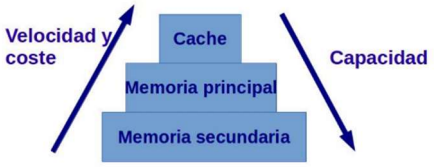
Tema 3: Gestión de la Memoria

- Una de las principales tareas del SO es **crear y administrar abstracciones de la memoria** del sistema.

Introducción – La memoria y el SO

- Idealmente, se tendría una memoria rápida, barata, grande y no volátil.
  Como en la realidad esto es imposible, se usa la **jerarquía de memoria** para conseguir estos objetivos:
  - ↳ La jerarquía está fundamentalmente formada por la caché, la memoria principal (RAM) y la memoria secundaria (disco).
    - Cuan más arriba esté un nivel, más cara, rápida y pequeña es la memoria.
    - Su funcionamiento se basa en el principio de la **localidad** (espacial y temporal).



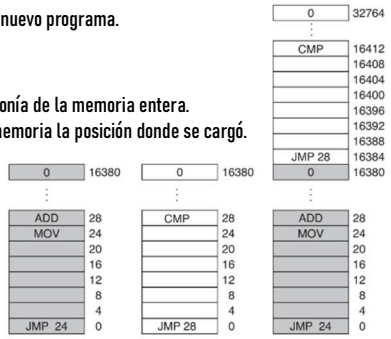
- La parte del SO que administra (parte de) la jerarquía de memoria se conoce como el ADMINISTRADOR DE MEMORIA. Sus objetivos son:
  - Llevar registro de qué **partes de la memoria están en uso**.
  - **Asignar memoria** a los procesos que la solicitan.
  - **Liberar memoria** de los procesos que terminan.
  - ▸ Generalmente es el hardware el que se ocupa de gestionar la memoria caché, por lo que nos centraremos en la **memoria secundaria**.

Sin abstracción de memoria

- En este modelo no se proporciona ninguna abstracción: los programas **ven directamente toda la memoria física**.
  - ↳ El modelo de programación presentado es directamente un conjunto direcciones **desde el 0 hasta un valor máximo**.
- Bajo estas condiciones es muy **difícil** tener **varios programas** ejecutándose en memoria a la vez,
  pues todos usan las mismas direcciones de memoria por lo que se pueden sobrescribir entre sí.
- ❖ Aunque no haya abstracción, hay varias **opciones para organizar la memoria** en este modelo →
- ▸ En la actualidad este modelo sólo se utiliza en algunos sistemas empotrados.

Ejecución de múltiples programas sin abstracción de memoria

- **Intercambio** → cada programa se carga en la totalidad de la memoria principal.
  Por tanto, cada vez que se cambia de programa se tiene que guardar todo el contenido de la memoria en el disco, y sustituirlo por el nuevo programa.
  - ↳ Naturalmente, es una opción **muy costosa**.
- **Carga consecutiva** → los programas se cargan enteros en la memoria, uno detrás de otro.
  Como consecuencia, las referencias a posiciones de memoria dejan de ser correctas, ya que se escribieron considerando que se disponía de la memoria entera.
  - Para solucionar esto se usa la **reubicación estática** → cuando se carga un programa, se les suma a todas sus referencias a memoria la posición donde se cargó.
    - ↳ Esto soluciona el problema, pero también hace mucho **más lenta la carga** de programas.

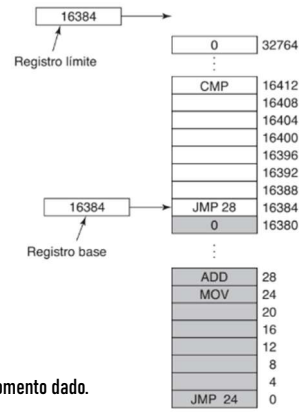


Abstracción de memoria – El espacio de direcciones

- Para poder permitir que haya **varios programas en memoria al mismo tiempo** hay que solucionar dos cuestiones importantes:
  - **Protección** → un programa no debe poder acceder a la información de otros (ni a la del SO).
  - **Reubicación** → se debe contemplar que durante la ejecución de un programa los datos cambien de posición en la memoria principal.
- El ESPACIO DE DIRECCIONES es el conjunto de direcciones de memoria que puede utilizar un proceso.
  - **Cada proceso** tiene su propio espacio de direcciones.
  - Los espacios de direcciones son **independientes** → la dirección x en el proceso A es una ubicación física diferente de la dirección x en el proceso B.
    - ↳ Esto se consigue mediante la **reubicación dinámica**.

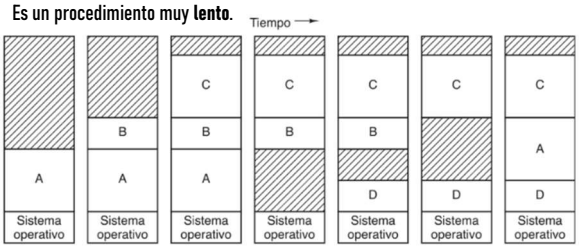
Reubicación dinámica

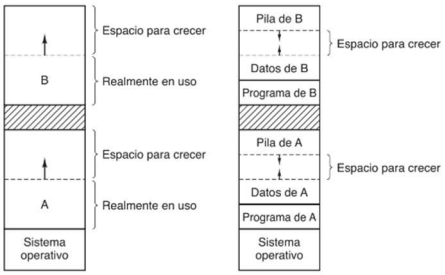
- Consiste en asociar el **espacio** de direcciones **de cada proceso** a una **parte distinta de la memoria física**, de manera que todas ellas sean **consecutivas**.
- Para **delimitar** cada espacio de direcciones se usan dos registros especiales que se establecen cuando se carga el programa:
  - Registro **base** → almacena la dirección física donde comienza el programa en memoria.
  - Registro **límite** → almacena la longitud del programa.
- En cada **referencia** a memoria, el hardware realiza las siguientes operaciones:
  1. Suma el valor base a la dirección referenciada.
  2. Comprueba si el resultado está más allá del final del espacio (dirección base + límite). Si es así, genera un fallo y aborta el acceso.
  - Como consecuencia, los **accesos a memoria son más lentos**, pues implican una comprobación (rápida) y una suma (lenta).
    - ↳ Como mínimo, se accede a memoria una vez por ciclo, para realizar el fetch de la instrucción a ejecutar.
- Este esquema funciona cuando la **memoria física** es lo **suficientemente grande** como para contener todos los procesos en ejecución en un momento dado.
  En la práctica, la memoria que requieren los procesos es mucho mayor que la memoria física disponible.
  - ▸ Hay dos esquemas alternativos que solucionan esto: el **intercambio** y la **memoria virtual**.



INTERCAMBIO

- Consiste en **cargar cada proceso entero** en memoria, **ejecutarlo durante un tiempo** y después **regresarlo al disco** (intercambiarlo).
- El mismo proceso podrá estar en distintos puntos del tiempo en distintas posiciones de memoria, por lo que se necesitará usar los **registros base y límite** de la reubicación dinámica para manejar las referencias a la memoria.
- El intercambio de procesos en memoria provoca que se vaya **fragmentando,** pues se van generando **huecos** demasiado **pequeños** para colocar un proceso en ellos.
  Cuando la fragmentación es muy alta, se puede usar la **compactación**, que combina todos los huecos pequeños en un hueco grande desplazando los procesos lo más abajo posible.
  - Es un procedimiento muy **lento**.



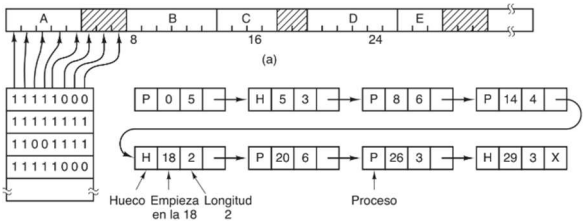


¿CUÁNTA MEMORIA SE LE ASIGNA A CADA PROCESO?

