Universidad de Santiago de Chile

Facultad de Ingeniería Informática

Sistemas Operativos

PEP 3
Administración de Memoria

Nicolás Aguilera

Julio 2023
v. 0.9

Índice

1. Requisitos de la gestión de memoria

1.1. Reubicación

El programador no sabe dónde su programa será ubicado en memoria principal cuando éste es cargado y ejecutado. Adicionalmente sería buena idea poder intercambiar procesos en la memoria principal para maximizar el uso del procesador. Una vez el programa ha terminado necesitamos **reubicar** el proceso en otro lado, de otra manera sería contraproducente.

Por tanto, no se puede conocer de forma anticipada dónde se va a colocar un programa y se debe permitir que los programas se puedan mover en la memoria principal, debido al intercambio o *swap*. Tal y como muestra la Figura 7.1 las referencias a memoria deben ser traducidas a direcciones de memoria física.

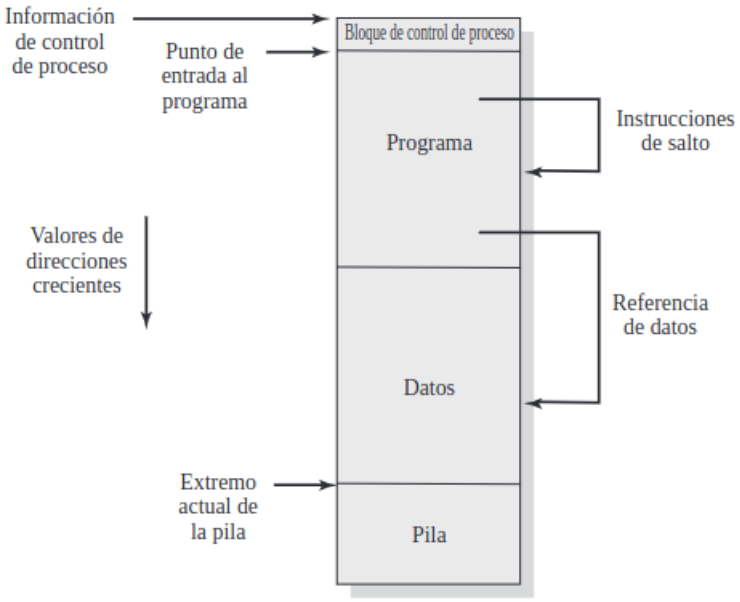


Figura 7.1. Requisitos de direccionamiento para un proceso.

1.2. Protección

Los procesos no deberían ser capaces de referenciar direcciones de memoria fuera de su espacio de direcciones, a excepción de la **memoria compartida**. La mayoría de lenguajes de programación permite el cálculo dinámico de direcciones en tiempo de ejecución, por lo que las referencias a memoria deben ser comprobadas en tiempo de ejecución para **asegurar que solo se refieren al espacio de memoria** de dicho proceso.

Obsérvese que los requisitos de protección de memoria deben ser satisfechos por el procesador (hardware) en lugar del sistema operativo (software). Esto es debido a que el

sistema operativo no puede anticipar todas las referencias de memoria que un programa hará.

1.3. Compartición

Se debe permitir que los procesos compartan áreas de memoria y la sincronización. Esto es buena idea ya que es más ecónomico en términos de memoria que cada proceso tenga una copia de los datos a los que accede.

1.4. Organización lógica

Si el sistema operativo y el hardware del computador pueden tratar de forma efectiva los programas de usuarios y los datos en la forma de módulos de algún tipo, entonces se pueden lograr varias ventajas:

- Los programas son organizados como módulos, y estos módulos (datos, código, etc.) pueden ser escritos y compilados de manera independiente.
- Se pueden proporcionar distintos grados de protección a los módulos: solo escritura, solo lectura.
- Existen mecanismos que permiten compartir los módulos entre procesos, proporcionando una ventaja al usuario que puede especificar la compartición deseada.

La herramienta que más adecuadamente satisface estos requisitos es la **segmentación**, que es una de las técnicas de gestión de la memoria exploradas en este capítulo.

1.5. Organización física

La memoria se organiza en dos grandes niveles: memoria principal y memoria secundaria. Una de las principales preocupaciones del sistema es la comunicación y el flujo entre estas dos memorias.

Esta preocupación podría ser entregada al programador. Por ejemplo, si la memoria principal es insuficiente para un programa más sus datos, se podría utilizar la ténica conocida como **overlaying** donde se puede asignar la misma región en memoria para varios módulos e intercambiarlos entre disco y memoria con la ayuda de técnicas de compilación. Sin embaro, esto quita tiempo al programador. Por lo tanto, esta debiera ser una preocupación de la que se encargue el sistema.

