MTask - Multitarea para i386

Ing. H. E. Etchegoyen
Cátedra de Sistemas Operativos
ITBA

Indice

Generalidades

MTask ha sido escrito con fines didácticos para ilustrar conceptos básicos de sistemas operativos. Proporciona una base de código sobre la cual se puede experimentar y hacer fácilmente agregados y modificaciones.

Es un sistema multitarea sencillo que corre sobre hardware de PC haciendo uso de un único procesador de tipo i386 en modo protegido. También puede correr sobre otros procesadores más avanzados de la misma familia, pero no soporta sus funcionalidades adicionales ni utiliza más de un núcleo.

Para simplificar el código y disminuir en lo posible la dependencia de la arquitectura i386, no se utiliza el mecanismo de segmentación típico de estos microprocesadores. Los registros de segmento se inicializan al arrancar el sistema y no se modifican en el resto del código. Se utiliza un modelo de memoria "flat", donde tanto el segmento de código como el de datos abarcan el total de la memoria accesible al i386, es decir, desde la dirección 0 hasta 2^{32}-1.

Tampoco se utiliza el mecanismo de paginación, aunque se prevé utilizarlo en futuras ampliaciones, para aislar al sistema operativo y a las tareas o procesos entre sí y para permitir que las tareas o procesos reciban zonas generosas de memoria lineal mapeadas por demanda, sin desperdicio de memoria física.

Planificación (scheduling)

MTask tiene las siguientes características:

- Es preemptivo.
- Cada tarea puede salir del modo preemptivo y correr en modo cooperativo durante el tiempo que desee. El modo en que corre una tarea no afecta a las demás.
- Cada tarea puede deshabilitar interrupciones durante el tiempo que desee. El estado de habilitación de interrupciones de una tarea no afecta a las demás.
- Tiene ranura de tiempo (100 ms).
- Las tareas tienen prioridades estáticas que se establecen al crearlas y pueden modificarse en cualquier momento.

El algoritmo de planificación (scheduling) es el siguiente: en cada momento se ejecuta la tarea de mayor prioridad entre las que están listas para ejecutar. Si hay más de una tarea con la prioridad máxima en condiciones de ejecutar, se elige la que ha estado esperando el procesador durante más tiempo. Si dos o más tareas de máxima prioridad corren sin bloquearse ni ceder la CPU, se alternan en el uso del microprocesador mediante el mecanismo de ranura de tiempo.

Cuando una tarea corre en modo preemptivo puede perder la CPU si se bloquea, se suspende o la cede voluntariamente, o si se cumple alguna de las siguientes condiciones: otra tarea con mayor prioridad que la actual está lista para ejecutar, o hay otra tarea de la misma prioridad que la actual lista para ejecutar y se agota la ranura de tiempo de la tarea actual. Estos eventos pueden ser producidos por una interrupción, y eso le da al sistema su carácter preemptivo, es decir, una tarea puede perder la CPU como resultado de una interrupción. En cambio, si la tarea actual está corriendo en modo no preemptivo, solamente puede perder la CPU por bloquearse, suspenderse o cederla voluntariamente, pero no como resultado de una interrupción.

Por ser fijas las prioridades, aún en modo preemptivo una tarea puede monopolizar la CPU. Si una tarea de mayor prioridad que las demás no se bloquea o suspende, las demás no tienen oportunidad de ejecutar. Es responsabilidad del programador evitar esta situación, así como otras posibilidades de monopolización, por ejemplo, tareas que deshabilitan interrupciones o salen del modo preemptivo y luego no se bloquean ni se suspenden ni ceden la CPU.

Implementación y limitaciones

MTask no es un sistema operativo completo. Es un kernel que permite crear hilos de ejecución o threads, que denominamos "tareas" y sincronizarlos mediante una variedad de primitivas de IPC:

- Colas de espera
- Pasaje de mensajes sin almacenamiento intermedio ("rendez-vous")
- Semáforos
- Mutexes
- Monitores y variables de condición
- Pipes
- Colas de mensajes

MTask carece de "procesos" en el sentido que se le da habitualmente a esta palabra, es decir, instancias de programas que han sido compilados por separado, con espacios de datos separados entre sí y del kernel, y que acceden al mismo a través de "llamadas al sistema" (system calls), generalmente implementadas mediante interrupciones de software. El soporte de verdaderos procesos es una dirección de posible desarrollo futuro.

Las funciones de creación y manejo de tareas y las primitivas de IPC pueden utilizarse incluyendo el archivo de cabecera mtask.h. Estas funciones tienen nombres que empiezan con letras mayúsculas y son "thread-safe", es decir, pueden llamarse desde distintos threads sin peligro. Hay otras funciones y variables globales que son de uso interno de MTask, en general tienen nombres que empiezan con "mt_" y están todas declaradas en kernel.h. Algunas son "thread-safe", otras no. Por último, hay funciones genéricas de biblioteca, como strcmp() y printk() (el printf del kernel) que conservan los nombres habituales. También en este caso algunas son thread-safe y otras no. Por ejemplo, malloc() y free() no lo son; se recomienda usar en su lugar las funciones de alocación de memoria de la API, Malloc() y Free().

La idea es que las "aplicaciones" (threads) usen la API principal contenida en mtask.h como "llamadas al sistema". Sin embargo, en el estado de desarrollo actual de MTask la API está incompleta, básicamente por la ausencia de un file system y de una interfaz uniforme a los drivers; ésta es otra dirección de desarrollo futuro.

En esta distribución de MTask el sistema de entrada/salida comprende:

- Módulo PS2 - Provee funciones que empiezan con "mt_ps2_" para leer y establecer la distribución del teclado. Genera eventos de teclado (make/break de teclas) y mouse (desplazamientos y botones) que se inyectan en la interfaz de eventos del módulo de entrada. También procesa los eventos de teclado teniendo en cuenta la distribución del mismo y genera caracteres que inyecta en la interfaz de caracteres del módulo de entrada.
- Módulo de entrada - Provee funciones que empiezan con "mt_input_" para inyectar y leer los eventos generados por el teclado y el mouse, y otras que empiezan con "mt_kbd_" para inyectar y leer los caracteres generados por el teclado.
- Consola - Provee funciones que empiezan con "mt_cons_" para manejar el display VGA en modo texto.
- Driver de discos IDE (ATA) - Provee funciones que empiezan con "mt_ide_" para leer y escribir sectores en un disco.

Todas las funciones y símbolos públicos "mt_..." están declarados en kernel.h, que también incluye a mtask.h (la API) y a lib.h (las funciones de biblioteca). En la práctica los threads de aplicación incluyen kernel.h para acceder a todas las partes de MTask. Sin embargo, por razones de buen diseño deberían limitarse en lo posible a usar solamente la API declarada en mtask.h, las funciones de biblioteca y las de entrada/salida.

El estudio de las aplicaciones de ejemplo contenidas en la distribución ilustrarán el uso práctico de todo esto.

Distribución y sistema de desarrollo

MTask debe compilarse con versiones relativamente modernas de gcc en ambiente Linux. También se necesita el utilitario genisoimage (llamado mkisofs en algunas distribuciones de Linux) para generar una imagen de CD booteable.

El directorio principal contiene un makefile, y todo está armado para generar el sistema completo parándose en dicho directorio y ejecutando "make". Esto dará como resultado una imagen de CD (mtask.iso) que incluye GRUB (el bootloader) y nuestro kernel (mtask). Esta imagen puede ejecutarse en una PC real capaz de arrancar desde un CD, o en una máquina virtual. En particular, funciona satisfactoriamente con versiones recientes de VirtualBox. Cuando se crea la máquina virtual deberían dársele como mínimo 16 MB de memoria, para poder ejecutar una cantidad razonable de tareas simultáneas. Si se desea utilizar el driver de disco, debe crearse también un disco virtual IDE (ATA).

Para limpiar objetos y dependencias generados previamente, se puede ejecutar "make clean". Si se ejecuta "make new" es lo mismo, pero además se retocan la fecha y hora de todos los archivos fuentes y de encabezado, lo cual puede ser útil cuando la fecha y hora de la PC donde se está compilando está atrasada con respecto a los archivos.

No hace falta declarar las dependencias de los fuentes con los archivos de encabezado, pues el makefile se ha configurado para generar automáticamente los archivos de dependencia, que se guardan en el directorio "dep".

Para agregar cualquier cosa a MTask, basta copiar los fuentes correspondientes al directorio "src" o a cualquier subdirectorio arbitrario dentro del árbol de directorios que comienza en "src" y agregar los prototipos de las funciones o los símbolos públicos en general en mtask.h, kernel.h o lib.h según corresponda. Si se trata de un thread de aplicación, deberá incluirse su función de entrada xxx_main() en apps.h, y habrá que editar el fuente del shell (shell.c) para poder ejecutarlo desde la línea de comando.

Dentro de los árboles de directorios que arrancan en "src" y en "include", los archivos fuente y los de encabezado pueden residir arbitrariamente en cualquier directorio.

Los nombres de los fuentes deben ser únicos y no pueden repetirse en distintos directorios. Son fuentes todos los archivos con extensión .c o .S cuyos nombres comiencen con una letra mayúscula o minúscula, vale decir, los archivos cuyos nombres satisfagan el patrón [a-zA-Z]*.[cS]. Serán detectados automáticamente, compilados o ensamblados y los objetos respectivos linkeados en mtask. Si se desea *excluir* algún archivo .c o .S de la lista de fuentes, se le puede cambiar la extensión o anteponer al nombre un carácter que no sea una letra. Por ejemplo, un archivo llamado "mouse.c" ubicado en src/drivers/ps2/ (o en cualquier otro directorio por debajo de "src") se compilará y linkeará en mtask, pero no si se llama "_mouse.c" o "mouse.c.ignore".

Hay un archivo fuente con nombre reservado: kstart.S. Contiene el encabezado multiboot requerido por GRUB y el punto de entrada del kernel (_start) y el objeto respectivo, kstart.o, debe aparecer primero en el orden de linkeado. El makefile se encarga de esto.

