

# Sistemas de Computación



## TP 3

Modo protegido

Sistemas de computación - 2025

---

### Integrantes

Franco Gabriel Lopez, 43271762

Kevin Bejarano, 42814288

Elly Indra Esparza, 39437553

### Profesor adjunto

Ing. Javier Alejandro Jorge

# Índice

<b>1. Marco Teorico</b>	<b>1</b>
1.1. ¿Qué es UEFI y cómo se usa? . . . . .	1
1.1.1. CSME e Intel MEBx . . . . .	2
1.2. ¿Qué es Coreboot? . . . . .	2
1.2.1. ¿Qué productos lo incorporan? . . . . .	3
1.3. Ventajas de la utilización de Coreboot . . . . .	3
1.4. ¿Qué es un <i>Linker</i> ? ¿Qué hace? . . . . .	4
<b>2. Modo real</b>	<b>4</b>
2.1. Scripts utilizados . . . . .	4
2.1.1. Script <code>link.ld</code> . . . . .	4
2.1.2. Código <code>main.S</code> . . . . .	5
2.2. ¿Qué es la dirección que aparece en el script del linker? ¿Por qué es necesaria? . . . . .	5
2.3. Objdump y Hexdump . . . . .	6
2.3.1. Objdump . . . . .	6
2.3.2. Hexdump . . . . .	6
2.4. Pasos para la ejecución . . . . .	6
2.5. Tamaño y contenido de <code>main.img</code> . . . . .	8
2.5.1. Sección <code>.text</code> . . . . .	9
2.6. Depuración con GDB . . . . .	9
2.7. Conexión de GDB al servidor de QEMU . . . . .	10
2.8. Inspección del código ensamblador y breakpoints . . . . .	10
<b>3. Modo protegido</b>	<b>12</b>
3.1. Modo Protegido y la Tabla GDT . . . . .	12
3.1.1. Atributos del descriptor . . . . .	13
3.2. Habilitar el modo protegido . . . . .	14
3.3. Configuración de registros de segmento . . . . .	14
3.4. Ejemplo: Hello World en Modo Protegido . . . . .	15
3.5. Compilación y Ejecución . . . . .	15
3.6. Cambios en los Permisos del Segmento de Datos . . . . .	15
3.6.1. Efecto del Cambio . . . . .	16
3.6.2. Depuración con GDB . . . . .	16
3.6.3. Conclusión . . . . .	16

## 1. Marco Teorico

### 1.1. ¿Qué es UEFI y cómo se usa?

La **Interfaz de Firmware Extensible Unificada** o *UEFI* (del inglés *Unified Extensible Firmware Interface*) es una especificación que define una interfaz entre el sistema operativo y el firmware. UEFI reemplaza la antigua interfaz del Sistema Básico de Entrada y Salida (BIOS) estándar, presentada en los PC IBM como *ROM BIOS de PC IBM*.

Se utiliza principalmente accediendo a su menú de configuración durante el arranque del sistema (generalmente presionando teclas como F2, F10, F12, **Supr** o **Esc**, dependiendo del fabricante). Desde allí, se pueden modificar opciones como:

- El orden de arranque.
- Configuraciones de seguridad.
- Habilitar o deshabilitar componentes de hardware.
- Actualizar el propio firmware UEFI.
- Configurar parámetros de rendimiento o perfiles de memoria RAM.

### Casos de *bugs* de UEFI

Investigadores de **ESET** han descubierto una vulnerabilidad que permite eludir el *UEFI Secure Boot*, afectando a la mayoría de los sistemas basados en UEFI. Esta vulnerabilidad, identificada como **CVE-2024-7344**, se encuentra en una aplicación UEFI firmada por el certificado de terceros de Microsoft (*Microsoft Corporation UEFI CA 2011*).

La explotación de esta vulnerabilidad permite la ejecución de código no confiable durante el arranque del sistema, posibilitando a los atacantes desplegar fácilmente *bootkits* maliciosos como *Bootkitty* o *BlackLotus*, incluso en sistemas con *UEFI Secure Boot* habilitado.

La aplicación UEFI vulnerable se incluye en diversas suites de recuperación desarrolladas por:

- Howyar Technologies Inc.
- Greenware Technologies.
- Radix Technologies Ltd.
- SANFONG Inc.
- Wasay Software Technology Inc.
- Computer Education System Inc.
- Signal Computer GmbH.

La causa de esta vulnerabilidad es el uso de un cargador PE personalizado en lugar de las funciones estándar y seguras *LoadImage* y *StartImage* de UEFI. Esto permite cargar cualquier binario UEFI (incluso no firmado) desde un archivo especialmente diseñado llamado *cloak.dat* durante el arranque, independientemente del estado del arranque seguro.

### 1.1.1. CSME e Intel MEBx

El **Converged Security and Management Engine (CSME)** es un subsistema crítico integrado en los chipsets de Intel que proporciona capacidades de seguridad y gestión. Es el núcleo de la tecnología **Intel® Active Management Technology (Intel® AMT)**, permitiendo la gestión remota de sistemas incluso cuando están apagados o el sistema operativo no funciona.

CSME opera de forma independiente del procesador principal, utilizando su propio firmware y memoria, siendo así una parte esencial de la arquitectura *Intel vPro®*. Entre sus funciones se incluyen:

- Acceso remoto seguro.
- Comunicación *out-of-band*.
- Resolución de problemas en sistemas fuera de línea.

