Capítulo 3

Memoria

El sistema operativo debe co-residir en la memoria principal con uno o más procesos que se estén ejecutando. La administración de memoria es responsable de la asignación de memoria a los procesos y de proteger esa memoria asignada a cada proceso de accesos no deseados de otros procesos. También es responsable de proteger la memoria asignada al sistema operativo de accesos no autorizados.

La administración de memoria no es sólo tarea del software. El sistema operativo requiere soporte de hardware para implementar cualquiera de los esquemas de administración de memoria no triviales. Por esto, algunos aspectos del diseño del sistema operativo también son aspectos de diseño del hardware. El hardware de administración de memoria típicamente está protegido del acceso por parte de los usuarios; el sistema operativo solamente es responsable de su control.

Esta fuera del alcance de este Capítulo extenderse sobre el hardware, pues el objetivo es manejarnos en un nivel de abstracción con respecto a la capa física. Se detallará solo cuando sea necesario.

3.1. El SO como administrador de memoria

Como administrador de la memoria, un sistema operativo debe cumplir con las siguientes funciones:

- llevar el control sobre qué partes de la memoria están en uso y cuáles no;
- asignar memoria a procesos cuando lo necesiten y liberarla cuando estos finalicen;
- administrar el intercambio entre memoria principal y secundaria durante el swapping.

Cuando debemos evaluar un sistema, con respecto a la memoria, nos interesa definir:

- ¿Que cantidad de memoria necesito?
- ¿A que velocidad?
- ¿A que costo?

Hay pautas a considerar:

- La cantidad de memoria dependerá del tipo de aplicaciones a ejecutar y al tipo de sistema: no es igual el requerimiento en un sistema de tiempo real, que en un batch.
- Para mayor rendimiento, la memoria debe compatibilizar la memoria del procesador.
- El costo debe ser razonable con respecto al resto de las componentes.
- A menor tiempo de acceso, mas costo por bit.
- A mayor capacidad, menos costo por bit y mayor tiempo de acceso.

Según sugiere Stallings en su libro (ver [Stallings, 2001]), para lograr un buen resultado, no deberíamos depender de un tipo de componente o tecnología. Existe una jerarquía de memoria que nos permite hacer un ranking de velocidad de acceso, costo, etc.

Jerarquía tradicional	Jerarquía moderna
Registros	Registros
Cache	Cache
Memoria principal	Memoria principal
Disco magnético	Caché de disco
Cinta magnética	Disco magnético
	Cinta magnética/disco óptico

Cuadro 3.1: Jerarquía de memorias

A medida que bajamos de nivel de jerarquía vamos obteniendo menos costo por bit, mas capacidad, mas tiempo de acceso.

Un punto sumamente importante es la frecuencia de acceso a la memoria de parte del procesador. Para entenderlo consideremos que a la información mantenida en el primer nivel (registros) el acceso es inmediato. Si está en el 2do nivel (cache), primero debe pasar al primer nivel y luego acceder el procesador.

Hay un principio llamado "cercanía de referencias" que observa que cuando se ejecuta un programa, las referencias a memoria desde el procesador, tienden a estar agrupadas.

Cuando está ejecutando el programa principal las referencias a instrucciones cercanas o al llamar una subrutina, también. Es el concepto de localidad que veremos mas profundamente en la subsección 3.4.8 de hiperpaginación (*thrashing*).

Lo ideal seria tener acceso rápido al conjunto de direcciones agrupadas (o localidad) pues son las de uso inminente, y que cada nueva localidad desplace a la anterior y ocupe este lugar de preferencia.

De la jerarquía, la memoria por registros es el método mas caro, pero el mas rápido. Normalmente las máquinas tienen decenas de registros, aunque hay otras que poseen cientos.

La cache (2do nivel)[1] es una memoria de alta velocidad, no visible normalmente para el programador o el procesador. Es un dispositivo para mejorar

[1] Sobre memoria cache hay una buena introducción en el libro de W. Stallings.

el rendimiento, acelerando el curso de los datos entre la memoria principal y los registros del procesador. Los registros, la cache y la principal, son volátiles. El resto de las jerarquías son residentes en dispositivos externos, y por lo tanto, no volátiles.

La memoria expandida es una forma de memoria interna, mas lenta y menos cara que la principal. No es una nueva jerarquía. Los datos se mueven entre la principal y la expandida, pero no entre la expandida y la externa.

Hay niveles adicionales, por software. Por ejemplo la Cache de disco, que mejora el rendimiento agrupando escrituras a disco, para transferir mas datos, menos veces,. Además esto permite que aunque un proceso haya terminado de usar un dato, este dato permanece un rato mas en memoria permitiendo ser accedido por un proceso posterior que lo requiera (este tema lo veremos en Buffer Cache en el sistema UNIX).

Es de fundamental importancia que quede muy bien entendido los "cuellos de botella" con los que se encuentra un proceso. Vea en el Cuadro 3.2 los tiempos típicos de los distintos dispositivos, pero compárelos con respecto al primero:

Tipo de dispositivo	Tiempo típico de servicio	Relativo al segundo
Buffer proc.	10 ns	1 s
Mem.acc.aleat.	60 ns	6 s
Llamada proc.	1 μs	2 m
Mem.expand.	25 μs	1 h
RPC local	100 μs	4 h
Disco estado sólido	1 ms	1 d
Disco "cacheado"	10 ms	12 d
Disco magnético	25 ms	4 sem
Disco via LAN alta vel.	27 ms	1 mes
Disco via LAN serv.	35-50 ms	6-8 sem
Disco via WAN serv.	1-2 s	3-6 años
Disco/cinta montable	3-15s	10-15 años

Cuadro 3.2: Tiempos de acceso a distintas memorias

Los requisitos que se deben satisfacer en la administración de memoria son:

- Reubicación: permitir que un programa pueda ser "movido", descargado y cargado dentro de la memoria sin problemas de direccionamiento (recordemos los cambios de contexto).
- Protección: contra interferencias no deseadas por parte de otros procesos, ni un proceso acceder a partes asignadas al SO.
- Posibilidad de compartir: que los procesos en memoria puedan compartir, código o datos comunes a ellos, manteniendo la integridad de la información y sin comprometer la protección.
- Organización lógica: permitir el uso de técnicas que referencien no direcciones reales, sino una abstracción mas cercana al usuario, como pueden ser los módulos. Por ejemplo, la segmentación (que veremos mas adelante).
- Organización física: Permitir una organización como mínimo de dos niveles (memoria real y memoria secundaria) que permitan la rápida ejecución de

los procesos por un lado, y el almacenamiento a largo plazo de programas y datos.

Analizando la evolución de los diferentes esquemas de administración de memoria, veremos que es semejante a la evolución de la administración del espacio en disco para la asignación de bloques a los archivos.

3.2. Modelo de memoria de un proceso

El sistema operativo gestiona el mapa de memoria de un proceso durante la vida del mismo. Dado que el mapa inicial de un proceso está muy vinculado con el archivo que contiene el programa ejecutable asociado al mismo, comenzaremos estudiando cómo se genera un archivo ejecutable y cuál es la estructura típica del mismo, cómo evoluciona el mapa a partir de ese estado inicial y qué tipos de regiones existen típicamente en el mismo identificando cuáles son sus características básicas.

3.2.1. Fases en la generación de un ejecutable

En general, una aplicación estará compuesta por un conjunto de módulos de código fuente que deberán ser procesados para obtener el ejecutable de la aplicación. Como se puede observar en la figura 3.1, este procesamiento típicamente consta de dos fases: compilación, que genera el código máquina correspondiente a cada módulo fuente de la aplicación, y enlace (*link*), que genera un ejecutable agrupando todos los archivos objeto y resolviendo las referencias entre módulos.

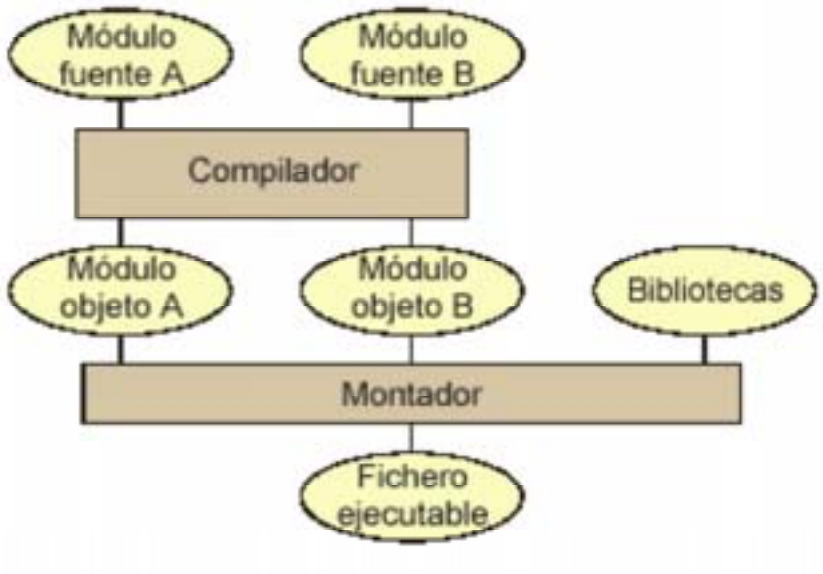

Figura 3.1: Fases en la generación de un ejecutable

Además de referencias entre módulos, pueden existir referencias a símbolos definidos en otros archivos objeto previamente compilados agrupados nor-

malmente en bibliotecas.

Una biblioteca es una colección de objetos normalmente relacionados entre sí. La manera de generar el ejecutable comentado hasta ahora consiste en compilar los módulos fuente de la aplicación y enlazar los módulos objeto resultantes junto con los extraídos de las bibliotecas correspondientes. De esta forma, se podría decir que el ejecutable es autocontenido: incluye todo el código que necesita el programa para poder ejecutarse. Existe, sin embargo, una alternativa que presenta numerosas ventajas: las bibliotecas dinámicas.

Con este nuevo mecanismo, el proceso de enlace de una biblioteca de este tipo se difiere y, en vez de realizarlo en la fase de enlace, se realiza en tiempo de ejecución del programa. Cuando en la fase de enlace el *linker* procesa una biblioteca dinámica, no incluye en el ejecutable código extraído de la misma, sino que simplemente anota en el ejecutable el nombre de la biblioteca para que ésta sea cargada y enlazada en tiempo de ejecución. El uso de bibliotecas dinámicas presenta múltiples ventajas: se reduce el tamaño de los ejecutables, se favorece el compartimiento de información y se facilita la actualización de la biblioteca.

La forma habitual de usar las bibliotecas dinámicas consiste en especificar al momento de enlace (*link time*) qué bibliotecas se deben usar, pero la carga y el enlace se pospone hasta el momento de ejecución (*run time*). Sin embargo, también es posible especificar al momento de ejecución qué biblioteca dinámica se necesita y solicitar explícitamente su enlace y carga (a esta técnica se la suele llamar *carga explícita de bibliotecas dinámicas*).

