Lección 18

Monoalgorítmez

18.1. Propósito

En esta Lección y en las dos siguientes estudiaremos modelos de *máquinas operativas* construidas sobre la máquina convencional Algorítmez.

Se puede observar una discontinuidad en las secuencias de cursos (y de textos) sobre ordenadores. Los dedicados a los sistemas operativos acostumbran a dar por sabidos algunos conceptos sobre el nivel de máquina convencional (registros, modos de direccionamiento, interrupciones, etc.) y sobre el de máquina simbólica (algún lenguaje de alto nivel), y rápidamente se introducen en temas «nobles», como la gestión de la memoria y los ficheros, los algoritmos de planificación, etc., despreciando los detalles de «bajo nivel». Este enfoque tiene una justificación: tales detalles están muy ligados a la UCP, y es imposible explicarlos sin recurrir a las particularidades de una determinada UCP. Pero el resultado es que el lector (o el alumno) se encuentra con dificultades para salvar la transición brusca de nivel de abstracción.

Estas Lecciones se han diseñado con la pretensión de facilitar esa transición. Como siempre, eludimos la dependencia de los detalles de los sistemas reales, recurriendo a modelos ficticios. Nuestra primera máquina operativa se llama «Monoalgorítmez» porque es el resultado de añadir a Algorítmez un *pequeño sistema operativo de monoprogramación*. Aquí veremos cómo los programas de aplicación, mediante llamadas al sistema (con la instrucción `BRK`), le piden a éste que se encargue de ciertas operaciones rutinarias. Veremos también el modelo estructural (componentes de software), la organización de los ficheros en los discos y un modelo procesal general. Cómo procede el sistema para manejar esas peticiones (estructuras de datos y modelos procesales de los componentes) será el objeto de la Lección siguiente.

La simplificación de los modelos es tal que algunas partes muy importantes de los sistemas operativos ni se mencionan. Así ocurre con la memoria virtual (que no puede implementarse sobre Algorítmez porque su espacio de direccionamiento sólo tiene 64 KB) y con los sistemas de ficheros jerarquizados (cuya implementación no presenta dificultades conceptuales importantes, y, sin embargo, es muy engorroso detallar). Nos vamos a centrar fundamentalmente en el uso y la implementación de las instrucciones virtuales para las comunicaciones con los periféricos y para la carga y ejecución de programas.

18.2. Modelo funcional

Algorítmez estará dotado de un teclado, una pantalla de texto, una impresora y dos unidades de disco (en realidad, «disquetes») de 1.440 KB, conectadas éstas a través de un controlador de ADM (Apartado 7.8). Cada disco tiene dos superficies de 1.440 sectores cada una, y cada sector contiene 512 bytes. Los sectores de un disco se identifican por un número entre 0 y 2.879.

Ficheros y directorio

Monoalgorítmez implementa sobre este hardware un sistema de ficheros *de un solo nivel* (Figura 17.2(a)). Puede haber hasta 256 ficheros en cada disco, con una longitud máxima de 133 KB cada uno. Desde un punto de vista funcional, los ficheros son, simplemente, *secuencias de bytes*. Para todo fichero abierto el sistema mantiene una variable, llamada **posición**, cuyo valor inicial es «0», o bien igual a la longitud (número de bytes) del fichero. Si se leen o se escriben n bytes en un fichero, su posición se incrementa en n.

El nombre de un fichero es una secuencia de uno a catorce caracteres precedida de «`A:`» o de «`B:`» para indicar el disco[1]. Es habitual el convenio de utilizar un punto como carácter «separador», pero para nuestro sistema el punto es un carácter como otro cualquiera.

Todo fichero tiene asociados unos **permisos de acceso** que se codifican mediante un dígito octal llamado «RWX», cuyo valor determina las operaciones permitidas: R (lectura), W (escritura) y X (ejecución). Así, RWX = 5 = B'101 significa que el fichero se puede leer y ejecutar, pero no se puede escribir en él.

En cada disco hay un **directorio** (Apartado 17.8), almacenado en un fichero que tiene como nombre «/» (`A:/` y `B:/`).

[1]Dar nombre explícitamente a los discos o discos es típico de MS-DOS. Salvo por este detalle, seguimos más bien el modelo funcional de UNIX: las letras mayúsculas y las minúsculas *no* son equivalentes, y, en principio, cualquier carácter es válido, salvo «nulo» (H'00) y «/». Este último es el nombre del directorio; en general, tiene también un significado en los nombres de trayectos (Apartado 17.8; véase también, para una mayor aproximación a UNIX, el Apartado 21.2). Algunos caracteres no deben usarse en el nombre del fichero, por tener un significado especial cuando este nombre forma parte de una orden. En nuestro caso, «espacio» (H'20) y (en Multialgorítmez) «&».

Ficheros especiales

Los nombres «`tty`», «`lp`», «`fd0`» y «`fd1`» están reservados para referirse directamente a los periféricos, que son *ficheros especiales*. El fichero `tty` representa tanto al teclado como a la pantalla: leer de él es leer del teclado, y escribir, escribir en la pantalla. La impresora se llama `lp` (fichero de sólo escritura), y `fd0` y `fd1` representan a los discos. `tty` y `lp` son *ficheros especiales de caracteres*: no tienen un tamaño definido, y la variable *posición* no tiene sentido en ellos (no podemos «ir a la posición 200» de un teclado; sólo cabe leer el último carácter tecleado, o esperar al siguiente). `fd0` y `fd1` son *ficheros especiales de bloques*. *En nuestro sistema un bloque es exactamente igual a un sector: 512 bytes*; se puede leer o escribir el bloque n ($0 \leq n \leq 2.879$) de `fd0` o de `fd1`. No obstante, salvo en algunos programas del sistema (los de arranque, los de mantenimiento y los propios del sistema de gestión de ficheros), lo normal será acceder a las informaciones almacenadas en los discos a través de la abstracción que ofrece el sistema de ficheros, que evita el trabajar con números de bloques.

Intérprete de órdenes

Una máquina operativa ofrece dos interfaces para su uso (Figura 17.1): la interfaz del usuario en general, definida por el lenguaje de órdenes, y la interfaz del programador, definida por las *llamadas al sistema* (Apartado 17.3).

El intérprete reconoce órdenes formadas por un *nombre* y, a veces, uno o varios *argumentos*. Por ejemplo:

- `A:` y `B:` sirven para designar al disco *activo*, o *por defecto*, aquél que el intérprete «sobreentiende» si no se antepone el nombre del disco al del fichero.
- `cp fich1 fich2` copia el contenido del fichero «`fich1`» en el fichero «`fich2`» (si éste existe, sustituye su contenido, y si no, lo crea).
- `lpr fich` imprime el fichero «`fich`».
- `ls` (o `ls A:`, o `ls B:`) presenta por la pantalla un listado de todos los ficheros existentes en el disco activo (o en el `A`, o en el `B`).
- `mv fich1 fich2` cambia el nombre de «`fich1`» a «`fich2`».
- `rm fich` borra el fichero «`fich`».
- `chmod p fich` cambia los permisos de «`fich`» ($0 \leq p \leq 7$).

A cada orden (salvo a la primera, véase el Ejercicio 10) le corresponde un programa ejecutable almacenado en un fichero con el nombre de esa orden. Lo único que hace el intérprete es cargar en la MP el fichero y pasar a su ejecución. Si tenemos un programa editor en un fichero de nombre «`WordGuay`», el intérprete reconocerá la orden «`WordGuay`», pidiéndole al sistema operativo que cargue y ejecute el código contenido en ese fichero (si el fichero no tiene permiso de ejecución, el sistema devolverá un error al intérprete, que a su vez informará al usuario). Lo que eventualmente siga al nombre de la orden, tras uno o más espacios, son argumentos para el programa.

Para introducir en un fichero una sucesión de caracteres escritos desde el teclado se puede dar la orden `cp tty fich`[2]. Inversamente, `cp fich tty` presenta en la pantalla los caracteres de `fich` tal y como están almacenados sucesivamente en el fichero. `lpr fich` no es necesariamente equivalente a `cp fich lp` (Ejercicio 14 y Apartado 21.4).

[2] De algún modo hay que decirle al sistema que ya hemos terminado y que no queremos introducir más caracteres; esto se indica con un carácter especial,«`^d`», que significa «lo que se genera pulsando las teclas "Ctrl" y "d" conjuntamente» (ASCII H'04).

Macroinstrucciones y RIP

Implementaremos las llamadas al sistema mediante macroinstrucciones predefinidas en el lenguaje ensamblador (Apartado 17.4). El ensamblador (traductor) genera para cada macroinstrucción una secuencia de instrucciones que incluye «`BRK`»; la ejecución de ésta produce, como sabemos, una interrupción cuya rutina de servicio empieza en la dirección indicada por el vector de interrupción, almacenado en la dirección D'260 (Tabla 6.10). A esta rutina, cuyo detalle estudiaremos en la Lección siguiente, la llamaremos **rutina de interrupciones de programa**. Como ya es habitual, abreviaremos el nombre con su acrónimo, aunque en este caso resulte algo escatológico: «**RIP**».

Cada llamada (cada macro) corresponde a una operación (cargar un programa, escribir en un fichero, etc.) identificada por un número que se pasa al sistema por R0. La mayoría de las llamadas van acompañadas de *parámetros* (nombre del programa a cargar, dirección de los datos, número de bytes, etc.), que se pasan al sistema por otros registros de la UCP.

En la Tabla 18.1 (página siguiente) hay un resumen de las llamadas al sistema. «*df*» significa «**descriptor del fichero**». Es un número positivo pequeño (menor que 24) que el sistema devuelve en R13 al atender a las llamadas `CREATE` y `OPEN`, y que el programa de usuario deberá guardar en un byte reservado al efecto. Una vez abierto un fichero, todas las referencias en las llamadas para operar sobre él se hacen por medio de ese descriptor. Para algunos parámetros (*dnom*, *dir*, *dav*, *darg*, *dinfo*) se pasa su *dirección*, y para otros (*p*, *m*, *df*, *nb*, *despl*, *k*), su *valor*.

El sistema puede devolver un código de error mediante un número negativo en el byte menos significativo de R13 (Apartado 18.7 y Tabla 18.2). Si ha podido procesar la llamada sin errores, en algunos casos devuelve un resultado (positivo) en R13 y/o R12 (el descriptor del fichero, el número de bytes transferidos, etc.), y, en otros, (R13) = 0.

	Nombre	Parámetros	Acción
0	`CREATE`	*dnom, p*	Crea un fichero cuyo nombre está apuntado por *dnom*, con permisos *p* ($0 \leq p \leq 7$) y lo abre en modo escritura. Devuelve el descriptor en R13.
1	`DELETE`	*dnom*	Borra del directorio el nombre del fichero.
2	`OPEN`	*dnom, m*	Abre el fichero y devuelve el descriptor en R13 ($0 \leq m \leq 4$).
3	`CLOSE`	*df*	Cierra el fichero de descriptor *df*.
4	`WRITE_A`	*df, dir, dav*	Escribe n bytes almacenados en la MP (en el programa de usuario) a partir de la dirección *dir* en el fichero de descriptor *df*. n depende del periférico (véase el texto). El sistema pone inicialmente «0» en el byte de dirección *dav*, y «1» (o «–1», si ha habido error) cuando la transferencia ha terminado.
5	`READ_A`	*df, dir, dav*	Lee n bytes del fichero de descriptor *df* y los almacena en la MP a partir de la dirección *dir*.
6	`WAIT_A`	*dav*	*dav* es la dirección de un byte («aviso») en el programa de usuario. Mientras su contenido sea «0» el programa queda bloqueado.
7	`WRITE`	*df, dir, nb*	Escribe, en el fichero de descriptor *df*, *nb* bytes almacenados a partir de la dirección *dir*. Devuelve en R12 el número de bytes realmente escritos
8	`READ`	*df, dir, nb*	Lee *nb* bytes del fichero de descriptor *df* y los almacena a partir de la dirección *dir*. Devuelve en R12 el número de bytes realmente leídos.
9	`POS`	*df, despl, k*	Cambia la posición del fichero de descriptor *df* (Apartado 18.5). Devuelve el valor resultante de la posición en R12 y R13.
10	`CHMOD`	*dnom, p*	Pone los permisos *p* al fichero.
11	`STAT`	*dnom, dinfo*	Pone en *dinfo* informaciones sobre el fichero (Apartado 18.5).
12	`LD_EX`	*dnom, darg*	Carga en la MP el programa almacenado en el fichero y lo ejecuta. En *darg* puede haber informaciones (argumentos) para el programa.
13	`LD_OV`	*dnom*	Carga el fichero, pero no pasa a ejecutarlo.
14	`EXIT`		Da por terminada la ejecución del programa.

Tabla 18.1 Llamadas al sistema.

18.3. Ocho llamadas

Empecemos describiendo un subconjunto de llamadas que nos permitirán, en el Apartado siguiente, desarrollar algunos ejemplos.

Creación y borrado de ficheros

`CREATE` sirve para *crear* un fichero *nuevo*, pero si el fichero nombrado ya existía, entonces «borra» su contenido previo (trunca su longitud a 0). Además, abre el fichero en modo de escritura (independientemente de los permisos, que se dan como segundo parámetro) y el programa de usuario puede recoger el descriptor en R13.

La expansión de `CREATE(#DNOMBRE,PERM)` (donde `DNOMBRE` es una etiqueta que identifica a una zona de bytes en la que están los caracteres del nombre, y `PERM` es la etiqueta de un byte donde están almacenados los permisos codificados como un número entre 0 y 7), es:

```
PUSH .0
PUSH .2
PUSH .6
CLR  .0
LD   .2,#DNOMBRE; dirección del nombre
LD.B .6,PERM
BRK
POP  .6
POP  .2
POP  .0
```

Lo que hace esta secuencia de instrucciones es transmitir al sistema, a través de los registros R0, R2 y R6, las informaciones sobre la operación a realizar: (R0) = 0 indica que la operación es «crear», R2 contiene la *dirección* donde se encuentra la cadena de caracteres que forman el nombre (el sistema leerá esos caracteres hasta llegar al carácter «nulo», H'00 en ASCII), y R6 contiene el dígito octal de permisos. También puede transmitirse este dígito con direccionamiento absoluto (poniendo «/PERM» en la macro), o inmediato (poniendo «#7», por ejemplo).

Las `PUSH` del principio y las `POP` del final salvan y restauran los registros para no perturbar al programa de usuario: la idea es que el programador haga uso de la macro sin preocuparse de los detalles de su expansión, y la llamada altera los contenidos de R0, R2 y R6. El sistema, en el tratamiento de la operación (que estudiaremos en la Lección siguiente), podrá hacer uso de otros, cuyos contenidos también deberá respetar.

`DELETE` *borra* del directorio el nombre de un fichero. La expansión de `DELETE(#DNOMBRE)` es:

```
PUSH .0
PUSH .2
LD.B .0,#1
LD   .2,#DNOMBRE
BRK
POP  .2
POP  .0
```

Apertura y cierre de ficheros

Aunque un fichero exista ya en un disco, para operar con él (escribir o leer) es preciso *abrirlo* con la macro `OPEN`. Es igual que `CREATE`, salvo que:

- En su expansión, pone el número de llamada, 2, en R0.
- El segundo parámetro es el modo en que se abre el fichero: 0 para lectura, 1 para escritura, 2 para lectura y escritura, 3 para escritura desde el final y 4 para lectura y escritura desde el final. Con los dos últimos el valor inicial de *posición* no es 0, sino la longitud del fichero.
- Si el nombre del fichero existe en el directorio del disco, lo abre en el modo indicado, que tiene que ser compatible con los permisos del fichero. Por ejemplo, el modo 2 no puede usarse si el dígito de permisos es 4 = B'100 (fichero de sólo lectura). En este caso el sistema no hace la operación y devuelve el código del error como un número negativo en R13.
- Si el nombre no existe, el sistema genera otro número de error y lo pone en R13.

Cuando el programa haya terminado de operar sobre el fichero, y antes de devolver control al sistema, deberá *cerrarlo* con `CLOSE(DF)` (donde `DF` es la etiqueta del byte en el que el programa ha guardado el descriptor que el sistema le devolvió en R13 al abrir el fichero con `CREATE` o con `OPEN`). Su expansión es:

```
PUSH .0
PUSH .3
LD.B .0, #3
LD.B .3, DF
BRK
POP  .3
POP  .0
```

(Siempre, por R0 se transmite el número de la llamada, y por R3 el descriptor del fichero).

En nuestro sistema, el único fichero que no necesita abrirse (ni tiene que cerrarse) es `tty`, que se puede utilizar para lectura (teclado) y para escritura (pantalla). Su descriptor es «0», y puede escribirse directamente en lugar de «DF» en las macros que veremos a continuación.

Escritura y lectura

Para salida de datos hacia un fichero cuyo descriptor está en `DF` se puede utilizar la macro `WRITE_A(DF,#ZONA,#AV)`. `ZONA` es la etiqueta del programa de usuario que identifica a la zona donde se encuentran los bytes a escribir. El número de bytes depende del tipo de fichero. En el caso de los ficheros ordinarios (ficheros en disco), o de los ficheros especiales `fd0` y `fd1`, la transferencia es siempre de un bloque completo (512 bytes). Por tanto, el programa de usuario deberá contener la línea «`ZONA RES.B 512`» (o equivalente). El bloque del disco en el que se escribe es aquél en el que se encuentra el byte indicado por la posición actual del fichero. Así, si $0 \leq posición < 512$, se

escribe en el primero de los bloques asignados al fichero, y la posición queda con el valor 512.

Si se trata de `tty` o `lp`, el número de bytes es variable: la operación termina cuando aparece codificado el carácter de «retorno» (ASCII H'0D) o el «nulo» (ASCII H'00). Por ejemplo:

```
ZONA DATA "mensaje",H'0D.
```

«AV» es la etiqueta de un byte en el programa de usuario que sirve para que éste y el sistema se comuniquen: cuando se ejecuta `WRITE_A`, el sistema pone en ese byte el valor «0», y lo cambia a «1» cuando termina la transferencia normalmente, o a «–1» si termina con error.

La expansión de `WRITE_A` utiliza los registros R0, R3, R4 y R5 para transmitir el *número* de la llamada (4), el *descriptor* y las *direcciones* del programa de usuario donde se encuentran los datos y el byte de aviso.

Puede ser interesante ver con un poco más de detalle la transmisión de informaciones desde el programa de usuario al sistema. Imaginemos un programa que contiene:

```
      ORG    0
D     RES.B  1
A     RES.B  1
Z     RES    256; (512 bytes)
      ---
ESC   WRITE_A(/D, #Z, #A)
      ---
```

En tiempo de traducción, y suponiendo que la dirección correspondiente a la etiqueta `ESC` es, por ejemplo, D'600, el ensamblador genera el código binario que muestra el Programa 18.1. Además, pone en el diccionario de reubicación las direcciones 613, 617 y 621.

En tiempo de carga, y suponiendo que el sistema operativo decide cargar el programa a partir de la dirección D'7.000, el cargador reubicador suma 7.000 a los contenidos de [613], [617] y [621]. Es decir, en las *palabras* de direcciones absolutas 7.613, 7.617 y 7.621 introduce los valores 7.000, 7.002 y 7.001.

En tiempo de ejecución, R0 se carga con el valor 4, R3 con el contenido del byte de direc-

```
 dir.       contenido
(dec.)     hex (simbólico)
------------------------------
[600]      4A  (PUSH .0)
[601]      00
[602]      4A  (PUSH .3)
[603]      03
[604]      4A  (PUSH .4)
[605]      04
[606]      4A  (PUSH .5)
[607]      05
[608]      8B  (LD.B .0,#4)
[609]      F0
[610]      04
[611]      AB  (LD.B .3,/D)
[612]      F3
[613]      00
[614]      00
[615]      8A  (LD   .4,#Z)
[616]      F4
[617]      02
[618]      00
[619]      8A  (LD   .5,#A)
[620]      F5
[621]      01
[622]      00
[623]      04  (BRK)
[624]      4B  (POP  .5)
[625]      05
[626]      4B  (POP  .4)
[627]      04
[628]      4B  (POP  .3)
[629]      03
[630]      4B  (POP  .0)
[631]      00
```

Programa 18.1. Expansión de `WRITE_A(/D,#Z,#A)`.

ción 7.000, R4 con el valor 7.002 y R5 con el valor 7.001. `BRK` hace entrar en juego a la RIP, y ésta, teniendo en cuenta el contenido del registro R0 (como veremos en la siguiente Lección), llama al sistema de gestión de ficheros para realizar la operación solicitada. Las informaciones que éste necesita las encuentra en R3, R4 y R5, que contienen, respectivamente, el descriptor y las direcciones absolutas de la zona de datos y del byte de aviso.

La llamada de lectura es similar: la expansión de `READ_A(DF,#ZONA,#AV)` sólo difiere de la de `WRITE_A(DF,#ZONA,#AV)` en que se introduce «5» en R0.

Espera y salida

La llamada `WAIT_A(#AV)` permite hacer que un programa espere a la terminación de una transferencia previamente iniciada con `READ_A` o con `WRITE_A`. Su expansión es:

```
PUSH .0
PUSH .5
LD.B .0, #6
LD   .5, #AV
BRK
POP  .5
POP  .0
```

La RIP, al encontrarse con (R0) = 6, sabe que se trata de `WAIT_A`. Consulta el byte etiquetado `AV` en el programa, y si contiene «0» es que aún no ha terminado la transferencia, por lo que no devuelve control al programa hasta que tal cosa ocurre; si, por el contrario, `AV` contiene «1» o «–1» es que la transferencia ya había terminado antes de llegar a la ejecución de esta macro, y vuelve inmediatamente al programa.

Cuando existe un sistema operativo, los programas de aplicación no pueden contener las instrucciones `HALT` ni `WAIT`[3] (son *instrucciones privilegiadas*). En su lugar, utilizan la macro `EXIT`, que no tiene parámetros, y se expande así:

```
LD.B .0, #14
BRK
```

En este caso no hace falta salvar R0: la ejecución del programa ha terminado.

18.4. Ejemplos de uso de las macros

Para ver cómo se simplifica la programación con el uso de llamadas (y, naturalmente, de un sistema operativo que toma a su cargo las operaciones oportunas) vamos a codificar de nuevo algunos ejemplos desarrollados anteriormente.

Comencemos con el programa del Apartado 7.4 que escribía en la pantalla los mensajes «Estoy con cálculo 1» y «Estoy con cálculo 2» mientras realizaba ciertos cálculos (Programa 7.4). Si el lector compara aquel programa con el Programa 18.2 observará que:

[3]No se confunda la *macro* `WAIT_A` con la *instrucción* (de la máquina convencional) `WAIT`.

```
      ---
MENS1 DATA.B "Estoy con cálculo 1",H'0D
MENS2 DATA.B "Estoy con cálculo 2",H'0D
AV    RES.B 1
      WRITE_A(#0,#MENS1,#AV)
      ---
      ---   ;  (cálculo 1)
      WAIT_A(#AV)
      WRITE_A(#0,#MENS2,#AV)
      ---
      ---   ;  (cálculo 2)
      WAIT_A(#AV)
      EXIT
      ---
```

Programa 18.2. Llamadas al sistema para escribir mensajes por la pantalla.

- El programador no tiene ya que preocuparse de dar valores iniciales a la rutina de servicio de interrupciones de la pantalla; es el sistema quien lo hará al interpretar las macros.
- Tampoco tiene que (ni puede) utilizar instrucciones de entrada/salida ni manejar el sistema de interrupciones.

Como segundo ejemplo, tomemos el de los Apartados 5.9 y 7.8. Se trataba de leer diez bloques seguidos de un disco, realizando algún procesamiento con cada uno de ellos. Sabemos que la máquina virtual (la definida por las llamadas y los nombres de ficheros) permite, en caso necesario, acceder a bloques de los discos por su número absoluto (mediante los ficheros especiales `fd0` y `fd1`). Pero lo normal es que esos bloques pertenezcan a un fichero previamente creado (y en el que están guardados los datos a procesar), de manera que el programador no tenga que preocuparse de la localización física de los bloques (en nuestro caso, igual a sectores) en el disco. Suponiendo que el nombre dado al fichero es «`abc.txt`» y que está


```
MODULE  LEE10BL
FROM    PR  IMPORT  PROCESA
EXPORT  ZONA
   DIEZ   EQU       10
   NFICH  DATA.B    "B:abc.txt",0
   ZONA   RES       256
   AVISO  RES.B     1
   DFICH  RES.B     1
   PROG   LD        .0,#DIEZ
          OPEN(#NFICH,#0)
          ST.B      .13,DFICH
   BUCLE  READ_A(DFICH,#ZONA,#AVISO)
          WAIT_A(#AVISO)
          CALL      /PROCESA
          SUB       .0,#1
          BZ        FIN
          BR        BUCLE
   FIN    CLOSE(DFICH)
          EXIT
         END       PROG
```

Programa 18.3. Llamadas al sistema para leer diez bloques de un fichero.

en el disco `B`, la solución se encuentra en el Programa 18.3.

El «0» tras los caracteres del nombre en `NFICH` es la codificación del carácter «nulo», indicador del fin de la cadena.

La variable interna *posición* asociada a cada fichero se pone a cero con `OPEN`. Cuando ha terminado la transferencia de los primeros 512 bytes (primer bloque) iniciada con `READ_A` esta variable tiene el valor 512, de modo que cuando empieza la segunda se lee el segundo bloque asignado al fichero, y así sucesivamente.

R0 hace de contador del número de bloques que faltan por leer (de 10 a 1). Como ya sabemos (Apartado 17.8), los bloques que forman el fichero no tienen por qué estar físicamente seguidos en el disco (y normalmente no lo están). Pero desde el punto de vista del usuario (usuario del sistema, es decir, el que escribe este programa) todo el fichero puede considerarse como una sucesión lineal de bytes (y, por tanto, de bloques).

El Programa 18.3 tiene exactamente el mismo inconveniente que presentaba el sugerido por el seudocódigo del Programa 5.1: no aprovecha la posibilidad de simultanear la lectura con el procesamiento. Y para solventarlo podemos hacer lo mismo que allí: utilizar dos zonas. Adaptando el Programa 5.2 (ahora no hay que preocuparse de dar valores iniciales a los registros del controlador de ADM, lo hará el sistema), y suponiendo que el subprograma `PROCESA` recibe como argumento de entrada la dirección de la zona a procesar por R2 y que no utiliza R0 ni R1, resulta el Programa 18.4.

```
MODULE  LEE10BL
FROM    PR  IMPORT  PROCESA
   DIEZ   EQU       10
   NFICH  DATA.B    "B:FICH",0
   ZONA0  RES.B     512
   ZONA1  RES.B     512
   AVISO  RES.B     1
   DFICH  RES.B     1
   PROG   LD.B      .0,#DIEZ
          CLR       .1
          OPEN(#NFICH,#0)
          ST.B      .13,DFICH
          READ_A(DFICH,#ZONA0,#AVISO)
   BUCLE  WAIT_A(#AVISO)
          SUB.B     .0,#1
          BZ        FIN
          CMP.B     .1,#0
          BZ        LECZ1
          READ_A(DFICH,#ZONA0,#AVISO)
          CLR       .1
          LD        .2,#ZONA1
          CALL      /PROCESA
          BR        BUCLE
   LECZ1  READ_A(DFICH,#ZONA1,#AVISO)
          LD.B      .1,#1
          LD        .2,#ZONA0
          CALL      /PROCESA
          BR        BUCLE
   FIN    LD        .2,#ZONA1
          CALL      /PROCESA
          CLOSE(DFICH)
          EXIT
         END       PROG
```

Programa 18.4. Lectura y procesamiento de diez bloques de un fichero utilizando dos zonas alternantes.

Vemos que el manejo de las zonas complica notablemente el trabajo del programador de aplicaciones. Parece razonable que el sistema se haga cargo también de esas manipulaciones. Además, el requisito de tener que acceder siempre a un bloque completo puede hacerse transparente al programa de usuario, de manera que éste pueda pedir la lectura o escritura de un número arbitrario de bytes. Veamos cómo.

18.5. Otras llamadas al sistema de ficheros

Escritura con dos zonas

`WRITE(DF,#DIR,NB)` tiene como parámetros el descriptor del fichero, la dirección de la zona donde se encuentran los datos a escribir (como en `WRITE_A`) y ahora, además, el número de bytes a escribir, que se encuentra en el byte de dirección `NB`. Su expansión es:[4]

```
PUSH .0          POP  .6
PUSH .3          POP  .4
PUSH .4          POP  .3
PUSH .6          POP  .0
LD.B .0, #7
LD.B .3, DF
LD   .4, #DIR
LD   .6, NB
BRK
```

Para entender su función hemos de adelantar algunos detalles procesales (que serán objeto de la Lección siguiente). El sistema maneja dos zonas de la MP (Figura 18.1). Una, a la que llamaremos **zona de usuario**, es la declarada en el programa, y puede tener cualquier longitud (siempre que no exceda 32 KB); la dirección de su primer byte se pasa como parámetro de la macro por R4, y la longitud (número de bytes), por R6. La otra es una **zona de entrada/salida** interna (transparente al usuario) asociada al periférico. Cada uno de los discos dispone de una zona de entrada/salida de un bloque (512 bytes). Las zonas para el teclado, la pantalla y la impresora son de ochenta bytes.

[4] A partir de ahora, las expansiones largas las presentamos en dos columnas: las instrucciones de la segunda columna irían detrás de las de la primera.

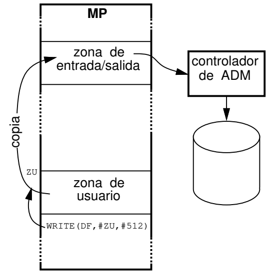


Figura 18.1. Escritura en un fichero a través de la zona de entrada/salida.

Cuando atiende a una llamada `WRITE`, el sistema empieza por mirar si se está procediendo a una transferencia iniciada anteriormente entre la zona de entrada/salida y el controlador del periférico; si es así, espera hasta que concluya. Luego, copia el contenido de la zona de usuario a la zona de entrada/salida e inicia la transferencia de ésta al controlador. Si `NB` no es mayor que el tamaño de la zona de entrada/salida, el sistema permite que el programa prosiga su ejecución *concurrentemente* con la operación de acceso directo a memoria (si se trata de un disco, como en la Figura 18.1) o con las interrupciones (si se trata de un periférico de caracteres). De este modo, el programa puede ir introduciendo nuevos contenidos en su zona de usuario para la siguiente operación de escritura mientras se van escribiendo los copiados en la zona de entrada/salida.

En el caso de que `NB` sea mayor que el tamaño de la zona de entrada/salida, el sistema se encarga de copiar sucesivamente trozos de la zona de usuario a la de entrada/salida, y no permite que el programa siga su ejecución hasta que se haya copiado el último trozo. (Si este último trozo es menor que 512 bytes el sistema tiene que hacer alguna cosa más, como veremos en la Lección siguiente).

Si se trata de un fichero en disco, la escritura se inicia a partir del byte señalado por el valor de la variable *posición*, y este valor se incrementa en el número de bytes escritos. Si se alcanza el límite de

longitud establecido para los ficheros (133 KB), la escritura se detiene. Si se trata de `tty` (que, tratándose de `WRITE`, identifica a la pantalla), la zona de entrada/salida tiene ochenta bytes, y la transferencia es mediante interrupciones y se detiene si aparece el carácter de retorno (H'0D) o el nulo (H'00). En cualquier caso, el sistema devuelve por R12 el número de bytes realmente escritos.

Lectura con dos zonas

La macro `READ(DF,#ZONA,NB)` tiene la misma expansión que `WRITE`, pero pone «8» en R0. Devuelve, también en R12, el número de bytes realmente leídos (que puede ser menor que `NB` si se alcanza el final del fichero, o si se llega a H'0D en el caso de `tty`).

Puede esperarse que el sistema opere de manera similar a la explicada para la macro `WRITE`. Efectivamente, tras esperar a que haya concluido una posible transferencia anterior, copia ahora de la zona de entrada/salida a la de usuario, inicia (si es necesario) la lectura sobre la zona de entrada/salida y devuelve control al programa. Ahora bien, esto supone que el sistema es «adivino»: ha leído un bloque en la zona de entrada/salida y al llegar `READ` transfiere `NB` bytes de ese bloque a la zona de usuario (y comienza, en su caso, la lectura del siguiente). La forma de «adivinar» del sistema es muy sencilla: si en la anterior `READ` el último byte leído era el último de un bloque entonces el sistema inició la lectura del siguiente bloque asignado a ese fichero (esperando que la siguiente `READ` pida ese bloque siguiente). Pero si los bytes leídos en una `READ` son, por ejemplo, los cinco primeros de un bloque, entonces cabe esperar que la siguiente `READ` trate de leer del mismo bloque, y el sistema no inicia nueva lectura. En cualquier caso, cuando se encuentra con `READ` comprueba primero si ha «adivinado» bien, en cuyo caso los `NB` pedidos ya están en la zona de entrada/salida; los copia en la de usuario y decide si tiene que leer otro bloque o dejar la zona de entrada/salida como está.

En resumen, la zona de entrada/salida contiene siempre un bloque leído del disco, y el sistema mantiene en una palabra de la MP el número de ese bloque. Cuando aparece `READ` el sistema compara el número de bloque que hay que leer con el que está en esa palabra. Si son iguales hace la transferencia a la zona de usuario, y si no lo son es porque el «algoritmo de adivinación» ha fallado; en este caso inicia la lectura del bloque correcto y espera a la terminación de esta lectura para copiar a la zona de usuario.

Además, atribuiremos a la macro `OPEN` la función de copiar el primer bloque del fichero en la zona de entrada/salida asignada a ese disco (salvo si el fichero está vacío, o si se abre en el modo 3 o el 4). Así, cuando se llega por primera vez a la macro `READ`, la transferencia del primer bloque del fichero a la zona de entrada/salida ya está realizada, o, por lo menos, está en curso (el «algoritmo de adivinación» supone que se empezará leyendo del primer bloque).

La maniobra descrita se llama **anticipación**, y permite aumentar la velocidad del sistema de dos modos: por una parte, reduciendo los accesos al disco (cuando se hacen lecturas sucesivas de bytes que están en el mismo bloque copiado en la zona de entrada/salida); por otra, anticipando la lectura de un bloque a su petición por el programa (y haciendo esa lectura mediante ADM mientras el programa realiza otras operaciones).

Unas observaciones sobre este modelo procesal simplificado del tratamiento de `READ`:

- **Lectura anticipada**: Para que siempre se hiciese lectura anticipada sería necesario que hubiese al menos una segunda zona de entrada/salida (o, mejor, una *memoria oculta de disco*, Apartado 15.7). Con el modelo que hemos esbozado, la anticipación sólo es posible cuando la lectura anterior ha terminado en el último byte de un bloque (dejando *posición* apuntando al primer byte del siguiente bloque).

- **Sincronización**: A diferencia de lo que ocurre con `READ_A` y `WRITE_A`, con `READ` y `WRITE` es el sistema y no el programa de usuario el que se encarga de sincronizar la operación de entrada/salida con la ejecución del programa, es decir, de controlar si ha terminado o no la transferencia anterior, y, en su caso, esperar. Otra observación es que el sistema opera sobre las dos zonas (la de entrada/salida y la de usuario) según el principio de que las funciones de las zonas no se van alternando, sino que cuando una está llena se copia en la otra (Apartado 5.9). Es necesario proceder así (aunque la copia de una zona a otra suponga una pérdida de tiempo) para mantener la zona de entrada/salida transparente a los programas de aplicaciones.

