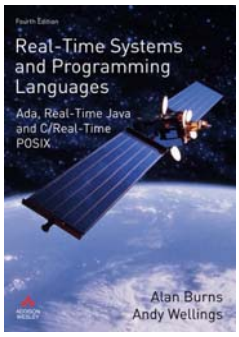
Planificación de tareas de tiempo real

Objetivos

- Plantear los problemas básicos relacionados con el cumplimiento de los requisitos temporales
- Conocer los principales métodos de planificación de tareas y sus características temporales
- Evaluar las ventajas e inconvenientes de los diversos métodos de planificación de tareas

Bibliografía



Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
4th ed. Addison-Wesley, 2009
Capítulo 11

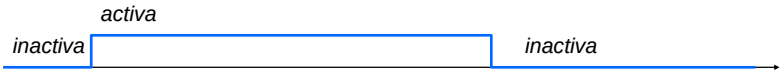
http://www.cs.york.ac.uk/rts/books/RTSBookFourthEdition.html

- *http://www.cs.york.ac.uk/rts/books/RTSBookFourthEdition.html#_Teaching_Aids*
  - *http://www.cs.york.ac.uk/rts/books/RTSbookFourthEdition/slides/*

Tareas y requisitos temporales

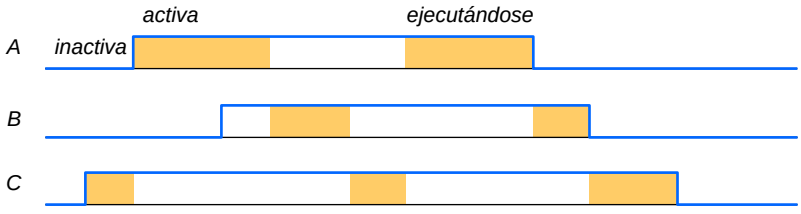
Tareas de tiempo real

- En un sistema de tiempo real se ejecutan una o más *tareas*
- Cada tarea ejecuta una *actividad* de forma repetida
  - cada vez que ejecuta la actividad se produce un *ciclo de ejecución*
  - durante el ciclo de ejecución la tarea permanece *activa*
  - cuando termina la actividad pasa a estar *inactiva* en espera de que comience el siguiente ciclo de ejecución



Concurrencia

- ✓ Los sistemas de tiempo real controlan actividades del mundo exterior que son simultáneas
- ✓ Para ello deben ejecutar varias tareas en paralelo (concurrentemente)
- ✓ La ejecución de las tareas se multiplexa en el tiempo en uno o varios procesadores

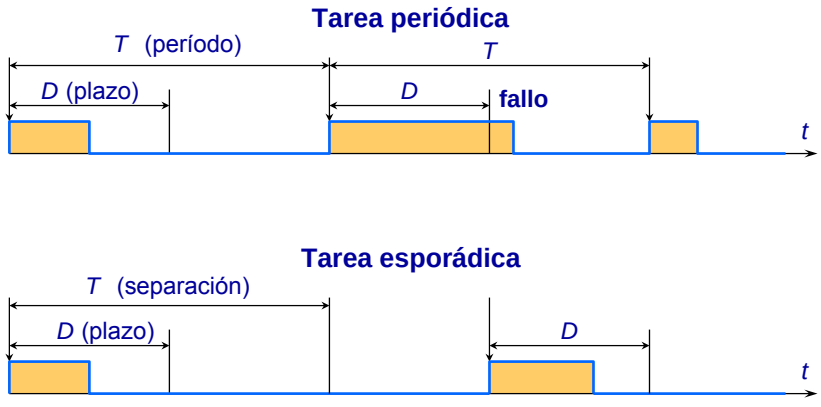


Requisitos temporales

Los requisitos de tiempo real se refieren a

- ✓ El *principio* del ciclo de ejecución (**esquema de activación**)
  - ✓ **Tareas periódicas**: se ejecutan a intervalos regulares
  - ✓ **Tareas esporádicas**: se ejecutan cuando ocurren determinados **sucesos** (en instantes distribuidos irregularmente)
- ✓ El *final* del intervalo de ejecución
  - ✓ Se suele especificar un **plazo** (relativo al instante de activación) para terminar la ejecución

Tareas periódicas y esporádicas



Planificación de tareas

Planificación de tareas

- ✓ Se trata de repartir el tiempo de procesador entre varias tareas ***de forma que se satisfagan los requisitos temporales***
- ✓ La relación biunívoca entre acciones y procesadores es un *plan de ejecución (schedule)*
- ✓ El componente del sistema que hace esto es el *planificador (scheduler)*
  - ✓ para ello utiliza un *algoritmo de planificación*

Esquemas de planificación

- Planificación dirigida por tiempo *(time/clock driven)*
  - el planificador se ejecuta cada vez que llega una señal de reloj
    - ejemplo: planificación cíclica
- Planificación por turno circular *(round robin)*
  - las acciones listas para ejecutarse se agrupan en una cola FIFO
  - cada acción se ejecuta durante una rodaja de tiempo y después se pone al final de la cola
    - variante: rodajas de tiempo desiguales (ponderadas)
- Planificación por prioridades
  - cada acción tiene una prioridad
  - se ejecuta siempre la acción de mayor prioridad entre las listas
    - la planificación está dirigida por sucesos *(event-driven)*

Planificación con y sin desalojo

- Planificación con desalojo (preemptive scheduling)
  - se puede desalojar del procesador una acción que se está ejecutando para dar paso a otra
    - se usa normalmente con prioridades
- Planificación sin desalojo (non-preemptive scheduling)
  - una acción que ha comenzado a ejecutarse sólo deja el procesador si
    - termina su ejecución
    - necesita un recurso que no está disponible
    - abandona el procesador voluntariamente

Prioridades fijas y variables

- ✓ Planificación con prioridades fijas
  - ✓ la prioridad de las acciones de una misma tarea es siempre la misma
    - ✓ puede variar si hay cambios de modo
  - ✓ ejemplo: prioridades monótonas en frecuencia *(rate-monotonic scheduling)*
    - ✓ mayor prioridad a la tarea con período más corto
- ✓ Planificación con prioridades variables (dinámicas)
  - ✓ la prioridad de una acción se decide en el momento de ejecutarla
  - ✓ ejemplo: primero el más urgente *(earliest deadline first)*
    - ✓ mayor prioridad a la acción que deba terminar antes

Modelos de tareas

- ✓ Un modelo de tareas especifica las características de las tareas de un sistema de tiempo real
  - ✓ se restringen para poder analizar el sistema y garantizar los requisitos temporales
- ✓ Ejemplos:
  - ✓ sólo tareas periódicas independientes
  - ✓ tareas periódicas y esporádicas independientes
  - ✓ tareas con comunicación y sincronización
  - ✓ tareas estáticas o dinámicas
- ✓ Empezamos con modelos sencillos
  - ✓ tareas periódicas independientes

Tiempo de cómputo

- Interesa el tiempo de ejecución en el peor caso *(WCET, worst case execution time)*
- Hay dos formas de obtener el WCET de una tarea:
  - **Medida** del tiempo de ejecución
    - no es fácil saber cuándo se ejecuta el peor caso posible
  - **Análisis** del código ejecutable
    - se descompone el código en un grafo de bloques secuenciales
    - se calcula el tiempo de ejecución de cada bloque
    - se busca el camino más largo
  - Puede ser muy pesimista
    - es difícil tener en cuenta los efectos de los dispositivos de hardware (caches, pipelines, estados de espera de la memoria, etc..)
    - hace falta tener un modelo adecuado del procesador

Análisis estático del WCET

- Generalmente se hace en tres pasos:
  1. Descomposición del código en un grafo dirigido compuesto por bloques básicos (secuencias)
  2. Cálculo del WCET de cada bloque básico a partir del código de máquina y del modelo del procesador
  3. Cálculo del WCET total a partir del camino más largo del grafo

Mejora con información semántica

Ejemplo:

```
for I in 1.. 10 loop
  if Cond then
    -- bloque básico con coste 100
  else
    -- bloque básico con coste 10
  end if;
end loop;
```

- ✓ Sin más información, el coste peor es 10 × 100 = 1000
- ✓ Si sabemos que Cond sólo es verdadera 3 veces, entonces el coste es 3×100 + 7×10 = 370

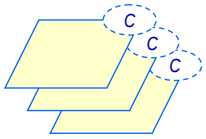
Restricciones en el código

- ✓ Para poder calcular el tiempo de cómputo hay que evitar utilizar estructuras con tiempo de cómputo no acotado, como:
  - ✓ bucles no acotados
  - ✓ recursión no acotada
  - ✓ objetos dinámicos
  - ✓ tareas dinámicas
- ✓ Para construir sistemas de tiempo real estricto se utilizan subconjuntos del lenguaje de programación que no usan estos elementos
- ✓ Ejemplos:
  - ✓ SPARK (parte secuencial de Ada)
  - ✓ Ravenscar (parte concurrente)

Planificación estática

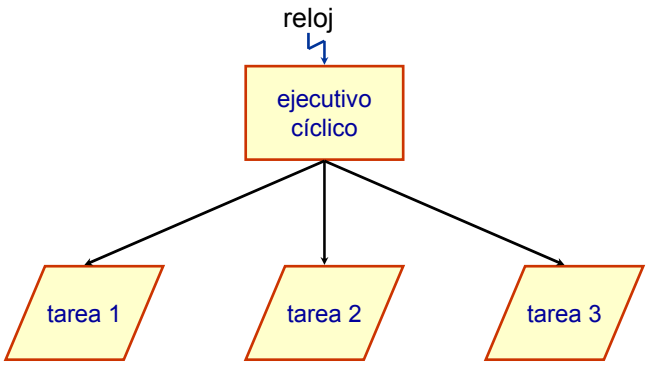
Modelo de tareas cíclico

- ✓ Hay muchos sistemas de tiempo real que sólo tienen tareas periódicas
  - ✓ son más fáciles de construir
  - ✓ su comportamiento está completamente determinado
- ✓ Inicialmente consideramos que no hay comunicación entre tareas (tareas independientes)



Arquitectura síncrona

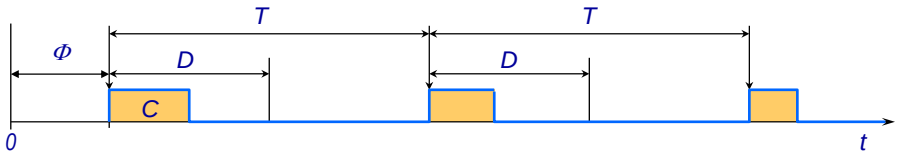
- Las tareas se ejecutan según un **plan de ejecución** fijo (realizado por el diseñador)
- El sistema operativo se reemplaza por un **ejecutivo cíclico**



Parámetros temporales

Una tarea periódica se define por su parámetros (Φ,T,C,D)

Φ es la *fase*
T es el *período* de activación de la tarea
C es su *tiempo de cómputo* en el peor caso
D es el *plazo de respuesta* relativo a la activación

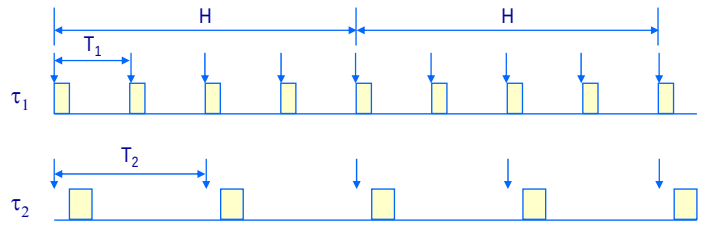


Hiperperíodo

- En un sistema formado únicamente por tareas periódicas con períodos T_i, $i = 1..N$, el comportamiento global se repite con un período

  $H = \text{mcm}(T_i)$

  H es el **hiperperíodo** del sistema



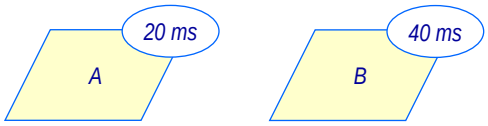
Planificación estática

- ✓ Si todas las tareas son periódicas, se puede confeccionar un *plan de ejecución* fijo
- ✓ Se trata de un esquema que se repite cada

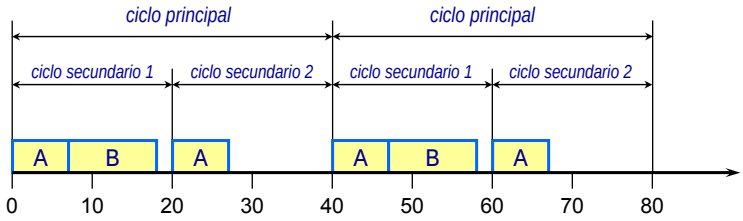
  $T_M = \text{mcm}(T_i)$ *(ciclo principal)*

  ✓ el período del ciclo principal es igual al hiperperíodo del sistema
- ✓ El ciclo principal se divide en *ciclos secundarios*, con período $T_S \quad (T_M = k \cdot T_S)$
- ✓ En cada ciclo secundario se ejecutan las actividades correspondientes a determinadas tareas

Ejemplo 1



Plan cíclico: T_M = 40 ms; T_S = 20 ms



Ejecutivo cíclico

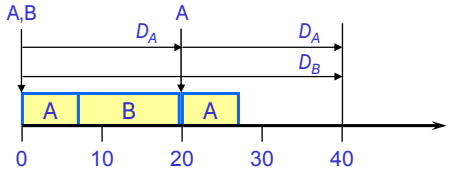
```
procedure Cyclic_Executive is
   type Frame is mod 2;
   Index :Frame := 0;
begin
   Set_Timer (Periodic, 0.020);
   loop
      Wait_Clock_Interrupt; -- cada 20ms
      case Index is
         when 0 => A; B;
         when 1 => A;
      end case;
      Index := Index + 1;
   end loop;
end Cyclic_Executive;
```

Plazos de respuesta

- Se comprueba que se cumplen los plazos directamente sobre el plan de ejecución
- Para ello hace falta conocer el tiempo de cómputo de cada tarea

Ejemplo:

A :	T = 20 ms	D = 20 ms	C = 8 ms
B :	T = 40 ms	D = 40 ms	C = 12 ms



Factor de utilización

- La cantidad $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$

  se denomina ***factor de utilización*** del procesador
- Es una medida de la carga del procesador para un conjunto de tareas
- Para poder elaborar un plan de ejecución que garantice los plazos de todas las tareas,
  debe ser $U \leq 1$

Parámetros del plan cíclico

(1) $T_s \geq \max C_i$

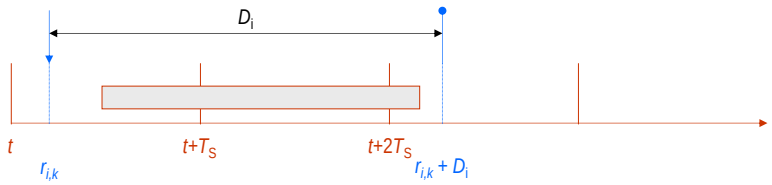
(2) $\exists i: \; T_i / T_S - \lfloor T_i / T_s \rfloor = 0$

(3) $\forall i: \; 2T_S - \text{mcd}(T_S, T_i) \leq D_i$

El período secundario debe cumplir ciertas condiciones (Baker&Shaw, 1989):

1. Todas las acciones deben caber en un marco
2. El período secundario divide al menos al período de una tarea (y por tanto al período principal)
3. Entre el instante de activación y el tiempo límite de cada acción debe haber al menos un marco completo (para poder comprobar si la acción termina a tiempo)
   - si la acción se activa al comienzo de un marco, basta que $T_S \leq D_i$

Condición (3)



$$t + 2T_s \leq r_{i,k} + D_i$$

$$2T_S - (r_{i,j} - t) \leq D_i$$

$$r_{i,j} - t \geq \text{mcd}(T_i, T_S)$$

$$2T_S - \text{mcd}(T_i, T_S) \leq D_i$$

Si $r_{i,k} = t$, basta con que sea $T_S \leq D_i$

Ejemplo 2

Tarea	C	T	D
T1	10	40	40
T2	18	50	50
T3	10	200	200
T4	20	200	200

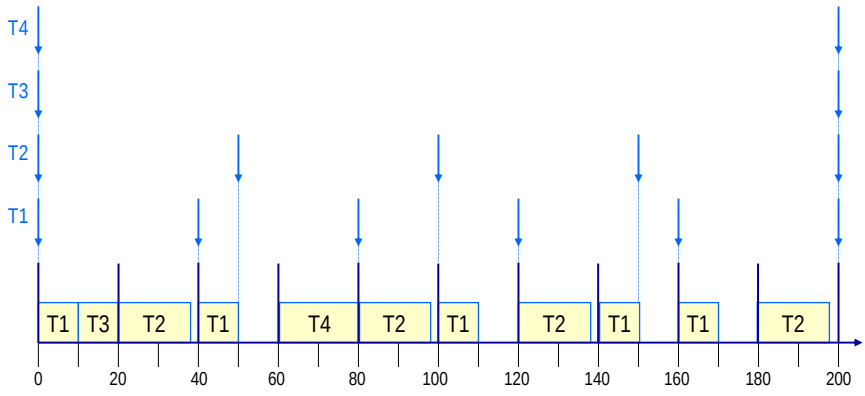
$U = 0{,}76$

$T_M = 200$

1) $T_S \geq 20$
2) $T_S \in \{20,25,40,50,100,200\}$
3) $2T_S - \text{mcd}(T_S,40) \leq 40$
   $2T_S - \text{mcd}(T_S,50) \leq 50$
   $2T_S - \text{mcd}(T_S,200) \leq 200$

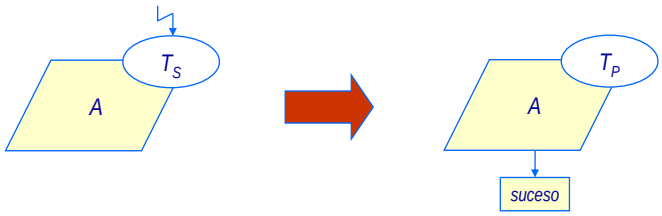
$T_S = 20$

Ejemplo 2: plan cíclico



Tareas esporádicas

- ✓ El ejecutivo cíclico sólo permite ejecutar tareas periódicas
- ✓ Las tareas esporádicas se ejecutan con un **servidor de consulta** *(polling server)*
- ✓ Es una tarea periódica que consulta si se ha producido el suceso esporádico o no
  - ✓ el período depende de la separación mínima entre eventos y del plazo de respuesta

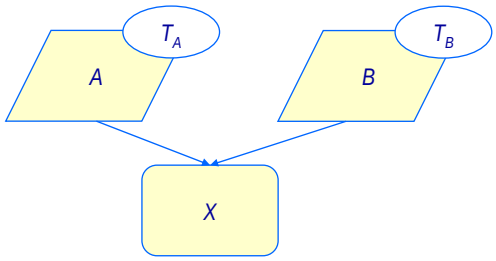


Ejemplo de servidor de consulta

```
procedure Polling_Server is
   Event_Occurred : Boolean := False;
begin
   -- invocado periódicamente por el ejecutivo cíclico
   Check (Event_Occurred);
  if Event_Occurred then
     Sporadic_Activity;
   end if;
end Polling_Server;
```

Recursos compartidos

- ✓ Una tarea (o segmento) se ejecuta sin interrupción hasta que termina
- ✓ No es necesario proteger los recursos compartidos
  - ✓ la exclusión mutua es automática



Segmentación de tareas

- ➢ A veces no es posible confeccionar un plan cíclico que garantice los plazos
- ➢ Si $U \leq 1$, es posible planificar la ejecución *segmentando* una o más tareas
- ➢ Los segmentos son secuencias de instrucciones de la tarea con un tiempo de cómputo conocido

Ejemplo 3

Tarea	C	T	D
τ_1	10	40	40
τ_2	20	100	100
τ_3	50	200	200

$U = 0{,}95$

$T_M = 200$

$$
\begin{aligned}
&1)\quad T_S \geq 50\\
&2)\quad T_S \in \{50,100,200\}\\
&3)\quad 2T_S - \text{mcd}(T_S,40) \leq 40\\
&\qquad 2T_S - \text{mcd}(T_S,100) \leq 100\\
&\qquad 2T_S - \text{mcd}(T_S,200) \leq 200
\end{aligned}
$$

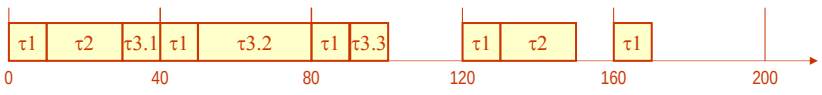
- Ningún valor cumple (1) y (3)
- No hay solución aceptable

Ejemplo 3 — segmentación

Tarea	C	T	D
τ1	10	40	40
τ2	20	100	100
τ3.1	10	200	200
τ3.2	30	200	200
τ3.3	10	200	200

$$
\begin{aligned}
&(1)\quad T_S \geq 30\\
&(2)\quad T_S \in \{40,50,100,200\}\\
&(3)\quad 2T_S - mcd(T_S,40) \leq 40\\
&\qquad 2T_S - mcd(T_S,100) \leq 100\\
&\qquad 2T_S - mcd(T_S,200) \leq 200
\end{aligned}
$$

T_S = 40 cumple todas las condiciones



Ejemplo 3: ejecutivo cíclico

```
procedure Cyclic_Executive is
   type Frame is mod 5;
   Index :Frame := 0;
begin
    loop
       Wait_clock_Interrupt; -- cada 40ms
       case Index is
          when 0 => T1; T2; T3_1;
          when 1 => T1;     T3_2;
          when 2 => T1;     T3_3;
          when 3 => T1; T2;
           when 4 => T1;
       end case;
    end loop;
end Cyclic_Executive;
```


Problemas de la segmentación

- ✓ A veces es difícil ajustar el tiempo de cómputo de los segmentos
- ✓ Si hay recursos compartidos, cada sección crítica debe estar incluida en un solo segmento
- ✓ Si se modifica una sola tarea hay que rehacer la planificación completa
  - ✓ y posiblemente volver a segmentar de otra manera

Construcción del plan cíclico

- ✓ Tres tipos de decisiones interdependientes:
  - ✓ ajustar el tamaño de los marcos
  - ✓ segmentar acciones
  - ✓ colocar los segmentos en marcos
- ✓ En general, el problema es NP duro
  - ✓ no hay algoritmos eficientes que resuelvan todos los casos
- ✓ Se usan algoritmos heurísticos
  - ✓ se construye un árbol de soluciones parciales
  - ✓ se puede empezar colocando las tareas más urgentes
  - ✓ se podan las ramas según algún criterio heurístico
- ✓ Es más fácil cuando el sistema es armónico
  - ✓ pero esto puede forzar una mayor utilización del procesador
- ✓ Cuando los períodos son muy dispares es más difícil
  - ✓ muchos ciclos secundarios en cada ciclo principal

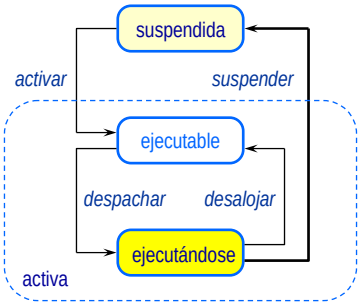
Conclusiones

- ✓ Los sistemas cíclicos, con arquitectura síncrona, tienen muchas ventajas
  - ✓ implementación sencilla y robusta
  - ✓ determinismo temporal
  - ✓ es posible certificar que son seguros
- ✓ Pero tienen inconvenientes importantes
  - ✓ mantenimiento difícil y costoso
    - ✓ si se cambia algo hay que empezar desde el principio
    - ✓ la segmentación añade mucha complejidad
  - ✓ es difícil incluir tareas esporádicas
- ✓ En general, es un método de bajo nivel
  - ✓ sólo es apropiado para sistemas que no se modifican unavez construidos

Planificación con prioridades

Multiprogramación

- ✓ Las tareas se realizan como hebras concurrentes
- ✓ Una tarea puede estar en varios estados
- ✓ Las tareas ejecutables se **despachan** para su ejecución de acuerdo con un *método de planificación:*
  - ✓ prioridades fijas (*fixed-priority scheduling*, FPS)
  - ✓ primero el más urgente (*earliest deadline first*, EDF)
  - ✓ primero el más valioso (*value-based scheduling*, VBS)



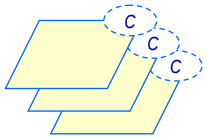
Planificación con prioridades fijas

- ✓ Es el método más corriente en sistemas operativos de tiempo real
- ✓ Cada tarea tiene una prioridad fija
  - ✓ planificación estática
- ✓ Las tareas ejecutables se **despachan** para su ejecución en orden de **prioridad**
- ✓ El despacho puede hacerse
  - ✓ **con desalojo**
  - ✓ **sin desalojo**
  - ✓ **con desalojo limitado**
- ✓ En general supondremos **prioridades fijas con desalojo**
  - ✓ **mejor tiempo de respuesta para las tareas de alta prioridad**

Tareas periódicas

Diseño de sistemas

- ✓ Cuando se diseña un sistema planificado con prioridades fijas hay dos problemas:
  - ✓ cómo asignar prioridades a las tareas
  - ✓ cómo analizar el sistema para ver si se garantizan los requisitos temporales
- ✓ La solución depende del modelo de tareas
- ✓ Empezamos con un modelo sencillo
  - ✓ conjunto estático de tareas periódicas e independientes



Parámetros de planificación

N	Número de tareas
T	Período de activación
C	Tiempo de ejecución máximo
D	Plazo de respuesta
R	Tiempo de respuesta máximo
P	Prioridad

De momento supondremos que para todas las tareas τ_i :

$$C_i \le D_i = T_i$$

Se trata de asegurar que

$$R_i \le D_i$$

Prioridades monótonas en frecuencia

- ✓ La asignación de mayor prioridad a las tareas de menor período *(rate monotonic scheduling)* es **óptima** para un modelo de tareas con
  - ✓ tareas periódicas,
  - ✓ independientes,
  - ✓ con plazos iguales a los períodos

Si se pueden garantizar los plazos de un sistema de tareas con otra asignación de prioridades, se pueden garantizar con la asignación monótona en frecuencia

(Liu & Layland, 1973)

Condición de garantía de los plazos basada en la utilización

- ➢ Para este modelo de tareas, con prioridades monótonas en frecuencia, los plazos están garantizados si

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N \cdot \left(2^{1/N} - 1\right)$$

- ➢ La cantidad

$$U_0(N) = N \cdot \left(2^{1/N} - 1\right)$$

es la **utilización mínima garantizada** para N tareas

Utilización mínima garantizada

N	U0
1	1,000
2	0,828
3	0,779
4	0,756
5	0,743

$$\lim_{n\to\infty} U_0(N) = \log 2 \approx 0{,}693$$

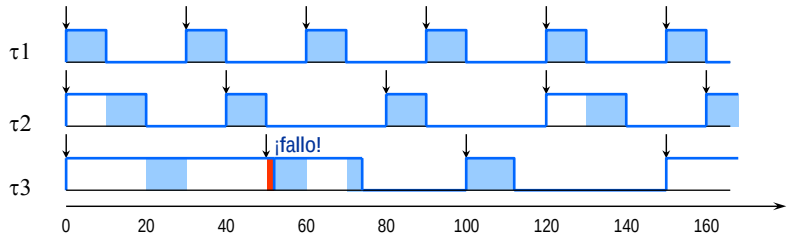
Ejemplo 1

Tarea	T	C	P	U
τ1	30	10	3	0,333
τ2	40	10	2	0,250
τ3	50	12	1	0,240
				0,823

El sistema no cumple la prueba de utilización

(U > 0,779)

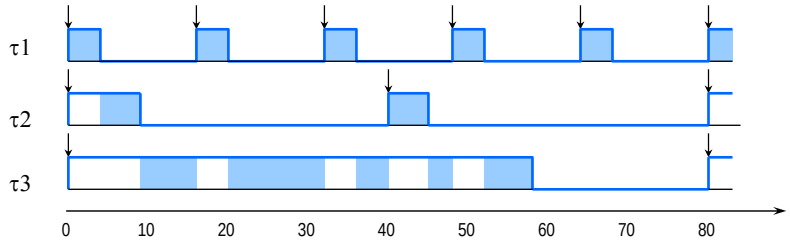
La tarea 3 falla en *t* = 50



Ejemplo 2

Tarea	T	C	P	U
τ1	16	4	3	0,250
τ2	40	5	2	0,125
τ3	80	32	1	0,400
				0,775

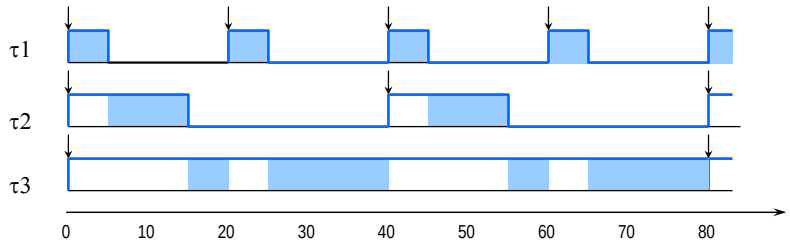
Este sistema está garantizado
(U < 0,779)



Ejemplo 3

Tarea	T	C	P	U
τ1	20	5	3	0,250
τ2	40	10	2	0,250
τ3	80	40	1	0,500
				1,000

Este sistema no pasa la prueba (U > 0,779),
pero se cumplen los plazos



Problemas del análisis de utilización

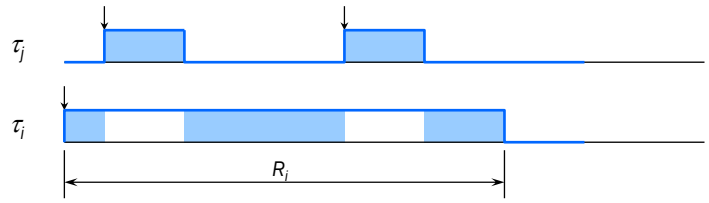
- ✓ La prueba del factor de utilización no es exacta, ni se puede generalizar a modelos de tareas más complejos
  - ✓ pero es eficiente, *O(N)*
- ✓ Veremos una prueba basada en el cálculo del tiempo de respuesta de cada tarea

Análisis del tiempo de respuesta

- ✓ Es un método más completo y flexible que el del factor de utilización para FPS
  - ✓ es fácil de generalizar a otros modelos de tareas
  - ✓ proporciona una condición necesaria y suficiente para que los plazos estén garantizados
- ✓ Se trata de calcular el tiempo de respuesta en el peor caso de cada tarea, R_i, y comprobar directamente que es menor que el plazo correspondiente:

$$R_i \leq D_i$$

Ecuación del tiempo de respuesta

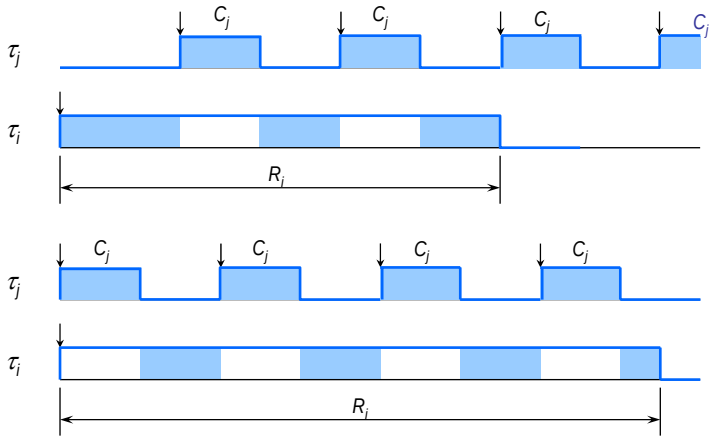


$$R_i = C_i + I_i$$

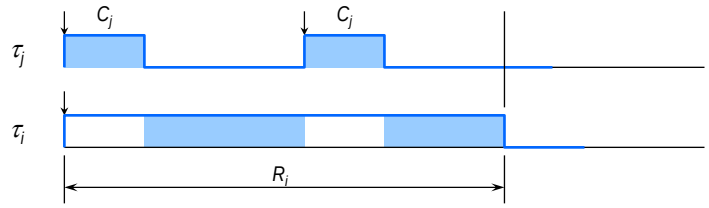
El tiempo de respuesta de una tarea es la suma de su tiempo de cómputo más la interferencia que sufre por la ejecución de tareas más prioritarias

Instante crítico

- ✓ La interferencia es máxima cuando todas las tareas se activan a la vez
  - ✓ el instante inicial se denomina **instante crítico**



Cálculo de la interferencia



- El número de veces que una tarea de prioridad superior τ_j se ejecuta durante el intervalo $[0,R_i)$ es:

$$\left\lceil \frac{R_i}{T_j} \right\rceil$$

función *techo:* $\lceil x \rceil = \min k \in Z : k \geq x$

Por tanto, el valor de la interferencia de τ_j sobre τ_i es

$$I_i^j = \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

Cálculo del tiempo de respuesta

- La interferencia total que sufre τ_i es

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \qquad hp(i) = \{ j : 1..N \mid P_j > P_i \}$$

- La ecuación del tiempo de respuesta queda así:

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

  - La ecuación no es continua ni lineal
  - No se puede resolver analíticamente

Iteración lineal

- ✓ La ecuación del tiempo de respuesta se puede resolver mediante la relación de recurrencia

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \cdot C_j$$

  - ✓ la sucesión ($w_i^0, w_i^1, \ldots w_i^n, \ldots$) es monótona no decreciente
  - ✓ un valor inicial aceptable es $w_i^0 = C_i$
  - ✓ se termina cuando
    - a) $w_i^{n+1} = w_i^n$ (y entonces $R_i = w_i^n$), o bien
    - b) $w_i^{n+1} > T_i$ (no se cumple el plazo)
  - ✓ converge siempre que $U < 100\%$

Ejemplo 4

Tarea	T	C	P	R
τ1	7	3	3	3
τ2	12	3	2	6
τ3	20	5	1	20

$$w_2^0 = 3$$
$$w_2^1 = 3 + \left\lceil \frac{3}{7} \right\rceil \cdot 3 = 6$$
$$w_2^2 = 3 + \left\lceil \frac{6}{7} \right\rceil \cdot 3 = 6; \quad R_2 = 6$$

$$w_3^0 = 5$$
$$w_3^1 = 5 + \left\lceil \frac{5}{7} \right\rceil \cdot 3 + \left\lceil \frac{5}{12} \right\rceil \cdot 3 = 11$$
$$w_3^2 = 5 + \left\lceil \frac{11}{7} \right\rceil \cdot 3 + \left\lceil \frac{11}{12} \right\rceil \cdot 3 = 14$$
$$w_3^3 = 5 + \left\lceil \frac{14}{7} \right\rceil \cdot 3 + \left\lceil \frac{14}{12} \right\rceil \cdot 3 = 17$$
$$w_3^4 = 5 + \left\lceil \frac{17}{7} \right\rceil \cdot 3 + \left\lceil \frac{17}{12} \right\rceil \cdot 3 = 20$$
$$w_3^5 = 5 + \left\lceil \frac{20}{7} \right\rceil \cdot 3 + \left\lceil \frac{20}{12} \right\rceil \cdot 3 = 20 \quad R_3 = 20$$

Todas las tareas tienen sus plazos garantizados

Ejemplo 3 (repaso)

Tarea	T	C	P	U	R
τ1	20	5	3	0,250	5
τ2	40	10	2	0,250	15
τ3	80	40	1	0,500	80
				1,000	

$$R_1 = 5$$

$$w_2^0 = 10$$

$$w_2^1 = 10 + \left\lceil \frac{10}{20} \right\rceil \cdot 5 = 15$$

$$w_2^2 = 10 + \left\lceil \frac{15}{20} \right\rceil \cdot 5 = 15; \quad R_2 = 15$$

$$w_3^0 = 40$$

$$w_3^1 = 40 + \left\lceil \frac{40}{20} \right\rceil \cdot 5 + \left\lceil \frac{40}{40} \right\rceil \cdot 10 = 60$$

$$w_3^2 = 40 + \left\lceil \frac{60}{20} \right\rceil \cdot 5 + \left\lceil \frac{60}{40} \right\rceil \cdot 10 = 75$$

$$w_3^3 = 40 + \left\lceil \frac{75}{20} \right\rceil \cdot 5 + \left\lceil \frac{75}{40} \right\rceil \cdot 10 = 80$$

$$w_3^4 = 40 + \left\lceil \frac{80}{20} \right\rceil \cdot 5 + \left\lceil \frac{80}{40} \right\rceil \cdot 10 = 80 \quad R_3 = 80$$

Todas las tareas tienen sus plazos garantizados

Propiedades del análisis de tiempo de respuesta

- ✓ Proporciona una condición necesaria y suficiente para la garantía de los plazos

$$\forall i \; R_i \leq D_i$$

- ✓ Permite un análisis del comportamiento temporal del sistema más exacto que la prueba del factor de utilización
- ✓ El elemento crítico es el cálculo del tiempo de cómputo de cada tarea
  - ✓ optimista: los plazos pueden fallar aunque el análisis sea positivo
  - ✓ pesimista: él análisis puede ser negativo aunque los plazos no fallen en realidad

Tareas esporádicas y aperiódicas

Tareas esporádicas

- ✓ Para incluir tareas esporádicas hace falta modificar el modelo de tareas:
  - ✓ El parámetro ***T*** representa la **separación** mínima entre dos sucesos de activación consecutivos
  - ✓ Suponemos que en el peor caso la activación es pseudoperiódica (con período *T)*
  - ✓ El plazo de respuesta puede ser menor que el período ($D \leq T$)
- ✓ El análisis de tiempo de respuesta sigue siendo válido
- ✓ Funciona bien con cualquier orden de prioridad

Prioridades monótonas en plazos

Cuando los plazos son menores o iguales que los períodos, la asignación de mayor prioridad a las tareas de menor plazo de respuesta *(deadline monotonic scheduling)* es **óptima**

- ✓ El tiempo de respuesta se calcula de la misma forma que con la asignación monótona en frecuencia
  - se termina cuando $w_i^{n+1} = w_i^n$,
  - o cuando $w_i^{n+1} > D_i$

Ejemplo 5

Tarea	T	D	C	P	R
τ1	20	5	3	4	3
τ2	15	7	3	3	6
τ3	10	10	4	2	10
τ4	20	20	3	1	20

Con prioridades monótonas en frecuencia los plazos no están garantizados:

Tarea	T	D	C	P	R
τ3	10	10	4	4	4
τ2	15	7	3	3	7
τ1	20	5	3	2	10
τ4	20	20	3	1	20

Tareas críticas y acríticas

- A menudo los tiempos de cómputo en el peor caso de las tareas esporádicas son mucho más altos que los medios
  - interrupciones en rachas, tratamiento de errores
  - planteamiento demasiado pesimista
- No todas las tareas esporádicas son críticas
  - Deben garantizarse los plazos de todas las tareas en condiciones "normales"
    - con separación entre activaciones y tiempos de cómputo medios
  - Puede haber una **sobrecarga transitoria**
  - Deben garantizarse los plazos de las tareas críticas en las peores condiciones
    - con separación entre activaciones y tiempos de cómputo peores
  - Esto asegura un comportamiento correcto en caso de sobrecarga transitoria

Tareas aperiódicas

- Son tareas acríticas sin separación mínima
- Se pueden ejecutar con prioridades más bajas que las tareas críticas (periódicas y esporádicas)
  - el tiempo de respuesta puede ser muy largo
  - en condiciones normales sobra tiempo de cómputo de las tareas críticas
- Es mejor utilizar un **servidor**
  - el servidor asegura que las tareas críticas tienen asegurados sus recursos
  - pero asignan los recursos que no se utilizan a las tareas acríticas

Servidor esporádico

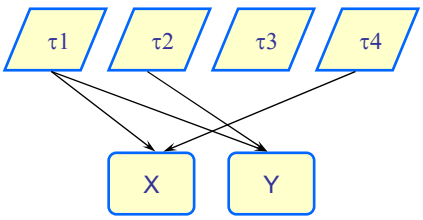
- Un **servidor esporádico** *(SS, sporadic server)* es un proceso periódico
  - Parámetros: período T_s, tiempo de cómputo C_s, prioridad máxima
    - C_s es la *capacidad inicial* del servidor
    - T_s y C_s se eligen de forma que las tareas críticas estén garantizadas
  - Cuando se activa una tarea aperiódica, se ejecuta con prioridad máxima mientras quede capacidad disponible
  - Cuando se agota la capacidad se ejecuta con prioridad baja
  - La capacidad se rellena cuando ha pasado un tiempo T_s desde la activación de la tarea aperiódica

Comunicación entre tareas de tiempo real

Interacción entre tareas

- ✓ En la mayoría de los sistemas de interés práctico las tareas interaccionan mediante
  - ✓ datos comunes (protegidos)
  - ✓ mensajes
- ✓ En todos estos casos puede ocurrir que una tarea tenga que esperar un suceso de otra menos prioritaria
- ✓ Esta situación se denomina **bloqueo**, y produce una **inversión de prioridad** indeseable
- ✓ La inversión de prioridad no se puede eliminar completamente, pero es posible limitar su duración

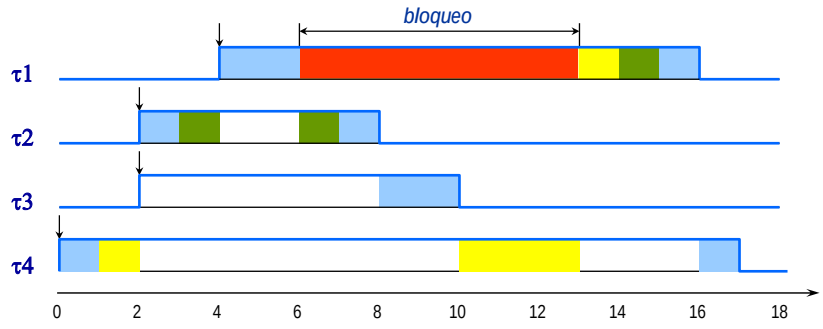
Ejemplo



Tarea	P	ta	Acciones
τ1	4	4	NNXYN
τ2	3	2	NYYN
τ3	2	2	NN
τ4	1	0	NXXXXN

N: ejecución de código propio
X: ejecución con acceso a X
Y: ejecución con acceso a Y
(durante 1 unidad de tiempo)

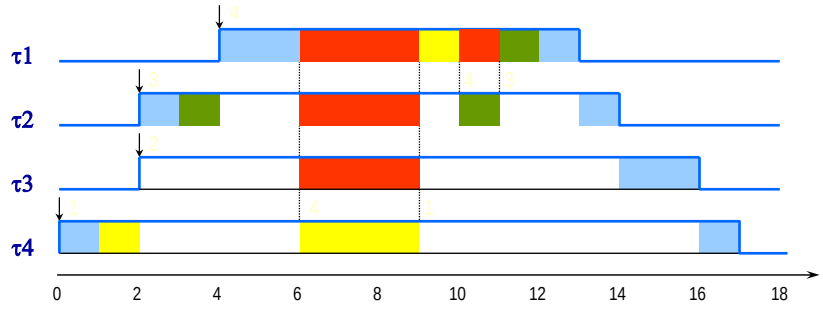
Ejemplo: inversión de prioridad



Herencia de prioridad

- ✓ Una forma de reducir la duración de los bloqueos es variar dinámicamente la prioridad de las tareas
- ✓ Cuando una tarea está bloqueando a otra más prioritaria, **hereda** la prioridad de ésta
- ✓ La prioridad dinámica de una tarea es el máximo de
  - ✓ su prioridad básica
  - ✓ las prioridades de todas las tareas bloqueadas por ella
- ✓ La herencia de prioridad es **transitiva**

Ejemplo: herencia de prioridad



Duración máxima del bloqueo

- ✓ Con el protocolo de herencia de prioridad, una tarea se pude bloquear como máximo
  - ✓ una vez por cada recurso
  - ✓ una vez por cada tarea de prioridad inferior
- ✓ La duración total máxima de los bloqueos es

$$B_i = \sum_{k=1}^{K} u(k,i) \cdot C(k)$$

K = número de secciones críticas

$u(k,i)$ = 1 si

- la tarea que ejecuta k es τ_j, $P_j < P_i$ y
- algún τj, Pj >= Pi, accede al recurso que se usa en k

0 si no

C_k = tiempo de ejecución de la sección crítica k

Ejemplo : cálculo del bloqueo

$$K = 4 \qquad (X_1, Y_1, Y_2, X_4)$$

$$B_1 = 0 \cdot 1 + 0 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 = 6$$

$$B_2 = 0 \cdot 1 + 0 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 = 4$$

$$B_3 = 0 \cdot 1 + 0 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 = 4$$

$$B_4 = 0 \cdot 1 + 0 \cdot 1 + 0 \cdot 2 + 0 \cdot 4 = 0$$

- Una tarea puede bloquearse por recursos a los que no accede (por ejemplo, τ_2)
- Una tarea puede sufrir bloqueo aunque no acceda a recursos compartidos (por ejemplo, τ_3)
- La tarea de menor prioridad (τ_4) no sufre bloqueo

Tiempo de respuesta con bloqueos

- Cuando hay bloqueos, la ecuación del tiempo de respuesta queda así:

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

La solución se obtiene mediante la relación de recurrencia

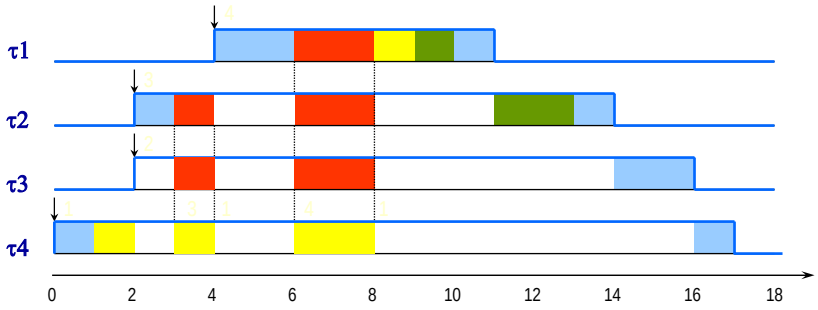
$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \cdot C_j$$

¡Ahora el cálculo puede ser pesimista!

Protocolos de techo de prioridad

- El **techo de prioridad** *(ceiling priority)* de un recurso es la máxima prioridad de las tareas que lo usan
- El **protocolo del techo de prioridad** ***(CPP****, ceiling priority protocol)* consiste en :
  - la prioridad dinámica de una tarea es el máximo de su prioridad básica y las prioridades de las tareas a las que bloquea
  - una tarea sólo puede usar un recurso si su prioridad dinámica es mayor que el techo de todos los recursos en uso por otras tareas

Ejemplo : techo de prioridad



Propiedades

- ✓ Cuando se usa el protocolo del techo de prioridad **en un sistema monoprocesador,**
  - ✓ Cada tarea se puede bloquear una vez, como máximo, en cada ciclo
  - ✓ No puede haber interbloqueos
  - ✓ No puede haber bloqueos encadenados
- ✓ La duración máxima del bloqueo es ahora

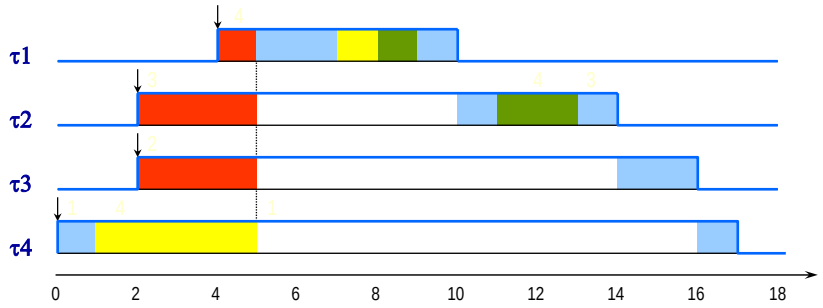
$$B_i = \max_{K \in (1..K)} u(k,i) \cdot C_k$$

Protocolo del techo de prioridad inmediato *(ICPP)*

- ✓ Con este protocolo, una tarea que accede a un recurso hereda inmediatamente el techo de prioridad del recurso
  - ✓ la prioridad dinámica de una tarea es el máximo de su prioridad básica y los techos de prioridad de los recursos que usa
- ✓ Las propiedades son las mismas que las del protocolo del techo de prioridad, y además,
  - ✓ si una tarea se bloquea, lo hace al principio del ciclo
- ✓ La duración máxima del bloqueo es igual que en CPP:

$$B_i = \max_{K \in (1..K)} u(k,i) \cdot C_k$$

Ejemplo : techo de prioridad inmediato



Ejemplo : cálculo del bloqueo con ICPP

$$K = 4 \qquad (X_1, Y_1, Y_2, X_4)$$

$$B_1 = \max(0 \cdot 1, 0 \cdot 1, 1 \cdot 2, 1 \cdot 4) = 4$$

$$B_2 = \max(0 \cdot 1, 0 \cdot 1, 0 \cdot 2, 1 \cdot 4) = 4$$

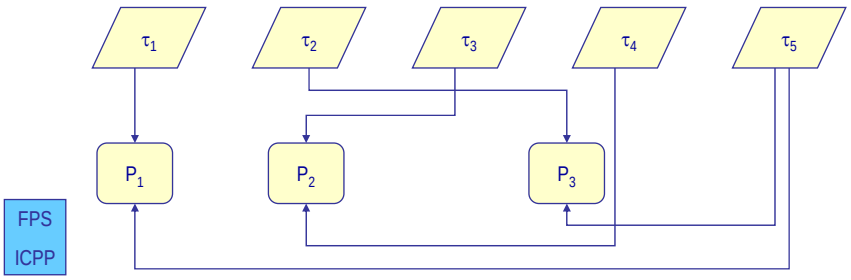
$$B_3 = \max(0 \cdot 1, 0 \cdot 1, 0 \cdot 2, 1 \cdot 4) = 4$$

$$B_4 = \max(0 \cdot 1, 0 \cdot 1, 0 \cdot 2, 0 \cdot 4) = 0$$

CPP e ICPP

- ✓ Los dos protocolos tienen las mismas propiedades,pero
  - ✓ ICPP es más fácil de realizar
    - ✓ no hay que seguir las relaciones de bloqueo transitivas
  - ✓ ICPP produce menos cambios de contexto
    - ✓ el bloqueo se produce antes de la ejecución
  - ✓ ICPP produce más cambios de prioridad
    - ✓ se hereda la prioridad techo aunque no haya bloqueo
- ✓ El protocolo ICPP se conoce también con otros nombres:
  - ✓ Ceiling Locking (Ada 95)
  - ✓ Priority Protect Protocol (POSIX)
  - ✓ Priority Ceiling Emulation (RT Java)
  - ✓ Highest Locker

Ejemplo



FPS
ICPP

τ		Atributos temporales			Acceso a OP		
		T	C	D	P1	P2	P3
τ1	S	120	2	5	1	—	—
τ2	P	50	10	50	—	—	1
τ3	P	30	6	30	—	1	—
τ4	S	300	16	32	—	2	—
τ5	S	120	12	15	2	—	2

Ejemplo — Prioridades

- Las prioridades de las tareas se asignan por DMS (más prioridad a las tareas de menor plazo)
- El techo de prioridad de cada objeto protegido es la prioridad máxima de las tareas que lo usan

τ		Atributos temporales				Acceso a OP		
		P	T	C	D	P1	P3	P2
τ1	S	**5**	120	2	5	1	—	—
τ5	S	**4**	120	12	15	2	2	—
τ3	P	**3**	30	6	30	—	—	1
τ4	S	**2**	300	16	32	—	—	2
τ2	P	**1**	50	10	50	—	1	—
					CP	**5**	**4**	**3**

Ejemplo — Bloqueos

$$B_i = \max_{K \in (1..K)} u(k,i) \cdot C_k$$

τ		Atributos temporales					Acceso a OP		
		P	T	C	B	D	P1	P3	P2
τ1	S	**5**	120	2	**2**	5	1	—	—
τ5	S	**4**	120	12	**1**	15	2	2	—
τ3	P	**3**	30	6	**2**	30	—	—	1
τ4	S	**2**	300	16	**1**	32	—	—	2
τ2	P	**1**	50	10	**0**	50	—	1	—
						CP	**5**	**4**	**3**

Ejemplo — Tiempos de respuesta

$$R_1 = C_1 + B_1 = 2 + 2 = 4$$

$$R_5 = C_5 + B_5 + \left\lceil \frac{R_5}{T_1} \right\rceil \cdot C_1 = 12 + 1 + \left\lceil \frac{R_5}{120} \right\rceil \cdot 2$$

$$R_3 = C_3 + B_3 + \left\lceil \frac{R_3}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_3}{T_5} \right\rceil \cdot C_5 = 6 + 2 + \left\lceil \frac{R_3}{120} \right\rceil \cdot 2 + \left\lceil \frac{R_3}{120} \right\rceil \cdot 12$$

$$R_4 = C_4 + B_4 + \left\lceil \frac{R_4}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_4}{T_5} \right\rceil \cdot C_5 + \left\lceil \frac{R_4}{T_3} \right\rceil \cdot C_3 = 16 + 1 + \left\lceil \frac{R_4}{120} \right\rceil \cdot 2 + \left\lceil \frac{R_4}{120} \right\rceil \cdot 12 + \left\lceil \frac{R_4}{30} \right\rceil \cdot 6$$

$$R_2 = C_2 + B_2 + \left\lceil \frac{R_2}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_2}{T_5} \right\rceil \cdot C_5 + \left\lceil \frac{R_2}{T_3} \right\rceil \cdot C_3 + \left\lceil \frac{R_2}{T_4} \right\rceil \cdot C_4 = 10 + 0 + \left\lceil \frac{R_2}{120} \right\rceil \cdot 2 + \left\lceil \frac{R_2}{120} \right\rceil \cdot 12 + \left\lceil \frac{R_2}{30} \right\rceil \cdot 6 + \left\lceil \frac{R_2}{300} \right\rceil \cdot 16$$

τ		Atributos temporales						Acceso a OP		
		P	T	C	B	R	D	P1	P3	P2
τ1	S	5	120	2	2	4	5	1	—	—
τ5	S	4	120	12	1	15	15	2	2	—
τ3	P	3	30	6	2	22	30	—	—	1
τ4	S	2	300	16	1	43	32	—	—	2
τ2	P	1	50	10	0	52	50	—	1	—
							CP	5	4	3

Resumen

- ✓ Cuando dos o más tareas se sincronizan se produce una *inversión de prioridades*
- ✓ Se puede limitar la duración de los bloqueos producidos por la inversión de prioridades utilizando un protocolo adecuado (PIP, CPP, ICPP)
  - ✓ sólo sirve en sistemas
    - ✓ planificados con prioridades fijas
    - ✓ con un conjunto estático de tareas periódicas y esporádicas
    - ✓ con comunicación mediante datos comunes protegidos (exclusión mutua)
  - ✓ no sirve cuando hay sincronización condicional, paso de mesajes u otras formas de sincronización

Planificación en POSIX

Modelo básico

- POSIX define un modelo de planificación con prioridades y varias políticas secundarias
  - FIFO
    - » una hebra (o un proceso) se ejecuta hasta que termina o se suspende
    - » puede ser desalojada por otra hebra (o proceso) de mayor prioridad
  - Round-Robin
    - » una hebra (o proceso) se ejecuta hasta que termina, se suspende, o expira su *cuanto* de tiempo de ejecución
  - Servidor esporádico
  - Otras, definidas por la implementación
- Las prioridades se pueden cambiar dinámicamente
- Se pueden especificar varios protocolos de acceso para los cerrojos
  - herencia de prioridad
  - techo de prioridad inmediato

Política de planificación

- La política de planificación dentro del mismo nivel de prioridad se define con

```
int pthread_attr_setschedpolicy (
   pthread_attr_t *attr,
   int policy);
```

- El parámetro `policy` puede ser
  - SCHED_FIFO (orden de llegada)
  - SCHED_RR (turno circular con cuantos de tiempo)
  - SCHED_SPORADIC (servidor esporádico)
  - SCHED_OTHER (dependiente de la implementación)

- Los atributos se usan al crear la hebra

Prioridades

- La prioridad de una hebra es un parámetro de planificación:

```
struct sched_param {
  …
  int sched_priority;
  …
}
```

y se especifica con

```
int pthread_attr_setschedparam (
                   const pthread_attr_t *attr,
                   const struct sched_param *param);
```

Protocolo de acceso a cerrojos

- Se puede especificar un protocolo de acceso con

```
int pthread_mutexattr_setprotocol (
                    pthread_mutex_attr_t *attr,
                    int *protocol);
```

- El protocolo puede ser
  - PTHREAD_PRIO_INHERIT (herencia de prioridad)
  - PTHREAD_PRIO_PROTECT (techo de prioridad inmediato)
  - PTHREAD_PRIO_NONE (no hay herencia de prioridad)

Prioridad techo

- Se especifica con

```
int pthread_mutexattr_setprioceiling (
                  pthread_mutexattr_t *attr,
                  int prioceiling);
```

Los atributos se usan para iniciar el cerrojo

Servidor esporádico

- Un servidor esporádico es una hebra que tiene una capacidad limitada para ejecutar tareas esporádicas con prioridad alta
  - si se agota la capacidad se sigue ejecutando con prioridad baja
  - parámetros: capacidad, intervalo de relleno, prioridades alta y baja

Resumen

- La interfaz de POSIX define un modelo de planificación flexible
- Se puede ajustar de forma que se pueda analizar el tiempo de respuesta de las tareas
  - planificación SCHED_FIFO y PTHREAD_PRIO_PROTECT

Gestión del tiempo en POSIX

Relojes en C / POSIX

Hay dos tipos de relojes

- **Reloj calendario (ANSI C)**
  - Proporciona valores de tiempo con resolución de 1s
- **Relojes de tiempo real (POSIX)**
  - Se pueden definir distintos relojes
  - Estándar: ***CLOCK_REALTIME*** **y** ***CLOCK_MONOTONIC***
  - La resolución de la representación es de 1ns
  - La granularidad depende de la implementación
    - » como máximo 20 ms

Reloj-calendario

```
typedef … time_t;
struct tm {
  int tm_Sec;  /* seconds since the hour   [0,60] */
  int tm_min;  /* minutes since the second [0,59] */
  int tm_hour; /* hours since midnight     [0,23] */
  int tm_mday; /* day of the month         [1,31] */
  int tm_ymon; /* months since January     [0,11] */
  int tm_year; /* year since 1900 */
  int tm_day;  /* days since Sunday        [0.6]  */
  int tm_yday; /* days since January 1     [0.365]*/
  int tm_isdst; /* DST flag */
}
double difftime (time_t time2,time_t time1);
time_t mktime (struct tm* tp);
time_t time (time_t* tp);
```


Relojes de tiempo real

- El tiempo se representa mediante el tipo *timespec*

```
struct timespec {
  time_t tv_sec;    /* segundos */
  long   tv_nsec;  /* nanosegundos */
 }
typedef .. clockid_t
```

El tipo *clockid_t* sirve para identificar distintos relojes

- *CLOCK_REALTIME* y *CLOCK_MONOTONIC* son constantes de este tipo
- Puede haber otros relojes

Operaciones con relojes

- Leer la hora:

```
int clock_gettime (clockid_t clockid,
                   struct timespec *tp);
```

- Poner en hora

```
int clock_settime (clockid_t clockid,
                   struct timespec *tp);
```

  - no se puede usar con *CLOCK_MONOTONIC*

- Resolución del reloj

```
int clock_getres (clockid_t clockid,
                  struct timespec *res);
```


Retardo relativo

- La función

```
unsigned sleep (unsigned seconds);
```

suspende la hebra que la invoca durante un número entero de segundos

- La función

```
int nanosleep (const struct timespec *rqtp,
               struct timespec *rmtp;)
```

permite especificar retardos con mayor precisión

  - la resolución es la de *CLOCK_REALTIME*)
  - la duración del retardo es *rqtp*
  - *rmtp* es el tiempo que queda si el retardo se interrumpe

Retardos con relojes de tiempo real

- La función

```
int clock_nanosleep (clockid_t clock_id,
                     int       flags,
                     const struct timespec *rtpq,
                     struct timespec *rmpt);
```

permite especificar el reloj en que se basa el retardo

  - por ejemplo, *CLOCK_MONOTONIC*
  - si se pone *TIMER_ABSTIME* en *flags* el retardo es absoluto

Ejemplo: tarea periódica

```
void periodic () {
  struct timespec next, period;

  if (clock_gettime (CLOCK_MONOTONIC, &next) != 0) error();
  period.tv_sec  = 0;
  period.tv_nsec = 10.0E6; /* 10 ms */

  while (1) {
    if (clock_nanosleep (CLOCK_MONOTONIC, TIMER_ABSTIME,
                         &next, 0) != 0) error();
    acción periódica
    next = next + period;
  }
}
```

Espera temporizada (1)

- La función

```
int pthread_cond_timedwait (pthread_cond_t  *cond,
                            pthread_mutex_t *mutex,
                      const struct timespec *abstime);
```

  permite limitar el tiempo durante el cual se espera una condición
- El límite es absoluto y su valor es *abstime*
- El reloj al que se refiere es un atributo de la condición

Espera temporizada (2)

- Para limitar el tiempo de espera de una señal se usa

```
int sigtimedwait (const sigset_t  *set,
                        siginfo_t *info,
                  const struct timespec *timeout);
```

- Aquí *timeout* es un intervalo de tiempo relativo, medido con *CLOCK_MONOTONIC*

Temporizadores

- Se pueden crear temporizadores asociados a relojes
- Cada temporizador se identifica mediante un valor del tipo *timer_t*
- El tiempo de espera se especifica mediante un valor de tipo *itimerspec:*


```
struct itimerspec {
   struct timespec it_value;    /* expiración */  struct
   timespec it_interval; /* período */
   }
```

Creación de temporizadores

- La función

```
int timer_create (clockid_t clock_id,
            struct sigevent *evp,
                    timer_t *timerid);
```

crea un temporizador asociado al reloj *clock_id*

- **evp* indica el tipo de notificación que se produce al expirar el temporizador
- el identificador del temporizador se devuelve en *timer_id*

Armar un temporizador

- Se usa la función

```
int timer_settime (timer_t timerid,
                   int flags,
   const struct itimerspec *value,
         struct itimerspec *ovalue);
```

  - La temporización puede ser relativa o absoluta, según el valor de *flags*
  - El funcionamiento se repite periódicamente si *value.it_period* > 0
  - En **ovalue* se devuelve el valor que quedaba de la temporización anterior

Ejemplo: tarea periódica (1)

```
void periodic () {
  int signum;            /* señal recibida */
  sigset_t set;          /* señales a las que se espera */
  struct sigevent sig;   /* información de señal */

  timer_t timer;
  struct itimerspec required, old;
  struct timespec first, period;

  sig.sigev_notify = SIGEV_SIGNAL;
  sig.sigev_signo  = SIGRTMIN;

  if (clock_gettime (CLOCK_MONOTONIC, &first) != 0) error();
  first.tv_sec   = first.tv_sec + 1;
  period.tv_sec  = 0;
  period.tv_nsec = 10.0E6;   /* 10 ms */
  required.it_value = first;
  required.it_interval = period;
```

Ejemplo: tarea periódica (2)

```
  if (timer_create(CLOCK_MONOTONIC,&sig,&timer) != 0) error();
  if (sigemptyset(&set) != 0) error ();
  if (sigaddset(&set, SIGRTMIN) != 0) error();
  if (timer_settime(timer,0, &required, &old) != 0) error ();

  while (1) {
    if (sigwait(&set, &signum) != 0) error();
    acción periódica
  }
}
```


Relojes de tiempo de ejecución

- Hay un reloj de tiempo de ejecución para cada hebra
  - se identifica como *CLOCK_THREAD_CPUTIME_ID*
- También hay se puede obtener el identificador con:

```
int pthread_clock_getcpuclockid (pthreadt_t threadid,
                                 clockid_t *clockid);
```

- Se pueden efectuar las mismas operaciones que con cualquier reloj:

```
clock_settime (CLOCK_THREAD_CPUTIME_ID, &timespec_value);
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &timespec_value);
```

Temporizadores de tiempo de ejecución

- Como los de tiempo real

```
timer_create  (CLOCK_THREAD_CPUTIME_ID,  &event, &cputimer);
timer_settime (&cputimer, 0, &required, &old);
```

  - Se pueden usar para detectar si una hebra sobrepasa el tiempo de cómputo especificado

Resumen

- POSIX contiene un repertorio completo de mecanismos para gestionar el tiempo de ejecución
  - relojes de tiempo real, reloj monótono
  - retardos absolutos y relativos
  - límites de tiempo en operaciones
  - temporizadores de tiempo real
  - relojes y temporizadores de tiempo de ejecución