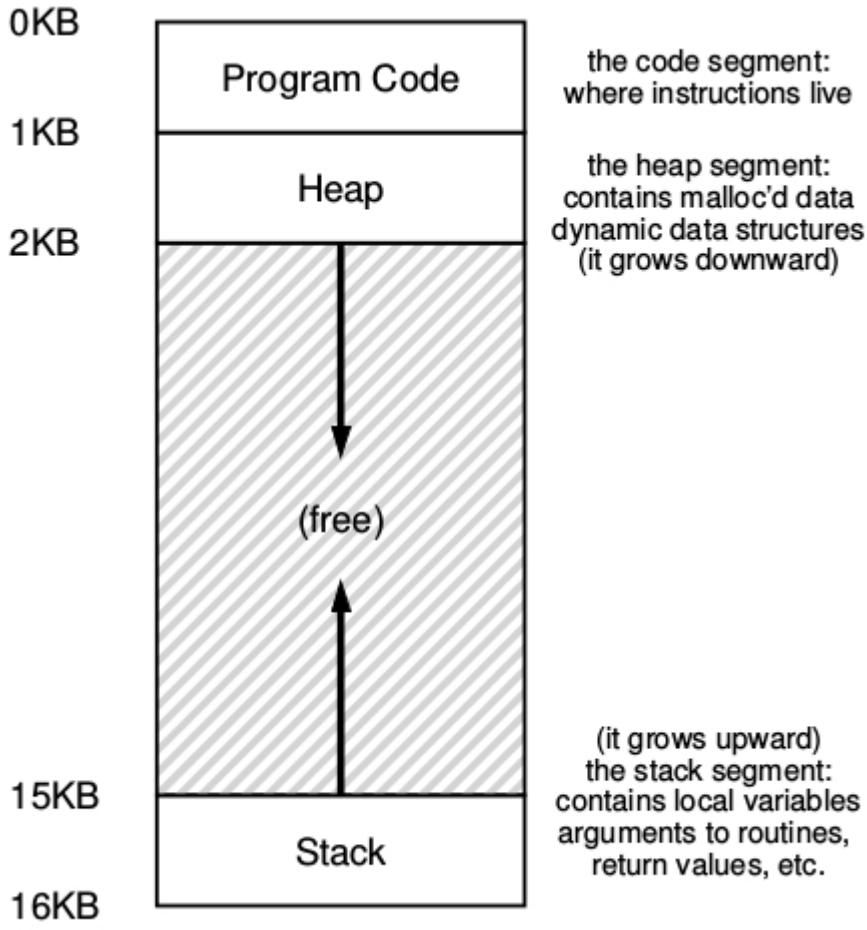
Memoria

Virtualización de Memoria

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



- El **código fuente** del programa vive en lo **alto del espacio de direcciones**. El **código fuente es estático** por ende se puede poner al principio del espacio de direcciones y **no necesitará más espacio** mientras que el programa se ejecute.
- Hay dos regiones del espacio de direcciones que pueden **crecer** o **achicarse** mientras el programa se esta ejecutando, el **Heap** y el **Stack**, debido a que ambas en algún momento van a querer crecer siempre se ponen en los **extremos** del espacio de direcciones **enfrentadas** entre si.
  De esta forma se permite tal crecimiento solamente que el mismo se dirige a **direcciones opuestas**:
  - El heap empieza justo después del código fuente y crece hacia abajo.
  - El stack empieza al final del espacio de direcciones y crece hacia arriba.

Cuando se describe el **espacio de direcciones**, se está describiendo la **abstracción** que el **Sistema Operativo** le proporciona al **programa en ejecución** sobre la memoria de la computadora.

Cuando el Sistema Operativo implementa esta abstracción, se dice que el O.S. está **Virtualizando la Memoria** ya que el programa en ejecución cree que está cargado en un lugar particular de la memoria y tiene potencialmente toda la memoria para él.

Cuando un proceso trata de cargar el contenido de la **dirección virtual 0**, el **sistema operativo** con ayuda de algún tipo de **mecanismo de hardware** tendrá que asegurarse que se cargue la **dirección física** en la cual el espacio de direcciones de ese proceso se encuentra alojado.

Metas principales de la virtualización:

- **Transparencia:**
  El sistema operativo debería implementar la virtualización de memoria de forma tal que sea **invisible** al programa que se esta ejecutando; 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 implementar la ilusión.
- **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 mas lentos) y **espacio** (no usar demasiada memoria para las estructuras necesarias para soportar la virtualización).
- **Protección:**
  El sistema operativo tiene que asegurarse de **proteger a los procesos unos de otros como también proteger al sistema operativo de los procesos**.
  La protección habilita una propiedad llamada **aislamiento entre procesos**; cada proceso tiene que ejecutarse en su propio caparazón aislado y seguro de otros procesos con fallas o incluso maliciosos.

> 🔥 **!**
>
> Un buen mecanismo de virtualización de memoria debe ser lo más flexible y eficiente posible.

Tipos de Memoria

En un programa en ejecución existen dos tipos de memoria, cuando se ejecuta un programa escrito en C existen dos tipos de memoria que se reservan.

1. **Stack**: su reserva y liberación es manejada implícitamente por el compilador en nombre del programador. La declaración de memoria en el stack en C es sencilla, se declara una variable de algún tipo de dato y el compilador se encarga de hacer el resto.

   Ejemplo

   ```
   void func() {
       int x; // declara un entero en el stack
       . . .
   }
   ```

2. **Heap**: este tipo de memoria es obtenida y liberada *explícitamente* por el programador.

