Modo real vs modo protegido

Agenda de hoy

Entorno posible de ejecución de programas x86

Bios y UEFI

Segmentación

Modo protegido

- GDT descriptor
- GDT

Entorno posible de ejecución de programas x86

Ejecutar programas en el hardware

Ejecutaremos programas directamente sobre el HW

- no usen su pc de trabajo
- usar pc vieja
- Virtualizar

Correr en HW (pref. viejo)

Quemar un pendrive con la imagen a probar

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

colocar el pen en la pc

encenderla e indicar que inicie desde la misma.

Como crear una imagen booteable

En la arquitectura x86 lo más simple es crear un sector de arranque MBR y colocarlo en un disco. Se puede crear un sector de arranque con una sola línea de printf

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

Structure of a classical generic MBR

Address		Description		Size (bytes)
Hex	Dec			
$+000_{hex}$	+0	Bootstrap code area		446
$+1BE_{hex}$	+446	Partition entry №1	*Partition table* (for primary partitions)	16
$+1CE_{hex}$	+462	Partition entry №2		16
$+1DE_{hex}$	+478	Partition entry №3		16
$+1EE_{hex}$	+494	Partition entry №4		16
$+1FE_{hex}$	+510	55_{hex}	*Boot signature*[a]	2
$+1FF_{hex}$	+511	AA_{hex}		
Total size: 446 + 4×16 + 2				**512**

main.img

main.img contiene:

\364 in octal == 0xf4 in hex: hlt instruction

cómo obtener la codificación de una instrucción en particular

echo hlt > a.S

as -o a.o a.S

objdump -S a.o

%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 una mbr

visualizar con

hd main.img

Correr la imagen

instalar y correr qemu con la imagen en cuestión.

sudo apt-get install qemu-system-x86

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

BIOS/UEFI

BIOS

Solo se puede acceder en modo real

Es vieja, pero es uno de los firmware mejor conocidos

UEFI es el nuevo estándar

Las funciones solo se acceden mediante interrupciones y los argumentos se pasan por registros

https://en.wikipedia.org/wiki/BIOS_interrupt_call#Interrupt_table

UEFI y coreboot

¿Que es UEFI ? ¿como puedo usarlo ?

¿Menciona casos de bugs de UEFI que puedan ser explotados?

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

¿Que es coreboot ? ¿que productos lo incorporan ? ¿cuales son las ventajas de su utilización?

Pequeño hello world

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"
```

link.ld

```
SECTIONS
{
    /* The BIOS loads t
     * We must tell tha
     * calculate the ad
     */
    . = 0x7c00;
    .text :
    {
        __start = .;
        *(.text)
        /* Place the ma
        . = 0x1FE;
        SHORT(0xAA55)
    }
}
```

compilar y linkear

```
as -g -o main.o main.S
ld --oformat binary -o main.img -T link.ld main.o
```

LD SCRIPTS https://www.math.utah.edu/docs/info/ld_3.html#SEC3
https://www.glamenv-septzen.net/en/view/6

Depuración de ejecutables con llamadas a bios int

gdb dashboard

