

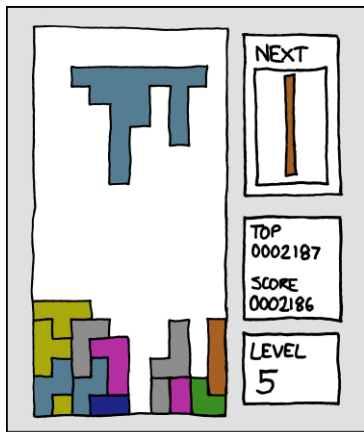
# Bochs, Bootloader y Modo Protegido

## Organización del Computador II

David Alejandro González Márquez

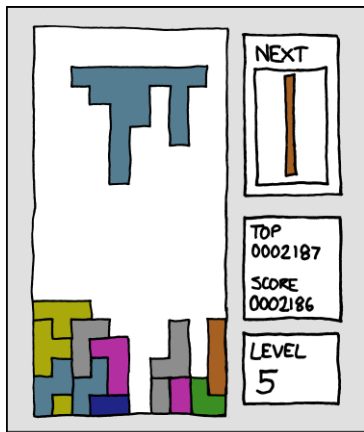
Departamento de Computación  
Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

8-10-19



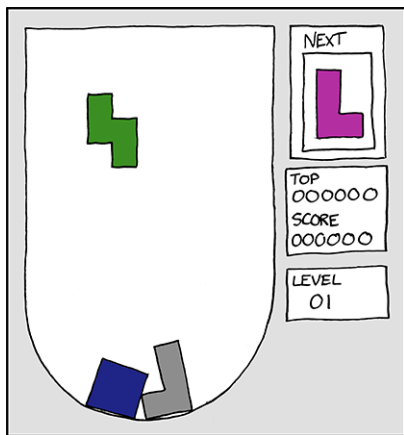
HEAVEN

Programación a nivel de Usuario



HEAVEN

Programación a nivel de Usuario



HELL

Programación de S. O.





Interrupciones

Tareas

Protección

Paginación

Segmentación

Scheduler

# Agenda

- Introducción a Bochs
- Bootloader, Compilado y Enlazado
- Pasaje a Modo Protegido
- Presentación del TP3

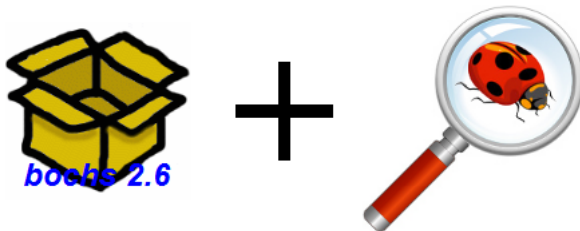
# ¿Porque usar bochs?

El procesador posee instrucciones que no se pueden usar a nivel de usuario.

Por lo tanto, si queremos acceder a estos mecanismos debemos estar en lugar del sistema operativo.

- Utilizar instrucciones de nivel privilegiado
- Acceder a los mecanismos de manejo de memoria
- Cambiar modos del procesador
- Controlar interrupciones

Un debugger como gdb es un **proceso** más. No puede monitorear el sistema operativo.



Bochs + Debugger

Necesitamos un debugger **en** la Virtual Machine.



Bochs es un simulador de una computadora por esto nos permite correr instrucción por instrucción.

Pero su versión oficial no esta compilada con esta posibilidad.

- bajar de:  
`https://sourceforge.net/projects/bochs/files/bochs/2.6.9/`  
el archivo `bochs-2.6.9.tar.gz`
- descomprimir: `tar -xvzf bochs-2.6.9.tar.gz`
- en la carpeta descomprimida hacer:
  - `./configure --enable-debugger --enable-disasm`  
`--disable-docbook --enable-readline`  
`LDFLAGS='-pthread'`  
`--prefix=/home/< usuario >/bochs/`
  - `make`
  - `make install`

## Bonus para los que usan su notebook

- Les puede faltar el paquete: `libgtk2.0-dev`
- Además puede instalar el paquete `libreadline`, guardar el buffer de la consola. (aka “flechitas”)

```
sudo apt-get install libgtk2.0-dev libreadline-dev
```

# Para gastar menos los dedos

- Saltar el primer breakpoint

Crear un archivo de nombre bochsdbg con el contenido:

```
continue
```

- Cargar el bochs:

```
bochs -q -rc bochsdbg
```

- Para usar bochs desde cualquier path, debemos incluirlo en la lista de path de nuestro usuario.

