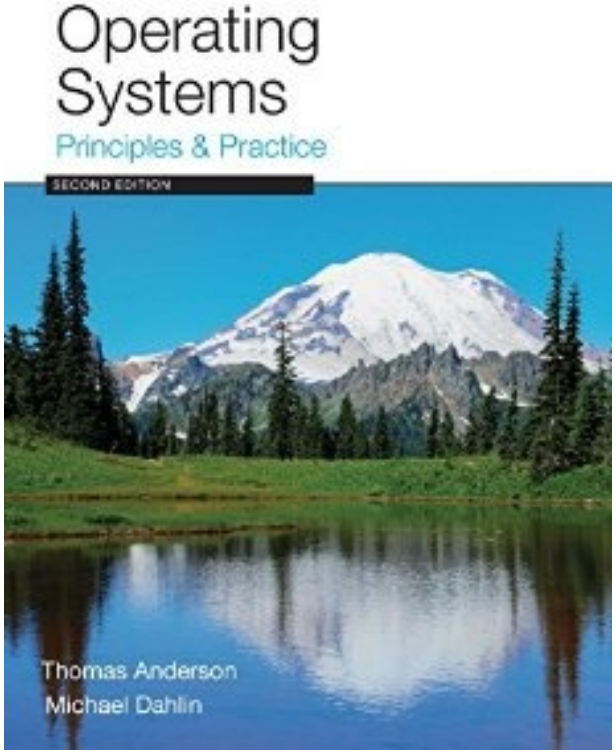
# Sistemas Operativos

## Capítulo 8  Conversión de Direcciones

Prof. Javier Cañas R.

Estos apuntes están tomados en parte del texto: “Operating System: Principles and Practice” de T. Anderson y M. Dahin



# Motivación

- Crear una realidad virtual para los programas. El programa se comporta correctamente independientemente donde se almacene y también en forma transparente al programador.
- La conversión de direcciones, es decir direcciones virtuales a direcciones físicas es un concepto simple, pero extremadamente poderoso.
- La idea principal es separar el espacio de direcciones virtuales del espacio de direcciones físicas.

# ¿Qué se logra?

- Aislar procesos.
- Comunicar procesos: una buena forma es compartir regiones de memoria.
- Compartir segmentos de código.
- Inicialización de programas: Se puede empezar a ejecutar sin tener todo el programa cargado en memoria.
- Asignación dinámica y eficiente de memoria.
- Manejo de Caché: Es posible mejorar la eficiencia de caché con una buena ubicación de memoria.

# …¿Qué se logra?

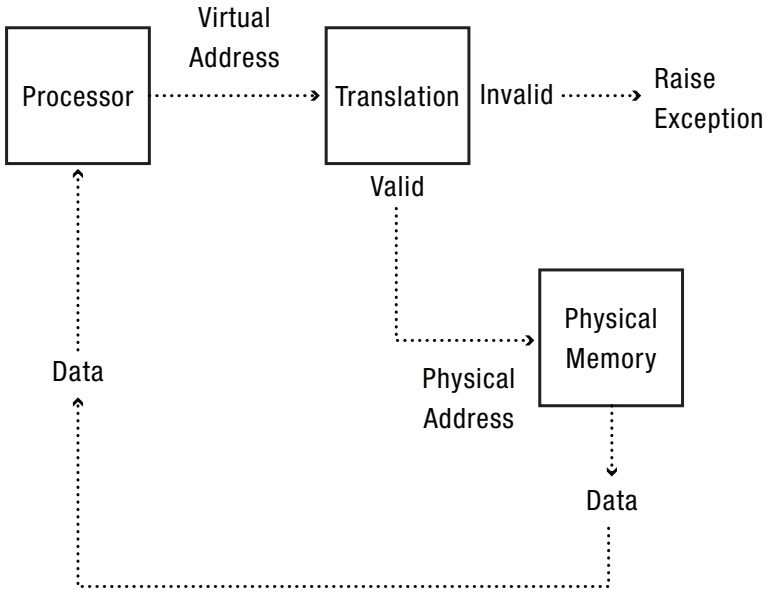
- E/S eficiente: permite que los datos se transfieran en modo seguro direcramente desde aplicaciones a dispositivos de E/S.
- Archivos memory mapped: Archivos se pueden mapear al espacio de direccioes.
- Memoria virtual
- Estructuras de datos persistentes: El SO puede generar la abstracción de una región de memoria persistente.
- Migración de procesos: Mover en forma transparente un programa en ejecución de un server a otro.

# Temario

1. Conceptos
2. Conversión de direcciones flexible
3. Conversión de direcciones eficiente

# 1 Conceptos

- Desde una perspectiva de sistema. la conversión de direcciones es una caja negra:



# … Conceptos

- La conversión es normalmente realizada en HW. El kernel configura este HW.
- ¿Qué hay en la caja negra?. Hay muchas soluciones porque hay muchos objetivos que se pueden lograr

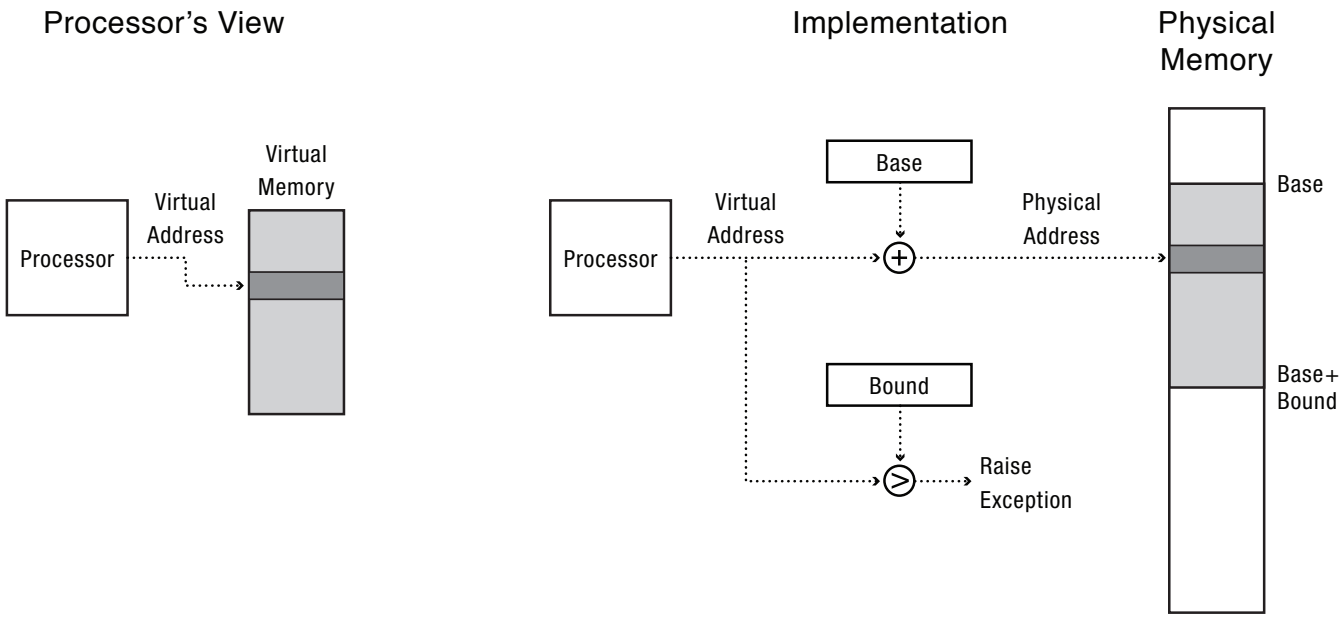
# Objetivos de la Conversión

- Protección de memoria
- Compartir memoria
- Ubicación flexible de proceos
- Dispersión de direcciones
- Eficiencia en tiempo de ejecución: la conversión ocurre en cada instrucción.
- Tablas de conversión compactas
- Portabilidad: Estructuras de conversión independientes de la arquitectura

# 2 Conversión flexible

- Un esquema simple es el visto en el Capítulo 2 utilizando un par de registros: base y límite (bound).
- En este esquema la dirección virtual comienza en 0 y la dirección física en base+bound.
- En cada context switch, el kernel actualiza estos registros con los valores apropiados para cada proceso.

