

Sistemas Operativos

Planificación de la CPU

¿A quién le toca correr ahora?

Gerson Garrido

Semana 3

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A · Planificación básica

1. Supuestos sobre la carga de trabajo y métricas
2. FIFO, SJF y STCF
3. Tiempo de respuesta y Round Robin
4. E/S y lo que el planificador no sabe

Sesión B · Multi-Level Feedback Queue

5. Reglas 1–4: aprender del pasado
6. Problemas: inanición y trampas
7. Boost, contabilidad y ajuste
8. Sistemas reales
9. Tiempo real: RM, EDF y Linux

Lectura

OSTEP cap. 7 *Scheduling: Introduction* y cap. 8 *Scheduling: The Multi-Level Feedback Queue*.

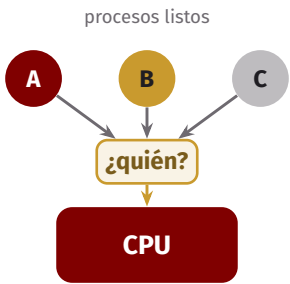
SECCIÓN 01

Supuestos y métricas

01

De mecanismo a política

- La semana pasada: el mecanismo. Interrupción del temporizador *(timer interrupt)*, trap al kernel y cambio de contexto *(context switch)*.
- Hoy: la política. Hay varios procesos listos, ¿a cuál le damos la CPU?
- Esa decisión la toma el **planificador** *(scheduler)* cada vez que el kernel recupera el control.



Cinco supuestos (poco realistas)

Para empezar, simplificamos la **carga de trabajo** *(workload)*. Cada proceso es un *trabajo (job)* y suponemos que:

1. Todos los trabajos duran lo mismo.
2. Todos llegan al mismo tiempo.
3. Una vez que empiezan, corren hasta terminar.
4. Solo usan la CPU: no hacen E/S.
5. Conocemos de antemano cuánto dura cada uno.

El plan de la sesión

Vamos a relajar los supuestos uno por uno. Cada vez que caiga uno, la política anterior falla y necesitamos una nueva. El supuesto 5 es el más irreal: el kernel no lee el futuro.

Una métrica: el tiempo de retorno

Tiempo de retorno *(turnaround time)*

$T_{\text{retorno}} = T_{\text{fin}} - T_{\text{llegada}}$: cuánto espera el usuario desde que lanza el trabajo hasta tener el resultado. Comparamos políticas por el promedio.

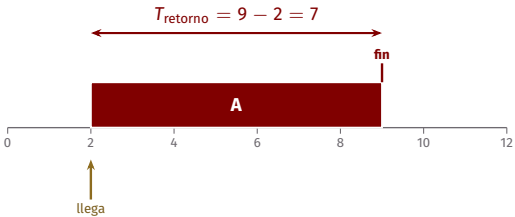


diagrama de Gantt: el tiempo avanza hacia la derecha, en ticks

Es una métrica de **rendimiento**. Otra es la **equidad** *(fairness)*; a menudo chocan.

SECCIÓN 02

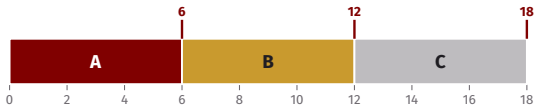
FIFO, SJF y STCF

02



FIFO: el primero que llega se atiende primero

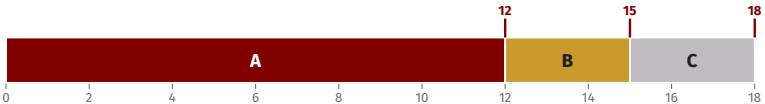
- *First In, First Out*, también FCFS.
- Simple de implementar: una cola.
- Ejemplo: A, B y C llegan en $t = 0$ (A un pelo antes) y cada uno dura **6**.



Trabajo	Llegada	Fin	Retorno
A	0	6	$6 - 0 = 6$
B	0	12	$12 - 0 = 12$
C	0	18	$18 - 0 = 18$
		Promedio:	$\frac{6+12+18}{3} = 12$

Cae el supuesto 1: el efecto convoy

Ahora A dura **12**; B y C, solo **3**. Todos llegan en $t = 0$.



B y C, cortos, atascados detrás de A

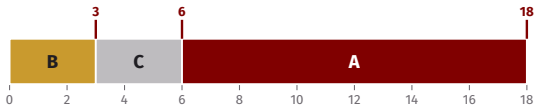
Retorno promedio: $\frac{12+15+18}{3} = 15$, aunque dos de los tres trabajos necesitan solo 3 ticks.

