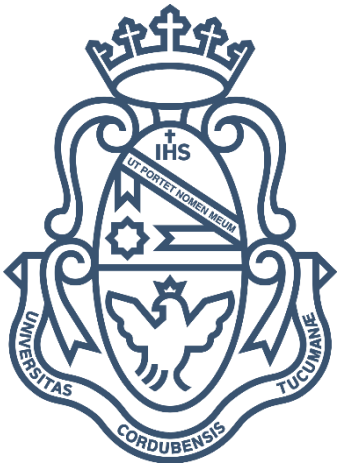

Universidad Nacional de Córdoba

Facultad de Ciencias Exactas, Físicas y Naturales

Sistemas de computación

TP3: Modo Protegido

Grupo:

Epsilon

Profesores (por órden alfabético):

Jorge, Javier Alejandro

Lamberti, Germán Andrés

Solinas, Miguel Ángel

Alumnos (por órden alfabético):

Campos, Mariano

Erlicher,Ezequiel

González, Damián Marcelo

Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales
Sistemas de computación

Actividad 1: Correr una imagen en QEMU

En esta primera instancia, se crea un MBR (master boot record). Un MBR es un sector (512 bytes), ubicado al comienzo de un dispositivo de almacenamiento (como un disco duro). Su función principal es ayudar al sistema a localizar y bootear el sistema operativo.

Dicha MBR se crea con el siguiente comando:

```
printf '\364%509s\125\252' > main.img
```

Donde:

- **`\364 in octal == 0xf4 in hex:`** Instrucción halt
- **`%509s:`** produce 509 espacios. Necesarios para completar la imagen hasta el byte 510.
- **`\125\252 en octal == 0x55 0xAA:`** requisito para que sea interpretada como un MBR
- **`main.img:`** Nombre del archivo que contiene la imagen

El contenido de main.img puede analizarse con el comando **hd (Hex dump)**. Se utiliza para mostrar el contenido de un archivo en un formato legible, específicamente en hexadecimal y ASCII. Muestra los datos binarios sin procesar del archivo y en un formato fácil de analizar

Al ejecutar el comando **hd main.img**, se abrirá el archivo main.img y se mostrará su contenido sin procesar en dos columnas:

Valores hexadecimales: Muestra los datos binarios sin procesar del archivo en formato hexadecimal.

Valores ASCII: Muestra la representación ASCII correspondiente de los datos a la derecha. Los caracteres no imprimibles suelen representarse con un punto (.).

Para ver la codificación de una instrucción puntual, se pueden ejecutar los siguientes comandos:

- **echo** hlt > a.S
- **as** -o a.o a.S
- **objdump** -S a.o

El primer comando crea un archivo assembly a.S que contiene la instrucción “hlt”. Luego, el segundo comando ensambla el archivo a.S en un archivo objeto a.o. Por último,el tercer comando “desensambla” el objeto a.o y muestra tanto el código fuente del ensamblado original como el código máquina resultante. Se creó en este caso un archivo bash llamado **create_image.sh** que al correrlo, crea los archivos **a.S** y **main.img**

```
1   # Crea una imagen llamada main.img
2   printf '\364%509s\125\252' > main.img
3
4   # Crea un archivo llamado a.S y escribe la instrucción "hlt" en él
5   echo hlt > a.S
6
7   # Crea el archivo assembly a.S y generar un archivo objeto llamado a.o
8   as -o a.o a.S
```

*Figura 1: **create_image.sh***

Una vez creado estos archivos, se pueden utilizar los comandos **objdump** y **hd** para inspeccionar la información que se mencionaba anteriormente**.** Se puede ver en el output de ambos comandos la codificación de la instrucción halt (f4).

```
ezerlich@ezerlich-Lenovo-V330-15IKB:~/Documents/TPS-SDC-2025/SDC-TP3-Epsilon$ objdump -S a.o

a.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   f4                      hlt
ezerlich@ezerlich-Lenovo-V330-15IKB:~/Documents/TPS-SDC-2025/SDC-TP3-Epsilon$ hd main.img
00000000  f4 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |.               |
00000010  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                |
*
000001f0  20 20 20 20 20 20 20 20  20 20 20 20 20 20 55 aa  |              U.|
00000200
```

*Figura 2: Ejecución de los comandos **objdump** y **hd***

QEMU (Quick Emulator) es un emulador y virtualizador de código abierto. En esencia, QEMU permite ejecutar diferentes sistemas operativos y entornos de software en un equipo host, incluso si están diseñados para arquitecturas de hardware completamente distintas. Haciendo uso de este programa, se corre el archivo main.img creado mediante el comando:

```
qemu-system-x86_64 --drive file=main.img,format=raw,index=0,media=disk
```

Se puede observar cómo se ejecuta la instrucción halt exitosamente, congelando la pantalla que se despliega.

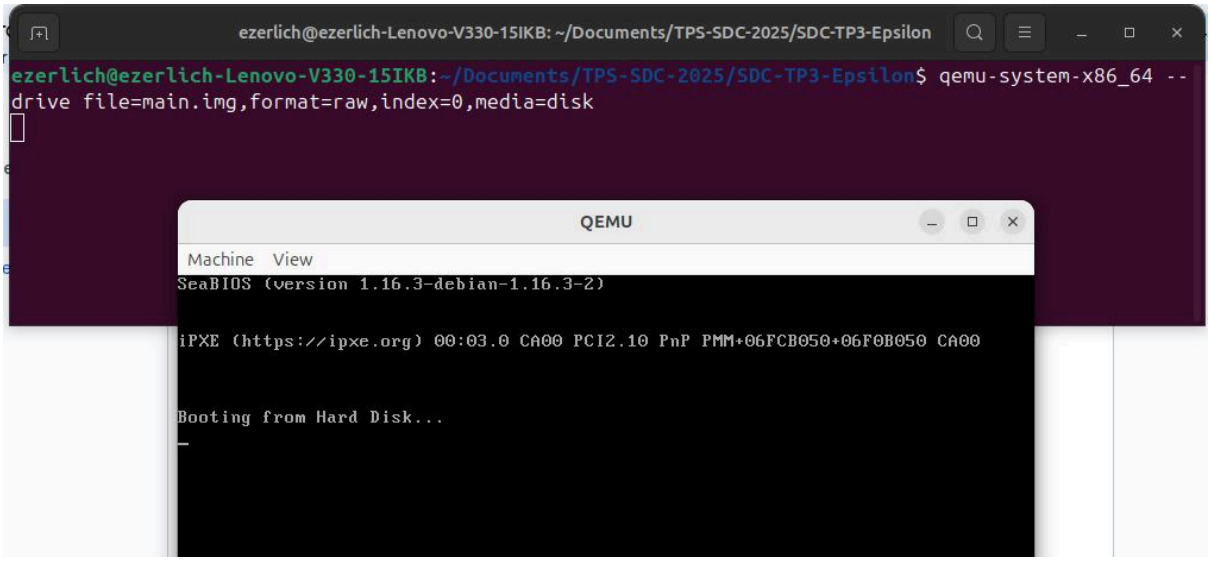

Figura 3: Ejecución de la imagen main.img en QEMU

Actividad 2: preguntas sobre UEFI y coreboot

¿Qué es UEFI? ¿cómo puedo usarlo? Mencionar además una función a la que podría llamar usando esa dinámica.