- Si los procesos tienen un tamaño **fijo**, la asignación es muy **sencilla**.
- Sin embargo, normalmente los procesos tendrán un tamaño **variable**, pues necesitarán **crecer** para acomodar su **pila** o asignar dinámicamente memoria en el ***heap***.
  Cuando un proceso necesite crecer intentará (en orden):
  1. Ocupar huecos adyacentes.
  2. Si no hay huecos adyacentes, se moverán a un hueco lo suficientemente grande.
  3. Si no hay huecos lo suficientemente grandes, se intentará compactar la memoria para crear uno y se moverá allí.
  4. Si nada de esto funciona, el proceso se tendrá que suspender hasta que se libere la memoria suficiente.
- Para evitar tener que hacer todo eso cada vez que un proceso necesite crecer, será conveniente asignar **memoria en exceso** cuando se carga o mueve uno en memoria.
  - En los procesos con **dos segmentos en crecimiento** (pila y *heap*), estos pueden crecer en direcciones opuestas en el espacio para el crecimiento hasta que ambos se encuentren.
    - Si es espacio para crecimiento se **agota**, habrá que mover o suspender el proceso como antes.

ADMNISTRACIÓN DE MEMORIA LIBRE

- Hay dos formas de que el administrador de memoria lleve registro del uso de la memoria: los **mapas de bits** y las **listas libres.**
- Ambos se sitúan en el kernel para que el administrador pueda acceder a ellos para averiguar dónde puede insertar nuevos procesos y cuándo debe compactar la memoria.

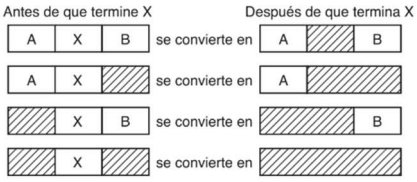


MAPAS DE BITS

- La memoria se divide en **unidades de asignación** de un tamaño fijo. Para cada unidad de asignación en de la memoria habrá un bit correspondiente en el mapa de bits, que será 0 si está libre y 1 si está ocupada.
- El **tamaño** del mapa será siempre **constante** pues depende del tamaño de la memoria y del de la unidad de asignación.
  - Unidad de asignación pequeña → mapa de bits grande.
  - Unidad de asignación grande → mapa de bits más pequeño, pero se puede desperdiciar mucha memoria en la última unidad de cada proceso si su tamaño no es un múltiplo del tamaño de la unidad de la asignación.
- El principal problema es que cada vez que se lleve un proceso a memoria el administrador tendrá que **recorrer el mapa de bits** contando ceros hasta encontrar una **secuencia de ceros** tan larga como el tamaño del proceso, lo cual es un procedimiento muy **lento**.

LISTAS LIGADAS

- La lista está formada por **segmentos de memoria** que o bien contienen un proceso o bien son un hueco.
  Cada entrada especificará si se trata de un **proceso** o un **hueco**, la **dirección** donde comienza, su **longitud** y un apuntador a la **siguiente entrada**.
- El **tamaño** de la lista **no es constante**, pues depende del número de huecos y procesos que haya en la memoria en un determinado momento.
- La lista estará **ordenada por dirección** de inicio.
  De esta manera, cuando un proceso **regrese a disco** se podrá **actualizar** muy **fácilmente**.
  - Si las dos entradas vecinas son procesos → se marca el proceso intercambiado como hueco, sin más.
  - Si una de las entradas vecinas es un hueco → la entrada del proceso intercambiado se fusiona con la del hueco, creando un hueco más grande.
  - Si las dos entradas vecinas son huecos → las tres entradas se fusionan, creando un hueco más grande.
    - Fusionando los huecos la memoria se **fragmenta menos**.

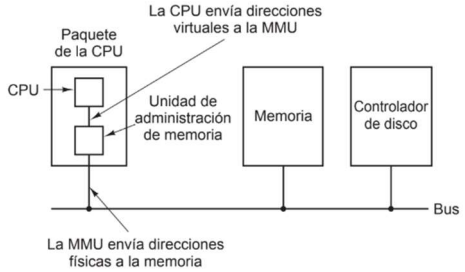


- Existen diversos **algoritmos de asignación de memoria** para insertar procesos.
  - **Primer ajuste** → se recorre la lista hasta encontrar un hueco lo suficientemente grande.
    - Es muy rápido pues recorre la lista lo menos posible.
  - **Siguiente ajuste** → funciona como el de primer ajuste, pero se empieza recorriendo desde donde se quedó el recorrido anterior.
    - Da un rendimiento peor al del primer ajuste.
  - **Mejor ajuste** → se recorre la lista entera y se selecciona el hueco más pequeño donde quepa el proceso.
    - Da un rendimiento peor al del primer ajuste.
    - Sorprendentemente, desperdicia más memoria que el primer ajuste pues genera huecos muy pequeños.
  - **Peor ajuste** → se recorre la lista entera y se selecciona el hueco más grande.
    - Da un rendimiento peor al del primer ajuste.
  - Todos estos algoritmos se pueden acelerar si se mantienen **listas separadas para huecos y procesos**.
    - La **lista de huecos** se puede mantener **ordenada por tamaño** para acelerar el **mejor ajuste**, que entonces funcionaría como el primer ajuste.
    - La **inserción** será muy **rápida** pues sólo habría que recorrer la lista de huecos.
    - La **liberación de memoria** será más **lenta** pues los huecos adyacentes ya no se podrán fusionar directamente.
  - **Ajuste rápido** → se maneja una lista para procesos y varias listas para los tamaños de hueco más habitualmente solicitados.

MEMORIA VIRTUAL

- Los sistemas con multiprogramación afrontan los siguientes **problemas**, que se solucionan usando la memoria virtual:
  - Los **espacios de direcciones** son cada vez más **grandes**, por lo que en ocasiones no caben en memoria principal enteros.
    - Esto se intentó solucionar haciendo que los programadores dividieran **manualmente** su código en porciones pequeñas (denominadas **sobrepuestos**) para poder cargarlos en memoria poco a poco conforme se fuera necesitando, pero era una solución muy propensa a errores.
  - Se necesita poder almacenar **muchos procesos en memoria a la vez**.
    - Esto se intentó solucionar con el **intercambio**, pero resulta demasiado lento, pues implica leer y almacenar porciones muy grandes de datos en el disco constantemente.

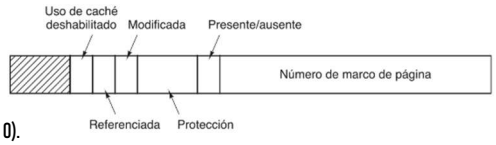
- En la MEMORIA VIRTUAL cada proceso tiene su propio espacio de direcciones, divido en **páginas**.
  - El espacio de direcciones de los procesos está formado por **direcciones virtuales**, que no coinciden con las direcciones físicas de la memoria principal.
    - Por tanto, para poder acceder a la información referenciada por un proceso, se necesita traducir la dirección virtual a la dirección física a la que está asociada.
      Esta tarea la realiza la MMU (unidad de manejo de memoria), que forma parte de la CPU.