Efecto convoy *(convoy effect)*

Varios trabajos cortos esperan detrás de uno largo. Es la cola del supermercado cuando adelante hay alguien con dos carritos llenos y tú llevas solo una gaseosa.

SJF: primero el trabajo más corto

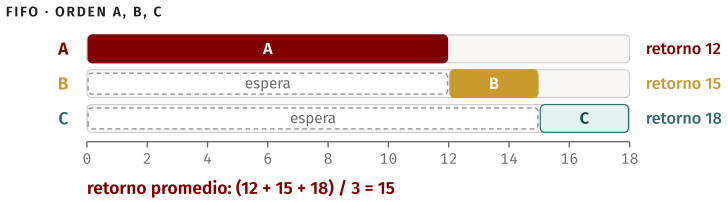
- *Shortest Job First*: ordenar por duración y correr en ese orden.
- Misma carga: A = 12, B = 3, C = 3.
- De 15 a 9: B y C ya no pagan la espera de A.



Trabajo	Llegada	Fin	Retorno
B	0	3	3
C	0	6	6
A	0	18	18

Promedio: $\frac{3+6+18}{3} = 9$

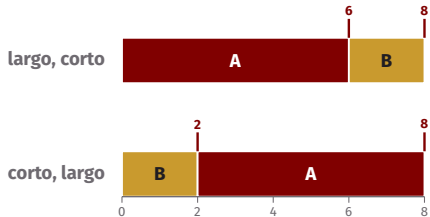
El convoy, carril por carril



A termina en 18 en ambos casos;
B y C ya no pagan su espera

El tramo punteado es la espera. Con SJF, B y C ya no esperan a A, y A termina igual en 18.

¿Por qué SJF es óptimo si todos llegan juntos?



Suma de los fines: $6 + 8 = 14$ vs. $2 + 8 = 10$.

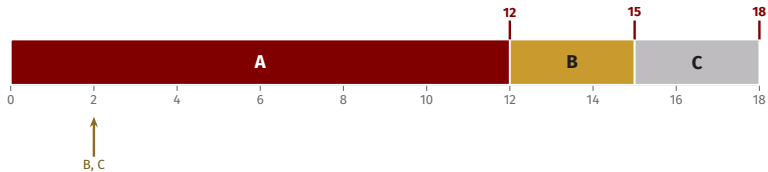
- El último termina en el mismo instante en cualquier orden.
- El que va primero retrasa a *todos* los demás: mejor que sea corto.
- Si un largo va antes que un corto, intercambiarlos baja el promedio. Al final queda el orden de SJF.

Pregunta de repaso 1

¿Cuándo dan FIFO y SJF el mismo retorno promedio? Cuando llegan ya ordenados de corto a largo, o todos duran lo mismo.

Cae el supuesto 2: llegadas en distinto momento

A (12) llega en $t = 0$; B (3) y C (3) llegan en $t = 2$. SJF **no desaloja**: cuando llegan, A ya está corriendo.



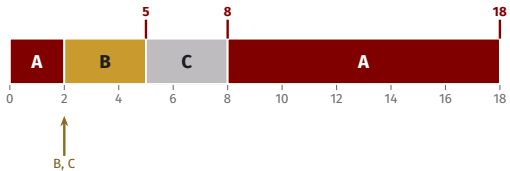
Trabajo	Llegada	Fin	Retorno
A	0	12	$12 - 0 = 12$
B	2	15	$15 - 2 = 13$
C	2	18	$18 - 2 = 16$

Promedio: $\frac{12+13+16}{3} \approx 13{,}67$. El convoy volvió.

Cae el supuesto 3: STCF desaloja

Shortest Time-to-Completion First: cada vez que llega un trabajo, correr el que tenga **menos tiempo restante**. Es SJF **apropiativo** *(preemptive)*.

- $t = 0$: solo está A. Corre A.
- $t = 2$: llegan B y C. Restan A 10, B 3, C 3. Desaloja a A; corre B.
- $t = 5$: termina B. Restan A 10, C 3. Corre C.
- $t = 8$: termina C. Solo queda A.



