Capítulo 4

Introducción a los Sistemas de Tiempo Real

4.1 Introducción

En la actualidad el desarrollo de aplicaciones para STR tiene una gran importancia porque un amplio rango de sistemas poseen características de tiempo real [Burns, 2003]. Hay que tener en cuenta que en la actualidad el 99 % de la producción mundial de procesadores se utiliza para la construcción de sistemas empotrados, cuyo software de control suele basarse en un núcleo simplificado de operativo de tiempo real. A menudo, el tiempo real se confunde con *en línea*, con *interactivo* o con *rápido* [Stankovic, 1988]. Los responsables de la mercadotécnia de sistemas son particularmente propensos a este error. Que un sistema esté *en línea* significa que se encuentra siempre disponible, pero eso no garantiza la *responsividad*, es decir, una respuesta en un tiempo acotado. A menudo los sistemas en línea ni siquiera garantizan una respuesta. Por *interactivo* se entiende que el tiempo de respuesta del sistema es adecuado desde el punto de vista de un usuario humano. Si bien es cierto que habitualmente los sistemas en tiempo real trabajan a frecuencias altas para una persona (algunas del orden de cientos o miles de veces por segundo), no tiene por qué ser siempre así: un sistema complejo podría tener un plazo de segundos o incluso de días y aún operar en tiempo real. La condición de tiempo real es que la respuesta se obtenga en un plazo prefijado, independientemente de su duración.

En los STR continuos es preciso, además, que el tiempo necesario para procesar la información sea inferior al tiempo de su llegada. Los sistemas de procesamiento de audio, vídeo o telefonía sobre Internet son sistemas continuos de tiempo real. Si el tratamiento de una señal de sonido necesita 1,1 segundos por segundo de señal, el sistema no es de tiempo real; si en cambio bastan 0,9 segundos, entonces es posible hacer que sea un sistema de tiempo real, de tal forma que se podría escuchar la señal al tiempo que se va procesando.

La definición más comúnmente aceptada de lo que es un sistema de tiempo real (STR) es: "Un sistema en tiempo real es aquel en el que la respuesta correcta a un cálculo no sólo depende de su corrección lógica, sino también de cuándo dicha respuesta está disponible" [Burns, 2003]. Un buen ejemplo es el de un robot que necesita tomar una pieza de una banda sinfín. Si el

robot llega tarde, la pieza ya no estará donde debía recogerla. Por lo tanto el trabajo se llevó a cabo incorrectamente, aunque el robot haya llegado al lugar adecuado. Si el robot llega antes de que la pieza llegue, la pieza aún no estará ahí y el robot puede bloquear su paso.

En el ámbito de los *sistemas operativos*, el estándar POSIX define a un sistema operativo de tiempo real como aquél que tiene la capacidad para suministrar un nivel de servicio requerido en un tiempo limitado y especificado de antemano. Es decir un sistema de este tipo debe permitir satisfacer plazos de entrega prefijados a todos sus procesos que estén etiquetados como de tiempo real.

Hay una serie de elementos o propiedades característicos que poseen todos los sistemas de tiempo real y que nos pueden ayudar a identificarlos correctamente:

- **Reactividad:** se dice que los STR son sistemas *reactivos* porque su funcionamiento se basa en una interacción continua con su entorno, a diferencia de los *transformacionales* cuyo comportamiento abstracto es parecido al de una función matemática: entrada de datos, cálculos y salida de resultados.
- **Determinismo:** es una cualidad clave en los sistemas de tiempo real. Es la capacidad de determinar con una alta probabilidad, cuánto es el tiempo que tarda una tarea en iniciarse. Esto es importante porque los sistemas de tiempo real necesitan que ciertas tareas se ejecuten antes de que otras se puedan iniciar. Este dato es importante saberlo porque casi todas las peticiones de interrupción se generan por estímulos que provienen del entorno del sistema, así que resulta muy importante para poder determinar el tiempo que el sistema tardará en dar el servicio.
- **Responsividad:** esta propiedad tiene que ver con el tiempo que tarda una tarea en ejecutarse una vez que la interrupción ha sido atendida. Los aspectos a los que se enfoca son: (a) la cantidad de tiempo que se lleva el iniciar la ejecución de una interrupción; (b) la cantidad de tiempo que se necesita para realizar la tarea que solicitó la interrupción y (c) los efectos de interrupciones anidadas.
- **Confiabilidad**: es otra característica clave en un sistema de tiempo real. El sistema no debe sólo estar libre de fallas sino, más aún, la calidad del servicio que presta no debe degradarse más allá de un límite determinado. El sistema debe de seguir en funcionamiento a pesar de catástrofes, o fallas mecánicas. Usualmente una degradación en el servicio en un sistema de tiempo real lleva consecuencias catastróficas.

4.1.1 Clasificación de los sistemas de tiempo real

Los STR se clasifican según su criticidad en permisivos o *suaves* (*"soft"*) y de misión crítica o no-permisivos (*hard*). Un sistema de *misión crítica* es aquel en que es inadmisible que los resultados lleguen tarde, mientras que en uno *permisivo* sólo se producirán pérdidas de rendimiento que, según su coste, pueden resultar aceptables en una versión entregable del sistema final. La pérdida de un tiempo límite supone un fallo total del sistema en el caso de los sistemas de *misión crítica*. Un sistema de misión crítica sería, por ejemplo, el encargado de controlar la maniobra de atraque del transbordador espacial Atlantis en la *International Space Station*, mientras que un reproductor de video que ocasionalmente pierde alguna trama puede ser un

ejemplo de sistema permisivo. El robot de la cinta de montaje de piezas de un ejemplo anterior exigiría un sistema de misión crítica si el llegar tarde a la pieza supusiese una paralización completa de la línea de montaje, mientras que podría ser permisivo si, como consecuencia de su retraso, disminuyese el ritmo de producción de la cadena de montaje.

Existe un tercer tipo de STR que se encuentra en medio de los dos anteriores (ver tabla 4.1), los denominados STR *estrictos* (*"firm"*). Con este tipo de sistemas, serían *tolerables* pérdidas infrecuentes del tiempo límite de las tareas de tiempo real, aunque dichas pérdidas pueden llegar a degradar la calidad de servicio del sistema. A diferencia de los STR permisivos, en los que se obtiene alguna ganancia con los resultados tardíos, la utilidad de los resultados que se producen después de cumplirse el tiempo límite es nula en los sistemas estrictos.

Los sistemas de tiempo real industriales suelen contener una mezcla de susbsistemas componentes de las tres clases anteriormente comentadas.

Denominación	Ejemplo	Complementos
Misión crítica	Control de aterrizaje	Tolerancia a fallos
Estrictos	Reservas de vuelos	Calidad de respuesta
Permisivos	Adquisición datos meteorológicos	Medidas de fiabilidad

Tabla 4.1: Clasificación de los sistemas de tiempo real atendiendo a su criticidad

4.1.2 Medidas de tiempo

La característica fundamental de un STR es su capacidad para ejecutar las instrucciones programadas en sus tareas dentro de intervalos de tiempo bien definidos. Por lo tanto, para poder desarrollar de forma apropiada cualquier STR hay que contar con mecanismos adecuados para la medida del tiempo, así como para controlar la duración de las instrucciones referidas anteriormente [Cheng, 2002].

En la actualidad resulta impensable desarrollar software con características de tiempo real sin contar con el apoyo de un lenguaje de programación de alto nivel y un sistema operativo apropiado , que nos proporcionen: (a) relojes de *tiempo real*, que nos ayuden a medir el tiempo con precisión; (b) mecanismos para activar tareas en instantes previamente determinados; (c) *tiempos límite de espera* ("timeouts"[1]); (d) planificadores de tareas de tiempo real adaptables por el programador a las necesidades de su aplicación.

El tiempo es una magnitud física fundamental, cuya unidad en el Sistema Internacional (SI) es el segundo. Para poder desarrollar software necesitaremos dos tipos de medidas del tiempo:

- Tiempo absoluto
- Intervalos o tiempo relativo

[1]un periodo de tiempo después del cual surge una condición de error si no se ha producido algún evento, se ha recibido una entrada, etc. Un ejemplo frecuente es el envío de un mensaje. Si el receptor no reconoce el mensaje dentro de un periodo de tiempo prestablecido, se infiere que ha ocurrido un error de transmisión y se levanta una excepción en el programa.

El tiempo absoluto necesita un sistema de referencia con un origen que se denomina *época*. Podemos utilizar varios sistemas de referencia: (a) *locales*: suelen coincidir con el tiempo transcurrido desde el arranque de nuestro sistema; (b) *astronómicos*: por ejemplo, el denominado *Tiempo Universal* (UT0), que es el término moderno usado para la medida internacional de tiempo utilizando un sistema basado en telescopio, y que fue adoptado en 1928 para reemplazar al sistema GMT ("Greenwich Mean Time") por la Unión Astronómica Internacional; (c) *atómicos*: el tiempo se mide en Ciencia con una cuenta continua de segundos basada en relojes atómicos ubicados alrededor del mundo, este sistema se conoce con el nombre de Tiempo Atómico Internacional (IAT); la duración de 1 segundo es constante, puesto que es definida a partir del inmutable periodo de transición del átomo de Cesio; (d) El *Tiempo Coordinado Universal* (UTC): actualmente la base para la medida del tiempo en la vida civil, desde el 1 de Enero de 1972, en que fue definido para seguir al IAT con una desviación dada por un número entero de segundos, cambiando sólo cuando se le añade 1 segundo para sincronizarse con la rotatión de la Tierra; (e) *satelital*: el Sistema de Posicionamiento Global (GPS) también transmite una *señal de tiempo* para todo el planeta, además de proporcionar instrucciones precisas para convertir el tiempo GPS a UTC.

Relojes de tiempo real

Un *reloj* en el contexto de esta asignatura es un módulo compuesto por elementos *hardware* y *software* que nos proporciona el tiempo real cuando se lee su valor, que mantiene actualizado durante la ejecución de nuestro sistema. Un *reloj* está compuesto por: (a) un circuito *oscilador* que genera impulsos eléctricos, (b) un *contador* que acumula los impulsos guardando su valor actualizado en una palabra específica en memoria, (c) un *software* que convierte el valor del contador a unidades de tiempo definidas en el SI. Las características más importantes de un reloj son:

- *Precisión*, es decir, cada cuánto llega un nuevo impulso que cambia la cuenta que lleva acumulada, también se conoce como *granularidad* o valor del menor *grano* de tiempo que es capaz de discriminar. La precisión dependerá de la frecuencia de su oscilador y de la forma de detectar y contar los impulsos.
- *Intervalo*: el mayor rango de valores de tiempo que es capaz de medir antes del *desbordamiento*. Esta característica depende directamente de la precisión del reloj y de la capacidad del contador. A mayor precisión, o menor grano de tiempo del reloj, se obtendrá un menor tamaño[2] del intervalo de valores para una capacidad constante del contador.

Dado que la capacidad del contador es limitada, al llegar a un valor determinado de la cuenta de impulsos acumulada se producirá un reinicio de dicha cuenta. A esto se le conoce como *desbordamiento* del contador del reloj de tiempo real. El principal efecto que tiene sobre la medida del tiempo en una aplicación de tiempo real consiste en el establecimiento de dos escalas temporales:

- Tiempo monótono: la *vida* de la aplicación coincide con el valor máximo acumulado en el contador, ya que la aplicación de tiempo real posee un tiempo de ejecución menor que el tiempo de desbordamiento del reloj.

[2] 1ns=$10^{-9}s$,1$\mu\, s = 10^{-6}s$, 1ms=$10^{-3}s$

Precisión	Intervalo
100 ns.	Hasta 429,5 s.
1 μs.	Hasta 71,58 m.
100 μs.	Hasta 119,3 h.
1 ms.	Hasta 49,71 días
1 s.	Hasta 136,18 años

Tabla 4.2: Tamaño del intervalo vs.precisión para un contador de 32 bits.

- Tiempo no monótono: en este caso, como se puede ver en la figura 4.1, la aplicación dura más que el tiempo de desbordamiento, pudiéndose obtener el tiempo total como la suma de varias cuentas del contador.

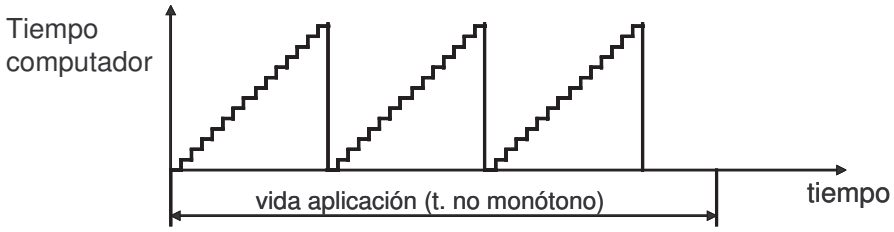


Figura 4.1: Representación del tiempo no-monótono en un reloj de tiempo real.

Los problemas que se derivan de la utilización de una escala de tiempo no–monótono se concretan en que no se dispone de una medida de tiempo *absoluta*, es decir, con un único origen temporal, y no se puede mantener la fecha, la hora, etc., ya que se reinicia el valor del contador varias veces. Tampoco se pueden utilizar intervalos temporales que se extiendan más allá del valor de desbordamiento del contador del reloj.

Temporizadores y retardos

Un temporizador (o *"timer"*) es un tipo de reloj especializado. Los sistemas operativos a menudo utilizan un único temporizador–hardware para implementar un conjunto extensible de temporizadores-*software*. En este escenario, la rutina de servicio de interrupción del reloj hardware manejaría el funcionamiento y la gestión de tantos temporizadores–software como hicieran falta, y su plazo se ajustaría para expirar cuando el siguiente temporizador–software haya de hacerlo.

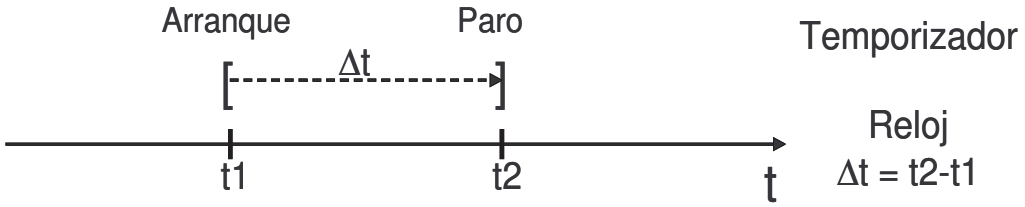


Figura 4.2: Temporizador–software programable

En cada expiración del plazo del temporizador–hardware se comprueba si ha vencido el plazo del siguiente temporizador–software, así como se iniciarían sus acciones pendientes.

Los temporizadores pueden ser utilizados para controlar la secuencia de actuación de un proceso o las consecuencias de la aparición de un evento del entorno de una aplicación de tiempo real. Su utilidad fundamental en las aplicaciones es la de medir intervalos temporales. Para programar un temporizador–software hay que indicar sus tiempos de *arranque* y *parada*, ver figura 4.2. Los temporizadores pueden ser de un solo disparo o periódicos. Los temporizadores de un solo disparo interrumpen una única vez, y después se paran definitivamente. Los temporizadores periódicos interrumpen cada vez que se alcanza un valor temporal específico. Esta interrupción es recibida a intervalos regulares desde el temporizador–hardware. El manejo de los temporizadores parece algo sencillo, sin embargo, hay que tener cuidado con aspectos tales como la *deriva* o las interrupciones retrasadas, que han de ser minimizados, cuando se implementan temporizadores–software, o de otra forma no conseguiríamos precisión en la activación de las acciones de la aplicación que dependan de temporizadores.

Los *retardos* permiten controlar el tiempo de activación de las tareas de tiempo real de una aplicación. Pueden programarse utilizado temporizadores, a nivel de sistema operativo; o bien, mediante instrucciones de los lenguajes de programación. En este último caso se conseguirá una máxima *transportabilidad*[3] del tiempo de retraso especificado. La forma de programarlos sería similar a: delay < duración >; , o bien, con más precisión: *nanosleep*< duración >; . Su efecto consiste en suspender la ejecución de la tarea durante, al menos, la duración especificada desde el momento en que se produjo la llamada a la instrucción anterior. La ejecución puede verse retrasada durante un tiempo mayor que el especificado en la llamada debido a una menor precisión del reloj, que la necesaria para servir la llamada con exactitud; o bien debido a que se activa una tarea más prioritaria cuando termina el tiempo del retardo y la tarea retrasada se ve pospuesta en el acceso al procesador.

La programación con retardos de tareas que se activan periódicamente podría causar la *deriva acumulativa* de la activación de las tareas en cada nuevo ciclo. Esto es debido a que los retrasos que se producirían en cada periodo, es decir, la *deriva local*, se van acumulando.

```
tarea periodica::
  periodo= 100; //milisegundos
  do {
    delay periodo; // en cada ciclo produce un retardo = 'periodo'
    // accion a realizar
  while (true);
  }
```

Figura 4.3: Tarea periódica afectada de deriva local

Se puede conseguir la eliminación de la deriva acumulativa de la programación de tareas periódicas haciendo que, en cada ciclo, la tarea se retrase hasta el siguiente instante de activación. De esta forma, cada activación de la tarea puede significar un tiempo efectivo de retraso diferente, ya que se descontarán los tiempos durante los cuales la tarea está desplazada del acceso al procesador por causas ajenas al propio retardo.

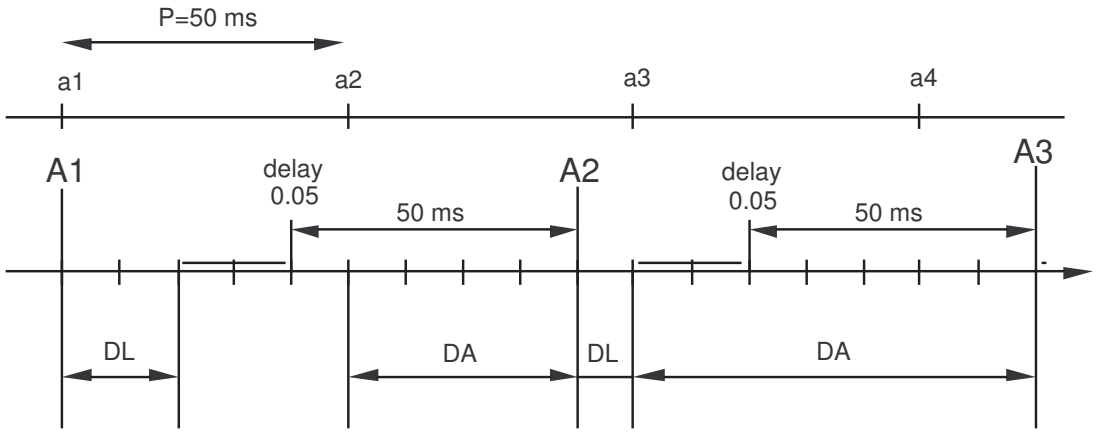
Tal como se puede ver en la siguiente figura, si en cada ciclo la tarea periódica experimenta un retraso constante ("primera aproximación"), la deriva acumulativa (DA) se irá incremen-

[3]es decir, para el código programado, se obtendría el mismo retraso, independientemente del sistema o de la plataforma de ejecución de la aplicación

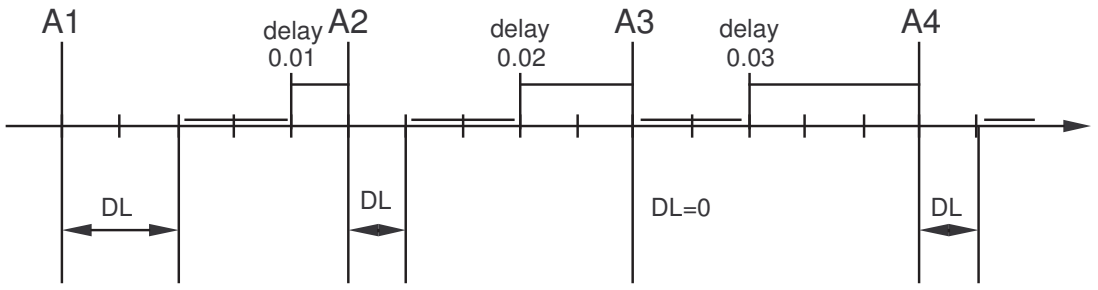
```
tarea periodica::
  periodo= 100; //milisegundos
  siguiente_instante= 0; //milisegundos
  ...
  siguiente_instante= clock();//funcion del sistema
  do {
    delay (siguiente_instante - clock());
    // accion a realizar
    siguiente_instante += periodo;
  while (true);
  }
```

Figura 4.4: Tarea periódica con deriva local eliminada

tando. Si, por el contrario, los retrasos son variables, para ajustarse al siguiente instante de expiración del retardo, entonces se consigue eliminar la deriva acumulativa ("aproximación buena").



Primera aproximación



Aproximación buena

Figura 4.5: Derivas en las tareas con retardos.

Tiempos límite de espera

A veces resulta necesario limitar el tiempo durante el cual una tarea espera que ocurra algún evento. Existen múltiples estados de las tareas durante los cuales éstas quedan suspendidas, por ejemplo, cuando solicita la realización de una operación de entrada/salida u otro servicio del sistema operativo, que se realiza de una forma síncrona con la ejecución de dicha tarea. En estos casos, si la tarea que coopera con la primera para realizar el servicio, o bien el dispositivo, fallasen, entonces la tarea suspendida no volverá a estar activa nunca más. La solución para evitarlo es programar la operación sujeta a un *tiempo límite de espera* (o "timeout"), que indicará el tiempo máximo durante el cual permanecerá suspendida. Si transcurrido dicho tiempo no se realiza el servicio, entonces la tarea suspendida vuelve, levantándose una excepción que puede ser utilizada para presentar un mensaje de aviso o devolver un código de error.

4.1.3 Modelo de tareas

Para poder analizar de una forma comprensible el comportamiento de una tarea durante su ejecución, en el peor caso posible de planificación[4], dentro un programa o aplicación de tiempo real arbitrariamente compleja, es necesario imponer algunas restricciones a su estructura e interacciones con el resto de tareas del programa. Las tareas de tiempo real que son conformes con las restricciones impuestas se dice que cumplen con el *modelo de tareas simple* [Buttazzo, 2005].

Aunque es un modelo básico, el modelo de tareas siemple tiene la capacidad descriptiva suficiente para que se puedan modelar esquemas de planificación estándar, tales como el basado en asignación estática de prioridad, que es el más utilizado en el desarrollo y análisis de STR hasta la fecha.

Características del modelo simple

Consideramos un programa de tiempo real como un conjunto fijo[5] de tareas, que se ejecutan compartiendo el tiempo de un solo procesador, es decir, (*concurrentemente*).

Las tareas son periódicas, con periodos conocidos e independientes entre sí. No existen semáforos, objetos compartidos, etc. que pudieran bloquear a una tarea más prioritaria debido a que el recurso que pretende bloquear ha sido ya adquirido por otra, posiblemente de menor prioridad que ella.

Todas las tareas poseen un tiempo límite (*deadline*), que se considera igual a su periodo. Una tarea, por tanto, está obligada a terminar completamente su ejecución antes de la siguiente activación, lo que ocurrirá cuando transcurra un tiempo igual a su periodo desde el instante de inicio de su activación actual.

Las sobrecargas o retrasos que pueda experimentar el sistema, por ejemplo, tiempos de cambio de contexto, etc. son ignorados. Suponemos que nada impide a una tarea en estado

[4]esto es, cuando sufre una mayor interferencia por parte de procesos más prioritarios
[5]durante la ejecución del programa no se crean ni se destruyem tareas

ejecutable obtener el procesador si en un determinado momento de la aplicación pasa a ser la tarea más prioritaria.

Los eventos no son almacenados, es decir, se pierden si no se atienden y el tiempo máximo de cómputo que una tarea necesita para ser procesada es fijo y conocido a-priori para cada proceso. Se denota con el símbolo `C`, el cual representa unidades de tiempo, y se le denomina "tiempo de ejecución de peor caso" (o WCET, según su acrónimo en inglés).

En este modelo no se contempla a las tareas esporádicas, que ocurren ocasionalmente durante la ejecución de una aplicación de tiempo real y que suelen tener una gran urgencia cuando se activan.

Atributos temporales asociados a una tarea

Una tarea de tiempo real τ_i puede ser caracterizada mediante un conjunto específico de atributos temporales.

En la figura 4.6 se puede observar una representación gráfica de algunos de los atributos anteriores, que se consideran como elementos característicos dentro de un esquema de planificación de tareas en un STR.

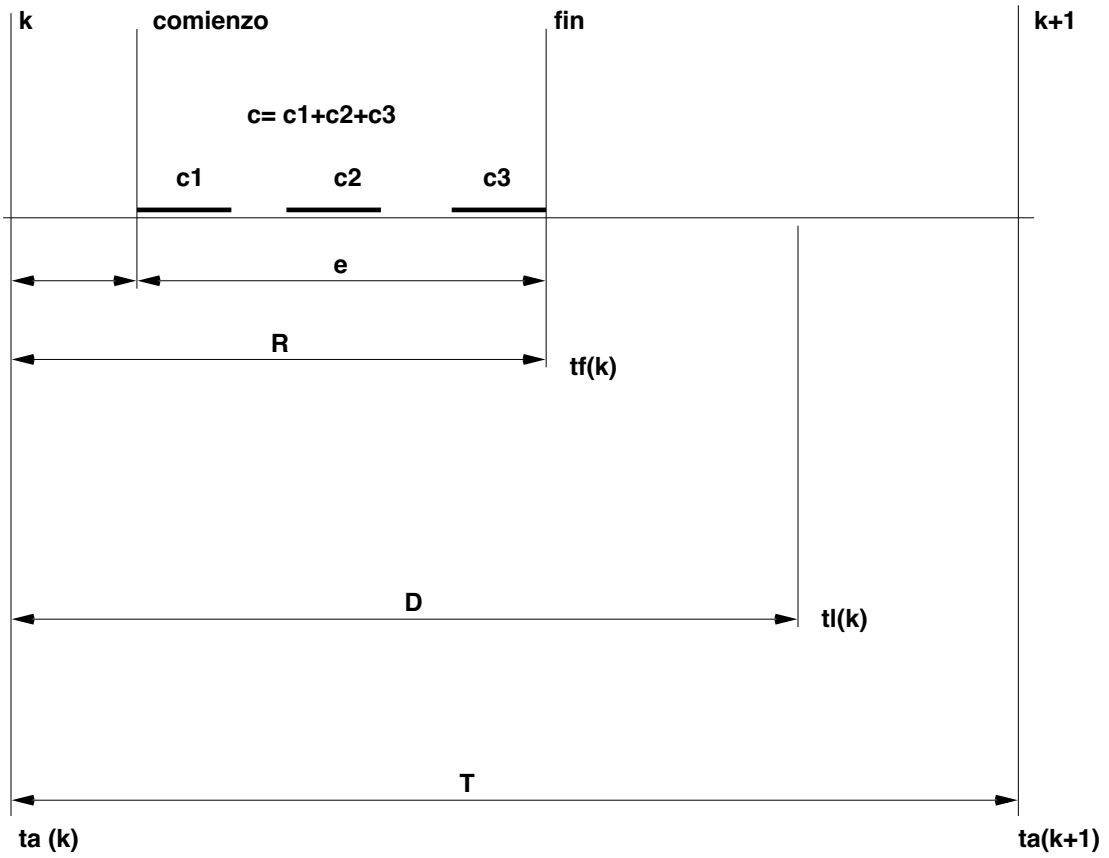


Figura 4.6: Representación de los atributos temporales de las tareas de tiempo real.

Notación	Atributo temporal	Descripción
P	Prioridad	Prioridad asignada al proceso (si fuera aplicable)
τ	Tarea	Nombre de la tarea
t_a	Instante o tiempo de activación de la tarea (arrival time, request time, release time)	Instante en el que la tarea está lista para su ejecución.
t_s	Instante o tiempo de comienzo (start time)	Instante de tiempo en el que la tarea comienza realmente su ejecución.
t_f	Instante o tiempo de finalización de la tarea (finishing time)	Instante en el que la tarea finaliza su ejecución.
t_l, d	Instante o tiempo límite (absolute deadline)	Instante de tiempo límite para la ejecución de la tarea. Es un valor fijo dado por $t_l(k) = t_a + D$
T	Periodo de ejecución	Intervalo de tiempo entre dos activaciones sucesivas de una tarea periódica. Es un valor fijo, dado por $T = t_a(k+1) - t_a(k)$
J	Latencia	Intervalo de tiempo desde que se activa la tarea hasta que se ejecuta. Viene dado por: $J(k) = t_s(k) - t_a(k)$ y, dependiendo de la sobrecarga del sistema, varía entre un valor mínimo de ($J_{mín}$) y máximo de (J_{max})
c	Tiempo de cómputo	Tiempo de ejecución del proceso
C	Tiempo de cómputo máximo	Tiempo de ejecución del proceso en el peor caso posible.
e	Tiempo de ejecución transcurrido	Tiempo transcurrido desde el instante de comienzo hasta la finalización del proceso. Viene dado por: $e(k) = t_f(k) - t_s(k)$
R	Tiempo de respuesta	Tiempo que ha necesitado el proceso para completarse totalmente. Es variable en cada activación de la tarea y viene dado por: $R(k) = J(k) + e(k)$.
D	Plazo de respuesta máximo (relative deadline)	Define el máximo intervalo de tiempo o máximo tiempo de respuesta hasta completar la ejecución de la tarea.
Φ	Desplazamiento o fase	Tiempo necesario para activar por primera vez una tarea periódica.
RJ	Fluctuación relativa o *jitter* (relative release jitter)	Máxima desviación en el tiempo de comienzo entre dos activaciones sucesivas de una tarea. Viene definido por: $RJ = \max((t_s(k+1) - t_a(k+1)) - (t_s(k) - t_a(k)))$
AJ	Fluctuación absoluta	Máxima desviación en el tiempo de comienzo de todas las activaciones de una tarea. Viene definido por: $AJ = J_max - J_min$
L	Retraso(lateness). Existe también el tiempo de *exceso*, dado por: $E(k) = \max(0, L(k))$	Retraso de la finalización del proceso respecto del tiempo límite. Viene definido por: $L(k) = t_f(k) - t_l(k)$. Si la tarea se completa antes del tiempo límite su valor es 0.
H	Holgura (laxity, slack time)	Tiempo máximo que una tarea puede permanecer activa dentro del plazo de respuesta máximo. Se define como: $H(k) = t_l - t_a(k) - c(k) = D - c(k)$

4.2 Planificación de tareas periódicas con asignación de prioridades

De forma general se puede definir la *planificación* de actividades o tareas como un área del conocimiento humano que estudia un conjunto de algoritmos y técnicas de *programación entera* cuyo objetivo es conseguir una asignación de recursos y tiempo a actividades, de forma que se cumplan determinados requisitos de eficiencia. La estrategia básica para resolver un problema de planificación de recursos suele ser utilizar una heurística que intenta maximizar una función objetivo.

En el caso de los sistemas de tiempo real, el principal recurso a asignar es normalmente el tiempo del procesador. Se suele requerir, además, que sea posible determinar *a-priori* si las tareas de un programa terminan, en todas sus activaciones, antes de que se alcancen sus tiempos límite, aun en el peor caso de planificación posible de dichos procesos.

La determinación de la planificabilidad de un conjunto de procesos se lleva a cabo utilizando un *esquema de planificación de procesos* que ha de contener los siguientes elementos:

1. Un algoritmo para ordenar el acceso de las tareas a los recursos del sistema.

2. Una forma de predecir el comportamiento del sistema en el peor de los casos.

Existen diferentes tipos de esquemas de planificación de tareas de tiempo real. Se habla, por ejemplo, de esquema de planificación *estático* cuando el orden de planificación de las tareas es fijo y puede ser, por tanto, determinado a-priori. Por contra, hablaremos de un esquema de planificación *dinámico* cuando la prioridad de los tareas varíe a lo largo de la ejecución. Con un esquema estático simplificamos el problema de la planificación de tareas de tiempo real, sin perder generalidad. Este modelo de tareas es seguido por la mayoría de la aplicaciones de tiempo real en sistemas muy críticos, como los que se utilizan en el control de vuelo de un avión comercial o en radio–medicina.

En modelos más elaborados de tareas habría que tener en cuenta a las tareas no–periódicas, ya que cuando ocurren suelen ser tareas muy urgentes y críticas para la seguridad del STR. De acuerdo con el modelo simple de tareas, no se van a tratar las tareas de tipo *esporádico* o aperiódico. Es decir aquellas tareas que cuando se activan tienen un tiempo límite mucho menor que su periodo o que carecen en absoluto de un patrón periódico de activación, respectivamente.

Se supone un esquema de planificación expulsivo (*preemptive*), es decir, se produce siempre un *desplazamiento* del procesador de las tareas menos prioritarias por parte de los más prioritarias. Para determinar si las tareas $\tau_1\ \tau_2\ \ldots\ \tau_n$, con periodos $T_1\ T_2\ \ldots\ T_n$, pueden ser totalmente ejecutadas dentro de sus plazos temporales, la línea temporal del gráfico de planificación habría de cubrir un periodo de tiempo de *al menos* igual al **m.c.m.**$(T_1\ T_2\ \ldots\ T_n)$. Aunque sólo será necesario analizar una línea temporal de longitud correspondiente al mayor T_i si se supone que todas las tareas se inician a la vez. Este instante representa un momento de la *máxima carga* posible del procesador; por tanto, se le denomina (*instante crítico*).

Conforme progresa la aplicación, las tareas se ejecutan en el orden dado por su prioridad. Cuando se utiliza este esquema, los parámetros que influyen en la planificación de las tareas se

tienen que fijar *estáticamente*. Por tanto, el máximo tiempo de ejecución de las tareas es fijo, así como lo son también sus prioridades asignadas.

La *prioridad* es un número positivo. Normalmente se toma la convención de asignar números enteros menores a los procesos más prioritarios. La prioridad de una tarea se determina a partir de su periodo, tiempo límite, etc., o cualquier otro atributo que se mantenga durante toda su ejecución.

4.2.1 Algoritmo de cadencia monótona

Aquí nos centraremos en un esquema de planificación estático para tareas periódicas, en el que la prioridad de una tarea de tiempo real sólo dependerá de su periodo. Es decir, se asignarán inicialmente las prioridades a las tareas de la aplicación en el orden dado por su menor frecuencia de activación. Las tareas con periodos de activación más cortos van a ser las más prioritarias, independientemente de su criticidad respecto de la aplicación a la que pertenecen. Matemáticamente, podemos decir que dichas prioridades se asignan mediante una función monótona de la cadencia temporal de los procesos periódicos: $T_i < T_j \Rightarrow P_i > P_j$, de ahí el nombre del algoritmo.

El algoritmo de *cadencia monótona* es óptimo entre los algoritmos de asignación estática de prioridades, esto es, un conjunto de procesos planificable con cualquier esquema de asignación fija de prioridades resultaría también planificable si en lugar de este se utilizase el algoritmo de cadencia monótona.

El algoritmo, propuesto en un trabajo histórico por Liu y Layland en 1973, es la base para desarrollar una teoría matemática de planificación de tareas de tiempo real. Dicha teoría dista mucho de ser algo exclusivamente teórico, ya que tiene en cuenta aspectos prácticos de la planificación de tiempo real impuestos por la cooperación con la industria norteamericana. De hecho, gran parte de los resultados se obtuvieron como consecuencia de la colaboración entre 3 instituciones: Software Engineering Institute (SEI), IBM y Carnegie Mellon University.

4.2.2 Tests de planificabilidad

Para poder determinar, antes de su ejecución, si un conjunto de tareas periódicas es planificable utilizando el algoritmo de cadencia monótona es necesario contar con criterios que permitan predecirlo. Dichos criterios se establecen en función de la utilización del procesador (U) por parte del conjunto de tareas, o bien calculando el tiempo de respuesta de cada tarea (R_i), los cuales pueden ser calculados estáticamente.

La obtención de tests de planificabilidad para un conjunto de tareas con prioridades asignadas estáticamente no ha sido fácil. De hecho, hasta muy recientemente, no se han descubierto condiciones suficientes y necesarias para determinar la planificabilidad de las tareas de un programa de tiempo real arbitrario.

Se va a comenzar estudiando algunas *condiciones necesarias*, que intuitivamente deberían cumplirse, para poder afirmar que un conjunto de tareas es planificable. Dichas condiciones

pueden utilizar datos estáticos de las tareas: peor tiempo de ejecución, periodo, etc. para poder ser definidas.

Condiciones necesarias de planificabilidad

1. El tiempo de ejecución en el peor de los casos de cualquier tarea ha de ser menor que su periodo.

   $$\forall\ i\ C_i\ <\ T_i$$

   Esta condición no es *suficiente*, ya que alguna de las otras tareas más prioritarias puede no haber terminado cuando llegue su tiempo límite. Por ejemplo, considérense:

-	Prio	T_i	C_i
τ_1	2	10	8
τ_2	1	5	3

   En el conjunto de tareas anterior, τ_2 perderá su primer tiempo límite en t=5. Sin embargo, se cumple el criterio 1.

2. La utilización del procesador por unidad de tiempo de todas las tareas con una prioridad superior a la del *nivel i* no puede superar la unidad. $\sum_{j=1}^{i}\ \frac{C_j}{T_j}\ \leq\ 1$. Esta condición no es suficiente, ya las tareas más prioritarios pueden interferir varias veces y hacer perder el límite de tiempo al proceso de nivel de prioridad i. Compruébese esto para el siguiente conjunto de tareas:

-	Prio	T_i	C_i
τ_1	2	6	3
τ_2	2	9	2
τ_3	1	11	2

   A τ_3 le faltaría por ejecutar una unidad de tiempo cuando ocurre el siguiente evento de activación y, como $\mathrm{D_i} = \mathrm{T_i}$, pierde su tiempo límite.

3. El procesamiento de todas las invocaciones en los niveles de prioridad mayores que i: 1, 2, ..., $(i-1)$, debe completarse como mucho en el tiempo $T_i\ -\ C_i$. Con referencia al ejemplo anterior, esto supondría que todas las activaciones de las tareas τ_1 y τ_2 que interfieren a τ_3 deberían completarse antes de 9 unidades de tiempo: $\forall\ i\ \ \sum_{j=1}^{i-1}(\frac{T_i}{T_j} \times C_j) \leq\ T_i\ -\ C_i$. Esta condición no es suficiente, ya que si $T_j\ >\ T_i$, entonces degenera en la primera condición (téngase en cuenta que las divisiones son enteras).

4. Para evitar que se puedan anular términos de la sumatoria que aparecen en la condición anterior, se amplia la *ventana temporal* de un periodo $\mathrm{T_i}$ a un marco temporal M_i = **m.c.m.**$\{T_1\ T_2\ ...\ T_i\}$, de esta forma se obtiene ahora el número exacto de activaciones de cada tarea T_j, más prioritaria que T_i, en el periodo de tiempo M_i, como el resultado de la fracción entera: $\frac{M_i}{T_j}$. La nueva condición, que adquiere el nombre de *razón de carga*, se expresa ahora como: $\forall\ i\ \ \sum_{j=1}^{i}(\frac{M_i/T_j}{M_i} \times C_j) \leq\ 1$. Sin embargo, aunque discrimina más que la condición 3, tampoco es una condición suficiente. Ya que si el tiempo de computación de una tarea τ_i excede al periodo de activación de otra τ_j, dado $i\ <\ j$,

entonces no sería factible la planificación conjunta ambas. Compruébese para el siguiente conjunto:

-	Prio	T_i	C_i
τ_1	2	12	5
τ_2	1	4	2

Condiciones suficientes de planificabilidad

Teorema 1 (Liu y Layland):

En un sistema de N tareas periódicas independientes con prioridades asignadas en orden de frecuencia[6], se cumplen todos los plazos de respuesta, para cualquier desfase inicial de las tareas, si:

$$\sum_{i=1}^{N}(\frac{C_i}{T_i}) \; < \; N\,(2^{\frac{1}{N}} \; - \; 1)$$

Si una tarea pasa el test anterior, entonces completará sus C_i unidades de cómputo antes de que expire el tiempo límite ($D_i = T_i$) en cada ciclo de activación de la tarea. Aun si falla el test anterior, podrían satisfacerse los tiempos límite de las tareas del conjunto anterior, porque se trata de una *condición suficiente*, pero no necesaria, de planificabilidad. Para comprobarlo se puede realizar un diagrama de ejecución en función del tiempo de todas las tareas, también denominado diagrama de Gantt. Si ninguna de las tareas perdiese su tiempo límite dentro de una ventana temporal dada por el $m.c.m\ de\ :\ T_1, T_2, ...T_i$ entonces podríamos afirmar la planificabilidad del conjunto, incluso si no se cumpliera el teorema anterior. El problema con esto es que la verificación de la planificabilidad de un conjunto grande de tareas en aplicaciones complejas es infactible salvo si cuenta con herramientas gráficas apropiadas, que suelen ser muy costosas.

A continuación, se puede ver una tabla con los límites de utilización del procesador para diferentes números de tareas:

N	límite utilización
1	100
2	82,85%
3	78,0%
4	75,7%
5	74,3%
10	71,8%
$\rightarrow \infty$	69,3%

[6]esto es, con el *algoritmo de cadencia monótona*

Por tanto, siempre que el límite de utilización del procesador (U), por parte de un conjunto de tareas, no supere el 69,3%, dicho conjunto será planificable utilizando un esquema de planificación expulsivo basado en la asignación estática de prioridades, dado por el algoritmo de la cadencia monótona. El test de Liu-Layland no es *exacto*, ya que dicho test, basado en la *utilización del procesador* (U), no es capaz de determinar la planificabilidad en algunos casos de un conjunto de tareas. El conjunto de tareas siguiente no pasa el test de Liu-Layland, ya que la utilización del procesador sería: $\Sigma_{i=1}^{3}(\frac{C_i}{T_i}) = .92857$ y, sin embargo, el test exige una utilización máxima del procesador $3 * (2^{\frac{1}{3}} - 1) = .77976$ como condición suficiente para asegurar la planificabilidad. Sin embargo, en la realidad, todos las tareas consiguen alcanzar sus tiempos límite:

-	Prio	T_i	C_i
τ_1	1	7	3
τ_2	1	12	3
τ_3	2	20	5

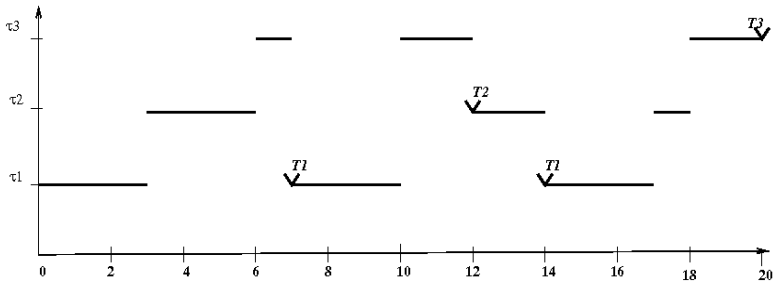


Figura 4.7: Conjunto planificable que no cumple el test de Liu

Además, el test de Liu no es aplicable, sin modificación, si se considera un modelo de tareas más general en el cual pueda existir aperiodicidad, bloqueos de tareas prioritarias, etc.

Los tests basados en la *utilización del procesador* no dan información acerca de los tiempos de respuesta de los procesos. De ahí que se propongan nuevos tests exactos, es decir, que proporcionan condiciones necesarias y suficientes de planificabilidad, basados en el cálculo de dichos tiempos de respuesta de las tareas de tiempoo real. El inconveniente consiste en su aplicación práctica porque el cálculo para un conjunto arbitario de tareas, que se pueden interrumpir unas a otras varias veces durante su ejecución, no tiene, en general, una solución matemática sencilla.

Teorema 2 (Liu y Layland):

> En un sistema de N tareas periódicas independientes con prioridades estáticas, se cumplen todos los tiempos límite, para cualquier desfase inicial de las tareas, si cuando se activan todas ellas simultáneamente, cada tarea acaba antes de que expire su tiempo límite en su primera activación.

4.2.3 Test de planificabilidad basado en la utilización para EDF

En el trabajo de Liu y Layland se propuso también un test para un esquema de asignación de prioridades basado en la *proximidad del tiempo límite* (EDF) de las tareas:

$$\Sigma_{i=1}^{N}(\frac{C_i}{T_i}) \leq 1$$

Si el límite de utilización derivado de la ejecución del conjunto de tareas es menor que la capacidad de cómputo total del procesador, entonces podemos afirmar que, para el modelo de procesos simple, se cumplirán todos los tiempos límite de las tareas.

El esquema de planificación EDF se considera dinámico, porque la prioridad de las tareas cambia conforme se acerca su tiempo límite durante la ejecución. EDF es superior a uno basado en la asignación estática de prioridades, ya que siempre que se pueda planificar un conjunto de tareas con un esquema estático, también se podrá con el EDF, pero no se cumple la inversa. No obstante, se prefieren los esquemas de planificación estáticos, ya que:

- Un esquema estático es más sencillo de implementar. Un esquema dinámico, como el EDF, induce mayor sobrecarga al sistema.
- Resulta más sencillo el incorporar tareas sin tiempos límite definidos en un esquema estático.
- El atributo de tiempo límite suele no ser el único parámetro de planificación a considerar en aplicaciones realistas. Resulta más sencillo incorporar otros factores que influyen en la planificación a la noción de prioridad cuando ésta no está asociada a un tiempo límite.
- Durante situaciones de sobrecarga transitoria un esquema de planificación estático resulta ser más predecible[7]. Además se suelen conseguir mayores límites de utilización del procesador con un esquema estático.

4.3 Modelos generales y específicos de tareas

El modelo simple ha de ser extendido para poder incluir los requisitos de planificación de las tareas esporádicas y aperiódicas, así como los bloqueos que sufren las tareas cuando intentan obtener recursos compartidos a los que acceden en exclusión mutua. Ya que en el modelo simple de tareas se considera a éstas totalmente independientes durante toda su ejecución.

Los tareas periódicas se ejecutan tras producirse eventos de activación locales. Las *tareas esporádicas* se ejecutan tras producirse un evento que se origina en un procesador remoto. El periodo T_i de una tarea τ_i esporádica se define como el mínimo[8] intervalo temporal medido entre dos eventos de activación sucesivos. El periodo de una aperiódica es el intervalo temporal promedio entre 2 eventos cualesquiera de activación sucesivos. Las tareas esporádicas, cuando

[7] las tareas con menor prioridad son las que pierden sus tiempos límite; sin embargo, esto no es necesariamente así si se utiliza un esquema EDF

[8] una tarea esporádica con $T_s = 20$ms. garantiza que no será activada más de 1 vez cada 20 ms. El intervalo entre 2 activaciones podría ser mayor algunas veces, pero ha de contemplarse el *peor caso* de planificación.

se presentan, suelen ser urgentes y tienen límites de tiempo estrictos mucho menores que su periodo de activación ($D_i << T_i$). Las tareas *aperiódicas* no tienen definido ningún tiempo límite estricto, como ocurre con las *esporádicas*. Suelen tener límites de tiempo permisivos, es decir, se admite que pierdan alguno de estos; a diferencia de las tareas esporádicas, en las cuales la pérdida de algún tiempo límite no sería admisible.

En aplicaciones de tiempo real que posean tareas esporádicas no se puede utilizar directamente el test de Liu para decidir la planificabilidad de un grupo de tareas. Ya que la utilización del tiempo del procesador, como criterio de planificabilidad de las tareas, suele propocionar un resultado demasiado "pesimista" cuando se le aplica a las esporádicas. Para este tipo de tareas se define una cadencia máxima y promedio de llegada de sus eventos de activación. El considerar el valor máximo suele conducir a la imposición de utilizaciones muy bajas del tiempo del procesador para garantizar la condición de planificabilidad del conjunto de tareas esporádicas.

4.3.1 Plazos de respuesta menores que el periodo

Para valores del tiempo límite que coincida con el periodo de las tareas, $D_i = T_i$, la asignación de prioridades según el *algoritmo de la cadencia monótona* resulta ser óptima para cualquier esquema de planificación estático. De forma análoga, el criterio de ordenación de las prioridades de las tareas atendiendo al menor valor de sus tiempos límite (*Deadline Monotonic Priority Ordering*) es *óptimo*[9] para conjuntos de tareas en los que se cumpla $D_i < T_i$. Un ejemplo en el que se cumple la afirmación anterior puede verse en la siguiente tabla:

τ	T	D	C	P	R
a	20	5	3	1	3
b	15	7	3	2	6
c	10	10	4	3	10
d	20	20	3	4	20

La ordenación de prioridades basada en el algoritmo de la cadencia monótona no puede asegurar una planificación del conjunto de tareas anterior, tal que se cumplan todos los tiempos límite de las tareas. Sin embargo, este conjunto de tareas nos saldría planificable si se asignan las prioridades con DMPO, tal como se hace en la 5ª columna de la tabla.

La demostración de la optimalidad de DMPO implica transformar las prioridades de cualquier esquema estático hasta que se obtiene la ordenación dada por DMPO. El esquema que se sigue en la demostración es el siguiente, aunque no se va a desarrollar completamente debido a su complejidad. Se parte de una ordenación de las prioridades a las tareas, dada, por ejemplo, por el algoritmo de cadencia monótona.

- Cada paso del procedimiento de transformación ha de preservar la planificabilidad de los procesos.
- Se intercambian tareas con prioridades adyacentes.

[9]DMPO es un criterio óptimo, ya que cualquier conjunto de tareas que sea planificable, atendiendo a un esquema de asignación de prioridades estático, también lo sería con este criterio.

- La tarea que disminuya su prioridad sigue siendo ejecutable.
- La tarea que aumenta su prioridad sólo interfiere a la otra.
- Como su tiempo límite es el menor de los dos, la otra tarea sigue siendo planificable después del intercambio.

Al final de proceso anterior se obtiene una ordenación de las tareas según la prioridad dada por el DMPO. Como durante la transformación del esquema inicial (cadencia monótona) al final (DMPO) no ha dejado de ser ninguna de las tareas planificable, se concluye que cualquier esquema de asignación de prioridades a las tareas estático es equivalente a DMPO, que se puede considerar óptimo entre estos.

4.3.2 Interacciones entre las tareas

Las entidades que constituyen un programa de tiempo real no son únicamente *tareas* que se ejecutan de forma asíncrona y sin interaccionar entre ellas, sino también existen *objetos protegidos*, es decir: secciones críticas, monitores, semáforos, etc., que proporcionan acceso en exclusión mutua a datos compartidos entre las tareas de la aplicación.

La utilización de objetos protegidos en los programas de tiempo real lleva a la posibilidad de que una tarea pueda ser bloqueada esperando a que se produzca algún evento futuro, distinto del de activación, que le permita continuar con su ejecución. Por ejemplo, una tarea podría quedar bloqueada esperando le ejecución de una operacion *signal*() de un semáforo por parte de otra tarea del programa. También, las tareas podrían entrar en una cola de un monitor hasta la ejecución de la operación *c.signal*(). O bien que la tarea realice una llamada a una cita, etc. Las *tareas* y los *objetos protegidos* pueden estar distribuidos a lo largo de un sistema físico con varios procesadores, por lo que se pueden producir esperas debidas a recepción de mensajes, acceso a buses, switches en multiprocesadores, etc.

El problema que surge si una tarea permanece bloqueada esperando a que otra menos prioritaria termine de ejecutar una sección de su código, es que deja de cumplirse una de las condiciones más importantes del modelo simple de tareas: *cuando una tarea tiene prioridad suficiente para ejecutarse, ha de hacerlo.* Según el modelo simple, en ningún caso podría verse bloqueada una tarea, o suspendida su ejecución durante algún tiempo, ya que esto influiría en la planificación del resto de las tareas y podría llegar a ocasionar la pérdida de sus tiempos límite. Por tanto, no sería de aplicación el análisis de planificabilidad basado en el Test de Liu-Layland, para el modelo simple de tareas, sin cuantificar cómo se vería afectada la desigualdad $\Sigma_{i=0}^{N} \frac{C_i}{T_i} \leq U_0(N)$ con el eventual bloqueo de las tareas al acceder a recursos comunes.

Inversión de prioridad

Podría suceder, incluso, que la espera de la tarea más prioritaria llegase a ser arbitrariamente larga si se ejecutan continuamente tareas menos prioritarias mientras que ésta se mantiene bloqueada. En ese caso, se produciría lo que se denomina una *inversión de prioridad* que

invalidaría cualquier previsión acerca de la planificabilidad de un conjunto de tareas (ver figura 4.8[10]). La inversión de prioridad no puede ser eliminada completamente si se utilizan objetos protegidos en los programas de tiempo real, pero los efectos adversos sobre la planificación de las tareas más prioritarias pueden ser minimizados, haciendo que el bloqueo derivado del acceso a dichos objetos sea siempre un tiempo acotado y medible en cualquier ejecución de la aplicación.

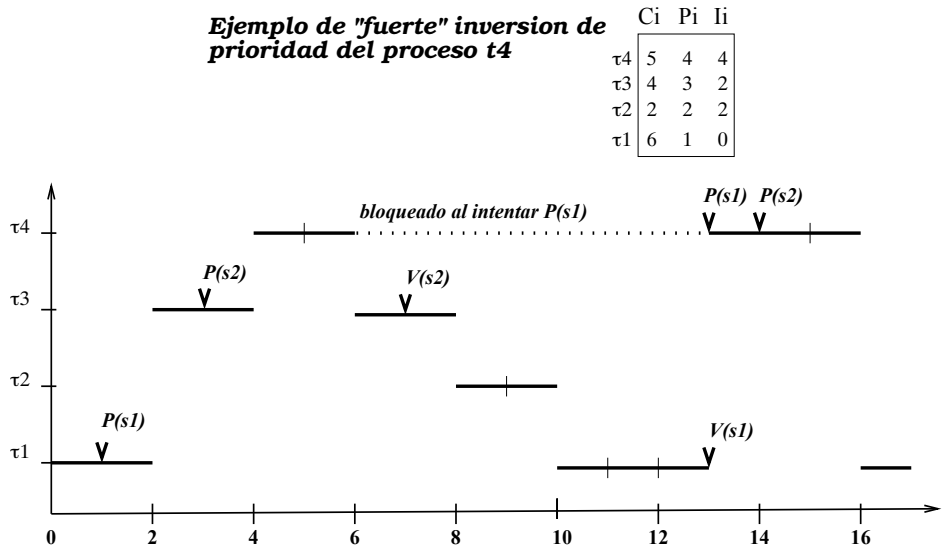


Figura 4.8: Inversión de prioridad de una tarea

Se puede decir que la *inversión de prioridad* es un inconveniente que se produce debido a un esquema estático de asignación de prioridades. En el ejemplo se puede observar como la tarea τ_4, la más prioritaria de todas, sufre una fuerte inversión de prioridad por parte de 3 tareas de menor prioridad:

La tarea τ_1 se activa primero, tras ejecutarse un *tick*[11], bloquea el cerrojo del semáforo s_1. Después resulta desplazada, cuando comienza la ejecución de la tarea τ_3, ya que ésta es más prioritaria que τ_1. Ésta última se ejecuta durante 1 *tick* y bloquea el cerrojo del semáforo s_2, después resulta desplazada por el comienzo de la tarea τ_4. La tarea τ_4 se ejecuta hasta que intenta adquirir el cerrojo del semáforo s_1, que lo tiene bloqueado τ_1, entonces τ_4 se bloquea a su vez. τ_3 vuelve a poseer el procesador, cuando termina de ejecutarse, τ_2 ejecuta las 2 unidades de tiempo que le quedan. Cuando ésta termina, puede volver a ejecutarse τ_1, hasta que libera el cerrojo de s_1, en ese momento, resulta ser desplazada por τ_4 que continúa su ejecución hasta terminar. Finalmente, la tarea τ_1 terminará de ejecutarse.

Sección crítica no expulsable

Es el protocolo más simple que se podría imaginar para evitar la inversión de prioridad en la implementación de un conjunto de tareas de tiempo real que comparten un recurso. Consiste en no permitir la expulsión del procesador de ninguna tarea cuando accede al recurso durante

[10] nótese que los números mayores indican más prioridad en la asignación del ejemplo de la figura
[11] unidad arbitraria de tiempo de ejecución

la ejecución de una sección crítica. El resultado de aplicar este protocolo a la ejecución de un conjunto de tareas es equivalente a la ejecución de las secciones críticas con una prioridad estática igual a la de la prioridad máxima del sistema.

Para aplicar este protocolo no se necesita tener un conocimiento previo de los requisitos asociados a los recursos. Sin embargo, puede inducir bloqueos de las tareas más prioritarias excesivamente largos si la duración del tiempo de ejecución de las secciones críticas no está acotado (o no puede acotarse antes de comenzar la ejecución del programa). Además, la puesta en práctica de este protocolo puede interferir en la ejecución de todas las tareas del sistema, aunque no hagan uso de los recursos compartidos. En la figura 4.9 puede verse un ejemplo[12] de aplicación del protocolo.

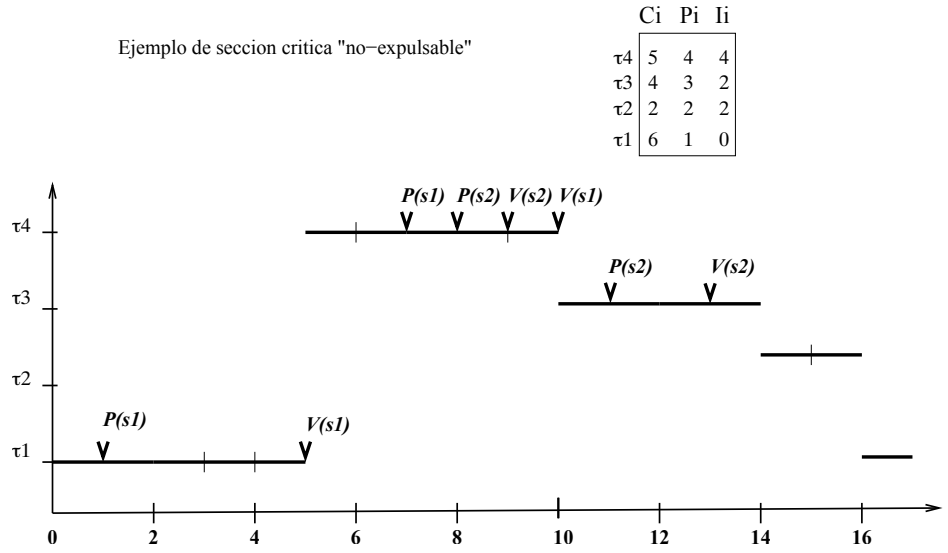


Figura 4.9: Escenario de tareas que usan "sección crítica no expulsable"

Tiempo de bloqueo

Se puede seguir utilizando el test de Liu-Layland si la inversión de prioridad representa un término constante en la desigualdad. En ese caso, se puede obtener, para cada tarea τ_i, la pérdida en la utilización del procesador que supone el que tareas menos prioritarias puedan bloquearle recursos que necesita mediante el cálculo del factor de bloqueo B_i. Luego, podemos aplicar el test de Liu–Layland para la tarea τ_i asumiendo que dicha tarea va a tener ahora un tiempo de ejecución de peor caso incrementado con el valor constante del factor de bloqueo calculado:

$$C_i^* = C_i + B_i$$

El factor de bloqueo se va a determinar considerando el peor caso posible de planificación, es decir, el mayor bloqueo que una tarea prioritaria podría experimentar debido a que comparte con varias tareas menos prioritarias que ella varias secciones críticas. Hay que tener en cuenta que cada protocolo que vamos a proponer para limitar el problema de la inversión de prioridad proporciona factores de bloqueo diferentes.

[12] nótese que los números mayores indican más prioridad en la asignación del ejemplo de la figura

Como con el protocolo de la sección crítica no expulsable una tarea prioritaria puede ser bloqueada como máximo durante la ejecución de una sección crítica. El factor de bloqueo vendrá dado por la siguiente expresión:

$$B_i = max_{j,\, j>i}(max_k(D(s_{jk}))$$

Donde s_{jk} es la sección crítica k ejecutada por la tarea τ_j menos prioritaria que τ_i y $D(s_{jk})$ es la duración de dicha sección crítica. La desigualdad $(j > i)$ referida a los índices de los identificadores de las tareas sirve para definir el conjunto de todas las tareas τ_j menos prioritarias que τ_i, pues suponemos que a las tareas más prioritarias se les asigna un índice menor.

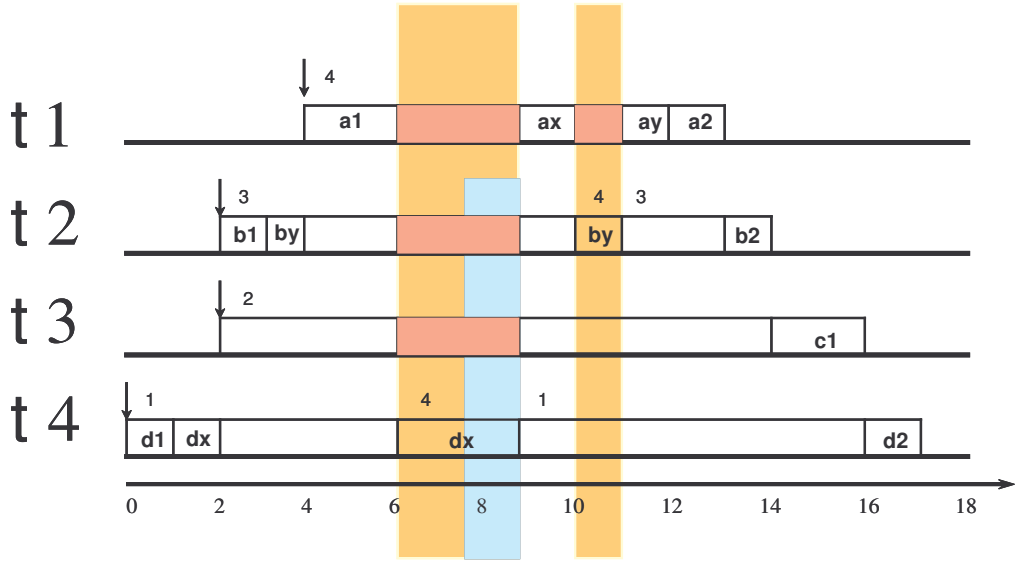


Figura 4.10: Escenario de 4 tareas con el protocolo de *herencia de prioridad*

Herencia de prioridad

Con este protocolo, las prioridades de las tareas no se mantienen estáticas durante toda la ejecución del programa y, como consecuencia de ello, se consigue minimizar el efecto de la *inversión de prioridad* en el aprovechamiento del tiempo del procesador por parte de las tareas.

Si la tarea más prioritaria τ_1 es bloqueada esperando que termine otra τ_2 de menor prioridad de ejecutar la sección crítica de un objeto protegido que comparte con la primera, entonces la prioridad de τ_2 se iguala a la de τ_1 durante el tiempo que τ_2 mantiene el cerrojo bloqueando a τ_1. Por tanto, durante su ejecución la prioridad de una tarea de la aplicación resultará ser el máximo entre su prioridad por defecto[13] y las prioridades de todas las demás tareas que en ese momento mantiene bloqueadas.

En la figura 4.10 se puede ver un escenario de ejecución de 4 tareas que acceden a 2 objetos protegidos, representados por las secciones críticas: $\{X,\ Y\}$. Los atributos de las tareas de la figura vienen dados en la siguiente tabla:

Las tareas pueden sufrir 2 tipos de bloqueos:

- Directo: lo sufre la τ_1 porque las tareas τ_4 y τ_2 la bloquean al acceder a las secciones críticas *dx* y *by*, que comparten con la primera.

[13] esto es, la prioridad con la que accede a la sección crítica

Tarea	P	C	Acciones
τ_1	1	5	a1(2);ax(1);ay(1);a2(1)
τ_2	2	4	b1(1);by(2);b2(1)
τ_3	3	2	c1(2)
τ_4	4	6	d1(1);dx(4);d2(1)

- Indirecto: las tareas τ_2 y τ_3 son bloqueadas por la tarea τ_4 por motivo de que la prioridad de ésta se ve elevada al valor máximo mientras está bloqueando a la tarea τ_1.

Con este protocolo las prioridades de las tareas cambiarán a menudo durante su ejecución, por lo que puede resultar ineficiente el implementarlo utilizando una cola de despacho de procesos ordenada por prioridad. Podría darse más de un bloqueo por inversión de prioridad durante la ejecución de una tarea prioritaria. Además, el protocolo de herencia de prioridad no evita ni el interbloqueo de las tareas en el acceso a recursos ni los bloqueos encadenados o *transitivos*.

Tiempo de bloqueo

La característica fundamental de este protocolo, con respecto a su influencia en el comportamiento dinámico de las tareas del programa, consiste en que las tareas sólo pueden verse bloqueadas un número limitado de veces por otras menos prioritarios. Con este protocolo, al elevarse la prioridad de la tarea que usa el objeto protegido a la de la tarea más prioritaria que esté bloqueando, ocasiona que la primera no pueda ser interrumpida por tareas de prioridad intermedia, que era la causa principal de la *inversión de prioridad*, por tanto:

1. Si una tarea tiene definidas en su código M secciones críticas, entonces el número máximo de veces que puede verse bloqueada durante su ejecución es M.
2. Si hay sólo $N < M$ tareas menos prioritarias, el máximo número de bloqueos que puede experimentar la tarea más prioritaria se reducirá a N.

El factor de bloqueo B_i en el caso del protocolo de herencia de prioridad vendrá dado como la sumatoria de todos los bloqueos que, en el peor de los escenarios de planificación posibles, podrían afectar a la tarea τ_i. Para calcular el factor de bloqueo de la tarea τ_i hay que establecer qué secciones críticas pueden estar en ejecución por otras tareas menos prioritarias cuando se active la primera y cuáles de éstas pueden bloquearla[14]. El factor de bloqueo para este protocolo suele ser difícil de calcular sistemáticamente y con exactitud. Por tanto, los algoritmos para hacerlo suelen proporcionar una estimación del mismo, que consiste en una cota superior del tiempo de bloqueo.

El factor de bloqueo vendrá dado como el mínimo entre 2 terminos a calcular,

1. B_i^l: bloqueo debido a tareas τ_j menos prioritarias, que acceden a secciones críticas k[15]:
   - $B_i^l = \Sigma_{j=i+1}^{n} \max_k [D_{j,k} : \text{Límite}(S_k) \geq P_i]$
2. B_i^s: bloqueo debido a todas las secciones críticas a las que accede la tarea τ_i,

[14] téngase en cuenta que el bloqueo aludido podría ser también del tipo *indirecto* –sin compartir sección crítica– por aumento de prioridad transitorio de las tareas

[15] aunque no necesariamente la comparten con la tarea τ_i, ya que su prioridad se podría elevar indirectamente.

- $B_i^s = \Sigma_{k=1}^{m} \max_{j>i}[D_{j,k} : \text{Límite}(S_k) \geq P_i]$

Límite(S_k) es la prioridad de la tarea *cliente* más prioritaria de las que utilizan la sección crítica k. Sólo se consideran secciones que poseen un límite de prioridad no inferior a la prioridad de la tarea para la que se calcula el factor de bloqueo. $D_{j,k}$ es la duración de la ejecución de la sección crítica k por parte de la tarea τ_j.

Escribimos, por tanto, el factor de bloqueo de la tarea τ_i como: $B_i = \text{Min}(B_i^l, B_i^s)$.

Protocolos basados en el techo de prioridad

El protocolo de herencia de prioridad proporciona un límite superior del tiempo que puede permanecer bloqueada una tarea de alta prioridad. Sin embargo, se trata de una estimación del factor de bloqueo que puede resultar ser inaceptablemente pesimista, ya que, como se ha comentado anteriormente, el protocolo de herencia de prioridad no está libre de bloqueos *transitivos* o cadenas de bloqueos. Si se tiene en cuenta esta posibilidad en el cálculo del factor de bloqueo, el resultado será un valor límite superior excesivamente alto.

Los protocolos de techo de prioridad más utilizados en la actualidad son los siguientes:

1. protocolo original de límite de prioridad (OCPP),
2. protocolo inmediato de límite de prioridad (ICPP).

Justificaremos que, cuando se implementan utilizando un sistema monoprocesador, los protocolos de techo de prioridad cumplen lo siguiente:

- Una tarea prioritaria sólo puede ser bloqueada como máximo una vez durante su ejecución por otras de menor prioridad.
- Los interbloqueos se previenen.
- Se previenen los bloqueos transitivos.
- Se asegura el acceso en exclusión mutua a los recursos compartidos.

Con este tipo de protocolos se establece que las tareas pueden ser bloquedas en uno de estos dos casos: (a)cuando intentan bloquear un recurso previamente bloqueado por otra tarea; (b) si al bloquear un recurso se pudiera dar lugar a un bloqueo múltiple de tareas de mayor prioridad. Vamos a discutir a continuación cómo se puede conseguir obtener las propiedades anteriores para el protocolo ICPP, que es el contempla la norma POSIX para sistemas operativos.

El *techo de prioridad* ("priority ceiling") de un recurso es la prioridad de la tarea más prioritaria que puede bloquear a dicho recurso. Por lo tanto, el techo de prioridad de un recurso representa la mayor prioridad a la cual una sección crítica protegida puede ser ejecutada por una tarea de la aplicación.

Protocolo de techo de prioridad inmediato

Este protocolo es llamado en POSIX: *Priority Protected Protocol*(PPP). Se define mediante las siguientes reglas:

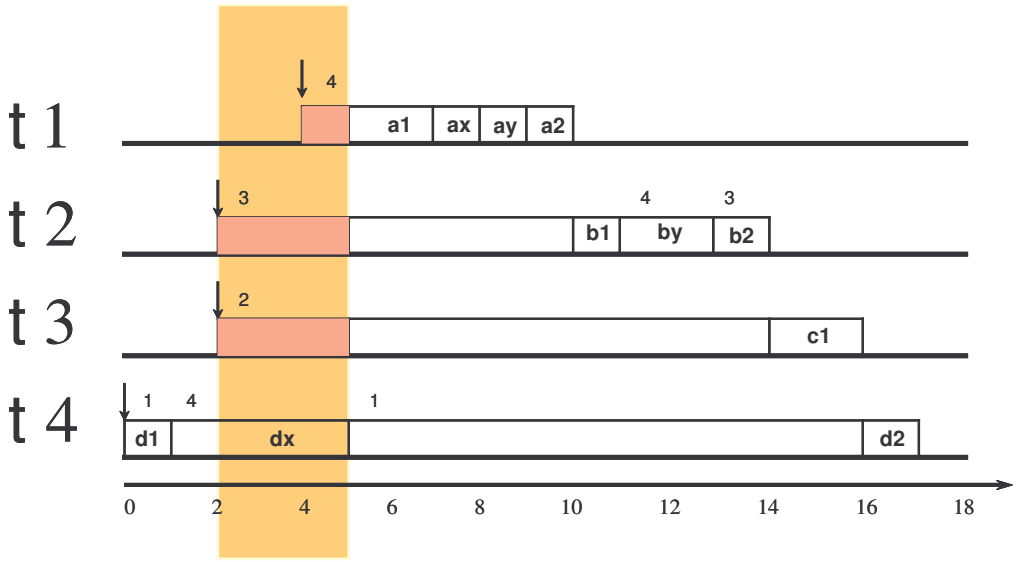


Figura 4.11: Escenario de 4 tareas con el protocolo *techo de prioridad inmediato*

1. Cada tarea tiene una prioridad estática asignada por defecto.

2. Cada recurso tiene un valor de techo de prioridad definido, que coincide con la máxima prioridad de las tareas que lo utilizan[16].

3. Una tarea tiene una prioridad dinámica que coincidirá con el máximo entre su propia prioridad estática inicial y los valores de los techos de prioridad de cualesquiera recursos que tenga bloqueados.

La tarea τ_4 de la figura 4.11 comienza a ejecutarse primero. Dado que el recurso X está libre cuando intenta acceder a él, adquiere el acceso al recurso y comienza a ejecutar la sección crítica elevándose su prioridad hasta el nivel máximo, que se corresponde con el techo de prioridad de dicho recurso. Hasta que la tarea τ_4 no termine de ejecutar las 4 unidades de tiempo de la sección crítica (dx) en ($t = 5$) ninguna de las otras tareas podrá comenzar a ejecutarse.

Las tareas sólo sufrirán bloqueo al principio de su ejecución. Ya que, cuando una tarea comienza, todos los recursos que vaya a utilizar han de estar libres, si no otras tareas tendrían una prioridad igual o mayor que dicha tarea (ya que en ese momento poseerían los recursos que la tarea pretende bloquear) y, por tanto, se vería bloqueada. En consecuencia, propiedades tales como la evitación de bloqueos transitivos y ausencia de interbloqueos son satisfechas por el protocolo.

El protocolo ICPP es más fácil de implementar que el protocolo de techo de prioridad original OCPP, pues produce menos cambios de contexto del planificador del sistema, ya que, como hemos comentado, el bloqueo de las tareas es previo a su ejecución. Por otra parte, la implementación del ICPP necesita, en general, más cambios de prioridad que el protocolo original, ya que estos se producen cada vez que una tarea bloquea recursos.

Los dos protocolos de límite de prioridad aseguran la exclusión mutua en el acceso a recursos compartidos, sin necesidad de usar primitivas de sincronización adicionales (semáforos, etc.). Ya que si una tarea obtiene acceso a algún recurso, entonces se ejecutará con el valor del límite

[16]Este valor se obtiene para recurso, observando el código de las tareas de la aplicación antes de la ejecución

de prioridad asignado a dicho recurso. Ninguna otra que utilice dicho recurso puede tener una prioridad mayor; por lo tanto, o bien la tarea accede al recurso sin ser interrumpida hasta termine, o la tarea resulta bloqueada antes de acceder a la sección crítica. En cualquier caso, la exclusión mutua en el acceso al recurso está asegurada con el protocolo ICPP mismo.

Tiempo de bloqueo

Dado que con los protocolos de techo de prioridad la tarea más prioritaria sólo puede sufrir un único bloqueo inicial, tal como se justificó anteriormente. Si utilizamos estos protocolos, entonces el valor máximo del tiempo de bloqueo B_i de una tarea τ_i es igual a la duración de la sección crítica más larga de las que acceden las tareas de prioridad inferior y que posean un límite de prioridad no menor que la prioridad de la primera ($prio(\tau_i)$).

Hay que tener en cuenta que una tarea puede verse bloqueada por otra menos prioritaria aunque no accedan a recursos comunes.

Por consiguiente, el factor de bloqueo para los protocolos de límite de prioridad vendrá dado como el resultado de evaluar la siguiente expresión:

$$B_i = Max_{\{j,k\}} \{D_{j,k} \mid prio(\tau_j) < prio(\tau_i),\ limite\_prioridad(S_k) \geq prio(\tau_i)\}$$

4.3.3 Algoritmos de planificación de tareas aperiódicas

Una forma simple de planificar tareas aperiódicas en un sistema de tiempo real sería la de asignarles una prioridad menor que la de las tareas cuya misión sea crítica. Sin embargo, actuando de esta manera, dichas tareas se ejecutarían sólo como actividades en *segundo plano*, e incumplirían frecuentemente sus tiempos límite. Por lo tanto, para mejorar la respuesta de las tareas que puedan perder ocasionalmente algún tiempo límite sin afectar a la seguridad del sistema[17] se puede emplear un *servidor* que consiste en una tarea real (o *conceptual*) que mantiene la planificación de los procesos cuya misión es crítica, pero, al mismo tiempo, permite que los procesos aperiódicos permisivos se puedan ejecutar tan pronto como sea posible.

Servicio de peticiones aperiódicas utilizando procesamiento de segundo plano

Esta solución, compatible con el test de Liu-Layland, consiste en asignar un tiempo extra para servir las peticiones de servicio de tareas aperiódicas. El denominado *tiempo aperiódico* se incrementa a intervalos regulares, siempre con la prioridad más baja del sistema, y se utiliza íntegramente para procesar las peticiones pendientes de activación tareas aperiódicas. Cuando no hay más peticiones aperiódicas pendientes se pierde, es decir, el tiempo aperiódico con este esquema no se preserva para atender las futuras peticiones aperiódicas que pudieran producirse.

[17] dichos sistemas los categorizamos como sistemas de tiempo real permisivos al inicio del tema

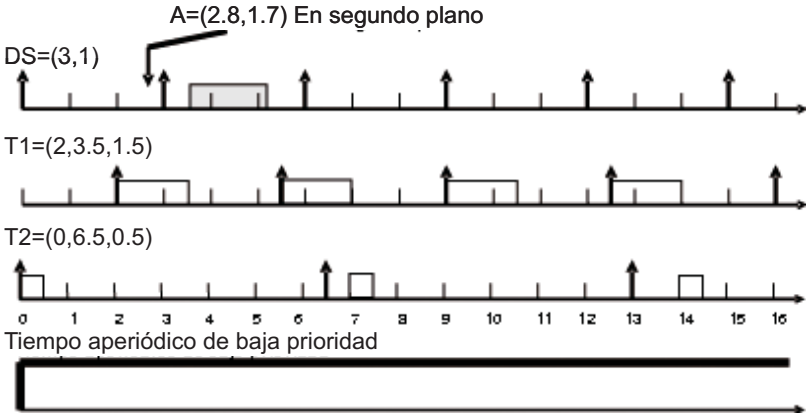


Figura 4.12: Escenario con peticiones aperiódicas en *segundo plano*

Servicio de peticiones aperiódicas utilizando sondeos

En este caso se mejoran un poco los tiempos de respuesta de las tareas aperiódicas respecto del método anterior. Se añade una tarea *sondeante* a las tareas periódicas de la aplicación, que puede ser una tarea física o conceptual. Es decir, no sería necesario crearla como tal, símplemente se puede reservar algún tiempo del procesador para sondear periódicamente si hay peticiones aperiódicas. Se trata de *tiempo periódico*, es decir, si no existen peticiones aperiódicas, se emplea en ejecutar las tareas periódicas activas. A diferencia del método anterior, se reserva un "tamaño" (C_s), equivalente al tiempo de ejecución de peor caso de una tarea periódica. La prioridad no se obtiene a partir del periodo (T_s), sino que se le puede asignar la prioridad que más convenga, para no provocar que las tareas periódicas pudieran perder algún límite de tiempo.

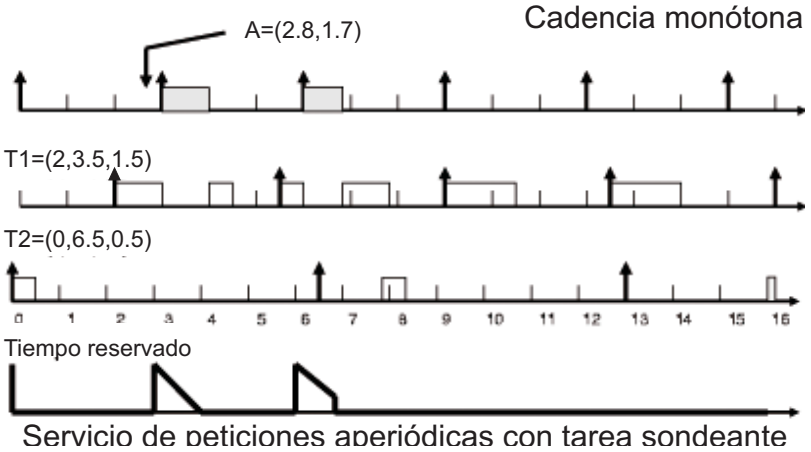


Figura 4.13: Escenario con peticiones aperiódicas y *tarea sondeante*

La planificabilidad de las tareas periódicas suele ser garantizada aplicando el test de Liu-Layland. Independientemente del número de tareas que pudieran solicitar servicio, en cada ciclo de sondeo se dedica un tiempo máximo igual a C_s para atender peticiones aperiódicas. La planificabilidad de un conjunto periódico con N tareas y asignación estática de prioridades según el algoritmo de cadencia monótona puede ser garantizado si y sólo si se cumple la desigualdad

siguiente:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \; + \; \frac{C_s}{T_s} \; \leq \; (N+1)[2^{\frac{1}{(N+1)}} \; - \; 1]$$

Algoritmos que preservan el tiempo asignado al procesamiento de peticiones aperiódicas.

Esta solución está basada en la implementación de un *servidor diferido*, que preserva el tiempo aperiódico, incluso si transitoriamente no hay peticiones de servicio de este tipo. Se asigna un tamaño al servidor (C_s) que se gasta en atender peticiones aperiódicas y se rellena hasta su valor máximo en cada ciclo del servidor (T_s). Se comienza realizando un análisis de planificabilidad del sistema de tiempo real, para determinar el *tamaño*[18](C_s) de una tarea servidora aperiódica de la máxima prioridad para incluirla en la ronda de planificación junto con el resto de las tareas periódicas.

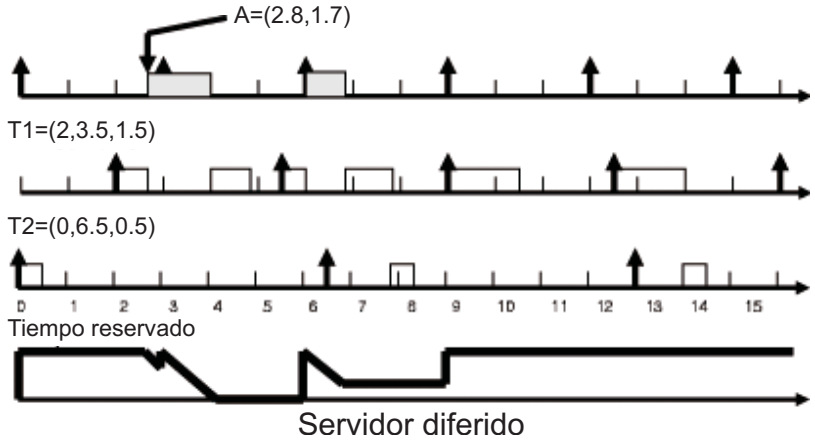


Figura 4.14: Escenario con peticiones aperiódicas y *servidor diferido*

Dichos valores se eligen de manera que todas las tareas clasificadas como de *misión crítica* del sistema mantengan siempre sus tiempos límite. De esta forma, se logra mantener el tiempo de ejecución asignado a las peticiones aperiódicas a un nivel de prioridad muy alto durante todo el ciclo de ejecución del servidor. Cuando sucede la activación de una tarea aperiódica se la sirve a la máxima prioridad, siempre que la capacidad del servidor no se haya agotado.

Análisis de planificabilidad

Considérese un conjunto de N tareas periódicas, $\tau_1 \ldots \tau_N$ y un *servidor diferido* con prioridad más alta. La condición de planificación en el peor caso se basa en un cálculo complejo, para obtener una función de la utilización del procesador por parte del servidor (U_s), es decir, se calcula el *menor límite superior de de utilización*[19]:

$$U_{mls} \; = \; U_s \; + \; N \left[\left(\frac{U_s \; + \; 2}{2U_s \; + \; 1} \right)^{\frac{1}{N}} \; - \; 1 \right]$$

[18]tiempo del procesador reservado para la ejecución de procesos aperiódicos

[19]es el mínimo de los límites de utilización entre los conjuntos de tareas que intentan usar todo el tiempo del procesador

Tomando el límite para $N \rightarrow \infty$, en el peor caso, encontramos como menor límite superior de utilización,

$$\lim_{N \rightarrow \infty} U_{mls} = U_s + \ln\left(\frac{U_s + 2}{2U_s + 1}\right)$$

Consecuentemente, dado un conjunto de N tareas periódicas y un *servidor diferido* con límites de utilización U_p y U_s, respectivamente, la planificabilidad del conjunto de tareas periódicas está garantizada, con el algoritmo de cadencia monótona, si $U_p + U_s \leq U_{mls}$; esto es, si se cunple la desigualdad siguiente:

$$U_p \leq \ln\left(\frac{U_s + 2}{2U_s + 1}\right)$$

4.4 Problemas resueltos

Ejercicio 1

Verificar la planificabilidad y construir el diagrama de ejecución de tareas utilizando el algoritmo de "cadencia monótona" (RM) para el siguiente conjunto de tareas periódicas

	C_i	T_i
τ_1	2	6
τ_2	2	8
τ_3	2	12

Solución:

Las tareas del conjunto anterior son planificables, ya que según el criterio RM, el factor de utilización del conjunto U $= \frac{2}{6} + \frac{2}{8} + \frac{2}{6} = 0.75$ es menor que la cota de utilización máxima RM: $U_0(3) = 3 \times (2^{\frac{1}{3}} - 1) \approx 0.78$. Tal como se puede ver en el diagrama de ejecución de las tareas de la figura 4.15.

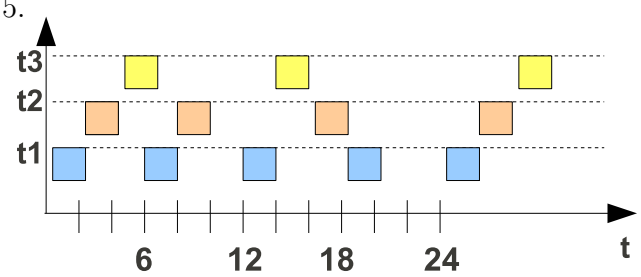


Figura 4.15: Diagrama de Gantt para las tareas del Ejercicio 1

Ejercicio 2

Verificar la planificabilidad y construir el diagrama de ejecución de tareas utilizando el algoritmo RM para el siguiente conjunto de tareas periódicas

	C_i	T_i
τ_1	3	5
τ_2	1	8
τ_3	2	10

Solución:

Con el test de planificabilidad RM no se puede afirmar que las tareas del conjunto anterior son planificables, ya que el factor de utilización del conjunto U $= \frac{3}{5} + \frac{1}{8} + \frac{2}{10} = 0.825$ es mayor que la cota de utilización máxima RM: $U_0(3)$. Sin embargo, se puede comprobar con el diagrama de ejecución de tareas de la figura 4.4 que son planificables.

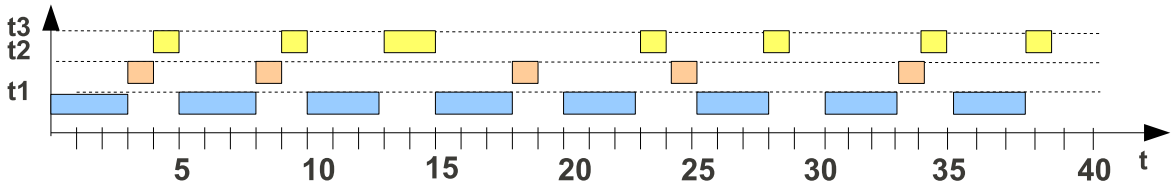


Figura 4.16: Diagrama de Gantt para las tareas del Ejercicio 2

Ejercicio 3

Verificar la planificabilidad del siguiente conjunto de tareas utilizando el algoritmo de "primero el plazo límite más cercano" (EDF)

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	10

Solución:

El conjunto de tareas es planificable con el algoritmo EDF si se toma el valor límite de D_i igual al del periodo de cada tarea ($D_i = T_i$), tal como se muestra en la figura 4.4. Se puede comprobar que factor de utilización del procesador que produce dicho conjunto es menor del 100%: $U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = 0.96$

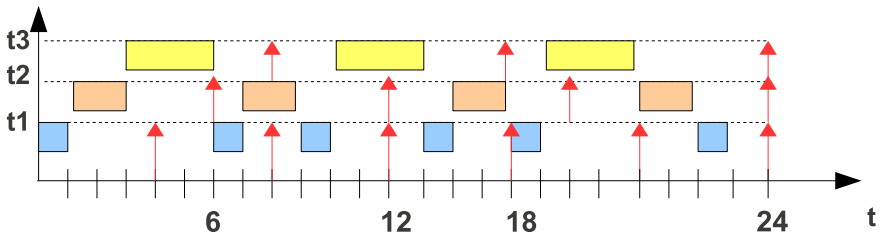


Figura 4.17: Diagrama de Gantt para las tareas del Ejercicio 3

Ejercicio 4

Verificar la planificabilidad utilizando el algoritmo EDF y construir el diagrama de ejecución de tareas del siguiente conjunto

	C_i	D_i	T_i
τ_1	2	5	6
τ_2	2	4	8
τ_3	4	8	12

Solución:

Este conjunto de tareas resulta ser planificable con el algoritmo EDF de asignación dinámica de prioridades a las tareas: $U_3 = \frac{2}{6} + \frac{2}{8} + \frac{4}{12} \approx 0.917 < 1.0$

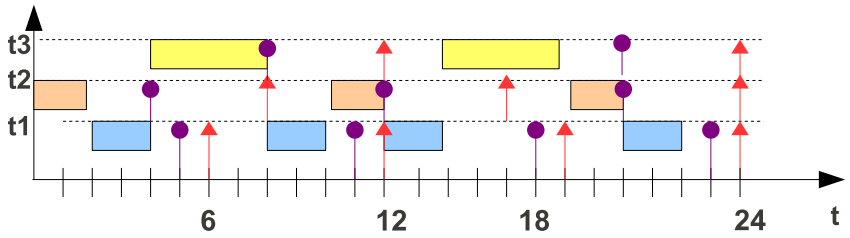


Figura 4.18: Diagrama de Gantt para las tareas del Ejercicio 4

Ejercicio 5

Verificar la planificabilidad del conjunto de tareas descrito en el Ejercicio 4, utilizando para ello el algoritmo del "plazo límite monótono" (*Deadline Monotonic* o DM)

Solución:

Este conjunto de tareas resulta no ser planificable con el algoritmo DM de asignación estática de prioridades a las tareas. Según se puede ver en la figura 4.4, la tarea τ_3 pierde su plazo de tiempo límite en t = $\{8, 20, \ldots\}$

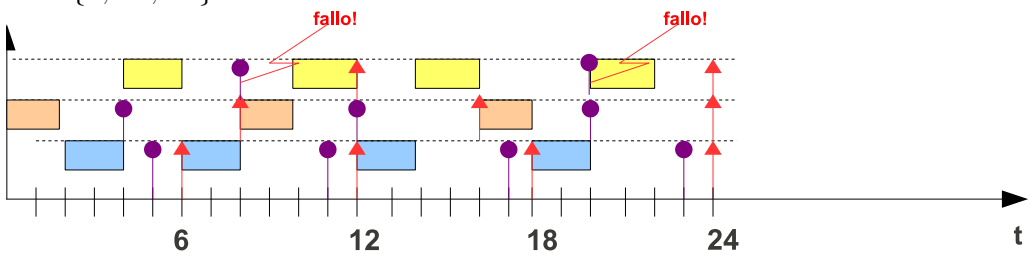


Figura 4.19: Diagrama de Gantt para las tareas del Ejercicio 5

Ejercicio 6

Calcular: a) la utilización máxima del procesador que se puede asignar al *Servidor Esporádico* para garantizar la planificabilidad del siguiente conjunto de tareas periódicas utilizando RM; b) la utilización del procesador máxima que puede ser asignada al Servidor Diferido (SD) para garantizar la planificabilidad del conjunto de tareas periódicas dado; c) un plan para planificar las siguientes tareas aperiódicas utilizando un *SE* que posea una utilización máxima y prioridad intermedia.

τ_1	1	5
τ_2	2	8

Solución (a):

El servidor esperádico (SE) se comporta como una tarea periódica. En la peor situación de planificación para el conjunto de tareas anterior, es decir, cuando exista la máxima interferencia entre ellas, el conjunto $\{\tau_1, \tau_2\}$ se puede garantizar como planificable con el algoritmo RM si se cumple la desigualdad $U_p \leq n \times ((\frac{2}{U_s+1})^{\frac{1}{n}} - 1) \Rightarrow U_s \leq 2 \times (\frac{U_p}{n} + 1)^n - 1$

Donde :

n	número de tareas periódicas
U_p	Utilización del procesador periódicas
U_s	Utilización del procesador aperiódicas

Para $n \to \infty$: $U_p \leq ln(\frac{2}{U_s+1})$ Luego, para n= 2: $U_{s\,max} = 0.33$ y $U_p = 0.45$.

Solución (b):

La utilización máxima U_s del procesador para el SD que garantice el mantenimiento de la planificabilidad del conjunto de tareas $\{\tau_1, \tau_2 \ldots\}$ viene dado por la siguiente inecuación:

$U_p \leq n \times ((\frac{U_s+2}{2 \cdot U_s+1})^{\frac{1}{n}} - 1)$

$\Rightarrow U_s \leq \frac{2-K}{2 \cdot K-1}$ donde $K = (\frac{U_p}{n+1})^n$

Para $n \to \infty$: $U_p \leq ln(\frac{U_s+2}{2 \cdot U_s+1})$ Luego, para n= 2: $U_{s\,max} = 0.25$ y $U_p = 0.45$.

Solución (c):

Sabemos que $U_{smax} = 0.33$ corresponde a la máxima utilización del procesador que podemos asignar al servidor para garantizar la planificabilidad del conjunto de tareas. Por tanto, asignando un periodo al servidor $T_s = 6$ (prioridad intermedia) y $C_s = 2$ se satisfacen las desigualdades y, por consiguiente, el conjunto de tareas $\{\tau_1, \tau_2\}$ se mantiene planificable aun con la aparición de las peticiones aperiódicas: $\{J_1, J_2, J_3\}$, según se puede ver en el siguiente diagrama de ejecución de tareas:

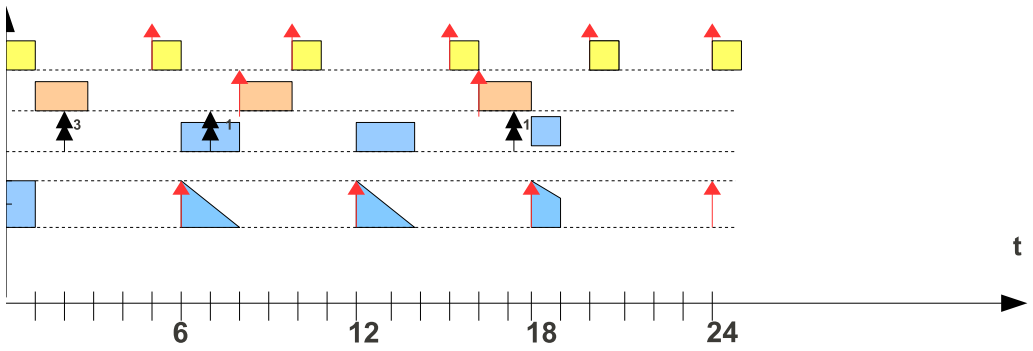


Figura 4.20: Diagrama de Gantt para las tareas del Ejercicio 6

4.5 Problemas propuestos

1. Verificar la planificabilidad (calcular el factor de utilización máxima y compararlo con U_3) y construir el diagrama de ejecución de tareas utilizando el algoritmo de "cadencia monótona" (RM) para el siguiente conjunto de tareas periódicas:

	C_i	T_i	Prio
τ_1	3	7	2
τ_2	3	12	2
τ_3	5	20	1

2. Verificar la planificabilidad y construir el diagrama de utilizando el algoritmo RM para el siguiente conjunto de tareas periódicas :

	C_i	T_i
τ_1	3	5
τ_2	1	8
τ_3	2	10

3. Verificar la planificabilidad y construir el diagrama de Gantt utilizando el algoritmo RM para el siguiente conjunto de tareas periódicas:

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	10

4. Calcular el *tiempo respuesta*(R) para cada una de las tareas del ejercicio anterior.

5. Verificar la planificabilidad y construir el diagrama de utilizando el algoritmo RM para el siguiente conjunto de tareas periódicas:

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	8

6. Calcular el *tiempo respuesta*(R) para cada una de las tareas del ejercicio anterior.

7. Calcular el tiempo de respuesta de cada una de las tareas de la tabla siguiente, calcular su tiempo de respuesta.

	C_i	T_i	Prio	D_i
τ_1	3	12	1	8
τ_2	6	20	2	10

8. Para el conjunto de tareas del ejercicio anterior, ¿Se puede encontrar una *implementación factible* para el conjunto de tareas, es decir, un programa de tiempo real asegurando que todas éstas terminarán cualquiera de sus activaciones antes que se cumplan los tiempos límite que tienen asignados en la tabla?

9. Para el conjunto de tareas $\{t_1,\ t_2,\ t_3\}$ cuyos datos se muestran más abajo, se pide:

   (a) Dibujar el diagrama de ejecución y obtener el tiempo de respuesta de cada una de las tareas.

   (b) Determinar, mediante inspección del gráfico, cuántas veces interfiere la tarea τ_1 a la tarea τ_3 durante el intervalo dado por el *tiempo de respuesta* de t_3.

   (c) Idem para las tareas τ_1 y τ_2.

   (d) Obtener el número máximo de veces que una tarea τ_j interfiere a otra tarea τ_i en función del periodo de la primera (T_j) y del tiempo de respuesta de la segunda (R_i).

	C_i	T_i	D_i
τ_1	1	3	2
τ_2	3	6	5
τ_3	2	13	13

10. Calcular el tiempo de finalización de la tarea τ_3 y determinar si las tareas A (servidor aperiódico), E (servidor esporádico) y las tareas periódicas: τ_1, τ_2 y τ_3 son todas ellas planificables.

	Tarea T_i	C_i	B(tiempo bloqueo acceso sscc)	Techo prio	P_i
A	40	2	0	-	2
τ_1	100	20	20	-	3
E	120	5	0	-	1
τ_2	150	40	20	-	4
τ_3	350	100	0	-	5
S_1	-	-	-	3	-
S_1	-	-	-	3	-

11. Utilizar un *Servidor Esporádico* con capacidad $C_s = 2$ y periodo $T_s = 5$ para planificar las siguientes tareas:

	C_i	T_i
τ_1	1	4
τ_2	2	6
	a_i	C_i
J_1	2	2
J_2	5	1
J_3	10	2

12. Para el conjunto de tareas cuyos datos se muestran más abajo, se pide:

   (a) Dibujar el gráfico de ejecución y obtener el tiempo de respuesta de cada tarea.

   (b) Determinar, mediante la inspección del gráfico anterior, cuántes veces interfiere la tarea τ_1 a la tarea τ_3 durante el intervalo temporal dado por el tiempo de respuesta de esta última tarea.

(c) Dibujar el diagrama de ejecución, obtener el tiempo de respuesta, e indicar el instante de tiempo en que cambia la prioridad dinámica de las tareas de la tabla siguiente. Suponemos que dichas tareas se planifican utilizando utilizando el protocolo denominado *herencia de prioridad*:

	C_i	T_i	I_i	Prio	recursos
τ_1	4	16	5	4	A, B
τ_2	4	16	5	3	-
τ_3	5	16	2	2	B,A
τ_4	5	18	0	1	A

13. Indicar si las afirmaciones siguientes son verdaderas:

(a) Suponiendo que las tareas se planifican con el protocolo de *herencia de prioridad*: la prioridad heredada por una tarea sólo se mantiene mientras dicha tarea esté utilizando un recurso compartido con otra tarea más prioritaria.

(b) Con el protocolo de *techo de prioridad*, cuando una tarea adquiere un recurso no puede verse interrumpida, hasta que termine su ejecución, por otras tareas que se activan después que ésta y que vayan a utilizar en el futuro un recurso con límite de prioridad igual o inferior.

(c) Con el protocolo de techo de prioridad (OCPPP), una tarea no puede comenzar a ejecutarse si no están libres todos los recursos que va a utilizar durante su primer ciclo.

(d) Si consideramos una tarea periódica que utilice el protocolo de techo de prioridad inmediato (ICPP) para cambiar su prioridad dinámica cuando accede a recursos, siempre se cumplirá que dicha tarea no puede ser interrumpida por otra menos prioritaria que ella.

(e) Con el protocolo de techo de prioridad las tareas más prioritarias del sistema pueden ser interrumpidas durante cada ciclo de su ejecución como máximo 1 vez cuando acceden a recursos que comparten con otras tareas menos prioritarias.

(f) El protocolo de techo de prioridad original (OCPP) producirá siempre tiempos de respuesta menores para las tareas que el algoritmo de *herencia de prioridad*.

14. Indicar si son ciertas las siguientes afirmaciones.

(a) Con asignación de prioridades basado en el menor tiempo límite relativo se obtiene un esquema de planificación dinámico (es decir, la prioridad de las tareas puede cambiar durante la ejecución del programa).

(b) Con asignación de prioridades según el algoritmo RM se puede asegurar la planificabilidad de un conjunto de n tareas periódicas si la utilización conjunta del procesador no supera el número $N \times (2^{\frac{1}{N}} - 1)$ siempre que todas las tareas comiencen al mismo tiempo.

(c) Si se utiliza asignación estática de prioridades a un conjunto de tareas periódicas independientes y todas acaban antes de que se cumpla el plazo de su tiempo límite en su primera activación, entonces podemos afirmar que siempre se cumplirán los plazos durante toda la ejecución de dichas tareas.

(d) Con un esquema de asignación de prioridades estático nunca se puede garantizar la planificabilidad de un conjunto de tareas periódicas independientes si la utilización conjunta del procesador $U_n = 100\%$.

(e) Suponiendo que el sistema utilice el algoritmo del *servidor diferido* para atender peticiones de servicio aperiódicas, afirmamos lo siguiente: si se agota la capacidad del servidor, es decir, el tiempo asignado para atender las tareas aperiódicas, entonces las dichas peticiones aperiódicas que pudieran llegar antes del comienzo de la siguiente reposición de tiempo del servidor se pierden.

(f) La tarea correspondiente al servidor diferido siempre se planifica con igual prioridad a la de la tarea periódica más prioritaria.

(g) El tiempo no utilizado del servidor diferido no se pierde, aunque no es acumulable.

(h) Suponiendo que se use el servicio de peticiones aperiódicas: si la capacidad del servidor aperiódico se ha terminado en el ciclo actual, se puede utilizar el tiempo sobrante asignado a las tareas periódicas.

(i) En cualquier caso, una petición aperiódica interrumpirá a cualquier tarea periódica activa en ese momento independientemente de la prioridad de esta última (suponer el algoritmo de intercambio de prioridad).

(j) Hay algoritmos de asignación estática de prioridades a las tareas en que la prioridad del servidor aperiódico puede cambiar con el tiempo.

(k) El tiempo que se haya consumido del servidor esporádico sólo se repone cuando se agota totalmente el tiempo del servidor y llega el siguiente ciclo de la tarea que implementa al servidor.

(l) Siempre que el nivel de prioridad del servidor esporádico vuelve a estar activo se repone el tiempo reservado para peticiones esporádicas.

(m) Podría ocurrir que la tarea que implementa el servidor esporádico fuese desplazada del procesador por una tarea periódica más prioritaria que ésta.

(n) Cuando se repone tiempo del servidor esporádico no ha de ser necesariamente hasta su nivel máximo.

(o) Suponiendo que las tareas se planifican con el protocolo de herencia de prioridad: la prioridad heredada por una tarea sólo se mantiene mientras utilice un recurso compartido con otra tarea más prioritaria.

(p) Con el protocolo de techo de prioridad, cuando una tarea adquiere un recurso no puede verse interrumpida, hasta que termine su ejecución, por otras tareas que vayan a utilizar un recurso de límite de prioridad igual o inferior.

(q) Con el protocolo de techo de prioridad, un proceso no puede comenzar a ejecutarse si no están libres todos los recursos que va a utilizar durante su primer ciclo.

15. Conceptos generales sobre STR. Seleccionar la respuesta correcta:

    (a) Los programas de tiempo real se caracterizan por la alta frecuencia de ejecución de las instrucciones de sus tareas.

    (b) Un sistema de tiempo real es *confiable* si sus procesos nunca se *cuelgan*.

    (c) Un sistema no puede ser de tiempo real si no se puede determinar el tiempo que necesitan todas sus tareas para terminar su primera activación.

    (d) La *responsividad* se refiere a la propiedad que tienen los sistemas de limitar el número de interrupciones anidadas que puede sufir cualquier tarea crítica.

16. Conceptos sobre medida del tiempo en los sistemas informáticos. Seleccionar la respuesta correcta:

    (a) Tiempo monónoto de un ordenador es el que nunca se agota y viene incluido en la fecha del sistema.

    (b) La *precisión* de un reloj de tiempo real se refiere a la unidad más pequeña de tiempo que se nos muestra en pantalla cuando ejecutamos la orden `gettimeofday(...)`.

    (c) Existen maneras de eliminar completamente la *deriva* en el tiempo fijado por un temporizador para el inicio de un tarea de tiempo real. En realidad sólo se trataría de descontar los retrasos externos a las tareas en cada ciclo de éstas.

    (d) Un computador puede tener tantos relojes de tiempo real POSIX como queramos programar.

17. Conceptos sobre el modelo simple de tareas de los STR. Seleccionar la respuesta correcta:

    (a) En este modelo las tareas nunca se pueden sincronizar.

    (b) El tiempo límite d_i de una tarea de tiempo real varía dependiendo del instante de activación.

    (c) El plazo de respuesta máximo D_i de una tarea depende del instante de tiempo en el que se produzca la siguiente activación de la tarea τ_i.

    (d) Si asignamos prioridades a las tareas según su menor plazo de respuesta, tenemos un esquema de prioridades estático; sin embargo, si consideramos la prioridad de una tarea mayor si su tiempo límite se encuentra más cercano, el esquema de prioridades será dinámico.

18. Conceptos generales sobre planificación de tareas periódicas. Seleccionar la respuesta correcta:

    (a) Con asignación de prioridades según el algoritmo RM se puede asegurar la planificabilidad de un conjunto de n tareas periódicas si la utilización conjunta del procesador no supera el número $N \times (2^{\frac{1}{N}} - 1)$ siempre que todas las tareas comiencen al mismo tiempo.

    (b) Si se utiliza asignación estática de prioridades a un conjunto de tareas periódicas independientes y todas acaban antes de que se cumpla el plazo de su tiempo límite en su primera activación, entonces podemos afirmar que siempre se cumplirán dichos plazos durante toda la ejecución de dichas tareas.

(c) Con un esquema de asignación de prioridades estático se podría tener conjuntos de tareas periódicas independientes y planificables llegando a la utilización conjunta del procesador $U_n = 100\%$.

(d) Con asignación de prioridades basado en el menor plazo de respuesta o tiempo límite relativo (D_i)se obtiene un esquema de planificación dinámico .

19. Servidores de tareas aperiódicas y esporádicas. Seleccionar la respuesta correcta:

(a) Si se agota la capacidad del *servidor diferido*, es decir, el tiempo asignado para atender las tareas aperiódicas, entonces las peticiones aperiódicas que pudieran llegar antes del comienzo de la siguiente reposición de tiempo del servidor se pierden.

(b) Suponiendo que se use el servicio de peticiones aperiódicas: si la capacidad del servidor aperiódico se ha terminado en el ciclo actual, se puede utilizar el tiempo sobrante asignado a las tareas periódicas.

(c) El tiempo no utilizado del servidor diferido no se pierde, aunque no es acumulable.

(d) En cualquier caso, una petición aperiódica interrumpirá a cualquier tarea periódica activa en ese momento independientemente de la criticidad de esta última.

20. Algoritmos para resolver el problema de la *inversión de prioridad*. Seleccionar la respuesta correcta:

(a) Suponiendo que las tareas se planifican con el protocolo de *herencia de prioridad*: la prioridad heredada por una tarea sólo se mantiene mientras dicha tarea esté utilizando un recurso compartido con otra tarea más prioritaria.

(b) Con el protocolo de *techo de prioridad inmediato*, una tarea no puede comenzar a ejecutarse si no están libres todos los recursos que va a utilizar durante su primer ciclo.

(c) Una tarea periódica que utilice el protocolo de techo de prioridad inmediato nunca puede ser interrumpida por otra menos prioritaria que ella.

(d) Con el protocolo de *herencia de prioridad* las tareas más prioritarias pueden ser interrumpidas durante cada ciclo de su ejecución como máximo 1 vez por otras tareas menos prioritarias.