Tema 8 Sistemas Operativos en Tiempo Real (SOTR)

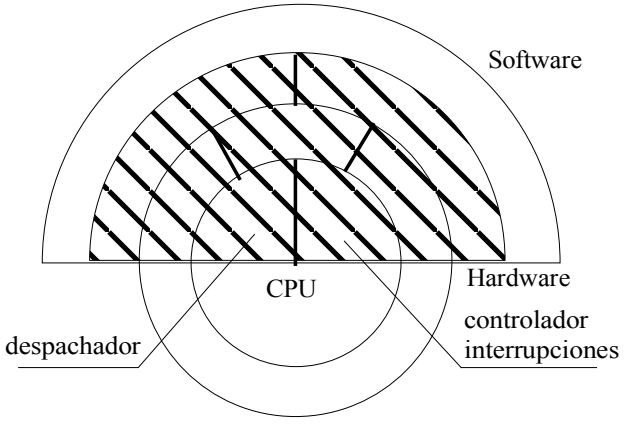
8.1 Introducción.

En este capítulo no se va a hacer un estudio a fondo de los SOTR, sino que se concentrará el esfuerzo en mostrar cuales son los aspectos más relevantes que caracterizan a un SOTR.

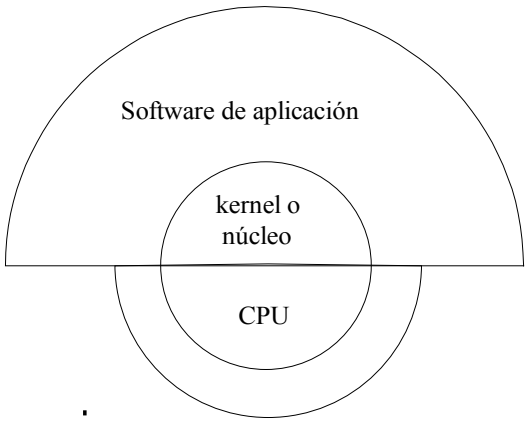
También se pretende que el usuario que, hasta ahora, ha tenido un contacto con los sistemas operativos en el nivel de la línea de órdenes y programas de aplicación, comience a apreciar algunas de las posibilidades que los S0 ofrecen para el programador de aplicaciones, especialmente en tiempo real.

De forma genérica se puede afirmar que la misión de un sistema operativo es doble: por una parte suministrar al usuario una *máquina virtual* más fácil de manejar que el propio *hardware* y que oculte al usuario los aspectos del diseño de éste. Por otra parte debe ser gestor de una serie de recursos, *hardware* y *software*, que son compartidos por los procesos y usuarios que estén trabajando simultáneamente en el sistema. En todo lo que resta de capítulo se entenderá siempre que se trata con sistemas operativos multiusario y multitarea a menos que se especifique lo contrario.

La forma más sencilla de entender un SO es concebirlo como un bloque monolítico. El acceso al *hardware* y a los dispositivos E/S se realiza a través del SO (en sistemas monousuario es posible eludir al SO para manejar el *hardware* directamente). El núcleo de un SO general, de tiempo compartido, a menudo ofrece una serie de facilidades que para una aplicación en particular pueden no ser necesarias, pero si están incluidas en el *Kernel* del SO lo están haciendo más sofisticado y penalizando el tiempo requerido para realizar las tareas que tenga encomendadas. Recientemente los sistemas operativos que proporcionan un *kernel* mínimo (*microkernel*) se están haciendo muy populares, el resto de las características pueden ser añadidas desde un lenguaje de alto nivel por un programador de aplicaciones (que no tiene por qué ser el usuario final). El término *microkernel* no quiere decir, estrictamente, que el tamaño que ocupa sea pequeño, sino que está diseñado de manera que realiza solamente las misiones fundamentales del sistema, dejando para otros procesos el atender a los restantes servicios del sistema operativo.



Sistema operativo de propósito general



Sistema operativo mínimo

Una de las necesidades fundamentales de un SO es asignar recursos del computador a las diversas actividades que se deben realizar. En un SOTR este proceso es complicado debido al hecho de que algunas actividades son temporalmente críticas y unas tienen mayor prioridad que otras. Debe haber algún medio para asignar prioridades a las tareas para que el planificador (scheduler) disponga su ejecución de acuerdo a un esquema de prioridades. Además hay que añadir cuestiones como la gestión de la memoria entre las distintas tareas y las características estándar para soportar un sistema de archivos, dispositivos de E/S y programas de utilidad. El control de todo el sistema recae sobre el módulo *gestor de tareas* responsable de la asignación de la CPU. Este módulo muchas veces se conoce como *monitor* o *ejecutivo*. Todas las características mencionadas anteriormente son compartidas tanto por SOTR como por otros sistemas multitarea y multiusuario en tiempo compartido como el ya conocido UNIX.

Un hecho verdaderamente diferencial entre los SOTR y los SO en tiempo compartido es el algoritmo de planificación para la asignación de la CPU. En los SOTR la aplicación y el SO se encuentran íntimamente acoplados, mucho más que en los sistemas en tiempo compartido. Un SOTR debe ser capaz de responder a intervalos prefijados o ante eventos externos de una manera determinista, y esto sin importar el impacto que pueda derivar sobre otras tareas o procesos. En un entorno en tiempo real, las tareas críticas deben recibir los recursos del sistema que necesiten y cuando los necesiten, a pesar de los efectos que puedan ocasionar a otras tareas. Este hecho también condiciona el comportamiento del SO frente a otros aspectos como pueden ser el sistema de archivos y la gestión de dispositivos de E/S.

8.2 Requisitos de los Sistemas Operativos en Tiempo Real.

Los SOTR se caracterizan por presentar requisitos especiales en cinco áreas generales [MORG92].

1. Determinismo.
2. Sensibilidad.
3. Control del usuario
4. Fiabilidad.
5. Tolerancia a los fallos.

Determinismo

Un sistema operativo es determinista si realiza las operaciones en instantes fijo s y predeterminados o en intervalos de tiempo predeterminados. Cuando hay varios procesos compitiendo por recursos, incluido el procesador, ningún sistema será por completo determinista. El punto hasta el cual un sistema puede satisfacer las solicitudes de manera determinista depende, en primer lugar, de la velocidad con que pueda responder a las interrupciones y, en segundo lugar, de si el sistema posee suficiente capacidad para gestionar todas las peticiones en el tiempo requerido. Una medida útil de la capacidad de un SO para operar de forma determinista es el retardo máximo que se produce desde la llegada de una interrupción de alta prioridad hasta que comience el servicio de la rutina asociada. En un SOTR este tiempo puede ir desde unos pocos microsegundos a 1 milisegundo, en los SO que no son de tiempo real el retardo puede caer en un rango desde decenas a cientos de milisegundos. ´

Sensibilidad

Es una característica semejante a la anterior, hace referencia a cuanto tiempo consume un sistema operativo en reconocer una interrupción, es el tiempo preciso para dar servicio a la interrupción después de haberla reconocido. Depende de:

La cantidad de tiempo necesaria para iniciar la gestión de la interrupción y empezar la ejecución de la rutina de tratamiento (ISR *Interrupt Service Routine*). Si la ejecución de la ISR requiere un cambio de proceso ese tiempo será mayor.

- La cantidad de tiempo necesario para ejecutar la ISR.
- El efecto de anidamiento de las interrupciones. El servicio se retrasará si el sistema debe atender la llegada de otra interrupción más prioritaria.
- El determinismo y la sensibilidad forman conjuntamente el tiempo de respuesta a sucesos externos.

Control del usuario

Es generalmente mucho mayor en un SOTR que en uno de tiempo compartido. En estos últimos un usuario no puede otorgar prioridades a sus procesos, decidir sobre el algoritmo de planificación, qué procesos deben estar siempre residentes en memoria etc.

