Scheduling

Workload Assumptions

El **Workload** es la carga de trabajo de un proceso corriendo en el sistema.

Cómo se calcula el **workload** es fundamental para determinar partes de las políticas de planificación. Cuanto mejor es el cálculo, mejor es la política.
Los supuestos sobre los procesos o **jobs** que se encuentran en ejecución son:

1. Cada proceso se ejecuta por la misma cantidad de tiempo.
2. Todos los jobs llegan al mismo tiempo.
3. Una vez que empieza un job, sigue hasta completarse.
4. Todos los jobs usan **únicamente cpu**. (no performan I/O)
5. El tiempo de ejecución (run-time) de cada job es conocido.

Scheduling Metrics

Para poder hacer algún tipo de análisis se debe tener algún tipo de métrica estandarizada para comparar las distintas **políticas de planificación o scheduling**. Bajo estas premisas, por ahora, se utilizará una única métrica llamada **turnaround time**.

Turnaround Time

Se define como el tiempo que se completa el proceso menos el tiempo en el que llega al sistema:

$$T_{turnaround} = T_{completion} - T_{arrival}$$

> Debido a que asumimos que todos los procesos llegan al mismo tiempo (2), el $T_{arrival} = 0$ y por lo tanto $T_{turnaround} = T_{completion}$.

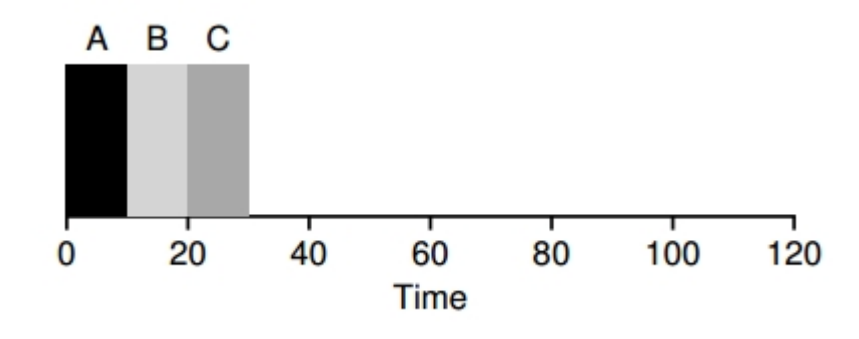
> Turnaround time es una métrica que mide **performance**.

First In, First Out (FIFO)

Es el algorítmo más básico de scheduling que se puede implementar.
Es simple y fácil de implementar. Dadas nuestras suposiciónes, funciona bastante bien.

Ejemplo simple de FIFO

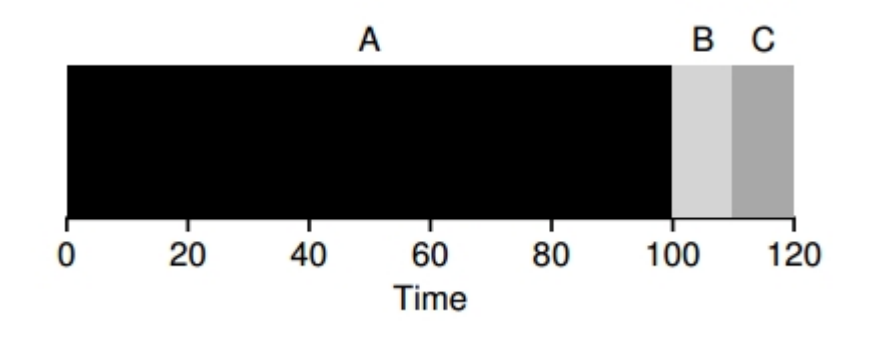
Llegan 3 procesos A, B y C al mismo tiempo ($T_{arrival} = 0$).
Cada proceso corre por 10 segundos.



Podemos ver que A termina en 10, B en 20, y C en 30.
El turnaround time promedio para los 3 procesos es: $\frac{10+20+30}{3} = 20$.