# Base y Límite



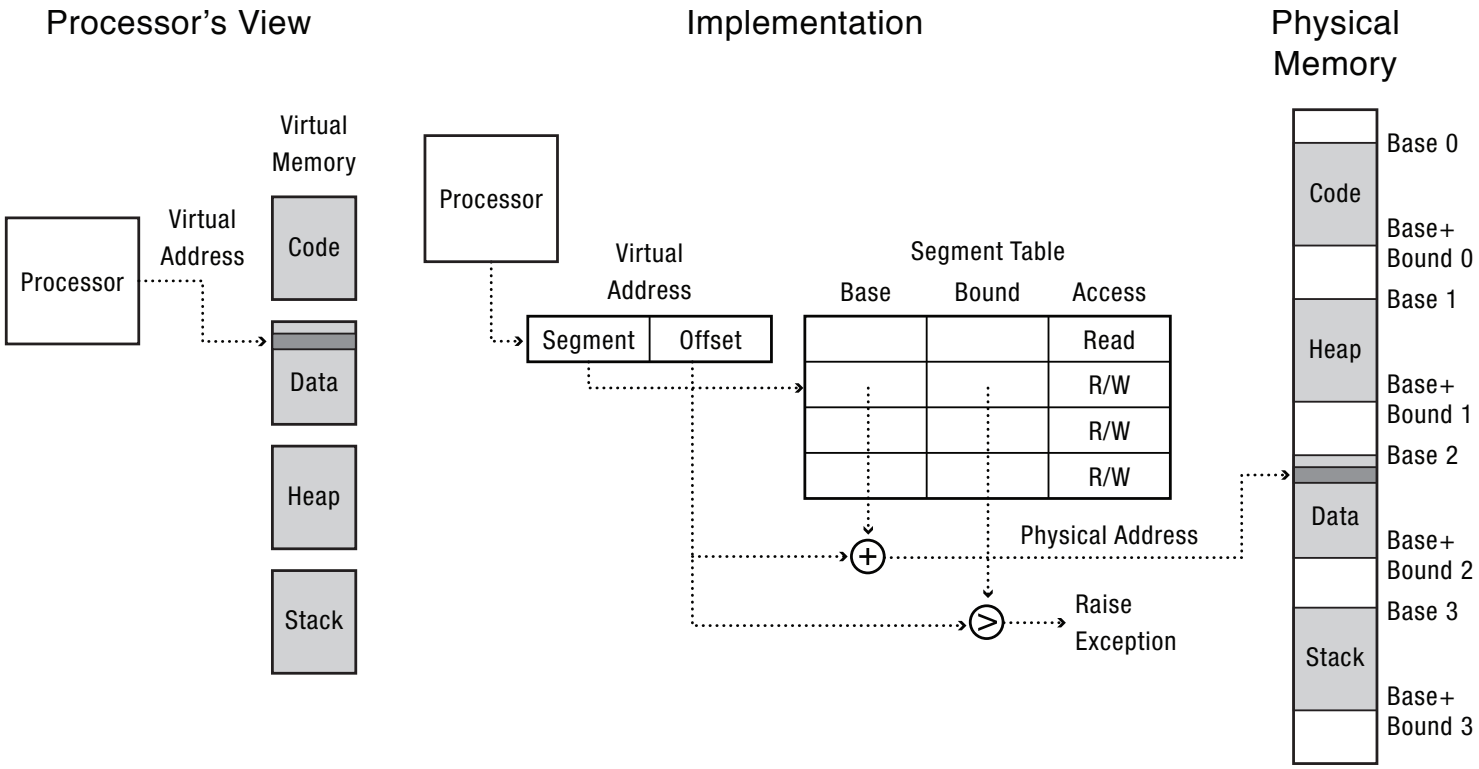
# Observaciones

- Sólo soporta protección de grano grueso, todo el proceso, pero no protección selectiva en un proceso. Un programa podría reescribir su propio código.
- Es difícil compartir regiones de memoria entre dos o más procesos.

# Segmentación

- Muchas de las limitaciones anteriores se solucionan con segmentación. En vez de mantener un par de registros (base, bound) por proceso, se tiene un arreglo de registros base y bound por proceso.
- Cada entrada del arreglo controla una porción o segmento del espacio virtual.
- Cada segmento se almacena en direcciones contiguas, pero diferentes segmentos pueden estar en diferentes localizaciones.

# Segmentación



# Observaciones

- El SO puede asignar distintos permisos a segmentes.
- Entre diferentes segmentos pueden existir gaps de memoria. Si un programa genera un salto a uno de estos gaps, se genera una excepción (*segmentation fault* en UNIX). Programas correctos no hacen esto!.
- Este esquema es simple y poderoso. La arquitectura x86 es segmentada (más otras cosas).
- El SO puede permitir compartir segmentos entre procesos.

Initially, pc = 240.

*What happens first? Simulate machine, building up physical memory contents as we go.*

*1. fetch 240? virtual segment #? 0; offset ? 240. Physical address: 4240 from base = 4000 + offset = 240. Fetch value at 4240. Store address of x into r2 (first argument to procedure).*

*2. fetch 244? virtual segment #? 0; offset ? 244. Physical address: 4244. Fetch value at 4244. Instruction to Store PC + 8 into r31 (return value).*

*5. What is PC? (Note PC is untranslated!) 244 +8 = 24c Instruction to execute after return.*

*6. Fetch 248? Physical address 4248. Fetch instruction: jump 360.*

*7. Fetch 360? Physaddr 4360. Fetch instruction: load (r2) into r3. Contents of r2? 1108.*

*8. r2 is used as a pointer. Contents of r2 is a virtual address. Therefore, need to translate it! Seg # ? 1 offset ? 108. Physaddr: 108. Fetch value a. Do remainder of strlen.*

*9. On return, jump to (r31) -- this holds return PC in *virtual space* -- 24c*

2 bit segment #
12 bit offset

| | Segment start | length |
|---|---|---|
| code | 0x4000 | 0x700 |
| data | 0 | 0x500 |
| heap | - | - |
| stack | 0x2000 | 0x1000 |