Fiabilidad

Es normalmente mucho más importante en SOTR Un sistema en tiempo real controla sucesos que están teniendo lugar en el entorno y en su propia escala de tiempos, las perdidas o degradaciones en el sistema que los controla pueden tener consecuencias catastróficas.

Tolerancia de fallos

Un SOTR debe diseñarse para responder incluso ante varias formas de fallo, se pretende que se pueda conservar la capacidad máxima y los máximos datos posibles en caso de fallo. Opciones como la de volcar el contenido de la memoria a un archivo y abortar el programa ante la aparición de un fallo están totalmente prohibidas. Un SOTR intentará corregir el problema o minimizar sus efectos antes de proseguir con la ejecución.

Asociada a la tolerancia a fallos está la **estabilidad.** Un sistema será estable si en los casos es los que es imposible cumplir todos los plazos de ejecución de las tareas se cumplen al menos los de las más críticas y de mayor prioridad.

8.2.1 Características de los Sistemas Operativos en Tiempo Real

Para poder satisfacer los requisitos anteriores las características que los SOTR deben ofrecer se pueden resumir en los siguientes:

Soporte para la planificación de procesos en tiempo real
Planificación por prioridad
Garantía de respuesta ante interrupciones
Comunicación interprocesos
Adquisición de datos a alta velocidad
Soporte de E/S
Control, por parte del usuario, de los recursos del sistema

1. Soporte para la planificación de procesos en tiempo real.

   Un SOTR debe proporcionar soporte para la creación, borrado y planificación de múltiples procesos, cada uno de los cuales monitoriza o controla parte de una aplicación. Típicamente, en un SOTR, es posible definir prioridades para procesos e interrupciones. En contraste, en un sistema de tiempo compartido, solo el propio sistema operativo determina el orden en que se ejecutan los procesos.

2. Planificación por prioridad (pre-emptive).

   Un SOTR debe asegurar que un proceso de alta prioridad, cuando esté listo para ejecutarse, pase por delante de un proceso de más baja prioridad. El SO deberá ser capaz de reconocer la condición (usualmente a través de una interrupción), pasar por delante del proceso que se está ejecutando y realizar un rápido cambio de contexto para permitir la ejecución de un proceso de más alta prioridad. Un SO de propósito general como el UNIX SYSTEM V si tiene un proceso corriendo en el *kernel* debe esperar a que finalice su ejecución en este espacio (en el cuanto de tiempo que le corresponde) para luego activar el proceso más prioritario.

3. Garantía de respuesta ante interrupciones.

   Un SOTR debe reconocer muy rápidamente la aparición de una interrupción o un evento, y tomar una acción determinística (bien definida en términos funcionales y temporales) para atender a ese evento. Debe responder tanto a interrupciones de tipo *hardware* como *software*. El propio SO debe ser interrumpible y reentrante (recordar que uno de los problemas del DOS, que no es un SOTR, se debe a no ser reentrante). Las interrupciones son una fuente introductoria de indeterminismo, imponen la aparición de **latencias.** Estudiaremos éstas al final del capítulo.

4. Comunicación interprocesos.

   Un SOTR debe ser capaz de soportar comunicaciones interprocesos de manera fiable y precisa, tales como semáforos, paso de mensajes y memoria compartida. Estas facilidades se emplean para sincronizar y coordinar la ejecución de los procesos, así como para la protección de datos y la compartición de recursos.

5. Adquisición de datos a alta velocidad.

   Es necesario que el sistema sea capaz de manejar conjuntos de datos con una alta velocidad de adquisición. De esta forma, un SOTR proporciona medios para optimizar el almacenamiento de datos en disco, sobre todo a través de E/S buffereada. Otras características adicionales pueden ser la posibilidad de preasignar bloques de disco contiguos a archivos (almacenamiento secuencial) y dar control al usuario sobre los buffers.

6. Soporte de E/S.

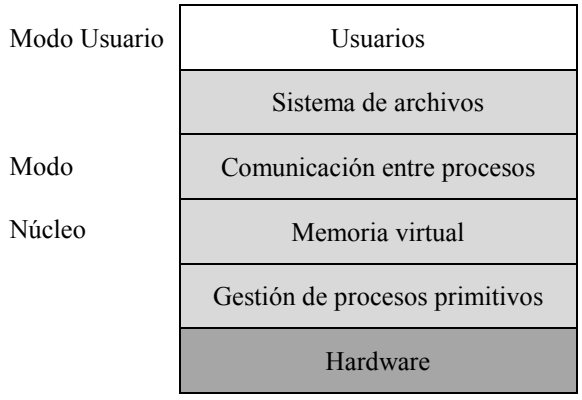
   Las aplicaciones TR típicamente incluyen cierto número de interfaces de E/S. Un SOTR debe proporcionar herramientas para incorporar fácilmente dispositivos de E/S específicos (incluso a medida). Para los dispositivos estándar la librería estándar de E/S debería ser suficiente. Deben además soportar E/S asíncrona, donde un proceso puede iniciar una operación de E/S, y luego continuar con su ejecución mientras concurrentemente se está realizando la operación de E/S. En este aspecto cabe recordar la existencia de procesadores específicos (canales de E/S, controladores de DMA etc.) dedicados a realizar operaciones de E/S sin el concurso de la CPU.

7. Control, por parte del usuario, de los recursos del sistema.

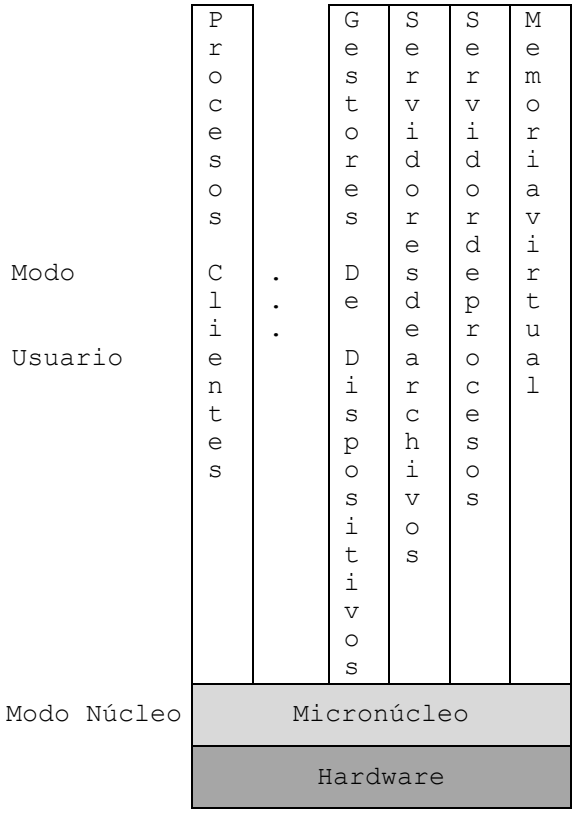
   Una característica clave de los SOTR es la capacidad de proporcionar a los usuarios el control específico de los recursos del sistema, incluyendo la propia CPU, memoria y recursos de E/S. El control de la CPU se logra sobre la base de una planificación por prioridades en la cual los usuarios pueden establecer las prioridades de los procesos. Además, se dispone de temporizadores en tiempo real y de funciones para manejarlos para planificar eventos y períodos de espera. Un SOTR debe también facilitar el bloqueo de la memoria (locking), de esta forma se puede garantizar que un programa, o parte de él, permanece en la memoria, a fin de poder realizar cambios de contexto de manera más rápida cuando ocurre una interrupción. Debería ser capaz de permitir al usuario la asignación de buffers y la posibilidad de bloquear o desbloquear archivos y dispositivos. El control que se ejerce es generalmente mucho mayor en los SOTR que en los SO genéricos. En estos últimos un usuario no tiene control sobre la prioridad de sus procesos, el algoritmo de planificación, el cierre de páginas virtuales en memoria, la reserva de espacio en disco etc.

