Ampliación de Sistemas Operativos: alloc.c y break.c

Manuel A. Díaz Sánchez y Joaquín Caraballo Moreno

Abril, 2001

Índice General

1 Introducción

En el manejo de memoria en Minix no se utiliza paginación ni segmentación. El manejador de memoria mantiene una lista encadenada de huecos ordenada por la dirección de cada hueco. Cuando se necesita memoria, debido a una llamada `fork()` o `execve()`, se recorre la lista encadenada buscando el primer hueco que sea lo suficientemente grande para satisfacer la petición. Una vez que un proceso se ha colocado en memoria, permanece exactamente en el mismo lugar hasta que termina su ejecución, el área ubicada para el proceso nunca crece ni disminuye.

La figura 1 muestra la disposición en la memoria física de los diferentes segmentos de un proceso. Cuando un proceso necesita más memoria (por ejemplo,

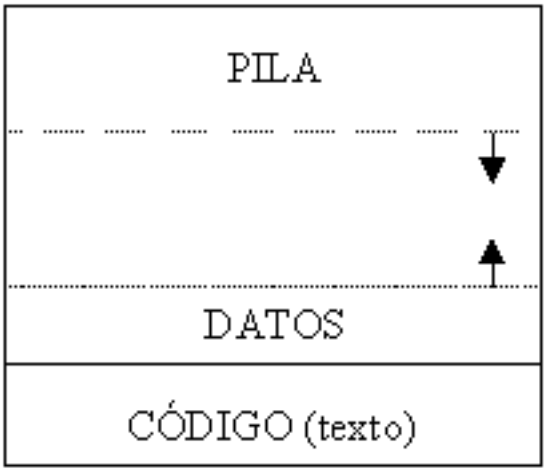

Figura 1: Segmentos de un proceso

para datos dinámicos), ésta se toma del hueco que hay entre datos y pila, haciendo crecer el tamaño del segmento de datos. Si las fronteras de los segmentos se llegaran a cruzar, el proceso simplemente se mata, resultando retirado de la memoria (con el consiguiente core dump).

2 Fichero `alloc.c`

El módulo `alloc.c` se utiliza para ubicar y liberar memoria destinada a la implementación de las llamadas `fork(void)` y `execve(name, argv, envp)`. En este fichero el sistema lleva una cuenta de los bloques de memoria física que están libres.

Los puntos de entrada en el módulo son las funciones:

- `mem_init()`
- `max_hole()`
- `alloc_mem()`
- `free_mem()`

En el mismo fichero se encuentran otras dos funciones que no son llamadas desde el exterior:

- `merge()`
- `del_slot()`

En la figura 2 se muestra el grafo de las llamadas que se van realizando para provocar la activación de las cuatro primeras funciones, así como la única comunicación provocada desde el propio módulo, que tiene lugar cuando `mem_init()` envía mensajes `SYS_MEM` al núcleo para solicitar que le indique los huecos de memoria libres que hay inicialmente.

El manejo de la memoria libre se implementa mediante un vector llamado `hole`. Este vector tiene `NR_HOLES` entradas, cada una con tres campos: el tamaño del hueco al que representa, su dirección base y un puntero hacia el nodo sucesor. Con estos datos se forma la lista de huecos libres, que permanecerá apuntada por `hole_head`. Al mismo tiempo, en el mismo vector se enlazan las entradas no usadas mediante una lista apuntada por free_slots.

```
#define NR_HOLES         128    /* nº de entradas en la tabla
                                 * de huecos */
#define NIL_HOLE (struct hole *) 0

PRIVATE struct hole {
 phys_clicks h_base;           /* ¿dónde empieza el hueco? */
 phys_clicks h_len;            /* ¿cuánto ocupa el hueco?  */
 struct hole *h_next;          /* puntero a la siguiente
                                * entrada en la lista      */
} hole[NR_HOLES];


PRIVATE struct hole *hole_head;  /* puntero al primer hueco */
PRIVATE struct hole *free_slots; /* puntero a la lista de
                                  * entradas libres de la
                                  * tabla */
```

En los siguientes subapartados, se describen cada una de las funciones que conforman `alloc.c`.

2.1 Función `alloc_mem`

Parámetros:

- `click`: el tamaño en clicks del bloque pedido

Llamadas a otras funciones:

- `del_slot`

La función `alloc_mem` es empleada para tomar un bloque memoria libre y proporcionárselo a un nuevo proceso. Los huecos disponibles se encuentran registrados en una lista encadenada apuntada por `hole_head`. Cada nodo de esta

