juanfrandm98
www.wuolah.com/student/juanfrandm98

Tema-3.pdf

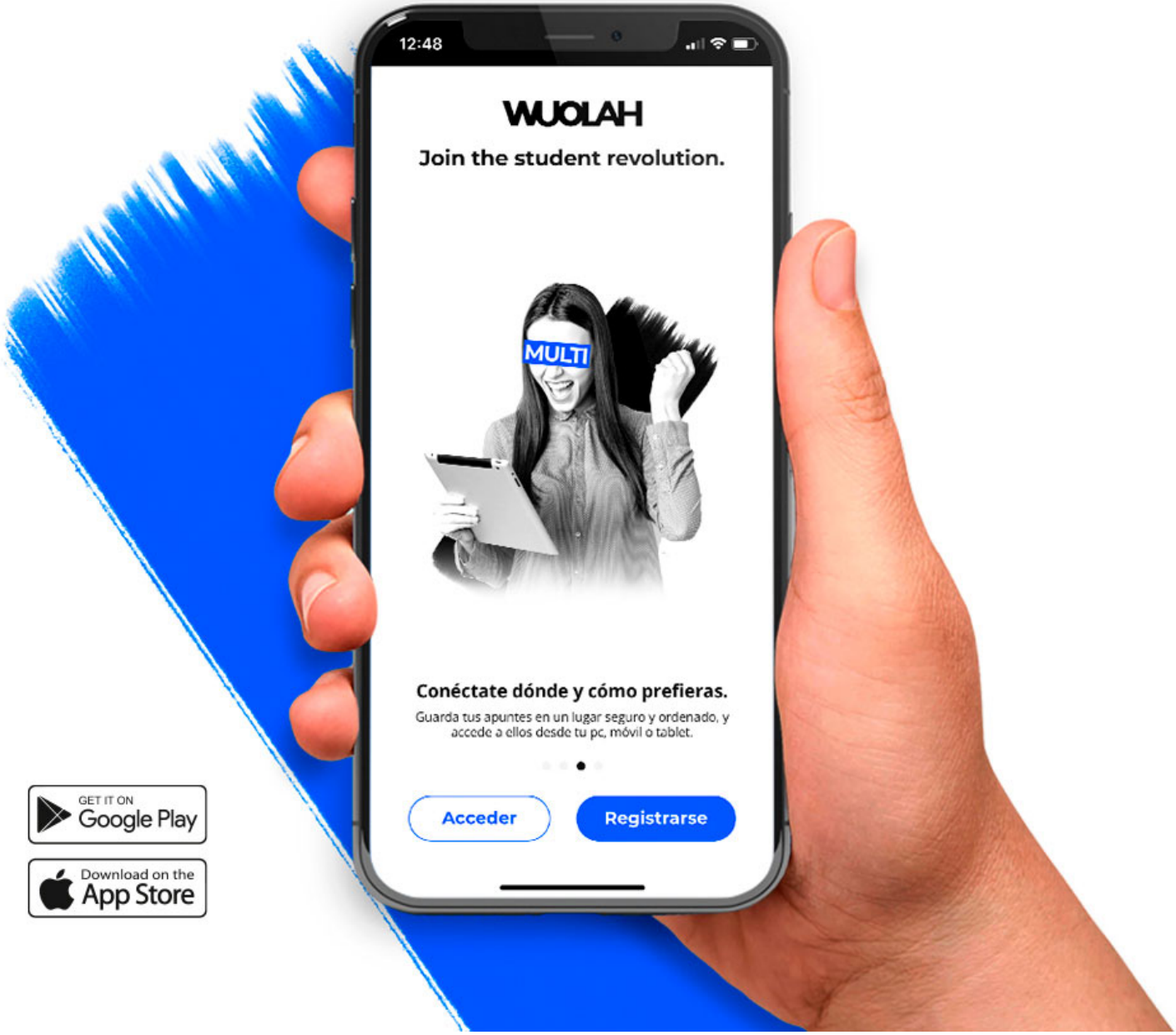
Apuntes de los Libros

2° Sistemas Operativos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Exámenes, preguntas, apuntes.



GET IT ON Google Play

Download on the App Store

UNIVERSIDAD DE GRANADA

Sistemas Operativos

Tema 3. Gestión de Memoria

Juan Francisco Díaz Moreno
Curso 20/21

La Jerarquía de memoria

El diseño de la memoria de un computador debe basarse en tres aspectos: capacidad, velocidad y coste.

En cuanto a la velocidad, para alcanzar un rendimiento máximo, la memoria debe ser capaz de mantener el ritmo del procesador para que según este ejecuta instrucciones, no haya pausas esperando la disponibilidad de instrucciones y operandos.

Para cualquier memoria se cumplen las siguientes relaciones:

- Cuanto menor tiempo de acceso, mayor coste por bit.
- Cuanta mayor capacidad, menor coste por bit.
- Cuanta mayor capacidad, menor velocidad de acceso.

El empleo de una jerarquía de memoria soluciona este dilema. Según se desciende en la jerarquía, disminuye el coste por bit y la frecuencia de acceso a la memoria y aumenta la capacidad y el tiempo de acceso.

Por ejemplo, si el procesador tiene acceso a dos niveles de memoria tales que:

- El nivel 1 contiene 1.000 B y tiene un tiempo de acceso de 0.1 μs. El acceso a los Bytes por el procesador es directo.
- El nivel 2 contiene 100.000 B y tiene un tiempo de acceso de 1 μs. El acceso a los Bytes requiere la transferencia primero al nivel 1.

La tasa de aciertos, A, es la fracción de todos los accesos a memoria que se encuentran en la memoria más rápida, en este caso, el nivel 1. Si la palabra accedida se encuentra en la memoria más rápida, se define como un acierto; mientras que si no lo está, se define como fallo.

Suponiendo que el 95% de los accesos a memoria se encuentran en la caché (A = 0.95), el tiempo medio para acceder a un Byte se expresa:

$$(0.95)(0.1\ \mu s) + (0.05)(0,1\ \mu s + 1\ \mu s) = 0.095 + 0.055 = 0.15\ \mu s$$

Para que esto se cumpla, se tienen que garantizar los cuatro principios de las jerarquías de memoria. El cuarto, la disminución de la frecuencia de acceso a la memoria por parte del procesador, se basa en el principio de la *proximidad de referencias*. Durante la ejecución de un programa, las referencias de memoria del procesador (instrucciones y datos) tienden a agruparse. Cuando se inicia un bucle o una subrutina, hay referencias repetidas a un conjunto de instrucciones. Del mismo modo, las operaciones con tablas y vectores involucran accesos a conjuntos agrupados de Bytes. En un periodo de tiempo largo, las agrupaciones en uso van cambiando; pero en uno corto, el procesador puede estar trabajando con grupos fijos de referencias a memoria.

Por ello, es posible organizar los datos a través de la jerarquía de manera que el porcentaje de accesos a cada nivel sucesivamente más bajo es considerablemente menor que al nivel inferior.

En el ejemplo anterior, si los datos e instrucciones de un programa están en el nivel 2, pueden llevarse temporalmente al 1 e ir intercambiándose según la necesidad.

Usualmente, el tipo de memoria más rápida, pequeña y costosa son los registros del procesador. Descendiendo dos niveles está la memoria principal. Cada posición de memoria principal tiene una única dirección, a las cuales hacen referencia las instrucciones máquina. Esta memoria se amplía usualmente con una caché, más pequeña y de mayor velocidad, que se trata de un dispositivo que controla el movimiento de datos entre la memoria principal y los registros del procesador para mejorar el rendimiento.

La memoria no volátil externa se denomina memoria secundaria o auxiliar. Se usa para almacenar los ficheros de programas y datos, siendo usualmente visibles al programador en términos de ficheros y registros.

Memoria Caché

Aunque es invisible para el SO, interactúa con otros elementos del hardware de gestión de memoria.

Motivación

En cada ciclo de instrucción, el procesador accede a memoria al menos una vez para leer la instrucción y, con frecuencia, una o más veces adicionales para leer y/o almacenar resultados. Por ello, la velocidad del procesador está limitada por el tiempo de ciclo de memoria (tiempo que se tarda en leer o escribir una palabra de la memoria).

La solución consiste en aprovecharse del principio de proximidad utilizando una memoria pequeña y rápida entre el procesador y la memoria principal, denominada caché.

Fundamentos de la Caché

Conceptualmente, hay una memoria principal relativamente grande y lenta junto con una memoria caché más pequeña y rápida. La caché guarda una copia de una parte de dicha memoria.

Cuando el procesador intenta leer un Byte de la memoria, comprueba si está en la caché. Si es así, se entrega al procesador, si no, se introduce dentro de la caché un bloque de memoria principal que contenga el Byte deseado y, posteriormente, se entrega dicho Byte.

Por el principio de la proximidad, al introducir un nuevo bloque en caché, es probable que muchas de las referencias a memoria en el futuro correspondan a Bytes de dicho bloque.

Suponiendo que la memoria principal consta de hasta 2^n palabras y consta de bloques de longitud fija de K palabras, hay $M = 2^n/K$ bloques. La caché consiste en C huecos (líneas) de K palabras cada uno, siendo el número de huecos mucho menor que el de bloques de la memoria principal.

Si se lee una palabra de un bloque de memoria que no está en caché, se transfiere ese bloque a uno de los huecos de la caché. Cada hueco incluye una etiqueta que identifica al bloque que almacena. Por ejemplo, con una dirección de 6 bits y una etiqueta de 2 bits, la etiqueta 01 se refiere al bloque con las direcciones 010000, 010001, 010010, 010011, 010100, ... , 011111.

