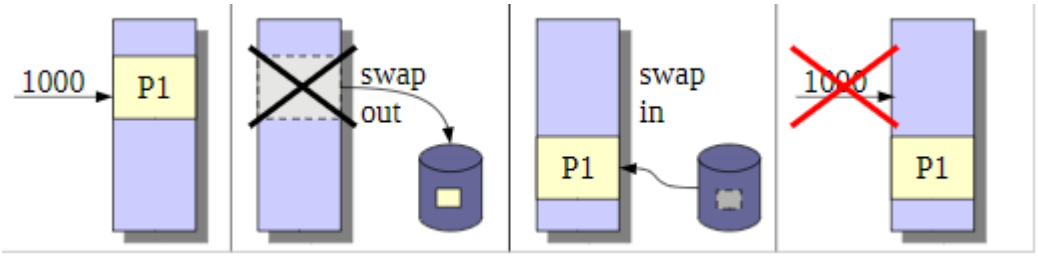
Administración de Memoria

Introducción:

Se encarga de llevar un registro de las partes de memoria que están ocupadas y libres. Además asigna espacio a los procesos. Se trata que la gran mayoría de procesos estén en memoria.

Introducción:

- Reubicación: el SO puede reubicar procesos en memoria (swapping , compactación). Un mismo proceso puede estar en diferentes secciones de la memoria durante su tiempo de vida, por lo que no se debe tener direcciones fijas atadas a un proceso



El proceso con dirección 1000 se mete en disco sacándolo de memoria (Swap Out) y luego se reubica desde disco a otra dirección de memoria (Swap In)

- Protección: los procesos no deben acceder a partes de memoria que no se les tiene permitido. No es posible verificar en tiempo de compilación o carga las direcciones ya que existe reubicación. Se debe chequear en tiempo de ejecución a través del hardware(por cuestiones de performance).

- Memoria Compartida: permitir a varios procesos que accedan a una porción de memoria sin sacrificar la protección de cada uno. Por ejemplo el manejo de Pipes, o bibliotecas que alojan funciones que pueden asociar cualquier proceso, como un print o malloc, entre otros.

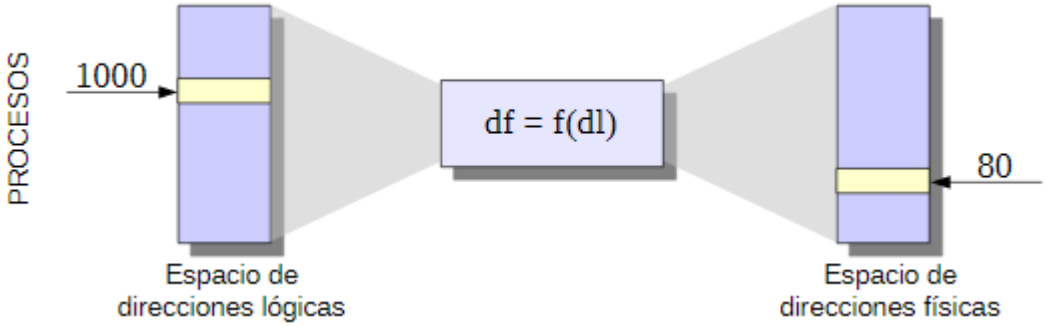
- Organización Lógica: de acuerdo a cómo esté definida la memoria. Estas pueden ser de solo lectura(códigos), solo escritura o lectura/escritura (datos) , memoria pública o privada.

- Organización Física: relacionado con la Jerarquía de Memoria. Se tiene que la memoria externa (Discos) alberga todos los programas guardados, y la Memoria Principal (RAM) posee el programa y/o procesos que se están ejecutando. Este manejo sólo debiera hacerlo el Administrador de Memoria, ya que si lo hace el programador puede provocar efectos no deseados.

Direcciones Lógicas y Físicas:

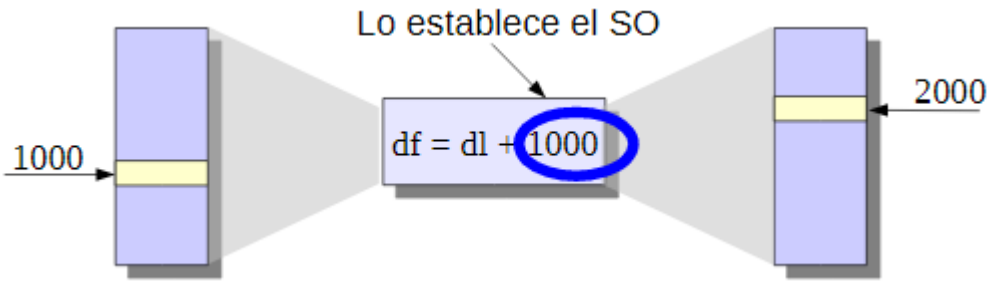
Para resolver el problema de las direcciones variables, por el tema de la reubicación, es **ocultar la verdadera ubicación de los procesos en memoria**, a través de **direcciones lógicas**.

Dirección Física: es un lugar físico de la memoria.
Dirección Lógica: referencia un lugar de memoria independientemente de la organización.



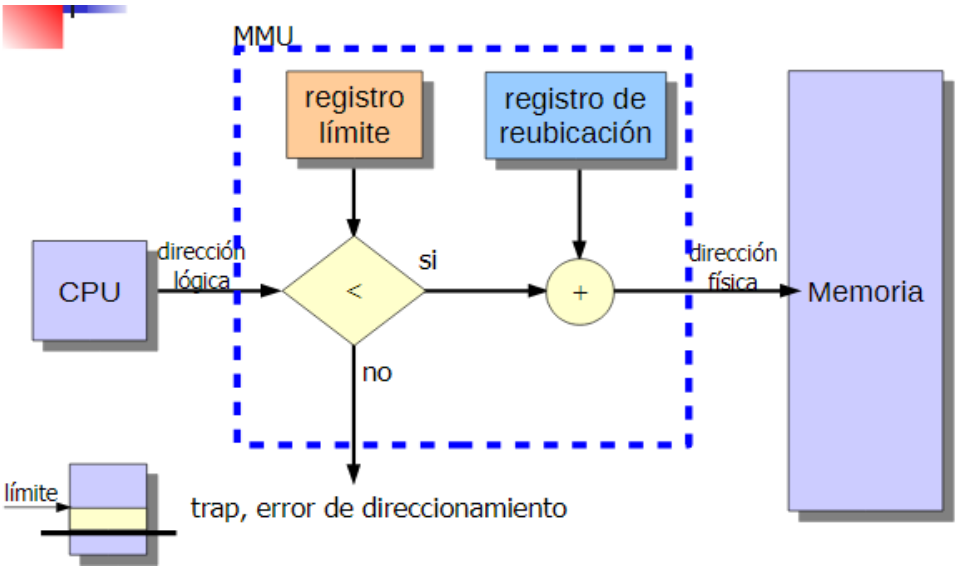
Para la obtención de la dirección física, se evalúa la dirección lógica en una función.