Agregar en el archivo /home/< *usuario* >/.bashrc

```
export PATH+=":/home/< usuario >/bochs/bin/"
```

- Cargar cambios en la consola actual:

```
source ~/.bashrc
```

```
-----  
Bochs Configuration: Main Menu  
-----
```

This is the Bochs Configuration Interface, where you can describe the machine that you want to simulate. Bochs has already searched for a configuration file (typically called bochsrc.txt) and loaded it if it could be found. When you are satisfied with the configuration, go ahead and start the simulation.

You can also start bochs with the `-q` option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [6]

# Bochs: Config file

- Una imagen de linux de ejemplo:  
<http://bochs.sourceforge.net/diskimages.html>

bochsrc

```
megs: 32
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
vga: extension=vbe
floppya: 1_44=a.img, status=inserted
floppyb: 1_44=b.img, status=inserted
ata0-master: type=disk, path=boot.img, cylinders=900, heads=15, spt=17
boot: c
log: bochsout.txt
mouse: enabled=0
clock: sync=slowdown
vga_update_interval: 150000
display_library: x, options="gui_debug" # use GTK debugger gui
# This enables the "magic breakpoint" feature when using the debugger.
# The instruction XCHG BX, BX causes Bochs to enter the debugger mode.
magic_break: enabled=1
```

- Activar log del en bochs

Descomentar la siguiente línea del bochsrc:

- `#log: bochs.log`

Comentar la siguiente:

- `log: /dev/null`

- Generar logs de todos los eventos, reemplazar lo siguiente:

- `debug: action=ignore` → `debug: action=report`

El tamaño del archivo puede ser muy grande, log de **TODO**

En computadoras de los laboratorios usar el Libre o /tmp

## Opciones de debugging

- s | step | stepi [count] - ejecuta [count] instrucciones
- n | next | p - ejecuta instrucciones sin entrar a las subrutinas
- c | cont | continue - continua la ejecución
- q | quit | exit - sale del debugger y del emulador
- Ctrl-C Detiene la ejecución y retorna al prompt

- `r | reg | regs | registers` - Lista los registros del CPU y sus contenidos

```
<bochs:12> registers
```

```
eax: 0x00000000 0
```

```
ecx: 0x00000000 0
```

```
edx: 0x00000543 1347
```

```
ebx: 0x00000000 0
```

```
esp: 0x00000000 0
```

```
ebp: 0x00000000 0
```

```
esi: 0x00000000 0
```

```
edi: 0x00000000 0
```

```
eip: 0x0000e05d
```

```
eflags 0x00000046
```

```
id vip vif ac vm rf nt IOPL=0 of df if tf sf ZF af PF cf
```



# Memory Dump

- x /nuf [addr] - Muestra el contenido de la dirección [addr]
- xp /nuf [addr] - Muestra el contenido de la dirección física [addr] nuf es número que indica cuantos valores se mostrarán, seguido de uno o más de los indicadores de formato.
  - x : hex
  - d : decimal
  - u : sin signo
  - o : octal
  - t : binario
  - c : char
  - s : ascii
  - i : instrucción

select the size:

- b : byte
- h : word = half-word
- w : doubleword = word

- `u | disasm | disassemble [count] [start] [end]` - desensambla instrucciones desde la dirección lineal `[start]` hasta `[end]` exclusive.
- `u | disasm | disassemble switch-mode` - Selecciona la sintaxis Intel o AT&T de assembler
- `u | disasm | disassemble size = n` - Setea el tamaño del segmento a desensamblar