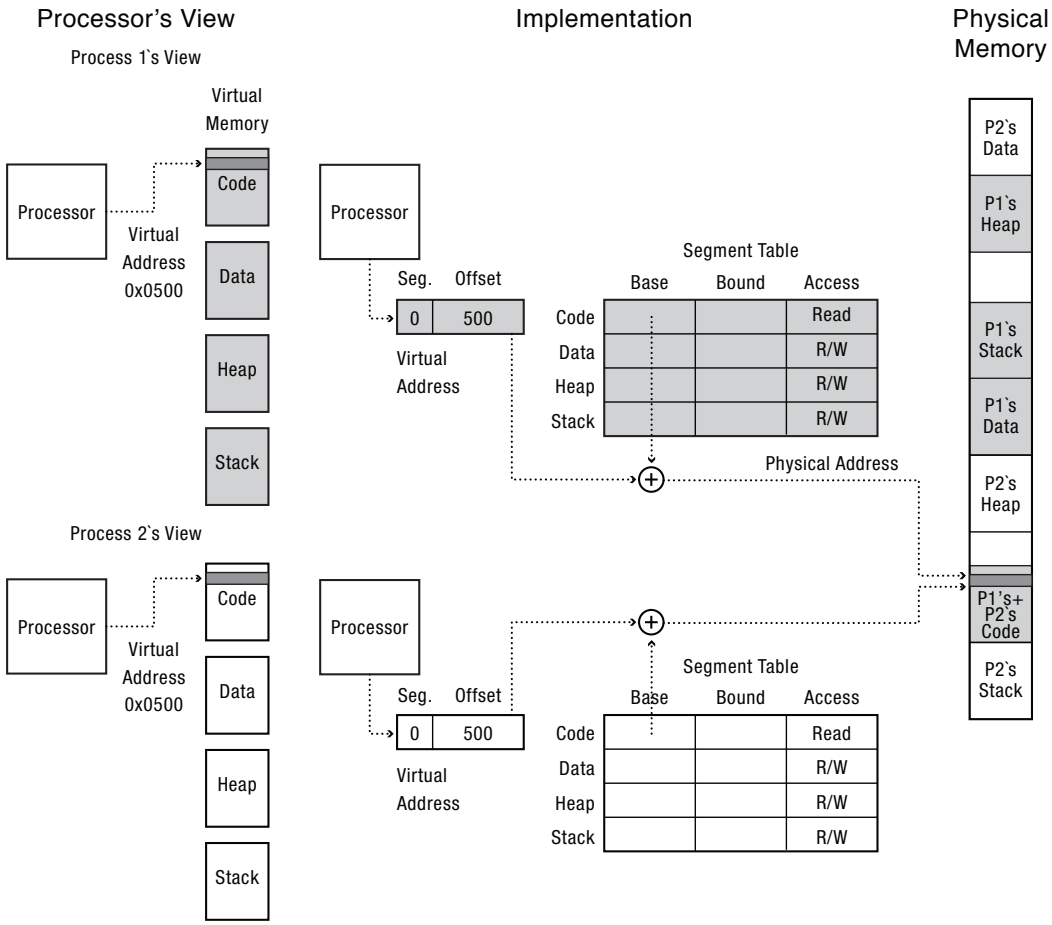
Virtual Memory

| | |
|---|---|
| main: 240 | store #1108, r2 |
| 244 | store pc+8, r31 |
| 248 | jump 360 |
| 24c | |
| … | |
| strlen: 360 | loadbyte (r2), r3 |
| … | … |
| 420 | jump (r31) |
| … | |
| x: 1108 | a b c \0 |
| … | |

Physical Memory

| | |
|---|---|
| x: 108 | a b c \0 |
| … | |
| main: 4240 | store #1108, r2 |
| 4244 | store pc+8, r31 |
| 4248 | jump 360 |
| 424c | |
| … | … |
| strlen: 4360 | loadbyte (r2),r3 |
| … | |
| 4420 | jump (r31) |
| … | |

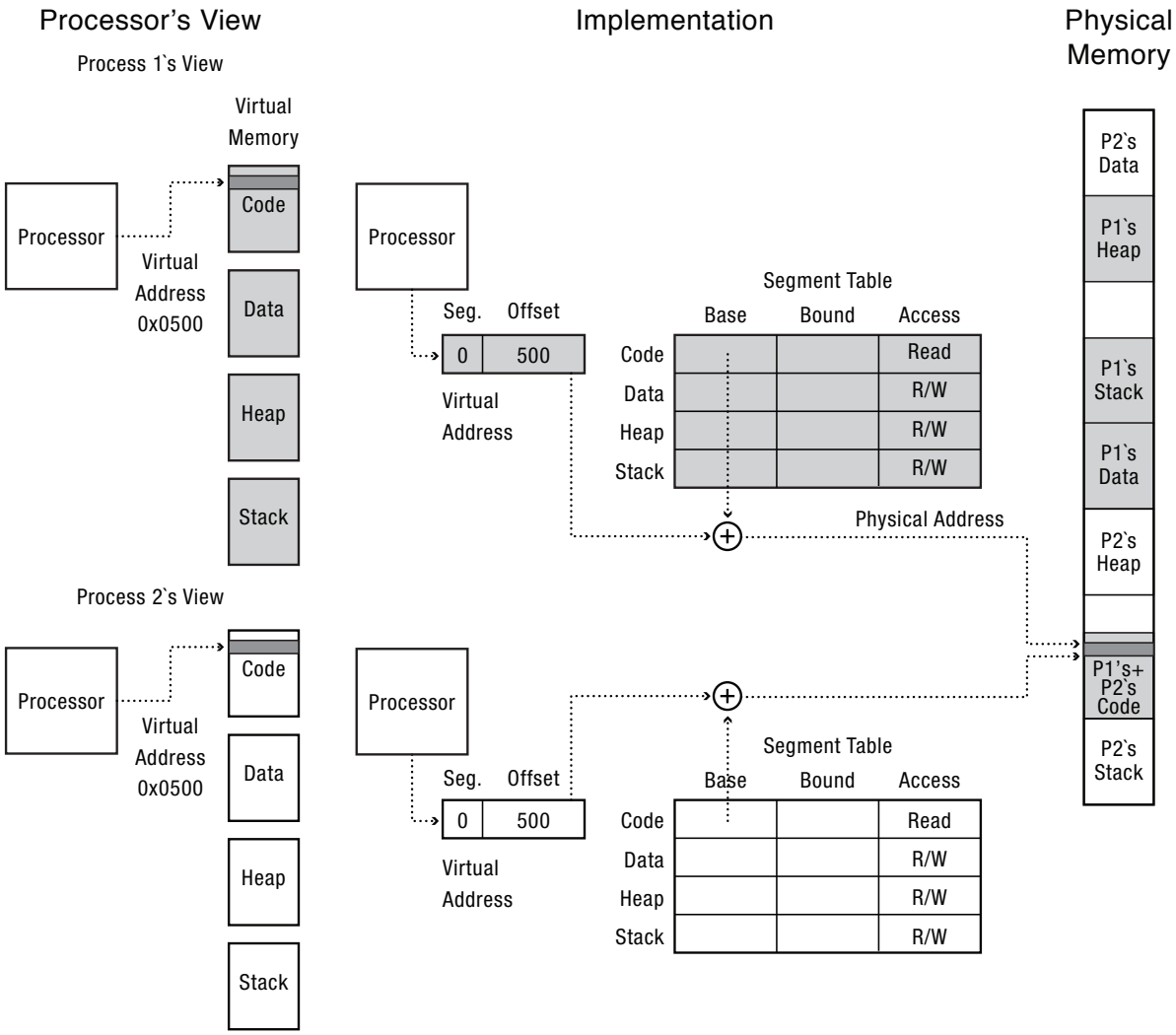
# Dos procesos compartiendo segmento de código



# Copy on Write

- Un fork en UNIX copia completamente un proceso.
- La segmentación (y también paginación), permiten una implementación más eficiente:
  - Se copia la tabla de segmentos al hijo
  - Se marcan segmentos del padre e hijo como read only
  - Se activa el proceso hijo. Return al padre
  - Si el hijo o el padre escriben en un segmento se genera un trap al kernel. El kernel hace una copia del segmento y reasume el proceso.

# Copy on Write



# Zero-on-reference

- La segmentación maneja eficientemente asignación dinámica de memoria. Al reutilizar memoria o disco, el SO limpia los bloque de memoria o disco por seguridad.
- **Zero-on-reference**: El SO asigna una región de memoria para el heap, pero sólo limpia (ceros) los primeros kilobytes y ajusta el registro bound en este límite. Si el programa expande, a través de una excepción el kernel limpia memoria adicional antes de reanudar la ejecución.
- Esta técnica permite asignar cero bytes al stack y heap.

# Limitaciones de la segmentación

- Existe un overhead al manejar un gran número de segmentos de tamaño variable y que pueden crecer dinámicamente.

