Qüestió - V. 9, n.º 4 (desembre 1985) pp. 257-266

GESTIÓN DEL ESPACIO LIBRE EN LA MEMORIA POR ACTUALIZACIÓN CONTINUA

LL. PÉREZ VIDAL, R. VILA
UNIVERSITAT POLITÈCNICA DE CATALUNYA

En la asignación dinámica de espacio en memoria se plantea el problema de la identificación y la compactación de los intervalos de memoria que ya no son utilizados por ningún proceso, estos intervalos están dispersos entre los utilizados y se trata de reunirlos todos en un extremo para conseguir un espacio libre único de mayor dimensión. En las aplicaciones gráficas que utilizan el estandar GKS se pueden asignar y desasignar segmentos gráficos en el "display file", y, debido a sus grandes requerimientos de memoria, se presenta, de forma aguda, el problema general descrito. Se propone un algoritmo para resolver el problema particular, que es susceptible de generalización sin modificaciones sustanciales.

Keywords: MEMORY MANAGEMENT, GARBAGE COLLECTION.

1. INTRODUCCION.

Los usuarios de un sistema informático (y también los ingenieros de sistemas) se enfrentan muy a menudo con el problema de que, por grande que sea la memoria disponible en el ordenador, la cantidad de posiciones de almacenamiento con que trabajan son un recurso escaso y que, a menudo, se les acaba.

Sin embargo, este agotamiento del "recurso memoria" es, a veces, aparente; puede ocurrir que el sistema operativo (u otro programa de control, en cualquier caso, quien se encargue de la gestión del espacio en memoria) vaya destinando espacio en memoria a cada demanda según un esquema FIFO ("first come, first served") y que"por detrás" de esta asignación vaya quedando espacio libre sin aprovechar.

El objetivo del trabajo descrito en este artículo ha sido precisamente el de diseñar un paquete de algoritmos destinados a pasar estos espacios libres (no utilizados) de memoria intercalados entre fragmentos ocupados hacia un extremo de la memoria, con la finalidad de conseguir la mayor fracción posible de memoria fisicamente adyacente (contínua) y que esté libre para el usuario.

La recuperación eficiente de los espacios vacíos en la memoria ha sido tratada por diferentes equipos de investigadores con perspectivas y finalidades diversas.

Algunos diseñadores de sistemas /5/, enfocan el tema de forma general, con la finalidad de proceder a una gestión puntual del espacio libre y del espacio ocupado: proceden a un recorrido secuencial completo de los sectores de memoria libres, que están organizados en forma de lista de nudos enlazados unidireccionalmente. Su preocupación principal no es tanto la de solucionar una memoria excesivamente fragmentada como la de lograr asignaciones de espacio libre lo más rápido que sea posible.

Algunos diseñadores de compiladores /1/, proponen algoritmos que asignan el espacio de memoria de forma dinámica (stacks), y otros autores /6/ interesados en utilizar el lenguaje Lisp en aplicaciones de Inteligencia Artificial presentan un algoritmo que reorganiza el espacio asignado en función de la

- Ll. Pérez Vidal i R. Vila - U.P.C. - Escola Tècnica Superior d'Enginyers Industrials de Barcelona Dep. de Mètodes Informàtics - Av. Diagonal, 647 - Barcelona 08028

- Article rebut el març del 1985.

utilización que el programa hace de los segmentos de memoria. Proponen una variante de la recuperación de segmentos a partir del tiempo transcurrido desde la última utilización.

Para la programación del sistema existe /2/ también un procedimiento que se basa en un contador de referencia: cada bloque de programa que referencia un determinado segmento de memoria incrementa el contador y cuando se desactiva el bloque se decrementa el contador. Todo segmento de memoria con contador cero es susceptible de compactación.