Por otra parte, los directorios contenidos en el árbol de directorios que arranca en "include" serán detectados automáticamente e incluidos en el path de búsqueda de encabezados del compilador.

Se debe ejecutar "make clean" cada vez que se retire o se mueva a otro directorio un fuente o un encabezado, para borrar objetos inútiles y actualizar las dependencias.

En el desarrollo de sistemas como éste, a veces es necesario consultar la salida del compilador antes del ensamblado, para determinar ciertos detalles de bajo nivel. Por ejemplo, al escribir el comando divz se plantea la necesidad de determinar dónde está el divisor en el momento en que se produce la división por cero, para poder modificarlo por un valor distinto. Para ello, puede ejecutarse:

```
make s/divz.s
```

Examinando divz.s se determina que el divisor está en la dirección 12(%ebp), es decir, 12 bytes más arriba de donde apunta el registro EBP.

Arranque del sistema

Cuando arranca MTask, la ejecución comienza en el punto de entrada _start en kstart.S. Aquí solamente se establece un stack adecuado y se llama a la función mt_main() de kernel.c, que continúa la inicialización. Esta función hace básicamente lo siguiente:

- Inicializa la GDT e IDT (tabla de descriptores de segmentos globales y tabla de descriptores de interrupción). Al cargar la GDT se inicializan de una vez para siempre los segmentos CS, SS, DS, ES, FS y GS cubriendo al rango de direcciones de 0 a 2^{32}-1.
- Inicializa el heap. El tamaño de la memoria disponible por encima de 1 MB es detectado por el bootloader y pasado como argumento a mt_main(). Esta función detecta el fin del segmento de datos de MTask (indicado por el rótulo _end en el mapa de linkeo, mtask.map) y arma el heap ocupando la memoria libre que se extiende desde ese punto hasta el fin de la memoria disponible. El tamaño del heap es aproximadamente igual a M - _end, siendo M la memoria total de la máquina.
- Inicializa el sistema de interrupciones. Se utilizan los descriptores 0 a 31 de la IDT para las excepciones generadas por el i386, y los descriptores 32 a 47 para las interrupciones de hardware 0 a 15 respectivamente.
- Configura el timer para interrumpir cada 10 milisegundos (100 Hz) y captura su interrupción.
- Inicializa el sistema de manejo de contexto del coprocesador aritmético (i387).
- Inicializa la tarea inicial, crea y pone a ejecutar la tarea nula.
- Habilita interrupciones.
- Calibra un lazo de software que se utiliza para generar pequeñas demoras en "busy waiting" (función UDelay()).
- Inicializa los drivers. Al inicializar el driver de consola se crean 9 consolas virtuales, y al inicializar el driver de entrada 9 canales de entrada virtuales.
- Dispara 7 tareas que ejecutan cada una un shell en un lazo sobre las consolas virtuales 2 a 8, a las que se accede mediante las combinaciones de teclas ALT-F2 a ALT-F8 respectivamente.
- Ejecuta un shell en un lazo sobre la consola 1, a la cual se accede mediante la combinación de teclas ALT-F1.

Las tareas internas creadas por los drivers y la tarea nula corren en la consola 0, la consola inicialmente activa durante el arranque, a la cual se accede mediante la combinación de teclas ALT-ESC. Los mensajes iniciales emitidos durante el arranque del sistema y los mensajes de depuración emitidos mediante el uso de la función print0(), especialmente indicada para las rutinas de interrupción, también pueden verse en la consola 0.

Comandos y aplicaciones

Una vez obtenido el prompt del shell, ejecutar el comando "help" para ver una lista de los comandos disponibles. En la distribución actual son los siguientes:

Comandos internos:

- **help** - Muestra la lista de comandos disponibles
- **exit [status]** - Sale del shell con el status de salida que se pasa, o 0 si no se pasa ninguno. Si se sale de alguno de los 8 shells iniciales (creados en la función mt_main() del kernel) MTask lo ejecuta nuevamente.
- **reboot** - Reinicia el sistema activando la línea de reset de la PC.

Aplicaciones:

- **shell** - Intérprete de comandos. Lee líneas de comando y ejecuta comandos internos o aplicaciones. Cuando ejecuta una aplicación, normalmente espera a que ésta termine y, si su

status de salida es distinto de cero, lo imprime. Si la línea de comando termina en '&' la aplicación ejecuta en background; el shell imprime la dirección del bloque de control de la tarea y lee inmediatamente la siguiente línea de comando.

- **setkb [distribución]** - Sin argumentos, muestra la distribución actual del teclado y las distribuciones disponibles. Si se pasa un argumento, establece esa distribución. Las distribuciones disponibles actualmente son "spanish" y "us-std".
- **sfilo**, **filo**, **xfilo, afilo** - Cuatro soluciones al problema de los 5 filósofos, las tres primeras utilizando un monitor. Sfilo es el más simple, usa una sola variable de condición. Filo usa una variable de condición por filósofo. Esto reduce (pero no elimina por completo) la probabilidad de que un filósofo necesite bloquearse más de una vez mientras espera sus tenedores. Xfilo agrega un array de estado de los filósofos para garantizar que haya como máximo un bloqueo mientras se esperan los tenedores. Afilo no usa monitor ni variables de condición, consigue el mismo resultado utilizando Atomic() y Unatomic() para delimitar las zonas críticas y Pause() y Ready() para suspender y despertar a los filósofos. Primitivas de IPC utilizadas: monitor y variables de condición, pasaje de mensajes.
- **camino**, **camino_ns [cantidad]** - Dos soluciones al problema de manejar un camino con un tramo de una sola vía evitando deadlocks. La primera solución, más simple, presenta probabilidad de inanición, es decir, mientras haya autos circulando en un sentido un auto circulando en sentido contrario puede esperar indefinidamente. La segunda carece de inanición ("ns" es por "no starvation"). Un auto circulando en un sentido no tendrá que esperar más tiempo que el necesario para que pasen 10 autos en sentido opuesto. El número 10 puede modificarse pasando un argumento a camino_ns. Primitivas de IPC utilizadas: semáforos (camino), pasaje de mensajes (camino_ns).
- **prodcons** - Ilustración de un sistema productor/consumidor. Dispara 4 tareas: un productor, un consumidor, una tarea que muestra el estado del productor y el consumidor en cada instante, y un reloj que muestra los segundos transcurridos desde que se inició la ejecución. Primitivas de IPC utilizadas: semáforos.
- **divz dividendo divisor** - Ilustra la captura de la excepción 0 (división por cero) y cómo puede utilizarse la estructura de registros empujada al stack por la excepción para solucionar el problema producido por un divisor nulo.
- **pelu** - Una solución al problema de la peluquería (Tanenbaum). Se simula una peluquería con 5 sillas para que los clientes esperen y 3 peluqueros. Primitivas de IPC utilizadas: monitor y variables de condición.
- **events** - Lee y muestra los eventos de teclado (make y break de teclas) y de mouse (desplazamiento y botones).
- **disk** - Intenta detectar los 4 discos que soporta el driver IDE. En el caso de los discos ATA, muestra su capacidad y realiza una prueba de lectura/escritura. Lee los primeros 128 sectores (64 KB) y los vuelve a escribir 16 veces (1 MB) para medir el tiempo de escritura.
- **ts [consola...]** - Muestra el estado de las tareas. Si no se pasa ningún argumento, muestra todas las tareas. Opcionalmente pueden pasarse uno o más números de consola; en ese caso solamente se mostrarán las tareas de las consolas indicadas. Los nombres de las tareas protegidas se muestran entre corchetes.
- **kill tarea [status]** - Intenta terminar una tarea (cuya dirección se pasa como primer argumento en formato hexadecimal) con el status de salida opcionalmente indicado, o 0 si no se pasa. Las direcciones de las tareas pueden averiguarse mirando la salida de ts.
- **test** - Cáscara vacía para hacer pruebas rápidas. Sólo imprime "Hola mundo".

Algunos detalles de implementación

El manejo de tareas se basa en el uso de colas ordenadas. Hay una cola global de tareas listas para ejecutar y otra de tareas que están esperando que transcurra el tiempo. También hay una cola de tareas terminadas. La tarea nula y las funciones que alocan memoria recorren esta cola, liberan los recursos de las tareas terminadas y la vacían.

Todos los objetos bloqueantes (como semáforos y variables de condición) y la transmisión de mensajes de una tarea a otra usan colas de espera donde las tareas pueden bloquearse.

Salvo la cola de tiempo, las demás colas están ordenadas en forma creciente por prioridad y, entre tareas de la misma prioridad, por el tiempo que llevan esperando. El planificador (scheduler) elige siempre la última tarea de la cola de tareas listas para ejecutar; será la de máxima prioridad o, entre dos o más con la máxima prioridad, la que lleva más tiempo esperando. Las señalizaciones de semáforos o variables de condición despertarán siempre a la última tarea de la cola asociada.

La cola de tiempo está ordenada en forma creciente por el valor del campo "ticks" en el bloque de control de cada tarea. Este valor representa la cantidad de ticks de timer que le faltan a la tarea para despertarse, medida en forma incremental con respecto a la tarea anterior. La interrupción de timer decrementa el campo "ticks" de la primera tarea en la cola de tiempo, si llega a cero la despierta y eventualmente a las que vengan detrás que también tengan 0 en este campo. Esta cola se usa cuando las tareas se bloquean esperando el transcurso de una cierta cantidad de tiempo, o cuando se bloquean por otros motivos pero con timeout.

En todas las funciones bloqueantes que acepten un timeout, FOREVER (2^{32}-1) significa "infinito".