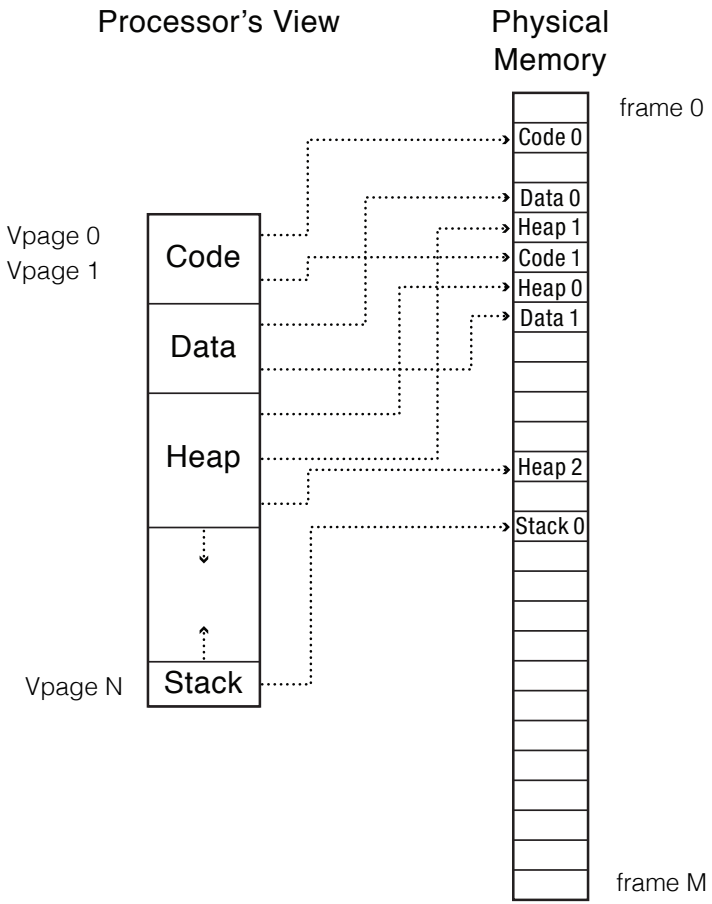
- Como los procesos tienen distintos tamaños, con el tiempo, al crearse y terminar van fragmentando la memoria con gaps de tamaño variable.

- Puede ocurrir que en algún momento la suma de los espacios libres sea suficiente, pero al estar distribuidos, no se pueden asignar generando ***fragmentación externa*** de memoria.

# Paginación

- Una alternativa a la paginación es la segmentación.
- Con paginación la memoria se asigna en trozos de tamaño fijo llamados *frames* (marcos de página).
- La conversión es similar a la segmentación y se utiliza una estructura llamada ***tabla de página***.

