GESTION DE MEMORIA

La memoria principal (RAM) es un recurso que se debe administrar. Los sistemas operativos crean abstracciones de la memoria y las administran. Básicamente llevan el registro de cual es partes de la memoria están en uso, asignan memoria a los procesos cuando lo necesiten y desasignan cuando estos terminen.

Administrador de Memoria

Su trabajo es administrar la memoria con eficiencia: llevar el registro de cuáles partes de la memoria están en uso, asignar memoria a los procesos cuando la necesiten y desasignarla cuando terminen.

Necesidad:

- Se ejecutan programas solo en memoria principal
- Se mejora la utilización de la CPU usando multiprogramación
- Por lo tanto, se deben mantener VARIOS procesos almacenados en memoria principal simultáneamente

Funciones:

- Saber que partes están en uso y cuales no
- Asignar más memoria a los procesos si las necesitan
- Recuperar la memoria cuando los procesos terminan
- Manejar el intercambio entre disco y memoria (cuando no entran todos los procesos en memoria)

Monoprogramacion sin abstracción de memoria

Solo se ejecuta un programa a la vez:

- Sin abstracción de memoria
- Comparte la memoria del SO y el programa
- Cuando se escribe un comando:
  - SO carga el programa a la RAM
  - Le da el control al programa
  - Cuando termina le da el control al SO

Básicamente en el caso **a)** en el espacio de memoria yo acomodo el sistema operativo en alguna parte de la RAM (Memoria de acceso aleatorio) y el resto queda para poner un programa, una vez que yo tipeo el programa que quiero ejecutar lo que hace es que el SO le da el control al programa, este se va a ejecutar y una vez que termine le vuelve a dar el control al SO y se queda esperando para ejecutar un nuevo programa.

Hay otros esquemas como el caso **b)** que el sistema operativo se encuentra en ROM (Memoria de sólo lectura), eso es bueno porque mi programa no puede acceder a ninguna posición del SO, ni reescribirla.

En el caso **c)** los controladores de dispositivos pueden estar en la parte superior de la memoria en una ROM y el resto del sistema en RAM más abajo. En este modelo la porción del sistema en la ROM se conoce como BIOS (Sistema básico de entrada y salida).

En los casos a) y c) tienen la desventaja de que un error en el programa de usuario puede borrar el sistema operativo.

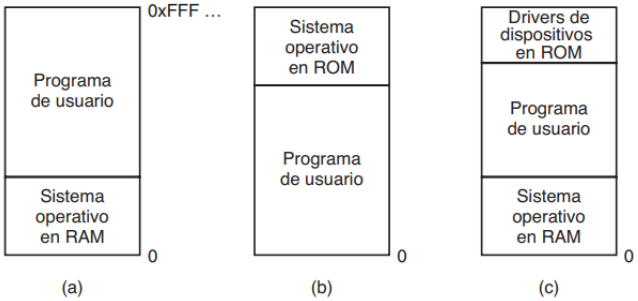
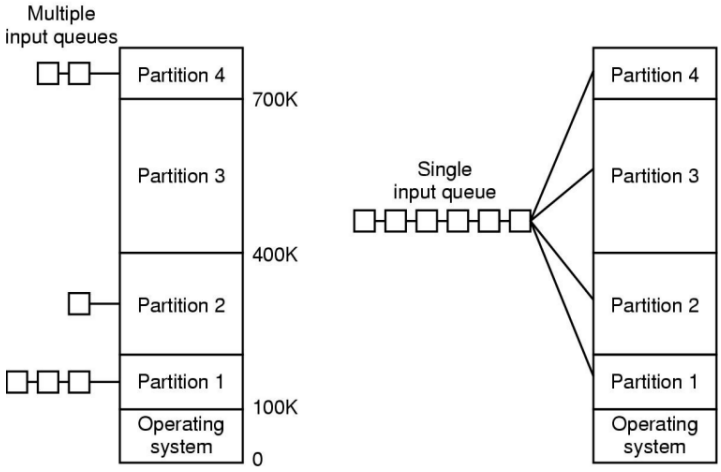


Figura 3-1. Tres formas simples de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

Cada programa veía únicamente la memoria física por lo tanto no era posible tener simultáneamente dos programas debido a que uno podía escribir en el otro. El sistema operativo debe guardar todo el contenido de la memoria en un archivo en el disco, para que después se traiga a memoria y se ejecute el siguiente programa. Si sólo hay un programa a la vez en la memoria, no hay conflictos. Este concepto es el swapping.

