Capítulo 9

Gestión de la memoria en tiempo de ejecución

9.1 Organización de la memoria durante la ejecución

Como ya se comentó en los primeros epígrafes, para que un programa se ejecute sobre un sistema operativo, es necesaria la existencia de un cargador que suministra al programa un bloque contiguo de memoria sobre el cual ha de ejecutarse. Por tanto, el código del programa resultante de la compilación debe organizarse de forma que haga uso de este bloque, por lo que el compilador debe incorporar al programa objeto todo el código necesario para ello.

Las técnicas de gestión de la memoria durante la ejecución del programa difieren de unos lenguajes a otros, e incluso de unos compiladores a otros. En este capítulo se estudia la gestión de la memoria que se utiliza en lenguajes imperativos como Fortran, Pascal, C, Modula-2, etc. La gestión de la memoria en otro tipo de lenguajes (funcionales, lógicos, etc.) es, en general, diferente de la organización que aquí se plantea.

Para lenguajes imperativos, los compiladores generan programas que tendrán en tiempo de ejecución una organización de la memoria similar (a grandes rasgos) a la que aparece en la figura 9.1.

En este esquema se distinguen claramente las secciones de:

- el Código
- la Zona de Datos de Tamaño Fijo
- la Pila o *stack*
- el Montón o *heap*

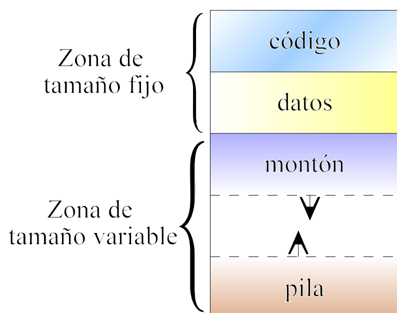

Figure 1 Estructura de la memoria durante la ejecución de un programa compilado

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

9.2 Zona de código

Es la zona donde se almacenan las instrucciones del programa ejecutable en código máquina, y también el código correspondiente a los procedimientos y funciones que utiliza. Su tamaño y su contenido se establecen en tiempo de compilación y por ello se dice que su tamaño es fijo en tiempo de ejecución.

De esta forma, el compilador, a medida que va generando código, lo va situando secuencialmente en esta zona, delimitando convenientemente el inicio de cada función, procedimiento y programa principal (si lo hubiera, como sucede en lenguajes como Pascal o Modula-2).

El programa ejecutable final estará constituido por esta zona junto con información relativa a las necesidades de memoria para datos en tiempo de ejecución, tanto del bloque de tamaño fijo como una estimación del tamaño del bloque dinámico de datos. Esta información será utilizada por el cargador (*linker*) en el momento de hacer la carga del ejecutable en memoria principal para proceder a su ejecución.

9.2.1 *Overlays*

Algunos compiladores fragmentan el código del programa objeto usando *overlays* o "solapas", cuando la memoria principal disponible es inferior al tamaño del programa completo. Estos *overlays* son secciones de código objeto que se almacenan en ficheros independientes y que se cargan en la memoria central dinámicamente, es decir, durante la ejecución del programa. Se les llama "solapas" porque dos *overlays* pueden ocupar el mismo trozo de memoria en momentos de tiempo diferentes, solapándose. Para hacer más eficiente el uso de la memoria, los *overlays* de un programa se agrupan en zonas y módulos, cada uno de los cuales contiene un conjunto de funciones o procedimientos completos. La figura 9.2 muestra cómo se ubican varios

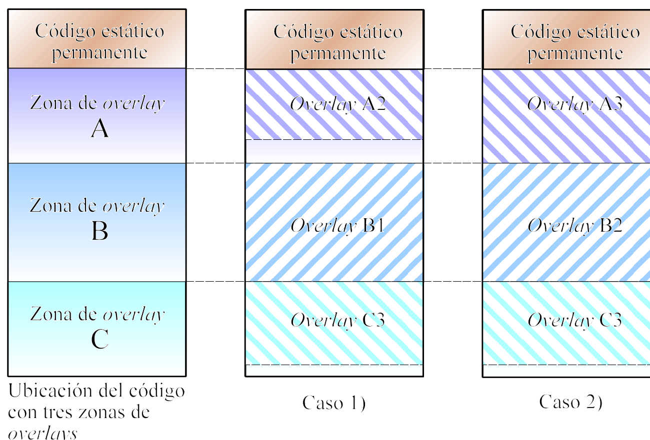

Figure 2 Ejemplos de configuraciones de la zona de código utilizando *overlays*

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

overlays en distintas zonas de memoria estática.

