SISTEMAS INFORMÁTICOS EN TIEMPO REAL

Planificación

Manuel Agustín Ortiz López
Área de Arquitectura y Tecnología de Computadores
Departamento de Arquitectura de Computadores,
Electrónica y Tecnología Electrónica
Universidad de Córdoba

Córdoba a 25 de enero de 2007

Planificación

- Índice:

1. Introducción

- Unas de las partes fundamentales del sistema operativo es el administrador de tareas que es el encargado de decidir que tarea se debe ejecutar en cada momento. Como ya habíamos comentado en el tema anterior el administrador de tareas suele estar dividido en dos partes: planificador y despachador. En este tema nos centraremos en el planificador o lo que es lo mismo en como decide el S.O. que tarea se debe ejecutar y durante cuanto tiempo.
- El planificador entra en ejecución por dos condiciones:
  - La interrupción de reloj de tiempo real y cualquier interrupción que señale que se ha completado una petición de entrada/salida.
  - La suspensión de una tarea debido a un retardo pedido por una tarea, o por la petición o finalización de una transferencia de entrada salida.

2. Estrategias de planificación.[Bennett,94]

- Existen diversas estrategias de planificación pero todas se pueden englobar en alguno de los siguientes grupos básicos:
  - **Estrategias cíclicas**.
  - **Estrategias expropiativas**.

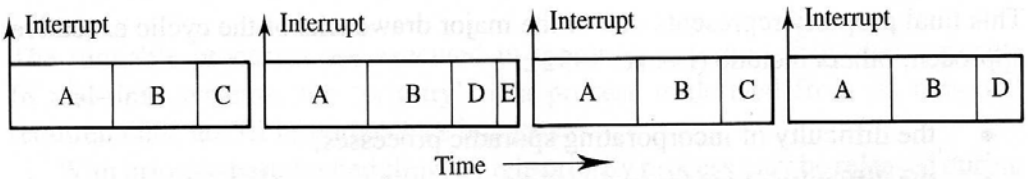
2.1. Estrategias cíclicas

- Esta estrategia consiste en asignar las tareas a la CPU por turno, no permitiéndose la interrupción de una tarea antes de su finalización. Una tarea utiliza la CPU tanto tiempo como desea. Cuando no necesita la CPU, el planificador la asigna a la siguiente tarea de la lista.
- Es la estrategia más simple y muy eficiente ya que minimiza el tiempo de conmutación de tareas. Es una estrategia efectiva para pequeños sistemas empotrados para los que el tiempo de ejecución de cada tarea está cuidadosamente calculado y el software se ha dividido dentro de apropiadas tareas.
- El inconveniente que presenta esta estrategia es que es demasiado restrictiva puesto que requiere que cada tarea tenga tiempos similares de ejecución. Otro inconveniente es que es difícil hacer frente a los eventos aleatorios.

Ejemplo

Cyclic executive process set.

Process	*Period, T*	*Computation Time, C*
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2



Time-line for process set.

- El conjunto de procesos se podría planificar en un ejecutivo cíclico como muestra el *Time-Line* de la figura anterior. Cada proceso se asimilaría a una función que el ejecutivo irá llamando. Se tendría un ciclo principal de 100ms, que es el tiempo en el que deben ejecutarse todos los procesos y cuatro ciclos secundarios de 25ms equivalente al proceso que tiene un período menor. El programa principal se podría realizar con un lazo de la siguiente forma:

```
Loop
    Wait_for_interrupt;
            Procedure_For_A;
            Procedure_For_B;
            Procedure_For_C;
    Wait_for_interrupt;
            Procedure_For_A;
            Procedure_For_B;
            Procedure_For_D;
            Procedure_For_E;
    Wait_for_interrupt;
            Procedure_For_A;
            Procedure_For_B;
            Procedure_For_C;
    Wait_for_interrupt;
            Procedure_For_A;
            Procedure_For_B;
            Procedure_For_D;
End loop;
```

2.2. Estrategias expropiativas

- Llamadas también estrategias apropiativas.
- Hay numerosas estrategias de prioridad expropiativas pero todas tienen la posibilidad de que una tarea pueda ser interrumpida (de ahí el término *apropiación*) antes de que haya finalizado.
- Como consecuencia de la interrupción el planificador (*executive*) tiene que grabar el entorno de cada tarea ya que posteriormente se le asignará de nuevo tiempo de CPU, y deberá continuar por el punto exacto donde fue interrumpida. Este proceso por el que se deja de ejecutar una tarea y se ejecuta otra se le llama cambio de contexto.
- La forma más simple de una planificación apropiativa es utilizar el método de turno rotatorio (*round-robin*) o también llamado de rodajas de tiempo (*time slicing*). Según esta estrategia a cada tarea se le asigna una cantidad fija de tiempo de CPU (un número determinado de *ticks* de reloj), al final del cual se da control a la siguiente tarea de la lista. Según esta estrategia a cada tarea en cada turno se le asigna el mismo tiempo de CPU. Si una tarea finalizase antes de que se cumpliera su rodaja de tiempo se continúa con la siguiente tarea de la lista.

- En el método de *round-robin* todas las tareas tienen asignada la misma prioridad. Existen otros métodos que veremos más adelante en los que las tareas están priorizadas de forma que una tarea sólo se interrumpe sí se va a ejecutar otra con mayor prioridad.
- Las prioridades pueden ser fijas (sistemas de prioridad estática), cada tarea tiene asignada una prioridad que se mantiene mientras la aplicación está en ejecución o pueden cambiarse durante la ejecución (sistemas de prioridad dinámica).
- Los esquemas de prioridad dinámica aumentan la flexibilidad del sistema, por ejemplo pueden utilizarse para aumentar la prioridad de alguna tarea particular en condiciones de alarma. Sin embargo el cambio de prioridades es arriesgado ya que es mucho más difícil predecir el comportamiento del sistema y aún más de poder chequearlo. También existe el riesgo de bloquear algunas tareas durante demasiado tiempo. Si el software está bien diseñado y se tiene la potencia de cómputo adecuada, en principio, no se necesita cambiar las prioridades de las tareas dinámicamente.
- Cuando se adopta una estrategia de planificación el sistema administrador de las tareas debe estar en sintonía con los manejadores de interrupciones hardware (en respuesta a eventos externos) o software (generadas por la ejecución de

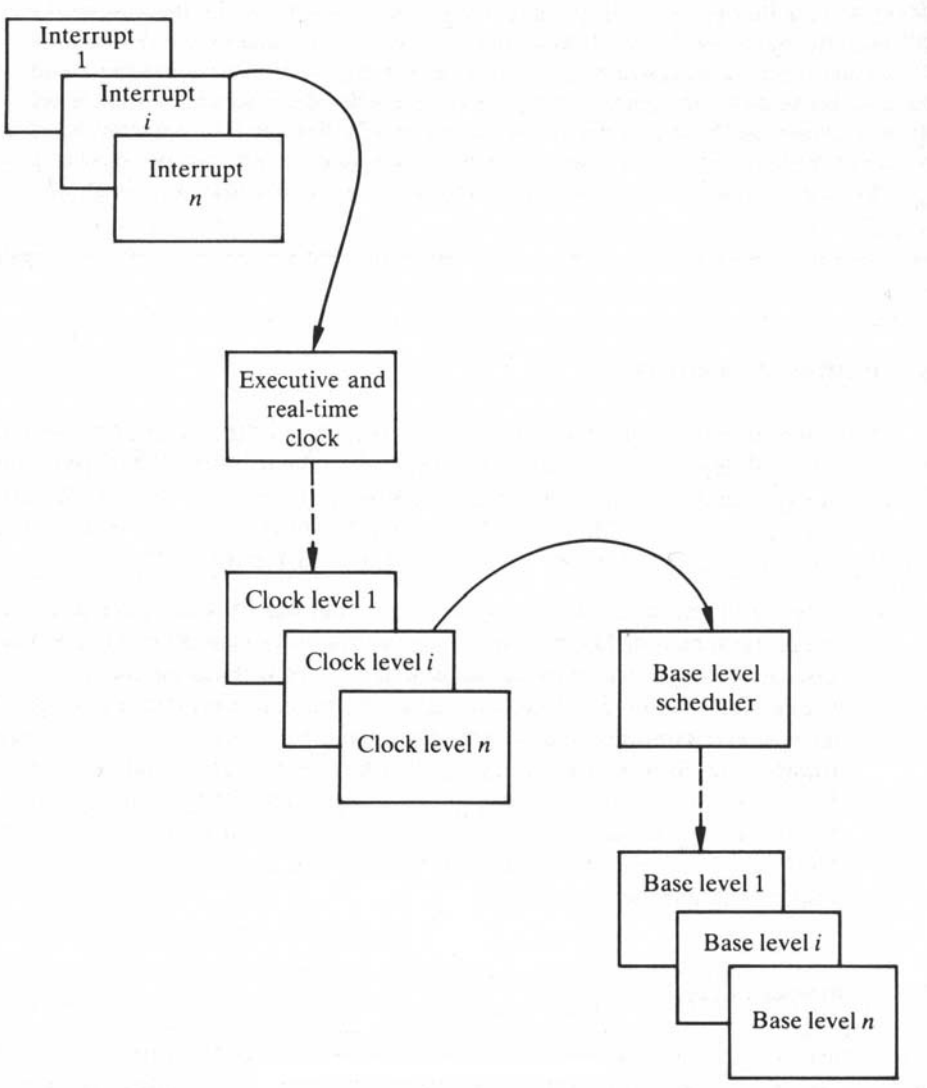
tareas), ya que las interrupciones fuerzan a un cambio de contexto.

- Cuando la ejecución de una tarea se suspende por una interrupción, entra en ejecución el manejador de la interrupción, por lo que debe ser pequeño en tamaño de código y debe ejecutarse rápidamente. Cuando el manejador acaba, o bien la tarea que fue interrumpida se restaura o el planificador entra y determina que tarea se debe ejecutar. El sistema operativo será el que decidirá que actitud tomar.

3. Estructuración de prioridades.[Bennett,94]

- En los sistemas de tiempo real el diseñador debe asignar prioridades a las tareas. La prioridad que asigne dependerá de cómo de rápido deba responder una tarea a un evento particular.
- Los eventos pueden ser o bien una actividad que deba realizar una tarea o simplemente el finalizar el tiempo de espera que hubiera solicitado previamente una tarea.
- En general la mayoría de las tareas de un sistema operativo de tiempo real se pueden clasificar en alguno de los siguientes niveles de prioridad:
  - **Nivel de interrupción**: en este nivel se encuentran los manejadores de interrupción y las rutinas de servicio a las tareas y dispositivos que requieren la más rápida respuesta posible (orden de milisegundos). Entre estas tareas se encuentra la tarea de reloj de tiempo real, y el nivel de reloj del despachador.
  - **Nivel de reloj**: En este nivel se encuentran las tareas que requieren un procesamiento repetitivo, como por ejemplo las tareas de muestreo y control o aquellas que requieren temporizaciones precisas. En el nivel más bajo de prioridad se encuentra el nivel de prioridad base del planificador.
  - **Nivel de prioridad base**: Las tareas que se encuentran en este nivel son las tareas de más baja prioridad, las que no tienen *deadline* exacto o aquellas tareas que tienen un amplio margen de error en las temporizaciones. Las tareas en este nivel pueden tener asignadas

distintas prioridades o pueden ejecutarse con un único nivel de prioridad que podría ser el mismo que el nivel de prioridad base del planificador.



Priority levels in an RTOS.

3.1. Nivel de interrupción

- Las interrupciones fuerzan a un cambio de planificación del trabajo de la CPU y el sistema no tiene control del tiempo de esta nueva planificación.
- Una interrupción genera una nueva planificación fuera del control del sistema y por ello es necesario que la rutina de manejo de la interrupción ocupe el mínimo tiempo posible de CPU. Usualmente estas rutinas sólo utilizan el tiempo necesario para guardar la dirección de retorno y algunos registros de la CPU y pasar información a futuras rutinas que trabajen a un nivel de prioridad inferior, bien a nivel de reloj o nivel de prioridad base.
- Las rutinas de manejo de interrupciones deben tener un mecanismo para cambiar a las tareas que se encarguen de grabar el contexto de la tarea interrumpida. Una vez ejecutada la rutina de interrupción o bien se restaurará el contexto y como consecuencia se retornará a la tarea interrumpida o bien entrará el planificador.
- Dentro de este nivel de interrupción habrá diferentes prioridades y se tendrá que evitar que interrupciones de prioridad baja interrumpan a las que se estén ejecutando con una prioridad mayor.

3.2. Nivel de reloj

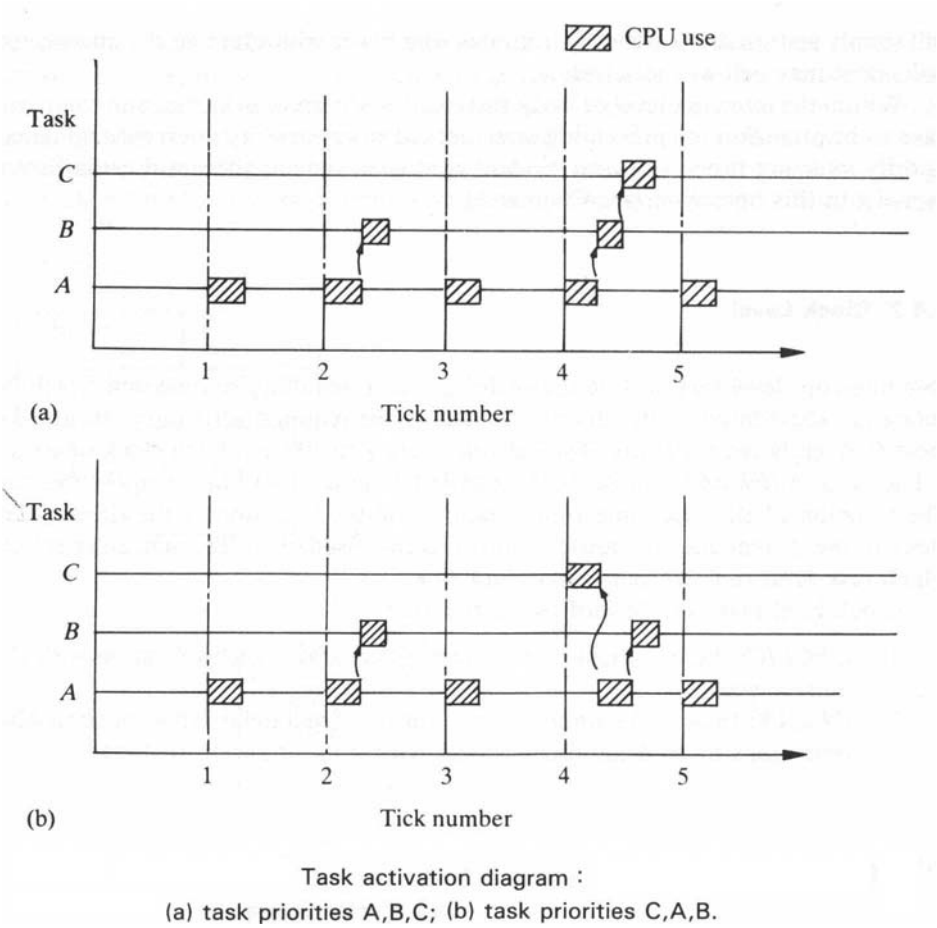
- Una de las tareas del nivel de interrupción será la rutina de manejo de reloj de tiempo real que se ejecutará a un intervalo determinado y que corresponderá en general con la frecuencia de la tarea que se active con más frecuencia. Los valores típicos varían entre 1 y 200 ms.
- Cada interrupción de reloj es lo que se conoce como un *tick* y representa el intervalo de tiempo más pequeño conocido por el sistema.
- La función de la rutina de interrupción del reloj será actualizar el *reloj-calendario* y transferir el control al despachador. Posteriormente el planificador seleccionará la tarea que deberá arrancar en cada *tick* de reloj concreto.
- Las tareas de nivel de reloj se dividen en dos categorías:
  - **Cíclicas**[1]: son las tareas que requieren una sincronización precisa con el mundo exterior.
  - **Delay**[1] (retardo): son las tareas que requieren tener un retardo fijo entre sucesivas activaciones o que retardan sus actividades por un periodo de tiempo.

[1] No necesariamente todas la tareas cíclicas o de retardo tienen por qué ejecutarse a nivel de reloj . Estos nombres indican el tipo por lo que también se podrán tener tareas de estos tipos a nivel de prioridad base.

Tareas cíclicas

- Las prioridades de las tareas cíclicas se suelen ordenar de acuerdo a la precisión de la temporización requerida por cada una de ellas. A la tarea más precisa se le asigna la prioridad más alta.
- La siguiente figura muestra un ejemplo de tareas cíclicas:

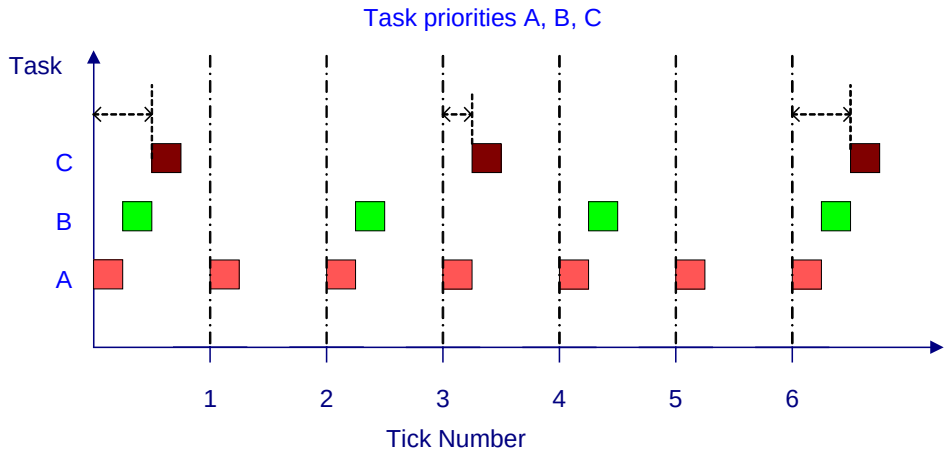
Tareas	**Período (Nº *Ticks*)**
A	**1**
B	**2**
C	**4**



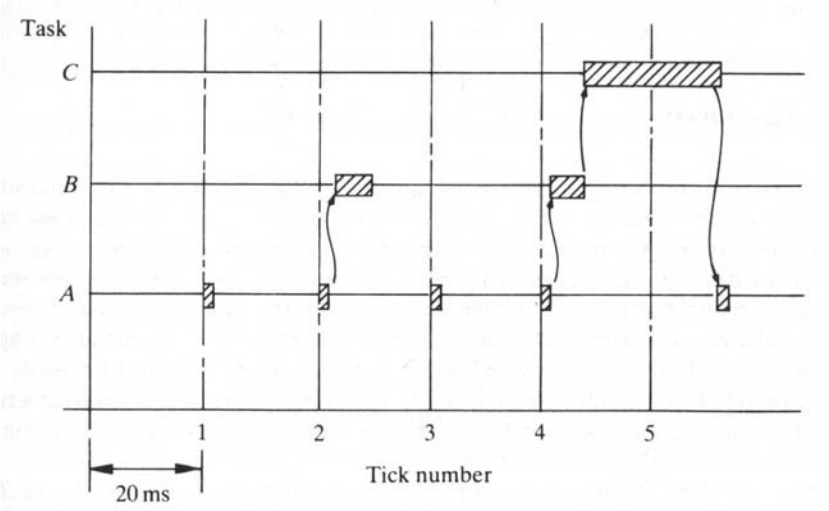
Task activation diagram :
(a) task priorities A,B,C; (b) task priorities C,A,B.

- Las tareas de más baja prioridad tendrán un desplazamiento variable (*jitter*) en su intervalo (período) de ejecución puesto que tendrán que esperar a que finalicen las tareas de prioridad más alta.

Tareas	Periodo (Nº Ticks)
A	1
B	2
C	3



- Uno de los posibles problemas que pueden presentarse es que haya tareas de nivel de reloj que tarden en ejecutarse más de un intervalo entre interrupciones de reloj. Obsérvese que para la operación satisfactoria del sistema tales tareas no pueden ejecutarse con elevada frecuencia. La siguiente figura muestra un ejemplo de esta problemática:



Task activation diagram

Tareas retardadas (*Delay tasks*)

- Cuando una tarea pide un retardo cambia de estado cambiando de ejecutable a suspendida hasta que ha finalizado el retardo.
- En este tipo de tareas que se ejecutan con nivel de prioridad de reloj se encuentran los gestores de temporizaciones que dan facilidades a otras tareas que se ejecutan con prioridad de nivel de base y desean detener su actividad durante un período de tiempo para:
  - Completar un evento externo. Por ejemplo para asegurarse que un *relé* se ha cerrado se debe esperar unos 20 ms.
  - Procesos que solo se ejecutan a intervalos. Por ejemplo para actualizar un *display*.
- Para implementar esta función de retardo lo usual es usar una cola para todos los procesos que lo han pedido. Esta cola puede ser una lista ordenada donde la primera tarea es la que tenga un tiempo de finalización más próximo. Cuando una tarea pide un

retardo llama a una tarea del administrador que calcula el tiempo y se inscribe en la lista.

- Una tarea que se ejecuta a nivel de reloj chequeará la primera tarea de la cola para ver si se ha cumplido el tiempo. Si ha cumplido, la tarea debe ejecutarse, por lo que se quita de la cola y se informa que está lista para ejecutarse.
- Esta tarea que realiza el chequeo puede ser el despachador de tareas que se ejecuta en cada interrupción del reloj de tiempo real u otra tarea que se ejecute a nivel de reloj pero con una frecuencia menor, por ejemplo cada 10 *ticks*, en cuyo caso esta tarea formaría parte del nivel de prioridad base del planificador.
- Muchos sistemas operativos de tiempo real no soportan operaciones cíclicas y debe ser el usuario el que cree una temporización precisa y repetitiva para estas tareas utilizando funciones de retardo.

<u>El planificador y el manejador de la interrupción del reloj de tiempo real</u>

- El manejador de la interrupción de tiempo real así como el planificador seleccionará las tareas de nivel de reloj que deben ejecutarse, y dado que estas rutinas se ejecutarán frecuentemente deben optimizarse sobre todo en el método utilizado para seleccionar la tarea que se ejecutará en cada intervalo para no aumentar el *overhead* del sistema.

3.3. Nivel de prioridad base

- Las tareas que se ejecutan en este nivel se ejecutan por petición más que por un intervalo determinado. Esta petición puede ser una entrada de un usuario desde un terminal, algún evento que espera un proceso, o algún requerimiento particular para un dato que se procesa.
- La manera en la que las tareas se planifican en este nivel varían desde un *round-robin* básico hasta otros más complejos. En los sistemas de tiempo real no se suele utilizar el *round-robin* básico ya que siempre hay actividades más prioritarias que otras, por lo que este algoritmo no suele ser satisfactorio.
- Casi todos los sistemas de tiempo real utilizan una estrategia de prioridad para las tareas de nivel de prioridad base. La dificultad estará en encontrar la estrategia que permita una adecuada operación:
  - Se puede tener una asignación estática de prioridades si la aplicación tiene tareas con una clara jerarquía de prioridades. O se puede tener una asignación dinámica de prioridades para adecuarse a las circunstancias. Por ejemplo: en el nivel más alto del planificador se puede tener una tarea que sea capaz de examinar la utilización de los recursos, chequeando cuanto tiempo están esperando las tareas por un recurso e incrementar la prioridad de las tareas que más tiempo están esperando. Aunque debe tenerse en cuenta que los algoritmos para la asignación no sean muy complicados y no introduzcan un *overhead* significativo sobre el sistema.

4. Asignación de prioridades de razón monótona y deadline monótono. [Burns,97]

- Las estrategias expropiativas suponen que de alguna manera se interrumpe a un proceso que se está ejecutando para ejecutar otro más prioritario. Por tanto esta estrategia implícitamente lleva consigo el concepto de prioridad. Hemos visto que existen dos modelos de prioridad:
  - Prioridades estáticas
  - Prioridades dinámicas
- En lo que sigue estudiaremos dos algoritmos clásicos de planificación en monoprocesadores cuyo objetivo es cumplir todos los *deadline* de las tareas. En estos algoritmos se supone:
  - S1. Las tareas se pueden expropiar en cualquier punto y el coste de la expropiación es despreciable.
  - S2. Las tareas sólo tienen requisitos importantes en cuanto al procesamiento. Los requisitos de memoria, I/O y otros recursos son despreciables.
  - S3. Todas las tareas son independientes.
- Con estas suposiciones se simplifica el análisis de estos algoritmos. La suposición S1 hace que el número de veces que se interrumpe una tarea no influya en la carga del procesador.

4.1. Algoritmo de razón monótona

- El algoritmo de razón monótona (RM) es el algoritmo más estudiado y utilizado en la práctica. Para éste se supone además de S1,S2 y S3 que:
  - S4. Todas las tareas son periódicas.
  - S5. El *deadline* relativo de una tarea es igual a su período.
- Este algoritmo asigna prioridades a las tareas de forma que la prioridad (P) de una tarea es inversamente proporcional a su período (T). Si $T_i<T_j$ entonces $P_i>P_j$. Las tareas de más prioridad pueden interrumpir a las de menos prioridad.
- El esquema de prioridades de RM es óptimo, en el sentido que si cualquier conjunto de procesos es planificable mediante un esquema de prioridades fijas entonces ese conjunto de procesos es planificable utilizando el esquema de RM.

Test de planificabilidad

- A continuación se describirá un test de planificabilidad que aunque no es exacto es atractivo por su sencillez.
- Liu y Layland (1973) mostraron que considerando un conjunto de procesos se puede obtener un test de planificabilidad cuando se usa RM:

  Si se cumple que para N procesos $\sum_{i=1}^{N}\left(\frac{C_i}{T_i}\right) < N(2^{1/N}-1)$ [Ec. 1]

  entonces todos los procesos cumplirán su *deadline*. Donde:

  C_i tiempo de ejecución en el peor caso del proceso i.

  T_i período de lanzamiento del proceso (período del proceso).

- La sumatoria representa la utilización total de CPU del conjunto de procesos. La siguiente tabla muestra el límite de utilización de acuerdo a la ecuación anterior [Ec. 1] para pequeños valores de N

Utilization bounds.

N	*Utilization bound (%)*
1	100.0
2	82.8
3	78.0
4	75.7
5	74.3
10	71.8

- Para valores grandes de N, el límite se aproxima asintóticamente a 69.3%. Si se consigue que para cualquier conjunto de procesos se tenga una utilización menor de 69.3%, los procesos serán siempre planificables con un esquema expropiativo basado en prioridades con prioridades asignadas por el algoritmo de razón monótona.
- Veamos la utilización del test con varios ejemplos.

Ejemplo 1

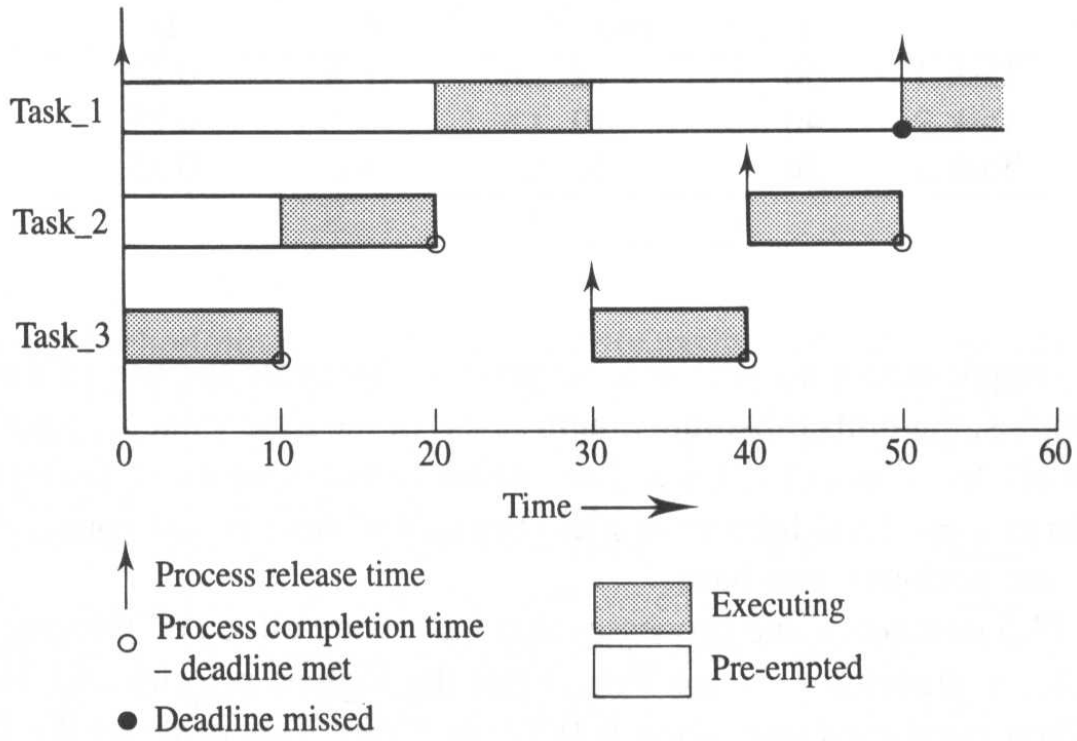
- Se suponen todos los valores en las mismas unidades que puede ser un número entero de *ticks*.

Process set A.

	Period, T	*Computation time, C*	*Priority, P*	*Utilization, U*
Task_1	50	12	1	0.24
Task_2	40	10	2	0.25
Task_3	30	10	3	0.33

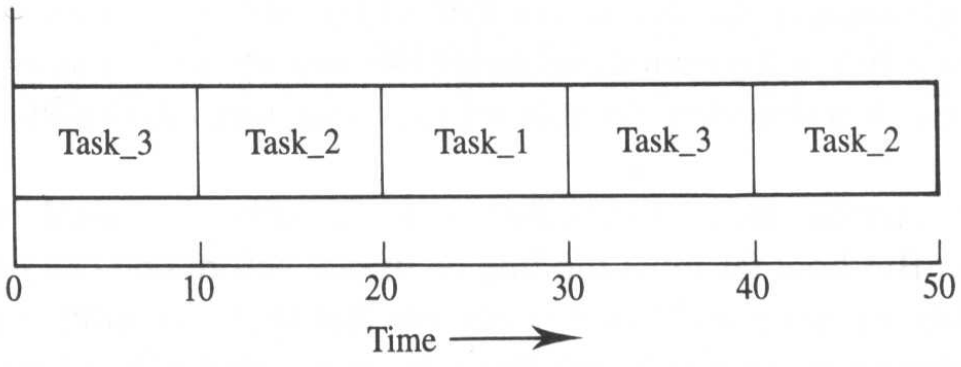
U=C/T

- Si calculamos la expresión [Ec. 1] resulta 0.82 para la sumatoria, que es superior al límite N(21/N -1) para tres procesos 3(21/3 -1) =0.78. Por tanto Para este conjunto de procesos falla el test y habría que observar el comportamiento de estos procesos mediante su *time-line* para comprobar si son planificables o no. La siguiente figura muestra el *time-line* para estos procesos si arranca en el instante 0.



Time-line for process set A.

- En el tiempo 50 Task_1 sólo se ha ejecutado durante 10 *ticks* y necesita 12, por tanto ha perdido su *deadline*.
- La siguiente figura muestra la carta de *Gantt*.



Gantt chart for process set A.

Ejemplo 2

Process set B.

	Period, T	*Computation time, C*	*Priority, P*	*Utilization, U*
Task_1	80	32	1	0.400
Task_2	40	5	2	0.125
Task_3	16	4	3	0.250

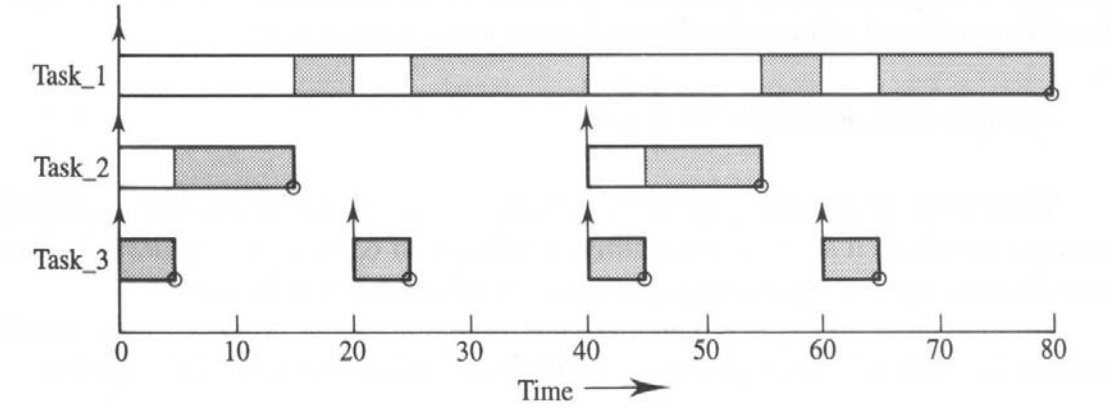
- Si aplicamos el test la utilización es 0.775 que es inferior al límite para los tres procesos (0.78). Por tanto este conjunto de procesos es planificable mediante RM.

- El test descrito es sólo una condición suficiente y no necesaria. Si se cumple el test entonces se cumplen todos los *deadline*, si falla pueden cumplirse o no. En el siguiente ejemplo se tiene un conjunto de procesos con una utilización del 100% y sin embargo son planificables:

Ejemplo 3

Process set C.

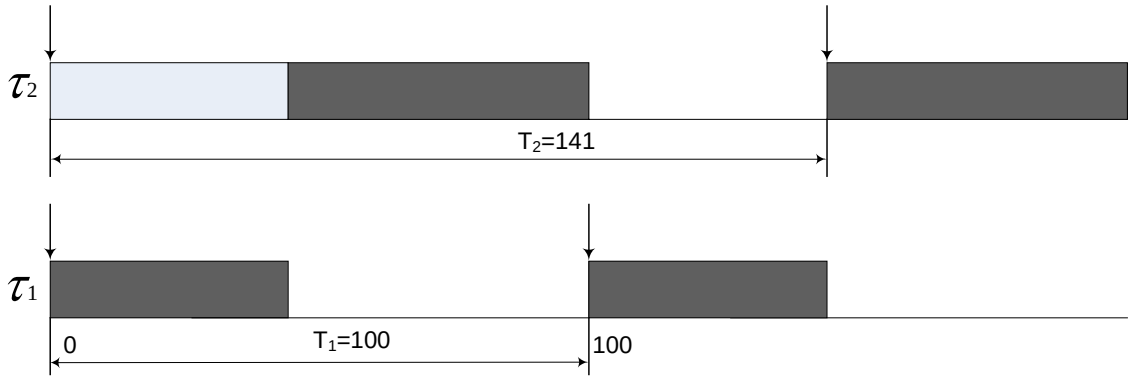
	Period, T	*Computation time, C*	*Priority, P*	*Utilization, U*
Task_1	80	40	1	0.50
Task_2	40	10	2	0.25
Task_3	20	5	3	0.25



Time-line for process set C.

¿Por qué el límite de utilización de CPU en el caso de RM es menor del 100%?

- Veámoslo con un ejemplo sencillo. Consideremos dos tareas periódicas τ_1={C_1 =41, T_1=100}, τ_2 ={C_2=59 , T_2=141}, y obsérvese el diagrama temporal siguiente:



τ_1 ={C_1=41, T_1=100} U=C_1/T_1 =0,41

τ_2 ={C_2=59 , T_2=141} U=C_2/T_2 =0,4184

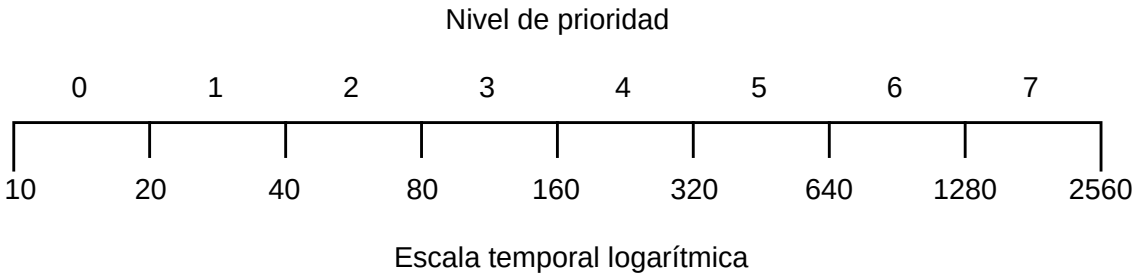
U_{total}=0,41+0,4184=0,8284

- Como puede apreciarse las tareas son planificables por RM. Sin embargo bastaría con una ligero aumento del tiempo de ejecución en alguna de las dos tareas para que no fuesen planificables, estamos por tanto justo en el límite. Si hacemos los cálculos de la utilización resulta 0,8284 que es el límite para dos tareas. La utilización menor del 100% aparece debido a la interferencia que sufre la tarea menos prioritaria.

Mapeo de prioridades RM a un Sistema operativo real

- La asignación de prioridades de acuerdo a RM establece una asignación de prioridades distinta para las tareas con distinto período. Si el número de tareas es elevado son necesarios un gran número de niveles de prioridad. Sin embargo en un sistema operativo de tiempo real comercial el número de niveles de prioridades es limitado. Se hace necesario realizar un mapeo entre los niveles de prioridad teórico y los reales.

- Uno de los mapeos que da mejor resultado es hacer un mapeo de tipo logarítmico. La siguiente figura muestra como se haría un mapeo entre un conjunto de tareas en las que la tarea con menor período fuese de 10 ms y el de mayor período 2.5 sg. a un sistema operativo comercial que tuviera 8 niveles de prioridad. A la tarea de período 10ms se le asigna la prioridad más alta y a la de período de 2.5 sg. se le asigna la prioridad más baja. Al resto de tareas se le asigna un nivel de prioridad de acuerdo a esa escala logarítmica.

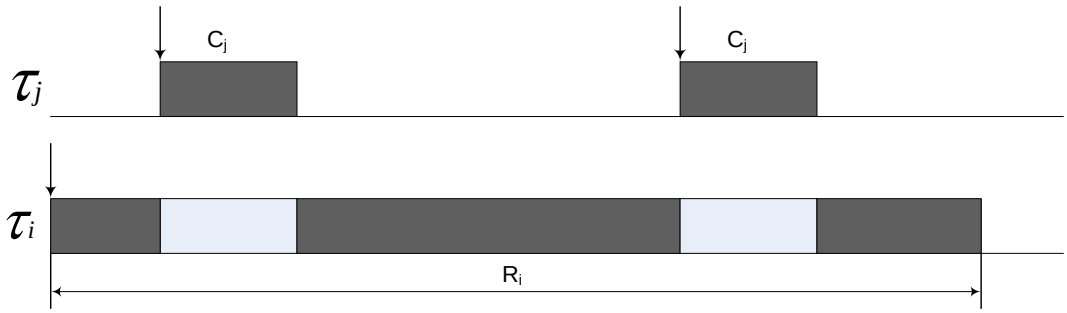


Análisis del tiempo de respuesta

- El test basado en el análisis del tiempo de respuesta da una condición necesaria y suficiente para tener criterios de planificabilidad utilizando RM (y DMPO que se verá más adelante). El test consiste en obtener el peor tiempo de respuesta para cada uno de los procesos y compararlos con su *deadline*. Veamos como se puede obtener una ecuación para el peor tiempo de respuesta:
- El peor tiempo de respuesta de una tarea es la suma de su tiempo de computo más la interferencia que sufre por la

ejecución de tareas más prioritarias. El análisis se realiza analíticamente estudiando el número de veces que se interrumpe y se lanza un proceso.

- Obsérvese la siguiente figura:



Si llamamos:

R_i : tiempo de respuesta de la tarea i

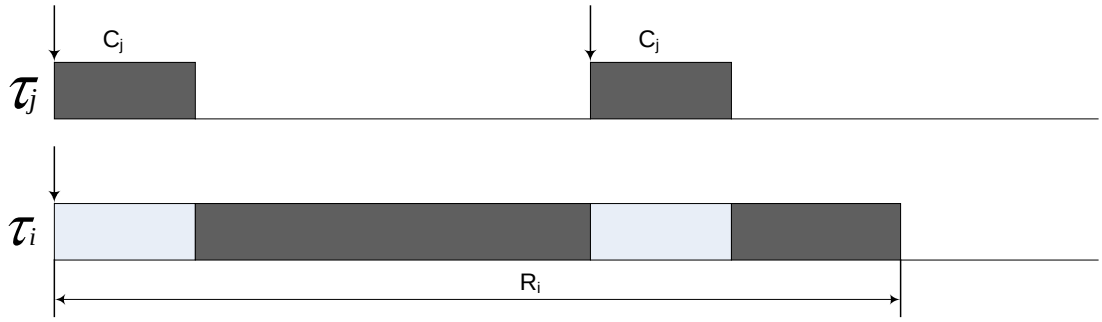
C_i : tiempo de computo de la tarea i

I_i : es la interferencia que sufre la tarea i debido a que la CPU tiene que ejecutar otras tareas más prioritarias

El tiempo de respuesta de la tarea i será:

$$R_i = C_i + I_i \qquad \text{[Ec. 2]}$$

- Veamos a continuación como se calcula la interferencia máxima. Supongamos que una tarea de prioridad superior j interfiere a la tarea en estudio. La duración de la interferencia será:



$$I_i^j = \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \qquad \text{[Ec. 3]}$$

Generalizando para todas las tareas que tienen una prioridad superior, éstas interrumpirán

$$I_i^j = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \quad \text{[Ec. 4]}$$

siendo hp(i) el conjunto de tareas de mayor prioridad que la tarea i

- La ecuación del tiempo de respuesta queda así:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

R_i es la solución mínima de la ecuación

$$W = C_i + \sum_{j \in hp(i)} \left\lceil \frac{W}{T_j} \right\rceil \cdot C_j$$

  - Se trata de una ecuación que no es continua ni lineal
  - No se puede resolver analíticamente
- Se puede resolver mediante iteraciones lineales.
- Se pueden ver ejemplos de resolución en el libro recomendado [burns,97].

Peor caso del tiempo de ejecución

- En todas las versiones de planificabilidad se supone que el tiempo de ejecución de cada proceso es conocido. La estimación del tiempo de ejecución de peor caso se puede obtener por medidas o análisis.
- El problema de las mediciones es que es difícil estar seguro que se ha observado el peor caso. El problema del análisis es que se debe tener un modelo efectivo del procesador (incluidos caché, pipeline, etc.).

4.2. Asignación de prioridades de *deadline* monótono (DMPO).

- El esquema RM es óptimo para D=T (como se supuso). Leung y Whitehead (1982) mostraron que para D<T se puede tener una formulación similar que llamaron "deadline monotonic priority ordering (DMPO)".
- La prioridad de un proceso es inversamente proporcional a su *deadline* ($D_i<D_j$ entonces $P_i>P_j$). La siguiente tabla muestra la asignación de prioridad para un conjunto de procesos. Se incluye también el peor tiempo de respuesta de cada proceso obtenido analíticamente y que como se comentó anteriormente es una condición necesaria y suficiente.
- Obsérvese que con RM (suponiendo el *deadline* igual al período) este conjunto de procesos no son planificables ya que resulta del cálculo del test de planificabilidad una utilización de 90 % superior al valor de 75,4 % para 4 procesos.

Example process set for DMPO.

	Period, T	*Deadline, D*	*Computation time, C*	*Priority, P*	*Response time, R*
Task_1	20	5	3	4	3
Task_2	15	7	3	3	6
Task_3	10	10	4	2	10
Task_4	20	20	3	1	20

- El esquema DMPO es óptimo para cualquier conjunto de procesos. Si Q es planificable por un esquema de prioridades cualquiera W (con las mismas suposiciones), también será planificable por DMPO.

5. Interacción entre procesos y bloqueo.[Burns,97]

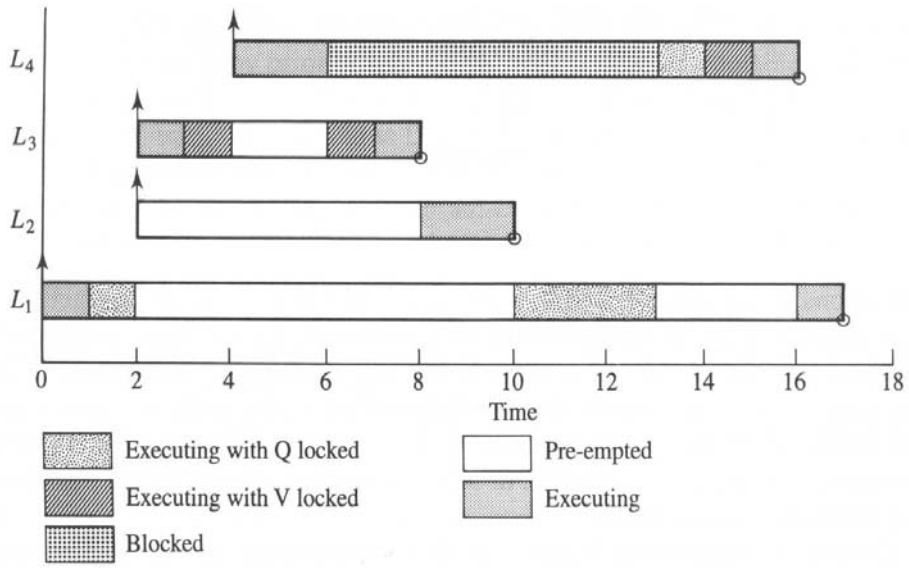
- Se había supuesto al definir los esquemas de planificación clásicos que los procesos debían ser independientes. Esta es una suposición muy optimista, si la suponemos para todas las aplicaciones.
- Ya se había estudiado anteriormente que los procesos pueden interaccionar de una forma segura utilizando algún mecanismo de protección de datos compartidos (semáforos, monitores) o directamente mediante paso de mensajes. También se había estudiado que los procesos debían quedar suspendidos hasta que ocurriera un evento.
- Si un proceso está suspendido esperando a que otro proceso de prioridad más baja complete alguna acción, entonces se dice que se ha producido una **inversión de prioridad**, y en este sentido los modelos estudiados tienen agujeros.
- En un modelo ideal, la inversión de prioridad no debería existir, sin embargo, aunque no puede ser eliminada si que pueden ser minimizados sus efectos.
- Si un proceso espera por causa de otro proceso de más baja prioridad se dice que el proceso está bloqueado. Para realizar los tests de planificabilidad el bloqueo debe acotarse y medirse y debería ser pequeño.
- El siguiente ejemplo muestra una inversión de prioridad:

- Se tienen cuatro procesos L_1, L_2, L_3, L_4
- Se le han asignado prioridades de acuerdo a un esquema de *deadline* monótono. L_4 tiene la prioridad más alta y L_1 la más baja.
- Supóngase que L_4 y L_1 tienen una sección crítica compartida (un recurso) denominado Q protegido por exclusión mutua. Y supóngase que L_4 y L_3 comparten una sección crítica (un recurso) denominado V protegido por exclusión mutua
- La siguiente tabla da los detalles de la secuencia de ejecución. "E" representa un *tick* de tiempo de ejecución y Q (o V) representa un *tick* de tiempo de ejecución con acceso a la sección crítica (recurso). Así L_3 ejecuta 4 *ticks*; el 2º y 3º accediendo a la sección crítica V.

Execution sequences.

Process	*Priority*	*Execution sequence*	*Release time*
L_4	4	EEQVE	4
L_3	3	EVVE	2
L_2	2	EE	2
L_1	1	EQQQQE	0

- La siguiente figura ilustra la secuencia de ejecución:

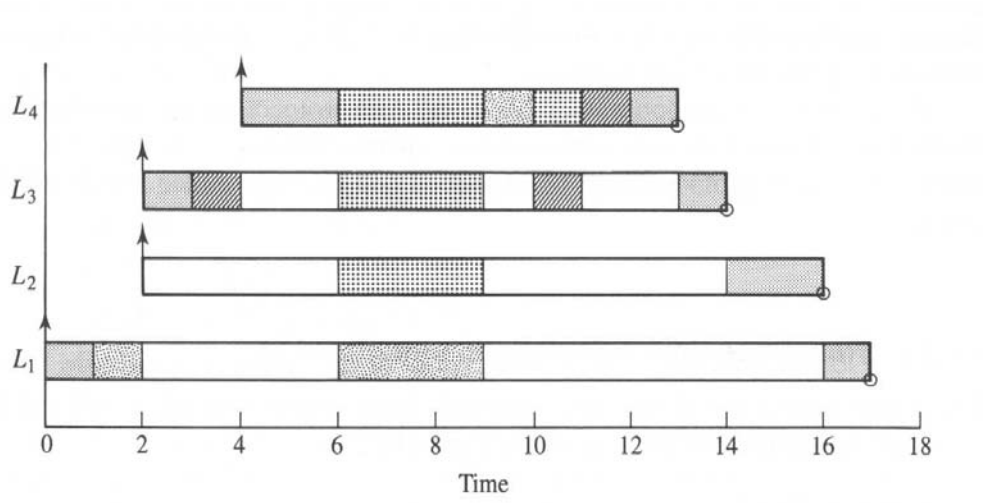


Example of priority inversion.

- Arranca primero L_1 y posteriormente adquiere la sección crítica Q. Posteriormente en el tiempo 2 se ejecuta L_3 que adquiere la sección crítica V. L_4 interrumpe a L_3, comienza a ejecutarse y dado que es la de más alta prioridad se ejecuta hasta que necesita acceder a la sección crítica Q, entonces se suspende porque Q está adquirida por L_1. L_3 retoma la ejecución hasta que finaliza. Comienza L_2 y finaliza también. Continúa entonces L_1 hasta que finaliza la sección crítica Q y continúa entonces L_4 hasta que finaliza. Por ultimo finaliza L_1.

- L_4 no sólo ha sido bloqueada por L_1 sino también por L_2 y L_3. El primer bloqueo es inevitable y la sección crítica de L_1 se debe ejecutar antes que L_4. Sin embargo el bloqueo de L_3 y L_2 es improductivo y afecta severamente a la planificabilidad del sistema.

- Esta inversión de prioridad es causada por un esquema de prioridad fija. Un método que limita el efecto de la inversión de prioridad es utilizar el método de **herencia de prioridad**.
- Con la herencia de prioridad, si un proceso p está suspendido esperando a que otro proceso q realice alguna actividad entonces la prioridad de q se hace igual a la prioridad de P (si era más baja).
- En el ejemplo anterior L_1tendrá la prioridad de L_4 y seguirá ejecutándose antes que L_3 y L_2



Example of priority inheritance.

- El modelo de herencia de prioridad es difícil de implementar en un lenguaje de tiempo real. Si se tienen semáforos estándar y variables de condición no hay enlace directo entre el acto de suspenderse y el proceso que lo debe despertar. En el paso de mensajes si se utiliza un nombramiento indirecto (canal, buzón, etc.) pudiera ser difícil identificar el proceso por el que se está esperando. Para hacer más efectiva la herencia de prioridad

cuando se utiliza paso de mensajes el nombramiento de los procesos debe ser directo y simétrico.

- El protocolo de herencia de prioridad tiene un límite en cuanto al número de veces que puede estar bloqueado por un proceso de prioridad más bajo. Si el proceso tiene *m* secciones criticas que le pueden producir el bloqueo entonces el máximo número de veces que puede estar bloqueado es *m*. Es decir en el peor caso, cada sección crítica será bloqueada por un proceso de prioridad más baja. Si solo hubiera *n* (*n*<*m*) procesos de prioridad más baja entonces se reduciría a *n*. Teniendo en cuenta este esquema de herencia de prioridad se puede acotar el tiempo de bloqueo (consúltese [Burns,97]).

Protocolos de techo de prioridad

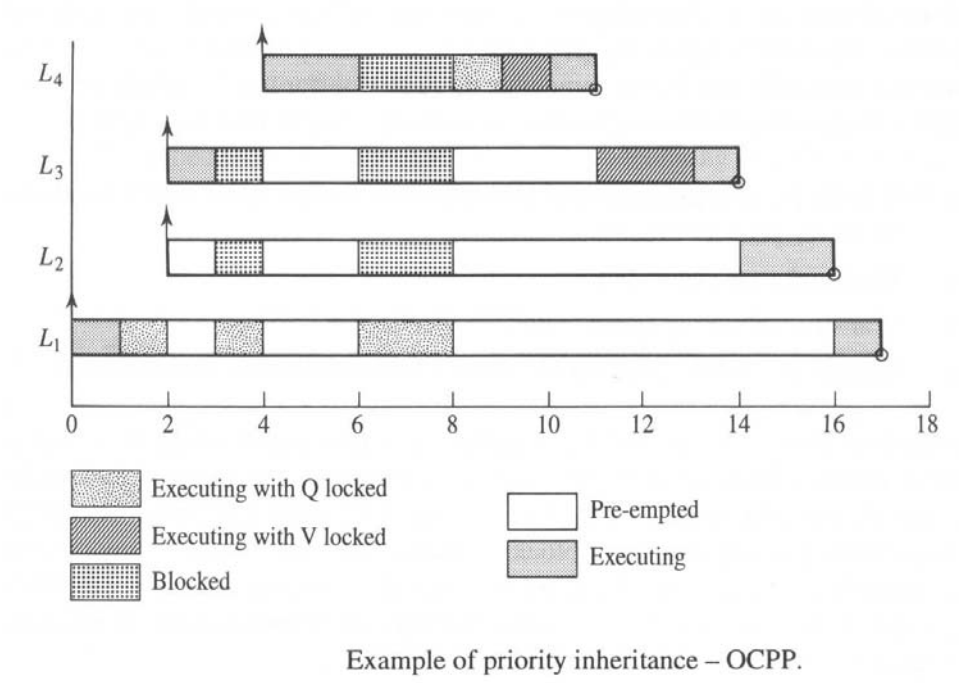
- El protocolo estándar de herencia de prioridad da un límite superior en el número de bloqueos que pueden tener los procesos de alta prioridad. Sin embargo este límite conduce a cálculos demasiado pesimistas. Por otra parte el protocolo de herencia de prioridad puede presentar problemas de *Deadlock*.
- Los dos problemas comentados anteriormente se tienen en cuenta en los protocolos de techo de prioridad. Vamos a considerar dos:
  - El protocolo original de techo de prioridad (OCPP).
  - y el protocolo de techo de prioridad inmediato (ICPP).

- Actúan de la siguiente forma en sistemas monoprocesador:
  - Un proceso de alta prioridad puede ser bloqueado como máximo una vez durante su ejecución por un proceso de más baja prioridad.
  - Se previenen los *deadlock*.
  - Se previenen bloqueos transitivos (ejemplo L_1 bloquea a L_2 y L_2 bloquea a L_3, entonces el resultado es que L_1 bloquea a L_3).
  - El acceso mutuamente exclusivo a los recursos está asegurado (por el propio protocolo).
- Los protocolos de techo de prioridad se describen mejor en términos de recursos protegidos que secciones críticas. En esencia el protocolo asegura que si un recurso se ha adquirido por un proceso P_1 y pudiera provocar el bloqueo de otro proceso de más prioridad P_2 entonces ningún otro recurso que pudiese bloquear P_2 puede ser adquirido por P_1. Un proceso puede ser retardado no sólo por intentar adquirir un recurso que ha sido adquirido previamente sino también cuando la adquisición pudiera conducir a múltiples bloqueos de procesos de mayor prioridad.
- El protocolo funciona así:
  - Cada proceso tiene una prioridad estática asignada por defecto (por ejemplo utilizando un esquema de *deadline* monótono).
  - Cada recurso tiene definido un valor de techo estático que es el valor más alto de entre todas las prioridades de los procesos que pudieran utilizar un recurso.

- Un proceso tiene una prioridad dinámica que es el máximo de su prioridad estática y cualquiera que pudiese heredar debido a un bloqueo que pudiera producir a un proceso de prioridad más alta.
- Un proceso puede adquirir un recurso sólo si su prioridad dinámica es más alta que el techo de cualquier recurso que esté adquirido (excluidos los que han sido adquiridos por él mismo).

- La adquisición de un primer recurso está permitido. El efecto del protocolo es asegurarse que un segundo recurso puede ser solamente adquirido si no existe un proceso de mayor prioridad que utilice ambos recursos.
- El siguiente ejemplo muestra como trabaja este algoritmo:

Sección crítica	Accedido por	Techo de prioridad
Q	L_1 y L_4	$P(L_4)$
V	L_3 y L_4	$P(L_4)$



Example of priority inheritance – OCPP.

- L_1 adquiere la primera sección crítica ya que no hay ningún recurso adquirido.

- Se interrumpe por L_3, pero el intento de coger la sección crítica V no es posible ya que su prioridad (3) no es superior al techo actual (4, ya que Q está adquirido y será utilizado por L_4). Solo se examinan los techos cuando los procesos solicitan recursos.
- L_1 está bloqueando a L_3 y como L_3 se ejecuta al nivel de prioridad 3 bloquea a L_2.
- El proceso de mayor prioridad L_4 interrumpe a L_1 pero queda bloqueado cuando intenta acceder a Q. Por tanto L_1 continuará (con prioridad 4) hasta que libere a Q y su prioridad baje a 1.
- L_4 puede continuar hasta que finalice.

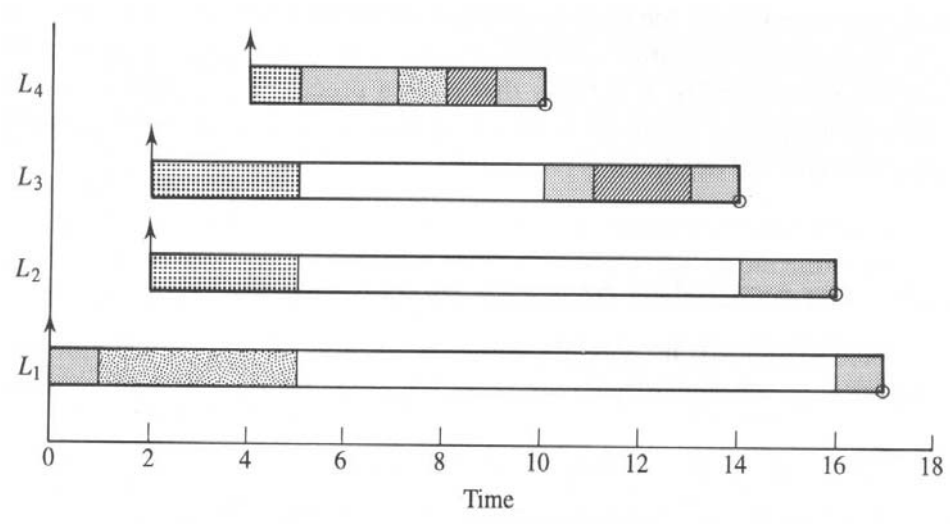
- El protocolo de techo de prioridad asegura que un proceso sólo se bloqueará una vez en cada invocación. En la figura anterior parece que L_2 y L_3 sufrieran dos bloqueos, sin embargo es el mismo bloqueo que se ha dividido en dos por la apropiación de L_4.

Protocolo de techo de prioridad inmediata

- El protocolo de techo de prioridad inmediata aumenta la prioridad de un proceso en cuanto el proceso adquiere el recurso (distinto del protocolo de techo de prioridad que únicamente lo aumentaba cuando estaba bloqueando a un proceso de mayor prioridad). Este protocolo se define:
  - Cada proceso tiene una prioridad estática por defecto (por ejemplo con un esquema de *deadline* monótono).

- Cada recurso tiene un techo de prioridad estático definido de igual forma que en el caso de techo de prioridad, como la máxima prioridad de entre todos los procesos que utilicen este recurso.
- Un proceso tiene una prioridad dinámica que es el máximo de su prioridad estática y los valores de techo de cualquier recurso que adquiera.

- Como consecuencia de la última regla un proceso sufrirá un bloqueo solo al comienzo de su ejecución. Una vez que comienza su ejecución todos los recursos que necesite deben estar libres; Si no lo estuviesen es porque algún proceso tendrá una prioridad (dinámica) igual o mayor y por esto el proceso se pospondrá.
- La siguiente figura muestra como se ejecutarán el conjunto de procesos del ejemplo utilizado en los ejemplos anteriores:



Example of priority inheritance – ICPP.

- L_1 ha adquirido a Q y se ejecuta durante los cuatro tick siguientes con prioridad 4, por lo que ni L_2, L_3,L_4 pueden comenzar.

- Una vez que L_1 libere a Q (se reduce su prioridad) los otros procesos se ejecutan según el orden de prioridad.

- Obsérvese que todos los bloqueos se producen antes de la ejecución y que el tiempo de respuesta del proceso L_4 es ahora de 6 (tiempo 10-tiempo 4).

- Aunque el tiempo de bloqueo del peor caso en este protocolo (ICPP) es el mismo que en el protocolo de techo de prioridad (OCPP) existe algunas diferencias:

  - ICPP es más fácil de implementar que el OCPP ya que no tienen que monitorizarse las relaciones entre procesos cuando se producen bloqueo.

  - ICPP conduce a menos cambios de contexto ya que los bloqueos se producen antes de que comiencen la ejecución de los procesos.

  - ICPP produce más cambios de prioridad (siempre que un proceso utilice o pudiera utilizar un recurso). OCPP sólo cambia la prioridad si ocurre un bloqueo.