Estados de una tarea

```
typedef enum
{
      TaskSuspended,
      TaskReady,
      TaskCurrent,
      TaskDelaying,
      TaskWaiting,
      TaskSending,
      TaskReceiving,
      TaskJoining,
      TaskZombie,
      TaskTerminated
}
TaskState_t;
```

- **TaskSuspended** - La tarea está suspendida y no puede ejecutar. No está en ninguna cola. Las tareas se crean en este estado, para que empiecen a ejecutar hay que pasárselas a la función Ready().
- **TaskReady** - La tarea está en condiciones de ejecutar. Está en la cola global de tareas listas para ejecutar.
- **TaskCurrent** - La tarea está utilizando la CPU. No está en ninguna cola. Por ser un sistema de un solo núcleo, solamente hay una tarea en este estado.
- **TaskDelaying** - La tarea está bloqueada esperando que transcurra el tiempo necesario para despertarse. Está en la cola de tiempo.
- **TaskWaiting** - La tarea está bloqueada esperando en una cola de tareas, probablemente asociada a algún semáforo o variable de condición. Si la espera es con timeout, está también en la cola de tiempo.
- **TaskSending** - La tarea está bloqueada esperando transmitirle un mensaje a otra. Está insertada en una cola de tareas asociada a la tarea receptora del mensaje. Si la transmisión es con timeout, también está en la cola de tiempo.
- **TaskReceiving** - La tarea está bloqueada esperando recibir un mensaje. Si la recepción es con timeout, está en la cola de tiempo.
- **TaskJoining** - La tarea está bloqueada esperando que termine otra tarea vinculada a ella. Si la espera es con timeout, está en la cola de tiempo.
- **TaskZombie** - La tarea ha terminado de ejecutar y está vinculada a otra que todavía no ha recogido su status de salida.
- **TaskTerminated** - La tarea está terminada. Está en la cola de tareas terminadas.

API principal

Creación y manejo de tareas

```
typedef int (*TaskFunc_t)(void *arg);
Task_t *CreateTask(TaskFunc_t func, unsigned stacksize, void *arg,
      const char *name, unsigned priority);
```

Crea una tarea y devuelve un puntero a su bloque de control. El parámetro stacksize es el tamaño del stack; el tamaño mínimo es 4 KB y si se pasa un valor menor se aloca el mínimo. El argumento arg se pasa a la función que implementa la tarea. El parámetro name es para darle un nombre, es opcional y puede pasarse NULL. La prioridad mínima es 0 y la máxima 2^{32}-1. La tarea creada está suspendida (TaskSuspended), tiene las interrupciones habilitadas y corre en modo preemptivo. Para que comience a ejecutar hay que aplicarle la función Ready(). Terminará de ejecutar cuando retorne del cuerpo de su función o ejecute Exit(). El valor de retorno de la función de una tarea, o el argumento que esta tarea pasa a Exit(), es el status de salida de la tarea. Solamente resulta útil en el caso de tareas vinculadas, donde será recogido por la función Join() ejecutada por la tarea vinculante.

```
bool DeleteTask(Task_t *task, int status);
```

Termina la ejecución de una tarea con el status indicado, como si la tarea hubiera retornado de su función o hubiera ejecutado Exit(status). Si se aplica a la tarea actual ejecuta inmediatamente Exit(status). En caso contrario, modifica el contexto de la tarea para que ejecute Exit(status) en modo preemptivo y con interrupciones habilitadas cuando vuelva a recibir el procesador. Si la tarea está bloqueada (por ejemplo esperando en una cola de tareas) la despierta para ponerla de inmediato en condiciones de ejecutar. Una tarea a la que se le aplique DeleteTask() no termina en forma inmediata, sino cuando vuelve a recibir el procesador. DeleteTask() fracasa si se aplica sobre una tarea que ya está ejecutando Exit() o si una tarea no protegida intenta aplicarla sobre una protegida.

```
bool Protect(Task_t *task);
```

Protege una tarea de modo que no puede terminase aplicándole DeleteTask(). Protect() fracasa si la tarea actual no está protegida.

```
bool Attach(Task_t *task);
bool Detach(Task_t *task);
```

Por defecto, en MTask las tareas corren sin ninguna vinculación con las demás. En esas condiciones, cuando terminan eventualmente efectúan una operación final de limpieza (cleanup) y desaparecen. La función Attach() permite vincular una tarea a la tarea actual. Esto cambia el comportamiento de la tarea en el momento de terminar: antes de desaparecer, la tarea debe sincronizarse con aquélla a la cual está vinculada. Del lado de la tarea a la cual está vinculada, la sincronización se realiza mediante la función Join(). El mecanismo es el siguiente:

```
Task_t *t = CreateTask(...);
Attach(t);
Ready(t);
```

En este caso t corre vinculada a la tarea actual. Para esperar a que t termine, la tarea actual debe ejecutar:

```
int status;
bool success = Join(t, &status);
```

Si t todavía no terminó cuando se ejecuta Join(t, &status), la tarea actual se bloquea (en el estado TaskJoining) hasta que t termine. Si en cambio t termina antes de que se ejecute Join(t, &status), t se bloquea (en el estado TaskZombie) hasta que la tarea actual ejecute el Join() o termine. Si la tarea actual termina sin ejecutar Join(t, &status), t queda desvinculada. Si estaba esperando el Join() (en el estado TaskZombie) se desbloquea y desaparece. Cuando Join(t, &status) termina exitosamente, status contiene el status de salida de t (es decir, el valor de retorno de la función de t o el valor pasado a Exit(status) por t, o el segundo argumento de DeleteTask(t, status). Attach()

fracasa si la tarea ya está vinculada, si está ejecutando Exit() o si una tarea no protegida intenta vincular a una protegida.

Detach() desvincula a una tarea previamente vinculada a la tarea actual. Si la tarea está en estado TaskZombie se desbloquea y termina inmediatamente. Detach() fracasa si la tarea no está vinculada a la tarea actual.

```
bool SetPriority(Task_t *task, unsigned priority);
```

Establecer la prioridad de una tarea. Fracasa si la tarea actual no está protegida e intenta aplicársela a una tarea protegida.

```
bool SetConsole(Task_t *task, unsigned consnum);
```

Establecer el número de consola donde corre una tarea. Cada tarea hereda el número de consola de la tarea que la creó. Esta función fracasa si una tarea no protegida intenta aplicársela a una tarea protegida.

```
typedef void (*SaveRestore_t)(void);
bool SetSaveRestore(Task_t *task, SaveRestore_t save,
      SaveRestore_t restore);
```

Asigna a una tarea un par de funciones de callback. Save() se llama en cada cambio de contexto inmediatamente antes de que la tarea pierda el procesador, y restore() inmediatamente después de que lo recupera. La idea es que las tareas puedan utilizar estos callbacks para guardar y reponer un contexto adicional al que guarda y repone MTask (registros del procesador y estado del coprocesador aritmético). El contexto en el que ejecutan es bastante restringido. La función CurrentTask() retorna la tarea actual y el TLS vigente es el que corresponde a dicha tarea.

El stack en uso puede no ser el propio de la tarea y puede tener el tamaño mínimo. Las interrupciones están deshabilitadas y no deben habilitarse en ningún momento. Tampoco puede ejecutarse ninguna función que directa o indirectamente pueda llevar a un cambio de contexto. En principio estos callbacks deberían limitarse a copiar información. Puede utilizarse la función printk() para emitir mensajes de depuración.

```
typedef void (*Cleanup_t)(void);
bool SetCleanup(Task_t *task, Cleanup_t cleanup);
```

Establecer un callback que se llamará cuando la tarea ejecute Exit(). Fracasa si una tarea no protegida intenta aplicársela a una tarea protegida.

Sirve para realizar tareas de limpieza antes de que una tarea termine. Cuando se ejecuta el callback el contexto es el de la tarea que está terminando, incluyendo el stack y el TLS, y no hay restricciones en las funciones que pueden ejecutarse. Si Exit() se ejecuta voluntariamente, ya sea llamándolo directamente o retornando de la función de la tarea, el callback ejecuta con el modo y el estado de habilitación de interrupciones que tiene la tarea en el momento de llamar a Exit(). Si Exit() ejecuta involuntariamente, porque la tarea fue terminada por un llamado a DeleteTask() por parte de otra tarea, el callback ejecuta en modo preemptivo y con interrupciones habilitadas, con independencia del estado previo de la tarea. En ambos casos, cuando el callback retorna la tarea deja de correr y ya no recupera el procesador.

Las funciones de cleanup no deben llamar directa o indirectamente a Exit(), pues esto llevaría a llamados recursivos. El sistema detecta esta situación y genera un Panic().

```
typedef struct
{
      Task_t *task;
      unsigned consnum;
      unsigned priority;
      TaskState_t state;
      void *waiting;
      bool is_timeout;
      unsigned timeout;
}
TaskInfo_t;

void GetInfo(Task_t *task, TaskInfo_t *info);
```

Retorna información del estado de una tarea en una estructura de tipo TaskInfo_t. El significado de cada campo de esta estructura es el siguiente:

- **task** - Puntero al bloque de control de la tarea.
- **consnum** - Número de consola.
- **priority** - Prioridad.
- **state** - Estado.
- **waiting** - Puntero al objeto sobre el cual se espera, de acuerdo al estado de la tarea. TaskWaiting: cola de tareas donde se está esperando; TaskSending: tarea destinataria del mensaje; TaskReceiving: tarea de la cual se espera recibir; TaskJoining: tarea vinculada; TaskZombie: tarea a la cual está vinculada. En los demás estados, NULL.
- **is_timeout** - Indica si la tarea está en la cola de tiempo.
- **timeout** - Si la tarea está en la cola de tiempo, tiempo para despertarse en milisegundos.

```
TaskInfo_t *GetTasks(unsigned *ntasks);
```

Esta función recorre la lista de todas las tareas no terminadas y aloca dinámicamente un array de estructuras TaskInfo_t que llena con la información de cada tarea. En la variable apuntada por ntasks escribe la cantidad total de tareas y devuelve el array de estructuras que, después de ser utilizado, deberá liberarse llamando a Free().

```
bool Ready(Task_t *task);
```

Pone una tarea en estado de ejecutar (TaskReady) y la coloca en la cola respectiva. Si la tarea estaba previamente bloqueada la desbloquea; si estaba en una función bloqueante, como esperar en una cola de tareas o intentar transmitir o recibir un mensaje, la función bloqueante fracasa.

```
bool Suspend(Task_t *task);
```

Suspende una tarea, sin importar su estado previo, y la saca de las colas en las que pueda estar bloqueada. Fracasa si una tarea no protegida intenta aplicársela a una tarea protegida. Una tarea suspendida solamente vuelve a ejecutar aplicándole Ready() o DeleteTask().

