

# Universidad Nacional de Córdoba



Facultad de Ciencias Exactas, Físicas y Naturales  
Escuela de Electrónica y Computación

---

## Cátedra de Sistemas de Computación Trabajo de Laboratorio 3

---

**Profesor Titular:** Ing. Javier Alejandro JORGE

**Profesor Adjunto:** -

**Integrantes:**

Trucchi, Genaro

Trachta, Agustín

Rodríguez, Mateo

27 de abril de 2025

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. ¿Qué es UEFI?</b>	<b>3</b>
2.1. ¿Cómo puedo usarlo? . . . . .	3
2.2. Ejemplo de llamada a función UEFI . . . . .	4
<b>3. Casos de bugs de UEFI explotables</b>	<b>5</b>
<b>4. Converged Security and Management Engine (CSME) e Intel Management Engine BIOS Extension (MEBx)</b>	<b>6</b>
<b>5. coreboot</b>	<b>6</b>
<b>6. Linker</b>	<b>7</b>
6.1. ¿Qué es un linker? . . . . .	7
6.2. ¿Qué hace? . . . . .	7
<b>7. ¿Qué es la dirección 0x7C00 y por qué es necesaria?</b>	<b>8</b>
<b>8. Demostración Práctica: Compilación, Enlace y Depuración</b>	<b>9</b>
8.1. Compilación con el Ensamblador (as) . . . . .	9
8.2. Enlace con el Linker (ld) . . . . .	9
8.3. Verificación (objdump vs. hd) . . . . .	10
8.4. Depuración con QEMU y GDB . . . . .	10
8.4.1. Iniciar QEMU en modo depuración . . . . .	10
8.4.2. Sesión de GDB . . . . .	11
<b>9. Creación y prueba de un bootloader básico</b>	<b>14</b>
9.1. Objetivo . . . . .	14
9.2. Conceptos fundamentales . . . . .	14
9.3. Procedimiento realizado . . . . .	14
9.3.1. Creación del código en ensamblador . . . . .	14
9.3.2. Compilación . . . . .	15
9.3.3. Prueba en máquina virtual . . . . .	15
9.3.4. Opcional: grabación en un pendrive físico . . . . .	15
9.4. Resultados . . . . .	15
9.5. Conclusiones . . . . .	16
9.5.1. Prueba en hardware real . . . . .	16
<b>10.Desafío Final: Implementación y Explicación</b>	<b>18</b>
10.1. Crear Código Assembler para Pasar a Modo Protegido (sin macros) . . . . .	18
10.1.1. Estructura del Código y Archivos . . . . .	18
10.1.2. El Problema del Formato Binario Directo (-f bin) . . . . .	18
10.1.3. Solución: Flujo de Trabajo ELF . . . . .	19
10.1.4. Código de Transición Detallado (en <code>switch_to_pm.asm</code> ) . . . . .	19
10.2. Programa con Descriptores de Código y Datos Diferenciados . . . . .	21
10.2.1. Espacios de Memoria Diferenciados (Concepto vs. Implementación Plana) . . . . .	22

<b>11.Prueba de Protección de Memoria (Read-Only)</b>	<b>23</b>
11.1. Modificaciones al Código . . . . .	23
11.2. Ejecución y Resultados: ¿Qué Sucede? . . . . .	25
11.3. ¿Qué Debería Suceder a Continuación (Teoría del Fallo)? . . . . .	26
11.4. Verificación con GDB . . . . .	26
11.5. Valor y Propósito de los Selectores en Modo Protegido . . . . .	27
<b>12.Conclusiones</b>	<b>28</b>

## 1. Introducción

En este trabajo práctico se aborda la arquitectura UEFI, sus diferencias fundamentales con el antiguo BIOS, la forma de interactuar con ella tanto a nivel usuario como a nivel desarrollador, y se muestra un ejemplo de llamada a una de sus funciones de *Runtime Services*.

## 2. ¿Qué es UEFI?

UEFI (*Unified Extensible Firmware Interface*) es la especificación que define una interfaz de software moderna entre el firmware de la plataforma (antes BIOS) y el sistema operativo. A diferencia del BIOS tradicional, UEFI:

- Está escrito en C y opera en modo 32 o 64 bits.
- Ofrece controladores propios para hardware básico (red, almacenamiento, gráficos).
- Soporta particiones GPT (superando el límite de 2 TB de MBR).
- Incorpora un entorno seguro (*Secure Boot*) y gestión de clave pública.
- Se organiza en *Boot Services* (solo durante el arranque) y *Runtime Services* (disponibles para el SO tras el `ExitBootServices`).

### 2.1. ¿Cómo puedo usarlo?

#### Como usuario final

1. Durante el arranque, pulsar la tecla indicada (DEL, F2, F10, F12, ESC) para entrar en el *Setup* de UEFI.
2. Desde allí se puede:
  - Cambiar el orden de arranque.
  - Habilitar/deshabilitar *Secure Boot*.
  - Ajustar parámetros de hardware (incluido overclocking).
3. En sistemas UNIX-like, utilidades como `efibootmgr` permiten listar y modificar entradas UEFI desde el sistema operativo.

#### Como desarrollador / cargador de arranque

1. El firmware UEFI pasa al cargador una estructura `EFI_SYSTEM_TABLE`, que incluye punteros a:
  - `BootServices` (servicios disponibles solo durante el arranque).
  - `RuntimeServices` (servicios que perduran tras el `ExitBootServices`).
2. Para invocar una rutina UEFI, se localiza el puntero en la tabla de servicios y se llama en C/C++ correctamente, respetando la ABI UEFI (registro de parámetros, alineamiento, etc.).

## 2.2. Ejemplo de llamada a función UEFI

Una aplicación UEFI puede acceder a las variables persistentes almacenadas en la NVRAM (Non-Volatile RAM) mediante los *Runtime Services*, que son parte del entorno estándar definido por la especificación UEFI. A continuación, se muestra un ejemplo de cómo utilizar la función `GetVariable` para leer la variable global `BootOrder`, la cual contiene el orden de dispositivos para el arranque del sistema. Esta función está documentada en la sección correspondiente de la especificación oficial UEFI.

```
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/UefiBootServicesTableLib.h>

extern EFI_GUID gEfiGlobalVariableGuid;

EFI_STATUS LeerBootOrder(VOID)
{
    EFI_STATUS status;
    CHAR16 *VariableName = L"BootOrder";
    UINTN DataSize = sizeof(UINT16) * 16;          // espacio para 16 entradas
    UINT16 BootOrder[16];
    UINT32 Attributes;

    status = gST->RuntimeServices->GetVariable(
        VariableName,
        &gEfiGlobalVariableGuid,
        &Attributes,
        &DataSize,
        BootOrder
    );

    if (EFI_ERROR(status)) {
        Print(L"Error al leer BootOrder: %r\n", status);
        return status;
    }
    // Procesar BootOrder...
    return EFI_SUCCESS;
}
```

Listing 1: Lectura de la variable `BootOrder`

Más detalles en la [especificación oficial](#).

### 3. Casos de bugs de UEFI explotables

Los bugs en UEFI son particularmente peligrosos porque el firmware se ejecuta antes que el sistema operativo y con privilegios muy altos (equivalente a Ring  $-2$  o superior, conceptualmente). Explotarlos puede llevar a ataques muy persistentes y sigilosos (bootkits). Algunos casos notables:

- **LoJax (2018)**: considerado el primer rootkit UEFI in the wild', usado por el grupo APT28 (Fancy Bear). Modificaba directamente la SPI flash del firmware para inyectar un módulo malicioso que persistía incluso tras reinstalar el sistema operativo o cambiar el disco duro. Aprovechaba vulnerabilidades y configuraciones inseguras que permitían escritura no autorizada en la flash de firmware.
- **MosaicRegressor (2020)**: framework de spyware basado en UEFI desarrollado por HackingTeam. Utilizaba imágenes de firmware comprometidas para reinstalar el malware en el sistema operativo cada vez que se eliminaba, asegurando persistencia a nivel de firmware.
- **ThinkPwn (2016)**: vulnerabilidad en ciertos firmwares de Lenovo (y potencialmente otros fabricantes) en la gestión de llamadas SMM (System Management Mode). Permitía a un atacante con privilegios de administrador en el SO escalar a SMM, desde donde podía modificar el firmware UEFI protegido.
- **LogoFAIL (2023)**: conjunto de desbordamientos de búfer en parsers de imágenes de logo personalizados usados por muchos fabricantes de UEFI durante la fase DXE (Driver Execution Environment). Un atacante podía cargar una imagen de logo maliciosa que explotara este bug, obteniendo ejecución de código arbitrario en arranque temprano.
- **Vulnerabilidades en drivers UEFI específicos**: de forma recurrente se descubren bugs (por ejemplo, buffer overflows) en drivers de red, USB, almacenamiento, etc. Si un atacante controla los datos procesados por estos drivers (p. ej. un USB malicioso o un paquete de red manipulado), puede desencadenar la vulnerabilidad y ejecutar código con privilegios de firmware.