3.2.2. Formato del ejecutable

Como parte final del proceso de compilación y enlace, se genera un archivo ejecutable que contiene el código máquina del programa. Como se puede observar en la figura 3.2, un ejecutable está estructurado como una cabecera y un conjunto de secciones.

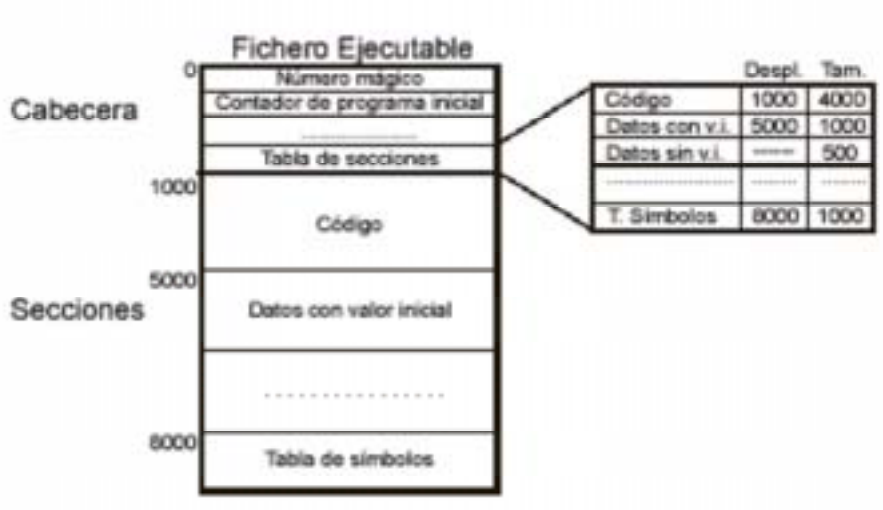

Figura 3.2: Formato simplificado de un ejecutable

La cabecera contiene información de control que permite interpretar el contenido del ejecutable. En cuanto a las secciones, cada ejecutable tiene un conjunto de secciones; típicamente, aparecen al menos las tres siguientes:

- Código (texto): contiene el código del programa.
- Datos con valor inicial: almacena el valor inicial de todas las variables globales a las que se les ha asignado un valor inicial en el programa.
- Datos sin valor inicial: Se corresponde con todas las variables globales a las que no se les ha dado un valor inicial. Como se muestra en la figura, este apartado aparece descrito en la tabla de secciones de la cabecera, pero, sin embargo, no se almacena normalmente en el ejecutable, ya que el contenido de la misma es irrelevante.

3.2.3. Mapa de memoria de un proceso

El mapa de memoria de un proceso no es algo homogéneo, sino que está formado por distintas regiones o segmentos. Cuando se activa la ejecución de un programa, se crean varias regiones dentro del mapa a partir de la información del ejecutable. Las regiones iniciales del proceso se van a corresponder básicamente con las distintas secciones del ejecutable.

Cada región es una zona contigua que está carcterizada por la dirección dentro del mapa del proceso donde comienza y por su tamaño. Además, tendrá asociada una serie de propiedades y características específicas, tales como las siguientes:

- Soporte de la región, donde está almacenado el contenido inicial de la región. Se presentan normalmente dos posibilidades:
  - Soporte en archivo: Está almacenado en un archivo o en parte del mismo.
  - Sin soporte: No tiene un contenido inicial.
- Tipo de uso compartido:
  - Privada: El contenido de la región sólo es accesible al proceso que la contiene. Las modificaciones sobre la región no se reflejan permanentemente.
  - Compartida: El contenido de la región puede ser compartido por varios procesos. Las modificaciones en el contenido de la región se reflejan permanentemente.
- Protección: Tipo de acceso a la región permitido. Típicamente, se proporcionan tres tipos:
  - Lectura: Se permiten accesos de lectura de operandos de instrucciones.
  - Ejecución: Se permiten accesos de lectura de instrucciones.
  - Escritura: Se permiten 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 básicamente con las secciones del ejecutable más la pila inicial del proceso, a saber:

- Código (texto): Se trata de una región compartida de lectura/ejecución. Es de tamaño fijo. El soporte de esta región está en el apartado correspondiente del ejecutable.
- Datos con valor inicial: Se trata de una región privada, ya que cada proceso que ejecuta un determinado programa necesita una copia propia de las variables del mismo. Es de lectura/escritura y de tamaño fijo. El soporte de esta región está en el apartado correspondiente del ejecutable.
- Datos sin valor inicial: Se trata de una región privada, de lectura/escritura y de tamaño fijo (el indicado en la cabecera del ejecutable). Como se comentó previamente, esta región no tiene soporte en el ejecutable, ya que su contenido inicial es irrelevante.
- Pila: Esta región es privada y de lectura/escritura. Servirá de soporte para almacenar los registros de activación de las llamadas a funciones (las variables locales, parámetros, direcciones de retorno, etc.) Se trata, por tanto, de una región de tamaño variable que crecerá cuando se produzcan llamadas a funciones y decrecerá cuando se retorne de las mismas. Típicamente, esta región crece hacia las direcciones más bajas del mapa de memoria. En el mapa inicial existe ya esta región que contiene típicamente los argumentos especificados en la invocación del programa.

Los sistemas operativos 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 relacionadas con otros aspectos, tales como los siguientes:

- Heap (montículo): La mayoría de los lenguajes de alto nivel ofrecen la posibilidad de reservar espacio en tiempo de ejecución. En el caso del lenguaje C, se usa la función "malloc" para ello. Esta región sirve de soporte para la memoria dinámica que reserva un programa en tiempo de ejecución. Comienza, típicamente, justo después de la región de datos sin valor inicial (de hecho, en algunos sistemas se considera parte de la misma) y crece en sentido contrario a la pila (hacia direcciones crecientes). Se trata de una región privada de lectura/escritura, sin soporte (se rellena inicialmente a cero), que crece según el programa vaya reservando memoria dinámica y decrece según la vaya liberando.
- Archivos proyectados: Cuando se proyecta un archivo, se crea una región asociada al mismo.

- Memoria compartida: Cuando se crea una zona de memoria compartida y se proyecta, se crea una región asociada a la misma. Se trata, evidentemente, de una región de carácter compartido cuya protección la especifica el programa a la hora de proyectarla.
- Pilas de *threads*: Cada *thread* necesita una pila propia que normalmente se corresponde con una nueva región en el mapa. Este tipo de región tiene las mismas características que la región correspondiente a la pila del proceso.

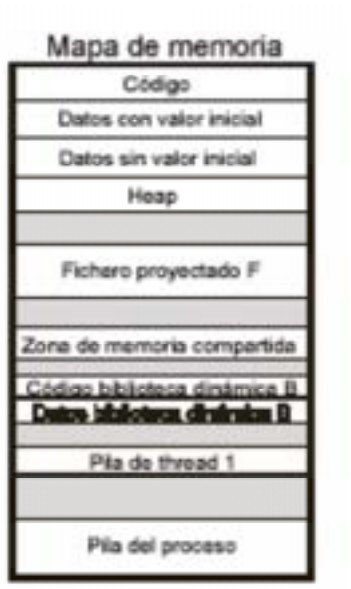

Figura 3.3: Mapa de memoria de un proceso hipotético

En la figura 3.3 se muestra un hipotético mapa de memoria que contiene algunos de los tipos de regiones comentados en este apartado.

Como puede apreciarse en la figura anterior, la carga de una biblioteca dinámica implicará la creación de un conjunto de regiones asociadas a la misma que contendrán las distintas secciones de la biblioteca (código y datos globales). Para visualizar estos conceptos pase a la subsección 3.7.1 de la práctica.

3.3. Diferentes esquemas de administración

3.3.1. Monoprogramación

3.3.1.1. Partición absoluta única

El esquema de administración de memoria más simple no requiere soporte de hardware. El espacio de memoria está dividido por convención en dos particiones: una partición está asignada al sistema operativo y la otra a un proceso en ejecución. Los así llamados programas de "buen comportamiento" se limitan a acceder las ubicaciones de memoria en la partición del proceso. Sin soporte de

hardware, sin embargo, no hay nada que prevenga que los programas corrompan el sistema operativo. Los primeros sistemas DOS usaban una variante de este esquema, que permitían que los programas de usuarios con "mal comportamiento" pudieran quebrar el sistema completo (lo que frecuentemente hacían).

Cuando un programa se carga en una partición, las direcciones en el código cargado deben corresponder con las direcciones apropiadas en la partición. Las direcciones en el código pueden estar *limitadas* a direcciones de memoria en particular ya sea al momento de compilación o al momento de la carga. Los programas que han sido limitados a ubicaciones de memoria al momento de compilación se dice que contienen *código absoluto*. Si esta ligazón (*binding*) ocurre cuando el programa se carga en memoria, se dice que el programa contiene *código reubicable*. Para que un programa contenga código reubicable, debe existir algún mecanismo para identificar los bytes en el programa que se refieren a direcciones de memoria. A pesar de que esto agrega complejidad a los procesos de compilación y carga, tiene la ventaja obvia de liberar al programa de ser cargado en cualquier ubicación disponible de la memoria.

Se le puede incorporar protección a un esquema de partición absoluta agregándole al hardware un *registro base*. El sistema operativo carga el registro base con la dirección más baja accesible por un programa de usuario. El hardware luego compara cada dirección generada por el programa de usuario con el contenido del registro base. Las direcciones menores que la dirección almacenada en el registro base provocan una trampa de fallo de memoria (*memory-fault trap*) en el sistema operativo.

3.3.1.2. Partición reubicable única

Mucho más útiles que los sistemas de registro base son los sistemas que contienen un *registro de reubicación (relocation register)*. Como con el registro base, el registro de reubicación es cargado por el sistema operativo con la dirección de comienzo del proceso. Pero en vez de comparar el contenido del registro de reubicación con la dirección generada por el proceso, sus contenidos se suman. El programa ahora opera en un espacio de direcciones *lógico*. Es compilado como si fuera a ser asignado a la memoria que comienza en la ubicación 0. El hardware de administración de memoria convierte las direcciones lógicas en las direcciones *físicas* verdaderas.

3.3.2. Multiprogramación

En un ambiente de multiprogramación hay varios procesos compartiendo la memoria. Por lo tanto, la protección se debe intensificar: proteger los espacios de los procesos entre si, y con respecto a la memoria del sistema.

Los esquemas de partición única limitan a una computadora a ejecutar un programa a la vez. Tales sistemas de monoprogramación en la actualidad son raros de encontrar (excepto las consolas de juegos) pero eran comunes en las primeras computadoras. Para cargar múltiples procesos, el sistema operativo debe dividir la memoria en múltiples particiones para esos procesos.

En un esquema de partición múltiple, se crean múltiples particiones para permitir que múltiples procesos de usuario queden residentes en la memoria en forma simultánea. Un segundo registro de hardware, para usarlo en conjunto con el registro de reubicación, marca el final de una partición. Este registro puede

contener ya sea el tamaño de la partición o la última dirección en la partición. Típicamente, el registro contiene el tamaño de la partición y se lo refiere como el *registro tamaño* o el *registro límite*.