UEFI es una especificación estándar que define cómo se interconectan el sistema operativo y el firmware de una computadora proporcionando una interfaz para el booteo y otras tareas de inicialización del sistema. UEFI reemplaza la antigua interfaz del sistema básico de entrada y salida (BIOS).

Para acceder a UEFI, se pueden usar los siguientes pasos generales:

1. Encender la computadora.
2. Durante el arranque, presionar la tecla adecuada para entrar en la configuración de la BIOS/UEFI. Esto varía según el fabricante, pero suelen ser teclas como **Esc**, **F2**, **F10**, **Del**, etc.
3. Una vez dentro del menú de configuración UEFI, se puede navegar entre diferentes opciones. Se pueden configurar aspectos tales como la prioridad de los dispositivos de arranque (por ejemplo, elegir arrancar desde un USB o un disco duro) . También se puede activar o desactivar el **"Secure Boot",**un mecanismo que ayuda a proteger el sistema contra malware que intenta cargar antes del sistema operativo**.** Si se quiere usar herramientas de arranque personalizadas es necesario desactivar esta función

Menciona casos de bugs de UEFI que puedan ser explotados

Algunas vulnerabilidades que presenta UEFI son las siguientes:

CVE-2024-7344: Esta vulnerabilidad permite eludir el arranque seguro (Secure Boot) en sistemas con ciertas versiones de software de recuperación, como Radix SmartRecovery, Greenware GreenGuard, y otros.**Mas info:** https://www.welivesecurity.com/es/investigaciones/uefi-secure-boot-vulnerabilidad-bootkit/

LogoFail: Explota las vulnerabilidades de las bibliotecas de análisis de imágenes utilizadas durante el proceso de arranque, lo que permite a los atacantes reemplazar el logotipo de arranque del fabricante con código malicioso.**Mas info:** **https://www.kaspersky.com/blog/logofail-uefi-vulnerabilities/50160/**

¿Qué es Converged Security and Management Engine (CSME), the Intel Management Engine BIOS Extension (Intel MEBx).?

Intel CSME es un subsistema integrado y un dispositivo PCIe (Peripheral Component Interconnect Express) diseñado para actuar como controlador de seguridad y administración en el PCH (Platform Controller Hub). Busca implementar un entorno informático aislado del software principal del host (SW) que ejecuta la CPU, como la BIOS (Sistema Básico de Entrada/Salida), el SO (Sistema Operativo) y las aplicaciones. Intel CSME puede acceder a un número limitado de interfaces, como GPIO (Entrada/Salida de Propósito General) y LAN/LAN Inalámbrica (WLAN), para realizar sus operaciones previstas. CSME es un componente de

Intel Platform Trust Technology (Intel PTT) y forma parte de una arquitectura más grande que también incluye el Intel Management Engine (ME).

Figure 1. Intel® CSME in the System

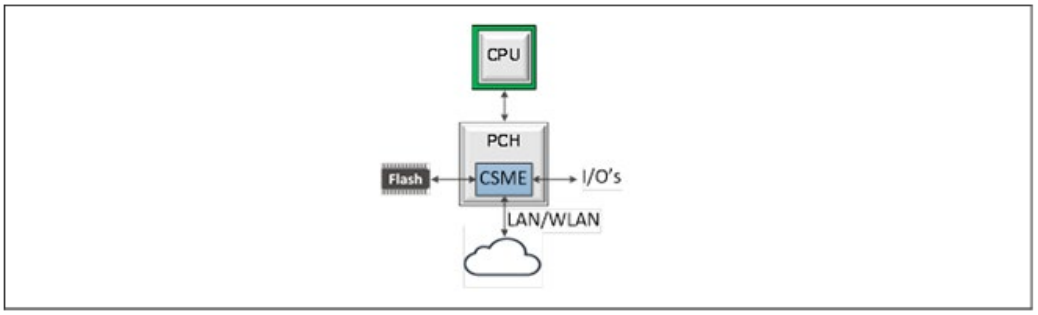

Figura 4: Intel CSME

Entre sus funciones se pueden mencionar:

- **Seguridad**: CSME ayuda a garantizar la integridad de los datos y la autenticación en el hardware, protegiendo contra ataques físicos y de software. Por ejemplo, puede gestionar la autenticación de dispositivos y encriptar datos en reposo.
- **Gestión remota**: Facilita la gestión remota de sistemas, permitiendo a los administradores realizar tareas de mantenimiento, actualizaciones o configuraciones incluso si el sistema operativo no está operativo.
- **Arranque seguro**: Ayuda en el proceso de arranque seguro del sistema operativo, verificando que el firmware y el software no hayan sido alterados maliciosamente.
- **Protección de datos**: CSME es responsable de implementar tecnologías como Intel's **Platform Trust Technology (PTT)** y **Intel TPM** (Trusted Platform Module), que son usadas para cifrar y proteger datos sensibles.

Mas info:
https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel-csme-security-white-paper.pdf

El **Intel Management Engine BIOS Extension** (Intel MEBx) es una interfaz de configuración y administración para el **Intel Management Engine (ME)**, que es una parte crucial de la arquitectura de gestión de sistemas Intel. MEBx es una utilidad de la BIOS (Basic Input/Output System) que permite configurar y gestionar las opciones de ME a nivel de firmware, antes de que el sistema operativo se cargue.Es importante destacar que MEBx solo es accesible a través de la interfaz de la BIOS/UEFI y que se utiliza principalmente en entornos empresariales donde la gestión remota y la seguridad son prioritarias.

El **CSME** es un subsistema que incluye el **Intel ME** como parte de su funcionalidad. Como el **Intel MEBx** es la interfaz de configuración para administrar las opciones del **Intel ME**, entonces, también gestiona algunas funciones que forman parte del **CSME**.

¿Qué es coreboot ? ¿Qué productos lo incorporan ?¿Cuales son las ventajas de su utilización?

Coreboot es un firmware de código abierto que apunta a reemplazar a las soluciones tradicionales propietarias de BIOS o UEFI en muchos dispositivos. El propósito de Coreboot es proporcionar un entorno más rápido y flexible para el proceso de booteo de un sistema, utilizando un conjunto de herramientas y módulos personalizables. El objetivo es hacer lo mínimo para inicializar el hardware y luego entregar el control al sistema operativo.