# Breakpoints

- `p | pb | break | pbreak [addr]` - Crea un breakpoint en la dirección física `[addr]`
- `vb | vbreak [seg:offset]` - Crea un breakpoint en la dirección virtual `[addr]`
- `lb | lbreak [addr]` - Crea un breakpoint en la dirección lineal `[addr]`
- `d | del | delete [n]` - Borra el breakpoint número `[n]`
- `bpe [n]` - Activa el breakpoint número `[n]`
- `bpd [n]` - Desactiva el breakpoint número `[n]`

- watch - Muestra el estado actual de los watches
- watch stop - Detiene la simulación cuando un watch es encontrado
- watch continue - No detiene la simulación si un watch es encontrado
- watch r | read [addr] - Agrega un watch de lectura en la dirección física [addr]
- watch w | write [addr] - Agrega un watch de escritura en la dirección física [addr]

- info break - Muestra los Breakpoint creados
- info eflags - Muestra el registro EEFLAGS
- info idt - Muestra el descriptor de interrupciones (idt)
- info ivt - Muestra la tabla de vectores de interrupción
- info gdt - Muestra la tabla global de descriptores (gdt)
- info tss - Muestra el segmento de estado de tarea actual (tss)
- info tab - Muestra la tabla de paginas

- sreg - Muestra los registros de segmento

```
<bochs:5> sreg
cs:s=0xf000, dh=0xff0093ff, dl=0x0000ffff, valid=7
ds:s=0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
ss:s=0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
es:s=0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
fs:s=0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
gs:s=0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
ldtr:s=0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:s=0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x00000000, limit=0xffff
idtr:base=0x00000000, limit=0xffff
```

- creg - Muestra los registros de control

```
<bochs:10> creg
```

```
CR0=0x60000010: pg CD NW ac wp ne ET ts em mp pe
```

```
CR2=page fault laddr=0x00000000
```

```
CR3=0x00000000
```

```
    PCD=page-level cache disable=0
```

```
    PWT=page-level writes transparent=0
```

```
CR4=0x00000000: osxsave smx vmx osxmmexcpt osfxsr pce pge mce pae pse de tsd pv
```

# Magic Breakpoint

- `xchg bx, bx` - Magic breakpoint

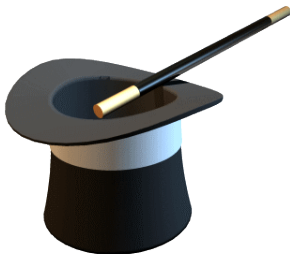
Esta instrucción detiene el flujo del programa y nos devuelve al prompt de bochs.



# Magic Breakpoint

- `xchg bx, bx` - Magic breakpoint

Esta instrucción detiene el flujo del programa y nos devuelve al prompt de bochs.



- Manual:  
`http://bochs.sourceforge.net/doc/docbook/user/`
- Información complementaria:  
`http://wiki.osdev.org/Bochs`
- Imágenes de Disco:  
`http://bochs.sourceforge.net/diskimages.html`

# Inicio - Pasos de Boot

0

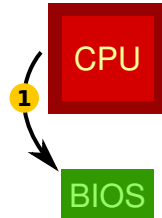
Presionamos el botón de encendido, la circuitería del mother da alimentación al microprocesador y arranca el sistema



# Inicio - Pasos de Boot

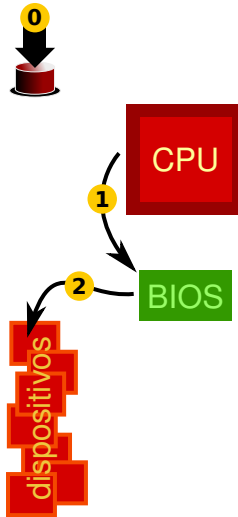
**0** Presionamos el botón de encendido, la circuitería del mother da alimentación al microprocesador y arranca el sistema