Durante el tiempo de ejecución sólo uno de los *overlays* de cada una de las zonas de *overlay* puede estar almacenado en memoria principal. El compilador reserva en la sección de código una zona contigua de memoria para cada conjunto de *overlays*. El tamaño de esta zona debe ser igual al del mayor módulo que se cargue sobre ella. Es función del programador determinar cuantas zonas de *overlay* se definen, qué funciones y procedimientos se encapsulan en cada módulo de *overlay*, y cómo se organizan estos módulos para ocupar cada una de las zonas de *overlay*. Una restricción a tener en cuenta es que las funciones de un módulo no deben hacer referencia a funciones de otro módulo del mismo *overlay*, ya que nunca estarán simultáneamente en memoria.

Evidentemente, el tiempo de ejecución de un programa estructurado con *overlays* es mayor que si no tuviese *overlays* y todo el código estuviese residente en memoria, puesto que durante la ejecución del programa es necesario cargar cada módulo cuando se realiza una llamada a alguna de las funciones que incluye. También es tarea del programador diseñar la estructura de *overlays* de manera que se minimice el número de estas operaciones. La técnica de *overlays* no sólo se utiliza cuando el programa a compilar es muy grande en relación con la disponibilidad de memoria del sistema, sino también cuando se desea obtener programas de menor tamaño que deben coexistir con otros del sistema operativo cuando la memoria es escasa.

9.3 Zona de datos

Los datos que maneja un programa se dividen actualmente en tres grandes bloques:

- Uno dedicado a almacenar las variables globales accesibles por cualquier línea de código del programa. Este bloque es la Zona de Datos de Tamaño Fijo.
- Otro dedicado a almacenar las variables locales a cada función y procedimiento. Éstas no pueden almacenarse en el bloque anterior por dos motivos principales: a) no es necesario almacenar las variables de una función hasta el momento en que ésta es invocada y, una vez que finaliza su ejecución, tampoco; y b) durante la ejecución de una función recursiva deben almacenarse varias instancias de sus variables locales, concretamente tantas como invocaciones a esa misma función se hayan producido. Este bloque es la Pila.
- Un último bloque dedicado a almacenar los bloques de memoria gestionados directamente por el usuario mediante sentencias **malloc/free**, **new/dispose**, etc. La cantidad de memoria requerida para estos menesteres varía de ejecución en ejecución del programa, y los datos ubicados no son locales a ninguna función, sino que perduran hasta que son liberados. Este bloque es el Montón.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
BY NC SA

9.3.1 Zona de Datos de Tamaño Fijo

La forma más fácil de almacenar el contenido de una variable en memoria en tiempo de ejecución es en memoria estática o permanente a lo largo de toda la ejecución del programa. No todos los objetos (variables) pueden ser almacenados estáticamente. Para que un objeto pueda ser almacenado en memoria estática, su tamaño (número de *bytes* necesarios para su almacenamiento) ha de ser conocido en tiempo de compilación. Como consecuencia de esta condición no podrán almacenarse en memoria estática:

- Las variables locales correspondientes a procedimientos o funciones recursivas, ya que en tiempo de compilación no se sabe el número de veces que estas variables serán necesarias.
- Las estructuras dinámicas de datos tales como listas, árboles, etc. ya que el número de elementos que la forman no es conocido hasta que el programa se ejecuta.

Por tanto, las variables locales a una función son datos de tamaño definido, pero para los que no se sabe el número de ocurrencias del ámbito del que dependen, por lo que deben almacenarse siguiendo algún criterio dinámico. Tampoco es posible almacenar en esta zona a las estructuras eminentemente dinámicas, esto es, las que están gestionadas a través de punteros mediante creación dinámica de espacio libre: listas dinámicas, grafos dinámicos, etc. Estos datos son de tamaño indefinido por lo que no es posible guardarlos en la zona de datos de tamaño fijo. Aunque pueda pensarse que el tamaño del código de una función recursiva también depende del número de veces que la función se llama a sí misma, esto no es cierto: las instrucciones a ejecutar son siempre las mismas, y lo que cambia son los datos con los que se trabaja, o sea, las variables locales.

