



FACULTAD DE CIENCIAS EXACTAS, FÍSICAS y NATURALES

SISTEMAS DE COMPUTACIÓN

TP3: MODO PROTEGIDO

Nombre	DNI	Carrera
Facundo Nahuel Galvagno	40815088	IE
Tomás Ignacio Corbalán	38503847	ICOMP
Hernan Imanol Rodriguez	41808427	ICOMP

Modo Protegido

Los procesadores x86 mantienen compatibilidad con sus antecesores y para agregar nuevas funcionalidades deben ir “evolucionando” en el tiempo durante el proceso de arranque. Todos los CPUs x86 comienzan en modo real en el momento de carga (boot time) para asegurar compatibilidad hacia atrás, en cuanto se los energiza se comportan de manera muy primitiva, luego mediante comandos se los hace evolucionar hasta poder obtener la máxima cantidad de prestaciones posibles. El modo protegido es un modo operacional de los CPUs compatibles x86 de la serie 80286 y posteriores. Este modo es el primer salto evolutivo de los x86. El modo protegido tiene un número de nuevas características diseñadas para mejorar la multitarea y la estabilidad del sistema, tales como la protección de memoria, y soporte de hardware para memoria virtual como también la conmutación de tareas.

En este trabajo práctico ejecutaremos un trozo de código que configura nuestro procesador para llevarlo desde el modo real al modo protegido.

Creación de una imagen bootable

Creamos un sector de arranque MBR

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

La función printf usando la cadena de formato \364%509s\125\252 crea un string que empieza con 0xF4 (instrucción halt), tiene 509 bytes nulos y luego termina con 0X55AA (firma de arranque). Esto nos crea una secuencia de 512 bytes de tamaño que es el estándar para una partición MBR.

Podemos verificar que 0xF4 corresponde a la instrucción hlt

```
echo hlt > a.S
```

A su vez, si compilamos el archivo assembler a.S

```
as -o a.o a.S
```

y luego lo decompilamos con objdump

```
objdump -S a.o
```

podemos verificar que 0xF4 corresponde a la instrucción hlt.

```
facundog@FacundoNotebook:~/Documents/sdc-modo-prottegido-real$ objdump -S a.o  
  
a.o:      file format elf64-x86-64
```

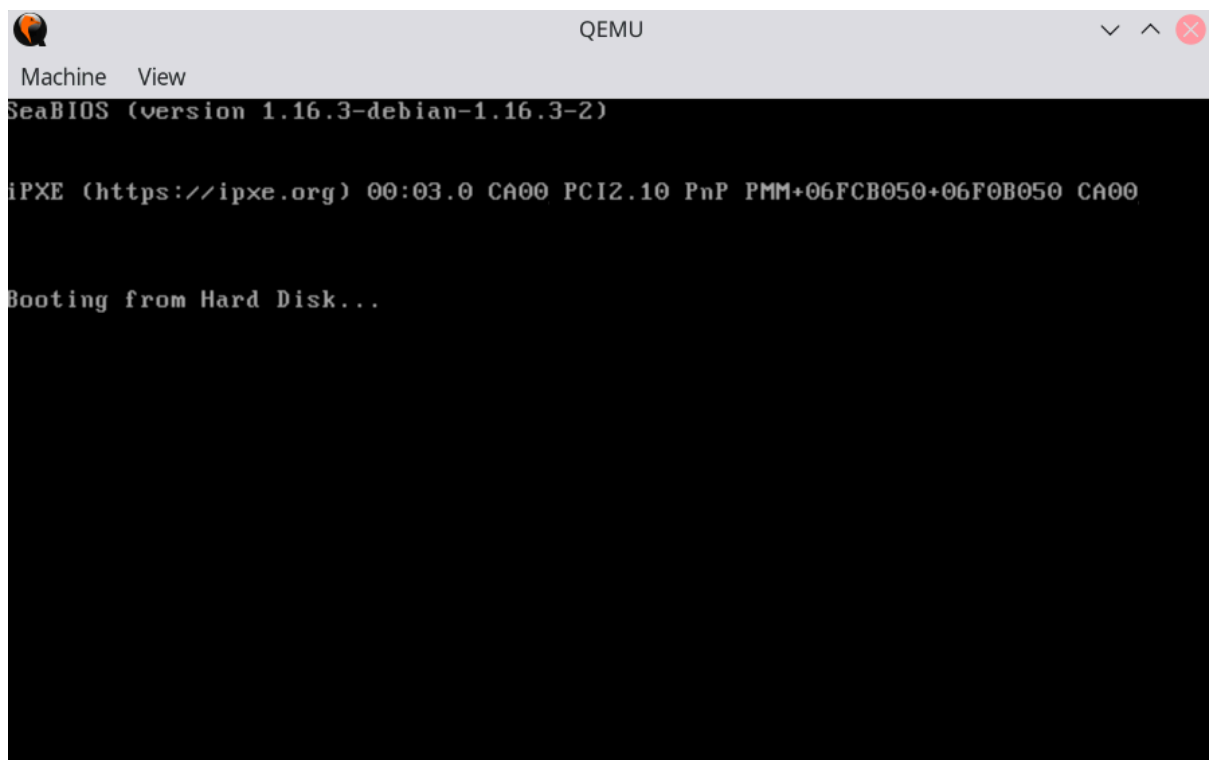
```
Disassembly of section .text:

0000000000000000 <.text>:
    0:  f4                      hlt
facundog@FacundoNotebook:~/Documents/sdc-modo-prottegido-real/00SimpleMBR
$ hd main.img

00000000  f4 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 |
|
*
000001f0  20 20 20 20 20 20 20 20 20 20 20 20 20 55 aa |
U.|
00000200
```