La **Intel Management Engine BIOS Extension (MEBx)** es una extensión de la BIOS que permite configurar el *Management Engine*, incluyendo opciones relacionadas con AMT. Se accede usualmente presionando **Ctrl+P** durante el arranque, accediendo así a una interfaz donde se pueden:

- Establecer configuraciones de red.
- Configurar contraseñas del CSME.
- Habilitar funcionalidades de AMT.

### 1.2. ¿Qué es Coreboot?

**Coreboot**, anteriormente conocido como *LinuxBIOS*, es un proyecto iniciado en 1999 en el *Advanced Computing Laboratory* de *Los Alamos National Laboratory*, cuyo objetivo es crear un firmware que arranque rápidamente y maneje errores de manera inteligente.

Su diseño filosófico se basa en realizar solo las tareas mínimas necesarias para inicializar el hardware (como configurar la memoria, CPU y periféricos), y luego entregar el control a un *payload*, que puede ser:

- **SeaBIOS** (servicios BIOS tradicionales),
- **EDK2** (para UEFI),
- **GRUB2** (usado por muchos distros Linux),
- **Depthcharge** (común en Chromebooks).

Esta separación maximiza la reutilización de rutinas de inicialización y permite adaptarse a múltiples escenarios, desde entornos estándar hasta configuraciones altamente personalizadas.

Coreboot está licenciado bajo la **GNU GPL v2** y es compatible con múltiples arquitecturas: **IA-32, x86-64, ARM, ARM64, MIPS y RISC-V**, incluyendo plataformas como AMD Geode, Intel Atom y Ryzen Embedded.

### 1.2.1. ¿Qué productos lo incorporan?

Coreboot se emplea en una amplia variedad de dispositivos, especialmente en notebooks y PCs dirigidas a usuarios que priorizan la libertad del software y la seguridad.

A continuación, se detallan productos y fabricantes que los utilizan, basados en fuentes oficiales y foros especializados.

Fabricante	Productos	Información
Google	Chromebooks (todos menos los tres primeros modelos)	Mayor despliegue de coreboot, usado en dispositivos Chrome OS.
Purism	Notebooks Librem (como Librem 13 v2, Librem 15 v3)	Enfocados en privacidad, con coreboot preinstalado y soporte para PureBoot.
Minifree Ltd	Notebooks y desktops con Libreboot	Ofrecen Libreboot preinstalado, junto con Debian Linux u otros BSD.
Lenovo Thinkpad	Modelos como T480, X220, X230	Compatible con coreboot, especialmente en distribuciones como Libreboot y Skulls.
StarLabs Systems	Notebooks	Ofrecen por defecto la alternativa de coreboot.

Cuadro 1: Fabricantes que ofrecen dispositivos con coreboot o Libreboot

### 1.3. Ventajas de la utilización de Coreboot

El uso de **Coreboot** ofrece múltiples beneficios, especialmente para usuarios técnicos, entusiastas del software libre y organizaciones que buscan mayor control y seguridad en sus sistemas. A continuación, se enumeran sus principales ventajas:

- **Rendimiento y eficiencia:** Coreboot está diseñado para realizar únicamente las tareas esenciales de inicialización. Por ejemplo, la versión para x86 ejecuta solo diez instrucciones antes de pasar al modo de 32 bits, similar a los sistemas UEFI modernos, lo que reduce considerablemente los tiempos de arranque.
- **Seguridad y transparencia:** Coreboot está escrito mayormente en lenguaje C (menos del 1 % en ensamblador), lo que facilita su auditoría y reduce la superficie de ataque. Al ser de código abierto, permite que usuarios y desarrolladores revisen, modifiquen y mejoren el firmware, incrementando la transparencia y facilitando la detección de vulnerabilidades.
- **Flexibilidad y personalización:** Soporta múltiples *payloads* como:
  - SeaBIOS para arranque tradicional (legacy),
  - iPXE para arranque por red,
  - EDK2 para entornos UEFI.

También permite personalizar la pantalla de arranque (por ejemplo, con imágenes JPG), utilizar consolas de depuración accesibles vía puertos seriales, USB, SPI o incluso el altavoz del sistema, y recuperar registros del arranque una vez cargado el sistema operativo.

- **Reutilización y mantenimiento:** Gracias a la separación de responsabilidades entre *Coreboot* y el *payload*, se facilita la reutilización de rutinas de inicialización de hardware. Todo el proyecto se mantiene en un único árbol de código fuente, evitando la fragmentación y facilitando el mantenimiento.

## 1.4. ¿Qué es un *Linker*? ¿Qué hace?

Un **linker** o *editor de enlaces* es una herramienta esencial en el proceso de construcción de un programa ejecutable. Su propósito es tomar uno o más archivos objeto —generados por la compilación de código fuente en C, C++ o ensamblador— y combinarlos en un único archivo ejecutable, biblioteca, o archivo objeto compuesto.

Las funciones principales del linker incluyen:

- **Resolución de referencias externas:** Une llamadas a funciones o referencias a variables que están definidas en otros archivos objeto. Por ejemplo, si un archivo A necesita usar una función definida en B, el linker resuelve esta dependencia.
- **Relocalización:** Ajusta las direcciones internas para reflejar las posiciones reales de cada sección del código en memoria, asegurando que los datos e instrucciones se ubiquen correctamente.
- **Generación de archivos ejecutables:** Produce un archivo final que puede ser cargado y ejecutado por el sistema operativo, o, como en el caso de un sistema embebido o bootloader, por la BIOS desde un sector de arranque.