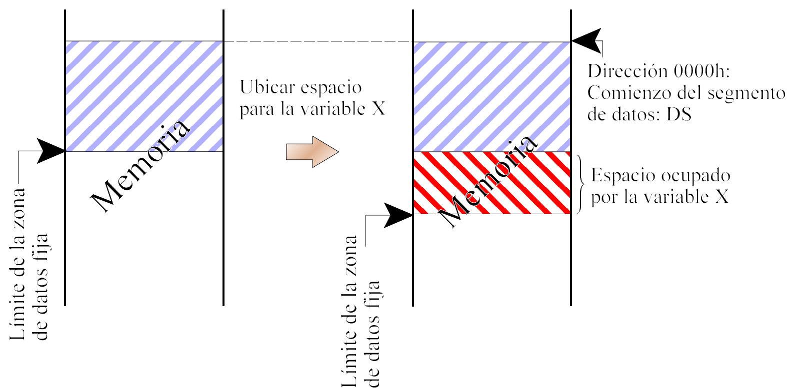

Figure 3 Asignación de un bloque de memoria para una variable global **X**

Las técnicas de asignación de memoria estática son sencillas, tal y como

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

ilustra la figura 9.3. A partir de una posición señalada por un puntero de referencia (límite de la zona de datos fija) se aloja la variable global **X**, y se avanza el puntero tantos *bytes* como sean necesarios para almacenar dicha variable. En otras palabras, a cada variable se le asigna una dirección y un trozo de memoria a partir de ella acorde con el tipo de la variable; estos trozos se van alojando secuencialmente a partir del comienzo del área de datos de tamaño fijo. Por ejemplo, si encontramos la declaración de Modula-2:

```
a, b : INTEGER;
c : ARRAY [1..20] OF REAL;
c : CHAR;
d : REAL;
```

y suponiendo que un INTEGER ocupa 2 bytes, un REAL ocupa 5 y un CHAR ocupa 1, y suponiendo que el comienzo del área de datos se encuentra en la posición 0000h (esta dirección es relativa al segmento de datos DS que, realmente, puede apuntar a cualquier dirección de memoria y que será inicializado convenientemente por el cargador), entonces se tiene la siguiente asignación de direcciones y tamaños:

Variable	Tamaño	Dir. comienzo
a	2 bytes	0000h
b	2 bytes	0002h
c	100 bytes	0004h
d	1 byte	0068h
e	5 byte	0069h

Esta asignación de memoria se hace en tiempo de compilación y los objetos están vigentes y son accesibles desde que comienza la ejecución del programa hasta que termina. La dirección asociada a cada variable global es constante y relativa al segmento de datos, o sea, siempre la misma a partir del punto de inicio de memoria en que se ha cargado el programa (punto que puede variar de una ejecución a otra).

En los lenguajes que permiten la existencia de subprogramas, y siempre que todos los objetos de estos subprogramas puedan almacenarse estáticamente (por ejemplo en Fortran-IV que no permite la recursión) se aloja en la memoria estática un registro de activación correspondiente a cada uno de los subprogramas, con una estructura como la de la figura 9.4.a). Estos registros de activación contendrán las variables locales, parámetros formales y valor devuelto por la función, y se distribuyen linealmente en la zona de datos de tamaño fijo tal como indica la figura 9.4.b).

Dentro de cada registro de activación las variables locales se organizan en secuencia. Existe un solo registro de activación para cada procedimiento y por tanto no están permitidas las llamadas recursivas. El proceso que se sigue cuando un procedimiento **P** llama a otro **Q** es el siguiente:

- **P** (el llamador) evalúa los parámetros reales de invocación, en caso de que se trate de expresiones complejas, usando para ello una zona de memoria temporal para el almacenamiento intermedio. Por ejemplo, si la llamada a **Q** es “Q((3*5)+(2*2),7)” las operaciones previas a la llamada propiamente dicha

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

en código máquina han de realizarse sobre alguna zona de memoria temporal. (En algún momento debe haber una zona de memoria que contenga el valor intermedio 15, y el valor intermedio 4 para sumarlos a continuación). En caso de utilización de memoria estática, esta zona de temporales puede ser común a todo el programa, ya que su tamaño puede deducirse en tiempo de compilación.

