Memoria es un capítulo de *Sistemas Operativos* de Martín Silva, quien amablemente nos autorizó a incluirlo como lectura complementaria de *Arquitectura de Computadoras* de Patricia Quiroga.

Capítulo 3

Memoria

La finalidad de una computadora es la de procesar información mediante la ejecución de los programas apropiados. Estos programas, junto con los datos a los que acceden, deben co-residir durante su ejecución en la memoria principal (RAM) con el sistema operativo. 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.

El precio de la memoria ha descendido drásticamente con el pasar de los años; la cantidad de memoria de las primeras computadoras era de unos pocos kilobytes y la de las actuales de varios gigabytes, no obstante, el *software* también ha crecido enormemente dándonos siempre la impresión de que nuestra memoria es escasa. Y, por otra parte, ciertas técnicas que veremos le permiten al sistema operativo hacerle creer a las aplicaciones que tienen más memoria principal que la que realmente tienen.

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 sólo cuando sea necesario.

3.1. Funciones y operaciones del administrador de memoria

Como administrador de la memoria, un sistema operativo multitarea debería cumplir con las siguientes funciones [Lister and Eager, 1993]:

Reubicación En sistemas con memoria virtual, los programas cargados en memoria deben ser capaces de residir en diferentes partes de la memoria en distintos momentos de su ejecución. Una vez que se ha descargado un programa al disco (*swap*) no es deseable que deba volver a cargarse en

la misma región de la memoria principal sino que debería poder reubicarse en otra y el administrador de memoria del sistema operativo debería ser capaz de manejar todas las referencias a memoria que hace el programa de manera que apunten a la dirección correcta.

Protección Los procesos no deben ser capaces de acceder a direcciones de memoria de otros procesos, a menos que esté explícitamente permitido, y con ello nos referimos tanto a interferencias accidentales como intencionadas. En última instancia, es el procesador (*hardware*), y no el sistema operativo (*software*), el que debe satisfacer las exigencias de protección de memoria. El sistema operativo no puede anticiparse a todas las referencias a la memoria que hará un programa.

Compartimiento No obstante lo anterior, muchas veces es deseable que los procesos puedan compartir información y, por lo tanto, acceder a un área de memoria ajena. La memoria compartida es una técnica muy utilizada en la comunicación entre procesos.

Organización lógica La memoria de un equipo informático está organizada de manera lineal o unidimensional, como una secuencia de *bytes*; la de un disco, de manera similar. Esta organización del *hardware* no es apta para los programas que la utilizan; a menudo los programas están organizados en módulos o áreas, que pueden ser modificables o no, ya sea porque contienen datos, o áreas de sólo lectura o solamente ejecutables. Una forma de organización lógica es la **segmentación** (ver 3.3.4)

Organización física De forma básica podemos decir que la memoria de un computador se divide en dos niveles, por una parte la memoria principal cuyo acceso es rápido, del órden de nanosegundos (10^{-9}) y accesible en forma directa por la CPU, y por otra, la memoria secundaria, de acceso lento (del órden de milisegundos (10^{-3}). Es requisito del código gestor de memoria del sistema operativo que se encargue de la transferencia de datos entre una y otra.

Podríamos agregar otro requisito a satisfacer, que es el de la **compactación** de la memoria, para lograr tener pocos grandes espacios libres de memoria contigua y no varios pequeños espacios libres y dispersos.

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 categoría de velocidad de acceso, capacidad y costo.

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 y 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 segundo nivel (*cache*), primero debe pasar al primer nivel y luego acceder el procesador.

Hay un principio, observado por Peter J. Denning en 1967 (ver [Denning, 2005]), llamado "cercanía de referencias" que observa que cuando se ejecuta un programa, las referencias a memoria que hace el procesador tienden a estar agrupadas, por ejemplo cuando se accede sucesivamente a todos los elementos de un arreglo. Lo mismo ocurre con las referencias a instrucciones cercanas y al llamar a subrutinas. Es el concepto de localidad que veremos mas profundamente en la subsección 3.4.2 y en hiperpaginación (3.4.7.2) (*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.

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

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.

> La proyección o el trazado de un mapa en la memoria (*memory mapping*) es una técnica que consiste en hacer que una parte del

> espacio de direcciones parezca contener un objeto tal como un archivo o dispositivo, de manera que los accesos a esa parte de memoria actúen sobre el objeto.

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 o montaje (*link*), que genera un ejecutable agrupando todos los archivos objeto y resolviendo las referencias entre módulos. (ver [Carretero Pérez, 2001])

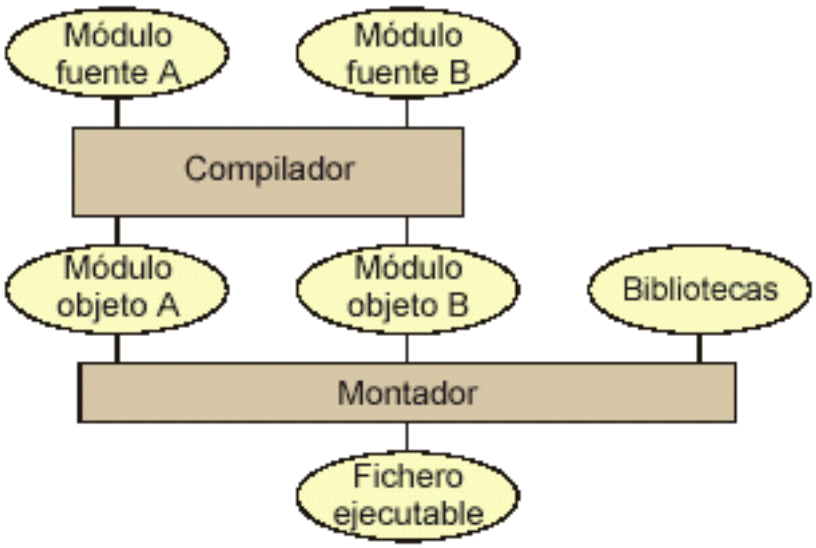


Figura 3.1: Fases en la generación de un ejecutable

Además de referencias entre módulos, pueden existir referencias a símbolos definidos en otros archivos objeto previamente compilados agrupados normalmente en bibliotecas (*library*).

> Una biblioteca es una colección de objetos tales como subrutinas (o *clases* en la Programación Orientada a Objetos) normalmente relacionados entre sí y se usa para desarrollar *software*.

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, es lo que se conoce como enlazarlo de manera **estática**. 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 durante la 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 durante la 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. Pero note que esta acción requiere de operaciones de entrada y salida, que demoran la ejecución del proceso (ver Tabla 3.2).

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*).

> **DLL (*dynamic-link library*)** Es un conjunto de subrutinas invocables por un proceso y enlazadas juntas en un archivo binario que puede ser cargado dinámicamente por aplicaciones que usen esas subrutinas. Por ejemplo `msvcrt.dll` es la biblioteca del lenguaje C que se invoca durante la ejecución de un proceso cuyo programa fue escrito en ese lenguaje, y `kernel32.dll` es una de las bibliotecas del susbsistema de la API de Windows. [Russinovich, 2005]

Los componentes de Windows que ejecutan en modo usuario y las aplicaciones usan extensivamente las DLLs. La ventaja que proveen las DLLs frente a las bibliotecas estáticas es que las aplicaciones pueden compartir las DLLs, y Windows se asegura que haya sólo una copia del código de la DLL en memoria entre las aplicaciones que la referencian. (Relacione con código reentrante en 3.3.9)

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.

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.

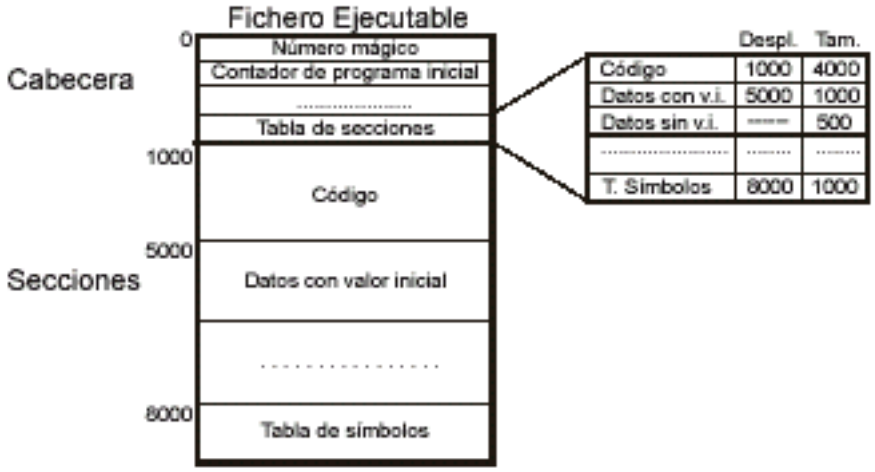


Figura 3.2: Formato simplificado de un ejecutable

- 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.

