forkexit.c

Llamadas al sistema para administración de procesos

Sergio Sosa de la Fé
a2039@dis.ulpgc.es

Ignacio José López Rodríguez
a2140@dis.ulpgc.es

Tabla de Contenidos

1 Introducción

1.1 Llamadas al sistema en MINIX

Existe un conjunto de llamadas al sistema en MINIX 2.0 que se encarga del manejo de procesos: `fork ()`, `waitpid ()`, `wait ()`, `execve ()`, `exit ()`, `brk ()`, `getpid ()`, `getpgrp()`, `setsid ()` y `ptrace ()`. En este documento veremos la implementación de las llamadas, que aparecen en el manejador de memoria, `fork ()`, `wait ()` y `exit ()`.

Los pasos que se siguen desde que un proceso de usuario hace una llamada al sistema hasta que obtiene el resultado son los siguientes:

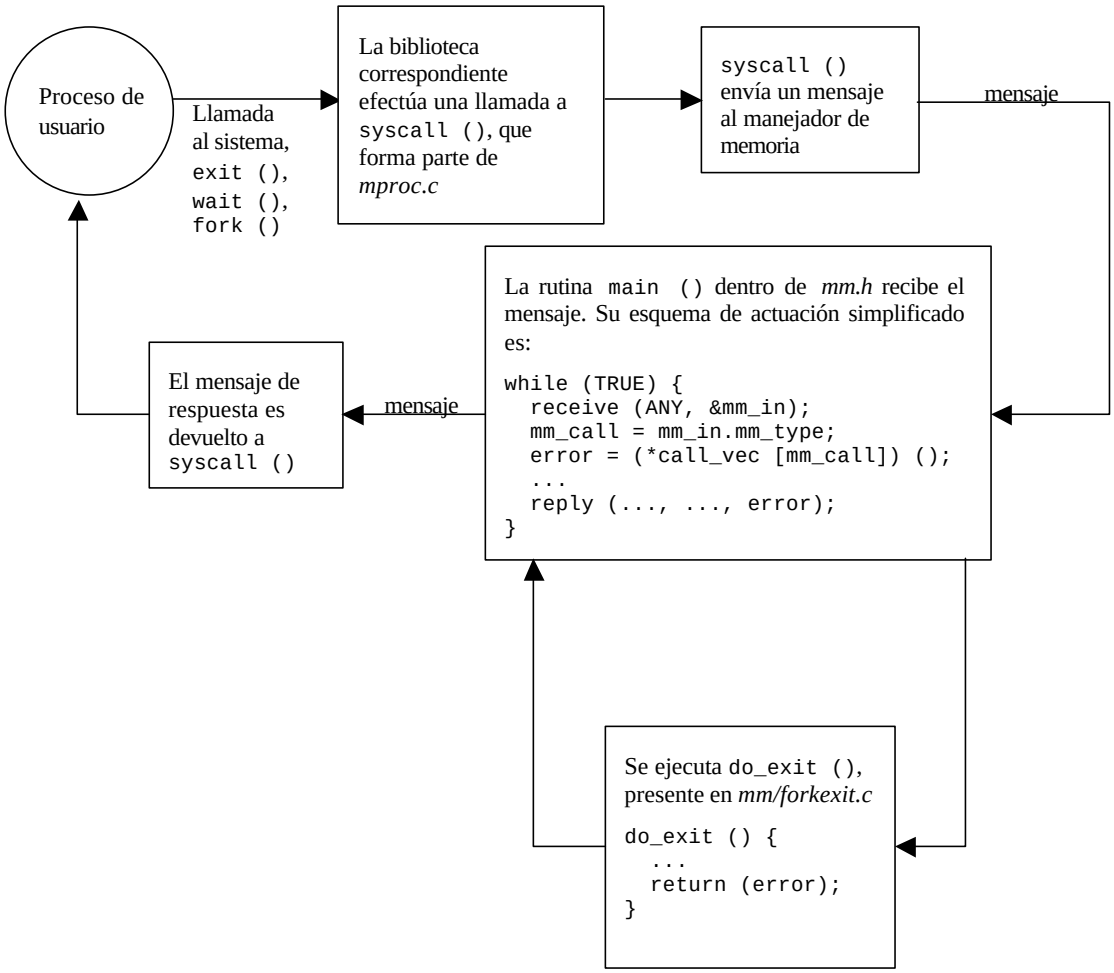

Figura 1.1. *Ejecución de una llamada al sistema.*

Como podemos apreciar en la figura, `call_vec []` es un vector de punteros a funciones (definido en */mm/table.c*). Este, basándose en el número de llamada pasado por un mensaje, permite invocar indirectamente a una función para realizar una tarea.

1.2 Introducción al manejo de la memoria

En MINIX, los procesos están divididos en tres segmentos: el segmento de texto (código), el de datos (con las variables del proceso) y el de pila. Como se puede observar en la *Figura 1.2*, el de datos crece hacia arriba y el de pila hacia abajo.

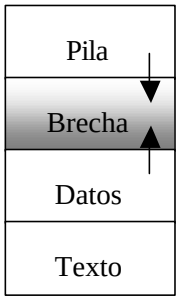

Figura 1.2. *Segmentos de un proceso.*

Podemos apreciar una brecha (*gap*) existente entre el segmento de pila y el de datos. Tanto uno como otro crecen hacia la brecha según sea necesario, con la diferencia de que la pila lo hace automáticamente, en tanto el segmento de datos precisa de la llamada al sistema `brk ()` para poder alterar su tamaño.

MINIX, para los procesos, ordenadas según su dirección, emplea una lista simple de ranuras (*slots*) de memoria. Cuando se necesita memoria, sea para un `fork ()` o un `exec ()`, se usa la primera ranura de la lista con tamaño suficiente. En MINIX, los procesos comparten las instrucciones y los datos en un único espacio de memoria que se asigna y libera como un solo bloque. No obstante, los procesos también pueden compilarse para usar espacios de instrucciones y datos separados.

En el caso de compartir instrucciones y datos en un mismo espacio, cuando se asigna memoria para un proceso (sea en un `fork ()` o en un `exec ()`), se toma memoria como un solo bloque que contendrá las instrucciones y los datos.

En el caso de espacios separados, cuando se hace el `fork ()` inicialmente solo se reserva espacio para los segmentos de pila y datos; el padre y el hijo compartirán el segmento de texto, que contiene el código. Si el proceso hijo hace un `exec ()`, se busca en la tabla de procesos para ver si el nuevo código que se quiere ejecutar ya está en memoria; si es así, se comparte el segmento de texto donde está, y si no, se ubica por primera vez. Consiguientemente, cuando un proceso termina, siempre libera sus segmentos de pila y datos, pero solo libera el de texto si, tras una búsqueda en la tabla de procesos, ve que no existe otro proceso compartiéndolo.

2 Estructuras de datos empleadas

El manejador de memoria tiene dos estructuras de datos fundamentales: la tabla de procesos y la lista de ranuras. Los campos de la tabla de procesos del sistema se reparten entre el manejo de memoria, el manejo de procesos y el sistema de ficheros. La parte del manejador de memoria de la tabla de procesos está definida en *mproc.h*, la cual presentamos a continuación:

```
/* Esta tabla tiene una ranura por proceso. Contiene toda la información para el
 * manejo de memoria de cada proceso. Entre otras cosas, define los segmentos de
 * texto, datos y pila, uids y gids y diversos flags. Un proceso referenciado en
 * esta tabla se encuentra referenciado por el mismo número en las tablas que
 * poseen tanto el núcleo como el sistema de ficheros
 */

EXTERN struct mproc {
  struct mem_map mp_seg [NR_SEGS]; /* Apunta a los tres segmentos del proceso  */
  char mp_exitstatus;              /* Estado del proceso cuando finaliza       */
  char mp_sigstatus;               /* Número de señal para procesos eliminados */
  pid_t mp_pid;                    /* Identificador del proceso                */
  pid_t mp_procgrp;                /* pid del grupo del proceso (para señales) */
  pid_t mp_wpid;                   /* pid por el que espera este proceso       */
  int mp_parent;                   /* Índice del proceso padre                 */

  /* uids y gids reales y efectivos */
  uid_t mp_realuid;                /* uid real del proceso     */
  uid_t mp_effuid;                 /* uid efectivo del proceso */
  gid_t mp_realgid;                /* gid real del proceso     */
  gid_t mp_effgid;                 /* gid efectivo del proceso */

  /* Identificación de ficheros a efectos de compartición */
  ino_t mp_ino;                    /* Nodo-i del fichero                  */
  dev_t mp_dev;                    /* Número de dispositivo del filesystem */
  time_t mp_ctime;                 /* Hora de alteración del nodo-i       */

  /* Información para manejo de señales */
  sigset_t mp_ignore;              /* 1: ignorar señal; 0: no ignorarla */
  sigset_t mp_catch;               /* 1: tratar señal; 0: no tratarla   */
  sigset_t mp_sigmask;             /* Máscara para bloqueo de señales   */
  sigset_t mp_sigmask2;            /* Copia de máscara de bloqueo       */
  sigset_t mp_sigpending;          /* Señales bloqueadas                */
  struct sigaction mp_sigact [_NSIG + 1];
  vir_bytes mp_sigreturn;       /* Dirección de la función __sigreturn ()     */

  /* Esto es para compatibilidad a nivel de señales */
  sighandler_t mp_func;

  unsigned mp_flags;            /* Bits para los flags                        */
  vir_bytes mp_procargs;        /* Puntero a los argumentos iniciales de pila */
} mproc [NR_PROCS];

/* Declaración de constantes */
#define IN_USE        001 /* Activo cuando hay una ranura en uso              */
#define WAITING       002 /* Activo por la llamada al sistema WAIT            */
#define HANGING       004 /* Activo por la llamada al sistema EXIT            */
#define PAUSED        010 /* Activo por la llamada al sistema PAUSE           */
#define ALARM_ON      020 /* Activo cuando se inicia el timer SIGALRM         */
#define SEPARATE      040 /* Activo si el proceso tiene espacios I&D separados */
#define TRACED       0100 /* Activo si el proceso va a ser depurado           */
#define STOPPED      0200 /* Activo si el proceso se detiene para depurado    */
#define SIGSUSPENDED 0400 /* Activo por la llamada al sistema SIGSUSPEND      */

#define NIL_MPROC ((struct mproc *) 0)
```