- **Q** inicializa sus variables y comienza su ejecución.

Dado que las variables están permanentemente en memoria es fácil implementar la propiedad de que conserven o no su contenido para cada nueva llamada (comportamiento parecido al del modificador **static** del lenguaje C).

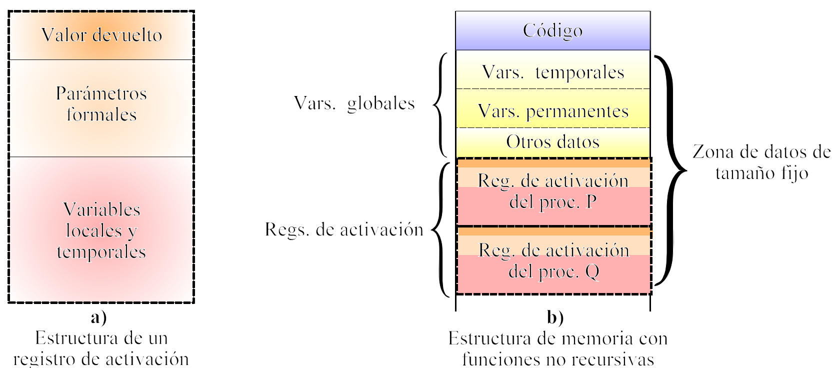

Figure 4 Estructura de un registro de activación asociado a una función no recursiva. Cada función o procedimiento tiene asociado un registro de activación de tamaño diferente, en función de cuantas variables locales posea, del tipo del valor devuelto, etc.

9.3.2 Pila (*Stack*)

Un lenguaje con estructura de bloques es aquél que está compuesto por módulos o trozos de código cuya ejecución es secuencial; estos módulos a su vez pueden contener en su secuencia de instrucciónes llamadas a otros módulos, que a su vez pueden llamar a otros submódulos y así sucesivamente.

La aparición de este tipo de lenguajes trajo consigo la necesidad de técnicas de alojamiento en memoria más flexibles, que pudieran adaptarse a las demandas de memoria durante la ejecución del programa debido a las invocaciones recursivas. En estos lenguajes, cada vez que comienza la ejecución de un procedimiento se crea un registro de activación para contener los objetos necesarios para su ejecución, eliminandolo una vez terminada ésta.

Durante una invocación recursiva sólo se encuentra activo (en ejecución) el

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
BY NC SA

último procedimiento invocado, mientras que la secuencia de invocadores se encuentra "dormida". Esto hace que, a medida que las funciones invocadas van finalizando, deban reactivarse los procedimientos llamadores en orden inverso a la secuencia de llamadas. Por esto, los distintos registros de activación asociados a cada bloque deberán colocarse en una pila en la que entrarán cuando comience la ejecución del bloque y saldrán al terminar el mismo. La estructura de los registros de activación varía de unos lenguajes a otros, e incluso de unos compiladores a otros, siendo éste uno de los problemas por

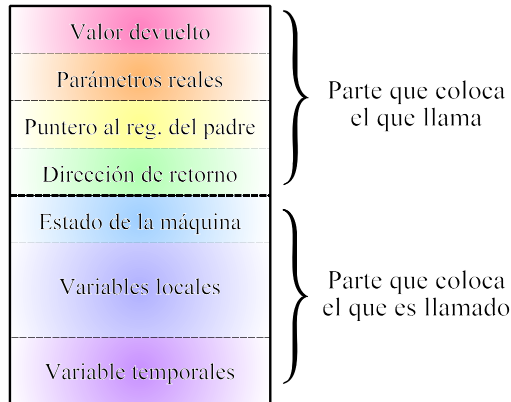

Figure 5 Estructura completa de un registro de activación en lenguajes que admiten recursión

los que a veces resulta difícil enlazar los códigos generados por dos compiladores diferentes. En general, los registros de activación de los procedimientos suelen tener algunos de los campos que pueden verse en la figura 9.5.