El programa para el último ejemplo desarrollado sería el mismo escrito antes para una sola zona (Programa 18.3), salvo que la macro sería `READ(DFICH,#ZONA,#512)` (no hace falta un byte en el programa de usuario para sincronizar con el fin de la transferencia), y que no pondríamos `WAIT_A(#AVISO)` (es el sistema el que se encarga de la sincronización).

• **Lectura del teclado**: El caso del teclado es distinto. Los caracteres no están previamente almacenados en ningún sitio. La apertura de `tty` (que el sistema realiza automáticamente durante el arranque) hace que las interrupciones del teclado queden permitidas. Cada vez que se pulsa una tecla la rutina de servicio introduce el carácter en la zona de entrada/salida y llama al gestor para que haga «eco» en la pantalla. De este modo, el usuario puede ir escribiendo caracteres aun cuando el programa que está en ejecución todavía no los haya pedido con `READ`. A esto se le llama **tecleo anticipado**. El tratamiento de `READ` consistirá en ver si el usuario ha pulsado la tecla de «retorno» (ASCII H'0D), o si la zona de entrada/salida está llena; si es así, se copia la zona de entrada/salida a la de usuario y se «limpia» la primera (poniendo, por ejemplo, H'00 en los ochenta bytes), y si no, se espera. Obsérvese que el gestor del teclado estará diseñado de modo que la operación `READ` se dé por terminada al pulsar «retorno», aun cuando no se hayan leído los `NB` bytes (normalmente, ochenta) especificados en la macro.

Cambio de posición, cambio de modo y petición de información

Como ya hemos dicho, para cada uno de los ficheros abiertos existe una variable, *posición*, que empieza con el valor 0 (al ejecutarse `OPEN` en modo 0, 1 o 2, o `CREATE`) o con un valor igual a la longitud del fichero (en el caso de `OPEN` en modo 3 o 4) y se incrementa cada vez que se ejecuta una operación de lectura o escritura en un número igual al de bytes transferidos[5]. Esto permite acceder *secuencialmente* a las informaciones del fichero. Permitir el *acceso aleatorio* es muy fácil dotando a la máquina operativa de una nueva instrucción virtual que modifique el valor de *posición*. Esta llamada se implementa con la macro `POS(DF,DESPL,K)`, cuya expansión es:

```
PUSH .0          POP  .7
PUSH .3          POP  .6
PUSH .6          POP  .3
PUSH .7          POP  .0
LD.B .0, #9
LD.B .3, DF
LD   .6, DESPL
LD   .7, K
BRK
```

`DESPL` contiene una cantidad a sumar, o *desplazamiento* ($0 \leq d \leq 65.535$), y `K` contiene un número k que tiene que ser 0, 1 u otro valor. Si el valor que tenía *posición* era p, y la longitud del fichero era l, el nuevo valor de la posición es:

- $np = d$, si $k = 0$.
- $np = p + d$, si $k = 1$.
- $np = l + d$, si $k \neq 0$ y $k \neq 1$.

El sistema devuelve el nuevo valor de *posición*, np, en R12 (los dieciséis bits menos significativos) y R13 (los dos más significativos). Obsérvese que el fichero puede «crecer» sin haberse escrito nada en él. Pero nunca puede superar el límite de 133 KB. El valor máximo resultante de la posición

[5] Excepto para los ficheros especiales `tty` y `lp`, en los que el acceso aleatorio no tiene sentido, y la variable *posición* no tiene valor definido.

es, pues, 133×1.024 – 1 = H'21377, que se expresa con dieciocho bits.

`CHMOD(#DNOM,P)` pone los permisos indicados por `P` en el fichero cuyo nombre se encuentra a partir de la dirección `DNOM`. Su expansión será la misma de `CREATE` y `OPEN` (salvo, naturalmente, que en R0 se pone «10»).

`STAT(#DNOM,#DINFO)` permite obtener informaciones del fichero: tipo (ordinario, especial, directorio), permisos (RWX), longitud, y día y hora de la última modificación. Estas informaciones están guardadas en los diez primeros bytes de una estructura de datos asociada al fichero, llamada «inodo», que estudiaremos un poco más adelante (Figura 18.4). `DINFO` debe ser la etiqueta de una zona de diez bytes, en los que el sistema devuelve esas informaciones. La expansión de `STAT` es similar a la de `LD_EX`, que veremos enseguida (ambas transmiten como parámetros dos direcciones).

18.6. Carga y ejecución

La llamada `LD_EX(#DPROG,#DARG)` pide que detrás del programa en el que está incluida (programa *padre*) se cargue otro (programa *hijo*), almacenado en un fichero cuyo nombre está en `DPROG`, y se ejecute. De hecho, esto es lo que hace el intérprete de órdenes: lee una orden y, suponiendo que se trata del nombre de un fichero en el que está guardado un programa ejecutable, pide su carga y ejecución (lo veremos en el Apartado 18.9). `DARG` es una dirección a partir de la cual pueden ponerse argumentos para el programa cargado.

En este caso es necesario guardar en la pila (y recuperar luego) los contenidos de *todos* los registros (salvo R14 = PP y R15 = CP, a los que el sistema devolverá los valores que tenían cuando el hijo ejecute `EXIT`, y salvo R13, por el que el sistema devuelve, en su caso, el número de error), y no sólo los que se utilicen en la expansión de la macro. En efecto, es imposible saber, en general, qué registros utilizará el programa hijo, y cuando vuelva a ejecutarse el padre los registros deben recuperar los valores que tenían. Por consiguiente, la expansión de `LD_EX(#DPROG,#DARG)` es:

```
PUSH .0          POP  .12
PUSH .1          POP  .11
PUSH .2          POP  .10
PUSH .3          POP  .9
PUSH .4          POP  .8
PUSH .5          POP  .7
PUSH .6          POP  .6
PUSH .7          POP  .5
PUSH .8          POP  .4
PUSH .9          POP  .3
PUSH .10         POP  .2
PUSH .11         POP  .1
PUSH .12         POP  .0
LD.B .0, #12
LD   .1, #DARG
LD   .2, #DPROG
BRK
```

En el capítulo siguiente veremos en qué parte de la MP se carga el programa hijo. Baste decir de momento que, como este hijo es un programa independiente (es decir, no se ha montado con el padre), sólo puede acceder a los argumentos sabiendo que su dirección se le pasa por R1[6].

Ya sabemos que la terminación de un programa se señala al sistema con `EXIT`. La RIP, al interpretarla, llama al sistema de gestión de la memoria, que libera el espacio ocupado por el programa.

Otra forma de cargar un programa es con `LD_OV(#DPROG)`, que tiene la expansión:

```
PUSH .0
PUSH .2
LD   .2, #DPROG
LD.B .0, #13
BRK
POP  .2
POP  .0
```

Al igual que `LD_EX`, carga el programa especificado, pero *no* le pasa control, sino que sigue ejecutándose el programa en que se encuentra `LD_OV`; este programa hará luego una llamada «normal»

[6] También se le podría pasar esa dirección por la pila (Apartado 20.3), o en el «prefijo» (Apartado 19.3).

	Errores	Llamadas
1	Llamada no definida (número > 14)	
2	Periférico no operativo	todas, salvo `WAIT_A`, `CLOSE` y `POS`
3	Error de transferencia	`WRITE_A`, `READ_A`, `WRITE`, `READ`, `LD_EX`, `LD_OV`
4	El nombre no existe en el directorio	`LD_EX`, `LD_OV`, `OPEN`, `DELETE`, `CHMOD`, `STAT`
5	El fichero no tiene permiso de ejecución	`LD_EX`, `LD_OV`
6	Fichero en formato no ejecutable	`LD_EX`, `LD_OV`
7	Espacio insuficiente en memoria	`LD_EX`, `LD_OV`
8	Error en la suma de prueba	`LD_EX`, `LD_OV`
9	Directorio lleno	`CREATE`
10	Permiso incorrecto	`CREATE`, `CHMOD`
11	Modo incorrecto o no permitido	`OPEN`
12	Tabla de ficheros abiertos llena	`OPEN`
13	Descriptor de fichero inexistente	`WRITE_A`, `READ_A`, `WRITE`, `READ`, `CLOSE`, `POS`
14	Operación incompatible con modo	`WRITE_A`, `READ_A`, `WRITE`, `READ`
15	El fichero no puede crecer más	`WRITE_A`, `POS`
16	Espacio insuficiente en disco	`WRITE`, `WRITE_A`
17	Fichero especial	`POS`
18	Nombre de fichero incorrecto	`CREATE`
19	Error desconocido	

Tabla 18.2 Errores del sistema.

(con `CALL`) al cargado. Éste es un recurso antiguo para permitir la ejecución de programas que no caben en la MP en sistemas que no disponen de memoria virtual. El programador debe estructurar su programa en **módulos solapables**: en un *módulo raíz*, que siempre permanecerá en la MP, incluirá las llamadas a los otros módulos (subprogramas) y las estructuras de datos comunes que permiten intercambiar informaciones entre los módulos. Las sucesivas `LD_OV` irán cargando cada módulo utilizando el espacio ocupado por el anterior (Figura 18.2). Como la llamada a cada módulo se hace con una instrucción `CALL`, el retorno al módulo raíz debe realizarse con `RET`, no con `EXIT`.

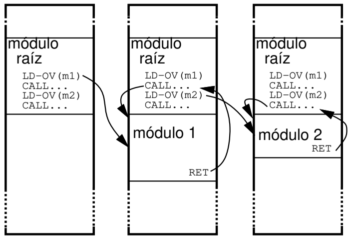


Figura 18.2. Estados sucesivos de la MP al cargar módulos solapables.

18.7. Errores

La Tabla 18.2 contiene la lista de todos los errores que el sistema puede detectar, acompañados de su número (que el sistema devuelve, en forma negativa y en ocho bits, en R13) y de los nombres de las llamadas que pueden provocarlos. El significado de muchos de estos errores se entenderá mejor en la próxima Lección.

Subprograma de mensajes de error

Resulta útil disponer de un subprograma que, a partir del número contenido en R13, escriba por la pantalla el mensaje de error oportuno. Es interesante que el lector analice (como ejercicio de programación, no directamente vinculado al propósito de esta Lección) la versión que se propone en el Programa 18.5 (página siguiente), porque contiene una especie de ardid para ahorrar instrucciones frente a la versión «ingenua» que consistiría en comprobar, uno a uno (con `CMP.B .13,#-1`, etc.), los errores que puede devolver el sistema.


```
MODULE RUTINAS
EXPORT MENS_ERROR
    MENSAJE    RES.B     80
    ERROR1     DATA.B      "Llamada no reconocida",H'0D
    ERROR2     DATA.B      "El periférico no está preparado",H'0D
    ERROR3     DATA.B      "Error en la transferencia, H'0D
    ERROR4     DATA.B      "No se encuentra el fichero en este disco",H'0D
    ERROR5     DATA.B      "No tiene permiso de ejecución",H'0D
    ERROR6     DATA.B      "Fichero en formato no ejecutable", H'0D
    ERROR7     DATA.B      "Espacio insuficiente en memoria",H'0D
    ERROR8     DATA.B      "Error al cargar en memoria",H'0D
    ERROR9     DATA.B      "El directorio está lleno",H'0D
    ERROR10    DATA.B      "Permiso incorrecto",H'0D
    ERROR11    DATA.B      "Modo incorrecto o no permitido",H'0D
    ERROR12    DATA.B      "No se pueden abrir más ficheros",H'0D
    ERROR13    DATA.B      "El descriptor del fichero no existe",H'0D
    ERROR14    DATA.B      "Operación incompatible con modo",H'0D
    ERROR15    DATA.B      "El fichero no puede crecer más",H'0D
    ERROR16    DATA.B      "Espacio insuficiente en el disco",H'0D
    ERROR17    DATA.B      "Operación no permitida en un fichero especial",H'0D
    ERROR18    DATA.B      "Nombre de fichero incorrecto",H'0D
    ERROR19    DATA.B      "Error del sistema",H'0D
    TAB_DIR_MENS DATA    ERROR1 ; tabla de direcciones
                 DATA    ERROR2 ; de mensajes
                 DATA    ERROR3
                 DATA    ERROR4
                 DATA    ERROR5
                 DATA    ERROR6
                 DATA    ERROR7
                 DATA    ERROR8
                 DATA    ERROR10
                 DATA    ERROR11
                 DATA    ERROR12
                 DATA    ERROR13
                 DATA    ERROR14
                 DATA    ERROR15
                 DATA    ERROR16
                 DATA    ERROR17
                 DATA    ERROR18
                 DATA    ERROR19
    ; punto de entrada de la rutina:
    MENS_ERROR  NEG    .13
                CMP.B  .13,#20
                BN     COPIA
                LD.B   .13,#19
    COPIA       SUB.B  .13,#1
                SHL    .13        ; (R13) = dirección relativa en TAB_DIR_MENS
                ADD    .13,#TAB_DIR_MENS
                LD     .1,[.13++] ; (R1) = dirección de comienzo de ERRORx
                LD     .2,#MENSAJE
    BUCLE       LD.B   .0,[.1++]  ; copia los caracteres del mensaje
                ST.B   .0,[.2++]  ; en MENSAJE
                CMP.B  .0,#H'0D
                BNZ    BUCLE
                WRITE(#0, #MENSAJE, #80)
               RET
    ---        ; otras rutinas
```

Programa 18.5 Subprograma para escribir mensajes de error.

18.8. Modelo estructural

La Figura 18.3 muestra los componentes de Monoalgorítmez y sus interrelaciones. El **núcleo** está formado por los seis módulos indicados[7] más un módulo principal y otros dos: el llamado «RUTINAS» en el Apartado anterior e «INICIA», que contiene un programa para realizar operaciones iniciales (por ejemplo, poner los valores de los vectores de interrupción), como veremos en el Apartado siguiente. Todos estos módulos se habrán ensamblado y montado dando lugar a un módulo de carga (Figura 3.16 y Apartado 6.8), residente en el disco `A` en un fichero llamado `sistema`.

El intérprete de órdenes es un programa «normal» (como otro programa de usuario cualquiera, no incluido en el sistema operativo), que hace llamadas al sistema mediante macros. Como sabemos, es la instrucción `BRK` la que realmente hace la llamada en el nivel de máquina convencional, mediante una interrupción. Por eso, las líneas que representan a las llamadas en la parte superior de la figura son de puntos: para el usuario (programador de aplicaciones), se trata de entradas al núcleo, pero en realidad (en la implementación) lo que ocurre es que el vector de interrupción de la `BRK` (en la dirección D'260, Tabla 6.10) contiene la dirección de comienzo de la RIP; ésta identifica a la llamada (mirando el contenido del registro R0) y bifurca al módulo del núcleo que corresponda.

El módulo principal del núcleo, Programa 18.6, es muy sencillo. Empieza bifurcando a `INICIA`, y éste termina volviendo al módulo principal, en `CARGAINT`, donde se pide la carga y ejecución del intérprete de órdenes, que se supone almacenado en un fichero del disco `A` con el nombre `int_ord`. Como veremos más adelante (Programa 18.8), el intérprete contiene un bucle sin fin. Por tanto, sólo se pasa a `CALL /MENS_ERROR` (símbolo importado del módulo `RUTINAS`, Programa 18.5) si no existe tal nombre en el directorio, o si se produce otro error; en tal caso, se vuelve a iniciar el sistema (Ejercicio 7).

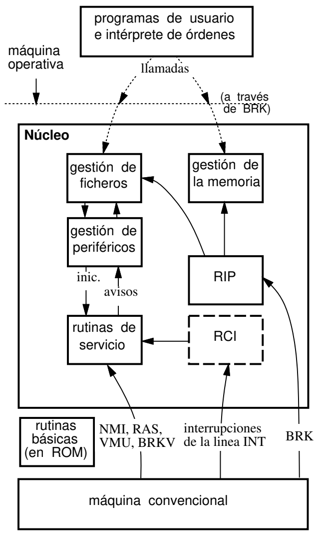


Figura 18.3 Modelo estructural de Monoalgorítmez.

```
MODULE MP_NUCLEO
    FROM INICIA IMPORT INI_ENT
    FROM RUTINAS IMPORT MENS_ERROR
    EXPORT CARGAINT
         ORG    0
         RES.B  268; reservado para los
              ; vectores de interrupción
PRINC    BR     /INI_ENT
CARGAINT LD_EX(#NOM_INT,#ARG)
         CALL   /MENS_ERROR
         BR     PRINC
NOM_INT  DATA.B "A:int_ord",0
ARG      DATA.B 0 ; int_ord no necesita
                  ; argumentos
         END    PRINC
```

Programa 18.6 Módulo principal del núcleo.

[7] La rutina de consulta de interrupciones, RCI, puede no ser necesaria (Apartado 7.6). Para `BRKV` sería necesario otro módulo, puesto que tiene su propio vector de interrupción, pero no vamos a usarla.

Una idea que debe quedar clara es que este módulo, una vez ensamblado, se monta con los demás módulos del núcleo y, al arrancar, se carga en la MP con un *cargador absoluto*. Por tanto, la dirección de comienzo de la ejecución será D'268 (determinada por la directiva `END PRINC`), quedando libres para los vectores de interrupción los 268 primeros bytes.

En el Apartado siguiente estudiaremos el proceso de arranque, que termina con la ejecución del módulo `INICIA`, y el intérprete de órdenes. Y en la Lección siguiente veremos el funcionamiento de los demás módulos y las estructuras de datos que es preciso tener en la MP. Ahora presentaremos la organización de los datos en los discos.

Bloques, ficheros e inodos

Como cada disco tiene 2.880 bloques (sectores), para representar el número de un bloque basta con doce bits. No obstante, utilizaremos dieciséis para simplificar las estructuras de datos[8].

En cada disco se pueden definir hasta doscientos cincuenta y seis ficheros (incluido el directorio, que, en nuestro modelo simplificado, es único). Para tener constancia de los números de bloques asignados a los ficheros, en Monoalgorítmez se sigue un método de *asignación indexada* (Apartado 17.8): asociado a cada fichero definido sobre un disco existe un conjunto de treinta y dos bytes, al que llamaremos **inodo**, en el que se encuentra toda la información sobre el fichero. El contenido de un inodo se interpreta así (Figura 18.4):

- En los dos bits más significativos de «tipo» se codifica si es un fichero ordinario, directorio, especial de caracteres o especial de bloques. En lo sucesivo supondremos que es ordinario (la mayoría de las informaciones que siguen no tienen sentido para los especiales).
- «RWX» son los bits de permiso.

[8] Y también porque de ese modo se podrían añadir al sistema discos de mayor capacidad sin tener que rediseñar todo el sistema de ficheros (Ejercicio 6).

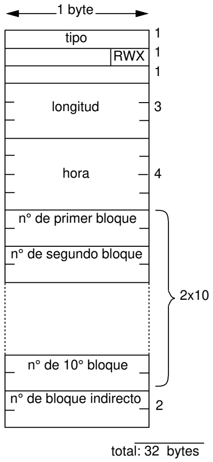


Figura 18.4 Estructura de un inodo.

- El tercer byte no se utiliza (en Multialgorítmez, Lección 20, servirá, junto con el byte anterior, para introducir informaciones necesarias en los sistemas multiusuario).
- «longitud» contiene el número total de bytes del fichero. El máximo valor de este número es (por la razón que enseguida veremos) 133 KB = 136.192 bytes, y para representarlo son necesarios dieciocho bits; en el inodo se reservan tres bytes.
- «hora» (cuatro bytes) contiene la fecha y la hora de la última modificación del fichero, codificada en treinta y dos bits: número (sin signo) de segundos transcurridos desde una fecha inicial arbitraria, el 1 de enero de 1970.
- Los veinte bytes siguientes contienen las direcciones de hasta diez bloques asignados al fichero; así, el inodo puede contener los números de bloques de ficheros cuya longitud no supere 10×512 = 5 KB.
- Para ficheros mayores de 5 KB, los números de los bloques siguientes se encuentran en otro bloque del disco, cuyo número figura en los dos últimos bytes del inodo. Este bloque con números de bloques permite hacer una asignación con «direc-

cionamiento indirecto» de hasta 256 bloques más, que en total pueden contener 256×512 = 128 KB. De ahí que el tamaño máximo de los ficheros sea 5 KB + 128 KB=133 KB.

Todos los inodos de los ficheros existentes en un disco deben estar presentes en ese disco. Para 256 ficheros necesitaremos 256 inodos, que requieren para su almacenamiento 256×32 = 8 KB o, lo que es lo mismo, dieciséis bloques.

Obsérvese que el inodo no contiene la información sobre la *posición* en el fichero. Ésta es una *información dinámica*, que sólo tiene sentido para los ficheros que están abiertos, y que estará incluida en la tabla de ficheros abiertos que se mantiene en la MP, como veremos en la Lección siguiente.

Directorio

El directorio de cada disco es un fichero que tiene siempre asignado el inodo de número 0. Puede representarse como una tabla de n filas (Figura 18.5), con $n \leq 256$. Cada fila tiene dieciséis bytes: dos para el número del inodo que corresponde a ese fichero (H'FFFF en estos bytes indica que la fila está libre)[9] y catorce para los caracteres que forman el nombre del fichero.

El directorio se almacena en un fichero de nombre «/», al que siempre le corresponde el inodo de número 0. La longitud máxima de este fichero será: 256×16 = 4 KB.

Formatos de los ficheros

Si bien el sistema operativo no «entiende» de tipos de ficheros (son sólo secuencias de bytes almacenados en sectores del disco), conviene establecer algún convenio para que los programas que trabajan con ellos (ensamblador, montador, cargador, etc.) puedan detectar errores y hacer que el

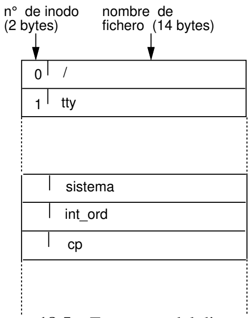


Figura 18.5 Estructura del directorio.

funcionamiento sea más robusto. El tratamiento de las llamadas `LD_EX` y `LD_OV` también puede hacerse más seguro si el sistema es capaz de detectar que el programa de usuario pretende ejecutar un fichero cuyo contenido no es ejecutable (error 6 de la Tabla 18.2). Ya hemos visto, en el Apartado 6.8, la manera de hacerlo: añadiendo al principio de todo fichero una **cabecera** que contiene, entre otras cosas, un **número mágico** que identifica al tipo de fichero. Repetimos aquí, y completamos, los formatos de los tres tipos de ficheros que más nos interesan: los que contienen módulos de programas fuente en ensamblador (que sirven de entrada al ensamblador), los que contienen módulos objeto (salidas del ensamblador) y los que contienen módulos de carga (salidas del montador).

- Los ficheros que contienen módulos fuente tienen una cabecera de catorce bytes. Los dos primeros contienen el número mágico H'0000, y en los doce siguientes está el nombre del módulo (si tiene menos de doce, se completa con «nulos», H'00), que deben coincidir con los primeros caracteres del nombre del fichero. A continuación, todos los caracteres ASCII del programa fuente.
- La cabecera de los módulos objeto es como la de los módulos fuente, pero con el número mágico H'A0A0. A continuación, las distintas partes del módulo (registros del código, diccionario de reubicación y tablas de símbolos), con el formato explicado en el Apartado 6.8

[9]Como no puede haber más de 256 inodos, bastaría con un byte. Reservando dos, queda abierta la posibilidad de definir fácilmente otros formatos de disco (véanse las notas al pie de la página siguiente). Y, además, tenemos una codificación para señalar que la fila está libre.

- Los ficheros de los módulos de carga se identifican por el número mágico H'0707. Los dos bytes siguientes contienen el número total de bytes del código a cargar en la MP (este número lo habrá calculado el montador al generar el módulo de carga). Y el resto, los registros de código, el diccionario de reubicación y la dirección de comienzo de ejecución, que ocupa los dos últimos bytes.

Organización de los discos

El primer bloque (bloque 0) es el **sector de arranque**, y contiene un programa que se carga en la MP y se ejecuta al arrancar el ordenador. En los diecisiete siguientes se mantienen «metadatos», es decir, informaciones sobre cómo están organizados los datos en el disco (Figura 18.6).

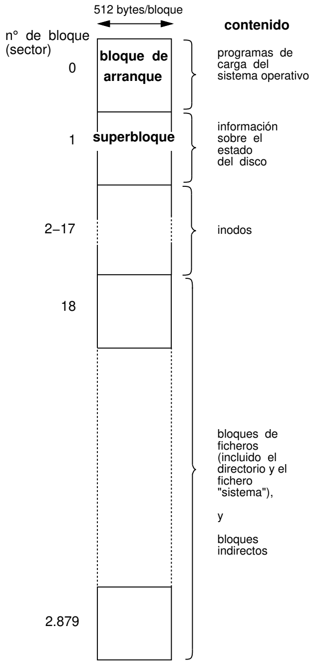


Figura 18.6 Organización de los discos.

En el **superbloque** se encuentran datos sobre el estado del disco:

- Informaciones generales sobre el disco, como el número de bloques, el número de sectores por bloque y el número de inodos[10].
- El *mapa de bits de inodos*, formado por 256 bits (32 bytes)[11]. Cada bit se corresponde con un inodo del disco: si es «0» indica que el inodo está libre, y si es «1», que está asignado a algún fichero. Los primeros inodos siempre estarán ocupados (corresponden al directorio, a `tty`, al fichero que contiene al sistema, etc.).
- El *mapa de bits de bloques*, con 2.880 bits (360 bytes)[12], con idéntica función respecto de los bloques del disco: «0» indica que el bloque está disponible, y «1» que está asignado a algún fichero (o que es un bloque indirecto). Naturalmente, los dieciocho primeros bits siempre tienen el valor «1», porque, como puede verse en la Figura 18.6, los bloques correspondientes siempre están ocupados.

Estas informaciones del superbloque se cargan inicialmente en la MP, y se mantienen permanentemente en ella para que los programas del sistema de gestión de ficheros no tengan que acceder al disco y puedan ejecutarse más rápidamente.

Después del superbloque se encuentran los dieciséis bloques que están reservados para los inodos (256×32 = 8 KB). El número de inodos asignados en un momento dado, $n \leq 256$, coincide con el de ficheros existentes en el directorio, y, de éstos, sólo un subconjunto estará normalmente en uso. Ésta es precisamente la razón de ser fundamental de la operación «**abrir**» un fichero («`OPEN`»): copiar en la MP el inodo del fichero con el que se va a trabajar.

[10]En los discos que estamos considerando estos números son 2.880, 1 y 256, respectivamente. Pero el sistema podría tener otros discos de otras características.

[11]O, de manera más general, con tantos bits como inodos estén previstos, según lo que indique el bloque de arranque.

[12]O con tantos bits como bloques tenga el disco.

18.9. Modelo procesal

El funcionamiento de los componentes de Monoalgorítmez lo estudiaremos en la Lección siguiente. Veamos aquí dos elementos del modelo procesal: el arranque del sistema y el funcionamiento del intérprete de órdenes.

Arranque

El proceso de arranque consta de tres partes:

1. Ejecución de un programa grabado en ROM, que termina cargando el sector de arranque del disco `A` en la MP.
2. Ejecución del programa de arranque cargado.
3. Ejecución del módulo `INICIA`.

Al conectar el ordenador, los circuitos de Algorítmez ponen H'0400 en el registro de estado (lo que inhibe las interrupciones y pone el modo supervisor), introducen en el contador de programa, R15, y en el puntero de pila, R14, la dirección H'FC00, y dan comienzo a un ciclo de instrucción. Entre H'FC00 y H'FFFF hay 1.024 bytes, los de direcciones más altas de la MP. Suponemos (Apartado 6.2) que la memoria que corresponde a estas direcciones es de sólo lectura (ROM), y contiene rutinas básicas grabadas en la ROM para iniciar al sistema y para realizar operaciones elementales como leer un carácter del teclado, poner valores en los registros del controlador de ADM (funciones «`iniciar`» y «`actualizar`» de los Programas 5.1 y 5.2), etc. Para la pila se reserva 1 KB, entre las direcciones H'F800 y H'FBFF[13].

Empezando en la dirección H'FC00 se encuentra un programa que comprueba los recursos de hardware del sistema. Por ejemplo, escribe y lee sucesivamente en todos los bytes de la MP (salvo en la zona ROM) para detectar posibles errores. A continuación lee el primer sector del disco `A` (poniendo los valores adecuados en los registros del controlador de ADM), lo introduce en una zona de la MP que luego podrá ser ocupada por otros programas, la comprendida entre las direcciones H'F600 y H'F7FF, y termina con una instrucción `BR /H'F600` (Apartado 7.8 y Programa 7.7).

Este programa que se acaba de cargar lee el superbloque del disco `A` para conocer el formato del disco. A continuación, lee el primer inodo del mismo disco (el de número 0, que corresponde al directorio), y, a través de sus informaciones, el primer bloque del directorio. Si en él se encuentra la entrada de `sistema`, trae su inodo a memoria, y si no, sigue leyendo bloques del directorio hasta encontrarla. Naturalmente, termina con el inodo en memoria, o dando el error de que el disco no contiene al sistema.

En los bloques del fichero `sistema` está almacenado el módulo de carga de todo el núcleo, con el formato que vimos en el Apartado 6.8 y con los cuatro bytes de cabecera que el montador añade a los ejecutables. El programa de arranque ejecuta un *cargador absoluto*, que lee esos bloques, comprueba las sumas de prueba, y va colocando los bytes de código en la MP a partir de la dirección 0. En el último de los registros de código del módulo está la dirección de comienzo de ejecución de `sistema`. Esta dirección es la que corresponde a la etiqueta `PRINC` en el Programa 18.6, es decir, la dirección absoluta D'268. El programa de arranque (que, recordemos, se había leído del sector de arranque del disco, y se encuentra transitoriamente en memoria) termina su misión introduciendo esa dirección en el contador de programa.

Quizás el lector sea ahora un poco más comprensivo con su ordenador y su sistema operativo[14] cuando está esperando que arranque. Y aún quedan algunas operaciones antes de que el sistema nos invite a dar una orden:

- Poner valores en los vectores de interrupción y en los puertos de los periféricos conectados.
- Poner valores en la tabla de periféricos (ésta es una estructura de datos que explicaremos en la siguiente Lección)

[13] Si el lector no entiende por qué a PP se le da como valor inicial H'FC00 (y no H'FBFE), consulte las descripciones de las instrucciones `PUSH` (Tabla 6.3) y `CALL` (Tabla 6.5).

[14] Bastante más complejos que Monoalgorítmez, aunque se trate de los ya venerables 8088 y MS-DOS 3.0.


```
MODULE INICIA
  FROM MP_NUCLEO IMPORT CARGAINT
  FROM RS IMPORT RS_TEC,RS_PAN,RS_IMP,RS_ADM,RS_RAS,RS_BKV,RS_NMI,RS_VMU,RS_ERR
  FROM RIP IMPORT RIP_ENT; comienzo de la RIP (Programa 19.1)
  FROM RCI IMPORT RCI_ENT; comienzo de la RCI (Programa 7.3)
  FROM GP IMPORT TBPRF; dirección de la tabla de periféricos, en el módulo GP
  FROM SGM IMPORT TBFA, PDL; tabla de ficheros abiertos y PDL, en el módulo SGM
  EXPORT INI_ENT
INI_ENT CLR   .0
BUC     LD    .1,#RS_ERR  ; Inicialmente, pone en todos los vectores la
        ST    .1,[.0++]   ;  dirección de RS_ERR (interrupción desconocida)
        CMP   .0,#268
        BN    BUC
        CLR   .0          ; Ahora ya pone los vectores de los periféricos
        LD.B  .1,#2       ; Aprovecha que lo que tiene en R0 es también la
        OUT   .1,[.0]     ;  dirección de e/s del puerto de estado
        LD    .2,#RS_TEC  ; Dirección de la rutina de servicio del teclado
        ST    .2,[.0++]   ;  al vector 0
        LD.B  .1,#1
        OUT   .1,[.0]     ; Lo mismo, con la pantalla
        LD    .2,#RS_PAN
        ST    .2,[.0++]
        OUT   .1,[.0]     ; Lo mismo, con la impresora
        LD    .2,#RS_IMP
        ST    .2,[.0++]
        LD    .0,#18      ; Vector (y puerto de estado) del controlador de ADM: D'18
        OUT   .1,[.0]
        LD    .2,#RS_ADM
        ST    .2,[.0++]
        LD    .0,#256     ; Ahora pone los vectores de las interrupciones:
        LD    .2,#RS_RAS  ;   internas (Tabla 6.7), ***********OJO ???***********
        ST    .2,[.0++]
        LD    .2,#RS_BKV  ;   de BRKV
        ST    .2,[.0++]
        LD    .2,#RIP_ENT ;   de programa (BRK)
        ST    .2,[.0++]
        LD    .2,#NMI     ;   no enmascarable
        ST    .2,[.0++]
        LD    .2,#VMU     ;   de violación de modo usuario
        ST    .2,[.0++]
        LD    .2,#RCI_ENT ;   y de la rutina de consulta de interrupciones
        ST    .2,[.0++]
        ...
        ...; FALTA: poner valores en TBPRF, comprobar funcionamiento de periféricos,
        ...;  poner inodos de tty y de los directorios en TBFA,
        ...;  cargar superbloques y poner valor inicial de PDL
        ...
        LD.E   #H'0100    ; Modo usuario, permiso de interrupciones
        BR     /CARGAINT
    END
```

Programa 18.7 Módulo `INICIA` (incompleto).

- Cargar en la MP los inodos de `tty` y del directorio (esta última operación ya la había hecho, provisionalmente, el programa de arranque) y los superbloques de los discos (para tener siempre en la MP los mapas de bits).

Estas operaciones las realiza ya el módulo `INICIA`. En el Programa 18.7 aparecen detalladas las del primer punto. Como hace referencia a símbolos definidos en otros módulos, que estudiaremos en la Lección siguiente, sugerimos al lector que se fije de momento en la parte detallada (iniciación de vectores y puertos), y que más adelante trate de completar el programa. Los módulos de los que se importan símbolos en éste son `MP_NUCLEO` (módulo principal, Programa 18.6), `SGM` (sistema de gestión de la memoria, Programa 19.2), `GP` (gestores de periféricos), `RS` (rutinas de servicio), `RIP` (rutina de interrupciones de programa, Programa 19.1) y `RCI` (rutina de consulta de interrupciones, Programa 7.3).

Todo este proceso se ha desarrollado con las interrupciones inhibidas. Ahora, cargados ya los vectores de interrupción y las rutinas de servicio, el programa `INICIA` termina por fin: carga los superbloques, pone un valor inicial en RE y vuelve al módulo principal, que carga el intérprete de órdenes. Si el lector adelanta unas páginas puede ver el mapa de memoria resultante, en la Figura 19.1.

Intérprete de órdenes

Como todo intérprete (Apartado 22.3), éste es un programa muy sencillo. Se limita a leer una línea del teclado, analizar sus primeros caracteres y llamar al programa que ejecuta la orden que hemos tecleado. Suponemos que en el disco `A` hay ficheros con nombres «`cp`», «`rm`», etc., que contienen a esos programas en formato de fichero ejecutable.