Probando la imagen en qemu

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



También podemos probar la imagen en hardware real. Para esto fue necesario deshabilitar el CSM, puesto que no es posible bootear directamente en modo BIOS desde un USB en algunas PC modernas.



UEFI y coreboot

¿Qué es UEFI y cómo puedo usarlo? UEFI (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 presentado en los PC IBM como ROM BIOS de PC IBM.

¿Cómo se usa? Se pueden programar aplicaciones que corran en UEFI usando C. Una guía para programar aplicaciones en UEFI es la especificación UEFI.

Un ejemplo de una de las funciones que se pueden usar es:

```
SystemTable->ConOut->OutputString SystemTable->ConOut, L"Hola  
Mundo!\r\n");
```

¿Existen bugs de UEFI que puedan ser explotados? Sí, UEFI ha sido objetivo de varios ataques importantes debido a su bajo nivel de ejecución y persistencia:

- LoJax: Primer rootkit UEFI usado en la práctica. Persistía incluso tras formatear el disco.
- BlackLotus: Malware que evade Secure Boot explotando fallas en UEFI y ejecutándose antes de Windows.
- MoonBounce: Malware detectado en la SPI flash que no requiere un disco duro para persistir.

¿Qué es CSME y MEBx? CSME (Converged Security and Management Engine) Es un subsistema dentro de los procesadores Intel que corre independientemente del sistema operativo. Se encarga de tareas como:

- Verificación del arranque seguro.
- Protección de claves.
- Administración remota.

Intel MEBx (Management Engine BIOS Extension) Es la interfaz de configuración del Intel ME. Puede usarse para:

- Habilitar funciones como AMT (Intel Active Management Technology).
- Configurar acceso remoto o contraseñas del firmware.

¿Qué es coreboot? ¿Qué productos lo incorporan? coreboot es un firmware de código abierto que reemplaza al BIOS propietario. Su objetivo es inicializar el hardware de forma rápida y libre de componentes innecesarios. Algunos dispositivos que usan coreboot son:

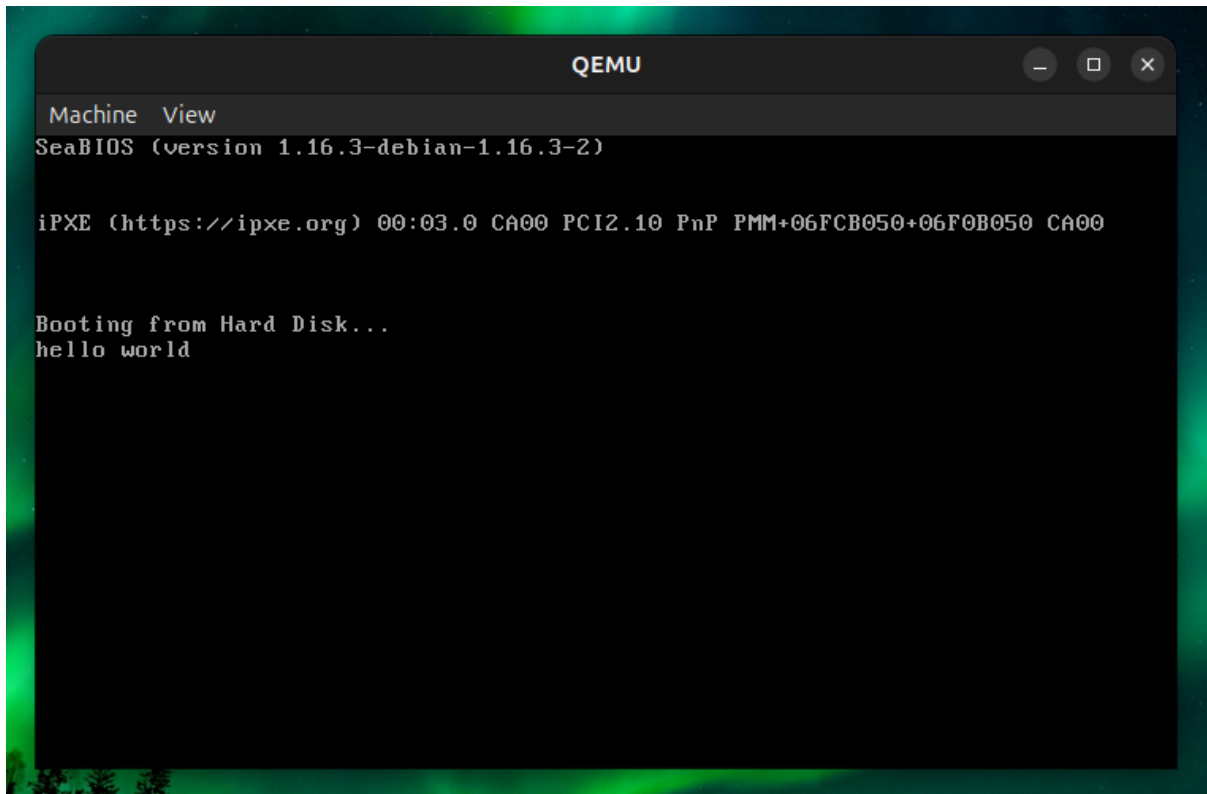
- Chromebooks (muchos modelos).
- Equipos de marcas como System76, Purism, TUXEDO Computers.
- Algunos servidores y dispositivos de red.