8.3 La arquitectura *microkernel*

La tendencia actual en los SO es el disponer de un *kernel* (núcleo) pequeño a cargo de un grupo de procesos cooperantes. Como se muestra en la figura, donde se refleja la estructura de los procesos del QNX, la estructura es más de tipo cooperativo que jerárquico, con varias tareas de igual rango comunicándose entre ellas a través del *kernel*. El *kernel* es el corazón de cualquier SO. En algunos sistemas el *kernel* engloba tantas funciones, para tantos propósitos diferentes, que es casi por completo el SO. Esta no es la tendencia de los *kernel* o ejecutivos en TR, para empezar son pequeños (menos de 8 k en el QNX), pero lo que es más importante, está dedicado sólo a las labores esenciales.



Sistema operativo por capas

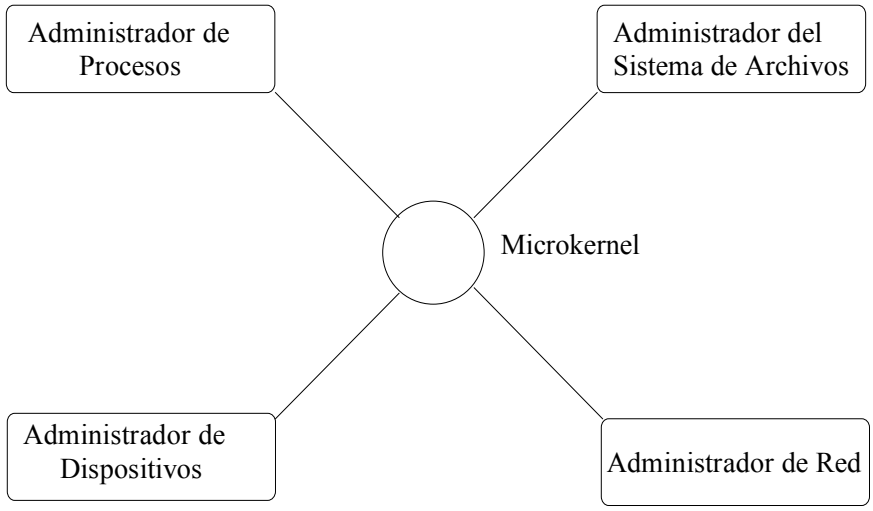


Micronúcleo

En la figura anterior se pueden apreciar dos organizaciones del núcleo. En la parte de la izquierda está la clásica organización por capas con una estratificación horizontal. Cada estrato se comunica con los adyacentes, no todas las estructuras de datos son visibles como sucedería en una estructura monolítica pero los cambios en una capa pueden afectar profundamente a las demás. Esto dificulta hacer versiones para cambiar de una plataforma a otra. Se aprecia que los servicios que ofrece el sistema operativo están contenidos en el núcleo, se ejecutan por tanto de manera protegida y con ciertos privilegios sobre los procesos de usuario.

En la arquitectura micronúcleo se cambia la estratificación horizontal por potra vertical donde los servicios se han sacado del núcleo pero pueden acceder a él de manera directa, sin pasar por otros estratos.

A continuación se muestra la organización de procesos del QNX, que sería semejante en otros operativos en TR, se tiene que las labores encomendadas al *kernel* son:



Organización de procesos del sistema en el QNX.

- Paso de mensajes, manejando el enrutamiento de todos los mensajes entre los procesos de todo el sistema, incluidos los de la red.
- *Scheduling*, decidiendo cuando otorgar el uso de la CPU a un proceso listo para ejecutarse.

A diferencia de otros procesos el *scheduler* nunca se planifica, se ejecuta como consecuencia de llamadas al *kernel*, que pueden haber sido realizadas por procesos o por interrupciones *hardware* (el caso más típico es la interrupción del reloj).

Los restantes servicios que ofrece el SO son manejados como procesos normales. La configuración en QNX tiene los siguientes procesos.

- *Manager* de Procesos (Proc).
- *Manager* del Sistema de Archivos (Fsys).
- *Manager* de Dispositivos (Dev).
- *Manager* de Red (Net).

Procesos del sistema y procesos de usuario.

Los procesos del sistema no se diferencian prácticamente del resto de procesos escritos por los usuarios, no disponen de interfaces privadas u ocultas al resto de procesos. De esta manera se pueden extender las utilidades soportadas por el SO, sin más que escribir un nuevo programa que incorpore un nuevo servicio.

No hay una diferencia muy clara entre los servicios del SO y las aplicaciones. El SO QNX maneja recursos para clientes, que son los procesos. En este sentido se podría establecer una semejanza entre un gestor de bases de datos y el SO manejando el sistema de archivos. Ambos aceptan una serie de órdenes (leer, escribir, abrir archivos) y las realizan independientemente de quien sea el usuario que las ha enviado, siempre que vengan con la interfaz adecuada. De la misma manera pueden dar servicio a clientes que se están ejecutando en su misma máquina o en una computadora diferente.

Manejadores de dispositivos.

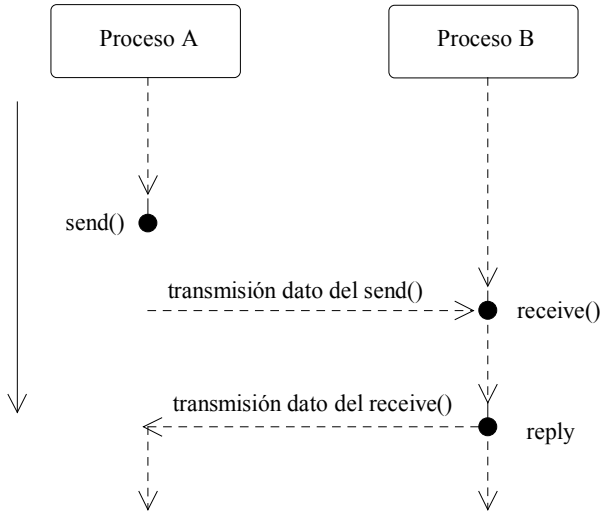
Son procesos que ocultan los detalles del manejo del *hardware*. Dado que los manejadores (*drivers*) se arrancan como procesos normales, añadir un nuevo *driver* no afecta al resto de las partes del SO. Esta política es seguida actualmente por otros SO como el LINUX, que pese a no ser de TR permite cargar y descargar los *drivers* on-line sin necesitar de recompilar todo el *kernel*, como era tradicional en el UNIX.

Comunicación interprocesos (IPC).

Cuando varios procesos se ejecutan concurrentemente en una típica aplicación en TR, el SO debe facilitar un mecanismo para la sincronización e intercambio de información entre los distintos procesos. La IPC es clave en el diseño de una aplicación como un conjunto de tareas cooperantes, en la cual cada proceso maneja una parte bien definida del todo. Son varios los mecanismos de IPC: semáforos, monitores, colas de mensajes y memoria compartida. Diversos autores han demostrado como a partir de semáforos y memoria compartida se pueden implementar colas de mensajes o monitores y viceversa en lo que se conoce como equivalencia de primitivas. La mayoría de las tendencias apuntan al empleo de las colas de mensajes como principal mecanismo de comunicación y sincronización. En efecto, éste es el mecanismo empleado por A. Tanembaum en el desarrollo del SO Minix, el del QNX y también el empleado por el ADA, que pese a no ser un SO implementa en el mismo lenguaje el concepto de proceso (task) y la comunicación entre procesos mediante el mecanismo de cita. En el caso del QNX el paso de mensajes se hace de manera transparente a través de todos los computadores conectados en la red. Hay que recordar que los mensajes son simplemente conjuntos de bytes sin ningún significado especial para el SO Son los procesos que los envían y reciben quienes les otorgan significado. Además el paso de mensajes es el mecanismo más eficaz para comunicar procesos a través de computadores conectados en red.

La sincronización de los procesos también se puede lograr mediante el paso de mensajes, ya que un proceso puede sufrir un cambio de estado cuando envía o se dispone a recibir un mensaje, pasando por ejemplo a bloquearse hasta que el destinatario lo haya leído o el emisor lo envíe.

A este respecto puede citarse como ejemplo el mecanismo de cita que implementa el SO QNX, semejante en cuanto a concepción al del lenguaje ADA. Supongamos dos procesos A y B, donde el proceso A enviará un mensaje al proceso B. Supongamos también que inicialmente el uso de la CPU corresponde a A, que llega hasta el punto de enviar el mensaje, llamada al sistema send, sin que el proceso B haya alcanzado el punto donde espera leer el mensaje, llamada al sistema receive. Llegado a este punto el proceso A se bloqueará tras ejecutar el send, (está send blocked) esperando a que B ejecute el receive.



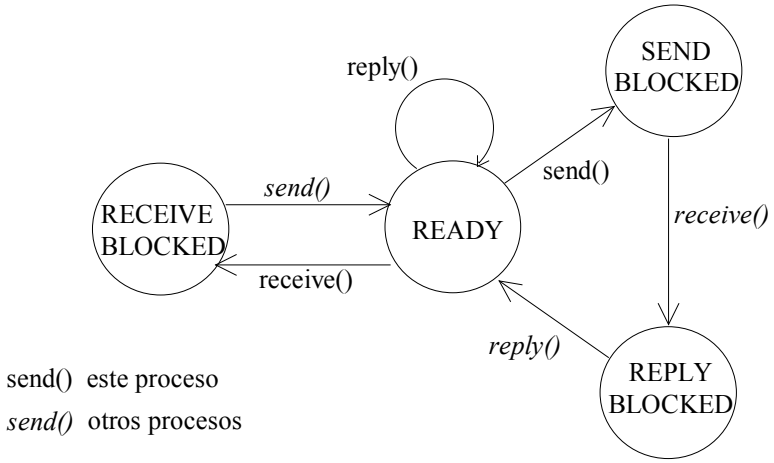
Sincronización con paso de mensajes

Ya que el proceso A se bloquea, el uso de la CPU queda a disposición de B, que ejecutará su código hasta llegar al punto de la llamada a receive. En este momento, el proceso A pasa al estado reply blocked, permanece bloqueado hasta que B le envía una respuesta. B por su parte no se bloquea, procesará los datos recibidos y mandará una contestación al proceso A, llamada al sistema reply.

El mensaje de réplica es transferido al proceso A que se desbloquea y pasa al estado de listo para ejecutarse. Ahora A y B están ambos listos y pueden por tanto competir por el uso de la CPU.

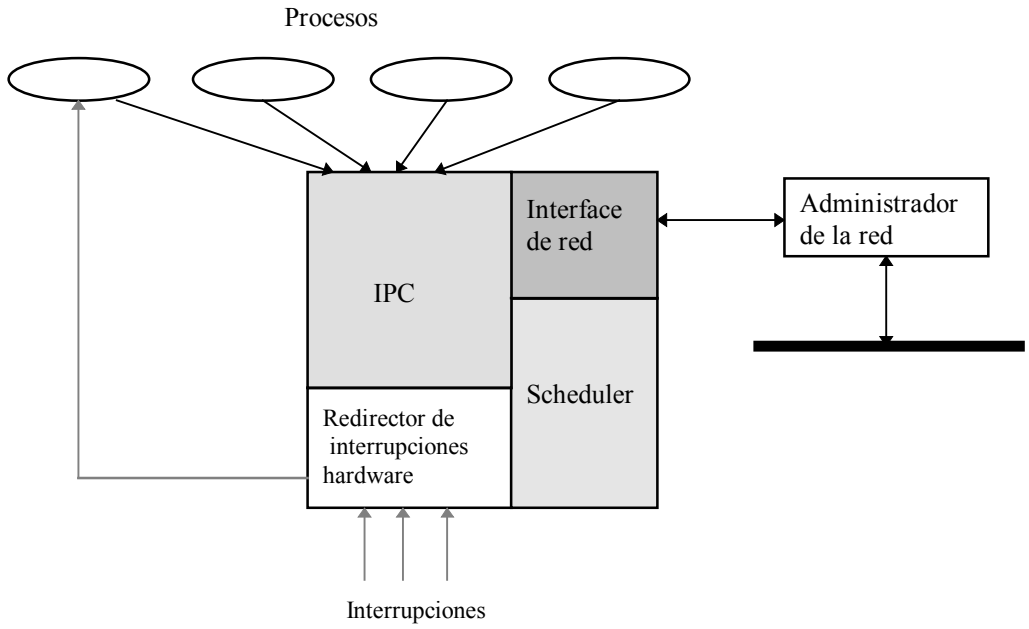
En caso de que el proceso B hubiera ejecutado la llamada a receive antes de que el proceso A hubiese ejecutado el send, B se habría bloqueado (receive blocked) hasta la llegada del mensaje.

Así, las transiciones típicas que puede tener un estado en un proceso de comunicación por paso de mensajes se reflejan en la siguiente figura.



Estados del proceso durante el paso de mensajes

En la siguiente figura aparece una representación de la estructura del *microkernel.*



Estructura del *Microkernel* del QNX.

De esta manera se tienen un conjunto de procesos cooperantes, comunicándose y sincronizándose según el mecanismo IPC antes descrito. Cada nueva aplicación, o servicio del sistema operativo, es simplemente un proceso más, como antes quedó dicho. Por tanto las labores que le quedan al *microkernel* o ejecutivo son las siguientes:

- Comunicación interprocesos.
- Planificación de procesos.
- Despachador de interrupciones.
- Interfaz con la red.

8.3.1 Ventajas de la organización micronúcleo.

La bibliografía comenta varias de las ventajas del uso de micronúcleos.

- Uniformidad de interfases

  Presentan la misma interfaz para las solicitudes provenientes de los procesos, no se diferencia entre procesos de núcleo y de usuario

- Extensibilidad

  Es fácil ampliar las prestaciones del operativo añadiendo nuevos servicios al área funcional correspondiente. Solo es preciso modificar los aspectos directamente implicados en la nueva operación.

- Flexibilidad

  Tiene que ver con la característica anterior, lo mismo que podemos crecer también podemos reducir prestaciones eliminando las partes que no sean necesarias o modificando lo necesario.

- Portabilidad

  La mayor parte del código dependiente de la arquitectura hardware reside en el micronúcleo, por tanto esta es la parte que se necesita modificar para realizar diferentes versiones.

- Fiabilidad

  Cuanto mayor es un producto más difícil resulta garantizar su funcionamiento correcto. Aunque el diseño modular ayuda a aumentar la fiabilidad el diseño micronúcleo mejora aún más este aspecto ya que se puede probar de un modo más riguroso. El uso de pequeños interfases de programa de aplicación (API) aumenta la probabilidad de producir código de calidad.

- Soporte para sistemas distribuidos.

  El micronúcleo ofrece por si mismo soporte para sistemas distribuidos. Cuando un cliente envía una petición a un servidor se debe incluir un identificador del servicio solicitado. Si el sistema está configurado para que todos los procesos y servicios tengan un identificador único, en realidad hay una única imagen del sistema en el ámbito del micronúcleo. Un proceso puede enviar un mensaje sin saber en que máquina reside el destinatario.

- Soporte para sistemas operativos orientados a objetos

  Un enfoque orientado a objetos puede aportar una disciplina al diseño del micronúcleo y al desarrollo de extensiones modulares del sistema operativo

8.4 Estrategias de planificación

La función de un algoritmo de planificación en tiempo real es determinar, para un conjunto dado de tareas, la secuencia y períodos de tiempos en que se deben ejecutar las tareas tal que las necesidades temporales, de precedencia y de recursos se vean satisfechas. Considerando el uso de una única CPU hay dos estrategias básicas:

- Por lotes (en modo batch)
- Pre-emptive (Apropiativa).

8.4.1 Por lotes

Reparte el uso de la CPU por turnos. Una tarea posee el uso de la CPU hasta que finalice el trabajo que está realizando. Cuando deja de utilizarla puede asignarse el procesador a la siguiente tarea en la lista. Esta estrategia es muy eficiente en cuanto al tiempo desperdiciado en la conmutación de procesos (context switching). Esta estrategia es efectiva en pequeños sistemas donde los tiempos de las tareas se puedan calcular cuidadosamente (contando ciclos de código máquina). En general éste no es el caso por lo que esta solución resulta muy restrictiva.

8.4.2 Pre-emptive

Hay muchas estrategias de este tipo; en general se considera que una tarea puede ser interrumpida y desasignarle el uso de la CPU antes de que haya finalizado. Debido a esto es necesario realizar un cambio de contexto cada vez que se conmuta de tarea.

La manera más simple de implementar estas estrategias es establecer un torneo (Round robin). De esta manera a cada tarea se le asigna una cantidad fija de tiempo, un número especificado de *ticks* del reloj conocido como *quanto* (timeslice). Al finalizar el *quanto* la tarea se detiene, se realiza un cambio de contexto y se atiende a otra de las tareas preparadas. Si una tarea finaliza o se bloquea antes de agotar su quanto la CPU es reasignada inmediatamente.

Con el esquema anterior se está suponiendo implícitamente que todas las tareas tienen la misma importancia. Sin embargo, la mayor parte de los SO establecen un esquema de planificación por prioridades. Cada tarea tiene un nivel de prioridad, un número entero, de forma que al final de cada quanto se ejecuta la tarea con mayor prioridad de las que se encuentran preparadas para ser ejecutadas. Dicha prioridad puede ser estática, estando entonces fijada desde el inicio de la tarea. O puede ser dinámica, de esta forma se parte de un determinado nivel de prioridad y a medida que la tarea se ejecuta su nivel de prioridad puede ir creciendo o por el contrario disminuyendo. Factores que afecten el nivel de prioridad son la prioridad base (importancia que de antemano se da a la tarea), la cantidad de tiempo de CPU consumida desde que se inicio la tarea (a mayor gasto menor prioridad), la cantidad de recursos de E/S que utiliza la tarea (a más recursos más prioridad ya que generalmente esto implica que la tarea ha pasado un tiempo bloqueada).

En los SOTR, a diferencia de los sistemas en tiempo compartido, los usuarios pueden establecer la prioridad de los procesos que van a ejecutar. Es frecuente encontrar sistemas con dos políticas de planificación, una para tiempo real y otra para tiempo compartido. Así en el REAL/IX (un UNIX para tiempo real) hay definidos 256 niveles de prioridad, de lo cuales los 128 primeros corresponden a procesos TR, mientras que los restantes son las correspondientes a los procesos en tiempo compartido. Otro ejemplo lo encontramos en Windows NT, donde hay 32 niveles de prioridad de los cuales los 16 más altos corresponden a los procesos de tiempo real, que tendrán prioridades estáticas. En los 16 niveles más bajos la prioridad es dinámica pudiendo oscilar entre ±2 sobre su prioridad base. No es posible que una tarea pase de una banda a otra.

Cualquiera que sea el sistema de planificación elegido es necesario que el SO atienda a las interrupciones. Estas pueden ser de tipo *hardware*, causadas por eventos externos que provienen de los periféricos, o bien de tipo *software*, ocasionadas por una tarea que se está ejecutando. Una interrupción fuerza un cambio de contexto. La tarea que estaba en ejecución se detiene para dejar la CPU a una rutina de servicio asociada a la interrupción. El código asociado a esta rutina de servicio (*handler*) debe ser corto y ejecutarse rápidamente. Una vez ejecutado puede seguirse con la rutina que se detuvo o puede ser que el scheduler planifique una nueva tarea de las que se encuentran listas.

Cuando un sistema es totalmente pre-emptive se debe garantizar que si mientras se está ejecutando una tarea, queda lista para ejecución otra de más alta prioridad, la CPU es asignada inmediatamente a la tarea más prioritaria, sin esperar a que la que se estaba ejecutando finalice o consuma su cuanto.

8.5 Estructura de prioridades

En un SOTR, el diseñador debe asignar prioridades a las tareas, otorgando mayor prioridad a las que tengan unas especificaciones de tiempo más severas y a las que resulten vitales para el correcto funcionamiento del sistema. A grandes rasgos, las tareas pueden ser divididas en tres niveles de prioridad.

Interrupciones: En este nivel se encuentran las rutinas de servicio de interrupción para las tareas y dispositivos que precisan gran rapidez en la respuesta (medida en milisegundos). **Algunas de estas tareas son el reloj en tiempo real y el reloj que maneja el despachador de tareas (*dispatcher*).**

Reloj: Tareas que requieren un procesamiento repetitivo a intervalos de tiempo, tales como algoritmos de control que deben ejecutarse en cada período de muestreo.

Base: Son tareas de baja prioridad y pueden soportar demoras en su ejecución por no tener unas fuertes especificaciones temporales o ser vitales para el sistema. Ejemplos típico de estas tareas

pueden ser las que se encargan de la interfaz con el usuario, pidiéndole datos o mostrándolos por la pantalla.

8.5.1 Nivel de interrupción

Una interrupción fuerza la replanificación del trabajo de la CPU y el sistema pierde el control sobre el tiempo de esta replanificación. Por esta razón, es necesario que el código de la rutina de interrupción se ejecute rápidamente. Además es también preciso que cuente con un fragmento de código para almacenamiento de información volátil, como el contenido de los registros que va a utilizar, para que luego una rutina, que corra en un nivel de prioridad más bajo (nivel de reloj o base), trabaje con esa información salvada y restablezca la planificación.

Dentro de las interrupciones *hardware* también se establecen niveles de prioridad. Generalmente vienen determinados por *hardware* específico que dispone de varias líneas de interrupción y un circuito de arbitrio de prioridades (ver como ejemplo el esquema de interrupciones familia 8086).

8.6 Gestor de tareas

Sus funciones son tres:

1. Mantener un registro del estado de cada tarea. Además, se encarga de los temas relacionados con las creación de procesos, filiación, herencia, gestión de la memoria etc.
2. Planificar el uso de la CPU; Decide qué tarea utiliza el procesador en cada instante.
3. Realizar el cambio de contexto de los procesos.

Estas funciones ya se han estudiado suficientemente en la asignatura Sistemas Operativos, por lo que no van a comentarse aquí. Por la misma razón no se verán las informaciones que caracterizan el contexto de un proceso ni las transiciones que se producen entre estados de un proceso (preparado, bloqueado, en ejecución etc.).

