4

Gestión de memoria

La memoria es uno de los recursos más importantes de la computadora y, en consecuencia, la parte del sistema operativo responsable de tratar con este recurso, el gestor de memoria, es un componente básico del mismo. El gestor de memoria del sistema operativo debe hacer de puente entre los requisitos de las aplicaciones y los mecanismos que proporciona el hardware de gestión de memoria. Se trata de una de las partes del sistema operativo que está más ligada al hardware. Esta estrecha colaboración ha hecho que tanto el hardware como el software de gestión de memoria hayan ido evolucionando juntos. Las necesidades del sistema operativo han obligado a los diseñadores del hardware a incluir nuevos mecanismos que, a su vez, han posibilitado el uso de nuevos esquemas de gestión de memoria. De hecho, la frontera entre la labor que realiza el hardware y la que hace el software de gestión de memoria es difusa y ha ido también evolucionando.

Por lo que se refiere a la organización del capítulo, en primer lugar se presentarán los requisitos que debe cumplir la gestión de memoria en un sistema con multiprogramación. A continuación, se mostrarán las distintas fases que conlleva la generación de un ejecutable y se estudiará cómo es el mapa de memoria de un proceso. En las siguientes secciones, se analizará cómo ha sido la evolución de la gestión de la memoria, desde los sistemas multiprogramados más primitivos hasta los sistemas actuales basados en la técnica de memoria virtual. Por último, se presentará el concepto de proyección de archivos y se estudiarán algunos de los servicios POSIX y Win32 de gestión de memoria. El índice del capítulo es el siguiente:

- *Objetivos del sistema de gestión de memoria.*
- *Modelo de memoria de un proceso.*
- *Esquemas de memoria basados en asignación contigua.*
- *Intercambio.*
- *Memoria virtual.*
- *Archivos proyectados en memoria.*
- *Servicios de gestión de memoria.*

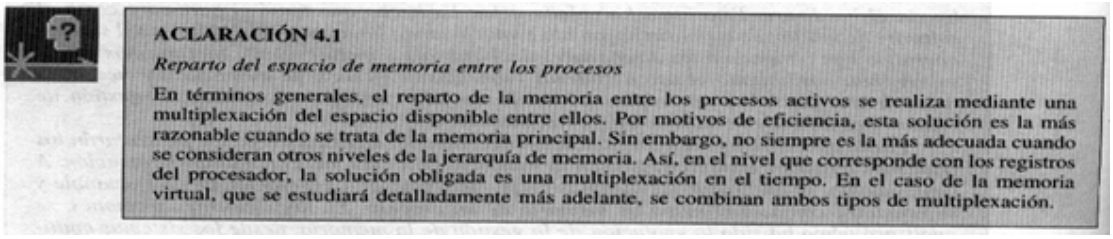
4.1. OBJETIVOS DEL SISTEMA DE GESTIÓN DE MEMORIA

En un sistema con multiprogramación, el sistema operativo debe encargarse de realizar un reparto transparente, eficiente y seguro de los distintos recursos de la máquina entre los diversos procesos, de forma que cada uno de ellos crea que «tiene una máquina para él solo». Esto es, el operativo debe permitir que los programadores desarrollen sus aplicaciones sin verse afectados por la posible coexistencia de su programa con otros durante su ejecución.

Como se ha analizado en capítulos anteriores, en el caso del procesador esta multiplexación se logra almacenando en el bloque de control de cada proceso el contenido de los registros del procesador correspondientes a dicho proceso, salvándolos y restaurándolos durante la ejecución del mismo.

En el caso de la memoria, el sistema operativo, con el apoyo del hardware de gestión memoria del procesador, debe repartir el almacenamiento existente proporcionando un espacio de memoria independiente para cada proceso y evitando la posible interferencia voluntaria o involuntaria de cualquier otro proceso.

Se podría considerar que, en el caso del procesador, se realiza un reparto en el tiempo, mientras que en el de la memoria, se trata de un reparto en el espacio (Aclaración 4.1). La acción combinada de estos dos mecanismos ofrece a los programas una abstracción de procesador virtual que les independiza del resto de los procesos.



ACLARACIÓN 4.1

Reparto del espacio de memoria entre los procesos

En términos generales, el reparto de la memoria entre los procesos activos se realiza mediante una multiplexación del espacio disponible entre ellos. Por motivos de eficiencia, esta solución es la más razonable cuando se trata de la memoria principal. Sin embargo, no siempre es la más adecuada cuando se consideran otros niveles de la jerarquía de memoria. Así, en el nivel que corresponde con los registros del procesador, la solución obligada es una multiplexación en el tiempo. En el caso de la memoria virtual, que se estudiará detalladamente más adelante, se combinan ambos tipos de multiplexación.

Sea cual sea la política de gestión de memoria empleada en un determinado sistema, se pueden destacar las siguientes características como objetivos deseables del sistema de gestión de memoria:

- Ofrecer a cada proceso un espacio lógico propio.
- Proporcionar protección entre los procesos.
- Permitir que los procesos compartan memoria.
- Dar soporte a las distintas regiones del proceso.
- Maximizar el rendimiento del sistema.
- Proporcionar a los procesos mapas de memoria muy grandes.

Espacios lógicos independientes

En un sistema operativo multiprogramado de propósito general no se puede conocer a *priori* 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 tanto, en sucesivas ejecuciones del mismo.

El código maquina de un programa contenido en un archivo ejecutable incluirá referencias a memoria, utilizando los diversos modos de direccionamiento del juego de instrucciones del proce

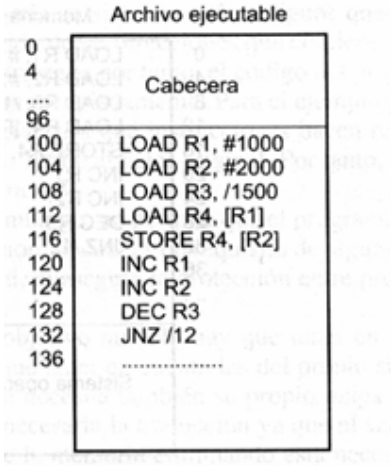


Figura 4.1, *Archivo ejecutable hipotético.*

sador, tanto para acceder a sus operandos como para realizar bifurcaciones en la secuencia de ejecución. Estas referencias típicamente estarán incluidas en un intervalo desde 0 hasta un valor máximo *N*.

Supóngase, por ejemplo, un fragmento de un programa que copia el contenido de un vector almacenado a partir de la dirección 1000 en otro almacenado a partir de la 2000, estando el tamaño del vector almacenado en la dirección 1500. El código almacenado en el archivo ejecutable, mostrado en un lenguaje ensamblador hipotético, seria el mostrado en la Figura 4.1, donde se ha supuesto que la cabecera del ejecutable ocupa 100 bytes y que cada instrucción ocupa 4. Observe que, evidentemente, tanto en el ejecutable como posteriormente en memoria, se almacena realmente código máquina. En esta figura y en las posteriores se ha preferido mostrar un pseudocódigo ensamblador para facilitar la legibilidad de las mismas.

El código maquina de ese programa incluye referencias a direcciones de memoria que corresponden tanto a operandos (las direcciones de los vectores y su tamaño) como a instrucciones (la etiqueta de la bifurcación condicional, JNZ).

En el caso de un sistema con monoprogramación, para ejecutar este programa sólo será necesario cargarlo a partir de la posición de memoria 0 y pasarle el control al mismo. Observe que, como se puede apreciar en la Figura 4.2, se esta suponiendo que el sistema operativo estará cargado en la parte de la memoria con direcciones más altas.