## 4. Converged Security and Management Engine (CSME) e Intel Management Engine BIOS Extension (MEBx)

**Converged Security and Management Engine (CSME) / Intel Management Engine (ME):** Es un subsistema de microcontrolador autónomo integrado en el chipset (Platform Controller Hub, PCH) de la mayoría de las placas base Intel desde 2006/2008. Funciona de forma independiente de la CPU y del sistema operativo, con su propio firmware (basado en MINIX), memoria y acceso directo a hardware crítico (RAM, red, periféricos). Sus principales funciones son:

- *Intel Active Management Technology (AMT):* administración remota fuera de banda (OOB) — encender/apagar, consola remota, incluso si el SO no responde.
- *Inicialización temprana de hardware:* parte de la configuración del sistema antes de que el BIOS/UEFI principal tome el control.
- *Funciones de seguridad:*
  - Intel Boot Guard: verifica la firma criptográfica del firmware UEFI.
  - Intel Platform Trust Technology (PTT): implementación de TPM en firmware.
  - Gestión de DRM (Protected Audio Video Path, PAVP), etc.

Su naturaleza de “caja negra” propietaria con acceso privilegiado ha generado preocupaciones de seguridad; a lo largo de los años se han descubierto vulnerabilidades críticas en el ME/CSME.

**Intel Management Engine BIOS Extension (MEBx):** Es el módulo de configuración integrado en el firmware principal (BIOS/UEFI) que permite ajustar las funciones del ME/AMT. Se accede normalmente al presionar **Ctrl+P** (o la combinación específica del fabricante) durante el POST. Desde MEBx se puede:

- Habilitar/deshabilitar AMT.
- Configurar parámetros de red para gestión OOB (dirección IP, VLAN, DNS).
- Establecer contraseñas y políticas de acceso al ME.
- Activar KVM remoto y otras opciones avanzadas de control.

## 5. coreboot

**¿Qué es coreboot?** coreboot es un proyecto de firmware libre y minimalista que reemplaza el BIOS/UEFI propietario. Su filosofía es inicializar únicamente el hardware esencial (CPU, RAM, chipset básico) y luego transferir el control a un *payload* especializado.

**Payloads comunes:**

- SeaBIOS: interfaz BIOS tradicional para compatibilidad con SOs antiguos.
- TianoCore (EDK II): implementación de UEFI completa para SOs modernos.
- GRUB2 / U-Boot: cargadores de arranque con gran flexibilidad.
- LinuxBoot: utiliza el kernel de Linux como payload para entornos Linux “bare-metal”.

### Productos que lo incorporan:

- *Chromebooks* de Google (firmware estándar).
- Laptops y desktops de fabricantes enfocados en Linux / privacidad (System76, Purism).
- Diversos sistemas embebidos, routers, servidores y dispositivos de red.
- Comunidad de porting a placas base de escritorio y portátiles de múltiples marcas.

### Ventajas de su utilización:

- *Velocidad de arranque* muy superior al firmware propietario.
- *Flexibilidad y personalización*: elección de payloads y fácil adaptación al hardware.
- *Seguridad y transparencia*: código auditable, menor superficie de ataque (aunque a veces requiere *blobs* propietarios para componentes críticos).
- *Control total* sobre el proceso de arranque y eliminación de código heredado innecesario.

## 6. Linker

### 6.1. ¿Qué es un linker?

Un linker (enlazador) es una herramienta fundamental en el proceso de desarrollo de software. Actúa como un “ensamblador” de piezas de código y datos provenientes de diferentes fuentes para crear un único archivo de salida.

### 6.2. ¿Qué hace?

Sus tareas principales son:

- **Combinación de secciones de código y datos:** Los archivos objeto (`.o`) generados por el ensamblador o el compilador contienen secciones como `.text` (código), `.data` (datos inicializados) y `.bss` (datos no inicializados). El linker toma todas las secciones homónimas de los distintos archivos y las une en una sola sección en el archivo de salida, siguiendo un script de linker (`link.ld`) o las reglas por defecto.
- **Resolución de símbolos:** Cuando el código referencia símbolos (por ejemplo, el nombre de una función como `_start` o una etiqueta de datos como `msg`), el linker busca su definición entre todos los archivos objeto y bibliotecas de entrada. Luego sustituye cada uso simbólico por la dirección final donde ese símbolo ha sido colocado en el archivo de salida. Si falta una definición, se produce un error de “referencia no definida”; si hay definiciones duplicadas, un error de “definición múltiple”.
- **Relocalización:** Los archivos objeto contienen direcciones provisionales o relativas. El linker ajusta (o “parchea”) estas direcciones para que apunten a las ubicaciones finales en el ejecutable resultante, teniendo en cuenta la dirección base especificada en el script del linker (por ejemplo, `0x7C00` en un bootloader). Esto incluye tanto saltos internos como referencias a símbolos externos.
- **Generación del archivo de salida:** Finalmente, escribe el resultado combinado, resuelto y relocalizado en un archivo. El formato puede ser un ejecutable estándar del sistema operativo (ELF, PE) o, si se solicita explícitamente (con `-oformat binary`), una imagen binaria cruda sin cabeceras adicionales.



## 7. ¿Qué es la dirección 0x7C00 y por qué es necesaria?

La dirección 0x7C00 es un estándar histórico en la arquitectura x86 que corresponde al punto de memoria física donde el BIOS carga el primer sector arrancable (MBR o VBR, 512 bytes) de un dispositivo seleccionado para el arranque. Su importancia y necesidad en el script del linker se sustenta en varios aspectos:

- **Convención del BIOS:** Cuando una PC compatible arranca en modo legado (BIOS o UEFI CSM), el firmware busca un dispositivo arrancable, lee su primer sector (512 bytes) y lo copia en la dirección física 0x7C00. Inmediatamente después, establece los registros CS:IP a 0000:7C00 y salta a esa dirección para comenzar la ejecución.
- **Cálculo de direcciones finales:** El código ensamblado (`main.S`) incluye referencias a sus propias etiquetas (por ejemplo, la dirección de la cadena `msg` o el destino del salto `jmp print_loop`). Para que esas referencias apunten a las ubicaciones correctas durante la ejecución (cuando el código resida en 0x7C00), el linker *\*debe\** conocer esta dirección base durante el proceso de enlace. Al indicar `. = 0x7C00` al principio de la sección `SECTIONS` en el script `link.ld`, le decimos al linker: “Considera que el inicio del código estará en 0x7C00 y calcula todas las direcciones relativas a esta base”.
- **Relocalización adecuada:** Gracias a esa directiva (`. = 0x7C00`), el linker puede calcular la dirección absoluta correcta para cada símbolo y referencia interna. Por ejemplo, si la etiqueta `msg` está a un desplazamiento de 26 bytes desde el inicio (`_start`), el linker codificará su dirección como  $0x7C00 + 26 = 0x7C1A$  dentro de la instrucción `movw $msg, %si`.
- **Evitar desajustes catastróficos:** Si el linker no supiera la dirección de carga 0x7C00 (por ejemplo, si asumiera una base 0), calcularía todas las direcciones internas de forma incorrecta respecto al punto de carga real. Cuando el BIOS cargase el código en 0x7C00, una instrucción como `movw $msg, %si` cargaría una dirección errónea (ej., 0x1A en lugar de 0x7C1A). El programa fallaría inmediatamente al intentar leer la cadena o realizar saltos, probablemente causando un cuelgue o reinicio.

