https://repository.uaeh.edu.mx/revistas/index.php/huejutla/issue/archive

Ciencias Huasteca Boletín Científico de la Escuela Superior de Huejutla

Ciencia Huasteca
Boletín Científico de la Escuela Superior de Huejutla

Publicación semestral No. 14 (2019) 23-32

Gestión de memoria

Memory management

Angélica Carrión [a], Manuel Celi [b], Eduardo Galvez [b], José García [b], Rafael Guerrero [b], Daniela Moreira [b], Miguel Suárez [b], Suanny Tigselema [b], Carlos Tulcán [b]

Abstract:

Memory management is one of the most important tasks performed by the Operating System, always looking for the best performance, and applying methods and operations that help us improve the use of resources. The system is going to be ordered, by means of its memory management burns, in such a way that the available memory is used appropriately. Working through the processor that is going to be designated for these operations as virtual memory.

Keywords:

Physical memory, virtual memory, paging, segmentation, fragmentation.

Resumen:

La administración de la memoria es una de las tareas más importantes que realiza el Sistema Operativo, buscando siempre el mejor rendimiento, aplicando métodos y operaciones que ayuden a mejorar el uso de los recursos. El sistema se va a encargar, por medio de sus esquemas de manejo de memoria, de tal forma que se aproveche de manera adecuada la memoria disponible. Trabajando por medio del procesador que va a ser designado para estas operaciones como memoria virtual.

Palabras Clave:

Memoria física, memoria virtual, páginación, segmentación, fragmentación

Introducción

El Sistema Operativo (SO) es el cerebro de una computadora, y como tal, trabaja en utilizar de manera óptima sus recursos; y como no es diferente, optimiza el uso de la memoria por medio de métodos, operaciones y algoritmos, para utilizar de la mejor manera la memoria disponible, tanto la virtual como la física. Dichas operaciones son muy importantes para optimizar el uso de los recursos disponibles, pero también cuidan la parte física de los componentes, como lo son disco duro y memoria RAM, y obviamente aumentando la velocidad de respuesta de cada proceso y petición de cualquier programa o del sistema. Cada operación o método maneja algunas maneras de realizar esa optimización, tal como uso de tablas, algoritmos, entre otros que van a permitir lograr el objetivo.

Desarrollo

La tarea que realiza el SO consiste en gestionar la jerarquía de memoria, en cargar y descargar procesos en memoria principal para que sean ejecutados. Para ello el SO gestiona la Unidad de Administración de Memoria (MMU), la cual es un dispositivo hardware que transforma las direcciones.

La función de la MMU es seguir la pista de las partes de la memoria que están en uso y libres, con el fin de poder asignar memoria a los procesos cuando la necesiten y recuperar esa memoria cuando dejen de necesitarla, así como gestionar el intercambio entre memoria principal y el disco cuando la memoria principal resulte demasiado pequeña para contener a todos los procesos. La administración o gestión de memoria es el uso de la misma por medio de métodos, operaciones y algoritmos,

[a] Autor de Correspondencia, Universidad Técnica Estatal de Quevedo, Facultad de Ciencias de la Ingeniería, Email: angelicangon.carrion@uteq.edu.ec

[b] Universidad Técnica Estatal de Quevedo, Facultad de Ciencias de la Ingeniería, Email: celiyelamc123@hotmail.com, e-galvez-carmigniani@hotmail.com, jgarcia24121996@gmail.com, raft_thebadboy1997@hotmail.com, mishellmoreira_1997@hotmail.com, migisuarez499@gmail.com, suannytigselema@hotmail.com, c_pvct@hotmail.com

para utilizar de la mejor manera la memoria disponible, tanto la virtual como la física.

Paginación

En la memoria virtual las direcciones virtuales van a una MMU que asocia las direcciones virtuales a las direcciones de memoria física [1]. Por tal razón, la mayor parte de los sistemas de memoria virtual utilizan la paginación ya para asociar las direcciones virtuales a las direcciones de memoria física. Debido a que los procesos varían de tamaño, si el procesador planifica un determinado número de procesos, es difícil almacenarlos compactamente en memoria principal, de esa manera nacen los sistemas de paginación [2]. El tamaño de los procesos no es fijo al contrario, es muy variable por lo tanto al procesador se le hace difícil almacenar compactamente dichos procesos en la memoria principal, por esta razón se usa la paginación para dividir la memoria principal y ubicar ordenadamente ciertas partes del proceso. En la paginación, el espacio virtual de direcciones se divide en unidades llamadas páginas, todas del mismo tamaño. La memoria principal se divide en marcos de página del mismo tamaño que las páginas virtuales y son compartidas por distintos procesos del sistema (en cada marco de página se carga una página de un proceso) [3]. Es decir, los procesos se dividen en pequeñas partes (páginas) al igual que la memoria principal en un conjunto de bloques de igual tamaño (marcos de página), de esta manera cada página de un proceso se ubica en los marcos de página disponibles de la memoria principal. Como es de imaginarse, los procesos más pequeños requieren menos páginas y los más grandes requieren más páginas.