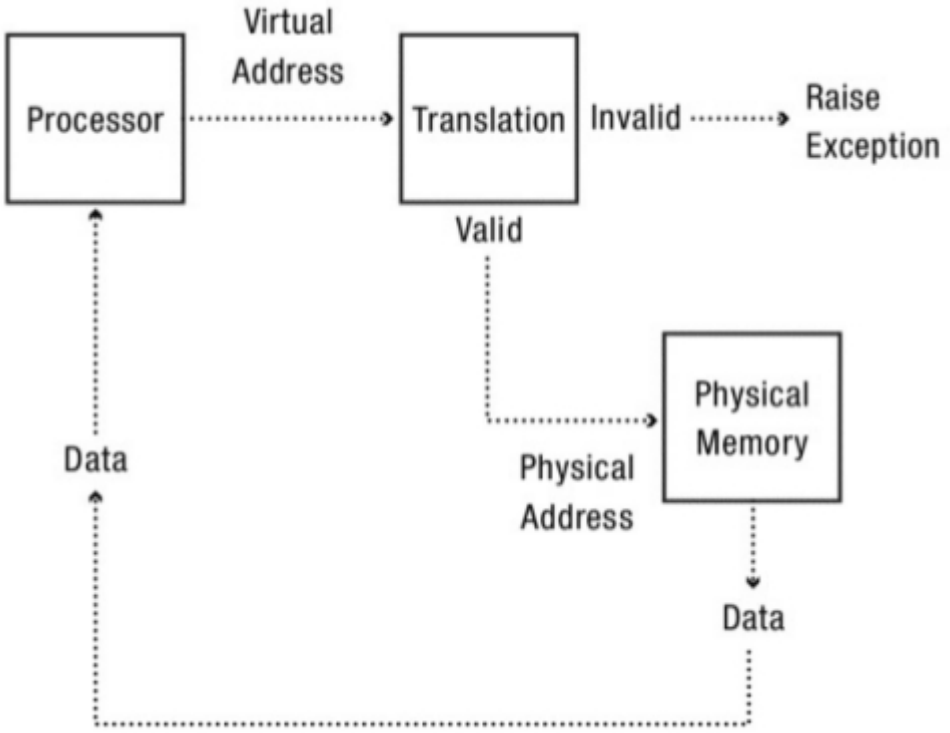
   Ejemplo

   ```
   void func() {
       int *x = (int *) malloc(sizeof(int));
       . . .
   }
   ```

Address Translation

Un buen mecanismo de virtualizacion de memoria debe ser lo mas **flexible** y **eficiente** posible.
Esto se logra mediante la utilización de una técnica llamada **Hardware-Based Address Translation** o como más comúnmente se conoce **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 es el término por el cual se conoce al mecanismo, proporcionado por el **hardware**, que permite a partir de una dirección virtual obtener la dirección física correspondiente. El S.O. está en el medio de este proceso y determina si el mismo se ha realizado correctamente. Lo que hace es gerenciar el manejo de la memoria:

1. Manteniendo registro de que parte está libre.
2. Que parte está en uso.
3. Manteniendo el control de la forma en la cual la memoria está siendo utilizada.

Dynamic Realocation o Memoria Segmentada (base and bound)

Sólo se necesitan dos registros de hardware dentro de cada cpu: Uno llamado **registro base** y el otro **registro límite** o **segmento** (bound).

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

En esta configuración, cada programa es escrito y compilado como si fuera cargado en la dirección física 0. A su vez cuando se inicia la ejecución del programa, el OS decide en que lugar va a cargar el mismo y para hacerlo setea el **registro base** en un determinado valor.

A partir de ahora cualquier referencia generada por el proceso, es traducida por el procesador de la siguiente manera:

physical address = virtual address + base

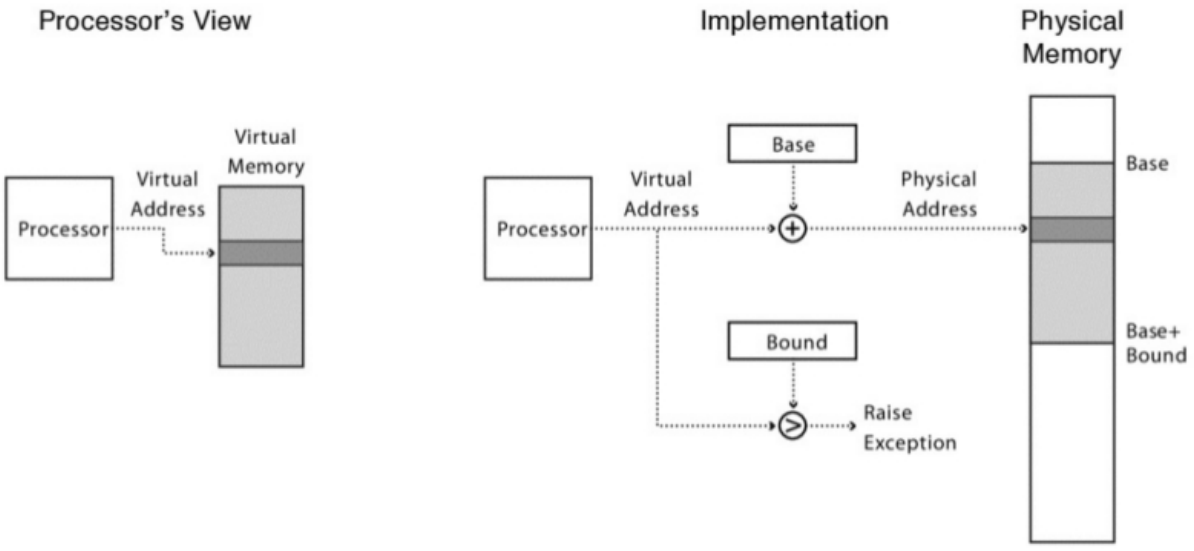
Cada referencia de memoria generada por el procesador es una **dirección virtual**; el hardware cada vez que se hace referencia a esta dirección tiene que sumarle el contenido del registro base y su resultado es la **dirección física** que tiene que ser utilizada en la memoria del sistema.