## 8. Demostración Práctica: Compilación, Enlace y Depuración

Para ilustrar los conceptos anteriores, se realizó el proceso completo para crear, verificar y depurar un pequeño programa "Hello World" que funciona como un sector de arranque MBR en modo real de 16 bits.

### 8.1. Compilación con el Ensamblador (as)

El primer paso es convertir el código fuente en ensamblador (`main.S`, que contiene la directiva `.code16` para indicar que es código de 16 bits) en un archivo objeto (`main.o`).

```
as -o main.o main.S
```

Listing 2: Comando de ensamblaje inicial

Sin embargo, al ejecutar esto en un sistema operativo host de 64 bits (como Ubuntu `x86_64`), el ensamblador `as`, aunque genera las instrucciones correctas de 16 bits gracias a `.code16`, crea un archivo objeto ELF que internamente está marcado con la arquitectura `x86-64`. Esto causa conflictos posteriores con el linker cuando se intenta generar una salida `i386`.

Para mitigar esto, se puede intentar forzar a `as` a usar un formato más tradicional o directamente usar una Máquina Virtual Linux de 32 bits o un compilador cruzado (*cross-compiler*) `i686-elf-as`. Una opción que a veces funciona es:

```
as --traditional-format -o main.o main.S
```

Listing 3: Intento de comando de ensamblaje (puede variar)

(Nota: La solución más fiable para evitar estos problemas es usar una VM de 32 bits o un compilador cruzado.)

### 8.2. Enlace con el Linker (ld)

Una vez obtenido el archivo objeto `main.o`, se utiliza el linker `ld` junto con el script `link.ld` para generar la imagen binaria final `main.img`.

```
ld -m elf_i386 -T link.ld -o main.img main.o
```

Listing 4: Comando de enlace

Explicación de las opciones:

- `-m elf_i386`: Esencial para resolver conflictos de arquitectura. Indica a `ld` que procese la entrada y genere (internamente antes de convertir a binario) para la arquitectura `i386`. Asegura la compatibilidad con el objetivo y `OUTPUT_ARCH(i386)` del script.
- `-T link.ld`: Especifica que se deben usar las reglas definidas en el archivo `link.ld`. Este script establece la dirección base en `0x7C00`, organiza las secciones `.text`, `.data`, etc., y asegura (mediante `.org/.word` en `main.S` o directivas del linker) que la imagen final tenga 512 bytes y la firma MBR `0xAA55` en los últimos dos bytes.
- `-o main.img`: Nombre del archivo de salida.
- `main.o`: Archivo objeto de entrada.

El resultado es el archivo `main.img`, una imagen binaria cruda de 512 bytes lista para ser arrancada.

### 8.3. Verificación (objdump vs. hd)

Para visualizar el trabajo de relocación realizado por el linker, comparamos la salida del desensamblador sobre el archivo objeto con un volcado hexadecimal de la imagen binaria final.

```
objdump -d main.o
hd main.img
# o hexdump -C main.img
```

Listing 5: Comandos de verificación

Qué comparar:

- **Desensamblado (objdump):** Muestra las instrucciones máquina y sus direcciones relativas (usualmente desde 0) dentro del archivo objeto. Por ejemplo, para `movw $msg, %si`, muestra la instrucción y una dirección *relativa* para `msg` (ej., `0x1a`). Los bytes hexadecimales codificarán esta dirección relativa (ej., `be 1a 00`).
- **Volcado Hexadecimal (hd):** Muestra los bytes crudos del archivo `main.img`. Al buscar la misma secuencia de bytes para `movw $msg, %si` (empezará con `be`), se observa que los bytes de la dirección *han cambiado*. En lugar de `1a 00`, ahora se verá la dirección *absoluta* calculada por el linker usando `0x7C00` como base (ej., `1a 7c`, correspondiente a `0x7C1A` en little-endian). Esta diferencia demuestra la relocación.
- **Firma MBR:** En la salida de `hd`, se verifica que los dos últimos bytes del archivo (offsets `0x1fe` y `0x1ff`) son `55 aa`.

### 8.4. Depuración con QEMU y GDB

Finalmente, se depura la ejecución del MBR usando el emulador QEMU y el depurador GDB.

#### 8.4.1. Iniciar QEMU en modo depuración

```
qemu-system-i386 -fda main.img -boot a -s -S
```

Listing 6: Lanzar QEMU esperando a GDB

Explicación de las opciones:

- `-fda main.img`: Carga `main.img` como disquete A. (Ajustar ruta si es necesario).
- `-boot a`: Indica a QEMU arrancar desde el disquete A.
- `-s`: Inicia un servidor GDB en `localhost:1234`.
- `-S`: Congela la CPU al inicio, esperando que GDB se conecte y dé la orden de continuar.

QEMU se inicia pero queda pausado.

### 8.4.2. Sesión de GDB

Se abre otra terminal y se inicia GDB.

#### 1. Conectar a QEMU:

```
(gdb) target remote localhost:1234
```

Se establece la conexión. GDB muestra la dirección inicial (0x0000fff0) y una advertencia sobre la falta de un ejecutable especificado, lo cual es normal.

#### 2. Cargar Símbolos (Opcional pero recomendado): Para usar etiquetas del código fuente.

```
(gdb) file main.o
```

GDB podría mostrar una advertencia sobre incompatibilidad de arquitecturas entre el archivo .o (marcado como x86-64 por as) y el objetivo i386 reportado por QEMU. Sin embargo, en la práctica, a menudo GDB logra cargar los símbolos de todas formas.

#### 3. Establecer Arquitectura (¡CRUCIAL!): Indicar a GDB que interprete el código como 16 bits.

```
(gdb) set architecture i8086
```

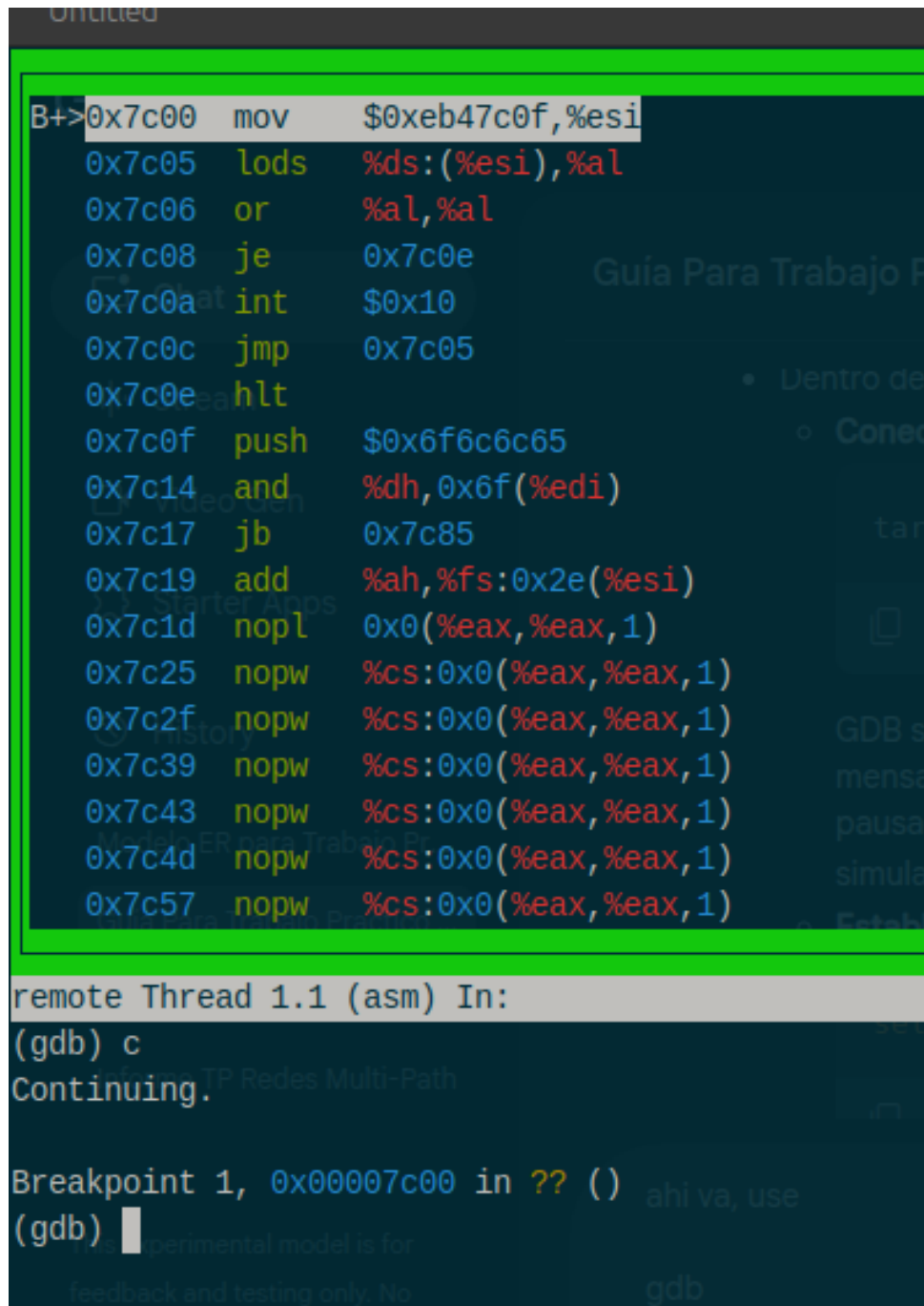
#### 4. Establecer Breakpoint Inicial: Detener la ejecución al inicio de nuestro código MBR.

```
(gdb) b *0x7c00
```