En un sistema con multiprogramación es necesario realizar un proceso de traducción (**reubicación**) de las direcciones de memoria a las que hacen referencia las instrucciones de un programa (**direcciones lógicas**) para que se correspondan con las direcciones de memoria principal asignadas al mismo (**direcciones físicas**). En el ejemplo planteado previamente, si al programa se le asigna una zona de memoria contigua a partir de la dirección 10000, habría que traducir todas las direcciones que genera el programa añadiéndoles esa cantidad.

Este proceso de traducción crea, por tanto, **un espacio lógico** (o **mapa**) independiente para cada proceso proyectándolo sobre la parte correspondiente de la memoria principal de acuerdo con una función de traducción:

Traducción(dirección lógica) → dirección física

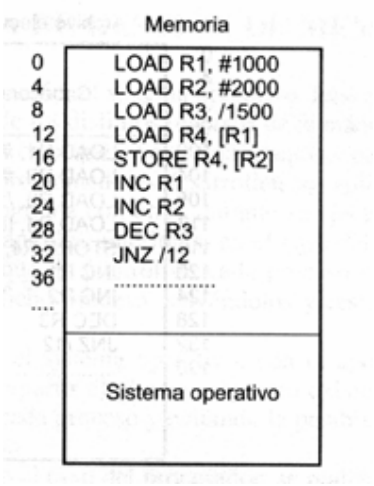


Figura 4.2. *Ejecución del programa en un sistema con monoprogramación.*

Dado que hay que aplicar esta función a cada una de las direcciones que genera el programa, es necesario que sea el procesador el encargado de realizar esta traducción. Como se explicó en el Capitulo 1, la función de traducción la lleva a cabo concretamente un módulo específico del procesador denominado unidad de gestión de memoria (MMU, *Memory Management Unit)*. El sistema operativo deberá poder especificar a la MMU del procesador que función de traducción debe aplicar al proceso que está ejecutando en ese momento. En el ejemplo planteado, el procesador, a instancias del sistema operativo, debería sumarle 10000 a cada una de las direcciones que genera el programa en tiempo de ejecución. Observe que el programa se cargaría en memoria desde el ejecutable sin realizar ninguna modificación del mismo y, como se muestra en la Figura 4.3, seria durante su ejecución cuando se traducirían adecuadamente las direcciones generadas. El sistema operativo debería almacenar 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.

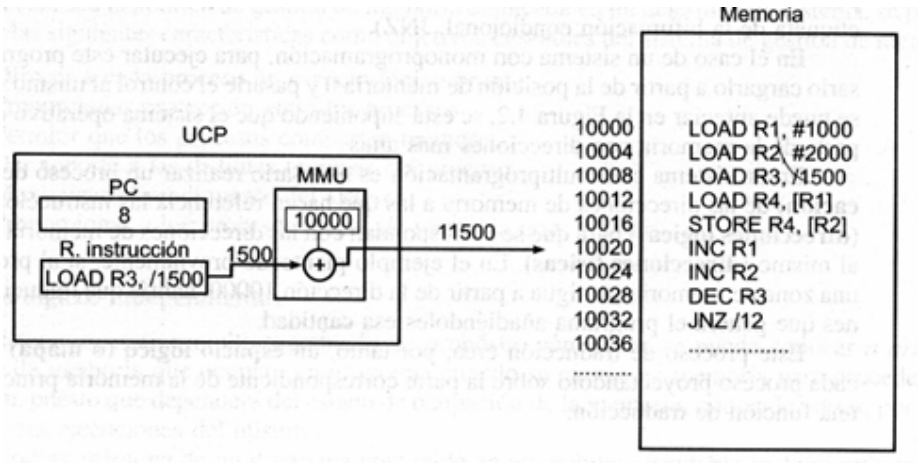


Figura 4.3. *Ejecución en un sistema con reubicación hardware.*

Una alternativa software, usada en sistemas más antiguos que no disponían de hardware de traducción, es realizar la reubicación de las direcciones que contiene el programa en el momento de su carga en memoria. Con esta estrategia, por tanto, el código del programa cargado en memoria ya contiene las direcciones traducidas apropiadamente. Para el ejemplo planteado, durante su carga en memoria, el sistema operativo detectaría que instrucciones hacen referencia a direcciones de memoria y las modificaría añadiendo 10000 al valor original. Por tanto, e1 código cargado en memoria resultaría el mostrado en la Figura 4.4.

Esta segunda solución, sin embargo, no permite que el programa o un fragmento del mismo se pueda mover en tiempo de ejecución, lo cual es un requisito de algunas de las técnicas de gestión de memoria como la memoria virtual, ni asegura la protección entre procesos como se apreciara en el siguiente apartado.

A la hora de analizar este objetivo no solo hay que tener en cuenta las necesidades de los procesos, sino que también hay que tener en cuenta las del propio sistema operativo. Al igual que los procesos, el sistema operativo necesita también su propio mapa de memoria. En este caso, sin embargo, no sería estrictamente necesaria la traducción ya que el sistema operativo puede situarse de forma contigua al principio de la memoria eliminando esta necesidad. De cualquier manera, el uso de una función de traducción puede ser interesante puesto que proporciona mas flexibilidad.

Como gestor de los recursos del sistema, el sistema operativo no sólo requiere utilizar su mapa sino que necesita acceder a toda la memoria del sistema. Es importante resaltar que, además de poder acceder a la memoria física, el sistema operativo necesita «ver» el espacio lógico de cada proceso. Para aclarar esta afirmación, considérese, por ejemplo, un proceso que realiza una llamada al sistema en la que pasa como parámetro la dirección donde esta almacenada una cadena de caracteres que representa el nombre de un archivo. Observe que se tratará de una dirección lógica dentro del mapa del proceso y que, por tanto, el sistema operativo, ya sea con la ayuda de la MMU o «a mano», deberá transformar esa dirección usando la función de traducción asociada al proceso.

Protección

En un sistema con monoprogramación es necesario proteger al sistema operativo de los accesos que realiza el programa en ejecución para evitar que, voluntaria o involuntariamente, pueda interferir en

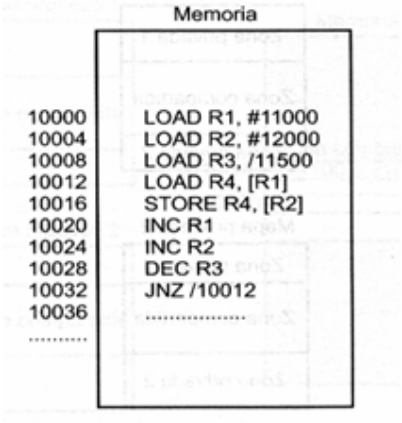


Figura 4.4. *Ejecución en un sistema con reubicación software*

el correcto funcionamiento del mismo. Todos los usuarios que han trabajado en un sistema que no cumple este requisito de protección, como por ejemplo MS-DOS, han experimentado cómo un error de programación en una aplicación puede causar que todo el sistema se colapse durante la ejecución de la misma al producirse una alteración imprevista del código o las estructuras de datos del sistema operativo.

