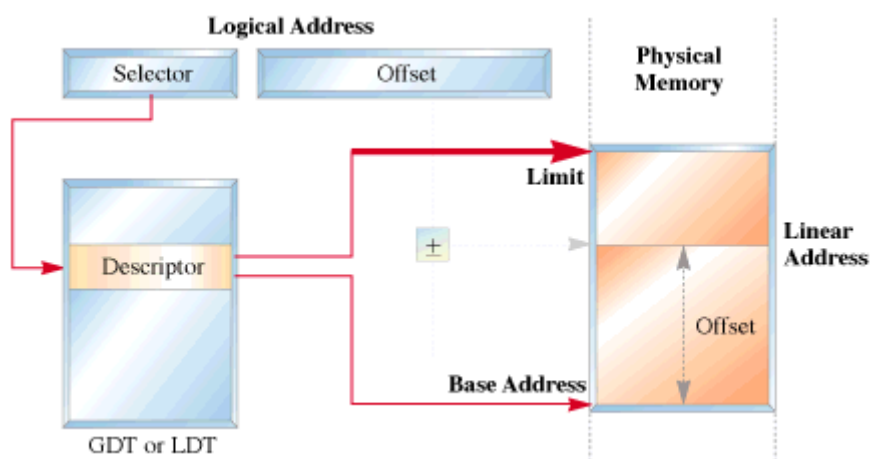


Sistemas de Computación

Modo Protegido BIOS, UEFI y MBR



Docentes:

- Ing. Miguel Angel Solinas
- Ing. Javier Alejandro Jorge

Autores:

Matrícula	Apellido y Nombre	Carrera
39402787	Gauna, Macarena del Sol	Ing. en Comp.
40502859	Coronati, Federico Joaquín	Ing. en Comp.



Índice

Índice	2
Objetivo	3
Introducción	4
Conclusión	10
Bibliografía	11



Objetivo

El objetivo de este trabajo es analizar y comprender los modos de funcionamiento de los procesadores de arquitectura x86, en concreto, poder distinguir entre modo real y modo protegido. Además, se comentan aspectos básicos sobre el arranque de las computadoras, como BIOS, UEFI y MBR, que se detallan más adelante.



Introducción

Antiguamente, con la serie de procesadores **8086** de 16 bits, estos operaban en “**Modo Real**”. El modo real está caracterizado por 20 bits de espacio de direcciones segmentado (significando que solamente se puede direccionar 1 MB de memoria), acceso directo del software a las rutinas del BIOS (Basic Input Output System) y el hardware periférico.

Los procesadores posteriores de **arquitectura x86** inician en modo real cada vez que arranca la PC, inclusive cuando se resetea la misma. Es un modo muy básico que no tiene conceptos de protección de memoria o multitarea a nivel de hardware.

Por otro lado, tenemos el “**Modo Protegido**”, el cual se añadió a los procesadores x86 de la serie 80286 y está incluido en los posteriores. Este modo tiene un número de nuevas características diseñadas para mejorar la **multitarea** y la **estabilidad** del sistema, tales como la **protección de memoria**, y **soporte de hardware para memoria virtual** como también la **conmutación de tarea**.

La gran mayoría de los sistemas operativos modernos se ejecutan en modo protegido, tales como GNU/Linux, Windows, Mac OS X, FreeBSD, entre otros.

Si bien el modo protegido añade muchas características que ayudan a mejorar el desempeño y la estabilidad de la computadora, los procesadores modernos aún siguen teniendo el modo real en su arranque, de modo de poder ejecutar código antiguo y aprovechar la **retrocompatibilidad** para ahorrarse el trabajo de refactorizar los viejos programas que se usaban en CPUs de 16 bits. En la mayoría de los casos, la retrocompatibilidad implica sacrificar recursos o trabajar con programas subóptimos, en pos de abaratar los costos de replantear el software antiguo. Este fue el impulso que le permitió a Intel posicionarse en el mercado de CPUs a fines del siglo XX.



UEFI

La sigla **UEFI** significa **Unified Extensible Firmware Interface** (es decir, interfaz de firmware extensible unificada). Esta interfaz especial se encarga de arrancar la mainboard o placa base del ordenador y los componentes de hardware relacionados con ella. De este modo, la interfaz es la responsable de que se cargue un gestor de arranque (**bootloader**) concreto en la memoria principal, que será el que iniciará las acciones rutinarias de arranque.

Para poder usar la interfaz UEFI, el ordenador necesita disponer de un firmware especial en la placa base. Al encender el ordenador, el firmware utiliza la interfaz UEFI como una capa operativa que actúa de intermediaria entre el mismo firmware y el sistema operativo. Para que el modo UEFI se pueda iniciar antes de que el ordenador en sí haya arrancado realmente, **se implementa de forma permanente en la placa base, en un chip de memoria**. Así, como parte integral del firmware de la placa base, la programación UEFI se mantiene incluso cuando la PC está apagada.