Tabla de páginas

En una implementación simple, la asociación de direcciones virtuales a direcciones físicas, la dirección virtual se divide en un número de página virtual (bits de mayor orden) y en un desplazamiento (bits de menor orden). El número de página virtual se utiliza como índice en la tabla de páginas para buscar la entrada para esa página virtual. En la entrada, en la tabla de páginas se encuentra el número de marco de página. El número del marco de página se adjunta al extremo de mayor orden del desplazamiento, reemplazando el número de página virtual, para formar una dirección física que se pueda enviar a la memoria. El propósito de la tabla de páginas es asociar páginas virtuales a los marcos de páginas, 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. Utilizando el resultado de esta función, el campo de la página virtual en una dirección virtual se puede reemplazar por un campo de marco de página, formando así una dirección de memoria física.

Estructura de una entrada en la tabla de páginas

La distribución de una entrada depende de la máquina, el tamaño varía de una computadora a otra, pero 32 bits es un tamaño común. El campo más importante es el número de marco de página. En seguida de este campo tenemos el bit de 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. Al acceder a una entrada en la tabla de página con este bit puesto en 0 se produce un fallo de página. En la

Figura 1. Estructura de una entrada en la tabla de páginas

se observa la estructura de una entrada en la tabla de páginas.

Figura 1. Estructura de una entrada en la tabla de páginas

A continuación, se describen algunos de los bits más importantes:

Bit de protección: Indica qué tipo de acceso está permitido, este campo contiene 1 bit, con 0 para lectura/escritura y 1 para sólo lectura. Un arreglo más sofisticado es tener 3 bits: uno para habilitar la lectura, otro para la escritura y el tercero para ejecutar la página.

Bit de modificada: Cuando se escribe en una página, el hardware establece de manera automática el bit de modificada. Este bit es valioso cuando el SO decide solicitar un marco de página. Si la página en él ha sido modificada, debe escribirse de vuelta en el disco. Si no se ha modificado sólo se puede abandonar, debido a que la copia del disco es aun válida. A este bit se le conoce algunas veces como bit sucio, ya que refleja el estado de la página.

Bit de referencia: Se establece cada vez que una página es referenciada, ya sea para leer o escribir, ayuda al SO a elegir una página para desalojarla cuando ocurre un fallo de página.

Implementación de las tablas de páginas

La paginación ayuda a resolver muchos de los problemas que ocurren en el particionamiento de memoria, al dividir la memoria principal utilizada por el SO en muchos marcos de página pequeños, todos estos de un mismo tamaño. Cada proceso de un programa (o los procesos directamente) se dividen en páginas de un mismo tamaño que los marcos de páginas, permitiendo que se establezcan el número de páginas que se van a ocupar en memoria (los procesos grandes ocupan más número de páginas que los procesos pequeños), las cuales van a permitir que el proceso se ejecute con normalidad. Al ejecutarse un proceso, todas las páginas en las cuales fue dividido se cargan en los marcos de página que tenga disponible la memoria, las mismas que crean una tabla de páginas la cual va a utilizar el procesador para interactuar entre la memoria lógica y la memoria física. El procesador utiliza la tabla de páginas que contiene la dirección del marco de memoria donde se encuentra ubicada la página (número de página y el desplazamiento dentro de ella) para traducirla a una dirección física a la cual accede para ejecutar el o los procesos cargados [2]. Para saber la forma en la que se ejecutan los procesos, debemos conocer las formas en las que se implementan las tablas de páginas, estas son:

Registros dedicados: El microprocesador cuenta con una serie de registros en los cuales almacena las tablas de página de los procesos que está ejecutando. Para ejecutar nuevos procesos, se guardan las tablas de páginas de los procesos que finalizan y se liberan de la CPU para dar paso a nuevas tablas de páginas. Este método permite que se ejecuten con mayor velocidad los procesos, aunque resulta muy costoso aplicarlo porque requiere de muchos registros para ponerlo en funcionamiento. Tiene un cambio de contexto lento al tener que guardar una tabla de página de un proceso para poder trabajar con las tablas de páginas de los nuevos procesos que va a ejecutar la CPU.

En memoria: La memoria se encarga de almacenar las tablas de páginas. Para leerlas, el microprocesador guarda en un registro la dirección en memoria de la tabla de página del proceso que está ejecutando. Este método es más lento que el anterior, porque para acceder a un proceso que realiza dos lecturas, la primera es para acceder a la tabla de página, estableciendo qué dirección de memoria ocupa y el segundo es para acceder a la dirección antes obtenida con el primer acceso.
El cambio de contexto es más rápido al solo tener que modificar el valor del registro de la tabla de página del proceso saliente por el valor del registro del nuevo proceso que va a ejecutar la CPU.