Entre sus ventajas se cuentan:

1. **Arranque rápido**: Coreboot está diseñado para ser mucho más rápido que los firmwares tradicionales como el BIOS o UEFI.
2. **Personalización y flexibilidad** : Al ser software de código abierto, Coreboot permite a los usuarios inspeccionar, modificar y personalizar el firmware según sus necesidades, añadiendo solo lo que necesitan, y excluyendo lo que no.
3. **Seguridad mejorada**: Con Coreboot, es posible implementar características de seguridad para proteger el sistema contra ataques de firmware. Además, al ser más simple y transparente, es más fácil detectar posibles vulnerabilidades.
4. **Compatibilidad con hardware antiguo**: Coreboot puede ser utilizado sobre hardware antiguo que ya no recibe soporte de los fabricantes, extendiendo la vida de dispositivos que de otro modo estarían obsoletos.
5. **Soporte de arquitecturas diversas**: Coreboot es compatible con una amplia gama de arquitecturas de procesador, como x86, ARM y RISC-V, lo que lo hace ideal para su implementación en una variedad de dispositivos.

Algunos de los productos que incorporan coreboot son:

Chromebooks: Muchos modelos de Chromebooks, especialmente los de marcas como Google, Dell, Lenovo y Acer, utilizan Coreboot para acelerar el proceso de arranque y mejorar la seguridad.

Motherboards: Algunas placas base para servidores o PCs de gama alta, de fabricantes como **Asus**, **Gigabyte** o **Supermicro**, pueden ser compatibles con Coreboot. Los usuarios pueden flashear el firmware de la placa base para utilizar Coreboot en lugar de UEFI.

Sistemas embebidos: En dispositivos como routers, sistemas de control industrial y algunos productos de automatización, Coreboot puede ser implementado debido a su eficiencia y capacidad para correr en hardware limitado.

Sistemas de computación personalizados: Usuarios de hardware abierto prefieren instalar Coreboot en sus PCs personalizados para tener un control total sobre el inicio del sistema.

Actividad 3: Linker

En esta actividad, se creó un código en Assembly llamado **main.S** que, luego de ejecutar la instrucción "halt", despliega el mensaje "*hello humans :)*".

```
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 humans :)"
```

Figura 5: código "Hello humans :)"

`.code16`: Indica que el código está destinado a ejecutarse en un entorno de 16 bits,

`mov $msg, %si`: Mueve la dirección de la cadena de texto msg (que está definida más abajo) al registro SI. El registro SI se usa aquí como un puntero que apunta al primer carácter de la cadena que queremos mostrar.

`mov $0x0e, %ah`: Este comando mueve el valor 0x0e (que es 14 en decimal) al registro AH. El valor 0x0e es un código de función para la interrupción INT 0x10 que se usa para mostrar texto en pantalla en modo de texto. Específicamente, 0x0e indica que se usará la función Teletipo (TTY) para imprimir un solo carácter.

`loop:` Define una etiqueta llamada loop, a la cual el código saltará repetidamente.

`lodsb:` es una instrucción que carga un byte de memoria (apuntado por SI) en el registro AL. Después de esta instrucción, el registro SI se incrementa automáticamente, apuntando al siguiente carácter de la cadena.

`or %al, %al:` Esta instrucción realiza la operación de OR bit a bit en el contenido del registro AL. Esto no cambia el valor de AL, pero modifica las banderas. Lo que estamos buscando es si el valor de AL es 0. Si AL es 0, eso significa que hemos llegado al final de la cadena de texto (porque la cadena termina con un byte 0, conocido como null terminator).

`jz halt:` JZ significa "Jump if Zero" (salta si el resultado de la operación anterior fue cero).Si el valor de AL es 0 (es decir, hemos llegado al final de la cadena), se salta a la etiqueta halt, lo que termina el programa.

`int $0x10:` es una interrupción de BIOS usada para funciones relacionadas con la pantalla en modo texto. En este caso, está utilizando la función 0x0e que ya se había configurado en AH.Esta instrucción muestra el carácter que está en AL en la pantalla en la posición actual del cursor.

`jmp loop:` JMP loop es un salto incondicional de vuelta a la etiqueta loop, lo que hace que el programa siga procesando y mostrando el siguiente carácter de la cadena.

`halt:` Define una etiqueta llamada halt, que es el destino del salto cuando el programa detecta que ha llegado al final de la cadena. Aquí se ejecuta la instrucción hlt, que detiene la ejecución del programa.

`msg: .asciz "Hello humans :)":` Aquí se define la cadena de texto "Hello humans :)" en memoria. La directiva .asciz se usa para definir una cadena de caracteres ASCII que termina en un byte nulo (valor 0), lo que marca el final de la cadena.

Qué es un linker? que hace?

Un **linker** es una herramienta de software que se encarga de combinar varios archivos objeto (.o), generalmente generados por el compilador, en un solo archivo ejecutable o librería. Su trabajo consiste en resolver las dependencias entre los diferentes módulos de un programa y organizar la memoria de manera adecuada para que el código pueda ejecutarse correctamente.

Por otra parte, para crear una imagen booteable desde el sector de arranque de un disco, se crea el archivo **link.ld,** que es un script que ayuda a controlar cómo el linker organiza las secciones de código y datos en el binario final

```
SECTIONS
{
    /* The BIOS loads the code from the disk to this location.
     * We must tell that to the linker so that it can properly
     * calculate the addresses of symbols we might jump to.
     */
    . = 0x7c00;
    .text :
    {
        __start = .;
        *(.text)
        /* Place the magic boot bytes at the end of the first 512 sector. */
        . = 0x1FE;
        SHORT(0xAA55)
    }
}
/*
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
*/
```

Figura 6: Linker script

El bloque SECCIONES especifica la distribución de la memoria.

`. = 0x7c00:` establece la dirección de memoria inicial para el código en 0x7c00, que es la dirección de inicio tradicional para un gestor de arranque en la arquitectura x86. Aquí es donde la BIOS suele cargar el primer sector de 512 bytes (el MBR).

`.text:` define la sección .text, que es la sección donde se almacena el código ejecutable.

`__start = .;` establece el símbolo `__start` para referirse a la dirección actual (.), que marca el punto de entrada o el punto de inicio del código en memoria.

`*(.text):` significa que el linker incluirá todas las secciones .text de los archivos objeto en esta parte del binario final. Aquí es donde se almacena el código del programa.

`. = 0x1FE;:` mueve el contador de ubicación al byte 510 (es decir, 0x1FE)

`SHORT(0xAA55):` es una directiva especial que coloca el valor de 16 bits 0xAA55 en la ubicación actual. Este es el "número mágico" estándar para un sector de arranque. La BIOS busca esta firma para verificar que el sector contenga código de arranque válido.