En un sistema con multiprogramación el problema se acentúa ya que no sólo hay que proteger al sistema operativo sino también a los procesos entre sí. El mecanismo de protección en este tipo de sistemas necesita del apoyo del hardware puesto que es necesario validar cada una de las direcciones que genera un programa en tiempo de ejecución. Este mecanismo está típicamente integrado en el mecanismo de traducción: la función de traducción debe asegurar que los espacios lógicos de los procesos sean disjuntos entre sí y con el del propio sistema operativo.

Observe que en el caso de un sistema que use un procesador con mapa de memoria y E/S común, el propio mecanismo de protección integrado en el proceso de traducción permite impedir que los procesos accedan directamente a los dispositivos de E/S, haciendo simplemente que las direcciones de los dispositivos no formen parte del mapa de ningún proceso.

Compartimiento de memoria

Para cumplir el requisito de protección, el sistema operativo debe crear espacios lógicos independientes y disjuntos para los procesos. Sin embargo, en ciertas situaciones, bajo la supervisión y control del sistema operativo, puede ser provechoso que los procesos puedan compartir memoria. Esto es, la posibilidad de que direcciones lógicas de dos o más procesos, posiblemente distintas entre sí, se correspondan con la misma dirección física. Nótese que, como puede observarse en la Figura 4.5, la posibilidad de que dos o más procesos compartan una zona de memoria implica que el sistema de gestión de memoria debe permitir que la memoria asignada a un proceso no sea contigua. Así, por ejemplo, una función de traducción como la comentada anteriormente, que únicamente sumaba una cantidad a las direcciones generadas por el programa, obligaría a que el espacio asignado al proceso fuera contiguo, imposibilitando, por tanto, el poder compartir memoria.

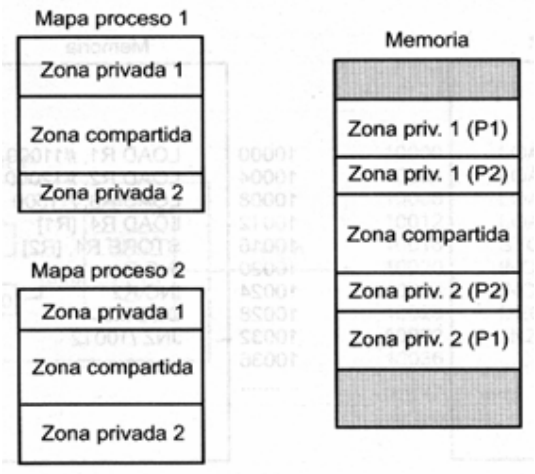


Figura 4.5.. *Dos procesos compartiendo una zona de memoria.*

El compartimiento de memoria puede ser beneficioso en varios aspectos. Por un lado, permite que cuando se estén ejecutando múltiples instancias del mismo programa (p. ej.: el intérprete de los mandatos), los procesos correspondientes compartan el código, lo que resulta

en un mejor aprovechamiento de la memoria disponible. Adicionalmente, como se analizará en el Capitulo 5, la memoria compartida permite una forma de comunicación muy rápida entre procesos cooperantes.

La posibilidad de que los procesos compartan una zona de memoria haciéndola corresponder con rangos diferentes de su espacio lógico presenta problemas en determinadas circunstancias. Considérese el caso de que una posición de memoria de la zona compartida tenga que contener la dirección de otra posición de dicha zona (o sea, una referencia). Como se puede apreciar en la Figura 4.6, no es posible determinar qué dirección almacenar en la posición origen puesto que cada proceso ve la posición referenciada en una dirección diferente de su mapa de memoria.

Esta situación se puede presentar tanto en zonas compartidas de código como de datos. En el caso del código, considérese que una posición de la zona contiene una instrucción de bifurcación a otra posición de la misma. Por lo que respecta al caso de los datos, supóngase que la zona compartida contiene una estructura de lista basada en punteros.

Soporte de las regiones del proceso

Como se analizará con más detalle en secciones posteriores, el mapa de un proceso no es homogéneo sino que está formado por distintos tipos de regiones con diferentes características y propiedades. Dado que el sistema operativo conoce que regiones incluye el mapa de memoria de cada proceso, el gestor de memoria con el apoyo del hardware debería dar soporte a las características especificas de cada región.

Así, por ejemplo, el contenido de la región que contiene el código del programa no debe poder modificarse. Se debería detectar cualquier intento de escritura sobre una dirección incluida en dicha región y tratarlo adecuadamente (p. ej.: mandando una señal al proceso). Observe que lo que se está detectando en este caso es un error de programación en la aplicación. El dar soporte al carácter de solo lectura de la región permite detectar inmediatamente el error, lo que facilita considerablemente la depuración del programa.

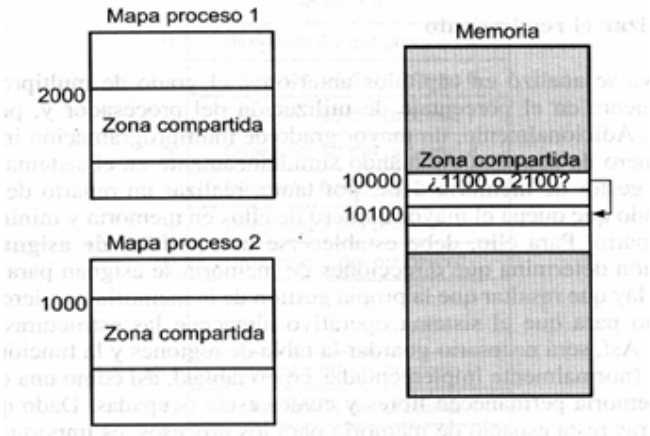


Figura 4.6. Problemas al compartir una zona en diferentes direcciones del mapa de cada proceso.

Otro aspecto a resaltar es que el mapa del proceso no es estático. Durante la ejecución de un programa puede variar el tamaño de una región o, incluso, pueden crearse nuevas regiones o eliminarse regiones existentes. El sistema de memoria debe controlar qué regiones están presentes en el mapa de memoria y cual es el tamaño actual de cada una. Tómese como ejemplo el comportamiento dinámico de la región de pila del proceso. El gestor de memoria debe asignar y liberar memoria para estas regiones según evolucionen la. mismas. Este carácter dinámico del

mapa de un proceso implica la existencia de zonas dentro del mapa de memoria del proceso que en un determinado momento de su ejecución no pertenecen a ninguna región y que, por tanto, cualquier acceso a las mismas implica un error de programación. El gestor de memoria debería detectar cuándo se produce un acceso a las mismas y, como en el caso anterior, tratarlo adecuadamente. Con ello además de facilitar la depuración, se elimina la necesidad de reservar memoria física para zonas del mapa que no estén asignadas al proceso en un determinado instante sistema operativo debe almacenar por cada proceso una **tabla de regiones** que indique las características de las regiones del proceso (tamaño, protección, etc.),

De acuerdo con los aspectos que se acaban de analizar, la función de traducción se generaliza para que tenga en cuenta las situaciones planteadas pudiendo, por tanto, devolver tres valores alternativos:

Traducción(dirección lógica tipo de operación) →
(Dirección física correspondiente,
Excepción por acceso a una dirección no asignada,
Excepción por operación no permitida}

Por último, hay que resaltar que el mapa de memoria del propio sistema operativo, como programa que es, también está organizado en regiones. Al igual que ocurre con los procesos de usuario, el sistema de memoria debería dar soporte a las regiones del sistema operativo. Así, se detectaría inmediatamente un error en el código del sistema operativo, lo que facilitaría su depuración a la gente encargada de esta ardua labor. Observe que 1a detección del error debería detener inmediatamente la ejecución del sistema operativo y mostrar un volcado de la información del sistema.

