TRABAJO PROFESIONAL F.I.U.B.A.

Micro Kernel en Tiempo real

rtKERNEL 1.00

Guillermo Pablo Tomasini

agosto de 1994

agradecimientos:

quiero expresar mi agradecimiento a Bernardo Arainty que me facilitó el material correspondiente al uCOS kernel (en el cual inspiré mi código) y los fuentes del ITBA Space Invaders con en el cual depuré gran parte del kernel.

asimismo a Luis María Aguirre por el apoyo técnico brindado y a la empresa Siemens Argentina por facilitarme la impresión del presente informe y parte de mi tiempo de becado en la misma, para la conclusión del presente proyecto.

ÍNDICE GENERAL

1. INTRODUCCIÓN AL PROYECTO

Según las revistas extranjeras de software de base, hoy en día existen dando vueltas en el mercado decenas de pseudo operativos o microkernels aptos para tiempo real, (o al menos eso rezan sus especificaciones). Pero ninguno de ellos está desarrollado en el país, por lo que resulta prácticamente imposible conseguir apoyo técnico adecuado de las empresas que los proveen Por otra parte, y como para empeorar las cosas, muy pocos incluyen los fuentes, por lo cual se torna difícil, sino imposible adaptarlos a nuestras necesidades particulares. Los pocos que incluyen los fuentes, tienen un precio muy elevado y por lo general exigen el pago de derechos o licencia, por cada plataforma donde vayan a ser empleados.

En nuestro país se comercializa QNX de Quantum software, un sistema operativo distribuido de tiempo real para plataformas de 32 bits (especificamente el i386). Dicho operativo (que ha sido estudiado y analizado en este proyecto) si bien presenta una performance muy buena y atractivos servicios, resulta muy caro para aplicaciones pequeñas de tiempo real. Puesto que el precio del operativo es de aproximadamente u$1500, a lo cual hay que sumarle el precio de todas las herramientas de desarrollo, dado que no sirven las de DOS.

Por todo esto no resulta descabellado el desarrollo de un kernel propio, que corra sobre DOS, sin perder de vista las herramientas aptas para tal entorno. Que pueda ser roomeable y adecuado para aplicaciones embebidas.

Se emplea el i8086 como plataforma de desarrollo dado que el mismo se popularizó ampliamente con la PC. Sin embargo el código es fácilmente migrable a otro entorno, dado que las funciones que lindan con la arquitectura del microprocesador se escribieron en módulos separados y son muy pocas las funciones escritas en lenguaje de máquina.

No se emplea el modo protegido del i386, ya que el tiempo de desarrollo se hubiera incrementado ostensiblemente (dado la escasez de herramientas para 32 bits) y se hubiera complicado en demasía el acceso a los servicios del DOS y el depurado del código. Quizás futuras versiones incluyan el modo protegido.

La única función que se emplea del DOS es el *alloc()* que es fácilmente reemplazable en caso de querer prescindir del DOS.

Se empleo como lenguaje de desarrollo el C, por ser el único adecuado para tal fin.

El operativo se proveerá un forma de biblioteca modelo *large*, para ser linkeada con aplicación del cliente.

Para probar y depurar el producto, se desarrolló un demo que hace uso intensivo de prácticamente la mayoría de los servicios del kernel. Se crean un total de ocho tareas que funcionan concurrentemente empleando como recursos: el teclado (entrada standard), la pantalla (salida standard), dos interfaces series y el coprocesador (para cálculo de estadísticas). Los accesos al DOS se serializaron por medio de un semáforo.

Los sistemas en tiempo real se caracterizan por el hecho de que deben encontrar resultados lógicos correctos en un tiempo determinado. En este entorno es más apropiado tener un respuesta no del todo correcta a tiempo, que tener una respuesta correcta cuando ya es demasiado tarde. (ver capítulo de tiempo real). Por lo general estos son sistemas embebidos, es decir que el microprocesador se encuentra dentro del mismo sistema, y por lo general el programa se almacena en una ROM.

Se implementó el kernel como sistema de *soft* de tiempo real (ver el capítulo *Introducción a los sistemas de tiempo real).*

1.1. Ámbito de aplicación

El ámbito de aplicación en aplicaciones embebidas de este kernel sería el siguiente:

- Control de procesos
  - Procesamiento de alimentos.
  - Plantas químicas.
  - Control de temperatura u otros parámetros físicos.

- Automóviles
  - Control de motores.
  - Alarmas.
  - Computadora de abordo.

- Automatización de oficinas
  - Centrales telefónicas.
  - Alarmas y control de acceso.
  - Control de temperatura.

- Periféricos de computadoras
  - Adquisidores de datos.

- Robots

- Electrodomésticos
  - Hornos de microondas.
  - Lavadoras de platos.
  - Lavarropas.
  - Control de aire acondicionado.
  - Alarmas para casa.

- Entretenimientos
  - Juegos.

Se ha estudiado a modo de ejemplo de los servicios que debía suministrar un sistema operativo en tiempo real (sotr) el QNX de la empresa Quantum Software System Ltd., representado en el país por Automatic S.A. Se ha dedicado un capítulo a este tema en particular, puesto que se consideró de vital importancia conocer las características de un producto de acabado reconocimiento técnico en el mercado, antes de lanzarnos a escribir un micro kernel.

Por otra parte se han introducido en este informe nociones básicas y generales sobre sistemas operativos y sistemas operativos de tiempo real, para facilitar la comprensión de la arquitectura del rtKERNEL y la utilidad de los servicios prestados por el mismo.

1.2. Especificaciones y lineamientos generales del proyecto:

- El rtKERNEL funcionará en modo real del i8086 y se cargará sobre el DOS, pudiendo emplear los servicios que este provee (por ejemplo el de archivo), pero serializándolos por medio de un semáforo. La misma advertencia debe hacerse a los servicios de la BIOS, los cuales tampoco son reentrantes. Por lo tanto el modelo de memoria utilizado es el global, con lo cual se evita las demoras en el cambio de contexto asociado a la protección de memoria.
- El rtKERNEL está especialmente pensado para aplicaciones embebidas y romeables. Puede ser usado sin problemas en una placa de PC industrial.
- El rtKERNEL tendrá el formato de un *.exe*, y será desarrollado en el modelo *large* de memoria (segmento de código > 64K, segmento de datos > 64K), pudiéndose llevar a un modelo *small* (segmento de código < 64K, segmento de datos < 64K), fácilmente a pedido del cliente.
- Será suministrado en forma de biblioteca (*large*) que será linkeada con la aplicación del cliente.
- Si bien está concebido para el i8086, es muy fácil adaptarlo a otro microprocesador. Para ello basta modificar los módulos i186l_x.xxx, donde se involucra código específico del i8086. En el caso del módulo i186l_c.c, contiene la función *os_task_create()* donde se arma el stack de la tarea, en este caso deberán pushearse los registros del microprocesador destino cuidando el orden adecuado. Si se migrará a otro procesador de la familia Intel, especificamente el 386, habrá que guardar todos los registros de 32 bits. El módulo i186l_a.asm deberá ser cambiado al nuevo assembler si se tratara de un microprocesador de otra familia.
- Tendrá un *time slice* de 12 a 54 milisegundos, pudiéndose variar dentro de este rango por medio de un servicio del mismo.
- El rtKERNEL es *preemptivo (preemptived)*, es decir que si las tareas no se bloquea antes de finalizar su correspondiente *time slice*, al finalizar éste el microprocesador es cedido a la siguiente tarea lista para correr.
- Se permiten un máximo de 64 tareas corriendo, pero este valor es fácilmente modificable cambiando la macro OS_MAX_TASKS a otro valor.
- Se implementan un total de cuatro prioridades, siendo cero la más alta y cuatro la más baja. El número de prioridades es fácilmente alterable, modificando la macro OS_LOW_PRIO. La prioridad de cada tarea se puede cambiar dinámicamente, por medio de un servicio del kernel..
- Se implementa el algoritmo de *round robin* entre tareas de la misma prioridad. De existir tareas listas en el nivel de prioridad cero, se ejecutan éstas .De lo contrario se pasa a la prioridad siguiente, es decir la uno, y así sucesivamente hasta llegar a la última. Se piensa incluir otros algoritmos en futuras versiones, por ejemplo alguno que le quite prioridad a las tareas que acaparan el microprocesador.
- Se implementa un servicio de primitivas de sincronización de bajo nivel, denominadas *wait* y *signal*, que equivale a un servicio de semáforos. También se pueden usar como sincronización las colas y los mailbox, ya que ambos son bloqueantes. Todos los servicios de sincronización tienen *timeout.* La tarea que se despierta al hacer un *signal* sobre un semáforo menor que cero, es la de más alta prioridad, o la primera que se bloqueó en el mismo, en el caso de que las tareas dormidas tengan idéntica prioridad.

- Se implementa un servicio de mailbox, donde se guarda a lo sumo el puntero a un mensaje, con bloqueo de tareas. Este servicio puede emplearse como broadcast, ya que el *send* despierta a todas las tareas bloqueadas en un buzón. Es necesario crear los buzones explícitamente, ya que no existen buzones asociados a tareas. Esta facilidad se incluirá en una futura versión. Los mailbox tienen *timeout.*
- Se implementa un servicio de cola circular, donde no se guardan punteros, sino que se copia el elemento. Si bien esto puede resultar más lento que guardar un puntero, es un método más fiable, como veremos más adelante. Es posible crear un máximo de OS_MAX_QS (ver macro en rtkernel.h). Las colas son bloqueantes (la tarea se duerme si la cola a leer está vacía) y poseen *timeout* .La tarea que se despierta al hacer un *write* sobre una cola vacía que retiene tareas dormidas a la espera de mensajes, es la de más alta prioridad, o la primera que se bloqueó en el mismo, en el caso de que las tareas dormidas tengan idéntica prioridad.
- Se provee *binding dinámico* es decir que las tareas se pueden crear o matar dinámicamente. También se da la posibilidad de cambiar las prioridades de las tareas dinámicamente por medio de un servicio del kernel.
- El reloj del sistema no se altera en absoluto, pese a modificar el *tick_size* ya que existe un algoritmo encargado de mantener la hora correcta del sistema DOS.
- Casi todo el kernel está escrito en lenguaje C, por lo cual es muy fácil su manutención y es posible de migrar a otra plataforma. El único módulo escrito en assembler es el i186l_a.asm donde ser realiza el cambio de contexto.
- El cambio de contexto se plasma a través de las interrupciones de timer. En la función de interrupción se cambia el stack de la tarea en ejecución a los de la tarea que sigue. En realidad se cambia SS:SP, para que apunten al nuevo stack. Luego se hace un retorno de interrupción (IRET).
- Cabe destacar que las tareas que corren en este kernel en realidad son *threads* ya que comparten el segmento de datos (hay un segmento de datos común a todos los tasks). De todas formas no haremos distinción entre *threads, tasks, procesos y tareas.*
- Dado que el rtKERNEL es multitarea hay que tener especial cuidado con las bibliotecas empleadas en el mismo, dado que la mayoría de las bibliotecas para DOS no prestan funciones reentrantes.
- Se dan los listados de assembler generados por el compilador, para analizar si fuera necesario el tiempo de ejecución de cada servicio del kernel, en el caso de aplicaciones típicas.
- Se eligió como herramienta de desarrollo el Borland C++ 3.1, por considerarla la más adecuada para este fin. Se empleó el *tasm, bcc, make, td386, tlib* entre otros utilitarios de este paquete.
- Se ha probado este kernel a su vez, con un juego típico, el *Space Invaders* (desarrollado por el Ing. Bernardo Arainty) pensado para otro entorno operativo (el uCOS kernel), con el cual se desempeñó perfectamente. Cabe destacar que los juegos son una aplicación típica de tiempo real. No se ha presentado este juego en este desarrollo, pues no estaba terminado de adaptar al momento de redactar el presente informe.
- Se proveen un total de casi treinta servicios detallados a continuación.

1.3. Servicios prestados por el rtKERNEL

1.3.1. Gestión de tareas:

os_task_create

declaración:

BYTE os_task_create (void (far *task)(void *pd), void *pdata, void *pstk, PRIO prio, TASK_ID *task_id);

propósito:
Permite crear un thread (task o tarea) de ejecución. Se le debe pasar la dirección de dicha tarea, la dirección de su zona de datos, la dirección de su stack y la prioridad.

retorna:
Código de error: 0 si OK. En task_id retorna el identificador del task creado.

os_task_delete

declaración:

BYTE os_task_del (TASK_ID task_id);

propósito:
Permite destruir un task creado, pasándole el identificador de task task_id.

retorna:
Código de error: 0 si OK.

os_task_change_prio

declaración:

BYTE os_task_change_prio (TASK_ID task_id, PRIO newp);

propósito:
Permite cambiar la prioridad de un task creado. Se le debe pasar el identificador de task, *task_id* y la nueva prioridad *newp*.

retorna:
Código de error: 0 si OK.

os_get_cur_taskid

declaración:

TASK_ID os_get_cur_taskid (void);

propósito:

permite obtener el identificador del task que invoca este servicio.

retorna:

task_id del task que invoca esta función.

void os_task_end (void)

propósito:
Se ejecuta automáticamente al finalizar cualquier tarea. Permite sacar su TCB de la lista de ready tasks y ponerla nuevamente en la tabla de *os_tcb_free_list.*

1.3.2. Gestión de tiempo

os_tick_dly

declaración:

void os_tick_dly (WORD ticks);

propósito:
Permite demorar el task en ejecución *ticks* número de ticks del sistema. El task pasa a estado de DELAY.

retorna:
nada.

os_msec_dly

declaración:

void os_msec_dly (TIME msecs);

propósito:
Permite demorar el task en ejecución msecs número de milisegundos. En realidad los que se hace es el cociente de msecs por el tick size y se llama a *os_tick_dly().*

retorna:
nada.

os_time_set

declaración:

void os_time_set (LONG ticks);

propósito:
permite setear el número de ticks acumulado del sistema.

retorna:
nada.

os_time_get

declaración:

LONG os_time_get (void);

propósito:

permite obtener el número de ticks del sistema.

retorna:

nada.

os_set_ticksize

declaración:

void os_set_ticksize (WORD ticksize);

propósito:

permite setear el tamaño del tick del sistema en milisegundos (quantum de tiempo). El rango permitido para *ticksize* va de 12 a 54 milisegundos.

retorna:

nada.

os_get_ticksize

declaración:

LONG os_get_ticksize (void);

propósito:

permite obtener el tamaño del tick del sistema en milisegundos..

retorna:

tamaño del tick size en milisegundos.

1.3.3. Primitivas de sincronización

os_sem_create

declaración:

OS_ECB *os_sem_create (SWORD value);

propósito:

permite inicializar un semáforo en un valor determinado por value.

retorna:

puntero a event control block (OS_ECB).

os_sem_signal

declaración:

RETCODE os_sem_signal (OS_ECB *pevent);

propósito:

señala un semáforo (incrementa en uno el contador), si el resultado de la operación es menor o igual que cero despierta a tarea de más alta prioridad bloqueada en el mismo. Si hay varias tareas de igual prioridad, despierta a la primera que entró en la cola del semáforo. Se le debe suministrar el puntero al OS_ECB, obtenido de *os_sem_create().*

retorna:
status de operación: 0 si OK.

os_sem_wait

declaración:

RETCODE os_sem_wait (OS_ECB *pevent, LONG timeout);

propósito:
realiza una operación de wait sobre el semáforo (decrementa en uno el contador). Si el resultado de dicha operación es negativo bloquea la tarea en este evento. Se le debe suministra un puntero a un OS_ECB obtenido de *os_sem_wait()* y un timeout en número de ticks del sistema, si el timeout es cero significa timeout infinito o no timeout.

retorna:
status de operación: 0 si OK.

os_event_dest

declaración:

void os_event_dest (OS_ECB *pecb);

propósito:

permite destruir un semáforo creado a partir de *os_sem_create()*. Se le debe suministrar el *pecb* obtenido en *os_sem_create()*. Existe una macro llamada *os_sem_dest (OS_ECB)* para emplear esta función.

retorna:

nada.

1.3.4. Primitivas de comunicaciones y sincronización

os_mbox_create

declaración:

OS_ECB *os_mbox_create (void *msg);

propósito:
permite crear un buzón, e inicializarlo con *msg*. Cabe destacar que lo único que se guarda en el buzón es el puntero al mensaje, por lo cual si el contenido de lo apuntado por dicho puntero cambia, o bien se destruye por tratarse de una variable automática, tendremos un puntero a basura. En estos casos es conveniente emplear el servicio de colas, donde el mensaje o elemento se copia a memoria del kernel.

retorna:
retorna puntero a OS_ECB.

os_mbox_send

declaración:

RETCODE os_mbox_send (OS_ECB *pecb, void *msg);

propósito:
permite enviarle un mensaje al buzón apuntado por *pecb*. Si el buzón está ocupado se pierde el mensaje apuntado por *msg*.
Si existen una o más tareas bloqueadas en este buzón, se despertarán todas ellas.

retorna:
código de error. Cero si OK

os_mbox_receive

declaración:

void *os_mbox_receive(OS_ECB *pecb, TIME timeout, RETCODE *err);

propósito:

permite obtener mensaje de buzón apuntado por *pecb*. Si el buzón está vacío la tarea se bloquea hasta recibir algo, o bien hasta que perezca el *timeout*. Si este último es cero quedará permanentemente dormida a la espera de un mensaje.

En *err* retorna código de error. 0 si OK.

retorna:

puntero a mensaje.

os_event_dest

declaración:

void os_event_dest (OS_ECB *pecb);

propósito:

permite destruir un mailbox (buzón) creado a partir de *os_mbox_create()*. Se le debe suministrar el *pecb* obtenido en *os_mbox_create()*. Existe una macro llamada *os_mbox_dest (OS_ECB)* para emplear esta función.

retorna:

nada.

os_q_create

declaración:

OS_ECB *os_q_create (WORD qsize, BYTE elem_size);

propósito:

Permite crear una cola circular FIFO (primero en entrar primero en salir) para albergar *qsize* elementos de tamaño *elem_size* cada uno. Cabe destacar que es el kernel el encargado de alojar espacio para la cola y liberarlo luego con *os_q_dest()*. Cada elemento del usuario a pushear se copia en esta cola, evitando de esta forma los consabidos problemas con punteros a variables automáticas, o variables de las cuales se conserva su dirección y luego cambian con el transcurso de la ejecución. No sucede lo mismo con el caso de los buzones, donde lo que se guarda es un puntero, por lo cual si lo apuntado por el mismo cambia, o se destruye el sentido del mailbox se diluye.

retorna:

puntero a OS_ECB

os_q_write

declaración:

RETCODE os_q_write (OS_ECB *pecb, void *msg);

propósito:
permite escribir un elemento en la cola apuntada por *pecb*. Si la cola se encuentra llena, dicho elemento se descarta. Si la cola tuviera tareas bloqueadas en ella, se despertará la de más alta prioridad, o bien la primera que llegó, y ésta será la candidata a llevarse el mensaje.

retorna:
código de error. 0 si OK.

os_q_read

declaración:

void *os_q_read (OS_ECB *pecb, TIME timeout, RETCODE *err);

propósito:
permite leer un elemento de la cola. Si la cola estuviera vacía la tarea se bloqueará hasta que alguien escriba algo en ella. De lo contrario quedará bloqueada hasta que expire *timeout* si este parámetro fuera distinto de cero.
En *err* retorna código de error.

retorna:
puntero a elemento.

os_q_dest

declaración:

void os_event_dest (OS_ECB *pecb);

propósito:

permite destruir una cola y la memoria contenida en ella, creada a partir de *os_q_create()*. Se le debe suministrar el *pecb* obtenido en *os_q_create()*. Es imprescindible llamar a esta función antes de finalizar el kernel, ya que de lo contrario la memoria pedida dentro de *os_q_create()* no será devuelta al DOS.

retorna:

nada.

1.3.5. Inicialización y finalización del kernel

os_init

declaración:

void os_init (void)

propósito:

permite inicializar las tablas del kernel y sus variables globales. Debe ser la primer función a llamar del kernel.

retorna:

nada.

os_start

declaración:

void os_start (void);

propósito:

permite arrancar el kernel, es decir comenzar el switcheo de tareas, de no existir tareas creadas, la única que correrá será *os_task_idle().*

retorna:

nada.

os_end

declaración:

void os_end (void);

propósito:

permite terminar el kernel, es decir termina de correr, todas las tareas, devuelve la memoria pedida y ejecuta la sentencia siguiente a *os_start()*.

retorna:

nada.

1.3.6. Generales

os_sched_lock

declaración:

void os_sched_lock (void);

propósito:

permite lockear el switcheo de tareas. En realidad lo que hace es impedir el switcheo de tareas cuando la variable interna *os_lock_nesting* es distinta de cero. Dicha variable se incrementa en uno en cada llamado a esta función y se decrementa en cada llamado a *os_sched_unlock ( )*

retorna:

nada.

os_sched_unlock

declaración:

void os_sched_unlock (void);

propósito:

permite deslockear el switcheo de tareas. En realidad lo que hace es permitir el switcheo de tareas cuando la variable interna *os_lock_nesting* vale cero. Dicha variable se decrementa en uno en cada llamado a esta función y se incrementa en cada llamado a *os_sched_lock ( )*

retorna:

nada.

1.3.7. Códigos de error

```
#define  OS_NO_ERR              0
#define  OS_TIMEOUT                10
#define  OS_NO_MEMORY           11
#define  OS_MBOX_FULL              20
#define  OS_Q_FULL              30
#define  OS_Q_NULL              31
#define  OS_PRIO_ERR            41
#define  OS_SEM_ERR             50
#define  OS_SEM_OVF                51
#define  OS_TASK_DEL_ERR        60
#define  OS_TASK_DEL_IDLE          61
#define  OS_NO_MORE_TCB            70
#define  OS_TSK_NO_EXIST           71
#define  OS_INTERNAL_ERROR80
#define  OS_NULL_PECB              81
```

```
#define  OS_BAD_TICKSIZE        90

#define  NULL_ID                -1
#define  NULL_PRIO              -1
```

1.4. Estudio de factibilidad económica.

Se puede considerar que el único recurso empleado en este trabajo (diseño, desarrollo, implementación y depuración) son las horas hombre de desarrollo y de investigación. Las cuales suman aproximadamente unas 240 en total, entre desarrollo y depuración, es decir unas seis semanas de trabajo, si se consideran días laborales de ocho horas y semanas de cinco días. Se puede agregar si se quiere el precio de los libros consultados, algunos de los cuales debieron adquirirse. Por otra parte se considera que ya se cuenta con las herramientas de desarrollo adecuadas, en este caso el paquete del compilador Borland C++ 3.1 y su plataforma (máquina PC i386 con DOS), editor (Brief), procesador de texto, etc.

1.5. Planeamiento (diagramas de Pert y Gantt)

Según el gráficode Gantt siguiente, vemos que el proyecto comienza el 3-4-94 y finaliza y el 7-6-94. Pero esta estimación, se hace considerendo que se cuentan con recursos necesarios para efectuar tareas concurrentemente, como no es así sino que solo disponemos de un programador, es decir, yo, la fecha de finalización real se obtiene de sumar el tiempo de todas las tareas, lo cual arrojaría un total de 74 días. Se han considerado días de unas ocho horas laborables y semanas de cinco días laborables. De todas maneras a la hora de ejecutar el proyecto, esta norma no se ha seguido en absoluto, puesto que al constituir el proyecto un trabajo personal y no dependiente de una empresa, se emplearon fines de semanas, horas de almuerzo, tiempo libre dentro del horario laborable y en general, horas fuera del mismo. Sin embargo el tiempo total del proyecto es apoximadamente el estipulado en los siguientes diagramas, por lo cual los mismos no pierden validez.

Esta página permanece intencionalmente en blanco.

Esta página permanece intencionalmente en blanco.

Esta página permanece intencionalmente en blanco.

Esta página permanece intencionalmente en blanco.

Esta página permanece intencionalmente en blanco.

2. INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS

2.1. Qué es un sistema operativo?

Antes de analizar de qué se trata un microkernel en tiempo real, debemos tener en claro qué es un sistema operativo, que servicios presta, cuales son sus aplicaciones y limitaciones.

Casi todos los usuarios de computadoras han tenido algún contacto con algún sistema operativo, pero resulta difícil definir con exactitud que es un sistema operativo. Parte del problema es que básicamente los sistemas operativos realizan dos funciones básicamente no relacionadas, y según quién dé el concepto se escucha más cerca de una función o de la otra. Ahora analizaremos ambas funciones.

2.1.1. El sistema operativo como una máquina ampliada.

La arquitectura de la mayoría de las computadoras en el nivel lenguaje de máquina es difícil y primitiva de programar, especialmente en cuanto a entrada/salida (I/O) se refiere.

Sin entrar en detalles reales, debe quedar claro no desea involucrarse con la programación de bajo nivel. Por otra parte sea por ejemplo los detalles de acceso a la placa de video, estos cambian según el tipo de placa. En su lugar, lo que el programador desea es una abstracción simple de alto nivel con la cual trabajar. Una capa de software que oculte los detalles del hardware y también nos independice de ellos. No es pensable que cada vez que cambiemos de placa de video debamos reescribir gran parte de nuestra aplicación.

El programa que oculta los detalles ásperos del hardware al programador y presenta una vista simple y agradable de los recursos de la máquina, haciendo la vida un poco más llevadera, se llama sistema operativo.

En esta visión, la función del sistema operativo es la de presentar al usuario el equivalente de una *máquina virtual* o *ampliada* que sea más fácil de usar que el hardware implícito, es decir una capa de software que oculte los detalles de implementación. Es esta una visión descendente del sistema operativo.

2.1.2. Sistema operativo como administrador de recursos

En esta función alternativa, el trabajo del sistema operativo consiste en administrar los recursos (microprocesador, memoria y dispositivos de I/O) entre las tareas que compiten por ellos.

Esta necesidad se hace aún más evidente cuando la computadora tiene múltiples usuarios, que compiten por tiempo de CPU, acceso a disco, acceso a impresora, etc.

Esta vista del sistema operativo es ascendente (de abajo hacia arriba) del sistema operativo.

2.2. Tipos de sistemas operativos

Existen sistemas monousuario, monotarea muy populares tal como el DOS, o multitarea monousuario como el Windows, o multitarea, multiusuario como el UNIX. También existen sistemas particulares de consulta de información o gestión de operaciones. Pero los sistemas operativos que más nos interesan en este proyecto son los sistemas operativos en tiempo real.

Este tipo de sistemas operativos se emplean generalmente para el control de procesos, tales como por ejemplo procesos industriales.

La característica común de estos sistemas es que existe un feedback, es decir la computadora recibe información del medio a controlar, la procesa y da una respuesta tendiente a mantener la estabilidad del mismo. Por ejemplo un control de temperatura de un horno recibirá como parámetro la temperatura, y como salida controlará la resistencia del mismo. Existe evidentemente un período crítico de tiempo en el cual hay que proporcionar la respuesta si se quiere que el sistema siga en equilibrio.

2.3. Conceptos sobre los sistemas operativos

La interface entre el usuario y el sistema operativo se define como un conjunto de instrucciones extendidas que el sistema proporciona. Estas "instrucciones extendidas" se definen como llamadas al sistema (system calls). Estas llamadas varían de un sistema operativo a otro por lo cual no nos detendremos en los detalles de cada una, sino que suministraré como ejemplo las llamadas al rtKERNEL, cuando sea apropiado.

2.3.1. Procesos

Un concepto importante en todos los sistemas operativos es el de proceso (task) . Un proceso es básicamente un programa en ejecución. Consta de sus segmento de código, su segmento de datos, su stack, contador de programa y otros registros.

En un sistema multitarea, en forma periódica el sistema operativo decide suspender la ejecución de un proceso y brindarle la CPU a otro, ya sea porque el primero haya completado su parte de tiempo de CPU o bien porque se bloqueó aguardando la finalización de una I/O.

Cuando un proceso se suspende temporalmente como éste, debe reiniciarse después exactamente en el mismo estado en que se encontraba cuando fue desalojado. Esto implica que toda la información relativa al mismo debe guardarse en algún lugar durante la suspensión. La misma suele almacenarse en una zona del sistema operativo denominada tabla de procesos, la cual es un arreglo o lista enlazada de estructuras, una para cada proceso.

Existe también el concepto de *thread* (hilo de ejecución). La diferencia con un task radica en que todos los *threads* comparten el segmento de código y el de datos. Sin embargo cada thread tiene su propio stack. No existe protección en cuanto a los derechos de lectura o escritura de cada thread, por lo tanto el cambio de contexto es más rápido que en el caso anterior, donde si existía protección y donde no se compartían los segmentos mencionados. El concepto de thread está implementado en OS/2.

Existirán servicios del sistema operativo para crear y matar procesos.

2.3.2. Archivos

La otra categoría vasta de llamadas al sistema se relaciona con el sistema de archivos. Una función importante del sistema operativo consiste en ocultar las peculiaridades de los discos y otros dispositivos de I/O y presentar al programador un modelo abstracto limpio y agradable de archivos independientes del dispositivo físico. Las llamadas al sistema se necesitan con claridad para crear archivos, eliminar archivos, leerlos y escribirlos. Antes de que se pueda leer un archivo, éste debe abrirse y después de leído cerrarse, de modo que las llamadas se dan para hacer estas cosas.

No nos detendremos más en este tema, porque no es la idea implementar un sistema de archivos para el rtKERNEL, dado que esto llevaría un tiempo de desarrollo enorme. En su lugar emplearemos el servicio de archivos provisto por DOS.

2.3.3. Shell (intérprete de comandos)

El Shell es en sí la interface con el usuario. El mismo interpreta los comandos que ejecuta el usuario. El Shell no es otra cosa que un proceso más, cuando el usuario ejecuta algún comando, el Shell crea otro proceso hijo, que será el programa invocado por el usuario, y espera a que este termine. Si se tratara de un sistema operativo multitarea, como el UNIX, dicho proceso podría ejecutarse en background.

Esta primera versión de rtKERNEL no involucra Shell, por lo que no ampliaremos este tema.

3. PROCESOS

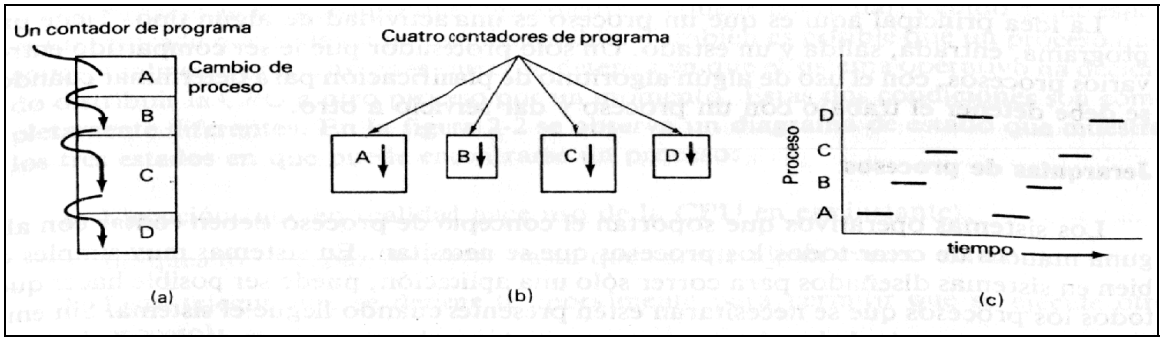
El concepto de proceso es el más importante dentro de un sistema operativo, el mismo es una abstracción de un programa en ejecución. Todo lo demás gira en torno a este concepto.

3.1. Introducción a los procesos

Todas las computadoras actuales pueden realizar varias tareas simultáneamente. Mientras ejecuta una aplicación puede hacer una impresión o bien leer el disco. En un sistema multitarea, al CPU cambia de un programa a otro en decenas o cientos de milisegundos. En rigor de verdad en cualquier instante la CPU está ejecutando un solo programa, pero a lo largo de al menos un segundo, brinda al usuario la ilusión de paralelismo, aunque en verdad se trate de *pseudo paralelismo.*

3.1.1. Modelo del proceso

Un proceso es básicamente un programa ejecutante, en donde intervienen los valores actuales del contador de programa, registros y variables. En forma conceptual cada proceso tiene su CPU virtual.
En la figura se observa cómo se resume esto en cuatro procesos, cada uno de los cuales tiene su flujo de control y cada uno corre independientemente de los otros. En la parte (c) de la misma se aprecia que, contemplados en un intervalo de tiempo bastante largo, todos los procesos han evolucionado , pero en cualquier instante de tiempo sólo uno corre en realidad.



Como la CPU conmuta entre un proceso y otro la velocidad a la cual un proceso realiza sus cálculos no será uniforme y probablemente no será reproducible si los mismos procesos se vuelven a ejecutar. Por los tantos las tareas no deben programarse con ideas preconcebidas respecto al tiempo.

La diferencia entre un proceso y un programa radica, en que un proceso es en realidad un programa en ejecución. Es decir un programa que se ha cargado en memoria, y se le brinda la CPU para que corra, es decir que un proceso es una instancia de ejecución de aquel programa. Puede existir más de una instancia de

ejecución del mismo programa, en este caso se comparte el segmento de código y cada proceso tendrá su propio segmento de datos.

La idea principal aquí es que un proceso es una actividad de algún tipo, tiene un programa, entrada, salida y un estado. Un solo procesador puede ser compartido entre varias tareas, con el uso de algún algoritmo de planificación para determinar cuando se debe detener la ejecución de la tarea corriente y alojar la siguiente.

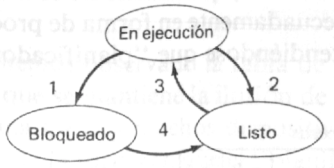
3.1.2. Estado de los procesos

Un proceso en ejecución puede bloquearse , debido a que por ejemplo espera una entrada que no está disponible, o bien simplemente el sistema operativo ha decidido darle la CPU a otro proceso. Estas condiciones son completamente diferentes. En la siguiente figura se observa un diagrama de estado que muestra los tres estados en que puede encontrarse una tarea:

1. En ejecución (hace uso de la CPU en este instante)

2. Bloqueado (incapaz de correr hasta que suceda algún evento externo)

3. Listo (es ejecutable, pero esta temporalmente detenido para permitir que se ejecute otra tarea).



Un proceso puede encontrarse en estado *de ejecución, bloqueado* o *listo* (que también se llama *ejecutable*).

Lógicamente los dos primeros estados son similares, en ambos casos los procesos correrán, pero en el segundo la CPU no está disponible temporariamente. El tercero es diferente de los dos primeros ya que el proceso no correrá aún cuando la CPU no tenga nada que hacer.

Cuatro transiciones son posibles entre dichos estados. La transición ocurre cuando una tarea descubre que no puede continuar, ya sea porque se bloquea en un semáforo, o porque espera la finalización de una I/O.

Las transiciones dos y tres las causa el scheduler (planificador de tareas del sistema operativo), sin que el proceso se entere. La transición dos se produce cuando la tarea en ejecución es desalojada de la CPU y se le da ésta a otra tarea lista para correr (transición tres). El tema de la planificación, es decir, la decisión de cual es la próxima tarea a ejecutarse se verá más adelante. Existen varios algoritmos al respecto, entre ellos el implementado en el rtKERNEL.

Empleando este modelo de tarea se vuelve mucho más simple pensar que es lo que ocurre dentro del sistema. Algunas tareas ejecutan comandos ingresados desde el Shell y otras son parte del mismo operativo, y dan por ejemplo servicios de archivos.

Esta observación da origen al modelo que se muestra en la siguiente figura. Aquí, el nivel inferior del sistema operativo es el planificador, con una diversidad de tareas además de ésta. Todos los detalles del manejo de interrupciones e inicio y suspensión

de las tareas están ocultos en scheduler, el cual es realmente muy pequeño. Generalmente corre a un nivel de privilegio más alto que las tareas del usuario.

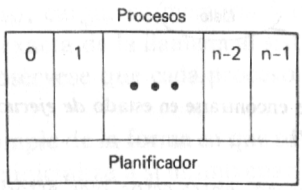


Fig. 2-3. El estrato inferior de un sistema operativo estructurado por procesos maneja las interrupciones y realiza la planificación. El resto del sistema consta de procesos secuenciales.

3.1.3. Implementación de tareas

Para implementar el modelo de proceso, el sistema operativo conserva una lista de estructuras, llamada tabla de procesos, con la información acerca del estado de cada tarea: el contador de programa, todos los registros, distribución de la memoria, archivos abiertos, estado de la tarea, y todo lo que deba guardarse cuando se pasa la tarea del estado de ejecución al de listo, de forma que pueda reiniciarse luego como si nada hubiera sucedido.

Como ejemplo podemos dar el contenido de la tabla de proceso de UNIX.

Manejo de procesos	*Manejo de memoria*	*Manejo de archivos*
Registros	Puntero al segmento de código	Máscara UMASK
Contador de programa	Puntero al segmento de datos	Directorio raíz
Program status word (psw)	Puntero al segmento BSS	Directorio de trabajo
Stack pointer	Condición de salida	Descriptores de los archivos
Tiempo en que la tarea comenzó	Condición de la señal	UID efectivo
Tiempo usado de CPU	Task ID	GID efectivo
Tiempo de CPU de la tarea hija	Proceso padre	Parámetros de llamada al sistema
Tiempo de la siguiente alarma	Grupo de procesos	Diversos bits de señalización
Puntero a la lista de espera de mensajes	UID real	
Bits de señal pendientes	UID efectivo	
Task ID	GID real	
Diversos bits de señalización	GID efectivo	
	Mapa de bits de señales	

Algunos de los campos de la tabla de procesos de UNIX

3.2. Comunicación entre procesos (IPC)

Los procesos frecuentemente desean comunicarse con otros procesos. Cuando un proceso del usuario desea leer el contenido de un archivo, éste debe indicar a la tarea de manejo de archivos cuál desea. Luego la tarea de manejo de archivos debe indicar al *driver* de disco el bloque que se pide. En pocas palabras, se necesita establecer comunicación entre procesos, preferentemente en forma bien estructurada sin emplear interrupciones. En lo que sigue se observarán algunos aspectos y contratiempos relacionados con la comunicación entre procesos o **IPC**
(interprocess communication).

3.2.1. Condiciones de concurso

En algunos sistemas operativos, los procesos que trabajan en conjunto con frecuencia comparten algún lugar de almacenamiento común donde cada uno pueda leer y escribir. El almacenamiento compartido puede ubicarse en la memoria principal o bien en un archivo compartido, esto no altera la naturaleza de los problemas que se presentan. Para ilustrar esto veamos el siguiente caso:

```
programa A                programa B

   .                .
   .                .
   .                .
   x  =  x + 1;          x  =  x+1;
   .                .
   .                .
```

Si los dos programas corren en paralelo, y x es una locación de memoria común, de la cual se conoce su valor inicial, no se puede determinar a priori, cual será el valor de *x* al culminar la ejecución de ambas tareas. Supongamos que *x* es cero, en el mejor de los casos *x* puede culminar valiendo dos. Pero puede acontecer que cuando la tarea A lee el valor de *x* ésta vale cero, en ese preciso momento el scheduler le cede el microprocesador a B, entonces B también ve en *x* el valor cero. Por tanto A y B pondrán en *x* el valor de uno.

Situaciones como ésta, donde dos o más tareas leen o escriben datos compartidos y el resultado final depende de cuál se ejecuta en un momento preciso, se denominan condiciones de concurso. La depuración de este tipo de programas no es algo sencillo en absoluto. La mayoría de las ejecuciones suelen resultar exitosas, en estos casos, pero de cuando en cuando sucede algo inexplicable. El rtKERNEL es uno de este tipo de programas.

3.2.2. Secciones críticas

La clave para prevenir el problema aquí donde interviene la memoria compartida, archivos compartidos, o todo aquel recurso que se comparte, consiste en evitar que

más de un proceso lea y escriba datos comunes simultáneamente. No existiría ningún problema si todas las tareas leen el dato y ninguna pretende alterarlo. Dicho en otras palabras, lo que se requiere es la **exclusión mútua** (una forma de asegurarse de que si una tarea esta accediendo a un dato común, los otros no puedan hacerlo). El problema anterior se presentó porque B quiso utilizar una de las variables compartidas antes de que A terminara con ella. La elección de operaciones primitivas adecuadas para lograr la exclusión mútua es un aspecto importante de diseño en cualquier sistema operativo y es un tema que se verá a continuación.

El problema de evitar condiciones de concurso también puede formularse en forma abstracta. Parte del tiempo un proceso permanece ocupado realizando tareas que no conducen a condiciones de concurso. Sin embargo, a veces un proceso puede estar accediendo variables comunes, o bien realizando algo que lleve a cuestiones de competencia. Esa parte del programa donde se accede a datos compartidos se denomina **sección crítica.** Si se pudieran arreglar las cosas para que dos tareas no estén en su sección crítica al mismo tiempo, se podrían evitar las condiciones de concurso.

Aunque este requisito impide que haya condiciones de concurso, no basta para hacer que los procesos paralelos cooperen en forma correcta y eficiente utilizando datos compartidos. Se necesita que se cumplan cuatro condiciones para tener una solución adecuada:

1. Nunca dos procesos pueden encontrarse simultáneamente dentro de sus secciones críticas.
2. No se hacen suposiciones acerca de las velocidades relativas de los procesos o del número de CPU's.
3. Ningún proceso suspendido fuera de la sección crítica debe bloquear a otros procesos.
4. Nunca un proceso debe querer entrar en forma arbitraria en su sección crítica.

3.2.3. Exclusión mútua

En esta sección examinaremos diversas proposiciones para lograr la exclusión mútua, de manera que mientras una tarea esté en su sección crítica, ningún otro proceso entre en su región crítica y ocasione problemas.

3.2.3.1. Desactivación de interrupciones

La solución más sencilla consiste que una tarea deshabilite (enmascare) todas las interrupciones antes de ingresar a su sección crítica y las rehabilite al salir. De esta forma se evita el cambio de contexto (dado que esto se realiza por medio de las interrupciones de clock o timer), con lo cual queda garantizado que otra tarea acceda a dicha sección.

La desventaja de este método consiste en que es imprudente darle al usuario la facultad de inhibir las interrupciones, dado que si la tarea en cuestión colapsa, junto con ella colapsaría el sistema. Además si la computadora tiene más de una CPU, este método no sirve.

Este método es adecuado como mecanismo general de exclusión mútua a nivel kernel, cuando por ejemplo este se encuentra actualizando listas o variables compartidas y quedaría en un estado inconsistente, si alguien accediera a ellas

simultáneamente. Este método es el empleado en el rtKERNEL (ver macro DISABLE() y ENABLE () que desactivan y activan interrupciones).

3.2.4. Con espera ocupada (busy wait)

3.2.4.1. Soluciones de software

Existen diversas soluciones de software al problema de la exclusión mútua, entre ellas la de Peterson, por lo general no carecen de complejidad y no son adecuadas para nuestro kernel, por lo tanto no serán enfocadas aquí. Estas soluciones figuran en todos los libros de sistemas operativos, a ellos remitimos los interesados en este tema.

3.2.4.2. Instrucción TSL

Esta solución debe estar implementada en hardware. Muchos microprocesadores, como la familia del i8086 cuentan con una instrucción llamada TEST AND SET LOCK (TSL) que opera como sigue. Esta lee el valor de una palabra de memoria y la copia a un registro y luego pone un valor distinto de cero en la misma posición de memoria. Todo esto se hace en una operación indivisible (nadie más puede acceder a la posición de memoria hasta tanto termine de ejecutarse la instrucción). La CPU que ejecuta la instrucción cierra el bus durante el transcurso de la misma.

El modo de emplear a esta instrucción es la siguiente:

```
enter_region:
    tsl register, flag
    cmp   register, #0
    jnz    enter_region
    ret

leave_region:
    mov   flag, #0
    ret
```

Cada tarea debe llamar a enter_region antes de entrar a su sección crítica y a leave_region al dejarla.

Cabe que esta instrucción consume tiempo de CPU mientras espera. Es una solución adecuada cuando se cuenta con un sistema con más de una CPU.

3.2.5. Sin espera ocupada

3.2.5.1. Semáforos

La contribución más importante a la comunicación entre procesos fue la introducida por Dijkstra en 1965 del concepto de semáforos, con las operaciones *wait* y *signal* definidas sobre ellos.

Un semáforo es un entero es un entero no negativo sobre el cual, aparte de su proceso de inicialización, puede actuarse sólo a través de las operaciones de *wait* y

signal. Estas operaciones actúan sólo sobre semáforos y su efecto se describe a continuación.

signal(s)

Su efecto consiste en incrementar el valor del semáforo s en uno, siendo ésta una operación considerada como indivisible. Es decir que dicha operación es atómica.

wait (s)

Su efecto consiste en decrementar el valor del semáforo *s* en uno en tanto que el resultado al que se llegue sea no negativo. Esta operación es también indivisible. La operación de *wait* representa un retardo en potencia ya que cuando actúa sobre un semáforo cuyo valor sea cero, el proceso que la ejecute podrá seguir sólo cuando algún otro proceso haya incrementado el semáforo en uno mediante una operación de *signal*. La indivisibilidad de esta operación significa que si el retardo mencionado afecta a varios procesos, sólo uno de ellos podrá seguir cuando el semáforo sea positivo. No se lleva a cabo suposición alguna sobre cuál será este proceso (en el rtKERNEL se desbloquea el proceso de más alta prioridad).

3.2.5.2. Monitores

Se trata de una primitiva de sincronización de alto nivel. Es un conjunto de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete. Los procesos pueden llamar a procedimientos en un monitor siempre que lo deseen, pero no podrán acceder las estructuras internas del mismo a través de rutinas declaradas fuera de él.

Los monitores tienen una propiedad importante que los hace útiles para lograr la exclusión mútua: sólo un proceso puede estar activo en un monitor en cualquier instante. Los monitores son una construcción de un lenguaje de programación, de forma que el compilador sabe como manejar la llamada a dichos procedimientos. Es tarea del compilador implementar la exclusión mútua en las captaciones del monitor.

Los monitores no se encuentran implementados en C, ni tampoco en el rtKERNEL, por lo que no ahondaremos más este tema.

3.2.5.3. Transmisión de mensajes

Un problema que se tiene con los monitores, y también con los semáforos, es que se diseñaron para resolver el problema de la exclusión mútua en una o más CPU que tienen acceso a una memoria común. Al colocar los semáforos en la memoria compartida y protegerlos con instrucciones TSL, se pueden evitar las competencias. Cuando migramos a un sistema distribuido que consta de múltiples CPU, cada una con su propia memoria, conectadas por una red de área local, estas primitivas se tornan inaplicables. La conclusión es que los semáforos son de un nivel muy bajo y los monitores no son utilizables salvo en contados lenguajes de programación. Además ninguna de estas primitivas ofrece intercambio de información entre máquinas.

La **transmisión de mensajes** viene a solucionar este problema. Este método de comunicación entre tareas hace uso de dos primitivas, *send()* y *receive()*, las cuales como los semáforos y a diferencia de los monitores, son llamadas al sistema más que

construcciones de un lenguaje. Como tales, pueden incluirse fácilmente en procedimientos de biblioteca, como

```
send ( destino, &mensaje );
```

y

```
receive ( fuente, &mensaje );
```

El primero envía un mensaje a un destino dado y el último recibe un mensaje de una fuente determinada (o de ninguna, si el receptor no tiene precaución). Si no está disponible ningún mensaje, el receptor podría bloquearse hasta que llegue alguno.

3.3. Deadlock (estancamiento)

Cuando varias tareas compiten por recursos es posible que se dé una situación en la que ninguno de ellos pueda proseguir debido a que los recursos que cada uno de ellos necesita estén ocupados por los otros. Esta situación se conoce con el nombre de *deadlock.* Es análoga al embotellamiento de tráfico que se produce cuando dos filas de autos que se desplazan en la misma dirección pero en sentidos opuestos intentan girar en el sentido de la hilera de autos contraria. El tráfico queda colapsado ya que cada fila de coches ocupa el espacio de calle que necesita la otra. El evitar los deadlocks o al menos limitar sus efectos, es una de las funciones de los istemas operativos.

Los estancamientos pueden ocurrir cuando a los procesos se les haya otorgado acceso exclusivo a dispositivos, archivos, etc. Para hacer lo más general posible el estudio de los estancamientos, nos referiremos a los objetos otorgados como **recursos**. Un recurso puede ser un dispositivo de hardware (por ejemplo una unidad de cinta) o una pieza de información (por ejemplo un registro lockeado en una base de datos). Una computadora normalmente tendrá muchos recursos diferente que pueden adquirirse. Para algunos recursos, pueden estar disponibles varios objetos idénticos, como tres unidades de cinta. Cuando se dispone de varios ejemplares de un recurso, cualquiera de ellos se puede utilizar para satisfacer cualquier solicitud del recurso. En resumen, un recurso es cualquier elemento que sólo puede ser empleado por un solo proceso en cualquier instante.

El orden de eventos que se requiere para utilizar un recurso es:

1. Solicitar el recurso

2. Utilizar el recurso

3. Devolver el recurso

Si el recurso no está disponible cuando se lo requiere, el proceso solicitante se ve forzado a esperar. En algunos sistemas operativos, el proceso se bloquea automáticamente cuando falla la solicitud de un recurso y se desbloquea cuando está disponible. En otros sistemas, la requisición falla con un código de error y corresponde al proceso solicitante esperar un poco y volver a intentar.

3.4. Modelo del estancamiento

El estancamiento se puede definir formalmente como sigue. Un conjunto de procesos se estanca si cada proceso del conjunto está esperando un evento que sólo otro conjunto del proceso puede provocar. Puesto que todos los procesos están en espera, ninguno de ellos podrá ocasionar nunca ninguno de los eventos que podrían desbloquear a alguno de los otros miembros del conjunto y todos los procesos sugerirán esperando indefinidamente.

En la mayoría de los casos, el evento que cada proceso está esperando es la devolución de algún recurso que es poseído corrientemente por otro miembro del conjunto. En otras palabras, cada miembro del conjunto de procesos estancados está esperando a que un recurso pueda ser liberado sólo por un proceso estancado. Ninguno de los procesos se puede ejecutar, ninguno de ellos puede liberar a ningún recurso y ninguno puede ser desbloqueado. El número de procesos y el número y tipo de recursos poseídos no son de importancia.

Coffman y otros (1971) demostraron que pueden cumplirse cuatro condiciones para que haya estancamiento:

1. Condición de exclusión mútua. Cada recurso se asigna por lo regular a un proceso o bien está disponible (recursos compatibles).
2. Condición de contención y espera. Los procesos que regularmente contienen recursos otorgados antes pueden solicitar nuevos recursos.
3. Condición de sin prioridad. Los recursos previamente otorgados no pueden extraerse por la fuerza de un proceso. Deben ser liberados en forma explícita por el proceso que los contiene.
4. Condición de espera circular. Debe haber una cadena circular de dos o más procesos, cada uno de los cuales esté esperando un recurso contenido en el siguiente miembro de la cadena.

3.5. Reentrada

Una función reentrante puede ser llamada más de una vez, sin corrupción de datos. Una función reentrante puede ser interrumpida en cualquier momento y reasumida luego sin pérdida de datos. Las funciones reentrantes usan variables automáticas o datos protegidos cuando se usan variables globales. Un ejemplo de una función reentrante es el siguiente:

```
void strcpy ( char *dest, char *scr )
{
    while  ( *dest++ = *src++ );
    *dest = NUL
}
```

Un ejemplo de una función no reentrante es la siguiente (Var1 es global).Desde que la función accede a una variable global, no es reentrante

```
void foo ( void )
```

```
{
.
.
Var1 += 23;
.
.
}
```

Pero puede hacerse reentrante protegiendo Var1 con un semáforo.

Los compiladores especificamente diseñados para software embebido típicamente proveen bibliotecas reentrantes. Hay que tener cuidado con este aspecto, dado que por lo general las bibliotecas pensadas para DOS no son reentrantes.

4. EL KERNEL (NÚCLEO) DEL SISTEMA

La principal interface entre el hardware de la máquina y el sistema operativo lo constituye el kernel, que corresponde al nivel más bajo del sistema operativo. La finalidad del núcleo es constituir un entorno adecuado donde puedan desarrollarse las distintas tareas.

4.1. Facilidades que se requieren en el hardware

4.1.1. Mecanismo de interrupciones.

Disponible en todos lo microprocesadores de hoy en día, por medio de ellas entraremos en el kernel y provocaremos el cambio de contexto.

4.1.2. Protección de memoria.

Cuando varias tareas se ejecutan concurrentemente es necesario proteger la memoria de cada una de ellas al acceso no autorizado por parte de cualquier otra. También es necesario proteger la zona de datos del kernel contra cualquier acceso indebido por parte del usuario. El i386 cuenta con los citados mecanismos, pero para usarlos es imprescindible pasarlo a modo protegido. El rtKERNEL 1.00 opera en modo real 8086, por lo cual no se han empleado dichos mecanismos en esta primera versión.

4.1.3. Repertorio de instrucciones reservadas

Por lo general los microprocesadores tienen al menos dos modos de ejecución: modo usuario y modo supervisor. En este último modo, el modo de más privilegio, se pueden ejecutar ciertas instrucciones reservadas como ser:

1. Habilitar y deshabilitar interrupciones.
2. Conmutar entre distintos procesos.
3. Acceder a los registros empleados por el hardware de protección de memoria
4. Llevar a cabo las entradas y salidas

Para disponer de estas facilidades en el i386 es necesario pasarlo a modo protegido, por lo tanto no se empleará el modo supervisor en el rtKERNEL, al menos en esta primera versión.

4.1.4. Reloj o timer de tiempo real

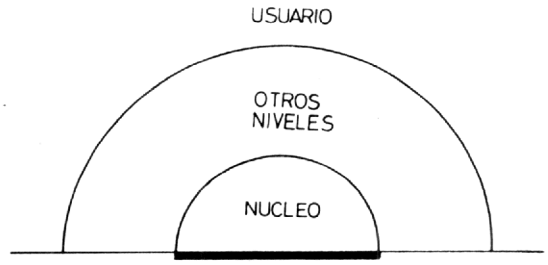
Es necesario asimismo contar con algún contador o reloj colgado de las interrupciones, es decir que produzca interrupciones periódicamente y con la facilidad de poder reprogramar dicho período. Estas operaciones se emplearán en caras a la implementaciones de las políticas de *scheduling*.

4.2. Esquema del núcleo

La relación que existe entre el núcleo y el resto del sistema se ilustra en la siguiente figura. La línea horizontal de la base del diagrama representa el hardware de la computadora; la parte más gruesa representa el set de instrucciones reservadas, cuyo empleo vamos a restringir al núcleo.

El kernel consta de tres bloques de programas:

1. el *controlador de interrupciones*, que lleva a cabo la gestión inicial de todas las interrupciones. En el rtKERNEL esta función se encuentra en el módulo de assembler, que constituye el único módulo que depende del microprocesador.
2. el *dispatcher*, que es quien conmuta el microprocesador entre las diferentes tareas
3. las primitivas de sincronización como ser el *wait* y el *signal.*



Estructura de nuestro sistema operativo ficticio

4.3. El dispatcher

La misión del *dispatcher* o *scheduler de bajo nivel* consiste en asignar el microprocesador a las distintas tareas del sistema. Se invoca cada vez que una tarea en curso no pueda continuar, o bien para determinar cual es la próxima tarea candidata a usar el microprocesador.

El funcionamiento del dispatcher es sencillo:

1. Sigue siendo el proceso en curso el más apropiado para ser ejecutado en este microprocesador? Si es así volver de la interrupción a la tarea en curso. De lo contrario...
2. Salvar el entorno volátil de la tarea en curso en su descriptor de proceso (en el rtKERNEL el OS_TCB).
3. Sacar de su descriptor de proceso el entorno volátil de la tarea más adecuada para ser ejecutada.
4. Transferir el control a la posición de memoria indicada por el contador de programa asociada a la nueva tarea.

Para poder determinar el proceso más adecuado para ser ejecutado, basta con ordenar todos los procesos ejecutables de acuerdo con algún criterio de prioridad. La asignación de estas prioridades a los distintos procesos es tarea del scheduler de alto nivel. En el rtKERNEL las prioridades las asigna el usuario.

El dispatcher en el rtKERNEL se encuentra plasmado en _new_tick_isr y las funciones a las que llama, en particular os_int_exit().