3.3.2.1. Múltiples particiones fijas

Si todas las particiones son del mismo tamaño, el sistema operativo sólo necesita llevar cuenta de cuales particiones están asignadas a cada proceso. La tabla de particiones de memoria almacena o bien la dirección de comienzo para cada proceso o el número de la partición asignada a cada proceso. Si está almacenado el número de partición y el tamaño de la partición es potencia de 2, la dirección de comienzo puede ser generada al concatenar el número apropiado de ceros al número de partición. El registro límite se establece al momento del arranque y contiene el tamaño de la partición. Cada vez que a un proceso se le asigna control de la CPU, el sistema operativo debe restablecer el registro de reubicación.

El espacio al final de una partición que no es usado por un proceso, se desperdicia. Al espacio desperdiciado dentro de una partición se le llama ***fragmentación interna***. Un método para reducir la fragmentación interna es usar particiones de diferentes tamaños [2]. Si se hace esto, el sistema operativo también debe restablecer el registro límite cada vez que un proceso es asignado a la CPU. El tamaño de la partición puede estar almacenado en la tabla de particiones de la memoria o bien puede deducirse a partir del número de partición.

El tener particiones de diferente tamaño complica la administración de procesos. Cada vez que a un proceso diferente se le da control de la CPU, el sistema operativo debe restablecer el registro de tamaño además del registro de reubicación. El sistema operativo también debe tomar decisiones acerca de en cual partición debería asignar a un proceso. Obviamente, un proceso no puede ser asignado a una partición menor que el tamaño del proceso. Pero ¿debería un proceso más pequeño ser asignado a una partición mas grande si cabría en una partición más pequeña, aun si la partición más pequeña actualmente está ocupada por otro proceso?

3.3.2.2. Múltiples particiones variables

En vez de dividir la memoria en un conjunto fijo de particiones, un sistema operativo puede elegir ubicar procesos en cualquier ubicación de memoria que esté sin usar. La cantidad de espacio asignado a un proceso es la cantidad exacta de espacio que requiere, eliminando la fragmentación interna. Sin embargo, el sistema operativo debe manejar mas datos. Se debe almacenar la ubicación exacta de comienzo y finalización de cada proceso, y se debe mantener los datos acerca de cuales ubicaciones de memoria están libres.

A medida que los procesos se van creando y terminando, el uso de la memoria evoluciona hacia secciones alternadas de espacio asignado y sin asignar, como un tablero de ajedrez (***checkerboarding***). Como resultado, a pesar de que puede haber mas memoria sin asignar que el tamaño de un proceso en espera, esta memoria no puede ser usada por un proceso porque está dispersa entre un número de ***huecos*** de memoria. Este espacio desperdiciado no asignado a ninguna partición se le llama ***fragmentación externa***.

[2] El grado de multiprogramación queda definido por la cantidad de particiones.

Se puede usar un mapa de bits para mantener un control de qué memoria ha sido asignada. La memoria está dividida en unidades de asignación y cada bit en el mapa indica si está asignada o no la unidad correspondiente. El incrementar el tamaño de la unidad de asignacion decrece el tamaño del mapa de bits, pero incrementa la cantidad de memoria desperdiciada cuando el tamaño del proceso no es un múltiplo del tamaño de la unidad de asignación.

También se puede usar una lista enlazada para mantener un control de la memoria libre. Cada agujero contendrá una entrada indicando el tamaño del agujero y un puntero al próximo agujero en la lista. El sistema operativo sólo necesita un puntero al primer agujero en la lista. Esta lista puede mantenerse por orden de memoria, orden de tamaño o en ningún orden en particular.

Se puede hacer uso de la ***compactación*** para hacer un uso más eficiente de la memoria. Al mover procesos en memoria, los huecos en la memoria se pueden recolectar en una seccion única de espacio sin asignar. La cuestión es si la sobrecarga extra que toma mover los procesos justifica la mayor eficiencia ganada al hacer mejor del espacio de memoria.

Algoritmos de selección de la partición: En situaciones en las que múltiples agujeros de memoria son suficientemente grandes como para contener un proceso, el sistema operativo debe usar un algoritmo para seleccionar en qué agujero se cargará el proceso. Se han estudiado y propuesto una variedad de algoritmos

- **Primer ajuste** (*first fit*): El sistema operativo revisa en todas las secciones de memoria libre. El proceso es asignado al primer hueco encontrado que sea mayor que el tamaño del proceso. A menos que el tamaño del proceso coincida con el tamaño del hueco, el agujero continúa existiendo, reducido por el tamaño del proceso.
- **Próximo ajuste** (*next fit*): En el primer ajuste, ya que todas las búsquedas empiezan al comienzo de la memoria, siempre se ocupan más frecuentemente los huecos que están al comienzo de la memoria que los que están al final. El "próximo ajuste" intenta mejorar el desempeño al distribuir sus búsquedas mas uniformemente sobre todo el espacio de memoria. Hace esto manteniendo el registro de qué hueco fue el último en ser asignado. La próxima búsqueda comienza en el último hueco asignado, no en el comienzo de la memoria.
- **Mejor ajuste** (*best fit*): El algoritmo del mejor ajuste revisa la lista completa de huecos para encontrar el hueco mas pequeño cuyo tamaño es mayor o igual que el tamaño del proceso.
- **Peor ajuste** (*worst fit*): En situaciones en las que el mejor ajuste encuentra una coincidencia casi perfecta, el hueco que queda es virtualmente inútil porque es demasiado pequeño. Para prevenir la creación de estos huecos inútiles, el algoritmo del peor ajuste trabaja de manera opuesta al del mejor ajuste; siempre elije el que deje el hueco remanente más grande.

First fit es la solución mas rápida. First fit y Best fit hacen un mejor aprovechamiento de la memoria que Worst fit.

El otro punto es como se organizan los procesos para ser asignados. El encolamiento se puede definir entre estos dos modelos:

Cola de procesos por partición: se encolan los procesos de acuerdo a la partición asignada, normalmente por tamaño. La ventaja es el aprovechamiento de los espacios de memoria. La desventaja es que puede haber encolamiento para una partición y que el resto de las particiones estén ociosas.

Cola única: Los procesos a la espera de ser cargados se encolan en una cola única y un modulo administra el uso de todas las particiones. Si hay una partición justa para ese proceso se asigna, si no hay de su tamaño pero hay mayor, se asigna la mayor. También puede influir en la decisión de la carga la prioridad del proceso.

3.3.2.3. Sistema de compañeras

Un sistema de compañeras (*buddy system*) un compromiso entre la asignación de tamaño fijo y la de tamaño variable. La memoria se asigna en unidades que son potencia de 2. Inicialmente hay una única unidad de asignación que comprende a toda la memoria. Cuando se le debe asignar memoria a un proceso, se le asigna una unidad de memoria cuyo tamaño es la menor potencia de 2 mayor que el tamaño del proceso. Por ejemplo, a un proceso de 50K se lo ubicará en una unidad de asignación de 64K. Si no existe una unidad de asignación de ese tamaño, la asignación disponible mas chica mayor que el proceso se dividirá en dos unidades "compañeras" de la mitad del tamaño que el original. La división continúa con una de las unidades compañeras hasta que se crea una unidad de asignación del tamaño apropiado. Cuando un proceso libera memoria provoca que se liberen dos unidades compañeras, las unidades se combinan para formar una unidad dos veces más grande. La figura XX muestra cómo las unidades de asignación se dividirán y se juntarán a medida que la memoria es asignada y liberada.

Dado un tamaño de memoria de 2^N, un sistema de compañeras mantiene un máximo de N listas de bloques libres, una para cada tamaño, (dado un tamaño de memoria de 2^8, son posibles unidades de asignación de tamaño 2^8 hasta 2^1). Con una lista separada de bloques disponibles para cada tamaño, la asignación o liberación de memoria puede ser procesada de forma eficiente. Sin embargo, el sistema de compañeras no usa la memoria de una forma eficiente a causa de tener tanto fragmentación interna como externa.

3.3.2.4. División de la memoria en particiones dinámicas

Las particiones son variables en número y longitud. Cuando se carga un proceso en memoria se le asigna toda la memoria que necesita, quedando bien delimitado su espacio de direcciones (no olvidemos que dentro de ese espacio esta su stack, de crecimiento dinámico).

A continuación se cargará otro proceso, y así sucesivamente hasta que no haya mas lugar. Cuando alguno de estos procesos termina y libera la memoria utilizada, ese espacio podrá ser usado por un proceso de igual o menor tamaño. No obstante a medida que se liberan y asignan espacios se puede dar la siguiente situación:

- que la memoria quede fragmentada en pequeños huecos que no sean lo suficientemente grandes como para poder asignarlos a los procesos listos;

- que la suma de estos fragmentos resulte en un espacio que sí pueda ser utilizado, pero no disponible por no ser contiguo.

A esta situación se le llama *fragmentación externa*. La forma de solucionarla puede ser la compactación de la memoria, desplazando los procesos para que estén contiguos dejando los espacios libres juntos en un bloque.

Los algoritmos de ubicación para este sistema son los ya vistos: First fit, best fit y worst fit, entre los huecos disponibles.

Puede darse la situación que en un momento dado estén bloqueados todos los procesos en memoria principal, y el espacio libre sea insuficiente para la carga de un proceso nuevo. En este caso el sistema operativo puede tomar la decisión de expulsar alguno de los procesos en memoria para dar lugar a un proceso nuevo, o listo pero suspendido. Deberá entonces elegir cual de los procesos sacará de memoria.

3.3.2.5. Superposiciones (Overlays)

Hace mucho tiempo se enfrentaban al problema de tener programas más grandes que la memoria. En ese momento se optó por dividir el programa en partes a las que se llamaba superposiciones (*overlays*).

Una superposición mantiene en memoria solo aquellas instrucciones y datos necesarios en un momento dado. Se definen previamente a la ejecución varias superposiciones, que estarán en diferentes momentos en memoria, reemplazando a la anterior, cuando el proceso lo requiera. Comienza a ejecutarse la superposición 0, al terminar se descarga y se carga la 1, y así sucesivamente cargando y descargando.

El S.O. se encargaba de la carga y descarga pero el trabajo del programador era tedioso pues debía descomponer su programa en módulos pequeños para crear las superposiciones.

En el libro de Silberschatz [Silberschatz y Galvin, 1999] está el ejemplo del Cuadro 3.3, bastante claro.

Supongamos un ensamblador de dos pasos.

En el Paso 1, construye una tabla de símbolos. En el paso 2, genera el código de máquina.

Por lo tanto, todos los elementos que intervienen y sus medidas son:

Código del paso 1	70k
Código del paso 2	80k
Tabla de símbolos	20k
Rutinas comunes a ambos pasos	30k
En total	200k

Cuadro 3.3: Superposiciones (*overlays*)

Supongamos que hay 150K disponibles.

Se construyen 2 overlays. Uno contiene Código del paso 1, Tabla de símbolos y rutinas comunes, es decir, todo lo necesario para ejecutar el paso1. Luego se descarga este overlay y se carga el otro overlay que contiene Código del paso 2, Tabla de símbolos y rutinas comunes.