La lista de ranuras (*slots*) está definida en *alloc.c*. Cada entrada de la lista de ranuras tiene tres campos: la dirección base de la ranura en *clicks* (256 bytes), la longitud de la ranura en *clicks*, y un puntero a la siguiente entrada de la lista. En *forkexit.c* no se emplea directamente.

La estructura `mproc` y sus campos son muy importantes, y por ello requieren un estudio más detallado. A continuación, y desde el punto de vista de las rutinas *fork*, *wait* y *exit*, se describen los campos de mayor interés:

2.1 Campo mp_seg []

El vector `mp_seg[]` tiene tres entradas, las cuales contienen información básica de los segmentos de texto, datos y pila de un proceso particular. Exactamente, en cada entrada se almacena la dirección virtual, la dirección física y la longitud de uno de esos segmentos.

En el código, para hacer referencia a cualquiera de los segmentos o, lo que es lo mismo, a cualquiera de las entradas del vector, no se usan los números enteros *0*, *1* y *2*; en vez de ello, y para mejorar la legibilidad, se usan las macros *T*, *D* y *S*, las cuales, respectivamente, representan a los índices del segmento de texto, de datos y de pila.

Los punteros asociados a los segmentos de texto y datos hacen referencia a la base de los mismos, mientras que para el segmento de pila, debido al sentido de su crecimiento, hacen referencia al top. En la *Figura 2.1* se explican gráficamente los datos almacenados en el vector `mp_seg []`.

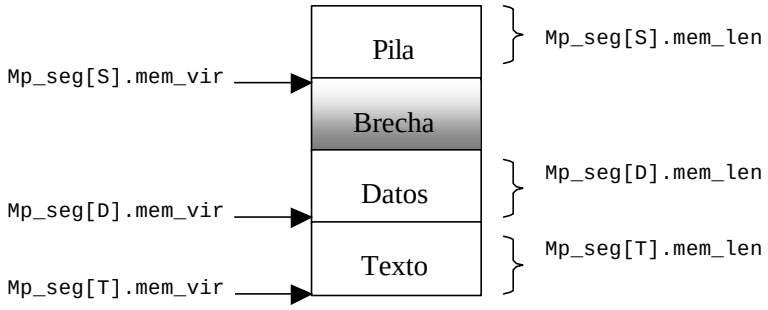

***Figura 2.1** Datos de la estructura mp_seg [] en un proceso con Texto y Datos separados.*

Lo más habitual es que los segmentos de texto y datos se encuentren juntos; en este caso los punteros asociados a ambos son iguales. En la *Figura 2.2* se muestra un ejemplo de dicha situación.

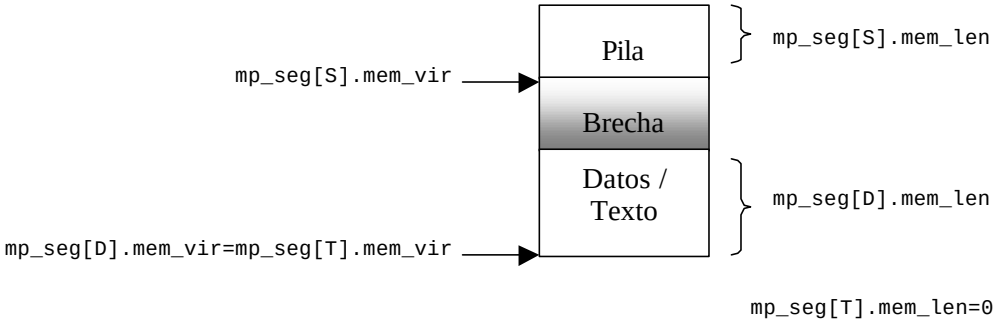

Figura 2.2. *Datos de la estructura mp_seg [] en un proceso con texto y datos juntos.*

En el código, para hallar datos que no se encuentran almacenados de forma explícita, como el espacio de memoria ocupado por un proceso, o la base del segmento de pila, se *juega* con los valores almacenados en la estructura `mp_seg []`. Por ejemplo, para hallar la memoria que se debe reservar para un nuevo proceso hijo se realizan las siguientes operaciones:

```
mp_seg[S].mem_len    +    (mp_seg[S].mem_vir – mp_seg[D].mem_vir)
Longitud del                  Espacio entre el top de la pila
segmento de pila              y la base del segmento de datos
```

A primera vista podría parecer que en el segundo operando de la suma, para hallar el espacio entre el top de la pila y la dirección base del proceso, en vez de la base del segmento de datos, se debería usar la base del segmento de texto; no se utiliza este último dato por las siguientes razones:

- Cuando los segmentos de texto y datos se encuentran separados, el proceso hijo que se crea con la función *fork* no hereda el primero de ellos y, por tanto, sólo necesita almacenar los segmentos de datos y pila.

- En el caso de que dichos segmentos no se encuentren separados, tal y como se vio en la *Figura 2.2*, el campo `mem_len` del puntero al segmento de datos contiene la longitud de la zona de datos y texto.

A cerca de los datos almacenados en la estructura deberíamos comentar que la dirección virtual, la dirección física y la longitud del segmento se encuentran medidos en *clicks* (256 bytes). Por último, apuntar que todos los segmentos empiezan en un límite de *click* y ocupan un número entero de ellos.

2.2 Campos mp_ino, mp_dev y mp_ctime

Tal y como se lee en los comentarios que acompañan en la estructura de datos, los campos `mp_ino []`, `mp_dev []` y `mp_ctime []` hacen referencia, respectivamente, al *nodo-i* de un fichero, al número de un dispositivo del *filesystem* y a la hora de alteración del *nodo-i*.

La utilidad de estos datos radica en que, en conjunción, la información aportada por los tres identifica unívocamente un fichero y que, por tanto, pueden ser utilizados como criterio de distinción de los segmentos de texto. Concretamente, en la implementación de la función *exit* se utilizan para averiguar si existe algún proceso activo que comparta el segmento de texto del proceso que está a punto morir; es decir, para averiguar, en definitiva, si se puede o no liberar el mencionado segmento de texto.

2.3 Campo mp_flags

En general, en el campo `mp_flags` se almacena información sobre el estado en el que se encuentra el proceso asociado a una determinada entrada de la tabla. Decimos *en general* porque, más bien, algún dato hace referencia a la propia entrada y no al proceso que contiene.

Tal y como aparece en la definición de la estructura `m_proc`, los valores que se almacenan en el campo `mp_flags` son los siguientes:

```
#define IN_USE   001 /* Activo cuando hay una ranura en uso   */
#define WAITING  002 /* Activo por la llamada al sistema WAIT */
#define HANGING  004 /* Activo por la llamada al sistema EXIT  */
#define PAUSED   010 /* Activo por la llamada al sistema PAUSE  */
#define ALARM_ON 020 /* Activo cuando se inicia el timer SIGALRM */
#define SEPARATE 040 /* Activo si el proceso tiene espacios I&D separados */
#define TRACED   0100 /* Activo si el proceso va a ser depurado   */
#define STOPPED  0200 /* Activo si el proceso se detiene para depurado  */
#define SIGSUSPENDED 0400 /* Activo por la llamada al sistema SIGSUSPEND  */
```

Prácticamente el significado de todos ellos se explica por los comentarios pero, en cualquier caso, a continuación se comentan los que creemos más importantes o, por lo menos, los que se usan en las funciones *fork*, *exit* y *waitpid.*

- **IN_USE:** Este valor con más propiedad hace referencia al propio slot de la tabla que al proceso que pudiera contener este último. Cuando una ranura tiene activo esta flag quiere decir que contiene un proceso y que, por tanto, no está libre. Como se podrá comprobar, en el código siempre se usa para encontrar un slot libre o para comprobar si de él se pueden leer datos válidos; datos pertenecientes a un proceso.

