

Bootloader

Se inicia el equipo, se ejecuta el BIOS, corre los POST, se copia el bootloader en la posición 0x1000, se copia el archivo kernel.bin a la pos 0x1200, se salta y se ejecuta el kernel

Modo Real vs Modo Protegido

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:

(segmento << 4) + offset

Por ejemplo:

(0x07C0 << 4) + 0x0120 = 0x7C00 + 0x0120 = 0x7D20

	Modo Real	Modo Protegido
Memoria disponible	1Mb	4gb
Privilegios	No	4 niveles
Manejo de Interrupciones	Rutinas de at.	Rutinas de at. Con priv.
Instrucciones	Todas	Depende del nivel de prot

Pasaje a modo protegido

```
;Deshabilitar interrupciones
cli
;Habilitar A20
call habilitar_A20

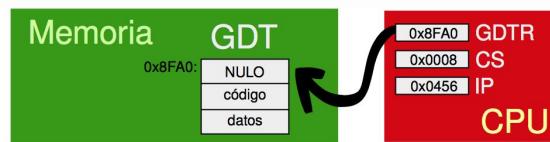
; Cargar la GDT
lgdt [GDT_DESC]

;Setear el bit PE del registro CR0
mov eax, cr0
or eax, 1
mov cr0, eax

; Saltar a modo protegido
jmp CODE_0:modoprotegido
```

BITS 32

modoprotegido:
; Establecer selectores de segmentos
xor eax, eax
mov ax, DATA_0
mov ds, ax
mov es, ax
mov ss, ax
mov gs, ax
mov ax, VIDEO
mov fs, ax



¿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!

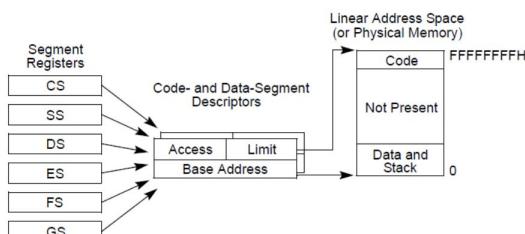
Administración de la Memoria

- Segmentación provee un mecanismo para separar distintas partes de un programa
- Paginación provee un mecanismo para la paginación por demanda, sistema de memoria virtual donde porciones del programa se mapean a memoria física según se necesita. Paginación tambien puede usarse para separar distintas tareas.

Segmentación

Segmentación Flat

El sistema y las aplicaciones tienen acceso a un espacio de direcciones lineal sin segmentar. Para implementar este modelo es necesario dos registros de segmentos, el de código y datos. Estos dos registros mapean el mismo espacio de dirección y el valor de ellos es cero y tienen un límite de 4Gb. Este mecanismo nunca arroja excepciones de fuera de límite, incluso cuando en la dirección no existe memoria física



Segmentación flat protegida

similar al modelo básico, pero los límites tienen valor igual a la cantidad de memoria física. De esta forma cuando se quiere acceder a direcciones que no existen en la memoria física se arroja una excepción de fuera de límite. A este modelo se le puede agregar adicionales protecciones para separar código de usuario y de supervisor. Por ejemplo para separar código y datos de usuario/supervisor se requieren cuatro registros de segmento.

GDT

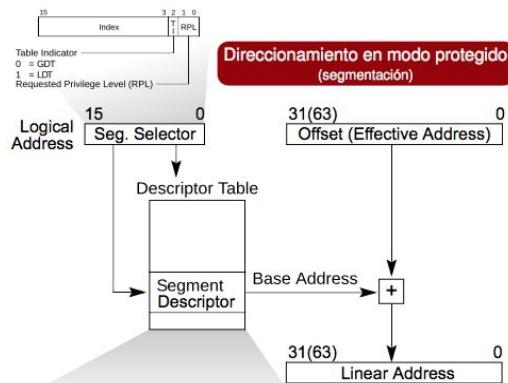
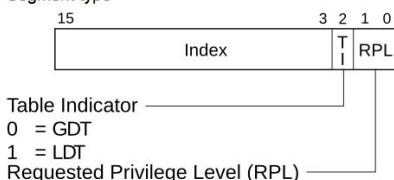
Es una **tabla** alojada en memoria donde cada entrada es de **8 bytes** y define alguno de los siguientes **descriptores**:

