

# **Sistemas Operativos**

## **Memoria**

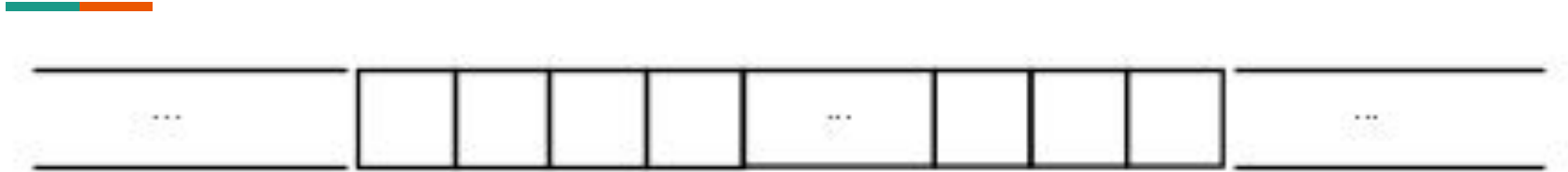
---

# Introducción

## Conceptos clave:

- Introducción histórica
- Address space
- Traducción de direcciones

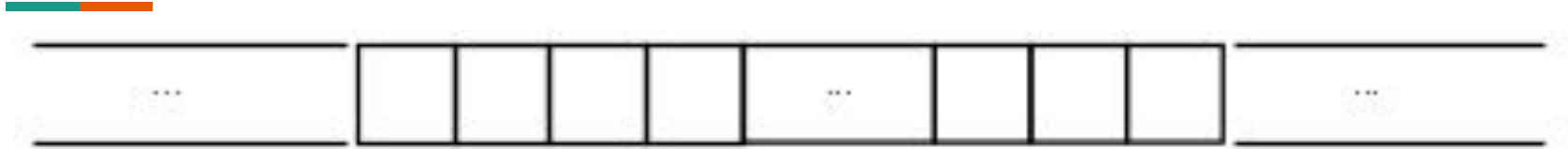
# ¿ Que es la Memoria?



El tema fundamental sobre la administración de memoria se encuentra en la forma adecuada de representarla.

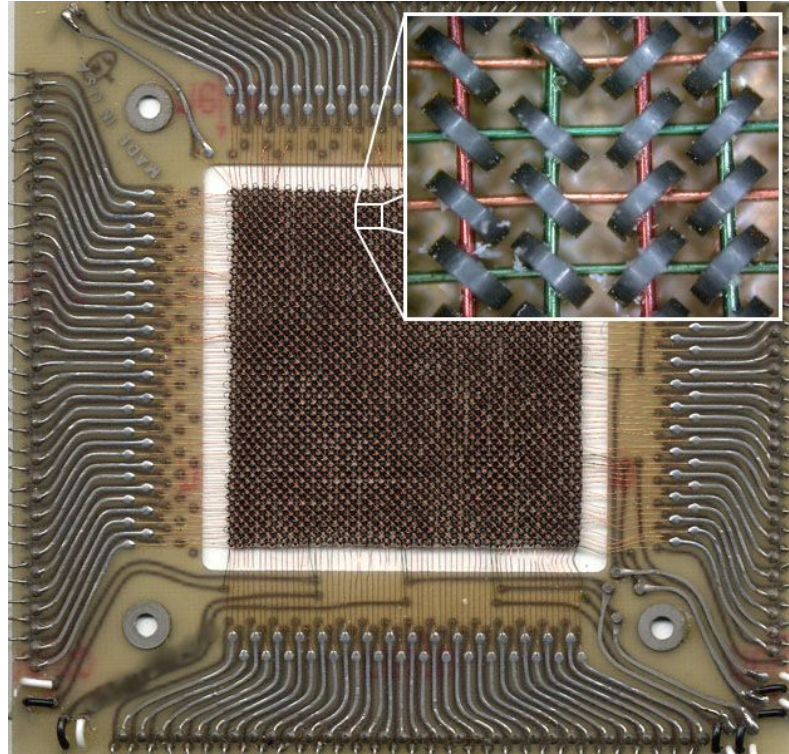
La memoria física puede ser imaginada como un arreglo de direcciones de memoria una detrás de otra.

# ¿ Qué es la memoria?



Esta “**abstracción**” es manejable mientras la cantidad de memoria necesitada está en un rango “manejable” y cual es ese rango.

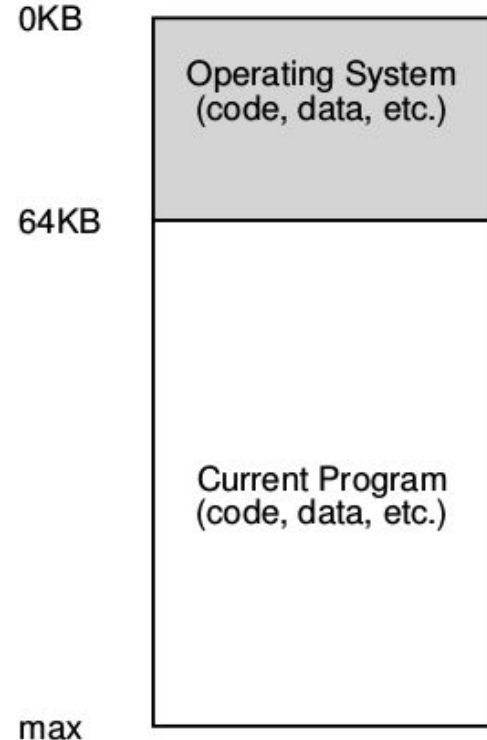
En DOS (Disk Operating System) ese rango lo daba la elección de una arquitectura determinada, la del 8086, que permite tener direcciones de memoria de hasta 2<sup>20</sup> bits, es decir 1 MegaByte de memoria.



memoria de núcleos de ferrita

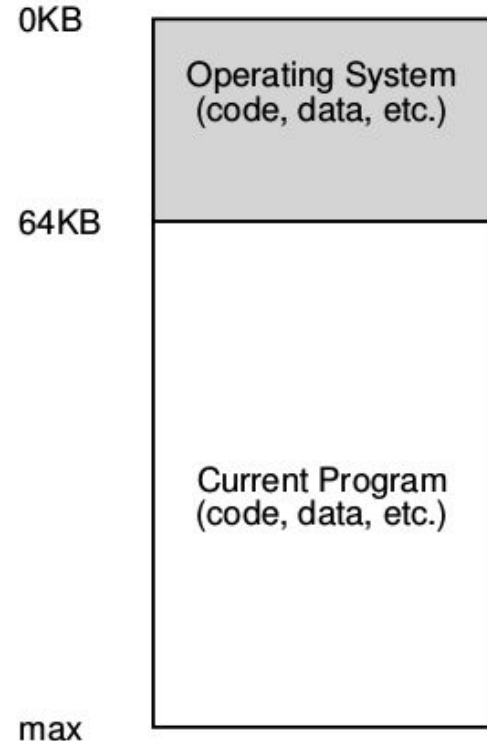
# Los Early Days

En los primeros Sistemas Operativos, en “the Early Days”, en los cuales el mismo S.O. se desplegaba en solo **64 KiloBytes** de memoria, *el resto de la misma estaba disponible para un único proceso ejecutándose a partir de los 64Kb de memoria física,*



# Los Early Days

El sistema operativo era más o menos un conjunto de funciones o rutinas (específicamente una biblioteca) que se alojaba en la memoria empezando en la dirección física 0, y después existía un único programa en ejecución (un proceso) que se encontraba en la memoria física de la computadora, esta memoria era el resto no utilizado por el sistema operativo.





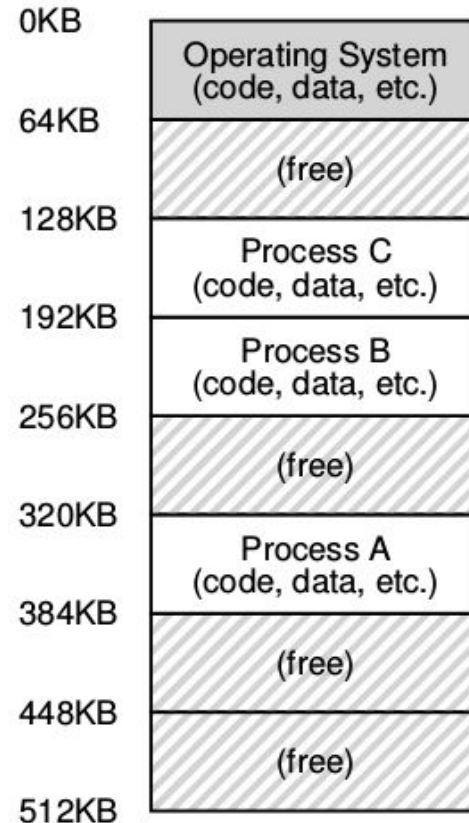
It used a paging mechanism to map the virtual addresses available to the programmer on to the real memory that consisted of 16,384 words of primary core memory with an additional 98,304 words of secondary drum memory.

**Atlas Univ. Manchester 1961**



# Multiprogramación

Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time on a uniprocessor. Since there is only one processor, **there can be no true simultaneous execution of different programs.** Instead, the operating system executes part of one program, then part of another, and so on. To the user it appears that all programs are executing at the same time.



# Multiprogramación



Más de un proceso estaba preparado para ser ejecutado en algún determinado momento, y el sistema operativo intercalaba dicha ejecución según la circunstancia. Haciendo esto se mejoró efectivamente el uso de la CPU, tal mejora en la eficiencia fue particularmente decisiva en esos días en la cual una computadora costaba cientos de miles o tal vez millones de dólares.

En esta era, múltiples procesos están listos para ser ejecutados un determinado tiempo según el S.O. lo decidiese en base a ciertas políticas de planificación o scheduling.

# Time Sharing

**Tiempo compartido** se refiere a **compartir de forma concurrente un recurso computacional** (tiempo de ejecución en la CPU, uso de la memoria, etc.) entre muchos usuarios por medio de las tecnologías de multiprogramación y la inclusión de interrupciones de reloj por parte del sistema operativo, permitiendo a este último acotar el tiempo de respuesta del computador y limitar el uso de la CPU por parte de un proceso dado.



COMPUTER

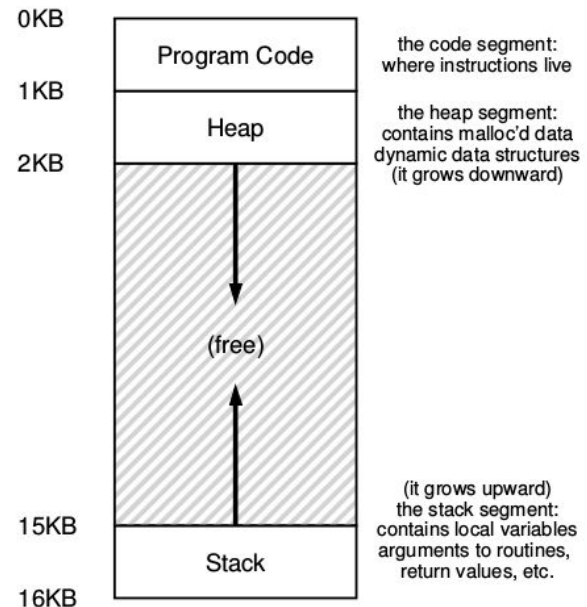


Time Sharing - IBM 704 - primer sistema Time Sharing

# El Espacio de Direcciones o Address Space

Crear un mecanismo que permita que la memoria física de una computadora sea utilizada de forma fácil y eficiente llevó paulatinamente a concebir el concepto de **espacio de direcciones**, la abstracción para la memoria.

El **Address Space** de un proceso contiene todo el estado de la memoria de un programa en ejecución.



# Virtualización de Memoria



El sistema operativo tiene que virtualizar memoria... pero lo tiene que hacer con estilo, para ello una de las metas principales de la virtualización es la transparencia.

El sistema operativo debería implementar la virtualización de memoria de forma tal que sea invisible al programa que se está ejecutando; es más, el programa debe comportarse como si él estuviera alojado en su propia área de memoria física privada.

Pero detrás de escena, el sistema operativo y el hardware hacen todo el trabajo para multiplexar memoria a lo largo de los diferentes procesos y por ende implementa la ilusión.

# Virtualización de Memoria



Otra meta importante de la virtualización es la **eficiencia**. El sistema operativo debe esforzarse para hacer que la *virtualización sea lo más eficiente posible en términos de tiempo* (no hacer que los programas corran más lentos) *y espacio* (no usar demasiada memoria para las estructuras necesarias para soportar la virtualización).

# Virtualización de Memoria



Por último, una tercera meta de la virtualización de memoria es la **protección**. El sistema operativo tiene que asegurarse de proteger a los procesos unos de otros como también proteger al sistema operativo de los procesos. Cuando un proceso realiza un load, un store, o un fetch de una instrucción este no tiene que ser capaz de hacerlo o afectar de ninguna forma al contenido de la memoria del proceso o del sistema operativo.



# Virtualización de Memoria



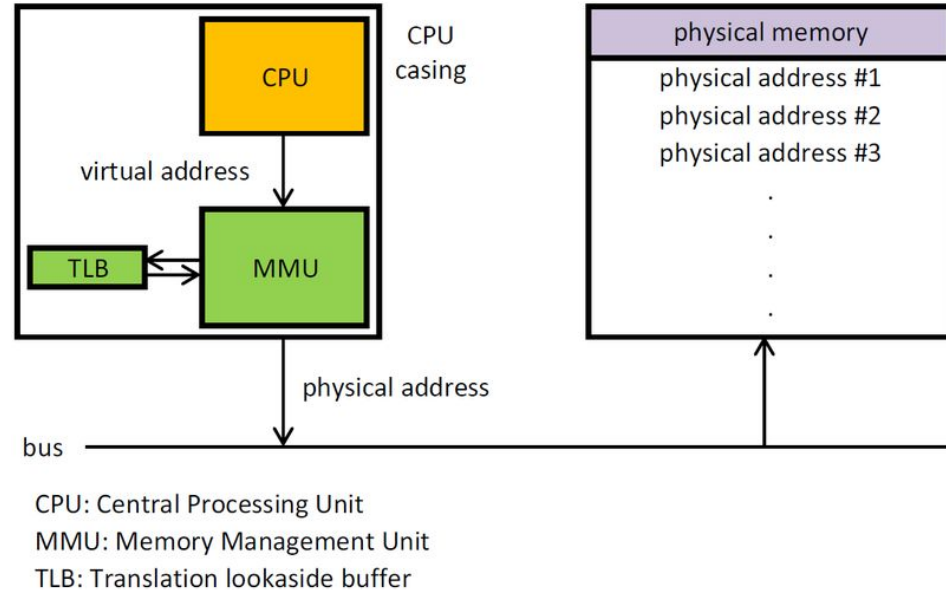
Existen dos puntos importantes a la hora de virtualizar memoria:

- la flexibilidad
- la eficiencia

Para llegar a ellos un buen mecanismo de virtualización de memoria debe ser lo más flexible y eficiente posible.

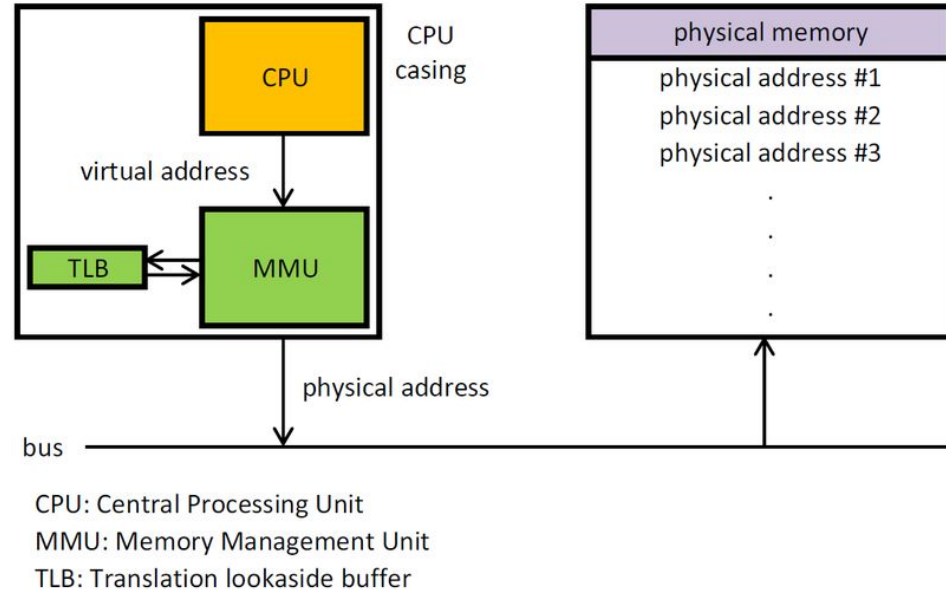
# Address Translation

Con esta técnica el hardware transforma cada acceso a memoria, transformando la virtual address que es provista desde dentro del espacio de direcciones en una physical address en la cual la información deseada se encuentra realmente almacenada.