Ventajas de coreboot:

- Código abierto: Transparente y auditable. Mayor control sobre la privacidad (sin Intel ME si se desactiva).
- Modularidad: Mediante el uso de payloads podemos modificar directamente lo que se ejecuta después de ejecutar el firmware de encendido.

Linker

Ejecución del ejemplo 01HelloWorld con QEMU:



¿Qué es un linker? ¿qué hace ?

En el contexto de ejecutar un programa directamente sobre arquitectura x86 sin un sistema operativo, el **linker** (enlazador) es la herramienta que construye el binario final, con direcciones de memoria absolutas, para ser cargado por el hardware o el firmware (como el BIOS). Este proceso es crucial porque, sin un sistema operativo, con un loader que gestione la memoria, interrupciones o carga de archivos, es el binario el que debe estar completamente preparado para ser cargado por el BIOS y ejecutado desde una dirección específica, como `0x7C00` en el caso de un bootloader.

En un entorno bare-metal, se utiliza frecuentemente un **script de linker** escrito en command language específico (como el lenguaje de scripts de ld en GNU), que especifica explícitamente dónde deben colocarse las secciones del programa (como `.text`, `.data`, `.bss`) y cuál es la dirección base del código. Esto asegura que, al momento de ejecución, la CPU interprete correctamente las instrucciones y acceda a los datos en las ubicaciones correctas. El linker también resuelve símbolos entre módulos y garantiza que todas las

referencias de memoria estén completamente definidas, algo imprescindible al no contar con un cargador de programas ni direccionamiento dinámico.

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

```
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)
    }
}
```

En el script del linker vemos definido el comando **SECTIONS**, este controla exactamente dónde se ubican las secciones de entrada en las secciones de salida, su orden en el archivo de salida y a qué secciones de salida se asignan. Las sentencias dentro del comando **SECTIONS** pueden realizar una de estas tres cosas:

1. definir el punto de entrada;
2. asignar un valor a un símbolo;
3. describir la ubicación de una sección de salida con nombre y qué secciones de entrada se incluyen en ella.

Para nuestro caso vemos que se define el punto de entrada **"_start = . ;" (1.)** mediante la asignación de un valor al símbolo **". = 0x7c00;" (2.)**

El lenguaje de comandos del linker incluye otro comando específico para definir la primera instrucción ejecutable en un archivo de salida (entry point). El comando puede colocarse como un comando independiente en el archivo de comandos o entre las definiciones de sección dentro del comando.

El comando **ENTRY** es solo una de las varias maneras de elegir el punto de entrada. Puede indicarlo de cualquiera de las siguientes maneras (mostradas en orden descendente de prioridad, los métodos superiores en la lista anulan los métodos inferiores):

- La opción **'-e'** de entrada en la línea de comandos;
- El comando **ENTRY** en un script de control del linker;

- El valor del símbolo `"start"`, si está presente;
- La dirección del primer byte de la sección `.text`, si está presente;
- La dirección `0x0`.

En nuestro caso lo que se hace es generar un punto de entrada con una sentencia de asignación definiendo el símbolo `"start"`:

```
start = 0x7c00;
```

Se podría usar una dirección absoluta como en la línea anterior, pero se puede usar cualquier expresión. Por ejemplo, si los archivos de objeto de entrada usan otra convención de nombres de símbolos para el punto de entrada, se puede simplemente asignar el valor del símbolo que contenga la dirección de inicio para start, y este es el caso de nuestro script linker.

```
start = . ;
```

Ahora bien, la dirección `"0x7C00"` es la dirección de memoria donde la BIOS carga el MBR (Registro de Arranque Maestro, el primer sector del disco duro/unidad de disquete). El desarrollador del sistema operativo o del gestor de arranque debe asumir que sus códigos ensambladores están cargados y comienzan desde `0x7C00`.

Compare la salida de objdump con hd, verifique donde fue colocado el programa dentro de la imagen.

