

Lección 13: Introducción a los sistemas de tiempo real

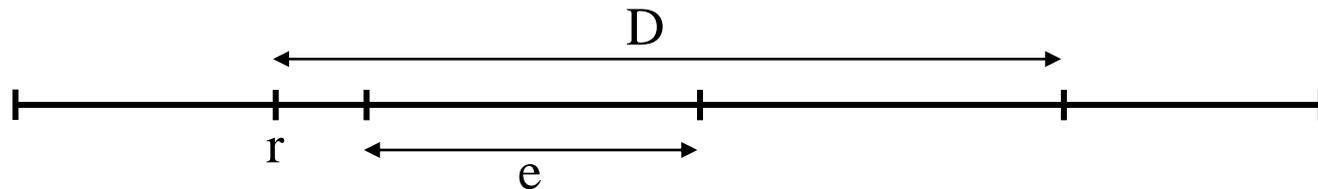
- Introducción
- Conceptos básicos
- Sistemas síncronos
- Sistemas asíncronos
- Sistemas dirigidos por interrupciones
- Algoritmos de *scheduling*

Introducción

- **Sistema reactivo:** bucle que recibe entradas de componentes hard y reacciona en consecuencia enviando salidas a componentes
- **Sistema tiempo real:** sistema reactivo con restricciones estrictas en los tiempos de respuesta
 - desde milisegundos hasta varios segundos
 - una especie de cliente-servidor
- Idea central: no se trata de hacer las cosas deprisa, sino cada cosa a tiempo

Conceptos básicos

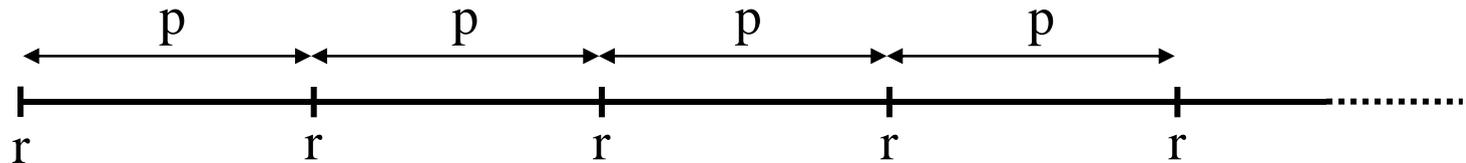
- **tarea:** proceso de un sistema TR
 - su especificación incluye requisitos temporales



- *r*: *release time*
 - *e*: *execution time*
 - *D*: *deadline* (el máximo *response time* permitido)
- una tarea puede tener un **hard** o **soft deadline**

Conceptos básicos

- **tarea periódica:** se lanza cada cierto tiempo prefijado
 - el intervalo entre los sucesivos *release times* se llama **periodo**



- existen también tareas
 - esporádicas (*hard deadlines*)
 - aperiódicas (*soft deadlines*)

Sistemas síncronos

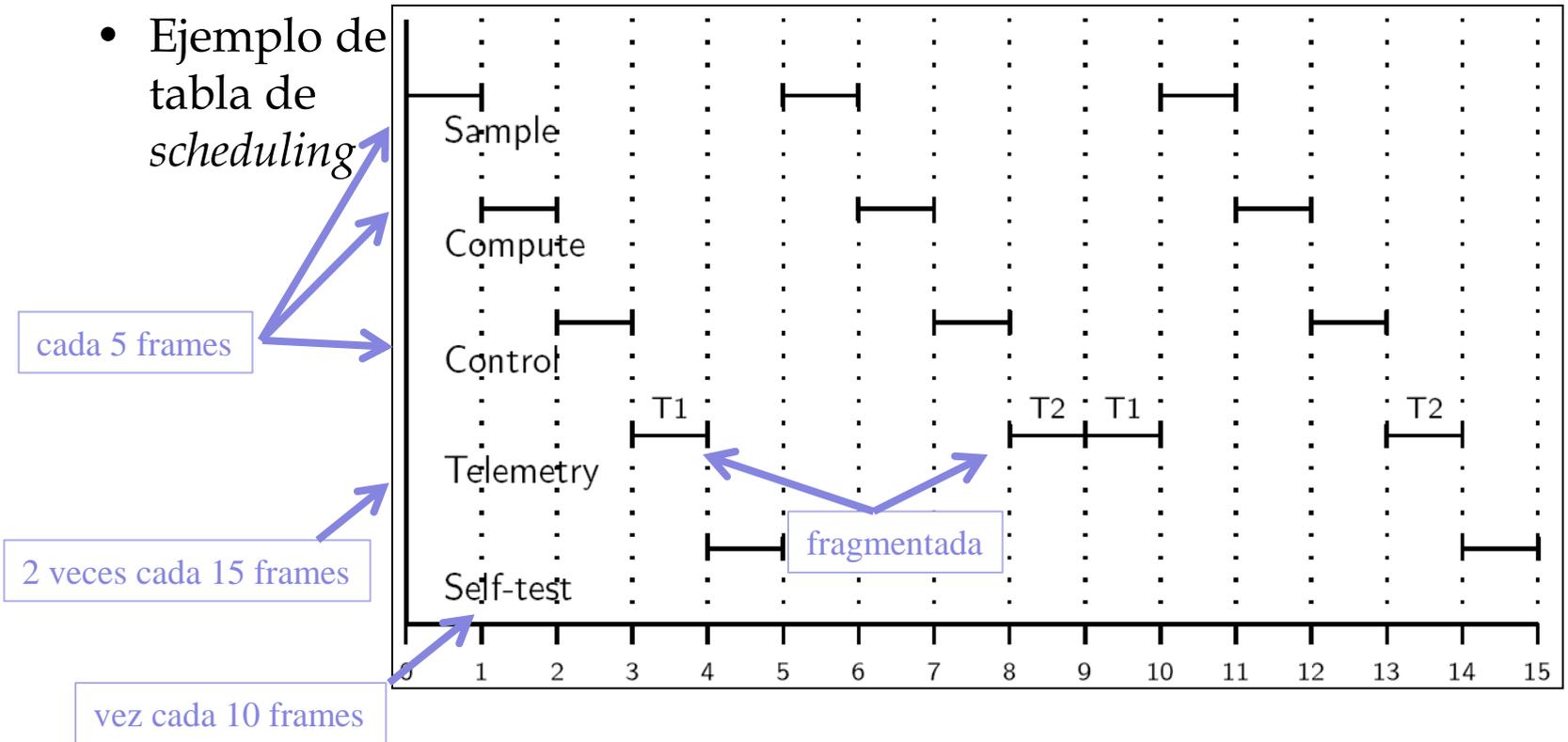
- un reloj síncrono divide el tiempo de CPU en *frames*
- el sistema se divide en tareas que duren menos que un frame
- se construye una tabla de *scheduling*
 - asigna tareas con *frames*
 - cada señal del reloj debe iniciar la ejecución de la tarea correspondiente
 - exige que tareas largas tengan que dividirse en tareas que quepan en un *frame*

Sistemas síncronos

- Supongamos un sistema con las siguientes tareas
- **Sample, Compute, Control**
 - requieren 1 frame, una vez cada 5 frames
- **Telemetry**
 - requiere 2 frames, 2 veces cada 15 frames
- **Self-test**
 - requiere 1 frame, 1 vez cada 10 frames

Sistemas síncronos

- Ejemplo de tabla de *scheduling*



M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

Sistemas síncronos

```
Process synchronous_scheduler
```

```
taskAddressType array[0..#frames] tasks := [add_1,...]
```

```
int currentFrame := 0
```

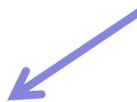
```
loop
```

```
  await beginning of frame
```

```
  invoke tasks[currentFrame]
```

```
  increment currentFrame modulo #frames
```

señal de reloj



0	1	2	3	4
Sample	Compute	Control	Telemetry 1	Self-test
5	6	7	8	9
Sample	Compute	Control	Telemetry 2	Telemetry 1
10	11	12	13	14
Sample	Compute	Control	Telemetry 2	Self-test

Sistemas síncronos

- El algoritmo es sencillo y eficiente, pero
 - es complicado dividir las tareas para que encajen en *frames*
 - subtareas requieren cambios de contexto
 - generan *overhead*
 - además, un cambio del código de una tarea puede requerir recalcular la tabla de *scheduling*
 - puede que haya ratos malperdidos de uso del procesador
 - el *frame* requiere compromiso entre *deadlines* y caso peor
- Y hay que tener presente los problemas de sincronización
 - ya que puede haber tareas divididas

Sistemas asíncronos

- Sistemas en que las tareas no se han de ejecutar en *frames*
 - las tareas se ejecutan hasta que acaban, una detrás de otra

```
Process asynchronous_scheduler  
queue of taskAddressType readyQueue  
taskAddressType currentTask  
  
loop  
  await readyQueue is not empty  
  currentTask := head of readyQueue  
  invoke currentTask
```

- eficiente
 - sencillo
 - procesador siempre ocupado
- difícil cumplir plazos
 - poco útil para TR

Sistemas asíncronos

- Se aplica un *scheduling* basado en
 - prioridades
 - *preemption* (apropiación)
 - *scheduling* events
 - llegada de nueva tarea a la cola
 - señal de reloj
 - timeslice consumido
 - ...

Sistemas asíncronos

```
Process preemptive_scheduler
```

```
queue of taskAddressType readyQueue
```

```
taskAddressType currentTask
```

```
loop
```

```
  await a scheduling event
```

```
  if currentTask.priority < highest priority in readyQueue
```

```
    save state of currentTask; insert currentTask into readyQueue
```

```
    currentTask := highest priority task from readyQueue
```

```
    invoke currentTask
```

```
  else if timeslice is past AND a task with same priority in readyQueue
```

```
    save state of currentTask; insert currentTask into readyQueue
```

```
    currentTask := a task with the same priority from readyQueue
```

```
    invoke currentTask
```

```
  else
```

```
    resume currentTask
```

Sistemas asíncronos

```
Process preemptive_scheduler
```

```
  queue of taskAddressType readyQueue
```

```
  taskAddressType currentTask
```

```
  loop
```

```
    await a scheduling event
```

```
    if currentTask.priority < highest priority in readyQueue
```

```
      save state of currentTask; insert currentTask into readyQueue
```

```
      currentTask := highest priority task from readyQueue
```

```
      invoke currentTask
```

```
    else if timeslice is past AND a task with same priority in readyQueue
```

```
      save state of currentTask; insert currentTask into readyQueue
```

```
      currentTask := a task with the same priority from readyQueue
```

```
      invoke currentTask
```

```
    else
```

```
      resume currentTask
```

Sistemas asíncronos

```
Process preemptive_scheduler
```

```
queue of taskAddressType readyQueue
```

```
taskAddressType currentTask
```

```
loop
```

```
  await a scheduling event
```

```
  if currentTask.priority < highest priority in readyQueue
```

```
    save state of currentTask; insert currentTask into readyQueue
```

```
    currentTask := highest priority task from readyQueue
```

```
    invoke currentTask
```

```
  else if timeslice is past AND a task with same priority in readyQueue
```

```
    save state of currentTask; insert currentTask into readyQueue
```

```
    currentTask := a task with the same priority from readyQueue
```

```
    invoke currentTask
```

```
  else
```

```
    resume currentTask
```

Sistemas asíncronos

```
Process preemptive_scheduler
```

```
  queue of taskAddressType readyQueue
```

```
  taskAddressType currentTask
```

```
  loop
```

```
    await a scheduling event
```

```
    if currentTask.priority < highest priority in readyQueue
```

```
      save state of currentTask; insert currentTask into readyQueue
```

```
      currentTask := highest priority task from readyQueue
```

```
      invoke currentTask
```

```
    else if timeslice is past AND a task with same priority in readyQueue
```

```
      save state of currentTask; insert currentTask into readyQueue
```

```
      currentTask := a task with the same priority from readyQueue
```

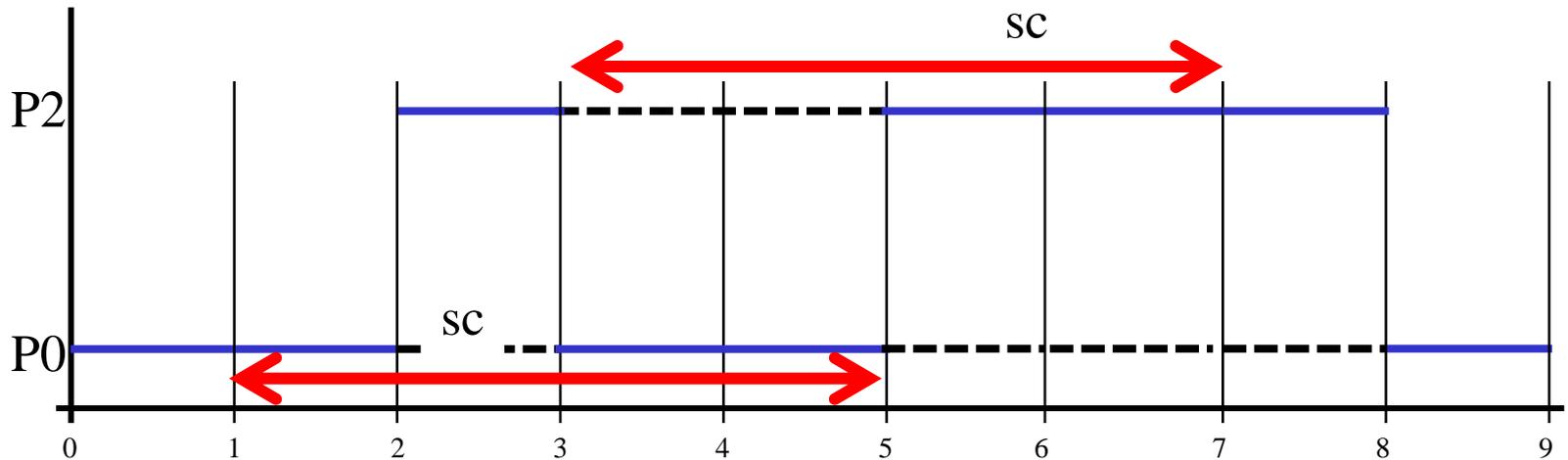
```
      invoke currentTask
```

```
  else
```

```
    resume currentTask
```

Sistemas asíncronos

- La interacción entre sincronización y *preemptive scheduling* basado en prioridades puede generar “inversión de prioridad”
 - procesos de menor prioridad “se cuelan” por la espera en sincronización

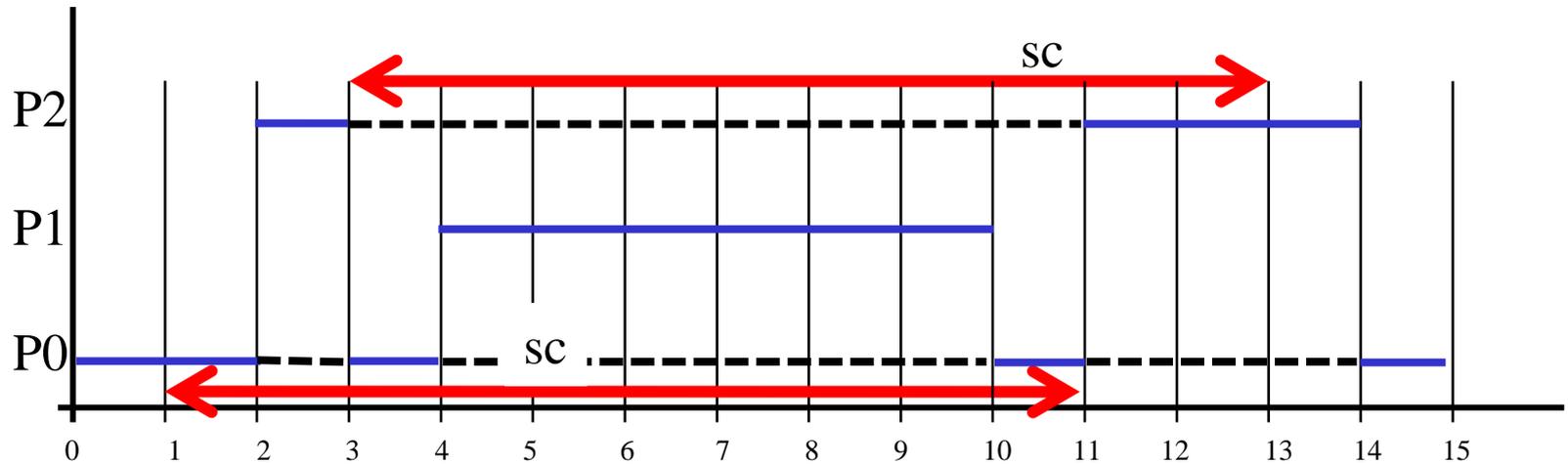


Sistemas asíncronos

- Pero este comportamiento es normal
 - cuando una tarea entra en una SC, ninguna otra puede actuar hasta que salga
 - para eso es una sección crítica
- Por eso
 - las secciones críticas deben ser tan cortas como se pueda

Sistemas asíncronos

- La inversión se produce cuando aparecen otras en discordia



- Una posible solución: la herencia de prioridad

Sistemas dirigidos por interrupciones

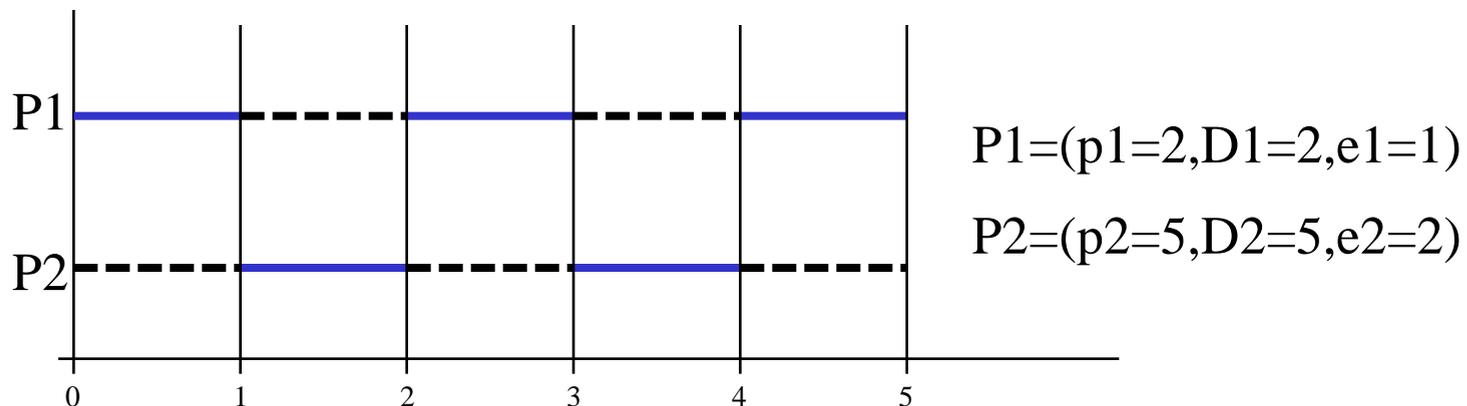
- **Interrupción:** invocación de una tarea por al *hardware*
 - *interrupt handler*
 - organización típica para sistemas empotrados
 - se puede ver el *handler* como una tarea con una prioridad mayor que cualquier tarea normal
 - no se interrumpe su ejecución: secciones críticas
 - el *hard* se puede programar para **enmascarar** interrupciones
 - alternativamente, dar prioridad entre interrupciones

Sistemas dirigidos por interrupciones

- Pero los *handlers* tienen que sincronizarse y comunicarse con el resto de tareas
 - mecanismos habituales como semáforos o monitores pueden ser muy "caros" o requerir demasiados recursos
 - uso de esquemas productor-consumidor no bloqueantes
 - ignorar datos recibidos cuando buffer lleno
 - sobreescritura de datos

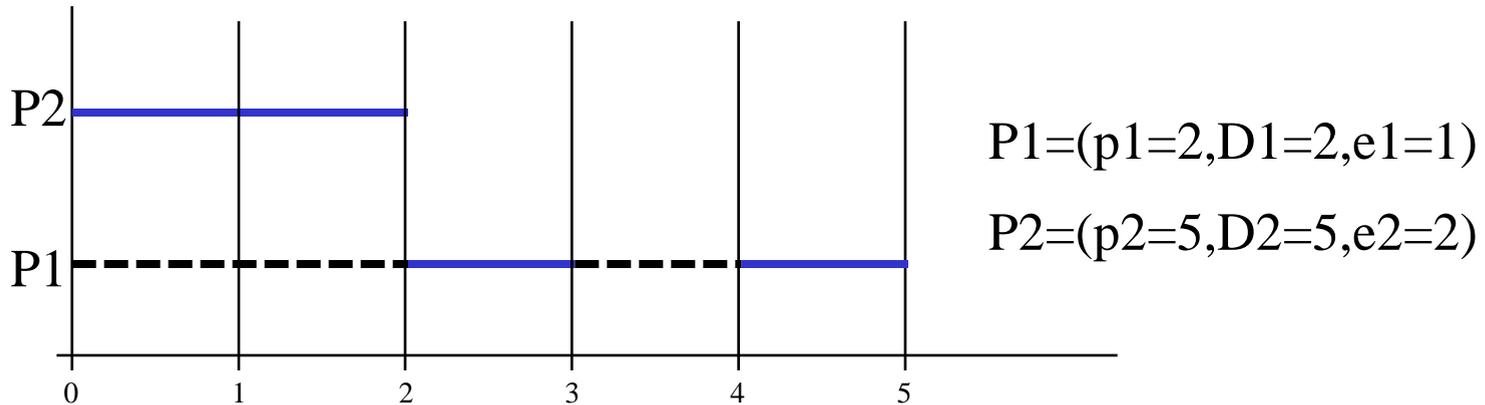
Algoritmos de *scheduling*

- Objetivo: encontrar la manera de asignar prioridades de forma que todas las tareas cumplan sus *deadlines*
 - no todas las asignaciones son viables
- Ejemplo 1: una asignación viable



Algoritmos de *scheduling*

- Ejemplo 2: una asignación no viable



Algoritmos de *scheduling*

- *Scheduling* mediante “rate monotonic algorithm” (RM)
 - asocia prioridades fijas en orden inverso al periodo requerido por la tarea
 - no considera la duración
 - si existe una asignación viable, el algoritmo la encuentra, bajo algunas restricciones
 - no recursos compartidos (no sincronización)

Algoritmos de *scheduling*

- *Scheduling* mediante “earliest deadline first algorithm” (EDF)
 - cada vez que sucede un evento de *scheduling*, se da prioridad máxima a la tarea con el *deadline* más próximo
 - si existe una asignación viable, el algoritmo encuentra también una
 - no siempre aplicable
 - interrupciones pueden tener prioridades estáticas
 - problemas de sobrecarga
 - prioridad dinámica, que requiere cálculos cada vez que hay un evento de *scheduling*