- **HANGING:** Si un proceso finaliza su ejecución y el padre del mismo no está esperando por él, entonces el primero entra en un estado que se denomina zombi (descrito en el punto 4.1); para representar este estado o situación se activa el flag `HANGING`. En la función *waitpid* se utiliza esta información para saber si se puede retornar o no; para saber si hay que seguir esperando o no por un proceso hijo.

- **WAITING:** Esta flag sirve para indicar si un proceso se encuentra o no en espera de uno de sus hijos. En la función *exit* se utiliza para comprobar si el padre del proceso que

termina está esperando por él, ya que si no es así éste entra en estado zombi. En la función *waitpid*, el flag `WAITING` se usa cuando es necesario poner en espera al padre de un proceso que no ha finalizado su ejecución.

- **ALARM_ON:** Este flag está activo cuando el proceso tiene un timer pendiente; es decir, indica que en el futuro el proceso va a ser interrumpido. Este flag es comprobado en la función `exit`, para saber si hay que anular el timer del proceso que termina, puesto que si no se podría intentar interrumpir a un proceso que ya no existe.

- **SEPARATE:** Cuando este flag se encuentra activo quiere decir que el proceso en cuestión ha sido compilado con los segmentos de datos y texto separados.

- **TRACED y STOPPED:** Estos sirven para advertir si un proceso se encuentra en modo depuración. El primero indica si el proceso está siendo *rastreado* y el segundo si se encuentra detenido.

Como se puede apreciar, todos los valores (octales) que se definen sólo tienen un bit activado y, además, estos entre sí no coinciden. Con esta forma de operar se logra que un proceso pueda tener más de un estado asociado y que la lectura independiente de ellos, así como otras operaciones, se puedan realizar de forma sencilla.

2.4 Otros campos

El resto de los campos se explican por sus comentarios. De todas formas, seguidamente comentamos brevemente, entre ellos, los más importantes o los que más se usan en el código que se expone en puntos ulteriores.

- **`mp_exitstatus:`** En este campo se almacena el estado del proceso cuando finaliza su ejecución. En el byte más significativo se almacena el valor de retorno y en el menos significativo el número de la señal, si la hubo, que produjo la salida (el campo `mp_sigstatus`).

- **mp_sigstatus:** En este campo, cuando un proceso es cancelado, se almacena el número de la señal que produjo tal evento; que produjo su eliminación.

- **mp_pid:** Número identificador de proceso (pid). Son siempre positivos.

- **mp_procgrp:** Número identificador del grupo del proceso. Un líder de sesión, para su fácil distinción, tiene su identificador de proceso y grupo iguales.

- **`mp_wpid:`** pid del proceso por el que se está esperando. Lógicamente, la información de este campo sólo es válida si el flag `WAITING` se encuentra activo; es decir, sólo si está esperando por un proceso.

- **`mp_parent:`** Indice a la entrada del padre en la tabla de procesos. En el código se usa, entre otros, para averiguar si el padre de un proceso que sale está esperando, o para saber si un proceso es hijo de otro.

3 Llamada al sistema `fork ()`

3.1 Descripción.

Cuando se invoca a `fork()` se crea un nuevo proceso, comúnmente denominado *proceso hijo*. Este nuevo proceso se inicia con una copia exacta de los datos del proceso que realiza la llamada, incluyendo las variables, descriptores de ficheros, registros, etc. Una vez se hayan iniciado estos datos, los dos procesos, el padre y el hijo, siguen flujos de ejecución independientes. La llamada a `fork()` devuelve un cero al hijo y el pid del hijo al padre.

3.2 Pasos de ejecución.

Tal y como se muestra en la bibliografía de la asignatura, a grandes rasgos, los pasos que realiza la función `fork` son los siguientes:

1. Verifica que no esté llena la tabla de procesos.
2. Trata de asignar memoria para los datos y la pila del hijo.
3. Copia los datos y la pila del padre en la memoria del hijo.
4. Encuentra una ranura de proceso libre y copia en ella la ranura del padre.
5. Introduce el mapa de memoria del hijo en la tabla de procesos.
6. Escoge un pid para el hijo.
7. Informa al Kernel y al sistema de archivos de la creación del nuevo proceso.
8. Informa al Kernel del mapa de memoria del hijo.
9. Envía mensajes de respuesta al padre y al hijo.

Expresados de forma algorítmica, los pasos son los siguientes:

```
SI la tabla de procesos está llena ENTONCES
  Proporcionar un mensaje de error
FIN SI

Intenta tomar memoria para los datos y la pila del hijo
Copia los datos y la pila del padre a la memoria del hijo

PARA todas las ranuras de la tabla de procesos HACER
  SI se encuentra una ranura libre ENTONCES
    Salir del bucle
  FIN SI
FIN PARA

Copia el contenido de la ranura del padre en la del hijo
Mete el mapa de memoria del hijo en la tabla de procesos

REPETIR
  Explorar pids del sistema
HASTA obtener un pid libre

Informar al núcleo y al sistema de ficheros de la existencia del hijo
Proporcionar el mapa de memoria del hijo al núcleo
Enviar mensajes de respuesta al padre y al hijo
```

3.3 Código comentado.

La llamada al sistema `fork ()` es implementada por la función `do_fork ()`, que se encuentra ubicada en el fichero */mm/forkexit.c*. A continuación se muestra el código comentado de esta:

Declaración de variables. Las más importantes de ellas son `rmp` y `rmc`, las cuales apuntan, en la tabla de procesos, a las ranuras del proceso padre y el hijo; es necesario anotar que existen zonas del código, al final, en que éstas se *reutilizan* para otros cometidos. El resto de las variables, si es necesario, se comentan a medida que vayan apareciendo.

```
16832 PUBLIC int do_fork ()
16833 {
16834   /* El proceso apuntado por 'mp' se ha bifurcado. Creamos un proceso hijo */
16835
16836   register struct mproc *rmp;  /* Puntero al padre */
16837   register struct mproc *rmc;  /* Puntero al hijo */
16838   int i, child_nr, t;
16839   phys_clicks prog_clicks, child_base = 0;
16840   phys_bytes prog_bytes, parent_abs, child_abs; /* Solo para INTEL */
```

Paso 1. El primer `if` se asegura que no existan tantos procesos activos como número de entradas en la tabla de procesos. Luego, si la condición anterior se cumple, el segundo `if` verifica que al menos en la tabla de procesos quedan dos (`LAST_FEW`) ranuras libres para el root. Para comprobar si es el root el que realiza la llamada, se lee el `uid efectivo` del proceso padre.

```
16841
16842   /* Si las tablas de proceso se llenan en algún momento, se proporciona un
16843    * mensaje de error
16844    */
16845   rmp = mp;
16846   if (procs_in_use == NR_PROCS) return (EAGAIN);
16847   if (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0) return (EAGAIN);
```

Paso 2. El cometido de las líneas 16852 y 16853 es almacenar en la variable `prog_clicks` el espacio necesario para el nuevo proceso; este cálculo se realiza según se explicó en el punto 2.1 de este documento. Luego, en la línea 16854 se pasa de clicks a bytes el espacio que se necesita. Por último, en la línea 16855 se invoca la función `alloc_mem` para reservar la memoria; la dirección base de dicha zona de memoria se almacena en `child_base`.

```
16848
16849  /* Se determina cuanta memoria se reservará. Solo se necesita copiar datos y
16850   * pila, dado que el segmento de texto o está compartido o es de longitud cero
16851   */
16852  prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
16853  prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
16854  prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
16855  if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(EAGAIN);
```

Paso 3. En las líneas 16858 y 16859 se pasa de clicks a bytes la dirección base de la zona de memoria del hijo y la dirección base de la zona de datos del padre. Luego, en la línea 16860, por medio de la función `sys_copy`, se copian todos los datos y la pila del padre en la memoria del hijo; el parámetro `ABS` indica que las direcciones son absolutas. Por último, en la línea 16861 se comprueba si se pudo realizar la copia de los datos; si no fue así, se muestra un mensaje de error.

```
16856
16857     /* Crea una copia de la imagen del padre para el hijo */
16858     child_abs = (phys_bytes) child_base << CLICK_SHIFT;
16859     parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
16860     i = sys_copy(ABS, 0, parent_abs, ABS, 0, child_abs, prog_bytes);
16861     if (i < 0) panic("do_fork can't copy", i);
```