## 2. Modo real

### 2.1. Scripts utilizados

Para continuar con los desafíos, a continuación se detallan los scripts relevantes.

#### 2.1.1. Script `link.ld`

Este archivo es un script de linker extraído desde `examples/HelloWorld`, utilizado para definir la disposición de las secciones en memoria. Específicamente, ubica el código en la dirección `0x7C00`, que es donde la BIOS carga el primer sector del disco:

Listing 1: Script de enlace `link.ld`

```
SECTIONS
{
    /* La BIOS carga el código desde el disco en esta ubicación.
     * Debemos indicárselo al linker para que pueda calcular
     * correctamente las direcciones de los símbolos.
     */
    . = 0x7c00;
    .text :
    {
```

```

__start = .;
*(.text)
/* Coloca la firma mágica del boot al final del primer sector (512
   bytes). */
. = 0x1FE;
SHORT(0xAA55)
}
}

```

### 2.1.2. Código main.S

Este archivo contiene código en ensamblador de 16 bits. La extensión `.S` indica que será procesado previamente por el preprocesador de C/C++, lo que permite usar directivas como `#include`, `#define`, etc.

Listing 2: Archivo main.S

```

.code16
    mov $msg, %si
    mov $0x0e, %ah
loop:
    lodsb
    or %al, %al
    jz halt
    int $0x10
    jmp loop
halt:
    hlt
msg:
    .asciz "hello world"

```

## 2.2. ¿Qué es la dirección que aparece en el script del linker? ¿Por qué es necesaria?

El script de enlace `link.ld` incluye la línea:

```
. = 0x7c00;
```

Esta línea establece el contador de ubicación (*location counter*) en la dirección de memoria `0x7c00`.

Esta dirección es muy importante porque la BIOS, al iniciar el sistema, lee los primeros 512 bytes de un dispositivo de arranque y los carga en memoria comenzando en esa dirección. Este valor de dirección es el estándar para sectores de arranque en sistemas x86.

Esta dirección asegura que el código ejecutable se coloque exactamente donde la BIOS espera encontrarlo. En caso de no especificarlo, el *linker* podría colocar el código ejecutable en una dirección incorrecta, lo cual impediría que la BIOS lo ejecute correctamente.

```

/* Place the magic boot bytes at the end of the first 512 sector. */
. = 0x1FE;
SHORT(0xAA55)

```

La dirección `0x1FE` es el byte final del sector de arranque, que tiene una longitud de 512 bytes. Básicamente, esta dirección se utiliza para colocar los “magic boot bytes”, en los cuales se agrega el valor hexadecimal `0xAA55`. Este valor corresponde a una firma especial que le indica a la BIOS que el dispositivo es un dispositivo de arranque válido.

## 2.3. Objdump y Hexdump

Para analizar la ubicación del programa dentro de la imagen, podemos utilizar dos tipos de herramientas: `objdump` y `hexdump`.

### 2.3.1. Objdump

Esta herramienta muestra información detallada sobre archivos objeto, como secciones, símbolos y desensamblado del código. Cuando se aplica a `main.o`, se visualiza la sección `.text` con direcciones relativas comenzando desde 0, ya que los archivos objeto no tienen direcciones absolutas hasta el enlace.

### 2.3.2. Hexdump

Esta herramienta muestra el contenido del archivo binario en formato hexadecimal, decimal u octal. Cuando se ejecuta en `main.img`, muestra los bytes crudos del archivo.

## 2.4. Pasos para la ejecución

Para realizar esta ejecución, se deben seguir los siguientes pasos:

```
as -g -o main.o main.S
ld --oformat binary -o main.img -T link.ld main.o
qemu-system-x86_64 -hda main.img
```

- El primer comando convierte el código ensamblador de `main.S` en un archivo objeto `main.o` utilizando el comando `as`. La opción `-g` añade información de depuración (útil para usar con `gdb`). La opción `-o` indica el nombre del archivo de salida.
- El segundo comando usa el *linker* de GNU para combinar los archivos objeto y generar una imagen binaria. La opción `--oformat binary` asegura que el formato de salida sea binario.
- El último comando ejecuta un emulador para probar la imagen generada usando `qemu`, el cual define una máquina con arquitectura `x86_64`, y la opción `-hda main.img` especifica que `main.img` sea tratado como el disco duro primario de la máquina virtual emulada.





Figura 1: Ejemplo del main.s

Primero vamos a analizar el archivo binario con hexdump. Para realizar esto, ejecutamos el siguiente comando:

```
hd main.img
```

Y lo que vamos a observar es la siguiente salida:

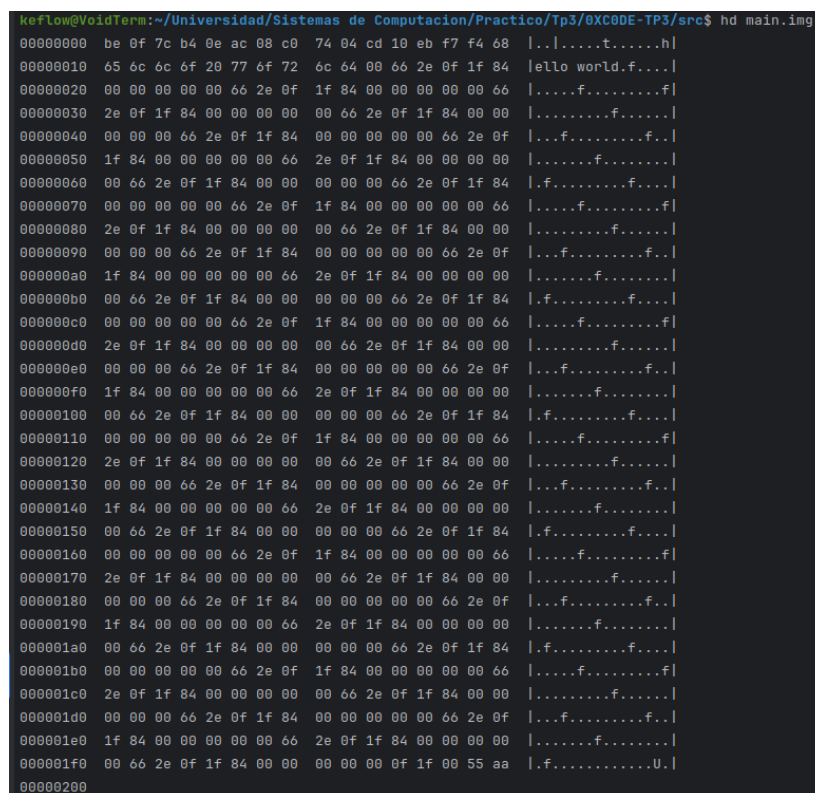


Figura 2: Salida de main.img

## 2.5. Tamaño y contenido de main.img

El tamaño del binario `main.img` va desde la posición de memoria `0x00000000` hasta la `0x00000200`, lo que indica claramente que tiene un tamaño de **512 bytes**.

En la posición de memoria `0x00000010` (incluyendo además el último byte de la posición `0x00000000`) comienza la cadena de texto mostrada. Nos damos cuenta de esto al traducir la siguiente secuencia hexadecimal a código ASCII:

68 65 6c 6c 6f 20 77 6f 72 6c 64

Esto nos da como resultado la frase:

"hello world"

También podemos notar que en los últimos dos bytes se encuentra la **firma de arranque** `0x55AA`, lo cual indica que se trata de un archivo de arranque válido reconocido por la BIOS.

## Análisis con Objdump

Utilizando la herramienta `objdump`, podemos ejecutar los siguientes comandos:

```
objdump -d main.o
objdump -D main.o
```

Estos comandos desensamblan el archivo objeto `main.o`. La opción `-d` muestra únicamente el contenido de la sección `.text`, mientras que `-D` muestra el contenido completo de todas las secciones del archivo objeto.

La salida mostrará instrucciones en lenguaje ensamblador con sus correspondientes direcciones relativas, permitiéndonos verificar cómo fue traducido el código fuente al código máquina.

```
keFlow@VoidTerm:~/Universidad/Sistemas de Computacion/Practico/TP3/0XC0DE-TP3/src$ objdump -D main.o
main.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <loop-0x5>:
  0:  be 00 00 b4 0e      mov     $0xeb40000,%esi

0000000000000005 <loop>:
  5:  ac                  lods    %ds:(%rsi),%al
  6:  08 c0              or      %al,%al
  8:  74 04              je      e <halt>
 a:  cd 10              int     $0x10
 c:  eb f7              jmp     5 <loop>

000000000000000e <halt>:
 e:  f4                  hlt

000000000000000f <msg>:
 f:  68 65 6c 6c 6f      push    $0x6f6c6c65
14:  20 77 6f            and     %dh,0x6f(%rdi)
17:  72 6c              jnb     85 <msg+0x76>
19:  64                  fs
...
```

Figura 3: Salida del `main.o`

### 2.5.1. Sección .text

Nos enfocamos especialmente en la parte de la sección `.text`, donde los bytes que visualizábamos anteriormente utilizando la herramienta `hexdump`, ahora los vemos representados por sus instrucciones respectivas en lenguaje ensamblador. Esto nos permite identificar con claridad cómo cada instrucción fue codificada en el archivo binario.

## 2.6. Depuración con GDB

Utilizando la herramienta `QEMU` en conjunto con `GDB`, podemos depurar el código contenido en `main.S`. Estas dos herramientas trabajan en conjunto para permitirnos ejecutar la imagen generada y examinar paso a paso su funcionamiento.

Para iniciar esta depuración, primero debemos ejecutar el siguiente comando:

```
qemu-system-i386 -drive file=main.img,format=raw -boot a -s -S -monitor stdio
```

Explicación de las opciones utilizadas:

- `-drive file=main.img,format=raw`: Carga la imagen generada `main.img` como un disco en formato crudo.
- `-boot a`: Le indica a `QEMU` que arranque desde el disco especificado (unidad A).
- `-s`: Inicia un servidor GDB que escucha en el puerto 1234, lo que permite conectar un depurador externo.
- `-S`: Le dice al CPU que se detenga inmediatamente al comenzar, hasta que el depurador (`GDB`) se conecte.
- `-monitor stdio`: Abre una interfaz de consola para enviar comandos a `QEMU` desde la línea de comandos.

Después de ejecutar este comando, `QEMU` se iniciará y mostrará una ventana “congelada”, esperando que `GDB` se conecte para comenzar la depuración. Esto permite realizar una inspección detallada del comportamiento del sistema desde su primera instrucción.