Funciones que afectan a la tarea actual

```
Task_t *CurrentTask(void);
```

Retorna la tarea actual.

```
void Pause(void);
```

Suspende la tarea actual. Es equivalente a Suspend(CurrentTask()).

```
void Yield(void);
```

Esta función equivale a Ready(CurrentTask()). Da al sistema la oportunidad de quitar la CPU a la tarea actual si se cumplen las condiciones para ello, vale decir, hay una tarea de la misma o mayor prioridad en condiciones de ejecutar. Es conveniente usarla en situaciones de "busy waiting", por ejemplo:

```
while ( !condition )
      Yield();
```

Una tarea que corre en modo no preemptivo y no necesita bloquearse o suspenderse debería llamar a esta función con razonable frecuencia para no monopolizar el uso de la CPU. De todos modos, hay que tener cuidado. Esta llamada no da ninguna chance de correr a tareas de menor prioridad que la actual. Esto puede llevar a una situación de deadlock, si una de las tareas que no pueden correr es justamente la encargada de hacer cumplir la condición que se está esperando. Este fenómeno se conoce como "inversión de prioridad". Para evitar el bloqueo de tareas de menor prioridad, si se puede tolerar cierto tiempo de latencia en la verificación de la condición, en el lazo de espera conviene reemplazar Yield() por una llamada a la función Delay() con un intervalo adecuado.

```
void Delay(unsigned msecs);
```

Dormir una cierta cantidad de tiempo. La tarea pasa al estado TaskDelaying y se coloca en la cola de tiempo.

```
void UDelay(unsigned usecs);
```

Dormir una pequeña cantidad de tiempo, expresada en microsegundos. La tarea no cambia de estado, la espera se realiza por "busy waiting". Esta función debe usarse con mucho cuidado, está pensada para drivers que necesitan hacer pequeñas demoras (del orden de microsegundos o unos pocos milisegundos) para las cuales es excesiva la granularidad ofrecida por la función Delay(), que en la distribución actual duerme en múltiplos de 10 milisegundos. Se utiliza en el driver de discos IDE, que necesita realizar algunas demoras del orden de 1 microsegundo.

```
bool Join(Task_t *task, int *status);
bool JoinCond(Task_t *task, int *status);
bool JoinTimed(Task_t *task, int *status, unsigned msecs);
```

Esperar la terminación de una tarea vinculada, con espera indefinida, condicional (timeout 0) o con un timeout finito. Mientras dura la espera la tarea permanece en el estado TaskJoining. Cuando retornan exitosamente, estas funciones dejan en *status el status de salida de la tarea (valor de retorno de la función de la tarea, o argumento pasado a Exit() por la tarea, o segundo argumento de DeleteTask() si se le aplicó a la tarea para hacerla terminar).

```
void Exit(int status);
```

Terminar la tarea actual. Puede ser ejecutada voluntariamente, ya sea por un llamado explícito o retornando de la función que implementa el cuerpo de la tarea. En este caso ejecuta en el estado en que está la tarea. También puede ejecutar involuntariamente, si la tarea actual ha sido señalizada desde otra tarea mediante DeleteTask(). En ese caso, ejecuta con interrupciones habilitadas y en modo preemptivo. Ejecuta el callback de limpieza (cleanup) si está instalado. Si la tarea está vinculada a otra y la otra está esperando a ésta en un llamado a Join(), le pasa el status de salida. En caso contrario, se bloquea en el estado TaskZombie hasta que la otra ejecute el Join() necesario para recibir el status. Luego pasa la tarea al estado TaskTerminated y la coloca en la cola de tareas terminadas, para que sus recursos sean posteriormente liberados.

```
void Panic(const char *format, ...);
```

Error fatal. Imprime un mensaje en consola y detiene el sistema.

```
void Atomic(void);
void Unatomic(void);
```

Funciones para entrar y salir del modo no preemptivo o cooperativo. Atomic() incrementa una variable en el bloque de control de la tarea actual, y Unatomic() la decrementa. Cuando la variable

es cero la tarea corre en modo preemptivo; cuando es distinta de cero, en modo cooperativo. Cuando la tarea está en modo cooperativo, solamente puede perder la CPU si se bloquea, suspende o llama a Yield(). Correr en modo cooperativo protege una tarea contra el uso concurrente de recursos globales compartidos por parte de otras tareas, pero no contra la ejecución de rutinas de interrupción que pudieran acceder a los mismos recursos. Para ello deben deshabilitarse las interrupciones.

```
bool SetInts(bool enabled);
```

Permite habilitar o deshabilitar interrupciones en la tarea actual, retornando el valor anterior de la habilitación para poder reponerlo posteriormente. Deshabilitar interrupciones ejecutando SetInts(false) proporciona un grado de protección superior a correr en modo cooperativo, porque protege contra el acceso concurrente a recursos globales compartidos no sólo por parte de otras tareas sino también de rutinas de interrupción. Es importante no deshabilitar interrupciones durante un tiempo excesivo sin ceder el procesador. Es lícito bloquearse con las interrupciones deshabilitadas, pues el estado de habilitación de interrupciones es parte del contexto de cada tarea: al producirse el cambio de contexto, las interrupciones quedarán habilitadas o no según el contexto de la tarea que recibe el procesador. Una tarea que corre con interrupciones deshabilitadas está al mismo tiempo en modo no preemptivo, pues al no haber interrupciones éstas no pueden quitarle la CPU.

El valor que devuelve SetInts() puede usarse luego para reponer la situación anterior. En general, se recomienda que las zonas de código que necesitan correr con interrupciones deshabilitadas se definan así:

```
bool ints = SetInts(false);         // deshabilita
...                                 // código que corre
...                                 // con interrupciones
...                                 // deshabilitadas
SetInts(ints);                      // repone
```

De esta manera, las zonas de deshabilitación de interrupciones pueden anidarse sin riesgo de habilitar indebidamente al finalizar. Muchas funciones de la API definen zonas de este tipo. Esto permite que, por ejemplo, una rutina de interrupción que está corriendo con interrupciones deshabilitadas pueda señalizar un semáforo sin riesgo de habilitar.

```
extern void *TLS;
#define TLS(type) ((type *)TLS)
```

MTask soporta el concepto de datos seudo-globales locales a una tarea (TLS - Thread Local Storage) mediante el uso de un puntero global (TLS) cuyo valor se guarda y repone en cada cambio de contexto como parte del contexto de la tarea. Cada tarea hereda el valor de TLS de la tarea que la creó; posteriormente puede modificarlo a voluntad asignándole la dirección de una estructura que contiene sus datos locales. El macro TLS(type) permite castear TLS a un puntero al tipo de datos adecuado para cada tarea.

Colas de tareas

Estas colas alojan tareas que se bloquean esperando alguna condición. Son las primitivas básicas que permiten que una tarea se bloquee y se despierte, y como tales son la base de artefactos de IPC como semáforos, mutexes, monitores, variables de condición, etc.

```
TaskQueue_t *CreateQueue(char *name);
void DeleteQueue(TaskQueue_t *queue);
bool WaitQueue(TaskQueue_t *queue);
bool WaitQueueTimed(TaskQueue_t *queue, unsigned msecs);
Task_t *SignalQueue(TaskQueue_t *queue);
void FlushQueue(TaskQueue_t *queue, bool success);
```

CreateQueue() aloca una cola y DeleteQueue() la destruye, despertando con status de error a las tareas que estuvieran esperando en ella.

WaitQueue() y WaitQueueTimed() bloquean una tarea (poniéndola en el estado TaskWaiting) y la insertan en la cola a esperar que la despierten, sin y con timeout respectivamente. Son funciones booleanas que retornan true si la espera termina con éxito, porque la tarea fue despertada por SignalQueue(), y false en caso contrario, por ejemplo, porque la tarea fue despertada por Ready() o porque se agotó el timeout.

SignalQueue() despierta a la última tarea de la cola y devuelve un puntero a la misma, o retorna NULL si la cola estaba vacía y no se despertó a ninguna tarea. Esto último es lo que llamamos un "wakeup perdido" - señalizar una cola en la que no hay ninguna tarea esperando. El valor de retorno de SignalQueue() permite detectar esta situación.

FlushQueue() vacía la cola despertando a todas las tareas que estén esperando en ella, y pasando el valor del parámetro sucess como status de salida de sus respectivas llamadas a WaitQueue() o WaitQueueTimed().

Pasaje de mensajes

MTask implementa un sistema de pasaje de mensajes del tipo "rendez-vous" (cita, en francés), es decir, sin almacenamiento intermedio. En este sistema hay dos funciones básicas, Send() y Receive(), para enviar y recibir un mensaje respectivamente. La tarea que ejecuta una de estas primitivas se bloquea hasta que la contraparte ejecuta la primitiva complementaria. En ese momento se transfiere el mensaje directamente del emisor al receptor y se desbloquea a quien estaba bloqueado.