En la zona correspondiente al **estado de la máquina** se almacena el contenido que hubiera en los registros del microprocesador antes de comenzar a ejecutarse el procedimiento. Estos valores deberán ser repuestos al finalizar su ejecución. El código encargado de realizar la copia del estado de la máquina es común para todos los procedimientos.

El puntero al registro de activación del padre permite el acceso a las variables declaradas en otros procedimientos dentro de los cuales se encuentra inmerso aquél al que pertenece este registro de activación. Por ejemplo, supongamos un bloque de programa como el siguiente:

```
PROCEDURE A1
VAR
    A1_a, A1_b : INTEGER;
    PROCEDURE B1
    VAR
        B1_c, B1_d : INTEGER;
    BEGIN
        ...
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
        END B1;
        PROCEDURE B2
        VAR
            B2_e, B2_f : INTEGER;
            PROCEDURE C2
            VAR
                C2_g, C2_h : INTEGER;
            BEGIN
                ...
                CALL B1;
                ...
            END C2;
        BEGIN
            ...
            CALL C2;
            ...
        END B2;
    BEGIN (* Programa principal *)
        ...
        CALL B2;
        ...
    END;
```

El puntero al registro de activación del padre se utiliza porque un procedimiento puede hacer uso de las variables locales de los procedimientos en los que se halla declarado. En este ejemplo, C2 puede hacer uso de las variables:

- C2_g y C2_h (que son suyas).
- B2_e y B2_f (que son de su padre, aquél en el que C2 está declarado).
- A1_a y A1_b (que son del padre de su padre).

pero no puede hacer uso ni de B1_c ni de B1_d, ya que C2 no está inmerso en B1.

Al igual que en la zona de datos de tamaño fijo, los registros de activación contienen espacio para almacenar los parámetros reales (valores asociados a las variables que aparecen en la cabecera) y las variables locales, (las que se definen dentro del bloque o procedimiento) así como una zona para almacenar el valor devuelto por la función y una zona de valores temporales para el cálculo intermedio de expresiones.

Cualesquiera dos bloques o procedimientos diferentes, suelen tener registros de activación de tamaños diferentes. Este tamaño, por lo general, es conocido en tiempo de compilación ya que se dispone de información suficiente sobre el espacio ocupado por los objetos que lo componen. En ciertos casos esto no es así como por ejemplo ocurre en C cuando se utilizan *arrays* de longitud indeterminada. En estos casos el registro de activación debe incluir una zona de desbordamiento al final cuyo tamaño no se fija en tiempo de compilación sino sólo cuando realmente llega a ejecutarse el procedimiento. Esto complica un poco la gestión de la memoria, por lo que algunos compiladores de bajo coste suprimen esta facilidad.

A pesar de que es imprescindible guardar sucesivas versiones de los datos

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

según se van realizando llamadas recursivas, las instrucciones que se aplican sobre ellos son siempre las mismas, por lo que estas instrucciones sólo es necesario guardarlas una vez. No obstante, es evidente que el mismo código trabajará con diferentes datos en cada invocación recursiva, por lo que el compilador debe generar código preparado para tal eventualidad, no haciendo referencia absoluta a las variables (excepto a las definidas globalmente cuya ubicación en memoria no cambia en el transcurso de la ejecución del programa), sino referencia relativa en función del comienzo de su registro de activación.

El procedimiento de gestión de la pila cuando un procedimiento **P** llama a otro procedimiento **Q**, se desarrolla en dos fases; la primera de ellas corresponde al código que se incluye en el procedimiento **P** antes de transferir el control a **Q**, y la segunda, al código que debe incluirse al principio de **Q** para que se ejecute cuando reciba el control. Un ejemplo de esta invocación puede venir dado por un código como:

```
PROCEDURE P( /* param. formales de P */) {
      VAR
            // var. locales de P
            PROCEDURE Q( /* param. formales de Q */) {
                  VAR
                        // var. locales de Q
                  BEGIN
                        ...
                        Q( ... ); // Invocación recursiva de Q a Q
                        ...
            END Q;
      BEGIN
            ...
            x = Q( /* param. reales a Q */); // Invocación de P a Q
END P;
```

La primera de estas fases sigue los siguientes pasos:

1.1 El procedimiento que realiza la llamada (**P**) evalúa las expresiones de la invocación, utilizando para ello su zona de variables temporales, y copia el resultado en la zona correspondiente a los parámetros reales del procedimiento que recibe la llamada. Previo a ello deja espacio para que **Q** deposite el valor devuelto.