**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



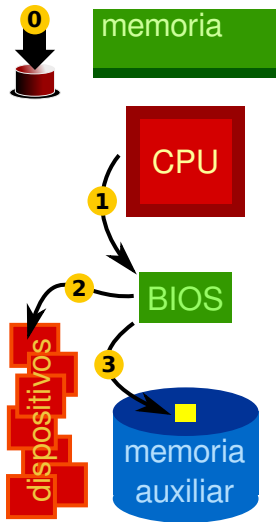
# Inicio - Pasos de Boot

- 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)



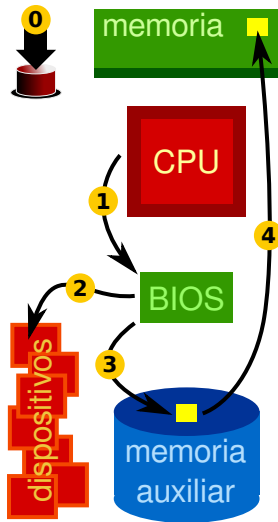
# Inicio - Pasos de Boot

- 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 "bootable" es decir, que en su sector de booteo los últimos dos bytes tengan la firma 0x55 y 0xAA respectivamente.

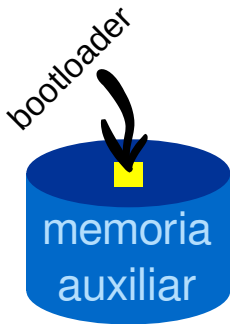


# Inicio - Pasos de Boot

- 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 "bootable" 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

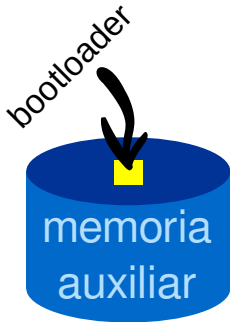


## Pasos del Bootloader





## Pasos del Bootloader

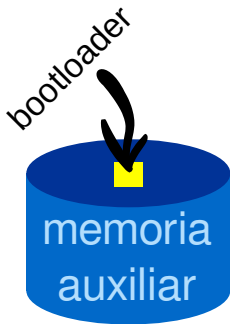


### 1- Determinar el 'disco' y la partición a bootear



# Bootloader

## Pasos del Bootloader



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

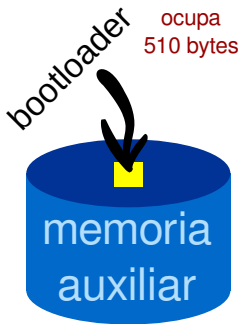


2- Determinar donde esta la imagen del kernel en ese 'disco'



# Bootloader

## Pasos del Bootloader



¡Todo esto en 510 bytes!

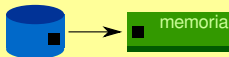
1- Determinar el 'disco' y la partición a bootear



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

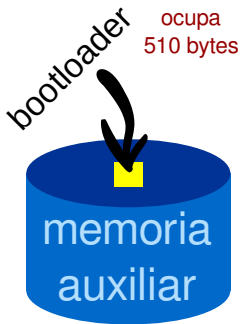


3- Cargar la imagen del kernel en memoria



# Bootloader

## Pasos del Bootloader



¡Todo esto en 510 bytes!

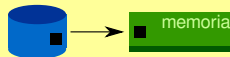
1- Determinar el 'disco' y la partición a bootear



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

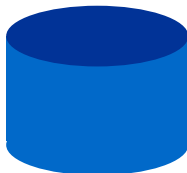
4.3- Preparar las estructuras del sistema

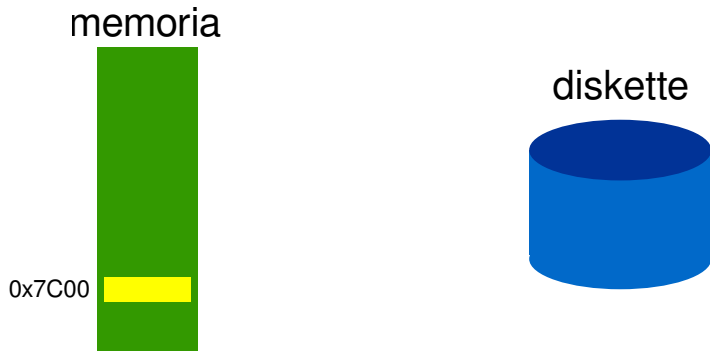
4.4- ¡Listo!