Hay variantes de Send() y Receive() con timeout y otras condicionales (timeout 0).

```
bool Send(Task_t *to, void *msg, unsigned size);
bool SendCond(Task_t *to, void *msg, unsigned size);
bool SendTimed(Task_t *to, void *msg, unsigned size, unsigned msecs);
bool Receive(Task_t **from, void *msg, unsigned *size);
bool ReceiveCond(Task_t **from, void *msg, unsigned *size);
bool ReceiveTimed(Task_t **from, void *msg, unsigned *size,
      unsigned msecs);
```

El primer argumento de Send() es la tarea destinataria, luego vienen un buffer conteniendo el mensaje y su tamaño. Los últimos dos argumentos son opcionales, puede pasarse NULL y 0. En este caso no se pasa ningún mensaje y solamente se obtiene una sincronización entre las dos tareas.

Receive() es parecida a Send() con las siguientes diferencias: el parámetro from no es un puntero a una tarea sino un puntero a puntero, y el parámetro size no es un entero sino un puntero a entero. Size es un puntero a una variable entera que debe inicializarse con el tamaño del buffer destinado a almacenar el mensaje recibido (msg). Cuando Receive() retorne, la variable contendrá el tamaño real del mensaje, que será igual o menor al valor inicial de la variable. Si el mensaje no cabe en el buffer, se produce un error fatal. El uso del parámetro from es el siguiente: si se pasa el valor NULL, se interpreta que la tarea está dispuesta a recibir un mensaje de cualquier remitente, y no le interesa saber quién se lo envía. En caso contrario, debe pasarse la dirección de una variable del tipo "puntero a tarea" (Task_t *). Si esta variable está inicializada en NULL, significa que la tarea está dispuesta a recibir un mensaje de cualquier otra, pero desea saber quién se lo envía. En ese caso, cuando Receive() retorne la variable contendrá un puntero al remitente. En cambio, si la variable se inicializa con un puntero a una tarea determinada, significa que Receive() sólo recibirá un mensaje de esa tarea y de ninguna otra.

Nombres de los objetos

```
char *GetName(void *object);
```

Todos los objetos de MTask (tareas, colas de tareas, semáforos, etc.) tienen un nombre opcional, que se pasa como un argumento de la función que los crea. Si no se desea utilizar este nombre puede pasarse NULL a la función de creación. El nombre puede resultar útil para mensajes de depuración y para los errores fatales (Panic). Se guarda una copia alocada dinámicamente, de modo que el string original puede reutilizarse para otros fines después de pasárselo a la función de

creación. GetName() devuelve el nombre de un objeto de cualquier tipo, siempre que en su construcción se respete la convención de colocar el puntero al nombre como primer campo de la estructura que representa el objeto (es decir, la dirección del puntero es la misma que la dirección del objeto que lo contiene).

Medición de tiempo

```
typedef unsigned long long Time_t;
Time_t Time(void);
```

Devuelve la cantidad de milisegundos transcurridos desde el arranque del sistema, con una granularidad de 10 ms determinada por la frecuencia de las interrupciones de timer (100 Hz).

Manejo de memoria dinámica

```
void *Malloc(unsigned size);
char *StrDup(const char *str);
void Free(void *mem);
```

Estas funciones son similares a sus equivalentes con minúscula de la biblioteca standard de C, pero son thread-safe, es decir, pueden llamarse simultáneamente desde distintas tareas (pero no desde manejadores de interrupción). Además Malloc() inicializa la memoria alocada con ceros, y StrDup() y Free() toleran punteros nulos. Estas funciones se utilizan internamente para realizar todas las alocaciones de memoria necesarias para la creación de tareas y objetos diversos. Antes de alocar memoria, Malloc() y StrDup() recorren la cola de tareas terminadas y la vacían, liberando la memoria utilizada.

TLS - Ejemplo de uso

El manejo de variables seudo-globales (Thread Local Storage - TLS) está soportado por el puntero global TLS y el macro TLS(type). En este ejemplo una tarea asigna la dirección de una estructura a TLS y utiliza sus miembros como variables seudo-globales.

```
typedef struct
{
      unsigned num;
      char *string
}
data;

#define num       TLS(data)->num
#define string    TLS(data)->string

...
TLS = Malloc(sizeof(data));
...
num = 3;
...
Free(TLS);
...
```

El macro TLS(type) permite transformar TLS en un puntero a la estructura utilizada. Los #defines permiten acceder a los miembros de la estructura como si fueran variables; la asignación "num = 3" se resuelve como "((data *)TLS)->num = 3". Si se usa este truco para facilitar la programación, se debe tener la precaución de no usar los mismos nombres para otras variables o campos de estructuras.

API de IPC

Semáforos

El semáforo tiene una cola de tareas donde bloquearse, y un contador de las veces que se intentó despertar una tarea pero la cola estaba vacía, es decir, un contador de wakeups perdidos. Cuando el valor del contador es mayor que cero, la cola está vacía. Cuando el valor es igual a cero, puede haber tareas esperando en la cola.

```
Semaphore_t *CreateSem(char *name, unsigned value);
void DeleteSem(Semaphore_t *sem);
```

Crear un semáforo con un valor inicial y darlo de baja.

```
bool WaitSem(Semaphore_t *sem);
bool WaitSemCond(Semaphore_t *sem);
bool WaitSemTimed(Semaphore_t *sem, unsigned msecs);
```

Funciones para esperar en un semáforo, sin timeout, con timeout o condicionalmente (timeout 0). Si el valor del contador es mayor que cero lo decrementan y retornan exitosamente. En caso contrario, la tarea se bloquea en la cola de espera. Retornan true si la espera es exitosa o false si termina por cualquier otro motivo (por ejemplo timeout).

```
void SignalSem(Semaphore_t *sem);
```

Señalizar el semáforo. Si hay tareas en la cola despierta a la última, en caso contrario (wakeup perdido) incrementa el contador. Esta función y las dos que siguen pueden ser utilizadas por rutinas de interrupción, pues no son bloqueantes ni alocan o liberan memoria.

```
unsigned ValueSem(Semaphore_t *sem);
```

Retorna el valor actual del contador.

```
void FlushSem(Semaphore_t *sem, bool wait_ok);
```

Despierta a todas las tareas que estén en la cola del semáforo con el status de salida wait_ok y deja el valor del semáforo en cero.

Mutexes

El mutex es un dispositivo de exclusión mutua. En su lugar puede usarse un semáforo inicializado en 1, pero el mutex tiene la ventaja de permitir la ocupación recursiva y realizar ciertas verificaciones de seguridad que evitan errores lógicos, como salir de un mutex sin haber entrado previamente en el mismo. Lleva cuenta del dueño (la tarea que lo está ocupando) y el nivel de anidamiento de la ocupación. Las funciones de mutex no están pensadas para ser utilizadas por interrupciones.

```
Mutex_t *CreateMutex(char *name);
void DeleteMutex(Mutex_t *mut);
```

Creación y destrucción.

```
bool EnterMutex(Mutex_t *mut);
bool EnterMutexCond(Mutex_t *mut);
bool EnterMutexTimed(Mutex_t *mut, unsigned msecs);
void LeaveMutex(Mutex_t *mut);
```

Entrar en un mutex (con sus tres variantes: sin timeout, con timeout y condicional) y salir de él. Debe salirse del mutex tantas veces como se entró para que éste quede libre para ser ocupado por otra tarea.

Monitores y variables de condición

```
Monitor_t *CreateMonitor(char *name);
void DeleteMonitor(Monitor_t *mon);
bool EnterMonitor(Monitor_t *mon);
bool EnterMonitorCond(Monitor_t *mon);
bool EnterMonitorTimed(Monitor_t *mon, unsigned msecs);
void LeaveMonitor(Monitor_t *mon);
```

El monitor es un mutex no recursivo. Las funciones son similares a las del mutex y no son aptas para ser utilizadas en rutinas de interrupción.

```
Condition_t *CreateCondition(char *name, Monitor_t *mon);
void DeleteCondition(Condition_t *cond);
```

Una variable de condición contiene una cola de tareas asociada a un monitor, donde las tareas pueden bloquearse esperando que se cumpla una determinada condición.

```
bool WaitCondition(Condition_t *cond);
bool WaitConditionTimed(Condition_t *cond, unsigned msecs);
```

Funciones para esperar en la cola de tareas, sin y con timeout. La tarea que las use debe poseer el monitor asociado. Al llamar a WaitCondition...() ocurre **atómicamente** lo siguiente:

- La tarea sale del mutex.
- La tarea se bloquea en la cola de tareas de la variable de condición.

El carácter atómico impide los wakeups perdidos. Cuando la tarea despierta, vuelve a ingresar al mutex y la función WaitCondition...() retorna con el status de la espera.

```
bool SignalCondition(Condition_t *cond);
void BroadcastCondition(Condition_t *cond);
```

Estas funciones señalizan la cola de tareas de la variable de condición, despertando a la última tarea o a todas. La tarea que las use debe poseer el monitor asociado. SignalCondition() returna true si despertó una tarea y false si la cola estaba vacía.

Pipes

Mecanismo de comunicación con buffer intermedio similar al de Unix. En su implementación se han utilizado un monitor, un par de variables de condición para la sincronización y un buffer circular para almacenar los datos. Las funciones de pipes no pueden llamarse desde rutinas de interrupción, porque utilizan un monitor. Ver, en cambio, las colas de mensajes.

```
Pipe_t *CreatePipe(char *name, unsigned size);
void DeletePipe(Pipe_t *p);
```

Creación y destrucción.

```
unsigned GetPipe(Pipe_t *p, void *data, unsigned size);
unsigned GetPipeCond(Pipe_t *p, void *data, unsigned size);
unsigned GetPipeTimed(Pipe_t *p, void *data, unsigned size,
      unsigned msecs);
unsigned PutPipe(Pipe_t *p, void *data, unsigned size);
unsigned PutPipeCond(Pipe_t *p, void *data, unsigned size);
unsigned PutPipeTimed(Pipe_t *p, void *data, unsigned size,
      unsigned msecs);
```

Funciones para leer y escribir el pipe, cada una con tres variantes (sin timeout, con timeout y condicional). La semántica de lectura y escritura es igual que en los pipes de Unix. PutPipe() intenta escribir una cantidad de bytes en el pipe, y se bloquea si el pipe está lleno. Retorna cuando puede escribir algo, que puede ser menor que la cantidad de bytes que se pedía. Similarmente, GetPipe() intenta leer una cantidad de bytes y se bloquea cuando el pipe está vacío. Retorna cuando puede

leer algo, que puede ser una cantidad menor que la pedida. Las variantes con timeout pueden retornar 0 bytes prematuramente por agotamiento del tiempo, y las variantes condicionales nunca se bloquean. Las funciones retornan la cantidad de bytes leídos o escritos.

```
unsigned AvailPipe(Pipe_t *p);
```

Retorna la cantidad de bytes contenidos en el pipe.