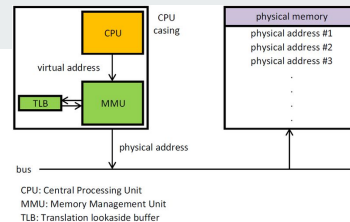


# Address Translation

Entonces, en todos y por cada una de las referencias a memoria un address translation es realizada por el hardware para redireccionar las referencias a la memoria de la aplicación hacia las posiciones reales en la memoria física.



# Address Translation



Es evidente, que el hardware por sí solo no puede virtualizar la memoria. Éste, sólo provee un mecanismo de bajo nivel para poder hacerlo eficientemente.

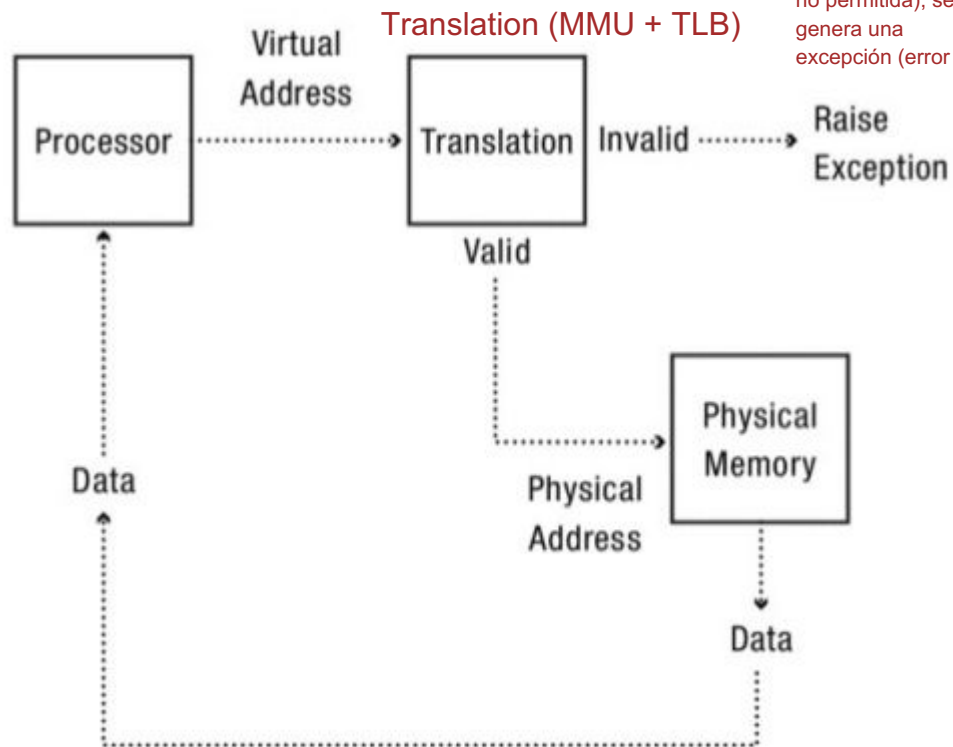
El sistema operativo tiene que involucrarse en los puntos claves para:

- **setear al hardware** de forma correcta para que esta traducción se de lugar;
- por eso debe entonces gerenciar la memoria, **manteniendo información de en qué lugar hay áreas libres y en qué lugar hay áreas en uso**;
- y por último **intervenir de forma criteriosa como mantener el control sobre toda la memoria usada**.

# Address Translation

El S.O. está en el medio de ese proceso y determina si el mismo se ha realizado correctamente. Lo que hace es gerenciar el manejo de la memoria:

- Manteniendo registro de qué parte está libre.
- Qué parte está en uso.
- Manteniendo el control de la forma en la cual la memoria está siendo utilizada.



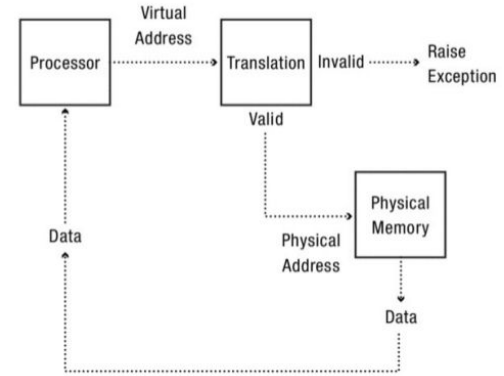
si el proceso intenta acceder a memoria no permitida, se genera una excepción (error)

# Address Translation

Formalmente, el proceso de address translation es un mapeo entre los elementos de un espacio de direcciones virtuales de N-elementos (VAS) y un espacio de direcciones físicas de M-elementos (PAS):

$MAP: VAS \Rightarrow PAS \cup \emptyset$

$$MAP(A) = \begin{cases} A' & \text{si los datos en la dir. virtual } A \text{ están presentes la dir. física } A' \text{ de PAS} \\ \emptyset & \text{si los datos de la dir. virtual } A \text{ no están presentes en la memoria física.} \end{cases}$$



---

# Implementaciones de memoria virtual

## Conceptos clave:

- Base and Bound
- Memoria segmentada
- Memoria paginada
- Memoria paginada multinivel

# Hacia una Flexible Address Translation



Para ir ganando comprensión sobre el *hardware-based address translation*, se verá su primera implementación, introducida en las primera máquinas que utilizaban time-sharing hacia el fin de los años 50, es una idea simple llamada base y segmento también puede ser vista como *dynamic reallocation*.

Específicamente solo se necesitan dos registros de hardware dentro de cada cpu: Uno llamado registro base y el otro registro límite o Segmento.



# Base and Bound



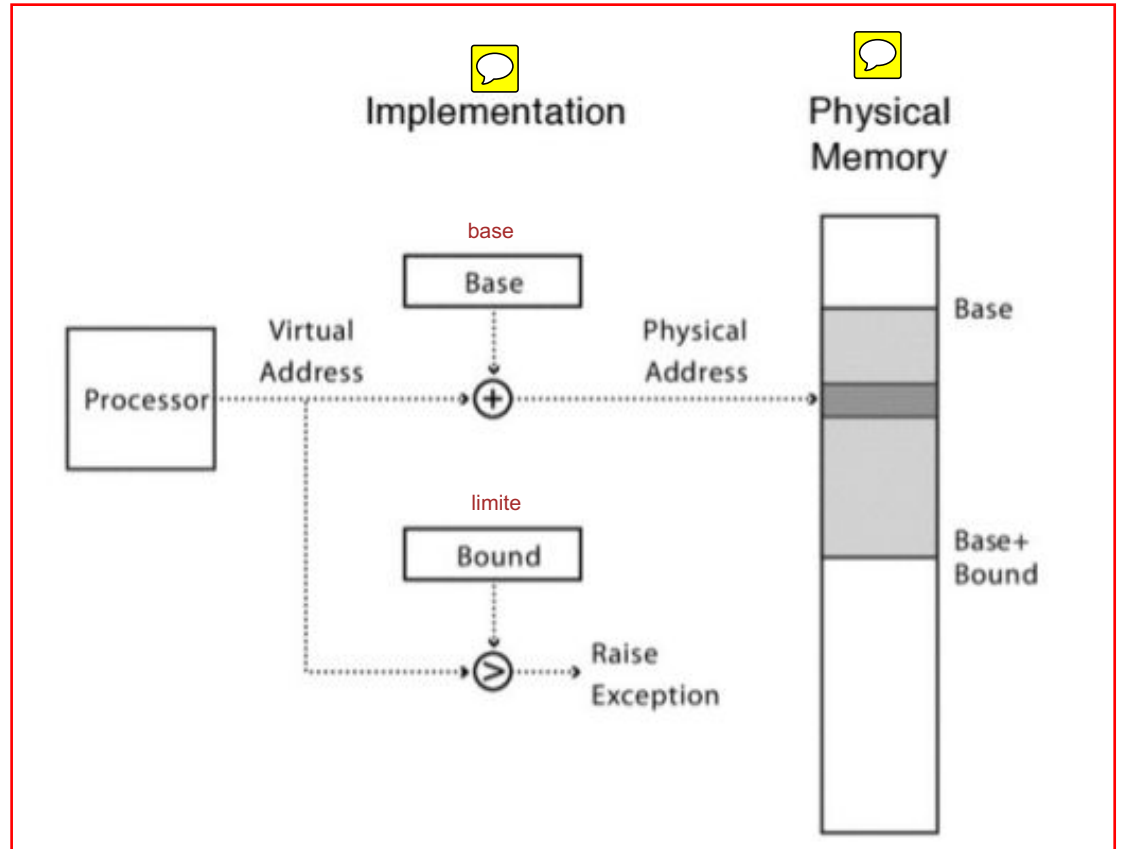
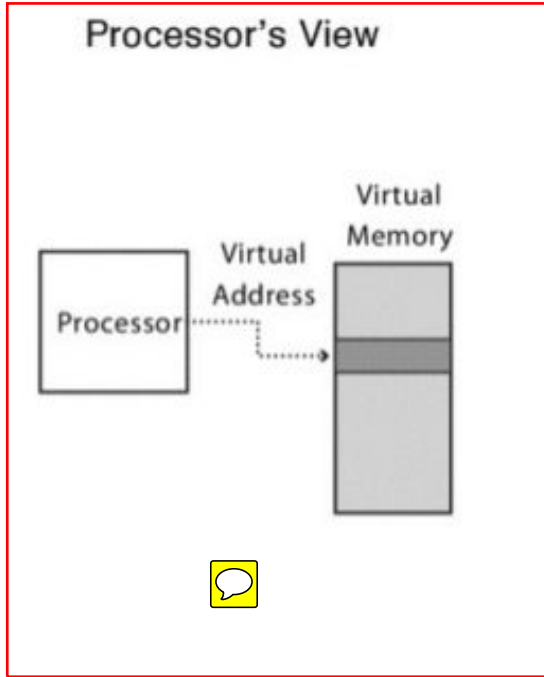
Base = Dirección base (de inicio)

Bound = Límite (la cantidad máxima o el final permitido)

Específicamente solo se necesitan dos registros de hardware dentro de cada cpu:

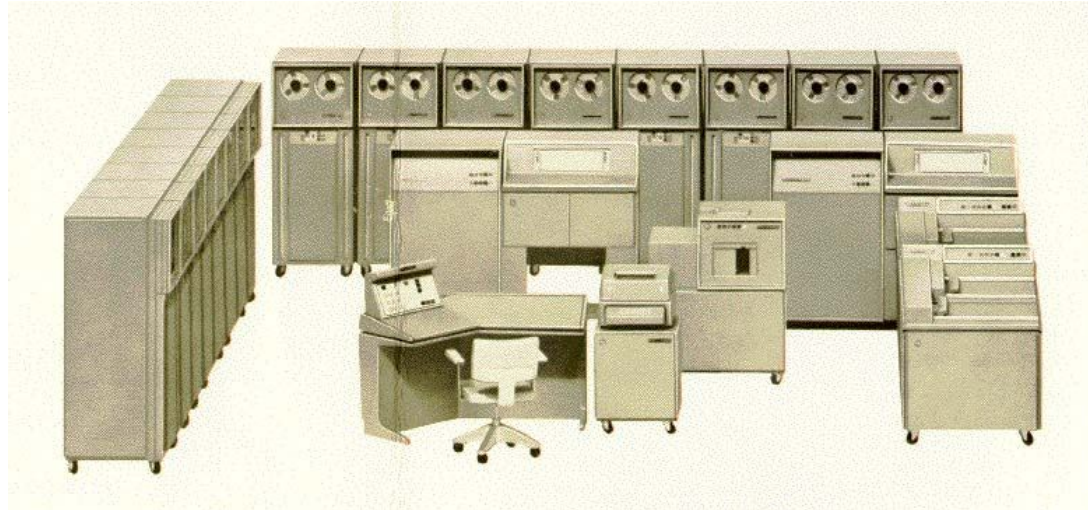
- registro base
- registro límite o Segmento.

Este par base-límite va a permitir que el address space pueda ser ubicado en cualquier lugar deseado de la memoria física, y se hará mientras el sistema operativo se asegura que el proceso solo puede acceder a su address space.



Base and Bound (segmentación)

# Segmentación B5000



# Address Translation con Tabla de Segmentos

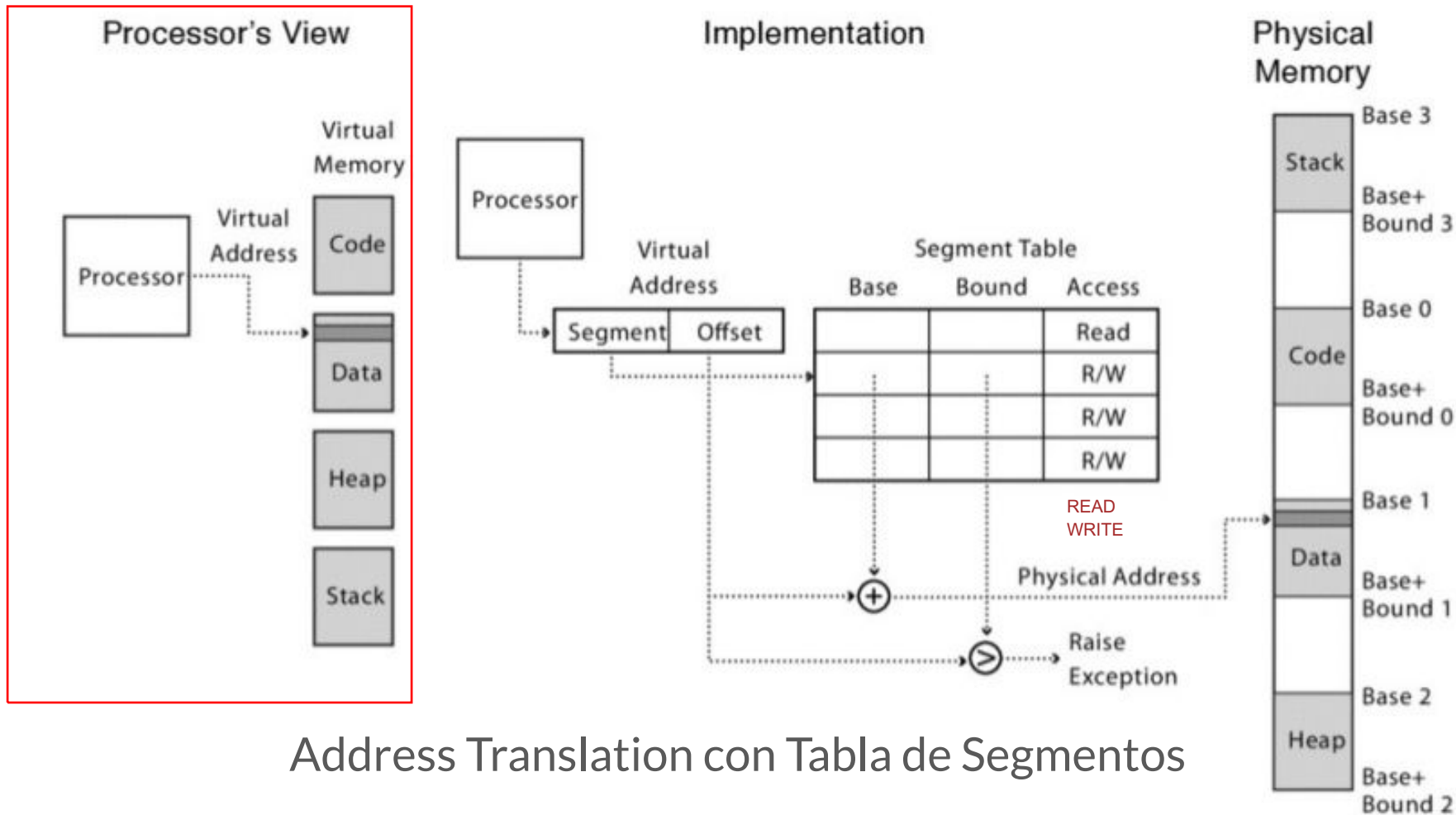


El problema de la técnica anterior es que se tiene un solo registro base y solo un segmento. La mejora a este método es mediante la aplicación de un pequeño cambio: en vez de tener un solo registro límite, se tiene un arreglo de pares de (registro base, segmento) por cada proceso.

Una dirección virtual tiene dos componentes:

**un número de segmento**: un offset de segmento

El número de segmento es el **índice de la tabla** para ubicar el inicio del segmento en la memoria física. El registro bound es chequeado contra la suma del registro base+offset para prevenir que el proceso lea o escriba fuera de su región de memoria.



Address Translation con Tabla de Segmentos

# Address Translation con Tabla de Segmentos



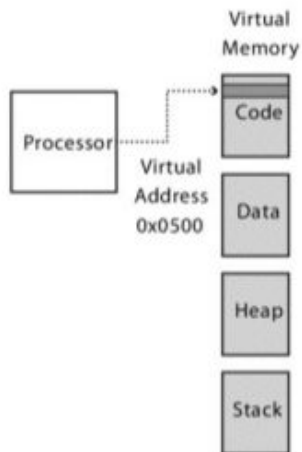
En una dirección virtual utilizando esta técnica, los bit de más alto orden son utilizados como índice en la tabla de segmentos. El resto se toma como offset y es sumado al registro base y comparado contra el registro bound.

El número de segmentos depende de la cantidad de bits que se utilizan como índice.

