Sistemas Operativos
Scheduling

Scheduling o Planificación de Procesos

Cuando hay múltiples cosas qué hacer ¿Cómo se elige cuál de ellas hacer primero?

Debe existir algún mecanismo que permita determinar cuanto tiempo de CPU le toca a cada proceso. Ese período de tiempo que el kernel le otorga a un proceso se denomina time slice o time quantum.

Multiprogramación

Más de un proceso estaba preparado para ser ejecutado en algún determinado momento, y el sistema operativo intercalaba dicha ejecución según la circunstancia. Haciendo esto se mejoró efectivamente el uso de la CPU, tal mejora en la eficiencia fue particularmente decisiva en esos días en la cual una computadora costaba cientos de miles o tal vez millones de dólares.

En esta era, múltiples procesos están listos para ser ejecutados un determinado tiempo según el S.O. lo decidiese en base a ciertas políticas de planificación o scheduling.

Time Sharing

Tiempo compartido se refiere a compartir de forma concurrente un recurso computacional (tiempo de ejecución en la CPU, uso de la memoria, etc.) entre muchos usuarios por medio de las tecnologías de multiprogramación y la inclusión de interrupciones de reloj por parte del sistema operativo, permitiendo a este último acotar el tiempo de respuesta del computador y limitar el uso de la CPU por parte de un proceso dado.

Time Sharing

A medida que los tiempos de respuesta entre procesos se fueron haciendo cada vez más pequeños, más procesos podían ser cargados en memoria para su ejecución.

Una variante de la técnica de multiprogramación consistió en asignar una terminal a cada usuario en línea.

Time Sharing

Teniendo en cuenta que los seres humanos tienen un tiempo de respuesta lento (0.25 seg para estímulos visuales) en comparación a una computadora (operación en nanosegundos). Debido a esta diferencia de tiempos y a que no todos los usuarios necesitan de la cpu al mismo tiempo, este sistema daba la sensación de asignar toda la computadora a un usuario determinado Corbató y otros. Este concepto fue popularizado por MULTICS.

Utilización de la CPU

Si se asume que el 20% del tiempo de ejecución de un programa es sólo cómputo y el 80% son operaciones de entrada y salida, con tener 5 procesos en memoria se estaría utilizando el 100% de la CPU.

Siendo un poco más realista se supone que las operaciones de E/S son bloqueantes (una operación de lectura a disco tarda 10 miliseg y una instrucción registro registro tarda 1-2 nanoseg), es decir, paran el procesamiento hasta que se haya realizado la operación de E/S.

Multiprogramación y Utilización de la CPU

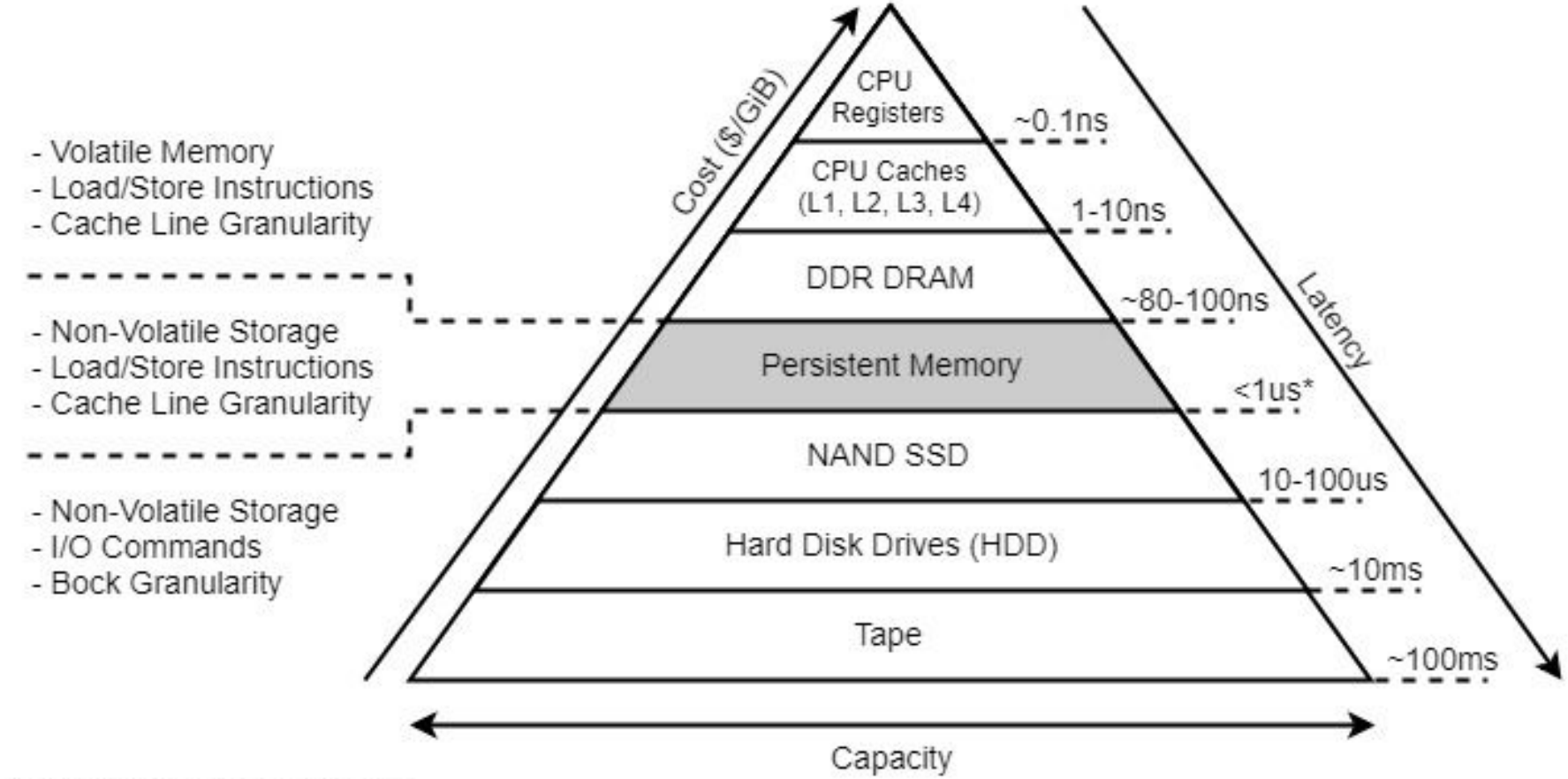
Entonces, el cálculo es más realista si se supone que un proceso gasta una fracción p, bloqueado en E/S. De esta manera, si tenemos n procesos esperando para hacer operaciones de entrada y salida, la probabilidad de que los n procesos estén haciendo E/S es p_n

Por ende la probabilidad de que se esté ejecutando algún proceso es $1-p_n$, esta fórmula es conocida como utilización de CPU.

Por ejemplo: si se tiene un solo proceso en memoria y este tarda un 80% del tiempo en operaciones de E/S el tiempo de utilización de CPU es 1-0.8 = 0.2 que es el 20%.

