UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

INTEGRANTES:

- UGALDE VELASCO ARMANDO
- SANTIAGO GUTIÉRREZ DIEGO

PLANIFICACIÓN JUSTA: LOTERÍA Y LINUX COMPLETELY FAIR SCHEDULER

SEMESTRE 2022-1

ÍNDICE

Planificación justa: Lotería y Linux Completely Fair Scheduler

Existen algoritmos de planificación que buscan optimizar ciertas métricas, como el tiempo de respuesta o el tiempo de reconversión. Sin embargo, también es posible que se busque garantizar que cada proceso en el sistema obtenga cierto porcentaje de tiempo de ejecución en el CPU. A este enfoque se le conoce como planificación de partes justas o proporcionales.

Planificación de lotería

Un excelente ejemplo de planificación de partes justas es la planificación de lotería, encontrada en investigación realizada por *Waldspurger* y *Weihl*. La idea es bastante simple: cada cierto tiempo se debe escoger un boleto de lotería para determinar qué proceso va a ejecutarse después; los procesos que deben ejecutarse más seguido deben de tener más oportunidades de ganar la lotería.

La planificación de lotería se basa principalmente en un concepto muy simple ya mencionado: los **boletos**, que se utilizan para representar la parte de cierto recurso que un proceso (o usuario, o cualquier otra entidad) debe recibir. El porcentaje de boletos que un proceso tiene representa su parte correspondiente del recurso en cuestión.

Por ejemplo, asumamos que se tienen dos procesos: *A* y *B*, y que *A* cuenta con *75* boletos, mientras que *B* solamente tiene *25*. Entonces, en este caso, lo que se desea es que *A* reciba el *75%* del tiempo del CPU, mientras que *B* el *25%* restante. La planificación de lotería logra esta distribución de forma probabilística (pero no determinística) al “jugar” la lotería cada cierto tiempo. Jugar a la lotería es sencillo: el planificador debe saber cuántos boletos existen en total (en el ejemplo existen 100), y posteriormente escoger un boleto ganador, que, en este caso, es un número de 0 a 99. Asumiendo que A tiene los boletos *0* a *74*, y *B* los boletos *75* a *99*, el boleto ganador simplemente determina si *A* o *B* se ejecutará después.

A continuación, se muestra una salida ejemplo de los boletos ganadores de la lotería:

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49
```

Tomando en cuenta los boletos correspondientes a los procesos *A* y *B*, la planificación resultaría de la siguiente forma:

```
A    A A    A A A A A A    A    A A A A A
  B       B                B    B
```

Como se puede observar, la utilización de aleatorización en la planificación de lotería resulta en un algoritmo correcto probabilísticamente que cumple con la proporción indicada. Sin embargo, lo anterior no se encuentra garantizado. En el ejemplo anterior, el proceso *B* solamente se ejecuta 4 de 20 partes del tiempo *(20%),* en lugar del *25%*

deseado. Sin embargo, mientras mayor tiempo se ejecuten estos procesos, aumentará la posibilidad de que ambos logren tener los porcentajes deseados.

Mecanismos de boletos

Moneda

La planificación de lotería también nos provee de un número de mecanismos para manipular los boletos en formas útiles. Una de éstas es el concepto de la *"moneda",* que permite que un usuario con cierto número de boletos reparta los mismos entre sus propios procesos, utilizando cualquier *moneda* que elija; el sistema después convierte automáticamente dicha moneda al valor correcto global.

Por ejemplo, asumamos que los usuarios *A* y *B* tienen *100* boletos cada uno. El usuario A se encuentra ejecutando dos procesos: *A1* y *A2*, y le da *500* boletos a cada uno (de 1000 en total) en la moneda de *A*. El usuario *B* se encuentra ejecutando únicamente un proceso, y le da *10* boletos (de *10* totales) en la moneda de B. El sistema después convierte la distribución de *500* boletos de *A1* y A2, dados en la moneda de A, a *50* boletos en la moneda global, para cada uno. Dado que el usuario B sólo se encuentra ejecutando un proceso y éste tiene todos los boletos, el sistema le asignará los 100 boletos correspondientes a B, en la moneda global. La lotería se juega finalmente en la moneda global (200 boletos en total).