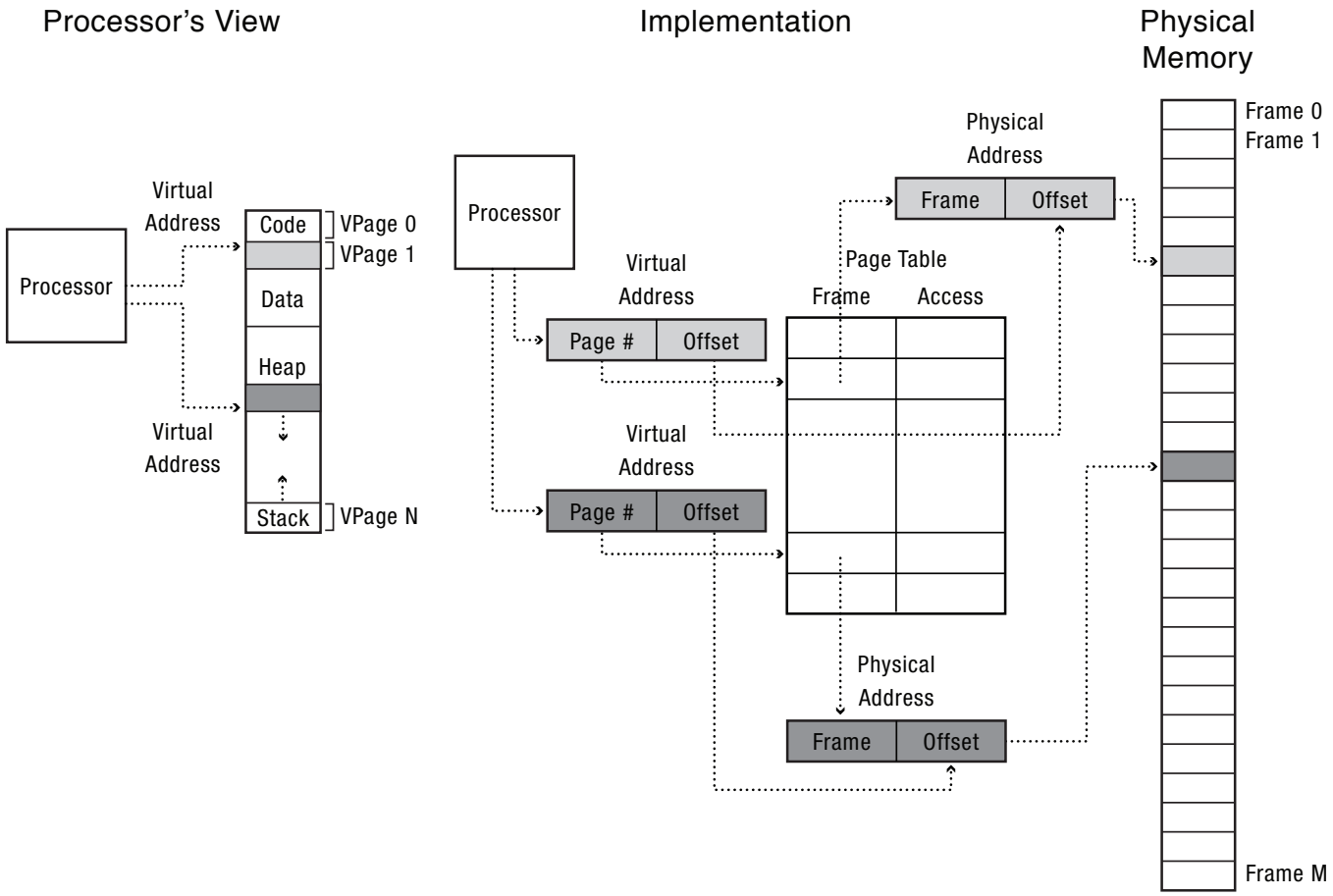
# Visión lógica

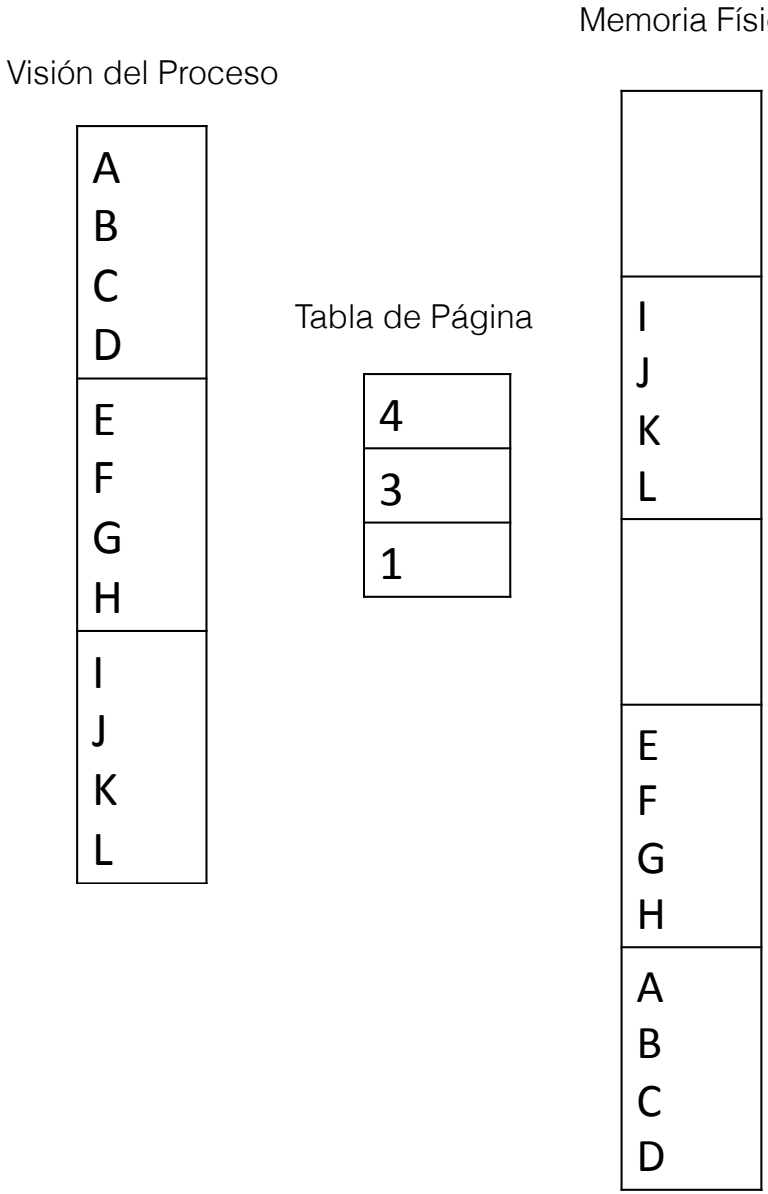


# Observaciones

- El espacio virtual es lineal, pero físicamente está repartido en la memoria. Por ejemplo, dos instrucciones consecutivas físicamente pueden estar en regiones diferentes, lo mismo ocurre con estructuras de datos.
- El SO puede representar la memoria física como un bit map donde un bit muestra si el frame está libre o asignado.
- El SO puede restringir el acceso a determinadas páginas.

# Conversión con Tablas de Página






**Ejercicio**

Páginas de tamaño 4B.
¿Dónde está la dirección virtual 6 y la 9?

# Paginación y partida rápida de programas

- ¿Es posible comenzar a correr un programa antes que su código esté en memoria?
  - Marcar todas las páginas de la tabla como inválidas
  - Cuando la página es referenciada la primera vez:
    - Trap al kernel
    - El kernel carga la página
    - Se reasume la ejecución
  - El resto de las páginas se cargan en background mientras el programa está corriendo

# Limitaciones de la Paginación

- El tamaño de la tabla de página es proporcional al tamaño del espacio virtual, no al tamaño de la memoria física.

- Mientras más disperso esté el espacio virtual, mayor overhead requiere la tabla de página. Muchas entradas estarán inválidas representando direcciones que aún no están en uso.

- Se puede achicar el tamaño de la tabla de página agrandando el tamaño del frame de memoria, pero se genera ***fragmentación interna***.

# Conversión Multi-nivel

- ¿Cómo diseñar un sistema de conversión eficiente?. La búsqueda en un arreglo no lo es.
- Sistemas basados en árboles pueden tienen un comportamiento mejor, pero todo sistema multi-nivel, el el nivel inferior, tienen paginación.
- Esquemas: segmentación+paginación, paginación multi-nivel, segmentación+paginación multinivel.

# Razones para utilizar paginación en menor nivel

1. **Asignación eficiente de memoria**: La paginación permite utilizar bitmaps para manejar espacio libre de memoria.

2. **Transferencias de disco eficientes**: Un disco rotacional está particionado en sectores. Si la página es múltiplo de sectores se simplifican transferencias.

3. **Búsqueda eficiente**: La paginación permite utilizar cachés eficientemente mediante una TLB.

# … Razones para utilizar paginación en menor nivel

4. **Búsqueda reversa eficiente**: permite también ir desde el frame de página físico al conjunto de direcciones virtuales que comparten el mismo frame.

5. **Granularidad en protección y compartición de páginas**.

# Segmentación paginada

- La segmentación paginada es un árbol de dos niveles. Cada entrada de la tabla de segmentos apunta a una página.
- Como la paginación se utiliza en el último nivel, el tamaño de los segmentos es un múltiplo del tamaño de página.
- La tabla de segmentos se puede almacenar en registros especiales de HW, la tabla de página de cada segmento es almacenada en memoria física.

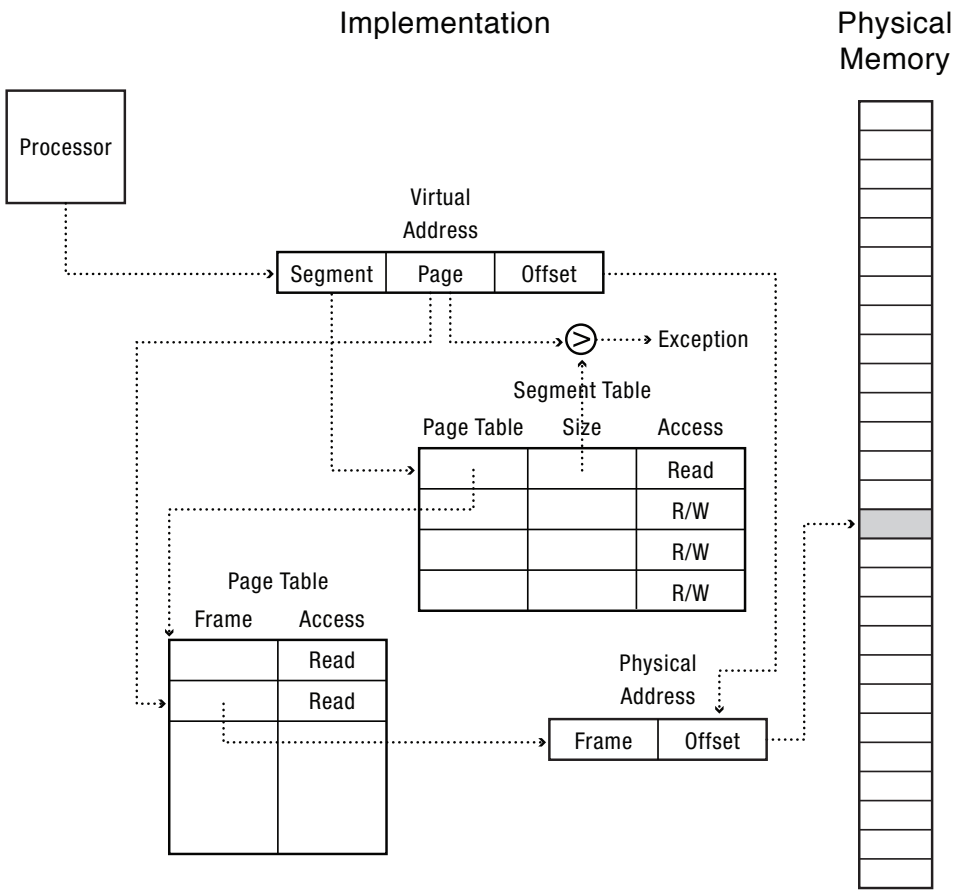
# Ejemplo

- Consideremos:
  - Dirección virtual de 32b
  - Páginas de 4KB
  - Dirección virtual

| N° segmento<br>10b | N° de página<br>10b | Offset de página<br>12b |
|---|---|---|

Si cada entrada de la tabla de página tiene 4B, la tabla de página de cada segmento cabe en un frame de memoria

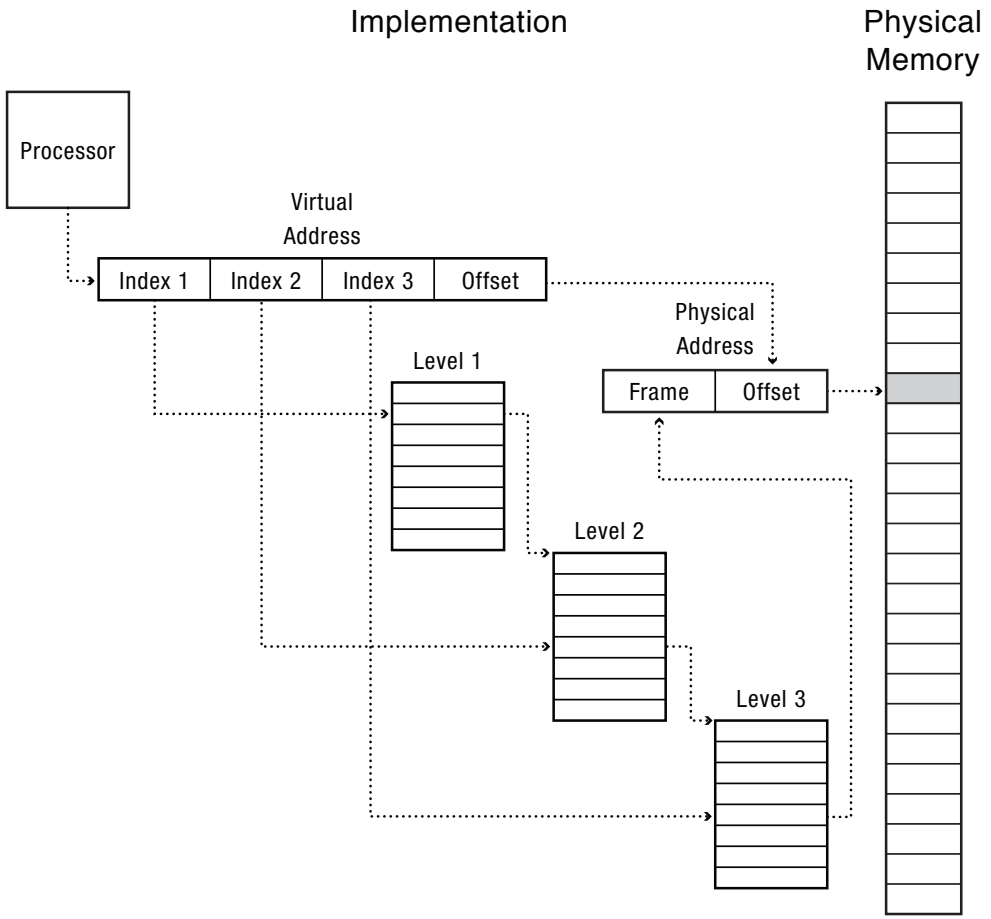
# Segmentación paginada



# Paginación Multi-nivel

- Un esquema similar a la segmentación paginada es utilizar niveles múltiples de paginación.
- Los sistema que utilizan paginación multi-nivel, cada nivel de la tabla de página se diseña para entrar en un frame de página en memoria física.
- Se pueden asignar permisos en cada nivlel y también se pueden compartir páginas en cada nivel por varios procesos.

# Paginación Multi-nivel



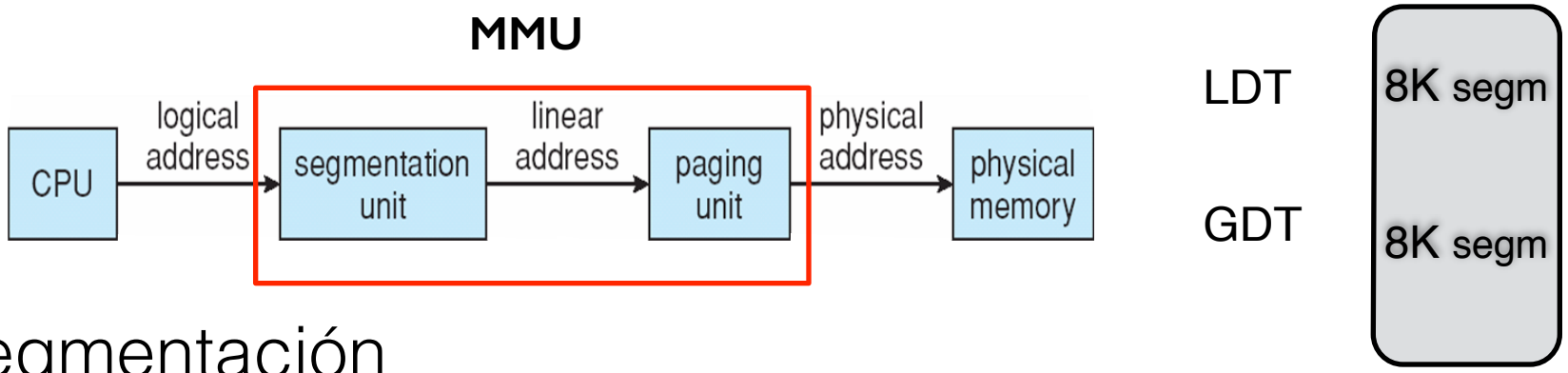
# Segmentación paginada multinivel

- Este esquema es utilizado por el x86 ya sea en 32 o 64bits

# Ejemplo: Pentium

- Soporta segmentación con paginación con un esquema de paginación de dos niveles.
- La CPU genera la dirección lógica. Esta dirección se transforma en direcciones lineales en la MMU.
- Las direcciones lineales alimentan la unidad de paginación que a su vez genera direcciones físicas de memoria principal.

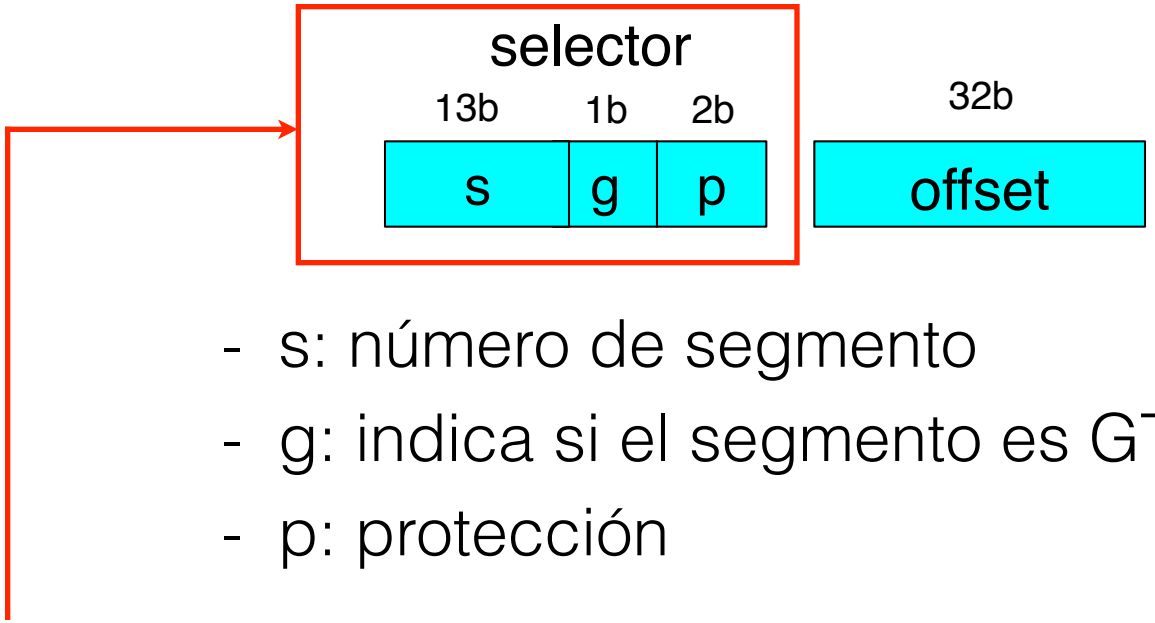
# Pentium




## Segmentación

- Los segmentos pueden llegar hasta 4GB
- El número máximo de segmentos por proceso es de 16K
- El espacio lógico se divide en dos partes:
    - 8K segmentos privados al proceso (LDT: Local Description Table)
    - 8K segmentos compartidos entre todos los procesos GDT: Global Description Table)

