# SISTEMAS OPERATIVOS

## UNIDAD 6: ADMINISTRACIÓN DE LA MEMORIA

**Licenciatura en Ciencias de la Computación**
**Licenciatura en Sistemas de Información**
**Tecnicatura Universitaria en Programación Web**

**Departamento de Informática**

**Facultad de Ciencias Exactas, Físicas y Naturales**

**Universidad Nacional de San Juan**

## Esquema de contenidos



# 1. INTRODUCCIÓN

El único espacio de almacenamiento que el procesador puede utilizar directamente, aparte de los registros (si bien son internos y sumamente rápidos, aunque de capacidad demasiado limitada) es la memoria RAM. Todas las arquitecturas de procesador tienen instrucciones para interactuar con la memoria, pero ninguna lo tiene para hacerlo con

medios masivos de almacenamiento, como las unidades de disco. Cuando se hace referencia al almacenamiento principal, siempre se referirá a la memoria RAM, mientras que el almacenamiento secundario se refiere a los discos u otros medios de almacenamiento permanente.
Otro elemento a considerar es la protección, ya que varios procesos se cargan simultáneamente en la memoria principal.
Todos los programas a ejecutar deben cargarse en la memoria principal (RAM) del sistema antes de ser utilizados. El SO administra la memoria para permitir que varios procesos la compartan lo que implica que se deben implementar mecanismos de protección, ya que, cualquier proceso podría modificar la memoria del resto de procesos e incluso del sistema operativo.
Actualmente, casi todos los sistemas operativos emplean implementaciones que requieren de hardware especializado: la Unidad de Manejo de Memoria (MMU, por sus siglas en inglés).
En esta unidad se estudiarán las siguientes técnicas de administración de memoria:

- Asignación Particionada Fija
- Asignación Particionada Dinámica
- Segmentación
- Paginación Simple
- Paginación por/bajo demanda

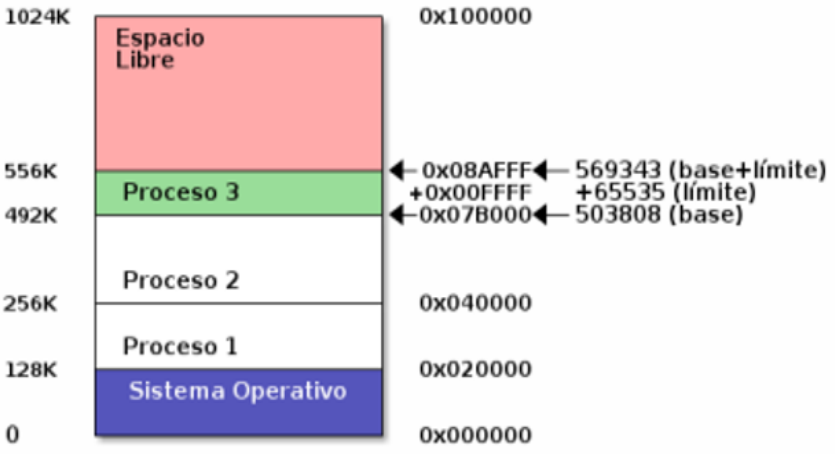
## 1.1. Espacio de direccionamiento

Este tema se estudió en la asignatura Estructura y Funcionamiento de Computadoras.
A modo de repaso se dirá que la memoria está estructurada como un arreglo direccionable de bytes. Al solicitar el contenido de una dirección específica de memoria, el hardware entregará un byte (8 bits), y no menos. En la mayoría de las arquitecturas el tamaño de palabra es mayor.
Un procesador que soporta un espacio de direccionamiento de 16 bits puede referirse (direccionar) hasta $2^{16}$ bytes, o sea, hasta 65536 bytes (64 KB). Los procesadores con arquitectura de 32 bits pueden referenciar desde el punto de vista teórico hasta $2^{32}$ bytes (4 GB) de RAM. No obstante, debido a las limitaciones tecnológicas (y tal vez estratégicas) para producir memorias con esta capacidad, pasaron años para que las memorias alcanzaran esa capacidad. Hoy en día los procesadores dominantes son de 32 o 64 bits. Un procesador de 64 bits podría direccionar hasta 16 Exabytes.

## 1.2. Hardware

### 1.2.1. La unidad de manejo de memoria (MMU)

Con la introducción de sistemas multitarea, es decir, dos o más programas ejecutándose alternativamente, se vio la necesidad de tener más de un programa cargado en memoria. Esto conlleva que el SO junto con la información del programa a ejecutar debe resolver cómo ubicar los programas en la memoria física disponible.
Con el propósito de ofrecer a los procesos más espacio de lo que puede direccionar el hardware empleado, ha resultado necesario *emplear más memoria de la que está realmente disponible*. La MMU emplea mecanismos para conseguir un **espacio virtualmente ilimitado** para realizar sus operaciones incluso cuando la memoria real es mucho menor a la solicitada y la ilusión de tener un bloque contiguo e ininterrumpido de memoria.
La MMU se encarga de brindar la **protección** necesaria verificando que un proceso no tenga acceso a leer o modificar los datos de otro.
Una primera aproximación a la protección de acceso se implementa usando un registro base y un registro límite: si la arquitectura ofrece dos registros del procesador que sólo pueden ser modificados por el SO (el hardware define la modificación de dichos registros como una operación privilegiada que requiere estar ejecutando en *modo supervisor)* la MMU puede comparar cada acceso a memoria para verificar que esté en el rango permitido. Por ejemplo, si a un proceso le fue asignado un espacio de memoria de 64 KB (65535 bytes) a partir de la dirección 503808 (492 KB), el **registro base** contendría 503808, y el **registro límite** 65535. Si hubiera una instrucción por parte de dicho proceso que solicitara una dirección menor a 503808 o mayor a 569343 (556 KB), la MMU lanzaría una excepción interrumpiendo el procesamiento, e indicando al SO que ocurrió una violación de segmento. El SO entonces procedería a terminar la ejecución del proceso, reclamando todos los recursos que tuviera asignados y notificando al usuario.

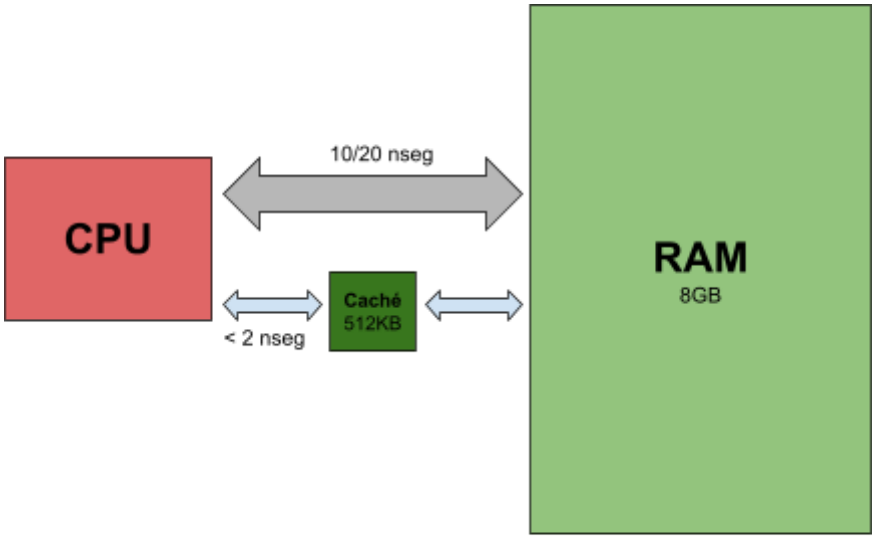



Espacio de direcciones válidas para el proceso 3 definido por un registro base y uno límite

### 1.2.2. La memoria caché

Conforme el procesador avanza en la ejecución de las instrucciones (aumentando el valor almacenado en el registro CP) se producen accesos a memoria. Tiene que buscar en memoria la siguiente instrucción a ejecutar. Las instrucciones pueden requerir uno o más operadores adicionales que deben ser leídos de la memoria. Además, la instrucción puede necesitar guardar un resultado en memoria. Hace años esto no era un problema, la velocidad del procesador estaba básicamente sincronizada con la del manejador de memoria, y el flujo podía mantenerse básicamente estable. Pero conforme los procesadores se fueron haciendo más rápidos, y conforme se ha popularizado el procesamiento en paralelo, la tecnología de las memorias no ha progresado a la misma velocidad.

Entonces, cuando el procesador solicita el contenido de una dirección de memoria y ésta no está aún disponible, tiene que detener su ejecución hasta que los datos estén disponibles, es decir, es tiempo perdido.