Maximizar el rendimiento

Como ya se analizó en capítulos anteriores, el grado de multiprogramación del sistema influye directamente en el porcentaje de utilización del procesador y, por tanto, en el rendimiento del sistema. Adicionalmente, un mayor grado de multiprogramación implica que puede existir un mayor numero de usuarios trabajando simultáneamente en el sistema.

El gestor de memoria debe, por tanto, realizar un reparto de la memoria entre los procesos intentando que quepa el mayor numero de ellos en memoria y minimizando el desperdicio inherente al reparto. Para ello, debe establecerse una **política de asignación** adecuada. La política de asignación determina qué direcciones de memoria se asignan para satisfacer una determinada petición. Hay que resaltar que la propia gestión de la memoria requiere un gasto en espacio de almacenamiento para que el sistema operativo almacene las estructura. de datos implicadas en dicha gestión. Así, será necesario guardar la tabla de regiones y la función de traducción asociada a cada proceso (normalmente implementadas como tablas), así como una estructura que refleje qué partes de la memoria permanecen libres y cuales están ocupadas. Dado que el almacenamiento de estas estructuras resta espacio de memoria para los procesos, es importante asegurar que su consumo se mantiene en unos términos razonables.

Así, por ejemplo, la situación óptima de aprovechamiento se obtendría si la función de traducción de direcciones permitiese hacer corresponder cualquier dirección lógica de un proceso con cualquier dirección física, como muestra la Figura 4.7. Observe que esto permitiría que cualquier palabra de memoria libre se pudiera asignar a cualquier proceso. Esta función de traducción es, sin embargo, irrealizable en la práctica puesto que implica tener unas tablas de traducción enormes, dado que habría que almacenar una entrada en la tabla por cada dirección de memoria del mapa de cada proceso.

Como ya se estudio en el Capitulo 1, la solución adoptada en la mayoría de los sistemas operativos actuales para obtener un buen aprovechamiento de la memoria, pero manteniendo las

tablas de traducción dentro de unos valores razonables, es la paginación que se volverá a tratar en las secciones siguientes.

Para optimizar el rendimiento, la mayoría de los sistemas operativos actuales no sólo intentan aprovechar la memoria lo mejor posible, sino que además utilizan la técnica de la memoria virtual presentada en el Capitulo 1, Con esta técnica, el grado de multiprogramación puede aumentar considerablemente, ya que no es preciso que todo el mapa de memoria del proceso tenga que residir en memoria principal para poder ejecutarse, sino que se irá trayendo de la memoria secundaria bajo demanda. Observe que el incremento del grado de multiprogramación no puede ser ilimitado puesto que, como ya se analizó en el Capitulo 1, cuando se supera un determinado umbral, el rendimiento del sistema decae drásticamente.

Como se analizara mas adelante, antes de la aparición de la memoria virtual, los sistemas operativos de tiempo compartido utilizaban una técnica denominada intercambio *(swapping).* Este mecanismo permite que en el sistema existan mas procesos de los que caben en memoria. Sin embargo, a diferencia de la memoria virtual, esta técnica sigue requiriendo que la imagen completa de un proceso resida en memoria para poder ejecutarlo. Los procesos que no caben en memoria en un determinado instante están suspendidos y su imagen reside en un dispositivo de almacenamiento secundario. Por tanto, el grado de multiprogramación sigue estando limitado por el tamaño de la memoria, por lo que el uso de esta técnica no redunda directamente en la mejoría del rendimiento del sistema. Sin embargo, proporciona un mejor soporte multiusuario, que fue el motivo principal para el que se concibió.

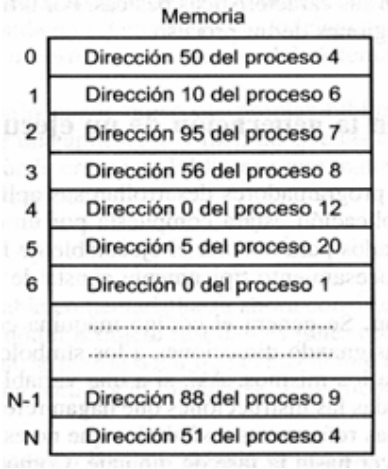


Figura 4.7. *Aprovechamiento óptimo de la memoria.*

Mapas de memoria muy grandes para los procesos

En los tiempos en los que 1a memoria era muy cara y, en consecuencia, los equipos poseían memoria bastante reducida, se producían habitualmente situaciones en las que las aplicaciones se veían limitadas por el tamaño de la memoria, Para solventar este problema, los programadores usaban la técnica de los *overlays.* Esta técnica consiste en dividir el programa en una serie de fases que se ejecutan sucesivamente, pero estando en cada momento residente en memoria sólo una fase. Cada fase se programa de manera que, después de realizar su labor, carga en memoria la siguiente fase y le cede el control. Es evidente que esta técnica no soluciona el problema de forma general dejando en manos del programador todo el trabajo.

Por ello, se ideo la técnica de la memoria virtual, que permite proporcionar a un proceso de

forma transparente un mapa de memoria considerablemente mayor que la memoria física existente en el sistema.

A pesar del espectacular abaratamiento de la memoria y la consiguiente disponibilidad generalizada de equipos con memorias apreciablemente grandes, sigue siendo necesario que el sistema de memoria, usando la técnica de la memoria virtual, proporcione a los procesos espacios lógicos más grandes que la memoria realmente disponible. Observe que a la memoria le ocurre lo mismo que a los armarios: cuanto más grandes son, más cosas metemos dentro de ellos. La disponibilidad de memorias mayores permite a los programadores obtener beneficio del avance tecnológico, pudiendo incluir características más avanzadas en sus aplicaciones o incluso tratar problemas que hasta entonces se consideraban inabordables,

4.2. MODELO DE MEMORIA DE UN PROCESO

El sistema operativo gestiona el mapa de memoria de un proceso durante la vida del mismo. Dado que el mapa inicial de un proceso esta muy vinculado con el archivo que contiene el programa ejecutable asociado al mismo, esta sección comenzara estudiando como se genera un archivo ejecutable y cuál es la estructura típica del mismo. A continuación, se analizará como 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. Por último, se expondrán cuáles son las operaciones típicas sobre las regiones de un proceso.

4.2.1. Fases en la generación de un ejecutable

Habitualmente los programadores desarrollan sus aplicaciones utilizando lenguajes de alto nivel. 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 4.8, este procesamiento típicamente consta de dos fases:

- **Compilación**. Se genera el código máquina correspondiente a cada módulo fuente de la aplicación asignando direcciones a los símbolos definidos en el módulo y resolviendo las referencias a los mismos. As, si a una variable se le asigna una determinada posición de memoria, todas las instrucciones que hagan referencia a esa variable deben especificar dicha dirección. Las referencias a símbolos que no están definidos en el módulo quedan pendientes de resolver hasta la fase de montaje. Como resultado de esta fase se genera modulo objeto por cada archivo fuente.