#### 5. Iniciar Ejecución: Decirle a GDB que deje correr a QEMU.

```
(gdb) c  
Continuing.
```

QEMU simula el final del BIOS, carga main.img y salta a 0x7c00, donde GDB detiene la ejecución debido al breakpoint.



The screenshot shows the GDB interface with the 'asm' layout selected. The top panel displays assembly code starting at address 0x7c00. The first instruction is 'mov \$0xeb47c0f,%esi' at 0x7c00, which is highlighted. Subsequent instructions include 'lods %ds:(%esi),%al' at 0x7c05, 'or %al,%al' at 0x7c06, 'je 0x7c0e' at 0x7c08, 'int \$0x10' at 0x7c0a, 'jmp 0x7c05' at 0x7c0c, 'hlt' at 0x7c0e, 'push \$0x6f6c6c65' at 0x7c0f, 'and %dh,0x6f(%edi)' at 0x7c14, 'jb 0x7c85' at 0x7c17, 'add %ah,%fs:0x2e(%esi)' at 0x7c19, and a series of 'nopw' instructions from 0x7c1d to 0x7c57. The bottom panel shows the GDB prompt '(gdb) c' followed by 'Continuing.' and 'Breakpoint 1, 0x00007c00 in ?? ()'. The prompt '(gdb) ' is followed by a cursor.

```
B+>0x7c00  mov  $0xeb47c0f,%esi
0x7c05  lods  %ds:(%esi),%al
0x7c06  or    %al,%al
0x7c08  je     0x7c0e
0x7c0a  int    $0x10
0x7c0c  jmp    0x7c05
0x7c0e  hlt
0x7c0f  push   $0x6f6c6c65
0x7c14  and    %dh,0x6f(%edi)
0x7c17  jb     0x7c85
0x7c19  add    %ah,%fs:0x2e(%esi)
0x7c1d  nopl   0x0(%eax,%eax,1)
0x7c25  nopw   %cs:0x0(%eax,%eax,1)
0x7c2f  nopw   %cs:0x0(%eax,%eax,1)
0x7c39  nopw   %cs:0x0(%eax,%eax,1)
0x7c43  nopw   %cs:0x0(%eax,%eax,1)
0x7c4d  nopw   %cs:0x0(%eax,%eax,1)
0x7c57  nopw   %cs:0x0(%eax,%eax,1)

remote Thread 1.1 (asm) In:
(gdb) c
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) 
```

Figura 1: GDB detenido en el breakpoint en 0x7c00. Layout 'asm' muestra el código.

6. **Ejecutar Paso a Paso y Observar:** Se utiliza si (Step Instruction) para avanzar instrucción por instrucción.
  - Con info registers (o layout regs), se observa el cambio en los registros ax, ds, es (inicialización a 0) y si (carga de la dirección de msg).

```
remote Thread 1.1 (asm) In:
eax      0xaa55      43605
ecx      0x0         0
edx      0x0         0
ebx      0x0         0
esp      0x6f08      0x6f08
ebp      0x0         0x0
esi      0x0         0
edi      0x0         0
eip      0x7c00      0x7c00
--Type <RET> for more, q to quit, c to continue without paging--
```

Figura 2: Salida parcial de ‘info registers’ al inicio.

- Se sigue el bucle `print_loop`:

```
B+>0x7c05  lods  %ds:(%esi),%al
      0x7c06  or   %al,%al
      0x7c08  je   0x7c0e
      0x7c0a  int  $0x10
      0x7c0c  jmp  0x7c05
```

Figura 3: Instrucciones del bucle de impresión en GDB.

Se observa cómo `lods` carga un carácter en `al` e incrementa `si`. Esa imagen representa el bucle donde se carga letra por letra el ‘hello world’

7. **Correr hasta el Final:** Para ver el resultado completo:

```
(gdb) b halt # Poner breakpoint en la etiqueta halt
(gdb) c      # Continuar hasta halt
```

Se observa el mensaje completo en QEMU. GDB se detiene en la instrucción `cli` en `halt`.

8. **Últimas Instrucciones:** Con `si`, se ejecuta `cli` (se verifica el flag `IF` desactivado con `info registers flags`) y luego `hlt`. Al ejecutar `hlt`, la CPU simulada se detiene. GDB ya no puede avanzar más. Se sale de GDB con `quit`.

```
QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCB050+06F0B050 CA00

Booting from Floppy...
hello world
```

Figura 4: Resultado final.

## 9. Creación y prueba de un bootloader básico

### 9.1. Objetivo

El objetivo de esta práctica es comprender el proceso de creación de un **bootloader** sencillo, capaz de ejecutarse directamente al arrancar una computadora, sin la intervención de un sistema operativo.

### 9.2. Conceptos fundamentales

Un bootloader es el primer fragmento de código ejecutado por el procesador luego del encendido. Cuando la BIOS finaliza sus rutinas de inicialización, busca un sector de arranque (*boot sector*) en los dispositivos conectados. Este sector debe cumplir las siguientes condiciones:

- Tener un tamaño de **512 bytes**.
- Terminar con la **firma mágica 0xAA55**.
- El código debe asumir que será cargado en la dirección de memoria física **0x7C00**.

### 9.3. Procedimiento realizado

#### 9.3.1. Creación del código en ensamblador

Se escribió un programa en lenguaje ensamblador NASM que imprime caracteres en pantalla utilizando servicios de la BIOS:

```
[org 0x7C00]

mov ah, 0x0E
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

hlt

times 510-($-$$) db 0
dw 0xAA55
```

Listing 7: Código del bootloader básico

**Explicación:**

- `[org 0x7C00]`: indica que el código debe pensar que se ejecuta a partir de la dirección `0x7C00`.
- `mov ah, 0x0E + int 0x10`: llama a una interrupción BIOS para imprimir un carácter en pantalla en modo texto.
- `times 510-($-$$) db 0`: rellena con ceros hasta llegar a 510 bytes.
- `dw 0xAA55`: agrega la firma mágica necesaria para que la BIOS reconozca el sector como booteable.

**9.3.2. Compilación**

Se utilizó NASM para ensamblar el código directamente a un archivo binario plano:

```
nasm -f bin main.asm -o main.img
```

Listing 8: Compilación del bootloader

**9.3.3. Prueba en máquina virtual**

Antes de grabarlo en hardware real, se utilizó **QEMU** para probar el funcionamiento de la imagen:

```
qemu-system-x86_64 -drive format=raw,file=main.img -display sdl
```

Listing 9: Ejecución en QEMU

Esto lanzó una ventana de emulación donde se pudo observar la impresión de “HELLO” en la pantalla al arrancar.

**9.3.4. Opcional: grabación en un pendrive físico**

Para probar el bootloader en hardware real, se puede grabar el archivo en un pendrive siguiendo precauciones extremas para no sobrescribir otros dispositivos:

```
sudo dd if=main.img of=/dev/sdX bs=512 count=1 status=progress
```

Listing 10: Grabación en dispositivo USB

donde `sdX` debe ser reemplazado por el identificador correcto del pendrive.