Ejemplo con Reubicación Base:



Unidad de Administración de Memoria (MMU)

A través de hardware, se genera esta unidad que traduce direcciones lógicas a físicas.



Con el comparador se aplica la protección, para que un proceso no vaya a parar a otra zona de memoria que no se le está permitido (a través del registro límite). Luego el registro de reubicación es aquella dirección "relativa a" para que traduzca la dirección lógica en física.

Carga de un Programa

Al cargar un programa se deben asignar instrucciones y datos a direcciones de memoria. En qué momento se hace esto:

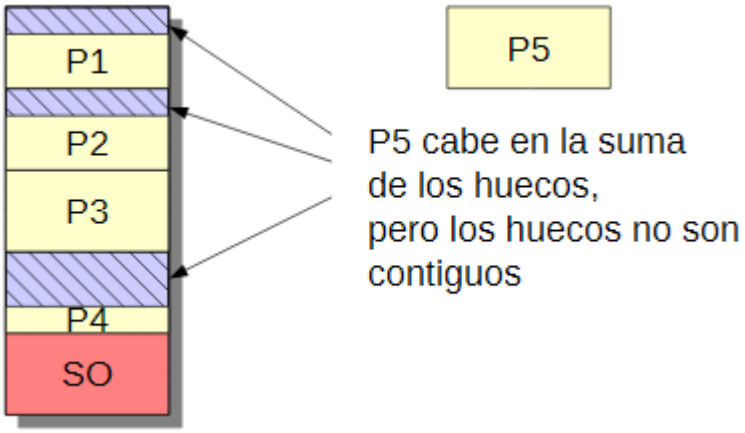
*Tiempo de Compilación: se debe conocer la dirección inicial a priori. No se permite hacer Reubicación porque es binding dinámico de almacenamiento (sin MMU).

*Tiempo de Carga: código relativo a una dirección (también sin reubicación ni MMU). Sin embargo sin esta unidad las traducciones se deben hacer por software, y esto degrada la performance.

*Tiempo de Ejecución: traducciones se atrasan hasta la ejecución para generar las reubicaciones necesarias (con MMU).

Asignación Contigua

Un proceso debe estar cargado de forma completa en una área contígua de memoria principal. Uno de los problemas que posee esta asignación es que al liberar procesos se van generando huecos en la memoria. Esta anomalía se llama **Fragmentación Externa**. Uno de los problemas es que al querer insertar un proceso con tamaño M, no entra en una memoria que posee espacio libre (sumando los huecos) da N, donde N > M. Sin embargo en el último hueco el tamaño es chico y no puede insertarse el proceso nuevo.

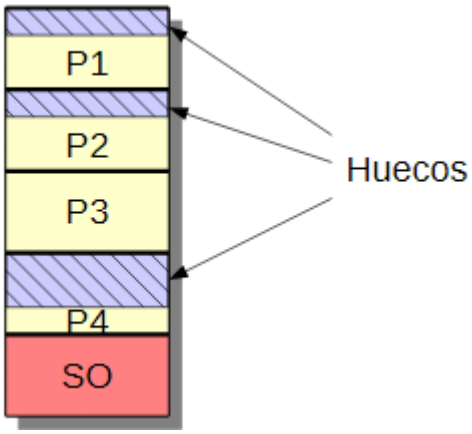


La solución trivial es generar compactación para eliminar los huecos intermedios y que queden más arriba. Sin embargo, compactar lleva mucho tiempo porque se requiere hacer muchas reubicaciones, y bloquea los procesos.

Particiones

Cuando se inserta un proceso se le crea una partición. Existen dos tipos de particiones. La estática y la dinámica.

*Particiones estáticas: las particiones no se solapan entre sí. Los tamaños pueden variar, pero una vez que se definen dichas particiones no se pueden modificar hasta hacer un rebooteo.
Para las asignaciones se procede de la siguiente forma: si un proceso entra en alguna de las particiones se inserta. Caso contrario se debe elegir un proceso para realizar el swapping y mandarlo a almacenamiento secundario por ejemplo.



Si se quiere ingresar un nuevo proceso, se vuelve a tener huecos y no puede entrar incluso cuando el tamaño del proceso sea menor a la memoria disponible. En este caso se tiene **Fragmentación Interna** porque se da entre particiones. Es peor que la F.Externa porque ni siquiera se puede compactar los procesos porque cada uno está asociado a una sola partición.

Existen dos escenarios, que poseen equitativamente un Trade-Off de uno respecto del otro:

-Se podría asociar a cada partición una cola de procesos de tamaños apropiados para que se adapte a dicha porción de memoria. Beneficia en que la fragmentación interna va a ser baja, pero baja la multiprogramación, porque al asignar procesos a una misma partición es más lento, hasta que uno pase a estado bloqueado/finalizado.

-El escenario actual, que solo se tiene una cola de procesos y va insertando los procesos a la partición que más le convenga usando un algoritmo de inserción como el Best Fit. Mejora la multiprogramación, pero habrá mayor fragmentación interna

*Particiones dinámicas: al crearse un nuevo proceso se le asigna un espacio lo suficientemente grande para contenerlo. Esto lo irá haciendo sucesivamente para cada uno. Si un nuevo proceso entra en una partición que ya existe, se inserta ahí. El dinamismo está presente ahí y también en el caso de que si uno saca de memoria el proceso, se elimina también la partición.
Ahora la pregunta es que si un proceso se tiene que insertar, y puede entrar en más de una partición, en cuàl elegir. Para esto se tienen los siguientes algoritmos de asignación de espacio:

-First-Fit: lo asigna a la primera partición con memoria suficiente. El problema de este algoritmo es que genera mucha fragmentación interna.

-Next-Fit: lo asigna a la última partición que realizó la asignación, sino aplica First-Fit. Genera mayor fragmentación interna que el anterior.

