TABLA DE CONTENIDOS

TABLA DE CONTENIDOS **2**

ADMINISTRACIÓN DE MEMORIA **3**

ADMINISTRACIÓN DE MEMORIA SIN INTERCAMBIO **8**

MONOPROGRAMACIÓN SIN INTERCAMBIO O MÁQUINA DESNUDA 8
MONITOR RESIDENTE 9
Hardware de Protección *9*
REUBICACIÓN 10

SWAPPING **13**

MULTIPROGRAMACIÓN **13**

PARTICIONES MULTIPLES 14
Multiprogramación con Particiones Fijas (MFT) *14*
Multiprogramación con Particiones Variables (MVT) *15*

ADMINISTRACIÓN DEL ESPACIO LIBRE **18**

ASIGNACIÓN DEL PRIMER AJUSTE 18
ASIGNACIÓN DEL MEJOR AJUSTE 19
ASIGNACIÓN DEL SIGUIENTE AJUSTE 19

ADMINISTRACIÓN DE ASIGNACIÓN DE MEMORIA **20**

ADMINISTRACIÓN CON MAPA DE BITS 20
ADMINISTRACIÓN DE MEMORIA CON LISTAS ENLAZADAS O LIGADAS 21
SISTEMAS DE LOS ASOCIADOS *22*

MEMORIA VIRTUAL **24**

PAGINACIÓN 25
Tablas de páginas *26*
Formato de entrada en la tabla de páginas *29*
Memoria Asociativa *29*
Algoritmos de reemplazo de páginas *30*
Aspectos de diseño *33*
SEGMENTACIÓN 34
Segmentación pura *36*
Segmentación con Paginación *36*

ANEXO I: **38**

RESUMEN 38

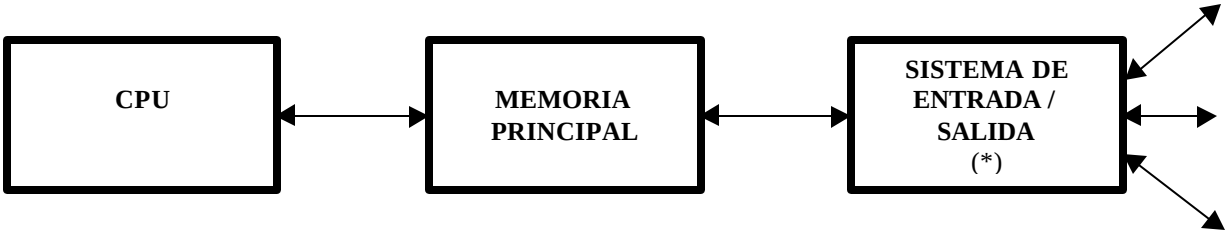
ANEXO II: **39**

RESUMEN 39

BIBLIOGRAFIA **40**

ADMINISTRACIÓN DE MEMORIA

Como mencionáramos en los módulos anteriores, este nuevo contacto con Ustedes, tiene por finalidad analizar los distintos enfoques y técnicas que un Sistema Operativo Moderno emplea para administrar un recurso tan valioso como es la Memoria Principal, la cual es, costosa y limitada.



(*) Temas a desarrollar en los módulos siguientes

Como se ve en la figura, tanto la CPU como el sistema de E/S interactuan con la memoria principal. Esto es un gran arreglo[1] de palabras[2], cada una con una dirección propia. La interacción se realiza a través de una secuencia de lecturas o escrituras en direcciones de memoria específicas. La CPU busca datos de la memoria principal o los almacena allí.

La memoria es un recurso de tipo apropiable (se asigna a los procesos), que debe ser administrado para obtener su mejor rendimiento. Nuestro enfoque ejercerá preferencia en la administración de memoria en sistemas multiprogramados, es decir, sistemas en donde coexisten varios procesos en memoria.

En general, podemos distinguir dos esquemas de administración de memoria para sistemas multiprogramados. El primero exige que los procesos deban estar cargados completamente en memoria para poder ejecutarse y el segundo esquema no requiere que el proceso completo esté cargado en memoria (se van cargando los trozos del proceso que son requeridos para ejecutarse).

Brevemente, podemos decir, que la memoria es un recurso utilizado para el almacenamiento de las instrucciones que forman un proceso. El ciclo típico de ejecución de instrucciones incluye la transferencia de instrucciones de memoria a CPU, decodificación de instrucciones, búsqueda de operandos y ejecución efectiva de las instrucciones. Posteriormente, los resultados también pueden ser almacenados en memoria.

Cuando se desea ejecutar un proceso, éste debe ser cargado en memoria previamente. Cuando un proceso entra en ejecución se utiliza un proceso del sistema que

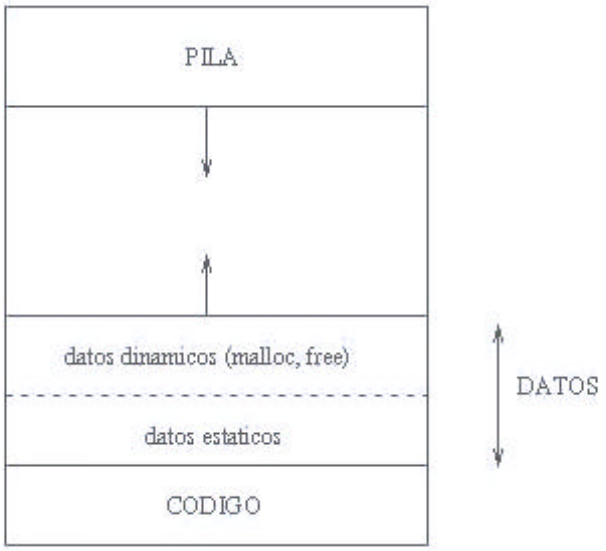
[1] Arreglo: Son estructuras que posibilitan la identificación de una posición determinada brindando sus coordenadas

[2] Palabra: es la unidad de asignación que se utiliza para asignar memoria a los procesos. Por lo general su tamaño es de dos bytes.

se denomina cargador, que lo transfiere a memoria. En Unix por ejemplo, el cargador almacena el proceso en memoria distinguiendo en él tres segmentos distintos:

- Segmento de código
- Segmento de Datos
- Pila

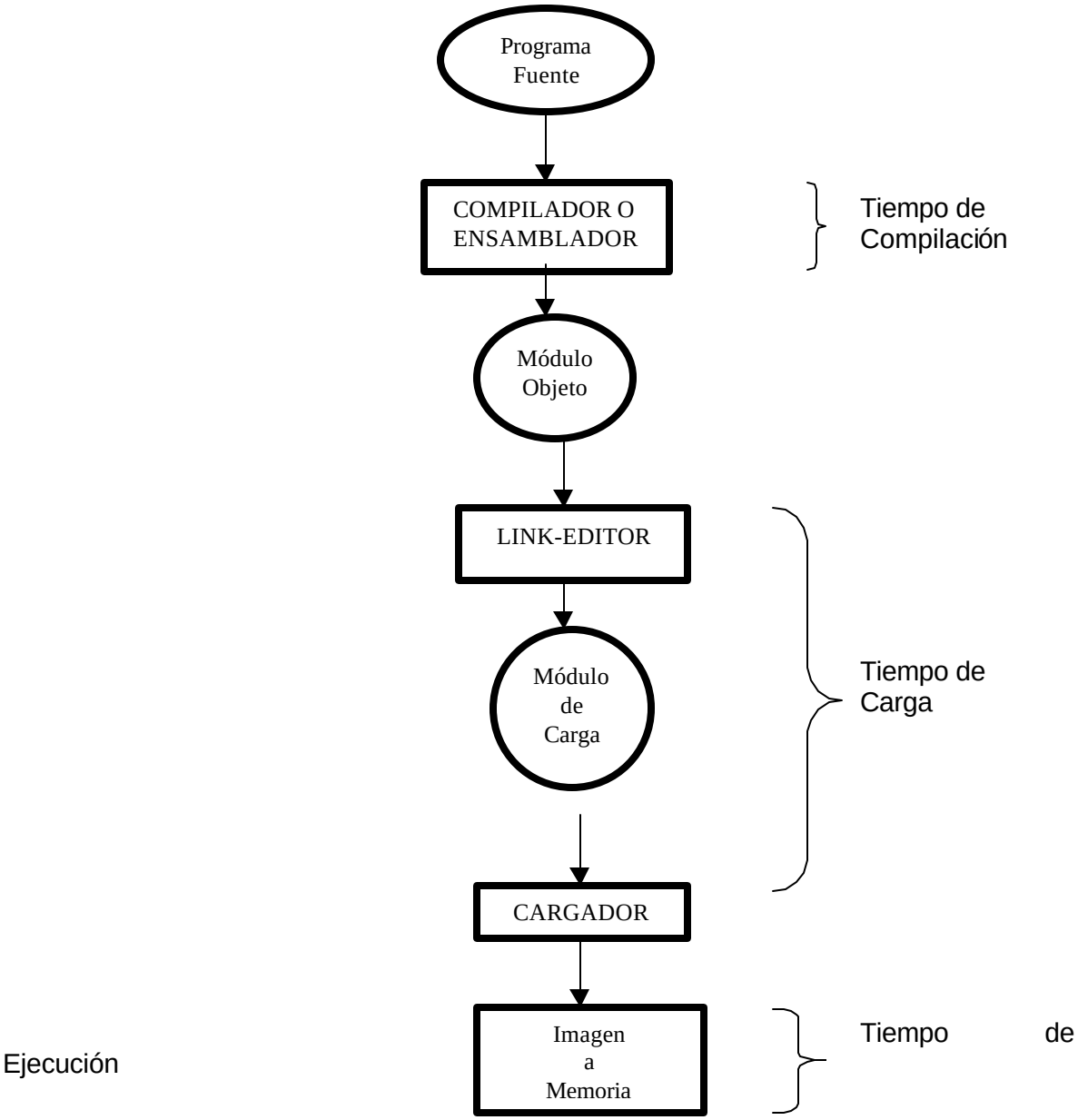
Dichos segmentos se distribuyen según muestra la siguiente figura:



La asignación de memoria en los segmentos es responsabilidad de diferentes etapas del sistema:

- Compilación
- Enlazado (linker)
- Carga (loader)
- Asignación de memoria en tiempo de ejecución (run time)

El siguiente diagrama de flujo intenta expresar este proceso:



Generalmente, un programa de usuario pasa por varios pasos antes de ser ejecutado (ver diagrama de flujo anterior). Las direcciones se representan de diferentes formas en cada paso. Las direcciones en un programa fuente son generalmente simbólicas (por ejemplo X). Un compilador, típicamente, hará una "asociación" de estas direcciones simbólicas convirtiéndolas en direcciones reubicables (tal como 14 bytes a partir del comienzo de este módulo).

El link-editor o cargador convertirán estas direcciones reubicables en direcciones absolutas (tal como 74014).

Un programa eventualmente podrá ser mapeado a direcciones absolutas y cargado a memoria para su ejecución.