Compare la Capacidad y Velocidad entre la Memoria Caché (L1) y la memoria RAM

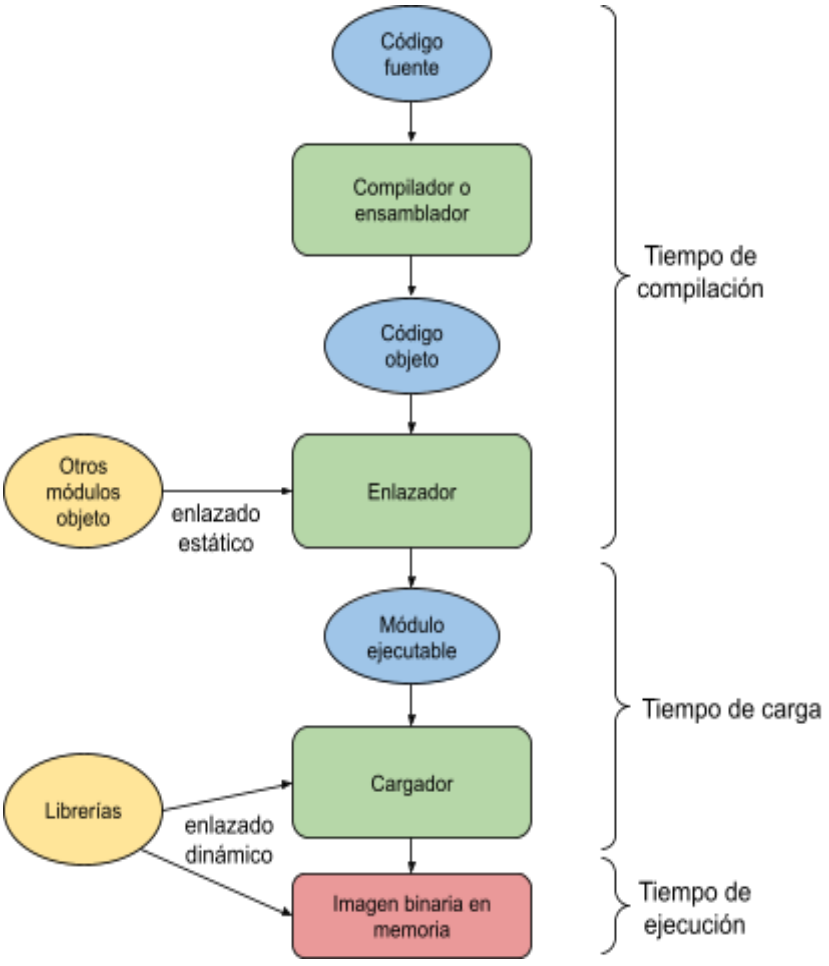
La respuesta para reducir esa espera es la memoria caché, que es una memoria de alta velocidad, situada entre la memoria principal y el procesador, que guarda información que el procesador debería necesitar en el futuro partiendo del principio de la *localidad* de referencia:

- **Localidad temporal**: Es probable que un recurso que fue empleado recientemente vuelva a emplearse en un futuro cercano.
- **Localidad secuencial**: Un recurso, y muy particularmente la memoria, tiende a ser requerido de forma secuencial.

Aplicando el concepto de localidad de referencia, cuando el procesador solicita al hardware determinada dirección de memoria, el hardware no sólo transfiere a la memoria caché el byte o palabra solicitada, sino que transfiere un bloque completo.

## 1.3. Resolución de direcciones

Un programa de usuario debe pasar por diferentes etapas antes de ser ejecutado, aunque algunas de ellas son opcionales, tal como se muestra en la figura.
El objetivo de esta figura, no es profundizar en lo que le sucede al programa en cada etapa hasta llegar a la imagen que se cargará en la memoria para su ejecución, sino mostrar que el programa soporta distintas acciones que lo van transformando hasta alcanzar su estado final (imagen binaria).

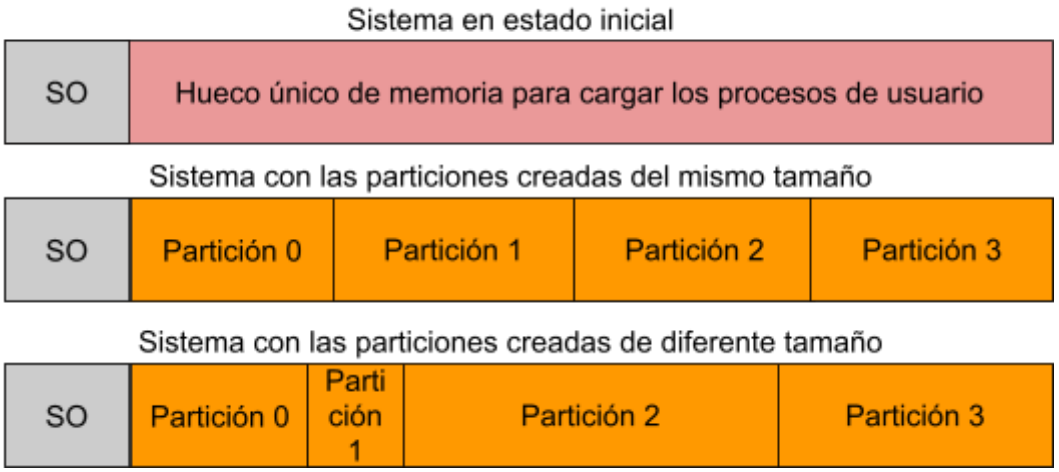



La mayor parte de los sistemas permiten que un proceso de usuario resida en cualquier parte de la memoria física. Así, aunque el espacio de direcciones del sistema comience en 0x00000000, la primera dirección del proceso de usuario no tiene por qué ser esa.

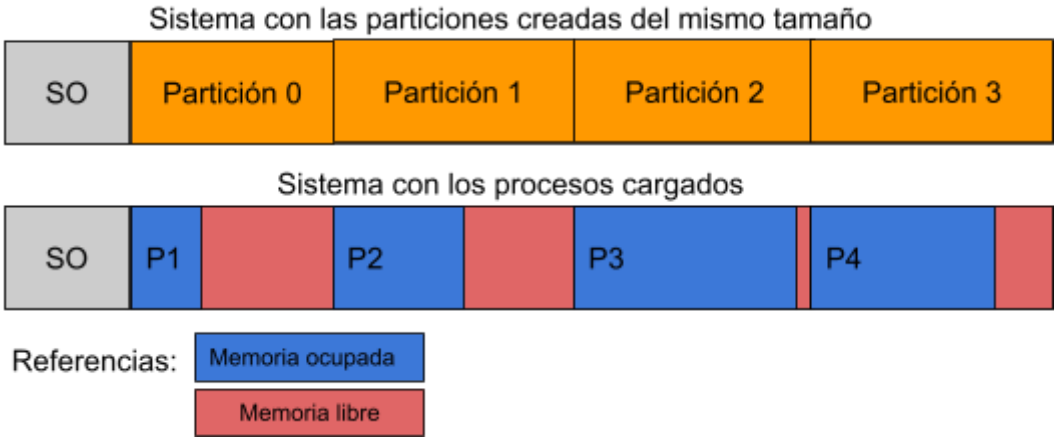
En cada una de las etapas vistas en la figura, las direcciones pueden representarse de formas distintas, por lo que en cada paso es necesario reasignar las direcciones usadas en una etapa en direcciones de la siguiente.
Por ejemplo, en el código fuente de un programa las direcciones son generalmente simbólicas, como los nombres de las variables y las funciones. A continuación, un compilador suele reasignar esas direcciones simbólicas en direcciones reubicables, como por ejemplo “140 bytes desde el comienzo del módulo”. Al final del proceso y para que el programa pueda ser ejecutado, esas direcciones reubicables se convierten en direcciones absolutas de la memoria, como 0x00123123.

# 2. ASIGNACIÓN PARTICIONADA FIJA

Si bien las técnicas de administración de la memoria particionadas (Fija y Dinámica) están prácticamente en desuso, se estudian porque permiten adquirir conceptos básicos necesarios para el entendimiento en profundidad de la administración de la memoria.
Algunos de estos conceptos elementales son: contigüidad, totalidad, estrategias de asignación de los espacios libres de memoria y fragmentación, tanto interna como externa.
Cuando el sistema se inicia, el SO se carga típicamente en la región baja de la memoria (desde la dirección en memoria 0x00000000 hacia arriba), y una vez terminado el espacio necesario para el núcleo y sus estructuras, el SO divide la memoria en espacios fijos (del mismo tamaño o no) que no podrán ser modificados hasta el próximo reinicio del sistema, tal como lo muestra la figura.



En un sistema en el cual las particiones son todas iguales, los procesos se cargan en las distintas particiones de la memoria, con la única condición que el proceso quepa en ese espacio de memoria.
Note en la figura que, una vez que el proceso es cargado en la partición, hay memoria que se desperdicia (fragmentación), ya que es muy improbable que partición y proceso tengan el mismo tamaño.




En el caso de que las particiones tengan distintos tamaños, el SO deberá recorrer la memoria para determinar en cuál de ellas cabe el proceso.
El hardware que debe tener un sistema que soporte esta técnica, debe poder delimitar el bloque asignado a cada uno de los procesos. Por ejemplo, con un *registro base* y un *registro límite*, esto sería suficiente para alojar en memoria varios procesos y evitar que interfieran entre sí.
Ventajas:

- Fácil de implementar
- Poca sobrecarga para el SO ya que la asignación se hace al comienzo

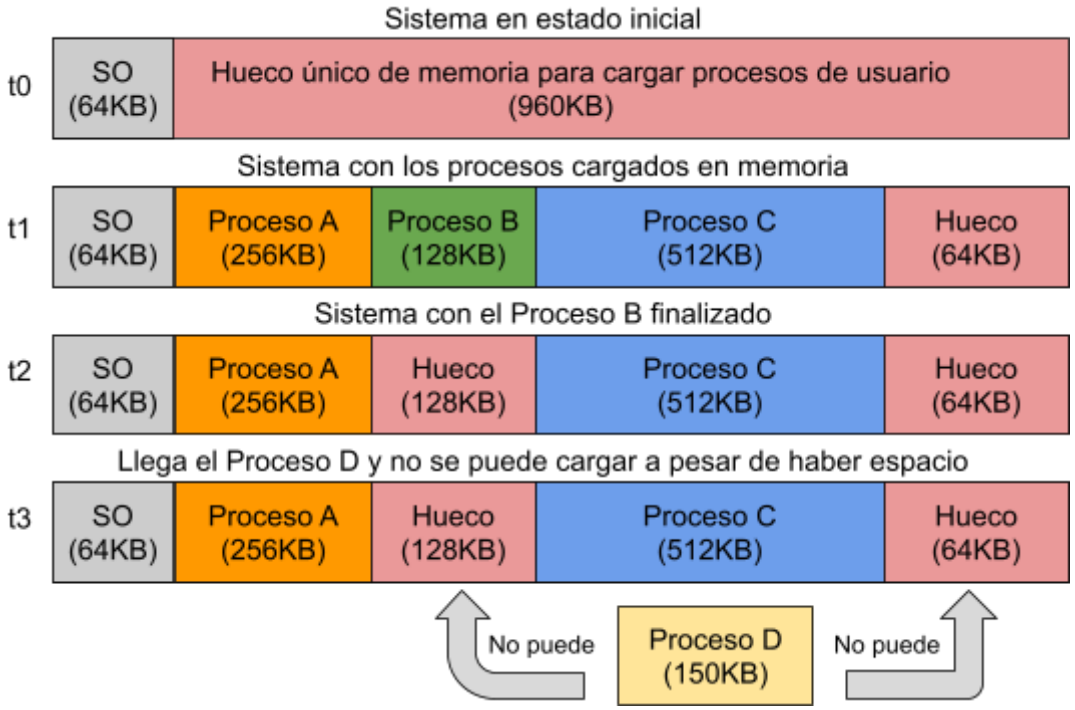
Desventajas:

- Tamaño máximo del proceso limitado al tamaño de la partición
- Número máximo de procesos en memoria determinado por la cantidad de particiones
- Desperdicio de memoria (Fragmentación interna)

# 3. ASIGNACIÓN PARTICIONADA DINÁMICA

Cuando el sistema se inicia, el SO se carga típicamente en la región baja de la memoria y una vez terminado el espacio necesario para el núcleo y sus estructuras, el SO inicia la carga de los procesos asignando a cada uno un tamaño adecuado de memoria para que se cargue. Esto significa que en la medida que se van cargando los

procesos se crean las particiones de memoria de manera dinámica. Por lo tanto, las particiones son variables en tamaño y en número.
Cuando un proceso termina, libera el bloque y se carga un nuevo proceso cuyo tamaño muy probablemente no sea igual al proceso terminado. Entonces el SO divide ese bloque en 2 bloques de menor tamaño. En una partición carga el nuevo proceso, con el tamaño adecuado para ese proceso. Y en la otra partición se carga otro proceso.
Como se puede ver en la medida que el sistema corre se van dividiendo las particiones en particiones de menor tamaño, lo cual es ineficiente ya que cada vez menos procesos se pueden cargar al ser particiones chicas, tal como muestra la figura.
Esto lleva a que frecuentemente el sistema deba ser compactado.




Ventajas:

- Desperdicio de memoria (Fragmentación interna)

Desventajas:

- Desperdicio de memoria (Fragmentación externa). Bloques cada vez más pequeños
- Se debe hacer compactación

## 3.1. Técnicas de asignación de espacio libre

A medida que los procesos finalizan, aparecen regiones de memoria disponible (t1 y t2), interrumpidas por regiones de memoria usadas por los procesos que aún se encuentran activos (ver figura). Al crear un nuevo proceso, el SO tiene tres estrategias según las cuales podría asignarle uno de los bloques disponibles:

### 3.1.1. Primer ajuste

El sistema toma el primer bloque con el tamaño suficiente para alojar el nuevo proceso. Este es el mecanismo más simple de implementar y el de más rápida ejecución. No obstante, esta estrategia causa el desperdicio de memoria si el bloque no es exactamente del tamaño requerido.

### 3.1.2. Mejor ajuste

El sistema busca entre todos los bloques disponibles cuál es el que mejor se ajusta al tamaño requerido por el nuevo proceso. Esto implica la revisión completa de la lista de bloques (más tiempo). Como desventaja, al ser tan justa la partición seleccionada, la división del bloque dejará un bloque muy pequeño que probablemente no permita cargar otro proceso.

### 3.1.3. Peor ajuste

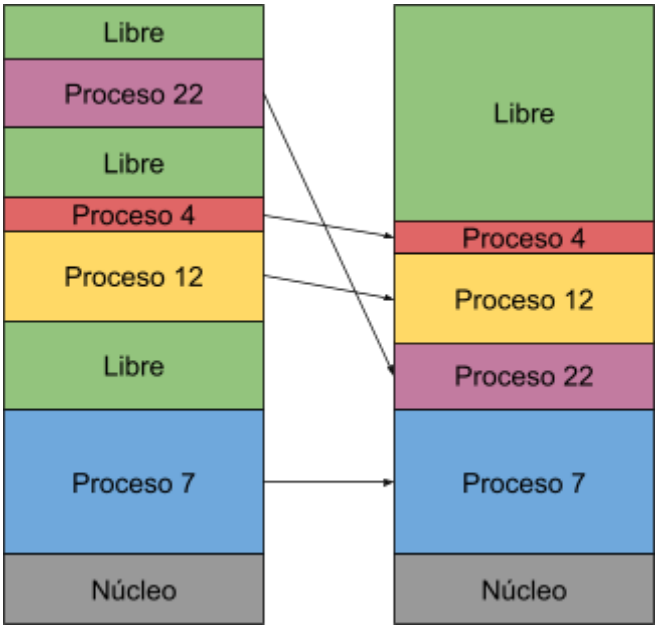
El sistema busca cuál es el bloque más grande disponible, y se lo asigna al nuevo proceso. Con este mecanismo se busca que los bloques que queden después de otorgarlos a un proceso sean tan grandes como sea posible, de cierto modo balanceando su tamaño.

## 3.2. El problema de la fragmentación

Se entiende por fragmentación de la memoria a espacios de memoria dispersos por la memoria (también llamados huecos) y que **no pueden ser utilizados** por el método de asignación de la memoria a los procesos. La fragmentación puede ser *interna* o *externa*.
Dependiendo del método de asignación (distribución) de la memoria se producirá algún tipo de fragmentación (desperdicio).
La **fragmentación externa** se produce cuando hay varios bloques libres dispersos entre bloques asignados a procesos y, si bien la sumatoria de esos huecos permitiría cargar un nuevo proceso en memoria, al estar dispersos no se puede y esos espacios de memoria siguen vacíos. Ver en la figura lo que sucede en t3.
La **fragmentación interna** se refiere a la cantidad de memoria dentro de un bloque que nunca se usará, por ejemplo, si el SO maneja bloques de 512 bytes y un proceso requiere sólo 768 bytes para su ejecución, el sistema le entregará dos bloques (1024 bytes), produciendo un desperdicio de 256 bytes.

## 3.3. Compactación

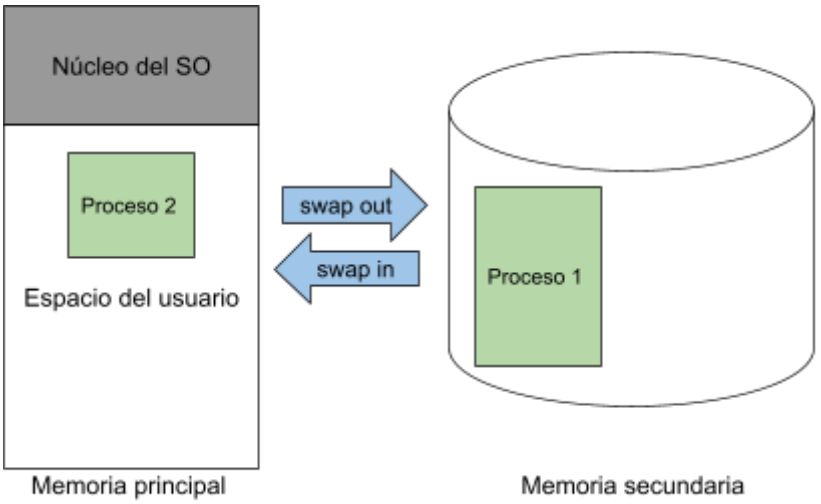
Un problema importante que surge como resultado de la fragmentación es que el espacio total libre de memoria puede ser mucho mayor que lo que requiere un nuevo proceso, pero al estar fragmentada en varios bloques, éste no encontrará una partición contigua donde ser cargado.
Una solución consiste en hacer una operación de compresión o compactación. Esta operación consiste en mover los contenidos en memoria de los bloques asignados para que ocupen espacios contiguos, permitiendo unificar varios bloques libres contiguos en uno solo. La compactación tiene un costo alto ya que involucra mover prácticamente la totalidad de la memoria (probablemente más de una vez por bloque).




## 3.4. Intercambio (swapping) con el almacenamiento secundario

Siguiendo de cierto modo la lógica requerida por la compactación se encuentran los sistemas que utilizan intercambio (swapping) entre la memoria primaria y secundaria. En éstos, el SO puede comprometer más espacio de memoria del que tiene físicamente disponible. Cuando la memoria se agota, el sistema suspende un proceso (normalmente un proceso bloqueado) y almacena una copia de su imagen en memoria en almacenamiento secundario para poder restaurarlo en el futuro.
Hay algunas restricciones que observar previo a suspender un proceso. Por ejemplo, se debe considerar si el proceso tiene pendiente alguna operación de entrada/salida, en la cual el resultado se deberá copiar en su espacio de memoria. Si el proceso resultara suspendido (es retirado de la memoria principal), el SO no tendría dónde continuar almacenando estos datos conforme llegan. Una estrategia ante esta situación podría ser

que todas las operaciones se realicen únicamente a buffers (regiones de memoria de almacenamiento temporal) en el espacio del SO, y éste transfiera el contenido del buffer al espacio de memoria del proceso suspendido una vez que la operación finalice.



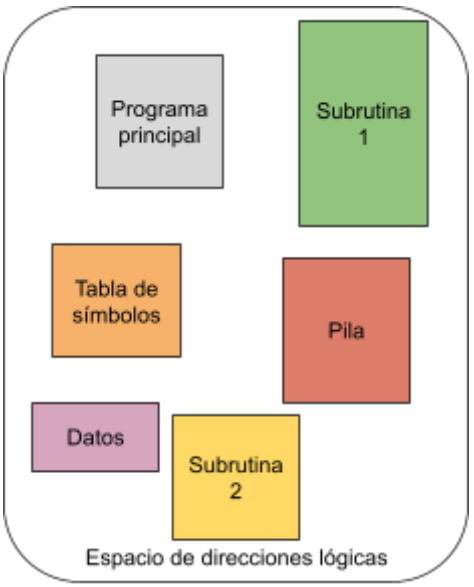

Se debe considerar que las unidades de disco son del orden de decenas de miles de veces más lentas que la memoria, por lo que este proceso resulta muy costoso.