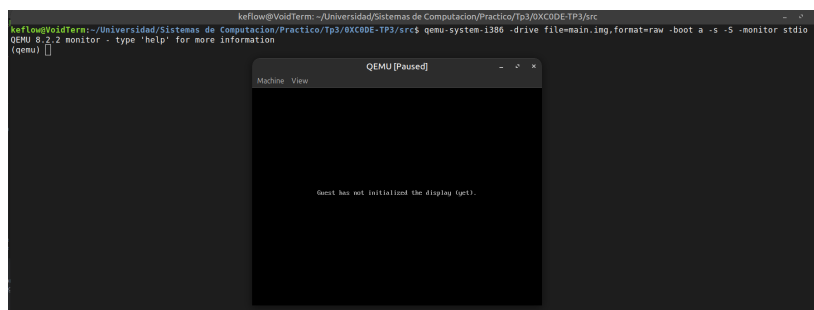


Figura 4: QEMU espera hasta conectarse con GDB

## 2.7. Conexión de GDB al servidor de QEMU

Luego, es necesario abrir otra terminal y ejecutar el depurador GDB. En esta terminal se escriben los siguientes comandos, que conectan al depurador con el servidor GDB que QEMU inició en el puerto local 1234:

```
gdb -q
(gdb) target remote localhost:1234
(gdb) set architecture i8086
(gdb) break *0x7c00
(gdb) continue
```

Explicación de los comandos:

- `gdb -q`: Inicializa el depurador GDB en modo silencioso (sin mostrar el mensaje de copyright).
- `target remote localhost:1234`: Conecta GDB al servidor GDB iniciado por QEMU en el puerto 1234.
- `set architecture i8086`: Establece que el CPU será depurado en modo real de 16 bits, correspondiente a la arquitectura i8086.
- `break *0x7c00`: Establece un punto de interrupción en la dirección de memoria 0x7c00, que corresponde a la ubicación donde la BIOS carga el sector de arranque (bootloader).
- `continue`: Continúa la ejecución hasta alcanzar el punto de interrupción establecido.

Con esta configuración, es posible comenzar la depuración paso a paso del código cargado en la dirección 0x7c00, observando y analizando el comportamiento del sistema desde el arranque.

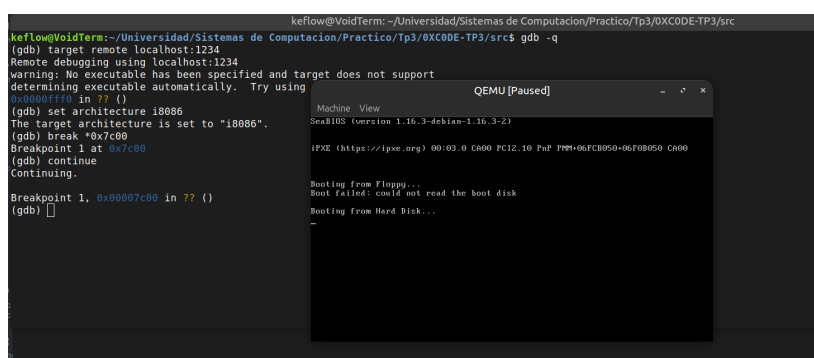


Figura 5: Depurador GDB

## 2.8. Inspección del código ensamblador y breakpoints

Una vez conectados al servidor de GDB y ejecutado el comando `continue`, el sistema se detiene en la dirección 0x7c00, que es el inicio del código a depurar.

Para ver el código ensamblador que comienza en esa dirección, podemos utilizar el siguiente comando en GDB:

```
(gdb) x/20i 0x7c00
```

Este comando muestra las próximas 20 instrucciones en lenguaje ensamblador a partir de la dirección de memoria 0x7c00.

El objetivo de esta inspección es colocar puntos de interrupción (*breakpoints*) que nos permitan visualizar la ejecución paso a paso del archivo `main.S`. En particular, busquemos identificar la dirección de memoria donde se agregan los caracteres que forman la cadena "hello world".


Para ello, podemos usar los siguientes comandos:

```
(gdb) x/20i 0x7c00
(gdb) break *0x7c0a
(gdb) continue
```

Explicación de los comandos:

- `x/20i 0x7c00`: Muestra en código ensamblador las siguientes 20 instrucciones a partir de la dirección 0x7c00.
- `break *0x7c0a`: Establece un punto de interrupción en la dirección de memoria donde comienza a escribirse la cadena de texto "hello world".
- `continue`: Reanuda la ejecución del código hasta alcanzar el siguiente punto de interrupción.

De esta manera, podemos observar detalladamente cómo se forma la salida en pantalla, instrucción por instrucción, dentro del entorno de depuración.



```

keflow@VoidTerm: ~/Universidad/Sistemas de Computacion/Practico/TP3/0XC0DE-TP3/src
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/20i 0x7c00
0x7c00: mov $0xb47c0f,%eax
0x7c05: lods %eax(%eax),%eax
0x7c06: or %eax,%eax
0x7c08: je 0x7c0e
0x7c0a: int $0x10
0x7c0c: jmp 0x7c05
0x7c0e: hlt
0x7c0f: push $0xb6c0c05
0x7c14: and %eax,%eax(%eax)
0x7c17: jb 0x7c05
0x7c19: add %eax,%eax(%eax)
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)
0x7c61: nopw %cs:0x0(%eax,%eax,1)
0x7c63: nopw %cs:0x0(%eax,%eax,1)
(gdb) break *0x7c0a
Breakpoint 2 at 0x7c0a
  