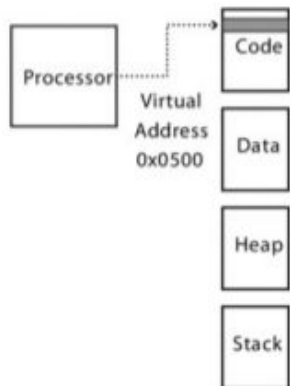
El error de **Segmentation Fault** era un error que se daba cuando, en las máquinas que implementan segmentación, se quería direccionar una posición fuera del espacio direccionable. Increíblemente este error aún se utiliza en sistemas operativos que no usan segmentación.

## Processor's View

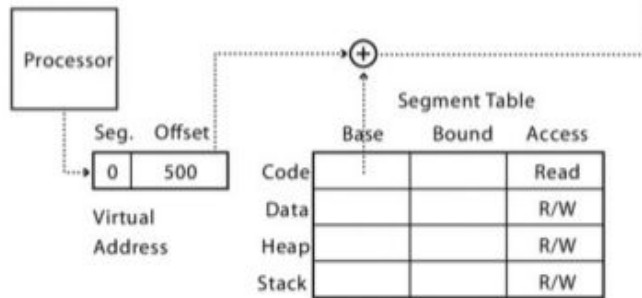
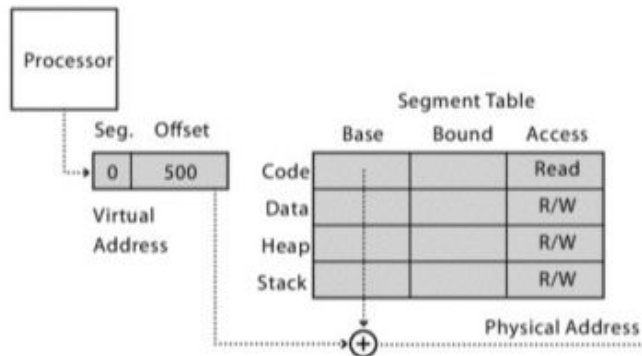
Process T's View



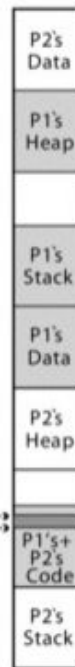
Process Z's View



## Implementation



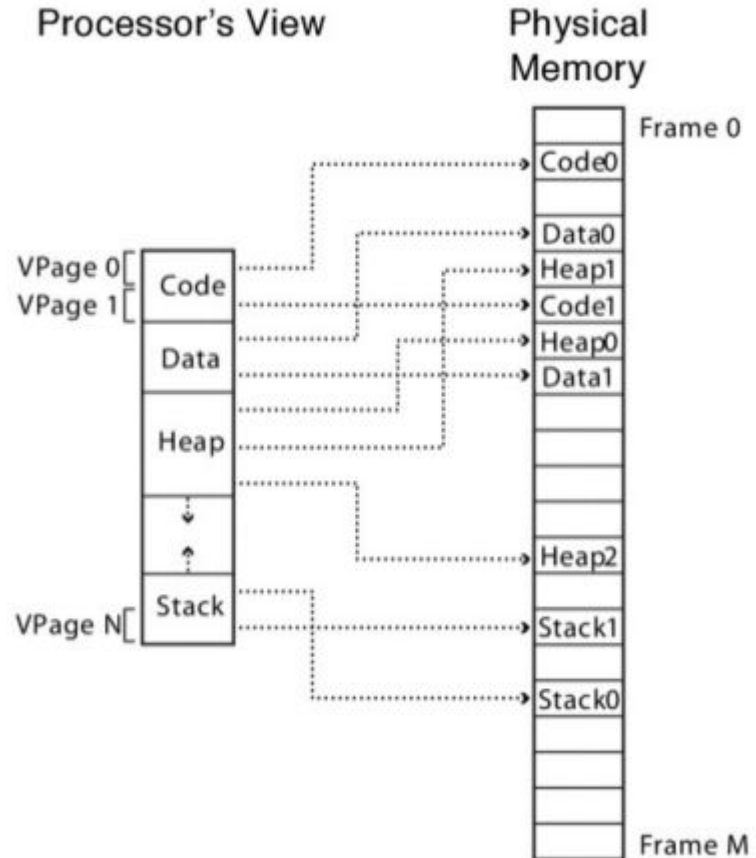
## Physical Memory



# Memoria Paginada

Con la paginación, la memoria es reservada en pedazos de tamaño fijo llamados page frames.

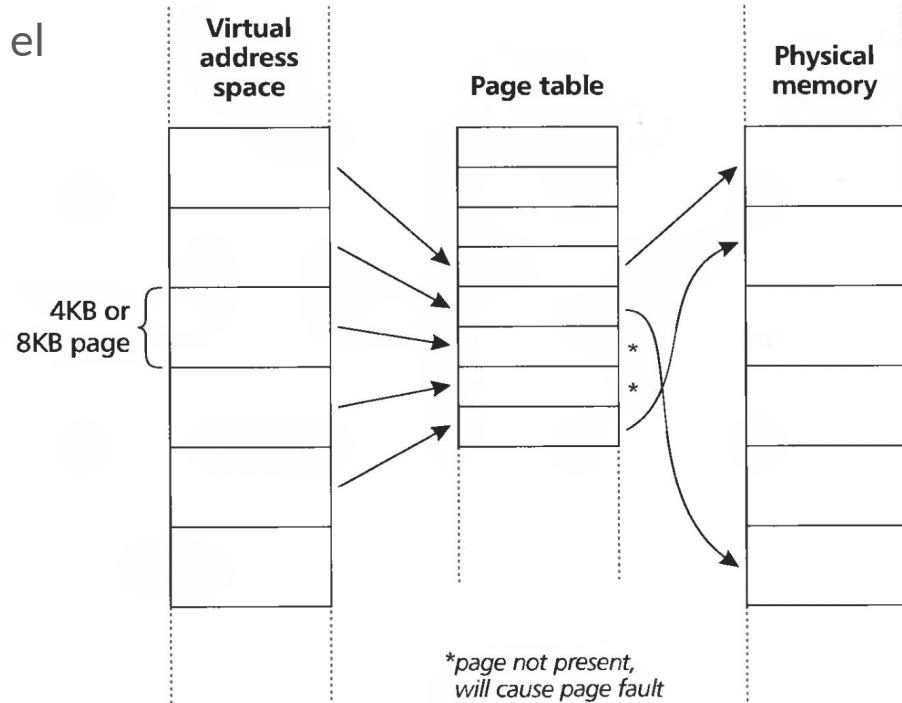
Cuadros de página





# Memoria Paginada

Las paginas y los frames tienen el mismo tamaño (eg. 4 Kb)



# Memoria Paginada



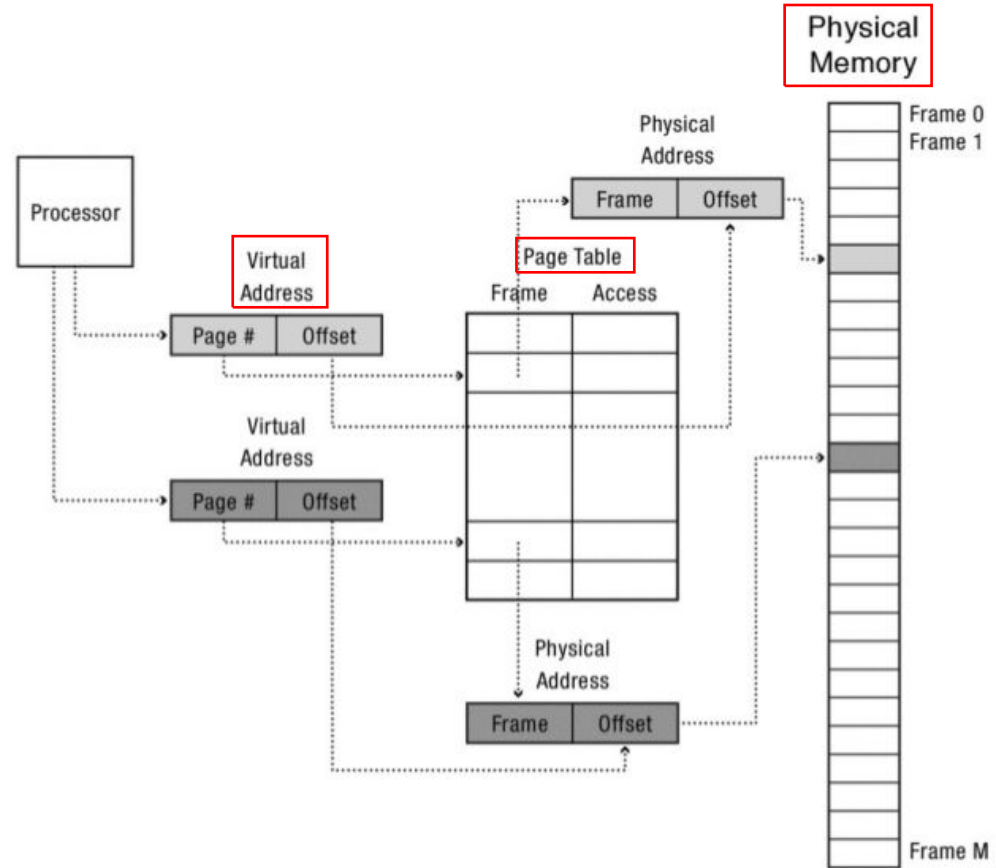
- Una alternativa a la memoria segmentada es la memoria paginada.
- El address translation es similar a como se trabaja con la segmentación.
- En vez de tener una página de segmentos cuyas entradas contienen punteros a segmentos, hay una tabla de páginas por cada proceso cuyas entradas contienen punteros a las page frames.
- Teniendo en cuenta que los page frames tienen un tamaño fijo, y son potencia de 2, las entradas en la page table sólo tienen que proveer los bit superiores de la dirección de la page frame. De esta forma van a ser más compactos. No es necesario tener un límite; la página entera se reserva como una unidad.

# Memoria Paginada

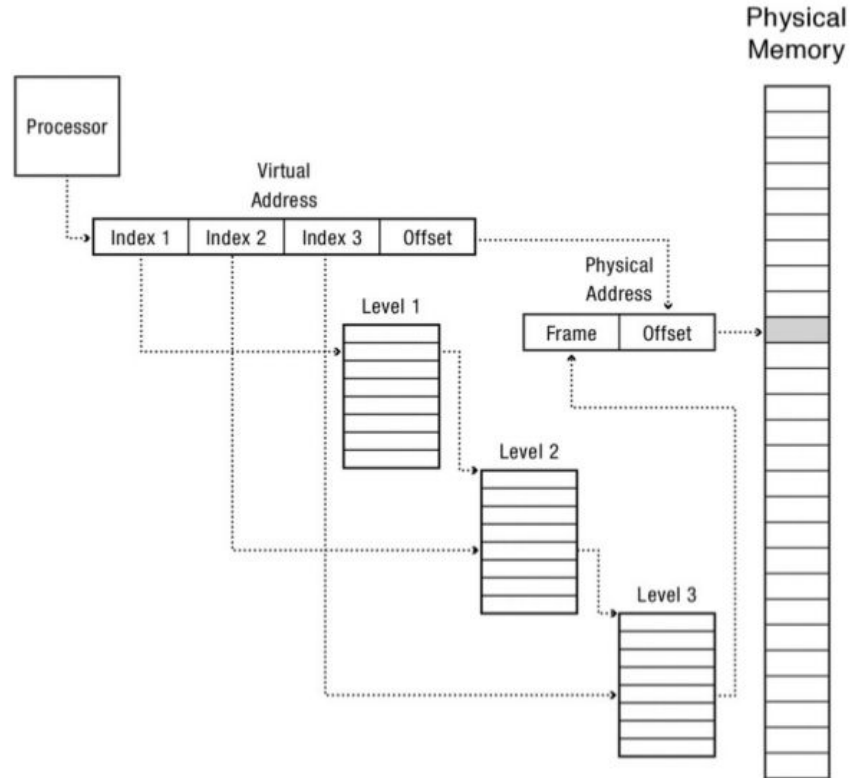
El número de la página virtual es el índice en la **page table** para obtener el **page frame** en la memoria física.

La dirección física está compuesta por la dirección física del **Frame Page** que se obtiene de la page table concatenado con el offset de la página que se obtiene de la virtual address. El sistema operativo maneja los accesos a la memoria.

Una de las cosas que pueden parecer raras sobre la paginación es que si bien el programa cree que su memoria es lineal, de hecho, su memoria está desparramada por toda la memoria física como si fuera un mosaico.



# Memoria paginada multinivel



---

# Casos de Estudio

## Conceptos clave:

- Base and Bound: 8086
- Memoria segmentada: x86
- Memoria paginada: x86

# Base and Bound en 8086

La arquitectura del procesador 8086 tiene un bus de datos de 20 bits , los registros tiene una longitud de 16 bits. Utiliza base and bound

Registros Generales:

AX, BX, CX, DX

Registros de Segmento

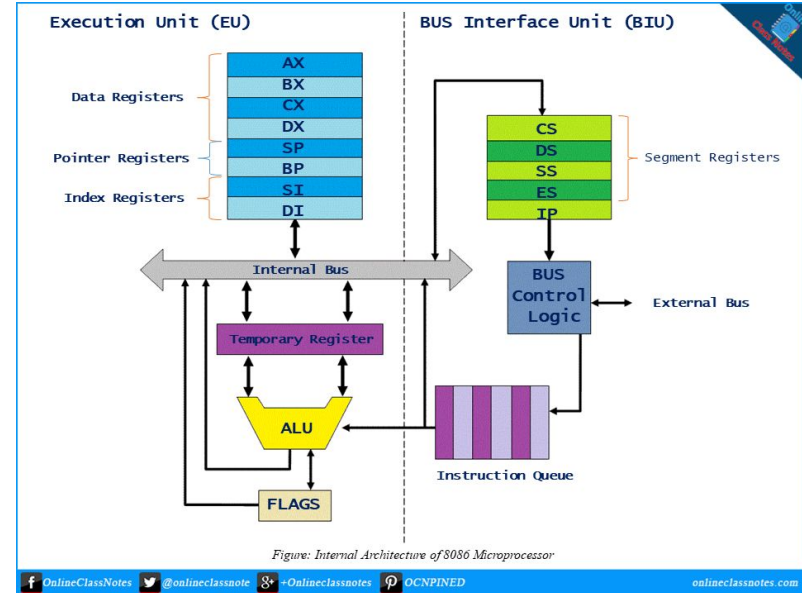
CS: Code Segment

DS: Data Segment

SS: Stack Segment

ES : Extra Data Segment

SI, DI, BP, SP, IP: Registros de Punteros y Registro de Índices. (Base Pointer, Stack Pointer y Instruction Pointer)



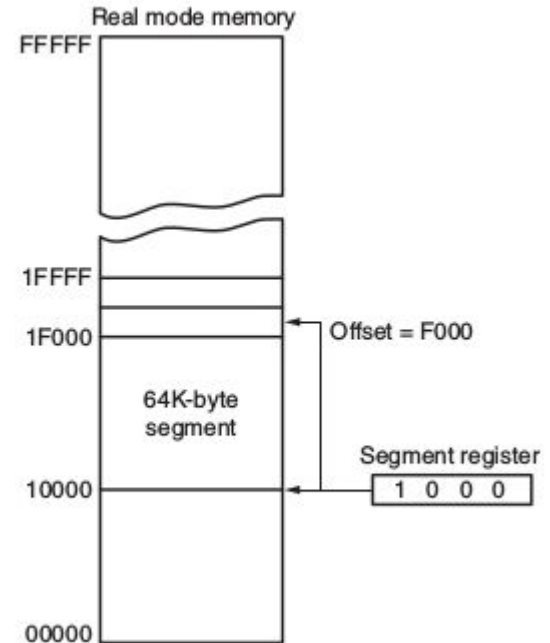
# Base and Bound en x86

Dado que era una arquitectura de 16 bits la cantidad máxima direccionable era  $2^{16}$  que es 65536 bytes. Como se quedaban cortos porque eso era solo 64k de memoria, decidieron agregar 4 bits más a las direcciones de memoria llevando el bus de datos a 20 bits

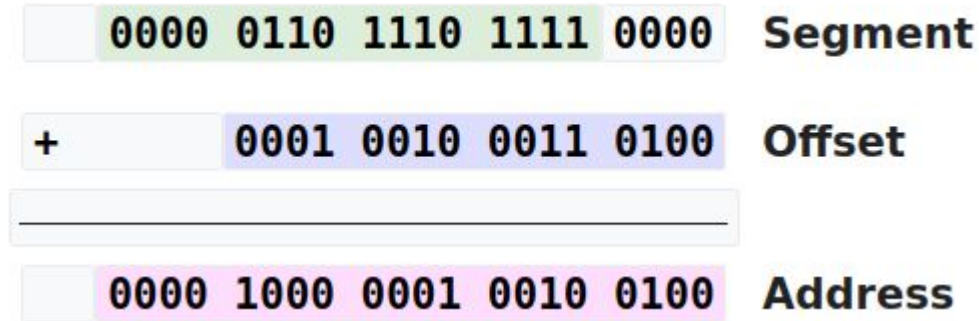
Una dirección virtual se considera como el valor del registro de segmento shifted 4 bit a izquierda, y luego se le suma el offset

1000:F00F virtual equivale a la dirección física:

```
10000
+ F00F
1F00F   direccion de 20 bits
```



# Base and Bound en x86



Combinaciones especiales:

- **CS:IP** localiza la próxima instrucción a ser ejecutada en modo real.
- **SS:SP** localiza la dirección del puntero al stack, a veces también puede ser SS:BP.
- **DS: BX,DI,SI** localizan el puntero a una dirección de memoria dentro del data address.
- **ES:DI** puntero al extra data address donde van los strings.