En este artículo presentamos un algoritmo de compactación del espacio libre que se apoya en la gestión contínua de la memoria: cada vez que se produce una demanda de memoria se actualizan los punteros de enlace de los espacios libres, asignándose a la demanda el mayor segmento disponible. Se contemplan también las acciones a ejecutar cuando se produce un desbordamiento aparente al llenar un segmento. El algoritmo se destina a casos en que la demanda no puede precisar el tamaño de memoria requerido.

2. DESARROLLO.

Este artículo parte de la gestión de memoria prevista para la implementación en ordenador de un standard gráfico: El GKS. El standard gráfico GKS establece la posibilidad de agrupar las primitivas de dibujo en unidades llamadas 'segmentos'. Estas unidades permiten identificar partes de un gráfico para luego aplicarles funciones de transformación geométrica o puramente de gestión de la información gráfica. Estas funciones también vienen establecidas en el standard.

Los segmentos están agrupados en una estructura de orden superior llamada 'display file'. El GKS sólo establece que información debe contener el display file, y que funciones debe soportar una implementación del mismo; pero no define que estructura concreta deben tener los datos, ni que formato.

Aqui se pretende explicar como se han resuelto estos 'cabos sueltos' en nuestra implementación.

2.1. ESTUDIO PREVIO.

Los segmentos están formados por dos tipos de información bien definidos:

1. La 'cabecera', que contiene información relativa al segmento propiamente dicho como es el nombre y los atributos. Esta información es cuantificable por cuanto está perfectamente definida.

2. El 'cuerpo' del segmento que contiene las primitivas de dibujo asociadas. Esta información es totalmente indefinida ya que puede variar en dos sentidos. El número de primitivas por segmento y el número de puntos por primitiva. Además, tampoco está definido el formato de estos datos.

Esta dualidad sugiere una división física de la información en dos estructuras distintas: Por un lado las cabeceras, formando una lista de un área fija de memoria y con una estructura definida, y por otro una zona de memoria del mayor tamaño posible segmentada en bloques de longitud variable (uno para cada cabecera) que contienen las primitivas de dibujo. A partir de ahora me referiré a las dos áreas como 'display file' (tratándose de la lista de cabeceras) y 'display program' (cuando se trate de los cuerpos de segmentos).

Por otra parte, las funciones que se realizarán con más frecuencia con los segmentos son las de búsqueda (por el nombre) y el dibujo del conjunto de segmentos según su prioridad (uno de los atributos asociados a cada segmento). Para conseguir esto con la máxima rapidez, el display file (o lista de cabeceras) no es secuencial, sino que está doblemente enlazado: por nombre (con los segmentos ordenados alfabeticamente) y por prioridad (con los segmentos ordenados de mayor a menor prioridad). Esto obliga a que las demás funciones de gestión de los segmentos (creación o borrado) así como las que modifiquen el nombre o la prioridad mantengan esta ordenación.

Tanto las cabeceras del display file como los bloques de memoria del display program, pueden tener dos status: 'vacío' o 'lleno'.

Inicialmente todas las cabeceras están 'vacías' ya que no existe ningún segmento definido. Cada vez que se cree un nuevo segmento deberá tomarse una de estas cabeceras -- 'vacías'. Para saber cuales están vacías u ocupadas por un segmento, las cabeceras vacías forman una pila enlazada (LIFO), con una variable auxiliar que hace de puntero - de pila (stack pointer) y utilizando uno de los dos punteros de enlace de las cabeceras (el de nombre) como puntero al siguiente -- elemento de la pila.

En cuanto a los bloques de memoria, cada -- vez que se borra un segmento, su bloque asociado pasa al status de 'vacío', ocupando una zona de memoria que debe ser aprovechada por sucesivos segmentos de la forma más eficiente posible. La gestión de estos bloques se realiza considerándolos como segmentos vacíos, o sea, que ocupan una cabecera aunque en este caso no están ordenados por nombre y por prioridad (ya que no tiene sentido) sino por la dirección absoluta de memoria donde empiezan y por su longitud, lo que nos da un total de cuatro listas a gestionar: dos para segmentos y dos para bloques vacíos, además de una pila. Se precisan además cuatro variables auxiliares que indiquen el primer elemento de cada lista.