- 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

31	24 23 22 21 20 19	16 15 14 13 12 11	8 7	0
Base 31:24	G / D B	L A V L	Seg. Limit 19:16	P D P L S Type Base 23:16
31	16 15			0
Base Address 15:00		Segment Limit 15:00		

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 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

En la GDT poner:

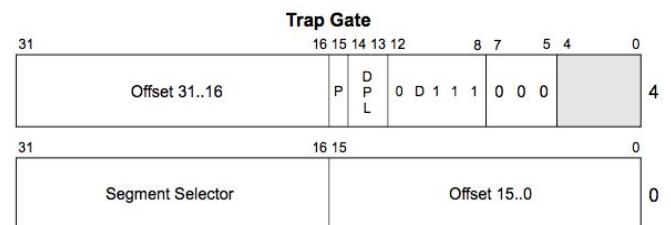
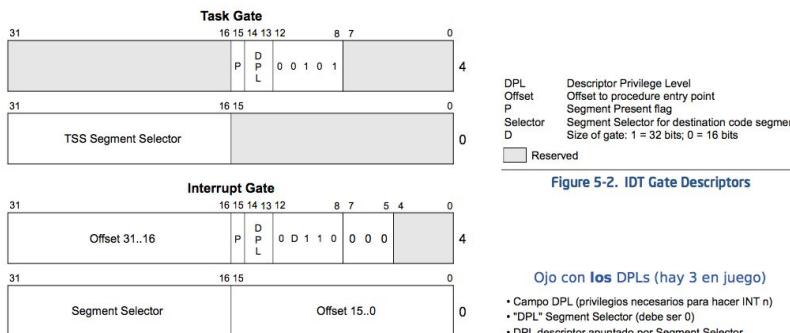
- primero una entrada en NULL
- entrada de código nivel 0
- entrada de código nivel 3
- entrada de datos nivel 0
- entrada de datos nivel 3

Interrupciones

- Soporta 256 interrupciones
- Se usa la tabla IDT que almacena descriptores de interrupción
- el registro IDTR almacena la dirección de la IDT
- La int 1 y las 20-31 están reservadas por intel
- De 32-255 son interrupciones definidas por el usuario

Tipos de interrupciones

- Fault: puede corregirse permitiendo al programa retomar la ejecución de esa instrucción sin perder continuidad. El procesador guarda en la pila la dirección de la instrucción que produjo la falla.
- Traps: Excepción producida después de una instrucción de trap. Algunas permiten al procesador retomar la ejecución sin perder continuidad. El procesador guarda en la pila la dirección de la instrucción a ejecutarse luego de la instrucción trapeada.
- Aborts: Excepción que no siempre puede determinar la instrucción que la causó, ni permite recuperar la ejecución de la tarea que la causó. Reporta errores severos de hardware o inconsistencias en tablas del sistema.



Error Code



Interrupciones Externas

Al igual que las otras interrupciones se declaran en la IDT y cada interrupción debe tener su rutina de atención. Además se debe remapear el PIC y activar las interrupciones con el comando sti.

Es un dispositivo (chip) al que le llegan las interrupciones de los demás dispositivos de la máquina. El PIC puede atender 15 interrupciones (IRQ0 - IRQ15, la IRQ2 no cuenta ya que es donde se conecta otro PIC en cascada). Por defecto, estas IRQs están mapeadas a las interrupciones 0x8 a 0xF (PIC1) y de 0x70 a la 0x77 (PIC2). En el tp usamos las funciones restear_pic, habilitar_pic y deshabilitar_pic.

- Preservar los registros que vayamos a romper (la interrupción debe ser transparente!)
- Comunicar al PIC que ya se atendió la interrupción (EOI) (rutina fin_intr_pic1) del archivo pic.h).
- Realizar la tarea correspondiente a la interrupción.
- Restaurar los registros.
- Retornar de la interrupción. (iret)

NOTA: No es necesario deshabilitar las interrupciones al comienzo de la rutina, ya que el procesador lo hace automáticamente (si definimos el descriptor como un interrupt gate). Tampoco es necesario volver a habilitarlas al finalizar.