# 4. SEGMENTACIÓN Simple

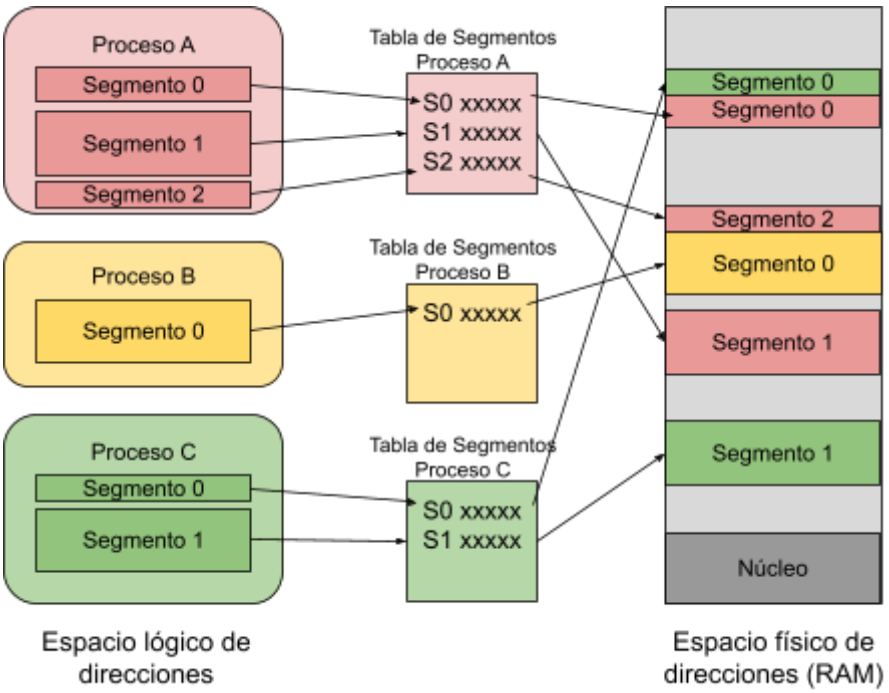
Al desarrollar un programa en un lenguaje de alto nivel, el programador usualmente no se preocupa por la ubicación en la memoria física de los diferentes elementos que lo componen. Esto se debe a que en estos lenguajes las variables y funciones son referenciadas por sus nombres, no por su ubicación. No obstante, cuando se compila el programa para una arquitectura que soporte segmentación, el compilador ubicará a cada una de las secciones en un **segmento** diferente.

Así, los elementos que conforman un programa se organizan en secciones (segmentos): una sección contiene el espacio para las variables globales, otra sección contiene el código compilado, otra sección contiene la tabla de símbolos, etc.

Una de las principales ventajas de la Segmentación es que, para cargar un proceso en la memoria, no es necesario que haya un espacio contiguo de memoria libre, sino que el proceso (a través de sus segmentos) se puede cargar disperso por la memoria. Pero siempre se carga el proceso **completamente**.




Otra ventaja importante consiste en permitir que cada uno de los segmentos tenga un distinto juego de permisos para el proceso en cuestión: el SO puede indicar, por ejemplo, que el segmento de texto (el código del programa) sea de lectura y ejecución, mientras que las secciones de datos, libres y pila (donde se almacena y trabaja la información misma del programa) serán de lectura y escritura, pero la ejecución estará prohibida.

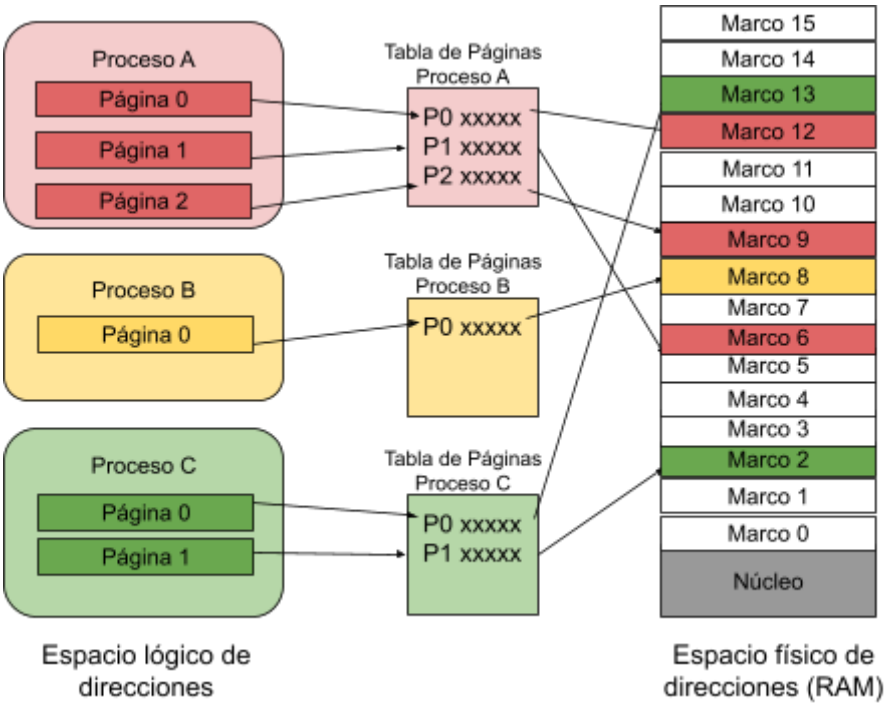



Los segmentos de cada proceso están dispersos por la memoria y la forma que tiene el sistema para hacer el mapeo (mapping) desde el mundo lógico al físico es a través de Tablas de Segmentos. Los campos de esta Tabla de Segmentos son: id de segmento,

dirección física base (de comienzo del segmento) y tamaño del segmento, además de atributos vinculados a la protección.
A partir de una dirección lógica que está formada por un (selector de) segmento y un desplazamiento dentro del segmento, la MMU debe tomar el selector, y accediendo a la tabla de segmentos del proceso, toma la dirección base del segmento y le suma el desplazamiento para obtener una dirección lineal física.
La segmentación soluciona la fragmentación interna (cada segmento tiene el tamaño necesario para almacenar esa porción del proceso). Sin embargo, sigue habiendo fragmentación externa al quedar huecos dispersos en la memoria en los cuales no se pueden cargar otros segmentos.

# 5. PAGINACIÓN Simple

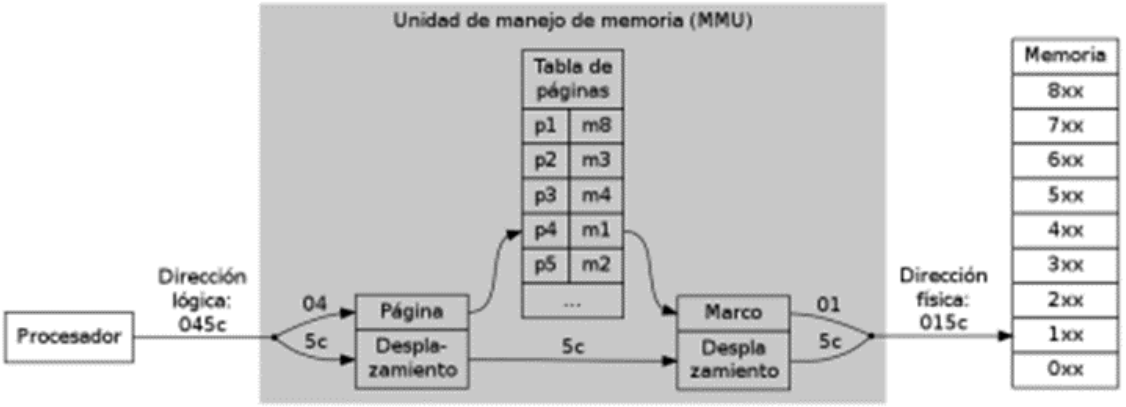
La fragmentación externa y la necesidad de compactación pueden evitarse empleando la paginación. En esta técnica, cada proceso está dividido en varios tramos de tamaño fijo llamados **páginas**, dejando de requerir que la asignación sea de un área contigua de memoria. Esto implica una mayor especialización por parte del hardware, y mayor información relacionada a cada uno de los procesos: no basta sólo con indicar dónde inicia y termina el área de memoria de cada proceso, sino que (al igual que en Segmentación) se debe establecer un mapeo entre la ubicación real (física) y la presentada a cada uno de los procesos (lógica).



