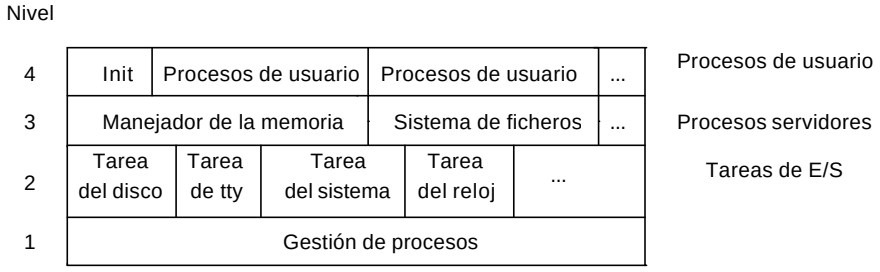
Llamada al sistema EXEC

Introducción

Recordemos la estructura del MINIX en sus cuatro niveles:



En el nivel 3 encontramos el manejador de memoria (MM) que proporciona las llamadas del sistema de MINIX que implica el manejo de memoria, tales como *fork*, *exec* y *brk*.

La llamada *fork* es la única forma de crear un nuevo proceso. Se crea un duplicado exacto del proceso original. Después del *fork*, el proceso original y la copia (padre e hijo) siguen caminos independientes. Todas las variables tienen el mismo valor cuando se termina de hacer el *fork*, pero los sucesivos cambios en los datos de uno de ellos no afectan a los del otro. El código, como no se modifica, es compartido entre el hijo y el padre. La llamada *exec* permite al proceso hijo ejecutar el comando especificado.

En muchos casos, después del *fork*, el proceso hijo necesitará ejecutar diferente código que el padre. Consideremos el caso del *shell*: se lee un comando del terminal, haciendo uso del *fork* se crea un proceso hijo y se espera por el hijo para ejecutar el comando y cuando el hijo termina se lee el siguiente comando. Para esperar que el hijo termine, el padre ejecuta *waitpid* (otra llamada del sistema), que indica cuando un proceso hijo ha terminado. La ejecución del comando por parte del hijo se hace usando *exec*. En las siguientes líneas lo vemos de una forma muy simplificada:

```
while ( TRUE ){
        read_command(comando,parametros); /* lee entrada del termianl */
        if ( fork() != 0 ){                     /* fork del proceso padre */
                /* codigo del padre */
                waitpid(-1,&status,0);  /* espera por la finalizacion del hijo */
        }
        else{
                /* codigo del hijo */
                execve( comando,parametros,0);   /* ejecuta el comando */
        }
}
```

En el caso más general, *exec* tiene tres parámetros:

- el nombre del fichero ha ser ejecutado.
- un puntero al vector de argumentos
- un puntero al vector de ambiente.

Las librerías contienen varias versiones del *exec* (*execl*, *execv*, *execle* y *execve*) que permiten omitir o especificar los parámetros de forma distinta. En este trabajo usaremos el nombre de *exec* para referirnos a cualquiera de ellas.

La llamada del sistema EXEC

Cuando un comando es escrito en el terminal , el *shell* realiza un *fork* y se crea un nuevo proceso hijo, que ejecuta el comando solicitado. Se podía haber hecho una sola llamada al sistema de forma más simple que hiciera tanto *fork* como *exec* en una sola función, pero se ha hecho dos llamadas distintas por una buena razón : **hacer fácil la implementación del rediccionamiento de la entrada/salida**. Cuando se realiza un *fork*, si la entrada estandar es redirecionada, el proceso hijo cierra la entrada estandar y abre una nueva entrada estandar antes de ejecutar el comando. De esta forma el nuevo proceso que comienza hereda la entrada estandar redireccionada. La salida estandar es manipulada de la misma manera.

La llamada al sistema *exec* es la más compleja de MINIX. Debe reemplazar la imagen actual en memoria por una nueva, incluyendo establecer una nueva pila. Esto se hace en los siguientes pasos :

1) Chequear permisos - ¿ es el fichero ejecutable ?
2) Leer la cabecera para obtener los tamaños de los distintos segmentos.
3) Obtener los argumentos y el ambiente del proceso que hace la llamada al sistema.
4) Asignar la nueva memoria y liberar la memoria antigua no necesitada.
5) Copiar la pila a la nueva imagen de memoria.
6) Copiar el segmento de datos (y posible código) a la nueva imagen de memoria.
7) Verificar y asignar los bits setuid, setgid del nuevo proceso.
8) Rellenar la tabla de procesos del MM con los datos del nuevo proceso.
9) Indicar al Kernel que ahora el proceso es ejecutable.

Cada uno de estos pasos, consta a su vez, de una serie de pasos menores algunos de los cuales puede fallar. El orden en que se realizan los pasos ha sido elegido de forma cuidadosa para evitar eliminar la copia de memoria antigua hasta no estar seguro de que el *exec* ha tenido éxito. Normalmente *exec* no retorna ningún valor, pero si falla, el proceso que hizo la llamada debe recuperar el control con una indicación de error.

Hay unos pasos que merecen algunos comentarios añadidos. Primero, hay que determinar cuanta memoria se necesita, para ello hemos de comprobar si el segmento de código puede ser compartido. Se busca en la tabla de huecos de memoria para verificar si hay suficiente memoria física antes de liberar la memoria antigua. Si la memoria antigua fuese liberada primero y no hubiera suficiente memoria sería muy difícil recuperar la antigua imagen de nuevo.

Un aspecto importante es la manera en que se forma la pila actual. La llamada a la bibilioteca que normalmente se utiliza para invocar *exec* con argumentos y un ambiente es :

execve (nombre, argv, envp);

Donde **nombre** es un apuntador al nombre del fichero que se ejecutará., **argv** es un puntero a un vector de punteros, donde cada uno apunta a un argumento, y **envp** es un puntero a un vector de punteros, cada uno de los cuales apunta a una cadena de ambiente.

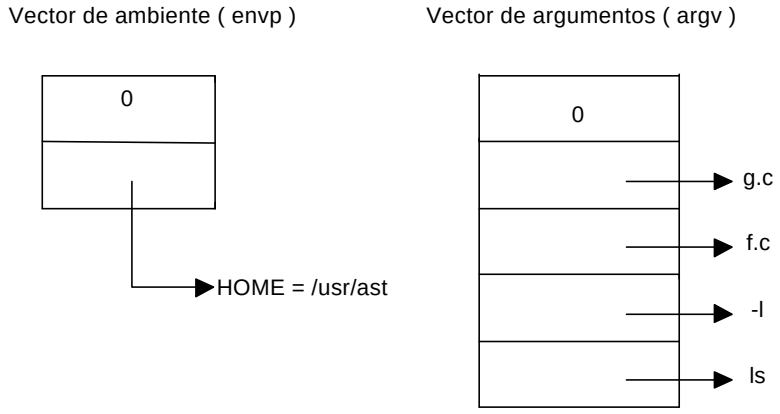
Se podía haber implementado *exec* de forma que los parámetros se pasasen al manejador de memoria mediante una política de mensajes, trayendo cada argumento y cada cadena a la vez. Hacer esto requiere al menos de uno o más mensajes para la tarea del sistema por cada argumento o cadena, ya que el manejador de memoria no tiene manera de saber cuandos mensajes son necesarios.