3.2.2.1. El formato COFF

COFF (*Common Object File Format*) es la especificación del formato para archivos objeto, ejecutables y bibliotecas compartidas que se usó en sistemas Unix a partir de System V Release 3 (SVR3), agregando la capacidad de contener varias secciones, característica que faltaba en el formato `a.out` original de Unix. Se le puede agregar información para depurar el programa, es decir, nombres simbólicos a funciones y variables, información del número de línea, que son útiles para establecer puntos de detención de la ejecución (*breakpoint*) o ver la traza de ejecución del proceso. Al generar un archivo COFF no se establece en qué parte de la memoria será cargado. La dirección virtual donde el primer byte del archivo se cargará se llama dirección base de la imágen. Este formato es la base de los formatos ELF y PE utilizados por Linux y Windows respectivamente.

3.2.2.2. El formato ELF

ELF (*Executable and Linking Format*) es el formato de archivos ejecutables, objeto, bibliotecas compartidas y volcados de memoria (*core dump*). Es muy flexible y extensible y no está limitado a un procesador o arquitectura en particular. Ha reemplazado a los formatos `a.out` y COFF y se utiliza en todos los sistemas operativos tipo Unix modernos, en particular en Linux. También es utilizados en las consolas de juego Playstation Portable, Playstation 2, Playstation 3 y Wii.

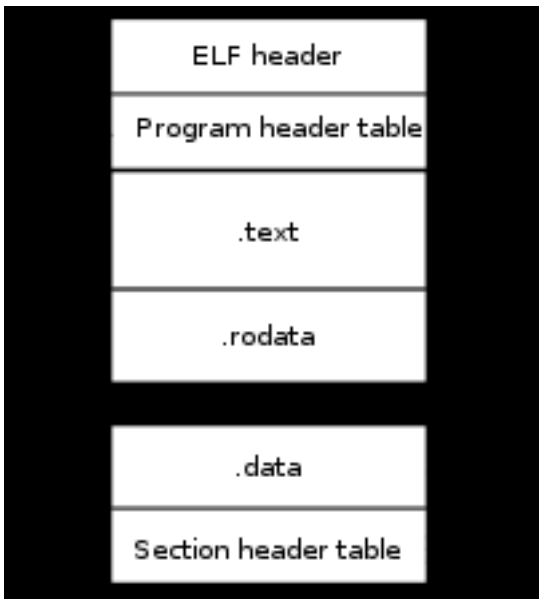


Figura 3.3: Estructura de un archivo ELF

En su estructura, consiste de una cabecera seguido por datos del archivo, que a su vez incluyen:

- Una tabla de cabecera del programa que describe cero o más segmentos.
- Una tabla de cabecera de sección que describe cero o más secciones.
- Datos referenciados por las entradas en la tabla de cabecera del programa o en la tabla de cabecera de la sección.

Los segmentos contienen información necesaria para la ejecución del archivo y las secciones contienen datos importantes para el enlace y la reubicación. Normalmente un segmento incluye una o más secciones. (Ver Figura 3.3)

3.2.2.3. El formato PE

Microsoft introdujo el formato de archivo PE (*Portable Executable*) como parte de la especificación original Win32. El archivo PE deriva del COFF que utilizaba el sistema operativo VAX/VMS, lo cual se comprende porque los arquitectos originales del equipo de desarrollo del Windows NT provenían de Digital Equipment Corporation. El término fue elegido para indicar un común denominador con todas las versiones de los sistemas operativos de Microsoft y para todas las CPUs soportadas. Se lo utiliza en los ejecutables, archivos objeto y bibliotecas de enlace dinámico (DLL).

Es una estructura de datos que encapsula la información necesaria para que el cargador del sistema operativo Windows gestione el código ejecutable envuelto dentro de la misma. Cada región puede requerir distinta protección en memoria, de manera que el comienzo de cada sección deberá estar alineada con un límite de una página. Por ejemplo, la sección *.text* que contiene el código del programa

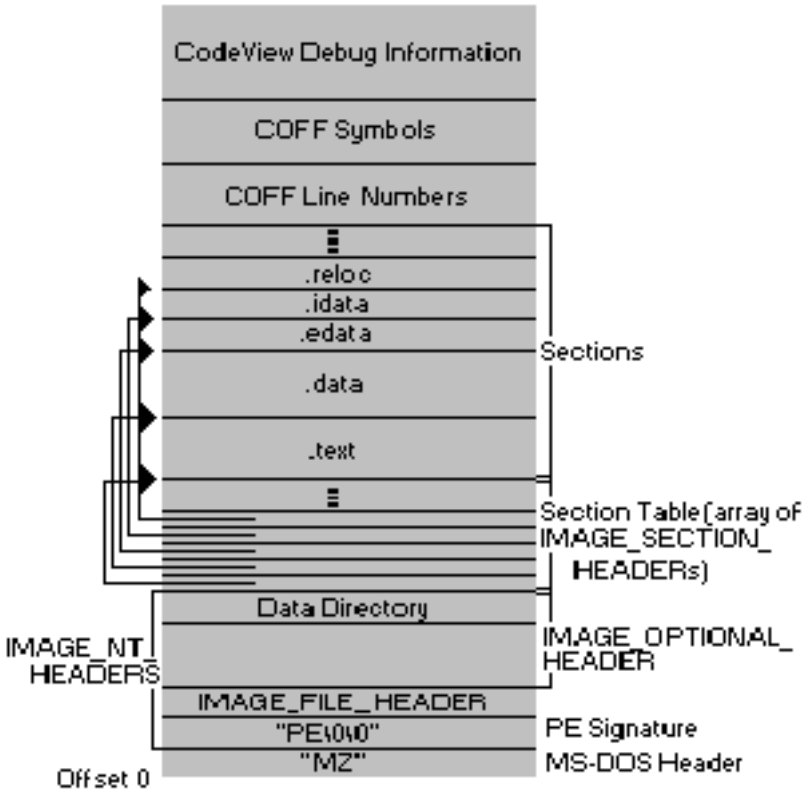


Figura 3.4: Estructura de un archivo PE

está marcado como de ejecución y de sólo lectura, mientras que la sección *.data* que tiene variables globales está marcada como de no ejecución y de lectura y escritura. (Ver Figura 3.4)

A diferencia de ELF, que usa código independiente de la posición (*position-independent code*), PE se compilan a una dirección base preferida (*preferred base address*) y si no puede ser cargado en esa dirección preferida, el sistema operativo tiene que recalcular la base, lo que implica recalcular todas las direcciones y modificar el código para usar los nuevos valores. [Pietrek, 1994]

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 caracterizada 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. Se la suele llamar `.text`
- 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. Por ejemplo, la declaración ``int i=7;'' en lenguaje C estaría en esta región. Se la suele llamar `.data`
- 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. Por ejemplo, la declaración ``int i[10];'' en lenguaje C estaría en esta región. Se le suele llamar `.bss`
- 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 durante su 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 durante su 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 con ceros), 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. Recuerde que una pila es una estructura de datos que se accede de modo LIFO.

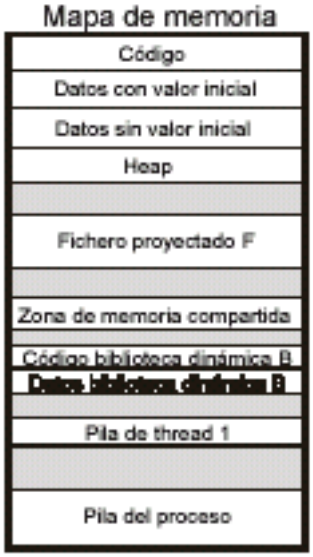


Figura 3.5: Mapa de memoria de un proceso hipotético

En la figura 3.5 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 práctica de la subsección 3.6.1.4 .

3.3. Diferentes esquemas de administración

La memoria principal debe dar cabida tanto al sistema operativo como a los diversos procesos de usuario. En los esquemas de **asignación contigua**, la memoria se divide en dos particiones, una para el sistema operativo residente y otra para los procesos de usuario. Es posible colocar el sistema operativo en memoria baja o alta. El factor principal que afecta esta decisión es la ubicación del vector de interrupciones. Puesto que el vector de interrupciones suele estar en la memoria baja, es más común colocar el sistema operativo en esa misma área.

3.3.1. Mono programación

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**. En cambio 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 (ver Figura 4-1). 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.

Figura 3.6: Direccionamiento con registro base

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.

3.3.1.3. Superposiciones

En todos los esquemas previos de administración de la memoria, la cantidad de memoria disponible a un proceso está limitado por la cantidad de memoria física. El problema de tener programas con requerimientos de memoria superiores a la memoria del computador data del comienzo de las computadoras. En aquél momento se optó por dividir el programa en partes a las que se llamaba superposiciones (*overlays*). (ver [Pankhurst, 1968])

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 sistema operativo 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 [Silberschatz y Galvin, 1999] está el ejemplo del Cuadro 3.3, bastante claro. Supongamos un ensamblador de dos pasadas. Durante la pasada 1, el ensamblador construye una tabla de símbolos; después, durante la pasada 2, genera código en lenguaje de máquina. Quizá podríamos dividir semejante ensamblador en código de la pasada 1, código de la pasada 2, tabla de símbolos y rutinas de soporte comunes utilizadas tanto por la pasada 1 como por la 2.