Vamos a distribuir en nuestro kernel, los descriptores de tarea a todos los procesos ejecutables (listos) en una lista circular ordenada por prioridades decrecientes. Existirá un puntero que vaya girando sobre esta lista y vaya apuntando al descriptor correspondiente a la tarea más apta a ser ejecutada.

4.3.1. Implementación del wait *y del* signal

A modo de ejemplo veamos como se implementan estas primitivas.

La operación de *wait* lleva implícita la idea de que las tareas se quedan bloqueados cuando el semáforo vale cero, y son liberados cuando una operación de *signal* incrementa este valor en uno. La forma natural de implementar esto es asociando a cada semáforo una cola de semáforo que contendrá las tareas bloqueadas en éste. En el rtKERNEL dicha cola está apuntada por **os_tcb_blk_task* dentro de *OS_ECB*. Cuando una tarea hace un wait sobre un semáforo que estaba a cero, entonces se la quita de la lista de tareas listas y se la coloca en la lista del semáforo. Recíprocamente, cuando se ejecuta una operación de *signal* sobre un determinado semáforo, se saca una tarea de la cola del mismo (a menos que la cola este vacía) y se lo hace nuevamente ejecutable. Debe implementarse, pues, un semáforo en base a un entero y a un puntero asociado a una cola (que puede ser nulo).

Llegados a este punto la implementación sería como sigue:

wait(s) : if (s != 0)
    s--l
  else
    añadir el proceso a la cola del semáforo y hacerlo no ejecutable

signal (s) : if (cola vacía)
    s++
  else
    sacar una tarea de la cola del semáforo y hacerla ejecutable.

Es de destacar que no hace falta incrementar el semáforo dentro de *signal* si se libera un proceso, ya que éste debería decrementarlo otra vez tan pronto completara su operación de *wait().*

4.4. Políticas de scheduling

Cuando existen varias tareas factibles de ser ejecutadas (listas), el sistema operativo debe decidir cual ejecutar primero y el algoritmo al cual apela el scheduler se llama justamente algoritmo de scheduler. Retrocediendo a los días de los sistemas por lotes con entradas en forma de imágenes de tarjetas en una cinta magnética, el algoritmo de planificación era simple: simplemente se ejecutaba el siguiente trabajo en la cinta. Hoy en día estos algoritmos son bastante más complejos.

Antes de analizar algoritmos de implementación específicos, debemos cuales son los requisitos de un buen algoritmo de planificación:

1. Imparcialidad: asegurar que cada proceso tenga la parte que le corresponde de la CPU.
2. Eficiencia: mantener la CPU ocupada el 100% del tiempo

3. Minimizar el overhead del algoritmo de scheduling

La estrategia de permitir que un proceso en ejecución sea suspendido temporalmente, se lo llama planificación por prioridad (preemteved) contrasta con el método de ejecución por terminación (no preemteved o cooperativo) en el cual un proceso cede el control porque así lo decide. El rtKERNEL funciona de la primer forma y el WINDOWS 3.1 y anteriores de la segunda. Si bien la primer forma conlleva el overhead del scheduler, no permite que un proceso se adueñe indefinidamente de la CPU, como ocurre en el Windows.

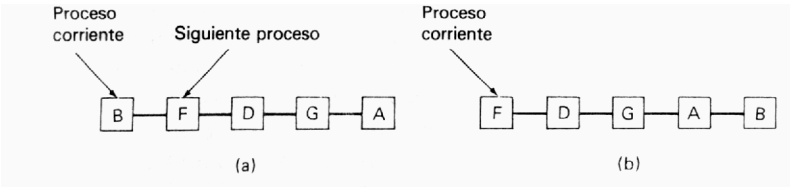
4.4.1. Round Robin

Uno de los algoritmos más antiguos, simples, imparciales y más ampliamente utilizados es el Round Robin, también llamado torneo. A cada tarea se le asigna un intervalo de tiempo, llamado **quantum** (time slice), en el cual se permite su ejecución. Al término de este se cambia de contexto y se le asigna la CPU a otro proceso. Si el proceso se bloquea, o bien finaliza antes de que expire su quantum de tiempo, el switcheo se hace en ese preciso instante. El round robin es fácil de implementar, todo lo que hace falta es hacer una lista circular con los procesos listo, y un puntero que gire alrededor de la misma. Cuando el quantum expira se adelanta dicho puntero una posición, y se cambia de contexto.

Un aspecto interesante del torneo es la duración del time slice. El cambio de una tarea a otra requiere cierta cantidad de tiempo para efectuar la administración (guardar y cargar registros y mapas de memoria, actualizar diversas tablas y listas). Supóngase que este cambio de contexto, involucra 5 mseg. y el quantum se fija en 20 mseg. Con estos parámetros, después de realizar 20 mseg. de trabajo útil, la CPU tendrá que gastar 5 mseg. en el cambio de proceso. El 20% del tiempo de la CPU se desperdiciará en sobrecarga administrativa.

Para mejorar la eficiencia de la CPU, el quantum se podría fijar por ejemplo en 500 mseg, de esta forma el tiempo que se pierde es menor al 1%, pero disminuye considerablemente el tiempo de respuesta.

Por lo tanto habrá que adoptar un compromiso razonable entre ambos extremos.



Planificación de un torneo. (a) Lista de procesos ejecutables. (b) Lista de p
cesos ejecutables después de que se termina la cantidad de *B*.

4.4.2. Planificación por prioridad

La planificación de un torneo hace la suposición implícita de que todos los procesos tienen la misma prioridad.

A menudo conviene agrupar las tareas en clases de prioridad y utilizar la planificación por prioridad entre las clases, pero la planificación de round robin en cada clase. La figura siguiente muestra un sistema con cuatro niveles de prioridad (tal como el rtKERNEL). El algoritmo de planificación es como sigue: en tanto haya

procesos ejecutables en la prioridad más alta es decir la 4 (en el rtKERNEL corresponde a la cero) se ejecutan estos en forma de torneo, y nunca nos preocupamos por las tareas de prioridad más bajas. Si las clases 4 y 3 permanecen vacías, ya que los procesos de las mismas o bien culminaron, o bien se bloquearon en algún evento, entonces se aplica el torneo en el nivel 2 y así sucesivamente.

Este es el algoritmo implementado en el rtKERNEL.

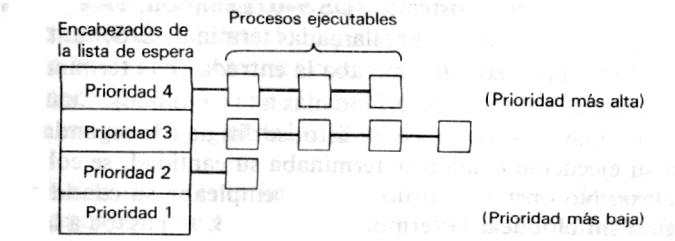


Fig. Algoritmo de planificación con cuatro clases de prioridad.

5. INTRODUCCIÓN A LOS SISTEMAS DE TIEMPO REAL

Tanto como las computadoras se tornan pequeñas, rápidas, confiables y baratas, tanto crece su rango de aplicación. Construidas inicialmente para resolver ecuaciones, su influencia se ha extendido en todos los caminos de la vida, desde lavarropas a control del tráfico aéreo. Una de las áreas más rápidas de expansión, es aquella que no requiere información de procesamiento para lograr su función primordial. Un microprocesador en un lavarropas es un buen ejemplo de tal sistema. Aquí la función primordial es lavar la ropa, pero dependiendo del tipo de ropa, se ejecutan diferentes programas de lavado. Este tipo de aplicaciones se denominan generalmente de *tiempo real* o *embebidas.*

5.1. Definición de sistemas de tiempo real

Antes de seguir es necesario definir la frase "sistemas de tiempo real" más precisamente. Hay muchas interpretaciones de la naturaleza exacta de un sistema de tiempo real; sin embargo todas tienen en común la noción de tiempo de respuesta ,el tiempo que le toma al sistema generar una salida desde una entrada determinada. El Diccionario de Computación de Oxford da la siguiente definición al respecto:

> Cualquier sistema en el cual el cual el tiempo en el cual se produce la salida es significante. Esto es usual porque la entrada corresponde a algún movimiento en el mundo físico, y la salida tiene que relacionarse a tal movimiento. El retraso desde el tiempo de la entrada al tiempo de la salida debe ser suficientemente pequeño para una respuesta aceptable.

Aquí la expresión respuesta aceptable se toma en el contexto del sistema total. Por ejemplo, en un sistema de control de misil, la salida se requiere dentro de pocos milisegundos, mientras que en una línea de ensamblado de autos la respuesta se puede requerir dentro del orden del segundo.

Young (1982) define un sistema de tiempo real de la siguiente forma:

> cualquier actividad o sistema de procesamiento de información el cual debe responder a un estímulo de una entrada externa en un período finito y especificado.

En el caso más general de ambas definiciones cubren un gran ancho de actividades de computación. Por ejemplo, un sistema operativo como UNIX puede ser considerado de tiempo real cuando los usuarios entran un comando y esperan una respuesta dentro de pocos segundos. Obviando el descontento de los usuarios, afortunadamente no ocurrirá ningún desastre si no se cumple esta premisa. Este tipo de sistemas puede ser separado de aquellos donde una demora en la respuesta puede

considerarse como mala o incorrecta. Efectivamente, para algunos, es en este aspecto donde se distingue un sistema de tiempo real de otros donde la respuesta en el tiempo es importante pero no crucial. Consecuentemente, *la corrección de un sistema de tiempo real depende no sólo de los resultado lógicos de la computación, sino también el tiempo en el que se producen los resultados.* Los especialistas en el campo de sistemas de computación de tiempo real distinguen entre **sistemas *hard* y *soft*** de tiempo real. Los primeros son aquellos donde es absolutamente imperativo que la respuesta ocurra dentro del plazo de tiempo específico. Los segundo son aquellos donde la respuesta en el tiempo es importante pero el sistema seguirá funcionando correctamente si los tiempos ocasionalmente no se cumplen. Los sistemas *soft* pueden a su vez distinguirse de los interactivos donde no hay plazos estipulados. Por ejemplo, un sistema de control de vuelo de una aeronave de combate es un sistema *hard* porque un plazo incumplido puede conducir a una catástrofe, mientras que un sistema de adquisición de datos para un control de proceso porque puede definirse un intervalo de tiempo al término del cual se monitorea el sensor de entrada, pero puede tolerar demoras intermitentes.

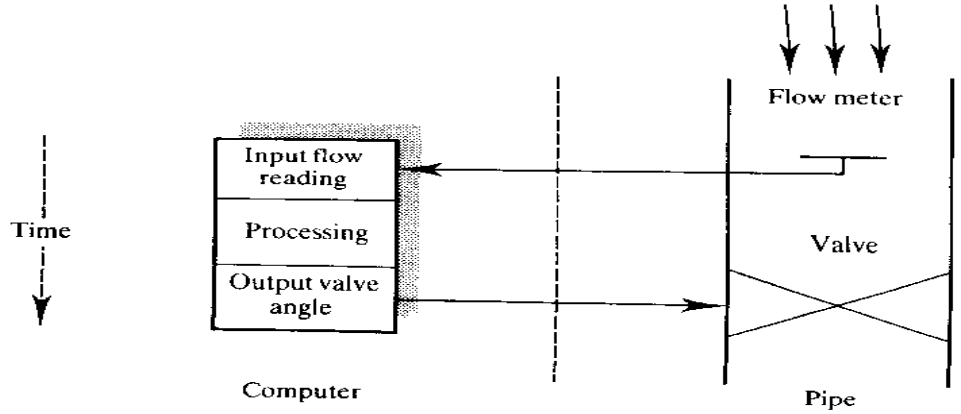
En cualquiera de ambos la computadora usualmente se interfacea directamente a algún equipo físico y se dedica a monitorear o controlar la operación del mismo. Una clave de todas estas aplicaciones es el rol de la computadora como un componente de procesamiento de información dentro de un sistema grande de ingeniería. Es por ello que tales aplicaciones se conocen como *sistemas embebidos*. El término *tiempo real* o *embebido* se usa en forma intercambiable en todo este informe.

5.2. Ejemplos de sistemas de tiempo real

Teniendo definido que significa un sistema embebido daremos alguno ejemplos.

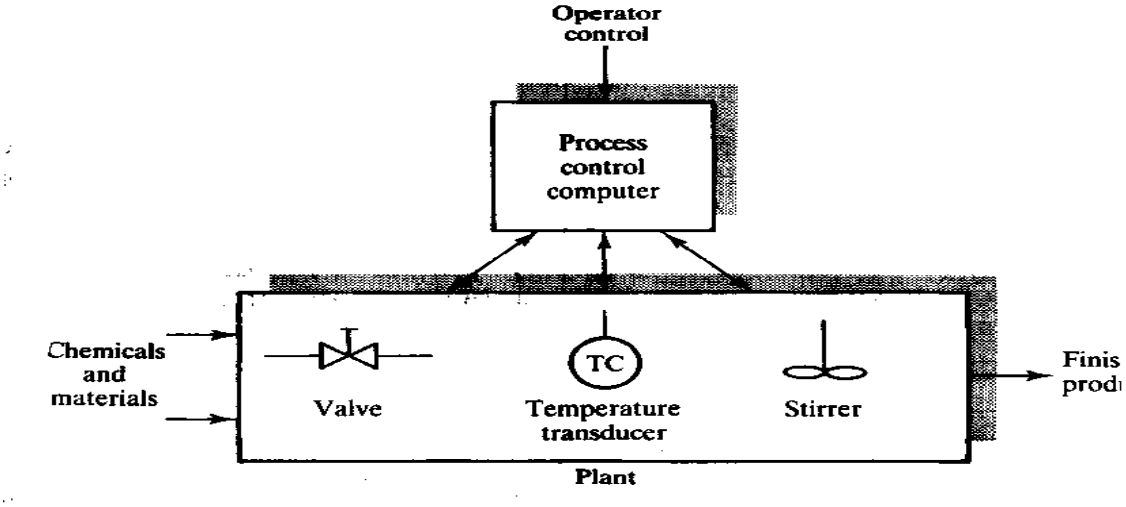
5.2.1. Control de procesos.

EL primer uso de un microprocesador en un sistema grande de ingeniería ocurrió en el área de control industrial al principio de 1960. Hoy en día, el uso de microprocesadores es la norma. Considere como ejemplo, la figura donde una computadora realiza una única actividad, asegurar el flujo constante de líquido en un tubo controlado por una válvula. Detectando un incremento en el flujo se debe responder alterando el ángulo de la válvula. Esta respuesta debe ocurrir dentro de un período finito si se desea que no se sobrecargue el equipo en el extremo del tubo. Note que en la respuesta puede haber involucrado un cálculo complejo en orden de calcular el nuevo ángulo.



Sistema de control de flujo

Este ejemplo muestra uno de los componentes de un gran sistema de control. La figura siguiente ilustra el rol de una computadora de tiempo real embebida en un entorno completo de control de proceso. La computadora actúa con el equipo usando sensores y actuadores. Una válvula es un ejemplo de un actuador y un transductor de presión o temperatura es un ejemplo de un sensor. Un transductor es un dispositivo que genera una señal eléctrica proporcional a la cantidad física a ser medida. La computadora controla la operación de los sensores y actuadores para asegurar que se realiza una correcta operación de planta en el tiempo apropiado. Donde sea necesario se insertarán conversores analógicos/digitales y viceversa entre la computadora y el proceso controlado.

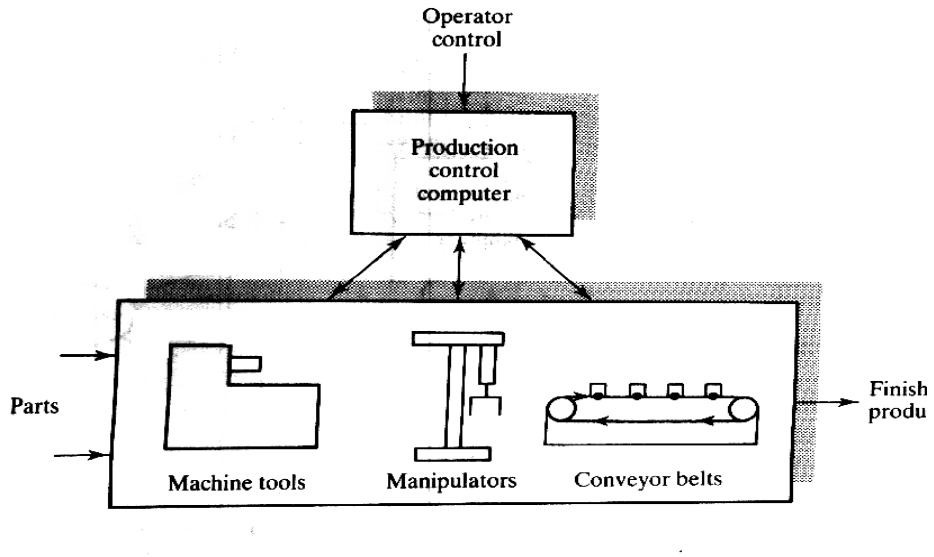


Un sistema de control de proceso

5.2.2. Industria

El uso de computadoras en la industria se ha convertido escencial en los últimos años en orden de decrementar los costos de producción y aumentar la productividad.

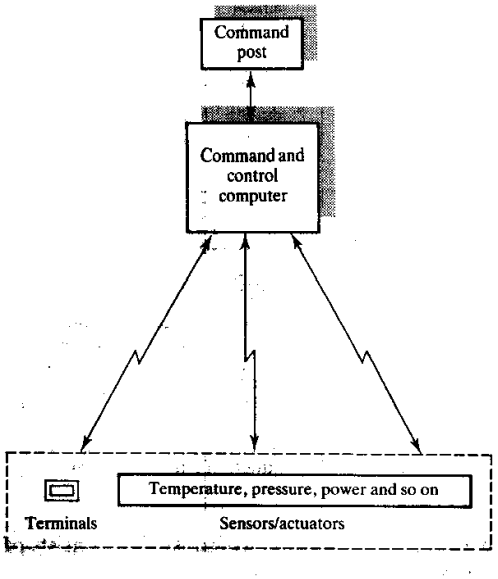
Las computadoras permitieron la integración de un proceso completo de industrialización. Es en el área de control de producción donde los sistemas embebidos son mejor ilustrados. La figura siguiente representa, en forma de diagrama, el rol de computadora como control de la producción en un proceso industrial. El sistema consiste de una variedad de dispositivos tal como máquinas herramientas, manipuladores y cintas transportadoras, todos los cuales necesitan ser controlados y coordinados por una computadora.



Un sistema de control de la producción

5.2.3. Comunicación, comando y control

A pesar de que *comunicación, comando y control* es un término militar que se aplica en un gran rango de aplicaciones dispares con similares características. Por ejemplo, una reservación de pasajes de aerolínea, centros médicos automáticos de cuidado de pacientes, control de tráfico aéreo y control de cuentas de banco remota. Todos estos sistemas consiste de un complejo conjunto de políticas, dispositivos de recolección de información y procedimientos administrativos los cuales habilitan las decisiones a ser soportadas, y proveen los medios por los cuales pueden ser implementadas. A veces, la información recolectada por los dispositivos e instrumentos se requieren para implementar decisiones sobre una gran área geográfica. La figura siguiente representa tal sistema



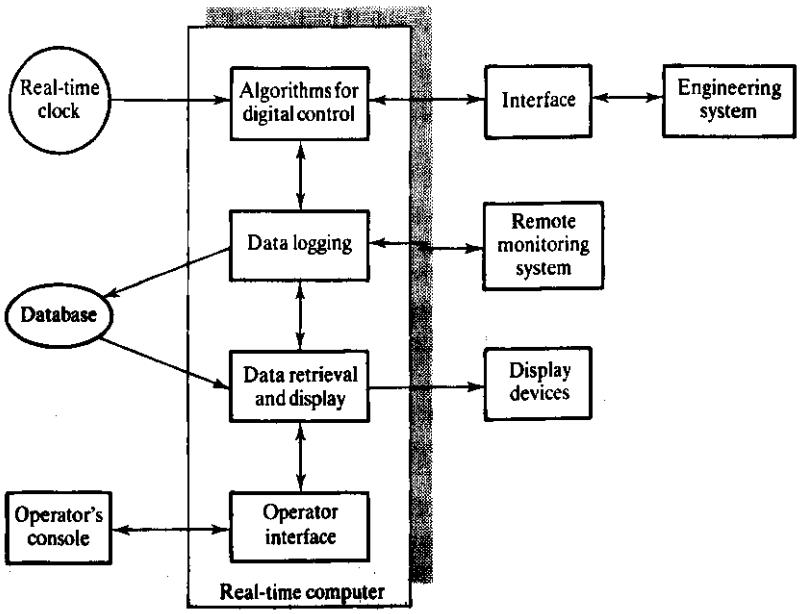
Un sistema de control y comando

5.2.4. Sistemas embebidos generalizados

En cada uno de los ejemplos mostrados la computadora es interfaceada directamente al equipo físico en el mundo real. En orden de controlar estos dispositivos del mundo real , la computadora necesita muestrear parámetros en intervalos regulares, por lo cual se requiere un reloj de tiempo real. Usualmente también existe un operador humano en una consola que permite una intervención manual. El operador es mantenido informado constantemente del estado del sistema por gráficos de varios tipos.

También se graba un archivo histórico con los cambios de estado del sistema para ser analizado luego en caso de falla, o para proveer información para propósitos administrativos. En efecto, esta información se usa para aportar decisiones día a día sobre el sistema corriendo. Por ejemplo, en una planta química y en un proceso industrial, el monitoreo de la planta es escencial para maximizar las ventajas económicas más que simplemente maximizar la producción. Las decisiones concernientes a la producción de una planta pueden tener serias repercusiones para otras plantas en sitios remotos, particularmente cuando el producto de un proceso es usado como material de alimentación para otra.

Un sistema embebido típico, puede ser representado, en consecuencia como en la figura siguiente. El software que controla las operaciones del sistema puede ser escrito en módulos los cuales reflejan la naturaleza física del entorno. Usualmente habrá un módulo que contiene los algoritmos necesarios para el control físico de los dispositivos, un módulo responsable de grabar los cambios de estado, un módulo para mostrar dichos cambios al operador y un módulo para interactuar con éste. En este caso cada módulo puede ser asimilado a una tarea de nuestro kernel.



Un sistema típico embebido

5.3. Características de un sistema de tiempo real

Un sistema de tiempo real puede tener varias características especiales (inherentes o impuestas) las cuales se identifican en las secciones siguientes. Claramente, no todos los sistemas de tiempo real exhibirán todas estas características.

5.3.1. Tamaño y complejidad

Es frecuente decir que la mayoría de los problemas asociados al desarrollo de software son aquellos relacionados con el tamaño y la complejidad. Escribiendo programas pequeños no presentan problemas al diseñados, codificados, mantenidos y comprendidos por una única persona. Si la persona deja la empresa o institución que está usando el software, luego alguien más puede aprender el programa en un período de tiempo relativamente corto. En efecto, para estos programas se aplica aquello de que lo *pequeño es hermoso*.

Desafortunadamente no todo el software es pequeño. Lehman y Beladay (1985) hicieron un intento para caracterizar a los sistemas grandes, rechazando la simple y quizás noción intuitiva de que el tamaño es proporcional al número de instrucciones, líneas de código, o módulos integrantes de un programa. En su lugar, ellos relacionan el tamaño a la *diversidad* y la cantidad de diversidad. Indicadores tradicionales, son el número de instrucciones y el desarrollo detrás de ellas, justamente constituyendo cotas de diversidad.

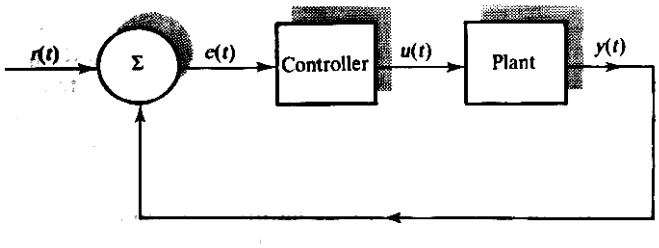
> La diversidad es aquella necesidad y actividad en el mundo real y su reflexión en un programa. Pero el mundo real está continuamente cambiando. Está evolucionando. Así también las necesidades y actividades de la sociedad. Así los programas grandes, son como sistemas complejos, deben evolucionar continuamente.

Los sistemas embebidos por definición deben responder a los eventos del mundo real. La diversidad asociada con estos eventos debe ser catalogada, para que los programas no aumenten indiscriminadamente de tamaño. Inherente a la noción de tamaño está la noción de *cambio continuo.* El costo de rediseñar o reescribir software respondiendo a los requerimientos de cambio continuo del mundo real es prohibitivo. Por ello, los sistemas de tiempo real padecen el mantenimiento constante y mejoramiento durante su vida útil. Ellos deben ser extensibles.

A pesar de que estos sistemas son frecuentemente complejo, las características provistas por los sistemas operativos de tiempo real dan la posibilidad de escindir sistemas complejos en piezas pequeñas.

5.3.2. Manipulación de número reales

Como ha sido visto anteriormente muchos sistemas de tiempo real involucran el control de alguna actividad ingenieril. La figura siguiente ejemplifica un simple sistema de control. La entidad de control, la planta, tiene un vector de salida variable *y(t)* que varía en el tiempo. La salida se compara con la señal de referencia deseada *r(t)* para producir una señal de error *e(t).* El controlador usa este vector de error para cambiar las variables de entrada a la planta *u(t).* Para un sistema simple puede tratarse de un dispositivo analógico trabajando con una señal continua.



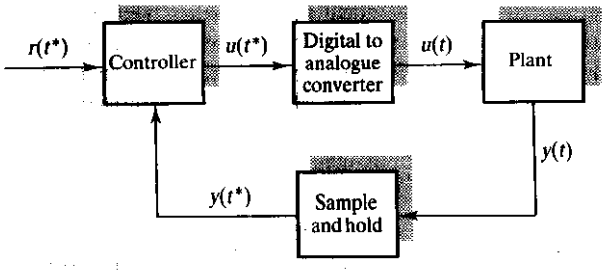
Un controlador simple

La figura anterior ilustra un control realimentado. Este es el más común forma de realimentación. En orden de calcular que cambio se debe hacer a las variables de entrada, así se produce un efecto deseable en el vector de salida, es necesario contar con un modelo matemático de la planta. La obtención de estos modelos corresponde a la ingeniería de control de procesos. Frecuentemente la planta se modela como un sistema de ecuaciones de primer orden, estas ligan la salida del sistema con el estado de la planta y las variables de entrada. Cuando se produce un cambio en la salida, se deben resolver estas ecuaciones para obtener los valores de entrada correspondientes. La mayoría de los sistemas físicos exhiben inercia, así los cambios no son instantáneos. Un requerimiento de tiempo real para moverse a un nuevo estado dentro de un período fijo, le sumará complejidad al modelo matemático y al físico. El hecho de que en realidad, las ecuaciones lineales de primer orden son sólo aproximaciones a las características actuales del sistema, también complica las cosas.

Gracias a estas dificultades, la complejidad del modelo y los número de entradas y salidas distintas (pero no independientes), la mayoría de los controladores se implementan como computadoras. La introducción de un componente digital en el sistema cambia la naturaleza del ciclo de control. La figura siguiente es una adaptación del modelo de la figura anterior. Los ítems marcados con * representan

ahora valores discretos. El sampleado se realiza por un convertidor a/d controlado por la computadora.

Dentro de la computadora las ecuaciones diferenciales se pueden resolver por técnicas numéricas, sin embargos los algoritmos mismos deben ser adaptados para contemplar el hecho de que las salidas de la planta ahora se samplean. El diseño de tales algoritmos escapa a los objetivos de este informe, sin embargo la implementación de tales algoritmos nos concierne. Ellos pueden ser matemáticamente complejos y requerir un alto grado de precisión. Un requerimiento fundamental de la programación de tiempo realla es pues, la habilidad de manipular números reales o de punto flotante.



Controlador simple digitalizado

5.3.3. Confiabilidad y seguridad

Cuanto más la sociedad relegue el control de sus funciones vitales a las computadoras, más imperativo se torna que las computadoras no fallen. La falla de un sistema de transferencia automática entre bancos puede conducir a la pérdida irremediable de millones de pesos. La falla en un sistema generador de energía eléctrica puede conducir a la falla en un sistema de soporte de vida en una unidad de terapia intensiva. Un proceso prematuro de *shutdown* de una planta química pude conducir a un daño irreparable del entorno. Estos ejemplos dramáticos ilustran que el hardware y el software deben ser confiables y seguros. Aún en entornos hostiles, tal como aquellos encontrados en aplicaciones militares, debe ser posible diseñar e implementar sistemas los cuales fallen en una forma controlada. Además donde se requiere la interacción de un operador, se de be tener especial cuidad en el diseño de la interface para minimizar los efectos de un error humano.

El gran tamaño y complejidad de los sistemas de tiempo real exacerban el problema de la confiabilidad. No sólo se deben esperar dificultades inherentes a la aplicación tomada en cuenta, sino aquellas introducidas por el mal diseño de software.

5.3.4. Control concurrente de los componentes del sistema

Un sistema embebido tenderá a consistir de computadoras y varios elementos externos consistente con el cual el programa de computación debe interactuar simultáneamente. Estos elementos deben existir en paralelo. En nuestro ejemplo típico la computadora debe interactuar con actividades concurrentes tales como robots, cintas transportadoras, sensores actuadores y los dispositivos de display, la consola del operador, el archivo histórico y el clock de tiempo real. Afortunadamente la velocidad de las computadoras modernas es tal que usualmente estas acciones pueden ser

conformadas secuencialmente, pero dando la ilusión de concurrencia. En otros sistemas embebidos puede no ser este el caso, por ejemplo, donde la información es colectada y procesada en varios sitios distantes geográficamente, o donde la respuesta de tiempo de los componentes individuales no puede ser obtenida por una computadora única. En este caso es necesario considerar sistemas embebidos distribuidos.

El mayor problema asociado con la producción de software para sistemas que exhiben concurrencia es como expresar la concurrencia en la estructura del programa. De aquí la necesidad de un sistema operativo multitarea en tiempo real.

5.3.5. Facilidades de tiempo real

Como hemos visto el tiempo de respuesta es crucial en cualquier sistema embebido. Desafortunadamente es difícil diseñar e implementar sistemas los cuales garanticen que la salida apropiada será generada en el tiempo apropiado bajo todas las condiciones posibles. Para hacer esto posible se hace uso de todos los recursos en todos los momentos que es posible. Por esta razón los sistemas de tiempo real, son usualmente construidos usando procesadores con considerable capacidad disponible, asegurándonos que el comportamiento de peor caso, no produce demoras en los períodos de operación críticos.

Dando una adecuada potencia de procesamiento, el programador debería estar habilitado para:

- Especificar momentos en los cuales las acciones deben ser realizadas.
- Especificar momentos en los cuales las acciones deben completarse
- Responder a situaciones donde todos los requerimientos de tiempos no pueden encontrarse.
- Responder a situaciones donde los requerimientos de tiempos son cambiados dinámicamente (cambio de modo).

Estas son llamadas facilidades de control de tiempo real. Ellas habilitan al programa a sincronizar con el tiempo mismo. Por ejemplo, con algoritmos de control digital es necesario monitorear, leyendo de los sensores en ciertos momentos del día, por ejemplo, 2pm, 3pm y así siguiendo, a intervalos regulares, por ejemplo 5 segundos. Como resultado de estas lecturas otras acciones serán efectuadas. Por ejemplo, en una estación eléctrica de potencia es necesario incrementar la potencia desde las 5pm de lunes a viernes a los consumidores domésticos.

Un ejemplo de cambio de modos se puede encontrar en un sistema aéreo de control de aire. Si un avión ha experimentado despresurización, allí hay una necesidad inmediata para que todos los recursos de la computadora se aboquen a la emergencia.

En orden de encontrar el tiempo de respuesta es necesario para el comportamiento del sistema que sea determinístico (y más aún predecible).

5.3.6. Interacción con las interfaces de hardware.

La naturaleza de los sistemas embebidos requieren que los componentes de la computadora interactuen con el mundo externo. Hay una necesidad de monitorear sensores y controlar actuadores para una gran variedad de dispositivos del mundo real. Estos dispositivos interfacean a la computadora por medio de registros o memoria de

entrada/salida y dependen de la computadora. Los dispositivos también pueden generar interrupciones en orden de conseguir que el procesador atienda ciertas operaciones, que se efectuarán con error si no se encuentran determinadas condiciones.

En el pasado, la interface a dichos dispositivos se dejaba bajo el control del sistema operativo, en el caso de los sistemas de tiempo real, por una cuestión de tiempos nos vemos obligados a manejar dichos dispositivos directamente.

Esta página permanece intencionalmente en blanco.

6. SISTEMAS OPERATIVOS EN TIEMPO REAL

Los sistemas de tiempo real son aquellos en los cuales la eficiencia del sistema depende no sólo de los resultados lógicos de los procesos, sino también del tiempo que insume la obtención de los resultados. Típicamente consisten de un *sistema controlado* y un *sistema controlador*.

En los sistemas de tiempo real, una respuesta inexacta pero rápida puede ser más importante que una lenta, pero exacta.

La información proporcionada a la computadora debe ser consistente con el estado actual de su entorno, caso contrario las acciones del sistema controlador pueden ser desastrosas. Esto torna imprescindible el monitoreo periódico y el procesamiento puntual de la información recibida.

Las restricciones más comunes en cuanto a tiempo para las tareas son *periódicas* o bien *aperiódicas.* En el primer caso una tarea tiene que comenzar dentro de un dado intervalo de tiempo y terminar dentro de otro bien especificado. En el segundo, la tarea debe ejecutarse una vez por período. Las tareas periódicas se emplean generalmente para monitorear variables externas, por ejemplo la temperatura de un horno y las aperiódicas por lo general surgen de un evento dinámico.

El hecho de ser rápido en promedio no garantiza que se cumpla un determinado plazo. Si un sistema de tiempo real puede demostrar ser capaz de cumplir sus plazos (utilizando un análisis del peor caso, más que uno de caso promedio), se dice que el sistema es predecible. Por ejemplo consideramos un algoritmo de control de temperatura que atraviesa un proceso químico sensible. Si el algoritmo se ejecuta dentro de su límite de tiempo predefinido cada instancia excepto una, y esa ejecución demorada permite que el proceso se torne exotérmico (es decir, que corra incontroladamente), ¿es un diseño exitoso de control?.

El requerimiento para las tareas críticas es que todas ellas deberían cumplir con sus plazos (un 100% de garantía), sujeto a suposiciones de probables fallas y cargas de trabajo.

Por ejemplo, debería evitarse en lo posible la utilización de memoria virtual, a causa de la sustancial impredictibilidad que ésta provoca, a menos que las tareas más importantes y sus datos se hagan inamovibles de memoria.

6.1. Atributos de los SOTRs

6.1.1. Determinismo o predictibilidad

Es la tendencia que tiene un sistema para llevar a cabo una operación en un período de tiempo bien definido o determinado. Un sistema totalmente determinista realiza operaciones en el mismo tiempo, cada vez, independiente de las condiciones que lo rodean.

Una de las características clave de los SOTRs es el grado de comportamiento determinista. El tiempo máximo durante el cual se demora una interrupción de dispositivo de alta prioridad se denomina LRI (latencia de reconocimiento de interrupción). Un SO convencional puede tener una LRI extensa de hasta varios milisegundos, en tanto que un SOTR típicamente tendrá un límite superior que va desde algunos microsegundos hasta un milisegundo.

Notemos que la diferencia de determinismo entre un SOTR y un SO convencional es en gran parte una cuestión de grados. Es así con cada característica de los SOTRs.

6.1.2. Sensibilidad

Es la habilidad que tiene un sistema de responder a un hecho con rapidez. Debido a que un hecho sincrónico, desencadena a una serie de operaciones predeterminadas cuya velocidad es importante en cualquier sistema, la sensibilidad a hechos sincrónicos no es una clave entre los sistemas en tiempo real y los que no lo son. Los hechos asincrónicos -en particular las interrupciones de I/O- representan el área en donde los sistemas de tiempo real tienen requerimientos más rigurosos, ya que estos interactuan con los sistemas externos vía las E/S.

Los tiempos de respuesta a una interrupción tienen varios componentes importantes. El primero es el tiempo máximo que se puede demorar el reconocimiento de una interrupción. El segundo de ellos es el tiempo requerido para manejar inicialmente la interrupción y comenzar la ejecución de la RSI (rutina de servicio de interrupción). Si la RSI involucra una tarea de usuario, y por ende un cambio de contexto, entonces el tiempo máximo de demora para iniciar y luego poner en funcionamiento el cambio de contexto se convierte en un factor de respuesta significativo. El tercer componente es el tiempo necesario para atender dicha interrupción. El cuarto, es el efecto que provocan las interrupciones anidadas sobre el tiempo de respuesta del sistema.

6.1.3. Control de usuario

Comienza con el concepto de prioridades fijas específicas para usuarios al ejecutar un comando desde el Shell del SO. El hecho de que el sistema le permita al usuario especificar las prioridades relativas de las tareas, también le permite especificar el rendimiento y los objetivos de sensibilidad sobre una base más fina. Por ejemplo se le puede suministrar más prioridad a un proceso que monitorea una variable externa de mucho peso, y menos prioridad al task que procesa la misma.

Con frecuencia, los SOTRs le dan al usuario la *capacidad privilegiada*, o sea el derecho de un proceso de usuario de pasar a modo supervisor, el modo que está estrictamente prohibido en los SO convencionales de tiempo compartido, excepto para el mismo kernel.

Esto le permite al usuario deshabilitar por completo el sistema de interrupciones, para asegurarse que una operación crítica en tiempo se ejecute sin perder el control de la CPU.

QNX brinda 32 niveles de privilegio de ejecución, con posibilidad de elegir algoritmo de scheduling y tres niveles de protección de hardware, relacionado con la familia de Intel.

El rtKERNEL brinda 4 niveles de prioridad o de privilegio de ejecución.

OS/2 da 4 niveles brinda 4 niveles de prioridad.

6.1.4. Confiabilidad

En un SOTR significa que el sistema puede correr continuamente durante períodos muy largos -inclusive años- sin fallar ni degradarse.

Muchos sistemas de tiempo real operan bajos severos requerimientos de confiabilidad; esto es, si ciertas tareas denominadas críticas faltan a sus plazos puede ocurrir una catástrofe.

Muchas veces los usuarios de DOS hacen que sus máquinas se cuelguen, de una manera u otra. Esto es impensable en un sistema de tiempo real, ya que un rebooteo

puede ser fatal, dado que estos sistemas controlan los sistemas del mundo real y pueden conducir a pérdidas monetarias cuantiosas, en el caso de una planta fabril.

Confiabilidad también significa asegurarse que los procesos de tiempo real críticos puedan ubicar todos sus recursos a tiempo, de tal manera que las tareas puedan cumplirse sin demora alguna.

6.1.5. Operación por falla de software o hardware

Es un concepto que resulta del empleo de computadoras falibles. Todos los sistemas de computadoras, inclusive aquellos con software a prueba de fallas, a veces fallan. Un SO moderno tal como el UNIX, realiza una operación de emergencia (operación de shutdown) cuando detecta corrupción o inconsistencia en la información del Kernel. Esta operación puede bajar la memoria a disco para su posterior análisis, avisando por consola de la falla y terminando la ejecución del sistema.

Un SOTR, en cambio no puede darse el lujo de derogar el sistema. En cambio, cuando se detecta corrupción en los datos del kernel, se toman los recaudos para continuar la ejecución, tal vez con alguna degradación en la performance o en el grado de servicio integral. El proceso de usuario notificado puede llevar a cabo los pasos necesarios para salvar la situación. Esta puede ir desde abortar el sistema previa alarma, hasta conmutar a una computadora de backup on-line, que tomará el control de la situación.

6.2. Alternativas de diseño para tiempo real

Las alternativas de diseño que afectan significativamente los atributos únicos de los SOTRs pueden clasificarse en seis áreas:

6.2.1. Modelo de memoria

Existen básicamente dos tipos fundamentales: un espacio de dirección global sin protección alguna entre tareas y espacios de dirección separados con alguna forma de protección entre tareas. Un modelo global es la única alternativa cuando la CPU no cuenta con los mecanismos de protección adecuados. Tales procesadores son simples y económicos. Pero el usar un modelo de memoria global con un procesador que provee instalaciones de protección es una incompatibilidad de software y de hardware.

Los entornos operativos multitarea generalmente se implementan usando un espacio de dirección global sin protección. Esto si bien es más rápido, ya que no debe salvarse demasiada información de contexto al conmutar de tarea, y más simple de programar ya que el compartir información no requiere precauciones especiales, tiene la gran desventaja que no ofrece protección entre tareas. La escritura o lectura errónea de una tarea no es atrapada por el hardware y puede no aparecer como un defecto a primera vista, hasta que son expuestos como un cambio en el entorno operativo. Adicionalmente el mismo kernel será vulnerable a las acciones de las tareas del usuario. En un entorno protegido, la tarea que escapa de sus límites asignados de memoria, será atrapada por medio de una excepción y será señalada y posiblemente exterminada sin afectar al sistema operativo o al resto del sistema. Por otra parte de esta manera es mucho más simple de depurar, ya que el SO dará cual es la causa de la muerte prematura de la tarea en cuestión.

QNX funciona en modo protegido a partir del i286 en adelante, al igual que UNIX u OS/2.

6.2.2. Modelo de tarea

Un diseño de tiempo real puede utilizar sólo una única división de ejecución, donde no existe el concepto de tareas corriendo concurrentemente. Tales sistemas se pueden clasificar en aquellos que permiten interrupciones y la ejecución de la RSI asincrónicamente de la ejecución de la tarea principal; y otro que no permite ningún tipo de interrupciones. Este último realiza todas las I/O sobre una base de polling más que en una manejada por interrupciones. La comunicación entre los diversos autómatas de estos sistemas puede implementarse a partir de funciones de colas o buffers.

Un sistema por polling de única tarea es altamente especializado y usualmente desarrollado para una operación repetitiva específica.

DOS es un SO monotarea que permite una ejecución RSI asincrónica. Las RSI asincrónicas pueden utilizarse para simular verdaderas multitareas en tales sistemas. Sin embargo esto trae aparejado el gran inconveniente que las interrupciones no pueden ejecutarse paso a paso. Salvo que se trabaje con algún depurador que use el i386 en modo protegido, como lo hace el *td386,* de Borland.

La mayoría de los SOTRs modernos proveen verdadera multirarea, si sólo porque las aplicaciones típicas con requerimientos de atributos de tiempo real involucran operaciones paralelas que se mapean más fácilmente sobre un diseño multitarea.

Modelo de binding un sistema con un número fijo de tareas construido inicialmente y un set específico de código para que ellas se ejecuten, se dice que provee un binding estático. Un sistema que permite que se cree y se cargue una nueva tarea con un nuevo código mientras el sistema está corriendo provee binding dinámico.

Sistemas estáticamente vinculados están íntimamente ligados con aplicaciones de tiempo real basadas en ROM, tales como un controlador de motor de automóvil.

El rtKERNEL, OS/2 QNX y UNIX proveen binding dinámico, por otra parte QNX 4.x es roomeable al igual que el rtKERNEL.

6.2.3. Modelo de reentrada

Un sistema operativo provee servicios a las tareas del usuario, algunos de los cuales son provistos por el kernel del sistema. La operación de la llamada del sistema cambia el estado del sistema global de modo usuario a modo kernel.

Un sistema que generalmente permite el intercambio entre tareas con origen en una tarea que se ejecuta en cualquier lugar en el modo kernel se llama *kernel generalmente reentrante.* A un sistema que sólo permite el intercambio de tareas con origen en una tarea que se ejecuta en momentos específicos en el modo kernel se lo denomina un *kernel con reentrada limitada*.

Un kernel generalmente reentrante todavía realiza operaciones no reentrantes que deben protegerse de la reentrada. Las técnicas que se emplean para lograr esto son: el intercambio de contexto se deshabilita temporalmente o bien se emplean semáforos para proteger la región crítica.

El control de la reentrada al kernel es el conductor primario de una métrica clave de performance del sistema tiempo real conocida como LIC (latencia del intercambio

de contexto). LIC es la mayor cantidad de tiempo durante el cual el sistema puede demorar la iniciación de una interrupción de contexto hacia una nueva tarea lista para ejecutar de prioridad más alta. Si ésta se encuentra en modo kernel en un kernel de reentrada limitada, entonces debe seguir ejecutando hasta que alcance un punto de desalojo. El camino más largo entre dos puntos de desalojo determina la LIC.

En un kernel generalmente reentrante, el camino de código más largo por el cual se deshabilita la interrupción de contexto determina la LIC. En el caso de emplearse semáforos, para proteger la región crítica, se puede permitir una interrupción de contexto, pero el nuevo proceso puede quedar bloqueado en la misma, lo cual crea una demora de la tarea de alta prioridad, disminuyendo la determinación global del sistema.

La implementación original de UNIX y sus sucesores usan una aproximación de punto de desalojo en la administración de reentrada kernel, y tienen un número limitado de puntos de desalojo.

6.2.4. Modelo de interrupción

Un kernel también podría ser reingresado debido a la concurrencia de una interrupción de I/O. Si esta es ejecutada por alguna tarea que requiera un interruptor de contexto para ejecutar entonces la reentrada debida a una interrupción no difiere de la reentrada del kernel por tareas. Si los interruptores de E/S son manejados por funciones especiales (drivers) entonces la reentrada debido a las interrupciones se convierte en un segundo tipo de interrupción a manejar por el kernel.

Todos los scaneos o manipulaciones de estructuras de información del kernel que causen problemas de acceso simultáneo están protegidos por la deshabilitación de hardware de reconocimiento de interrupciones de E/S durante el período de operación. El tiempo máximo en el cual se deshabilitan las interrupciones de E/S son enmascaradas es la LRI, la cual como la LIC, es una característica de performance clave de un SOTR.

6.2.5. Modularidad

Los SOTRs sirven a gran cantidad de aplicaciones y toman diversas formas para poder satisfacer las necesidades específicas de esas aplicaciones. En consecuencia, un SOTR debe tener un gran nivel de modularidad.

Por ejemplo una aplicación que necesite correr desde ROM, no necesita del sistemas de archivos, por lo cual debería poder quitarse de memoria el administrador de archivos, sin que afecte al sistema.

6.2.6. Control de reentrada con semáforos

En un sistema de tiempo real, una aproximación puede llevar a sutiles pero no menos significativos problemas de respuesta. Consideremos que un proceso A corriendo con prioridad 10, ha efectuado un wait sobre un semáforo a cero, cuando se produce un fin de E/S. El driver de disco despierta el proceso B que estaba en espera con una prioridad de 3 (aquí 3 tiene más prioridad que 10). El kernel switchea ahora a B, el cual debe acceder a la misma sección crítica que B, por lo tanto se bloqueará en el mismo semáforo. El sistema ahora señala ahora el citado semáforo despertando a A, que continua hasta salir de la región crítica, despertando en este momento a B, el cual pasará al estado de ready y luego al de ejecución.

En este escenario un proceso de más alta prioridad (B), es switcheado inmediatamente por el contexto. Sin embargo no está capacitado para obtener todos

los recursos que necesita, por lo cual se bloquea, hasta que un proceso de más baja prioridad (A), libera los recursos y permita que B continúe su ejecución.

Se debe tener en cuenta al momento de diseñar un SOTR la condición más sutil y peligrosa que pueda ocurrir en las siguientes circunstancias. Supóngase que luego que A comienza su ejecución, de nuevo B está bloqueado en el semáforo. Entonces ocurre otro fin de E/S despertando al proceso C con prioridad 7, dado que A es menos prioritario que C, se conmuta a C inmediatamente. El proceso B de prioridad 3, deberá quedando ahora en espera de que un proceso de más baja prioridad le devuelva la CPU. Este problema es llamado *inversión de prioridades*; su solución se conoce como herencia de prioridades. De esta forma el dueño de un recurso trabado hereda la prioridad de cualquiera que solicite ese recurso. En este escenario el proceso A hereda la prioridad de B, es decir 3, de esta forma C no puede desalojar a A, permitiendo que el mismo complete la región crítica, y B obtenga el recurso.

La solución por inversión de la prioridad presentada aquí es simple, pero la implementación puede llegar a ser bastante compleja cuando se consideran los requerimientos para la anidación de recursos trabados.

7. QNX, SISTEMA OPERATIVO ABIERTO, MULTITAREA, MULTIUSUARIO EN TIEMPO REAL

La presentación en sociedad de este sistema operativo fue en una muestra sobre computadoras personales auspiciada por IBM en Atlantic City, EE.UU., en 1982.

QUNIX, tal fue su nombre original, fue el primer sistema operativo multiusuario en tiempo real.

Ante un reclamo del departamento legal de AT&T por el uso de la palabra UNIX, pasó a llamarse QNX.

Los creadores de este S.O. fueron dos estudiantes canadienses, Dan Dodge y Gordon Bell. La primera microcomputadora en la que corrió se denominaba ICON, había sido creada por Dodge en 1973. Diez años más tarde, el Ministerio de Educación de Ontario, Canadá -en cooperación con Unisys- promovió un proyecto a desarrollar y mejorar la capacidad de trabajo en red de QNX, la cual hoy en día constituye una característica distintiva del mismo.

Este sistema operativo es altamente recomendable en aplicaciones donde se requiera controlar varios procesos en tiempo real, los cuales pueden ser mapeados en distintas tareas, o bien, mejor aún, se puede dedicar una máquina a cada proceso, con una o más máquinas supervisoras, conectadas todas en red.

Sea, por ejemplo, una fábrica de galletas. Los ingredientes que componen deben ser suministrados en proporciones determinadas, mezclados y horneados, proceso que es controlado por una PC, donde correrán las tareas que administran los ingredientes, la que controla el mezclado y otra para el horneado. Luego las galletas han de ser empaquetadas, proceso del cual se encargará otra PC, que controlará el grupo de máquinas empaquetadoras. Por último una tercera PC se encargará de monitorear las máquinas encargadas de pesar los paquetes, debiendo reingresar al proceso anterior aquellos que no pesen lo estipulado. Eventualmente pude haber una terminal común y corriente, la cual se podrá conectar a cualquiera de estas tres PC, por medio de una RS-422/485 o bien un modem, desde donde se podrá monitorear en forma totalmente transparente el proceso. Es decir que el operario, en forma remota, podrá utilizar la terminal como si estuviera operando el mismo teclado de algunas de aquellas máquinas (podrá inclusive tener acceso a las mismas pantallas de aquellas). Esto es posible gracias a un utilitario del QNX, llamado Ditto, el cual permite loguearse al sistema desde una puerta serie, en forma totalmente transparente, al sistema.

Para llevar esto a la práctica, se necesita en primer lugar una base de datos en tiempo real que pueda fijar (lock) registros para excluir la posibilidad de que distintas tareas accedan simultáneamente a la información (es decir que una lea mientras otra escribe, o bien más de una escriba). Esto también se aplica a los archivos. Es decir que se requiere que el S.O. brinde servicios de semáforos, mensajes, y colas. También se requiere desde cualquier máquina poder acceder en forma transparente a los recursos de las otras. Comunicarse y tener cierto control sobre las tareas que corren en otra máquina, es también imprescindible.

QNX, de Quantum Software Systems Ltd. ha sido pensado partiendo de estas premisas. Por ora parte, a parte a partir de la versión 4.0, QNX sigue el grupo de standares de IEEE POSIX, proveyendo todos los beneficios de un sistema UNIX. POSIX significa también portabilidad, dado que una plataforma UNIX convencional puede ser llevada fácilmente al entorno QNX.

Desde el punto de vista del entorno del usuario tenemos las características del conjunto de utilidades POSIX 1003.2, las cuales incluyen todas las utilidades standard de UNIX como *cat, cp, grep, lex y yacc.*

También están soportadas las primitivas de creación de procesos *fork(), exec() y wait().* La primera crea un nuevo proceso que es imagen del que llamó a la función. *fork()* devuelve el *id* del proceso hijo en caso de tratarse del padre, y cero en el caso del hijo. *wait() e*spera que muera el proceso hijo para seguir corriendo.

A partir de la versión 4 todos los servicios de entradas y salidas se basan en las primitivas *read()* y *write(),* lo cual brinda todas las operaciones que se esperaría de un entorno UNIX, como herencia y compartición de archivos.

Los módulos que constituyen QNX, son independientes de los restantes, es decir que pueden ser instalados, arrancados y detenidos en cualquier momento.

El sistema de archivos es *multi-threaded* y opera con prioridades *client-driven.* Lo cual significa, que los dispositivos tal como los floppies no degradan la velocidad de otras aplicaciones atadas a dispositivos más veloces. Así también se adoptaron prioridades *client-driven* en los servicios de entradas/salidas, es decir que si un task de baja prioridad, requiere los servicios de la tarea filesystem, esta no hereda su prioridad, sino que sigue trabajando con su propia prioridad, impidiendo que interfiera con las tareas de más alta prioridad.

Cabe destacar que este sistema operativo es apto para funcionar sobre la familia de microprocesadores Intel y por ahora solamente corre en PC's o máquinas compatibles.

Funciona en modo real en el 8086, y en modo protegido a partir del 286, explotando al máximo los mecanismos de protección del procesador (segmentación, niveles de privilegio, etc.).A partir del 386, usa el paginado aunque no se dispone de memoria virtual.

Para la operación en red, se requiere del hardware de Ethernet, Token Ring, Arcnet o bien alguna plaqueta de red, de la cual se disponga del driver correspondiente. Cada máquina de la red a partir de ahora pasará a denominarse *nodo.* Pueden instalarse una, o dos placas de distinto tipo de red independientes eléctricamente, en cada máquina, cada una se mapeará con una dirección lógica diferente.

7.1. Arquitectura

QNX, a diferencia del UNIX, se compone de un microkernel muy pequeño, el cual ocupa menos de 8kb de memoria, con lo cual es factible de ser alocado en la cache del 486, y de tareas específicas que se ocupan de los distintos servicios que provee el sistema operativo, denominadas *administradores del sistema*.

Básicamente, se pueden dar las siguientes características:

***** *modularidad*, cada función se maneja por un componente del sistema independiente.

***** *escalabilidad*, se puede elegir correr uno o varios módulos según los requerimientos. Cada administrador puede arrancarse o matarse en cualquier momento.

***** *distribución transparente,* no existen diferencias en correr una tarea en el nodo local, o bien en otra máquina de la red.

Estructuralmente hablando, el microkernel puede dividirse en cuatro partes:

*** IPC** maneja el pasaje de mensajes entre procesos, ya sea local o remotamente.
*** network interface** para transmitir los mensajes a través de toda la red, en forma automática.
*** hardware interrupt redirector** el cual atrapa todas las interrupciones y las redirige a sus respectivos handlers dentro de los procesos
*** realtime scheduler** el cual implementa tres algoritmos, siguiendo la especificación 1003.4 de POSIX.

Se dice que QNX es un sistema distribuido (y no en red), ya que el IPC se halla dentro de su estructura en el nivel más bajo, desde el punto de vista físico. Ver diagrama. Cada función en el tope de IPC,esta naturalmente pensada para un acceso transparente a la red. En consecuencia es muy simple lanzar procesos a través de la red, abrir archivos o dispositivos en forma transparente para la aplicación.

En cambio en un sistema de red, el IPC va debajo de la capa física de manejo de archivos, por lo cual se requiere mucho más trabajo por parte de la aplicación, lanzar un proceso remotamente o bien abrir un archivo.

Los algoritmos de scheduling son los siguientes:

**round robin:* cada proceso en la cola de listos para correr tiene un quantum de tiempo asignado (time slice), y se ejecuta durante ese quantum. Si al exterminar este, el proceso sigue corriendo, será quitado de contexto, y la cpu es cedida al próximo proceso de la cola de listos. Cabe destacar que ya que QNX provee un conjunto de 32 prioridades, dicho algoritmo comienza a buscar procesos listos a nivel 0, de no encontrar ninguno allí (por estar todos bloqueados), sigue por el nivel 1. El scheduler no baja hasta el próximo nivel de prioridad, hasta no terminar de ejecutar todos los procesos en la prioridad corriente y de desbloquearse alguno en una prioridad superior pasará a esta (es decir que el algoritmo es *preemptivo*: le cede el control a los procesos de mayor prioridad).