Para evitar tantos mensajes, se elige una estrategia diferente. El procedimiento de biblioteca *execve* construye toda la pila inicial dentro de su propio espacio de memoria y pasa su dirección de base y tamaño al manejador de memoria. La construcción de la nueva pila dentro del espacio del usuario es altamente efectiva, ya que las referencias que se hacen a los argumentos y cadenas son tan solo referencias a memorias locales.

Para aclarar como funciona esta estrategia, consideremos un ejemplo. Cuando un usuario teclea

ls -l f.c g.c

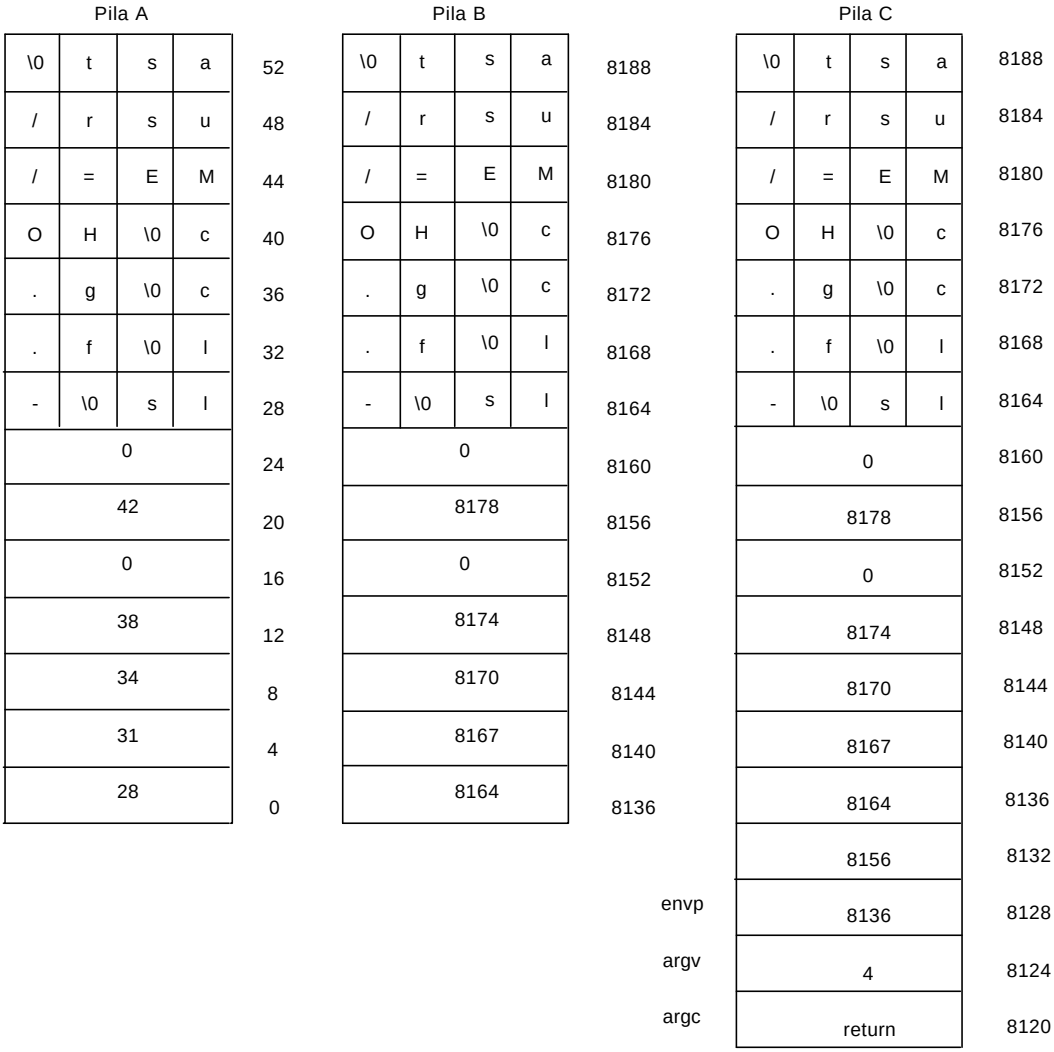
al shell, éste lo intepreta y hace la llamada **execve ("/bin/ls", argv, envp);** al procedimiento de bibilioteca. El contenido de los dos vectores de apuntadores se muestran en la siguiente figura :



El procedimiento *execve*, contenido en el espacio de dirección del shell, construye ahora la pila inicial como se muestra en la pila A

Esta pila se copia intacta a la memoria del manejador durante el procesamiento de la llamada *exec*. Vemos la pila después de la relocación en la memoria del manejador de memoria en la pila B.

Cuando la pila se copia finalmente en el proceso de usuario, ésta no se colocará en la dirección virtual 0. En su lugar, se colocará al final de la asignación de la memoria, como lo determina el tamaño de la memoria total que está en el encabezado del fichero ejecutable. Como ejemplo, asumamos que el tamaño total es de 8192 bytes, así que el último byte disponible para el programa está en la dirección 8191. Corresponde al manejador de memoria actualizar los punteros dentro de la pila para posicionarlos en las nuevas direcciones, como se observa en la pila B.



Pila A : la pila construida por *execve*. Pila B : la pila redireccionada por el manejador de memoria. Pila C : la pila como aparece al main a cuando empieza la ejecución.

Cuando la llamada *exec* completa y el programa comienza a ejecutarse, la pila en realidad se verá como la pila B, con el puntero de pila tendrá el valor 8136. Sin embargo, hemos de hablar de otro problema. El programa principal del fichero a ejecutar se declarará probablemente de la siguiente forma :

main (argc, argv, envp)

EL compilador de C tratará a main como otra función. No sabe que main es especial, de modo que compilará el código para acceder a los tres parámetros indicados en la llamada. Este acceso se realizará de forma estandar, es decir, el último parámetro el primero. Un

entero (número de argumentos), dos punteros (argv y envp) y la dirección de retorno ocuparán las cuatro últimas palabras de la pila, tal como se observa en la pila C.