Algoritmos de Paginación

Cuando se trata de la administración de memoria existe un algoritmo llamado *Algoritmo de Paginación*. Para tal efecto, son 5 los algoritmos comúnmente utilizados que a continuación se describen:

Algoritmo de sustitución de página optima

El algoritmo de sustitución de página óptimo tiene como característica principal eliminar la página que tenga el rotulo más alto, trata de aplazar los sucesos desagradables el mayor tiempo que se pueda y es fácil de describir, pero difícil de implementar [4].
Debido a que, para la implementación de este algoritmo se debe conocer el contenido de la página que no se puede realizar la implementación, pero según [5] este algoritmo es utilizado como referencia para el desarrollo de los demás algoritmos.

Las características según [6] son las siguientes:
- Reemplaza la página que se requerirá en el punto más lejano.
- Optimo, pero no lograble.
- La estimación se basa en el registro de uso de las corridas anteriores del proceso.

Sigue siendo poco práctico, pero como ya se ha mencionado anteriormente, este algoritmo es muy útil ya que nos elimina las páginas que no utilizaremos en el momento minimizando el tiempo al citar cada página.

Algoritmo de sustitución de páginas no usadas

Este algoritmo se aplica cuando ocurre una falla de página, el SO examina todas las páginas y las divide en cuatro categorías con base a sus valores. También supone que es mejor eliminar una página modificada a la que por lo menos no se ha hecho referencia en por lo menos un tic del reloj que una página limpia que no se está usando [4]. Cuando se inicia un proceso, el SO pone en 0 los bits de todas sus páginas. Periódicamente (ej. en cada interrupción de reloj), se apaga el bit R, a fin de distinguir las páginas a las que no se ha hecho referencia recientemente de las que si se han leído [7]. Los cuatro estados en que se clasifican: - Clase 0: no referida, no modificada - Clase 1: no referida, modificada - Clase 2: referida, no modificada - Clase 3: referida, modificada.

Algoritmo FIFO

Es de paginación con bajo gasto extra, el SO mantiene una lista de todas las páginas que están en la memoria, siendo la página que está en la cabeza de la lista más

vieja y del final, lo más reciente [4]. El termino FIFO seguramente usado en la programación se refiere a, *el primero en llegar a la fila será el primero en ser atendido*, en este caso será la primera página que se encuentre referenciada al inicio y mientras van llegando serán referenciadas en el orden en que se encuentran las páginas. En base a la Figura 2, se explicará el ejemplo:

A	B	C	D	E	F	G
1	2	3	4	5	6	7

Figura 2. Algoritmo FIFO

Según se observa, en el ejemplo en cada parte inferior de las celdas está el número de orden en las cuales serán referenciadas las páginas, cada página se va referenciada en el orden en que se encuentran desde la primera hasta la última.

Algoritmo de sustitución de página por reloj

Este algoritmo mantiene todas las páginas en una lista circular con forma de reloj, y una manecilla apunta a la página más vieja [4]. Se tiene un beneficio, ya que con este algoritmo se puede saber qué páginas liberar de la memoria en el caso de que no quede espacio y necesitemos agregar una nueva página. Cuando se presenta un fallo de página, el algoritmo revisa la página a la que está apuntando la manecilla. Si el bit de referencia es 0, la página es reemplazada con la nueva y la manecilla avanza una posición. Si el bit es 1, entonces se limpia (cambia a 0) y la manecilla avanza a la siguiente página y así sucesivamente hasta encontrar una con bit 0 [5].
En la Figura 3 se explica el funcionamiento del algoritmo. Tal como se observa, las manecillas recorren el conjunto de páginas en memoria, como se explicó anteriormente la manecilla gira dentro de las páginas buscando la página cuyo bit sea 0 y esta es sacada para ubicar una página nueva y así ser referenciada.

Figura 3. Algoritmo de sustitución de página por reloj

Algoritmo LRU

Este algoritmo consiste en que si las páginas que se han usado mucho en las últimas instrucciones probablemente se usaran mucho en las siguientes. Se aplica cuando ocurre una falla de página, este desalojará la página que haya estado más tiempo sin usarse. El SO examina todos los contadores de la tabla de páginas hasta encontrarse el más bajo [4]. Este algoritmo es una buena aproximación al óptimo y se basa en la observación de que las páginas de uso frecuente en las últimas instrucciones se utilizan con cierta probabilidad en las siguientes [8]. Este algoritmo busca deshacerse de la página menos utilizada para remplazarla con una página nueva, De igual manera, busca también eliminar las páginas que no hayan sido usadas en un largo lapso de tiempo liberando así más memoria para su utilización. Por lo cual, el algoritmo recibe el nombre de LRU por sus siglas en inglés (Least Recent Use).