No debemos olvidar cargar un driver de overlays antes de cargar el primer overlay. Una vez ejecutado este overlay, será el driver el encargado de leer el segundo overlay a memoria, sobreescribiendo el primero.

Hay que considerar que esta técnica aporta E/S adicional en la lectura de los overlays pues estos residen en disco como imágenes absolutas de memoria. Son implementados por el programador sin soporte especial por parte del SO.

Sobre las direcciones En el esquema de particiones fijas de dimensión variable, podemos pensar que un proceso es cargado siempre en la misma partición. Si se saca temporariamente de memoria, al volverse a cargar lo hará en la misma partición. Pero cuando las particiones son fijas, el proceso al descargarse puede luego volver a cargarse en cualquiera. Pero para ello es importante definir el momento en que se hace la resolución o vinculación de direcciones (*binding*) es decir, cuando se pasa de la dirección lógica a la física.

Normalmente un programa reside en disco como un archivo binario ejecutable. Cuando se quiere ejecutar se transforma en un proceso que debe ser cargado en memoria para su ejecución.

Las direcciones de un proceso pueden ser lógicas, reubicables o físicas. Las lógicas y las reubicables necesitan una transformación para convertirse en físicas o reales. Es lo que llamamos *binding*.

Hay tres momentos donde puede realizarse el *binding*: en la compilación, en la carga y en la ejecución.

- Compilación: en este momento se generan las direcciones absolutas del programa, lo que exige que cada vez que se ejecute se deba cargar en el mismo lugar de memoria. Si se quiere cambiar de dirección, se debe recompilar para adaptarlo.
- Carga: si no se sabe en que lugar de memoria se ubicará el proceso, se generarán en la compilación direcciones reubicables, que se resolverán cuando se conozca la dirección de inicio de carga del proceso. Allí se hace el binding, exigiendo que de ahí en mas, el proceso se ejecute siempre en el mismo espacio hasta el final de su ejecución.
- Ejecución: si el proceso será movido de memoria por descarga temporal, compactación o técnicas de memoria virtual, se retrasa el binding hasta el momento de ejecución. Para esto es necesario contar con un hardware especial.

En este último caso, normalmente llamamos a la dirección lógica como dirección virtual.

El conjunto de direcciones lógicas de un proceso forman su espacio de direcciones lógicas, mientras que las físicas constituyen el espacio de direcciones físicas.

Estos espacios difieren en un esquema de binding en momento de ejecución.

La transformación la realiza la Unidad de Manejo de Memoria (MMU), que es un dispositivo de hardware. En este hardware, el registro base es el registro de reubicación, y su contenido es agregado a cada dirección generada por el proceso de usuario en el momento de la carga a memoria. El usuario nunca ve direcciones reales.

3.3.2.6. Carga dinámica (dynamic loading)

Para poder ejecutar un proceso éste debe estar en memoria. No obstante hay partes de ese proceso que es probable que no se ejecuten, por ejemplo, una rutina de errores. En este caso, si la hubiéramos cargado en memoria, sería espacio desaprovechado. ¿Por que no cargar las rutinas cuando el proceso en ejecución lo requiera, es decir, postergar su carga hasta asegurarse que el código a cargar es necesario?

Para poder realizar esto es imprescindible que esa rutina se mantenga en disco en formato reubicable para poder ser cargada cuando se la requiera.

Cuando el proceso invoca a una rutina se chequea si ya está en memoria. Si no está, el cargador de reubicables es invocado para cargarla a memoria y modificar las tablas activas de memoria para indicar este cambio. Luego se pasa el control a la rutina. Este esquema es llamado carga dinámica (*dynamic loading*). No requiere un soporte especial del SO, salvo ayudar al programador proveyendo la biblioteca (*library*) de rutinas para implementar carga dinámica.

3.3.2.7. Encadenamiento o enlace dinámico (dynamic linking)

La mayoría de los SO proveen enlace estático (*static linking*) donde las bibliotecas (mal llamadas "librerías") del lenguaje del sistema se combinan en la imagen binaria de programa como cualquier otro módulo objeto. Vimos en el punto anterior la posibilidad de postergar la carga de un módulo hasta el momento en que lo requiera el proceso en ejecución y eso es la carga dinámica. El encadenamiento dinámico, en cambio, es postergar el encadenamiento de módulos de bibliotecas (librerías) hasta el momento de la ejecución.

Cuando no hay enlace dinámico, como los módulos de las bibliotecas de lenguaje son muy utilizados, cada proceso debería tener "insertado" en él un módulo que es muy probable que también tenga otro proceso cargado en memoria. Tenemos entonces código que se repite en memoria. ¿Por qué no cargarlo una sola vez y ser compartido entre los diferentes procesos? ¿Por qué no postergar su carga hasta el momento que es invocada la rutina por primera vez?

En el enlace dinámico, se incluye en la imagen del proceso un fragmento (*stub*) donde se hace referencia a una rutina de lenguaje. Este pequeño trozo de código indica dónde se ubica esa rutina y cómo cargarla si no está en memoria en el momento que se la requiera.

Cuando se ejecute ese fragmento (*stub*), si la rutina no está en memoria, se carga y la próxima vez que se invoque ya estará en memoria. Si está, el fragmento (*stub*) es reemplazado por la dirección de memoria de la rutina.

De esta manera hay una sola copia de las rutinas de lenguaje en memoria. Esta técnica es muy útil cuando hay nuevas versiones de bibliotecas, por mejorados o errores de la versión previa.

Si hay enlace dinámico, entonces cuando se invoque una rutina, automáticamente referenciará a la nueva versión, sin necesidad de tener que volver a "linkear" los módulos. Hay SO's que soportan bibliotecas compartidas (*shared libraries*). Esto significa que puede optarse por mantener las dos versiones de biblioteca en memoria: la vieja y la nueva.

Si los programas harán referencia a la biblioteca que quieren usar. Para ello se debe incluir en el momento del linking a qué versión harán referencia.

El enlace dinámico (*dynamic linking*), a diferencia de la carga dinámica

(*dynamic loading*), sí requiere ayuda del SO, pues puede ocurrir que varios procesos hagan referencia al mismo módulo de biblioteca, lo que debe manejarse con cuidado para que no haya invasión del espacio de direcciones de los procesos involucrados.

3.3.2.8. Swapping o intercambio

Puede ocurrir que el SO decida que un proceso en memoria salga temporariamente, porque necesita mas espacio o por que llega un proceso de mayor prioridad. La técnica de llevar temporariamente un proceso a memoria secundaria se llama intercambio o *swapping*. Y el espacio en disco donde se almacena, se llama le solía llamar *backing store*.

Cuando se saca de memoria, se hace *swap-out*; cuando se trae nuevamente, *swap-in*.

En el caso de un algoritmo de planificación de CPU basado en prioridades cuando llega un proceso de mayor prioridad el manejador de memoria puede hacer *swap-out* a un proceso de menor prioridad para poder cargar el de mayor. Cuando termina, se vuelve a cargar el de menor prioridad. A esta variedad se la llama *roll-out/roll-in*.

El lugar donde se volverá a ubicar el proceso descargado depende del tipo de *binding* que haga: solo puede cambiar de lugar si tiene *binding* en el momento de ejecución. *Backing storage* es normalmente un disco rápido y con suficiente capacidad para mantener las imágenes de los procesos de los usuarios y debe proveer acceso directo a estas imágenes.

El sistema mantiene una cola de listos con aquellos procesos cuyas imágenes están listas para ejecutar en *backing store* o en memoria. Según vimos, cuando la CPU quiere ejecutar un proceso llama al *dispatcher* que chequea si el próximo proceso en la cola está en memoria. Si no está y no hay memoria disponible, el dispatcher decidirá hacer *swap-out* de un proceso de memoria para hacer *swap-in* del proceso deseado. Luego actúa como siempre, transfiriendo el control al nuevo proceso.

El proceso de *swapping*, que involucra cambios de contextos (*context switch*) debe ser muy rápido. El tiempo de ejecución de un proceso debe ser mucho mayor que el tiempo que toma hacer *swapping*. En un sistema con algoritmo de CPU round-robin, el quantum deber ser mucho mayor que el tiempo de *swapping*.

El tiempo de *swapping* involucra el tiempo de transferencia hacia/desde disco, y por lo tanto, directamente proporcional a la cantidad de memoria que se transfiere.

Para "intercambiar" un proceso éste debe estar ocioso.

Otro tema es qué hacer con los procesos con Entrada/Salida pendiente, pues cuando termine la operación de Entrada/Salida (I/O) puede ser que el proceso ya no esté en memoria y se quiera ubicar la información en un espacio que ahora pertenece a otro proceso. Las dos soluciones probables son:

- no intercambiar procesos con I/O pendiente.
- utilizar para la I/O buffers del sistema operativo para la información y pasarla a memoria de usuario cuando el proceso vuelva a ser cargado en memoria.

En pocos sistemas se usa el swapping estándar. Normalmente se utilizan variaciones: algunos sistemas habilitan el swapping cuando lo necesitan y lo deshabilitan si ya no es necesario; puede ser que la carga o descarga la decida el usuario en vez del planificador.

3.4. Memoria virtual

Las alternativas de administración de memoria vistas cargan todo el proceso de manera contigua. Esto exige mayores espacios de memoria con secciones del proceso que no se utilizarán enseguida, pues la ejecución de un proceso es secuencial y se adapta al modelo de localidad. Esto significa que en un período de tiempo se usarán direcciones de memoria "cercanas". ¿Por que no tener en memoria solo la parte del proceso que se está ejecutando? Estas partes serían de menor tamaño y por lo tanto, se podría aprovechar mejor la memoria. Es más: la medida del proceso o de la suma de los procesos listos puede ser mayor que la memoria disponible, pues no la ocupan toda a la vez.

En el año 1961, Fotheringham, crea esta técnica de administración que se llamó memoria virtual [Fotheringham, 1961]. La idea básica es que el tamaño del programa, los datos y la pila combinados pueden ser mayores que la memoria disponible. El S.O. guarda aquellas partes del programa de uso corriente en la memoria principal y el resto, en disco. La memoria virtual y la multiprogramación se corresponden bien entre sí pues, por ejemplo, si tengo 8 programas de 1MB puedo asignar una partición de 256 KB en una memoria de 2MB. Cada programa trabaja como si tuviera una máquina privada de 256 KB. Además, la memoria virtual es útil pues mientras un programa hace swapping, otro puede tener el procesador y de esa manera tengo menos CPU ociosa.

3.4.1. Paginación

Las direcciones que puede generar un programa pueden ser por indexación, relativo al registro base, etc. Las direcciones virtuales forman parte del espacio de direcciones virtuales del proceso. Cuando no hay memoria virtual, la dirección generada por el proceso se coloca directamente en el bus de memoria. Cuando hay memoria virtual, la dirección pasa al MMU (unidad de administración de memoria, hardware). Este es un chip o un conjunto de ellos que transforman las direcciones virtuales en direcciones reales de memoria.