```

Figura 6: Breakpoint1

Además, con un comando podemos ver el registro al que corresponde a los 8 bits de la parte baja del registro AX. Donde se puede ver claramente que se muestra en hexadecimal el valor 0x68 que representa en código ascii a la letra h. Además, el monitor de QEMU todavía no muestra ninguna letra ya que se le debe dar un step.

```

keflow@VoidTerm: ~/Universidad/Sistemas de Computacion/Practico/TP3/0XC0DE-TP3/src
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)
0x7c61:  nopw %cs:0x0(%eax,%eax,1)
0x7c6b:  nopw %cs:0x0(%eax,%eax,1)
(gdb) break *0x7c8a
Breakpoint 2 at 0x7c8a
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c8a in ?? ()
(gdb) info registers al
al                0x68      104
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c8a in ?? ()
(gdb) info registers al
al                0x65      101
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c8a in ?? ()
  
```

```

keflow@VoidTerm: ~/Universidad/Sistemas de Computacion/Practico/TP3/0XC0DE-TP3/src
0x7c05:  lods %ds:(%eax),%eax
0x7c06:  or %eax,%eax
0x7c08:  je 0x7c0c
0x7c0a:  int $0x10
0x7c0c:  jmp 0x7c05
0x7c0e:  hlt
0x7c0f:  push $0x0f6c6c65
0x7c14:  and %edx,%edx(%edi)
0x7c17:  jb 0x7c05
0x7c19:  add %eax,%eax(%eax,1)
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)
0x7c61:  nopw %cs:0x0(%eax,%eax,1)
0x7c6b:  nopw %cs:0x0(%eax,%eax,1)
(gdb) break *0x7c8a
Breakpoint 2 at 0x7c8a
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c8a in ?? ()
(gdb) info registers al
al                0x68      104
(gdb)
  
```

Figura 7: Registro AX

Después de ejecutar el comando de ‘continue’ observamos lo siguiente:

```

keflow@VoidTerm: ~/Universidad/Sistemas de Computacion/Practico/TP3/0XC0DE-TP3/src
0x7c0c:  jmp 0x7c05
0x7c0e:  hlt
0x7c0f:  push $0x0f6c6c65
0x7c14:  and %edx,%edx(%edi)
0x7c17:  jb 0x7c05
0x7c19:  add %eax,%eax(%eax,1)
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)
0x7c61:  nopw %cs:0x0(%eax,%eax,1)
0x7c6b:  nopw %cs:0x0(%eax,%eax,1)
(gdb) break *0x7c8a
Breakpoint 2 at 0x7c8a
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c8a in ?? ()
(gdb) info registers al
al                0x68      104
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c8a in ?? ()
  
```

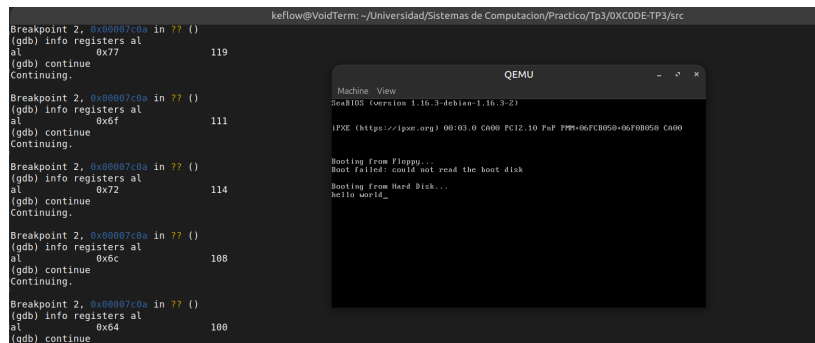
Figura 8: Ejecutamos el comando continue

Y así, sucesivamente podemos ir ejecutando el comando de ‘continue’ y ver en QEMU cómo se va escribiendo el mensaje letra por letra.

## 3. Modo protegido

### 3.1. Modo Protegido y la Tabla GDT

Para realizar un código en ensamblador que utilice el **modo protegido**, lo primero es crear una Tabla de Descriptores Globales (GDT). La estructura de la GDT puede definirse de la siguiente manera:



```

Breakpoint 2, 0x00007c0a in ?? ()
(gdb) info registers al
al                0x77                119
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c09 in ?? ()
(gdb) info registers al
al                0x6f                111
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c0a in ?? ()
(gdb) info registers al
al                0x72                114
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c0a in ?? ()
(gdb) info registers al
al                0x6c                108
(gdb) continue
Continuing.

Breakpoint 2, 0x00007c0a in ?? ()
(gdb) info registers al
al                0x64                100
(gdb) continue
  
```

```

gdt_start:
gdt_null:
    .long 0x0
    .long 0x0

gdt_code:
    .word 0xffff
    .word 0x0
    .byte 0x0
    .byte 0b10011010
    .byte 0b11001111
    .byte 0x0

gdt_data:
    .word 0xffff
    .word 0x0
    .byte 0x0
    .byte 0b10010010
    .byte 0b11001111
    .byte 0x0

gdt_end:

gdt_descriptor:
    .word gdt_end - gdt_start
    .long gdt_start
  
```

La tabla comienza en `gdt_start` e incluye tres descriptores: `gdt_null`, `gdt_code` y `gdt_data`.