Se ejecutó `hd main.img`:


```

Terminal
tomas-ignacio-corbalan@:/mnt/compartido/Carpetas/Mi Carpeta/develop/tomas-ignacio-corbalan@:/mnt/compartido/Carpetas/Mi Carpeta/develop/facultad/SdeC/tp3-SdeC-2025/protected-mode-sdc/01HelloWorld$ hd main.img
00000000  be 0f 7c b4 0e ac 08 c0 74 04 cd 10 eb f7 f4 68 |..|....t.....h|
00000010  65 6c 6c 6f 20 77 6f 72 6c 64 00 66 2e 0f 1f 84 |ello world.f...|
00000020  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 |....f.....f|
00000030  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
00000040  00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f |...f.....f..|
00000050  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 |.....f.....|
00000060  00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
00000070  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 |....f.....f|
00000080  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
00000090  00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f |...f.....f..|
000000a0  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 |.....f.....|
000000b0  00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
000000c0  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 |....f.....f|
000000d0  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
000000e0  00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f |...f.....f..|
000000f0  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 |.....f.....|
00000100  00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
00000110  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 |....f.....f|
00000120  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
00000130  00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f |...f.....f..|
00000140  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 |.....f.....|
00000150  00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
00000160  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 |....f.....f|
00000170  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
00000180  00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f |...f.....f..|
00000190  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 |.....f.....|
000001a0  00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
000001b0  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 |....f.....f|
000001c0  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
000001d0  00 00 00 66 2e 0f 1f 84 00 00 00 00 00 66 2e 0f |...f.....f..|
000001e0  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 |.....f.....|
000001f0  00 66 2e 0f 1f 84 00 00 00 00 00 0f 1f 00 55 aa |.f.....U.|
tomas-ignacio-corbalan@:/mnt/compartido/Carpetas/Mi Carpeta/develop/facultad/SdeC/t
tomas-ignacio-corbalan@:/mnt/compartido/Carpetas/Mi Carpeta/develop/facultad/SdeC/t
p3-SdeC-2025/protected-mode-sdc/01HelloWorld$ 

```

Se observa que:

- El código empieza en el byte `0x00`, como debe ser en un boot sector.
- En `0x0F` comienza "hello world" en ASCII, seguido por un `00` (el null terminator de .asciz).
- En `0x1FE` está el magic number `0xAA55` (ordenado little endian: `55 aa`), que indica que el BIOS puede bootear desde esta imagen.
- El archivo completo tiene `512` bytes.

Se ejecutó `objdump -D -b binary -m i386 --adjust-vma=0x7C00 main.img`:

Con `--adjust-vma=0x7C00` se le especifica a objdump que ese binario se cargará en la dirección `0x7C00`, que es la que el BIOS usa al leer el primer sector del disco. Sin especificar este parámetro las direcciones irían desde `0x000` a `0x1FF` (512 bytes).

```
tomas-ignacio-corbalan@:/mnt/compartido/Carpetas/Mi Carpeta/develop/facultad/SdeC/t
p3-SdeC-2025/protected-mode-sdc/01HelloWorld$ objdump -D -b binary -m i386 --adjust
-vma=0x7C00 main.img

main.img:      formato del fichero binary

Desensamblado de la sección .data:

00007c00 <.data>:
 7c00:    be 0f 7c b4 0e      mov     $0xeb47c0f,%esi
 7c05:    ac                  lods    %ds:(%esi),%al
 7c06:    08 c0               or      %al,%al
 7c08:    74 04               je      0x7c0e
 7c0a:    cd 10               int     $0x10
 7c0c:    eb f7               jmp     0x7c05
 7c0e:    f4                  hlt
 7c0f:    68 65 6c 6c 6f      push    $0x6f6c6c65
 7c14:    20 77 6f             and     %dh,0x6f(%edi)
 7c17:    72 6c               jb      0x7c85
 7c19:    64 00 66 2e          add     %ah,%fs:0x2e(%esi)
 7c1d:    0f 1f 84 00 00 00    nopl    0x0(%eax,%eax,1)
 7c24:    00
 7c25:    66 2e 0f 1f 84 00    nopw    %cs:0x0(%eax,%eax,1)

 7df1:    66 2e 0f 1f 84 00    nopw    %cs:0x0(%eax,%eax,1)
 7df8:    00 00 00
 7dfb:    0f 1f 00             nopl    (%eax)
 7dfe:    55                   push    %ebp
 7dff:    aa                   stos    %al,%es:(%edi)

tomas-ignacio-corbalan@:/mnt/compartido/Carpetas/Mi Carpeta/develop/facultad/SdeC/t
p3-SdeC-2025/protected-mode-sdc/01HelloWorld$
```

Se observa que:

- Que el primer byte de código (`mov $0x0e, %ah`) aparece en offset `0` de `hd`, y en la dirección `0x7C00` en `objdump`.
- La cadena "hello world" comienza en la dirección de memoria `0x7C0F`.
- Que el valor `0x7C0F` está exactamente en el offset `510` (`0x1FE`).
- Que el código se alinea con el `"start symbol"` del script del linker (`. = 0x7c00`).

¿Para qué se utiliza la opción `--oformat binary` en el linker?

La opción `--oformat binary` en el linker (como en `ld` de GNU) se utiliza para generar un

archivo binario plano (raw binary), es decir, un archivo que contiene únicamente los datos binarios del programa, sin encabezados, metadatos ni estructuras propias de formatos ejecutables como ELF o PE.

En un entorno como x86 con BIOS y sin sistema operativo, se requiere que el código esté en un formato que el BIOS pueda cargar directamente en memoria y ejecutar. El BIOS, al cargar el primer sector del disco (512 bytes) en 0x7C00, espera un binario plano, no un archivo ELF ni ningún formato de ejecutable de alto nivel. Si no se usa `--oformat binary`, el linker incluiría encabezados del formato por defecto (como ELF), lo cual haría que el código no sea ejecutable directamente por la CPU.

Depuración de ejecutables con llamadas a bios int

Se ejecutó el ejemplo 01/HelloWorld con QEMU junto con GDB para depuración con llamadas a BIOS mediante int 0x10 para impresión de la cadena de caracteres "hello world".

Ejecución de QEMU:

```
tomas-ignacio-corbalan@:/mnt/compartido/Carpetas/Mi Carpeta/dev
elop/facultad/SdeC/tp3-SdeC-2025/protected-mode-sdc/01HelloWorl
d$ qemu-system-i386 -fda main.img -boot a -s -S -monitor stdio
WARNING: Image format was not specified for 'main.img' and prob
ing guessed raw.
      Automatically detecting the format is dangerous for ra
w images, write operations on block 0 will be restricted.
      Specify the 'raw' format explicitly to remove the rest
rictions.
QEMU 8.2.2 monitor - type 'help' for more information
(qemu) █
```

Significado:

- `-s` = abre un servidor GDB en el puerto 1234.
- `-S` = no arranca la CPU del emulador hasta que GDB se lo diga.
- `-monitor stdio` = muestra la consola de QEMU.

Ejecución de GDB:

```
tomas-ignacio-corbalan@:/mnt/compartido/Carpetas/Mi Carpeta/dev
elop/facultad/SdeC/tp3-SdeC-2025/protected-mode-sdc/01HelloWorl
d$ gdb -ex "target remote localhost:1234"
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
```

`-ex "target remote localhost:1234"`: Para debug de puerto localhost:1234.

Luego, se setea la arquitectura i8086.

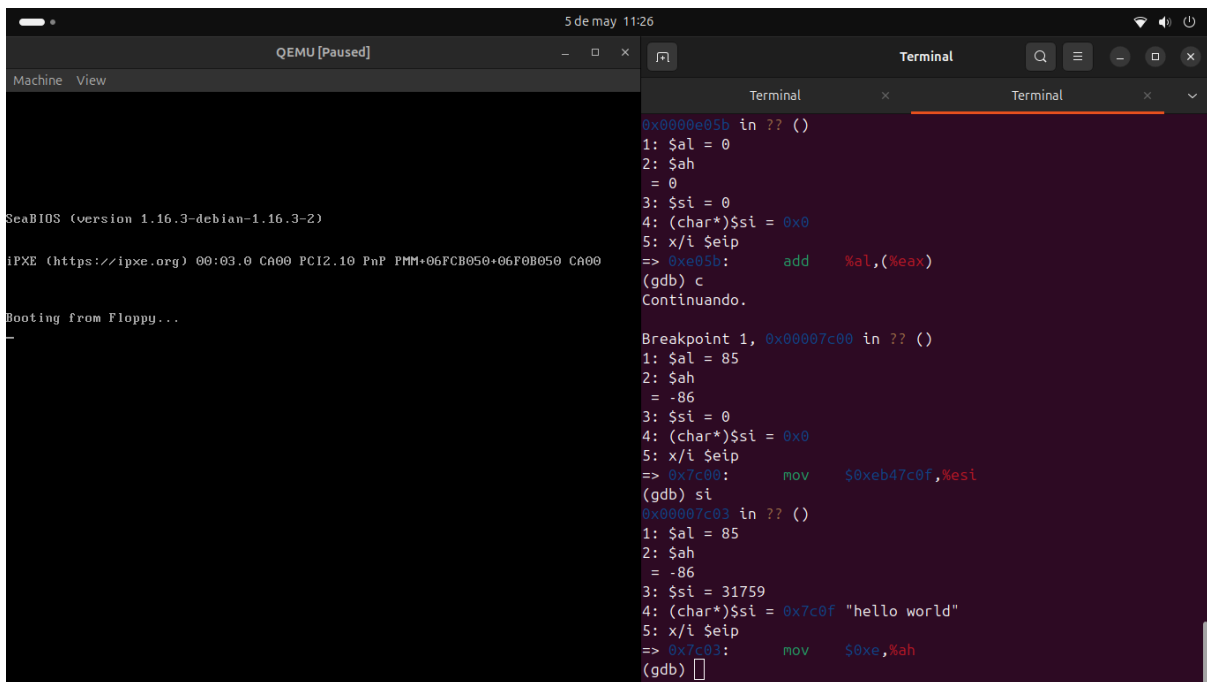
Se colocan los break points:

- en `0x7c00` (entry point al boot sector)
- en `0x7c0c` (dirección posterior a la interrupción)

Y se fijan en el display los registros `al`, `ah`, `si` y `*si` para observar como varían en el loop.

```
(gdb) set architecture i8086
The target architecture is set to "i8086".
(gdb) br *0x7c00
Punto de interrupción 1 at 0x7c00
(gdb) break *0x7c0c
Punto de interrupción 2 at 0x7c0c
(gdb) display $al
1: $al = 0
(gdb) display $ah
2: $ah
   = 0
(gdb) display $si
3: $si = 0
(gdb) display (char*)$si
4: (char*)$si = 0x0
(gdb) display/i $eip
5: x/i $eip
```

El emulador se queda en espera de que comience el debug con GDB.



```

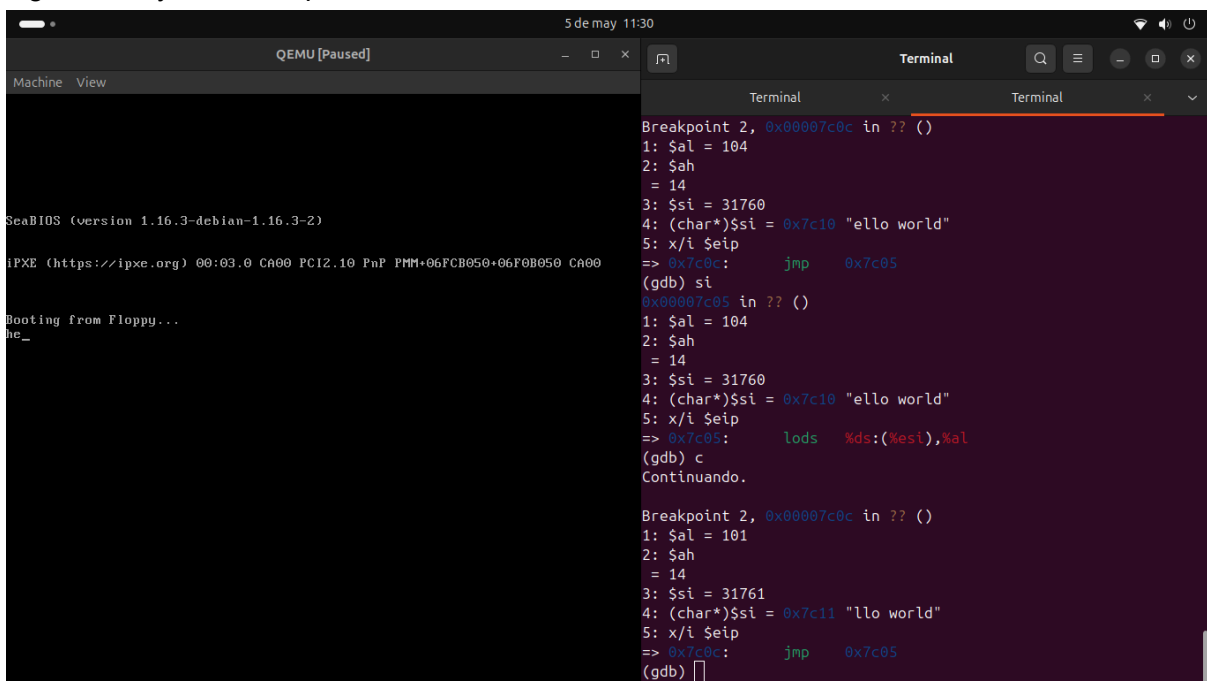
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCB050+06F0B050 CA00
Booting from Floppy...

Terminal
0x0000005b in ?? ()
1: $al = 0
2: $ah
= 0
3: $si = 0
4: (char*)$si = 0x0
5: x/i $eip
=> 0xe05b: add %al,(%eax)
(gdb) c
Continuando.

Breakpoint 1, 0x00007c00 in ?? ()
1: $al = 85
2: $ah
= -86
3: $si = 0
4: (char*)$si = 0x0
5: x/i $eip
=> 0x7c00: mov $0xeb47c0f,%esi
(gdb) si
0x00007c03 in ?? ()
1: $al = 85
2: $ah
= -86
3: $si = 31759
4: (char*)$si = 0x7c0f "hello world"
5: x/i $eip
=> 0x7c03: mov $0xe,%ah
(gdb)