Una posible implementación del intérprete es la del Programa 18.8. Las instrucciones esenciales son las marcadas con asteriscos. El resto sirve para comprobar que no hay errores y añadir, en su caso, la especificación del disco activo al nombre dado en la orden.

```
MODULE INT_ORD
FROM RUTINAS
   IMPORT ANALIZA, COMPLETA, MENS_ERROR
EXPORT ORDEN, NOMBRE, DISCO
 ORDEN     RES.B   80
 NOMBRE    RES.B   16
 INVIT     DATA    "Mono>", 0
 DISCO     DATA    "A:" ; valor inicial

 INT_ORD  WRITE(#0,#INVIT,#80) ; ***
          READ(#0,#ORDEN,#80)  ; ***
          CALL      /ANALIZA
          CMP.B     .13,#0
          BN        INT_ORD
          CALL      /COMPLETA
          CMP.B     .13,#2
          BNN       INT_ORD
          LD_EX(#NOMBRE,#ORDEN); ***
          CMP.B     .13,#0
          BNN        INT_ORD
          CALL      /MENS_ERROR
          BR        INT_ORD    ; ***
         END     INT_ORD
```

Programa 18.8 Intérprete de órdenes.

«`Mono`» es la **invitación** del sistema a escribir una orden (el nombre de un programa ejecutable)[15].

«`ANALIZA`» es un subprograma que explora los caracteres de `ORDEN`, y devuelve en R13:

- (R13) = 0 si es un nombre correcto, precedido de `A:` o `B:`.
- (R13) = 1 si es un nombre correcto no precedido de `A:` ni de `B:`.
- (R13) = 2 si es `A:` seguido de «espacio» o de «retorno».
- (R13) = 3 si es `B:` seguido de «espacio» o de «retorno».

[15] En ejemplos anteriores (Programas 7.4 y 18.2), la cadena a escribir terminaba con H’0D (retorno). En este caso termina con H’00 (nulo) para que la orden del usuario se escriba inmediatamente a continuación de la invitación. Esto obligaría a hacer una pequeña modificación en la rutina de servicio de la pantalla del Programa 7.5. Además, necesitaría algunos otros cambios; véase el Ejercicio 13 de la Lección 19.

- (R13) =–1 si es un nombre incorrecto. Por ejemplo, si pasa de catorce bytes sin encontrar H’20 (espacio) o H’0D (retorno). En este caso escribe en la pantalla el mensaje de error.

Por su parte, «`COMPLETA`» hace lo siguiente:

- Si (R13) = 0, extrae los caracteres de `ORDEN` hasta el primer «espacio» o «retorno» (es decir, los que forman el nombre) y los pone en `NOMBRE` (quitándolos de `ORDEN`).
- Si (R13) = 1, pone en `NOMBRE` los dos caracteres que están almacenados en `DISCO` y a continuación los que forman el nombre.
- Si (R13) = 2, o (R13) = 3, pone `A:` o `B:`, respectivamente, en la palabra de etiqueta «`DISCO`».

18.10. Resumen

«Monoalgorítmez» es Algorítmez acompañado de unos programas que constituyen un *pequeño sistema operativo sin multiprogramación* (y sin memoria virtual, ni sistema de ficheros jerarquizado, ni protección).

Con la instrucción `BRK` *el programador puede provocar una interrupción* en cualquier punto del programa para pedirle al sistema que se encargue de operaciones de entrada/salida y de carga y ejecución de programas. Éste es el principio fundamental sobre el que se construyen las instrucciones virtuales del nivel de máquina operativa.

No basta con `BRK`: el programa debe comunicar al sistema qué operación desea, de qué fichero hay que leer (o en cuál hay que escribir) y la zona de la MP donde se depositan los datos leídos del periférico o donde están los datos a escribir en el periférico. En Monoalgorítmez, esas informaciones se transmiten (pasando valores o direcciones) a través de registros de la UCP (de R0 a R7).

Para facilitar más la programación, el ensamblador (lenguaje) puede acoger *macroinstrucciones* («macros»). Las macros predefinidas son construcciones simbólicas que el ensamblador (procesador) de Algorítmez reconoce y traduce por secuencias de instrucciones que cargan los registros y terminan con `BRK`. Hemos visto cómo se pueden utilizar algunas de estas macros para programar fácilmente (al menos, más fácilmente que directamente sobre la máquina convencional) las operaciones de entrada/salida.

Las macros son una manera de implementar lo que en general se denominan *llamadas al sistema*. En los sistemas actuales se implementan más bien con funciones de biblioteca a las que pueden llamar los programas en lenguajes de alto nivel.

El *modelo estructural* general está formado por la máquina convencional y los módulos de software para consulta de interrupciones («RCI»), reconocimiento de las interrupciones de programa («RIP»), servicio de interrupciones, gestión de periféricos, gestión de memoria y gestión de ficheros. Los ficheros pertenecen todos a un directorio único para cada disco; se ha descrito la organización de la información en los discos, basada en el uso de *inodos* para la asignación de espacio a los ficheros.

Finalmente, hemos detallado el proceso que ocurre durante el arranque del sistema y el de la ejecución del *intérprete de órdenes*. Éste no es más que un programa que se ejecuta en modo usuario, llamando al sistema para pedir la carga y ejecución del programa que se indique en la orden.

18.11. Observaciones terminológicas

Aunque ya lo hemos señalado en el Apartado 2.11 conviene repetir que lo que llamamos «zona» (de la MP) corresponde a «buffer». También (Apartado 17.12) que el «intérprete de órdenes» se llama «shell» en UNIX. El equivalente en MS-DOS de nuestro programa «`int_ord`» es el «COMMAND.COM».

«Tecleo anticipado» es traducción de «type ahead», «módulo solapable» de «overlay», e «invitación» de «prompt».

Los «inodos» son simplificaciones de los «inodes» de UNIX[16], nombre que procede de «index node».

[16] Que contienen más informaciones, y doble y triple indirección (Figura 21.3); cada inode tiene sesenta y cuatro bytes.

Los nombres de los ficheros especiales «tty» (por «teletype») y «lp» (por «line printer») son los típicos de UNIX, como también lo es el término «descriptor» (que, en otros contextos, se llama *handle*, «asa», o «mango»).

El nombre de la orden `chmod` es una abreviatura de «cambio de modo». A los permisos de un fichero es frecuente llamarles el «modo» del fichero, lo que resulta confuso, porque el *modo de apertura* (para leer, para escribir o para leer y escribir) es otra cosa.

Los nombres elegidos para las macros son una mezcla de los que reciben las llamadas en los sistemas UNIX y MS-DOS, como puede verse en la Tabla 18.3. (En MS-DOS los nombres no están tan normalizados: los programadores conocen a las llamadas más bien por el valor a cargar en el registro AH, que es equivalente en esta función a nuestro R0).

En UNIX existe «wait», pero con un significado distinto a nuestra `WAIT_A` (Apartado 20.3).

Monoalg.	UNIX	MS-DOS
`CREATE`	creat	create (H'16)
`DELETE`	unlink	delete (H'13)
`OPEN`	open	open (H'0F)
`CLOSE`	close	close (H'10)
`WRITE_A`		
`READ_A`		
`WAIT_A`		
`WRITE`	write	write (H'40)
`READ`	read	read (H'3F)
`POS`	lseek	pos (H'42)
`CHMOD`	chmod	chmod (H'43)
`STAT`	stat	get-file (H'23, H'57)
`LD_EX`	fork/exec	load-and-exec (H'4B, (AL) = 0)
`LD_OV`		load-overlay (H'4B, (AL) = 3)
`EXIT`	exit	end-prog (H'00)

Tabla 18.3. Equivalencia de las macros de Monoalgorítmez con las llamadas de UNIX y MS-DOS.

18.12. Notas históricas

Las macros eran típicas de los sistemas operativos de los años 60, cuando la programación de sistemas se hacía fundamentalmente en ensamblador. Actualmente sólo se programan en ensamblador las rutinas básicas que controlan directamente los recursos de hardware; el resto del sistema se programa en algún lenguaje de alto nivel (en el caso de UNIX, el lenguaje C). Las llamadas al sistema se implementan con funciones de este lenguaje. Conceptualmente, sin embargo, no hay diferencia en los mecanismos básicos (Apartado 21.2).

18.13. Orientaciones bibliográficas

Si el lector tiene interés por estudiar con más detalle el uso de macros en sistemas «clásicos» puede acudir a Freeman y Perry (1977). Las llamadas al sistema MS–DOS se describen en muchos libros, como los de Gear (1985), Tisher (1992) y Schulman *et al.* (1994).

La funcionalidad de las llamadas al sistema en UNIX se describe detalladamente en las páginas de manual accesibles en todo sistema de este tipo. En el Apartado 19.13 se dan referencias sobre la relación de Monoalgorítmez con sistemas operativos reales.

18.14. Ejercicios

1. El formato descrito para los discos de Monoalgorítmez es uno bastante común para los disquetes de 3,5 pulgadas. Frecuentemente se dice que la capacidad de estos disquetes es «1,44 MB». ¿Es esto correcto?

2. Escriba las expansiones de las macros que aparecen en los Programas 18.2 y 18.3.

3. Modifique el Programa 18.2 para que utilice la macro `WRITE` y para que escriba también por la impresora los mensajes.

4. Perfeccione los Programas 18.3 y 18.4 para que en el caso de un error en la transferencia escriban un mensaje. (Considere tanto los errores que el sistema puede devolver por R13 como el que puede devolver en «aviso»).

5. El convenio para representar la fecha y la hora (Figura 18.4) es el habitualmente seguido en UNIX. En MS–DOS se representan los componentes por separado: el año (con dos dígitos decimales), el mes, etc. Con el convenio de UNIX (segundos desde 1/1/1970 en treinta y dos bits sin signo), calcule cuándo puede presentarse un problema similar al del «año 2000».

6. Teniendo en cuenta que en los inodos se utilizan dieciséis bits para representar los números de bloques, ¿hasta qué capacidad de disco podríamos utilizar? (suponiendo bloques de 512 bytes). ¿Y si los bloques se redefinen de tamaño igual a cuatro sectores? ¿Qué sería preciso modificar en los superbloques?

7. Lo que se hace en el Programa 18.6 en el caso de error es absurdo: si se repite todo, lo más probable es que se vuelva a generar el mismo error. Una alternativa es que se le diese la oportunidad al usuario de escribir otro nombre para el intérprete, o para un programa ejecutable. Modifique el programa para incluir esta alternativa.

8. Con relación al mismo programa, lo anterior se puede completar dando la opción al usuario de que dé un nuevo nombre o que se pare la máquina (`HALT`). Ahora bien, esto último es imposible. ¿Por qué? ¿Qué tendríamos que añadir en el modelo funcional para que esta opción fuese posible?

9. Hemos dicho que la pila ocupará las direcciones H'F800 a H'FBFF. Pero no hay nada en el hardware de Algorítmez que permita detectar y evitar que los programas de usuario se salgan de esa zona. Escriba subprogramas `METER` y `SACAR` para ser utilizados en lugar de `PUSH` y `POP`, que detecten estas situaciones y en caso de desbordamiento (o «agotamiento») generen una interrupción de programa con (R0) = 15.

10. En los intérpretes de órdenes suele distinguirse entre **órdenes internas** (las que ejecuta el propio intérprete) y **órdenes externas** (las que ejecuta un programa llamado por el intérprete). A la vista de la implementación del intérprete (Programa 18.8), clasifique las órdenes mencionadas en el Apartado 18.2.

11. En el intérprete está previsto que el nombre de la orden pueda ir precedido o no de `A:` o `B:`, con un valor por defecto (que cambia el subprograma `COMPLETA`), pero no que se pase ese valor por defecto a los programas ejecutables. Esboce una forma de hacerlo. (Pista: por R1, el subprograma cargado recibe una dirección).

12. Escriba programas para interpretar las órdenes `cp`, `ls`, `mv`, `rm` y `chmod`.

13. Escriba un programa para `wc` (*word count*), una orden que presenta en pantalla el número de bytes, de palabras y de líneas de un fichero de texto.

14. «`lpr fich`» no es equivalente a «`cp fich lp`»: el programa «`lpr`» hace algunas operaciones adicionales. Así, entre otras cosas más sutiles (Apartado 21.4), cuando encuentra el carácter «tab» (H'09) lo sustituye por una sucesión de, por ejemplo, cinco «espacios» (H'20); los ficheros de texto pueden seguir el convenio de almacenar solamente la codificación de «nueva línea» (H'0A) en los cambios de línea, y «`lpr`» añadiría la de «retorno» (H'0D). Escriba el programa para «`lpr`» teniendo en cuenta estas dos características.

15. Al final del módulo `INICIA` se da el permiso de interrupciones (al poner H'0100 en RE). Pero inmediatamente se pasa, en el módulo principal, a ejecutar `LD_EX`, cuya expansión, como sabemos, contiene `BRK`, lo que tiene como efecto inhibir las interrupciones. ¿Por qué es necesario permitirlas antes? (No es válida la respuesta «porque `BRK` provoca una interrupción», ya que esta interrupción *no* es enmascarable).

16. Escriba los subprogramas `ANALIZA` y `COMPLETA` utilizados en el Programa 18.8. Observe que se ha omitido algo: el paso de argumentos. Estudie alternativas para hacerlo (Apartado 6.13).

17. En nuestro intérprete de órdenes no hemos previsto que admita «guiones» (**shell scripts**, o, en la terminología de MS-DOS, ficheros «`.BAT`»). Se trata de *ficheros de texto* que contienen, en cada línea, una orden a ejecutar. Antes de llamar a `LD_EX`, el intérprete debe investigar si el nombre es el de un fichero binario o de texto. En este segundo caso, lee sucesivamente las líneas, y, para cada una, hace `LD_EX` del mismo programa intérprete (es decir, siempre que se ejecuta un guión hay dos copias del intérprete en la MP). Añada lo necesario al Programa 18.8 para que acepte guiones. Suponga que:
   - Los ficheros de texto tienen en la cabecera el número mágico H'A7A7.
   - Las órdenes están en líneas separadas, es decir, al final de cada orden hay un carácter «`^J`» (H'0A).
   - El final del fichero se identifica por el carácter «`^Z`» (H'1A).

Lección 19

Núcleo de Monoalgorítmez

19.1. Propósito

A la vista del modelo estructural de Monoalgorítmez (Figura 18.3), podemos identificar tres componentes de software que se ejecutan sobre la máquina convencional Algorítmez:

- Las *rutinas básicas* de arranque y de acceso a los periféricos, residentes permanentemente en ROM en los últimos 1.024 bytes de la MP.
- El *núcleo*, formado por un conjunto de módulos ensamblados y montados, que se carga a partir de la dirección 0 en el arranque y ocupa las direcciones más bajas de la MP.
- El *intérprete de órdenes*, que se carga a continuación del núcleo, y que, en realidad, no es propiamente parte del sistema operativo.

Así pues, en este diseño, el núcleo es, prácticamente, todo el sistema operativo: incluye las rutinas de interrupciones, los gestores de periféricos, y los programas que realizan la gestión de la memoria y la gestión de ficheros. Y todos estos programas funcionan (salvo en algunos segmentos pequeños, como veremos) con las interrupciones inhibidas: desde que se entra en el núcleo con `BRK` (que, automáticamente, al generar una interrupción, hace que la UCP ponga 0 en PIN) hasta que se sale de él con la instrucción `RETI` (que, también automáticamente, vuelve a permitirlas).

Además, todo el sistema operativo permanece en la MP (la parte en ROM siempre; el resto, siempre que Algorítmez esté encendido).

Esta Lección se centra en las *facetas procesales del núcleo*. Se pretende que el lector se haga una idea de los procesos que se desencadenan como consecuencia de las llamadas al sistema operativo, procesos complejos pero transparentes para la mayoría de los usuarios, y gracias a los cuales la programación de aplicaciones se salva de convertirse en una pesadilla.

Dada la simplicidad del modelo funcional que hemos presentado en la Lección anterior, sería relativamente fácil escribir este núcleo en algún lenguaje de alto nivel (con llamadas a algunas rutinas en ensamblador para las operaciones muy ligadas al hardware, como las de manejo del sistema de interrupciones). Además, reflejaría mejor la realidad: así se procede actualmente para implementar un sistema operativo. Pero no es éste nuestro objetivo. Se trata de *comprender los procesos que tienen lugar en el nivel de máquina convencional*.

19.2. Algoritmos y estructuras de datos

Presentar todos los programas en ensamblador resultaría excesivamente largo, y tanto detalle oscurecería la visión de conjunto. Optaremos por el siguiente enfoque:

- Por lo que respecta a los *algoritmos*, veremos la codificación en ensamblador de los más sencillos: la RIP (rutina de interrupciones de programa) y el SGM (sistema de gestión de la memoria). El SGF (sistema de gestión de ficheros) es bastante más complejo, especialmente por lo que respecta al tratamiento de las llamadas de lectura y escritura, y lo describiremos de manera informal, apoyándonos en algunos ejemplos para ilustrar el procesamiento de esas llamadas. Los gestores de periféricos (y las rutinas de interrupción) son muy variados y están muy ligados a los detalles particulares de cada periférico. Veremos esquemáticamente dos de ellos.
- Para las *estructuras de datos* utilizaremos *tablas*, que permiten intuir fácilmente la forma de su almacenamiento en la MP. En el Apartado 7.4 ya hicimos referencia a una de ellas, la tabla de periféricos. En general, una tabla tiene un determinado número de *filas*, N_F, y todas estas filas (o «registros») tienen la misma estructura: una sucesión de *campos*, $C1, C2, \ldots\ Cn$, cada uno con un determinado número de bytes, $N_{C1}, N_{C2}, \ldots\ N_{Cn}$. Una tabla en la MP ocupará pues un número *fijo* de bytes, $N_F \times (N_{C1} + N_{C2} + \ldots + N_{Cn})$, almacenados de forma lineal: primero los N_{C1} del primer campo de la primera fila, y así sucesivamente[1].

19.3. Mapa de memoria

Enlazando con el final de la Lección anterior, la Figura 19.1 muestra cómo queda la MP después de terminado el proceso de arranque.

[1] Si el lector prefiere pensar en términos de un lenguaje de alto nivel, hay una traducción inmediata: una tabla es un *array* de *registros* (Apartado 22.9).

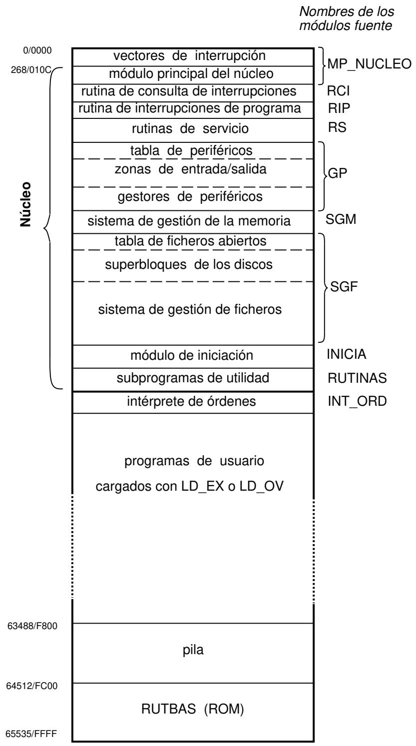


Figura 19.1 Mapa de memoria.

En las direcciones más bajas se encuentran los 134 vectores de interrupción. De acuerdo con lo explicado en el Apartado 6.14, el último (en la dirección D'266) es la dirección de comienzo de la RCI (rutina de consulta de interrupciones, Programa 7.3). (El módulo `INICIA`, Programa 18.7, se encarga de que así sea). Esta RCI sólo se ejecutará si los controladores de periféricos no ponen la dirección de su vector en el bus A, y tampoco existe controlador de interrupciones (y en lugar de este controlador se utiliza el circuito de la Figura 14.7).

En las direcciones determinadas por los vectores de interrupción que ha puesto INICIA se cargan las RS (rutinas de servicio) y la RIP (rutina de interrupciones de programa). En principio, el núcleo incluye cuatro RS: una para los dos discos (que comparten un único controlador de ADM), para el teclado, para la pantalla y para la impresora. Pero se podrían añadir otros periféricos. Sólo habría que escribir sus rutinas de servicio y sus gestores y volver a ensamblar (o compilar) todo el programa.

El módulo GP incluye a los gestores de periféricos junto con las estructuras de datos asociadas: la tabla de periféricos y las zonas de entrada/salida. Hay dos zonas de 512 bytes, una para cada disco, y tres de ochenta bytes, para los tres periféricos de caracteres.

El sistema de gestión de la memoria es un programa pequeño, cuya codificación en ensamblador veremos en el Apartado 19.6, y el sistema de gestión de ficheros contiene la tabla de ficheros abiertos (donde están copiados los inodos de los ficheros abiertos) y los superbloques de los discos (con los mapas de bits que informan sobre los inodos y los bloques que están libres, para que el sistema de gestión de ficheros pueda consultar y decidir rápidamente, accediendo a la MP y no al disco, sobre las asignaciones de bloques del disco a ficheros).

La distribución de los módulos en la MP no tiene por qué ser necesariamente la que muestra el mapa de la Figura 19.1. Esa distribución la decidirá, en definitiva, el montador. De lo que sí podemos estar seguros es de que el módulo MP_NUCLEO (que incluye a los vectores de interrupción) está al principio.

Padres e hijos

Tras la ejecución de MP_NUCLEO (Programa 18.6), el intérprete de órdenes queda permanentemente cargado en la MP. Cuando, como consecuencia de una orden, se ejecuta la macro LD_EX incluida en él (Programa 18.8), el sistema de gestión de la memoria carga el programa indicado en la orden a partir de la primera dirección libre. Si este programa contiene a su vez LD_EX o LD_OV, el nuevo programa se cargará a continuación de él, y así sucesivamente. Obsérvese que, según esto, puede haber varios programas de usuario en la MP (normalmente hay dos: el intérprete y el que se está ejecutando), pero *no por eso nuestro sistema es de multiprogramación*. En efecto, si un programa se queda bloqueado (por ejemplo, porque ha ejecutado WAIT_A y la operación correspondiente no ha terminado) el sistema simplemente espera (un sistema con multiprogramación, como el que describiremos en la Lección siguiente, pasaría control a otro programa de usuario).

Salvo el núcleo, todos los programas tienen un *padre*: el programa que contiene la macro con la que éste se ha cargado. El padre del intérprete es el núcleo. Cuando en un programa aparece EXIT, el núcleo tiene que pasar control al padre (en la instrucción siguiente a LD_EX) y liberar la memoria que ocupa (es decir, que la carga posterior de otro programa *hijo* pueda hacerse sobre la zona que ha ocupado éste).

Punteros y prefijos

Es el sistema de gestión de la memoria el que se encarga de estas operaciones de asignación y liberación de memoria para los programas, y de la carga de los mismos (transferencia del fichero en disco a la memoria y reubicación). Y para que pueda funcionar correctamente hay que mantener ciertas informaciones en forma de punteros: primera dirección libre, dirección a la que volver tras EXIT, etc. Concretamente, en Monoalgorítmez se utilizan dos punteros generales y otros dos particulares para cada programa. Los primeros son «DPA» (dirección del programa activo) y «PDL» (primera dirección libre); ambos se encuentran en dos palabras reservadas al efecto en el módulo SGM, y su función se entenderá al estudiar la gestión de la memoria en el Apartado 19.6. Por otra parte, delante de cada programa cargado en la MP como respuesta a LD_EX, el sistema de gestión de la memoria pone cuatro bytes con dos punteros que forman lo que llamaremos el **prefijo**. Los dos primeros bytes apuntan al prefijo del padre, y los otros dos contienen el valor que tenía el puntero de pila (R14) tras la ejecución de la BRK incluida en LD_EX. A los módulos cargados con LD_OV no se les añade prefijo: el programa activo sigue siendo el padre (que llama al módulo con CALL).

La Figura 19.2 ilustra estas funciones de DPA, PDL y los prefijos.

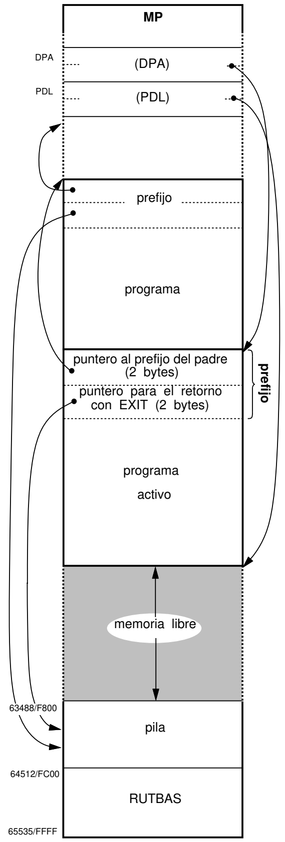


Figura 19.2 Funciones de los prefijos.

19.4. Estructuras de datos

Antes de entrar a describir los algoritmos de las distintas partes del núcleo, veamos las estructuras de datos en las que se apoyan. Es posible que las funciones de algunos de sus componentes no queden muy claras de momento; el lector deberá esperar a la explicación de los algoritmos para terminar de comprenderlas.

La tabla de periféricos

En el Apartado 7.4 hemos adelantado algunos detalles de esta tabla, que eran necesarios para programar la rutina de consulta de interrupciones (RCI). En la Figura 19.3 puede verse su estructura. Tiene dieciséis filas, de las que en principio sólo hay cinco ocupadas (teclado, pantalla, impresora y los dos discos), quedando otras once para posibles adiciones de otros periféricos.

Cada fila consta de veinte bytes, y conviene considerarla formada por tres partes, a las que llamaremos *registro del periférico*, *registro de la demanda* y *registro de la transferencia*.

El **registro del periférico** tiene tres campos, los dos primeros de un byte y el tercero de dos:

- En el primer byte está la *dirección del puerto de estado* del controlador (entre 0 y D'254). Las filas de los periféricos conectados están agrupadas al principio de la tabla. La primera fila libre se señala por el contenido D'255=H'FF en este campo (detalle del que ya habíamos hecho uso en la RCI, Programa 7.3).
- El segundo byte, «*estado*», incluye cuatro informaciones: «operativo» (un bit), «error» (un bit) «ocupado» (un bit) y «tipo» (tres bits).

  El bit «operativo» se refiere a que el periférico esté conectado y listo para funcionar (1) o no (0). Entre las cosas que no habíamos detallado en el módulo INICIA (Programa 18.7) está la comprobación de periféricos. Si el periférico pasa los programas de prueba iniciales, se pone «1» en ese bit, y si no, «0».

  El bit «error» se pone a cero al iniciarse una

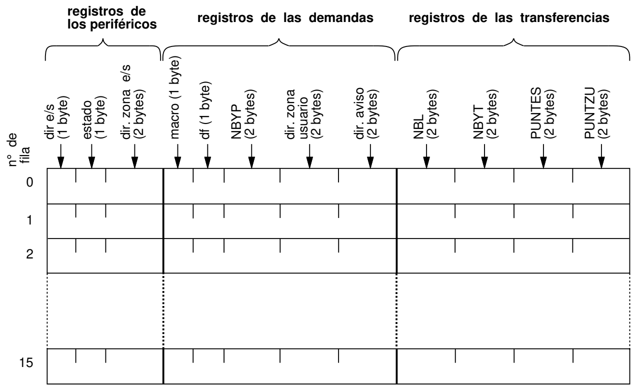


Figura 19.3 Tabla de periféricos.

transferencia, y el gestor del periférico lo cambia a uno si detecta un error en el transcurso de esa transferencia.

El bit «ocupado» es «0» cuando no se está realizando ninguna transferencia con el periférico. Inicialmente, todos los bits «ocupado» de todos los periféricos tienen ese valor. Cuando, como consecuencia de una llamada de lectura o escritura, el sistema inicia una transferencia, pone «1» en el bit del periférico que corresponda. Y cuando la transferencia concluye, la rutina de servicio de la interrupción llama al gestor del periférico, que restaura el «0».

El «tipo» indica las formas de transferencia válidas para ese periférico: 0,1 y 2 corresponden, respectivamente, a un periférico de sólo lectura, de sólo escritura y de lectura y escritura en periféricos de caracteres (teclado, pantalla, impresora, y, en su caso, ratón, módem, etc.); 3, 4 y 5 corresponden a las mismas formas para periféricos de bloques[2].

- El tercer campo, de dos bytes, contiene la *dirección de comienzo de la zona de entrada/salida* correspondiente a ese periférico. Para cada uno de los discos hay una fila, porque cada uno tiene su zona de entrada/salida. Pero como el controlador de ADM es selector (Apartados 5.7 y 7.8), la dirección de puerto es la misma en ambas filas, y el bit «ocupado» siempre tiene el mismo valor en las dos filas.

Obsérvese que casi todas estas informaciones son permanentes: salvo los bits «error» y «ocupado», los valores no cambian en el curso del funcionamiento del sistema. Por el contrario, el **registro de la demanda** asociado a cada periférico tiene contenidos dinámicos: el sistema rellena sus campos cuando en el programa aparece una demanda de entrada/salida (`WRITE`, etc.), trasladando a él las informaciones que aparecen en la macro:

- El primer campo, de un byte, contiene el número (4, 5, 7 u 8) de la *macro que originó la demanda*.
- El segundo campo, de un byte, contiene el *descriptor del fichero*.
- El tercer campo, de dos bytes, contiene *inicialmente* la *cantidad* de bytes a transferir. En la Figura 19.3 este campo se llama NBYP porque va a ir conteniendo, en el curso de la transferencia, el número de bytes *pendientes* de transferir.

[2]En principio, en nuestro sistema sólo tiene sentido «5», pero podría tenerlo también «3» si instalásemos un lector de CD–ROM.

- En otros dos bytes se copia el *puntero a la dirección de comienzo de la zona de usuario*.
- A continuación, y sólo para las demandas originadas por `WRITE_A` y `READ_A`, dos bytes para el *puntero al byte de «aviso»*.

Salvo el tercer campo (NBYP), los contenidos de este registro de la demanda permanecen constantes para una determinada operación de entrada/salida. Por el contrario, los campos del **registro de la transferencia** (y NBYP) pueden ir cambiando mientras el sistema satisface una demanda: como ya adelantamos en el Apartado 18.5 (y luego veremos con más detalle), la transferencia tiene a veces que descomponerse en «trozos», y, junto con NBYP, estos campos contienen informaciones para la transferencia de cada uno de esos trozos:

- NBL es el *número (o dirección física) en el disco* del bloque a leer o escribir (este campo no tiene utilidad para los ficheros especiales de caracteres).
- NBYT es el *número de bytes* a transferir.
- PUNTES contiene un *puntero con una dirección correspondiente a la zona de entrada/salida* del periférico.
- PUNTZU contiene un *puntero con una dirección correspondiente a la zona de usuario* del programa que hizo la demanda.

La tabla de ficheros abiertos

Para todo fichero abierto hay una fila en esta tabla (Figura 19.4) con los siguientes campos:

- El byte llamado «*situación*» contiene tres bits con la codificación del modo de apertura dado en la llamada `OPEN` (Apartado 18.3), cuatro bits con un índice a la tabla de periféricos (si es un fichero en disco, el número de fila correspondiente al disco 0 o al 1; si es un fichero especial de caracteres o de bloques, el número de fila que corresponda al periférico), y un bit que indica si el inodo se ha modificado (al abrir el fichero, el sistema pone «0» en este bit; si se escribe y es un fichero en disco, es posible que crezca y se le añadan bloques, cuyos números se ponen en la copia del inodo; cuando el fichero se cierre habrá que actualizar el inodo residente en el disco, y para señalar este hecho el sistema pone «1» en el bit)[3].

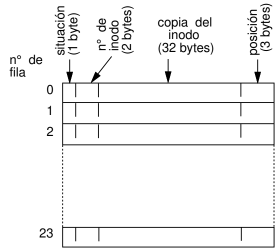


Figura 19.4 Tabla de ficheros abiertos.

- Otros dos bytes con el *número de inodo* en el disco (para poder devolver al disco su copia actualizada, en caso necesario).
- La *copia del inodo* (treinta y dos bytes).
- La *posición* (tres bytes), que se actualiza con las operaciones de lectura y escritura y con la macro `POS` (este campo no se utiliza con los ficheros especiales de caracteres).

El contenido H'FF en el primer byte indica que la fila está libre. Inicialmente sólo están ocupadas las tres primeras: 0 (`tty`), 1 (`A:/`) y 2 (`B:/`). Aunque los directorios no son realmente «ficheros abiertos», sus inodos se mantienen permanentemente en la tabla para hacer más rápida la localización de ficheros[4].

Esta tabla es la estructura básica para el funcionamiento del sistema de gestión de ficheros. El «descriptor de fichero» que el sistema devuelve al abrir un fichero con `OPEN` es el número de fila de la tabla (entre 3 y 23).

[3]En nuestro sistema las modificaciones del fichero se escriben inmediatamente en el disco; en un sistema con memoria oculta de disco (Apartado 19.9) normalmente no es así, y al cerrar el fichero, si se ha modificado, hay que actualizar en el disco no sólo el inodo, también el fichero.

[4]Dadas las limitaciones de nuestro sistema, podríamos mantener en memoria incluso los directorios completos.

```
MODULE RIP
  FROM SGM IMPORT SGM_LD_EX, SGM_LD_OV, SGM_EXIT
  FROM SGF IMPORT SGF_CREATE,SGF_DELETE,SGF_OPEN,SGF_CLOSE,SGF_WRITE,SGF_READ
  FROM SGF IMPORT SGF_WRITE_A, SGF_READ_A, SGF_POS, SGF_STAT, SGF_CHMOD
           EXPORT RIP_ENT
TAB_DIR      DATA   SGF_CREATE    ; tabla de direcciones de
             DATA   SGF_DELETE    ; tratamiento de llamadas
             DATA   SGF_OPEN
             DATA   SGF_CLOSE     ; importadas del sistema
             DATA   SGF_WRITE_A   ; de gestión de ficheros
             DATA   SGF_READ_A
             DATA   TR_WAIT_A     ; definida en este módulo
             DATA   SGF_WRITE
             DATA   SGF_READ
             DATA   SGF_POS
             DATA   SGF_CHMOD
             DATA   SGF_STAT
             DATA   SGM_LD_EX     ; importadas del sistema
             DATA   SGM_LD_OV     ; de gestión de la memoria
             DATA   SGM_EXIT
RIP_ENT   CMP.B  .0,#15
          BNN    NODEF
          CMP.B  .0,#0
          BN     NODEF
          CLR    .13          ; no error, de momento
          SHL    .0           ; 2*(R0) = dirección relativa en TAB_DIR
          ADD    .0,#TAB_DIR
          BR     [/0[.0]]     ; (o BR [[.0++]]): al tratamiento de la llamada
TR_WAIT_A LD.B   .0,/0[.5]    ; (R5) = dirección de "aviso"
          CMP.B  .0,#0
          BNZ    RIP_SAL      ; "aviso" es 1 o -1: retornar
          EI
          WAIT                ; espera interrupción
          DI
          BR     TR_WAIT_A
NODEF     LD.B   .13,#-1      ; error 1: llamada no definida (Tabla 18.2)
RIP_SAL   RETI
        END
```

Programa 19.1 Módulo de la rutina de interrupciones de programa y `WAIT_A`.

