {
        uival = va_arg(ap, int);
        itoa(buf, uival, 16);
        size = 8;

        buflen = strlen(buf);
        if (buflen < size)
            for (i = size, j = buflen; i >= 0;
                 i--, j--)
                buf[i] =
                    (j >=
                     0) ? buf[j] : '0';

        print("0x%s", buf);
    } else if (c == 's') {
        print((char *) va_arg(ap, int));
    }
} else
    putc(c);
}

return;
}

```

Assembly interface

A large number of instructions are available in Assembly but there is not equivalent in C (like cli, sti, in and out), so we need an interface to these instructions.

In C, we can include Assembly using the directive "asm()", gcc use gas to compile the assembly.

Caution: gas uses the AT&T syntax.

```
/* output byte */
void Io::outb(u32 ad, u8 v){
    asmv("outb %%al, %%dx" :: "d" (ad), "a" (v));
}

/* output word */
void Io::outw(u32 ad, u16 v){
    asmv("outw %%ax, %%dx" :: "d" (ad), "a" (v));
}

/* output word */
void Io::outl(u32 ad, u32 v){
    asmv("outl %%eax, %%dx" :: "d" (ad), "a" (v));
}

/* input byte */
u8 Io::inb(u32 ad){
    u8 _v;           \
    asmv("inb %%dx, %%al" : "=a" (_v) : "d" (ad)); \
    return _v;
}

/* input word */
u16 Io::inw(u32 ad){
    u16 _v;           \
    asmv("inw %%dx, %%ax" : "=a" (_v) : "d" (ad)); \
    return _v;
}

/* input word */
u32 Io::inl(u32 ad){
    u32 _v;           \
    asmv("inl %%dx, %%eax" : "=a" (_v) : "d" (ad)); \
    return _v;
}
```

Chapter 6: GDT

Thanks to GRUB, your kernel is no longer in real-mode, but already in [protected mode](#), this mode allows us to use all the possibilities of the microprocessor such as virtual memory management, paging and safe multi-tasking.

What is the GDT?

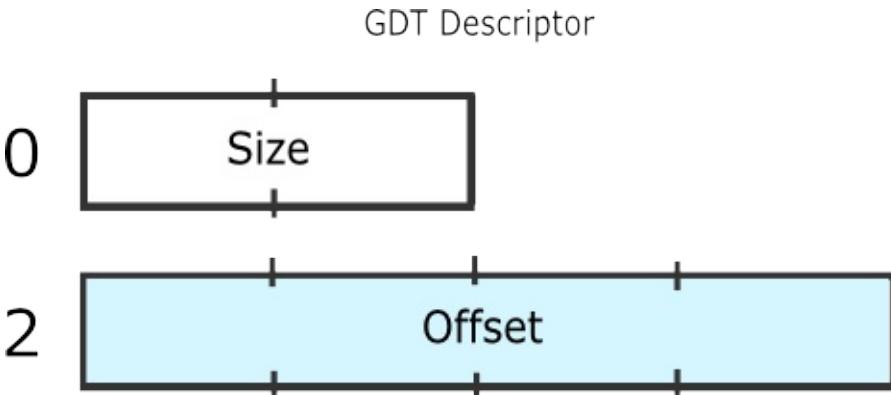
The [GDT](#) ("Global Descriptor Table") is a data structure used to define the different memory areas: the base address, the size and access privileges like execute and write. These memory areas are called "segments".

We are going to use the GDT to define different memory segments:

- "code": kernel code, used to stored the executable binary code
- "data": kernel data
- "stack": kernel stack, used to stored the call stack during kernel execution
- "ucode": user code, used to stored the executable binary code for user program
- "udata": user program data
- "ustack": user stack, used to stored the call stack during execution in userland

How to load our GDT?

GRUB initializes a GDT but this GDT is does not correspond to our kernel. The GDT is loaded using the LGDT assembly instruction. It expects the location of a GDT description structure:



And the C structure:

```
struct gdtr {
    u16 limite;
    u32 base;
} __attribute__ ((packed));
```

Caution: the directive `__attribute__ ((packed))` signal to gcc that the structure should use as little memory as possible. Without this directive, gcc include some bytes to optimize the memory alignment and the access during execution.

Now we need to define our GDT table and then load it using LGDT. The GDT table can be stored wherever we want in memory, its address should just be signaled to the process using the GDTR registry.

The GDT table is composed of segments with the following structure:

GDT Entry



And the C structure:

```
struct gdtdesc {
    u16 lim0_15;
    u16 base0_15;
    u8 base16_23;
    u8 acces;
    u8 lim16_19:4;
    u8 other:4;
    u8 base24_31;
} __attribute__ ((packed));
```

How to define our GDT table?

We need now to define our GDT in memory and finally load it using the GDTR registry.

We are going to store our GDT at the address:

```
#define GDTBASE 0x000000800
```

The function **init_gdt_desc** in [x86.cc](#) initialize a gdt segment descriptor.

```
void init_gdt_desc(u32 base, u32 limite, u8 acces, u8 other, struct gdtdesc *desc)
{
    desc->lim0_15 = (limite & 0xffff);
    desc->base0_15 = (base & 0xffff);
    desc->base16_23 = (base & 0xff0000) >> 16;
    desc->acces = acces;
    desc->lim16_19 = (limite & 0xf0000) >> 16;
    desc->other = (other & 0xf);
    desc->base24_31 = (base & 0xff000000) >> 24;
    return;
}
```

And the function **init_gdt** initialize the GDT, some parts of the below function will be explained later and are used for multitasking.

```

void init_gdt(void)
{
    default_tss.debug_flag = 0x00;
    default_tss.io_map = 0x00;
    default_tss.esp0 = 0x1FFF0;
    default_tss.ss0 = 0x18;

    /* initialize gdt segments */
    init_gdt_desc(0x0, 0x0, 0x0, 0x0, &kgdt[0]);
    init_gdt_desc(0x0, 0xFFFFF, 0x9B, 0x0D, &kgdt[1]);      /* code */
    init_gdt_desc(0x0, 0xFFFFF, 0x93, 0x0D, &kgdt[2]);      /* data */
    init_gdt_desc(0x0, 0x0, 0x97, 0x0D, &kgdt[3]);        /* stack */

    init_gdt_desc(0x0, 0xFFFFF, 0xFF, 0x0D, &kgdt[4]);      /* ucode */
    init_gdt_desc(0x0, 0xFFFFF, 0xF3, 0x0D, &kgdt[5]);      /* udata */
    init_gdt_desc(0x0, 0x0, 0xF7, 0x0D, &kgdt[6]);        /* ustack */

    init_gdt_desc((u32) & default_tss, 0x67, 0xE9, 0x00, &kgdt[7]); /* descripteur
de tss */

    /* initialize the gdtr structure */
    kgdtr.limite = GDTSIZE * 8;
    kgdtr.base = GDTBASE;

    /* copy the gdtr to its memory area */
    memcpy((char *) kgdtr.base, (char *) kgdt, kgdtr.limite);

    /* load the gdtr registry */
    asm("lgdtl (kgdtr)");

    /* initiliaz the segments */
    asm("    movw $0x10, %ax    \n \
        movw %ax, %ds    \n \
        movw %ax, %es    \n \
        movw %ax, %fs    \n \
        movw %ax, %gs    \n \
        ljmp $0x08, $next    \n \
        next:        \n");
}

```

Chapter 7: IDT and interrupts

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.

There are 3 types of interrupts:

- **Hardware interrupts:** are sent to the processor from an external device (keyboard, mouse, hard disk, ...). Hardware interrupts were introduced as a way to reduce wasting the processor's valuable time in polling loops, waiting for external events.
- **Software interrupts:** are initiated voluntarily by the software. It's used to manage system calls.
- **Exceptions:** are used for errors or events occurring during program execution that are exceptional enough that they cannot be handled within the program itself (division by zero, page fault, ...)

The keyboard example:

When the user pressed a key on the keyboard, the keyboard controller will signal an interrupt to the Interrupt Controller. If the interrupt is not masked, the controller will signal the interrupt to the processor, the processor will execute a routine to manage the interrupt (key pressed or key released), this routine could, for example, get the pressed key from the keyboard controller and print the key to the screen. Once the character processing routine is completed, the interrupted job can be resumed.

What is the PIC?

The [PIC](#) (Programmable interrupt controller) is a device that is used to combine several sources of interrupt onto one or more CPU lines, while allowing priority levels to be assigned to its interrupt outputs. When the device has multiple interrupt outputs to assert, it asserts them in the order of their relative priority.

The best known PIC is the 8259A, each 8259A can handle 8 devices but most computers have two controllers: one master and one slave, this allows the computer to manage interrupts from 14 devices.

In this chapter, we will need to program this controller to initialize and mask interrupts.

What is the IDT?

The Interrupt Descriptor Table (IDT) is a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions.

Our kernel is going to use the IDT to define the different functions to be executed when an interrupt occurred.

Like the GDT, the IDT is loaded using the LIDTL assembly instruction. It expects the location of a IDT description structure:

```
struct idtr {  
    u16 limite;  
    u32 base;  
} __attribute__ ((packed));
```

The IDT table is composed of IDT segments with the following structure:

```
struct idtdesc {  
    u16 offset0_15;  
    u16 select;  
    u16 type;  
    u16 offset16_31;  
} __attribute__ ((packed));
```

Caution: the directive `__attribute__ ((packed))` signal to gcc that the structure should use as little memory as possible. Without this directive, gcc includes some bytes to optimize the memory alignment and the access during execution.

Now we need to define our IDT table and then load it using LIDTL. The IDT table can be stored wherever we want in memory, its address should just be signaled to the process using the IDTR registry.

Here is a table of common interrupts (Maskable hardware interrupt are called IRQ):

IRQ	Description
0	Programmable Interrupt Timer Interrupt
1	Keyboard Interrupt
2	Cascade (used internally by the two PICs. never raised)
3	COM2 (if enabled)
4	COM1 (if enabled)
5	LPT2 (if enabled)
6	Floppy Disk
7	LPT1
8	CMOS real-time clock (if enabled)
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

How to initialize the interrupts?

This is a simple method to define an IDT segment

```
void init_idt_desc(u16 select, u32 offset, u16 type, struct idtdesc *desc)
{
    desc->offset0_15 = (offset & 0xffff);
    desc->select = select;
    desc->type = type;
    desc->offset16_31 = (offset & 0xffff0000) >> 16;
    return;
}
```

And we can now initialize the interrupts:

```
#define IDTBASE 0x0000000000
#define IDTSIZE 0xFF
idtr kidtr;
```

```

void init_idt(void)
{
    /* Init irq */
    int i;
    for (i = 0; i < IDTSIZE; i++)
        init_idt_desc(0x08, (u32)_asm_schedule, INTGATE, &kidt[i]); //

    /* Vectors 0 -> 31 are for exceptions */
    init_idt_desc(0x08, (u32) _asm_exc_GP, INTGATE, &kidt[13]);           /* #GP */
    init_idt_desc(0x08, (u32) _asm_exc_PF, INTGATE, &kidt[14]);           /* #PF */

    init_idt_desc(0x08, (u32) _asm_schedule, INTGATE, &kidt[32]);
    init_idt_desc(0x08, (u32) _asm_int_1, INTGATE, &kidt[33]);

    init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt[48]);
    init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt[128]); //48

    kidtr.limite = IDTSIZE * 8;
    kidtr.base = IDTBASE;

    /* Copy the IDT to the memory */
    memcpy((char *) kidtr.base, (char *) kidt, kidtr.limite);

    /* Load the IDTR registry */
    asm("lidtl (kidtr)");
}

```

After initialization of our IDT, we need to activate interrupts by configuring the PIC. The following function will configure the two PICs by writing in their internal registries using the output ports of the processor `io.outb`. We configure the PICs using the ports:

- Master PIC: 0x20 and 0x21
- Slave PIC: 0xA0 and 0xA1

For a PIC, there are 2 types of registries:

- ICW (Initialization Command Word): reinit the controller
- OCW (Operation Control Word): configure the controller once initialized (used to mask/unmask the interrupts)

```

void init_pic(void)
{
    /* Initialization of ICW1 */
    io.outb(0x20, 0x11);
    io.outb(0xA0, 0x11);

    /* Initialization of ICW2 */
    io.outb(0x21, 0x20);      /* start vector = 32 */
    io.outb(0xA1, 0x70);      /* start vector = 96 */

    /* Initialization of ICW3 */
    io.outb(0x21, 0x04);
    io.outb(0xA1, 0x02);

    /* Initialization of ICW4 */
    io.outb(0x21, 0x01);
    io.outb(0xA1, 0x01);

    /* mask interrupts */
    io.outb(0x21, 0x0);
    io.outb(0xA1, 0x0);
}

```

PIC ICW configurations details

The registries have to be configured in order.

ICW1 (port 0x20 / port 0xA0)

```

|0|0|0|1|x|0|x|x|
  |   | +--- with ICW4 (1) or without (0)
  |   +---- one controller (1), or cascade (0)
  +----- triggering by level (level) (1) or by edge (edge) (0)

```

ICW2 (port 0x21 / port 0xA1)

```

|x|x|x|x|x|0|0|0|
  | | | | |
  +----- base address for interrupts vectors

```

ICW2 (port 0x21 / port 0xA1)

For the master:

```
|x|x|x|x|x|x|x|x|x|  
| | | | | | | |  
+----- slave controller connected to the port yes (1), or no (0)
```

For the slave:

```
|0|0|0|0|0|x|x|x| pour l'esclave  
| | |  
+----- Slave ID which is equal to the master port
```

ICW4 (port 0x21 / port 0xA1)

It is used to define in which mode the controller should work.

```
|0|0|0|x|x|x|x|1|  
| | | +----- mode "automatic end of interrupt" AEOI (1)  
| | +----- mode buffered slave (0) or master (1)  
| +----- mode buffered (1)  
+----- mode "fully nested" (1)
```

Why do idt segments offset our ASM functions?

You should have noticed that when I'm initializing our IDT segments, I'm using offsets to segment the code in Assembly. The different functions are defined in [x86int.asm](#) and are of the following scheme:

```
%macro SAVE_REGS 0
    pushad
    push ds
    push es
    push fs
    push gs
    push ebx
    mov bx, 0x10
    mov ds, bx
    pop ebx
%endmacro

%macro RESTORE_REGS 0
    pop gs
    pop fs
    pop es
    pop ds
    popad
%endmacro

%macro INTERRUPT 1
    global _asm_int_%1
    _asm_int_%1:
        SAVE_REGS
        push %1
        call isr_default_int
        pop eax    ;;a enlever sinon
        mov al, 0x20
        out 0x20, al
        RESTORE_REGS
        iret
%endmacro
```

These macros will be used to define the interrupt segment that will prevent corruption of the different registries, it will be very useful for multitasking.

Chapter 8: Theory: physical and virtual memory

In the chapter related to the GDT, we saw that using segmentation a physical memory address is calculated using a segment selector and an offset.

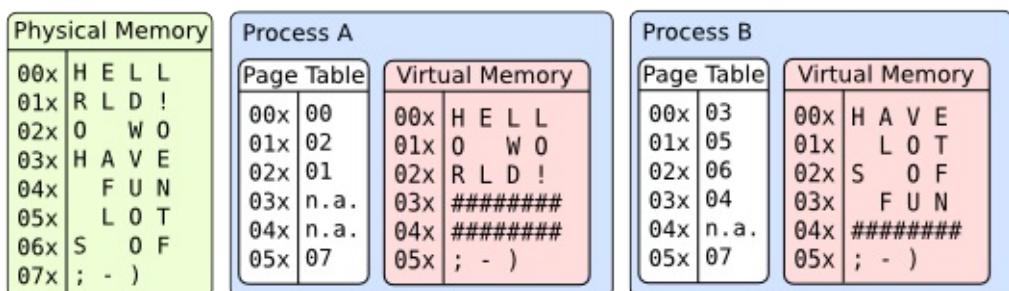
In this chapter, we are going to implement paging, paging will translate a linear address from segmentation into a physical address.

Why do we need paging?

Paging will allow our kernel to:

- use the hard-drive as a memory and not be limited by the machine ram memory limit
- to have a unique memory space for each process
- to allow and unallow memory space in a dynamic way

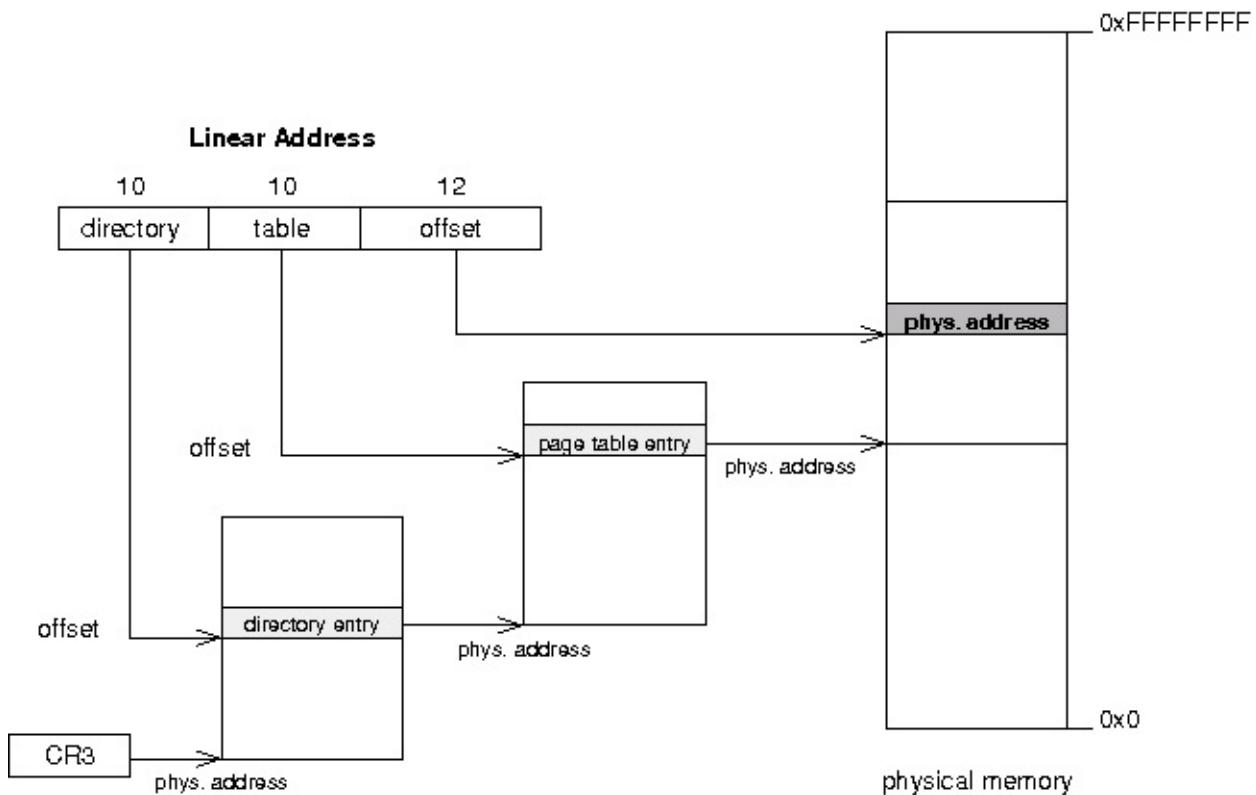
In a paged system, each process may execute in its own 4gb area of memory, without any chance of effecting any other process's memory, or the kernel's. It simplifies multitasking.



How does it work?

The translation of a linear address to a physical address is done in multiple steps:

1. The processor use the registry `CR3` to know the physical address of the pages directory.
2. The first 10 bits of the linear address represent an offset (between 0 and 1023), pointing to an entry in the pages directory. This entry contains the physical address of a pages table.
3. the next 10 bits of the linear address represent an offset, pointing to an entry in the pages table. This entry is pointing to a 4ko page.
4. The last 12 bits of the linear address represent an offset (between 0 and 4095), which indicates the position in the 4ko page.



Format for pages table and directory

The two types of entries (table and directory) look like the same. Only the field in gray will be used in our OS.

31	12	0
Page Table physical address	Avail. G P S 0 A C D P P U T S R W P	
Page physical address	Avail. G 0 D A C D P P U T S R W P	

- **P** : indicate if the page or table is in physical memory
- **R/W** : indicate if the page or table is accessible in writing (equals 1)
- **U/S** : equals 1 to allow access to non-preferred tasks
- **A** : indicate if the page or table was accessed
- **D** : (only for pages table) indicate if the page was written
- **PS** (only for pages directory) indicate the size of pages:
 - 0 = 4kb
 - 1 = 4mb

Note: Physical addresses in the pages directory or pages table are written using 20 bits because these addresses are aligned on 4kb, so the last 12 bits should be equal to 0.

- A pages directory or pages table used $1024 \times 4 = 4096$ bytes = 4k

- A pages table can address $1024 * 4k = 4 \text{ Mb}$
- A pages directory can address $1024 (1024 \text{ Mb}) = 4 \text{ Gb}$

How to enable pagination?

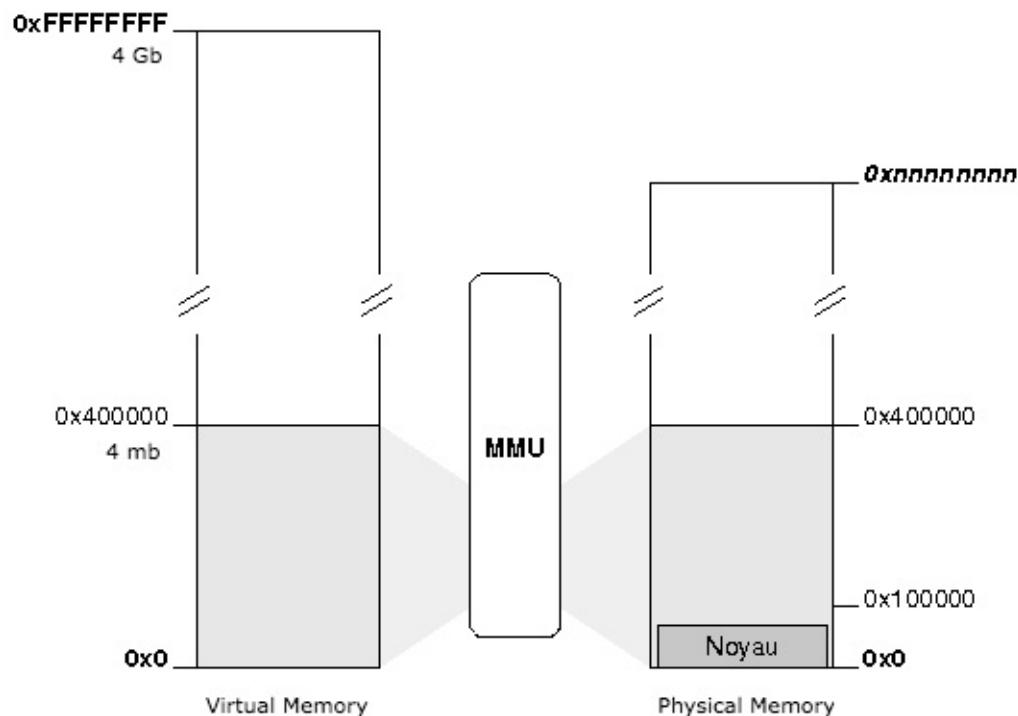
To enable pagination, we just need to set bit 31 of the `CR0` registry to 1:

```
asm("  mov %%cr0, %%eax; \
      or %1, %%eax;      \
      mov %%eax, %%cr0" \
      :: "i"(0x80000000));
```

But before, we need to initialize our pages directory with at least one pages table.

Identity Mapping

With the identity mapping model, the page will apply only to the kernel as the first 4 MB of virtual memory coincide with the first 4 MB of physical memory:



This model is simple: the first virtual memory page coincide to the first page in physical memory, the second page coincide to the second page on physical memory and so on ...

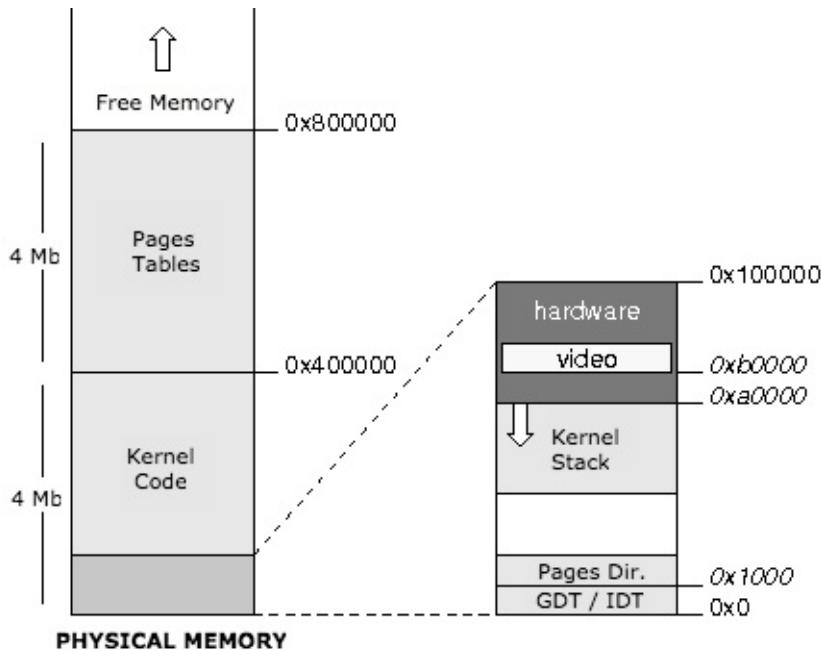
Memory management: physical and virtual

The kernel knows the size of the physical memory available thanks to [GRUB](#).

In our implementation, the first 8 megabytes of physical memory will be reserved for use by the kernel and will contain:

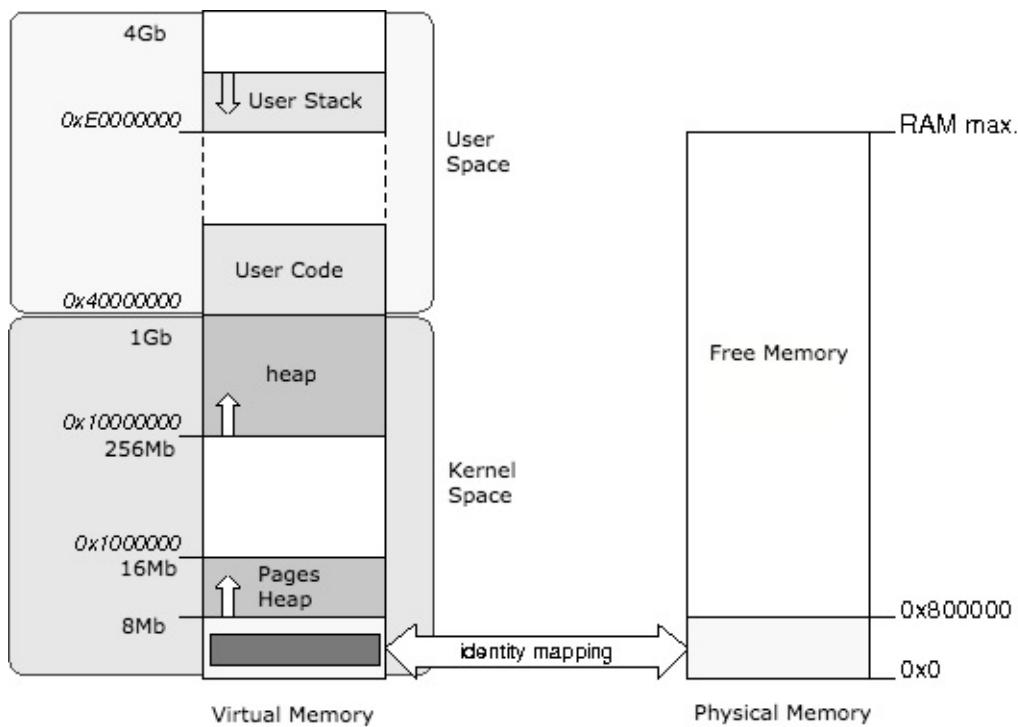
- The kernel
- GDT, IDT et TSS
- Kernel Stack
- Some space reserved to hardware (video memory, ...)
- Page directory and pages table for the kernel

The rest of the physical memory is freely available to the kernel and applications.



Virtual Memory Mapping

The address space between the beginning of memory and `0x40000000` address is the kernel space, while the space between the address `0x40000000` and the end of the memory corresponds to user space:



The kernel space in virtual memory, which is using 1Gb of virtual memory, is common to all tasks (kernel and user).

This is implemented by pointing the first 256 entries of the task page directory to the kernel page directory (In [vmm.cc](#)):

```
/*
 * Kernel Space. v_addr < USER_OFFSET are addressed by the kernel pages table
 */
for (i=0; i<256; i++)
    pdir[i] = pd0[i];
```