

Bootloader, Compilación/Linker y Modo Protegido

Organización del Computador II

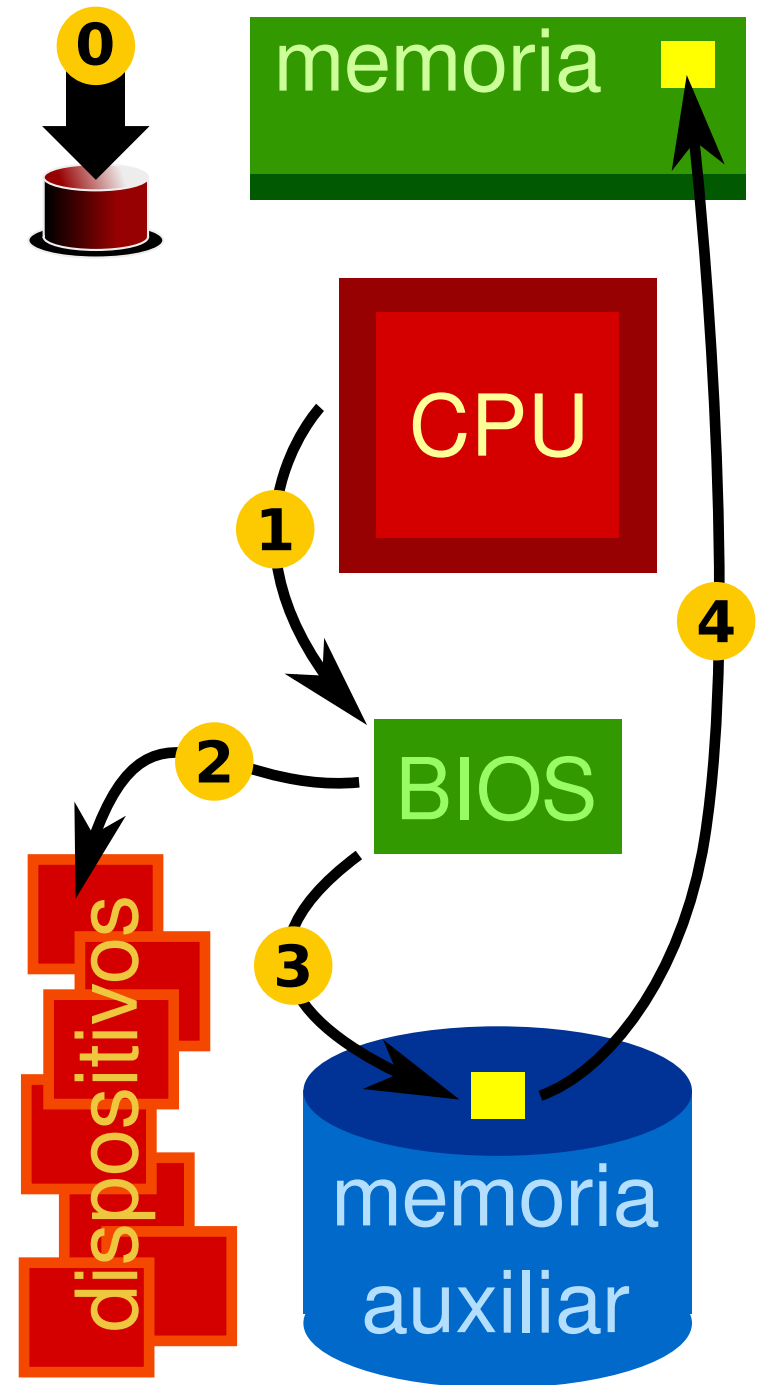
David Alejandro González Márquez

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

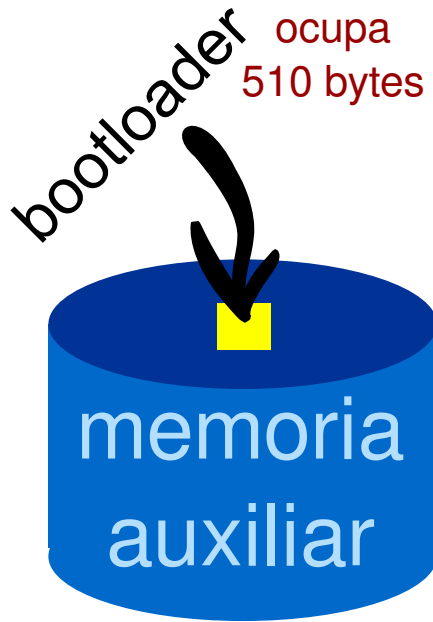
06-10-2016

■ Inicio

- 0** Presionamos el botón de encendido, la circuitería del mother da alimentación al microprocesador y arranca el sistema
- 1** El CPU comienza a ejecutar el BIOS (Basic Input Output System), que consiste de una memoria ROM en el mother con las primeras instrucciones para el CPU
- 2** El BIOS se encarga de correr una serie de diagnósticos llamados POST (Power On Self Test)
- 3** Busca un dispositivo "booteable" es decir, que en su sector de booteo los últimos dos bytes tengan la firma 0x55 y 0xAA respectivamente.
- 4** Se copia a memoria a partir de la dirección 0x7C00, el sector de booteo