PAGINACIÓN

- El espacio de direcciones de un proceso se divide en **páginas**, que son un rango contiguo de direcciones de 2^{n-p} unidades direccionables (bytes o palabras).
  - El tamaño del espacio de direcciones será de 2^n unidades direccionables, siendo n la longitud de palabra del sistema (el ancho de los registros del procesador).
  - El espacio de direcciones se divide en 2^p páginas.
  - Cada dirección del espacio de direcciones se divide en $\begin{cases} \text{campo de página } (p \text{ MSB}) \rightarrow \text{indica a qué página pertenece la dirección.} \\ \text{campo de desplazamiento}(n-p \text{ LSB}) \rightarrow \text{indica a qué unidad direccionable de la página se refiere la dirección.} \end{cases}$
- La memoria física se dividirá en **marcos de página** del mismo tamaño que las páginas del espacio de direcciones.
  - El tamaño de la memoria física tendrá que ser, por lo tanto, un múltiplo del tamaño de página.
  - La memoria física se divide en $tamano\_total/tamano\_pagina$ marcos.
- **Las páginas se asocian a marcos de manera completamente asociativa**, es decir, cualquier página de cualquier espacio de direcciones puede estar en cualquier marco.
  - De esta manera, se provoca el mínimo número de fallos de página.
- **No todas** las páginas de un proceso **estarán en memoria principal** todo el tiempo que se están ejecutando.
  De hecho, cuando un proceso se empieza a ejecutar por primera vez, ninguna de las páginas de su espacio de direcciones virtuales lo está.
- Cuando un programa referencia una dirección virtual de su espacio de direcciones, se **comprobará si está en memoria principal**.
  - Si está en memoria principal, la MMU traducirá su dirección virtual a la dirección física asociada y esta se colocará en el bus para buscarla en la memoria principal.
  - Si no está en memoria principal, se producirá un **fallo de página**: el proceso se bloquea mientras se busca y copia la página entera del disco a la memoria principal.
    - Si cuando se pretende traer una página del disco a la memoria principal esta ya está llena, habrá que **reemplazar** alguna de sus páginas.

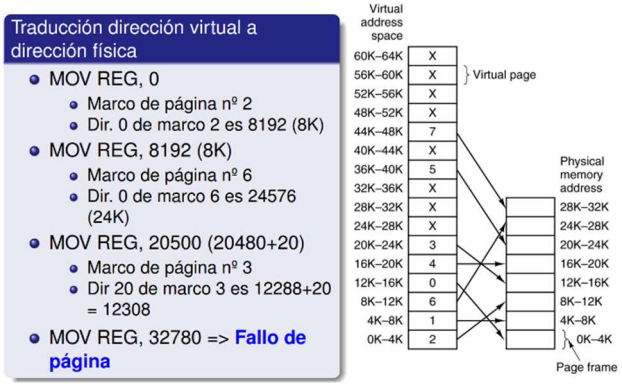
TABLA DE PÁGINAS

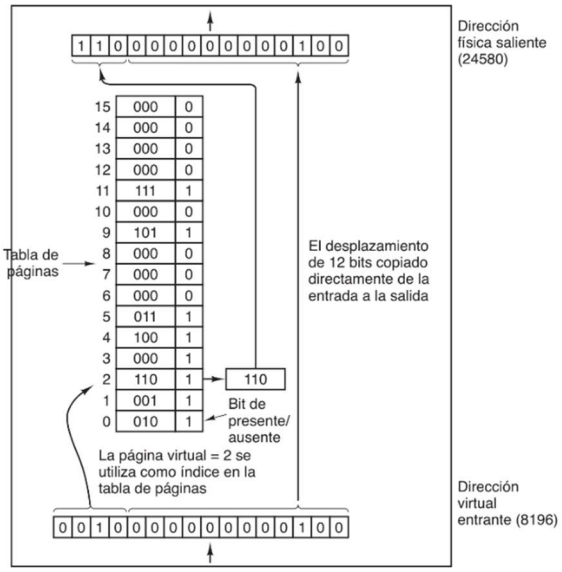


- La **tabla de páginas** (TP) de un proceso asocia páginas virtuales a un marco en memoria principal.
- El **índice** de la TP es el número de página virtual.
- La estructura de una entrada de la TP suele estar formada por:
  - **Número de marco de página** al que está asociada la página (si no está en memoria principal será 0).
  - **Bit presente/ausente** → si es 1, significa que la página está en memoria principal. Si es 0, no está en memoria principal.
  - **Bits de protección** → (r,w,x) si uno es 1, la página tiene ese permiso. Si es 0, no lo tiene.
    - Por tanto, no se deben mezclar datos e instrucciones en la misma página, pues eso implicaría activar todos sus bits de protección.
  - **Bit modificada** → se pone a 1 cuando se escribe en la página.
    - Se usa cuando se quita la página de memoria principal para determinar si habrá que **copiar** su contenido en el disco.
      Si se tiene que hacer, se copia la página **entera**, pues no se sabe dónde fue modificada.
  - **Bit referenciada** → se pone a 1 cuando se referencia la página (para escritura o lectura).
    - Se usa como apoyo al **algoritmo de reemplazo**, pues las páginas que no han sido usadas desde que se cargaron serán mejores candidatas a ser reemplazadas.
  - **Bit de deshabilitación de uso de caché** → si es 1, ninguna de las palabras de la página podrá almacenarse en la caché.
    - Se usa en páginas compartidas o páginas donde se ejecutan operaciones de lectura y escritura de E/S para mantener la **coherencia de cache**.
      - Por ejemplo, si hay dos hilos de un proceso ejecutándose a la vez cada uno en un núcleo y uno de ellos pretende escribir en una página, la llevará a la caché de su núcleo. La modificación se almacenará en la caché y no será visible por el otro hilo hasta que esa línea de caché sea reemplazada.
  - Puede haber bastantes más bits para ayudar a los **algoritmos de reemplazo**.
  - Normalmente, la **dirección de disco** usada para guardar la página cuando no está en memoria **no se almacena en la tabla**.
    - En algunos sistemas, se aprovecha que el campo de marco de página no se usa cuando la página no está en memoria para indicar la zona de disco donde está.

TRADUCCIÓN DE DIRECCIÓN VIRTUAL A DIRECCIÓN FÍSICA

1. Se envía la dirección virtual a la MMU.
2. La MMU comprueba la TP
3. Si el bit de presente/ausente es 1:
   3.1. La MMU cambia el campo de página de la dirección virtual por el número del marco asociado a ella.
   3.2. El campo de desplazamiento se mantiene idéntico.
   3.3. Se coloca la dirección obtenida en el bus de memoria.
4. Si el bit de presente/ausente es 0:
   4.1. Se bloquea el proceso.
   4.2. Se busca y copia la dirección buscada del disco a la memoria principal.
   4.3. Si la memoria principal está llena, habrá que reemplazar alguna de sus páginas.





ACELERACIÓN DE LA PAGINACIÓN

Las principales cuestiones que debe resolver un sistema de paginación son:

- La **traducción** de una dirección virtual a una dirección física debe ser **rápida** → TLP.
  - ↳ Se realiza una traducción cada vez que se referencia la memoria, y esto se hace como mínimo una vez por instrucción, pero puede ser que más.
- Si el espacio de direcciones es grande, la **TP** será **grande** → TP multinivel, TP invertida.
  - ↳ La mayoría de sistemas funcionan con direcciones de 32 o 64 bits, así que los espacios de direcciones tendrán 2^{32} o 2^{64}, por lo que tendrán muchas páginas. Además, cada proceso tiene su propia TP, pudiendo haber muchos procesos en ejecución en cada momento en el sistema.

Hay 2 opciones para tratar las TPs:

– Mantener una **única TP** como un conjunto de **registros** de hardware, uno para cada página.
Cuando se inicia un proceso, el SO carga los registros con la tabla de páginas del proceso almacenada en memoria principal.
  - ✓ Es muy simple.
  - ✓ No requiere acceder a memoria principal durante la traducción.
  - × Es muy caro si las tablas de páginas son grandes.
  - × Las conmutaciones de procesos serán más lentas ya que implicarán la carga de su TP en los registros.

– Mantener **una TP por proceso** en la **memoria principal**. Esto es lo que se hace en la mayoría de sistemas modernos.
Para realizar una traducción sólo se necesita un registro que apunte al inicio de la TP del proceso en ejecución.
  - ✓ Las conmutaciones de procesos sólo implican cargar una dirección en un registro.
  - × Se requiere acceder a memoria principal para leer la TP cada vez que se tenga que traducir una dirección.