Resumiendo, hay tres tipos de cabeceras en el display file:

1. Cabeceras vacías (o no utilizadas) formando una pila.

2. Cabeceras llenas conteniendo un segmento, formando dos listas enlazadas: por nombre y por prioridad.

3. Cabeceras llenas conteniendo un bloque libre de memoria y formando también -- dos listas: por dirección y por tamaño.

La necesidad de mantener enlazadas las cabeceras 'segmento' por nombre y por prioridad ya ha sido justificada anteriormente. En -- cuanto a los bloques libres, se mantienen enlazados por longitud porque al crear un nuevo segmento (ya que a priori no puede saberse que longitud tendrá) se le asigna el bloque libre de mayor tamaño, y el enlace - por dirección será de gran utilidad cuando la fragmentación del display program obligue a hacer una compactación (garbage collection). Estos puntos se desarrollarán más detenidamente en el capítulo 3 al describir las rutinas de gestión de display file.

2.2. ESTRUCTURA DE DATOS

El comportamiento del display file se simula con estructura del tipo ARRAY:

```
TYPE
vis= (visible, invisible);
hig= (highlighted, normal);
det= (detectable, indetectable);
cabecera=
        RECORD
        nombre          : CHAR    ;
        direccion       : INTEGER ;
        longitud        : INTEGER ;
        prioridad       : REAL    ;
        visibilidad     : vis     ;
        highlighting    : hig     ;
        detectabilidad  : det     ;
        epl             : INTEGER ;
        end             : INTEGER ;
        END ;
VAR
dfile           : ARRAY [-2..maxsegmentos] OF cabecera ;
dprogram        : ARRAY [1..total_memoria] OF INTEGER ;
stack_pointer   : INTEGER ; Stack pointer de cabeceras.
punseg          : INTEGER ; Apuntador al segmento
                            actualmente abierto.
longseg         : INTEGER ; Longitud actaul ocupada
                            de segmento abierto.
gacofet         : BOOLEAN ; Indica si se ha hecho o no
                            un Garbage Collection.
```

donde cabecera es un registro cuyos campos significan:

nombre	: Nombre del segmento (Dos caracteres).
direccion	: Direccion fisica de memoria donde empieza el bloque que contiene el segmento.
longitud	: Longitud del bloque.
prioridad, visibilidad, highlighting, detectabilidad	: Atributos del segmento.
epl	: Enlace prioridad-longitud. Cuando la cabecera pertenece a un segmento, este sirve para enlazar por prioridad (p); si pertenece a un bloque libre, sirve para enlazar por longitud (l).
end	: Enlace nombre-direccion. Cuando la cabecera pertenece a un segmento, este sirve para enlazar por nombre (n) ; si pertenece a un bloque libre, sirve para enlazar por direccion (d).

El display file es un ARRAY de cabeceras 'válidas' (del 1 a un máximo que depende de la implementación) al que se añaden tres cabeceras 'ficticias' (del -2 al 0) con funciones auxiliares, que contienen los apuntadores a los primeros elementos de cada lista y sirven para simplificar los algoritmos de inserción, borrado, etc....

2.3. INICIALIZACIONES.

Para el correcto funcionamiento de los algoritmos es necesario que las estructuras de datos estén correctamente inicializados. La situación inicial de las estructuras definidas en el apartado anterior es la siguiente:

El display program se inicializa a ceros.

```
FOR i:= 1 TO total_memoria DO dprogram[i]:= 0 ;
```

Las cabeceras del display file se inicializan formando una pila con el enlace nombre-dirección (end). Los demás valores son los de defecto que fija el GKS.

```
FOR i:= -2 TO maxsegmentos DO
 WITH dfile[i] DO
  BEGIN
  nombre:= '__' ; direccion:= 0 ; longitud:= 0 ;
  prioridad:= 0.0 ; visibilidad:= visible ;
  highlighting:= normal ; detectabilidad:= indetectable ;
  epl:= 0 ; end:= i + 1
  END ;
dfile[maxsegmentos].end:= 0 ;
stack_pointer:= 2 ;
```