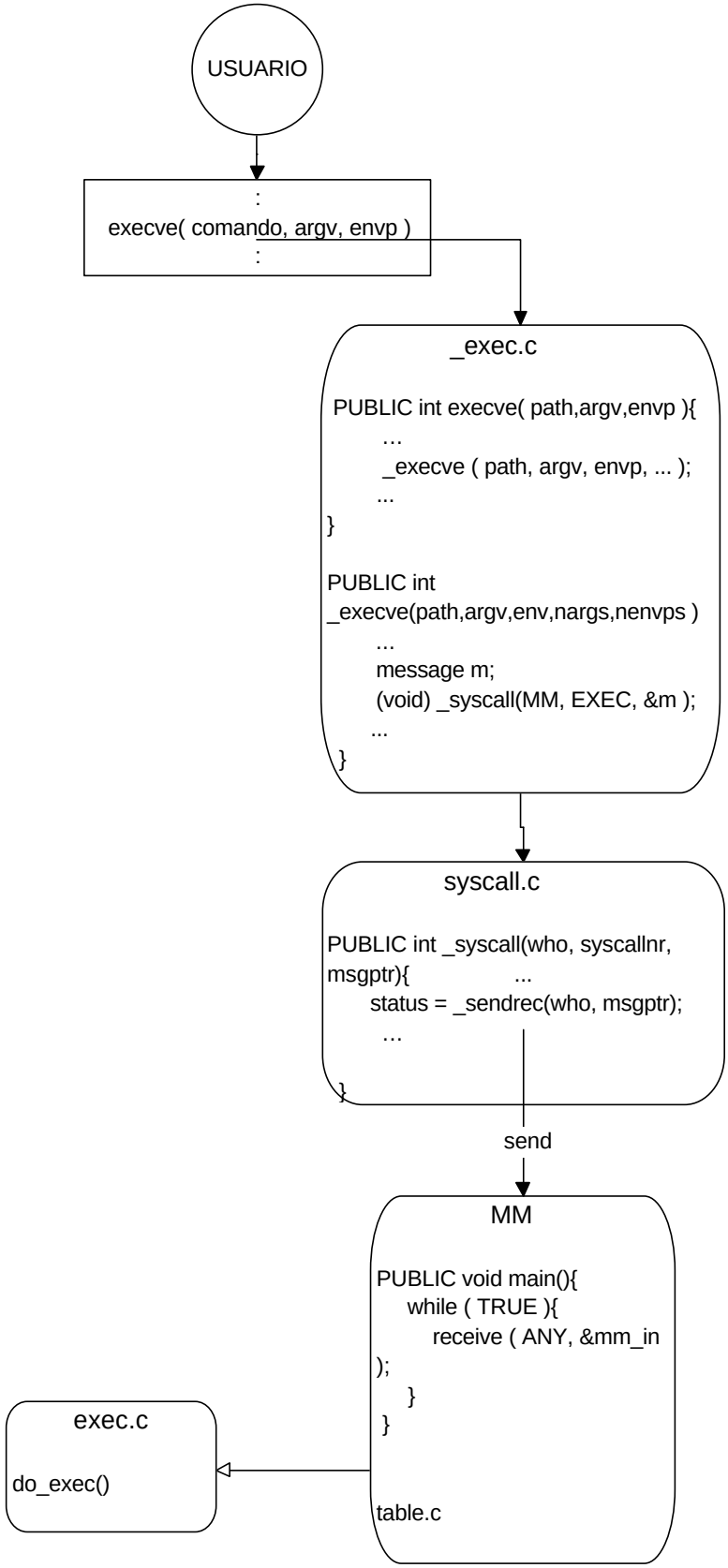
La solución es que los programas no comiencen con main. En su lugar una pequeña rutina en lenguaje ensamblador, denominada **C run-time start-off procedure** (crtso - procedimiento de arranque en el momento de la ejecución de C), linkada en la dirección de código 0, asi que toma el control primero. Su labor consiste en meter tres palabras más en la pila (los dos punteros y el entero) y después llamar a main usando la instrucción de llamada estandar. Esto da origen a la formación de la pila C en el momento en que main empieza la ejecución, por lo tanto se "engaña" a main creyéndose que fue llamada de la forma usual.

Si el programador no efectúa un exit al final de main el control regresará a la rutina crtso cuando el main termine y ésta llamará a exit.

Liberar la memoria anterior y asignar la nueva.

Podría ocurrir que no hubiese memoria libre suficiente. La verificación se realiza al buscar en la lista de huecos antes de liberar la memoria ocupada por el proceso que ejecuta la llamada . Esto es así, por que si se eliminase antes la memoria ocupada por el proceso padre, y la ejcución *exec* fracasa, no habría forma de volver atrás. Sin embargo esta prueba es extremadamente estricta. A veces rechaza llamadas a EXEC por falta de memoria, que de hecho podrían tener exito. Una implementación más compleja podría manejar esta situación un poco mejor.

Llamada exec : desde un proceso usuario hasta


USUARIO
:
execve(comando, argv, envp)
:
_exec.c
PUBLIC int execve(path,argv,envp){
…
_execve (path, argv, envp, ...);
...
}
PUBLIC int
_execve(path,argv,env,nargs,nenvps)
...
message m;
(void) _syscall(MM, EXEC, &m);
...
}
syscall.c
PUBLIC int _syscall(who, syscallnr,
msgptr){ ...
status = _sendrec(who, msgptr);
…
}
send
MM
PUBLIC void main(){
while (TRUE){
receive (ANY, &mm_in
);
}
}
table.c
exec.c
do_exec()

Implementación del exec

El código para el *exec* sigue los 9 pasos anteriormente comentados. Están en el procedimiento **do_exec**. Después de hacer unas verificaciones, el manejador de memoria trae el nombre del fichero ha ser ejecutado desde espacio de usuario. Se envía un mensaje al sistema de ficheros para cambiar al directorio de usuario, y así el path traído se interpretará relativo al directorio de trabajo del usuario y no al manejador de memoria.

Si el fichero existe y es ejecutable, el manejador de memoria lee la cabecera para extraer los tamaños de los segmentos. Entonces, trae la pila del espacio de usuario, verifica si el nuevo proceso puede compartir código con otro proceso ya en ejecución, localiza memoria para la nueva imagen, actualiza los punteros (ver diferencia entre la pila B y C), y lee el segmento de código si es necesario y el segmento de datos. Finalmente, se procesa los bits setuid y setgid, se actualiza la entrada en la tabla de procesos y se le dice al Kernel que se ha terminado y el proceso puede ser encolado otra vez.

Aunque el control de todos los pasos están en ***do_exec***, muchas acciones se apoyan de otras funciones dentro del *exec.c*. Así, ***read_header*** no sólo lee la cabecera y retorna los tamaños de los segmentos, sino que también verifica que el fichero es un ejecutable para la misma CPU en que se compiló el sistema operativo. También comprueba que todos los segmentos estén edecuadamente situados en el espacio de direcciones virtuales.

El procedimiento ***new_mem*** se encarga de comprobar si hay suficiente memoria disponible para la nueva imagen de memoria. Busca un hueco suficientemente grande en la tabla de huecos de memoria para los datos y la pila, si el código es compartido; en caso contrario busca un hueco para los datos, la pila y el código. Si se encuentra suficiente espacio de memoria, la memoria antigua es eliminada y se proporciona la nueva memoria. En el caso de no disponer de memoria el *exec* causa un error. Después de localizar la nueva memoria, ***new_mem*** actualiza el mapa de memoria (en *mp_seg*) e informa al Kernel llamando al procediento de librería *sys_newmap*.

El siguiente procedimiento es ***patch_ptr***, que hace el trabajo de redireccionar los punteros de la pila B a la forma de la pila C. La manera de hacerlo es muy simple : examina la pila para encontrar todos los punteros y le suma la dirección base a cada uno.

El procedimiento ***load_seg***, se llama una o dos veces en *exec* para guardar el segmento de datos y de código. Más que leer el fichero bloque a bloque y copiarlos a los bloques del usuario, se realiza un truco para permitir al sistema de ficheros guardar directamente todo el segmento al espacio de usuario. En efecto, la llamada es decodificada por el sitema de ficheros de una forma un tanto especial para permitir leer el segmento entero por el propio proceso de usuario. La carga hecha de esta manera es apreciablemente más rápida.

El último procedimiento en *exec.c* es ***find_share***. Se encarga de buscar un proceso que comparta el código. Lo hace comparando los i-nodo, los dispositivos y modificaciones del fichero a ser ejecutado con los procesos ya existentes.

Estructura de datos :

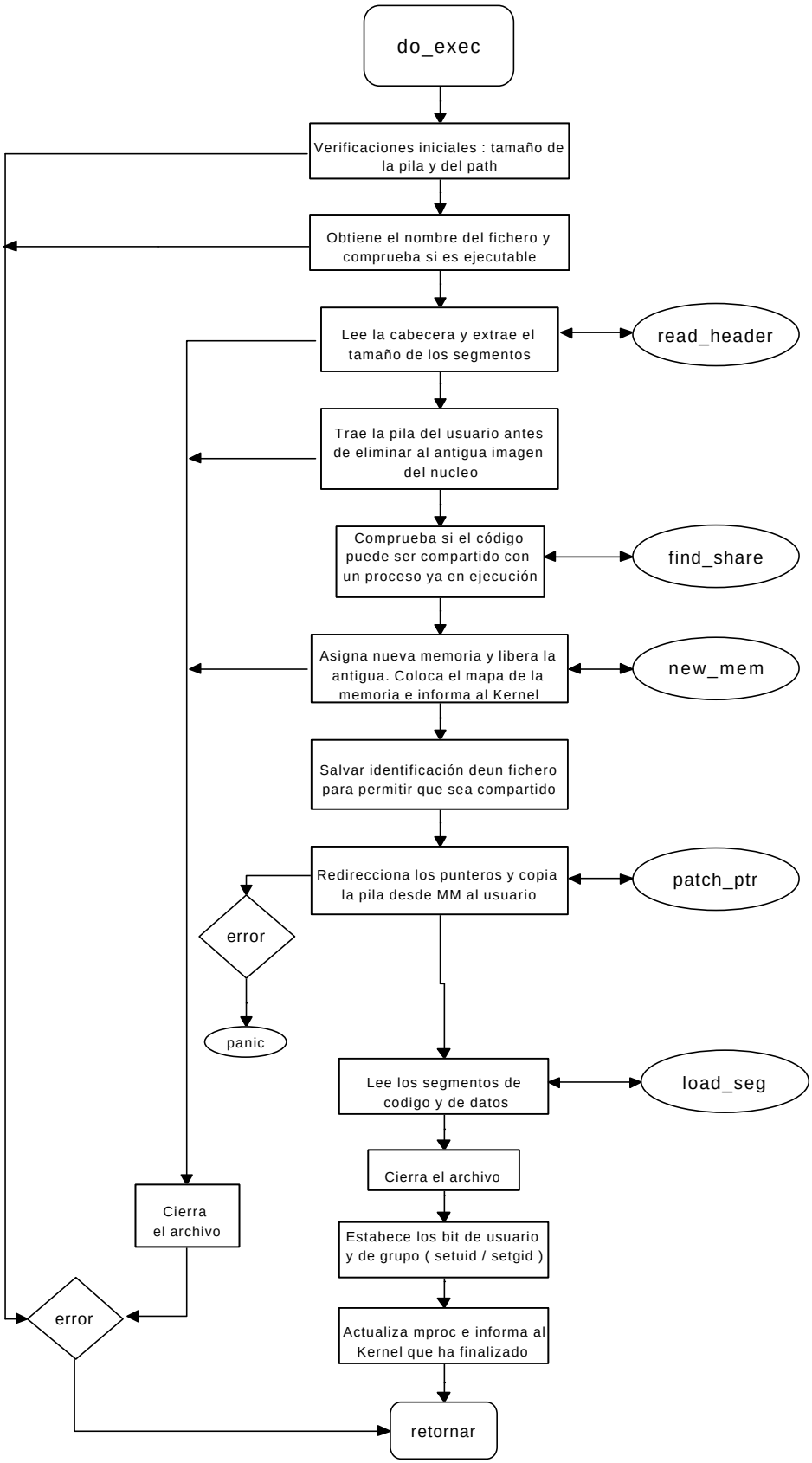
estructura **MPROC** : en mproc.h, contiene toda la información que usa el manejador de procesos para cada uno de ellos. Entre otras cosas se define los segmentos de datos, codigo y de pila, valores uid (usuario) y gid (grupo) y varios flags.

Fichero exec.c

```
/* Este fichero maneja la llamada EXECl.  Sigue los siguientes pasos:
 *    - si los permisos permiten al fichero ser ejecutado
 *    - leer la cabecera y extraer los tamaños
 *    - traer los argumentos iniciales y el ambiente del espacio de usuario
 *    - localizar memoria para el nuevo proceso
 *    - copiar la pila inicial desde el MM al nuevo proceso
 *    - leer en el segmento de datos y codigo y copiar al proceso
 *    - actualizar los bits de setuid y setgid
 *    - Actualizar la tabla 'mproc'
 *    - informa al Kernel sobre el  EXEC
 *    - salve el desplazamiento para el argv inicial  (para ps)
 *
 * Los puntos de entrada son:
 *   do_exec:     realiza la llamada EXEC
 *   find_share: encuentra un proceso cuyo segmento de  codigo puede ser compartido
 */

#include "mm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <a.out.h>
#include <signal.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

FORWARD _PROTOTYPE( void load_seg, (int fd, int seg, vir_bytes seg_bytes) );
FORWARD _PROTOTYPE( int new_mem, (struct mproc *sh_mp, vir_bytes text_bytes,
                vir_bytes data_bytes, vir_bytes bss_bytes,
                vir_bytes stk_bytes, phys_bytes tot_bytes)               );
FORWARD _PROTOTYPE( void patch_ptr, (char stack [ARG_MAX ], vir_bytes base) );
FORWARD _PROTOTYPE( int read_header, (int fd, int *ft, vir_bytes *text_bytes,
                vir_bytes *data_bytes, vir_bytes *bss_bytes,
                phys_bytes *tot_bytes, long *sym_bytes, vir_clicks sc,
                vir_bytes *pc)                                          );
```

Procedimiento *do_exec*

Este es el punto de entrada a la llamada del sistema *exec*. Su misión es reemplazar el mapa de memoria actual, incluyendo la creación de una nueva pila e informar al Kernel de que un nuevo proceso es ejecutable.


do_exec
Verificaciones iniciales : tamaño de la pila y del path
Obtiene el nombre del fichero y comprueba si es ejecutable
Lee la cabecera y extrae el tamaño de los segmentos
read_header
Trae la pila del usuario antes de eliminar al antigua imagen del nucleo
Comprueba si el código puede ser compartido con un proceso ya en ejecución
find_share
Asigna nueva memoria y libera la antigua. Coloca el mapa de la memoria e informa al Kernel
new_mem
Salvar identificación deun fichero para permitir que sea compartido
Redirecciona los punteros y copia la pila desde MM al usuario
patch_ptr
error
panic
Lee los segmentos de codigo y de datos
load_seg
Cierra el archivo
Cierra el archivo
Estabece los bit de usuario y de grupo (setuid / setgid)
error
Actualiza mproc e informa al Kernel que ha finalizado
retornar

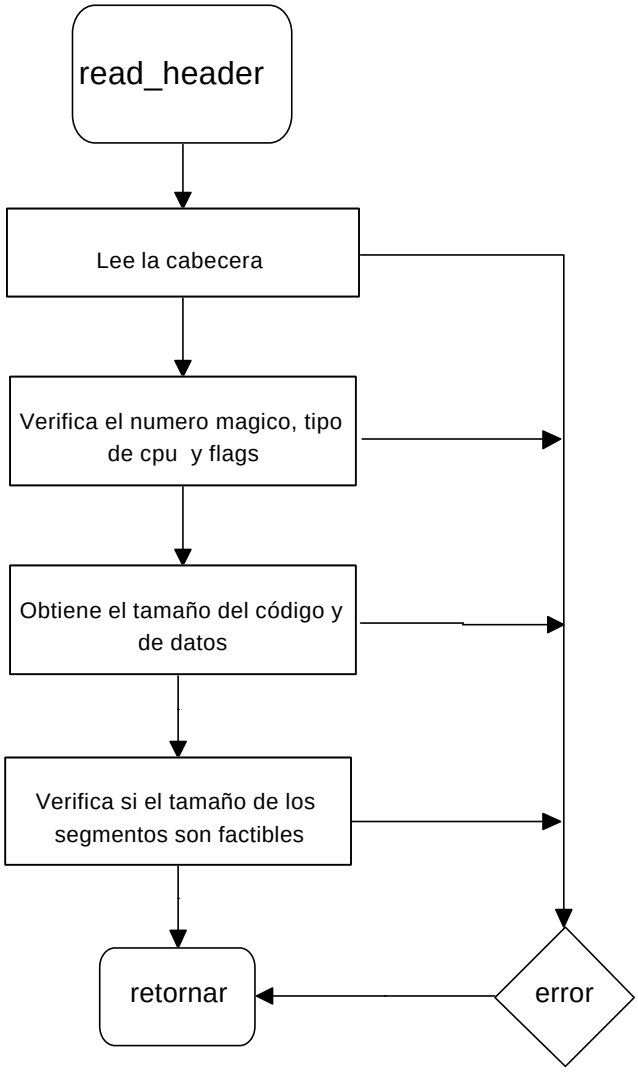
```
/*============== ========================================*
 *                                do_exec                                 *
 *========================================================*/
PUBLIC int do_exec()
{
/* Realiza la llamada execve(name, argv, envp). La biblioteca del usuario construye
 * una imagen completa de la pila, incluyendo punteros, argumentos , ambiente , etc. La
 * pila se copia a un buffer dentro de MM y entonces a la nueva imagen del núcleo.
 */
 register struct mproc *rmp;
 struct mproc *sh_mp;
 int m, r, fd, ft, sn;
 static char mbuf[ARG_MAX];         /* buffer para la pila e inicializaciones  */
 static char name_buf[PATH_MAX]; /* el nombre del fichero a ejecutar */
 char *new_sp, *basename;
 vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
 phys_bytes tot_bytes;                  /* espacio total del programa incluye el gap */
 long sym_bytes;
 vir_clicks sc;
 struct stat s_buf;
 vir_bytes pc;

 /* Verificaciones de validez:
        - el tamaño de la pila demasiado grande
        - el path del archivo demasiado grande
*/
 rmp = mp;
 stk_bytes = (vir_bytes) stack_bytes;
 if (stk_bytes > ARG_MAX) return(ENOMEM);      /* pila demasiado grande */
 if (exec_len <= 0 || exec_len > PATH_MAX) return(EINVAL);

 /* Onbtiene el fichero y comprueba que es ejecutable. */
 src = (vir_bytes) exec_name;
 dst = (vir_bytes) name_buf;
 r = sys_copy(who, D, (phys_bytes) src,
                MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes) exec_len);
 if (r != OK) return(r);           /* el nombre del fichero no en segmento de datos de usuario */
 tell_fs(CHDIR, who, FALSE, 0);     /* cambiar al ambiente del usuario. */
 fd = allowed(name_buf, &s_buf, X_BIT);    /* ¿ es el fichero ejecutable ? */
 if (fd < 0) return(fd);  /* fichero no ejecutable */

 /* Lee la cabecera  y extrae el tamaño de los segmentos. */
 sc = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
 m = read_header(fd, &ft, &text_bytes, &data_bytes, &bss_bytes,
                                        &tot_bytes, &sym_bytes, sc, &pc);
 if (m < 0) {
        close(fd);                  /* algo erroneo en la cabecera */
        return(ENOEXEC);
 }
```

```
  /* Trae la pila del espacio de usuario antes de destruir la antigua imagen  */
  src = (vir_bytes) stack_ptr;
  dst = (vir_bytes) mbuf;
  r = sys_copy(who, D, (phys_bytes) src,
                          MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes)stk_bytes);
  if (r != OK) {
          close(fd);              /* no puedo traer la pila (ej. direccion virtual erronea) */
          return(EACCES);
  }

  /* ¿ Puede el código del proceso ser compartido con uno ya en ejecución ? */
  sh_mp = find_share(rmp, s_buf.st_ino, s_buf.st_dev, s_buf.st_ctime);

  /* Localiza nueva memoria y libera la antigua memoria. Coloca el mapa e informa al
       Kernel. */
  r = new_mem(sh_mp, text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes);
  if (r != OK) {
          close(fd);              /* nucleo insuficiente o programa demasiado grande */
          return(r);
  }

  /* Salva la identificación del fichero para permitir que sea compartido */
  rmp->mp_ino = s_buf.st_ino;
  rmp->mp_dev = s_buf.st_dev;
  rmp->mp_ctime = s_buf.st_ctime;

  /* Trae la pila y la copia desde el MM a la nueva imagen del nucleo*/
  vsp = (vir_bytes) rmp->mp_seg[S].mem_vir << CLICK_SHIFT;
  vsp += (vir_bytes) rmp->mp_seg[S].mem_len << CLICK_SHIFT;
  vsp -= stk_bytes;
  patch_ptr(mbuf, vsp);
  src = (vir_bytes) mbuf;
  r = sys_copy(MM_PROC_NR, D, (phys_bytes) src,
                          who, D, (phys_bytes) vsp, (phys_bytes)stk_bytes);
  if (r != OK) panic("do_exec stack copy err", NO_NUM);

  /* Lee el código y el segmento de datos */
  if (sh_mp != NULL) {
          lseek(fd, (off_t) text_bytes, SEEK_CUR);  /* shared: skip text */
  } else {
          load_seg(fd, T, text_bytes);
  }
  load_seg(fd, D, data_bytes);

#if (SHADOWING == 1)
  if (lseek(fd, (off_t)sym_bytes, SEEK_CUR) == (off_t) -1) ;          /* error */
  if (relocate(fd, (unsigned char *)mbuf) < 0)          ;             /* error */
  pc += (vir_bytes) rp->mp_seg[T].mem_vir << CLICK_SHIFT;
#endif
```

```
 close(fd);                     /* ya no se necesita el fichero exec */

 /* Ajusta los bits setuid/setgid: si esta en modo traza, resetea dicho modo y asigna los
     identificadores de grupo y de usuario ( tell_fs ) */
 if ((rmp->mp_flags & TRACED) == 0) { /* suprime si modo traza */
        if (s_buf.st_mode & I_SET_UID_BIT) {
                rmp->mp_effuid = s_buf.st_uid;
                tell_fs(SETUID,who, (int)rmp->mp_realuid, (int)rmp->mp_effuid);
        }
        if (s_buf.st_mode & I_SET_GID_BIT) {
                rmp->mp_effgid = s_buf.st_gid;
                tell_fs(SETGID,who, (int)rmp->mp_realgid, (int)rmp->mp_effgid);
        }
 }

 /* Salva el desplazamiento del argumuento inicial, argc*/
 rmp->mp_procargs = vsp;

 /* Actualiza campos de  'mproc' , informa al  kernel que el exec se ha ejecutado,  resetea las
     señales cpaturadas */
 for (sn = 1; sn <= _NSIG; sn++) {
        if (sigismember(&rmp->mp_catch, sn)) {
                sigdelset(&rmp->mp_catch, sn);
                rmp->mp_sigact[sn].sa_handler = SIG_DFL;
                sigemptyset(&rmp->mp_sigact[sn].sa_mask);
        }
 }

 rmp->mp_flags &= ~SEPARATE;    /* turn off SEPARATE bit */
 rmp->mp_flags |= ft;           /* turn it on for separate I & D files */
 new_sp = (char *) vsp;

 tell_fs(EXEC, who, 0, 0);      /* permite al FS manejar ficheros FD_CLOEXEC  */

 /* System will save command line for debugging, ps(1) output, etc. */
 basename = strrchr(name_buf, '/');
 if (basename == NULL) basename = name_buf; else basename++;
 sys_exec(who, new_sp, rmp->mp_flags & TRACED, basename, pc);
 return(OK);
}
```

Procedimiento *read_header*

Read_header no sólo lee la cabecera y retorna los tamaños de los segmentos, sino que también verifica que el fichero es un ejecutable para la misma CPU en que se compiló el sistema operativo. También comprueba que todos los segmentos estén adecuadamente situados en el espacio de direcciones virtuales.



```
/*===========================================*
 *                read_header                *
 *===========================================*/
PRIVATE int read_header(fd, ft, text_bytes, data_bytes, bss_bytes,
                                        tot_bytes, sym_bytes, sc, pc)

int fd;                 /* descriptor del aechivo exec */
int *ft;                /* lugar para devolver numero ft  */
vir_bytes *text_bytes;  /* lugar para devolver tamaño de codigo */
vir_bytes *data_bytes;          /* lugar para devolver tamaño de datos inicilizados */
vir_bytes *bss_bytes;   /* lugar para devolver tamaño de bss  */
phys_bytes *tot_bytes;  /* lugar para devolver tamaño total  */
long *sym_bytes;        /* lugar para devolver tamaño de la tabla de simbolos */
vir_clicks sc;          /* tamaño de la pila en clicks */
vir_bytes *pc;          /* punto de entrada al programa ( PC inicial) */

{
/* Lee la canecera y extrae el texto, datos, bss y su tamaño total */

  int m, ct;
  vir_clicks tc, dc, s_vir, dvir;
  phys_clicks totc;
  struct exec hdr;              /* cabecera de a.out leida aqui */

  /* Lee la cabecera y verifica los numeros magicos.  La cabecera estandar de MINIX
   * se define en <a.out.h>.  Cosiste en 8 caracteres seguidos por 6 longs.
   * Seguidos por 4 longs que no se usan aqui
   *      Byte 0: numero magico 0x01
   *      Byte 1: numero magico 0x03
   *      Byte 2: normal = 0x10 (no verificado, 0 is OK),  I/D separados = 0x20
   *      Byte 3:  tipo CPU , Intel 16 bit = 0x04, Intel 32 bit = 0x10,
   *          Motorola = 0x0B, Sun SPARC = 0x17
   *      Byte 4: longitud de la cabecera = 0x20
   *      Bytes 5-7 no son usados
   *
   *      Ahora vienen  6 longs
   *      Bytes  8-11: tamaño del segmento de texto en bytes
   *      Bytes 12-15: tamaño de los datos inicializdos en byte
   *      Bytes 16-19: tamaño de bss en bytes
   *      Bytes 20-23: punto de entrada al programa
   *      Bytes 24-27: memoria total del programa ( datos + stack )
   *      Bytes 28-31: tamaño de la tabla de símbolos en bytes.
   *  Los longs se representan en una máquina dependiendo del orden,
   * little-endian en el 8088, big-endian en el 68000.
   * Siguiendo directamente a la cabecera esta el segmento de código y de datos, y la
   * tabla de símbolos ( si hay ). Los tamaños estan en la cabecera. Solo los
   * los segmentos de datos y codigo se copian a memoria por el exec. La cabecera
   * solo se usa aqui. La tabla de simbolos es solo de utilidad para el debugger y
   * se ignora aqui
```

```
   */

  if (read(fd, (char *) &hdr, A_MINHDR) != A_MINHDR) return(ENOEXEC);

  /* Verifica  numero magico, tipo cpu y flags. */
  if (BADMAG(hdr)) return(ENOEXEC);
#if (CHIP == INTEL && _WORD_SIZE == 2)
  if (hdr.a_cpu != A_I8086) return(ENOEXEC);
#endif
#if (CHIP == INTEL && _WORD_SIZE == 4)
  if (hdr.a_cpu != A_I80386) return(ENOEXEC);
#endif
  if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);

  *ft = ( (hdr.a_flags & A_SEP) ? SEPARATE : 0);    /*  I & D separadas o no */

  /* Obtiene el tamaño de datos y codigo */
  *text_bytes = (vir_bytes) hdr.a_text;           /* tamaño de codigo en  bytes */
  *data_bytes = (vir_bytes) hdr.a_data;           /* tamaño de datos en bytes */
  *bss_bytes  = (vir_bytes) hdr.a_bss; /* tamaño de bss en bytes */
  *tot_bytes  = hdr.a_total;                /*  bytes total */
  *sym_bytes  = hdr.a_syms;               /* tamaño de tabla de simbolos en bytes */
  if (*tot_bytes == 0) return(ENOEXEC);

  if (*ft != SEPARATE) {

#if (SHADOWING == 0)
        /* Si el espacio de I & D no esta separadpo, todo se consideran datos. Text=0*/
        *data_bytes += *text_bytes;
        *text_bytes = 0;
#else
        /*
         * Treating text as data increases the shadowing overhead.
         * Bajo la suposición de que los programas  NO MODIFICAN CODIGO
         * podemos compartir el codigo entre los procesos padre e hijo.
         * Es similar a la opcion UNIX V7 -n de ld(1).
         * Sin embargo, para MINIX el linker no proporciona alineacion
         * para los limites de los clicks , asi un click incompleto de codigo
         * sera tratado como un dato.
         * Corregir tot_bytes, hasta que se excluya el segmento de codigo.
         */
        *data_bytes += *text_bytes;
        *text_bytes = (*text_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
        *data_bytes -= *text_bytes;
        *tot_bytes -= *text_bytes;
#endif

  }
  *pc = hdr.a_entry;     /* direccion inicial para empezar la ejecucion */
```

```
  /* Verifica si el tamaño de los segmentos son factibles . */
  tc = ((unsigned long) *text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  dc = (*data_bytes + *bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  totc = (*tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  if (dc >= totc) return(ENOEXEC);    /* pila debe de tener de tamaño por lo menos 1 click */
  dvir = (*ft == SEPARATE ? 0 : tc);
  s_vir = dvir + (totc - sc);
  m = size_ok(*ft, tc, dc, sc, dvir, s_vir);
  ct = hdr.a_hdrlen & BYTE;               /* longitud de la cabecera */
  if (ct > A_MINHDR) lseek(fd, (off_t) ct, SEEK_SET); /* saltar   cabecera no utilizada */
  return(m);
}
```

```
/*===========================================*
 *                                           *
 *===========================================*/
PRIVATE int new_mem(sh_mp, text_bytes, data_bytes,bss_bytes,stk_bytes,tot_bytes)
struct mproc *sh_mp;          /* codigo puede ser compartido con este proceso */
vir_bytes text_bytes;         /* tamaño del segmento de codigo en bytes */
vir_bytes data_bytes;         /* tamaño de los datos inicializados en bytes */
vir_bytes bss_bytes;          /* tamaño de bss en bytes */
vir_bytes stk_bytes;          /* tamaño del segmento de la pila inicial en bytes */
phys_bytes tot_bytes;         /* memoria total a localizar , incluyendo el gap */
{
/* Localiza la nueva memoria y libera la memoria antigua.  Cambia el mapa e informa
 * del nuevo mapa al kernel.  Anula la nueva imagen del  bss, gap y pila del nucleo
 */

  register struct mproc *rmp;
  vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks, tot_clicks;
  phys_clicks new_base;

#if (SHADOWING == 1)
  phys_clicks base, size;
#else
  static char zero[1024];               /* usado para anular  bss */
  phys_bytes bytes, base, count, bss_offset;
#endif

  /* No necesita asignar codigo si puede ser compartido. */
  if (sh_mp != NULL) text_bytes = 0;

  /* Adquiere la nueva memoria..  Cada una de las  4 partes: text, (data+bss), gap,
   * y stack ocupan un numero entero de clicks, empezando en el click
   * limite.  La parte de datos y bss se ejecuta juntos sin espacio.
   */

  text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  gap_clicks = tot_clicks - data_clicks - stack_clicks;
  if ( (int) gap_clicks < 0) return(ENOMEM);

  /* Verifica si hay un hueco bastante grande.  Si es asi, podemos atrevernos primero
   * a eliminar la memoria antigua antes de colocar la nueva, ya que
   * sabemos que tendra exito.  Si no hubiera bastante , retornar error
   */
  if (text_clicks + tot_clicks > max_hole()) return(EAGAIN);

  /* Hay bastante memoria para la nueva imagen del nucleo.  Eliminar la antigua. */
  rmp = mp;
```

```
#if (SHADOWING == 0)
  if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
        /* El segmento de texto no es compartido por otro proceso, liberarlo. */
        free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
  }
  /* Liberar el segento de datos y la pila. */
  free_mem(rmp->mp_seg[D].mem_phys,
     rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
#endif

  /* Hemos pasado un punto de no retorno.  La imagen antigua del nucleo ha sido
   * eliminada para siempre.    */
  new_base = alloc_mem(text_clicks + tot_clicks);    /* nueva imagen del nucleo */
  if (new_base == NO_MEM) panic("MM hole list is inconsistent", NO_NUM);

  if (sh_mp != NULL) {
        /* Comparte el segmento de codigo. */
        rmp->mp_seg[T] = sh_mp->mp_seg[T];
  } else {
        rmp->mp_seg[T].mem_phys = new_base;
        rmp->mp_seg[T].mem_vir = 0;
        rmp->mp_seg[T].mem_len = text_clicks;
  }
  rmp->mp_seg[D].mem_phys = new_base + text_clicks;
  rmp->mp_seg[D].mem_vir = 0;
  rmp->mp_seg[D].mem_len = data_clicks;
  rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
  rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
  rmp->mp_seg[S].mem_len = stack_clicks;

#if (CHIP == M68000)
#if (SHADOWING == 0)
  rmp->mp_seg[T].mem_vir = 0;
  rmp->mp_seg[D].mem_vir = rmp->mp_seg[T].mem_len;
  rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + rmp->mp_seg[D].mem_len +
gap_clicks;
#else
  rmp->mp_seg[T].mem_vir = rmp->mp_seg[T].mem_phys;
  rmp->mp_seg[D].mem_vir = rmp->mp_seg[D].mem_phys;
  rmp->mp_seg[S].mem_vir = rmp->mp_seg[S].mem_phys;
#endif
#endif

#if (SHADOWING == 0)
  sys_newmap(who, rmp->mp_seg);   /* informa del nuevo mapa al kernel */

  /* Anula los segmentos   bss, gap y  pila */
  bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
  base = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
```

```
  bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
  base += bss_offset;
  bytes -= bss_offset;

  while (bytes > 0) {
        count = MIN(bytes, (phys_bytes) sizeof(zero));
        if (sys_copy(MM_PROC_NR, D, (phys_bytes) zero,
                                                ABS, 0, base, count) != OK) {
              panic("new_mem can't zero", NO_NUM);
        }
        base += count;
        bytes -= count;
  }
#endif

#if (SHADOWING == 1)
  sys_fresh(who, rmp->mp_seg, (phys_clicks)(data_bytes >> CLICK_SHIFT),
                          &base, &size);
  free_mem(base, size);
#endif

  return(OK);
}
```

```
/*===========================================*
 *                patch_ptr                  *
 *===========================================*/
PRIVATE void patch_ptr(stack, base)
char stack[ARG_MAX];            /* puntero a la imagen de la pila en  MM */
vir_bytes base;                 /* direccion virtual de la base de la pila en el usuario */
{
/* Cuando se realiza una llamada exec(name, argv, envp) , se crea una imagen de
 * la pila con los punteros relativos  arg t env  al comienzo de la pila. Ahora esos
 * punteros deben ser  ajustados ya que la pila no se posiciona  en la dierccion 0
 * del espacio de usario
 */

  char **ap, flag;
  vir_bytes v;

  flag = 0;                     /* contador de punteros nulos */
  ap = (char **) stack;         /* apunta a  'nargs' numero de argumentos */
  ap++;                         /* ahora apunta a argv[0] */
  while (flag < 2) {
        if (ap >= (char **) &stack[ARG_MAX]) return;      /* too bad */
        if (*ap != NIL_PTR) {
                v = (vir_bytes) *ap;    /* v es un puntero relativo */
                v += base;              /* ajusta el puntero de la pila */
                *ap = (char *) v;       /* lo pone en la pila */
        } else {
                flag++;
        }
        ap++;
  }
}
```

```
/*===========================================*
 *                load_seg                   *
 *===========================================*/
PRIVATE void load_seg(fd, seg, seg_bytes)
int fd;                 /* descriptor del fichero a leer */
int seg;                /* segmento de codigo o datos */
vir_bytes seg_bytes;    /* tamaño del segmento*/
{
/* Lee el codigo o datos desde fichero exec y los copia a una nueva imagen del nucleo.
 * Este procedimiento es una pequeña truco.  La forma logica de cargar un segmento
 * sería leerlo bloque a bloque y copiar cada bloque al espacio de usuario cada vez.
 * Esto es demasiado lento, asi que vamos hacer trampa.
 * Enviamos el espacio de usuario y la direccion virtual al FS en los 10 bits superiores
 * del descriptor de ficheros, y pasando la direccion virtual del usuario
 * en vez de la direccion del MM.  El sistema de ficheros recuperará esos parámetros cuando
 * le llega la llamada read del MM, que es el unico proceso al que se le permite hacer
 * este truco.  El FS copia entonces el segmento completo directamente al espacio
 * de usuario, evitando completamente al MM.
 */

  int new_fd, bytes;
  char *ubuf_ptr;

  new_fd = (who << 8) | (seg << 6) | fd;
   /* calcula la direccion virtual del segmento que se va a tratar, dicho resultafo se pasa
       en clicks */
  ubuf_ptr = (char *) ((vir_bytes)mp->mp_seg[seg].mem_vir << CLICK_SHIFT);
  while (seg_bytes != 0) {
          /* realiza mediante un bucle transferencias de bloques de 32k salvo el ultimo bloque
              que sera menor o igual a 32 k */
          bytes = (INT_MAX / BLOCK_SIZE) * BLOCK_SIZE;  /*  ( 32767 / 1024 ) * 1024
*/
          if (seg_bytes < bytes)
                  bytes = (int)seg_bytes;
          if (read(new_fd, ubuf_ptr, bytes) != bytes)
                  break;          /* error */
          ubuf_ptr += bytes;
          seg_bytes -= bytes;
   }
}
```

```
/*===========================================*
 *                find_share                 *
 *===========================================*/
PUBLIC struct mproc *find_share(mp_ign, ino, dev, ctime)
struct mproc *mp_ign;                   /*  procesos que no deben ser comparados */
ino_t ino;                              /* parametros que identifican univocamente a un
fichero*/
dev_t dev;
time_t ctime;
{
/* Busca un proceso que es  fichero<ino, dev, ctime> en ejecucion.  No
 * "encuentra" en mp_ign, debido a que es el proceso en cuyo codigo se
 * hizo esta llamada.
 */
  struct mproc *sh_mp;

  for (sh_mp = &mproc[INIT_PROC_NR]; sh_mp < &mproc[NR_PROCS]; sh_mp++) {
        if ((sh_mp->mp_flags & (IN_USE | HANGING | SEPARATE))
                                        != (IN_USE | SEPARATE)) continue;
        if (sh_mp == mp_ign) continue;
        if (sh_mp->mp_ino != ino) continue;
        if (sh_mp->mp_dev != dev) continue;
        if (sh_mp->mp_ctime != ctime) continue;
        return sh_mp;
  }
  return(NULL);
}
```

Cuestiones :

1) ¿ Por qué es preferible el uso de las pilas para pasar los parámetros en vez de usar mensajes ?

2) Cuando un usuario quiere hacer un execve, ¿ cual es el flujo hasta que se empieza a ejecutar *do_exec* ?

3) Traza de la llamada cuando un usuario teclea ls -l f.c g.c