Trabajo	Llegada	Fin	Retorno
A	0	18	18
B	2	5	$5 - 2 = 3$
C	2	8	$8 - 2 = 6$

Promedio: $\frac{18+3+6}{3} = 9$

Verificar con un simulador

bash

```
$ ./sched stcf 12 3@2 3@2
política: STCF
          0    5    10   15
gantt  AABBBCCCAAAAAAAAAA
tramos A[0,2) B[2,5) C[5,8) A[8,18)

trabajo llegada ... fin  retorno
   A         0   ...  18       18
   B         2   ...   5        3
   C         2   ...   8        6
promedio                    9.00
```

- `code/week-03/sched.c`: unas 250 líneas de C.
- Un trabajo es `duración@llegada`.
- Políticas: `fifo`, `sjf`, `stcf`, `rr`.
- Avanza de a un tick y desempata siempre igual, por eso sus números coinciden con estas diapositivas.

Salida recortada: también imprime la duración, el inicio y la respuesta de cada trabajo.

SECCIÓN 03

Round Robin y E/S

03



Una nueva métrica: el tiempo de respuesta

Tiempo de respuesta *(response time)*

$$T_{\text{respuesta}} = T_{\text{primera ejecución}} - T_{\text{llegada}}$$

Cuánto tarda el sistema en *empezar* a atender un trabajo: lo que siente quien usa una terminal.

STCF del ejemplo anterior:
A: $0 - 0 = 0$ B: $2 - 2 = 0$ C: $5 - 2 = 3$
Promedio: **1**. Bien…

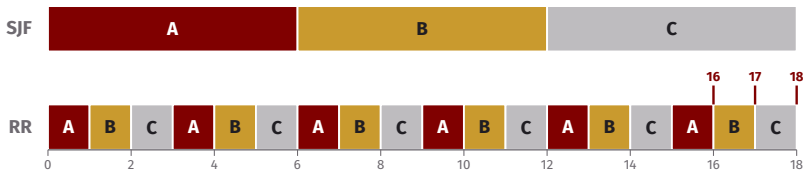
SJF con A, B y C de 6 ticks:
A: 0 B: 6 C: 12
Promedio: 6. C espera 12 ticks.

SJF y STCF optimizan el retorno, no la respuesta

El tercero de la fila no corre un solo tick hasta que los dos primeros terminan.

Round Robin: un turno corto para cada uno

Correr cada trabajo durante un **quantum** *(time slice)* y pasar al siguiente de la cola. El quantum es un múltiplo del periodo del temporizador. A, B, C de 6 ticks, quantum = 1:



	Respuesta	Retorno
SJF	$\frac{0+6+12}{3} = 6$	$\frac{6+12+18}{3} = 12$
RR, quantum 1	$\frac{0+1+2}{3} = 1$	$\frac{16+17+18}{3} = 17$

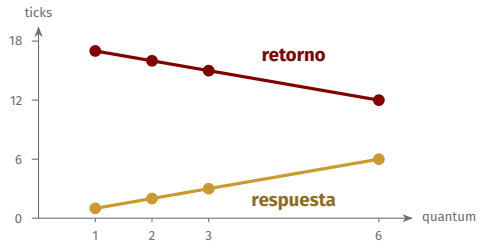
RR estira a todos: cada uno termina casi al final. Para el retorno, RR es de lo peor.

El quantum decide el compromiso

A, B, C de 6 ticks, todos en $t = 0$:

Quantum	Respuesta	Retorno
1	1	17
2	2	16
3	3	15
6	6	12

Con quantum 6, RR es igual a FIFO.

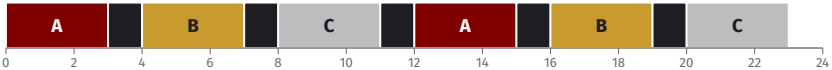


Pregunta de repaso 2

Quantum corto: buena respuesta, mal retorno, más cambios de contexto. Largo: lo contrario. Ser justo en escalas cortas se paga en retorno.

El cambio de contexto no es gratis

Mismo ejemplo con quantum 3, y cada cambio de contexto cuesta 1 tick (en negro):



Quantum	Ticks perdidos	Total	CPU perdida
1	17	35	49 %
3	5	23	22 %
6	2	20	10 %

Amortizar

Un costo fijo pesa menos si se paga menos veces: quantum $\gg$ cambio de contexto. Y el costo real incluye cachés, TLB y predictor de saltos que se enfrían.