Por lo tanto, un ciclo de ejecución de una instrucción, típicamente primero buscara una instrucción de memoria. Esta será decodificada y por ejemplo causara la búsqueda de operandos en la memoria. Luego de procesar la instrucción usando estos operandos, el resultado puede ser almacenado nuevamente en memoria. Notemos que la memoria solamente ve un flujo de direcciones de memoria: no se entera como fueron generadas (no se preocupa por el contador de programa, indexaciones, etc.) o que son (datos o instrucción). Por lo tanto, podemos ignorar como un programa genera una dirección de memoria. Se podrá usar cualquier técnica para su generación: nosotros solo estamos interesados en la secuencia de direcciones de memoria generadas por el programa en ejecución.

Con la intención de ser genéricos en el enfoque, cabe mencionar que un programa fuente puede estar formado por uno o un conjunto de archivos en donde se definen indistintamente variables, funciones y/o procedimientos. Por ejemplo, en el lenguaje C, se usa el encabezado externo para indicar que una función definida en un archivo puede ser utilizada por otro archivo después de su declaración correspondiente.

En el caso que un programa esté definido mediante un conjunto de archivos fuentes, el compilador genera el código objeto de cada uno de los archivos correspondientes. Luego, es el enlazador el encargado de generar un solo archivo ejecutable a partir de todos los módulos objetos. Finalmente, el cargador es el encargado de levantar a memoria el archivo ejecutable.

En el afán de reforzar y formalizar los conceptos más importantes, les presentamos la siguiente síntesis:

Compilación

Durante el proceso de compilación se generan los archivos objetos de cada uno de los archivos fuentes del programa.

Enlazado

En esta etapa se reagrupan todos los archivos objetos pertenecientes al programa y se genera un solo archivo ejecutable. Las referencias externas son resueltas, así como las llamadas a funciones de biblioteca del sistema.

Carga

Durante este proceso se carga en memoria el programa ejecutable. Este proceso se realiza cuando se ejecuta el programa.

Asignación de memoria en tiempo de ejecución

Alternativamente también es posible que los procesos adquieran y liberen memoria del sistema en forma dinámica utilizando, por ejemplo, funciones de C como malloc o free. Esta labor de asignación y liberación son realizadas conjuntamente entre el compilador y el sistema operativo.

En términos generales, la asignación de memoria para los procesos en ejecución puede realizarse en el momento de compilación y/o enlazado; en el momento de carga o en ejecución. Que la memoria se asigne en tiempo de compilación y/o enlazado, significa que el código resultante de módulos genera código absoluto, es decir, que se utilizarán directamente las direcciones que aquí se referencian. Los programas .COM de MS-DOS se cargan en forma absoluta en memoria.

En los sistemas modernos los compiladores no generan código absoluto sino que generan código relocalizable, es decir que el código donde las direcciones que genera el compilador no son las direcciones de memoria reales que se utilizarán, sino que el Cargador traducirá las direcciones entregadas por el compilador y/o enlazador a las direcciones de memoria que realmente se utilizarán. Este proceso muchas veces es conocido como **mapeo de direcciones.**

ADMINISTRACIÓN DE MEMORIA SIN INTERCAMBIO

Dentro de este tipo de administración se encuentran los sistemas que utilizan monoprogramación, es decir, que mantienen en memoria sólo un proceso de usuario, por lo tanto, no es posible ejecutar concurrentemente o simultáneamente más de un proceso de usuario en el sistema; y también los sistemas multiprogramados, es decir que permiten mantener mas de un proceso de usuario en memoria (tal y como lo habíamos visto en el módulo I).

Dentro de este último esquema existen sistemas que manejan multiprogramación con particiones fijas y sistemas que manejan la multiprogramación con particiones variables.

La característica que poseen en común estos tipos de administración es que ninguno de ellos permite intercambiar procesos entre memoria y disco, es decir, que los procesos no pueden permanecer por períodos de tiempo en disco, sino que durante el tiempo que dure su ejecución, necesariamente, deben permanecer en memoria.

El esquema sin intercambio puede ser práctico para sistemas en donde la cantidad de procesos presente es manejable por la CPU y la memoria. Sin embargo, para aquellos sistemas donde, por lo general, existen muchos usuarios los cuales ejecutan muchos procesos, la capacidad de la CPU y memoria es inferior a la cantidad total de procesos que requieren los recursos. En este último caso, es atractiva la idea de aprovechar mejor los recursos, y que, por ejemplo aquellos procesos que cambian de estado puedan ser almacenados temporalmente en disco, para luego continuar en otro momento su ejecución y así darle la oportunidad a otros procesos de ser ejecutados.

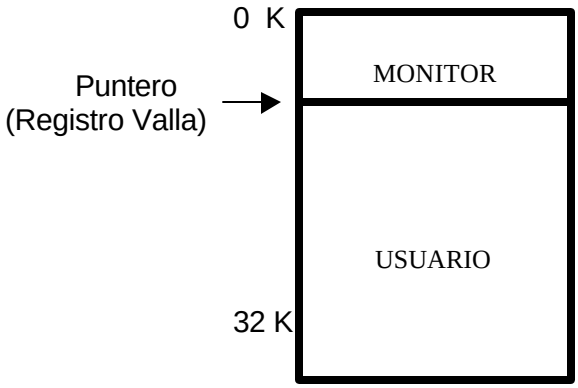
Monoprogramación sin intercambio o máquina desnuda

A este tipo de administración también se la conoce con el nombre de *asignación de sola partición o máquina desnuda*, y corresponde al esquema de administración más sencillo y barato porque es el que le brinda al usuario el control completo de la memoria.

Sin embargo este esquema no se utiliza porque, al no existir sistema operativo, no existe control sobre interrupciones; no hay un monitor residente que procese las llamadas al mismo, ni los errores, ni espacio para control de secuencia de trabajos. Debido a esto, sólo se usa este método en sistemas dedicados, donde el usuario requiere flexibilidad y simplicidad y está dispuesto a programar sus propias rutinas de soporte.

Monitor Residente

Por lo expuesto, el esquema más sencillo que se implementa hasta hoy consiste en que la memoria se divide en dos espacios: uno para que resida el sistema operativo y otro espacio queda disponible al proceso de usuario. Este método también recibe el nombre de **Monitor Residente.**



Este tipo de administración se caracteriza porque, en primer lugar, es monousuario, es decir, soporta un usuario y éste tiene disponible toda la memoria de usuario. En este caso el usuario sólo puede cargar un proceso en memoria. En el caso de tener un sistema compuesto por un espacio de memoria para el sistema operativo y otro espacio para el proceso de usuario, tienen que existir protecciones que eviten que un proceso de usuario perturbe el comportamiento del sistema operativo. Dicha protección, por lo general, es provista por el hardware, a través de un registro denominado **Registro Valla**.

Un ejemplo de este tipo de administración de memoria es el que posee el PC-IBM. A este tipo de administración de la memoria, también se lo conoce con el nombre de MONITOR RESIDENTE (módulo coordinador de asignaciones).

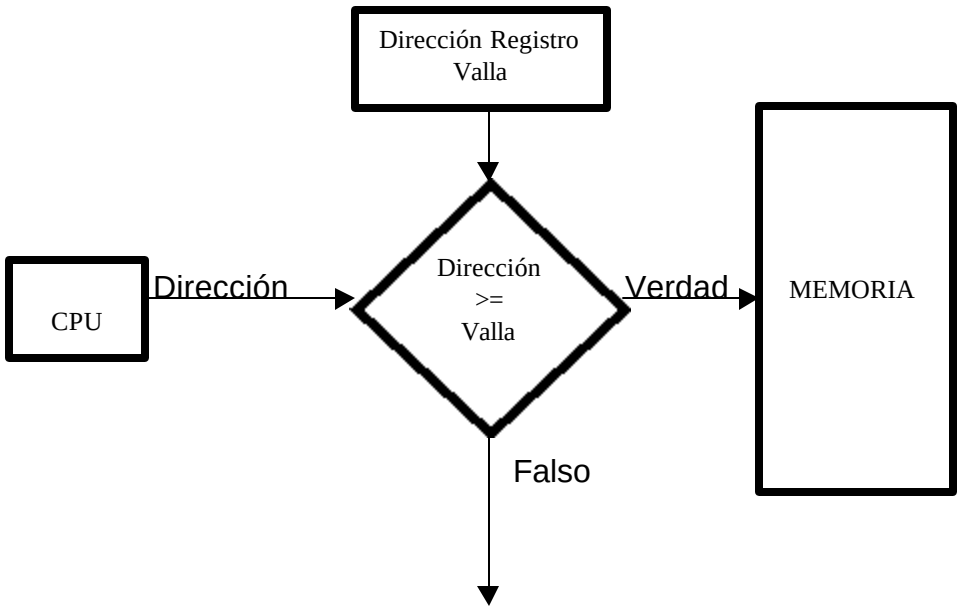
El monitor puede estar ubicado en las direcciones iniciales o finales de memoria. El factor determinante de elección es generalmente la posición del vector de interrupciones[3]. Debido a que generalmente está ubicado en posiciones bajas de memoria (en el gráfico 0 K), es más común encontrar el monitor residente al comienzo de la memoria (como en el gráfico).

Hardware de Protección

Si el monitor reside en posiciones bajas de memoria y el programa de usuario se ejecuta en posiciones altas, necesitamos proteger el código del monitor y sus datos asociados, de cambios causados (accidentalmente o premeditados) por este programa de usuario.

Esta protección debe realizarse por hardware y puede implementarse de diferentes maneras. Una solución general se muestra en la siguiente figura.

[3] Estructura que posee funciones y procedimientos que se ejecutan cuando son invocadas por el SO.



Trap (error de direccionamiento)

Cada dirección (de dato o instrucción) generada por el programa del usuario se compara con una dirección de puntero. Si es una dirección legal es enviada a memoria. Caso contrario, es una dirección ilegal que afecta al monitor residente, dando como resultado un Trap (en este caso, error de direccionamiento), por lo que el sistema operativo toma la acción apropiada , Generalmente termina la ejecución del programa con un adecuado código de error.

Notemos que cada referencia a memoria del programa de usuario será chequeada. Generalmente, este trabajo implica una disminución de velocidad en acceso a memoria.

El sistema operativo tiene acceso irrestricto tanto a la zona de memoria del monitor residente como a la de usuario, lo que le permite cargar programas de usuario en esa zona, sacarlo en caso de error, acceder y modificar parámetros de llamadas al sistemas, etc.

Reubicación

Otro problema a considerar es la carga de los programas de usuarios. Aunque el espacio de direccionamiento de la computadora comienza en 00000, ésta no es la primera dirección del programa, sino la primera luego del puntero valla.

