8

Planificación: La Cola Multinivel Retroalimentada

En este capítulo, abordaremos el problema de desarrollar uno de los enfoques más conocidos para la programación, conocido como la **Cola Multinivel Retroalimentada** (**MLFQ** por sus siglas en inglés, Multi-Level Feedback Queue). El planificador de cola multinivel retroalimentada (MLFQ) fue descripto por primera vez por Corbato et al. en 1962 [C+62] en un sistema conocido como el Sistema de Tiempo Compartido Compatible (CTSS), y este trabajo, junto con el trabajo posterior en Multics, llevó a la ACM a otorgarle a Corbato su más alto honor, el **Premio Turing**. Posteriormente, el planificador se fue perfeccionando a lo largo de los años hasta las implementaciones que vas a encontrar en algunos sistemas modernos.

El problema fundamental que la MLFQ intenta abordar tiene dos partes. Primero, busca optimizar el tiempo de entrega, que, como vimos en la nota anterior, se realiza ejecutando los trabajos más cortos primero; por desgracia, el sistema operativo generalmente no sabe por cuánto tiempo se ejecutará un trabajo, que es exactamente el conocimiento que requieren los algoritmos como SJF (o STCF). En segundo lugar, MLFQ quiere hacer que el sistema se sienta receptivo a los usuarios interactivos (es decir, los usuarios que se sientan y miran la pantalla, esperando a que finalice un proceso), y así minimizar el tiempo de respuesta; desafortunadamente, algoritmos como Round Robin reducen el tiempo de respuesta, pero son terribles para el tiempo de entrega. Entonces, nuestro problema: dado que en general no sabemos nada sobre un proceso, ¿cómo podemos construir un planificador para lograr estos objetivos? ¿Cómo puede el planificador aprender, mientras el sistema se está ejecutando, las características de los trabajos que está corriendo, y así tomar mejores decisiones de planificación?

LA CUESTIÓN: ¿CÓMO PROGRAMAR SIN UN PERFECTO CONOCIMIENTO?
¿Cómo podemos diseñar un planificador que minimice el tiempo de respuesta para trabajos interactivos y al mismo tiempo minimice el tiempo de entrega sin un conocimiento previo de la duración del trabajo?

TIP: APRENDER DE LA HISTORIA
La cola multinivel retroalimentada es un excelente ejemplo de un sistema que aprende del pasado para predecir el futuro. Estos enfoques son comunes en los sistemas operativos (y en muchos otros lugares de las Ciencias de la Computación, incluyendo los predictores de rama de hardware y los algoritmos de almacenamiento en caché). Estos enfoques funcionan cuando los trabajos tienen fases de comportamiento y, por lo tanto, son predecibles; por supuesto que hay que tener cuidado con tales técnicas, ya que fácilmente pueden ser incorrectas y hacer que un sistema tome peores decisiones de las que hubiese tomado sin ningún conocimiento.

8.1 MLFQ: Reglas Básicas

Para construir un planificador de este tipo, en este capítulo describiremos los algoritmos básicos detrás de una cola multinivel retroalimentada; si bien los detalles de muchas MLFQ implementadas pueden diferir [E95], la mayoría de los enfoques son similares.

En nuestro tratamiento, la MLFQ tiene varias **colas** distintas, a cada una de las cuales se le asigna un **nivel de prioridad** diferente. En un momento dado, cada trabajo que está listo para ejecutarse se encuentra en una sola cola. MLFQ usa las prioridades para decidir qué trabajo debe ejecutarse en un momento determinado: se elige para ser ejecutado un trabajo con una mayor prioridad (es decir, un trabajo en una cola más alta).

Por supuesto, puede haber más de un trabajo en una cola determinada y, por lo tanto, tener la misma prioridad. En este caso, simplemente usaremos planificación round-robin entre estos trabajos.

De esta forma, llegamos a las dos primeras reglas básicas para MLFQ:

- **Regla 1**: Si Prioridad(A) $>$ Prioridad(B), se ejecuta A (B no).
- **Regla 2**: Si Prioridad(A) = Prioridad(B), se ejecutan A y B en RR.