En xv6: Round Robin sobre la tabla de procesos

kernel/proc.c · scheduler()

```
for (;;) {
  // ... (interrupciones)
  for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->state == RUNNABLE) {
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;  // ...
    }
    release(&p->lock);
  }
  // ... si nadie estaba listo: wfi
}
```

- Recorre `proc[]` en orden y corre el primer `RUNNABLE` que encuentra.
- `swtch` salta al proceso. Vuelve aquí cuando el proceso cede la CPU.
- Luego sigue con el *siguiente* de la tabla: eso es Round Robin.
- Sin prioridades ni historia. En P2 lo reemplazarán por un planificador por **lotería**.

En xv6: el quantum lo pone el temporizador

kernel/trap.c

```
// usertrap(): ceder la CPU si
// fue el temporizador
if (which_dev == 2)
  yield();
// clockintr(): pedir la próxima
// interrupción; 1000000 es una
// décima de segundo
w_stimecmp(r_time() + 1000000);
```

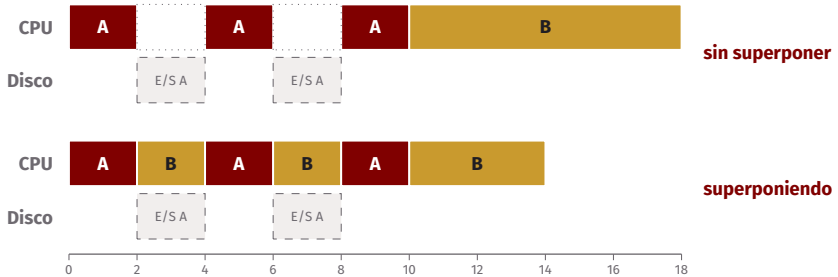
kernel/proc.c

```
void yield(void) {
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();  // a scheduler()
  release(&p->lock);
}
```

- Cada interrupción del temporizador termina en `yield()`: el proceso pasa a `RUNNABLE` y `sched()` vuelve al bucle del planificador.
- Quantum de xv6: un tick, unos 100 ms.
- Es la semana 2 vista como política: el mecanismo (trap y `swtch`) ya lo conocían.

Cae el supuesto 4: superponer E/S y CPU

A usa 2 ticks de CPU, espera 2 de disco, y así tres veces. B solo usa CPU: 8 ticks.



- Truco: cada **ráfaga** de CPU de A es un trabajo independiente de 2 ticks.
- Con STCF, mientras A espera el disco corre B; al volver, A (2) desaloja a B (resto 6).
- La CPU nunca queda ociosa: B termina en 14 en vez de 18.

Cae el supuesto 5: no conocemos el futuro

Supuesto	Al relajarlo…	Respuesta
1. Misma duración	efecto convoy	SJF
2. Llegan juntos	SJF no desaloja	STCF
3. Corren hasta el fin	—	desalojo con el temporizador
Nueva métrica	respuesta mala con SJF/STCF	Round Robin
4. Sin E/S	CPU ociosa	ráfagas como trabajos
5. Duración conocida	¡SJF y STCF son imposibles!	?

La pregunta de la sesión B

¿Buen retorno (como SJF) *y* buena respuesta (como RR) sin saber cuánto dura cada trabajo? Idea: observar el pasado para predecir el futuro.

SECCIÓN 04

MLFQ: reglas básicas

04



Multi-Level Feedback Queue

- Corbató, 1962 (CTSS). Premio Turing en parte por esto.
- Varias colas, cada una con una **prioridad** distinta.
- **Regla 1**: si Prioridad(A) $>$ Prioridad(B), corre A y B no.
- **Regla 2**: si Prioridad(A) $=$ Prioridad(B), A y B se turnan en Round Robin.
- La clave: la prioridad *cambia* según cómo se comporta cada trabajo.



Fernando Corbató. Jason Dorfman, CC BY-SA 3.0, Wikimedia Commons.



¿Cómo cambia la prioridad?

Asignación *(allotment)*

Cuánta CPU puede usar un trabajo en un nivel antes de bajar. Por ahora, igual a un quantum.

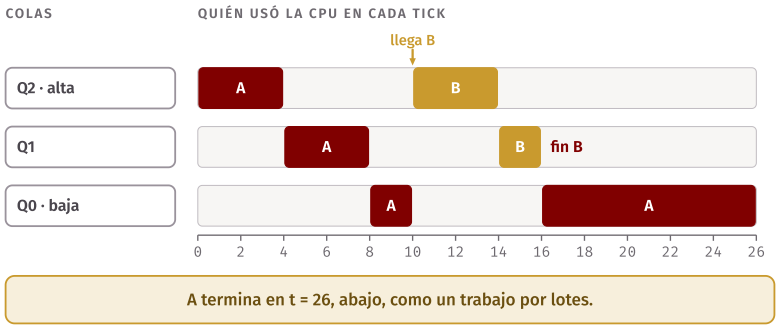
- **Regla 3**: un trabajo nuevo entra en la cola más alta.
- **Regla 4a**: si usa toda su asignación, baja un nivel.
- **Regla 4b**: si suelta la CPU antes (por ejemplo, para hacer E/S), se queda en su nivel y su cuenta vuelve a cero.

La apuesta

No sabemos si un trabajo es corto o largo: lo tratamos como corto. Si lo es, termina rápido arriba; si es largo, baja y corre como un trabajo por lotes.

Ejemplo: un trabajo largo, y luego uno corto

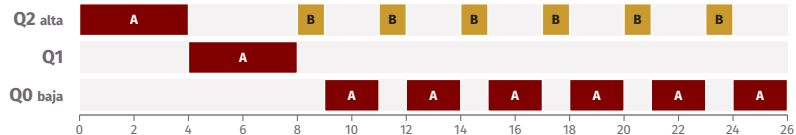
Tres colas, quantum 4. A dura 20 y llega en $t = 0$; B dura 6 y llega en $t = 10$.



- B entra en Q2 y, por la **regla 1**, desaloja a A.
- B termina en $t = 16$: retorno 6, igual a su duración. MLFQ se parece a SJF sin conocer duraciones.

Ejemplo: un trabajo interactivo

A dura 20. B llega en $t = 8$: corre **1** tick y hace E/S de **2** ticks, una y otra vez (como un editor que espera teclas).



- B suelta la CPU antes de agotar su quantum. Por la **regla 4b** se queda en Q2.
- Mientras B espera su E/S, corre A en Q0: CPU y disco superpuestos, como en la sesión A.
- B tiene tiempo de respuesta 0 cada vez que vuelve: es lo que quiere un usuario.

Generado con `./mlfq -o 20 6@8:1/2` (`code/week-03/mlfq.c`).

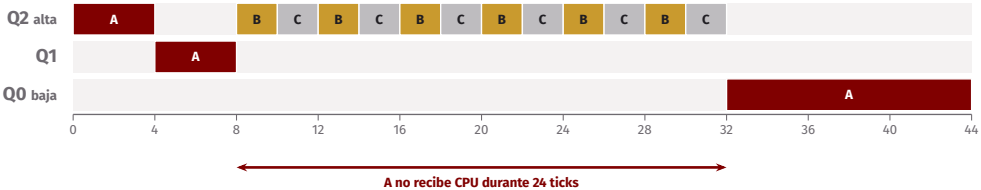
SECCIÓN 05

Problemas y cómo se corrigen

05

Problema 1: inanición

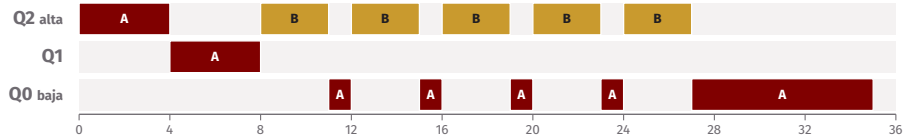
A dura 20. B y C llegan en $t = 8$; los dos son interactivos: 2 ticks de CPU, 2 de E/S.



- Entre los dos mantienen la CPU ocupada: cuando uno hace E/S, corre el otro.
- Por la regla 1, A en Q0 no corre mientras haya alguien en Q2.
- Con más trabajos interactivos, A podría no correr nunca: **inanición** *(starvation)*.

Problema 2: engañar al planificador

B hace trampa *(gaming)*: corre 3 ticks, justo antes de agotar su quantum de 4, y hace una E/S inútil de 1 tick. Por la regla 4b, nunca baja.

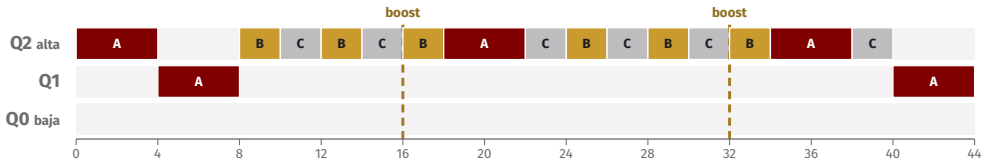


- Desde $t = 8$, B usa 3 de cada 4 ticks. Entre $t = 8$ y $t = 27$, A recibe solo 4 de 19.
- Un programa puede tomar más CPU de la que le toca solo con hacer E/S en el momento justo.
- **Problema 3**: un trabajo también puede *cambiar*. Uno de CPU que se vuelve interactivo queda abajo para siempre: nunca vuelve a Q2.

`./mlfq -o 20 15@8:3/1`. La opción `-o` usa las reglas 4a/4b.

Regla 5: priority boost

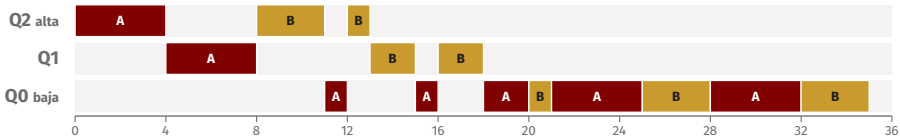
Regla 5: cada cierto periodo S, mover todos los trabajos a la cola más alta. Mismo ejemplo de la inanición, con $S = 16$:



- En $t = 16$, A sube a Q2 y entra en el Round Robin con B y C. Usa su asignación y vuelve a bajar.
- Resuelve la **inanición**: A recibe al menos 4 ticks cada 16.
- Resuelve el **cambio de comportamiento**: un trabajo que se volvió interactivo sube y se queda arriba.
- ¿Cuánto debe valer S? Muy grande: inanición. Muy chico: los interactivos comparten Q2 con todos. Ousterhout llamaba a estas constantes *voo-doo constants*.

Regla 4: llevar la cuenta

Regla 4 (reemplaza a 4a y 4b): cuando un trabajo agota su asignación en un nivel, baja, sin importar cuántas veces haya soltado la CPU.



- En Q2, B usa 3 ticks, hace E/S y usa 1 más: completa 4 y baja, aunque soltó la CPU.
- En Q1 le pasa lo mismo.
- En Q0, B y A se turnan en Round Robin. Entre $t = 8$ y $t = 27$, A recibe 8 ticks en vez de 4.
- **Pregunta de repaso 3**: la regla 4 evita las trampas; la regla 5 evita la inanición.

`./mlfq 20 15@8:3/1`, sin `-o`.

Las dos versiones en el simulador

reglas 4a/4b

```
$ ./mlfq -o 20 15@8:3/1
     0    5    10   15   20   25
Q2   AAAA....BBB.BBB.BBB.BBB.BBB..
Q1   ....AAAA.....................
Q0   ...........A...A...A...A...AA
```

regla 4

```
$ ./mlfq 20 15@8:3/1
     0    5    10   15   20   25
Q2   AAAA....BBB.B................
Q1   ....AAAA.....BB.BB...........
Q0   ...........A...A..AABAAAABBBA
```

- Cada carácter es un tick; cada fila, una cola. Se muestran los primeros 29 ticks.
- `-b S` agrega el boost; `-a` cambia la asignación; `-q`, el quantum.
- Con el simulador `mlfq.py` de OSTEP, `-S` equivale a nuestra `-o` y `-B` al boost.

SECCIÓN 06

MLFQ en la práctica

06



Las cinco reglas

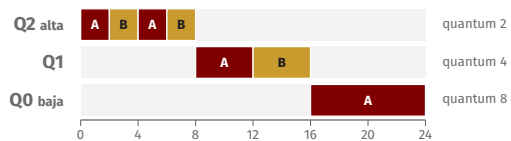
1. Si Prioridad(A) $>$ Prioridad(B), corre A (B no).
2. Si Prioridad(A) $=$ Prioridad(B), A y B se turnan en Round Robin con el quantum de su cola.
3. Un trabajo nuevo entra en la cola más alta.
4. Cuando un trabajo agota su asignación en un nivel, baja un nivel, sin importar cuántas veces haya soltado la CPU.
5. Cada periodo S, todos los trabajos suben a la cola más alta.

Aprender del pasado

MLFQ no pide la duración de nadie: la *mide*. Los trabajos cortos e interactivos se ven arriba y reciben buena respuesta; los largos bajan y reciben un trato justo gracias al boost.

Ajuste: cuántas colas, qué quantum

- Colas altas: trabajos interactivos, quantum corto (por ejemplo, 10 ms).
- Colas bajas: trabajos de CPU, quantum largo (cientos de ms). Menos cambios de contexto donde no importa la respuesta.
- Parámetros a elegir: número de colas, quantum y asignación por cola, periodo S del boost.
- No hay valores correctos para todas las cargas: se ajustan *(tuning)* midiendo.



Quantum distinto por cola (diagrama ilustrativo).

MLFQ en sistemas reales

- **Solaris** (clase *Time-Sharing*): una tabla que el administrador puede editar. Por defecto, 60 colas; quantum de 20 ms arriba a unos cientos de ms abajo; boost cada segundo, más o menos.
- **BSD Unix** (FreeBSD 4.3): no usa una tabla, sino una *fórmula*. La prioridad depende de cuánta CPU usó el proceso, y ese uso se olvida poco a poco *(decay-usage)*: el boost aparece de forma gradual.
- **Windows NT** y sucesores: también una variante de MLFQ.
- **Consejos del usuario**: `nice` sube o baja (un poco) la prioridad de un proceso. Algunos sistemas reservan los niveles más altos para el kernel.

Y Linux

Linux usa otra idea: CFS reparte la CPU en proporción, sin colas de prioridad. Próxima clase.

SECCIÓN 07

Tiempo real

07

Tiempo real: llegar tarde es fallar

- Hasta ahora medimos promedios. Un freno ABS no puede llegar tarde ni una vez en mil.
- **Tarea periódica**: cada T ticks suelta una instancia que necesita C ticks de CPU (peor caso) y debe terminar antes de su **plazo** *(deadline)* D. Aquí, $D = T$.
- **Duro** *(hard)*: perder un plazo es una falla (airbag, marcapasos). **Blando** *(soft)*: se degrada la calidad (video, juegos).
- Vuelve el supuesto 5: C se mide de antemano.

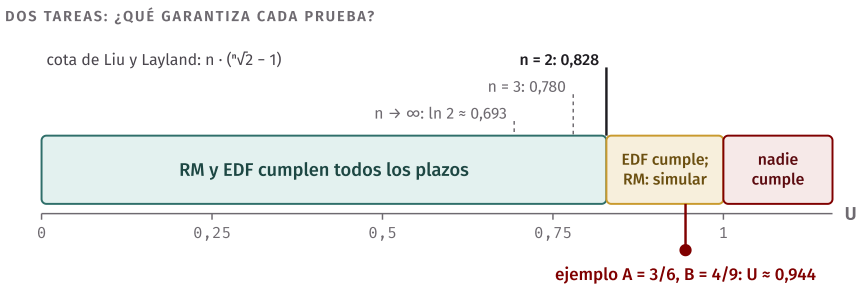
Utilización

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Qué fracción de la CPU piden todas las tareas juntas. Si $U > 1$, ninguna política cumple todos los plazos.

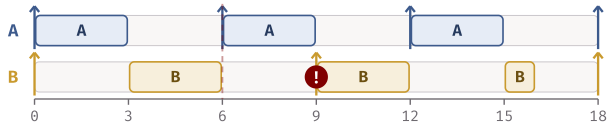
Rate-monotonic y EDF

- **Rate-monotonic (RM)**: prioridad fija, más alta cuanto más corto el periodo. Cumple si $U \leq n\,(2^{1/n} - 1)$ (Liu y Layland, 1973): prueba suficiente, no necesaria.
- **EDF** *(Earliest Deadline First)*: prioridad dinámica, el plazo más cercano primero. Cumple si y solo si $U \leq 1$.



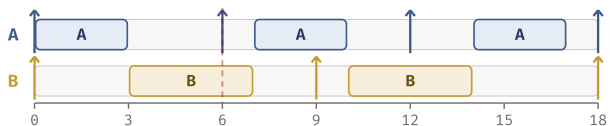
A = 3/6, B = 4/9 ($U \approx 0{,}944$): RM pierde un plazo, EDF no

RM · PRIORIDAD FIJA: A (T = 6) ANTES QUE B (T = 9)



B pierde su plazo en t = 9: le faltaba 1 tick de 4

EDF · PRIORIDAD DINÁMICA: EL PLAZO MÁS CERCANO



cumple todos los plazos

t = 6, llega A: RM corre A porque su periodo es menor;
EDF termina B porque su plazo (9) es antes que el de A (12)

↑ llegada de una instancia y plazo de la anterior (D = T)

El ejemplo en el simulador

bash

```
$ ./rtsched 3/6 4/9
U = 0.944   cota de RM para n = 2: 0.828
RM (prioridad fija: menor periodo primero)
gantt  AAABBBAAABBBAAAB..
plazo           !
plazos perdidos: 1
  B en t = 9 (le faltaba 1)
EDF (prioridad dinámica: plazo más ...)
gantt  AAABBBBAAABBBBAAA.
plazos perdidos: 0
```

- `code/week-03/rtsched.c`: cada tarea es `C/T`.
- Simula RM y EDF hasta el **hiperperiodo** (mcm de los periodos: 18); desde ahí todo se repite.
- Una instancia que pierde su plazo se descarta.

Salida recortada: también imprime la tabla de tareas, qué prueba aplica y los tramos.

En Linux: tres clases de tiempo real

- `SCHED_FIFO`: prioridad fija 1–99, sin quantum, antes que todo proceso normal. RM: más prioridad al periodo más corto.
- `SCHED_RR`: igual, con Round Robin entre iguales.
- `SCHED_DEADLINE`: EDF con runtime (C), deadline (D) y period (T). Si la suma de runtime/period pasa del 95 %, `EBUSY`: el kernel hace la prueba de U.
- Seguro: el tiempo real usa a lo más 0,95 s de cada segundo.

bash

```
$ chrt -m | grep -E 'FIFO|RR'
SCHED_FIFO min/max priority : 1/99
SCHED_RR min/max priority : 1/99
$ chrt -f 50 ./control
$ chrt -d \
    --sched-runtime  3000000 \
    --sched-deadline 6000000 \
    --sched-period   6000000 \
    0 ./control
```

`./control`: su programa. La tarea A en ns: 3 ms cada 6 ms. Cambiar de clase requiere `CAP_SYS_NICE`.

¿Por qué Linux no garantiza plazos?

Las pruebas de RM y EDF suponen desalojo instantáneo, C conocido y nada más en la CPU.

- **El peor caso de C es enorme**: cachés y TLB fríos, fallos de página, swap.
- **Partes del kernel no se desalojan**: interrupciones deshabilitadas, spinlocks tomados; las interrupciones de dispositivos van antes que todo.
- **Inversión de prioridades**: la tarea urgente espera un lock que tiene una de baja prioridad, y a esta no la dejan correr las de prioridad media.
- `SCHED_FIFO` y `SCHED_RR` **no hacen control de admisión**: nadie suma U.

Entonces

`PREEMPT_RT` (en el kernel principal desde 6.12) baja la latencia a decenas de µs: suficiente para mucho tiempo real blando. Para tiempo real duro certificado se usa un **RTOS** (FreeRTOS, Zephyr, QNX, VxWorks).

Resumen

- **Tiempo de retorno** = fin − llegada; **tiempo de respuesta** = primera ejecución − llegada.
- **FIFO** sufre el efecto convoy. **SJF** y **STCF** (apropiativo) minimizan el retorno, pero necesitan conocer las duraciones.
- **Round Robin** minimiza la respuesta y castiga el retorno. El quantum debe amortizar el costo del cambio de contexto.
- Superponer E/S y CPU: cada ráfaga de CPU cuenta como un trabajo.
- **MLFQ** aprende del pasado. Reglas 1–3: prioridades y RR. Regla 4: contabilidad contra las trampas. Regla 5: boost contra la inanición.
- xv6 hace Round Robin simple sobre `proc[]`, con un quantum de un tick del temporizador.
- **Tiempo real**: $U = \sum C_i/T_i$. **RM** cumple bajo $n(2^{1/n} - 1)$; **EDF**, si y solo si $U \leq 1$. Linux: `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE`, sin garantías duras.

Para la próxima clase

- Leer OSTEP cap. 9: *Scheduling: Proportional Share*.
- **Laboratorio**: simuladores de OSTEP.
  - `scheduler.py`: preguntas 1–7 del cap. 7; a mano y luego con `-c`.
  - `mlfq.py`: preguntas 1–5 del cap. 8.
  - Construir una carga en la que MLFQ *sin boost* produzca inanición.
- **P0**: entrega esta semana.
- **P1**: se publica la shell `wish`. Entrega en la semana 6.

bash

```
$ ./scheduler.py -p SJF \
    -l 30,10,20
$ ./scheduler.py -p RR -q 5 \
    -l 30,10,20 -c
$ ./mlfq.py -n 3 -q 10 -S \
    -l 0,100,0:20,40,9 -c
```

Pregunta para pensar

Si cada quantum se sortea y A tiene el doble de boletos que B, ¿cuánta CPU recibe A?