La paginación consiste en dividir el proceso (memoria virtual) en páginas. La memoria física se divide en *frames* o marcos. Las páginas son cargadas en memoria a medida que se van necesitando y en espacio no contiguo: basta que haya un marco libre en memoria para cargar la página.

La ventaja de la paginación reside en la falta de fragmentación interna y la posibilidad de almacenar las páginas de un proceso de manera no contigua. Cada proceso tiene una tabla de páginas con una entrada por página indicando en qué marco está, y un conjunto de bits de control que dan información de acceso y modificación de la página mientras estuvo en memoria.

Pero: ¿es necesario tener todas las páginas en memoria? Si una página se usó pocas veces, ¿es necesario que siga ocupando lugar? Si considero la posibilidad de tener sólo las páginas que necesito, intercambiando páginas "viejas" por páginas "nuevas", puedo llegar al ideal de tener procesos más grandes que la memoria

disponible. Cada página que necesita el proceso, la "demanda" al SO. De ahí el nombre de paginación por demanda.

Supongamos que tenemos una computadora de 32KB de memoria física y que puede generar direcciones de 16 bits, de 0 a 64K. Estas serían las direcciones virtuales. Las páginas y los marcos deben ser de igual tamaño. En nuestro ejemplo serán de 4K. Por lo tanto tengo 16 páginas virtuales y 8 cuadros de página.

Tabla páginas	
0-4K	2
4K-8K	1
12	6
16	0
20	4
24	3
28	
32	
36	
40	5
44	
48	7
52	
56	
60	
64	

Marcos mem real
0-4K
4-8
8
12
16
20
24
28

El programa quiere acceder a la dirección 0 por una instrucción por ejemplo `move reg,0`.

La dirección 0 se manda a la MMU que observa que la dirección virtual 0 queda en la página 0 (0-4095) y que el cuadro de página es 2 (8192-12387).

Por lo tanto transforma la dirección en 8192 y la pone en el bus. La memoria advierte que hay una solicitud de memoria a la dirección 8192. La instrucción `move reg, 8192` se transforma en `move reg,24576` pues 8192 está en la página 2 (virtual) que se transforma en la 24576 pues apunta al cuadro de página 6. En un tercer ejemplo, la dirección virtual 20500 tiene 20 bytes desde el inicio de la página virtual 5 (20480-24575), por lo tanto se transforma en 12288 · 20=12308 (la página virtual 5 apunta al cuadro de página 3).

Cuando un proceso hace referencia a una dirección que no está en esa página, el SO debe determinar si está en memoria y, si es así, en qué página. Si no está, se debe cargar en memoria. Si no hay lugar se debe elegir una página "víctima" que se sacará temporariamente de memoria para cargar la que se necesita.

Consideremos que en cada entrada de la tabla de páginas un bit representa la presencia/ausencia, o sea, si la página está cargada o no en un *frame*.

Si el programa intenta acceder a una página que no está mapeada se produce un *page fault* o falla de página. El sistema operativo toma una página con poco uso y, si fue modificada mientras estuvo en memoria, escribe su contenido al disco. Carga la página recién referida en el marco recién liberado, cambia la tabla de páginas y se reinicia la ejecución[3].

[3]En algunos sistemas se guardan frames libres para que, ante un page fault, se cargue la página que se necesita en uno de esos *frames*, mientras se va descargando la página víctima. Luego el *frame* de la página víctima, queda como libre. De esta manera el proceso no espera

Veamos como se transforma una dirección virtual de 16 bits en una física. Supongamos la dirección 8196

0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
N de pag virt				Desplazamiento											

En la tabla de páginas de la MMU se accede a la página 2 y de ahí se toma el valor, que es 6 (110) y se forma la dirección:

0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0

Esto ocurre si la página está en memoria, si no, es un *page fault*.

Los 4 primeros bits de la dirección virtual indican la página, y los 12 bits restantes, desplazamiento.

De allí podemos deducir que puede haber 2^4 páginas, y que el tamaño de las páginas será de 2^{12} bytes. La medida de la página la determina el HW, siendo siempre potencia de 2, ver nota[4].

En el único caso en que podemos tener fragmentación interna es en el último *frame*, si la última página del proceso no está completamente llena. El tamaño de la página es una importante decisión: muy chica, reducirá la fragmentación interna del último *frame*, pero habrá mas *page faults*; muy grande reducirá la cantidad de *frames* en memoria y habrá menos *frames* por proceso. Existe también una tabla de *frames* para que el SO pueda saber en todo momento cuales *frames* están ocupados y cuales no, qué procesos tiene *frames* en memoria, etc.

La protección de la memoria en la administración paginada se maneja a través de bits de protección que se asocian a cada página en la tabla de páginas, indicando si es read only, read/write.

También está el bit válido/inválido que indica si la página está en memoria virtual o en real y ayuda para que cuando un proceso referencia una dirección de memoria, se sepa si es un *page fault* o un direccionamiento ilegal.

3.4.1.1. La memoria secundaria en la paginación

La memoria virtual del proceso, es decir, su conjunto de páginas, reside inicialmente en memoria secundaria. A medida que comienzan a cargarse las páginas en memoria, el resto se mantendrá en esta memoria secundaria. El área donde residen estas páginas se llama área de paginado, área de *swap*, o *backing store*. El disco donde se halla, es el *swap device*. Normalmente son discos rápidos, y, en aquellos sistemas que es posible, se elige tener en lugar de un gran disco, dos discos mas chicos pero que permiten balancear la carga, de manera tal de no mezclar las solicitudes de lectura/escritura de páginas con excesivos requerimientos de I/O de usuario.

la descarga de la pagina víctima y la carga de la que necesita: se hacen asincrónicamente.

[4]Si la medida del espacio lógico de direcciones es una potencia m de 2 y la medida de la página en unidades de direccionamiento es una potencia n de 2, los bits de mayor orden m-n de la dirección lógica designan el número de página y los n de bajo orden designan el desplazamiento.

3.4.2. Actividades asociadas al page fault

El procedimiento para manejar un *page fault* es simple. Supongamos que un proceso que se está ejecutando, hace referencia a una dirección de una página que no está en memoria. Esto produce un *trap* o excepción por *page fault*. El HW asociado al paginado al trasladar la dirección física a la lógica notó que el bit asociado a esa página en la tabla de páginas, esta marcado como inválido (la página no está en memoria. Es importante que el SO pueda determinar si es en realidad un *page fault* o simplemente el proceso está haciendo un direccionamiento ilegal. Veamos los pasos para tratar el *page fault*.

1. Se chequea en la tabla interna para ver si es una referencia valida o invalida (si esa dirección pertenece al proceso, o no).
2. Si es inválida, se aborta el proceso. Si es válida, pero aun no está en memoria, se procede a realizar el *page-in*.
3. El SO busca un marco libre (se puede fijar en una lista de *frames* libres).
4. Se hace un requerimiento de lectura de la página a disco, pues está en *backing store*.
5. Una vez que se lee la página solicitada al marco libre, entonces se modifica la tabla para indicar donde está la página y se fijan los bits de control.
6. Se reanuda la instrucción interrumpida por el *trap*, pues la dirección ya está en memoria.

Cuando ocurrió el *trap*, se salvaron los registros y demás información del proceso interrumpido. Considérese la necesidad de volver a ejecutar la instrucción donde ocurrió el *trap*.

En un esquema de paginado por demanda puro, el proceso comienza haciendo *page fault* con la primer instrucción a ejecutar, y así continúa, con una alta tasa de *page fault*, hasta que se estabiliza al tener todas las páginas necesarias en memoria. El momento en que se produce el *page fault* dentro del ciclo de instrucción, es también un punto a considerar, pues la mayoría de las veces se hace necesario volver a ejecutar la instrucción. Si ocurre en el *fetch*, habrá que realizar el *fetch* nuevamente. Si ocurre al tratar de resolver un operando, se deberá rehacer el *fetch* de la instrucción, decodificar de nuevo el operando y entonces, hacer el *fetch* del operando.

3.4.3. El tiempo de acceso a memoria

Representamos el tiempo de acceso efectivo a memoria, T_{ae}, con la siguiente formula:

$T_{ae} = t_{am} \star (1 - p) + t_{pf} \star p$

donde,

t_{am}: tiempo acceso a memoria

p: probabilidad de *page fault*

t_{pf}: tiempo que dura el *page fault*.

Ante un *page fault*, ocurre la siguiente secuencia:

1. Trap al SO.

2. Salvado de los registros y estado del proceso.
3. Resolver si el *trap* es por un *page fault* o mal direccionamiento.
4. Chequear la ubicación de la página en disco.
5. Solicitar una lectura de disco a un *frame* libre, lo que implica:
   a) Esperar en la cola de disco, para servir la lectura.
   b) Esperar el tiempo de ubicación de la cabeza del disco en el lugar indicado (*seek latency time*).
   c) Comenzar a transferir la página al *frame* libre designado.
6. En un ambiente multiprogramado, considerar que la CPU la tiene ahora otro proceso.
7. Interrupción desde disco para indicar que finalizó la transferencia.
8. Si hay otro proceso ejecutándose, realizar el salvado de su información.
9. Determinar que la interrupción vino desde el disco.
10. Actualizar la tabla de páginas y demás estructuras para reflejar que hay una nueva página en memoria.
11. Esperar, en estado de listo, que se le asigne nuevamente CPU.
12. Restaurar la información del proceso para reanudar la ejecución.

Los tres mayores componentes en el servicio a un *page fault* son, entonces:

1. Servir el *trap* de *page fault*
2. Leer la página
3. Reanudar el proceso.

De acuerdo a la formula enunciada arriba, el tiempo de acceso efectivo será directamente proporcional al tiempo de servicio de un *page fault*.

También es muy importante el manejo del espacio de *swap*.

Los bloques del disco de *swap* deben ser mayores que los asignados al *filesystem* de usuario. Otra ventaja sería copiar inicialmente al área de *swap* todo el proceso para tener localizadas allí todas sus páginas (en los casos en que no se puede hacer esto las páginas se demandan desde el sistema de archivos).

Otro punto es el manejo de aquellas páginas que fueron modificadas mientras estuvieron en memoria. Una posibilidades es bajarlas al área de *swap* y al finalizar el proceso, se copia todo al sistema de archivos (no por cada página que se baja modificada).

3.4.4. Selección de la pagina víctima

Cuando ocurre un *page fault*, se hace necesario cargar en memoria la página solicitada. Pero puede ocurrir que no haya marcos libres. Entonces se debe elegir una "víctima" entre las páginas que están en memoria para descargarla y darle su lugar a la nueva página requerida.

Para explicar cómo funcionan los diferentes algoritmos de reemplazo debemos considerar un *string* de referencia de páginas, es decir, de un conjunto de direcciones referenciadas se construye un *string* de las páginas referenciadas por esas direcciones. Una variable que se debe considerar es la cantidad de marcos asignados al proceso (no es lo mismo hacer un análisis para 3 marcos que para 4).