**FIFO* es similar al anterior, salvo que no hay quantums de tiempo, cada proceso en cada nivel de prioridad de corre hasta bloquearse. Esto se aplica en el caso de un sistema altamente cooperante. En este caso no se necesita el uso de semáforos para proteger las regiones críticas: uno sabe que el scheduler no me quitará la CPU, hasta bien yo lo crea conveniente.

**adaptive scheduling* es un algoritmo de tiempo compartido similar al algoritmo convencional de UNIX. Bajo este método, si un proceso utiliza un ancho de banda muy grande de la CPU sin bloquearse por cualquier razón, el sistema le irá retaceando prioridad, hasta llegar a una instancia donde el proceso queda fuera de competencia, a partir de allí se le comenzará a subir gradualmente, para darle una chance de operar en el procesador nuevamente. Cuando el proceso finalmente se bloquea haciendo una E/S, se le restaura la prioridad que tenía originalmente. Este es el protocolo por defecto.

Considere una aplicación como ser una hoja de cálculo , usualmente bloqueada esperando una tecla. Cuando usted presiona cualquier tecla, la aplicación repentinamente comienza a consumir tiempo del procesador. En este caso la prioridad

adaptiva comenzará a reducir su prioridad temporalmente, para no tomar ventajas de la respuesta interactiva del sistema.

Es importante recordar, que en cualquier sistema operativo de tiempo compartido (sea el UNIX, OS/2, etc.) un proceso (programa en ejecución) puede estar básicamente en tres estados: listo (ready), bloqueado o en ejecución (run). Existirá una cola para los procesos listos y otra para los bloqueados. Por otra parte existe una cola por semáforo, donde se almacenan los procesos bloqueados en el mismo. Puede haber tantos procesos en ejecución, como número de procesadores se dispongan (por lo general uno). Un proceso en ejecución sale de este estado bien por bloquearse en un semáforo a cero, en cuyo caso se pasará el mismo a la cola del semáforo involucrado y la lista de procesos bloqueados (cuando otro proceso señale aquel semáforo, se lo pasará a la cola de listos para correr). Es el scheduling quien decide que proceso debe tomar de la cola de listos, para ponerlo en estado de ejecución. También es el scheduling quien quita a un proceso del estado de ejecución cuando vence su time slice, para pasarlo a la cola de listos.

Por otra parte en todos los sistemas operativos multitarea, cada instancia de un programa, comparte la misma zona de código en memoria (segmento), pero tendrá distintos segmentos de memoria para cada conjunto de variables de aquellos.

Desde el punto de vista del hardware, se aprovechan los niveles de privilegios otorgados por el 286/386. El microkernel se encuentra en el nivel cero, mientras que en el uno están todos los administradores. El nivel dos se ha dejado vacante, y por último en el tres se encuentran las aplicaciones del usuario.

Existen varios administradores del sistema:

Task Administrator: Es responsable de crear y destruir tasks, y de alocar memoria para los mismos. Corre a prioridad uno, la más alta del sistema. Este es el único administrador del cual no se puede prescindir para correr QNX.

File System Administrator: este task es el responsable del sistema de archivos. Maneja todos los pedidos para abrir, leer, escribir y cerrar archivos (*open(), write(), read(), close()* y otras funciones). Corre a prioridad tres.

Device Administrator: Este task es responsable de manejar todos los dispositivos de caracteres del sistema. Esto incluye la terminal, la impresora, etc. Este maneja todos los pedidos para abrir, leer, escribir y cerrar archivos (*open(), write(), read(), close()* y otras funciones). Corre a prioridad dos.

Idle Administrator: Este task simplemente consume cualquier tiempo muerto del procesador. Como corre a la prioridad más baja permitida en el sistema (32), no afecta la performance.

Network Administrator: Es responsable de la comunicación sobre la red de área local (lan). Maneja todos los pedidos de datos que deben ser transmitidos por el sistema. Corre a prioridad tres. Si no se cuenta con la placa de lan, se puede prescindir de este task.

Estos son los cinco administradores básicos de QNX, existen otros con mayor grado de prescindibilidad, a saber:

Timer Administrator: Se ocupa de brindar un servicio de timers a las tareas, que así los requieran, por ejemplo para esperar un evento y ser despertadas en caso de no acontecer el primero. Las funciones con las que se acceden a estos, están definidas dentro del header "timers.h".

Spooldev Administrator: Esta tarea crea dispositivos virtuales los cuales retienen la información a enviar al dispositivo físico en disco. Esto permite que mucha gente comparta simultáneamente el mismo dispositivo físico. Cuando se cierra el dispositivo, el archivo es enviado al dispositivo real, en el orden de primero que llega, primero en ser servido. Un ejemplo de uso del spooler es el de impresora, ya que esta no puede ser compartida por varias tareas.

Queue Administrator: Este task soporta la implementación de colas, para intercambio de mensajes inter-tareas, cuyos servicios serán explicados más adelante.

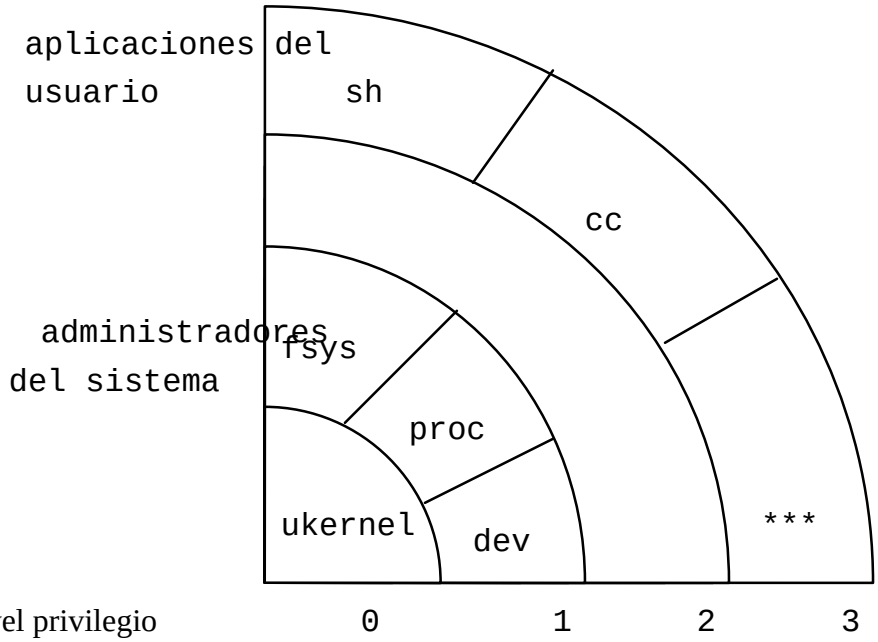
Locker Administrator: Esta tarea soporta el bloqueo (lock) de registros en sistemas de archivos a los cuales se accede concurrentemente. El bloqueo de registros (record locking) es compatible con el que se encuentra en UNIX System V.

Shared Library: Esta tarea brinda un servicio de bibliotecas dinámicas, es decir que provee funciones a las aplicaciones, que se linkean en tiempo de ejecución. En otras palabras, cuando una tarea requiere el uso de alguna de las funciones incluidas en este tipo de biblioteca, se comunica con este administrador (el cual previamente aguardaba bloqueado el ingreso de algún pedido), procesa el pedido, mientras la aplicación permanece bloqueada. Cuando la función de la biblioteca dinámica retorna, le responde a la aplicación despertándola. La ventaja de esta biblioteca respecto de las estáticas, es que permite que las funciones que se repiten en todas las tareas, ocupen un solo lugar en memoria, ahorrando una cantidad apreciable de espacio en aquella. Cada instancia de una función de esta biblioteca tendrá su propia zona de datos. Cabe señalar que inclusive las bibliotecas creadas por el usuario, pueden hacerse dinámicas.

Cuando una tarea requiere un servicio de alguno de estas tareas administradoras le envían un mensaje (*send()*). En este momento la tarea del usuario se bloquea, el administrador recibe el mensaje (*receive()*), lo procesa, y replica con cierto resultado (*reply()*). NUNCA un administrador envía un mensaje a las tareas del usuario, de lo contrario quedaría a merced de las mismas. La vida de un administrador del sistema puede ser sumarizada como sigue:

- Permanece bloqueado esperando un mensaje.
- Cuando un mensaje arriva, procesa el pedido y si el pedido se puede satisfacer replica indicando el resultado.
- Si el pedido no se puede satisfacer de inmediato, se recuerda y se bloquea esperando otro mensaje. Cuando se puede satisfacer el pedido inicial, a veces como consecuencia de la llegada de otro mensaje, se le replica a la tarea en ese momento.

Cuando se trabaja en red, los mensajes se pueden originar de otros nodos. El task destinatario no hace discriminación entre mensajes locales y remotos, dado que el IPC están entre mensajes locales y remotos, dado que el IPC está en la capa más baja del SOTR.



7.2. Comunicaciones inter-task.

QNX soporta tres tipos de comunicación inter-task.

7.2.1. Mensajes:

int send(int tid, void *send_msg, void *reply_msg, unsigned send_size, unsigned reply_size)

Función:

El mensaje apuntado por *send_msg* se envía al task identificado por *tid*. Los parámetros *send_size* y *reply_size*, puede ir de 0 a 65535 bytes. El task emisor se bloqueará hasta que el destinatario reciba y replique el mensaje. La réplica se coloca en el buffer *reply_msg. Su largo máximo es reply_size.

La función retorna normalmente el task identifier del task destinatario. Enviar un mensaje a un task que no existe o ya murió causa que el emisor no se bloquee y send() retorna cero. Un -1 implica que se produjo una excepción.

Esta no es una función estandar de ANSI C, y puede no ser portable en distintos SO.

Si el destinatario se encuentra receive-blocked y listo para recibir un mensaje, la transferencia de los datos en su dirección de datos, se realiza inmediatamente, y el receptor se desbloquea pasando al estado de listo para correr. De lo contrario el emisor se coloca en una cola (tal vez con otras tareas que también se hallan send-blocked) y la transferencia no se realiza hasta que el destinatario no haga un receive(), el cual satisfacerá al send().

El mensaje es copiado físicamente de la tarea origen a la tarea destino. Este copiado físico efectivamente "desconecta" a las tareas.

Notar que lo que viaja es el mensaje y NO un puntero al mismo. Si bien esto puede parecer ineficiente, tiene la ventaja que las tareas puedan ejecutarse en diferentes

procesadores o nodos de la red (no se puede compartir memoria, en nodos distintos). En aplicaciones especiales que impliquen mensajes muy largos y que no corran en red, el mensaje puede consistir de un puntero.

QNX no hace ninguna verificación de la estructura del mensaje transferido. Dicha estructura es acordada previamente por las tareas a comunicarse. Generalmente se emplea el primer byte de la estructura para discriminar el tipo de estructura a transferir.

Ejemplo:

```
#include <systids.h>

char       msg_buf[100];
unsigned   tid;        //a tid se le asignará el id del destino, luego veremos como

if ( tid != send(tid, msg_buf, reply_buf, sizeof(msg_buf), sizeof(reply_buf)))
   error("send failed.\n");
```

*int receive(int tid, void *msg, unsigned msg_size)*

Función:

El task que hace un *receive()* se bloquea, si es que su cola de mensajes se halla vacía, esperando un mensaje del task direccionado por tid, y no saldrá de este estado hasta que no reciba un mensaje de aquella tarea. Si tid es cero el primer task que le envíe un mensaje satisfacerá el receive(), momento en cual se desbloquea el receptor. Es decir que el primer *receive()* es específico y el segundo general.
La función retorna normalmente el task identifier del task emisor. Si se produce algún error devuelve cero. Un -1 implica que se produjo una excepción.

Esta no es una función standard de ANSI C, y puede no ser portable en distintos SO.

receive() se puede usar también para esperar en un port (luego veremos que es un port). En este caso tid debe ser cero. Luego veremos como se identifica a un port.

receive() cambia el estado del task emisor de send-blocked a reply-blocked.

Ejemplo:

```
#include <systids.h>

char   msg_buf[100];

tid=receive(0, msg_buf, sizeof(msg_buf)); //recibe un mensaje de cualquier tarea

//procesa mensaje
```

*int reply(int tid, void *msg, unsigned msg_size)*

Función:

El mensaje apuntado por msg se le envía como réplica al task especificado por tid, el cual hizo un send() y se encontraba esperando un reply(). El reply() debe efectuarse luego de un receive(). La transferencia de datos se realiza inmediatamente y el task que replica no se bloquea.

Replicar a un task que no existe o no espera un reply(), resulta en una operación nula.

Esta función retorna el valor de tid si tiene éxito, de lo contrario retorna cero.

Esta no es una función standard de ANSI C, y puede no ser portable en distintos SO.

Para obtener el tid de la tarea a la cual se le desea transmitir un mensaje se usa la función:

int name_locate(const char *name, unsigned node, unsigned size)

F*unción:*

Esta función retorna el id de la tarea registrada con el nombre apuntado por *name*, del nodo *node*, caso contrario retorna cero. El parámetro *size* es el tamaño máximo de los mensajes a intercambiar.

Por ejemplo para encontrar el *id* de la tarea registrada con el nombre "stm" en el nodo local, hacemos:

```
//tarea mac
int tid;
char   name[10]="stm";

if (!(tid=name_locate(name, 0, 10000)))
    printf("no encuentro a %s\n", name);
```

Para registrar el nombre de una tarea se emplea la función:

*int name_atach(const char *name, unsigned node)*

Donde *name* es un puntero al nombre simbólico que le vamos a dar al task, y *node* es el nodo donde deseamos registrarnos. Cabe destacar que **name* no puede tener más de ocho caracteres. Registrando el nombre de una tarea es posible que otra halle su tid para poder comunicarse, como en el ejemplo anterior.

name_atach() retorna el tid del propio task en caso de ejecutarse con éxito y cero sino.

Es decir que asociado a cada nombre hay un número de *tid* y un número de nodo. Cuando la asociación es local se maneja mediante una tabla en la misma máquina, de lo contrario, la tabla se encuentra en uno de los nodos de la red, mantenida por una denominada server.

Asociarse a un nombre es muy similar que asociarse a un port.

Un ejemplo de *name_atach()* sería:

```
//tarea "stm"

if (tid != name_atach("stm", 0))
    printf("stm: error al intentar registrarme\n");
```

Para resumir los dos casos posibles de comunicación por mensajes tenemos en primer caso:

```
TAREA A                    TAREA B

A sends to B                        .
   . A blocked                      .
   .                          .
   .                          .
   .                      B receives from A
   .                          .
   .                          . B processes the message
   .                          .
   .                          .
   .                      B replys to A
   . A continues
```

Es decir que A queda bloqueado hasta que B replica a A. En cambio B no se bloquea en ningún momento ya que cuando hace el *receive()* en su cola de mensajes ya se encuentra el mensaje proveniente de A.

El segundo caso se da cuando B hace un *reply()* antes de que A haga el *send().*

```
TAREA A                    TAREA B

   .                      B receives from A
   .                          . B blocked
   .                          .
   .                          .
   .                          .
A sends to B                        .
   .                          . B continues          . A blocked          .
        .                             . B processes the message
   .                          .
   .                          .
   .                      B replys to A
   . A continues
```

Vemos que B permanece bloqueado apenas efectúa el *receive().*

Es importante no olvidar que en caso de que A haga un *send()* a B, y B haga lo propio, A y B quedarán bloqueados en un deadlock. Cuando sea necesario que ambas

tareas hagan un send a la recíproca, debe adoptarse un protocolo muy rígido, de modo de no caer en un deadlock, o bien usar una tercera tarea, como intermediaria (agente).

Cuando es necesario comunicarse con una tarea de otro nodo, debe crearse previamente un circuito virtual, entre ambos con la siguiente primitiva:

int vc_create(unsigned node_id, int task_id, unsigned maximum_msg_size)

donde *task_id* es el task en el otro nodo *node_id*. Esto crea una entrada virtual en ambas máquinas que son imágenes de los task reales.

Retorna un identificador virtual de task (VID), o -1 en caso de producirse algún error. Este VID puede usarse por send(), receive(), reply(), etc. El circuito creado es bidireccional.

Existe también una función *int vc_release(int vtask_id)* que libera el circuito. De morir el task involucrado, libera todos los circuitos virtuales abiertos por éste.

7.2.2. Ports

Esta es la segunda forma de comunicación disponible en QNX. Con ellos es posible:

- implementar gestores de interrupción
- comunicación simple no bloqueante
- identificación de task no relacionados en el mismo nodo
- semáforos

Conociendo la identidad de un task da la posibilidad de comunicarse con el. Los *tid* son fijos y conocidos por todas las tareas. Cuando una tarea crea una nueva tarea, ambos, el padre y el hijo pueden obtener la identidad del otro , lo cual les da la posibilidad de comunicarse por mensajes. Pero cuando se desconoce la identidad de la tarea con la cual debemos comunicarnos, los mensajes no sirven.

Un port es un nombre globalmente conocido y una tarea puede "ligarse" a el de forma de comunicarse con las tareas desconocidas. Los ports se encuentran identificados numéricamente. Un task puede "ligarse", "desligarse", u obtener la identidad de una tarea ligada a un port. La siguiente tabla resume lo anterior. Por cada primitiva el port puede estar libre, o bien ya ocupado por otra tarea.

	el port está libre	el port está ocupado
atach()	retorna cero	retorna *tid* de la tarea ligada al port
detach()	retorna cero	retorna *tid* de la tarea ligada al port

Ligarse a un port libre, se obtiene el mismo, mientras que luego un *detach()* lo libera. Haciendo un *atach()* o un *detach()* de un port ocupado, puede conocerse el tid de la tarea dueña del port.

Estos mecanismos se aplican en una única máquina.

Mediante las primitivas *atach()* y *detach()* es posible ligarse a un port libre. Entiéndase por port, semáforo.

El finalidad más importante de los ports es permitir que el gestor de interrupciones se comunique con las tareas. El gestor de interrupciones señala un port al cual el task está "ligado". La señal despertará al task, lo cual permitirá que procese la interrupción.

Semáforos.

Las operaciones *atach()* y *detach* implementan un semáforo el cual puede ser usado para obtener acceso a un recurso. Por ejemplo el comando LIST del Shell de QNX, el cual envía un archivo a la impresora, no puede ser usado concurrentemente, por varios usuarios, o por un solo usuario desde más de una consola. Por lo cual LIST trata de ligarse al port 16, si lo consigue continua y envía el archivo a la impresora. De lo contrario la impresora ya está siendo usada por otra tarea. En ese caso LIST le envía un mensaje al LIST que se encuentra haciendo uso de la impresora. Como LIST nunca hace un receive, el segundo LIST se bloquea. Cuando el primero finaliza y muere, este se desengancha del port por lo cual todos los tasks *send-blocked* se desbloquean. El LIST que efectuó el *send()* tratará de "atarse" a ese port. El código dentro de LIST para hacer esto consiste de escasas líneas.

Los semáforos trabajan en una máquina individual.

Señales

Una tarea puede *señalar* a un port, resultando en un mensaje para el task atado al mismo. Esta forma de comunicación permite:

1. El task que hizo el *signal()* no se bloquea.
2. Los mensajes enviados al port no contienen datos.
3. El *signal()* puede realizarlo un interrupt handler (gestor o gestor de interrupción)
4. Los mensajes enviados por los ports se encolan afuera de la cola de los mensajes regulares, lo cual permiten ser recibidos primero.

El primer punto permite una comunicación sin bloqueos. El task asociado al port recibirá el mensaje cuando haga un *receive()* general (tid=0) o un *await()* de un port específico. El *tid* retornado por el *receive()* será el número de port, el cual como veremos es claramente distinguible de un *id* de tarea. Si un task efectúa múltiples *signal()* antes de que estos sean recibidos, los mismos serán recordados. Existe también un *signal* condicional (*csignal()*) el cual permite no señalar el port si allí ya hay una señal pendiente.

La identidad del port que envía la señal, es toda la información que el task asociado recibe. Esto es mucho más limitados que la primitiva de *send()* pero es adecuado para informar a la tarea que ha sucedido un evento. Por ejemplo si la tarea estaba esperando la interrupción de alguna placa de E/S, se asocia a un port preestablecido, el cual será señalado por el handler de la interrupción involucrada. No se necesita ninguna información extra.

El cuarto punto por último simplemente provee comunicación con una prioridad mayor que la correspondiente a los mensajes convencionales. Cuando los ports son usados en conjunción con *signal()* desde los handlers de interrupción, les da prioridad sobre pedidos de software de otras tareas.

7.2.3. Excepciones

Todas los tipos de comunicaciones mencionadas son sincrónicas con el receptor. El receptor explícitamente hace una operación de *receive()* y espera un mensaje. En cambio las excepciones son asincrónicas con el task receptor. Generalmente se generan por un evento anormal y pueden ocurrir en cualquier momento de la ejecución del task. El mejor ejemplo es la excepción generada por el teclado haciendo un BREAK. Esto tiene el efecto de setear una excepción en la tarea asociada al teclado. Si la tarea no está protegida contra la interrupción, la acción por defecto es matarla.

Las operaciones definidas por el sistema operativo son:

SIGABRT. Terminación anormal de programa, la que se genera por una llamada a *abort()*, entre otras cosas.

SIGALRM. La alarma del clock.

SIGALLOC. Error de alocación de memoria. Se genera por una llamada a *alloc()* que no puede satisfacerse.

SIGCOMM. Error de comunicaciones.

SIGDIVZ. División por cero.

SIGFPE. Excepción de punto flotante.

SIGHUP. Dejó de detectarse portadora en el modem o la línea de comunicaciones.

SIGILL. Instrucción ilegal.

SIGINT. Keyboard interrupt (break).

SIGPRIV. Violación de privilegio.

SIGQUIT. Termina el programa.

SIGSEGV. Violación de segmento de memoria.

SIGSLIB. Se pierde biblioteca compartida (dinámica).

SIGTERM. Terminación (kill).

SIGUSR1. Libre para ser definida por el usuario.

SIGUSR2. Libre para ser definida por el usuario.

No todas las excepciones pueden manejarse, SIGTERM y SIGQUIT, no pueden protegerse y siempre desembocan en la muerte de la tarea.

Más adelante veremos como una tarea puede atender las excepciones.

7.3. Estados de las tareas

En QNX una tarea se encuentra siempre en uno de los siguientes estados:

DEAD - La tarea no existe o esta muriendo.

READY - La tarea está lista para correr.

SEND_BLOCKED - La tarea hizo un *send()* que no fue recibido

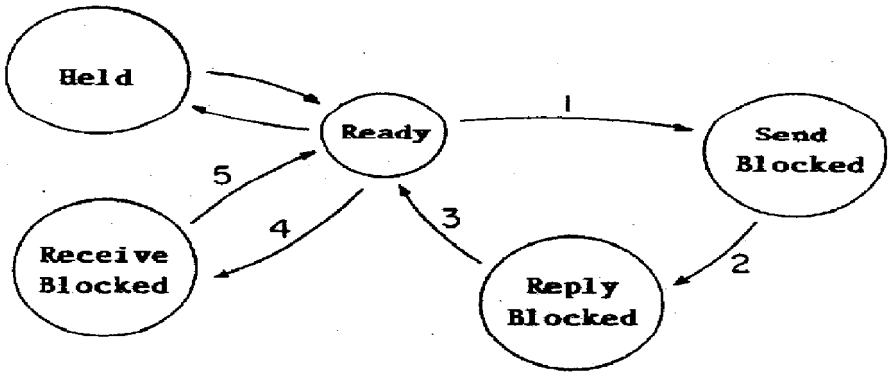
REPLY_BLOCKED - La tarea hizo un *send()* que fue recibido, pero no replicado.

HELD - La tarea está lista para correr, pero ha sido explícitamente demorada por otra.

NETWORK_BLOCKED - La tarea está bloqueada sobre un pedido a la red.

Los tres estados de bloqueo tienen que ver con las primitivas de comunicación. El estado de HELD indica que una tarea puede correr, pero esta siendo privada de ello por otra tarea que hizo un *hold* sobre la primera.

El task de mayor prioridad en estado de READY será el que se ejecute. De haber más de una tarea lista en la más alta prioridad, serán conmutadas (time sliced) cada *quantum* de tiempo.



1. La tarea envía un mensaje.
2. La tarea recibe un mensaje.
3. El destinatario contesta.
4. La tarea aguarda un mensaje
5. Mensaje recibido.

Task Ids

Un *tid* consiste de un word de 16 bits, donde el byte menos significativo es el número de task y los siete bits más altos es el número de encarnación(versión). Cada vez que se crea una tarea se le asigna el número de una tarea muerta, y el número de versión se incrementa por uno. Esto asegura que un número de task puede ser reusado 128 veces antes de reusar un dado tid. Esta es una política similar al usado por las empresas telefónicas al asignar números. Manteniendo el número viejo de abonado por un tiempo, es posible que alguien llame al viejo número y obtenga una respuesta. El bit más significativo indica si la tarea es local o remota (tarea virtual).

El número de tarea está en el rango de 1 a 254.

Los valores retornados por una llamada al sistema están resumidos en:

TASK NAME	SIGNIFICADO
0000	La tarea no existe
0100 a ff00	Nombre de port válido
xx01 a xxfe	Nombre de tarea válido
ffff	System call failed

Ejemplos:

0500 - port cinco
0508 - quinta encarnación de tarea local número 8.
8227 - 7ma encarnación de tarea remota número 28.

Jerarquía de Tareas

Las tareas corriendo en el sistema tienen una estructura jerárquica (árbol) muy parecida al de estructura de archivos. Cada tarea tiene un padre cero o más hijos y hermanos.

Ejemplo: cualquier comando ejecutado en el Shell (intérprete de comandos) es hijo de este.

La muerte de una tarea provoca la muerte de todos sus hijos, esto se hace para mantener la consistencia de jerarquías. Imagine remover un subdirectorio, sin borrar los archivos que están bajo él.

Creación de Tareas

Cuando una tarea crea una tarea hijo, existen tres opciones principales disponibles:

1. El padre queda bloqueado hasta la muerte de todos los hijos para seguir ejecutando.
2. El padre continua ejecutando concurrentemente con los hijos.
3, Pueden seguir ejecutándose ambos en paralelo, pero no se preserva la relación paternal (background)

El intérprete de comandos (Shell) por defecto elige la opción 1, al menos que explícitamente se le indique que corra el task en background (3). La opción 2 es poco común.

7.4. Manejo de excepciones

Función:

#include<signal.h>

void (*signal(int sig, void (*func)(int)))(int)

Esta función setea la respuesta a una señal especificada por *sig*. El primer argumento es uno del tipo signal, definido en *signal.h* (son las señales listadas en la primera parte) y el segundo es la función gestora de la excepción.

Cuando ocurre una excepción se ejecuta el gestor de interrupción. Se definen dos respuestas del sistema. El gestor por defecto termina el programa para la mayoría de las señales. Para algunas señales se escribe un mensaje a la pantalla antes de terminar. La otra respuesta del sistema es ignorar la señal. Si existe un gestor de señal del usuario, se ejecuta luego de ejecutar el gestor del sistema.

Una señal puede ocurrir en forma asincrónica (ej: ctrl-C) o bien cuando se da una situación particular durante la ejecución del programa (ej: división por cero) o por una

llamada a la función *raise()* (la cual produce una excepción del tipo especificado en el argumento de la misma).

Por ejemplo:

```
main()
{
signal( my_handler, SIGUSR1);
//fuerza interrupción por medio de timer
set_timer(TIMER_SET_EXCEPTION, RELATIVE, 2, SIGUSR1);
...
...
}

int my_handler()
{
//aquí manejo la excepción
signal(my_handler, SIGUSR1);
set_timer(TIMER_SET_EXCEPTION, RELATIVE, 2, SIGUSR1);
}
```

7.5. Uso del *fork()*

Ejemplo:

```
main()
{
if (fork()==0)
    {
    //soy el hijo
    exec("/my_user/my_program); //ejecuto mi programa
    }
else
    {
    //soy el padre, espero que muera mi hijo para seguir
    wait();
    }
}
```

7.6. Uso de los ports

Un port de QNX es un semáforo de software. Las tareas se refieren a un port por su número, el cual es único.

#include <portio.h>
unsigned attach_port(unsigned start, unsigned end)

Esta función tratará de asociarse a un port libre, comenzando desde *start* y siguiendo hasta *end.* Si *start* es cero, tratará de atarse al port 20. Si *end* es cero tratará de asociarse con el último port libre del sistema.

Si la función tiene éxito retorna el número de port, de lo contrario cero.
Ejemplo:

```
//para asociarse a cualquier port libre
if ((port=attach_port(0, 0)==0)
   printf("error al asociarme con un port libre\n");

//para asociarse a un port en un rango específico
if ((port=attach_port(20, 25)==0)
   printf("error al intentar asociarme con un port libre\n");
```

nota: esta primitiva es distinta de *atach()*, la cual trata de asociarse con un port de hardware, es decir de I/O.

#include <portio.h>
int await(int port)

Esta función se bloquea hasta que el port especificado por el argumento es señalado.

Cuando el port es señalado, la función retorna el número de port. Si el port no está asociado con un task, retorna cero.

Si el port ya ha sido señalado, *await* no se bloquea. Si ha sido señalado *n* veces previamente, *n* veces no se bloquea.

#include <portio.h>
int signal_port(int port)

Esta función señala el port especificado por el argumento, causando un mensaje de largo cero al task asociado al mismo, desbloqueándolo.

Esta función puede usarse dentro de un driver de interrupción para despertar a la tarea, que esperaba el evento de hardware.

Retorna el *tid* de la tarea asociada al port, si la hay, sino retorna cero.

#include <portio.h>
int detach(int port)

Esta función tratará de desasociarse del port indicado por el parámetro.

Retorna el *tid* de la tarea asociada al *port*, de no haber tarea asociada retorna cero.

7.7. Escribiendo un administrador de interrupciones

Procedimiento:

1. Convertir a la tarea en un administrador, lo cual se logra por medio de la función:

```
modify_flags(1, ADMIN);
```

2. Atarse a un port libre.

```
int port;
port=attach_port(0, 0);
```

3. Atar el handler de interrupciones al vector de interrupción:

```
unsigned        intnum;

set_int_vec(intnum, handler, 0);
```

4. Habilitar la interrupción:

```
unsigned    intnum;
set_int_mask(1, intnum);  //habilita la interrupción intnum
```

5. Esperar los mensajes y/o señales del handler de interrupción:

```
int port;

await(port);
```

ó

```
int tid;
char  buffer[BUFSIZE];

tid=receive(0, buffer, sizeof(buffer));

if (tid & 0xff)
   {
   //mensaje de una tarea
   ;
   }
else
   {
   //mensaje de un  port
   ;
   }
```

Si la tarea desea terminar, podría deshabilitar la interrupción, antes de salir:

```
unsigned    intnum;

set_int_mask(0, intnum);       //deshabilita la interrupción intnum
```

Un handler de interrupciones no es una tarea y por lo tanto no puede hacer cualquier pedido al sistema. Esto incluye las primitivas de mensajes, send(), receive(), etc. Esto también significa que el handler de interrupción no puede efectuar I/O. Tampoco podrá llamar a funciones de la biblioteca standard, ya que esta contiene llamada al S.O.. Las interrupciones deben estar deshabilitadas en todo momento del handler de interrupción. Se deben evitar las llamadas a la BIOS dado que estas habilitan las interrupciones. El handler de interrupciones debe ser la más corto y rápido posible, para minimizar el tiempo en que las interrupciones están deshabilitadas. Cuando el handler desea señalar una tarea para que corra, usa los métodos vistos anteriormente.

Ejemplo:

Este interrupt handler en particular, simplemente cuenta interrupciones y señala una tarea atada al port, cuando la cuenta alcanza 200. Primero, la función *main()*

```
#include<stdio.h>
#include<systids.h>
#include<qnx.h>
#include<portio.h>

void  far int_handler(void);

unsigned    port, count=0;

main()
{
int tid;
char  buffer[1];

if  ((port=atach_port(0,0))==0)
    {
    //error
    return(-1);
    }

set_int_vec(3, int_handler, 0);

//habilito int 3

set_int_mask(1, 3);
```

```
for(;;)
    {
    int i=0;

    await(port);
    fprintf(stdout, "Señal!!!!!!!\n");

    // sale luego de diez señales
    if (++i==10)
        break;
    }

set_int_mask(0, 3);       //disable interrupts
return 0;
}
```

Ahora el int handler, compilado en otro archivo por separado:

```
extern    int port, count;

void   int_handler()
{
if (++count==200)
    {
    signal_hw_port(port, 0);
    }
    //otherwise, just return normally
}
```

7.8. Manejo de Colas

Las funciones de colas en QNX se implementan a partir de las primitivas *send* y *receive*, a través de un administrador llamado *queue*, que fue mencionado en la parte I.

Los prototipos de las mismas se encuentran en *queue.h.* En todos los caso omitiré el *#include <queue.h>* que deberá ir a la cabeza de cada archivo que emplee las siguientes funciones:

*int queue_open(const char *queue_name, const char *open_mode, int window_size)*

Esta función abre una cola para leer o escribir mensajes. El parámetro *queue_name* es el nombre de la cola (de hasta 16 caracteres).

El parámetro *open_mode* se refiere el tipo de acceso a la cola: lectura (r), escritura (w), crear cola (c), etc.

El parámetro *window_size*, determina el número de mensajes que la cola aceptará antes de que la tarea que llamó a *queue_write()* se bloquee. Un valor de cero hace que cada escritor se bloquee, hasta que el mensaje sea leído mediante *queue_read()*.

Si tiene éxito esta función retorna un entero positivo pequeño que puede usarse como un identificador de cola (*queue_id)* para que las restantes tareas puedan referirse a ella.

*int queue_read(int queue_id, int wait_type, void *msg, int msg_size)*

Esta función lee un mensaje el cual fue puesto en la cola especificada con *queue_id,* y lo copia al buffer apuntado por **msg*. El último argumento es el tamaño máximo del paquete a leer, el cual no incluye el header de 8 bytes (el mensaje del usuario comienza en el byte 9).

Si existe más de un mensaje en la cola estos serán leídos en el orden en el que fueron colocados (FIFO).

El parámetro *wait_type* se usa para seleccionar uno de varios eventos a esperar. *queue_read,* normalmente se bloquea, si no hay mensaje esperando en la cola. El valor particular de cero, se puede usar para tratar de leer sin bloquearse. Otro evento es el timeout, es decir que luego de un tiempo determinado, de no arribar mensajes, se desbloquea.

Retorna el número de bytes leídos, o el status error en caso de que el valor sea menor que cero.

*int queue_write(int queue_id, void *msg, int *msg_size, int priority)*

Pone un mensaje en la cola determinada por el primer argumento. Esta función asume que el mensaje del usuario, apuntado por *msg, contiene un header de 8 bytes. El último argumento no tiene uso.

Retorna el número de bytes escritos, o bien un código de error menor que cero.

Adjunto los fuentes del administrador de colas para todo aquel que le interese.

7.9. Gestión de memoria compartida

Las funciones de *shared memory* se usan para crear zonas de datos compartidas, es decir variables comunes a todas las tareas. Estas funciones son muy similares a las encontradas en UNIX.

#include <memory.h>
*void far *memory_alloc(long size, char *name, unsigned flags)*

Esta función aloca un nuevo segmento de memoria del sistema de memoria y la hace disponible al programa . El tamaño del segmento debe ser menor que 64kbytes. El nombre *name* se puede usar para referirse al nuevo segmento. Otras tareas pueden referirse al mismo segmento usando este nombre. El parámetro *flags* especifica los derechos de accesos al segmento. *memory_alloc* retorna un puntero al nuevo segmento.

*memory_free(void far *pointer)*

Esta función libera un segmento de memoria. Si existe más de un link al segmento, el selector de este puntero será removido sin liberar la memoria.

*memory_link(char *name)*

Esta función crea un link a un segmento de memoria existente, cuyo nombre es *name*. Los derechos de accesos serán aquellos especificados por *memory_alloc()*. Retorna un puntero al segmento.

*void far *memory_pass(int src_tid, void far *pointer, int dst_tid, unsigned flags, unsigned delete)*

Esta función puede pasar un selector existente desde un task (*src_tid*) a otro (*dst_tid)* así este puede ser linkeado al segmento y un puntero que este puede usar. Los derechos de acceso del nuevo selector serán los del parámetro *flags.*

Un ejemplo típico de memoria compartida es el siguiente:

Consiste de tres programas *prog1*, *prog2*, y *prog3*. *prog1* carga *prog2*. *prog2* reserva cierta memoria, la inicializa a cierto valor, se la cede a *prog1* y muere. *prog1* luego carga a *prog3*, el cual obtiene la memoria de *prog1* y escribe el contenido en la pantalla. Para simplificar, no se hace manejo de errores.

```
//prog1

#include<stdio.h>

#include<process.h>

main()

{

createq(0, "prog2", NULL);       //createq(nro. nodo, path, parámetros)

createq(0, "prog3", NULL);

}
```

```
//prog2

#include<memory.h>

#include<magic.h>
```

```
#include<string.h>

#define  SIZE 1024L
main()
{
void  far*region;
char  buffer[80]="algunos datos interesantes aquí\n"
region=memory_alloc(SIZE, "share", FLAGS);     //pide la zona de memoria
memxcpy(region, buffer, strlen(buffer));       //copia los datos en ella
                        //se usa memxcpy() en lugar de strcpy(), pues se
    //trata de punteros far
memory_pass(My_tid, region, Dads_tid, FLAGS);    //se la pasa a prog1
}

//prog3
#include<memory.h>
#include<magic.h>
#include<string.h>
main()
{
void far *region, far *cp;
char far *cp;

region=memory_link("share"); //localiza la zona común por su nombre
cp=region;
while(*cp)
    putchar(*cp++)   //imprime el contenido de zona común
```

}

//Note que la única comunicación entre prog2 y prog3 es mediante el segmento común de memoria. //prog1 no hace nada excepto ejecutar a 2 y 3. En lugar de un array of char's se podría haber usado un //array de enteros o bien, distintos tipos de variables, las cuales se alojarían en el segmento común en un //orden preconvenido por todas las tareas, para luego poder ser referidas. Para obtener el tamaño que //ocupará cada variable se usará la macro sizeof(*variable*) .

8. CONFIABILIDAD DEL SOFTWARE

8.1. Introducción

Es difícil definir el término confiabilidad aplicado al software. En general, un programa de computación no se abre o cortocircuita, no presenta fallas catastróficas ni paramétricas. Es decir, lo aplicable a un componente no lo es a un programa.
Algunos dicen que si un programa presenta uno o más errores (bugs) es 0% confiable, y si presenta cero errores es confiable 100%.

Cuando se trata de un sistema, la confiabilidad del mismo debe ser considerada en el contexto de su utilización. Entonces un error de diseño o de construcción sólo afecta la confiabilidad en cuanto interactúa con ese error y provoca la falla. En programación alguien crea un error durante el diseño o codificación del mismo. Este error, si no es descubierto en la revisión del diseño o mediante herramientas especiales de programación, da origen a una o más fallas en el software.

El programador, algunas veces, detecta y corrige errores e inconscientemente introduce nuevos errores. Cuando el programa es ejecutado, los errores introducidos pueden quizás no resultar en errores de ejecución del programa. No así con otros datos de entrada, dando origen a pasos equívocos en el mismo. Un ejemplo clásico es la división cuando el divisor resulta el paso final de una serie de operaciones previas, haciendo casi imposible saber con anticipación el dominio del mismo (por ejemplo saber cuándo es cero). El programador comete un error si no verifica el valor del divisor, comparándolo con cero y proveyendo un paso alternativo.

Cuando los valores de las entradas (medio ambiente) interaccionan con el paso de programa erróneo, resulta un error en el programa. Sin embargo bien puede suceder que el conjunto de entradas (datos) del programa no haga nunca nulo el denominador de la fracción o cociente, en cuyo caso no surgirá error alguno en el programa. Es así que comienza a tener sentido hablar del MTBF (tiempo medio entre fallas) de un sistema debido a errores de programación. Nótese que se está hablando indistintamente de programación o software. Estos parámetros característicos de la confiabilidad en el software resultan de gran utilidad sobre todo en la planificación de recursos para el mantenimiento de programas de computación.

Si definimos la confiabilidad de un programa en términos de probabilidad, podemos decir que la confiabilidad de un programa es la probabilidad de que un programa se ejecute correctamente, de acuerdo a especificaciones dadas y para un determinado período (tiempo operativo).

Podemos enunciar otra definición si tenemos en cuenta la clasificación de errores de programación que enunciamos a continuación:

- Errores previos fijos: son aquellos que persisten luego que el programador ha trabajado en el programa corrigiendo (debugging) un error o cambiando un código.
- Errores generados: son aquellos que no existían, hasta que, como resultado de un cambio de código o corrección de un error, son creados.

Entonces, decimos que la confiabilidad de un programa es la probabilidad de que un dado programa opere por un determinado período sin errores de programación (bugs) en el sistema para el que fue diseñado y dentro de los límites especificados para el mismo.

Finalmente, si relacionamos la confiabilidad con las propiedades de un programa de computación obtenemos que la confiabilidad de un programa es la probabilidad de que no existan fallas de ejecución en una secuencia de n ejecuciones.

Un programa de computación especifica una función computable F sobre el conjunto E de entradas, definidas por ciertas expresiones:

E=[Ei] i=1,2,3,...,N

Siendo N el número de entradas posibles. Este número es finito para todos los programas que tengan un número finito de variables de entrada, cada una con un rango de variación finito. Cada Ei es un conjunto de valores de entrada suficiente para la ejecución del programa. La función F es una ley que asigna a cada Ei un valor F(Ei).

La ejecución del programa dado con entradas Ei computa el valor de la función F(Ei) como salida.

Los problemas respecto de la confiabilidad del programa aparecen cuando el valor de la función de salida F(Ei), para una ejecución, no coincide con el estimado $\hat{F}(Ei)$. Esta diferencia entre el valor de la función real y el estimado es lo que se denomina "falla en la ejecución" o "falla de ejecución" y ocurre cuando:

a) $F(Ei) <> \hat{F}(Ei)$
b) Final prematuro o truncamiento
c) Falla en el final

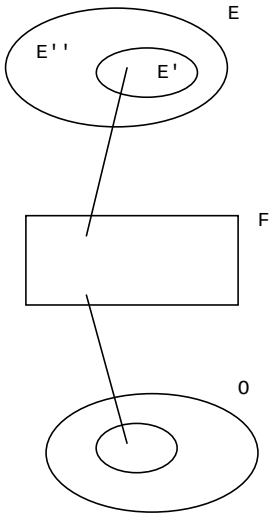
Un programa totalmente confiable es aquel para el que se cumple:

$F(Ei)=\hat{F}(Ei)$ para todo Ei de E.

Muchos programas producen salidas correctas para ejecuciones con algún conjunto de entradas Ei. Sin embargo presentan fallas de ejecución para otros conjuntos de entradas Ei. Haciendo una partición del conjunto E en dos, E' y E" tenemos que:

$F(Ei)=\hat{F}(Ei)$ para todo $Ei \in E'$

y existirá falla de ejecución para todo $Ei \in E"$.



Entonces la confiabilidad de un software será la probabilidad de que las entradas al programa, para una secuencia de n ejecuciones, pertenezcan todas al conjunto E'. Como esta probabilidad depende del medio ambiente en el cual trabaja el programa, este efecto se incluye en el modelo definiendo lo que se denomina "condición de contorno". Estas condiciones de contorno se ponen de manifiesto en forma de probabilidad p_i.

p_i será la probabilidad de encontrar a Ei como entrada al programa. Entonces la probabilidad de que en una sola ejecución se produzca la salida correcta será C(1):

$$C(1) = \sum_{i=1}^{N} p_i \, x_i$$ donde: $x_i=1$ si $F(Ei) = \hat{F}(Ei)$

$x_i=0$ si existe falla de ejecución

Si los sucesivos componentes de una secuencia de n entradas no son independientes, C(1) debe ser diferente para cada componente y C(n) debe ser calculado a partir de cada componente j de la secuencia como:

$$C(n) = C_1(1)\ C_2(1) \ldots C_j(1) \ldots C_n(1)$$

o bien:

$$C(n) = e^{\sum_{j=1}^{n} Ln\, C_j(1)}$$

Existe una clasificación general de sistemas que agrupa a los términos "reparación" y "disponibilidad" del software en dos grandes clases: los reparables y los no reparables. Así un sistema de instrumentación nuclear de un reactor de potencia se encuentra entre los primeros, en cambio un sistema de navegación inercial de un satélite entre los últimos. A los efectos de la programación también el software puede repararse o no, dependiendo casi exclusivamente de la accesibilidad al mismo.

Definimos la reparación de un programa como "la corrección (debugging) de la codificación de un programa o el reinicio del mismo para borrar un error ocurrido como resultado de un valor particular de los datos de entrada".

Si se trata de corrección de errores la "tasa de reparación" se mide en "correcciones por hora" para la corrección completa. El tiempo utilizado para la corrección completa incluye:

- Identificación del problema
- Diagnóstico del error
- Corrección del error
- Prueba de la corrección realizada
- Reinicio del programa

El tiempo de reparación es una variable aleatoria y los modelos más sencillos toman como MTTR (tiempo medio de reparación) la inversa de la tasa de reparación. Vemos que a medida que se corrigen errores en el programa la tasa de fallas baja.
Funcionalmente es lo mismo localizar y corregir errores que reiniciar un programa. Ambos llevan al mismo objetivo: colocar al software en el estado operativo. Conceptualmente no es lo mismo: cuando se trata de reiniciar un programa, la tasa de reparación es la inversa del tiempo medio de reinicio del software; si no se encuentran errores para corregir, la tasa de fallas permanece constante.

Cuando un programa tiene la capacidad de ser reparado, ya sea por accesibilidad o por estructura propia, entonces uno de los parámetros de importancia para medir la bondad o desempeño del mismo es la "disponibilidad". Se define disponibilidad como "la probabilidad de que un programa se ejecute correctamente, de acuerdo a especificaciones dadas y en un determinado instante de tiempo".

La diferencia fundamental entre la confiabilidad y la disponibilidad de un software es que la primera implica cero fallas en el intervalo (0,t), mientras que la segunda sólo indica que aquél se halla operativo en el instante t. Esto quiere decir que pudo haber fallado una vez en el intervalo (0,t) reparado y vuelto a operar; o haber fallado dos veces en el mismo intervalo de tiempo reparado y vuelto a operar en el mismo intervalo, o simplemente no haber fallado en el mencionado intervalo y así sucesivamente.

En general la disponibilidad resulta función del tiempo y es unitaria para t=0, decreciendo con el tiempo a algún valor constante después de varios ciclos de reparación. Este valor se denomina muchas veces "disponibilidad en estado estacionario"(steady-state availability). En etapas preliminares es posible emplear como elemento de decisión la disponibilidad en estado estacionario en su expresión más generalizada:

$$A_{ss} = \frac{MTTF}{MTTF + MTTR}$$

El MTTF es el tiempo medio en que el programa se encuentra operativo (tiempo medio hasta la falla) y el MTTR es el tiempo medio en que el programa se encuentra no operativo (tiempo medio de reparación). En función de las tasas de fallas λ y de reparación μ también puede ponerse:

$$A_{ss} = \frac{\mu}{\mu + \lambda}$$

8.2. Modelos de remoción de errores

En general los errores en un programa pueden ser clasificados en dos categorías: los previos fijos y los generados. Ejemplos de errores clásicos son:

- Una matriz sobreimpresa
- Errores en el bit bandera (flag bit)
- Errores en el bit de indexado o de desplazamiento (indexing - shifting)
- Errores de complementación aritmética
- Problemas de puntero (pointer)
- Errores de transferencia de control
- Problemas de direccionamiento indirecto
- Problemas en el reinicio del programa

Existen factores que contribuyen al aumento de errores como ser la escasa memoria disponible o la gran carga del sistema. En cambio la optimización de un programa disminuye el número de errores. En lenguajes de alto nivel se tienen menos errores por linea de programa que en lenguaje de máquina. Una de las hipótesis de trabajo en los modelos de remoción de errores es que el número de errores en programas similares es aproximadamente una constante.
Llamando:

E_T : Número total de errores
I_T : Número total de errores en lenguaje de máquina
$\frac{E_T}{I_T} =$ e : Densidad de errores

A los fines de comparar el número de errores para grandes y pequeños programas, el número de errores se normaliza dividiendo por I_T . Entonces es posible afirmar que:
"En programas similares la densidad de errores es aproximadamente constante".
Una guía razonable en cuanto a valores de densidad de errores es:

$$0{,}5\ 10^{-2} < \frac{E_T}{I_T} < 2\ 10^{-2}$$

En pequeños programas el 70% de los errores son encontrados en pruebas modulares (sobre determinados sectores del programa), el 25% en pruebas integrales (que incluyen varios sectores del programa) y el 5% restante luego de la corrida total del programa. Para sistemas operativos estos porcentajes varían sustancialmente: el 55% de los errores son hallados vía pruebas integrales y el 45% restante después de la corrida total del programa.
Si se llama:

$E_c(\tau)$: número de errores corregidos en el intervalo (o, τ)
$\gamma_c(\tau)$: tasa de corrección de errores
$\rho_c(\tau)$: densidad de corrección de errores
τ : tiempo de corrección de errores

Tenemos que:

$$\gamma_c(\tau) = \frac{\partial E_c(\tau)}{\partial \tau}$$

$$\rho_c(\tau) = \frac{\gamma_c(\tau)}{I_T} = \frac{1}{I_T}\frac{\partial E_c(\tau)}{\partial \tau}$$

Para el modelo de remoción exponencial utilizamos las siguientes hipótesis:

1. La tasa de detección de errores es proporcional al número de errores remanentes en el programa. Si $E_d(\tau)$ es el número de errores detectados en el intervalo (o, τ), y $E_r(\tau)$ el número de errores remanentes en el mismo intervalo, se tiene:

   $$\frac{\partial E_d(\tau)}{\partial \tau} = KE_r(\tau)$$

   Normalizando, dividiendo por I_T

   $$\frac{\partial e_d(\tau)}{\partial \tau} = Ke_r(\tau)$$

2. Todos los errores detectados son corregidos inmediatamente. Si $E_c(\tau)$ es el número de errores corregidos en el intervalo (o, τ), se tiene:

   $$E_d(\tau) = E_c(\tau)$$

   Normalizando nuevamente:

   $$e_d(\tau) = e_c(\tau)$$

3. El número de errores generados durante la corrección es despreciable:

   $$E_g(\tau) \cong 0$$

   o bien normalizando:

   $$e_g(\tau) \cong 0$$

   Siendo $E_g(\tau)$ el número de errores generados en el intervalo (o, τ)

Entonces la cantidad de errores remanentes $E_r(\tau)$ es la diferencia entre los totales E_T y los corregidos en ese intervalo (o, τ). Así:

$$E_r(\tau) = E_T - E_c(\tau)$$

O bien, normalizando:

$$e_r(\tau) = e_T - e_c(\tau)$$

Introduciendo las hipótesis enunciadas anteriormente tenemos que:

$$\frac{d\, e_c(\tau)}{d\, \tau} = K[\frac{E_T}{I_T} - e_c(\tau)]$$

Cuya solución general es:

$$e_c(\tau) = \frac{E_T}{I_T} + Ae^{-K\tau}$$

Con la condición inicial:

$$e_c(0) = 0$$

Se obtiene:

$$e_c(\tau) = \frac{E_T}{I_T}[1 - e^{-K\tau}]$$

Luego:

$$\frac{d\,e_c(\tau)}{d\,\tau} = \frac{d\,e_d(\tau)}{d\,\tau} = K(\frac{E_T}{I_T})e^{-K\tau}$$

Si M es el número de programadores trabajando en pruebas y corrección durante un mes, entonces será:

K=K'M

Luego:

$$\frac{d\,e_c(\tau)}{d\,\tau} = \frac{d\,e_d(\tau)}{d\,\tau} = \frac{K'\,ME_T}{I_T}e^{-K'M\tau}$$

8.3. Generación de errores durante una correción

El proceso de corrección de errores no es perfecto, debido a ello es que cierta cantidad de errores se introduce durante la implementación del mismo. El concepto es que si E_T es el número de errores con los que se comienza la depuración, después de suficiente tiempo de trabajo en corrección el número de errores removidos se aproxima a E_T.

Denominando:

$E_r(\tau_i)$: errores remanentes en el intervalo $(0, \tau_i)$

$E_r(\tau_{i-1})$: errores remanentes en el intervalo $(0, \tau_{i-1})$

$E_g(\tau_{i-1}; \tau_i)$: errores generados en el intervalo $(\tau_{i-1}; \tau_i)$

$E_c(\tau_{i-1}; \tau_i)$: errores corregidos en el intervalo $(\tau_{i-1}; \tau_i)$

Entonces es posible plantear la siguiente ecuación de balance de errores:

$$E_r(\tau_i) = E_r(\tau_{i-1}) + E_g(\tau_{i-1}; \tau_i) - E_c(\tau_{i-1}; \tau_i)$$

Llamando al intervalo $(\tau_{i-1}; \tau_i)=\Delta\tau$ se tiene:

$$\frac{E_r(\tau_i) - E_r(\tau_{i-1})}{\Delta\tau} = \frac{E_g(\tau_{i-1}; \tau_i)}{\Delta\tau} - \frac{E_c(\tau_{i-1}; \tau_i)}{\Delta\tau}$$

Pasando al límite cuando $\Delta\tau$ tiende a cero tenemos:

$$\frac{1}{I_T}\left[\frac{d\,E_r(\tau)}{d\,\tau}\right] = \frac{\gamma_g(\tau) - \gamma_c(\tau)}{I_T}$$

Escribiéndola de otra forma llegamos a que:

$$\frac{d\,e_r(\tau)}{d\,\tau} = \rho_g(\tau) - \rho_c(\tau)$$

donde:

$$\rho_g = \frac{\gamma_g}{I_T} = \frac{1}{I_T}\left[\frac{d\,E_g(\tau)}{d\,\tau}\right]$$

$$\rho_c = \frac{\gamma_c}{I_T} = \frac{1}{I_T}\left[\frac{d\,E_c(\tau)}{d\,\tau}\right]$$

Si $\rho_g(\tau) = 0$ no existe generación de errores.

Si en cambio resulta:

$$\rho_g(\tau) << \rho_c(\tau)$$

Entonces existe generación de errores a una tasa igual a la de remoción. Por otro lado si resulta:

$$\rho_g(\tau) >> \rho_c(\tau)$$

La tasa de generación de errores excede a la de remoción.

Evaluación del número inicial de errores de un programa

Tres son las hipótesis utilizadas en la evaluación del número inicial de errores en un programa:

- Las características de los errores permanecen invariables durante todo el proceso de depuración. En otras palabras "cuando se corrige un programa extenso los errores encontrados en las primeras semanas de trabajo son representativos del número total".
- En programas similares resultan correcciones independientes. "Cuando dos correctores trabajan en forma independiente uno del otro en un programa extenso, entonces la evolución de la depuración del programa es tal que la diferencia entre las dos versiones del mismo es despreciable".
- Los errores comunes son representativos. "Cuando dos correctores trabajan independientemente en un programa extenso, los errores comunes encontrados por ambos son representativos de la población total".

El modelo utilizado en esta evaluación se denomina modelo de semilla. El concepto es el siguiente: se quiere averiguar el número de bolillas blancas que existen en una urna (N_b). De acuerdo al modelo de semilla debe agregarse al contenido de la urna un número conocido de bolillas distintas de las de la urna, por ejemplo un número determinado de bolillas negras (N_n). Se mezcla su contenido y se extrae una muestra de n bolillas. Entre ellas habrá n_b blancas y n_n negras. Debe mantenerse la relación que sigue:

$$\frac{N_n}{N_n + N_b} = \frac{n_n}{n_n + n_b}$$

Si resulta:

$$\frac{N_b}{N_n} >> 1 \frac{n_b}{n_n} >> 1$$

Entonces:

$$N_b \cong \frac{N_n\, n_b}{n_n}$$

Llevado al campo de la programación será:

B_0: Número inicial de errores en el programa (igual a E_T)

B_1: Número de errores encontrados en el programa por el corrector número uno, hasta $\tau = \tau_1$.