BÚFER DE TRADUCCIÓN ADELANTADA

Se observa que:

- Cada **referencia** a memoria implica un **acceso extra a memoria** para leer la TP, lo cual limita mucho el rendimiento del sistema.
- La mayoría de programas **referencian muchas veces** un **pequeño número de páginas**, por lo que sólo se lee con mucha frecuencia una pequeña fracción de la TP.

○ La solución a esto es equipar la MMU con un **búfer de traducción adelantada** (TLB), que es un dispositivo hardware que asocia direcciones virtuales a direcciones físicas sin usar la tabla de páginas de la memoria principal.

- La TLB consiste en un **pequeño número de entradas** (≤ 64) cuyos campos son:
número de página virtual, bit modificada, bits protección, marco asociado y un **bit de validez** que indica si la entrada es válida (es 0 cuando está vacía).
  - ↳ Todos estos campos (excepto el bit de validez) tienen una correspondencia uno a uno con la información almacenada en la TP.

- Cada vez que se le presenta una dirección virtual a la MMU para que la traduzca:
  1. Se comprueba si el número de la página está en la TLB.
     - ↳ Esta comprobación se realiza mediante hardware (un comparador en cada entrada de la tabla) de manera paralela.
  2. Si se encuentra el número de página en la TLB y el bit de validez es 1:
     2.1. Se toma el marco de la TLB sin pasar por la tabla de páginas.
  3. Si no se encuentra el número de página en la TLB:
     3.1. La MMU busca el número de página en la TP.
     3.2. Se copia la entrada de la TP en una entrada de la TLB.
     3.3. Si la TLB está llena, habrá que reemplazar alguna de sus entradas.
       - ↳ Si el bit modificado de la entrada desalojada es 1, se establece el bit modificado de la entrada de la TP a 1 también.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TABLAS DE PÁGINAS MULTINIVEL

Se observa que:

- El espacio de direcciones de un proceso es muy grande, por lo que su TP lo es también.
- Almacenar una tabla de 2^p entradas en memoria principal en todo momento para cada proceso activo es muy costoso.

○ La solución a esto es la **tabla de páginas multinivel,** que consiste en dividir la TP de cada proceso en varios bloques del tamaño de una página, de manera que cada uno de ellos puede ocuparía un marco de memoria principal pero no todos ellos tengan que estar en memoria principal a la vez.

↳ Por tanto, esta división de la TP es análoga a la paginación del espacio de direcciones.

Para esto, usaremos **traducciones en dos niveles**. Para cada proceso existen:

- Varias **tablas de páginas de segundo nivel**, TP2, que contendrán las entradas de la TP correspondientes a un bloque.
  - Si consideramos que cada entrada de TP ocupa 2^x bits (pues normalmente será una potencia entera de 2), entonces en cada TP2 habrá $2^{n-p}/2^x$ entradas. Para abreviar, diremos que tendrá 2^α entradas.
  - Por tanto, habrá $2^{p-\alpha}$ TP2 para cada proceso.
  - Cada TP2 puede estar, en un momento dado, en memoria o no, igual que las páginas del espacio de direcciones.
- Una **tabla de páginas de primer nivel**, TP1, que asocia las TP2 al marco de memoria principal que ocupan.
  - Tendrá una entrada por cada TP2, así que tendrá $2^{p-\alpha}$ entradas.
  - Habrá una única TP1 para cada proceso.
  - La TP1 de un proceso siempre estará en memoria principal.
- Ahora las direcciones virtuales se dividen en $\begin{cases} \text{campo de TP1 (}p-a\text{ MSB del campo de palabra)} \rightarrow \text{indica en qué TP2 está la entrada de TP de la dirección.} \\ \text{campo de TP2 (}a\text{ LSB del campo de palabra)} \rightarrow \text{indica a qué página pertenece la dirección.} \\ \text{campo de desplazamiento(}n-p\text{ LSB)} \rightarrow \text{indica a qué unidad direccionable de la página se refiere la dirección.} \end{cases}$.

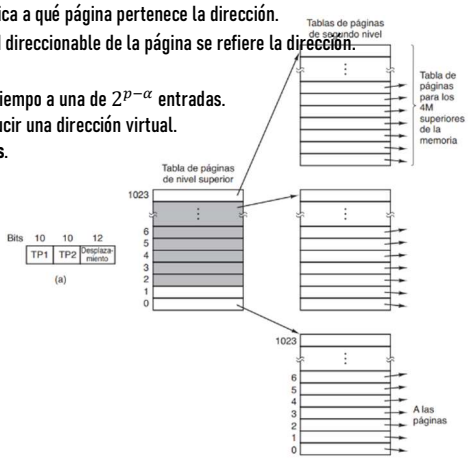
✓ Así, se pasa de tener que almacenar una tabla de 2^p entradas para cada proceso en memoria principal todo el tiempo a una de $2^{p-\alpha}$ entradas.

× Sin embargo, ahora habrá que acceder numerosas veces a la memoria principal cada vez que se tenga que traducir una dirección virtual.

- Cuando el sistema tiene una palabra ancha (64), se puede realizar de manera similar una traducción en **3 niveles**.

↳ Se podrían añadir incluso más niveles, pero crearía mucha complejidad y sería poco útil.

- Cada vez que se le presenta una dirección virtual a la MMU para que la traduzca:
  1. Se lee el campo de TP1 de la dirección.
  2. Se comprueba en la entrada correspondiente de la TP1 si la TP2 está en memoria.
  3. Si la TP2 correspondiente no está en memoria, se trae del disco como si fuera una página normal.
  4. Si la TP2 Correspondiente sí está en memoria:
     - 4.1. Se lee el campo TP2 de la dirección.
     - 4.2. Se comprueba en la entrada correspondiente de la TP2 si la página está en memoria.
     - 4.3. Se procede con normalidad.



TABLAS DE PÁGINAS INVERTIDA

Se observa que:

- En los sistemas de 64 bits, las TP son tan grandes que ni siquiera las TP multinivel las hacen manejables en memoria principal.

↳ Por ejemplo, si se tienen páginas de 2^{12} bytes, la TP1 de cada proceso será de 30PB.

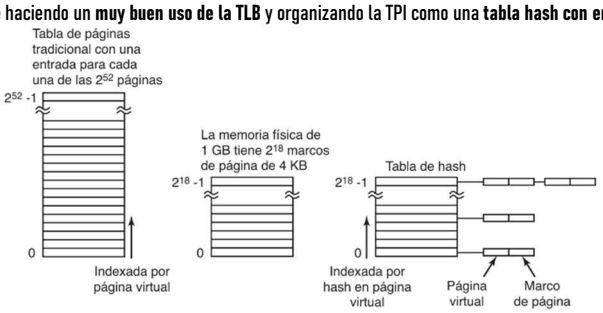
○ La solución a esto es la **tabla de páginas invertida**, que se mantendrá siempre en memoria principal y tendrá una entrada por cada marco de la memoria principal. Cada una de estas entradas especificarán el **número de página** y **proceso** al que pertenece la página que se encuentra en ese momento en el marco.

↳ Así, si se tienen páginas de 2^{12} bytes y una memoria principal de 2^{30} bytes, sólo se necesitarán 2^{18} entradas.

✓ Ahorran grandes cantidades de espacio de la memoria principal.

× Sin embargo, la traducción será muy lenta, ya que en cada referencia de memoria se tiene que recorrer la tabla entera.

↳ Esto puede optimizarse haciendo un **muy buen uso de la TLB** y organizando la TPI como una **tabla hash con encadenamiento** cuya clave sea la dirección virtual.



ALGORITMOS DE REEMPLAZO DE PÁGINAS

○ Cuando ocurre un fallo de página y la memoria principal está llena, se usará un ALGORITMO DE REEMPLAZO para decidir qué página se desalojará para cargar la nueva en el marco que ocupaba.

↳ Interesa no eliminar una página de uso frecuente, pues seguramente se referenciará otra vez rápidamente y habrá que volver a cargarla desde disco, lo cual es una operación muy lenta.