Veamos algunos algoritmos:

- Fifo: es el mas simple, tanto para entender como para programar. Consiste en elegir la página que haga mas tiempo que está en memoria, sin considerar cuando fue la referencia.
- Óptimo: es el que produce menor tasa de *page fault*, pero es teórico pues exige conocer las futuras referencias a página, eligiendo como víctima aquella que no volverá a ser referenciada nunca más o muy posteriormente. Sirve para estudios comparativos, cuando se analizan nuevos algoritmos.
- LRU (usado menos recientemente): Si consideramos que una referencia pasada reciente puede significar una futura y próxima referencia, éste es el algoritmo. Elige como víctima aquella página de entre las que están en memoria que hace mas tiempo que no es referenciada. Es el usado normalmente y el punto es cómo implementarlo. Puede ser a través de variables asociadas a las páginas en la tabla de páginas, con tiempo de acceso que se incrementa en cada *tick*, o haciendo una pila (*stack*) de números de páginas, ordenado por referencia: al ser referenciada una página se pone al tope del *stack* (último quedará el LRU).
- Por bits adicionales de referencia: se asocia a la página un bit de referencia y uno de modificación. De esta manera, elige como página víctima aquella que no haya sido referenciada recientemente y que no haya sido modificada (no se debe hacer la actualización a disco y gano tiempo). Si no hay ninguna en esta condición elige una que no haya sido modificada; si no una que no haya sido referenciada recientemente, etc.

Hay mas algoritmos: 2da oportunidad, LFU (menos frecuentemente usado), MFU (mas frecuentemente usado), etc.

3.4.5. La anomalía de Belady

Se da la situación que, ante una determinada tasa de *page faults*, se decida aumentar la cantidad de *frames* asignados a ese proceso. Repitiendo entonces la experiencia, que intuimos que debería ofrecer una menor tasa de *page faults*, nos encontramos con que esta ha aumentado. A esta situación se le llama anomalía de Belady [Belady, 1969].

El algoritmo óptimo no sufre de esta anomalía; el LRU tampoco. En cambio, puede ocurrir en el FIFO.

3.4.6. Estructura de la tabla de páginas

La mayoría de los SO crean una tabla de página para cada proceso. El puntero a esa tabla como a otros datos propios del proceso son almacenados en la PCB.

3.4.6.1. Implementación de la tabla de páginas por hardware

Por registros dedicados Deben estar en una lógica de alta velocidad para permitir traslación rápida de virtual a física. Para cargar o modificar estos registros hay instrucciones privilegiadas.

Por mantenimiento en memoria principal La tabla se mantiene en memoria y el registro base de la tabla de páginas (PTBR) apunta a dicha tabla. La ventaja ante un cambio de contexto es que sólo hay que cambiar el puntero a la nueva tabla. El problema son los accesos a memoria: hay un acceso por el número de página; luego combino el marco con el desplazamiento (*offset*) para lograr la dirección real y se produce el nuevo acceso a memoria. Por lo tanto, hay dos accesos a memoria por cada dirección virtual que resuelvo. Decimos que la memoria se perjudica con un factor de lentitud igual a 2. Esto se puede solucionar mediante una cache (*associative registers* ó *translation look aside*, TLB).

3.4.6.2. TLB's

La cache sería un almacenamiento confiable de las últimas páginas referenciadas.

Las TLB's se implementan así:

- Se tienen un conjunto de registros en memoria de alta velocidad.
- Cada registro tiene una clave y un valor. Cuando estoy buscando por un ítem en particular, éste se compara simultáneamente con todas las claves (*keys*).
- Si el ítem está entre las keys (*cache hit*) se toma el valor asociado.

Si bien la búsqueda es rápida, la tecnología necesaria es cara. En nuestro caso, las claves (*keys*) son los números de página y el valor el número de marco correspondiente a esa página. Si logro un *cache hit* tendré rápidamente el número de marco asociado para hacer la traslación. Sino, tendré que acceder a la tabla de páginas. Esta TLB tendrá que ser limpiada y actualizada en cada cambio de contexto, por ejemplo, para evitar una traslación incorrecta con el nuevo espacio de direcciones del proceso ingresante.

3.4.6.3. Paginado multinivel

Hoy las computadoras tienen un espacio virtual muy grande. Proporcionalmente la tabla de páginas será muy extensa (sabemos que hay una proporción definida por el tamaño de la página y la cantidad de páginas). Cuanto mayor es la tabla de páginas, más lenta y dificultosa se hará la búsqueda. Y mayor será la cantidad de memoria necesaria para su residencia. Una solución para el acceso a

grandes tablas es dividirla en partes más pequeñas, y acceder a la sección donde está la página que me interesa.

Veamos el esquema en dos niveles.

Supongamos una computadora con 32 bits de direcciones, 20 para el número de página y 12 para el desplazamiento. En un esquema de dos niveles el número de página puede dividirse en 10 bits para el número de página y 10 para el desplazamiento dentro de la página.

Número de página		Desplazamiento
p_1	p_2	d
10 bits	10 bits	12 bits

- p_1 es el índice en la tabla de página inicial (o más externa), *the outer page table*.
- p_2 es el desplazamiento dentro de la tabla externa (*outer table*).

Esto nos permite no asignar toda la tabla de página de manera contigua en la memoria principal.

3.4.6.4. Tabla de página invertida

En el esquema visto hasta ahora tenemos una tabla de página por proceso, indexada por el número de página. El S.O. traslada la referencia virtual a una física. Debe calcular cómo acceder a la dirección física.

Otra forma de manejar la traslación es a través de las tablas invertidas de página. La ventaja de este tipo de tabla es: hay una para todos los procesos y el tamaño lo definen el número de marcos. Por lo tanto habrá tantas entradas en la tabla como marcos.

Si bien se soluciona el problema de ocupar mucha memoria para las tablas de página, la búsqueda en las tablas invertidas puede llevar mucho tiempo pues está ordenada por marco.

Buscar por <pid,p>puede hacer que se deba recorrer toda la tabla. Puede agilizarse con una tabla por *hashing* ó con registros de memoria asociativa que almacenan las entradas más recientes.

3.4.6.5. Ventaja adicional del paginado: las páginas compartidas

Sea un ambiente de tiempo compartido (*time sharing*). Tiene 15 usuarios de desarrollo que están usando el editor para escribir programas. El editor ocupa 150k y 50k de espacio de datos. Trabajando los 15 usuarios a la vez necesitaríamos (150k * 15) + (50K +15) de memoria real para el uso simultáneo del editor. Si el **código** del editor es **reentrante** o **puro**, es decir, no se automodifica, puede compartirse. Supongamos tres procesos p_1, p_2 *y* p_3 que usan el editor. El editor consta de tres páginas, c/u de ellas de 50 K.

Al ser código reentrante uno o más procesos pueden ejecutarlo al mismo tiempo. Cada proceso tiene sus propios registros y sus datos. Así sólo una copia del editor está en memoria. Ahora, para 15 usuarios necesitaremos (150K) + (50K * 15). Puede compartirse todo proceso que sea reentrante y, principalmente el de uso común: compiladores, sistemas de bases de datos, etc. No es posible implementar páginas compartidas en un esquema de tabla de páginas invertida.

3.4.6.6. Asignación de marcos (frames)

La discusión con este punto se centra en cómo asignar los marcos. ¿Asignar sólo los marcos del área de memoria de usuario ó incluir las partes de memoria del sistema operativo no usadas? En algunos sistemas se dejan algunos marcos libres (depósito de marcos libres) para que, cuando se produce un *page fault*, se coloque en algún marco del depósito la nueva página mientras la página "víctima" (la página seleccionada por el algoritmo de sustitución para ser reemplazada) es descargada de memoria. Luego de desocupado, ese marco pasa a formar parte del "depósito de marcos libres" que se utilizan para este fin. Debe comprenderse que los marcos del depósito no son fijos.

Por ejemplo: pueden ser marcos libres el 2, el 5 y el 8. Se produce un *page fault* y se debe traer una nueva página. Se elige por el algoritmo de sustitución la página que ocupa el marco 7. Se carga la nueva página en el marco 2 (que formaba parte del depósito) y se va liberando el 7. Cuando termina de liberarse el 7 (puede incluir una escritura a disco) el depósito de marcos libres será ahora 5,7 y 8.

En el caso de un sistema monousuario es más fácil la decisión. Si le asignan al proceso todos los marcos libres a medida que van ocurriendo los *page faults*, llegará un momento que ocuparé todos los marcos. Los nuevos *page faults* producirán reemplazos pero siempre con toda la memoria para ese usuario.

Pero en un esquema de multiprogramación es más complicado, pues los procesos compiten entre sí.

Restricciones: No se pueden asignar más marcos de los que se tienen, salvo que se compartan páginas entre procesos.

Existe un número mínimo de marcos para asignar a un proceso, pues un número menor provocaría demasiados *page faults* (aumenta la tasa de fallas, lo contrario de lo que queremos lograr) y toda la actividad improductiva que ello implica.

La asignación de marcos depende también de la arquitectura del conjunto de instrucciones. La falla de página ocurre en la mitad de la instrucción, por lo tanto al traer a memoria la página solicitada la instrucción debe volverse a ejecutar. Por lo tanto no puede elegirse esa página (la que posee la instrucción) como víctima en la elección.

1. Caso 1: Supongamos que las instrucciones que se refieren a memoria tienen sólo una dirección. Necesito una dirección para la instrucción y otra para la referencia a memoria. Si son de páginas diferentes, debo tener dos marcos en memoria para permitir la ejecución de la instrucción.

2. Caso 2: Algunas máquinas pueden tener una arquitectura donde una instrucción pueda tomar más de una palabra y podría ocasionalmente, quedar una parte en una página y el resto en otra. Supongamos la instrucción Mover Carácter, que es de memoria a memoria y ocupa 6 bytes y podría ocupar dos páginas. Puede ser que los caracteres que se moverán se distribuyan en dos páginas también. Necesito tener activas las páginas de la instrucción y de los campos a mover.

3. Caso 3: Hay arquitecturas que permiten varios niveles de indirección. Una instrucción puede involucrar que distintas páginas estén en memoria para

su ejecución. Por lo tanto: el número mínimo de marcos está definido por la arquitectura. El número máximo por la cantidad de memoria física disponible.

3.4.7. Algoritmos de asignación

3.4.7.1. Asignación equitativa

Dados m marcos y p procesos, lo más sencillo sería dividir los m marcos por los p procesos, es decir $\frac{m}{p}$ marcos para cada proceso. Si la división no fuera exacta, podrían utilizarse los marcos del resto como depósito de marcos libres.

El problema es si están compitiendo dos procesos, de los cuales uno es chico y le bastan pocos marcos y al otro proceso se le hacen necesarios muchos, no se justifica darle a ambos la misma cantidad de marcos: el chico tendrá los suyos desocupados, mientras el otro tiene una alta tasa de fallas de páginas.

3.4.7.2. Asignación proporcional

En este algoritmo, la memoria disponible se asigna proporcionalmente a lo que el proceso necesita.

Sea v_i el espacio virtual del proceso p_i.

Sea V la suma de los espacios virtuales de los procesos.