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*)

Para cargar todo a la vez, necesitaríamos 200K de memoria. Si sólo contamos con 150K, no podremos ejecutar nuestro proceso. Pero la pasada 1 y la 2 no tienen que estar en la memoria al mismo tiempo. Definimos dos superposiciones: la A, contiene la tabla de símbolos, las rutinas comunes y la pasada 1, y la superposición B es la tabla de símbolos, las rutinas comunes, y la pasada 2.

No debemos olvidar cargar un manejador (*driver*) de superposiciones antes de cargar la primera. Una vez ejecutada esta superposición, será el manejador el encargado de leer y cargar la segunda superposición a memoria, sobreescribiendo la primera.

Hay que considerar que esta técnica aporta entrada/salida adicional en la lectura de las superposiciones pues estas residen en disco como imágenes absolutas de memoria. Son implementados por el programador sin soporte especial por parte del sistema operativo.

Esta técnica de programación suele utilizarse actualmente en los sistemas integrados (*embedded systems*), por sus limitaciones de memoria física, que es una memoria interna dentro de un "sistema en chip" o SoC (*system on chip*) y porque no tienen memoria virtual.

Un sistema integrado (*embedded system*) es una computadora

> diseñada para realizar unas pocas funciones dedicadas, a menudo con limitaciones de tiempo real, en contraste con una computadora personal que está diseñada para ser flexible en sus funciones.

Por ejemplo, podemos considerar que un dispositivo ADSL (*Asymmetric Digital Subscriber Line*) enrutador inalámbrico es un sistema integrado, como también puede serlo una consola de video-juegos, esto puede deducirse si observa que normalmente no hay actividad de entrada/salida en un video-juego mientras se permanece en un determinado nivel, y al superarlo se descarga de memoria para cargar el siguiente.

La mayor desventaja de las superposiciones es el esfuerzo que le demanda al programador. Identificar segmentos a superponer no es tarea fácil ni confiable. Una solución muy superior es la utilización de un esquema de administración de la memoria que libere al usuario de las limitaciones de la memoria física. A tales sistemas se los denomina **memoria virtual**.

3.3.2. Multiprogramación

En un ambiente de multiprogramación hay varios procesos compartiendo la memoria, por lo tanto, la protección se debe intensificar protegiendo 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 mono programación 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 varias 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 llama el **registro de tamaño** o el **registro límite** (ver Figura 3.7).

Figura 3.7: Registros de tamaño y reubicación

3.3.2.1. Múltiples particiones fijas

Si todas las particiones son del mismo tamaño, el sistema operativo sólo necesita llevar la cuenta de cuáles 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 [1]. Si se hace esto, el sistema operativo también debe restablecer el registro límite cada vez que se le asigna un proceso 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 cuál 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*) [Harris, 2002] (ver Figura 4-3). 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**.

Figura 3.8: Tablero de ajedrez y compactación

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 asignación 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.

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

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 sección 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, lo que a los fines prácticos podría considerarse lo mejor y más sencillo. *First fit* y *Best fit* hacen un mejor aprovechamiento de la memoria que *Worst fit*.

El otro punto a tener en cuenta es cómo 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 módulo 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

El sistema de compañeras (*buddy system*) es un compromiso entre la asignación de tamaño fijo y la de tamaño variable [Knuth, 1997]. 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 3.4 muestra cómo las unidades de asignación se dividirán y se juntarán a medida que la memoria es asignada y liberada.

Acción	Memoria					
Comienzo	1M					
A solicita 150 K	A	256K			512K	
B solicita 100 K	A	B	128K		512K	
C solicita 50 K	A	B	C	64K	512K	
B libera	A	128K	C	64K	512K	
D solicita 200K	A	128K	C	64K	D	256K
E solicita 60K	A	128K	C	E	D	256K
C libera	A	128K	64K	E	D	256K
A libera	256K	128K	64K	E	D	256K
E libera	512K				D	256K
D libera	1M					

Cuadro 3.4: Sistema de compañeras

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. Reubicación

Al movernos de un sistema de monoprogramación a otro de multiprogramación es necesario tener en cuenta que en este último no se puede conocer con antelación la posición de memoria que ocupará un programa cuando se cargue en memoria para proceder a su ejecución, puesto que dependerá del estado de ocupación de la memoria, pudiendo variar, por lo tanto, en sucesivas ejecuciones del mismo. En un sistema con multiprogramación es necesario realizar un proceso de traducción o **reubicación** de las direcciones de memoria a las que hacen referencia las instrucciones de un programa para que se correspondan con las direcciones de memoria principal asignadas al mismo.

> Las direcciones físicas son los números binarios que apuntan a posiciones en la memoria física (normalmente en RAM), es decir a la circuitería electrónica de las celdas de memoria.
>
> Las direcciones lógicas son las que utiliza el programa. El *hardware* de administración de memoria convierte las direcciones lógicas en direcciones físicas.
>
> Las direcciones relativas son un caso particular de las direcciones lógicas, en las cuales las direcciones se expresan como una posición relativa a algún punto conocido, normalmente el principio del programa.

De esta manera los programas pueden cargarse y descargarse de memoria a lo largo de la ejecución. Este proceso de traducción crea un **espacio lógico independiente** para cada proceso proyectándolo sobre la parte correspondiente de la memoria principal de acuerdo con una función de traducción. Esta función la lleva a cabo un módulo específico del procesador llamado MMU (*Memory Management Unit*). El programa se carga en memoria desde el ejecutable sin modificaciones y durante su ejecución se traducen las direcciones generadas. El sistema operativo almacena asociado a cada proceso cuál es la función de traducción que le corresponde al mismo y en cada cambio de proceso debería indicar al procesador qué función debe usar para el nuevo proceso activo.

> La Unidad de Administración de Memoria, en adelante MMU (*Memory Management Unit*) es un componente *hardware* de la computadora responsable de manejar los accesos a memoria solicitados por la CPU. Las funciones que realiza son las de traducción de direcciones virtuales a direcciones físicas, protección de la memoria, control de la *cache*, arbitraje del acceso al bus y, originariamente, cambio de banco.

3.3.3. Paginación simple

Una solución al problema de la fragmentación externa es permitir que el espacio de direcciones lógico de un proceso sea **no contiguo**. Así, se puede asignar memoria física a un proceso siempre que hay alguna disponible. Una forma de implementar esta solución es adoptar un esquema de **paginación**. Con este esquema de administración, ya no es necesario tener el proceso completo cargado en memoria. El concepto de paginación puede acreditarse a los diseñadores del sistema Atlas, que ha sido descrito por [Kilburn et al, 1961] y [Howarth et al, 1961].

En un sistema que utiliza paginación simple, la memoria se divide en bloques de tamaño fijo llamados **marcos** (*page frames*). A los procesos se los divide en bloques llamados **páginas** (*pages*) los cuales tienen el mismo tamaño que los marcos. El tamaño de un marco (y, por ende, de una página) está determinado por el *hardware*. Cuando se carga un proceso en la memoria, el sistema operativo carga cada página en un marco sin usar, los cuales no necesariamente deben estar contiguos.

Una **tabla de páginas** almacena el número de marco de página asignado para cada página. Así, el número del marco en el cual la página N fue cargado es almacenado en la $N - ésima$ entrada en la tabla de páginas. El tamaño de la página es una potencia de 2. Así, si una dirección lógica contiene L bits, y el tamaño de página es de 2^P bytes, P bits de una dirección especifican el desplazamiento dentro de una página y los restantes $L - P$ bits especifican el número de página. El *hardware* genera la dirección física añadiendo el desplazamiento de página al número de marco extraído de la tabla de páginas (Figura 4-5). Es totalmente transparente al proceso la traducción de la dirección lógica a una dirección física, incluyendo la división de la dirección lógica en el número de página y el desplazamiento.

Figura 3.9: Paginación simple

En un sistema paginado se puede usar un registro de tamaño para atrapar las direcciones fuera de rango. Pero más frecuentemente se estila que la entrada de la tabla de páginas contenga un bit que indique válido o inválido. Para el caso de un proceso que sólo tenga n páginas, sólo las primeras n entradas de la tabla de páginas estarán marcadas como válidas. En algunos sistemas se pueden incluir también uno o más bits de protección en una entrada de la tabla de páginas. Sin embargo, las capacidades de protección están más comúnmente asociadas con los sistemas con segmentación.

En los sistemas en los que los procesos pueden tener un gran número de páginas se puede usar un sistema multinivel (Figura 4-6).

Figura 3.10: Tablas de páginas multinivel

El número de página puede estar partido en dos o más componentes. El primer componente sirve como índice a la tabla de páginas de máximo nivel. La entrada en esa tabla apunta a la tabla del próximo nivel. El próximo componente en el número de página sirve como índice en esa tabla. El proceso continúa hasta que es accedida la tabla del último nivel. Esa entrada contendrá el número de marco asociado con la página.

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.

3.3.4. Segmentación simple

Un aspecto importante de la gestión de memoria que se hizo inevitable al surgir la paginación es la separación entre la visión que el usuario tiene de la memoria y la memoria física real. La visión del usuario no es igual a la memoria física real; sólo tiene una correspondencia con ella. [Dennis, 1965] fue el primero en tratar el concepto de segmentación.

