S.O. Tema 2.2: Planificación de procesos

1. Conceptos básicos

1.1. Axiomas de trabajo

-Nuestro **modelo** de **ejecución** asume que los **programas intercalan** tiempos de **computación** en **CPU** (bursts o ráfagas) con otros en los que usan los periféricos (entrada/salida, I/O).

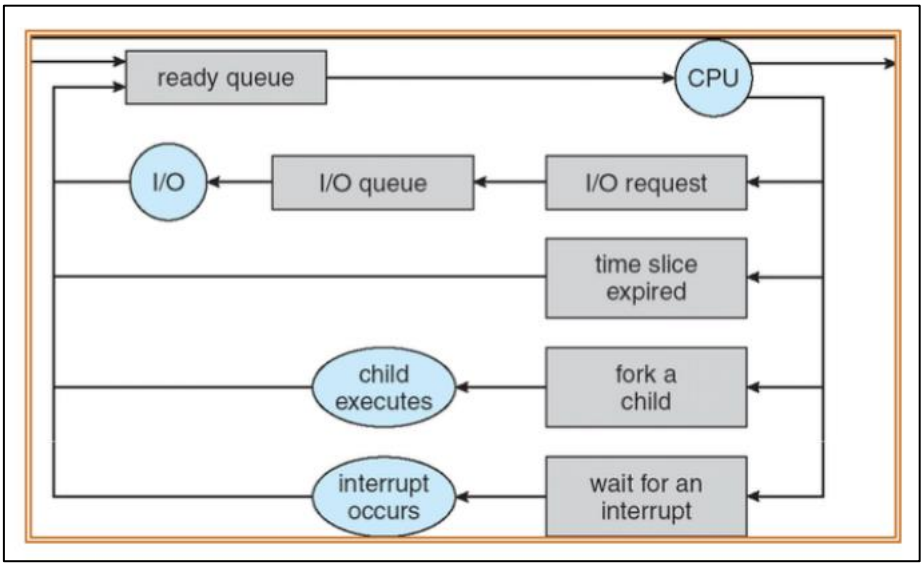
- Los **tiempos** de **CPU** son más **frecuentes**, pero también más cortos.
- Si **predomina** el tiempo de **CPU**, el proceso se califica como **CPUbound** (limitado por la CPU). En el polo opuesto, tenemos **I/O-bound**.

-El **planificador** (scheduler) del S.O. se encarga de ocupar la CPU por **quantums** de **tiempo fijos** para todos los **procesos** al **margen** de su **intensidad** computacional.

-Si un **proceso** necesita una **ráfaga** de **CPU mayor** que el **quantum** que le asigna el S.O., **abandonará la CPU** y reanudará su uso en el siguiente quantum, así hasta terminar.

1.2. Matices de la planificación

-El S.O. implementa un **algoritmo** para **elegir** el **siguiente proceso** que ocupa la **CPU** (y cada recurso compartido), de entre todos los que se encuentran en la cola de listos (Ready).



1.3. Tipos de planificadores

-**A corto plazo (short-term)**: Elige el siguiente proceso que ocupa la CPU.

- Se **invoca** con **mucha frecuencia** (cada pocos milisegundos) y debe ser **ágil para que el rendimiento** no decaiga.

-**A medio plazo**: Elige los procesos que se quedan en memoria y los que se alojan en disco.

- **Determina** la **presión** que se ejerce sobre la **memoria** por parte del grado de **multiprogramación**, tratando de evitar el riesgo de thrashing.

-**A largo plazo (long-term)**: Selecciona los procesos que se meten en la cola de listos (Ready).

- Se invoca con **poca frecuencia** (cada pocos segundos), y debe **controlar**, además, el **grado** de **multiprogramación** vigente en cada momento (máximo número de procesos activos).

1.4. Métricas de planificación

-**Utilización de la CPU**: Maximizar el % que está ocupada.

-**Throughput** (ritmo de ejecución): Maximizar el número de procesos que completan su ejecución por unidad de tiempo.

-**Tiempo de compleción (Tr):** Minimizar el que engloba los 4 tiempos siguientes:

- La espera para alojarse en memoria.
- La espera en la cola de listos (Ready).
- El tiempo en CPU.
- El tiempo de E/S.

-**Tiempo de espera (Te)**: Sólo la espera en la cola de listos.

-**Tiempo de respuesta (Ta)**: Lo que tarda en responderse una petición. También se denomina latencia.

1.5. Política de planificación

-Es la **estrategia** que sigue el **algoritmo de planificación** para optimizar la **métrica** que **mejor refleje** los **objetivos**.

-En **sistemas por lotes (batch**):

- % CPU.
- Tiempo de compleción (Tr).

-En **sistemas de tiempo compartido**:

- Varianza del tiempo de respuesta.
- Proporción entre la complejidad y el tiempo de respuesta.

-En **sistemas de tiempo real:**

- No superar tiempos críticos.
- Predecibilidad o consistencia.

2. Algoritmos de planificación

2.1. Planificador de la CPU

-Elige el **proceso** que va a ocupar la **CPU** de entre los que se **encuentran** en la **cola** de **listos**. Actúa cuando un proceso:

- **Conmuta del estado Running al estado Waiting**.
- **Conmuta del estado Waiting al estado Ready**.
- **Conmuta del estado Running al estado Ready.**
- **Finaliza**.

-Un **planificador** con **desalojo** o **expropiativo** (preemptive) permite a un proceso **desalojar** de la **CPU** de forma inmediata al que la está ocupando (por ejemplo, por ser más prioritario).

-El **planificador** sin **desalojo** (non-preemptive) esperará siempre a que salga el proceso que ocupa la CPU para relevarle.

2.2. Síntesis de los algoritmos de planificación que estudiaremos

No desalojan la CPU (no expropiativos, *non-preemptive*)	Desalojan la CPU (expropiativos, *preemptive*)	Métrica a la que conceden más importancia
FCFS	Tiempo compartido *(round-robin)*	Instante en que cada proceso llega al sistema
SJF	SRTF	Tiempo medio de compleción
Prioridad	Prioridad con desalojo	Respetar la prioridad

2.3. FCFS (First Come First Served)

-Los **procesos** se **ejecutan** en el **orden** de **llegada**, es decir, como nos atienden en la **cola** de un **comercio**.

-**Ventaja**:

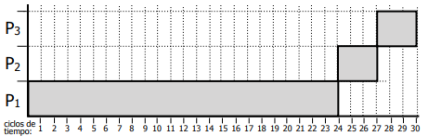
- Sencillo de implementar.

-**Inconvenientes**:

- Le cuesta maximizar el porcentaje de uso de la CPU y los dispositivos.
- Le cuesta minimizar los tiempos de compleción (Tr) y espera (Te).
- No es buena opción en los sistemas de tiempo compartido, en los que cada usuario necesita mantener una cuota de uso de la CPU a intervalos regulares.
- Adolece del convoy effect: Los procesos cortos eternizan su llegada a la CPU por encontrarse ésta monopolizada por un proceso largo.

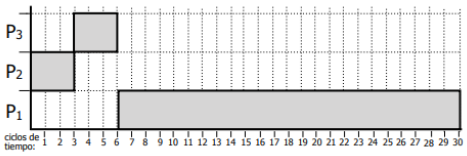
-A continuación vemos un ejemplo en el que tres procesos llegan en orden P1, P2 y P3:

- Supongamos un sistema en el que los procesos llegan en el orden P_1, P_2 y P_3, requiriendo 24, 3 y 3 ciclos de CPU, respectivamente. Usaremos diagramas de Gantt para ilustrar la evolución temporal de los procesos. En este caso:



- El cálculo de métricas de rendimiento es el siguiente:
  - Tiempos de espera (T_e): $P_1 = 0$; $P_2 = 24$; $P_3 = 27$. Media: 17.
  - Tiempos de compleción (T_r): $P_1 = 24$; $P_2 = 27$; $P_3 = 30$. Media: 27.
- Se produce el *convoy-effect*: Dos procesos cortos esperan demasiado por estar precedidos de otro mucho más largo.

- Si los procesos llegan en el orden P_2, P_3 y P_1, el diagrama de Gantt cambia por completo:



- Y las métricas revelan un resultado mucho mejor, al desaparecer el *convoy effect*:
  - T. de espera (T_e): $P_1 = 6$; $P_2 = 0$; $P_3 = 3$. Media: 3. (antes 17)
  - T. de compleción (T_r): $P_1 = 30$; $P_2 = 3$; $P_3 = 6$. Media: 13 (antes 27).

2.4. Tiempo compartido

-**Reparte** el **tiempo** de la **CPU** en **intervalos** o **quantums** de unos **5 msg**. que **asigna** de forma **rotatoria** entre los **procesos** que **compiten** por el uso de la **CPU** en cada momento.

-El **rendimiento depende** del **quantum**:

- Si es **grande**, converge al algoritmo FCFS y es menos justo.
- Si es **pequeño**, penaliza más la sobrecarga del cambio de contexto.
- **Mejor pequeño** si hay muchos procesos y grande si hay pocos.

-Al **repartir** el **tiempo democráticamente** entre todos los procesos, los más cortos salen ganando en el reparto.

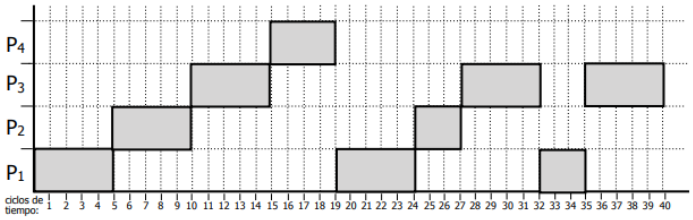
-Es el **algoritmo** que **más** se **aplica** en los **sistemas reales**, no sólo en los S.O.

-En el **siguiente ejemplo** consideramos **4 procesos** y un **sistema** con un **quantum** de 5:

- Sea un *quantum q=5* ciclos y los siguientes procesos:

Proceso	Ráfaga de CPU (en ciclos)
P_1	13
P_2	8
P_3	15
P_4	4

- Diagrama de Gantt:



- Tiempos de espera (T_e):
  - $P_1 = (19-5)+(32-24) = 22$
  - $P_2 = (24-10)+(5-0) = 19$
  - $P_3 = 10+12+3 = 25$
  - $P_4 = 15$
  - T_e medio: $82/4 = 20.25$
- $T_r = (35+27+40+19)/4 = 30.25$

2.5. ¿Qué pasaría si conociéramos el futuro?

-**Asignaríamos** la **CPU** a los **procesos** que les **queda menos** para **acabar**, **minimizando** el **tiempo medio** de **compleción**.

-En **caso** de **desconocer** este **dato futuro**, podemos **estimarlo** desde el **pasado**:

- Si el **proceso** viene siendo **intensivo** en **CPU**, lo seguirá siendo.
- Se **aproxima** así la **próxima ráfaga** de **CPU** de un proceso en función de sus últimas ráfagas, bien calculando la media de las N últimas ráfagas, o mejor aún, dando más importancia a las ráfagas más recientes en una media ponderada.

-Esa **idea** la aplican los **algoritmos SJF** y **SRTF**, que usan esta próxima ráfaga, ya sea conocida o estimada, para decidir el proceso que ocupa la CPU en cada momento.

2.6. Shortest Job First (SJF)

-**Asocia a cada proceso** la **longitud** de su **próxima ráfaga** de **CPU**, ocupando la CPU el que tenga la ráfaga más corta.

-Es un **algoritmo sin desalojo**, es decir, el proceso que solicita el uso de la **CPU esperará** a que **deje** de **usarla** el que la tiene **asignada** en ese **momento**.

-**Ventaja**:

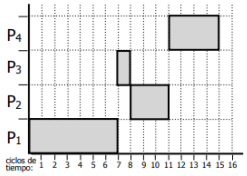
- Reduce el tiempo medio de compleción.

-**Inconveniente**:

- Perjudica a los procesos intensivos en el uso de la CPU, que apenas avanzan en abundancia de procesos cortos.

Proceso	Instante de llegada a la cola de procesos Ready	Ráfaga de CPU
P_1	Ciclo 0	7 ciclos
P_2	Ciclo 2	3 ciclos
P_3	Ciclo 4	1 ciclos
P_4	Ciclo 5	4 ciclos

- Diagrama de Gantt:



- $T_{e\,=}$ [0 + (8-2) + (7-4) + (11-5)] / 4 = 3.75 (tiempo de espera)
- T_r = [(7-0)+(11-2)+(8-4)+(15-5)] / 4 = 7.5 (tiempo de compleción)

2.7. Shortest Remaining Time First (SRTF)

-Es la **versión** con **desalojo** (**preemptive**) del **algoritmo SJF**: En cuanto llegue un proceso con ráfaga de CPU más corta que el que tiene la CPU, se apropiará de ella.

-**Ventaja**:

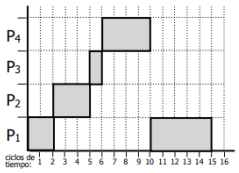
- Minimiza el tiempo medio de respuesta.

-**Inconveniente**:

- Al ser más agresivo, acentúa el problema del SJF: En abundancia de procesos cortos, los largos apenas progresan.

Proceso	Ciclo de llegada	Ráfaga de CPU (en ciclos)
P_1	0	7
P_2	2	3
P_3	4	1
P_4	5	4

- Diagrama de Gantt:



- $T_{e\,=}$ [(10-2) + 0 + (5-4) + (6-5)] / 4 = 2.5 (tiempo de espera)
- T_r = [(15-0)+(5-2)+(6-4)+(10-5)] / 4 = 6.25 (t. de compleción)

2.8. Planificación con prioridad

-**Asignamos** a cada **proceso** un **número entero positivo**, **más bajo** a **mayor prioridad**, para otorgar la **CPU** al proceso más **prioritario** en cada **momento**:

- **Algoritmo sin desalojo**: La cola Ready se ordena según prioridades.

- **Algoritmo con desalojo**: Cuando un proceso llega a la cola Ready, compara su prioridad con el proceso en ejecución y ocupa la CPU si tiene una prioridad mayor.

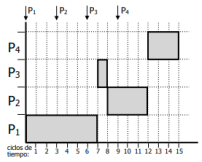
-**SJF** y **SRTF** son ejemplos **sin/con desalojo** si se toma el **tiempo** de la **próxima ráfaga** como **número** de **prioridad**.

-**Riesgo**: Relegar demasiado a procesos de baja prioridad.

-**Solución**: Implementar un mecanismo de "envejecimiento" (aging) que suba la prioridad con el tiempo de espera.

Proceso	Ciclo de llegada	Ráfaga de CPU	Prioridad
P_1	0	7	3
P_2	3	4	2
P_3	6	1	1
P_4	9	3	4

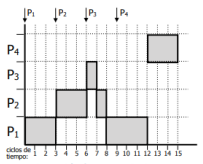
D. de Gantt sin desalojo:



$T_e = [0 + (8-3) + (7-6) + (12-9)] / 4 = 2.25$

$T_r = [(7-0) + (12-3) + (8-6) + (15-9)] / 4 = 6$

D. de Gantt con desalojo:



$T_e = (5 + 1 + 0 + 3) / 4 = 2.25$

$T_r = [(12-0) + (8-3) + (7-6) + (15-9)] / 4 = 6$

2.9. Planificación por colas multinivel

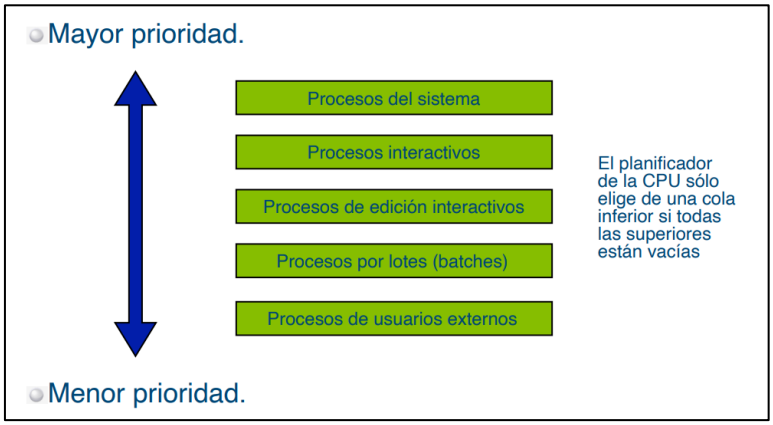
-**Consiste** en **escindir** la **cola** de **procesos Ready** en varias de **distinta prioridad**, y **establecer** un **algoritmo distinto** en cada cola. Ejemplo:

- **Procesos interactivos** en una **cola** en **primer plano** (foreground), que asigne la CPU mediante round-robin.
- **Procesos batch** en otra **cola menos prioritaria** (background), que seleccione los procesos mediante FCFS.

-Para **prevenir** la **espera indefinida** en los **procesos menos prioritarios**, podemos **establecer**:

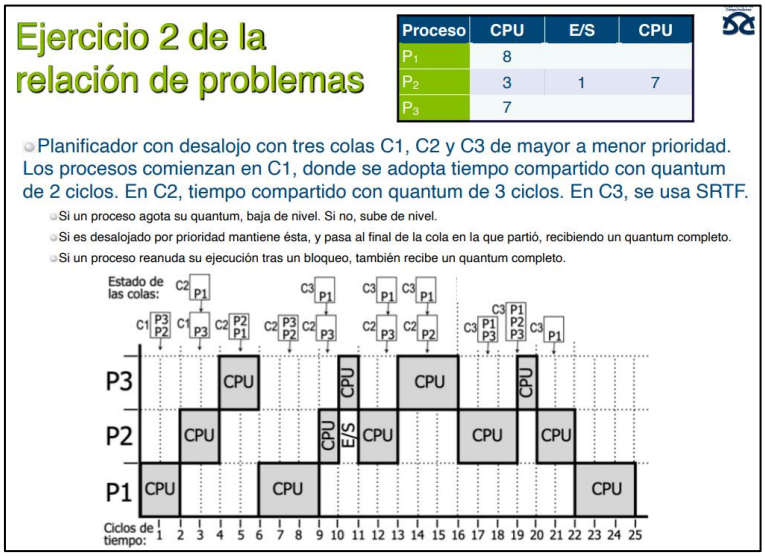
- **Reglas** para que un **proceso migre** de una **cola** a **otra** (como aging).
- Un **porcentaje** de **uso** de la **CPU** en cada **cola** (por ejemplo, 80% para los procesos interactivos y 20% para los procesos batch).

-Ejemplo de sistema con 5 colas:



-Ejemplo de sistemas con 3 colas multinivel:

-Ejercicio 2 de la relación de problemas:



2.10. Planificador para múltiples CPUs

-**Resulta** más **complejo** de **administrar**.

-Si **todas** las **CPUs** son **gemelas** (SMP, Symmetric Multi-Processor), que es lo habitual, las colas **Ready** admiten dos implementaciones:

- Todos los **procesos** en una **cola común** (como en las cajas de Carrefour).
- Cada **CPU** tiene su **cola propia** (como en las cajas de Mercadona).

-Si las **CPUs** son **distintas**, la más **potente** suele **ocuparse** de **mantener** las **estructuras** de **datos** del **sistema**, aliviando la **necesidad** de **compartir información** entre todas ellas.

-Cada **proceso** debe tener **afinidad** por la **CPU** en la que **comienza** a **ejecutarse**, reanudando su **ejecución** en ella, aunque se pueden establecer **mecanismos** de **migración**:

- Las **CPUs** más **sobrecargadas** pueden **delegar procesos**.
- Las **CPUs** más **aliviadas** pueden **tomar procesos** de las **anteriores**.

3. Ejemplos

3.1. El planificador de Linux

-**Selecciona** tanto **procesos** como **hilos** (tasks).

-Es un **planificador** con **desalojo** y **prioridades**.

-Nuestro punto de partida será el **kernel 2.6**, que **adopta** un **planificador** constante, O(1), con dos colas multinivel:

- **Procesos en tiempo real**: sin realimentación, con prioridades estáticas entre 1 y 99 (que siempre prevalecen frente a las normales).
- **Procesos normales**: con realimentación y prioridades dinámicas entre 100 y 139, que tratan de favorecer a los procesos con más E/S.
  - Por defecto, los procesos comienzan con prioridad 120.
  - Se les sube la prioridad (hasta -5) según se prodiguen en operaciones de E/S.
  - Se les baja la prioridad (hasta +5) si son intensivos en CPU (agotan quantum).
- Para **cada prioridad**, la cola se escinde en dos:
  - Active/Expired, con los procesos que no/sí agotaron su quantum.
  - El siguiente proceso a ejecutar se toma de la cola Active.

3.2. Gestión de prioridades en el planificador O(1)

-Cada **proceso comienza** con la **prioridad heredada** de su **proceso padre** o una **prioridad base** que puede asignarle el usuario por medio de la llamada nice().

-Los **procesos** con **igual prioridad** (valor nice) usan tiempo compartido en su cola, donde el quantum se calcula así:

- Si la prioridad está entre 100 y 119, quantum = (140-prioridad) x 20.
- Si la prioridad está entre 120 y 139, quantum = (140-prioridad) x 5.

-Ejemplos:

	Prioridad	Valor nice	Quantum
Máxima	100	-20	800 msg.
Alta	110	-10	600 msg.
Por defecto	120	0	100 msg.
Baja	130	10	50 msg.
Mínima	139	19	5 msg.

3.3. El planificador mejorado CFS (Completely Fair Scheduler) de Linux

-Se **introduce** a partir del **núcleo 2.6.23** (Octubre de 2007).

-La idea es compartir la **CPU** de **forma proporcional** a **nice**, asignando un **%** a **cada proceso** en **lugar** de un **quantum**.

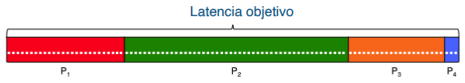
- Se define la latencia objetivo como el lapso mínimo de tiempo en el que rotarán todos los procesos (por defecto, 20 msg.).
- Si hay N procesos compitiendo por la CPU, el quantum será 20/N.
- El valor nice se usa para ponderar este quantum medio, de forma que a cada proceso se le asignará un tiempo diferente de la CPU.

Nice	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11
Weight	88761	71755	56483	46273	36291	29154	23254	18705	14949	11916
Nice	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
Weight	9548	7620	6100	4904	3906	3121	2501	1991	1586	1277
Nice	0	1	2	3	4	5	6	7	8	9
Weight	1024	820	655	526	423	335	272	215	172	137
Nice	10	11	12	13	14	15	16	17	18	19
Weight	110	87	70	56	45	36	29	23	18	15

3.4. Ejemplo del planificador CFS

	P1	P2	P3	P4
Valor nice	0	-3	2	10
Peso (weight)	1024	1991	820	110
Peso ponderado	26 %	50 %	21 %	3 %

Y la línea de tiempo del planificador que refleja el uso de la CPU de forma similar a nuestros diagramas de Gantt quedaría de la siguiente forma:



3.5. El planificador de Windows 7 y Windows 10

-Se **implementa** con **desalojo** y **prioridades**, teniendo al **hilo** como **unidad** de **planificación**.

- **Clase estática** para las **16 prioridades más altas** (procesos en tiempo real), que sólo puede lanzar el administrador.
- **Clase variable** para las **16 prioridades más bajas** (aplicaciones de usuario), que pueden subir su prioridad según su interactividad.

-Cada **prioridad** tiene su **cola**, que se **gestiona** por **tiempo compartido** con **quantum fijo**. Reglas:

- Cada vez que se usa un **quantum completo**, se baja 1 la prioridad.
- Cuando un **proceso** se **bloquea** por **E/S**, al **desbloquearse** se tiene en **cuenta** el **tiempo** de **bloqueo**: Teclado y ratón suben 6. HD sube 1.
- Si un proceso es desalojado por otro de prioridad mayor, se coloca el primero de su cola.