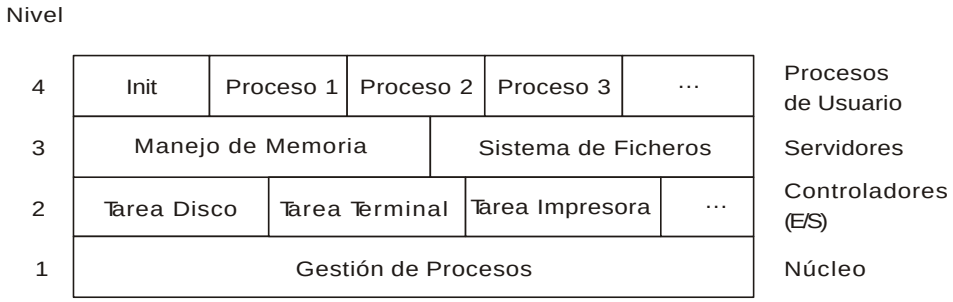
ÍNDICE

1.-Visión general de MINIX

El núcleo de MINIX tiene un diseño tipo cliente/servidor. Esto implica que es más modular que implementaciones más "tradicionales" de UNIX (como Linux), que son monolíticas. Por ello, servicios básicos como el sistema de ficheros o el gestor de memoria se tratan como procesos independientes del núcleo. Esto es una ventaja porque el núcleo queda más pequeño, y tiene un diseño más limpio. Además, los servicios se pueden recompilar para mejorarlos o corregir fallos, con independencia del núcleo. Sin embargo, si queremos añadir servicios (como los de red, que por defecto no vienen compilados), *sí* tenemos que recompilar el núcleo.

En el gráfico adjunto se explican los niveles en los que se divide el núcleo de MINIX.



NOTAS

1. Las dos primeras capas están compiladas y enlazadas en un solo ejecutable, llamado kernel.

2. Cada capa tiene menos privilegios que la anterior: el núcleo puede ejecutar cualquier instrucción y leer/escribir en cualquier parte de la memoria, las tareas pueden escribir en cualquier parte de la memoria asignada a procesos de niveles 3 y 4, para poder llevar a cabo las operaciones de E/S, etc.

Las funciones de las diferentes capas que componen MINIX son las siguientes:

1.1.- Funciones del núcleo (nivel 1)

El núcleo de MINIX, tal y como se dijo antes, tiene menos trabajo que los núcleos monolíticos arquetípicos de UNIX. Básicamente su trabajo es:

- Recoger las interrupciones físicas de los dispositivos y las llamadas al sistema de los procesos
- Gestionar la CPU

- Dar un modelo más abstracto a los procesos

La primera función es necesaria para abstraer los detalles de las interrupciones, tanto físicas como lógicas. Para ello, el núcleo se encarga de convertir estas interrupciones en mensajes, para homogeneizar la comunicación entre procesos. Es decir, que los procesos ven las interrupciones como simples mensajes.

La segunda es básica en cualquier sistema operativo multiusuario y multitarea, y principalmente está para aprovechar al máximo la capacidad de procesamiento de la máquina. Esta gestión debe repartir lo más equitativamente posible el tiempo de CPU entre los procesos y evitar que alguno de ellos se quede sin tiempo de procesamiento.

La última simplemente consiste en hacer el "trabajo sucio" del que alguien tiene que encargarse. Principalmente consiste en que guardar y restaurar los registros en los cambios de contexto, y resuelver el envío de mensajes, comprobando que los destinos son correctos, copiando los mensajes, etc.

1.2.- Funciones de la capa de controladores (nivel 2)

Esta capa comprende todas las llamadas "tareas", que son lo que otros sistemas operativos llaman manejadores o controladores de dispositivo (*drivers*). Por cada dispositivo a controlar, es decir, por cada manejador, hay una tarea ejecutándose en este nivel. Por ejemplo, generalmente habrá una tarea para la impresora, otra para la terminal, otra para el disco, etc. en la mayoría de los sistemas.

Además de las tareas "normales" de control de dispositivos, hay una adicional, llamada "tarea del sistema", que copia entre regiones de memoria. No es estrictamente una tarea de E/S, ya que trabaja exclusivamente con memoria, pero hace trabajos de copia entre regiones de memoria cuando otros procesos no pueden hacerlo, por ejemplo al mandar mensajes.

Aunque las tareas tiene mayores privilegios que los procesos de usuario (para poder acceder a los dispositivos físicos, poder copiar a zonas de memoria de otros procesos, etc.), y tienen mayor prioridad a la hora de ejecutarse, a efectos de comunicación son igual que los procesos, es decir, mandan y reciben mensajes. Esto hace el diseño del núcleo más limpio y homogéneo.

1.3.- Funciones de la capa de servicios (nivel 3)

La capa de servicios la componen principalmente los servicios del sistema de ficheros (FS) y el manejo de memoria (MM). Además, se pueden añadir servicios de red, que no vienen compilados por defecto en MINIX, y uno puede crearse sus propios servicios.

El gestor de memoria sirve las peticiones como `fork`, `exec`, o `brk`, mientras que el sistema de ficheros responde a las llamadas como `read`, `mount`, o `chdir`.

Estos servicios tienen más *prioridad* que los procesos de usuario, pero no pueden realizar la E/S directamente (tienen los mismos *privilegios*). Es decir, *tienen* que hacer peticiones a las tareas del nivel 2. Tampoco pueden acceder a zonas de memoria que no se les hayan asignado, por lo que tienen que pedir servicios a la tarea del sistema. Esto facilita la modificación de los servicios de este nivel por depender de llamadas al sistema del núcleo, que por una parte son de mayor nivel, y por otra parte hace que el nivel 3 no dependa de la configuración física del sistema informático en cuestión.

Los servicios, además, son ficheros ejecutables independientes del núcleo. La ventaja más evidente es la rapidez y comodidad para cambiar la implementación del nivel 3, por no vernos obligados a cambiar todo el núcleo.

1.4.- Funciones de la capa de nivel de usuario (nivel 4)

La capa superior comprende todos los procesos “normales” de nivel de usuario, es decir, los intérpretes de órdenes, como el `bash` o el `ash`; los compiladores, como el `gcc` o el `g++`; los servicios del sistema, como el `lpd` o el `syslogd`, etc. Como es lógico, es la capa que tiene menores privilegios y menor prioridad de ejecución de todas.

Ningún proceso de esta capa puede acceder a zonas de memoria de otros procesos, estén en la capa que estén. Si se tienen que comunicar, se mandan mensajes, y las copias en memoria las hace el núcleo mismo o el servicio del sistema.

Por otro lado, por tener los menores privilegios, si hay un solo proceso de otra capa intentando ejecutarse, obtendrá la CPU antes. Es decir, que no se ejecutará *ningún* procesos del nivel 4 mientras haya procesos de otras capas intentándolo.

Es importante resaltar que aunque se carguen servicios como el `lpd` que estén ejecutándose todo el tiempo que esté la máquina encendida, éstos *no* tendrán los mismos privilegios que los procesos de otras capas.

2.- Los procesos en Minix

Como hemos visto, muchos servicios que en otros sistemas operativos se tratan como simples llamadas al sistema atendidas por el núcleo, en Minix se tratan como procesos. Por ello, es muy importante tener unas primitivas que permitan que los procesos, sean del nivel que sean, puedan comunicarse de alguna manera. Esta necesidad se cubre con el empleo de mensajes, tema que será discutido en profundidad en un apartado posterior del presente documento. Sin embargo, todos estos procesos deben de ser planificados de forma que hagan un uso adecuado

de la CPU y se sigan unos criterios que aseguren un buen desempeño de las labores principales del sistema. Esta es la temática en la que se centrará el siguiente apartado.

Pero antes de empezar a ver funciones que implementen ya sea la planificación de los procesos como la comunicación entre los mismos deberíamos de fijar algunos conceptos previos referentes a los procesos en Minix. Así, veremos en el presente apartado cuál es la estructura que soporta a los procesos en Minix.

Instamos al lector a prestar especial atención a estos conceptos previos puesto que de su buen entendimiento dependerá en gran medida la facilidad de asimilación de todo lo que sigue referente a planificación y comunicación entre procesos.

2.1.- La estructura del proceso

Todo proceso en el sistema operativo Minix se representa por una estructura definida en el fichero de cabecera proc.h y que es la siguiente:

```
struct proc {
  struct stackframe_s p_reg;    /* registros del proceso guardados en marco pila */

#if (CHIP == INTEL)
  reg_t p_ldt_sel;              /* selector en gdt da base y límite ldt */
  struct segdesc_s p_ldt[2];    /* descriptores locales para códigos y datos */
                                /* 2 es LDT_SIZE. Evita incluir protect.h */
#endif /* (CHIP == INTEL) */

  reg_t *p_stguard;             /* palabra guardia de la pila */

  int p_nr;                     /* número de este proceso (para acceso rápido) */

  int p_int_blocked;            /* no cero si mensaje interrupción bloqueado por tarea ocupada */
  int p_int_held;               /* no cero si mensaje interrupción detenido por llamada ocupada */
  struct proc *p_nextheld;      /* siguiente en cadena de procesos interrupción detenidos */

  int p_flags;                          /* P_SLOT_FREE, SENDING, RECEIVING, etc. */
  struct mem_map p_map[NR_SEGS];/* mapa de memoria */
  pid_t p_pid;                          /* id de proceso pasado desde MM */

  clock_t user_time;            /* tiempo de usuario en ticks */
  clock_t sys_time;             /* tiempo de sistema en ticks */
  clock_t child_utime;                  /* tiempo de usuario acumulado de hijos */
  clock_t child_stime;                  /* tiempo de sistema acumulado de hijos */
  clock_t p_alarm;              /* tiempo de siguiente alarma en ticks, o 0 */

  struct proc *p_callerq;       /* cabeza de lista de procesos que quieren enviar */
  struct proc *p_sendlink;      /* vinculo a siguiente proceso que quiere enviar */
  message *p_messbuf;                   /* puntero a buffer de mensaje */
  int p_getfrom;                /* ¿de quién quiere recibir el proceso? */
  int p_sendto;

  struct proc *p_nextready;     /* puntero a siguiente proceso listo */
  sigset_t p_pending;                   /* mapa de bits para señales pendientes */
  unsigned p_pendcount;                 /* cuenta de señales pendientes e inconclusas */

  char p_name[16];              /* nombre del proceso */
};
```

De entre todos estos campos que definen a un proceso nos centraremos en algunos de ellos que presentan especial interés en cuanto a la comunicación y planificación se refiere, campos que pasamos a comentar a continuación:

- **`p_int_blocked`.** Este campo indica si el proceso tiene una interrupción física esperándole. Si su valor es distinto de cero, el proceso está bloqueado.
- **`p_int_held`.** Indica si el proceso está bloqueado por alguna razón que no le deje ejecutarse. De nuevo, si vale distinto de cero, el proceso está bloqueado.
- **`p_nextheld`.** Este campo apunta al siguiente proceso en la lista de procesos bloqueados.
- **`p_messbuf`.** Puntero a un "almacén" en memoria donde se guardan los mensajes recibidos.
- **`p_callerq`.** Apunta al primer proceso que está bloqueado intentando enviar un mensaje al proceso actual.
- **`p_sendlink`.** Este campo es un puntero al siguiente proceso que está bloqueado intentando mandar un mensaje al mismo proceso que el actual.
- **`p_getfrom`.** Indica al proceso del que está intentando recibir el proceso actual. Sólo tiene sentido si el proceso está efectivamente bloqueado esperando un mensaje.
- **`p_nextready`.** Apunta al siguiente proceso que esté preparado para ejecutarse.
- **`p_flags`.** Es un campo de indicadores. Si el valor es cero, es que el proceso puede ejecutarse. Si no, hay algún problema. Este campo se interpreta como una máscara de bits, que puede tener los siguientes valores (más de uno, en general):

  - *`P_SLOT_FREE`:* La entrada no se está utilizando.
  - *`NO_MAP`:* El proceso todavía no tiene mapa de memoria. Esto sólo ocurre cuando está a medio crear, justo después de una llamada incompleta a `fork`.
  - *`SENDING`:* El proceso no puede ejecutarse por estar bloqueado intentando enviar un mensaje.
  - *`RECEIVING`:* El proceso no puede ejecutarse por estar esperando a recibir un mensaje.

Según todo lo anterior, debemos tener siempre presente que cualquier tarea que implique ejecución se representará en Minix por un proceso (interrupciones, llamadas al sistema, procesos de usuario, tareas del sistema), o al menos uno que lo represente o incluya; y que cada proceso tendrá asociado una estructura con todos los campos anteriores que servirán para desempeñar las labores de planificación y comunicación de tales procesos.

2.2.- La tabla de procesos

Una vez conocida la representación de un proceso en Minix el siguiente paso es saber donde se encuentran o como se encontrarán organizados tales procesos en el sistema. La respuesta a esto la encontramos también en el fichero proc.h, en concreto en la variable externa proc:

```
EXTERN struct proc proc[NR_TASKS + NR_PROCS];          /* tabla de procesos */
```

Este vector de estructuras de procesos, o vector de procesos definidos según lo dicho en la sección anterior constituye lo que se conoce comunmente como la **tabla de procesos**. Esta tabla recoge y agrupa todos los procesos existentes en el sistema operativo en un momento dado, de forma que existirá una entrada para cada proceso en esta tabla, entrada que será del tipo anteriormente indicado para cada proceso. Por otra parte, se utiliza una variable externa

relacionada con la tabla de procesos para referenciar a las posiciones de ésta de una forma rápida sin tener que calcular la dirección para una determinada entrada, y se declara también en proc.h. Dicha variable externa es:

```
EXTERN struct proc* pproc_addr[NR_TASKS + NR_PROCS];
```

Como podemos ver en la declaración de la tabla de procesos, se usan dos macros para definir su tamaño:

NR_TASKS : Número de tareas
NR_PROCS : Número de procesos (servidores y usuarios)

Además, un rápido vistazo al fichero proc.h nos permitirá descubrir dos grupos de macros definidas para servir de apoyo a las funciones que manejan la tabla de procesos. Dichas macros son las siguientes:

```
/* Direcciones mágicas de tabla de procesos */
#define BEG_PROC_ADDR (&proc[0] )
#define END_PROC_ADDR (&proc[ NR_TASKS + NR_PROCS ] )
#define END_TASK_ADDR (&proc[ NR_TASKS ] )
#define BEG_SERV_ADDR (&proc[ NR_TASKS ] )
#define BEG_USER_ADDR (&proc[ NR_TASKS + LOW_USER ] )

#define NIL_PROC        ((struct proc *) 0 )
```

El primer grupo lo constituyen los punteros más comunmente conocidos como **Direcciones Mágicas**. Estas direcciones apuntan a zonas de la tabla de procesos y sirven para establecer una distribución de la misma en función del tipo de procesos existentes en el sistema. En la Figura 1 podemos ver esta distribución de la tabla de procesos en 3 zonas:

- Tareas
- Servidores
- Usuarios

Estas zonas veremos que cobran gran importancia a la hora de planificar los procesos y permiten la rápida y fácil identificación del nivel al que pertenece cada tarea (Tareas = Nivel 2, Servidores = Nivel 3 y Usuarios = Nivel 4) y con ello los privilegios y prioridades asociados a cada uno.