2. Particionamiento de la memoria

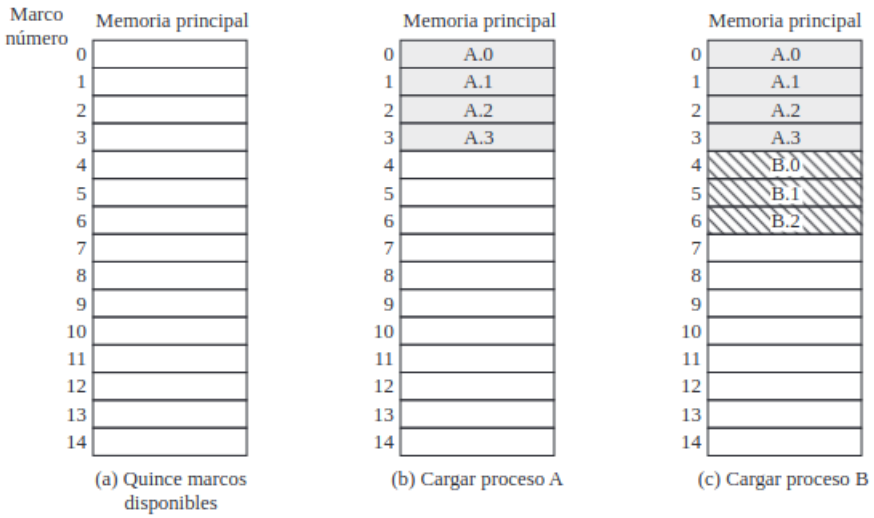
La principal cuestión de la gestión de memoria es traer los procesos a la memoria principal para que el procesador los pueda ejecutar. Lo anterior implica el uso de la llamada **memoria virtual**. Sin embargo, antes de adentrarnos en eso, es necesario facilitar el camino entiendo otros métodos de gestión de la memoria más sencillos como lo son el particionamiento fijo y dinámico.

2.1. Particionamiento fijo

3. Paginación

Supongamos por un momento que tanto la memoria como los procesos se dividen en porciones pequeñas del mismo tamaño que denominaremos **páginas** a las cuales se les asigna porciones de memoria conocidas como **marcos** o marcos de páginas. Aquí se verá que la memoria malgastada por un proceso debido a la fragmentación interna corresponde solamente a una fracción de la última página del proceso.

La Figura 1 y Figura 2 explican el uso de marcos y páginas. Imagina que un proceso A conformado por cuatro páginas quiere ser cargado en memoria y dado que está libre se carga directamente. Luego un proceso B (tres páginas) y C (cuatro páginas) también son cargados en espacios contiguos. Por algún motivo el proceso B se suspende y por lo tanto deja de estar en memoria. Ahora, un proceso D de cinco páginas requiere ser cargado en memoria pero no hay cinco marcos contiguos.



A pesar de esto el proceso D es cargado en memoria debido a la existencia de una **tabla de páginas** (Figura 2) que el sistema operativo tiene por cada proceso. Esta tabla de página muestra la ubicación del marco por cada página del proceso. En este caso la traducción de direcciones lógicas a físicas son realizadas por el procesador presentando una dirección lógica (número de página y *offset*[1]) y entregando una dirección física (número de marco y *offset*).

Para hacer un uso correcto de este método, tanto el tamaño del marco y de la página deben ser 2^n.

[1] Campo de desplazamiento.

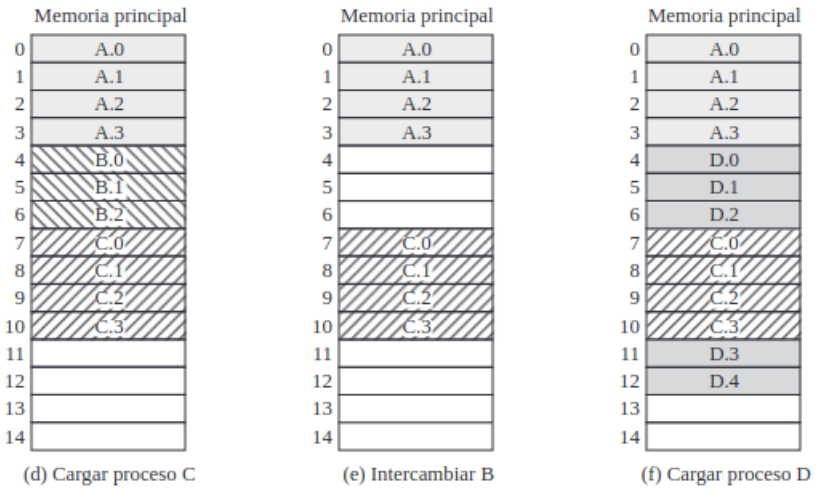


Figura 1: Ocupación de marcos por parte de procesos.