- `gdt_null` es un segmento nulo que previene referencias accidentales. Es obligatorio en x86.
- `gdt_code` y `gdt_data` definen segmentos de código y datos, respectivamente.

Cada entrada incluye el límite del segmento (`0xffff`), la dirección base (`0x0`), y varios atributos en los últimos bytes.

### 3.1.1. Atributos del descriptor

- Byte de acceso (Access Byte):

- Bit 7: Segmento accedido.
- Bits 6-4: Tipo de segmento (101 para código, 001 para datos).
- Bit 3: Segmento normal (1).
- Bits 2-1: Nivel de privilegio (0 = mayor privilegio).
- Bit 0: Presente en memoria (1).

■ **Byte de flags y límite alto:**

- Bits 0-3: Límite de segmento (bits 16-19).
- Bit 3: Disponible.
- Bit 2: Reservado.
- Bit 1: Tamaño (1 = 32 bits, 0 = 16 bits).
- Bit 0: Granularidad (0 = bytes, 1 = páginas de 4KB).

### 3.2. Habilitar el modo protegido

Al inicio del archivo ensamblador, agregamos lo siguiente:

```
.code16
.equ CODE_SEG, 8
.equ DATA_SEG, gdt_data - gdt_start

lgdt gdt_descriptor

/* Habilitar el bit PE (Protection Enable) en CR0 */
mov %cr0, %eax
orl $0x1, %eax
mov %eax, %cr0

ljmp $CODE_SEG, $protected_mode
```

- `lgdt gdt_descriptor`: Le indica al procesador dónde está la GDT.
- Se habilita el modo protegido estableciendo el bit PE en el registro CR0.
- El salto largo `ljmp` transfiere el control al segmento de código en modo protegido.

### 3.3. Configuración de registros de segmento

Una vez en modo protegido, comenzamos nuestro código en `.code32`:

```
.code32
protected_mode:
    mov $DATA_SEG, %ax
    mov %ax, %ds
    mov %ax, %es
    mov %ax, %fs
    mov %ax, %gs
    mov %ax, %ss
```

Aquí configuramos todos los registros de segmento para apuntar al descriptor de datos. También podríamos haber definido un segmento especial para la pila y asignarlo al registro `ss`.



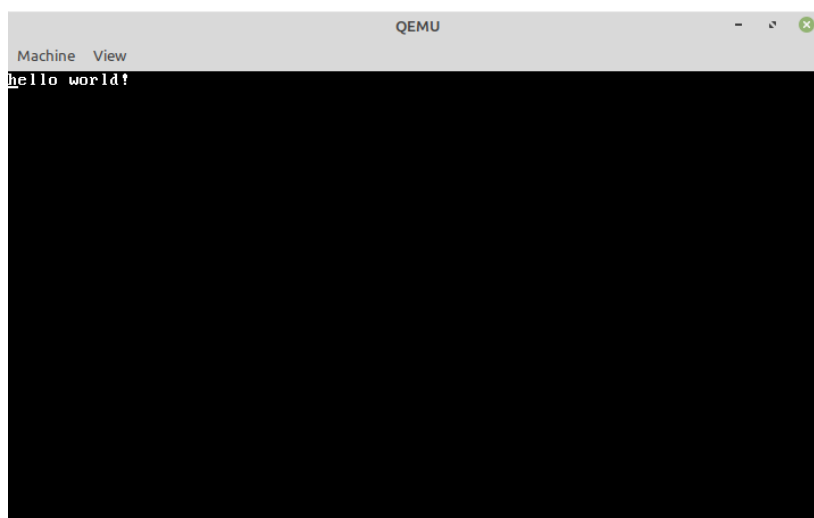


Figura 9: Resultado

### 3.4. Ejemplo: Hello World en Modo Protegido

A continuación se puede ejecutar cualquier código, como un “Hello World”. Aunque no explicamos este ejemplo en detalle, lo importante es que está contenido dentro del entorno en modo protegido.

### 3.5. Compilación y Ejecución

Para compilar y ejecutar el programa se utilizan los siguientes comandos:

```
as --32 -o main.o main.S
ld -m elf_i386 --oformat binary -o main.img -T link.ld main.o
qemu-system-x86_64 -drive format=raw,file=main.img
```

- `as --32`: Ensambla el código para una arquitectura de 32 bits.
- `ld -m elf_i386 --oformat binary`: Enlaza el código en formato binario plano.
- `qemu-system-x86_64`: Ejecuta la imagen binaria en una máquina virtual x86.

El resultado fue:

### 3.6. Cambios en los Permisos del Segmento de Datos

¿Qué sucede si cambiamos los **bits de acceso** del segmento de datos para que sea de **sólo lectura**? Para comprobarlo, modificamos la línea 39 del archivo `main.S` ubicado en el directorio `src/modo protegido`:

```
.byte 0b10010010
```

Cambiamos esta línea a:

```
.byte 0b10010000
```

Esto elimina los permisos de escritura sobre el segmento de datos.

### 3.6.1. Efecto del Cambio

Al volver a ejecutar el código usando QEMU, notamos que el sistema se **reinicia constantemente**. Esto ocurre porque se está generando una interrupción debido a un acceso inválido de escritura en un segmento marcado como sólo lectura. Dado que no se ha definido una rutina para manejar dicha excepción, la CPU reinicia el sistema.