Colas de mensajes

La cola de mensajes es un FIFO de mensajes de tamaño fijo, que se leen y escriben de a uno. La implementación usa un buffer circular y dos semáforos.

```
MsgQueue_t *CreateMsgQueue(char *name, unsigned msg_max,
      unsigned msg_size);
void DeleteMsgQueue(MsgQueue_t *mq);
```

Creación y destrucción. Los parámetros msg_max y msg_size describen la máxima cantidad de mensajes y el tamaño de cada uno, respectivamente.

```
bool GetMsgQueue(MsgQueue_t *mq, void *msg);
bool GetMsgQueueCond(MsgQueue_t *mq, void *msg);
bool GetMsgQueueTimed(MsgQueue_t *mq, void *msg, unsigned msecs);
bool PutMsgQueue(MsgQueue_t *mq, void *msg);
bool PutMsgQueueCond(MsgQueue_t *mq, void *msg);
bool PutMsgQueueTimed(MsgQueue_t *mq, void *msg, unsigned msecs);
unsigned AvailMsgQueue(MsgQueue_t *mq);
```

Estas funciones son similares a las del pipe, pero escriben y leen un mensaje de tamaño fijo por vez. Las variantes condicionales (timeout 0) pueden ser utilizadas en rutinas de interrupción. Por ejemplo, en el driver de teclado (ps2.c) la interrupción de teclado coloca en una cola de mensajes los códigos leídos (scan codes).

Manejo del coprocesador aritmético

El contexto del coprocesador aritmético se guarda y restablece para las tareas que lo utilizan, pero solamente cuando es necesario. Al inicializarse el sistema de manejo del coprocesador, se captura la excepción 7. Esta excepción se genera cuando se intenta ejecutar una instrucción de coprocesador y está levantado el bit TS (task switch) en el registro de control CR0. Este bit es levantado por la función mt_select_task() cuando cambia la tarea actual y ésta no coincide con la última tarea que utilizó el coprocesador. El manejador de esta excepción (en el archivo math.c), resetea el bit y luego analiza si debe guardar el contexto del coprocesador en la última tarea que lo utilizó, y restaurarlo en la actual. El sistema es muy eficiente: si una sola tarea utiliza el coprocesador no se produce ningún guardado o restauración de contexto del mismo (ni siquiera se produce la excepción 7).

Este mecanismo supone que solamente las tareas utilizan el coprocesador. **Las rutinas de interrupción no deben utilizar el coprocesador,** a menos que guarden y restablezcan su contexto manualmente.

Manejo de interrupciones y excepciones

```
typedef void (*exception_handler)(unsigned except_number,
      unsigned error, mt_regs_t *regs);
typedef void (*interrupt_handler)(unsigned irq_number);

void mt_set_int_handler(unsigned irq_num, interrupt_handler handler);
void mt_set_exception_handler(unsigned except_num,
      exception_handler handler);
void mt_enable_irq(unsigned irq);
void mt_disable_irq(unsigned irq);
```

MTask utiliza una IDT de 48 entradas. Las entradas 0 a 31 están destinadas al manejo de excepciones del i386, y las entradas 32 a 47 al manejo de las interrupciones de hardware 0 a 15 respectivamente. Estos 48 vectores de interrupción son pequeños stubs que llaman a un manejador único.

El manejador único empuja en el stack el contexto de la tarea actual e incrementa una variable que lleva cuenta del nivel de anidamiento de interrupciones. Si se trata de una interrupción de primer nivel (es decir, el nivel de anidamiento pasa de 0 a 1) guarda el valor del puntero al stack (esp) en el bloque de control de la tarea actual (mt_curr_task) y cambia el stack a un stack interno dedicado al manejo de interrupciones. Este stack quedará puesto hasta que se retorne de la interrupción de primer nivel (es decir, cuando el nivel de anidamiento pase de 1 a 0).

Para atender la interrupción o excepción el manejador indexa un par de tablas de punteros a manejadores de interrupción o de excepción, respectivamente.

Los manejadores de excepción reciben como argumento el número de excepción (0 a 31), un código de error y un puntero a una estructura (definida en kernel.h) que describe el contenido del stack frame que contiene el contexto guardado, es decir, los registros del procesador en el momento de producirse la excepción. El código de error es el que genera el i386 para algunas excepciones, y vale 0 para las demás. Mediante el puntero a la estructura de registros pueden no solamente leerse, sino también modificarse los valores de dichos registros; los valores nuevos se harán efectivos cuando el manejador de excepción retorne y se reponga el contexto. Se los llama con interrupciones deshabilitadas, pero pueden habilitarlas si es oportuno.

Los manejadores de interrupciones de hardware reciben como argumento el número de interrupción (0 a 15). Estos manejadores deben hacer lo necesario para resetear la interrupción en el periférico que la generó, pero no deben ocuparse de enviar el código de fin de interrupción (EOI) a los PICs. El manejador único se encarga de eso. Se los llama con interrupciones deshabilitadas, pero pueden habilitarlas si lo consideran adecuado. Los PICs garantizan que una interrupción no será interrumpida por sí misma, de modo que no es necesario que el manejador sea reentrante.

Cuando retorna el manejador de excepción o interrupción, el manejador común decrementa la variable que lleva el nivel de anidamiento de interrupciones. Si esta variable pasa de 1 a 0 (es decir, se está retornando de una interrupción de primer nivel), llama a la función mt_select_task(), que es la encargada de evaluar si la tarea actual debe conservar la CPU o conviene dársela a otra. En este último caso, mt_select_task() modifica el valor de la variable global mt_curr_task, que apunta al bloque de control de la tarea actual. Luego el manejador cambia el puntero al stack (esp) al valor almacenado en el bloque de control de la tarea actual (que puede ser distinta de la que había cuando se produjo la interrupción), recupera el contexto del stack y retorna. Este es el mecanismo que da a MTask su carácter preemptivo: la tarea actual puede cambiar al retornar una interrupción de primer nivel.

Ni los manejadores de excepción ni los de interrupción pueden llamar a ninguna función potencialmente bloqueante (como esperar en un semáforo) o a ninguna otra función que potencialmente cambie el estado de la tarea actual, como Yield(). El sistema no puede realizar un cambio de contexto mientras se está utilizando el stack de interrupciones. Adicionalmente, los manejadores de interrupción no pueden utilizar el coprocesador aritmético (salvo que guarden y repongan su contexto) ni alocar o liberar memoria dinámica.

Las únicas funciones de la API que pueden llamarse desde manejadores de interrupción son las siguientes:

API principal

- Panic()
- CurrentTask()
- GetName()
- SetInts()
- SignalQueue()
- FlushQueue()
- SendCond() (o SendTimed() con timeout 0)

API de IPC

- ValueSem()
- SignalSem()
- FlushSem()
- AvailMsgQueue()
- PutMsgQueueCond() (o PutMsgQueueTimed() con timeout 0)

Para imprimir mensajes de depuración, las interrupciones deben utilizar la función print0(). No se debe usar usar printk(), pues los mensajes irían a parar a la consola que está puesta en el momento de producirse la interrupción, la cual puede tener el foco o no. Esto traería dos consecuencias: (1) los mensajes podrían no ser visibles en el momento en que se producen y (2) estropearían aleatoriamente las pantallas de distintas consolas.

Con respecto a las demás funciones de biblioteca, conviene examinar el código de cada una para verificar que sean reentrantes o estén protegidas contra interrupciones.

Inicialmente todas las interrupciones y excepciones están atendidas por un par de manejadores por defecto, que se limitan a imprimir un mensaje de interrupción no manejada o excepción no manejada y detienen el sistema. Para establecer un manejador para una excepción o interrupción, pueden utilizarse las funciones mt_set_exception_handler() y mt_set_int_handler(). Las funciones mt_enable_irq() y mt_disable_irq() habilitan y deshabilitan interrupciones de hardware específicas modificando las máscaras de los controladores de interrupciones maestro y esclavo (PICs).

Al arrancar, MTask establece un manejador para la interrupción de timer (IRQ 0) y la habilita. Luego, al inicializar el driver de teclado, se establecen manejadores para la interrupción de teclado (IRQ 1) y la de mouse (IRQ 12) y se las habilita. El driver IDE captura y habilita las IRQs 14 y 15.

Consola

El módulo de consola maneja el display VGA en modo texto con 25 líneas de 80 caracteres, escribiendo en la memoria de video. El sistema de coordenadas es el siguiente: **x** indica la posición horizontal o número de columna, de 0 a 79, e **y** indica la posición vertical o número de línea, de 0 a 24. El extremo superior izquierdo de la pantalla tiene las coordenadas x=0 e y=0. El extremo inferior derecho tiene x=79 e y=24. Vale decir que x se mide de izquierda a derecha, e y se mide de arriba hacia abajo.

Los glifos de los caracteres son los del BIOS de la PC, es decir, corresponden a la página de código 437.