También hay que mencionar que esta tabla de procesos contiene una serie de colas de punteros incrustados, de las que iremos hablando en detalle a medida que vayamos avanzando en la planificación y comunicación de procesos en Minix, y cuyos punteros se encuentran contenidos en la estructura de proceso que representa a cada proceso que haya en el sistema. No desespere, a medida que vayamos viendo cada una de estas colas iremos haciendo referencia a

los campos de la estructura de procesos relacionados con las mismas. De momento nos contentaremos con saber de su existencia y dejaremos los detalles para más adelante.

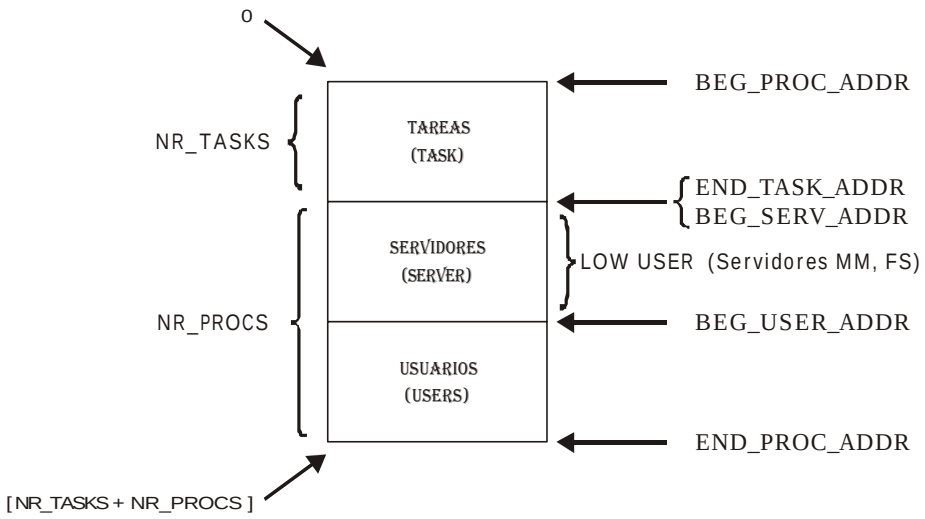


Figura 1

Por último, tenga en cuenta que el tiempo de permanencia de un proceso en esta tabla de procesos se extiende a toda la vida de dicho proceso en el sistema, insertándose cuando comienza su ejecución (cuando se lanza) y extrayéndose cuando concluye definitivamente (llega al final de su ejecución).

3.- Planificación de procesos en Minix

Minix emplea un algoritmo de planificación multinivel que sigue de cerca la estructura que se muestra en la Figura 2. En esa figura vemos tareas de E/S en la capa 2, procesos de servidor en la capa 3 y procesos de usuario en la capa 4.

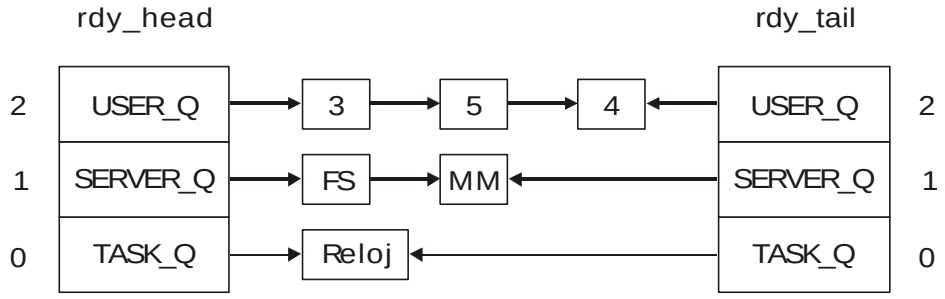


Figura 2

El planificador mantiene tres colas de procesos ejecutables, una para cada capa, como se ve en la Figura 2. Sin embargo, estas colas son “lógicas” ya que no existe ninguna estructura claramente definida para cada cola. En lugar de esto, las colas se encuentran incrustadas en la tabla de procesos por medio de punteros empotrados en cada una de las entradas a dicha tabla.

De esta forma, la Figura 2 tiene una interpretación “física” en la Figura 3. En esta se puede ver lo que podría ser cada una de las colas según aparecen realmente implementadas en el sistema.

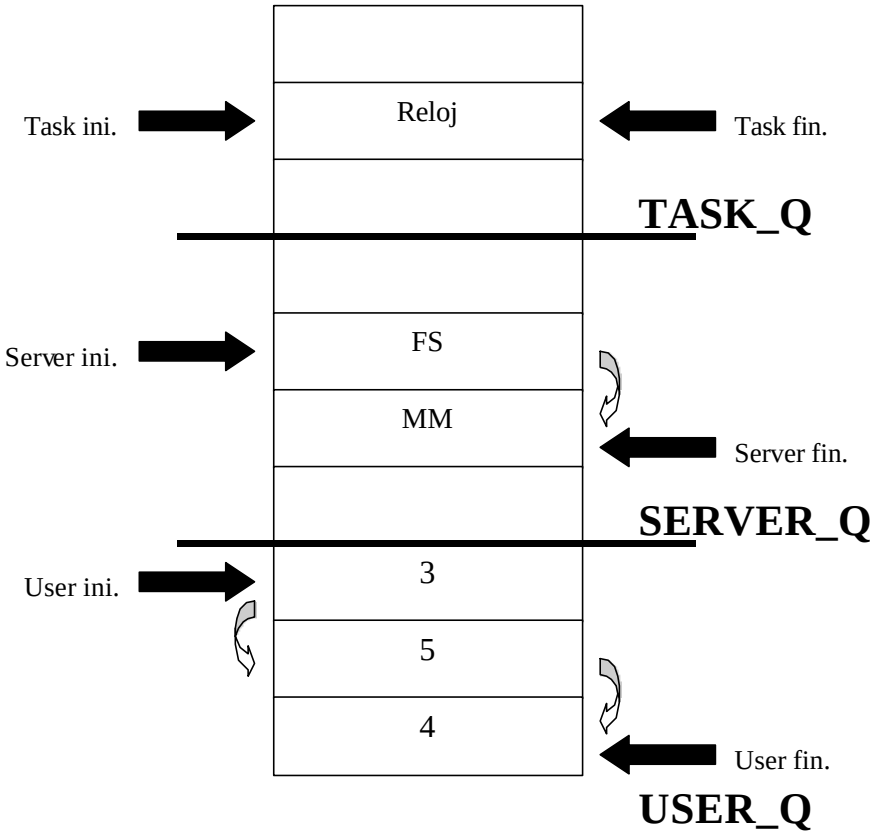


Figura 3

Si recordamos la estructura de un proceso, según lo definido en proc.h, existen una serie de punteros incluidos en cada proceso de Minix. De todos ellos es **p_nextready** el que se encarga de apuntar al siguiente proceso en la cola de ejecución actual. O sea que los enlaces que podemos ver en la Figura 3 cobran forma en este campo del proceso.

En la Figura 3 también vemos que existen unos enlaces, más gruesos, que apuntan al inicio y fin de cada una de las colas de procesos ejecutables. Estos enlaces se implementan por medio de un par de arrays que se declaran en proc.h :

*EXTERN struct proc *rdy_head [NQ];*
*EXTERN struct proc *rdy_tail[NQ];*

El array **rdy_head** tiene una entrada por cada cola, y cada entrada apunta al proceso que está a la cabeza de la cola correspondiente. De forma similar, **rdy_tail** es un array cuyas entradas apuntan al último proceso de cada cola. De esta forma, se agrupan estos punteros de los que hablábamos en la Figura 3 (los gruesos) en dos arrays de punteros: los que apuntan a los inicios de las colas y los que apuntan al final de las colas.

Para dar una explicación para mantener ambos arrays de punteros, en lugar de mantener uno solo, hay que hablar del desbloqueo de los procesos y del algoritmo de planificación de procesos del Minix. Así, cada vez que se despierta un proceso bloqueado se le anexa al final de la cola. La existencia del array **rdy_tail** hace eficiente esta acción de agregar un proceso al final de una cola. Por su parte, para buscar el siguiente proceso a ejecutar se busca por el inicio de las colas, siguiendo un criterio de prioridad del que hablaremos en breve, por lo que la existencia del array rdy_head posibilita un acceso eficiente al inicio de las colas de procesos ejecutables.

En cuanto a las tres colas hay que decir que en ellas sólo se ponen los procesos ejecutables. Así, cuando un proceso en ejecución se bloquea, o un proceso ejecutable es terminado por una señal, ese proceso se quita de la correspondiente cola del planificador. Como veremos a continuación esta labor la lleva a cabo la función **unready**, que forma parte del fichero proc.c.

Dadas las estructuras de cola que hemos descrito, el **algoritmo de planificación** se puede resumir en encontrar la cola con más alta prioridad que no está vacía y escoger el proceso que está a la cabeza de esa cola. Si todas las colas están vacías, se ejecuta la **rutina de "marcha en vacío" o IDLE**. En la Figura 2, TASK_Q tiene la más alta prioridad. El código de planificacion está en proc.c, en concreto se refleja en la función **pick_proc** , cuya labor podríamos resumir en escoger el siguiente proceso a ejecutar (como su nombre, en inglés, bien indica). Esto lo lleva a cabo haciendo que el puntero **proc_ptr**, declarado en proc.h, apunte al proceso, en la tabla de procesos, que se está ejecutando actualemente o que se va a comenzar a ejecutar. Cualquier cambio a las colas que pudiera afectar la selección del siguiente proceso por ejecutar requiere una nueva invocación de pick_proc. Siempre que el proceso en curso se bloquea, se invoca a pick_proc para replanificar la CPU.

Hasta ahora hemos hablado de cómo se escoge el siguiente proceso a ejecutar y cómo se eliminan procesos de las colas de procesos ejecutables pero , sorprendentemente, aún no hemos mencionado cómo se insertan procesos a dichas colas. Dicha labor recae en la función **ready**, contrapartida de unready, que también se implementa en el fichero proc.c. Como resumen se puede decir que ready desempeña la labor de ingresar procesos que pueden ser ejecutados en las colas de procesos ejecutables.

Poco a poco vamos conociendo la forma en la que Minix planifica sus procesos pero de lo que no hemos mencionado nada es de *cuándo se lleva a cabo dicha planificación*. Por norma general existen dos momentos principales en los que entra en juego la planificación de procesos:

- En el bloqueo y desbloqueo de procesos.
- En el fin de la rodaja de tiempo, o cuanto, de ejecución.

El primero de estos momentos tiene que ver con las funciones ready y unready, a las que ya se ha hecho mención y que pasaremos a detallar en profundidad en la siguiente sección. El segundo de estos momentos tiene que ver con la función **sched** (*schedule, horario o planificación*) que como su propio nombre, en inglés, indica se encarga del cumplimiento de los plazos de tiempo concedidos a cada proceso para su ejecución. Como las anteriores, sched se implementa en el fichero proc.c y , aunque describiremos en profundidad dicha función así como todo lo relacionado con la política de tiempo compartido usada por Minix, adelantaremos

que esta labor sólo se llevará a cabo sobre los procesos de usuario, siguiendose una estrategia diferente para las tareas de E/S y los servidores/servicios (MM y FS).

Hasta el momento hemos visto por encima algunos de los procesos más importantes que se llevan a cabo en la planificación de procesos de Minix, así como algunas estructuras y conceptos clave que tienen que ver con ella. En la siguiente sección vamos a tratar en profundidad todos los aspectos relacionados con la planificacion de procesos, llegando a la implementación de los mismos.

3.1.- La implementación

Como bien anunciábamos ya al principio de este apartado, la implementación de la planificación de procesos en Minix tiene lugar en los ficheros proc.c y proc.h. Las funciones implementadas en proc.c que tienen relación con la planificación de procesos se pueden ver en la Figura 4. En ellas vemos que se puede hacer una primera distinción entre las que tienen en cuenta la concurrencia (sirven de punto de conexión con el exterior del nucleo junto a interrupt y sys_call, que veremos en comunicación de procesos) y las que no. Entre las que sí, encontramos un puñado de funciones auxiliares que, a excepción de unhold, siguen el formato **lock_*funcionSin***. Dichas funciones no son más que rutinas de apoyo a la planificación que establecen un semáforo, usando la variable switching antes de invocar a la *funcionSin* correspondiente, y liberan el semáforo al completar su trabajo. En cuanto a la función unhold ya hablaremos en detalle de ella al final de esta sección pero a modo de anticipo diremos que se encarga de procesar las interrupciones detenidas, para comunicarlas a los procesos pendientes.

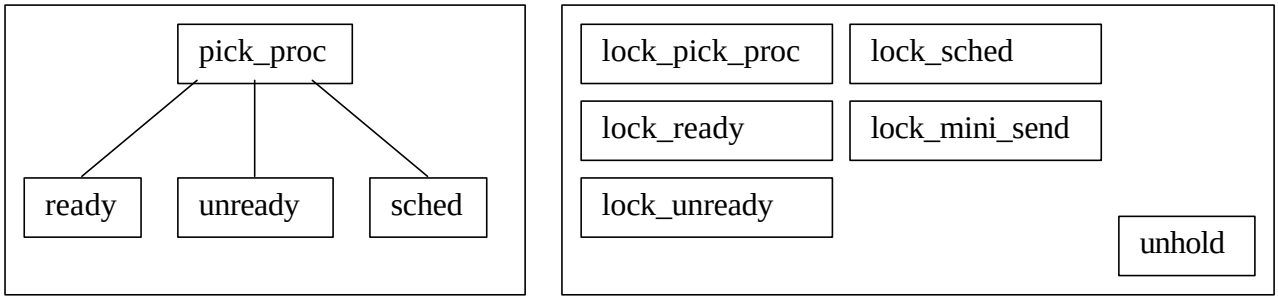


Figura 4

A lo largo de la presente sección vamos a descubrir todas estas funciones en profundidad a excepción de las lock_funcionSin de las que sólo se hará una mención general, al tiempo que mostramos sus códigos fuente, por ser su funcionamiento muy relacionado con cada *funcionSin* correspondiente.

3.1.1.- La función pick_proc( )

Como ya hemos mencionado anteriormente, la función pick_proc es la encargada de escoger el proceso que se va a ejecutar a continuación. Por ello, **esta función implementa el algoritmo de planificación de procesos**, en el cual se recoge el criterio de prioridad de unos procesos sobre otros a la hora de entrar en ejecución. Así, se da mayor prioridad a las tareas de E/S , en segundo lugar a los procesos de servicios/servidores y en último lugar a los procesos de usuario, escogiendo entre estos según su posición dentro de la cola de procesos de usuario ejecutables.

El funcionamiento de esta función se puede ver descrito en la Figura 5, que representa en forma de diagrama de flujo el código que implementa a dicha función y que se muestra a continuación.