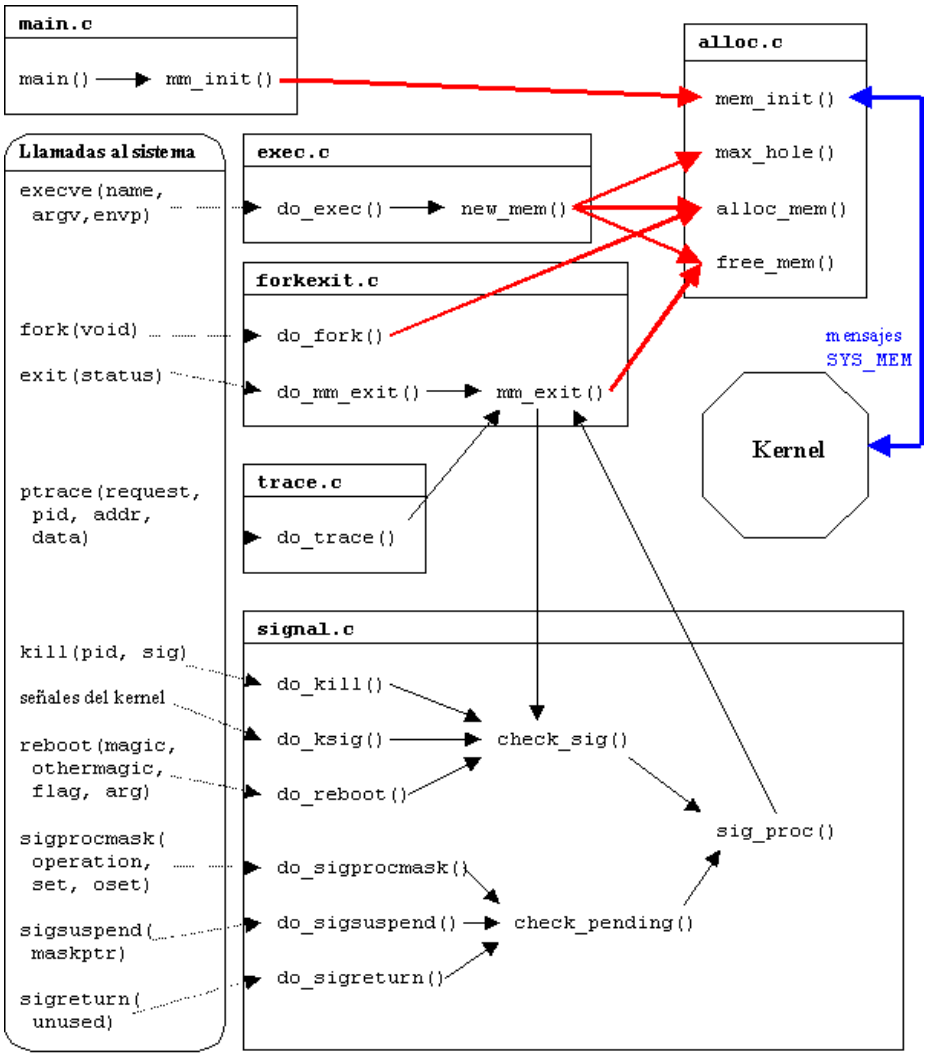

Figura 2: Relación de llamadas exteriores a `alloc.c`

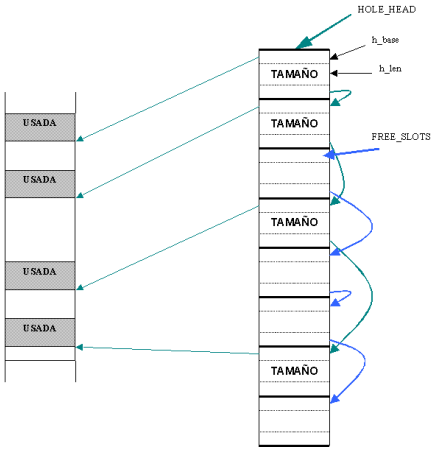

Figura 3: Listas `hole_head` y `free_slots` en el vector `hole`

lista contiene el tamaño en clicks[1] (campo `h_len`) y la ubicación en memoria física de cada hueco libre (campo `h_base`). La función `alloc_mem(clicks)` recorre la lista buscando el primer hueco libre que pueda satisfacer la petición, es decir, cuya longitud, contenida en el campo `h_len`, verifique que `h_len` $\geq$ `clicks`, donde el parámetro `clicks` indica el tamaño en clicks del bloque pedido. Una vez encontrado un hueco lo suficientemente grande, se decrementa el tamaño `h_len` del hueco en un número de clicks igual al valor de `clicks` y se incrementa la dirección de comienzo del hueco contenida en `h_base`.

Es posible que, al decrementar, el hueco libre quede vacío, en cuyo caso el nodo de la lista es inútil y se invoca a la función `del_slot` (ver más abajo) para retirarlo de la lista. Durante la iteración de búsqueda del primer hueco libre por la lista encadenada se mantiene un puntero hacia el nodo antecesor del nodo actual en la lista. Este puntero, llamado `prev_ptr` es imprescindible para efectuar la llamada a la función `del_slot` en caso necesario.

Por último, si se encontró un hueco suficientemente grande, se devuelve la dirección de comienzo de la porción de hueco extraída, que viene a ser la dirección en la que comenzaba originalmente el hueco que hemos reducido, guardada antes del incremento `old_base`. Si la búsqueda falló, es decir, si se llegó al final de la lista sin hallar ningún hueco libre del tamaño adecuado, de devuelve el código de error `NO_MEM`.

2.2 Función `free_mem`

Parámetros:

- `base`, posición de memoria física del bloque
- `clicks`, longitud en clicks.

[1] 1 click = 16 bytes

Llamadas a otras funciones:

- `merge`

La función `free_mem` se utiliza para devolver bloques de memoria física, de un proceso que ha dejado de existir, a la lista de huecos libres. Los parámetros que `free_mem(base, clicks)` acepta son la posición en memoria física del bloque, dada por `base`, y su longitud, dada por `clicks`. Lo primero que hace la función es comprobar si queda espacio en el vector `hole` para la entrada correspondiente a un nuevo bloque libre, lo que hace mirando si quedan nodos libres en una pila de nodos contenida en el propio vector `hole` y cuya cima está apuntada por `free_slots`. Si `free_slots` es `NULL`, entonces es que no quedan nodos libres para poder añadir la información de este nuevo hueco a la lista y entonces el sistema entra en un estado de pánico, invocando la función panic[2]. El número máximo de nodos que puede tener la lista de huecos es 128, pero se puede cambiar con sólo modificar el `#define NR_HOLES` que aparece en este mismo fichero.

Si quedan nodos libres en la pila se toma uno y se rellenan en él los campos `h_len` (tamaño) y `h_base` (comienzo en memoria física) con los valores de los parámetros `clicks` y `base`, respectivamente. Luego hay que insertar este bloque en la lista de huecos libres. Esta lista se mantiene ordenada por la dirección base de los huecos. Como en todo algoritmo de inserción en una lista encadenada, primero hay que mirar si el nuevo hueco va al comienzo de la lista o si la lista está vacía, ya que en estos casos la inserción se ejecuta de manera diferente: no hay nodo predecesor cuyo puntero sucesor haya que modificar y, además, tenemos que actualizar el puntero `hole_head` que apunta a la cabecera de la lista. Si el nuevo hueco no va al comienzo de la lista (es decir, si su dirección base es mayor que la dirección del primer hueco de la lista) entonces hay que buscar su posición en la lista mediante un bucle, manteniendo un puntero `prev_ptr` hacia el nodo predecesor del nodo actual, que será necesario para efectuar la inserción.

Finalmente, en ambos casos, se invoca a la función `merge` (ver más abajo). Esta función intenta fusionar el hueco recién insertado con su predecesor y/o sucesor, si esto fuera posible.

2.3 Función `merge`

Parámetros:

- `hp`: puntero hacia un nodo de la lista de nodos libres.

Llamadas a otras funciones:

- `del_slot`

[2]Esta función informa del error a la Tarea del Sistema, la cual hace que Minix entre en un estado de halt.