Los caracteres pueden mostrarse con un color de texto (foreground) y un color de fondo (background). Los colores posibles forman la siguiente enumeración:

```
enum COLORS
{
	/* oscuros */
	BLACK,
	BLUE,
	GREEN,
	CYAN,
	RED,
	MAGENTA,
	BROWN,
	LIGHTGRAY,

	/* claros */
	DARKGRAY,
	LIGHTBLUE,
	LIGHTGREEN,
	LIGHTCYAN,
	LIGHTRED,
	LIGHTMAGENTA,
	YELLOW,
	WHITE
};
```

Por defecto, el color de fondo es BLACK y el de texto LIGHTGRAY.

```
void mt_cons_clear(void);
void mt_cons_clreol(void);
void mt_cons_clreom(void);
```

Estas funciones borran distintas partes de la memoria de video, poniendo en la zona borrada los atributos de color por defecto (LIGHTGRAY, BLACK). Mt_cons_clear() borra toda la pantalla y coloca el cursor en (0, 0); mt_cons_clreol() borra desde donde está el cursor hasta el final de la línea, y mt_cons_clreom() borra desde donde está el cursor hasta el final de la pantalla, en ambos casos sin modificar la posición del cursor.

```
unsigned mt_cons_nrows(void);
unsigned mt_cons_ncols(void);
unsigned mt_cons_nscrolls(void);
```

Estas funciones devuelven la cantidad de filas (25) y columnas (80) del display, y la cantidad de veces que se desplazó la pantalla una línea hacia arriba (scrolling). El scrolling se produce automáticamente cuando se intenta escribir más allá del punto (79, 24) o el cursor está en la línea 24 y se intenta moverlo una línea más abajo.

```
void mt_cons_getxy(unsigned *x, unsigned *y);
void mt_cons_gotoxy(unsigned x, unsigned y);
```

Permiten leer y establecer la posición del cursor.

```
void mt_cons_getattr(unsigned *fg, unsigned *bg);
void mt_cons_setattr(unsigned fg, unsigned bg);
```

Permiten leer y establecer los atributos de color que se utilizarán en lo sucesivo para escribir caracteres en la pantalla. Los parámetros "fg" y "bg" significan "foreground" (color del texto) y "background" (color del fondo). Los colores tienen valores de la enumeración que se vio más arriba.

```
bool mt_cons_cursor(bool on);
```

Muestra u oculta el cursor, devolviendo el estado previo.

```
void mt_cons_putc(char ch);
void mt_cons_puts(const char *str);
```

Escriben caracteres en la pantalla a partir de la posición del cursor y desplazan el cursor. Cuando la consola no está en modo crudo, se interpretan en forma especial los siguientes caracteres:

- Retorno de carro (\r) : Coloca el cursor en el borde izquierdo.
- Nueva línea (\n): Mueve el cursor una línea hacia abajo, o produce un scrolling si está en la línea 24.
- Tabulador (\t): Escribe los espacios necesarios para llevar el cursor a una posición horizontal múltiplo de 8.
- Backspace(0x08): Mueve el cursor un lugar hacia la izquierda, a menos que ya esté en el borde izquierdo.

Los demás caracteres no tienen ninguna interpretación especial y generan los glifos correspondientes.

```
bool mt_cons_raw(bool on);
void mt_cons_cr(void);
void mt_cons_nl(void);
void mt_cons_tab(void);
void mt_cons_bs(void);
```

La función mt_cons_raw() permite poner la consola en modo crudo, devolviendo el estado anterior. En este modo no se interpreta ningún carácter en forma especial, es decir, se generan glifos para cualquier valor de carácter. Cuando la consola está en modo crudo pueden utilizarse las funciones mt_cons_cr(), etc. para mover el cursor produciendo los mismos efectos que producen normalmente los caracteres especiales. Por ejemplo, ejecutar mt_cons_cr() produce el mismo efecto que un retorno de carro en el modo normal.

```
void mt_cons_setfocus(unsigned consnum);
void mt_cons_setcurrent(unsigned consnum);
unsigned mt_cons_set0(void);
```

El módulo de consola maneja 9 consolas virtuales (0-8), a las que puede accederse mediante las combinaciones de teclas ALT-ESC para la consola 0 y ALT-F1 a ALT-F8 para las consolas 1 a 8 respectivamente. Decimos que la consola que está visible en cada momento es la que "tiene el foco". El foco se cambia llamando a la función mt_cons_setfocus(), que es lo que el módulo PS2 hace indirectamente - a través de una llamada a mt_input_setfocus() - cuando procesa las combinaciones de teclas mencionadas.

Por otra parte, cada tarea tiene asignada una consola, ya sea heredada de la tarea que la creó o establecida mediante una llamada a SetConsole(). La consola activa, o "consola actual", es la que corresponde a la tarea que está ejecutando y se establece llamando a la función mt_cons_setcurrent(), cosa que el scheduler hace indirectamente en cada cambio de contexto, llamando a mt_input_setcurrent().

La función mt_cons_set0() pone como consola actual la consola 0, y retorna el número de consola que estaba puesto antes. Es utilizada por la función print0() para imprimir mensajes en la consola 0 y luego reponer la consola que estaba puesta.

Las funciones de consola trabajan sobre la consola actual. Si la consola actual no coincide con la que tiene el foco, los cambios se harán en la consola virtual correspondiente, pero no serán visibles en pantalla hasta que esta consola no tome el foco. En cambio, si la consola actual tiene el foco los cambios se realizan directamente en la consola física y resultan inmediatamente visibles. Cada vez que se cambia la consola que tiene el foco, el estado de la consola física (incluyendo el contenido de la memoria de video) se guarda en la consola virtual que ha perdido el foco, y se recupera de la consola virtual que lo ha tomado.

La función Panic() cambia el foco a la consola actual, para garantizar que el mensaje de error sea visible cuando se detiene el sistema.

La familia printk

```
int vprintk(const char *fmt, va_list args);
int printk(const char *fmt, ...);
int print0(const char *fmt, ...);
void cprintk(unsigned fg, unsigned bg, char *fmt, ...);
```

Printk() se ha utilizado tradicionalmente como printf() en el kernel, para imprimir directamente en consola. Aquí se da en dos variantes, printk() y vprintk(), a las cuales se agrega una tercera que permite imprimir con determinados atributos de color. Cprintk() modifica temporariamente los atributos de color de la consola, pero luego repone los valores originales.

Print0() es igual a printk(), pero imprime siempre en la consola 0, con independencia de cuál es la consola actual. Está pensada especialmente para utilizar en rutinas de interrupción para imprimir mensajes de depuración, pues de lo contrario los mensajes de las interrupciones aparecerían en la consola de la tarea interrumpida. Sin embargo también puede usarse fuera de las rutinas de interrupción, cuando se desea que los mensajes de depuración no estropeen la pantalla en la cual trabaja la aplicación. Se recomienda que todas las impresiones en consola 0, incluyendo los eventuales mensajes de depuración de las tareas internas que corren sobre la consola 0, también utilicen print0() en vez de printk(). Printk() está serializada mediante Atomic()/Unatomic(), ya que está previsto que solamente la usen las tareas para imprimir en consolas distintas de la 0. En cambio, print0() está serializada mediante deshabilitación de interrupciones pues está previsto que éstas la usen.

Teclado y mouse

El módulo de teclado y mouse (ps2.c) tiene capturada la interrupción de teclado (IRQ 1). La rutina de interrupción lee los códigos del teclado (scan codes) y los coloca en una cola de mensajes.

Una tarea de alta prioridad lee los códigos de esta cola, genera eventos de teclado, que coloca en una cola de eventos en el módulo de entrada, y los procesa. El procesamiento consiste en atender teclas especiales como las teclas de función con sus posibles combinaciones con las teclas de control, alt izquierdo y alt derecho y mapear los códigos que no son teclas especiales generando caracteres que pueda leer una aplicación. El mapeado se realiza utilizando los mapas de distribución de teclado de Minix (una buena parte del código del driver está copiada casi literalmente de Minix). En la actualidad están soportadas las distribuciones de teclado "spanish" y "us-std", pero es muy simple agregar otras distribuciones. Los caracteres generados se colocan en el módulo de entrada en una segunda cola de mensajes, para ser leídos desde allí por las aplicaciones.

```
const char *mt_ps2_getlayout(void);
bool mt_ps2_setlayout(const char *name);
const char **mt_ps2_layouts(void);
```

La primera función devuelve el nombre de la distribución de teclado actual, por ejemplo, "us-std". La segunda permite estaclecer una distribución. Actualmente el argumento debe ser "spanish" o "us-std". La tercera función devuelve una lista de las distribuciones disponibles, bajo la forma de un array de char * terminado en NULL; actualmente devuelve { "spanish", "us-std", NULL }.

El módulo también atrapa la interrupción de mouse (IRQ 12), genera eventos de mouse y los coloca en la cola de eventos de entrada, junto con los eventos de teclado.

Módulo de entrada

Las aplicaciones que necesiten recibir los eventos de teclado sin procesar y los eventos de mouse pueden leerlos de una cola de mensajes de eventos. Esta es la interfaz de eventos del módulo de entrada (input.c):

```
enum { KBD_EVENT, MOUSE_EVENT };

typedef struct __attribute__((packed))
{
      unsigned char scan_codes[3];
}
kbd_event_t;

typedef struct __attribute__((packed))
{
      union __attribute__((packed))
      {
            struct __attribute__((packed))
            {
                  unsigned char left_button:1;
                  unsigned char right_button:1;
                  unsigned char middle_button:1;
                  unsigned char always_1:1;
                  unsigned char x_sign:1;
                  unsigned char y_sign:1;
                  unsigned char x_ovfl:1;
                  unsigned char y_ovfl:1;
            };
            unsigned char header;
      };
      unsigned char x;
      unsigned char y;
}
mouse_event_t;

typedef struct __attribute__((packed))
{
      unsigned char type;
      union
      {
            kbd_event_t kbd;
            mouse_event_t mouse;
      };
}
input_event_t;
```

Los eventos de teclado corresponden al make o break de una tecla y tienen de 1 a 3 bytes (códigos de scan). Si el primer byte es 0xE0 se trata de una secuencia de dos códigos, si es 0xE1 se trata de una secuencia de 3 códigos, y en todos los demás casos solamente 1 código. El evento de mouse es un paquete de 3 bytes tal como lo envía el mouse, sin ningún procesamiento especial. Si se desea consultar la interpretación y uso de estos eventos, ver el fuente de la aplicación "events" (events.c).

```
bool mt_input_put(input_event_t *ev);
```

Esta función es llamada desde el módulo PS2 cada vez que genera un evento de teclado o mouse, para colocar el evento en la cola de mensajes correspondiente.

```
bool mt_input_get(input_event_t *ev);
bool mt_input_get_cond(input_event_t *ev);
bool mt_input_get_timed(input_event_t *ev, unsigned timeout);
```

Estas funciones son llamadas por las aplicaciones para leer los eventos de la cola de mensajes correspondiente.