Por qué FIFO no es tan genial

Ahora relajemos la suposición 1, ya no asumimos que cada proceso corre por la misma cantidad de tiempo.
Esta vez, A corre por 100 segundos mientras que B y C corren por 10 cada uno.

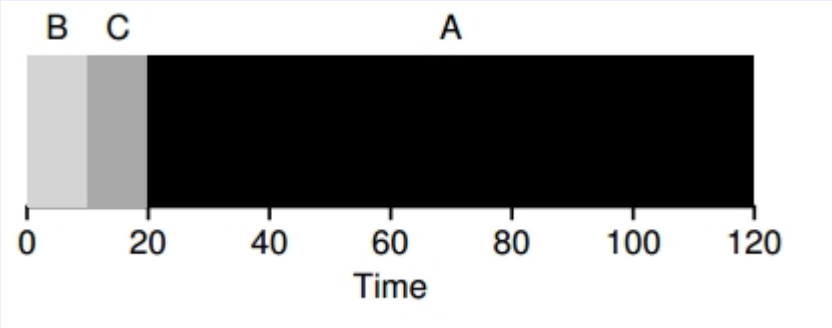


A corre primero por los 100 segundos antes que B y C tengan chance de correr.
El turnaround time promedio es alto: $\frac{100+110+120}{3} = 110$.
Este problema se conoce como **Convoy Effect**. Debido a que los procesos solo pueden tener tiempo de CPU cuando el proceso anterior se termine de ejecutar, el Sistema Operativo entero se retrasa debido a procesos lentos.

Shortest Job First (SJF)

Para resolver el problema que se presenta en la política FIFO, se modifica la política para que se ejecute primero el proceso de duración mínima. Una vez finalizado este proceso, se ejecuta el siguiente proceso de duración mínima y así sucesivamente.

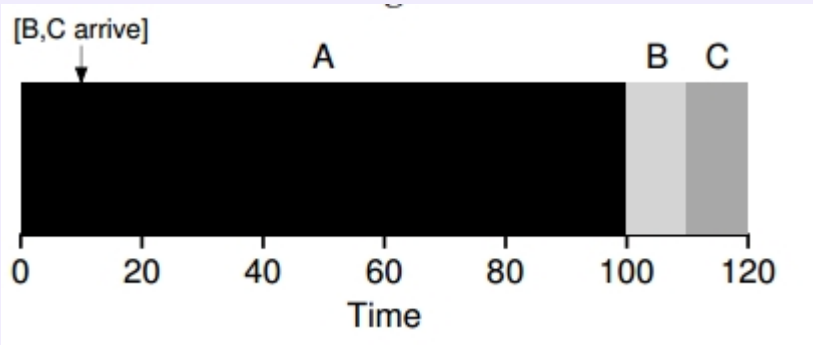
Ejemplo simple de SJF



Simplemente corriendo B y C antes que A, SJF reduce el turnaround time promedio de 110 a 50 segundos.
($\frac{10+20+120}{3} = 50$)

⁝≡ SFJ con llegadas tarde de B y C

Ahora relajemos la suposición 2, asumimos ahora que los procesos pueden llegar a cualquier tiempo en vez de todos a la misma vez.
Esta vez, A llega a t=0 y necesita correr por 100 segundos, B y C llegan a t=10 y cada uno necesita correr por 10 segundos.



Aunque B y C llegaron apenas después que A, estan forzados a esperar hasta que A termine, y nuevamente tenemos el problema del convoy.
El turnaround time promedio para estos 3 procesos es 103.33 segundos ($\frac{100+(110-10)+(120-10)}{3} = 103.33$).