■ Bootloader

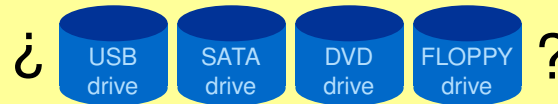


Es un programa sencillo diseñado exclusivamente para preparar todo lo que necesita el sistema operativo para funcionar. Normalmente se utilizan los cargadores de arranque multietapas, en los que varios programas pequeños se suman los unos a los otros, hasta que el ultimo de ellos carga el sistema operativo.

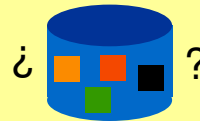
Pasos a seguir en el booteo

¡Todo esto en 510 bytes!

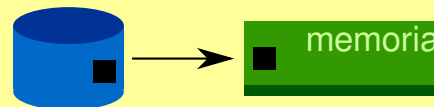
1- Determinar el 'disco' y la partición a bootear



2- Determinar donde esta la imagen del kernel en ese 'disco'



3- Cargar la imagen del kernel en memoria



4- Correr el "kernel"

4.1- Pasar a modo protegido

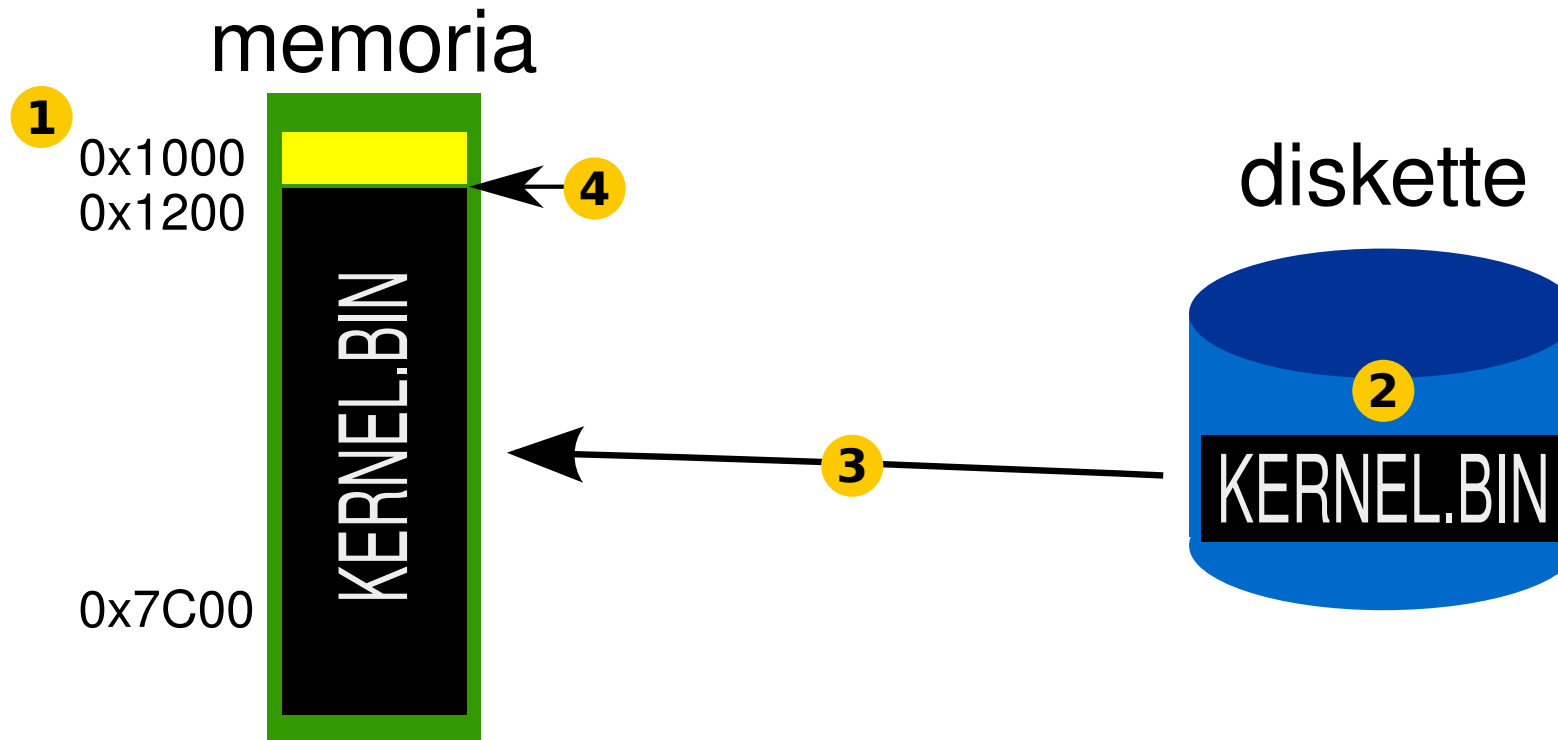
4.2- Preparar las estructuras para administrar la memoria