Particiones

De las opciones de concesión de memoria, las particiones permiten asignar a cada programa a ser ejecutado un bloque contiguo de memoria de un tamaño fijo. En tanto los programas permitieran la resolución de direcciones en tiempo de carga o de ejecución, podrían emplear este esquema.
El SO emplearía una región específica de la memoria del sistema (comúnmente la región baja desde la dirección en memoria 0x00000000 hacia arriba), y una vez terminado el espacio necesario para el núcleo y sus estructuras, el sistema asigna espacio a cada uno de los procesos. Si la arquitectura en cuestión permite limitar los segmentos disponibles a cada uno de los procesos, esto sería suficiente para alojar en memoria varios procesos y evitar que interfieran entre sí, tal como se observa en la Figura 4.

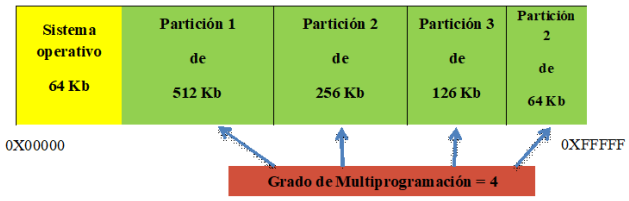

Figura 4. Uso de particiones

Desde la representación del SO, cada uno de los espacios asignados a un proceso es una partición. Cuando el SO inicia, toda la memoria disponible es vista como un solo bloque, y conforme se van ejecutando procesos, este bloque va siendo subdividido para satisfacer sus requisitos. Al cargar un programa el SO calcula cuánta memoria va a requerir a lo largo de su vida prevista. Esto incluye el espacio requerido para la

asignación dinámica de memoria con la familia de funciones *malloc* y free usadas para la gestión de memoria por el SO [9].

Asignación de memoria contigua

La memoria principal debe albergar tanto el SO como los diversos procesos de usuario. Por tanto, es necesario asignar las distintas partes de la memoria principal de la forma más eficiente posible. Esta sección explica uno de los métodos más comúnmente utilizados, la asignación contigua de memoria. La memoria está usualmente dividida en dos particiones: una para el SO residente y otra para los procesos de usuario. Podemos situar el SO en la zona baja 0 en la zona alta de la memoria. El principal factor que afecta a esta decisión es la ubicación del vector de interrupciones. Puesto que el vector de interrupciones se encuentra a menudo en la parte baja de la memoria, los programadores tienden a situar también el SO en dicha zona. Por tanto, en la explicación, sólo se analiza la situación en la que el SO reside en la parte baja de la memoria, aunque las consideraciones aplicables al otro caso resultan similares.
Normalmente, se pretende tener varios procesos de usuario residentes en memoria al mismo tiempo. Por tanto, tenemos que considerar como asignar la memoria disponible a los procesos que se encuentren en la cola de entrada, esperando a ser cargados en memoria. En este esquema de asignación contigua de memoria, cada proceso está contenido en una única sección contigua de memoria [10].

Fragmentación

Es un fenómeno que se manifiesta a medida que los procesos terminan su ejecución, y el SO libera la memoria asignada a cada uno de ellos. Si los procesos se encontraban en regiones de memoria, apartadas entre sí, comienzan a aparecer regiones de memoria disponible, interrumpidas por regiones de memoria usada por los procesos que aún se encuentran activos. Si la computadora no tiene hardware específico que permita que los procesos resuelvan sus direcciones en tiempo de ejecución, el SO no puede reasignar los bloques existentes, y aunque pudiera hacerlo, mover un proceso entero en memoria puede resultar una operación costosa en tiempo de procesamiento.
Al crear un nuevo proceso, el SO tiene tres estrategias según las cuales podría asignarle uno de los bloques disponibles:

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

Mejor ajuste: El sistema busca entre todos los bloques disponibles cuál es el que mejor se ajusta al tamaño requerido por el nuevo proceso. Esto implica la revisión completa de la lista de bloques, pero permite que los bloques remanentes, una vez que se ubicó al nuevo proceso, sean tan pequeños como sea posible (esto es, que haya de hecho un mejor ajuste).

Peor ajuste: El sistema busca cuál es el bloque más grande disponible, y se lo asigna al nuevo proceso. Empleando una estructura de datos como un montículo, esta operación puede ser incluso más rápida que la de primer ajuste. Con este mecanismo se busca que los bloques que queden después de otorgarlos a un proceso sean tan grandes como sea posible, de cierto modo balanceando su tamaño [9]. La fragmentación externa se produce cuando hay muchos bloques libres entre bloques asignados a procesos; la fragmentación interna se refiere a la cantidad de memoria dentro de un bloque que nunca se usará; por ejemplo, si el SO gestiona bloques de 512 bytes y un proceso requiere sólo 768 bytes para su ejecución, el sistema le entregará dos bloques (1024 bytes), con lo cual desperdicia 256 bytes.