Shortest Time-to-Completion First (STCF)

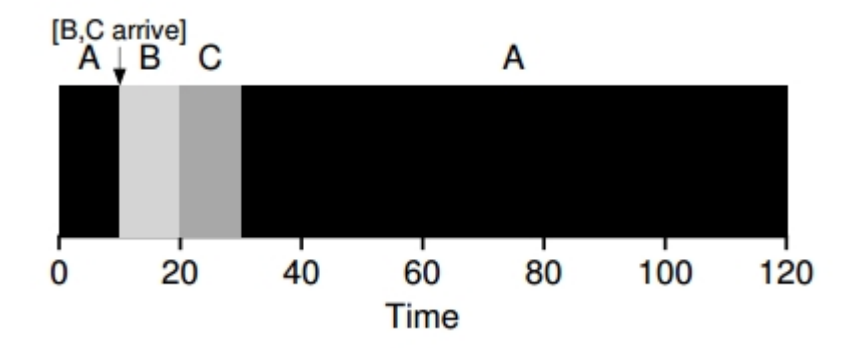
Para poder solucionar el problema anterior, se necesita relajar la suposición 3 (que los procesos deben correr hasta que terminen).
La idea es que el scheduler pueda adelantarse y determinar qué proceso debe ser ejecutado. Entonces, cuando los procesos B y C llegan, el shceduler se puede adelantar al proceso A y decidir correr otro proceso, tal vez retomar la ejecución del proceso A más tarde.

> SFJ es una política **non-preemptive.**

Cada vez que un proceso entra al sistema, STCF determina cual de los procesos restantes (incluido el nuevo proceso) tiene el menor tiempo restante, y planifica ese.

Ejemplo simple de STCF

STFC se adelanta a A y corre B y C hasta que terminen. Solo cuando estos dos terminen se terminará de correr A.



El resultado es un turnaround time promedio muy mejorado: 50 segundos.
($\frac{(120-0)+(20-10)+(30-10)}{3} = 50$)

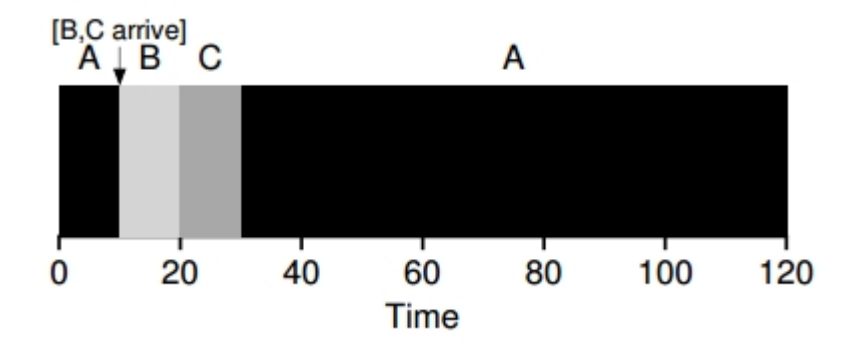
Response Time

Response Time

Se define como el tiempo desde que el proceso llega al sistema hasta la primera vez que es planificado para correr.

$$T_{response} = T_{firstrun} - T_{arrival}$$

Ejemplo



Si tenemos que A llega en t=0, y B y C en t=10, el response time de cada proceso es:
0 para A, 0 para B, y 10 para C (promedio: 3.33).

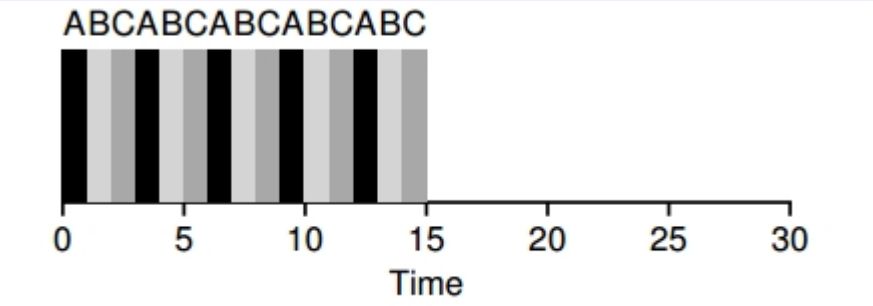
> STFC y SFJ no son buenos en response time aunque si en turnaround time.

Round Robin

La idea básica es: en vez de correr los procesos hasta que terminen, RR corre un proceso por un **time slice** y luego cambia al siguiente proceso en la cola. Hace esto repetidamente hasta que los procesos terminen.

Ejemplo

Tres procesos A, B y C llegan al mismo tiempo al sistema, y cada uno desea correr por 5 segundos.
RR con un time-slice de 1 segundo va a rotar por los procesos rapidamente.
El response time promedio de RR es: $\frac{0+1+2}{3} = 1$



Lo importante de RR es la elección de un **buen time slice**, se dice que el time slice tiene que amortizar el costo del cambio de contexto sin hacer que esto resulte en que el sistema no responda más.

> RR es bueno en response time pero malo en turnaround time.

Multi-Level Feedback Queue

MLFQ intenta atacar principalmente dos problemas:

1. Intenta optimizar el **Turnaround Time**, que se realiza mediante la ejecución de la tarea más corta primero. Desafortunadamente el sistema operativo nunca sabe a priori cuanto va a tardar en correr una tarea.
2. Intenta que el scheduler haga sentir al usuario que el sistema tiene tiempo de respuesta interactivo, por ende minimizar el **Response Time.**
   Desafortunadamente los algoritmos como Round-Robin reducen el **Response Time** pero tienen un terrible **Turnaround Time**.

MLFQ: Basic Rules

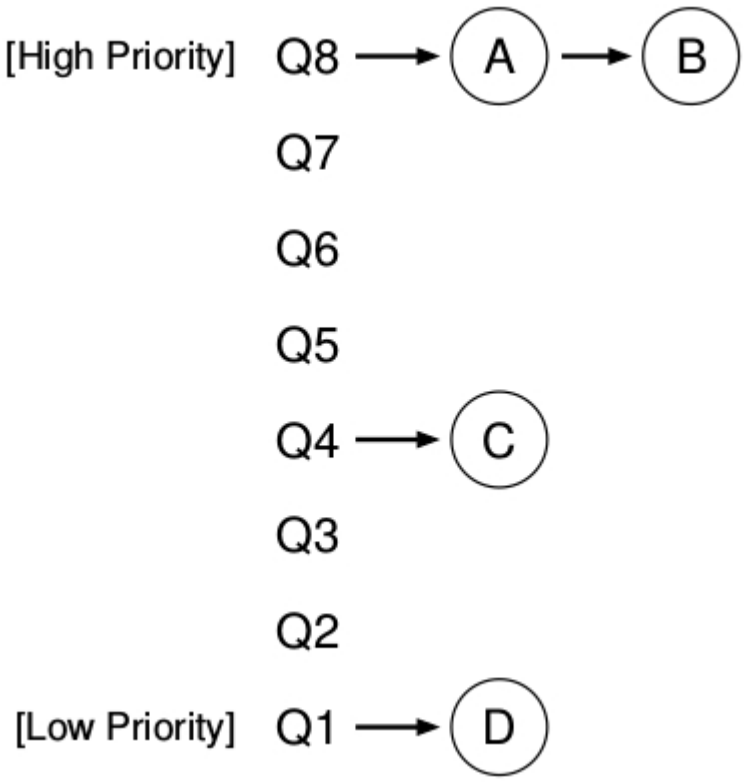
MLFQ tiene un **conjunto de distintas colas**, cada una de estas colas tiene asignado un nivel de **prioridad**.
En un determinado tiempo, un proceso que está listo para correr está en una única cola. MLFQ usa las prioridades para decidir cual proceso debería correr en un determinado tiempo: el proceso con mayor prioridad o el proceso en la cola de mayor prioridad, será elegidp para correr.

Dado el caso que existan más de un proceso con la misma prioridad entonces se utilizara el algoritmo de **Round Robin** para planificar estos procesos.

Las 2 reglas básicas de MLFQ:

- **REGLA 1**: si la prioridad de (A) **es mayor** que la prioridad de (B), (A) se ejecuta y (B) no.
- **REGLA 2**: si la prioridad de (A) **es igual** a la prioridad de (B), (A) y (B) se ejecutan en **Round-Robin**

Si un proceso repetidamente renuncia al uso de la CPU mientras espera el input del teclado, MLFQ mantendrá su prioridad alta.
Si un proceso usa la CPU por largos periodos de tiempo, MLFQ reducirá su prioridad.
MLFQ intentará aprender sobre los procesos mientras corren, y usar el historial del proceso para predecir su comportamiento futuro.

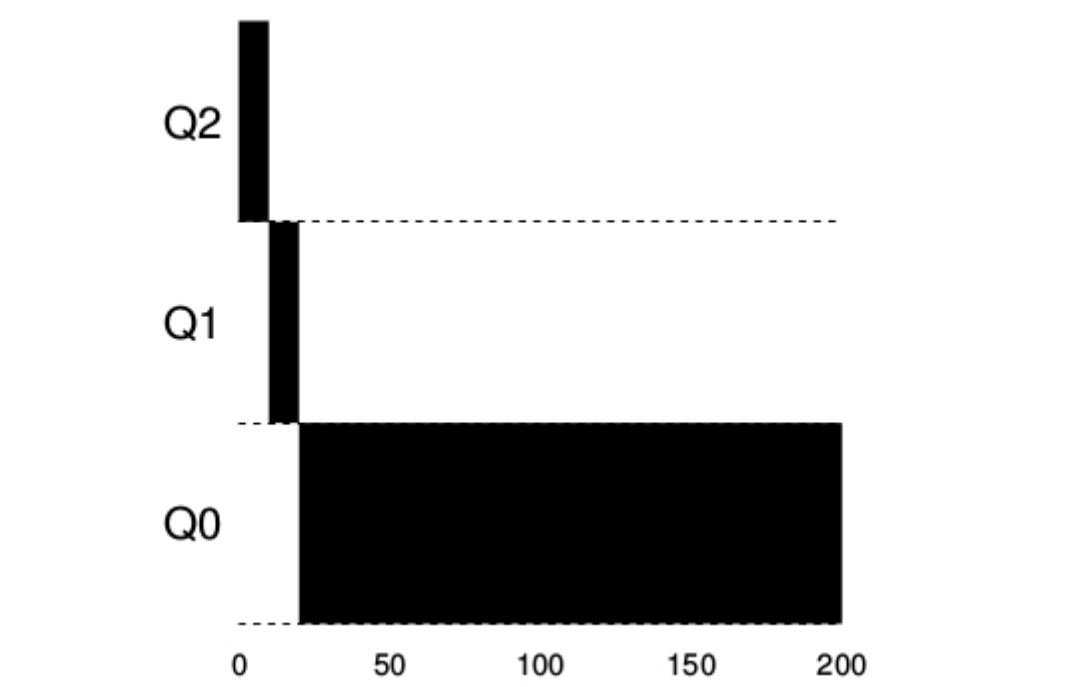


Attempt #1: Como cambiar la prioridad

Se debe decidir como MLFQ va a cambiar el nivel de prioridad de un proceso durante toda la vida del mismo (por ende en que cola esta va a residir).
Para esto hay que tener en cuenta nuestra carga de trabajo (workload): una mezcla de tareas interactivas que tienen un corto tiempo de ejecución (y que pueden renunciar a la utilización de la CPU), y algunas tareas de larga ejecución ligadas a la CPU que necesitan mucho tiempo de CPU, pero dond el tiempo de respuesta no es importante.

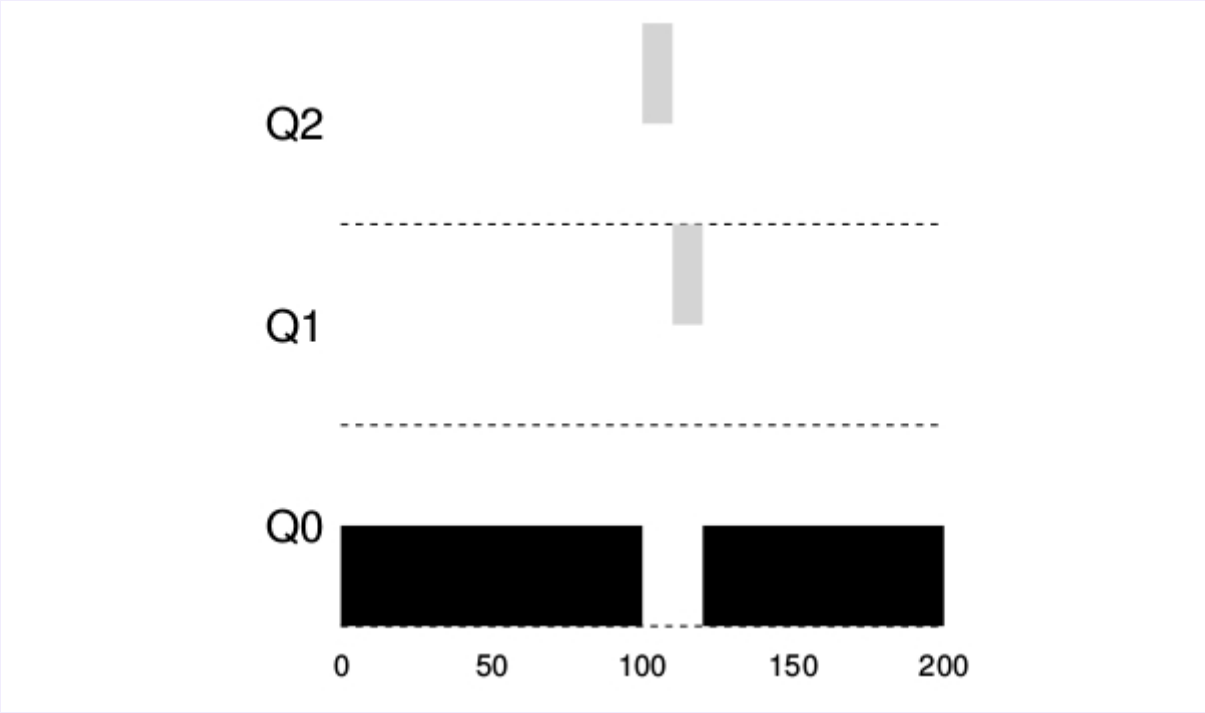
- **REGLA 3**: Cuando un proceso entra en el sistema se pone con la más alta prioridad (la cola más alta).
- **REGLA 4a**: Si un proceso usa un time slice completo mientras se está ejecutando, su prioridad se reduce (baja a una cola menor).
- **REGLA 4b**: Si un proceso renuncia al uso de la CPU antes de un time slice completo, se queda en el mismo nivel de prioridad.

Ejemplo 1: Un solo proceso largo



El proceso entra con la prioridad más alta. Después de un time-slice de 10ms, el scheduler reduce su prioridad en una unidad, ahora el proceso está en Q1. Después de correr en Q1 por un time-slice, el proceso es llevado a la menor prioridad en el sistema (Q0), donde permanece.

Ejemplo 2: Llega un proceso corto



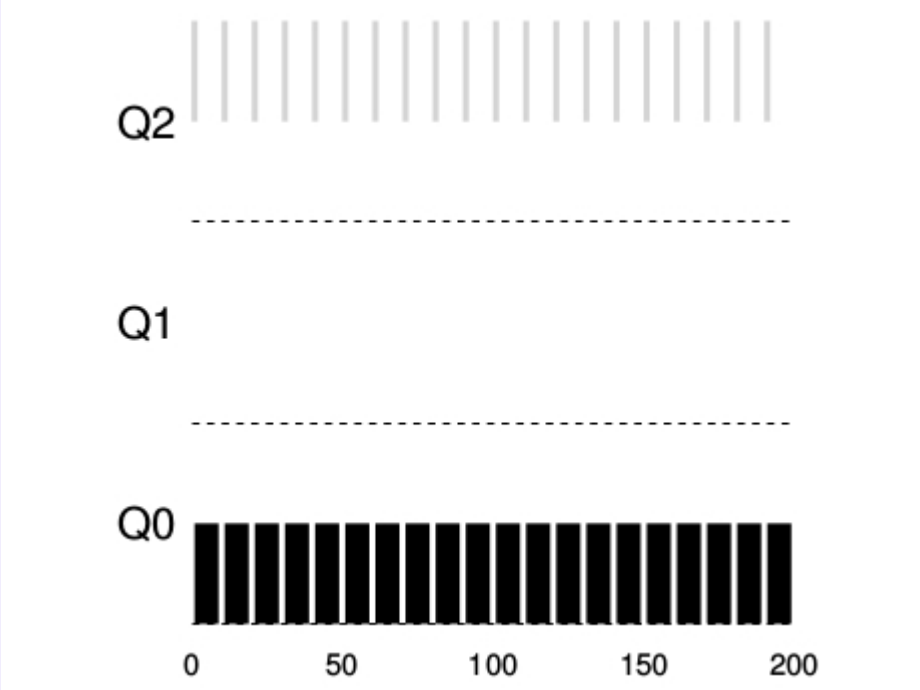
Hay dos procesos: A que es un proceso largo que usa CPU, y B que es un proceso corto e interactivo.
A corrió por un tiempo, y luego llega B.
A (negro) está corriendo en la cola de menor prioridad; B (gris) llega en T=100, y está en la cola más alta por que su run-time es corto (solo 20ms).
B termina antes de alcanzar la cola del fondo, en dos time-slices.
Luego A sigue corriendo (en la prioridad mas baja).

El scheduler no sabe si un proceso va a ser corto o largo, primero asume que puede ser un proceso corto entonces le da la mayor prioridad. Si realmente es un proceso corto, va a correr rapido y terminar; si no es un proceso corto, se va a mover lentamente hacia colas de menor prioridad.

Ejemplo 3: Que pasa con E/S?

Consideramos la regla 4b, si el proceso renuncia al uso del procesador antes de usar todo su time slice, lo mantenemos en el mismo nivel de prioridad.
Si un proceso es interactivo (E/S), va a renunciar al uso del CPU antes de que se complete su time slice. En ese caso, no queremos penalizar al proceso y

simplemente lo mantenemos en el mismo nivel.



Problemas con este MLFQ

1. **Starvation**: si hay muchos procesos interactivos en el sistema, estos van a consumir todo el tiempo de CPU y entonces los procesos largos nunca van a recibir tiempo de CPU (they **starve**).
2. Un usuario inteligente podría reescribir sus programas para sacar ventaja del scheduler y obtener más tiempo de CPU. Por ejemplo, hacer que el programa haga una operación de E/S antes de que termine su time slice. Hacer eso le permite seguir en la misma cola de prioridad y asi obtener un mayor porcentaje de tiempo de CPU.
3. Un programa puede cambiar su comportamiento con el tiempo; puede transicionar a interactivo cuando era ligado a usar la CPU. Con este MLFQ, ese proceso puede tener mala suerte y no ser tratado de la misma manera que a los demas procesos interactivos del sistema.

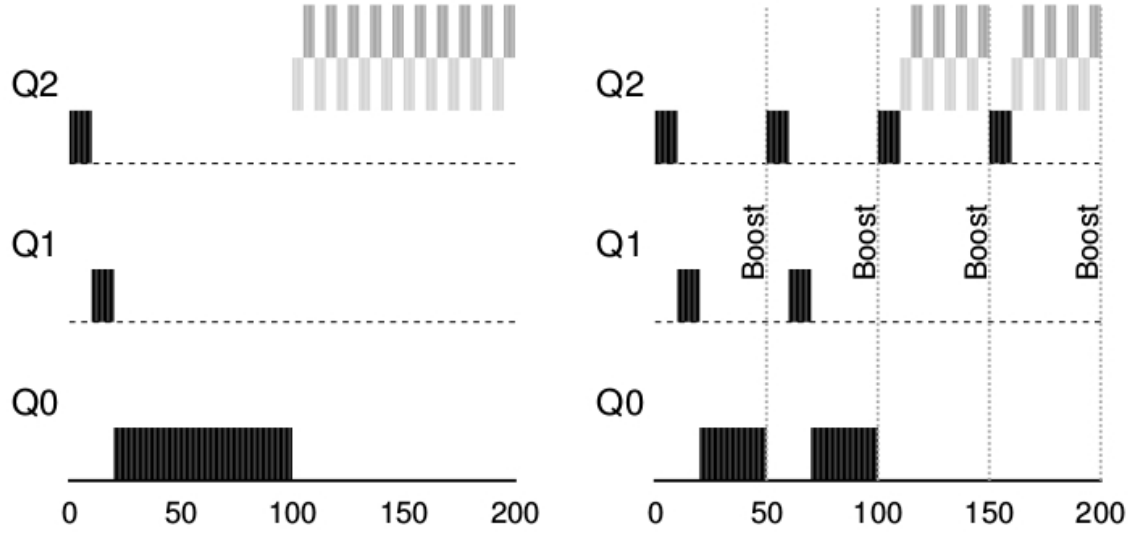
Attempt #2: Priority Boost

La idea es periodicamente hacer un boost en la prioridad de todos los procesos en el sistema.

- **REGLA 5:** Después de algun periodo de tiempo **S**, mover a todos los procesos del sistema a la cola más alta.

Esta nueva regla resuelve dos problemas:

1. Se garantiza a los procesos que no se van a **starve**: Al ubicarse en la cola más alta con los procesos de alta prioridad, se van a ejecutar utilizando **round-robin** y por ende en algún momento recibirá atención.
2. Si un proceso que consume CPU se transforma en interactivo, el scheduler lo tratará como tal una vez que haya recibido el boost de prioridad.



Cuanto debería ser el valor del tiempo S?. Algunos investigadores suelen llamar a este tipo de valores dentro de un sistema **VOO-DOO CONSTANTs** porque parece que requieren cierta magia negra para ser determinados correctamente.

Si el valor de **S** es demasiado **alto**, los procesos de ejecución larga van a caer en **starvation**; si se setea a ***S*** con valores **muy pequeños** los procesos interactivos no van a poder compartir adecuadamente la CPU.

Attempt #3: Better Accounting

Se debe solucionar otro problema: Como prevenir que **ventajeen (gaming)** al scheduler.

La solución es llevar una mejor **contabilidad del tiempo de uso de la CPU** en todos los niveles del MLFQ.

En lugar de que el planificador se olvide de cuanto time slice un determinado proceso utiliza en un determinado nivel, el planificador debe seguir la pista desde que un proceso ha sido asignado a una cola hasta que es trasladado a una cola de distinta prioridad. Ya sea si usa su time slice de una o en pequeños trocitos.

Se reescriben las reglas 4a y 4b en una única regla:

- **REGLA 4**: Una vez que una tarea usa su asignación de tiempo en un nivel dado (independientemente de cuantas veces haya renunciado al uso de la CPU), su

prioridad se reduce (baja un nivel en la cola de prioridad).