



Sistemas Operativos

Scheduling



Scheduling o Planificación de Procesos

Cuando hay múltiples cosas que 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 solo 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

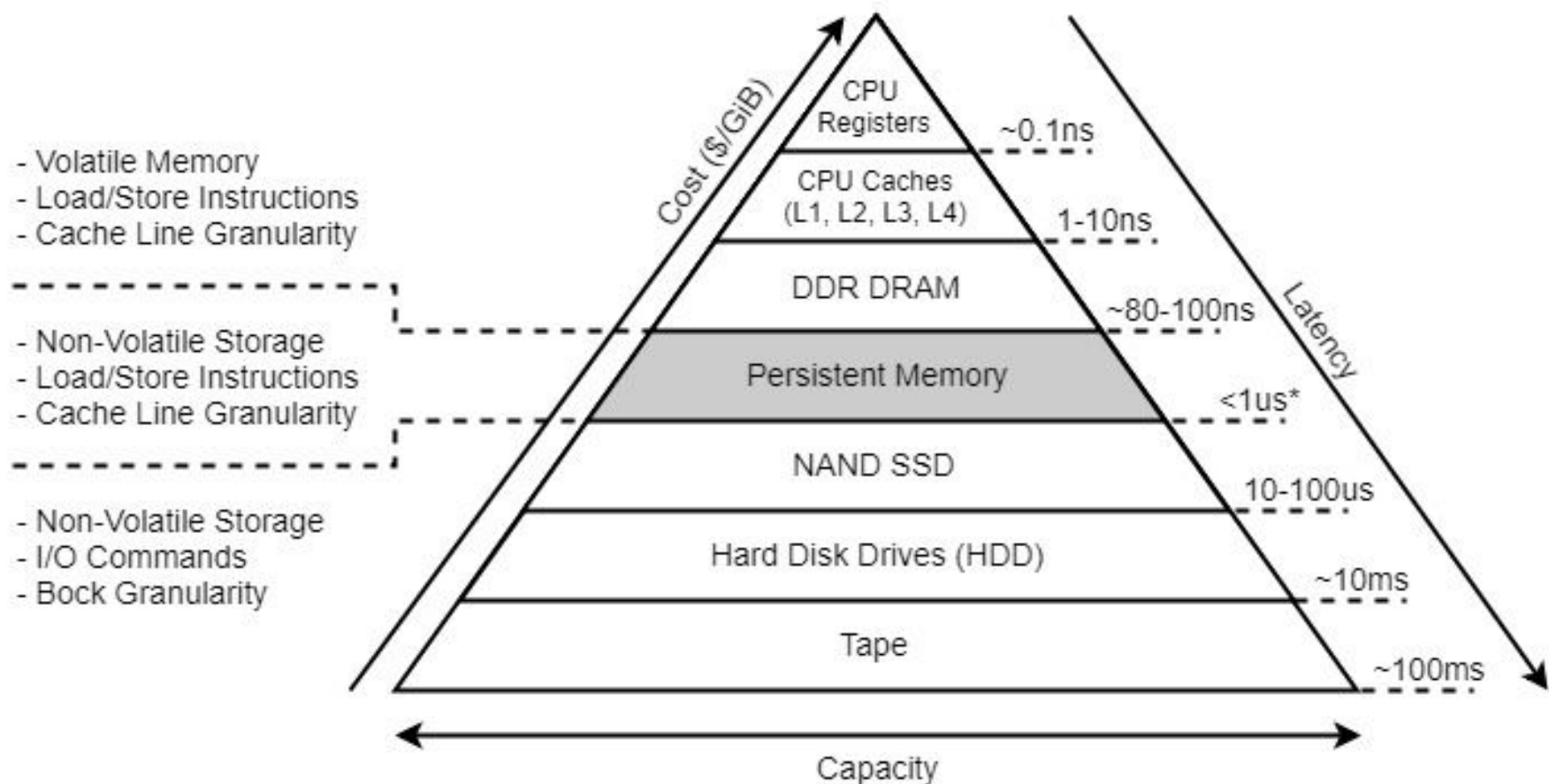
Entonces, el cálculo es más realista si se supone que un proceso gasta una fracción p , bloqueado en E/S. De esta manera, si tenemos n procesos esperando para hacer operaciones de entrada y salida, la probabilidad de que los n procesos estén haciendo E/S es p^n

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 $1-0.8^3 = 0.488$ es decir el 48 de ocupación de la cpu.

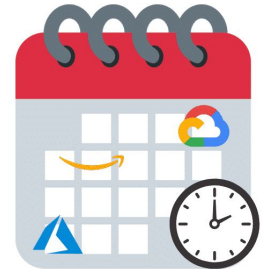
Si se supone que se tienen 10 procesos, entonces la fórmula cambia a $1-0.8^{10} = 0.89$ el 89% de utilización, aquí es donde se ve la IMPORTANCIA de la Multiprogramación.



(*) 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 **Scheduler** 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*:

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

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

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)

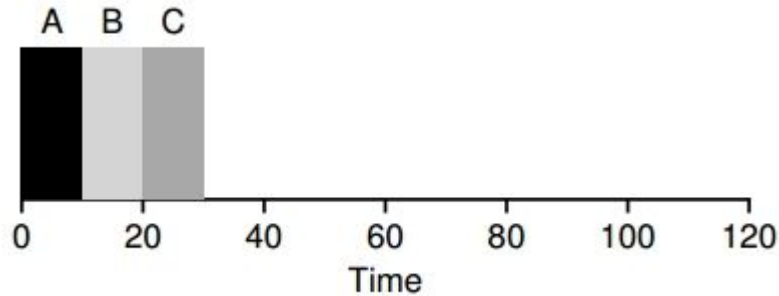
El algoritmo más básico para implementar como política de planificaciones es el First In First Out o First Come, First Served.

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 $T_{arrival}=0$.

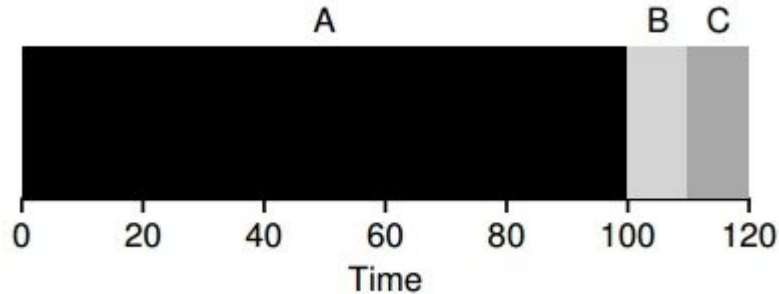


Si bien llegan todos al mismo tiempo llegaron con un insignificante retraso de forma tal que llegó A, B y C. Si se asume que todos tardan 10 segundos en ejecutarse... ¿cuánto es el Taround?

$$(10+20+30)/3=20$$

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 Iaround?

$$(100+110+120)/3=110$$

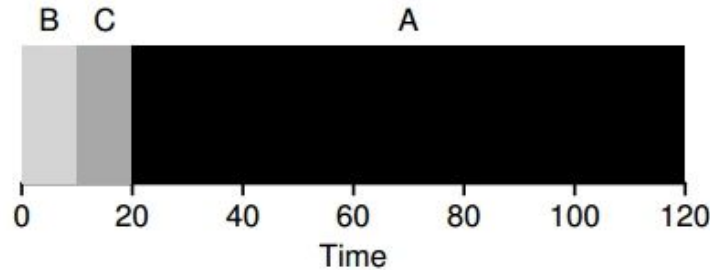
First In, First Out (FIFO)

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.

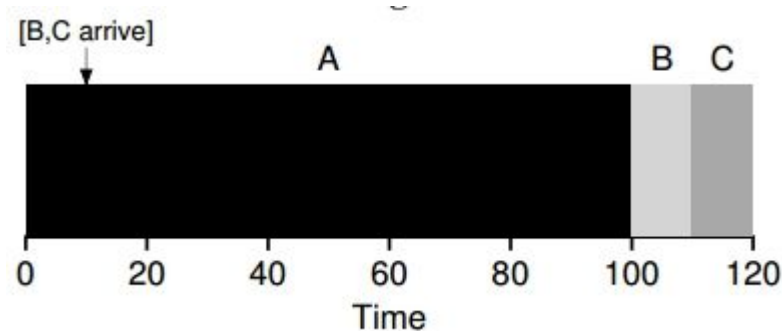


En el mismo caso de arriba, se mejora el turnaround time con el sencillo hecho de ejecutar B, C y A en ese orden:

$$(10+20+120)/3=50$$

Shortest Job First (SJF)

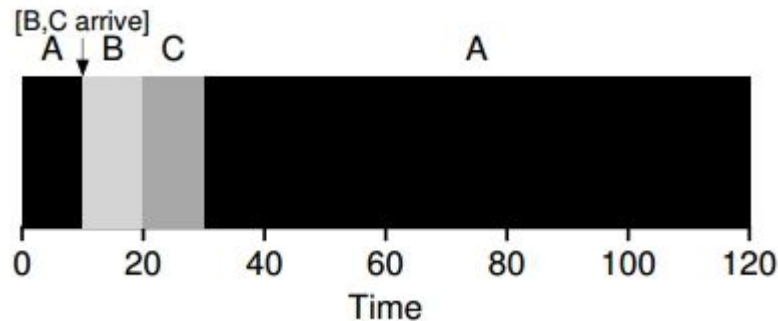
Utilizando SJF se obtuvo una significativa mejora... pero con las suposiciones iniciales que son muy poco realistas. Si se relaja la suposición 2, en la cual no todos los procesos llegan al mismo tiempo, por ejemplo llega el proceso A y a los 10 segundos llegan el proceso B y el proceso C. ¿Cómo sería el cálculo, ahora? $t_0 = 10$ 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.

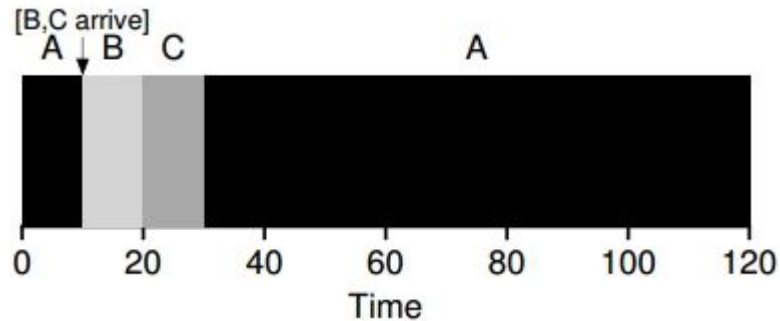


El caso anterior el de SFJ es una política non-preemptive

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:

$$T_{\text{response}} = T_{\text{first run}} - T_{\text{arrival}}$$

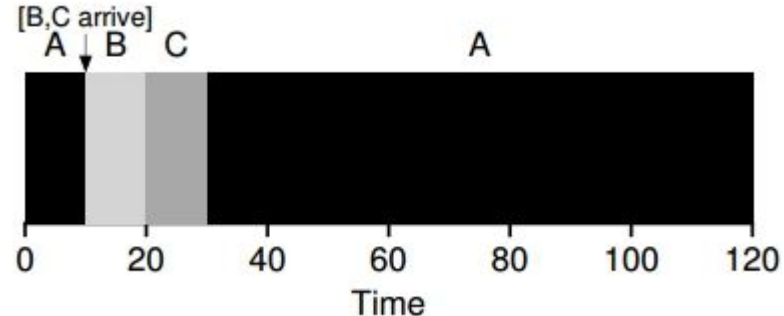
Una nueva métrica: Tiempo de Respuesta

El T_{response} de 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)

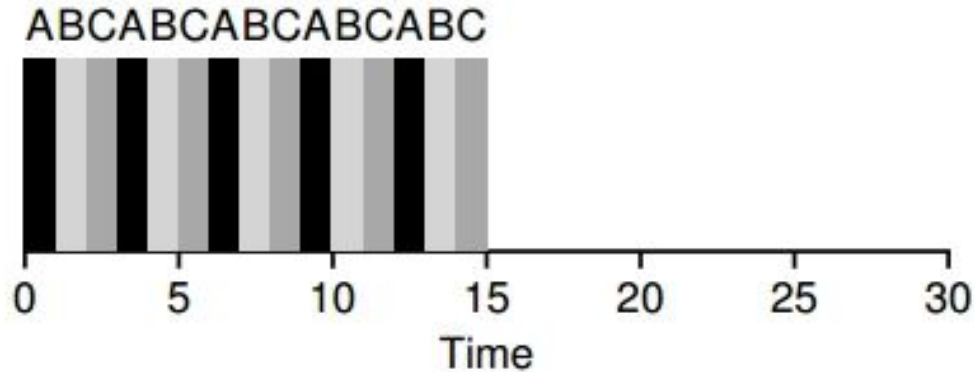
En promedio el T_{response} es de 3.33 seg. Entonces



¿cómo escribir un planificador que tenga noción del tiempo de respuesta?

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



Multi Level Feedback Queue

Esta técnica llamada Multi-Level Feedback Queue de planificación fue descrita 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 mas corta primero, desafortunadamente el sistema operativo nunca sabe a priori cuanto 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*.



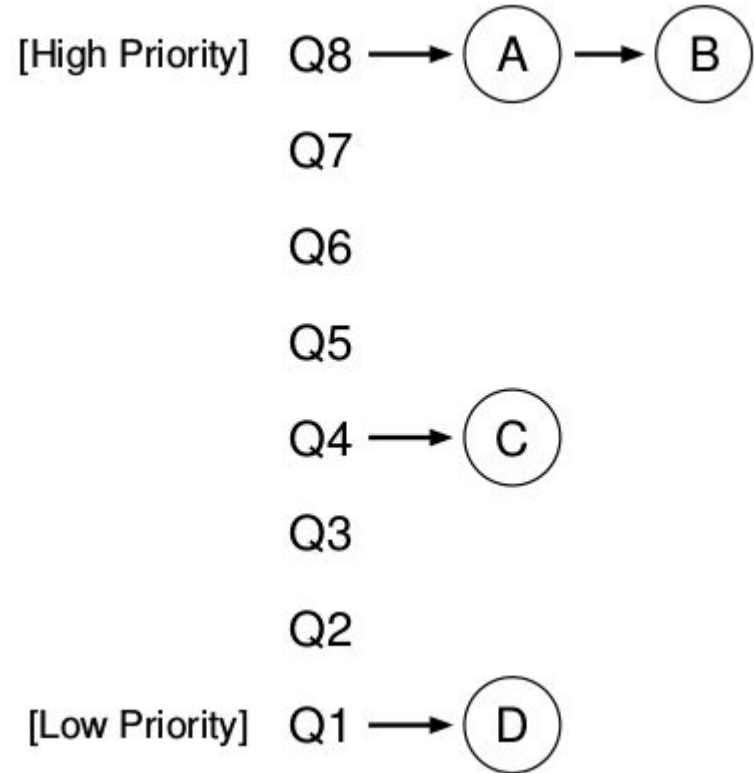
MLQF: Las reglas básicas

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 t_0 : 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 varia 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



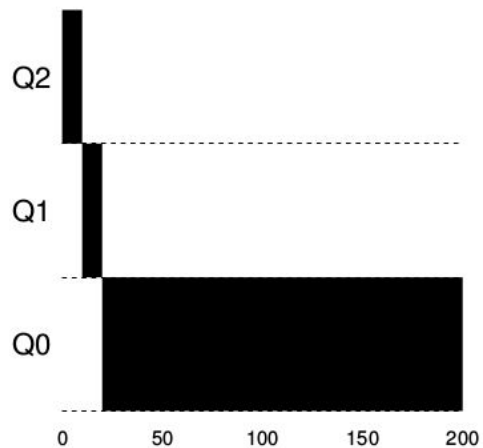
Primer intento: ¿Cómo cambiar la prioridad ?

Se debe decidir como MLFQ va a cambiarle 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. A continuación e muestra un primer intento de algoritmo de ajuste de prioridades:

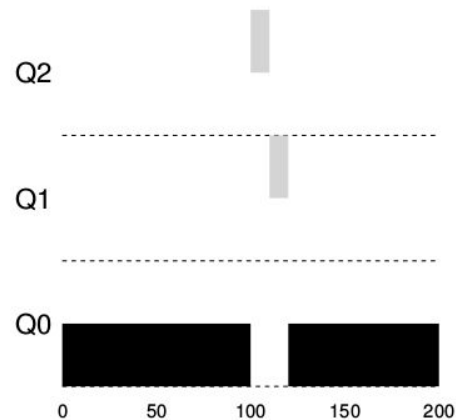
REGLA 3: Cuando una tarea entra en el sistema se pone con la mas alta prioridad

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.



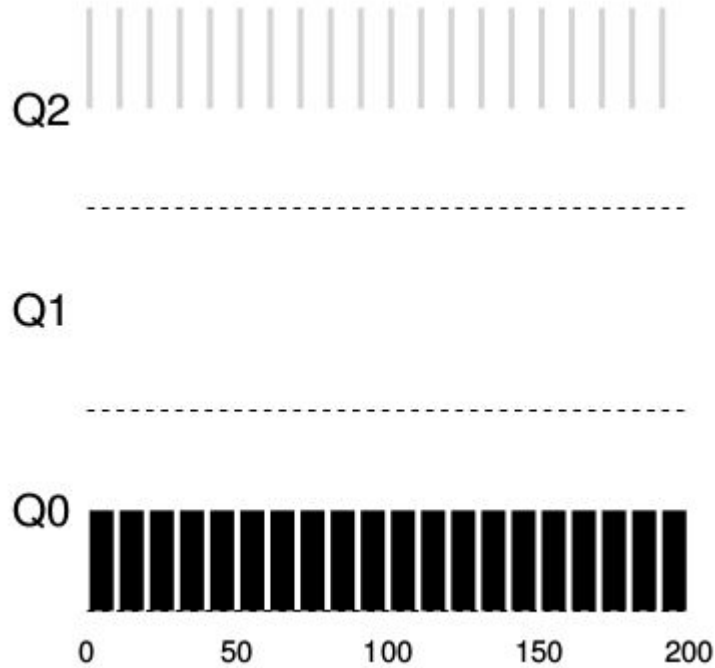
Una única tarea con
ejecución larga.



una tarea corta.



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 .



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.

Que pasa con la entrada y salida.



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 mas 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.



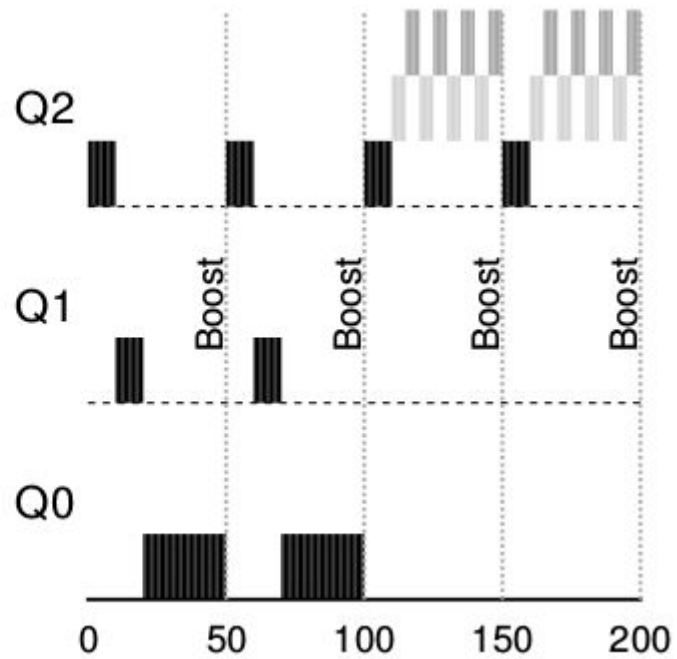
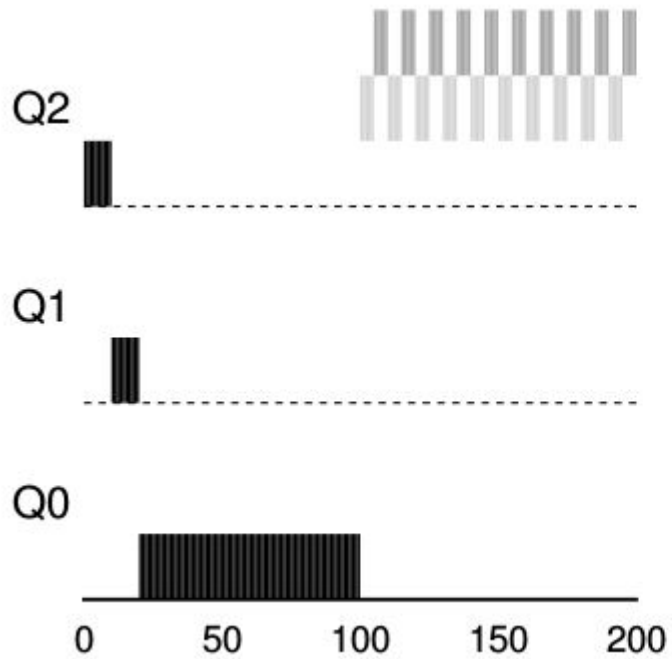
Segundo Approach: como mejorar la prioridad

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 mas prioridad.

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.



Boost Time!!!



Boost Time!!!

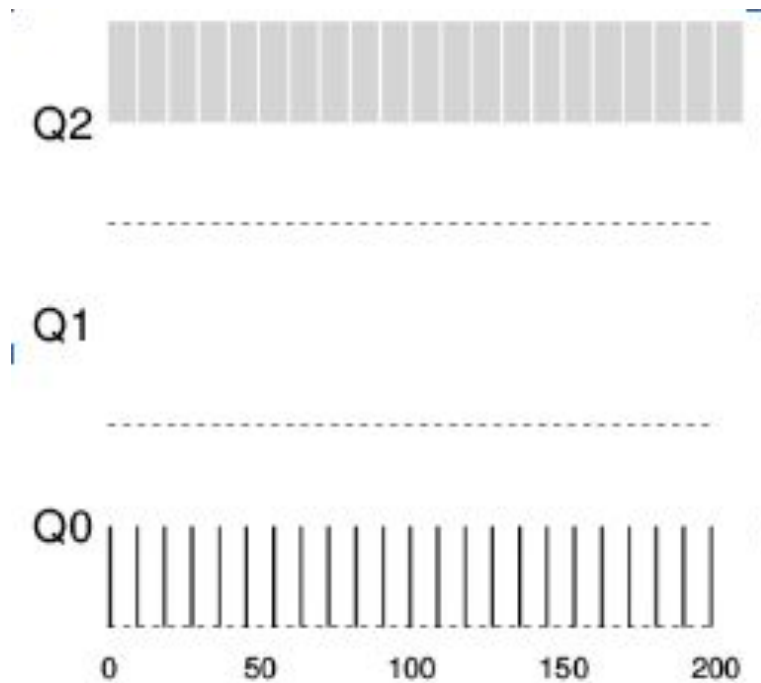
Obviamente el agregado del periodo de tiempo S va a desembocar en la pregunta obvia: *Cuánto debería ser el valor del tiempo S* . Algunos investigadores suelen llamar a este tipo de valores dentro de un sistema **VOO-DOO CONSTANTS** porque parece que requieren cierta magia negra para ser determinados 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.

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



Cheating!





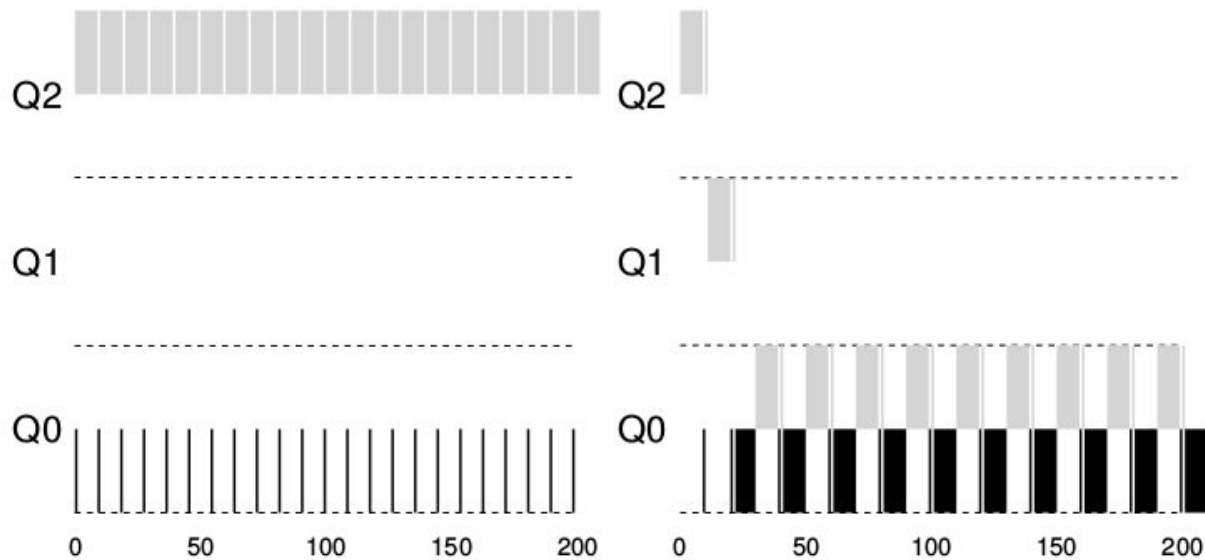
Cheating!

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, 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)

Cheating!





Resumen

Se vio la técnica de planificación conocida como multi-level feed back queue (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. 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 cuántas 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 más prioridad.

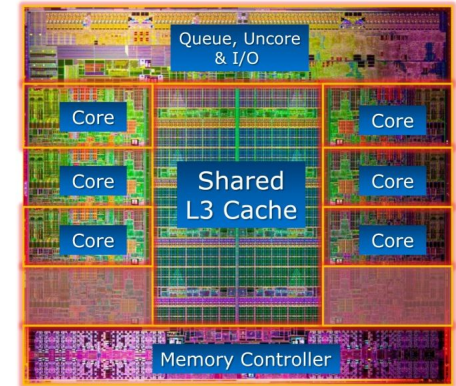
Planificación Avanzada

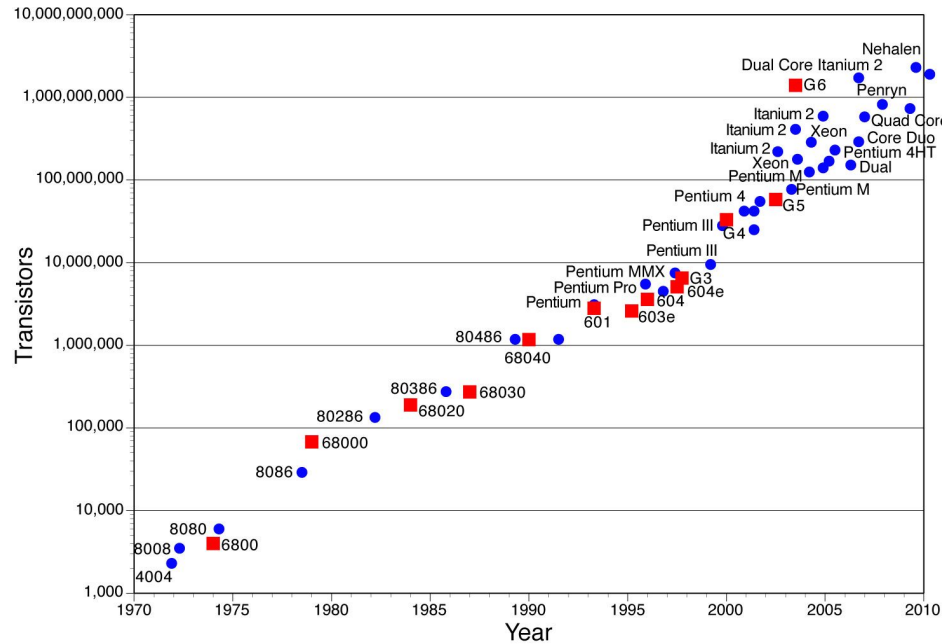
Planificación multiprocesador

En los últimos años los sistemas multi-procesadores han ido creciendo en los lugares comunes de la informática como por ejemplo en las computadores desktop, laptops y dispositivos móviles.

El advenimiento de los procesadores multi-núcleo, en los cuales múltiples núcleos de CPU están empaquetados en un único chip, en nuestros días esa arquitectura está en plena proliferación.

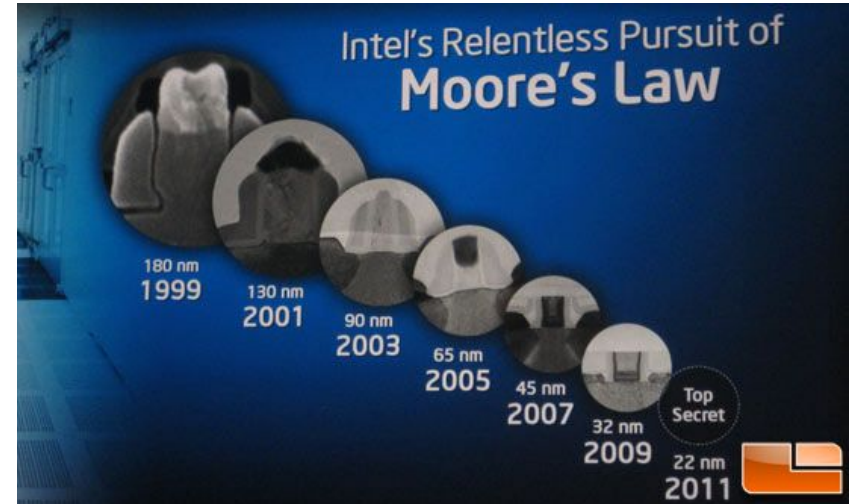
Este tipo de arquitectura de procesadores se volvió popular debido a que es complicado construir CPU cada vez más rápidas y que a su vez las mismas no se fundan por el calor producido por la potencia que consumen.





En 1965 Gordon Moore, cofundador de Intel formuló una ley empírica que se ha podido constatar hasta nuestros días que dice:

“Aproximadamente cada dos años se duplica el número de transistores en un microprocesador por unidad de área”





Ley de Amdahl

La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.

$$T_m = T_a \cdot \left((1 - F_m) + \frac{F_m}{A_m} \right)$$

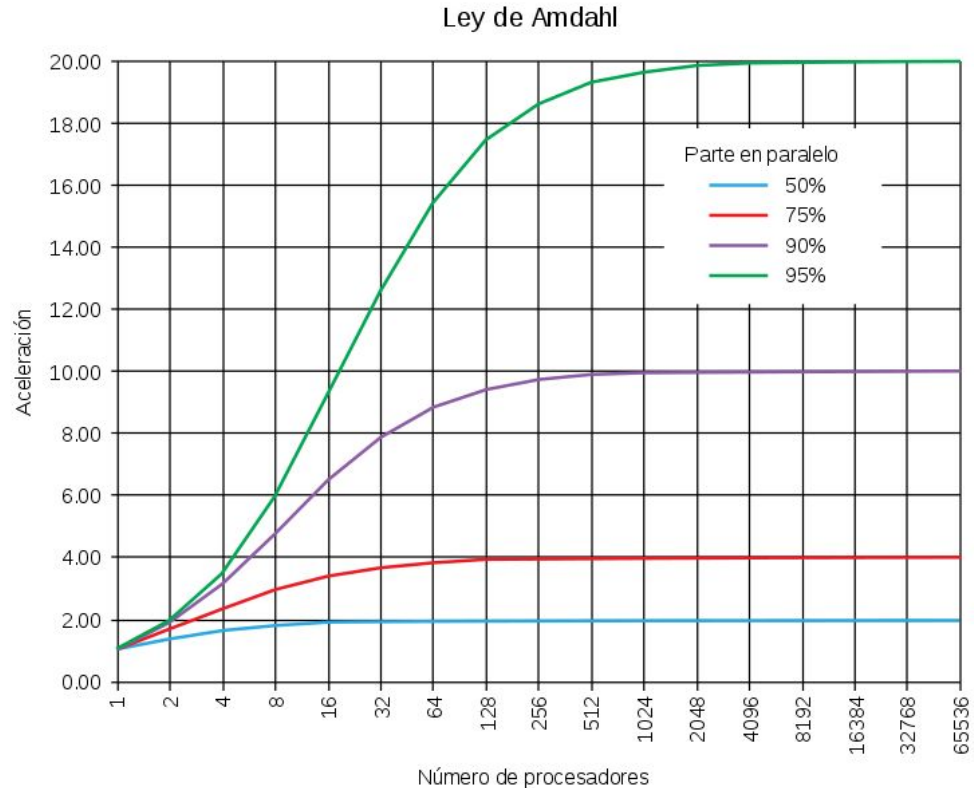
- F_m = fracción de tiempo que el sistema utiliza el subsistema mejorado
- A_m = factor de mejora que se ha introducido en el subsistema mejorado.
- T_a = tiempo de ejecución antiguo.
- T_m = tiempo de ejecución mejorado.

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}}$$

Ley de Amdahl

El incremento de velocidad de un programa utilizando múltiples procesadores en computación distribuida está limitada por la fracción secuencial del programa.

Por ejemplo, si la porción 0.5 del programa es secuencial, el incremento de velocidad máximo teórico con computación distribuida será de 2 ($1/(0.5+(1-0.5)/N)$) cuando N sea muy grande.





Planificación multiprocesador

Por supuesto que tener muchos procesadores en un único chip conlleva un conjunto de dificultades:

Una aplicación típica por ejemplo un programa escrito en C usa únicamente una CPU:

- por lo cual agregar más CPU no implica que la aplicación corra de forma más rápida. El remedio a este problema es la necesidad de escribir aplicaciones que corran en paralelo, por ejemplo usando threads.
- Las aplicaciones multithreads pueden diseminar el trabajo a lo largo de múltiples CPUs y por ende correr más rápido cuantas más CPU hayan.

Más allá de las aplicaciones sale a la luz una nueva problemática que es la planificación en multiprocesadores.



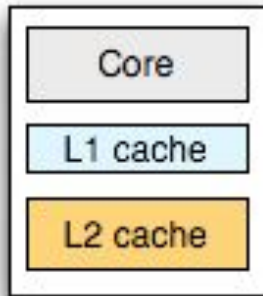
La arquitectura multiprocesador

Para poder entender cuales son los problemas que atañan a la planificación multiprocesador en primer lugar hay que entender cuál es la diferencia fundamental entre hardware monoprocesador y hardware multi-procesador.

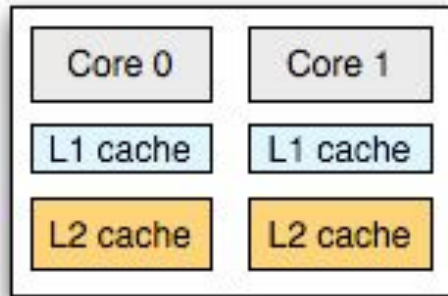
La diferencia se centra básicamente alrededor de un tipo de hardware llamado cache, y de qué forma exactamente los datos en la cache son compartidos a través de los multiprocesadores.

En un sistema con único CPU hay una jerarquía de el hardware de cache que generalmente ayuda al procesador a correr los programas más rápidamente. Las cache son pequeñas y rápidas memorias que (generalmente) mantienen copias de datos que son comúnmente utilizados que se encuentran en la memoria principal del sistema.

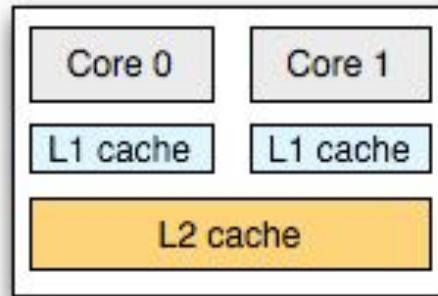
La arquitectura multiprocesador



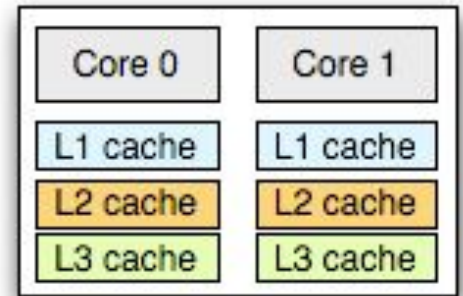
single core



AMD Opteron, Athlon



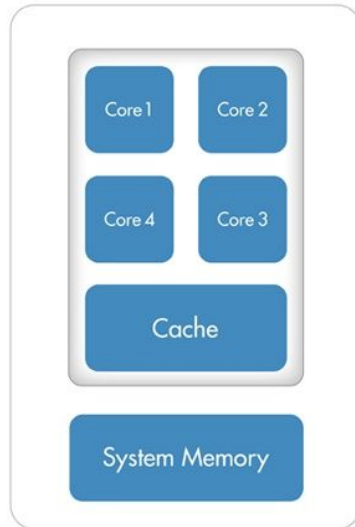
Intel Core Duo, Xeon



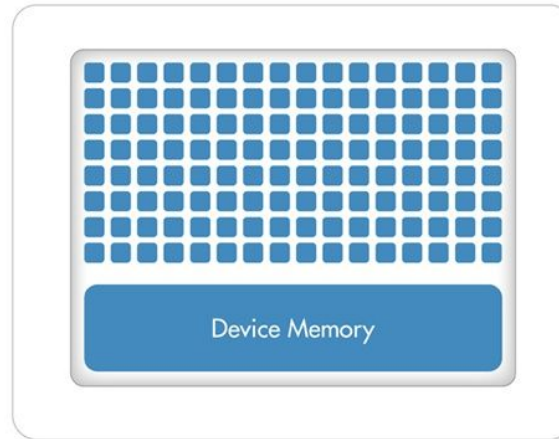
Intel Itanium 2

La arquitectura multiprocesador

CPU (Multiple Cores)



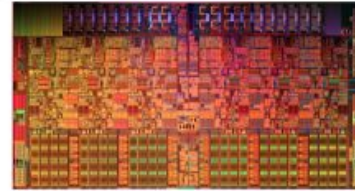
GPU (Hundreds of Cores)



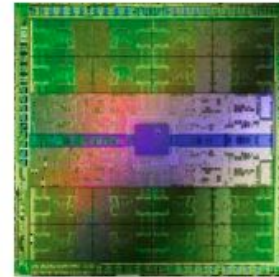
La arquitectura multiprocesador

Specifications	Westmere-EP	Fermi (Tesla C2050)
Processing Elements	6 cores, 2 issue, 4 way SIMD @3.46 GHz	14 SMs, 2 issue, 16 way SIMD @1.15 GHz
Resident Strands/Threads (max)	6 cores, 2 threads, 4 way SIMD: 48 strands	14 SMs, 48 SIMD vectors, 32 way SIMD: 21504 threads
SP GFLOP/s	166	1030
Memory Bandwidth	32 GB/s	144 GB/s
Register File	6 kB (?)	1.75 MB
Local Store/L1 Cache	192 kB	896 kB
L2 Cache	1536 kB	0.75 MB
L3 Cache	12 MB	-

# transistors & area:	1.2 B, 240 mm ²	3 B, 520 mm ²
thermal design power:	130 Watts	160+ Watts? (240 W/card)



Westmere-EP (32nm)



Fermi (40nm)

This is how CPUs cores works





Cache

La idea detrás de la localidad temporal es que cuando cierta cantidad de datos se han accedido, es probable que sean accedidos otra vez en un futuro cercano; imaginar por ejemplo variables o instrucciones que se ejecutan una y otra vez en un ciclo.

```
for(i = 0; i < 20; i++)  
    for(j = 0; j < 10; j++)  
        a[i] = a[i]*j;
```

La idea detrás de la localidad espacial se basa en que un programa que accede a una dirección X es muy probable que necesite volver a acceder cerca de X. Aquí podría pensarse en un programa sobre un arreglo. Teniendo en cuenta que este tipo de localidad existe en la mayoría de los programas los sistemas de hardware pueden hacer buen uso de las cache.

¿Que sucede cuando múltiples procesadores en un único sistema tiene que compartir una única memoria principal?



Cache

Como se verá el cacheo con múltiples CPU es mucho más complicado. Imaginarse, por ejemplo, que un programa que se está ejecutando en la CPU1 lee un dato cuyo valor es D en la dirección de memoria A:

- Debido a su este dato no está cacheado en la CPU1, el sistema lo trae de la memoria principal y toma este valor D.
- El programa entonces modifica el valor en la dirección A, esto se realiza actualizando su valor D1; dado que escribir los datos directamente en la memoria principal es muy lento, el sistema habitualmente lo deja para más tarde.
- Entonces se asume que el OS decide parar de ejecutar un programa y mover este programa a la CPU 2.
- EL programa entonces vuelve a ejecutarse en la CPU 2 y relee el valor en la dirección A;

porque no existe tal valor en la cache de la CPU2, entonces el sistema trae el valor de memoria desde la memoria principal y obtiene el viejo valor D en vez del correcto valor D prima. OOPS!.



Afinidad de Cache

El último tema a tener en cuenta cuando se arma un planificador con multiprocesadores con cache, es conocida como la afinidad de cache o cache affinity.

El concepto es básico:

Cuando un proceso corre sobre una CPU en particular va construyendo un cachito del estado de si mismo en las cache de esa CPU

Entonces la próxima vez que el proceso se ejecute seria bastante interesante que se ejecutara en la misma CPU, ya que se va a ejecutar mas rápido si parte de su estado esta en esa CPU.

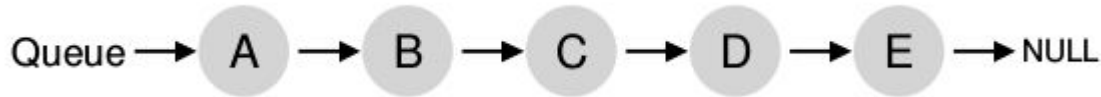
Si, en cambio se ejecuta el proceso en una CPU diferente cada vez, el rendimiento del proceso va a ser peor, ya que tendrá que volver a cargar su estado o parte del mismo cada vez que se ejecute.

Por ende un planificador multiprocesador debería considerar la afinidad de cache cuando toma sus decisiones de planificación, tal vez prefiriendo mantener a un proceso en un determinado CPU si es posible.

Planificación de cola única

SINGLE QUEUE MULTIPROCESSOR SCHEDULING o SQMS.

Esta forma de plantear la planificación tiene la ventaja de la simplicidad ya que no requiere mucho trabajo tomar la política existente que agarra la mejor tarea y la pone a ejecutar y adaptarla para que trabaje con mas de una CPU.





Planificación de cola única

No es escalable

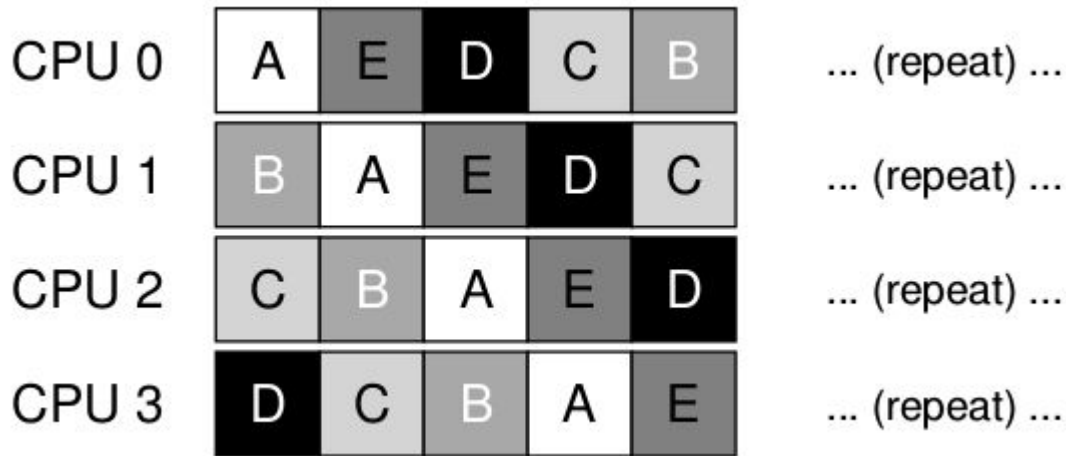
Para asegurarse que la planificación trabaje correctamente en una arquitectura de múltiple CPU los desarrolladores tienen que insertar algún tipo de bloqueo en su código fuente. Es decir el bloqueo tiene que asegurar que cuando SQMS accede a una única cola (como para encontrar la próxima tarea a ejecutar), un resultado correcto se ha obtenido. EL bloqueo desafortunadamente va a reducir en mucho la performance particularmente a medida que el número de CPU del sistema empieza a crecer. Téngase en cuenta que con un único bloqueo, el sistema pierde más tiempo sobrecargándose en el bloqueo y menos tiempo en lo que debería estar haciendo.

Otro gran problema de SQMS es la afinidad del cache.

Planificación de cola única

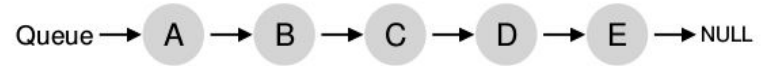


A lo largo del tiempo, asumiendo que cada trabajo se ejecuta en un determina time slice otro trabajo es elegido para ser ejecutado y entonces un posible esquema de planificación a través de la CPU podría ser:



Planificación de cola única

En algunos casos se podría proveer a ciertas tareas con cierta afinidad y a otras dejarlas cambiando de CPU para balancear la carga por ejemplo mirar:



CPU 0



... (repeat) ...

CPU 1



... (repeat) ...

CPU 2



... (repeat) ...

CPU 3



... (repeat) ...



Multi-Queue Planificacion

Debido a los problemas que tiene single queue scheduler varia gente opto por crear un planificador multi queue que se llama MULTI-QUEUE MULTIPROCESSOR SHCHEDULING en MQMS.

El esquema básico de la planificación consiste en múltiples colas. Cada cola va a seguir una determinada disciplina de planificación, por ejemplo, round robin, cuando una tarea entra en el sistema ésta se coloca exactamente en una única cola de planificación, de acuerdo con alguna heurística. Esto implica que es esencialmente planificada en forma independiente.

Multi-Queue Planificacion

Por ejemplo, Si se asume que se está trabajando con un sistema de dos procesadores (CPU0 y CPU1) y una determinada cantidad de tareas ingresan al sistema (A, B, C y D , por ejemplo). Dado que cada CPU tiene exactamente 2 colas, el planificador podría decidir distribuir las tareas de la siguiente forma:



Multi-Queue Planificacion

Dependiendo de la política de planificación por ejemplo Round Robin, la planificación a lo largo del tiempo podría verse así :

MQMS tiene la ventaja sobre SQMS debido a que es enteramente escalable. A medida que las CPU van creciendo también lo hacen las colas, lo que conlleva a que los lock y las cache no sean ya un problema.

MQMS intrínsecamente provee afinidad de cache, es decir las tareas intentan mantenerse en la CPU en la que fueron planificadas.



Multi-Queue Planificacion

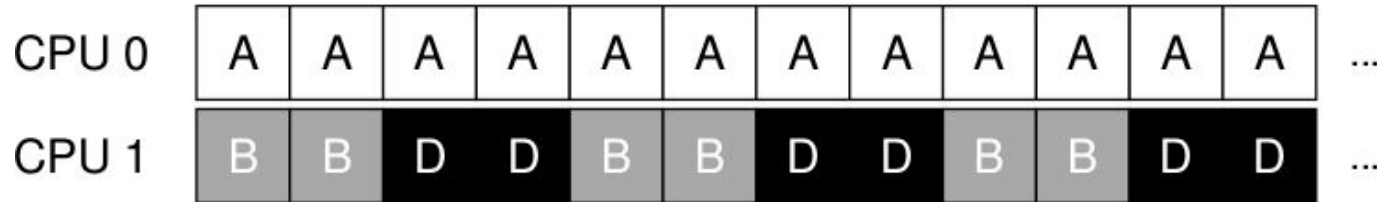
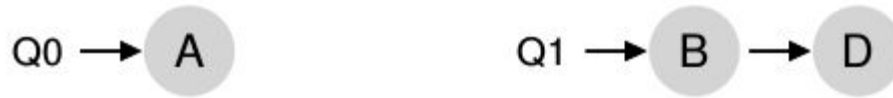
Dependiendo de la política de planificación por ejemplo Round Robin, la planificación a lo largo del tiempo podría verse así :

MQMS tiene la ventaja sobre SQMS debido a que es enteramente escalable. A medida que las CPU van creciendo también lo hacen las colas, lo que conlleva a que los lock y las cache no sean ya un problema.

MQMS intrínsecamente provee afinidad de cache, es decir las tareas intentan mantenerse en la CPU en la que fueron planificadas.



Multi-Queue Planificacion: Load Imbalance



Multi-Queue Planificacion: Load Imbalance

Q0 →

Q1 → B → D

CPU 0

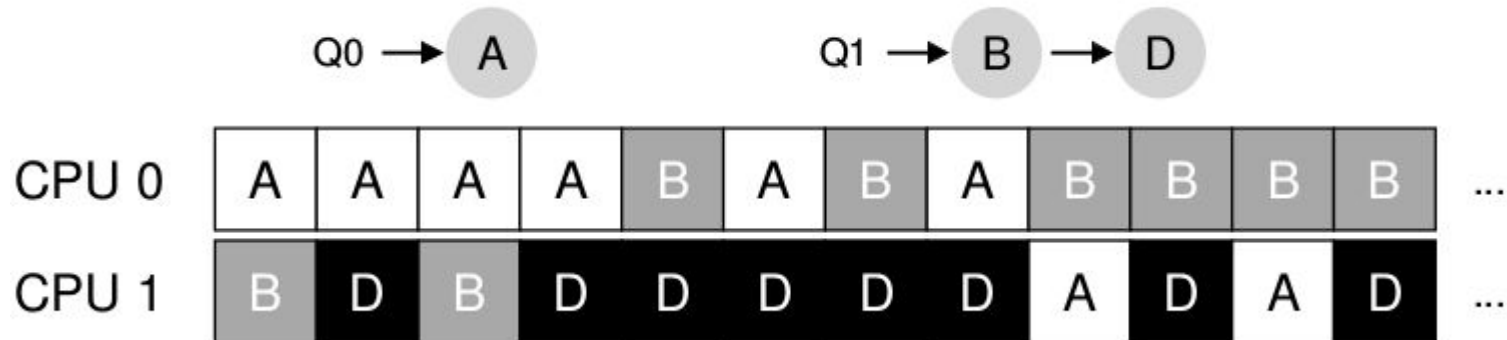
CPU 1



Multi-Queue Planificacion: Load Imbalance

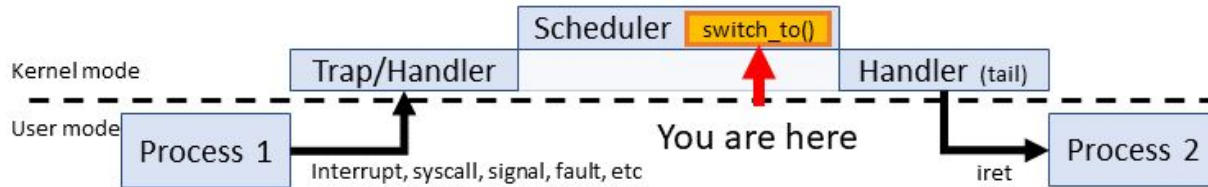
La pregunta que se plantea es ¿Cómo se resuelve el problema del **load imbalance**?

La respuesta más obvia es aquella de mover las tareas de un lado a otro, esta técnica se conoce como **migración o migration**. Mediante la migración de una tarea a otra cpu se puede conseguir un verdadero balance de carga.

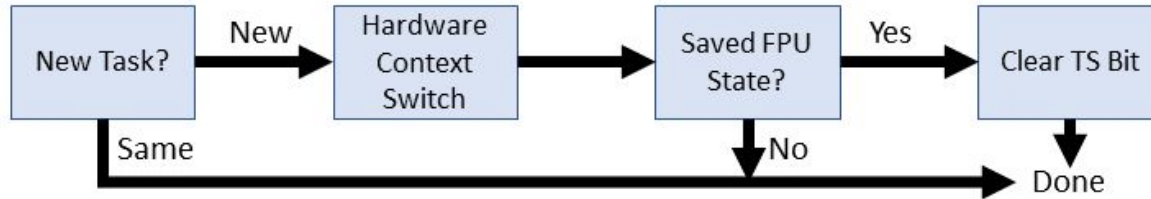


La Vida Real

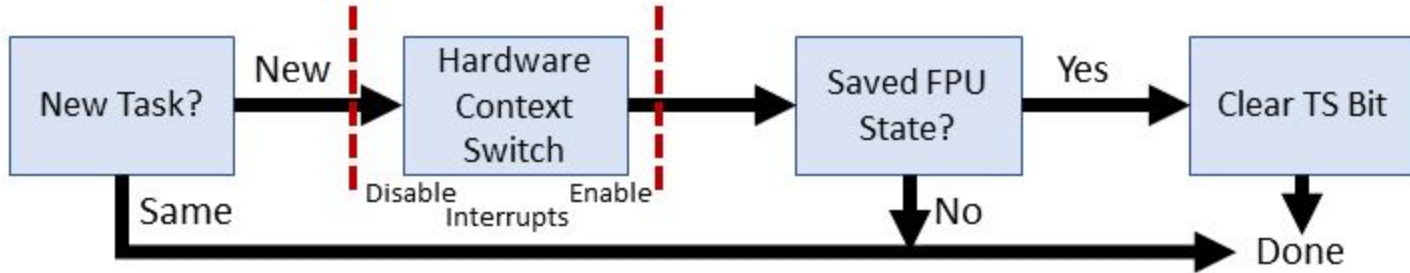
Linux Context Switch



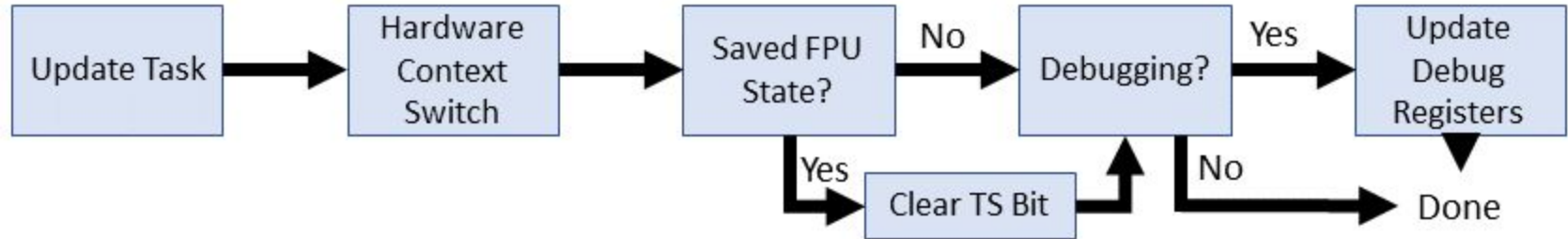
Linux Pre-1.0 - Ancient History (1991)



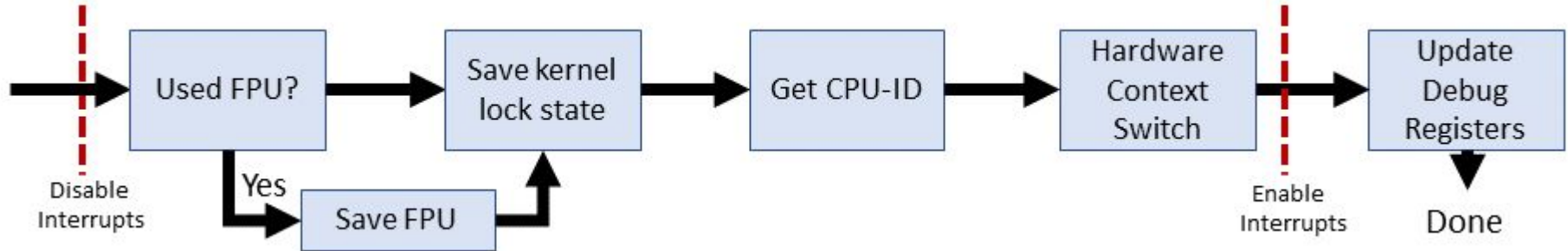
Linux 1.x - Proof of concept (1994)



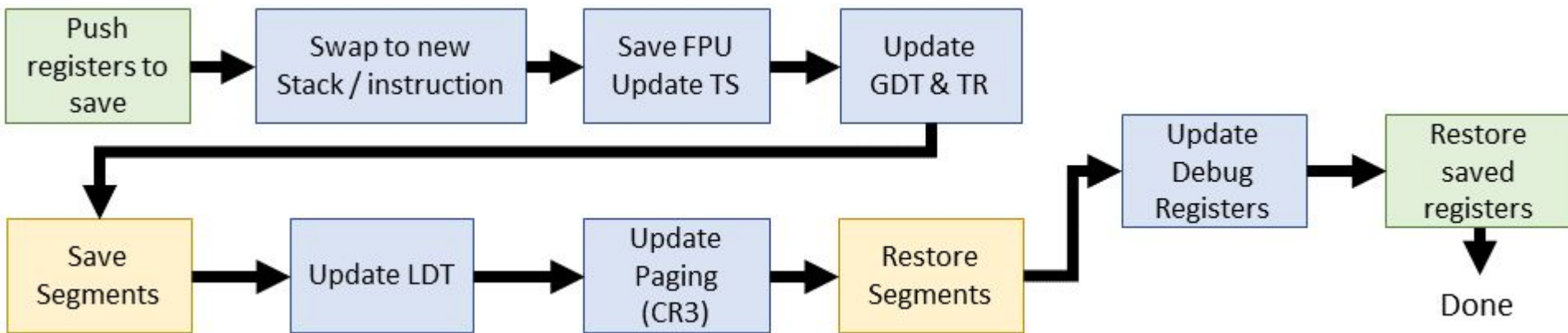
Linux 2.0.1 - Uniprocessing edition (UP)



Linux 2.0.1 - Symmetric multiprocessing edition (SMP)

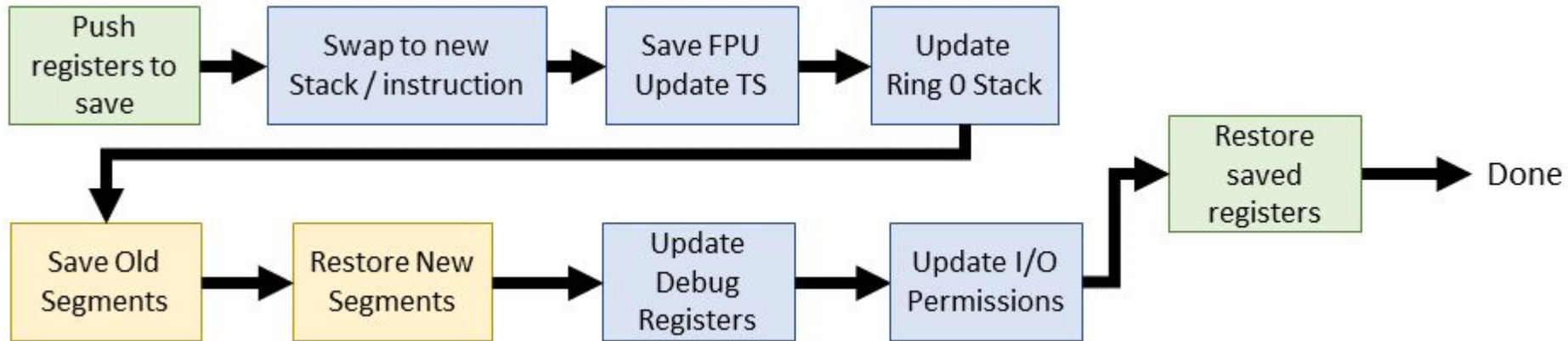


Linux 2.2 (1999)

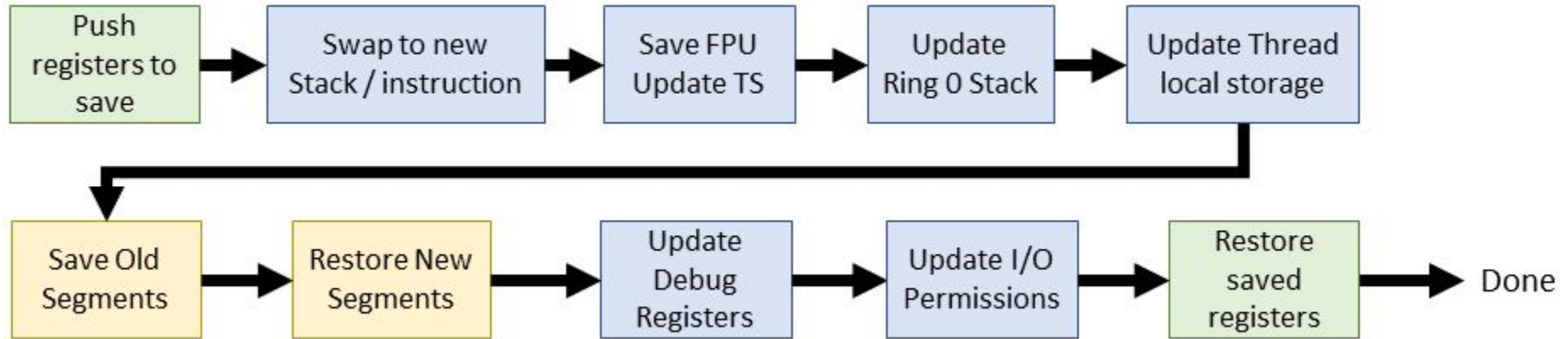


Linux 2.2 was worth waiting for: software context switching! We're still using the task register (TR) to reference a TSS. SMP and UP procedures merged with uniform handling of FPU state. Much of the context switch now performed in C.

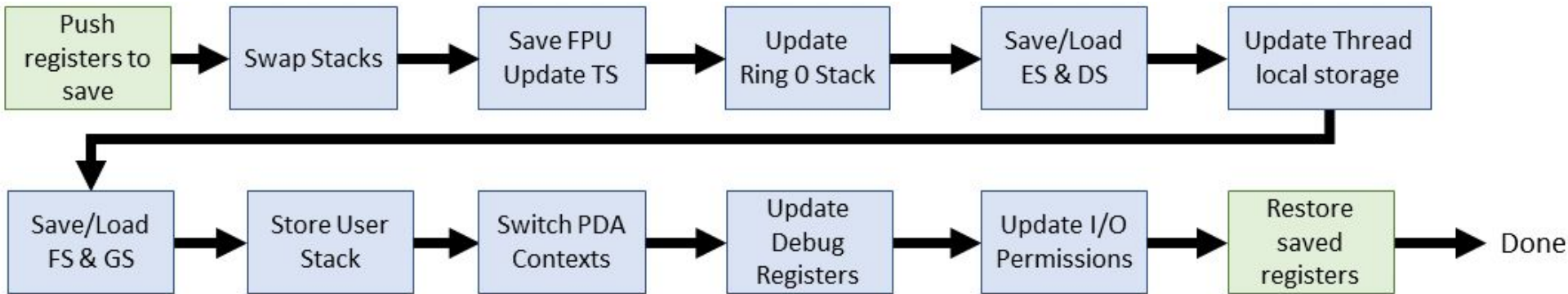
Linux 2.4 (2001)



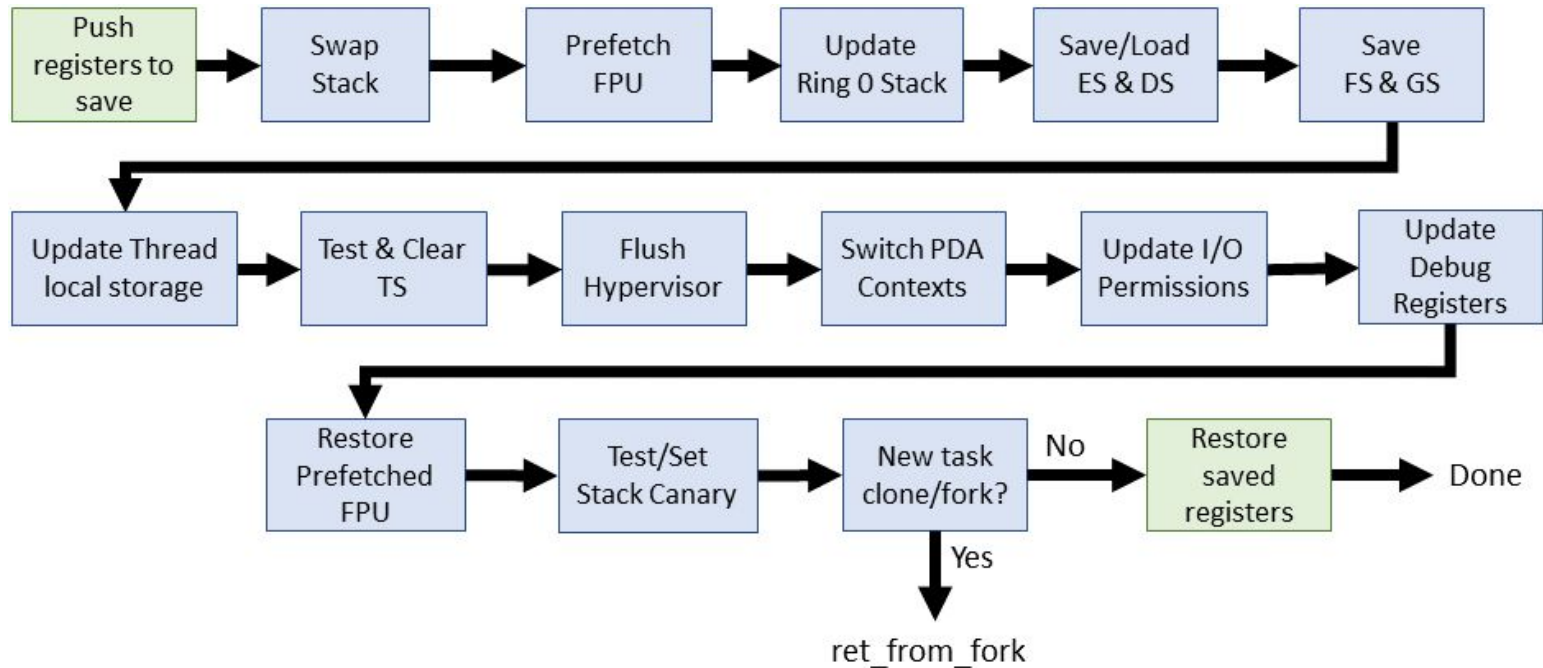
Linux 2.6.0 - i386 edition



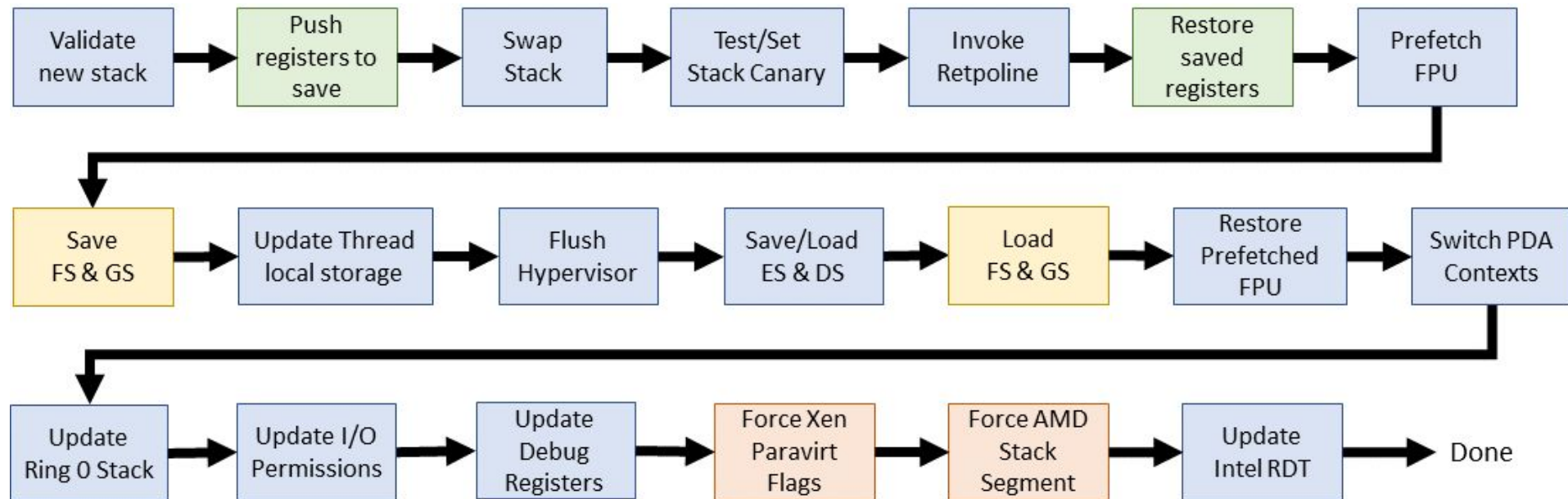
Linux 2.6.0 - x86_64 edition



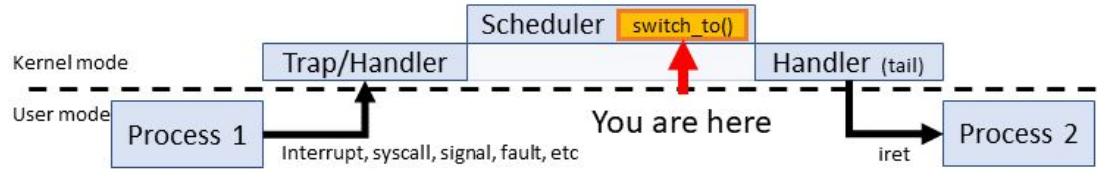
Linux 3.0 - The modern operating system (2011)



Linux 4.14.67 - The latest LTS kernel (2018)



Linux Scheduler



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:

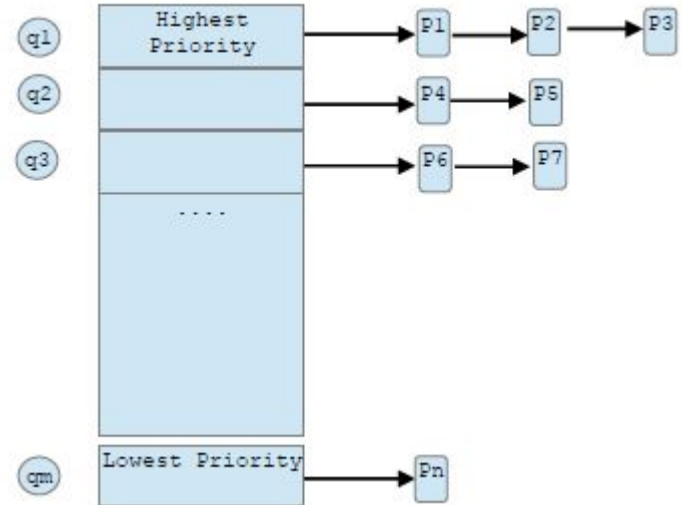
- el time-slice

- las colas de ejecución por proceso

Si bien el planificador $O(1)$ estaba muy bueno para equipos servidores no funcionaba tan bien para procesos con interacción con usuarios procesos interactivos. E

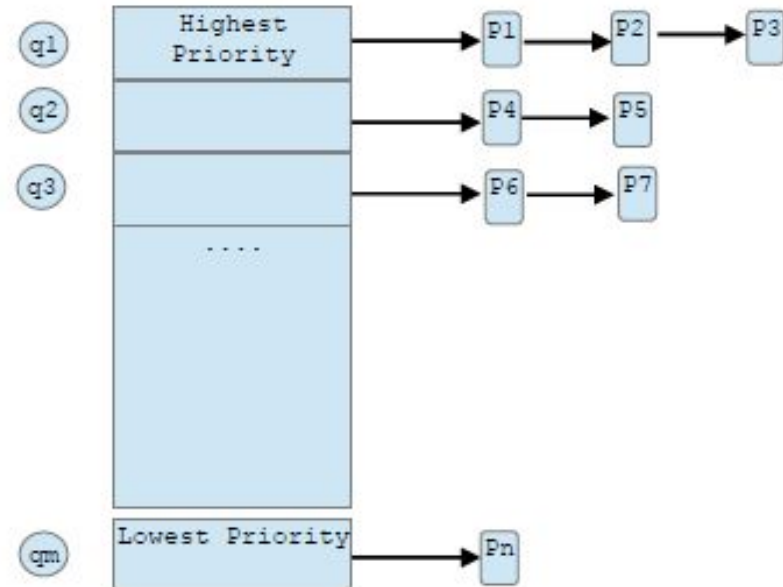
Linux Scheduler

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.



rotating Starircase Deadline scheduler

- Staircase is an array of priorities
- Each step corresponds to a specific priority
- Each step maintains a linked list of waiting processes
- Processes in the list are processed in round-robin fashion
- Each step has its own quota (upper limit)
- When a step has used the quota, all the processes on that step are pushed down to the next step
- When pushed down, the processes have to contend with those processes already on that step
- Processes on lower steps run only after those in upper processes
- Interactive processes sleep more often and so they do not consume their time even while at higher priority levels. So, they will usually run immediately
- Whenever the processes use their quota, they are placed in a the expired array but the same original priority level
- Similar to $O(1)$ scheduler, when the active array becomes empty, both active and expired arrays are swapped



Linux Scheduler

Ingo Molnár es un hacker de Linux húngaro. Actualmente trabaja para Red Hat.1

Es conocido por sus contribuciones al núcleo Linux relacionados con seguridad y rendimiento. Algunas de sus aportaciones al núcleo incluyen el planificador $O(1)$ de Linux-2.6.0, el Completely Fair Scheduler («Planificador Completamente Justo») de Linux-2.6.23, el servidor HTTP/FTP integrado en el kernel TUX, así como su trabajo para mejorar el manejo de hilos. También programó una característica de seguridad para el núcleo llamada «Exec Shield», que previene contra exploits de desbordamiento de búfer basados en la pila en la arquitectura x86 al deshabilitar los permisos de ejecución para la pila.

Entre Linux 2.6.21 y Linux 2.6.24, Ingo trabajó en el Completely Fair Scheduler (CFS) inspirado por el trabajo sobre planificadores de Con Kolivas. CFS reemplazó el planificador de Linux anterior en Linux-2.6.23.2





Linux: Completely Fair Scheduler (CFS)

El planificador de linux llamado Completely Fair implementa lo 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



Modo de Operación Básico

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 mas que que el runtime (es decir el tiempo que se esta ejecutando el proceso) normalizado por el número de procesos runnable (se mide en nanosegundos)

A medida que un proceso se ejecuta este acumula vruntime. En el caso más básico cada vruntime 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 esta corriendo y correr otro proceso?



Modo de Operación Básico

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 los procesos se ejecuten a costa de pérdida 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).

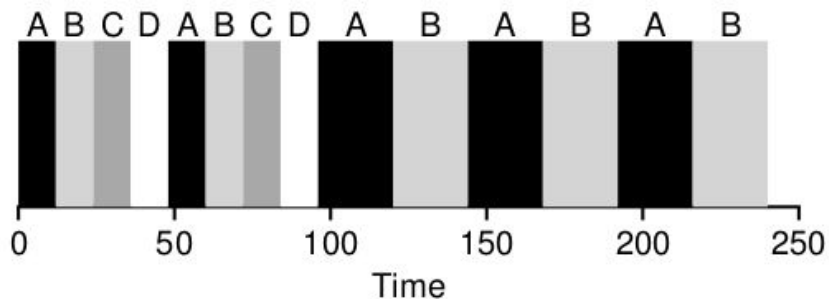
La forma en que CFS maneja esta tensión es mediante varios parámetros de control.

`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.

Modo de Operación Básico

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 swiches y por ende pequeños time slices? Sí.





Modo de Operación Básico

Para lidiar con este problema se introduce otro parámetro llamado `min_granularity`, que normalmente se setea con el valor de 6 ms. Entonces CFS nunca cedería 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)



Weighting (Niceness)

CFS tiene control sobre las prioridades de los procesos, de forma tal que los usuario y administradores puedan 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.

```
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,  
};
```



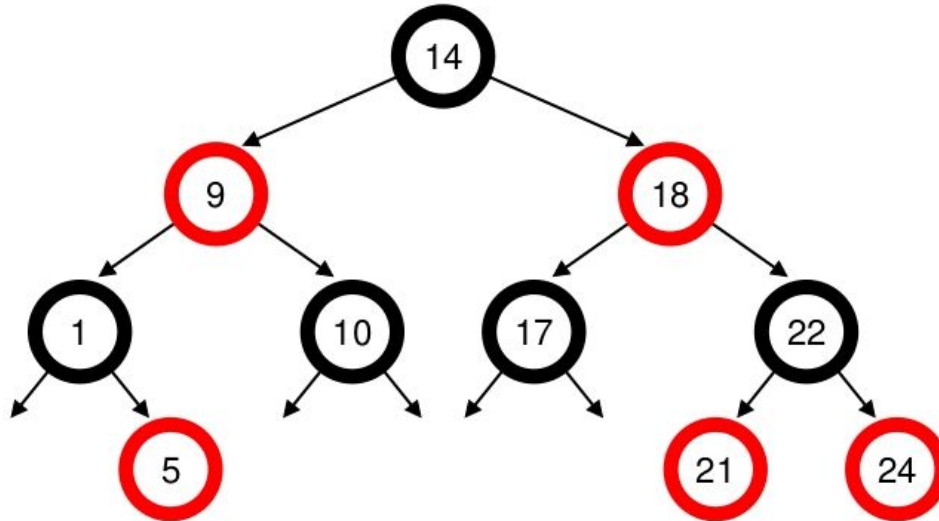
Utiliza Árboles Rojo-Negro

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 $O(\log n)$

Utiliza Arboles Rojo-Negro



```

struct task_struct {
    volatile long state;
    void *stack;
    unsigned int flags;
    int prio, static_prio normal_prio;
    const struct sched_class *sched_class;
    struct sched_entity se;
    ...
};

```

```

struct ofs_rq {
    ...
    struct rb_root tasks_timeline;
    ...
};

```

```

struct sched_entity {
    struct load_weight load;
    struct rb_node run_node;
    struct list_head group_node;
    ...
};

```

```

struct rb_node {
    unsigned long rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
};

```