La memoria física se divide en **marcos** (frames), todos ellos del mismo tamaño, y el espacio para cada proceso se divide en páginas del mismo tamaño que los marcos, de manera de poder emparejar o asignar (matching) las páginas en los marcos. La MMU se encarga del mapeo entre páginas y marcos mediante *Tablas de páginas*, donde cada proceso tiene su propia tabla de páginas.
Observando el esquema se puede ver similitudes y diferencias entre la Segmentación y la Paginación. A grandes rasgos:

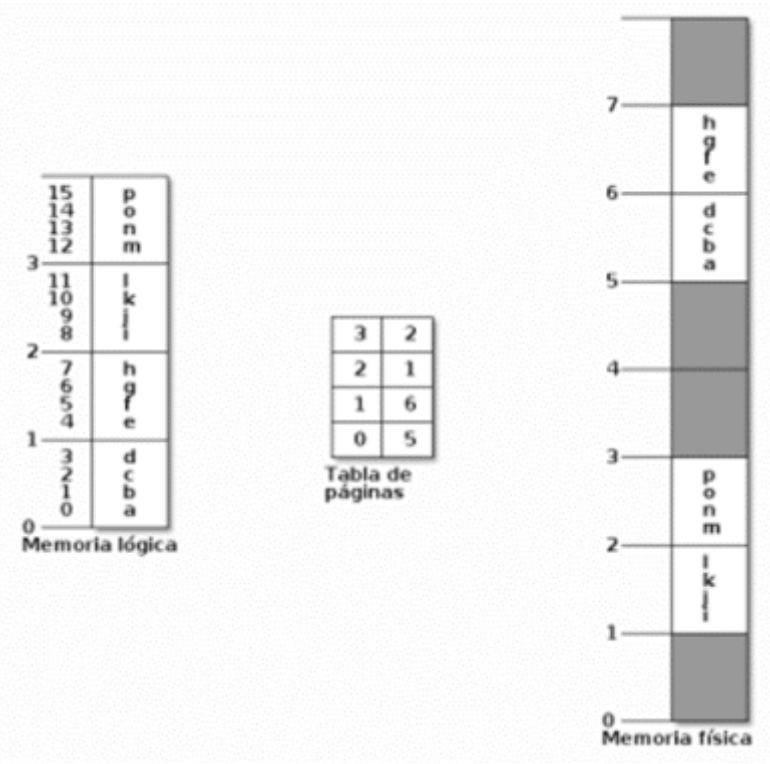
- Ambas técnicas cargan el proceso **completamente** en memoria
- Ambas técnicas necesitan de un elemento de unión (tabla) entre el mundo lógico y el mundo físico
- El tamaño de las tablas utilizadas en la Segmentación es más grande que las tablas de Paginación debido a que la cantidad de información que se almacena en los segmentos es mayor
- Se puede ver que mientras el tamaño de los segmentos varía, en las páginas no (son todas iguales)
- Los bits de cada dirección se separan en un identificador de página y un desplazamiento, de forma similar a lo presentado en Segmentación

El tamaño de los marcos (y, por tanto, las páginas) debe ser una potencia de dos, de modo que la MMU pueda discernir fácilmente la porción de una dirección de memoria que se refiere a la página del desplazamiento. El rango varía entre los 512 bytes y 16 MB; la MMU separa la dirección en memoria entre los primeros m bits (referentes a la página) y los últimos n bits (referentes al desplazamiento).
Para poder realizar este mapeo, la MMU requiere de una estructura de datos denominada Tabla de páginas, que resuelve la relación entre páginas y marcos, convirtiendo una dirección lógica (en el espacio del proceso) en la dirección física (la ubicación en que realmente se encuentra en la memoria del sistema).




Esquema del proceso de paginación, mostrando el rol de la MMU

Se puede tomar como ejemplo para explicar este mecanismo el esquema presentado en la figura a continuación (Silberschatz, Galvin et al). Muestra un esquema pequeño de paginación: un espacio de direccionamiento de 32 bytes (se necesitan cinco bits), organizado en ocho páginas de cuatro bytes cada una (la página es representada con los tres bits más significativos de la dirección, y el desplazamiento con los dos bits menos significativos).
El proceso que se presenta tiene una visión de la memoria como la columna del lado izquierdo: para el proceso hay cuatro páginas, y tiene sus datos distribuidos en orden desde la dirección 00000 (0D) hasta la 01111 (15D). Cuando el proceso quiere referirse a la letra f, lo hace indicando la dirección 00101 (5). De esta dirección, los tres bits más significativos (001) se refieren a la página uno, y los dos bits menos significativos (01) indican al desplazamiento dentro de ésta.
La MMU verifica en la tabla de páginas, y encuentra que la página 1 corresponde al marco número 6 (110), por lo que traduce la **dirección lógica** 00101 (5) a la **dirección física** 11001 (26).

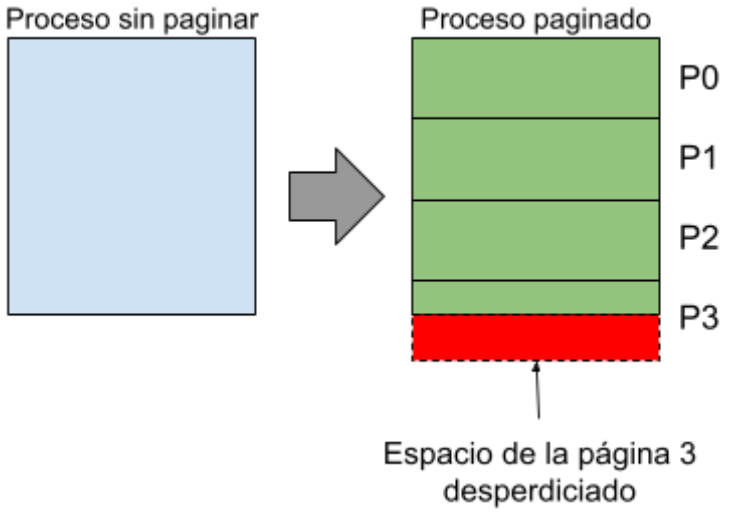



Ej. de paginación, con un espacio de direccionamiento de 32 bytes y páginas de cuatro bytes

## 5.1. Tamaño de la página

Ahora, si bien la fragmentación externa se resuelve al emplear paginación, el problema de la fragmentación interna persiste: al dividir la memoria en bloques de longitud preestablecida de $2^n$ bytes, un proceso en promedio desperdiciará $2^n/2$ (y, en el peor de los casos, hasta $2^n - 1$) bytes. Multiplicando esto por el número de procesos

que están vivos en el sistema y para evitar que una proporción sensible de la memoria se pierda en fragmentación interna, se podría tomar como estrategia emplear un tamaño de página tan pequeño como fuera posible.




Sin embargo, la sobrecarga administrativa (el tamaño de la Tabla de páginas) en que se incurre por gestionar demasiadas páginas pequeñas se vuelve una limitante en sentido opuesto:

- Las transferencias entre unidades de disco y memoria son mucho más eficientes si pueden mantenerse como recorridos continuos. El controlador de disco puede responder a solicitudes de acceso directo a memoria (DMA) siempre que tanto los fragmentos en disco como en memoria sean continuos; fragmentar la memoria demasiado jugaría en contra de la eficiencia de estas solicitudes
- El bloque de control de proceso (BCP) incluye la información de memoria. Entre más páginas tenga un proceso (aunque éstas fueran muy pequeñas), haría más grande su BCP, y más información requerirá intercambiar en un cambio de contexto.

Estas consideraciones opuestas apuntan a que se debe mantener el tamaño de página más grande, y se regulan con las primeras expuestas en esta sección. Actualmente, el tamaño habitual de las páginas es de 4 u 8 KB. Hay algunos SO que soportan múltiples tamaños de página, como por ejemplo Solaris que puede emplear páginas de entre 8 KB y 4 MB, dependiendo del tipo de información que se especifique que se almacenarán.

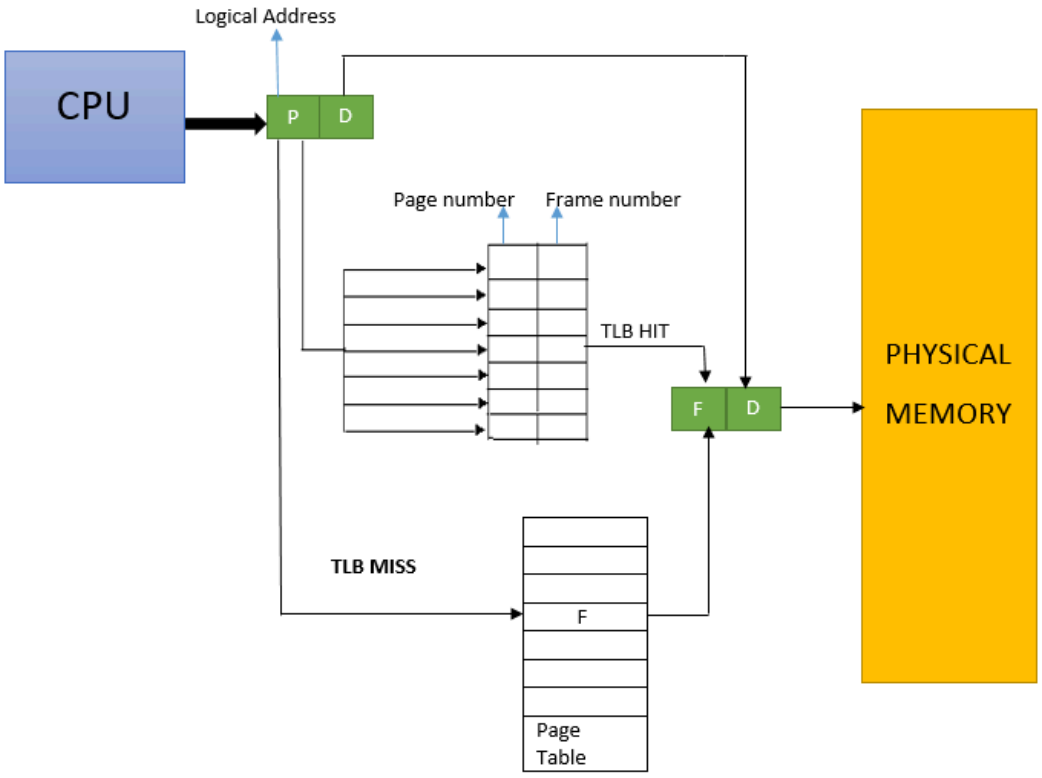
## 5.2. Almacenamiento de la tabla de páginas

Algunos de los primeros sistemas en manejar memoria paginada empleaban un conjunto especial de registros para representar la tabla de páginas. En los equipos actuales, mantener la tabla de páginas en registros resultaría claramente imposible (por

tamaño y costo de los registros): teniendo un procesador de 64 bits, e incluso si se definiera un tamaño de página muy grande (por ejemplo, 4 MB) existirían $2^{42}$ páginas posibles.
Además, nuevamente, cada proceso tiene una tabla de páginas distinta, por lo tanto, se haría necesario hacer una transferencia de información muy grande en cada cambio de contexto.
Una estrategia para enfrentar esta situación es almacenar la propia tabla de páginas en memoria, y apuntar al inicio de la tabla con un juego de registros especiales: el *registro de base de la tabla de páginas* (RBTP) y el *registro de longitud de la tabla de páginas* (RLTP). De esta manera, en el cambio de contexto sólo hay que cambiar estos dos registros, y además se cuenta con un espacio muy amplio para guardar las tablas de páginas que se necesiten. El problema con este mecanismo es la velocidad: Se estaría penalizando a cada acceso a memoria con uno adicional, ya que, si para resolver una dirección lógica a su correspondiente dirección física hace falta consultar la tabla de páginas en memoria, el tiempo efectivo de acceso a memoria se duplica.