- Existen dos tipos de algoritmos de reemplazo:
  - Algoritmos **locales** → siempre seleccionarán una página del mismo proceso que la que se está intentando cargar.
  - Algoritmos **globales** → pueden seleccionar una página de cualquier proceso.

▸ El problema del reemplazo de páginas aparece también en otras áreas del diseño computacional como las memorias caché y los servidores web.

↳ En las memorias caché, la escala temporal del problema será mucho menor ya que sus fallos se resuelven en memoria principal, que es mucho más rápida que el disco.

ALGORITMO ÓPTIMO

○ Selecciona la página que se vaya a **referenciar más tarde** con objeto de **posponer el fallo** de página lo máximo posible.

- Es **imposible de implementar**, pues el SO no tiene manera de saber cuándo será la próxima referencia a cada página.

↳ Se puede implementar en un simulador en la segunda corrida utilizando la información de referencia de páginas recolectada durante la primera.

- Se usa como **referencia**, comparándolo con otros algoritmos que sí son implementables para averiguar su rendimiento.

NRU - NO USADAS RECIENTEMENTE

- Este algoritmo basa su funcionamiento en los bits referenciada (R) y modificada (M) de cada entrada de las TP:
  - Al iniciar un proceso, se establecen ambos a 0 en todas las entradas de la TP.
  - Se actualizan en cada referencia a memoria (lectura o escritura).
  - Una vez alguno de ellos se establece a 1, permanece así hasta que el SO lo restablece.
    - El bit R también se restablece en cada interrupción de reloj.
- Se dividen todas las páginas de cada proceso en 4 categorías:
  0. $R = 0$ y $M = 0$.
  1. $R = 0$ y $M = 1$.
  2. $R = 1$ y $M = 0$.
  3. $R = 1$ y $M = 1$.
- Selecciona una página al azar de la categoría de **menor numeración no vacía**.
- Tiene una **implementación** muy **sencilla**.

ALGORITMOS FIFO, SEGUNDA OPORTUNIDAD Y RELOJ

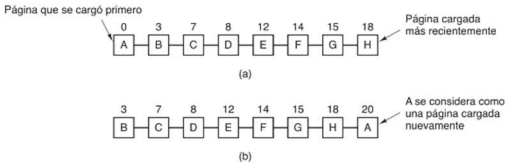
- El SO mantiene una lista de todas las páginas en memoria donde la más vieja estará al principio y la más nueva al final.

FIFO – Primera en entrar, primera en salir

- Selecciona la **primera página de la lista** y la nueva página se añade al final.
- Por tanto, se podría desalojar una página de uso frecuente.

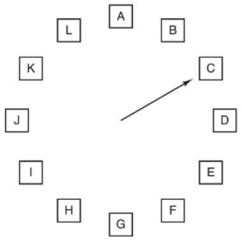
SEGUNDA OPORTUNIDAD

- Observa el bit R de la primera página de la lista:
  - Si $R = 0 \rightarrow$ se **selecciona**.
  - Si $R = 1 \rightarrow$ se lleva la página al final de la lista, se restablece R y se vuelve a empezar.
    - Evita el desalojo de páginas de uso frecuente.
- Si todas las páginas de memoria fueron usadas en el último intervalo de reloj, es decir, si todos los R de la lista están a 1, se comporta como el FIFO.



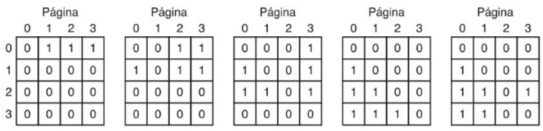
RELOJ

- La lista se mantiene en forma **circular** con una **manecilla** que apunta a la página más antigua.
  - Evita tener que estar moviendo constantemente las páginas en la lista.
- Observa el bit R de la página a la que apunta la manecilla:
  - Si $R = 0 \rightarrow$ se **selecciona**, se inserta la nueva página en su lugar y se avanza la manecilla una posición.
  - Si $R = 1 \rightarrow$ se restablece R y se avanza la manecilla una posición.



LRU - MENOS USADAS RECIENTEMENTE

- Se basa en el principio de **localidad temporal**.
- Selecciona la página que lleve **más tiempo sin ser referenciada**.
- Implementación por **software**:
  - Se crea una lista de páginas en memoria donde la última que ha sido usada estará al principio y la que más lleve sin usarse estará al final.
  - En cada referencia la página referenciada se mueve al frente de la lista.
  - La página a seleccionar será la primera de la lista.
  - Es una implementación muy cara, pues en cada referencia hay que buscar la página en la lista, eliminarla y después pasarla al frente.
- Implementación por **hardware sencilla**:
  - Se crea un contador C que se incrementa después de cada instrucción.
  - En cada referencia el valor actual de C se almacena en la entrada de la TP de la página referenciada.
  - La página a seleccionar será aquella con menor valor de C.
- Implementación por **hardware sofisticada**:
  - Se crea una matriz $n \times n$ inicializada a 0, donde n es el número de marcos de la memoria principal.
  - En cada referencia se establecen todos los bits de la fila correspondiente a la página referenciada a 1 y todos los de la columna a 0.
  - La página a seleccionar será aquella con el menor valor binario almacenado en su fila.

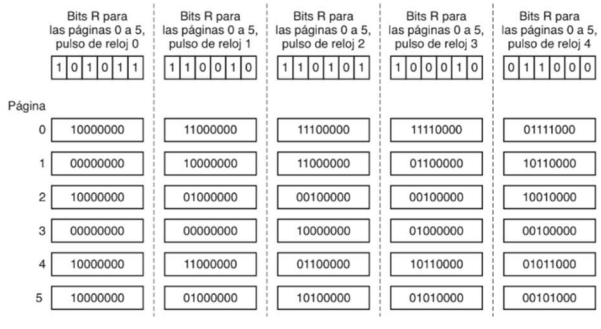


NFU – NO USADAS FRECUENTEMENTE

- Es una **variante del LRU** que sí se puede implementar en **software**.
- Se basa en asignar a cada página un contador (iniciado a 0) de manera que en cada interrupción de reloj el SO explora todas las páginas en memoria y le suma a su contador el valor de su bit R antes de restablecerlo.
  - Así, los contadores llevan una cuenta aproximada de la frecuencia con la que se usa cada página.
- Su principal **problema** es que no tiene en cuenta el tiempo que lleva sin ser utilizada cada página, sólo cuántas veces se ha usado.
  - Entonces, páginas antiguas que se usaron hace mucho tiempo y ya no se necesitan más tendrán privilegios sobre páginas nuevas que aún no tuvieron tiempo de ser usadas pero que la CPU necesita.

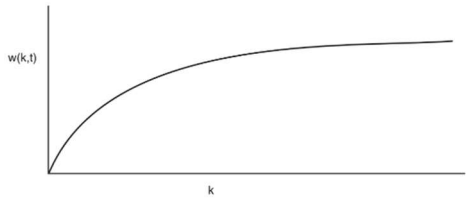
ENVEJECIMIENTO

- Cuando llega una interrupción de reloj, cada contador se desplaza un bit a la derecha y después se les agrega R a la izquierda.
- Selecciona la página con el **menor valor en el contador**.



CONJUNTO DE TRABAJO

- En los algoritmos explicados antes se usa **paginación bajo demanda**, es decir, las páginas se cargan según las va pidiendo la CPU, no por adelantado.
- Sin embargo, sabemos que la mayoría de procesos cumplen el principio de **localidad espacial** (o de referencia), que asegura que referencian una pequeña fracción de sus páginas.
  - Esta propiedad se puede aprovechar con el concepto del CONJUNTO DE TRABAJO (WS), que es el conjunto de páginas que referencia un proceso en un momento dado.
    - Si todo el WS de un proceso está en memoria → se ejecutará sin producir muchos fallos de página.
    - Si no todo el WS de un proceso está en memoria → producirá fallos de página cada pocas instrucciones (estará **sobrepaginado**), así que se ejecutará muy lentamente.
      - La sobrepaginación sucede cuando el WS no cabe en memoria principal, bien porque esta es muy pequeña o porque el proceso referencia muchas páginas distintas.
- Definimos el $w(k,t)$ de un proceso en el instante t como el conjunto de páginas utilizadas en sus últimas k referencias.
  - Su límite conforme k avanza con un t dado es finito, pues un proceso no puede referenciar más páginas que las que hay en su espacio de direcciones.



- Entonces, cada vez que el SO traiga un proceso a memoria para ejecutarlo, este producirá fallos de página hasta que se haya cargado su WS, desperdiciando tiempo de CPU.
  - Para evitar este desperdicio se usa el MODELO DEL CONJUNTO DE TRABAJO, que consiste en asegurarse de que el WS de un proceso esté en memoria antes de comenzar a ejecutarlo.
    - Se usa la **prepaginación**, se cargan las páginas del WS del proceso en memoria antes de empezar a ejecutarlo, es decir, antes de que las solicite la CPU.
    - El **algoritmo de reemplazo** seleccionará las páginas que no se encuentran en el WS del proceso.

POSIBLES IMPLEMENTACIONES

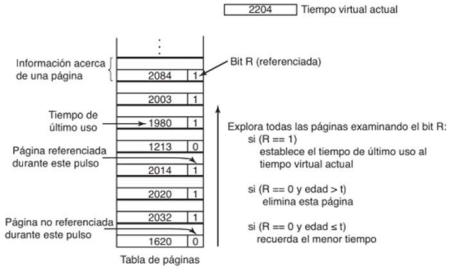
- Para poder implementar este modelo, el SO necesitará conocer el WS que tiene cada proceso en todo momento.
  - Una implementación que mantenga la definición de $w(k,t)$ de manera estricta es **inviable**, pues implica actualizarlo en cada referencia a memoria.
    - Se podría usar un registro de desplazamiento de longitud k que se corresponda con el $w(k,t)$ del proceso. En cada referencia a memoria se desplaza una posición a la izquierda e inserta el número de la página referenciada a la derecha.
      - Sin embargo, para hacer esto habría que recorrerlo, quitar todas las páginas duplicadas y ordenarlas otra vez en cada referencia, lo que sería muy costoso.
  - En su lugar, se usa una aproximación del modelo en la que se **redefine el WS** como el conjunto de páginas utilizadas durante los últimos τ segundos de tiempo virtual (tiempo de ejecución) del proceso.

ALGORITMO DE REEMPLAZO BASADO EN WS

- Como se dijo antes, selecciona una página que no esté en el WS.
- Cada entrada de la TP contendrá, al menos, el tiempo virtual en el que la página fue usada por última vez, el bit R y el bit M.
- Partimos de que τ abarca varios pulsos de reloj.

En cada fallo de página, se recorre la TP observando el bit R:

- Si $R = 1 \rightarrow$ la página fue usada en el último pulso de reloj, por lo que está en el WS, así que no es candidata.
- Si $R = 0 \rightarrow$ la página no fue usada en el último pulso de reloj, así que puede ser que sea candidata o no.
  - Se calcula la $edad$ de la página como $tiempo\ virtual\ actual - tiempo\ de\ último\ uso$.
    - Si $edad < \tau \rightarrow$ está en el WS, así que no es candidata.
    - Si $edad > \tau \rightarrow$ no está en el WS, se **selecciona** para desalojar.
- Si se explora toda la tabla sin encontrar ninguna página apta para seleccionarse, es que todas ellas están en el WS. Entonces, se seleccionará la página con $R = 0$ más antigua.
  - Si todas tienen $R = 1$, se seleccionará una al azar (de preferencia una con $M = 0$, si existe alguna).

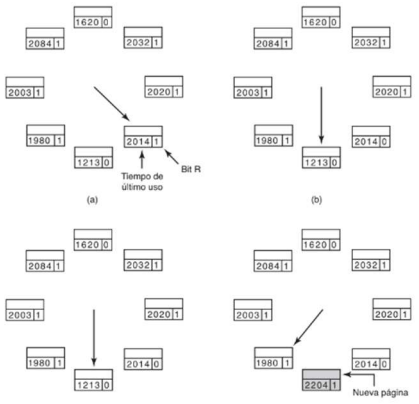


ALGORITMO DE REEMPLAZO WSCLOCK

- El algoritmo de reemplazo basado en WS básico es **ineficiente** pues implica explorar toda la TP de un proceso cada vez que se da un fallo de página.
- Para evitar esto, usaremos una **lista circular** en la que se almacenarán las entradas de la TP de las páginas que son residentes en memoria en un momento dado.

En cada fallo de página, se recorre la lista observando los bits R y M:

- Si $R = 1 \rightarrow$ la página fue usada en el último pulso de reloj, por lo que está en el WS, así que no es candidata.
  1. Se restablece el bit R.
  2. Se avanza la manecilla una posición.
- Si $R = 0 \rightarrow$ la página no fue usada en el último pulso de reloj, así que puede estar en el WS o no.
  - Si $edad < \tau \rightarrow$ está en el WS, así que no es candidata.
  - Si $edad > \tau \rightarrow$ no está en el WS, así que es candidata.
    - Si $M = 0 \rightarrow$ no está en el WS y tiene una copia en el disco, se **selecciona** para desalojar.
    - Si $M = 1 \rightarrow$ no está en el WS pero no tiene una copia en el disco.
      1. Se planifica la escritura en el disco.
      2. Se avanza la manecilla una posición.



RESUMEN

- Los dos **mejores** algoritmos son los de **envejecimiento** y **WSClock**, basados en LRU y WS respectivamente, pues dan un buen rendimiento y se pueden implementar con eficiencia.

Algoritmo	Comentario
Óptimo	No se puede implementar, pero es útil como punto de comparación
NRU (No usadas recientemente)	Una aproximación muy burda del LRU
FIFO (primera en entrar, primera en salir)	Podría descartar páginas importantes
Segunda oportunidad	Gran mejora sobre FIFO
Reloj	Realista
LRU (menos usadas recientemente)	Excelente, pero difícil de implementar con exactitud
NFU (no utilizadas frecuentemente)	Aproximación a LRU bastante burda
Envejecimiento	Algoritmo eficiente que se aproxima bien a LRU
Conjunto de trabajo	Muy costoso de implementar
WSClock	Algoritmo eficientemente bueno

CUESTIONES DE DISEÑO DE LOS SISTEMAS DE PAGINACIÓN

ASIGNACIÓN LOCAL CONTRA ASIGNACIÓN GLOBAL

- En las políticas de **asignación local** el algoritmo de reemplazo siempre desalojará una página del mismo proceso que la que se está intentando cargar.
  - Asignan a cada proceso una **cantidad fija de marcos** de memoria.
  - × Si en algún momento el tamaño del WS supera el número de marcos asignados, se produce sobrepaginación.
    Si en algún momento el tamaño del WS es menor que el número de marcos asignados se estará desperdiciando memoria principal.
- En las políticas de **asignación global** el algoritmo de reemplazo puede desalojar una página de cualquier proceso.
  - Asignan a cada proceso una **cantidad dinámica de marcos** de memoria que cambia conforme este se ejecuta.
  - ✓ Se pueden ajustar al tamaño del WS conforme este va cambiando.
  - Inicialmente, se asigna a cada proceso una cantidad de marcos proporcional a su tamaño.
    - ↳ Se garantiza un número **mínimo** de marcos para que los procesos muy pequeños se puedan ejecutar.
  - Las asignaciones aumentan o disminuyen conforme se ejecutan todos los procesos.
  - El **algoritmo PFF** (frecuencia de fallos de página) es el encargado de administrar las asignaciones, indicando cuándo se deben incrementar o decrementar.
    - La proporción de fallos disminuye conforme se asignan más páginas, así que el PFF se basa en medir el número de fallos por segundo de cada proceso.
    - En base a esto, establece un rango aceptable de proporción de fallos en el que intenta mantener todos los procesos.
      - El límite A establece una proporción de fallos demasiado alta ⟶ cuando un proceso lo sobrepasa se incrementa su número de marcos asignados.
      - El límite B establece una proporción de fallos tan baja que se puede suponer que el proceso tiene demasiada memoria ⟶ cuando un proceso lo sobrepasa se decrementa su número de marcos asignados.