memoria



diskette







- 1 - Se copia el Bootloader en la posición **0x1000** de la memoria

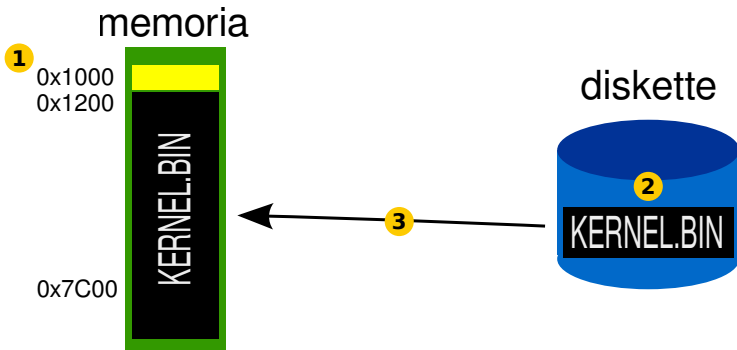
# Bootloader de Orga2



- 1 - Se copia el Bootloader en la posición **0x1000** de la memoria
- 2 - Se busca el archivo **KERNEL.BIN** en el diskette

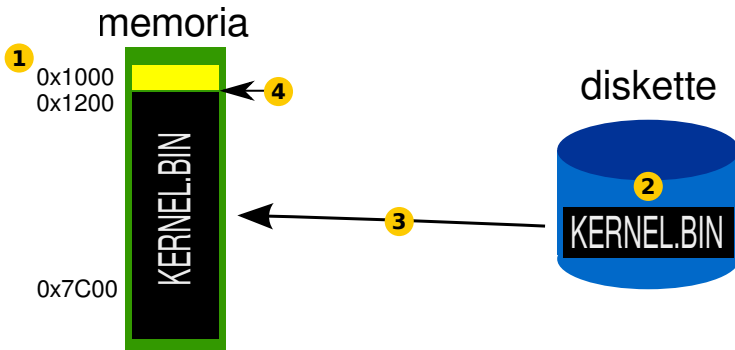


# Bootloader de Orga2

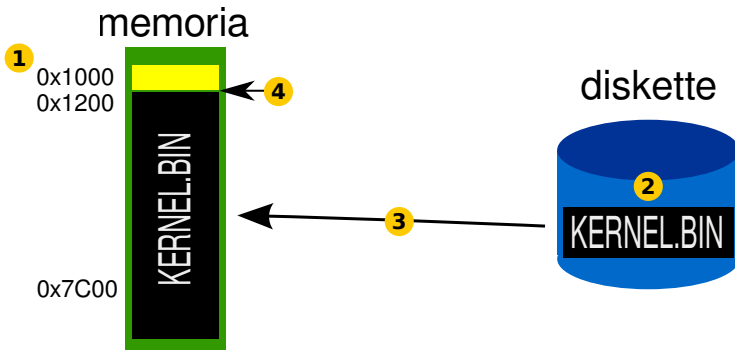


- 1- Se copia el Bootloader en la posición **0x1000** de la memoria
- 2- Se busca el archivo **KERNEL.BIN** en el diskette
- 3- Se copia ese archivo en la posición **0x1200** de la memoria

# Bootloader de Orga2



- 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



- 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

- Un compilador construye un programa de forma que pueda correr sobre un **sistema operativo** determinado
- Para resolver direcciones, toma una **dirección de inicio**, por ejemplo 0x00000000
- Cada etiqueta se traduce a una dirección contando **bytes** desde la dirección de inicio
- Los **destinos** de saltos o llamadas a funciones se debe conocer en tiempo de enlazado
- ¿Y si no estamos en un sistema operativo?  
Ej: el archivo `KERNEL.BIN` se carga en la dirección 0x1200

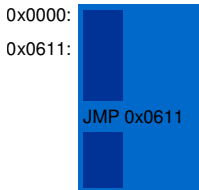
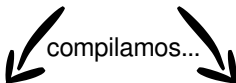
## Mi programa

CICLO:



# Compilando y Enlazando

Mi programa



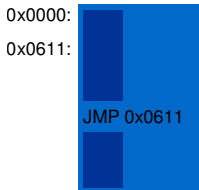
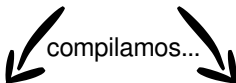
**SIN dirección de origen**



**CON dirección de origen 0x1200**

# Compilando y Enlazando

Mi programa



**SIN dirección de origen**



**CON dirección de origen 0x1200**

memoria

0x0000:

# Compilando y Enlazando

Mi programa



y si cargamos  
el programa sin  
dirección de origen

memoria

0x0000:

0x1200:

0x1811:



compilamos...

0x0000:

0x0611:



**SIN** dirección de  
origen

0x1200:

0x1811:

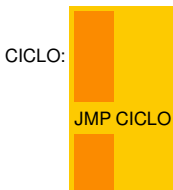


**CON** dirección de  
origen 0x1200

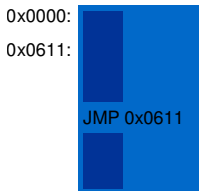
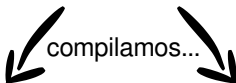


# Compilando y Enlazando

## Mi programa



y si cargamos  
el programa sin  
dirección de origen



**SIN** dirección de  
origen



**CON** dirección de  
origen 0x1200

## memoria

0x0000:

0x0611:

FRUTA

0x1200:

0x1811:

JMP 0x0611

cuando  
quiere ejecutar  
esto explota!

## Indicar la dirección de origen

Hay 2 formas de hacerlo, según como se compile:

- De **ensamblador a binario**, usamos la directiva `ORG` al inicio del archivo `.asm` para indicar la dirección de origen

```
ORG 0x1200
```

- De **ensamblador a elf**, usamos el parámetro `-Ttext` en el linker

```
-Ttext 0x1200
```

## Compilación de ensamblador a binario

- 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 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 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.

## Compilación de un bootloader y creacion de diskette

- Creamos un diskette vacio:  
`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`
- Copiado del KERNEL.BIN dentro del diskette  
`mcoppy -i diskette.img kernel.bin ::/`

## 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, ...)

## Programación en 16bits

Estamos solos contra el mundo...

No hay librerías no hay printf, ¡no hay nada!

Tenemos un binario plano, y *chau*

¡Ojo con ejecutar los datos!

`section .data section .text ... ja ja ja`

¡Ojo con modificar el código en tiempo de ejecución!

No hay segmentation fault

Toda la memoria es casi nuestra

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

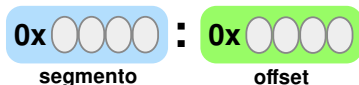


# Direccionamiento en 16bits

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)



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



(**segmento** << 4) + **offset**

*Por ejemplo:*

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

segmento                      offset

# Modo Real vs Modo Protegido

	Modo Real	Modo Protegido
		
Memoria disponible	1Mb*	4Gb*
Privilegios	¿cuac?	4 niveles de protección
Manejo de Interrupciones	rutinas de atención	rutinas de atención con privilegios
Acceso a instrucciones	todas	depende del nivel de protección

**Tabla** en memoria donde **cada entrada es de 8 bytes**.

Define alguno de los siguientes descriptores:

- **Descriptor de segmento de memoria** (S=1)
- Descriptor de Task State Segment (TSS) (S=0)  
Guarda el estado de una tarea, sirve para intercambiar tareas
- Descriptor de call gate (S=0)  
Permite transferir control entre niveles de privilegios  
Actualmente no se usan en SO modernos
- Descriptor de LDT (S=0)

El primer descriptor de la tabla siempre es NULO

Tabla en memoria, igual que la GDT.