Las cabeceras 'ficticias' se inicializan como sigue: La cabecera -2 contiene los apuntadores al primer segmento de las listas de

segmentos. Como a efectos prácticos es el primer segmento tanto por nombre como por prioridad, tiene una prioridad superior a la máxima permitida (1.0) y un nombre 'menor' al mínimo permitido (A). Como inicialmente no hay ningún segmento creado, los enlaces tienen valor cero.

```
WITH dfile[-2] DO
 BEGIN
 nombre:= '@' ; prioridad:= 2.0 ; epl:= 0 ; end:= 0
 END ;
```

La cabecera -1 contiene los apuntadores al primer bloque de las listas de bloques libres. Como a efectos prácticos es el primer bloque tanto por dirección como por longitud, tiene una dirección menor a la mínima permitida (0) y una longitud igual al total de memoria disponible más 1. Inicialmente existe un único bloque de dirección 1 y longitud igual al total de memoria, ocupando la cabecera 1; por eso, los enlaces de la cabecera -1 apuntan a este bloque.

```
WITH dfile[-1] DO
 BEGIN
 direccion:= 0 ; longitud:= total_memoria + 1 ;
 epl:= 1 ; end:= 1
 END ;
```

La cabecera 0 hace las funciones de tope superior de las 4 listas. Tiene un nombre 'mayor' que el máximo permitido, una dirección también mayor a la máxima permitida, una longitud menor que la mínima permitida y prioridad también menor a la mínima permitida.

```
WITH dfile[0] DO
 BEGIN
 nombre:= '[[' ; direccion:= total_memoria+1 ; longitud:= 0 ;
 prioridad:= -1.0 ; epl:= 0 ; end:= 0
 END ;
```

La cabecera 1 es la primera de las 'válidas' y inicialmente está ocupada por el primer bloque libre.

```
WITH dfile[1] DO
 BEGIN
 direccion:= 1 ; longitud:= total_memoria ;
 epl:= 0 ; end:= 0
 END ;
```

Gracias al uso de las cabeceras ficticias se simplifican mucho las funciones de gestión (Abrir, cerrar, borrar ...) de segmentos, ya que no hay que diferenciar casos de insertar o borrar el primer elemento de la lista; ni hay que controlar los bulces de búsqueda de

un segmento ya que existe un tope final.

3. DESCRIPCION DE LAS RUTINAS DE GESTION DEL DISPLAY FILE.

3.1. APERTURA DE SEGMENTO.

Como ya se ha comentado, cuando se efectúa la apertura de un segmento, se le asigna el bloque libre de mayor tamaño que se conozca en este instante. El motivo de este proceder es específico de nuestra aplicación ya que al abrir un segmento no se sabe cual será su longitud final. El procedimiento de abrir -- segmento queda:

```
PROCEDURE open_segment ;
BEGIN
 - Seleccionar como bloque escogido el primero por tamano.
 - Actualizar el enlace a primer bloque por longitud
   de la cabecera -1.
 - Desenlazar el bloque escogido de la lista de bloques
   por direccion.
 - Inicializar la cabecera del bloque escogido.
END ;
```