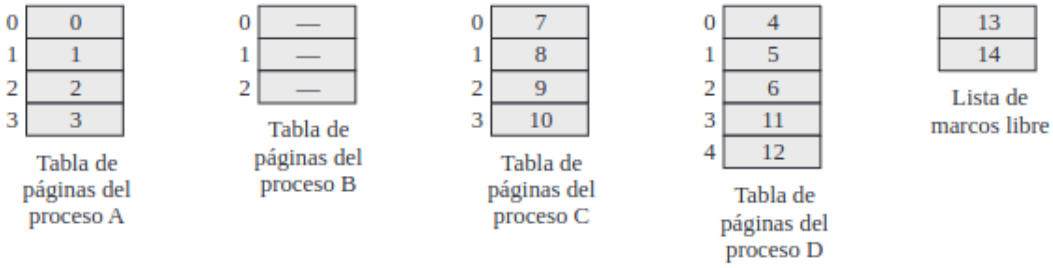


Figura 2: Tabla de página para los procesos en Figura 1.f.

A continuación se presenta un ejemplo como se muestra en la Figura 3. Se consideran direcciones de 16 bits con un tamaño de página de $1K = 1024$ bytes. La dirección relativa 05DE es 0000 0101 1101 1110. El tamaño de página de 1024 bytes equivale a 2^{10}, es decir 10 bits de *offset*, por lo que el resto corresponde a 6 bits para el número de página.

Por lo tanto, el *offset* es 01 1101 1110 en hexadecimal 1DE y el número de página es 000001 en hexadecimal 1, es decir, se está haciendo referencia a la página 1 del proceso, la cual está haciendo referencia a 000110, es decir al marco 06. Para obtener la dirección física concatenamos el número de marco con el *offset*, es decir 0001100111011110, en hexadecimal 19DE. Por lo tanto, la dirección relativa 05DE equivale a 19DE en memoria física.

4. Segmentación

En este caso el programa se divide en un número de **segmentos** lógicos desde el punto de vista del programador: programa principal, datos y funciones. Si bien no se requiere que todos los programas sean del mismo tamaño existe una longitud máxima para los segmentos. En este caso una dirección lógica se compone de un **número de segmento** y

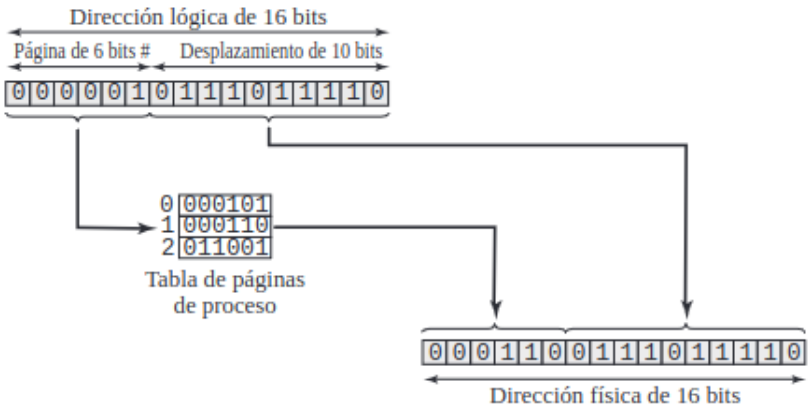


Figura 3: Paginación.

un **offset** dentro del segmento.

Al igual que con la paginación, consideremos direcciones de 16 bits donde $n = 4$ bits son del número de segmento y $m = 12$ bits de *offset*. Se necesita llevar a cabo los siguientes pasos para la traducción de direcciones:

- Extraer el número de segmento como los n bits de la izquierda de la dirección lógica.
- Utilizar el número de segmento como un índice a la **tabla de segmentos** del proceso para encontrar la dirección física inicial del segmento.
- Comparar el desplazamiento, expresado como los m bits de la derecha, y la longitud del segmento. Si el desplazamiento es mayor o igual que la longitud, la dirección no es válida.
- La dirección física deseada es la suma de la dirección física inicial y el desplazamiento tal y como muestra la Figura 4.

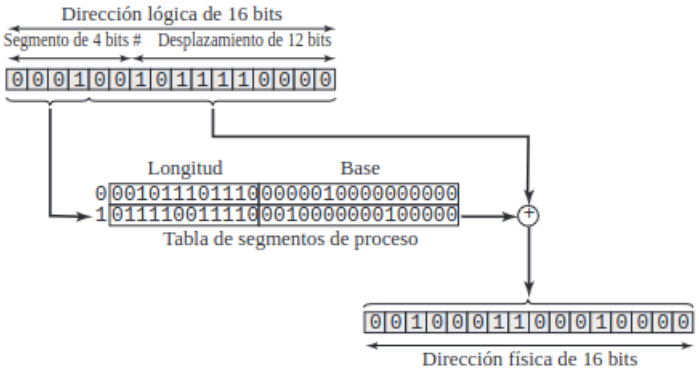


Figura 4: Paginación.