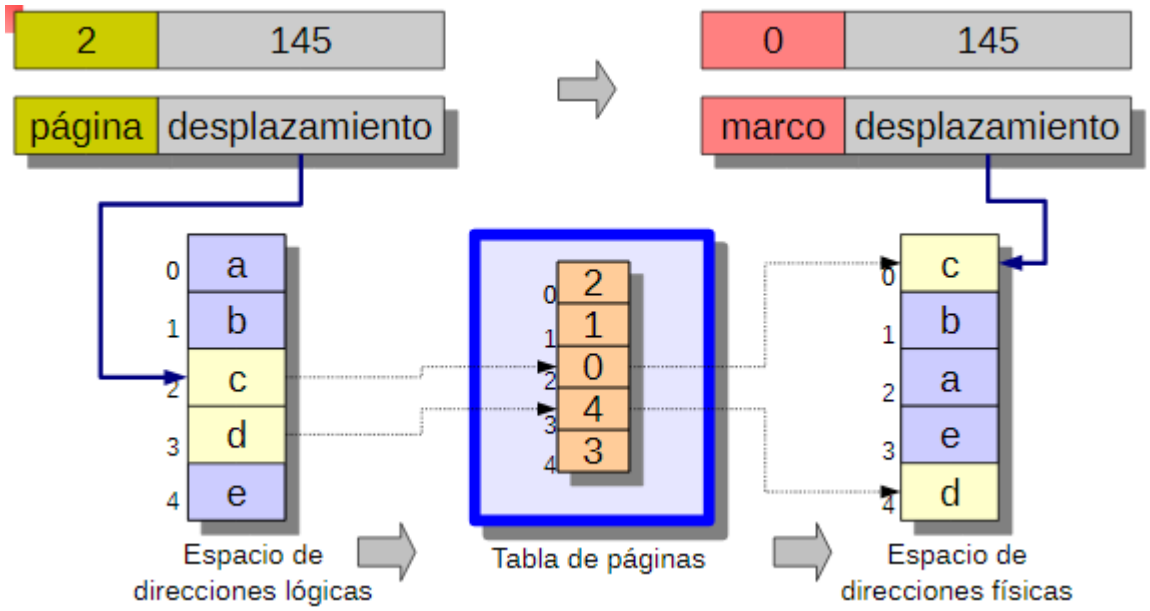
-Best-Fit: asigna el proceso a la partición que mejor se ajuste. La fragmentación interna es más baja, pero es muy lento.

-Worst-Fit: asigna el proceso a la partición que peor se ajuste. La fragmentación interna es mayor, y es muy lento.

Pregunta: Por qué al final de la filmina de particiones variables se tiene Fragmentación Externa y no Interna?

Paginación

Para las asignaciones de memoria que se vieron anteriormente, todas tenían un problema en común que era sobre las consecuencias de que un proceso pueda llegar a crecer. La paginación se encarga de dividir la memoria física en bloques de tamaño fijo llamados **marcos**. También se encarga de dividir la memoria lógica de la misma forma que la memoria física, y ahora estos bloques se llaman **páginas**. Entonces la traducción será de páginas a marcos.

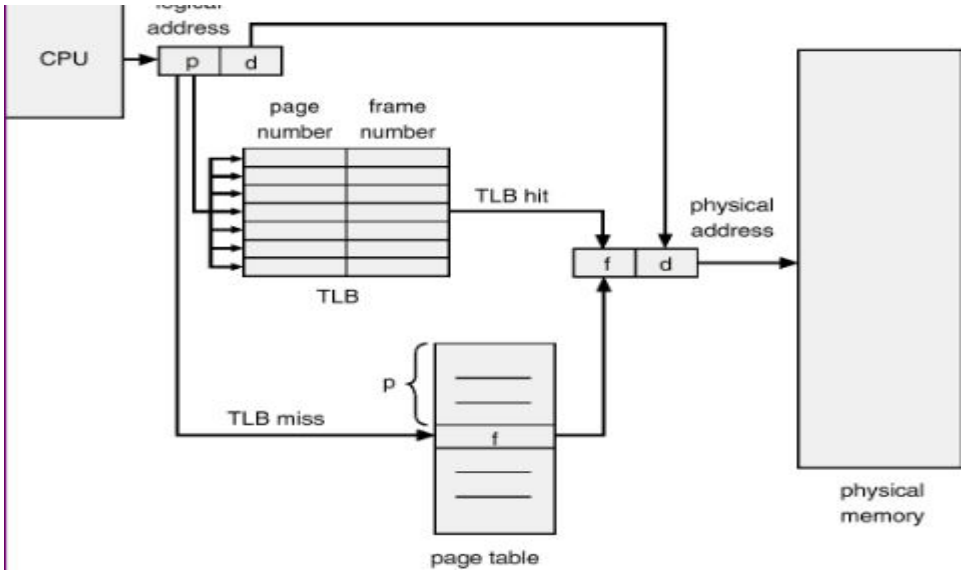


Como se observa, la página sería 2, y su traducción sería el marco 0. Como la página/marco debe vinculado en la traducción debe ser del mismo tamaño, se refleja en el offset que coinciden.

La paginación permite que el SO tema mejor libertad para organizar la memoria, si bien desaparece la fragmentación externa, sigue habiendo fragmentación interna, aunque en estos casos es despreciable si las páginas son chicas. El principal problema de la paginación es que la tabla también se debe guardar en memoria, y esta traducción puede llevar mucha memoria, y no alcance el tamaño para guardarla.

Implementación de las Tablas de Páginas

La tabla de página se debe guardar en la memoria principal, y el problema que esto conlleva, además del espacio requerido, es que se realizan 2 accesos a memoria principal, uno para la instrucción en sí, y otro para la tabla de página. Para solucionar esto, se puede hacer uso de una memoria asociativa como una caché o una TLB que contenga las direcciones de la instrucción, y así se haría un solo acceso que contenga todo lo necesario.



Lo que es importante recalcar es que no siempre se tendrá éxito conseguir esa dirección en el TLB. Si falla, se debe meter en la tabla de página para obtener la dirección. Esto es similar a jerarquías de memoria cuando uno si no encuentra la dirección en la Caché, se debe ir a la RAM a obtener la dirección.

Eficiencia de tiempo de acceso

Asumiendo que el tiempo de acceso a la RAM es de 1 microseg., el análisis depende si se tiene éxito o no encontrar la dirección en el TLB. Entonces si α es la probabilidad de éxito y ε el tiempo que tarda en buscar en el TLB, se considera qué:

*Se entra al TLB con un tiempo ε , si tiene éxito α la dirección lógica se traduce y se accede 1 vez a la RAM.

*Se entra al TLB con un tiempo ε , si tiene falla 1- α se debe ir a la tabla de páginas que gasta 1 tiempo ya que dicha tabla está en RAM, al encontrarla se accede nuevamente a RAM con 1 acceso más.

Por lo tanto la eficiencia se calcula como:

EAT: $(1+\varepsilon)\alpha + (2+\varepsilon)(1-\alpha) = 2 + \varepsilon - \alpha$

Para aumentar la probabilidad de éxito se debería aumentar el tamaño de la caché, pero esto es costoso.

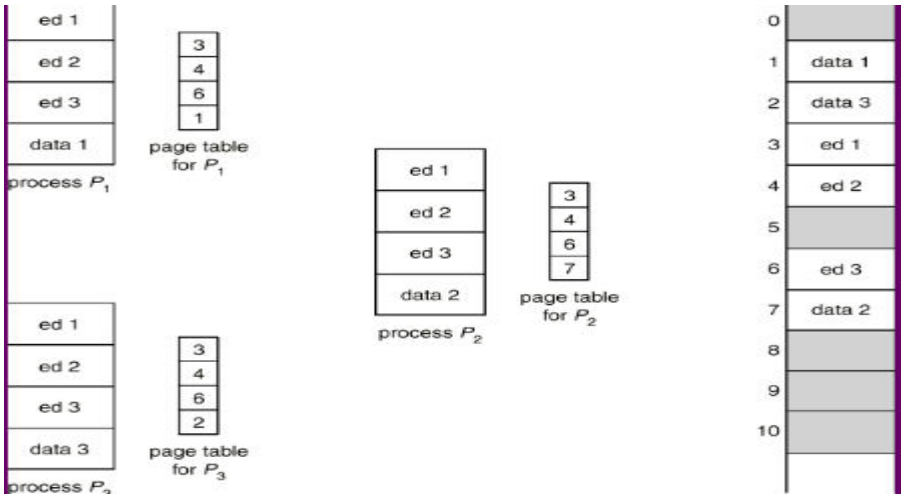
Protección de Memoria

Se implementa asociando un bit de protección para cada marco. Si este bit es válido, indica que este marco

existe, en caso contrario no existe. En estos casos el existir implica que en un marco particular esté presente algún proceso o no. Esto significa que cuando un nuevo proceso arriba a un bloque de la memoria principal, en la tabla de página de pone como válido a este bit. Caso contrario, si un proceso se desaloja de la RAM, se pone como inválido al bit del marco correspondiente.
Esta validación pega en el CPU que se encarga de operar ese bit de validez, y también en la estructura del MMU que ahora tiene un bit más por cada página a validar.

Páginas Compartidas

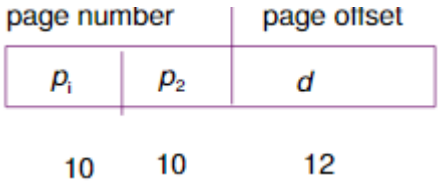
En general se puede una copia de sola lectura para muchos procesos que compartan ciertas cantidades de páginas, como editores de texto, compiladores, sistema de ventanas, etc.

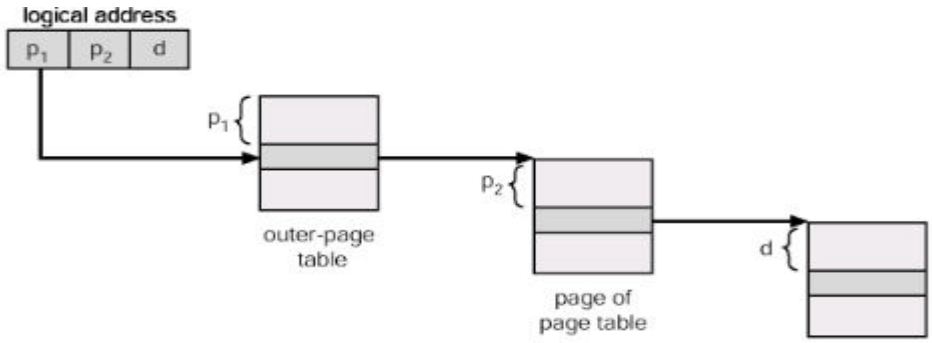


Estructuras de la Tabla de Páginas

Existen algunas formas elegir estructuras para las tablas de páginas, depende para qué se quiera tener de ganancia.

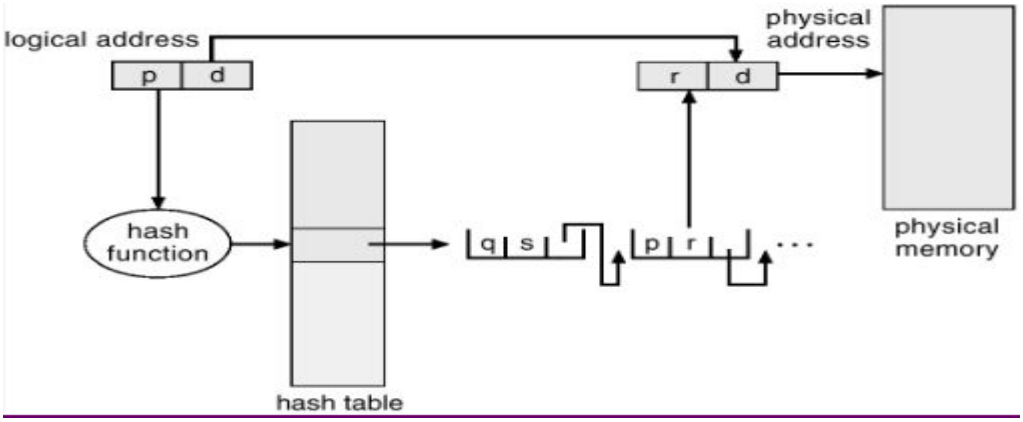
*Jerárquica: se encarga de dividir el espacio de direccionamiento lógico en múltiples niveles. El caso más simple en dividirla en dos.



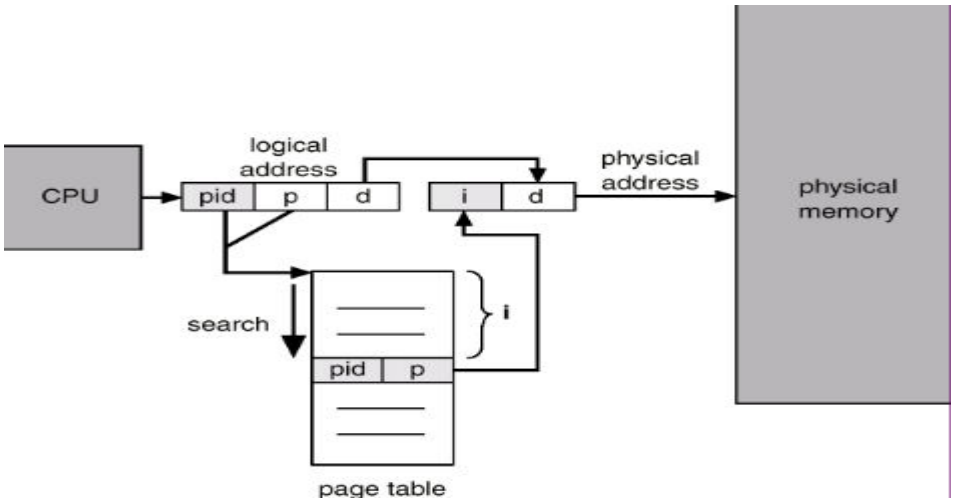


Sale perdiendo la cantidad de accesos a RAM que se debe hacer para ir obteniendo esto. Sin embargo si las tablas intermedias se meten en una caché, es más rápido, y al tener esta disposición jerárquica el hit rate aumenta. También se pierde en que el CPU debe hacer más cálculos y el MMU es más complejo.

*Hashing: la búsqueda en la tabla se hace a través de una función de Hashing. El gran problema que existe en este caso es que para una función de hashing podría dar más de una vez el mismo número, entonces en la tabla de página se debería tener una lista de factoreo para evitar caer en problemas de clustering. Pero nada garantiza que esa lista sea muy larga, y eso degrada la pérformance de la traducción. En el peor de los casos debe ir al final de esa lista, y luego el acceso a la RAM.

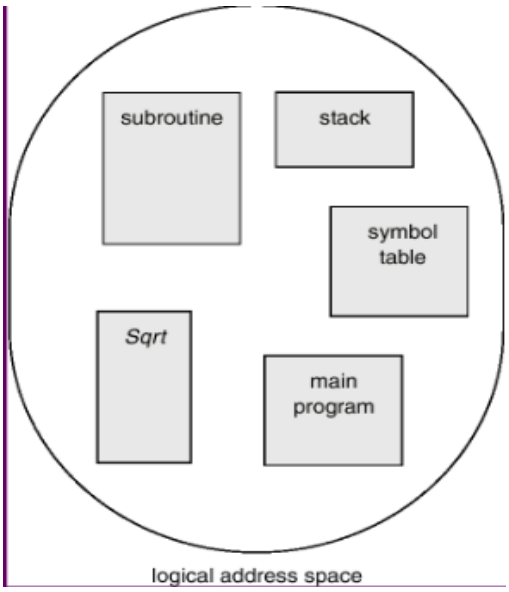


*Invertida: se realiza una búsqueda secuencial en la tabla de página, sin indexar. Poco frecuente.



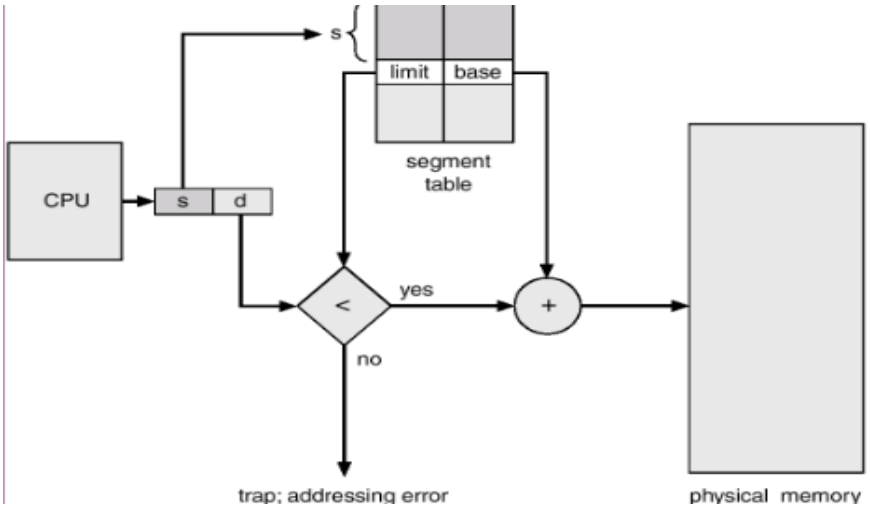
Segmentación

Es un esquema de manejo de memoria que soporta la vista de usuario de la memoria. Entonces esta separación está dividida por elementos de un programa, como un procedimiento/función, variables, stack, etc.



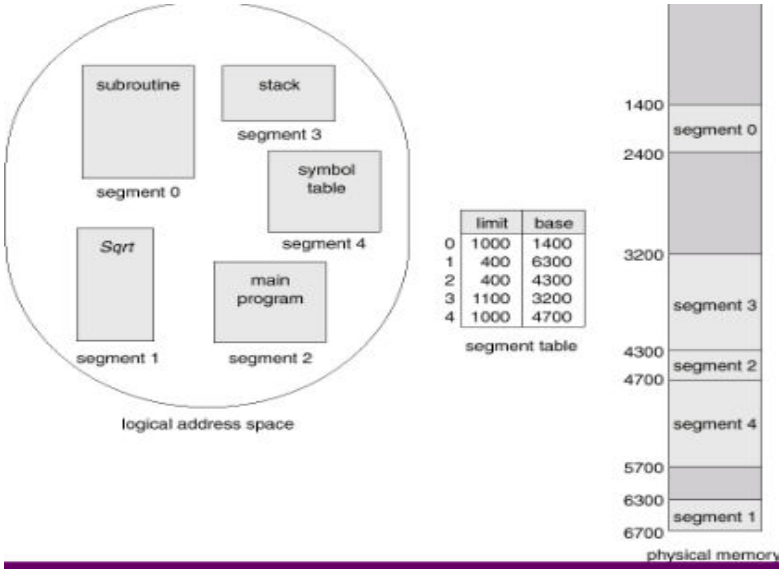
Ahora la dirección lógica está compuesta por el identificador de segmento y el desplazamiento de siempre. Depende de la tabla que se tenga se necesita saber la base y/o límite de ese segmento para saber de donde

partir. Respecto de la protección funciona similar que en las páginas con el bit de validación.



Por debajo tiene el mismo MMU de siempre, y arriba cambia la tabla, que ahora es de segmentación

Ejemplo básico de Segmentación:



Viendo esta imagen, en la base se indica el inicio del segmento. Y en el limit se marca qué tan grande es.

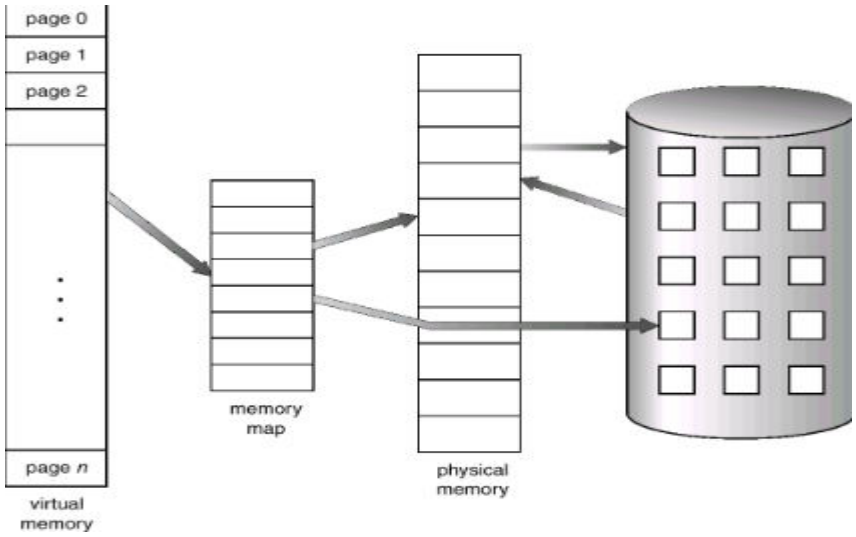
El problema de la segmentación es que retrocede en la solución que se había propuesto con paginación que es eliminar la fragmentación externa, y ahí sigue existiendo esto porque si se quiere meter un nuevo proceso en el ejemplo anterior, de la suma de los huecos que hay, no se podría.

Entonces una de los soluciones definitivas que se plantea es generar segmentación paginada. Esto significa generar segmentación, y luego a cada uno de los segmentos dividirlos en páginas, y luego se haría la traducción como en páginas/marcos. Esto lo tiene incorporado la familia de las i386.

Memoria Virtual

Introducción

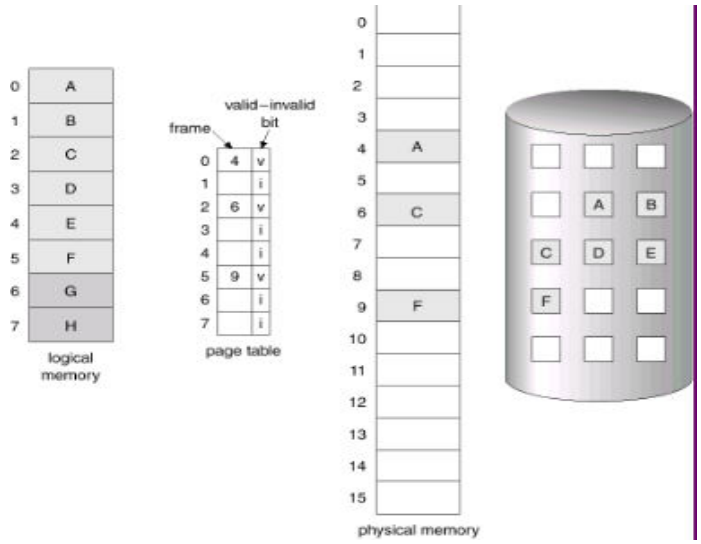
El concepto de generar traducciones de direcciones lógicas a físicas se sigue manteniendo en memoria virtual. Pero ahora se tiene un nuevo agregado: se le hace creer también a un dicho proceso que el espacio lógico en donde está "insertado" es mucho más grande que el espacio físico.Esto permite compartir que en una mismo espacio de direcciones esté compuesto por muchos procesos.
La implementación de la memoria virtual se realiza mediante demanda por paginación o por segmentación.



Como se ve en la imagen, la memoria virtual es "mucho más grande" que la RAM y Disco. La realidad es que no es así, en algún momento esto se va a llenar. En el mapa de memoria se distingue qué página corresponde a la memoria principal, y cual al almacenamiento secundario

Paginado por demanda

Brinda un espacio a memoria solo cuando es necesario. Lo que genera esto que es no necesita de de procesos de I/O y sin necesidad de memoria. La necesidad se dispara cuando un programa se transforma en un proceso, ya que necesita lectura/escritura. En caso de que la referencia sea inválida, se aborta, y si no está en memoria, se le asigna.

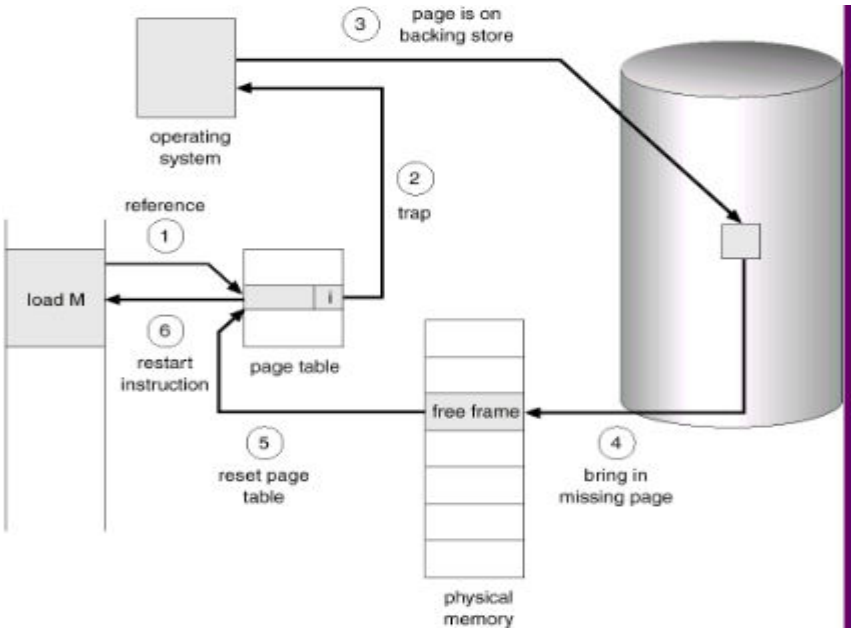


Siguiendo este ejemplo, se puede pensar de la siguiente manera:

"En la ejecución de un código, va a buscar el proceso en la primer página, y al haber un bit inválido se produce un **fallo de página**. Las consecuencias de eso es que la tiene que ir a buscar al disco, y resulta que es la A, entonces la termina insertando en algún lugar de la RAM. Esto genera que se realice la modificación en la tabla de página"

Por lo tanto se deduce que un fallo de página es el principal disparador de ir a buscar bloques a disco.

Diagrama del manejo que realiza el fallo de página:



Performance de Paginado por demanda

A priori se puede decir que el fallo de página puede estar asociado a un porcentaje de cuándo puede producirse. Entonces si ese porcentaje se define como p, si vale 0, nunca tendrá fallas, en caso de que valga 1 tendrá siempre. La consideración es la siguiente:

*Si no se produce un fallo de página (1-p) se accede a la memoria a ejecutar el proceso.

*Si se produce un fallo de página (p) se tiene un overhead al realizar un cambio de modo usuario a modo kernel, entonces de acuerdo al marco que se eliga como victima para hacer el cambio, se realiza un swap out de ese marco al disco, y luego en almacenamiento secundario se busca el proceso buscado y se realiza un swap in desde el disco a la RAM. Al final, antes de realizar el cambio se realiza el cambio de modo kernel a modo usuario.

Por lo tanto, la eficiencia de tiempo de acceso es:

EAT = (1 – p)* 1 acceso a memoria + p (page fault overhead+ swap page out + swap page in+ restart overhead)

Si se quiere aumentar el rendimiento de esto se debe hacer inferencia en los swap in/out ya que los tiempos en los cambios de modo son despreciables al lado de los movimientos RAM/disco.

Creación de Procesos

La memoria permite otras ventajas a la hora de una creación de procesos. Estas son:

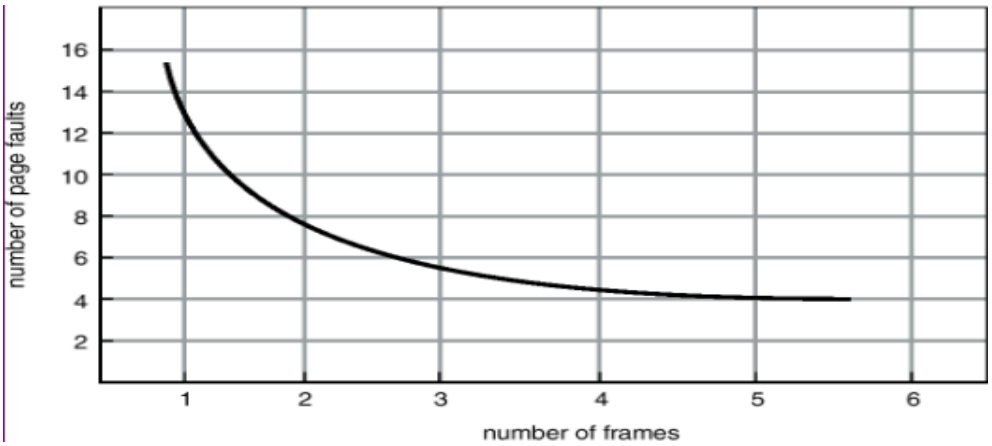
*Copy-on-Write: permite que un proceso padre e hijo inicialmente compartan las mismas páginas. Si un

proceso modifica esa página, solo se modifica la página.

También existe otra forma que es a través de Memory-Mapped Files (no es necesario para este examen)

Reemplazo de Páginas

Se busca que el fallo de página sea lo más bajo posible. Para esto existen distintos algoritmos que permiten ir seleccionando marcos víctimas de acuerdo a lo que priorice el algoritmo en cuestión.



Se supone que a mayor cantidad de marcos en una memoria RAM, menor número de fallo de páginas habrá.

Para explicar los algoritmos se usará la siguiente cadena de caracteres, que representarían marcos:

1 2 3 4 1 2 5 1 2 3 4 5

*FIFO (First IN - First OUT) : el primero que entra, es el primero que sale. Por ejemplo si se tienen 3 marcos:

Pasada 1: 1 FP=1

Pasada 2: 1 2 FP=2

Pasada 3: 1 2 3 FP=3

Pasada 4: 4 2 3 FP=4

Pasada 5: 4 1 3 FP=5 (en la pasada 4 se hizo swap out de 1, y ahora swap in) INGENUO

Pasada 6: 4 1 2 FP=6 (en la pasada 5 se hizo swap out de 2, y ahora swap in) INGENUO

Pasada 7: 5 1 2 FP=7

Pasada 8: 5 1 2 FP=7 (como el 1 ya está en la RAM, no hay fallo de página)

Pasada 9: 5 1 2 FP=7 (como el 2 ya está en la RAM, no hay fallo de página)

Pasada 10: 5 3 2 FP=8

Pasada 11: 5 3 4 FP=9

Pasada 12: 5 3 4 FP=9 (como el 5 ya está en la RAM, no hay fallo de página)

Ahora, si se tuviera 4 marcos:

Pasada 1: 1 FP=1

Pasada 2: 1 2 FP=2

Pasada 3: 1 2 3 FP=3

Pasada 4: 1 2 3 4 FP=4

Pasada 5: 1 2 3 4 FP=4 (como el 1 ya está en la RAM, no hay fallo de página)

Pasada 6: 1 2 3 4 FP=4 (como el 2 ya está en la RAM, no hay fallo de página)

Pasada 7: 5 2 3 4 FP=5

Pasada 8: 5 1 3 4 FP=6

Pasada 9: 5 1 2 4 FP=7

Pasada 10: 5 1 2 3 FP=8

Pasada 11: 4 1 2 3 FP=9

Pasada 12: 4 5 2 3 FP=10

La curiosidad que se da en este caso es que a mayor frames, más fallos de página hay, y esto en general no es así. Esto es un problema que se presenta en el algoritmo de FIFO llamado **Anomalía de Belady**.

*Algoritmo Óptimo: este algoritmo al principio funciona igual que el FIFO, pero luego cuando entra un nuevo proceso y tiene que elegir una victima, se para a partir de ahí en adelante y ve cuàl de todos los procesos existentes en ese momento en la tabla de páginas es el que se llama más adelante.