El **registro bound** está allí para ayudar con la protección. Específicamente, el procesador antes que hacer nada va a checkear que la **referencia de memoria** este dentro de los **limites** del **address space** para asegurarse que la misma sea legal.

Si un proceso genera una **dirección virtual** que es mayor que los límites o una dirección que es negativa, la CPU va a generar una excepción y el proceso va a terminarse.

El **bound register**, puede ser definidos en dos formas diferentes:

1. Este registro mantiene el tamaño del address space, entonces el hardware chequea la dirección virtual contra el bound register, sumándole primero el registro base.
2. El bound register almacena la dirección física del fin del espacio de direcciones.



Address Translation con Tabla de Segmentos

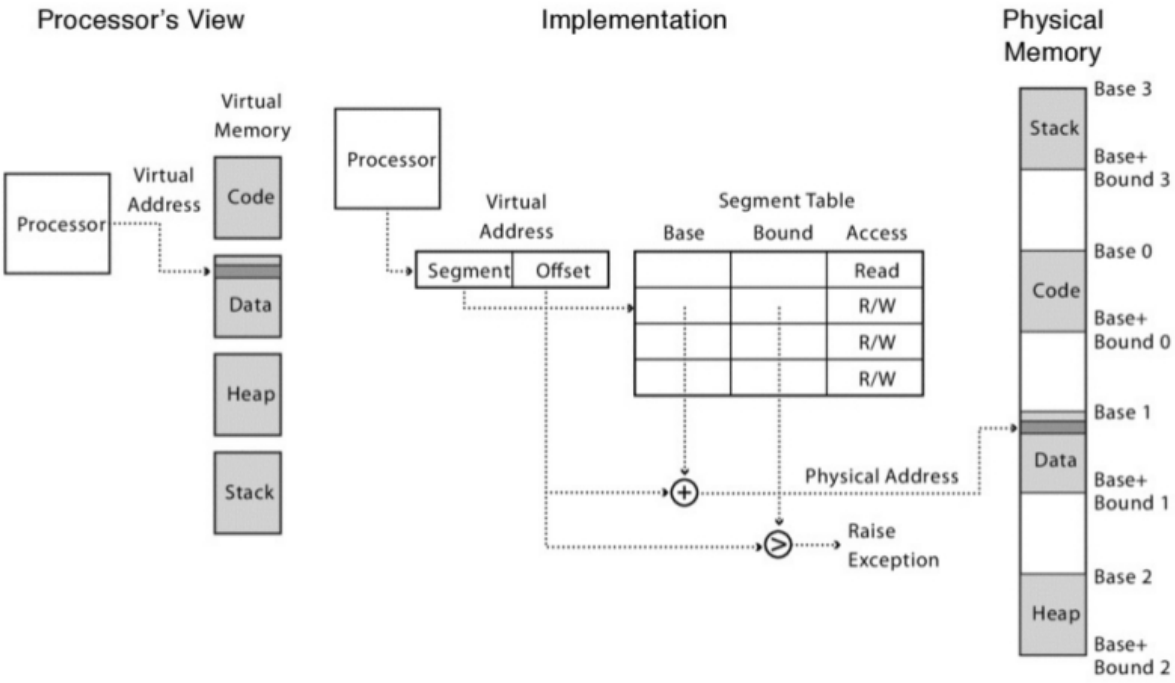
En este método en vez de tener un solo registro limite, se tiene un arreglo de **pares de registro base, segmento por cada proceso**.
Cada entrada en el arreglo controla una porción del virtual address space. La memoria

física de cada segmento es almacenada continuamente, pero distintos segmentos pueden estar ubicados en distintas partes de la memoria física.

Una dirección virtual tiene dos componentes:

- Un **número de segmento**: es el índice de la taba para ubicar el inicio del segmento en la memoria física.
- Un **offset de segmento**: 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.

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



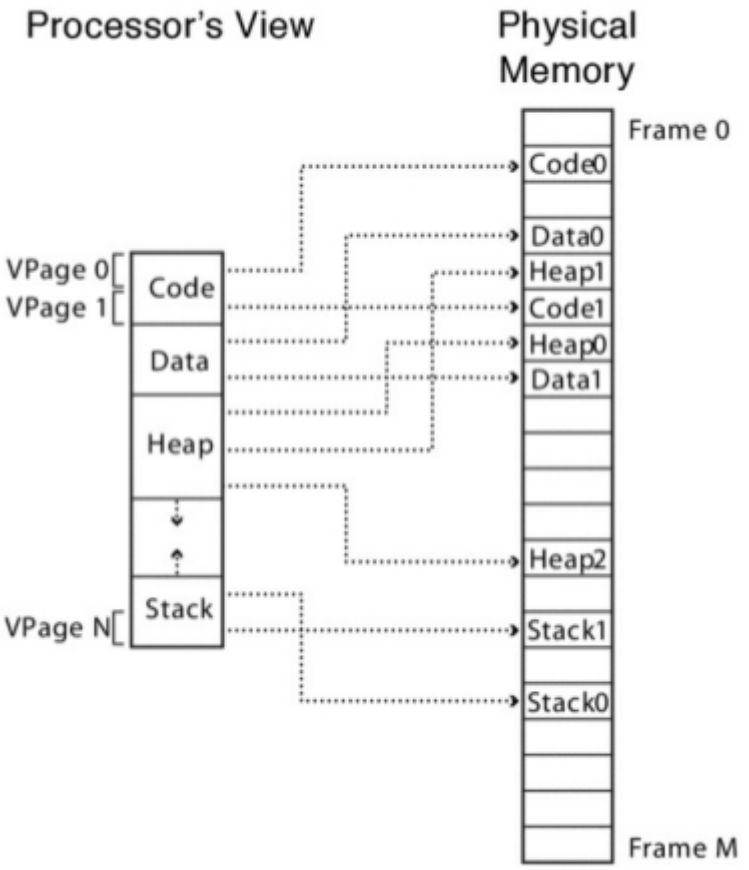
Memoria Paginada

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

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. No es necesario tener un límite en el offset; la página entera en memoria fisica se reserva como una unidad.



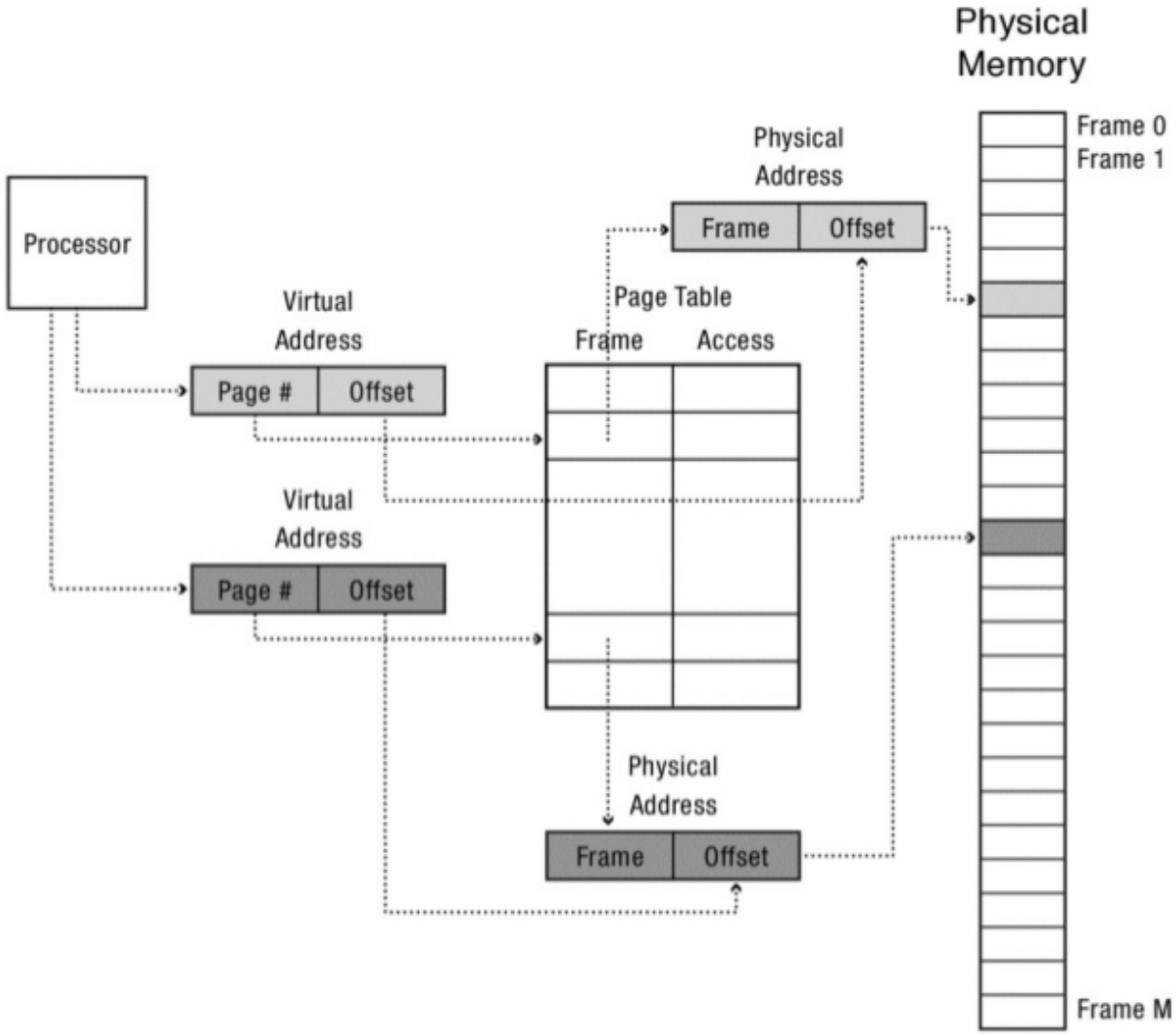
Address translation con Page Table

La vitual address tiene dos componentes:

- El **número de página virtual**: es el índice en la page table para obtener el page frame en la memoria física.
- El **offset dentro de esa página**.

La dirección física está compuesta por la dirección física de la 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.

Si bien el programa cree que su memoria es lineal, de hecho, su memoria está desparramada por toda la memoria física.



Multilevel page segmentation

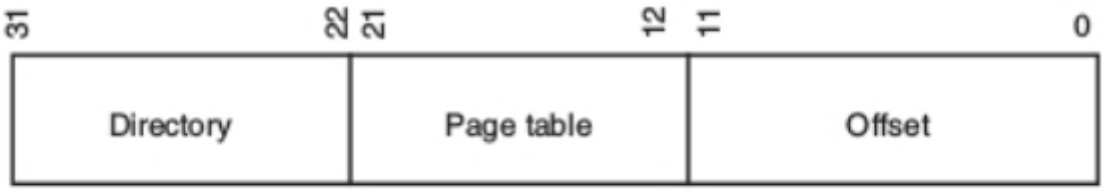
Este es el sistema utilizado por las arquitecturas x86 para 64 y 32 bit.

En la arquitectura x86 cada proceso posee una **Global Descriptor Table (GDT)**, que es equivalente a la segment table. La GDT es almacenada en la memoria; cada entrada a esta tabla apunta a una tabla de paginas (multinivel) para ese segmento. Para inicializar el proceso de address translation el sistema operativo setea la GDT e inicializa un registro llamado gdtr GDTR que contiene la dirección y la longitud de la GDT.

Por cada 32 bit en la arquitectura x86 la virtual address posee un segmento a una tabla de 2 niveles:

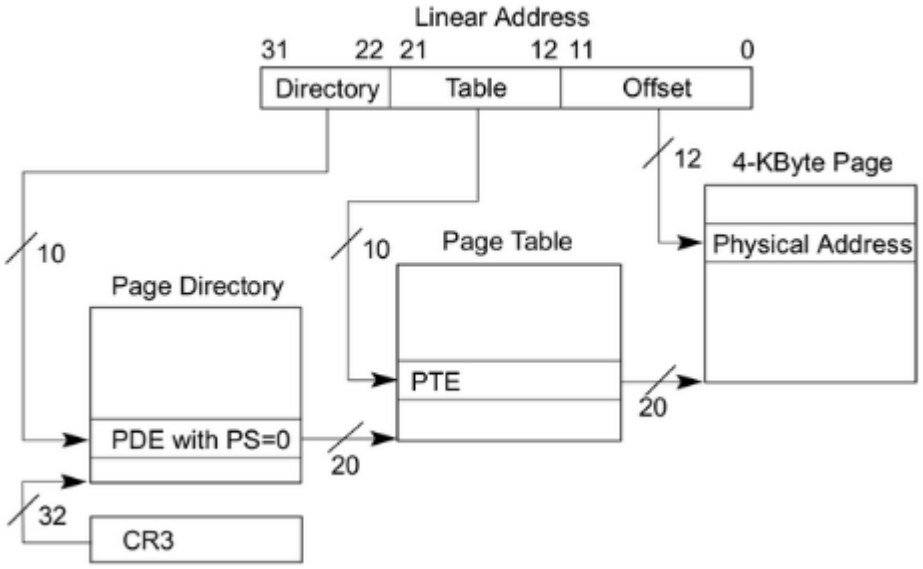
- Los primeros 10 bit de la virtual address son el indice de la paged table de primer nivel, llamada **page directory**,
- los otros 10 bit son el indice de una page table de segundo nivel
- y finalmente los 12 bit restantes son el offset dentro de la pagina.