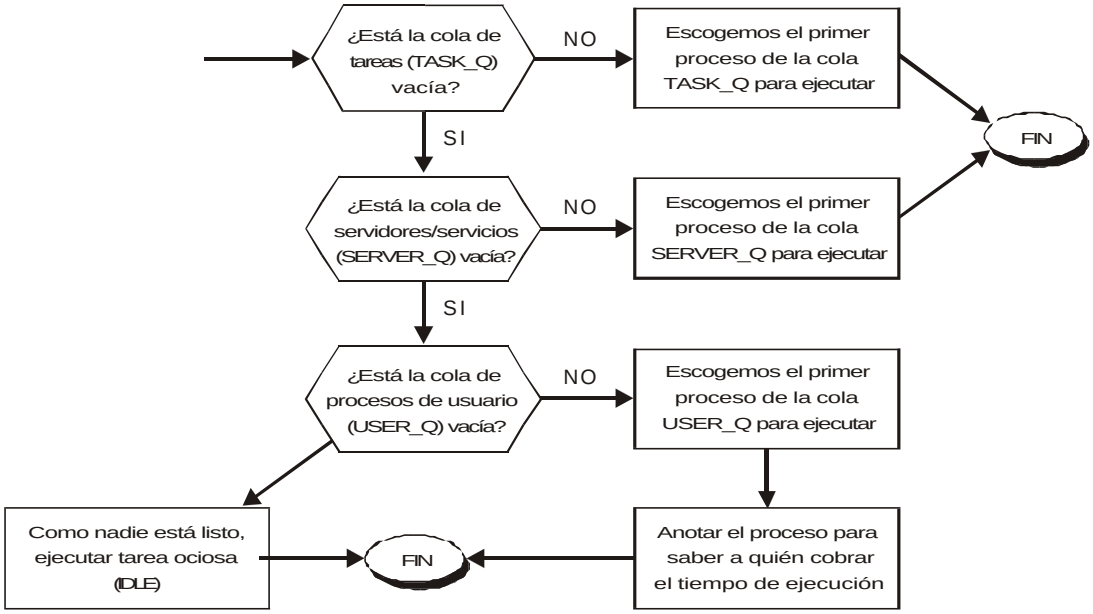


Figura 5

```
07176   /*===========================================================================*
07177    *                                    pick_proc                              *
07178    *===========================================================================*/
07179   PRIVATE void pick_proc()
07180   {
07181   /* Decidir a quién ejecutar ahora. Se escoge un nuevo proceso fijando 'proc_ptr'.
07182    * Si se escoge un proceso de usuario nuevo (u ocioso), asentarlo en 'bill_ptr'
07183    * para que la tarea de reloj sepa a quién cobrar tiempo de sistema.
07184    */
07185
07186     register struct proc *rp;         /* proceso por ejecutar */
07187
07188     if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
07189                 proc_ptr = rp;
07190                 return;
07191     }
07192     if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
07193                 proc_ptr = rp;
07194                 return;
07195     }
07196     if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
07197                 proc_ptr = rp;
```

```
07198                   bill_ptr = rp;
07199                   return;
07200           }
07201           /* Nadie está listo. Ejecutar tarea ociosa. Ésta podría hacerse tarea
07202            * de usuario siempre lista para evitar este caso especial.
07203            */
07204           bill_ptr = proc_ptr = proc_addr(IDLE);
07205   }
```

A partir de la Figura 5 y de la relación entre cada uno de los nodos de éste y el código podemos ver cual es la forma en la que se implementa esta función.

3.1.2.- La función ready( )

Se invoca para ingresar un proceso que puede ser ejecutado en su cola de procesos ejecutables correspondiente (Tareas, Servidores o Usuarios). Se llama desde **mini_send** y **mini_rec**, funciones implementadas en proc.c pero que como veremos más adelante tienen relación directa con la comunicación de procesos, pero también se podría haber llamado desde interrupt, sólo que en ésta se insertó el código de ready dentro de la propia función para agilizar el proceso (por optimización del código) ya que es ésta una función muy recurrida e interesa que sea todo lo rápida posible.

Lo primero que se hace es almacenar el proceso a ejecutar en registros del procesador para optimizar y agilizar los cálculos. Ready hace uso de dos macros definidas en proc.h:

- istaskp (p)

  # define istaskp(p) ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))
- isuserp (p)

  #define isuserp(p) ((p) >= BEG_USER_ADDR)

Podríamos resumir su funcionalidad básicamente a determinar si un proceso dado, p, es una tarea o es un proceso de usuario, y que no es el proceso IDLE. Dicho de otra forma estas macros nos permiten determinar el nivel al que pertenece el proceso : si es del nivel 2 (tarea) o si es del nivel 4 (proceso de usuario) , o si es del nivel 3 (servidores) por pura exclusión, y con ello determinar su prioridad y cola de procesos asociada.

La forma en la que se lleva esto a cabo es por medio de la comprobación de la ubicación del proceso dentro de la tabla de procesos. Así el proceso pertenecería al tipo de procesos de la zona de la tabla en la que se halla ubicado. Para determinar estas zonas se hace uso de las *Direcciones mágicas*, de las que ya se habló cuando vimos la tabla de procesos.

En la Figura 6 se puede ver el funcionamiento de esta función, que representa en forma de diagrama de flujo el código que implementa a dicha función y que se muestra a continuación.

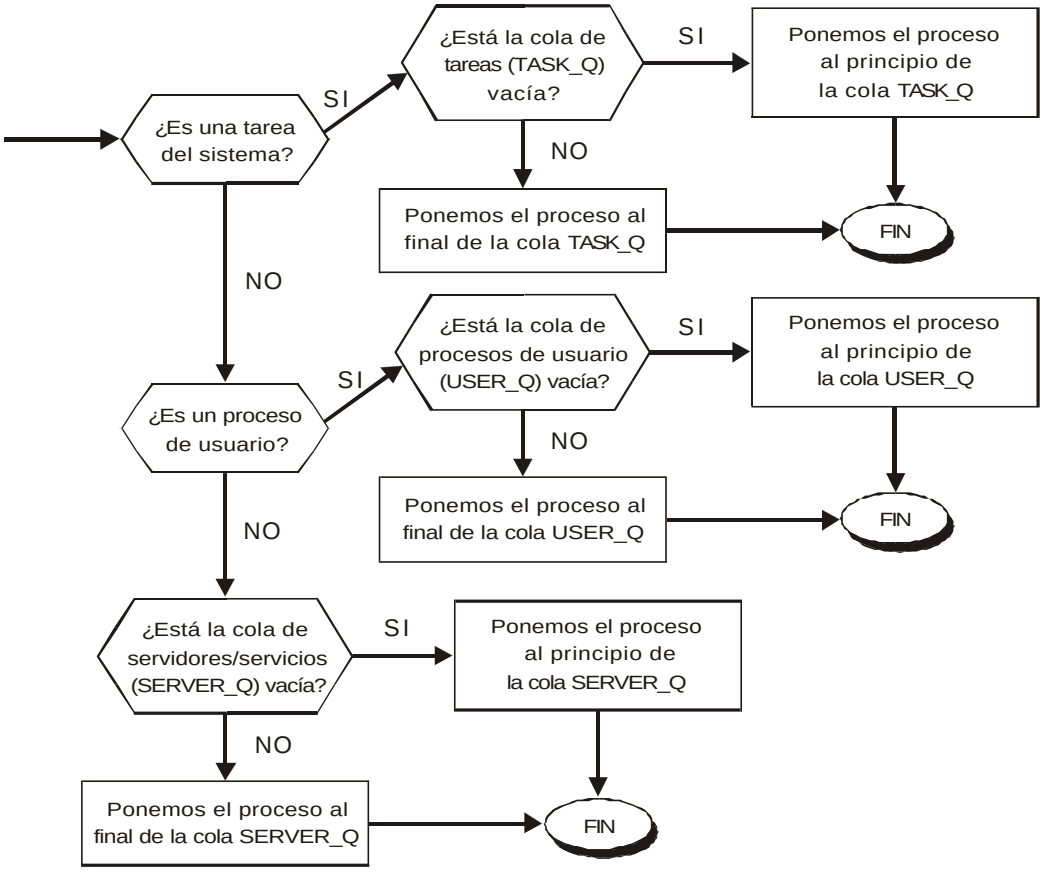


Figura 6

```
07207   /*===========================================================================*
07208    *                                    ready                                  *
07209    *===========================================================================*/
07210   PRIVATE void ready(rp)
07211   register struct proc *rp;          /* este proceso ya es ejecutable */
07212   {
07213   /* Añadir 'rp' al final de una de las colas de procesos ejecutables.
07214    * Se mantienen tres colas:
07215    *   TASK_Q   - (más alta prioridad) para tareas ejecutables
07216    *   SERVER_Q – (prioridad media) sólo para MM y FS
07217    *   USER_Q   - (más baja prioridad) para procesos de usuario
07218    */
07219
07220     if (istaskp(rp)) {
07221             if (rdy_head[TASK_Q] != NIL_PROC)
07222                     /* Agregar al final de cola no vacía. */
07223                     rdy_tail[TASK_Q]->p_nextready = rp;
07224             else {
07225                     proc_ptr =                  /* ejecutar tarea nueva ahora */
07226                     rdy_head[TASK_Q] = rp;      /* agregar a cola vacía */
07227             }
07228             rdy_tail[TASK_Q] = rp;
07229             rp->p_nextready = NIL_PROC;         /* nueva entrada no tiene sucesor */
07230             return;
07231     }
07232     if (!isuserp(rp)) {                 /* las demás son similares */
07233             if (rdy_head[SERVER_Q] != NIL_PROC)
07234                     rdy_tail[SERVER_Q]->p_nextready = rp;
07235             else
07236                     rdy_head[SERVER_Q] = rp;
07237             rdy_tail[SERVER_Q] = rp;
07238             rp->p_nextready = NIL_PROC;
07239             return;
```

```
07240       }
07241       if (rdy_head[USER_Q] == NIL_PROC)
07242               rdy_tail[USER_Q] = rp;
07243       rp->p_nextready = rdy_head[USER_Q];
07244       rdy_head[USER_Q] = rp;
07245     /*
07246       if (rdy_head[USER_Q] != NIL_PROC)
07247               rdy_tail[USER_Q]->p_nextready = rp;
07248       else
07249               rdy_head[USER_Q] = rp;
07250       rdy_tail[USER_Q] = rp;
07251       rp->p_nextready = NIL_PROC;
07252     */
07253     }
```

A partir de la Figura 6 y de la relación entre cada uno de los nodos de ésta y el código podemos ver cual es la forma en la que se implementa esta función.

3.1.3.- La función unready( )

Se invoca para quitar de su cola de ejecución a un proceso. Esto ocurre cuando el proceso se bloquea. En Minix, las colas de procesos ejecutables (*Tareas* o TASK_Q, *Servidores* SERVER_Q y *Usuarios* o USER_Q) sólo contienen a los procesos en ejecución por lo que habrá que extraer a un proceso de dichas colas cuando se interrumpe dicha ejecución por un bloqueo. Además, normalmente el proceso que quita está a la cabeza de su cola, ya que un proceso necesita estar ejecutándose para poder bloquearse. En tal caso, unready invoca a pick_proc antes de retornar, para que escoja el siguiente proceso a ejecutar en sustitución del actual que va a dejar de ejecutarse. Sin embargo, esto no es así para todos los casos ya que los procesos de usuario pueden ser desalojados de la cola de ejecución aún cuando no están en ejecución (cuando no están a la cabeza de su cola y por tanto en ejecución). Esto ocurre cuando se le envía una señal.

De la misma forma que para las funciones anteriores, en la Figura 7 se puede ver de una forma fácil y rápida el funcionamiento de esta función, que representa en forma de diagrama de flujo el código que implementa a dicha función y que se muestra a continuación.

```
07255   /*===========================================================================*
07256    *                                unready                                    *
07257    *===========================================================================*/
07258   PRIVATE void unready(rp)
07259   register struct proc *rp;           /* este proceso ya no es ejecutable */
07260   {
07261   /* Un proceso se bloqueó. */
07262
07263     register struct proc *xp;
07264     register struct proc **qtail;  /* TASK_Q, SERVER_Q, o USER_Q rdy_tail */
07265
07266     if (istaskp(rp)) {
07267             /* ¿pila de tareas aún OK? */
07268             if (*rp->p_stguard != STACK_GUARD)
07269                     panic("stack overrun by task", proc_number(rp));
07270
07271             if ( (xp = rdy_head[TASK_Q]) == NIL_PROC) return;
07272             if (xp == rp) {
07273                     /* Quitar cabeza de la cola */
07274                     rdy_head[TASK_Q] = xp->p_nextready;
07275                     if (rp == proc_ptr) pick_proc();
```

```
07276                   return;
07277           }
07278           qtail = &rdy_tail[TASK_Q];
07279   }
07280   else if (!isuserp(rp)) {
07281           if ( (xp = rdy_head[SERVER_Q]) == NIL_PROC) return;
07282           if (xp == rp) {
07283                   rdy_head[SERVER_Q] = xp->p_nextready;
07284                   pick_proc();
07285                   return;
07286           }
07287           qtail = &rdy_tail[SERVER_Q];
07288   } else
07289   {
07290           if ( (xp = rdy_head[USER_Q]) == NIL_PROC) return;
07291           if (xp == rp) {
07292                   rdy_head[USER_Q] = xp->p_nextready;
07293                   pick_proc();
07294                   return;
07295           }
07296           qtail = &rdy_tail[USER_Q];
07297   }
07298
07299   /* Enviar cuerpo de la cola. Puede hacerse que un proceso no esté listo
07300    * aunque no se esté ejecutando enviándole una señal que lo termine.
07301    */
07302   while (xp->p_nextready != rp)
07303           if ( (xp = xp->p_nextready) == NIL_PROC) return;
07304   xp->p_nextready = xp->p_nextready->p_nextready;
07305   if (*qtail == rp) *qtail = xp;
07306 }
```

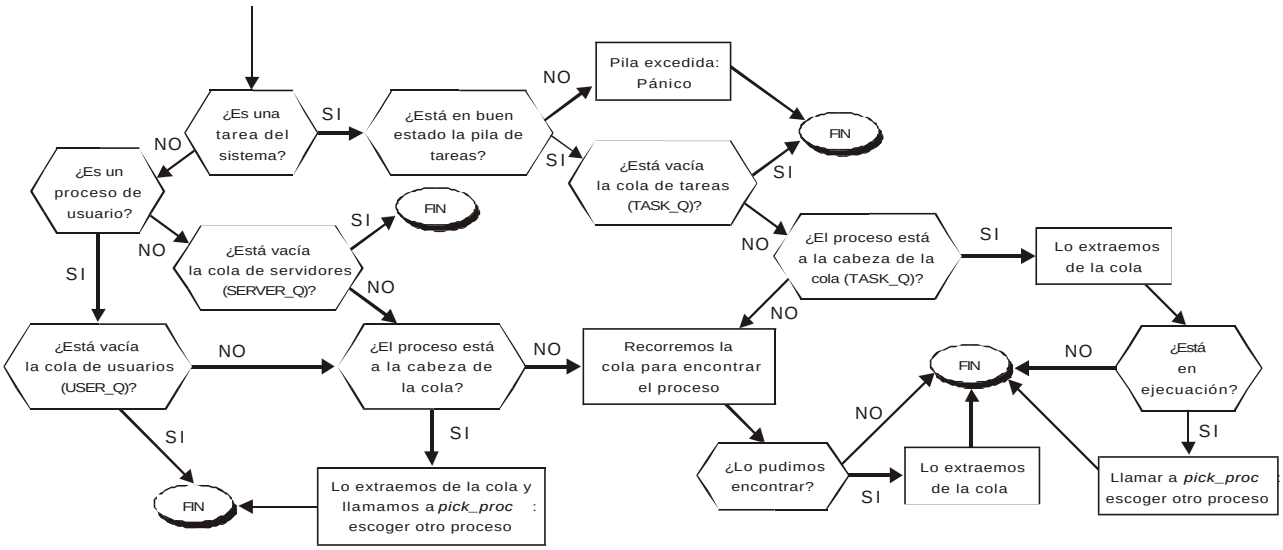


Figura 7

A partir de la Figura 7 y de la relación entre cada uno de los nodos de éste y el código podemos ver cuál es la forma en la que se implementa esta función, que junto con la anterior (ready) constituyen la forma de manejar (insertar, extraer) las colas de procesos ejecutables (TASK_Q, SERVER_Q y USER_Q).

3.1.4.- La función sched( )

Aunque la mayor parte de las decisiones de planificación se toman cuando un proceso se bloquea o desbloquea, también debe llevarse a cabo planificación de procesos cuando la tarea del reloj se da cuenta de que el proceso de usuario actual excedió su cuanto de tiempo. En este caso la tarea del reloj invoca a la función **shed** (*scheduling, horario o planificación*) para pasar el proceso de usuario que está a la cabez de la cola USER_Q al final de la misma. Nótese que concretamos la cola, USER_Q, que se va a modificar sin previa comprobación. El motivo de tal actuación es que **Minix sólo planifica según un esquema de Round Robin Simple a los procesos de usuario**, teniendo el resto de los procesos de otros niveles una total disponibilidad, al menos en cuanto a tiempo se refiere, de la CPU para su ejecución. O sea, que el Sistema de Ficheros (FS), el Administrador de Memoria (MM) y las tareas de E/S nunca se colocan al final de sus colas respectivas por haberse estado ejecutando durante demasiado tiempo; se confía en que funcionarán correctamente y se bloquearán después de haber terminado su trabajo.

Una vez más, en la Figura 8 se puede ver de una forma fácil y rápida el funcionamiento de esta simple función, que representa en forma de diagrama de flujo el código que implementa a dicha función y que se muestra a continuación.

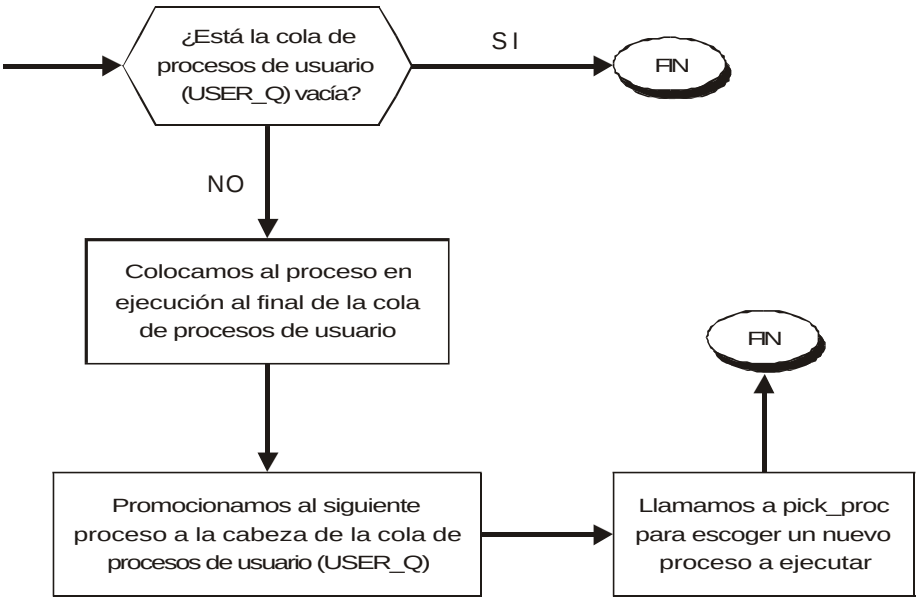


Figura 8

```
07308    /*===========================================================================*
07309     *                                sched                                      *
07310     *===========================================================================*/
07311    PRIVATE void sched()
07312    {
07313    /* El proceso actual se ejecutó demasiado tiempo. Si otro proceso de baja prioridad
07314     * (usuario) es ejecutable, colocar el actual al final de la cola de usuario,
07315     * tal vez promoviendo otro usuario a la cabeza de la cola.
07316     */
07317
07318      if (rdy_head[USER_Q] == NIL_PROC) return;
07319
07320      /* Uno o más procesos de usuario en cola. */
```

```
07321       rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
07322       rdy_tail[USER_Q] = rdy_head[USER_Q];
07323       rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
07324       rdy_tail[USER_Q]->p_nextready = NIL_PROC;
07325       pick_proc();
07326     }
```

A partir de la Figura 8 y de la relación entre cada uno de los nodos de éste y el código podemos ver cula es la forma en la que se implementa esta función.

3.1.5.- Las funciones lock_*funcionSin*( )

Aquí nos centramos en el conjunto de funciones auxiliares **lock_mini_send**, **lock_pick_proc**, **lock_ready**, **lock_unready** y **lock_sched**. Su funcionamiento es simple: se basan en una llamada a la *funcionSin* correspondiente, pero usando el "semáforo" *switching* de forma que se mantenga un control sobre la ejecución concurrente de las mismas.

En la Figura 9 se puede ver un esbozo general de lo que se lleva a cabo en cada una de estas funciones.

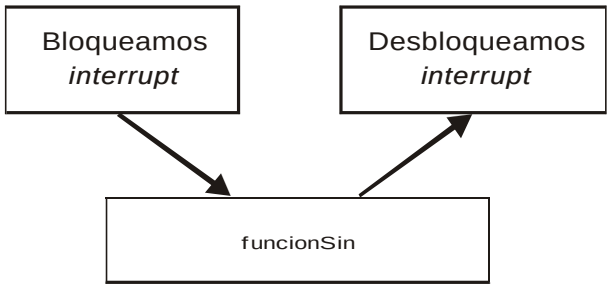


Figura 9

A continuación se muestran los códigos de cada una de estas funciones, en los que se pueden distinguir fácilmente cada uno de los nodos de que consta la Figura 9.

```
07328   /*===========================================================================*
07329    *                                lock_mini_send                             *
07330    *===========================================================================*/
07331   PUBLIC int lock_mini_send(caller_ptr, dest, m_ptr)
07332   struct proc *caller_ptr;        /* ¿quién trata de enviar mensaje? */
07333   int dest;                       /* ¿a quién se envía el mensaje? */
07334   message *m_ptr;                         /* puntero al buffer del mensaje */
07335   {
07336   /* Entrada segura a mini_send() para tareas. */
07337
07338     int result;
07339
07340     switching = TRUE;
07341     result = mini_send(caller_ptr, dest, m_ptr);
07342     switching = FALSE;
07343     return(result);
07344   }
07345
```

```
07346   /*===========================================================================*
07347    *                                lock_pick_proc                             *
07348    *===========================================================================*/
07349   PUBLIC void lock_pick_proc()
07350   {
07351   /* Entrada segura a pick_proc() para tareas. */
07352
07353     switching = TRUE;
07354     pick_proc();
07355     switching = FALSE;
07356   }
07357
07358   /*===========================================================================*
07359    *                                lock_ready                                 *
07360    *===========================================================================*/
07361   PUBLIC void lock_ready(rp)
07362   struct proc *rp;              /* este proceso ya es ejecutable */
07363   {
07364   /* Entrada segura a ready() para tareas. */
07365
07366     switching = TRUE;
07367     ready(rp);
07368     switching = FALSE;
07369   }
07370
07371   /*===========================================================================*
07372    *                                lock_unready                               *
07373    *===========================================================================*/
07374   PUBLIC void lock_unready(rp)
07375   struct proc *rp;              /* este proceso ya no es ejecutable */
07376   {
07377   /* Entrada segura a unready() para tareas. */
07378
07379     switching = TRUE;
07380     unready(rp);
07381     switching = FALSE;
07382   }
07383
07384   /*===========================================================================*
07385    *                                lock_sched                                 *
07386    *===========================================================================*/
07387   PUBLIC void lock_sched()
07388   {
07389   /* Entrada segura a sched() para tareas. */
07390
07391     switching = TRUE;
07392     sched();
07393     switching = FALSE;
07394   }
```

3.1.6.- La función unhold( )

Ésta es una función relacionada estrechamente con las capas más bajas de implementación en lenguaje ensamblador (función *_restart* implementada en *mpx386.s*) ya que su misión es procesar cíclicamente la cola de interrupciones detenidas (procesos que esperen una interrupción detenida), invocando la función interrupt (también implementada en proc.h pero relacionada con la comunicación de procesos) para cada una, a fin de convertir todas las interrupciones pendientes en mensajes antes de que se permita a otro proceso ejecutarse.

La forma de implementar esta cola de interrupciones detenidas es por medio del campo p_nextheld de la estructura de un proceso de Minix (en proc.h) :

*struct proc p_nextheld / * sigte. dn cadena de proc. int. detenidos */*

Gracias a este campo se incrustan las colas de procesos detenidos por mandarle una interrupción a cada proceso.

En el siguiente capítulo hablaremos de la forma en la que se comunican los procesos en Minix y veremos que una forma adicional de comunicación son las interrupciones, que se homogeinizan para construir un único sistema de comunicación de procesos.

También, se hace uso de otros dos punteros , **held_head** y **held_tail**, cuyo significado es similar al de rdy_head y rdy_tail , apuntando estos respectivamente al principio y final de la cola de procesos bloqueados por interrupciones detenidas.

Por otro lado, existe otro campo perteneciente a la estructura de un proceso que está relacionado directamente con esta función. Se trata del campo p_int_held, cuya misión es indicar que el proceso está bloqueado porque quiere mandar una interrupción a otro proceso y dicha interrupción se encuentra detenida por estar el otro proceso ocupado y no poder atenderla.

Si hacemos memoria acerca de la clasificación que hacíamos al principio de esta sección recordaremos que unhold pertenecía al grupo de funciones que usaban el semáforo switching. Así, la primera comprobación que hace unhold es sobre esta variable, retornando en caso de que esté bloqueado dicho semáforo. Se espera que la cola de procesos bloqueados por interrupciones detenidas no esté vacía (held_head # NIL_PROC) puesto que unhold la vaciará.

En la Figura 10 se puede ver el funcionamiento de esta función, en forma de diagrama de flujo, que representa el código que implementa a dicha función y que se muestra a continuación.

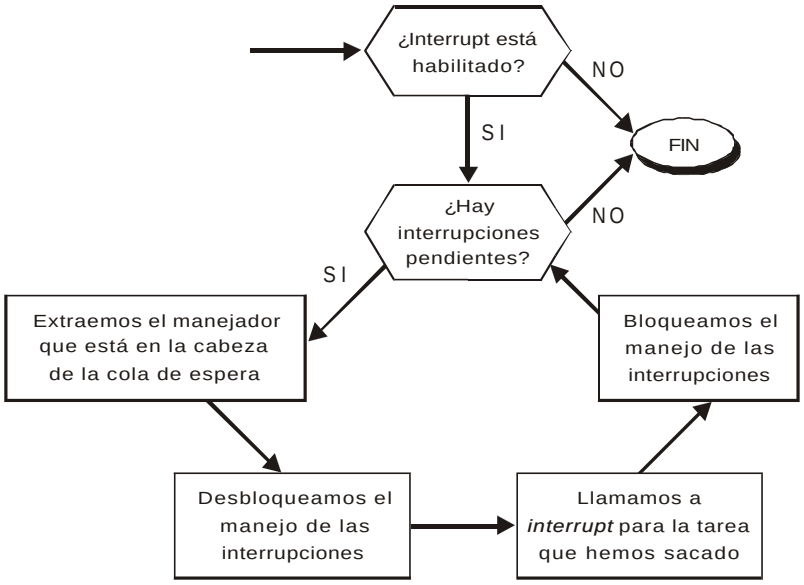


Figura 10

```
07397   /*===========================================================================*
07398    *                                unhold                                     *
07399    *===========================================================================*/
07400   PUBLIC void unhold()
07401   {
07402   /* Vaciar interrupciones detenidas. K_reenter debe ser 0. held_head no debe ser
07403    * NIL_PROC. Las interrupciones deben estar inhabilitadas. Se habilitarán
07404    * pero cuando éstas regresen se inhabilitarán.
07405    */
07406
07407     register struct proc *rp;         /* cabeza actual de cola de detenidas */
07408
07409     if (switching) return;
07410     rp = held_head;
07411     do {
07412             if ( (held_head = rp->p_nextheld) == NIL_PROC) held_tail = NIL_PROC;
07413             rp->p_int_held = FALSE;
07414             unlock();               /* reduce latencia; ¡cola detenidas podría cambiar! */
07415             interrupt(proc_number(rp));
07416             lock();                         /* proteger otra vez cola de detenidas */
07417     }
07418     while ( (rp = held_head) != NIL_PROC);
07419   }
```

A partir de la Figura 10 y de la relación entre cada uno de los nodos de éste y el código podemos ver cuál es la forma en la que se implementa esta función.

3.2.- Resumen

A modo de resumen podríamos decir que Minix emplea un algoritmo de planificación multinivel que se basa en el mantenimiento de tres colas de procesos ejecutables (Tareas, Servicios y Procesos de Usuario) de distintos niveles de prioridad. Dicho algoritmo consiste en encontrar la cola con más alta prioridad que no esté vacía y escoger el proceso que está a la cabeza de esa cola. En caso de que todas las colas estuviesen vacías, se ejecuta la rutina de "marcha en vacío" o IDLE.

En cuanto al momento en el que se produce la planificación destacamos dos:

- En el bloqueo y desbloqueo de procesos.
- En el fin de la rodaja de tiempo, o cuanto, de ejecución.

Por último, la planificación se implementa a partir de una serie de funciones que se encuentran en el fichero proc.c. Las principales son:

- **pick_proc :** Escoge el siguiente proceso a ejecutar.
- **ready :** Ingresa un proceso que se puede ejecutar en la cola de procesos ejecutables correspondiente.
- **unready :** Desloja a un proceso de la cola de procesos ejecutables correspondiente.
- **sched :** Coloca al proceso de usuario actual al final de su cola de ejecución por haber cumplido todo su tiempo asignado de ejecución. Esto sólo se lleva a cabo con los procesos de usuario puesto que estos son los únicos que Minix planifica según una política de Round Robin.
- **unhold :** procesa cíclicamente la cola de interrupciones detenidas convirtiendo todas las interrupciones pendientes en mensajes antes de que se permita a otro proceso ejecutarse.

4.- Comunicación entre procesos

La comunicación entre procesos se implantó a partir de la necesidad de emplear algún tipo de medio para poder cooperar entre ellos. Entre las distintas maneras que se puede llevar a cabo esta función, nos centraremos en el método implementado por MINIX: **mensajes**.

Para poder entender mejor el funcionamiento de esta metodología no hay nada mejor que entender las características básicas que describen cómo es, al igual que la estructura de datos que forma un mensaje.

4.1.- Conceptos Previos

La comunicación se puede clasificar en dos grupos: llamadas realizadas desde hardware y que van dirigidas hacia procesos (interrupciones); el otro grupo viene determinado por la forma en que se comunica un proceso al kernel para solicitar servicios del sistema (llamadas al sistema). El objetivo de ambos grupos es el mismo, incluso, en ambos casos la comunicación se realiza por mensajes. Sin embargo, están implementados en funciones separadas, porque el tratamiento que se le da a la comunicación es distinto. Por tanto sería más complicado ponerlo todo en una única función.

Un aspecto muy importante, es la principal diferencia que existe con respecto a otro sistema de comunicación, en concreto **memoria compartida** (área de memoria en la que se comparten variables para poder comunicarse distintos procesos), porque hace referencia, de una manera muy concisa, quién se encarga de manejar la comunicación: cuando manejamos mensajes la responsabilidad de la transmisión de los mismos recae sobre el sistema operativo. Es él quien se encargará de controlarlos y administrarlos para que todos lleguen a sus destinos. Sin embargo, cuando empleamos memoria compartida esta responsabilidad la ejerce el programador, ya que es él quien ha de gestionar la memoria.

La comunicación mediante mensajes está caracterizada por dos principios fundamentales:

- **Homogeneidad:** Esta característica se puede aplicar desde dos puntos de vista. El primero es que el kernel considera que todo lo que está en ejecución son procesos como procesos. Otro punto de vista es que todo intento de comunicación lo traduce a la transmisión de un mensaje, aunque primero se tenga que realizar una pequeña transformación, como ocurre en el caso de las interrupciones.

- **Sistema Basado en Citas:** En toda comunicación hay que establecer un mecanismo de planificación y organización para resolver casos en los que no se puede finalizar una transmisión porque ya se está atendiendo otra. En este caso, hay un parecido al uso de citas. ¿Qué queremos decir con esto? Pues bien, en el caso del emisor, si envía un mensaje y el kernel detecta que el proceso receptor no está esperando ningún mensaje del emisor, este no puede continuar su ejecución. Ha de esperar en una cola específica para poder enviar dicho mensaje. Lo mismo ocurrirá en el caso de un receptor que ordena recibir un mensaje. Si en

ese momento no existe ningún proceso deseando emitir, el proceso receptor se queda bloqueado hasta que recibe un mensaje.

Estos dos principios fundamentales describen, a grandes rasgos, este mecanismo. Sin embargo, podremos entenderlos mucho mejor cuando expliquemos, de una manera más profunda, el funcionamiento de cada una de las funciones que permiten este tipo de comunicación.

4.2.- Estructura *message*

En el apartado anterior hemos visto los principales rasgos que caracterizan este sistema de comunicación. Ahora bien, cada uno de los mensajes está constituido por un conjunto de campos que forman una estructura. Ésta se encuentra definida en el fichero *type.h* y se llama *message*. Su composición es la siguiente:

```
typedef struct {
    int m_source;                   // Identificación del proceso emisor
    int m_type;                     // Tipo de mensaje

    union {
        mess_1 m_m1;
        mess_2 m_m2;
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_6 m_m6;
    } m_u;
} message;
```

Con los dos primeros campos podemos identificar qué proceso emitió el mensaje (*m_source*) y qué tipo de mensaje es éste (*m_type*). El campo restante (*m_u*) identifica el "contenido" del mensaje. Es decir, en MINIX los mensajes se distribuyen según el destinatario, ya que cada conjunto se forma de una manera u otra. Como pudimos ver en temas anteriores, existen 6 clases de mensajes. Esto lo podemos comprobar observando la declaración de dichas clases:

```
typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
typedef struct {int m3i1, m3i2; char *m3p1; char m3ca1[M3_STRING];} mess_3;
typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
typedef struct {char m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;} mess_5;
typedef struct {int m6i1, m6i2, m6i3; long m6l1; sighandler_t m6f1;} mess_6;
```

A su vez todos estos parámetros están definidos en el mismo fichero, *type.h*, sin embargo, no mostraremos el resto del contenido de dicho fichero porque entendemos que no merece la pena, ya que no ayudaría a entender mejor el tema que estamos estudiando y nos desviaríamos demasiado de nuestro centro de interés.

4.3.- Comunicación entre Hardware y Procesos

Cuando un dispositivo hardware envía una interrupción, ésta tiene un tratamiento especial, tal y como pudimos ver al estudiar "Inicialización y manejo de interrupciones". Sin embargo, sólo estudiamos parte del tratamiento de la interrupción.

El camino completo que recorre una interrupción hasta que es atendida es el siguiente: El dispositivo lanza una señal física que se recoge en las patas del procesador encargado de las interrupciones y se lo comunica al procesador lanzando una instrucción ***INT*** que irá dirigida a su posición correspondiente en la IDT (Tabla de Descriptores de Interrupciones). En este punto, esta llamada se procesa, tal y como pudimos ver en temas anteriores, en el módulo ***HWINT_MASTER*** o ***SLAVE***, dependiendo de la interrupción producida. Esta rutina se encarga de *salvar* la situación actual y hacer una llamada a la entrada de la tabla IRQ que le corresponda. En ella encontraremos el manejador del dispositivo que ha generado la interrupción, es decir, su driver. Es un proceso que pertenece a la tarea asociada a ese dispositivo. También es conocido como ***handler***. Es éste quien hace una llamada a nuestra función para traducir la interrupción en un mensaje (***interrupt***).

Si el driver es el encargado de hacer esto, ¿a quién va dirigido el mensaje? Pues va dirigido a un proceso que está esperando por recibir un mensaje. Este proceso está dentro de la tarea asociada al dispositivo. Por lo que podríamos decir que es como si se llamara a sí mismo. Sin embargo, lo que ocurre es que quién llama a ***interrupt*** sólo se activa ante una llamada a la IRQ asociada y el que está esperando para actuar en consecuencia a dicha interrupción se activará cuando reciba el mensaje que estaba esperando.

Quizás podrá servir de ayuda el siguiente diagrama, en el que reflejamos cada uno de los pasos comentados.

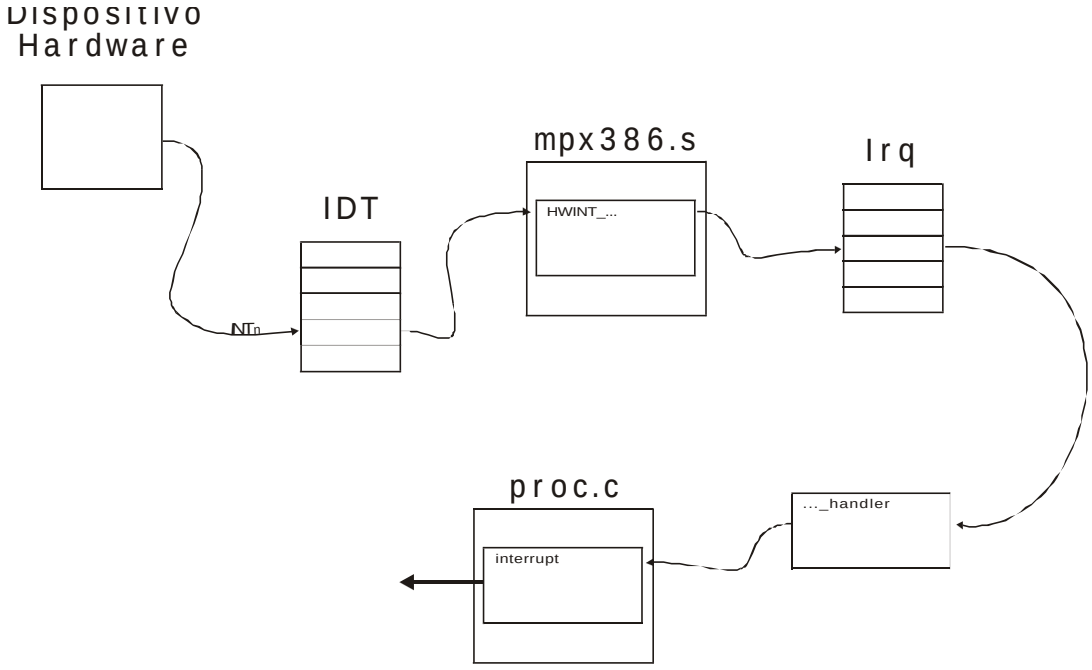


Figura 11

En concreto, estuvimos analizando hasta el punto en que llamamos al vector de interrupciones, el cual, a su vez, nos proporcionará la dirección de la tarea que se encarga de manejar dicha interrupción. Llegados a este punto tenemos que lanzar a ejecutar el "driver" que se encarga de actuar en consecuencia a la interrupción recibida.¿Cómo hacemos esto? Pues el manejador que llama el vector de interrupciones realiza una llamada a la función ***interrupt***, para que esta se encargue de hacer las comprobaciones pertinentes y, así, poder lanzar a ejecución el manejador del dispositivo que estaba esperando esa interrupción.

En resumen, éste es el tratamiento que recibe una interrupción. Ahora bien, nosotros nos centraremos en la función que nos ocupa: ***interrupt***, ya que el resto del tratamiento lo veremos en otros temas. Esta función tiene, como principal objetivo, recoger esa interrupción que se ha recibido, transformarla en un mensaje y lanzar a ejecución el manejador que estaba esperando por él.

Como único parámetro de la función es el número o identificador de la tarea (***task***) a la que va destinada el mensaje de la interrupción. ¿Por qué remarcamos que el mensaje va destinado a una tarea? Pues bien, si recordamos la organización "jerárquica" de la tabla de procesos en el nivel de mayor prioridad existía el nivel ***TASK_Q***. En él residirán todas aquellos procesos que se encargan de controlar los dispositivos de E/S. En otras palabras, las rutinas que tratan con el hardware. Por tanto, cuando se genera una interrupción hardware, después de realizar el correspondiente de haber ido al vector de interrupciones, si tiene que enviar un mensaje a un proceso, lo enviará hacia aquel proceso que se encargue de gestionarlo. Por eso el parámetro de entrada de ***interrupt***, aunque es un identificador de un proceso, se llama tarea. Para remarcar aún más que se comunicará con un proceso que está en el nivel ***TASK_Q***. A través de este identificador podemos obtener el puntero al proceso receptor mediante ***proc_addr(task)*** (*línea 6945*). Ahora ya podemos decir que estamos apuntando al proceso que espera recibir el mensaje que representa es interrupción. Esta dirección se la asignamos a una variable que es del tipo ***proc***. Esta estructura viene definida en el fichero ***proc.h***. Una breve descripción de la misma se hizo al describir, anteriormente, los campos más significativos.

Una vez que ya tenemos apuntado el proceso comprobamos si actualmente se está atendiendo a otra interrupción mediante la variable global ***k_reenter*** (*líneas 6962-6975*), la cual muestra el nivel de anidamiento de interrupciones. Si es negativo significa que aún no existe ninguna interrupción en espera de ser atendida. Si su valor es cero significa que hay una única interrupción esperando. Por último, si es mayor que cero, quiere decir que hay más de una interrupción esperando por ser atendidas. De esta manera, si no se da el caso de tener una única interrupción en espera de ser atendida o bien el vector de interrupciones ha sido inhabilitado por medio de la variable ***switching*** tendremos que poner el proceso en una cola para que espere y, cuando se termine de atender a la interrupción actual, despertar para recibir el mensaje de la interrupción.

Debido a que vamos a tener que tocar la cola de procesos que están esperando por una interrupción, debemos protegerla de una posible inconsistencia en su actualización. Para ello, antes de encolar se bloquea mediante la función ***lock()***. En realidad, lo que estamos haciendo es deshabilitar las interrupciones. Por supuesto, cuando hayamos terminado de actualizar la cola, desbloquearemos llamando a ***unlock()***, o lo que es lo mismo, volviendo a habilitar las interrupciones.

Volviendo un paso atrás, cuando ya tenemos la protección adecuada para actualizar la tabla de procesos tenemos que comprobar si nuestro proceso ya está en la cola de espera (*líneas*

6964-6972). Si no es así se activa el campo ***p_int_held*** para registrar su incorporación a la cola. A continuación se evalúa el estado de la cola. Si se encuentra vacía, ponemos el proceso en la primera posición de la misma. Si no es así, hacemos que sea el proceso que siga al último de la lista. En ambos casos nuestro proceso se convertirá en el último de la cola. Una vez actualizada la tabla de procesos se regresa. De manera que esta interrupción será atendida cuando se llame a la función ***unhold***. Función que se encarga de repetir todas las interrupciones, para que "despierten" las tareas que están esperando en la cola..

Si pudo pasar el filtro anterior, ahora se presenta una nueva comprobación: ¿realmente el proceso estaba esperando una interrupción? ¿esa interrupción proviene de un dispositivo hardware (función ***isrxhardware***)? (*líneas 6978-6982*) Esta pregunta se hace porque se puede dar el caso que un dispositivo lance una interrupción y la tarea no está preparada para recibirla. ¿Cuándo se dará ese caso? Pues una situación en la que se puede dar es cuando, por ejemplo, la tarea del disco duro no está esperando un mensaje del dispositivo hardware, sino del FS. En este caso se bloquea el proceso activando su campo ***p_int_blocked***. Por supuesto, después de hacer esto se sale de la función ***interrupt***.

Hay que diferenciar la cola enlazada por el campo ***p_nextheld*** de la activación del campo ***p_int_blocked***. Este último indica que la tarea está ocupada atendiendo otra interrupción o, también puede indicar que no esperaba una interrupción del dispositivo hardware. Sea uno u otro caso, indica que se le ha intentado mandar una interrupción y no se ha podido. De esta manera, como veremos al final de este tema, la función ***mini_rec*** se encargará de comprobar este flag y reconstuir el mensaje. Sin embargo, la lista enlazada correspondiente a ***p_nextheld***, estarán aquellas tareas que esperan por atender una interrupción y no lo pueden hacer porque ya hay otra atendiéndose. Ahora bien, no es nuestra tarea quien está atendiendo la interrupción, sino otra. Otra gran diferencia es que para poder reconstruir la emisión de este mensaje y vaciar la cola se empleará la función ***unhold***.

Retomando el curso de nuestra función, una vez sobrepasadas todas estas comprobaciones ya estamos preparados para transmitir la interrupción. Aunque primero tendremos que traducirla en un mensaje. Por tanto rellenamos el contenido de los campos de la estructura que forma un mensaje indicando como emisor HARDWARE y tipo de mensaje HARD_INT. También tenemos que poner a cero el flag RECEIVING y desbloquear el proceso, aunque no hay problema si se hace sin estar bloqueado (*líneas 6989-6992*). Si nos fijamos, estamos tocando directamente los campos para formar el mensaje, en vez de llamar a una función para que lo haga.

Por último, actualizamos las colas de ejecución de la tabla de procesos (*líneas 6997-7002*). En nuestro caso, al tratarse de un dispositivo hardware estaremos dentro de la cola ***TASK_Q***. En otras palabras, desde el punto de vista de la tabla de procesos, el dispositivo hardware se ha transformado en un proceso que se está tratando en esta zona de la tabla. Por tanto, si la cola estaba vacía insertamos como cabeza a nuestro proceso y, si no lo estaba, lo insertamos al final. Como ocurría en la actualización de la cola de procesos en espera para atender una interrupción, en ambos casos la nueva cola de la lista será nuestro proceso.

El funcionamiento que hemos acabado de explicar se ve reflejado en el siguiente diagrama de flujo, Figura 12:

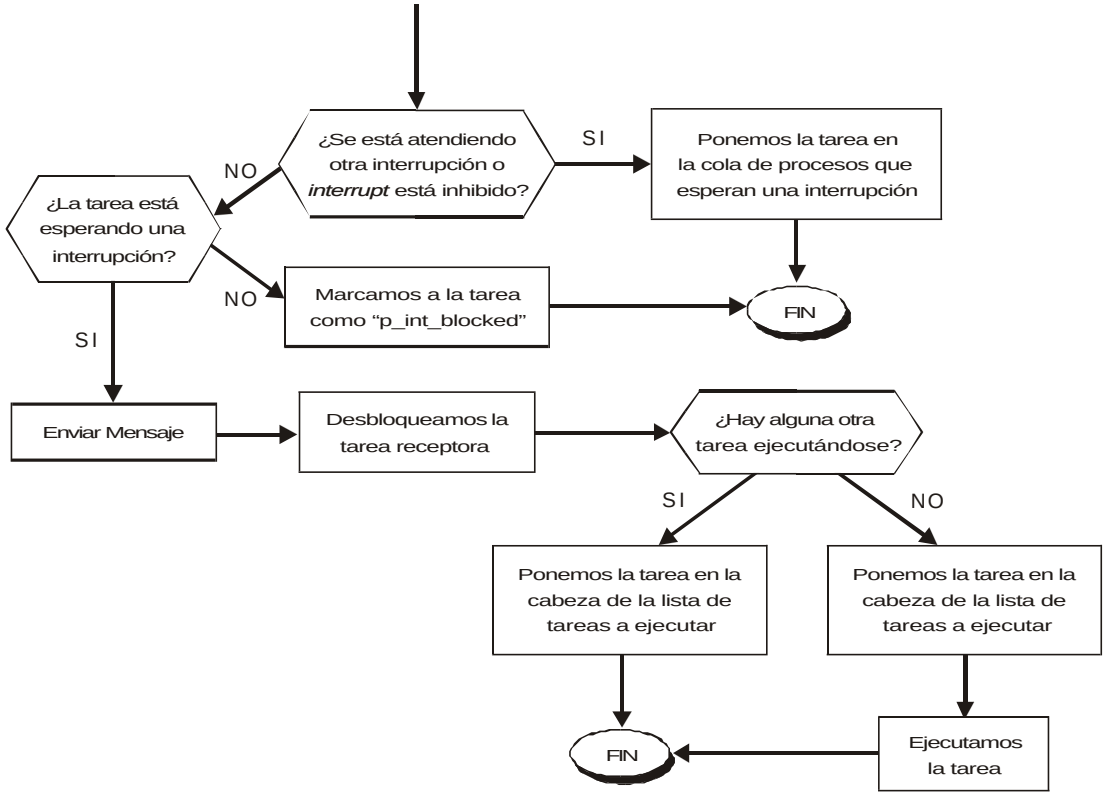


Figura 12

Finalmente, el código fuente de esta función es el siguiente:

```
06935   /*===========================================================================*
06936    *                                interrupt                                  *
06937    *===========================================================================*/
06938   PUBLIC void interrupt(task)
06939   int task;                       /*  # de la tarea que se iniciará */
06940   {
06941   /* Ocurrió interrupción. Planificar la tarea que la maneja. */
06942
06943     register struct proc *rp;       /* puntero a la entrada del proceso de la tarea */
06944
06945     rp = proc_addr(task);
06946
06947     /* Si esta función competiría con otras funciones de conmutación de procesos,
06948      * ponerla en la cola 'detenidos' que se vaciará en el siguiente restart().
06949      * Las condiciones de competición son:
06950      * (1) k_reenter == (typeof k_reenter) -1:
06951      *     Llamada desde el nivel de tareas, normalmente de una rutina
06952      *     de interrupciones de salida.  Un manejador de  interrupciones podría reingresar a interrupt().
06953      *     Poco frecuente; no tiene tratamiento especial.
06954      * (2) k_reenter > 0:
06955      *     Llamada desde un  manejador de interrupciones anidado.  Un manejador de interrupciones
06956      *     previo podría estar dentro de interrupt() sys_call().
06957      * (3) switching != 0:
06958      *     Una función de conmutación de procesos distinta a interrupt() está siendo invocada
06959      *     del nivel de tareas, por lo común sched() de CLOCK.  Un manejador
06960      *     de interrupciones podría invocar interrupt y pasar la prueba de k_reenter.
06961      */
06962     if (k_reenter != 0 || switching) {
06963             lock();
06964             if (!rp->p_int_held) {
06965                     rp->p_int_held = TRUE;
06966                     if (held_head != NIL_PROC)
06967                             held_tail->p_nextheld = rp;
```

```
06968                           else
06969                                   held_head = rp;
06970                           held_tail = rp;
06971                           rp->p_nextheld = NIL_PROC;
06972                   }
06973                   unlock();
06974                   return;
06975           }
06976
06977           /* Si la tarea no espera interrupción, registrar el bloqueo. */
06978           if ( (rp->p_flags & (RECEIVING | SENDING)) != RECEIVING ||
06979               !isrxhardware(rp->p_getfrom)) {
06980                   rp->p_int_blocked = TRUE;
06981                   return;
06982           }
06983
06984           /* El destino está esperando una interrupción.
06985            * Enviarle el mensaje con origen HARDWARE y tipo HARD_INT.
06986            * No puede proporcionarse, de forma confiable, más infoprmación porque los
06987            * mensajes de interrupción no se ponen en cola.
06988            */
06989           rp->p_messbuf->m_source = HARDWARE;
06990           rp->p_messbuf->m_type = HARD_INT;
06991           rp->p_flags &= ~RECEIVING;
06992           rp->p_int_blocked = FALSE;
06993
06994            /* Hacer a rp listo y ejecutarlo se otra tarea no se está ejecuntando ya. Esto
06995             * es ready(rp) en-linea para mayor velocidad.
06996             */
06997           if (rdy_head[TASK_Q] != NIL_PROC)
06998                   rdy_tail[TASK_Q]->p_nextready = rp;
06999           else
07000                   proc_ptr = rdy_head[TASK_Q] = rp;
07001           rdy_tail[TASK_Q] = rp;
07003           rp->p_nextready = NIL_PROC;
07003       }
```

4.4.- Comunicación generada por Software: Llamadas al Sistema

Hasta ahora hemos estado explicando cómo se comunica un dispositivo hardware con un proceso. En cambio, en este apartado explicaremos el funcionamiento de una comunicación generada por el software. Mejor conocida como *llamada al sistema*. ¿Qué casos abarca esta llamada? Pues todos aquellos casos que requiramos la ayuda del núcleo. Nos podemos encontrar, entre otras, leer o escribir en disco, comunicarnos con la impresora o solicitar la hora.

Así pues, éste es el mecanismo que emplea un proceso de usuario cuando quiere ponerse en contacto con el núcleo, pero, como ocurrió en el apartado anterior, el transcurso de este proceso no es tan simple como parece. De hecho tiene un cierto parecido con el tratamiento de una interrupción generada por un dispositivo hardware. Por tanto, empezaremos desde el principio, o lo que es lo mismo, desde un programa en ejecución. Pongamos, por ejemplo, que en alguna parte de su código realiza una llamada al sistema o, lo que es lo mismo, llamar a una función que está en una de las librerías de nuestro programa. Esta función realiza una serie de procesos que veremos en temas posteriores. Ahora bien, llegará un momento que realizará una llamada mediante ***INT***, que se hará junto a un identificador. Éste es la dirección de la IDT asociada a las llamadas al sistema. Como podemos comprobar, todo este proceso es muy parecido al tratamiento de una interrupción hardware.

A continuación, en esa posición de la tabla se trata debidamente a la llamada mediante la función ***s_call*** perteneciente al fichero ***mpx386.s***. Se encarga de "salvar" el estado actual y llamar a ***sys_call***, es decir, nuestra función. A partir de este momento el proceso que continua se ve reflejado en el análisis de esta función.

Este funcionamiento lo mostramos en el siguiente diagrama:

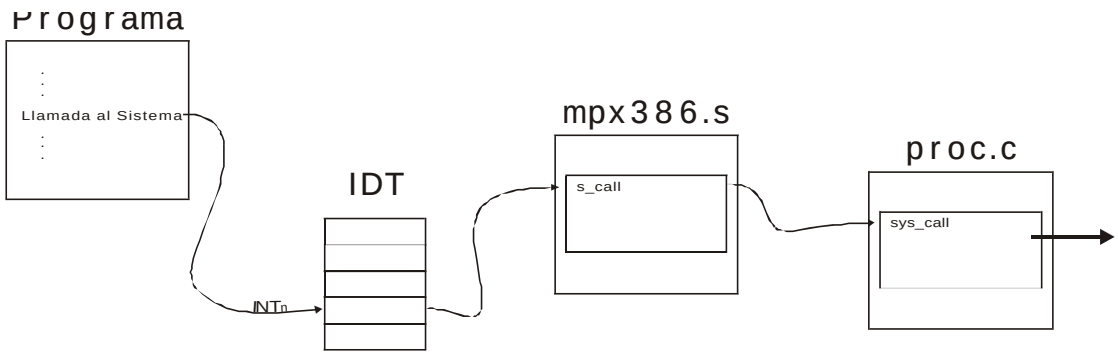


Figura 13

Antes de empezar, es necesario hacer un comentario previo. El objetivo principal es traducir la interrupción generada por software en un mensaje y enviarlo, muy parecido a ***interrupt***. Sin embargo, para una mejor fiabilidad y funcionamiento del sistema es recomendable implementarlas por separado, pues en el caso software, el tratamiento tiene un mayor número de variantes. En cambio, en el otro caso sólo existe un posible tipo de mensajes o de funcionamiento, transmitir un mensaje de parte de un dispositivo hardware.

Por otra parte, ***sys_call*** tiene una larga lista de cosas que hacer en la comunicación, pero las distribuye modularmente, de forma que empleará otras funciones. Ahora bien, para comenzar su análisis hay que entender los parámetros que contiene en su declaración:

- ***function:*** Indica qué función de comunicación desea hacer el proceso que ha realizado la llamada al sistema. Hay tres tipos: enviar (***SEND***), recibir (***RECEIVE***) y ambos (***BOTH***).
- ***src_dest:*** Fuente o proceso emisor del mensaje que vamos a recibir o destino del mensaje que vamos a enviar. Es un identificador numérico.
- ***m_ptr:*** Puntero al mensaje. Es del tipo ***message***.

Ahora ya estamos preparados para comprender los pasos que se van haciendo. Así pues, en primer lugar nos encontramos con un primer filtro (*línea 7023*). Hacemos uso de una macro: ***isoksrc_dest(src_dest)***. Está definida en el fichero ***proc.h***. Ya que es una macro muy simple, creemos conveniente saber cuál es su funcionamiento y, de esta manera, comprender la condición que se pone para poder realizar una comunicación. Por tanto, su código es el siguiente:

```
#define isokprocn (n)          ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isoksrc_dest (n)       (isokprocn (n) || (n) == ANY)
```

Como podemos comprobar, la propia macro hace referencia a otra (***isokprocn***). Ésta última comprueba que el número identificador del proceso que poseemos no sobrepasa el límite establecido en la tabla de procesos. ¿Cómo lo sabe? Muy fácil. Por una parte tiene el número de ranuras en la tabla de procesos (***NR_PROCS***) y por otra parte tenemos el número de tareas en el vector de transferencia. Observando el código anterior, se comprende que comprueba que el valor del identificador del proceso sea inferior al límite máximo de ranuras para poder referenciar a un proceso. Así pues, la misión de la macro ***isoksrc_dest*** es verificar si el identificador del proceso ***src_dest*** está dentro del dominio de valores que aceptamos o, en todo caso, estemos referenciando al identificador ***ANY***. Este identificador significa que se trata de un mensaje de "cualquier" fuente.

Volviendo a la comprobación original que estábamos realizando, si detecta que hay error para la ejecución de la función y devuelve un error. Parece extraño que hagamos una comprobación de este estilo cuando se supone que no deben producirse errores de este tipo. Sin embargo, la función ***sys_call*** es una función que va a ser llamada muchas veces y es importante que sea robusta.

Una vez ya sabemos que el identificador del proceso ***src_dest*** es correcto cogemos el puntero al proceso "protagonista", o sea, el proceso que se está ejecutando el cual deberá pasar un filtro de comprobación (*línea 7026*). Éste consiste en comprobar, mediante la función ***isuserp***, si el proceso está dentro de la zona de la tabla de procesos correspondiente al nivel de usuario, ya que si es así y además queremos enviar y recibir un mensaje a la vez, pararemos y emitiremos un error. Esto ocurre porque no está permitido que un proceso correspondiente a un usuario pueda hacer ambas cosas a la vez. Envía o recibe, pero las dos cosas a la vez no puede ser.

Después de haber pasado todos los filtros anteriores ya estamos en situación de realizar la comunicación. Por tanto, comprobando el parámetro correspondiente (***function***) nos dirá qué tenemos que hacer. Si vamos a enviar (*líneas 7029-7034*), llamamos a ***mini_send***, función que se encargará de transmitir el mensaje. Ésta nos devolverá un valor para el control de posibles errores (*líneas 7032-7033*). Si hubo un error o bien lo único que pretendíamos era enviar un mensaje, ***sys_call*** termina en este punto. Sin embargo, si también queríamos leer o, ni tan siquiera pretendíamos enviar, pasaremos a la última parte de esta función: llamar a ***mini_rec*** para recibir un mensaje. Ahora bien, para ejecutar esta llamada no tenemos que comprobar si queremos leer, ya que si no queremos recibir ningún mensaje habremos salido en el punto anterior, es decir, al finalizar ***mini_send***. Estas dos funciones las explicaremos en los dos próximos apartados.

Al igual que hicimos anteriormente, a continuación vamos a reflejar en un diagrama de flujo el funcionamiento que hemos estado explicando, así será más fácil de asimilar como trabaja ***sys_call***.

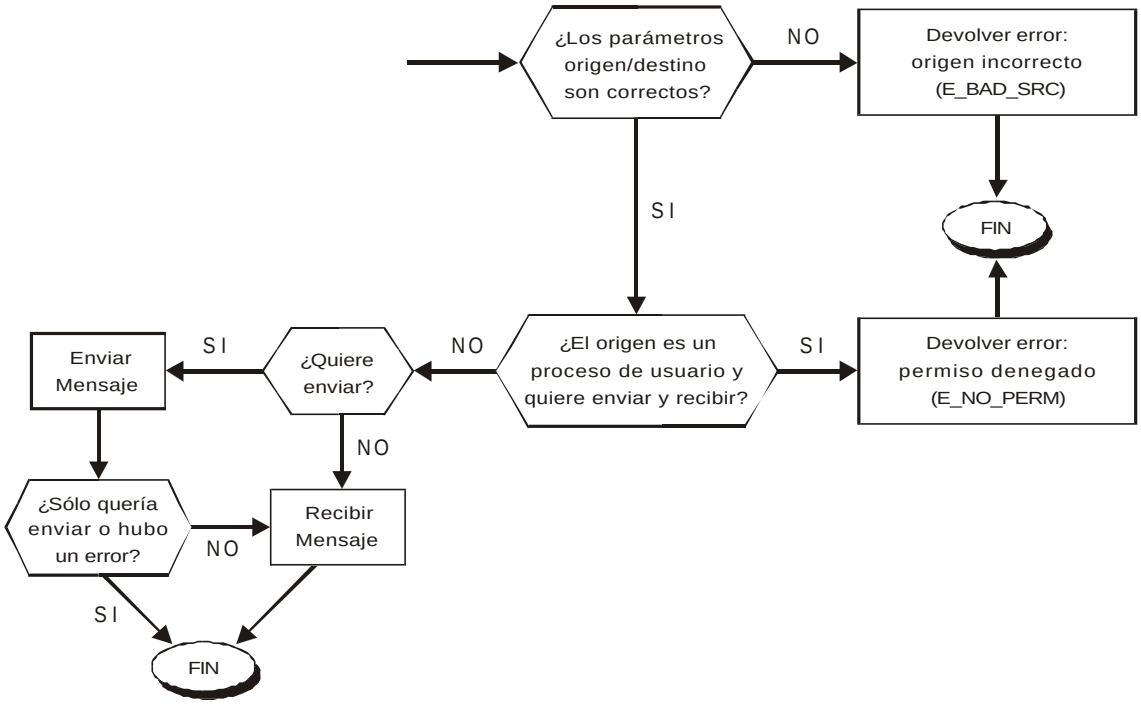


Figura 14

Ahora ya estamos en situación de mostrar el código fuente:

```
07005   /*===========================================================================*
07006    *                                sys_call                                   *
07007    *===========================================================================*/
07008   PUBLIC int sys_call(function, src_dest, m_ptr)
07009   int function;                   /* SEND, RECEIVE, or BOTH */
07010   int src_dest;                   /* source to receive from or dest to send to */
07011   message *m_ptr;                 /* pointer to message */
07012   {
07013   /* Las únicas llamadas al sistema que existen en MINIX son enviar y recibir mensajes,
07014    * y se efectúan entrando por trampa al kernel con una instrucción INT.
07015    * Se recoge la trampa y se llama a sys_call() para enviar o recibir un mensaje
07016    * (o ambas cosas). Proc_ptr siempre indica quién llama.
07017    */
07018
07019    register struct proc *rp;
07020    int n;
07021
07022    /* Detectar párametros de llamada al sistema erróneos. */
07023    if (!isoksrc_dest(src_dest)) return(E_BAD_SRC);
07024    rp = proc_ptr;
07025
07026    if (isuserp(rp) && function != BOTH) return(E_NO_PERM);
07027
07028    /* Los parámetros están correctos. Efectuar la llamada. */
07029    if (function & SEND) {
07030            /* Function = SEND or BOTH. */
07031            n = mini_send(rp, src_dest, m_ptr);
07032            if (function == SEND || n != OK)
07033                    return(n);  /* hecho, o falló SEND */
07034    }
07035
07036    /* Función = RECEIVE o BOTH.
07037     * Comprobamos que las llamadas de usuario son BOTH, y confiamos en 'function' por lo demás.
07038     */
07039    return(mini_rec(rp, src_dest, m_ptr));
07040   }
```

4.4.1.- mini_send

El objetivo de esta función se centra en el envío de un mensaje que emite un proceso fuente dirigido a un proceso destino. No requiere hacer ninguna comprobación a nivel de proceso porque todas ellas ya se han hecho en ***sys_call***.

Antes de empezar a comentar los parámetros que vamos a emplear, podemos comprobar que se trata de una función privada, al igual que ocurre con ***mini_rec***. Son las dos únicas que no son públicas. Tiene una sencilla explicación: estas funciones sólo serán invocadas por las dos funciones anteriores, es decir, ***interrupt*** y ***sys_call***.

Una vez analizado este detalle, pasamos a estudiar los parámetros. Como pudimos ver en el código fuente del apartado anterior, tiene tres parámetros de entrada:

- **caller_ptr:** identificador del proceso emisor.
- **dest:** receptor del mensaje.
- **m_ptr:** puntero al buffer del mensaje, o lo que es lo mismo, la zona de memoria donde reside el mensaje.

Con respecto al mensaje nos podríamos preguntar ¿dónde se encuentra el buffer? Pues bien, se encuentra en una zona de memoria que está dentro de la tabla de procesos (lo podremos observar en comprobaciones posteriores). Sin embargo, vemos una cosa que causa cierta incertidumbre: ¿por qué hacemos referencia a las posiciones de memoria con "***CLICKS***"? La respuesta es porque, como veremos en el tema de manejo de memoria en MINIX, hay una clase de asignación de memoria que se llama por pulsos o ***clicks***. Éstos son la unidad básica de asignación de memoria. Aunque al no ser una asignación de memoria normal sino empleando los clicks, se dice que empleamos *memoria virtual*, ya que las direcciones serán múltiplos de 16 bytes.

Aclaradas estas dudas, nos disponemos a empezar a analizar el código, encontrándonos, en primer lugar, con el primer filtro (*línea 7060*) encargado de verificar que, si estamos en el caso de tener como emisor un proceso de usuario, éste sólo podrá enviar al sistema de ficheros, FS, y al manejador de memoria, MM. Si se excede se producirá un error.

El siguiente paso es comprobar que el proceso al que va dirigido el mensaje esté "vivo". Esta comprobación la haremos analizando el contenido de los flags de la ranura de la tabla de procesos que apunta el identificador, que poseemos, del receptor (*línea 7062*). Si la ranura no está en uso, se producirá un error. ¿Cuándo se puede dar este caso? Un claro ejemplo es el siguiente: cuando el emisor va a enviar el mensaje el receptor, que hasta ese momento estaba vivo, muere.

El siguiente filtro se encarga de comprobar el contenido del mensaje (*líneas 7071-7073*). Primero calcula la transformación necesaria de la dirección de memoria del mensaje en el dominio de la *memoria virtual*, o lo que es lo mismo, hace un desplazamiento para coger los

bits que emplea para direccionar. Así podremos tener la dirección inicial y final del mensaje, por lo que ahora analizamos en qué zona de segmento del proceso está. Verifica si el mensaje cae totalmente en el segmento de datos, cae en el segmento de código o en el espacio entre ellos (estas zonas lo veremos cuando estudiemos la distribución en segmentos de la zona de memoria desiganda para un proceso). Si no es así devolveremos un error.

A continuación, buscamos un bucle cerrado de transferencia de mensajes (*líneas 7076-7085*). Vamos recorriendo, a partir del proceso al que quiero enviar un mensaje, el siguiente proceso al que este último le enviará. Es decir, recorre la lista dinámica de procesos que están relacionados por enviar un mensaje al siguiente. Si en algún momento vuelvo a llegar al mío, significa que existe un bucle cerrado y entonces devolvemos un error. En cambio, si llegamos al final de la lista y no ha ocurrido esto, podremos proseguir.

Por fin hemos llegado a la última prueba antes de enviar el mensaje (*lineas 7088-7112*). Ahora tenemos que comprobar si el proceso destino está esperando recibir algo, de mí o de cualquiera. Si es así invocaremos a la macro ***CopyMess*** que está definida al principio del fichero ***proc.c*** para copiar el mensaje a su buffer. A continuación, desactivamos el flag de recibir del proceso destino y si éste no pensaba enviar nada, es decir, si el receptor de nuestro mensaje sólo esperaba recibir un mensaje llamamos a la función ***ready*** para continuar ejecutándose. Si no fuera así y quisiera enviar algo no podría continuar hasta que envíe su mensaje correspondiente.

En cambio, si el destino no está esperando recibir nada, entonces el proceso emisor pasa a estar bloqueado, ya que no ha podido enviar el mensaje. No podrá continuar ejecutándose hasta que no haya finalizado su envío. Por esta razón decimos que el mecanismo de comunicación de MINIX se basa en el principio de cita. No podré continuar hasta no haber completado esta misión. Al igual que pasa cuando el receptor está esperando, se quedará bloqueado hasta que no reciba lo que esperaba.

A parte de marcar al proceso como “no preparado” debemos encolarlo en la lista de procesos que están esperando para enviar al proceso destino (*líneas 7104-7111*). ¿Cómo se hace? Muy fácil. Cómo vimos en la explicación de los campos de cada ranura de la tabla de procesos, hay un campo, ***callerq***, encargado de apuntar al primer proceso que está esperando por enviar. Es como si fuera la cabeza de la cola. Ahora bien, ha de estar con este contenido en la ranura del proceso receptor. La lista se enlaza por medio de la información que tenga el campo ***sendlink*** de la ranura del emisor apuntado por ***callerq***. Si tiene el identificador de otro proceso, entonces nos vamos a ese para analizar el contenido de su campo ***sendlink***. La lista continuará hasta que en este campo haya un ***NIL_PROC***.

A continuación mostramos el digrama de funcionamiento y el código fuente de esta función:

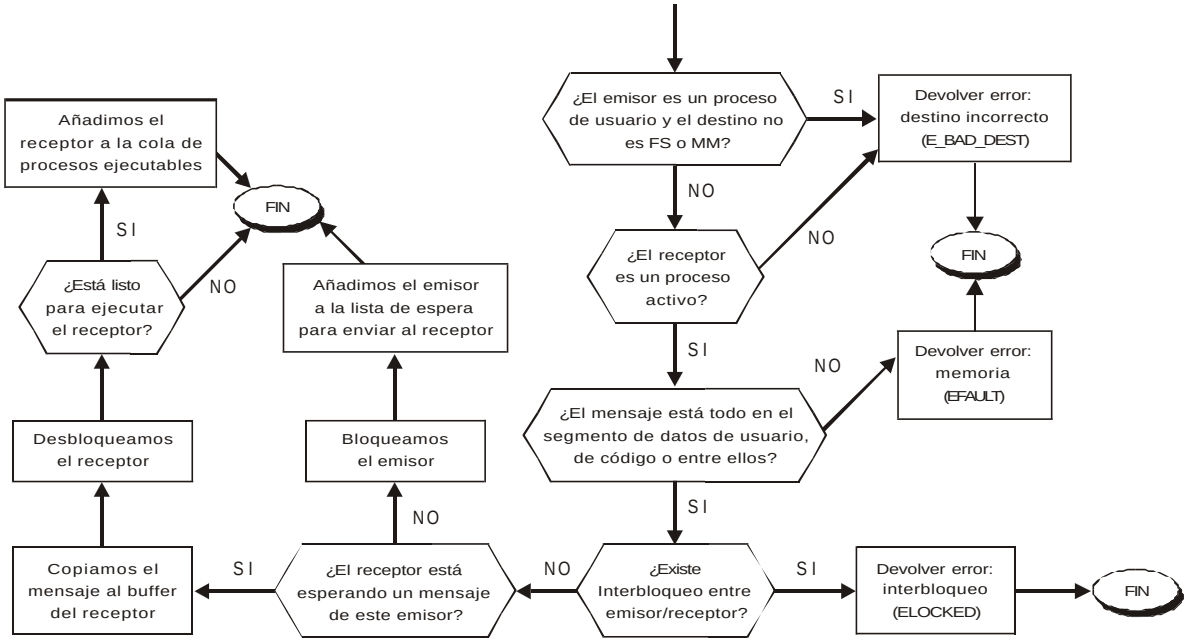


Figura 15

```
07042   /*===========================================================================*
07043    *                                mini_send                                  *
07044    *===========================================================================*/
07045   PRIVATE int mini_send(caller_ptr, dest, m_ptr)
07046   register struct proc *caller_ptr;       /* ¿quién está intentando enviar un mensaje? */
07047   int dest;                       /* ¿a quién va dirigido ese mensaje? */
07048   message *m_ptr;                 /* puntero al buffer de mensaje */
07049   {
07050   /* Enviar mensaje de 'caller_ptr' a 'dest'. Si 'dest' está bloqueado
07051    * esperando este mensaje, copiarlo en él y desbloquear 'dest'. Si 'dest'
07052    * no espera, o espera otro origen, formar en cola 'caller_ptr'.
07053    */
07054
07055    register struct proc *dest_ptr, *next_ptr;
07056    vir_bytes vb;                   /* puntero al buffer mensaje como vir_bytes */
07057    vir_clicks vlo, vhi;            /* clicks virtuales con mensaje por enviar */
07058
07059    /* Proceso de usuario sólo pueden enviar a FS y MM. Comprobar esto. */
07060    if (isuserp(caller_ptr) && !issysentn(dest)) return(E_BAD_DEST);
07061    dest_ptr = proc_addr(dest);  /* puntero a entrada de proceso destino */
07062    if (dest_ptr->p_flags & P_SLOT_FREE) return(E_BAD_DEST);        /* destino muerto */
07063
07064    /* Esta prueba permite que el mensaje esté en datos, pila o espacio
07065     * intermedio. Tendrá que hacerse más completa en máquinas
07066     * que no tienen el espacio mapeado.
07067     */
07068    vb = (vir_bytes) m_ptr;
07079    vlo = vb >> CLICK_SHIFT;    /* clicks virtuales para la base del mensaje */
07070    vhi = (vb + MESS_SIZE - 1) >> CLICK_SHIFT;    /* clicks virtuales para el tope del mensaje */
07071    if (vlo < caller_ptr->p_map[D].mem_vir || vlo > vhi ||
07072        vhi >= caller_ptr->p_map[S].mem_vir + caller_ptr->p_map[S].mem_len)
07073          return(EFAULT);
07074
07075    /* Detectar bloque mortal si 'caller_ptr' y 'dest' se envían mutuamente. */
07076    if (dest_ptr->p_flags & SENDING) {
07077            next_ptr = proc_addr(dest_ptr->p_sendto);
07078            while (TRUE) {
07079                    if (next_ptr == caller_ptr) return(ELOCKED);
07080                    if (next_ptr->p_flags & SENDING)
07081                            next_ptr = proc_addr(next_ptr->p_sendto);
07082                    else
```

```
07083                                   break;
07084                   }
07085           }
07086
07087           /*Ver si 'dest' está bloqueado esperando este mensaje. */
07088           if ( (dest_ptr->p_flags & (RECEIVING | SENDING)) == RECEIVING &&
07089                   (dest_ptr->p_getfrom == ANY ||
07090                    dest_ptr->p_getfrom == proc_number(caller_ptr))) {
07091                   /* El destino sí está esperando este mensaje. */
07092                   CopyMess(proc_number(caller_ptr), caller_ptr, m_ptr, dest_ptr,
07093                                   dest_ptr->p_messbuf);
07094                   dest_ptr->p_flags &= ~RECEIVING;        /* desbloquear destino */
07095                   if (dest_ptr->p_flags == 0) ready(dest_ptr);
07096           } else {
07097                   /* El destino no espera. Bloqueo y formar en cola invocador. */
07098                   caller_ptr->p_messbuf = m_ptr;
07099                   if (caller_ptr->p_flags == 0) unready(caller_ptr);
07100                   caller_ptr->p_flags |= SENDING;
07101                   caller_ptr->p_sendto= dest;
07102
07103                   /* El proceso ya está bloqueado. Ponerlo en la cola del destino. */
07104                   if ( (next_ptr = dest_ptr->p_callerq) == NIL_PROC)
07105                           dest_ptr->p_callerq = caller_ptr;
07106                   else {
07107                           while (next_ptr->p_sendlink != NIL_PROC)
07108                                   next_ptr = next_ptr->p_sendlink;
07109                           next_ptr->p_sendlink = caller_ptr;
07110                   }
07111                   caller_ptr->p_sendlink = NIL_PROC;
07112           }
07113           return(OK);
07114   }
```

4.4.2.- mini_rec

Básicamente, la función que vamos a ver en este apartado se encarga de entregar un mensaje que emite un proceso emisor a su correspondiente destino. Al igual que ocurría en el apartado anterior la función se ve liberada de algunas comprobaciones al realizarlas ***sys_call*** antes de llamarlas.