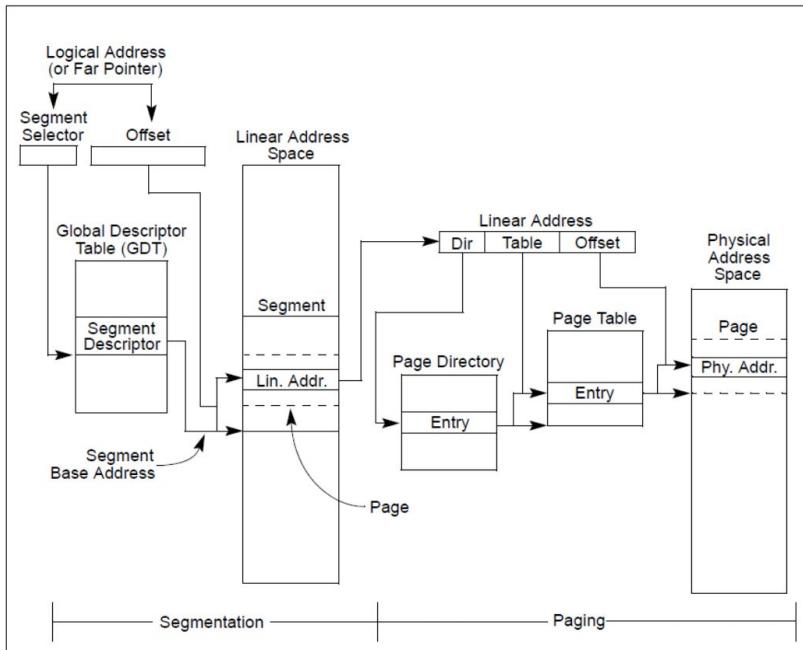
Cuando es una interrupción del teclado, obtenemos un scan code que nos dice que tecla es la que está siendo apretada (no exactamente, hace falta hacer una transformación con el código)

```
call idt_inicializar
; Cargar IDT
lidt [IDT_DESC]
call resetear_pic
call habilitar_pic

; Cargar tarea inicial
mov ax, 0x0068
ltr ax

; Habilitar interrupciones
sti
```

Paginación



La paginación es una forma de mapear el espacio lineal de dirección en pequeñas partes a la memoria virtual y disco secundario. Cuando se usa paginación el procesador divide el espacio lineal en fragmentos (4kb, 2MB o 4MB) que pueden ser mapeados a la memoria física o disco. El programa usa direcciones lógicas, que luego el procesador traduce a direcciones físicas. Al ir obteniendo la dirección física, se puede saber si la página o tabla está en memoria física, gracias a los datos que hay en el directorio o tabla de páginas que indican si está o no en memoria.

Traducción de dirección lógica a lineal

En el modo protegido el procesador usa dos etapas para traducir una dirección lógica a física.

La primera etapa **logical-address-translation** y la segunda **linear-address-translation**. Incluso con el mínimo uso de segmentos el procesador mapea la dirección física por medio de las direcciones lógicas. Una dirección lógica consiste en un selector de segmento de 16 bits y un offset de 32 bits.

Para traducir una dirección lógica a una lineal el procesador realiza lo siguiente :

- Use el offset en el selector de segmento para ubicar el descriptor de segmento en la GDT o LDT (este paso solo se requiere cuando el procesador carga un nuevo selector de segmento en el registro de segmento)
- Examina el descriptor de segmento para chequear privilegios de acceso y que el offset esté dentro de los

Límites

- Del descriptor de segmento usa la base para sumar con el offset (el de 32 bits) y forma la dirección lineal.