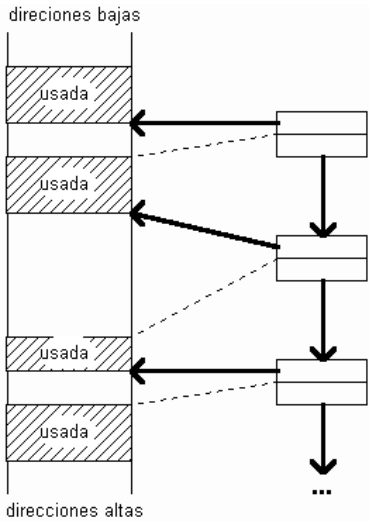

Figura 4: Cómo `merge` fusiona los huecos de memoria libre (antes)

Esta función intenta fusionar dos o tres huecos que han quedado contiguos en memoria física. Acepta un puntero `hp` hacia un nodo de la lista de nodos libres e intenta fusionarlo con su sucesor o con los dos siguientes nodos, si es posible. Esta función se invoca normalmente cuando se ha producido una liberación de algún bloque de memoria, siendo `hp` la dirección del nodo predecesor (en la lista encadenada de huecos) al nodo recién insertado[3].

Básicamente, la función trata de hacer dos fusiones sucesivas. Primero de `hp` con su sucesor; si lo consigue hp quedará sin modificar, ya que está apuntando al comienzo del nuevo hueco; si no lo consigue, `hp` se actualiza a la dirección del segundo hueco de los tres, para ver si puede fundirlo con el tercero. En ambos casos, la fusión no es posible si hp es el último nodo de la lista. Después de cada fusión que tiene éxito se hace `del_slot(hp, next_prt)` para retirar el nodo que ya no se usa de la lista encadenada.

2.4 Función `del_slot`

Parámetros:

- `prev_prt`: puntero hacia en nodo predecesor en la lista de huecos libres.
- `hp`: puntero hacia el nodo a extraer.

La función `del_slot` simplemente extrae el elemento indicado de la lista encadenada que enlaza las entradas del vector `hole` correspondientes a huecos de memoria libre. Se ha colocado aparte porque el mismo código se usa un par de

[3] ver figura 4

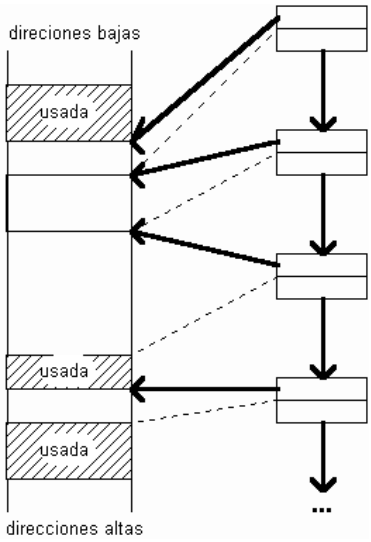

Figura 5: Cómo `merge` fusiona los huecos de memoria libre (después)

veces en el fichero. Los parámetros que acepta son el puntero `prev_ptr` hacia el nodo predecesor en la lista de huecos libres y el puntero `hp` hacia el propio nodo que vamos a extraer. La extracción se efectúa de la manera habitual (asignando el campo sucesor del nodo predecesor con el valor del campo sucesor `h_next` del nodo a extraer), con la salvedad hecha de que, si el nodo a extraer está en la cabecera de lista, entonces no hay predecesor y simplemente se avanza el puntero cabecera `hole_head` (se asigna con el valor del campo sucesor del nodo a extraer). La función no devuelve nada.

2.5 Función `max_hole`

Esta función es una pequeña utilidad para encontrar la longitud del mayor hueco de la lista encadenada de huecos libres. Su implementación no tiene gran relevancia. Simplemente recorre la lista encadenada registrando el tamaño del hueco de mayor longitud hallado hasta ahora. Al final devuelve el mayor tamaño encontrado. Devuelve cero si la lista de huecos libres está vacía.

2.6 Función `mem_init`

Parámetros:

- `total`: tamaño total de memoria ocupada.
- `free`: tamaño total de memoria libre.

Esta función se invoca sólo una vez durante la inicialización del manejador de memoria[4] y se encarga de inicializar la lista encadenada de nodos que representan huecos libres de la memoria física. En primer lugar, la función tiene que crear una pila de nodos libres cuyo top está señalado por el puntero `free_slots`. Los nodos para fabricar esta pila se toman del vector `hole`, el cual tiene un tamaño de 128 entradas y es por ello este valor el máximo número de huecos libres que puede mantener el manejador de memoria.

Luego, la función ejecuta un bucle donde va pidiendo a la Tarea del Sistema (su mediador con la memoria física) cada uno de los bloques de memoria física disponibles que puede utilizar el sistema. Para ello, en cada iteración del bucle, se envía un mensaje de tipo `SYS_MEM` a la Tarea del Sistema, la cual, en respuesta, le envía otro mensaje con la dirección física y el tamaño en clicks del siguiente bloque disponible de memoria. Se aprovecha el código de la función `free_mem` para insertar este nuevo bloque disponible en la lista encadenada de huecos libres, y se actualizan los valores de dos variables de salida apuntadas por `total` y `free`, que llevan la cuenta del tamaño total de la memoria y de la cantidad de memoria libre acumulada, respectivamente. El bucle termina cuando el mensaje de vuelta contiene un cero en el campo del tamaño del bloque libre (que indica que no hay más memoria disponible para registrar).