O sea: $V = \Sigma v_i$

Sea m el números de marcos disponibles.

Entonces la asignación de marcos a_i para el proceso p_i sería:

$a_i = \frac{v_i}{V \star m}$

a_i debe ser entero, mayor al número mínimo de marcos y no debe exceder los marcos disponibles.

En cualquiera de ambas asignaciones, depende el nivel de multiprogramación. Al aumentar el grado de multiprogramación se deberán ceder marcos libres para el nuevo proceso. Si decrece, los procesos pueden utilizar los marcos liberados por el proceso saliente.

Otro punto a considerar es la prioridad de los procesos que compiten por marcos. Podrían otorgársele más marcos a un proceso de mayor prioridad para que finalice antes. En algunos sistemas se plantea la asignación proporcional donde la variable es la prioridad no el tamaño del proceso, o el tamaño y la prioridad. También un proceso de mayor prioridad podría usar para su reemplazo un marco de un proceso de prioridad menor.

A continuación aplicaremos los dos diferentes tipos de asignación a un mismo ejemplo.

3.4.7.3. Ejemplo

Sean dos procesos: P_1, de 15 K y P_2, de 135 K. La cantidad de memoria disponible es de 70 K que se reparte en 70 marcos.

Por asignación equitativa: se le otorgan 35 marcos a cada uno, aunque P_1 usará sólo 15K y el resto están libres pero no disponibles. Para P_2 irán 35 marcos: para los 135 que necesita, tendrá una alta tasa de fallas.

Por asignación proporcional:
V=15+135; V=150

$$Marcos\,(P_1) = \frac{15}{150x70} = 7$$

$$Marcos\,(P_2) = \frac{135}{150x70} = 63$$

3.4.8. Hiperpaginación

Si el número de marcos asignados a un proceso de baja prioridad se reduce por debajo del mínimo requerido por la arquitectura del computador, será preciso suspender la ejecución de ese proceso, paginar el resto de sus páginas al disco y liberar así los marcos que tiene asignados. Esta posibilidad introduce un nivel intermedio de planificación de la CPU con intercambio hacia adentro y hacia fuera.

De hecho, veamos que cualquier proceso que no tiene "suficientes" marcos. Aunque técnicamente es posible reducir el número de marcos asignados al mínimo, hay cierto número (mayor) de páginas que están en uso activo. Si el proceso no cuenta con este número de marcos, causará muy pronto un fallo de página. En ese momento, el proceso deberá reemplazar alguna página y, dado que todas sus páginas están en uso activo, deberá reemplazar una que volverá a necesitar de inmediato. Por tanto, se generará rápidamente otro fallo de página, y otro, y otro. El proceso seguirá causando fallos, reemplazando páginas por las que entonces causará otro fallo y traerá de nuevo a la memoria.

> Esta frenética actividad de paginación se denomina **hiperpaginación**. Un proceso está hiperpaginando si pasa más tiempo paginando que ejecutando.

Si un proceso no tiene suficiente memoria para su conjunto de trabajo, sufrirá hiperpaginación. Es posible que se necesite intercambiar y planificar procesos para proporcionar suficientes marcos a cada proceso y evitar la hiperpaginación.

3.5. Segmentación de memoria

La administración de memoria por segmentación se acerca mas a la visión que tiene el usuario del almacenamiento de su programa, que el esquema de la paginación. Un programa tiene subrutinas, procedimientos, funciones, arreglos, *stacks* y tablas, y, normalmente, cada una de estas estructuras tiene un identificador asociado y son de longitud variable. Una dirección específica dentro de ella se puede ver como un *offset* (desplazamiento) de la dirección de inicio.

En la segmentación, el espacio de direcciones lógicas esta formado por una colección de segmentos, cada uno de ellos con un nombre y una longitud. Una dirección puede verse como <nombre del segmento, desplazamiento dentro de él>.

La diferencia con el paginado es que el HW toma una dirección y lo divide en <página, offset>, de manera totalmente transparente para el usuario. Veamos el caso de un compilador Pascal.

La salida del compilador diferencia los segmentos de variables globales, del *stack* para procedimientos (almacenando parámetros y direcciones de retorno), la porción de código de cada procedimiento y función.

Un compilador Fortran crea un segmento separado para cada bloque COMMON y los arreglos pueden llegar a formar segmentos separados.

La implementación de la segmentación consiste en transformar una dirección de 2 dimensiones <segmento, offset>en una dirección física unidimensional. Esta transformación se realiza a través de la tabla de páginas, donde cada entrada tiene la dirección física de inicio de ese segmento en memoria y la longitud, <base, limit>.

3.5.1. Implementación

La tabla puede mantenerse en registros rápidos o en memoria principal. Con los registros ganaremos tiempo, pero debo contar con un número suficiente.

Cuando el programa tiene una gran cantidad de segmentos y, por lo tanto, no alcanzan los registros para mantener la tabla, la tabla estará en memoria.

Esta tabla es apuntada por el STBR (*segment table base register*) y la medida de ella se mantiene en STLR (*segment table length register*).

Dada una dirección lógica (s,d) donde s es el número de segmento y d el desplazamiento u *offset*, para obtener la entrada en la tabla de segmentos correspondiente a ese segmento, se suma el número de segmento al STBR. Leída esa entrada, de allí se calcula que d no supere el límite del segmento (si lo superara tendría un direccionamiento ilegal) y se suma d a la dirección del segmento, obteniéndose la dirección real (unidimensional).

3.5.2. Ventajas de la segmentación

Una ventaja importante es la posibilidad de implementar mecanismos de protección de una manera relativamente fácil.

Los segmentos son de instrucciones o datos. Los de instrucciones (rutinas, procedimientos, funciones) pueden especificarse como de sólo-lectura (*read only*) o sólo-ejecución (*execute only*). Directamente el HW chequeará los bits asociados a la entrada del segmento que lo identifican como no modificable y no permitirá acciones de este tipo.

Si un arreglo forma un segmento, el HW detectará si se quiere acceder a una componente cuyo índice supera el rango del arreglo.

La otra ventaja es la posibilidad de compartir.

Supongamos que dos programas utilizan una rutina. Ambos programas están en memoria. ¿Es necesario que la rutina se cargue dos veces?

Analicemos que la rutina es un segmento, y como tal estará cargado en memoria. Cada programa puede tener este segmento referenciado en su tabla de página, y compartirlo así entre ellos (consideremos que es código y que no se modifica). Pero... el segmento debe tener el mismo número en ambos programas (no puede ser el segmento 4 para uno y el 12 para el otro) para las referencias a sí mismo que puede hacer una dirección dentro del segmento.

3.5.3. Desventajas y/o complicaciones

Los segmentos son variables y hay que asignarles memoria. En la paginación, la medida de la página es fija. Para asignar memoria a un segmento de manera dinámica, normalmente se utiliza el algoritmo *best-fit* o *first-fit*. Puede haber fragmentación externa si los lugares que quedan en memoria son demasiado chicos para poner un nuevo segmento. Puede necesitarse compactar o que el proceso se duerma hasta que haya lugar suficiente y que el planificador elija un proceso mas chico para correr.

3.6. Ejercicios

3.6.1. Ejercicio 1

Consideren un sistema con swaping en donde la memoria consiste de la siguientes particiones fijas:

10K, 4K, 20K, 18K, 7K, 9K, 12K y 15K

Cual partición es tomada para los siguientes requerimientos sucesivos de memoria:

- 12K
- 10K
- 9K
- 15K

Hacerlo según los siguientes algoritmos:

1. Primer ajuste (First Fit)
2. Mejor ajuste (Best Fit)
3. Peor ajuste (Worst Fit)

3.6.2. Ejercicio 2

Dada la siguiente secuencia de llegada de jobs:

Job	Ins. Llegada	Tiempo CPU	Mem. requerida
A	0	10	200k
B	2	7	720k
C	2	5	1000k
D	4	5	1500k
E	5	8	800k

Se desea saber cual es el mapa de memoria para cada instante en que se produzca un cambio en la misma. La memoria consta de las siguientes particiones estáticas:

Partición	Tamaño
1	2000k
2	1200k
3	780k
4	3000k
5	5000k
6	512k

Realizar el mapa de memoria de acuerdo a los siguientes algoritmos de asignación de memoria, indicando la fragmentación producida por cada uno de ellos:

- Primer ajuste
- Peor ajuste
- Mejor ajuste

3.6.3. Ejercicio 3

Supongamos un sistema con memoria compartida particionada dinámicamente de 180K. Se tienen los siguientes jobs donde cada uno tiene asignada una CPU distinta:

Job	Memoria requerida	Tiempo de CPU	Tiempo de llegada
1	70 Kbytes	8 μ	0
2	30 Kbytes	3 μ	1
3	60 Kbytes	5 μ	1
4	25 Kbytes	2 μ	4
5	25 Kbytes	3 μ	5
6	70 Kbytes	2 μ	6

Considere que los jobs son cargados en memoria por orden de llegada y permanecen fijos en memoria hasta terminar su ejecución (reasignando si es necesario). Realizar un gráfico de la memoria indicando las fragmentaciones que se producen a cada instante.

3.6.4. Ejercicio 4

Considere la siguiente tabla de segmentos:

Segmento	Base	Longitud
0	64	60
1	2048	14
2	1024	800
3	4096	580
4	128	196

¿Qué direcciones físicas generan las siguientes direcciones lógicas?
(0, 43), (1, 10), (3, 11), (2, 500), (3, 400), (4, 112), (2, 125), (1, 2), (4, 52), (3, 422), (4, 22)

3.6.5. Ejercicio 5

Suponga un sistema con memoria segmentada. En una máquina de 150K de memoria y que en un momento dado existen 2 procesos los que utilizan los siguientes segmentos:

Proceso	Segmento	Descripción
1	0	Editor
	1	Datos
	2	Utilitarios de discos
2	0	Editor
	1	Datos
	2	Planilla de cálculos

Sean sus correspondientes tablas de segmentos:

Proceso	Segmento	Base	Longitud
1	0	85600	35000
	1	3000	12000
	2	25000	20000
2	0	85600	35000
	1	1000	1000
	2	50000	20000

- Realizar un mapa de memoria.
- Indicar qué característica debe tener el programa editor.
- Calcular la fragmentación producida.
- ¿Por qué el segmento que contiene el editor debe ser el mismo en ambos usuarios?
- Si el editor no cumpliera con la condición indicada, cuáles serían las consecuencias.

3.6.6. Ejercicio 6

Considere la siguiente secuencia de referencias de página:
1, 2, 15, 4, 6, 2, 1, 5, 6, 10, 4, 6, 7, 9, 1, 6, 12, 11, 12, 2, 3, 1, 8, 1, 13, 14, 15, 3, 8

¿Cuántos fallos de página se producirán con los algoritmos de reemplazo LRU, FIFO, segunda chance y Optimo, suponiendo que se disponen de 2, 3, 4, 5 o 6 frames?