## 5.3. El buffer de traducción adelantada (TLB)

La salida obvia a este dilema es el uso de una memoria caché. Sin embargo, más que una caché genérica, la MMU utiliza una caché especializada en el tipo de información que maneja: el *buffer de traducción adelantada o anticipada*. (**T**ranslation **L**ookaside **B**uffer). El TLB es una tabla asociativa en memoria de alta velocidad, un conjunto de registros residentes dentro de la MMU, donde las claves son las páginas y los valores son los marcos correspondientes. De este modo, las búsquedas se efectúan en tiempo constante.
El TLB típicamente tiene entre 64 y 1024 entradas. Cuando el procesador efectúa un acceso a memoria, si la página solicitada está en el TLB, la MMU tiene la dirección física de inmediato. En caso de no encontrarse la página en el TLB, la MMU lanza un fallo de página (page fault), con lo cual consulta cuál es el marco correspondiente de la memoria principal. Esta nueva entrada es agregada al TLB, la probabilidad de que las regiones más empleadas de la memoria durante un área específica de ejecución del programa sean cubiertas por relativamente pocas entradas del TLB son muy altas.




Esquema de paginación empleando un buffer de traducción adelantada

Por lo expuesto, se puede concluir que en sistemas con páginas pequeñas hay:

- ➢ Menos fragmentación interna
- ➢ Más páginas para el proceso
- ➢ Muchas páginas por proceso
- ➢ La tabla de páginas crecerá el tamaño
- ➢ Se necesita más memoria para cargar la tabla
- ➢ Los fallos de página se reducen

En cambio, en sistemas que empleen paginación con páginas grandes hay:

- ➢ Más fragmentación interna
- ➢ Cada página contiene más porciones del proceso
- ➢ Se ocupa memoria innecesariamente
- ➢ Los fallos de página se incrementan

## 5.4. Memoria compartida

Hay muchos escenarios en que diferentes procesos pueden beneficiarse de compartir áreas de su memoria. Un caso frecuente es compartir código.

Si un mismo programa es ejecutado varias veces, y dicho programa no emplea mecanismos de código auto-modificable, no tiene sentido que las páginas en que se representa cada una de dichas instancias ocupe un marco independiente (el SO puede

asignar a páginas de diversos procesos el mismo conjunto de marcos, con lo cual puede aumentar la capacidad percibida de memoria).


Uso de memoria compartida: tres procesos comparten la memoria ocupada por el texto del programa; difieren sólo en los datos

En este modelo, el SO debe garantizar que las páginas (también aplica para segmentos) correspondientes al código del programa sean de sólo lectura. Un programa que está desarrollado y compilado de forma tal que permita que todo su código sea de sólo lectura posibilita que diversos procesos entren a su espacio en memoria sin tener que sincronizarse con otros procesos que lo estén empleando.

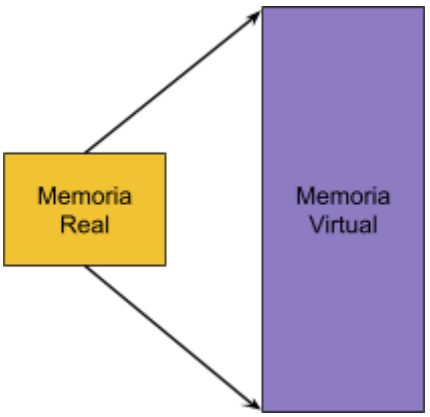
# 6. LA MEMORIA VIRTUAL

La memoria virtual es una técnica mediante la cual el SO simula tener más memoria principal de la que existe físicamente. Esta simulación o "engaño" es posible de implementar si se utiliza un medio de almacenamiento masivo o secundario de alta velocidad. Esta ilusión permite que los procesos se ejecuten creyendo que están cargados completamente en memoria RAM, cuando en realidad está cargada una porción de los mismos.

Memoria física o real  Memoria virtual

En un sistema que emplea paginación los procesos trabajan con una idealización de la memoria, en la cual ocupan el espacio completo de direccionamiento, desde el cero

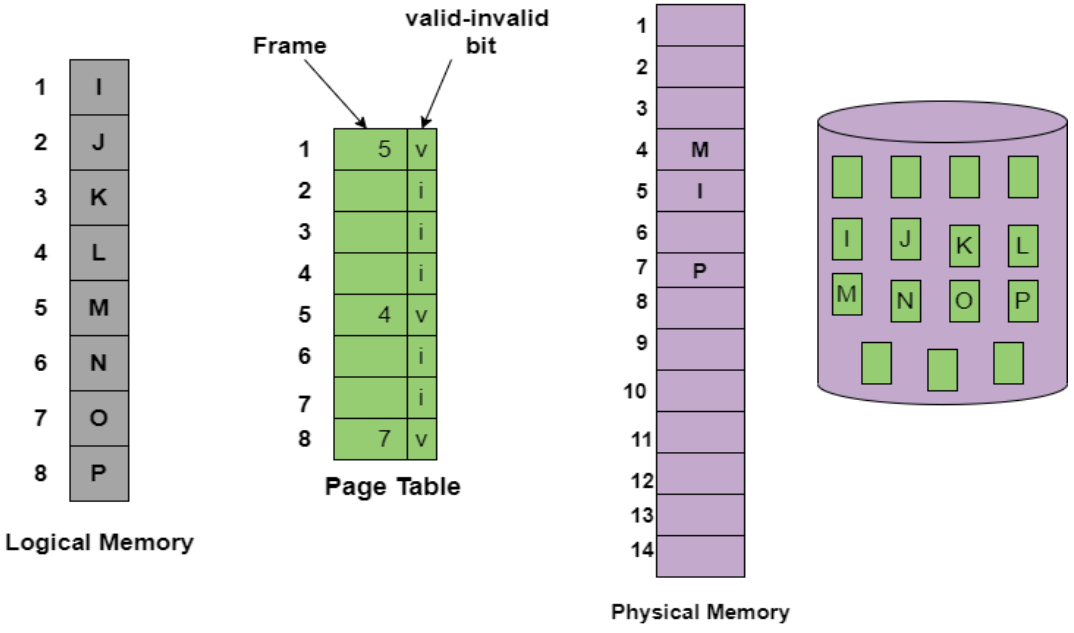
hasta el límite lógico de la arquitectura, independientemente del tamaño físico de la memoria disponible.




En el modelo de paginación los diferentes procesos pueden compartir regiones de memoria y direccionar más memoria de la físicamente disponible, ahí entra en juego la memoria virtual: para ofrecer a los procesos mayor espacio en memoria del que se cuenta físicamente, el sistema emplea espacio en almacenamiento secundario (disco) mediante un esquema de intercambio (swap) almacenando y cargando páginas enteras.

# 7. PAGINACIÓN POR/BAJO DEMANDA

En las 4 técnicas de administración de la memoria vistas hasta aquí (Asignación Particionada Fija y Dinámica, Segmentación y Päginación Simple) los procesos se cargan **completamente** en memoria principal. En la Paginación bajo demanda el requerimiento de completitud deja de ser un requerimiento.

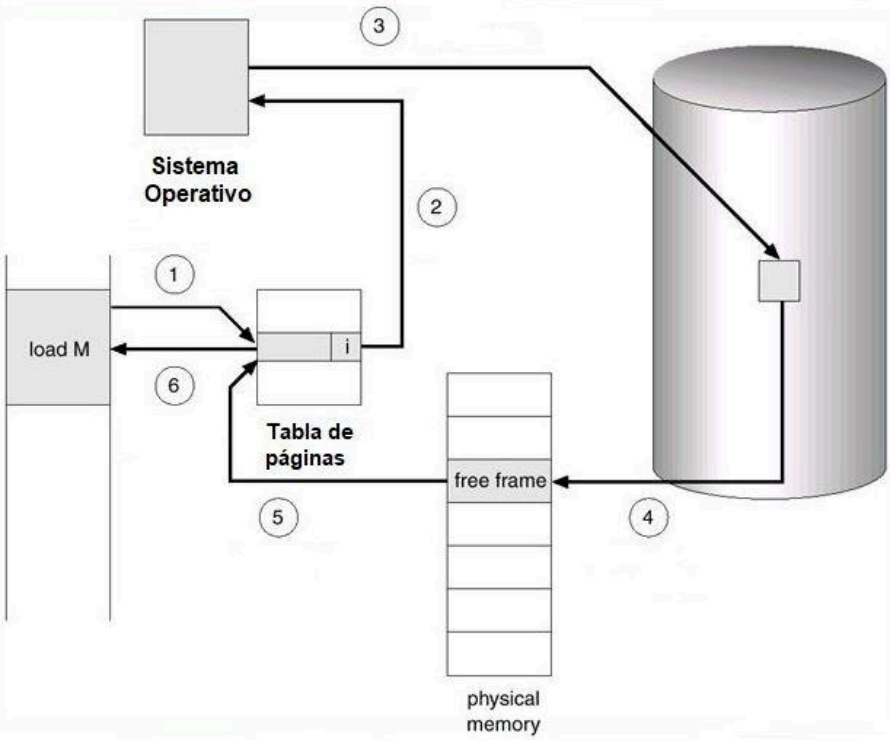


La paginación por demanda significa que, para comenzar a ejecutar un proceso, el SO carga solamente la porción necesaria para comenzar la ejecución (posiblemente una página o pocas) y que, a lo largo de la ejecución, el paginador, cuya característica es ser *flojo* o *perezoso* (*lazy*), sólo carga a memoria las páginas cuando van a ser utilizadas.