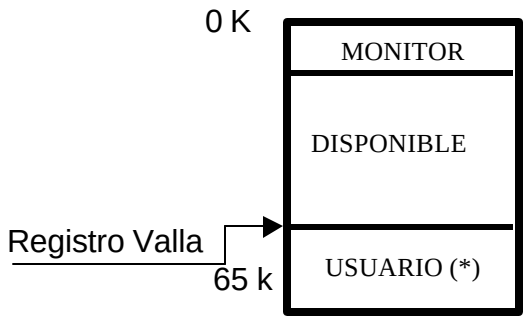
Usualmente, la conversión de direcciones de memoria de instrucciones y datos puede realizarse en tiempo de compilación o en tiempo de carga. Si se conoce la dirección del puntero al momento de la compilación, entonces ya se puede generar código absoluto. Así, este código comienza luego del puntero. Si el puntero cambia, debe recompilarse este código. Como una alternativa, el compilador puede generar código reubicable. En este caso, la generación de direcciones absolutas se posterga hasta el

momento de la carga del programa a memoria. Si cambia el puntero, el código del usuario debe ser recargado solamente.

Sin embargo, en ambos casos el puntero debe permanecer estático durante la ejecución del programa: el puntero puede cambiar sólo cuando no se esten ejecutando programas de usuarios. Existen casos, sin embargo, en que es deseable cambiar el tamaño del monitor residente (y por ende el puntero) durante la ejecución de programas de usuarios. Por ejemplo, el monitor contiene código y espacio para buffers de los drivers de dispositivos. Si un drivers no se usa muy a menudo, no es deseable guardar ese código y datos en memoria, ya que podría usar ese espacio para otros propósitos. Este tipo de código se lo conoce como **código monitor transitorio**: se lo carga y se lo saca de memoria cuando es necesario. Así, el tamaño del monitor varía durante la ejecución de programas.

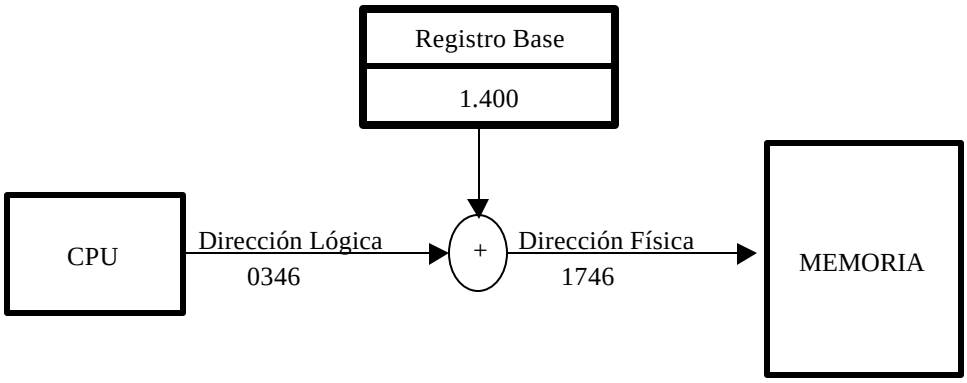
Existen dos formas para permitir que el monitor varíe dinámicamente:

1. Como lo hacían los SO primitivos, cargando los programas de usuario en la zona alta de memoria: aquí la ventaja es que el espacio no usado queda en el medio, pudiendo entonces expandirse el monitor.



(*****) Carga del programa de usuario en zona de memoria alta.

2. Una solución más general es demorar la generación de direcciones absolutas (binding) hasta tiempo de ejecución. Este tipo de reubicacion dinámica requiere un hardware de soporte un poco distinto.

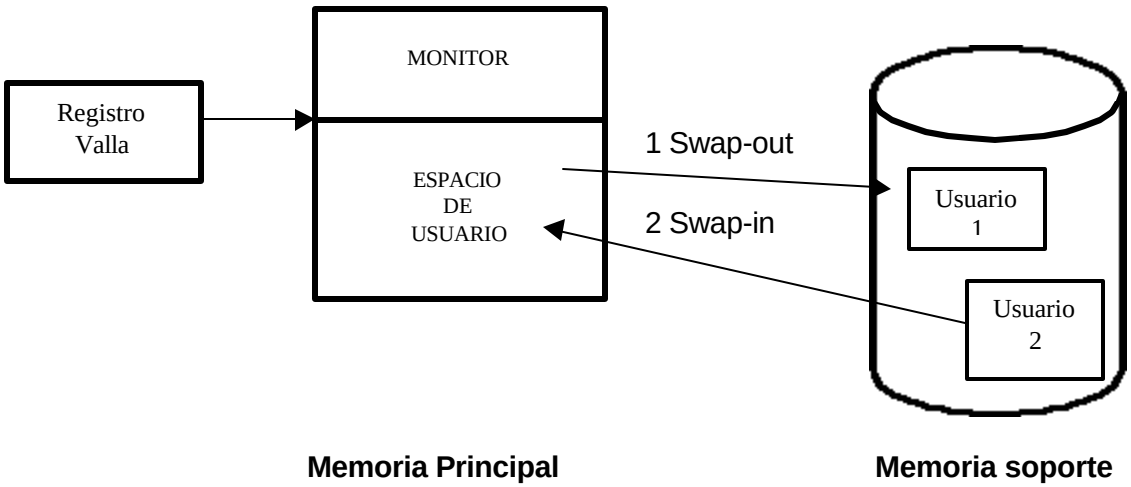


El registro ahora se llama registro base o reubicable. El valor de ese registro se adiciona a cada dirección generada por un proceso usuario al momento de ser enviado a memoria. Observemos que el usuario nunca ve la dirección física real, ya que siempre maneja direcciones lógicas. El hardware de mapeo de memoria es quien convierte las direcciones lógicas en direcciones físicas.

Para el hardware, un cambio del puntero sólo requiere cambiar el registro base y mover todas las direcciones de memoria del usuario a las direcciones correctas relativas al nuevo puntero. Este método puede requerir que una cantidad significativa de memoria sea copiada, pero permite que el puntero cambie en cualquier momento.

SWAPPING

El esquema de administración de memoria por monitor residente aparece como de poco uso ya que es inherentemente para un solo proceso. Sin embargo, fue el método básico usado en sistemas de tiempo compartido primitivos. Estos usaban un monitor residente y la memoria remanente para un solo usuario. Cuando se atendía otro usuario, el contenido de memoria se escribía en un almacenamiento soporte (disco rígido) y se cargaba el código del siguiente usuario.



El swapping requiere una memoria soporte, siendo comúnmente un disco rígido. Debe ser lo suficientemente grande como para almacenar las copias de las imágenes de memoria de todos los usuarios y debe poseer acceso directo a las mismas. La cola de listos consiste en todos los procesos cuyas imágenes de memoria están en la memoria soporte y que están listos para ser ejecutados. Una variable del sistema separada indica qué proceso reside en memoria principal.

Cuando el planificador de CPU decide ejecutar un proceso, llama al despachador, quien chequea si está en memoria, y si no lo está hace un "**swap-out**" al proceso que se encuentra en memoria y un "**swap-in**" del proceso deseado. Luego recarga los registros y transfiere el control al proceso seleccionado.

MULTIPROGRAMACIÓN

La multiprogramación consiste en que el espacio de memoria de usuario es compartido por múltiples procesos, es decir, pueden ser almacenados en memoria un conjunto de procesos simultáneamente. La utilización de multiprogramación trae aparejadas varias ventajas entre las cuales se cuenta el mejor aprovechamiento de la CPU, pues cuando un proceso está realizando E/S (estado BLOQUEADO) , la CPU

podría ser utilizada por otro proceso (estado de EJECUCION); por otro lado, la multiprogramación es atractiva para los sistemas multiusuarios y procesos interactivos.

Particiones Multiples

Bajo este esquema de administración, la memoria es dividida en *n partes*, donde cada una de esas partes pueden ser de tamaños distintos, y cada proceso es asignado a una de estas particiones cuando entra en ejecución.

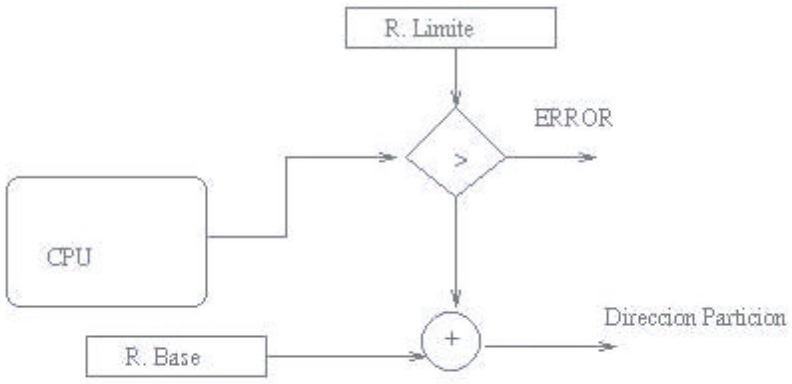
Son posibles dos esquemas principales de administración de memoria. Cada uno divide la memoria en un número de particiones. La mayor diferencia de cada uno de ellos radica en definir las particiones como estáticas o dinámicas.

Multiprogramación con Particiones Fijas (MFT)

Lógicamente, el tamaño de una partición a asignar a un proceso debe ser mayor o igual al tamaño del proceso.

A su vez, existen dos esquemas mediante los cuales se pueden asignar los procesos a una partición. Un esquema consiste en que cada partición tiene una cola de procesos asociados que desean ser cargados en esa partición, o bien, existe una sola cola en donde se va asignando una partición a cada proceso desde el comienzo de la cola. En el primer caso se puede producir sobrecarga en algunas de las particiones, es decir, se puede formar una cola de espera con muchos procesos; mientras que otra partición puede tener una cola de espera vacía. Este problema se soluciona con el segundo esquema mencionado, con lo cual el comportamiento logrado es similar al de una cola para pagar servicios en un banco por ejemplo, en la que si existen tres (3) cajeros, estos atenderán a los clientes en la medida que se vayan desocupando.

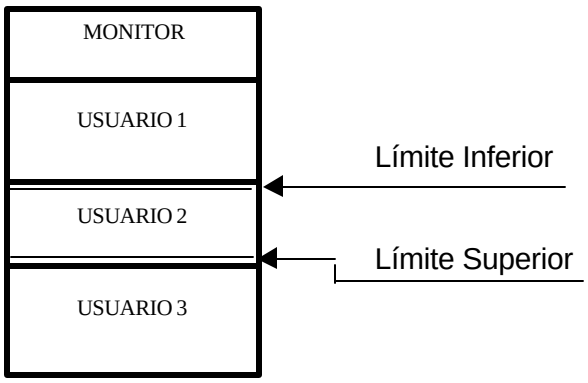
Este esquema de administración contempla la protección de las particiones, que permite proteger cada una de las particiones de las particiones adyacentes o vecinas. Esto significa que cada proceso de usuario debe estar confiado que no se verá perturbado por otro proceso. En los sistemas multiprogramados la protección es realizada por el hardware, mediante dos registros: un registro *base* y un registro *límite*.



Estos dos registros proveen los límites superior e inferior de direcciones que pueda generar legalmente un programa de usuario. Pueden definirse de dos formas:

- **Registro límite:** con los valores de la mayor y la menor dirección física legal.
- **Registro Base y Límite:** el valor de la menor dirección física y el rango de direcciones lógicas. Las direcciones lógicas legales van de 0 al *Límite* y las direcciones físicas reubicables van de *Base* a *Base+Límite*

Con la intención de aclarar aún más esta cuestión proponemos el siguiente gráfico:



Con este tipo de administración se presenta un problema denominado *fragmentación interna*, el cual consiste en que si el proceso que ocupa la partición no la utiliza en un 100% entonces se produce pérdida de memoria, es decir memoria que no puede ser utilizada por otro proceso, porque está asignada a este proceso, pero éste no la utiliza totalmente. Por ejemplo, si tenemos una partición fija de 50 K, y el proceso necesita 40 K, el SO asignará los 50 K de los cuales sólo 40 K serán utilizados por el proceso, generando una fragmentación de 10 K (asignación).

Multiprogramación con Particiones Variables (MVT)

Este tipo de administración de memoria consiste en que la memoria se divide en un conjunto de particiones que se caracterizan porque su tamaño es variable, y se crean dinámicamente en el transcurso de la ejecución de los distintos procesos en el sistema.

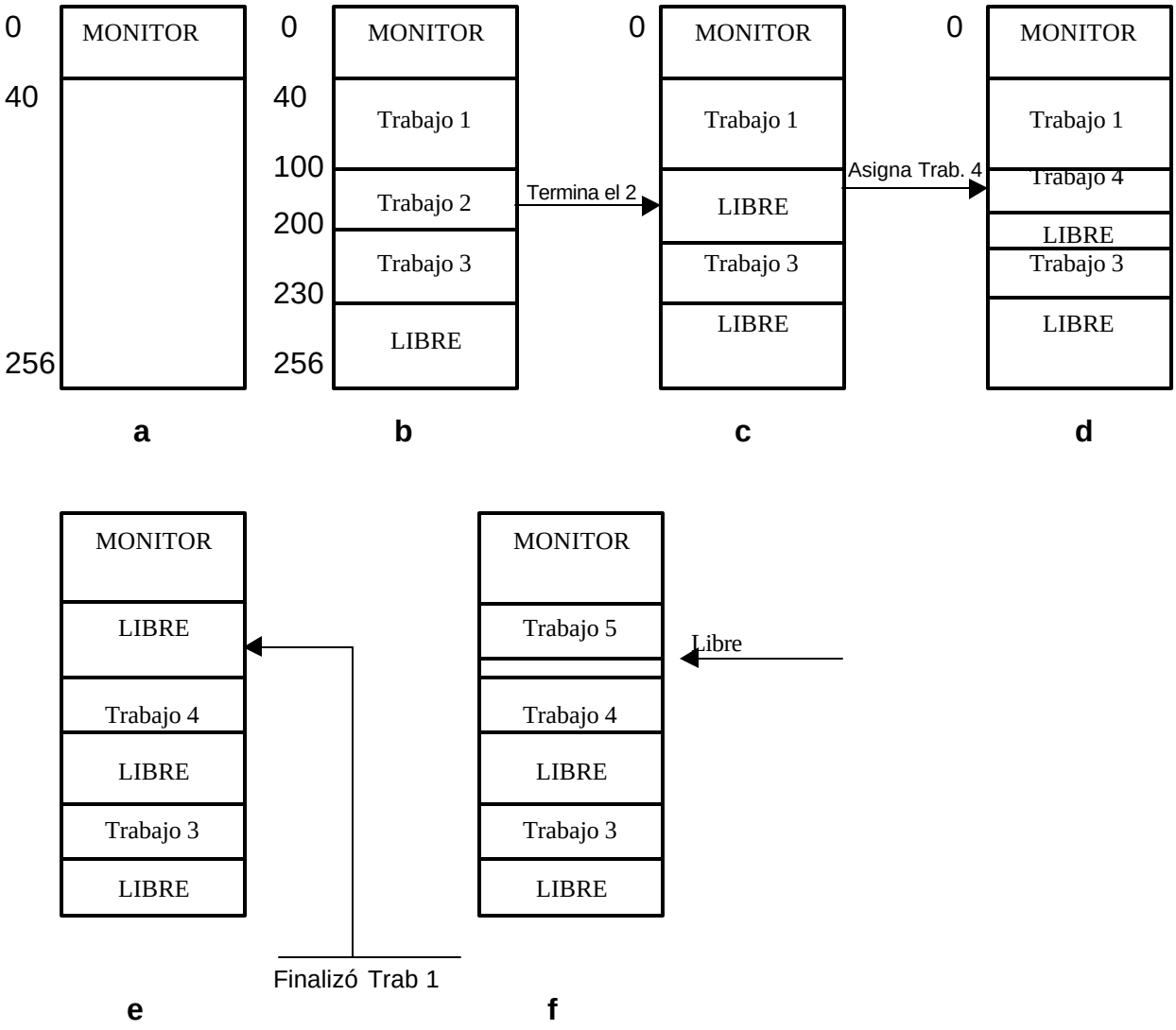
Al igual que la administración con particiones fijas este tipo de administración considera el almacenamiento del proceso completo en memoria para que éste pueda ser ejecutado.

El algoritmo en muy simple: el SO guarda en una tabla indicando las partes de memoria ocupadas. Inicialmente toda la memoria está disponible y es considerada como un gran hueco. Cuando llega un trabajo buscamos por un hueco lo suficientemente grande. Si lo encontramos, asignamos sólo lo necesario para ese trabajo, dejando el resto disponible para otros trabajos futuros.

Ejemplo: Supongamos que tenemos que ejecutar los siguientes trabajos siguiendo una planificación de CPU que utiliza el algoritmo FCFS ya analizado en módulos anteriores:

Trabajo	Memoria	Tiempo
1	60 K	10
2	100 K	5
3	30 K	20
4	70 K	8
5	50 K	15

Supongamos tenemos 256 K de memoria y un monitor residente de 40 K, dejando en consecuencia, 216 K para los usuarios. Por lo tanto la evolución de las asignaciones de espacio es la que se muestra en el siguiente esquema:



Este ejemplo ilustra varios puntos acerca de MVT. En general, hay en cualquier momento hay un conjunto de huecos de varios tamaños en memoria. Cuando llega un trabajo, se busca por un hueco lo suficientemente grande como para asignarlo. Si el hueco es muy grande se lo parte en dos: una parte se asigna al trabajo y la otra queda como otro hueco en la tabla.

Otro aspecto importante de señalar en este tipo de administración es que también existe fragmentación, y ésta ocurre por el continuo ingreso y salida de procesos a memoria. Este fenómeno posibilita que se generen huecos en la memoria que no son suficientes para contener a los procesos que requieren ejecutarse. A este tipo de fragmentación se le conoce con el nombre de **fragmentación externa**.

La pérdida de memoria, producto de la fragmentación externa, se puede recuperar realizando la **compactación**, es decir, reagrupando las porciones de memoria asignadas hacia un lado, y las porciones de memoria libre hacia el otro extremo de la memoria. Sin embargo, esta técnica se debe practicar bajo determinadas circunstancias, pues es un proceso demasiado lento y costoso en términos de recursos que insume al computador, debido a todo el trabajo de mover los usuarios asignados a posiciones altas o bajas de memorias (según donde se aloje el monitor residente).

ADMINISTRACIÓN DEL ESPACIO LIBRE

La asignación de memoria a los distintos procesos que la requieren, se puede concretar, siempre y cuando el sistema tenga conocimiento de las particiones de memoria (fijas, variables o simplemente porciones) que se encuentran ocupadas y libres.

Las particiones que tiene libres deben ser otorgadas a los procesos de la manera que el sistema considere más eficiente. Dentro de los criterios más conocidos de asignación de memoria se encuentran los siguientes:

- Asignación del primer ajuste
- Asignación del mejor ajuste
- Asignación del siguiente ajuste

A continuación proponemos un ejemplo general con la intención de ilustrar las diferencias entre los métodos.

Ejemplo:

En un instante determinado, se dispone de los siguientes huecos o espacios disponibles para ser utilizado por los usuarios 30K, 80K, 15K, 50K, 7K y 77K, en tanto que el tamaño de los procesos que deber ser ejecutados son de 15K, 30K, 75K, 5K y 50K.

Asignación del primer ajuste

Este criterio considera que es mejor asignar al proceso la primera partición de memoria que encuentra libre en el sistema.

Particiones (Huecos Disponibles)	Proceso Asignado
30 K	15 K
80 K	30 K
15 K	5 K
50 K	50 K
7 K	libre
77 K	75 K

Asignación del mejor ajuste

Este criterio considera en otorgar al proceso aquella partición que mejor se ajuste al proceso, es decir, la que mejor se ajuste al hueco. La intención es hacer mínima la fragmentación.

Particiones (Huecos Disponibles)	Proceso Asignado
30 K	30 K
80 K	Libre
15 K	15 K
50 K	50 K
7 K	5 K
77 K	75 K

Asignación del Siguiente ajuste

Este criterio considera que es mejor otorgar aquella partición que sea la siguiente a la última asignación efectuada y que además, tenga el tamaño adecuado como para contener al proceso que solicita memoria.

Particiones (Huecos Disponibles)	Proceso Asignado
30 K	5 K
80 K	50 K
15 K	15 K
50 K	30 K
7 K	Libre
77 K	75 K

ADMINISTRACIÓN DE ASIGNACIÓN DE MEMORIA

El Sistema Operativo puede y debe asignar o desasignar memoria a los procesos. Para ello utiliza esquemas que básicamente le brindan información sobre la cantidad de memoria ocupada y libre o disponible. Algunos de los esquemas utilizados son los siguientes:

Administración con mapa de bits

Este esquema consiste en que la memoria se divide en **bloques** o **unidades de asignación** de cierto tamaño y se mantiene en memoria una tabla o mapa de bits, en la que cada bit se utiliza para identificar un bloque ocupado o un bloque libre. Si un bloque esta ocupado el bit asociado a él está en 1, en tanto que, si el bloque está libre el bit está en 0 (o viceversa).

Supongamos que la unidad de asignación es de 8 bit y que la memoria que se debe administrar utilizando un mapa de bits es de 32 K. Entonces tenemos:

1 Kbyte = 1024 bytes
1 byte = 8 bits

Haciendo cálculos, tenemos:

32 K * 1024 bytes = 32.768 bytes
32.768 bytes * 8 bits = 262144 bits de memoria total.

Pero, como la unidad de asignación definida es de 8 bits llegamos a que la memoria está organizada en 32.768 palabras.

A cada palabra le corresponderá un bit en el mapa que indicará si está ocupado o libre.

1 = palabra asignada
0 = palabra libre o no asignada

Palabra	Mapa de bit
1	1
2	1
3	0
4	0
5	1
6	1

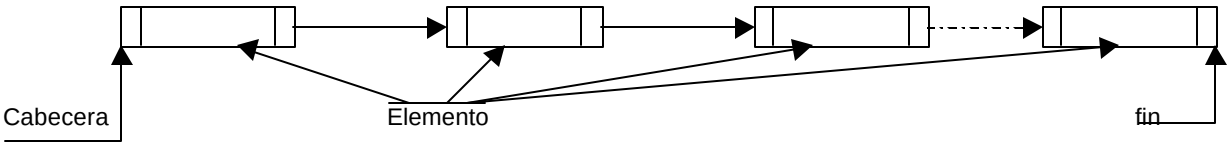
.....
.....
.....
.....
.....
.....

32765	0
32766	0
32767	1
32768	1

Un aspecto importante a mencionar de este enfoque es que si el tamaño de asignación es pequeño, entonces, el mapa de bits es más grande, en tanto que, comparativamente con un bloque de asignación más grande, el mapa de bits es más pequeño. El tamaño del mapa de bits solamente está determinado por el tamaño de la memoria y el tamaño del bloque de asignación.

Administración de memoria con listas enlazadas o ligadas

Este enfoque consiste en asignar la memoria a los procesos que la requieran, mediante un bloque de asignación. El sistema la mantiene el control de la memoria ocupada o libre, mediante una lista enlazada.



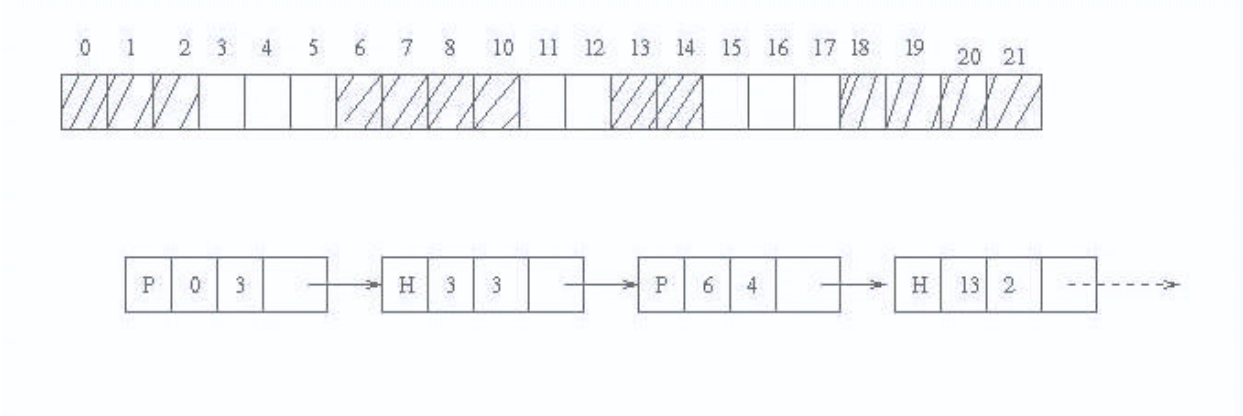
Cada nodo o elemento de la lista está compuesto por los siguientes campos:

- Bit que representa un proceso o un hueco
- Número de bloque de inicio
- Número de bloques utilizados

- Puntero al siguiente nodo en la lista

En el caso de tener una lista ordenada por direcciones de memoria, la ventaja de este enfoque radica en que la actualización de la lista es directa, pues cada nodo tiene dos vecinos, cada uno de los cuales es un proceso o es un hueco. Luego, al desocuparse un segmento representado por un nodo en la lista pasa a ser un hueco, el cual podría eventualmente agruparse a un hueco vecino formando un espacio mas amplio de memoria contigua.

La desventaja de este enfoque es que la búsqueda de un hueco libre suficiente de satisfacer a un proceso es lenta. Para solucionar esta lentitud se pueden tener dos listas ligadas, una para los huecos y otra para los procesos. Sin embargo, esto obliga a mantener estas dos listas, que deben actualizarse cuando un proceso termina y cuando se otorga memoria a un nuevo proceso.



Sistemas de los asociados

Este enfoque consiste en tener la memoria inicialmente como un gran hueco. A medida que un proceso la requiera se particiona, y dicho particionamiento, está regido por segmentos múltiplos de potencias de dos (2 elevado a la n).

Suponga que la memoria es de 1 Mbytes, y un proceso de 56 Kbytes requiere ser cargado. El método funciona de la siguiente manera: la memoria se particiona en dos grandes segmentos de 512 Kbytes; luego este segmento sigue siendo muy grande para contener al proceso, entonces se vuelve a particionar uno de estos dos segmentos en dos segmentos de 256 Kbytes, estos aún siguen siendo grandes, por lo que, otra vez se particiona uno de ellos; luego; se tienen dos segmentos de 128 Kbytes, los que siguen siendo grandes, entonces se vuelven a particionar teniendo dos de 64 Kbytes. Luego, aquí se tienen dos segmentos de 64 Kbytes, el cual es el mínimo tamaño potencia de dos que soporta el proceso que requiere memoria.

Cabe mencionar que para este ejemplo se asignan los 64 Kbytes para un proceso de 56 Kbytes, por lo que existe una fragmentación interna de 8 Kbytes. Además, una vez que el proceso es ejecutado, la memoria que éste libera podrá fusionarse con su asociado para formar un bloque mayor, siempre y cuando el asociado esté libre.

Para el caso del ejemplo, si esta memoria de 1 Mbytes (1024 Kbytes) no hubiera sido requerida por ningún otro proceso, en momento en que el proceso de 56 Kbytes concluye, se irán fusionando los bloques asociados hasta reconstruir los 1024 Kbytes.

En este punto recomendamos fuertemente, leer y contestar las preguntas del ***ANEXO Nº 1*** que encontrarán al final del módulo.

MEMORIA VIRTUAL

Todas las estrategias desarrolladas hasta este punto consisten en asignar espacio en memoria para todo el proceso. Esto significa que si un proceso es más grande que el tamaño del mayor hueco o partición existente (en el caso extremo sería toda la memoria que no es utilizada por el monitor residente) no podría ejecutarse.

Para posibilitar que un proceso o aplicación que posee un tamaño mayor que el de la memoria principal pueda ser ejecutado, se desarrollaron métodos que puedan ir ejecutando trozos del proceso - primero uno, luego otro y así sucesivamente - hasta completarlo. Estos métodos o algoritmos de administración se denominan "memoria virtual".

Los sistemas que operan con memoria virtual generan un espacio de direccionamiento que no corresponde al espacio de direccionamiento real en la memoria principal.

Las direcciones de memoria que generan los procesos por lo general se le conocen como direcciones virtuales o lógicas (porque no existen en memoria principal). Estas direcciones, en el instante de ser ejecutadas, son traducidas a su correspondiente dirección real mediante alguna técnica que por lo general está apoyada en hardware.

El espacio de direccionamiento que generan los procesos corresponde a un espacio ficticio, y debido a ello es que se lo llama **virtual** pues no está formado por las direcciones reales que se colocan en el bus de direcciones, sino que deben pasar por una etapa de traducción o conversión previa (pasar de una dirección virtual a una real).

La utilización de memoria virtual tiene algunas consecuencias importantes desde el punto de vista de los recursos. Conceptualmente, un proceso puede direccionar más memoria que la memoria física disponible. Esto quiere decir, que un proceso de 777 Kbytes, puede ejecutarse con una memoria real de por ejemplo 512 Kbytes.

Esto se realiza en base a que un proceso no necesita estar completamente contenido en memoria para poder ejecutarse, sino que puede estar parcialmente en memoria, y el resto del proceso en disco (memoria secundaria). Luego, el proceso completo puede ser más grande que la memoria física disponible.

La visión del programador es simplificada con el uso de memoria virtual pues ya no tiene que preocuparse por la memoria física; él sólo debe preocuparse del programa.

Muchos programas no requieren tener todo su proceso contenido en memoria. De hecho, en muchos de ellos existe por ejemplo, control de errores, los cuales no siempre se producen, por lo cual, el código asociado a su tratamiento no requiere estar permanentemente en memoria. La reserva de memoria para arreglos grandes tampoco se requiere completamente, pues, en general, el programador sobredimensiona los arreglos al momento de definirlos (por si acaso).

Por lo general, se usan dos técnicas para la implementación de memoria virtual, éstas se conocen como ***Paginación*** **y** ***Segmentación*** .

Paginación

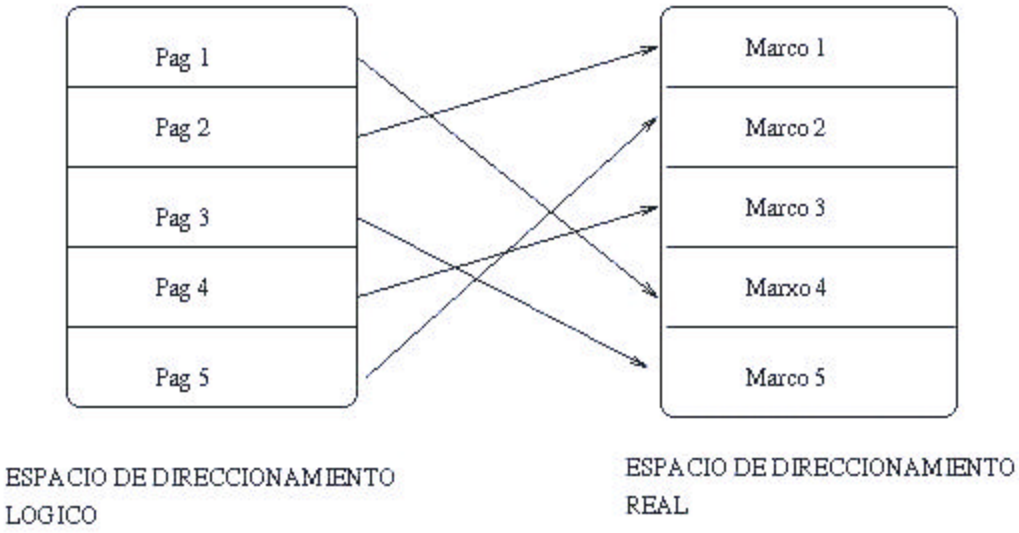
Tanto las particiones de tamaño fijo como las de tamaño variable hacen un uso ineficiente de la memoria; las primeras generan fragmentación interna, mientras que las segundas originan fragmentación externa. Supóngase, que la memoria principal se encuentra particionada en trozos iguales de tamaño fijo relativamente pequeños y que cada proceso está dividido también en pequeños trozos de tamaño fijo y del mismo tamaño que los de la memoria.

Los sistemas que utilizan memoria virtual utilizan como mecanismo de implementación la "paginación"; los trozos del proceso son conocidos como **páginas**, y pueden asignarse a los trozos libres de memoria conocidos como **marcos** o **frame**.

Por lo tanto, la paginación es una técnica que consiste en dividir a los procesos en un conjunto de bloques denominados páginas. Cada página posteriormente va a ser asignada a un marco de página.

Lógicamente, van a existir mucho mas páginas que marcos: de ahí el nombre de virtual.

La utilización de paginación constituye una valiosa ganancia respecto a los sistemas de particiones fijas o variables, vistas anteriormente. Estos mecanismos, obligaban a almacenar completamente el proceso en memoria contigua o vecina, en tanto que la paginación permite no contener al proceso completamente en memoria y por otro lado posibilita la utilización de memoria dispersa. Es decir, no se requiere colocar el proceso en memoria contigua, sino que puede estar en distintas partes de la memoria. A continuación se presenta un esquema de cómo funciona esto.



La traducción de las direcciones virtuales en direcciones reales tiene un componente de hardware agregado, que se denomina, por lo general, como *MMU (Memory Management Unit)* o bien *Mapa de Memoria*. Esta unidad puede estar formada por un chip o un conjunto de chips y, por lo general, se ubican en la tarjeta madre del procesador.

Un esquema simple de administración de memoria, usando **memoria virtual** con **paginación** se puede representar mediante la siguiente dirección virtual:

P	D

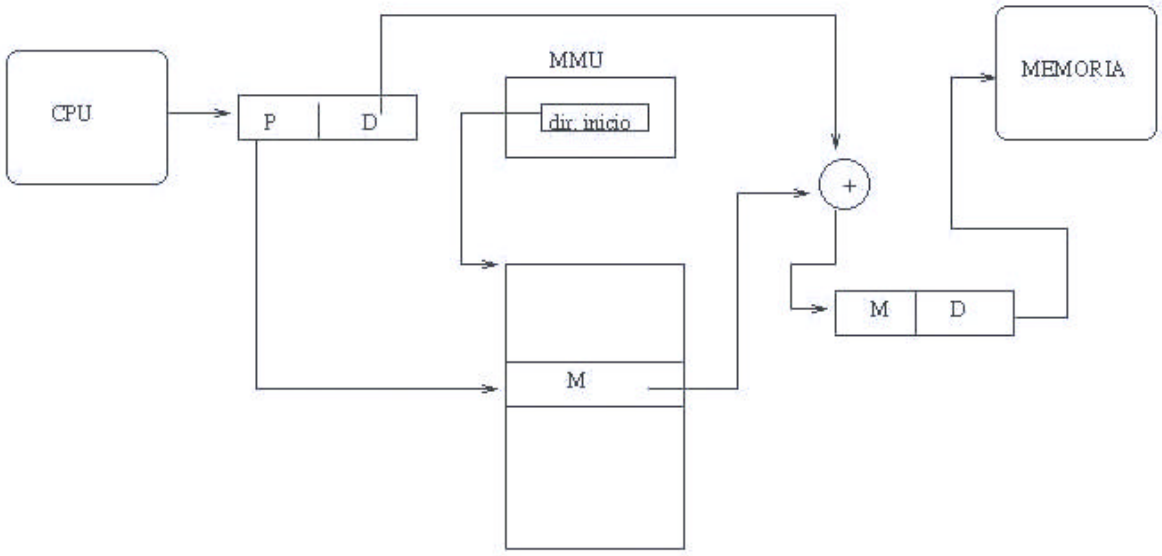
Donde **P** corresponde al número de página y **D** indica el desplazamiento dentro de la página.

Tablas de páginas

La MMU traduce las direcciones virtuales a direcciones reales con el apoyo de una ***tabla de páginas***.

La tabla de páginas corresponde a una tabla que es direccionada o accedida por la primera parte de la dirección virtual, especificada mediante P. Este campo o elemento, se usa como un índice a la tabla de páginas, según la figura anterior. Cada entrada en la tabla de páginas tiene el marco de página asociado a la página y un bit denominado *Presente/Ausente*, que indica si la página se encuentra presente o ausente en la memoria principal. Si el bit esta en 1, entonces, la página tiene asociado un marco de página en memoria

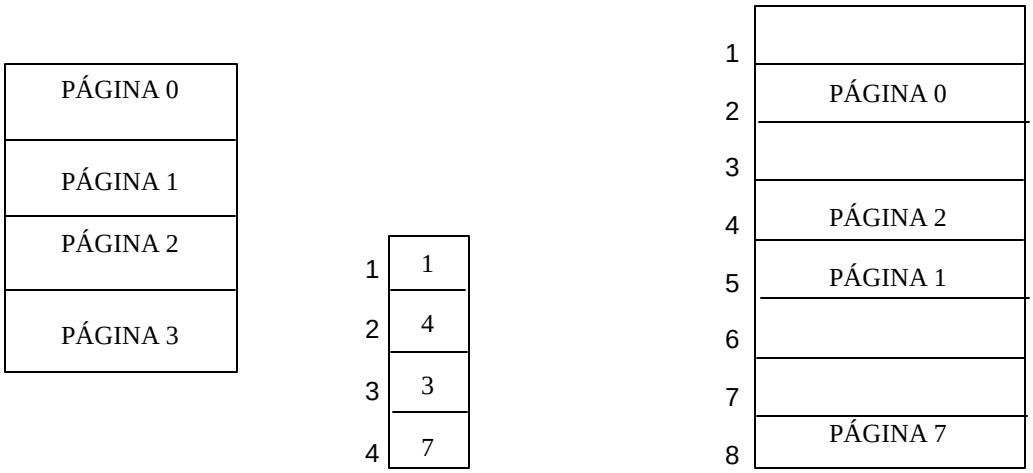
En un sistema con memoria virtual y paginación el mecanismo con una tabla de páginas se presenta a continuación:



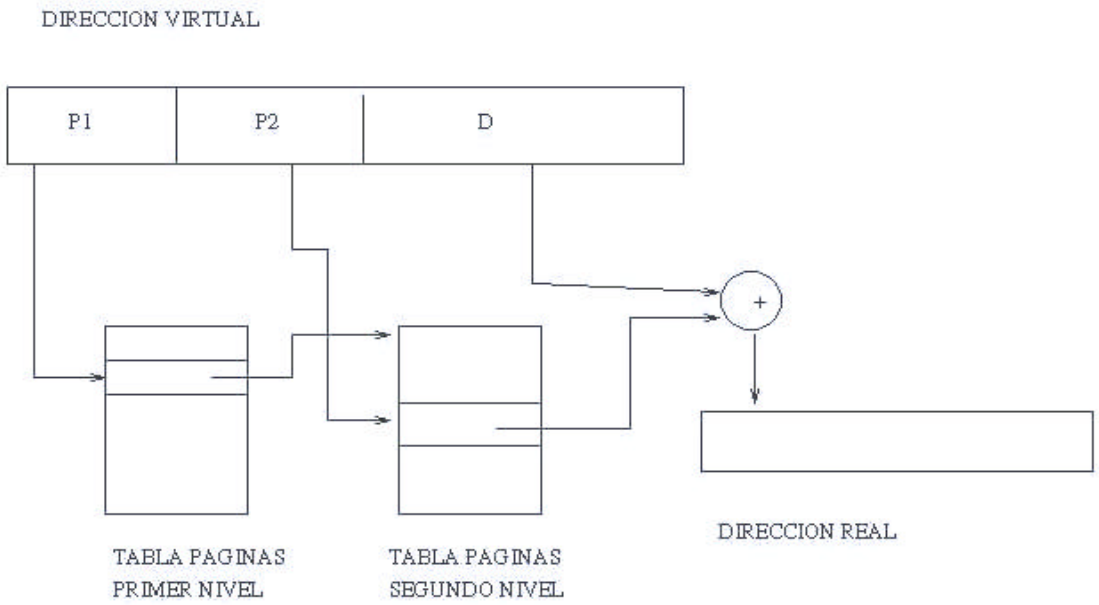
En este gráfico se aprecia claramente que la CPU genera direcciones lógicas, y que la misma está dividida en dos partes:

a) Un número de página (p)
b) Un desplazamiento dentro de la página (d)

El número de página se usa como un índice en la tabla de página. La tabla de páginas contiene la dirección base de cada página de la memoria física. Esta dirección base se combina con el desplazamiento para definir la dirección física que es enviada a la unidad de memoria. El modelo de paginación se muestra en el siguiente esquema:



En la mayoría de los sistemas operativos, la tabla de páginas se encuentra en memoria todo el tiempo. Sin embargo, en los sistemas donde los procesos pueden ser muy grandes se necesitan mecanismos adicionales que permitan aumentar el desempeño del sistema. De esta manera existen MMU donde en vez de trabajar con una única tabla de páginas trabajan con tablas de varios niveles.



En un esquema con múltiples páginas, la dirección virtual esta compuesta por tantos campos como tablas de páginas posee y además el desplazamiento. A continuación se presenta un esquema de memoria virtual con múltiples tablas de páginas.

Formato de entrada en la tabla de páginas

En términos generales, la entrada en la tabla de páginas puede estar representada por el siguiente formato:

En donde se encuentran los siguientes campos:

- Bit *Presente/Ausente*: si es igual a 1 entonces indica que la página tiene asociado un marco de página; si es igual a 0 quiere decir que no tiene asociado un marco de página. Cuando el bit Presente/asusente es 0 al momento de la consulta se dice que ocurre un *fallo de página*. Esto significa que el programa necesita ejecutar instrucciones que no están cargadas en memoria principal, por lo que ésta página deberá cargarse a memoria en el espacio que dejará libre otra página que no sea necesaria en ese momento.

- Bits de *protección*: Puede estar formado por 1 bit o por 3; si es un bit, entonces si el bit esta en 0 indica que tiene privilegios de lectura y escritura; si está en 0, entonces tiene privilegios de solo lectura. Si está formado por 3 bits, entonces cada uno de ellos representa los privilegios de lectura, escritura y ejecución; usándose un 1 para habilitarlos y un 0 para deshabilitarlos.

- Bit *modificado* : Si está en 1, entonces indica que el marco de página asociado a la página ha sido modificado.

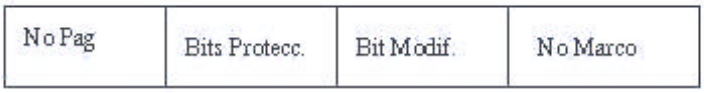
- Bit de *referencia*: Si está en 1 indica qué página ha sido referenciada (leída o escrita). Esta información es utilizada por algunos algoritmos de reemplazo de páginas, que son los mecanismos que actúan cuando se genera un fallo de página y posibilitan cargar a memoria las instrucciones urgentes para que el proceso pueda continuar normalmente.

Memoria Asociativa

Otro mecanismo que utiliza la mayoría de los sistemas de administración de memoria con el fin de aumentar el desempeño es la Memoria Asociativa o también conocida como TLB (Translation Lookaside Buffers).

La memoria asociativa consiste básicamente en un conjunto de registros que se ubican en la MMU con el objetivo de almacenar allí las direcciones de las páginas más referenciadas con sus correspondientes marcos de páginas. La idea con ello es obtener a partir del número de página el marco de página inmediatamente, sin necesidad de consultar la tabla de páginas.

El formato de cada entrada en la memoria asociativa es por lo general de la siguiente manera:



FORMATO MEMORIA ASOCIATIVA

Como se aprecia en la figura, el formato contempla: Número de la página, Bits de protección, el Bit Modificado y el Número de Marco de Página.

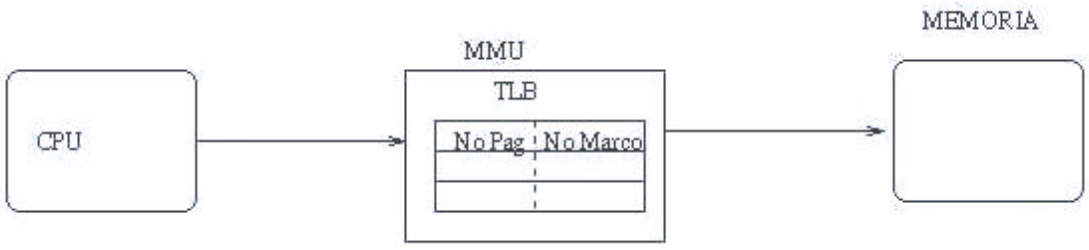
El mecanismo de traducción de dirección virtual a dirección real ocurre de la siguiente manera:

Con el número de página se verifica con cada uno de los registros de la TLB si el número de página coincide con alguno; si se encuentra, entonces se obtiene el marco de página directamente si los bits de protección se lo permiten. Si la página no se encuentra en la TLB, entonces se busca en la tabla de páginas y si hay registros libres en la TLB se anota la información correspondiente a esa página, en ella.

La información mantenida en la memoria asociativa está regida por el grado de utilización de las páginas. Aquellas que se referencien o utilizan con mayor frecuencia se encontrarán ella.

El número de registros pertenecientes a la memoria asociativa es dependiente de la arquitectura. Por lo general son de 16 o 32 registros.

A continuación se presenta un esquema de cómo funciona este mecanismo.



Algoritmos de reemplazo de páginas

En un sistema multiprogramado con intercambio, y en particular sistemas que trabajan con memoria virtual con paginación, el reemplazo de páginas está implícito.

Debido a que un sistema con memoria virtual permite que un proceso no esté contenido en memoria completamente, esto significa que parte de él está en disco o, como se denomina en la jerga "en memoria secundaria". Por lo tanto, si el sistema de memoria virtual trabaja con paginación y el proceso no esta completamente en memoria, entonces posee páginas que tienen asociado un marco de página (están cargadas en memoria principal) y hay otras páginas que no tienen asociado un marco, por lo tanto, esas páginas están en disco.

Cuando las instrucciones que desean ejecutarse están en páginas que no tienen asociadas marcos, se debe buscar marcos libres para asociarlas y cargarlas a memoria principal. Sin embargo, puede ocurrir que el sistema no posea marcos de páginas libres, o bien, el número de marcos libres sea inferior a los exigidos para el buen funcionamiento del sistema. En este caso, el sistema debe ejecutar alguna acción que le permita liberar marcos de páginas para asignárselos a las páginas que lo necesitan. Las páginas que tienen asociadas marcos de páginas que se elijan como donadoras o víctimas de marcos deben ser guardadas en disco. Este mecanismo de reemplazo de páginas, es esencial para poder cargar una página que está siendo referenciada pero no esta cargada (también se lo conoce como **Fallo de Página**).

De estas circunstancias nace la necesidad del sistema por poseer algoritmos que le permitan reemplazar páginas de memoria a disco.

Con el fin de utilizar los mejores criterios para el reemplazo de páginas y a título informativo, es que le proponemos algunos de ellos:

Algoritmo Optimo

La idea de este algoritmo es tener claro cuáles serán las instrucciones que serán ejecutadas en el futuro. Con esta información se pueden planificar qué instrucciones se ejecutarán antes que otras. Una vez que se tenga esta información, el algoritmo puede decidir reemplazar la página asociada a la instrucción con una etiqueta o planificación más alta (utilización más inmediata).

El problema de este algoritmo es que es ideal, pero no es realizable. Al momento de realizarse el reemplazo de página, no se sabe cuál será la siguiente página que se referenciará. Una situación parecida ocurre en uno de los algoritmos de planificación de la CPU denominado SJF.

Algoritmo NRU (reemplazo según el uso no tan reciente)

Este algoritmo consiste en reemplazar aquellas páginas que no han sido utilizadas últimamente. La aplicación de este algoritmo requiere del apoyo de dos de los bits que ya hemos analizado, del bit R (bit de referencia) y del bit M (bit modificado). El bit R es activado cuando se hace referencia a la página (lectura) y el bit M es activado cuando la página es modificada (escritura). El bit R es desactivado periódicamente para poder diferenciar sólo aquellas referencias más recientes.

El algoritmo funciona como un conjunto de clases que se definen de acuerdo a los valores que toman los bits R y M. La clasificación en distintas clases se presenta a continuación:

- Clase 0: R=0 y M=0 , no se ha hecho referencia a la página ni ha sido modificada.
- Clase 1: R=0 y M=1, no se ha hecho referencia a la página, pero sí ha sido modificada.
- Clase 2: R=1 y M=0, se ha hecho referencia a la clase, pero no se ha modificado.
- Clase 3: R=1 y M=1, se ha referenciado y se ha modificado la página.

Cuando ocurre un fallo de página el algoritmo contempla reemplazar aquellas páginas que pertenecen a la clase 0, 1, 2 y 3 en ese orden; es decir, si existen páginas de la clase 0 ellas son las que se reemplazan; si no existen páginas de esa clase se reemplazan páginas de la clase 1, si no hay de la clase 1; se reemplazan las de la clase 2, y así sucesivamente.

Algoritmo FIFO (se reemplaza la página mas antigua)

Este algoritmo se implementa con una lista que posee el sistema operativo acerca de todas las páginas que están en memoria. Dicha lista está ordenada por orden de llegada, es decir, la primera página de la lista es la más antigua y la última, la más nueva.

Cuando ocurre un fallo de página se reemplaza la primera página de la lista. Este algoritmo, no es muy usado, pues no considera la importancia de las páginas ni con qué frecuencia se están utilizando.

Algoritmo de reemplazo de segunda oportunidad

Este algoritmo es el mismo que el FIFO sólo con una variación: se considera la frecuencia con la que son referenciadas las páginas. El algoritmo actúa de la siguiente manera:

Cuando ocurre un fallo de página se revisa la lista de páginas, desde la más antigua a las mas nueva. Si el bit R de la página más antigua es 0, entonces se reemplaza esa página pero si el bit R es 1 entonces se coloca el bit R en 0 y se coloca al final de la lista, luego se consulta por la siguiente página más antigua, y así sucesivamente.

El algoritmo se llama de segunda oportunidad porque se le da a cada página una segunda oportunidad de quedarse en memoria.

El problema de este algoritmo es que es de implementación lenta, pues la lista debe modificarse, es decir se debe sacar un nodo de un sitio para ser insertado en otro lugar.

Algoritmo del reloj

Este algoritmo básicamente es igual al de segunda oportunidad sólo que con una implementación distinta. En este caso se tiene una lista circular, donde existe un puntero que apunta a la página más antigua; luego, cuando ocurre un fallo de página se consulta por el bit R de la página, si es 0 entonces esa página se elige para el reemplazo. En cambio, si el bit R es igual a 1, entonces, el bit R se pone en 0 y se incrementa el puntero de manera que apunte al siguiente nodo de la lista que corresponde a la siguiente página más antigua.

Algoritmo del reloj con dos manecillas

El algoritmo aquí es igual al anterior, sólo que existen dos punteros: un puntero va delante del otro. El primer puntero verifica el bit R de la página, si éste es igual a 1 se pone en 0, luego cuando pase el segundo puntero verifica el estado del bit R. Si es 0 entonces intercambia la página, sino lo pone en 0 y apunta a siguiente página. El objetivo de este algoritmo es privilegiar a aquellas páginas más referenciadas para que se queden en memoria.

Algoritmo LRU (reemplazo de página de menor uso reciente)

Este algoritmo consiste en reemplazar aquella página que no hubiere sido referenciada en el mayor período posible de tiempo. La idea de este algoritmo es que las páginas que no se han referenciado en un buen período de tiempo difícilmente serán referenciadas en un tiempo cercano.

La implementación de este algoritmo, por lo general, es cara. Una forma de hacerlo es mantener una lista enlazada de todas las páginas que tienen asociado un marco de página ordenada de tal manera que la primera página de la lista es la de uso más reciente y la última es la de uso menos reciente. El problema surge porque la lista debe ser actualizada continuamente después que cada página sea referenciada, y luego mover dicha página al principio de la lista.

Utilizando hardware adicional es posible implementar una solución en hardware. Una de estas soluciones consiste en tener un contador por página el cual se incrementa después que la página es referenciada. Luego, cuando ocurre un fallo de página se elige aquella página que posea el contador con menor valor numérico.

Aspectos de diseño

Tamaño de la página

El tamaño de la página es un aspecto de diseño importante de considerar. Si la página es muy pequeña se necesita utilizar una gran cantidad de recursos del sistema para poder administrarlas, además, cada proceso requerirá de más páginas para almacenarse en memoria.

Por otro lado, si la página es muy grande se pierde más memoria por concepto de fragmentación interna.

El tamaño óptimo debe calcularse considerando los aspectos mencionados, más el tamaño promedio de los procesos que se ejecutan en el sistema.

Páginas compartidas

Otro aspecto de diseño importante de considerar se refiere a la compartición de páginas. En un sistema multiusuario–multitarea es frecuente que distintos usuarios ejecuten una misma aplicación simultáneamente. En estas condiciones el sistema debería permitir compartir las páginas asociadas al código de la aplicación.

Suponga, por ejemplo que el usuario 1 y el usuario 2 están usando el mismo editor de texto, el sistema debería dejar como compartidas las páginas que estén relacionadas con el código ejecutable de la aplicación y las páginas de datos de cada uno de los usuarios deberían ser particulares a cada uno de ellos.

Dado que pueden surgir algunas situaciones no deseadas con el manejo de páginas compartidas, el sistema maneja estructuras especiales para el control de estas páginas.

Una de estas situaciones, por ejemplo, puede ser que el usuario1 termine de editar su archivo y salga del editor. Lo que podría ocurrir es que las páginas fueran reemplazadas, lo cual iría en desmedro del usuario que tendría que volver a cargarlas en memoria.

Algunos autores asocian el concepto de reentrancia con las páginas compartidas, en el sentido que si el código es reentrante entonces puede ser compartido. Que el código sea reentrante no se modifica a sí mismo, de manera que dos o más procesos pueden ejecutar el mismo código al mismo tiempo.

Segmentación

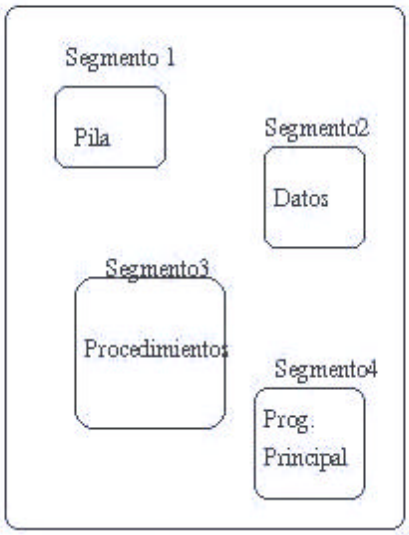
La paginación es una técnica que permite aislar al usuario de la perspectiva de la memoria real, en la que el usuario tiene una visión sobre un espacio de direccionamiento lógico exclusivamente. Sin embargo, la perspectiva del usuario con la paginación es un único gran espacio de direccionamiento en donde se encuentra tanto los datos, procedimientos, funciones y el programa principal, sin ningún orden predeterminado.

Esta visión no es la que por lo general poseen los usuarios respecto a la forma que se almacena esta información. Los usuarios tienden a pensar que los datos se almacenan en algún lugar determinado; los procedimientos y funciones en otro lugar y el programa principal en otro lugar. Esta perspectiva es mas bien lógica sobre el lugar donde se encuentra almacenada la información.

De esta visión surge la técnica denominada segmentación. Con ella, en un sistema de administración de memoria se distinguen diferentes espacios de direccionamiento, los cuales reciben el nombre de segmentos. Cada segmento tiene una serie lineal de direcciones de un tamaño determinado. Una de las principales características de los

segmentos es que son de longitud variable, es decir que pueden crecer, a diferencia de las páginas. Por otro lado, en un sistema de administración de memoria existen segmentos de variados tamaños, vale decir, no son todos iguales. Dado que los segmentos son espacios de dirección independientes uno de otros y además como son de longitud variable, cada uno de ellos puede crecer en forma independiente sin afectar ni ser afectado por los demás.

A continuación se presenta un esquema que refleja el concepto de segmento.



ESPACIO DE DIRECCIONAMIENTO LOGICO

Desde el punto de vista de "información compartida", los segmentos proporcionan más facilidades que en el caso del manejo de páginas compartidas, puesto que como los segmentos son entidades lógicas es posible tratar un segmento como compartido, sin necesidad que los procesos contengan el segmento en su propio espacio de direccionamiento. Con la utilización de páginas compartidas vimos que habían ciertos inconvenientes en el uso de ellas.

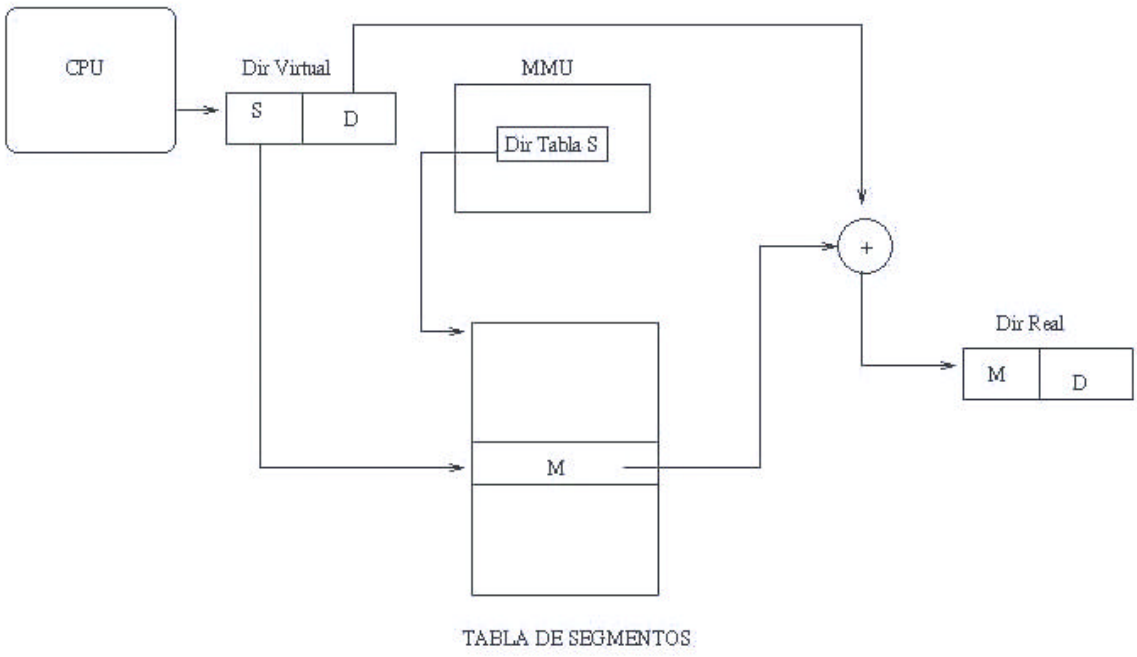
Un problema que adolece a la administración con segmentación es la fragmentación externa, pues cuando salen continuamente diversos segmentos de memoria y se van incorporando otros nuevos, la memoria es fragmentada en diversos huecos que en algunos casos pueden ser demasiado pequeños para alcanzar a contener a otro.

La implementación de administración de memoria con segmentación puede ser básicamente de dos tipos:

- Segmentación pura
- Segmentación con paginación

Segmentación pura

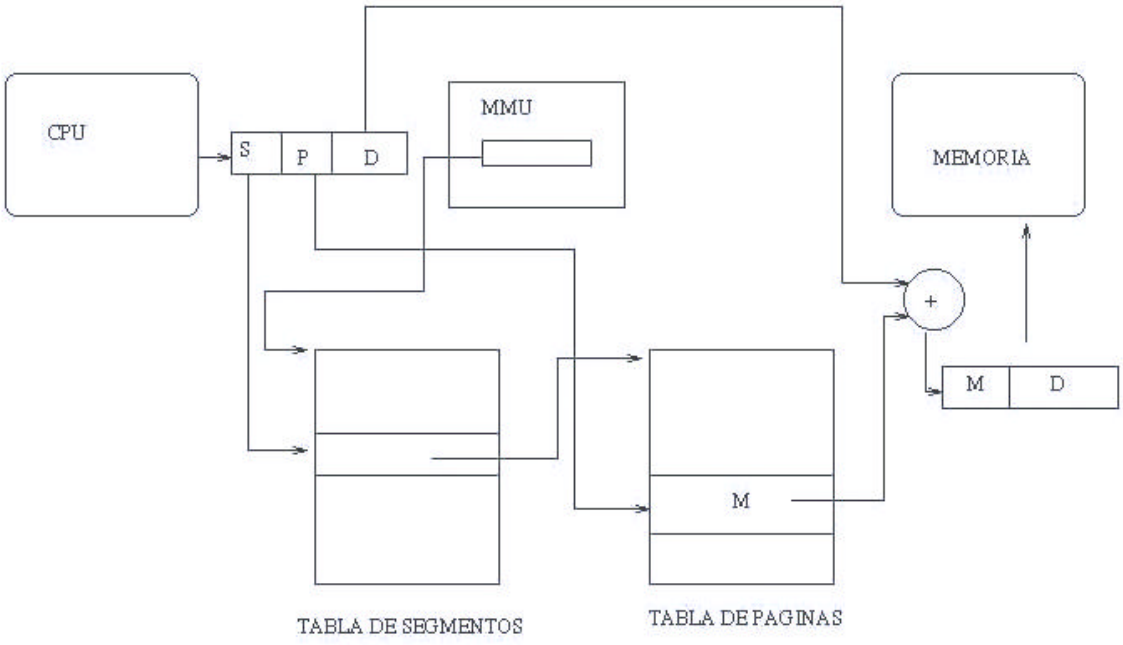
En este esquema sólo se tienen segmentos para representar el direccionamiento lógico. El sistema de administración de memoria con segmentación pura se representa de manera lógica mediante un nombre y un tamaño que puede variar. Sin embargo, para facilitar la implementación, los segmentos se identifican mediante un número. La implementación de un sistema de administración con segmentación contempla un par de registros *base* y *límite* y una tabla de segmentos. Su relación se representa mediante el siguiente diagrama.



Segmentación con Paginación

En este esquema se identifican las dos técnicas que hemos visto: segmentación y paginación. Este tipo de administración contempla la división del espacio de direccionamiento lógico en un conjunto de segmentos, en donde cada segmento tiene asociado a su vez un conjunto de páginas. A continuación se presenta un esquema de como funciona este tipo de administración.

La combinación de estos dos esquemas ha sido adoptada por muchos microprocesadores con el fin de aminorar las desventajas que poseen cada una de estas técnicas.



En este punto recomendamos fuertemente, leer y contestar las preguntas del ***ANEXO Nº II*** que encontrarán al final del módulo.

ANEXO I:

Resumen

Una de las tareas más importantes y complejas de un sistema operativo es la gestión de memoria. La gestión de memoria implica tratar la memoria principal como un recurso para asignar a los procesos y compartir entre varios procesos activos. Para un uso eficiente del procesador y de los servicios de E/S, resulta interesante mantener en memoria principal tantos procesos como sea posible.

Hasta este punto hemos considerado en todos los casos tratados, que el proceso a ejecutarse debe cargarse totalmente en memoria principal.

Arrancamos nuestro estudio de la siguiente manera:

1. Planteamos la necesidad de un MONITOR RESIDENTE en memoria para que asigne y administre al único proceso que podía cargarse en memoria

2. Luego, se propuso trabajar con multiprogramación, es decir, posibilitar que más de un trabajo pudiera estar cargado en memoria principal. Para lograr este propósito explicamos Multiprogramación con Particiones Variables (MVT) y Multiprogramación con Particiones Fijas (MFT), pasando previamente por analizar la técnica conocida como Swapping (la cual constituye la transición o etapa intermedia entre la Monoprogramación y la Multiprogramación). Hemos mencionado también los problemas que ellas generan: fragmentación interna y externa

3. Por último, expusimos algunos métodos o algoritmos que utiliza el SO para administrar y manejar los espacios asignados y ocupados de la memoria. Para ello vimos: mapa de bits, sistemas de los asociados, asignación del primer ajuste, del siguiente y mejor.

ANEXO II:

Resumen

Para un aprovechamiento eficiente del procesador y de los servicios de E/S es conveniente mantener tantos procesos en memoria principal como sean posibles. Con memoria virtual, todas las referencias a direcciones son referencias lógicas que se traducen a direcciones reales o físicas durante la ejecución.

La memoria virtual permite dividir un proceso en fragmentos, trozos o pedazos para su ejecución. Estos fragmentos no tienen por qué estar situados de forma contigua en la memoria principal durante la ejecución y no es ni siquiera necesario que estén cargados en memoria principal todos los fragmentos del proceso durante la ejecución.

Los dos enfoques básicos de memoria virtual son la paginación y la segmentación.

Con la Paginación, cada proceso se divide en páginas de tamaño fijo y relativamente pequeños, por lo que la fragmentación interna se genera únicamente en la última página asociada al proceso (es mínima).

La segmentación, por el contrario, permite el uso de tamaños variables y minimiza aún más la fragmentación.

BIBLIOGRAFIA

- SISTEMAS OPERATIVOS, Williams Stallings, Segunda Edición, Ed PRENTICE HALL
- SISTEMAS OPERATIVOS MODERNOS, Andrew Tanenbaum, Tercera Edición, Ed PRENTICE HALL
- Apuntes de la Cátedra COMPUTACION II, Universidad Tecnólogica Nacional -FRR
- Apuntes de la Cátedra SISTEMAS OPERATIVOS I, Instituto Universitario Gastón Dachary
- http://www.inf.udec.cl