1.2 El llamador **P** coloca en la pila un puntero al registro de activación del procedimiento en el cual se halla declarado **Q**, con objeto de que éste pueda acceder a las variables no locales declaradas en su padre. Por último, **P** transfiere el control a **Q**, lo que hace colocar la dirección de retorno encima de la pila.

1.3 El receptor de la llamada (**Q**) salva el estado de la máquina antes de comenzar su ejecución usando para ello la zona correspondiente de su registro de activación.

1.4 **Q** inicializa sus variables y comienza su ejecución.

Al terminar **Q** su ejecución se desaloja su registro de activación procediendo

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

también en dos fases. La primera se implementa mediante instrucciones al final del procedimiento que acaba de terminar su ejecución (**Q**), y la segunda en el procedimiento que hizo la llamada (**P**), tras recobrar el control:

2.1 El procedimiento saliente (**Q**) antes de finalizar su ejecución coloca el valor de retorno al principio de su registro de activación.

2.2 Usando la información contenida en su registro, **Q** restaura el estado de la máquina y coloca el puntero de final de pila en la posición en la que estaba originalmente.

2.3 El procedimiento llamador **P** copia el valor devuelto por el procedimiento invocado **Q** dentro de su propio registro de activación (el de **Q**).

La figura 9.6 ilustra el estado de la pila como consecuencia de realizar la invocación de **P** a **Q**, y luego una sola invocación recursiva de **Q** a **Q**. Nótese en esta figura que la pila crece hacia abajo, como suele dibujarse siempre, ya que crece de las direcciones de memoria superiores a las inferiores.

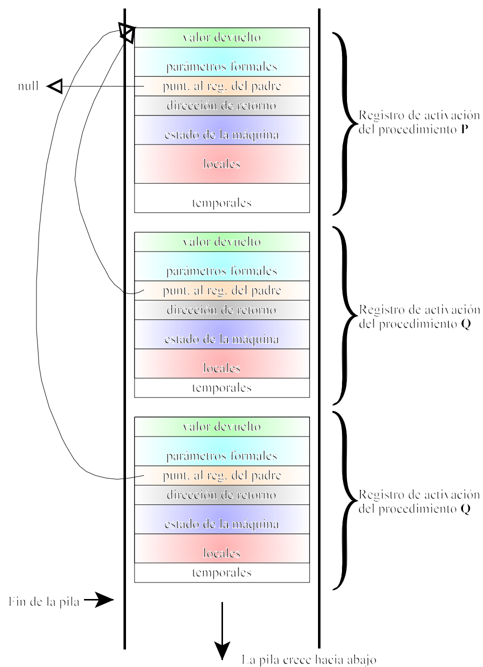

Figure 6 Pila de registros de activación cuando una función **P** invoca a otra **Q** y ésta sellama a sí misma una sola vez. Nótese que el padre de **Q** es **P**, ya que **Q** se halla declarada localmente a **P**. **P** no tiene padre puesto que se supone declarada globalmente

Dentro de un procedimiento o función, las variables locales se referencian siempre como direcciones relativas al comienzo de su registro de activación, o bien al comienzo de la zona de variables locales. Por tanto, cuando sea necesario acceder desde un procedimiento a variables definidas de otros procedimientos cuyo ámbito sea accesible, será necesario proveer dinámicamente la dirección de comienzo de las

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

variables de ese procedimiento. Para ello se utiliza dentro del registro de activación el puntero al registro de activación del padre. Este puntero, señalará al comienzo de las variables locales del procedimiento inmediatamente superior (en el que el llamado se encuentra declarado). El puntero de enlace ocupa una posición fija con respecto al comienzo de sus variables locales. Cuando los procedimientos se llaman a sí mismos recursivamente, el ámbito de las variables impide por lo general que una activación modifique las variables locales de otra activación del mismo procedimiento, por lo que en estos casos el procedimiento inmediato superior será, a efectos de enlace, el que originó la primera activación, tal como puede apreciarse en la figura 9.6. Es importante notar que el concepto de "procedimiento padre" es independiente del de "procedimiento llamador", aunque puede coincidir con éste.