Diseño de la Caché

Elementos fundamentales:

- Tamaño de la caché. Si es lo suficientemente pequeño, puede tener un impacto significativo en el rendimiento.
- Tamaño del bloque: unidad de datos que se intercambia entre la caché y la memoria principal. Según se incrementa, aumenta la tasa de aciertos; pero cuando es excesivamente grande, disminuye, ya que la probabilidad de volver a usar los datos recientemente leídos se hace menor que la de utilizar los que se van a expulsar para dejar sitio al nuevo bloque.
- Función de correspondencia: determina qué posición de la caché ocupará un bloque que se introduce en ella. Tiene dos restricciones:
    - Cuando se introduce un bloque y se debe reemplazar otro, habría que minimizar la probabilidad de que se reemplazase un bloque que se necesitara en un futuro inmediato. Cuánto más flexible es la función de correspondencia, mayor grado de libertad a la hora de diseñar un algoritmo de reemplazo que maximice la tasa de aciertos.
    - Cuanto más flexible es la función de correspondencia, más compleja es la circuitería requerida para buscar en la caché y determinar si un bloque dado está allí.
- Algoritmo de reemplazo: selecciona, dentro de las restricciones de la función de correspondencia, qué bloque reemplazar cuando uno nuevo va a cargarse y no quedan huecos libres. Sería deseable reemplazar un bloque que no se vaya a necesitar en un futuro inmediato. Una estrategia eficiente es reemplazar el que ha estado en caché más tiempo sin haber sido usado (algoritmo del *menos recientemente usado*, LRU). Para identificarlo, se necesitan mecanismos hardware.
- Política de escritura: determina cuándo tiene lugar la operación de escritura en memoria cuando se altera el contenido de un bloque en la caché. Puede hacerse cada vez que se actualiza o cuando se reemplaza. Esta última minimiza las operaciones de escritura pero deja la memoria principal temporalmente en estado obsoleto, lo que puede afectar en un entorno multiprocesador.

Modelo de Memoria de un Proceso

El SO gestiona el mapa de memoria de un proceso durante su vida, estando inicialmente vinculado al programa ejecutable asociado.

Fases en la generación de un ejecutable

1. Compilación. Se genera un módulo objeto por cada archivo fuente con el código máquina correspondiente, asignando direcciones a los símbolos definidos y resolviendo las referencias. Por ejemplo, si a una variable se le asigna una posición de memoria, todas las instrucciones que hagan referencia a dicha variable deben especificar dicha dirección.
2. Montaje o enlace. Se genera un ejecutable agrupando los archivos objeto y resolviendo las referencias entre módulos. Debe incluirse también objetos extraídos de bibliotecas, y resolver también las direcciones correspondientes.

Bibliotecas de objetos

Una biblioteca es una colección de objetos relacionados. Los sistemas tienen bibliotecas predefinidas que proporcionan servicios a las aplicaciones.

Las bibliotecas pueden ser creadas por los usuarios para modularizar sus aplicaciones y facilitar que estas compartan módulos.

Bibliotecas dinámicas

Hasta ahora hemos visto un sistema que genera ejecutables autocontenidos, con fases de compilación y enlace de objetos. Pero esto tiene desventajas:

- El archivo ejecutable puede ser muy grande, pues incluye su código y el de todas las funciones externas que utilice.
- Cualquier programa que utilice una función de una biblioteca tendrá una copia de la misma.
- Si estos programas se ejecutan a la vez, existirán en memoria múltiples copias de dicha función, aumentando el gasto de memoria.
- La actualización de una biblioteca implica la regeneración de todos los ejecutables que la incluyen.

Para resolver estas deficiencias, se utilizan bibliotecas dinámicas, denominándose estáticas las vistas hasta ahora.

En este caso, el montaje de las bibliotecas se realiza en tiempo de ejecución. El código del ejecutable no incluirá el de la biblioteca, sino que anota el nombre de la misma y se enlaza en tiempo de ejecución. Para ello, el ejecutable incluirá un módulo de montaje dinámico que cargará la biblioteca cuando se haga referencia por primera vez.

El proceso de resolución de referencias afecta al programa que utiliza las bibliotecas, ya que modifica en tiempo de ejecución las instrucciones para que apunten a la dirección real del símbolo.

El uso de las bibliotecas dinámicas resuelve las deficiencias de las estáticas:

- El tamaño de los ejecutables disminuye ya que no contienen el código de las bibliotecas dinámicas.
- Las rutinas de una biblioteca dinámica estarán almacenadas únicamente en el archivo de la biblioteca.
- Varios procesos que utilicen a la vez la misma biblioteca dinámica compartirán su código.
- La actualización de una biblioteca dinámica no requiere volver a montar los ejecutables. Cabe destacar que si el cambio se produce en la interfaz, el programa utilizará la versión antigua de las bibliotecas. Éstas contendrán para ello su número de versión para que, en tiempo de ejecución, se cargue la biblioteca cuya versión corresponda con la del programa.

La compartición del código de las bibliotecas dinámicas supone un problema de referencias, pues puede tener asociado un rango de direcciones diferente en cada proceso. Existen tres alternativas:

- Establecer un rango de direcciones predeterminado y específico para cada biblioteca dinámica, de manera que todos los procesos incluirán dicho rango en su mapa de memoria. Esto limita el número de bibliotecas que pueden existir en el sistema y puede causar que el mapa de un proceso sea muy grande con amplias zonas vacías.
- Reubicar las referencias presentes en el código de la biblioteca durante su carga para que se ajusten a las direcciones establecidas en el mapa de memoria del proceso que la usa. Esto elimina el exceso de espacio, pero impide la compartición completa del código al estar adaptado a la zona de memoria donde le ha tocado vivir.
- Generar el código de la biblioteca de manera que sea independiente de la posición (PIC, *Position Independent Code*). El compilador generará el código con direccionamientos relativos a un registro de manera que no se ve afectado por la posición de memoria donde ejecuta, permitiendo la compartición del código a costa de una pequeña disminución en la eficiencia del código.

La desventaja del uso de bibliotecas dinámicas es que el tiempo de ejecución de un programa puede aumentar debido al tiempo de carga de las bibliotecas.

La mayoría de los sistemas que utilizan bibliotecas dinámicas tienen una versión estática y otra dinámica de cada una. Por defecto, el montador usará la versión dinámica, aunque si el usuario hubiera pedido explícitamente la utilización de la estática, también sería posible.

Montaje explícito de bibliotecas dinámicas

La forma de usar las bibliotecas dinámicas más habitual es el enlace dinámico implícito: se especifica en tiempo de montaje qué bibliotecas se deben usar y se pospone la carga hasta el tiempo de ejecución.

Pero, por ejemplo, un navegador que procese distintos formatos de archivos con distintas bibliotecas no sabría en tiempo de montaje qué biblioteca necesitará. Por el método anterior, se tendría que volver a montar el navegador con cada tipo de archivo distinto que utilice.

Esto requiere la utilización del enlace dinámico explícito, por el que se puede decidir en tiempo de ejecución qué biblioteca dinámica se necesita y solicitar su montaje y carga.

Formato del ejecutable

Existen distintos formatos para archivos ejecutables. En UNIX, uno de los más utilizados es el *Executable and Linkable Format* (ELF).

Un ejecutable está estructurado como una cabecera y un conjunto de secciones. La cabecera contiene información de control para interpretar el contenido del ejecutable, incluyendo:

- Número mágico: identifica al ejecutable. En el formato ELF, el primer Byte del archivo debe contener el valor hexadecimal 7f y los caracteres 'E', 'L' y 'F'.
- Dirección del punto de entrada del programa: primera dirección del contador de programa.
- Tabla que describe las secciones que contiene. Cada una especifica el tipo, la dirección donde comienza en el archivo, su tamaño...

El contenido de las secciones es variado. Una puede contener información de depuración, otra la lista de bibliotecas dinámicas que requiere... Las tres que más influyen en el mapa de memoria de un proceso son:

- Código del programa.
- Datos con valor inicial: variables globales que tienen valor inicial.
- Datos sin valor inicial: sólo se especifica el tamaño de la sección, ya que su contenido es irrelevante.

Los ejecutables no cuentan con una sección para variables locales y parámetros de las funciones, ya que tienen características muy diferentes a las variables globales:

- Las variables globales son estáticas, existen durante toda la vida del programa y tienen una dirección fija en el mapa de memoria.
- Las variables locales y parámetros son dinámicas. Se crean cuando se invoca la función correspondiente y se destruyen cuando termina la llamada, por lo que no tienen espacio asignado en el mapa inicial ni en el ejecutable. Se crean usando la pila del proceso y su dirección se determina en tiempo de ejecución, pues depende de la secuencia de llamadas que genere el programa.

Mapa de memoria de un proceso

El mapa de memoria de un proceso está formado por distintos segmentos, cada uno con determinada información asociada. El objeto de memoria es el conjunto de información relacionada. La asociación de una región de un proceso con un objeto de memoria permite al proceso tener acceso a su información.

Cuando se activa la ejecución de un programa, se crean varias regiones dentro del mapa a partir de la información del ejecutable. Cada región constituye un objeto de memoria, siendo las iniciales las distintas secciones del ejecutable.

Cada región es una zona contigua caracterizada por la dirección dentro del mapa de proceso donde comienza y por su tamaño y tiene asociada una serie de propiedades como:

- Soporte de la región: el objeto de memoria asociado a la región. Almacena el contenido inicial, existiendo dos posibilidades:
  - Soporte en archivo: el objeto está almacenado en un archivo o en parte de este.
  - Sin soporte: el objeto no tiene contenido inicial, denominándose objeto anónimo.
- Tipo de uso compartido:
  - Privado: el contenido de la región sólo es accesible al proceso que la contiene y sus modificaciones no se reflejan en el objeto de memoria.
  - Compartido: el contenido puede ser compartido por varios procesos y las modificaciones sí se reflejan.
- Protección: tipo de acceso permitido. Hay tres tipos:
  - Lectura: accesos de lectura de operandos de instrucciones.
  - Ejecución: accesos de lectura de instrucciones (*fetch*).
  - Escritura: accesos de escritura.
- Tamaño fijo o variable: en el caso de regiones de tamaño variable, se suele distinguir si la región crece hacia direcciones de memoria menores o mayores.