Pasada 1: 1 FP=1

Pasada 2: 1 2 FP=2

Pasada 3: 1 2 3 FP=3

Pasada 4: 1 2 3 4 FP=4

Pasada 5: 1 2 3 4 FP=4 (como el 1 ya está en la RAM, no hay fallo de página)

Pasada 6: 1 2 3 4 FP=4 (como el 2 ya está en la RAM, no hay fallo de página)

---------- Hasta acá igual que que FIFO----------

Ahora al querer insertar el 5, se fija los elementos a futuro que son: 1 2 3 4 5
de los elementos que hay presentes en la tabla (1,2,3,4) el algoritmo elige el 4 porque es el más lejano que se llama, entonces:

Pasada 7: 1 2 3 5 FP=5

Pasada 8: 1 2 3 5 FP=5 (como el 1 ya está en la RAM, no hay fallo de página)

Pasada 9: 1 2 3 5 FP=5 (como el 2 ya está en la RAM, no hay fallo de página)

Pasada 10: 1 2 3 5 FP=5 (como el 3 ya está en la RAM, no hay fallo de página)

Pasada 11: 4 2 3 5 FP=6 (de 1,2,3,5 se descarta 5 porque es el próximo a llamar y como 1,2,3 ya no se llamarán se elige el 1 porque es el que más tiempo estuvo)

Pasada 12: 4 2 3 5 FP=6 (como el 5 ya está en la RAM, no hay fallo de página)

La diferencia que generó el algoritmo óptimo con el FIFO fue en la pasada 7, porque el óptimo elige el proceso que se llamará más tarde porque al mantener los procesos que pronto serán llamadas, no provocarán fallos de página.

*LRU(Menos recientemente usando): el algoritmo se aplica en el momento que hay un fallo de página y se quiere insertar un proceso que no está en la tabla y no hay marcos disponibles (al igual que FIFO y Opt). En este caso, desde el momento que se quiere ver el reemplazo, se mirará hacia atrás viendo cuál de los procesos que están en la memoria fue el primero en insertarse, y a ese se le hará el swap out para hacer el swap in con el nuevo.

Al igual que FIFO y Opt:

Pasada 1: 1 FP=1

Pasada 2: 1 2 FP=2

Pasada 3: 1 2 3 FP=3

Pasada 4: 1 2 3 4 FP=4

Pasada 5: 1 2 3 4 FP=4 (como el 1 ya está en la RAM, no hay fallo de página)

Pasada 6: 1 2 3 4 FP=4 (como el 2 ya está en la RAM, no hay fallo de página)

Se quiere insertar el 5 y genera un fallo de página, entonces el LRU mira la lista de los anteriores que es 1-2-3-4-1-2 . Y de los presentes en la tabla de página (1,2,3,4) el menos recientemente llamado es el 3, entonces se reemplaza por ese número:

Pasada 7: 1 2 5 4 FP=5

Pasada 8: 1 2 5 4 FP=5 (como el 1 ya está en la RAM, no hay fallo de página)

Pasada 9: 1 2 5 4 FP=5 (como el 2 ya está en la RAM, no hay fallo de página)

Pasada 10: 1 2 5 3 FP=6 (se aplica LRU, y viendo la lista 1-2-3-4-1-2-5-1-2 del {1,2,5,4} el 4 es el menos recientemente llamado)

Pasada 11: 1 2 4 3 FP=7 (se aplica LRU, y viendo la lista 1-2-3-4-1-2-5-1-2-3 del {1,2,5,3} el 5 es el menos recientemente llamado)

Pasada 12: 5 2 4 3 FP=8 (se aplica LRU, y viendo la lista 1-2-3-4-1-2-5-1-2-3-4 del {1,2,4,3} el 1 es el menos recientemente llamado)

Por cómo se dio esta secuencia de números, el LRU no fue para nada bueno, porque entre las pasadas 11 y 12 se generaron fallos de página porque el algoritmo fue haciendo swap out a procesos que se iban a llamar a continuación.

*LFU(Menos frecuentemente usado): es un algoritmo muy similar al LRU, con la diferencia que el proceso a desalojar es el que menos se haya usado. Si hay más de un proceso con la misma frecuencia, se aplica LRU.

Pasada 1: 1 FP=1

Pasada 2: 1 2 FP=2

Pasada 3: 1 2 3 FP=3

Pasada 4: 1 2 3 4 FP=4

Pasada 5: 1 2 3 4 FP=4 (como el 1 ya está en la RAM, no hay fallo de página)

Pasada 6: 1 2 3 4 FP=4 (como el 2 ya está en la RAM, no hay fallo de página)

Se tiene que insertar el proceso 5, y aplicando LFU se cuenta con los procesos en memoria y su frecuencia: (1 -> Frec=2 ; 2 -> Frec=2; 3 -> Frec=1 ; 4 -> Frec=1), los procesos 3 y 4 poseen menor frecuencia, entonces aplicando LRU entre los dos, el proceso 3 es el menos recientemente llamado. Entonces se hace el reemplazo.

Pasada 7: 1 2 5 4 FP=5

Pasada 8: 1 2 5 4 FP=5 (como el 1 ya está en la RAM, no hay fallo de página)

Pasada 9: 1 2 5 4 FP=5 (como el 2 ya está en la RAM, no hay fallo de página)

Ahora se tiene que insertar el proceso 3, y los procesos en RAM con su frecuencia son:
(1-> Frec=3 ; 2-> Frec=3 ; 5-> Frec=1 ; 4-> Frec=1) como 4 y 5 poseen la misma frecuencia, por LRU se sacrifica el proceso 4.

Pasada 10: 1 2 5 3 FP=6

Pasada 11: 1 2 4 3 FP=7 (Por LFU, 5 y 3 tienen Frec=1, pero 5 es más viejo)

Pasada 12: 1 2 4 5 FP=8 (Por LFU, 4 y 3 tienen Frec=1, pero 3 es más viejo)

Para esta secuencia tuvo los mismos fallos de página que en LRU, y casi los mismos efectos en los procesos que iban entrando a memoria. El cambio se dio recién en la última pasada.

Asignación de Procesos

La idea consiste en cuántos marcos se asignan a cada proceso. Existen algunas maneras de dividir la memoria.

*Asignación fija:

-Asignación equitativa: por ejemplo si se tienen 100 marcos y 5 procesos, a cada uno se le da 20 páginas.

-Asignación proporcional: un proceso se asigna de acuerdo a su tamaño. Si se quiere insertar un proceso P_i con un tamaño S_i y el tamaño del marco es S:

$P_i : S_i / S$

*Asignación por prioridad: usa asignación proporcional teniendo en cuenta prioridades.

*Asignación Global y Local