Paso 4. El bucle que comienza en la línea 16864 busca para el hijo una ranura en la tabla de procesos que no tenga asociado ningún proceso activo; esta comprobación se realiza mediante el flag `IN_USE` (comentado en el punto 2.3). La sentencia de la línea 16868 halla el índice de dicha ranura. Luego, en la línea 16869 se incrementa el número de procesos activos (`procs_in_use`). En la línea 16870 se copia la estructura `m_proc` del padre en la ranura del hijo. En las líneas 16872 y 16873 se cambian algunos parámetros de dicha estructura; se indica al hijo quien es el padre (variable global `who`) y se desactiva, si existe, el estado de traza que pudo heredar de él (`~TRACED`).

```
16862
16863   /* Busca un ranura existente en 'mproc' para el proceso hijo */
16864   for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
16865     if ( (rmc->mp_flags & IN_USE) == 0) break;
16866
16867   /* Prepara el hijo y su mapa de memoria; copia la ranura del padre */
16868   child_nr = (int)(rmc - mproc); /* Número de ranura del hijo */
16869   procs_in_use++;
16870   *rmc = *rmp;                  /* Copia la ranura del proceso padre al hijo */
16871
16872   rmc->mp_parent = who;         /* Almacena el apuntador al padre */
16873   rmc->mp_flags &= ~TRACED;     /* El hijo no hereda el estado de traza */
```

Paso 5. En el `if` de la línea 16877 se comprueba, por medio del flag `SEPARATE`, si los datos y el texto no van separados; en ese caso, no se comparte código y, como se explicó en el punto 2.1, el puntero del segmento de texto del hijo apunta a la misma dirección que el puntero del segmento de datos. Independientemente de si se comparte o no el texto, como también se explicó en el punto 2.1, el segmento de datos apunta a la zona de memoria reservada (línea 16878). Luego, en la línea 16879 se halla la dirección del top de la pila (explicado en el punto 2.1). En las dos líneas que vienen a continuación (16881 y 16882) se inician algunos campos (estado de salida) de la estructura `m_proc`.

```
16874   /* Un hijo con I&D separados mantiene el segmento de texto del padre.
16875    * Los segmentos de datos y pila deben referenciar la nueva copia
16876    */
16877   if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
16878     rmc->mp_seg[D].mem_phys = child_base;
16879     rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
16880                         (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
16881   rmc->mp_exitstatus = 0;
16882   rmc->mp_sigstatus = 0;
```

Paso 6. El objetivo de las siguientes instrucciones es hallar, para el hijo, un pid que no se encuentre asociado a ningún proceso. Esto se realiza mediante dos bucles: en el primero se escoge de forma circular (límite 30000) un pid, y en el segundo se comprueba si dicho pid ya se encuentra asignado; en caso de que no sea así, se sale de los bucles. En la línea 16887 se coge el nuevo pid a comprobar. El bucle que comienza en la línea 16888 recorre la tabla de procesos. En la línea 16889 se comprueba si el proceso de la entrada actual está usando ese pid (`rmp->mp_pid==next_pid`) o pertenece a un grupo con ese pid (`rmp->mp_prcgrp==pid`); si está siendo usado se vuelve al `do-while` externo (línea 16891), si no, como `t` es igual a cero, se sale del `do-while`.

```
16883
16884   /* Encuentra un pid libre para el hijo y lo pone en la tabla */
16885   do {
16886      t = 0;               /* 't' = 0 implica un pid aún libre */
16887      next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PID + 1);
16888      for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
16889        if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
16890          t = 1;
16891          break;
16892        }
16893      rmc->mp_pid = next_pid; /* Asigna pid al hijo */
16894   } while (t);
```

Paso 7. En la línea 16897, la función de librería `sys_fork` envía un mensaje al núcleo indicando que se ha creado un nuevo proceso e informando de algunos de sus datos: pid (`rmc->mp_pid`), base de su zona de memoria (`child_base`), índice en la tabla de procesos (`child_nr`) y el índice del padre (`who`). Luego, en la línea 16898, la función `tell_fs` , también a través de un mensaje, informa al sistema de ficheros sobre la creación de un nuevo proceso; le pasa como datos: su índice en la tabla (`child_nr`), el pid (`rmc->mp_pid`) y el índice del padre (`who`).

```
16895
16896   /* Avisa al núcleo y sistema de ficheros del FORK efectuado con éxito */
16897   sys_fork(who, child_nr, rmc->mp_pid, child_base);
16898   tell_fs(FORK, who, child_nr, rmc->mp_pid);
```

Paso 8. La función de librería `sys_newmap` envía un mensaje al núcleo con los punteros de los segmentos del nuevo proceso (`rmc->mp_seg`); de esta forma el núcleo puede construir el mapa de memoria del hijo (`child_nr`).

```
16899
16900   /* Informa del mapa de memoria del hijo al núcleo */
16901   sys_newmap(child_nr, rmc->mp_seg);
```

Paso 9. La función `replay` despierta al hijo para comience su ejecución. Por último, al proceso padre, por medio del `return`, se le devuelve el pid del proceso hijo (`next_pid`).

```
16902
16903   /* Respuesta para que el hijo despierte */
16904   reply(child_nr, 0, 0, NIL_PTR);
16905   return(next_pid);             /* pid del hijo */
16906 }
```

4 Llamada al sistema `exit ()`

4.1 Descripción.

Se considera que un proceso ha finalizado cuando se han producido los dos siguientes eventos conjuntamente:

- El proceso ha efectuado un `exit ()` o ha sido abortado por una señal.
- El padre del proceso ha ejecutado un `wait ()` para averiguar el estado de su hijo.

Un proceso que ha salido o que ha sido cancelado, pero cuyo padre todavía no ha realizado un `wait` por él, queda en una especie de animación suspendida, denominada comúnmente *estado zombi*. Cuando un proceso entra en este estado no puede ser planificado, se cancela, si estaba activo, su temporizador de alarma y se libera su memoria; pero no se retira su entrada de la tabla de procesos. Cuando el padre finalmente ejecuta el `wait`, la ranura de la tabla de procesos se libera y se informa de ello al sistema de archivos y al kernel.

Un problema que podría surgir es que el padre del proceso que está saliendo esté muerto, es decir, que haya sido cancelado con anterioridad; si no se realizara ninguna acción especial para controlar esta situación, el proceso saliente, el que ahora termina, quedaría en estado zombi indefinidamente. Para solucionar este problema se opta por modificar las tablas y convertir a todos los hijos de un proceso que sale en hijos del proceso `init`. Si se recuerda, este último entra en un bucle infinito en el que espera por la muerte de algunos de sus hijos (terminales); con esta forma de operar los zombis huérfanos se eliminan rápidamente.

La llamada al sistema `exit ()` está implementada en las funciones `do_mm_exit ()` y `mm_exit ()`. La primera de ellas es la que acepta la llamada, pero prácticamente todo el trabajo corre a cargo de la llamada `mm_exit ()`. La razón por la que se hace esta división es que la función `mm_exit` es invocada por los procesos que han sido terminados por una señal; el trabajo es el mismo, pero los parámetros son diferentes.

4.2 Pasos de ejecución.

Los pasos de ejecución de la función `mm_exit` se podrían resumir en los siguientes puntos:

1. Se averigua si el proceso que termina es *líder de sesión*.
2. Se elimina el timer pendiente que pudiera tener el proceso.
3. Se informa al núcleo y al sistema de ficheros de que ya no se puede ejecutar el proceso.
4. Se libera la memoria ocupada por el proceso.
5. Si el padre del proceso que ahora termina estaba esperando (`wait`) por él, se libera de la tabla de procesos su ranura (la del hijo).
6. Si el proceso que sale tiene hijos, estos se desheredan y se incorporan como hijos del proceso `init`.
7. En caso de que el proceso fuera líder de sesión, se envía una señal de *cuelgue* (suspender) a todos los procesos de su grupo.

Los pasos expresados de forma algorítmica son los siguientes:

```
Se comprueba si el proceso que termina es líder de sesión
SI el proceso que termina tiene un timer pendiente ENTONCES
  Eliminar el timer
FIN SI

Se informa al núcleo y al FS de que el proceso ya no es ejecutable

SI el segmento de texto no está siendo compartido ENTONCES
  Liberar el segmento de texto del proceso
FIN SI

Liberar segmentos de datos y pila del proceso

SI el padre del proceso que termina ha hecho un wait ()
   y el proceso por el que espera es justamente este ENTONCES
  Se invoca a cleanup () para liberar la ranura del proceso
  Se devuelve el código de salida al padre
SI NO
  Se suspende el proceso que termina (pasa a ser zombi)
FIN SI

SI el proceso que termina tiene hijos ENTONCES
  Hacer que el nuevo padre del proceso sea init

  SI init está esperando (siempre cierto en condiciones normales)
      y el proceso es zombi ENTONCES
    Invocar a cleanup () para eliminar al hijo definitivamente
  FIN SI
FIN SI

SI el proceso que termina era líder de sesión ENTONCES
   Se envía una señal de cuelgue a todos los procesos de su grupo
FIN SI
```

Como puede observarse, en este último paso se controla que los procesos hijos del que termina no queden eternamente en el sistema. Por ejemplo, en la figura siguiente se observa como los hijos del proceso 12, que hace un `exit ()`, se enganchan como hijos del proceso `init`. Como `init` estará esperando, estos procesos hijos (52 y 53) terminarán poco después.