Al emplear un paginador flojo, las páginas que no sean requeridas nunca serán cargadas a memoria. La estructura empleada por la MMU para implementar un paginador perezoso incluye un bit de validez para la tabla de páginas, indicando para cada página del proceso si está presente o no en memoria. Si el proceso intenta emplear una página que esté marcada como no válida, esto causa un **fallo de página**, que lleva a que el SO lo bloquee y traiga a memoria la página solicitada para luego continuar con su ejecución.

Los pasos para gestionar un fallo de página son:

1. Verifica en el BCP si esta solicitud corresponde a una página válida para este proceso.
2. En caso de que la referencia sea inválida, el proceso es terminado. Pero si es una página válida, se solicita al SO que la cargue en memoria.
3. El SO busca un marco libre en la memoria y, si todos estuvieran ocupados, se selecciona dentro de los marcos asignados al proceso, una página *víctima* para ser reemplazada.
4. Se produce la lectura de la página en cuestión hacia el marco especificado o, bien, el reemplazo de la página.




5. Una vez que finaliza la lectura de disco, actualiza el BCP y la Tabla de Páginas para indicar que la página está en memoria.

6. Termina el bloqueo del proceso, continuando con la instrucción que desencadenó el fallo de página. El proceso puede continuar sin notar que la página ha sido intercambiada.

Llevando este proceso al extremo, se puede pensar en un sistema de paginación puramente sobre demanda. En un sistema así, ninguna página se carga en memoria si no es mediante un fallo de página. Un proceso, al iniciarse, comienza su ejecución sin ninguna página en memoria, y con el registro de siguiente instrucción del procesador apuntando a una dirección que no está en memoria (la dirección de la rutina de inicio). El SO responde cargando esta primera página, y conforme avanza el flujo del programa, el proceso irá ocupando el espacio que el SO le asignó.

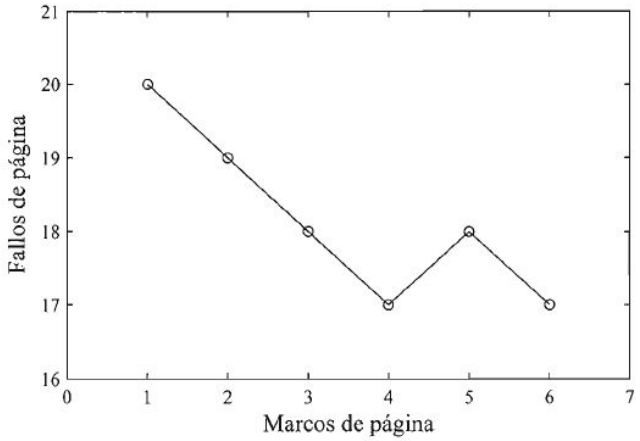
## 7.1. Reemplazo de Páginas

El reemplazo de páginas es una parte fundamental de la Paginación bajo demanda, ya que es la pieza que posibilita una verdadera separación entre memoria lógica y física. El mecanismo básico a ejecutar es simple: si todos los marcos asignados al proceso están ocupados, el SO debe encontrar una página que pueda reemplazar (una página **víctima**) y llevarla a disco. Luego, se puede emplear el marco liberado para cargar la página requerida, y continuar con la ejecución del proceso.
Esto implica una doble transferencia al disco (una para grabar la página víctima y una para traer la página de reemplazo) por lo tanto, una doble demora. Se puede, con un mínimo de burocracia adicional (aunque requiere de apoyo de la MMU) implementar un mecanismo que disminuya la probabilidad de tener que realizar esta doble transferencia: agregar un bit de modificación o bit de página sucia (*dirty* bit) a la tabla de páginas. Este bit se establece como apagado siempre que se carga una página a memoria, y es automáticamente encendido por hardware cuando se realiza un *acceso de escritura* a esa página (la página fue modificada).
Cuando el SO elige una página víctima, si su bit de página sucia está encendido, es necesario grabarla al disco, pero si está apagado, se garantiza que la información en disco es idéntica a su copia en memoria, y permite ahorrar la mitad del tiempo de transferencia.
Por lo tanto, la decisión de qué páginas reemplazar requiere implementar un algoritmo de *reemplazo de páginas* que permita obtener el menor número de fallos de página.
Para analizar los algoritmos de reemplazo se usará una cadena de referencia: una lista de referencias a memoria. Estas cadenas modelan el comportamiento de un conjunto de procesos en el sistema, así, diferentes comportamientos llevarán a resultados distintos. Para analizar un algoritmo de reemplazo, si se busca la cantidad de fallos de página producidos, además de la cadena de referencia, es necesario conocer la

cantidad de páginas y marcos del sistema que se está modelando. Por ejemplo, considerar la cadena de 12 solicitudes de acceso a página:

1, 4, 3, 4, 1, 2, 4, 2, 1, 3, 1, 4

Al recorrerla en un sistema con cuatro o más marcos, sólo se presentan cuatro fallos (el fallo inicial que hace que se cargue por primera vez cada una de las páginas). Si, en el otro extremo, se cuenta con sólo un marco, se presentan 12 fallos, dado que a cada solicitud se debería reemplazar el único marco disponible.
Un fenómeno interesante que se presenta con algunos algoritmos es la *Anomalía de Belady* (científico húngaro): si bien la lógica indica que a mayor número de marcos disponibles se tendrá una menor cantidad de fallos de página, como lo ilustra la figura, con algunas de cadenas de referencia y bajo ciertos algoritmos *puede haber* una regresión o degradación, en la cual la cantidad de fallos aumenta aún con una mayor cantidad de marcos.




Anomalía de Belady

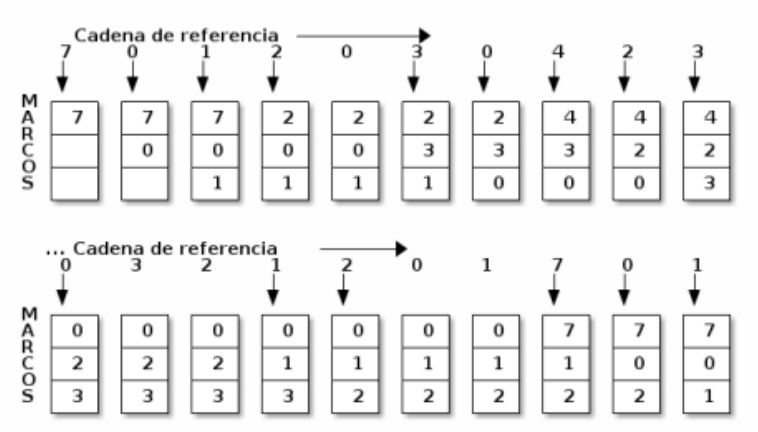
Para los algoritmos que se presentan a continuación, se asume que el proceso tiene asignados tres marcos en la memoria, y con la siguiente cadena de referencia:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

### 7.1.1. Primero en entrar, primero en salir (FIFO)

El algoritmo más simple y obvio de implementar es, nuevamente, la cola (FIFO): al cargar una página en memoria, se toma nota de en qué momento fue cargada, y cuando sea necesario reemplazar una página, se elige la que haya sido cargada hace más tiempo. Partiendo de un estado inicial en que las tres páginas están vacías, necesariamente las tres primeras referencias a distintas páginas de memoria (7, 0, 1) causarán fallos de página. La siguiente (2) causará uno, pero la quinta referencia (0) puede ser satisfecha sin requerir una nueva transferencia.

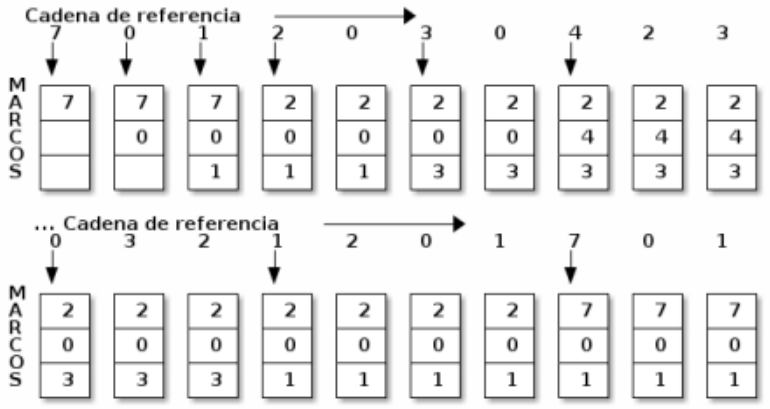
La principal ventaja de este algoritmo es la simplicidad. Su implementación puede ser tan simple como una lista enlazada circular, cada solicitud que recibe se agrega en el último elemento de la lista. Su desventaja es que no toma en cuenta la historia de las últimas solicitudes, por lo que puede causar un bajo rendimiento. Todas las páginas tienen la misma probabilidad de ser reemplazadas, sin importar su frecuencia de uso.