```
# gdb
GNU gdb (GDB) 7.6.1-ubuntu
[...]
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000fff0 in ?? ()
(gdb) set architecture i8086
[...]
(gdb) br *0x7c00
```

una vez lanzado qemu

depurar con gdb

colocar un breakpoint en la dirección de arranque,

luego colocar otro a continuación de la llamada a la interrupción

Utilizar continue antes de las interrupciones y si para ejecutar una sola instrucción.

https://stackoverflow.com/questions/24491516/how-to-step-over-interrupt-calls-when-debugging-a-bootloader-bios-with-gdb-and-q

Ejercicio

Crear un documento donde respondan a las siguientes preguntas

¿Que es un linker? ¿que hace ?

¿Que es la dirección que aparece en el script del linker?¿Porqué es necesaria ?

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

Grabar la imagen en un pendrive y probarla en una pc y subir una foto

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

Modo protegido

Modos de funcionamiento x86

- Real-address, "real mode"
- Protected
- System management
- IA-32e. Has two sub modes:
  Compatibility
  64-bit

Real mode, protected mode, virtual 8086 mode, and system management mode. These are sometimes referred to as legacy modes.

Modelos de memoria

Segmentación

Paginación

note:The x86-64 architecture does not use segmentation

Más información sobre registros, **CR0**

Control registers in x64 series [edit]

CR0 [edit]

The CR0 register is 32 bits long on the 386 and higher processors. On x86-64 processors in long mode, it (and the other control registers) is 64 bits long. CR0 has various control flags that modify the basic operation of the processor.

Bit	Name	Full Name	Description
0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode
1	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0
2	EM	Emulation	If set, no x87 floating-point unit present, if clear, x87 FPU present
3	TS	Task switched	Allows saving x87 task context upon a task switch only after x87 instruction used
4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
16	WP	Write protect	When set, the CPU can't write to read-only pages when privilege level is 0
18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
29	NW	Not-write through	Globally enables/disable write-through caching
30	CD	Cache disable	Globally enables/disable the memory cache
31	PG	Paging	If 1, enable paging and use the § CR3 register, else disable paging.

Segmentación en modo real

Ver Ejemplo
https://github.com/cirosantilli/x86-bare-metal-examples/tree/18772b1403133b2328d5ad44791445f9859de320#real-mode-segmentation

the Linux kernel uses the GS segment to store per-CPU data.

Más segmentación

En general los segmentos se alteran de manera indirecta mediante otro registro o con instrucciones propias.

CS se altera con ljmp

SS afecta instrucciones que usen el SP como PUSH and POP (16 * SS + SP)

Modo protegido

Bios ya no está disponible

Utilizar VGA para salidas

Es necesario crear una GDT para arrancar

Las instrucciones dejan de ser de 16bits para ser de 32bits (.code32)

Permite el uso de anillos o rings de seguridad.

Proceso:

Deshabilitar interrupciones

Habilitar la línea a20

Cargar la GDT

Fijar el bit más bajo del CR0 en 1

saltar a la sección de código de 32bits

Configurar el resto de los segmentos

GDT descriptor y GDT

The GDT Descriptor

Bits	Function	Description
0-15	Limit	Size of GDT in bytes
16-47	Address	GDT's memory address

1st Double word:

Bits	Function	Description
0-15	Limit 0:15	First 16 bits in the segment limiter
16-31	Base 0:15	First 16 bits in the base address

2nd Double word:

Bits	Function	Description
0-7	Base 16:23	Bits 16-23 in the base address
8-12	Type	Segment type and attributes
13-14	Privilege Level	0 = Highest privilege (OS), 3 = Lowest privilege (User applications)
15	Present flag	Set to 1 if segment is present
16-19	Limit 16:19	Bits 16-19 in the segment limiter
20-22	Attributes	Different attributes, depending on the segment type
23	Granularity	Used together with the limiter, to determine the size of the segment
24-31	Base 24:31	The last 24-31 bits in the base address

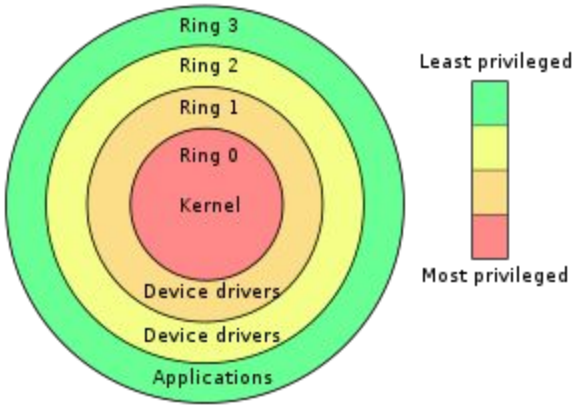
http://www.osdever.net/tutorials/view/the-world-of-protected-mode

Ver ejemplo impresion en memoria de video

https://github.com/cirosantilli/x86-bare-metal-examples/blob/master/common.h#L135

```
#include "common.h"
BEGIN
    CLEAR
    PROTECTED_MODE
    VGA_PRINT_STRING $message
    jmp .
message:
    .asciz "hello world"
```

Seguridad Anillos



Ejemplo para verificar junto con la sección de módulos de kernel

https://github.com/cirosantilli/linux-kernel-module-cheat/tree/ed5fa984c6226f81cb1a07f980d319ee9ee88e00#ring0

Actividad final

Crear un código assembler que pueda pasar a modo protegido y tenga dos descriptores de memoria diferentes para cada segmento (código y datos) en espacios de memoria diferenciados.

Cambiar los bits de acceso del segmento de datos para que sea de solo lectura, intentar escribir, que sucede? documentar los resultados que debería suceder a continuación?

ref: https://github.com/cirosantilli/linux-kernel-module-cheat

ref: https://github.com/cirosantilli/x86-bare-metal-examples