# Modo Protegido x86



En x86 además del **modo nativo**, real mode, donde las direcciones accedidas son direcciones físicas, existe otro modo llamado **Modo Protegido** en el cual se proveen los mecanismos necesarios para la protección de memoria.

Se implementó a partir del 80286

El modo protegido permite direccionar a datos y programas más allá de 1 Mb de memoria física, así como también dentro del primer mega de memoria física.

Es el sistema de virtualización que se utiliza es el de tabla de segmentos

# Memoria segmentada en x86



*Figure 2-1. Logical address translation*

La Unidad de Gestión de Memoria (**MMU**) transforma una dirección lógica en una dirección lineal mediante un circuito de hardware llamado **unidad de segmentación**; posteriormente, un segundo circuito de hardware llamado **unidad de paginación** transforma la dirección lineal en una dirección física.

# Memoria segmentada en x86



## Dirección lógica

Incluida en las instrucciones del lenguaje de máquina para especificar la dirección de un operando o de una instrucción. Este tipo de dirección encarna la conocida arquitectura segmentada  $80 \times 86$ , que obliga a los programadores de MS-DOS y Windows a dividir sus programas en segmentos. Cada dirección lógica consta de un segmento y un desplazamiento (o "offset") que denota la distancia desde el inicio del segmento hasta la dirección real.

## Dirección lineal (también conocida como dirección virtual)

Un único número entero sin signo de 32 bits que puede utilizarse para direccionar hasta 4 GB, es decir, hasta 4,294,967,296 celdas de memoria. Las direcciones lineales suelen representarse en notación hexadecimal; sus valores varían desde 0x00000000 hasta 0xffffffff. *(equivalente para 64 bits)*

## Dirección física

Se utiliza para direccionar las celdas de memoria en los chips de memoria. Corresponden a las señales eléctricas enviadas a lo largo de los pines de dirección del microprocesador al bus de memoria. Las direcciones físicas se representan como números enteros sin signo de 32 bits o 36 bits.

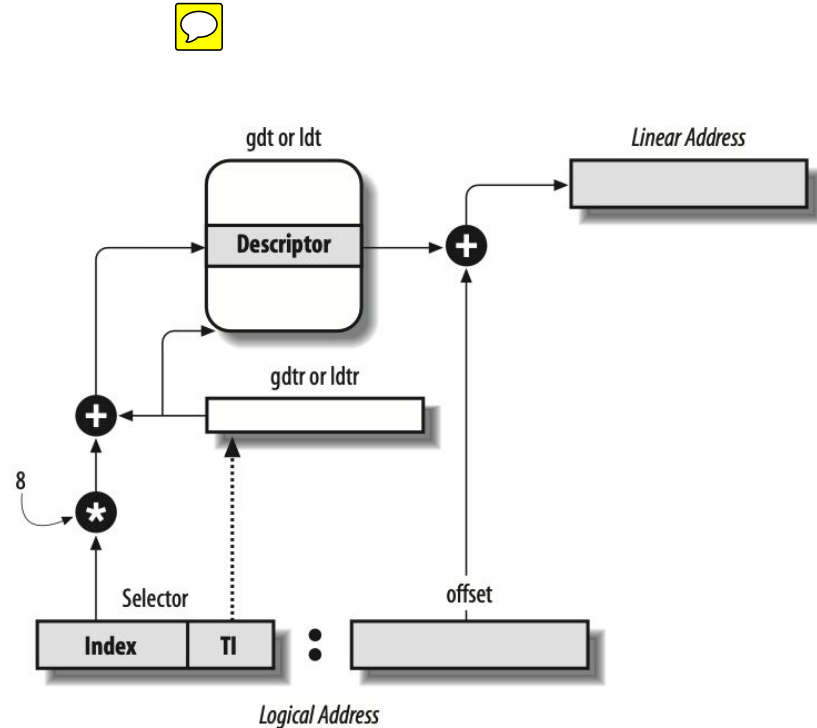
# Memoria segmentada en x86

TI (Table Indicator)  
GDT (Global Descriptor Table)  
LDT (Local Descriptor Table).

- Examina el campo TI del Selector de Segmento para determinar en qué Tabla de Descriptores se almacena el Descriptor de Segmento. Este campo indica si el Descriptor está en la GDT (en cuyo caso la unidad de segmentación obtiene la dirección base lineal de la GDT desde el registro gdttr) o en la LDT activa (en cuyo caso la unidad de segmentación obtiene la dirección base lineal de esa LDT desde el registro ldtr).

- Calcula la dirección del Descriptor de Segmento a partir del campo de índice del Selector de Segmento. El campo de índice se multiplica por 8 (el tamaño de un Descriptor de Segmento), y el resultado se suma al contenido del registro gdttr o ldtr.

- Suma el desplazamiento ("offset") de la dirección lógica al campo Base del Descriptor de Segmento, obteniendo así la dirección lineal.



# Memoria segmentada en x86



**Global Descriptor Table (GDT):** tabla guardada en memoria, apuntada por el registro llamado GDTR. Una por todo el sistema y siempre accesible. Kernel y memoria compartida.

**Local Descriptor Table (LDT):** normalmente una por tarea, programa o proceso. Puede haber varias en el sistema pero solo una está activa en un momento dado.

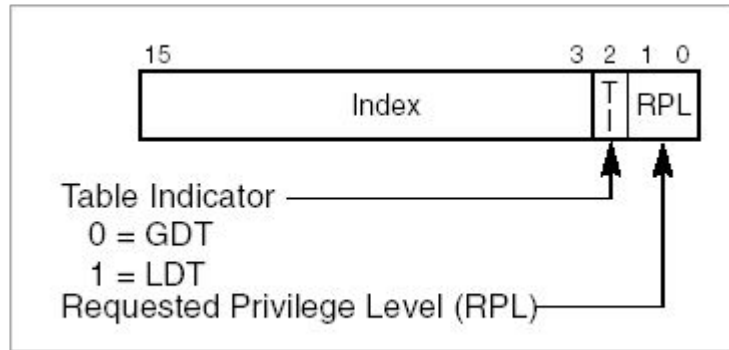
**Segment Selector:** índice en la GDT que apunta a un Segment Descriptor, cada tabla tiene 8192 entradas

**Segment Descriptor :** es una entrada de 8 bytes en la GDT:

- 32 bits segmento

- 20 bits de segmento límite dependiendo del G-bit

# Memoria segmentada en x86

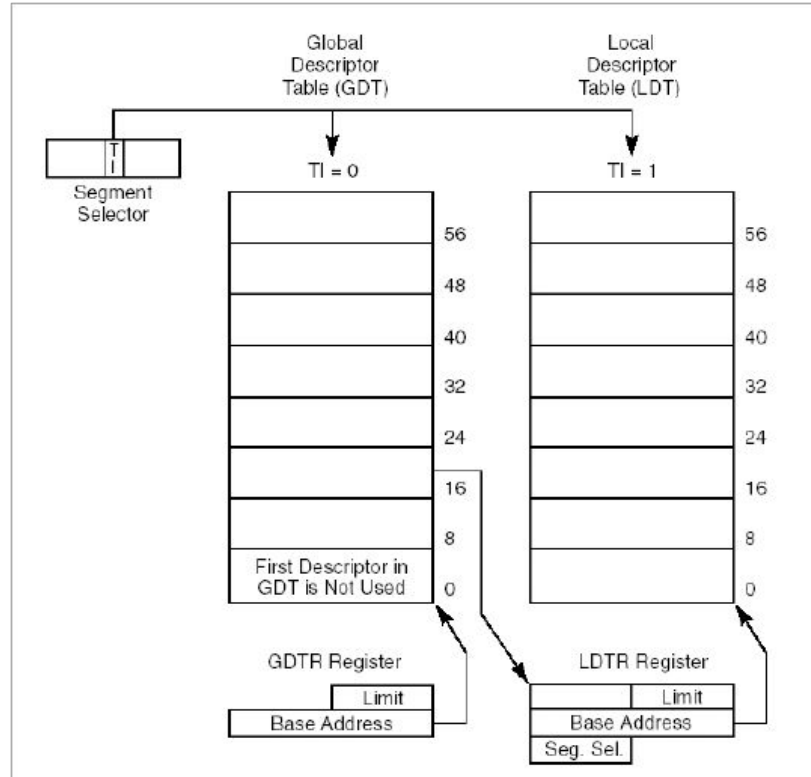


CS, DS, SS -> Selector registers

indice 13 bits

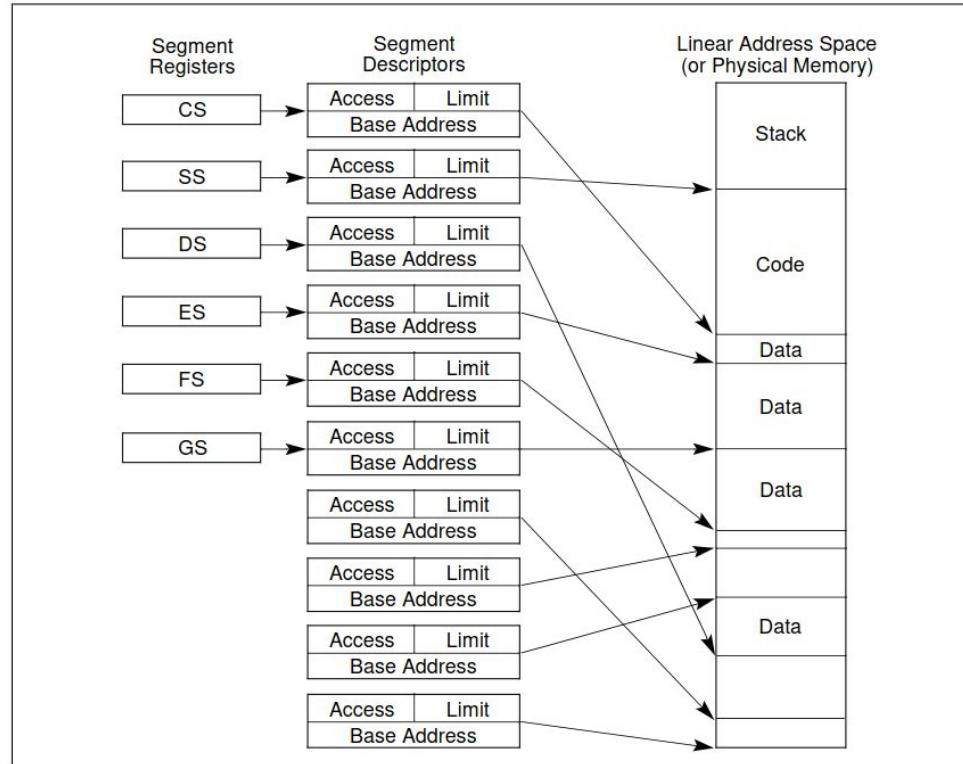
TI

RPL: requested privilege level



ST(0) MM0	ST(1) MM1	AL AH AX EAX RAX	R8B R8W R8D R8	R12B R12W R12D R12	MSW CR0	CR4			
ST(2) MM2	ST(3) MM3	BL BH BX EBX RBX	R9B R9W R9D R9	R13B R13W R13D R13	CR1	CR5			
ST(4) MM4	ST(5) MM5	CL CH CX ECX RCX	R10B R10W R10D R10	R14B R14W R14D R14	CR2	CR6			
ST(6) MM6	ST(7) MM7	DL DH DX EDX RDX	R11B R11W R11D R11	R15B R15W R15D R15	CR3	CR7			
		BPL BP EBP RBP	DIL DI EDI RDI	IP EIP RIP	MXCSR	CR8			
CW	FP_IP	FP_DP	FP_CS			CR9			
SW						CR10			
TW						CR11			
FP_DS						CR12			
FP_OPC	FP_DP	FP_IP	CS	SS	DS	DR0	DR6		
			ES	FS	GS	DR1	DR7		
						DR2	DR8		
						DR3	DR9		
						DR4	DR10	DR12	DR14
						DR5	DR11	DR13	DR15

# Memoria segmentada en x86





# Memoria segmentada en x86



A is the Accessed bit;

R is the Readable bit;

C (Bit 42) depends on X:

if X = 1 then C is the Conforming bit, and determines which privilege levels can far-jump to this segment (without changing privilege level):

if C = 0 then only code with the same privilege level as DPL may jump here;

if C = 1 then code with the same or a lower privilege level relative to DPL may jump here.

if X = 0 then C is the direction bit:

if C = 0 then the segment grows up;

if C = 1 then the segment grows down.

X is the Executable bit:[38]

if X = 1 then the segment is a code segment;

if X = 0 then the segment is a data segment.

S is the Segment type bit, which should generally be cleared for system segments:[38]

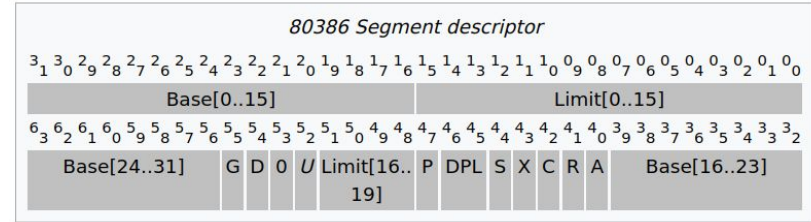
DPL is the Descriptor Privilege Level;

P is the Present bit;

D is the Default operand size;

G is the Granularity bit;

Bit 1 of the 80386 descriptor is not used by the hardware.



# Segmentación en Linux



Linux utiliza la segmentación de una manera muy limitada. De hecho, la segmentación y la paginación son algo redundantes, ya que ambas pueden utilizarse para separar los espacios de direcciones físicas de los procesos: la segmentación puede asignar un espacio de direcciones lineales diferente a cada proceso, mientras que la paginación puede mapear el mismo espacio de direcciones lineales en diferentes espacios de direcciones físicas. Linux prefiere la paginación a la segmentación por las siguientes razones:

- La gestión de memoria es más simple cuando todos los procesos usan los mismos valores de los registros de segmento, es decir, cuando comparten el mismo conjunto de direcciones lineales.
- Uno de los objetivos de diseño de Linux es la portabilidad a una amplia gama de arquitecturas; en particular, las arquitecturas RISC tienen un soporte limitado para la segmentación.