Ahora bien, si se tienen 3 procesos con la misma propiedad, el grado de utilización de la cpu es $1-0.83= 0.488$ es decir el 48 de ocupación de la cpu.

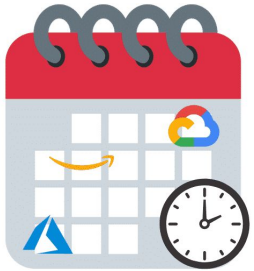
Si se supone que se tienen 10 procesos, entonces la fórmula cambia a $1-0.810=0.89$ el 89% de utilización, aquí es donde se ve la IMPORTANCIA de la Multiprogramación.



(*) See vendor specifications

Planificación

Cuando un Sistema Operativo se dice que realiza multi-programación de varios procesos debe existir una entidad que se encargue de coordinar la forma en que estos se ejecutan, el momento en que estos se ejecutan y el momento en el cual paran de ejecutarse. En un sistema operativo esta tarea es realizada por el **Planificador** o S**cheduler** que forma parte del Kernel del Sistema Operativo.



Políticas Para Sistemas Mono-procesador

El Workload

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

Determinar cómo se calcula el workload es fundamental para determinar partes de las políticas de planificación. Cuanto mejor es el cálculo, mejor es la política. Las suposiciones que se harán para el cálculo del workload son más que irreales.

Los supuestos sobre los procesos o jobs que se encuentran en ejecución son:

- **Cada proceso se ejecuta la misma cantidad de tiempo.**
- **Todos los jobs llegan al mismo tiempo para ser ejecutados.**
- **Una vez que empieza un job sigue hasta completarse.**
- **Todos los jobs usan únicamente cpu.**
- **El tiempo de ejecución (run-time) de cada job es conocido.**

Métricas de Planificación

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, para que todo sea simple se utilizará una única métrica llamada ***turnaround time.*** Que se define como ***el tiempo en el cual el proceso se completa menos el tiempo de arribo al sistema***:

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

Debido a 2 el $T_{arrival} = 0$

Hay que notar que el turnaround time es una métrica que mide performance.

Políticas de Scheduling Mono Core

- First In, First Out (FIFO)
- Shortest Job First (SJF)
- Shortest Time-to-Completion (STCF)
- Round Robin (RR)

First In, First Out (FIFO)

El algoritmo más básico para implementar como política de planificaciones es el First In First Out o Fist Come, First Served.

Ventajas:

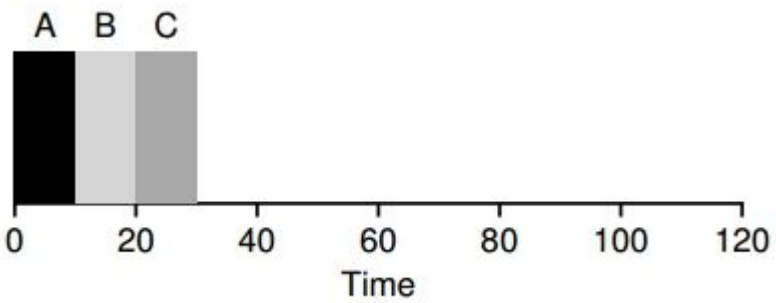
1. Es simple.

2. Por 1 es fácil de implementar.

3. Funciona bárbaro para las suposiciones iniciales.

First In, First Out (FIFO)

Por ejemplo se tiene tres procesos A, B y C con Tarrival=0.

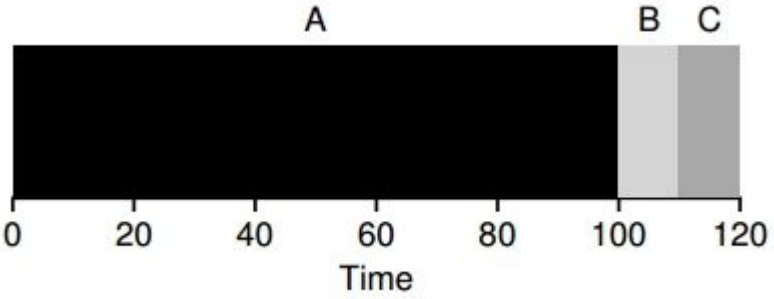


Si bien llegan todos al mismo tiempo llegaron con un insignificante retraso de forma tal que llegó A, B y C. S i se asume que todos tardan 10 segundos en ejecutarse… ¿cuánto es el Taround?

(10+20+30)/3=20

First In, First Out (FIFO)

Ahora relajemos la suposición 1 y no se asume que todas las tareas duran el mismo tiempo. A



¿Cuánto es el Taround?

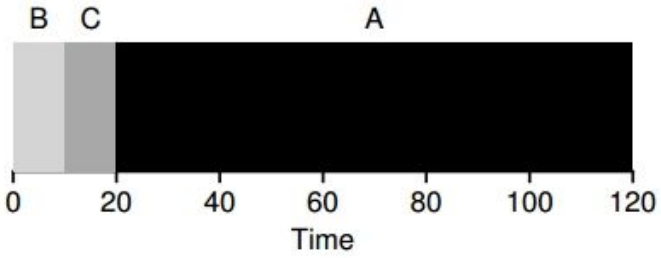
(100+110+120)/3=110

First In, First Out (FIFO)

Convoy effect

Shortest Job First (SJF)

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



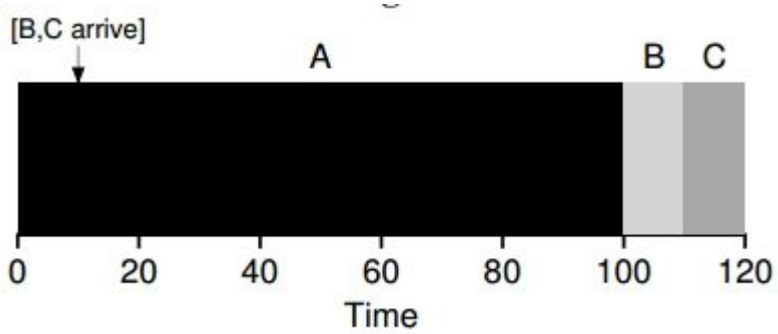
En el mismo caso d n el sencillo hecho de ejecutar B, C y A en ese orden:

(10+20+120)/3=50

Shortest Job First (SJF)

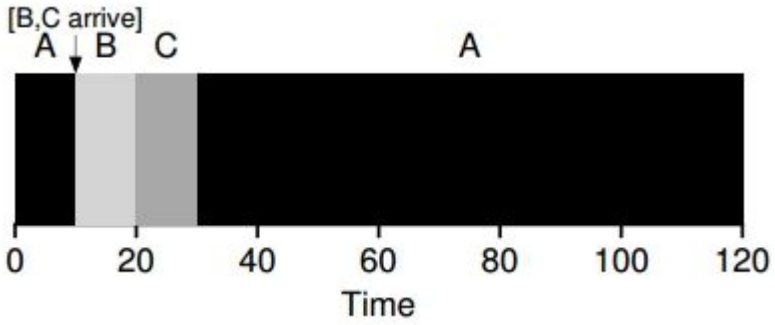
Utilizando SJF se obtuvo una significativa mejora… pero con las suposiciones iniciales que son muy poco realistas. Si se relaja la suposición 2, en la cual no todos los procesos llegan al mismo tiempo, por ejemplo llega el proceso A y a los 10 segundos llegan el proceso B y el proceso C. ¿Cómo sería el cálculo, ahora? _t_=10 seg

(100+110−10+120−10)/3=103.33



Shortest Time-to-Completion (STCF)

Para poder solucionar este problema se necesita relajar la suposición 3 (los procesos se tienen que terminar hasta el final). La idea es que el planificador o scheduler pueda adelantarse y determinar qué proceso debe ser ejecutado. Entonces cuando los procesos B y C llegan se puede desalojar (preempt [^1]) al proceso A y decidir que otro proceso se ejecute y luego retomar la ejecución del proceso A.