Empezando por los parámetros, podemos ver que los tiene distribuidos de forma similar a la que hemos estado viendo hasta ahora: el identificador del proceso receptor (***caller_ptr***), el del emisor (***src***) y un puntero al buffer del mensaje (***m_ptr***).

Lo primero que hacemos es recorrer la cola de procesos que están esperando por enviar un mensaje a nuestro receptor (*líneas 7139-7150*). Si encontramos un emisor que sea compatible con ***src***, es decir, o estamos esperando cualquier emisor (***src == ANY***) o encontramos al emisor que buscamos. Como un proceso se bloquea cuando no puede finalizar su emisión (o recepción), en esta cola cada proceso aparecerá, como máximo, una vez. Así, una vez encontrado nuestro proceso emisor finalizamos la función sin seguir buscando. Además, lo tenemos que hacer porque se había dado la orden de recibir un único mensaje. No podremos recibir el resto de los mensajes que están esperando por ser enviados hasta que el proceso no espere más mensajes.

Por tanto, si encontramos nuestro mensaje, realizamos la copia del mismo, invocando la macro ***CopyMess***. Al terminar esta transmisión actualizamos, en dos pasos, la cola de procesos en espera (*líneas 7143-7148*):

- Primero actualizamos los punteros que enlazan la lista dinámica, teniendo especial cuidado por si estamos en la primera posición.

- A continuación se comprueba el estado de los flags del emisor, ya que si estaba bloqueado sólo por esta transmisión, podemos invocar la función ***ready*** para que pueda continuar su ejecución.

Si no encontramos el emisor que estábamos esperando, comprobamos si el receptor está bloqueado debido a que ***interrupt*** le intentó enviar un mensaje y nuestro proceso no estaba preparado para recibir. Además de esto, si el emisor es un dispositivo hardware, realizamos la copia o transferencia del mensaje de la misma manera que en ***interrupt***, para finalmente desbloquearlo y finalizar (*líneas 7154-7159*).

Si finalmente no se da ninguna de las situaciones anteriores, bloqueamos el proceso receptor ya que está esperando un mensaje que todavía no existe o, en otras palabras, que el emisor todavía no ha intentado enviar.

Las últimas instrucciones están orientadas al MM (*líneas 7171-7172*). Si detectamos que hay señales generadas por el kernel, tales como ***SIGINT*** *(interrupción producida por DEL)*, ***SIGQUIT****(salir)* y ***SIGALARM*** *(alarma del reloj)*, por atender y el receptor es el proceso MM que está esperando un mensaje de cualquier proceso, realizamos la transmisión del mensaje mediante la invocación de la función ***inform***, que estudiaremos dentro de algunos temas.

En último lugar, nos disponemos a mostrar el código fuente de ***mini_rec*** y el diagram de funcionamiento correspondiente:

```
07116   /*===========================================================================*
07117    *                                    mini_rec                               *
07118    *===========================================================================*/
07119    PRIVATE int mini_rec(caller_ptr, src, m_ptr)
07120    register struct proc *caller_ptr;       /* proceso que trata de obtener mensaje */
07121    int src;                        /* fuente del mensaje (o ANY) */
07122    message *m_ptr;                 /* Puntero al buffer del mensaje */
07123    {
07124    /* Un proceso o tarea quiere obtener un mensaje. Si hay uno en cola,
07125     * adquirirlo y desbloquear el emisor. Si no hay mensaje del origen deseado,
07126     * bloquear invocador. No hay que comprobar validez de parámetros. Las llamadas
07127     * de usuario diempre son sendrec(), y mini_send() ya comprobó. Se confía
07128     * en las llamadas de las tareas, MM y FS.
07129     */
07130
07131     register struct proc *sender_ptr;
07132     register struct proc *previous_ptr;
07133
07134     /* Ver si ya está disponible un mensaje del origen deseado. */
07135     if (!(caller_ptr->p_flags & SENDING)) {
07136             /* Verificar la cola de emisores. */
07137       for (sender_ptr = caller_ptr->p_callerq; sender_ptr != NIL_PROC;
07138               previous_ptr = sender_ptr, sender_ptr = sender_ptr->p_sendlink) {
07139             if (src == ANY || src == proc_number(sender_ptr)) {
07140                     /* Se encontró un mensaje aceptable. */
07141                     CopyMess(proc_number(sender_ptr), sender_ptr,
07142                             sender_ptr->p_messbuf, caller_ptr, m_ptr);
07143                     if (sender_ptr == caller_ptr->p_callerq)
07144                             caller_ptr->p_callerq = sender_ptr->p_sendlink;
07145                     else
```

```
07146                                   previous_ptr->p_sendlink = sender_ptr->p_sendlink;
07147                           if ((sender_ptr->p_flags &= ~SENDING) == 0)
07148                                   ready(sender_ptr);    /* desbloquear emisor */
07149                           return(OK);
07150                   }
07151           }
07152
07153           /* Detectar interrupción bloqueada. */
07154           if (caller_ptr->p_int_blocked && isrxhardware(src)) {
07155                   m_ptr->m_source = HARDWARE;
07156                   m_ptr->m_type = HARD_INT;
07157                   caller_ptr->p_int_blocked = FALSE;
07158                   return(OK);
07159           }
07160         }
07161
07162         /* No hay mensaje adecuado. Bloquear el proceso que intenta recibir. */
07163         caller_ptr->p_getfrom = src;
07164         caller_ptr->p_messbuf = m_ptr;
07165         if (caller_ptr->p_flags == 0) unready(caller_ptr);
07166         caller_ptr->p_flags |= RECEIVING;
07167
07168         /* Si MM acaba de bloquearse y hay señales de kernel pendientes,
07169          * es el momento de informar de ellas al MM, pues podrá aceptar el mensaje.
07170          */
07171         if (sig_procs > 0 && proc_number(caller_ptr) == MM_PROC_NR && src == ANY)
07172                   inform();
07173         return(OK);
07174       }
```

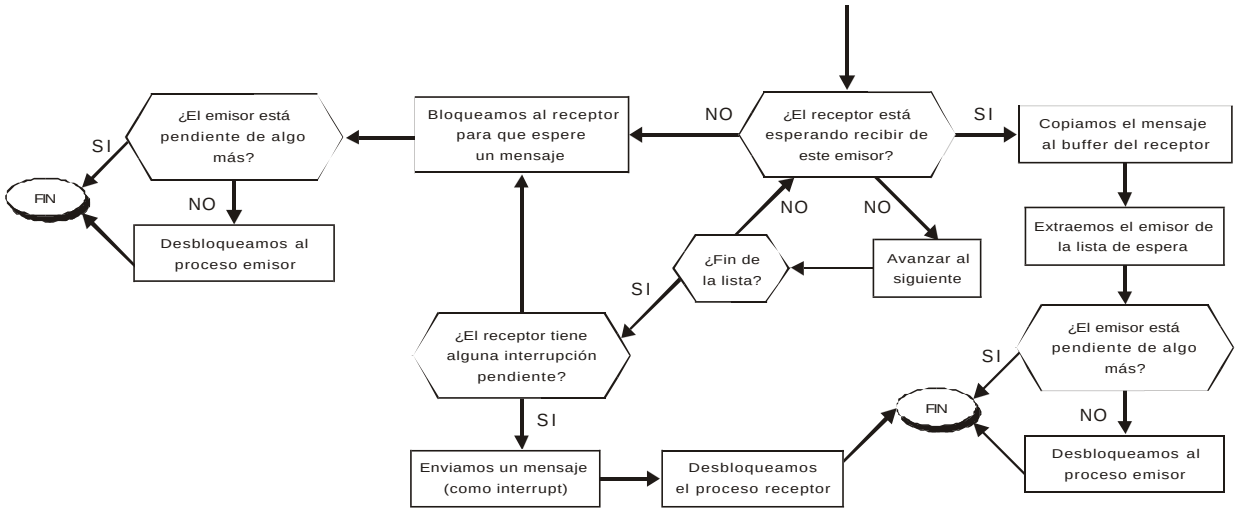


Figura 16

4.5.- Resumen

Hemos comprobado el cumplimiento de dos características básicas de MINIX funcionamiento de los dos tipos de comunicación que existe: interrupciones y llamadas al sistema. Viendo sus similitudes y diferencias. También se ha tratado de abarcar un amplio dominio sobre el que se produce el proceso de las mismas sobrepasando los límites de nuestro fichero ***proc.c***. Aunque se ha hecho para poder entender mejor esta parte del Sistema Operativo, enlazando con otros módulos que ya se han visto o se verán dentro de pocos temas.

Desde un punto de vista muy general, las dos funciones que hemos analizado recogen un aviso, lo transforman a mensaje y lo transmiten. Todo el proceso restante son comprobaciones y enlaces que se hacen para un correcto funcionamiento.

5.- Preguntas y respuestas

A parte de las preguntas que nos hemos encontrado durante este tema, aquí tenemos algunas questiones más con su correspondiente respuesta.

¿Cómo está dividida la tabla de procesos?

Tiene tres niveles separados por los números mágicos. Estos niveles se llaman TASK_Q, SERVER_Q y USER_Q. Cada uno de ellos tiene una prioridad asociada que va de mayor a menor respectivamente.

¿Qué es el Algoritmo de Planificación?

Podemos resumir que se trata de encontrar la cola con más alta prioridad que no está vacía y escoger el proceso que está a la cabeza de esa cola. Si todas las colas están vacías, se ejecuta la **rutina de "marcha en vacío" o IDLE**. En la Figura 2, TASK_Q tiene la más alta prioridad. El código de planificacion está en proc.c, en concreto se refleja en la función **pick_proc** , cuya labor podríamos resumir en escoger el siguiente proceso a ejecutar.

Ejemplo de comunicación de una interrupción hardware

En la simulación de una counicación hardware, podemos poner el siguiente caso, también comnetado en las diapositivas. Tratan de transmitir tres interrupciones a la vez. Por ejemplo, reloj, disco duro y teclado. Aquella que primero se produzca o la que tenga mayor prioridad será atendida en primer lugar. Si suponemos que no estamos tratando ninguna interrupción y la tarea del primer dispositivo este esperando una, la atenderemos.

A continuación, cuando se produzcan las otras dos, como estamos ocupados atendiendo otra, las añadimos en la cola de espera held y marcamos a esas tareas para que estén bloqueadas. En cuanto terminemos, se comprueba si dicha lista está vacía. Como no es así la tratamos de vaciar mediante la función ***unhold***, que hace un ***interrupt*** para cada tarea que está esperando, pero si, por ejemplo, la primera de la lista no estaba esperando un mensaje del dispositivo, tendremos que ir a la siguiente tarea que está esperando en la cola. De todas formas, aunque no se pueda transmitir quitamos la tarea de la cola, porque ya no pertenece a este grupo de procesos en espera. Ahora estará marcado gracias al flag ***p_int_blocked***. Si la siguiente tarea sí lo estaba

esperando, realizamos la transmisión y la quitamos de la cola. Cuando la tarea restante intente volver a realizar la transmisión y, por fin, ya está esperando un mensaje, se lo damos y quitamos su flag de bloqueo para que pueda seguir ejecutándose.

Este ejemplo, al igual que el siguiente están representados en las diapositivas de los apuntes.

Ejemplo de comunicación generada por software

Para poner un ejemplo de comunicación de este tipo, nos basaremos en un caso parecido al anterior. Suponemos los procesos A, B y C que quieren enviar al D y a su vez E quiere enviar a C. También suponemos que los tres primeros procesos intentan transmitir en el orden en que se han nombrado.

Por tanto, A intenta enviar, pero si se encuentra el caso en que D no está esperando ningún mensaje, tendrá que pasar a la cola de espera de este proceso, es decir, el campo ***p_callerq*** del proceso D apuntará a la ranura de la tabla de procesos donde se encuentra el proceso A. Posteriormente, los procesos B y C hacen lo mismo, con lo que también pasan a la lista enlazados por el campo ***p_sendlink*** de los procesos A y B, respectivamente. De repente, el proceso D la sale la necesidad de recibir un mensaje de B con lo que primeromira si hay alguien esperando por mandarle. Así, empieza a recorrer su lista y cuando llega a dicho proceso, realiza la comunicación y lo quita de la lista para que pueda seguir ejecutándose. De tal manera que ahora la cola del proceso D pasa a estar formada por los procesos A y C. En este momento, el proceso E, intenta enviar al proceso C, sin embargo, éste no está esperando nada, por lo que pasa a estar dentro de la cola de espera de este. O sea, el campo ***p_callerq*** del proceso C apunta al proceso E. En la siguiente iteración los que estaban esperando en la cola del proceso D consiguen enviar debido a que este pedía un mensaje de ellos.

Ahora bien, este caso de comunicación no ha terminado todavía, ya que resta el proceso E para enviar. Éste lo podrá hacer, como ha ocurrido anteriormente, cuando el proceso receptor pida un mensaje de él.