# Segmentación en Linux

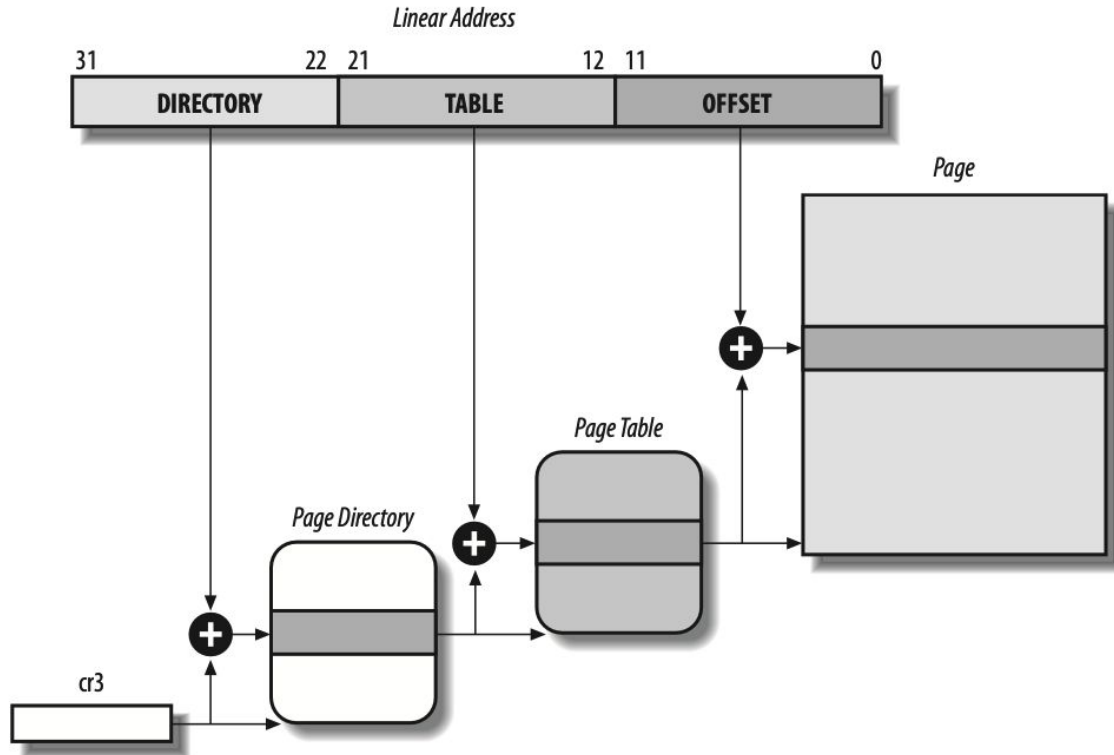
*Table 2-3. Values of the Segment Descriptor fields for the four main Linux segments*

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xffffffff	1	10	3	1	1
user data	0x00000000	1	0xffffffff	1	2	3	1	1
kernel code	0x00000000	1	0xffffffff	1	10	0	1	1
kernel data	0x00000000	1	0xffffffff	1	2	0	1	1

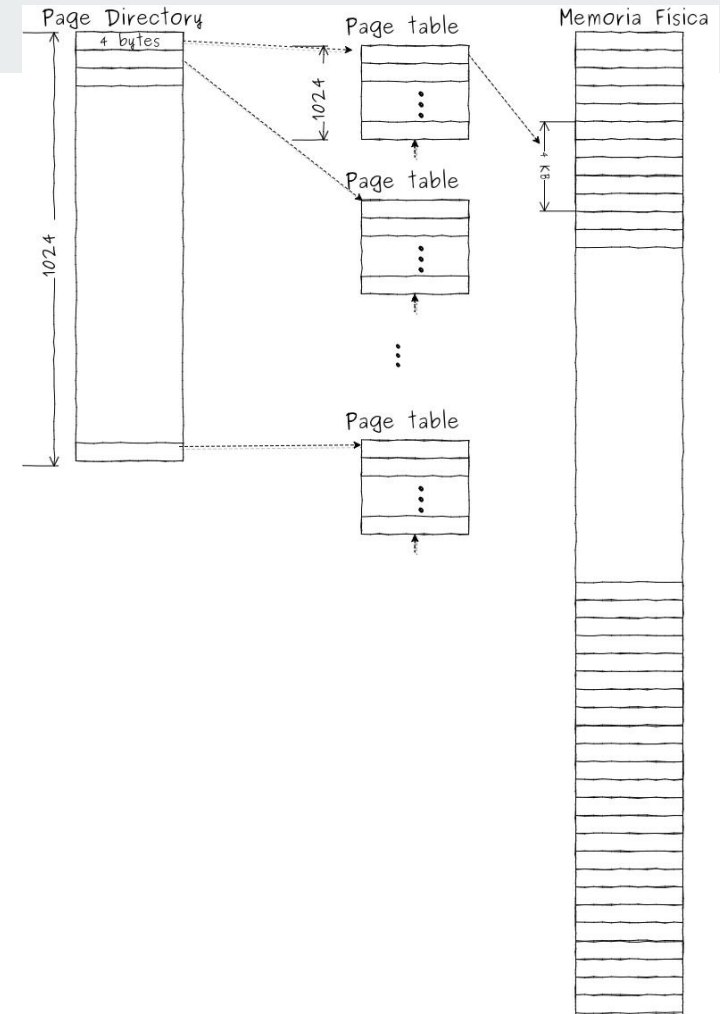
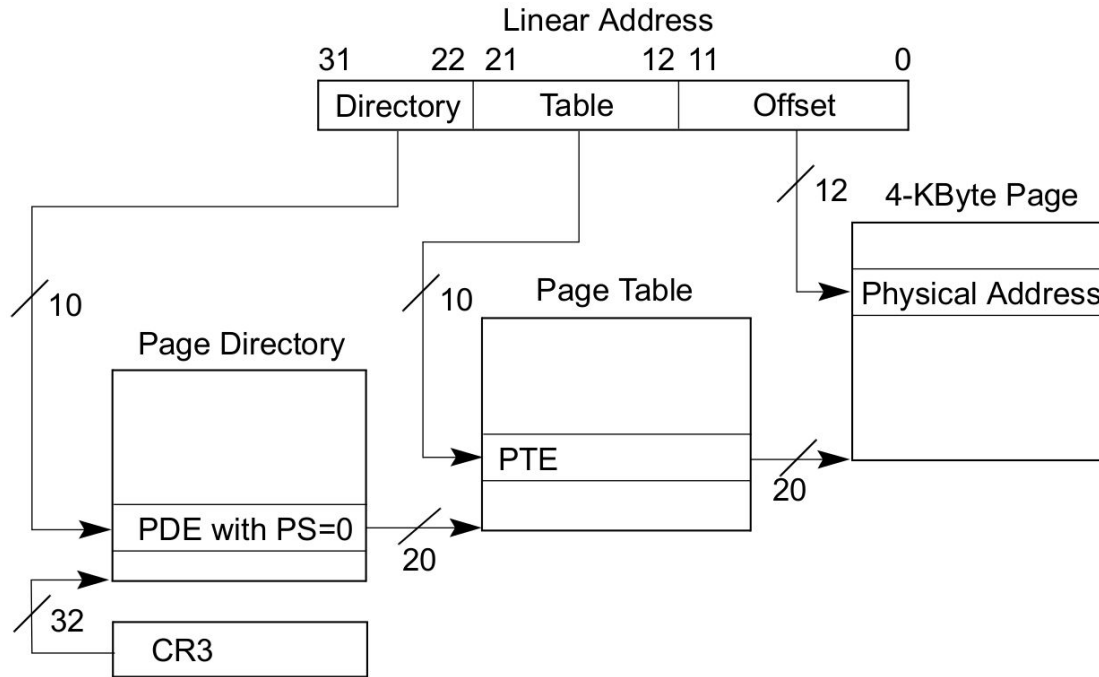
Todos los segmentos comienzan en 0x00000000 y eso hace que en Linux las direcciones lógicas coinciden con las direcciones lineales; es decir, el valor del campo de desplazamiento ("Offset") de una dirección lógica siempre coincide con el valor de la dirección lineal correspondiente.

Observa que las direcciones lineales asociadas con dichos segmentos comienzan en 0 y alcanzan el límite de direccionamiento de  $2^{32} - 1$ .

# Memoria Paginada en x86



# Memoria Paginada en x86



# Memoria Paginada en x86: Page Directory Entry

Una entrada de la page directory, ocupa 4 bytes.

La page directory posee 1024 entradas

## Page Directory Entry

31	...	12	11	...	8	7	6	5	4	3	2	1	0
Bits 31-12 of address			AVL			P S (0)	A V L	A	P C D	P W T	U / S	R / W	P

<b>P:</b> Present	<b>D:</b> Dirty
<b>R/W:</b> Read/Write	<b>PS:</b> Page Size
<b>U/S:</b> User/Supervisor	<b>G:</b> Global
<b>PWT:</b> Write-Through	<b>AVL:</b> Available
<b>PCD:</b> Cache Disable	<b>PAT:</b> Page Attribute Table
<b>A:</b> Accessed	

# Memoria Paginada en x86: Page Directory Entry



**Present (P):** Este bit indica si la entrada de la tabla de páginas está presente en la memoria física. Si está establecido (1), la entrada está presente y se puede utilizar para traducir direcciones virtuales. Si está desactivado (0), la entrada no está presente y cualquier intento de acceso a ella causará un fallo de página.

**Read/Write (R/W):** Este bit controla si la región de memoria mapeada por esta entrada es de solo lectura (0) o lectura/escritura (1).

**User/Supervisor (U/S):** Este bit determina los privilegios de acceso a la región de memoria mapeada. Si está configurado, la región es accesible en modo usuario y modo supervisor. Si está desactivado, solo es accesible en modo supervisor.

**Write-through (W/T):** Este bit controla la política de escritura en caché para la región de memoria. Si está establecido, las escrituras se realizan a través de la caché. Si está desactivado, las escrituras se pueden realizar directamente en la memoria principal.

**Cache Disable (C):** Este bit se utiliza para desactivar la caché para la región de memoria mapeada. Si está establecido, la caché está desactivada para la región. Si está desactivado, la caché puede utilizarse.

**Accessed (A):** Este bit se establece por hardware cada vez que se accede a la región de memoria mapeada. Es útil para la gestión de la memoria y la optimización de algoritmos de reemplazo de páginas.

**Dirty (D):** Este bit se establece por hardware cada vez que se escribe en la región de memoria mapeada. Indica que la página ha sido modificada desde la última vez que se limpió.

**Large Page (PS):** Si este bit está configurado, indica que la entrada PDE apunta a una tabla de páginas de tamaño grande (4 MB en lugar de 4 KB). Esto se utiliza para el soporte de páginas grandes y puede mejorar el rendimiento en ciertos casos.

**Global (G):** Este bit se utiliza para páginas globales. Si está establecido, la página no se elimina del caché de traducción de direcciones (TLB) cuando se cambia el contexto del proceso.

**Available (Avail):** Estos bits están disponibles para el uso del software y pueden ser utilizados para almacenar información adicional específica del sistema operativo o de la aplicación.

# Memoria Paginada en x86 : Page Table Entry

Una entrada de la page table, ocupa 4 bytes.

La page directory posee 1024 entradas

## Page Table Entry

31	...	12	11... 9	8	7	6	5	4	3	2	1	0
Bits 31-12 of address			AVL	G	P A T	D	A	P C D	P W T	U / S	R / W	P

<b>P:</b> Present	<b>D:</b> Dirty
<b>R/W:</b> Read/Write	<b>G:</b> Global
<b>U/S:</b> User/Supervisor	<b>AVL:</b> Available
<b>PWT:</b> Write-Through	<b>PAT:</b> Page Attribute Table
<b>PCD:</b> Cache Disable	
<b>A:</b> Accessed	



# Memoria Paginada en x86 : Page Table Entry



**Present (P):** Este bit indica si la página de memoria física asociada con esta entrada de la tabla de páginas está presente en la memoria física. Si está establecido (1), la página está presente y se puede acceder. Si está desactivado (0), cualquier intento de acceso a la página causará un fallo de página.

**Read/Write (R/W):** Este bit controla si la página de memoria mapeada por esta entrada es de solo lectura (0) o lectura/escritura (1).

**User/Supervisor (U/S):** Este bit determina los privilegios de acceso a la página de memoria mapeada. Si está configurado, la página es accesible en modo usuario y modo supervisor. Si está desactivado, solo es accesible en modo supervisor.

**Write-through (W/T):** Este bit controla la política de escritura en caché para la página de memoria mapeada. Si está establecido, las escrituras se realizan a través de la caché. Si está desactivado, las escrituras se pueden realizar directamente en la memoria principal.

**Cache Disable (C):** Este bit se utiliza para desactivar la caché para la página de memoria mapeada. Si está establecido, la caché está desactivada para la página. Si está desactivado, la caché puede utilizarse.

**Accessed (A):** Este bit se establece por hardware cada vez que se accede a la página de memoria mapeada. Es útil para la gestión de la memoria y la optimización de algoritmos de reemplazo de páginas.

**Dirty (D):** Este bit se establece por hardware cada vez que se escribe en la página de memoria mapeada. Indica que la página ha sido modificada desde la última vez que se limpió.

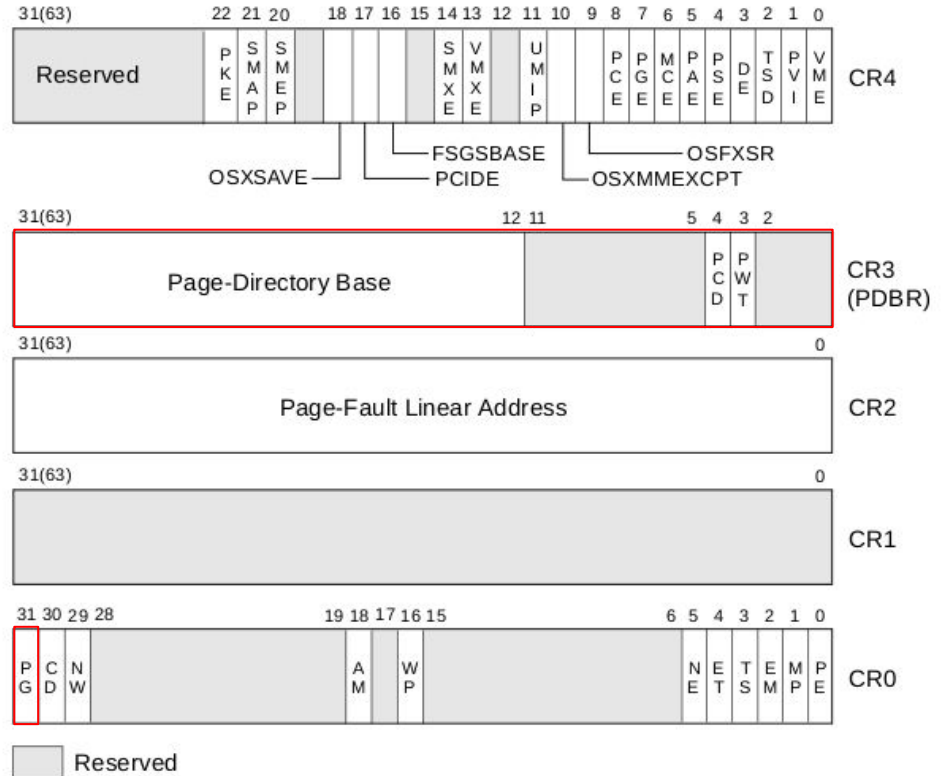
**Global (G):** Este bit se utiliza para páginas globales. Si está establecido, la página no se elimina del caché de traducción de direcciones (TLB) cuando se cambia el contexto del proceso.

# Memoria Paginada en x86 : Registros especiales

Los registros importantes para la paginación son: CR0 y CR3.

El bit más a la izquierda de el registro **CR0** si está en 0 determina que la lineal address se convierte directamente en physical address para acceder a la memoria. Si PG está en 1 la lineal address debe ser convertida en physical address a través del mecanismo de paginación.

**CR3** contiene page directory base que contiene 1024 entradas de 4 bytes cada una, cada entrada en el page directory ocupa 4 bytes y direcciona a una page table que contiene 1024 entradas.



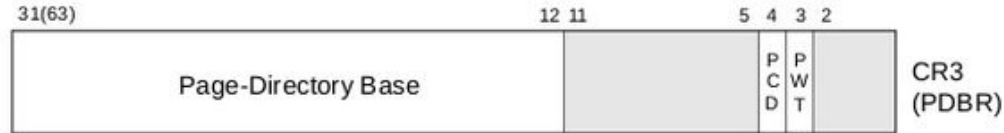
# Memoria Paginada: CR3



El registro CR3 es uno de los registros de control y es fundamental para la administración de la memoria virtual.

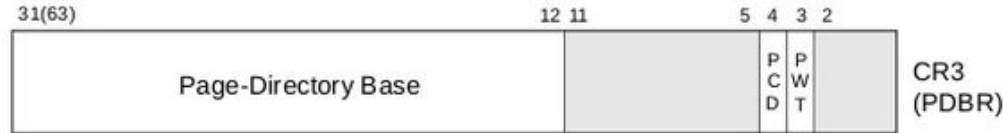
- **Función Principal:** El registro CR3 se utiliza para apuntar a la tabla de directorios de páginas en la memoria física. La tabla de directorios de páginas es la estructura de datos de nivel superior en el mecanismo de paginación de x86, que a su vez apunta a tablas de páginas individuales.
- **Papel en la Paginación:** Cuando la paginación está habilitada (es decir, el bit PG en CR0 está establecido), cada vez que el procesador necesita traducir una dirección virtual a una dirección física, consulta la estructura de paginación que comienza en la dirección física especificada en CR3.
- **Cambios en CR3:** Al cambiar el valor de CR3 (por ejemplo, al cambiar de contexto o al cambiar a un proceso diferente), efectivamente se está cambiando el espacio de direcciones virtual activo, ya que se está apuntando a una tabla de directorios de páginas diferente.

# Memoria Paginada en x86 : CR3



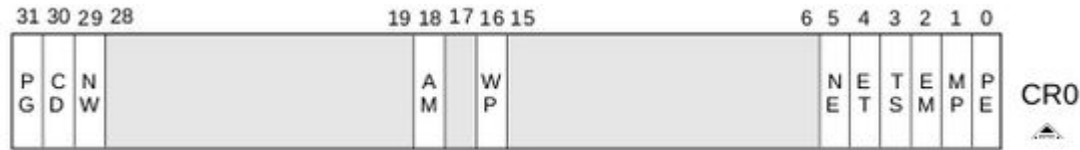
- **Bits de Dirección de la Base de la Tabla de Directorios de Páginas (PDBR):** Estos bits contienen la dirección base de la tabla de directorios de páginas. En sistemas x86 sin PAE, los bits 31-12 de CR3 apuntan a la base de la tabla de directorios de páginas y los bits 11-0 son ignorados (y generalmente se establecen en 0),
- **PCD (Page-Level Cache Disable, Bit 4):** Si se establece, la paginación no se cachea. Sin embargo, este bit es a menudo ignorado, con las características de cacheo determinadas por las entradas individuales de la tabla de páginas.
- **PWT (Page-Level Write-Through, Bit 3):** Controla las características de caching para las tablas de páginas. Si se establece, se utiliza la política de cacheo de **Write-Through**; si se borra, se utiliza **Write-Back**. Son políticas de caché utilizadas en sistemas informáticos, especialmente en relación con la memoria caché de la CPU y algunas veces en sistemas de almacenamiento.