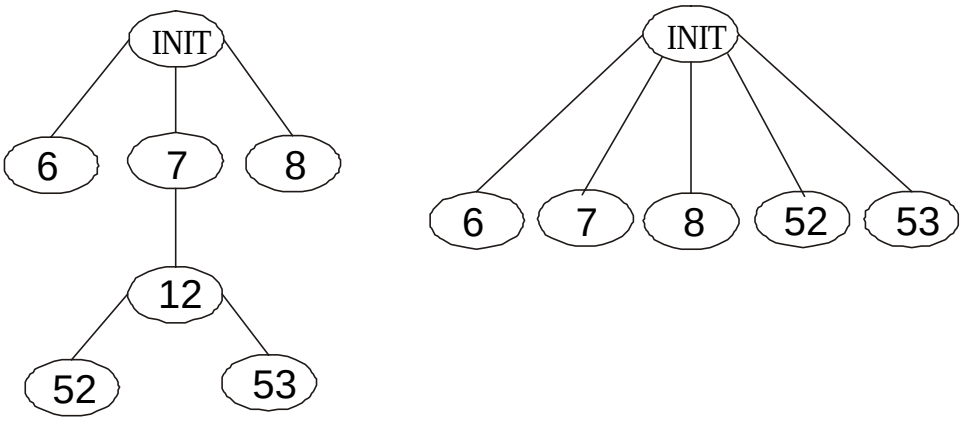

Figura 4.1. *Ejemplo de llamada al sistema exit ().*

4.3 Código comentado.

El código de `do_mm_exit` es muy corto:

```
16912 PUBLIC int do_mm_exit ()
16913 {
16914   /* Ejecuta la llamada al sistema EXIT. La auténtica llamada es realizada por
16915    * mm_exit (), también invocada cuando un proceso muere por una señal
16916    */
16917
16918   mm_exit(mp, status);
16919   dont_reply = TRUE;      /* No responder al proceso recientemente terminado */
16920   return(OK);
16921 }
```

Vemos que `do_mm_exit ()` básicamente lo que hace es llamar a `mm_exit ()`, que es la función que realmente realiza todo el trabajo. Quizás, cabría destacar que en `do_mm_exit ()` se asigna a `dont_reply` el valor `TRUE` para que no se dé un mensaje de respuesta desde aquí.

A continuación se comenta el código de la función `mm_exit ()`.

Declaración e iniciación de variables. De las variables que se declaran, la más importante de ellas es `rpm`, que es un puntero al slot del proceso que termina. Luego, en la línea 16940, haciendo uso de aritmética de punteros, se halla el índice de dicho slot y se almacena en `proc_nr`.

```
16927 PUBLIC void mm_exit(rmp, exit_status)
16928 register struct mproc *rmp;  /* Puntero al proceso a finalizar */
16929 int exit_status;             /* Código de salida del proceso (para el  padre) */
16930 {
16931   /* Ha finalizado un proceso. Hay que liberar las posesiones del mismo. Si el
16932    * padre está esperando, libera el resto, y si no, se cuelga.
16933    */
16934
16935   register int proc_nr;
16936   int parent_waiting, right_child;
16937   pid_t pidarg, procgrp;
16938   phys_clicks base, size, s;        /* base y size solo se usan en 68000 */
16939
16940   proc_nr = (int) (rmp - mproc);    /* Obtiene el número de ranura del proceso */
```

Paso 1. En la línea 16943 se pregunta si el pid del proceso es igual al del grupo, lo que equivale a preguntar si el proceso es líder de sesión; en ese caso se almacena el valor de dicho pid y, si no es así, se almacena un cero.

```
16941
16942   /* Recordar grupo de procesos de un líder de sesión */
16943   procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;
```

Paso 2. En la línea 16946 se comprueba si el proceso tiene una alarma activa. En ese caso, mediante la función `set_alarm`, ésta se anula, puesto que el proceso va a dejar de existir y, por tanto, no se debe intentar interrumpirlo posteriormente.

```
16944
16945   /* Si el proceso tiene un timer pendiente, lo elimina */
16946   if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);
```

Paso 3. En la línea 16949 se invoca a la función `tell_fs` para informar al sistema de ficheros de que un proceso ha finalizado su ejecución; realmente lo que hace `tell_fs` es enviar un mensaje al *filesystem* indicando el evento de salida (`EXIT`) y el número del slot del proceso que sale (`proc_nr`). Luego, en la línea 16950 se avisa al núcleo de la finalización de un proceso. La función de librería `sys_xit` envía al núcleo un mensaje con información del proceso que sale: número del slot del proceso (`proc_nr`), base (`base`) y tamaño (`size`) de su zona de memoria y el número del slot del padre (`rmp->mp_parent`)

```
16947
16948   /* Avisa al núcleo y al FS de que ya no se puede ejecutar el proceso */
16949   tell_fs(EXIT, proc_nr, 0, 0);  /* file system can free the proc slot */
16950   sys_xit(rmp->mp_parent, proc_nr, &base, &size);
```

Paso 4. En la línea 16953 se invoca a la función `find_share` para averiguar si el segmento de texto del proceso que va a salir está compartido; si no es el caso, en la línea 16955 éste se libera. En el punto 2.2 de este documento se explica el significado de los tres últimos parámetros que se le pasan a `find_share`. Luego, en las líneas 16958 y 16959 se libera el espacio ocupado por el segmento de datos, por el de pila y la brecha; básicamente, para hallar la longitud del espacio ocupado, se resta la base de la pila a la base del segmento de datos.

```
16951
16952     /* Libera la memoria ocupada por el hijo */
16953     if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
16954        /* No hay otros procesos compartiendo texto, luego se elimina */
16955        free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
16956     }
16957     /* Libera los segmentos de datos y de pila */
16958     free_mem(rmp->mp_seg[D].mem_phys,
16959        rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
```

Paso 5. En la línea 16962 se almacena el estado de salida del hijo (la variable `exit_status` se pasó por parámetro). En la línea 16964 se comprueba si el padre está esperando por algún hijo, y en la línea 16963 se recoge el pid del proceso hijo por el que está esperando (`pidarg`). Luego en las líneas 16965-16968, con la información anterior, se realizan tests para comprobar si él, el proceso que termina, está siendo esperado por el padre; el resultado del test se almacena en la variable lógica `right_child`. Si `right_child` es `TRUE`, en la línea 16970 se llama a `cleanup` para liberar la ranura del proceso que sale; en el otro caso (`right_child` es `FALSE`), en la línea 16972 se añade a los flags del proceso el estado zombi (`HANGING`).

```
16960
16961   /* El slot del proceso solo puede liberarse si el padre ha hecho un WAIT */
16962   rmp->mp_exitstatus = (char) exit_status;
16963   pidarg = mproc[rmp->mp_parent].mp_wpid;       /* ¿Por quién se espera? */
16964   parent_waiting = mproc[rmp->mp_parent].mp_flags & WAITING;
16965   if (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp)
16966      right_child = TRUE;       /* El hijo cumple uno de los tres tests */
16967   else
16968      right_child = FALSE;      /* El hijo falla todos los tests */
16969   if (parent_waiting && right_child)
16970      cleanup(rmp);             /* Avisa al padre y libera el slot del hijo */
16971   else
16972      rmp->mp_flags |= HANGING; /* El padre no espera; se suspende el hijo */
```

Paso 6. El bucle de las líneas 16974-16982 recorre la tabla de procesos en busca de hijos del proceso que ha finalizado su ejecución. Con más detalle, en la línea 16976 se pregunta si el proceso de la ranura actual está activo (flag `IN_USE`) y si el padre de éste es el que ha finalizado (`rmp->parent==proc_nr`). Si estas condiciones se cumplen, en la línea 16978 al proceso de la ranura actual se le pone como padre el `init`; luego, en la línea 16980, si el `init` está esperando y el proceso desheredado se encuentra en estado zombi, se llama a la función `cleanup` para que lo borre de la tabla de procesos.

```
16973
16974   /* Si el proceso tiene hijos, se desheredan (init los adoptará) */
16975   for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
16976     if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
16977       /* 'rmp' apunta a un hijo que será desheredado */
16978       rmp->mp_parent = INIT_PROC_NR;
16979       parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
16980       if (parent_waiting && (rmp->mp_flags & HANGING)) cleanup(rmp);
16981     }
16982   }
```

Paso 7. Si el proceso que finaliza su ejecución es un líder de sesión (`procgrp`), entonces se envía una señal de *cuelgue* (suspender) al resto de los procesos de su grupo. – En el *paso 1* se halla si el proceso que termina es líder de sesión.

```
16983
16984   /* Envía una señal de suspensión al grupo del proceso si era líder de sesión */
16985   if (procgrp != 0) check_sig(-procgrp, SIGHUP);
16986 }
```

5 Llamadas al sistema `wait ()` y `waitpid ()`

5.1 Descripción.

La llamada al sistema `wait ()` pone al proceso que la ejecuta en espera por la muerte de un hijo. En MINIX 2.0 se introduce la llamada al sistema `waitpid ()` que permite, además, indicar el pid del hijo por el que se espera. Ambas son implementadas por la función `do_waitpid ()` de `forkexit.c`.

La llamada a `waitpid ()` tiene 3 parámetros. El primero puede ser un pid válido (positivo), para indicar que se espera por un hijo concreto; –1 para indicar que se espera por la muerte de uno cualquiera; o 0 para especificar que espera por la muerte de cualquier hijo que tenga su identificador de grupo. En el tercer parámetro, con el valor `WNOHANG`, se le puede indicar que, si no ha finalizado alguno de los hijos especificados en el primer argumento, no se quiere esperar a que lo hagan; es decir, se quiere una llamada no bloqueante. El segundo parámetro de `waitpid`, y a la vez el único que admite `wait`, es un puntero a la variable donde se tiene que almacenar el estado de salida del proceso hijo.

Por ejemplo, para hacer que un padre espere por cualquiera de sus hijos se podrían usar cualquiera de las siguientes sentencias:

```
wait(&status);

waitpid(-1,&status,0);
```

Si lo que queremos es que el padre no se detenga si no existen hijos que hayan finalizado su ejecución, tenemos que teclear:

```
waitpid(-1,&status,WNOHANG);
```

Y como último ejemplo, si queremos que el padre espere por un hijo determinado habría que escribir:

```
waitpid(pid_hijo,&status,0);
```

5.2 Pasos de ejecución.

Los pasos que se siguen en esta función son:

1. Hallar los parámetros de la llamada, puesto que esta puede realizarse como `wait ()` o `waitpid ()`.
2. Se recorre la tabla de procesos, y para cada hijo se realizan las siguientes operaciones:
   2.1. Se comprueba si el pid del proceso hijo se ajusta a la llamada del `wait` o del `waitpid`.
   2.2. Si el proceso hijo supera el test anterior y se encuentra en estado zombi, se libera el slot de este último y se informa al padre (se sale de `do_waitpid`).
   2.3. Si el proceso hijo supera el test del punto 2.1 y está siendo depurado, se informa al padre de la situación (se sale de `do_waitpid`).
3. Si al menos un proceso superó el test del punto 2.1, según el tercer parámetro (`WNOHANG`), se pone o no al padre en estado de espera.

Estos puntos expresados de forma algorítmica quedan de la siguiente forma:

```
Obtener el pid del hijo por el cual se quiere esperar

SI la llamada fue waitpid () ENTONCES
  Obtener opciones adicionales
FIN SI

Pone el número de hijos del proceso que no han terminado a cero

PARA todo proceso i HACER
  SI i es hijo del proceso que hace wait () o waitpid () ENTONCES
    SI se espera por un pid concreto diferente del de i ENTONCES
      Continuar (siguiente vuelta de bucle)
    FIN SI

    Incrementa el número de hijos aceptables

    SI i es un zombi ENTONCES
      Invocar a cleanup () para eliminarlo
      Salir de la rutina
    FIN SI

    SI i está detenido (en estado de traza) ENTONCES
      Enviar un mensaje de respuesta al padre
      Salir de la rutina
    FIN SI
  FIN SI
FIN PARA

SI el número de hijos pendientes de salir es positivo ENTONCES
  SI el padre no estaba esperando ENTONCES
    Salir de la rutina
  FIN SI

  Poner al padre en espera y salir
SI NO
  El padre no tiene hijos por los que esperar (es un error)
FIN SI
```

5.3 Código comentado.

Declaración de variables. La variables que, quizás, habría que destacar es `children` (línea 17000). Esta se usa para averiguar si existe algún proceso hijo que no haya finalizado; es decir, para saber si el proceso padre tiene que esperar.

```
16992 PUBLIC int do_waitpid()
16993 {
16994   /* Un proceso quiere esperar a que un hijo termine. Si al menos uno ya está
16995    * esperando, eliminarlo y finalizar la llamada. En otro caso, se espera
16996    * realmente por un hijo. Tanto wait () como waitpid () se llevan a cabo aquí
16997    */
16998
16999   register struct mproc *rp;
17000   int pidarg, options, children, res2;
```

Paso 1. En la línea 17009 se halla el primer parámetro de la llamada; si la invocación fue a `wait` el valor de éste es –1, y si fue a `waitpid` su valor es pid. En la línea 17010 se halla el tercer parámetro de la llamada; si la invocación fue a `wait` éste es 0, si no, se recoge el valor de la variable `sig_nr`, la cual puede contener `WNOHANG`. En la línea 17011 se comprueba si el pid pasado como parámetro fue 0, en cuyo caso se guarda, en negativo, el grupo del proceso invocador.

```
17001
17002   /* Un proceso que invoca WAIT nunca recibe una respuesta por la vía normal del
17003    * reply () en el bucle principal (a menos que se haya activado WNOHANG o
17004    * no haya hijos apropiados). Si un hijo ya ha finalizado, cleanup ()
17005    * envía la respuesta para despertar al invocador
17006    */
17007
17008   /* Prepara algunas variables internas, según se haga wait () o waitpid () */
17009   pidarg  = (mm_call == WAIT ? -1 : pid);    /* Primer parámetro de waitpid () */
17010   options = (mm_call == WAIT ?  0 : sig_nr); /* Tercer parámetro de waitpid () */
17011   if (pidarg == 0) pidarg = -mp->mp_procgrp; /* pidarg < 0 ==> proc grp */
```

Paso 2. El bucle definido en las líneas 17019-17041 recorre la tabla en busca de hijos del proceso invocador (definido por el puntero `rp`). En la línea 17020 se comprueba si en el slot actual está ocupado por un proceso (IN_USE) y si, además, éste es hijo del invocador (`rp->mp_parent==who`). Es necesario apuntar que la variable `who` contiene el índice del proceso llamador.

```
17012
17013   /* ¿Hay algún hijo esperando su finalización? Aquí, pidarg != 0:
17014    *   pidarg  >  0 implica que pidarg es el pid de un proceso por el que esperar
17015    *   pidarg == -1 implica esperar por cualquier hijo
17016    *    pidarg  < -1 implica esperar por cualquier hijo cuyo grupo sea -pidarg
17017    */
17018   children = 0;
17019   for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
17020     if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who) {
                  .
                  .
                  .
17040     }
17041   }
```

Paso 2.1. En la línea 17022, si en la llamada se especificó un pid (`pidarg > 0`) y éste no coincide con el del proceso de la entrada actual (`pidarg !=rp->mp_pid`), se salta a la siguiente iteración del bucle. En la línea 17023, si se especificó un pid de grupo (`pidarg < -1`) y éste no coincide con del el del proceso de la entrada actual (`-pidarg != rp->mp_procgrp`), se salta a la siguiente iteración del bucle. Por último, en la línea 17025 se incrementa el número de hijos que han pasado los tests anteriores (`children`).

```
17021       /* El valor de pidarg determina qué hijo tomar */
17022       if (pidarg  > 0 && pidarg != rp->mp_pid) continue;
17023       if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;
17024
17025        children++;             /* Este hijo resulta apropiado */
```

Paso 2.2. A este punto sólo se llega si el proceso de la ranura actual se ajusta a los parámetros de la llamada. En la línea 17026 se comprueba si el proceso está en estado zombi (`HANGING`); si es así, como el padre está esperando por él, en la línea 17028 se borra de la tabla de procesos (`cleanup`). Luego, en la línea 17029 se indica que en la función `main` no se responda al padre (`dont_reply=TRUE`), puesto que esto se hace desde `cleanup`. Por último, en la línea 17030 se sale del `do_waitpid`.

```
17026       if (rp->mp_flags & HANGING) {
17027         /* Este hijo supera el test de pid y ha finalizado */
17028         cleanup(rp);    /* Este hijo ya finalizó */
17029         dont_reply = TRUE;
17030         return(OK);
17031       }
```

Paso 2.3. A este punto sólo se llega si el proceso de la ranura actual se ajusta a los parámetros de la llamada. En la línea 17032 se comprueba si el proceso está siendo depurado. En ese caso, en la línea 17034 se copia el número de la señale en el byte más significativo de la variable `res2`; en la línea 17035 se envía una respuesta con dicha información; en la línea 17036 se indica que en la función `main` no debe responderse al padre, puesto que ya se ha hecho; en la línea 17037 se resetea el número de las señales; y en la línea 17038 se sale de `do_waitpid`.

```
17032       if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
17033         /* Este supera el test de pid y está siendo depurado */
17034         res2 =  0177 | (rp->mp_sigstatus << 8);
17035         reply(who, rp->mp_pid, res2, NIL_PTR);
17036         dont_reply = TRUE;
17037         rp->mp_sigstatus = 0;
17038         return(OK);
17039       }
17040     }
17041   }
```