Para el caso del ejemplo, la dirección lógica es 0001 0010 1111 0000. Como el número de segmento considera los 4 bits iniciales, esta apunta al número de segmento $0001_2 = 1_{10}$, el cual tiene como base en la memoria física la dirección de 0010 0000 0010 0000. Luego, la dirección física es:

$$\begin{array}{rrl} & 0010\ 0000\ 0010\ 0000 & \text{base} \\ + & 0010\ 1111\ 0000 & \text{offset} \\ \hline & 0010\ 0011\ 0001\ 0000 & \end{array}$$

5. Memoria virtual

5.1. Hardware y estructuras de control

Las características de paginación y segmentación que son la clave para esta parte son:

- Todas las referencias a memoria dentro de un proceso se hacen mediante direcciones lógicas, las cuales se traducen en direcciones físicas durante la ejecución.
- Un proceso puede dividirse en varias porciones (segmentos o páginas) que no necesariamente deben estar de forma contigua en la memoria, esto debido a la existencia de las tablas de páginas o segmentos.

5.2. Ejecución de un programa

Supongamos que se quiere traer un nuevo programa a memoria. El SO comienza trayendo solamente unos pocos trozos de este, es decir, la porción inicial del programa y la porción inicial de datos sobre la cual las primeras instrucciones acceden. Este conjunto es llamado **conjunto residente**.

Si el programa intenta acceder a una dirección que no se encuentra en el conjunto residente, entonces se produce una falla en el sistema, un *page-fault*, el programa en cuestión se bloquea y el SO realiza una interrupción en la que permite a otro proceso ejecutarse mientras carga los datos del programa bloqueado para reanudar su ejecución. Este modelo tiene dos grandes ventajas:

- **Pueden existir mayor cantidad de procesos en memoria principal**, dado que se cargan porciones de los programas, permitiendo un uso más eficiente del procesador.
- **Un proceso puede ser más grande que toda la memoria principal**, ya que el programa es cargado y ejecutado en porciones se permite que este sea más grande que la memoria física.

La memoria donde el programa se ejecuta es llamada **memoria real**, mientras que el programador percibe una memoria potencialmente más grande llamada **memoria virtual**, permitiendo sortear las restricciones de la memoria real.

5.3. Proximidad y memoria virtual

Si se considera un programa de gran tamaño más un conjunto de vectores con datos, en un corto periodo de tiempo la ejecución de este se acota a una pequeña porción del programa y al acceso a unos pocos vectores. Si acceso $a[i]$ en tiempo t es muy probable que accese $a[i+1]$ en $t+1$.

Por lo tanto, sería un desperdicio agregar todo el programa cuando solo unas pocas porciones serán usadas. Este supuesto se basa en el **principio de localidad**, el cual dice que en un corto periodo de tiempo, solamente se utilizará una porción del programa.

Cuando una porción necesita ser cargada en memoria se realiza un **swap** con un programa que esté cargado, colocando la nueva porción del programa en memoria. Sin embargo, ¿qué sucedería si una porción de programa que fue expulsada de la memoria necesita ser usada de nuevo? La porción expulsada debe ser reincorporada inmediatamente. El abuso del *swap-in* y *swap-out* es conocido como **thrashing**, y sucede cuando el SO se pasa la mayor parte del tiempo en esta actividad en vez de la ejecución de los programas.

5.4. Memoria virtual con paginación

Al igual que la paginación simple, aquí cada proceso tiene una tabla de páginas, sin embargo, en este caso se considera un **bit de precedencia** (P) que indica la existencia de la página en memoria principal. Además, la entrada de la tabla de página incluye un **bit de modificación** (M) que indica si la página ha sido modificada desde que se cargó. Esto se muestra en la Figura 5.

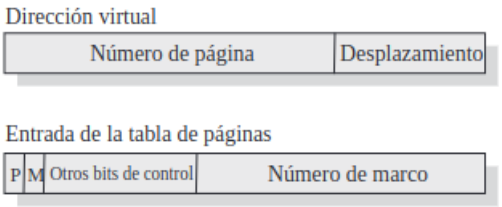


Figura 5: Paginación con MV.

El sistema de traducciones de memoria usando paginación se explica en la Figura 6. Si se considera una dirección de $n+d$ bits, los n bits representan el número de página, al indexar ese valor en la tabla obtenemos el número de marco de página m bits.

Dado que cada proceso puede utilizar una gran cantidad de memoria virtual, las tablas de páginas se guardan también en la memoria virtual, por lo que están sujetas a **paginación** como cualquier página. Para organizar estas tablas de gran tamaño, algunos sistemas utilizan un **esquema de dos niveles**, el cual se muestra en las Figuras 7 y 8.

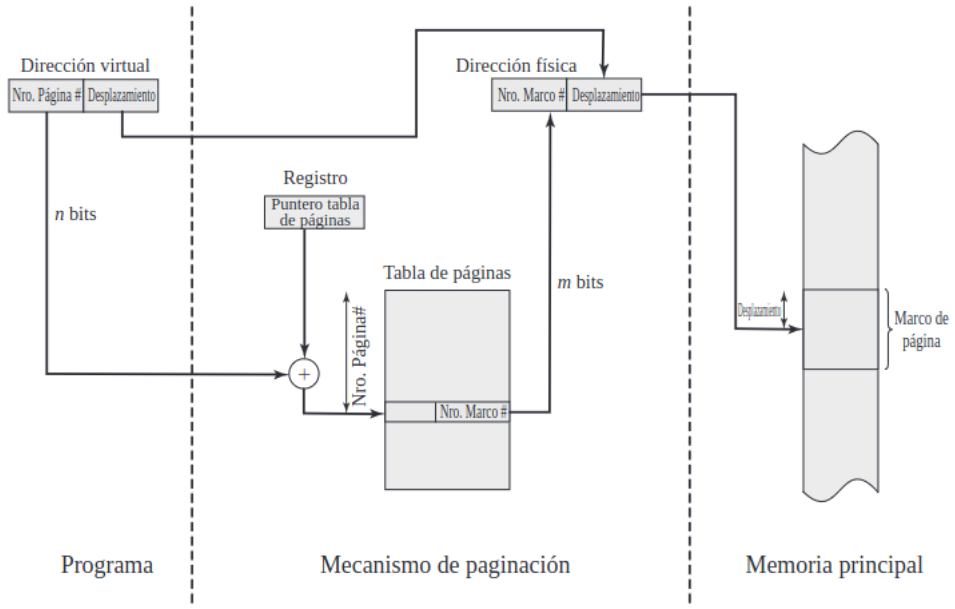


Figura 6: Traducción de direcciones mediante paginación.

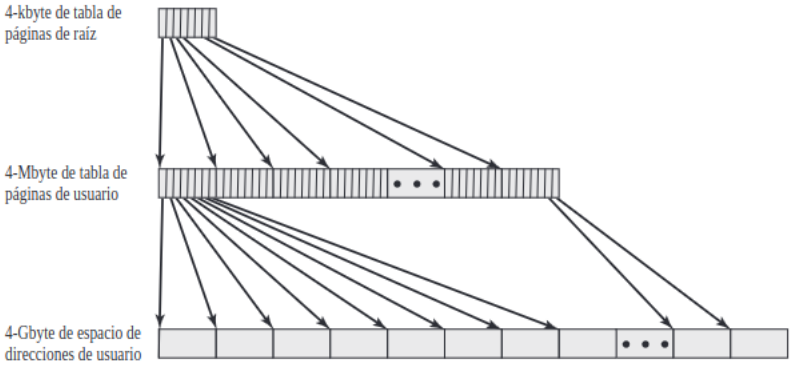


Figura 7: Traducción de direcciones mediante paginación en un sistema de dos niveles.

5.4.1. Tabla de páginas invertida

Una desventaja del tipo de tablas de páginas que hemos visto es que su tamaño es proporcional al espacio de direcciones virtuales. En el esquema de tabla invertida de páginas el tamaño es proporcional al tamaño del direccionamiento físico, y es independiente del número de procesos o páginas virtuales. El número de página es mapeado sobre la tabla invertida por una función de **hashing**.

La Figura 9 muestra una implementación típica de la técnica de tabla de páginas invertida. Para un tamaño de memoria física de 2^m marcos, la tabla de páginas invertida contiene 2^m entradas, de forma que la entrada en la posición i-esima se refiere al marco i. La entrada en la tabla de páginas incluye la siguiente información:

- **Número de página**

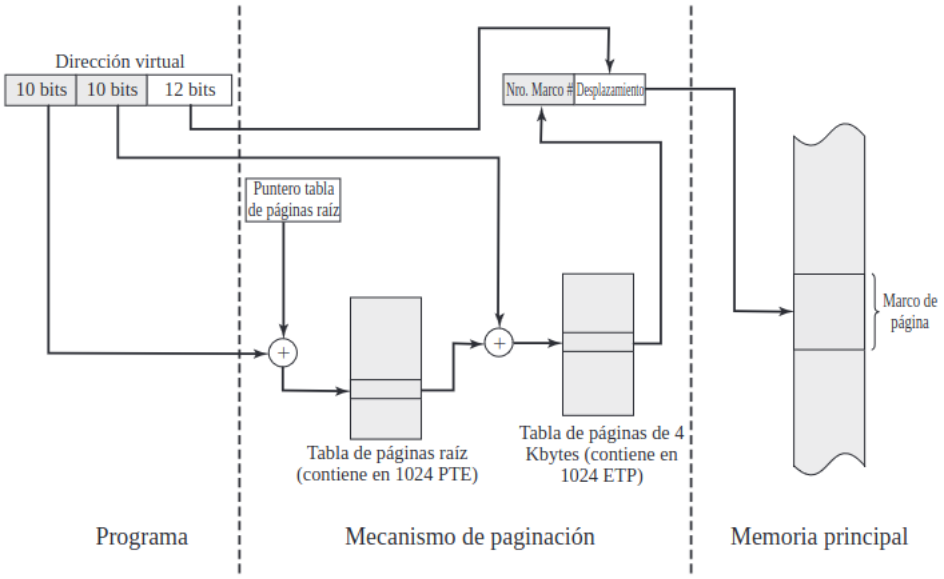


Figura 8: Tabla de páginas jerárquicas de dos niveles.

- **PID**
- **Bits de control**
- **Puntero de la cadena** de páginas que mapean a la misma posición.

5.4.2. Buffer de traducción anticipada (TLB)

En principio, toda referencia a la memoria virtual puede causar dos accesos a memoria física: uno para buscar la entrada a tabla de páginas apropiada y otro para buscar los datos solicitados. Para solucionar este problema, se usa una memoria cache de alta velocidad que almacena una parte pequeña de la tabla de páginas, habitualmente denominada buffer de traducción anticipada (*translation lookaside buffer*).

La Figura 10 muestra un esquema de implementación de una TLB dada una dirección virtual, el cual puede ser explicado mediante tres casos:

- Dada una dirección virtual el procesador primero examina la TLB, si la entrada de la tabla de páginas se encuentra, entonces se puede construir la dirección real directamente.
- Si la entrada de la tabla de páginas no se encuentra (fallo en la TLB), el procesador utiliza el número de página para indexar en la tabla de páginas y si el bit de precedencia (P) es igual a 1, entonces se puede recuperar el número del marco para construir la dirección física.
- Si en el caso anterior el bit de precedencia no es igual a 1, ocurre un **page fault** entonces recurrimos al SO para agregar la página a la tabla de páginas.

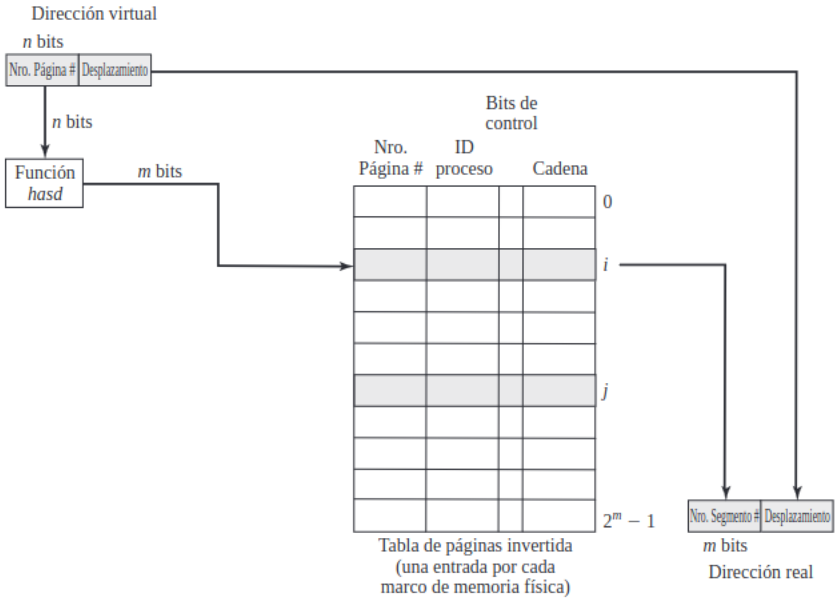


Figura 9: Estructura de tabla de página invertida.

Debido a que la TLB sólo contiene algunas de las entradas de toda la tabla de páginas, no es posible indexar simplemente la TLB por medio de número página. En lugar de eso, cada entrada de la TLB debe incluir un número de página así como la entrada de la tabla de páginas completa. El procesador proporciona un hardware que permite consultar simultáneamente varias entradas para determinar si hay una conciencia sobre un número de página. Esta técnica se denomina resolución asociativa (asociative mapping) que contrasta con la resolución directa, o indexación, utilizada para buscar en la tabla de páginas en la Figura 11.

5.4.3. Tamaño de la página

Una decisión de diseño hardware importante es el tamaño de página a usar. Hay varios factores a considerar:

- **Fragmentación interna.** Evidentemente, cuanto mayor es el tamaño de la página, menor cantidad de fragmentación interna. Para optimizar el uso de la memoria principal, sería beneficioso reducir la fragmentación interna.
- Cuanto menor es la página, mayor número de páginas son necesarias para cada proceso. Un mayor número de páginas por proceso significa también mayores tablas de páginas. Para programas grandes esto significa que gran parte de la tabla se encuentra en memoria virtual.