Con las condiciones aquí presentadas, un esquema FIFO causará 15 fallos de página en un total de 20 accesos requeridos. El algoritmo FIFO es vulnerable a la anomalía de Belady. La figura ilustra este fenómeno al pasar de tres a cuatro marcos asignados al proceso:


### 7.1.2. Reemplazo de páginas óptimo (OPT)

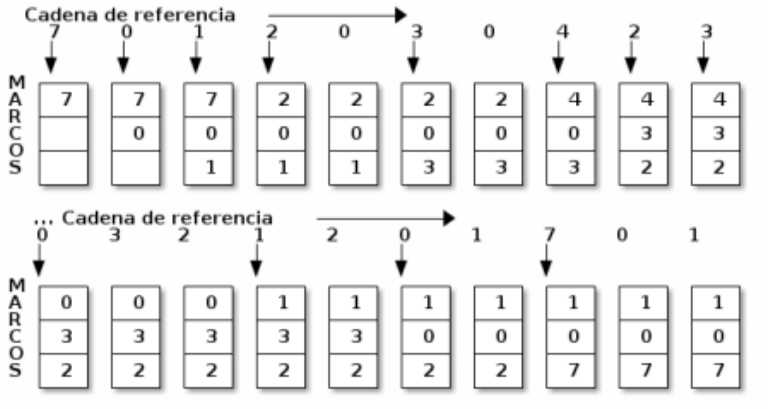
Un algoritmo, de interés puramente teórico, conocido como OPT o MIN. Bajo este algoritmo se elige como página víctima a aquella página que no vaya a ser utilizada por un tiempo máximo (o nunca más).




Si bien este algoritmo está demostrado como óptimo o mínimo, se mantiene como referencia teórica porque requiere conocimiento a priori de las necesidades a **futuro** del sistema. Su principal utilidad reside en que, calculando el número de fallos que se presentan al seguir OPT, es posible ver qué tan cercano resulta otro algoritmo respecto al caso óptimo. Para esta cadena de referencia, y con tres páginas, se tiene un total de nueve fallos.

### 7.1.3. Menos recientemente utilizado (LRU)

En este algoritmo, cuando se necesita elegir una página víctima, se elige la página que no ha sido empleada hace más **tiempo**.




Para la cadena de referencia, LRU genera 12 fallos, en el punto medio entre OPT y FIFO.
Se ha demostrado que LRU y OPT están libres de la anomalía de Belady.

### 7.1.4. Más/Menos frecuentemente utilizada (MFU/LFU)

Estos dos algoritmos se basan en mantener un contador, tal como lo hace LRU, pero en vez de medir el tiempo, miden la **cantidad** de referencias que se han hecho a cada página.
El MFU parte de la lógica que, si una página fue empleada muchas veces, probablemente vuelva a ser empleada muchas veces más; LFU parte de que una página que ha sido empleada pocas veces es probablemente una página recién cargada, y va a ser empleada en el futuro cercano.
Estos dos algoritmos son tan costosos de implementar como LRU, y su rendimiento respecto a OPT no es tan cercana, por lo cual casi no son empleados.

### 7.1.5. LRU con bit de referencia

Dada la complejidad que presenta la implementación de LRU en hardware, existen variantes. En una de ellas se busca una aproximación, en donde todas las entradas de la tabla de páginas tengan un bit adicional, denominado bit de referencia o de acceso.
Al iniciar la ejecución, todos los bits de referencia están apagados (0). Cada vez que se accede a un marco, su bit de referencia se enciende (esto, en general, lo realiza el hardware). El SO periódicamente hace que se apaguen todos los bits de referencia. En caso de presentarse un fallo de página, se elige por FIFO sobre el subconjunto de marcos que no hayan sido referenciados en el periodo actual (esto es, entre todos aquellos para los cuales el bit de referencia sea 0).

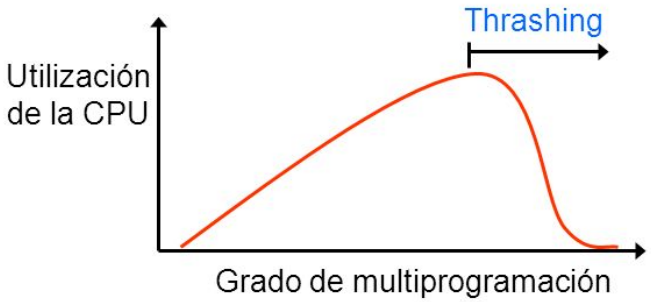
### 7.1.6. Algoritmos con manejo de buffers

Un mecanismo que se emplea cada vez con mayor frecuencia es que el sistema no espere a enfrentarse a la necesidad de reemplazar un marco, sino que proactivamente busque tener siempre espacio vacío en memoria. Para hacerlo, conforme la carga lo permita, el SO busca las páginas sucias más proclives a ser paginadas a disco y va actualizando el disco (marcándolas nuevamente como limpias). De este modo, cuando tenga que traer una página nueva del disco, siempre habrá espacio donde ubicarla sin tener que esperar a que se transfiera una para liberarla.

## 7.2. Hiperpaginación o thrashing

Se denomina hiperpaginación (en inglés *thrashing*, cuya traducción literal es *golpiza o paliza*) a la situación en la que se cargan y descargan sucesiva y constantemente partes de la imagen de un proceso desde y hacia la memoria principal y la memoria virtual (disco) degradando el rendimiento del sistema.

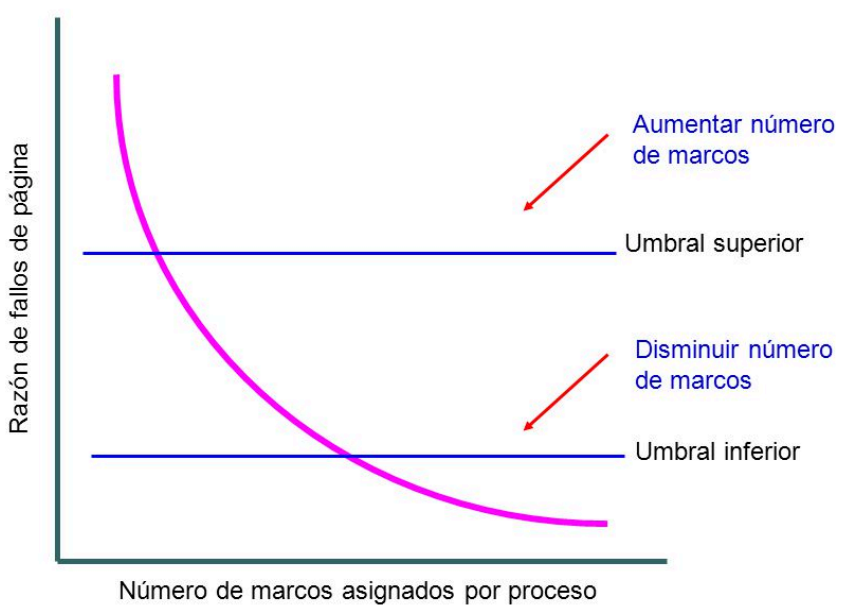
En general, cualquier proceso que no cuente con marcos suficientes, ya sea porque el sistema dispone de una memoria pequeña o porque se le asignaron pocos marcos al proceso, provocará fallos de página muy frecuentemente. Si se reemplazan páginas que están activas, se estará sustituyendo una página que casi de inmediato se volverá a necesitar. Por lo tanto, pronto vuelve a generarse otro fallo de página, ocurriendo esto una y otra vez.
Esta situación provoca que el sistema de paginación vaya elevando su actividad de intercambio de páginas entre memoria principal y memoria secundaria, mientras que, al mismo tiempo, el procesador va reduciendo su actividad porque debe esperar que las páginas se carguen.




Por lo tanto, para evitar la hiperpaginación se le debe ofrecer al proceso la cantidad de marcos que necesita; pero la pregunta es ¿cuántos marcos necesita? La estrategia de **frecuencia** de fallos de página es una de las técnicas para responder esta pregunta.
Un proceso a lo largo de su ejecución produce fallos de página, lo cual es normal mientras esté dentro de un rango. Entonces, se puede establecer un *límite superior e inferior* para la tasa deseada de fallos de página. Si la tasa de fallos de página excede el límite superior, se asignan más marcos a ese proceso; si la tasa cae por debajo del límite inferior, se quitan marcos al proceso. Así se puede medir y controlar directamente la tasa de fallos de página para evitar la hiperpaginación.
De manera que, si aumenta mucho la tasa de fallos y no hay marcos disponibles, se debe seleccionar un proceso (preferentemente alguno bloqueado desde hace tiempo) y suspenderlo (enviarlo a memoria secundaria). Los marcos liberados se distribuyen entre los procesos con tasa elevada de fallos de página.
Por lo tanto, la frecuencia de fallos está muy ligada con el número de marcos que son asignados a los procesos:

- Frecuencias altas de fallas de página → el proceso necesita más marcos
- Frecuencias bajas de fallas de página → el proceso necesita menos marcos

Teniendo esto en cuenta, el SO supervisa la utilización de la CPU. Si ésta es baja, aumenta el nivel de multiprogramación introduciendo un nuevo proceso en el sistema. Para ello, quita algunos marcos asignados a otros procesos con el objeto de hacer espacio y para cargar el nuevo proceso.




Ahora, si un proceso entra en una nueva fase de su ejecución y necesita más marcos y comienza a generar fallos de página, el SO responde asignando marcos de otros procesos. Sin embargo, estos procesos necesitan esos marcos, por lo que también fallan. Todos estos procesos generando fallos de página dan comienzo a la hiperpaginación y se desploma la productividad del sistema.
La ejecución de procesos es mínima ya que hay una alta tasa de fallos de páginas y el procesador permanece subutilizado.
En definitiva, es un juego de equilibrio en el cual el SO debe dar cabida en la memoria a la mayor cantidad de procesos sin que se produzca la hiperpaginación.

# 8. BIBLIOGRAFÍA

Fundamentos de Sistemas Operativos, A. Silberschatz
Sistemas Operativos. Aspectos internos y principios de funcionamiento, W. Stallings
Sistemas operativos modernos. A. Tanenbaum
Sistemas operativos. Un enfoque basado en conceptos. D. Dhamdhere