4.3- Preparar las estructuras del sistema

4.4- ¡Listo!

■ Como somos buenos... bootloader ORGA2

Vamos a dar un bootloader para que hagan sus ejercicios



El proceso que realiza es el siguiente:

- 1- Se copia el Bootloader en la posición **0x1000** de la memoria
- 2- Se busca el archivo **KERNEL.BIN** en el diskette
- 3- Se copia ese archivo en la posición **0x1200** de la memoria
- 4- Se salta y se ejecuta la instrucción en la posición **0x1200** de la memoria

Ustedes deben crear un archivo **KERNEL.BIN** y guardarlo en el diskette

■ Compilando y Enlazando

- Cuando un compilador construye un programa, lo hace de forma que pueda correr sobre un sistema operativo determinado
- Para resolver las direcciones de las etiquetas toma una dirección de inicio, por ejemplo la dirección 0x00000000
- Cada etiqueta se traduce a una dirección contando bytes desde la dirección de inicio
- Cuando se hacen saltos o llamadas a direcciones absolutas se debe conocer en tiempo de compilación la dirección del salto o de la llamada

¿Y si no estamos en un sistema operativo?

Por ejemplo, el archivo KERNEL.BIN se carga en la dirección 0x1200

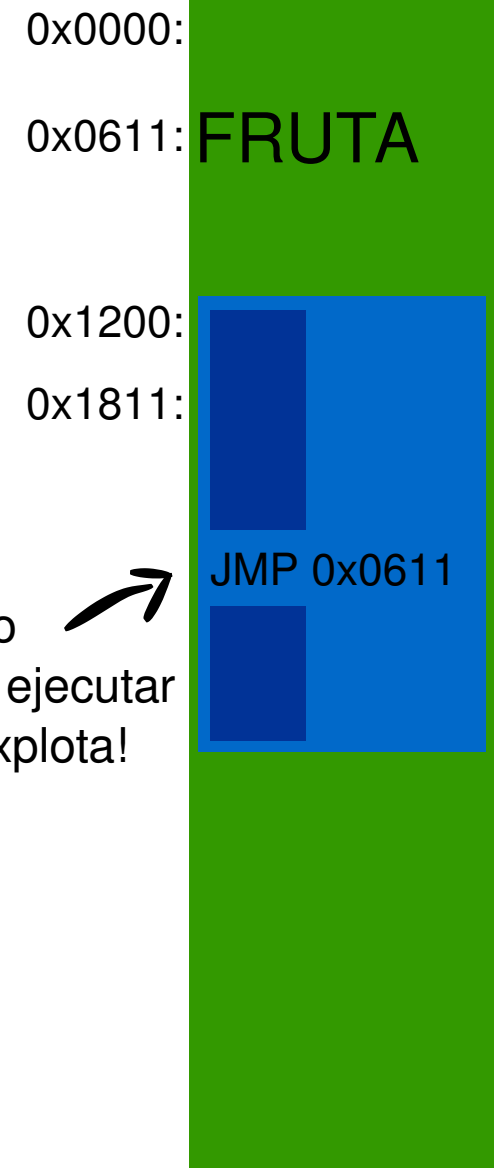
■ Compilando y Enlazando - ejemplo

Mi programa

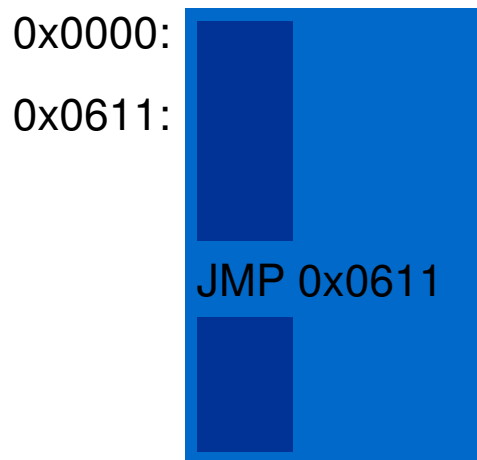


y si cargamos
el programa sin
dirección de origen

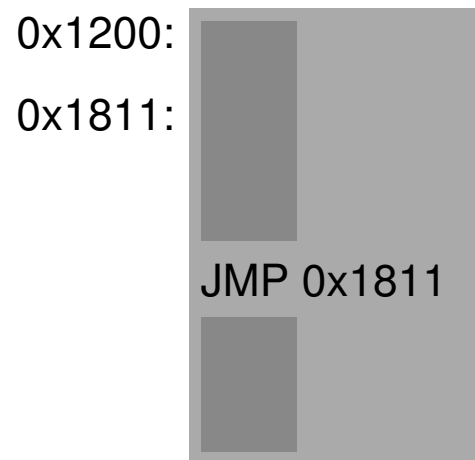
memoria



compilamos...



**SIN dirección de
origen**



**CON dirección de
origen 0x1200**

cuando
quiere ejecutar
esto explota!

■ Compilando y Enlazando

Indicar la dirección de origen

Hay 2 formas de hacerlo, según como se compile:

Si compilamos de **assembler a binario**, usamos la directiva ORG al inicio del archivo.asm para indicar la dirección de origen

Ej: ORG 0x1200

Si compilamos de **assembler a elf**, usamos el parámetro -Ttext en el linker

Ej: -Ttext 0x1200

Compilación de C en formato elf32 y linkeo

Compilación:

```
gcc -m32 -fno-zero-initialized-in-bss -fno-stack-protector  
-ffreestanding -c -o archivo.elf archivo.c
```

Linkeo:

```
ld -static -m elf_i386 -nostdlib -N -b elf32-i386 -e start -Ttext 0x1200  
-o archivo.elf archivo.o
```

Lo convertimos en binario:

```
objcopy -S -O binary archivo.elf archivo.bin
```

Compilación de un bootloader y creación de diskette

Creamos un diskette vacío:

```
dd bs=512 count=2880 if=/dev/zero of=diskette.img
```

Formateamos la imagen en FAT12:

```
sudo mkfs.msdos -F 12 diskette.img -n ETIQUETA
```

Escribimos en el sector de booteo:

```
dd if=bootloader.bin of=diskette.img count=1 seek=0 conv=notrunc
```

Compilación de assembly en formato bin

Ensamblado:

```
nasm -fbin archivo.asm -o archivo.bin
```

Consideraciones:

Todo el código ejecutable tiene que estar incluido
(No hay bibliotecas)

Se ejecuta tal cual se escribió, no hay entry point.

Compilación de assembly en formato elf32 y linkeo

Compilación:

```
nasm -felf32 archivo.asm -o archivo.o
```

Linkeo:

```
ld -static -m elf_i386 --oformat binary -b elf32-i386 -e start  
-Ttext 0x1200 archivo.o -o archivo.bin
```

Consideraciones: OJO código de 32 bits (en modo protegido)

Se pueden usar bibliotecas. No se respeta el entry point.

El parámetro **Ttext** da el origen de la sección .text.