B_2: Número de errores encontrados en el programa por el corrector número dos, hasta $\tau = \tau_1$.

b_c : Número de errores encontrados en el programa por el corrector número dos, independientemente del corrector número uno, hasta $\tau = \tau_1$.

Entonces:

$$\hat{B}_0 = B_1 \frac{B_2}{b_c}$$

Siendo $\hat{B}_0$ una estimación de E_T . La varianza del estimador será:

$$\text{Var } \hat{B}_0 = \frac{(B_1 B_2)^2}{b_c^3}$$

Todo esto es valido para:

$$b_c \approx 0$$

y

$$B_1, B_2 > 1$$

Si en un programa el primer corrector encuentra 55 errores y el segundo 25 errores y existen 10 errores comunes a ambos correctores, entonces:

$$B_1 = 55, \quad B_2 = 25, \quad b_c = 10$$

$$\hat{B}_0 = \frac{55*10}{10} = 137,5$$

$$\text{Var } \hat{B}_0 = \frac{(55*10)^2}{10^3}$$

8.4. Macromodelos

Todas las fallas en un software son debidas a errores residuales y en consecuencia la tasa de fallas será proporcional al número de errores remanentes en el programa.

Como todo error de software se debe a que el programa atraviesa una determinada porción que contiene errores remanentes, la probabilidad de que uno de estos errores sea encontrado en un intervalo Δt, después de t hores de operación correcta es z(t) Δt. Entonces esta probabilidad es proporcional al número de errores remanentes:

$$z(t)\Delta t = Ke_r(t)\Delta t$$

Luego: $$z(t) = Ke_r(t) = K[\frac{E_T}{I_T} - e_r(\tau)] \quad (*)$$

Llamando: $$\gamma = K[\frac{E_T}{I_T} - e_r(\tau)]$$

La probabilidad de operación satisfactoria, sin errores de software, será una función exponencial negativa del tiempo operativo:

$$C(t) = e^{-\gamma t}$$

En la siguiente figura se observa la variación de la confiabilidad C(t) con el tiempo de depuración. τ_0 es el tiempo que insumió la última corrección. El eje de abscisas corresponde al tiempo operativo normalizado. El tiempo medio hasta la falla MTTF será la inversa de la tasa de fallas:

$$\text{MTTF} = \frac{1}{\text{K}} e_r(\tau)$$

Si la tasa de remoción de errores es constante:

$$e_r(\tau) = \rho_0 \ \tau$$

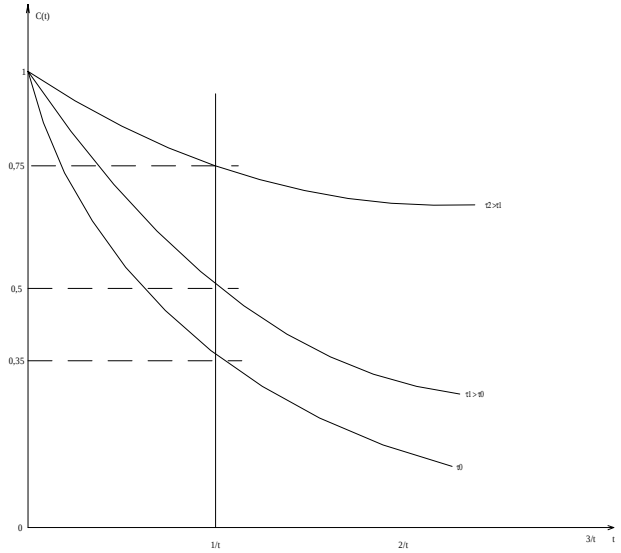
Entonces el MTTF queda:

$$\text{MTTF} = \frac{1}{\beta(1 - \alpha\tau)}$$

Con:

$$\beta = K\frac{E_T}{I_T}$$

y $$\alpha = \rho_0 \frac{E_T}{I_T}$$



Si E_T =1000 y el programador corrige 250 errores en el primer cuarto del período de depuración, la diferencia entre el MTTF resultante (programa con 750 errores residuales) es sólo el 33% más que en el comienzo de la corrección (pues el MTTF es inversamente proporcional al número de errores residuales). Cuando se llega a la última fase de la corrección con 900 errores corregidos y sólo 100 residuales, el MTTF aumentó 10 veces. La remoción de otros 50 errores aumentará el MTTF en un factor dos. El aumento del MTTF será en total de 20. Deducimos entonces que la corrección de errores en igual cantidad tiene menor efecto en el MTTF al comienzo del proceso de depuración que al fin del mismo.

Cuando: $\tau = \frac{1}{\alpha}$

Entonces el MTTF tenderá a infinito lo que es matemáticamente correcto pero no del todo satisfactorio desde el punto de vista técnico, pues se ha postulado que nunca se alcanzará el punto de cero errores.

Si se supone que $e_r(\tau)$ no puede bajar de un mínimo, tenemos que:

$$e_r(\tau) = e_r(\tau - \tau_i) \quad \text{para el punto } \tau_i$$

y el MTTF se fijará en:

$$\frac{1}{K} e_r(\tau)$$

Por ejemplo si se asume que sólo existirá 1 error remanente, la cota superior del MTTF será 1/K.

Existe otro macromodelo que propone una función de tasa de fallas de la siguiente forma:

$$z(t) = k[N - (i-1)]$$

donde:

k: constante de proporcionalidad

N: número total de errores presentes

i: número de errores encontrados después del tiempo de depuración

La conexión entre estos parámetros y los del modelo general (*) son:

$$k = \frac{K}{I_T}$$
$$N = E_r$$
$$I = e_r(\tau)\ I_T + 1$$

Si ahora hacemos intervenir el tiempo de ejecución y el de depuración del programa obtenemos que el modelo de confiabilidad queda:

$$C(\tau') = e^{\frac{-\tau'}{T}}$$

Donde T será el tiempo medio hasta la falla después del proceso de corrección:

$$T = T_0\ e^{\frac{\Theta\tau}{M_0 T_0}}$$

τ : tiempo de ejecución utilizado en la prueba del programa
τ' : tiempo de ejecución del programa después de corregido
Θ : razón entre el tiempo operativo y el de prueba
T_0 : tiempo medio hasta la falla al comienzo del test o prueba ($\tau = 0$)
M_0 : número de fallas que deben ocurrir para poner de manifiesto los errores ($M_0 = E_T$)

Estimación de parámetros de macromodelos
Los parámetros característicos de los macromodelos son en general dos, K y E_T . Estos pueden ser estimados utilizando tres técnicas diferentes:

- Estimación por momentos
- Estimación de maxima verosimilitud
- Estimación por cuadrados mínimos

Un software puede ser monitoreado en lo que respecta a su confiabilidad registrando los tiempos operativos y documentando cada falla en detalle. Si son *n* las corridas de un programa de las cuales falla *r*, las corridas exitosas serán (*n*-*r*).
Los tiempos de funcionamiento sin falla son:

$$T_1,\ T_2,\ \ldots,\ T_{n-r}$$

Los tiempos de funcionamiento hasta la falla serán:

$$t_1,\ t_2,\ \ldots,\ t_r$$

Las horas totales de funcionamiento exitoso serán:

$$H = \sum_{i=1}^{n-r} T_i + \sum_{i=1}^{r} t_i$$

La tasa de fallas resultará por definición:

$$\lambda = \frac{\gamma}{H}$$

y el MTTF su inversa:

$$MTTF = \frac{H}{\gamma}$$

Todo esto considerando tasa de fallas constante que es el caso a tener en cuenta. Para la estimación de parámetros se supone conocido I_T y $e_c(\tau)$ o en otras palabras la longitud del programa y la recolección cuidadosa de errores en el software.

- Estimación por momentos

Haciendo correr un test o prueba funcional después de dos diferentes tiempos de depurado τ_1 , τ_2 con $\tau_1 < \tau_2$ y $e_c(\tau_1) < e_c(\tau_2)$ se tiene:

$$\frac{H_1}{\gamma_1} = \frac{1}{\lambda_1} = \frac{1}{K[\frac{E_T}{I_T} - e_c(\tau_1)]}$$

$$\frac{H_2}{\gamma_2} = \frac{1}{\lambda_2} = \frac{1}{K[\frac{E_T}{I_T} - e_c(\tau_2)]}$$

De este sistema de ecuaciones pueden estimarse $\hat{E_T}$ y $\hat{K}$. Así:

$$\hat{E_T} = \frac{-[\frac{\lambda_2}{\lambda_1} E_c(\tau_1) - E_c(\tau_2)]}{\frac{\lambda_2}{\lambda_1} - 1}$$

$$\hat{K} = \frac{\lambda_1 I_T}{E_T - E_c(\tau_1)}$$

Si τ_1 y τ_2 son tales que no se corrigen errores, entonces:

$$E_c(\tau_1) = E_c(\tau_2)$$

y $$\lambda_1 = \lambda_2$$

Luego numerador y denominador se anulan y la estimación falla.

- Estimación de máxima verosimilitud

Es necesario formar la función de máxima verosimilitud como la probabilidad de obtener los resultados observados de tiempos de falla $t_1, t_2, ..., t_r$ y los tiempos sin falla $T_1, T_2, ..., T_{n-r}$.

Luego maximizar esta función y hallar de ella los mejores valores de los parámetros E_T y K que cumplen con ese requisito. Para el caso de m pruebas funcionales, los valores estimados de estos parámetros resultan:

$$\hat{E}_T = \frac{\sum_{j=1}^{m} \frac{\gamma_j}{\frac{E_T}{I_T} - e_c(\tau_j)}}{\sum_{j=1}^{m} H_j}$$

$$\hat{K} = \frac{\sum_{j=1}^{m} \gamma_j}{\sum_{j=1}^{m} [\frac{E_T}{I_T} - e_c(\tau_j)] H_j}$$

- Estimación por cuadrados mínimos

Es la estimación característica de una recta representada en este caso por al siguiente expresión:

$$\frac{E_c(\tau_i)}{I_T} = \frac{E_T}{I_T} - \frac{\lambda_i}{K}$$

Esta recta debe interpolarese utilizando el método de los cuadrados mínimos. Los resultados obtenidos con este método son bastante satisfactorios.

8.5. Micromodelos

En general todos los micromodelos tienen en cuenta la estructura interna del programa. Se trata de descomponer el software en estructuras básicas o bloques estructurales y analizar el pasaje del programa por esas estructuras básicas de acuerdo a los datos de entrada. El modelo de descomposición por pasos es el más conocido. En él se tienen en cuenta las siguientes hipótesis:

- El programa es estructurado; tiene una estructura natural
- Los distintos pasos del programa son estadísticamente independientes
- El modelo se desarrolla en base a la noción de frecuencia relativa
- Existe una secuencia de prueba que, o encubre un error (falla) o corre el programa completo sin cubrir un error (no falla)
- Definiciones:

  N : Número de pruebas

  i : Número de pasos del programa

  t_i : Tiempo empleado en correr el paso i

  q_i : Probabilidad de error en cada corrida del paso i

  f_i : Frecuencia con que es corrido el paso i

  n_f : Número total de fallas en N pruebas

  H : Tiempo total acumulado de prueba

El número esperado de errores en N pruebas será N q_j , si N es el número de pruebas y q_j la probabilidad de error en el paso j. Esto es válido al probar el paso j del programa. En general si las N pruebas están distribuidas por los distintos pasos del programa, N f_1 pruebas atravesarán el paso 1; N f_2 pruebas atravesarán el paso 2 y N f_i pruebas atravesarán el paso i. Entonces el número total de fallas en N pruebas será n_f :

$$n_f = \sum_{j=1}^{i} N f_j q_j$$

La probabilidad de falla en cualquier recorrida del programa será:

$$q_0 = \frac{n_f}{N} = \sum_{j=1}^{i} f_j q_j$$

Se puede evaluar la tasa de fallas del programa z_0 , calculando el número total de horas de prueba para corridas exitosas y no exitosas.
Si N f_i es el número total de pasajes por el paso i, de éstos, la fracción p_i será exitosa y acumulará n f_i p_i t_i horas de operación exitosa.
Si la distribución de tiempo hasta la falla para los N f_i q_i pasajes es uniforme, $t_i/2$ serán las horas en promedio acumuladas de no operatividad. Entonces el número total de horas de prueba acumuladas H será:

$$H = N f_1 p_1 t_1 + N f_2 p_2 t_2 + \ldots + N f_i p_i t_i + N f_1 p_1 \frac{t_1}{2} + N f_2 p_2 \frac{t_2}{2} + \ldots + N f_i p_i \frac{t_i}{2}$$

Operando queda:

$$H = N \sum_{j=1}^{i} f_j \, t_j (1 - \frac{q_j}{2})$$

Entonces la tasa de fallas z_0 será:

$$z_0 = \lim_{H \to \infty} (\frac{n_f}{H})$$

Es decir:

$$z_0 = \frac{\sum_{j=1}^{i} f_j \, q_j}{\sum_{j=1}^{i} f_j \, t_j (1 - \frac{q_j}{2})}$$

Midiendo f_j, q_j, y t_j para todos los i pasos del programa pueden evaluarse tanto q_0 como z_0.

Los valores de f_j y t_j dependerán en la mayoría de los casos de la estructura de la programación y de la complejidad de la estructura de control de todo el sistema donde se aplica el software. En particular el parámetro f_j podría ser evaluado colocando contadores en las distintas ramas del programa. Más difícil es la evaluación de q_j; durante el período de desarrollo puede estimarse en base a datos anteriores, históricos en la mayoría de los casos.

8.6. Modelos de disponibilidad

La técnica a utilizar para estudiar la disponibilidad del software es generalmente la de los modelos de Markov. Las hipótesis utilizadas en su proposición son:

- El número de palabras de código es grande. Del orden de 10^4 o más
- Para t=0 el programa contiene un número desconocido de errores n; siendo t el tiempo operativo del software
- A lo sumo un solo error es corregido en cualquier instante de tiempo
- Las transiciones entre un estado operativo y uno no operativo son independientes de los estados anteriores y sólo dependen del estado de donde se parte y del estado al que se llega.

La secuencia de estados operativos será:

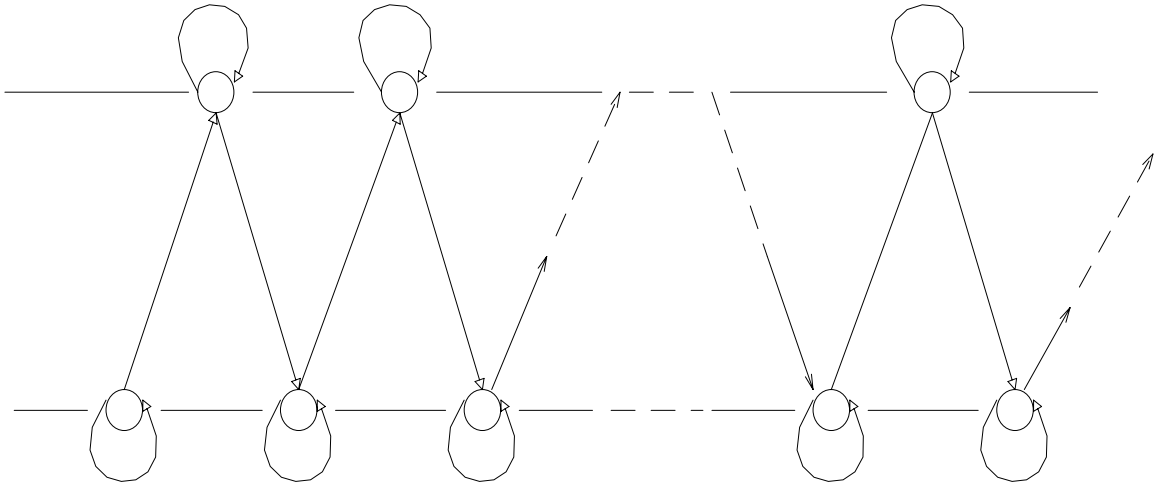
n, n-1, n-2, ..., n-k, n-k-1, ...

La secuencia de estados no operativos será:

m, m-1, m-2, ..., m-k, m-k+1, ...

El software se encontrará en el estado (n-k) si (k-1) errores han sido corregidos.

El software se encontrará en el estado (m-k) después de que k errores hayan sido detectados pero aún no corregidos. El esquema de transiciones de Markov puede observarse en la siguiente figura:



Este es un modelo de estados discretos y tiempo continuo. Asociado a las transiciones existen probabilidades de transición.

Así la probabilidad de transición del estado (n-k) al (m-k) será:

$\lambda_{n-k}\Delta t$ para k=0,1,2,...

En forma análoga la probabilidad de transición del estado (m-k) al estado (n-k-1) será:

$\mu_{m-k}\Delta t$ para k=0,1,2,...

Las probabilidades de transición $\lambda_j \Delta t$; $\mu_j \Delta t$ dependen en general del estado en que se encuentre el software.

λ_j representará la tasa de ocurrencia de errores en el programa y μ_j la tasa de corrección de errores del programa.

La disponibilidad del software será la probabilidad de que el mismo se encuentre operativo en t. Así:

$$A(t) = P_n(t) + P_{n-1}(t) + P_{n-2}(t) + \ldots$$

Es decir:

$$A(t) = \sum_{k=0}^{\infty} P_{n-k}(t)$$

Donde $P_{n-k}(t)$ será la probabilidad de estar en el estado (n-k) en t.

Dos son los modelos que pueden ser derivados a partir de la dependencia de λ_j y μ_j respecto del número de errores corregidos k y del tiempo (datos históricos de errores).

En el primer caso se tendrá:

Modelo I
$$\lambda = \lambda_{n-k} = \lambda(k)$$
$$\mu = \mu_{m-k} = \mu(k)$$

Y en el segundo:

Modelo II
$$\lambda = \lambda(t)$$
$$\mu = \mu(t)$$

Para el modelo I el sistema de ecuaciones de Markov presenta el siguiente aspecto:

$$\dot{P}_n(t) = -\lambda P_n(t)$$
$$\dot{P}_m(t) + \mu P_m(t) = \lambda P_n(t)$$
$$\dot{P}_{n-1}(t) + \lambda P_{n-1}(t) = \mu P_m(t)$$
$$\dot{P}_{m-1}(t) + \mu P_{m-1}(t) = \lambda P_{n-1}(t)$$

Donde el punto sobre las probabilidades representa la derivada primera de ellas respecto del tiempo. Este sistema de ecuaciones diferenciales puede resolverse vía transformada de Laplace en forma más o menos compleja. Los métodos de resolución por análisis numérico resultan mucho más eficientes y permiten su tratamiento computacional.

Para el modelo II el sistema de ecuaciones de Markov es:

$$\dot{P}_n(t) = -\lambda(t)\, P_n(t)$$
$$\dot{P}_{n-k}(t) + \lambda(t)\, P_{n-k}(t) = \mu(t)\, P_{m-k+1}(t)$$
$$\dot{P}_{m-k}(t) + \mu(t)\, P_{m-k}(t) = \lambda(t)\, P_{n-k}(t)$$

Su resolución vía método de Euler o Runga-Kutta es más o menos sencilla. Las condiciones iniciales para ambos modelos son:

$P_n(0) = 1$

$P_{n-k}(0) = 0$ k=1, 2, ...

$P_{m-k}(0) = 0$ k=0, 1, ...

9. FIABILIDAD DE LOS SISTEMAS OPERATIVOS

Una de las características deseables en un sistema operativo es la fiabilidad: es evidente que un sistema que no sea fiable es de poca utilidad. En este apartado vamos a estudiar con más detalle que se entiende por fiabilidad y vamos a discutir procedimientos mediante los cuales pueda incrementarse la fiabilidad de un sistema operativo. Veremos que la fiabilidad no constituye un detalle a añadir sino que es algo a tener en cuenta desde las primeras etapas de un sistema operativo.

9.1. Objetivos y terminología

Es evidente, que a partir de lo visto anteriormente, un sistema operativo es un complejo fragmento de software que debe llevar a cabo una gran variedad de funciones. Es importante que estas funciones se lleven a cabo en forma satisfactoria, siendo uno de los principales objetivos del diseñador de sistemas, velar porque así sea. La tarea de este diseñador se hace tanto más difícil si se considera que el producto acabado no trabajará en un entorno perfecto: es muy probable que se vea sometido a una gran variedad de circunstancias que puedan afectar en forma adversa a su funcionamiento. Tales circunstancias incluyen el mal funcionamiento del hardware de la computadora, errores por parte de los operadores así como instrucciones ilegales suministradas por el usuario. Un sistema operativo debería seguir prestando sus servicios -posiblemente degradados- incluso frente a este tipo de circunstancias. Estas observaciones nos conducen a definir la fiabilidad de un sistema operativo como el grado en que éste cumple sus especificaciones por lo que respecta al servicio de sus usuarios incluso en condiciones inesperadas de funcionamiento o bien en condiciones hostiles. Esta definición pone de relieve que el concepto de fiabilidad es algo relativo y no algo absoluto: ningún sistema operativo puede ser totalmente fiable ya que, tomando un ejemplo extremo, sería difícil seguir suministrando servicio de archivos si falla el disco de almacenamiento masivo (disco magnético u óptico). Un sistema operativo altamente fiable seguirá sus especificaciones incluso en el caso de un notable mal funcionamiento del hardware así como en el caso de un error grave del usuario. Un sistema menos fiable es posible que deje de seguir sus especificaciones tan pronto como se le suministre una instrucción sin sentido.

El nivel de fiabilidad que debería alcanzar un sistema operativo depende, evidentemente, de la responsabilidad que sus usuarios depositen en él. Una gran responsabilidad, tal como la asociada a un sistema que controle una cápsula espacial o bien el tráfico aéreo, requiere una gran fiabilidad. Como que una fiabilidad elevada aumenta considerablemente los costos, el diseñador del sistema debería intentar conseguir un nivel de fiabilidad que esté en relación con la confianza depositada por los usuarios (es importante destacar, sin embargo, que esta confianza puede a su vez venir influida por la fiabilidad del sistema).

No debe confundirse la noción de fiabilidad con la de corrección: ambos conceptos están relacionados pero son distintos. Un sistema operativo es correcto si al ser ejecutado en un determinado entorno exhibe un cierto comportamiento deseado. La corrección es, desde luego, un atributo deseable en un sistema operativo aunque constituye una condición insuficiente para la fiabilidad. Ello es así debido a que la demostración de la corrección, sea a través de comprobaciones, sea mediante una

prueba del tipo formal, está basada en unas suposiciones acerca del entorno que no son generalmente válidas. Estas suposiciones se refieren habitualmente a la naturaleza de las entradas y al correcto funcionamiento del hardware y puede que estén muy poco justificadas en la práctica. En otras palabras: el entorno en el que opera un sistema raras veces se corresponde con el que se supuso en la demostración de su corrección.

La corrección no sólo no constituye una condición suficiente para la fiabilidad sino que sorprendentemente, tampoco es una condición necesaria. Distintas partes de un sistema operativo pueden ser incorrectas en el sentido de que los algoritmos que gobiernen determinadas tareas pueden que no produzcan los resultados deseados. pero el sistema puede seguir funcionando en forma fiable. Como ejemplo consideremos un sistema de archivos en el que la rutina *close* omita en según que circunstancias almacenar la longitud de un archivo en la correspondiente entrada de directorio. Mientras el archivo termine con una marca EOF que pueda reconocerse las funciones de I/O seguirán pudiendo leer con éxito el archivo, funcionando correctamente, funcionando correctamente el sistema a pesar del incorrecto funcionamiento de uno de sus componentes. Este ejemplo muestra el empleo de información redundante para detectar y corregir errores, técnica a la que volveremos posteriormente.

El lector no debería concluir de lo anterior que la corrección no es importante. No constituye una condición ni necesaria ni suficiente de la fiabilidad pero, evidentemente, ayuda a conseguirla. Un sistema operativo correcto es mucho más probable que sea fiable que uno que no lo sea. Es, desde luego, poco recomendable que un diseñador dependa de los mecanismos de fiabilidad para anular los efectos de algunas deficiencias conocidas. Tampoco justifica la fiabilidad la falta de esfuerzo hacia la corrección. El enfoque adecuado consiste en intentar elaborar un sistema que sea correcto de acuerdo con unos determinados supuestos acerca del entorno y, al mismo tiempo, diseñar mecanismos de fiabilidad para superar con éxito aquellas ocasiones en los que estos supuestos no sean válidos. Estudiaremos en posteriores apartados estos enfoques complementarios.

Llegados a este punto vamos a introducir tres términos más. Definiremos como *error* a una desviación del sistema con respecto a su comportamiento específico. Así pues, un error constituye un evento. Son ejemplos de errores la asignación de un recurso no compatible a dos procesos a la vez, o bien el borrado de un elemento de un directorio asociado a un archivo que se sigue utilizando. Un error puede estar ocasionado por un mal funcionamiento del hardware, por una intervención incorrecta de un usuario o un operador o bien por una deficiencia en alguno de los programas del sistema. En todos estos casos se emplea el término *fallo* para hacer referencia a la causa de un error. Cuando ocurre un error es probable que algunos elementos de información del sistema se vean afectados. Vamos a referirnos a este hecho como al *daño* ocasionado por el error. Este daño puede, evidentemente, ocasionar fallos que conduzcan a más errores. Veremos que una de las formas de conseguir una alta fiabilidad consiste en acotar el daño que pueda ocasionar un error limitando, en consecuencia, la propagación de errores a lo largo del sistema.

A partir de todas estas consideraciones podemos ver que los esfuerzos para producir un sistema operativo altamente fiable deberán centrarse en los siguientes aspectos:

a) evitar fallos: eliminar los fallos en las etapas de diseño e implementación, o sea, producir un sistema correcto.

b) detectar errores: detectar los errores tan pronto como se pueda con el fin de reducir todo lo posible el daño ocasionado.
c) tratar los fallos: identificar y eliminar cualquier fallo que produzca un error.
d) recuperación de los errores: análisis y reparación del daño resultante de un error.

En los apartados siguientes vamos a analizar cada uno de estos temas punto por punto.

9.2. Evitación de fallos

Tal como hemos visto, los fallos en un sistema operativo pueden provenir del *hardware*, del operador, de la incompetencia o ignorancia del usuario o bien de errores de *software*. Vamos a ver a continuación formas de evitar cada uno de estos tipos de fallos.

9.2.1. Fallos procedentes del operador o del usuario

Vamos a decir muy poco acerca de los fallos generados por operadores o usuarios: sólo que no pueden eliminarse. Todo lo que puede hacerse es reducirlos en lo que a número se refiere a través de unos programas adecuados de educación e instrucción de los usuarios.

9.2.2. Fallos de hardware

La forma más evidente de reducir la incidencia de los fallos de *hardware* consiste en utilizar los componentes más fiables que pueden obtenerse dentro de las limitaciones impuestas por el costo total del equipo. Los diseñadores de *hardware* emplean diversos métodos para mejorar la fiabilidad: desde el más elemental -emplear componentes fiables a nivel individual- a las técnicas más complejas que incorporan algún tipo de forma de detección y recuperación de errores dentro de cada subsistema.

La detección de errores está basada habitualmente en el almacenamiento o bien en la transmisión de información redundante, tal como bits de paridad o *checksums*. La recuperación se intenta a menudo repitiendo la operación que dio lugar al error. Constituyen ejemplos de aplicación de técnicas de este tipo los transportes de cintas y de disco magnéticos en cuyo diseño se incorpora la posibilidad de repetir las distintas operaciones que llevan a cabo un cierto número de veces si ocurre un error de paridad o de *checksum.* El volver a intentar estas operaciones evidentemente es útil sólo en el caso de errores transitorios(esto es, fallos debidos a condiciones temporales tales como interferencias, partículas de polvo, etc.) ya que los fallos de tipo permanente seguirán dando lugar a errores.

Otra técnica que se emplea a menudo para la transmisión de datos es la utilización de códigos detectores y correctores de errores en los que la información redundante que se transmite con los datos constituye un medio de recuperación de algunos errores de transmisión. Otra técnica empleada es la de la *mayoría*, en la que se disponen de varios (normalmente tres) componentes idénticos a los que se aplica la misma entrada y cuyas salidas se comparan. Si difieren se toma como valor correcto el de la mayoría, tomándose nota del componente minoritario como sospechoso de mal funcionamiento. Un componente que aparezca repetidas veces como sospechoso, se

supone estropeado y puede ser sustituido. Este método de la mayoría constituye una técnica cara ya que se basa en la redundancia de componentes, utilizándose normalmente sólo cuando es esencial una gran fiabilidad.

La fiabilidad de todas estas técnicas es la de enmascarar los fallos de hardware mediante el software que se ejecuta en él. Así pues, puede que haya fallos en el hardware, pero el software anulará o al menos disminuye los efectos de los mismos. Veremos que puede extenderse esta noción de enmascaramiento al mismo sistema operativo, de forma que los errores que se produzcan a un determinado nivel de sistema, se vean ocultos desde los niveles superiores.

9.2.3. Fallos del software.

Hay varios enfoques complementarios para reducir la incidencia de los fallos de *software*. Vamos a discutir a sólo uno de ellos.

El primero consiste en adoptar técnicas de gestión, metodologías de programación y herramientas de software que constituyan una ayuda positiva hacia la consecución de productos sin fallos. A mediados de los sesenta creían algunos grupos que cuanto mayor fuese el producto de software a producir, mayor número de programadores habría que asignársele. La experiencia de IBM con el ejército de programadores que intervino en el OS/360 destruyó ese mito: la opinión actual es que la utilización indiscriminada de fuerza de trabajo es posible que cree muchos más problemas de los que resuelva. La agrupación de varios programadores en pequeños equipos, conocidos popularmente como equipos del jefe de programación, se considera ahora una forma mucho más adecuada de organizar la producción de software. Cada equipo es responsable de un módulo de software que tiene perfectamente definida sus comunicaciones externas y especificaciones. Dentro del equipo cada miembro tiene su función bien definida tal como la de codificación, documentación o bien enlace con otros equipos.

La forma en la que los programadores escriban sus programas puede ejercer una profunda influencia en la calidad del producto final en términos de coherencia, comprensibilidad y ausencia de errores. Durante los últimos años se han propuesto diversas metodologías de programación siendo la más conocida la programación estructurada y más recientemente la orientada a objetos.

9.3. Detección de errores

La clave para la detección de errores es la redundancia o , sea, el empleo de información superflua que pueda utilizarse para comprobar la validez de la información principal contenida en el sistema. El término redundancia refleja el hecho de que la información empleada para llevar a cabo esta comprobación es redundante por lo que se refiere a los algoritmos principales del sistema. Ya hemos visto en el último apartado como podía emplearse la información redundante, tal como bits de paridad, checksums y crc, para detectar errores en el ámbito del hardware. La codificación constituye un útil método de detección de errores, pudiéndose utilizar en algunos casos para la recuperación. Tal como mencionábamos antes, los errores detectados por el hardware pueden ser enmascarados por el propio hardware, o bien pueden ser comunicados al sistema operativo por medio de señales de error al gestor

de interrupciones de primer nivel. Son ejemplos de este último tipo los rebasamientos aritméticos así como las violaciones de memoria y de protección. Las acciones que puede llevar a cabo el sistema operativo en estos casos van a comentarse más adelante.

El propio sistema operativo también puede llevar a cabo una detección de errores. Una comprobación habitual consiste en que los procesos verifiquen la consistencia de las estructuras que utilicen. Un ejemplo ello puede ser de la lista doblemente enlazada de la cola de procesos, que el scheduler puede recorrer en ambos sentidos. Otro ejemplo puede ser la comprobación de las modificaciones de una tabla manteniendo en la misma un checksum de todos sus elementos.

Una generalización de esta forma de comprobación consiste en asociar a cada principal acción de un proceso una prueba de aceptación que puede utilizarse como criterio de decisión acerca de si esta acción ha sido llevada a cabo correctamente o no. La prueba de aceptación es una expresión lógica que es evaluada (como parte de la ejecución de un proceso) después de finalizar la acción. Si el resultado es un cero lógico es señal de que se ha producido un error. Esta prueba de aceptación será todo lo rigurosa que haga falta. En general será formulada para comprobar la no existencia de aquellos errores que el diseñador considere más probable. Como ejemplo, consideramos la acción de un scheduler que vaya modificando de forma periódica las prioridades de las tareas y reordene de acuerdo con ello la lista de procesos. La prueba de aceptación para esta operación podría ser simplemente el comprobar que los descriptores de procesos de la lista reordenada estén realmente por orden de prioridades. Una prueba de aceptación más rigurosa sería exigiría que el número de descriptores de la lista fuera el mismo que antes, previniendo así que se perdiese o duplicase alguno de estos descriptores. Existe evidentemente, un compromiso entre el rigor de una prueba de aceptación y el overhead que ella representa. El diseñador debe ponderar los costos tanto como se refiere al equipo, como a la degradación de la eficiencia, frente a los beneficios que se obtengan de ellos.Es frecuente, desafortunadamente, que no puedan medirse con precisión ni los costos ni los beneficios.

Hasta ahora hemos considerado sólo los componentes que puedan darse en un único módulo de software o de hardware. Los errores que se produzcan en las interfaces entre componentes son más difíciles de detectar, aunque pueda conseguirse algo comprobando la verosimilitud de cualquier información que pueda transmitirse por ellas. Así por ejemplo pueden comprobarse algunos parámetros de funciones para verificar si están en rango adecuado, siendo posible también verificar el que los mensaje transferidos entre procesos sigan algún protocolo en concreto.

Vamos a concluir este apartado destacando que la pronta detección de un error es la mejor forma de limitar el daño que este pueda causar así como acotar la cantidad de proceso desperdiciado por su culpa. La capacidad de un sistema operativo para reaccionar pronto frente a errores después que hayan ocurrido puede mejorarse notablemente mediante la utilización de unos adecuados mecanismos de protección implementados en hardware.

9.4. Tratamiento de los fallos

El tratamiento de un fallo requiere en primer lugar su detección y en segundo lugar su reparación. Cualquier programador de sistemas confirmará que la detección de un error y la identificación del fallo correspondiente no son la misma cosa. Un error puede tener varias causas posibles localizadas sea en el hardware, sea en el software, no siendo evidente ninguna de ellas. Una de las principales ayudas en la identificación de un fallo consiste en la detección de los errores antes de que quede enmascarada su causa por daños o errores posteriores. De ahí que la temprana detección de los errores sea de capital importancia, tal como se indicaba anteriormente.

En algunos casos puede que sea posible adoptar la actitud fácil de ignorar los fallos, aunque ello represente una serie de supuestos acerca de la frecuencia errores así como de la amplitud del daño que puedan ocasionar. Así, por ejemplo, puede que se considere que no vale la pena intentar identificar un fallo transitorio de hardware hasta que la frecuencia de los errores que lleve asociados no supere un determinado umbral (Avizienis, 1977). Normalmente, sin embargo, es importante localizar un fallo y tratarlo convenientemente. La búsqueda de uno de estos fallos vendrá dirigida generalmente por el conocimiento del investigador acerca de la estructura del sistema. Si este conocimiento es incompleto o si el fallo ha afectado a dicha estructura, la tarea del investigador será difícil.

Una forma de ayudar a este investigador consiste en que el sistema produzca una traza (o registro) de su actividad reciente. Los eventos a registrar pueden ser activaciones de procesos, entradas de rutinas, transferencias de I/O, etc. Desafortunadamente, la cantidad y el detalle de esta información deben ser bastante grandes si se quiere que esta traza tenga algún valor práctico, pudiendo ser muy elevado el trabajo que represente llevar a cabo dicho registro. Algo puede ganarse haciendo que esta traza sea opcional de forma que sólo haga falta activarla cuando existan razones para pensar que el sistema está funcionando mal. Es conveniente también que la traza pueda ser selectiva de modo que sólo deba registrar la actividad de las partes del sistema que se consideren sospechosas. Ello, evidentemente, representa un juicio subjetivo acerca de cuáles son las partes a considerar sospechosas, juicio él mismo sujeto a error.

Una vez localizado un fallo su tratamiento representa algún tipo de reparación. En el caso de los fallos de hardware esta reparación representa normalmente la sustitución del componente estropeado. Ello puede llevarse a cabo a mano o bien de forma automática, dependiendo de la habilidad del hardware para localizar sus propios fallos y aislar el componente defectuoso. La sustitución manual puede representar el tener fuera de servicio a todo el sistema durante un cierto período de tiempo, o puede llevarse a cabo preferiblemente en paralelo con el servicio (posiblemente degradado). La reparación de un transporte de disco, por ejemplo no debería interrumpir el servicio normal de haber disponibles otros transportes. Sin embargo, la sustitución de una placa de circuito impreso del procesador central de un sistema con una sola CPV, es muy posible que represente el tener que parar todo el sistema y volver a relanzarlo.

Los fallos en los componentes de software provienen de deficiencias en el diseño y en la implementación, o bien de daños ocasionados por errores anteriores (a diferencia del hardware, el software no presenta fallos debido al envejecimiento). La corrección de un fallo de diseño o de implementación representa la sustitución de un cierto número de líneas de programa (es de esperar pocas) mientras que un programa que haya resultado dañado por una u otra razón, puede sustituirse mediante una copia de seguridad guardada en algún otro sitio. La reparación de datos dañados constituye un tema que dejamos para el próximo apartado que versará sobre la recuperación de

errores en general. Sería deseable que de la misma forma que ocurre con el hardware, se pudiesen sustituir los diferentes componentes de software sin tener que dejar fuera de servicio todo el sistema. Esto es algo a tener en cuenta en el diseño de este sistema.

9.5. Recuperación de fallos

La recuperación de un error implica una apreciación del daño ocasionado seguido de un intento de repararlo. La apreciación puede estar basada completamente en un concepto a priori por parte del investigador, o bien puede involucrar al propio sistema a través de la ejecución de una serie de comprobaciones para determinar el daño causado. En cualquier caso esta apreciación (al igual que la identificación del fallo) será guiada por supuestas relaciones causa efecto definidas por la estructura del sistema. Un error al actualizar un directorio de archivos puede esperarse que afecte por ejemplo, al sistema de archivos, pero no a la estructura de procesos, o bien a los descriptores de periféricos. Existe evidentemente, el peligro de que la estructura del sistema y en consecuencia cualquier relación supuesta, se vea asimismo dañada, aunque la probabilidad de que ello ocurra se ve notablemente reducida si se usan los mecanismos de hardware de protección adecuados.

La técnica habitual para reparar el daño consiste en devolver los procesos afectados al estado en el que se encontraban antes de que se produjese el error. Este enfoque se basa en la existencia de *puntos de recuperación* en los cuales se registra información suficiente acerca del estado del proceso como para permitir su reestablecimiento si fuera necesario. Esta información comprende al menos el entorno volátil y una copia del descriptor de proceso. Puede incluir también una copia del contenido del espacio de memoria utilizado por la tarea en cuestión. El intervalo entre puntos de recuperación determina la cantidad de proceso que es de esperar que se pierda al ocurrir un error. La pérdida de información después de un error puede reducirse mediante técnicas de *audit-trailk*, por las cuales todas las modificaciones al estado de un proceso son registradas a medida que van ocurriendo. La recuperación del error consiste entonces en volver al último punto de recuperación y realizar los cambios de estado indicados por este registro. La copia incremental puede asimilarse a un mecanismo particular de *audi-trail.*

9.6. Tratamiento a varios niveles de los errores

En los últimos apartados hemos descrito diversas técnicas de gestión de errores. En este apartado vamos a ver cómo pueden incorporarse estas técnicas a la estructura por niveles de nuestro sistema operativo.

El objetivo principal es el de enmascarar en cada nivel de sistema operativo los errores que ocurran en los niveles inferiores. Así pues, cada nivel del sistema debe ser en todo lo que pueda responsable de su propia recuperación de errores de forma que aparezca a los niveles superiores como libre de fallos. En el caso en los que este enmascaramiento no sea posible, los errores de un nivel inferior que ocurran al llevar a cabo alguna función para un nivel superior deberían ser comunicados a este último en forma ordenada (por ejemplo, a través de un retorno de subrutina asociado a un error específico). El nivel superior puede que sea capaz de llevar algún tipo de recuperación enmascarando de esta forma el error a los niveles por encima de él. Los errores que no puedan enmascararse en ninguno de los niveles deben comunicarse

eventualmente al usuario o operador. En el nivel inferior del sistema, los errores detectados por el hardware de la CPU pero que no puedan ser enmascarados son comunicados al núcleo por medio de señales de error dirigidas al gestor (handler) de interrupciones de primer nivel.

Como ejemplo de lo que queremos dar a entender, consideremos un proceso de usuario que quiera abrir un archivo. El proceso invocará al sistema de archivos a través de una llamada al servicio *open* la cual a su vez invocará al sistema de I/O para leer los bloques correspondientes de disco. La transferencia de I/O solicitada será eventualmente iniciada por el gestor de periférico de disco. En el caso de un error de paridad o de CRC, puede obtenerse un enmascaramiento a un primer nivel a través del propio hardware del controlador de disco que estará probablemente diseñado con el fin de detectar tal error y volver a leer el bloque de disco correspondiente. Si en varios intentos no se corrige la situación, el error será comunicado al gestor de periférico a través de un registro de estado del periférico. El gestor puede decidir enmascarar el error reiniciando toda la operación, aunque si ello no tiene éxito deberá comunicar el error a la función *open()*. En el sistema de archivos la recuperación es posible si existe en algún lugar otra copia del archivo de seguridad. Si ello también fallase, el error no podrá enmascararse por más tiempo y deberá comunicarse al usuario.

Algunos errores, desde luego, no deben enmascararse ya que son indicativos de un fallo grave en un programa del usuario. Estos no son errores del sistema sino del usuario, con lo que la misión del sistema operativo se reduce a comunicárselos. Un ejemplo de ello constituye el rebasamiento aritmético detectado por el procesador y comunicado al núcleo del sistema del gestor de interrupciones de primer nivel. La función de interrupción del núcleo puede identificar finalmente al responsable como el proceso con un adecuado mensaje de error. Otros errores que no deben ser enmascarados son las violaciones de memoria y de protección que deriven de la ejecución de programas del usuario.

9.7. Conclusiones

Hemos descrito en este capítulo técnicas para mejorar la fiabilidad de un sistema operativo. A medida que las computadoras se introducen en más y más áreas de la actividad humana y la responsabilidad que recae sobre ellas aumenta, la fiabilidad se vuelve de capital importancia. Las técnicas que hemos descrito, aunque están sujetas aún a considerables mejoras, constituyen una base para alcanzar la fiabilidad necesaria.

10. DESCRIPCIÓN DE ARQUITECTURA INTERNA DEL RTKERNEL

10.1. Estructuras

10.1.1. Event Control Block (OS_ECB)

Esta estructura contiene la información sobre los eventos, estos pueden ser semáforos, mailbox o colas.

os_event_cnt: es un contador usado cuando el evento es un semáforo. Cuando el mismo es menor que cero indica el número de tareas bloqueadas en el mismo.

**os_event_ptr*: puntero a un mensaje en el caso de mailbox o a una estructura de colas (OS_Q) en el caso de tratarse de una cola. Cuando el OS_ECB se encuentra en la lista de os_ecb_free_list, este campo se emplea para apuntar al próximo (OS_ECB).

**os_tcb_blk_task*: puntero al primer task bloqueado en este evento, este primer proceso es el de mayor prioridad en la lista de bloqueados en este evento. El siguiente task esta apuntado por el puntero *next_dly* de OS_TCB (ver más adelante). Todos los tasks se ordenan en prioridad decreciente. En el caso de misma prioridad se ordenaran en forma de fifo (first input first output), es decir que al producirse el evento se despertará al task que primero ingresó en la cola de bloqueados.

10.1.2. Task Control Block (OS_TCB)

Esta estructura contiene la información concerniente sobre los tasks existentes.

**stack_ptr:* puntero al tope del stack de la tarea

**allocmem_ptr*: puntero a la memoria alocada para el stack, tope máximo del stack.

task_stat: estado del task. Puede ser:

OS_TASK_RUN: signfica que el task está corriendo.
OS_TASK_RDY: el task está listo para correr
OS_TASK_SEM: el task está bloqueado en un semáforo
OS_TASK_MBOX: el task está bloqueado en un mailbox
OS_TASK_Q: el task está bloqueado en una cola
OS_TASK_DLY: el task está demorado
OS_TASK_NULL: el task no existe

prio: prioridad del task, al momento va de 0 a OS_LOW_PRIO, siendo 0 la más alta y OS_LO_PRIO la más baja.

task_id: identificador de task, es un número que identifica univocamente al task y va de 0 a OS_MAX_TASKS.

dly_time: en el caso de estar demorado, indica el número de ticks remanente del sistema, durante el cual seguirá demorado. En el caso de estar bloqueado en un evento indica el timeout en ticks, que permanecerá en ese evento. Si este valor es cero significa que no existe timeout.

**pecb*: puntero a OS_ECB, en el caso de que el task esté bloqueado en un evento.

timeout_flag: este flag indica si el task salió del evento por timeout en caso de encontrarse en algún valor distinto de cero.

**next_free*: este puntero se emplea cuando el OS_TCB se encuentra en la lista de os_tcb_free_list y apunta al próximo OS_TCB libre.

**next_rdy*: este puntero se emplea para formar la lista circular de tareas listas (ready tasks), con el mismo se enlaza a la próxima tarea lista a ejecutarse, la tarea asociada al TCB apuntado por este puntero será la próxima tarea a jecutarse.

**prev_rdy*: cumple la función de apuntar a la tarea anterior ejecutalble o lista. Se lo utiliza para que la lista de tareas listas sea doblemente enlazada y de esta forma facilitar la inserción o borrado de tareas de esta lista.

**next_blk*: apunta a la próxima tarea bloqueada en el caso de que el task seencuentre bloqueado en un evento, que puede ser un semáforo, una cola o un buzón.

**next_dly*: apunta a la próxima tarea demorada, en este caso el task se encuentra en la lista de tasks demorados (*os_tcb_dlylist*) y no es pasible de ser ejecutado.

Queue Control Block (OS_QCB)

Contiene la información sobre cada cola creada por el usuario. Sus campos son los siguientes:

**os_qcb_ptr:* puntero a próximo QCB libre en la lista de QCB libres (os_qcb_free_list).

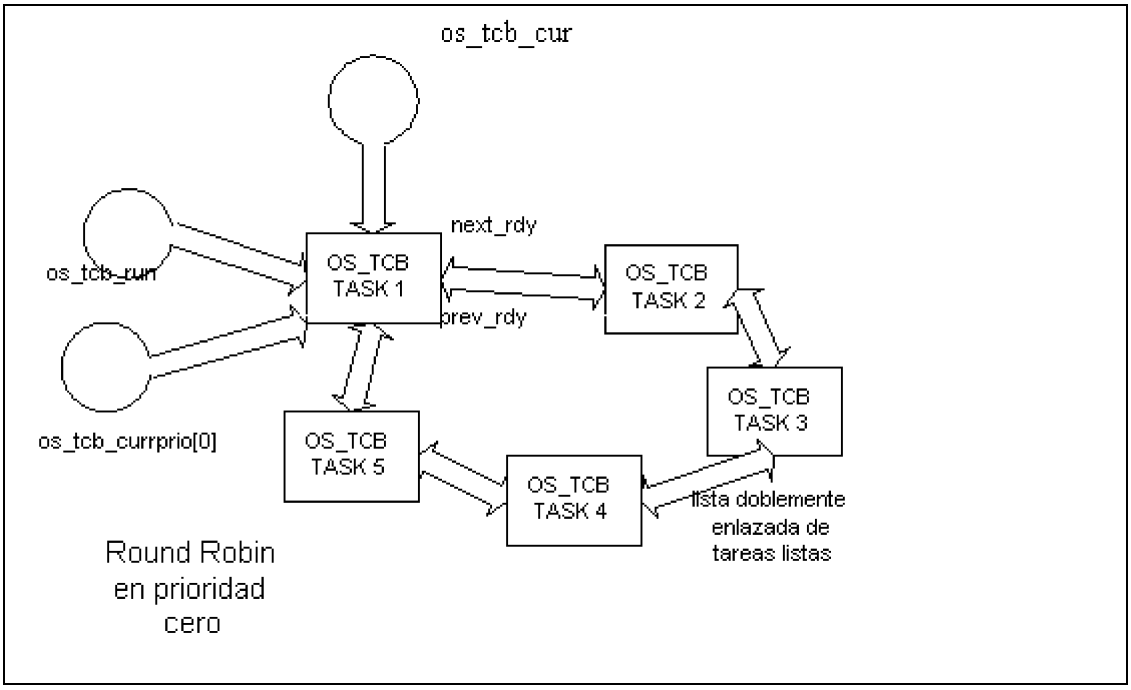
**qstart*: puntero al buffer de memoria que aloja a la cola, tomado por un far alloc().

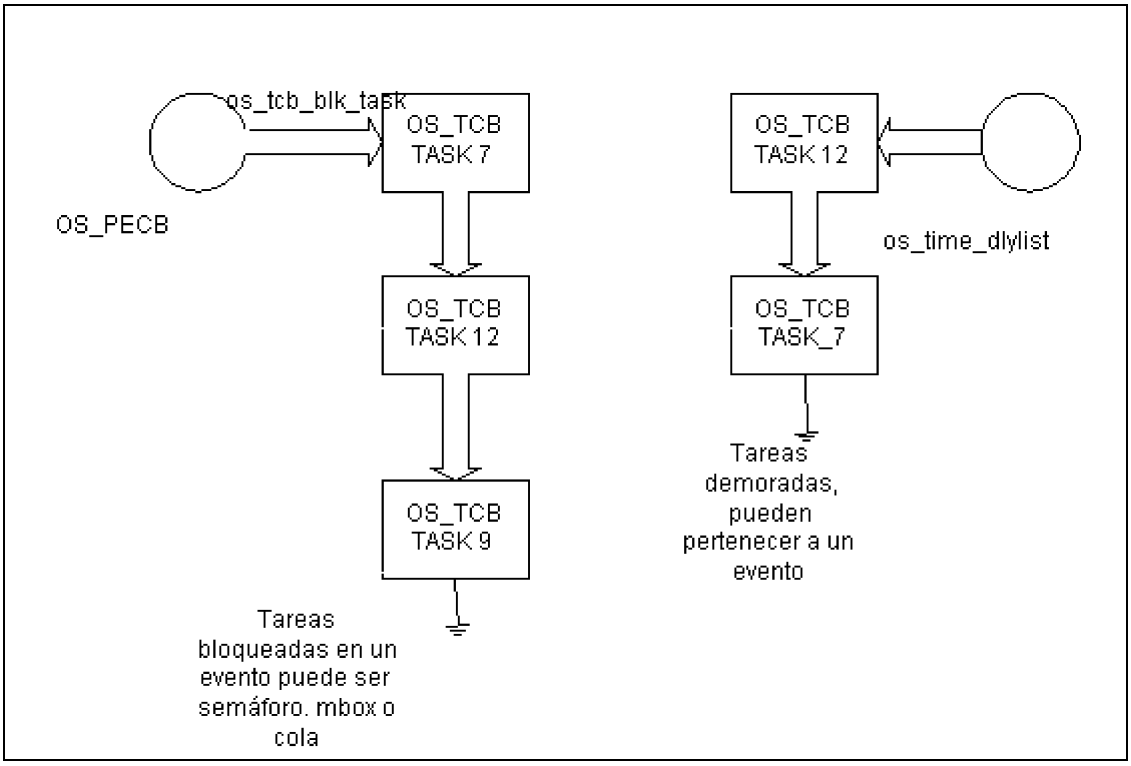
tail_offset: offset (desplazamiento) al próximo lugar de la cola donde se insertará el elemento. Para calcular esta posición de memoria debe multiplicarse *tail_offset* por el tamaño del elemento *elem_size* y sumar el resultado a *qstart.*

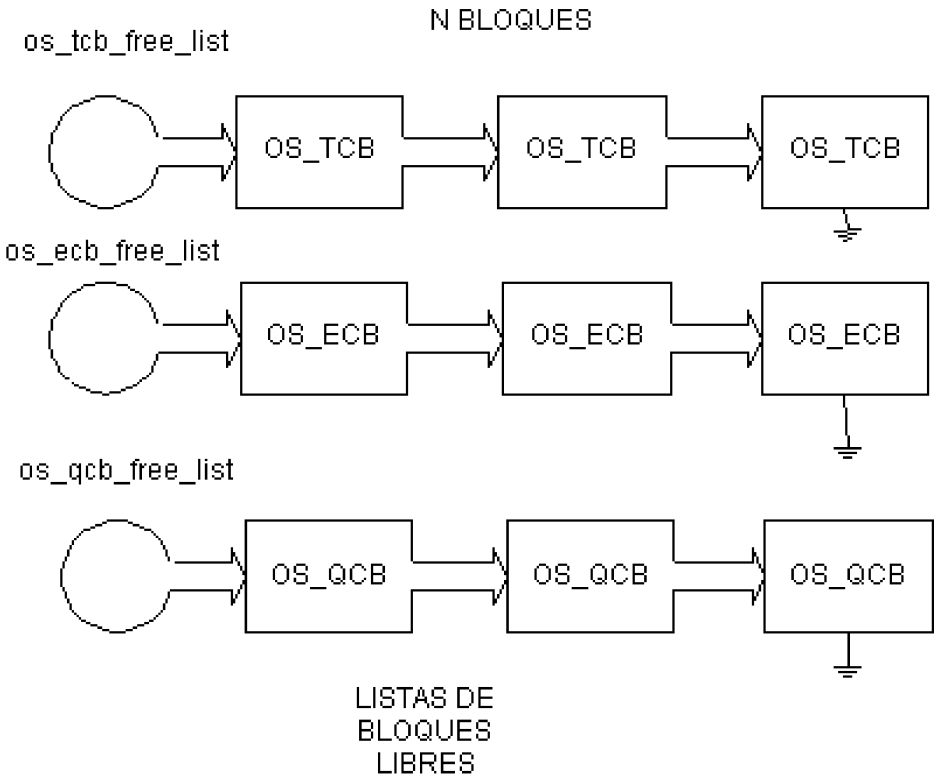
head_offset: offset al próximo elemento de la cola a extraer. Para calcular esta posición de memoria debe multiplicarse *head_offset* por el tamano del elemento *elem_size* y sumar el resultado a *qstart.*

elem_size: tamaño de cada elemento de la cola en bytes.

entries: número de entradas en la cola.


os_tcb_cur
next_rdy
os_tcb_run
OS_TCB
TASK 1
OS_TCB
TASK 2
prev_rdy
OS_TCB
TASK 3
os_tcb_currprio[0]
OS_TCB
TASK 5
OS_TCB
TASK 4
lista doblemente
enlazada de
tareas listas
Round Robin
en prioridad
cero


os_tcb_blk_task
OS_TCB
TASK 7
OS_PECB
OS_TCB
TASK 12
OS_TCB
TASK 9
Tareas
bloqueadas en un
evento puede ser
semáforo. mbox o
cola
OS_TCB
TASK 12
os_time_dlylist
OS_TCB
TASK_7
Tareas
demoradas,
pueden
pertenecer a un
evento



10.2. Variables globales

OS_TCB **os_tcb_cur*: apunta al OS_TCB del task corriente en ejecución.

OS_TCB **os_tcb_run*: puntero al OS_TCB del task a ejecutarse en el próximo cambio de contexto. Es decir que si *os_tcb_cur* es distinto de *os_tcb_run* es necesario plasmar un cambio de contexto a *os_tcb_cur*.

WORD *os_ctx_sw_ctr*: contador que se incrementa en uno en cada cambio de contexto.

LONG *os_idle_ctr*: contador que se incrementa dentro del loop del *idle_task* (tarea que se ejecuta cuando no hay otra cosa que hacer).

BOOLEAN *os_running*: flag, se pone en uno luego de llamar a *os_start().*

Variables locales al módulo rtkernel.c (statics var)

WORD *os_tcb_current_prio*: indica la prioridad del task en ejecución (de 0 a OS_LO_PRIO). Con este valor podemos saber en que nivel de lista circular de tareas estamos corriendo. Se usa como índice en *os_tcb_curprio*[ ]

BYTE *os_lock_nesting_level*: indica el nivel de anidamiento de lockeo del multitarea, si este valor es distinto de cero no se produce cambio de contexto.

BYTE *os_int_nesting*: indica el nivel de anidamiento de interrupciones, si este valor es distinto de cero no se produce anidamiento.