8.7 El planificador y el controlador de interrupciones del reloj de tiempo real

El controlador de interrupciones del reloj de tiempo real (real-time clock handler) y el planificador de las tareas que se ejecutan en el nivel de reloj deben ser diseñados cuidadosamente. Se estudiará con atención el método a seguir para otorgar la CPU a las tareas en cada intervalo del reloj. El planificador tiene dos condiciones para ejecutarse:

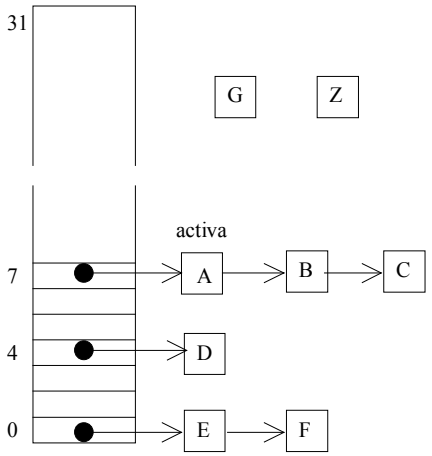
1. Una interrupción del reloj de tiempo real y cualquier otra interrupción que señala la finalización de alguna tarea de E/S y que pueda desbloquear una tarea.
2. La suspensión temporal de una tarea (bloqueo de la tarea), por encontrarse a la espera de algún evento (espera por mensaje, puesta a 1 de un semáforo etc.) o estar realizando una tarea E/S.

En respuesta a la primera condición el planificador busca en la lista de tareas de más alta prioridad y las comprueba en orden de prioridad. De esta manera, si las tareas que se repiten con un período muy corto se les asigna una alta prioridad entonces serán tratadas como si fueran tareas en el ámbito de reloj.

En respuesta a la segunda de las condiciones el planificador buscará la siguiente tarea de prioridad inferior a la que estaba ejecutando. Ahora no puede ser una de prioridad más alta, ya que sino, estaría ejecutándose anteriormente.

Habitualmente las ordenes para el manejo de las tareas se programan como llamadas a procedimientos en ensamblador, enviándose los parámetros dentro de los registros de la CPU o como una palabra de control justo después de la llamada al procedimiento para obtener una mayor eficiencia.

Para clarificar estos importantes conceptos se van a exponer las técnicas de gestión de procesos de un SOTR comercial llamado QNX. Aquí cada proceso tiene asignada una prioridad, de 0 la más baja, hasta 31, la más alta. La prioridad predefinida de un proceso es la misma que la definida para su proceso padre. Lo normal, para las aplicaciones creadas por el Shell, es 10.



Listas de tareas preparadas para ejecutarse ordenadas por prioridades

Como siempre, el planificador solo estudiará la lista de tareas listas para ejecución, escogiendo de estas a la de más alta prioridad para darle el uso de la CPU. Para cada nivel de prioridad va a establecerse una lista separada. Se cuenta con llamadas al sistema para que **un proceso de usuario** pueda averiguar y establecer su nivel de prioridad.

El usuario cuenta también con la posibilidad de decidir entre tres algoritmos de planificación diferentes, disponiendo también de llamadas al sistema para conocer cual es el algoritmo que está utilizando, o establecer uno diferente. Estos tres algoritmos son los siguientes:

- FIFO.
- Round robin.
- Adaptativo.

En la figura anterior, cualquiera que sea el algoritmo que se esté utilizando, el proceso activo será el A, pues es el primero de la lista entre los de más alta prioridad de los que se encuentran listos para ejecución (*ready*). Los procesos entre G y Z están bloqueados y por tanto no se planifican, aún cuando pudieran ser de muy alta prioridad.

Si se sigue la planificación FIFO el proceso A seguirá ejecutándose hasta que:

- Finalice.
- Se bloquee y libere el uso de la CPU.
- Sea pasado por alto por un proceso de más alta prioridad. Bien porque aparezca una interrupción, se cree un proceso de más alta prioridad, se desbloquee un proceso de más alta prioridad etc.

Si se sigue la planificación round-robin el proceso A seguirá ejecutándose hasta que:

- Finalice.
- Se bloquee y libere el uso de la CPU.
- Sea pasado por alto por un proceso de más alta prioridad, por algunas de las razones vistas anteriormente.
- Consuma su quanto de tiempo asignado. En este momento el uso de la CPU pasa al siguiente proceso que esté preparado dentro de la lista de más alta prioridad de procesos listos para ejecutar. En QNX la duración del quanto es de 100 ms.

Si se sigue la planificación adaptativo, el proceso A se comporta como sigue.

- Si el proceso consume su quanto, es decir, no se bloqueó, su prioridad se reduce en 1, siempre que exista un proceso preparado de la misma prioridad.
- Si el proceso vio decrementada su prioridad y permanece sin planificar durante un segundo, su prioridad es incrementada en 1, pero nunca se incrementará por encima de su prioridad original.
- Si el proceso se bloquea vuelve inmediatamente a poseer su prioridad original.

Puede emplearse este algoritmo adaptativo en entornos donde puede haber procesos de fondo (en *background*) que hacen un uso intensivo de la CPU (muchos cálculos y poca E/S), que están compartiendo la computadora con usuarios interactivos (mucha E/S). Este algoritmo da a los procesos de cálculo bastante uso de CPU sin perder velocidad para las respuestas interactivas de otros procesos. El algoritmo adaptativo es el utilizado de forma predefinida por los procesos creados por el Shell del QNX.

Podemos ahora analizar un caso práctico. Se trata del esquema cliente servidor, donde el servidor tiene capacidad para atender peticiones de diversos clientes, lo cual es muy frecuente. Por esta razón es normal que la prioridad del servidor sea superior a la de todos sus clientes para ofrecer un mejor servicio. Cualquiera de los tres

métodos de planificación vistos anteriormente puede ser utilizado, pero probablemente el más adecuado sea el de round-robin. Si un cliente de baja prioridad envía su petición al servidor, esta petición al ser atendida pasa a tener la prioridad del servidor. Si el tiempo que consume el servidor para atender la petición del cliente es pequeño, no hay ningún problema, pero si este tiempo se alarga, resulta que clientes de más alta prioridad que el que realizó esta petición no se pueden planificar, puesto que se ejecuta el servidor. De alguna manera, un proceso está colapsando a otros de más alta prioridad. Para resolver este dilema se puede adoptar una solución donde el servidor toma la prioridad de la petición del cliente que solicita el servicio, sin cambiar el algoritmo de planificación. Si, mientras el servidor se está ejecutando, llega un nuevo mensaje, la prioridad del servidor se incrementará en caso de que el nuevo cliente tenga una prioridad más alta que el original. El servidor finaliza con la petición original para luego atender a la nueva, pero ya con una prioridad mayor.

Se dispone de una llamada al sistema que permite a un proceso servidor adoptar la prioridad de los procesos clientes.

8.8 El Sistema Operativo UNIX y el Tiempo Real

El SO UNIX System V desarrollado por AT&T ha pasado a ser un estándar, ganando rápida aceptación debido a su flexibilidad, portabilidad y al elevado número de herramientas que soporta. Pero el SO UNIX fue diseñado originalmente para la multitarea y el tiempo compartido, por estas razones no tiene una respuesta adecuada a las especificaciones temporales que requieren las aplicaciones en tiempo real. Entrando más en detalle sobre el UNIX, veremos sus deficiencias en los siguientes aspectos.

Planificación: Está diseñada para proporcionar un tratamiento equitativo en el acceso a la CPU y a la memoria. Cada proceso dispone de una porción de tiempo para ejecutarse. El planificador periódicamente recalcula las prioridades de los procesos para asegurar un reparto equitativo de la CPU por cada proceso.

Manejo de las interrupciones: En el UNIX estándar, un proceso trabajando en el espacio del sistema (llamadas del sistema en modo *kernel*) no puede ser pasado por alto por una interrupción. Esto es así aunque el proceso que haya ejecutado la llamada al sistema sea de baja prioridad. En algunos casos, como llamadas a fork donde se deban copiar muchos datos, la finalización de la llamada puede requerir varios segundos, por lo cual esto es inaceptable para aplicaciones en tiempo real.

Comunicaciones interprocesos: El UNIX posee semáforos, colas de mensajes y mecanismos de compartición de memoria. También los pipes y las señales pueden utilizarse a efectos de comunicación. Su única pega en este aspecto es que las llamadas al sistema pueden consumir mucho tiempo de CPU.