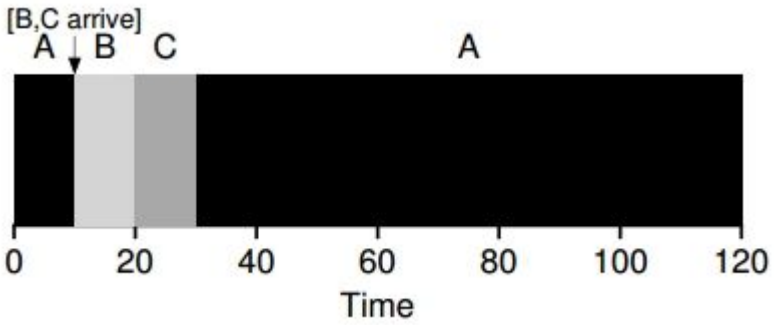


El caso anterior el de SFJ es una p

Shortest Time-to-Completion (STCF)

El cálculo para el turnaround time sería

(120−0+20−10+30−10)/3=50



Una nueva métrica: Tiempo de Respuesta

El tiempo de respuesta o response time surge con el advenimiento del time-sharing ya que los usuarios se sientan en una terminal de una computadora y pretenden una interacción con rapidez. Por eso nace el response time como métrica:

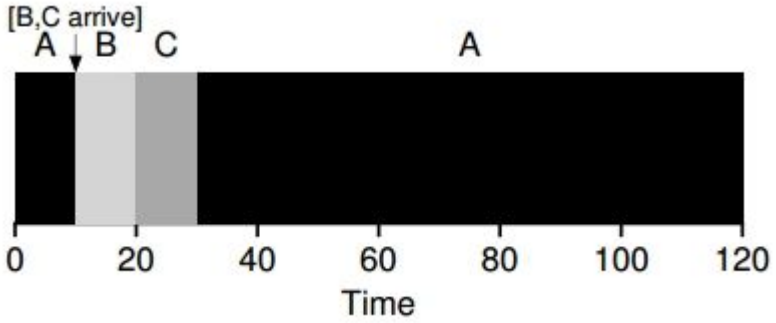
$$T_{response} = T_{firstrun} - T_{arrival}$$

Una nueva métrica: Tiempo de Respuesta

El $T_{response}$ del proceso A es 0.
El $T_{response}$ del proceso B es… 0… llega en 10 pero tarda 10 (10-10)
El $T_{response}$ del proceso C es… 10… llega en 10 pero termina en 20 (20-10)

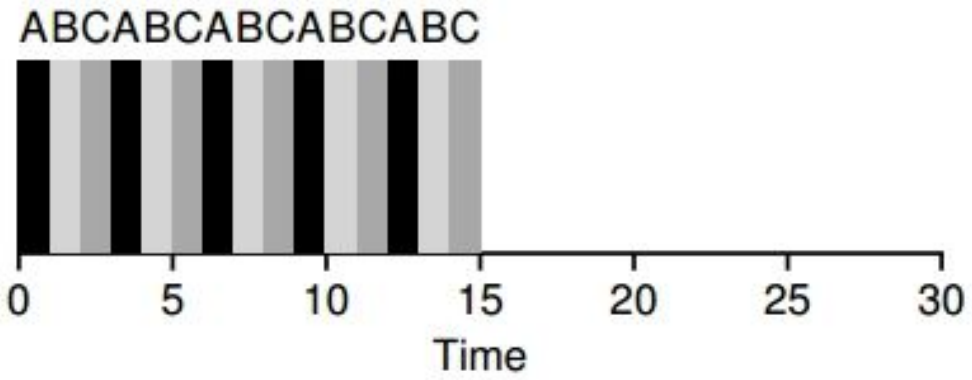
En promedio el $T_{response}$ es de 3.33 seg. Entonces

¿Cómo escribir un planificador que t



Round Robin

La idea del algoritmo es bastante simple, se ejecuta un proceso por un período determinado de tiempo (slice) y transcurrido el período se pasa a otro proceso, y así sucesivamente cambiando de proceso en la cola de ejecución.



Round Robin

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

Por ejemplo, si el tiempo de cambio de contexto está seteado en 1 ms y el time slice está seteado en 10 ms, el 10% del tiempo se estará utilizando para cambio de contexto.

Sin embargo, si el time slice se setea en 100 ms, solo el 1% del tiempo será dedicado al cambio de contexto. ¿Qué pasa si se trae a colación a la métrica del turnaround time ?

La Vida Real

Planificación en la Vida Real

¿Qué debería proporcionar un marco de trabajo básico que permita pensar en políticas de planificaciones ?

¿Cuáles deberían ser las suposiciones a tener en cuenta?

¿Cuáles son las métricas importantes?

Multi Level Feedback Queue

Esta técnica llamada Multi-Level Feedbak Queue de planificación fue descripta inicialmente en los años 60 en un sistema conocido como Compatible Time Sharing System CTSS. Este trabajo en conjunto con el realizado sobre MULTICS llevó a que su creador ganara el Turing Award.

Este planificador ha sido refinado con el paso del tiempo hasta llegar a las implementaciones que se encuentran hoy en un sistema moderno.

Multi Level Feedback Queue

MLQF intenta atacar principalmente 2 problemas:

- 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 cuánto va a tardar en correr una tarea.

- MLFQ intenta que el planificador haga sentir al sistema con un tiempo de respuesta interactivo para los usuarios por ende minimizar el **response time**; desafortunadamente los algoritmos como round-robin reducen el **response time** pero tienen un terrible **turnaround time**.

Multi Level Feedback Queue

Entonces:

- ¿Cómo se hace para que un planificador pueda lograr estos dos objetivos si generalmente no se sabe nada sobre el proceso a priori?.

- ¿Cómo se planifica sin tener un conocimiento acabado?

- ¿Cómo se construye un planificador que minimice el tiempo de respuesta para las tareas interactivas y también minimice el **timearound time** sin un conocimiento a priori de cuanto dura la tarea?

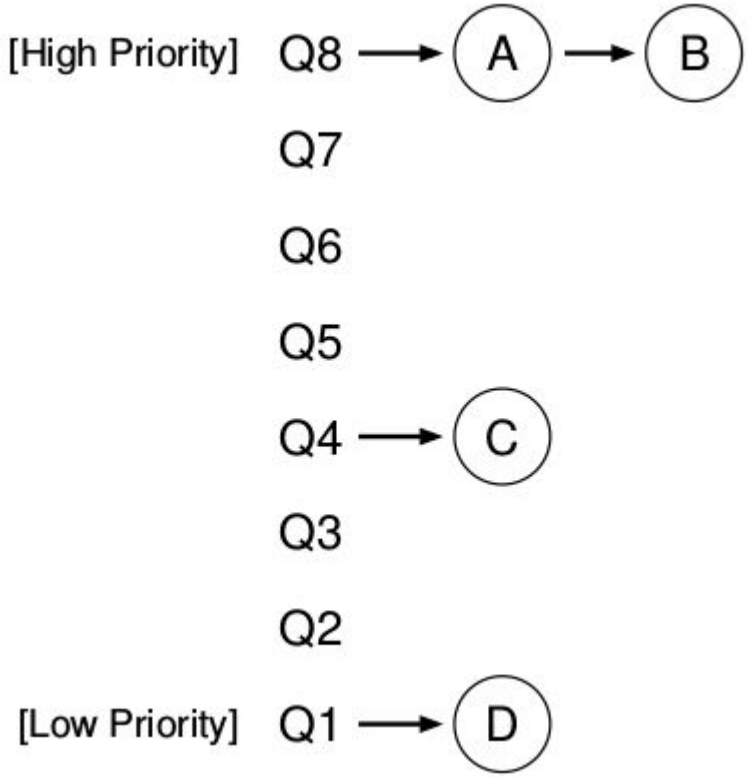
MLQF: Las reglas básicas

MLFQ tiene un conjunto de distintas colas, cada una de estas colas tiene asignado un nivel de prioridad.

En un determinado tiempo, una tarea que está lista para ser corrida está en una única cola. MLFQ usa las prioridades para decidir cual tarea debería correr en un determinado tiempo t0: la tarea con mayor prioridad o la tarea en la cola mas alta sera elegida para ser corrida.

Dado el caso que existan más de una tarea con la misma prioridad entonces se utilizará el algoritmo de Round Robin para planificar estas tareas entre ellas.

MLQF: Las reglas básicas



MLQF: Las reglas básicas

Las 2 reglas básicas de MLFQ:

REGLA 1: si la prioridad (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.

La clave para la planificación MLFQ subyace entonces en cómo el planificador setea las prioridades. En vez de dar una prioridad fija a cada tarea, MLFQ varía la prioridad de la tarea basándose en su comportamiento observado.

MLQF: Las reglas básicas

- Por ejemplo, si una determinada tarea repetidamente no utiliza la CPU mientras espera que un dato sea ingresado por el teclado, MLFQ va a mantener su prioridad alta, así es como un proceso interactivo debería comportarse.
- Si por lo contrario, una tarea usa intensivamente por largos periodos de tiempo la CPU, MLFQ reducirá su prioridad. De esta forma MLFQ va a aprender mientras los procesos se ejecutan y entonces va a usar la historia de esa tarea para predecir su futuro comportamiento

Primer intento: ¿Cómo cambiar la prioridad ?

Se debe decidir cómo MLFQ va a cambiar el nivel de prioridad a una tarea durante toda la vida de la misma (por ende en que cola esta va a residir).

Para hacer 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 basadas en la CPU que necesitan tiempo de CPU , pero poco tiempo de respuesta*.

Primer intento: ¿Cómo cambiar la prioridad ?

A continuación e muestra un primer intento de algoritmo de ajuste de prioridades:

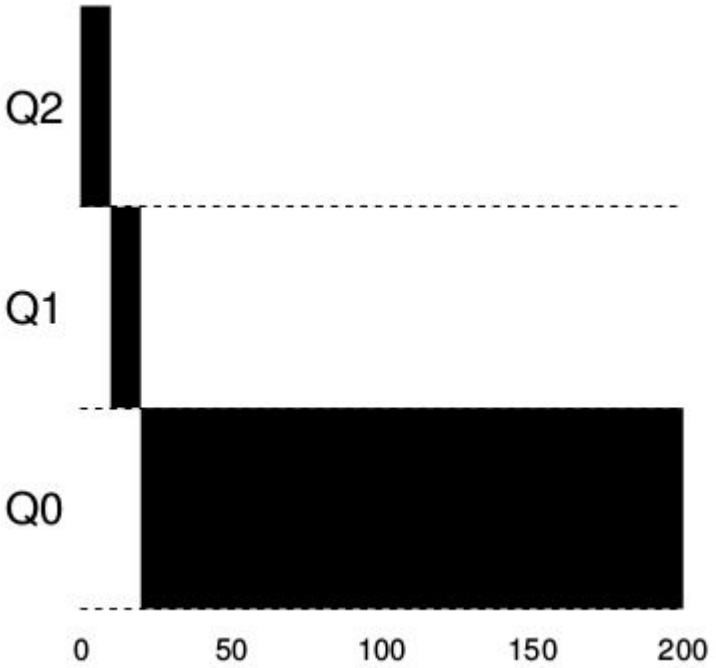
REGLA 3: Cuando una tarea entra en el sistema se pone con la mas alta prioridad

REGLA 4a: Si una tarea usa un time slice mientras se está ejecutando su prioridad se reduce de una unidad (baja la cola una unidad menor)

REGLA 4b: Si una tarea renuncia al uso de la CPU antes de un time slice completo se queda en el mismo nivel de prioridad.

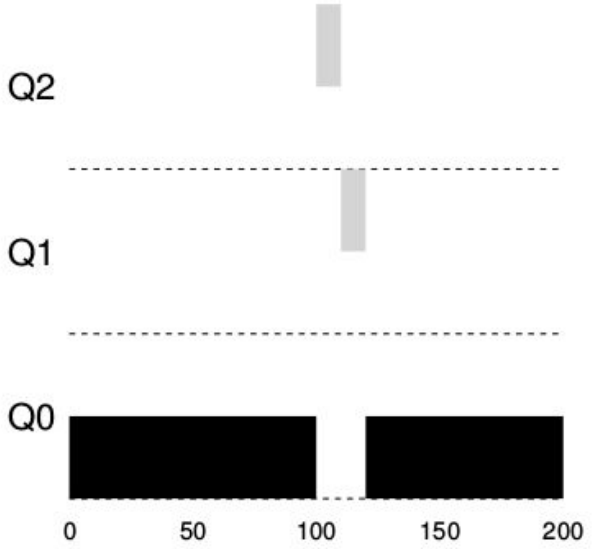
Primer intento: ¿Cómo cambiar la prioridad ?

Ejemplo 1: Una única tarea con ejecución larga.



Primer intento: ¿Cómo cambiar la prioridad ?

Ejemplo 1: Llega una tarea corta.



Primer intento: ¿Cómo cambiar la prioridad ?