En el caso más habitual, las aplicaciones leen la entrada de teclado a nivel de caracteres. La interfaz de caracteres del módulo de entrada está formada por las siguientes funciones:

```
bool mt_kbd_putch(unsigned char c);
bool mt_kbd_puts(unsigned char *s, unsigned len);
```

Estas funciones son llamadas desde el módulo PS2 para poner un carácter o una secuencia de

caracteres en la cola de mensajes de caracteres. Mt_kbd_puts() garantiza atomicidad, es decir, los caracteres se ponen todos en la cola, o ninguno si no hay lugar suficiente.

```
bool mt_kbd_getch(unsigned char *c);
bool mt_kbd_getch_cond(unsigned char *c);
bool mt_kbd_getch_timed(unsigned char *c, unsigned timeout);
```

Estas funciones son llamadas desde las aplicaciones para leer un carácter de la cola de mensajes de caracteres, ya sea sin timeout, condicionalmente (timeout = 0) y con timeout.

Análogamente al módulo de consola, el módulo de entrada maneja 9 canales de entrada virtuales, cada uno de los cuales está formado por una cola de mensajes de eventos y una de caracteres.

```
void mt_input_setfocus(unsigned consnum);
```

El canal de entrada que "tiene el foco" tiene una correspondencia directa con la consola que tiene el foco. Las funciones que ponen eventos o caracteres en el módulo de entrada lo hacen en las respectivas colas de mensajes que tienen el foco. La función mt_input_setfocus() selecciona estas colas y llama a mt_cons_setfocus() para cambiar el foco de la consola. Es llamada por el módulo PS2 cuando se procesan las secuencias de teclas ALT-ESC y ALT-F1 a ALT-F8.

```
void mt_input_setcurrent(unsigned consnum);
```

El canal de entrada actual tiene una correspondencia directa con la consola actual. Esta función selecciona las colas de las que leen las funciones mt_input_get...() y mt_kbd_getch...() y llama a mt_cons_setcurrent() para cambiar la consola actual. Es llamada desde el scheduler con cada cambio de contexto.

Leer caracteres

```
int getch(void);
int getch_cond(void);
int getch_timed(unsigned timeout);
```

Estas funciones leen caracteres del teclado, con espera indefinida, en forma condicional (timeout 0) o con timeout. Retornan el carácter leído o EOF (-1).

Leer una línea

```
#define BACK      (-'A')
#define FWD       (-'B')
#define FIRST     (-'H')
#define LAST      (-'Y')

int getline(char *buf, unsigned size);
```

Esta función se usa para leer una línea del teclado simulando la lectura de una terminal en modo canónico. Se le pasa un buffer y su tamaño, retorna una línea leída (terminada en cero) y la cantidad de caracteres leídos (sin contar el cero final).

Getline() respeta el contenido inicial del buffer que se le pasa como si lo hubiera leído del teclado. Esto permite implementar historia de líneas. Para comenzar con una línea vacía, se debe vaciar el buffer (asignando *buf = 0) antes de llamar a getline(). La entrada de la línea termina cuando se oprime ENTER, que genera un retorno de carro, o cuando se llena el buffer. Cuando el ingreso termina por la tecla ENTER, el retorno de carro aparece reemplazado por un carácter de nueva línea (\n). Mientras no termina el ingreso de la línea, puede utilizarse la tecla BACKSPACE para borrar los últimos caracteres ingresados. La función va haciendo eco sobre la pantalla mostrando en cada momento el estado del buffer, expandiendo los tabuladores sobre la pantalla (pero no en el buffer) y llevando cuenta de la posición del cursor en cada momento. Soporta el ingreso de líneas que ocupan más de una línea en la pantalla.

Para soportar historia de líneas, getline() retorna ciertos valores negativos cuando detecta las flechas hacia arriba o hacia abajo o las teclas INICIO (HOME) y FIN (END) del keypad, dejando en el buffer el contenido ingresado hasta el momento. Estos valores negativos pueden ser utilizado por quien llama a getline() para cargar líneas previamente memorizadas en el buffer de entrada y así navegar por la historia. El shell utiliza esta función para leer las líneas de comando.

Discos IDE

El driver reconoce e identifica hasta 4 discos IDE (ATA) en los controladores primario y secundario, discos maestro y esclavo. Reconoce, pero no maneja, los dispositivos ATAPI (como CD-ROM). Espera que los controladores primario y secundario estén en sus direcciones de entrada/salida normales (0x1F0 y 0x3F6 para el primario y 0x170 y 0x376 para el secundario) y utilicen las interrupciones habituales (14 y 15 respectivamente).

Es un driver rudimentario que utiliza el modo PIO por interrupción y requiere las siguientes características de los dispositivos:

- Interfaz ATA (no ATAPI). Los dispositivos ATAPI son reconocidos, pero se les asigna capacidad 0.
- Soporte de comandos "read multiple" y "write multiple", que permiten transferir múltiples sectores generando una sola interrupción.
- Direccionamiento LBA. No soporta el viejo sistema CHS (cilindro/cabeza/sector).

```
enum ide_minors
{
      IDE_PRI_MASTER,
      IDE_PRI_SLAVE,
      IDE_SEC_MASTER,
      IDE_SEC_SLAVE
};

unsigned mt_ide_read(unsigned minor, unsigned block, unsigned nblocks,
      void *buffer);
unsigned mt_ide_write(unsigned minor, unsigned block, unsigned nblocks,
      void *buffer);
```

Leen y escriben una cantidad de sectores (nblocks) a partir de un sector determinado (block). El primer sector del disco es el sector 0. Retornan la cantidad de sectores leídos o escritos, que puede ser menor que la solicitada. La cantidad de sectores que pueden leerse o escribirse por cada llamado está limitada por la máxima cantidad de sectores que soportan los comandos "read multiple" y "write multiple" del disco, normalmente 128. También se reduce automáticamente la cantidad de sectores si los últimos exceden la capacidad del disco. Cuando encuentran errores, estas funciones retornan cero.

```
char *mt_ide_model(unsigned minor);
```

Devuelve el modelo del disco, NULL si no está presente.

```
unsigned mt_ide_capacity(unsigned minor);
```

Devuelve la capacidad del disco en sectores, 0 si no está presente o es de tipo ATAPI.

Funciones de biblioteca

Por distintos motivos, quien escribe un kernel debe resignarse también a escribir su propia biblioteca de funciones standard de C, pues a menudo es difícil o imposible utilizar la biblioteca que viene con el compilador. Aparte de que la biblioteca del compilador suele hacer llamadas al sistema operativo para el cual está construída, aún las funciones que no las hacen a menudo dependen de ciertas inicializaciones y de cierto ambiente creado por el código de arranque de los programas, código que no se ejecuta cuando estamos construyendo un sistema operativo.

En el desarrollo de MTask fue necesario definir algunas funciones de biblioteca comunes. La mayoría han sido copiadas o adaptadas de distintas fuentes; en los archivos fuente se dan los créditos correspondientes. Algunas de estas funciones no son thread-safe, de modo que es mejor mirar los fuentes antes de usarlas.

Entrada/salida de 8, 16 y 32 bits

```
unsigned inb(unsigned ioaddr);
void outb(unsigned ioaddr, unsigned data);
unsigned inw(unsigned ioaddr);
void outw(unsigned ioaddr, unsigned data);
unsigned inl(unsigned ioaddr);
void outl(unsigned ioaddr, unsigned data);
```

Manejo de strings

```
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, unsigned count);
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, unsigned count);
int strcmp(const char *cs, const char *ct);
int strncmp(const char *cs, const char *ct, unsigned count);
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
unsigned strlen(const char *s);
unsigned strnlen(const char *s, unsigned count);
void *memcpy(void *to, const void *from, unsigned n);
void *memmove(void *dest, const void *src, unsigned n);
void *memchr(const void *cs, int c, unsigned count);
void *memset(void *s, int c, unsigned count);
```

Generador de números seudo-aleatorios

```
int rand(void);
void srand(unsigned seed);
```

La familia sprintf

```
int vsprintf(char *buf, const char *fmt, va_list args);
int sprintf(char *buf, const char *fmt, ...);
```

Manejador de memoria

```
void *malloc(unsigned nbytes);
void free(void *ap);
```

Separar un string en campos

```
const char *setfs(const char *fs);
unsigned split(char *s, char *field[], unsigned nfields);
unsigned separate(char *s, char *field[], unsigned nfields);
```

Los tokens o campos pueden estar delimitados por espacio en blanco u otros separadores (split) o espacio en blanco y comillas (separate). El shell usa la función separate() para separar una línea de comando en argumentos.

Convertir un string en un número entero

```
int atoi(const char *s)
long strtol(const char *nptr, char **endptr, int base);
unsigned long strtoul(const char *nptr, char **endptr, int base);
```

String que describe el estado de una tarea

```
const char *statename(unsigned state);
```

Strings que devuelve para cada estado:

TaskSuspended	"suspended"
TaskReady	"ready"
TaskCurrent	"current"
TaskDelaying	"delay"
TaskWaiting	"wait"
TaskSending	"send"
TaskReceiving	"receive"
TaskJoining	"join"
TaskZombie	"zombie"
TaskTerminated	"finished"