UEFI es el sucesor de **BIOS, Basic Input Output System**. UEFI presenta varias ventajas:

1. Fácil de programar (lenguaje de programación C)
2. Estructura modular, que aporta flexibilidad y permite hacer modificaciones para entornos concretos de hardware
3. UEFI puede ampliarse con funciones y programas especiales (por ejemplo, gestión de derechos digitales, juegos, navegadores web, funciones de monitoreo de hardware o control del ventilador)
4. Manejo más fácil con ayuda del ratón y de la interfaz gráfica
5. Herramienta de línea de comandos para el diagnóstico y la búsqueda de errores (UEFI Shell)
6. Conectividad de red incluso sin el sistema operativo no está activo
7. Mayor seguridad gracias al modo Secure Boot

Para acceder UEFI basta con teclear repetidamente la tecla asignada durante el **Power On Self Test (POST)** es decir apenas se enciende la PC, antes de que comience a cargarse el sistema operativo.

Además se hacen posibles los sistemas multiarranque o multiboot, es decir, permite instalar varios sistemas operativos con sus propios **bootmanagers** de forma paralela. Si es necesario, el usuario también puede decidir durante el proceso de arranque si quiere iniciar, por ejemplo, Linux en lugar de Windows. También se pueden ejecutar aplicaciones pre-boot (por ejemplo, para activar y utilizar herramientas de diagnóstico o soluciones de backup).

Se puede desarrollar software UEFI a partir de cualquier distribución Linux, gracias a la librería **gnu-efi**. A continuación se muestra un código de ejemplo, para mostrar un mensaje "Hello World".



```
#include <efi.h>
#include <efilib.h>

EFI_STATUS
efi_main (EFI_HANDLE image, EFI_SYSTEM_TABLE *systab)
{
    SIMPLE_TEXT_OUTPUT_INTERFACE *conout;

    conout = systab->ConOut;
    InitializeLib(image, systab);
    uefi_call_wrapper(conout->OutputString, 2, conout, L"Hello World!\n\r");

    return EFI_SUCCESS;
}
```

Un **bug conocido de UEFI** fue en las **Lenovo Thinkcentre M92p**, las cuales sólo permitían bootear a partir de Windows Boot Manager y Red Hat Enterprise Linux, lo cual deja muchas distribuciones y sistemas operativos sin poder bootear en este modelo de computadora.

Se suele considerar el cierto riesgo de seguridad que supone UEFI como otro inconveniente de la interfaz. La conexión directa de red en la fase de arranque puede hacer posible la entrada de malware al ordenador antes de que los mecanismos de seguridad actúen.

Intel denominó al sucesor de MEBx (Management Engine BIOS Extension) a partir de 2017 como **CSME**, el cual sería una versión totalmente nueva y muy mejorada del primero. **CSME** hace referencia a las siglas de **Converged Security & Manageability Engine**, algo así como el motor de seguridad y capacidad de administración convergente de cada plataforma. Este módulo se usa, por ejemplo, para implementar versiones antiguas de 32 bits en Windows 7, 8 o en hardware UEFI moderno.

Actualmente engloba tres apartados/módulos mucho más conocidos que su propio término y tremendamente relevantes para cada placa base como son:

- Management Engine (ME)
- Servicios de plataforma de servidor (Server Platform Services) o SPS
- Trusted Execution Engine (TXE)

Su función principal a groso modo es el **proporcionar un entorno de ejecución aislado** y protegido desde el host a modo de software que se ejecuta directamente en la CPU mediante un firmware que para esta última es totalmente transparente.

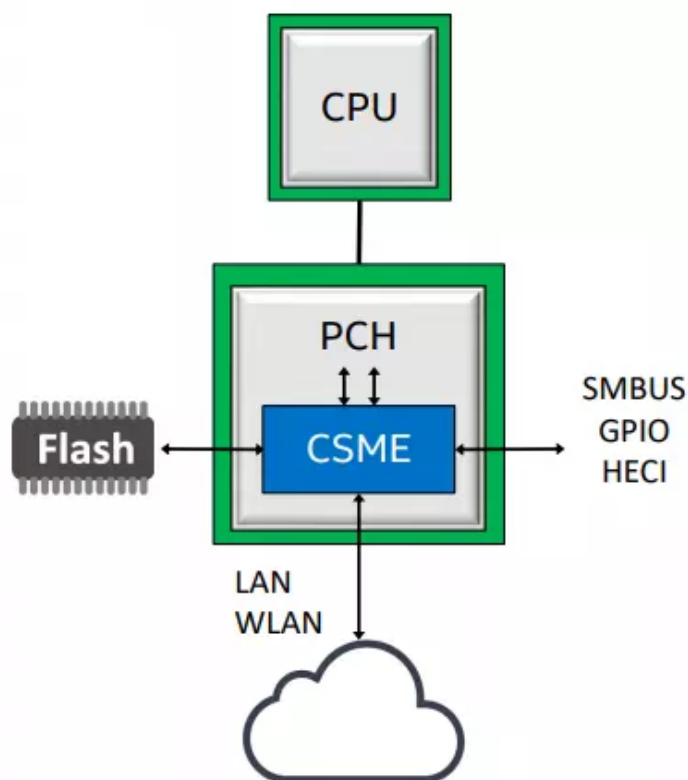


Figura 1: CSME para aislar del CPU

Por otro lado, **coreboot** es una alternativa a estos firmwares pero **Open Source**, es decir de código abierto. Ya que debe ejecutarse directamente en la Motherboard, debe ser compatible con los chipsets de estas, por lo tanto **no está disponible para todas las placas** del mercado, sino para un número limitado de modelos.

Mayor rapidez en el arranque, evitar el malware o adware de los fabricantes en la BIOS, capacidad de habilitar virtualización por hardware en aquellos equipos que no lo posean o una mejor gestión de la energía en portátiles, son algunas de las ventajas que puede ofrecer Coreboot, el cual está escrito en lenguaje de programación C.

El nuevo **Oryx Pro** es el primer portátil de System76 que combina Coreboot y NVIDIA.

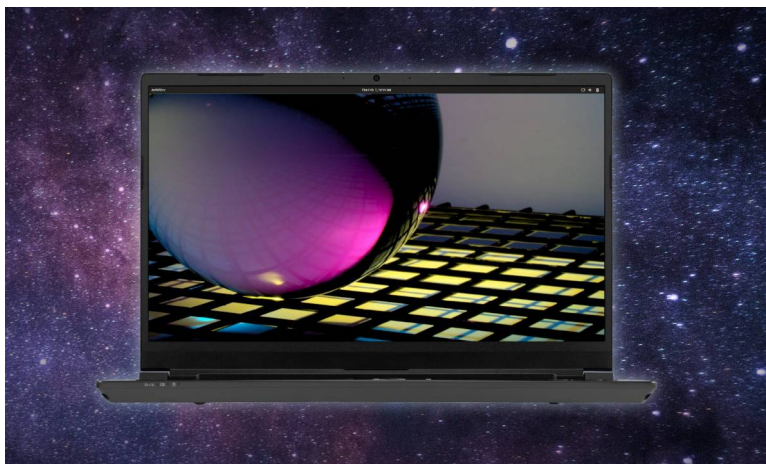


Figura 2: Portátil Oryx Pro, que utiliza Coreboot

Linker

El **Linker o Enlazador** es un programa de computadoras que toma uno o varios ficheros object (.o .obj), que se generan a partir de un **compilador** o de assembler, y los combina en un único archivo ejecutable, de librería o incluso otro object, que será representado en forma binaria para representar las instrucciones y código que debe ejecutar el procesador.

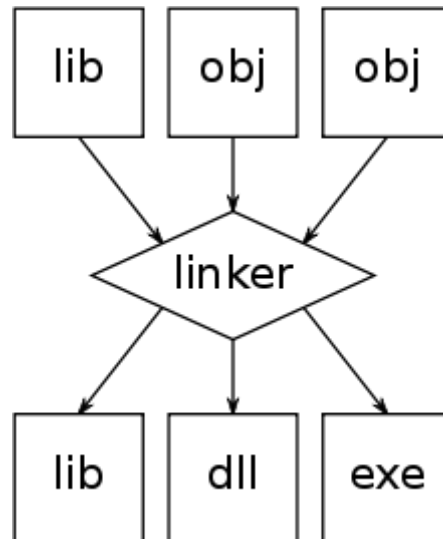


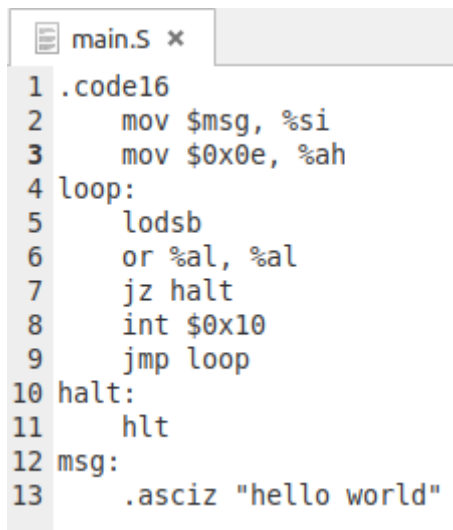
Figura 3: Proceso de Enlazado posterior al Compilado

El **Linker** comienza asignando la variable especial `__start = 0x7c00`, que indica una posición de memoria absoluta en donde se buscarán 512 bytes correspondientes al **MBR (Master Boot Record)**, es decir que especificarán las instrucciones de arranque del sistema operativo y la tabla de particiones del dispositivo de almacenamiento.



Desarrollo

Para llevar a cabo este trabajo de laboratorio, se comenzó por cargar en un pendrive el código **HelloWorld** suministrado por los docentes para probarlo en una PC. Este código tiene por objetivo imprimir en la pantalla el mensaje “hello world” y luego detener el procesador, que permanece detenido hasta reiniciar la PC. Es importante destacar que este código operará en modo real, con instrucciones de 16 bits. A continuación se presenta el código assembly de este pequeño programa destinado a imprimir un mensaje.



```
main.S x
1 .code16
2     mov $msg, %si
3     mov $0x0e, %ah
4 loop:
5     lodsb
6     or %al, %al
7     jz halt
8     int $0x10
9     jmp loop
10 halt:
11     hlt
12 msg:
13     .asciz "hello world"
```

Figura 4: Código Assembly HelloWorld en Modo Real

Para ejecutar este código, se puede usar QEMU, que es un emulador y virtualizador open source ideal para realizar estas pruebas. Una vez probado el código en QEMU se procedió a experimentar en distintas máquinas.

Empleando una herramienta llamada Plop Boot Manager, la cual es un pequeño gestor de arranque usado por técnicos de PC para poder bootear desde USB en Motherboards que no permitían esta función gracias a los controladores que ofrece Plop. Esta herramienta se virtualizó con VirtualBox y el resultado fue similar al del QEMU, a continuación se ilustra con una imagen.

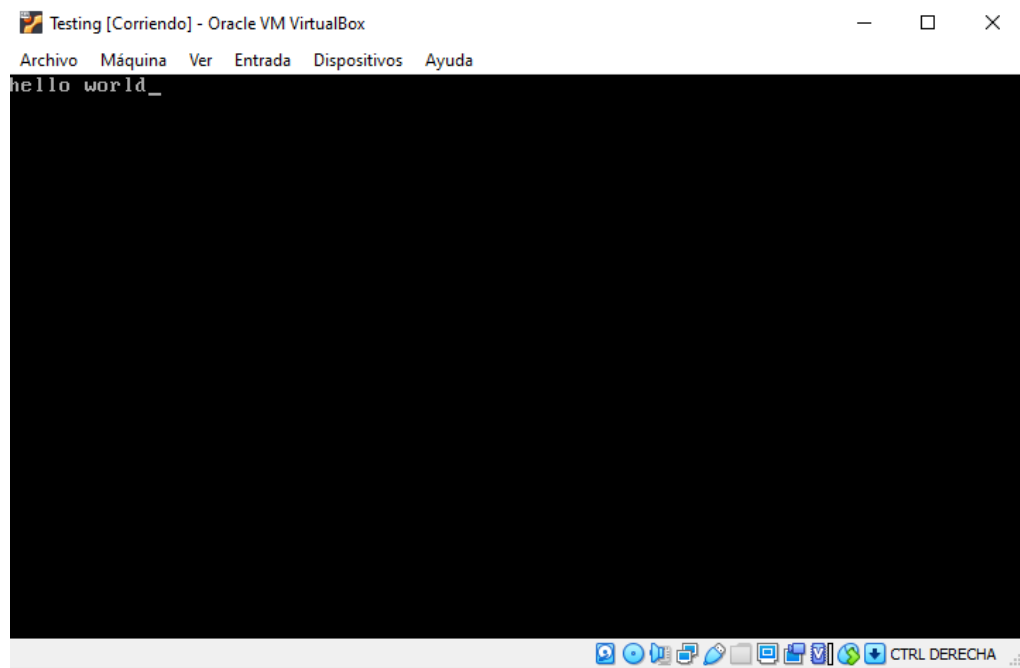


Figura 5: HelloWorld booteado con Plop en VirtualBox

Sin embargo, cuando el mismo código se quiso probar en computadoras reales, se obtuvo un único carácter, una cara feliz “☺”. Esto se probó en 2 computadoras diferentes, siendo el mismo pendrive que se ejecutó previamente con Plop Boot Manager. Esto puede deberse a que las computadoras reales deben requerir algún controlador o código extra para manipular los datos en la memoria de video del modo real.