- Algunos algoritmos de reemplazo, como FIFO o LRU, pueden funcionar con **asignación local o global**.
- Otros, los **basados en WS**, sólo pueden funcionar con **asignación local**.
  - ↳ Por definición, no existe un WS para todo el sistema, sólo para un proceso concreto. Al intentar definir un WS total se perdería el fundamento del WS, el principio de localidad.

CONTROL DE CARGA

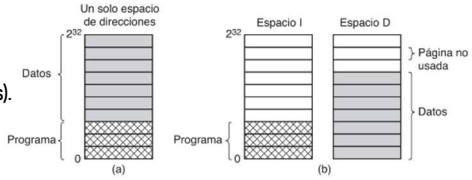
- Aunque se use el mejor algoritmo de reemplazo y una asignación óptima de marcos, el sistema sobrepaginará si la suma de los WS de todos los procesos en ejecución es más grande que la memoria principal.
  - ↳ Un síntoma de que ha sucedido esto es que el PFF determine que algunos procesos en memoria necesitan más marcos, pero no haya ninguno que necesite menos para poder cedérselos.
- La única solución es deshacerse temporalmente de algunos procesos **bloqueados, intercambiándolos a disco** para liberar sus marcos y que los puedan usar los que los necesitan.
  - Si aun así el sistema sigue sobrepaginado, se volverá a hacer hasta que deje de estarlo.
- Al aplicar esto se debe tener en cuenta el **grado de multiprogramación**, pues si el número de procesos en memoria es demasiado bajo, la CPU puede estar inactiva durante largos periodos de tiempo.

DETERMINACIÓN DEL TAMAÑO DE PÁGINA

- El tamaño de página es una **constante** que puede ser **elegida por el SO**, aunque esta elección difiera de la que se tomó en el diseño del hardware.
  - Tiene que ser una potencia entera de 2.
  - Normalmente será múltiplo de (o igual a) el tamaño de un sector del disco.
- En la determinación del mejor tamaño de página intervienen varios factores competitivos, así que **no hay un tamaño óptimo en general**.
  - Si se usa un tamaño de página **pequeño**:
    - ✓ Menor fragmentación interna (espacio no usado al final de las últimas páginas de los segmentos de texto, datos o pila).
    - × TP más grande.
  - Si se usa un tamaño de página **grande**:
    - ✓ TP más pequeña.
    - × Mayor fragmentación interna.
  - El tiempo de transferencia de una página desde disco **apenas depende de su tamaño** (depende sobre todo del retraso de búsqueda y rotacional).

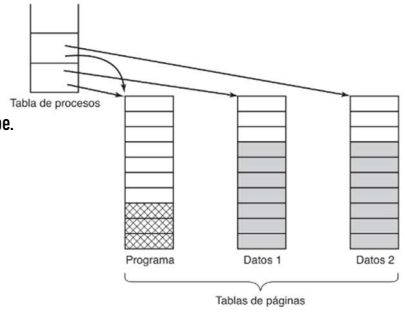
ESPACIOS SEPARADOS DE INSTRUCCIONES Y DATOS

- En la mayoría de sistemas cada proceso tiene **un único espacio** de direcciones que contiene tanto instrucciones como datos.
  - ↳ A menudo, este espacio es demasiado pequeño para contener toda esa información.
- Para solucionar esto se crean **dos espacios** de direcciones, el **espacio I** (para las instrucciones) y el **espacio D** (para los datos). Cada uno de ellos tendrá el mismo tamaño que el original, 2^n, por lo que se **duplicará el espacio** de direcciones disponible.
  - Ambos espacios se **paginan** de manera **independiente**
  - Cada uno tiene su propia TP con su propia asignación de páginas a marcos.



PÁGINAS COMPARTIDAS

- Cuando varios usuarios ejecutan el mismo programa, compartir sus páginas es mucho más eficiente que tener dos copias de la misma página en memoria todo el tiempo.
- Compartir las **páginas de código** (que son sólo de lectura) es muy **fácil**:
  - Si se admiten espacios de direcciones separados, los procesos con el mismo programa usarán la misma TP para su espacio I y distintas TP para los espacios D.
    - ↳ Para poder hacer esto cada proceso tendrá dos apuntadores en la tabla de procesos, uno para la TP del espacio I y otro para la TP del espacio D. Para compartir la TP del espacio I, se hará que sus apuntadores apunten a la misma tabla.
  - Si no hay espacios de direcciones separados, también se puede conseguir que dos procesos compartan las páginas de código, pero el mecanismo es más complicado.
  - El problema es que al terminar o intercambiar uno de los procesos que comparten código, se retirarán todas sus páginas de memoria, provocando que el otro tenga muchos fallos de página hasta que vuelva a traer su código.
    - ↳ Buscar en todas las TPs de todos los procesos antes de desalojar una página es muy caro, por lo que se usan estructuras de datos especiales para llevar cuenta de las páginas compartidas.
- Compartir las **páginas de datos** (que son de lectura y escritura) es más **difícil**:
  - Esto sucede en los procesos padre e hijo, que comparten tanto el código como los datos del programa.
    - Cada uno tendrá su propia TP, que apuntará al mismo conjunto de páginas para evitar tener que copiarlas.
    - Tan pronto como cualquiera de los dos realice alguna escritura, se hará una copia de la página en la que se escribe. Esto se conoce como **copiar en escritura**.
    - Así, aquellas páginas que nunca se modifican (incluyendo las del programa) no se copiarán.



BIBLIOTECAS COMPARTIDAS

En los sistemas modernos hay muchas **bibliotecas extensas** utilizadas por **muchos procesos**. Se pueden enlazar a los programas que las usan de dos maneras:

- **Enlace estático** ⟶ las funciones usadas de las bibliotecas se incluyen en tiempo de compilación en el binario ejecutable.
  - Las funciones de la biblioteca que no se llaman en el programa no se incluyen.
  - El binario ejecutable contiene todo el código necesario para ejecutarse de manera independiente.
  - ✕ Si muchos programas usan la misma biblioteca, se desperdicia mucho espacio tanto en el disco como en la memoria principal.
- **Enlace dinámico** ⟶ durante la compilación se incluye una pequeña rutina auxiliar para enlazar las funciones de la biblioteca al llamarlas en tiempo de ejecución.
  - Las funciones de la biblioteca se cargan en memoria cuando algún programa las llama por primera vez, por lo que las que no se llaman nunca no se cargan.
  - Una vez algún proceso las haya cargado, ya las pueden usar todos los demás que la tengan enlazada dinámicamente.

ARCHIVOS ASOCIADOS

- Un proceso puede emitir una syscall para **asociar un archivo a una porción de su espacio** de direcciones virtuales.
  - El archivo no se carga entero al momento de la asociación, se va cargando bajo demanda.

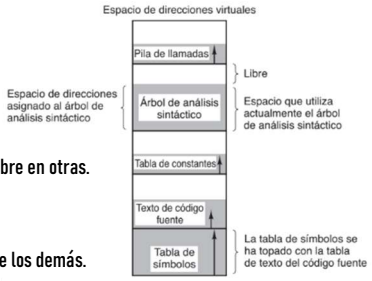
Los archivos asociados se usan como:

- **Método alternativo para la E/S** ⟶ en vez de realizar lecturas y escrituras se puede acceder al archivo como un array de datos en memoria.
- **Canal de comunicación entre procesos** ⟶ si varios procesos se asocian a la vez al mismo archivo, se pueden comunicar a través de la memoria que comparten.
  - Las escrituras que realice uno de ellos en la memoria compartida serán inmediatamente visibles por el resto.