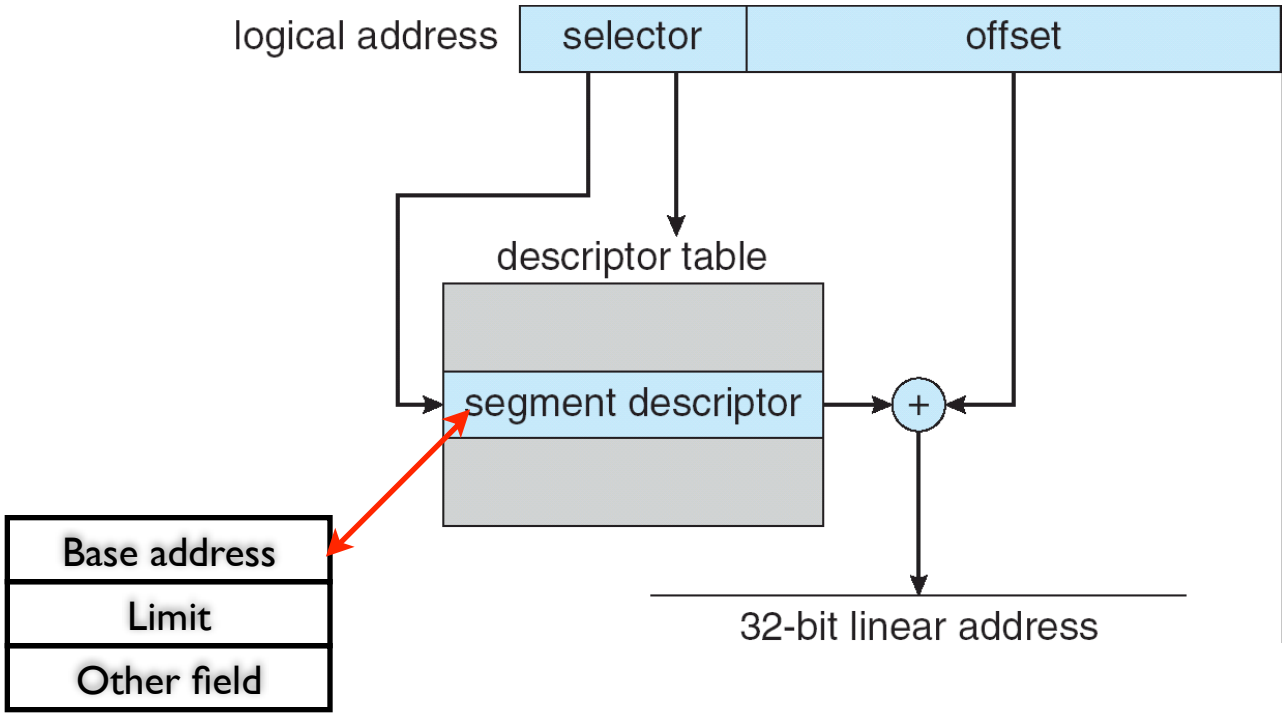
- La dirección lógica es el par <selector, offset>




- s: número de segmento
- g: indica si el segmento es GTD o LTD
- p: protección

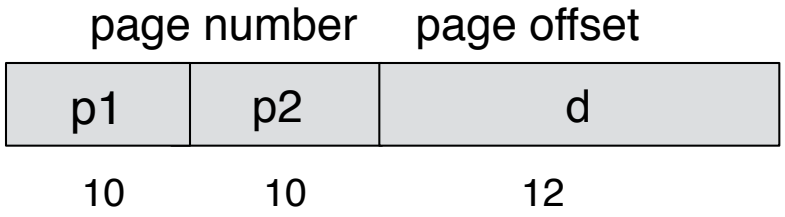
La CPU del pentium tiene 6 registros de 16 bits llamados Selectors

# Unidad de segmentación

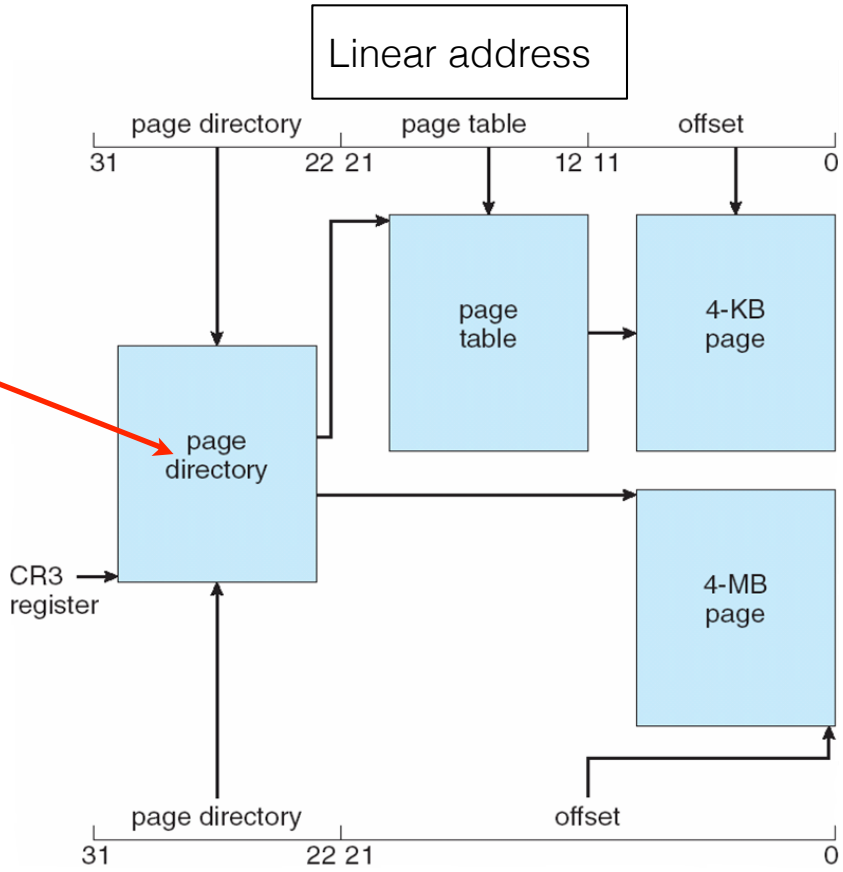


# Paginación en Pentium

- Los tamaños de página sólo pueden ser de  4KB o 4MB.
- Para páginas de 4KB, el Pentium usa un esquema de paginación en dos niveles, dividiendo el espacio lógico de 32b en:



Una entrada en el directorio de páginas indica que el tamaño de la página es de 4KB o 4MB



# 3 Conversión de direcciones eficiente

- La mayoría de los mecanismos vistos, involucran al menos dos o más referencias extras a memoria, lo cual resulta impráctico que un procesador deba referenciar un par de veces la memoria en cada instrucción.

- ¿Cómo mejorar el desempeño sin alterar el comportamiento lógico de la conversión?.

- La solución es utilizar cachés. Las cachés son ampliamente utilizadas en distintos contextos de los sistemas computacionales.

# TLB: Translation Lookaside Buffers

- Si dos instrucciones están en la misma página virtual, entonces están en el mismo frame físico.