Con el archivo **link.ld** creado, el script de bash **compile_and_run.sh** crea y corre la imagen booteable, imprimiendo exitosamente el mensaje.

```
1    #!/bin/bash
2
3    as -g -o main.o main.S
4    ld --oformat binary -o main.img -T link.ld main.o
5    qemu-system-x86_64 -hda main.img
```

Figura 7: compile_and_run.sh

La opción **--oformat binary** en el linker se utiliza para generar un archivo binario “plano” a partir de los archivos objeto, en lugar de un ejecutable en formato ELF (o cualquier otro formato de archivo ejecutable/linkeable estándar). En lugar de crear un archivo ejecutable con una estructura específica (como encabezados, tablas de símbolos, etc.), el linker simplemente concatena los datos de los archivos de código objeto en un solo archivo binario.

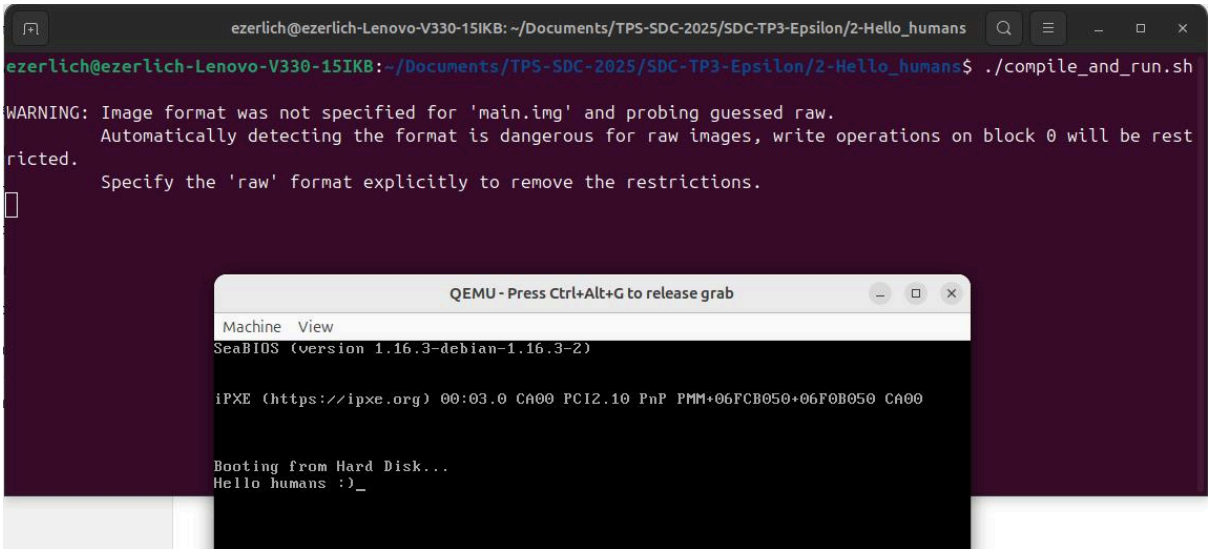

Figura 8: ejecución d ela imagen en QEMU con el mensaje “hello humans :)”

Haciendo uso de los comandos **hd** y **objdump** que se habían usado en la actividad 1 y comparando sus salidas, se puede concluir lo siguiente:

- En la primera línea del binario, **be 0f 7c b4 0e** corresponde a la primera instrucción **mov**. Luego,**ac 08 c0 74 04 cd 10 eb f7** corresponde a las instrucciones que se ejecutan dentro del for loop
- El mensaje “hello humans :)” se ubica entre las primeras 2 líneas del binario y su codificación en hexadecimal es: **48 65 6c 6c 6f 20 68 75 6d 61 6e 73 20 3a 29**
- La última línea **00000200** es la siguiente dirección de memoria después del último byte que se muestra (final de los 512 bytes del MBR)

```
ezerlich@ezerlich-Lenovo-V330-15IKB:~/Documents/TPS-SDC-2025/SDC-TP3-Epsilon/2-Hello_humans$ 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:   48                      rex.W
  10:   65 6c                   gs insb (%dx),%es:(%rdi)
  12:   6c                      insb   (%dx),%es:(%rdi)
  13:   6f                      outsl  %ds:(%rsi),(%dx)
  14:   20 68 75                and    %ch,0x75(%rax)
  17:   6d                      insl   (%dx),%es:(%rdi)
  18:   61                      (bad)
  19:   6e                      outsb  %ds:(%rsi),(%dx)
  1a:   73 20                   jae    3c <msg+0x2d>
  1c:   3a 29                   cmp    (%rcx),%ch
```

Figura 9: Output del comando objdump

```
ezerlich@ezerlich-Lenovo-V330-15IKB:~/Documents/TPS-SDC-2025/SDC-TP3-Epsilon/2-Hello_humans$ hd main.img
00000000  be 0f 7c b4 0e ac 08 c0  74 04 cd 10 eb f7 f4 48  |..|.....t......H|
00000010  65 6c 6c 6f 20 68 75 6d  61 6e 73 20 3a 29 00 66  |ello humans :).f|
00000020  2e 0f 1f 84 00 00 00 00  00 66 2e 0f 1f 84 00 00  |.........f......|
00000030  00 00 00 66 2e 0f 1f 84  00 00 00 00 00 66 2e 0f  |...f.........f..|
00000040  1f 84 00 00 00 00 00 66  2e 0f 1f 84 00 00 00 00  |.......f........|
00000050  00 66 2e 0f 1f 84 00 00  00 00 00 66 2e 0f 1f 84  |.f.........f....|
00000060  00 00 00 00 00 66 2e 0f  1f 84 00 00 00 00 00 66  |.....f.........f|
00000070  2e 0f 1f 84 00 00 00 00  00 66 2e 0f 1f 84 00 00  |.........f......|
00000080  00 00 00 66 2e 0f 1f 84  00 00 00 00 00 66 2e 0f  |...f.........f..|
00000090  1f 84 00 00 00 00 00 66  2e 0f 1f 84 00 00 00 00  |.......f........|
000000a0  00 66 2e 0f 1f 84 00 00  00 00 00 66 2e 0f 1f 84  |.f.........f....|
000000b0  00 00 00 00 00 66 2e 0f  1f 84 00 00 00 00 00 66  |.....f.........f|
000000c0  2e 0f 1f 84 00 00 00 00  00 66 2e 0f 1f 84 00 00  |.........f......|
000000d0  00 00 00 66 2e 0f 1f 84  00 00 00 00 00 66 2e 0f  |...f.........f..|
000000e0  1f 84 00 00 00 00 00 66  2e 0f 1f 84 00 00 00 00  |.......f........|
000000f0  00 66 2e 0f 1f 84 00 00  00 00 00 66 2e 0f 1f 84  |.f.........f....|
00000100  00 00 00 00 00 66 2e 0f  1f 84 00 00 00 00 00 66  |.....f.........f|
00000110  2e 0f 1f 84 00 00 00 00  00 66 2e 0f 1f 84 00 00  |.........f......|
00000120  00 00 00 66 2e 0f 1f 84  00 00 00 00 00 66 2e 0f  |...f.........f..|
00000130  1f 84 00 00 00 00 00 66  2e 0f 1f 84 00 00 00 00  |.......f........|
00000140  00 66 2e 0f 1f 84 00 00  00 00 00 66 2e 0f 1f 84  |.f.........f....|
00000150  00 00 00 00 00 66 2e 0f  1f 84 00 00 00 00 00 66  |.....f.........f|
00000160  2e 0f 1f 84 00 00 00 00  00 66 2e 0f 1f 84 00 00  |.........f......|
00000170  00 00 00 66 2e 0f 1f 84  00 00 00 00 00 66 2e 0f  |...f.........f..|
00000180  1f 84 00 00 00 00 00 66  2e 0f 1f 84 00 00 00 00  |.......f........|
00000190  00 66 2e 0f 1f 84 00 00  00 00 00 66 2e 0f 1f 84  |.f.........f....|
000001a0  00 00 00 00 00 66 2e 0f  1f 84 00 00 00 00 00 66  |.....f.........f|
000001b0  2e 0f 1f 84 00 00 00 00  00 66 2e 0f 1f 84 00 00  |.........f......|
000001c0  00 00 00 66 2e 0f 1f 84  00 00 00 00 00 66 2e 0f  |...f.........f..|
000001d0  1f 84 00 00 00 00 00 66  2e 0f 1f 84 00 00 00 00  |.......f........|
```