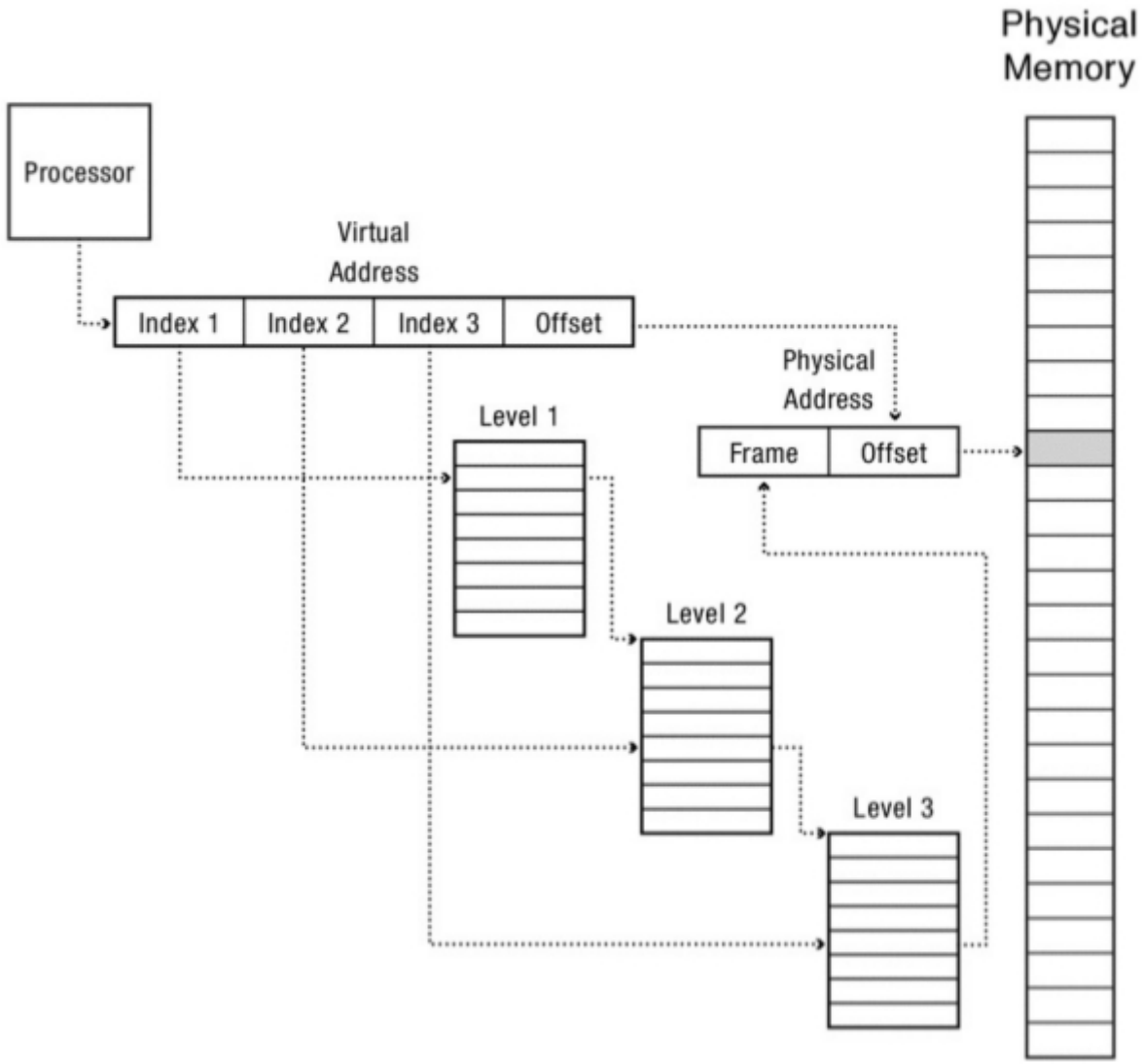
Cada entrada en la page table ocupan 4 bytes y existen 1024 entradas por ende el tamaño total de la page directory es de 4k, sucede lo mismo con la page table de segundo nivel y justo coincide con el tamaño de una pagina en la memoria física.



(a)

Page Directory bits 31-22 (10 bits)
Page Table Entry bits 21-12 (10 bits)
Memory Page offset address bits 11-0 (12 bits)





Hacia una Eficiente Address Translation

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

Para ello se usará un **caché**, que consiste en una copia de ciertos datos que pueden ser accedidos mas de una vez más rápidamente.

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

La TLB es parte de la MMU y es simplemente un mecanismo de caché de las traducciones más utilizadas entre los pares virtual to physical address.

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).

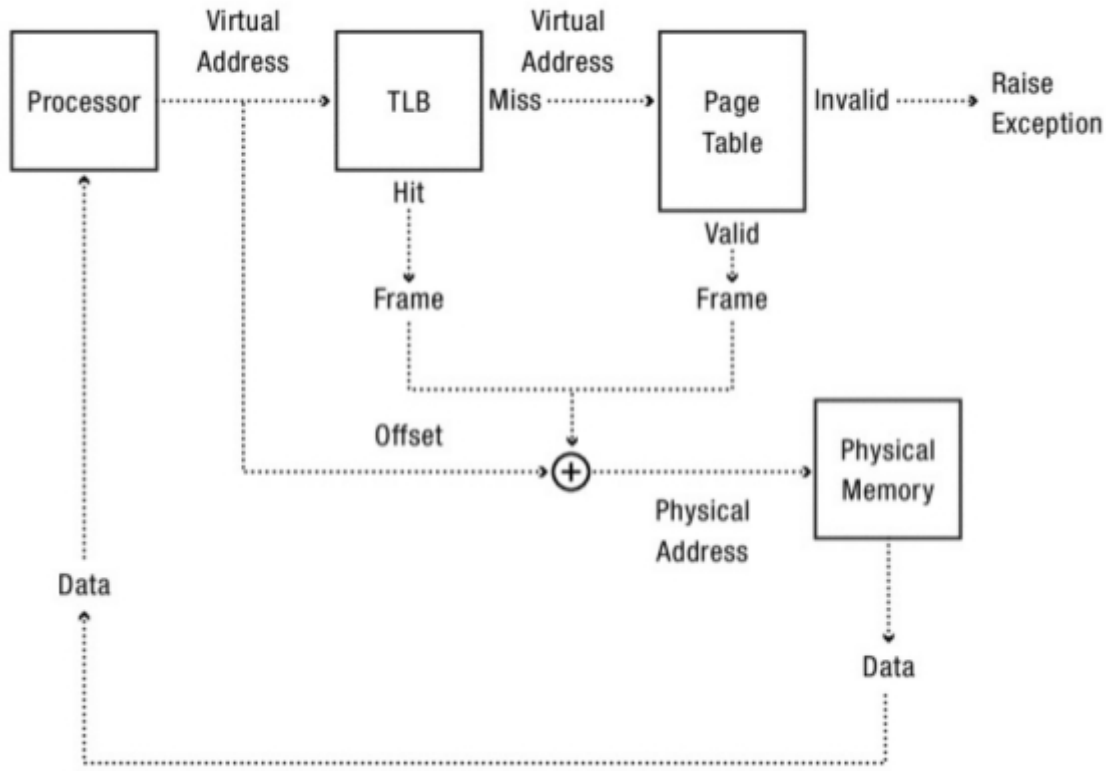
Este mecanismo tiene un tremendo impacto en la velocidad de la traducción.