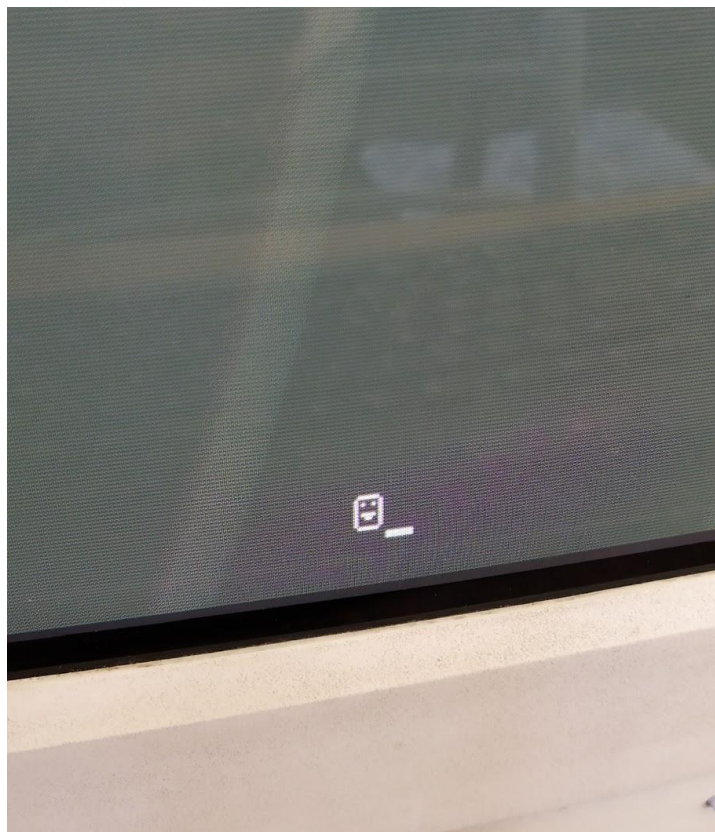


Figura 6: Caracter ☺ obtenido luego de bootear HelloWorld en una PC real



Modo Protegido

Luego, para trabajar en modo protegido se tomó como ejemplo el post de dbritos, cuyo link se encuentra en el final de este documento. Si bien este post expresa el código assembly en **sintaxis Intel**, se optó por trabajar en **sintaxis AT&T** ya que los anteriores ejemplos estaban en esta sintaxis, de esta manera no sería necesario modificarlas.

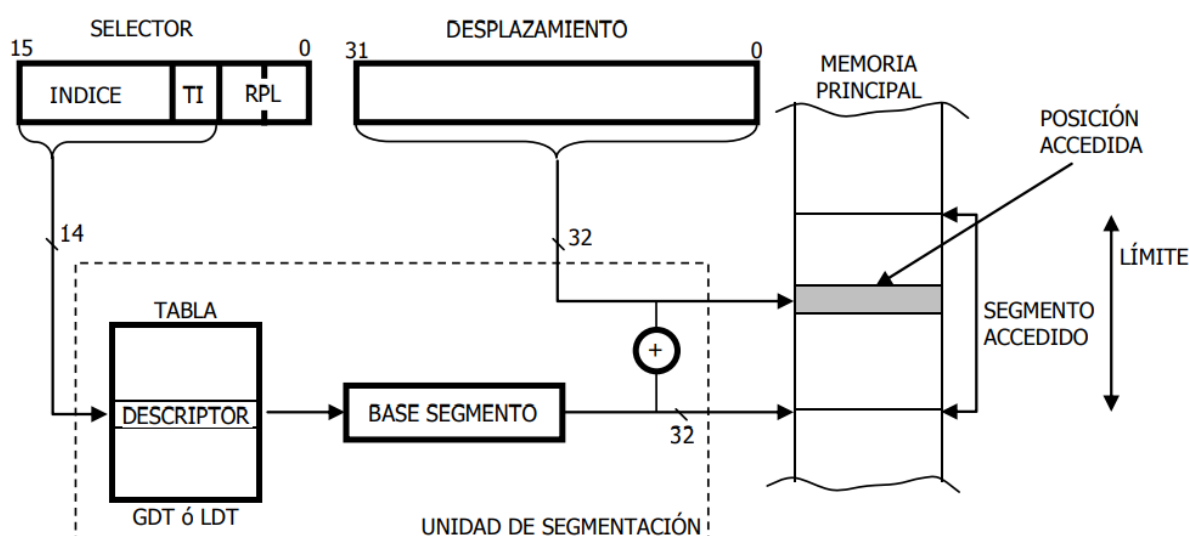
Para un mejor monitoreo de la ejecución del código en QEMU se descargó y configuró **GDB Dashboard**, una herramienta que permite visualizar el valor de los registros del procesador, permitiendo seleccionar cuáles registros ver. Estos se configuran mediante el siguiente comando.

dashboard registers -style list 'rax rbx rcx rdx rsi rdi rbp rsp r8 r9 r10 r11 r12 r13 r14 r15 rip eflags cs ss ds es fs gs'

El código assembly empleado para imprimir un mensaje en la memoria de video VGA, se encuentra en el repositorio de Github del proyecto con el nombre de "main2.S". En el mismo, a grandes rasgos se realiza lo siguiente.

Se deshabilitan las interrupciones, luego se cargan las direcciones de memoria de la tabla **GDT o Global Descriptor Table**, luego se escribe un "1" en el bit menos significativo de **CR0 (Control Register 0)**, de esta manera estamos pasando del modo real (modo en el cual inicia la computadora) al modo protegido, que brinda todas las mejoras de rendimiento y seguridad mencionadas en la introducción.

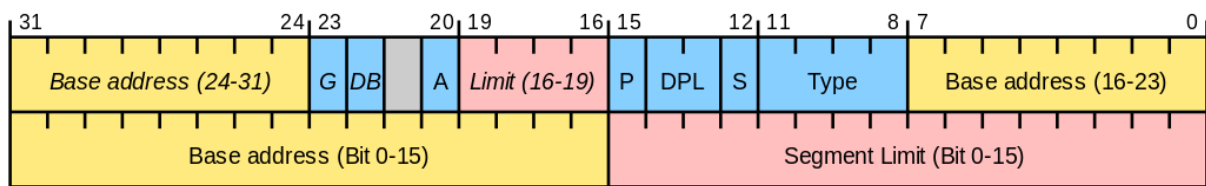
La tabla de descriptores global ayuda a direccionar la memoria virtual en memoria física, en lo que se conoce como unidad de segmentación, que nos permite direccionar una dirección de memoria virtual de 46 bits (selector + desplazamiento) a una dirección física de 32 bits. Si un programa tiene el código y los datos en direcciones independientes de la memoria física, debe indicarse con 2 entradas en la tabla GDT (es decir 2 descriptores) cada uno denotando una base de segmento, a la cual se añade el offset o desplazamiento para obtener la dirección de la memoria principal de las instrucciones y datos de ese programa.



En la GDT, se distinguen el **segmento descriptor de código** y el **segmento descriptor de datos**, que si nos fijamos detenidamente, la configuración de los bits de ambos segmentos son iguales excepto por el bit de tipo, que indica código o datos,



respectivamente. A continuación se ilustra una imagen de la disposición de los bits de configuración de una GDT.



La descripción detallada de estos bits no se dará en este informe para no extender demasiado el texto, pero se revisó en el libro Arquitectura de Microprocesadores, de José María Angulo.

Una vez que se está en modo protegido, se actualiza el valor de los registros de segmento DS ES FS GS y SS. Luego se procede a configurar la **memoria de video VGA** para imprimir el mensaje "Esta computadora se autodestruiría en 3 2 1". Se eligieron los colores rojo y blanco para destacar el mensaje en la consola negra. Nótese que se asigna la dirección de memoria **0xb8000**, que es la correspondiente a VGA para monitores multicolor.

Para ejecutar este código en QEMU se dispone de un archivo Makefile, que compila, linkea y ejecuta QEMU simplemente al ejecutar el comando make. Luego, para debuggear con GDB Dashboard, se usan los siguientes comandos.

```
gdb  
target remote localhost:1234  
dashboard registers -style list 'rax rbx rcx rdx rsi rdi rbp rsp r8 r9 r10 r11 r12  
r13 r14 r15 rip eflags cs ss ds es fs gs'  
set architecture i8086  
br *0x7c00  
continue
```

A continuación se presentan capturas de pantalla del proceso de debuggeo.