Figura 10: Output del comando hd

Grabación de main.img en un pendrive y arranque de una computadora:

Se grabó la imagen main.img en un USB drive con el fin de bootear una computadora desde el mismo con este comando:

```
sudo dd if=main.img of=/dev/sdX
```

donde **`sdX`** es el dispositivo que se desea escribir.

Se logró que la computadora corriera la imagen mostrando de manera parcial el mensaje “Hello humans :)”. No se encontró la razón por la cual no despliega el mensaje completo. El video puede observarse en el archivo **boot_form_USB.mp4** en la carpeta *“*hello_humans_and_*debugging”*

Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales
Sistemas de computación

Actividad 4: Depuración de ejecutables con llamadas a bios int

Para esta actividad, se hace una depuración de la imagen main.img que imprime **“hello humans :)”** de la actividad interior. Para ello, se hace uso del debugger gdb y una herramienta gráfica llamada **gdb dashboard** (https://github.com/cyrus-and/gdb-dashboard)

En primera instancia, se ejecuta la imagen mencionada desde una terminal con el siguiente comando:

```
qemu-system-x86_64  -fda main.img -boot a -s -S -monitor stdio
```

Esto permite inicializar la imagen en QEMU en pausa para poder llevar cabo el debugging.

Luego, desde otra pestaña de la terminal, se abre gdb y ejecuta el comando `target remote localhost:1234` para poder debuggear desde la terminal con **gdb dashboard** el programa en assembly

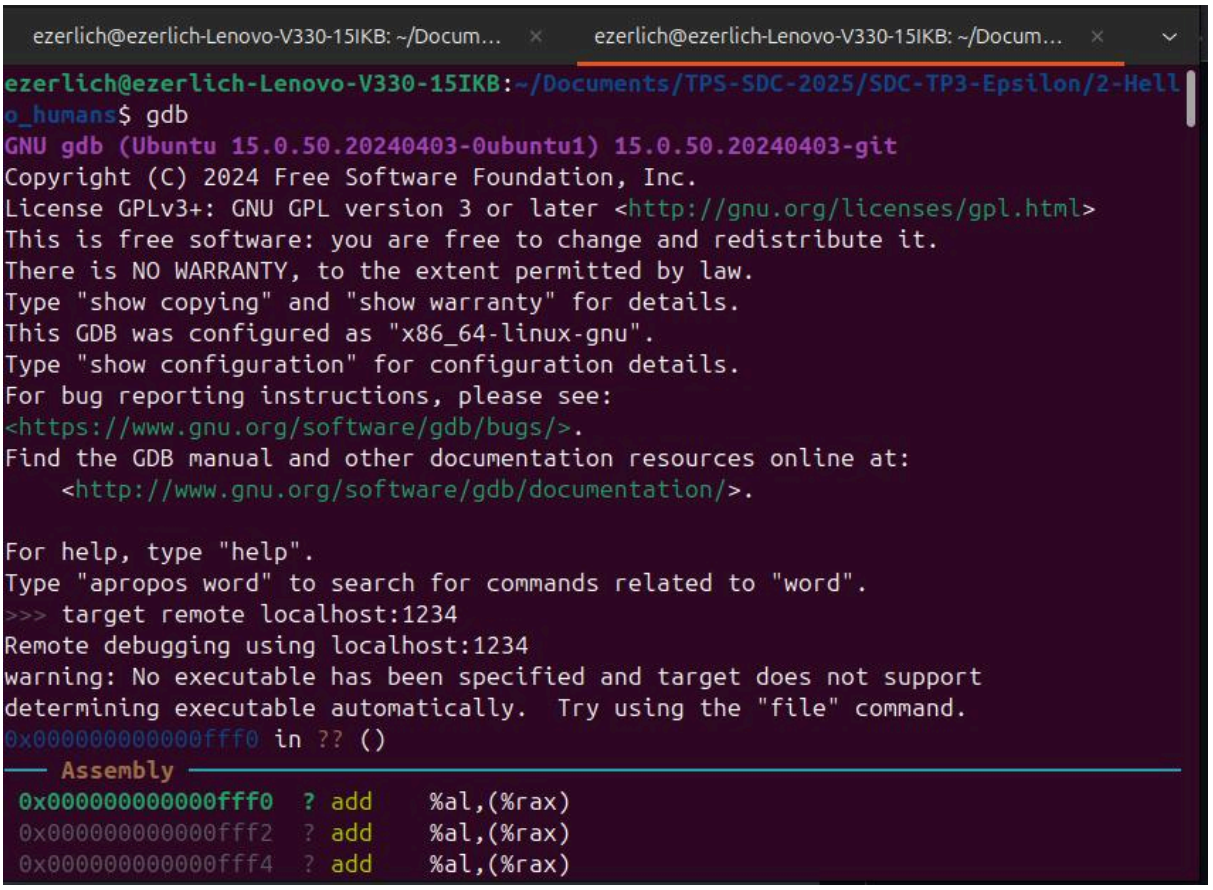

Figura 11: Inicialización gdb y ejecución del comando target remote localhost:1234

A continuación, se configuran 2 breakpoints. Uno en al comienzo del programa **(br *0x7c00)** y otro en la interrupción (**br *0x7c0c)**