Compactación

Un problema importante que va surgiendo como resultado de esta fragmentación es que el espacio total libre de memoria puede ser mucho mayor que lo que requiere un nuevo proceso, pero al estar fragmentada en muchos bloques este no encontrará una partición contigua donde ser cargado. Si los procesos emplean resolución de direcciones en tiempo de ejecución, cuando el SO comience a detectar un alto índice de fragmentación, puede lanzar una operación de compresión o compactación. Esta operación consiste en mover los contenidos en memoria de los bloques asignados para que ocupen espacios contiguos, permitiendo unificar varios bloques libres contiguos en uno solo. La compactación tiene un costo alto de cómputo debido a que involucra mover prácticamente la totalidad de la memoria (posiblemente más de una vez por bloque) [9].

Intercambio (*swap*) con el almacenamiento secundario

Continuando, de cierto modo la lógica requerida por la compactación se encuentran los sistemas que utilizan intercambio (*swap*) entre la memoria primaria y secundaria. En éstos, el SO puede comprometer más espacio de memoria del que tiene físicamente disponible. Cuando la memoria se acaba, el sistema suspende un proceso (comúnmente lo hace a un proceso "bloqueado") y almacena una copia de su imagen en memoria en almacenamiento secundario para luego poder restaurarlo. Hay algunas restricciones que observar previo a suspender un proceso. Por ejemplo, se debe considerar si el proceso tiene pendiente alguna operación de E/S, en la cual el resultado se deberá copiar en su espacio de memoria. Si el proceso resultara suspendido (retirándolo de la memoria principal), el SO no tendrá dónde continuar almacenando estos datos conforme llegan. Una estrategia ante esta situación podría ser que todas las operaciones se realicen únicamente a *buffers* (regiones de memoria de almacenamiento temporal) en el espacio del SO, y éste transfiera el contenido del buffer al espacio de memoria del proceso suspendido una vez que la operación finalice.

Esta técnica se popularizó en los sistemas de escritorio hacia finales de los 1980 y principios de los 1990, en que las computadoras personales tenían típicamente entre 1 y 8 MB de memoria. Se debe considerar que las unidades de disco son del orden de decenas de miles de veces más lentas que la memoria, por lo que este proceso resulta muy caro. Por ejemplo, si la imagen en memoria de un proceso es de 100 MB, bastante conservador hoy en día, y la tasa de transferencia sostenida al disco de 50 MB/s, intercambiar un proceso al disco toma dos segundos. Cargarlo de vuelta a la memoria toma otros dos segundos, y a esto debe sumarse el tiempo de posicionamiento de la cabeza de E/S del disco, especialmente si el espacio a emplear no es secuencial y contiguo. Resulta obvio porque esta técnica ha caído en desuso conforme aumenta el tamaño de los procesos [9].

Segmentación

La segmentación es un esquema de manejo de memoria mediante el cual la estructura del programa refleja su división lógica, llevándose a cabo una agrupación lógica de la información en bloques de tamaño variable denominados segmentos.

La segmentación es una técnica de gestión de memoria que pretende acercarse más al punto de vista del usuario. Los programas se desarrollan, generalmente, en torno a un núcleo central (principal) desde el que se ramifica a otras partes (rutinas) o se accede a zonas de datos (tablas, pilas, etc.). La segmentación de un programa la realiza el compilador y en ella cada dirección lógica se expresará mediante dos valores: número de segmento y desplazamiento dentro del segmento.

Cada uno de los segmentos tiene información lógica del programa: subrutina, arreglo, etc. Luego, cada espacio de direcciones de programa consiste de una colección de segmentos, que generalmente reflejan la división lógica del programa. Este sistema de gestión de memoria es utilizado en sistemas operativos avanzados.

Objetivos alcanzados

- Modularidad de programas: Cada rutina del programa puede ser un bloque sujeto a cambios y recopilaciones, sin afectar por ello al resto del programa.
- Estructuras de datos de largo variable: Cada estructura tiene su propio tamaño y este puede variar.
- Protección: Se puede proteger los módulos del segmento contra accesos no autorizados.
- Compartición: Dos o más procesos pueden ser un mismo segmento, bajo reglas de protección, aunque no sean propietarios de los mismos.
- Enlace dinámico entre segmentos: Puede evitarse realizar todo el proceso de enlace antes de comenzar a ejecutar un programa. Los enlaces se establecerán solo cuando sea necesario.

Ventajas

- El programador puede conocer las unidades lógicas de su programa, dándoles un tratamiento particular.
- Es posible compilar módulos separados como segmentos, el enlace entre los segmentos se realiza hasta que se haga una referencia entre segmentos.
- Debido a que es posible separar los módulos se hace más fácil la modificación de los mismos. Cambios dentro de un módulo no afecta al resto de los módulos.
- Es fácil compartir segmentos.
- Es posible que los segmentos crezcan dinámicamente según las necesidades del programa en ejecución.
- Existe la posibilidad de definir segmentos que aún no existan. Así no se asignará memoria, sino a partir del momento que sea necesario hacer usos del segmento.