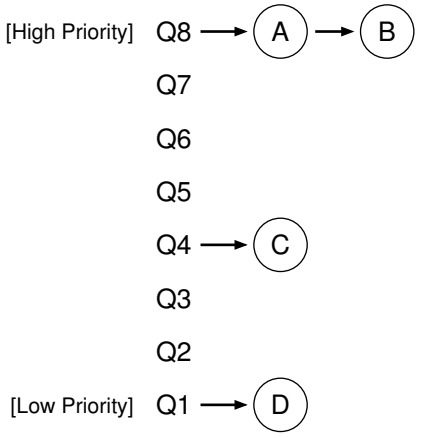

Figure 8.1: **Ejemplo de MLFQ**

Por lo tanto, la clave para la planificación MLFQ radica en cómo el planificador establece las prioridades. En lugar de dar una prioridad fija a cada trabajo, MLFQ varía la prioridad de un trabajo en función de su *comportamiento observado*. Si, por ejemplo, un trabajo renuncia repetidamente a la CPU mientras espera la entrada del teclado, MLFQ mantendrá su prioridad alta, ya que así es como podría comportarse un proceso interactivo. Si, en cambio, un trabajo usa la CPU de manera intensiva durante largos períodos de tiempo, MLFQ reducirá su prioridad. De esta manera, MLFQ intentará aprender sobre los procesos a medida que se ejecutan y, por lo tanto, utilizará el historial del trabajo para predecir su comportamiento *futuro*.

Si tuviéramos que presentar una imagen de cómo se verían las colas en un instante dado, podríamos ver algo como lo siguiente (Figura 8.1). En la figura, dos trabajos (A y B) están en el nivel de prioridad más alto, mientras que el trabajo C está en el medio y el trabajo D tiene la prioridad más baja. Dado nuestro conocimiento actual de cómo funciona MLFQ, el planificador simplemente alternaría segmentos de tiempo entre A y B porque son los trabajos de mayor prioridad en el sistema; los pobres trabajos C y D ni siquiera llegarían a funcionar, ¡un escándalo!

Por supuesto que, mostrar una foto estática de algunas colas no te da una idea real de cómo funciona MLFQ. Lo que necesitamos es comprender cómo cambia la prioridad de un trabajo con el tiempo. Y eso, para sorpresa de aquellos que estén leyendo un capítulo de este libro por primera vez, es exactamente lo que haremos a continuación.

8.2 Primer Intento: Cómo Cambiar la Prioridad

Ahora debemos decidir cómo MLFQ va a cambiar el nivel de prioridad de un trabajo (y por lo tanto, en qué cola se encuentra) durante su vida útil. Para esto, debemos tener en cuenta nuestra carga de trabajo: una combinación de trabajos interactivos que son de ejecución corta (y que con frecuencia pueden ceder la CPU) y algunos trabajos "CPU-bound" de ejecución más larga que necesitan de la CPU por mucho tiempo, pero donde el tiempo de respuesta no es importante. Aquí está nuestro primer intento de un algoritmo de ajuste de prioridad:

- **Regla 3**: Cuando un trabajo ingresa al sistema, se coloca en la prioridad más alta (la cola de más arriba).
- **Regla 4a**: Si un trabajo consume un segmento de tiempo completo mientras se ejecuta, su prioridad se reduce (es decir, se mueve una cola más abajo).
- **Regla 4b**: Si un trabajo cede la CPU antes de que finalice el segmento de tiempo, permanece en el *mismo* nivel de prioridad.

Ejemplo 1: Un Solo Trabajo de Larga Duración

Veamos algunos ejemplos. Primero, veremos qué sucede cuando ha habido un trabajo de larga duración en el sistema. La figura 8.2 muestra lo que le sucede a este trabajo a lo largo del tiempo en un planificador de tres colas.

Como se puede ver en el ejemplo, el trabajo ingresa con la prioridad más alta (Q2). Después de un solo segmento de tiempo de 10 ms, el planificador reduce la prioridad del trabajo en uno y, por lo tanto, el trabajo está en Q1. Después de ejecutarse en Q1 durante un segmento de tiempo, el trabajo finalmente se rebaja a la prioridad más baja del sistema (Q0), donde permanece. Bastante simple, ¿no?

Ejemplo 2: Y Entonces Vino un Trabajo Corto

Ahora veamos un ejemplo más complicado y, con suerte, veamos cómo MLFQ intenta aproximarse a SJF. En este ejemplo, hay dos trabajos: A, que es un trabajo intensivo de CPU de ejecución prolongada, y B, que es un trabajo interactivo de ejecución corta. Supongamos que A viene funcionando durante un tiempo y entonces llega B. ¿Qué va a pasar? ¿MLFQ se va a aproximar a SJF para B?