2.7 Código de `alloc.c`

```
#include "mm.h"
#include <minix/com.h>

#define NR_HOLES         128    /* nº de entradas en la tabla
                                 * de huecos */
#define NIL_HOLE (struct hole *) 0

PRIVATE struct hole {
 phys_clicks h_base;           /* ¿dónde empieza el hueco? */
 phys_clicks h_len;            /* ¿cuánto ocupa el hueco?  */
 struct hole *h_next;          /* puntero a la siguiente
                                * entrada en la lista      */
} hole[NR_HOLES];


PRIVATE struct hole *hole_head;  /* puntero al primer hueco */
PRIVATE struct hole *free_slots; /* puntero a la lista de
                                  * entradas libres de la
                                  * tabla */

FORWARD _PROTOTYPE( void del_slot, (
        struct hole *prev_ptr, struct hole *hp ) );
```

[4] véase la figura 2

```
FORWARD _PROTOTYPE( void merge, (
        struct hole *hp )                       );


/*============================================================*
 *                          alloc_mem                         *
 *============================================================*/
PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks; /* cantidad de memoria que se pide */
{
/* Ubica un bloque de memoria de la lista libre usando la
 * politica First Fit. El bloque consiste en una secuencia de
 * bytes consecutivos y su longitud viene dada en 'clicks'. Lo
 * que se devuelve es un puntero a un bloque. Esta funcion se
 * invoca cuando FORK o EXEC necesitan memoria */

  register struct hole *hp, *prev_ptr;
  phys_clicks old_base;

  hp = hole_head;
  while (hp != NIL_HOLE) {
        /* Mira si el hueco es lo suficientemente grande para
         * usarlo */
        if (hp->h_len >= clicks) {
                /* almacena el comienzo del hueco */
                old_base = hp->h_base;
                /* toma un trozo del hueco */
                hp->h_base += clicks;
                /* ajusta el tamaño */
                hp->h_len -= clicks;

                /* Si el hueco no se usa del todo lo reduce y
                 * sale. */
                if (hp->h_len != 0) return(old_base);
                /* Si se usa el hueco entero hay que manipular
                 * la lista */
                del_slot(prev_ptr, hp);
                return(old_base);
        }

        prev_ptr = hp;
        hp = hp->h_next;
  }
  return(NO_MEM);
}
```

```
/*===========================================================*
 *                          free_mem                         *
 *===========================================================*/
PUBLIC void free_mem(base, clicks)
phys_clicks base;   /* dirección del comienzo del bloque de
                     * memoria a liberar */
phys_clicks clicks; /* tamaño en clicks del bloque de memoria a
                     * liberar */
{
/* Devuelve un bloque de memoria libre a la lista de huecos.
 * Los parámetros indican la dirección física de comienzo del
 * bloque y su tamaño. Si el bloque es contiguo a otro anterior-
 * mente existente en la lista, el nuevo bloque es unido al
 * bloque o bloques contiguos */

 register struct hole *hp, *new_ptr, *prev_ptr;

  if (clicks == 0) return;
  if ( (new_ptr = free_slots) == NIL_HOLE)
        panic("Hole table full", NO_NUM);
  new_ptr->h_base = base;
  new_ptr->h_len = clicks;
  free_slots = new_ptr->h_next;
  hp = hole_head;

  /* Si la dirección del bloque es numéricamente menor que la
   * del bloque de dirección más baja actualmente disponible,
   * o si no hay ninguno disponible, se pone este hueco como
   * el primero de la lista */
  if (hp == NIL_HOLE || base <= hp->h_base) {
        new_ptr->h_next = hp;
        hole_head = new_ptr;
        merge(new_ptr);
        return;
  }

  /* El bloque no es el primero de la lista. */
  while (hp != NIL_HOLE && base > hp->h_base) {
        prev_ptr = hp;
        hp = hp->h_next;
  }

  /* Se encontró la posición y se inserta detrás de prev_ptr */
  new_ptr->h_next = prev_ptr->h_next;
  prev_ptr->h_next = new_ptr;
```

```
  merge(prev_ptr);  /* el orden de los bloques es 'prev_ptr',
                     * 'new_ptr', 'hp' */
}


/*============================================================*
 *                          del_slot                          *
 *============================================================*/
PRIVATE void del_slot(prev_ptr, hp)
register struct hole *prev_ptr; /* puntero al hueco inmediata-
register struct hole *hp;     /* puntero al hueco a eliminar */
{
/* Borra una entrada en la lista de huecos. Esta función se
 * llama cuando una petición de asignación de memoria toma un
 * hueco completo, por lo que se debe reducir el número de hue-
 * cos en memoria, para lo que se elimina una de las entradas
 * de la lista. */

  if (hp == hole_head)
        hole_head = hp->h_next;
  else
        prev_ptr->h_next = hp->h_next;
  hp->h_next = free_slots;
  free_slots = hp;
}


/*============================================================*
 *                           merge                            *
 *============================================================*/
PRIVATE void merge(hp)
register struct hole *hp; /* puntero al hueco a unir con sus
                           * sucesores */
{
/* Comprueba si hay huecos contiguos y los fusiona. Los huecos
 * contiguos aparecen depués de liberar un bloque de memoria,
 * pudiendo el bloque recién liberado ser contiguo tanto al
 * anterior como al siguiente. El puntero 'hp' apunta al primero
 * de una serie de tres huecos potencialmente fusionables. */

  register struct hole *next_ptr;

  /* Si 'hp' apunta al último hueco no hay fusión posible, si
   * no, se intenta unir a su sucesor y extraer la entrada de
   * éste de la lista de huecos libres. */
  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
```

```
  if (hp->h_base + hp->h_len == next_ptr->h_base) {
        hp->h_len += next_ptr->h_len; /* al primer hueco se le
                                       * añade la memoria del
                                       * segundo */
        del_slot(hp, next_ptr);
  } else {
        hp = next_ptr;
  }

  /* Si ahora 'hp' es el último se retorna; si no, se intenta
   * fusionar con su sucesor */
  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
  if (hp->h_base + hp->h_len == next_ptr->h_base) {
        hp->h_len += next_ptr->h_len;
        del_slot(hp, next_ptr);
  }
}


/*===========================================================*
 *                          max_hole                         *
 *===========================================================*/
PUBLIC phys_clicks max_hole()
{
/* Recorre la lista de huecos hasta encontrar el más grande. */

  register struct hole *hp;
  register phys_clicks max;

  hp = hole_head;
  max = 0;
  while (hp != NIL_HOLE) {
        if (hp->h_len > max) max = hp->h_len;
        hp = hp->h_next;
  }
  return(max);
}


/*===========================================================*
 *                          mem_init                         *
 *===========================================================*/
PUBLIC void mem_init(total, free)
phys_clicks *total, *free;  /* tamaños de memoria resultantes */
{
/* Inicializa la lista de huecos. Hay dos listas: 'hole_head'
```

```
 * apunta a una lista enlazada de todos los huecos (bloques de
 * memoria sin usar) del sistema; 'free_slots' apunta a una lis-
 * ta enlazada de las entradas de la tabla que no están siendo
 * usadas. Inicialmente, la primera tiene una entrada por cada
 * trozode de memoria física, y la segunda une las restantes en-
 * tradas de la tabla. A medida que la memoria se va fragmentan-
 * do (por ejemplo, los grandes bloques iniciales se fragmentan
 * en bloques más pequeños), se van necesitando nuevas entradas
 * para los nuevos huecos. Estas entradas se tomarán de la lista
 * encabezada por 'free_slots'. */

  register struct hole *hp;
  phys_clicks base; /* dirección base del trozo de memoria */
  phys_clicks size; /* tamaño del trozo de memoria */
  message mess;

  /* Pone todos los huecos en la lista de huecos de memoria
   * libre */
  for (hp = &hole[0]; hp < &hole[NR_HOLES]; hp++)
        hp->h_next = hp + 1;
  hole[NR_HOLES-1].h_next = NIL_HOLE;
  hole_head = NIL_HOLE;
  free_slots = &hole[0];

  /* Pide al kernel trozos de memoria y coloca cada uno de los
   * huecos. La llamada SYS_MEM responde con la base y el tama-
   * ño del siguiente bloque y con la cantidad total de
   * memoria. */
  *free = 0;
  for (;;) {
        mess.m_type = SYS_MEM;
        if (sendrec(SYSTASK, &mess) != OK)
              panic("bad SYS_MEM?", NO_NUM);
        base = mess.m1_i1;
        size = mess.m1_i2;
        if (size == 0) break;      /* ¿no hay más? */

        free_mem(base, size);
        *total = mess.m1_i3;
        *free += size;
  }
}
```