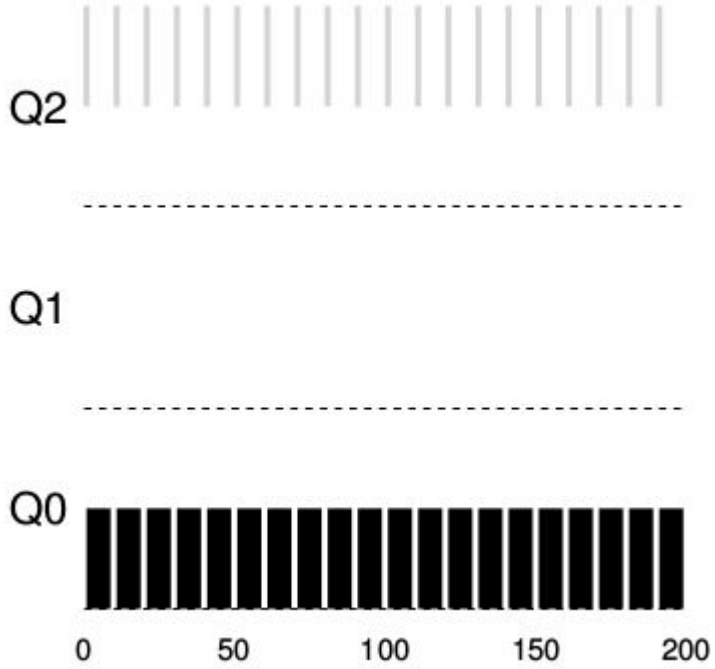
Existen 2 tareas, una de larga ejecución de CPU, A y B con una ejecución corta e interactiva. B tarda 20 milisegundos en ejecutarse.

De este ejemplo se puede ver una de las metas del algoritmo dado que no sabe si la tarea va a ser de corta o larga duración de ejecución, inicialmente asume que va a ser corta, entonces le da la mayor prioridad.

Si realmente es una tarea corta se va a ejecutar rápidamente y va a terminar, si no lo es se moverá lentamente hacia abajo en las colas de prioridad haciéndose que se parezca más a un proceso BATCH .

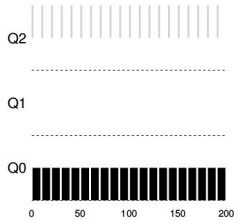
Primer intento: ¿Cómo cambiar la prioridad ?

Ejemplo 1: Que pasa con la entrada y salida.



Primer intento: ¿Cómo cambiar la prioridad ?

Como se considera en la regla 4 si la tarea renuncia al uso del procesador antes de un time slice se mantiene en el mismo nivel de prioridad. EL objetivo de esta regla es simple: si una tarea es interactiva por ejemplo entrada de datos por teclado o movimiento del mouse esta no va a requerir uso de CPU antes de que su time slice se complete en ese caso no será penalizada y mantendrá su mismo nivel de prioridad.



PROBLEMA con este Approach de MLFQ

Starvation: Si hay demasiadas tareas interactivas en el sistema se van a combinar para consumir todo el tiempo del CPU y las tareas de larga duración nunca se van a ejecutar.

Un usuario inteligente podría reescribir sus programas para obtener más tiempo de CPU por ejemplo: Antes de que termine el time slice se realiza una operación de entrada y salida entonces se va a relegar el uso de CPU haciendo esto se va a mantener la tarea en la misma cola de prioridad. Entonces la tarea puede monopolizar toda el tiempo de CPU.

Segundo Approach

¿ Cómo mejorar la prioridad? Para cambiar el problema del starvation y permitir que las tareas con larga utilización de CPU puedan progresar lo que se hace es aplicar una idea simple, se mejora la prioridad de todas las tareas en el sistema. Se agrega una nueva regla:

Regla 5: Después de cierto periodo de tiempo S, se mueven las tareas a la cola con más prioridad.

monopolizar toda el tiempo de CPU.

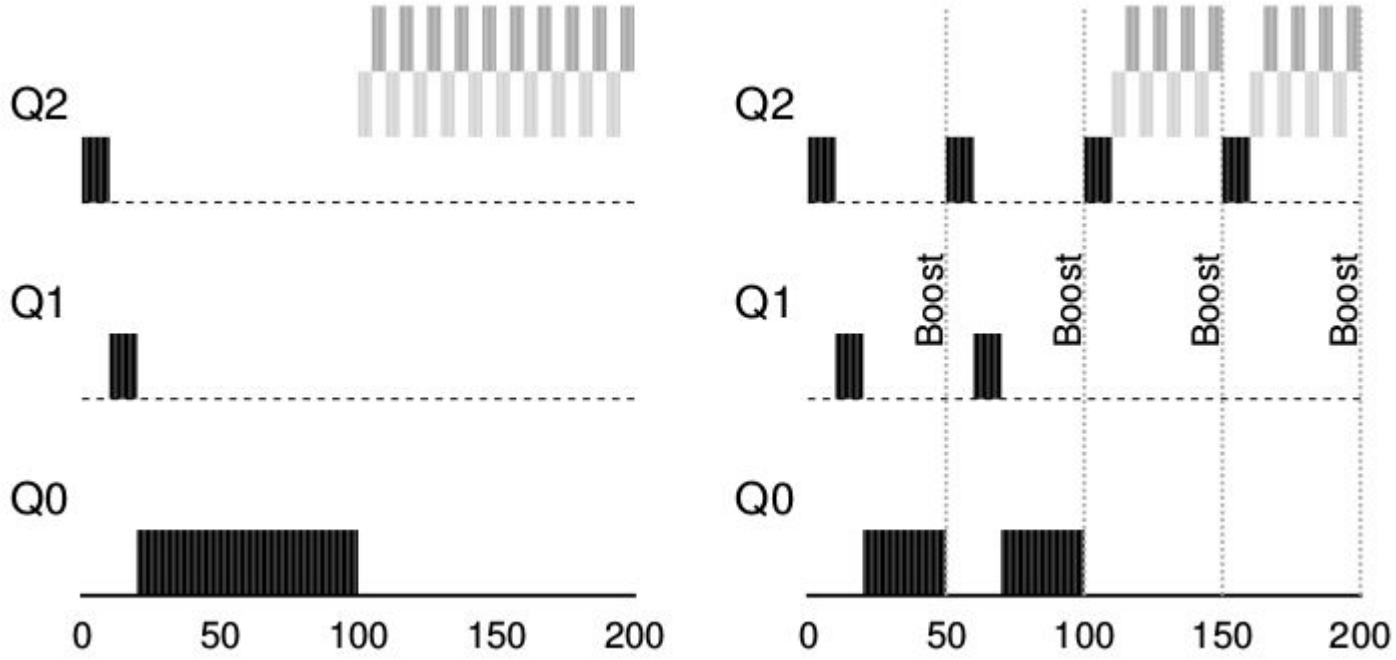
Segundo Approach

Haciendo esto se matan 2 pájaros de 1 tiro:

Se garantiza que los procesos no se van a starve: Al ubicarse en la cola tope con las otras tareas de alta prioridad estos se van a ejecutar utilizando round-robin y por ende en algún momento recibirá atención.

si un proceso que consume CPU se transforma en interactivo el planificador lo tratara como tal una vez que haya recibido el boost de prioridad.

Segundo Approach: Boost



Segundo Approach

Obviamente el agregado del periodo de tiempo **S** va a desembocar en la pregunta obvia: Cuánto 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.

Este es el caso de **S**, si el valor es demasiado alto,los procesos que requieren mucha ejecución van a caer en starvation; si se setea a **S** con valores muy pequeños las tareas interactivas no van a poder compartir adecuadamente la CPU.

Segundo Approach

Se debe solucionar otro problema: Cómo prevenir que ventajeen (gaming) al planificador.

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

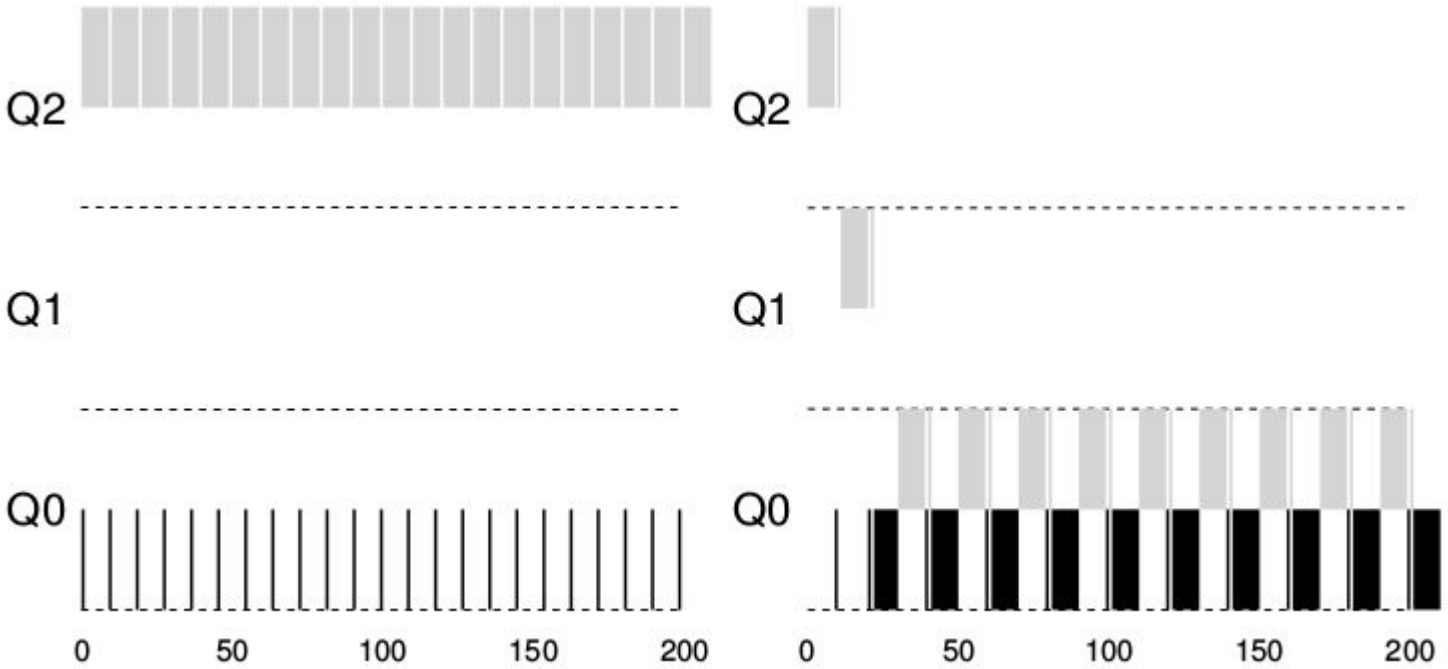
En lugar de que el planificador se **olvide** de cuanto time slice un determinado proceso utiliza en un determinado nivel el planificador debe seguir la pista desde que un proceso ha sido asignado a una cola hasta que es trasladado a una cola de distinta prioridad.

Segundo Approach

Ya sea si usa su time slice de una o en pequeños trocitos, esto no importa por ende 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: (Por ejemplo baja un nivel en la cola de prioridad)

Segundo Approach



Segundo Approach

Se vio la técnica de planificación conocida como multi-level feed back qeque (MLFQ). Se puede ver porque es llamado así, tiene un conjunto de colas de multiniveles y utiliza feed back para determinar la prioridad de una tarea dada. La historia es su guía: Poner atención como las tareas se comportan a través del tiempo y tratarlas de acuerdo a ello.

Segundo Approach

Las reglas que se utilizan son:

REGLA 1: si la prioridad (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.

REGLA 3: Cuando una tarea entra en el sistema se pone con la mas alta prioridad

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: (Por ejemplo baja un nivel en la cola de prioridad).

Regla 5: Después de cierto periodo de tiempo S, se mueven las tareas a la cola con mas prioridad.

Linux

El planificador de tareas de Linux desde su primer versión en 1991 pasando por todas hasta el kernel versión 2.4 fue sencillo y casi pedestre en lo que respecta a su diseño [LOV], cap. 4. Durante la versión 2.5 del kernel el scheduler sufrió una serie de revisiones.

Un nuevo planificador vio la luz del día llamado O(1) scheduler debido a su comportamiento algorítmico. Este introdujo un algoritmo que calculaba en tiempo constante:

- el time-slice
- las colas de ejecución por proceso

Linux

- Si bien el planificador O(1) estaba muy bueno para equipos servidores no funcionaba tan bien para procesos con interacción con usuarios procesos interactivos. En las revisiones del kernel 2.6 se introdujeron nuevas mejoras en lo que respecta a los procesos interactivos del planificador O(1), Con Kolivas, introdujo el concepto de **Rotating Starircase Deadline scheduler**, que introduce el concepto de fair scheduling.
- Una característica importante de CFS es que el mismo no otorga un determinado time slice a un proceso. Sino que este le otorga una proporción del procesador. Esta porción del tiempo de procesador que el proceso recibe es función de la carga del sistema.

Linux: Completelly Fair Scheduler (CFS)

El planificador de linux llamado **Completely Fair implementa** que se denomina **fair-share scheduling** de una forma altamente eficiente y de forma escalable.

Para lograr la meta de ser eficiente, CFS intenta gastar muy poco tiempo tomando decisiones de planificación, de dos formas :

- Por su diseño
- Debido al uso inteligente de estructuras de datos para esa tarea

Linux: Completelly Fair Scheduler (CFS)

Mientras que los planificadores tradicionales se basan alrededor del concepto de un time-slice fijo, CFS opera de forma un poco diferente. Su objetivo es sencillo: dividir de forma justa la CPU entre todos los procesos que están compitiendo por ella.

Esto lo hace mediante una simple técnica para contar llamada **virtual runtime (Vruntime)**. El vruntime no es más que el runtime (es decir el tiempo que se está ejecutando el proceso) normalizado por el número de processos runnable (se mide en nanosegundos)

Linux: Completelly Fair Scheduler (CFS)

A medida que un proceso se ejecuta este acumula vruntime. En el caso más básico cada vruntaime de un proceso se incrementa con la misma tasa, en proporción al tiempo (real) físico. Cuando una decisión de planificación ocurre, CFS seleccionará el proceso con menos vruntime para que sea el próximo en ser ejecutado.

Esto lleva a una pregunta:¿Como sabe el planificador cuando parar de ejecutar el proceso que está corriendo y correr otro proceso?

Linux: Completelly Fair Scheduler (CFS)

El punto clave aquí es que hay un punto de tensión entre performance y equitatividad

si el CFS switchea de proceso en tiempos muy pequeños estará garantizando que todos lo proceso se ejecuten a costa de perdida de performance, demasiados context switches

si CFS switchea pocas veces, la performance del scheduler es buena pero el costo está puesto del lado de la equitatividad (fairness).

La forma en que CFS maneja esta tensión es mediante varios parámetros de control.

Linux: Completelly Fair Scheduler (CFS)

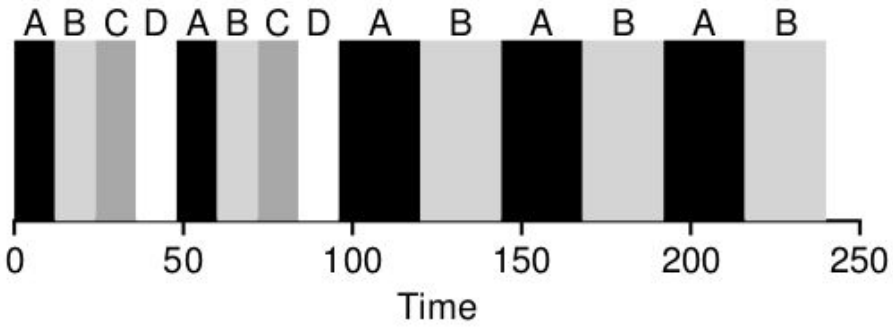
La forma en que CFS maneja esta tensión es mediante varios parámetros de control.

sched_latency: este valor determina por cuánto tiempo un proceso tiene que ejecutarse antes de considerar su switcheo. (es como un time-slice pero dinámico). Un valor típico de este parámetro es de 48 ms, CFS divide este valor por el número de procesos (**n**) ejecutándose en la CPU para determinar el time-slice de un proceso, y entonces se asegura que por ese periodo de tiempo, CFS va a ser Completamente justo.

Linux: Completelly Fair Scheduler (CFS)

Por ejemplo, si **n=4** procesos ejecutándose, CFS divide el valor de sche_latency por **n** asignándole a cada proceso 12 ms. CFS planifica el primer job y lo ejecuta hasta que ha utilizado sus 12 ms de (virtual) runtime, y luego chequea si hay algún otro proceso con menos vruntime, si lo hay switchea a este.

Bueno, pero si hay muchos procesos ejecutándose esto no llevaría a muchos context swtiches y por ende pequeños time slices? Sí.



Linux: Completelly Fair Scheduler (CFS)

Para lidiar con este problema se introduce otro parámetro llamado **min_granularity**, que normalmente se setea con el valor de 6 ms. Entonces en CFS nunca se daría el time-slice de un proceso por debajo de ese número, por ende con esto se asegura que no haya overhead por context switch.

CFS utiliza una interrupción periódica de tiempo, lo que significa que sólo puede tomar decisiones en periodos de tiempos fijos(1ms)

Linux: Weighting (Niceness)

CFS tiene control sobre las prioridades de los procesos, de forma tal que los usuario y administradores pueden asignar más CPU a un determinado proceso. Esto se hace con un mecanismo cálcico de unix llamado nivel de proceso nice , este valor va de -20 a +19, con un valor por defecto de 0. Con una característica un poco extraña: los valores positivos de nice implican una prioridad más baja, y los valores negativos de nice implican una prioridad más alta.

Linux: Weighting (Niceness)

CFS mapea el nice value con un peso como se muestra :

```
static const int prio_to_weight[40] = {
/* -20 */     88761,   71755,   56483,   46273,   36291,
/* -15 */     29154,   23254,   18705,   14949,   11916,
/* -10 */      9548,    7620,    6100,    4904,    3906,
/*  -5 */      3121,    2501,    1991,    1586,    1277,
/*   0 */      1024,     820,     655,     526,     423,
/*   5 */       335,     272,     215,     172,     137,
/*  10 */       110,      87,      70,      56,      45,
/*  15 */        36,      29,      23,      18,      15,
};
```

Linux: Time-Slice

Estos pesos permite calcular efectivamente el time slice para cada proceso (como se hizo antes) peeero teniendo en cuenta las distintas prioridades para cada proceso:

$$time\_slice_k = \frac{weight_k}{\sum_{n=o}^{n-1} weight_i} * sched\_latency$$

Linux: Weighting (Niceness)

Dado que:

$weight_A$=3121 (para el proceso A con nice=−5)

$weight_B$=1024 (para el proceso B con nice=0)

$$time\_slice_i = T \times (weight_i / \sum weights)$$

Dado que en tu escenario solo tienes dos procesos, la fórmula se simplifica:

$$time\_slice_A = T \times (3121/(3121+1024))$$

$$time\_slice_B = T \times (1024/(3121+1024))$$

$$time\_slice_A = 24 \times (3121 + 10243121)$$

$$time\_slice_B = 24 \times (3121 + 10241024)$$

T=sched_latency ---> 48/2=24

$time\_slice_A$≈18.06 (redondeando a dos decimales)

$time\_slice_B$≈5.94 (redondeando a dos decimales)

Linux: Vruntime

Con esto se debe generalizar el cálculo de vruntime:

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} * runtime_i$$

Este se calcula tomando el tiempo de ejecución real que el proceso$_i$ ha acumulado (runtime$_i$) y lo escala de manera inversa según el peso del proceso.

Linux: Vruntime

Un aspecto inteligente de la construcción de la tabla de pesos anterior es que la tabla conserva las proporciones de ratio de la CPU cuando la diferencia en valores de nice es constante.

Por ejemplo, si el proceso A en su lugar tenía un buen valor de 5 (no -5), y el proceso B tenía un buen valor de 10 (no 0), CFS los programaría exactamente de la misma manera que antes. Probarlo el cálculo usted mismo para ver por qué.

Linux: Delta Vruntime

Para Calcular el delta del vruntime se aplica esta formula:

$$delta\_vruntime = time\_slice \times \frac{weight}{NICE\_ZERO\_LOAD}$$

donde:

time_slice es la cantidad de tiempo que el proceso ha sido ejecutado.

NICE_ZERO_LOAD es una constante que representa la carga por defecto de un proceso con un valor nice de 0. Esta constante suele tener un valor de 1024.

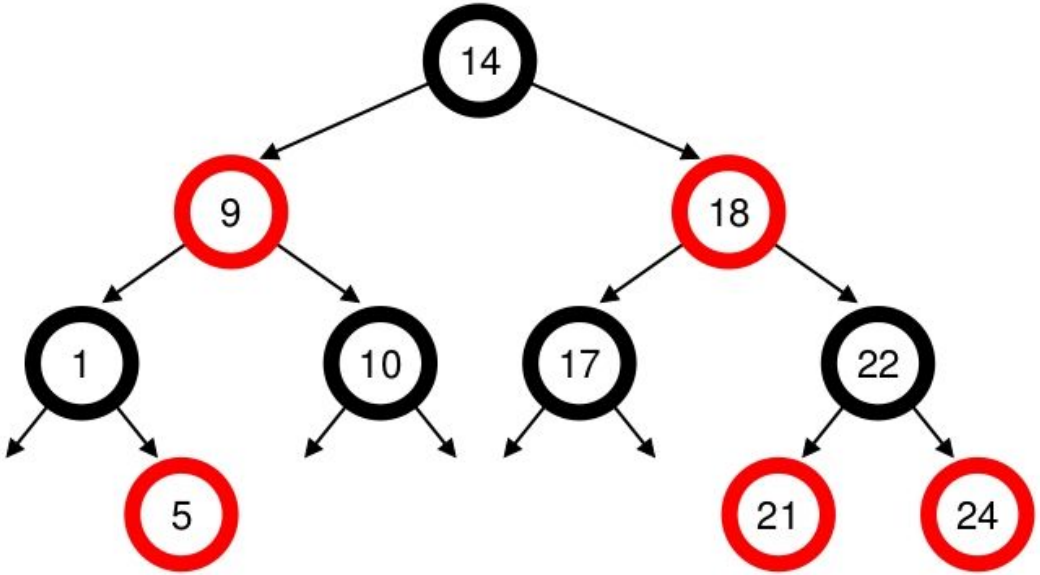
Weight es el peso del proceso, que está determinado por su valor nice.

Linux: Arbol Rojo Negro

Uno de los focos de eficiencia del CFS está en la implementación de las políticas anteriores. Pero también en una buena selección del tipo de dato cuando el planificador debe encontrar el próximo job a ser ejecutado.

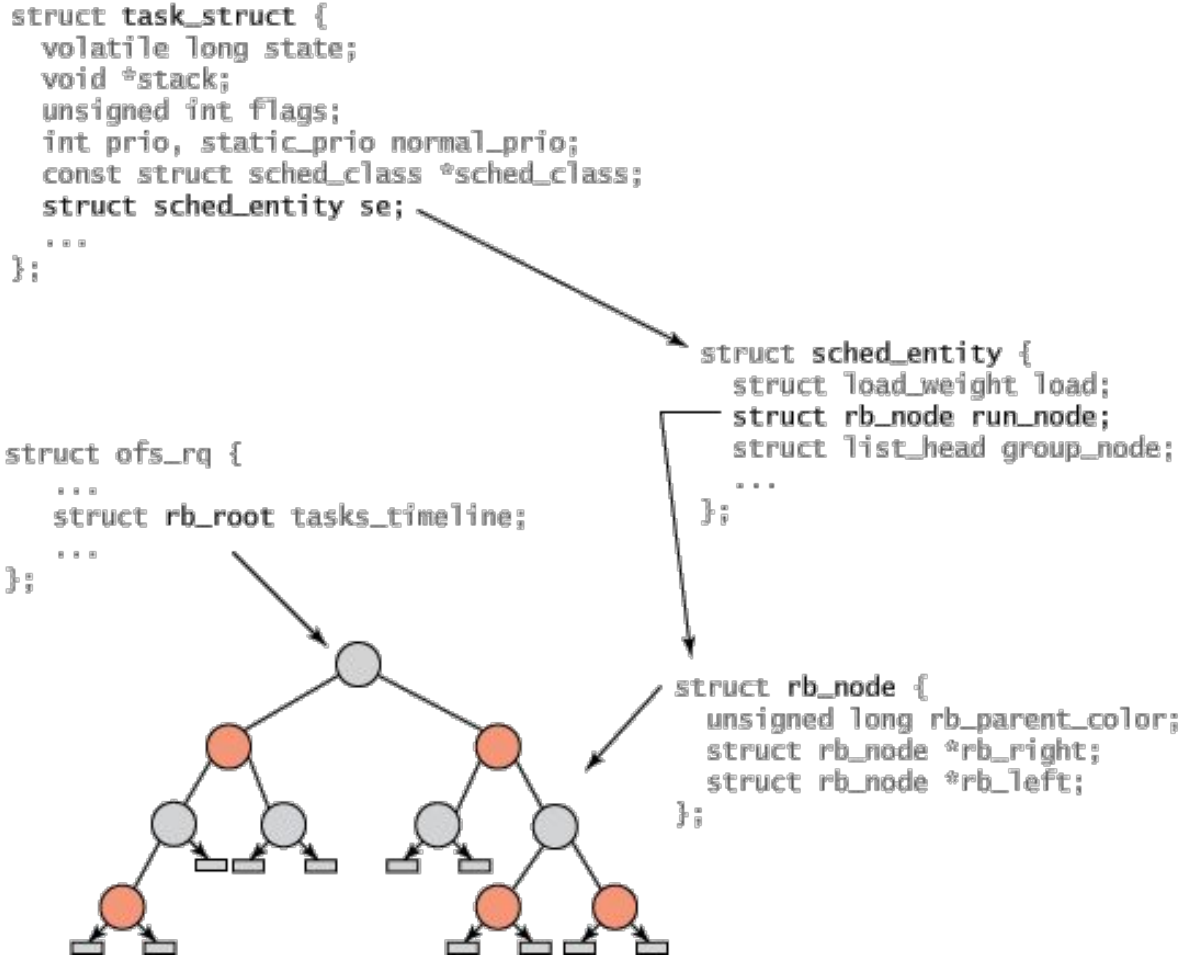
- Las listas no escalan bien O(n)
- Los árboles sí, en este caso los árboles Rojo-Negro O(logn)

Linux: Arbol Rojo Negro



Linux: Arbol Rojo Negro

Para tener una idea cerrada de donde aparece el árbol rojo y negro dentro del kernel de linux a partir de la versión del kernel 2.6.0, ver la siguiente imagen:



Linux: El Algoritmo:

Cuando el scheduler es invocado para correr un nuevo proceso, la forma en que el scheduler actúa es la siguiente:

1. El nodo más a la izquierda del árbol de planificación es es elegido (ya que tiene el tiempo de ejecución más bajo), y es enviado a ejecutarse.
2. Si el proceso simplemente completa su ejecución, este es eliminado del sistema y del árbol de planificación.
3. Si el proceso alcanza su máximo tiempo de ejecución o de otra forma se para la ejecución del mismo voluntariamente o vía una interrupción) este es reinsertado en el árbol de planificación basado en su nuevo tiempo de ejecución (vruntime).
4. El nuevo nodo que se encuentre más a la izquierda del árbol será ahora el seleccionado, repitiéndose así la iteración.