19.5. La rutina de interrupciones de programa

La RIP (cargada donde indique el vector de interrupción de `BRK`, que el programa `INICIA` ha puesto en la palabra de dirección D'260) tiene que averiguar qué macro ha originado la interrupción, para bifurcar al módulo que la trata. Tarea muy fácil, teniendo en cuenta que el número de la macro está en el registro R0. Si la macro es `WAIT_A` (número 6), el tratamiento se hace en la misma RIP.

El Programa 19.1 es la codificación en ensamblador de la RIP. Detecta el posible error de que el número no esté comprendido entre 0 y 14, y luego calcula la dirección a la que tiene que ir en función del contenido de R0 y de los contenidos de una tabla de direcciones que se rellena en tiempo de traducción (cuando se ejecuta el montador).

El tratamiento de `WAIT_A` es muy sencillo. R5 contiene la dirección del byte de «aviso». Si su contenido es «1» o «–1», es que la operación que se había iniciado con alguna `READ_A` o `WRITE_A` anterior ya ha terminado, normalmente o con error, y no hay más que retornar con `RETI`. Si no es así, hay que esperar antes del retorno. Pero como estamos dentro del núcleo las interrupciones están inhibidas, y lo único que puede poner «aviso» a «1» es una rutina de servicio de interrupción, la del periférico cuando termine la transferencia. Por tanto, no queda más remedio que hacer una excepción y permitir las interrupciones con `EI`. Hecho esto, se pone a la UCP en el estado de espera (con la *instrucción* `WAIT`); cuando aparece una interrupción se comprueba nuevamente (tras ejecutarse la rutina de servicio que corresponda) si «aviso» ha cambiado, y si no es así se vuelve a esperar (la interrupción habrá sido por otra causa).

19.6. El sistema de gestión de la memoria

Sin memoria virtual ni mecanismos de protección, nuestro sistema de gestión de la memoria es casi trivial. Sólo tiene que ocuparse de tres llamadas: `EXIT` y las de carga de programas (`LD_EX` y `LD_OV`). El Programa 19.2 contiene todo el módulo. Parece conveniente añadir algunos comentarios a los que aparecen en el listado, para facilitarle al lector el análisis del programa:

- PDL («primera dirección libre») tendrá inicialmente un valor que habrá puesto el módulo `INICIA`. Este valor es la dirección siguiente a la última del núcleo. Para averiguarla, `INICIA` lee el primer bloque asignado al fichero `sistema` y obtiene su longitud de la cabecera.
- Para `LD_EX` no hay que salvar registros: se hace en la expansión de la macro (Apartado 18.6). Primero se llama a un subprograma, «`BUSCAFICH`», cuya función consiste en buscar el nombre del programa (cadena de caracteres apuntada por R2, que empezará por `A:` o `B:`) en el directorio del disco y traer su inodo a `INODO`. Como el directorio reside sólo en el disco, este subprograma, teniendo en cuenta las informaciones del inodo del directorio de `A:` o de `B:`, llamará a su vez a las rutinas que permiten leer bloques (y aquí se pueden producir los errores 2 o 3, que `BUSCAFICH` devuelve en R13) para ir explorando este directorio. Si el nombre no se encuentra en ninguna fila *ocupada* (las filas ocupadas son las que tienen en los dos primeros bytes un contenido distinto de H'FFFF), `BUSCAFICH` devuelve -4 en R13.
- Antes de cargar el programa en la MP hay que ver si cabe. El inodo contiene la longitud del fichero, pero esto no nos sirve, porque, debido al formato (que incluye sumas de prueba, diccionario de reubicación, etc.), esta longitud es mayor que la del código a cargar. El subprograma `LONGEJEC` se ocupa de comprobar si se puede ejecutar y de averiguar la longitud. Examinando el inodo, si no tiene permiso de ejecución pone -5 en R13 y termina. Si lo tiene, lee el primer bloque asignado al fichero, y si el número mágico no es H'0707 pone -6 en R13 y termina. Si es ejecutable, lee la longitud del código (contenida también en la cabecera del fichero, Apartado 6.8) y si es mayor que 64 KB pone -7 en R13 (seguro que no va a caber). Si se han pasado todas las comprobaciones con éxito, devuelve la longitud en R8.
- Para asegurarnos de que el programa cabe en la memoria disponible hemos de tener en cuenta el valor de PDL: la suma de los contenidos de PDL y R8 no puede superar H'F7FF-4 (el «4» es por los cuatro bytes del prefijo). La comprobación hay que hacerla con la instrucción `BC` (bifurca si acarreo), porque estamos trabajando con números de dieciséis bits sin signo.
- Si todo es correcto, se actualizan DPA y PDL y se llama a `CARGAPROG`, un cargador reubicador que toma como dirección de carga (constante de reubicación) el contenido de R10. En el proceso de carga pueden aparecer los errores 2, 3 y 8. Si termina sin error, devuelve la dirección de ejecución (que se encuentra al final del módulo de carga, Apartados 6.8 y 18.8) en R11. Esta dirección se introduce en la pila, y, sobre ella, el valor inicial H'0100 para el registro de estado: así, al salir con `RETI` se ejecutará el programa cargado en modo usuario y

```
MODULE SGM
          FROM RUTINAS IMPORT BUSCAFICH, LONGEJEC, CARGAPROG
          EXPORT SGM_EXIT, SGM_LD_EX, SGM_LD_OV
  R_EST       EQU   H'0100; RE inicial: SUP = RAS = 0, PIN = 1, H = C = V = N = Z = 0
  INODO       RES.B 32    ; zona temporal para un inodo
  DPA         RES   1     ; dirección del programa activo
  PDL         RES   1     ; primera dirección libre
; punto de entrada de LD_EX
  SGM_LD_EX    LD.B   .12,#INODO ; pasa dirección a BUSCAFICH, LONGEJEC y CARGAPROG
               CALL /BUSCAFICH   ; pone inodo del fichero del programa en INODO
               CMP.B  .13,#0     ; posibles errores (2, 3, 4) en R13
               BNZ    EX_SAL
               CALL   /LONGEJEC  ; longitud código → R8
               CMP.B  .13,#0     ; posibles errores (2, 3, 5, 6, 7) en R13
               BNZ    EX_SAL
               ADD    .8,PDL
               ADD    .8, #4
               CMP    .8,#H'F800
               BC     EX_NOCABE  ; (PDL)+longitud+4 >= D'63.488 (=H'F800)
               LD     .9, DPA
               LD     .10, PDL
               ST     .9, /0[.10]; (DPA) → (PDL): pone puntero a prefijo anterior
               ST     .14,/2[.10]; (PP) → (PDL)+2: pone (PP) en el prefijo
               ST     .10, DPA   ; (PDL) → DPA: actualiza DPA
               ST     .8, PDL    ; (PDL)+longitud+4 → PDL
               ADD    .10,#4     ; dirección de carga a R10
               CALL   /CARGAPROG ; dirección ejecución → R11
               CMP.B  .13,#0     ; posibles errores (2, 3, 8) en R13
               BNZ    SGM_EXIT   ; deshace lo hecho con DPA y PDL
               LD     .12,#R_EST
               PUSH   .11        ; apila dirección ejecución
               PUSH   .12        ; apila RE inicial
  EX_SAL      RETI
  EX_NOCABE    LD.B   .13,#-7    ; error 7: espacio insuficiente en la MP
              RETI
; punto de entrada de LD_OV
  SGM_LD_OV    PUSH   .8
               PUSH   .10
               PUSH   .11
               PUSH   .12
               LD.B   .12,#INODO ; pasa dirección a BUSCAFICH, LONGEJEC y CARGAPROG
               CALL   /BUSCAFICH ; pone inodo del fichero del programa en INODO
               CMP.B  .13,#0     ; posibles errores (2, 3, 4) en R13
               BNZ    OV_SAL
               CALL   /LONGEJEC
               CMP.B  .13,#0     ; posibles errores (2, 3, 5, 6, 7) en R13
               BNZ    OV_SAL
               ADD    .8,PDL
               CMP    .8,#H'F800
               BC     OV_NOCABE  ; (PDL)+longitud >= D'63.488 (=H'F800)
               LD     .10, PDL
               CALL   /CARGAPROG ; si error (2, 3, 8), lo devuelve en R13
  OV_SAL       POP    .12
               POP    .11
               POP    .10
               POP    .8
              RETI
  OV_NOCABE    LD.B   .13,#-7
               BR     OV_SAL
; punto de entrada de EXIT
  SGM_EXIT     LD     .8, DPA
               ST     .8, PDL    ; (DPA)→ PDL: la nueva PDL es la antigua DPA
               LD     .14,/2[.8] ; ((DPA)+2)→ PP: recupera el PP que dejó LD_EX
               LD     .8, /0[.8]
               ST     .8, DPA    ; ((DPA))→ DPA: el nuevo programa activo es el padre
              RETI
            END
```

Programa 19.2 Sistema de gestión de la memoria.

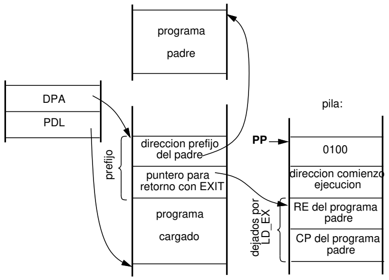


Figura 19.5. Inmediatamente antes de la `RETI` con la que finaliza `SGM_LD_EX`.

con las interrupciones permitidas (Figura 19.5).

- El tratamiento de `LD_OV` es más sencillo: como el control se devuelve al programa padre, y el hijo no tiene prefijo, la operación termina simplemente cargando el programa a partir de la primera dirección libre. Como `PDL` no se actualiza, si en el programa padre (o en el propio hijo) se ejecuta otra `LD_OV` (o una `LD_EX`) el nuevo programa se carga sobre la zona asignada a éste («machacándolo»).
- Para `EXIT` el sistema hace dos cosas: libera la memoria (actualizando DPA y PDL) y se ocupa de restaurar el valor del puntero de pila (R14), guardado en el prefijo, para que `RETI` recupere el registro de estado del padre y la dirección de retorno (la de la instrucción siguiente a la `BRK` en la llamada `LD_EX` del padre) (Figura 19.6).

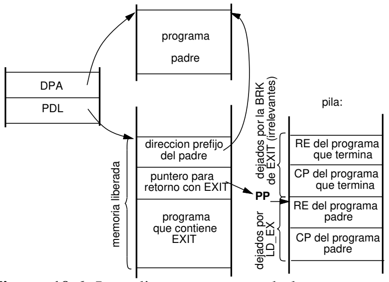


Figura 19.6. Inmediatamente antes de la `RETI` con la que finaliza `SGM_EXIT`.

19.7. El sistema de gestión de ficheros

El sistema de gestión de ficheros atiende a las llamadas para creación, borrado, apertura, cierre (0–3), lectura, escritura (4, 5, 7 y 8), posicionamiento, cambio de permisos y petición de información (9–11). Veamos, en líneas generales, las operaciones que hay que hacer para cada una. El lector puede ir comprobando que estas operaciones permiten satisfacer a las especificaciones funcionales que dimos en la Lección anterior.

Creación y borrado

Para crear un fichero, el SGF comprueba previamente que el nombre es correcto (véase la nota al pie de la página 520) (si no, –18→R13) y que el dígito de permisos (contenido en R6) está comprendido entre 0 y 7 (si no, –10→R13). A continuación busca el nombre en el directorio, con la misma función `BUSCAFICH` que utiliza el SGM.

Si el fichero ya existe (`BUSCAFICH` devuelve (R13) = 0 y el inodo en una zona de treinta y dos bytes reservada en el SGF), la «creación» consiste en modificar el inodo, poniendo «0» en longitud, los permisos (están en R6), y la hora actual. Además, libera los bloques que el fichero tenía asignados, poniendo «0» en los lugares correspondientes del mapa de bits de bloques[5]. Luego continúa como si se tratase de `OPEN` con modo escritura.

Si el fichero no existe ((R13) $= -4$), el SGF busca primero un inodo libre, buscando un «0» en el mapa de bits del superbloque cargado en memoria. Si no lo encuentra, la operación termina poniendo -9 en R13. Si el «0» encontrado está en el bit n, lo cambia a «1», escribe en el inodo de número n la longitud (0), el modo (R6) y la hora, y escribe en la primera fila libre del directorio el número de inodo (n) y el nombre del fichero. Ac-

[5] En la descripción funcional decíamos que la operación de creación sobre un fichero existente «borra» el contenido del fichero. Ahora vemos que no es que se borre nada físicamente, pero el efecto es el mismo.

tualiza en el disco el mapa de bits de inodos que acaba de modificar y continúa como `OPEN` con modo escritura.

Borrar un fichero requiere menos comprobaciones. Si el nombre es correcto, pasa a explorar (con `BUSCAFICH`) las filas activas del directorio para comprobar que el nombre existe (si no, $-4 \rightarrow$ R13). Una vez encontrado, el SGF libera los bloques que aparecen en el inodo devuelto por `BUSCAFICH`, poniendo a «0» los bits de mapa de bloques (si el fichero ocupa más de diez bloques tendrá que traer del disco el bloque indirecto para seguir liberando bloques). Libera el inodo (poniendo «0» en el bit del mapa de inodos) y desactiva la fila del directorio (poniendo H'FFFF en los dos primeros bytes). Finalmente, copia el superbloque actualizado en el disco.

Apertura y cierre

Las operaciones para la macro `OPEN`, y los posibles errores son:

- Comprobar que el nombre está en el directorio. En caso contrario, señalar el error: $-4 \rightarrow$ R13.
- Comprobar que hay coherencia entre el modo de `OPEN` y los permisos que figuran en el inodo. Si no la hay, $-11 \rightarrow$ R13.
- Comprobar que hay una fila libre en la tabla de ficheros abiertos, explorando el primer byte de cada fila hasta encontrar H'FF. Si no se encuentra, $-12 \rightarrow$ R13.
- Construir la fila en la tabla de ficheros abiertos: situación («modo», dado por R6 en la expansión de la macro, índice a la tabla de periféricos, determinado por el nombre del fichero, y «0» en el bit «modificado»), número de inodo, copia del inodo (que se lee del disco), *posición* = 0 si el modo es 0, 1 o 2 o *posición* = *longitud* (que figura en el inodo) si modo es 3 o 4 (Apartado 18.3).
- Poner en R13 el número de fila asignada en la tabla (que es el descriptor del fichero).
- Salvo que se trate de un fichero especial (lo que está registrado en el byte «tipo» del inodo), si su longitud es mayor que cero (la longitud también se encuentra en el inodo) y el modo es 0, 1 o 2, empezar la lectura del primer bloque que tenga asignado en el disco hacia la zona de entrada/salida (para ello deberá rellenar los campos adecuados de la tabla de periféricos y llamar al gestor del periférico, como veremos al estudiar el tratamiento de las llamadas de lectura).

Para `CLOSE`, primero hay que comprobar que el descriptor del fichero (contenido en R2), DF, realmente está en uso (el primer byte de la fila de número DF en la tabla de ficheros abiertos debe tener un valor distinto de H'FF). Si es así, se señala como libre (poniendo H'FF en ese byte), y si no, $-13 \rightarrow$ R13. Adicionalmente, y si el fichero ha sido modificado, deberá actualizarse su inodo en el disco; para comprobarlo, el sistema no tiene más que mirar el valor del bit «modificado» (en el byte «situación» de la fila DF): si es «1» entonces escribe en su lugar del disco la copia del inodo.

Comprobaciones previas para las macros de transferencia

En el tratamiento de las cuatro macros que sirven para las transferencias de entrada/salida (`WRITE_A`, `READ_A`, `WRITE` y `READ`) hay tres comprobaciones previas comunes:

- Que el descriptor del fichero está en uso (si no, $-13 \rightarrow$ R13).
- Que la operación solicitada no es incompatible con el modo dado en la apertura del fichero, información que se encuentra en el byte «situación» de la fila de número DF de la tabla de ficheros abiertos (si lo es, $-14 \rightarrow$ R13).
- Que el periférico que corresponde a ese fichero (el índice a la tabla de periféricos también se encuentra en el byte «situación») está en funcionamiento («operativo»=1 en el segundo byte del registro del periférico). De no ser así, $-2 \rightarrow$ R13.

En el caso de `WRITE_A` y de `WRITE`, y si se trata de un fichero en disco, este fichero puede crecer. El sistema debe comprobar previamente que no va a superar el límite de 133 KB, y que hay suficientes bloques libres en el disco, teniendo en cuenta:

- El número de bytes a escribir, `NB` (para `WRITE_A`, 512; para `WRITE`, el contenido de R6).
- Los valores actuales de *longitud*, l (en la copia del inodo) y de *posición*, p (como el anterior, en la tabla de ficheros abiertos): si $p+\texttt{NB} \leq l$ (en el caso de `WRITE`), o si $p < l$ (en el caso de `WRITE_A`), no hay problema (se va a escribir en los mismos bloques ya ocupados), pero si no es así el sistema tiene que calcular el número de bloques adicionales necesarios, comprobar que la longitud resultante no supera 133 KB, y considerar también:
- El mapa de bits de bloques libres (en la copia del superbloque).

Si no hay espacio suficiente en el disco se termina la operación poniendo –16 en R13. Si la longitud resulta superior a 133 KB, en el caso de `WRITE_A` se termina también, con –15 en R13, pero en el caso de `WRITE` se trunca: el sistema sustituye el contenido de R6 por $133 \times 1.024 - p$, realiza la transferencia y devuelve ese valor por R12.

Por último, cuando se trata de `READ_A` y `READ` hay que comprobar que *posición*<*longitud*. De no ser así no se genera ningún error, pero se devuelve «0» en R12 (número de bytes leídos) y se termina la operación.

Si todas las condiciones se cumplen, el sistema de gestión de ficheros continúa de acuerdo con el tipo de llamada de que se trate.

Preparación de la transferencia

Pasadas con éxito las comprobaciones previas, el sistema de gestión de ficheros debe trasladar las informaciones contenidas en la expansión de la macro al registro de la demanda. R3 contiene el descriptor del fichero, que sirve como índice (número de fila) a la tabla de ficheros abiertos, donde se encuentra un índice a la tabla de periféricos, la copia del inodo, y el valor de la posición, p.

Si la posición p es tal que $(p/512) < 10$, el número de bloque a leer o escribir en el disco se obtiene directamente del inodo; en caso contrario, hay que leer previamente el bloque indirecto para obtener el número de bloque al que hay que acceder. Con esto, ya se pueden dar valores a algunos campos de los registros de la tabla de periféricos (compruébelo, consultando las expansiones de las macros y la Figura 19.3):

- (R0)→«macro».
- (R3)→«df».
- (R4)→«dir. zona usuario».
- Número de bloque→«NBL».
- Si `READ_A` o `WRITE_A`, (R5)→«aviso» y 512 → «NBYP».
- Si `READ` o `WRITE`, (R6)→«NBYP».

Las operaciones que el sistema debe realizar a continuación son diferentes según se trate de un fichero especial («`tty`», «`fd0`», etc.) o no (formado por bloques de un disco cuyos números se encuentran en el inodo del fichero). La abstracción que ofrece el sistema de ficheros a los programas de aplicaciones implica alguna complejidad del sistema (transparente a los programas), especialmente para el tratamiento de los ficheros ordinarios. *En adelante nos referimos sólo a éstos* (ficheros residentes en disco).

Esta abstracción tiene dos consecuencias:

- Los programas «ven» *objetos abstractos* llamados «ficheros», consistentes en secuencias lineales de bytes, mientras que internamente cada fichero está formado por un conjunto de bloques de 512 bytes desperdigados por el disco (veremos un ejemplo algo más adelante, en la Figura 19.8). La correspondencia entre el nombre de un fichero y la localización física en el disco se establece, como ya sabemos, a través de dos estructuras de datos: el directorio y el inodo.
- Con las llamadas `READ` y `WRITE`, los programas realizan *operaciones de acceso abstractas*: leer o escribir un número arbitrario de bytes a partir de una posición también arbitraria (que puede modificarse con la macro `POS`). Sin embargo, las únicas operaciones que el sistema de gestión de ficheros

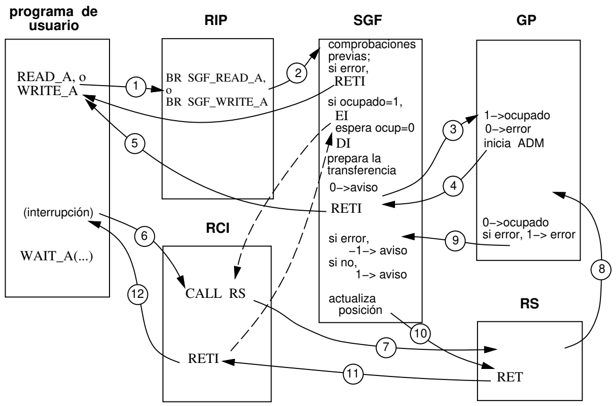


Figura 19.7 Transferencias de control para el tratamiento de `READ_A` y `WRITE_A`.

puede hacer son leer y escribir un bloque del disco dando su número (entre 0 y 2.879)[6]. El sistema de gestión de ficheros ha de ocuparse de traducir las «demandas abstractas» de las llamadas a «demandas concretas» para el gestor del disco.

Tratamiento de `READ_A` y `WRITE_A`

El procesamiento de estas llamadas es relativamente sencillo. De hecho, la principal justificación[7] para haberlas incluido en el modelo funcional de Monoalgorítmez es el estudiarlas aquí como introducción a las que normalmente se utilizarían, `READ` y `WRITE`, cuyo modelo procesal es bastante más complejo.

La Figura 19.7 muestra las transferencias de control que tienen lugar, en orden cronológico, desde que se inicia el tratamiento de la llamada hasta que la demanda ha quedado satisfecha, suponiendo que esto ocurre antes de que en el programa de usuario se ejecute `WAIT_A` (de no ser así, el programa no habría pasado de `WAIT_A`: habría entrado en la RIP y en ésta se estaría ejecutando el bucle del Programa 19.1).

El SGF comprueba primero que el periférico no está ocupado. Si lo está, se queda en un bucle de espera similar al del tratamiento de `WAIT_A` en la RIP, sólo que esperando a «ocupado» = 0. Después, se limita a llamar al gestor del disco pertinente, comunicándole el número de bloque, la operación a realizar (leer o escribir, dependiendo del número de la macro el en registro de la demanda) y la dirección de la zona de usuario (recuerde que con estas llamadas el usuario tiene siempre que reservar una zona de 512 bytes)[8].

El gestor pone «1» en el bit «ocupado» y «0» en el bit «error» del campo «estado» del registro del periférico. Luego calcula, a partir del número de bloque, la superficie, número de pista y número de sector en el disco, y finalmente pone valores iniciales en el controlador de ADM (Apartado 7.8, Figura 7.5):

- 256→CDP
- dirección de la zona de usuario→DEC
- (disco,superficie,pista,sector)→LOC
- si lectura, H'0E→EP; si escritura, H'06→EP

Finalmente, sale del núcleo con `RETI`, volviendo al programa de usuario cuya ejecución continúa concurrentemente con la transferencia.

Al final de la transferencia se produce la interrupción del controlador de ADM (el controlador pone «1» en el bit PR del registro EP). Su rutina de servicio llama al gestor, que pone «0» en «ocupado», mira si ha habido algún error (leyendo el registro EP) y pone, según sea el caso, «0» o «1» en el bit «error». El gestor pasa al SGF, comunicándole el número del periférico. Si ha habido error, el SGF (que sabe que la operación que ha terminado era debida a una `READ_A` o una `WRITE_A` por el valor del campo «macro» en el registro de la demanda) pone -1 en el byte *aviso* del programa de usuario (cuya dirección está también en el registro de la demanda) y regresa a la rutina de servicio, que volverá a la rutina de consulta de interrupciones, y ésta al programa de usuario (si no se pasa por la RCI, entonces la RS vuelve directamente con `RETI`). Si no había error, pone 1 en el byte *aviso* y actualiza el valor de *posición* en la tabla de ficheros abiertos (la fila de la tabla la conoce por el valor de «df» en el registro de la demanda). Como el campo «macro» contiene «5» (`READ_A`) o «6» (`WRITE_A`), *posición* tiene que quedar apuntando al primer byte del bloque siguiente, es decir, su valor debe actualizarse así:
((*posición* mód 512)+1)×512→*posición*.

El caso de `READ` y `WRITE`

Las funciones que proporcionan estas llamadas son de un nivel más alto, y, en consecuencia, su tratamiento es más complicado. Recordemos las diferencias funcionales con respecto a `READ_A` y `WRITE_A`:

- El programa de usuario puede pedir la transferencia de un número arbitrario de bytes (siempre que sea menor que 32 KB) entre un fichero y una zona reservada en el programa, que ya no tiene que ser obligatoriamente de 512 bytes.
- La sincronización del procesamiento de los datos con el final de una transferencia iniciada anteriormente es transparente al programa de usuario. Cuando éste pide una lectura, puede estar seguro de que al ejecutarse la instrucción siguiente los datos leídos ya están en la zona reservada para ellos. Y si pide una escritura de datos que ha puesto previamente en la zona, no tiene que preocuparse de esperar para volver a rellenar la zona con otros datos.

Para hacer posible ambas cosas, existen zonas de entrada/salida transparentes al programa de usuario (que no se utilizan en el procesamiento de `READ_A` ni en el de `WRITE_A`).

Como antes, vamos a considerar sólo los ficheros ordinarios. Para cada disco hay una zona de entrada/salida de 512 bytes. Lo único que «sabe hacer» el gestor del disco es transferir el bloque de número N a esta zona (o a la que se le diga, pero siempre una zona de 512 bytes), o transferir de la zona al bloque N. Por su parte, la zona de usuario puede ser menor, igual o mayor que 512 bytes, y una operación de lectura o escritura puede involucrar varios bloques aun en el caso de que el número de bytes sea menor que 512. Por ejemplo, consideremos un fichero abierto, de descriptor `DF`, cuyo tamaño en un momento determinado es 2 KB, que tiene asignados los bloques 400, 500, 800 y 900. Si en el programa de usuario aparece `POS(DF,#1000,#0)` seguida de `READ(DF,#ZONAUSU,#100)` (es decir, se pide la lectura de cien bytes del fichero a partir del byte 1.000), el SGF deberá tener en cuenta que los primeros 24 bytes a leer se encuentran en el bloque 500 y los 76 restantes en el bloque 800. Tendrá

[6] En realidad, puede leerse o escribirse un número cualquiera de bytes comprendido entre 0 y 65.535, dando el valor inicial adecuado al registro CDP del controlador de ADM (Apartado 7.8). Pero el sistema de gestión de ficheros accede al disco a través del gestor, y éste está programado para leer y escribir únicamente bloques.

[7] Otra justificación: con ellas se puede controlar mejor desde los programas de usuario la concurrencia entre operaciones de procesamiento en la UCP y operaciones de ADM y aumentar la velocidad, véase una nota sobre el tratamiento de `READ`, más adelante en este mismo Apartado.

[8] Observe también que `OPEN` habrá comenzado, y quizás terminado, la lectura del primer bloque del fichero, pero `READ_A` no lo usa: estúpidamente, inicia siempre la lectura en el disco. (`READ` *sí* lo usa).

pues que hacer lo siguiente:

1. Pedirle al gestor que lea el bloque 500; terminada la operación, copiar los 24 últimos bytes de la zona de entrada/salida a los 24 primeros de la zona de usuario.
2. Pedirle al gestor que lea el bloque 800; terminada la operación, copiar los 76 primeros bytes de la zona de entrada/salida a la zona de usuario, a continuación de los 24 anteriores.

Por tanto, a veces el sistema de gestión de ficheros tiene que descomponer la demanda en «trozos». Las informaciones necesarias para la transferencia de cada trozo son las que se introducen en el registro de la transferencia (Figura 19.3).

Tratamiento de `READ`

Como ya sabemos de la Lección anterior, además del uso de zonas de usuario de longitud arbitraria y de dejar la sincronización al cuidado del sistema operativo, hay otra diferencia entre `READ_A` y `READ`: el mecanismo de **anticipación**.

Para `READ_A`, el sistema de gestión de ficheros, después de esperar, en su caso, a que el periférico quede libre, llama *siempre* al gestor para que inicie la transferencia (del bloque especificado en el registro de la transferencia a la zona de 512 bytes reservada en el programa de usuario). Pero si se trata de `READ`, el sistema ha aplicado antes ese «algoritmo de adivinación», mediante el cual «presagia» cuál será el próximo bloque a leer y, si no es el mismo que hay copiado en la zona de entrada/salida, lo transfiere del disco a esa zona. Si el presagio ha sido correcto se gana tiempo: el sistema copia de la zona de entrada/salida a la de usuario (como ambas están en la MP es mucho más rápido que pedir la lectura en el disco y esperar), aplica de nuevo el algoritmo y, en caso necesario, le pide al gestor que inicie la transferencia del nuevo bloque presagiado. El programa puede así continuar con los datos correctos leídos previamente, mientras el probable bloque siguiente se está leyendo con ADM. Naturalmente, el algoritmo puede fallar (especialmente, cuando antes de `READ` hay una `POS`), y en ese caso el sistema tendrá que pedir la lectura del bloque correcto para que sustituya al fallido y esperar.

Este algoritmo no tiene nada de misterioso. Ya lo hemos esbozado en el Apartado 18.5. Concretando un poco más:

- Cuando un fichero se abre con `OPEN`, si tiene algún contenido (es decir, si *longitud*>0 en el inodo) se inicia la transferencia de su primer bloque a la zona de entrada/salida.
- Cuando termina una transferencia iniciada con `READ`, *posición* queda con un determinado valor. Lo más probable es que la siguiente `READ` quiera leer a partir de esa posición. Entonces, el bloque «presagiado» es aquél dentro del cual se encuentra el byte indicado por *posición*: si en la última transferencia el último byte estaba justamente al final de un bloque, *posición* apunta al primer byte del siguiente bloque del fichero (cuyo número se encuentra en el inodo), y el sistema inicia su lectura; pero en otro caso (*posición* ha quedado en medio de un bloque) lo más probable es que se vaya a seguir leyendo del mismo bloque, por lo que no se hace ninguna lectura nueva.
- Cuando termina la transferencia de un «trozo» (pero no la transferencia total) es seguro que el próximo bloque a leer es el siguiente de los asignados al fichero.

Para implementar este algoritmo, en el gestor de cada disco hay dos palabras reservadas con las siguientes funciones:

- NBZ: número del bloque actualmente presente en la zona de entrada/salida (`OPEN` introduce aquí el número del primer bloque, si éste existe, es decir, si *longitud*>0, e inicia su lectura).
- NBS: número del bloque, según el inodo, siguiente al último leído, si existe.

Un ejemplo

Sigamos detenidamente las operaciones con un ejemplo: un fichero llamado «`XFile`» que ocupa 1.200 bytes y que tiene asignados los bloques 2.502, 1.287 y 420. La Figura 19.8 ilustra la correspondencia entre el objeto abstracto «`Xfile`» y el espacio físico en el disco, establecida a través del inodo que corresponde a ese fichero.

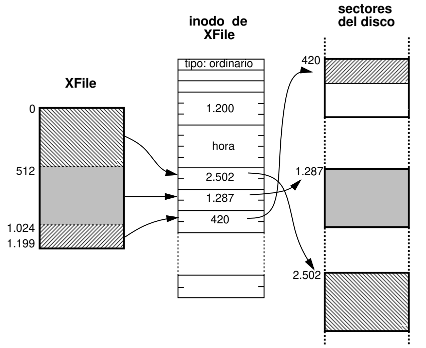


Figura 19.8. Ejemplo de asignación de bloques en disco a un fichero.

Vamos a analizar las operaciones que tienen lugar cuando se ejecuta un programa en el que aparecen las siguientes seudoinstrucciones, instrucciones y macroinstrucciones:

```
        ---
NOMFICH DATA.B  "A:XFile",0
ZU1     RES.B   500
ZU2     RES.B   600
DFICH   RES.B   1
        ---
        OPEN(#NOMFICH,#2)
        ST.B    .13, DFICH
        ---
        READ(DFICH,#ZU1,#500)
        ---
        READ(DFICH,#ZU2,#600)
        ---
```

Para tratar `OPEN`, el sistema de gestión de ficheros busca una fila libre de la tabla de ficheros abiertos y pone en ella: el modo (2: lectura y escritura); el índice a la tabla de periféricos (disco `A`); «0» en el bit «modificado»; el número de inodo de `XFile` (obtenido del directorio), la copia del inodo (leída del disco) y *posición* = 0. Luego le pide al gestor del periférico que inicie la transferencia del primer bloque asignado al fichero hacia la zona de entrada/salida del disco `A`, pone NBZ = 2.502 y NBS = 1.287, y retorna al programa de usuario devolviendo el número de la fila de la tabla de ficheros abiertos (el descriptor) en R13.

Tratamiento de la primera `READ`

El siguiente resumen puede irse comprobando con la descripción gráfica de la Figura 19.9.

Al interpretar `READ(DFICH,#ZU1,#500)`, el sistema (tras pasar con éxito las comprobaciones previas) consulta en la tabla de ficheros abiertos la fila determinada por `DFICH`, que le lleva a localizar también la fila oportuna de la tabla de periféricos, y rellena en ésta el registro de la demanda con las informaciones que le da la expansión de la macro. Como *posición* = 0 y se pide la transferencia de 500 bytes, la operación sólo requiere un bloque del disco. Pero primero ha de asegurarse de que el disco no está ocupado. Suponiendo que aún lo está, porque no ha terminado la transferencia iniciada con `OPEN`, espera a una interrupción. Cuando se produce la interrupción del ADM, hay una sucesión de acontecimientos que puede seguirse en la figura, entre los puntos 3 y 8. Resumiendo, el gestor pone «0» en ocupado (suponiendo que no ha habido error), y el SGF puede continuar. Rellena el registro de la transferencia del siguiente modo: NBL = 2.502 (número del primer bloque, que lo encuentra en el inodo), NBYT = 500 (transferir 500 bytes), PUNTES = dir._zona_e/s (transferir desde el principio de la zona de entrada/salida) y PUNTZU = dir._zona_usuario (hacia el principio de la zona de usuario). Como NBL = NBZ, el bloque pedido ya está en la zona de entrada/salida; el SGM no tiene más que copiar los 500 primeros bytes en la zona de usuario, con lo que la demanda ha quedado satisfecha. El nuevo valor de *posición* es 500: como se ha quedado en mitad de un bloque (es decir, (*posición* mód 512) $\neq 0$), no hace nada más (en caso contrario, pediría al gestor la lectura del bloque cuyo número está en NBS).