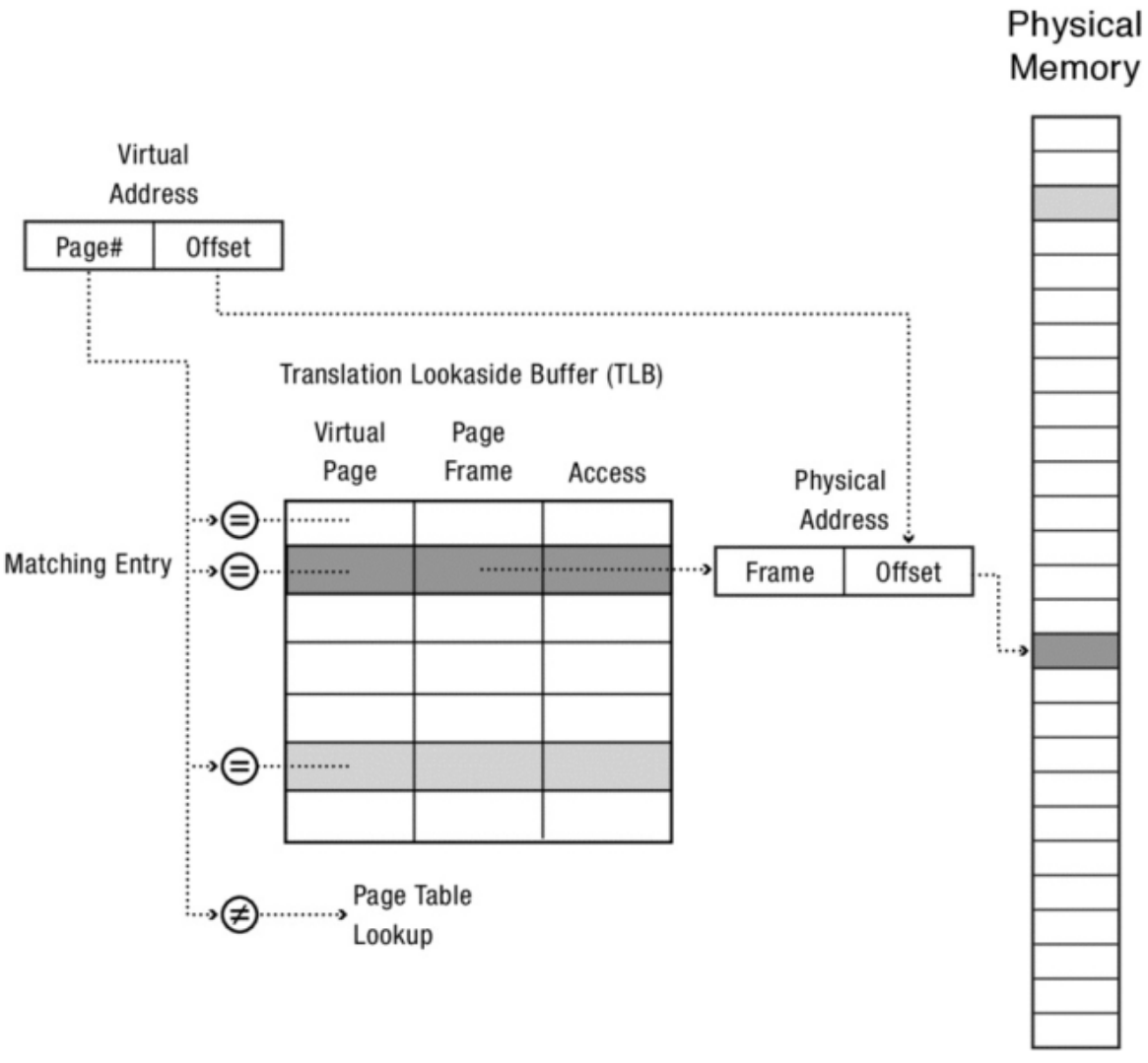
Translation-Lookaside Buffer (TLB)

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**.

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

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**.


Processor
Virtual Address
TLB
Virtual Address
Miss
Page Table
Invalid
Raise Exception
Hit
Valid
Frame
Frame
Offset
Physical Memory
Physical Address
Data
Data