3.2. CLAUSURA DE SEGMENTO.

Al clausurar un segmento, hay que hacer dos puestas al día: primero, en las cabeceras de segmentos llenos, hay que insertar el segmento recién cerrado en la lista por priorida-- des y en la lista por nombres. Y, segundo,es previsible que el segmento no haya ocupado - la totalidad del espacio disponible en el -- agujero que se le asignó al abrirlo, por lo que hay que tratar el espacio restante, colocándolo en su lugar correspondiente de la -- lista de bloques vacíos tanto por tamaño como por dirección:

```
PROCEDURE close_segment ;
BEGIN
 - Insertar al final de la lista de segmentos por
   prioridad.
 - Insertar en la lista de segmentos por nombre.
 - Calcular el espacio ocupado.
 - Tratar el espacio libre sobrante:
   IF longitud utilizada < espacio_total AND
      stack_pointer <> 0 THEN
      - Crear una nueva cabecera del tipo bloque
        con el espacio sobrante.
      - Insertar en la lista de bloques por longitud
        el nuevo bloque.
      - Insertar en la lista de bloques por direccion
        el nuevo bloque.
   ENDIF ;
 - Actualizar los valores de la cabecera del
   segmento a cerrar.
END ;
```

3.3. BORRADO DE UN SEGMENTO

Para borrar un segmento hay que extraerlo de ambas listas de segmentos llenos, y luego introducir el bloque que ocupa en las listas de bloques vacíos. Esto último admite dos tratamientos: el primero, más simple (que adoptaremos), consiste en insertar este bloque en las listas de bloques vacíos, por tamaño y por dirección; el segundo consistiría en comprobar además si hay algún bloque contiguo al cual se podría añadir para formar un bloque mayor:

```
PROCEDURE delete_segment ;
BEGIN
 - Eliminar de la lista de segmentos por nombre.
 - Eliminar de la lista de segmentos por prioridad.
 - Incluir en la lista de bloques por tamano.
 - Incluir en la lista de bloques por direccion.
END ;
```

3.4. COMPACTACION (GARBAGE COLLECTION).

Durante la ejecución del programa de gestión del display file se observa que el display program tiene una fuerte tendencia a fragmentarse en bloques cada vez más pequeños, lo que impide un aprovechamiento eficaz de la memoria disponible. Ante este hecho es necesario compactar los segmentos en memoria, desplazándolos hacia las posiciones más bajas de memoria, de forma que se obtenga al final un único bloque de memoria libre detrás de los segmentos. Graficamente:

```
Estado inicial:
-----------------------------------------------------------------
111111111122222AAAAAAAAABBBBBBBBB3333333CCCCCCCCC4444455555
-----------------------------------------------------------------

Estado final:
-----------------------------------------------------------------
AAAAAAAAABBBBBBBBBCCCCCCCCC6666666666666666666666666666666
-----------------------------------------------------------------
```

donde las letras significan distintos segmentos, y los números distintos bloques de memoria libre.

La compactación tiene además la propiedad de 'liberar' cabeceras ocupadas por bloques de memoria libre. En efecto, en el ejemplo anterior, se precisaban seis cabeceras (una para cada bloque) en la situación inicial, mientras que después de la compactación sólo se precisa una; las otras se ponen en la pila y quedan disponibles para futuros segmentos.

Se han estudiado dos situaciones 'límite' en las que hay que compactar: La primera ocurre cuando se vacía la pila de cabeceras libres (como hemos visto, la compactación libera cabeceras). La segunda ocurre cuando durante la creación de un segmento se precisa de más espacio del que se le ha asignado al abrirlo.

En el segundo caso caben dos alternativas, que estarán igualmente disponibles para el usuario:

1. La primera es evidente y consiste en cerrar el segmento, de forma que el usuario pierde la información restante; quedando bajo su responsabilidad el borrar otros segmentos para conseguir espacio libre.

2. La segunda consiste en cerrar provisionalmente el segmento, hacer una compactación, copiar sobre un nuevo bloque (con más espacio) el contenido del antiguo dejándolo abierto para seguir trabajando, y finalmente, borrar el antiguo segmento.

La segunda alternativa no garantiza que se consiga siempre más espacio, pero da soporte al usuario, al cual (en último término) siempre le queda el recurso de aplicar la primera.