Multiprogramación con Particiones Fijas

Para tener multiprogramación (2 programas en memoria), la primera aproximación fue dividir la memoria en N particiones de distinto tamaño y en cada partición pongo un programa.



Estos no varían en el tiempo, ósea su asignación es estática. Hay 2 alternativas de como ejecutar los programas a medida que van llegando:

Múltiples colas: Que cada partición tenga su planificador FIFO, los programas se quedan esperando a que se libere esa partición y a continuación se ejecuta el siguiente. Metiéndose en la cola de acuerdo al tamaño de cada partición. Desventaja: Desperdicio dentro de las particiones y por lo que puede pasar que haya particiones grandes sin uso y varios procesos chicos encolados.

Colas simples: Tengo una sola cola donde van llegando los programas y se van derivando a las particiones que se encuentran libres. El concepto de esta lista, es de tener todas las particiones ocupadas.

Problemas de la multiprogramación con particiones fijas:

- ✓ Reubicación de memoria
- ✓ Protección entre programas

Multiprogramación sin abstracción: pero añadiendo cierto hardware es posible la multiprogramación sin swapping, pero tendríamos dos problemas:

- ✓ ***Protección:*** esto se logra particionando la memoria en 2Kb y a cada parte se le asigna una llave de 4bits, por lo tanto, cualquier intento por acceder a una parte de memoria con distinta llave se genera un trap evitando que los usuarios modifiquen otro programa.
- ✓ ***Reubicación de memoria:*** si tenemos dos programas en memoria principal los dos programas hacen referencia a la memoria física absoluta. Lo que deseamos es que cada programa haga referencia a un conjunto privado de direcciones locales para él, si no una instrucción podría especificar un salto al otro programa tenemos una solución que es la reubicación estática que modificaba el programa al instante a medida que se cargaba en memoria, no es muy conveniente debido tengo que saber cuáles palabras so reubicables y cuáles no pero no se utiliza porque es lento y complicado. Por lo tanto, recurrimos a la abstracción.

Multiprogramación con abstracción: la reubicación de memoria se resuelve con el espacio de direcciones que es el conjunto de direcciones que puede utilizar un proceso para direccionar la memoria. Cada proceso tiene su propio espacio de direcciones, independiente de los que pertenecen a otros procesos. Podemos asociar el espacio de direcciones de cada proceso sobre una parte distinta en memoria gracias al ***registro base y limite***, es decir proporcionamos a cada proceso su espacio de direcciones privado.

Multiprogramación con Particiones Variables

Se logro una optimización en la memoria gracias a una asignación dinámica.