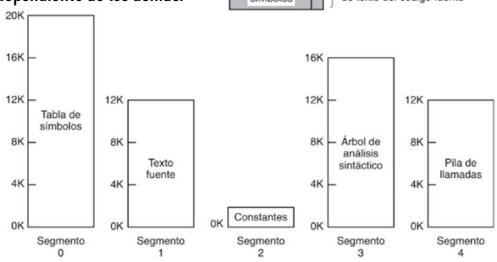
POLÍTICA DE LIMPIEZA

- La **paginación** funciona **mejor** cuando hay **muchos marcos libres** que se pueden reclamar nada más ocurra un fallo.
  Si todos los marcos están ocupados y han sido modificados, cuando se produzca un fallo se tendrá que realizar una copia en el disco antes de cargar la página que se necesitaba.
- Existe un proceso en segundo plano conocido como **demonio de paginación** que asegura que hay un conjunto abundante de marcos de página libres en memoria en todo momento.
  - Está inactivo la mayor parte del tiempo, pero se despierta de forma periódica para inspeccionar el estado de la memoria.
    - Si determina que hay pocos marcos libres, seleccionará páginas para desalojarlas.
    - Si las páginas desalojadas han sido modificadas, las copiará en el disco.
    - Como el desalojamiento no se hizo por un fallo de página, si la página se vuelve a necesitar se puede volver a reclamar mientras su marco no sean reclamado por otra página.
  - Así, asegura que todos los marcos libres están limpios, por lo que no se necesita escribir en el disco cuando se carga una página.

SEGMENTACIÓN



- Hasta ahora hemos analizado una memoria virtual **unidimensional**, en la que cada proceso tiene un único espacio de direcciones.
  - Los espacios de direcciones se dividen en varias partes cuyo tamaño cambia durante la ejecución del proceso, de manera que a una de ellas se le asigna un determinado espacio, por lo que una podría llenarse e impedir la carga de más datos mientras hay espacio libre en otras.
  - ↳ Se necesita un método que permita liberar al programador de tener que administrar la expansión de la memoria.
- Una solución simple es usar una memoria virtual **bidimensional** en la que cada proceso tenga varios **segmentos**
- Cada **segmento** es un espacio de direcciones, es decir, una secuencia lineal de direcciones desde 0 a un máximo, completamente independiente de los demás.
  - La longitud de cada segmento puede ser cualquier valor entre 0 y el máximo.
  - Distintos segmentos pueden (y suelen) tener distintas longitudes.
  - La longitud de cada segmento puede cambiar durante la ejecución del proceso.
  - ▸ Como cada segmento es un espacio de direcciones, puede crecer o decrecer sin afectar al resto.



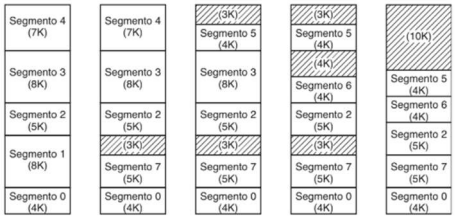
- Los segmentos son **unidades lógicas** de las que el **programador es consciente**.
  Un segmento puede contener un procedimiento, un array, una pila, etc., pero normalmente **no contendrá una mezcla** de estas cosas.
- Las **direcciones** de la memoria bidimensional se dividen en dos partes { número del segmento. / dirección dentro del segmento.

VENTAJAS

- Simplifica la administración de la expansión de la memoria.
- Permite vincular procesos más fácilmente ⟶ si cada procedimiento está en su propio segmento, cuando se modifique alguno y se vuelva a compilar de manera que su longitud cambie, esto no afectará a las invocaciones a otros, pues no modificará sus direcciones iniciales.
- Facilita la compartición de procedimientos, datos o bibliotecas entre procesos ⟶ se colocan en un segmento al que acceden todos los procesos que los comparten.
- Cada segmento tiene su tipo de protección ⟶ como en un segmento no se mezclarán datos con código, se asegura que los permisos sean los adecuados. En un sistema paginado no se puede garantizar que no se mezclan.

INCONVENIENTES

- Conforme se cargan y retiran segmentos en memoria, irán apareciendo huecos demasiado pequeños para que quepan la mayoría de segmentos, es decir, aparece **fragmentación externa**, se desperdicia memoria.
  - ▸ Se soluciona mediante la **compactación** de todos los huecos pequeños en uno grande.



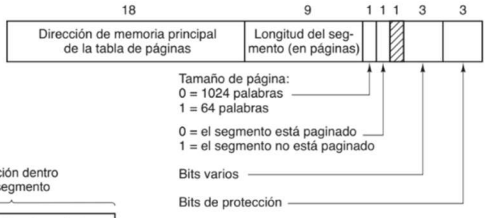
DIFERENCIAS ENTRE PAGINACIÓN Y SEGMENTACIÓN

- La principal diferencia entre la paginación y la segmentación es que **las páginas tienen un tamaño fijo y los segmentos no**.

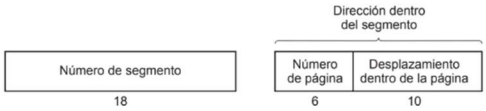
	Paginación	**Segmentación**
¿Por qué se inventó esta técnica?	Para obtener un espacio de direcciones lineales grande independientemente del tamaño de la memoria física.	Para permitir que los programas y datos se dividan en espacios de direcciones lógicamente independientes, ayudando a la compartición y protección.
¿Necesita el programador ser consciente de que se está usando esta técnica?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1 por proceso	Tantos como segmentos
¿Puede el espacio de direcciones total exceder el tamaño de la memoria física?	Sí	
¿Pueden los procedimientos y datos diferenciarse y protegerse por separado?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Pueden las tablas de tamaño cambiante acomodarse con facilidad?	No	Sí

SEGMENTACIÓN CON PAGINACIÓN

- Si los **segmentos** son **extensos**, puede ser muy difícil (o imposible) mantenerlos enteros en memoria principal.
  - Para solucionar esto, se usa la **segmentación con paginación**, en la que cada segmento se comportará como una memoria virtual paginada.
    - Así, sólo las páginas que se necesiten de cada segmento estarán en memoria principal.
- Pretende combinar las ventajas de la segmentación explicadas antes con las de la paginación (el tamaño de página es uniforme y no hay que mantener el segmento entero en memoria, sólo la parte de él que se usa).



  - Cada proceso tendrá una **tabla de segmentos**, con un **descriptor** (entrada) por cada uno de sus segmentos.
    - Como podría tener muchísimas entradas, la **TS** será también un **segmento** y como tal, estará **paginada**.
  - Un **descriptor** de un segmento indica si este está en memoria principal o no.
    - Se considera que un segmento está en memoria si cualquiera de sus páginas lo está.
    - Si un segmento está en memoria, su TP estará en memoria también.
    - Se produce un **fallo de segmento** cuando la TP del segmento no está en memoria.
- Un descriptor de una TS estará formado por:
  - Dirección en memoria principal de la TP del segmento.
  - Longitud del segmento.
  - Bits de protección del segmento.
  - Otros bits.



- Las direcciones de la memoria con segmentación paginada se dividen en { número del segmento. ; dirección dentro del segmento { número de página. ; desplazamiento en página. }

TRADUCCIÓN

1. Se observa el número de segmento para encontrar su descriptor en la TS.
2. Se comprueba si la TP del segmento está en memoria. Si no lo está, se produce un fallo de segmento. Si lo está, se continúa.
3. Se observa el número de página para encontrar su entrada en la TP. Si la página no está en memoria, se produce un fallo de página. Si lo está, se continúa.
4. Se extrae de la TP el marco de página en el que está la página buscada.
5. Se le **suma** el desplazamiento al marco de página.
6. Se realiza el acceso a memoria.