El procedimiento puede expresarse básicamente como:

```
PROCEDURE garbage_collection ;
BEGIN
WHILE exista bloque DO
  - Eliminar bloques vacios contiguos.
  - Buscar el primer segmento que este detras.
  - IF existe THEN
      - Permutar el segmento con el bloque.
      - Actualizar direcciones.
    ENDIF
ENDWHILE
END ;
```

Seguidamente se presenta un ejemplo de compactación en forma "gráfica". En el ejemplo, las letras (A,B y C) representan segmentos; y los números (1,2,3,4 y 5), bloque de memoria libre. El algoritmo empieza con el primer bloque libre por dirección y mientras encuentra bloques contiguos se va expansionando. Al llegar a un segmento, permuta el bloque libre con el segmento. Con estas dos acciones básicas (expandir y permutar) se consigue que los segmentos "bajen" a las direcciones bajas de memoria y que el espacio libre "suba" a las direcciones más altas.

4. CONCLUSION.

Se han presentado los algoritmos y las rutinas para la gestión de segmentos de memoria en una aplicación gráfica interactiva. Frente a algoritmos clásicos de compactación de memoria, el presente se caracteriza por una actualización de los punteros en cada demanda; esto representa un tiempo ligeramente superior para el tratamiento de cada demanda, pero permite una compactación mucho más rápida cuando ésta se hace necesaria.

5. REFERENCIAS.

/1/ BISHOP, P.: "Computer systems with a very large address space and garbage collection". Tech.Rept. TR-178, MIT Lab for Computer -- Science. Cambridge, Mass. Mayo 1977.

/2/ DEUTSCH, L.P. Y BOBROW, D.G.: "An efficient incremental, automatic garbage collector". Commun ACM 19,9 (sept 1976), 522-526.

/3/ DIJKSTRA, E., LAMPORT, L.et.al.: "On-the-fly garbage collection: an excercise in cooperation". Commun.ACM 21,11(nov.78)966-975

/4/ FRIEDMAN, D. AND WISE, D.: "Garbage collecting a heap which includes a scatter table". Inf. Process. Lett. 5,6 (dic.1976).

/5/ HOROWITZ, E. AND SAHNI, S.: "Fundamentals of Data Structures". Pitman. London. 1976.

/6/ LIEBERMAN, H. AND HEWITT, C.: "A real time garbage collector based on the lifetimes of objects". Commun ACM 26,6(june 83) 419-429.

/7/ KNUTH, D.: "Garbage collection in real time" Apuntes del curso CS144C. Standford Univ., Standford, Calif. 1981.

