Scheduling

Gisela Confalonieri

Departamento de Computación - FCEyN

11 de abril de 2023

Estructura de la clase

Introducción

- Multiprogramación: Tener procesos corriendo todo el tiempo para maximizar la utilización de CPU.
- Sólo 1 proceso puede correr a la vez (1 core), los demás deben esperar.
- Muchos procesos 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**.

Repaso: Estados de un proceso

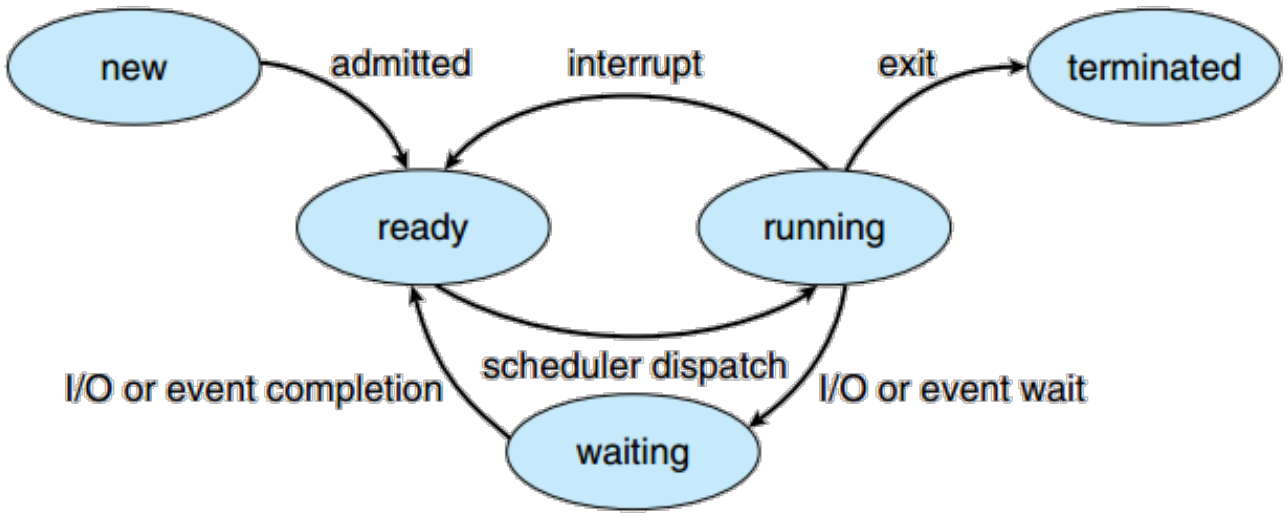


Figura: Diagrama de estados de un proceso

Fue como una ráfaga tu amor

- La ejecución de un proceso consiste en un **ciclo** de ejecución de CPU y espera por I/O.
- Un programa **intensivo en I/O** típicamente tiene ráfagas de CPU cortitas. Un programa **intensivo en CPU** suele tener unas pocas ráfagas de CPU largas.

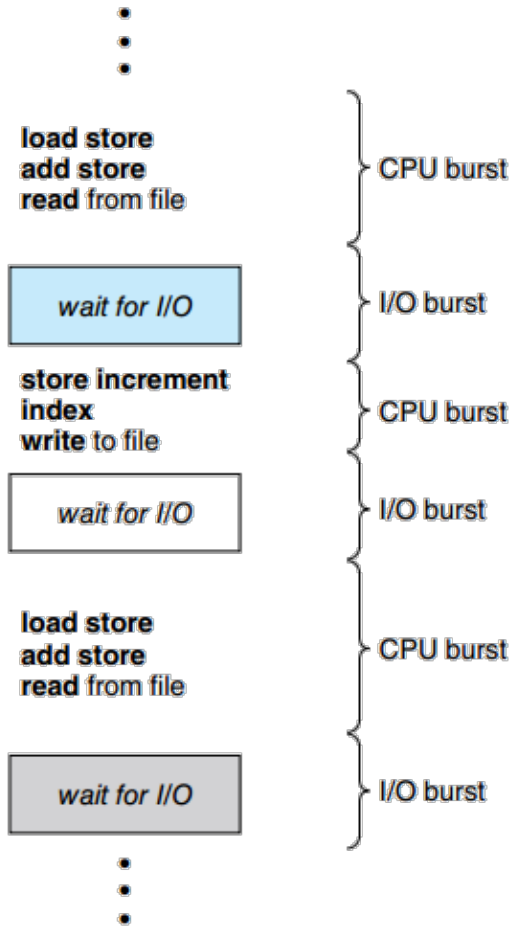


Figura: Secuencia de ráfagas de CPU y de I/O

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:** Sin desalojo. Cuando un proceso está usando la CPU, debe liberarla voluntariamente → cuando termina o se bloquea esperando I/O.
  - **Preemptive:** Con desalojo. El scheduler puede determinar cuándo un proceso debe liberar la CPU.

Criterios y objetivos

- **Utilización de CPU:** queremos mantener la CPU tan ocupada como sea posible. **MAXIMIZAR** ↑
- **Throughput:** la cantidad de procesos terminados por unidad de tiempo. **MAXIMIZAR** ↑
- **Turnaround:** cuánto le toma a un proceso terminar de ejecutar → tiempo esperando en *ready* + tiempo ejecutando en CPU + tiempo haciendo I/O. **MINIMIZAR** ↓
- **Waiting time:** la suma de los periodos en *ready*. **MINIMIZAR** ↓
- **Response time:** el tiempo que tarda en empezar a responder. ¡No es lo mismo que turnaround! **MINIMIZAR** ↓

¿Qué vamos a usar?

¿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** (salvo que se indique otra cosa).

First-Come, First-Served (FCFS)

Se le da el procesador al primer proceso que llega.

Ejemplo. Considerar los siguientes procesos, llegados en ese orden y con el largo de la ráfaga de CPU indicada en milisegundos:

Proceso	Ráfaga de CPU
P1	24
P2	3
P3	3

Diagrama de Gantt:

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

First-Come, First-Served (FCFS)

Para pensar:

- ¿Cómo cambiaría la métrica si el orden de procesos fuese P2 - P3 - P1? → el waiting time promedio va a depender mucho de la variación de las ráfagas de CPU de los procesos.
- ¿Qué pasaría si primero llegase un proceso intensivo en CPU y luego varios intensivos en I/O?→ *Convoy Effect*.
- ¿Qué pasaría si usásemos FCFS en un sistema interactivo?

Round-Robin (RR)

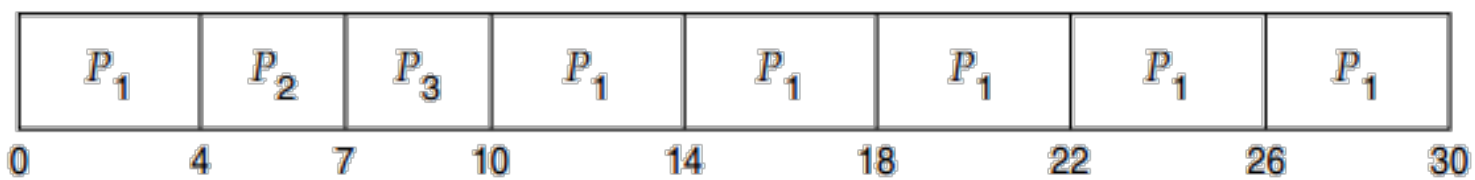
- El scheduler va siguiendo la cola *ready* en orden, dándole el CPU a cada proceso por un **quantum** de tiempo.
- Si el proceso ejecutando termina su ráfaga de CPU antes del *quantum*, se va del CPU voluntariamente y el scheduler asigna al siguiente proceso.
- Si el proceso sigue ejecutando al terminar el *quantum*, la CPU lo desaloja y lo pone al final de la cola *ready*.

Round-Robin (RR)

Ejemplo. Considerar los siguientes procesos, llegados en ese orden y con el largo de la ráfaga de CPU indicada en milisegundos, y considerar un *quantum* de 4 milisegundos:

Proceso	Ráfaga de CPU
P1	24
P2	3
P3	3

Diagrama de Gantt:



Waiting time promedio: $(6 + 4 + 7)/3 = 5{,}66$ milisegundos

Round-Robin (RR)

Para pensar:

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

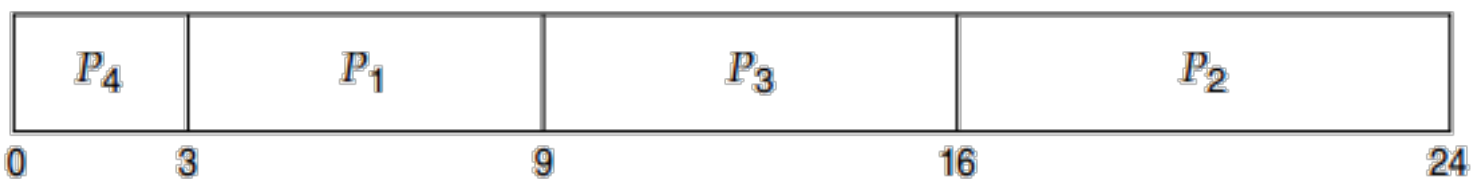
Shortest-Job-First (SJF)

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

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

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. Si el scheduler es *nonpreemptive*, el proceso en ejecución continúa hasta que termina su ráfaga de CPU. Si es *preemptive*, el scheduler lo desaloja (Shortest Remaining Job First).
- 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

- Se asocia a cada proceso una prioridad, y se elige para ejecutar al proceso más prioritario.
- SJF es un caso particular de esquema por prioridades (la prioridad p es el inverso 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 el libro. 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

Prioridades

Para pensar:

- ¿Qué pasa si constantemente llegan procesos de alta prioridad? → *Starvation*.
- ¿Hay formas de prevenir esto? → Sí:
  - **Aging:** gradualmente incrementar la prioridad de los procesos que están esperando hace mucho.
  - Combinar con RR: correr procesos con misma prioridad en RR.
- ¿Cuánto cuesta buscar en la cola *ready* el proceso con mayor prioridad? → $O(n)$.

Multilevel Queue

- 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.
- Se suele usar RR dentro de cada cola.

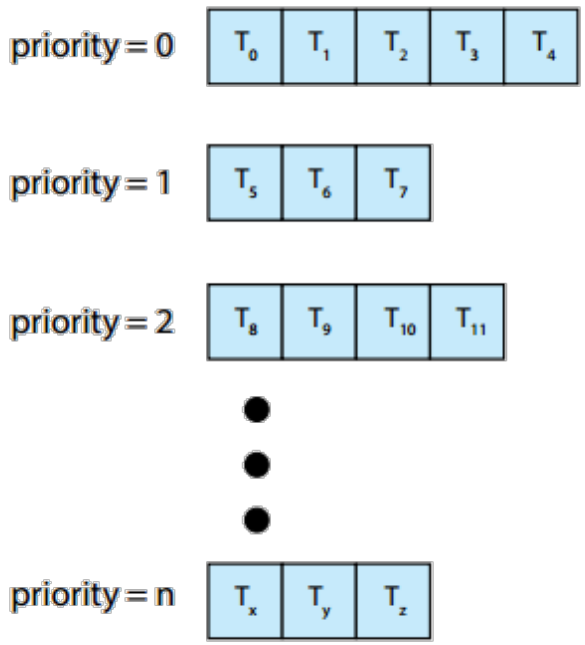


Figura: Colas separadas para cada prioridad

Multilevel Queue

Para pensar:

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

Multilevel Feedback Queue

- Se permite a los procesos cambiarse de cola.
- Se separa a los procesos según sus ráfagas de CPU: a mayor uso de CPU, menor la prioridad.
- *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.

Ejercicio de parcial (1C2022)

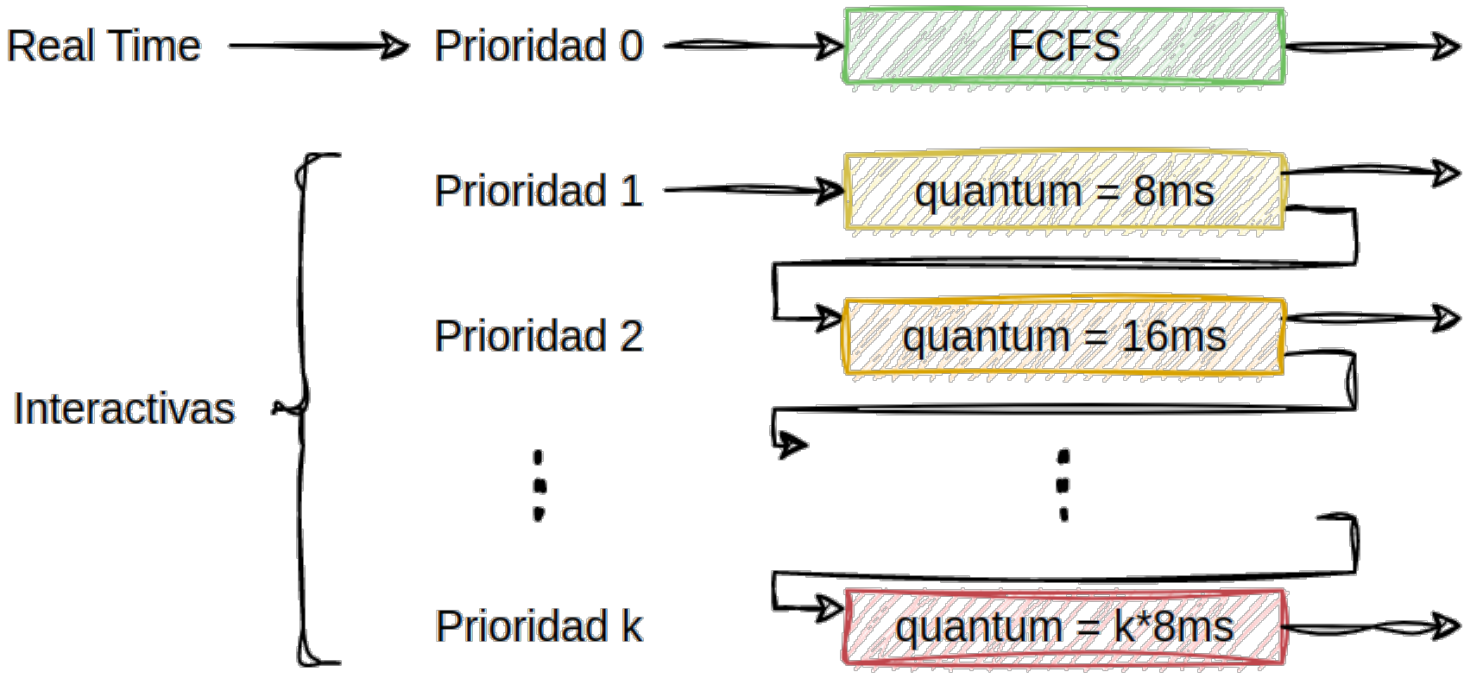
Se busca diseñar un *scheduler* para un sistema operativo que tiene que soportar, en principio, dos tipos de tareas: *real time* e interactivas. Las tareas *real time* tendrán un único nivel de prioridad estático, y se debe proveer una cota para el peor caso del *response time* de estas tareas. Por otro lado, las tareas interactivas podrán pertenecer a una de k clases de prioridades. En este sentido, el sistema debería siempre favorecer a tareas con comportamiento intensivo en E/S, y se espera que se pueda adaptar al cambio en el comportamiento de las tareas: si una tarea intensiva en CPU comienza a volverse intensiva en E/S, habría que recompensarla; análogamente, habría que penalizarla si se diera el caso opuesto.

Ejercicio de parcial (1C2022)

a) Proponer una estrategia de *scheduling* adecuada para el escenario propuesto. Es importante dejar en claro qué estructuras de datos del *kernel* se van a necesitar, cómo se actualizarán y justificar cómo el algoritmo propuesto busca cumplir los objetivos de diseño.

b) ¿Podría darse una situación donde exista *starvation* para las tareas interactivas? ¿Y para las *real time*? Justificar.

Ejercicio de parcial (1C2022) Posible solución



- Este diseño utilizará *k+1* colas.
- En la cola más prioritaria se atenderán sólo los procesos RT con dinámica FCFS, ya que se espera que tengan breves ráfagas de CPU. Así, no habrá *starvation* de tareas RT.
- El *response time* estará acotado por $P \cdot t$, siendo P la cantidad procesos RT en estado *ready* y t el tiempo máximo de uso de CPU de estos procesos.

Ejercicio de parcial (1C2022) Posible solución

- En las siguientes k colas se atenderán los procesos interactivos con una dinámica RR.
- Todos comienzan con prioridad 1. Si al terminar su *quantum* un proceso aún no terminó su ráfaga de CPU, se ubica al final de la cola de prioridad siguiente. Si un proceso es desalojado por haber llegado otro de mayor prioridad, se vuelve a ubicar al final de la misma cola.
- Si un proceso se bloquea esperando I/O, al volver a estado *ready* se ubica al final de la siguiente cola de mayor prioridad.
- Cada cierto tiempo (1 segundo), los procesos que no estuvieron ejecutando se trasladan a la siguiente cola de mayor prioridad. Así, se evita *starvation* en caso de que lleguen constantemente procesos más prioritarios.

Resumen de la clase:

- 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

Con esto se puede resolver toda la guía práctica 2.

Nota: Todas las figuras de estas slides pertenecen al libro **Operating Systems Concepts**, Abraham Silberschatz & Peter B. Galvin.


SO ASK ME
QUESTIONS
quickmeme.com