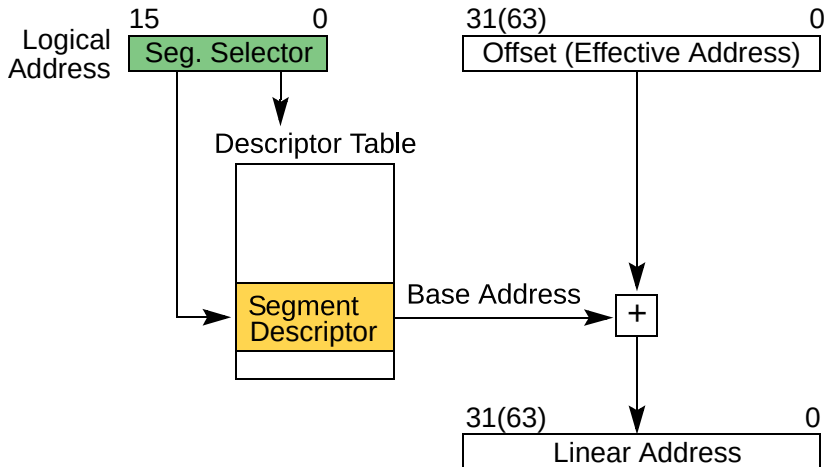
Puede conterner las **mismas entradas que la GDT**

Se diferencia en:

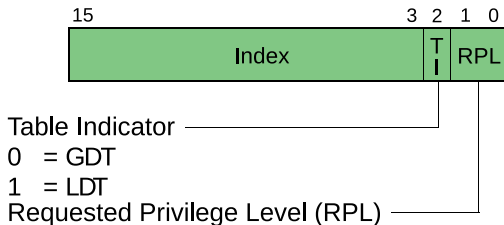
- La GDT tiene los descriptores globales y es única para todo el sistema.
- La LDT tiene los descriptores locales a una tarea y puede existir más de una LDT en el sistema, una por cada tarea.

**Tabla obsoleta** por el uso del mecanismo de paginación

# Unidad de Segmentación



# Selector de Segmento



CS: Para acceder a código

SS: Para acceder a pila

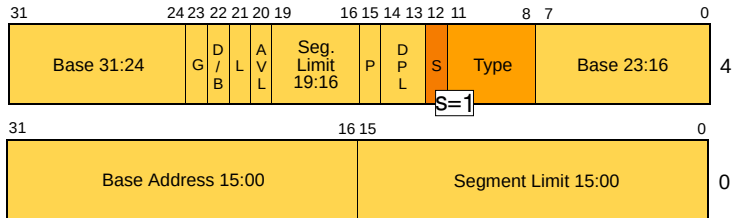
DS: Para acceder a datos (default)

ES: Para acceder a datos

GS: Para acceder a datos

FS: Para acceder a datos

# Descriptor de Segmento

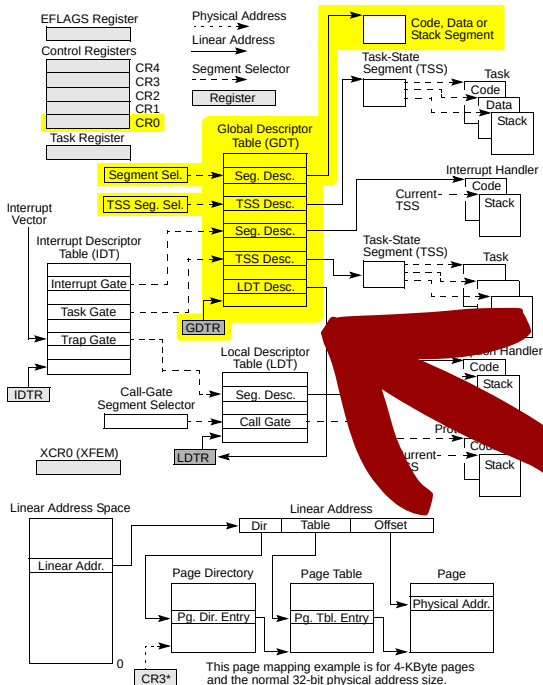


- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor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

## Type

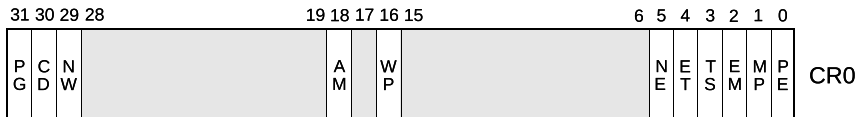
Decimal	Type Field				Descriptor Type	Description
	11	10	9	8		
	E		W	A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, 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



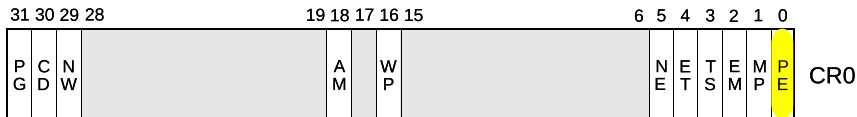


Usted  
está  
aquí

# Pasar a modo protegido



# 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?

# Pasar a Modo Protegido - ¿Por qué cataplum?

¿Cómo sabemos donde esta la GDT?

Cargar el registro GDTR utilizando LGDT

¿Qué tiene la GDT?

# Pasar a Modo Protegido - ¿Por qué cataplum?

¿Cómo sabemos donde esta la GDT?

Cargar el registro GDTR utilizando LGDT

¿Qué tiene la GDT?

Al menos, un descriptor nulo, un descriptor de código y uno de datos

¿Cuál es la próxima instrucción a ejecutar?

# Pasar a Modo Protegido - ¿Por qué cataplum?

¿Cómo sabemos donde esta la GDT?

Cargar el registro GDTR utilizando LGDT

¿Qué tiene la GDT?

Al menos, un descriptor nulo, 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?

# Pasar a Modo Protegido - ¿Por qué cataplum?

¿Cómo sabemos donde esta la GDT?

Cargar el registro GDTR utilizando LGDT

¿Qué tiene la GDT?

Al menos, un descriptor nulo, 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:

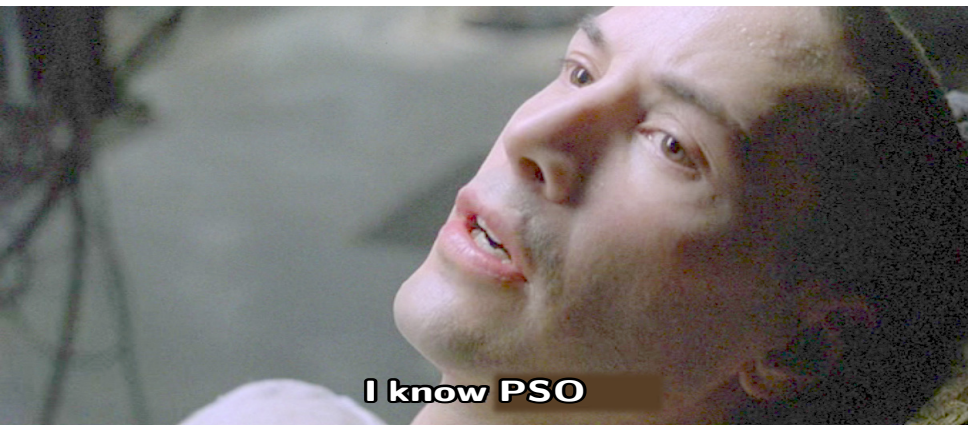
¡GRAN SALTO!

.... ; esto se ejecuta en modo protegido



# Pasar a Modo Protegido - Pasos

- 0- Completar la GDT
- 1- Deshabilitar interrupciones (CLI)
- 2- Cargar el registro GDTR con la dirección base 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)



**I know PSO**

¡¡¡Gracias!!!

¿Preguntas?