OS_TCB **os_tcb_free_list*: puntero al primer TCB de la lista de TCB's libres, a partir de allí se encadenan mediante el campo **next_free* correspondiente a OS_TCB.

OS_ECB **os_ecb_free_list*: puntero al primer ECB de la lista de ECB's libres, a partir de allí se encadenan mediante el campo **os_event_ptr* correspondiente a OS_ECB.

OS_QCB **os_qcb_free_list*: puntero al primer QCB de la lista de QCB's libres, a partir de allí se encadenan mediante el campo **os_qcb_ptr* correspondiente a OS_QCB.

TIME *os_time:* valor corriente en ticks del tiempo del sistema desde que comenzó a correr.

OS_TCB *os_tcb_tbl*[OS_MAX_TASKS]: array o tabla de TCB, corresponde al lugar en memoria de los TCB.

OS_ECB *os_ecb_tbl*[OS_MAX_EVENTS]: array o tabla de ECB, corresponde al lugar en memoria de los ECB.

OS_QCB *os_qcb_tbl*[OS_MAX_QCB]: array o tabla de QCB, corresponde al lugar en memoria de los QCB.

TASK_ID *idle_task_id:* identificador del idle task.

OS_TCB **os_tcb_task_tbl*[OS_MAX_TASKS]: array o tabla de punteros a los TCB

OS_TCB **os_tcb_dly_list:* puntero al primer elemento de la lista de tasks demorados.

OS_TCB **os_tcb_curprio_tbl* [OS_LO_PRIO]: array de punteros que apuntan al primer task a ejecutar en cada nivel de prioridad. Por medio de esta tabla es posible alcanzar todos los TCB de los ready tasks de todas las prioridades.

Existen tantas listas circulares de TCB doblemente enlazadas como número de prioridades existan. Por medio de esta tabla es factible alcanzar dichas listas circulares.

WORD *tick_size*: tick size en milisegundos

WORD old_bp: valor de BP antes de comenzar a correr el rtKERNEL

WORD old_sp: valor de SP antes de comenzar a correr el rtKERNEL

WORD old_ss: valor de SS antes de comenzar a correr el rtKERNEL

10.3. Descripción del propósito de cada función:

10.3.1. Módulo rtkernel.c:

void os_init()

1. Inicializa variables globales y arma las tablas del kernel y las listas de estructuras libres de TCB, ECB y QCB.
2. Setea el vector de interrupción de timmer a nuestro handler de timmer, denominado *new_tic_isr* que se encuentra en el módulo i186l_a.asm.
3. Crea el *os_task_idle* con la más baja prioridad existente. Este task se ejecutará siempre que no haya otra cosa que hacer.

Esta debe ser la primer función llamada del rtKERNEL.

void os_end()

1. Restituye el vector de interrupción de timer al handler original.
2. Restituye la periodicidad del timer a 55msec, es decir como estaba originalmente.
3. Restituye los valores originales de BP, SP,y SS.
4. Libera la memoria de stack del task corriente en execución, es decir el que llamó a *os_end().*

Esta función luego de ser llamada vuelve a la instrucción siguiente a *os_start()* terminando el operativo.
Es importante destacar que ***no deben declararse variables automáticas en esta función***, ya que si no, no se restauraría correctamente SS:SP por lo cual no se podría volver de aquí.

void os_dest_alltasks()

1. Destruye todos los tasks existentes por medio de la función *os_task_del ( )*;

*static void far os_task_idle (void *data)*

1. Código del idle task. Contiene un loop infinito con el contador *os_idle_ctr*. Se evitan cuestiones de competencia deshabilitando las interrupciones por medio de la macro DISABLE (cli) y se rehabilitan luego con ENABLE (sti).
2. El parámetro **data* no se emplea en absoluto.

void os_start (void)

1. Guarda los valores de BP, SP y SS.
2. Echa a correr el kernel por medio de la función *os_start_run* del módulo i186l_a.asm

void os_sched (void)

1. Si *os_lock_nesting && os_int_nesting && *os_tcb_cur != os_tcb_run* llama a *os_ctx_sw* en el módulo i186l_a.asm por medio de una interrupción de software (la definida en rtKERNEL interrupt)

void os_sched_lock (void)

1. Incrementa *os_lock_nesting*

void os_sched_unlock (void)

2. Decrementa *os_lock_nesting*

RETCODE os_tcb_init (PRIO *prio, void far *stk, void far *pmem, TASK_ID *tsk_id)*

1. Inicializa los campos de una entrada en la tabla os_tcb_tbk[]
2. Quita esta entrada de *os_tcb_free_list*
3. En **stk* se le pasa el puntero al tope del stack que se guarda en *stack_ptr*.
4. En **pmem* se le pasa el puntero a la memoria pedida para el stack.
5. En *tsk_id devuelve el identificador de task
6. En *prio* se le pasa la prioridad deseada del task
7. Inserta el TCB en la lista de ready correspondiente al nivel *prio* por medio de la función *os_tcb_ins_rdylist*.

void os_int_enter ( )

1. Incrementa *os_int_nesting*

Esta función se invoca desde *os_tick_isr()* en i186l_a.asm.

void os_int_exit ( )

1. Decrementa *os_int_exit ( )*
2. Si (*!os_int_nesting || os_lock_nesting)* Busca en las listas circulares de tareas listas a correr, a partir del nivel más alto de prioridad, es decir el cero, si existe alguna tarea. Cuando la encuentra redirecciona *os_tcb_run* para que apunte al TCB de dicha tarea y retorna -1.

Esta función se invoca desde *os_tick_isr()* en i186l_a.asm y al retornar un valor distinto de cero en ax, *os_tick_isr* realiza el cambio de contexto a *os_tcb_run.*

void os_task_del (TASK_ID task_id)

Destruye el task con identificador *task_id* sea cualfuere el estado en que se encuentra. Lo vuelve a insertar en la lista de *os_tcb_free_list*. Puede estar bloqueado en un evento, listo o bien ejecutándose. En este último caso la función no retorna y se invoca a *os_sched().*

void os_task_end ()

Destruye el task que invocó a esta función sea cualfuere el estado en que se encuentra. Lo vuelve a insertar en la lista de *os_tcb_free_list*. La función no retorna y se invoca a *os_sched().*

RETCODE os_task_change_prio (TASK_ID task_id, PRIO newprio)

Cambia la prioridad de un task dinámicamente, o sea que lo saca de la actual lista circular de tasks listos y lo pone en la nueva correspondiente a *newprio.* Si el task estuviera bloqueado en un evento o bien demorado, se le cambiará la prioridad de todas formas, pero en el caso de los eventos no será reubicada en la lista de tasks bloqueados. Por lo tanto hasta bien salga del evento, el cambio de prioridad no surtirá efecto. Esta función no ha sido depurada en su totalidad.

void os_tick_dly (LONG ticks)

Demora el task en ejecución *ticks* número de ticks del sistema del siguiente modo:

1. Guarda en el campo *dly_time* de TCB el parámetro *ticks.*
2. Saca al TCB correspondiente a la tarea de la lista de ready tasks de este nivel por medio de un llamado a *os_tcb_del_rdy_list (os_tcb_cur).*
3. La coloca en la lista de demorados *os_tcb_dlylist* por medio de la función *os_tcb_ins_dlylist (os_tcb_cur)*
4. Cambia el estado del task a OS_TASK_DLY.

void os_msec_dly (LONG msecs)

Demora el task en ejecución, es decir quien invocó a la tarea *msecs* número de msecs. Para ello llama a la función anterior con el cociente *msecs / ticks.*

void os_time_tick (void)

Procesa el tick del sistema.

Se la llama desde *new_tick_isr*.

1. Decrementa el campo de *dly_time* del primer task de la lista de *os_tcb_dlylist,* si esta no es nula. Si este valor llegara cero se lo quita de dicha lista y se coloca al TCB en la lista correspondiente de tasks listos a correr, por medio de *os_tcb_ins_rdy_list().*
2. Si la tarea estuviera también en alguna lista de eventos, se la quita de allí por medio de *os_tcb_quit_evnlist().*
3. Incrementa *os_time.*

void os_time_set (LONG ticks)

Setea el tiempo del sistema en número de ticks.

TASK_ID os_get_cur_taskid (void)

Retorna el identificador de la tarea que invocó al servicio.

LONG os_time_get (void)

Devuelve el tiempo del sistema en número de ticks.

RETCODE os_set_ticksize (WORD tick_size)

Permite ajustar el time slice del sistema, reprogramando el timer. El rango del tick size es de 12 a 54 milisegundos.

WORD os_get_ticksize (void)

Retorna el tamaño del tick en milisegundos.

*OS_ECB *os_sem_create (SWORD cnt)*

Retorna un puntero a un *OS_ECB.*
Crea un semáforo inicializado a *cnt*, quitando de la tabla os_ecb_free_list un ECB.

*RETCODE os_sem_wait (OS_ECB *pecb, WORD timeout)*

Realiza una operación de *wait* sobre el semáforo apuntado por *pecb.*

1. Decrementa el semáforo en una unidad.
2. Si el valor del semáforo es menor que cero, quita a la tarea de la lista de ready tasks y lo coloca en el lista del semáforo. Cambia el estado de la tarea a OS_TASK_SEM y si *timeout != 0,* coloca la tarea en la lista de tareas demoradas. Cambia de contexto.
3. Al despertarse por timeout o por un *signal* sobre el semáforo la tarea retorna de la función.
4. El parámetro *timeout* es en número de ticks del sistema.

*RETCODE os_q_dest (OS_ECB *pecb)*

Destruye una cola colocando el OS_QCB apuntado por *pecb*. Libera la memoria alojada para la cola y destruye el evento *pecb* por medio de *os_event_dest (pecb).*
Esta función debe llamarse al terminar el rtKERNEL ya que de lo contrario quedaría tomada la memoria de las colas.

*RETCODE os_event_dest (OS_ECB *pecb)*

Destruye un evento, purgando la cola *pecb* por medio de *os_tcb_purge_evnlist (pecb).* Reincerta *pecb* en *os_ecb_free_list.* Se emplea este servicio para destruir semáforos y mailboxes (buzones).

*RETCODE os_sem_signal (OS_ECB *pecb)*

1. Si el semáforo es menor que INT_MAX, incrementa el semáforo en uno.
2. Si el semáforo (campo *os_event_cnt*) es menor igual que cero, se despierta a la tarea apuntada por *os_tcb_blk_task* de *pecb* que será la de mayor prioridad de la cola del semáforo, o al menos la primera puesta en la cola. Es decir se saca el OS_TCB de esta cola de eventos y se coloca en la lista de ready tasks correspondiente a la prioridad del task, en último lugar, inmediatamente antes del *os_tcb_cur* de modo que será el último de la lista circular en ejecutarse. Inmediatamente se hace un cambio de contexto por medio de *os_sched().*
3. En todo momento *os_event_cnt*, cuando es negativo da el número de tareas bloqueada en el semáforo.

*OS_ECB *os_mbox_create (void *msg)*

Crea e inicializa un mailbox con lo apuntado por *msg.* Para ello quita un OS_ECB de la lista *os_ecb_free_list* y retorna puntero al mismo.

*void *os_mbox_receive (OS_ECB *pecb, WORD timeout, RETCODE *err)*

Espera recibir un mensaje en el mailbox apuntado por *pecb.*

1. Si el buzón está lleno se retorna el puntero al mensaje y la tarea retorna de inmediato.
2. Si el buzón está vacío se bloquea a la espera de un mensaje. Para ello se quita el OS_TCB de la tarea de la lista de ready tasks y se lo coloca en la cola (lista) del evento por medio de *os_tcb_ins_evnlist (pecb).* Asimismo se coloca el TCB en la lista de tareas demoradas o retardads, si el parámetro *timeout* (en número de ticks) es distinto de cero. Inmediatamente se cambia de contexto a la próxima tarea lista.
3. La tarea se despierta cuando se hace un *os_mbox_send ( )* sobre este *pecb.*

*RETCODE os_mbox_send (OS_ECB *pecb, void *msg)*

Envía un mensaje al buzón apuntado por *pecb.*

1. Si el buzón se encuentra ocupado el mensaje se descarta y se retorna OS_MBOX_FULL a la tarea invocante del servicio.

2. Se copia el puntero al mensaje en el buzón.
3. Si existiera alguna tarea bloqueada en la cola de este evento se la despierta y se cambia de contexto por medio de *os_sched( ).*
4. Si existiera más de una tarea bloqueada en esta cola, se despiertan a todas las tareas y todas se llevan el mensaje (broadcast).

*OS_ECB *os_q_create (WORD qsize, BYTE elem_size)*

Crea una cola circular de tamaño *qsize* con un tamaño de cada elemento de *elem_size.* Aloja la memoria por una llamada a far alloc. Devuelve puntero a cola. La cola es del tipo FIFO (first input, first output).

*void *os_q_read (OS_ECB *pecb, WORD timeout, RETCODE *err)*

Lee un elemento de la cola si ésta no se encuentra vacía.
De lo contrario la tarea queda bloqueada en la lista de este evento, ordenada por prioridad y FIFO. Cuando otra tarea haga un *os_q_write( ),* en este *pecb* la tarea será despertada.
1. Si el parámetro *timeout* (expresado en número de ticks del sistema) es distinto de cero, el OS_TCB correspondiente a esta tarea se coloca en la lista de *os_tcb_dly_list( ).*
2. Si el timeout expira la tarea se despierta por *timeout* número de ticks.
3. Retorna código de error en *err.*

*void *os_q_write (OS_ECB *pecb, void *msg)*

Escribe un elemento en la cola apuntada por *pecb.*
1. Si hay tareas bloqueadas en la cola de este evento, se despertará a la primera de la lista. Es decir la de mayor prioridad, o la primera que llegó. Luego se llama a *os_sched()* para concretar el cambio de contexto, si fuera necesario.
2. Si la cola se encuentra llena, el mensaje u objeto se descarta.
3. Cabe destacar que se aloja memoria para un elemento más, para no perder el puntero al mensaje cuando la cola está llena, y se hace un *os_q_write()* antes de retirar el mensaje o elemento.

*void os_tcb_ins_rdylist (OS_TCB *ptcb)*

Inserta el OS_TCB apuntado por *ptcb* en la cola ready correspondiente a la prioridad *pecb->prio*. El lugar a insertar será siempre anterior a lo apuntado por *os_tcb_cur* por lo tanto la tarea en cuestión será la última a ejecutar de este torneo, dado que el puntero *os_tcb_cur* tiene que dar toda la vuelta a la lista circular de esta prioridad, doblemente enlazada. Cambia el estado *pecb->task_stat* a OS_TASK_RDY.

*void os_tcb_del_rdylist (OS_TCB *ptcb)*

Quita el OS_TCB apuntado por *ptcb* de la cola de ready correspondiente a la prioridad *pecb->prio.* Si fuera necesario se cambia *os_tcb_run*, dado que este task *pecb->task_id* puede ser el corriente en ejecución. En ese caso busca cual será el próximo en ejecutarse, y cambia el estado del favorecido a OS_TASK_RUN.

*void os_tcb_ins_evnlist (OS_TCB *ptcb)*

Inserta el OS_TCB apuntado por *ptcb* en la lista de tasks bloqueados del evento *ptcb->pecb* , por orden de prioridad (los de prioridad más alta son los primeros que salen) y de llegada (FIFO).

*void os_tcb_purge_evnlist (OS_ECB *pecb)*

Purga la lista de tareas bloqueadas apuntadas por *pecb.* A medida que extra cada OS_ECB lo inserta en la lista circular de listos correspondientes por medio de la función *os_tcb_ins_rdylist ( ).*

*void os_tcb_del_evnlist (OS_ECB *pecb)*

Extrae la primer tarea de la lista de tareas bloqueadas apuntada por *pecb*, es decir aquella apuntada por *pecb->os_tcb_blk_task* que será la primer tarea de la más alta prioridad de la lista del evento *pecb*.

*void os_tcb_ins_dlylist (OS_TCB *ptcb)*

Inserta en la lista *os_tcb_dlylist* la tarea demorada en *dly_time* número de ticks, en el lugar que le corresponda según este valor (ver algoritmo plasmado en el código).

*void os_tcb_del_dlylist (OS_TCB *ptcb)*

Quita de la lista *os_tcb_dlylist* la tarea apuntada por *ptcb* y reajusta los *timeouts* de las demás tareas de la lista según el algoritmo.

*void os_tcb_quit_evnlist (OS_TCB *ptcb)*

Extrae de la lista *ptcb->pecb->os_tcb_blk_task* la tarea correspondiente a *ptcb*

10.3.2. Módulo i186l_c.c

*RETCODE os_task_create (void (far *task)(void *pd), void *pdata, WORD stksize, PRIO p, TASK_ID *tsk_id)*

Crea una tarea nueva por medio de la llamada a *os_tcb_init ( ).*
1. Aloca lugar para el stack por medio de la llamada al DOS *farmalloc (stksize).*
2. Luego inicializa el stack de la futura tarea. Simula la llamada a función *task* con parámetros guardando el offset y el segmento de *pdata*. Como dirección *far* de retorno se guarda la de *os_task_end*. Es decir que al terminar la función, *task* se ejecutará *os_task_end()*. Luego se guarda PSW con las interrupciones habilitadas. A continuación la dirección *far* de retorno de *task().* Ahora se guardan los registros AX, CX, DX, BX, SP, BP, SI, DI, ES (todos estos con el valor 0x00) y DS (con el valor

corriente del DS). De esta forma el stack queda listo para volver de la rutina de interrupción por medio de un IRET (*return from interrupt*).
3. Si el OS está corriendo (*os_running = 1*) realiza el cambio de contexto si fuera necesario por medio de *os_sched().*

10.3.3. Módulo i186l_a.asm

os_start_run

Se la llama para empezar a correr la primer tarea del operativo. Cambia al contexto de *os_tcb_run.*

1. Se hace *os_tcb_cur = os_tcb_run*
2. SS:SP = *os_tcb_cur->stack_ptr.*
3. POP DS, ES
4. POPA

os_ctx_sw

Se la llama desde una tarea por medio de una interrupción (ver macro rtKERNEL), en la función *os_sched()*. Cambia al contexto de *os_tcb_run.*

1. PUSHA
2. PUSH ES, PUSH DS
3. Carga DS con DGROUP
4. Hace *os_tcb_cur->stk_ptr* = SS:SP
5. Se hace *os_tcb_cur = os_tcb_run*
6. SS:SP = *os_tcb_cur->stack_ptr.*
7. POP DS, ES
8. POPA

new_tick_isr

Se llama desde la interrupción de timer. Se encadena con la interrupción de DOS 0x81 si es necesario para mantener la hora del sistema inalterada.

1. PUSHA
2. PUSH ES, PUSH DS
3. Carga DS con DGROUP
4. Hace int 0x81 si es necesario para mantener la hora del DOS.
5. Llama a *os_int_enter()* en el módulo rtkernel.c
6. Llama a *os_time_tick()* en el módulo rtkernel.c
7. Llama a *os_int_exit()* en el módulo rtkernel.c. Si ésta devuelve un valor distinto de cero en AX no realiza cambio de contexto (es el caso de *os_tcb_cur = os_tcb_run*) hace POP DS, POP ES, POPA, IRET. De lo contrario:

8. Hace *os_tcb_cur->stk_ptr* = SS:SP
9. Se hace *os_tcb_cur = os_tcb_run*
10 SS:SP = *os_tcb_cur->stack_ptr.*

11 OP DS, ES
12 POPA

10.3.4. Descripción del demo.

El demo es un programa de prueba que se empleó para depurar y testear el kernel. Consta de un total de ocho tareas funcionando concurrentemente, las cuales usan intensivamente los recursos de la máquina, las primitivas del kernel y algunos servicios del DOS.

A continuación se describe el programa ex3.c:

main()

Aquí se crea el semáforo para acceder a los servicios de DOS, *dos_sem.* Se crean dos buzones *key_mbox_ptr1* y *key_mbox_ptr2.* Se crean un total de tres colas *rx_qp1, rx_qp2* y t*x_qp* para las interfaces series. Por último se crea la tarea *stat_task* que es la encargada de medir las estadísticas.

Descripción de tareas.

key_task()

Es la encargada de procesar la entrada de teclado. Las teclas procesadas son las siguientes:

F1: le envía un mensaje al buzón *key_mbox_ptr1*.
F2 le envía un mensaje al buzón *key_mbox_ptr2.*
PAGE_UP incrementa el tick size en una unidad
PAGE_DOWN decrementa el tick size en una unidad
F10 finaliza el operativa
Cualquier otra tecla extendida se ignora
Las restantes teclas se encolan en *tx_qp* para ser enviadas por la puerta serie por la tarea *tx_task()*

A_task()

Se queda bloqueada en el buzón *key_mbox_ptr1* a la espera de algún mensaje con un timeout de 20 ticks.

B_task()

Se queda bloqueada en el buzón *key_mbox_ptr2* a la espera de algún mensaje con un timeout de 20 ticks.

ran_task()

Imprime tres caracteres al azar en la ventana superior derecha de la pantalla, se demora 1 tick en cada loop. No tiene comunicación con las tareas restantes.

rx1_task()

Lee caracteres de la cola *rx_qp1* y los envía a la ventana inferior izquierda de la pantalla. Estos caracteres son puestos en la cola por el handler de puerta serie y son los que se reciben del *com1*.

rx2_task()

Lee caracteres de la cola *rx_qp2* y los envía a la ventana inferior izquierda de la pantalla. Estos caracteres son puestos en la cola por el handler de puerta serie y son los que se reciben del *com2.*

tx_task()
Lee caracteres de la cola *tx_qp* y los envía por la puerta serie (*com1)* y también a la pantalla. Los caracteres son puestos en dicha cola por *task_key().*

stat_task()
Crea a las tareas restantes. Imprime estadísticas cada 10 ticks del sistema.

Las tareas descritas hasta aquí hacen uso de las funciones escritas en el módulo *util.c*, las mismas son:

disp_char (BYTE x, BYTE y, char c)
Imprime un caracter *c* en la posición columna *x* fila *y* de la pantalla.

*disp_str (BYTE x, BYTE y, char *s)*
Imprime un string apuntado por *s* en la posición columna *x* fila *y* de la pantalla.

void init_uart (long baud_rate,
BYTE num_data_bits,
BYTE num_stop_bits,
BYTE parity,
unsigned base_address,
BYTE irq_vect)

Setea la uart con los parámetros dados y el handler de interrupciones de puerta serie.

interrupt my_uart_handler (void)

handler de la interrupción 3 y 4 de hardware (vectores 11 y 12).Atiende las interrupciones de la uart para recepción.

void goto_xy (BYTE x, BYTE y, BYTE page)

Coloca el cursor en la columna *x,* posición *y*, página *page* de pantalla. Usa servicio de BIOS.

void scroll_up (
BYTE lines,
BYTE x1,
BYTE y1,
BYTE x2,
BYTE y2,

BYTE color
)

Hace un scroll de n *lines* para arriba de la ventana definida por *x1*, *y1*, *x2* e *y2*.

void scroll_down (
BYTE lines,
BYTE x1,
BYTE y1,
BYTE x2,
BYTE y2,
BYTE color
)

Hace un scroll de n *lines* para abajo de la ventana definida por *x1*, *y1*, *x2* e *y2*.

*RETCODE init_screen (char *fname)*

Lee un archivo de texto y lo imprime en la pantalla.

11. LISTADO DE FUENTES DEL RTKERNEL 1.00

Se adjuntan a continuación todos los listados fuentes del código del kernel.

11.1. rtk.mak

```
#
# rtKERNEL 1.00
#
# MAKEFILE
#
# FILE: rtk.mak
#
# 30-8-94
#
# Guillermo Pablo Tomasini
#
# Se crea una bliblioteca llamada rtkernel.lib apta para ser linkeada con aplicación de usuario
# modelo: large

.AUTODEPEND

#       *Translator Definitions*
CC = bcc +rtk.CFG
TASM = TASM
TLIB = tlib
TLINK = tlink
LIBPATH = c:\BC\LIB;d:\bc\lib
INCLUDEPATH = c:\BC\INCLUDE;d:\bc\include


#       *Explicit Rules*
rtkernel.lib:: rtk.cfg rtkernel.obj
    tlib rtkernel /c -+rtkernel.obj

rtkernel.lib:: rtk.cfg i186l_a.obj
    tlib rtkernel /c -+i186l_a.obj

rtkernel.lib:: rtk.cfg i186l_c.obj
    tlib rtkernel /c -+i186l_c.obj

#       *Individual File Dependencies*
rtkernel.obj: rtk.cfg rtkernel.c
    $(CC) -c rtkernel.c

i186l_c.obj: rtk.cfg i186l_c.c
    $(CC) -c i186l_c.c

i186l_a.obj: rtk.cfg i186l_a.asm
    $(TASM) /MX /ZI /O I186L_A.ASM,I186L_A.OBJ

#       *Compiler Configuration File*
rtk.cfg: rtk.mak
  copy &&|
-ml
```

-1
-v
-G
-O
-Og

11.2. rtkernel.h

```
/*
-------------------------------------------------------------------------------
                                rtKERNEL
                        PC Real-Time Multitasking Operating System
                                KERNEL


  File : RTKERNEL.C

  Author:   Guillermo Pablo Tomasini

  Date:     20-8-94

-------------------------------------------------------------------------------
*/
#ifndef     rtkernel_h
#define     rtkernel_h

#include    "80186l.h"

/*
-------------------------------------------------------------------------------
                            rtKERNEL CONFIGURATION
-------------------------------------------------------------------------------
*/

#define rtKERNEL                0x80        // Interrupt vector assigned to rtKERNEL

#define OS_MAX_TASKS            64      // Maximum number of tasks in your application
#define OS_MAX_EVENTS           20      // Maximum number of event control blocks in your
application
#define OS_MAX_QS               5       // Maximum number of queue control blocks in your
application

#define OS_IDLE_TASK_STK_SIZE 1024   // Idle task stack size (BYTEs)

#define     OS_LO_PRIO                  4   // IDLE task priority

/*
-------------------------------------------------------------------------------
                            rtKERNEL ERRROR CODES
-------------------------------------------------------------------------------
*/

#define     OS_NO_ERR               0   // ERROR CODES
#define     OS_TIMEOUT              10
#define     OS_NO_MEMORY            11
#define     OS_MBOX_FULL            20
#define     OS_Q_FULL               30
#define     OS_Q_NULL           31
#define     OS_PRIO_ERR             41
#define     OS_SEM_ERR              50
#define     OS_SEM_OVF              51
#define     OS_TASK_DEL_ERR         60
#define     OS_TASK_DEL_IDLE        61
#define     OS_NO_MORE_TCB          70
#define     OS_TSK_NO_EXIST         71
```

```
#define    OS_INTERNAL_ERROR  80
#define    OS_NULL_PECB       81
#define    OS_BAD_TICKSIZE    90

#define    NULL_ID            -1
#define    NULL_PRIO          -1


/*$PAGE*/
/*
---------------------------------------------------------------------------
                        EVENT CONTROL BLOCK (ECB)
---------------------------------------------------------------------------
*/

typedef    WORD    TASK_ID;
typedef    SBYTE   PRIO;
typedef    LONG    TIME;

typedef    WORD    RETCODE;

typedef struct os_event
{
    SWORD          os_event_cnt;      // count of used when event is a semaphore
    void           *os_event_ptr;     // pointer to message or queue structure
//  WORD           os_task_cnt;       // blocked task's number in this event
    struct os_tcb  *os_tcb_blk_task;  // first task blocked in this event
}                  OS_ECB;

/*
---------------------------------------------------------------------------
                        rtKERNEL TASK CONTROL BLOCK (TCB)
---------------------------------------------------------------------------
*/

// TASK STATUS

typedef enum task_state
{
    OS_TASK_RUN,       // running
    OS_TASK_RDY,       // ready to run
    OS_TASK_SEM,       // pending on semaphore
    OS_TASK_MBOX,      // pending on mailbox
    OS_TASK_Q,         // pending on queue
    OS_TASK_DLY,       // task delayed
    OS_TASK_NULL       // task no exist
};

typedef struct os_tcb
{
    void far           *stack_ptr;      // pointer to current top of stack
    void far           *allocmem_ptr;   // pointer to alloc memory
    enum task_state    task_stat;       // task status
    PRIO               prio;            // task priority (0 == highest, 3 == lowest)
    TASK_ID            task_id;         // task id
    TIME               dly_time;        // nbr. ticks to delay task or, timeout waiting for event
    OS_ECB             *pecb;           // pointer to event control block (ECB)
    struct os_tcb      *next_free;      // ptr. to next TCB in the TCB free list
    struct os_tcb      *next_rdy;       // ptr. to next task ready in the same prio. level
```

```
    struct os_tcb       *prev_rdy;          // ptr. to prev. task ready in the same prio. level
    struct os_tcb       *next_blk;          // ptr. to next blocked task if exist
    struct os_tcb       *next_dly;        // ptr. to next delayed task if exist
}                                           OS_TCB;

/*
------------------------------------------------------------------------------
                        rtKERNEL QUEUE CONTROL BLACK (QCB)
------------------------------------------------------------------------------
*/

typedef struct os_q
{
    struct os_q     *os_qcb_ptr;    // link to next queue control block in list of free blocks
    void            *qstart;        // pointer to start of queue data
    BYTE            tail_offset;    // offset to where the next element will be inserted
    BYTE            head_offset;    // offset to where the next element will be extracted
    BYTE            elem_size;      // element size
    WORD            qsize;          // size of queue (maximum number of entries)
    WORD            entries;        // current number of entries in the queue
}               OS_QCB;

/*$PAGE*/
/*
------------------------------------------------------------------------------
                        rtKERNEL GLOBAL VARIABLES
------------------------------------------------------------------------------
*/

                                            // SYSTEM VARIABLES
extern WORD         os_ctx_sw_ctr;      // Counter of number of context switches
extern LONG         os_idle_ctr;        // Idle counter
extern WORD         os_running;         // Flag indicating that kernel is running
extern OS_TCB       *os_tcb_cur;        // Pointer to currently running TCB
extern OS_TCB       *os_tcb_high_rdy;   // Pointer to highest priority TCB ready to run
extern OS_TCB       *prio_tbl[];        // Table of pointers to all created TCBs
extern WORD         divisor;

/*
------------------------------------------------------------------------------
                        rtKERNEL EXPORTABLE SERVICES
------------------------------------------------------------------------------
*/

void        os_init             ( void );
void        os_start            ( void );
void        os_end                  ( void );
void        os_start_run        ( void );
void        os_sched            ( void );
void        os_sched_lock       ( void );
void        os_sched_unlock ( void );

RETCODE os_task_create ( void (far *task)(void *pd), void *pdata, WORD stksize, PRIO
prio,TASK_ID                            tsk_id );

RETCODE os_task_del ( TASK_ID tsk_id );

RETCODE os_task_change_prio ( TASK_ID tsk_id, PRIO newp);
```

```
void        os_task_end         ( void );

void        os_tick_dly         ( LONG ticks );
void        os_msec_dly         ( LONG msecs );
void        os_time_tick        ( void );
void        os_time_set         ( LONG ticks );
LONG        os_time_get         ( void );

RETCODE     os_set_ticksize     ( WORD ticksize );
WORD        os_get_ticksize     ( void );

OS_ECB      *os_sem_create      ( SWORD value );
RETCODE     os_sem_signal       ( OS_ECB *pevent );
RETCODE     os_sem_wait         ( OS_ECB *pevent, TIME timeout );

OS_ECB      *os_mbox_create( void *msg );
RETCODE     os_mbox_send        ( OS_ECB *pevent, void *msg );
void        *os_mbox_receive    ( OS_ECB *pevent, TIME timeout, RETCODE *err );

OS_ECB      *os_q_create        ( WORD qsize, BYTE esize );
BYTE        os_q_write          ( OS_ECB *pevent, void *msg );
void        *os_q_read          ( OS_ECB *pevent, TIME timeout, RETCODE *err );

RETCODE     os_event_dest           ( OS_ECB *pecb );

TASK_ID     os_get_cur_taskid   ( void );

#define     os_sem_dest( pecb )     os_event_dest( pecb )
#define     os_mbox_dest( pecb )    os_event_dest( pecb )
RETCODE     os_q_dest ( OS_ECB *pecb );

/*
---------------------------------------------------------------------------
  EXPORTABLE FUNCTIONS FOR OTHER MODULES
---------------------------------------------------------------------------
*/

void        os_int_enter        ( void );
WORD        os_int_exit         ( void );
RETCODE     os_tcb_init         ( PRIO priority, void far *stk, void far *mem, TASK_ID *tsk_id );

#endif
```

11.3. 80186l.h

```
#ifndef     i80186l_h
#define     i80186l_h
/*
*********************************************************************************
*                                    rtKERNEL 1.00
*                    Microcomputer Real-Time Multitasking Kernel
*
*                                 80186/80188 Specific code
*                                LARGE MEMORY MODEL
*
* File : 80186L.H
*********************************************************************************
*/

/*
*********************************************************************************
*                                    CONSTANTS
*********************************************************************************
*/

#define     FALSE     0
#define     TRUE 1

/*
*********************************************************************************
*                                     MACROS
*********************************************************************************
*/
#define  DISABLE() asm      cli
#define  ENABLE()  asm      sti

#define  OS_TASK_SW()   asm  INT   rtKERNEL

/*
*********************************************************************************
*                                    DATA TYPES
*********************************************************************************
*/

typedef unsigned char     BOOLEAN;
typedef unsigned char     BYTE;          // Unsigned  8 bit quantity
typedef signed   char SBYTE;         // Signed    8 bit quantity
typedef unsigned int  WORD;          // Unsigned 16 bit quantity
typedef signed   int     SWORD; // Signed   16 bit quantity
typedef unsigned long     LONG;          // Unsigned 32 bit quantity
typedef signed   long SDWORD;    // Signed   32 bit quantity

#endif
```

11.4. rtkernel.c

```
/*
---------------------------------------------------------------------------
                        rtKERNEL 1.00
                PC Real-Time Multitasking
                        KERNEL


 File :  RTKERNEL.C

 Author:    Guillermo Pablo Tomasini

 Date:      20-8-94

---------------------------------------------------------------------------
*/


#include   <alloc.h>  //because NULL
#include   <dos.h>    //because getvect(), setvect(), FP_OFF & FP_SEG
#include   <limits.h> //becuase INT_MAX
#include   <mem.h>  //because _fmemcpy()

#include    "80186l.h"
#include    "rtkernel.h"

#ifdef   TURBOC
#pragma  inline
#endif

/*
---------------------------------------------------------------------------
                            CONSTANTS
---------------------------------------------------------------------------
*/

#define    WCONTROL                (0x00<<6) | (0x03<<4) | (0x03<<1) | 0x00      //word control for
8253
#define    BASE_ADRESS_TIMER   0x40                                          //base address of 8253



/*
---------------------------------------------------------------------------
                        EXTERN VARIABLES
---------------------------------------------------------------------------
*/

extern void far     new_tick_isr ( void );
extern void far     os_ctx_sw ( void );


/*
---------------------------------------------------------------------------
                        GLOBAL VARIABLES
---------------------------------------------------------------------------
*/
OS_TCB       *os_tcb_cur;            // pointer to current TCB
```

```
OS_TCB      *os_tcb_run;          // ptr to higher priority task
WORD        os_ctx_sw_ctr;        // context switch counter
LONG        os_idle_ctr;          // counter used in os_task_idle()
WORD        os_running;           // flag indicating that rtKERNEL has been started and is running
WORD        divisor     = 0xffff; //by default, 18 interrupts by second
LONG        count_ticks;          // used by i186l_a for count ticks

/*
---------------------------------------------------------------------------
                        LOCAL VARIABLES
---------------------------------------------------------------------------
*/

static WORD     os_tcb_current_prio;                // priority of running task
static BYTE     os_lock_nesting;                    // multitasking lock nesting level
static BYTE     os_int_nesting;                     // interrupt nesting level
static OS_TCB   *os_tcb_free_list;                      // pointer to list of free TCBs
static OS_ECB   *os_ecb_free_list;                  // pointer to list of free EVENT control
blocks
static OS_Q     *os_q_free_list;                    // pointer to list of free QUEUE control
blocks
static TIME     os_time;                            // current value of system time (in ticks)
static OS_TCB   os_tcb_tbl [ OS_MAX_TASKS ];        // table of TCBs
static OS_ECB   os_ecb_tbl [ OS_MAX_EVENTS ];       // table of EVENT control blocks
static OS_Q     os_q_tbl [ OS_MAX_QS ];             // table of QUEUE control blocks
static TASK_ID  idle_task_id;                       // idle task id
static OS_TCB   *os_tcb_task_tbl [ OS_MAX_TASKS ];  // ptr table to created tasks
                                                    // order by task_id
static OS_TCB   *os_tcb_dlylist;                    // ptr to delay task list
static OS_TCB   *os_tcb_curprio [ OS_LO_PRIO ];     // ptr array to current task
                                                    // in each priority level
static WORD     tick_size   = 55;                   // by default 55.5 msec

static WORD     old_bp;
static WORD     old_sp;
static WORD     old_ss;

/*
---------------------------------------------------------------------------
                    LOCAL FUNCTION PROTOTYPES
---------------------------------------------------------------------------
*/

static void     os_task_idle            ( void *data    );
static void     os_tcb_ins_rdylist      ( OS_TCB *ptcb );
static void     os_tcb_del_rdylist      ( OS_TCB *ptcb );
static void     os_tcb_ins_evnlist      ( OS_TCB *ptcb );
static void     os_tcb_purge_evnlist    ( OS_ECB *pecb);
static void     os_tcb_del_evnlist      ( OS_ECB *pecb);
static void     os_tcb_ins_dlylist      ( OS_TCB *ptcb );
static void     os_tcb_del_dlylist      ( OS_TCB *ptcb );
static void     os_tcb_quit_evnlist     ( OS_TCB *ptcb );
static void     os_dest_alltasks        ( void );

static void     interrupt (*old_tick_isr)(void);                    // old tick int. handler


/*
---------------------------------------------------------------------------
```

```
                        rtKERNEL INITIALIZATION
--------------------------------------------------------------------------------
*/

void os_init ( void )
{
    int i;

    DISABLE();
    old_tick_isr = getvect ( 0x08 );                                    // get MS-DOS's tick vector
    setvect ( 0x81, old_tick_isr );                                     // store MS-DOS's tick to chain
    setvect ( rtKERNEL, (void interrupt (*)(void))os_ctx_sw );  // rtKERNEL context switch vector
    setvect ( 0x08, (void interrupt (*)(void))new_tick_isr );       // new int. handler
    ENABLE();

    os_time             = 0L;
    os_tcb_cur          = NULL;
    os_tcb_run          = NULL;
    os_int_nesting      = 0;
    os_lock_nesting     = 0;
    os_running          = FALSE;
    os_idle_ctr         = 0L;
    os_ctx_sw_ctr       = 0;

    for ( i = 0; i < OS_MAX_TASKS; i++ )                                // init. list of free TCBs
    {
        if ( i < OS_MAX_TASKS-1 )
            os_tcb_tbl [ i ].next_free= &os_tcb_tbl [ i + 1 ];
        else
            os_tcb_tbl [ i ].next_free= NULL;

        os_tcb_tbl [ i ].task_id        = i;
        os_tcb_tbl [ i ].task_stat      = OS_TASK_NULL;
        os_tcb_task_tbl [ i ]           = NULL;
    }

    os_tcb_free_list                    = os_tcb_tbl;

    for ( i = 0; i < (OS_MAX_EVENTS - 1); i++ )                 // init. list of free ECB
    {
        os_ecb_tbl [ i ].os_event_ptr       = &os_ecb_tbl [ i + 1 ];
        os_ecb_tbl [ i ].os_tsk_cnt         = 0;
        os_ecb_tbl [ i ].os_tcb_blk_task    = NULL;
    }

    os_ecb_tbl [ OS_MAX_EVENTS - 1 ].os_event_ptr = NULL;
    os_ecb_free_list                        = os_ecb_tbl;

    for ( i = 0; i < (OS_MAX_QS - 1); i++ )                         // init. list of free QCB
        os_q_tbl [ i ].os_q_ptr = &os_q_tbl [ i + 1 ];

    os_q_tbl [ OS_MAX_QS - 1 ].os_q_ptr     = NULL;
    os_q_free_list                          = os_q_tbl;

    for ( i = 0; i < OS_LO_PRIO; i++ )
        os_tcb_curprio [ i ]   = NULL;

    os_task_create ( os_task_idle, NULL, 512, OS_LO_PRIO-1, &idle_task_id );
}
```

```
/*
--------------------------------------------------------------------------------
                        rtKERNEL END
--------------------------------------------------------------------------------
*/

void os_end ( void )
{
//WARNING: no allowed automatic var here

    os_running     = FALSE;
    DISABLE();
    outportb ( BASE_ADRESS_TIMER + 3, WCONTROL );   // 8253 mode
    outportb ( BASE_ADRESS_TIMER + 0, 0xff );       // 1ro. low
    outportb ( BASE_ADRESS_TIMER + 0, 0xff );       // 2do. high
    os_dest_alltasks ( );                           //destroy all tasks
    _BP = old_bp;                                   //restore old bp
    _SP = old_sp;                                   //restore old sp
    _SS     = old_ss;                               //restore old ss
    setvect ( 0x08, old_tick_isr );                 //restore old int. handler
    farfree ( os_tcb_cur->allocmem_ptr );           //free memory of cur. task
    ENABLE();
}                                                   //return to next inst. after os_start();

/*
--------------------------------------------------------------------------------
                        DESTROY ALL TASK'S
--------------------------------------------------------------------------------
*/

void os_dest_alltasks ( void )
{
    int i;
    OS_TCB *ptcb;

    for ( i=0; i<OS_MAX_TASKS; i++)
        if ( (ptcb = os_tcb_task_tbl [ i ]) != NULL && ptcb != os_tcb_cur )
            os_task_del ( ptcb->task_id );
}

/*
--------------------------------------------------------------------------------
                            IDLE TASK
--------------------------------------------------------------------------------
*/

static void far os_task_idle ( void *data )
{
    data = data;

    while (1)
    {
        DISABLE();
        os_idle_ctr++;
        ENABLE();
    }
}
```

```
/*
------------------------------------------------------------------------
                        START MULTITASKING
------------------------------------------------------------------------
*/

void os_start ( void )
{
//WARNING: no allowed automatic var. here

    old_bp      = _BP;                          // salvage bp
    old_sp      = _SP;                          // freeze return address
    old_ss      = _SS;                          // freeze return address
    os_running  = 1;
    os_start_run();                             // no return
}


/*
------------------------------------------------------------------------
                        rtKERNEL SCHEDULER
------------------------------------------------------------------------
*/

void os_sched ( void )
{
    DISABLE();

    if ( !( os_lock_nesting | os_int_nesting ) )// task scheduling must be enabled and not ISR level
        if ( os_tcb_run != os_tcb_cur )
        {
            os_ctx_sw_ctr++;        // increment context switch counter
            OS_TASK_SW();           // context switch through interrupt
        }
    ENABLE();
}

/*
------------------------------------------------------------------------
                        PREVENT SCHEDULING
------------------------------------------------------------------------
*/

void os_sched_lock ( void )
{
    DISABLE();
    os_lock_nesting++;              // increment lock nesting level
    ENABLE();
}

/*
------------------------------------------------------------------------
                        ENABLE SCHEDULING
------------------------------------------------------------------------
*/

void os_sched_unlock ( void )
{
```

```
    DISABLE();
    if ( os_lock_nesting )
    {
        os_lock_nesting--;                          // decrement lock nesting level
        if ( !( os_lock_nesting | os_int_nesting) )  // See if scheduling re-enabled and not an ISR
        os_sched();                                 // See if a higher priority task is ready
        else
        ENABLE();
}
    else
        ENABLE();
}


/*
------------------------------------------------------------------------------
                            INITIALIZE TCB
------------------------------------------------------------------------------
*/

RETCODE os_tcb_init ( PRIO prio, void far *stk, void far *pmem, TASK_ID *tsk_id )
{
    OS_TCB *ptcb;

    *tsk_id     = NULL_ID;

    if ( prio > OS_LO_PRIO - 1 )
        return OS_PRIO_ERR;

    DISABLE();
    ptcb = os_tcb_free_list;                        // get a free TCB from the free TCB list

    if ( ptcb != NULL )
    {
        os_tcb_free_list        = ptcb->next_free;  // Update pointer to free TCB list
        ENABLE();
        ptcb->stack_ptr         = stk;              // Load Stack pointer in TCB
        ptcb->allocmem_ptr      = pmem;
        ptcb->prio              = prio;             // Load task priority into TCB
        ptcb->dly_time          = 0L;
        ptcb->pecb              = NULL;             // Task is not pending on an event
        DISABLE();
        *tsk_id                 = ptcb->task_id;
        os_tcb_task_tbl [ *tsk_id ]  = ptcb;

        os_tcb_ins_rdylist ( ptcb );
        ENABLE();

        return ( OS_NO_ERR );
    }
    else
    {
        ENABLE ();
        return ( OS_NO_MORE_TCB );
    }
}


/*
```

```
---------------------------------------------------------------------------
                            ENTER ISR
---------------------------------------------------------------------------
*/

void os_int_enter ( void )
{
    DISABLE();
    os_int_nesting++;                           // increment ISR nesting level
    ENABLE();
}


/*
---------------------------------------------------------------------------
                            EXIT ISR
---------------------------------------------------------------------------
*/

WORD os_int_exit ( void )
{
    DISABLE();

    if ( !( --os_int_nesting || os_lock_nesting ) )    // reschedule only if all ISRs completed & not locked
    {
        register int i;

        for ( i=0 ; i < OS_LO_PRIO; i++ )
        {
            if ( os_tcb_curprio [ i ] != NULL )
            {
                if ( i == os_tcb_current_prio )
                {
                    if ( os_tcb_cur->next_rdy != os_tcb_run )
                    {
                        os_tcb_run->task_stat    = OS_TASK_RDY;
                        os_tcb_run               = os_tcb_run->next_rdy;
                        os_tcb_run->task_stat    = OS_TASK_RUN;
                        os_ctx_sw_ctr++;
                        return -1;       // perform interrupt level context switch
                }
                }
                else
                {
                    os_tcb_run->task_stat    = OS_TASK_RDY;
                    os_tcb_run               = os_tcb_curprio [ i ];
                    os_tcb_current_prio      = i;
                    os_ctx_sw_ctr++;
                    return -1;  // perform interrupt level context switch
                }
            break;
            }
        }
    }
    ENABLE();
    return 0;   // no perform interrupt level context switch
}
```

```
/*
-------------------------------------------------------------------------------
                            DELETE A TASK
-------------------------------------------------------------------------------
*/

RETCODE os_task_del ( TASK_ID tsk_id )
{

    if ( tsk_id > OS_MAX_TASKS )
        return OS_TSK_NO_EXIST;

    if ( os_running && tsk_id == idle_task_id )
        return ( OS_TASK_DEL_IDLE );  // Not allowed to delete idle task

    ptcb    = os_tcb_task_tbl [ tsk_id ];
    if ( ptcb == NULL )                     // Task to delete must exist
        return OS_TSK_NO_EXIST;

    if ( ptcb->task_stat == OS_TASK_NULL )
        return OS_TSK_NO_EXIST;

    DISABLE ( );

    os_tcb_task_tbl   [ tsk_id ]  = NULL;

    switch ( ptcb->task_stat )
    {
        case OS_TASK_RUN:
        case OS_TASK_RDY:
            os_tcb_del_rdylist ( ptcb );
            break;

        case OS_TASK_DLY:
            os_tcb_del_dlylist ( ptcb );
            break;

        case OS_TASK_SEM:
        case OS_TASK_MBOX:
        case OS_TASK_Q:
            os_tcb_quit_evnlist ( ptcb );
            os_tcb_del_dlylist ( ptcb );
            break;
    }
    ptcb->next_free       = os_tcb_free_list;       //insert in tcb free list
    os_tcb_free_list  = ptcb;
    ptcb->task_stat       = OS_TASK_NULL;
    farfree ( ptcb->allocmem_ptr );               // free stack memory
    if ( ptcb == os_tcb_cur )// task to delete is the current active task?
        os_sched();                 // never returns
    ENABLE();
    return OS_NO_ERR;
}

/*
-------------------------------------------------------------------------------
                            END OF TASK
-------------------------------------------------------------------------------
*/
```

```
void os_task_end ( void )
{
    DISABLE ( );
    os_tcb_task_tbl   [ os_tcb_cur->task_id ]  = NULL;
    os_tcb_del_rdylist ( os_tcb_cur );
    os_tcb_cur->next_free                      = os_tcb_free_list;        //insert in tcb free list
    os_tcb_free_list                           = os_tcb_cur;
    os_tcb_cur->task_stat                      = OS_TASK_NULL;
    farfree ( os_tcb_cur->allocmem_ptr );
    os_sched();                                                           // never returns
}

/*
-------------------------------------------------------------------------------
                        CHANGE PRIORITY OF A TASK
-------------------------------------------------------------------------------
*/

RETCODE os_task_change_prio ( TASK_ID tsk_id, PRIO newprio )
{
    register OS_TCB      *ptcb = os_tcb_task_tbl [ tsk_id ];

    if ( tsk_id > OS_MAX_TASKS )
        return ( OS_TASK_DEL_ERR );

    if ( ptcb == NULL )                 // Task to delete must exist
        return OS_TSK_NO_EXIST;

    if ( ptcb->task_stat == OS_TASK_NULL )
        return OS_TSK_NO_EXIST;

    if ( newprio > OS_LO_PRIO - 1 || ptcb->prio == newprio )
        return OS_PRIO_ERR;
    DISABLE();

    switch ( ptcb->task_stat )
    {
        case OS_TASK_RUN:
            os_tcb_del_rdylist ( ptcb );
            ptcb->prio      = newprio;
            os_tcb_ins_rdylist ( ptcb );
            os_sched ( );
            break;

        case OS_TASK_RDY:
            os_tcb_del_rdylist ( ptcb );
            ptcb->prio      = newprio;
            os_tcb_ins_rdylist ( ptcb );
            ENABLE ( );
            break;

        case OS_TASK_DLY:
        case OS_TASK_SEM:
        case OS_TASK_MBOX:
        case OS_TASK_Q:
            ptcb->prio      = newprio;
            ENABLE ( );
            break;
```

```
    }
    return OS_NO_ERR;
}

/*
------------------------------------------------------------------------------
                    DELAY TASK 'n' TICKS   (n = 1 WORD)
------------------------------------------------------------------------------
*/

void os_tick_dly ( LONG ticks )
{
    if ( ticks )
    {
        DISABLE();

        os_tcb_cur->dly_time = ticks;             // Load ticks in TCB

        // delay current task
        os_tcb_del_rdylist ( os_tcb_cur );
        os_tcb_ins_dlylist ( os_tcb_cur );
        os_tcb_cur->task_stat    =   OS_TASK_DLY;
        os_sched();
    }
}

/*
------------------------------------------------------------------------------
                        DELAY TASK 'n' msecs
------------------------------------------------------------------------------
*/
// if msecs<tick_size => no delay

void os_msec_dly ( LONG msecs )
{
    os_tick_dly ( (LONG) msecs/tick_size );
}

/*
------------------------------------------------------------------------------
                        PROCESS SYSTEM TICK
------------------------------------------------------------------------------
*/

void os_time_tick ( void )
{
    DISABLE ( );
    if ( os_tcb_dlylist != NULL )
    {       // Delayed or waiting for event with TO
        register OS_TCB      *aux_ptcb = os_tcb_dlylist;

        --os_tcb_dlylist->dly_time;  // decrement nbr of ticks to end of delay
        while ( !os_tcb_dlylist->dly_time )
        {
            os_tcb_dlylist          = os_tcb_dlylist->next_dly;
            if ( aux_ptcb->pecb != NULL )
                os_tcb_quit_evnlist ( aux_ptcb );
            os_tcb_ins_rdylist ( aux_ptcb );
            aux_ptcb->timeout_flag = 1;
```

```
            aux_ptcb->task_stat       = OS_TASK_RDY;
            if ( os_tcb_dlylist == NULL )
                break;
            aux_ptcb                  = os_tcb_dlylist;
        }
    }
    os_time++;
    ENABLE();
}

/*
---------------------------------------------------------------------------
                        SET SYSTEM CLOCK
---------------------------------------------------------------------------
*/

void os_time_set ( LONG ticks )
{
    DISABLE();
    os_time = ticks;
    ENABLE();
}

TASK_ID os_get_cur_taskid ( void )
{
    return os_tcb_cur->task_id;
}

/*
---------------------------------------------------------------------------
                      GET CURRENT SYSTEM TIME
---------------------------------------------------------------------------
*/

LONG os_time_get ( void )
{
    TIME ticks;

    DISABLE();
    ticks = os_time;
    ENABLE();
    return ( ticks );
}

/*
---------------------------------------------------------------------------
                        SET TICK SIZE
---------------------------------------------------------------------------
*/

RETCODE os_set_ticksize ( WORD ticksize )     //ticksize in msecs.
{
    if ( ticksize > 54 || ticksize < 12 )
        return OS_BAD_TICKSIZE;

    DISABLE ( );

    tick_size   = ticksize;
    divisor     = (1193180L * ticksize/1000);
```

```
    // TICK SIZE SET
    // see "PROGRAMMERS PROBLEM SOLVER PC XT/AT" pags. 45-49
    // channel #0, write low-then-high byte, mode=3, binary data

    outportb ( BASE_ADRESS_TIMER + 3, WCONTROL );           // 8253 mode
    outportb ( BASE_ADRESS_TIMER + 0, divisor      & 0xff );    // 1ro. low
    outportb ( BASE_ADRESS_TIMER + 0, (divisor >> 8) & 0xff );  // 2do. high
    ENABLE ( );
    return OS_NO_ERR;
}

/*
---------------------------------------------------------------------------
                        GET TICK SIZE
---------------------------------------------------------------------------
*/

WORD os_get_ticksize ( void )
{
    return tick_size;
}

/*
---------------------------------------------------------------------------
                        INITIALIZE SEMAPHORE
---------------------------------------------------------------------------
*/

OS_ECB *os_sem_create ( SWORD cnt )
{
    register OS_ECB *pecb;

    if ( cnt < 0 )   // semaphore cannot start negative
        return NULL;

    DISABLE();
    pecb = os_ecb_free_list;                    // get next free event control block
    if ( os_ecb_free_list != NULL )             // See if pool of free ECB pool was empty
        os_ecb_free_list = (OS_ECB *)os_ecb_free_list->os_event_ptr;

    ENABLE();
    if ( pecb != NULL )
    {                                           // Get an event control block
        pecb->os_event_cnt  = cnt;              // Set semaphore value
        return ( pecb );
    }
    else
        return ( NULL );                        // Ran out of event control blocks
}

/*
---------------------------------------------------------------------------
                        PEND ON SEMAPHORE
---------------------------------------------------------------------------
*/

RETCODE os_sem_wait ( OS_ECB *pecb, WORD timeout )
{
```

```
    if ( pecb    == NULL)
        return OS_NULL_PECB;

    DISABLE();
    if ( --pecb->os_event_cnt < 0 )
    // must wait until event occurs
    {
        os_tcb_cur->pecb          = pecb;              // store pointer to event control block in TCB
        os_tcb_ins_evnlist ( os_tcb_cur );             // insert in event list
        os_tcb_del_rdylist ( os_tcb_cur );             // delete of rdy list
        os_tcb_cur->task_stat     = OS_TASK_SEM;       // resource not available, pend on semaphore
        os_tcb_cur->dly_time      = timeout;           // store pend timeout in TCB
        os_tcb_cur->timeout_flag= 0;
        if ( timeout )                                 // timeout?...
            os_tcb_ins_dlylist ( os_tcb_cur );         // insert in delay list
        os_sched ( );                                  // find next highest priority task ready to run
        DISABLE ( );
        os_tcb_cur->pecb          = NULL;
        ENABLE ( );
        if ( os_tcb_cur->timeout_flag )
            return OS_TIMEOUT;
        else
        {
            DISABLE ( );
            os_tcb_del_dlylist ( os_tcb_cur );
            ENABLE ( );
            return OS_NO_ERR;
        }
    }
    else
    {                                                  // semaphore > 0, resource available
        ENABLE();
        return  OS_NO_ERR;
    }
}

/*
---------------------------------------------------------------------------
                        DELETE A QUEUE
---------------------------------------------------------------------------
*/

RETCODE os_q_dest ( OS_ECB *pecb )
{
    OS_Q *os_q;

    if ( pecb == NULL )
        return OS_NULL_PECB;

    if ( ( os_q = (OS_Q*)pecb->os_event_ptr ) == NULL )
        return OS_Q_NULL;

    os_q->os_q_ptr            = os_q_free_list; //insert in free list
    os_q_free_list            = os_q;
    farfree(os_q->qstart);
    os_event_dest ( pecb );
    return OS_NO_ERR;
}
```

```
/*
--------------------------------------------------------------------------------
                        DELETE AN EVENT
--------------------------------------------------------------------------------
*/

RETCODE    os_event_dest ( OS_ECB *pecb )
{
    if ( pecb == NULL )
        return OS_NULL_PECB;

    DISABLE ( );
    os_tcb_purge_evnlist ( pecb );
    ENABLE ( );
    pecb->os_event_ptr  = os_ecb_free_list;
    os_ecb_free_list  = pecb;
    return OS_NO_ERR;
}

/*
--------------------------------------------------------------------------------
                        POST TO A SEMAPHORE
--------------------------------------------------------------------------------
*/

RETCODE os_sem_signal ( OS_ECB *pecb )
{
    if ( pecb    == NULL)
        return OS_NULL_PECB;

    if ( pecb->os_event_cnt < INT_MAX )     // make sure semaphore will not overflow
    {
        DISABLE();
        pecb->os_event_cnt++;
        if ( pecb->os_event_cnt <= 0 )
        {
            os_tcb_del_evnlist ( pecb );          //signal the waiting task
            os_sched ( );                         // find highest priority task ready to run
            return OS_NO_ERR;
    }
        else
            ENABLE();
        return OS_NO_ERR;
    }
    else
        return OS_SEM_OVF;
}

/*
--------------------------------------------------------------------------------
                        INITIALIZE MESSAGE MAILBOX
--------------------------------------------------------------------------------
*/

OS_ECB *os_mbox_create ( void *msg )
{
    OS_ECB *pecb;

    DISABLE();
```

```
    pecb = os_ecb_free_list;                    // Get next free event control block
    if ( os_ecb_free_list != NULL )             // See if pool of free ECB pool was empty
        os_ecb_free_list = (OS_ECB *)os_ecb_free_list->os_event_ptr;
    ENABLE();
    if ( pecb != NULL )
        pecb->os_event_ptr    = msg;            // Deposit message in event control block

    return ( pecb );                            // Return pointer to event control block
}

/*
---------------------------------------------------------------------------
                        PEND ON MAILBOX FOR A MESSAGE
---------------------------------------------------------------------------
*/

void    *os_mbox_receive ( OS_ECB *pecb, WORD timeout, RETCODE *err )
{
    void  *msg;

    if ( pecb    == NULL)
    {
        *err     =   OS_NULL_PECB;
        return NULL;
    }

    DISABLE();
    if ( (msg = pecb->os_event_ptr) != NULL )
    {                                           // See if there is already a message
        pecb->os_event_ptr = NULL;              // Clear the mailbox
        ENABLE();
        *err = OS_NO_ERR;
    }
    else
    {
        os_tcb_cur->pecb          = pecb;                   // Store pointer to event control block in
TCB
        os_tcb_ins_evnlist ( os_tcb_cur );
        os_tcb_del_rdylist ( os_tcb_cur );
        os_tcb_cur->task_stat         = OS_TASK_MBOX;   // Message not available, task will pend
        os_tcb_cur->dly_time          = timeout;        // Load timeout in TCB
        os_tcb_cur->timeout_flag    = 0;
        if ( timeout )
            os_tcb_ins_dlylist ( os_tcb_cur );

        os_sched();                                     // Find next highest priority task ready to
run
        DISABLE();
        msg                     = pecb->os_event_ptr;   // Message received
        pecb->os_event_ptr  = NULL;                     // Clear the mailbox
        ENABLE();
        if ( os_tcb_cur->timeout_flag )
            *err     = OS_TIMEOUT;
        else
        {
            DISABLE ( );
            os_tcb_del_dlylist ( os_tcb_cur );
            ENABLE ( );
            *err     = OS_NO_ERR;
```

```
        }
    }
    return msg;                                     // Return the message received (or NULL)
}

/*
------------------------------------------------------------------------------
                        POST MESSAGE TO A MAILBOX
------------------------------------------------------------------------------
*/

RETCODE os_mbox_send ( OS_ECB *pecb, void *msg )
{
    if ( pecb   == NULL)
        return  OS_NULL_PECB;

    DISABLE();
    if ( pecb->os_event_ptr != NULL )
    {                                               // Make sure mailbox doesn't already contain a msg
        ENABLE();
        return ( OS_MBOX_FULL );
    }
    else
    {
        pecb->os_event_ptr = msg;                   // Place message in mailbox
        os_tcb_purge_evnlist ( pecb );
        os_sched ( );                               // Find highest priority task ready to run
        return OS_NO_ERR;
    }
}

/*
------------------------------------------------------------------------------
                        INITIALIZE MESSAGE QUEUE
------------------------------------------------------------------------------
*/

OS_ECB *os_q_create ( WORD qsize, BYTE elem_size )
{
    OS_ECB      *pecb;
    OS_Q        *pq;
    void far    *start;

    if ( (start=farmalloc( (qsize+1)*elem_size ))==NULL)
        return NULL;

    DISABLE();
    pecb = os_ecb_free_list;                        // Get next free event control block
    if ( os_ecb_free_list != NULL )                 // See if pool of free ECB pool was empty
    os_ecb_free_list = (OS_ECB *)os_ecb_free_list->os_event_ptr;
    ENABLE();
    if ( pecb != NULL )
    {                                               // See if we have an event control block
        DISABLE();                                  // Get a free queue control block
        pq = os_q_free_list;
        if ( os_q_free_list != NULL )
            os_q_free_list = os_q_free_list->os_q_ptr;
        ENABLE();
        if ( pq != NULL )                           // See if we were able to get a queue control block
```

```
        {
            pecb->os_event_ptr  = pq;
            pq->qstart          = start;                    // Yes, initialize the queue
            pq->tail_offset     = pq->head_offset = pq->entries = 0;
            pq->qsize           = qsize+1;
            pq->elem_size           = elem_size;
        }
        else
        {                                   // No,  since we couldn't get a queue control block
            DISABLE();                      // Return event control block on error
            pecb->os_event_ptr = (void *)os_ecb_free_list;
            os_ecb_free_list   = pecb;
            ENABLE();
            pecb = NULL;
        }
    }
    return ( pecb );
}

/*
-------------------------------------------------------------------------------
                        PEND ON A QUEUE FOR A MESSAGE
-------------------------------------------------------------------------------
*/

void *os_q_read ( OS_ECB *pecb, WORD timeout, RETCODE *err )
{
    void  *msg;
    OS_Q  *pq = pecb->os_event_ptr;       // Point at queue control block

    DISABLE();
    if ( pq->entries )                          // see if any messages in the queue
    {
        pq->entries--;                          // update the number of entries in the queue
        msg =  (BYTE*) (pq->qstart) + pq->head_offset*pq->elem_size;    // yes, extract oldest message
                                                                        //from the queue
        pq->head_offset = ++pq->head_offset % pq->qsize;
        ENABLE();
        *err = OS_NO_ERR;
    }
    else
    {
        os_tcb_cur->pecb          = pecb;               // store pointer to event control block in TCB
        os_tcb_ins_evnlist ( os_tcb_cur );
        os_tcb_del_rdylist ( os_tcb_cur );
        os_tcb_cur->task_stat     = OS_TASK_Q;          // task will have to pend for a message to be
posted
        os_tcb_cur->dly_time      = timeout;            // load timeout into TCB
        os_tcb_cur->timeout_flag= 0;
        if ( timeout )                                  // timeout?...
            os_tcb_ins_dlylist ( os_tcb_cur );          // insert in delay list
        os_sched();                                     // Find next highest priority task ready to run
        DISABLE();
        pq->entries--;                                  // update the number of entries in the queue
        msg =  (BYTE*) (pq->qstart) + pq->head_offset*pq->elem_size;// message received, extract
oldest message from the queue
        pq->head_offset = ++pq->head_offset % pq->qsize;
        ENABLE();
        if ( os_tcb_cur->timeout_flag )
```

```
            *err     = OS_TIMEOUT;
        else
        {
            DISABLE ( );
            os_tcb_del_dlylist ( os_tcb_cur );
            ENABLE ( );
            *err     = OS_NO_ERR;
        }
    }                                                   // Return message received (or NULL)
    return (msg);
}

/*
---------------------------------------------------------------------------
                        POST MESSAGE TO A QUEUE
---------------------------------------------------------------------------
*/

BYTE os_q_write ( OS_ECB *pecb, void *msg )
{
    OS_Q  *pq = pecb->os_event_ptr;       // point to queue control block

    DISABLE();
    if ( pq->entries < pq->qsize )        // make sure that queue is not full
    {
        // insert message into queue
        _fmemcpy ( (BYTE *)pq->qstart + pq->elem_size*pq->tail_offset, msg, pq->elem_size );
        pq->tail_offset     = ++pq->tail_offset % pq->qsize;
        pq->entries++;                    // update the number of entries in the queue
        os_tcb_del_evnlist ( pecb );
        os_sched();                       // find highest priority task ready to run
        return OS_NO_ERR;
    }
    else
    {
        ENABLE ( );
        return ( OS_Q_FULL );
    }
}

/*
---------------------------------------------------------------------------
                    INSERT A TASK IN READY LIST
---------------------------------------------------------------------------
*/

void os_tcb_ins_rdylist ( OS_TCB *ptcb )
{
    WORD     prio     =    ptcb->prio;

    ptcb->task_stat    = OS_TASK_RDY;

    if ( os_tcb_run == NULL )
    {
        os_tcb_run = ptcb->next_rdy    = ptcb->prev_rdy = ptcb;
        os_tcb_current_prio            =    prio;
        os_tcb_curprio [ prio ]        = os_tcb_run;
    }
    else
```

```
        if ( os_tcb_curprio [ prio ] == NULL )
            os_tcb_curprio [ prio ] = ptcb->next_rdy = ptcb->prev_rdy = ptcb;
        else
        {
            os_tcb_curprio [ prio ]->prev_rdy->next_rdy  = ptcb;
            ptcb->prev_rdy                               = os_tcb_curprio [ prio ]->prev_rdy;
            os_tcb_curprio [ prio ]->prev_rdy            = ptcb;
            ptcb->next_rdy                               = os_tcb_curprio [ prio ];
        }
}

/*
--------------------------------------------------------------------------------
                    DELETE A TASK FROM READY LIST
--------------------------------------------------------------------------------
*/

void os_tcb_del_rdylist ( OS_TCB *ptcb )
{
    WORD    prio    =   ptcb->prio;

    if ( ptcb == os_tcb_run )
    {
        if ( os_tcb_cur->next_rdy != os_tcb_run )
        {   //change os_tcb_run ( round robin )
            os_tcb_run                  = os_tcb_run->next_rdy;
            os_tcb_run->task_stat       = OS_TASK_RUN;
            os_tcb_curprio [ prio ]     = os_tcb_run;
            ptcb->prev_rdy->next_rdy    = ptcb->next_rdy;
            ptcb->next_rdy->prev_rdy    = ptcb->prev_rdy;
        }
        else    //was the only task in this priority
        {
            int i;

            os_tcb_curprio [ prio ]   = NULL;

            //search next highest priority task ready tu run
            for ( i=0 ; i < OS_LO_PRIO; i++ )
                if ( os_tcb_curprio [ i ] != NULL )
                {
                    os_tcb_run              = os_tcb_curprio [ i ];
                    os_tcb_current_prio     = i;
                    os_tcb_run->task_stat   = OS_TASK_RUN;
                    break;
                }
        }
    }
    else
    {
        if ( ptcb->next_rdy == ptcb )//was the only task in this priority?
            os_tcb_curprio [ prio ]   = NULL;
        else
        {
            ptcb->prev_rdy->next_rdy    = ptcb->next_rdy;
            ptcb->next_rdy->prev_rdy    = ptcb->prev_rdy;
        }
    }
}
```

```
/*
-------------------------------------------------------------------------------
                    INSERT A TASK IN AN EVENT LIST
-------------------------------------------------------------------------------
*/

void os_tcb_ins_evnlist ( OS_TCB *ptcb )
{
    OS_ECB  *pecb =   ptcb->pecb;

    if ( pecb->os_tcb_blk_task == NULL ||
                    ptcb->prio < pecb->os_tcb_blk_task->prio )
    {
        ptcb->next_blk              = pecb->os_tcb_blk_task;
        pecb->os_tcb_blk_task   = ptcb;
    }
    else
    {
        OS_TCB  *p;

        for ( p = pecb->os_tcb_blk_task; p != NULL; p=p->next_blk )
            if ( ptcb->prio < p->prio || p->next_blk == NULL )
            {
                ptcb->next_blk          = p->next_blk;
                p->next_blk             = ptcb;
                break;
            }
    }
}

/*
-------------------------------------------------------------------------------
                    PURGE AN EVENT LIST
-------------------------------------------------------------------------------
*/

void os_tcb_purge_evnlist ( OS_ECB *pecb )
{
    OS_TCB  *ptcb   = pecb->os_tcb_blk_task;

    for (; ptcb != NULL; ptcb=ptcb->next_blk )
    {
        ptcb->pecb    = NULL;
        os_tcb_ins_rdylist ( ptcb );
    }

    pecb->os_tcb_blk_task   = NULL;
}

/*
-------------------------------------------------------------------------------
                    DELETE A TASK FROM AN EVENT LIST
-------------------------------------------------------------------------------
*/

void os_tcb_del_evnlist ( OS_ECB *pecb )
{
    if ( pecb->os_tcb_blk_task != NULL )
```

```
    {
        os_tcb_ins_rdylist ( pecb->os_tcb_blk_task );
        pecb->os_tcb_blk_task       = pecb->os_tcb_blk_task->next_blk;
    }
}

/*
------------------------------------------------------------------------------
                        INSERT A TASK IN DELAY LIST
------------------------------------------------------------------------------
*/

void os_tcb_ins_dlylist ( OS_TCB *ptcb )
{
    if ( os_tcb_dlylist == NULL )
    {
        ptcb->next_dly              = NULL;
        os_tcb_dlylist              = ptcb;
    }
    else if ( os_tcb_dlylist->dly_time > ptcb->dly_time )
    {
        ptcb->next_dly              = os_tcb_dlylist;
        os_tcb_dlylist->dly_time    = ptcb->dly_time;
        os_tcb_dlylist              = ptcb;
    }
    else
    {
        OS_TCB  *p      = os_tcb_dlylist;
        TIME  time      = 0;

        for ( ; p != NULL; p=p->next_dly )
        {
            time    += p->dly_time;
            if ( p->next_dly == NULL )
            {
                ptcb->dly_time   = time;
                p->next_dly      = ptcb;
                ptcb->next_dly   = NULL;
                break;
            }
            else if ( ( time + p->next_dly->dly_time ) > ptcb->dly_time )
            {
                ptcb->dly_time          = time;
                p->next_dly->dly_time   = ptcb->dly_time;
                ptcb->next_dly          = p->next_dly;
                p->next_dly             = ptcb;
                break;
            }
        }
     }
}

/*
------------------------------------------------------------------------------
                        DELETE A TASK FROM DELAY LIST
------------------------------------------------------------------------------
*/

void os_tcb_del_dlylist ( OS_TCB *ptcb )
```

```
{
    OS_TCB  *aux_ptcb = os_tcb_dlylist;

    if ( aux_ptcb == ptcb )
    {
         os_tcb_dlylist                    = aux_ptcb->next_dly;
        aux_ptcb->next_dly->dly_time += aux_ptcb->dly_time;
    }
    else
        for ( ; aux_ptcb != NULL; aux_ptcb=aux_ptcb->next_dly )
            if ( aux_ptcb->next_dly == ptcb )
            {
                aux_ptcb->next_dly          = ptcb->next_dly;
                if ( ptcb->next_dly != NULL )
                    ptcb->next_dly->dly_time   += ptcb->dly_time;
                ptcb->next_dly              = NULL;
                break;
            }
}

/*
---------------------------------------------------------------------------
                    DELETE A TASK FROM EVENT LIST
---------------------------------------------------------------------------
*/

void os_tcb_quit_evnlist ( OS_TCB *ptcb )
{
    OS_ECB  *pecb      = ptcb->pecb;
    OS_TCB  *auxptcb   = pecb->os_tcb_blk_task;

    if ( auxptcb == ptcb )
    {
        pecb->os_tcb_blk_task  =   ptcb->next_blk;
        ptcb->next_blk         =   NULL;
        ptcb->pecb             =   NULL;
        return;
    }

    for (;  auxptcb != NULL; auxptcb = auxptcb->next_blk )
        if ( auxptcb->next_blk == ptcb )
        {
            auxptcb->next_blk      = ptcb->next_blk;
            ptcb->next_blk         = NULL;
            ptcb->pecb             = NULL;
            break;
        }
}

//end of file
```

11.5. i186l_c.c

```
#include   "80186l.h"
#include   "rtkernel.h"
#include   <dos.h>
#include   <alloc.h>

/*
---------------------------------------------------------------------------
                          CREATE A TASK
---------------------------------------------------------------------------
*/

RETCODE os_task_create ( void (far *task)(void *pd), void *pdata, WORD stksize, PRIO p,
TASK_ID *tsk_id )
{
    WORD          *stk;
    RETCODE       err;
    void far      *pmem;

    if ( (pmem=farmalloc(stksize))==NULL)       // alloc memory for stack task
        return OS_NO_MEMORY;

    stk      = (WORD*)((BYTE *)pmem+stksize);
    *--stk   = (WORD)FP_SEG(pdata);                    // simulate call to function with argument
    *--stk   = (WORD)FP_OFF(pdata);
    *--stk   = (WORD)FP_SEG(os_task_end);              // at end of task execute os_task_end()
    *--stk   = (WORD)FP_OFF(os_task_end);
    *--stk   = (WORD)0x0200;                           // PSW = Interrupts enabled
    *--stk   = (WORD)FP_SEG(task);                     // put pointer to task   on top of stack
    *--stk   = (WORD)FP_OFF(task);
    *--stk   = (WORD)0x0000;                           // AX = 0
    *--stk   = (WORD)0x0000;                           // CX = 0
    *--stk   = (WORD)0x0000;                           // DX = 0
    *--stk   = (WORD)0x0000;                           // BX = 0
    *--stk   = (WORD)0x0000;                           // SP = 0
    *--stk   = (WORD)0x0000;                           // BP = 0
    *--stk   = (WORD)0x0000;                           // SI = 0
    *--stk   = (WORD)0x0000;                           // DI = 0
    *--stk   = (WORD)0x0000;                           // ES = 0
    *--stk   = _DS;                                    // Save current value of DS
    err      = os_tcb_init ( p, (void far *)stk, pmem, tsk_id ); // Get and initialize a TCB
    if ( err == OS_NO_ERR )
        if ( os_running )
            os_sched ();// Find highest prio. task if multitasking has started
    return ( err );
}
```

11.6. i186_a.asm

```
;--------------------------------------------------------------------------
;                                rtKERNEL 1.00
;          Microcomputer Real-Time Multitasking Kernel
;
;                        80186/80188 Specific code
;                     LARGE MEMORY MODEL
;
;--------------------------------------------------------------------------

PUBLIC _os_start_run
PUBLIC _os_ctx_sw
PUBLIC _new_tick_isr

EXTRN  _os_tcb_cur:dword
EXTRN  _os_tcb_run:dword

EXTRN _count_ticks:dword
EXTRN _divisor:word
EXTRN _os_running:word

EXTRN  _os_int_enter:FAR
EXTRN  _os_int_exit:FAR
EXTRN  _os_time_tick:FAR

.MODEL large
.386
.CODE

;-----------------------------------------------------------------------------------------
;                                  START MULTITASKING
;                               void os_start_run(void)
;
;-----------------------------------------------------------------------------------------

_os_start_run      proc far

    mov    ax, DGROUP                      ; reload DS
    mov    ds, ax

    mov    ax, word ptr ds:_os_tcb_run+2   ; os_tcb_cur = os_tcb_run
    mov    dx, word ptr ds:_os_tcb_run
    mov    word ptr ds:_os_tcb_cur+2, ax
    mov    word ptr ds:_os_tcb_cur,     dx
    les    bx, dword ptr ds:_os_tcb_run    ; SS:SP = os_tcb_cur->stack_ptr
    mov    sp, es:[bx]
    mov    ss, es:[bx+2]

    pop    ds                              ; pop task's stack
    pop    es
    popa

    iret                                   ; run task

_os_start_run      endp
```

```
;---------------------------------------------------------------------------------------------
;                          PERFORM A CONTEXT SWITCH ( from task level )
;                                  void os_ctx_sw(void)
;
;---------------------------------------------------------------------------------------------

_os_ctx_sw    proc   far

    pusha                                   ; save current task's context
    push   es
    push   ds

    mov    ax, DGROUP                       ; reload ds with DGROUP
    mov    ds, ax

    les    bx, dword ptr ds:_os_tcb_cur     ; os_tcb_cur->stk_ptr = ss:sp
    mov    es:[bx], sp
    mov    es:[bx+2], ss

    mov    ax, word ptr ds:_os_tcb_run+2    ; os_tcb_cur = os_tcb_run
    mov    dx, word ptr ds:_os_tcb_run
    mov    word ptr ds:_os_tcb_cur+2, ax
    mov    word ptr ds:_os_tcb_cur,     dx

    les    bx, dword ptr ds:_os_tcb_run     ; ss:sp = os_tcb_run->stk_ptr;
    mov    sp, es:[bx]
    mov    ss, es:[bx+2]
    pop    ds                               ; restore context of task to run
    pop    es
    popa

    iret                                    ; return to new task

_os_ctx_sw    endp


;-------------------------------------------------------------------------------
;
;                                HANDLE TICK ISR
;
;-------------------------------------------------------------------------------
_new_tick_isr PROC   FAR

    sti                                     ; allow interrupt nesting

    pusha                                   ; save interrupted task's context
    push   es
    push   ds

    mov    ax, DGROUP                       ; reload ds with DGROUP
    mov    ds, ax

    mov    bx, _divisor
    and    ebx, 0ffffh
    add    _count_ticks, ebx
    cmp    _count_ticks, 0ffffh
    jbe no_chain
    and    _count_ticks, 0ffffh
```

```
    int 081h                        ; chain into DOS;s tick ISR

no_chain:

    mov   al, 020h                  ; end of interrupt to 8253
    out 020h, al

    mov   ax, _os_running           ; os running?
    jz  no_ctx_sw

    call    _os_int_enter           ; notify rtKERNEL about ISR
    call    _os_time_tick           ; handle system tick
    call    _os_int_exit            ; exit rtKERNEL through scheduler if HPT ready
    or      ax, ax
    jnz     ctx_sw                  ; context switch

no_ctx_sw:                          ; no context switch

    pop     ds                      ; restore interrupted task's context
    pop     es
    popa
    iret                            ; return to interrupted task

;   PERFORM A CONTEXT SWITCH ( From an ISR )

ctx_sw:
;   mov   ax, DGROUP                ; reload ds with DGROUP
;   mov   ds, ax

    les     bx, dword ptr ds:_os_tcb_cur    ;os_tcb_cur->stack_ptr = ss:sp
    mov   es:[bx], sp
    mov   es:[bx+2], ss

    mov   ax, word ptr ds:_os_tcb_run+2 ; os_tcb_cur = os_tcb_run
    mov   dx, word ptr ds:_os_tcb_run
    mov   word ptr ds:_os_tcb_cur+2, ax
    mov   word ptr ds:_os_tcb_cur, dx

    les bx, dword ptr ds:_os_tcb_run    ; ss:sp = os_tcb_run->stack_ptr;
    mov   sp, es:[bx]
    mov   ss, es:[bx+2]

    pop     ds                      ; restore new task's context
    pop     es
    popa

    iret                            ; return to new task

_new_tick_isr ENDP

end
```

11.7. Fuentes del programa de demostración

11.7.1. util.h

```
/*
---------------------------------------------------------------------------
                    rtKERNEL 1.00 DEMO
            PC Real-Time Multitasking  KERNEL


 File :  UTIL.H

 Author:    Guillermo Pablo Tomasini

 Date:      30-8-94

---------------------------------------------------------------------------
*/
#ifndef     util_h
#define util_h

#include "rtkernel.h"

void    disp_char (BYTE x, BYTE y, char  c);
void    disp_str (BYTE x,  BYTE y, char *s);
void    scroll_down( BYTE lines, BYTE x1, BYTE y1, BYTE x2, BYTE y2, BYTE color );
void    scroll_up ( BYTE lines, BYTE x1, BYTE y1, BYTE x2, BYTE y2, BYTE color );
RETCODE    init_screen ( char *fname );
void    init_uart (long baud_rate, BYTE num_data_bits, BYTE num_stop_bits,\
                    BYTE parity, unsigned base_address, BYTE irq_vect );

#define    COM1_BASE_ADR     0x3f8
#define    COM2_BASE_ADR     0x2f8

#define    IRQ4_VECT         0x0c
#define    IRQ3_VECT         0x0b

typedef enum { PAR=0x00, IMPAR=0x01, NONE=0x02 };

// offsets de los registros de UART respecto de direccion base

typedef enum { RBR=0x00, THR=0x00, IER=0x01, MSB=0x01, LSB=0x00,
                    IIR=0x02, LCR=0x03, MCR=0x04, LSR=0x05, MSR=0x06  };

#endif
```

11.7.2. util.c

```
/*
---------------------------------------------------------------------------
                rtKERNEL 1.00 UTILS FOR DEMO
            PC Real-Time Multitasking
                        KERNEL


 File :  UTIL.C

 Author:   Guillermo Pablo Tomasini

 Date:     20-8-94

---------------------------------------------------------------------------
*/
#include   <conio.h>
#include   <stdio.h>
#include   <dos.h>

#include   "80186l.h"
#include   "rtkernel.h"
#include   "util.h"


/*
*****************************************************************************************
                              CONSTANTS
*****************************************************************************************
*/

#define VIDEO                 0x10

#define RESET_DLAB(base_address) outportb(base_address+LCR, inportb(base_address+LCR) &
~0x80)
#define SET_DLAB(base_addres)      outportb(base_address+LCR, inportb(base_address+LCR) &
~0x80)

#define    BASE_ADDRESS_8259    0x20
#define    END_OF_INTERRUPT     0x20
#define    CONTROL_WORD1        0x01
#define    CONTROL_WORD2        0x00

#define LINE_BUFFER_LEN       100

/*
*****************************************************************************************
                            FUNCTION PROTOTYPES
*****************************************************************************************
*/

void goto_xy ( BYTE x, BYTE y, BYTE page );
void write_char ( BYTE character, WORD nbr, BYTE page, BYTE color );
static void interrupt my_uart_handler(void);

/*
*****************************************************************************************
                              VARIABLES
```

```
*********************************************************************************************
*/

extern OS_ECB       *dos_sem;             // Pointer to display  semaphore
extern OS_ECB       *rx_qp1;
extern OS_ECB       *rx_qp2;

/*
*********************************************************************************************
                        DISPLAY CHARACTER FUNCTION
*********************************************************************************************
*/

void disp_char ( BYTE x, BYTE y, char c )
{
    os_sem_wait ( dos_sem, 0 );
    goto_xy ( x , y, 0 );
    putchar ( c );
    os_sem_signal ( dos_sem );
}

/*
*********************************************************************************************
                          DISPLAY STRING FUNCTION
*********************************************************************************************
*/

void disp_str( BYTE x, BYTE y, char *s )
{
    os_sem_wait ( dos_sem, 0 );
    goto_xy ( x , y, 0 );
    puts ( s );
    os_sem_signal ( dos_sem );
}

/*
*********************************************************************************************
                          INIT UART FUNCTION
*********************************************************************************************
*/

void init_uart ( long baud_rate, BYTE num_data_bits, BYTE num_stop_bits,
                 BYTE parity, unsigned base_address, BYTE irq_vect )
{
    unsigned int divisor  =   115200/baud_rate;
    BYTE IIR_var;

    RESET_DLAB ( base_address );   //DLAB=0

    if ((inportb(base_address + IER) & 0xf0) != 0x00)
        fprintf(stderr, "No esta presente COM en I/O port %04x\n", base_address);

    // deshabilita las interrupciones

    RESET_DLAB(base_address);

    outportb(base_address+IER, 0x00);
    // limpia las interrupciones
    inportb(base_address+IIR);
```

```
inportb(base_address+RBR);
inportb(base_address+LSR);
inportb(base_address+MSR);

// BAUD RATE ----------------------------------------------------

SET_DLAB(base_address);           // DLAB = 1
outportb(base_address + LSB, (BYTE) divisor);
outportb(base_address + MSB, (BYTE)((divisor & 0xff00)>>8));

// LINE CONTROL (8, par, etc... ) -------------------------------------

// cantidad de BITS
// los ultimos dos bits en cero

RESET_DLAB(base_address);
outportb(base_address+LCR, inportb(base_address+LCR)&~0x03);

switch(num_data_bits)
{
    case 8:
        outportb(base_address+LCR, inportb(base_address+LCR)|0x03);
        break;

    case 7:
        outportb(base_address+LCR, inportb(base_address+LCR)|0x02);
        break;

    case 6:
        outportb(base_address+LCR, inportb(base_address+LCR)|0x01);
        break;

    case 5:
        outportb(base_address+LCR, inportb(base_address+LCR)|0x00);
        break;

    default:
        fprintf(stderr, "ERROR EN num_data_bits, adopto 8\n");
        outportb(base_address+LCR, inportb(base_address+LCR)|0x03);
    }

// STOP BITS
switch(num_stop_bits)
{
    case 1:
        outportb(base_address+LCR, inportb(base_address+LCR)&~0x04);
        break;

    case 2:
        outportb(base_address+LCR, inportb(base_address+LCR)|0x04);
        break;

    default:
        fprintf(stderr, "ERROR EN num_stop_bits, adopto 1\n");
        outportb(base_address+LCR, inportb(base_address+LCR)&~0x04);
        break;
}

switch(parity)
```

```
{
    case NONE:
        outportb(base_address+LCR, inportb(base_address+LCR)&~0x08);
        break;

    case PAR:
        outportb(base_address+LCR, inportb(base_address+LCR)|0x18);
        break;

    case IMPAR:
        outportb(base_address+LCR, inportb(base_address+LCR)&~0x10);
        outportb(base_address+LCR, inportb(base_address+LCR)|0x08);
        break;

    default:
        fprintf(stderr, "ERROR EN parity, adopto paridad PAR\n");
        outportb(base_address+LCR, inportb(base_address+LCR)|0x18);
    }

// SET BREAK DISABLED y STICK PARITY DISABLED
outportb(base_address+LCR, inportb(base_address+LCR)&~0x60);

// FIN DE LINE CONTROL

// LIMPIA INTERRUPCIONES 8250
while ((IIR_var=inportb(base_address+IIR)) != 0x01)
{
    #ifdef DEBUG
    printf("IIR=%02x\n", IIR_var);
    #else
    (void) IIR_var;
    #endif
    inportb(base_address + LSR);
    inportb(base_address + RBR);
    inportb(base_address + MSR);
}

#ifdef DEBUG
printf("IIR=%02x\n", IIR_var);
#endif

//INTERRUPT ENABLE REGISTER
//enable tx & rx interrupt

RESET_DLAB(base_address);
outportb(base_address+IER, 0x01);

//Loop Back en off
outportb(base_address+MCR, inportb(base_address+MCR) & ~0x10);

outportb(base_address+MCR, inportb(base_address+MCR)|0x0c);

//enable 8253

disable();

switch ( irq_vect )
{
    case IRQ4_VECT:
```

```
            outportb ( BASE_ADDRESS_8259+CONTROL_WORD1,
            inportb ( BASE_ADDRESS_8259+CONTROL_WORD1 ) &  ~0x10);
            break;

        case IRQ3_VECT:
            outportb ( BASE_ADDRESS_8259+CONTROL_WORD1,
            inportb ( BASE_ADDRESS_8259+CONTROL_WORD1 ) &  ~0x08);
            break;

        default:
            fprintf(stderr, "ERROR IN 8253 interrupt enbable.\n");
    }

    setvect ( irq_vect, my_uart_handler );
}


/*
*********************************************************************************************
                    MY_UART_HANDLER FUNCTION
*********************************************************************************************
*/

void interrupt my_uart_handler ( void )
{
    BYTE rx_data;
    BYTE status;

//  os_int_enter();
    status  = inportb ( COM1_BASE_ADR+LSR );

    if ( (status & 0x01) && !(status & 0x0c) )
    {
        rx_data     = inportb ( COM1_BASE_ADR+RBR );       //lee RBR
        os_q_write ( rx_qp1, &rx_data );

        //se recibi¢ algo no hay parity error ni framming error
        //lo pongo en la cola de recepci¢n y contesto
    }

    status=inportb ( COM2_BASE_ADR+LSR );

    if ( (status & 0x01) && !(status & 0x0c) )
    {
        rx_data     = inportb ( COM2_BASE_ADR+RBR );       //lee RBR

        os_q_write ( rx_qp2, &rx_data );
        //se recibi¢ algo no hay parity error ni framming error
        //lo pongo en la cola de recepci¢n y contesto
    }
//  os_int_exit();
    outportb ( CONTROL_WORD2+BASE_ADDRESS_8259, END_OF_INTERRUPT );
    enable();
}


/*
*********************************************************************************************
                        GOTO_XY FUNCTION
```

```
*****************************************************************************************
*/

void goto_xy ( BYTE x, BYTE y, BYTE page )
{
    union REGS regs;

    regs.h.ah = 2;          // set cursor position
    regs.h.dh = y;
    regs.h.dl = x;
    regs.h.bh = page;      // video page
    int86 ( VIDEO, &regs, &regs );
}


/*
*****************************************************************************************
                        SCROLL_UP FUNCTION
*****************************************************************************************
*/

void scroll_up ( BYTE lines, BYTE x1, BYTE y1, BYTE x2, BYTE y2, BYTE color )
{
    union REGS regs;

    regs.h.ah = 6;
    regs.h.al = lines;
    regs.h.ch = x1;
    regs.h.cl = y1;
    regs.h.dh = x2;
    regs.h.dl = y2;
    regs.h.bh = color;
    os_sem_wait ( dos_sem, 0 );
    int86(VIDEO, &regs, &regs);
    os_sem_signal ( dos_sem );
}


/*
*****************************************************************************************
                        SCROLL_DOWN FUNCTION
*****************************************************************************************
*/

void scroll_down ( BYTE lines, BYTE x1, BYTE y1, BYTE x2, BYTE y2, BYTE color )
{
    union REGS regs;

    regs.h.ah = 7;
    regs.h.al = lines;
    regs.h.ch = x1;
    regs.h.cl = y1;
    regs.h.dh = y2;
    regs.h.dl = x2;
    regs.h.bh = color;  /* video page 0 */
    os_sem_wait ( dos_sem, 0 );
    int86(VIDEO, &regs, &regs);
    os_sem_signal ( dos_sem );
}
```

```
/*
*****************************************************************************************
                        WRITE_CHAR FUNCTION
*****************************************************************************************
*/

void write_char ( BYTE character, WORD nbr, BYTE page, BYTE color )
{
    union REGS regs;

    regs.h.ah = 0x0a; // write char
    regs.h.al = character;
    regs.x.cx = nbr;
    regs.h.bl = color;
    regs.h.bh = page; // video page 0
    os_sem_wait ( dos_sem, 0 );
    int86 ( VIDEO, &regs, &regs );
    os_sem_signal ( dos_sem );
}


/*
*****************************************************************************************
                          INIT_SCREEN FUNCTION
*****************************************************************************************
*/

RETCODE init_screen ( char *fname )
{
    FILE   *fp;
    char    buffer [ LINE_BUFFER_LEN ];
    int     line=0;

    if ( ( fp=fopen( fname, "r" ) ) == NULL )
        return -1;

    clrscr();
    _setcursortype(_NOCURSOR);//avoid the cursor

    while ( fgets ( buffer, LINE_BUFFER_LEN, fp ) )
    {
        buffer[80] = '\0';
        if ( line++ )
            printf ( "%s", buffer );
    }
    scroll_down ( 1, 0, 0, 79, 24, 0 );
    fseek ( fp, 0, SEEK_SET );
    fgets ( buffer, LINE_BUFFER_LEN, fp );
    buffer[80] = '\0';
    gotoxy ( 1, 1 );
    puts ( buffer );
    fclose ( fp );
    return 0;
}
```

11.7.3. ex3.c

```
/*
---------------------------------------------------------------------------
                    rtKERNEL 1.00 DEMO
                PC Real-Time Multitasking
                        KERNEL


 File :  EX3.C

 Author:   Guillermo Pablo Tomasini

 Date:     30-8-94

---------------------------------------------------------------------------
*/

#include   <stdio.h>
#include   <string.h>
#include   <stdlib.h>
#include   <conio.h>
#include   <dos.h>

#include   "80186l.h"
#include   "rtkernel.h"
#include   "util.h"
/*
*****************************************************************************************
                              CONSTANTS
*****************************************************************************************
*/

#define          TASK_STK_SIZE   1024          // Size of each task's stacks (# of bytes)
#define          RX_Q_SIZE       255

#define          F1              0x3b
#define          F2              0x3c
#define          F3              0x3d
#define          F4              0x3e

#define          F10             0x44

#define          PAGE_UP         0x49
#define          PAGE_DOWN       0x51

/*
*****************************************************************************************
                               VARIABLES
*****************************************************************************************
*/


OS_ECB      *rx_qp1;             // Pointer to rx queue
OS_ECB      *rx_qp2;             // Pointer to rx queue
OS_ECB      *tx_qp;              // pointer to tx queue

OS_ECB      *key_mbox_ptr1;      // Pointer to keyboard mailbox
OS_ECB      *key_mbox_ptr2;      // Pointer to keyboard mailbox
```

```
OS_ECB       *dos_sem;                  // Pointer to DOS  semaphore

char *message[]={    "press F1 to send message to task A   ",
                     "press F2 to send message to task B   ",
                     "press P_UP to increment system tick  ",
                     "press P_DOWN to decrement system tick",
                     "press F10 to exit                    ",
                     "press any other key to send by tx1   "
                };


/*
*********************************************************************************************
                          FUNCTION PROTOTYPES
*********************************************************************************************
*/

void far    key_task ( void *data );
void far    A_task ( void *data );
void far    B_task ( void *data );
void far    ran_task ( void *data );
void far    rx1_task ( void *data );
void far    rx2_task ( void *data );
void far    stat_task ( void *data );
void far    tx_task ( void *data );

/*
*********************************************************************************************
                               MAIN
*********************************************************************************************
*/

int main ( void )
{
    TASK_IDtask_id;

    init_screen ( "screen.txt" );
    init_uart ( 2400, 8, 1, NONE, COM1_BASE_ADR, IRQ4_VECT );
    init_uart ( 2400, 8, 1, NONE, COM2_BASE_ADR, IRQ3_VECT );
    os_init();
    key_mbox_ptr1   = os_mbox_create( NULL );          // Keyboard mailbox
    key_mbox_ptr2   = os_mbox_create( NULL );          // Keyboard mailbox
    dos_sem         = os_sem_create ( 1 );             // DOS  semaphore
    rx_qp1          = os_q_create ( RX_Q_SIZE, 1 );
    rx_qp2          = os_q_create ( RX_Q_SIZE, 1 );
    tx_qp           = os_q_create ( RX_Q_SIZE, 1 );
    os_task_create ( stat_task, NULL, TASK_STK_SIZE, 0, &task_id );
    os_start();
    os_q_dest ( rx_qp1 );
    os_q_dest ( rx_qp2 );
    os_q_dest ( tx_qp );
    clrscr();
    _setcursortype ( _NORMALCURSOR );
    return 0;
}

/*
*********************************************************************************************
```

```
                                KEY TASK
*********************************************************************************************
*/

void far key_task ( void *data )
{
    WORD    ctr;
    char    s[30];
    TASK_ID task_id;
    int     loop=0;
    BYTE    cnt_dly=0;

    data    = data;// Prevent compiler warning
    task_id = os_get_cur_taskid();

    ctr     = 0;

    sprintf ( s, "%d", task_id );
    disp_str ( 18, 2, s );

    while ( 1 )
    {
        int key, key2;

        sprintf ( s, "%05d", ctr );
        disp_str ( 21, 2, s );
        ctr++;

        if ( kbhit() )
        {
            key = getch();

            if ( !key )
                switch ( getch() ) //extended code
                {
                case F1:        //F1
                        os_mbox_send ( key_mbox_ptr1, (void *)1);
                        break;

                case F2:        //F2
                        os_mbox_send ( key_mbox_ptr2, (void *)2);
                        break;

                    case PAGE_UP:       //PAGE UP
                        os_set_ticksize ( os_get_ticksize()+1 );
                        break;

                    case PAGE_DOWN:     //PAGE DOWN
                        os_set_ticksize ( os_get_ticksize()-1 );
                        break;

                    case F10:
                        os_end();
                }
            else
                os_q_write ( tx_qp, &key );
        }
        if ( ++cnt_dly==30 )
        {
```

```
            cnt_dly     = 0;
            if ( ++loop == 6 )
                loop=0;
            disp_str ( 1, 3, message [ loop ] );
        }
    os_tick_dly ( 1 );
    }
}

/*
*********************************************************************************************
                                A TASK
*********************************************************************************************
*/

void far A_task(void *data)
{
    RETCODE    err;
    WORD       toctr;
    WORD       msgctr;
    char       s[30];
    TASK_ID    task_id;

    data    = data;
    task_id     = os_get_cur_taskid();
    sprintf ( s, "%d", task_id );
    disp_str ( 18, 6, s );
    msgctr = 0;
    toctr  = 0;
    while ( 1 )
    {
        sprintf ( s, "%05d", toctr );
        disp_str ( 34, 6, s );
        sprintf ( s, "%05d", msgctr );
        disp_str ( 34, 7, s );
        os_mbox_receive ( key_mbox_ptr1, 20, &err);
        switch ( err )
        {
        case OS_NO_ERR:
            msgctr++;
            break;

        case OS_TIMEOUT:
            toctr++;
            break;
        }
    }
}


/*
*********************************************************************************************
                                B TASK
*********************************************************************************************
*/

void far B_task(void *data)
{
    RETCODE    err;
```

```
    WORD    toctr;
    WORD    msgctr;
    char    s[30];
    TASK_ID task_id;

    data    = data;
    task_id     = os_get_cur_taskid();
    sprintf ( s, "%d", task_id );
    disp_str ( 18, 8, s );
    msgctr = 0;
    toctr  = 0;
    while ( 1 )
    {
        sprintf ( s, "%05d", toctr );
        disp_str ( 34, 8, s );
        sprintf ( s, "%05d", msgctr );
        disp_str ( 34, 9, s );
        os_mbox_receive ( key_mbox_ptr2, 20, &err);
        switch ( err )
        {
        case OS_NO_ERR:
            msgctr++;
            break;

        case OS_TIMEOUT:
            toctr++;
            break;
        }
    }
}

/*
*********************************************************************************************
                        RANDOM TASK
*********************************************************************************************
*/

void far ran_task ( void *data )
{
    BYTE    x, y, z;
    char        s[30];
    TASK_ID task_id;

    data        = data;
    task_id     = os_get_cur_taskid();

    sprintf ( s, "task id %d:", task_id );
    disp_str ( 65, 2, s);

    while (1)
    {
        os_tick_dly (1);
        x = random ( 37 );          // Find X position where task number will appear
        y = random ( 6 );           // Find Y position where task number will appear
        z = random ( 4 );
        switch ( z )
        {
            case 0:
                disp_char( x + 41, y + 4, '*' );
```

```
                break;

            case 1:
                disp_char( x + 41, y + 4, 'X' );
                break;

            case 2:
                disp_char( x + 41, y + 4, '#' );
                break;

            case 3:
                disp_char( x + 41, y + 4, ' ' );
                break;
        }
    }
}


/*
*********************************************************************************************
                            RX1 TASK
*********************************************************************************************
*/

#define XLW1    2
#define X_RW1  37
#define Y_TW1  20
#define Y_BW1  23

void far rx1_task ( void *data )
{
    BYTE status;
    BYTE x=X_LW1,y=Y_TW1;
    TASK_IDtask_id= os_get_cur_taskid();
    char    s[30];

    data = data;

    sprintf ( s, "%d:", task_id );
    disp_str ( 24, 18, s );

    while ( 1 )
    {
        RETCODE err;
        char    *rx_data;

        rx_data     = os_q_read ( rx_qp1, 0, &err );

        disp_char ( x , y, *rx_data );

        if ( ++x > X_RW1 )
        {
            x   = X_LW1;
            if ( ++y > Y_BW1 )
            {
                y   = Y_BW1;
                scroll_up ( 1, Y_TW1, X_LW1, Y_BW1, X_RW1, 0 );
            }
        }
```

```
    }
}

/*
*********************************************************************************************
                                RX2 TASK
*********************************************************************************************
*/

#define X_LW2  42
#define X_RW2  78
#define Y_TW2  20
#define Y_BW2  23

void far rx2_task ( void *data )
{
    BYTE      status;
    BYTE      x=X_LW2,y=Y_TW2;
    TASK_ID   task_id= os_get_cur_taskid();
    char      s[30];

    data = data;

    sprintf ( s, "%d:", task_id );
    disp_str ( 65, 18, s );

    while ( 1 )
    {
        RETCODE err;
        char    *rx_data;

        rx_data     = os_q_read ( rx_qp2, 0, &err );

        disp_char ( x , y, *rx_data );

        if ( ++x > X_RW2 )
        {
            x   = X_LW2;
            if ( ++y > Y_BW2 )
            {
                y   = Y_BW2;
                scroll_up ( 1, Y_TW2, X_LW2, Y_BW2, X_RW2, 0 );
            }
        }
    }
}

/*
*********************************************************************************************
                                TX TASK
*********************************************************************************************
*/

#define X_LW_TX  42
#define X_RW_TX  78
#define Y_TW_TX  14
#define Y_BW_TX  15

void far tx_task ( void *data )
```

```
{
    BYTE status;
    BYTE x=X_LW_TX,y=Y_TW_TX;
    TASK_ID task_id= os_get_cur_taskid();
    char    s[30];

    data = data;

    sprintf ( s, "%d:", task_id );
    disp_str ( 65, 12, s );

    while ( 1 )
    {
        RETCODE err;
        char    *tx_data;

        tx_data     = os_q_read ( tx_qp, 0, &err );
        outportb ( COM1_BASE_ADR+THR, *tx_data );

        disp_char ( x , y, *tx_data );

        if ( ++x > X_RW_TX )
        {
            x   = X_LW_TX;
            if ( ++y > Y_BW_TX )
            {
                y   = Y_BW_TX;
                scroll_up ( 1, Y_TW_TX, X_LW_TX, Y_BW_TX, X_RW_TX, 0 );
            }
        }
    }
}

/*
*********************************************************************************************
                            STAT TASK
*********************************************************************************************
*/

void far stat_task ( void *data )
{
    double max;
    char    s[30];
    TASK_ID task_id;

    data    = data;

    os_tick_dly ( 10 );
    max     = (double)os_idle_ctr;
    os_task_create( key_task, NULL, TASK_STK_SIZE, 1, &task_id );
    os_task_create( A_task, NULL, TASK_STK_SIZE, 1, &task_id );
    os_task_create( B_task, NULL, TASK_STK_SIZE, 1, &task_id );
    os_task_create( ran_task, NULL, TASK_STK_SIZE, 1, &task_id );
    os_task_create( rx1_task, NULL, TASK_STK_SIZE, 1, &task_id );
    os_task_create( rx2_task, NULL, TASK_STK_SIZE, 1, &task_id );
    os_task_create( tx_task, NULL, TASK_STK_SIZE, 1, &task_id );

    task_id= os_get_cur_taskid();
```

```
    sprintf ( s, "%d:", task_id );
    disp_str ( 28, 12, s );

    while (1)
    {
        LONG  idle;
        WORD  ctxsw;
        char   s[80];
        double usage;
        struct dostime_t t;

        DISABLE();
        ctxsw           = os_ctx_sw_ctr;
        idle            = os_idle_ctr;
        os_ctx_sw_ctr   = 0;                    // reset statistics counters
        os_idle_ctr     = 0L;
        ENABLE();
        usage = 100.0 - (100.0 * (double)idle / max);  // Compute and display statistics
        sprintf( s, "%d", ctxsw );
        disp_str( 17, 13, s);
        sprintf( s, "%5.2f %%", usage );
        disp_str( 32, 13, s);
        sprintf( s, "%7.0f / %7.0f ", (double)idle, max );
        disp_str( 12, 14, s );
        _dos_gettime(&t);
        sprintf( s, "%2d:%02d:%02d.%02d\n", t.hour, t.minute, t.second, t.hsecond);
        disp_str( 8, 15, s );
        sprintf( s, "%d", os_get_ticksize() );
        disp_str( 35, 15, s);
        os_tick_dly ( 10 );         // Wait 10 system ticks
    }
}
```

11.8. Resultado de compilación

```
MAKE Version 3.6  Copyright (c) 1992 Borland International

Available memory 4896192 bytes

      bcc +rtk.CFG -c rtkernel.c
Borland C++  Version 3.1 Copyright (c) 1992 Borland International
rtkernel.c:
Loaded pre-compiled headers.

      Available memory 3945940
      tlib rtkernel /c -+rtkernel.obj
TLIB 3.02 Copyright (c) 1992 Borland International
      TASM /MX /ZI /O I186L_A.ASM,I186L_A.OBJ
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland International

Assembling file:   I186L_A.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  244k

      tlib rtkernel /c -+i186l_a.obj
```

```
TLIB 3.02 Copyright (c) 1992 Borland International
        bcc +rtk.CFG -c i186l_c.c
Borland C++  Version 3.1 Copyright (c) 1992 Borland International
i186l_c.c:
Loaded pre-compiled headers.

        Available memory 3925140
        tlib rtkernel /c -+i186l_c.obj
TLIB 3.02 Copyright (c) 1992 Borland International
MAKE Version 3.6  Copyright (c) 1992 Borland International

Available memory 4896192 bytes

        bcc +EX3.CFG -c util.c ex3.c
Borland C++  Version 3.1 Copyright (c) 1992 Borland International
util.c:
Loaded pre-compiled headers.
ex3.c:
Loaded pre-compiled headers.

        Available memory 3931916
        tlink /v/x/c/P-/LC:\BC\LIB;d:\bc\lib @MAKE0000.$$$
Turbo Link  Version 5.1 Copyright (c) 1992 Borland International
```

11.9. Código generado por el compilador

```
File RTKERNEL.ASM:
        .386p
        ifndef  ??version
        endm
publicdll macro name
        public  name
        endm
        endif
RTKERNEL_TEXT   segment byte public use16 'CODE'
RTKERNEL_TEXT   ends
DGROUP  group   _DATA,_BSS
        assume  cs:RTKERNEL_TEXT,ds:DGROUP
_DATA   segment word public use16 'DATA'
d@      label   byte
d@w     label   word
_DATA   ends
_BSS    segment word public use16 'BSS'
b@      label   byte
b@w     label   word
_BSS    ends
_DATA   segment word public use16 'DATA'
_divisor        label   word
        db      255
        db      255
tick_size       label   word
        db      55
        db      0
_DATA   ends
RTKERNEL_TEXT   segment byte public use16 'CODE'
  ;
  ;   void os_init ( void )
  ;
        assume  cs:RTKERNEL_TEXT
_os_init        proc    far
        push    bp
        mov     bp,sp
        sub     sp,8
        push    si
        push    di
  ;
```

```
;   {
;       int     i;
;
;       DISABLE();
;
    cli
;
;       old_tick_isr = getvect ( 0x08 );                                    // get MS-DOS's tick vector
;
    push    8
    call    far ptr _getvect
    add     sp,2
    mov     word ptr DGROUP:old_tick_isr+2,dx
    mov     word ptr DGROUP:old_tick_isr,ax
;
;       setvect ( 0x81, old_tick_isr );                                     // store MS-DOS's tick to chain
;
    push    dx
    push    ax
    push    129
    call    far ptr _setvect
    add     sp,6
;
;       setvect ( rtKERNEL, (void interrupt (*)(void))os_ctx_sw );      // rtKERNEL context switch vector
;
    push    seg _os_ctx_sw
    push    offset _os_ctx_sw
    push    128
    call    far ptr _setvect
    add     sp,6
;
;      setvect ( 0x08, (void interrupt (*)(void))new_tick_isr );   // new int. handler
;
    push    seg _new_tick_isr
    push    offset _new_tick_isr
    push    8
    call    far ptr _setvect
    add     sp,6
;
;       ENABLE();
;
    sti
;
;
;       os_time         = 0L;
;
    mov     dword ptr DGROUP:os_time,large 0
;
;       os_tcb_cur      = NULL;
;
    mov     dword ptr DGROUP:_os_tcb_cur,large 0
;
;       os_tcb_run              = NULL;
;
    mov     dword ptr DGROUP:_os_tcb_run,large 0
;
;       os_int_nesting  = 0;
;
    mov     byte ptr DGROUP:os_int_nesting,0
;
;       os_lock_nesting = 0;
;
    mov     byte ptr DGROUP:os_lock_nesting,0
;
;       os_running      = FALSE;
;
    mov     word ptr DGROUP:_os_running,0
;
;       os_idle_ctr     = 0L;
;
    mov     dword ptr DGROUP:_os_idle_ctr,large 0
;
;       os_ctx_sw_ctr   = 0;
```

```
  ;
      mov     word ptr DGROUP:_os_ctx_sw_ctr,0
  ;
  ;
  ;           for ( i = 0; i < OS_MAX_TASKS; i++ )    // init. list of free TCBs
  ;
      xor     si,si
      xor     di,di
      mov     word ptr [bp-2],offset DGROUP:os_tcb_task_tbl
@1@170:
  ;
  ;           {
  ;               if ( i < OS_MAX_TASKS-1 )
  ;
      cmp     si,63
      jge     short @1@226
  ;
  ;               os_tcb_tbl [ i ].next_free      = &os_tcb_tbl [ i + 1 ];
  ;
      mov     ax,di
      add     ax,offset DGROUP:os_tcb_tbl+42
      mov     word ptr DGROUP:os_tcb_tbl[di+24],ds
      mov     word ptr DGROUP:os_tcb_tbl[di+22],ax
      jmp     short @1@254
@1@226:
  ;
  ;               else
  ;                   os_tcb_tbl [ i ].next_free      = NULL;
  ;
      mov     word ptr DGROUP:os_tcb_tbl[di+24],0
      mov     word ptr DGROUP:os_tcb_tbl[di+22],0
@1@254:
  ;
  ;
  ;               os_tcb_tbl [ i ].task_id        = i;
  ;
      mov     word ptr DGROUP:os_tcb_tbl[di+11],si
  ;
  ;               os_tcb_tbl [ i ].task_stat      = OS_TASK_NULL;
  ;
      mov     word ptr DGROUP:os_tcb_tbl[di+8],6
  ;
  ;               os_tcb_task_tbl [ i ]           = NULL;
  ;
      mov     bx,word ptr [bp-2]
      mov     word ptr [bx+2],0
      mov     word ptr [bx],0
      add     di,42
      add     word ptr [bp-2],4
      inc     si
      cmp     di,2688
      jne     short @1@170
  ;
  ;           }
  ;
  ;           os_tcb_free_list                            = os_tcb_tbl;
  ;
      mov     word ptr DGROUP:os_tcb_free_list+2,ds
      mov     word ptr DGROUP:os_tcb_free_list,offset DGROUP:os_tcb_tbl
  ;
  ;
  ;           for ( i = 0; i < (OS_MAX_EVENTS - 1); i++ )     // init. list of free ECB
  ;
      xor     si,si
      mov     word ptr [bp-4],0
@1@450:
  ;
  ;           {
  ;               os_ecb_tbl [ i ].os_event_ptr               = &os_ecb_tbl [ i + 1 ];
  ;
      mov     ax,word ptr [bp-4]
      add     ax,offset DGROUP:os_ecb_tbl+10
      mov     bx,word ptr [bp-4]
```

```
    mov     word ptr DGROUP:os_ecb_tbl[bx+4],ds
    mov     word ptr DGROUP:os_ecb_tbl[bx+2],ax
;
;   //              os_ecb_tbl [ i ].os_tsk_cnt                     = 0;
;                   os_ecb_tbl [ i ].os_tcb_blk_task        = NULL;
;
    mov     word ptr DGROUP:os_ecb_tbl[bx+8],0
    mov     word ptr DGROUP:os_ecb_tbl[bx+6],0
    add     word ptr [bp-4],10
    inc     si
    cmp     word ptr [bp-4],190
    jne     short @1@450
;
;           }
;
;           os_ecb_tbl [ OS_MAX_EVENTS - 1 ].os_event_ptr   = NULL;
;
    mov     dword ptr DGROUP:os_ecb_tbl+192,large 0
;
;           os_ecb_free_list                                                = os_ecb_tbl;
;
    mov     word ptr DGROUP:os_ecb_free_list+2,ds
    mov     word ptr DGROUP:os_ecb_free_list,offset DGROUP:os_ecb_tbl
;
;
;           for ( i = 0; i < (OS_MAX_QS - 1); i++ ) // init. list of free QCB
;
    xor     si,si
    mov     word ptr [bp-6],0
@1@618:
;
;               os_qcb_tbl [ i ].os_qcb_ptr = &os_qcb_tbl [ i + 1 ];
;
    mov     ax,word ptr [bp-6]
    add     ax,offset DGROUP:os_qcb_tbl+15
    mov     bx,word ptr [bp-6]
    mov     word ptr DGROUP:os_qcb_tbl[bx+2],ds
    mov     word ptr DGROUP:os_qcb_tbl[bx],ax
    add     word ptr [bp-6],15
    inc     si
    cmp     word ptr [bp-6],60
    jne     short @1@618
;
;
;           os_qcb_tbl [ OS_MAX_QS - 1 ].os_qcb_ptr = NULL;
;
    mov     dword ptr DGROUP:os_qcb_tbl+60,large 0
;
;           os_qcb_free_list                                        = os_qcb_tbl;
;
    mov     word ptr DGROUP:os_qcb_free_list+2,ds
    mov     word ptr DGROUP:os_qcb_free_list,offset DGROUP:os_qcb_tbl
;
;
;           for ( i = 0; i < OS_LO_PRIO; i++ )
;
    xor     si,si
    mov     word ptr [bp-8],offset DGROUP:os_tcb_curprio
@1@786:
;
;                   os_tcb_curprio [ i ]    = NULL;
;
    mov     bx,word ptr [bp-8]
    mov     word ptr [bx+2],0
    mov     word ptr [bx],0
    add     word ptr [bp-8],4
    inc     si
    cmp     word ptr [bp-8],offset DGROUP:os_tcb_curprio+16
    jne     short @1@786
;
;
;           os_task_create ( os_task_idle, NULL, 512, OS_LO_PRIO-1, &idle_tsk_id );
;
```

```
        push    ds
        push    offset DGROUP:idle_tsk_id
        push    3
        push    large 002000000h
        push    0
        push    seg os_task_idle
        push    offset os_task_idle
        call    far ptr _os_task_create
        add     sp,16
    ;
    ;   }
    ;
        pop     di
        pop     si
        leave
        ret
_os_init        endp
    ;
    ;   void os_end ( void )
    ;
        assume  cs:RTKERNEL_TEXT
_os_end proc    far
    ;
    ;   {
    ;   //WARNING: no allowed automatic var here
    ;
    ;           os_running      = FALSE;
    ;
        mov     word ptr DGROUP:_os_running,0
    ;
    ;           DISABLE();
    ;
        cli
    ;
    ;           outportb ( BASE_ADRESS_TIMER + 3, WCONTROL );   // 8253 mode
    ;
        mov     dx,67
        mov     al,54
        out     dx,al
    ;
    ;           outportb ( BASE_ADRESS_TIMER + 0, 0xff );               // 1ro. low
    ;
        mov     dx,64
        mov     al,255
        out     dx,al
    ;
    ;           outportb ( BASE_ADRESS_TIMER + 0, 0xff );               // 2do. high
    ;
        out     dx,al
    ;
    ;           os_dest_alltasks ( );                           //destroy all tasks
    ;
        call    far ptr os_dest_alltasks
    ;
    ;           _BP = old_bp;                                       //restore old bp
    ;
        mov     bp,word ptr DGROUP:old_bp
    ;
    ;           _SP = old_sp;                                       //restore old sp
    ;
        mov     sp,word ptr DGROUP:old_sp
    ;
    ;           _SS     = old_ss;                                       //restore old ss
    ;
        mov     ss,word ptr DGROUP:old_ss
    ;
    ;           setvect ( 0x08, old_tick_isr );                 //restore old int. handler
    ;
        push    dword ptr DGROUP:old_tick_isr
        push    8
        call    far ptr _setvect
        add     sp,6
    ;
```

```
  ;         farfree ( os_tcb_cur->allocmem_ptr );   //free memory of cur. task
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      push    dword ptr es:[bx+4]
      call    far ptr _farfree
      add     sp,4
  ;
  ;         ENABLE();
  ;
      sti
  ;
  ;   }                                                   //return to next inst. after os_start();
  ;
      ret
_os_end endp
  ;
  ;   void os_dest_alltasks ( void )
  ;
      assume  cs:RTKERNEL_TEXT
os_dest_alltasks        proc    far
      push    bp
      mov     bp,sp
      sub     sp,4
      push    si
      push    di
  ;
  ;   {
  ;         int i;
  ;         OS_TCB *ptcb;
  ;
  ;         for ( i=0; i<OS_MAX_TASKS; i++)
  ;
      xor     di,di
      mov     si,offset DGROUP:os_tcb_task_tbl
@3@86:
  ;
  ;               if ( (ptcb = os_tcb_task_tbl [ i ]) != NULL && ptcb != os_tcb_cur )
  ;
      mov     ax,word ptr [si+2]
      mov     dx,word ptr [si]
      mov     word ptr [bp-2],ax
      mov     word ptr [bp-4],dx
      or      dx,ax
      je      short @3@170
      mov     eax,dword ptr [bp-4]
      cmp     eax,dword ptr DGROUP:_os_tcb_cur
      je      short @3@170
  ;
  ;                       os_task_del ( ptcb->task_id );
  ;
      les     bx,dword ptr [bp-4]
      push    word ptr es:[bx+11]
      call    far ptr _os_task_del
      add     sp,2
@3@170:
      add     si,4
      inc     di
      cmp     si,offset DGROUP:os_tcb_task_tbl+256
      jne     short @3@86
  ;
  ;   }
  ;
      pop     di
      pop     si
      leave
      ret
os_dest_alltasks        endp
  ;
  ;   static void far os_task_idle ( void *data )
  ;
      assume  cs:RTKERNEL_TEXT
os_task_idle    proc    far
      push    bp
```

```
        mov     bp,sp
    ;
    ;   {
    ;           data = data;
    ;
        mov     eax,dword ptr [bp+6]
        mov     dword ptr [bp+6],eax
@4@58:
    ;
    ;
    ;           while (1)
    ;           {
    ;           DISABLE();
    ;
        cli
    ;
    ;               os_idle_ctr++;
    ;
        inc     dword ptr DGROUP:_os_idle_ctr
    ;
    ;           ENABLE();
    ;
        sti
        jmp     short @4@58
    ;
    ;       }
    ;   }
    ;
        pop     bp
        ret
os_task_idle    endp
    ;
    ;   void os_start ( void )
    ;
        assume  cs:RTKERNEL_TEXT
_os_start       proc    far
    ;
    ;   {
    ;   //WARNING: no allowed automatic var. here
    ;
    ;           old_bp          = _BP;          // salvage bp
    ;
        mov     word ptr DGROUP:old_bp,bp
    ;
    ;           old_sp          = _SP;          // freeze return address
    ;
        mov     word ptr DGROUP:old_sp,sp
    ;
    ;           old_ss                  = _SS;          // freeze return address
    ;
        mov     word ptr DGROUP:old_ss,ss
    ;
    ;           os_running              = 1;
    ;
        mov     word ptr DGROUP:_os_running,1
    ;
    ;           os_start_run();                         // no return
    ;
        call    far ptr _os_start_run
    ;
    ;   }
    ;
        ret
_os_start       endp
    ;
    ;   void os_sched ( void )
    ;
        assume  cs:RTKERNEL_TEXT
_os_sched       proc    far
    ;
    ;   {
    ;           DISABLE();
    ;
```

```
        cli
    ;
    ;
    ;       if ( !( os_lock_nesting | os_int_nesting ) )// task scheduling must be enabled and not ISR level
    ;
        mov     al,byte ptr DGROUP:os_lock_nesting
        or      al,byte ptr DGROUP:os_int_nesting
        mov     ah,0
        or      ax,ax
        jne     short @6@142
    ;
    ;           if ( os_tcb_run != os_tcb_cur )
    ;
        mov     eax,dword ptr DGROUP:_os_tcb_run
        cmp     eax,dword ptr DGROUP:_os_tcb_cur
        je      short @6@142
    ;
    ;           {
    ;           os_ctx_sw_ctr++;        // increment context switch counter
    ;
        inc     word ptr DGROUP:_os_ctx_sw_ctr
    ;
    ;               OS_TASK_SW();           // context switch through interrupt
    ;
        INT     080H
@6@142:
    ;
    ;           }
    ;       ENABLE();
    ;
        sti
    ;
    ;   }
    ;
        ret
_os_sched       endp
    ;
    ;   void os_sched_lock ( void )
    ;
        assume  cs:RTKERNEL_TEXT
_os_sched_lock  proc    far
    ;
    ;   {
    ;       DISABLE();
    ;
        cli
    ;
    ;       os_lock_nesting++;                      // increment lock nesting level
    ;
        inc     byte ptr DGROUP:os_lock_nesting
    ;
    ;       ENABLE();
    ;
        sti
    ;
    ;   }
    ;
        ret
_os_sched_lock  endp
    ;
    ;   void os_sched_unlock ( void )
    ;
        assume  cs:RTKERNEL_TEXT
_os_sched_unlock        proc    far
    ;
    ;   {
    ;       DISABLE();
    ;
        cli
    ;
    ;       if ( os_lock_nesting )
    ;
        cmp     byte ptr DGROUP:os_lock_nesting,0
```

```
        je      short @8@170
  ;
  ;             {
  ;                os_lock_nesting--;                 // decrement lock nesting level
  ;
        dec     byte ptr DGROUP:os_lock_nesting
  ;
  ;                if ( !( os_lock_nesting | os_int_nesting) )
  ;
        mov     al,byte ptr DGROUP:os_lock_nesting
        or      al,byte ptr DGROUP:os_int_nesting
        mov     ah,0
        or      ax,ax
        jne     short @8@114
  ;
  ;                    // See if scheduling re-enabled and not an ISR
  ;                    os_sched();                    // See if a higher priority task is ready
  ;
        push    cs
        call    near ptr _os_sched
        ret
@8@114:
  ;
  ;                    else
  ;                  ENABLE();
  ;
        sti
  ;
  ;           }
  ;
        ret
@8@170:
  ;
  ;             else
  ;             ENABLE();
  ;
        sti
  ;
  ;     }
  ;
        ret
_os_sched_unlock        endp
  ;
  ;     RETCODE os_tcb_init ( PRIO prio, void far *stk, void far *pmem, TASK_ID *tsk_id )
  ;
        assume  cs:RTKERNEL_TEXT
_os_tcb_init    proc    far
        push    bp
        mov     bp,sp
        sub     sp,4
  ;
  ;     {
  ;             OS_TCB *ptcb;
  ;
  ;             *tsk_id = NULL_ID;
  ;
        les     bx,dword ptr [bp+16]
        mov     word ptr es:[bx],00000FFFFh
  ;
  ;
  ;             if ( prio > OS_LO_PRIO - 1 )
  ;
        cmp     byte ptr [bp+6],3
        jle     short @9@86
  ;
  ;                 return OS_PRIO_ERR;
  ;
        mov     ax,41
        leave
        ret
@9@86:
  ;
  ;
```

```
;       DISABLE();
;
    cli
;
;       ptcb = os_tcb_free_list;        // get a free TCB from the free TCB list
;
    mov     eax,dword ptr DGROUP:os_tcb_free_list
    mov     dword ptr [bp-4],eax
;
;
;       if ( ptcb != NULL )
;
    cmp     dword ptr DGROUP:os_tcb_free_list,large 0
    je      short @9@338
;
;       {
;       os_tcb_free_list              = ptcb->next_free;     // Update pointer to free TCB list
;
    les     bx,dword ptr [bp-4]
    mov     eax,dword ptr es:[bx+22]
    mov     dword ptr DGROUP:os_tcb_free_list,eax
;
;           ENABLE();
;
    sti
;
;       ptcb->stack_ptr                     = stk;                      // Load Stack pointer in TCB
;
    les     bx,dword ptr [bp-4]
    mov     eax,dword ptr [bp+8]
    mov     dword ptr es:[bx],eax
;
;               ptcb->allocmem_ptr                  = pmem;
;
    mov     eax,dword ptr [bp+12]
    mov     dword ptr es:[bx+4],eax
;
;           ptcb->prio                      = prio;                     // Load task priority into TCB
;
    mov     al,byte ptr [bp+6]
    mov     byte ptr es:[bx+10],al
;
;           ptcb->dly_time                  = 0L;
;
    mov     dword ptr es:[bx+13],large 0
;
;       ptcb->pecb                          = NULL;                     // Task is not pending on an event
;
    mov     dword ptr es:[bx+17],large 0
;
;           DISABLE();
;
    cli
;
;               *tsk_id                             = ptcb->task_id;
;
    les     bx,dword ptr [bp-4]
    mov     ax,word ptr es:[bx+11]
    les     bx,dword ptr [bp+16]
    mov     word ptr es:[bx],ax
;
;               os_tcb_task_tbl [ *tsk_id ]     = ptcb;
;
    mov     bx,word ptr es:[bx]
    shl     bx,2
    mov     ax,word ptr [bp-2]
    mov     dx,word ptr [bp-4]
    mov     word ptr DGROUP:os_tcb_task_tbl[bx+2],ax
    mov     word ptr DGROUP:os_tcb_task_tbl[bx],dx
;
;
;               os_tcb_ins_rdylist ( ptcb );
;
```

```
        push    ax
        push    dx
        call    far ptr os_tcb_ins_rdylist
        add     sp,4
    ;
    ;               ENABLE();
    ;
        sti
    ;
    ;
    ;               return ( OS_NO_ERR );
    ;
        xor     ax,ax
        leave
        ret
@9@338:
    ;
    ;           }
    ;           else
    ;           {
    ;               ENABLE ();
    ;
        sti
    ;
    ;               return ( OS_NO_MORE_TCB );
    ;
        mov     ax,70
    ;
    ;       }
    ;   }
    ;
        leave
        ret
_os_tcb_init    endp
    ;
    ;   void os_int_enter ( void )
    ;
        assume  cs:RTKERNEL_TEXT
_os_int_enter   proc    far
    ;
    ;   {
    ;           DISABLE();
    ;
        cli
    ;
    ;           os_int_nesting++;                   // increment ISR nesting level
    ;
        inc     byte ptr DGROUP:os_int_nesting
    ;
    ;           ENABLE();
    ;
        sti
    ;
    ;   }
    ;
        ret
_os_int_enter   endp
    ;
    ;   WORD os_int_exit ( void )
    ;
        assume  cs:RTKERNEL_TEXT
_os_int_exit    proc    far
    ;
    ;   {
    ;           DISABLE();
    ;
        cli                                         ; ~3
    ;
    ;
    ;           if ( !( --os_int_nesting | os_lock_nesting ) )  // reschedule only if all ISRs completed & not locked
    ;
        mov     al,byte ptr DGROUP:os_int_nesting               ;
        add     al,255
```

```
    mov     byte ptr DGROUP:os_int_nesting,al
    or      al,byte ptr DGROUP:os_lock_nesting
    mov     ah,0
    or      ax,ax
    je short        @@0
    jmp     @11@338
@@0:
  ;
  ;         {
  ;               register int i;
  ;
  ;               for ( i=0 ; i < OS_LO_PRIO; i++ )
  ;
    xor     cx,cx
    mov     dx,offset DGROUP:os_tcb_curprio
@11@114:
  ;
  ;               {
  ;                     if ( os_tcb_curprio [ i ] != NULL )
  ;
    mov     bx,dx
    mov     ax,word ptr [bx]
    or      ax,word ptr [bx+2]
    je      short @11@254
  ;
  ;                     {
  ;                           if ( i == os_tcb_current_prio )
  ;
    cmp     cx,word ptr DGROUP:os_tcb_current_prio
    jne     short @11@226
  ;
  ;                           {
  ;                                 if ( os_tcb_cur->next_rdy != os_tcb_run )
  ;
    les     bx,dword ptr DGROUP:_os_tcb_cur
    mov     eax,dword ptr es:[bx+26]
    cmp     eax,dword ptr DGROUP:_os_tcb_run
    je      short @11@338
  ;
  ;                                 {
  ;                                       os_tcb_run->task_stat   = OS_TASK_RDY;
  ;
    les     bx,dword ptr DGROUP:_os_tcb_run
    mov     word ptr es:[bx+8],1
  ;
  ;                                       os_tcb_run                      = os_tcb_run->next_rdy;
  ;
    mov     eax,dword ptr es:[bx+26]
    mov     dword ptr DGROUP:_os_tcb_run,eax
  ;
  ;                                       os_tcb_run->task_stat   = OS_TASK_RUN;
  ;
    les     bx,dword ptr DGROUP:_os_tcb_run
    mov     word ptr es:[bx+8],0
  ;
  ;                                       os_ctx_sw_ctr++;
  ;
    inc     word ptr DGROUP:_os_ctx_sw_ctr
  ;
  ;                                       return -1;              // perform interrupt level context switch
  ;
    mov     ax,00000FFFFh
    ret
@11@226:
  ;
  ;                           }
  ;                           }
  ;                           else
  ;                           {
  ;                                 os_tcb_run->task_stat   = OS_TASK_RDY;
  ;
    les     bx,dword ptr DGROUP:_os_tcb_run
    mov     word ptr es:[bx+8],1
```

```
  ;
  ;                                 os_tcb_run                          = os_tcb_curprio [ i ];
  ;
      mov     bx,cx
      shl     bx,2
      mov     ax,word ptr DGROUP:os_tcb_curprio[bx+2]
      mov     dx,word ptr DGROUP:os_tcb_curprio[bx]
      mov     word ptr DGROUP:_os_tcb_run+2,ax
      mov     word ptr DGROUP:_os_tcb_run,dx
  ;
  ;                                 os_tcb_current_prio             = i;
  ;
      mov     word ptr DGROUP:os_tcb_current_prio,cx
  ;
  ;                                 os_ctx_sw_ctr++;
  ;
      inc     word ptr DGROUP:_os_ctx_sw_ctr
  ;
  ;                                 return -1;      // perform interrupt level context switch
  ;
      mov     ax,00000FFFFh
      ret
@11@254:
      add     dx,4
      inc     cx
      cmp     dx,offset DGROUP:os_tcb_curprio+16
      jne     short @11@114
@11@338:
  ;
  ;                               }
  ;                       break;
  ;                       }
  ;                   }
  ;           }
  ;           ENABLE();
  ;
      sti
  ;
  ;           return 0;       // no perform interrupt level context switch
  ;
      xor     ax,ax
  ;
  ;   }
  ;
      ret
_os_int_exit    endp
  ;
  ;   RETCODE os_task_del ( TASK_ID tsk_id )
  ;
      assume  cs:RTKERNEL_TEXT
_os_task_del    proc    far
      push    bp
      mov     bp,sp
      sub     sp,4
      push    si
      mov     si,word ptr [bp+6]
  ;
  ;   {
  ;           register OS_TCB         *ptcb;
  ;
  ;           if ( tsk_id > OS_MAX_TASKS )
  ;
      cmp     si,64
      jbe     short @12@86
  ;
  ;               return OS_TSK_NO_EXIST;
  ;
      mov     ax,71
      pop     si
      leave
      ret
@12@86:
  ;
```

```
  ;
  ;         if ( os_running && tsk_id == idle_tsk_id )
  ;
      cmp     word ptr DGROUP:_os_running,0
      je      short @12@170
      cmp     si,word ptr DGROUP:idle_tsk_id
      jne     short @12@170
  ;
  ;             return ( OS_TASK_DEL_IDLE );        // Not allowed to delete idle task
  ;
      mov     ax,61
      pop     si
      leave
      ret
@12@170:
  ;
  ;
  ;         ptcb    = os_tcb_task_tbl [ tsk_id ];
  ;
      mov     bx,si
      shl     bx,2
      mov     ax,word ptr DGROUP:os_tcb_task_tbl[bx+2]
      mov     dx,word ptr DGROUP:os_tcb_task_tbl[bx]
      mov     word ptr [bp-2],ax
      mov     word ptr [bp-4],dx
  ;
  ;         if ( ptcb == NULL )                             // Task to delete must exist
  ;
      cmp     dword ptr [bp-4],large 0
      jne     short @12@226
  ;
  ;             return OS_TSK_NO_EXIST;
  ;
      mov     ax,71
      pop     si
      leave
      ret
@12@226:
  ;
  ;
  ;         if ( ptcb->task_stat == OS_TASK_NULL )
  ;
      les     bx,dword ptr [bp-4]
      cmp     word ptr es:[bx+8],6
      jne     short @12@282
  ;
  ;             return OS_TSK_NO_EXIST;
  ;
      mov     ax,71
      pop     si
      leave
      ret
@12@282:
  ;
  ;
  ;         DISABLE ( );
  ;
      cli
  ;
  ;
  ;         os_tcb_task_tbl [ tsk_id ]      = NULL;
  ;
      mov     bx,si
      shl     bx,2
      mov     word ptr DGROUP:os_tcb_task_tbl[bx+2],0
      mov     word ptr DGROUP:os_tcb_task_tbl[bx],0
  ;
  ;
  ;         switch ( ptcb->task_stat )
  ;
      les     bx,dword ptr [bp-4]
      mov     bx,word ptr es:[bx+8]
      cmp     bx,5
```

```
        ja      short @12@506
        shl     bx,1
        jmp     word ptr cs:@12@C418[bx]
@12@422:
  ;
  ;         {
  ;               case OS_TASK_RUN:
  ;               case OS_TASK_RDY:
  ;                       os_tcb_del_rdylist ( ptcb );
  ;
        push    dword ptr [bp-4]
        call    far ptr os_tcb_del_rdylist
        add     sp,4
  ;
  ;                       break;
  ;
        jmp     short @12@506
@12@450:
  ;
  ;
  ;               case OS_TASK_DLY:
  ;                       os_tcb_del_dlylist ( ptcb );
  ;
        push    dword ptr [bp-4]
        call    far ptr os_tcb_del_dlylist
        add     sp,4
  ;
  ;                       break;
  ;
        jmp     short @12@506
@12@478:
  ;
  ;
  ;               case OS_TASK_SEM:
  ;               case OS_TASK_MBOX:
  ;               case OS_TASK_Q:
  ;                       os_tcb_quit_evnlist ( ptcb );
  ;
        push    dword ptr [bp-4]
        call    far ptr os_tcb_quit_evnlist
        add     sp,4
  ;
  ;                       os_tcb_del_dlylist ( ptcb );
  ;
        push    dword ptr [bp-4]
        call    far ptr os_tcb_del_dlylist
        add     sp,4
  ;
  ;                       break;
  ;
@12@506:
  ;
  ;         }
  ;         ptcb->next_free         = os_tcb_free_list;           //insert in tcb free list
  ;
        les     bx,dword ptr [bp-4]
        mov     eax,dword ptr DGROUP:os_tcb_free_list
        mov     dword ptr es:[bx+22],eax
  ;
  ;         os_tcb_free_list        = ptcb;
  ;
        mov     eax,dword ptr [bp-4]
        mov     dword ptr DGROUP:os_tcb_free_list,eax
  ;
  ;         ptcb->task_stat         = OS_TASK_NULL;
  ;
        mov     word ptr es:[bx+8],6
  ;
  ;         farfree ( ptcb->allocmem_ptr );                     // free stack memory
  ;
        push    dword ptr es:[bx+4]
        call    far ptr _farfree
        add     sp,4
```

```
  ;
  ;         if ( ptcb == os_tcb_cur )       // task to delete is the current active task?
  ;
      mov     eax,dword ptr [bp-4]
      cmp     eax,dword ptr DGROUP:_os_tcb_cur
      jne     short @12@562
  ;
  ;             os_sched();                         // never returns
  ;
      push    cs
      call    near ptr _os_sched
@12@562:
  ;
  ;         ENABLE();
  ;
      sti
  ;
  ;         return OS_NO_ERR;
  ;
      xor     ax,ax
  ;
  ;   }
  ;
      pop     si
      leave
      ret
_os_task_del    endp
@12@C418        label   word
      dw      @12@422
      dw      @12@422
      dw      @12@478
      dw      @12@478
      dw      @12@478
      dw      @12@450
  ;
  ;   void os_task_end ( void )
  ;
      assume  cs:RTKERNEL_TEXT
_os_task_end    proc    far
  ;
  ;   {
  ;         DISABLE ( );
  ;
      cli
  ;         os_tcb_task_tbl [ os_tcb_cur->task_id ] = NULL;
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     bx,word ptr es:[bx+11]
      shl     bx,2
      mov     word ptr DGROUP:os_tcb_task_tbl[bx+2],0
      mov     word ptr DGROUP:os_tcb_task_tbl[bx],0
  ;
  ;         os_tcb_del_rdylist ( os_tcb_cur );
  ;
      push    dword ptr DGROUP:_os_tcb_cur
      call    far ptr os_tcb_del_rdylist
      add     sp,4
  ;
  ;         os_tcb_cur->next_free   = os_tcb_free_list;           //insert in tcb free list
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     eax,dword ptr DGROUP:os_tcb_free_list
      mov     dword ptr es:[bx+22],eax
  ;
  ;         os_tcb_free_list                = os_tcb_cur;
  ;
      mov     eax,dword ptr DGROUP:_os_tcb_cur
      mov     dword ptr DGROUP:os_tcb_free_list,eax
  ;
  ;         os_tcb_cur->task_stat   = OS_TASK_NULL;
  ;
      mov     word ptr es:[bx+8],6
  ;
```

```
  ;         farfree ( os_tcb_cur->allocmem_ptr );
  ;
      push    dword ptr es:[bx+4]
      call    far ptr _farfree
      add     sp,4
  ;
  ;         os_sched();                     // never returns
  ;
      push    cs
      call    near ptr _os_sched
  ;
  ;   }
  ;
      ret
_os_task_end    endp
  ;
  ;   RETCODE os_task_change_prio ( TASK_ID tsk_id, PRIO newprio )
  ;
      assume  cs:RTKERNEL_TEXT
_os_task_change_prio    proc    far
      push    bp
      mov     bp,sp
      sub     sp,4
      push    si
      mov     si,word ptr [bp+6]
  ;
  ;   {
  ;         register OS_TCB         *ptcb = os_tcb_task_tbl [ tsk_id ];
  ;
      mov     bx,si
      shl     bx,2
      mov     ax,word ptr DGROUP:os_tcb_task_tbl[bx+2]
      mov     dx,word ptr DGROUP:os_tcb_task_tbl[bx]
      mov     word ptr [bp-2],ax
      mov     word ptr [bp-4],dx
  ;
  ;
  ;         if ( tsk_id > OS_MAX_TASKS )
  ;
      cmp     si,64
      jbe     short @14@86
  ;
  ;             return ( OS_TASK_DEL_ERR );
  ;
      mov     ax,60
      pop     si
      leave
      ret
@14@86:
  ;
  ;
  ;         if ( ptcb == NULL )                     // Task to delete must exist
  ;
      cmp     dword ptr [bp-4],large 0
      jne     short @14@142
  ;
  ;               return OS_TSK_NO_EXIST;
  ;
      mov     ax,71
      pop     si
      leave
      ret
@14@142:
  ;
  ;
  ;         if ( ptcb->task_stat == OS_TASK_NULL )
  ;
      les     bx,dword ptr [bp-4]
      cmp     word ptr es:[bx+8],6
      jne     short @14@198
  ;
  ;               return OS_TSK_NO_EXIST;
  ;
```

```
        mov     ax,71
        pop     si
        leave
        ret
@14@198:
  ;
  ;
  ;         if ( newprio > OS_LO_PRIO - 1 )
  ;
        cmp     byte ptr [bp+8],3
        jle     short @14@254
  ;
  ;                 return OS_PRIO_ERR;
  ;
        mov     ax,41
        pop     si
        leave
        ret
@14@254:
  ;
  ;         DISABLE();
  ;
        cli
  ;
  ;
  ;         switch ( ptcb->task_stat )
  ;
        les     bx,dword ptr [bp-4]
        mov     bx,word ptr es:[bx+8]
        cmp     bx,5
        ja      short @14@534
        shl     bx,1
        jmp     word ptr cs:@14@C402[bx]
@14@366:
  ;
  ;         {
  ;               case OS_TASK_RUN:
  ;                       os_tcb_del_rdylist ( ptcb );
  ;
        push    dword ptr [bp-4]
        call    far ptr os_tcb_del_rdylist
        add     sp,4
  ;
  ;                       ptcb->prio      = newprio;
  ;
        les     bx,dword ptr [bp-4]
        mov     al,byte ptr [bp+8]
        mov     byte ptr es:[bx+10],al
  ;
  ;                       os_tcb_ins_rdylist ( ptcb );
  ;
        push    word ptr [bp-2]
        push    bx
        call    far ptr os_tcb_ins_rdylist
        add     sp,4
  ;
  ;                       os_sched ( );
  ;
        push    cs
        call    near ptr _os_sched
  ;
  ;                       break;
  ;
        jmp     short @14@534
@14@394:
  ;
  ;
  ;               case OS_TASK_RDY:
  ;                       os_tcb_del_rdylist ( ptcb );
  ;
        push    dword ptr [bp-4]
        call    far ptr os_tcb_del_rdylist
        add     sp,4
```

```
  ;
  ;                     ptcb->prio      = newprio;
  ;
      les     bx,dword ptr [bp-4]
      mov     al,byte ptr [bp+8]
      mov     byte ptr es:[bx+10],al
  ;
  ;                     os_tcb_ins_rdylist ( ptcb );
  ;
      push    word ptr [bp-2]
      push    bx
      call    far ptr os_tcb_ins_rdylist
      add     sp,4
  ;
  ;                     ENABLE ( );
  ;
      sti
  ;
  ;                     break;
  ;
      jmp     short @14@534
@14@478:
  ;
  ;
  ;                 case OS_TASK_DLY:
  ;                 case OS_TASK_SEM:
  ;                 case OS_TASK_MBOX:
  ;                 case OS_TASK_Q:
  ;                     ptcb->prio      = newprio;
  ;
      les     bx,dword ptr [bp-4]
      mov     al,byte ptr [bp+8]
      mov     byte ptr es:[bx+10],al
  ;
  ;                     ENABLE ( );
  ;
      sti
@14@534:
  ;
  ;                     break;
  ;
  ;
  ;         }
  ;         return OS_NO_ERR;
  ;
      xor     ax,ax
  ;
  ;   }
  ;
      pop     si
      leave
      ret
_os_task_change_prio    endp
@14@C402        label   word
      dw      @14@366
      dw      @14@394
      dw      @14@478
      dw      @14@478
      dw      @14@478
      dw      @14@478
  ;
  ;   void os_tick_dly ( LONG ticks )
  ;
      assume  cs:RTKERNEL_TEXT
_os_tick_dly    proc    far
      push    bp
      mov     bp,sp
  ;
  ;   {
  ;         if ( ticks )
  ;
      cmp     dword ptr [bp+6],large 0
      je      short @15@114
```

```
  ;
  ;         {
  ;            DISABLE();
  ;
      cli
  ;
  ;
  ;            os_tcb_cur->dly_time = ticks;                    // Load ticks in TCB
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     eax,dword ptr [bp+6]
      mov     dword ptr es:[bx+13],eax
  ;
  ;
  ;               // delay current task
  ;               os_tcb_del_rdylist ( os_tcb_cur );
  ;
      push    word ptr DGROUP:_os_tcb_cur+2
      push    bx
      call    far ptr os_tcb_del_rdylist
      add     sp,4
  ;
  ;               os_tcb_ins_dlylist ( os_tcb_cur );
  ;
      push    dword ptr DGROUP:_os_tcb_cur
      call    far ptr os_tcb_ins_dlylist
      add     sp,4
  ;
  ;               os_tcb_cur->task_stat   =       OS_TASK_DLY;
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     word ptr es:[bx+8],5
  ;
  ;            os_sched();
  ;
      push    cs
      call    near ptr _os_sched
@15@114:
  ;
  ;       }
  ;   }
  ;
      pop     bp
      ret
_os_tick_dly    endp
  ;
  ;   void os_msec_dly ( LONG msecs )
  ;
      assume  cs:RTKERNEL_TEXT
_os_msec_dly    proc    far
      push    bp
      mov     bp,sp
  ;
  ;   {
  ;         os_tick_dly ( (LONG) msecs/tick_size );
  ;
      movzx   ebx,word ptr DGROUP:tick_size
      mov     eax,dword ptr [bp+6]
      xor     edx,edx
      push    eax
      push    cs
      call    near ptr _os_tick_dly
      add     sp,4
  ;
  ;   }
  ;
      pop     bp
      ret
_os_msec_dly    endp
  ;
  ;   void os_time_tick ( void )
  ;
      assume  cs:RTKERNEL_TEXT
```

```
_os_time_tick   proc    far
        push    bp
        mov     bp,sp
        sub     sp,4
  ;
  ;     {
  ;           DISABLE ( );
  ;
        cli
  ;
  ;       if ( os_tcb_dlylist != NULL )
  ;
        cmp     dword ptr DGROUP:os_tcb_dlylist,large 0
        je      short @17@254
  ;
  ;           {                       // Delayed or waiting for event with TO
  ;               register OS_TCB         *aux_ptcb       = os_tcb_dlylist;
  ;
        mov     eax,dword ptr DGROUP:os_tcb_dlylist
        mov     dword ptr [bp-4],eax
  ;
  ;
  ;               --os_tcb_dlylist->dly_time;     // decrement nbr of ticks to end of delay
  ;
        les     bx,dword ptr DGROUP:os_tcb_dlylist
        dec     dword ptr es:[bx+13]
        jmp     short @17@226
@17@114:
  ;
  ;               while ( !os_tcb_dlylist->dly_time )
  ;               {
  ;                   os_tcb_dlylist                  = os_tcb_dlylist->next_dly;
  ;
        les     bx,dword ptr DGROUP:os_tcb_dlylist
        mov     eax,dword ptr es:[bx+38]
        mov     dword ptr DGROUP:os_tcb_dlylist,eax
  ;
  ;                   if ( aux_ptcb->pecb != NULL )
  ;
        les     bx,dword ptr [bp-4]
        cmp     dword ptr es:[bx+17],large 0
        je      short @17@170
  ;
  ;                       os_tcb_quit_evnlist ( aux_ptcb );
  ;
        push    word ptr [bp-2]
        push    bx
        call    far ptr os_tcb_quit_evnlist
        add     sp,4
@17@170:
  ;
  ;                   os_tcb_ins_rdylist ( aux_ptcb );
  ;
        push    dword ptr [bp-4]
        call    far ptr os_tcb_ins_rdylist
        add     sp,4
  ;
  ;                   aux_ptcb->timeout_flag  = 1;
  ;
        les     bx,dword ptr [bp-4]
        or      byte ptr es:[bx+21],1
  ;
  ;                   aux_ptcb->task_stat             = OS_TASK_RDY;
  ;
        mov     word ptr es:[bx+8],1
  ;
  ;                   if ( os_tcb_dlylist == NULL )
  ;
        cmp     dword ptr DGROUP:os_tcb_dlylist,large 0
        je      short @17@254
  ;
  ;                       break;
  ;                   aux_ptcb                        = os_tcb_dlylist;
```

```
  ;
      mov     eax,dword ptr DGROUP:os_tcb_dlylist
      mov     dword ptr [bp-4],eax
@17@226:
      les     bx,dword ptr DGROUP:os_tcb_dlylist
      cmp     dword ptr es:[bx+13],large 0
      je      short @17@114
@17@254:
  ;
  ;                   }
  ;           }
  ;           os_time++;
  ;
      inc     dword ptr DGROUP:os_time
  ;
  ;           ENABLE();
  ;
      sti
  ;
  ;   }
  ;
      leave
      ret
_os_time_tick   endp
  ;
  ;   void os_time_set ( LONG ticks )
  ;
      assume  cs:RTKERNEL_TEXT
_os_time_set    proc    far
      push    bp
      mov     bp,sp
  ;
  ;   {
  ;           DISABLE();
  ;
      cli
  ;
  ;           os_time = ticks;
  ;
      mov     eax,dword ptr [bp+6]
      mov     dword ptr DGROUP:os_time,eax
  ;
  ;           ENABLE();
  ;
      sti
  ;
  ;   }
  ;
      pop     bp
      ret
_os_time_set    endp
  ;
  ;   TASK_ID os_get_cur_taskid ( void )
  ;
      assume  cs:RTKERNEL_TEXT
_os_get_cur_taskid      proc    far
  ;
  ;   {
  ;           return os_tcb_cur->task_id;
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     ax,word ptr es:[bx+11]
  ;
  ;   }
  ;
      ret
_os_get_cur_taskid      endp
  ;
  ;   LONG os_time_get ( void )
  ;
      assume  cs:RTKERNEL_TEXT
_os_time_get    proc    far
      push    bp
```

```
        mov     bp,sp
        sub     sp,4
    ;
    ;   {
    ;           TIME ticks;
    ;
    ;           DISABLE();
    ;
        cli
    ;
    ;           ticks = os_time;
    ;
        mov     eax,dword ptr DGROUP:os_time
        mov     dword ptr [bp-4],eax
    ;
    ;           ENABLE();
    ;
        sti
    ;
    ;           return ( ticks );
    ;
        mov     eax,dword ptr DGROUP:os_time
        shld    edx,eax,16
    ;
    ;   }
    ;
        leave
        ret
_os_time_get    endp
    ;
    ;   RETCODE os_set_ticksize ( WORD ticksize )       //ticksize in msecs.
    ;
        assume  cs:RTKERNEL_TEXT
_os_set_ticksize        proc    far
        push    bp
        mov     bp,sp
        mov     cx,word ptr [bp+6]
    ;
    ;   {
    ;           if ( ticksize > 54 || ticksize < 12 )
    ;
        cmp     cx,54
        ja      short @21@86
        cmp     cx,12
        jae     short @21@114
@21@86:
    ;
    ;               return OS_BAD_TICKSIZE;
    ;
        mov     ax,90
        pop     bp
        ret
@21@114:
    ;
    ;
    ;           DISABLE ( );
    ;
        cli
    ;
    ;
    ;           tick_size       = ticksize;
    ;
        mov     word ptr DGROUP:tick_size,cx
    ;
    ;           divisor         = (1193180L * ticksize/1000);
    ;
        movzx   eax,cx
        imul    eax,large 0001234DCh
        mov     ebx,large 1000
        cdq
        idiv    ebx
        mov     word ptr DGROUP:_divisor,ax
    ;
```

```
  ;
  ;         // TICK SIZE SET
  ;         // see "PROGRAMMERS PROBLEM SOLVER PC XT/AT" pags. 45-49
  ;         // channel #0, write low-then-high byte, mode=3, binary data
  ;
  ;         outportb ( BASE_ADRESS_TIMER + 3, WCONTROL );                    // 8253 mode
  ;
      mov     dx,67
      mov     al,54
      out     dx,al
  ;
  ;         outportb ( BASE_ADRESS_TIMER + 0, divisor       & 0xff );       // 1ro. low
  ;
      mov     al,byte ptr DGROUP:_divisor
      and     al,255
      mov     dx,64
      out     dx,al
  ;
  ;         outportb ( BASE_ADRESS_TIMER + 0, (divisor >> 8) & 0xff );      // 2do. high
  ;
      mov     ax,word ptr DGROUP:_divisor
      shr     ax,8
      and     al,255
      out     dx,al
  ;
  ;         ENABLE ( );
  ;
      sti
  ;
  ;         return OS_NO_ERR;
  ;
      xor     ax,ax
  ;
  ;   }
  ;
      pop     bp
      ret
_os_set_ticksize        endp
  ;
  ;   WORD os_get_ticksize ( void )
  ;
      assume  cs:RTKERNEL_TEXT
_os_get_ticksize        proc    far
  ;
  ;   {
  ;         return tick_size;
  ;
      mov     ax,word ptr DGROUP:tick_size
  ;
  ;   }
  ;
      ret
_os_get_ticksize        endp
  ;
  ;   OS_ECB *os_sem_create ( SWORD cnt )
  ;
      assume  cs:RTKERNEL_TEXT
_os_sem_create  proc    far
      push    bp
      mov     bp,sp
      sub     sp,4
      mov     dx,word ptr [bp+6]
  ;
  ;   {
  ;      register OS_ECB *pecb;
  ;
  ;         if ( cnt < 0 )  // semaphore cannot start negative
  ;
      or      dx,dx
      jge     short @23@86
  ;
  ;                 return NULL;
  ;
```

```
        xor     dx,dx
        xor     ax,ax
        leave
        ret
@23@86:
  ;
  ;
  ;       DISABLE();
  ;
        cli
  ;
  ;       pecb = os_ecb_free_list;            // get next free event control block
  ;
        mov     eax,dword ptr DGROUP:os_ecb_free_list
        mov     dword ptr [bp-4],eax
  ;
  ;       if ( os_ecb_free_list != NULL )     // See if pool of free ECB pool was empty
  ;
        cmp     dword ptr DGROUP:os_ecb_free_list,large 0
        je      short @23@170
  ;
  ;           os_ecb_free_list = (OS_ECB *)os_ecb_free_list->os_event_ptr;
  ;
        les     bx,dword ptr DGROUP:os_ecb_free_list
        mov     eax,dword ptr es:[bx+2]
        mov     dword ptr DGROUP:os_ecb_free_list,eax
@23@170:
  ;
  ;       ENABLE();
  ;
        sti
  ;
  ;       if ( pecb != NULL )
  ;
        cmp     dword ptr [bp-4],large 0
        je      short @23@254
  ;
  ;           {                       // Get an event control block
  ;               pecb->os_event_cnt      = cnt;  // Set semaphore value
  ;
        les     bx,dword ptr [bp-4]
        mov     word ptr es:[bx],dx
  ;
  ;               return ( pecb );
  ;
        mov     dx,word ptr [bp-2]
        mov     ax,word ptr [bp-4]
        leave
        ret
@23@254:
  ;
  ;       }
  ;           else
  ;           return ( NULL );        // Ran out of event control blocks
  ;
        xor     dx,dx
        xor     ax,ax
  ;
  ;   }
  ;
        leave
        ret
_os_sem_create  endp
  ;
  ;   RETCODE os_sem_wait ( OS_ECB *pecb, TIME timeout )
  ;
        assume  cs:RTKERNEL_TEXT
_os_sem_wait    proc    far
        push    bp
        mov     bp,sp
  ;
  ;   {
  ;           if ( pecb       == NULL)
```

```
  ;
      cmp     dword ptr [bp+6],large 0
      jne     short @24@86
  ;
  ;                   return OS_NULL_PECB;
  ;
      mov     ax,81
      pop     bp
      ret
@24@86:
  ;
  ;
  ;       DISABLE();
  ;
      cli
  ;
  ;       if ( --pecb->os_event_cnt < 0 )
  ;
      les     bx,dword ptr [bp+6]
      dec     word ptr es:[bx]
      jl short        @@1
      jmp     @24@478
@@1:
  ;
  ;           // must wait until event occurs
  ;           {
  ;           os_tcb_cur->pecb                = pecb;                 // store pointer to event control block in TCB
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     eax,dword ptr [bp+6]
      mov     dword ptr es:[bx+17],eax
  ;
  ;                   os_tcb_ins_evnlist ( os_tcb_cur );              // insert in event list
  ;
      push    word ptr DGROUP:_os_tcb_cur+2
      push    bx
      call    far ptr os_tcb_ins_evnlist
      add     sp,4
  ;
  ;                   os_tcb_del_rdylist ( os_tcb_cur );              // delete of rdy list
  ;
      push    dword ptr DGROUP:_os_tcb_cur
      call    far ptr os_tcb_del_rdylist
      add     sp,4
  ;
  ;           os_tcb_cur->task_stat   = OS_TASK_SEM;  // resource not available, pend on semaphore
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     word ptr es:[bx+8],2
  ;
  ;           os_tcb_cur->dly_time    = timeout;      // store pend timeout in TCB
  ;
      mov     eax,dword ptr [bp+10]
      mov     dword ptr es:[bx+13],eax
  ;
  ;                   os_tcb_cur->timeout_flag= 0;
  ;
      and     byte ptr es:[bx+21],254
  ;
  ;                   if ( timeout )                                          // timeout?...
  ;
      cmp     dword ptr [bp+10],large 0
      je      short @24@198
  ;
  ;                           os_tcb_ins_dlylist ( os_tcb_cur );      // insert in delay list
  ;
      push    word ptr DGROUP:_os_tcb_cur+2
      push    bx
      call    far ptr os_tcb_ins_dlylist
      add     sp,4
@24@198:
  ;
  ;           os_sched ( );           // find next highest priority task ready to run
```

```
  ;
      push    cs
      call    near ptr _os_sched
  ;
  ;                   DISABLE ( );
  ;
      cli
  ;
  ;                   os_tcb_cur->pecb              = NULL;
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     dword ptr es:[bx+17],large 0
  ;
  ;                   ENABLE ( );
  ;
      sti
  ;
  ;                   if ( os_tcb_cur->timeout_flag )
  ;
      les     bx,dword ptr DGROUP:_os_tcb_cur
      mov     al,byte ptr es:[bx+21]
      and     ax,1
      or      ax,ax
      je      short @24@366
  ;
  ;                         return OS_TIMEOUT;
  ;
      mov     ax,10
      pop     bp
      ret
@24@366:
  ;
  ;                   else
  ;                   {
  ;                         DISABLE ( );
  ;
      cli
  ;
  ;                         os_tcb_del_dlylist ( os_tcb_cur );
  ;
      push    dword ptr DGROUP:_os_tcb_cur
      call    far ptr os_tcb_del_dlylist
      add     sp,4
  ;
  ;                         ENABLE ( );
  ;
      sti
  ;
  ;                         return OS_NO_ERR;
  ;
      xor     ax,ax
      pop     bp
      ret
@24@478:
  ;
  ;                   }
  ;       }
  ;           else
  ;           {             // semaphore > 0, resource available
  ;           ENABLE();
  ;
      sti
  ;
  ;           return  OS_NO_ERR;
  ;
      xor     ax,ax
  ;
  ;       }
  ;   }
  ;
      pop     bp
      ret
_os_sem_wait    endp
```

```
  ;
  ;   RETCODE os_q_dest ( OS_ECB *pecb )
  ;
      assume  cs:RTKERNEL_TEXT
_os_q_dest      proc    far
      push    bp
      mov     bp,sp
      sub     sp,4
  ;
  ;   {
  ;         OS_QCB  *os_q;
  ;
  ;         if ( pecb == NULL )
  ;
      cmp     dword ptr [bp+6],large 0
      jne     short @25@86
  ;
  ;               return OS_NULL_PECB;
  ;
      mov     ax,81
      leave
      ret
@25@86:
  ;
  ;
  ;         if ( ( os_q = (OS_QCB*)pecb->os_event_ptr ) == NULL )
  ;
      les     bx,dword ptr [bp+6]
      mov     eax,dword ptr es:[bx+2]
      mov     dword ptr [bp-4],eax
      cmp     eax,large 0
      jne     short @25@142
  ;
  ;               return OS_Q_NULL;
  ;
      mov     ax,31
      leave
      ret
@25@142:
  ;
  ;
  ;         os_q->os_qcb_ptr                    = os_qcb_free_list;     //insert in free list
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr DGROUP:os_qcb_free_list
      mov     dword ptr es:[bx],eax
  ;
  ;         os_qcb_free_list                    = os_q;
  ;
      mov     eax,dword ptr [bp-4]
      mov     dword ptr DGROUP:os_qcb_free_list,eax
  ;
  ;         //farfree(os_q->qstart);
  ;         os_event_dest ( pecb );
  ;
      push    dword ptr [bp+6]
      call    far ptr _os_event_dest
      add     sp,4
  ;
  ;         return OS_NO_ERR;
  ;
      xor     ax,ax
  ;
  ;   }
  ;
      leave
      ret
_os_q_dest      endp
  ;
  ;   RETCODE os_event_dest ( OS_ECB *pecb )
  ;
      assume  cs:RTKERNEL_TEXT
_os_event_dest  proc    far
```

```
        push    bp
        mov     bp,sp
    ;
    ;   {
    ;           if ( pecb == NULL )
    ;
        cmp     dword ptr [bp+6],large 0
        jne     short @26@86
    ;
    ;                   return OS_NULL_PECB;
    ;
        mov     ax,81
        pop     bp
        ret
@26@86:
    ;
    ;
    ;           DISABLE ( );
    ;
        cli
    ;
    ;           os_tcb_purge_evnlist ( pecb );
    ;
        push    dword ptr [bp+6]
        call    far ptr os_tcb_purge_evnlist
        add     sp,4
    ;
    ;           ENABLE ( );
    ;
        sti
    ;
    ;           pecb->os_event_ptr      = os_ecb_free_list;
    ;
        les     bx,dword ptr [bp+6]
        mov     eax,dword ptr DGROUP:os_ecb_free_list
        mov     dword ptr es:[bx+2],eax
    ;
    ;           os_ecb_free_list        = pecb;
    ;
        mov     eax,dword ptr [bp+6]
        mov     dword ptr DGROUP:os_ecb_free_list,eax
    ;
    ;           return OS_NO_ERR;
    ;
        xor     ax,ax
    ;
    ;   }
    ;
        pop     bp
        ret
_os_event_dest  endp
    ;
    ;   RETCODE os_sem_signal ( OS_ECB *pecb )
    ;
        assume  cs:RTKERNEL_TEXT
_os_sem_signal  proc    far
        push    bp
        mov     bp,sp
    ;
    ;   {
    ;           if ( pecb       == NULL)
    ;
        cmp     dword ptr [bp+6],large 0
        jne     short @27@86
    ;
    ;                   return OS_NULL_PECB;
    ;
        mov     ax,81
        pop     bp
        ret
@27@86:
    ;
    ;
```

```
  ;       if ( pecb->os_event_cnt < INT_MAX )// make sure semaphore will not overflow
  ;
      les     bx,dword ptr [bp+6]
      cmp     word ptr es:[bx],32767
      jge     short @27@254
  ;
  ;           {
  ;              DISABLE();
  ;
      cli
  ;
  ;                   pecb->os_event_cnt++;
  ;
      les     bx,dword ptr [bp+6]
      inc     word ptr es:[bx]
  ;
  ;           if ( pecb->os_event_cnt <= 0 )
  ;
      cmp     word ptr es:[bx],0
      jg      short @27@198
  ;
  ;                   {
  ;                         os_tcb_del_evnlist ( pecb );    //signal the waiting task
  ;
      push    word ptr [bp+8]
      push    bx
      call    far ptr os_tcb_del_evnlist
      add     sp,4
  ;
  ;                         os_sched ( );   // find highest priority task ready to run
  ;
      push    cs
      call    near ptr _os_sched
  ;
  ;                         return OS_NO_ERR;
  ;
      xor     ax,ax
      pop     bp
      ret
@27@198:
  ;
  ;           }
  ;                else
  ;              ENABLE();
  ;
      sti
  ;
  ;           return OS_NO_ERR;
  ;
      xor     ax,ax
      pop     bp
      ret
@27@254:
  ;
  ;       }
  ;           else
  ;           return OS_SEM_OVF;
  ;
      mov     ax,51
  ;
  ;   }
  ;
      pop     bp
      ret
_os_sem_signal  endp
  ;
  ;   OS_ECB *os_mbox_create ( void *msg )
  ;
      assume  cs:RTKERNEL_TEXT
_os_mbox_create proc    far
      push    bp
      mov     bp,sp
      sub     sp,4
```

```
  ;
  ;   {
  ;       OS_ECB *pecb;
  ;
  ;       DISABLE();
  ;
      cli
  ;
  ;       pecb = os_ecb_free_list;            // Get next free event control block
  ;
      mov     eax,dword ptr DGROUP:os_ecb_free_list
      mov     dword ptr [bp-4],eax
  ;
  ;       if ( os_ecb_free_list != NULL )             // See if pool of free ECB pool was empty
  ;
      cmp     dword ptr DGROUP:os_ecb_free_list,large 0
      je      short @28@114
  ;
  ;           os_ecb_free_list = (OS_ECB *)os_ecb_free_list->os_event_ptr;
  ;
      les     bx,dword ptr DGROUP:os_ecb_free_list
      mov     eax,dword ptr es:[bx+2]
      mov     dword ptr DGROUP:os_ecb_free_list,eax
@28@114:
  ;
  ;       ENABLE();
  ;
      sti
  ;
  ;       if ( pecb != NULL )
  ;
      cmp     dword ptr [bp-4],large 0
      je      short @28@198
  ;
  ;           pecb->os_event_ptr    = msg;              // Deposit message in event control block
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr [bp+6]
      mov     dword ptr es:[bx+2],eax
@28@198:
  ;
  ;
  ;       return ( pecb );                // Return pointer to event control block
  ;
      mov     dx,word ptr [bp-2]
      mov     ax,word ptr [bp-4]
  ;
  ;   }
  ;
      leave
      ret
_os_mbox_create endp
  ;
  ;   void    *os_mbox_receive ( OS_ECB *pecb, TIME timeout, RETCODE *err )
  ;
      assume  cs:RTKERNEL_TEXT
_os_mbox_receive        proc    far
      push    bp
      mov     bp,sp
      sub     sp,4
  ;
  ;   {
  ;       void  *msg;
  ;