Traducción de dirección lineal a física

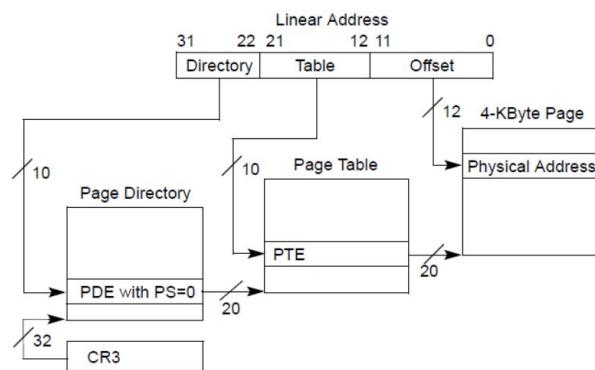
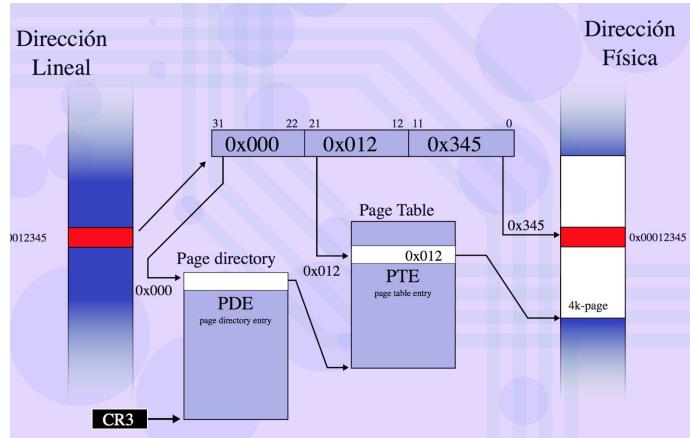
Cuando se obtiene la dirección lineal y se utilizan páginas de 4kb la dirección se separa en tres partes

- bits 22 a 31 proveen un offset para una entrada en el directorio de páginas, que dicha entrada corresponde a la dirección física a una tabla de páginas.

- bits del 12 al 21 proveen un offset para una entrada en la tabla de página que contiene la base de la página.

- bits 0 al 11 es un offset que junto con la base de la página se suman para obtener la dirección física del dato.

Cuando se utiliza páginas de 4Mb, la dirección lineal se separa en dos partes. Una para el directorio y la otra es el offset. El funcionamiento es similar a página de 4kb, solo que tiene un nivel menos.



Mappear páginas

- Base del Page directory = cr3
- Page directory index = virtual >> 22
- Base de page table = directorio[indice].address << 12
- Page table index = (virtual << 10) >> 22
- Tablas[indice].address = fisica >> 12

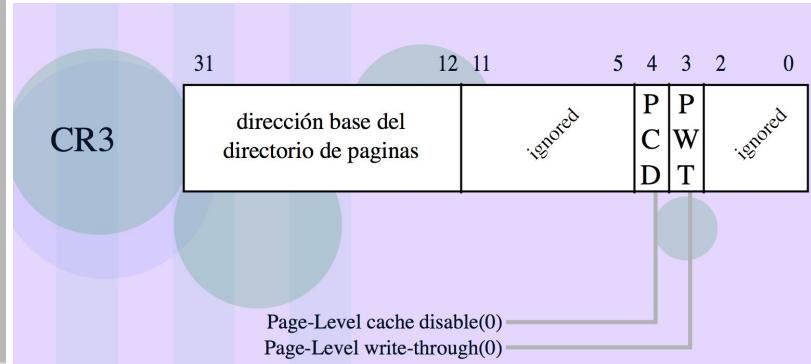
```
void mmu_mappear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica, int read_write, int user_supervisor){  
    // desarmar la virtual en sus indices  
    unsigned int pde_i = virtual >> 22;  
    unsigned int pte_i = (virtual << 10) >> 22;  
    pde* directorio = (pde*) cr3;  
    if(directorio[pde_i].present == 0){  
        mmu_inicializar_directorio(pde_i, cr3, read_write, user_supervisor);  
    }  
    pte* tablas = (pte*) (directorio[pde_i].address << 12);  
    mmu_inicializar_tabla(pde_i, pte_i, cr3, read_write, user_supervisor);  
    tablas[pte_i].address = fisica >> 12;  
    // attr  
  
    tlbflush();  
}
```

```
void mmu_inicializar_tabla(int pde_i,int pte_i, int cr3, int read_write, int user_supervisor){  
    pde* directorio = (pde*) cr3;  
    pte* tabla = (pte*) (directorio[pde_i].address << 12);  
    tabla[pte_i].present = 1;  
    tabla[pte_i].read_write = read_write;  
    tabla[pte_i].user_supervisor = user_supervisor;  
    tabla[pte_i].accesed = 0;  
    tabla[pte_i].dirty = 0;  
    tabla[pte_i].page_cache_disable = 0;  
    tabla[pte_i].page_write_through = 0;  
    tabla[pte_i].pat = 0;  
    tabla[pte_i].global = 0;  
    tabla[pte_i].ignored = 0;  
}
```

```

void mmu_inicializar_directorio(int indice, int cr3, int
read_write, int user_supervisor) {
    unsigned int pde_i = indice;
    pte* tabla = (pte*) dame_libre();
    //pte* tabla = (pte*) KP_ADDRESS;
    pde* directorio = (pde*) cr3;
    directorio[pde_i].present = 1;
    directorio[pde_i].read_write = read_write;
    directorio[pde_i].user_supervisor = user_supervisor;
    directorio[pde_i].address = ((int) tabla) >> 12;
    directorio[pde_i].accesed = 0;
    directorio[pde_i].ignored1 = 0;
    directorio[pde_i].page_size = 0;
    directorio[pde_i].ignored2 = 0;
    directorio[pde_i].page_cache_disable = 0;
    directorio[pde_i].page_write_through = 0;
    directorio[pde_i].global = 0;
    for (int i = 0; i < 1024; i++) {
        tabla[i].present = 0;
    }
}

```



```

typedef struct pde_s {
    int present:1;
    int read_write:1;
    int user_supervisor:1;
    int page_write_through:1;
    int page_cache_disable:1;
    int accesed:1;
    int ignored1:1;
    int page_size:1;
    int global:1;
    int ignored2:3;
    int address:20;
} pde;

```

```

typedef struct pte_s {
    int present:1;
    int read_write:1;
    int user_supervisor:1;
    int page_write_through:1;
    int page_cache_disable:1;
    int accesed:1;
    int dirty:1;
    int pat:1;
    int global:1;
    int ignored:3;
    int address:20;
} pte;

```

```

; Habilitar paginacion
mov eax, 0x27000
;page_directory == 0x27000
mov cr3, eax

mov eax, cr0
or eax, 0x80000000
; habilito paginacion
mov cr0, eax

```

Cada tarea tiene su propio espacio lineal de dirección : Para lograr esto, gracias al TSS se guarda el PDBR (que contiene el CR3) y permite que cada tarea tenga su propio directorio de páginas, ya que al cargar o cambiar de tarea se cambia el valor del CR3 (que apunta al directorio de páginas)

Tareas y Scheduler

Una tarea está compuesta por:

1. Espacio de ejecución:
 - Segmento de código.
 - Segmento de datos/pila (uno o varios).
2. Segmento de estado (TSS):

Almacena el estado de la tarea (su contexto) para poder reanudarla desde el mismo lugar.

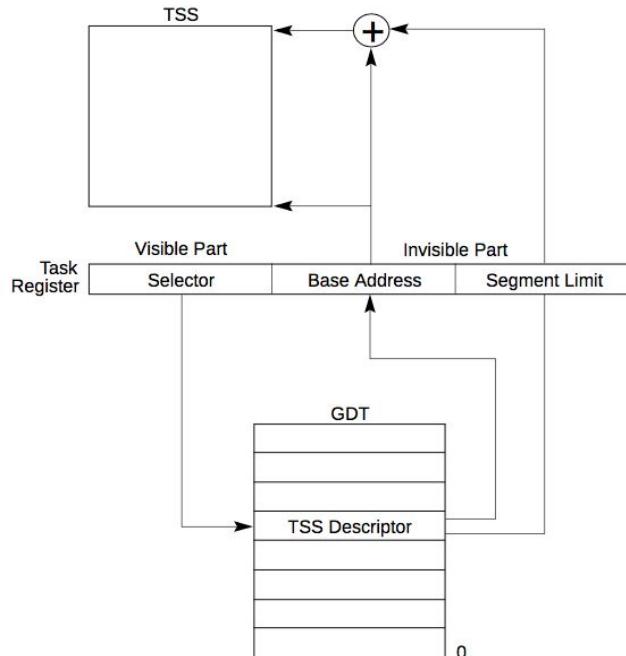
TSS

La TSS es un segmento. Debe estar descrito en la GDT del mismo modo que se describen los segmentos de código y datos.