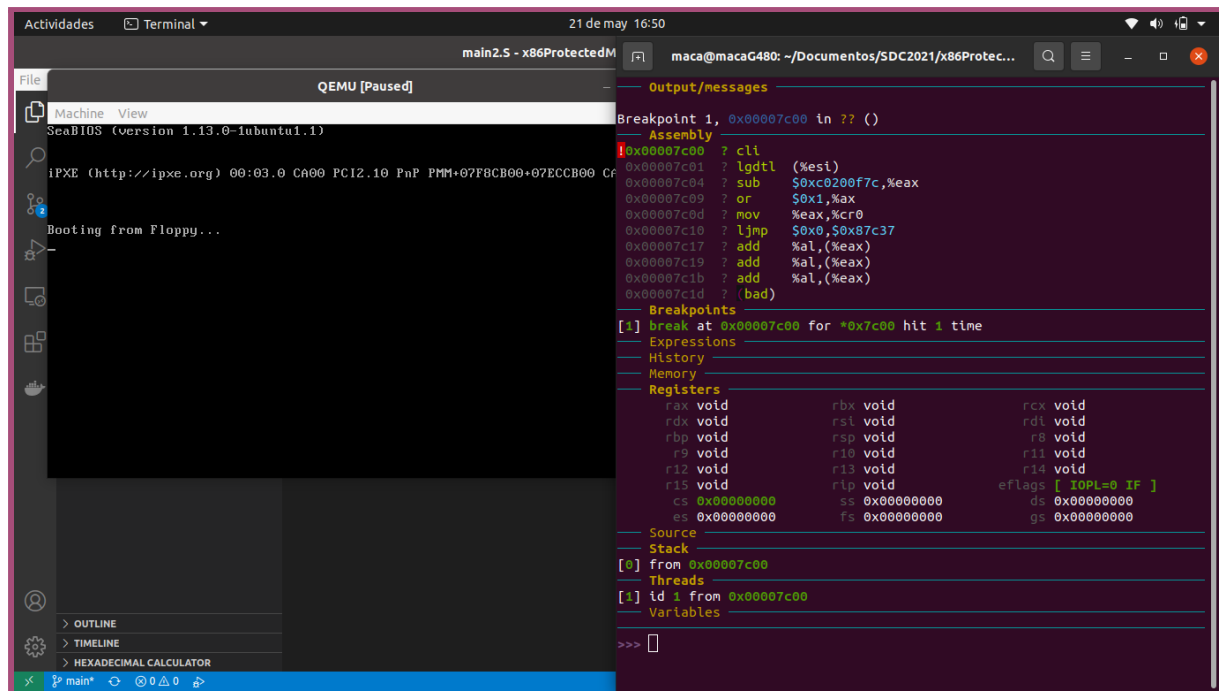


Figura 7: Debug pausado en el breakpoint en 0x7c00

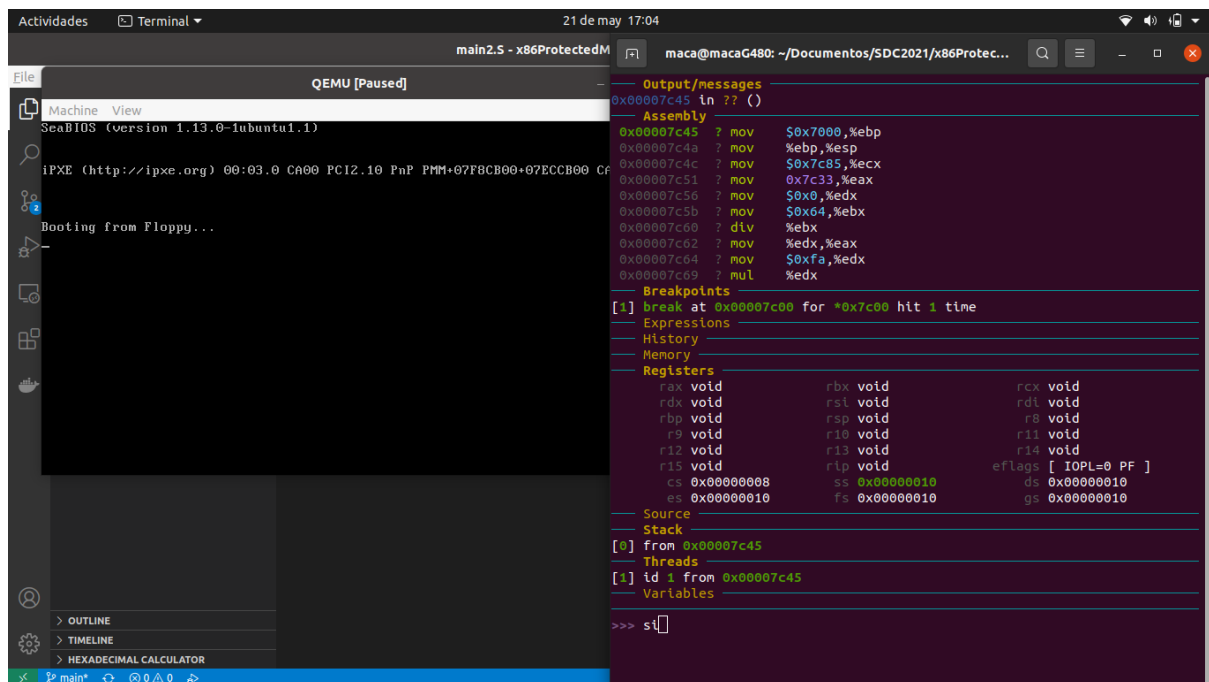


Figura 8: Variación de los registros con el single instruction

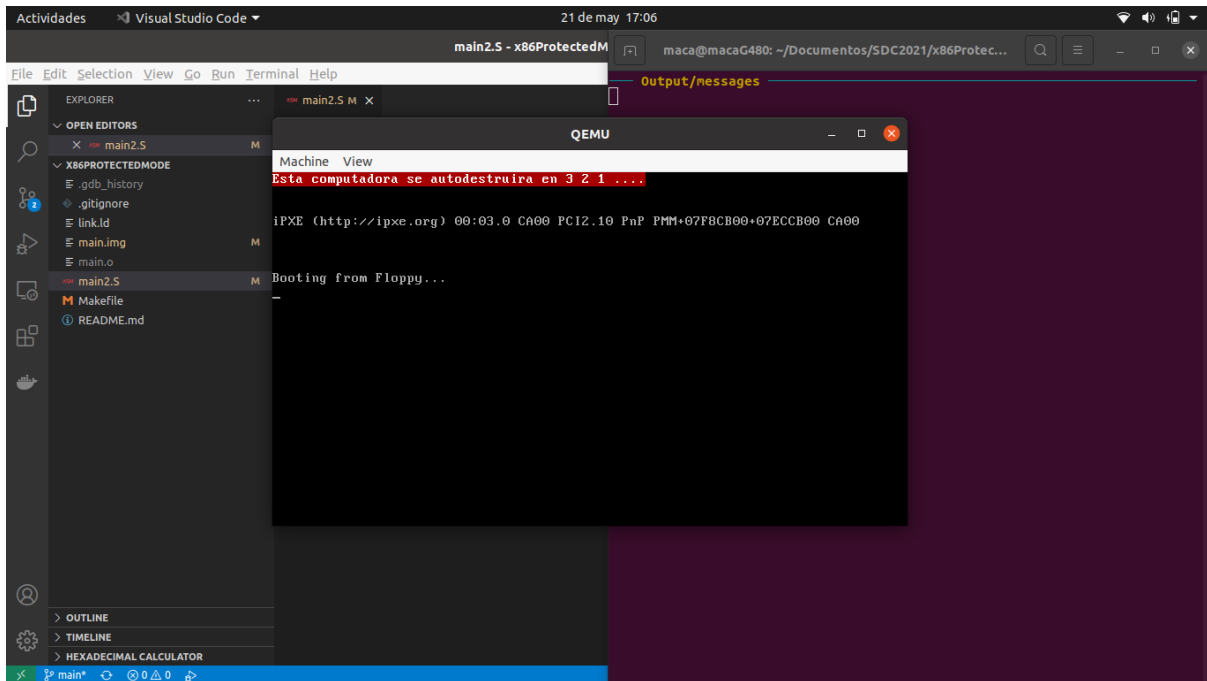


Figura 9: Luego de realizar continue, el programa termina y queda en hlt

En caso de que la tabla GDT se defina con segmento de datos de solo lectura, al ejecutar el programa, cuando se intenta escribir en la memoria el mensaje, se activa la bandera de interrupción IF, del registro eflags, impidiendo la ejecución del código restante sin importar cuantas veces se intente continuar, ya que se ingresa a un bucle infinito.

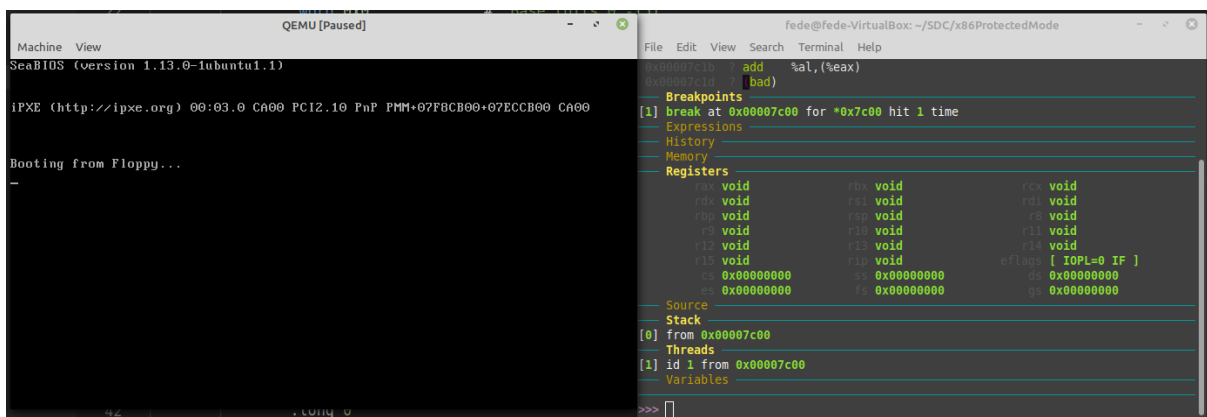


Figura 10: Segmento de Datos solo lectura

En modo protegido los **registros de segmento** se cargan con el valor 0x10, que representa el 16 en decimal, que es la cantidad de bytes que se deben dejar disponibles en la tabla GDT para almacenar el descriptor nulo (que no se utiliza de acuerdo a la documentación) y el descriptor de código que se carga primero.