6. ANEXO: ALGORITMOS DE COMPACTACION

Seguidamente se incluye los listados de los programas de compactación de segmentos de la implementación del standard GKS antes mencionado.

```
(*--GARBAGE COLLECTION-----------------------------------*)

PROCEDURE garbage_collection ;
VAR     i, dir_inicial, dir_final,
        desplazamiento,
        sactual, bactual, bsiguiente     :        INTEGER ;
        hay_segmento, parar              :        BOOLEAN ;
BEGIN
bactual:= dfile[-1].end ; hay_segmento:= TRUE ;
WHILE (bactual <> 0) AND hay_segmento DO
  BEGIN

  { Elimina bloques vacios contiguos }
  bsiguiente:= dfile[bactual].end ; parar:= FALSE ;
  WHILE (bsiguiente <> 0) AND (NOT parar) DO
    WITH dfile[bsiguiente] DO
      BEGIN
      i:= dfile[bactual].direccion + dfile[bactual].longitud
      IF direccion = i THEN
        BEGIN
        direccion:= dfile[bactual].direccion ;
        longitud:= longitud + dfile[bactual].longitud ;
        dfile[-1].end:= bsiguiente ;
        WITH dfile[bactual] DO
          BEGIN
          nombre:= '__' ; direccion:= 0 ; longitud:= 0 ;
          ep1:= 0 ; end:= stack_pointer
          END ;
        stack_pointer:= bactual ;
        bactual:= bsiguiente ;
        bsiguiente:= dfile[bactual].end
        END
      ELSE parar:= TRUE ;
      END ;

  dir_inicial:= dfile[bactual].direccion ;
  desplazamiento:= dfile[bactual].longitud ;
  dir_final:= dir_inicial + desplazamiento ;

  { Busca el primer segmento que este detras }

  sactual:= dfile[-2].end ; parar:= FALSE ;
  WHILE (sactual <> 0) AND NOT parar DO
    IF dfile[sactual].direccion = dir_final
      THEN parar:= TRUE
      ELSE sactual:= dfile[sactual].end ;
  IF sactual=0 THEN hay_segmento:= FALSE
  ELSE
    BEGIN
    {Si existe desplaza el segmento y actualiza direcciones}

    dir_final:= dir_inicial + dfile[sactual].longitud - 1 ;
    FOR i:= dir_inicial TO dir_final DO

       BEGIN
       dprogram[i]:= dprogram[i + desplazamiento] ;
       dprogram[i + desplazamiento]:= CHR (bactual + 48)
       END ;
     dfile[sactual].direccion:= dir_inicial ;
     dfile[bactual].direccion:= dir_final + 1
     END
  END ;
bactual:= dfile[-1].end ;
dfile[-1].ep1:= bactual ;
WITH dfile[bactual] DO BEGIN ep1:= 0 ; end:= 0 END ;
FOR i:= 1 TO dfile[bactual].longitud DO
  dprogram[dfile[bactual].direccion+i-1]:= CHR(bactual+48) ;
gacofet:= TRUE ;
END ;
```

```
(*--CASO LIMITE-----------------------------------------------*)

PROCEDURE casolimite ;
LABEL   01 ;
VAR
        actual, k, segmento,
        dir_i_move, dir_f_move,
        direccion_bl, longitud_bl,
        dir_inicial, bytesquedan        :       INTEGER ;
        acabar                          :       BOOLEAN ;

  PROCEDURE mover (longitud : INTEGER) ;
  BEGIN
    FOR k:= 0 TO longitud - 1 DO
      dprogram[direccion_bl+k]:= dprogram[dir_inicial+k] ;
    dir_i_move:= dir_inicial + longitud ;
    dir_f_move:= direccion_bl + longitud - 1 ;
    FOR k:= dir_i_move TO dir_f_move DO
      dprogram[k-longitud]:= dprogram[k]
  END ;

BEGIN
k:= dfile[-1].epl ; IF NOT (k > 0) THEN GOTO 01 ;
segmento:= punseg ;
WITH dfile[k] DO
  BEGIN
  direccion_bl:= direccion ; longitud_bl:= longitud
  END ;
WITH dfile[segmento] DO
  BEGIN
  dir_inicial:= direccion ; bytesquedan:= longitud
  END ;
acabar:= FALSE ;
REPEAT

  IF bytesquedan > longitud_bl THEN
    BEGIN
    mover (longitud_bl) ;
    bytesquedan:= bytesquedan - longitud_bl
    END
  ELSE
    BEGIN mover (bytesquedan) ; acabar:= TRUE END
UNTIL acabar ;

{ Actualizar enlaces }

actual:= dfile[-2].end ;
WHILE actual <> 0 DO
  WITH dfile[actual] DO
    BEGIN
    IF direccion > dfile[segmento].direccion THEN
      direccion:= direccion - dfile[segmento].longitud ;
    actual:= end
    END ;
WITH dfile[segmento] DO
  BEGIN
  direccion:= direccion_bl - longitud ;
  longitud:= total_memoria - direccion+1 ;
  END ;
k:= dfile[-1].epl ;
WITH dfile[k] DO
  BEGIN
  direccion:= 0 ; longitud:= 0 ;
  end:= stack_pointer
  END ;
stack_pointer:= k ;
WITH dfile[-1] DO BEGIN epl:= 0 ; end:= 0 END ;
01 : ;
END ;
```