Figura 4.8. *Fases en la generación de un ejecutable.*

- **Montaje o enlace.** Se genera un ejecutable agrupando todos los archivos objeto y resolviendo las referencias entre módulos, o sea, haciendo que las referencias a un determinado símbolo apunten a la dirección asignada al mismo. Además de este tipo de referencias, pueden existir referencias a símbolos definidos en otros archivos objeto

previamente compilados agrupados normalmente en bibliotecas. El montador, por tanto, debe generalmente incluir en el ejecutable otros objetos extraídos de las bibliotecas correspondientes. Así, por ejemplo, si la aplicación usa una función matemática como la que calcula el coseno, el montador deberá extraer de la biblioteca matemática el objeto que contenga la definición de dicha función, incluirlo en el ejecutable y resolver 1a referencia a dicha función desde la aplicación de manera que se corresponda con la dirección de memoria adecuada.

Bibliotecas de objetos

Una biblioteca es una colección de objetos normalmente relacionados entre sí. En el sistema existe un conjunto de bibliotecas predefinidas que proporcionan servicios a las aplicaciones. Estos servicios incluyen tanto los correspondientes a un determinado lenguaje de alto nivel (el API del lenguaje) como los que permiten el acceso a los servicios del sistema operativo (el API del sistema operativo).

Asimismo, cualquier usuario puede crear sus propias bibliotecas para de esta forma poder organizar mejor los módulos de una aplicación y facilitar que las aplicaciones compartan módulos. Así, por ejemplo, un usuario puede crear una biblioteca que maneje números complejos y utilizar esta biblioteca en distintas aplicaciones.

Bibliotecas dinámicas

La manera de generar el ejecutable comentada 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. Este modo de trabajo presenta varias desventajas:

- El archivo ejecutable puede ser bastante grande ya que incluye, además del código propio de la aplicación, todo el código de las funciones «externas» que usa el programa.

- Todo programa en el sistema que use una determinada función de biblioteca tendrá una copia del código de la misma, Como ejemplo, el código de la función printf, utilizada en casi todos los programas escritos en C, estará almacenado en todos los ejecutables que la usan.
- Cuando se estén ejecutando simultáneamente varias aplicaciones que usan una misma función de biblioteca, existirán en memoria múltiples copias del código de dicha función aumentando el gasto de memoria.
- La actualización de una biblioteca implica tener que volver a generar los ejecutables que la incluyen por muy pequeño que sea el cambio que se ha realizado sobre la misma. Supóngase que se ha detectado un error en el código de una biblioteca o que se ha programado una versión más rápida de una función de una biblioteca. Para poder beneficiarse de este cambio, los ejecutables que la usan deben volver a generarse a partir de los objetos correspondientes. Observe que esta operación no siempre puede realizarse ya que dichos objetos pueden no estar disponibles.

Para resolver estas deficiencias se usan las bibliotecas dinámicamente enlazadas *(Dinamic Link Libraries)* o, simplemente, **bibliotecas dinámicas.** Por contraposición, se denominarán **bibliotecas estáticas** a las presentadas hasta el momento.

Con este nuevo mecanismo, el proceso de montaje de una biblioteca de este tipo se difiere y

en vez de realizarlo en la fase de montaje se realiza en tiempo de ejecución del programa. Cuando en la fase de montaje el montador procesa una biblioteca dinámica, no incluye en el ejecutable código extraído de la misma, sino que simplemente anota en el ejecutable el nombre de la biblioteca para que ésta sea cargada y enlazada en tiempo de ejecución.

Como parte del montaje, se incluye en el ejecutable un módulo de montaje dinámico que encargará de realizar en tiempo de ejecución la carga y el montaje de la biblioteca cuando se haga referencia por primera vez a algún símbolo definido en la misma. En el código ejecutable original del programa, las referencias a los símbolo de la biblioteca, que evidentemente todavía pendientes de resolver, se hacen corresponder con símbolos en el modulo de montaje dinámico. De esta forma, la primera referencia a uno de estos símbolos produce la activación del módulo que realizará en ese momento la carga de la biblioteca y el proceso de montaje necesario. Como parte del mismo, se resolverá la referencia a ese símbolo de manera que apunte al objeto real de la biblioteca y que, por tanto, los posteriores accesos al mismo no afecten al módulo de montaje dinámico. Observe que este proceso de resolución de referencias afecta al programa que hace uso de la biblioteca dinámica ya que implica modificar en tiempo de ejecución alguna de sus instrucciones para que apunten a la dirección real del símbolo. Esto puede entrar en conflicto con el típico carácter de no modificable que tiene la región de código.

El uso de bibliotecas dinámicas resuelve las deficiencias identificadas anteriormente:

- El tamaño de los archivos ejecutables disminuye considerablemente ya que en ellos no se almacena el código de las funciones de bibliotecas dinámicas usadas por el programa.
- Las rutinas de una biblioteca dinámica estarán almacenadas únicamente en archivo de la biblioteca en vez de estar duplicadas en los ejecutables que las usan.
- Cuando se estén ejecutando varios proceso que usan una biblioteca dinámica, éstos podrán compartir el código de la misma, por lo que en memoria habrá solo una copia de la misma.
- La actualización de una biblioteca dinámica es inmediatamente visible a los programas que la usan sin necesidad de volver a montarlos.

Con respecto a este ultimo aspecto, es importante resaltar que cuando la actualización implica un cambio en la interfaz de la biblioteca (p. ej.:una de las funciones tiene un nuevo parámetro), el programa no debería usar esta biblioteca sino la versión antigua de la misma. Para resolver este problema de incompatibilidad, en muchos sistemas se mantiene un número de versión asociado a cada biblioteca. Cuando se produce un cambio en la interfaz, se crea una nueva versión con un nuevo número. Cuando en tiempo de ejecución se procede a la carga de una biblioteca, se busca la versión de la biblioteca que coincida con la que requiere el programa, produciéndose un error en caso de no encontrarla. Observe que durante el montaje se ha almacenado en el ejecutable el nombre y la versión de la biblioteca que utiliza el programa.

Por lo que se refiere al compartimiento del código de una biblioteca dinámica entre los procesos que la utilizan en un determinado instante, es un aspecto que hay que analizar más en detalle. Como se planteó en la sección anterior, si se permite que el código de una biblioteca pueda estar asociado a un rango de direcciones diferente en cada proceso, surgen problemas con las referencias que haya en el código de la biblioteca a símbolos definidos en la misma. Ante esta situación se presentan tres alternativas:

- Establecer un rango de direcciones predeterminado y específico para cada biblioteca dinámica de manera que todos los procesos que la usen la incluirán en dicho rango dentro de su mapa de memoria. Esta solución permite compartir el código de la biblioteca, pero es poco flexible ya que limita el numero de bibliotecas que pueden existir en el sistema y puede causar que el mapa de un proceso sea considerablemente grande presentando amplias zonas sin utilizar.
- Reubicar las referencias presentes en el código de la biblioteca durante la carga de la misma de manera que se ajusten a las direcciones que le han correspondido dentro del mapa de memoria del proceso que la usa. Esta opción permite cargar la biblioteca en cualquier zona libre del mapa del proceso. Sin embargo, impide el poder compartir su código (o, al menos, la totalidad de su código) al estar adaptado a la zona de memoria donde le ha tocado vivir.
- Generar el código de la biblioteca de manera que sea independiente de la posición (en ingles se suelen usar las siglas PIC, *Position Independent Code).* Con esta técnica, el compilador genera código que usa direccionamientos relativos a un registro (p. ej.: al contador de programa) de manera que no se ve afectado por la posición de memoria donde ejecuta. Esta alternativa permite que la biblioteca resida en la zona del mapa que se considere conveniente y que, además, su código sea compartido por los procesos que la usan. El único aspecto negativo es que la ejecución de este tipo de código es un poco menos eficiente que el convencional (Gingell, 1987), pero, generalmente, este pequeño inconveniente queda compensado por el resto de las ventajas.

