Planificación de tareas en Linux

Esteban Ruiz
eruiz0@fceia.unr.edu.ar

Sistemas Operativos II - 2016
Departamento de Ciencias de la Computación
FCEIA-UNR

Índice

1. Conceptos generales **1**
1.1. Objetivos del planificador 1
1.2. Tipos de tareas 1
1.3. Estados de una tarea 1
1.4. Clases de planificación 2
1.5. Prioridades 2

2. Un poco de historia **3**
2.1. Kernel 1.2 3
2.2. Kernel 2.2 3
2.3. Kernel 2.4 3

3. Una planificación más moderna: el scheduler O(1) **3**

4. CFS: hacia la justicia perfecta **4**
4.1. Multitarea ideal y precisa 4
4.2. Simulando la CPU ideal 4
4.3. Prioridades 5
4.4. Kernel 2.6.24: CFS reimplementado 5
4.5. Planificación de grupos 5
4.6. Multiprocesadores 5

1. Conceptos generales

1.1. Objetivos del planificador

Los sistemas GNU/Linux se utilizan en entornos muy diversos: es habitual ver GNU/Linux en sistemas de escritorio pero también en servidores de cómputos de alta prestación, servidores de datos, servidores de red así como en sistemas mucho más acotados y sencillos como dispositivos móviles o dispositivos embebidos.

Uno de los objetivos de la planificación entonces es poder adaptarse lo mejor posible a cualquiera de estos entornos, sin requerir demasiado poder de cómputo pero logrando un buen rendimiento.

El otro objetivo perseguido por el planificador es optimizar el rendimiento de los procesadores pero sin dejar de lado a las tareas interactivas: una tarea vinculada a la entrada/salida no puede ser demorada durante mucho tiempo.

1.2. Tipos de tareas

Basado en los objetivos anteriores, GNU/Linux intenta hacer una clara distinción entre tareas:

Interactivas: estas tareas suelen estar en el estado bloqueado la mayor parte de su tiempo, esperando algún evento externo (típicamente la interacción del usuario). Cuando se recibe el evento esperado esa tarea debe poder reaccionar cuanto antes.

Batch: estas tareas hacen un uso intensivo de la CPU (tareas CPU-bound o CPU-intensivas) y su progreso podría demorarse un poco (aunque no indefinidamente). Un ejemplo de esto es el proceso `gcc` durante la compilación de un programa en `C`.

1.3. Estados de una tarea

El ciclo de una tarea es similar al de cualquier sistema UNIX, sin embargo, al registrarse el estado de una tarea se hacen algunas distinciones (fig. 1)

Los estados posibles son los siguientes:

TASK_RUNNING: lista o en ejecución (notar que se usa el mismo valor para los dos estados posibles).

TASK_INTERRUPTABLE: bloqueada, pero se le pueden retirar recursos (p. ej. memoria).

TASK_UNINTERRUPTABLE: bloqueada, no se le pueden retirar recursos (p. ej. se necesita que sus buffers permanezcan en memoria para realizar I/O, típicamente durante una llamada al sistema).

TASK_STOPPED: bloqueada por una señal (SIGSTOP y otras).

TASK_TRACED: bloqueada, está siendo depurada en un debugger.

EXIT_ZOMBIE: zombie

EXIT_DEAD: terminada, el sistema puede liberar todas las estructuras de datos relacionadas al estado de esta tarea.

El planificador, entonces, se ocupa de las tareas en estado `TASK_RUNNING`

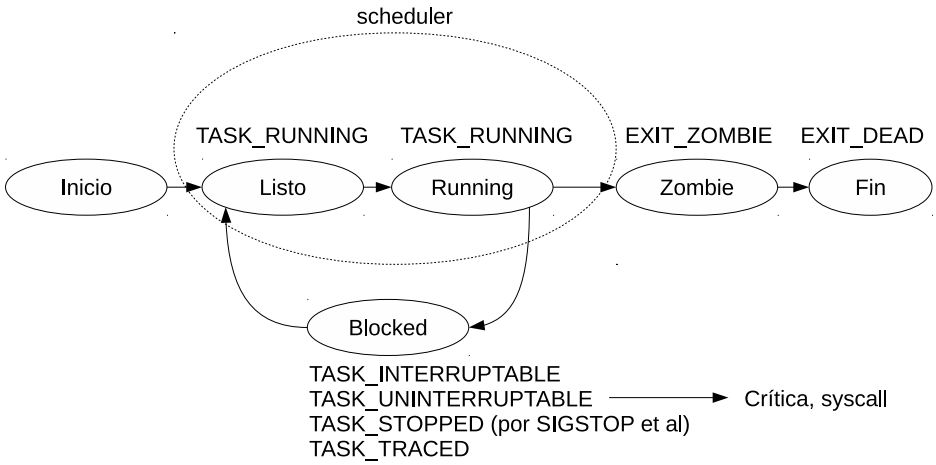

Figura 1: Estados de una tarea

1.4. Clases de planificación

En los sistemas GNU/Linux se consideran dos tipos de tareas: tareas en tiempo real (RT por sus siglas en inglés) y tareas normales.

Las tareas de tiempo real requieren atención y tiempos de respuesta cortos en relación a la velocidad del procesador (p. ej. dentro de unos cientos o miles de instrucciones), usualmente se corresponden con drivers de dispositivos (en los que los datos llegan o salen con determinada frecuencia) o con procesamiento de sonido o video. El sistema tiene dos clases diferentes de planificación para tratar tareas de tiempo real, dentro de cada clases hay distintos niveles de prioridad:

SCHED_FIFO: las tareas afectadas por esta clase sólo pueden ser reemplazadas por tareas de mayor prioridad (en caso contrario se ejecutan hasta terminar o bloquearse).

SCHED_RR: se utiliza la política de round robin con un quantum fijo dentro de la misma prioridad.

El resto de las tareas son tratadas como de clase: `SCHED_OTHER`. Nos ocuparemos principalmente de la planificación de estas tareas.

1.5. Prioridades

Cada tarea tiene una prioridad **estática** que se puede ajustar con `nice` y `set_priority`. Los procesos hijos y hilos creados por esa tarea heredan esa prioridad.

Por otro lado, el planificador asigna a cada tarea una prioridad **dinámica** utilizada internamente por el planificador. Esta prioridad dinámica está basada en la prioridad

estática pero puede varias según el tipo de tarea (interactiva/batch) y el estado actual del sistema.

GNU/Linux utiliza prioridades enteras entre 0 y 139, en donde 0 es la mayor prioridad y 139 es la prioridad menor.

Las prioridades 0 a 99 están reservadas para tareas de tiempo real (`SCHED_RR` y `SCHED_FIFO`) y el resto de las prioridades (100 a 139) son para las tareas normales.

2. Un poco de historia

Analizar la historia de la planificación utilizada en GNU/Linux puede dar cuenta de cómo fue adaptándose el núcleo a los avances de la tecnología y a sistemas cada vez más eficientes y avanzados.

2.1. Kernel 1.2

En el Kernel 1.2 se utilizó una planificación muy simple, basada en round robin.

2.2. Kernel 2.2

Aquí aparece el concepto de clases de planificación (tiempo real, no apropiable) y soporte para multiprocesadores (SMP).

2.3. Kernel 2.4

Se implementa un nuevo planificador basado en **épocas**: se garantiza que durante una época todas las tareas reciben atención (pueden utilizar la CPU), de esa forma ninguna tarea puede quedar esperando por siempre a pesar de tener baja prioridad. Las tareas de mayor prioridad reciben un lapso de tiempo (time-slice) mayor al de las tareas de baja prioridad y una tarea que no consuma todo su time-slice (tarea interactiva) es beneficiada en la siguiente época.

El problema de esta política es que la elección de la próxima tarea a beneficiar es de `O(N)` -en donde `N` es el número de tareas listas- y comienza a ser costosa en sistemas con cientos o miles de tareas (poco escalable).

3. Una planificación más moderna: el scheduler O(1)

En el kernel 2.4 se incorporó un nuevo planificador, desarrollado por Ingo Molnar que resolvía el problema de la escalabilidad: se lo llamó **scheduler O(1)** pues la elección de la próxima tarea se realizaba en tiempo constante, independientemente del número de tareas en estado `TASK_RUNNING`.

La idea es bastante sencilla: se utiliza un arreglo de colas de tareas activas y otro arreglo de colas de tareas expiradas (cada arreglo contiene 140 colas de prioridad)[1] para ir pasando las tareas activas al arreglo de las expiradas a medida que reciben su time-slice. Una vez que no quedan más tareas en el "arreglo de activas" (todas las tareas están en el "arreglo de expiradas") se intercambian los arreglos y se vuelve a repetir el ciclo.

[1] Notar que las tareas expiradas siguen estando en estado TASK_RUNNING

Así, el proceso de elegir la próxima tarea es sencillamente tomar la primera tarea activa de la cola no vacía de mayor prioridad.

Las tareas interactivos reciben un "priority-bonus" basado en un cálculo del tiempo promedio dormido (`average sleep time`). En cambio las tareas batch son penalizadas decrementando su prioridad.

A pesar de las indiscutible eficiencia de este planificador, las heurísticas para tratar con los tareas interactivas y tareas batch se volvieron demasiado complejas, propensas a errores y difíciles de mantener.

4. CFS: hacia la justicia perfecta

Dados los problemas que presentaba el scheduler O(1), Con Kolivas decidió implementar un nuevo planificador basado en sus publicaciones recientes sobre un Rotating StairCase Deadline Scheduler (RSDL, algo así como "Planificador de plazos de escalera de caracol"). La implementación resultó en un parche (patch) al kernel que tuvo mucha repercusión y se sugirió incorporarlo definitivamente al kernel en las siguientes versiones.

Sin embargo, luego de algunas discusiones en las listas de desarrollo, Ingo Molnar (que había desarrollado el scheduler O(1)) decidió implementar (casi desde cero) un nuevo planificador basado en algunas de las ideas de Con Kolivas.

Este nuevo planificador (que se sigue utilizando actualmente) se denomina CFS: planificador completamente justo (por sus siglas en inglés Completely Fair Scheduler). Es completamente justo (o equitativo) en el sentido de que intenta emular lo que ocurriría en una "CPU multitarea ideal y precisa".

4.1. Multitarea ideal y precisa

¿Y qué es lo que ocurriría en una "CPU multitarea ideal y precisa"?

El concepto de **CPU multitarea ideal y precisa** permite definir el concepto de equidad entre tareas. Esta CPU es una CPU teórica, que tiene un cierto poder de cómputo.

Si hay una sola tarea, esa tarea usará el 100 % del poder de cómputo. Pero esta CPU, al ser multitarea permite ejecutar simultáneamente dos o más tareas: en ese caso el poder de cómputo se dividirá equitativamente entre las tareas que la utilicen (es decir, si hay 4 tareas, cada una recibirá el 25 % del poder de cómputo).

Así, lo que pretende CFS es simular el comportamiento de esta CPU ideal: dividir el poder de cómputo equitativamente entre todas las tareas listas. Pero en una CPU real no pueden ejecutarse dos tareas a la vez, así que se opta por intentar repartir el tiempo de uso de la CPU.

4.2. Simulando la CPU ideal

Para esto se toma como base un "fair clock" (`cfs_rq->fair_clock`). La frecuencia de incremento de este reloj se calcula dividiendo el tiempo real (`wall time`, en nanosegundos) por el número de tareas y representa el poder de cómputo asignado a una tarea en una "CPU multitarea, ideal y precisa".

Cada tarea tiene un `wait_runtime` asociado: el tiempo (del `fair_clock`) que pasó esperando por la CPU. Este tiempo se incrementa mientras la tarea está esperando para usar la CPU y se decrementa cuando la tarea está utilizando la CPU.

El cambio de contexto se realiza cuando alguna de las tareas en espera tiene mayor `wait_runtime` que la tarea que está en ejecución. Notar que sólo interesan los tiempos y el concepto de `quantum` no aparece.

Para mantener las tareas ordenadas según su necesidad de tiempo (y así saber cuál es la más atrasada con respecto al "fair clock") se utiliza un red-black-tree (RBT). El red-black-tree tiene la propiedad de que las operaciones que se necesitan (búsqueda, inserción, borrado) son todas de $O(logN)$, en donde N es el número de nodos del árbol (y en este caso, el número de tareas en espera). Así, se utiliza la diferencia: `rq->fair_clock - p->wait_runtime` para ordenar el árbol y mantener siempre a la izquierda la tarea que más necesidad de tiempo tiene. Decidir cuál es el siguiente tarea es entonces tomar el nodo de más a la izquierda.

4.3. Prioridades

Para poder tener en cuenta las prioridades de las tareas se asocia un peso a cada prioridad (que es usado cuando se decrementa el `wait_runtime`), de forma que la tareas de baja prioridad ven pasar el tiempo más rápido que las de alta prioridad (su `wait_runtime` expira más rápido).

4.4. Kernel 2.6.24: CFS reimplementado

En el kernel 2.6.24 se hizo una reestructuración del CFS: en vez de utilizar `rq->fair_clock`, las tareas se persiguen entre sí según un tiempo "virtual": `vruntime`. El `vruntime` se incrementa con el reloj real (con un factor relacionado a la prioridad) y la tarea con menor `vruntime` es entonces la indicada para tomar el procesador. Así, nuevamente se usa un RBT, pero esta vez ordenado por `vruntime`.

Además en esta versión se agregó "group scheduling" (planificación de grupos).

4.5. Planificación de grupos

La planificación de grupos permite hacer más equitativo el uso del procesador basándose en distintas formas de agrupar las tareas.

Un caso típico es el siguiente: si no existe ninguna agrupación de tareas y el usuario A lanza 24 tareas y el usuario B lanza sólo una tarea, entonces B recibirá un 4 % del tiempo de CPU y el usuario A recibirá un 94 % del uso del CPU. Si lo vemos desde el punto de vista del usuario B, resulta que A recibe mucho más uso del procesador que B (lo que no es justo o equitativo si el sistema pretende ser equitativo con el tiempo de CPU brindado a todos los usuarios activos).

CFS permite agrupar tareas por usuario, según la jerarquía de procesos o haciendo una combinación de ambos. Más detalles en [4].

4.6. Multiprocesadores

En un equipo multiprocesador, cada procesador tiene su propia runqueue (cola de procesos listos, en este caso representada por el RBT de CFS). Además cada runqueue lleva una medida de su carga (`load`). Esta medida se usa para mover tareas de un procesador a otro cuando uno de los procesadores está muy cargado y otro está demasiado ocioso. En esos casos se produce lo que se llama una **migración** de tarea: se quita una

tarea del procesador más cargado y se lo hace formar parte de la lista del procesador menos cargado.

La migración de tareas tiene una incidencia negativa en la eficiencia de los cachés, así que sólo debería realizarse cuando el cambio compense la perdida de eficiencia de los cachés.

Además se añade el concepto de `scheduling domains`: estos dominios permiten agrupar uno o más procesadores jerárquicamente para realizar balance de carga o segregación.

Referencias

[1] Daniel Bovet & Marco Cesati, *Understanding the linux kernel*, 3rd ed., O'Reilly

[2] *Inside the Linux 2.6 Completely Fair Scheduler*, artículo de IBM Developer Works en http://www.ibm.com/developerworks/library/l-completely-fair-scheduler

[3] *Completely Free Scheduler*, artículo de Linux Journal en http://www.linuxjournal.com/magazine/completely-fair-scheduler

[4] *Linux v2.6.28.4 documentation: CFS Scheduler* http://lxr.linux.no/#linux+v2.6.28.4/Documentation/scheduler/sched-design-CFS.txt