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

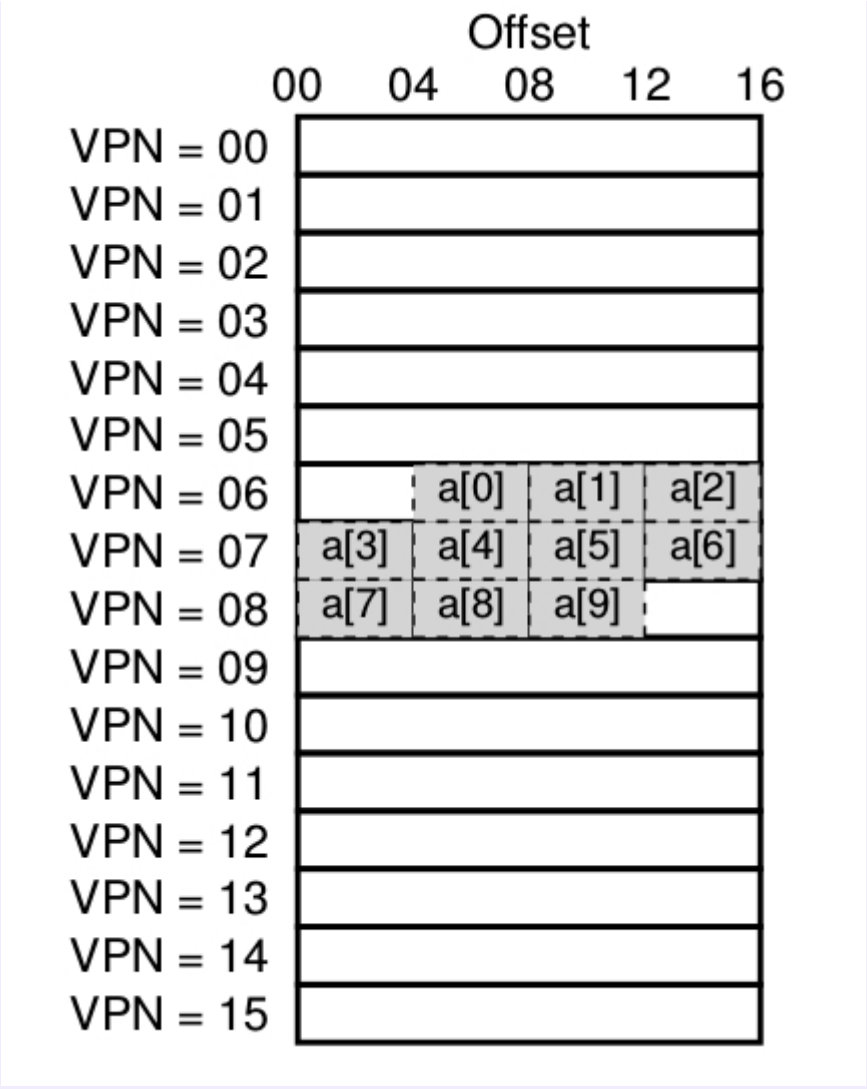
1. Primero se extrae la **virtual page number (VPN)** de la virtual address y se chequea si la TLB tiene esa traducción para esa VPN. Si existe esa traducción se tiene un **TLB HIT** que significa que la TLB posee esa traducción.
2. A partir de ahí, se puede extraer el **Page Frame Number (PFN)** de la entrada en la TLB.
3. Se concatena con el offset de la virtual address original y con ello se forma la **physical address** deseada y se accede a la memoria (suponiendo que todos los chequeos de protección no fallaron).

Si la CPU no encuentra esa traducción en la TLB (se tiene un **TLB MISS**), hay que hacer un poco más de trabajo para obtener la physical address:

1. El hardware accede la page table para encontrar la traducción.
2. Suponiendo que la referencia a la memoria virtual generada por el proceso es válida y accesible, se actualiza la TLB con la traducción hecha por el hardware. Este conjunto de acciones es costoso debido a que existen accesos extras a memoria.
3. Finalmente, una vez que la TLB fue actualizada, el hardware vuelve a buscar la instrucción en la TLB y por ende la referencia a la memoria es procesada rápidamente.

Ejemplo: accediendo a un arreglo

Se asumirá que se tiene un arreglo de 10 números enteros de 4 bytes en memoria cuya dirección virtual inicial es 100.
Asumiendo además, que se tiene un pequeño espacio de direcciones virtual de 8 bits con paginas de 16 bytes, entonces una virtual address se divide en 4 bits (que representan 16 paginas virtuales) y offsets de 4 bits (que representan los bytes de cada una de estas páginas). Es decir, el espacio de direcciones esta compuesto por 16 paginas y cada página posee 16 bytes.



Entonces, teniendo en cuenta la imagen, el primer elemento del arreglo **a[0]** empieza en la **vpn=06 offset=04**, tener en cuenta que en esa página solo encajan 3 enteros de 4 bytes (a[0], a[1], a[2]).
El arreglo después continua en la siguiente página **vpn=07** ocupando las 4 entradas (a[3], a[4], a[5], a[6]) .
Y finalmente las últimas 3 entradas del arreglo (a[7], a[8], a[9]) se localizan en la siguiente pagina del espacio de direcciones VPN=08.

1. El hardware extrae la VPN para esta dirección (VPN=06) y utiliza eso para validar en la TLB la traducción.
2. Asumiendo que es la primera vez que el programa accede al arreglo, el resultado de esta operación va a ser una **TLB MISS**.
   El próximo acceso a un elemento del arreglo, va a generar un **TLB HIT**, debido a que el segundo elemento esta puesto a continuación del primero y vive por ende en la misma página; y debido a que ya se ha accedido a esa página la primera vez que se accedió al arreglo, esta traducción se encuentra cargada en la TLB.
3. Acceder al tercer elemento del arreglo va a caer en la misma situación, otro **HIT** porque este también vive en la misma página.

4. Desafortunadamente, cuando el programa quiere acceder al elemento 4 **a[3]** se encontrará con otro **TLB MISS**. Sin embargo, otra vez los elementos 5, 6 y 7 del arreglo van a ser **TLB HIT**, ya que estos residen en la misma página de memoria.
5. Finalmente, cuando se acceda al octavo elemento otra vez se obtendrá un **TLB MISS**.
   El hardware volverá a consultar la page table para averiguar la ubicación de esa página virtual en la memoria física y por ende actualizará la TLB de acuerdo a eso.
   Los últimos 2 accesos reciben un beneficio porque la TLB se encuentra al día, logrando que la TLB obtenga 2 **HIT** como resultado.