Las regiones que presenta el mapa de memoria inicial del proceso se corresponden con las secciones del ejecutable más la pila inicial del proceso:

- Código: región compartida de lectura/ejecución de tamaño fijo, indicado en la cabecera del ejecutable. El soporte está en la sección correspondiente del ejecutable.
- Datos con valor inicial: región privada, pues cada proceso que ejecuta necesita una copia de las variables del mismo. Es de lectura/escritura y de tamaño fijo, indicado en la cabecera. El soporte está en la sección correspondiente del ejecutable.
- Datos sin valor inicial: región privada de lectura/escritura de tamaño fijo, indicado en la cabecera. No tiene soporte, ya que su contenido inicial es irrelevante. En muchos sistemas se le da un valor inicial a cero por motivos de confidencialidad.
- Pila: región privada y de lectura/escritura. Almacena los registros de activación de las llamadas a funciones (variables locales, parámetros, dirección de retorno...). Es de tamaño variable que crece hacia direcciones más bajas cuando se producen llamadas y decrece

cuando se retornan. En el mapa inicial existe ya esta región, que contiene los argumentos especificados en la invocación del programa.

Los SO modernos ofrecen un modelo de memoria dinámico en el que el mapa de un proceso está formado por un número variable de regiones que pueden añadirse o eliminarse durante la ejecución del mismo. Además de las regiones iniciales ya analizadas, durante la ejecución del proceso pueden crearse nuevas regiones como:

- Heap: soporte para la memoria dinámica que reserva un programa en tiempo de ejecución. Comienza después de la región de datos sin valor inicial. Es una región privada de lectura/escritura sin soporte, que crece hacia direcciones crecientes según el programa va reservando memoria dinámica y decrece según la vaya liberando.
- Archivos proyectados: región compartida cuyo soporte es el archivo que proyecta.
- Memoria compartida: región compartida cuya protección la especifica el programa a la hora de proyectarla.
- Pila de *threads*: cada *thread* necesita una pila propia que corresponde a una nueva región en el mapa con las mismas características que la pila del proceso.

La carga de una biblioteca dinámica implica la creación de un conjunto de regiones asociadas a la misma que contendrán las distintas secciones de la biblioteca (código y datos).

Dado el carácter dinámico del mapa de memoria de un proceso, existirán en determinados instantes zonas sin asignar (huecos) dentro del mapa. Un acceso a los huecos representa un error y debería ser detectado y tratado por el SO.

Dado que el SO es un programa, su mapa de memoria contendrá también regiones de código, datos y heap, ya que usa también memoria dinámica.

Operaciones sobre regiones

Durante la vida de un proceso, su mapa de memoria va evolucionando. Existen distintas operaciones que se pueden realizar sobre una región del mapa:

- Crear una región vinculada al objeto en el lugar correspondiente, asignando los recursos necesarios y estableciendo las características y propiedades de la misma.
- Eliminar una región del mapa de un proceso, liberando todos los recursos vinculados. Cuando un proceso termina, se liberan implícitamente todas sus regiones. En el caso de archivos proyectados o memoria compartida, el proceso puede solicitar explícitamente su eliminación del mapa.
- Cambiar el tamaño de una región, ya sea por una petición explícita del programa (como en el caso del *heap*) o de forma implícita (como sucede al expandir la pila). En el caso de un aumento, el sistema asignará recursos comprobando que no se produce solapamiento, en cuyo caso no se llevaría a cabo. En el caso de una reducción, se liberan los recursos.
- Duplicar una región del mapa de un proceso en el mapa de otro. Esta operación crea una nueva región asociada a un objeto de memoria que es una copia del anterior, por lo que las modificaciones de una región no afectarían a la otra.

Memoria virtual

Hardware y Estructuras de Control

Características clave de la paginación y la segmentación:

1. Todas las referencias a la memoria dentro de un proceso se realizan a direcciones lógicas, que se traducen dinámicamente en direcciones físicas durante la ejecución. Esto se traduce a que un proceso puede ser llevado y traído a memoria, pudiendo ocupar diferentes regiones de la memoria principal cada vez.
2. Un proceso puede dividirse en porciones (páginas o segmentos) que no tienen que estar localizadas en memoria de forma contigua gracias a las tablas de páginas o segmentos y al uso de la traducción de direcciones dinámicas en ejecución.

Estas características permiten que no todas las páginas o segmentos (porciones) de un proceso estén en memoria principal durante la ejecución. Para que una instrucción pueda ejecutarse, basta con que la página donde se encuentra dicha instrucción y la de los datos que necesite estén en memoria.

Cuando un nuevo proceso se tiene que traer a memoria, el SO trae únicamente una o dos porciones, incluyendo la porción inicial del programa y la porción inicial de datos sobre la que acceden las primeras instrucciones. La parte del proceso que se encuentra en un instante dado en memoria principal se denomina conjunto residente.

A través de una tabla de segmentos o páginas, el procesador puede comprobar si las referencias de memoria se encuentran en el conjunto residente. Si encuentra una dirección lógica que no lo está, genera una interrupción indicando fallo de acceso a memoria. El SO coloca al proceso en estado Bloqueado y toma el control. Para que el proceso pueda retomarse, el SO necesita traer a memoria principal la porción del proceso que contiene la dirección lógica que ha causado el fallo, para lo que realiza una petición de E/S, una lectura a disco. Mientras se produce, el SO puede activar otro proceso. Cuando se completa, se lanza una nueva interrupción de E/S, que provoca que el SO coloque al proceso afectado en estado Listo.

Para justificar que esto mejora la eficiencia, existen dos implicaciones:

1. Pueden mantenerse un mayor número de procesos en memoria principal, dado que al mantenerse sólo porciones de procesos, hay más espacio. Esto aumenta la eficiencia del procesador al aumentar la probabilidad de que alguno de los procesos esté en estado Listo.
2. Un proceso puede ser mayor que toda la memoria principal. Sin memoria virtual, un programador debe estructurar los programas grandes en fragmentos que puedan cargarse de forma separada con un tipo de estrategia de superposición (*overlay*). Pero con memoria

virtual basada en paginación o segmentación, este trabajo se delega al SO y al hardware, que cargan las porciones a memoria principal cuando se necesitan.

Como los procesos se ejecutan sólo en memoria principal, se le denomina memoria real; y la memoria que se encuentra en disco, memoria virtual. Esta última permite una multiprogramación muy efectiva que libera al usuario de restricciones de la memoria principal.

Proximidad y Memoria Virtual

La memoria virtual basada en paginación o paginación con segmentación se ha convertido, demostrando su eficiencia, en una componente esencial de todos los SO actuales.

Un proceso puede ser de gran tamaño y contener un gran número de vectores de datos, aunque en periodos cortos de tiempo, la ejecución se acote a una pequeña sección y alguno de los vectores de datos. Por ello, cargar todo el programa en memoria principal supondría un desperdicio.

Para conseguir un mejor uso de la memoria, se pueden cargar únicamente unas pocas porciones. Si el programa hace referencia a una porción que no esté en memoria, se dispara un fallo que provoca que el SO consiga dicha porción. De este modo, sólo unas pocas porciones de cada proceso se encuentran en memoria, por lo que esta puede alojar más procesos. Además, se ahorra tiempo ya que las porciones de proceso no usadas no se intercambian en el *swapping*.

El SO debe ser capaz de manejar con inteligencia este esquema. En estado estable, casi toda la memoria principal se encontrará ocupada con porciones del mayor número de procesos posible. Por ello, cuando se traiga una porción a memoria deberá expulsarse otra, teniendo que evitar expulsar porciones que vayan a ser utilizadas inmediatamente, lo que generaría trasiego (*thrashing*): el sistema consume mucho tiempo enviando y trayendo porciones de *swap* en lugar de ejecutando instrucciones. Para evitarlo, el SO trata de adivinar, en base a lo ocurrido recientemente, qué porciones son menos probables de ser utilizadas pronto.

Este razonamiento se basa en el principio de proximidad, que afirmaba que las referencias al programa y a los datos tienen a agruparse.

Para que la memoria virtual sea útil, se necesita que exista un soporte hardware para el esquema de paginación y/o segmentación; y que el SO incluya el código para gestionar el movimiento de páginas y/o segmentos entre memoria principal y secundaria.

Paginación

En un sistema de paginación sencilla, cada proceso dispone de su propia tabla de páginas, que se encuentran en memoria principal. Cada entrada de la tabla consiste en un número de marco de la correspondiente página en la memoria principal.

En un sistema con memoria virtual, la tabla de páginas también es necesaria, aunque con entradas más complejas que indican si cada página está presente (P) en memoria principal o no. Si el bit indica que la página está en memoria, la entrada también debe indicar el número de marco de dicha página.

La entrada de la tabla de páginas incluye un bit de modificado (M), que indica si los contenidos de la página han sido alterados desde que se cargó en la memoria principal. Si no hubo ningún cambio, no es necesario escribir la página cuando se reemplace por otra en el marco de página que actualmente ocupa.

Estructura de la tabla de páginas.

La lectura de una palabra de la memoria implica la traducción de la dirección lógica, que consiste en un número de página y un desplazamiento, a la dirección física, consistente en un número de marco y un desplazamiento, usando para ello la tabla de páginas.

La tabla de páginas es de longitud variable dependiendo del tamaño del proceso, por lo que se encuentra en memoria principal. Cuando un proceso está ejecutando, un registro contiene la dirección de comienzo de su tabla de páginas. El número de página de la dirección virtual se utiliza para indexar esa tabla y buscar el correspondiente marco de página que, junto con la parte de desplazamiento de la dirección virtual, genera la dirección real deseada. El campo correspondiente al número de página normalmente es mayor que el campo correspondiente al número de marco de página ($n > m$).

En la mayoría de sistemas existe una única tabla de páginas por proceso, aunque cada uno puede ocupar mucha memoria virtual, generando una tabla de página también muy grande. Para paliar, la mayoría de esquemas de memoria virtual sitúan las tablas en memoria virtual, peinándose también.

Algunos procesadores utilizan un esquema de dos niveles para organizar las tablas de páginas grandes, con un directorio de páginas en el que cada entrada apunta a una tabla de páginas. Normalmente, la longitud máxima de una tabla de páginas se restringe para que sea igual a una página.

Tabla de páginas invertida.

La desventaja del tipo de tablas de páginas visto es que su tamaño es proporcional al espacio de direcciones virtuales.