```
User A -> 500 (A's currency) to A1 ->  50 (global currency)
       -> 500 (A's currency) to A2 ->  50 (global currency)
User B ->  10 (B's currency) to B1 -> 100 (global currency)
```

Transferencia

Mediante este mecanismo, un proceso puede *"prestarle"* temporalmente un número de sus boletos a otro. Esta capacidad es principalmente útil en un ambiente cliente-servidor, donde un proceso cliente envía un mensaje a un servidor, pidiéndole que realice cierto trabajo para él. Para acelerar el trabajo, el cliente puede pasar sus boletos al servidor para tratar de maximizar su rendimiento, mientras éste está procesando la solicitud del cliente. Cuando termina, el servidor transfiere de nuevo los boletos al cliente, regresando el estado a la normalidad.

Inflación

Utilizando este mecanismo, un proceso puede aumentar o disminuir temporalmente el número de boletos que tiene. Por supuesto, en un escenario competitivo con procesos que no confían entre sí, esto no tiene sentido: un proceso voraz puede aumentar a un número grande de boletos y monopolizar el CPU. En lugar de lo anterior, la inflación puede utilizarse en un ambiente donde un grupo de procesos confían entre sí: en dicho caso, si algún proceso sabe que necesita más recursos, puede aumentar el valor de sus

boletos a un número razonable como una forma de reflejar su necesidad al sistema, sin la necesidad de comunicarse con otros procesos.

Implementación

Probablemente, el aspecto más fascinante de la planificación de lotería es la simplicidad de su implementación. Lo único que se requiere es un generador de números aleatorios para escoger el boleto ganador, una estructura de datos para almacenar los procesos en el sistema (es decir, una lista), y el número total de boletos.

Asumamos que almacenamos los procesos en una lista. A continuación, se muestra un ejemplo compuesto de tres procesos: *A, B y C*, cada uno con cierto número de boletos:

Para tomar una decisión de planificación, primero se debe elegir un número aleatorio (el del boleto ganador) del total de boletos (400, en este caso). Asumamos que se eligió el número 300. Entonces, simplemente recorremos la lista, utilizando un contador para auxiliarnos en encontrar el boleto ganador.

En el código, se puede observar que se recorre la lista de procesos, sumando el número de boletos de cada proceso en el que se encuentra el puntero, hasta que el valor del contador excede *winner* (el número de boleto ganador). Con nuestro ejemplo del boleto ganador 300, lo siguiente toma lugar. Primero, *counter* es incrementado a 100 para contar los boletos de A; debido a que 100 es menor que 300, el ciclo continúa. Después, *counter* debe ser actualizado a 150 (incrementando los boletos de B), lo cual sigue siendo menor a 300, y, por lo tanto, se continúa ejecutando el ciclo. Finalmente, *counter* se actualiza a 400 (claramente mayor a 300), y, por lo tanto, se deja de ejecutar el ciclo con la variable *current* apuntando a C, el ganador.

Para llevar a cabo este procedimiento de forma más eficiente, generalmente podría ser mejor ordenar la lista, empezando con el mayor valor de boletos al menor. El orden no afecta la validez del algoritmo, sin embargo, nos asegura que se realizará el menor número de iteraciones posibles al buscar el proceso ganador.

```
1  // counter: used to track if we've found the winner yet
2  int counter = 0;
3
4  // winner: use some call to a random number generator to
5  //         get a value, between 0 and the total # of tickets
6  int winner = getrandom(0, totaltickets);
7
8  // current: use this to walk through the list of jobs
9  node_t *current = head;
10 while (current) {
11     counter = counter + current->tickets;
12     if (counter > winner)
13         break; // found the winner
14     current = current->next;
15 }
16 // 'current' is the winner: schedule it...
```

¿Cómo asignar boletos?

Un problema fundamental que no se ha discutido es la forma en la que se asignan boletos a los procesos. Éste es un problema difícil, ya que el comportamiento del sistema depende de la forma en la que los boletos son distribuidos. Un enfoque es asumir que los usuarios saben qué hacer; en tal caso, a cada usuario se le asigna cierto número de boletos, y cada uno los distribuye entre cada uno de sus procesos como desea. Sin embargo, esta solución realmente no define qué hacer exactamente. Por lo tanto, dado un número de procesos, el problema de asignación de boletos queda abierto.

¿Por qué no utilizar un enfoque determinístico?

Como ya se observó, la aleatorización nos permite implementar de forma sencilla un planificador aproximadamente correcto. Sin embargo, ocasionalmente no producirá planificaciones en las proporciones exactas deseadas, especialmente en periodos de tiempo cortos. Por esta razón, Waldspurger inventó la planificación de zancadas, un algoritmo de planificación justo y determinístico.

La planificación de zancadas también es sencilla de comprender: cada proceso en el sistema tiene un **tamaño de zancada**, que es inversamente proporcional al número de boletos que tiene. Cada que un proceso se ejecuta, se incrementará un contador asignado a éste (cuyo nombre es **pasos**) por el tamaño de su zancada para llevar la cuenta de su progreso global.

El planificador entonces utiliza el *tamaño de zancada* y los *pasos* que llevan los procesos para determinar cuál será ejecutado a continuación. La idea es simple: en cierto tiempo,

se debe escoger el proceso que tenga el menor valor de pasos, y, cuando se ejecuta un proceso, se debe aumentar su contador de pasos por el tamaño de su zancada.

Por ejemplo, asumamos que se tienen tres procesos: *A, B y C*, con valores de zancada de 100, 200 y 40, respectivamente, y con valores de pasos iniciales de 0. Entonces, al principio, cualquiera de los procesos puede correr, ya que sus valores de pasos son iguales. Asumamos que se escoge *A* de forma arbitraria. A se ejecuta; cuando termina su parte de tiempo, se actualiza su valor de pasos a 100. Después, B se ejecuta, cuyo valor de pasos se actualiza a 200. Finalmente, se ejecuta C, cuyo valor de pasos se incrementa a 40. En este punto, el algoritmo escogerá el valor de pasos mínimo, que es el de C, y lo ejecutará, actualizando su valor de pasos a 80. Después, C se ejecutará de nuevo, aumentando su valor a 120. A se ejecutará ahora, aumentando su valor a 200 (igual al de B). Después, C se ejecutará dos veces más, aumentando su valor de pasos a 160 y luego a 200. En este punto, todos los valores de pasos son iguales de nuevo, y el proceso se repetirá de forma infinita. A continuación, se muestra una figura que muestra el comportamiento explicado:

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Como se puede observar, C se ejecutó 5 veces, A dos veces y B solamente una, exactamente en proporción a sus valores de boletos de 250, 100 y 50, respectivamente. Mientras que la planificación de lotería logra estas proporciones de forma probabilística conforme pasa el tiempo, la planificación de zancadas logra las proporciones de forma exacta.

Entonces, dada la precisión de la planificación de zancadas, ¿por qué utilizar la planificación de lotería? El enfoque de la lotería tiene una propiedad interesante que su contraparte no: no es necesario tomar en cuenta un estado global. Supongamos que un nuevo proceso entra en la mitad del ejemplo anterior mostrado: ¿cuál debe ser su valor de pasos? ¿Debería ser cero? En este caso, monopolizaría el CPU hasta que alcance el valor de los demás procesos, que pueden ser muy grandes. Utilizando la lotería, no se mantiene un estado global por proceso, simplemente se agrega un nuevo proceso a la lista con el número de boletos que tiene, y se actualiza el número global de boletos

existentes. De esta forma, es más sencillo agregar un nuevo proceso utilizando planificación de lotería, y es un ejemplo de muchos en los que es más conveniente evitar el mantenimiento de estado global o utilizar aleatorización a cambio de cierta precisión en el algoritmo.

El Linux Completely Fair Scheduler (CFS)

A pesar de los algoritmos antiguos discutidos, el enfoque actual de Linux logra metas similares de una forma alternativa. El planificador, llamado el *Completely Fair Scheduler*, implementa planificación de partes justas, pero de una forma muy eficiente y escalable.

Para cumplir con sus metas de eficiencia, el CFS se enfoca en gastar muy poco tiempo realizando decisiones de planificación, aprovechando su diseño inherente y la utilización de ciertas estructuras de datos apropiadas para las tareas pertinentes. Estudios recientes han demostrado que la eficiencia de los planificadores es sumamente importante; específicamente, en un estudio de los centros de datos de Google, por Kanev et al., se demuestra que incluso después de numerosas optimizaciones, la planificación representa aproximadamente el 5% de toda la utilización del CPU en los centros de datos. Reducir este tiempo tanto como sea posible es, entonces, una meta clave en la arquitectura de planificadores modernos.

Operación básica

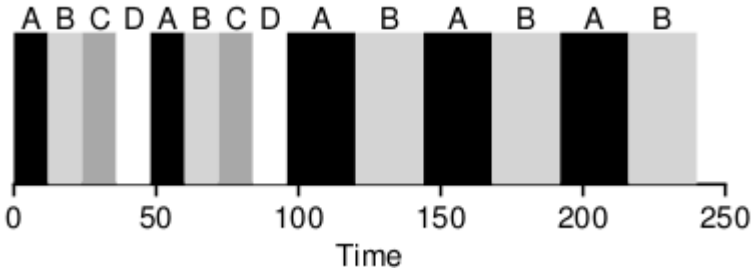
Mientras que la mayoría de planificadores están basados en el concepto de un ***"pedazo"*** fijo de tiempo, el CFS opera de forma distinta. Su objetivo es simple: dividir de forma justa la utilización de un CPU entre todos los procesos competidores. Realiza la tarea anterior mediante una técnica simple basada en conteo conocida como ***virtual runtime (vruntime)***.

Cada que un proceso se ejecuta, acumula *vruntime*. En el caso más básico, el *vruntime* de cada proceso aumenta en la misma proporción, acorde al tiempo físico o real. Cuando una decisión de planificación ocurre, el CFS escoge el proceso con el *vruntime* más bajo, de forma muy similar a la operación de los algoritmos ya discutidos.

Entonces, ¿cómo es que el planificador sabe cuándo parar el proceso ejecutándose actualmente, y ejecutar el siguiente? Existe un dilema: si CFS cambia de proceso muy pronto, será más justo, ya que se asegurará de que cada proceso reciba su parte del CPU en partes muy pequeñas de tiempo, pero a costa de un rendimiento disminuido (muchos cambios de contexto); si CFS cambia de procesos menos seguido, el rendimiento mejora (menos cambios de contexto), pero a costa de ser menos justo a corto plazo.

CFS maneja esta situación a través de varios parámetros de control. El primero es **sched_latency.** CFS utiliza este valor para determinar cuánto tiempo debe ejecutarse un proceso antes de considerar un cambio (efectivamente determinando su parte de tiempo, pero de forma dinámica). Un valor típico de *sched_latency* es de ***48*** milisegundos; CFS divide este valor entre el número *(n)* de procesos ejecutándose en el CPU para determinar la parte de tiempo para un proceso, y, por lo tanto, se asegura de que, en este periodo de tiempo, el algoritmo sea justo (ya que a cada proceso le corresponde una parte proporcional del CPU).

Por ejemplo, si hay *n = 4* procesos ejecutándose, CFS divide el valor de *sched_latency* entre n, resultando en pedazos de 12 milisegundos por proceso. CFS después planifica el primer proceso y lo ejecuta hasta que utiliza 12 milisegundos de tiempo de ejecución virtual (virtual runtime), y luego comprueba si hay un proceso con menor *vruntime* para ejecutarlo en su lugar. En este caso, sí existe, por lo que CFS podría cambiar a cualquiera de los otros procesos, y así sucesivamente. La siguiente figura muestra un ejemplo donde los cuatro procesos (A, B, C y D) se ejecutan cada uno por dos pedazos de tiempo de esta forma; dos de ellos completan su ejecución (C, D), dejando únicamente dos restantes, que se ejecutan cada uno por *24 ms* de forma “round robin”.



Sin embargo, ¿qué pasa si hay muchos procesos ejecutándose al mismo tiempo? Eso provocaría que las partes de tiempo fueran muy pequeñas, y a su vez causaría que existieran muchos cambios de contexto. Para resolver este problema, CFS agrega otro parámetro: ***min_granularity (granularidad mínima)***, al que usualmente se le asigna un valor cercano a ***6 ms***. CFS nunca le asignará un valor a un pedazo de tiempo menor a este valor, asegurándose de que no se gaste mucho tiempo en el *overhead* de planificación.

Por ejemplo, si existen diez procesos ejecutándose, nuestro cálculo original dividiría *sched_latency* entre 10 para determinar los pedazos de tiempo, resultando en 4.8 ms. Sin embargo, gracias a *min_granularity*, CFS asignaría las partes de tiempo a 6 ms en lugar de 4.8. A pesar de que CFS no será perfectamente justo durante el periodo de sched_latency, se encontrará cercano, mientras logra buena eficiencia del CPU.

Es importante notar que CFS utiliza un temporizador de interrupciones periódicas, lo cual significa que únicamente tomará decisiones en intervalos fijos de tiempo. Estas interrupciones toman lugar frecuentemente (por ejemplo, cada 1 ms), dándole la oportunidad a CFS de despertar y determinar si el proceso actual ha llegado al fin de su

parte del tiempo de ejecución. Si un proceso tiene una parte de tiempo que no es un múltiplo perfecto del intervalo de interrupciones, no hay ningún problema, ya que CFS lleva cuenta del vruntime de forma precisa, lo cual significa que, a lo largo del tiempo, eventualmente aproximará un compartido ideal del CPU.

Ponderación (Niceness)

CFS también permite cierto control sobre la prioridad de los procesos, permitiendo a los usuarios o administradores dar a los procesos un porcentaje mayor del CPU. Lo anterior no se realiza mediante boletos, sino mediante un mecanismo clásico de UNIX conocido como el nivel **nice** de un proceso. El parámetro *nice* puede tomar valores desde ***–20*** hasta ***+19***, con un valor por default de ***0***. Los valores positivos implican menor prioridad y los negativos implican mayor prioridad: cuando se es muy *bueno* (nice), no se obtiene mucha atención (de planificación).

CFS mapea los valores *nice* de cada proceso a cierto peso, como se muestra en la siguiente imagen:

```
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,
};
```

Estos pesos nos permiten calcular las partes de tiempo efectivas de cada proceso (de la misma forma en que se realizó anteriormente), pero ahora tomando en cuenta sus diferencias en prioridad. La fórmula utilizada para realizar lo anterior es la siguiente:

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency}$$

A continuación, se muestra un ejemplo para analizar el funcionamiento de la fórmula anterior. Asumamos que se tienen dos procesos: *A* y *B*. A es un proceso importante, por lo que se le da un valor alto de prioridad, asignándole un valor nice de *–5*. B no es tan importante, por lo que solamente tiene un valor nice por default de 0. Esto significa que, según la tabla de pesos, el peso de A será *3121* y el de B será *1024*. Si se calculan las partes de tiempo correspondientes a cada proceso, se obtendrá que la de A es

aproximadamente ¾ de sched_latency (es decir, 36 ms), mientras que la de B es aproximadamente de ¼ de sched_latency (es decir, 12 ms).

Además de generalizar el cálculo de las partes de tiempo, la forma en la que CFS calcula el vruntime también debe ser adaptada. A continuación, se muestra la nueva fórmula para calcularlo, que toma en cuenta el tiempo de ejecución que el proceso ha acumulado, y lo escala inversamente por el peso del proceso. En el ejemplo anterior, el tiempo de ejecución de A se acumulará a una razón de 1/3 con respecto al tiempo acumulado de B.

$$\mathtt{vruntime}_i = \mathtt{vruntime}_i + \frac{\mathtt{weight}_0}{\mathtt{weight}_i} \cdot \mathtt{runtime}_i$$

Un aspecto interesante de la construcción de la tabla de pesos mostrada, es que se preserva la proporcionalidad entre el uso de CPU cuando la diferencia entre los valores nice es constante. Por ejemplo, si el proceso A tuviera un valor de 5 en lugar de –5, y el proceso B tuviera un valor nice de 10 en lugar de 0, CFS los planificaría de la misma manera que antes.

Utilizando árboles rojinegros

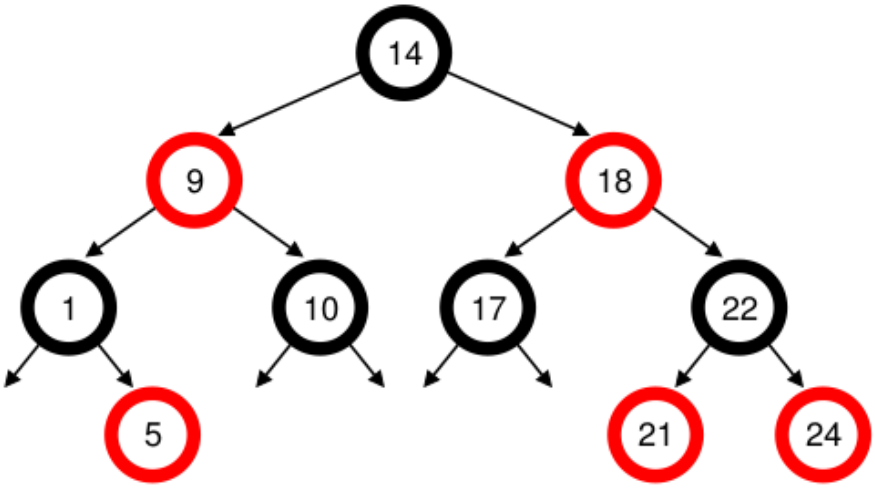
Como se discutió, uno de los objetivos principales del CFS es la eficiencia. Para un planificador, existen diferentes partes donde esto se involucra, pero una de ellas es muy simple: cuando el planificador tiene que encontrar el siguiente proceso a ejecutar, lo debe hacer de la forma más rápida posible. Estructuras simples como listas no son escalables, ya que sistemas modernos tienen miles de procesos ejecutándose, y buscar en una lista larga cada cierto número de milisegundos es desperdiciar tiempo valioso.

El CFS ataca este problema al almacenar a los procesos en un árbol rojinegro, que es un árbol binario autobalanceado. Es decir, a diferencia de un árbol binario simple que se puede degenerar hasta tener alturas de O(n), éste siempre tendrá alturas de O(log n), garantizando que las operaciones a realizar (como inserción, búsqueda, o eliminación) sean de tiempo logarítmico.

CFS no mantiene a todos los procesos en esta estructura, únicamente a procesos que se encuentran ejecutándose o listos para ejecutar. Si un proceso “duerme” (por ejemplo, esperando que se complete una solicitud de entrada/salida), se removerá del árbol y se mantendrá en otra parte del planificador.

Por ejemplo, asumamos que se tienen diez procesos, y que tienen valores de vruntime de *1, 5, 9, 10, 14, 18, 17, 21, 22 y 24*. Si se mantienen estos procesos en una lista ordenada, encontrar el siguiente proceso a ejecutar es sencillo: sólo se debe remover el primer elemento. Sin embargo, cuando se regresa ese proceso a la lista (en orden), se

tendría que recorrer la lista buscando por el lugar correcto donde insertarlo, lo cual tomaría tiempo lineal.



El árbol rojinegro permite que las operaciones se lleven a cabo de forma eficiente. Los procesos se encuentran ordenados según su vruntime, y las operaciones como inserción y eliminación toman tiempo O(log n), como ya se mencionó, lo cual es más eficiente que el tiempo lineal discutido cuando n crece, por ejemplo, a los miles.

Tratando con entrada/salida y procesos durmientes

Un problema que surge al escoger al proceso con menor vruntime, surge cuando hay procesos que duermen por períodos largos de tiempo. Por ejemplo, asumamos que se tienen dos procesos: A y B, y que A se ejecuta de forma continua mientras que B duerme por 10 segundos. Cuando B despierta, su vruntime estará 10 segundos por debajo del de A, y, por lo tanto, si no se tiene cuidado, B ahora monopolizará el CPU por los diez segundos siguientes hasta que su vruntime iguale al de A.

CFS resuelve este problema al alterar el vruntime de un proceso cuando despierta. Específicamente, CFS le asigna un nuevo valor de vruntime igual al valor mínimo encontrado en el árbol. De esta forma, CFS evita la inanición, pero a cierto costo: los procesos que duermen por periodos de tiempo cortos de forma frecuente no obtienen su parte justa del CPU.

Más funcionalidades del CFS

El CFS tiene muchas otras funciones: cuenta con numerosas heurísticas para mejorar el rendimiento del caché, estrategias para manejar múltiples procesadores de forma efectiva, puede planificar largos grupos de procesos (en vez de tratar a cada proceso

como una entidad individual), y muchas otras características interesantes. Lo presentado en este texto es únicamente una introducción, y se recomienda al lector consultar recursos adicionales, como las investigaciones realizadas por Bouron (encontradas en la sección de fuentes consultadas).

Resumen

Se introdujo el concepto de planificación de partes justas o proporcionales, y se discutieron tres enfoques: planificación de lotería, de zancadas y el Completely Fair Scheduler de Linux. La planificación de lotería utiliza la aleatorización para lograr "repartir" el CPU en partes proporcionales de forma aproximada; el enfoque de zancadas logra lo mismo, pero de forma determinística. Finalmente, el CFS, que es el único planificador "real" presentado en este texto, puede resumirse de forma burda como un round-robin con pesos y partes de tiempo dinámicas, pero construido para escalar y rendir bien con cargas grandes de trabajo.

Ningún planificador es óptimo para todos los casos de uso, y los planificadores de partes justas tienen ciertos problemas. Uno de ellos es que, como se discutió, no funcionan de forma excelente con la entrada/salida. Sin embargo, algunos de estos problemas no son relevantes en ciertos dominios. Por ejemplo, en el caso de un centro de datos virtualizado (o la nube), donde se desea asignar cierta cantidad de los ciclos de CPU a una máquina virtual, y el resto a otros recursos, la planificación de partes justas puede ser una solución adecuada, simple y efectiva.

Fuentes consultadas

- *REMZI H. ARPACI-DUSSEAU, ANDREA C. ARPACI-DUSSEAU. (2008). Scheduling: Proportional Share. En Operating Systems: Three Easy Pieces (743). Wisconsin, Estados Unidos: Arpaci-Dusseau Books, LLC.*
- *Lottery Scheduling: Flexible Proportional-Share Resource Management, por Carl A. Waldspurger y William E. Weihl. OSDI '94, Noviembre 1994.*
- *Inside The Linux 2.6 Completely Fair Scheduler, por M. Tim Jones. Diciembre 15, 2009.*
- *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management, por Carl A. Waldspurger. Tesis de doctorado, MIT, 1995.*