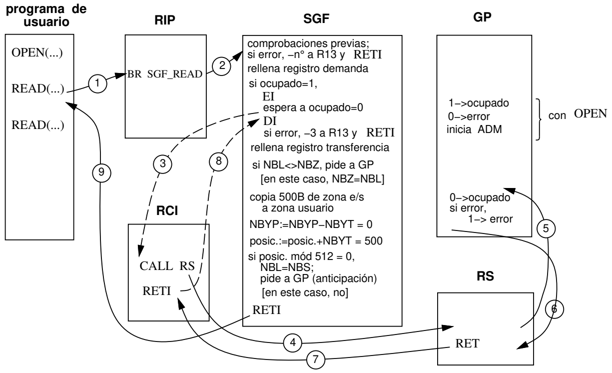
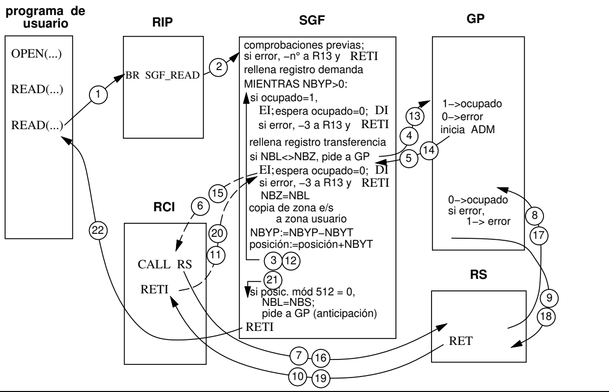


Figura 19.9 Procesamiento de `READ(DFICH,#ZU1,#500)`.



	NBYP	NBL	NBYT	PUNTES	PUNTZU
2–3	600→588	2.502	12	dir._zona e/s+500	dir._zona_usuario
4–12	588→76	1.287	512	dir._zona e/s	dir._zona_usuario+12
13–21	76→0	420	76	dir._zona_e/s	dir._zona_usuario+524

Figura 19.10 Procesamiento de `READ(DFICH,#ZU2,#600)`.

Tratamiento de la segunda `READ`

El tratamiento de la `READ(DFICH,#ZU2,#600)` del ejemplo es aún más laborioso, porque requiere la lectura de *tres* bloques del fichero. En efecto, como *posición* ha quedado con el valor 500, los doce primeros bytes a transferir están en el bloque 2.502 (que está ya en la zona de entrada/salida), los 512 siguientes en el bloque 1.287 y los 76 restantes en el bloque 420. No cansaremos al lector con una nueva descripción verbal; en su lugar le pedimos que dedique unos minutos a analizar la Figura 19.10, en la que puede comprobar que el bucle «mientras NBYP > 0» se recorre tres veces.

En la primera no se pide nada al gestor, porque los doce primeros bytes requeridos ya están en memoria, y no hay más que copiarlos en los doce primeros bytes de la zona de usuario. Luego hay que pedir la transferencia del bloque 1.287 y copiar sus 512 bytes a continuación de los anteriores en la zona de usuario. Y finalmente, pedir la transferencia del bloque 420 y copiar sus 76 primeros bytes. En la tabla de la parte inferior de la figura pueden verse los valores que el sistema de gestión de ficheros va introduciendo en NBYP y en el registro de la transferencia. Los números de la primera columna corresponden a rótulos de las flechas de la figura.

Observe que no se devuelve el control al programa de usuario hasta que se ha satisfecho la demanda. Es obvio que no puede procederse de otra manera si se quiere cumplir con el modelo funcional: tras la ejecución de `READ`, los datos pedidos han de estar en la zona de usuario[9].

Hemos omitido en esta explicación algunos detalles, pero no serían difíciles de incluir. Por ejemplo, el SGF debe comprobar que no se intenta leer más allá del final del fichero (mirando los valores de *posición*, *longitud* y `NBYP`). De no ser así, reduce el valor de `NBYP` pedido por el programa de usuario. Y, en todo caso, tiene que devolver (en R12) el número de bytes leídos.

Tratamiento de `WRITE`

Si se ha conseguido captar la idea de los procesos que ocurren en la lectura, la comprensión de la escritura no requiere mucho esfuerzo adicional: las operaciones son las mismas, pero «al revés» (de la zona de usuario a la de entrada/salida, y de ésta al disco). Hay, desde luego, algunas particularidades a considerar:

- Si resulta *posición* $+$ NBYP $>$ *longitud*, el fichero crece. Será preciso (además de detectar el posible error de número 16) asignar nuevos bloques, poner sus números en el inodo (o en el bloque indirecto apuntado por el inodo) y modificar el valor de *longitud* (en el inodo) y el mapa de bits de bloques libres (en el superbloque).
- Antes de copiar de la zona de usuario a la de entrada/salida, si NBYT < 512 *primero hay que leer el bloque del disco* en la zona de entrada/salida, luego copiar los NBYT bytes en el sitio oportuno y entonces ya se podrá llamar al gestor para que escriba el bloque.
- Una vez que se ha copiado el último (o el único) trozo de la zona de usuario a la de entrada/salida y se ha iniciado al gestor, se puede volver al programa de usuario.
- Al terminar la escritura del último (o único) trozo se deberá poner el valor actual de la hora en el inodo.

[9] A un programador de aplicaciones podría interesarle más utilizar la llamada `READ_A` si quiere minimizar el tiempo de ejecución de su programa, pero entonces tendría que ocuparse de todas las operaciones de descomposición en «trozos».

Cambio de posición, cambio de modo y petición de información

Comparados con la complejidad de los procesos que hemos visto, los tratamientos de estas llamadas son francamente triviales.

Para `POS`, tras las comprobaciones previas (posibles errores 13 y 17), se calcula el nuevo valor de *posición* en función del actual (en la fila de la tabla de ficheros abiertos determinada por el contenido de R3), del desplazamiento (en R6) y del valor de k (en R7), de acuerdo con lo dicho en el Apartado 18.5. Si el valor resultante de *posición* es mayor que H'21377 (133 KB), se devuelve -15 en R13, y si no, se pone en la fila de la tabla de ficheros abiertos ese nuevo valor y se devuelve por (R13,R12).

Para `CHMOD` las comprobaciones previas son sobre los posibles errores 4 y 10. Si no se da ninguno, se lee el inodo del disco (con el número que ha devuelto la rutina `BUSCAFICH`, pero si el fichero está abierto no es necesario leerlo), se pone el nuevo valor de los permisos (que está en R6) en los bits RWX, y se escribe en el disco.

Y para `STAT`, tras comprobar que no se da el error 4, se lee el inodo y se copian sus diez primeros bytes en la zona apuntada por R1.

19.8. Gestores de periféricos y rutinas de servicio

Como ya sabemos, el gestor del periférico se encarga, tomando los datos contenidos en el registro del periférico, de iniciar la transferencia, y de informar de su buena (o mala) terminación. Previamente realiza algunas comprobaciones con esos datos para asegurarse de que la operación es válida; por ejemplo, si el periférico es sólo de lectura (tipo = 0 en el campo correspondiente del registro del periférico) no iniciará ninguna transferencia si la macro es `WRITE`. En caso de anomalía devuelve control al sistema de gestión de ficheros, transmitiéndole un número de error.

Consideremos, para concretar, el fichero «`tty`» en su función de salida por la pantalla. Por ejemplo, para la llamada `WRITE(#0,#MENS,#30)` debe ocurrir lo siguiente:

1. Antes de pasar al gestor, el sistema de gestión de ficheros rellena los campos de los registros de la demanda y de la transferencia con estos valores:
   - «macro» = 7,
   - «df» = (R3) = 0,
   - «PUNTZU» = «dir._zona_usuario» = = (R4) = `MENS`,
   - «PUNTES» = «dir._zona_e/s» (puesta por el programa `INICIA`),
   - «NBYT» = «NBYP» = (R6) = 30.

   Luego, copia el número de bytes indicado por R6 de la zona de usuario a la zona de entrada/salida, y pasa al gestor.
2. El gestor pone «1» en el bit «ocupado» y valores iniciales en la rutina de interrupción: el número de bytes a escribir (NBYT) y la dirección en la que se encuentra el primero (PUNTES), tras lo cual devuelve control al sistema de gestión de ficheros, y éste vuelve, con `RETI`, al programa de usuario.
3. La rutina de servicio de interrupción, cada vez que transfiere un carácter, incrementa PUNTZU, decrementa NBYP y comprueba si ha terminado (NBYP = 0); en caso afirmativo, llama al gestor, y si no, vuelve a la RCI, o, si ésta no existe, directamente al programa de usuario.
4. Cuando se han escrito todos los caracteres, el gestor pone «0» en «ocupado» y llama al sistema de gestión de ficheros (para «decirle» que no ha habido errores; si la macro hubiese sido `WRITE_A` se pondría «1» o «−1» en «*aviso*»). Después se retorna ordenadamente: del sistema de gestión de ficheros al gestor, de éste a la rutina de servicio y de ésta al programa interrumpido.

En el caso de los discos, la rutina de servicio de interrupción sólo interviene al final de la transferencia. El gestor calcula y pone valores iniciales (números de pista y sector, cantidad, etc.) en el controlador de ADM del mismo modo que en los programas de los Apartados 5.9 y 7.8, de acuerdo con las informaciones de la fila de la tabla de periféricos, y al final comprueba si ha habido errores.

En la realidad, los detalles que hay que tener en cuenta tanto en los gestores de periféricos como en las rutinas de servicio pueden ser muy variados y numerosos. Por ejemplo,

- El teclado no pone en su puerto de datos la codificación ISO, sino un número de tecla (véase nota al pie de la página 202) que el gestor, o la rutina de servicio, tienen que traducir.
- Ciertas teclas o combinaciones de teclas deben tener efectos especiales (congelar la presentación en pantalla, interrumpir la ejecución de un programa, etc.); la rutina de servicio deberá detectar esas codificaciones y llamar al gestor, que, a su vez, puede llamar a otra parte del sistema.
- En la gestión del disco hay muchos detalles de los que hicimos abstracción en la Lección 5: arranque y parada del motor, entrelazado, memoria oculta de disco, etc.

19.9. Comparación con sistemas reales

Para el diseño de Monoalgorítmez hemos tomado, simplificándolos todo lo posible, algunos elementos de dos sistemas operativos bien conocidos, UNIX y MS-DOS.

Llamar «`A:`» y «`B:`» a las unidades de disco es típico de MS-DOS. En UNIX no es necesario aludir explícitamente a las unidades, porque una operación inicial (que se llama «montar») permite construir un árbol de directorios único con los directorios de distintos discos. Lo veremos en el Apartado 21.2.

De UNIX es, como vimos en la Lección anterior, el concepto de inodo, así como los superbloques y los «mapas de bits». Y también el que todo el sistema esté formado por un único programa binario (resultado de ensamblar, o compilar, y enlazar todos los subprogramas). Como decíamos en el Apartado 19.3, «se podrían añadir otros periféricos; sólo habría que escribir sus rutinas de servicio y sus gestores y volver a ensamblar (o compilar) todo el programa». En MS-DOS no es así: al tratarse de un sistema «personal» se le da más flexibilidad para que cada usuario incluya los gestores que necesita sin tener que compilar. Los gestores en MS-DOS son programas independientes que se cargan en el proceso de arranque del sistema de acuerdo con las informaciones contenidas en un fichero de configuración («CONFIG.SYS»).

Los «prefijos» de nuestros programas son simplificaciones de lo que en MS-DOS se llaman «PSP» («program segment prefix»), que incluyen más informaciones (cada PSP tiene 256 bytes). Entre ellas, las *variables de entorno* (de las que hemos prescindido aquí), y la *línea de la orden* (nosotros pasamos un puntero a los argumentos a través de R1).

También es similar a MS-DOS el mapa de la MP, con los vectores de interrupción al principio y las rutinas básicas «RUTBAS» (que en MS-DOS se llaman «BIOS», por «basic input–output system») en ROM en las direcciones altas.

Si hemos incluido dos versiones de llamadas de transferencia (`READ_A` y `WRITE_A` por un lado, y `READ` y `WRITE`, por otro) ha sido solamente por razones didácticas, para presentar dos versiones, una más elemental y otra más elaborada, de las transferencias. Algunos sistemas tienen llamadas del primer tipo (con `WAIT_A` o algún mecanismo similar), para permitir el control de la concurrencia desde los programas de usuario. Pero tanto UNIX como MS-DOS utilizan versiones más parecidas a la segunda (en ninguno existe la llamada `WAIT_A`; UNIX dispone de una `WAIT`, pero tiene otra función, que veremos en las Lecciones siguientes).

El uso de la lectura anticipada en la zona de entrada/salida es una simplificación de lo que se implementa en los sistemas operativos reales: una **memoria oculta de disco** (o «*cache de disco*») (Apartado 15.7).

19.10. Resumen

El núcleo en Monoalgorítmez es todo el sistema operativo (salvo un conjunto de rutinas básicas que suponemos residentes en ROM). Se carga durante el arranque y reside por completo en la MP. En esta Lección hemos descrito algunos de los procesos que lleva a cabo el núcleo cuando interpreta las llamadas hechas con macroinstrucciones desde los programas de usuario.

En el SGM (sistema de gestión de la memoria) hay dos punteros, DPA (dirección del programa activo) y PDL (primera dirección libre) que sirven para que este SGM sepa en dónde puede cargar los programas. Los programas cargados con `LD_EX` tienen un *prefijo* con dos punteros, para retornar adecuadamente al padre (el programa que contiene la `LD_EX`) cuando terminen con `EXIT`. Los cargados con `LD_OV` no tienen prefijo.

Hay dos estructuras de datos importantes para el SGF (sistema de gestión de ficheros): la tabla de periféricos y la tabla de ficheros abiertos. La primera contiene informaciones para cada uno de los periféricos existentes. Algunas de estas informaciones son fijas (como la dirección del puerto de estado, o la dirección de la zona de entrada/salida asignada a ese periférico), otras se rellenan cuando hay una demanda (como el número de bytes a transferir, o la dirección de la zona de usuario), y otras van cambiando conforme progresa la transferencia (como los punteros a las zonas de entrada/salida y de usuario). En la tabla de ficheros abiertos se mantienen copias de los inodos de todos los ficheros abiertos, y los valores de las variables *posición* de cada fichero.

La RIP (rutina de interrupciones de programa) es muy simple: sólo se ocupa de la macro `WAIT_A` (cuyo tratamiento consiste en esperar hasta que el byte de «aviso» tenga un contenido distinto de cero). Para las otras llamadas, bifurca al módulo que corresponda, el SGM o el SGF (sistema de gestión de ficheros). El módulo SGM es también, en Monoalgorítmez, bastante sencillo, y, lo mismo que para el de la RIP, hemos dado su codificación en ensamblador.

No puede decirse lo mismo del SGF. Como era de esperar, la facilidad que ofrecen al programador de aplicaciones las llamadas al sistema para leer y escribir de los periféricos (que le permiten olvidarse de puertos, rutinas de servicio, etc.) tiene como consecuencia que *el software del sistema puede ser bastante complicado*. En Monoalgorítmez, aun siendo una gran simplificación de los sistemas reales, hemos podido percibir esta complejidad al estudiar los procesos que ocurren en las operaciones de transferencia de datos entre un programa de usuario y un fichero en disco.

19.11. Observaciones terminológicas

«Núcleo» es la traducción de «kernel» (también llamado, aunque más raramente, «nucleus»). Y lo que hemos llamado «gestor de periférico» se conoce en la literatura técnica como «device driver», «device manager» o «device handler».

19.12. Notas históricas

El primer sistema operativo ampliamente utilizado para monoprogramación de pequeños ordenadores (basados, en los años 70, en microprocesadores de ocho bits) fue el CP/M. De él se tomaron muchas ideas para el diseño de MS–DOS, que se impuso rápidamente a partir del año 1981 con el auge de los ordenadores personales IBM y compatibles (Apartado 6.18). La primera versión (MS-DOS 1.0) tenía un sistema de ficheros con un solo directorio (igual que CP/M), y todo el sistema estaba implementado con 4.000 instrucciones en ensamblador, ocupando 12 KB (el primer *pecé* tenía 64 KB de memoria principal).

19.13. Orientaciones bibliográficas

Hay multitud de libros para el lector que desee profundizar en MS–DOS. Dos recomendables son los de Schulman *et al.* (1994) y Podanoffsky (1994). En este último se describe un sistema compatible, RxDOS, cuyos programas fuentes pueden obtenerse en la Red (Podanoffsky, 2004).

También son compatibles y accesibles por la *Internet* los programas fuente del proyecto FreeDOS (Hall, 2004), una versión libre (Apartado 20.10) de MS–DOS.

Una de las tareas más habituales no para quien diseña un sistema operativo «desde cero» (cosa realmente inusual), sino para quien tiene necesidad de adaptarlo a situaciones no previstas (lo que sí es frecuente) consiste en desarrollar gestores de periféricos. Remitimos para este asunto específico a Lai (1992).

19.14. Ejercicios

1. Para su aplicación en el control de una unidad de cuidados intensivos, añadimos al hardware de Algorítmez un conversor analógico digital que recibe señales electrocardiográficas, las muestrea y las pone en la MP (mediante un controlador de acceso directo a memoria) en bloques de 512 muestras de ocho bits (al final de la transferencia de un bloque, el controlador genera una interrupción). Las tablas del sistema operativo se han modificado para que este periférico se reconozca como un fichero especial de bloques, con nombre «`CAD`».

   El programa de aplicación debe leer sucesivamente las muestras *sin perder ninguna* (las muestras llegan con una frecuencia constante, cuyo valor no hace al caso), procesarlas y, en caso de alguna anomalía, generar un aviso sonoro (a través de otro controlador, del que no nos vamos a ocupar) y escribir en un fichero del disco (de nombre «`ECG`») el último bloque leído. Supondremos que del procesamiento (que se realiza en menos tiempo que la lectura de las 512 muestras para cumplir el requisito de no perder muestras) y la generación del aviso sonoro se ocupa un subprograma, «`PROCESAR`». Este subprograma recibe en el registro R0 la dirección de comienzo de la zona a procesar, y deja en el mismo registro R0 una codificación de la anomalía. Si no ha detectado ninguna, deja «0» en R0.

   Escriba, en el lenguaje macroensamblador de Algorítmez, el programa de aplicación (no el subprograma `PROCESAR`), haciendo uso de las macros `OPEN`, `CLOSE`, `READ` y `WRITE`. ¿Podría utilizarse `READ_A` en lugar de `READ`? ¿Qué ventajas y qué inconvenientes tendría?

2. Suponga que hemos escrito, compilado y montado en otro ordenador un ensamblador de dos pasos para Algorítmez (Apartados 3.7 y 6.8). El programa fuente está estructurado en tres módulos: el primero corresponde al primer paso (que construye la tabla de símbolos), el segundo, al segundo paso (que, haciendo uso de esa tabla, genera el código objeto), y el tercero es la propia tabla de símbolos. Tras el montaje ha resultado un módulo de carga que ocupa 80 KB, por lo que no podemos ejecutarlo en Monoalgorítmez.

   Ahora bien, caemos en la cuenta de que 30 KB corresponden al primer módulo, 40 KB al segundo y 10 KB a la tabla de símbolos. Por tanto, es factible cargar en la MP el primer módulo junto con la tabla de símbolos ($30 + 10 = 40$ KB), y luego el segundo módulo con la misma tabla ($40 + 10 = 50$ KB). Decidimos, pues, hacer uso de la técnica de módulos solapables.

   Los módulos que implementan los pasos se convierten en subprogramas (ambos terminan con la instrucción `RET`), y sus módulos de carga residen en dos ficheros del disco «`A`», con nombres `ENSP1` y `ENSP2`, y tamaños 30 KB y 40 KB, respectivamente.

   Escriba el módulo raíz (que incluye a la tabla de símbolos) teniendo en cuenta que:

   - El módulo `ENSP1` recibe como argumento, a través de un registro, una dirección de la MP. En esta dirección tiene que haber un puntero a otra dirección de la MP en la que están almacenados los caracteres correspondientes al nombre del fichero del programa fuente (suponga, por ejemplo, que se llama «`PROG.ENS`»); después de los dos bytes reservados para ese puntero estarán los 10 KB reservados para la tabla de símbolos.

   - Lo mismo es aplicable a `ENSP2`, pero el puntero apuntará a otra dirección de la MP que contiene el nombre del fichero para el programa objeto («`PROG.OBJ`»).

   - El módulo raíz que hay que escribir tendrá que llamar a esos dos módulos, transmitiéndoles el argumento adecuado a cada uno. Antes de la llamada al primer paso iniciará la escritura por la pantalla del mensaje «`Paso 1`», y antes de llamar al segundo iniciará la escritura del mensaje «`Paso 2`».

   Suponiendo que el intérprete de órdenes ocupa quinientos bytes, calcule cuántos bytes de la zona de usuario se están ocupando durante la ejecución de cada uno de los pasos.

3. En el programa dado para la RIP (Programa 19.1), ¿es necesaria la instrucción `WAIT`? ¿Que ocurre si se omite? ¿Y es necesaria la `BR    TR_WAIT_A`? ¿Qué puede ocurrir si se sustituye por `RETI`?

4. Complete la codificación del módulo `INICIA` (Programa 18.7).

5. Estudie esta otra alternativa para el sistema de ficheros de Monoalgorítmez, con asignación encadenada de los bloques (Apartado 17.8), y más parecida a la de MS-DOS:

   - No existen inodos.

   - Las filas del directorio, además del nombre de cada fichero, contienen parte de la información que iba en los inodos: día y hora (codificados como en el Ejercicio 5 de la Lección 18), longitud y número del primer bloque asignado al fichero.

   - Una tabla adicional *común*, la **tabla de asignación de ficheros**, contiene la información necesaria para saber los números de los bloques que se asignan a cada fichero. La tabla tiene tantas filas como bloques tiene el disco. En cada fila se indica el número del bloque siguiente. Se reservan números de bloques que no pueden asignarse a ficheros porque están ocupados (los primeros) para indicar que el bloque correspondiente a esa fila está libre, que es el último bloque del fichero, o que el bloque está inutilizado (el sistema de gestión de ficheros ha detectado algún error al acceder a él y lo señala como «malo»).

Analice en particular los requerimientos para el almacenamiento del directorio y de la tabla de asignación de ficheros, así como las informaciones que conviene mantener en la MP y las repercusiones en el tratamiento de las llamadas al sistema de gestión de ficheros.

6. Encuentre los valores concretos de los componentes del inodo correspondiente al fichero «`sistema`», suponiendo que la longitud de este fichero es 5KB, y que los bloques asignados a él son los siguientes a los de los inodos.

7. Todo el proceso de arranque descrito en la Lección anterior (Apartado 18.9) se desarrolla con las interrupciones inhibidas. Pero durante ese proceso deben realizarse operaciones de lectura por ADM de los discos, y, por otra parte, el programa tiene que prever posibles mensajes de error por la pantalla (por ejemplo, si en el disco no se encuentra el fichero «`sistema`»). ¿Cómo pueden escribirse esos mensajes? ¿Y cómo puede saber el programa cuándo ha terminado la lectura de cada sector para seguir adelante? Estudie dos soluciones: una sin permitir interrupciones, y otra con permisos temporales y «controlados» de las interrupciones en los programas de arranque.

8. Si el directorio es un fichero, podría llegar a tener 133 KB. Sin embargo, su longitud está limitada. ¿Por qué? ¿Cuál es el límite?

9. Como hemos reducido el número de llamadas a un subconjunto mínimo, algunas funciones no podrían implementarse en nuestro sistema. Si ha tratado de resolver el Ejercicio 12 de la Lección anterior, es posible que lo haya percibido al escribir el programa de `cp`, pero si se le ha escapado, observe las informaciones que se guardan en los inodos (Figura 18.4). ¿Qué nuevas llamadas son necesarias para poder escribir todas las informaciones de un inodo? (Pista: en UNIX se llaman «TIME» y «STIME»; véase también el Apartado 21.7).

10. Las llamadas del ejercicio anterior serían imposibles sin añadir otra cosa a un nivel más bajo. Retroceda al Ejercicio 24 de la Lección 7. Suponiendo que existe en el módulo `RS` la rutina de servicio que allí se menciona, ¿qué hay que añadir en el módulo `INICIA`?

11. En un momento determinado se está ejecutando bajo Monoalgorítmez un programa, al que llamaremos `PROG1`, cuya primera instrucción está cargada a partir del byte de dirección D'6.952. Este programa contiene una llamada `LD_EX` para pedir la ejecución de otro programa cuyo nombre es `PROG2`; el primer byte de la primera instrucción de la expansión de esta macro está en la dirección D'7.000. Es decir, la estructura de `PROG1` es:

```
[6952]  ...
        NPROG2  DATA.B  "PROG2",0
 ...             ...
 ...             ...
[7000]          LD_EX(#NPROG2,#DARG)
[xxxx]  ... ; (instrucción siguiente)
 ...    ...
 ...    ...
[7050]          EXIT
```

Por su parte, `PROG2` (del que sólo sabemos que, una vez traducido a binario, ocupa 250 bytes) tendrá un conjunto de instrucciones y seudoinstrucciones que no hacen al caso:

```
[yyyy]      ...
 ...        ...
[zzzz]      EXIT
```

• Escriba los valores concretos en decimal que tendrán las direcciones «`xxxx`», «`yyyy`» y «`zzzz`».

• Sitúese en el momento en que empiezan a ejecutarse las instrucciones correspondientes a `LD_EX`, y suponga que en ese momento el contenido de PP (puntero de pila) es D'64.000 y el contenido de RE (registro de estado) es D'1.280. Como debe saber, la `BRK` contenida en `LD_EX` provoca la ejecución de la RIP, ésta bifurca al SGM, y éste llama sucesivamente a varios subprogramas. Escriba en la siguiente tabla los valores que van teniendo el puntero de pila (PP), el contenido de la cima de la pila (CIP), DPA (dirección del programa activo) y PDL (primera dirección libre) conforme van teniendo lugar esos acontecimientos (suponga que la primera instrucción `CALL /CARGAPROG` del Programa 19.1 está almacenada a partir de la dirección D'4.050).

	(PP)	(CIP)	(DPA)	(PDL)
al empezar `LD_EX`	64.000	-----		
tras la instrucción `BRK`				
tras la instrucción `CALL /CARGAPROG`				
tras la instrucción `EX_SAL RETI`				

12. El siguiente programa pretende operar con los datos de un fichero llamado «`TEMP`»:

```
DESCR   RES.B   1
NOMBRE  DATA.B  "TEMP",0
ZLEC    RES.B   2000
ZESC    RES.B   1000
        ---
        OPEN(#NOMBRE,#0)
        ; suponemos que no hay error
        ST      .13,DESCR
        READ(DESCR,#ZLEC,#2000)
        ---
        ---    ; procesa ZLEC
        ---    ; y rellena ZESC
        POS(DESCR,#1000,#0)
        WRITE(DESCR,#ZESC,#1000)
        ---
```

Hay dos cosas mal en este programa (una daría un error en tiempo de traducción, y la otra daría el error en tiempo de ejecución). Corríjalas, y escriba en las tablas (a) y (b) los valores sucesivos que van tomando las variables NBYP, NBL, NBYT, PUNTES y PUNTZU del registro del periférico en el que está el fichero `TEMP`. Necesitará los siguientes datos:

• El inodo de `TEMP` contiene las siguientes informaciones:

tipo:	ordinario
modo:	6
longitud:	3.000
bloques:	1.417, 122, 392, 55, 1.005, 87

• La zona reservada por el sistema operativo para la entrada/salida del disco `A:` ocupa los bytes de direcciones D'1.500 a D'2.011.

• A la etiqueta `ZLEC` le corresponde la dirección absoluta D'7.000, y a la `ZESC`, la D'9.000.

(a) Durante la ejecución de `READ(DESCR,#ZLEC,#2000)`:

NBYP	NBL	NBYT	PUNTES	PUNTZU

(b) Durante la ejecución de `WRITE(DESCR,#ZESC,#1000)`:

NBYP	NBL	NBYT	PUNTES	PUNTZU

13. Reescriba la rutina de servicio de la pantalla (Programa 7.5) de acuerdo con lo dicho en el Apartado 19.8 y en la nota al final del Apartado 18.9.

Lección 20

Multialgorítmez

20.1. Propósito

El objetivo de esta Lección es mostrar, de manera simplificada pero concreta, la *función*, la *estructura* y el *funcionamiento* de un sistema operativo con multiprogramación.

Partiremos de los modelos de Monoalgorítmez explicados en las dos Lecciones anteriores. Propondremos en primer lugar unas *ampliaciones funcionales mínimas* para que el sistema atienda a varios usuarios «simultáneamente» y para que cada usuario pueda ordenar la ejecución, también «concurrente», de varios programas[1]. Veremos luego que algunas de las *estructuras de datos* de Monoalgorítmez han de sufrir ciertas modificaciones, y que hay que introducir otras adicionales. Y finalmente describiremos los cambios más importantes del *modelo procesal*.

Tratar de implementar sobre un hardware tan limitado como el de Algorítmez (Apartado 6.15) un sistema operativo multiusuario y multitarea sería una tarea disparatada. Podríamos haber recurrido a Regístrez, pero, para no distraer la atención del lector diversificando detalles, nos parece más oportuno apoyarnos sobre lo ya explicado (Monoalgorítmez) y obviar los problemas de implementación. Se trata de ilustrar los principios de la multiprogramación con datos concretos, sin importar que, en realidad, el sistema sea poco viable por limitaciones del hardware.

La limitación más importante de Algorítmez radica en su escasa capacidad de direccionamiento. Aliviaremos algo esta limitación mediante el uso de la UGM descrita en el Apartado 7.10. Además de permitirnos disponer de 1 MB para albergar al sistema operativo y a los programas de aplicaciones (o «de usuario»), la reubicación de programas es automática (sin necesidad de un cargador reubicador), y proporciona un mecanismo seguro de protección de la memoria que impide que un programa de usuario pueda afectar a otros, sea fortuita o malintencionadamente (como el llamado «`ATILA`» en el Ejercicio 16 de la Lección 7; véase el Ejercicio 1 de ésta). El lector debería volver a estudiar el Apartado 7.10 antes de seguir con esta Lección.

Los modelos de Multialgorítmez son adaptaciones simplificadas del sistema operativo UNIX. En la Lección siguiente tendremos ocasión de percibir el alcance de esas simplificaciones.

[1]Si las palabras entrecomilladas le sorprenden, retroceda por un momento a los últimos párrafos del Apartado 2.8.

20.2. Procesos

Antes de empezar con el modelo funcional conviene tener presentes algunas ideas sobre los procesos. Ante todo, hay que recordar el «modelo procesal de los procesos» del Apartado 17.9.

En la explicación de Monoalgorítmez no hemos hecho alusión alguna a procesos. No es que no los haya: en todo momento hay un proceso *activo* (el del núcleo, o el del intérprete de órdenes, o el del programa que éste ha cargado, o, en general, el resultante de una `LD_EX`), que puede pasar a *bloqueado* (el intérprete, cuando espera una orden; en general, cualquier proceso, cuando ejecuta `WAIT_A(#AV)` y `(AV)` = 0, o cuando hace una petición a un periférico y éste está ocupado). Y cuando un proceso se bloquea el núcleo no activa a ningún otro proceso de usuario: entra en un bucle de espera hasta que termina la causa del bloqueo.

Lo que ocurre en Monoalgorítmez es que la tarea de *distribución* (asignación de la UCP a los procesos) es trivial: después de toda interrupción, el proceso que estaba activo sigue estándolo, salvo si la causa fue `LD_EX` o `EXIT`. No hay necesidad de salvar explícitamente el estatus del proceso al entrar en el núcleo: el hardware ya se encarga de guardar en la pila, y luego recobrar (cuando se ejecuta `RETI`), los contenidos de los registros CP y RE, y los demás registros se salvan y se recuperan con las instrucciones `PUSH` y `POP` de las expansiones de las macros en el caso de interrupción de programa, o con las operaciones previstas en las rutinas de servicio de otras interrupciones.

En un sistema con multiprogramación no es así. Tras una interrupción, el proceso activo puede pasar a bloqueado o a preparado, y activarse cualquier otro proceso que no tenga nada que ver con él. En el cambio de contexto es preciso salvar el estatus del proceso suspendido para que más adelante, cuando se reactive, pueda continuar su ejecución. Para ello, el núcleo debe tener una estructura de datos (en Multialgorítmez será, como veremos, la *tabla de procesos* de la Figura 20.5) en la que guardar, entre otras cosas, los contenidos de los registros de la UCP en el momento en que el proceso dejó de estar activo. También debe tener un criterio, o algoritmo de *planificación*, para que el *distribuidor* sepa qué proceso tiene que activar y recupere su estatus. En el Apartado 20.6 explicaremos el planificador y el distribuidor de Multialgorítmez.

Prioridades

Como veremos, ese algoritmo de planificación calcula para cada proceso una prioridad, que en Multialgorítmez es un número entero comprendido entre 0 (prioridad máxima) y 19 (prioridad mínima), de manera que lo único que tiene que hacer el distribuidor es activar al proceso de máxima prioridad. Esa prioridad es dinámica, pero tiene una componente estática, llamada *prioridad de base*, que se fija en el momento de la creación del proceso, y es un número comprendido entre 0 (máxima) y 10 (mínima), siendo 5 la prioridad normal.

Pilas de los procesos

En Monoalgorítmez reservábamos una zona de la MP para una pila *única* (Figura 19.2). Cuando el proceso activo termina (con `EXIT`), el sistema de gestión de la memoria (Programa 19.2) se encarga de restaurar el valor del puntero de pila, guardado en el prefijo, para que la instrucción `RETI` recupere de la pila los valores de CP y RE del padre (Figura 19.6). Si un proceso crea a otro (con `LD_EX`), y este a otro, y así sucesivamente, las pilas de los procesos se van apilando una sobre otra, y se van quitando de la pila conforme van apareciendo las `EXIT` (en el orden inverso al que aparecieron las `LD_EX`). No hay problema, porque el proceso activo es siempre el último cargado, y cuando termina es su padre el que pasa a ser activo.

Pero en Multialgorítmez no podemos tener una pila única, porque cuando el proceso activo deja de serlo, pasa a activarse alguno de los preparados, cuya pila, si estuviese en medio de la «pila de las pilas», no podría crecer sin afectar a las que están por encima (Ejercicio 3). Por eso, a cada proceso se le asigna una zona de memoria para su pila. Y el núcleo tiene también su propia pila.

Imagen de memoria

Multialgorítmez va a utilizar la UGM del modo que sugeríamos al final del Apartado 7.10: a cada proceso se le asigna el número necesario de páginas (de 8 KB cada una), y sólo puede acceder a esas páginas. Al cargar un programa en la MP, el núcleo añade una página para su pila. Como el espacio de direccionamiento lógico de un proceso es 64 KB (ocho páginas), los programas ejecutables en Multialgorítmez tendrán un tamaño máximo de siete páginas, es decir, 56 KB.

La **imagen de memoria** de un proceso es el conjunto de bytes de la MP a los que puede acceder, con sus direcciones lógicas. Así, si un programa (y sus variables y datos declarados con `RES` y `DATA`) ocupa, por ejemplo, 35 KB, el núcleo, al crear el proceso, reserva para él seis páginas: cinco para el código del programa (40 KB) y una para la pila. Para este proceso, la imagen de memoria sería la que muestra la Figura 20.1 (aunque las páginas lógicas se traducen en páginas físicas esparcidas por la memoria real, Figura 7.13). Si una instrucción del programa intenta acceder a una dirección lógica superior a H'BFFF, el sistema no lo permitirá, porque previamente habrá puesto «0» en los bits de validez de las páginas lógicas 6 y 7, y cualquier intento de acceso a estas páginas genera una interrupción. A propósito de este ejemplo, es interesante señalar dos cosas:

- Salvo que el tamaño del programa sea justamente un múltiplo de 8 KB, siempre aparecerá una «zona no utilizada» en la imagen de memoria (5 KB en el ejemplo). En la MP habrá tantas zonas no utilizadas como procesos estén presentes. Es la *fragmentación interna* asociada a la técnica de paginación, que ya comentamos en el Apartado 15.8. Pero no es una «zona prohibida» para el proceso (a diferencia de la comprendida entre las direcciones H'C000 y H'FFFF). El programa puede acceder a ella y utilizarla, por ejemplo, para guardar variables. Como estas direcciones están fuera del espacio de direccionamiento ocupado por el código, esta zona de memoria no es «estática» (reservada con `RES` o `DATA`) sino «dinámica» (asignada en tiempo de ejecución, véase el Ejercicio 4).