3.6.7. Ejercicio 7

Considere la siguiente secuencia de referencias a memoria de un programa de 1.240 palabras:
10, 42, 104, 185, 309, 245, 825, 688, 364, 1100, 967, 73, 1057, 700, 456, 951, 1058

- Indique la secuencia de referencias a página suponiendo un tamaño de la misma de 100 palabras.
- Calcule la cantidad de fallos de página para esta secuencia de referencias, suponiendo que el programa dispone de una memoria de 200 palabras y un algoritmo de reemplazo FIFO.
- ¿Cuál sería la cantidad de fallos si utilizamos un algoritmo de reemplazo LRU?
- ¿Cuál con reemplazo Optimo?
- ¿Cuál con segunda chance?

3.6.8. Ejercicio 8

Considere la siguiente cadena de referencias de página:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6
Cuántos fallos de página se producirán con los algoritmos de reemplazo siguientes, suponiendo 1, 3, 5, o 7 celdas?

- FIFO
- LRU
- Optimo

3.7. Trabajos prácticos

3.7.1. El archivo /proc/pid/maps

Basándonos en lo que vimos en la Sección 3.2, veremos ahora que el sistema operativo Linux ofrece un tipo de sistema de archivos muy especial: el sistema de archivos *proc*. Este sistema de archivos no tiene soporte en ningún dispositivo. Su objetivo es poner a disposición del usuario datos del estado del sistema en la forma de archivos. Esta idea no es original de Linux, ya que casi todos los sistemas UNIX la incluyen. Sin embargo, Linux se caracteriza por ofrecer más información del sistema que el resto de variedades de UNIX. En este sistema de archivos se puede acceder a información general sobre características y estadísticas del sistema, así como a información sobre los distintos procesos existentes. La información relacionada con un determinado proceso se encuentra en un directorio que tiene como nombre el propio identificador del proceso (*pid*). Así, si se pretende obtener información de un proceso que tiene un identificador igual a "`1234`", habrá que acceder a los archivos almacenados en el directorio "`/proc/1234/`". Para facilitar el acceso de un proceso a su propia información, existe, además, un directorio especial, denominado "`self`". Realmente, se trata de un enlace simbólico al directorio correspondiente a dicho proceso. Así, por ejemplo, si el proceso con identificador igual a "`2345`" accede al directorio "`/proc/self/`", está accediendo realmente al directorio "`/proc/2345/`".

En el directorio correspondiente a un determinado proceso existe numerosa información sobre el mismo. Sin embargo, en esta práctica nos vamos a centrar

en el archivo que contiene información sobre el mapa de memoria de un proceso: el archivo "`maps`". Cuando se lee este archivo, se obtiene una descripción detallada del mapa de memoria del proceso en ese instante. Como ejemplo, se incluye a continuación el contenido de este archivo para un proceso que ejecuta el programa "`cat`".

Coloque el comando:

```
cat /proc/self/maps
```

Y verá una salida similar a ésta:

```
08048000-0804a000 r-xp 00000000 08:01 65455 /bin/cat
0804a000-0804c000 rw-p 00001000 08:01 65455 /bin/cat
0804c000-0804e000 rwxp 00000000 00:00 0
40000000-40013000 r-xp 00000000 08:01 163581 /lib/ld-2.2.5.so
40013000-40014000 rw-p 00013000 08:01 163581 /lib/ld-2.2.5.so
40022000-40135000 r-xp 00000000 08:01 165143 /lib/libc-2.2.5.so
40135000-4013b000 rw-p 00113000 08:01 165143 /lib/libc-2.2.5.so
4013b000-4013f000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Cada línea del archivo describe una región del mapa de memoria del proceso. Por cada región aparece la siguiente información:

- Rango de direcciones virtuales de la región (en la primera línea, por ejemplo, de la dirección "08048000" hasta "0804a000").
- Protección de la región: típicos bits "`r`" (permiso de lectura), "`w`" (permiso de escritura) y "`x`" (permiso de ejecución).
- Tipo de compartimiento: "`p`" (privada) o "`s`" (compartida). Hay que resaltar que en el ejemplo todas las regiones son privadas.
- Desplazamiento de la proyección en el archivo. Por ejemplo, en la segunda línea aparece "00001000" (4096 en decimal), lo que indica que la primera página de esta región se corresponde con el segundo bloque del archivo (o sea, el byte 4096 del mismo).
- Los siguientes campos identifican de forma única al soporte de la región. En el caso de que sea una región con soporte, se especifica el dispositivo que contiene el archivo (en el ejemplo, "08:01") y su nodo-i (para el comando "cat, 65455"), así como el nombre absoluto del archivo. Si se trata de una región sin soporte, todos estos campos están a cero.

A partir de la información incluida en este ejemplo, se puede deducir a qué corresponde cada una de las nueve regiones presentes en el ejemplo de mapa de proceso:

- Código del programa. En este caso, el comando estándar "`cat`".
- Datos con valor inicial del programa, puesto que están vinculados con el archivo ejecutable.

- Datos sin valor inicial del programa, puesto que se trata de una región anónima que está contigua con la anterior.
- Código de la biblioteca "`ld`", encargada de realizar todo el tratamiento requerido por las bibliotecas dinámicas que use el programa.
- Datos con valor inicial de la biblioteca "`ld`".
- Código de la biblioteca dinámica "`libc`", que es la biblioteca estándar de C usada por la mayoría de los programas.
- Datos con valor inicial de la biblioteca dinámica "`libc`".
- Datos sin valor inicial de la biblioteca dinámica "`libc`".
- Pila del proceso.

Habiendo llegado a este punto, experimente libremente investigando el mapa de memoria de otros procesos, por ejemplo, si se cambia al directorio

```
cd /proc
```

puede ver que hay un directorio por cada PID. Experimente investigando qué es lo que contiene el mapa de memoria del proceso cuyo PID es "1", luego con el de su propio intérprete de comandos "`bash`". Puede dejar ejecutando en una consola el proceso "`top`" y desde otra ver el mapa de memoria. Tome notas.

3.7.2. Llamadas a biblioteca

3.7.3. Memoria virtual en Windows

Si en Windows 98 hacemos click con el botón derecho del mouse sobre el ícono "Mi PC" y seleccionamos "Propiedades", obtenemos la ventana "Propiedades de sistema"; si hacemos click sobre la pestaña "Rendimiento", vemos que contiene un botón llamado "Memoria virtual".

Puede observar que normalmente Windows se encarga de administrar la configuración de la memoria virtual (comportamiento recomendado), pero usted puede especificar la configuración de memoria virtual y aún deshabilitar la memoria virtual, lo cual no se recomienda, salvo que usted sea un usuario con experiencia o administrador del sistema.

Por otra parte, el "Solucionador de problemas de memoria" dice:

> Si tiene demasiados documentos abiertos o se están ejecutando demasiados programas simultáneamente, puede que no tenga suficiente memoria libre para ejecutar otro programa.

Y más adelante agrega:

> ¿Tiene suficiente espacio libre en el disco duro para el archivo de paginación virtual? Windows 98 usa espacio de disco duro en la forma de un archivo de paginación virtual, para simular memoria RAM.

Con toda esta información ahora responda:

- Manteniendo el tamaño de la memoria principal, una de las ventajas de introducir memoria virtual sería ...
  - poder reducir el grado de multiprogramación
  - poder aumentar el grado de multiprogramación
  - poder mantener estable el grado de multiprogramación
  - que el sistema operativo no tenga que gestionar la memoria

3.8. Autoevaluación

1. Indique cuál de estas operaciones NO es ejecutada por el activador (dispatcher):
   a) Restaurar los registros de usuario con los valores almacenados en la tabla del proceso.
   b) Restaurar el contador de programa.
   c) Restaurar el puntero que apunta a la tabla de páginas del proceso.
   d) Restaurar la imagen de memoria de un proceso.

2. Considere un sistema con memoria virtual cuya CPU tiene una utilización del 15 % y donde el dispositivo de paginación está ocupado el 97 % del tiempo, ¿qué indican estas medidas?
   a) El grado de multiprogramación es demasiado bajo.
   b) El grado de multiprogramación es demasiado alto.
   c) El dispositivo de paginación es demasiado pequeño.
   d) La CPU es demasiado lenta.

3. ¿Cuándo se produce hiperpaginación (thrashing)?
   a) Cuando hay espera activa.
   b) Cuando se envía un carácter a la impresora antes de que se realice un retorno de carro.
   c) Cuando no hay espacio en la TLB.
   d) Cuando los procesos no tienen en memoria principal su conjunto de trabajo.

4. Respecto a un sistema operativo sin memoria virtual que usa la técnica del intercambio (*swapping*), ¿cuál de las siguientes sentencias es correcta?
   a) El uso del intercambio facilita que se puedan ejecutar procesos cuyo tamaño sea mayor que la cantidad de memoria física disponible.
   b) El intercambio aumenta el grado de multiprogramación en el sistema.
   c) En sistemas de tiempo compartido, el intercambio permite tener unos tiempos de respuesta similares para todos los usuarios, con independencia de su número.

d) El uso del intercambio aumenta la tasa de utilización del procesador.

5. En un sistema con memoria virtual ¿cuál no es una función del sistema operativo?

   a) Tratar las violaciones de dirección (accesos a direcciones no asignadas).

   b) Tratar los fallos de la TLB.

   c) Tratar los fallos de página.

   d) Tratar las violaciones de protección (accesos no permitidos a direcciones asignadas).

6. ¿Qué es falso respecto al bit Presente/Ausente de una entrada de la tabla de páginas?

   a) Es modificado por la MMU.

   b) Es leído por la MMU.

   c) Es modificado por el Sistema Operativo.

   d) Es leído por el Sistema Operativo.

7. En un sistema de memoria virtual paginado con capacidad para 5 páginas en memoria física y con una política de gestión LRU, ¿cuántos fallos de página se producirán para el siguiente patrón de referencia de páginas?: 1,2,3,7,2,4,5,2,1,7,8.

   a) 6

   b) 8

   c) 10

   d) 9

8. En un sistema con memoria virtual, ¿cuál de las siguientes afirmaciones es cierta?

   a) La traducción de direcciones es realizada por el sistema operativo.

   b) El uso de segmentación pura produce fragmentación externa.

   c) El uso de segmentación no requiere la traducción de direcciones lógicas a físicas.

   d) El uso de segmentación impide que se produzcan fallos en el acceso a memoria.

9. ¿Cuál de las siguientes tareas relativas a la gestión de memoria virtual del Sistema Operativo debe ser realizada por el hardware?

   a) Aplicar un algoritmo de reemplazo para determinar que página expulsar.

   b) Mantener ordenadas las páginas residentes en memoria principal para aplicación para aplicación de una política de expulsión FIFO.

c) Activar el bit de usada de una página.

d) Desactivar el bit de modificada de una página.

10. ¿Para cuál de los siguientes mecanismos de gestión de memoria NO es problemático que un dispositivo haga DMA directamente sobre el buffer del proceso?

a) Memoria virtual basada en paginación.

b) Multiprogramación con particiones variables y con swapping.

c) Multiprogramación con particiones fijas sin swapping.

d) Memoria virtual basada en segmentación paginada.