### 3.6.2. Depuración con GDB

Para encontrar la causa exacta del reinicio, volvemos a compilar el programa con información de depuración, al igual que lo hicimos anteriormente, pero esta vez desde el directorio `src/modo protegido`:

```
as -g -o main.o main.S
ld --oformat binary -o main.img -T link.ld main.o
qemu-system-x86_64 -hda main.img
```

Antes de ejecutar GDB, utilizamos `objdump` para averiguar la dirección exacta de la etiqueta `protected_mode`:

```
objdump -d main.o
```

Este comando nos mostrará el contenido del archivo objeto desensamblado, incluyendo las direcciones relativas. Buscamos en la salida la línea correspondiente a `protected_mode`, lo cual nos permitirá colocar puntos de interrupción precisos en GDB y observar el comportamiento del código cuando se produce el fallo por intento de escritura.

### 3.6.3. Conclusión

Quitar los permisos de escritura del segmento de datos provoca un error de protección cuando el código intenta escribir en dicho segmento. Como no se ha configurado una IDT ni una rutina para manejar excepciones, el sistema se reinicia. Esta es una demostración práctica de cómo los bits de acceso en la GDT afectan directamente la ejecución del programa y su estabilidad en modo protegido.

```
0000000000000040 <protected_mode>:
40: 66 b8 10 00      mov     $0x10,%ax
44: 8e d8            mov     %eax,%ds
46: 8e c0            mov     %eax,%es
48: 8e e0            mov     %eax,%fs
4a: 8e e8            mov     %eax,%gs
4c: 8e d0            mov     %eax,%ss
4e: b9 00 00 00 00   mov     $0x0,%ecx
53: a1 00 00 00 00 ba 00  movabs  0xba00000000,%eax
5a: 00 00
5c: 00 bb 19 00 00 00   add     %bh,0x19(%rbx)
62: f7 f3           div     %ebx
64: 89 d0            mov     %edx,%eax
66: ba a0 00 00 00     mov     $0xa0,%edx
6b: f7 e2           mul     %edx
6d: 8d 90 00 80 0b 00   lea     0xb8000(%rax),%edx
73: b4 0f           mov     $0xf,%ah
```

Figura 10: Resultado

El resultado del `objdump` se recorta para mostrar sólo las partes relevantes, especialmente la dirección de la etiqueta `protected_mode`, que en este caso se encuentra en la dirección `0x40` relativa al inicio del archivo binario.

Como sabemos que el bootloader es cargado por la BIOS en la dirección de memoria `0x7C00`, debemos sumar esta dirección base al valor relativo obtenido por `objdump`. Es decir:

$$0x7C00 + 0x40 = 0x7C40$$

Esta es la dirección absoluta en memoria donde comienza la rutina `protected_mode`.

Utilizamos esta información para colocar un punto de interrupción en GDB con el siguiente comando:

```
(gdb) break *0x7c40
```

Esto nos permite inspeccionar la ejecución justo al ingresar en modo protegido, y observar el momento en que se produce la falla si intentamos escribir en un segmento de sólo lectura.



```

$ ./src/modo_protegido$ gdb -q
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
aviso: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
(gdb) break *0x7c40
Punto de interrupción 1 at 0x7c40
(gdb) continue
Continuando.
Breakpoint 1, 0x0000000000007c40 in ?? ()
(gdb) x/20i 0x7c40
0x7c40: mov $0x19,%ax
0x7c41: mov %eax,%edi
0x7c42: mov %edi,%ecx
0x7c43: mov %ecx,%ecx
0x7c44: mov %ecx,%edi
0x7c45: mov %edi,%edi
0x7c46: mov %edi,%edi
0x7c47: mov $0x7c00,%ecx
0x7c48: movabs $0xa0000000,%eax
0x7c49: add %eax,%ecx
0x7c4a: div %ecx
0x7c4b: mov %eax,%eax
0x7c4c: mov %eax,%eax
0x7c4d: mov %eax,%eax
0x7c4e: lea 0xb000(%rip),%ecx
0x7c4f: mov %ecx,%eax
0x7c50: mov (%rip),%edi
0x7c51: cmp %edi,%edi
0x7c52: je 0x7c56
0x7c53: mov %eax,%eax
0x7c54: add %eax,%eax
(gdb) _
  
```

Figura 11: Direccion de la etiqueta

Luego de ejecutar el comando:

```
(gdb) x/20i 0x7c40
```

Observamos que la salida coincide exactamente con el código ensamblador definido en `main.S`, específicamente después de la etiqueta `protected_mode`.

Colocamos un punto de interrupción adicional en la dirección `0x7c4c`, que corresponde al final del bloque de instrucciones donde se setean los registros de segmento:

```
(gdb) break *0x7c4c
(gdb) continue
```

Una vez alcanzado ese punto, avanzamos un paso con:

```
(gdb) stepi
```

```
(gdb) break *0x7c4c
Punto de interrupción 2 at 0x7c4c
(gdb) continue
Continuando.

Breakpoint 2, 0x00000000000007c4c in ?? ()
(gdb) si
0x0000000000000e05b in ?? ()
(gdb) _
```

```
mov $message, %ecx
```

GDB nos muestra que la CPU salta a una dirección inesperada, 0xe05b, en lugar de continuar secuencialmente a 0x7c4e. Esto es indicio claro de que se ha disparado una **interrupción por fallo de segmentación**, debido a un intento de escritura en un segmento de datos configurado como de solo lectura.

## Referencias

- [1] <https://github.com/francolop/0xCODE-TP3>