El único aspecto negativo del uso de las bibliotecas dinámicas es que el tiempo de ejecución del programa puede aumentar ligeramente debido a que con este esquema el montaje de la biblioteca se realiza en tiempo de ejecución. Sin embargo, este aumento es perfectamente tolerable en la mayoría de los casos y queda claramente contrarrestado por el resto de los beneficios que ofrece este mecanismo.

Otro aspecto que conviene resaltar es que el mecanismo de bibliotecas dinámicas es transparente al modo de trabajo habitual de un usuario. Dado que este nuevo mecanismo sólo atañe a la fase de montaje, tanto el código fuente de un programa como el código objeto no se en afectados por el mismo. Además, aunque la fase de montaje haya cambiado considerablemente, los mandatos que se usan en esta etapa son típicamente lo mismos. De esta forma, el usuario no detecta ninguna diferencia entre usar bibliotecas estática. y dinámicas excepto, evidentemente, los beneficios antes comentados.

En la mayoría de los sistemas que ofrecen bibliotecas dinámicas, el sistema tiene una

versión estática y otra dinámica de cada una de las bibliotecas predefinidas. Dados los importantes beneficios que presentan las bibliotecas dinámicas, el montador usará por defecto la versión dinámica, Si el usuario, por alguna razón, quiere usar la versión estática, deberá pedirlo explícitamente.

Montaje explícito de bibliotecas dinámicas

La forma de usar las bibliotecas dinámicas que se acaba de exponer es la mas habitual: se especifica en tiempo de montaje que bibliotecas se deben usar y se pospone la carga y el montaje hasta el tiempo de ejecución. Este esquema se suele denominar **enlace dinámico implícito**. Sin embargo no es la única manera de usar las bibliotecas dinámicas.

En algunas aplicaciones no se conoce en tiempo de montaje qué bibliotecas necesitará programa. Supóngase, por ejemplo, un navegador de Internet que maneja hojas que contienen archivos en distintos formatos y que usa las funciones de varias bibliotecas dinámicas para procesar cada uno de los posibles formatos. En principio, cada vez que se quiera dotar al navegador de la capacidad de manejar un nuevo tipo de formato, sería necesario volver a montarlo especificando también la nueva biblioteca que incluye las funciones para procesarlo. Observe que esto sucede aunque se usen bibliotecas dinámicas.

Lo que se requiere en esta situación es poder decidir en tiempo de ejecución que biblioteca dinámica se necesita y solicitar explícitamente su montaje y carga. El mecanismo de bibliotecas dinámicas presente en Windows NT y en la mayoría de las versiones de UNIX ofrece esta funcionalidad, denominada típicamente **enlace dinámico explícito**. Un programa que pretenda usar funcionalidad deberá hacer uso de los servicios que ofrece el sistema para realizar esta solicitud explícita. Observe que, en este caso, no habría que especificar en tiempo de montaje el nombre de la biblioteca dinámica. Pero, como contrapartida, el mecanismo de carga y montaje de la biblioteca dinámica deja de ser transparente a la aplicación.

Formato del ejecutable

Como parte final del proceso de compilación y montaje, se genera un archivo ejecutable que contiene el código máquina del programa. Distintos fabricantes han usado diferentes formatos para este tipo de archivos. En el mundo UNIX, por ejemplo, uno de los formatos más utilizados actualmente denominado *Executable and Linkable Format* (ELF). A continuación, se presentará de simplificada cómo es el formato típico de un ejecutable.

Como se puede observar en la Figura 4.9, 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 la cabecera típicamente se incluye, entre otras cosas, la siguiente información:

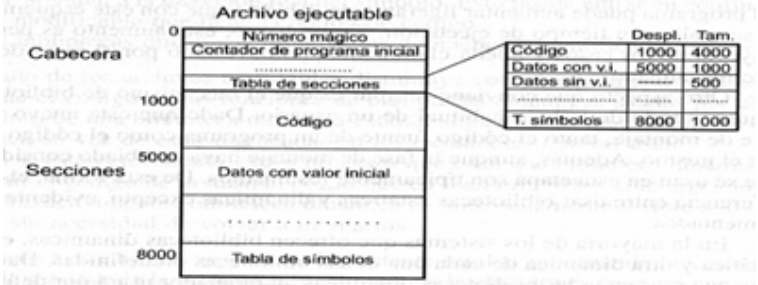


Figura 4.9. Formato simplificado de un ejecutable.

- Un «número mágico» que identifica al ejecutable. Así, por ejemplo, en el formato ELF, el primer byte del archivo ejecutable debe contener el valor hexadecimal *7* f y los tres siguientes los caracteres 'E', 'L' y 'F'.

- La dirección del punto de entrada del programa, esto es, la dirección que se almacenará inicialmente en el contador de programa del proceso.
- Una tabla que describe las secciones que aparecen en el ejecutable. Por cada una de ellas, se especifica, entre otras cosas, su tipo, la dirección del archivo donde comienza y su tamaño.

En cuanto a las secciones, Cada ejecutable tiene un conjunto de secciones. El contenido de estas secciones es muy variado. Una sección puede contener información para facilitar la depuración (p. ej.: una tabla de símbolos). Otra sección puede contener la lista de bibliotecas dinámicas que requiere el programa durante su ejecución. Sin embargo, esta exposición se centrará en las tres que más influyen en el mapa de memoria de un proceso:

- Código (texto). Contiene el código del programa.
- Datos con valor inicial. Almacena el valor inicial de todas las variables globales a las que se les ha asignado un valor inicial en el programa.
- Datos sin valor inicial. Se corresponde con todas las variables globales a las que no se les ha dado un valor inicial. Como muestra la figura, esta sección aparece descrita 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. En la tabla únicamente se especifica su tamaño.

Observe que en el ejecutable no aparece ninguna sección que corresponda con las variables locales y parámetros de funciones. Esto se debe a que este tipo de variables presentan unas características muy diferentes a las de Las variables globales, a saber:

- Las variables globales tienen un carácter estático. Existen durante toda la vida del programa. Tienen asociada una dirección fija en el mapa de memoria del proceso y, por tanto, también en el archivo ejecutable puesto que éste se corresponde con la imagen inicial del proceso. En el caso de que la variable no tenga un valor inicial asignado en el programa, no es necesario almacenarla explícitamente en el ejecutable, sino que basta con conocer el tamaño de la sección que contiene este tipo de variables.
- Las variables locales y parámetros tienen carácter dinámico. Se crean cuando se invoca la función correspondiente y se destruyen cuando termina la llamada. Por tanto, estas variables no tienen asignado espacio en el mapa inicial del proceso ni en el ejecutable. Se crean dinámicamente usando para ello la pila del proceso. La dirección que corresponde a una variable de este tipo se determina en tiempo de ejecución ya que depende de la secuencia de llamadas que genere el programa Observe que, dada la posibilidad de realizar llamadas recursivas directas o indirectas que proporciona la mayoría de los lenguajes, pueden existir en tiempo de ejecución varias instancias de una variable de este tipo.

En el Programa 4.1 se muestra el esqueleto de un programa en C que contiene diversos tipos de variables. En él se puede observar que las variables **x** e ***y*** son globales, mientras que **z** es local y es **t** un parámetro de una función. La primera de ellas tiene asignado un valor inicial, por lo que dicho valor estará almacenado en la sección del ejecutable que corresponde a este tipo de variables. Con respecto a la variable **y,** no es necesario almacenar ningún valor en el ejecutable

asociado a la misma, pero su existencia quedará reflejada en el tamaño total de la sección de variables globales sin valor inicial. En cuanto a la variable local **z** y al parámetro **t**, no están asociados a ninguna sección del ejecutable y se les asignará espacio en tiempo de ejecución cuando se produzcan llamadas a la función donde están definidos.

Programa 4.1. Esqueleto de un programa.

```
int x=8;
int y;
f(int t){
    int Z;
    ..............
}
main( ) {
  ..............
}
```

4.2.2. Mapa de memoria de un proceso

Como se comentó previamente, el mapa de memoria de un proceso no es algo homogéneo sino que está formado por distintas regiones o segmentos. Una región tiene asociada una determinada información. En este capítulo se denominará **objeto de memoria** a este conjunto de información relacionada. La asociación de una región de un proceso con un objeto de memoria permite al proceso tener acceso a la información contenida en el objeto.

Cuando se activa la ejecución de un programa (servicio exec en POSIX y CreateProcess en Win32), se crean varias regiones dentro del mapa a partir de la información del ejecutable. Cada sección del ejecutable constituye un objeto de memoria. 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 de proceso donde comienza y por su tamaño. Además, tendrá asociadas una serie de propiedades características especificas como las siguientes:

- Soporte de la región. El objeto de memoria asociado a la región. En él está almacenado el contenido inicial de la región. Se presentan normalmente dos posibilidades:

— *Soporte en archivo.* El objeto está almacenado en un archivo o en parte del mismo.
— *Sin soporte.* El objeto no tiene un contenido inicial. En algunos entornos se les denomina objetos anónimos.

- 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 en el objeto de memoria.
— *Compartida.* El contenido de la región puede ser compartido por varios procesos. Las modificaciones en el contenido de la región se reflejan en el objeto de memoria.

- Protección. Tipo de acceso permitido. Típicamente se distinguen tres tipos:
  — *Lectura.* Se permiten accesos de lectura de operandos de instrucciones.
  — *Ejecución.* Se permiten accesos de lectura de instrucciones *(fetch).*
  — *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.

Como se puede observar en la Figura 4.10, 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 (o texto). Se trata de una región compartida de lectura/ejecución. Es de tamaño fijo (el indicado en la cabecera del ejecutable). El soporte de esta región está en la sección correspondiente del ejecutable.
- Datos con valor inicial. Se trata de una región privada ya que cada proceso que ejecuta un determinado programa necesita una copia propia de las variables del mismo. Es de lectura/escritura y de tamaño fijo (el indicado en la cabecera del ejecutable). El soporte de esta región está en la sección correspondiente del ejecutable.
- Datos sin valor inicial. Se trata de una región privada de lectura/escritura y de tamaño fijo (el indicado en la cabecera del ejecutable). Como se comento previamente, esta región no tiene soporte en el ejecutable ya que su contenido inicial es irrelevante. En muchos sistemas se le da un valor inicial de cero a toda la región por motivos de confidencialidad.
- 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, dirección 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. De manera similar a la región de datos sin valor inicial , por razones de confidencialidad de la información, cuando crece la pila se rellena a cero la zona expandida. En el mapa inicial existe ya esta región que contiene típicamente los argumentos especificados en la invocación del programa (en el caso de POSIX, estos argumentos son los especificados en el segundo parámetro de la función execvp).

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:

.

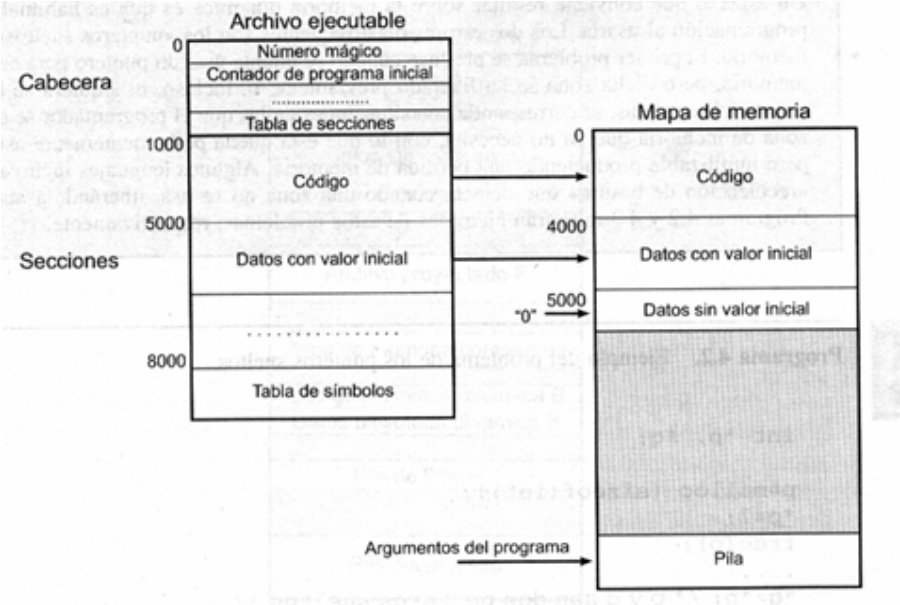


Figura 4.10. *Mapa de memoria inicial a partir del ejecutable*

• *Heap.* La mayoría de los lenguajes de alto nivel ofrecen la posibilidad de reservar espacio en tiempo de ejecución. En el caso del lenguaje C se usa la función malloc para ello. Esta región sirve de soporte para la memoria dinámica que reserva un programa en tiempo de ejecución. Comienza, típicamente, justo después de la región de datos sin valor inicial (de hecho, en algunos sistemas se considera parte de la misma) y crece en sentido contrario a la pila (hacia direcciones crecientes), Se trata de una región privada de lectura/escritura, sin soporte (se rellena inicialmente a cero), que crece según el programa vaya reservando memoria dinámica y decrece según la vaya liberando. Normalmente, cada programa tendrá un único *heap.* Sin embargo, algunos sistemas, como Win32, permiten crear múltiples *heaps.*

En la Aclaración 4.2 se intenta clarificar algunos aspectos de la memoria dinámica.

• Archivos proyectados. Cuando se proyecta un archivo, se crea una región asociada al mismo. Este mecanismo será realizado con más profundidad al final del capítulo, pero, a *priori,* se puede resaltar que se trata de una región compartida cuyo soporte es el archivo que se proyecta.

• Memoria compartida. Cuando se crea una zona de memoria compartida y se proyecta, como se analizará detalladamente en el Capitulo 4, se origina 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.* Como se estudió en el Capítulo 3, cada *thread* necesita una pila propia que normalmente 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.

ACLARACIÓN 4.2

El sistema operativo sólo realiza una gestión básica de la memoria dinámica. Generalmente, las aplicaciones no usan directamente estos servicios. La biblioteca de cada lenguaje de programación utiliza estos servicios básicos para construir a partir de ellos unos más avanzados orientados a las aplicaciones. Un aspecto que conviene resaltar sobre la memoria dinámica es que es habitual cometer errores de programación al usarla. Los dos errores más frecuentes son los «punteros sueltos» y las «goteras» en memoria. El primer problema se produce cuando se intenta usar un puntero para acceder a una zona de memoria, pero dicha zona se ha liberado previamente o, incluso, ni siquiera se había reservado. En cuanto a las goteras, se corresponde con situaciones en las que el programador se olvida de liberar una zona de memoria que ya no necesita, con lo que ésta queda permanentemente asignada al programa, pero inutilizable produciendo una pérdida de memoria. Algunos lenguajes incluyen un mecanismo de «recolección de basura» que detecta cuando una zona no se usa, liberándola automáticamente. Los Programas 4.2 y 4.3 muestran ejemplos de estos problemas, respectivamente.