La Figura 8.3 traza los resultados de este escenario. A (mostrado en negro) está corriendo en la cola de menor prioridad (al igual que cualquier trabajo intensivo de CPU de larga ejecución); B (mostrado en gris) llega en el momento $T = 100$ y, por lo tanto, se inserta en la cola más alta; como su tiempo de ejecución es corto (solo 20 ms), B

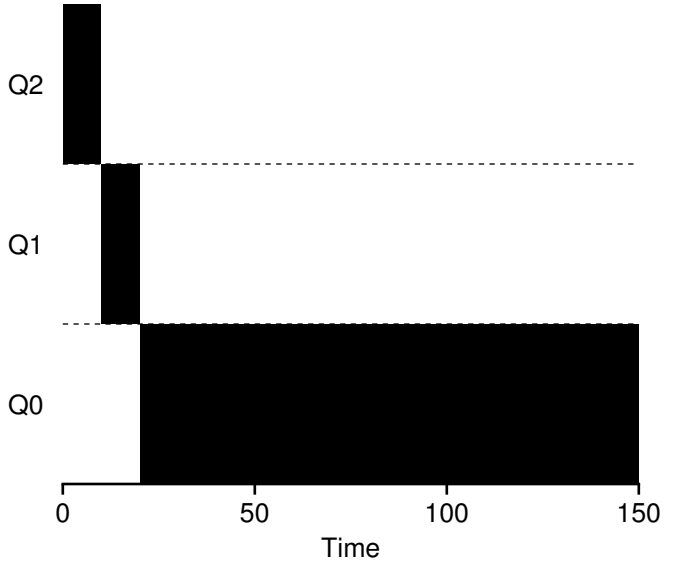

Figure 8.2: **Trabajo de Larga Duración a lo Largo del Tiempo**

termina antes de llegar a la cola inferior, en dos segmentos de tiempo; luego, A reanuda su ejecución (con una prioridad baja).

A partir de este ejemplo, es posible que entiendas uno de los principales objetivos del algoritmo: debido a que no sabe si un trabajo será un trabajo corto o un trabajo de larga duración, primero asume que podría ser un trabajo corto, dando así al trabajo una prioridad alta. Si en realidad es un trabajo corto, se ejecutará rápidamente y se completará; si no es un trabajo corto, se desplazará lentamente por las colas y, por lo tanto, pronto demostrará ser un proceso de larga duración más del tipo por lotes. De esta manera, MLFQ se aproxima a SJF.

Ejemplo 3: ¿Qué Pasa con la E/S?

Veamos ahora un ejemplo con algunas E/S. Según lo establecido por la regla 4b anteriormente, si un proceso abandona el procesador antes de usar su segmento de tiempo, lo mantenemos en el mismo nivel de prioridad. La intención de esta regla es simple: si un trabajo interactivo, por ejemplo, está haciendo una gran cantidad de E/S (por ejemplo, esperando la entrada del usuario desde el teclado o el ratón), abandonará la CPU antes de que se complete su segmento de

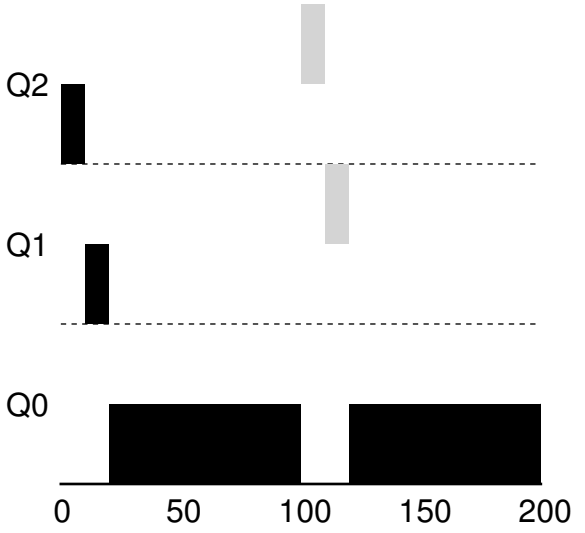

Figure 8.3: **Y Entonces Vino un Trabajo Interactivo**

tiempo; en tal caso, no queremos penalizar el trabajo y, por lo tanto, simplemente lo mantenemos en el mismo nivel.

La Figura 8.4 muestra un ejemplo de cómo funciona esto, con un trabajo interactivo B (mostrado en gris) que necesita la CPU solo durante 1 ms antes de realizar una E/S que compite por la CPU con un trabajo por lotes de larga duración A (mostrado en negro). El enfoque de MLFQ mantiene a B en la prioridad más alta pues B sigue liberando la CPU; si B es un trabajo interactivo, MLFQ logra aún más su objetivo de ejecutar trabajos interactivos rápidamente.

Problemas con nuestra MLFQ actual

Así, tenemos una MLFQ básica. Parece hacer un buen trabajo, compartiendo la CPU justamente entre trabajos de larga duración y dejando que los trabajos interactivos cortos o intensivos de E/S se ejecuten rápidamente. Lamentablemente, el enfoque que hemos desarrollado hasta ahora contiene graves defectos. ¿Se te ocurre alguno?

(*Acá es donde hacés una pausa y pensás de la forma más retorcida que se te ocurra*)

En primer lugar, tenemos el problema de la inanición: si hay "demasiados" trabajos interactivos en el sistema, se combinarán para consumir todo el tiempo de CPU, y por lo tanto los trabajos de larga duración nunca recibirán ningún tiempo de CPU (**mueren de ham-**

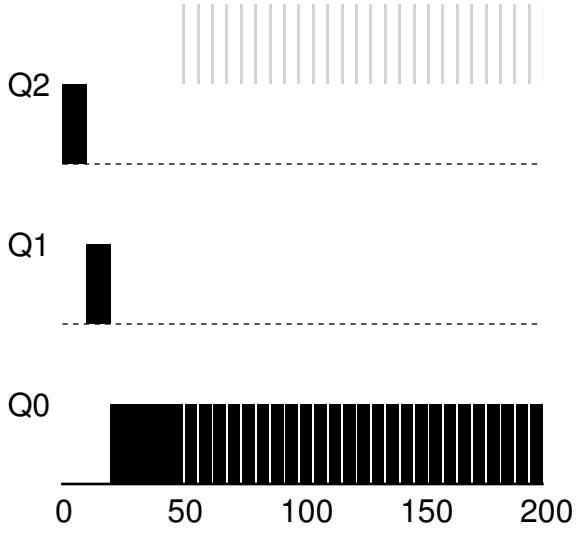

Figure 8.4: **Una carga de trabajo mixta con uso intensivo de E/S y de CPU**

bre). Nos gustaría hacer algunos progresos en estos trabajos, incluso en este escenario.

En segundo lugar, un usuario inteligente podría reescribir su programa para **engañar al planificador**. Engañar al planificador generalmente se refiere a la idea de hacer algo astuto para manipular al planificador a darte más de tu parte justa del recurso. El algoritmo que hemos descripto es susceptible al siguiente ataque: antes de que termine el segmento de tiempo, emití una operación de E/S (a algún archivo que no te importe), renunciando de esta forma a la CPU; hacer esto te permite permanecer en la misma cola y, por lo tanto, ganar un mayor porcentaje de tiempo de CPU. Si se hace bien (por ejemplo, ejecutando el 99% de un segmento de tiempo antes de ceder la CPU), un trabajo podría casi monopolizar la CPU.

Finalmente, un programa puede cambiar su comportamiento con el tiempo; lo que antes era CPU-bound puede pasar a una fase de interactividad. Con nuestro enfoque actual, tal trabajo no tendría suerte y no sería tratado como los otros trabajos interactivos en el sistema.

8.3 Segundo intento: El ascenso de prioridad

TIP: LA PROGRAMACIÓN DEBE SER SEGURA ANTE ATAQUES
Podrías llegar a pensar que una política de planificación, ya sea dentro del mismo SO (como se discute aquí), o en un contexto más amplio (por ejemplo, en el manejo de solicitudes de E/S de un sistema de almacenamiento distribuido [Y+18]), no es una preocupación de **seguridad**, pero en cada vez más casos, es exactamente eso. Considere un centro de datos moderno, en el que los usuarios de todo el mundo comparten CPUs, memorias, redes y sistemas de almacenamiento; sin cuidado en el diseño y la aplicación de políticas, un solo usuario puede dañar negativamente a los otros y obtener ventajas para sí mismo. Por lo tanto, la política de planificación forma una parte importante de la seguridad de un sistema, y debe ser construida cuidadosamente.

Tratemos de cambiar las reglas y veamos si podemos evitar el problema de la inanición. ¿Qué podríamos hacer para garantizar que los trabajos CPU-bound progresen (incluso si no es por mucho)?.

La idea simple sería **elevar** periódicamente la prioridad de todos los trabajos en el sistema. Hay muchas maneras de lograr esto, pero hagamos algo simple: tirarlos a todos en la cola superior; por lo tanto, una nueva regla:

- **Regla 5**: Después de un período de tiempo determinado S, mover todos los trabajos del sistema a la cola más alta.

Nuestra nueva regla resuelve dos problemas a la vez. En primer lugar, garantiza que los procesos no mueran de hambre: al quedarse en la cola superior, un trabajo compartirá la CPU con otros trabajos de alta prioridad al estilo round-robin y así, con el tiempo, recibirá servicio. En segundo lugar, si un trabajo CPU-bound se vuelve interactivo, el planificador lo tratará correctamente una vez que haya recibido el ascenso de prioridad.

Veamos un ejemplo. En este escenario, solo mostramos el comportamiento de un trabajo de ejecución larga cuando competimos por la CPU con dos trabajos interactivos de ejecución corta. En la Figura 8.5 se muestran dos gráficos. En la izquierda, no hay un ascenso de prioridad y, por lo tanto el trabajo de larga duración se muere de hambre una vez que llegan los dos trabajos cortos; en la derecha, hay un ascenso de prioridad cada 50 ms (que probablemente sea un valor demasiado pequeño, pero se utiliza aquí para el ejemplo) y, por lo tanto, al menos garantizamos que el trabajo de larga duración progresará, obteniendo un ascenso a la máxima prioridad cada 50 ms y así llegar a ejecutarse periódicamente.

Por supuesto, la adición del período de tiempo S conduce a la pregunta obvia: ¿A qué valor se debe establecer S? John Ousterhout, un

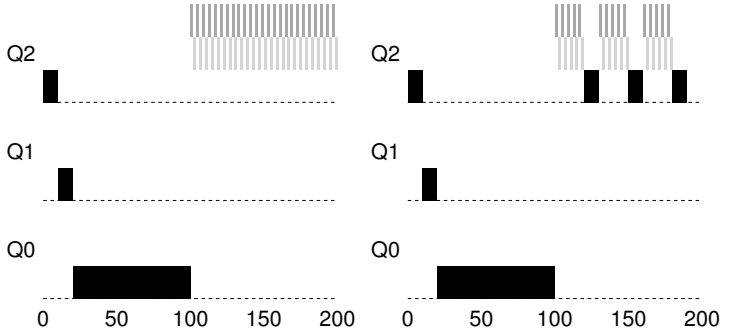

Figure 8.5: **Sin (izquierda) y con (derecha) ascenso de prioridad**

reconocido investigador de sistemas [O11], solía llamar tales valores en los sistemas **constantes vudú**, porque parecían requerir alguna forma de magia negra para establecerlos correctamente. Desafortunadamente, S tiene esta característica. Si se le asigna un valor demasiado alto, los trabajos de larga duración podrían morir de hambre; demasiado bajo, y es posible que los trabajos interactivos no obtengan una parte adecuada de la CPU.

8.4 Tercer intento: Mejor contabilidad

Ahora tenemos un problema más que resolver: ¿Cómo evitar los engaños a nuestro planificador? Las verdaderas culpables, como habrás adivinado, son las reglas 4a y 4b, que permiten que un trabajo conserve su prioridad al renunciar a la CPU antes de que expire el segmento de tiempo. ¿Entonces, qué debemos hacer?

La solución es realizar una mejor **contabilidad** del tiempo de la CPU en cada nivel de la MLFQ. En lugar de olvidarse de la porción del segmento tiempo que un proceso usa en un nivel dado, el planificador debería realizar un seguimiento; una vez que un proceso haya utilizado todo su tiempo asignado, es degradado a la siguiente cola de prioridad. No importa si utiliza el segmento de tiempo en una ráfaga larga o en muchas pequeñas. Por lo tanto, reescribimos las Reglas 4a y 4b a la siguiente y única regla:

- **Regla 4**: Una vez que un trabajo utilice su tiempo asignado en un nivel dado (independientemente de cuántas veces haya renunciado a la CPU), su prioridad se reduce (es decir, se mueve una cola hacia abajo).

Veamos un ejemplo. La Figura 8.6 muestra lo que sucede cuando una carga de trabajo intenta engañar al planificador con las antiguas

reglas 4a y 4b (a la izquierda), así como la nueva regla 4 anti-engaños. Sin ninguna protección contra los engaños, un proceso puede emitir una E/S justo antes de que finalice un segmento de tiempo y así dominar el tiempo de la CPU. Con las respectivas protecciones en su lugar, independientemente del comportamiento de E/S del proceso, se desplaza lentamente por las colas y, por lo tanto, no puede ganar una parte injusta de la CPU.

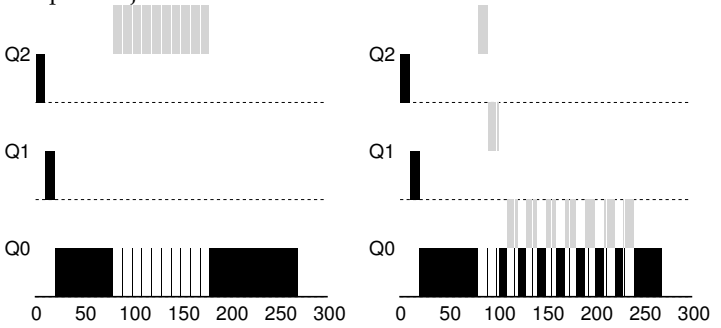

Figure 8.6: **Sin (izquierda) y con (derecha) Protección Contra Engaños**

TIP: EVITAR LAS CONSTANTES VUDÚ (LEY DE OUSTERHOUT)
Evitar constantes vudú es una buena idea siempre que sea posible. Desafortunadamente, como en el ejemplo anterior, a menudo es difícil. Se podría intentar que el sistema aprenda un buen valor, pero eso tampoco es sencillo. El resultado frecuente: un archivo de configuración lleno de valores de parámetros predeterminados que un administrador experimentado puede modificar cuando algo no funciona correctamente. Como podés imaginar, estos a menudo no se modifican y, por lo tanto, tenemos la esperanza de que los valores predeterminados funcionen bien en el campo. Este consejo es traído por nuestro antiguo profesor de SO, John Ousterhout, y por eso lo llamamos la **Ley de Ousterhout**.

8.5 Ajustando la MLFQ y Otros Problemas

Hay algunos otros problemas que surgen con la planificación con MLFQ. Una gran pregunta es cómo **parametrizar** dicho planificador. Por ejemplo, ¿cuántas colas debería haber? ¿Qué tan grande debe ser el segmento de tiempo por cola? ¿Con qué frecuencia se debe elevar la prioridad para evitar la inanición y tener en cuenta los cambios de comportamiento? No hay respuestas fáciles a estas preguntas y, por

lo tanto, solo un poco de experiencia con las cargas de trabajo y el ajuste posterior del planificador conducirán a un equilibrio satisfactorio.

Por ejemplo, la mayor parte de las variantes de MLFQ permiten variar la duración del segmento de tiempo en diferentes colas. A las colas de alta prioridad se les suele asignar intervalos de tiempo breves; después de todo, se componen de trabajos interactivos y, por lo tanto, tiene sentido alternar rápidamente entre ellos (por ejemplo, 10 milisegundos o menos). Las colas de baja prioridad, por el contrario, contienen trabajos CPU-bound de larga ejecución; por lo tanto, los segmentos de tiempo más largos funcionan mejor (por ejemplo, cientos de ms). La figura 8.7 (página 8) muestra un ejemplo en el que dos trabajos se ejecutan durante 20 ms en la cola más alta (con un segmento de tiempo de 10 ms), 40 ms en el medio (segmento de tiempo de 20 ms) y con un segmento de tiempo de 40 ms en el nivel más bajo.

La implementación de MLFQ de Solaris (el tipo de planificación de tiempo compartido, o TS) es particularmente fácil de configurar; proporciona un conjunto de tablas que determinan exactamente cómo se modifica la prioridad de un proceso a lo largo de su vida útil, cuánto dura cada segmento de tiempo y con qué frecuencia elevar la prioridad de un trabajo [AD00]; un administrador puede jugar con esta tabla para que el planificador se comporte de diferentes maneras. Los valores predeterminados para la tabla son 60 colas, con longitudes de segmento de tiempo que aumentan lentamente desde los 20 milisegundos (prioridad más alta) a unos cientos de milisegundos (la más baja), y las prioridades se elevan más o menos cada 1 segundo.

Otros planificadores MLFQ no usan una tabla ni las reglas exactas descritas en este capítulo; más bien ajustan las prioridades usando fórmulas matemáticas. Por ejemplo, el planificador FreeBSD (versión 4.3) usa una fórmula para calcular el nivel de prioridad actual de un trabajo, basándose en cuánta CPU ha usado el proceso [LM+89]; además, el uso decae con el tiempo, proporcionando el ascenso de prioridad deseado de una manera diferente a la descrita en este documento. Consultá el artículo de Epema para obtener una excelente descripción general de dichos algoritmos de descomposición y sus propiedades [E95].

Finalmente, muchos planificadores tienen algunas otras características que podés llegar a encontrar. Por ejemplo, algunos planificadores reservan los niveles de prioridad más altos para los trabajos del sistema operativo; por lo tanto, los trabajos de usuario típicos no pueden obtener nunca los niveles más altos de prioridad en el sistema. Algunos sistemas también permiten algunos consejos del usuario para ayudar a establecer las prioridades; por ejemplo, con la utilidad de línea de comandos `nice`, se puede aumentar o disminuir la prioridad de un trabajo (más o menos) y así aumentar o disminuir

sus posibilidades de ejecución en un determinado momento. Consultá la página del manual para obtener más información.

TIP: USAR CONSEJOS CUANDO SEA POSIBLE
Dado que el sistema operativo rara vez sabe qué es lo mejor para todos y cada uno de los procesos del sistema, a menudo es útil proporcionar interfaces que permitan a los usuarios o administradores proporcionar algunas **sugerencias** sobre el sistema operativo. A menudo llamamos consejos a estas sugerencias, ya que el sistema operativo no necesariamente debe prestarles atención, pero puede tenerlas en cuenta para tomar una mejor decisión. Tales sugerencias son útiles en muchas partes del sistema operativo, incluyendo el planificado (por ejemplo, con `nice`), el administrador de memoria (por ejemplo, `madvise`) y el sistema de archivos (por ejemplo, precarga informada y almacenamiento en caché [P+95]).

8.6 Resumen de MLFQ

Hemos descripto un enfoque de planificación conocido como la Cola de Multinivel Retroalimentada (MLFQ). Con suerte, ahora podrás ver por qué se llama así: tiene mútiples niveles de colas, y utiliza la retroalimentación para determinar la prioridad de un trabajo determinado. La historia es tu guía: prestá atención a cómo se comportan los trabajos a lo largo del tiempo y tratalos como corresponde.

El conjunto de las reglas de MLFQ refinadas, distribuidas a lo largo del capítulo, serán reproducidas a continuación por placer visual:

- **Regla 1**: Si Prioridad(A) $>$ Prioridad(B), se ejecuta A (B no).
- **Regla 2**: Si Prioridad(A) = Prioridad(B), se ejecutan A y B en RR.
- **Regla 3**: Cuando un trabajo ingresa al sistema, se coloca en la prioridad más alta (la cola de más arriba).
- **Regla 4**: Una vez que un trabajo utilice su tiempo asignado en un nivel dado (independientemente de cuántas veces haya renunciado a la CPU), su prioridad se reduce (es decir, se mueve una cola hacia abajo).
- **Regla 5**: Después de un período de tiempo determinado S, mover todos los trabajos del sistema a la cola más alta.

MLFQ es interesante por la siguiente razón: en lugar de exigir un conocimiento *a priori* de la naturaleza de un trabajo, observa la ejecución de un trabajo y lo prioriza según corresponda. De esta manera, se las arregla para obtener lo mejor de ambos mundos: puede ofrecer un excelente rendimiento general (similar a SJF/STCF) para trabajos interactivos de ejecución corta, y es justo y realiza progreso para las cargas de trabajo intensivas en CPU de ejecución prolongada. Por esta razón, muchos sistemas, incluidos los derivados de BSD UNIX [LM+89, B86], Solaris [M06] y Windows NT y los subsiguientes sistemas operativos de Windows [CS97] utilizan alguna forma de MLFQ como su planificador base.

8.7 Referencias

[AD00] "Multilevel Feedback Queue Scheduling in Solaris" por Andrea Arpaci-Dusseau. Disponible en: `http://www.ostep.org/Citations/notes-solaris.pdf`. *Un gran conjunto de notas breves sobre los detalles del planificador de Solaris escritas por uno de los autores. Está bien, puede que esta descripción esté sesgada, pero las notas son bastante buenas.*

[B86] "The Design of the UNIX Operating System" por M.J. Bach. Prentice-Hall, 1986. *Uno de los libros clásicos antiguos sobre cómo se construye un sistema operativo UNIX real; una lectura obligatoria para los hackers del kernel.*

[C+62] "An Experimental Time-Sharing System" por F. J. Corbato, M. M. Daggett, R. C. Daley. IFIPS 1962. *Es un poco difícil de leer, pero es la fuente de muchas de las primeras ideas en la planificación multinivel retroalimentada. Gran parte de esto pasó posteriormente a Multics, que se podría argumentar que fue el sistema operativo más influyente de todos los tiempos.*

[CS97] "Inside Windows NT" por Helen Custer y David A. Solomon. Microsoft Press, 1997. *El libro de la NT, si querés aprender algo más que UNIX. Por supuesto, ¿por qué lo harías? Está bien, estamos bromeando; puede que algún día trabajes para Microsoft.*

[E95] "An Analysis of Decay-Usage Scheduling in Multiprocessors" por D.H.J. Epema. SIGMETRICS '95. *Un buen artículo sobre el estado del arte de la planificación a mediados de la década de 1990, que incluye una buena descripción general del acercamiento básico detrás de los planificadores de descomposición.*

[LM+89] "The Design and Implementation of the 4.3BSD UNIX Operating System" por S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman. Addison-Wesley, 1989. *Otro clásico de SOs, escrito por cuatro de las personas principales detrás de BSD. Las versiones posteriores de este libro, aunque están más actualizadas, no coinciden con la belleza de este.*

[M06] "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture" por Richard McDougall. Prentice-Hall, 2006. *Un buen libro sobre Solaris y su funcionamiento.*

[O11] "La página de inicio de John Ousterhout" por John Ousterhout. www.stanford.edu/~ouster/. *La página de inicio del famoso profesor Ousterhout. Los dos coautores de este libro tuvieron el placer de hacer el posgrado en sistemas operativos con Ousterhout; de hecho, la escuela de posgrado fue en donde los dos coautores se conocieron, lo que eventualmente los llevó al matrimonio, a los hijos e incluso a este libro. Por lo tanto, realmente deberías culpar a Ousterhout por todo este lío en el que estás metido.*

[P+95] "Informed Prefetching and Caching" por R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka. SOSP '95, Copper Mountain, Colorado, Octubre de 1995. *Un artículo divertido sobre algunas ideas geniales en sistemas de archivos, incluyendo cómo las aplicaciones*

pueden aconsejar al SO sobre a qué archivos se están accediendo y cómo se planea acceder a ellos.

[Y+18] "Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models" por Suli Yang, Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. ArpaciDusseau. OSDI '18, San Diego, California. *Un trabajo reciente de nuestro grupo que demuestra la dificultad de planificar solicitudes de E/S dentro de sistemas modernos de almacenamiento distribuido como Hive/HDFS, Cassandra, MongoDB y Riak. Si no se tiene cuidado, un solo usuario podría monopolizar los recursos del sistema.*

8.8 Tarea (Simulación)

Este programa, `mlfq.py`, te permite ver cómo se comporta el planificador MLFQ presentado en este capítulo. Consultá el archivo README para obtener más detalles.

Preguntas

1. Ejecutá algunos problemas generados aleatoriamente con solo dos trabajos y dos colas; calculá la traza de ejecución de MLFQ para cada uno. Limitá la duración de cada trabajo y desactivá las E/S para hacer tu vida más fácil.
2. ¿Cómo ejecutarías el planificador para reproducir cada uno de los ejemplos en el capítulo?
3. ¿Cómo configurarías los parámetros del planificador para que se comporten como un planificador round-robin?
4. Creá una carga de trabajo con dos trabajos y con parámetros del planificador de modo que un trabajo aproveche las antiguas reglas 4a y 4b (activadas con la bandera -S) para engañar el planificador y obtener el 99% de la CPU en un intervalo de tiempo determinado.
5. Dado un sistema con una longitud de quantum de 10 ms en su cola más alta, ¿con qué frecuencia tendrías que elevar los trabajos al nivel de prioridad más alto (con la bandera -B) para garantizar que un solo trabajo de ejecución prolongada (y potencialmente hambriento) obtenga al menos el 5% de la CPU?
6. Una pregunta que surge en la planificación es en qué extremo de la cola hay que agregar un trabajo que acaba de terminar E/S; la bandera -I cambia este comportamiento para este simulador de planificación. Jugá con distintas cargas de trabajo y fijate si podés ver el efecto de esta bandera.