- El TLB es una tabla implementada en hardware que contiene el resultado de la reciente conversión de dirección. Cada una de sus entradas mapea una página virtual en una página física.

```
TLB entrada = {
 Número página virtual,
 Número frame físico,
 permisos de accesos,
}
```

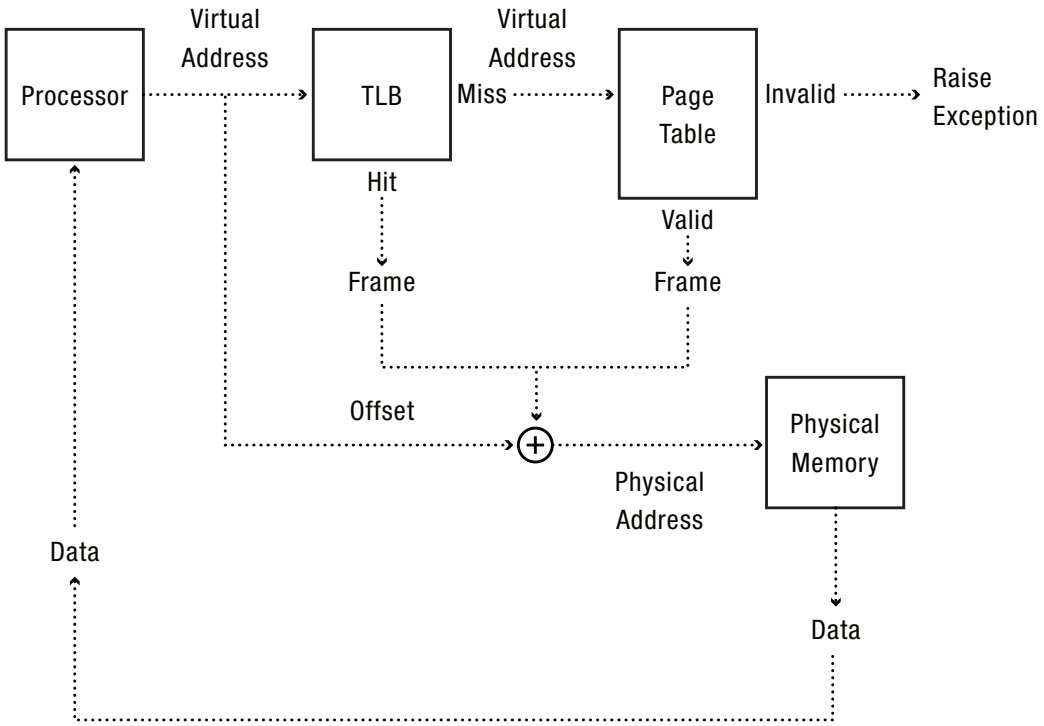
# … TLB

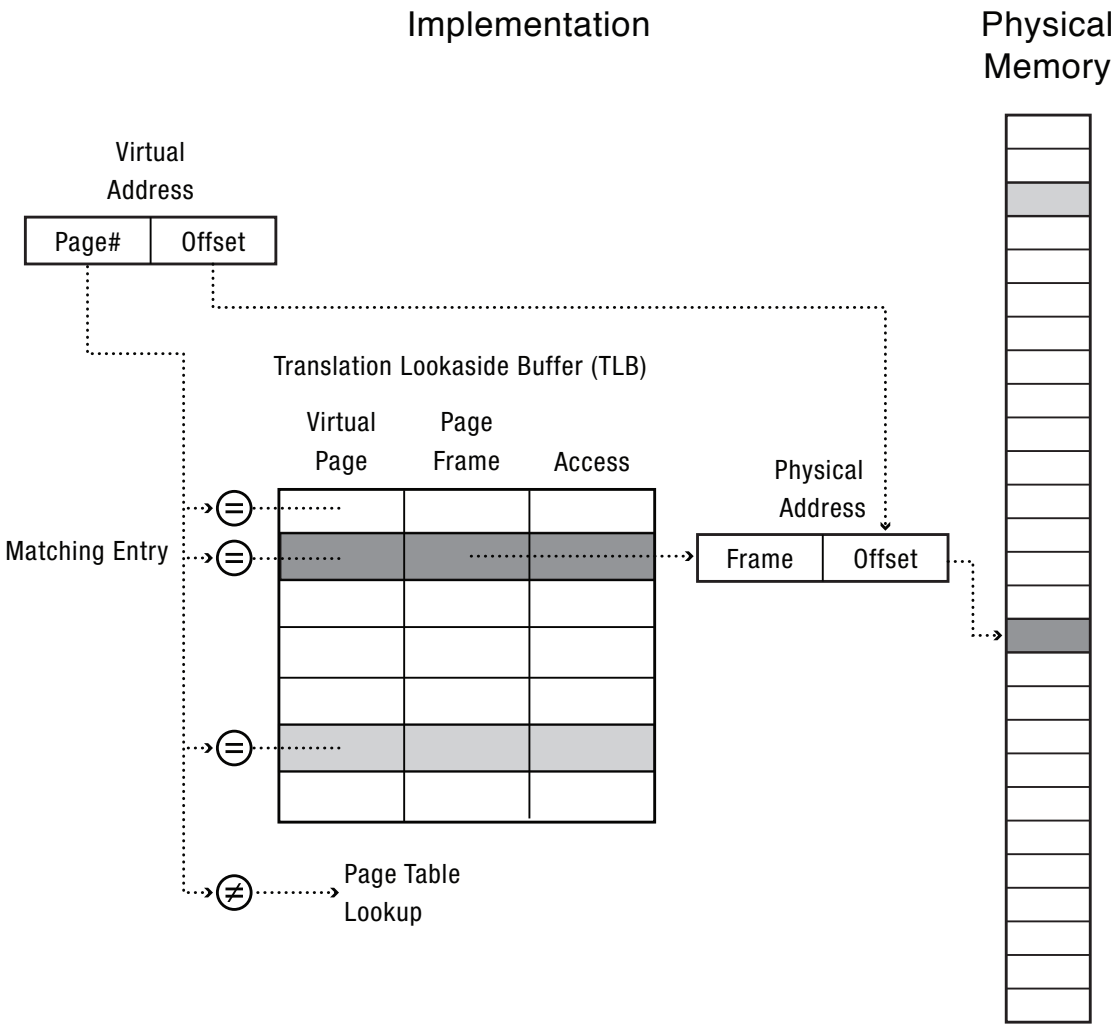
- El TLB hace búsquedas en paralelo (memoria asociativa: busca por contenido). Si hay un calce, utiliza la entrada encontrada, omitiendo los siguientes pasos: ***TLB hit***.

- Un ***TLB miss*** ocurre si no encuentra ningún calce. En este caso el HW debe hacer la conversión completa y el resultado es instalado en la tabla de la TLB.

- Se implementa en memoria estática muy cerca del procesador.

# Visión abstracta TLB




Implementation
Physical Memory
Virtual Address
Page#
Offset
Translation Lookaside Buffer (TLB)
Virtual Page
Page Frame
Access
Physical Address
Frame
Offset
Matching Entry
Page Table Lookup

# TLB: Costo de conversión

- Sea P(hit) la probabilidad que la TLB contenga la dirección.
- El costo de la conversión utilizando TLB es:

Costo= Costo (TLB lookup)+ Costo(conversión total)×(1-P(hit))

# Tiempo de Acceso a Memoria Efectivo (TAE)

- Sea:
  - Tiempo de Lookup=$\varepsilon$
  - Tiempo de acceso memoria=$\eta$
  - P(hit)=$\zeta$

- $TAE=(\eta+\varepsilon)\zeta+(2\eta+\varepsilon)(1-\zeta)$

  $=\eta(2-\zeta)+\varepsilon$

- Se aprecia que para mejorar el TAE se requiere aumentar el $\zeta$ lo que significa un mayor tamaño de la TLB.

# Sistemas Operativos

## Capítulo 8  Conversión de Direcciones

Prof. Javier Cañas R.