Además de lo anterior hay que considerar la existencia de la memoria secundaria, la cual favorece la transferencia de grandes bloques de datos, por lo que se podría pensar que es favorable tener páginas más grades. A pesar de esto, números de páginas pequeño disminuyen la probabilidad de fallos de página. A medida que el proceso se ejecuta, las páginas

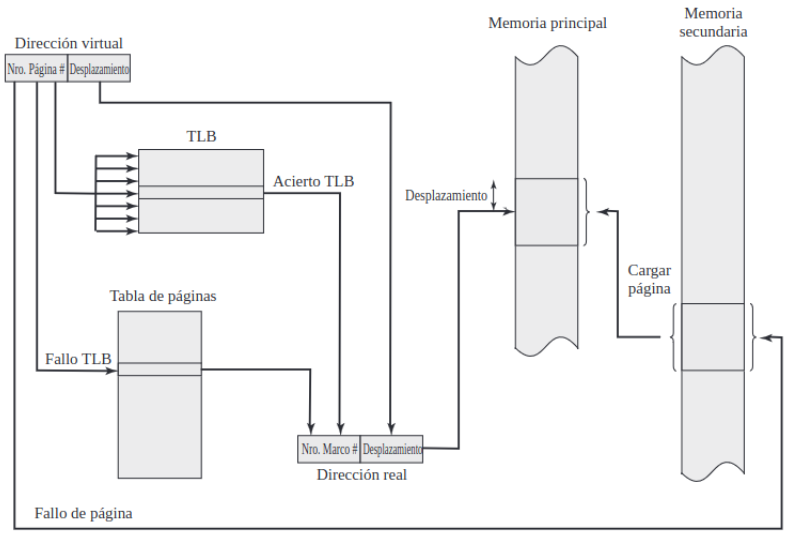


Figura 10: Uso de la TLB.

en memoria tendrán porciones del proceso cercanos a referencias recientes, por lo que páginas muy grandes pierden la localidad temporal y espacial, aumentando la posibilidad de **page fault**. Sin embargo, cuando el tamaño de página se acerca al tamaño del proceso completo, disminuye la probabilidad de tener fallo de página. Este comportamiento se explica en la Figura 12.

5.5. Memoria virtual con segmentación

La segmentación permite al programador ver la memoria como si se tratase de diferentes espacios de direcciones o segmentos de tamaño dinámico. Una referencia a la memoria consiste en un formato de dirección del tipo (número de segmento, desplazamiento). Este tipo de organización tiene las siguientes ventajas:

- Simplifica el tratamiento de estructuras dinámicas.
- Permite que los programas sean modificados y recompilados independientemente.
- **Da soporte a la compartición entre procesos.** El programador puede situar un programa de utilidad o una tabla de datos que resulte útil en un segmento al que pueda hacerse referencia desde otros procesos.
- **Soporta los mecanismos de protección.** Esto es debido a que un segmento puede definirse para contener un conjunto de programas o datos bien descritos, el programador o el administrador de sistemas puede asignar privilegios de acceso de una forma apropiada.

Al igual que en segmentación simple, las direcciones virtuales de memoria aquí se componen de un **número de segmento** y un **offset** tal y como se muestra en la Figura 14. Cada entrada de la tabla de segmentos contiene la dirección de comienzo del correspondiente

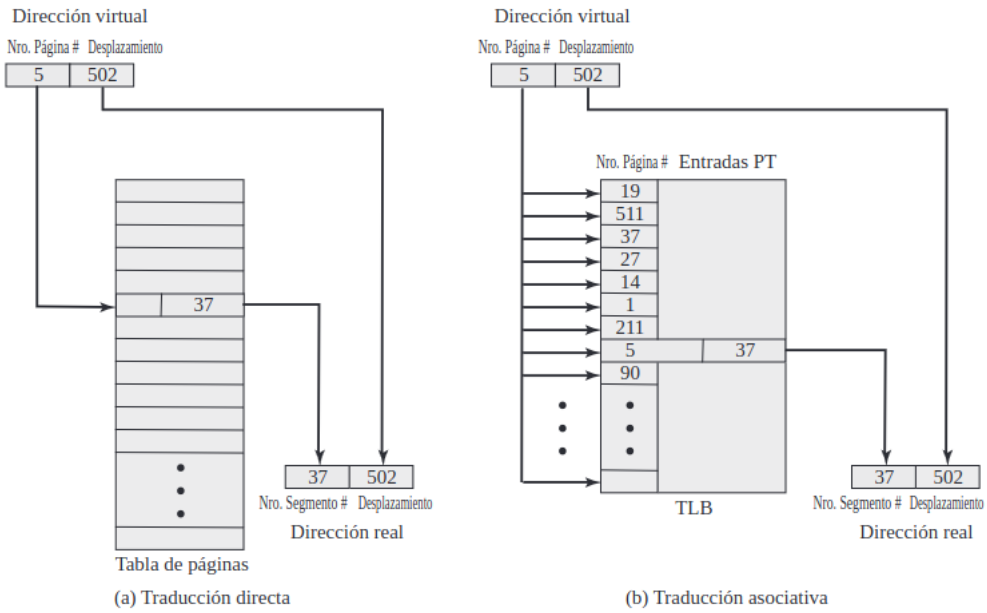


Figura 11: Resolución directa vs. asociativa para las entradas en la tabla de páginas.

segmento en la memoria principal, así como la longitud del mismo. Además se agrega un **bit de precedencia** (P) y un **bit de modificación** (M).