La segmentación, como la paginación, divide un programa en un número de bloques más pequeños llamados **segmentos** y cada uno de ellos se asigna a la memoria de forma independiente. A diferencia de la paginación, los segmentos son de tamaño variable. La responsabilidad de dividir un programa en segmentos recae en el usuario (o compilador), el sistema operativo no está involucrado. El *hardware* establece un límite superior en el tamaño de cualquier segmento. El programa objeto generado por un compilador puede tener diferenciados los segmentos de variables globales, de la pila o *stack* para procedimientos o funciones (almacenando parámetros y direcciones de retorno), de la porción de código de cada procedimiento y función.

Una tabla de segmentos es muy similar a una tabla de páginas. No obstante, dado que los segmentos son de tamaño variable, la memoria no puede ser dividida previamente en nada parecido a un marco. El sistema operativo debe mantener una lista de segmentos libres y asignarlos a los huecos en memoria. El problema de cómo se realiza la asignación es el mismo que los encontrados en el esquema de partición variable.

Una entrada de la tabla de segmentos debe tener campos para almacenar la dirección de comienzo del segmento en memoria principal y el tamaño del segmento. Si el tamaño máximo del segmento es m bits, los últimos m bits de la dirección lógica especifican el desplazamiento del segmento. Los bits restantes especifican el número del segmento. La dirección lógica se traduce a una dirección física extrayendo el número de segmento y el desplazamiento a partir de la dirección lógica. El número de segmento se usa como índice a la tabla de segmentos. El desplazamiento se compara con el tamaño del segmento. Si el desplazamiento es mayor que el tamaño del segmento se genera un fallo por dirección inválida, provocando que se aborte el programa. Caso contrario, el desplazamiento es agregado a la dirección de comienzo del segmento para generar la dirección física (Figura 4-7). Para incrementar la velocidad, la comprobación del tamaño y la generación de la dirección física se pueden realizar concurrentemente.

Figura 3.11: Segmentación simple

Dado que los segmentos están definidos por el usuario (por el compilador, más bien), es posible definir que ciertos segmentos sean de sólo lectura. Agregando un bit que indique "sólo lectura" en una entrada de la tabla de segmentos,

el sistema de administración puede comprobar las operaciones de escritura a segmentos de sólo lectura y generar un fallo si detecta tales operaciones.

Si existe el soporte para segmentos de sólo lectura, también puede hacerse posible que varios procesos compartan los segmentos de sólo lectura y que, por lo tanto, disminuyan el uso de la memoria. Típicamente esto ocurre cuando dos procesos están ejecutando el mismo programa y el código para ese programa está diseñado para estar en uno o mas segmentos de sólo lectura.

Los segmentos compartidos también pueden usarlos los procesos que ejecutan diferentes programas pero que usan las mismas subrutinas de biblioteca. En esta situación, se debe tener cuidado que las direcciones dentro de un segmento compartido funcionen con ambos programas. Para las direcciones de ubicaciones dentro del segmento, la solución más simple es usar direccionamiento relativo. Éste usa el desplazamiento a partir del valor actual del contador de programa para generar la dirección de destino. Para todas las direcciones también es una posibilidad utilizar direccionamiento indirecto a partir de un registro que apunte al segmento apropiado. El direccionamiento absoluto, en el cual se especifica el segmento y desplazamiento de la dirección de destino, es posible solamente si todos los programas usan el mismo número de segmento.

Una ventaja importante de la segmentación es la posibilidad de implementar mecanismos de protección de una manera relativamente fácil, por ejemplo si definiéramos únicamente segmentos de instrucciones o datos, los de instrucciones (es decir, rutinas, procedimientos o funciones) pueden especificarse como de sólo-lectura (*read only*) o sólo-ejecución (*execute only*). Otra ventaja es la posibilidad de compartir segmentos (código o datos de sólo-lectura).

3.3.5. Segmentación con paginación

La segmentación puede combinarse con la paginación para proveer la eficiencia de la paginación con las posibilidades de poder compartir y proteger de la segmentación. Tal como pasa con la segmentación simple, la dirección lógica especifica el número de segmento y el desplazamiento dentro del segmento. Sin embargo, cuando se agrega paginación, el desplazamiento del segmento se lo subdivide en número de página y un desplazamiento de página. La entrada de la tabla de segmentos contiene la dirección de la tabla de páginas de segmentos. El *hardware* agrega los bits del número de página de la dirección lógica a la dirección de la tabla de páginas para ubicar la entrada de la tabla de páginas. La dirección física se forma añadiendo el desplazamiento de página al número de marco de página especificado en la entrada de la tabla de páginas (Figura 4-8).

Figura 3.12: Segmentación con paginación

El primer sistema que manejó la segmentación paginada fue el GE 645, en el que se implementó originalmente MULTICS [Organick, 1972].

3.3.6. Tablas de páginas y de segmentos

Cada proceso tiene sus propias tablas de páginas y/o segmentos las cuales el sistema operativo almacena en memoria. En algunos sistemas un registro de

administración de la memoria apunta a la tabla de segmento o de página para el actual proceso en ejecución. Cuando se le otorga control de la CPU a un proceso nuevo sólo se necesita reiniciar un único registro para conmutar al espacio de direcciones lógicas del nuevo proceso.

Si las tablas de páginas son suficientemente pequeñas, la unidad de administración de la memoria puede tener un conjunto de registros dedicados para mantener la tabla de páginas. Los registros por lo general son más rápidos que la memoria convencional y por esto se acelera el proceso de traducción de las direcciones, para lo cual se utilizan instrucciones privilegiadas.

El puntero a esa tabla, como a otros datos propios del proceso, son almacenados en la PCB.

3.3.7. Memoria asociativa

En los sistemas que tienen tablas de páginas extremadamente grandes se puede usar la **memoria asociativa** (*translation lookaside buffer* o TLB) que es una memoria ultrarápida (*cache*) de la CPU administrada por la MMU.

Cada elemento en una memoria asociativa está identificado mediante un valor índice. A diferencia de la memoria convencional, la memoria asociativa no está referenciada por una dirección, sino con un valor de búsqueda; se revisan todos los elementos buscando uno con un valor índice correspondiente. La búsqueda de todos los elementos se realiza en paralelo de manera que el tiempo de acceso es el tiempo necesario para acceder a un elemento.

Los sistemas de paginación usan memorias asociativas pequeñas y de alta velocidad para mejorar el rendimiento. Una entrada en la memoria asociativa debería ser idéntica a una entrada de la tabla de páginas excepto que debería contener también un campo para el número de página. Cuando se especifica una página se busca la memoria asociativa al mismo tiempo que se accede a la tabla de páginas. Si se encuentra una entrada en la memoria asociativa (*cache hit* o *TLB hit*) con el número de página coincidente, se aborta el acceso a la tabla de páginas y se usa la entrada de la memoria asociativa. Si no se encuentra una coincidencia en la memoria asociativa se usa la entrada de la tabla de páginas (*page walk*), lo que tardará varios ciclos más, sobretodo si la página que contiene la dirección buscada no está en memoria primaria. Si en la tabla de páginas no se encuentra la dirección buscada, se provoca un fallo de página (Figura 4-9).

Figura 3.13: Memoria asociativa - TLB

A la misma vez que está siendo traducida la dirección lógica, la memoria asociativa reemplaza una de sus páginas, típicamente la entrada menos usada recientemente, con la entrada de la tabla de páginas para la páginas que está siendo accedida actualmente. De esta manera, la memoria asociativa mantiene una copia de las entradas de las páginas más usadas recientemente.

Cuando se le da el control de la CPU a un proceso nuevo (cambio de contexto), la unidad de administración de la memoria no debe usar entradas de la memoria asociativa de un proceso previo. La manera más fácil de prevenir esto se logra si el sistema operativo marca cada entrada en la memoria asociativa como inválida. Como alternativa, la memoria asociativa podría agregar un campo para contener un número de identificación del proceso. De manera tal que cada

vez que se intenta un acceso, una coincidencia también implica una coincidencia entre el número de identificación de proceso con el valor en un registro especial que contenga al número de identificación del proceso actual.

La mayor mejora en el desempeño se logra cuando la memoria asociativa es significativamente más rápida que la búsqueda en la tabla de páginas normal y la **tasa de aciertos** (*hit ratio*) es alta.

> La tasa de aciertos es el cociente entre los accesos que encuentran una coincidencia en la memoria asociativa (los exitosos) y aquellos que no lo encuentran.

3.3.8. Tabla de páginas invertida

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

En los sistemas paginados que tienen un gran espacio de direcciones virtual que usan una memoria asociativa, se puede usar una tabla de páginas invertida para reducir el tamaño de la tabla de páginas. En lugar de contar con una tabla de páginas que contenga una entrada para cada una de un número muy grande de páginas, la tabla contiene una entrada para cada marco de página, por lo tanto habrá tantas entradas en la tabla como marcos. La entrada almacena el número de página asignado a ese marco. Cuando una dirección no puede ser resuelta mediante una entrada en la memoria asociativa, se busca en la tabla invertida una entrada que contenga el número de página. 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, puede hacer que se deba recorrer toda la tabla. Para acelerar la búsqueda se usan técnicas de *hashing*.

> *Hash* se refiere a una función o método para generar claves o llaves que representen de manera casi unívoca a un documento, registro, archivo, etc, resumir o identificar un dato a través de la probabilidad, utilizando una "función *hash*" o "algoritmo *hash*". Un *hash* es el resultado de dicha función o algoritmo.

3.3.9. 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, cada una 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.

> *Código reentrante*: Una función reentrante es una función que puede ser usada por más de una tarea sin temor a la corrupción de los datos. Una función reentrante puede ser interrumpida en cualquier momento y retomada en cualquier momento posterior sin pérdida de datos. Las funciones reentrantes usan o bien variables locales (es decir, registros de la CPU o variables en la pila) o datos protegidos cuando se usan variables globales.

Un ejemplo de una función reentrante es la siguiente:

```
void strcpy(char *dest, char *src)
{
     while (*dest++ = *src++){
        ;
     }
     *dest = NUL;
}
```

Un ejemplo de una función no reentrante se muestra a continuación (`var1` está declarada global). Dado que la función accede a una variable global, no es reentrante.

```
void foo(void)
{
  ...
  ...
  var1 += 23;
  ...
  ...
}
```

Sin embargo, la función se puede hacer reentrante, protegiendo la sección crítica del código (ver sección 5.2).

```
void foo(void)
{
  ...
  deshabilitar_interrupciones();
  var1 += 23;
  habilitar_interrupciones();
  ...
}
```

Por ejemplo, los compiladores específicamente diseñados para *software* embebido (*embedded*) típicamente proveen de bibliotecas reentrantes.

> Windows NT y posteriores
>
> Windows NT es completamente reentrante - partes significativas de Windows 95 no eran reentrantes, principalmente el código de 16

> bits tomado de Windows 3.1. Este código no reentrante incluía a la mayoría de las funciones gráficas y de administración de la ventana. Cuando una aplicación de 32 bits sobre Windows 95 intentaba llamar a un servicio de sistema implementado en código de 16 bits no reentrante, debía obtener primero un candado global (*system wide lock*) o *mutex*, para evitar que otros hilos ingresaran a la base de código no reentrante. Peor aún, una aplicación de 16 bits mantenía a este candado mientras duraba su ejecución. Como resultado, a pesar de que el núcleo de Windows 95 contenía un planificador multi hilado de 32 bits desalojable, las aplicaciones a menudo ejecutaban de manera mono hilada porque una buena parte del sistema estaba todavía implementado en código no reentrante.[Russinovich, 2005]

3.3.10. Intercambio

Puede ocurrir que el sistema operativo decida que un proceso en memoria salga temporariamente, porque necesita mas espacio, para incrementar el número de procesos que comparten la CPU o por que llega un proceso de mayor prioridad.

> La técnica de llevar temporariamente un proceso a memoria secundaria se llama intercambio o *swapping* (ver [Corbató et al, 196]). Y al espacio en disco donde se almacena, se le solía llamar *backing store*.

Cuando se saca de memoria, se hace *swap-out*; cuando se trae nuevamente, *swap-in*.[2]

En el caso de un algoritmo de planificación de CPU basado en prioridades (ver 4.3.3) 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 ligazón (*binding*) que haga: solo puede cambiar de lugar si tiene *binding* en el momento de ejecución. *Backing storage* era normalmente un disco rápido (distinto del que almacenaba archivos de los usuarios) y con suficiente capacidad para mantener las imágenes de los procesos de los usuarios y debía proveer acceso directo a estas imágenes, actualmente el mismo disco que almacena archivos de usuario también se usa para intercambio.

El sistema mantiene una cola de listos (ver 4.2.1) con aquellos procesos cuyas imágenes están listas para ejecutar en *backing store* o en memoria. Según veremos, 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

[2] Como esta técnica ahora se usa en combinación con el paginado, los términos actualmente son *page-in* y *page-out*, respectivamente.

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.

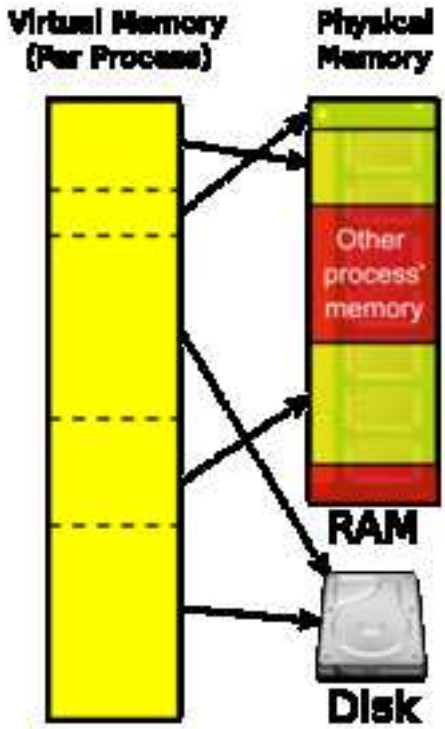
En pocos sistemas se usa el *swapping* estándar, sino mas bien combinado con otras técnicas. La decisión acerca de qué proceso o procesos son intercambiados a y desde memoria principal le corresponde al planificador del sistema operativo.

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.

Llegados a este punto de nuestro estudio, coincidente con la evolución que tuvieron los sistemas operativos y el *hardware*, se desprende el siguiente avance a partir de dos características de la paginación y la segmentación; por una parte, que todas las referencias a la memoria dentro de un proceso son direcciones lógicas que se traducen dinámicamente a direcciones físicas durante la ejecución, de tal forma que un proceso puede cargarse y descargarse de la memoria principal ocupando regiones diferentes en distintos momentos de su ejecución, como ya se dijo. Y por otra parte, que un proceso puede dividirse en varias partes y no es necesario que esas partes se encuentren contiguas en la memoria principal durante la ejecución.

En el año 1961, Fotheringham en la Universidad de Manchester y para la computadora Atlas, 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 (real) disponible. El programador se desentiende de la cantidad de memoria real que necesitará su proceso. El proceso hace uso de su propio **espacio de direcciones virtuales**. El sistema operativo guarda aquellas partes del programa de uso corriente en la memoria principal y el resto permanece en disco, ver 3.4.


Virtual Memory
(Per Process)
Physical
Memory
Other
process'
memory
RAM
Disk

La memoria virtual y la multiprogramación se corresponden bien entre sí pues, por ejemplo, si tengo 8 programas de 1 MB puedo asignar una partición de 256 KB en una memoria de 2 MB. 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.

En un sistema de memoria virtual, el espacio de memoria lógica disponible para un programa es totalmente independiente del espacio de memoria física. Un programa que necesite 2^{20}bytes de memoria puede ejecutar en un sistema con memoria virtual con solamente 2^{16}bytes de memoria real. La memoria virtual libera a los programas de las restricciones de las limitaciones de la memoria física.

3.4.1. Paginación por demanda

La paginación por demanda combina las características de la paginación simple y las superposiciones para implementar memoria virtual. En un sistema con paginación por demanda cada página de un programa se almacena en forma contigua en el espacio de intercambio de paginación en almacenamiento secundario. A medida que se hace referencia a ubicaciones en páginas, estas páginas se copian en los marcos de página de memoria. Una vez que la página está en la memoria, se accede a ella como en la paginación simple.

Cada entrada en la tabla de páginas tiene como mínimo dos campos: el marco de la página y el bit dentro/fuera (*in/out bit*). Cuando se genera una dirección virtual el *hardware* de administración de memoria extrae el número de página de la dirección y se accede la entrada correspondiente en la tabla de páginas. Se comprueba el bit dentro/fuera y si la página está en memoria se genera la dirección física añadiendo el desplazamiento de página al número de marco de página. Si la página no está en memoria, se produce una trampa (*trap*) o excepción por **fallo de página** (*page fault*) transfiriendo el control a la rutina de fallos de página del sistema operativo.

Cuando se produce un fallo de página, el sistema operativo comprueba si están libres algunos marcos de página en memoria. Sino, selecciona una página para eliminarla. La página marcada para eliminación se copia a almacenamiento secundario y el bit dentro/fuera en su entrada en la tabla de páginas se cambia a "fuera". Si hay un marco de página libre, el sistema operativo copia la página en el marco libre. La entrada de la tabla de páginas de la página ingresada en memoria se modifica indicando el número de marco y que la página está ahora en memoria. Se completa la rutina de tratamiento del fallo de página a continuación y el *hardware* vuelve a ejecutar la instrucción que generó la trampa.

Se le puede agregar un **bit sucio** (*dirty bit*) a cada entrada de la tabla de páginas. El *hardware* de administración de la memoria cambia el bit sucio cuando hay una referencia de escritura a esa página. Cuando se carga en memoria una página por primera vez, se "limpia" el bit sucio. Si el bit sucio de una página seleccionada para reemplazo está limpio, significa que la página no ha sido modificada desde que fue cargada en memoria. Solo las páginas reemplazadas que han sido modificadas necesitan ser escritas de nuevo al espacio de intercambio.

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 a la MMU.

En un esquema de paginado por demanda puro, el proceso comienza haciendo fallos de página con la primer instrucción a ejecutar, y así continúa, con una alta tasa de fallos, hasta que se estabiliza al tener todas las páginas necesarias en memoria. El momento en que se produce el fallo de página 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 la captura o *fetch*, habrá que realizarlo 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.

3.4.1.1. Ejemplo

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).

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. 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. Localidad de referencia

Cuando hay referencias a páginas ya cargadas en memoria, la traducción de direcciones es realizada totalmente por el *hardware*, rindiendo un desempeño comparable a sistemas sin memoria virtual. Pero la referencia a páginas intercambiadas a disco provocan trampas de fallo de página retardando el tiempo de acceso por más de un órden de magnitud (ver Tabla 3.1)

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*.

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

El rendimiento global sería inaceptable si las referencias a memoria fueran aleatorias. Sin embargo, las referencias tienden a estar localizadas a un pequeño conjunto de páginas. Esta **localidad** da como resultado que la tasa de aciertos de páginas en memoria respecto de las que no lo están sea suficientemente alta como para tiempos de acceso promedio aceptables. A la lista ordenada de números de páginas accedida por un programa se le llama su **cadena de referencias** (*reference string*).

Podemos ver dos formas de localidad; por una parte, cuando se ha hecho referencia a una ubicación en memoria hay mucha posibilidad de que sea referenciada otra vez a corto tiempo, a esto le llamamos **localidad temporal** (*temporal locality*); y por otra, la **localidad espacial** (*spatial locality*) se refiere al hecho de que si es accedida una ubicación en memoria es probable que sea accedida una ubicación cercana a ella en la próxima instrucción.

Los experimentos han mostrado que durante la ejecución de un proceso, las referencias tienden a agruparse en un número de ciertas localidades [Hatfield, 1972]. Por ejemplo, un programa que ejecuta un lazo accederá un conjunto de páginas

que contienen las instrucciones y datos referenciados en ese lazo. Las llamadas a funciones tenderán a incrementar los números de páginas en la localidad. Cuando el programa salga del lazo se moverá a otra sección del código cuya localidad de páginas es virtualmente distinta de la localidad previa.

3.4.3. Traba de páginas

El sistema de paginación debe ser capaz de trabar páginas en memoria de manera tal que no puedan ser intercambiadas a almacenamiento secundario. Gran parte del sistema operativo debe estar "encerrado con llave" en memoria. Y lo más importante es que el código que selecciona la próxima página que debe ser intercambiada a memoria nunca debería estar intercambiada a almacenamiento secundario porque nunca podría ejecutarse a sí misma para ser intercambiada de regreso a la memoria.

> ***Aclaración:*** En la bibliografía en inglés se utilizan los términos *lock* y *block* ambos traducidos al español como "bloqueo". En español utilizamos el término "bloquear" en el sentido de "interceptar", "obstruir" o "impedir", pero el término *lock* se refiere a colocar una llave o candado a algo, situación que puedo revertir por mí mismo. En cambio, si me encuentro impedido u obstruido no es tan sencillo salir por mí mismo de la situación porque normalmente obedece a factores externos. Por ejemplo, puedo salir de casa por mí mismo si abro la cerradura (*unlock*) pero no puedo salir por mí mismo de un atasco de tránsito.

El diseño del sistema de entrada y salida puede requerir también que permanezcan en memoria algunas de las páginas del proceso. Si las operaciones de entrada y salida de un dispositivo pueden leer o escribir datos directamente a una ubicación de memoria del proceso, las páginas que contengan esas ubicaciones de memoria no deberían estar intercambiadas a almacenamiento secundario. Si bien pueden no estar actualmente referenciadas por el proceso, están siendo referenciadas por el dispositivo de entrada/salida. Al colocar un bit cerrojo (*lock bit*) en una entrada de una tabla de páginas prevenimos que la página sea intercambiada a almacenamiento secundario.

A pesar de que se debe tener cuidado con evitar que algunas páginas sean intercambiadas a almacenamiento secundario, no es necesario el bit cerrojo en todos los casos. El sistema debe estar diseñado de manera tal que ninguna página del sistema operativo sea intercambiada a almacenamiento secundario. El sistema de entrada y salida podría estar diseñado de manera tal que no permita la entrada/salida al espacio de direcciones del proceso directamente, se les debería requerir a todos los dispositivos que transfieran los datos desde y hacia un *buffer* del sistema operativo. En un sistema como éste, el intercambio se limita a las páginas de procesos de usuario y el sistema puede diseñarse de manera que cualquier página de usuario pueda ser intercambiada en cualquier momento.

3.4.4. Tamaño de la página

Un número de factores afectan la determinación del tamaño de la página:

- El tamaño de la página siempre es una potencia de 2, con rangos que van típicamente desde los 512 bytes hasta los 16K.
- Un tamaño pequeño de página reduce la fragmentación interna.
- Un tamaño grande de página reduce el número de páginas que se necesitan, por lo tanto reduciendo el tamaño de la tabla de páginas. Una tabla de páginas mas pequeña requiere menos memoria (al espacio perdido para almacenamiento de la tabla de páginas se le conoce como **fragmentación de la tabla**). La tabla de páginas debe cargarse en registros de páginas, el tiempo de carga se reduce al tener tablas mas pequeñas.
- Un tamaño grande de página reduce la sobrecarga involucrada en el intercambio de páginas desde y hacia memoria. Además del tiempo de procesamiento requerido para manejar un fallo de página, la operación de entrada/salida incluye el tiempo requerido para mover la cabeza de lectura/escritura del disco a la pista apropiada (tiempo de búsqueda o *seek time*), el tiempo requerido para que el sector gire hasta quedar debajo de la cabeza de lectura/escritura (tiempo de latencia o *latency time*) y el tiempo para leer la página (tiempo de transferencia o *transfer time*). Dado que los tiempos de búsqueda y latencia típicamente representan más del 90 % del tiempo total, la lectura de dos bloques de 1K puede llevar casi el doble de tiempo que la lectura de un bloque de 2K.
- Un tamaño de página más pequeño, con su resolución más fina, está en mejores condiciones de acertarle a la localidad de referencias del proceso. Esto reduce la cantidad de información no utilizada copiada de ida y vuelta entre la memoria y el almacenamiento de intercambio. También reduce la cantidad de información no utilizada almacenada en memoria principal dejando más memoria disponible para fines útiles.

3.4.5. Algoritmos de reemplazo de páginas

El algoritmo que selecciona la página para ser intercambiada a almacenamiento de intercambio se llama **algoritmo de reemplazo de páginas**. El algoritmo de reemplazo de páginas óptimo selecciona para remover a la página que no será referenciada de nuevo en lapso más largo de instrucciones ejecutadas. Desafortunadamente, este algoritmo es solamente teórico, el implementarlo requeriría conocimientos de lo que ocurrirá en el futuro. Por lo tanto, su uso está limitado a servir como punto de referencia (*benchmark*) con el cual puedan ser comparados otros algoritmos de reemplazo de páginas.

El algoritmo de reemplazo de páginas FIFO (*First-in First-out*) selecciona a la página que ha estado en memoria por más tiempo. Para implementar este algoritmo, las entradas de la tabla de páginas deben incluir un campo para el tiempo en el que fue cargada en memoria. Cuando se carga la página el sistema operativo graba el campo con la hora actual. La página seleccionada para reemplazo será la que tenga la hora de cargada más temprana. A pesar de que es fácil de implementar, FIFO no es muy eficiente. A las páginas usadas con frecuencia, aún si han estado en memoria por mucho tiempo, obviamente no se las debería intercambiar. FIFO no toma en consideración la cantidad de veces que se han usado y las intercambia de todas formas.

El algoritmo del **Menos Usado Recientemente** (*Least Recently Used - LRU*) mantiene el registro de la última vez que fue usada cada página, no cuando fue cargada en memoria. El *hardware* de administración de la memoria usa un contador que se incrementa durante cada referencia a memoria. Cada entrada de la tabla de páginas tiene un campo que almacena el valor del contador. Cuando se hace referencia a una página se almacena el valor del contador en la correspondiente entrada de la tabla de páginas. El algoritmo LRU busca en la tabla una entrada con el valor de contador más bajo y selecciona esa página para reemplazo.

En un sistema de n marcos de página, una matriz $n\,x\,n$provee un método alternativo para implementar el algoritmo LRU. A la matriz se la inicializa con ceros, cuando se accede el marco de página k todos los bits de la fila k se ponen en 1, luego todos los bits en la columna k se ponen en cero. Si la fila i contiene el valor binario más bajo entonces el marco de página i es el menos usado recientemente.

El algoritmo **No Usado Recientemente** (*Not Recently Used - NRU*) es una aproximación a LRU. Además del bit sucio, cada entrada de la tabla de páginas contiene el **bit de referencia**. Cuando se carga una página, el sistema operativo pone el bit de referencia en cero. Cuando una página es accedida, el *hardware* pone el bit en 1. Además, a intervalos periódicos el sistema operativo pone todos los bits de referencia en la tabla de páginas de vuelta en cero. Un valor de cero en el bit de referencia significa que no ha sido referenciado "recientemente" y un 1 significa que sí.

Las entradas pueden dividirse en conjuntos dependiendo del valor en el bit de referencia y el bit sucio. La primera prioridad es seleccionar una página con su bit de referencia en cero. La segunda prioridad es elegir una página cuyo bit sucio sea cero, ya que intercambiar a almacenamiento una página sucia consume más tiempo que intercambiar una página limpia. El sistema operativo es libre de elegir cualquier página a partir de varias páginas con los mismos valores en el bit de referencia o sucio.

Otra aproximación a LRU llamada **envejecimiento** (*aging*) se logra al agregar un byte de referencia a las entradas de la tabla de páginas. Cuando se accede una página, el *hardware* pone en 1 el bit más significativo en el byte de referencia. Como en NRU, se usan interrupciones de temporizador (*timer interrupts*) para activar una rutina del sistema operativo, a pesar de que el intervalo de interrupción típicamente es menor en este algoritmo que en NRU. El sistema operativo desplaza todos los bits a la derecha un lugar en los bytes de referencia. Se selecciona para eliminar a una página con el valor binario más bajo en su byte de referencia. Tal como en NRU, el sistema operativo puede elegir cualquier página entre todas las páginas con el valor más bajo.

El algoritmo de reemplazo de la **segunda oportunidad** (*Second Chance*) es una combinación de los algoritmos FIFO y NRU. Selecciona a la página más vieja como candidata para remover y la remueve si su bit de referencia es cero. Sin embargo, si su bit de referencia es 1, su hora de carga se restablece a la hora actual y su bit de referencia se pone en cero. El algoritmo luego se repite con la segunda página más vieja siendo ahora la más vieja. El proceso continúa hasta que es seleccionada una página.

3.4.6. El rendimiento de los algoritmos

El rendimiento de los diferentes algoritmos se puede comparar dadas diferentes secuencias de referencias de memoria. A la lista de páginas accedidas por un proceso se la llama **cadena de referencia** (*reference string*), tal como vimos. Dados un algoritmo, una cadena de referencia y el número de marcos de página, se puede determinar el número de fallos de página disparados y éste se puede usar como la base para comparar los algoritmos.

También se puede comparar el rendimiento de un algoritmo en particular dados diferentes números de marcos de página asignados a un proceso. Uno podría esperar que al incrementar el número de marcos de página debería provocar la misma o menor fallos de página. Pero éste no es siempre el caso, bajo ciertas circunstancias ocurre una situación llamada **la anomalía de Belady** [Belady, 1969] en la que a mayor marcos de página ocurren más fallos de página. El algoritmo óptimo no sufre de esta anomalía; el LRU tampoco. En cambio, puede ocurrir en el FIFO.

3.4.7. Políticas de asignación

El *hardware* de la máquina es el que dicta el número mínimo absoluto de marcos que se le pueden asignar a cualquier proceso. Cuando ocurre un fallo de página se carga la página necesaria en memoria y la instrucción se ejecuta de nuevo. Si la instrucción hace referencia a dos direcciones pero el proceso es asignado solo un único marco, ocurrirá un ciclo infinito de fallos de página. En las máquinas con direccionamiento indirecto y con instrucciones con operandos múltiples, una única instrucción puede hacer referencia a direcciones en varios marcos de página.

Por motivos de rendimiento, el número mínimo de marcos por proceso típicamente es mayor que el mínimo absoluto que dicta el *hardware*. No obstante la anomalía de Belady, por lo general el rendimiento mejora a medida que se incrementa el número de marcos de página asignados. Cualquiera que sea el mínimo, se puede lograr sólo mediante la limitación del número de procesos asignados a marcos de página. Esta es la tarea que realiza el planificador de largo plazo, ya que el planificador de largo plazo limita el número de procesos que comparten marcos de página, de manera tal que la frecuencia de fallos de página dan un nivel aceptable de rendimiento.

Es el sistema operativo el que debe decidir cuantos marcos de página asignar a cada proceso. Una estrategia llamada **distribución equitativa** (*equal allocation*) divide a los marcos de página disponibles en partes iguales entre los procesos.

Dados m marcos y p procesos, la distribución equitativa divide 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.

Pero la limitación radica en 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.

La **distribución proporcional** (*proportional allocation*) divide a los marcos en proporción a los tamaños de los procesos. 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, se depende del 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. Es decir, podrían otorgársele más marcos a un proceso de mayor prioridad para que finalice antes. En algunos sistemas se plantea la asignación proporcional en la que la variable es la prioridad y no el tamaño del proceso, o en combinación, el tamaño y la prioridad. También un proceso de mayor prioridad podría usar para su reemplazo un marco de un proceso de prioridad menor.

3.4.7.1. 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

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

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

En ambos casos, a cada proceso se le asigna un número fijo de marcos. La asignación fija puede usarse con un algoritmo de reemplazo de **ámbito local** (*local scope*). Un algoritmo de reemplazo de ámbito local selecciona para remover a una página perteneciente al proceso que generó el fallo de página.

Alternativamente, un sistema operativo puede emplear un algoritmo de **ámbito global** (*global scope*) y **asignación variable** (*variable allocation*). En un sistema así, cuando el algoritmo de reemplazo busca una página para remover, selecciona la página a partir de todos los procesos en la máquina. El número de marcos asignados a un proceso variaría ya que se miden con respecto a las páginas de otros procesos. Si el algoritmo selecciona una página para remover que haría caer al proceso por debajo de la asignación mínima, o bien se selecciona otra página o se intercambian a almacenamiento secundario todas las páginas del proceso.

Una estrategia de compromiso es utilizar asignación variable con ámbito local. El número de páginas asignadas a un proceso varía de acuerdo con las necesidades del proceso. Sin embargo, la página a ser reemplazada se selecciona siempre a partir del proceso que está ejecutando actualmente. El número de páginas asignadas a un proceso típicamente está basado en el **conjunto de trabajo** (*working set*) del proceso.

3.4.7.2. Hiperpaginación

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.

> Al rendimiento degradado motivado por el intercambio excesivo se lo conoce como **hiperpaginación**. (*thrashing*, literalmente "paliza" o "fustigamiento"). Un proceso está hiperpaginando si pasa más tiempo paginando que ejecutando.

3.4.8. Conjunto de trabajo

[Denning, 1968] Define al conjunto de trabajo de un proceso en un determinado tiempo como al conjunto de páginas referenciadas en un cierto intervalo de tiempo precedente. Con frecuencia se expresa el conjunto de trabajo usando la notación funcional $W(t, \Delta)$ donde W representa al conjunto de trabajo, t representa el tiempo y Δ representa el intervalo. Usualmente es más simple para el tiempo el ser medido en términos de instrucciones ejecutadas (una unidad de tiempo de uno significa una instrucción ejecutada).

El objetivo es elegir un valor para Δ de manera tal que el conjunto de trabajo refleje la localidad del programa. Si el valor de Δ es demasiado pequeño, el conjunto de trabajo no incluirá todas las páginas en la localidad actual. Si el valor de Δ es demasiado grande, el conjunto de trabajo incluirá páginas de una localidad previa.

A pesar de que el conocimiento de las páginas del conjunto de trabajo de un proceso es de valor insignificante para el sistema operativo, el conocimiento del tamaño del conjunto de trabajo se puede usar para determinar cuántas páginas asignar a cada proceso. El número de procesos asignados a memoria se ajusta de manera tal que a cada proceso se le asigna el número de marcos indicados por el tamaño del conjunto de trabajo.

3.4.9. Prepaginado

Si el sistema operativo conoce el conjunto de trabajo al momento de que el proceso fue intercambiado a almacenamiento secundario, puede **prepaginar** (*prepage*) todas las páginas en ese conjunto de trabajo cuando el proceso sea intercambiado nuevamente a memoria. El prepaginado previene que la referencia inicial a las páginas del conjunto de trabajo generen un fallo de página. Esto le

ahorra al sistema operativo la sobrecarga extra de procesar esos fallos de página. Sin embargo, algunas de las páginas cargadas pueden nunca ser referenciadas. Prepaginar una página que no es referenciada degrada el rendimiento de la entrada/salida y desperdicia marcos de página.

También se puede usar el prepaginado al momento de un fallo de página. Además de cargar la página que generó el fallo de página, también se puede cargar a la página siguiente en memoria virtual. Para aquellos programas cuya ejecución es primordialmente secuencial, el acceso a una página puede ser un predictor de un acceso a la página siguiente. El costo en tiempo por intercambiar ambas páginas es mínimo ya que la latencia del disco y el tiempo de búsqueda son los mismos ya sea que se hayan intercambiado una única página o dos páginas contiguas. La mayor desventaja de esta forma de prepaginado son los marcos desperdiciados por páginas que son cargadas pero nunca referenciadas. A causa de esta desventaja, esta técnica no ha mostrado ser efectiva en general.

3.4.10. Segmentación

Así como la paginación simple puede modificarse para crearse paginación por demanda, la segmentación también se puede modificar para crear segmentación por demanda. Sin embargo, la variabilidad de los tamaños de los segmentos complica muchos de los problemas encontrados en la paginación por demanda. En la decisión del intercambio, el tamaño de los segmentos se convierte en un factor en la decisión acerca de qué segmentos intercambiar. Toda la noción del conjunto de trabajo se distorsiona por los tamaños diferentes de los segmentos.

Una forma mucho más práctica de implementar segmentación en un sistema de memoria virtual es combinarlo con paginación por demanda. O bien los segmentos siempre se consideran intercambiados en memoria, o bien el estado de intercambio de un segmento depende de si su tabla de páginas está intercambiada en memoria. Por lo tanto, la manipulación de los segmentos trabaja básicamente del mismo modo que lo hace en un sistema de paginación y segmentado simples.

> La capacidad de memoria virtual se crea por paginación por demanda de los segmentos.

3.5. Ejercicios

3.5.1. Ejercicio 1

Consideren un sistema con swapping 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)

3.5.2. 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

3.5.3. 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.

3.5.4. 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)

3.5.5. 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.

3.5.6. 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 Óptimo, suponiendo que se disponen de 2, 3, 4, 5 o 6 frames?

3.5.7. 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 Óptimo?
- ¿Cuál con segunda chance?

3.5.8. 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
- Óptimo

3.6. Trabajos prácticos

3.6.1. Práctica con Linux

3.6.1.1. Estadísticas de la memoria virtual en Linux

El término "intercambio" (*swap*) se refiere estrictamente a guardar el proceso entero, no una parte, al disco[3]. Esta forma de intercambio es rara de encontrar, antes bien se usa en combinación con la paginación.

Un *page-in* es un hecho común, normal y no es motivo de preocupación. Cuando arranca una aplicación, su imagen se va cargando en memoria haciendo *page-in*, esto es normal. Pero el *page-out* puede ser señal de problemas. Cuando el núcleo detecta que se está quedando sin memoria intenta liberarla haciendo *page-out*. Si bien esto puede ocurrir brevemente y de tanto en tanto, si esta situación se vuelve constante podemos caer en hiperpaginación.[Tanaka, 2005]

El programa `vmstat`, tal como su nombre sugiere, muestra estadísticas de la memoria virtual. Cuánta memoria virtual hay, cuánta está libre y la actividad de paginación. Se puede observar los *page-in* y *page-out* a medida que ocurren.

Es mejor si nos muestra la tabla actualizándola periódicamente, el retardo o *delay* se lo podemos pasar como primer parámetro y como segundo el conteo para que no nos lo muestre indefinidamente, por ejemplo coloque el comando:

```
vmstat 5 10
```

Nos muestra 10 líneas de estadísticas tomadas cada 5 segundos. Las columnas están explicadas en el manual (`man vmstat`) pero las que nos interesan son `free`, `si` y `so`. La columna `free` nos muestra la cantidad de memoria libre, `si` nos muestra los *page-in* y `so` los *page-out*, nos mostrará una salida similar a ésta:

```
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 1183880 104564 545096    0    0    56    15  344  365  3  2 92  2  0
 0  0      0 1179308 104572 549592    0    0     0     4  272  329  1  1 98  0  0
 0  0      0 1183572 104580 545112    0    0     0     7  247  266  0  0 98  1  0
 2  0      0 1183612 104592 545092    0    0     0     8  249  285  0  0 98  1  0
 0  0      0 1183644 104592 545080    0    0     0     0  214  232  0  0 99  0  0
 1  0      0 1183652 104592 545072    0    0     0    18  266  314  1  0 99  0  0
 0  0      0 1183652 104600 545108    0    0     0    14  208  265  1  0 98  1  0
 0  0      0 1183768 104600 545064    0    0     0     1  212  276  1  0 99  0  0
 0  0      0 1183644 104604 545072    0    0     0     1  190  256  1  0 98  1  0
 0  1      0 1173544 104612 553040    0    0  1589    29  539  501  3  2 90  5  0
```

Nos muestra que hay suficiente memoria libre, como en este momento no estamos comenzando a ejecutar una aplicación nueva, no vemos *page-in*, luego tampoco vemos *page-outs*. Pero si comenzamos a ejecutar varias aplicaciones que requieran de mucha memoria y colocamos nuevamente el mismo comando la situación cambiará y nos mostrará una salida similar a ésta:

```
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0  13344   1444   1308  19692    0  168   129    42 1505  713 20 11  69
 1  0  13856   1640   1308  18524   64  516   379   129 4341  646 24 34  42
 3  0  13856   1084   1308  18316   56   64    14     0  320 1022 84  9   8
```

[3]Cabría aclarar en este punto que Linux normalmente usa una **partición** para intercambio, en tanto que Windows utiliza **archivos**.

Note los valores no nulos de `so`, están indicando que no hay suficiente memoria física y el núcleo está generando *page-outs*. Como vimos en la Práctica 2.7.1.1, podemos usar `top` y `ps` para ver cuáles son los procesos que más recursos están consumiendo. Pero también podemos usar el mismo programa `vmstat` para ver otras estadísticas interesantes. Instamos al lector que experimente libremente una vez que haya leído las páginas del manual.

En Linux contamos con los comandos administrativos `swapon` y `swapoff` para activar y desactivar respectivamente dispositivos o archivos definidos como áreas de intercambio.

En el pseudo sistema de archivos `/proc` tenemos al archivo `/proc/swaps` que mide el espacio de intercambio y su utilización. Por ejemplo, en un sistema con una única partición de intercambio si colocamos este comando:

```
$ cat /proc/swaps
```

la salida será similar a ésta:

```
Filename          Type            Size    Used    Priority
/dev/sda5         partition       2104472 0       -1
```

En el que podemos ver el nombre del archivo de intercambio, el tipo de espacio de intercambio, el tamaño total y la cantidad de espacio en uso, medidos en kilobytes. La columna de `Prioridad` es útil cuando se usan varios archivos o particiones de intercambio, a menor número mayor es la preferencia de uso.

3.6.1.2. El programa mapa.c

El siguiente programa ilustra los segmentos

```
#include <stdio.h>
#include <stdlib.h>
int uig;
int ig=5;
int func()
{

     return 0;

}
int main()
{

     int local;
     int *ptr;
     ptr=(int *) malloc(sizeof(int));
     printf(“Una direccion del BSS: %p\n”, &uig);
     printf(“Una direccion del segmento de datos: %p\n”, &ig);
     printf(“Una direccion del segmento de codigo: %p\n”, &func);
     printf(“Una direccion del segmento de pila: %p\n”, &local);
     printf(“Una direccion del monticulo: %p \n”, ptr);
     printf(“Otra direccion de la pila: %p\n”, %ptr);
     free(ptr);
     return 0;

}
```

3.6.1.3. La estructura de ELF

Traducir y simplificar este articulo, tal vez el programa podría ser el famoso HolaMundo, plantear qué pasaría si en vez de pasarlo como cadena dentro del printf, declararlo como una variable local y como una variable global.

3.6.1.4. 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.

3.6.2. Práctica con Windows

3.6.2.1. Memoria virtual en Windows

> Si bien cada proceso en Windows tiene su propio espacio de memoria privado, el código en modo núcleo del sistema operativo y los manejadores de dispositivos comparten un espacio de direcciones virtuales único. Cada página en la memoria virtual está rotulada con el modo de acceso en el que el procesador debe estar para leer y/o escribir la página. Las páginas en el espacio del sistema pueden accederse sólo desde el modo núcleo mientras que todas las páginas en el espacio de direcciones del usuario son accesibles desde el modo usuario del procesador. Las páginas de sólo lectura, tales como las que contienen código ejecutable, no son escribibles desde ningún modo. [Russinovich, 2005]

Si en Windows hacemos click con el botón derecho del mouse sobre el icono "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 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

3.6.2.2. La estructura de PE

3.6.2.3. PhysMem

Con el fin de demostrar la capacidad de ver la memoria física y de ofrecer la oportunidad de navegar a través de la RAM de su computadora, Mark Russinovich escribió PhysMem. Es un programa de consola para Win32 que abrirá la

sección de memoria física y volcará los contenidos de regiones (en hexadecimal y ASCII) que le solicite en una interfaz simple de línea de comando.

Al navegar por la memoria, algunos lugares de interés a los que les gustaría darles una mirada es en la dirección 0x1000, que es donde está ubicado NTLDR (*NT Loader* - Cargador de NT) y en el rango 0xF9000-0xFFFFF, que es donde está proyectada la ROM BIOS.

PhysMem usa la API nativa para abrir y proyectar `\Device\PhysicalMemory` porque ese nombre es inaccesible a través de la API Win32. También usa la API nativa (todas documentadas en el Windows NT DDK - *Driver Development Kit*) para proyectar vistas de la sección, a pesar de que esto se podría haber hecho usando Win32.

Puede descargar PhysMem desde:
`http://download.sysinternals.com/Files/PhysMem.zip`

3.6.2.4. CacheSet

CacheSet es un applet que le permite manipular los parámetros del conjunto de trabajo de la cache de archivos del sistema. Además de darle la capacidad de controlar los tamaños mínimos y máximos del conjunto de trabajo, también le permite restablecer el conjunto de trabajo de la Cache, forzándolo a crecer tanto como sea necesario desde un punto de partida mínimo.

3.7. 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 o fustigamiento (*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.