Project Report

# Environment Setup

## Introduction

We need a base system to design and make our Kernel. We will be using a \*nix system, with GNU tool chain. For a windows system we can use Cygwin (\*nix emulation environment).

## Directory Structure

project  
 |  
 +-- src  
 |  
 +-- docs

All source files are stored in ‘src’ and all the documentation is stored at ‘docs’

## Compiling

GUN tool chain: gcc, ld, gas

Assembly codes in Intel Syntax (much human readable than AT&T syntax)

NASM: Netwide Assembler

Bootloader: GRUB (Grand Unified Bootloader). Floppy Disk with GRUB preloaded onto it.

## Running

Can’t get bare hardware test bed system due to cost and bug management

We are using VMWare for virtualization due to ability of bugTracking with logfile. The testBed system configurations:

## VMWare

Processor: 1, 32 bit processor, 1GB HDD, 250 MB RAM, Floppy/CD/DVD Support, Network Adapter, Sound and VGA compatible.

## Useful Scripts:

Scripts are useful for automation of various tasks. As we are going to do lots of testing and making(compiling and linking) with the project, so we will be using scripts to automate that task for us.

### Makefile

Compiles all the files in ‘src’ and links them together into one ELF binary, ‘kernel’.

### link.ld

Linker script, which makes sure that everything, goes in the correct place. It tells LD how to set up the kernel image.

### update\_image.sh

This script pokes the new kernel binary into the floppy image file.

# Genesis

## Boot Code

### boot.s

It’s the kernel start location. It defines multiboot header.

Multiboot is a standard to chich GRUB expects a kernel to comply. It’s a way for the bootloader to

Know exactly what environment the kernel wants/needs when it boots

Allow the kernel to query the environment it is in.

To make kernel multiboot compatible, we need to add the header structure in our kernel. This is done in the initial 4 Mb memory of the Kernel.

dd – The command that lets us embed specific constants in our code.

MBOOT\_HEADER\_MAGIC

A Magic Number (0x1BADB002). This identifies the kernel as multiboot-compatible.

MBOOT\_HEADER\_FLAGS

A field of flags, We ask for GRUB to page-align all the kernel sections.

MBOOT\_CHECKSUM

Magic number + Flags + Checksum = 0 (ZERO). It is used for error checking.

mboot

The address of the structure that we are currently writing. GRUB uses this to tell if we are expected to be relocated.

code, bss, end, start

These symbols are all defined by the linker. We use them to tell GRUB where the different section of our kernel can be located.

On bootup, GRUB will load a pointer to another information structure into the EBX register. This can be used to query the environement GRUB set up for us.

So, immediately on bootup, the asm snippet tells the CPU to push the contents of EBX onto the stack (EBX now contains a pointer to the multiboot information structure), disable interrupts (CLI), call our 'main' C function, then enter an infinite loop.

## Adding C Code

GCC on x86 uses the \_\_cdecl calling convention:

* All parameters to a function are passed on the stack.
* The parameters are pushed *right-to-left*.
* The return value of a function is returned in EAX.

so the function call:

d = func (a, b, c);

push [c]  
push [b]  
push [a]  
call func  
mov [d], eax

So, our asm snippet above, that 'push ebx' is actually passing the value of ebx as a parameter to the function main().

### main.c

Defines the C-Code kernel entry point, calls initialization routines

## Compiling, linking and running

CLFLAGS: Stop GCC trying to link to Linux C library with our kernel.

Set CLFLAGS with

-nostdlib: No standard library

-nostdinc: No standard include

-fno-builtin: No inbuilt function

-fno-stack-protector: No stack protector

Now make file and update image using the scripts. The resultant image is available for run.

Boot into the test-bed using this disk and the system will freeze while saying ‘starting up…’.

Use log file to check for the EAX register value. It’ll be set at ‘deadbaba’ – the return value of main().

# Screen

So, now that we have a 'kernel' that can run and stick itself into an infinite loop, it's time to get something interesting appearing on the screen. Along with serial I/O, the monitor will be our most important ally in the debugging.

## Theory

Our kernel gets booted by GRUB in text mode. That is, it has available to it a framebuffer (area of memory) that controls a screen of characters (not pixels) 80 wide by 25 high. This will be the mode our kernel will operate in.

The area of memory known as the framebuffer is accessible just like normal RAM, at address 0xB8000. It is important to note, however, that it is not actually normal RAM. It is part of the VGA controller's dedicated video memory that has been memory-mapped via hardware into our linear address space. This is an important distinction.

C:\Users\Vikram Tiwari\Desktop\Project Presentations\the_screen_word_format.pngThe framebuffer for word format is just an array of 16-bit words, each 16-bit value representing the display of one character. The offset from the start of the framebuffer of the word that specifies a character at position x, y is:

(y \* 80 + x) \* 2

What's important to note is that the '\* 2' is there only because each element is 2 bytes (16 bits) long. If we're indexing an array of 16-bit values, for example, our index would just be y\*80+x.

In ASCII (unicode is not supported in text mode), 8 bits are used to represent a character. That gives us 8 more bits which are unused. The VGA hardware uses these to designate foreground and background colours (4 bits each). The splitting of this 16-bit value is shown in the diagram to the right.

4 bits for a colour code gives us 15 possible colours we can display:

0:black, 1:blue, 2:green, 3:cyan, 4:red, 5:magenta, 6:brown, 7:light grey, 8:dark grey, 9:light blue, 10:light green, 11:light cyan, 12:light red, 13:light magneta, 14: light brown, 15: white.

The VGA controller also has some ports on the main I/O bus, which we can use to send it specific instructions. (Among others) it has a control register at 0x3D4 and a data register at 0x3D5. We will use these to instruct the controller to update it's cursor position (the flashy underbar thing that tells us where our next character will go).

## Code:

### common.c and common.h

Commonly-used global functions include functions for writing to and reading from the I/O bus, and some typedefs that will make it easier for us to write portable code. They are also the ideal place to put functions such as memcpy/memset etc.

### monitor.h

Defines the interface for monitor.c

#### Moving the cursor

To move the hardware cursor, we must firstly work out the linear offset of the x,y cursor coordinate. We do this by using the equation above. Next, we have to send this offset to the VGA controller. For some reason, it accepts the 16-bit location as two bytes. We send the controller's command port (0x3D4) the command 14 to tell it we are sending the high byte, and then send that byte to port 0x3D5. We then repeat with the low byte, but send the command 15 instead.

#### Scrolling the screen

At some point we're going to fill up the screen with text. It would be nice if, when we do that, the screen acted like a terminal and scrolled up one line.

#### Writing a character to the screen

Now the code gets a little more complex. But, if we look at it, we'll see that most of it is logic as to where to put the cursor next - there really isn't much difficult there.

The bit that actually does the writing is here:

location = video\_memory + (cursor\_y\*80 + cursor\_x);  
\*location = c | attribute;

Set 'location' to point to the linear address of the word corresponding to the current cursor position (see equation above).

Set the value at 'location' to be the logical-OR of the character and 'attribute'. Remember that we shifted 'attribute' left 8 bits above, so actually we're just setting 'c' as the lower byte of 'attribute'.

#### Clearing the screen

Clearing the screen is also dead easy. Just fill it with loads of spaces.

# GDT and IDT

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

## The Global Descriptor Table (theory)

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

With segmentation, every memory access is evaluated with respect to a segment. That is, the memory address is added to the segment's base address, and checked against the segment's length. We can think of a segment as a window into the address space - The process does not know it's a window; all it sees is a linear address space starting at zero and going up to the segment length.

With paging, the address space is split into (usually 4KB, but this can change) blocks, called pages. Each page can be mapped into physical memory - mapped onto what is called a 'frame'. Or, it can be unmapped. Like this we can create virtual memory spaces.

Both of these methods have their advantages, but paging is much better. Segmentation is, although still usable, fast becoming obsolete as a method of memory protection and virtual memory. In fact, the x86-64 architecture requires a flat memory model (one segment with a base of 0 and a limit of 0xFFFFFFFF) for some of its instructions to operate properly.

Segmentation is, however, totally in-built into the x86 architecture. It's impossible to get around it. So here we're going to set up our own Global Descriptor Table - a list of segment descriptors.

As mentioned before, we're going to try and set up a flat memory model. The segment's window should start at 0x00000000 and extend to 0xFFFFFFFF (the end of memory). However, there is one thing that segmentation can do that paging can't, and that's set the ring level.

A ring is a privilege level - zero being the most privileged, and three being the least. Processes in ring zero are said to be running in kernel-mode, or supervisor-mode, because they can use instructions like sti and cli, something which most processes can't. Normally, rings 1 and 2 are unused. They can, technically, access a greater subset of the supervisor-mode instructions than ring 3 can. Some microkernel architectures use these for running server processes, or drivers.

A segment descriptor carries inside it a number representing the ring level it applies to. To change ring levels, among other things, we need segments that represent both ring 0 and ring 3. GRUB sets up a GDT for us. The problem is to know where it is, or what’s in it. So we can overwrite it, and our machine gets triple-fault and resets.

In the x86, we have 6 segmentation registers. Each holds an offset into the GDT. They are cs (code segment), ds (data segment), es (extra segment), fs, gs, ss (stack segment). The code segment must reference a descriptor which is set as a 'code segment'. There is a flag for this in the access byte. The rest should all reference a descriptor which is set as a 'data segment'.

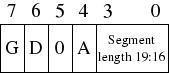
V

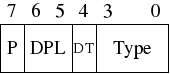
## Code:

### descriptor\_tables.h

The format of the access byte and the format of the granularity byte is given.

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

 Access Byte Format

 Granularity Byte Format

The base is the address of the first entry in our GDT, the limit being the size of the table minus one (the last valid address in the table).

### descriptor\_tables.c

The gdt\_flush function - this will be defined in an ASM file, and will load our GDT pointer for us.

init\_gdt initially sets up the gdt pointer structure - the limit is the size of each gdt entry \* 5 - we have 5 entries. Why 5? Well, we have a code and data segment descriptor for the kernel, code and data segment descriptors for user mode, and a null entry. This must be present.

gdt\_init then sets up the 5 descriptors, by calling gdt\_set\_gate. gdt\_set\_gate just does some severe bit-twiddling and shifting. The only thing that changes between the 4 segment descriptors is the access byte - 0x9A, 0x92, 0xFA, 0xF2. We can see, if we map out the bits and compare them to the format diagram above, the bits that are changing are the type and DPL fields. DPL is the descriptor privilege level - 3 for user code and 0 for kernel code. Type specifies whether the segment is a code segment or a data segment (the processor checks this often). Finally, we have our ASM function that will write the GDT pointer.

# The Interrupt Descriptor Table (theory)

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

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

## Faults, traps and exceptions

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

The special, CPU-dedicated interrupts are shown below.

0 - Division by zero exception  
1 - Debug exception  
2 - Non maskable interrupt  
3 - Breakpoint exception  
4 - 'Into detected overflow'  
5 - Out of bounds exception  
6 - Invalid opcode exception  
7 - No coprocessor exception  
8 - Double fault (pushes an error code)  
9 - Coprocessor segment overrun  
10 - Bad TSS (pushes an error code)  
11 - Segment not present (pushes an error code)  
12 - Stack fault (pushes an error code)  
13 - General protection fault (pushes an error code)  
14 - Page fault (pushes an error code)  
15 - Unknown interrupt exception  
16 - Coprocessor fault  
17 - Alignment check exception  
18 - Machine check exception  
19-31 – Reserved

## Code:

### descriptor\_tables.h

Very similar to the GDT entry and ptr structs. The flags field format is shown on the right. The lower 5-bits should be constant at 0b0110 - 14 in decimal. The DPL describes the privilege level we expect to be called from - in our case zero, but as we progress we'll have to change that to 3. The P bit signifies the entry is present. Any descriptor with this bit clear will cause an "Interrupt Not Handled" exception.

## interrupt.s

When the processor receives an interrupt, it saves the contents of the essential registers (instruction pointer, stack pointer, code and data segments, flags register) to the stack. It then finds the interrupt handler location from our IDT and jumps to it.

Now, just like POSIX signal handlers, we don't get given any information about what interrupt was called when our handler is run. So, unfortunately, we can't just have one common handler, we must write a different handler for each interrupt we want to handle. We want to keep the amount of duplicated code to a minimum. We do this by writing many handlers that just push the interrupt number (hardcoded in the ASM) onto the stack, and call a common handler function.

That's all gravy, but unfortunately, we have another problem - some interrupts also push an error code onto the stack. We can't call a common function without a common stack frame, so for those that don't push an error code, we push a dummy one, so the stack is the same.

Our common interrupt handler firstly uses the 'pusha' command to push all the general purpose registers on the stack. It uses the 'popa' command to restore them at the end. It also gets the current data segment selector and pushes that onto the stack, sets all the segment registers to the kernel data selector, and restores them afterwards. This won't actually have an effect at the moment, but it will when we switch to user-mode. It also calls a higher-level interrupt handler - isr\_handler.

When an interrupt fires, the processor automatically pushes information about the processor state onto the stack. The code segment, instruction pointer, flags register, stack segment and stack pointer are pushed. The IRET instruction is specifically designed to return from an interrupt. It pops these values off the stack and returns the processor to the state it was in originally.

### isr.c

The interrupt handler prints a message out to the screen, along with the interrupt number it handled. It uses a structure registers\_t, which is a representation of all the registers we pushed, and is defined in isr.h

### isr.h

Interface and structures for high level interrupt service routines.

## Testing

Causes two software interrupts: 3 and 4