Desventajas

- Hay un incremento en los costos de hardware y de software para llevar a cabo la implantación,

así como un mayor consumo de recursos: memoria, tiempo de CPU, etc.

- Debido a que los segmentos tienen un tamaño variable se pueden presentar problemas de fragmentación externas, lo que puede conseguir un plan de reubicación de segmentos en memoria principal.
- Se complica el manejo de memoria virtual, ya que los discos almacenan la información en bloques de tamaños fijos, mientras los segmentos son de tamaño variable. Esto hace necesaria la existencia de mecanismos más costosos que los existentes para paginación.
- Al permitir que los segmentos varíen de tamaño, puede ser necesarios planes de reubicación a nivel de los discos, si los segmentos son devueltos a dicho dispositivo, lo que conlleva a nuevos costos.
- No se puede garantizar, que, al salir un segmento de la memoria, este pueda ser traído fácilmente de nuevo, ya que será necesario encontrar nuevamente un área de memoria libre ajustada a su tamaño.

Como consecuencia del empleo de segmentos de distinto tamaño, la segmentación resulta similar a la partición dinámica. En ausencia de un esquema de superposición o del uso de memoria virtual, sería necesario cargar en memoria todos los segmentos de un programa para su ejecución. La diferencia, en comparación con la partición dinámica, radica en que, con segmentación, un programa puede ocupar más de una partición y éstas no tienen por qué estar contiguas. La segmentación elimina la fragmentación interna, pero, como la partición dinámica, sufre de fragmentación externa. Sin embargo, debido a que los procesos se dividen en un conjunto de partes más pequeñas, la fragmentación externa será menor. Mientras que la paginación es transparente al programador, la segmentación es generalmente visible y se proporciona como una comodidad para la organización de los programas y datos. Normalmente, el programador o el compilador asignan los programas y los datos a diferentes segmentos. En la programación modular, el programa o los datos pueden ser divididos de nuevo en diferentes segmentos. El principal inconveniente de este servicio es que el programador debe ser consciente de la limitación de tamaño máximo de los segmentos.

Estrategias de asignación

El SO tiene que llevar contabilidad de la memoria asignada, mientras que la segmentación supone llevar una lista de zonas ocupadas y espacios libres. A la hora de asignar un segmento en memoria puede haber varios huecos disponibles donde dicho segmento puede colocarse.
De las estrategias empleadas por el SO se consideran las siguientes:

1. *Primer ajuste*: El SO asigna el primer bloque de memoria libre con espacio suficiente para satisfacer la información. La búsqueda de este bloque es de manera secuencial.
2. *Mejor ajuste*: El SO busca el bloque de memoria que represente el menor desperdicio según el requerimiento.
3. *Peor ajuste*: El SO asigna el bloque más grande que encuentre.

El proceso considera su espacio de direcciones compuesto de entidades de tamaño variable llamadas segmentos. La dirección lógica se compone de segmento y desplazamiento. Con el número de segmento se obtiene una entrada en una tabla de segmentos, en donde hay una dirección base y un límite. Si el desplazamiento es mayor que el límite se produce un error de direccionamiento. La dirección de memoria física a la que se accede se obtiene sumando el desplazamiento a la base.

Fragmentación en los sistemas con segmentación

Dado que los segmentos son de distinto tamaño, a medida que se asignan y desasignan segmentos van quedando huecos: es posible que un segmento no pueda colocarse en memoria pues los huecos no están contiguos lo cual se soluciona mediante una compactación. El tamaño de segmento suele ser múltiplo de alguna cantidad de memoria. Por ejemplo, si el tamaño de segmento es múltiplo de 16 bytes hay algo de fragmentación interna.

Espacio de direcciones

El espacio de direcciones es una abstracción de memoria que fue diseñada para permitir que se ejecuten varias aplicaciones concurrentemente sin que interfieran entre sí, es decir que crea un tipo de memoria abstracta para que los programas habiten allí. En otras palabras, un espacio de memoria es el conjunto de direcciones que usa un proceso para direccionar la memoria. Generalmente cada proceso tiene su propio espacio de direcciones [1] tal como se muestra en la Figura 5.

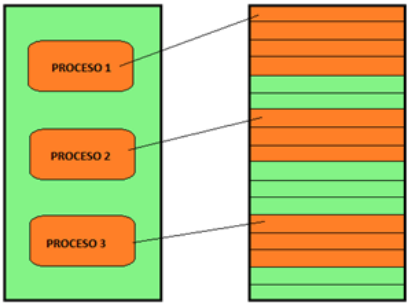

Figura 5. Espacio de direcciones

No es necesario que los espacios de direcciones sean numéricos, ya que pueden implementarse combinando letras, números y guiones cortos, siempre y cuando tengan de 2 a 63 caracteres [1].