El selector de segmento de la tarea que se está ejecutando actualmente se encuentra en el registro Task Register (TR).

Posee los valores para la pila (para usar en SS y ESP) para los niveles 0,1,2, el de nivel 3 estará en SS : ESP.

31	15	0
I/O Map Base Address	Reserved	T
Reserved	LDT Segment Selector	100
Reserved	GS	96
Reserved	FS	92
Reserved	DS	88
Reserved	SS	84
Reserved	CS	80
Reserved	ES	76
	EDI	72
	ESI	68
	EBP	64
	ESP	60
	EBX	56
	EDX	52
	ECX	48
	EAX	44
	EFLAGS	40
	EIP	36
	CR3 (PDBR)	32
Reserved	SS2	28
Reserved	ESP2	24
Reserved	SS1	20
Reserved	ESP1	16
Reserved	SS0	12
Reserved	ESP0	8
Reserved	Previous Task Link	4
		0



Conmutación de Tareas

1. Se necesita siempre una tarea inicial para dar una tss en donde se pueda guardar el contexto actua
2. Se ejecuta la instr jmp 0x20:0 (ejemplo)
3. Se busca el descriptor correspondiente en la GDT
4. Se lee el TR y se busca la TSS apuntada por el mismo
5. Se guarda el contexto actual (los registros)
6. Se busca el TSS apuntado por el descriptor de la tarea a ejecutar
7. Se obtiene el nuevo contexto
8. Se actualiza el TR
9. Continua la ejecución con el nuevo contexto

Al iniciar las tareas completar siempre EIP, ESP, selectores de segmento, CR3, EFLAGS EFLAG por defecto con interrupciones habilitadas: 0x202. Sin ints: 0x2

```

; Inicializar tss
call tss_inicializar
; Inicializar tss de la tarea Idle
call tss_inicializar_idle
;cargo las tareas en la gdt

; Cargar tarea inicial
mov ax, 0x0068
ltr ax
; Habilitar interrupciones
sti

;esto no se que es
push 0x10000
push 0x00
push 0x00

call mmu_inicializar_dir_pirata
xchg bx, bx
mov cr3, eax

; Saltar a la primera tarea: Idle
jmp 0x0070:0

```

Protección

Protección en la segmentación

CPL = Current Privilege Level, DPL del segmento de código que estoy ejecutando.

- El bit P tiene que estar en 1
- Chequear límite:
 - Granularidad = 1
 - Offset + bytes a leer - 1 ≤ ((límite + 1) << 12) -1.
 - Granularidad = 0
 - offset + bytes a leer - 1 ≤ límite.
- Segmento de datos -> Max(CPL,RPL) ≤ DPL
- Segmento de Código:
 - Non-conforming -> Max(CPL,RPL) = DPL
 - Conforming -> CPL ≥ DPL
- Chequear el type
- Chequear el bit de sistema

Protección en la paginación

- El bit de presente en 1
- El bit de R/W para ver si puedo leer o escribir
- El bit de U/S para ver si puede acceder con User o Supervisor (Modo supervisor si CPL ≤ 2)

El resultado se calcula según los acceso a ambas tablas

- Si la entrada del PD o PT tiene Supervisor → solo es accesible por supervisor.
- Solo si la entrada del PD y PT tiene User → es accesible por user.
- Solo si ambas entradas tienen Read Write → es posible escribir.

Protección en las interrupciones

- El bit de presente en 1
- Selector de segmento (que entrada de la GDT será utilizada para correr la int)
- DPL :Privilegio necesario para llamar a la int (CPL ≤ DPL). En ints por hardware se ignora.

Protección en switching de tareas

- Bit de presente en 1
- Solo es posible saltar a una tarea si el CPL ≤ DPL.