  ;           if ( pecb       == NULL)
  ;
      cmp     dword ptr [bp+6],large 0
      jne     short @29@86
  ;
  ;           {
  ;                   *err    =       OS_NULL_PECB;
  ;
      les     bx,dword ptr [bp+14]
```

```
    mov     word ptr es:[bx],81
  ;
  ;                 return NULL;
  ;
    xor     dx,dx
    xor     ax,ax
    leave
    ret
@29@86:
  ;
  ;         }
  ;
  ;     DISABLE();
  ;
    cli
  ;
  ;     if ( (msg = pecb->os_event_ptr) != NULL )
  ;
    les     bx,dword ptr [bp+6]
    mov     eax,dword ptr es:[bx+2]
    mov     dword ptr [bp-4],eax
    cmp     eax,large 0
    je      short @29@226
  ;
  ;         {    // See if there is already a message
  ;         pecb->os_event_ptr = NULL;              // Clear the mailbox
  ;
    mov     dword ptr es:[bx+2],large 0
  ;
  ;         ENABLE();
  ;
    sti
  ;
  ;         *err = OS_NO_ERR;
  ;
    les     bx,dword ptr [bp+14]
    mov     word ptr es:[bx],0
  ;
  ;     }
  ;
    jmp     @29@562
@29@226:
  ;
  ;         else
  ;         {
  ;         os_tcb_cur->pecb                = pecb;         // Store pointer to event control block in TCB
  ;
    les     bx,dword ptr DGROUP:_os_tcb_cur
    mov     eax,dword ptr [bp+6]
    mov     dword ptr es:[bx+17],eax
  ;
  ;                 os_tcb_ins_evnlist ( os_tcb_cur );
  ;
    push    word ptr DGROUP:_os_tcb_cur+2
    push    bx
    call    far ptr os_tcb_ins_evnlist
    add     sp,4
  ;
  ;                 os_tcb_del_rdylist ( os_tcb_cur );
  ;
    push    dword ptr DGROUP:_os_tcb_cur
    call    far ptr os_tcb_del_rdylist
    add     sp,4
  ;
  ;         os_tcb_cur->task_stat           = OS_TASK_MBOX; // Message not available, task will pend
  ;
    les     bx,dword ptr DGROUP:_os_tcb_cur
    mov     word ptr es:[bx+8],3
  ;
  ;         os_tcb_cur->dly_time            = timeout;              // Load timeout in TCB
  ;
    mov     eax,dword ptr [bp+10]
    mov     dword ptr es:[bx+13],eax
```

```
;
;               os_tcb_cur->timeout_flag        = 0;
;
    and     byte ptr es:[bx+21],254
;
;               if ( timeout )
;
    cmp     dword ptr [bp+10],large 0
    je      short @29@282
;
;                       os_tcb_ins_dlylist ( os_tcb_cur );
;
    push    word ptr DGROUP:_os_tcb_cur+2
    push    bx
    call    far ptr os_tcb_ins_dlylist
    add     sp,4
@29@282:
;
;
;           os_sched();                             // Find next highest priority task ready to run
;
    push    cs
    call    near ptr _os_sched
;
;               DISABLE();
;
    cli
;
;               msg                 = pecb->os_event_ptr;   // Message received
;
    les     bx,dword ptr [bp+6]
    mov     eax,dword ptr es:[bx+2]
    mov     dword ptr [bp-4],eax
;
;               pecb->os_event_ptr    = NULL;                           // Clear the mailbox
;
    mov     dword ptr es:[bx+2],large 0
;
;               ENABLE();
;
    sti
;
;               if ( os_tcb_cur->timeout_flag )
;
    les     bx,dword ptr DGROUP:_os_tcb_cur
    mov     al,byte ptr es:[bx+21]
    and     ax,1
    or      ax,ax
    je      short @29@450
;
;                       *err    = OS_TIMEOUT;
;
    les     bx,dword ptr [bp+14]
    mov     word ptr es:[bx],10
    jmp     short @29@562
@29@450:
;
;               else
;               {
;                       DISABLE ( );
;
    cli
;
;                       os_tcb_del_dlylist ( os_tcb_cur );
;
    push    dword ptr DGROUP:_os_tcb_cur
    call    far ptr os_tcb_del_dlylist
    add     sp,4
;
;                       ENABLE ( );
;
    sti
;
```

```
  ;                       *err    = OS_NO_ERR;
  ;
      les     bx,dword ptr [bp+14]
      mov     word ptr es:[bx],0
@29@562:
  ;
  ;                   }
  ;       }
  ;       return msg;                             // Return the message received (or NULL)
  ;
      mov     dx,word ptr [bp-2]
      mov     ax,word ptr [bp-4]
  ;
  ;   }
  ;
      leave
      ret
_os_mbox_receive        endp
  ;
  ;   RETCODE os_mbox_send ( OS_ECB *pecb, void *msg )
  ;
      assume  cs:RTKERNEL_TEXT
_os_mbox_send   proc    far
      push    bp
      mov     bp,sp
  ;
  ;   {
  ;           if ( pecb       == NULL)
  ;
      cmp     dword ptr [bp+6],large 0
      jne     short @30@86
  ;
  ;                   return  OS_NULL_PECB;
  ;
      mov     ax,81
      pop     bp
      ret
@30@86:
  ;
  ;
  ;       DISABLE();
  ;
      cli
  ;
  ;       if ( pecb->os_event_ptr != NULL )
  ;
      les     bx,dword ptr [bp+6]
      cmp     dword ptr es:[bx+2],large 0
      je      short @30@198
  ;
  ;           {       // Make sure mailbox doesn't already contain a msg
  ;           ENABLE();
  ;
      sti
  ;
  ;           return ( OS_MBOX_FULL );
  ;
      mov     ax,20
      pop     bp
      ret
@30@198:
  ;
  ;       }
  ;           else
  ;           {
  ;           pecb->os_event_ptr = msg;               // Place message in mailbox
  ;
      les     bx,dword ptr [bp+6]
      mov     eax,dword ptr [bp+10]
      mov     dword ptr es:[bx+2],eax
  ;
  ;                   os_tcb_purge_evnlist ( pecb );
  ;
```

```
        push    word ptr [bp+8]
        push    bx
        call    far ptr os_tcb_purge_evnlist
        add     sp,4
    ;
    ;               os_sched ( );   // Find highest priority task ready to run
    ;
        push    cs
        call    near ptr _os_sched
    ;
    ;           return OS_NO_ERR;
    ;
        xor     ax,ax
    ;
    ;       }
    ;   }
    ;
        pop     bp
        ret
_os_mbox_send   endp
    ;
    ;   OS_ECB *os_q_create ( WORD qsize, BYTE elem_size )
    ;
        assume  cs:RTKERNEL_TEXT
_os_q_create    proc    far
        push    bp
        mov     bp,sp
        sub     sp,12
        push    si
        mov     si,word ptr [bp+6]
    ;
    ;   {
    ;   // the queue size is qsize-1
    ;       OS_ECB              *pecb;
    ;       OS_QCB              *pq;
    ;           void far        *start;
    ;
    ;           if ( (start=farmalloc( (qsize+1)*elem_size ))==NULL)
    ;
        mov     al,byte ptr [bp+8]
        mov     ah,0
        push    ax
        mov     ax,si
        inc     ax
        pop     dx
        imul    dx
        movzx   eax,ax
        push    eax
        call    far ptr _farmalloc
        add     sp,4
        mov     word ptr [bp-10],dx
        mov     word ptr [bp-12],ax
        or      ax,dx
        jne     short @31@86
    ;
    ;               return NULL;
    ;
        xor     dx,dx
        xor     ax,ax
        pop     si
        leave
        ret
@31@86:
    ;
    ;
    ;       DISABLE();
    ;
        cli
    ;
    ;       pecb = os_ecb_free_list;            // Get next free event control block
    ;
        mov     eax,dword ptr DGROUP:os_ecb_free_list
        mov     dword ptr [bp-4],eax
```

```
  ;
  ;       if ( os_ecb_free_list != NULL )     // See if pool of free ECB pool was empty
  ;
      cmp     dword ptr DGROUP:os_ecb_free_list,large 0
      je      short @31@170
  ;
  ;           os_ecb_free_list = (OS_ECB *)os_ecb_free_list->os_event_ptr;
  ;
      les     bx,dword ptr DGROUP:os_ecb_free_list
      mov     eax,dword ptr es:[bx+2]
      mov     dword ptr DGROUP:os_ecb_free_list,eax
@31@170:
  ;
  ;       ENABLE();
  ;
      sti
  ;
  ;       if ( pecb != NULL )
  ;
      cmp     dword ptr [bp-4],large 0
      jne short       @@2
      jmp     @31@506
@@2:
  ;
  ;           {               // See if we have an event control block
  ;           DISABLE();                  // Get a free queue control block
  ;
      cli
  ;
  ;           pq = os_qcb_free_list;
  ;
      mov     eax,dword ptr DGROUP:os_qcb_free_list
      mov     dword ptr [bp-8],eax
  ;
  ;           if ( os_qcb_free_list != NULL )
  ;
      cmp     dword ptr DGROUP:os_qcb_free_list,large 0
      je      short @31@310
  ;
  ;               os_qcb_free_list = os_qcb_free_list->os_qcb_ptr;
  ;
      les     bx,dword ptr DGROUP:os_qcb_free_list
      mov     eax,dword ptr es:[bx]
      mov     dword ptr DGROUP:os_qcb_free_list,eax
@31@310:
  ;
  ;           ENABLE();
  ;
      sti
  ;
  ;               if ( pq != NULL )   // See if we were able to get a queue control block
  ;
      cmp     dword ptr [bp-8],large 0
      je      short @31@394
  ;
  ;                   {
  ;                       pecb->os_event_ptr      = pq;
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr [bp-8]
      mov     dword ptr es:[bx+2],eax
  ;
  ;               pq->qstart                  = start;       // Yes, initialize the queue
  ;
      les     bx,dword ptr [bp-8]
      mov     eax,dword ptr [bp-12]
      mov     dword ptr es:[bx+4],eax
  ;
  ;               pq->tail_offset     = pq->head_offset = pq->entries = 0;
  ;
      xor     ax,ax
      mov     word ptr es:[bx+13],ax
      mov     byte ptr es:[bx+9],al
```

```
        mov     byte ptr es:[bx+8],al
    ;
    ;                       pq->qsize                       = qsize+1;
    ;
        mov     ax,si
        inc     ax
        mov     word ptr es:[bx+11],ax
    ;
    ;               pq->elem_size               = elem_size;
    ;
        mov     al,byte ptr [bp+8]
        mov     byte ptr es:[bx+10],al
    ;
    ;           }
    ;
        jmp     short @31@506
@31@394:
    ;
    ;                   else
    ;                   {                           // No,  since we couldn't get a queue control block
    ;               DISABLE();                   // Return event control block on error
    ;
        cli
    ;
    ;               pecb->os_event_ptr = (void *)os_ecb_free_list;
    ;
        les     bx,dword ptr [bp-4]
        mov     eax,dword ptr DGROUP:os_ecb_free_list
        mov     dword ptr es:[bx+2],eax
    ;
    ;               os_ecb_free_list   = pecb;
    ;
        mov     eax,dword ptr [bp-4]
        mov     dword ptr DGROUP:os_ecb_free_list,eax
    ;
    ;               ENABLE();
    ;
        sti
    ;
    ;               pecb = NULL;
    ;
        mov     dword ptr [bp-4],large 0
@31@506:
    ;
    ;           }
    ;       }
    ;       return ( pecb );
    ;
        mov     dx,word ptr [bp-2]
        mov     ax,word ptr [bp-4]
    ;
    ;   }
    ;
        pop     si
        leave
        ret
_os_q_create    endp
    ;
    ;   void *os_q_read ( OS_ECB *pecb, TIME timeout, RETCODE *err )
    ;
        assume  cs:RTKERNEL_TEXT
_os_q_read      proc    far
        push    bp
        mov     bp,sp
        sub     sp,10
    ;
    ;   {
    ;       void  *msg;
    ;       OS_QCB  *pq = pecb->os_event_ptr;           // Point at queue control block
    ;
        les     bx,dword ptr [bp+6]
        mov     eax,dword ptr es:[bx+2]
        mov     dword ptr [bp-8],eax
```

```
;
;
;       DISABLE();
;
        cli
;
;       if ( pq->entries )                  // see if any messages in the queue
;
        les     bx,dword ptr [bp-8]
        cmp     word ptr es:[bx+13],0
        je      short @32@198
;
;           {
;           pq->entries--;                  // update the number of entries in the queue
;
        dec     word ptr es:[bx+13]
;
;           msg =  (BYTE*) (pq->qstart) + pq->head_offset*pq->elem_size;    // yes, extract oldest message from the queue
;
        mov     al,byte ptr es:[bx+9]
        mov     byte ptr [bp-9],al
        mov     ah,0
        mov     dl,byte ptr es:[bx+10]
        mov     dh,0
        imul    dx
        mov     dx,word ptr es:[bx+6]
        mov     bx,word ptr es:[bx+4]
        add     bx,ax
        mov     word ptr [bp-2],dx
        mov     word ptr [bp-4],bx
;
;               pq->head_offset = ++pq->head_offset % pq->qsize;
;
        mov     al,byte ptr [bp-9]
        inc     al
        mov     bx,word ptr [bp-8]
        mov     byte ptr es:[bx+9],al
        mov     ah,0
        xor     dx,dx
        div     word ptr es:[bx+11]
        mov     byte ptr es:[bx+9],dl
;
;           ENABLE();
;
        sti
;
;           *err = OS_NO_ERR;
;
        les     bx,dword ptr [bp+14]
        mov     word ptr es:[bx],0
;
;       }
;
        jmp     @32@534
@32@198:
;
;           else
;           {
;           os_tcb_cur->pecb                = pecb;       // store pointer to event control block in TCB
;
        les     bx,dword ptr DGROUP:_os_tcb_cur
        mov     eax,dword ptr [bp+6]
        mov     dword ptr es:[bx+17],eax
;
;               os_tcb_ins_evnlist ( os_tcb_cur );
;
        push    word ptr DGROUP:_os_tcb_cur+2
        push    bx
        call    far ptr os_tcb_ins_evnlist
        add     sp,4
;
;               os_tcb_del_rdylist ( os_tcb_cur );
;
```

```
    push    dword ptr DGROUP:_os_tcb_cur
    call    far ptr os_tcb_del_rdylist
    add     sp,4
  ;
  ;         os_tcb_cur->task_stat   = OS_TASK_Q;    // task will have to pend for a message to be posted
  ;
    les     bx,dword ptr DGROUP:_os_tcb_cur
    mov     word ptr es:[bx+8],4
  ;
  ;         os_tcb_cur->dly_time    = timeout;      // load timeout into TCB
  ;
    mov     eax,dword ptr [bp+10]
    mov     dword ptr es:[bx+13],eax
  ;
  ;             os_tcb_cur->timeout_flag= 0;
  ;
    and     byte ptr es:[bx+21],254
  ;
  ;             if ( timeout )                                  // timeout?...
  ;
    cmp     dword ptr [bp+10],large 0
    je      short @32@254
  ;
  ;                     os_tcb_ins_dlylist ( os_tcb_cur );      // insert in delay list
  ;
    push    word ptr DGROUP:_os_tcb_cur+2
    push    bx
    call    far ptr os_tcb_ins_dlylist
    add     sp,4
@32@254:
  ;
  ;         os_sched();                         // Find next highest priority task ready to run
  ;
    push    cs
    call    near ptr _os_sched
  ;
  ;         DISABLE();
  ;
    cli
  ;
  ;         pq->entries--;                                  // update the number of entries in the queue
  ;
    les     bx,dword ptr [bp-8]
    dec     word ptr es:[bx+13]
  ;
  ;         msg =  (BYTE*) (pq->qstart) + pq->head_offset*pq->elem_size;// message received, extract oldest message from the
queue
  ;
    mov     al,byte ptr es:[bx+9]
    mov     byte ptr [bp-9],al
    mov     ah,0
    mov     dl,byte ptr es:[bx+10]
    mov     dh,0
    imul    dx
    mov     dx,word ptr es:[bx+6]
    mov     bx,word ptr es:[bx+4]
    add     bx,ax
    mov     word ptr [bp-2],dx
    mov     word ptr [bp-4],bx
  ;
  ;             pq->head_offset = ++pq->head_offset % pq->qsize;
  ;
    mov     al,byte ptr [bp-9]
    inc     al
    mov     bx,word ptr [bp-8]
    mov     byte ptr es:[bx+9],al
    mov     ah,0
    xor     dx,dx
    div     word ptr es:[bx+11]
    mov     byte ptr es:[bx+9],dl
  ;
  ;         ENABLE();
  ;
```

```
        sti
    ;
    ;               if ( os_tcb_cur->timeout_flag )
    ;
        les     bx,dword ptr DGROUP:_os_tcb_cur
        mov     al,byte ptr es:[bx+21]
        and     ax,1
        or      ax,ax
        je      short @32@422
    ;
    ;                   *err    = OS_TIMEOUT;
    ;
        les     bx,dword ptr [bp+14]
        mov     word ptr es:[bx],10
        jmp     short @32@534
@32@422:
    ;
    ;               else
    ;               {
    ;                   DISABLE ( );
    ;
        cli
    ;
    ;                   os_tcb_del_dlylist ( os_tcb_cur );
    ;
        push    dword ptr DGROUP:_os_tcb_cur
        call    far ptr os_tcb_del_dlylist
        add     sp,4
    ;
    ;                   ENABLE ( );
    ;
        sti
    ;
    ;               *err    = OS_NO_ERR;
    ;
        les     bx,dword ptr [bp+14]
        mov     word ptr es:[bx],0
@32@534:
    ;
    ;               }
    ;       }                               // Return message received (or NULL)
    ;       return (msg);
    ;
        mov     dx,word ptr [bp-2]
        mov     ax,word ptr [bp-4]
    ;
    ;   }
    ;
        leave
        ret
_os_q_read      endp
    ;
    ;   BYTE os_q_write ( OS_ECB *pecb, void *msg )
    ;
        assume  cs:RTKERNEL_TEXT
_os_q_write     proc    far
        push    bp
        mov     bp,sp
        sub     sp,6
    ;
    ;   {
    ;       OS_QCB  *pq = pecb->os_event_ptr;           // point to queue control block
    ;
        les     bx,dword ptr [bp+6]
        mov     eax,dword ptr es:[bx+2]
        mov     dword ptr [bp-4],eax
    ;
    ;
    ;       DISABLE();
    ;
        cli
    ;
    ;       if ( pq->entries < pq->qsize )      // make sure that queue is not full
```

```
;
    les     bx,dword ptr [bp-4]
    mov     ax,word ptr es:[bx+13]
    cmp     ax,word ptr es:[bx+11]
    jae     short @33@142
;
;           {
;               // insert message into queue
;               _fmemcpy ( (BYTE *)pq->qstart + pq->elem_size*pq->tail_offset, msg, pq->elem_size );
;
    mov     al,byte ptr es:[bx+10]
    mov     ah,0
    mov     word ptr [bp-6],ax
    push    ax
    push    dword ptr [bp+10]
    mov     dl,byte ptr es:[bx+8]
    mov     dh,0
    imul    dx
    mov     dx,word ptr es:[bx+4]
    add     dx,ax
    push    word ptr es:[bx+6]
    push    dx
    call    far ptr __fmemcpy
    add     sp,10
;
;               pq->tail_offset = ++pq->tail_offset % pq->qsize;
;
    les     bx,dword ptr [bp-4]
    mov     al,byte ptr es:[bx+8]
    inc     al
    mov     byte ptr es:[bx+8],al
    mov     ah,0
    xor     dx,dx
    div     word ptr es:[bx+11]
    mov     byte ptr es:[bx+8],dl
;
;               pq->entries++;                          // update the number of entries in the queue
;
    inc     word ptr es:[bx+13]
;
;               os_tcb_del_evnlist ( pecb );
;
    push    dword ptr [bp+6]
    call    far ptr os_tcb_del_evnlist
    add     sp,4
;
;               os_sched();                             // find highest priority task ready to run
;
    push    cs
    call    near ptr _os_sched
;
;               return OS_NO_ERR;
;
    mov     al,0
    leave
    ret
@33@142:
;
;           }
;           else
;           {
;           ENABLE ( );
;
    sti
;
;           return ( OS_Q_FULL );
;
    mov     al,30
;
;       }
;   }
;
    leave
```

```
        ret
_os_q_write     endp
  ;
  ;     void os_tcb_ins_rdylist ( OS_TCB *ptcb )
  ;
        assume  cs:RTKERNEL_TEXT
os_tcb_ins_rdylist      proc    far
        push    bp
        mov     bp,sp
  ;
  ;     {
  ;             WORD    prio    =       ptcb->prio;
  ;
        les     bx,dword ptr [bp+6]
        mov     al,byte ptr es:[bx+10]
        cbw
        mov     cx,ax
  ;
  ;
  ;             ptcb->task_stat         = OS_TASK_RDY;
  ;
        mov     word ptr es:[bx+8],1
  ;
  ;
  ;             if ( os_tcb_run == NULL )
  ;
        cmp     dword ptr DGROUP:_os_tcb_run,large 0
        jne     short @34@86
  ;
  ;             {
  ;                   os_tcb_run = ptcb->next_rdy     = ptcb->prev_rdy = ptcb;
  ;
        mov     eax,dword ptr [bp+6]
        mov     dword ptr es:[bx+30],eax
        mov     dword ptr es:[bx+26],eax
        mov     dword ptr DGROUP:_os_tcb_run,eax
  ;
  ;                   os_tcb_current_prio             =       prio;
  ;
        mov     word ptr DGROUP:os_tcb_current_prio,cx
  ;
  ;                   os_tcb_curprio [ prio ] = os_tcb_run;
  ;
        mov     bx,cx
        shl     bx,2
        mov     ax,word ptr DGROUP:_os_tcb_run+2
        mov     dx,word ptr DGROUP:_os_tcb_run
        mov     word ptr DGROUP:os_tcb_curprio[bx+2],ax
        mov     word ptr DGROUP:os_tcb_curprio[bx],dx
  ;
  ;             }
  ;
        pop     bp
        ret
@34@86:
  ;
  ;             else
  ;                   if ( os_tcb_curprio [ prio ] == NULL )
  ;
        mov     bx,cx
        shl     bx,2
        mov     ax,word ptr DGROUP:os_tcb_curprio[bx]
        or      ax,word ptr DGROUP:os_tcb_curprio[bx+2]
        jne     short @34@142
  ;
  ;                         os_tcb_curprio [ prio ] = ptcb->next_rdy = ptcb->prev_rdy = ptcb;
  ;
        les     bx,dword ptr [bp+6]
        mov     ax,word ptr [bp+8]
        mov     dx,word ptr [bp+6]
        mov     word ptr es:[bx+32],ax
        mov     word ptr es:[bx+30],dx
        mov     word ptr es:[bx+28],ax
```

```
        mov     word ptr es:[bx+26],dx
        mov     bx,cx
        shl     bx,2
        mov     word ptr DGROUP:os_tcb_curprio[bx+2],ax
        mov     word ptr DGROUP:os_tcb_curprio[bx],dx
        pop     bp
        ret
@34@142:
  ;
  ;                 else
  ;                 {
  ;                       os_tcb_curprio [ prio ]->prev_rdy->next_rdy     = ptcb;
  ;
        mov     bx,cx
        shl     bx,2
        les     bx,dword ptr DGROUP:os_tcb_curprio[bx]
        les     bx,dword ptr es:[bx+30]
        mov     ax,word ptr [bp+8]
        mov     dx,word ptr [bp+6]
        mov     word ptr es:[bx+28],ax
        mov     word ptr es:[bx+26],dx
  ;
  ;                       ptcb->prev_rdy                                          = os_tcb_curprio [ prio ]->prev_rdy;
  ;
        mov     bx,cx
        shl     bx,2
        les     bx,dword ptr DGROUP:os_tcb_curprio[bx]
        mov     ax,word ptr es:[bx+32]
        mov     dx,word ptr es:[bx+30]
        les     bx,dword ptr [bp+6]
        mov     word ptr es:[bx+32],ax
        mov     word ptr es:[bx+30],dx
  ;
  ;                       os_tcb_curprio [ prio ]->prev_rdy                       = ptcb;
  ;
        mov     bx,cx
        shl     bx,2
        les     bx,dword ptr DGROUP:os_tcb_curprio[bx]
        mov     ax,word ptr [bp+8]
        mov     dx,word ptr [bp+6]
        mov     word ptr es:[bx+32],ax
        mov     word ptr es:[bx+30],dx
  ;
  ;                       ptcb->next_rdy                                          = os_tcb_curprio [ prio ];
  ;
        mov     bx,cx
        shl     bx,2
        mov     ax,word ptr DGROUP:os_tcb_curprio[bx+2]
        mov     dx,word ptr DGROUP:os_tcb_curprio[bx]
        les     bx,dword ptr [bp+6]
        mov     word ptr es:[bx+28],ax
        mov     word ptr es:[bx+26],dx
  ;
  ;   }
  ;
        pop     bp
        ret
os_tcb_ins_rdylist      endp
  ;
  ;   void os_tcb_del_rdylist ( OS_TCB *ptcb )
  ;
        assume  cs:RTKERNEL_TEXT
os_tcb_del_rdylist      proc    far
        push    bp
        mov     bp,sp
  ;
  ;   {
  ;           WORD    prio    =       ptcb->prio;
  ;
        les     bx,dword ptr [bp+6]
        mov     al,byte ptr es:[bx+10]
        cbw
        mov     dx,ax
```

```
;
;
;       if ( ptcb == os_tcb_run )
;
    mov     eax,dword ptr [bp+6]
    cmp     eax,dword ptr DGROUP:_os_tcb_run
    je short        @@3
    jmp     @35@338
@@3:
;
;       {
;           if ( os_tcb_cur->next_rdy != os_tcb_run )
;
    les     bx,dword ptr DGROUP:_os_tcb_cur
    mov     eax,dword ptr es:[bx+26]
    cmp     eax,dword ptr DGROUP:_os_tcb_run
    je      short @35@114
;
;           {       //change os_tcb_run ( round robin )
;               os_tcb_run                          = os_tcb_run->next_rdy;
;
    les     bx,dword ptr DGROUP:_os_tcb_run
    mov     eax,dword ptr es:[bx+26]
    mov     dword ptr DGROUP:_os_tcb_run,eax
;
;               os_tcb_run->task_stat           = OS_TASK_RUN;
;
    les     bx,dword ptr DGROUP:_os_tcb_run
    mov     word ptr es:[bx+8],0
;
;               os_tcb_curprio [ prio ]         = os_tcb_run;
;
    mov     bx,dx
    shl     bx,2
    mov     ax,word ptr DGROUP:_os_tcb_run+2
    mov     dx,word ptr DGROUP:_os_tcb_run
    mov     word ptr DGROUP:os_tcb_curprio[bx+2],ax
    mov     word ptr DGROUP:os_tcb_curprio[bx],dx
;
;               ptcb->prev_rdy->next_rdy        = ptcb->next_rdy;
;
    les     bx,dword ptr [bp+6]
    mov     ax,word ptr es:[bx+28]
    mov     dx,word ptr es:[bx+26]
    les     bx,dword ptr es:[bx+30]
    mov     word ptr es:[bx+28],ax
    mov     word ptr es:[bx+26],dx
;
;               ptcb->next_rdy->prev_rdy        = ptcb->prev_rdy;
;
    les     bx,dword ptr [bp+6]
    mov     ax,word ptr es:[bx+32]
    mov     dx,word ptr es:[bx+30]
    les     bx,dword ptr es:[bx+26]
    mov     word ptr es:[bx+32],ax
    mov     word ptr es:[bx+30],dx
;
;           }
;
    pop     bp
    ret
@35@114:
;
;           else    //was the only task in this priority
;           {
;               int     i;
;
;               os_tcb_curprio [ prio ] = NULL;
;
    mov     bx,dx
    shl     bx,2
    mov     word ptr DGROUP:os_tcb_curprio[bx+2],0
    mov     word ptr DGROUP:os_tcb_curprio[bx],0
```

```
  ;
  ;
  ;                     //search next highest priority task ready tu run
  ;                     for ( i=0 ; i < OS_LO_PRIO; i++ )
  ;
      xor     cx,cx
      mov     dx,offset DGROUP:os_tcb_curprio
@35@170:
  ;
  ;                             if ( os_tcb_curprio [ i ] != NULL )
  ;
      mov     bx,dx
      mov     ax,word ptr [bx]
      or      ax,word ptr [bx+2]
      je      short @35@226
  ;
  ;                             {
  ;                                   os_tcb_run                      = os_tcb_curprio [ i ];
  ;
      mov     bx,cx
      shl     bx,2
      mov     ax,word ptr DGROUP:os_tcb_curprio[bx+2]
      mov     dx,word ptr DGROUP:os_tcb_curprio[bx]
      mov     word ptr DGROUP:_os_tcb_run+2,ax
      mov     word ptr DGROUP:_os_tcb_run,dx
  ;
  ;                                   os_tcb_current_prio           = i;
  ;
      mov     word ptr DGROUP:os_tcb_current_prio,cx
  ;
  ;                                   os_tcb_run->task_stat   = OS_TASK_RUN;
  ;
      les     bx,dword ptr DGROUP:_os_tcb_run
      mov     word ptr es:[bx+8],0
  ;
  ;                                   break;
  ;
      pop     bp
      ret
@35@226:
      add     dx,4
      inc     cx
      cmp     dx,offset DGROUP:os_tcb_curprio+16
      jne     short @35@170
  ;
  ;                             }
  ;                 }
  ;           }
  ;
      pop     bp
      ret
@35@338:
  ;
  ;           else
  ;           {
  ;                 if ( ptcb->next_rdy == ptcb )//was the only task in this priority?
  ;
      les     bx,dword ptr [bp+6]
      mov     eax,dword ptr es:[bx+26]
      cmp     eax,dword ptr [bp+6]
      jne     short @35@394
  ;
  ;                       os_tcb_curprio [ prio ] = NULL;
  ;
      mov     bx,dx
      shl     bx,2
      mov     word ptr DGROUP:os_tcb_curprio[bx+2],0
      mov     word ptr DGROUP:os_tcb_curprio[bx],0
      pop     bp
      ret
@35@394:
  ;
  ;                 else
```

```
  ;                 {
  ;                       ptcb->prev_rdy->next_rdy        = ptcb->next_rdy;
  ;
        les     bx,dword ptr [bp+6]
        mov     ax,word ptr es:[bx+28]
        mov     dx,word ptr es:[bx+26]
        les     bx,dword ptr es:[bx+30]
        mov     word ptr es:[bx+28],ax
        mov     word ptr es:[bx+26],dx
  ;
  ;                       ptcb->next_rdy->prev_rdy        = ptcb->prev_rdy;
  ;
        les     bx,dword ptr [bp+6]
        mov     ax,word ptr es:[bx+32]
        mov     dx,word ptr es:[bx+30]
        les     bx,dword ptr es:[bx+26]
        mov     word ptr es:[bx+32],ax
        mov     word ptr es:[bx+30],dx
  ;
  ;                 }
  ;           }
  ;   }
  ;
        pop     bp
        ret
os_tcb_del_rdylist      endp
  ;
  ;   void os_tcb_ins_evnlist ( OS_TCB *ptcb )
  ;
        assume  cs:RTKERNEL_TEXT
os_tcb_ins_evnlist      proc    far
        push    bp
        mov     bp,sp
        sub     sp,8
  ;
  ;   {
  ;           OS_ECB  *pecb   =       ptcb->pecb;
  ;
        les     bx,dword ptr [bp+6]
        mov     eax,dword ptr es:[bx+17]
        mov     dword ptr [bp-4],eax
  ;
  ;
  ;           if ( pecb->os_tcb_blk_task == NULL ||
  ;
  ;
  ;                                       ptcb->prio < pecb->os_tcb_blk_task->prio )
  ;
        les     bx,dword ptr [bp-4]
        cmp     dword ptr es:[bx+6],large 0
        je      short @36@86
        les     bx,dword ptr [bp+6]
        mov     al,byte ptr es:[bx+10]
        mov     dl,al
        les     bx,dword ptr [bp-4]
        les     bx,dword ptr es:[bx+6]
        cmp     al,byte ptr es:[bx+10]
        jge     short @36@114
@36@86:
  ;
  ;           {
  ;                 ptcb->next_blk                  = pecb->os_tcb_blk_task;
  ;
        les     bx,dword ptr [bp-4]
        mov     eax,dword ptr es:[bx+6]
        les     bx,dword ptr [bp+6]
        mov     dword ptr es:[bx+34],eax
  ;
  ;                 pecb->os_tcb_blk_task   = ptcb;
  ;
        les     bx,dword ptr [bp-4]
        mov     eax,dword ptr [bp+6]
        mov     dword ptr es:[bx+6],eax
```

```
  ;
  ;         }
  ;
      leave
      ret
@36@114:
  ;
  ;         else
  ;         {
  ;               OS_TCB  *p;
  ;
  ;               for ( p = pecb->os_tcb_blk_task; p != NULL; p=p->next_blk )
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr es:[bx+6]
      mov     dword ptr [bp-8],eax
      jmp     short @36@282
@36@170:
  ;
  ;                       if ( ptcb->prio < p->prio || p->next_blk == NULL )
  ;
      les     bx,dword ptr [bp-8]
      cmp     dl,byte ptr es:[bx+10]
      jl      short @36@226
      cmp     dword ptr es:[bx+34],large 0
      jne     short @36@254
@36@226:
  ;
  ;                       {
  ;                               ptcb->next_blk          = p->next_blk;
  ;
      les     bx,dword ptr [bp-8]
      mov     eax,dword ptr es:[bx+34]
      les     bx,dword ptr [bp+6]
      mov     dword ptr es:[bx+34],eax
  ;
  ;                               p->next_blk                     = ptcb;
  ;
      les     bx,dword ptr [bp-8]
      mov     eax,dword ptr [bp+6]
      mov     dword ptr es:[bx+34],eax
  ;
  ;                               break;
  ;
      leave
      ret
@36@254:
      les     bx,dword ptr [bp-8]
      mov     eax,dword ptr es:[bx+34]
      mov     dword ptr [bp-8],eax
@36@282:
      cmp     dword ptr [bp-8],large 0
      jne     short @36@170
  ;
  ;                       }
  ;         }
  ;   }
  ;
      leave
      ret
os_tcb_ins_evnlist      endp
  ;
  ;   void os_tcb_purge_evnlist ( OS_ECB *pecb )
  ;
      assume  cs:RTKERNEL_TEXT
os_tcb_purge_evnlist    proc    far
      push    bp
      mov     bp,sp
      sub     sp,4
  ;
  ;   {
  ;         OS_TCB  *ptcb   = pecb->os_tcb_blk_task;
  ;
```

```
        les     bx,dword ptr [bp+6]
        mov     eax,dword ptr es:[bx+6]
        mov     dword ptr [bp-4],eax
        jmp     short @37@114
@37@58:
  ;
  ;
  ;         for (; ptcb != NULL; ptcb=ptcb->next_blk )
  ;         {
  ;               ptcb->pecb      = NULL;
  ;
        les     bx,dword ptr [bp-4]
        mov     dword ptr es:[bx+17],large 0
  ;
  ;               os_tcb_ins_rdylist ( ptcb );
  ;
        push    word ptr [bp-2]
        push    bx
        push    cs
        call    near ptr os_tcb_ins_rdylist
        add     sp,4
        les     bx,dword ptr [bp-4]
        mov     eax,dword ptr es:[bx+34]
        mov     dword ptr [bp-4],eax
@37@114:
        cmp     dword ptr [bp-4],large 0
        jne     short @37@58
  ;
  ;         }
  ;
  ;         pecb->os_tcb_blk_task   = NULL;
  ;
        les     bx,dword ptr [bp+6]
        mov     dword ptr es:[bx+6],large 0
  ;
  ;   }
  ;
        leave
        ret
os_tcb_purge_evnlist    endp
  ;
  ;   void os_tcb_del_evnlist ( OS_ECB *pecb )
  ;
        assume  cs:RTKERNEL_TEXT
os_tcb_del_evnlist      proc    far
        push    bp
        mov     bp,sp
  ;
  ;   {
  ;         if ( pecb->os_tcb_blk_task != NULL )
  ;
        les     bx,dword ptr [bp+6]
        cmp     dword ptr es:[bx+6],large 0
        je      short @38@86
  ;
  ;         {
  ;               os_tcb_ins_rdylist ( pecb->os_tcb_blk_task );
  ;
        push    dword ptr es:[bx+6]
        push    cs
        call    near ptr os_tcb_ins_rdylist
        add     sp,4
  ;
  ;               pecb->os_tcb_blk_task           = pecb->os_tcb_blk_task->next_blk;
  ;
        les     bx,dword ptr [bp+6]
        les     bx,dword ptr es:[bx+6]
        mov     ax,word ptr es:[bx+36]
        mov     dx,word ptr es:[bx+34]
        les     bx,dword ptr [bp+6]
        mov     word ptr es:[bx+8],ax
        mov     word ptr es:[bx+6],dx
@38@86:
```

```
  ;
  ;         }
  ;   }
  ;
      pop     bp
      ret
os_tcb_del_evnlist      endp
  ;
  ;   void os_tcb_ins_dlylist ( OS_TCB *ptcb )
  ;
      assume  cs:RTKERNEL_TEXT
os_tcb_ins_dlylist      proc    far
      push    bp
      mov     bp,sp
      sub     sp,8
  ;
  ;   {
  ;         if ( os_tcb_dlylist == NULL )
  ;
      cmp     dword ptr DGROUP:os_tcb_dlylist,large 0
      jne     short @39@86
  ;
  ;         {
  ;               ptcb->next_dly                  = NULL;
  ;
      les     bx,dword ptr [bp+6]
      mov     dword ptr es:[bx+38],large 0
  ;
  ;               os_tcb_dlylist                  = ptcb;
  ;
      mov     eax,dword ptr [bp+6]
      mov     dword ptr DGROUP:os_tcb_dlylist,eax
  ;
  ;         }
  ;
      leave
      ret
@39@86:
  ;
  ;         else if ( os_tcb_dlylist->dly_time > ptcb->dly_time )
  ;
      les     bx,dword ptr DGROUP:os_tcb_dlylist
      mov     eax,dword ptr es:[bx+13]
      les     bx,dword ptr [bp+6]
      cmp     eax,dword ptr es:[bx+13]
      jbe     short @39@142
  ;
  ;         {
  ;               ptcb->next_dly                  = os_tcb_dlylist;
  ;
      mov     eax,dword ptr DGROUP:os_tcb_dlylist
      mov     dword ptr es:[bx+38],eax
  ;
  ;               os_tcb_dlylist->dly_time        -= ptcb->dly_time;
  ;
      mov     eax,dword ptr es:[bx+13]
      les     bx,dword ptr DGROUP:os_tcb_dlylist
      sub     dword ptr es:[bx+13],eax
  ;
  ;               os_tcb_dlylist                  = ptcb;
  ;
      mov     eax,dword ptr [bp+6]
      mov     dword ptr DGROUP:os_tcb_dlylist,eax
  ;
  ;         }
  ;
      leave
      ret
@39@142:
  ;
  ;         else
  ;         {
  ;               OS_TCB  *p              = os_tcb_dlylist;
```

```
  ;
      mov     eax,dword ptr DGROUP:os_tcb_dlylist
      mov     dword ptr [bp-4],eax
  ;
  ;               TIME    time    = 0;
  ;
      mov     dword ptr [bp-8],large 0
      jmp     @39@310
@39@170:
  ;
  ;
  ;               for ( ; p != NULL; p=p->next_dly )
  ;               {
  ;                     time    += p->dly_time;
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr es:[bx+13]
      add     dword ptr [bp-8],eax
  ;
  ;                     if ( p->next_dly == NULL )
  ;
      cmp     dword ptr es:[bx+38],large 0
      jne     short @39@226
  ;
  ;                     {
  ;                           ptcb->dly_time  -= time;
  ;
      les     bx,dword ptr [bp+6]
      mov     eax,dword ptr [bp-8]
      sub     dword ptr es:[bx+13],eax
  ;
  ;                           p->next_dly             = ptcb;
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr [bp+6]
      mov     dword ptr es:[bx+38],eax
  ;
  ;                           ptcb->next_dly  = NULL;
  ;
      les     bx,dword ptr [bp+6]
      mov     dword ptr es:[bx+38],large 0
  ;
  ;                           break;
  ;
      leave
      ret
@39@226:
  ;
  ;                     }
  ;                     else if ( ( time + p->next_dly->dly_time ) > ptcb->dly_time )
  ;
      les     bx,dword ptr [bp-4]
      les     bx,dword ptr es:[bx+38]
      mov     eax,dword ptr [bp-8]
      add     eax,dword ptr es:[bx+13]
      les     bx,dword ptr [bp+6]
      cmp     eax,dword ptr es:[bx+13]
      jbe     short @39@282
  ;
  ;                     {
  ;                           ptcb->dly_time                  -= time;
  ;
      mov     eax,dword ptr [bp-8]
      sub     dword ptr es:[bx+13],eax
  ;
  ;                           p->next_dly->dly_time   -= ptcb->dly_time;
  ;
      mov     eax,dword ptr es:[bx+13]
      les     bx,dword ptr [bp-4]
      les     bx,dword ptr es:[bx+38]
      sub     dword ptr es:[bx+13],eax
  ;
  ;                           ptcb->next_dly                  = p->next_dly;
```

```
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr es:[bx+38]
      les     bx,dword ptr [bp+6]
      mov     dword ptr es:[bx+38],eax
  ;
  ;                               p->next_dly                     = ptcb;
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr [bp+6]
      mov     dword ptr es:[bx+38],eax
  ;
  ;                               break;
  ;
      leave
      ret
@39@282:
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr es:[bx+38]
      mov     dword ptr [bp-4],eax
@39@310:
      cmp     dword ptr [bp-4],large 0
      je short        @@4
      jmp     @39@170
@@4:
  ;
  ;                       }
  ;                   }
  ;           }
  ;   }
  ;
      leave
      ret
os_tcb_ins_dlylist      endp
  ;
  ;   void os_tcb_del_dlylist ( OS_TCB *ptcb )
  ;
      assume  cs:RTKERNEL_TEXT
os_tcb_del_dlylist      proc    far
      push    bp
      mov     bp,sp
      sub     sp,8
  ;
  ;   {
  ;           OS_TCB  *aux_ptcb       = os_tcb_dlylist;
  ;
      mov     eax,dword ptr DGROUP:os_tcb_dlylist
      mov     dword ptr [bp-4],eax
  ;
  ;
  ;           if ( aux_ptcb == ptcb )
  ;
      cmp     eax,dword ptr [bp+6]
      jne     short @40@226
  ;
  ;           {
  ;                   os_tcb_dlylist                          = aux_ptcb->next_dly;
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr es:[bx+38]
      mov     dword ptr DGROUP:os_tcb_dlylist,eax
  ;
  ;                   aux_ptcb->next_dly->dly_time    += aux_ptcb->dly_time;
  ;
      mov     eax,dword ptr es:[bx+13]
      les     bx,dword ptr es:[bx+38]
      add     dword ptr es:[bx+13],eax
  ;
  ;           }
  ;
      leave
      ret
@40@86:
```

```
  ;
  ;         else
  ;               for ( ; aux_ptcb != NULL; aux_ptcb=aux_ptcb->next_dly )
  ;                     if ( aux_ptcb->next_dly == ptcb )
  ;
      les     bx,dword ptr [bp-4]
      mov     eax,dword ptr es:[bx+38]
      mov     dword ptr [bp-8],eax
      cmp     eax,dword ptr [bp+6]
      jne     short @40@198
  ;
  ;                     {
  ;                           aux_ptcb->next_dly                  = ptcb->next_dly;
  ;
      les     bx,dword ptr [bp+6]
      mov     eax,dword ptr es:[bx+38]
      les     bx,dword ptr [bp-4]
      mov     dword ptr es:[bx+38],eax
  ;
  ;                           if ( ptcb->next_dly != NULL )
  ;
      les     bx,dword ptr [bp+6]
      cmp     dword ptr es:[bx+38],large 0
      je      short @40@170
  ;
  ;                                 ptcb->next_dly->dly_time        += ptcb->dly_time;
  ;
      mov     eax,dword ptr es:[bx+13]
      les     bx,dword ptr es:[bx+38]
      add     dword ptr es:[bx+13],eax
@40@170:
  ;
  ;                           ptcb->next_dly                      = NULL;
  ;
      les     bx,dword ptr [bp+6]
      mov     dword ptr es:[bx+38],large 0
  ;
  ;                           break;
  ;
      leave
      ret
@40@198:
      mov     eax,dword ptr [bp-8]
      mov     dword ptr [bp-4],eax
@40@226:
      cmp     dword ptr [bp-4],large 0
      jne     short @40@86
  ;
  ;                     }
  ;   }
  ;
      leave
      ret
os_tcb_del_dlylist      endp
  ;
  ;   void os_tcb_quit_evnlist ( OS_TCB *ptcb )
  ;
      assume  cs:RTKERNEL_TEXT
os_tcb_quit_evnlist     proc    far
      push    bp
      mov     bp,sp
      sub     sp,12
  ;
  ;   {
  ;         OS_ECB  *pecb           = ptcb->pecb;
  ;
      les     bx,dword ptr [bp+6]
      mov     eax,dword ptr es:[bx+17]
      mov     dword ptr [bp-4],eax
  ;
  ;         OS_TCB  *auxptcb        = pecb->os_tcb_blk_task;
  ;
      les     bx,dword ptr [bp-4]
```

```
        mov     eax,dword ptr es:[bx+6]
        mov     dword ptr [bp-8],eax
    ;
    ;
    ;           if ( auxptcb == ptcb )
    ;
        cmp     eax,dword ptr [bp+6]
        jne     short @41@170
    ;
    ;           {
    ;                   pecb->os_tcb_blk_task   =       ptcb->next_blk;
    ;
        les     bx,dword ptr [bp+6]
        mov     eax,dword ptr es:[bx+34]
        les     bx,dword ptr [bp-4]
        mov     dword ptr es:[bx+6],eax
    ;
    ;                   ptcb->next_blk                  =       NULL;
    ;
        les     bx,dword ptr [bp+6]
        mov     dword ptr es:[bx+34],large 0
    ;
    ;                   ptcb->pecb                              =       NULL;
    ;
        mov     dword ptr es:[bx+17],large 0
    ;
    ;                   return;
    ;
        leave
        ret
@41@86:
    ;
    ;           }
    ;
    ;           for (;  auxptcb != NULL; auxptcb = auxptcb->next_blk )
    ;                   if ( auxptcb->next_blk == ptcb )
    ;
        les     bx,dword ptr [bp-8]
        mov     eax,dword ptr es:[bx+34]
        mov     dword ptr [bp-12],eax
        cmp     eax,dword ptr [bp+6]
        jne     short @41@142
    ;
    ;                   {
    ;                           auxptcb->next_blk               = ptcb->next_blk;
    ;
        les     bx,dword ptr [bp+6]
        mov     eax,dword ptr es:[bx+34]
        les     bx,dword ptr [bp-8]
        mov     dword ptr es:[bx+34],eax
    ;
    ;                           ptcb->next_blk                  = NULL;
    ;
        les     bx,dword ptr [bp+6]
        mov     dword ptr es:[bx+34],large 0
    ;
    ;                           ptcb->pecb                              = NULL;
    ;
        mov     dword ptr es:[bx+17],large 0
    ;
    ;                           break;
    ;
        leave
        ret
@41@142:
        mov     eax,dword ptr [bp-12]
        mov     dword ptr [bp-8],eax
@41@170:
        cmp     dword ptr [bp-8],large 0
        jne     short @41@86
    ;
    ;                   }
    ;   }
```

```
  ;
      leave
      ret
os_tcb_quit_evnlist     endp
RTKERNEL_TEXT   ends
_BSS    segment word public use16 'BSS'
old_tick_isr    label   dword
      db      4 dup (?)
old_ss  label   word
      db      2 dup (?)
old_sp  label   word
      db      2 dup (?)
old_bp  label   word
      db      2 dup (?)
os_tcb_curprio  label   dword
      db      16 dup (?)
os_tcb_dlylist  label   dword
      db      4 dup (?)
os_tcb_task_tbl label   dword
      db      256 dup (?)
idle_tsk_id     label   word
      db      2 dup (?)
os_qcb_tbl      label   word
      db      75 dup (?)
os_ecb_tbl      label   word
      db      200 dup (?)
os_tcb_tbl      label   word
      db      2688 dup (?)
os_time label   word
      db      4 dup (?)
os_qcb_free_list        label   dword
      db      4 dup (?)
os_ecb_free_list        label   dword
      db      4 dup (?)
os_tcb_free_list        label   dword
      db      4 dup (?)
os_int_nesting  label   byte
      db      1 dup (?)
os_lock_nesting label   byte
      db      1 dup (?)
os_tcb_current_prio     label   word
      db      2 dup (?)
_count_ticks    label   word
      db      4 dup (?)
_os_tcb_run     label   dword
      db      4 dup (?)
_os_tcb_cur     label   dword
      db      4 dup (?)
_os_running     label   word
      db      2 dup (?)
_os_idle_ctr    label   word
      db      4 dup (?)
_os_ctx_sw_ctr  label   word
      db      2 dup (?)
_BSS    ends
_DATA   segment word public use16 'DATA'
s@      label   byte
_DATA   ends
RTKERNEL_TEXT   segment byte public use16 'CODE'
RTKERNEL_TEXT   ends
_old_tick_isr   equ     old_tick_isr
_os_dest_alltasks       equ     os_dest_alltasks
_os_tcb_quit_evnlist    equ     os_tcb_quit_evnlist
_os_tcb_del_dlylist     equ     os_tcb_del_dlylist
_os_tcb_ins_dlylist     equ     os_tcb_ins_dlylist
_os_tcb_del_evnlist     equ     os_tcb_del_evnlist
_os_tcb_purge_evnlist   equ     os_tcb_purge_evnlist
_os_tcb_ins_evnlist     equ     os_tcb_ins_evnlist
_os_tcb_del_rdylist     equ     os_tcb_del_rdylist
_os_tcb_ins_rdylist     equ     os_tcb_ins_rdylist
_os_task_idle   equ     os_task_idle
_old_ss equ     old_ss
_old_sp equ     old_sp
```

```
_old_bp equ     old_bp
_tick_size      equ     tick_size
_os_tcb_curprio equ     os_tcb_curprio
_os_tcb_dlylist equ     os_tcb_dlylist
_os_tcb_task_tbl        equ     os_tcb_task_tbl
_idle_tsk_id    equ     idle_tsk_id
_os_qcb_tbl     equ     os_qcb_tbl
_os_ecb_tbl     equ     os_ecb_tbl
_os_tcb_tbl     equ     os_tcb_tbl
_os_time        equ     os_time
_os_qcb_free_list       equ     os_qcb_free_list
_os_ecb_free_list       equ     os_ecb_free_list
_os_tcb_free_list       equ     os_tcb_free_list
_os_int_nesting equ     os_int_nesting
_os_lock_nesting        equ     os_lock_nesting
_os_tcb_current_prio    equ     os_tcb_current_prio
        public  _count_ticks
        public  _os_tcb_run
        extrn   _os_ctx_sw:far
        extrn   _new_tick_isr:far
        public  _os_tcb_init
        public  _os_int_exit
        public  _os_int_enter
        public  _os_q_dest
        public  _os_get_cur_taskid
        public  _os_event_dest
        public  _os_q_read
        public  _os_q_write
        public  _os_q_create
        public  _os_mbox_receive
        public  _os_mbox_send
        public  _os_mbox_create
        public  _os_sem_wait
        public  _os_sem_signal
        public  _os_sem_create
        public  _os_get_ticksize
        public  _os_set_ticksize
        public  _os_time_get
        public  _os_time_set
        public  _os_time_tick
        public  _os_msec_dly
        public  _os_tick_dly
        public  _os_task_end
        public  _os_task_change_prio
        public  _os_task_del
        extrn   _os_task_create:far
        public  _os_sched_unlock
        public  _os_sched_lock
        public  _os_sched
        extrn   _os_start_run:far
        public  _os_end
        public  _os_start
        public  _os_init
        public  _divisor
        public  _os_tcb_cur
        public  _os_running
        public  _os_idle_ctr
        public  _os_ctx_sw_ctr
        extrn   __fmemcpy:far
        extrn   _setvect:far
        extrn   _getvect:far
        extrn   _farmalloc:far
        extrn   _farfree:far
_s@     equ     s@
        end


File I186L_C.ASM:
        ifndef  ??version
        endm
publicdll macro name
        public  name
        endm
```

```
        endif
_TEXT   segment byte public 'CODE'
_TEXT   ends
DGROUP  group   _DATA,_BSS
        assume  cs:_TEXT,ds:DGROUP
_DATA   segment word public 'DATA'
d@      label   byte
d@w     label   word
_DATA   ends
_BSS    segment word public 'BSS'
b@      label   byte
b@w     label   word
_BSS    ends
_TEXT   segment byte public 'CODE'
   ;
   ;    RETCODE os_task_create ( void (far *task)(void *pd), void *pdata, WORD stksize, PRIO p, TASK_ID *tsk_id )
   ;
        assume  cs:_TEXT
_os_task_create proc    near
        push    bp
        mov     bp,sp
        sub     sp,6
        push    si
        push    di
        mov     di,word ptr [bp+8]
   ;
   ;    {
   ;            WORD            *stk;
   ;            RETCODE         err;
   ;            void far        *pmem;
   ;
   ;            if ( (pmem=farmalloc(stksize))==NULL)// alloc memory for stack task
   ;
        xor     ax,ax
        push    ax
        push    word ptr [bp+10]
        call    near ptr _farmalloc
        pop     cx
        pop     cx
        mov     word ptr [bp-4],dx
        mov     word ptr [bp-6],ax
        or      ax,dx
        jne     short @1@114
   ;
   ;                return OS_NO_MEMORY;
   ;
        mov     ax,11
@1@86:
        jmp     @1@226
@1@114:
   ;
   ;
   ;            stk             = (WORD*)((BYTE *)pmem+stksize);
   ;
        mov     ax,word ptr [bp-6]
        add     ax,word ptr [bp+10]
        mov     si,ax
   ;
   ;            *--stk  = (WORD)FP_SEG(pdata);          // simulate call to function with argument
   ;
        sub     si,2
        mov     word ptr [si],ds
   ;
   ;            *--stk  = (WORD)FP_OFF(pdata);
   ;
        sub     si,2
        mov     word ptr [si],di
   ;
   ;            *--stk  = (WORD)FP_SEG(os_task_end);// at end of task execute os_task_end()
   ;
        sub     si,2
        mov     word ptr [si],cs
   ;
```

```
;           *--stk  = (WORD)FP_OFF(os_task_end);
;
    sub     si,2
    mov     word ptr [si],offset _os_task_end
;
;           *--stk  = (WORD)0x0200;                     // PSW = Interrupts enabled
;
    sub     si,2
    mov     word ptr [si],512
;
;           *--stk  = (WORD)FP_SEG(task);           // put pointer to task   on top of stack
;
    sub     si,2
    mov     ax,word ptr [bp+6]
    mov     word ptr [si],ax
;
;           *--stk  = (WORD)FP_OFF(task);
;
    sub     si,2
    mov     ax,word ptr [bp+4]
    mov     word ptr [si],ax
;
;           *--stk  = (WORD)0x0000;                     // AX = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = (WORD)0x0000;                     // CX = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = (WORD)0x0000;                     // DX = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = (WORD)0x0000;                     // BX = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = (WORD)0x0000;                     // SP = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = (WORD)0x0000;                     // BP = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = (WORD)0x0000;                     // SI = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = (WORD)0x0000;                     // DI = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = (WORD)0x0000;                     // ES = 0
;
    sub     si,2
    mov     word ptr [si],0
;
;           *--stk  = _DS;                              // Save current value of DS
;
    sub     si,2
    mov     word ptr [si],ds
;
;           err             = os_tcb_init ( p, (void far *)stk, pmem, tsk_id );     // Get and initialize a TCB
```

```
  ;
      push    word ptr [bp+14]
      push    word ptr [bp-4]
      push    word ptr [bp-6]
      push    ds
      push    si
      mov     al,byte ptr [bp+12]
      push    ax
      call    near ptr _os_tcb_init
      add     sp,12
      mov     word ptr [bp-2],ax
  ;
  ;           if ( err == OS_NO_ERR )
  ;
      cmp     word ptr [bp-2],0
      jne     short @1@198
  ;
  ;                   if ( os_running )
  ;
      cmp     word ptr DGROUP:_os_running,0
      je      short @1@198
  ;
  ;                           os_sched ();    // Find highest prio. task if multitasking has started
  ;
      call    near ptr _os_sched
@1@198:
  ;
  ;           return ( err );
  ;
      mov     ax,word ptr [bp-2]
      jmp     @1@86
@1@226:
  ;
  ;   }
  ;
      pop     di
      pop     si
      mov     sp,bp
      pop     bp
      ret
_os_task_create endp
_TEXT   ends
_DATA   segment word public 'DATA'
s@      label   byte
_DATA   ends
_TEXT   segment byte public 'CODE'
_TEXT   ends
      extrn   _farmalloc:near
      extrn   _os_tcb_init:near
      extrn   _os_task_end:near
      public  _os_task_create
      extrn   _os_sched:near
      extrn   _os_running:word
_s@     equ     s@
      end
```

REFERENCIAS:

1. *Byte (varias notas)*
2. *C86 user's manual*
3. *CompuMagazine nro.29 (informe especial)*
4. *Fundamentos de los Sistemas Operativos, Lister (1986)*
5. *Modern Operating Systems, Tanembaum (1992)*
6. *Operating System Concepts, Silberschatz, Peterson (1992)*
7. *Programming the 80386, Crawford & Gelsinger (1987)*
8. *QNX 2.15 MANUAL*
9. *QNX NEWS, the QNX user's newsletter*
10. *Real Time System Design, Levi & Agrawala (1990)*
11. *Real Time Systemas and their Programming Languages, Burns & Wellings (1990)*
12. *Real Time Software. Robert L. Glass (1983)*
13. *Real-time Software Techniques. Walte S. Heat (1991)*
14. *Sistemas Operativos, Diseño e implementación, Tanembaum (1988)*
15. *Telegráfica Electrónica, Confiabilidad en el Software, Ing. José L. Roca*
16. *uC/OS The Real Time Kernel, Jean J. Labrose (1992)*

MARCAS REGISTRADAS: