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

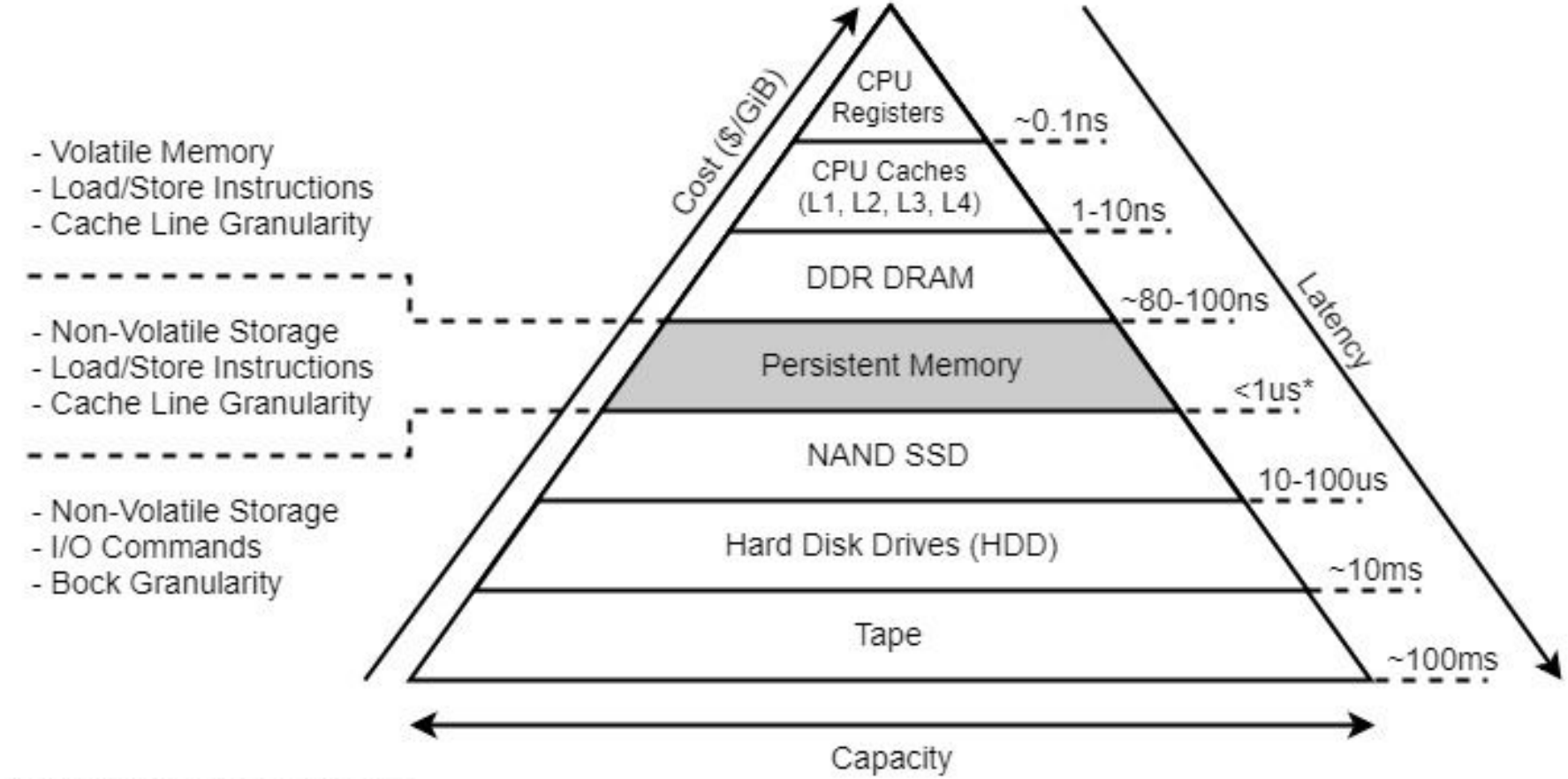
Entonces, el cálculo es más realista si se supone que un proceso gasta una fracción p, bloqueado en E/S. De esta manera, si tenemos n procesos esperando para hacer operaciones de entrada y salida, la probabilidad de que los n procesos estén haciendo E/S es p_n

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.

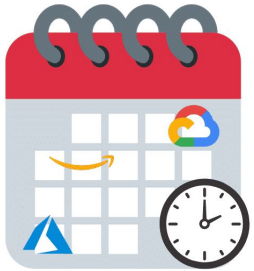
Si se supone que se tienen 10 procesos, entonces la fórmula cambia a $1-0.810=0.89$ el 89% de utilización, aquí es donde se ve la IMPORTANCIA de la Multiprogramación.



(*) 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:

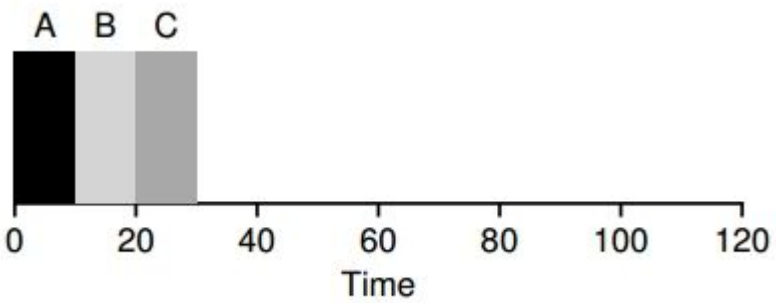
1. Es simple.

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.

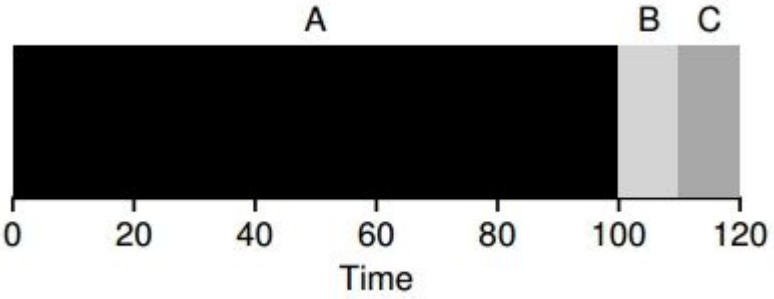


Si bien llegan todos al mismo tiempo llegaron con un insignificante retraso de forma tal que llegó A, B y C. S i se asume que todos tardan 10 segundos en ejecutarse… ¿cuánto es el Taround?

(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?

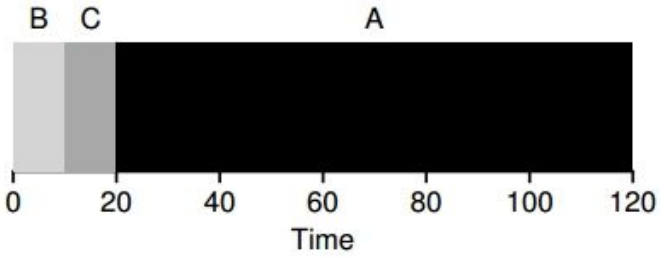
(100+110+120)/3=110

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.



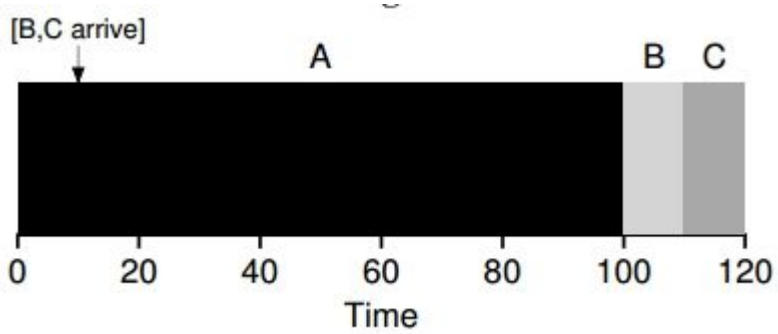
En el mismo caso d n el sencillo hecho de ejecutar B, C y A en ese orden:

(10+20+120)/3=50

Shortest Job First (SJF)

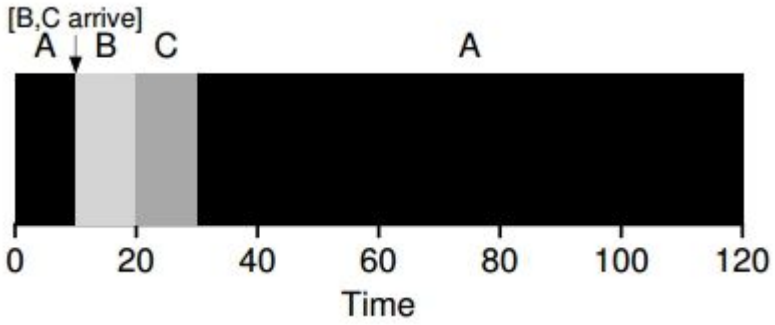
Utilizando SJF se obtuvo una significativa mejora… pero con las suposiciones iniciales que son muy poco realistas. Si se relaja la suposición 2, en la cual no todos los procesos llegan al mismo tiempo, por ejemplo llega el proceso A y a los 10 segundos llegan el proceso B y el proceso C. ¿Cómo sería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

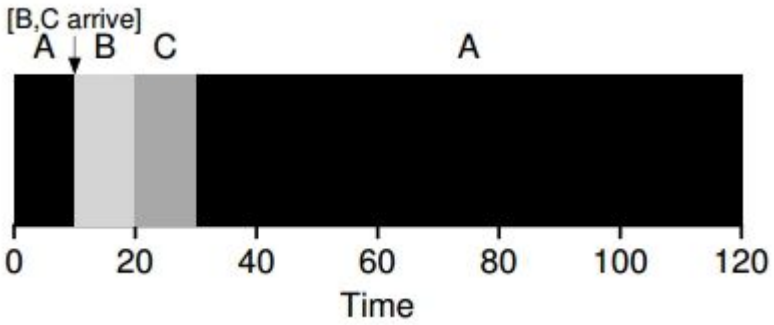


El caso anterior el de SFJ es una p

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:

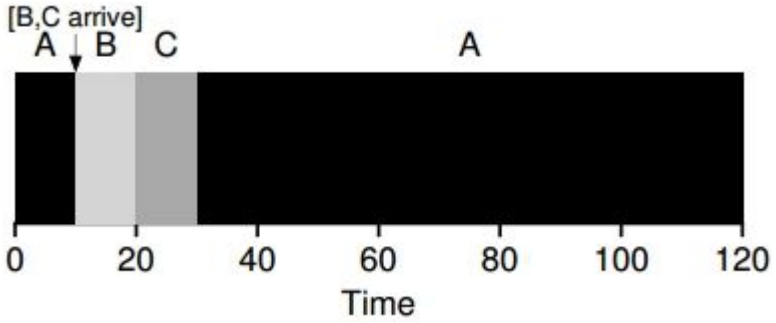
$$T_{response} = T_{firstrun} - T_{arrival}$$

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)

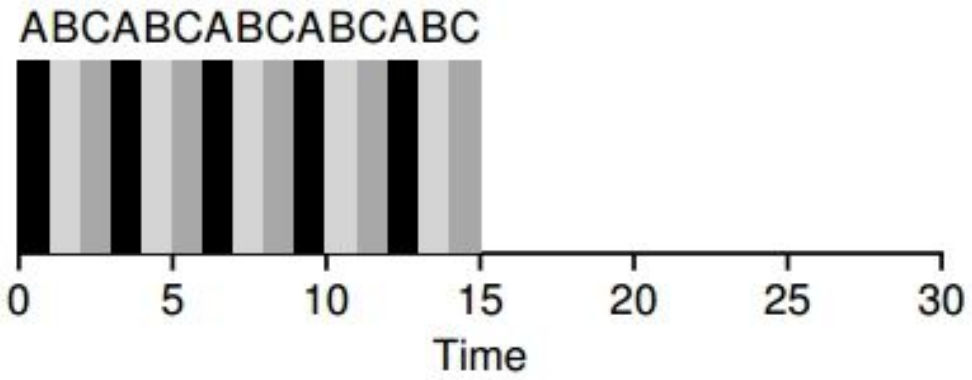
En promedio el $T_{response}$ es de 3.33 seg. Entonces

¿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?