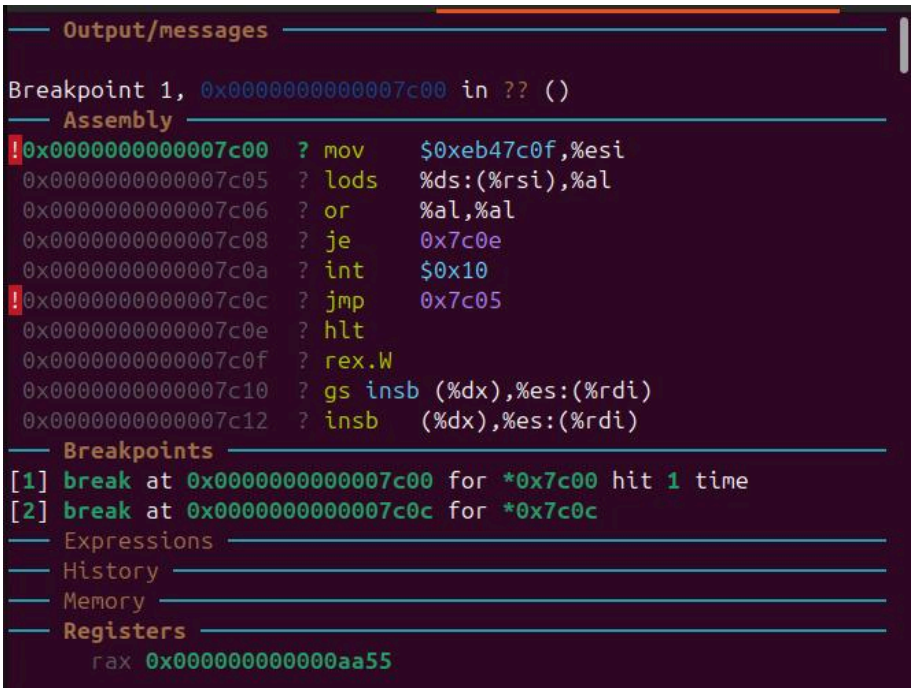

Figura 12: Breakpoints configurados

Finalmente, mediante sucesivas instrucciones **continue**, se puede ver como el programa se ejecuta paso a paso y se va completando el mensaje "**Hello Humans :)**" (Ver video **step_by_step_run_gdb.mp4** en la carpeta "hello_humans_and_*debugging"* del repositorio)

Actividad 5: Modo protegido

El modo real y el modo protegido son dos estados de funcionamiento de la CPU en arquitecturas x86. El modo real es el estado inicial al encender el procesador, donde el sistema operativo aún no ha tomado el control completo. En este modo, el acceso a la memoria y los dispositivos es directo y limitado, sin protección de memoria ni multitarea. Es simple pero carece de las características avanzadas de gestión y seguridad.

Por otro lado, el modo protegido es más avanzado y está diseñado para sistemas operativos modernos. Proporciona características como la protección de memoria mediante segmentación y paginación, gestión de privilegios mediante niveles de privilegio (anillos), y soporte para multitarea segura y eficiente. Permite a los sistemas operativos gestionar de manera más efectiva los recursos del sistema y proteger las aplicaciones entre sí. Es el modo predominante utilizado por los sistemas operativos actuales para ofrecer un entorno seguro y multitarea a los usuarios.

En modo protegido, la memoria se organiza usando un sistema de segmentación y, opcionalmente, paginación. A diferencia del modo real, donde la segmentación es fija (segmento:desplazamiento con 20 bits efectivos), en modo protegido cada segmento se define mediante una estructura llamada descriptor, que se almacena en una tabla especial llamada GDT (Global Descriptor Table). La GDT es una tabla que contiene descriptores de segmento. Cada descriptor define las características de un segmento de memoria: su dirección base, su límite (tamaño), los privilegios de acceso (como si es de código, datos, o pila, y a qué nivel de privilegio pertenece), y otros bits de control como el bit de presencia y el tipo de segmento. Cuando un programa accede a memoria usando un selector de segmento (por ejemplo, en un registro cs, ds, ss, etc.), ese selector se usa como índice para acceder a un descriptor dentro de la GDT. Luego, el procesador traduce la dirección lógica (selector:offset) a una dirección lineal usando el base del descriptor + offset. Esta dirección lineal puede ser usada directamente, o bien puede pasar por un mecanismo adicional llamado paginación, que permite dividir la memoria en páginas y asignarlas independientemente para una gestión más flexible y protegida.

A continuación mostraremos un código para pasar a modo protegido:

```
; boot_pm.asm
[BITS 16]
[ORG 0x7C00]           ; Punto de entrada del bootloader

start:
    cli                ; Deshabilitar interrupciones

    lgdt [gdt_descriptor] ; Cargar GDT

    mov eax, cr0       ; Leer CR0
    or eax, 1          ; Activar modo protegido (bit PE)
    mov cr0, eax

    jmp 0x08:protected_mode_start ; Salto lejano: cambia a modo protegido

; -----------------------------------
; GDT con solo un descriptor de código
; -----------------------------------
gdt_start:
    dq 0x0000000000000000       ; Descriptor nulo (index 0)

    ; Descriptor de código plano (index 1 → selector 0x08)
    dw 0xFFFF                   ; Límite (15:0)
    dw 0x0000                   ; Base (15:0)
    db 0x00                     ; Base (23:16)
    db 0x9A                     ; Acceso: código presente, ejecutable, legible
    db 0xCF                     ; Flags: gran = 1, 32-bit = 1, límite alto
    db 0x00                     ; Base (31:24)

gdt_end:

gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; Tamaño GDT - 1
    dd gdt_start               ; Dirección base GDT

; -------------------------
; Código en modo protegido
; -------------------------
[BITS 32]
protected_mode_start:
    hlt                ; Detiene la CPU (útil para testear que llegamos)

; -------------------------
; Boot sector (512 bytes)
; -------------------------
times 510-($-$$) db 0
dw 0xAA55              ; Firma del bootloader
```

Figura 13: Código para pasar a modo protegido

Explicación del código boot_pm.asm:

[BITS 16]: El procesador arranca en modo real (16 bits).

[ORG 0x7C00]: Indica a NASM que este código se cargará en 0x7C00, la dirección donde la BIOS carga el primer sector del disco (512 bytes).

cli: Desactiva las interrupciones. Esto es crítico para evitar interrupciones mientras cambiamos al modo protegido, ya que en ese modo la BIOS no puede manejar interrupciones.

lgdt: Carga la dirección de la tabla de descriptores globales (GDT), necesaria para habilitar segmentación en modo protegido.

En esta porción de código manipula el registro de control CR0: Bit 0 (PE, Protection Enable): Se pone en 1 ⟶ activa modo protegido. Esta es la transición más importante del modo real al modo protegido.

```
mov eax, cr0        ; Leer CR0
or eax, 1           ; Activar modo protegido (bit PE)
mov cr0, eax
```

jmp 0x08:protected_mode_start: Salto lejano a la dirección “protected_mode_start”, usando el selector de segmento 0x08 (el código plano en la GDT). Este salto actualiza CS (code segment) con el nuevo descriptor y termina la transición al modo protegido.

[BITS 32]: Ahora estamos en modo protegido (el procesador interpreta instrucciones de 32 bits).

hlt: Instrucción que detiene la CPU. Se usa aquí como prueba de que llegamos al modo protegido exitosamente.

times 510-($-$$) db 0: Rellena el archivo hasta los 510 bytes.

dw 0xAA55: Firma del bootloader (sector de arranque). La BIOS requiere este valor en los últimos 2 bytes para considerar el sector como válido.

Ahora vamos a analizar en detalle la parte de la tabla de descriptores. En modo protegido, los descriptores de segmento son estructuras de 8 bytes (64 bits) que definen cómo se accede a una región de memoria. Cada descriptor contiene información como:

- Base del segmento (dirección física desde donde comienza).
- Límite (cuánto mide el segmento).
- Tipo y privilegios (si es código o datos, si es ejecutable o no, nivel de privilegio, etc.).
- Flags para granularidad y tamaño de instrucción.

Estos descriptores son almacenados en la GDT (Global Descriptor Table) y cada vez que se carga un segmento , el selector apunta a uno de estos descriptores por índice. Por ejemplo, si ponés “mov ax, 0x10” y después “mov ds, ax”, estás diciendo:"Quiero que el registro ds use el descriptor número 2 de la GDT (índice 2, porque 0x10 >> 3 = 2)".

Un selector es el valor que se carga en registros de segmento como ds, cs, ss, etc. Este valor tiene tres partes:

- Índice: número que indica cuál descriptor queremos usar dentro de la GDT.
- TI (Table Indicator): 0 si es GDT, 1 si es LDT (en nuestro caso siempre es 0).
- RPL (Requested Privilege Level): nivel de privilegio (por ahora lo ignoramos).

Estructura de un descriptor de segmentos 8 bytes (64b) :

Bits	Nombre del campo	Descripción
0–15	Límite (15:0)	Parte baja del límite del segmento.
16–31	Base (15:0)	Parte baja de la dirección base del segmento.
32–39	Base (23:16)	Parte media de la dirección base.
40–43	Tipo	Tipo de segmento: código, datos, si es ejecutable, legible, etc.
44	S (descriptor de sistema)	1 = segmento de código o datos, 0 = descriptor de sistema (TSS, LDT, etc.).
45–46	DPL (privilegio)	Nivel de privilegio (0 = kernel, 3 = user).
47	P (presente)	1 = segmento válido en memoria.
48–51	Límite (19:16)	Parte alta del límite.
52	AVL (uso del sistema)	Bit disponible para uso del sistema operativo.
53	L (modo largo)	1 = segmento de 64 bits (modo largo). En 32 bits debe ser 0.
54	D/B (tamaño del operando)	1 = segmento de 32 bits, 0 = 16 bits.
55	G (granularidad)	1 = límite se multiplica por 4KB, 0 = límite en bytes.
56–63	Base (31:24)	Parte alta de la dirección base del segmento.

En nuestro caso, el segmento de código queda definido con las siguientes características:

```
; ---------------------------------
; GDT con solo un descriptor de código
; ---------------------------------
gdt_start:
    dq 0x0000000000000000       ; Descriptor nulo (index 0)

    ; Descriptor de código plano (index 1 → selector 0x08)
    dw 0xFFFF                   ; Límite (15:0)
    dw 0x0000                   ; Base (15:0)
    db 0x00                     ; Base (23:16)
    db 0x9A                     ; Acceso: código presente, ejecutable, legible
    db 0xCF                     ; Flags: gran = 1, 32-bit = 1, límite alto
    db 0x00                     ; Base (31:24)

gdt_end:

gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; Tamaño GDT - 1
    dd gdt_start               ; Dirección base GDT
```

Campo	Valor	Significado
Límite	`0xFFFFF` (20 bits)	Límite máximo (1 MB × 4 KB = 4 GB si G=1)
Base	`0x00000000`	El segmento comienza en la dirección 0
Acceso (`0x9A`)	`10011010b`	Segmento presente, ejecutable, legible, nivel de privilegio 0
Flags (`0xCF`)	`11001111b`	Granularidad (G=1), 32-bit (D=1), y los 4 bits altos del límite

En las dos últimas líneas se calcula el tamaño y la dirección de la tabla, esto se pasa como argumento a la instrucción “lgdt”.

Nota: Las instrucciones “db”, “dw”, “dd” y “dq” se usan para definir datos de distintos tamaños,1 byte, 2 byte, etc.

Un último detalle es que el descriptor en el índice 0 siempre debe estar vacío (cero) por reglas de la arquitectura x86. Si alguna operación trata de usar el selector 0x00, el procesador detecta que es inválido y genera una excepción (error).

Ahora vamos a ver un ejemplo de una GDT con dos descriptores uno para código y otro para datos:

```
; boot_pm_2.asm
[BITS 16]
[ORG 0x7C00]                ; Dirección de carga del BIOS

start:
    cli                     ; Deshabilita interrupciones

    lgdt [gdt_descriptor]   ; Cargar dirección y tamaño de la GDT

    mov eax, cr0
    or eax, 1               ; Setear bit PE (Protection Enable)
    mov cr0, eax

    jmp 0x08:protected_mode_start ; Salto lejano al código 32 bits (selector
código = 0x08)

; -------------------------
; GDT con dos descriptores:
; -------------------------
gdt_start:
    dq 0x0000000000000000   ; Descriptor nulo (indice 0)

    ; Descriptor de código (indice 1 → selector 0x08)
    dw 0xFFFF               ; Límite (15:0)
    dw 0x0000               ; Base (15:0)
    db 0x00                 ; Base (23:16)
    db 0x9A                 ; Acceso: ejecutable, presente, RPL=0
    db 0xCF                 ; Flags: Granularidad=4KB, 32-bit
    db 0x00                 ; Base (31:24)

    ; Descriptor de datos (indice 2 → selector 0x10)
    dw 0xFFFF               ; Límite (15:0)
    dw 0x0000               ; Base (15:0) → luego sobreescribimos con base
0x00100000
    db 0x10                 ; Base (23:16) = 0x00100000 >> 16
    db 0x92                 ; Acceso: datos legible/escribible, presente, RPL=0
    db 0xCF                 ; Flags: Granularidad=4KB, 32-bit
    db 0x00                 ; Base (31:24)

gdt_end:

gdt_descriptor:
    dw gdt_end - gdt_start - 1
    dd gdt_start
```

```
; ---------------------------------
; Código ejecutado en modo protegido
; ---------------------------------
[BITS 32]
protected_mode_start:
    ; Cargar el selector de datos (0x10 = indice 2 << 3)
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    ; Inicializar pila (dentro del segmento de datos)
    mov esp, 0x00102000     ; Dirección dentro del segmento de datos

    ; Código de prueba
    mov dword [0x00100000], 0x12345678 ; Escribir en el segmento de datos (1 MB)

    hlt                     ; Detener CPU

; ---------------------------
; Boot sector (512 bytes)
; ---------------------------
times 510 - ($ - $$) db 0
dw 0xAA55
```

Figura 14: Código Modo protegido con GDT para 2 descriptores

Este código funciona exactamente igual que el anterior con la salvedad que en la tabla de descriptores tenemos un segundo descriptor de segmentos, en las siguientes líneas podemos ver como asigna a los selectores el índice que corresponde a los segmentos de datos:

```
protected_mode_start:
    ; Cargar el selector de datos (0x10 = indice 2 << 3)
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
```

Como vimos anteriormente el selector tiene 13 bits dedicados al índice, como cada descriptor ocupa 8 bytes por lo tanto los índices de la tabla GDT se desplazan cada 8 posiciones (indice 0 0x00, indice 1 0x08, indice 2 0x10).

Selector	Índice GDT	Descripción
0x00	0	Descriptor nulo (inválido)
0x08	1	Código (base 0x00000000)
0x10	2	Datos (base 0x00100000)

Los registros de segmento (ds, ss, es, etc.) solo referencian descriptores en la GDT. Cada descriptor describe un rango de direcciones de memoria (segmento), con una base y un límite. Si todos apuntan al mismo descriptor (por ejemplo, índice 2 → selector 0x10), entonces todos están usando el mismo segmento de memoria. Pero cada registro (ds, ss, etc.) tiene su función específica:

- ds: datos generales
- ss: pila
- es, fs, gs: a veces para buffers o estructuras paralelas

Aunque apunten al mismo segmento, el uso que hace la CPU de cada registro es distinto, y no se pisan entre sí. Solo usarías la misma base/limite para simplificar, como en un sistema plano.

Es fundamental configurar ss antes de usar la pila, para que las instrucciones que dependan de la pila funcionen bien (como push, call, ret).

mov esp, 0x90000: Se establece el stack pointer (esp) a la dirección virtual 0x90000 (576 KB). Como ya se cargo ss antes, esta dirección apunta al segmento de datos definido por el descriptor en la GDT. Esto marca el tope de la pila, desde donde empezarán a decrecer los valores (push y llamadas). La dirección 0x9000 es una dirección segura y libre en RAM, bien por encima del código del bootloader (que empieza en 0x7C00) y lejos de áreas peligrosas como la GDT o el BIOS.

Por último vamos a realizar cambios en el descriptor de segmento de datos, de modo tal que sea solo lectura, y posteriormente vamos a intentar escribir sobre dicho segmento para ver qué ocurre, los cambios que hay que implementar en el código son los siguientes:

Reemplazar:

db 0x92 ; tipo: datos, lectura/escritura

db 0x90 ; tipo: datos, solo lectura

Agregar:

mov dword [0x1000], 0xDEADBEEF

Ahora vamos a compilar el programa para generar el binario, para posteriormente simularlo en QEMU y ver lo que pasa:

```
mariano-campos@mariano-campos-HP-Laptop-14-dk1xxx:~/Escritorio/SdC/Lab3/develop/
test_modo_protegido$ nasm -f bin -o boot_pm_2.bin boot_pm_2.asm
mariano-campos@mariano-campos-HP-Laptop-14-dk1xxx:~/Escritorio/SdC/Lab3/develop/
test_modo_protegido$ qemu-system-i386 -fda boot_pm_2.bin -s -S
WARNING: Image format was not specified for 'boot_pm_2.bin' and probing guessed
raw.
         Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
```

Seteamos la arquitectura y el local-host en gdb:

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) set architecture i386
The target architecture is set to "i386".
(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.
0x0000fff0 in ?? ()
```

Realizamos pasos con la instrucción “si” hasta notar un salto grande en la dirección de memoria, y hacemos un comando “info registers” para ver que ocurrió:

```
(gdb) info registers
eax            0x0                 0
ecx            0x0                 0
edx            0x663               1635
ebx            0x0                 0
esp            0x0                 0x0
ebp            0x0                 0x0
esi            0x0                 0
edi            0x0                 0
eip            0xe05b              0xe05b
eflags         0x2                 [ IOPL=0 ]
cs             0xf000              61440
ss             0x0                 0
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x0                 0
gs_base        0x0                 0
k_gs_base      0x0                 0
cr0            0x60000010          [ CD NW ET ]
cr2            0x0                 0
cr3            0x0                 [ PDBR=0 PCID=0 ]
cr4            0x0                 [ ]
```

cs:eip = F000:E05B → Dirección física 0xFFFF0, justo en el área donde reside la BIOS. Esto es el vector de reset, y solo se salta allí en situaciones graves.

cr0 tiene el bit PE activado (cr0 & 0x1 = 1) → estás en modo protegido.

Pero el **cs = 0xF000** y **eip = 0xE05B** son de modo real, no protegido. Esto implica que el CPU ha hecho un fallback.

esp = 0 → El stack no está configurado, otro signo de que el sistema fue "sacado" de contexto de ejecución normal.

Explicación del error: El procesador ha entrado en modo protegido (cr0 muestra el bit PE activado), pero se intentó realizar una operación inválida como ejecutar código desde un segmento no ejecutable. En modo protegido, estas acciones disparan excepciones como la #GP (General Protection Fault) o #PF (Page Fault). Dado que no se configuró una tabla IDT válida ni un manejador para esas excepciones, el procesador no puede entregar el control al sistema operativo y en su lugar realiza un salto a la dirección de fallback 0xFFFF0, donde típicamente comienza la BIOS. Este comportamiento representa un fallo fatal, impidiendo continuar la ejecución del sistema y mostrando que la transición a modo protegido o el acceso a memoria no fue manejado correctamente.

Nota: La IDT (Interrupt Descriptor Table) es una estructura fundamental en los sistemas x86 que, en modo protegido, permite al procesador manejar interrupciones y excepciones de forma controlada y segura. Esta tabla contiene hasta 256 entradas, cada una correspondiente a un número de interrupción o excepción, e incluye información esencial como la dirección del manejador (offset), el selector de segmento (CS) y atributos que definen el tipo de puerta (por ejemplo, interrupt gate o trap gate) y el nivel de privilegio requerido. Cuando ocurre una interrupción —ya sea por hardware (como un teclado) o por software (como una excepción por división por cero)— el procesador consulta la IDT para saber a qué dirección de memoria debe saltar para ejecutar el código que maneja dicha interrupción. En ausencia de una IDT correctamente configurada, el procesador no puede gestionar errores ni eventos externos, lo que puede derivar en comportamientos impredecibles como reinicios, cuelgues del sistema o saltos a direcciones inválidas. Por ello, la instalación de la IDT es un paso obligatorio e inicial en la configuración del modo protegido de un sistema operativo o entorno de bajo nivel.