Una estrategia alternativa al uso de varios niveles es la estructura de tabla de páginas invertida, en las que la parte correspondiente al número de página de la dirección virtual se referencia por medio de un valor *hash* obtenido con una función *hash* sencilla, que es un puntero para la tabla de páginas invertida que contiene las entradas de tablas de página.

Hay una entrada en la tabla de páginas invertida por cada marco de página real en lugar de uno por cada página virtual, lo que hace que estas tablas siempre requieran una proporción fija de la memoria real. Debido a que más de una dirección virtual puede traducirse en la misma entrada de la tabla *hash*, el desbordamiento se gestiona por encadenamiento.

La estructura de la tabla de páginas se denomina invertida debido a que se indexan sus entradas de la tabla de páginas por el número de marco en lugar de por el número de página virtual.

Para un tamaño de memoria física de 2^m marcos, la tabla de páginas invertida contiene 2^m entradas, de forma que la entrada de la posición i-ésima se refiere al marco i. La entrada en la tabla de páginas incluye la siguiente información:

- Número de página: número de página de la dirección virtual.
- Identificador del proceso: proceso propietario de la página. La combinación de número de página e identificador del proceso identifica a una página dentro del espacio de direcciones virtual de un proceso en particular.
- Bits de control: flags como válido, referenciado, modificado... Información de protección y cerrojos.
- Puntero de la cadena: es nulo si no hay más entradas encadenadas; en otro caso, contiene el valor del índice (número entre 0 y 2^{m-1}) de la siguiente entrada.

Buffer de traducción anticipada.

Toda referencia a memoria virtual puede causar dos accesos a memoria física, uno para buscar la entrada de la tabla de páginas y otro para buscar los datos solicitados, lo que se estaría duplicando el tiempo de acceso a memoria.

Para solucionarlo, la mayoría de esquemas de memoria virtual utilizan una caché de alta velocidad para las entradas de la tabla de páginas, el *buffer* de traducción anticipada (*translation lookaside buffer* - TLB), que contiene las entradas de la tabla de páginas que han sido usadas recientemente.

Así, dada una dirección virtual, el procesador primero examina la TLB. Si la entrada está presente (acierto), se recupera el número de marco y se construye la dirección real. Si se produce un

fallo, el procesador utiliza el número de página para indexar la tabla de páginas del proceso y examinar la correspondiente entrada de la tabla de páginas. Si el bit de presente está puesto a 1, la página solicitada no se encuentra en memoria principal y se produce un fallo de acceso a memoria llamado fallo de página. El SO cargará la página necesaria y actualizada de la tabla de páginas.

Debido a que la TLB sólo contiene algunas páginas de las entradas de toda la tabla de páginas, no es posible indexar simplemente la TLB por medio de número de página. Cada entrada debe incluir además la entrada de la tabla de páginas completa. El procesador proporciona un hardware que permite consultar simultáneamente varias entradas para determinar si hay una conciencia sobre un número de página. Esta técnica se denomina resolución asociativa (*asociative mapping*) que contrasta con la resolución directa o indexación.

El diseño de la TLB debe considerar la forma mediante la cual las entradas se organizan en ella y qué entrada se debe reemplazar cuando se necesite traer una nueva entrada, lo que influye en el diseño de la caché hardware.

El mecanismo de memoria virtual debe interactuar con el sistema de caché de la memoria principal. Una dirección virtual tendrá generalmente el formato número de página, desplazamiento. Primero, el sistema de memoria consulta la TLB para ver si se encuentra presente una entrada de la tabla de página que coincide. Si es así, la dirección real se genera combinando el número de marco con desplazamiento. Si no, la entrada se busca en la tabla de páginas. Una vez se ha generado la dirección real, que mantiene el formato de etiqueta (*tag*) y resto (*remainder*), se consulta la caché para ver si el bloque que contiene esa palabra se encuentra ahí. Si es así, se le devuelve la CPU; si no, la palabra se busca en la memoria principal.

Tamaño de página.

Para seleccionar el tamaño de página a utilizar, hay varios factores a considerar.

Por un lado, cuanto mayor es el tamaño de página, menor cantidad de fragmentación interna, lo que optimiza el uso de la memoria principal. Por otro lado, cuanto menor es la página, mayor número de páginas son necesarias para cada proceso y mayores son las tablas de páginas. En entornos multiprogramados, partes de las tablas de páginas de los procesos activos deben encontrarse en memoria virtual, lo que provocará fallos de página dobles para una referencia sencilla a memoria: traer la tabla de páginas de la parte solicitada y traer la página del propio proceso. Además, la mayoría de los dispositivos de memoria secundaria son de tipo giratorio, favoreciendo tamaños de página grandes para mejorar la eficiencia de transferencia de bloques de datos.

Por el principio de proximidad, si el tamaño de página es muy pequeño, habrá un número relativamente alto de páginas en memoria principal para cada proceso, disminuyendo la tasa de fallos. A medida que el tamaño de páginas se incrementa, la página en particular contendrá información más lejos de la última referencia realizada, debilitando el principio de proximidad y

aumentando la tasa de fallos. En algún momento, la tasa de fallos comenzará a caer a medida que el tamaño de página se aproxima al tamaño del proceso completo.

Segmentación

Las implicaciones en la memoria virtual

La segmentación permite al programador ver la memoria como un conjunto de segmentos de tamaños dinámicos. Una referencia a memoria consiste en un formato de dirección del tipo (número de segmento, desplazamiento).

Esto conlleva a una serie de beneficios frente a espacios no segmentados:

1. Simplifica el tratamiento de estructuras de datos que pueden crecer. Con un sistema sin segmentación, el programador tendrá que estimar el tamaño que puede alcanzar. Con segmentación, se le puede asignar un segmento que el SO podrá expandir o reducir bajo demanda. Si se requiere expandir un segmento y no hay espacio, el SO podrá moverlo a un área de la memoria principal mayor, si se encuentra disponible, o enviarlo a *swap*.
2. Permite programas que se modifican o recopilan de forma independiente, sin requerir que el conjunto de programas se re-enlace y se vuelvan a cargar.
3. Da soporte a la compartición entre procesos, situando algún recurso en un segmento que pueda ser referenciado desde otros procesos.
4. Soporta los mecanismos de protección asignando privilegios de acceso a los segmentos donde se almacenan datos o programas.

Organización

En la segmentación sencilla, cada proceso tiene su tabla de segmentos que se crea y se carga en la memoria principal cuando los segmentos se han cargado en la misma. Cada entrada contiene la dirección de comienzo de un segmento en la memoria principal, así como su longitud.

Cuando existe memoria virtual, la tabla de segmentos también es necesaria. Sigue habiendo una tabla de segmentos por proceso, aunque es más compleja: se necesita un bit para indicar si el segmento está en memoria principal; y si lo está, la entrada debe incluir la dirección de comienzo y su longitud. También posee un bit de modificado, que indica si el contenido del segmento ha sido alterado desde que se cargó en memoria principal.

El mecanismo de lectura de una palabra de memoria implica la traducción de una dirección lógica (número de segmento y desplazamiento) en física utilizando la tabla. Como esta tabla es de longitud variable, debe encontrarse en memoria principal.

Cuando un proceso está en ejecución, un registro mantiene la dirección de comienzo de su tabla de segmentos. El número de segmento de la dirección virtual se utiliza para indexar dicha tabla

y para buscar la dirección de la memoria principal donde comienza dicho segmento. Ésta es añadida a la parte de desplazamiento de la dirección virtual para producir la dirección real solicitada.

Paginación y segmentación combinadas

Paginación y segmentación tienen sus propias ventajas:

- La paginación es transparente al programador y elimina fragmentación externa, proporcionando un uso eficiente de la memoria principal. Gracias a la movilidad de los fragmentos y su tamaño fijo, es posible desarrollar algoritmos de gestión de memoria que exploten el comportamiento de los programas.
- La segmentación es visible al programador y permite manejar estructuras de datos que crecen, modularidad y soporte a la compartición y protección.

En un sistema combinado de paginación/segmentación, el espacio de direcciones de usuario se divide en segmentos. Cada segmento se divide a su vez en páginas de tamaño fijo (el de los marcos de la memoria principal).

Para el programador, una dirección lógica sigue conteniendo un número de segmento y un desplazamiento; pero para el sistema, el desplazamiento corresponde al número de página del segmento, más un desplazamiento dentro de la página.

Asociadas a cada proceso existen una tabla de segmentos y varias tablas de páginas (una por segmento). Cuando un proceso se ejecuta, un registro mantiene la dirección de comienzo de su tabla de segmentos. A partir de la dirección virtual, el procesador utiliza el número de segmento para indexar dentro de la tabla de segmentos y encontrar la tabla de páginas de dicho segmento. La parte correspondiente al número de página de la dirección virtual original se utiliza para indexar la tabla de páginas y buscar el número de marco, que se combina con el desplazamiento para generar la dirección real.

Una entrada de la tabla de segmentos contiene la longitud del segmento y el campo base, que hace referencia a la tabla de páginas. Los bits de presente y modificado son gestionados a nivel de página, aunque los de protección y compartición si se pueden mantener.

La entrada de la tabla de páginas es esencialmente la misma que para un sistema de paginación puro. El número de página se proyecta en su número de marco correspondiente si la página se encuentra presente en la memoria. El bit de modificado indica si la página necesita escribirse cuando se expulse.

Protección y compartición

La segmentación proporciona un mecanismo de protección y compartición. Como cada entrada de una tabla de segmentos incluye la longitud y la dirección base, un programa no puede acceder a una posición fuera del mismo. En cuanto a la compartición, un segmento puede estar referenciado en las tablas de varios procesos

Sin embargo, la estructura de páginas y los datos no son visibles para el programador, haciendo menos cómodos las especificaciones de protección y los requisitos de compartición.

Un esquema habitual más sofisticado consiste en utilizar la estructura de protección en anillo. Los anillos con números bajos (interiores) tienen más privilegios que los de numeración alta (exteriores). El anillo 0 se reserva para funciones del núcleo y algunas de las utilidades del SO pueden ocupar anillos intermedios.

Los principios básicos de los sistemas de anillos son:

- Un programa puede acceder sólo a los datos residentes en el mismo anillo o en anillos con menos privilegios.
- Un programa puede invocar servicios residentes en el mismo anillo o anillos con más privilegios.

Software del SO para la Gestión de Memoria

El diseño de la gestión de la memoria del SO depende de tres opciones a elegir:

- Si el sistema usa o no técnicas de memoria virtual.
- El uso de paginación,segmentación o ambas.
- Los algoritmos utilizados para los diferentes aspectos de la gestión de memoria.

Las elecciones de las dos primeras dependen de la plataforma hardware. Las primeras implementaciones de UNIX no proporcionaban memoria virtual porque los procesadores donde ejecutaban no daban soporte para paginación o segmentación. La plataforma hardware debe poder traducir direcciones y tener otras funciones básicas.

Las elecciones relativas a la tercera opción entran dentro del dominio del software del SO y, en cualquier caso, el aspecto central es el rendimiento: se tratará de minimizar la tasa de ocurrencia de fallos de página que causan sobrecarga. Esto incluye la decisión de qué páginas se van a reemplazar y el *swap* de dichas páginas. El SO deberá también planificar la ejecución de otro proceso durante estas operaciones de E/S. También habrá que intentar minimizar la probabilidad de referenciar palabras de una página que no se encuentre presente en memoria principal

El rendimiento de un conjunto de políticas depende del tamaño de la memoria, la velocidad relativa de la memoria principal y secundaria, del tamaño y el número de procesos que compiten por los recursos, y del comportamiento en ejecución de los distintos programas de manera individual.

Política de Recuperación

La política de recuperación determina cuándo una página se trae a memoria principal. Hay dos alternativas habituales: bajo demanda y paginación adelantada.

Con paginación bajo demanda, una página se trae a memoria sólo cuando se hace referencia a una posición en dicha página. Cuando un proceso arranca inicialmente, va a haber una ráfaga de fallos de página. Según se van trayendo páginas, el principio de proximidad sugiere que futuras referencias se encontrarán en páginas traídas, por lo que tras un tiempo la situación se estabilizará y caerá el número de fallos a un nivel muy bajo.

Con paginación adelantada (*prepaging*), se traen a memoria también otras páginas, además de la que ha causado el fallo de página. Tiene en cuenta que la mayoría de dispositivos de memoria secundaria tienen tiempos de búsqueda y latencia de rotación: si las páginas se encuentran almacenadas de forma contigua, es mucho más eficiente traer a memoria páginas contiguas en lugar de una en una. Esto es ineficiente si la mayoría de las páginas traídas no se referencia a posteriori.

La política de paginación adelantada puede emplearse bien cuando el proceso se arranca (si el programador asigna bien las páginas necesarias) o cada vez que ocurra un fallo de página. Sin embargo, su utilidad no se encuentra reconocida.

La paginación adelantada no debe confundirse con el *swapping*. Cuando un proceso pasa a estado suspendido, todas sus páginas también se expulsan de memoria; y cuando el proceso se recupera, también retornan todas.

Política de Ubicación

La política de ubicación determina en qué parte de la memoria real van a residir las porciones de la memoria de un proceso. En sistemas que usan paginación pura o paginación combinada con segmentación, la ubicación es irrelevante debido a que el hardware de traducción de direcciones y el hardware de acceso a memoria principal pueden realizar sus funciones en cualquier combinación de página-marco con la misma eficiencia.

En un sistema de investigación y desarrollo, la ubicación sí tiene una implicación importante. En los procesadores de acceso a la memoria no uniforme (*nonuniform memory access multiprocessors*, NUMA), la memoria distribuida puede referenciarse por cualquier procesador, pero con un tiempo de acceso dependiente de su localización física. El rendimiento depende de la

distancia a la cual reside el dato en relación al procesador, por lo que una estrategia de ubicación aceptable es la que asigna las páginas al módulo de memoria que finalmente proporcionará mejor rendimiento.

Política de Reemplazo

La política de reemplazo trata de la selección de una página en la memoria principal como candidata para reemplazarse cuando se va a traer una nueva página. Existen una serie de conceptos relacionados que dificultan su explicación:

- ¿Cuántos marcos de página se van a reservar para cada uno de los procesos?
- ¿El conjunto de páginas que se van a considerar para el reemplazo se limita a aquellas del mismo proceso que ha causado el fallo de página, o se consideran todos los marcos de página de la memoria principal?
- Entre el conjunto de páginas a considerar, ¿qué página es la que se reemplaza?

Las dos primeras cuestiones se refieren a la gestión del conjunto residente, mientras que la política de reemplazo se refiere a la tercera.

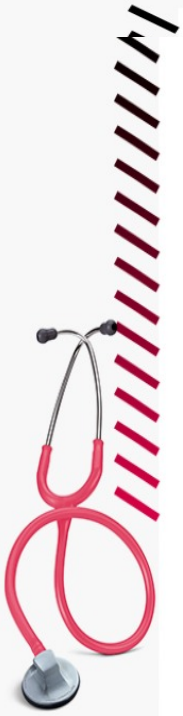
Cuando todos los marcos de la memoria principal están ocupados y es necesario traer una nueva página, la política de reemplazo determina qué página de las que actualmente están en memoria van a reemplazarse. Ésta debe ser la que menos probabilidades tenga de volver a ser referenciada. Por el principio de proximidad, existe a menudo una alta correlación entre el histórico de referencias recientes y los patrones de referencia futuros, por lo que la mayoría de las políticas tratan de predecir el comportamiento en base al que se ha estado produciendo. Sin embargo, cuanto más complicada sea la política de reemplazo, mayor sobrecarga supondrá al implementarla.

Bloqueo de marcos

Las políticas de reemplazo tienen una restricción: algunos marcos de la memoria principal pueden encontrarse bloqueados. Cuando un marco está bloqueado, la página almacenada en él no puede reemplazarse.

Gran parte del núcleo del SO se almacena en marcos bloqueados, así como otras estructuras de control claves. Los *buffers* de E/S y otras áreas críticas también se ponen en marcos bloqueados.

El bloqueo se puede realizar asociando un bit de bloqueo a cada marco, que se puede almacenar en la tabla de marcos o incluirse en la tabla de páginas actual.



Algoritmos básicos

Independientemente de la estrategia de gestión del conjunto residente, existen ciertos algoritmos básicos que se utilizan para la selección de la página a reemplazar:

- Óptimo.
- Usado menos recientemente (*least recently used*, LRU).
- FIFO (*first-in-first-out*).
- Reloj.

La política óptima tomará como reemplazo la página cuya próxima referencia esté más alejada. Se basa en conseguir el menor número posible de fallos de página. Es imposible de implementar, ya que el SO necesitaría un conocimiento perfecto de los eventos futuros, símplemente es un estándar.

La política óptima asume una reserva de marcos fija (tamaño del conjunto residente fijo), por ejemplo tres. Si la ejecución de un proceso requiere la referencia de cinco páginas diferentes y su flujo es: 2 3 2 1 5 2 4 5 3 2 5 2, la política es la siguiente, que produce tres fallos de página después de que la reserva de marcos se haya ocupado completamente:

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

La política de reemplazo de la página usada menos recientemente (LRU) seleccionará como candidata la página de memoria que no se haya referenciado desde hace más tiempo. Por el principio de proximidad, sería la que tiene menos probabilidad de volver a ser referenciada, por lo que proporciona resultados casi tan buenos como la política óptima.

El problema es su dificultad de implementación. Una opción sería etiquetar cada página con el instante de tiempo de su última referencia, ya sean reservas de memoria, instrucciones o datos, generando una sobrecarga tremenda. De forma alternativa, se puede mantener una pila de referencias a páginas, aunque sigue siendo una opción costosa.

Para el mismo ejemplo anterior, esta política produce cuatro fallos de página:

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

La política FIFO trata los marcos de página ocupados como un *buffer* circular, reemplazando las páginas cíclicamente como round-robin. Sólo necesita un puntero que recorra los marcos de forma circular, siendo una de las políticas más sencillas de implementar. De esta forma, se reemplaza la página que lleve más tiempo en memoria, que puede haber dejado de utilizarse.

Esto es erróneo, ya que en los programas existen zonas o regiones de datos que se utilizan de forma intensiva durante el ciclo de vida del proceso, por lo que en este caso serían expulsadas y traídas de forma repetida, haciendo que su rendimiento sea más pobre.

Para el ejemplo anterior, esta política produce seis fallos de página:

2	3	2	1	5	2	4	5	3	2	5	2
2*	2	2	2	5*	5	5	5	3*	3*	3	3
	3*	3*	3	3	2*	2	2	2	2	5*	5
			1*	1	1	4*	4*	4	4	4	2*
				F	F	F		F		F	F

A lo largo de los años se ha buscado aproximarse a los resultados obtenidos por LRU reduciendo la sobrecarga. Muchos de estos algoritmos son variantes del esquema de política de reloj.

En su forma más sencilla, la política de reloj requiere la inclusión de un bit adicional en cada marco de página, el bit de usado. Cuando una página se trae por primera vez a memoria, su bit de usado se pone a 1, al igual que cuando vuelva a utilizarse. Para el algoritmo de reemplazo de páginas, el conjunto de candidatas (de este proceso, ámbito local; de toda la memoria principal, ámbito global) se dispone como un *buffer* circular, al cual se asocia un puntero. Cuando se reemplaza una página, el puntero indica el siguiente marco del *buffer* justo después del marco que acaba de utilizarse. Cuando llega el momento de reemplazar una página, el SO recurre el *buffer* buscando un marco con el bit de usado a 0, siendo el primero que encuentre el reemplazado. Cada vez que encuentra un marco con dicho bit a 1, lo cambia a 0 y continúa. Si todos los marcos tuvieran inicialmente un 1, el puntero volvería a su posición original, encontrando ahora un 0.

En el mismo ejemplo, y suponiendo que el puntero indica que el bit de usado es 1 y la flecha la posición actual del puntero:

2	3	2	1	5	2	4	5	3	2	5	2
2*	2*	2*	>2*	5*	5*	>5*	>5*	3*	3*	>3*	>3*
>	3*	3*	3*	>3	2*	2*	2*	>2	>2*	2	2*
	>	>	1*	1	>1	4*	4*	4	4	5*	5*
				F	F	F		F		F	F

Si analizamos los cuatro algoritmos en las mismas condiciones, con tamaños de marcos fijo, y los organizamos en función del número de fallos de página producidos, el resultado es el siguiente de peor a mejor: FIFO < RELOJ < LRU < OPT.

En el caso en el que los tamaños son variables y se aplican ámbitos de reemplazamiento tanto global como local, el algoritmo de reloj se acerca aún más al LRU.

El algoritmo de reloj puede hacerse más potente incrementando el número de bits que utiliza, asemejándose al FIFO si se reduce. En todos los procesadores que soportan paginación, se asocia un bit de modificado a cada página, que implica que una página no puede reemplazarse hasta que se haya escrito de nuevo en memoria secundaria. Si tenemos en cuenta los bits de usado y modificado, cada marco de página cae en una de las siguientes características:

- No se ha accedido recientemente, no modificada (u = 0; m = 0).
- Accedida recientemente, no modificada (u = 1; m = 0).
- No se ha accedido recientemente, modificada (u = 0; m = 1).
- Accedida recientemente, modificada (u = 1; m = 1).

Con esta clasificación, el algoritmo funciona de la siguiente manera:

1. Comenzando por la posición actual del puntero, se recorre el *buffer* de marcos, sin hacer cambios en el bit de usado. El primer marco que se encuentre con (u = 0; m = 0) se selecciona para el reemplazo.
2. Si el paso 1 falla, se recorre el *buffer* de nuevo, buscando un marco con (u = 0; m = 1). El primer marco que se encuentre se selecciona para el reemplazo, poniendo el bit de usado a 0 en cada uno de los marcos que se vayan saltando.
3. Si el paso 2 falla, el puntero debe haber vuelto a la posición original y todos los marcos tendrán el bit de usado a 0. Se repite el paso 1 y, si fuera necesario, el 2, encontrando ahora un marco para el reemplazo.

En resumen, el algoritmo busca una página que no se haya modificado desde que se ha traído y que no haya sido accedida recientemente para evitar que tenga que escribirse primero en memoria secundaria, ahorrando tiempo inmediato. Si no encuentra una con esas características, da una segunda vuelta buscando una que haya sido modificada pero que no se haya accedido

recientemente. Aunque tenga que ser escrita, por el principio de proximidad puede no necesitarse en un futuro próximo. Si falla también, todos los marcos habrían sido accedidos recientemente, realizando una tercera pasada.

Buffering de Páginas

A pesar que las políticas LRU y de reloj son superiores a FIFO, ambas incluyen más complejidad y sobrecarga. Además, hay que tener en cuenta que el coste de reemplazo de una página que se haya modificado es superior a una que no lo haya sido.

Una política más sencilla que mejora el rendimiento de la paginación es el *buffering* de páginas. Teniendo en cuenta que el algoritmo más sencillo es el FIFO, una página reemplazada no se pierde, si no que se asigna a una lista de páginas libres si no se ha modificado, o a una de páginas modificadas si lo ha sido. La página no se mueve físicamente de la memoria, sino que su entrada se elimina de la tabla de páginas y se coloca en la lista correspondiente.

La lista de páginas libres es una lista de marcos disponibles para lectura de nuevas páginas, intentando mantener un pequeño número en todo momento. Cuando una página se va a leer, se utiliza el marco de página en cabeza de esta lista, eliminando la página que contenía.

La lista de páginas que se va a reemplazar se mantiene en memoria, por lo que si es referenciada de nuevo, se devuelve al conjunto residente con bajo coste, actuando las nuevas listas como caché de páginas.

La lista de páginas modificadas permite que estas se escriban en grupos en lugar de una en una, reduciendo el número de operaciones de E/S y, por tanto, el tiempo de acceso a disco.

Política de Reemplazo y Tamaño de la Caché

Actualmente, grandes tamaños de cachés (incluso megabytes) son alternativas de diseño abordables que mejoran el rendimiento del reemplazo de páginas. Si el marco de página destinado al reemplazo está en caché, el bloque de caché se pierde al mismo tiempo que la página que lo contiene.

En sistemas que utilizan algún tipo de *buffering* de páginas, se puede mejorar el rendimiento de la caché añadiendo a la política de reemplazo una política de ubicación en el *buffer* de páginas. La mayoría de los SO ubican las páginas seleccionando un marco procedente del *buffer* con una disciplina FIFO.

Gestión del Conjunto Residente

Tamaño del Conjunto Residente

Con la memoria virtual paginada casi nunca es posible traer a memoria todas las páginas de un proceso. El SO sí que debería conocer cuántas páginas debe traerse, es decir, cuánta memoria principal reservar para cada proceso. Esto afecta en diversos factores:

- Cuanto menor es la cantidad de memoria reservada para un proceso, mayor es el número de procesos que pueden residir en memoria principal a la vez, aumentando la probabilidad de que al menos un proceso esté Listo y disminuyendo el *swapping*.
- Si el conjunto de páginas de un proceso en memoria es relativamente pequeño, por el principio de proximidad, la probabilidad de fallo de página es mayor.
- La reserva de más memoria principal para un determinado proceso no tendrá efectos apreciables sobre su tasa de fallos de página, por el principio de proximidad.

Por ello encontramos dos tipos de políticas en los SO contemporáneos:

La política de asignación fija proporciona un número fijo de marcos de memoria disponibles para ejecución, decidido en el momento de la creación del proceso en base a su tipo o las especificaciones del programador. Siempre que se produzca un fallo de página del proceso en ejecución, la página necesitada reemplazará a una de las del proceso.

Una política de asignación variable permite que el número de marcos reservados para un proceso varíe a lo largo de su vida. Así, a un proceso que esté provocando fallos de página constantemente (el principio de proximidad se le aplica de forma débil) se le otorgarán marcos de página adicionales para reducir dicha tasa; mientras que a uno con una tasa de fallos excepcionalmente baja se le reducirán, intentando que su tasa no aumente en exceso.

La política de asignación variable parece más potente, pero provoca que el SO tenga que saber el comportamiento del proceso activo, con su correspondiente sobrecarga.

Ámbito de Reemplazo

La estrategia del ámbito de reemplazo se puede clasificar en global y local. Ambos tipos se activan por medio de un fallo de página cuando no existen marcos de página libres.

Una política de reemplazo local selecciona únicamente entre las páginas residentes del proceso que ha generado el fallo de página; mientras que una política de reemplazo global considera todas las páginas que no se encuentren bloqueadas.

Las políticas locales son más fáciles de analizar, pero no existen evidencias de que proporcionen un rendimiento mejor que las globales, que son más fáciles de implementar y producen una sobrecarga menor.

Existe una correlación entre el ámbito de reemplazo y el tamaño del conjunto residente: un conjunto residente fijo implica una política de reemplazo local para mantener el tamaño del conjunto residente, mientras que una política de asignación variable puede emplear una política de reemplazo global. La asignación variable y el reemplazo local también es una combinación válida.

Asignación fija, ámbito local

Se parte de un proceso en ejecución con un número de marcos fijo. Si genera un fallo de página, el SO debe elegir una de sus páginas residentes para el reemplazo.

Con la política de asignación fija, es necesario saber por adelantado la cantidad de espacio reservado para cada proceso, que puede ser en base de su tipo y del tamaño del programa.

Tiene dos desventajas:

- Si las reservas resultan ser demasiado pequeñas, va a aumentar la tasa de fallos, ralentizando el sistema multiprogramado.
- Si las reservas resultan demasiado grandes, habrá muy pocos programas en memoria principal, mucho tiempo del procesador ocioso y mucho tiempo perdido en *swapping*.

Asignación variable, ámbito global

Es la combinación más sencilla de implementar y ha sido adoptada por muchos SO. En todo momento existen una serie de procesos en memoria principal, cada uno con una serie de marcos asignados. El SO también suele mantener una lista de marcos libres.

Cuando sucede un fallo de página, se añade un marco libre al conjunto residente de un proceso y se trae la página a dicho marco, de tal forma que un proceso que vaya sufriendo fallos irá aumentando su tamaño, lo que reduciría la tasa de fallos global.

La dificultad viene cuando el SO tiene que elegir un reemplazo al no quedar marcos libres. La selección se produce entre todos los marcos que no estén bloqueados, pudiendo ser una página de un proceso cualquiera y no el óptimo.

Este problema de rendimiento puede reducirse con el uso de *buffering* de páginas, haciendo que la selección de los reemplazos no sea tan significativa debido a que se pueden reclamar si se hace una referencia antes de que se sobreescriban.

Asignación variable, ámbito local

La asignación variable con reemplazo de ámbito local intenta resolver los problemas de la estrategia de ámbito global. Se puede resumir así:

1. Cuando se carga un nuevo proceso en memoria, se le asignan un cierto número de páginas al conjunto residente en función del tipo de aplicación, solicitudes del programa u otros criterios. Se puede utilizar para ello paginación adelantada o paginación por demanda.
2. Cuando ocurre un fallo de página, el reemplazo se seleccionará del conjunto residente del proceso que causó el fallo.
3. De vez en cuando, se reevaluará la asignación proporcionada a cada proceso, incrementándose o reduciéndose para mejorar el rendimiento.

Con esta estrategia, las decisiones de aumentar o disminuir el tamaño del conjunto residente se toman en función de posibles demandas futuras de los procesos activos. Esta estrategia es más compleja por esta valoración, pero puede mejorar el rendimiento.

Los elementos clave para esta estrategia se utilizan para determinar el tamaño del conjunto residente y la periodicidad de estos cambios. Una estrategia específica muy estudiada es la del conjunto de trabajo, a pesar de que es difícil de implementar.

El conjunto de trabajo con parámetro □ para un proceso en el tiempo virtual *t*, *W(t,□)*, es el conjunto de páginas del proceso a las que se ha referenciado en las últimas □ unidades de tiempo virtual, que contabiliza el número de direcciones virtual generadas.

La variable □ es la ventana de tiempo virtual a través de la cual se observa el proceso. Para mayor tiempo ventana, el tamaño del conjunto de trabajo también es mayor: $W(t+1) \supseteq W(t,)$.

El conjunto de trabajo es también una función del tiempo. Si un proceso ejecuta durante □ unidades de tiempo, y terminando el tiempo utiliza una única página, entonces $|W(t,) = 1$. Un conjunto de trabajo también puede crecer hasta llegar a las *N* páginas del proceso si se accede rápidamente a muchas páginas diferentes y si el tamaño de la ventana lo permite: $1 \leq W(t,) \leq min(,N)$.

Para muchos programas, los períodos relativamente estables del tamaño de su conjunto de trabajo se alternan con periodos de cambio rápido. Cuando un proceso comienza a ejecutarse, va construyendo su conjunto de trabajo conforme referencia nuevas páginas. Debido al principio de proximidad, el proceso se estabilizará sobre un conjunto determinado de páginas. Los periodos transitorios posteriores reflejan el cambio de un programa a una nueva región de referencia. Durante dicha fase, algunas páginas permanecerán dentro de la ventana □ causando un rápido incremento del tamaño del conjunto de trabajo conforme se van referenciando nuevas páginas. A medida que la ventana se desplaza de estas referencias, el tamaño del conjunto de trabajo se reduce hasta que contiene únicamente aquellas páginas de la nueva región de referencia.

El concepto de conjunto de trabajo se puede usar para crear la estrategia del tamaño del conjunto residente:

1. Monitorizando el conjunto de trabajo de cada proceso.
2. Eliminando periódicamente del conjunto residente aquellas páginas que no se encuentran en el conjunto de trabajo, lo que es en esencia la política LRU.

3. Un proceso puede ejecutar sólo si su conjunto de trabajo se encuentra en memoria principal.

Esta estrategia funciona debido a que parte de un principio aceptado, el de proximidad, y lo explota para conseguir una estrategia de gestión de memoria que minimice los fallos de página. Sin embargo, siguen existiendo varios problemas en la estrategia del conjunto de trabajo:

- El pasado no siempre predice el futuro. Tanto el tamaño como la pertenencia al conjunto de trabajo cambian a lo largo del tiempo.
- Una medición verdadera del conjunto de trabajo para cada proceso no es practicable, ya que cada proceso tendría que almacenar en una lista ordenada por tiempo sus páginas.
- El valor óptimo de □ es desconocido y puede variar.

Sin embargo, el espíritu de esta estrategia es válido y muchos SO se aproximan a ella centrándose en la tasa de fallos de página, que cae a medida que se incrementa el tamaño del conjunto residente de un proceso, en lugar de en el tamaño del conjunto de trabajo.

De esta forma, si la tasa de fallos de páginas de un proceso está por debajo de un límite, el sistema se beneficia reduciendo el tamaño de su conjunto residente; mientras que si está por encima de un umbral máximo, el sistema puede permitirse incrementar el tamaño del conjunto residente.

Un algoritmo que sigue esta estrategia es el de frecuencia de fallos de página (*page fault frequency*, PFF), que asocia un bit de usado a cada página que se pondrá a 1 cuando se haya accedido a la misma. Cuando se produce un fallo de página, el SO anotará el tiempo virtual desde el último fallo de página para dicho proceso gracias a un contador de las referencias a página. Fijando un umbral *F*, si la diferencia de tiempo con el último fallo de página es menor que este, se añade una página al conjunto residente del proceso; en otro caso, se descartan todas las páginas con el bit de usado a 0. Esto puede refinarse usando dos umbrales, uno máximo para disparar el crecimiento y uno inferior para disparar su reducción.

Existe un fallo grave en esta estrategia, que hace que su comportamiento no sea bueno durante los periodos transitorios de desplazamiento a una nueva región de referencia. Con PFF, ninguna página sale del conjunto residente antes de que hayan pasado *F* unidades de tiempo desde su última referencia. La rápida sucesión de fallos de página en estas transiciones pueden provocar cambios de proceso y sobrecargas de *swapping* no deseables.

Una estrategia que intenta manejar este problema es una sobrecarga más baja es la política de conjunto de trabajo con muestreo sobre intervalos variables (*variable-interval sampled working set*, VSWS), que evalúa el conjunto de trabajo del proceso en instantes de muestreo basados en el tiempo virtual transcurrido.

Al comienzo del intervalo de muestreo, los bits de usado de las páginas residentes de procesos se ponen a 0; al final, solo las páginas a las que se ha hecho referencia durante el intervalo mantendrán ese bit a 1. Las que tengan el bit a 1 se mantendrán en memoria durante el siguiente intervalo, mientras que el resto se descartan. Así, el tamaño del conjunto residente sólo disminuye al

final del intervalo. Durante el intervalo, todas las páginas que han causado fallo se añaden al conjunto residente, que se mantiene fijo o crece durante el intervalo.

La política VSWS toma tres parámetros:

- *M*: duración mínima del intervalo de muestreo.
- *L*: duración máxima del intervalo de muestreo.
- *Q*: número de fallos de página que se permite que ocurran entre dos instantes de muestreo.

La política VSWS es la siguiente:

1. Si el tiempo virtual entre el último muestreo alcanza *L*, se suspende el proceso y se analizan los bits de usado.
2. Si antes de que el tiempo virtual transcurrido llegue a *L* ocurren *Q* fallos de página,
    a. Si el tiempo virtual desde el último muestreo es menor que *M*, se espera hasta que alcance dicho valor para suspender el proceso y analizar sus bits.
    b. Si el tiempo virtual desde el último muestreo es mayor o igual a *M*, se suspende el proceso y se analizan sus bits de usado.

Los valores de los parámetros se toman de forma que el muestreo se dispare habitualmente cuando ocurre el fallo de página *Q* después del último muestreo (caso 2b). Los parámetros *M* y *L* suponen fronteras de protección para condiciones excepcionales.

La política VSWS intenta reducir el pico de solicitudes de memoria causadas por una transición abrupta entre dos áreas de referencia, incrementando la frecuencia de muestreo y la tasa a la cual las páginas no utilizadas se descartan.

Política de Limpieza

La política de limpieza se encarga de determinar cuándo una página que está modificada se debe escribir en memoria secundaria, resultando el opuesto de la política de recuperación.

Las dos alternativas más comunes son la limpieza bajo demanda, con la que una página se escribe en memoria secundaria sólo cuando se ha seleccionado para un reemplazo; y la limpieza adelantada, en la que se escribe antes de que se necesite su marco de página, de tal forma que se pueden escribir en bloques.

Ninguna puede llevarse al extremo. Con la limpieza adelantada, una página que se escriba en memoria puede volver a ser modificada antes de que se reemplace, por lo que se realizarían operaciones de limpieza innecesarias. La limpieza bajo demanda coincide con la lectura de una nueva página, por lo que un proceso debe esperar a que se completen dos transferencias de páginas antes de poder desbloquearse, reduciendo la utilización del procesador.

Una estrategia más apropiada incorpora *buffering* de páginas para limpiar sólo las páginas que son reemplazables, desacoplando las operaciones de limpieza y reemplazo. Con *buffering* de páginas, las páginas reemplazadas pueden ubicarse en dos listas: modificadas y no modificadas. Las páginas de la lista de modificadas pueden escribirse periódicamente por lotes y moverse a la lista de no modificadas. Una página en la lista de no modificadas puede ser reclamada si se referencia o perderse cuando su marco se asigna a otra página.

Control de Carga

El control de carga determina el número de procesos que residirán en memoria principal, es decir, el grado de multiprogramación. Es una función crítica para una gestión de memoria efectiva.

Si hay muy pocos procesos en memoria a la vez, habrá muchos momentos en los que estén bloqueados y gran parte del tiempo se gastará realizando *swapping*. Si hay demasiados, el tamaño del conjunto residente de cada proceso será poco adecuado y se producirán muchos fallos de página, resultando en trasiego (*thrashing*).

Grado de Multiprogramación

A medida que el nivel de multiprogramación aumenta hasta cierto punto, aumenta la utilización del procesador ya que hay menos posibilidades de que los procesos residentes se encuentren bloqueados. Sin embargo, cuando se alcanza dicho punto, el número de fallos de páginas se incrementa de forma dramática, colapsando la utilización del procesador.

Esto puede abordarse de varias maneras. Un algoritmo del conjunto de trabajo o de frecuencia de fallos incorporan de forma implícita control de carga. Sólo se pueden ejecutar aquellos procesos con el conjunto residente lo suficientemente grande. Así, de forma automática se regula el número de procesos activos.

Otra estrategia es el criterio de L = S, que ajusta el nivel de programación de forma que el tiempo medio entre fallos de página se iguala al tiempo medio necesario para procesar un fallo, que es el punto en el que la utilización del procesador es máxima.

Una política con un efecto similar es el criterio del 50%, que intenta mantener la utilización del dispositivo de paginación aproximadamente al 50%, donde la utilización del procesador también es máxima.

Otra alternativa es adaptar el algoritmo de reemplazo de páginas del reloj, monitorizando la tasa a la cual el puntero recorre el *buffer* circular de marcos. Si la tasa está por debajo de un nivel de umbral dado, éste indica una o las dos circunstancias siguientes:

1. Si están ocurriendo pocos fallos de página, que implican pocas peticiones para avanzar el puntero.

2. Por cada solicitud, el número de marcos medio que se recorren por el puntero es pequeño, lo que indica que hay muchas páginas residentes a las que no se hace referencia y son reemplazables.

En ambos casos, el grado de multiprogramación puede implementarse con seguridad. Si la tasa de recorrido circular del puntero supera el umbral máximo, la tasa de fallos sería alta o habría dificultad para encontrar páginas reemplazables, indicando un grado de multiprogramación demasiado alto.