Sistema de archivos: En el UNIX estándar el espacio en disco es asignado a los archivos cuando se realizan operaciones de escritura, no cuando se efectúa la creación del archivo. Esto tiene dos efectos: en primer lugar no se garantiza que cuando se vaya a escribir en un archivo ya creado haya realmente espacio; en segundo lugar las operaciones de escritura consumen más tiempo por tener que ocuparse también de la asignación de memoria. Además, para un mejor aprovechamiento del disco, los bloques que ocupa un archivo no tienen por qué estar contiguos. Esto también causa un retraso en las operaciones de L/E.

Soporte de E/S: No soporta operaciones asíncronas de E/S, no hay capacidad para conectarse directamente a dispositivos de E/S. Si proporciona la posibilidad de escrituras y lecturas en forma de no bloques.

Control, por parte del usuario, de los recursos del sistema: El UNIX System V proporciona la posibilidad de dejar permanente en memoria un fragmento de código o datos, para responder rápidamente en un cambio de contexto cuando se produzca algún evento, pero no da medios para garantizarse el uso de dispositivos.

Como se ve el UNIX, construido para procesos en tiempo compartido, presenta deficiencias para su uso como SOTR, pero se le pueden añadir funciones adecuadas para transformarlo en un operativo adecuado para el tratamiento de sistemas en tiempo real [3]

8.8.1 Estándares de UNIX para el tiempo real.

La próxima generación de sistemas en tiempo real usará sistemas abiertos, que incorporarán estándares industriales. Estos sistemas abiertos pueden especificar estándares tanto en el ámbito de *hardware* (caso típico los buses) como de *software* (lenguajes, sistemas operativos, plataformas X) ofreciendo una serie de ventajas. Tales ventajas quedan expuestas en el siguiente cuadro en contraposición con los sistemas específicos.

Ventajas para los usuarios	Sistemas específicos	Sistemas abiertos
Portabilidad *software*	Meses/Años	Horas/Semanas
Conversión de bases de datos	Años	Horas/Días
Reciclado y disponibilidad del programador	Grandes tiradas	Insignificante
Flujo de mejoras	Controlado por el fabricante	Todo el mercado

Con los años han aparecido una gama de sistemas operativos UNIX y varios esfuerzos para estandarizarlos. De estos cabe destacar los de AT&T, que dieron como fruto el UNIX System V y el del comité ANSI/IEEE que ofrece el POSIX. En la estandarización POSIX (P.1003.x, donde x hace referencia a un área) se incluyen un número de corporaciones que trabajan en varios subestandares, muchos aún en fase de desarrollo. Así se encuentra el estándar P.1002.2 relativo al shell y a las herramientas, métodos de test (P.1003.3), tiempo real (P.1003.4), ADA (P.1003.5), seguridad (P.1003.6), administración del sistema (P.1003.7), funcionamiento en red (P.1003.P) y otras materias.

FUNCION	DESCRIPCION
1. Temporizadores	Capacidad de programar y leer temporizadores internos de alta resolución
2. Planificación por prioridad	Planificación pre-emptive, garantizando la ejecución de las tareas de alta prioridad en cuanto están preparadas
3. Compartición de memoria	Mapeo de un espacio físico de memoria común dentro de los espacios virtuales de cada proceso, de esta forma varias aplicaciones pueden acceder a fragmentos de datos o código comunes.
4. Archivos en tiempo real	Creación y acceso a archivos con características deterministas. Un proceso puede preasignarse espacio en disco para un archivo y determinar características que mejorarán las operaciones con archivos
5. Semáforos	Debe garantizar las primitivas de sincronización P y V, elementos básicos para la sincronización de tareas.
6. Comunicación interprocesos	Paso de mensajes síncrono y asíncrono con facilidades para el control de recursos. Debe buscarse la posibilidad de trabajar en red de manera transparente al usuario
7. Notificación de eventos asíncronos	Un mecanismo que proporcione el almacenamiento en colas de los eventos, entrega determinística y paso de datos mínimo
8. Cierre (bloqueo) de memoria	Capacidad para dejar residentes secciones del espacio virtual de un proceso. El fin de esta propiedad es eliminar tiempos de latencia en los cambios de contexto.
9 E/S asíncrona	Posibilidad de que una aplicación procese colas de datos con notificaciones de la finalización de las operaciones de E/S iniciadas por la aplicación
10. E/S sincronizada	Capacidad para establecer garantías para que se complete una operación de E/S a diferentes niveles lógicos de finalización

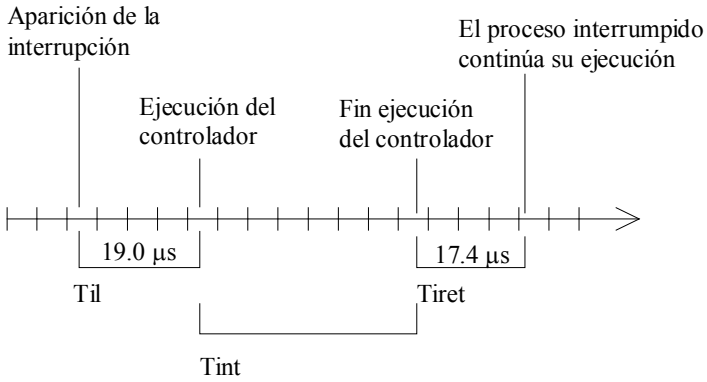
El propósito del estándar POSIX en tiempo real (P.1003.4) es desarrollar un conjunto de interfaces que permitan la portabilidad de aplicaciones en tiempo real. Estas necesidades conducen a 10 funciones muy importantes en el tiempo real. Se está incluso avanzando en el desarrollo de estándares de métrica que permitan la medición de las prestaciones.

8.9 Prestaciones de los sistemas en tiempo real.

A pesar del incremento de velocidad de los computadores, ésta no es infinita, y se ha visto acompañada por un incremento de similar magnitud de la complejidad de sistemas a controlar. Por tanto, en algunas aplicaciones, es crucial no desperdiciar ciclos de CPU. Es también de máxima importancia que se minimice el tiempo transcurrido desde la aparición de un evento externo a la ejecución del código relacionado con dicho evento. Este tiempo se conoce como latencia. En cualquier sistema pueden encontrarse varias formas de latencia, que son a menudo utilizadas para caracterizar las prestaciones del sistema.

8.9.1 Latencia ante interrupciones

Es el tiempo transcurrido desde la recepción de una señal de interrupción *hardware* hasta que la primera instrucción *software* del controlador de interrupciones es ejecutada. Hay que considerar que cuando aparece una interrupción debe guardarse en la pila el valor del contador de programa, probablemente también la palabra de estado. Se debe acudir a una tabla de vectores de interrupción y cargar el valor correspondiente en el contador de programa. Todas estas operaciones son realizadas automáticamente por el *hardware* y consumen poco tiempo, salvo que las interrupciones, por alguna necesidad concreta del programa, se encuentren desactivadas temporalmente, en cuyo caso la latencia será mayor.