Paso 3. En la línea 17044 se comprueba si existen hijos que se ajustan a los parámetros de la llamada pero que no han terminado su ejecución. Si no es el caso, en la línea 17053 se retorna un error, puesto que el proceso invocador ha realizado un `wait` sin hijos que se ajusten a esos parámetros . En caso de que si se cumpla la condición de la línea 17044 y de que se haya usado la opción `WNOHANG`, en la línea 17046 se sale de `do_waitpid`, puesto que significa que el padre no quiere esperar. Por otro lado, si no se usó la opción `WNOHANG`, en la línea 17047 al proceso llamador se le añade el flag de espera (`WAITING`), se almacena en el campo `mp_wpid` el pid del proceso por el que espera (línea 17048), se indica a la función `main` que no envíe una respuesta al padre (línea 17049) y se sale de `do_waitpid` (línea 17050).

```
17042
17043   /* No ha finalizado ningún hijo. Se espera por uno, a menos que no exista */
17044   if (children > 0) {
17045     /* Al menos un hijo supera el test de pid, pero no ha finalizado */
17046     if (options & WNOHANG) return(0);    /* El padre no quiere esperar */
17047     mp->mp_flags |= WAITING;             /* El padre quiere esperar */
17048     mp->mp_wpid = (pid_t) pidarg;        /* Se almacena el pid para luego */
17049     dont_reply = TRUE;                   /* No se producirá una respuesta */
17050     return(OK);                          /* Se esperará para salir */
17051   } else {
17052     /* No hay ningún hijo que supere el test, y eso es un error */
17053     return(ECHILD);           /* El padre no tiene hijos que superen el test */
17054   }
17055 }
```

6 Función `cleanup ()`.

Esta subrutina, como se puede ver en el código anterior, es invocada desde `mm_exit` y desde `do_waitpid`; su objetivo principal es liberar una ranura de la tabla de procesos. Los pasos de ejecución que sigue esta sencilla función son:

1. Se devuelve al padre un mensaje de respuesta con el estado de salida y el pid del hijo.
2. Se actualiza el estado de espera del padre (se borra).
3. Se libera la ranura del proceso que terminó.

A continuación se comentan estos puntos:

Paso 1. En la variable `exitstatus` (línea 17068) se almacena el estado de salida del proceso e información adicional sobre las señales; el estado de salida del proceso se almacena en el byte más significativo (`child->mp_exitstatus << 8`), y el número de la señal que produjo su salida en el byte menos significativo (`child->mp_sigstatus & 0377`). Luego, en la línea 17072, se le envía al padre un mensaje con dicho estado salida (`exitstatus`) y el pid del proceso hijo (`child->mp_pid`).

```
17061 PRIVATE void cleanup(child)
17062 register struct mproc *child;   /* Indica el proceso que está finalizando */
17063 {
17064   /* Liquida la finalización de un proceso. El proceso ha finalizado o ha sido
17065    * eliminado por una señal, y su padre está esperando
17066    */
17067
17068   int exitstatus;
17069
17070   /* Despierta al padre */
17071   exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
17072   reply(child->mp_parent, child->mp_pid, exitstatus, NIL_PTR);
```

Paso 2. En la línea 17073 se borra el estado de espera del padre. Esta operación se realiza añadiendo la negación del flag `WAITING` en el campo `mp_flags` de la ranura del padre.

```
17073   mproc[child->mp_parent].mp_flags &= ~WAITING; /* El padre ya no espera */
```

Paso 3. En la línea 17076 se resetean los flags de la ranura a borrar. De esta forma, cuando se examine el flag `IN_USE` de este slot se observará que está vacío y que puede ser usado por otro proceso. Después, en la línea 17077, se decrementa en una unidad el número de procesos en uso (`procs_in_use`).

```
17074
17075   /* Libera la entrada de la tabla de procesos */
17076   child->mp_flags = 0;
17077   procs_in_use--;
17078 }
```

7 Código.

7.1 Definiciones y prototipos.

```
16800 /* Este archivo crea procesos (vía FORK) y los elimina (vía
16801  * EXIT/WAIT). Si un proceso se bifurca, se le asigna una nueva ranura en la tabla
16802  * 'mproc' y se hace una copia de la imagen de núcleo del padre para el hijo.
16803  * Luego se informa al kernel y a FS. Se quita un proceso de 'mproc'
16804  * cuando ocurren dos eventos: (1) salió o se mató por una señal, y
16805  * (2) el padre hizo un WAIT. Si el proceso sale primero, sigue ocupando su ranura
16806  * hata que el padre ejecuta WAIT.
16807  *
16808  * Los puntos de entrada a este archivo son:
16809  *   do_fork:    realizar llamada al sistema FORK
16810  *   do_mm_exit: realizar llamada al sistema EXIT (invocando mm_exit())
16811  *   mm_exit:    salir realmente
16812  *   do_wait:    realizar llamadas al sistema WAITPID o WAIT
16813  */
16814
16815
16816 #include "mm.h"
16817 #include <sys/wait.h>
16818 #include <minix/callnr.h>
16819 #include <signal.h>
16820 #include "mproc.h"
16821 #include "param.h"
16822
16823 #define LAST_FEW          2 /* últimas ranuras reservadas para el superusuario */
16824
16825 PRIVATE pid_t next_pid = INIT_PID+1;  /* siguiente pid por asignar */
16826
16827 FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child) );
```

7.2 fork()

```
16832 PUBLIC int do_fork ()
16833 {
16834   /* El proceso apuntado por 'mp' se ha bifurcado. Creamos un proceso hijo */
16835
16836   register struct mproc *rmp;  /* Puntero al padre */
16837   register struct mproc *rmc;  /* Puntero al hijo */
16838   int i, child_nr, t;
16839   phys_clicks prog_clicks, child_base = 0;
16840   phys_bytes prog_bytes, parent_abs, child_abs; /* Solo para INTEL */
16841
16842   /* Si las tablas de proceso se llenan en algún momento, se proporciona un
16843    * mensaje de error
16844    */
16845   rmp = mp;
16846   if (procs_in_use == NR_PROCS) return (EAGAIN);
16847   if (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0) return (EAGAIN);
16848
16849   /* Se determina cuanta memoria se reservará. Solo se necesita copiar datos y
16850    * pila, dado que el segmento de texto o está compartido o es de longitud cero
16851    */
16852   prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
16853   prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
16854   prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
16855   if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(EAGAIN);
16856
16857   /* Crea una copia de la imagen del padre para el hijo */
16858   child_abs = (phys_bytes) child_base << CLICK_SHIFT;
16859   parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
16860   i = sys_copy(ABS, 0, parent_abs, ABS, 0, child_abs, prog_bytes);
16861   if (i < 0) panic("do_fork can't copy", i);
16862
16863   /* Busca un ranura existente en 'mproc' para el proceso hijo */
16864   for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
16865     if ( (rmc->mp_flags & IN_USE) == 0) break;
16866
16867   /* Prepara el hijo y su mapa de memoria; copia la ranura del padre */
16868   child_nr = (int)(rmc - mproc); /* Número de ranura del hijo */
```

```
16869   procs_in_use++;
16870   *rmc = *rmp;                  /* Copia la ranura del proceso padre al hijo */
16871
16872   rmc->mp_parent = who;         /* Almacena el apuntador al padre */
16873   rmc->mp_flags &= ~TRACED;     /* El hijo no hereda el estado de traza */
16874   /* Un hijo con I&D separados mantiene el segmento de texto del padre.
16875    * Los segmentos de datos y pila deben referenciar la nueva copia
16876    */
16877   if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
16878   rmc->mp_seg[D].mem_phys = child_base;
16879   rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
16880                           (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
16881   rmc->mp_exitstatus = 0;
16882   rmc->mp_sigstatus = 0;
16883
16884   /* Encuentra un pid libre para el hijo y lo pone en la tabla */
16885   do {
16886     t = 0;              /* 't' = 0 implica un pid aún libre */
16887     next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PID + 1);
16888     for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
16889       if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
16890         t = 1;
16891         break;
16892       }
16893     rmc->mp_pid = next_pid; /* Asigna pid al hijo */
16894   } while (t);
16895
16896   /* Avisa al núcleo y sistema de ficheros del FORK efectuado con éxito */
16897   sys_fork(who, child_nr, rmc->mp_pid, child_base);
16898   tell_fs(FORK, who, child_nr, rmc->mp_pid);
16899
16900   /* Informa del mapa de memoria del hijo al núcleo */
16901   sys_newmap(child_nr, rmc->mp_seg);
16902
16903   /* Respuesta para que el hijo despierte */
16904   reply(child_nr, 0, 0, NIL_PTR);
16905   return(next_pid);             /* pid del hijo */
16906 }
```

7.3 exit ()