# Memoria Paginada: CR3



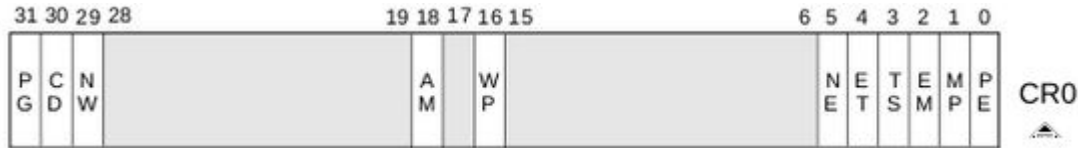
- **Invalidación de TLB:** Cada vez que se escribe en CR3, la caché de la tabla de búsqueda (TLB) se invalida automáticamente. Esto se debe a que la TLB podría contener entradas antiguas basadas en la antigua estructura de paginación, y al cambiar CR3, estas entradas ya no serían válidas.
- **Formato:** La dirección almacenada en CR3 debe estar alineada a una página, lo que significa que los bits inferiores de CR3 (que especificarían un desplazamiento dentro de una página) son cero y no se utilizan en la dirección. Estos bits inferiores, sin embargo, tienen otros usos en versiones más recientes de la arquitectura x86.

# Memoria Paginada en x86 : CR0



**Función Principal:** El registro CR0 alberga varios flags que controlan cómo opera el procesador en varios aspectos. Es especialmente importante para habilitar o deshabilitar la paginación y el modo protegido.

# Memoria Paginada en x86 : CR0



Bits Principales:

**PE (Bit 0, Modo Protegido):** Cuando este bit está establecido, el procesador opera en modo protegido. De lo contrario, opera en modo real.

**WP (Bit 16, Protección de Escritura):** Cuando está establecido, determina el comportamiento de las páginas de solo lectura en modo supervisor.

**PG (Bit 31, Paginación):** Cuando este bit está establecido, la paginación está habilitada. Si está desactivado, el procesador usa una traducción de dirección lineal a física directa.





# Ejemplo de traducción



Considere un sistema x86 de memoria virtual paginada de dos niveles con un espacio de direcciones de 32 bits, donde cada página tiene un tamaño de 4096 bytes.

Como se realiza la traducción de 0x01FBD000

Dirección virtual:

**0x01FBD000 = 00000001111110111101000000000000**

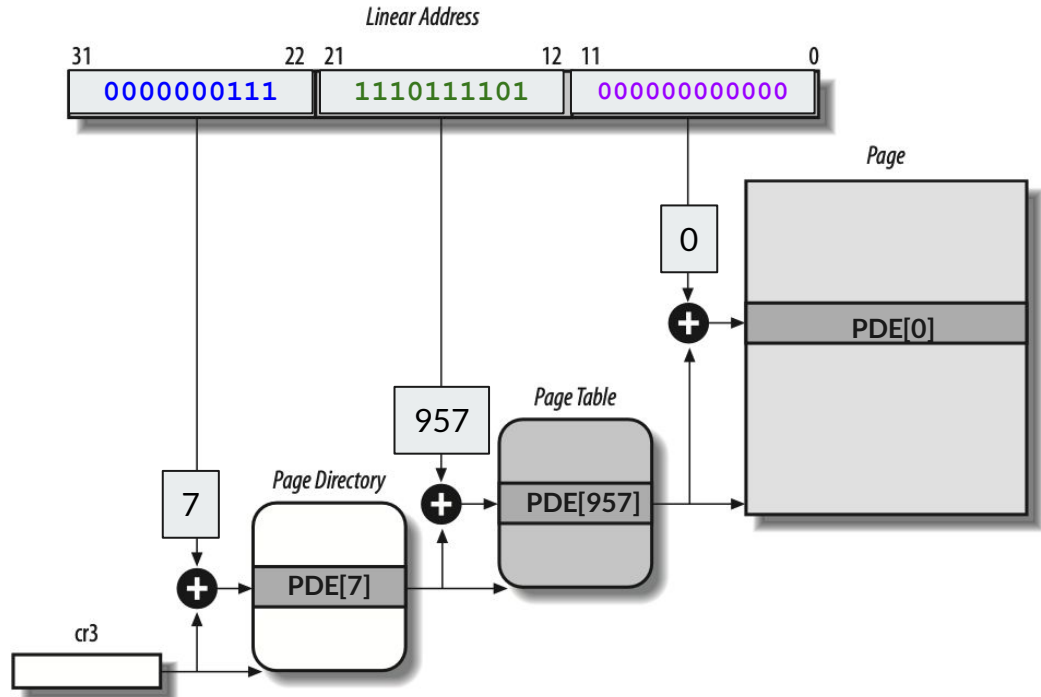
# Ejemplo de traducción

0x01FBD000 = 000000011111101111010000000000

$$0000000111_2 = 7_{10}$$

$$1110111101_2 = 957_{10}$$

$$000000000000_2 = 0_{10}$$



# Ejemplo de parcial



Considere un sistema x86 de memoria virtual paginada de dos niveles con un espacio de direcciones de 32 bits, donde cada página tiene un tamaño de 4096 bytes.

Un entero ocupa 4 bytes y se tiene un array de 50.000 enteros que comienza en la dirección virtual 0x01FBD000

El arreglo se recorre completamente, accediendo a cada elemento una vez. En este proceso, ¿a cuántos frames de memoria física distintos (no la cantidad total de accesos) necesita acceder el sistema operativo para conseguir esto?

*Recuerde contar las tablas de páginas intermedias, no solo las páginas que contienen los elementos del array.*

# Ejemplo de parcial



Cantidad total de bytes que ocupa el array:  $50,000 \times 4 = 200,000$

Posición del primer byte: 0x01FBD000 (definido en el enunciado)

Posición del último byte:  $0x01FBD000 + (50000 \times 4 - 1) = 0x01FEDD3F$

El rango de direcciones que el sistema operativo va a recorrer es entonces:

**[0x01FBD000, 0x01FEDD3F]**

# Ejemplo de parcial



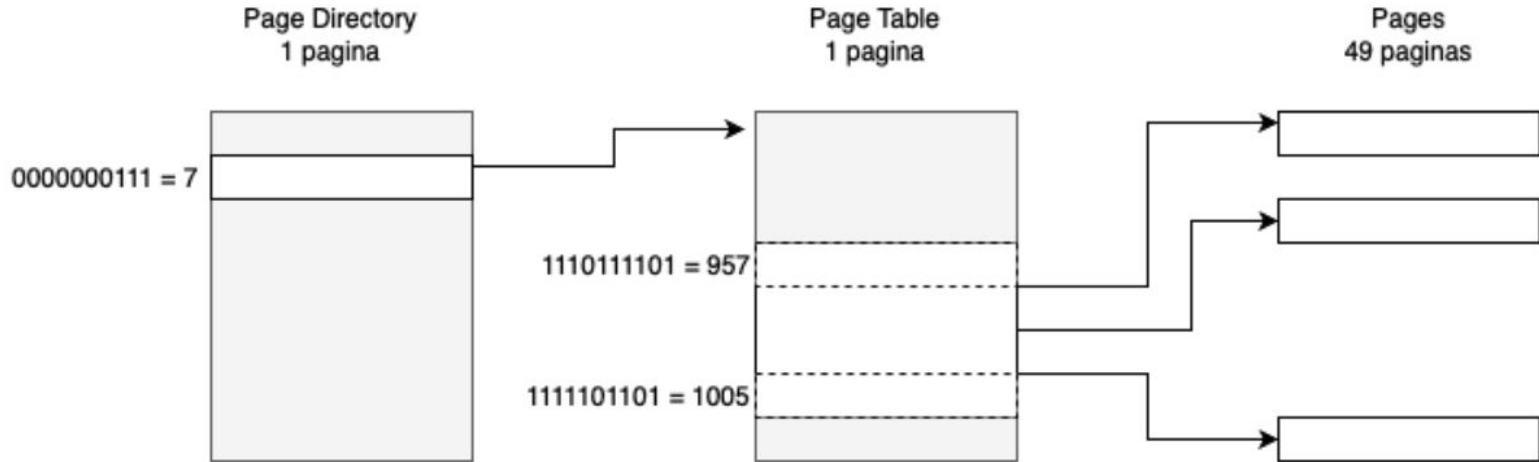
- `0x01FBD000 = 0000000111 1110111101 000000000000`
- `0x01FEDD3F = 0000000111 1111101101 110100111111`

Expresados en decimal para mayor claridad

- `0x01FBD000 = [7] [957] [0]`
- `0x01FEDD3F = [7] [1005] [3391]`

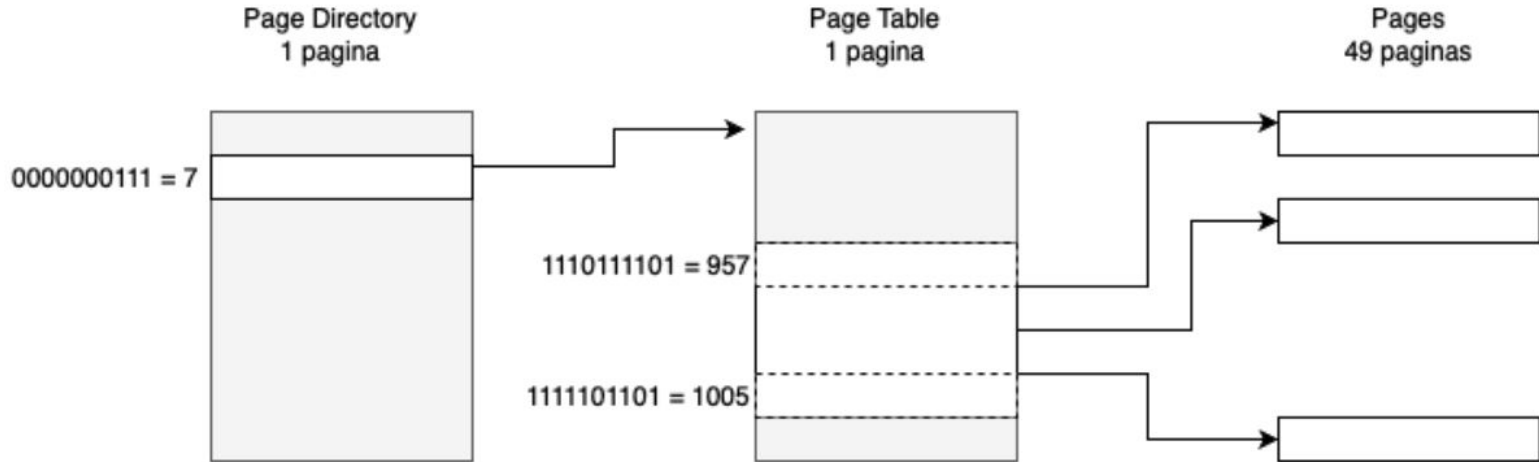
En el primer nivel (el directorio de páginas), se usa un solo registro, en el índice 7. Este contiene la dirección (y metadata) de la **única tabla de páginas** que será necesaria en el segundo nivel. Si se requirieran más tablas de páginas de segundo nivel, este índice cambiaría para algunas direcciones involucradas en el rango. Esto no ocurre para el rango propuesto en este ejercicio.

# Ejemplo de parcial



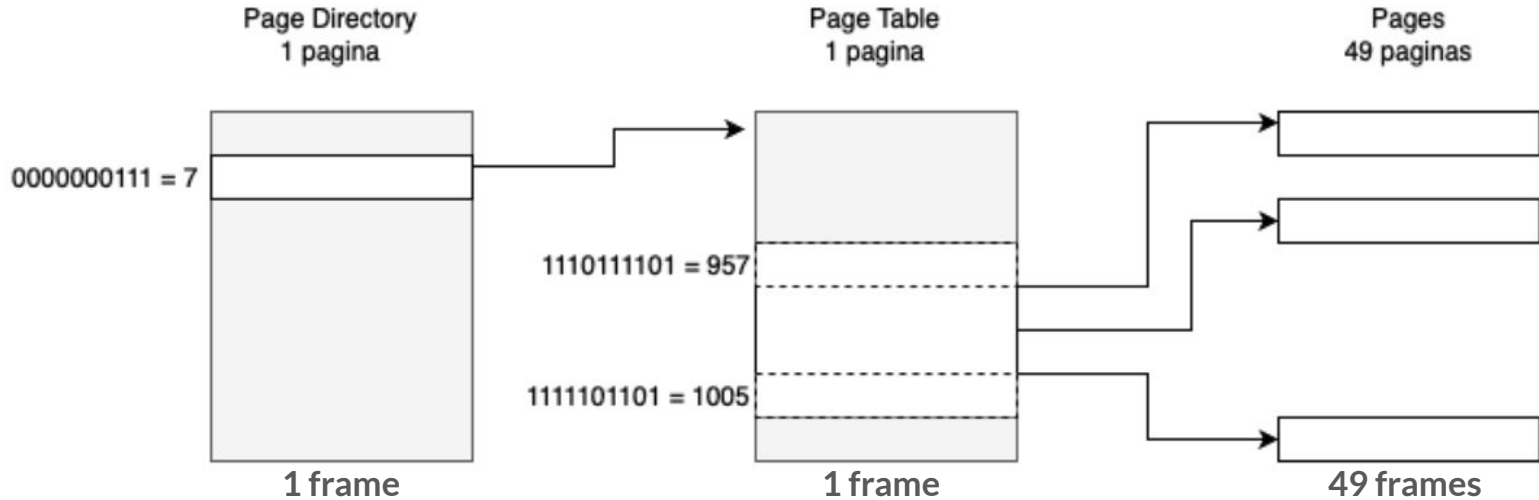
En el segundo nivel, se utilizan varios registros, cada uno contiene la dirección de las páginas donde están los datos del array. Concretamente, se usan los registros desde el 957 hasta el 1005, es decir, 49 registros, lo que implica 49 páginas de datos.

# Ejemplo de parcial



En el tercer nivel, cada página contiene 4096 bytes de datos. Si queremos guardar 200,000 bytes entonces necesitamos 48.8 páginas, es decir, necesitamos utilizar 49 páginas. Esto es consistente con el cálculo que hicimos antes a partir de las direcciones de los extremos del rango de bytes del array.