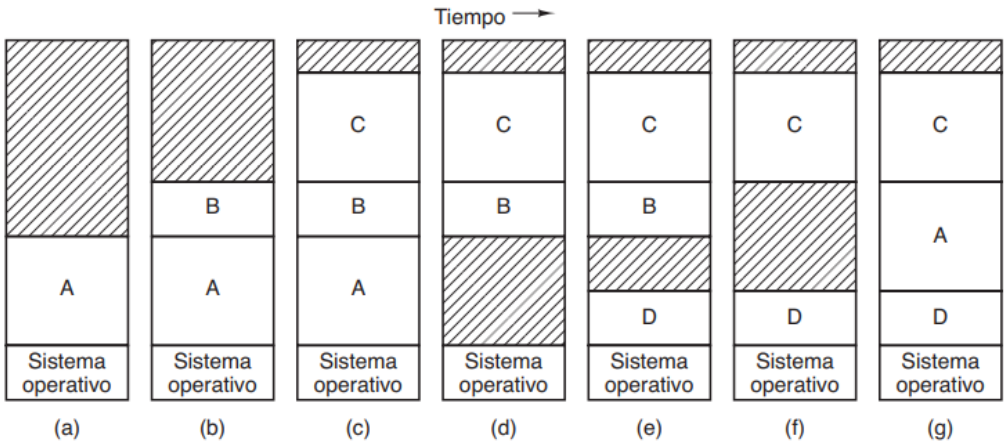


Figura 3-4. La asignación de la memoria cambia a medida que llegan procesos a la memoria y salen de ésta. Las regiones sombreadas son la memoria sin usar.

La memoria principal RAM es a menudo más chica que la que necesitan todos los procesos, entonces lo que se hace es llevar cada proceso completo a memoria y lo ejecuta durante cierto tiempo y después lo regresa al disco. Para esto debemos tener una reubicación dinámica de los

procesos que están en la memoria que se hace posible gracias a los registros base y limite. Pero ahora tenemos dos problemas:

- ✓ **Fragmentación externa:** es todo espacio de memoria vacío (hueco) que no se le puede reasignar a ningún proceso debido a que no entra. Esta se puede evitar con compactación, pero lleva tiempo y es lento.
- ✓ **Fragmentación interna:** Si hay un hueco adyacente al proceso, puede asignarse y se permite al proceso crecer en el hueco; es aquel espacio que se le asigna a un proceso y esta demás.

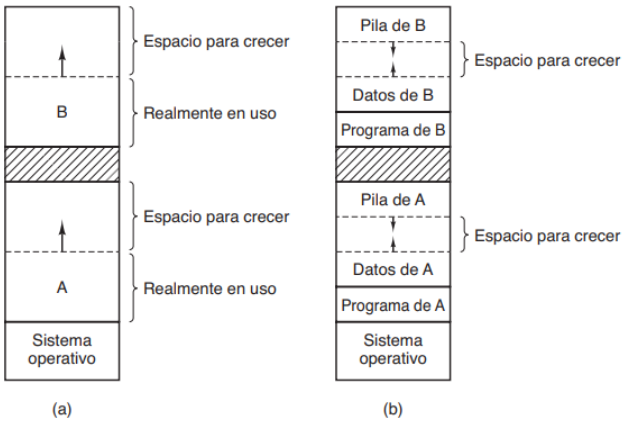


Figura 3-5. (a) Asignación de espacio para un segmento de datos en crecimiento. (b) Asignación de espacio para una pila en crecimiento y un segmento de datos en crecimiento.

El kernel debe tener registro de la distribución de la memoria, para poder asignar memoria, esto se actualiza en cada movimiento.

Administración Dinámica de la Memoria

Cuando la memoria se asigna en forma dinámica, el sistema operativo debe administrarla. En términos generales, hay dos formas de llevar el registro del uso de la memoria:

Mapa de bits: Con un mapa de bits, la memoria se divide en unidades de asignación tan pequeñas como unas cuantas palabras y tan grandes como varios kilobytes. Para cada unidad de asignación hay un bit correspondiente en el mapa de bits, que es 0 si la unidad está libre y 1 si está ocupada. Entre más pequeña sea la unidad de asignación, mayor será el mapa de bits.

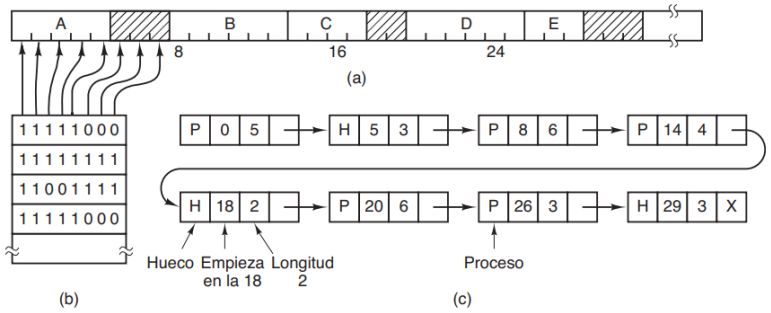


Figura 3-6. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas de graduación muestran las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libres. (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

¿Cada Bits que tamaño de memoria representa? Si hago que represente muy pocos bits, mi mapa de bits va a ser muy grande y si hago que cada bit represente a varios K, la granularidad va a ser mala, el mapa de bits será más pequeño. Lo conveniente es que cada bit sea de 1K.

Si la unidad de asignación se elige de manera que sea grande, el mapa de bits será más pequeño, pero se puede desperdiciar una cantidad considerable de memoria en la última unidad del proceso si su tamaño no es un múltiplo exacto de la unidad de asignación.

Desventaja: dificultad en buscar espacio debido a que tengo que ir bit por bit y concatenar con los demás 0.

Ventaja: más fácil actualizar espacios vacíos.

Lista Enlazada: Cada entrada en la lista especifica un hueco (H) o un proceso (P), la dirección en la que inicia, la longitud y un apuntador a la siguiente entrada.

Desventaja: más lento el actualizar los espacios vacíos debido a que tengo que sacar un elemento y sumarlo con otro para que me del total del espacio vacío que se me acaba de generar.

Ventaja: facilidad en buscar espacios vacíos ya que se fija si hay un hueco y si no salta y sigue buscando espacio.

Se puede usar diferentes algoritmos:

- **First Fit:** el primer hueco en el que entra es el que usa.
- **Next Fit:** conveniente para usar parte final de la memoria, es como que deja una referencia en el último hueco que se usó para usar el que sigue.
- **Best Fit**: usa el hueco que mejor le quede analizando primero todos los huecos (genera mucha fragmentación externa).
- **Worst Fit:** los fragmentos externos que me queden sean muy grandes para que me entre un nuevo programa.
- **Quick Fit:** tengo varias listas con distintos tamaños de huecos. Dependiendo del espacio que necesite mi proceso elijo de donde sacar un hueco y asignarlo a mi proceso.

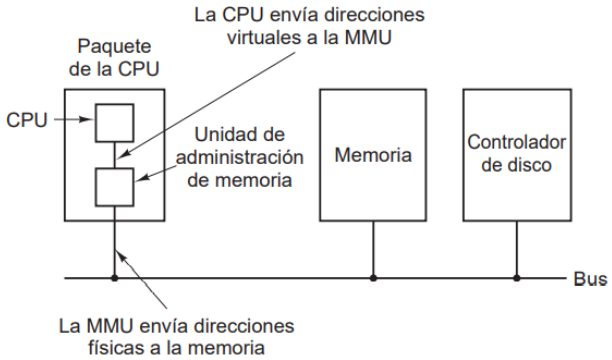
Memoria Virtual

Básicamente es, que lo que ve el procesador no es lo mismo que la memoria física. La idea básica detrás de la memoria virtual es que cada programa tiene su propio espacio de direcciones, el cual se divide en trozos llamados páginas. Cada página es un rango contiguo de direcciones. Estas páginas se asocian a la memoria física, pero no todas tienen que estar en la memoria física para poder ejecutar el programa. Cuando el programa hace referencia a una parte de su espacio de direcciones que está en la memoria física, el hardware realiza la asociación necesaria al instante. Cuando el programa hace referencia a una parte de su espacio de direcciones que no está en la memoria física, el sistema operativo recibe una alerta para buscar la parte faltante y volver a ejecutar la instrucción que falló.

Las implementaciones que tiene son: Paginación, Segmentación y Segmentación Paginada.

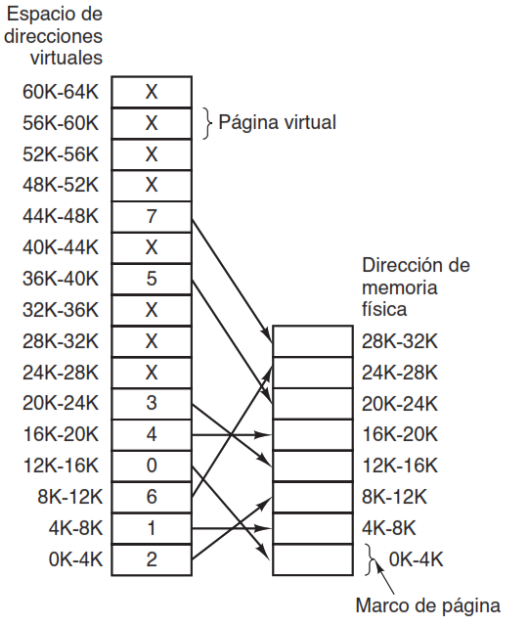
Paginación

En paginación los programas hacen referencia a un conjunto de direcciones de memoria.



Estas direcciones generadas por el programa se conocen como ***direcciones virtuales*** y forman el espacio de ***direcciones virtuales***. Cuando se utiliza memoria virtual, las direcciones virtuales no van directamente al bus de memoria. En vez de ello, van a una MMU (Unidad de administración de memoria) que asocia las direcciones virtuales a las direcciones de memoria físicas.

En la paginación el espacio de direcciones que tiene cada uno de los procesos los separo en ***páginas*** y después la memoria física (RAM) la separo en ***marco de páginas***. Entonces la MMU lo que hace es: asignar paginas de memorias a marcos de páginas.



En este ejemplo, tenemos 16 páginas virtuales y 8 marcos de página. Por sí sola, la habilidad de asociar 16 páginas virtuales a cualquiera de los ocho marcos de página mediante la configuración de la apropiada asociación de la MMU no resuelve el problema de que el espacio de direcciones virtuales sea más grande que la memoria física. Como sólo tenemos ocho marcos de página físicos, sólo ocho de las páginas virtuales se asocian a la memoria física. Las demás, que se muestran con una cruz en la figura, no están asociadas. En el hardware real, un ***bit de presente/ausente*** lleva el registro de cuáles páginas están físicamente presentes en la memoria.

¿Qué ocurre si el programa hace referencia a direcciones no asociadas?

La MMU detecta que la página no está asociada (lo cual se indica mediante una cruz en la figura) y hace que la CPU haga un trap al sistema operativo. A este trap se le llama ***fallo de página.*** El sistema operativo selecciona un marco de página que se utilice poco y escribe su contenido de vuelta al disco (si no es que ya está ahí). Después obtiene la página que se acaba de referenciar en el marco de página que se acaba de liberar, cambia la asociación y reinicia la instrucción que originó el trap.

El número de página se utiliza como índice en la ***tabla de páginas***, conduciendo al número del marco de página que corresponde a esa página virtual. Si el bit de presente/ausente es 0, se provoca un trap al sistema operativo. Si el bit es 1, el número del marco de página encontrado en la tabla de páginas se copia a los 3 bits de mayor orden del registro de salida, junto con el desplazamiento de 12 bits, que se copia sin modificación de la dirección virtual entrante. En conjunto forman una dirección física de 15 bits. Después, el registro de salida se coloca en el bus de memoria como la dirección de memoria física.

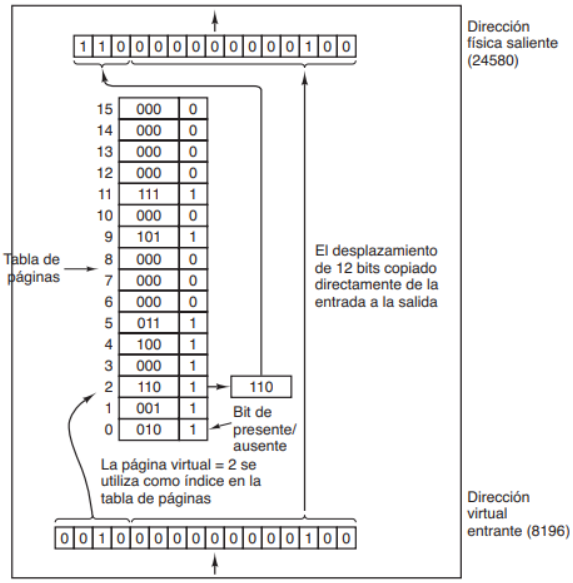


Figura 3-10. La operación interna de la MMU con 16 páginas de 4 KB.

Tablas de páginas

El propósito de la tabla de páginas es asociar páginas virtuales a los marcos de página. Hablando en sentido matemático, la tabla de páginas es una función donde el número de página virtual es un argumento y el número de marco físico es un resultado. Entrada de una tabla de página tiene los siguientes campos bits:

Estructura de una entrada en la tabla de páginas

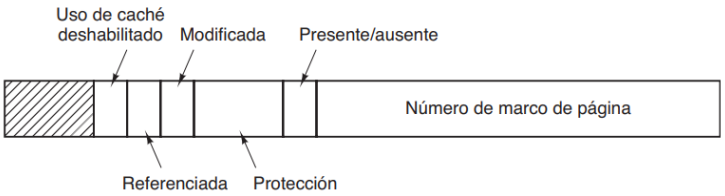


Figura 3-11. Una típica entrada en la tabla de páginas.

- ***Numero de marco de página:*** Es el campo mas importante. El objetivo de la asociación de páginas es mostrar este valor
- ***Bit presente/ausente:*** Si este bit es 1, la entrada es válida y se puede utilizar. Si es 0, la página virtual a la que pertenece la entrada no se encuentra actualmente en la memoria.
- ***Bits de protección:*** Indican qué tipo de acceso está permitido. En su forma más simple, este campo contiene 1 bit, con 0 para lectura/escritura y 1 para sólo lectura.
- ***Bits de modificada:*** Lleva el registro del uso de páginas. Cuando se escribe en una página, el hardware establece de manera automática el bit de modificada.
- ***Bits de referenciada:*** Lleva el registro del uso de páginas. Se establece cada vez que una página es referenciada, ya sea para leer o escribir. Su función es ayudar al sistema operativo a elegir una página para desalojarla cuando ocurre un fallo de página.

El problema con paginación es que tengo un espacio de direcciones unidireccional entonces si yo quisiera modificar algún segmento en memoria tengo que modificar todos mis espacios en memoria.

La solución para asociar direcciones virtuales a direcciones físicas sin pasar por la tabla de páginas usamos un TLB (Buffer de traslación).

Si hay fallo de paginas y todos los marcos están llenos usamos algún algoritmo de sustitución de páginas.

Algoritmo de Sustitución de Paginas

La asociación de una dirección virtual a una dirección física debe ser rápida por lo tanto se implementan los siguientes algoritmos:

Algoritmo no usado recientemente:

Los bits R (escritura/lectura *referencia* a la página) y M (cuando se *modifica* la página) estos bits se deben actualizar en cada referencia a la memoria, por lo que es imprescindible que se establezcan mediante el hardware.

Cuando se inicia un proceso, ambos bits, para todas sus páginas se establecen en 0 mediante el sistema operativo. El bit R se borra en forma periódica (en cada interrupción de reloj) para diferenciar las páginas a las que no se ha hecho referencia recientemente de las que si se han referenciado. Cuando ocurre un fallo de página, el sistema operativo inspecciona todas las páginas y las divide en 4 categorías con base en los valores actuales de sus bits R y M:

- ✓ Clase 0: no ha sido referenciada, no ha sido modificada.
- ✓ Clase 1: no ha sido referenciada, ha sido modificada.
- ✓ Clase 2: ha sido referenciada, no ha sido modificada.
- ✓ Clase 3: ha sido referenciada, ha sido modificada.

Se selecciona la menor clase cuando se produce el fallo de página.

Primera en entrar, primera en salir (FIFO):

Este método no se fija en esos bits, si no básicamente lo que hace es: pone una lista enlazada y la primera que se cargo en un marco, es la primera que voy a desalojar. No sabe si la pagina desalojada se usó hace poco o se va a usar hace poco. Este algoritmo es malo.

Algoritmo de segunda oportunidad:

Este algoritmo hace una modificación simple al algoritmo FIFO que evita el problema de descartar una página de uso frecuente es inspeccionar el bit R de la página más antigua. Si es 0, la página es antigua y no se ha utilizado, por lo que se sustituye de inmediato. Si el bit R es 1, el bit se borra, la página se pone al final de la lista de páginas y su tiempo de carga se actualiza, como si acabara de llegar a la memoria. La desventaja es que si mi programa es muy grande no voy a tener las suficientes paginas cargadas en memoria a tiempo y va a ser lento.

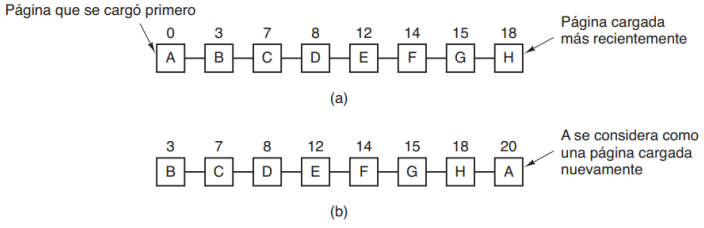


Figura 3-15. Operación del algoritmo de la segunda oportunidad. (a) Páginas ordenadas con base en FIFO. (b) Lista de las páginas si ocurre un fallo de página en el tiempo 20 y *A* tiene su bit *R* activado. Los números encima de las páginas son sus tiempos de carga.

El algoritmo del reloj:

Es una lista circular (Lista doblemente enlazada) igual que en el caso anterior se fija el bit R si es cero se remplaza la nueva página que se quiere usar y si no se pone el bit R = 0 y el puntero sigue a la siguiente página. Esto evita tener que andar copiando en la cola los elementos.

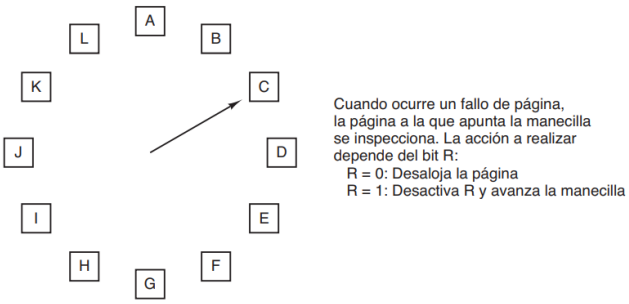


Figura 3-16. El algoritmo de reemplazo de páginas en reloj.

El algoritmo menos usadas recientemente (LRU):

Se crea una matriz NxN donde N es el número de páginas. Lo que hace es poner en 1 toda la fila y luego 0 toda la columna del marco usado. El objetivo es saber en todo momento cual es la página que hace más tiempo que no se usa. Esto se determina viendo la página que se usó tachando la fila y la columna de esa página y viendo en que fila tengo más cero es la página que menos se ha usado.

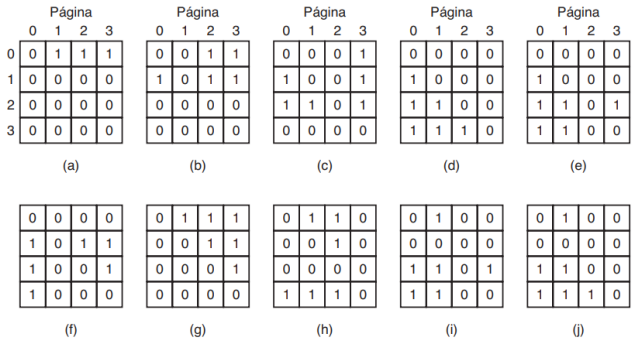


Figura 3-17. LRU usando una matriz cuando se hace referencia a las páginas en el orden 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

Conjunto de trabajo: Bastante difícil, no lo vi.

El algoritmo WSClock: Determina qué conjunto de trabajo (El conjunto de páginas que utiliza un proceso en un momento dado) debería sacar misma analogía del reloj con grupo de páginas. El objetivo de este algoritmo es que no ocurra fallos de página.

RESUMEN DE LOS ALGORITMOS DE REEMPLAZO DE PÁGINAS

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

SEGMENTACION

Dado que el problema de página, tengo un solo espacio de direcciones por proceso, varían en el tiempo, entonces lo que se necesita realmente es una forma de liberar al programador de tener que administrar las tablas en expansión y contracción. Una solución simple es proporcionar la máquina con muchos espacios de direcciones por completo independientes, llamados ***segmentos***. Debido a que cada segmento constituye un espacio de direcciones separado, los distintos segmentos pueden crecer o reducirse de manera independiente, sin afectar unos a otros.

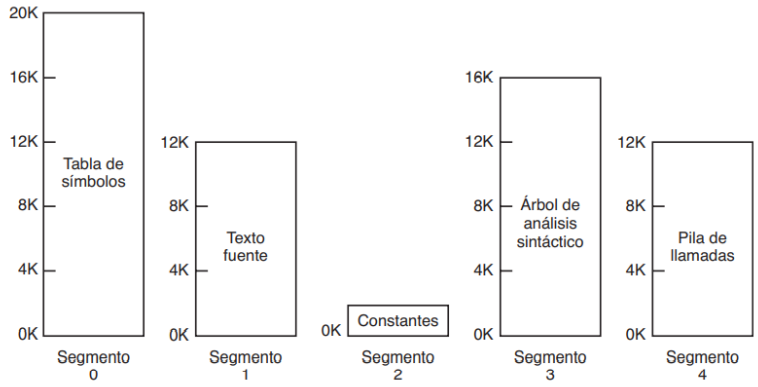


Figura 3-32. Una memoria segmentada permite que cada tabla crezca o se reduzca de manera independiente a las otras tablas.

Para especificar una dirección en esta memoria segmentada o ***bidimensional***, el programa debe suministrar una ***dirección en dos partes***, un número de segmento y una dirección dentro del segmento. Esto soluciona el problema de que si tengo que sacar algún segmento y después ponerlo basta solo cambiar el segmento independiente y no tendría que modificar todo el programa.

Consideración	Paginación	Segmentación
¿Necesita el programador estar consciente de que se está utilizando esta técnica?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1	Muchos
¿Puede el espacio de direcciones total exceder al tamaño de la memoria física?	Sí	Sí
¿Pueden los procedimientos y los datos diferenciarse y protegerse por separado?	No	Sí
¿Pueden las tablas cuyo tamaño fluctúa acomodarse con facilidad?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Por qué se inventó esta técnica?	Para obtener un gran espacio de direcciones lineal sin tener que comprar más memoria física	Para permitir a los programas y datos dividirse en espacios de direcciones lógicamente independientes, ayudando a la compartición y la protección

Segmentación Paginada

Como en el caso de segmentación los segmentos no son todos iguales por lo tanto quedan pedacitos desocupados que no se pueden ocupar lo cual genera fragmentación externa. Solución tanto la fragmentación interna como la externa es que particiono mis segmentos en partes de 4K, entonces voy a cargar en memoria principal RAM una porción del segmento de 4K cuando lo necesite y debido a esto van a ser todas mis partes iguales evitando

fragmentación externa. Si pongo páginas de segmento en la memoria me estoy evitando fragmentación interna debido a que me evito asignarle más espacio a los segmentos por las dudas que crezcan.