**9.4. Resultados**

Se logró exitosamente ejecutar un programa de arranque que imprime texto en pantalla, tanto en entorno virtualizado como en condiciones aptas para hardware real.



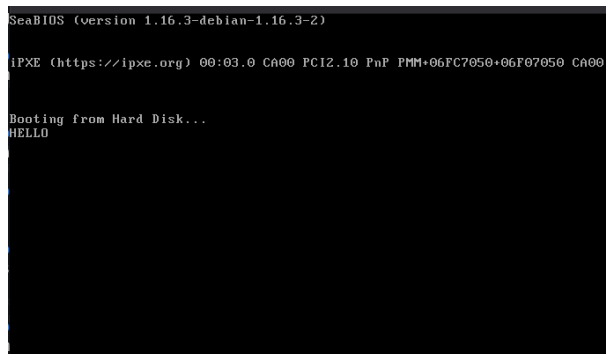


Figura 5: Bootloader mostrando “HELLO” en QEMU.

## 9.5. Conclusiones

### 9.5.1. Prueba en hardware real

Una vez comprobado el funcionamiento del bootloader en entorno virtualizado, se procedió a grabarlo en un dispositivo USB físico y probarlo en una computadora real.

Los pasos realizados fueron:

1. Conectar únicamente el pendrive para evitar confusiones con otros discos.
2. Identificar el dispositivo correcto mediante el comando:

```
lsblk
```

3. Desmontar la partición montada del pendrive:

```
sudo umount /dev/sdc1
```

4. Grabar la imagen del bootloader en el dispositivo completo:

```
sudo dd if=main.img of=/dev/sdc bs=512 count=1 status=progress
```

5. Expulsar de forma segura el pendrive:

```
sudo eject /dev/sdc
```

6. Insertar el pendrive en la computadora de prueba, acceder al *Boot Menu* durante el arranque (presionando F12, F10, ESC o la tecla correspondiente) y seleccionar el pendrive como dispositivo de arranque **en modo Legacy (no UEFI)**.

Al bootear desde el pendrive, el sistema ejecutó correctamente el bootloader, mostrando el texto programado.



Figura 6: Bootloader mostrando “HELLO” desde pendrive en hardware real.

Esta práctica permitió:

- Comprender el flujo de arranque de una PC desde nivel BIOS.
- Aplicar conocimientos de lenguaje ensamblador en un entorno de bajo nivel.
- Experimentar con herramientas de emulación de hardware.

Se consolidó así el conocimiento sobre cómo interactuar directamente con el hardware sin sistemas operativos intermedios.

## 10. Desafío Final: Implementación y Explicación

### 10.1. Crear Código Assembler para Pasar a Modo Protegido (sin macros)

El objetivo era crear un sector de arranque (MBR) que, iniciando en Modo Real de 16 bits, realizara los pasos necesarios para entrar en Modo Protegido de 32 bits utilizando las instrucciones máquina explícitas, sin depender de macros que pudieran ocultar el proceso.

#### 10.1.1. Estructura del Código y Archivos

Para una mejor organización y reutilización, el código se dividió en varios archivos fuente ensamblador (usando sintaxis NASM):

- **pm.asm**: Archivo principal que contiene el punto de entrada (`_start`), la inicialización básica, las llamadas a las funciones de impresión y transición, y las definiciones de mensajes.
- **print\_string.asm**: Contiene la función `print_string` para imprimir cadenas en Modo Real usando la interrupción 0x10 del BIOS.
- **gdt.asm**: Define la Tabla de Descriptores Globales (GDT) necesaria para el Modo Protegido, incluyendo el descriptor nulo obligatorio, un descriptor para el segmento de código y un descriptor para el segmento de datos. También define el descriptor especial que apunta a la propia GDT (requerido por la instrucción `lgdt`).
- **switch\_to\_pm.asm**: Contiene la función `switch_to_pm` que ejecuta la secuencia de transición (deshabilitar interrupciones, cargar GDT, activar bit PE en CR0, salto largo) y la etiqueta `init_pm` donde aterriza el código de 32 bits para configurar los segmentos y la pila en Modo Protegido.
- **print\_string\_pm.asm**: Contiene la función `print_string_pm` para imprimir cadenas directamente en la memoria de video (0xB8000), necesaria ya que el BIOS no está disponible en Modo Protegido.

El archivo principal `pm.asm` utiliza la directiva `%include` de NASM para incorporar el contenido de los otros archivos durante el ensamblaje.

#### 10.1.2. El Problema del Formato Binario Directo (-f bin)

Inicialmente, se intentó ensamblar directamente a un formato binario crudo usando `nasm pm.asm -f bin -o pm.img`. Sin embargo, este enfoque presentó serios problemas al depurar con GDB:

- **Desensamblado Incorrecto**: GDB mostraba secuencias de instrucciones inválidas ((bad)) o incorrectas en direcciones donde se esperaba código válido (especialmente después de los `call` o cerca de los límites de los archivos incluidos).
- **Fallo Prematuro (Triple Fallo)**: La ejecución en QEMU fallaba catastróficamente (probablemente por un Triple Fallo) justo al intentar realizar el salto largo (`ljmp`) a Modo Protegido, indicando que la CPU no podía encontrar o validar el descriptor de código especificado o la dirección de destino.

La causa raíz es que NASM, al generar un binario crudo (`-f bin`) a partir de múltiples archivos incluidos y con cambios de modo (`[bits 16]`, `[bits 32]`), no tiene suficiente información contextual para calcular y colocar correctamente todos los offsets, direcciones y bloques de código/datos relativos a la dirección de carga final (`0x7C00`) y resolver referencias entre los diferentes «módulos» incluidos. El resultado era un archivo `pm.img` corrupto o mal estructurado.

### 10.1.3. Solución: Flujo de Trabajo ELF

Para solucionar estos problemas, se adoptó un flujo de trabajo más robusto y estándar utilizando un formato intermedio:

1. **Ensamblar a ELF:** Se ensambló el archivo principal `pm.asm` (que incluye a los demás) a un formato de objeto ELF de 32 bits usando `nasm -f elf pm.asm -o pm.o`. ELF (Executable and Linkable Format) es un formato que preserva la información sobre secciones (`.text`, `.data`), símbolos (etiquetas como `_start`, `print_string`, `gdt_start`), y datos de relocalización necesarios para el linker.
2. **Enlazar con LD y Script:** Se utilizó el linker GNU `ld` con la opción `-m elf_i386` (para asegurar compatibilidad con el objetivo i386) y un script de linker (`link.ld`). Este script instruye a `ld` para:
  - Establecer la dirección de carga base en `0x7C00`.
  - Combinar las secciones `.text`, `.data`, etc., del archivo `pm.o` en una única sección de salida posicionada en `0x7C00`.
  - Resolver todas las direcciones (como la dirección de `gdt_start` usada por `lgdt`, la dirección de `MSG_REAL_MODE` usada por `mov si`, y las direcciones de las funciones `print_string` y `switch_to_pm` usadas por `call`) relativas a la base `0x7C00`.
  - Asegurar que el archivo de salida tenga exactamente 512 bytes, añadiendo relleno si es necesario y la firma MBR `0xAA55` al final (usando `. = 0x7C00 + 510; BYTE(0x55) BYTE(0xAA);`).

Comando: `ld -m elf_i386 -T linker.ld -o pm.elf pm.o`

Este enfoque delega la compleja tarea de posicionamiento y resolución de direcciones al linker (`ld`), que está diseñado específicamente para ello, resultando en un archivo `pm.img` correctamente estructurado y funcional.

### 10.1.4. Código de Transición Detallado (en `switch_to_pm.asm`)

El núcleo de la transición reside en la función `switch_to_pm`, que realiza los siguientes pasos indispensables (sin macros):

```

1  [bits 16]
2  switch_to_pm:
3  cli                                ; 1. Deshabilitar interrupciones
   enmascarables.
4
   ; para evitar que una IT ocurra
   durante la transicion
5   ; cuando el estado de la CPU es
   inconsistente.
6
7  lgdt [gdt_descriptor]             ; 2. Cargar el Registro Descriptor de
   la GDT (GDTR).
8
   ; Lee 6 bytes desde la direccion de
   la etiqueta 'gdt_descriptor'.
9   ; Esos 6 bytes contienen el limite (
   tamaño-1) de la GDT
10  ; y la direccion base lineal donde
   reside la GDT en memoria.
11  ; Esto informa a la CPU donde
   encontrar la tabla GDT.
12
13  ; 3. Activar Modo Protegido (Bit PE en CR0)
14  mov eax, cr0                     ; 3a. Leer el registro de control CR0
   en EAX (32 bits).
15  or  eax, 0x01                    ; 3b. Poner el bit 0 (PE - Protection
   Enable) a 1 usando OR.
16
   ; Los otros bits de CR0 se
   mantienen.
17  mov cr0, eax                     ; 3c. Escribir el valor modificado de
   vuelta a CR0.
18
   ; A partir de este instante, la CPU
   opera en Modo Protegido.
19
20  ; 4. Salto Largo (Far Jump)
21  ; Es OBLIGATORIO inmediatamente despues de activar PE. Sirve
   para:
22  ; a) Limpiar la cola de prefetch de instrucciones de la CPU, que
   podria
23  ; contener instrucciones decodificadas como si fueran de Modo
   Real.
24  ; b) Cargar el registro de segmento deCodigo (CS) con un
   SELECTOR valido
25  ; de nuestra nueva GDT.
26  ; c) Transferir la ejecucion al codigo disenado para 32 bits.
27  jmp 0x08 : init_pm               ; Salto a: Selector deCodigo (0x08,
   segundo descriptor)
28
   ; Offset (direccion de la
   etiqueta 'init_pm')
29
30  [bits 32]                         ; A partir de aqui, el ensamblador
   genera codigo de 32 bits
31  init_pm:                          ; Punto de aterrizaje en Modo Protegido

```

```

32 ; 5. Configurar Segmentos de Datos y Pila (32 bits)
33 mov ax, 0x10          ; Cargar SELECTOR del segmento de Datos
    (0x10) en AX.
34
    ; Se usa AX (16 bits) porque los
    selectores son de 16 bits.
35 mov ds, ax          ; Cargar DS con el selector de datos.
36 mov ss, ax          ; Cargar SS (segmento de pila) con el
    mismo selector.
37 mov es, ax          ; Cargar ES.
38 mov fs, ax          ; Cargar FS (opcional).
39 mov gs, ax          ; Cargar GS (opcional).
40 ; Ahora todos los accesos a datos o pila usaran el descriptor GDT
    en el indice 2.
41
42 ; 6. Configurar el Puntero de Pila (ESP)
43 mov esp, 0x90000      ; Establecer ESP (32 bits) a una
    direccion segura y valida
44
    ; (ej. 0x90000), ubicada dentro de los
    limites del
45
    ; segmento de datos/pila cargado en SS.
46
47 ; 7. Continuar con la ejecucion en Modo Protegido
48 call BEGIN_PM          ; Llamar a la rutina principal de PM (
    en pm.asm)

```

Listing 11: switch\_to\_pm.asm (Nucleo de la transicion)

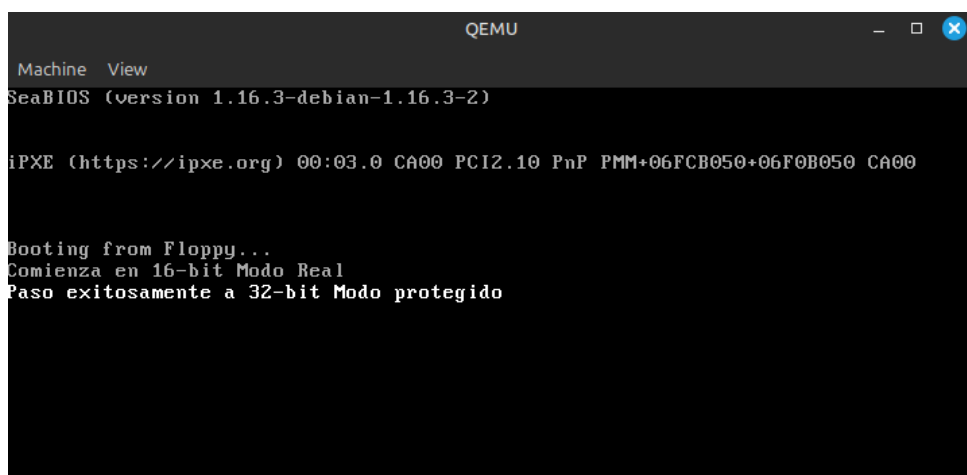


Figura 7: Cambio de modo real a modo protegido funcionando.

## 10.2. Programa con Descriptores de Código y Datos Diferenciados

El Modo Protegido requiere definir explícitamente los segmentos de memoria que el programa utilizará a través de la GDT. Nuestro programa utiliza los descriptores mínimos necesarios para un funcionamiento básico:

- **Descriptor Nulo (Índice 0):** Es obligatorio por la especificación Intel. Todos sus

campos deben ser cero. Intentar cargar un selector que apunte a este descriptor causa una excepción.

- **Descriptor de Código (Índice 1, Selector 0x08):** Define las propiedades del segmento donde reside nuestro código ejecutable.
  - **Base** = 0x00000000: El segmento empieza al inicio de la memoria física.
  - **Límite** = 0xFFFFF (con Granularidad=1): El segmento se extiende por 4 GiB  $((0xFFFF + 1) \times 4096)$ . Cubre todo el espacio de direcciones direccionable en 32 bits. Esto se conoce como modelo «plano» (flat model).
  - **Acceso** = 0x9A: Indica que es un segmento Presente (P=1), de nivel de privilegio 0 (DPL=00, el más alto, Ring 0), de tipo Código (S=1, Type=1010), que es Ejecutable y Leíble (R=1).
  - **Flags** = 0xCF: Indica Granularidad de 4KiB (G=1), tamaño de operando/dirección de 32 bits (D=1), y los bits altos del límite.

El registro CS se carga con el selector 0x08 mediante el `ljmp`.

- **Descriptor de Datos (Índice 2, Selector 0x10):** Define las propiedades del segmento para datos y la pila.
  - **Base** = 0x00000000: Igual que el código, empieza al inicio de la memoria.
  - **Límite** = 0xFFFFF (con Granularidad=1): Igual, 4 GiB. Modelo plano.
  - **Acceso** = 0x92: Indica Presente (P=1), DPL=00, de tipo Datos (S=1, Type=0010), que es Leíble y Escribible (W=1).
  - **Flags** = 0xCF: Idéntico al segmento de código (32 bits, granularidad 4K).

Los registros DS, ES, FS, GS, SS se cargan con el selector 0x10 después de entrar en modo protegido.

### 10.2.1. Espacios de Memoria Diferenciados (Concepto vs. Implementación Plana)

La pregunta pide específicamente segmentos en «espacios de memoria diferenciados». Con los descriptors definidos arriba (modelo plano), tanto el código como los datos *pueden* acceder teóricamente al mismo espacio lineal completo de 4 GiB porque ambos tienen Base=0 y Límite=4 GiB.

Para tener espacios verdaderamente *diferenciados* y separados por hardware (más allá de solo los permisos R/W/X), necesitaríamos definir descriptors con **bases y/o límites diferentes**. Por ejemplo:

- **Código Separado:** Se podría definir el descriptor de código con Base=0x100000 y Límite=0xFFFFF (cubriendo desde 1 MiB hasta 4 GiB+1 MiB con granularidad), mientras que el de datos tiene Base=0 y Límite=0x1FFFF (cubriendo los primeros 128 KiB con granularidad de byte para la pila y datos iniciales).
- **Datos Separados:** Podríamos tener Código Base=0, Límite=Y, y Datos Base=Y+1, Límite=Z.

Esto requeriría:

1. Modificar las definiciones `dw base_low`, `db base_mid`, `db base_high` y `dw limit_low`, `db flags_limit_high` en `gdt.asm` para cada descriptor.
2. Asegurarse de que el linker coloque el código y los datos en las regiones de memoria física correspondientes a esas bases definidas en los descriptors. Esto requeriría un script `link.ld` más complejo, definiendo secciones `.text` y `.data` con direcciones de carga (AT) diferentes y especificando sus direcciones virtuales (`. = direccion_virtual`) correspondientes a las bases de los descriptors.

Sin embargo, para la funcionalidad básica de este TP, el **modelo plano** (**Base=0, Límite=4 GB para ambos**) es mucho más simple de implementar y es el modelo predominante usado por los sistemas operativos modernos (que luego utilizan paginación para la separación y protección fina). Nuestro código actual, aunque no usa bases/límites diferentes, sí tiene descriptors funcionalmente distintos para código (Ejecutable/Leíble) y datos (Leíble/Escribible).

## 11. Prueba de Protección de Memoria (Read-Only)

Una de las características fundamentales introducidas por el Modo Protegido es la capacidad del hardware para hacer cumplir permisos de acceso a los segmentos de memoria definidos en la GDT. Para verificar esto experimentalmente, modificamos nuestro bootloader para intentar escribir en un segmento de datos configurado como "solo lectura".

### 11.1. Modificaciones al Código

1. **Definición de Descriptor Read-Only en `gdt.asm`:** Se añadió un tercer descriptor de datos a la GDT (después del descriptor de código y el de datos R/W). Este nuevo descriptor, etiquetado como `gdt_data_ro`, se configuró con los mismos Base (0) y Límite (4 GiB) que los demás, pero con un **Byte de Acceso** específico: `0x90`.

```

; Descriptor 3: Datos Read-Only (Selector 0x18)
gdt_data_ro:
    dw 0xFFFF                ; Limite (0-15)
    dw 0x0000                ; Base (0-15)
    db 0x00                  ; Base (16-23)
    db 0x90                  ; <<< Byte de Acceso
                           (10010000b) >>>
                                   ; P=1, DPL=0, S=1(
                                   ;   Datos), Type=0000
                                   ;   -> Writable=0
    db 0xCF                  ; Flags (G=1, D=1 ->
                           32 bit) + Limit(16-19)
    db 0x00                  ; Base (24-31)

gdt_end:                      ; Etiqueta movida al
                           final

```

Listing 12: Descriptor Read-Only en `gdt.asm`



El byte de acceso 0x90 (10010000b) indica explícitamente que el segmento está presente (P = 1), es para Ring 0 (DPL = 00), es un segmento de datos (S = 1) y, crucialmente, el bit **Writable (W)** es 0, marcándolo como solo lectura. La etiqueta `gdt_end` se movió para asegurar que el tamaño de la GDT (usado por `lgdt`) incluyera este nuevo descriptor. El selector correspondiente a este tercer descriptor (índice 3) es 0x18.

2. **Intento de Escritura en Modo Protegido (en `switch_to_pm.asm`):** Dentro de la rutina `BEGIN_PM` (que se ejecuta en modo protegido), después de imprimir el mensaje de bienvenida y verificar opcionalmente una escritura válida en el segmento R/W, se añadió el siguiente bloque para probar la protección:

```

; Prueba con segmento R/O (Selector 0x18)
mov ax, 0x18                ; Cargar Selector Datos
    ↪ Read-Only
mov ds, ax                  ; Actualizar DS para
    ↪ usar el segmento R/O

mov dword [0x3000], 0xDEADBEEF ; usando el segmento
    ↪ definido por DS (ahora R/O)
                                ; ESTO DEBE FALLAR.

;
.write_protection_failed:
;
mov edi, 0xb8000 + 320      ;
mov ax, 0x4f45              ; 'E' roja
mov [edi], ax
mov ax, 0x4f52              ; 'R' roja
mov [edi+2], ax
mov ax, 0x4f52              ; 'R' roja
mov [edi+4], ax
.hang_error:
cli
hlt
jmp .hang_error

```

Listing 13: Intento de escritura R/O en `switch_to_pm.asm`

Este código primero carga el selector 0x18 en el registro de segmento DS. Luego intenta escribir el valor 0xDEADBEEF en la dirección de memoria 0x3000 utilizando implícitamente el segmento apuntado por DS. Se añadió también una sección `.write_protection_failed` que solo se alcanzaría si, por error, la protección no se activara.

## 11.2. Ejecución y Resultados: ¿Qué Sucede?

Tras recompilar y enlazar el código con las modificaciones, se ejecutó la imagen `pm_final.img` resultante. El comportamiento observado dependió del entorno de ejecución:

- **En QEMU (modo TCG por defecto):**

Al ejecutar `qemu-system-i386 -fda pm_final.img -boot a` se observó lo siguiente:

```
Booting from Floppy...
Comienza en 16-bit Modo Real
Paso exitosamente a 32-bit Modo protegido
ERR
```

La aparición del mensaje **ERR** indica que la instrucción `mov dword [0x3000], 0xDEADBEEF` *no generó la excepción esperada*; la ejecución alcanzó el manejador de error `.write_protection_fault`. Esto se debe a una limitación de QEMU en modo TCG, que no realiza todas las comprobaciones de permisos R/W de los descriptores de segmento.

- **En Bochs (emulador preciso):**

Al lanzar la misma imagen con `bochs -q` (usando un `.bochsrc` adecuado) el comportamiento cambió:

- (a) Se mostraron los mensajes iniciales.
- (b) La emulación se detuvo de inmediato.
- (c) El log `bochsout.txt` registró:

```
... (estado de registros justo antes del fallo) ...
00XXX[CPU0 ] exception(): 3rd (13) exception with no resolution,
                shutdown status is 00h, resetting
... (mensajes de reinicio) ...
```

Bochs reportó la **excepción 13** (GP – General Protection Fault); al no haber IDT pasó a triple falta y reinició la VM.

- **En QEMU con KVM:**

Ejecutando `qemu-system-i386 -enable-kvm -fda pm_final.img -boot a` se ven los dos mensajes iniciales y QEMU se cierra abruptamente: de nuevo se trata de la triple falta, que KVM/QEMU resuelve terminando la emulación.

**Conclusión del Experimento:** La ejecución en Bochs y QEMU con KVM **verifica exitosamente** que la CPU (real o emulada con precisión) detecta el intento de escritura en un segmento marcado como Read-Only (W=0 en el descriptor) y genera la excepción de Falla de Protección General (GP) como se espera según la arquitectura x86.

### 11.3. ¿Qué Debería Suceder a Continuación (Teoría del Fallo)?

Cuando la CPU ejecuta una instrucción que viola las reglas de protección definidas por los descriptores de segmento (como intentar escribir en un segmento R/O), ocurre la siguiente secuencia teórica:

1. **Generación de Excepción GP (Vector 13):** La CPU detiene la ejecución de la instrucción infractora y genera una Falla de Protección General. Busca en la Tabla de Descriptores de Interrupción (IDT) la entrada correspondiente al vector 13.
2. **Intento de Invocar al Manejador GP:** La CPU intenta transferir el control al manejador de excepciones definido en la entrada 13 de la IDT.
3. **Fallo por Falta de IDT/Manejador (Doble Falta):** En nuestro bootloader, no hemos configurado ni cargado una IDT para el Modo Protegido usando la instrucción `lidt`. La CPU no encuentra un descriptor válido para el vector 13. Este fallo al manejar una excepción genera una nueva excepción: la **Doble Falta (Excepción DF - Vector 8)**.
4. **Intento de Invocar al Manejador DF:** La CPU intenta transferir el control al manejador de Doble Falta definido en la entrada 8 de la IDT.
5. **Fallo por Falta de Manejador DF (Triple Falta):** Como sigue sin haber una IDT válida, la CPU falla de nuevo al intentar manejar la Doble Falta. Una excepción ocurrida mientras se intenta invocar al manejador de Doble Falta resulta en una condición irre recuperable conocida como **Triple Falta**.
6. **Shutdown (Apagado/Reinicio):** Ante una Triple Falta, la CPU entra en un estado de "shutdown". En hardware real, esto generalmente provoca un reinicio inmediato de la placa base. Los emuladores como Bochs y QEMU+KVM simulan este comportamiento deteniendo la emulación o reportando el fallo fatal.

Por lo tanto, el resultado esperado al intentar escribir en el segmento R/O en nuestro contexto (sin IDT) es, efectivamente, un Triple Fallo que detiene o reinicia el sistema.

### 11.4. Verificación con GDB

Aunque QEMU en modo TCG no genera el GP, GDB sigue siendo útil para verificar que los prerequisites para el fallo estaban presentes:

1. Se conectó GDB a QEMU ('target remote').
2. Se cargó el archivo ELF ('file pm.elf') para obtener símbolos (aunque GDB mostró advertencias, pudo usarlos parcialmente).
3. Se establecieron las arquitecturas correctas ('set architecture i8086' / 'set architecture i386').
4. Se verificó la GDT en memoria antes de 'lgdt' usando 'x/32bx 'direccion\_gdt\_start'', confirmando que el tercer descriptor tenía el byte de acceso 0x90.
5. Se avanzó la ejecución hasta justo antes de la instrucción `mov dword [0x3000], 0xDEADBEEF`.

6. Se verificó con `info registers ds` que el registro DS contenía correctamente el selector 0x18.
7. Al ejecutar `si` sobre la instrucción de escritura, se observó que GDB avanzaba a la siguiente instrucción (llevando a `.ERR`) en lugar de reportar `SIGSEGV / General Protection Fault`.

Esta depuración confirma que el código cargó el selector correcto para el segmento R/O, pero la excepción no fue generada por el entorno de emulación TCG. Si se hubiera depurado bajo Bochs o KVM, el último `si` habría mostrado la recepción de la señal SIGSEGV.

## 11.5. Valor y Propósito de los Selectores en Modo Protegido

**Pregunta:** En modo protegido, ¿Con qué valor se cargan los registros de segmento? ¿Porque?

En Modo Protegido, los registros de segmento (CS, DS, ES, SS, FS, GS) **NO** se cargan con direcciones base como en Modo Real. En su lugar, se cargan con valores de 16 bits llamados **Selectores**. En nuestro ejemplo, DS, ES, SS, etc., se cargaron con el selector 0x10 para el segmento de datos R/W, y luego DS se cambió al selector 0x18 para el segmento R/O.

La estructura de un selector es:

- **Bits 15-3 (13 bits): Índice.** Especifica el número de entrada (descriptor) a utilizar dentro de una tabla de descriptors.
- **Bit 2 (TI - Table Indicator):** Indica si se usa la GDT (TI=0) o la LDT (TI=1). Nuestros selectores 0x08, 0x10, 0x18 tienen TI=0.
- **Bits 1-0 (RPL - Requested Privilege Level):** Indica el nivel de privilegio solicitado para la operación. Generalmente es 0 para código de kernel. Nuestros selectores tienen RPL=0.

Por ejemplo, el selector 0x18 (binario 0000000000011000) tiene: Índice = 3, TI = 0 (GDT), RPL = 0.

**¿Por qué se usan Selectores en lugar de direcciones base?**

Este mecanismo de indirección es fundamental para las características clave del Modo Protegido:

1. **Protección de Memoria:** Cuando se realiza un acceso a memoria (ej. leer de `DS:[offset]`), la CPU no usa directamente el valor en DS. Utiliza el *selector* en DS para buscar el *descriptor* correspondiente en la GDT. Este descriptor contiene la dirección base real, el límite del segmento y los bits de permiso (R/W/X, DPL). La CPU realiza comprobaciones automáticas por hardware: ¿Está el `offset` dentro del `límite`? ¿Permite el descriptor la operación (lectura/escritura)? ¿Tiene el código actual (CPL) y el selector (RPL) suficientes privilegios para acceder a un segmento con este DPL? Si alguna comprobación falla, se genera una excepción (GP, SS). Esto impide que un programa acceda o corrompa memoria fuera de sus segmentos asignados o con permisos insuficientes.
2. **Niveles de Privilegio (Rings):** Los campos DPL (en el descriptor) y CPL/RPL (en CS/selector) permiten implementar anillos de protección, donde el código del sistema operativo (Ring 0) está protegido de las aplicaciones de usuario (Ring 3).

3. **Flexibilidad y Memoria Virtual:** Permite al sistema operativo reubicar segmentos en la memoria física simplemente actualizando sus descriptors en la GDT, sin necesidad de cambiar el código que usa los selectores. También se integra con el sistema de paginación para implementar memoria virtual compleja.

En esencia, los selectores y descriptors trasladan la responsabilidad de la gestión y protección de la memoria del software (como en Modo Real) al hardware de la CPU, proporcionando una base mucho más robusta y segura para los sistemas operativos modernos.

## 12. Conclusiones

El presente trabajo práctico permitió recorrer desde los conceptos teóricos de la arquitectura **UEFI** hasta la implementación y prueba de un *bootloader* propio, revelando la interacción detallada entre firmware, herramientas de *toolchain* y hardware.

- **UEFI como reemplazo de BIOS.** Comprendimos que UEFI no es una simple evolución sino una plataforma firmware escrita en C, capaz de operar en 32 / 64 bits, con servicios de arranque y ejecución en tiempo de *runtime*, soporte GPT, *Secure Boot* y controladores integrados. Aprender a invocar sus tablas (**BootServices** y **RuntimeServices**) habilita el desarrollo de utilidades de muy bajo nivel que conviven con el sistema operativo.
- **Superficie de ataque y riesgos.** Casos como *LoJax*, *ThinkPwn* o *LogoFAIL* evidencian que vulnerar el firmware permite *bootkits* persistentes aun tras reinstalar el SO. La seguridad del arranque exige tanto configuraciones correctas (protección de la flash) como la aplicación oportuna de parches.
- **CSME / ME y MEBx.** Las funciones de gestión remota (AMT) y arranque verificado (Boot Guard) ofrecen ventajas, pero el motor autónomo de Intel sigue siendo una *caja negra* con fallos críticos publicados. Conocer MEBx resulta clave para equilibrar administración y exposición al riesgo.
- **coreboot: firmware libre.** coreboot demuestra que es posible reemplazar firmware propietario por una solución mínima y auditada, con arranques más veloces y menor superficie de ataque, manteniendo la flexibilidad gracias a los *payloads* (SeaBIOS, Tianocore, LinuxBoot, etc.).
- **Herramientas de toolchain.** Revisar el *linker* y la convención histórica de 0x7C00 mostró que los scripts de enlace son tan cruciales como el código; si el *linker* desconoce la dirección final, los desplazamientos fallan y el arranque se aborta.
- **Bootloader “HELLO”.** Desde ensamblar 512 bytes y firmarlos con 0xAA55, hasta probar en QEMU y grabar en un pendrive, cerramos el ciclo completo: escribir, enlazar, testear y ejecutar código que arranca sin sistema operativo.

Además, el desafío final permitió poner en práctica conceptos clave de la transición al Modo Protegido:

- **Transición Manual a Modo Protegido.** El diseño de un bootloader completamente manual, sin macros ni abstracciones, obligó a comprender en profundidad el

flujo de activación del PE (Protection Enable), la importancia del salto largo posterior (`ljmp`), y la configuración explícita de los registros de segmento y la pila en 32 bits.

- **Uso Correcto de ELF y Linker Scripts.** La necesidad de ensamblar primero a ELF y luego enlazar usando un `link.ld` a `0x7C00` destacó el rol crítico del linker en sistemas bare-metal, resolviendo problemas típicos de formatos binarios directos (`-f bin`) como desincronizaciones y referencias incorrectas.
- **Segmentación y Protección.** Implementar descriptores diferenciados de código, datos R/W y datos R/O permitió experimentar directamente con la protección por hardware: definir un segmento de sólo lectura y verificar mediante ejecución que la CPU detecta y bloquea accesos inválidos.
- **Verificación de Protección por Hardware.** El intento de escribir en un segmento R/O, seguido de la observación de excepciones de Protección General (GP) y triple fallo en emuladores precisos como Bochs y QEMU-KVM, demostró empíricamente cómo el hardware asegura el cumplimiento de permisos segmentados, incluso antes de introducir mecanismos de paginación.

Estos ejercicios finales consolidaron no sólo la teoría del modelo de segmentación x86 y del proceso de arranque, sino también la práctica detallada de debugging en entornos reales (GDB, QEMU, Bochs), el entendimiento del rol de los selectores en modo protegido y la importancia de diseñar firmwares mínimos pero seguros.