7.3.1 do_mm_exit ()

```
16912 PUBLIC int do_mm_exit ()
16913 {
16914   /* Ejecuta la llamada al sistema EXIT. La auténtica llamada es realizada por
16915    * mm_exit (), también invocada cuando un proceso muere por una señal
16916    */
16917
16918   mm_exit(mp, status);
16919   dont_reply = TRUE;      /* No responder al proceso recientemente terminado */
16920   return(OK);
16921 }
```

7.3.2 mm_exit ()

```
16927 PUBLIC void mm_exit(rmp, exit_status)
16928 register struct mproc *rmp;  /* Puntero al proceso a finalizar */
16929 int exit_status;             /* Código de salida del proceso (para el  padre) */
16930 {
16931   /* Ha finalizado un proceso. Hay que liberar las posesiones del mismo. Si el
16932    * padre está esperando, libera el resto, y si no, se cuelga.
16933    */
16934
16935   register int proc_nr;
16936   int parent_waiting, right_child;
16937   pid_t pidarg, procgrp;
16938   phys_clicks base, size, s;        /* base y size solo se usan en 68000 */
16939
16940   proc_nr = (int) (rmp - mproc);    /* Obtiene el número de ranura del proceso */
16941
```

```
16942   /* Recordar grupo de procesos de un líder de sesión */
16943   procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;
16944
16945   /* Si el proceso tiene un timer pendiente, lo elimina */
16946   if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);
16947
16948   /* Avisa al núcleo y al FS de que ya no se puede ejecutar el proceso */
16949   tell_fs(EXIT, proc_nr, 0, 0);  /* file system can free the proc slot */
16950   sys_xit(rmp->mp_parent, proc_nr, &base, &size);
16951
16952   /* Libera la memoria ocupada por el hijo */
16953   if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
16954     /* No hay otros procesos compartiendo texto, luego se elimina */
16955     free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
16956   }
16957   /* Libera los segmentos de datos y de pila */
16958   free_mem(rmp->mp_seg[D].mem_phys,
16959        rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
16960
16961   /* El slot del proceso solo puede liberarse si el padre ha hecho un WAIT */
16962   rmp->mp_exitstatus = (char) exit_status;
16963   pidarg = mproc[rmp->mp_parent].mp_wpid;       /* ¿Por quién se espera? */
16964   parent_waiting = mproc[rmp->mp_parent].mp_flags & WAITING;
16965   if (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp)
16966     right_child = TRUE;       /* El hijo cumple uno de los tres tests */
16967   else
16968     right_child = FALSE;      /* El hijo falla todos los tests */
16969   if (parent_waiting && right_child)
16970     cleanup(rmp);             /* Avisa al padre y libera el slot del hijo */
16971   else
16972     rmp->mp_flags |= HANGING; /* El padre no espera; se suspende el hijo */
16973
16974   /* Si el proceso tiene hijos, se desheredan (init los adoptará) */
16975   for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
16976     if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
16977       /* 'rmp' apunta a un hijo que será desheredado */
16978       rmp->mp_parent = INIT_PROC_NR;
16979       parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
16980       if (parent_waiting && (rmp->mp_flags & HANGING)) cleanup(rmp);
16981     }
16982   }
16983
16984   /* Envía una señal de suspensión al grupo del proceso si era líder de sesión */
16985   if (procgrp != 0) check_sig(-procgrp, SIGHUP);
16986 }
```

7.4 wait () y waitpid ()

```
16992 PUBLIC int do_waitpid()
16993 {
16994   /* Un proceso quiere esperar a que un hijo termine. Si al menos uno ya está
16995    * esperando, eliminarlo y finalizar la llamada. En otro caso, se espera
16996    * realmente por un hijo. Tanto wait () como waitpid () se llevan a cabo aquí
16997    */
16998
16999   register struct mproc *rp;
17000   int pidarg, options, children, res2;
17001
17002   /* Un proceso que invoca WAIT nunca recibe una respuesta por la vía normal del
17003    * reply () en el bucle principal (a menos que se haya activado WNOHANG o
17004    * no haya hijos apropiados). Si un hijo ya ha finalizado, cleanup ()
17005    * envía la respuesta para despertar al invocador
17006    */
17007
17008   /* Prepara algunas variables internas, según se haga wait () o waitpid () */
17009   pidarg  = (mm_call == WAIT ? -1 : pid);    /* Primer parámetro de waitpid () */
17010   options = (mm_call == WAIT ?  0 : sig_nr); /* Tercer parámetro de waitpid () */
17011   if (pidarg == 0) pidarg = -mp->mp_procgrp; /* pidarg < 0 ==> proc grp */
17012
17013   /* ¿Hay algún hijo esperando su finalización? Aquí, pidarg != 0:
17014    *   pidarg  >  0 implica que pidarg es el pid de un proceso por el que esperar
17015    *   pidarg == -1 implica esperar por cualquier hijo
17016    *    pidarg  < -1 implica esperar por cualquier hijo cuyo grupo sea -pidarg
17017    */
17018   children = 0;
```

```
17019   for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
17020     if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who) {
17021       /* El valor de pidarg determina qué hijo tomar */
17022       if (pidarg  > 0 && pidarg != rp->mp_pid) continue;
17023       if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;
17024
17025       children++;             /* Este hijo resulta apropiado */
17026       if (rp->mp_flags & HANGING) {
17027         /* Este hijo supera el test de pid y ha finalizado */
17028         cleanup(rp);    /* Este hijo ya finalizó */
17029         dont_reply = TRUE;
17030         return(OK);
17031       }
17032       if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
17033         /* Este supera el test de pid y está siendo depurado */
17034         res2 =  0177 | (rp->mp_sigstatus << 8);
17035         reply(who, rp->mp_pid, res2, NIL_PTR);
17036         dont_reply = TRUE;
17037         rp->mp_sigstatus = 0;
17038         return(OK);
17039       }
17040     }
17041   }
17042
17043   /* No ha finalizado ningún hijo. Se espera por uno, a menos que no exista */
17044    if (children > 0) {
17045     /* Al menos un hijo supera el test de pid, pero no ha finalizado */
17046     if (options & WNOHANG) return(0);    /* El padre no quiere esperar */
17047     mp->mp_flags |= WAITING;             /* El padre quiere esperar */
17048     mp->mp_wpid = (pid_t) pidarg;        /* Se almacena el pid para luego */
17049     dont_reply = TRUE;                   /* No se producirá una respuesta */
17050     return(OK);                          /* Se esperará para salir */
17051   } else {
17052     /* No hay ningún hijo que supere el test, y eso es un error */
17053     return(ECHILD);           /* El padre no tiene hijos que superen el test */
17054    }
17055 }
```

7.5 Función cleanup ()

```
17061 PRIVATE void cleanup(child)
17062 register struct mproc *child;   /* Indica el proceso que está finalizando */
17063 {
17064   /* Liquida la finalización de un proceso. El proceso ha finalizado o ha sido
17065    * eliminado por una señal, y su padre está esperando
17066    */
17067
17068   int exitstatus;
17069
17070   /* Despierta al padre */
17071   exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
17072   reply(child->mp_parent, child->mp_pid, exitstatus, NIL_PTR);
17073   mproc[child->mp_parent].mp_flags &= ~WAITING; /* El padre ya no espera */
17074
17075   /* Libera la entrada de la tabla de procesos */
17076   child->mp_flags = 0;
17077   procs_in_use--;
17078 }
```

8 Cuestiones.

1. Pasos de ejecución de la llamada al sistema `fork ()`. (Mirar puntos 3.2 y 3.3. Página 8)

2. ¿Qué significa que un proceso está en estado zombi? ¿Cuáles son sus características?

Si un proceso termina su ejecución y el padre no ha realizado un `wait` por él, entonces entra en un estado denominado *zombi*. Cuando un proceso se encuentra en estado zombi no puede ser planificado, se libera su zona de memoria y se cancela, si tiene, su temporizador de alarma; tan sólo se deja su entrada en la tabla de procesos. Cuando finalmente el padre realiza un `wait` por él, la ranura de la tabla de procesos se libera y se informa de esta circunstancia al sistema de archivos y al kernel.

3. ¿Qué sucede con los hijos de un proceso que efectúa un `exit ()`?

Si no se realizara ninguna acción con los hijos de un proceso que termina, éstos, cuando a su vez terminaran, quedarían indefinidamente en estado zombi, puesto que ya ningún padre esperará por ellos. Por evitar esta situación, la función `exit` convierte a todos los hijos del proceso que sale en hijos del proceso init. La razón de ser de esta política radica en que el proceso init entra en un bucle infinito en el que, entre otros, espera por la muerte de alguno de sus hijos; de esta forma los procesos huérfanos que han terminado se eliminan rápidamente. (Mirar puntos 4.1 y 4.2. Página 12)

4. Pasos de ejecución de la llamada al sistema `exit ()`. (Mirar puntos 4.2 y 4.3. Página 12)

5. Pasos de ejecución de la llamada al sistema `wait ()`. (Mirar puntos 5.2 y 5..3. Página 18)