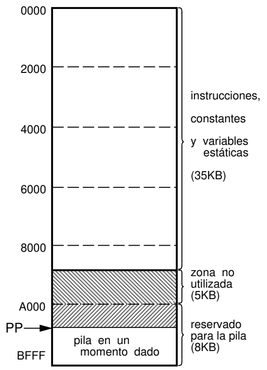


Figura 20.1. Imagen de memoria de un proceso cuyo código ocupa 35 KB.

- Hemos dicho que el núcleo añade la última página «para la pila», pero es obvio que el programa puede utilizar esa zona como mejor le convenga. Si la pila no va a crecer más de 1 KB, puede hacer uso de los 7 KB restantes para «memoria dinámica». La pila crece hacia arriba, y la memoria dinámica hacia abajo. En Multialgorítmez, es responsabilidad del programador de aplicaciones asegurarse de que no lleguen a «chocar» (Apartado 21.5 y Ejercicio 9 de la Lección 22).

Supondremos que la imagen de memoria del núcleo ocupa 64 KB (incluyendo las rutinas básicas de la ROM) y que sus ocho páginas lógicas se traducen (con los valores adecuados en los registros de la UGM) en las páginas físicas 0 a 7. Así, el mapa de los primeros 64 KB de la memoria física es el mismo que en Monoalgorítmez (Figura 19.1), con la salvedad de que el espacio reservado para la pila del núcleo es mayor (7 KB, ya que la ROM ocupa los últimos 1.024 bytes de la página 7) y que todo el espacio está reservado para el núcleo (no se cargan en él programas de usuario).

A medida que se van creando procesos, el sistema les va asignando páginas físicas para contener a sus imágenes: 8, 9, etc. En la tabla de procesos se guardan también los números de páginas físicas asignadas a cada proceso (valores a cargar en los registros de la UGM cuando ese proceso se active). Y a medida que los procesos van «muriendo» se van liberando páginas. Como los «huecos» que van quedando no tienen longitud fija (tienen entre dos y ocho páginas), llegará un momento en que el sistema operativo no pueda asignar páginas físicas contiguas a los nuevos procesos, y la correspondencia entre páginas lógicas y físicas para un proceso será similar a la mostrada en la Figura 7.13.

Cambios de contexto

Ahora se puede entender mejor la justificación de algunas características de la UCP y de la UGM explicadas en las Lecciones 6 y 7. El hardware está diseñado para que en este sistema con multiprogramación se aseguren dos cosas:

- Que cuando se produce una interrupción se pase del contexto del proceso de usuario al del núcleo, guardando el estatus del primero (todos los registros de la UCP, incluyendo PP y CP, y el registro de estado, RE), y que cuando se active un proceso se pase del contexto del núcleo al del proceso, recuperando su estatus.
- Que en ningún caso pueda un proceso de usuario acceder a páginas que no sean las que tiene asignadas, ni ejecutarse en modo supervisor.

En efecto, según el modelo procesal de las interrupciones (Apartado 6.14, o, con más detalle, Programa 13.6 y Tabla 13.15), cuando la UCP atiende a una interrupción empieza guardando los contenidos de CP y RE en la pila del proceso. A continuación pone «0» en PIN y «1» en SUP. Esto último tiene el efecto inmediato de hacer que la señal de salida de la UCP «**modo**» pase de «0» a «1» y que la UGM, al detectar este cambio (Apartado 7.10) ponga automáticamente en los registros de la tabla de páginas los valores por defecto, que son, precisamente, los que corresponden al núcleo. A partir de este momento, cualquier dirección lógica que genere la UCP se traduce en una dirección física de las páginas del núcleo. Siguiendo con el modelo procesal de las interrupciones, la UCP genera la dirección del vector (si ha sido una interrupción interna o no enmascarable) o la recoge del bus de direcciones (si la han generado los circuitos externos), lee ese vector y lo introduce en CP para bifurcar a la rutina que corresponda (la RCI, la RIP o una rutina de servicio). Esa lectura se hace ya en la página física 0 de la MP, que es donde están los vectores. Si la UGM no hubiese cambiado la función de correspondencia, se habría leído de una página asignada al proceso de usuario, y éste se encontraría con todo el control sobre la máquina (puesto que en la dirección correspondiente al «falso vector» el programador habría puesto una dirección de su propio programa, que se seguiría ejecutando ¡con (SUP) = 1!). El cambio de contexto se completa salvando los registros R0 a R14 = PP.

Veamos ahora el otro cambio de contexto, el del núcleo al de un programa de usuario. Se realiza dentro del distribuidor, una vez que se ha decidido qué proceso va a activarse. Primero se cargan en los registros del puerto 21 (puerto de la UGM) los valores que establecen la función de correspondencia para ese proceso, es decir, las páginas físicas asignadas a sus páginas lógicas. Pero recuérdese que esos registros son temporales: hasta que no se transfieran a los registros que forman la tabla de páginas no tiene efecto la nueva función de correspondencia. Por tanto, seguimos ejecutando instrucciones del distribuidor. A continuación se cargan en los registros R0 a R14 = PP de la UCP los valores correspondientes al proceso que va a activarse. Y finalmente se ejecuta `RETI`. Y aquí hay que fijarse bien en las microoperaciones asociadas a esta instrucción (Tabla 6.2, o, con más detalle, Programa 13.1 y Tabla 13.10). La primera consiste en poner «0» en el indicador SUP de RE, lo que provoca que la señal «**modo**» pase de «1» a «0» y que la UGM, al detectar este cambio, cargue los nuevos valores en los registros de la tabla de páginas. A continuación se cargan los registros RE y CP con los valores leídos de la pila. Como ya se ha cambiado la función de correspondencia (y el puntero de pila se había recuperado antes), se leerá, efectivamente, de la pila del proceso. Aunque al cargar RE se pone a cero SUP, es esencial ponerlo previamente, porque de lo contrario no se cargaría RE con su valor correcto en la pila del proceso, sino con el contenido de alguna dirección del espacio del núcleo. Y observe que también es esencial que el cambio de la función de correspondencia se haga dentro de la ejecución de `RETI`. Si lo hiciésemos antes de `RETI` (con una instrucción `OUT` que pusiese el valor H'02 en el puerto 20, como vimos en el Apartado 7.10), la `RETI` no llegaría nunca a ejecutarse, porque está en el espacio de memoria del núcleo.

Algunas UCP proceden de manera diferente, guardando CP y RE en la pila del núcleo, no en la del proceso, lo que tiene ciertas ventajas (Ejercicios 6, 7 y 8).

Procesos zombis

En un sistema con multiprogramación, dos procesos pueden ser independientes entre sí, pero también pueden estar íntimamente relacionados. Típicamente, un proceso (padre) crea a otro (hijo) con una llamada, `FORK`, que explicaremos en el siguiente Apartado. Y entre el padre y el hijo pueden establecerse vías de comunicación (por ejemplo, el hijo realiza cierto procesamiento cuyos resultados deja en un fichero, de donde los lee el padre). Si el padre no puede seguir hasta que el hijo no haya acabado, utiliza una llamada `WAIT` (que también explicaremos en el siguiente Apartado; de momento, no hay que confundirla con la *instrucción* `WAIT` ni con la llamada `WAIT_A`). Esta llamada hace que el padre quede bloqueado hasta que en el hijo se ejecute `EXIT`. Ahora bien, hay que prever la posibilidad de que el hijo llegue a `EXIT` (o que otro proceso intente «matarlo») antes de que el padre haya llegado a `WAIT`. En tal caso el sistema no debe desembarazarse del todo del proceso hijo (que es lo que normalmente hace con un proceso cuando en él aparece `EXIT`), porque puede tener informaciones importantes para el padre (por ejemplo, la información de que su procesamiento se ha desarrollado sin errores; como veremos más adelante, estas informaciones se guardan también en la tabla de procesos). Y sin embargo, a todos los demás efectos, está «muerto» (no tiene que volver a activarse, ni tiene por qué seguir ocupando espacio en la MP). Por esto, se dice que está **zombi**. El sistema lo señala como tal, y cuando ejecuta la `WAIT` del padre (o cuando se hace terminar a éste con `KILL`), lo «mata» definitivamente.

Diagrama de estados

La Figura 20.2 muestra el diagrama de transiciones, incluyendo el estado zombi, que no aparecía en la Figura 17.3. Se indican las llamadas al sistema cuya ejecución provoca que el sistema lleve a cabo las transiciones entre estados. Estas llamadas se explicarán en los Apartados siguientes.

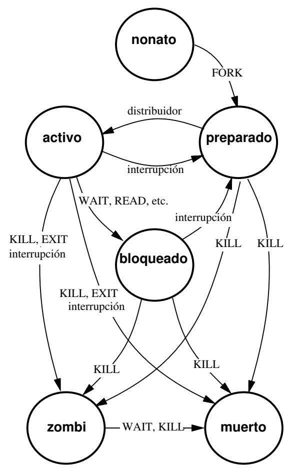


Figura 20.2. Diagrama de estados de un proceso en Multialgorítmez.

Las interrupciones que aparecen en algunas transiciones de la Figura 20.2 proceden de causas variadas. Por ejemplo:

- La interrupción de un periférico que pone fin a una transferencia de entrada o de salida: el proceso que había iniciado esa transferencia quedó bloqueado, y debe pasar a preparado.
- La interrupción de reloj, que, como veremos en el Apartado 20.6, actualiza las prioridades de los procesos: uno de los procesos preparados puede alcanzar una prioridad mayor que el actualmente activo, y, por tanto, debe activarse, pasando el activo a preparado.
- La interrupción por violación del modo usuario, o la interrupción de la UGM por intento de acceso a página no asignada, que mata al proceso, o lo pasa a zombi si su padre no ha hecho `WAIT`.

Proceso nulo

Puede ocurrir que, en un momento dado, todos los procesos de usuario estén pendientes de algún acontecimiento, y, por tanto, bloqueados. El distribuidor pasa entonces a activar un proceso especial que consiste, simplemente, en esperar[2]. Naturalmente, para que pueda ser interrumpido será preciso salir del modo supervisor ininterrumpible en el que estamos suponiendo que, salvo excepciones, se ejecuta el núcleo. Es decir, el proceso nulo es como cualquier proceso de usuario, en el sentido de que puede ser interrumpido.

[2] Igualmente podría hacer cualquier tarea, como calcular indefinidamente los dígitos del número π, simular la evolución de las especies, jugar a las «torres de Hanoi» (Ejercicio 5 de la Lección 22)... El caso es que no requiera entrada/salida, no necesite a otros procesos, y tenga la prioridad mínima. Pero si, como ocurre en Algorítmez, el hardware lo permite (existencia de una *instrucción* `WAIT`), es preferible que el proceso nulo no sea una espera activa, sino una inhibición del funcionamiento de la UCP: de este modo no está continuamente accediendo a la MP para extraer instrucciones, quedando los buses libres y disponibles para que otros procesadores puedan acceder a MP. Algorítmez no tiene previsto en su hardware mecanismos para construir sistemas multiprocesadores, pero sí para permitir el acceso directo a memoria.

Esto ya lo hacíamos en Monoalgorítmez: cuando el proceso del usuario no podía seguir, poníamos a la UCP en espera mediante una *instrucción* `WAIT`, permitiendo previamente las interrupciones (veánse, en particular, el Programa 19.1 y las Figuras 19.7, 19.9 y 19.10). Ahora podemos decir que aquél sistema operativo es de «biprogramación»: un proceso es el del usuario, y el otro es el nulo.

Implementado con esta instrucción `WAIT`, el proceso nulo no es, en rigor, un «proceso» (un «programa en ejecución», Apartado 17.9), pero, en el nivel de máquina operativa, es una forma útil de referirse a este estado de la UCP.

Identificadores de los procesos

Descontando las ocho páginas ocupadas por el núcleo, la memoria queda con una capacidad de ciento veinte páginas para los procesos de usuario. El número máximo de procesos que caben en la memoria depende de su tamaño. Si todos tuviesen la longitud máxima, ocho páginas, sólo podría haber quince (120/8). El extremo opuesto es que todos los procesos fuesen de longitud mínima, dos páginas, y en ese caso podríamos llegar a sesenta (sin contar los procesos zombis y el nulo, que no ocupan memoria).

El sistema de gestión de la memoria llevará la cuenta, naturalmente, de las páginas libres en cada momento, y no dejará crear un proceso nuevo si no tiene sitio para él. En el Apartado 21.4 veremos que hay formas más razonables de proceder.

El núcleo asigna a todo proceso de usuario, sea cual sea su estado (salvo «nonato» y «muerto»), un número llamado **identificador del proceso**, o **«pid»**. El proceso nulo tiene *pid* $= 0$.

Además del *pid*, para todo proceso se mantiene un conjunto de informaciones en una estructura de datos (la tabla de procesos) que explicaremos en el siguiente Apartado. Para fijar el tamaño de esta tabla, limitaremos a sesenta y cuatro el número de procesos de usuario (incluyendo los zombis, pero excluyendo al nulo). El *pid* será, pues, un número comprendido entre 0 y 64.

20.3. Modelo funcional

Usuarios y grupos

Multialgorítmez es un sistema **multiusuario**. Puede haber hasta doscientos cincuenta y seis usuarios «registrados», que el sistema reconoce por un número, el **identificador de usuario** (abreviadamente, «**uid**»). Supondremos que sólo hay cuatro terminales (de modo que en un momento dado pueden estar conectados, como máximo, cuatro usuarios[3]), y dotaremos al sistema de algunos mecanismos para la protección de los ficheros de unos usuarios frente a otros.

El sistema mantiene también hasta ocho *grupos de usuarios*, cada uno con un **identificador de grupo** («**gid**»). Todo usuario pertenece a uno (y sólo uno) de estos grupos. Por ejemplo, en un entorno académico pueden definirse los grupos «profesores», «investigadores», «alumnos», «secretaría», «administración», etc.

Sólo un usuario especial, el de *uid* = 0, tiene en todo momento acceso a todos los ficheros existentes y puede realizar ciertas operaciones (como dar de alta o de baja a usuarios, asignar o cambiar *uids* y *gids*, etc.). A este usuario se le conoce como **administrador del sistema**, o «**superusuario**».

Autenticación

Un asunto importante en todo sistema de información compartido por varios usuarios es el de la *seguridad*. La seguridad tiene muchas facetas (comentaremos algunas en el Apartado 21.10). Una de ellas es la **autenticación** del usuario: el sistema debe identificar al usuario que quiera «entrar» en él, para permitirle o no el acceso a ciertas operaciones (lectura o escritura de ficheros según sus permisos, ejecución de programas, etc.).

En Multialgorítmez seguiremos el mecanismo de **contraseñas cifradas**. Asociadas a cada *uid* hay dos informaciones: el nombre del usuario y la contraseña. El primero lo registra el superusuario junto con el *uid* y el *gid*; la contraseña puede elegirla el usuario y cambiarla cuando lo desee (con una orden que veremos enseguida). Las informaciones sobre *uids*, nombres y contraseñas se mantienen en una tabla almacenada en un fichero de nombre «`psswd`». Las contraseñas se guardan *cifradas* de acuerdo con un algoritmo que tiene una propiedad básica: el algoritmo es una función prácticamente imposible de invertir. Es decir, dada la contraseña «x» el algoritmo de cifrado obtiene «$f(x)$». Pero a la vista de «$f(x)$» no puede deducirse «x». De este modo, no importa que cualquier usuario pueda leer el fichero de contraseñas (pero no escribir en él; solamente en su propio registro y a través de la orden que permite cambiar la contraseña).

La «entrada» en el sistema de un usuario registrado se realiza del siguiente modo:

En la pantalla de un terminal libre aparece escrito «`login:`». El usuario escribe en el teclado su nombre seguido de «`ret`» (tecla de retorno). El sistema presenta en la pantalla «`password:`». El usuario teclea su contraseña (y «`ret`») y el sistema (que, excepcionalmente, no hace «eco» en la pantalla por motivos obvios) aplica el algoritmo de cifrado y acude a la tabla de contraseñas en la fila correspondiente al nombre. Si la contraseña (cifrada) coincide con la registrada, el sistema carga el intérprete y el usuario puede empezar a trabajar. Si no, vuelve a aparecer «`login:`».

El nombre del superusuario (*uid* = 0) es «**root**» (raíz)[4].

Ficheros y permisos de acceso

Para no añadir más complejidad, seguiremos manteniendo la limitación de que el sistema de ficheros sólo tiene un directorio (lo cual en un sistema multiusuario sería realmente enojoso: un usuario no puede dar un nombre de fichero si otro ya ha creado un fichero con ese mismo nombre). Todo fichero tiene registrado en su inodo el *uid* del usuario que lo creó y el *gid* de su grupo, así como nueve bits que determinan los *permisos de acceso* (Apartado 18.2): quién tiene derecho y quién no para leer, escribir y ejecutar ese fichero. Estas informaciones se encuentran en los tres primeros bytes del inodo (Figura 18.4) siguiendo el formato que indica la Figura 20.3[5].

[3]O menos, si alguno se conecta en más de un terminal.

[4]La diversión favorita de algunos usuarios de sistemas UNIX consiste en tratar de averiguar la contraseña asociada a «root», o de adquirir los derechos de «root» sin conocer su contraseña.

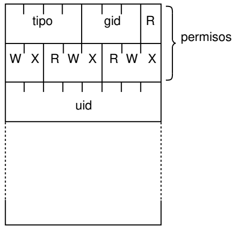


Figura 20.3 Los tres primeros bytes de un inodo.

De los nueve bits de permisos, los tres primeros se refieren al propietario del fichero (aquél cuyo *uid* está en el tercer byte), los tres siguientes a todos los usuarios de su grupo (cuyo *gid* está en el primer byte) y los tres últimos a cualquier usuario en general. Conjuntamente, los bits de permiso se representan como una sucesión de tres dígitos octales que indican, de izquierda a derecha los permisos del propietario del fichero, de su grupo y de cualquier otro usuario. Por ejemplo, «750» significa que el propietario tiene permiso para leer, modificar y ejecutar (111), que los usuarios de su grupo tienen permiso para leer y ejecutar, pero no para modificar (escribir en él, o borrarlo) el fichero (101), y que el resto de usuarios no tiene permiso para nada (salvo el superusuario, que siempre lo tiene para todo).

Intérprete de órdenes

Cada uno de los usuarios conectados a un terminal dispone de una máquina virtual cuyo modelo funcional es el mismo de Monoalgorítmez, con algunas ampliaciones. En el resto de este Apartado detallaremos esas ampliaciones. Cómo hace el sistema para que cada usuario tenga su propia máquina es algo que pertenece al modelo procesal, y lo trataremos en el Apartado 20.5.

El intérprete reconoce las órdenes explicadas en el Apartado 18.2, pero en su ejecución hay ahora ciertas restricciones que dependen del usuario y de los permisos. Por ejemplo, `chmod 750 fich` pondría en el inodo del fichero `fich` los permisos que hemos indicado más arriba, pero el sistema sólo acepta la orden si el *uid* de `fich` es el del usuario que da la orden (o bien si éste es *uid* = 0).

Las órdenes nuevas son:

- `logout`, para «salirse» del sistema. El terminal quedará libre, apareciendo en la pantalla «`login:`»
- `passwd`, que permite cambiar la contraseña. El sistema preguntará por la contraseña actual y la comprobará[6], pedirá luego la nueva, cifrará ésta y la introducirá en el fichero de contraseñas en el lugar de la antigua.
- `ps` para obtener un listado de los procesos existentes, con sus *pids*, sus *uids* y sus estados.
- `kill` seguido de un número de *pid* «mata» al proceso (hace abortar su ejecución y libera su *pid* y la memoria que ocupa).
- `nice`, que, acompañada de un número entre 0 y 10 y antepuesta a otra orden, sirve para modificar la prioridad de base (normalmente, 5) del proceso que se ejecuta como consecuencia de esta otra orden. Como ya hemos dicho, «0» es la máxima prioridad y «10» la mínima, pero sólo al superusuario se le permite utilizar un número inferior a 5. Esta característica le ofrece

[5]En el «tipo», dos bits indican si el fichero es ordinario, directorio, especial de caracteres o especial de bloques, como vimos en el Apartado 18.8. Un tercer bit especifica si el fichero puede compartirse o no para que dos procesos independientes entre sí puedan leer de él, y el cuarto si puede o no compartirse para escritura. La utilidad de estas últimas informaciones se verá en el Apartado 20.5.

[6]Precaución obvia: el usuario podría haberse ido a tomar café sin hacer «`logout`», por lo que el sistema vuelve a autenticar al usuario.

al usuario la oportunidad de ser «*nice*» (amable) con los demás: si ordena ejecutar un proceso que requiere mucho tiempo de UCP y no tiene prisa en obtener los resultados puede reducir con ella la prioridad de ese proceso, dando así más tiempo de UCP a los procesos de otros usuarios[7].

Además, el intérprete admite varias órdenes que permiten al usuario designar varios programas para su ejecución concurrente. Para explicarlas es preciso introducir antes los conceptos de ejecución «*en escena*» y ejecución «*entre bastidores*».

Tomemos como ejemplo un programa que está en un fichero llamado `procesa` y que realiza cierto procesamiento de imágenes (suponemos que esta tarea requiere mucho tiempo de UCP), al que se le da como argumento el nombre de otro fichero que contiene los bits de la imagen a procesar. Si damos la orden:

```
procesa imag1
```

el intérprete carga en la MP a `procesa` y éste empieza a ejecutarse tomando los datos del fichero `imag1`. Durante todo el tiempo que dura su ejecución nuestro terminal está bloqueado, porque está bajo el control de `procesa` (suponemos que éste no pide datos al usuario; todos los toma de `imag1`). Decimos que `procesa` se está ejecutando **en escena**. Pero podemos también ordenar que, mientras la ejecución procede, el terminal quede liberado para que vuelva a ejecutarse el intérprete y desde éste pedir la ejecución, en paralelo con `procesa`, de otro programa. Basta con dar la orden de este modo:

```
procesa imag1 &
```

El sistema responde a «`&`» volviendo a activar el intérprete de órdenes inmediatamente después de cargar `procesa`, y éste se ejecuta **entre bastidores**. Mientras, podemos ordenar la ejecución de otro programa, por ejemplo, un editor de textos, y seguir trabajando en otra cosa mientras progresa la ejecución de `procesa`.

Observe que esto sólo es posible gracias a la multiprogramación, y que el hecho de que un proceso se esté ejecutando entre bastidores no quiere decir que esté inactivo, sino más bien todo lo contrario. Para demostrarlo, volvamos al ejemplo en el momento en que el sistema ha cargado `procesa` y ha activado al intérprete (dejando `procesa` en el estado «preparado»). El intérprete de órdenes contiene, como en Monoalgorítmez (Programa 18.8), una llamada al sistema para leer una línea del terminal; normalmente aún no nos habrá dado tiempo de teclear nada, y al menos tardaremos unos segundos en hacerlo y terminar la nueva orden con «`ret`». Por tanto, el intérprete queda bloqueado y el sistema activa al proceso preparado que tenga mayor prioridad, que, si no hay otro, será `procesa`. Así, durante esos segundos el proceso entre bastidores se está ejecutando (interrumpido esporádicamente por la rutina de servicio de `tty` cada vez que pulsamos una tecla). Y una vez que tengamos cargado el editor y estemos trabajando con él, el mayor porcentaje de uso de la UCP (los intervalos de tiempo que transcurren desde la pulsación de una tecla hasta la siguiente son mucho mayores que el tiempo de ejecución de la rutina de servicio, como ya comentábamos en el Apartado 2.8) corresponderá a `procesa`.

El usuario puede ejecutar entre bastidores cuantos procesos desee: no tiene más que añadir «`&`» a la orden de ejecución de cada uno[8].

Pues bien, hay algunas acciones sobre el teclado y algunas órdenes que permiten al usuario pasar procesos de escena a bastidores, y viceversa:

- `^Z` (es decir, pulsación simultánea de las teclas «Ctrl» y «z») detiene (bloquea) el proceso que está en escena y activa al intérprete de órdenes.
- `^C` «mata» al proceso en escena.
- `fg` (por «foreground»: en escena) detiene al intérprete y trae a escena el proceso detenido con `^Z` o el primero que esté entre bastidores.
- `bg` (por «background»: entre bastidores) activa (entre bastidores) al proceso detenido con `^Z`.

Las dos primeras no son órdenes para el intérprete (como hay otro proceso en escena, el intérprete no tiene control del terminal). Lo que ocurre es que la rutina de servicio del teclado reconoce `^Z` (ASCII H'1A) y `^C` (ASCII H'03) y el sistema toma las acciones oportunas.

He aquí algunos ejemplos de su utilidad, refiriéndonos al ejemplo anterior del programa `procesa` y el editor de textos:

- Al dar la orden de ejecución del primer programa (`procesa imag1`) nos olvidamos de añadir «`&`» y nos quedamos con el terminal bloqueado. Podemos remediarlo haciendo «`^Z`» (que detiene el proceso `procesa` y nos devuelve al intérprete) seguido de `bg` (que envía a bastidores el proceso detenido y lo activa). A partir de ese momento ya podemos cargar con el intérprete al editor.
- Editando, queremos volver al intérprete momentáneamente para ver (con `ls`) los ficheros que hay en el directorio. Como antes, `^Z` detiene al proceso en escena (ahora, el editor). Ya en el intérprete podemos dar la orden `ls`. Y para volver a poner al editor en escena no tenemos más que dar la orden `fg`.
- Hemos tecleado `procesa imag2 &`, cuando lo que queremos es procesar `imag1`. Podemos proceder de dos maneras: una es obtener el *pid* del proceso `procesa` con `ps` y «matarlo» con `kill`; otra, traerlo a escena con `fg` y «matarlo» con `^C`.

Como vemos, Multialgorítmez no es sólo un sistema **multiusuario**, sino también **multitarea**: cada usuario puede tener varios procesos ejecutándose, aunque en un momento dado sólo uno de ellos esté en escena y los demás entre bastidores, situación que el usuario puede cambiar con las órdenes comentadas.

Llamadas

Vista la *interfaz del usuario*, pasemos a la *interfaz del programador*, es decir, las llamadas al sistema. Son las mismas macros que en Monoalgorítmez, salvo `LD_EX` y `LD_OV`, que quedan sustituidas por otras dos (`FORK` y `EXEC`), y tres nuevas: `WAIT`, `KILL` y `SETPRIORITY`.

Las macros de Monoalgorítmez se tendrán que adaptar al nuevo modelo funcional. Por ejemplo, `CHMOD(#NFICH,PERM)`: el sistema comprueba primero que el usuario está autorizado (*uid* del proceso en curso igual al *uid* del inodo del fichero, o bien *uid* = 0), y luego modifica los bits de permiso del inodo de acuerdo con el contenido de `PERM`. Como siempre que se transmite un argumento por valor, se puede poner su valor inmediato: `CHMOD(#NFICH,#755)` pone permisos de lectura, escritura y ejecución para el propietario del fichero, y de lectura y ejecución para los demás usuarios.

En cuanto a las nuevas macros, no nos detendremos en escribir sus expansiones. El lector puede inventarlas fácilmente, teniendo en cuenta sus parámetros y siguiendo las mismas ideas de la Lección 18.

`FORK` crea un nuevo proceso (hijo), cuya imagen de memoria es exactamente un duplicado de la del padre: el mismo programa y las mismas variables con los mismos valores tanto en la parte de código como en la de pila (matizaremos esto en el Apartado 21.5). El hijo «hereda» también los ficheros abiertos por el padre[9]. La única diferencia es que el sistema asigna un *pid* (mayor que «0») al hijo, cuyo número devuelve al padre en el byte

[7] No cabe esperar que un usuario utilice esta orden directamente (desde el teclado), pero sí resulta muy útil para programas del sistema que no son urgentes y se ejecutan entre bastidores. Estos programas pueden ser guiones (Apartado 21.2) en los que se incluye directamente `nice`, o programas en ensamblador o lenguaje de alto nivel, en los que se utiliza la llamada correspondiente, `SETPRIORITY`.

[8] Como hemos dicho más arriba, el número máximo de procesos en Multialgorítmez es sesenta y cuatro para el conjunto de todos los usuarios. Sería razonable poner también una limitación para cada uno de los usuarios, pero para simplificar no lo haremos.

[9] Más adelante, en la Figura 20.8, veremos las estructuras de datos que permiten que padre e hijos, o, en general, procesos cualesquiera, compartan ficheros.

menos significativo de R13[10], mientras que al hijo le devuelve «0» en R13. Finalmente, pone en estado «preparado» a ambos procesos antes de pasar al distribuidor.

Aunque estemos explicando el modelo funcional (el procesal lo describiremos en el Apartado 20.5) es necesario, para comprender la función de `FORK`, tener una idea general de lo que ocurre internamente en el núcleo. Consideremos este trozo de programa:

```
      FORK
      CMP.B  .13,#0
      BZ     HIJO
      ---       ; instrucciones para el
      ---       ;    proceso padre
      EXIT
HIJO  ---       ; instrucciones para el
      ---       ;    proceso hijo
      EXIT
```

Es muy importante darse cuenta de que tras la ejecución de `FORK` (o, más precisamente, tras la ejecución del núcleo como consecuencia de la llamada) *en la MP hay dos copias idénticas del mismo programa ejecutándose concurrentemente*. Ya hemos dicho que antes de salir del núcleo el distribuidor elige a uno de entre los procesos preparados y lo activa. Más adelante (Apartado 20.6) veremos con qué criterio hace la elección, pero supongamos de momento que activa al padre. Antes de pasarle control pone en R13 el *pid* que ha asignado al hijo. Por tanto, la ejecución proseguirá, después de `BZ HIJO`, con las «instrucciones para el proceso padre». En el momento en que el proceso padre quede bloqueado, o, en general, cuando el distribuidor decida quitarle la UCP (por ejemplo, porque, transcurrido un tiempo, la prioridad del hijo es superior), empezará a ejecutarse el hijo a partir de la instrucción `CMP.B .13,#0`. Como ahora (R13) = 0, las instrucciones que se ejecutarán serán las que vienen tras la etiqueta `HIJO`.

La Figura 20.4 (en la página siguiente) puede ayudar a comprender lo que ocurre con `FORK`. El lector debe seguir la sucesión indicada, teniendo en cuenta lo dicho en el párrafo anterior, y prestando atención a la dirección apuntada en cada momento por el contador de programa, CP. En realidad, CP contiene una dirección lógica, y la dirección física (de veinte bits) en la MP se obtiene en la UGM. Para simplificar la interpretación, hemos representado un caso en el que los programas se cargan ocupando zonas de bytes seguidos en la MP, pero ya sabemos que, en general, la imagen de memoria del proceso se distribuirá en páginas físicas que no necesariamente son contiguas. Se ha supuesto que el núcleo, tras crear al hijo, pasa control al padre (situación «3»), y que éste contiene una llamada `READ` que hace que el núcleo lo bloquee, pasándose el control al hijo (situación «7»). Este hijo contiene una llamada `EXEC`, cuya función vamos a ver inmediatamente.

Considerada aisladamente, no parece, en principio, que `FORK` tenga mucha utilidad. ¿Para qué sirve tener dos procesos idénticos en ejecución? (véase el Ejercicio 16). Normalmente, lo que hace el hijo es pedir la carga y ejecución de otro programa con la llamada `EXEC`.

`EXEC(#NPROG,#ARG,LONG)` *sustituye la imagen en memoria del proceso* por la del módulo de carga almacenado en el fichero cuyo nombre está a partir de la dirección `NPROG` del programa de usuario y deja preparado al proceso (que sigue teniendo el mismo *pid*, aunque haya cambiado el tamaño y el contenido de su imagen de memoria). `ARG` tiene la misma función que en la llamada `LD_EX` de Monoalgorítmez: identifica la dirección de la memoria en el programa de usuario en la que se pueden poner previamente argumentos para el programa cargado: números, cadenas de caracteres y/o descriptores de ficheros. `LONG` es el número de bytes que ocupan esos argumentos. Aquí se plantea un problema: si esa zona de memoria está en el código del programa que contiene `EXEC`, que queda «machacado» por el del programa cargado, ¿cómo

[10] Si se produce un error (por ejemplo, porque no hay sitio en la tabla de procesos o en la memoria), el sistema no crea al hijo, y devuelve al padre un número negativo en el byte menos significativo de R13.

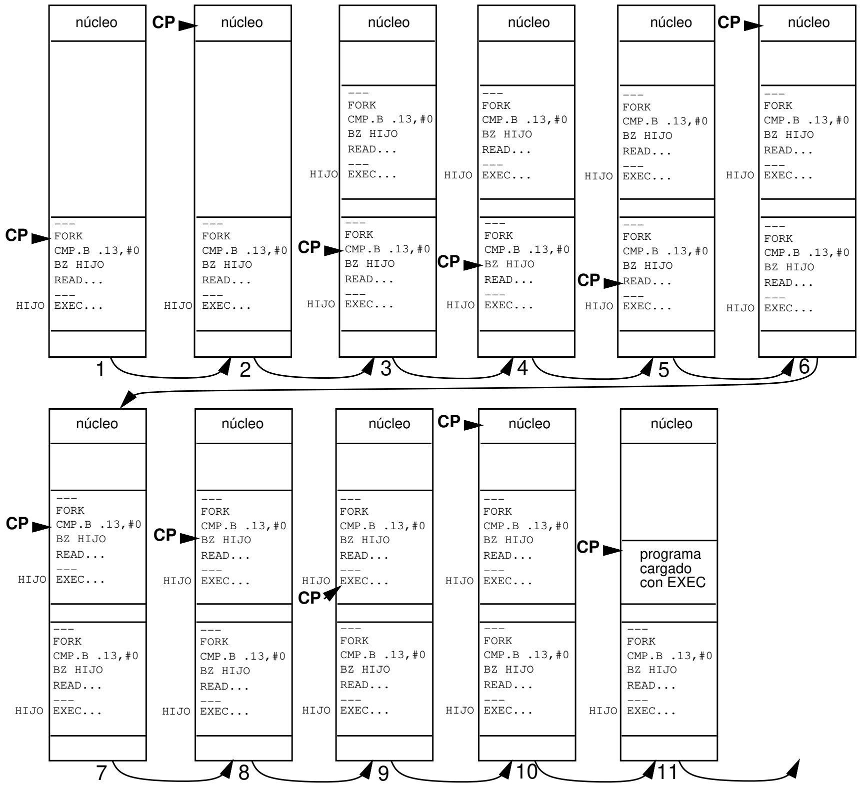


Figura 20.4 Sucesión de estados tras la ejecución de `FORK`, según se explica en el texto.

recoge éste los argumentos? Ahora no podemos proceder como con `LD_EX`, pasándole la dirección de `ARG`, porque cuando el programa cargado fuese a leer en esa dirección los contenidos ya habrían cambiado. Pues bien, no hemos mencionado a la pila:

- Si no hay argumentos, («`#0`» en `LONG`), el sistema deja una pila «limpia», con sólo dos valores (cuatro bytes): el valor inicial de CP (dirección lógica de comienzo de la ejecución, obtenida del módulo cargado) y un valor inicial para RE (H'0100). En la tabla de procesos cuyo detalle veremos en el Apartado siguiente están los valores a poner en los registros R0 a R14 antes de activar al proceso. En el lugar reservado para R14 = PP el sistema pone la dirección lógica donde se ha almacenado ese valor inicial de RE (en el ejemplo de la Figura 20.1 sería H'BFFC).
- Si la llamada contiene argumentos, el sistema, antes de «machacar» al programa, hace una co-

pia interna y luego los mete en la pila, antes (por debajo) de CP y RE, y pone el valor adecuado (dirección donde queda RE) en el lugar reservado para R14. El programa cargado puede recoger los argumentos con instrucciones `POP`, o con `LD` con direccionamiento indexado sobre R14 = PP (Apartado 6.13). Naturalmente, el número y el tipo de argumentos que se pasan debe coincidir con los que espera el programa.