Si usan el Bootloader de Orga 2, deben usar 0x1200 como origen de la sección .text.

Copiado del KERNEL.BIN dentro del diskette

```
mcop -i diskette.img kernel.bin ::/
```

■ Modo Real

Programación en 16bits

- ✓ No hay protección de memoria
- ✓ No se pueden restringir las instrucciones
- ✓ AX, CX y DX no son de propósito general, no se pueden usar para acceder a memoria
- ✓ Los compiladores modernos no generan código para modo real, no queda otra que el assembler
- ✓ Podemos usar la BIOS, sus rutinas de acceso a dispositivos (por ejemplo, para imprimir por pantalla)
- ✓ Tenemos Registros de Segmento (CS, DS, SS, ...)

OJO

Estamos solos contra el mundo

(no hay librerías no hay printf, ¡no hay **nada**!).

Tenemos un binario plano, chau
section .data section .text...

¡Ojo con ejecutar los datos!

¡Ojo con modificar el código en tiempo de ejecución!

No hay segmentation fault

Toda la memoria es “nuestra”



**Hasta que no pasemos a modo protegido,
tenemos el mejor 8086 de la historia**

Direccionamiento en 16bits

Modos de
direccionamiento

[val]	[BX + val]	[BX + SI + val]
	[SI + val]	[BX + DI + val]
	[DI + val]	[BP + SI + val]
	[BP + val]	[BP + DI + val]

Cada dirección de memoria esta definida
por un **segmento** y un **offset** (de 16 bits cada uno)

0x  : 0x 
segmento offset

La forma de calcular a que dirección física que corresponde es:

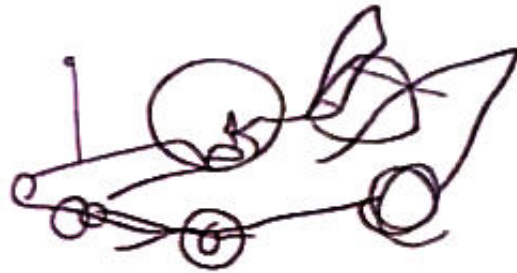
$(\text{segmento} \ll 4) + \text{offset}$

Por ejemplo:

$(0x07C0 \ll 4) + 0x0120 = 0x7C00 + 0x0120 = 0x7D20$
segmento offset

■ Modo Protegido vs Modo Real

Modo Real



Memoria
disponible

1Mb*

Privilegios

¿cuac?

Manejo de
Interrupciones

rutinas de
atención

Acceso a
instrucciones

todas

Modo Protegido



4Gb*

4 niveles de
protección

rutinas de atención
con privilegios

depende del nivel
de protección

Modo Protegido - Tablas

GDT - Global Descriptor Table

Es una **tabla** alocada en memoria donde cada entrada es de **8 bytes** y define alguno de los siguientes **descriptores**:

- Descriptor de segmento de memoria
- Descriptor de Task State Segment (TSS)
Guarda el estado de una tarea, sirve para intercambiar tareas
- Descriptor de LDT
- Descriptor de call gate
Permite transferir control entre niveles de privilegios
Actualmente no se usan en SO modernos

El primer descriptor de la tabla siempre es NULO

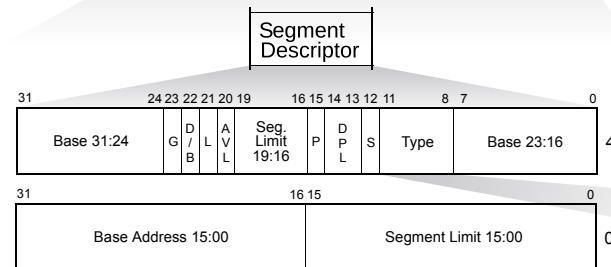
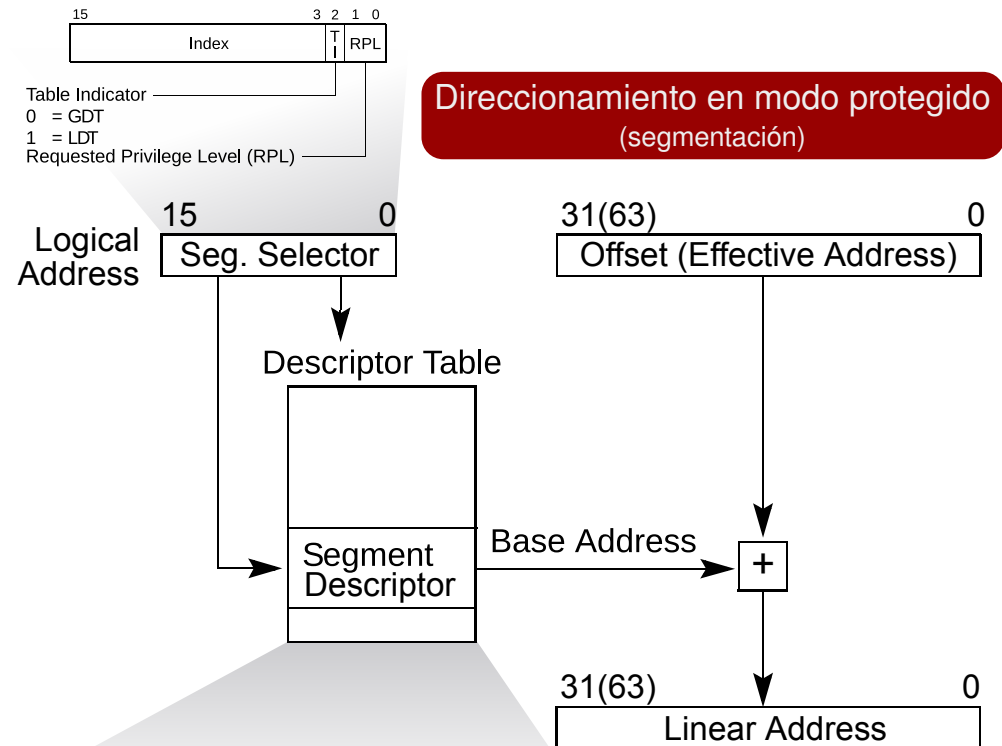
LDT - Local Descriptor Table

Es una tabla alocada en memoria que puede contener las mismas entradas que la GDT

Se diferencia en:

- La **GDT** tiene los descriptores **globales** y es única para todo el sistema
- La **LDT** tiene los descriptores **locales** a una tarea y puede existir más de una ldt en el sistema, una por cada tarea

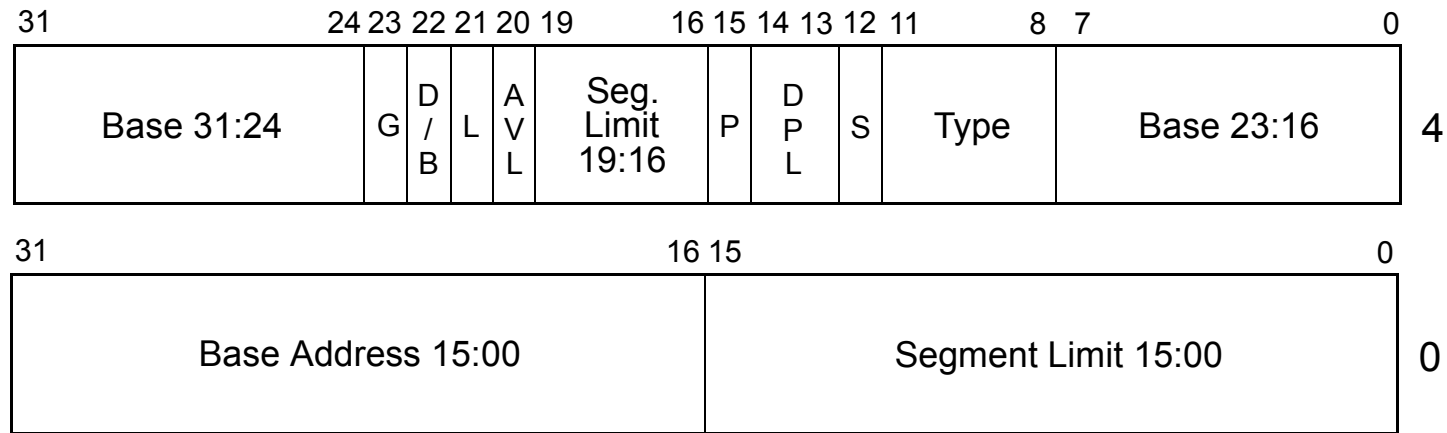
Esta tabla quedo obsoleta por el uso del mecanismo de paginación



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Type Field					Descriptor Type	Description
Decimal	11	10	9	8		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read, conforming
15	1	1	1	1	Code	Execute/Read, conforming, accessed

■ Modo Protegido - Tablas



- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type



Table Indicator —

0 = GDT

1 = LDT

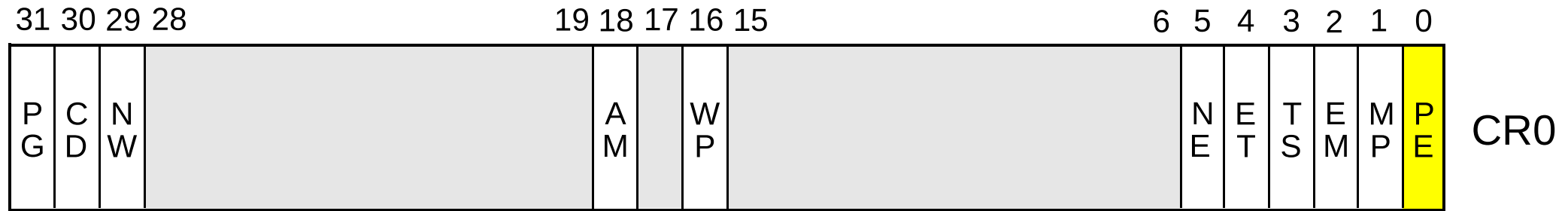
Requested Privilege Level (RPL) —

■ Modo Protegido - Tablas

Type

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read, conforming
15	1	1	1	1	Code	Execute/Read, conforming, accessed

■ Pasar a Modo Protegido



Activar **Modo Protegido** es setear en 1 el bit **PE** del registro de control **CR0**



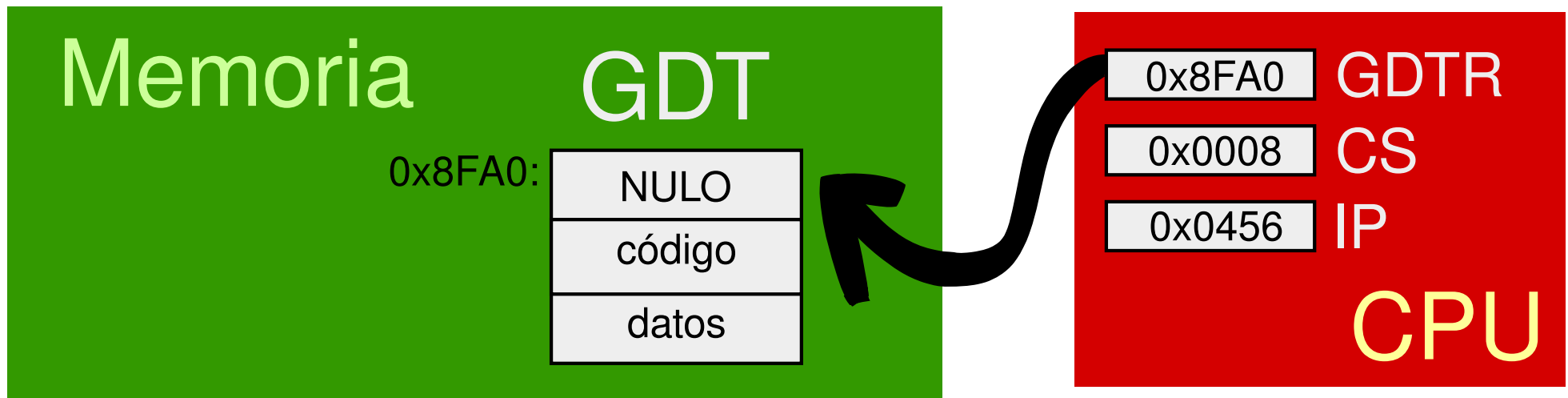
Protected Environment

cataplum.

1. onomat. U. para expresar ruido, explosión o golpe.

rae.org

■ Pasar a Modo Protegido - ¿por qué cataplum?



¿Cómo sabemos donde esta la GDT?

cargar el registro **GDTR** utilizando **LGDT**

¿Qué tiene la GDT?

por ejemplo, el descriptor nulo, luego un descriptor de código y uno de datos

¿Cuál es la próxima instrucción a ejecutar?

La instrucción en la dirección **CS:EIP**

¿Qué valor tiene que tener **CS** y cómo lo cambiamos?

..... ; esto se ejecuta en modo real

jmp 0x08:modoprotegido

modoprotegido:

.... ; esto se ejecuta en modo protegido

← ¡GRAN SALTO!

■ Pasar a Modo Protegido

- 0 - Completar la **GDT**
- 1 - Deshabilitar interrupciones (CLI)
- 2 - Cargar el registro **GDTR** con la dirección bas de la **GDT** (LGDT <offset>)
- 3 - Setear el bit **PE** del registro **CR0** (MOV eax,cr0 | OR eax,1 | MOV cr0,eax)
- 4 - **FAR JUMP** a la siguiente instrucción (JMP <selector>:<offset>)
- 5 - Cargar los registros de segmento (**DS**, ES, GS, FS y **SS**)

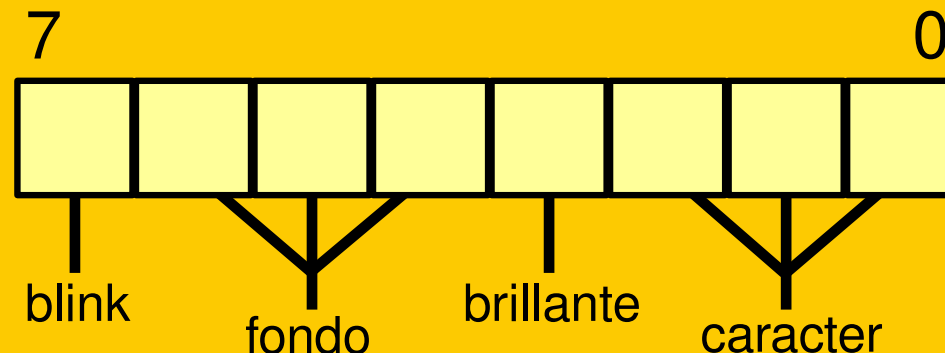
■ Cosas en el tintero

A20 Gate

- Es una línea de control para controlador de memoria, que habilita el acceso a direcciones superiores a los 2^{20} bits
- Para poder direccionar direcciones sobre el MB de memoria se debe habilitar este pin de nombre A20

Memoria de video

- La memoria de video comienza en el **segmento 0xB800** (dir. **0xB8000**)
- La matriz de video tiene **80 columnas** y **25 filas**
- Cada elemento es de **2 bytes** (modo y caracter ASCII)
- Cada bit del color del modo de video indica las componentes RGB (8 colores)



■ Ejercicio

Hacer un kernel que,

- Imprima por pantalla que esta en **modo real**
- Pase a modo protegido
- Imprima por pantalla que esta en **modo protegido**

■ Ejercicio - ejemplo de GDT

```
gdt:
; El primer descriptor de segmento es nulo.
dd 0x0
dd 0x0

; Segmento de código de nivel 0
sc0: dd 0x00003FFF
     dd 0x00C09A00

; Segmento de datos de nivel 0
sd0: dd 0x00003FFF
     dd 0x00C09200

; Segmento de datos de nivel 0 que empieza en b8000 (video)
video: dd 0x80000FBF
       dd 0x0040920B

gdt_desc: dw $-gdt ; tamaño de la tabla
          dd gdt    ; dirección de la tabla
; | 2 bytes tamaño de la tabla | 4 Bytes dirección de la tabla |
```

■ Ejercicio - ejemplo, cargar GDT y a modo protegido

ORG 0x1200

BITS 16

```
cli
lgdt [gdt_desc] ; cargar GDT
mov eax, cr0
or eax, 1
mov cr0, eax
jmp 0x8:mp      ; saltamos a modo protegido
```

BITS 32

```
mp:
    ; cargar los selectores de segmento
    xor eax, eax
    mov ax, 10000b    ; { index: 2 | gdt/ldt: 0 | rpl:00 }
    mov ds, ax        ; data segment
    mov es, ax
    mov gs, ax
    mov ax, 11000b    ; { index: 3 | gdt/ldt: 0 | rpl:00 }
    mov fs, ax        ; fs selector de segmento de video
```



¡Gracias!

¿Preguntas?