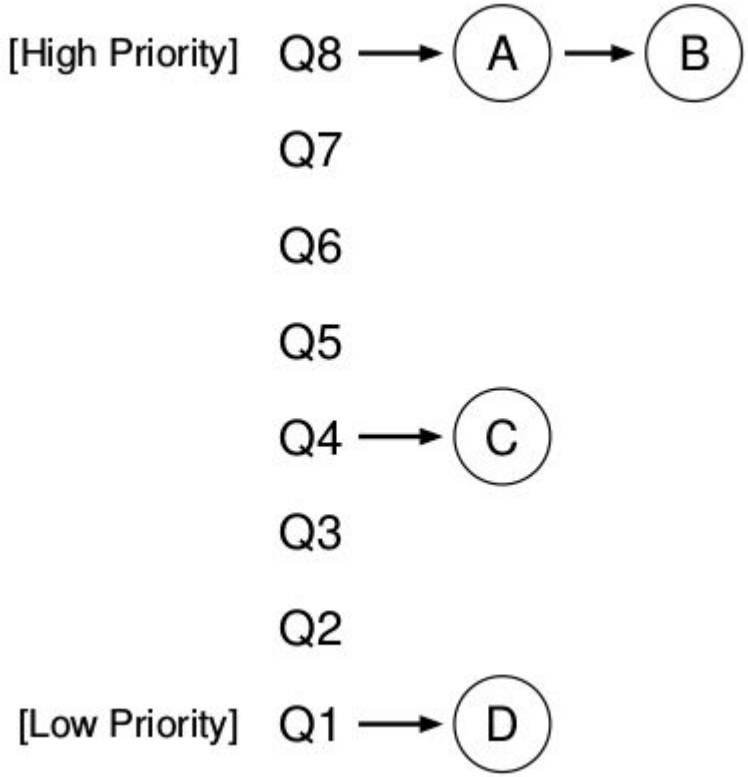
MLQF: Las reglas básicas

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:

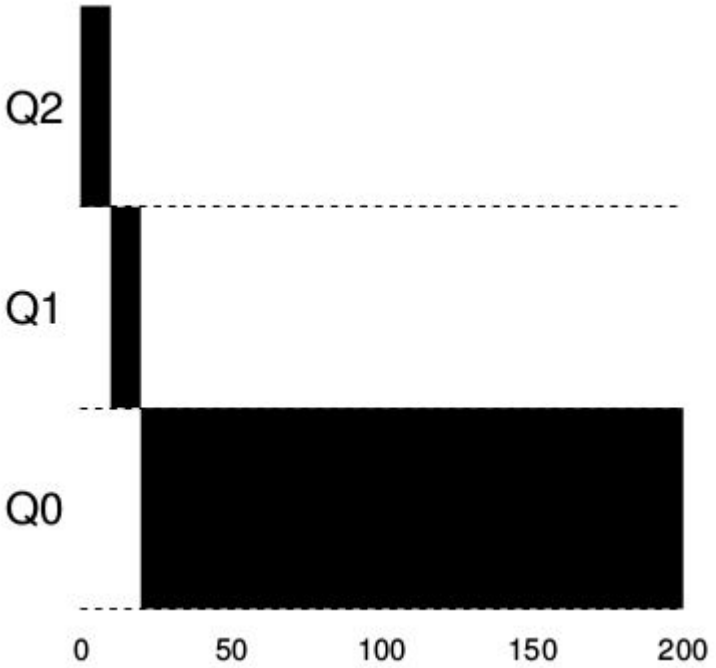
REGLA 3: Cuando una tarea entra en el sistema se pone con la mas alta prioridad

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.

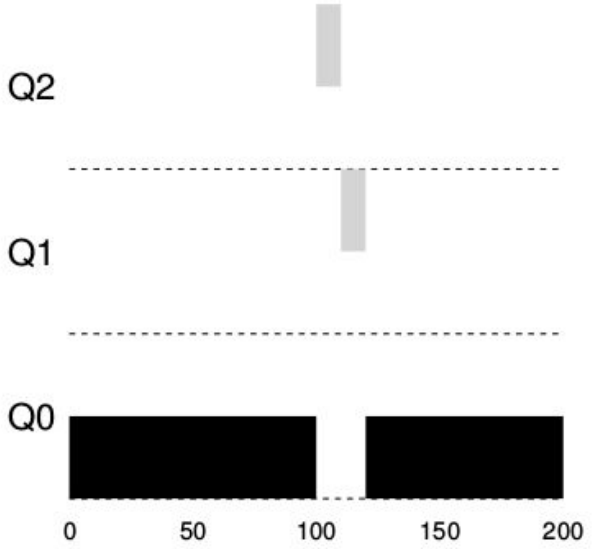
Primer intento: ¿Cómo cambiar la prioridad ?

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 ?

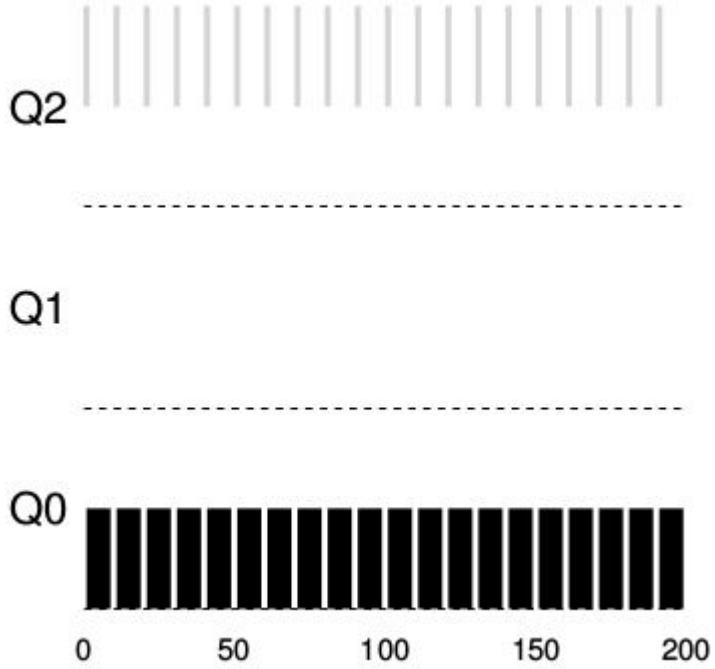
Existen 2 tareas, una de larga ejecución de CPU, A y B con una ejecución corta e interactiva. B tarda 20 milisegundos en ejecutarse.

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 .

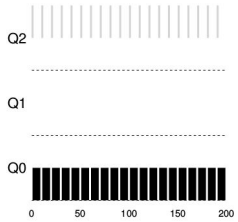
Primer intento: ¿Cómo cambiar la prioridad ?

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.

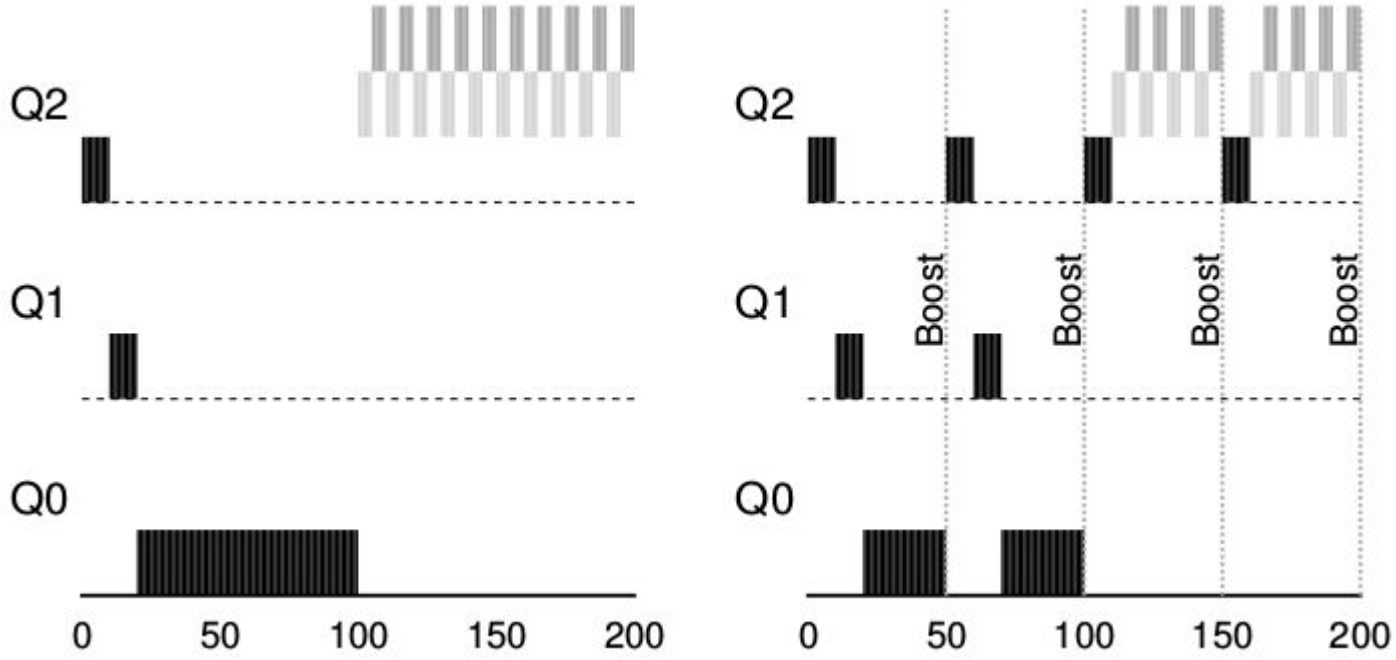
Segundo Approach

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.

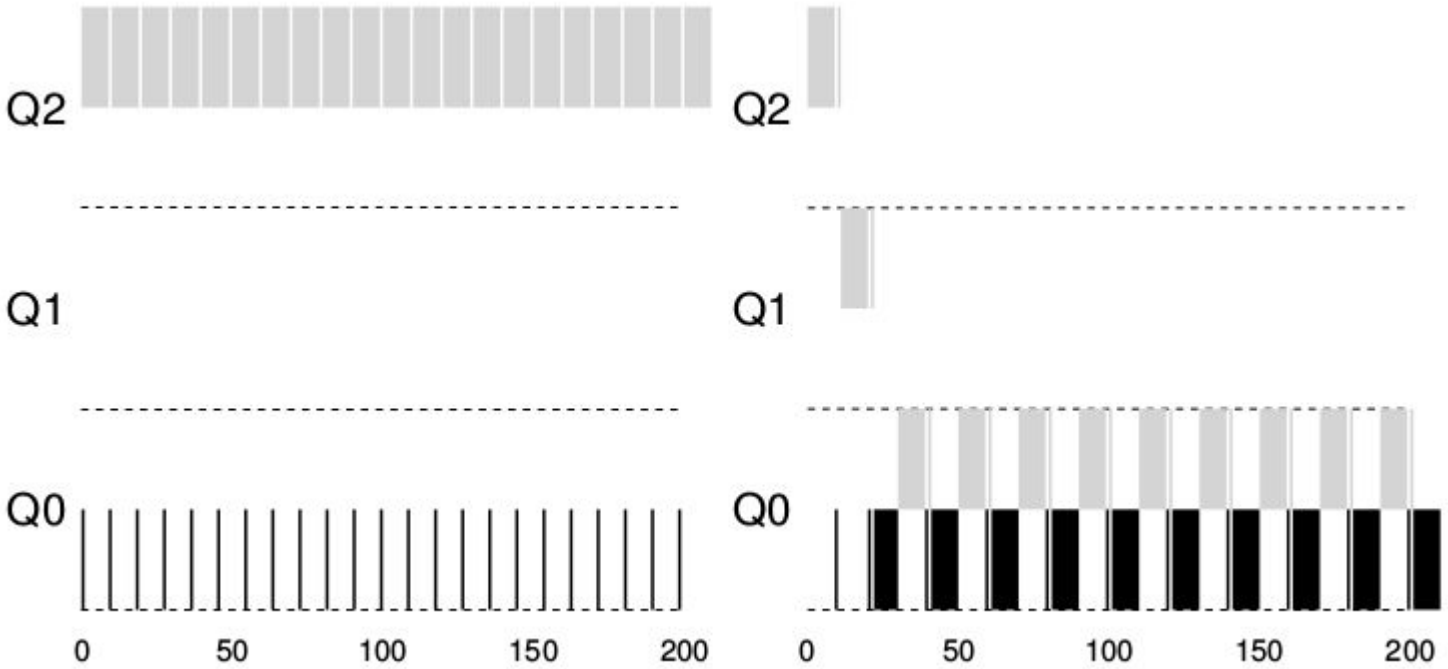
En lugar de que el planificador se **olvide** de cuanto time slice un determinado proceso utiliza en un determinado nivel el planificador debe seguir la pista desde que un proceso ha sido asignado a una cola hasta que es trasladado a una cola de distinta prioridad.

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 más 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

Linux

Linux 2.4 y anteriores: En las primeras versiones de Linux, el scheduler era bastante básico, conocido como un scheduler de tipo **"round-robin"** que simplemente rotaba a través de los procesos, asignándoles el CPU por un tiempo fijo antes de moverse al siguiente proceso. Este enfoque era simple pero no manejaba eficientemente la carga cuando los sistemas empezaron a ser más demandantes y con múltiples procesos.

Linux 2.6 (O(1) Scheduler): Introducido en la versión 2.5 y estabilizado en la 2.6, **el scheduler O(1)** fue diseñado para ser extremadamente eficiente con complejidad constante, lo que significa que el tiempo que toma tomar decisiones de scheduling es constante, sin importar cuántos procesos estén corriendo. Esto fue un gran paso adelante en términos de **escalabilidad** y **rendimiento** en sistemas con muchos procesos.

Linux

Completely Fair Scheduler (CFS) en Linux 2.6.23: En 2007, con la versión 2.6.23, Linux introdujo el Completely Fair Scheduler, diseñado por **Ingo Molnar**. Este scheduler está basado en el concepto de **planificación proporcional (fair scheduling)**, donde se intenta dar a cada proceso una cantidad de tiempo del **CPU proporcional a su prioridad.** CFS usa una estructura de datos de tipo **red-black tree** para mantener un seguimiento del tiempo de ejecución de cada proceso, permitiendo un manejo más eficiente y justo del tiempo de CPU, especialmente en sistemas con muchos núcleos.

Cambios en versiones más recientes: Desde la introducción del CFS, han habido varias mejoras y ajustes menores en las versiones subsecuentes para optimizar el rendimiento y la capacidad de respuesta del scheduler. Por ejemplo, mejoras en la balance de carga entre CPUs, soporte para nuevas tecnologías de hardware, y optimizaciones para sistemas de tiempo real.

Linux

Cgroups y control de recursos: A lo largo del tiempo, Linux ha integrado mejor el manejo de grupos de control (cgroups), que permiten al scheduler no solo planificar procesos individualmente, sino también gestionar recursos entre grupos de procesos, permitiendo un control más fino sobre el uso de recursos del sistema.

Linux

Linux 1.2: circular queue w/ round-robin policy.

Linux 2.2: introduced scheduling classes (real-time, non-real-time)

```
/* Scheduling Policies
*/
#define SCHED_OTHER 0 // Normal user tasks (default)
#define SCHED_FIFO 1 // RT: Will almost never be preempted
#define SCHED_RR 2 // RT: Prioritized RR queues
```

Linux 2.4: O(N) scheduler.

Epochs → slices: when blocked before the slice ends, half of the remaining slice is added in the next epoch.

-Simple.

-Lacked scalability.

-Weak for real-time systems.

Linux

Linux 2.6: O(1) scheduler

- Tasks are indexed according to their priority [0,139]
- Real-time [0, 99]
- Non-real-time [100, 139]

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.

Round-Robin: El método básico de planificación era round-robin, que asigna a cada proceso un intervalo de tiempo fijo, llamado quantum de tiempo, durante el cual el proceso puede ejecutar. Una vez que el quantum de tiempo de un proceso se agota, el scheduler pone al proceso en espera y pasa al siguiente proceso en la cola.

Multinivel de Feedback Queue: Linux utilizaba un enfoque de múltiples colas para manejar procesos con diferentes prioridades. Los procesos se organizaban en diferentes colas basadas en sus prioridades y comportamiento (por ejemplo, si un proceso utilizaba mucho el CPU o si cedía frecuentemente el CPU). Un proceso podía moverse entre colas si su comportamiento cambiaba, lo que permitía al sistema adaptarse dinámicamente y dar más tiempo de CPU a los procesos que parecían necesitarlo más urgentemente.

Linux

Prioridades y preemption: Aunque el núcleo Linux de la serie 2.4 no era completamente preemptivo en su manejo de procesos en espacio de usuario, sí tenía soporte básico para la preemption en el manejo de interrupciones y algunas operaciones críticas del kernel. Esto significa que ciertas tareas críticas podían interrumpir otras menos críticas para mejorar la capacidad de respuesta del sistema. Sin embargo, en el espacio de usuario, un proceso en ejecución no sería necesariamente interrumpido hasta que finalizara su quantum de tiempo, a menos que esperara por I/O u otra interrupción.

Linux

El **Completely Fair Scheduler (CFS)** de Linux es un algoritmo avanzado de planificación que reemplazó a los schedulers más antiguos basados en **time slices fijos y prioridades estáticas**. Introducido en la versión 2.6.23 del kernel de Linux por **Ingo Molnar**, el CFS tiene como objetivo principal proporcionar un reparto equitativo del tiempo de CPU entre los procesos.

Linux: Principios Básicos del CFS

Equidad: El CFS busca ofrecer a cada proceso una porción justa del CPU, basándose en el concepto de equidad. No se asignan cuotas fijas de tiempo (time slices), sino que se intenta que cada proceso reciba una porción del tiempo de CPU proporcional a su peso de prioridad.

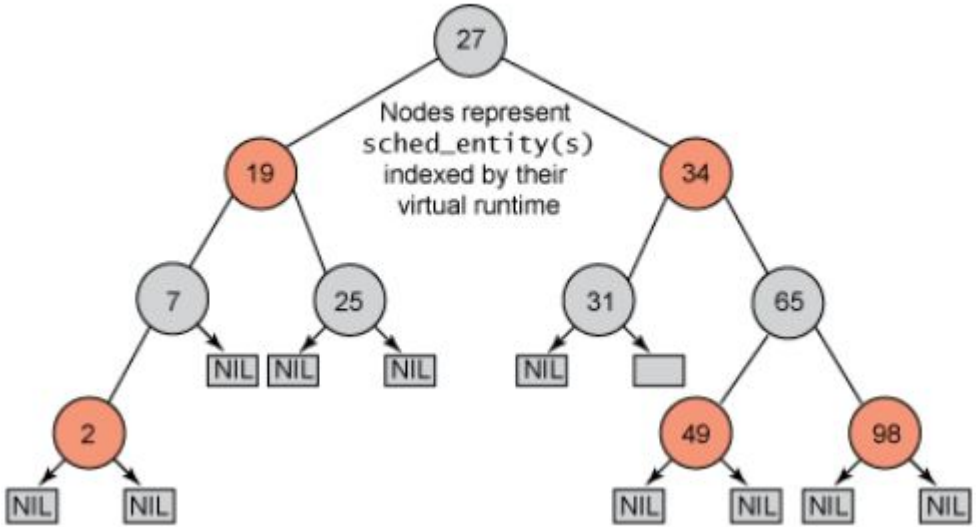
Virtual Runtime (vruntime): Cada proceso tiene ***asociado un contador llamado vruntime, que representa la cantidad de tiempo que el proceso ha estado ejecutándose en el CPU***. El vruntime se incrementa con cada tick del reloj mientras el proceso está en ejecución. ***Los procesos con menor vruntime son los que tienen mayor prioridad para ser ejecutados.***

Pesos de Prioridad: Los procesos en Linux pueden tener diferentes prioridades, que en el CFS se traducen en pesos. Un **peso más alto implica una mayor prioridad**, lo que significa que el vruntime de un proceso con alta prioridad crece más lentamente que el de uno con baja prioridad.

Linux: Funcionamiento Detallado

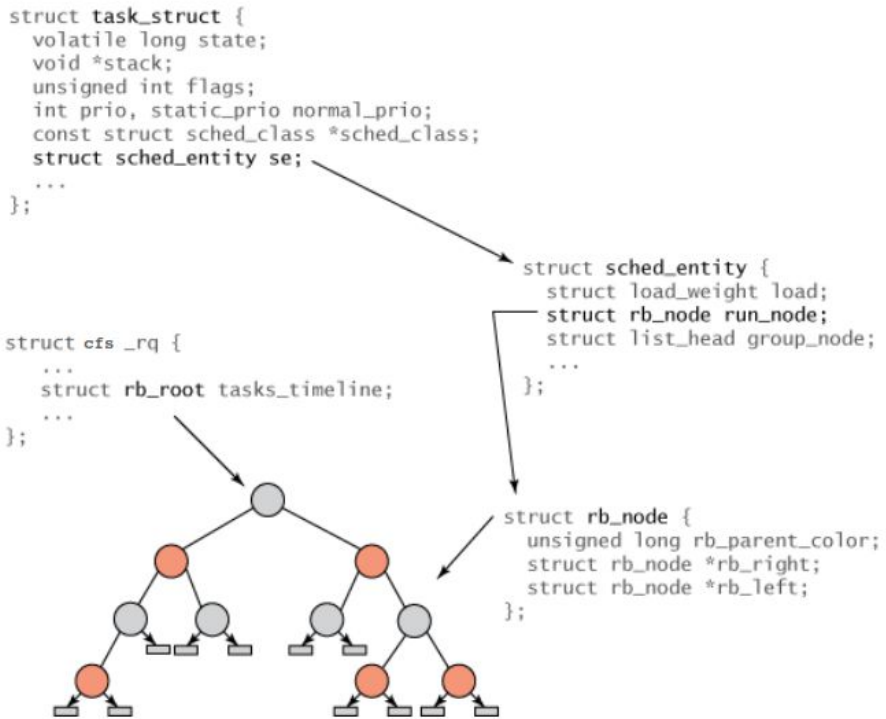
1. **Selección del Proceso:** El scheduler siempre intenta ejecutar el proceso con el **menor vruntime**. Esto asegura que todos los procesos tengan la oportunidad de ejecutarse de manera justa, proporcionando un tiempo de ejecución proporcional a su peso.
2. **Uso de Red-Black Trees**: Para organizar eficientemente los procesos por su vruntime, el CFS utiliza una estructura de datos de **árbol rojo-negro**. Este tipo de árbol permite inserciones, eliminaciones y búsquedas en tiempo logarítmico, lo que es crucial para mantener el rendimiento del scheduler con un gran número de procesos.
3. **Sleeping and Waking Up**: Cuando un proceso se bloquea, por ejemplo, esperando por I/O, se **elimina del árbol rojo-negro**. Cuando se despierta, se vuelve a insertar en el árbol con su vruntime ajustado si es necesario, de manera que no sea penalizado por el tiempo que estuvo bloqueado.

Linux: Arbol Rojo Negro



- O(1) access to leftmost node
- (lowest virtual time).
- O(log n) insert
- O(log n) delete
- • self-balancing

Linux: Arbol Rojo Negro



Linux: Funcionamiento Detallado

1. **Balance de Carga:** El CFS también se encarga de equilibrar la carga de trabajo entre múltiples CPUs. Esto se logra **migrando procesos** de una CPU a otra si se detecta que una CPU está sobrecargada mientras otra está inactiva o menos cargada.

2. **Soporte para Multi-threading**: CFS maneja los hilos de ejecución (threads) de manera similar a como maneja los procesos. Cada hilo es tratado como un proceso separado con su propio vruntime.

Linux: vruntime

El vruntime en el Completely Fair Scheduler (CFS) de Linux es una medida fundamental para determinar cuánto tiempo de CPU se ha asignado a cada proceso. El cálculo del vruntime es esencial para mantener la equidad en el acceso al procesador entre diferentes procesos. El vruntime ***se calcula en función del tiempo que un proceso ha ejecutado en el CPU, ajustado por su prioridad***. La fórmula general para actualizar el vruntime de un proceso es:

$$\text{vruntime} += \frac{\text{delta\_exec}}{\text{weight}} \times \text{load\_weight}$$

delta_exec es la cantidad de tiempo que el proceso ha estado ejecutando desde la última actualización.
weight es el peso asociado a la prioridad del proceso.
l**oad_weight** es el peso de carga del sistema o de la CPU en ese momento.

Linux: Constantes y Factores en el CFS

Pesos de Prioridad (Weight): Cada proceso tiene un peso asignado basado en su prioridad de nice, que puede variar de -20 (más favorable) a 19 (menos favorable). Estos valores de nice se mapean a un peso específico utilizando una tabla predefinida en el kernel. Los pesos ayudan a determinar ***cuán rápido un proceso acumula vruntime en comparación con otros procesos***.

Granularidad del Scheduler (sched_latency): Este es un valor que define el período de tiempo durante el cual el scheduler intenta que todos los procesos ejecutables se ejecuten al menos una vez. Por defecto, este período es de alrededor de 48 ms en sistemas con un número considerable de tareas ejecutándose. Este valor se ajusta según el número de tareas activas para mantener el rendimiento del scheduler.

Linux: Constantes y Factores en el CFS

Mínimo tiempo de ejecución garantizado (min_granularity): Este valor define el **mínimo tiempo de ejecución que un proceso debería tener antes de ser desalojado del CPU**. Esto ayuda a evitar costos excesivos de conmutación de contexto en sistemas muy cargados. El tiempo es típicamente una fracción del sched_latency.

Normalización de tiempos: Para garantizar que los cálculos se mantengan dentro de límites manejables y evitar el desbordamiento de enteros, los tiempos de ejecución pueden ser normalizados periódicamente. Esto implica ajustar los vruntime de todos los procesos, reduciendo todos los valores en función del proceso con el menor vruntime.

Linux

El parámetro sched_latency, conocido también como latencia del scheduler, es un componente crucial del Completely Fair Scheduler (CFS) en el kernel de Linux. Su función principal es definir el período de tiempo en el que el scheduler intenta que todos los procesos ejecutables reciban al menos un poco de tiempo de CPU. Este concepto es fundamental para garantizar la equidad en el reparto de recursos del procesador entre los procesos.

Linux: Funcionamiento de sched_latency

Definición del Período de Latencia: El sched_latency representa la ventana de tiempo total durante la cual el scheduler espera poder ejecutar cada tarea ejecutable al menos una vez. Por defecto, el valor de sched_latency puede ser de 48 milisegundos, pero este valor puede ajustarse automáticamente por el sistema dependiendo del número de tareas activas.

Adaptabilidad según el número de tareas: En sistemas **con muchos procesos**, mantener un sched_latency bajo puede llevar a un **alto overhead** debido a cambios de contexto frecuentes. Por lo tanto, **el scheduler ajusta este valor en función del número de tareas ejecutables**. Si hay más tareas que el número ideal que puede manejar dentro del período de latencia estándar, el sched_latency aumentará para asegurar que todas las tareas reciban atención sin incrementar desproporcionadamente los cambios de contexto.

Linux: Funcionamiento de sched_latency

Interacción con min_granularity: El sched_latency trabaja en conjunto con otro parámetro llamado **min_granularity**. Este último establece el tiempo mínimo que un proceso debe ejecutarse antes de ser desalojado del CPU, para evitar que los procesos sean interrumpidos demasiado rápidamente lo cual podría llevar a un aumento en el overhead por cambios de contexto. min_granularity suele ser una fracción del sched_latency, típicamente alrededor del 10-20%.

Ejemplo de Ajuste de sched_latency

Supongamos que un sistema tiene 100 tareas activas y un sched_latency estándar de 48 ms. Si el número de tareas supera el valor ideal que el scheduler puede manejar efectivamente dentro de ese período, digamos 25 tareas, el sistema puede decidir duplicar el sched_latency a 96 ms para permitir que todas las tareas sean planificadas adecuadamente sin aumentar demasiado la frecuencia de cambio de contexto.

Linux: nice

El valor **nice** es una característica del sistema operativo Linux que se usa para ajustar la prioridad de los procesos en cuanto a su planificación para el uso del CPU. En esencia, el valor nice permite a los usuarios manipular la prioridad de un proceso de manera que los procesos con una mayor prioridad obtengan más tiempo de CPU comparado con los que tienen una menor prioridad.

Linux: Funcionamiento del valor nice

Rango de valores: El valor nice puede variar de -20 hasta 19. Un valor nice de -20 es el más alto nivel de prioridad (menos "nice", es decir, menos amable con los demás procesos), y 19 es el nivel más bajo (más "nice", más amable). Por defecto, los procesos suelen iniciarse con un valor nice de 0.

Asignación de CPU: Un valor nice bajo (más cercano a -20) hace que el proceso sea más "egoísta", obteniendo más tiempo de CPU. Por el contrario, un valor nice alto (más cercano a 19) hace que el proceso sea más "generoso", cediendo el tiempo de CPU a otros procesos.

Modificación del valor nice: Los usuarios pueden cambiar el valor nice de un proceso para afectar su prioridad de planificación. Esto se hace generalmente a través de la línea de comandos con herramientas como nice y renice. Por ejemplo, para iniciar un proceso con un valor nice específico, se utiliza el comando nice -n [valor] [comando]. Para cambiar el valor nice de un proceso ya en ejecución, se usa renice [nuevo_valor] -p [pid].

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: Esquema del Funcionamiento del CFS

Inicialización

Cada proceso recibe un valor vruntime inicial, que es cero cuando el proceso es creado.

Cálculo de Pesos

Basado en el valor nice de cada proceso, se asigna un peso específico.

La tabla de mapeo de nice a pesos determina cuán rápido se acumula vruntime para un proceso dado.

Gestión de vruntime

El vruntime de un proceso se incrementa en función del tiempo que el proceso pasa ejecutándose en el CPU.

La tasa de incremento se ajusta por el peso del proceso: procesos con mayor peso (menos nice) acumulan vruntim más lentamente.

Uso de Red-Black Tree

Todos los procesos listos para ejecutarse son almacenados en un árbol rojo-negro, organizados por su vruntime.

El proceso con el menor vruntime se encuentra siempre el más a la izquierda de la raíz del árbol y es seleccionado para ejecutarse.

Linux: Esquema del Funcionamiento del CFS

Selección de Procesos

El scheduler selecciona el proceso con el menor vruntime del árbol para ejecución.

Este proceso se ejecuta hasta que su vruntime ya no es el mínimo, o hasta que se bloquea o termina su quantum de ejecución.

Preemptividad y Voluntariedad

Si un proceso se bloquea esperando I/O, se retira del árbol temporalmente.

Cuando un proceso despierta o es creado, se calcula su vruntime ajustado y se inserta en el árbol.

Ajuste Dinámico

sched_latency y min_granularity se utilizan para ajustar la cantidad de tiempo que cada proceso puede ejecutarse.

En sistemas con muchos procesos, estos valores se ajustan para balancear equidad y eficiencia.

Balanceo de Carga entre CPUs

El CFS también maneja la distribución de procesos entre múltiples CPUs.

Los procesos pueden ser migrados entre CPUs para optimizar la carga y minimizar la latencia.

Linux: Ejemplo

Detalles de los Procesos y Valores Nice

Proceso A: nice = 0
Proceso B: nice = -5
Proceso C: nice = 10
Proceso D: nice = 0
Proceso E: nice = -10

Linux: Cálculo de Pesos

El peso de un proceso se basa en su valor nice. En Linux, los pesos están predefinidos y se pueden ver en el código fuente del kernel. Para este ejemplo, vamos a suponer algunos valores hipotéticos para los pesos:

Valor nice de -10 (Proceso E) corresponde a un peso de 1024.

Valor nice de -5 (Proceso B) corresponde a un peso de 512.

Valor nice de 0 (Proceso A y D) corresponde a un peso de 256.

Valor nice de 10 (Proceso C) corresponde a un peso de 128.

Linux

Inicio: Todos los procesos empiezan con vruntime = 0.

Ciclo 1:

Todos los procesos tienen el mismo vruntime inicial, pero el Proceso E tiene el peso más alto y, por lo tanto, es el menos penalizado en la acumulación de vruntime.

El Proceso E se ejecuta, digamos, por una unidad de tiempo. Su vruntime incrementa menos debido a su alto peso.

Nuevo vruntime de E = 0 + (1 unidad de tiempo / 1024) = 0.0009765625.

Ciclo 2:

El Proceso B tiene el siguiente peso más alto. Se ejecuta por una unidad de tiempo.

Nuevo vruntime de B = 0 + (1 unidad de tiempo / 512) = 0.001953125.

Ciclo 3:

Proceso A o D (ambos tienen nice = 0 y el mismo peso).

Supongamos que se selecciona el Proceso A. Se ejecuta por una unidad de tiempo.

Nuevo vruntime de A = 0 + (1 unidad de tiempo / 256) = 0.00390625.

Linux

Ciclo 4:

Se ejecuta el Proceso D bajo las mismas condiciones que A.

Nuevo vruntime de D = 0 + (1 unidad de tiempo / 256) = 0.00390625.

Ciclo 5:

El Proceso C tiene el peso más bajo y por lo tanto, su vruntime crece más rápidamente. Se ejecuta por una unidad de tiempo.

Nuevo vruntime de C = 0 + (1 unidad de tiempo / 128) = 0.0078125.

Revisión y Selección Continua:

Al final de estos ciclos, el scheduler revisa todos los vruntime. El Proceso E, que ha acumulado el menor vruntime (0.0009765625), sería normalmente seleccionado nuevamente, ya que su vruntime sigue siendo el más bajo.