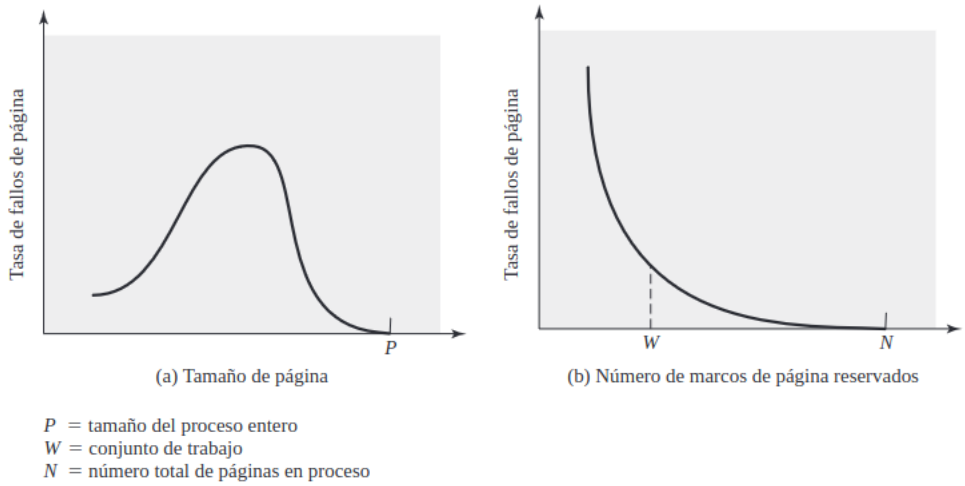


Figura 12: Comportamiento típico de la paginación de un programa.

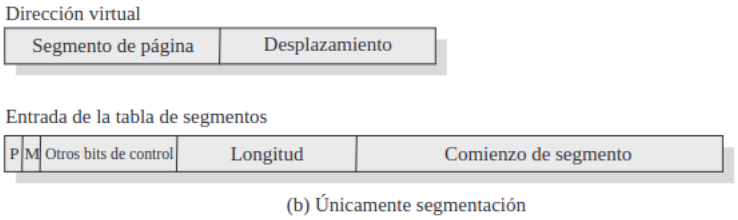


Figura 13: Memoria virtual basada en segmentación.

6. Bibliografía y referencias

1. Stallings W. (2005). *Sistemas Operativos. Aspectos internos y aspectos de diseño.* (5ta ed.). Pearson.

2. Modelo GPT-3.5, OpenAI (2021). ChatGPT. [Software informático]. Disponible en https://openai.com

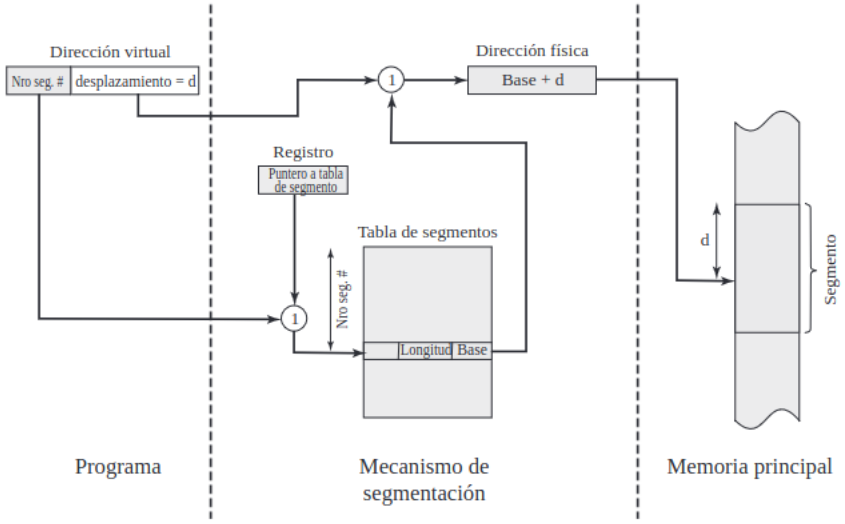


Figura 14: Traducción de dirección virtual a física basada en segmentación.