De este modo, se separan las funciones de *creación* de procesos (`FORK`) y *carga y ejecución* de programas (`EXEC`). Las otras llamadas son más fáciles de entender.

`EXIT` tiene la misma función que en Monoalgorítmez: terminar la ejecución del proceso.

`WAIT(PID)` bloquea al proceso hasta que termina el hijo cuyo identificador está en `PID` (byte reservado en el programa de usuario), pero si este hijo ya había terminado (estaba zombi) el proceso no se bloquea. En cualquier caso, el sistema deja en el registro R12 del padre el modo de terminación del hijo (por ejecución de `EXIT`, o porque se le ha matado, o por error de periférico, o por intento de acceso a memoria no asignada, o por violación del modo usuario). Si no existe el proceso identificado por `PID`, o no es hijo del que contiene `WAIT(PID)`, el sistema devuelve un error (número negativo en el byte menos significativo de R13) y no bloquea al proceso.

Una secuencia típica de llamadas es:

```
NFICH DATA    "A:FicheroX",0
PID   RES.B   1
ARG   RES.B   80
      ---
      FORK
      CMP.B   .13,#0
      BN      TRATA_ERROR
      BZ      HIJO
      ST.B    .13,PID
      WAIT(PID)
      ---
HIJO  EXEC(#NFICH,#ARG,#80)
      ---
```

El padre, tras crear al hijo, queda bloqueado (suponiendo que el padre llega a `WAIT(PID)` antes de que el hijo termine) hasta que éste termina (con `EXIT` o de otro modo). En el Apartado 20.5 veremos un ejemplo concreto (el Programa 20.1).

Si se pone «`#0`» como argumento, el proceso espera hasta que termina uno cualquiera de sus hijos; el primero que termina desbloquea al padre y le pasa su *pid* en el byte menos significativo de R13 y el modo de terminación en R12. Si tiene hijos zombis el sistema elige a uno y no bloquea al padre, y si no tiene hijos devuelve un error en R13.

Las dos llamadas restantes se utilizan en el procesamiento de las órdenes `kill` y `nice`. En ambas, «#0» en `PID` identifica al proceso en el cual se hace la llamada, no al proceso nulo:

`KILL(PID)` comprueba primero que el proceso en curso pertenece a *uid* = 0 o bien que el *uid* es igual al del proceso `PID` y «mata» a éste (libera la memoria ocupada y su fila en la tabla de procesos) o lo deja zombi (sólo libera la memoria).

`SETPRIORITY(PID,BASE)` cambia la prioridad de base de un proceso, haciendo antes la misma comprobación anterior, y, además, si $uid \neq 0$, no se permite un número inferior a 5 en `BASE`.

20.4. Modelo estructural

El modelo estructural de Multialgorítmez es como el de Monoalgorítmez (Figura 18.3), añadiéndole los programas que ya hemos mencionado: el planificador y el distribuidor. En el Apartado 20.6 describiremos estos programas. Veamos ahora las nuevas estructuras de datos necesarias.

Tabla de procesos y cola del procesador

Cuando un proceso deja de ejecutarse (es decir, de estar activo) y pasa a bloqueado (porque solicita una transferencia de entrada/salida, o a causa de `WAIT`) o a preparado (porque el distribuidor asigna la UCP a otro proceso con mayor prioridad), hay que conservar en alguna parte de la MP su estatus

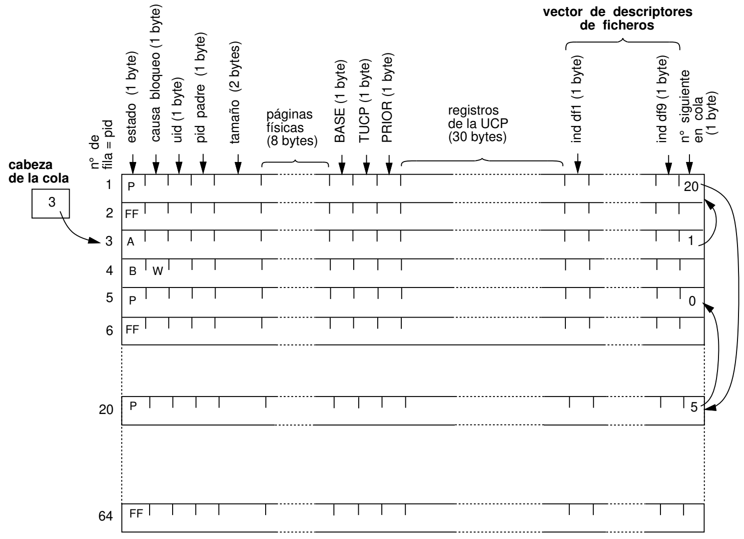


Figura 20.5 Tabla de procesos (y un ejemplo de cola del procesador).

(Apartado 17.9), para que pueda más adelante seguir ejecutándose. En Multialgorítmez, el estatus está materializado por los contenidos de los registros R0 a R15 y RE de la UCP.

Para todo proceso creado, mientras está «vivo» (o zombi), se mantiene un **registro del proceso** (también llamado «descriptor del proceso», o «bloque de control del proceso», o «estructura de tarea»), que contiene su estatus y algunos otros datos también necesarios para la gestión de los procesos. Cada fila de la tabla de la Figura 20.5 es un registro de proceso, que contiene estos campos:

- El estado del proceso: uno de los cuatro de la Figura 20.2 en los que puede estar. Si el estado de un proceso es «nonato» o «muerto» entonces no tiene asignado ningún registro en la tabla; obsérvese que, realmente, estos dos «estados» (es discutible llamarles así) son el mismo.
- La causa del bloqueo (si el proceso está bloqueado): si es porque el proceso ha llegado a `WAIT(PID)` y el hijo aún no ha llegado a `EXIT`, se guarda aquí el *pid* del hijo; si es porque el proceso ha tratado de abrir un fichero que no puede compartirse, se guarda aquí la fila de la tabla de ficheros abiertos, como veremos al explicar el tratamiento de la llamada `OPEN`.
- El identificador del usuario *(uid)* al que pertenece el proceso.
- El identificador *(pid)* del proceso padre.
- El tamaño del programa en bytes.
- Los números de páginas físicas asignadas al proceso, o, más exactamente, los valores a cargar en los registros de la UGM antes de activar a este proceso. Si una de las páginas asignadas es, por ejemplo, la de número D'20 (B'10100), en el correspondiente registro de la UGM se debe cargar este número y el *bit de validez*, es decir, B'10010100 (D'148). Cualquier valor que tenga «0» en el bit más significativo (es decir comprendido entre D'0 y D'127) indicará que

la correspondiente página *no* está asignada a este proceso.

- La prioridad de base del proceso y el tiempo de UCP consumido «recientemente» (informaciones que, como veremos al final de la Lección, sirven para calcular la prioridad en cada momento, y son esenciales para la planificación).
- La prioridad del proceso (el planificador reajusta todas las prioridades cada segundo).
- El estatus del proceso, pero sólo los contenidos de los registros R0 a R14 = PP de la UCP cuando se produce una interrupción y este proceso estaba activo, porque, como ya sabemos, R15 = CP y RE se salvan automáticamente en la pila del proceso, y cuando vuelva a activarse se recuperan con la instrucción `RETI`.
- El **vector de descriptores de fichero**, formado por una sucesión de nueve bytes. Para cada fichero que este proceso tenga abierto (como máximo nueve) el sistema de gestión de ficheros habrá devuelto un descriptor, DF (número entre 1 y 9). El byte de número DF de este vector contiene un índice que, como veremos, permite localizar al fichero en la tabla de ficheros abiertos. (Por la razón que se verá más adelante, el descriptor de un fichero *ya no es*, como en Monoalgorítmez, el número de fila de la tabla de ficheros abiertos).
- El número del siguiente registro en la cola del procesador (lo que explicaremos enseguida).

La **tabla de procesos** tiene sesenta y cuatro filas, para albergar hasta sesenta y cuatro registros de proceso (Figura 20.5). El *pid* de un proceso es el número de fila (entre 1 y 64; el proceso nulo, *pid* = 0, no necesita registro) del registro que tiene asignado en esta tabla. En el byte «estado» se codifican los cuatro estados mencionados, y una codificación (por ejemplo, H'FF) se reserva para indicar que la fila está libre.

Observe que las informaciones de los dos punteros del «prefijo» de los programas cargados en la MP en Monoalgorítmez (Figura 19.2) están ahora incluidas en el registro del proceso, y el conjunto de todas estas informaciones ya no está «repartido» por la MP en los prefijos, sino «centralizado» en la tabla de procesos.

Todos los procesos que en un momento determinado estén *preparados* deberán esperar, o «hacer cola», para conseguir la UCP. Esta cola, llamada **cola del procesador**, se ordena de mayor a menor prioridad, de manera que el distribuidor, cuando haya de asignar la UCP a un proceso, no tiene más que elegir al que está en la cabeza. La reordenación de la cola se hace siempre en el núcleo, antes de pasar al distribuidor. El último proceso de la cola (o el único, si no hay procesos preparados) es siempre el proceso nulo.

Para implementar esta cola podemos hacer como sugiere la Figura 20.5: una variable que contiene el *pid* correspondiente a la cabeza de la cola, y en cada registro de proceso un byte que contiene el *pid* del siguiente proceso en la cola. Reordenar la cola implica únicamente escribir nuevos valores en estos bytes y actualizar el valor de la cabeza. En la situación indicada en la Figura 20.5, el proceso activo es el 3 y después están en la cola el 1, el 20 y el 5. En el último byte de éste, «0» indica que sólo le sigue el proceso nulo. Los demás procesos que pueda haber en la tabla están bloqueados o zombis.

La tabla de periféricos y la cola de demandas de entrada/salida

La estructura de la Figura 19.3 no sirve para Multialgorítmez. La razón es que sólo tiene un registro de demanda para cada registro de periférico. Y basta con uno en Monoalgorítmez, porque si en el momento en que aparece una demanda (`READ_A`, `WRITE_A`, `READ` o `WRITE`) el periférico está ocupado, el núcleo entra en un bucle de espera (proceso nulo) hasta que el periférico queda libre (Apartado 19.7). Pero con multiprogramación hemos de proceder de otro modo. Si un proceso hace una demanda y el periférico está ocupado es porque otro proceso (o, quizás, este mismo) está haciendo uso del periférico. No hay que activar sin más al proceso nulo, puesto que puede haber un tercer proceso preparado. Lo que hay que hacer es *poner en cola a las demandas*.

Por tanto, en las filas de la tabla de periféricos no ponemos un «registro de la demanda», sino un puntero al comienzo de una cola de registros de demanda. En la Figura 20.6 se ilustra esta disposición. En lugar de un puntero a memoria (dos bytes) utilizamos un byte con el número del primer registro de la cola, y en cada uno de éstos el número del siguiente registro. H'FF puede reservarse para señalar el final de la cola[11].

Sin embargo, sí que podemos mantener un solo «registro de la transferencia» para cada periférico, con las mismas informaciones que en la Figura 19.3, porque el periférico no puede atender a más de una petición en cada momento.

Creando una nueva tabla, la **tabla de demandas**, cuyas filas (que limitaremos arbitrariamente a sesenta y cuatro) pueden contener registros de demandas, el nuevo diseño puede verse en la Figura 20.7. Si se compara con la Figura 19.3 se verá que aparecen algunos campos nuevos:

- En los registros de los periféricos, «demandas pendientes», es decir, el número de demandas que hay en cola; este dato es necesario para que el núcleo sepa cuándo debe pasar al estado preparado el proceso correspondiente al gestor del periférico.
- También en los registros de los periféricos (y en lugar de un registro de demanda), el número del primer registro de demanda de la cola (si la hay, es decir, si el contenido de «demandas pendientes» es mayor que cero).
- En los registros de las demandas, además del número del siguiente en la cola (H'FF si es el último), el *pid* del proceso que ha solicitado la transferencia. Esto es necesario para que cuando termine

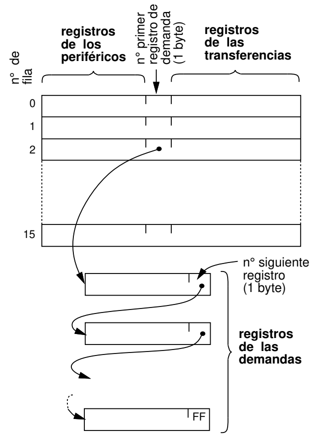


Figura 20.6. Cola de registros de demandas de entrada/salida asociada a un periférico.

la transferencia, y la rutina de servicio de la interrupción llame al gestor del periférico, éste cambie el estado del proceso de bloqueado a preparado y lo ponga en la cola del procesador.

Como hemos dicho, la tabla de demandas puede contener varias colas, asociada cada una a un periférico. Estas colas pueden organizarse de acuerdo con las prioridades de los procesos o, simplemente, siguiendo el principio «FIFO» (First-In, First-Out): las demandas se van poniendo en cola conforme van llegando, y se van sirviendo en el orden en que han llegado.

Por último, en relación con los periféricos, ahora tenemos cuatro terminales (y dos zonas de entrada/salida para cada uno), que ocuparán ocho filas de la tabla de periféricos.

La tabla de ficheros abiertos y la tabla de posiciones de ficheros

Tampoco sirve la estructura de la Figura 19.4 para la tabla de ficheros abiertos. La razón es parecida a la esgrimida más arriba para la tabla de peri-

[11]Podría haber así hasta 255 registros en una cola, pero este número queda muy reducido, porque, como siempre (Apartado 19.2), utilizaremos una estructura de datos estática, la tabla de demandas, con un número limitado de filas, para contener a todas las posibles colas de registros de demanda.

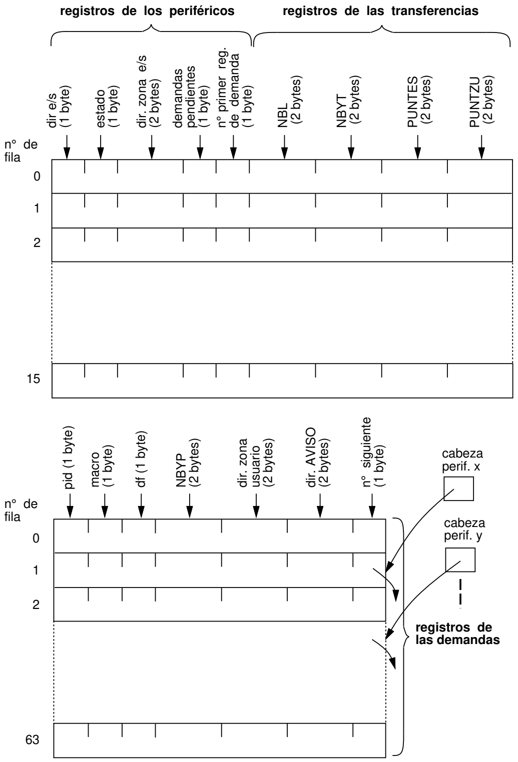


Figura 20.7 Tabla de periféricos y tabla de demandas de entrada/salida.

féricos. Pueden coexistir procesos independientes entre sí que hacen uso del mismo fichero, y cada uno tendrá su «posición». Por ejemplo, dos usuarios leyendo en una base de datos.

Podríamos pensar en una alternativa: si cada proceso debe tener para cada fichero abierto su propia posición, ¿por qué no incluir estas posiciones en el registro del proceso, una al lado de cada uno de los descriptores de fichero? Pero esto tampoco funciona, porque *a veces sí es necesario* que dos o más procesos compartan las posiciones de sus ficheros abiertos. Así ocurre con la llamada `FORK`: el hijo hereda los ficheros abiertos por el padre, e interesa que cuando termine (con `EXIT`) el padre tenga las posiciones actualizadas. Por ejemplo, el padre abre un fichero y crea (con `FORK`) un hijo que (tras `EXEC`) escribe 500 bytes; termina el hijo y el padre crea otro hijo que vuelve a escribir 500 bytes en el mismo fichero. Si queremos que estos 500 bytes no se escriban borrando los primeros, sino a continuación de ellos, será necesario que la posición actualizada por el primer hijo

(500) esté compartida con el padre y con el segundo hijo.

Por tanto, hay que considerar dos casos: el de los procesos «emparentados» (en el sentido de que uno es descendiente del otro), que comparten las posiciones de sus ficheros abiertos, y el de los «extraños», en los que para un mismo fichero compartido las posiciones son independientes en cada proceso. La solución es disponer una tabla intermedia entre la de procesos y la de ficheros abiertos, la **tabla de posiciones de ficheros**. A cada descriptor de fichero de los registros de los procesos se le asigna un número que es un índice a la tabla de posiciones; si dos procesos están emparentados, los índices del mismo fichero coincidirán en ambos. En la tabla de posiciones, cada fila contiene, además de la posición, un índice a la tabla de ficheros abiertos, el número de procesos que están apuntando a esta fila y el modo (lectura o escritura) en que está abierto el fichero. Finalmente, en la tabla de ficheros abiertos se incluirán las informaciones de la Figura 19.4 (menos la posición) y un campo nuevo que contiene el número total de procesos que están haciendo uso de ese fichero.

La cosa no es tan complicada como puede parecer en una primera lectura del párrafo anterior. Un ejemplo con una ilustración gráfica ayuda a comprenderla. En la Figura 20.8 se muestra una situación en la que:

- Los procesos 1 y 3 están emparentados (por ejemplo, 1 es padre de 3), y el proceso 2 es independiente de ambos. Los tres están operando sobre un mismo fichero (el de fila 9 en la tabla de ficheros abiertos).
- El proceso 1 ha abierto el fichero para escritura y ha recibido como descriptor el número 8. Es decir, se ha ejecutado `OPEN(#NFICH,#1)`, y el sistema ha devuelto 8 en R13 El sistema también ha asignado una fila, la 1, en la tabla de posiciones y otra, la 9, en la de ficheros abiertos. En la octava componente («df8») del vector de descriptores de ficheros (dentro del registro del proceso 1) el sistema ha puesto «1» (índice a la tabla de posiciones), y en la fila 1 de la tabla de posiciones, «9» (índice a la tabla de ficheros abiertos).

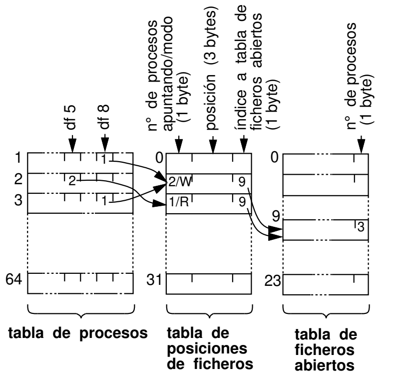


Figura 20.8. Tabla de posiciones de ficheros, intermedia entre la tabla de procesos y la de ficheros abiertos.

- El proceso 1 ha creado después, con `FORK`, al proceso 3. Por tanto, éste ha heredado todos los ficheros que su padre tenía abiertos, con sus índices. Concretamente, en el descriptor de fichero 8 tiene, como su padre, «1». La fila 1 de la tabla de posiciones contiene «2» en «número de procesos apuntando».
- El proceso 2 ha abierto el mismo fichero para lectura (`OPEN(#NFICH,#0)`). El sistema ha devuelto en este caso `DF` = 5. Como el proceso es independiente de los otros dos, en el descriptor de fichero 5 tiene otro índice, concretamente «2», para que el proceso tenga su propia fila en la tabla de posiciones. Pero la fila 2 de la tabla de posiciones apunta también a la fila 9 de la tabla de ficheros abiertos, porque se trata del mismo fichero.
- En la tabla de ficheros abiertos, la fila 9 del fichero abierto por dos veces contiene en este momento «3», porque 3 son los procesos que están haciendo uso de ese fichero.

No es necesario llegar al modelo procesal para darse cuenta de la función del campo «número de procesos» de la tabla de ficheros abiertos.

Si, por ejemplo, el proceso 2 cierra el fichero (`CLOSE(DF)`) no por eso hay que liberar la fila de la tabla, porque aún quedan otros dos procesos. El sistema considerará el fichero «definitivamente cerrado» y liberará la fila (y actualizará en su caso el inodo en el disco) cuando el valor de este campo se reduzca a «0».

Mapas de bits

En el Apartado 18.8 vimos que los datos sobre los inodos libres y sobre los bloques libres se mantienen condensados en dos mapas de bits contenidos en los superbloques de los discos y copiados permanentemente en la MP. A éstos añadiremos en Multialgorítmez otro, el de *páginas físicas libres*, esencial para el sistema de gestión de la memoria. A diferencia de los anteriores, éste sólo existe en memoria (en la zona del núcleo), no en el disco.

El fichero de contraseñas

Las estructuras de datos comentadas residen permanentemente en la MP. El fichero de contraseñas, sin embargo, se encuentra en el disco (con el nombre `psswd`). Contiene 256 registros (uno por cada posible usuario) con los siguientes campos:

- *uid* y *gid*.
- «login» del usuario.
- Contraseña cifrada.
- Nombre del usuario y otros datos personales.
- Nombre del fichero del intérprete de órdenes.

El último campo carece de sentido si sólo existe un intérprete. Pero normalmente se dispone de varios, y un usuario puede elegir entre uno y otro (o incluso escribir el suyo propio). En el proceso de «entrada» al sistema (o «login») se carga el intérprete que corresponda, de acuerdo con lo indicado en el fichero de contraseñas. Para que esto fuese posible en nuestro sistema tendríamos que añadir una orden al modelo funcional del Apartado 20.3: `chsh` (cambia la *shell*), acompañada del nombre del fichero.

20.5. Modelo procesal

Describiremos el modelo procesal de manera más sintética que el de Monoalgorítmez, señalando las diferencias con éste en cuanto al tratamiento de las llamadas al sistema, y presentando esquemáticamente el funcionamiento de los nuevos componentes de software.

Obviaremos muchos pormenores para concentrarnos en lo fundamental. Como, por ejemplo, las comprobaciones (conceptualmente triviales, pero complicadas en sus detalles) que el sistema ha de hacer antes de proceder a la ejecución de las llamadas para ver si al proceso se le puede permitir la operación que pide, en función de su *uid* y de los bits de permiso en los inodos. Tampoco entraremos en los detalles de la implementación, cuya codificación en ensamblador sería bastante engorrosa. Varias llamadas transmiten al núcleo una dirección en la que se encuentran parámetros: el nombre de un fichero en `CREATE`, `EXEC`, etc., el byte de aviso en `WAIT_A`, los argumentos para el programa en `EXEC`, etc. Naturalmente, esta dirección pertenece a la imagen de memoria del proceso, y para acceder a ella el núcleo tendrá que cambiar temporalmente el contenido de algún registro de la UGM y escribir H'02 en el puerto 20 (Apartado 7.10). Lo mismo ocurre para los intercambios entre zonas de entrada/salida y de usuario en el tratamiento de las llamadas `READ` y `WRITE`.

Iniciación

El proceso de arranque es, en líneas generales, igual que el detallado para Monoalgorítmez (Apartado 18.9): ejecución de un programa en ROM que hace algunas comprobaciones del hardware y termina leyendo y cargando en memoria el primer sector del disco, carga del núcleo en memoria, y ejecución de un programa como «`INICIA`» que lleva a memoria los inodos de los terminales y los superbloques y pone los valores de los vectores de interrupción y de las tablas de periféricos y de ficheros abiertos.

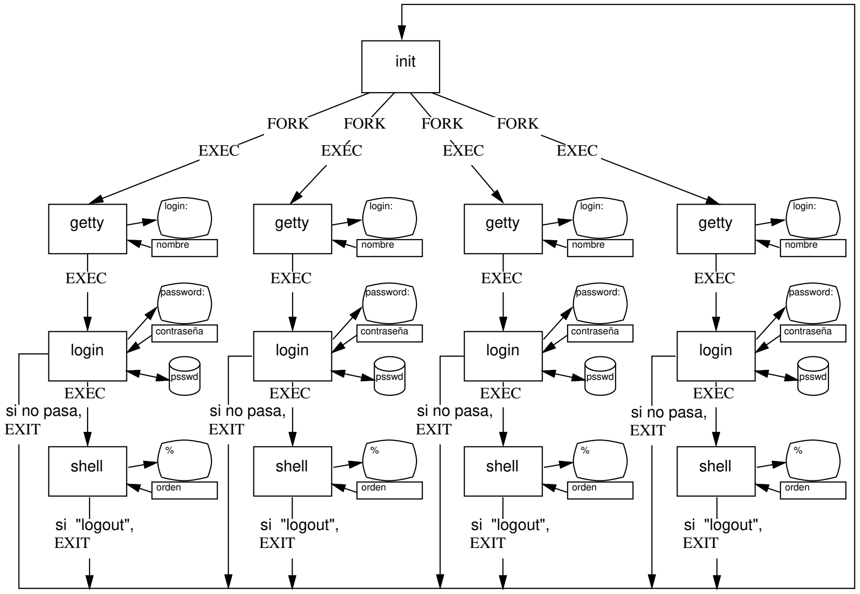


Figura 20.9 Modelo procesal global.

Pero, a diferencia de Monoalgorítmez, este proceso no termina cargando el intérprete de órdenes[12], sino un programa llamado «`init`» cuya función es repartir inicialmente el sistema entre los cuatro terminales. `init` es ya un proceso que se ejecuta en modo usuario, y es el único proceso de usuario que está siempre en memoria (aunque la mayor parte del tiempo está bloqueado).

La Figura 20.9 ilustra el funcionamiento global. `init` incluye cuatro llamadas `FORK` para crear cuatro procesos hijos. *El lector debería analizar cuidadosamente el Programa 20.1* para comprobar que ha comprendido bien cómo se crean procesos hijos con `FORK` y cómo éstos se ejecutan concurrentemente entre sí y con su padre. Cada uno de los hijos pide la ejecución de un programa, «`getty`», que escribe en un terminal (cuyo número, entre 0 y 3, se le pasa como argumento en la llamada `EXEC`) los caracteres «`login:`» y queda a la espera de que un usuario teclee algo. Si inicialmente no hay nadie en ninguno de los cuatro terminales el sistema se queda con cinco procesos bloqueados (`init` y los cuatro `getty`) y pasa a ejecutar el proceso nulo.

Cuando en un terminal se escribe un nombre, se activa el proceso bloqueado (uno de los `getty`). El programa `getty`, cuya codificación ya omitimos, contendrá llamadas para escribir «`login:`» y esperar la entrada desde el terminal, y seguirá con una `EXEC` para ejecutar el programa `login`, pasándole como argumento el nombre recibido del usuario. Este programa `login` empieza haciendo algo parecido a `getty`: escribe «`password:`» en el terminal y queda esperando a que el usuario teclee

[12]Cuando un usuario entre en el sistema se cargará el intérprete indicado en su fila del fichero `psswd`.

```
MODULE INIT
EXPORT INIT_ENT
  PID0     RES.B  1
  PID1     RES.B  1
  PID2     RES.B  1
  PID3     RES.B  1
  NGETTY   DATA.B "A:getty",0
  NTTY0    DATA.B 0
  NTTY1    DATA.B 1
  NTTY2    DATA.B 2
  NTTY3    DATA.B 3
  INIT_ENT FORK
           CMP.B  .13,#0
           BZ     TTY0
           ST.B   .13,PID0
           FORK
           CMP.B  .13,#0
           BZ     TTY1
           ST.B   .13,PID1
           FORK
           CMP.B  .13,#0
           BZ     TTY2
           ST.B   .13,PID2
           FORK
           CMP.B  .13,#0
           BZ     TTY3
           ST.B   .13,PID3
  SIGUE    WAIT(#0)
           CMP.B  .13,PID0
           BZ     F0
           CMP.B  .13,PID1
           BZ     F1
           CMP.B  .13,PID2
           BZ     F2
           CMP.B  .13,PID3
           BZ     F3
           BR    SIGUE
F0    FORK
      CMP.B  .13,#0
      BZ     TTY0
      ST.B   .13,PID0
      BR     SIGUE
F1    FORK
      CMP.B  .13,#0
      BZ     TTY1
      ST.B   .13,PID1
      BR     SIGUE
F2    FORK
      CMP.B  .13,#0
      BZ     TTY2
      ST.B   .13,PID2
      BR     SIGUE
F3    FORK
      CMP.B  .13,#0
      BZ     TTY3
      ST.B   .13,PID3
      BR     SIGUE
TTY0  EXEC(#NGETTY,#NTTY0,#1)
TTY1  EXEC(#NGETTY,#NTTY1,#1)
TTY2  EXEC(#NGETTY,#NTTY2,#1)
TTY3  EXEC(#NGETTY,#NTTY3,#1)
     END
```

Programa 20.1 El programa `init`.

su contraseña. Cuando tal cosa ocurre se compara el nombre y la contraseña con los almacenados en el fichero de contraseñas, `psswd`. Si no hay coincidencia, el proceso termina (con `EXIT`; si el lector sigue el programa 20.1, verá que `init` vuelve a llamar a `getty` sobre ese mismo terminal). Si la hay, el usuario ha quedado autenticado, y el proceso `login` ejecuta una `EXEC` para cargar al intérprete en la MP. «`init`», «`getty`», «`login`» y «`shell`»[13] son los nombres típicos de UNIX. El intérprete escribe la invitación[14] y queda a la espe-

[13]Como normalmente hay varios intérpretes a elegir, su nombre en realidad puede ser «`tcsh`» o «`bash`», etc.

[14]«`%`», «`#`» y «`$`» son invitaciones usuales, pero puede ser cualquier carácter, o una secuencia de caracteres, a elección del usuario (Apartado 21.4).

ra de una orden. Este intérprete es, esencialmente, como el que vimos en Monoalgorítmez (Programa 18.8), pero con algunas diferencias:

- En lugar de `LD_EX` se utilizarán `FORK`, `WAIT` y `EXEC`: el efecto de una orden es crear un proceso hijo del intérprete.
- Habrá que tener en cuenta si está presente «`&`», para que el proceso creado se ejecute entre bastidores (haciendo `FORK` y `EXEC`, pero no `WAIT`).
- Cuando la orden sea «`logout`» el intérprete hará la llamada `EXIT` que «despierta» al padre, `init`.
- La ejecución de las órdenes precedidas de `nice` se realiza haciendo primero `FORK`, luego la llamada `SETPRIORITY(#0,BASE)` en el hijo y finalmente `EXEC` sobre el nombre de la orden.

La rutina de interrupciones de programa

El programa de la RIP explicado para Monoalgorítmez (Programa 19.1) necesitaría asimismo algunas modificaciones:

- El tratamiento de `WAIT_A(#AV)` es distinto: si (`AV`) = 0, ahora no hay que pasar a ejecutar el proceso nulo, sino bloquear al proceso activo y activar al primero de los preparados. Por tanto, la RIP actuará sobre la tabla de procesos, poniendo «bloqueado» en el campo «estado» del registro de proceso activo (el primero de la cola) y sacándolo de la cola. El último campo de ese registro señala al primero de los procesos preparados (que el distribuidor activará luego). Si el contenido de ese último campo es «0» es que no hay ningún proceso en la cola, y el distribuidor pasará control al proceso nulo, como en el caso de Monoalgorítmez.
- Tiene que incorporar las tres nuevas llamadas: `WAIT` y `SETPRIORITY` se tratan en la misma RIP, y `KILL` en el sistema de gestión de la memoria. (`FORK` y `EXEC` también son nuevas, pero sustituyen a `LD_EX` y `LD_OV`, y, como ellas, se tratan en el sistema de gestión de la memoria).

El tratamiento de `SETPRIORITY` es trivial. El de `WAIT(PID)` es similar al de `WAIT_A(#AV)`[15], pero la condición para que el proceso quede bloqueado es el estado del proceso hijo. El valor de `PID` identifica al registro del hijo en la tabla de procesos. Si éste no está zombi, la RIP procede bloqueando al proceso activo, como antes. Pero si su estado es zombi, es porque este proceso hijo ha ejecutado `EXIT` previamente. Cuando un proceso ejecuta `EXIT` (o bien cuando se le mata con la llamada `KILL` o por algún error en el SGM o el SGF) y su padre aún no ha ejecutado `WAIT`, se mantiene su registro en la tabla de procesos, pero se saca de la cola, se libera la memoria ocupada por su imagen, y en el campo «estado» se marca como «zombi». En otro campo del registro (uno cualquiera, puesto que el proceso ya no se va a activar más; por ejemplo, en el byte menos significativo de los reservados para R0) se introduce la codificación del modo de terminación (por ejemplo, H'00 para `EXIT`, H'01 para `KILL`, etc.). La RIP, al interpretar `WAIT(PID)` «mata» definitivamente al zombi `PID` (pone «libre» en su campo de estado), deja preparado al proceso padre e introduce la codificación sobre el modo de terminación (puesta anteriormente en el registro del zombi) en el byte menos significativo de los reservados para R12 en el registro del proceso padre, y el *pid* de este zombi en el byte menos significativo de los reservados para R13.

En el caso de `WAIT(#0)` la RIP explora la tabla de procesos buscando procesos hijos y realiza las operaciones oportunas para satisfacer el modelo funcional que hemos descrito al final del Apartado 20.3.

Pero las *modificaciones más importantes conceptualmente* son también las más fáciles de entender. En la RIP, en la RCI, y en todas las rutinas de servicio que no sean llamadas por la RCI, deben realizarse las siguientes operaciones:

[15]En una se pasa la dirección del parámetro y en otra su valor. Piense el lector en las diferencias entre ambas, y, si es necesario, vuelva a leer el Apartado 6.13, y no le costará mucho encontrar la explicación.

Al empezar:

- Salvar el estatus del proceso activo (es decir, los contenidos de los registros de la UCP, excepto R15 = CP y RE, cuyos valores se han guardado en la pila del proceso) en su registro de la tabla de procesos.

- Cargar en el registro R14 = PP el valor actual del puntero de pila del núcleo, que se mantiene en una variable global (una palabra) dentro del módulo principal del núcleo. El valor inicial de esta variable es H'FC00 (la imagen de memoria del núcleo es la de la Figura 19.1, salvo que no contiene al intérprete ni procesos de usuario y que la pila ocupa 7 KB).