Til tiempo de latencia de la interrupción
Tint tiempo de procesamiento de la interrupción
Tiret tiempo de retorno tras la finalizacuón

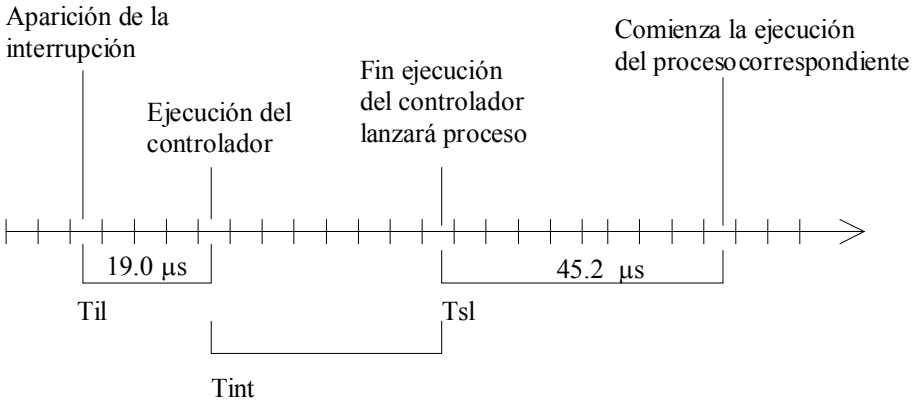
Los tiempos corresponden a QNX con un procesador 386 a 20 MHz en modo protegido

Este tiempo de latencia podría subdividirse en otros más atómicos correspondientes al almacenamiento de los registros del procesador, u otras latencias que se puedan dar cuando varias interrupciones ocurran simultáneamente y el *hardware* deba decidir a cual atender primero.

8.9.2 Latencia ante la planificación

En muchas ocasiones, ante la aparición de un evento, será necesario lanzar un nuevo proceso. Se definirá la latencia en la planificación como el tiempo transcurrido entre la finalización de un controlador de interrupción y la ejecución de un proceso provocada por esa interrupción. Este tiempo comprende las demoras necesarias para verificar qué proceso se debe planificar, realizar el cambio de contexto y un tiempo que el sistema pudiera tener desactivada la *preemption* de nuevos procesos. Este tiempo es necesario para ajustar punteros y acceder a estructuras de datos exclusivas del sistema.

En ambos tipos de latencias se pueden estudiar los tiempos promedios, pero para casos críticos deben considerarse también los peores tiempos de latencia. Estos tiempos aparecerían cuando se dieran varias solicitudes de interrupción simultáneamente, que la atención a las interrupciones esté desactivada temporalmente, lo mismo que la *preemption*, que las listas de procesos estén muy llenas etc.



Til tiempo de latencia de la interrupción
Tint tiempo de procesamiento de la interrupción
Tsl tiempo de latencia de planificación

Los tiempos corresponden a QNX con un procesador 386 a 20 MHz en modo protegido

8.10 Algunas características de un operativo en tiempo real

A modo de resumen, y sin ánimo publicitario, se relacionan a continuación algunas de las características del QNX, sistema operativo en tiempo real que corre, entre otras, en arquitecturas PC.

En el nivel del sistema operativo:

- ◊ Escalable hacia arriba para permitir aplicaciones multiprocesador.
- ◊ Escalable hacia abajo para poder trabajar en aplicaciones incrustadas (códigos contenidos en ROM).
- ◊ Integra el procesamiento distribuido a través de la red; de esta forma los computadores que forman la red son presentados como una sola máquina lógica.
- ◊ Compatibilidad POSIX en la gestión de procesos.
- ◊ Compatibilidad POSIX en el subsistema de archivos y de E/S.
- ◊ Tiempos de respuesta en tiempo real deterministas.
- ◊ Soporta el procesamiento multitarea y multiusuario.
- ◊ Compatibilidad POSIX 1003.1 en la Interfaz de Aplicaciones (API).
- ◊ Compatibilidad POSIX 1003.2 en las utilidades y shell.
- ◊ Compatibilidad POSIX 1003.4 en las extensiones en tiempo real
- ◊ Configurable dinámicamente y extensible en tiempo real.
- ◊ El gestor de recursos y los controladores de dispositivos son totalmente dinámicos, y pueden ser puestos en marcha o detenidos sin rebotar o reconstruir el *kernel* del sistema operativo.
- ◊ Posibilidad de tener hasta 300 procesos concurrentes por cada nodo.

A nivel del *microkernel*.

- ◊ El *microkernel* ocupa menos de 10 Kbytes.
- ◊ Planificación en tiempo real compatible POSIX 1003.4.
- ◊ Soporta 32 niveles de interrupción.
- ◊ Tiene 3 algoritmos de planificación: FIFO, round-robin y adaptativo seleccionable por proceso.
- ◊ Cambio de contexto totalmente priorizado y pre-emptive.

- ◊ Los servidores pueden tener su prioridad conducida por los mensajes que reciben a través de las primitivas de pasos de mensajes.
- ◊ Solo dispone de 16 llamadas, el resto de servicios del sistema son proporcionados por otros procesos.
- ◊ El *microkernel* simplemente maneja la planificación de procesos y el paso de mensajes.
- ◊ El paso de mensajes en el *microkernel* es totalmente pre-emptive.

En el nivel del gestor de procesos.

- ◊ Gestión de relojes y temporizadores según POSIX 1003.4.
- ◊ Cada proceso dispone de múltiples temporizadores.
- ◊ Los temporizadores pueden ser síncronos o asíncronos, de un solo disparo o repetitivos.
- ◊ La resolución de los temporizadores es de nanosegundos.
- ◊ Soporta la herencia total del entorno de los procesos, incluso a través de una red para poder permitir un procesamiento distribuido transparente. Esto incluye los descriptores de archivos abiertos, directorio de trabajo actual, variables de entorno e identificador de usuario.
- ◊ Admite interrupciones anidadas.
- ◊ Primitivas flexibles para la compartición de memoria.
- ◊ Primitivas que permiten la depuración de programas, incluso a través de la red.
- ◊ Límites y recursos del sistema configurables por el usuario.
- ◊ Los controladores de las interrupciones pueden ser aplicados y eliminados dinámicamente.

En el nivel del gestor del sistema de archivos.

- ◊ El sistema admite que varios sistemas de archivos funcionen concurrentemente.
- ◊ Sistema de archivos de altas prestaciones basado en una extensión POSIX para sistemas de archivos.
- ◊ Robustez: toda la información importante para la consistencia del sistema de archivos es escrita directamente en el disco.
- ◊ Soporta completamente el uso de tuberías (pipes), tanto en estructuras monoprocesadoras como a través de la red.
- ◊ Las aplicaciones pueden pasar por alto el sistema de archivos y operar directamente con los controladores de los dispositivos para la transmisión de datos "puros".
- ◊ Nombres de archivos de hasta 48 caracteres.
- ◊ Límites y recursos del sistema de archivos configurables por el usuario.

En el nivel del gestor de dispositivos.

- ◊ Escrituras buffereadas.
- ◊ Las entradas buffereadas a través de los dispositivos tty (terminales) pueden ser de 256 bytes hasta 64 Kbytes para aplicaciones de altas velocidades.
- ◊ Las peticiones de entradas pueden retornar si no han sido atendidas en un tiempo prefijado (timeout)
- ◊ Proporciona primitivas asíncronas de E/S.

El QNX tiene además muchas otras aplicaciones y entornos de trabajo desarrollados, existiendo versiones para diferentes arquitecturas, no solo para PC's. El hecho de que se haga referencia a él no debe interpretarse como que sea el mejor operativo en tiempo real, es un operativo que se adapta a muchas de las características deseables de los sistemas en tiempo real y del cual disponemos de cierta información.

Bibliografía

[1] Stuart Bennet . ‘Real Time Computer Control’. Ed Prentice Hall 94 ISBN 0-13-764176-1

[2] Christopher Vickery. ‘Real Time and Systems Programming for PC's. Using the iRMX for Windows Operating System.’ Ed Prentice Hall 94 ISBN 0

[3] Borko Furht y otros. ‘Real Time UNIX Systems. Design and Application Guide’. Ed Kluwer 91 ISBN 0-7923-9099-7

[4] William Stallings: Sistemas Operativos. Ed Prentice Hall 2001 ISBN 84-205-3177-4

[5] Fco. Manuel Marquez García. ‘UNIX Programación avanzada’ Ed Ra-ma 93 ISBN 84-7897-112-2