```
fed@fed-VirtualBox: ~/SDC/x86ProtectedMode
File Edit View Search Terminal Help
0x00007c95 ? and %dh,0x65(%ebx)
Breakpoints
[1] break at 0x00007c00 for *0x7c00 hit 1 time
Expressions
History
Memory
Registers
rax void      rbx void      rcx void
rdx void      rsi void      rdi void
rbp void      rsp void      r8 void
r9 void       r10 void     r11 void
r12 void      r13 void     r14 void
r15 void      rip void     eflags [ IOPL=0 7F PF ]
cs 0x00000008  ss 0x00000010  ds 0x00000010
es 0x00000010  fs 0x00000010  gs 0x00000010
Source
Stack
[0] from 0x00007c85
[1] from 0x00000000
Threads
[1] id 1 from 0x00007c85
Variables
>>>
```

Figura 11: Captura donde se ve en GDB Dashboard los registros de segmento de datos



Referencias

- Arquitectura de Microprocesadores, Los Pentium a Fondo - José María Angulo Usategui
- Repositorio: <https://github.com/FeedehC/x86ProtectedMode>
- [Paso a modo protegido x86](#)
- [Printing To Screen](#)
- [Protected Mode Tutorial | TAJ Operating System](#)
- [Protected Mode Memory Addressing](#)