Programa 4.2. Ejemplo del problema de los punteros suelto..

```
int *p, *q ;

p=malloc (sizeof (int)) ;
*p=7;
free(p);
.....
*q=*p;  /* p y q son dos punteros sueltos */
```

Programa 4.3. Ejemplo del problema de las goteras.

```
int *p, *q;

p=malloc (sizeof(int) );
q=malloc (sizeof(int) );
p=q;
/* la primera zona reservada queda inutilizable */
```

En la Figura 4.11 se muestra un hipotético mapa de memoria que contiene algunos de los tipos de regiones comentadas en esta sección.

Como puede apreciarse en la figura, 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).

Hay que resaltar que, dado el carácter dinámico del mapa de memoria de un proceso (se crean y destruyen regiones, algunas regiones cambian de tamaño, etc.), existirán, en un determinado instante, zonas sin asignar (**huecos**) dentro del mapa de memoria del proceso. Cualquier acceso a estos huecos representa un error y debería ser detectado y tratado por el sistema operativo.

Por último, hay que recalcar que, dado que el sistema operativo es un programa, su mapa de memoria contendrá también regiones de código, datos y *heap* (el sistema operativo también usa memoria dinámica).

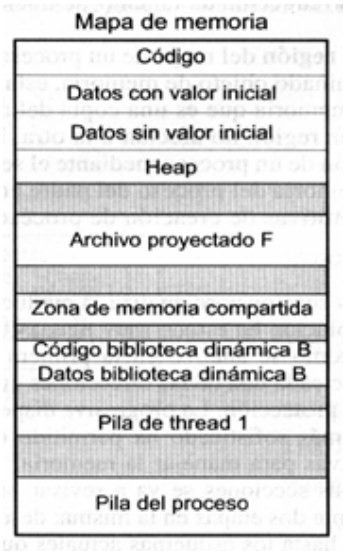


Figura 4.11. *Mapa de memoria de un proceso hipotético.*

4.2.3. Operaciones sobre regiones

Durante la vida de un proceso, su mapa de memoria va evolucionando y con el las regiones incluidas en el mismo. En aras de facilitar el estudio que se realizará en el resto de este capítulo, sección se identifican que operaciones se pueden realizar sobre una región dentro del mapa proceso. En esta exposición se plantean las siguientes operaciones genéricas:

- **Crear una región** dentro del mapa de un proceso asociándola un objeto de sistema operativo crea una nueva región vinculada al objeto en el lugar correspondiente del mapa asignándola los recursos necesarios y estableciendo las características y propiedades de la misma (tipo de soporte, carácter privado o compartido, tipo de protección y tamaño fijo o variable). En la creación del mapa de un proceso asociado a un determinado programa se realiza esta operación por cada una de las regiones iniciales (código, datos con valor, datos sin valor inicial y pila). Durante la ejecución del programa, también se lleva a diferentes situaciones como, por ejemplo, cuando se carga una biblioteca dinámica o se proyecta un archivo.
- **Eliminar una región del mapa de un proceso.** Esta operación libera todos los recursos vinculados a la región que se elimina. Cuando un proceso termina, voluntaria o involuntariamente, se liberan implícitamente todas sus regiones. En el caso de una región que corresponda con una zona de memoria compartida o un archivo proyectado el proceso puede solicitar explícitamente su eliminación del mapa.
- **Cambiar el tamaño de una región.** El tamaño de la región puede cambiar ya sea por una petición explícita del programa, como ocurre con la región del *heap*, o de forma implícita, como sucede cuando se produce una expansión de la pila. En el caso de un aumento de tamaño, el sistema asignará los recursos necesarios a la región comprobando previamente que la expansión no provoca un solapamiento, en cuyo caso no se llevaría a cabo. Cuando se trata de una reducción de tamaño, se liberan los recursos vinculados al fragmento liberado.
- **Duplicar una región** del mapa de un proceso en el mapa de otro. Dada una región

asociada a un determinado objeto de memoria, esta operación crea una nueva región asociada a un objeto de memoria que es una copia del anterior. Por tanto, las

modificaciones que se realizan en una región no afectan a la otra. Esta operación serviría de base para lleva a cabo la creación de un proceso mediante el servicio fork de POSIX, que requiere duplicar el mapa de memoria del proceso del padre en el proceso hijo. En sistemas operativos que no tengan primitivas de creación de procesos de este estilo no se requeriría esta operación.

En las siguientes secciones se analiza la evolución de la gestión de memoria en los sistemas operativos. Dicha evolución ha estado muy ligada con la del hardware de gestión de memoria del procesador ya que, como se analizo en la primera sección del capítulo, se necesita que sea el procesador el que trate cada una de las direcciones que genera un programa para cumplir los requisitos de reubicación y protección. La progresiva disponibilidad de procesadores con un hardware de gestión de memoria más sofisticado ha permitido que el sistema operativo pueda implementar estrategias más efectivas para manejar la memoria.

En estas próximas secciones se va a revisar brevemente esta evolución, distinguiendo por simplicidad, únicamente dos etapas en la misma: desde los esquemas que realizaban una asignación contigua de memoria hasta los esquemas actuales que están basados en la memoria virtual. Como etapa intermedia entre estos dos esquemas, se presentará la técnica del intercambio.

4.3. ESQUEMAS DE MEMORIA BASADOS EN ASIGNACIÓN CONTIGUA

Un esquema simple de gestión de memoria consiste en asignar a cada proceso una zona contigua de memoria para que en ella resida su mapa de memoria. Dentro de esta estrategia general hay diversas posibilidades. Sin embargo, dado el alcance y objetivo de este tema, la exposición se centrará sólo en uno de estos posibles esquemas: la gestión contigua basada en particiones dinámicas. Se trata de un esquema que se usó en el sistema operativo OS/MVT de IBM en la década de los sesenta.

Con esta estrategia, cada vez que se crea un proceso, el sistema operativo busca un hueco en memoria de tamaño suficiente para alojar el mapa de memoria del mismo. El sistema operativo reservara la parte del hueco necesaria, creara en ella el mapa inicial del proceso y establecerá una función de traducción tal que las direcciones que genera el programa se correspondan con la zona asignada.

En primer lugar, se presentará qué hardware de gestión de memoria se requiere para realizar este esquema para, a continuación, describir cuál es la labor del sistema operativo.

Hardware

Como ya se estudió en el Capítulo 1, este esquema requiere un hardware de memoria relativamente simple. Típicamente el procesador tendrá dos registros *valla,* únicamente accesibles en modo privilegiado, que utilizará para tratar cada una de las direcciones que genera un programa. Como se puede observar en la Figura 4.12, la función de estos registros será la siguiente:

- **Registro límite**. El procesador comprueba que cada dirección que genera el proceso no es mayor que el valor almacenado en este registro. En caso de que lo sea, se generará una excepción.

- **Registro base.** Una vez comprobado que la dirección no rebasa el limite permitido, el procesador le sumará el valor de este registro, obteniéndose con ello la dirección de memoria física resultante.