```

Y a medida que se avanza en el loop vemos cómo se van extrayendo los caracteres del registro “si” y se van imprimiendo en QEMU.



```

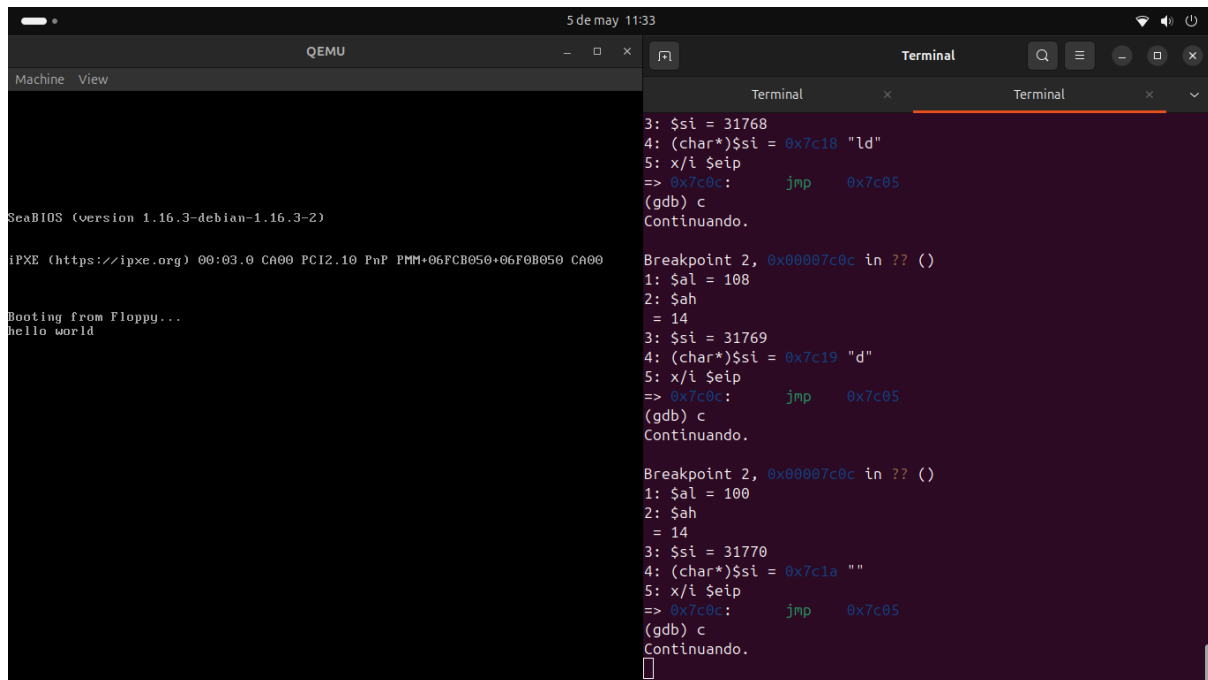
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCB050+06F0B050 CA00
Booting from Floppy...
he_

Terminal
Breakpoint 2, 0x00007c0c in ?? ()
1: $al = 104
2: $ah
= 14
3: $si = 31760
4: (char*)$si = 0x7c10 "ello world"
5: x/i $eip
=> 0x7c0c: jmp 0x7c05
(gdb) si
0x00007c05 in ?? ()
1: $al = 104
2: $ah
= 14
3: $si = 31760
4: (char*)$si = 0x7c10 "ello world"
5: x/i $eip
=> 0x7c05: lods %ds:(%esi),%al
(gdb) c
Continuando.

Breakpoint 2, 0x00007c0c in ?? ()
1: $al = 101
2: $ah
= 14
3: $si = 31761
4: (char*)$si = 0x7c11 "llo world"
5: x/i $eip
=> 0x7c0c: jmp 0x7c05
(gdb)

```

En esta captura se ve que el programa ya ha sido finalizado, vemos en el emulador impresa toda la cadena de caracteres.



The screenshot shows the QEMU emulator interface. On the left, the SeaBIOS boot process is visible, including the IPXE logo and the text "Booting from Floppy... hello world". On the right, a GDB terminal window is open, showing the execution of a program. The terminal output includes the following commands and results:

```
3: $si = 31768
4: (char*)$si = 0x7c18 "ld"
5: x/i $eip
=> 0x7c0c: jmp 0x7c05
(gdb) c
Continuando.

Breakpoint 2, 0x00007c0c in ?? ()
1: $al = 108
2: $ah
= 14
3: $si = 31769
4: (char*)$si = 0x7c19 "d"
5: x/i $eip
=> 0x7c0c: jmp 0x7c05
(gdb) c
Continuando.

Breakpoint 2, 0x00007c0c in ?? ()
1: $al = 100
2: $ah
= 14
3: $si = 31770
4: (char*)$si = 0x7c1a ""
5: x/i $eip
=> 0x7c0c: jmp 0x7c05
(gdb) c
Continuando.
```

En el debug se pudo seguir el desarrollo del programa paso a paso y verificar que las salidas tanto de objdump como de hd coincidan con la ubicación del programa dentro de la imagen en el emulador, tal como se esperaba.

Desafío final: Modo protegido

Creamos un código que permite pasar de modo real a modo protegido. La secuencia del programa es:

1. Arranca en modo real (.code16).
2. Desactiva interrupciones (cli).
3. Carga la GDT (lgdt).
4. Activa el modo protegido.
5. Salta a la sección de código de modo protegido (ljmp).
6. En modo protegido (.code32): configura los registros de segmento y ejecuta operaciones.

```
#include "common.h" /*Lo usamos solo para imprimir en pantalla*/

BEGIN
    .code16
    cli
```

```

/* ENTRAMOS AL MODO PROTEGIDO*/

/* Para entrar al modo protegido, es necesario configurar la tabla
global de descriptores (GDT)*/
.equ CODE_SEG, 8
.equ DATA_SEG, gdt_data - gdt_start

/* Le decimos al procesador donde esta la GDT en la memoria */
lgdt gdt_descriptor

/* Set PE (Protection Enable) bit in CR0 (Control Register 0),*/
mov %cr0, %eax
orl $0x1, %eax
    mov %eax, %cr0

/* Saltamos a la sección de código de 32 bits*/
ljmp $CODE_SEG, $protected_mode

/*ACA YA ESTAMOS EN MODO PROTEGIDO, EL RESTO ES CONFIGURACIÓN DE LOS
SEGMENTOS DE LA MEMORIA*/

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
vga_current_line:
    .long 0

```

```
.code32
protected_mode:
    /* Configuramos los otros segmentos*/
    mov $DATA_SEG, %ax
    mov %ax, %ds
    mov %ax, %es
    mov %ax, %fs
    mov %ax, %gs
    mov %ax, %ss

    mov $0X7000, %ebp
    mov %ebp, %esp

    push %eax
    push %ebx
    push %ecx
    push %edx

    mov $message, %ecx
    mov vga_current_line, %eax
    mov $0, %edx

    mov $25, %ebx
    div %ebx
    mov %edx, %eax

    mov $160, %edx
    mul %edx
    /* 0xb8000 == VGA BUFFER */
    lea 0xb8000(%eax), %edx
    mov $0x0f, %ah

loop:
    mov (%ecx), %al
    cmp $0, %al
    je end
    mov %ax, (%edx)
    add $1, %ecx
    add $2, %edx
    jmp loop
end:
    incl vga_current_line
```



```

    pop %edx
    pop %ecx
    pop %ebx
    pop %eax

    jmp .

    /* Fin VGA_PRINT_STRING */
message:
    .asciz "hello world, de parte de NullPointerException"

```

Para que un programa tenga dos descriptores de memoria diferentes, deberíamos colocar el segmento de datos y de código en lugares diferentes de la memoria. Por ejemplo

```

gdt_null:
    .long 0x0
    .long 0x0
gdt_code: /*Segmento de código: base=0x00000000, límite=0x000FFFFF*/
    .word 0xffff
    .word 0x0
    .byte 0x0
    .byte 0b10011010
    .byte 0b11001111
    .byte 0x0
gdt_data: /*Segmento de datos: base=0x00100000, límite=0x000FFFFF*/
    .word 0xffff
    .word 0x0
    .byte 0x10
    .byte 0b10010010
    .byte 0b11001111
    .byte 0x0

```

Vamos a cambiar los bits de acceso para que los datos sean de solo lectura, es decir

```

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 0b10010000
    .byte 0b11001111
    .byte 0x0
```

```
Output/messages
0x00007c3e in ?? ()
Assembly
0x00007c3e ? mov    %eax,%ss
0x00007c40 ? movl   $0x22222222,0x100000
0x00007c4a ? jmp    0x7c4a
0x00007c4c ? xchg   %ax,%ax
0x00007c4e ? xchg   %ax,%ax
0x00007c50 ? xchg   %ax,%ax
0x00007c52 ? xchg   %ax,%ax
0x00007c54 ? xchg   %ax,%ax
0x00007c56 ? xchg   %ax,%ax
0x00007c58 ? xchg   %ax,%ax
Breakpoints
[1] break at 0x00007c00 for * 0x7c00 hit 2 times
Expressions
History
Memory
Registers
    eax 0x00000010
    ecx 0x00000000
```

El procesador detecta un general protection fault (GDP) debido a una violación de acceso. A partir de esto, busca en la IDT (Interrupt Descriptor Table) el handler para el GP. Como no hay ningún handler cargado, el programa se reinicia.

En modo protegido, los registros de segmento (CS, DS, SS, ES, FS, GS) no contienen direcciones físicas ni lógicas, sino selectores de segmento que se usan para indexar la GDT o LDT.

Esto es porque en modo protegido, los accesos a memoria son controlados por:

- Selectores de segmento en los registros de segmento.
- Las entradas de la GDT o LDT, que contienen:
 - Base del segmento
 - Límite (tamaño)
 - Bits de acceso (RW, privilegio, etc.)

El selector permite al procesador localizar la entrada en la GDT, y con ella, verificar acceso, calcular la dirección efectiva, y aplicar protección por nivel de privilegio.