3 Fichero `break.c`

La memoria total asignada a un proceso en el momento de su creación nunca cambia. Si durante el tiempo de ejecución del proceso éste necesitara más memoria, se tomará del espacio que hay entre los segmentos de datos y pila. Recordemos de la figura 1 que el segmento de pila crece desde posiciones altas a posiciones bajas de memoria y el segmento de datos en sentido contrario.

Cuando un proceso pide memoria al sistema, lo hace a través de la llamada `brk()`. Esta llamada al sistema se implementa mediante el envío de un mensaje al manejador de memoria; a su vez, el bucle de recepción de mensajes del MM, al recibir un mensaje `BRK`, llama a la función `do_brk`, definida el fichero `break.c`, para que procese la llamada. En la figura 6, el grafo de llamadas nos muestra cómo `brk()`, así como otras llamadas al sistema, provocan una serie de llamadas que desencadenan la ejecución de las funciones de éste módulo. También se explicitan los mensajes que `do_brk()` y `adjust()` envían al kernel para desempeñar su cometido.

Las tres funciones básicas que Minix encomienda al módulo `break.c` son:

- Impedir que exista colisión entre los segmentos de datos y pila
- Impedir que el puntero de pila sobrepase la base del segmento de pila
- Actualizar el tamaño del segmento de pila cuando éste cambie

El fichero `break.c` implementa estas tareas con las tres funciones que contiene `do_brk()`, `adjust()`, y `size_ok()`, pudiendo ser cualquiera de ellas llamada[5] desde otro módulo.

3.1 Función `do_brk`

Acepta como parámetros el proceso que generó el mensaje y la nueva dirección virtual del límite superior del segmento de datos de ese proceso. Estos parámetros no se le pasan directamente a la función sino que los extrae directamente de variables globales. Por una parte, toma de la variable global `mp` la dirección de la entrada del proceso en la tabla de procesos del sistema. Y en cuanto a la dirección de memoria virtual, la extrae de la variable `addr` (fichero `param.h`) que apunta al campo `m1_p1` del buffer para mensajes llamado `mm_in`.

La secuencia de operaciones que realiza do_brk es la siguiente:

1. Convierte la dirección virtual del segmento de datos a clicks
2. Comprueba que se trata de una dirección válida (que no sea menor que el comienzo del segmento de datos).
3. Obtiene el tamaño del nuevo bloque restando el límite del segmento y la dirección de comienzo.

[5] La figura 6 muestra como `do_brk()` puede ser llamada para satisfacer la llamada al sistema `brk()`, así como `adjust()` puede ser llamada por `dump_core()` o `sigproc()` y `size_ok()` por `read_header()`.

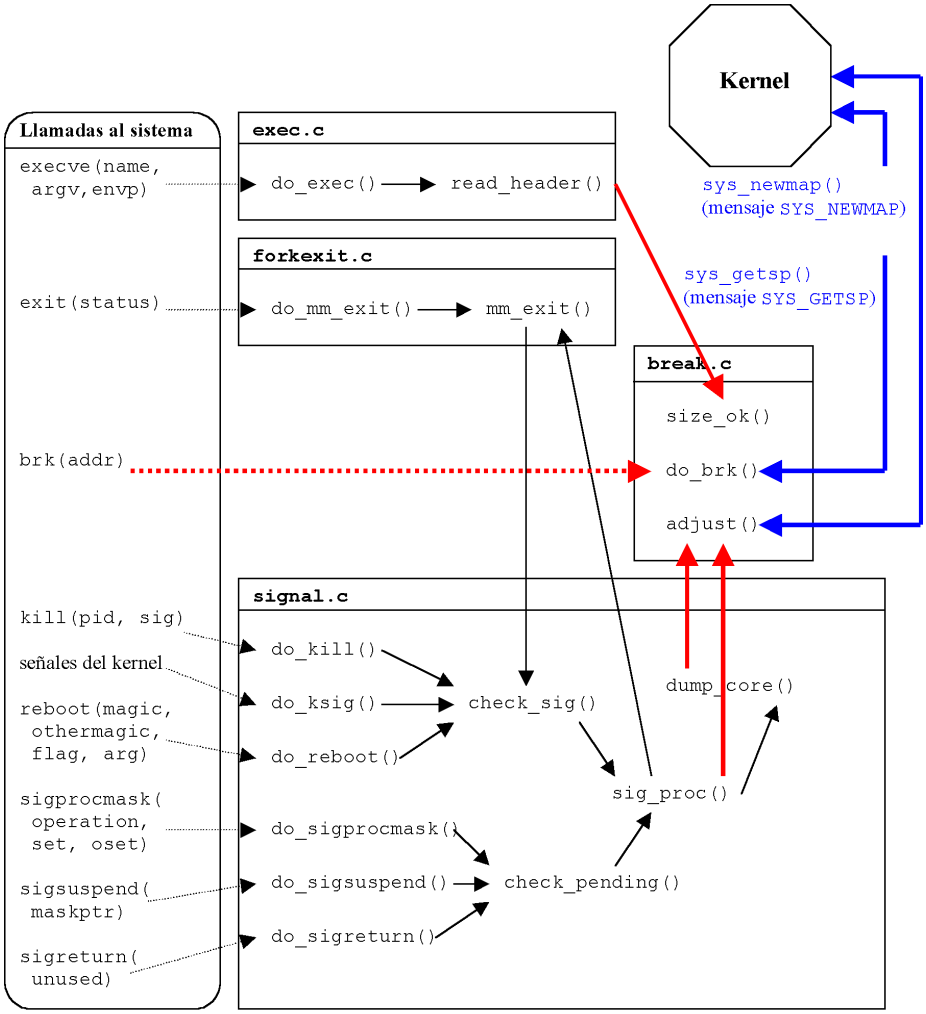

Figura 6: Relación de llamadas exteriores a `break.c`

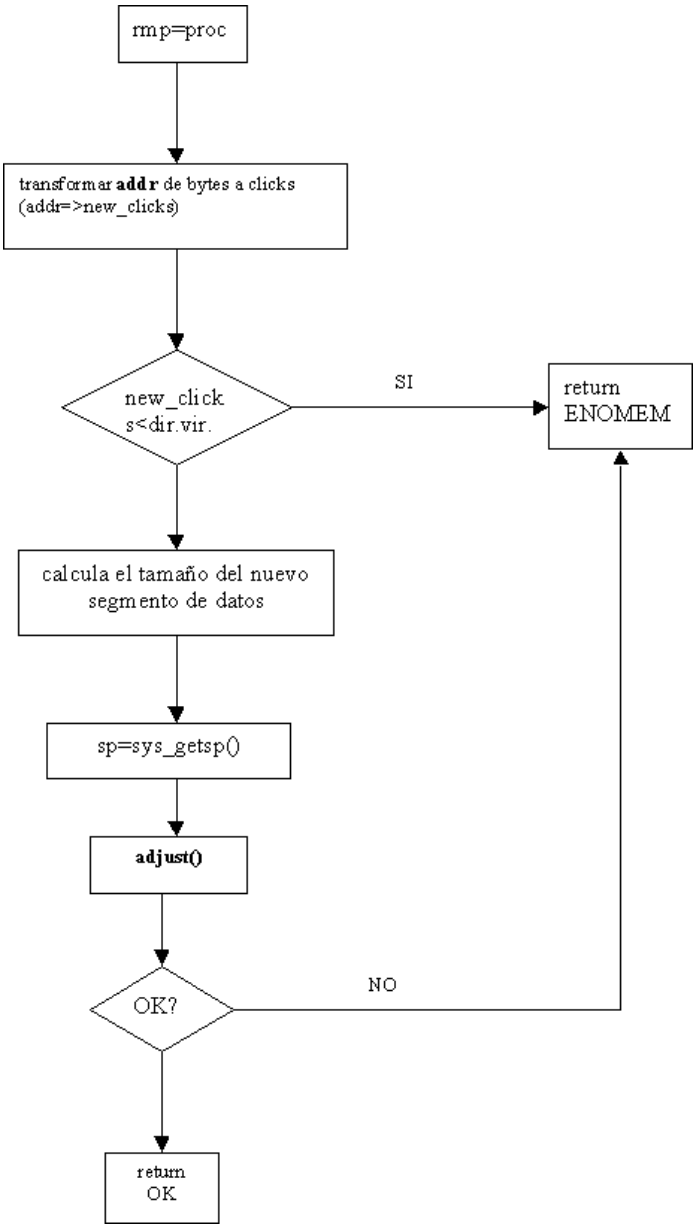

Figura 7: Organigrama de `do_brk()`

4. Invoca a la función `adjust` pasando como parámetros el tamaño del nuevo bloque, la dirección del puntero de pila y el puntero a la tabla de procesos.

5. Si la función `adjust` retorna con éxito se devuelve la nueva dirección de comienzo del segmento de datos, en caso contrario se devuelve -1.

3.2 Función `adjust`

La función `adjust(rmp, data_clicks, sp)` acepta como parámetros un puntero `rmp` a la tabla de procesos, que identifica al proceso cuya llamada al sistema ha provocado la llamada a la función[6], el puntero de pila `sp` del proceso, y el tamaño en clicks `data_clicks` su segmento de datos .

Su misión es ajustar el tamaño de los segmentos de un proceso al tamaño indicado en los parámetros de entrada.

La secuencia de ejecución de la función es la siguiente:

1. Averigua si el puntero de pila hace que la pila se desborde inferiormente. Para esto comprueba que el puntero no esté por encima del puntero de pila. En caso contrario devuelve un error.

2. Calcula el tamaño del bloque de memoria que hay entre el segmento de datos y la pila. Para hacer esto calcula el límite inferior de la pila (variable `lower`), después calcula el límite del segmento de datos (suma dada por el comienzo del segmento, su nuevo límite y un margen de seguridad `SAFETY_CLICKS`) y lo guarda en `gap_base`. Posteriormente comprueba si ambos valores se han cruzado y, si es así, termina y devuelve un error.

3. Si se llega a este punto se produce la actualización real de los tamaños de los segmentos de datos y pila. Para el segmento de datos sencillamente se asigna el valor del parámetro `data_clicks` al campo que indica el tamaño del segmento de datos. En el caso de la pila, además de actualizar el tamaño hay que mover el comienzo del segmento en la misma cantidad ya que la pila crece negativamente.

4. El último paso consiste en llamar a la función `size_ok`, que es la encargada de realizar las comprobaciones sobre el tamaño total de la memoria asignada al proceso y los solapamientos de segmentos. Si esta función devuelve error se deshacen los cambios realizados, en caso contrario se llama a `sys_newmap()` para informar al sistema de que el proceso tiene un nuevo mapa de memoria[7] y se devuelve `OK`.

3.3 Función `size_ok`

Esta función admite como parámetros el tipo de esquema de memoria (datos y código separados o no) y los comienzos y tamaños de los distintos segmentos.

[6] La llamada al sistema puede haber sidocualquiera de las que indica la figura 6

[7] Lo que se traduce en un mensaje SYS_NEWMAP (ver figura 6)

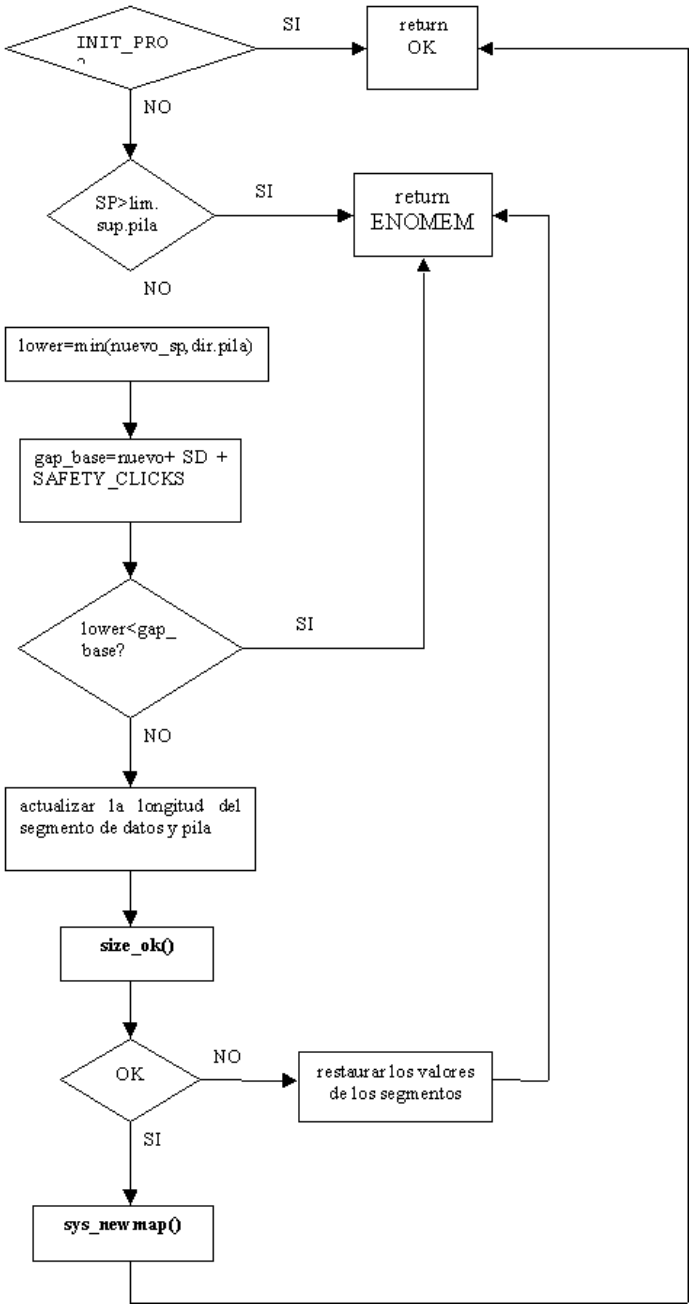

Figura 8: Organigrama de `adjust()`

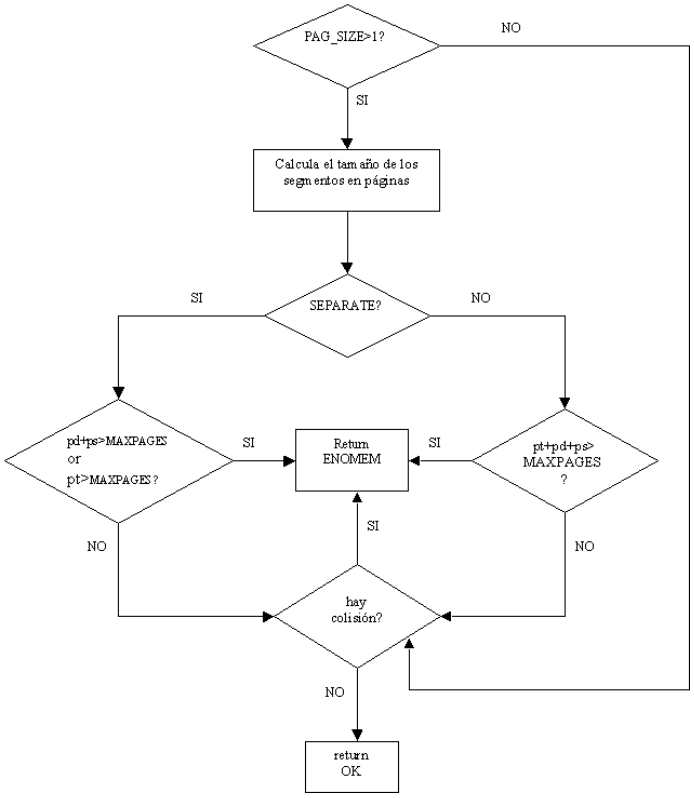

Figura 9: Organigrama de `size_ok`

La función `size_ok` es la encargada de realizar las últimas comprobaciones en cuanto a tamaño de segmentos de un programa. Comprueba que los tamaños de los segmentos quepan en el espacio de direcciones y que no haya colisión entre los segmentos (mientras en `adjust()` se miraba el límite inferior de la pila a partir de su puntero).

3.4 Código de `break.c`

```
#include "mm.h"
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define DATA_CHANGED    1 /* valor del flag cuando el segmen-
                           * to de datos ha cambiado */
#define STACK_CHANGED   2 /* valor del flag cuando el segmen-
                           * to de pila ha cambiado */
```

```
/*===========================================================*
 *                           do_brk                          *
 *===========================================================*/
PUBLIC int do_brk()
{
/* Lleva a cabo la llamada al sistema brk(addr).
 *
 * Existe la dificultad añadida de que en algunas máquinas
 * (como el 8088) el puntero de pila puede superar la base sin
 * que sea percibido.
 * El parámetro, 'addr', es la nueva dirección virtual en el
 * espacio D. */

  register struct mproc *rmp;
  int r;
  vir_bytes v, new_sp;
  vir_clicks new_clicks;

  rmp = mp;
  v = (vir_bytes) addr;
  new_clicks = (vir_clicks) ( ((long) v + CLICK_SIZE - 1) >>
          CLICK_SHIFT );
  if (new_clicks < rmp->mp_seg[D].mem_vir) {
        res_ptr = (char *) -1;
        return(ENOMEM);
  }
  new_clicks -= rmp->mp_seg[D].mem_vir;
  sys_getsp(who, &new_sp);      /* le pregunta al kernel el
                                 * valor actual de sp */
  r = adjust(rmp, new_clicks, new_sp);
  res_ptr = (r == OK ? addr : (char *) -1);
  return(r);                    /* devuelve la nueva dirección
                                 * o -1 */
}


/*===========================================================*
 *                           adjust                          *
 *===========================================================*/
PUBLIC int adjust(rmp, data_clicks, sp)
register struct mproc *rmp; /* ¿a quién se le va a ajustar la
                             * memoria?
vir_clicks data_clicks;     /* ¿cómo de grande va a ser el
                             * segmento de datos? */
vir_bytes sp;               /* nuevo valor de sp */
```

```
{
/* Comprueba si los segmentos de datos y pila pueden coexistir,
 * ajustándolos si es necesario. La memoria nunca se asigna o
 * libera; en vez de eso, se añade o se quita del espacio que
 * hay entre es segmento de pila y el de datos. Si el espacio
 * se hace negativo, no se puede ajustar la memoria y se de-
 * vuelve ENOMEM. */

  register struct mem_map *mem_sp, *mem_dp;
  vir_clicks sp_click, gap_base, lower, old_clicks;
  int changed, r, ft;
  long base_of_stack, delta; /* long para evitar ciertos pro-
                              * blemas */

  mem_dp = &rmp->mp_seg[D];  /* puntero al mapa del segmento de
                              * datos */
  mem_sp = &rmp->mp_seg[S];  /* puntero al mapa del segmento de
                              * pila */
  changed = 0;               /* indica si cambia cualquiera de
                              * los segmentos */

  if (mem_sp->mem_len == 0) return(OK); /* ni nos molestamos */

  /* Comprueba si el tamaño de la pila se ha hecho negativo
   * (por ej., cuando sp está demasiado cerca de 0xFFFF...) */
  base_of_stack = (long) mem_sp->mem_vir +
          (long) mem_sp->mem_len;
  sp_click = sp >> CLICK_SHIFT; /* clicks que indica sp */
  if (sp_click >= base_of_stack)
        return(ENOMEM); /* sp demasiado grande */

  /* Calcula el tamaño del hueco entre los segmentos de pila y
   * datos */
  delta = (long) mem_sp->mem_vir - (long) sp_click;
  lower = (delta > 0 ? sp_click : mem_sp->mem_vir);

  /* Añade un margend de seguridad por si la pila crece en el
   * futuro. Es imposible hacerlo 'bien' */
#define SAFETY_BYTES  (384 * sizeof(char *))
#define SAFETY_CLICKS ( (SAFETY_BYTES + CLICK_SIZE - 1) /
                        CLICK_SIZE )
  gap_base = mem_dp->mem_vir + data_clicks + SAFETY_CLICKS;
  if (lower < gap_base) return(ENOMEM); /* colisión entre pila
                                         * y datos */

  /* Actualiza la longitud de el segmento de datos (pero no la
```

```
   * dirección base) a causa de la llamada al sistema brk(). */
  old_clicks = mem_dp->mem_len;
  if (data_clicks != mem_dp->mem_len) {
        mem_dp->mem_len = data_clicks;
        changed |= DATA_CHANGED;
  }

  /* Actualiza el tamaño y el origen del segmento de pila ya
   * ha sido modificado. */
  if (delta > 0) {
        mem_sp->mem_vir -= delta;
        mem_sp->mem_phys -= delta;
        mem_sp->mem_len += delta;
        changed |= STACK_CHANGED;
  }

  /* ¿Los nuevos segmentos de datos y pila se ajustan al espa-
   * cio? */
  ft = (rmp->mp_flags & SEPARATE);
  r = size_ok( ft, rmp->mp_seg[T].mem_len,
          rmp->mp_seg[D].mem_len, rmp->mp_seg[S].mem_len,
          rmp->mp_seg[D].mem_vir, rmp->mp_seg[S].mem_vir );
  if (r == OK) {
        if (changed)
              sys_newmap((int)(rmp - mproc), rmp->mp_seg);
        return(OK);
  }

  /* Los nuevos tamaños no cuadran o requieren demasiadas
   * páginas por segmentos. Se dejan como estaban. */
  if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
  if (changed & STACK_CHANGED) {
        mem_sp->mem_vir += delta;
        mem_sp->mem_phys += delta;
        mem_sp->mem_len -= delta;
  }
  return(ENOMEM);
}


/*============================================================*
 *                          size_ok                           *
 *============================================================*/
PUBLIC int size_ok(file_type, tc, dc, sc, dvir, s_vir)
int file_type;            /* SEPARATE ó 0 */
vir_clicks tc;            /* tamaño del código en clicks */
```

```
vir_clicks dc;            /* tamaño de los datos en clicks */
vir_clicks sc;            /* tamaño de la pila en clicks */
vir_clicks dvir;          /* dirección virtual del comienzo del
                           * segmento de datos */
vir_clicks s_vir;         /* dirección virtual del comienzo del
                           * segmento de pila */
{
/* Comprueba si los tamaños son factibles y hay suficientes re-
 * gistros de segmento. En una máquina con 8 páginas de 8K, los
 * tamaños del código, datos y pila podrían ser (32K, 16K,
 * 16K), pero no (33K, 17K, 13K), incluso aunque el primero
 * fuera mayor (64K) que el segundo (63K). Esta comprobación es
 * necesaria incluso en el 8088, ya que los datos y la pila no
 * pueden sobrepasar los 4096 clicks. */

#if (CHIP == INTEL && _WORD_SIZE == 2)
  int pt, pd, ps;     /* tamaños de los segmentos en páginas */

  pt = ( (tc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
  pd = ( (dc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
  ps = ( (sc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;

  if (file_type == SEPARATE) {
        if (pt > MAX_PAGES || pd + ps > MAX_PAGES)
              return(ENOMEM);
  } else {
        if (pt + pd + ps > MAX_PAGES) return(ENOMEM);
  }
#endif

  if (dvir + dc > s_vir) return(ENOMEM);

  return(OK);
}
```

4 Cuestiones

- ¿ Para que se utiliza el vector hole ?
- ¿En que consiste la fusión de bloques de memoria ?
- ¿De donde saca Minix la memoria que proporciona a un proceso cuando éste ejecuta un malloc?
- ¿Qué ocurre cuando el puntero la pila y el límite superior del segmento de datos se cruzan?