Suspensión de Procesos

Si se va a reducir el grado de multiprogramación, uno o más de los procesos residentes debe suspenderse, habiendo seis posibilidades:

- Procesos con baja prioridad: es una decisión de la política de activación, sin tener relación con el rendimiento.
- Procesos que provocan muchos fallos: hay una gran probabilidad de que la tarea que causa los fallos no tenga su conjunto de trabajo residente, por lo que el rendimiento se incrementaría si se suspendiese. Además, se estaría suspendiendo un proceso que estaría a punto de bloquearse, evitando la sobrecarga del reemplazo de páginas y la operación de E/S.
- Proceso activado hace más tiempo: es el proceso que tiene menor probabilidad de tener su conjunto de trabajo residente.
- Proceso con el conjunto residente de menor tamaño: éste requerirá menor esfuerzo al cargarse de nuevo, aunque penaliza a aquellos programas con una proximidad de referencias muy fuerte.
- Proceso mayor: esto proporciona un mayor número de marcos libres, ideal para una memoria sobrecargada, haciendo que futuras desactivaciones sean poco probables a corto plazo.
- Proceso con la mayor ventana de ejecución restante: esto se aproxima a la disciplina de activación de primero el proceso con menor tiempo de ejecución, añadiendo la característica de que los procesos sólo pueden ejecutar un determinado *quantum* seguido.

Hiperpaginación

Si el número de marcos de página asignados a un proceso no es suficiente para almacenar las páginas referenciadas activamente por el mismo, se producirá un número elevado de fallos de página, a lo que se denomina hiperpaginación (*thrashing*).

Cuando se produce hiperpaginación, el proceso pasa más tiempo en la cola del servicio del dispositivo de paginación que ejecutando, lo que puede afectar al proceso individual y a todo el sistema.

En un SO con asignación fija, la hiperpaginación conlleva el aumento considerable del tiempo de ejecución del proceso que la sufre, pero el resto de procesos no se ven afectados directamente.

Con asignación dinámica, el número de marcos se va adaptando a las necesidades de cada proceso, por lo que, en principio, no debería producirse hiperpaginación. Sin embargo, si el número de marcos de página existentes no es suficiente para almacenar los conjuntos de trabajo de todos los procesos, se producirían fallos de página frecuentes, sufriendo así hiperpaginación. La utilización del procesador disminuirá drásticamente al aumentar el tiempo de tratamiento de fallos de página, ya que hay más y la cola de servicio del dispositivo será más larga.

Cuando se produce esta situación, se deben suspender uno o varios procesos, liberando sus páginas. La estrategia de control de carga es la que tiene que evitar que se produzca la hiperpaginación controlando el grado de multiprogramación. Para ello, hay tres políticas principales.

Estrategia del Conjunto de Trabajo

Cuando un proceso tiene residente su conjunto de trabajo, se produce una tasa baja de fallos de página, por lo que una estrategia consiste en determinar los conjuntos de trabajo de todos los procesos activos para intentar mantenerlos en memoria principal.

El conjunto de trabajo de un proceso es el conjunto de páginas asignadas por un proceso en las últimas *n* referencias, lo que se denomina la ventana del conjunto de trabajo. El valor *n* es un factor crítico para el funcionamiento de la estrategia, ya que si es demasiado grande, se podrían sobreestimar las necesidades del proceso; mientras que si es demasiado pequeño, la ventana no englobaría la situación actual del proceso, generando demasiados fallos de página.

Suponiendo que el SO puede identificar el conjunto de trabajo de cada proceso, una estrategia con asignación dinámica con reemplazo local y control de carga sería:

- Si el conjunto de trabajo de un proceso decrece, se liberan los marcos asociados a las páginas que ya no están en el mismo.
- Si el conjunto de trabajo de un proceso crece, se asignan marcos que puedan contener las nuevas páginas que han entrado en este. Si no hay marcos libres, se debe realizar un control de carga suspendiendo procesos y liberando sus páginas.

El problema de la estrategia es cómo determinar el conjunto de trabajo de cada proceso, ya que se necesitaría una MMU específica que fuera controlando las páginas accedidas por cada proceso durante las últimas *n* referencias.

Estrategia de Administración basada en la Frecuencia de Fallos de Página

Esta estrategia busca una solución más directa al problema de la hiperpaginación controlando la frecuencia de fallos de página de cada proceso mediante una cuota superior y otra inferior.

Con esta idea, se describe una estrategia de asignación dinámica con reemplazo local y control de carga:

- Si la frecuencia de fallos de un proceso supera el límite superior, se le asignan páginas adicionales. Si no hay marcos libres, se suspende algún proceso, liberando sus páginas.
- Si la frecuencia de fallos de un proceso es menor que el límite inferior, se liberan marcos de página asignados al mismo mediante un algoritmo de reemplazo.

Estrategia de Control de Carga para Algoritmos de Reemplazo Globales

Los algoritmos de reemplazo globales no controlan la hiperpaginación, por lo que necesitan trabajar junto con un algoritmo de control de carga.

Por ejemplo, el sistema de gestión de memoria de UNIX BSD usa *buffering* de páginas. Existe un proceso (*page daemon*) que se despierta periódicamente y comprueba si hay suficientes marcos de página libres. Si no es así, aplica el algoritmo de reemplazo y libera el número necesario.

La estrategia de reemplazo es global y utiliza una modificación del algoritmo de reloj denominada reloj con dos manecillas, ya que utiliza dos punteros. Cuando se ejecuta, se desactiva el bit de referencia de la página a la que apunta el primer puntero y comprueba el bit de referencia de la que señala el segundo. Si está desactivado, se libera la página y la escribe en disco si estuviera modificada. El algoritmo avanza ambos punteros y repite el proceso hasta que se liberan suficientes marcos.

Cuando la tasa de paginación del sistema es demasiado alta y el número de marcos libres está frecuentemente por debajo del mínimo, otro proceso especial (*swapper*) selecciona procesos para suspenderlos y liberar sus marcos. El *swapper* comprueba periódicamente si alguno de los procesos expulsados debe reactivarse, y sólo lo lleva a cabo si hay suficientes marcos libres.

Gestión de Memoria en Linux

Linux comparte muchas características de los esquemas de gestión de memoria de UNIX, pero añade otras características, haciéndolo bastante complejo. Los dos aspectos principales son la memoria virtual de los procesos y la asignación de la memoria del núcleo.

Memoria Virtual en Linux

Direccionamiento de la Memoria Virtual

Linux usa una estructura de tablas de páginas de tres niveles, dividiéndose en los siguientes tipos:

- Directorio de páginas. Un proceso activo tiene un directorio de páginas único que tiene el tamaño de una página. Cada entrada apunta a una página en el directorio intermedio de páginas. Debe residir en la memoria principal cuando el proceso está activo.
- Directorio intermedio de páginas. Se expande a múltiples páginas. Cada entrada apunta a una página que contiene una tabla de páginas.
- Tabla de páginas. También puede expandirse a múltiples páginas. Cada entrada hace referencia a una página virtual del proceso.

Para utilizar esta estructura, una dirección virtual en Linux es consistente en cuatro campos:

- Directorio global: se utiliza como índice en el directorio de páginas, siendo el más significativo.
- Directorio intermedio: sirve como índice en el directorio intermedio de páginas.
- Tabla de páginas: se utiliza para indexar la tabla de páginas.
- Desplazamiento: proporciona el desplazamiento dentro de la página seleccionada.

La estructura de la tabla de páginas en Linux es independiente de la plataforma, aunque se diseñó para el procesador Alpha de 64 bits. Con direcciones de 64 bits, la utilización de dos niveles resultaría en tablas de páginas y directorios de gran tamaño. Linux define el tamaño del directorio intermedio de páginas como 1. Las referencias a este nivel se eliminan en la optimización de la compilación, por lo que no hay sobrecarga de rendimiento en la utilización de este diseño de tres niveles en plataformas que soportan dos.

Reserva de Páginas

Para mejorar la eficiencia de la lectura y escritura de páginas en memoria principal, Linux define un mecanismo para manejar bloques de páginas contiguas que se proyectarán sobre bloques de marcos de página también contiguos. Para ello, se utiliza el sistema *buddy*. El núcleo mantiene una lista de marcos de página contiguos de tamaño fijo, ya sean de 1, 2, 4, 8, 16 ó 32 marcos. A lo largo de su uso, las páginas se asignan y liberan de memoria principal, los grupos disponibles se dividen y juntan utilizando el algoritmo del sistema *buddy*.

Algoritmo de Reemplazo de Páginas

El algoritmo que utiliza Linux se basa en un algoritmo de reloj. En el sencillo, se asocia un bit de usado y otro bit de modificado con cada una de las páginas de memoria principal. En Linux, el de

usado se reemplaza por una variable de 8 bits, de tal forma que cada vez que se accede a una página, la variable se incrementa.

En segundo plano, Linux recorre la lista de páginas y decrementa la variable de edad de cada página a medida que va rotando por ellas. Una página de edad 0 es una página vieja a la que no se ha hecho referencia desde hace algún tiempo, siendo el mejor candidato para el reemplazo. Cuando el valor de edad es más alto, la frecuencia con la que se ha accedido a la página recientemente es mayor, por lo que tiene menos posibilidad de elegirse para el reemplazo. EL algoritmo de reemplazo de Linux resulta así en una variante de la política LRU.

Reserva de Memoria del Núcleo

La gestión de la memoria del núcleo se realiza en base a los marcos de página de la memoria principal. Su función básica es asignar y liberar marcos para los diferentes usos. Los posibles propietarios de un marco incluyen procesos en espacio de usuario, datos del núcleo reservados dinámicamente, código estático del núcleo y caché de páginas.

Los fundamentos de la reserva de memoria de núcleo para Linux son los mecanismos de reserva de páginas ya usados para la gestión de la memoria virtual de usuario. Se utiliza un algoritmo *buddy* de forma que la memoria del núcleo se pueda reservar y liberar en unidades de una o más páginas.

La unidad mínima de reserva es la página, y dado que el núcleo requiere fragmentos que se utilizan durante poco tiempo y de tamaño variable, puede ser ineficiente. Para ajustarse a esos pequeños tamaños, Linux utiliza un esquema de asignación por láminas (*slab allocation*) dentro de una página ya reservada. En esencia, Linux mantiene un conjunto de listas enlazadas, una para cada tamaño de fragmento. Todos los fragmentos se pueden dividir y agregar de una manera similar al indicado por el algoritmo *buddy*, y también se pueden mover entre las distintas listas.