9.3.3 Montón (*heap*)

Cuando el tamaño de un objeto a colocar en memoria puede variar en tiempo de ejecución, no es posible su ubicación en la pila, y mucho menos en la zona de datos de tamaño fijo. Son ejemplos de este tipo de objetos las listas y los árboles dinámicos, las cadenas de caracteres de longitud variable, etc. Para manejar este tipo de objetos el compilador debe disponer de un área de memoria de tamaño variable y que no se vea afectada por la activación o desactivación de procedimientos. Este trozo de memoria se llama montón (del inglés *heap*). En aquellos lenguajes de alto nivel que requieran el uso del *heap*, el compilador debe incorporar en el programa objeto generado, todo el código correspondiente a la gestión de éste; este código es siempre el mismo para todos los programas construidos. Las operaciones básicas que se realizan sobre el *heap* son:

- Alojar: se solicita un bloque contiguo de memoria para poder almacenar un ítem de un cierto tamaño.
- Desalojar: se indica que ya no es necesario conservar la memoria previamente alojada para un objeto y que, por lo tanto, ésta debe quedar libre para ser reutilizada en caso necesario por otras operaciones de alojamiento..

Según sea el programador o el propio sistema el que las invoque, estas operaciones pueden ser explícitas o implícitas respectivamente. En caso de alojamiento explícito, el programador debe incluir en el código fuente, una por una, las instrucciones que demandan una cierta cantidad de memoria para la ubicación de cada dato o registro (por ejemplo, Pascal proporciona la instrucción **new**, C proporciona **malloc**, etc.). La cantidad de memoria requerida en cada operación atómica de alojamiento, puede ser calculada por el compilador en función del tipo correspondiente al objeto que se desea alojar, o bien puede especificarla el programador directamente. El resultado de la función de alojamiento es por lo general un puntero a un trozo contiguo de memoria dentro del *heap* que puede usarse para almacenar el valor del objeto. Los lenguajes de programación imperativos suelen utilizar alojamiento y desalojamiento explícitos. Por el contrario los lenguajes declarativos (lógicos y funcionales) hacen la mayoría de estas operaciones implícitamente debido a los

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

mecanismos que subyacen en su funcionamiento.

La gestión del *heap* requiere técnicas adecuadas que optimicen el espacio que se ocupa (con objeto de impedir la fragmentación de la memoria) o el tiempo de acceso, encontrándose contrapuestos ambos factores como suele suceder casi siempre en informática. A este respecto, las técnicas suelen dividirse en dos grandes grupos: aquéllas en las que se aloja exactamente la cantidad de memoria solicitada (favorecen la fragmentación), y aquéllas en las que el bloque alojado es de un tamaño parecido, pero generalmente superior, al solicitado, lo que permite que el *heap* sólo almacene bloques de determinados tamaños prefijados en el momento de construir el compilador (favorece la desfragmentación).

Para finalizar, una de las características más interesantes de lenguajes imperativos muy actuales, como Java, es que incluyen un recolector automático de basura (*garbage collection*). El recolector de basura forma parte del gestor del *heap*, y lleva el control sobre los punteros y los trozos de memoria del *heap* que son inaccesibles a través de las variables del programa, ya sea directa o indirectamente, esto es, de los trozos de memoria alojados pero que no están siendo apuntados por nadie y que, por lo tanto, constituyen basura. Mediante este control pueden efectuar de manera automática las operaciones de desalojamiento, liberando al programador de realizar explícitamente esta acción. Entre las técnicas utilizadas para implementar los recolectores de basura podemos citar los contadores de referencia, marcar y barrer, técnicas generacionales, etc.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike