Scheduling o planificación de procesos

Sistemas Operativos
DC - FCEN - UBA

03 de abril de 2025

Menú para hoy

Menú para hoy

Repaso

Multiprogramación

- Multiprogramación: Tener procesos corriendo todo el tiempo para maximizar la utilización de CPU.
- Cuando los procesos "llegan", se inicializa su **PCB**.
- Sólo 1 proceso puede correr a la vez (1 core), los demás deben esperar.
- Muchos procesos (o sea, sus PCBs) se mantienen en memoria al mismo tiempo. Cuando la CPU se libera, se debe elegir a otro proceso de la cola de *ready* y darle la CPU $\rightarrow$ CPU scheduling.

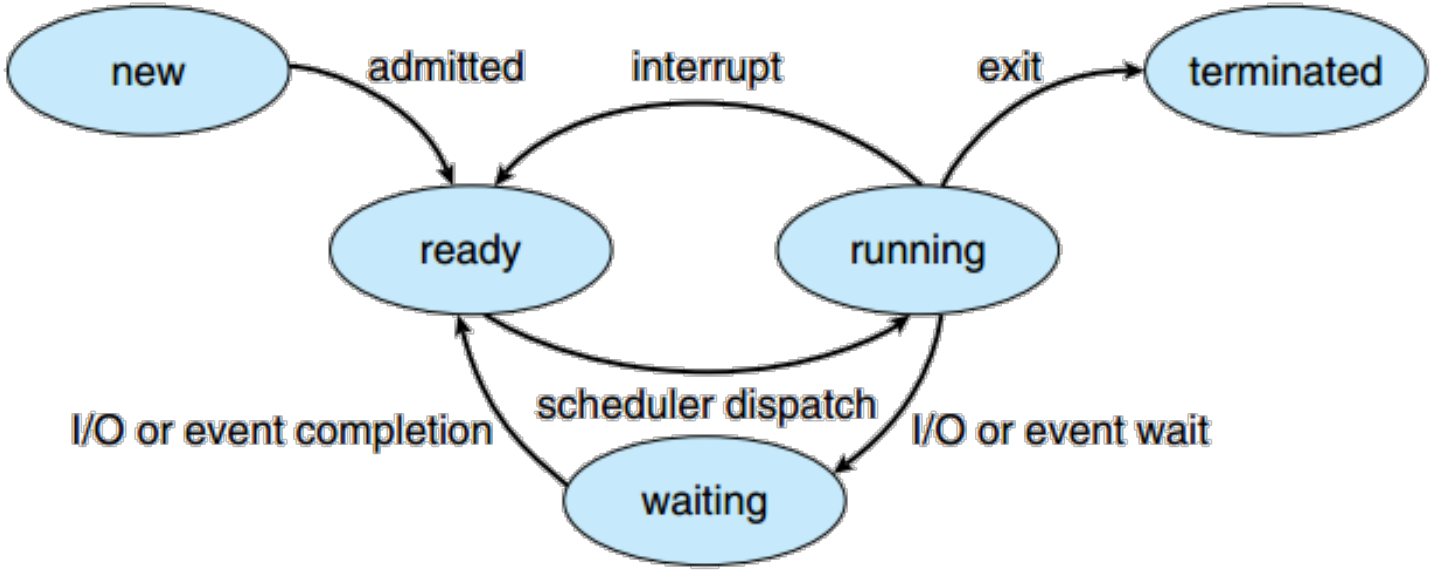


Figura: Diagrama de estados de un proceso[1]

[1] **Operating Systems Concepts**, Abraham Silberschatz & Peter B. Galvin.

Menú para hoy

Ráfagas de I/O y CPU

- La ejecución de un proceso consiste en un **ciclo**, entre ejecución de CPU y espera por I/O.
- La ejecución siempre comienza con una **ráfaga de CPU**. Luego le sigue una **ráfaga de I/O**. Luego CPU, luego I/O, luego CPU... y así hasta que termina.
- Un programa **intensivo en I/O** típicamente tiene muchas ráfagas de CPU cortitas.
- Un programa **intensivo en CPU** suele tener unas pocas ráfagas de CPU largas.

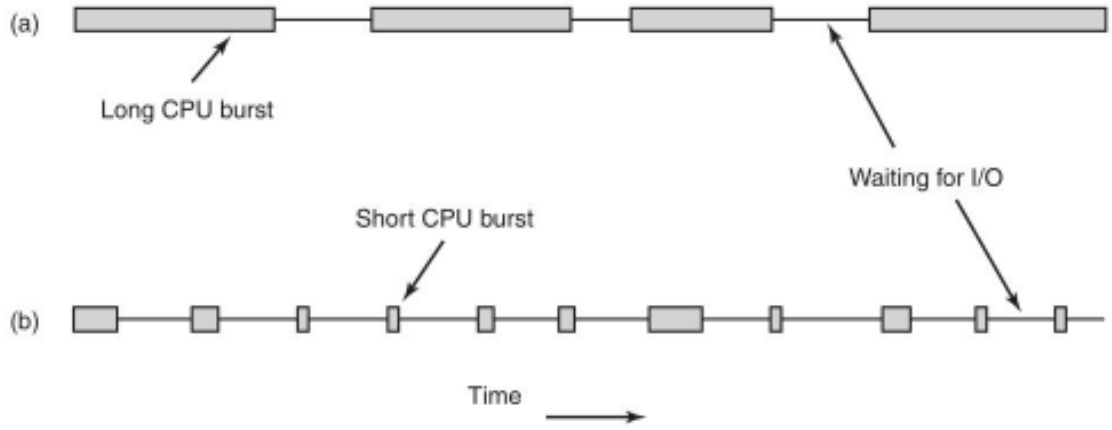


Figura: Secuencia de ráfagas de CPU y de I/O. (a) Un proceso intensivo en CPU. (b) Un proceso intensivo en I/O.[2]

[2] **Modern Operating Systems**, Andrew S. Tanenbaum.

CPU Scheduler o Planificador de CPU

- El **scheduler del Sistema Operativo** es responsable de seleccionar un proceso de todos los que están en memoria listos para ejecutar, y darle la CPU para que procese.
- *Preemptive* o *nonpreemptive*, esa es la cuestión.
  - Nonpreemptive o cooperativo: Sin desalojo. Una vez que un proceso obtiene la CPU, ejecuta hasta liberarla de forma voluntaria $\rightarrow$ si termina o si pasa a estado *waiting*.
  - Preemptive o no cooperativo: Con desalojo. El scheduler puede determinar cuándo sacarle la CPU a un proceso.

Momento para preguntas sobre los conceptos enunciados

Menú para hoy

Criterios y objetivos

Criterio	Descripción	Objetivo
Uso de CPU	Mantener la CPU tan ocupada como sea posible.	**MAXIMIZAR** $\uparrow$
Throughput	Cantidad de procesos terminados por unidad de tiempo.	**MAXIMIZAR** $\uparrow$
Turnaround	Cuánto le toma a un proceso terminar de ejecutar (tiempo esperando en *ready* + tiempo ejecutando en CPU + tiempo haciendo I/O).	**MINIMIZAR** $\downarrow$
Waiting time	Suma de los periodos en *ready*.	**MINIMIZAR** $\downarrow$
Response time	Tiempo transcurrido desde que el proceso es "lanzado" (desde que se encuentra en *idle* por primera vez) HASTA la primera vez que pasa a estar en *running* ¡No es lo mismo que turnaround!	**MINIMIZAR** $\downarrow$

Criterios y objetivos

¿Cómo sabemos que una política de scheduling es mejor que otra?

- Hay que definir los criterios para comparar.
- Para medir bien, deberíamos considerar muchos procesos con muchas ráfagas de CPU y muchas ráfagas de I/O.
- Por simplicidad, en la materia vamos a considerar sólo algunas **ráfagas de CPU** por proceso, y comparar por **waiting time promedio** y el **turnaround promedio** (salvo que se indique otra cosa).
- Si bien la mayoría de las arquitecturas de CPU actuales son *multicore*, trabajaremos en el contexto de un **único core disponible**.

Menú para hoy

First-Come, First-Served (FCFS)

Definición

- Se da el procesador al primer proceso que lo pide.
- Es *nonpreemptive.*

First-Come, First-Served (FCFS)

Ejemplo

Considerar los siguientes procesos, llegados en el instante 0 y ordenados por el scheduler tal como se muestra en la tabla y con el largo de la ráfaga de CPU indicada en milisegundos:

Proceso	Ráfaga de CPU (ms)
P1	24
P2	3
P3	3

Diagrama de Gantt:

- Waiting time promedio: $(0 + 24 + 27)/3 = 17$ milisegundos
- Turnaround promedio: $(24 + 27 + 30)/3 = 27$ milisegundos

First-Come, First-Served (FCFS)

Para pensar:

- ¿Cómo cambiarían las métricas si el orden de procesos fuese P2 - P3 - P1? → el waiting time promedio depende del orden de ejecución de los procesos considerando sus tiempos de ráfagas de CPU.
- ¿Qué pasaría si primero llegase un proceso intensivo en CPU y luego varios intensivos en I/O?→ *Convoy Effect*:

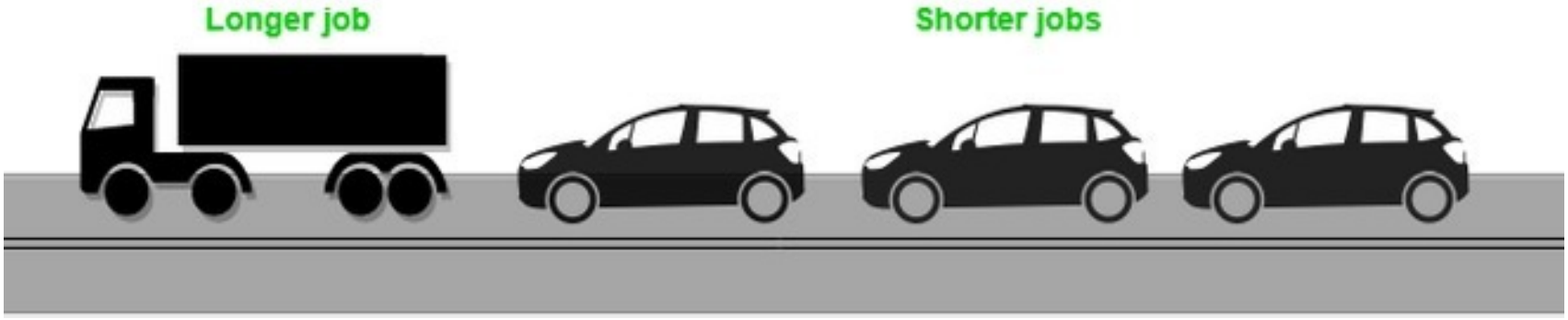


Figura: Convoy effect gráficamente

- ¿Qué pasaría si usásemos FCFS en un sistema interactivo?

Round-Robin (RR)

Definición

- El scheduler va siguiendo la cola *ready* en orden de llegada de cada proceso, dándole la CPU a cada proceso por un *quantum* de tiempo.
- Si el proceso que se está ejecutando consume su ráfaga de CPU antes de terminar su *quantum* (es decir, que ingresa en una ráfaga de I/O o el proceso termina por completo), deja la CPU voluntariamente y el scheduler le asigna la CPU al siguiente proceso.
- Si el proceso no termina su ráfaga de CPU al terminar el *quantum*, el scheduler lo desaloja y lo pone al final de la cola *ready*.

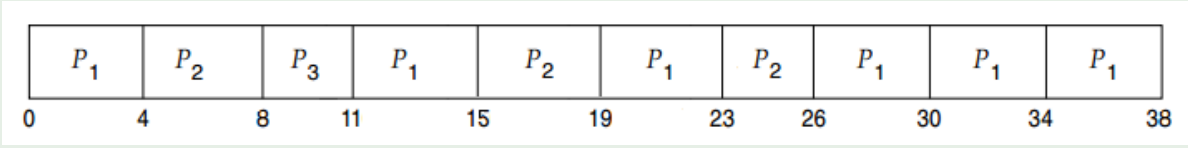
Round-Robin (RR)

Ejemplo

Considerar los siguientes procesos, llegados en el instante 0 y ordenados tal como se muestra la tabla y con el largo de la ráfaga de CPU indicada en milisegundos. Considerar un *quantum* de 4 milisegundos:

Proceso	Ráfaga de CPU
P1	24
P2	11
P3	3

Diagrama de Gantt:



- Waiting time promedio: $(14 + 15 + 8)/3 = 12,33$ milisegundos
- Turnaround promedio: $(38 + 26 + 11)/3 = 25$ milisegundos

Round-Robin (RR)

Para pensar:

- ¿Qué pasa si el *quantum* dura mucho tiempo? $\rightarrow$ Se parece a FCFS.
- ¿Qué pasa si el *quantum* dura muy poco? $\rightarrow$ Se va mucho tiempo en *context switch*.
- ¿Qué tan largo debe ser el *quantum* para tener buena *performance*? $\rightarrow$ Lo suficientemente grande como para sacar ventaja respecto al tiempo de *context switch*.

Shortest-Job-First (SJF)

Definición

- Se asocia a cada proceso el largo de su próxima ráfaga de CPU y se elige para ejecutar al proceso que tenga la menor.

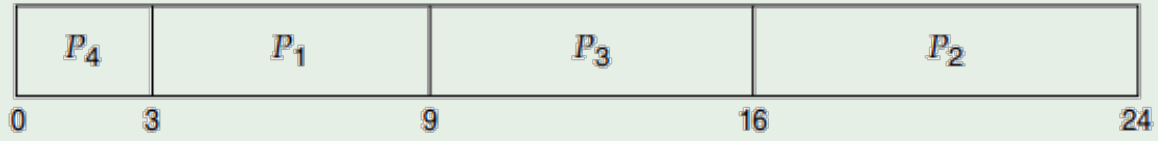
Shortest-Job-First (SJF)

Ejemplo

Considerar los siguientes procesos, con el largo de la ráfaga de CPU indicada en milisegundos:

Proceso	Ráfaga de CPU
P1	6
P2	8
P3	7
P4	3

Diagrama de Gantt:



- Waiting time promedio: $(3 + 16 + 9 + 0)/4 = 7$ milisegundos
- Turnaround promedio: $(9 + 24 + 16 + 3)/4 = 13$ milisegundos

Shortest-Job-First (SJF)

Para pensar

- ¿Qué pasa si llega un proceso cuya próxima ráfaga de CPU es menor a lo que le falta al proceso ejecutando en ese momento? → Depende.
  - Versión *nonpreemptive*: el proceso en ejecución continúa hasta que termina su ráfaga de CPU.
  - Versión *preemptive*: el scheduler desaloja al proceso en ejecución y le da la CPU al nuevo (shortest-remaining-time-first o SRTF).
- SJF es óptimo respecto al waiting time promedio. ¿Por qué no se implementa en la práctica? → no se puede saber de antemano la longitud de la próxima ráfaga de CPU. Se puede aproximar, pero igual es difícil.

Prioridades

Definición

- Se asocia a cada proceso una prioridad, y se elige para ejecutar al proceso más prioritario en estado *ready*.
- Si hay varios procesos con la misma prioridad, se ejecutan en orden FCFS.
- SJF es un caso particular de esquema por prioridades (la prioridad p es proporcional al tiempo de la siguiente ráfaga de CPU).
- Puede ser *preemptive* o *nonpreemptive*.
- ¿Una mayor prioridad se denota con un número más chico o uno más grande? Depende de la documentación de se lea. En la práctica vamos a considerar el valor 0 como la mayor prioridad.

Prioridades

Ejemplo

Considerar los siguientes procesos, con el largo de la ráfaga de CPU indicada en milisegundos:

Proceso	Ráfaga de CPU	Prioridad
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Diagrama de Gantt:

- Waiting time promedio: $(6 + 0 + 16 + 18 + 1)/5 = 8{,}2$ milisegundos
- Turnaround promedio: $(16 + 1 + 18 + 19 + 6)/5 = 12$ milisegundos

Prioridades

Para pensar

- ¿Qué pasa si constantemente llegan procesos de alta prioridad? $\rightarrow$ *Starvation*.
- ¿Hay formas de prevenir esto? $\rightarrow$ Sí:
  - **Aging:** gradualmente incrementar la prioridad de los procesos que están esperando hace mucho.
- ¿Cuánto cuesta buscar en la cola *ready* el proceso con mayor prioridad? (asumiendo que la cola esta modelada con una lista) $\rightarrow O(n)$.

Multilevel Queue

Definición

- Se mantienen colas separadas para cada prioridad.
- Se ejecutan primero los procesos en la cola de mayor prioridad.
- Cada cola tiene prioridad absoluta sobre las colas de menor prioridad.
- La prioridad de cada proceso es estática, así que cada proceso vive siempre en la misma cola.
- Se suele usar RR dentro de cada cola, aunque puede variar.
- Se suele aplicar para particionar los procesos que requieren distinto *response-time*.

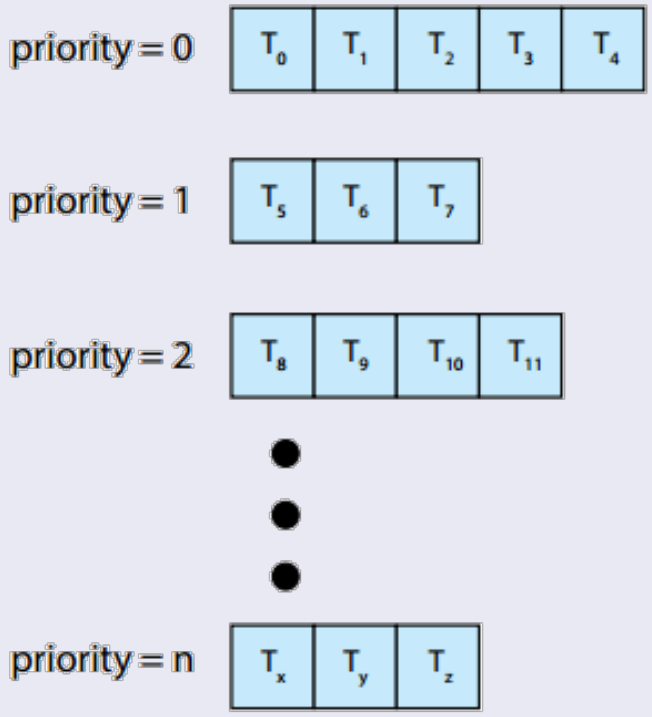


Figura: Colas separadas para cada prioridad[3]

[3] **Operating Systems Concepts**, Abraham Silberschatz & Peter B. Galvin.

Multilevel Queue

Para pensar

Una forma de particionar procesos

1. **Real-time:** procesos con restricciones de tiempo fijas y bien definidas, deben devolver el resultado correcto dentro de sus limitaciones de tiempo (ej: sistemas de imágenes médicas, sistemas de control industrial, etc.).
2. **Procesos interactivos:** programas de propósito general que se ejecutan tomando *input* por parte del usuario, quien espera una respuesta rápida (ej: procesadores de texto, planillas de cálculo, etc.).
3. **Procesos batch (lotes):** tareas periódicas y repetitivas con *input* predeterminado desde archivos u otra fuente de datos, generalmente de gran volumen (ej: copias de seguridad, recopilación de datos, facturaciones, generación de informes, etc.).

- ¿Qué tipo de procesos tendrán más prioridad? ¿Cuáles tendrán menos?
- ¿Qué pasa si constantemente llegan procesos a la cola más prioritaria? $\rightarrow$ *Starvation*.

Multilevel Feedback Queue

Definición

- Se permite a los procesos cambiarse de cola.
- Suele separar a los procesos según sus ráfagas de CPU: a mayor uso de CPU, más baja la prioridad.
- Puede implementar *aging* para evitar *starvation*.

Para pensar: ¿Qué tipos de procesos terminan quedando con mayor prioridad? $\rightarrow$ intensivos en I/O e interactivos (generalmente tienen ráfagas de CPU más cortas).

Multilevel Feedback Queue

Tener en cuenta al diseñar un *Multilevel Feedback Queue Scheduling*:

- Cantidad de colas.
- Algoritmo de scheduling en cada cola.
- Por qué un proceso podría pasar a una cola de mayor prioridad.
- Por qué un proceso podría pasar a una cola de menor prioridad.
- Cómo elijo a qué cola va un proceso cuando recién llega.
- ¿Puede haber *starvation*? ¿Por qué? ¿Puedo/quiero evitarlo? ¿Cómo?

Momento para preguntas

Menú para hoy

Ejercicio de diseño

Enunciado

Un sistema que atiende tareas interactivas de varias sucursales bancarias está conectado en forma directa a la central policial, y frente a un caso de robo, genera un proceso que activa la alarma central.

Diseñar un algoritmo de scheduling que permita que, una vez generado ese proceso de alarma, tenga prioridad sobre el resto de las tareas. Tener en cuenta que pueden generarse distintas alarmas desde distintas sucursales.

Ejercicio de diseño

Analicemos el enunciado

- Hay tareas interactivas.→RR
  - Tienden a ser intensivas en I/O (ráfagas cortas de CPU).
  - Deben tener buen tiempo de respuesta.
  - Son potencialmente muchas.
- Eventualmente se lanzan alarmas.→Priority Queue
  - Deben priorizarse sobre las otras tareas.→Prioridades
  - Pueden suceder varias alarmas en simultáneo.→Separar alarmas de otros procesos

Ejercicio de diseño

Posible solución

- Proponemos un scheduler que funcione con dos colas de prioridades estáticas.
- La cola más prioritaria se asignará a los procesos de alarma.
  - Esta cola utilizará un esquema FCFS: atenderá las alarmas en orden de llegada en caso de haber más de una.
  - Contamos con que este tipo de proceso es corto: enciende la alarma y termina.
- La cola menos prioritaria se asignará a los procesos interactivos.
  - Esta cola utilizará un esquema RR: se asignará un quantum razonable a cada proceso.
  - Además, al ser intensivos en I/O, se bloquearán seguido, permitiendo ejecutar otros procesos en la misma cola.

Ejercicio de diseño

Posible solución

- Posibilidades de *starvation*:
  - Entre los procesos interactivos no puede haber *starvation*: todos ejecutarán un quantum a su tiempo.
  - Entre los procesos de alarma no puede haber *starvation*: aunque lleguen constantemente alarmas, se resolverán rápido y permitirán pasar a las siguientes.
  - Entre colas podría existir *starvation* en caso de llegar constantes alarmas.
    - Si llegan múltiples alarmas, están robando en muchas sucursales, y pocos usuarios estarán utilizando el sistema interactivo. **No me preocupo por salvar estos procesos**.

Ejercicio de diseño

Para pensar

¿Cómo cambia el diseño si...

- ...hay que garantizar el funcionamiento de los procesos interactivos aunque haya múltiples alarmas?
- ...también hay procesos nocturnos que realizan backups?
- ...los procesos interactivos pueden eventualmente solicitar reportes que demandan mayor procesamiento?

Menú para hoy

Scheduling de Tiempo Real

Definición

- El scheduling de CPU para sistemas operativos de tiempo real tiene sus propios problemas.
- Podemos clasificarlos en:
  - Sistemas soft real-time
    - No dan garantías sobre cuándo se va a poner a ejecutar un proceso RT.
    - Sólo garantizan que los procesos van a ejecutar con más prioridad que los procesos no críticos.
  - Sistemas hard real-time
    - Tienen requerimientos más estrictos: una tarea crítica debe ser ejecutada dentro de un *deadline*.
    - Ejecutar con *deadline* vencido es lo mismo que no haber ejecutado.

Scheduling de Tiempo Real

Minimizar la latencia

- Un sistema RT típicamente está esperando que ocurra un evento RT.
- Los eventos pueden surgir desde el software (ej: un timer) o desde el hardware (ej: señal desde un sensor).
- Cuando ocurre un evento, el sistema debe responder lo más rápido posible.

El tiempo que pasa entre que ocurre un evento hasta que el proceso RT se ejecuta es llamado event latency.

- Algunos sistemas pueden tener requerimientos de latencia de pocos milisegundos, como el sistema que controla los frenos de un auto.
- Otros sistemas pueden tolerar varios segundos de latencia, como el sistema de control de un radar.

Prioridades

- La característica más importante de sistemas operativos RT es que tienen que poder responder a un proceso RT que requiere la CPU tan pronto se pueda.
- Por lo tanto, un sistema RT debe soportar un algoritmo de scheduling basado en prioridades con preemption.
- Notar que esto sólo garantiza el funcionamiento de un sistema soft RT. Hard RT requiere más garantías.

Earliest-Deadline-First (EDF)

- EDF asigna prioridades dinámicamente según el *deadline*. Cuanto más pronto el *deadline*, más prioridad.
- Cuando un proceso puede ejecutar, debe anunciar sus requerimientos de *deadline* al sistema. Las prioridades deben ajustarse para reflejar el *deadline* del nuevo proceso.
- En teoría, EDF es óptimo ya que cada proceso puede cumplir su *deadline* y se maximiza el uso de CPU. En la práctica, no siempre es factible.

Scheduling de Tiempo Real

Para pensar

- En el ejemplo dado, ¿qué tipo de proceso es la activación de alarma?
- ¿Cambia en algo la solución propuesta?

Menú para hoy

Resumen

Hoy vimos...

- Describimos varios algoritmos de scheduling de CPU.
  - First-Come, First-Served
  - Round-Robin
  - Shortest-Job-First
  - Prioridades
  - Multilevel Queue
  - Multilevel Feedback Queue
- Comparamos los algoritmos basándonos en criterios específicos de scheduling.
  - Throughput
  - Turnaround
  - Waiting time
  - Response time
- Introducimos sistemas de tiempo real
  - Soft vs Hard
  - Prioridades vs EDF

Cómo seguimos...

Con esto se puede resolver toda la guía práctica 2.