Al terminar:

- Salvar el valor actual del PP del núcleo (que, normalmente, habrá quedado con su valor inicial, H'FC00) en la variable reservada para ello,

- En lugar de `RETI` (como se hacía en el Programa 19.1), bifurcar al distribuidor, que se encargará de activar al primer proceso de la cola, de restaurar sus registros y de volver a ese proceso con `RETI`, como veremos en el Apartado siguiente.

El sistema de gestión de la memoria

Las llamadas de las que se ocupa el sistema de gestión de la memoria (SGM) son `FORK`, `EXEC`, `EXIT` y `KILL`. Las dos primeras implican asignación de memoria, y las otras dos, liberación de memoria.

Para la asignación de memoria el SGM tiene que explorar el mapa de bits de páginas físicas y comprobar que queda espacio suficiente en función de la cantidad de memoria que se necesita, determinada, en el caso de `FORK`, por el tamaño de la imagen de memoria (que se encuentra en el registro del proceso), y en el caso de `EXEC`, por la longitud del código (para lo cual hay que leer del disco el primer bloque asignado al fichero y mirar su cabecera).

Esquemáticamente (y obviando detalles, como lo que hay que hacer en caso de fallos y las manipulaciones necesarias en los registros de la UGM), las acciones a realizar para `FORK` son:

1. Comprobar si hay un registro libre en la tabla de procesos.
2. Comprobar (explorando el mapa de bits de páginas físicas) si hay sitio en la MP para copiar la imagen de memoria.
3. Copiar las páginas del proceso activo en las páginas asignadas. Como todas estas páginas están fuera de la imagen de memoria del núcleo, esto requiere manipular algunos registros de la UGM.
4. Actualizar la tabla y la cola de procesos (añadiendo el nuevo y poniéndolo preparado) y el mapa de bits de páginas físicas.
5. Rellenar adecuadamente los campos del registro asignado en la tabla de procesos (poniendo, entre otras cosas, los números de las páginas asignadas, y copiando en los campos de los registros R0 a R12 y R14 los mismos valores que tiene el padre; en el campo correspondiente a R13 pone 0, y en el campo correspondiente a R13 en el registro del padre, el número del registro asignado al hijo).
6. Volver a la RIP, que bifurca al distribuidor para que active al primer proceso de la cola.

Y para `EXEC`:

1. Comprobar permisos (el fichero nombrado en `EXEC` debe tener X = 1 para todos, o para el *gid* al que pertenece el usuario, o para este *uid*).
2. Hacer una copia de los argumentos de `EXEC`.
3. Si el código del programa a cargar ocupa menos que el actual, liberar las páginas que no sean necesarias. Si ocupa más, asignar las que falten (tras comprobar que hay páginas suficientes). En cualquier caso, actualizar el mapa de bits de páginas físicas.

4. Copiar el fichero en las páginas asignadas. Como en el paso 3 anterior, será necesario cambiar algún registro de la UGM.
5. Pasar por la pila del programa cargado (manipulando también la UGM) los argumentos y los valores iniciales de CP y RE, como hemos explicado en el Apartado 20.3.
6. Actualizar el registro del proceso en la tabla de procesos. Concretamente, poner en el lugar reservado para R13 «0» si no se ha producido error, o un código que identifica el error, y en el de R14 el valor adecuado para el puntero de pila, de acuerdo con el número de argumentos que se pasan. Los demás registros pueden dejarse como están, o ponerse a cero.
7. Volver a la RIP.

En cuanto a `EXIT`, el SGM tiene que:

1. Poner una codificación en algún lugar del registro del proceso, indicando que la causa (modo) de terminación es `EXIT` (por ejemplo, como hemos dicho antes, H'00 en el byte menos significativo reservado para R0).
2. De acuerdo con el *pid* del padre de este proceso, mirar si este padre está bloqueado esperando a esta `EXIT`:
   - Si es así, liberar el registro de este proceso de la tabla de procesos (poniendo «libre» en el campo «estado» y sacándolo de la cola del procesador), desbloquear al padre (poniendo «preparado» en el campo «estado» de su registro e introduciéndolo en la cola del procesador) y comunicar el *pid* de este proceso al padre, en el registro R13, y el modo de terminación en R12.
   - Si no, marcar como zombi a este proceso (poniendo «zombi» en «estado» y sacándolo de la cola).
3. Liberar la memoria poniendo «0» en los bits del mapa de páginas físicas.
4. Explorar la tabla de procesos para ver si este proceso tiene hijos zombis; a todos ellos se les libera definitivamente su correspondiente registro del proceso (se pone «libre» en el campo «estado»).
5. Volver a la RIP.

Si el proceso que hace `EXIT` tiene hijos «vivos» no hay que matarlos, porque pueden estar realizando todavía su tarea. Esto plantea un problema: cuando esos hijos terminen, como su padre ya no existe, ¿quedarían zombis para siempre? La solución es muy sencilla: a los procesos que quedan «huérfanos» se les modifica su registro en la tabla de procesos de modo que pasen a ser hijos de `init` (el antepasado común de todos, que nunca muere, Programa 20.1). Cuando éste ejecute `WAIT(#0)`, morirán todos sus hijos zombis.

El tratamiento de `KILL` es casi igual que el de `EXIT`. Sólo cambia en que:

1. El SGM debe realizar una comprobación previa: que el *uid* del proceso activo es o bien *uid* = 0 o bien igual que el contenido en el registro del proceso.
2. Las operaciones descritas para `EXIT` se realizan ahora no sobre el proceso activo, sino sobre el indicado por `PID`.
3. Introduce otra codificación (por ejemplo, H'01) sobre la causa de terminación.

El sistema de gestión de ficheros

El sistema de gestión de ficheros (SGF) atiende a las llamadas para creación, borrado, apertura, cierre, lectura y escritura, y a `POS`, `STAT` y `CHMOD`. Sólo comentaremos, en líneas generales, las modificaciones que serían necesarias con respecto a lo explicado en el Apartado 19.7.

En general habrá que tener en cuenta:

- Las comprobaciones previas sobre permisos de acceso: un usuario no puede borrar ficheros ajenos (o, dicho en términos más ligados a la implementación, un proceso sólo puede ejecutar `DELETE(#NFICH)` si el *uid* que hay en el inodo del fichero coincide con el que hay en el registro del proceso activo, o si éste es un proceso del superusuario); un usuario no puede acceder a un fichero si no tiene permiso para la operación solicitada (de acuerdo con los bits de permiso del inodo de ese fichero), etc.

- Las nuevas estructuras para localizar la fila de la tabla de ficheros abiertos (Figura 20.8).

En las llamadas para transferencias la modificación más importante se refiere a las acciones a realizar cuando el periférico está ocupado. Ahora no hay que activar al proceso nulo («EI; espera a ocupado = 0; DI» en las Figuras 19.9 y 19.10), sino bloquear al proceso activo (marcándolo como tal en el campo «estado» de su registro dentro de la tabla de procesos), sacarlo de la cola del procesador y poner el registro de demanda en la cola de demandas asociada a ese periférico (Figuras 20.6 y 20.7). El distribuidor se encargará de activar al primer proceso de la cola del procesador.

Para las llamadas de apertura y cierre hay que tomar una precaución especial, porque si permitiéramos que dos o más procesos compartieran, por ejemplo, la misma impresora, correríamos el riesgo de obtener un listado mezclado e ininteligible de los resultados. Imaginemos dos procesos, P1 y P2, que envían textos de 1.000 caracteres a la impresora en dos «trozos» de 800 y 200 bytes. Ambos tienen las llamadas:

```
DLP    RES.B  1
NLP    DATA   "A:lp",0
ZONA1  RES.B  800
ZONA2  RES.B  200
       ---
       OPEN(#NLP,#1)
       ST.B   .13,/DLP
       WRITE(/DLP, #ZONA1, #800)
       ---
       WRITE(/DLP, #ZONA2, #200)
       ---
```

Tras la primera petición, el proceso que la ha hecho, supongamos que es P1, queda bloqueado después de iniciarse la transferencia; se activa el otro proceso, P2, y al llegar a WRITE su demanda se pone en la cola y se bloquea también. Si no hay ningún proceso más se activará el nulo, hasta que el gestor de la impresora, tras la transferencia de los 800 bytes, ponga preparado a P1. Pero como existe una demanda pendiente en la cola de la impresora, el gestor iniciará el tratamiento de esa demanda, es decir, la escritura de los 800 bytes solicitada por P2. El resultado es que se habrán mezclado en el listado final los trozos de ambos procesos: primer trozo de P1, primero de P2, segundo de P1 y segundo de P2.

Una forma de evitar esta situación es el «*spooling*», artificio que explicaremos en la siguiente Lección. Otra consiste, sencillamente, en asignar la impresora (o cualquier otro recurso que no pueda compartirse) al primer proceso que la solicite y no permitir que ningún otro proceso haga uso de ella hasta que el primero la libere. Veamos cómo se puede implementar esta idea en Multialgorítmez.

En los inodos, dentro del campo «tipo», hay dos bits que indican si el fichero puede compartirse o no, uno para lectura y otro para escritura. El campo «tipo» del inodo de lp, además de señalar que se trata de un fichero especial de caracteres, tendrá el bit correspondiente a escritura con el valor «0» (no puede compartirse). Cuando la RIP transfiere la llamada OPEN(#NLP, #1) al SGF, éste, tras las comprobaciones previas (por ejemplo, sólo acepta «1» como tercer argumento, que indica que se abre para escritura, porque en el inodo de lp los permisos para lectura estarán a «0») mira en la tabla de ficheros abiertos. Si el inodo de lp ya está en ella, como en él aparece la información de que no puede compartirse, el SGF bloquea al proceso (y lo saca de la cola del procesador) y pone en su registro de la tabla de procesos que la causa es la espera por un fichero que no puede compartirse, así como el número de fila del inodo en la tabla de ficheros abiertos (para poder terminar la operación OPEN cuando la impresora quede libre).

Para tratar la llamada CLOSE, el SGF decrementa en una unidad el contenido del campo «número de procesos» de la tabla de ficheros abiertos. Si el resultado es «0» explora la tabla de procesos para ver si los hay bloqueados por estar esperando abrir este fichero, y si es así vuelve a poner «1» en el campo, desbloquea (pone preparado) al que tenga mayor prioridad y lo mete en la cola del procesador. Si ninguno está esperando, el fichero puede cerrarse definitivamente: deja «0» en el campo «número de procesos» de la tabla de ficheros abiertos significando que la fila está libre, y si el inodo ha sido modificado y era un fichero ordinario lleva al disco la copia del inodo.

20.6. El distribuidor y el planificador

Como ya hemos dicho al hablar de la RIP, al final de esta RIP, de la RCI y de todas las rutinas de servicio de interrupción a las que no se entre por RCI, en lugar de RETI se bifurca al distribuidor. En este momento pueden haber pasado, durante la ejecución del núcleo, muchas cosas. Por ejemplo:

- Que se haya ejecutado una llamada WAIT_A(#AV) con (AV) = 1, en cuyo caso el estado del proceso que provocó la ejecución de la RIP sigue siendo «activo», y los otros no han cambiado su estado, por lo que hay que seguir ejecutando ese proceso.
- Que se haya ejecutado WAIT_A(#AV) con (AV) = 0. El proceso habrá quedado bloqueado, la RIP habrá actualizado la cola del procesador (sacando de ella a ese proceso), y el distribuidor tiene que activar al primer proceso de la cola.
- Que se haya ejecutado WAIT. El proceso habrá quedado bloqueado o no según que el hijo (o uno de ellos) haya o no terminado, y la RIP también habrá actualizado la cola.
- Que se haya ejecutado una EXIT, que habrá sacado al proceso de la cola y habrá actualizado a ésta.
- Que se haya ejecutado OPEN, que puede haber bloqueado (y sacado de la cola) o no al proceso en curso.
- Que se haya ejecutado CLOSE, que puede haber desbloqueado (e introducido en la cola) a un proceso.
- Que se haya ejecutado una llamada de transferencia; el proceso puede igualmente haber quedado bloqueado o no, y el sistema de gestión de ficheros habrá actualizado la cola.
- Que se haya ejecutado una rutina de servicio de interrupciones; puede ocurrir que ésta no haya modificado el estado de ningún proceso (porque haya transferido caracteres, pero sin concluir toda la transferencia), o que, por haber terminado una transferencia, haya dejado preparado (e insertado en la cola del procesador) algún proceso previamente bloqueado[16]. En este segundo caso puede ocurrir que el proceso tenga mayor prioridad que los demás preparados y que el que hasta el momento de la interrupción estaba activo; entonces, el distribuidor tiene que activar el proceso de mayor prioridad (el primero de la cola) y pasar a preparado el activo (cuyo *pid* guarda la rutina de servicio al comenzar, lo mismo que hace la RIP, y se lo pasa como argumento al distribuidor).
- Que se haya ejecutado una rutina de servicio de interrupción correspondiente a otra causa (reloj, desbordamiento, etc.). También pueden darse, como antes, dos casos: que no se hayan modificado los estados de los procesos o que sí (como veremos enseguida, la rutina de servicio del reloj modifica las prioridades y la cola del procesador).

Por tanto, cuando se ejecuta el distribuidor la cola del procesador está actualizada, en el sentido de que cualquier proceso que haya pasado a «preparado» se ha metido en ella en el lugar determinado por su prioridad, y cualquiera que haya pasado a «bloqueado» se ha sacado de la cola. Así, todo lo que tiene que hacer el distribuidor es mirar si la cola está vacía y en tal caso pasar al proceso nulo

[16] En realidad no es la rutina de servicio directamente la que cambia el estado del proceso e inserta su registro en la cola, sino el SGF, llamado por el gestor del periférico, a su vez llamado por la rutina de servicio. Recuerde lo dicho antes sobre las operaciones a realizar por el SGF.

(EI, WAIT, DI), y si no es así:

1. Marcar «activo» (en el byte «estado» de su registro) al primer proceso de la cola.
2. Preparar en la UGM la nueva función de correspondencia.
3. Restaurar en los registros R0 a R14 de la UCP los valores que encuentra en el registro (de la tabla de procesos) de este proceso.
4. Retornar al proceso de usuario con RETI (recuerde que al empezar la ejecución de esta instrucción cambia la función de correspondencia de la UGM, y se recuperan los valores de RE y CP guardados en la pila de usuario).

El planificador

Para ordenar los procesos preparados en la cola del procesador, cada proceso tiene, en cada momento, una prioridad relativa, almacenada en el byte «PRIOR» (Figura 20.5). Esta prioridad es un número comprendido entre 0 y 19, donde 0 es la máxima prioridad y 19 la mínima. Las operaciones de «introducir en» o «actualizar» la cola del procesador a que nos hemos venido refiriendo en las descripciones anteriores se realizan de modo que la cola quede ordenada por la prioridad de los procesos. Pero ¿quién establece esas prioridades y con qué criterios? Ésta es la función del **planificador**. Los criterios y, acorde con ellos, el funcionamiento del planificador, dependen de los objetivos globales establecidos en el diseño del sistema operativo. La solución que vamos a explicar para Multialgorítmez está pensada para un sistema multiusuario interactivo.

Los elementos para el cálculo de la prioridad son los contenidos de los campos «BASE» y «TUCP» de cada registro de proceso. El planificador recalcula cada segundo la prioridad de cada uno de los procesos que están en la tabla, sumando, en cada fila de la tabla, los contenidos de «BASE» y «TUCP» y poniendo el resultado en el campo «PRIOR».

Como ya hemos dicho, el contenido de «BASE» es estático y está comprendido entre 0 (prioridad máxima) y 10 (prioridad mínima).

Veamos ahora la componente dinámica, el contenido de «TUCP» (tiempo de UCP). Cada cien milisegundos se produce una interrupción de reloj. La rutina de servicio de esa interrupción actualiza del valor de una variable del núcleo en la que se mantiene el número de segundos transcurridos desde el 1 de enero de 1970[17]. Y a continuación, y antes de RETI, pasa a ejecutar unas instrucciones que son las constituyentes del planificador.

Cada vez que se ejecuta la rutina (cada 100 ms) se incrementa en una unidad el valor de «TUCP» *del proceso activo*. Una de cada diez veces (es decir, cada segundo) todos los contenidos de «TUCP» de todos los procesos se dividen por dos (de esta manera, si un proceso está inactivo, conforme pasen los segundos se irá reduciendo su «TUCP» que es así una medida del tiempo de UCP consumido «recientemente») y se calculan los nuevos valores de «PRIOR» (sumando «BASE» y «TUCP»). Resumiendo, las operaciones son las indicadas en el seudocódigo del Programa 20.2. «shr» indica «división entera por 2», que en la máquina se realiza fácilmente con la instrucción SHR.

```
para el proceso activo:
        TUCP = TUCP+1;
K = K+1;
si K = 10 entonces
        K = 0;
        para todos los procesos:
        TUCP = shr(TUCP);
        PRIOR = BASE+TUCP
actualizar la cola del procesador;
BR DISTRIB
```

Programa 20.2. El planificador (parte de la rutina de servicio de la interrupción de reloj).

[17]Esa «variable global» será, en el nivel de máquina convencional, el contenido de una palabra de la MP que los otros componentes del núcleo pueden leer. Por ejemplo, el sistema de gestión de ficheros la lee para poner la hora en el inodo de un fichero modificado, Apartado 18.8.

Algunas observaciones adicionales sobre este algoritmo:

- En el arranque del sistema, a la variable K se le da el valor inicial «0», y cuando se crea un proceso se le pone (TUCP) = 0.
- El lector puede comprobar que el valor máximo que puede llegar a tomar «PRIOR» es 19 (mínima prioridad).
- Un proceso creado con FORK hereda la prioridad del padre.
- Si hay un «empate» de prioridad entre dos procesos puede resolverse de manera arbitraria: transcurrido un segundo, si se ha activado uno de ellos en detrimento del otro su prioridad se habrá reducido, y dejará de ser el primero.
- La prioridad de un procesos bloqueado por una petición de entrada/salida crece rápidamente, de manera que en cuanto termina la operación (por ejemplo, la entrada de un dato desde el terminal) pasa a activarse inmediatamente.

20.7. Resumen

En esta Lección nos hemos ocupado de la *ampliación del núcleo* estudiado en las dos anteriores *para que permita multiprogramación*. En el sistema resultante, al que hemos llamado «Multialgorítmez», pueden trabajar simultáneamente hasta cuatro usuarios, compartiendo la máquina y ejecutando varios procesos cada uno de ellos.

Hemos empezado con algunos elementos del modelo procesal que eran necesarios para comprender el modelo funcional. Elementos abstractos, como la necesidad del concepto de *proceso*, de *imagen de memoria* (espacio de direccionamiento lógico del proceso, incluyendo su pila), el estado *zombi* (estado de un proceso que ha terminado pero su padre no está esperando por él), y el diagrama de estados (Figura 20.1). Y elementos concretos, como los detalles, muy ligados al hardware de la UCP y la UGM, de los cambios de contexto en las interrupciones (paso de un proceso de usuario al núcleo) y en la vuelta del núcleo a un proceso de usuario.

En el modelo funcional, tras presentar las ideas de identificadores de usuario y de grupo, de autenticación y de permisos de acceso a ficheros, hemos presentado la interfaz del usuario (las órdenes) y la interfaz del programador (las llamadas al sistema). Ambas son ampliaciones de las estudiadas en Monoalgorítmez. Las llamadas más importantes son FORK, que crea un proceso nuevo, y EXEC, que sustituye la imagen de memoria del proceso por la de un nuevo programa.

Después hemos visto la necesidad de adaptar algunas de las estructuras de datos (las tablas de periféricos y de ficheros abiertos) y de introducir otras nuevas (las tablas de procesos, de demandas de entrada/salida y de posiciones de ficheros, así como un fichero de nombres de usuario y contraseñas).

Finalmente hemos comentado lo más importante de la implementación, incluyendo los componentes nuevos para la distribución y la planificación de los procesos.

20.8. Observaciones terminológicas

Como ya se ha indicado en el texto, hemos utilizado «en escena» y «entre bastidores» para traducir «foreground» y «background», respectivamente (en lugar de lo más inmediato, que sería «primer plano» y «fondo»).

El «registro del proceso» es lo que en muchos textos se llama «process control block» o «task structure», y la «imagen de memoria», «core image» (reminiscencia de cuando la MP se implementaba con «core», núcleos de ferrita).

Otros términos son:

cifrado, cifrar = encryption, encrypt.
distribución, distribuidor = dispatching, dispatcher (o «short-term scheduler», o «CPU scheduler»).
planificación, planificador = scheduling, scheduler (o «long-term scheduler», o «job scheduler»).

Esta última traducción no es muy acertada, pero es la habitual en el campo de los sistemas operativos.

20.9. Notas históricas

El primer prototipo de sistema de tiempo compartido (multiusuario) fue el CTSS («Compatible Timesharing System»), desarrollado en el M.I.T. (a partir de una primera versión probada en el Darmouth College, que permitía a varios usuarios ejecutar concurrentemente programas en BASIC, Corbató *et al.*, 1962). De él se derivó un proyecto más ambicioso, MULTICS («Multiplexed Information and Computing Service»), financiado por el M.I.T., Laboratorios Bell y General Electric, cuyo objetivo era proporcionar servicio a múltiples usuarios siguiendo el modelo de la distribución de energía eléctrica (Corbató y Vyssotsky, 1965): conectar terminales, distribuidos por una ciudad, a través de líneas telefónicas, a un ordenador de gran potencia. Se llegó a implementar (en un programa escrito en el lenguaje PL/1 con unas 300.000 sentencias) pero circunstancias empresariales y técnicas impidieron la consecución del objetivo final, aunque llegaron a instalarse más de cien unidades, y la última estuvo operativa hasta octubre de 2000 (Multicians, 2004). El diseño de MULTICS influyó de manera importante en sistemas posteriores, y, particularmente, en UNIX.

En el Apartado 21.13 continuaremos la historia con la evolución de UNIX.

20.10. Orientaciones bibliográficas

Si el lector desea continuar en la línea iniciada en esta Lección, estudiando con más detalles la implementación de un sistema operativo de tipo UNIX, el libro más indicado es el de Tanenbaum y Woodhull (1997). Su sistema, «MINIX», es mucho más cercano a UNIX que Multialgorítmez, se instala sobre un ordenador personal y está detalladamente documentado, con los listados de los programas fuente en C. Más información sobre MINIX puede encontrarse en Tanenbaum (2004).

Ahora bien, la decisión de que MINIX pueda ejecutarse sobre cualquier *pecé* (desde los basados en el microprocesador 8088) obligó a su autor a prescindir de algunas características importantes de los sistemas operativos (similarmente, en otra escala, a lo que ocurre con Multialgorítmez). En particular, no dispone de memoria virtual. El texto de Switzer (1993) es de un nivel más alto (y también requiere más conocimientos previos del lector). Su sistema, «TUNIX» (explicado en seudocódigo y con listados en C), incluye, además de memoria virtual, características más avanzadas, como hebras (Apartado 17.10). Sin embargo, tiene que ejecutarse sobre una máquina que ya disponga de un sistema UNIX. Otro sistema operativo didáctico, también con hebras y programado con orientación a objetos (en C++) es «Nachos», de Silberschatz *et al.* (1993).

Otra opción es Linux. Comenzó en 1991 como un proyecto de Linus Benedict Torvalds, a la sazón estudiante de la Universidad de Helsinki, tratando de «hacer un MINIX mejor que MINIX» (Torvals, 1992), y se ha transformado en la versión de UNIX para *pecés* (y también para ordenadores basados en los microprocesadores 680x0, Alpha, Sparc, Mips y PowerPC) más popular, con millones de usuarios (Alvestrand, 2004). En su desarrollo han acompañado a Torvalds cientos de programadores de todo el mundo, y las «distribuciones» disponibles contienen un gran número de programas de aplicación. Linux, así como la mayoría de las aplicaciones que lo acompañan, es *software libre* (novalis, 2004), lo que significa, entre otras cosas, que se distribuye con los programas fuente.

Hay que advertir, sin embargo, que Linux se ha convertido en un sistema realmente complejo, y el aprendizaje a partir de sus programas fuente sólo está al alcance de iniciados. Si el objetivo de este Apartado es orientar al lector para que, con los conocimientos básicos introducidos con Multialgorítmez, profundice algo más en los modelos de un sistema de tipo UNIX, la fuente más indicada es MINIX, con el mencionado libro de Tanenbaum y Woodhull (1997). No obstante, le recomendamos que, como usuario, instale Linux (u otro sistema operativo libre: en FreeOS, 2004, puede elegir entre cuarenta y ocho) en su *pecé* y se olvide de otros sistemas, más extendidos pero menos robustos y en absoluto libres.

20.11. Ejercicios

1. ¿Qué efecto tendría sobre la MP de Multialgorítmez el programa «ATILA» del Ejercicio 16 de la Lección 7? ¿Podría ejecutarse el propuesto como Ejercicio 14?

2. Suponga que decidimos mantener una pila única para Multialgorítmez, reservando la página física 7 para ella. Al crear un proceso para un programa que ocupa 1 KB, el sistema le asigna para el código la página física 88. ¿Qué valores deberá introducir en los registros de la UGM antes de activar a ese proceso?

3. Considere un programa que contiene:

```
DIR    RES.B  500
DFICH  RES.B  1
       ---
       CALL   SP1_1
       ----
SP1_1  READ(DFICH,#DIR,#500)
       LD     .0,#DIR
SIGUE  LD     .1,[.0++]
       PUSH   .1
       CMP    .0,#DFICH
       BN     SIGUE
       CALL   SP1_2
       ----
SP1_2  ----
       ; SP1_2 lee los datos de
       ; la pila y los envía
       ; a la pantalla
```

y este otro:

```
MENS   RES.B  80
       ---
       CALL   SP2
       ----
SP2    READ(#0,#MENS,#80)
       ----
```

Si en un momento dado el proceso correspondiente al primer programa está activo y ejecutando la primera de las instrucciones listadas y el del segundo está preparado también en la primera de las instrucciones, y no hay más procesos de usuario, y *si Multialgorítmez utilizase una pila única*, analice los estados sucesivos de esta pila.

4. Suponga que en el ejemplo del Apartado 6.8 (Programa 6.1) sustituimos
   - las seudoinstrucciones
     `NUM RES.B 1` y `BYTES RES.B 1`
     por:
     `PUNT DATA DHALT+1`
   - la instrucción
     `ST.B .0,/NUM`
     por estas dos:
     `LD .2,PUNT`
     `ST.B .0,[.2++]`
   - la instrucción
     `ST.B .1,BYTES`
     por:
     `ST.B .1,[.2++]`

   ¿dónde se almacenarían ahora los resultados? ¿En qué momento se asigna espacio de memoria para guardar estos resultados en el programa original y en el modificado?

5. Un programa de usuario empieza con esta instrucción:
   `ADD .14,#1000`
   y más adelante tiene:
   `CALL SUBPR`
   La imagen de memoria del proceso es la de la Figura 20.1. ¿Qué ocurre en su ejecución?

6. Imagine ahora que durante la ejecución del mismo programa, después de la instrucción `ADD` y antes de la `CALL`, aparece una interrupción. ¿Qué ocurre en el cambio de contexto?

7. En algunas UCP (por ejemplo, en los microprocesadores 680x0) los contenidos de CP y RE en una interrupción no se guardan en la pila del proceso, sino en la del núcleo. De proceder así, ¿cómo cambia la respuesta al ejercicio anterior?

8. Estudie las modificaciones necesarias en el hardware de Algorítmez para que CP y RE se guarden en la pila del núcleo. (La solución adoptada en los 680x0 implica duplicar el registro PP de manera transparente al programador: en modo usuario se utiliza un PP, y en modo supervisor otro).

9. Usted ha escrito un informe sobre virus y lo ha almacenado en un fichero llamado «`virus1.inf`». Es un fichero de texto, y, por tanto, no se va a ejecutar. Quiere que sólo puedan leerlo los usuarios de su grupo, pero que nadie, salvo usted, pueda modificarlo. ¿Qué orden daría al sistema?

10. También ha escrito un programa llamado «`antivirus`» que quiere hacer de dominio público: todo el mundo puede acceder a él, pero sólo para ejecutarlo y/o copiarlo; sólo usted puede borrarlo o modificarlo. ¿Qué orden daría?

11. Esboce en seudocódigo un programa que facilite al superusuario las tareas de dar de alta y de baja a los usuarios (escribiendo en el fichero `psswd`). ¿Qué permisos debe tener el fichero que contiene a este programa? ¿Cómo se puede evitar que un usuario normal escriba un programa igual y lo ejecute?

12. Debe existir un programa `passwd` que ejecute la orden que tiene ese nombre, y que sea ejecutable

por cualquier usuario. Escríbalo en seudocódigo. ¿Cómo pueden hacerse compatibles la ejecución de este programa con las precauciones de seguridad estudiadas en el ejercicio anterior? (Si no encuentra respuesta, lea el Ejercicio 8 de la Lección siguiente; si la encuentra, también).

13. Escriba programas en ensamblador para `getty` y para `login`.

14. Partiendo de la idea básica y esquemática del intérprete de órdenes para Monoalgorítmez dado en el Programa 18.8, escriba uno para Multialgorítmez que reconozca «&» y que acepte argumentos en la línea de órdenes. Por ejemplo, si «`un_programa`» es un programa que lee datos de un fichero y genera unos resultados que guarda en otro fichero y otros resultados que guarda en un tercer fichero, una orden podría ser:

```
un_programa f_ent f_sal1 f_sal2 &
```

15. Considere el siguiente fragmento de programa con llamadas al núcleo de Multialgorítmez:

```
PID RES.B  1
DLP RES.B  1
CAB DATA.B "Soy el padre",H'0D,H'0A
PIE DATA.B "Termino, hijos",H'0D,H'0A
NLP DATA.B "lp",0
NP1 DATA.B "PROGH1",0
NP2 DATA.B "PROGH2",0
     ---
     OPEN(#NLP,#2)
     ST.B    .13,DLP
     WRITE(DLP,#CAB,#18)
     FORK
     CMP.B   .13,#0
     BZ      HIJO1
     ST.B    .13,PID
     FORK
     CMP.B   .13,#0
     BZ      HIJO2
     ST.B    .13,PID
     WRITE(DLP,#PIE,#16)
     ---
HIJO1  EXEC(#NP1,#DLP,#1)
HIJO2  EXEC(#NP2,#DLP,#1)
     ---
```

«`PROGH1`» y «`PROGH2`» son nombres de ficheros en los que residen programas ejecutables. Entre otras cosas que no hacen al caso, el código fuente de `PROGH1` contiene lo siguiente:

```
DLP  RES.B 1
MENS DATA  "Soy el hijo 1",H'0D,H'0A
     ---
     POP   .0
     ST    .0,DLP
     ---
     WRITE(DLP,#MENS,#19)
     ---
```

`PROGH2` tiene una estructura similar (pero en `MENS` contiene `"Soy el hijo 2"`).

- ¿Sería necesario corregir el primer programa reservando dos bytes, `PID1` y `PID2`, en lugar de sólo uno? ¿Por qué? ¿Es necesaria la variable `PID`?
- ¿Sería necesario corregir los programas de los hijos para que no coincidiese la etiqueta `DLP` con la del padre? ¿Por qué?
- ¿Sería necesario que los hijos abriesen también la impresora? ¿Por qué?
- Escriba el resultado (o los posibles resultados) de la ejecución.

16. La separación de la creación de procesos (`FORK`) y la carga de programas (`EXEC`) sigue el modelo de UNIX, en el que ambas funciones están perfectamente diferenciadas, y se justifica porque facilita la implementación de ciertas características que no se han incluido en el modelo funcional de Multialgorítmez (como la «redirección», Apartado 21.2). Pero no es estrictamente necesaria. Otros sistemas operativos (por ejemplo, VMS) combinan las dos funciones en una sola llamada, que crea un nuevo proceso con la imagen en memoria correspondiente a un nuevo programa (y también hay sistemas, como Windows/NT, que permiten las dos formas). Si en Multialgorítmez siguiésemos el segundo modelo, muchos programas, como el correspondiente a «`init`», se simplificarían. Vamos a comprobarlo.

Suponga que, en lugar de `FORK` y `EXEC`, tenemos una llamada `FK_EX` con los mismos parámetros que `EXEC`: `FK_EX(#NPROG,#ARG)`. En el procesamiento de esta llamada, el sistema crea un nuevo proceso (hijo) y le hace corresponder una imagen en memoria de acuerdo con el programa cuyo

nombre está en `NPROG`, dejando preparados a los dos procesos (el padre y el hijo).

Codifique en ensamblador el programa `init` haciendo uso de esta nueva llamada.

17. Analice el siguiente programa, y escriba el resultado de su ejecución:

```
MODULE GENERACIONES
MAX EQU    H'34
DLP RES.B  1
GEN DATA.B H'31; ASCII de "1"
RET DATA.B H'0D,H'0A
M1  DATA.B "Soy de la generación "
M2  DATA.B ". Muero sin haber tenido hijos"
M3  DATA.B "Mis descendientes han muerto"
NLP DATA.B "A:lp",0
GN   OPEN(#NLP,#1)
     ST.B   .13,DLP
BUC  WRITE(DLP,#M1,#21)
     WRITE(DLP,#GEN,#3)
     LD.B   .0,/GEN
     ADD.B  .0,#1
     ST.B   .0,/GEN
     FORK
     CMP.B  .13,#0
     BNZ    ESP
     LD.B   .1,/GEN
     CMP.B  .1,#MAX
     BN     BUC
     WRITE(DLP,#M1,#21)
     WRITE(DLP,#GEN,#1)
     WRITE(DLP,#M2,#30)
     WRITE(DLP,#RET,#2)
    EXIT
ESP  WAIT(#0)
     WRITE(DLP,#M3,#28)
     WRITE(DLP,#RET,#2)
    EXIT
   END GN
```

18. Imagine tres procesos en Multialgorítmez, P1, P2 y P3, con prioridades de base 5, 7 y 9, respectivamente, y que durante un tiempo indefinidamente grande ninguno de ellos queda bloqueado, y la única interrupción que aparece es la del reloj. Inicialmente está activo P1. Calcule y ponga en el cuadro adjunto los valores sucesivos, segundo a segundo, de T1, T2 y T3 (valores de los campos «TUCP» de la Figura 20.5 para cada uno de los procesos) y PR1, PR2 y PR3 (valores respectivos de los campos «PRIOR»), señalando qué proceso pasa a estar activo al final de cada período de un segundo. Como consecuencia, calcule también el porcentaje de tiempo de la UCP que ocupa cada proceso si la situación se repite durante un período de tiempo muy grande. Suponga que en caso de «empate» (es decir, si coinciden los valores de PRIOR para dos procesos) el algoritmo da preferencia a P1 sobre P2 y a éste sobre P3.

seg.	T1	PR1	T2	PR2	T3	PR3	Act.
0	0	5	0	7	0	9	P1
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							

19. Compruebe que con el algoritmo de planificación explicado en el Apartado 20.6, los valores que puede tomar la prioridad de un proceso están limitados entre 0 y 19.

20. En esta Lección no nos hemos ocupado, entre otras cosas, de cómo puede el controlador de ADM acceder a la nueva memoria de 1MB. Hay dos soluciones: en una, al controlador de ADM se le dan direcciones físicas, y en otra se le dan direcciones lógicas. Estudie las modificaciones necesarias en el hardware (en el primer caso afectan esencialmente al controlador de ADM, y en el segundo, a la UGM).