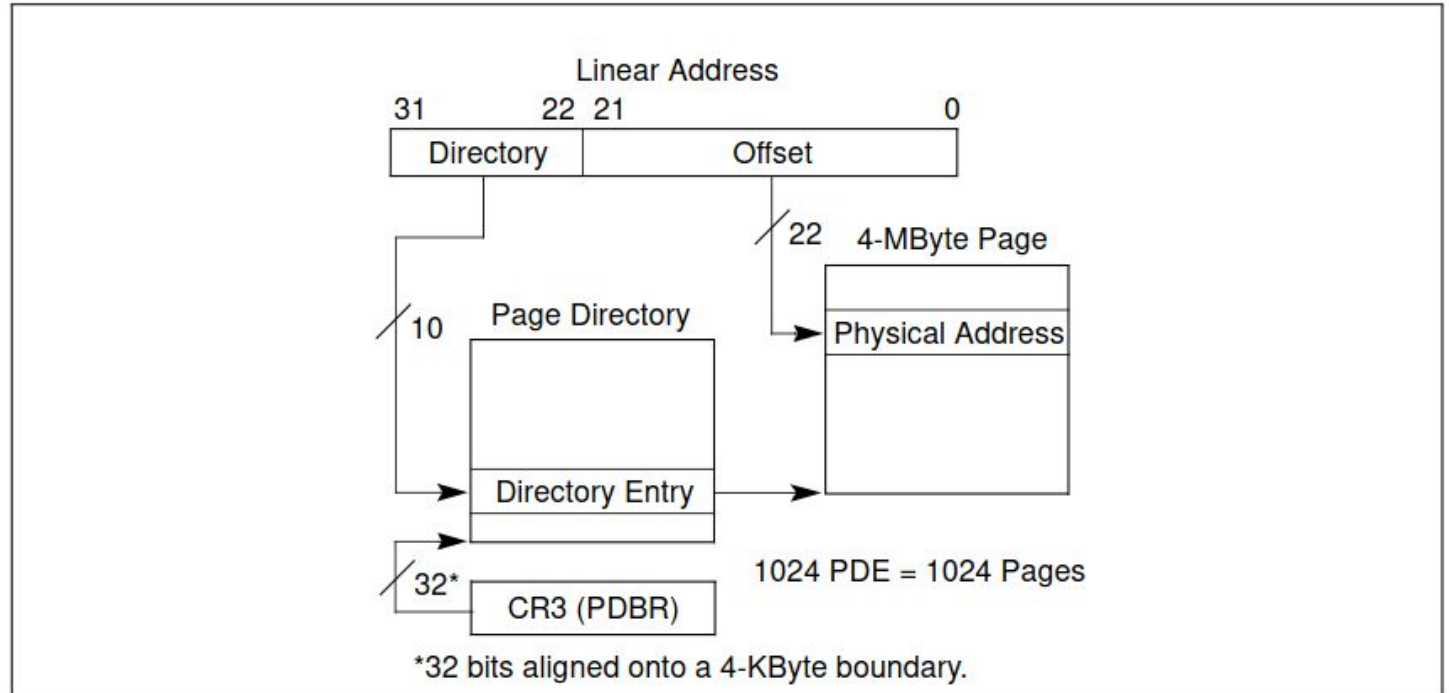
# Ejemplo de parcial



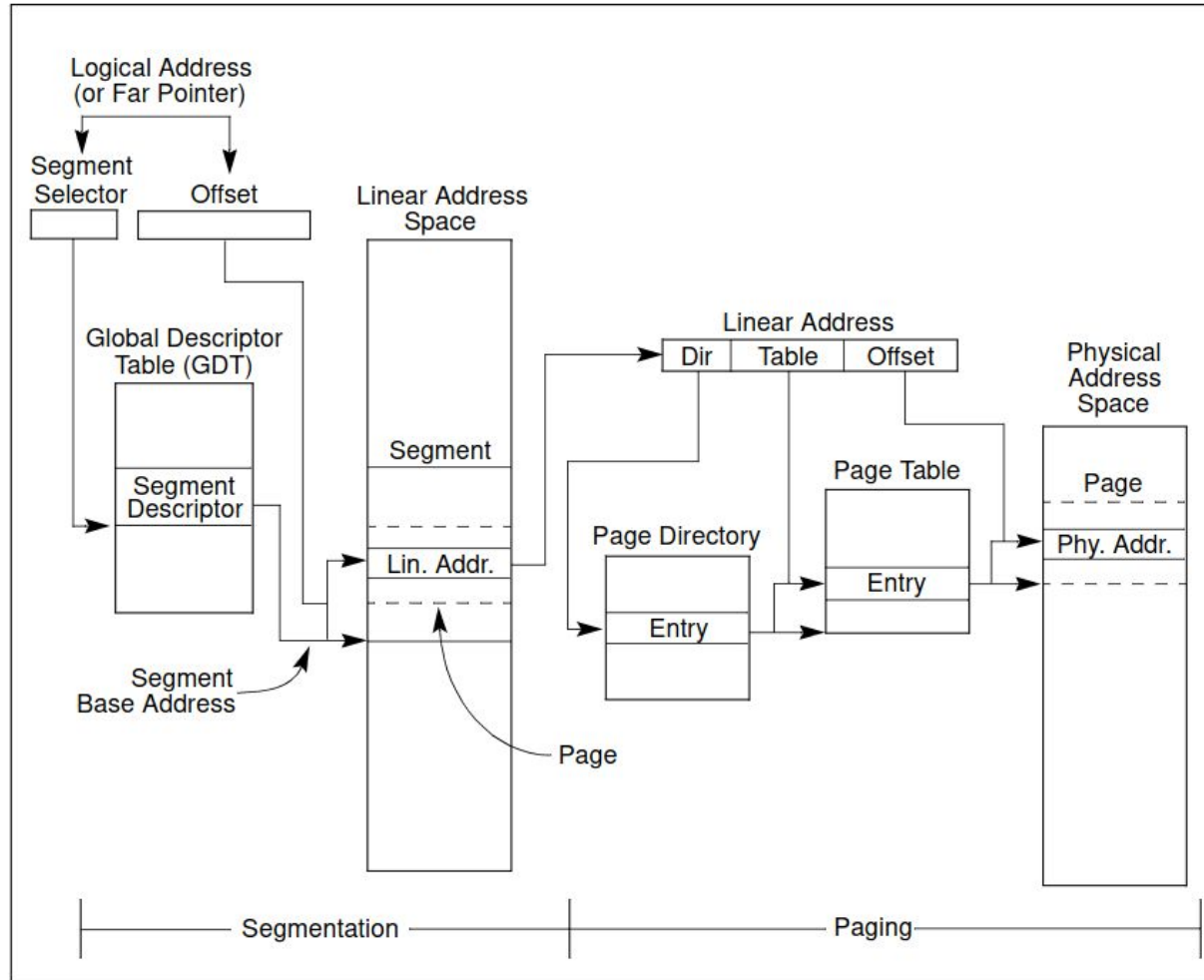
**Resultado:** 1 page directory (por definición de la arquitectura x86) + 1 page table (justificado anteriormente) + 49 páginas de datos (justificado anteriormente) = **51 páginas en total** que el sistema operativo necesita utilizar para guardar y acceder a este array.



# Memoria Paginada extendida



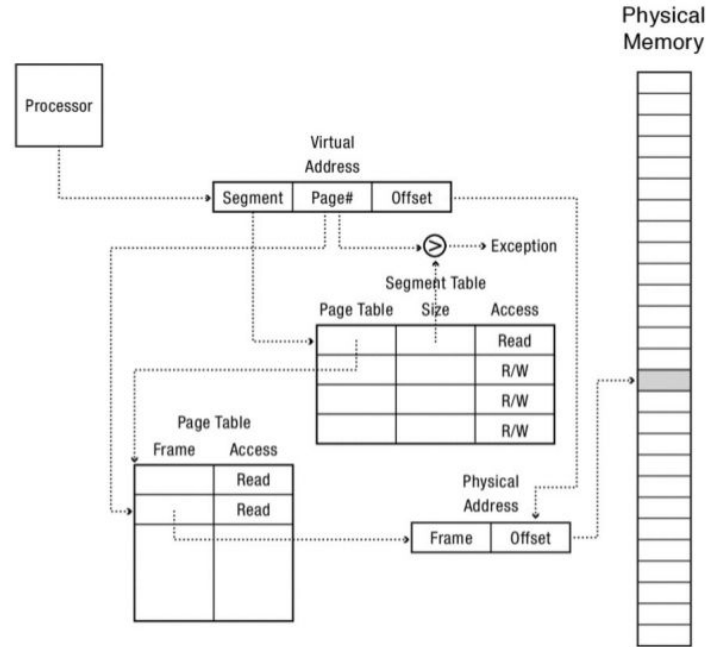
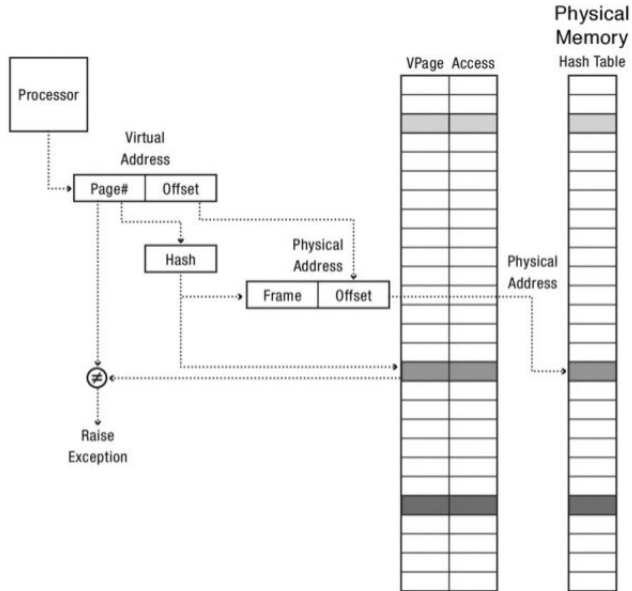
# x86



Listo se termino?



# Listo se termino?



Leer Dahlin!!!

---

# Translation Lookaside Buffer

## Conceptos clave:

- Caches
- TLB

# Hacia una Eficiente Address Translation



Hasta aquí ya uno puede estar un poco cansado de todos los mecanismos de hardware para realizar la traducción de direcciones de memoria.

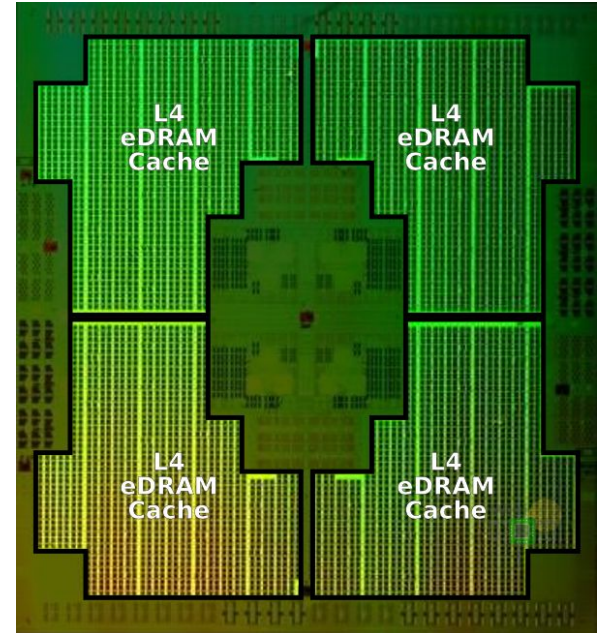
El tema es que muchos de estos métodos tienen varios niveles, hasta 4 en algunos casos, para alcanzar una dirección física, entonces eso lo hace realmente poco práctico para el procesador.

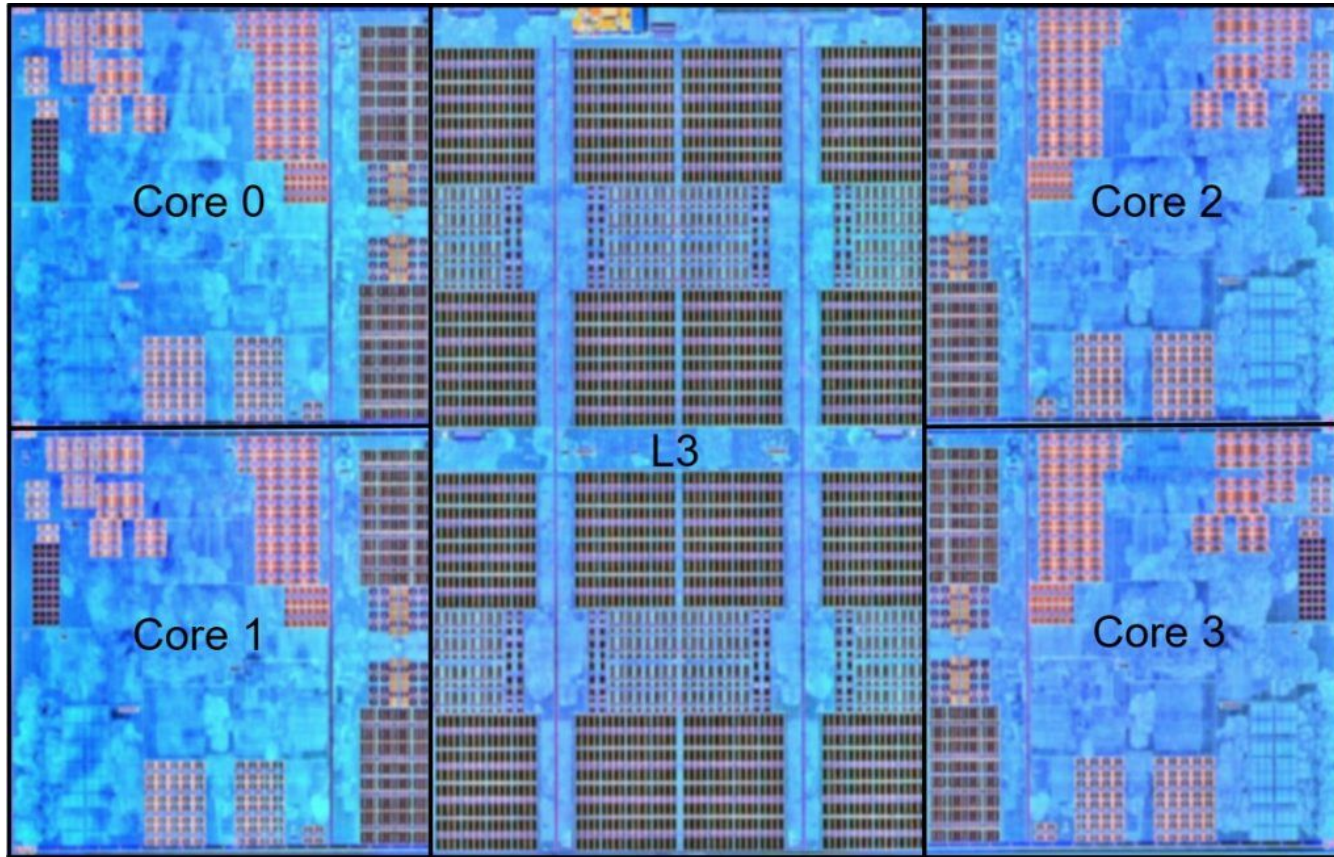
A continuación se mostrarán mecanismos para mejorar el rendimiento de la traducción de las direcciones.

# Hacia una Eficiente Address Translation

Este nuevo mecanismo usará un caché (o escondrijo), que consiste en una copia de ciertos datos que pueden ser accedidos más de una vez más rápidamente.

El concepto de Cache es ampliamente utilizado en muchas ramas de las ciencias de la computación: arquitectura de computadoras, sistemas operativos, sistemas distribuidos.





**Hacia una Eficiente Address Translation**



# Hacia una Eficiente Address Translation: TLB




Uno de los problemas del **address translation** reside en la velocidad de la traducción para ello se utilizan técnicas que mejoran la velocidad de esta traducción.

Para mejorar el address translation se utiliza un mecanismo de hardware llamado **Translation-Lookaside Buffer**; o también conocido como TLB.

La TLB es parte de la MMU y es simplemente un mecanismo de cache de las traducciones más utilizadas entre los pares virtual to physical address. Por ende un mejor nombre para este mecanismo podría ser address translation cache.

Por cada referencia a la memoria virtual, el hardware primero chequea la TLB para ver si esa traducción esta guardada ahí; si es así la traducción se hace rápidamente sin tener que consultar a la page table (la cual tiene todas las traducciones).

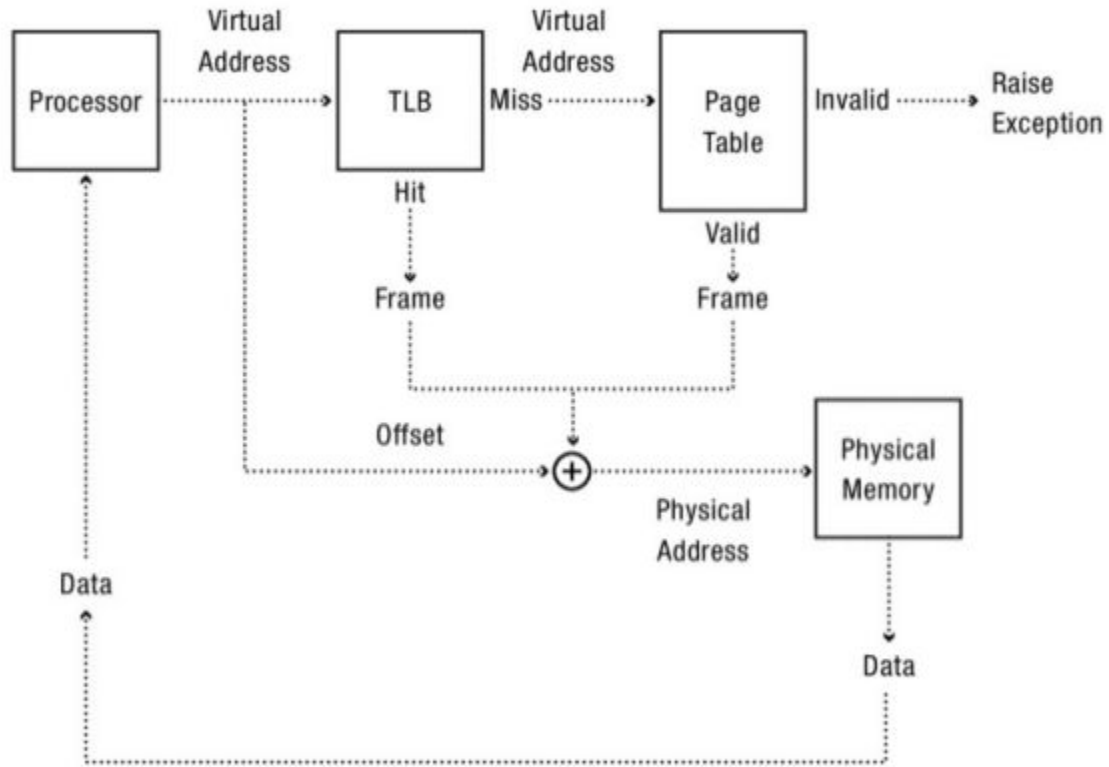
# Translation-Lookaside Buffer



```
...  
mov r1,r2  
mult r1,r2  
...
```

```
TLB entry = {  
    virtual page number,  
    physical page frame number,  
    access permissions  
}
```

- Secuencia y localización
- El hardware tiene que hacer el **fetch** de la instrucción add, después pasear por todos los multiniveles de las tablas de traducción para encontrar la dirección física de la instrucción add; ejecutar la instrucción , incrementar el contador de programa y volver a hacer todo esto otra vez para la próxima instrucción y además para sus datos pero esto es muy ineficiente.
- La Translation Lookaside Buffer (TLB) es una pequeña tabla a nivel hardware que contiene los resultados de la recientes traducciones de memorias realizadas. Cada entrada de la tabla mapea una virtual page a una physical page



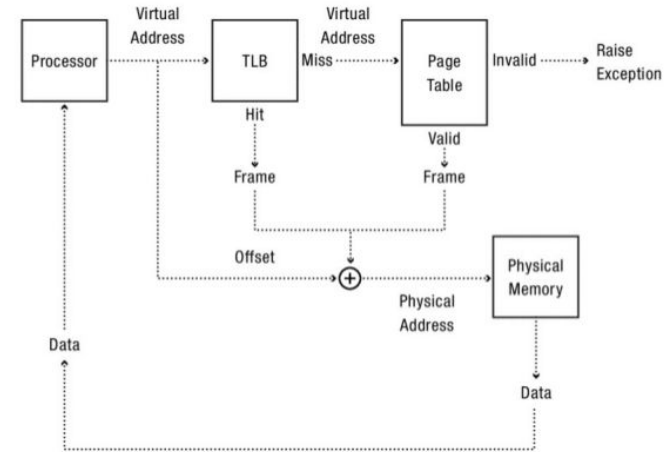
## Translation-Lookaside Buffer

# Translation-Lookaside Buffer

Normalmente se chequean todas las entradas de la TLB contra la virtual page, si existe matcheo el procesador utiliza ese matcheo para formar la physical address, ahorrándose todos los pasos de la traducción.

Esto se llama un TLB hit

Cuando del proceso anterior no existe matcheo en la TLB, se dice que se tiene un TLB miss



# Translation-Lookaside Buffer



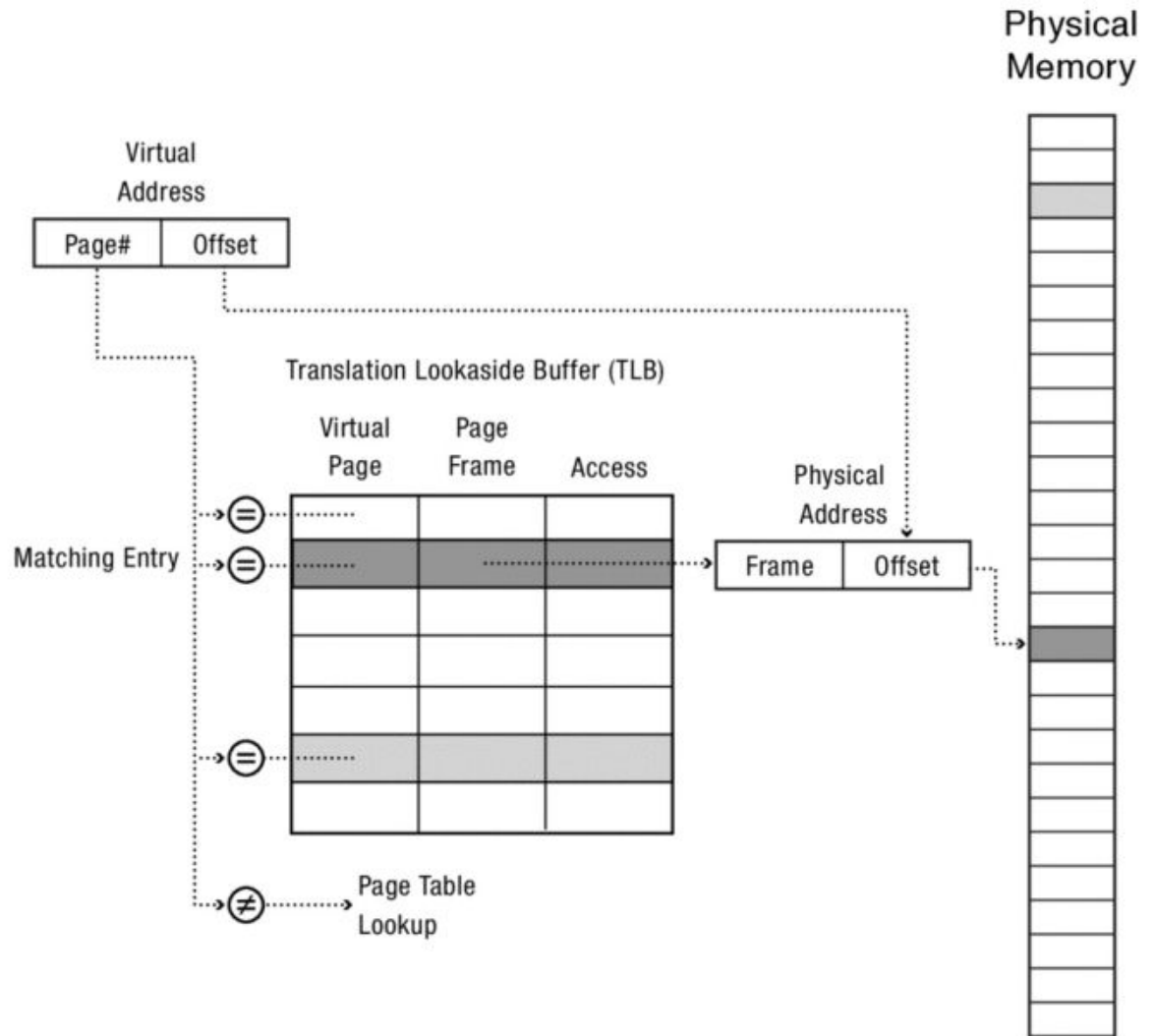
Para que sea útil, la búsqueda de la TLB necesita ser mucho más rápido que realizar una traducción completa de una dirección de memoria.

Por ende, las entradas de la tabla de la TLB son implementadas en una memoria muy rápida, memoria estática on-chip, situada muy cerca del procesador. De hecho, para mantener esta búsqueda rápida, muchos sistemas en la actualidad incluyen múltiples niveles de TLB. En general, cuanto más pequeña es la memoria, más rápida es la búsqueda.

Si el primer nivel de la TLB produce un TLB miss existe otro nivel, que guarda más datos, y este es consultado y la traducción se realiza si se falla en ambos niveles

# La TLB

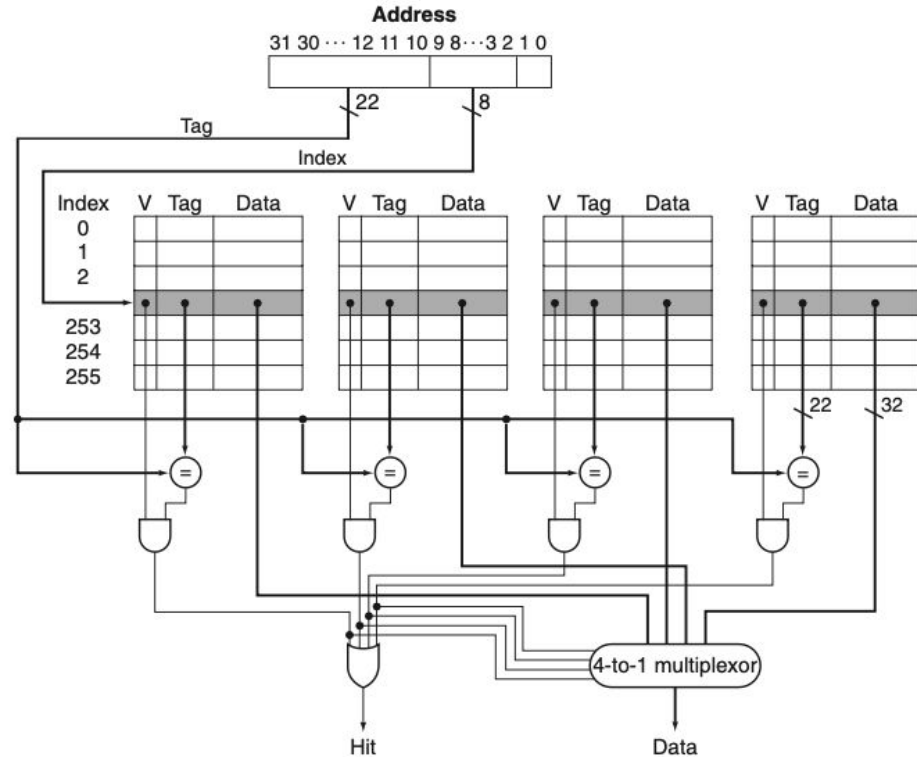
El page number se busca en paralelo en todas las entradas del cache. Este esquema es un full-associative cache.



# La TLB

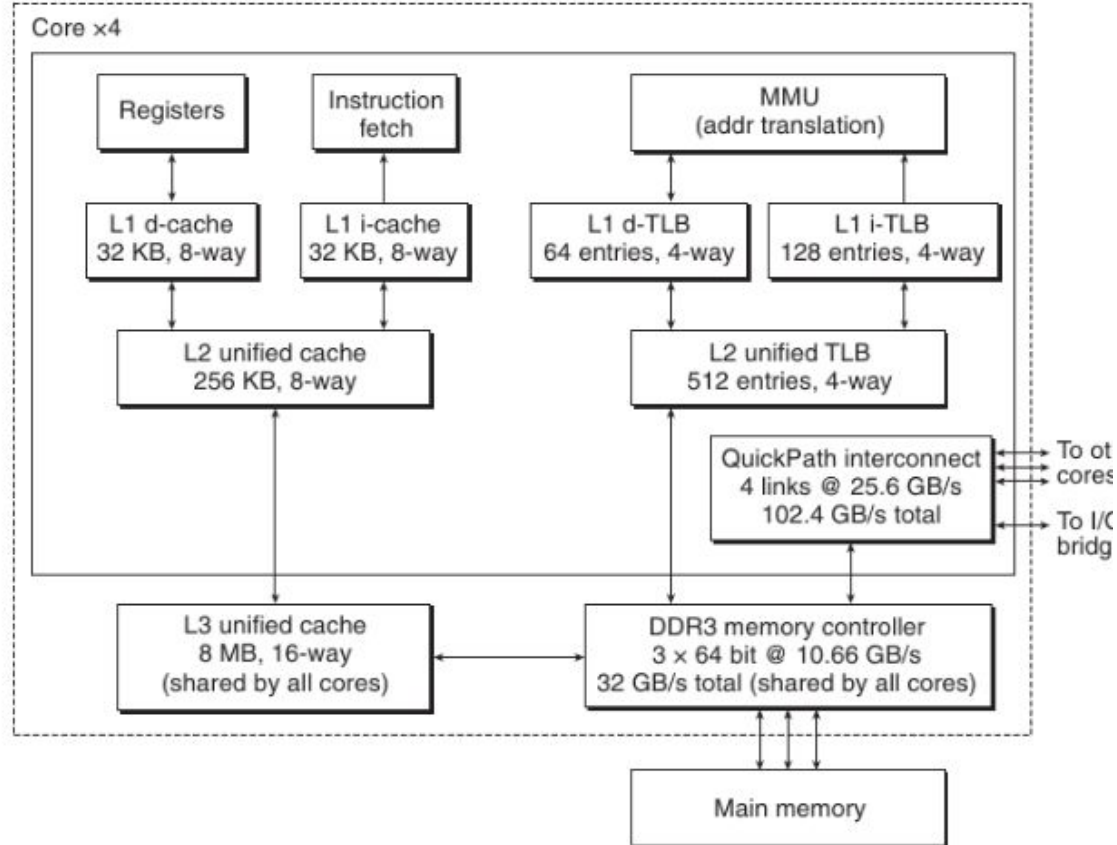
En un set-associative cache cada page number se guarda en una posición específica y definida por los bits de la dirección para optimizar el acceso.

Para reducir el efecto de colisiones, se replican los bancos de cache y se buscan en paralelo con cada acceso.



**FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor.** The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

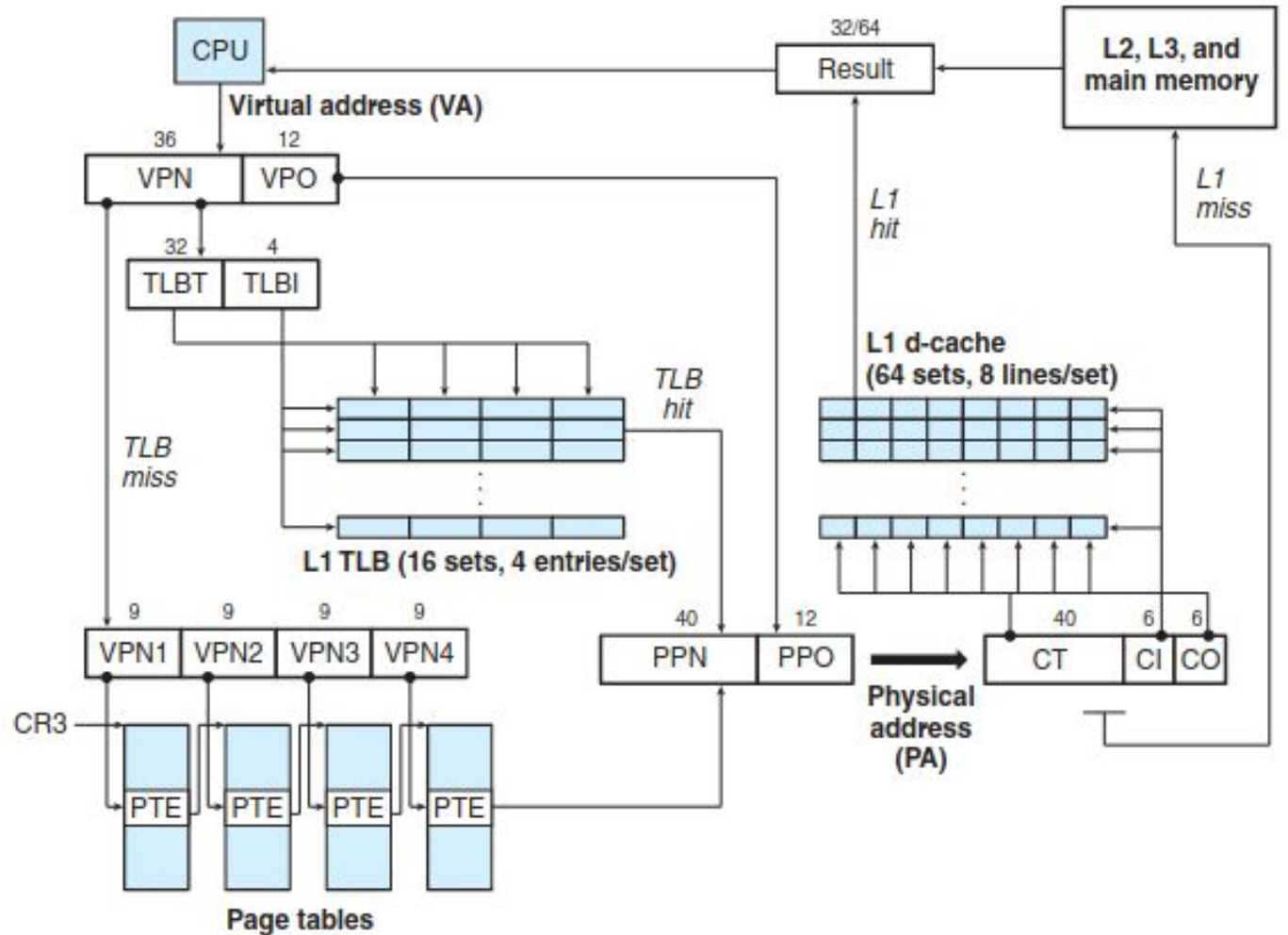
## Processor package



# Hacia una Eficiente Address Translation



Todo  
completo



# Consistencia de la TLB



Cada vez que se introduce un cache en el sistema, se necesita considerar la forma de asegurar la consistencia del cache con los datos originales cuando las entradas en el mismo son modificadas. Una TLB no es la excepción.

Para una ejecución correcta y segura de un programa, el sistema operativo tiene que asegurarse que cada programa ve su propia memoria y la de nadie más.

# Consistencia de la TLB



**Context switch:** Las direcciones virtuales del viejo proceso ya no son más válidas, y no deben ser válidas, para el nuevo proceso.

De otra forma, el nuevo proceso sería capaz de leer las direcciones del viejo proceso. Frente a un context switch, se necesita descartar el contenido de TLB en cada context switch.

Este approach se denomina flush de TLB. Debido a que este proceso acarrearía una penalidad, los procesadores taguean la TLB de forma tal que la misma contenga el id del proceso que produce cada transacción.

Ver: <https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/trampoline.S#L91-L95>

# Consistencia de la TLB



**Reducción de Permiso:** Qué sucede cuando el sistema operativo modifica una entrada en una page table? Normalmente no se provee consistencia por hardware para la TLB; mantener la TLB consistente con la page table es responsabilidad del sistema operativo.

# Consistencia de la TLB



**TLB shutdown:** En un sistema multiprocesador cada uno puede tener cacheada una copia de una transacción en su TLB. Por ende, para seguridad y correctitud, cada vez que una entrada en la page table es modificada, la correspondiente entrada en todas las TLB de los procesadores tiene que ser descartada antes que los cambios tomen efecto.

Típicamente sólo el procesador actual puede invalidar su propia TLB, por ello, para eliminar una entrada en todos los procesadores del sistema, se requiere que el sistema operativo mande una interrupción a cada procesador y pida que esa entrada de la TLB sea eliminada.

Esta es una operación muy costosa y por ende tiene su propio nombre y se denomina TLB shutdown.