Registros base y límite

Cuando se utilizan estos registros, según la Figura 6, los programas se ubican en donde haya espacio de memoria sin reubicarse. Cuando un proceso se ejecuta, el registro base se carga con la dirección física donde inicia el programa y el registro límite con la longitud del programa [1]. Cuando un proceso requiere cargar o almacenar datos, la memoria comprueba si está en el intervalo de registros, caso contrario emitirá un fallo [11].

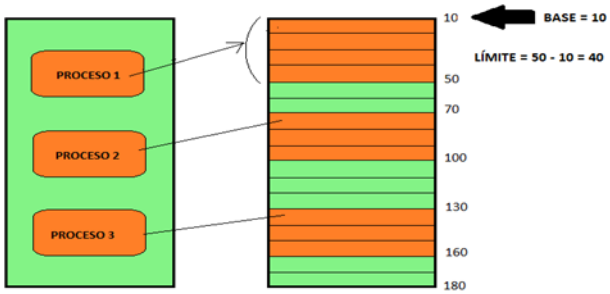

Figura 6. Registros base y límite

Reubicación

No se puede conocer de forma anticipada donde se colocará un programa, y se debe permitir moverlo en la memoria. Luego de que un programa se lleva al disco, podría ser necesario reubicar el proceso a un espacio de memoria diferente de donde se encontraba anteriormente [2].
Existen dos tipos de reubicación [11]:

Estática: Consiste en la carga de un proceso por medio de direcciones absolutas. El programa no se puede reubicar, debido a que tiene particionado fijo, es decir que la memoria se divide en particiones de igual tamaño.
Dinámica: Consiste en la carga de un proceso por medio de direcciones referentes al registro base. Los programas pueden reubicarse luego de iniciar su ejecución.

Protección

Protege los espacios de memoria a los cuales los procesos no deben acceder porque no les pertenecen [12]. Los programas de otros procesos deben tener permiso para acceder a los espacios de memoria de otros procesos, ya sea en modo lectura o escritura. Por otra parte, cumplir con los requisitos de reubicación genera un gran problema para cumplir con los requisitos de protección, aunque existen algunos mecanismos que favorecen a ambos requisitos [11]. El procesador es el encargado de abortar los accesos que no están permitidos durante la ejecución, es decir las referencias de memoria que no tienen permiso [11]. Debido a que el SO no es capaz de saber anticipadamente las referencias de memoria que realizará cada programa, y aunque esto sea posible, saberlo implicaría una gran pérdida de tiempo, los requisitos de protección de memoria son satisfechos por el hardware, mas no por el software [2].

Intercambio

Generalmente en los sistemas operativos Windows y Linux permite iniciar entre 40 y 60 procesos cada vez que se enciende el computador. Debido a que muchas veces el total de RAM que requieren todos los procesos es mayor a lo que puede acomodarse en memoria, es que la estrategia de intercambio fue diseñada para lidiar con la sobrecarga de memoria, que consiste en llevar cada proceso completo a memoria, ejecutarlo durante cierto tiempo y después regresarlo al disco. Los procesos inactivos no ocupan memoria ya que estarán almacenados en el disco [1].

En otras palabras, el intercambio consiste en intercambiar los procesos entre la memoria y el almacenamiento secundario, para esto se debe emplear dispositivos auxiliares rápidos, de lo contrario esto ralentizaría el proceso [1].

En la Figura 7 se describe un ejemplo de intercambio de memoria entre procesos. El proceso 1 está en memoria. Se crea o se intercambia del disco el proceso 2 y 3. El proceso 1 pasa al disco. Se crea o se intercambia del disco el proceso 4 en el primer espacio de direcciones disponible, con el tamaño necesario para dicho proceso. El proceso 3 pasa al disco. El proceso 1 regresa a la

memoria y ocupa un espacio de direcciones distinto a donde se encontraba anteriormente, debido a que ese espacio de direcciones ya fue ocupado por otro proceso. El proceso 3 regresa a la memoria y ocupa el primer espacio de direcciones disponibles según el tamaño que necesite.

Figura 7. Ejemplo de intercambio de memoria

Compartición

Cualquier mecanismo de protección debe permitir que un proceso de un programa tenga acceso a otro programa, esto se aplica a programas que cooperan entre sí para llevar a cabo una tarea, debido a esto necesitarían compartir el acceso a la misma estructura de datos, pero este acceso debe ser controlado para no comprometer la protección fundamental [2].

Administración de memoria libre

Cuando la memoria se asigna dinámicamente, el SO debe administrarla, y para ello se utilizan mapas de bits y listas libres [1].

Mapa de bits

Con un mapa de bits, la memoria se divide en unidades de asignación, desde el punto de vista hace referencia a un conjunto de palabras como una asignación pequeña y a un conjunto de KB como una asignación grande. En el mapa de bits existe un bit para cada unidad de asignación. Generalmente se utiliza 0 (cero) cuando la unidad está desocupada y 1 (uno) si la unidad está ocupada, aunque en algunos casos lo utilizan de forma contraria [1]. Según se muestra en la Figura 8, si la unidad de asignación es pequeña el mapa de bits será más grande, pero si la unidad de asignación es grande el mapa de bits será más pequeño, aunque en el segundo caso se podría desperdiciar una gran cantidad de memoria en la última unidad del proceso si su tamaño no es múltiplo exacto de la unidad de asignación [1]. Un mapa de bits es una manera sencilla de llevar el registro de las palabras de memoria en una cantidad determinada de memoria, debido a que éste solo depende de la memoria y de la unidad de asignación. La búsqueda en un mapa de bits es su principal desventaja ya que este proceso es lento, debido a que la sucesión puede traspasar los límites entre las palabras en el mapa [1].

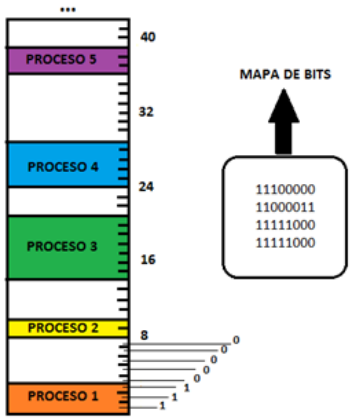

Figura 8. Registro de mapa de bits

Listas Libres

Las listas libres o ligadas son otro método para llevar el registro de la memoria, en donde un segmento contiene un proceso o hueco vacío entre dos procesos [1] tal como se muestra en la Figura 9.

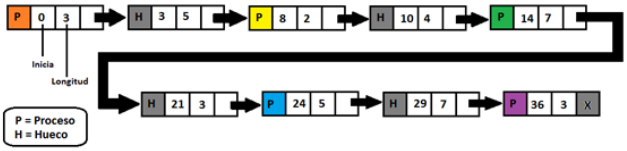

Figura 9. Lista libre

Conclusiones

La gestión de memoria es una de las principales actividades del sistema operativo, la cual, mediante el uso de algoritmos y operaciones, se asigna la memoria a un proceso.

Además de optimizar el uso de los recursos y asignarlo a los procesos, cuida del hardware, evitando calentamiento, o posibles daños.

Cada proceso llevado a cabo en la gestión, está pensado para satisfacer una necesidad, y optimizar el uso de los recursos, y se realizan mediante operaciones, o aplicando algoritmos, que servirán para seguir una línea en la repartición de los elementos de memoria.

Como dato interesante, ciertos algoritmos son aplicados en muchos campos, como lo vimos en los algoritmos de planificación, y ahora en los algoritmos de paginación.

Referencias

[1] A. S. Tanenbaum, Sistemas operativos modernos. Pearson Educación, 2003.

[2] W. Stallings, Sistemas operativos. Prentice Hall, 2004.

[3] D. González, "Diseño de sistemas operativos modernos". Tesis, Universidad de León, España, 2005.

[4] G. Suley, Algoritmos de Paginación, [Online] Disponible: http://geovanasanchezmolina.blogspot.com/2010/11/algoritmosde-paginacion.html, 2010.

[5] Sistemas Operativos Memoria Virtual, [Online] Disponible: http://www.dc.fi.udc.es/so-grado/SO-MemoriaVirtual.pdf, 2012.

[6] CARACTERÍSTICAS Y ALGORITMOS DE PAGINACIÓN, [Online] Disponible: https://chsosunal20161912551.wordpress.com/2016/05/12/caracteristicasy-algoritmos-de-paginacion/, 2016.

[7] V. Hernandez, Algoritmos de sustitución de páginas, [Online] Disponible: http://vicentehdez.blogspot.com/2010/11/44-algoritmos-desustitucion-de-paginas.html, 2010.

[8] Grupozota Fundamentos de SO Version 3, [Online] Disponible: https://es.scribd.com/doc/43779993/Fundamentos-de-So-Version-3, (2010).

[9] G. Wolf, Fundamentos de sistemas operativos. Lulu.com, 2015.

[10] A. Silberschatz; G. Gagne; P. B. Galvin, Fundamentos de sistemas operativos. McGraw-Hill, 2006.

[11] J. N. Camazón, Sistemas operativos monopuesto. Editex, 2011.

[12] B. Sánchez Raya. SISTEMAS OPERATIVOS I. [Online] Disponible: https://www.yumpu.com/pt/document/read/32295794/sistemas-operativos-i-mc-benito-sanchez-raya. 1991.