Taller de sistemas operativos

PLANIFICADOR

Agenda

- Introducción
- Clases de planificación
- Prioridades
- Timeslice
- Prioridad dinámica
- Estructuras del planificador
- Planificador en funcionamiento
- Nuevo planificador
- Multiprocesadores, dominios de planificación
- Procesos de tiempo real

Introducción

- El planificador es el componente del sistema operativo que se encarga de repartir el procesador entre los procesos listos
- El planificador del Linux es preemptivo
- Fue reescrito completamente para la versión 2.6 (dos veces)
- Intenta lograr buen tiempo de respuesta interactivo por lo que favorece a los procesos limitados por I/O
- También asegura que los procesos limitados por CPU puedan ejecutar alguna vez

Clases del planificación

- Depende de la clase de planificación del proceso (scheduling class)
  - SCHED_FIFO
  - SCHED_RR
  - SCHED_NORMAL
- Las dos primeras son para procesos de tiempo real, la segunda es para los demás
- Usa algoritmos distintos para cada una de ellas

Primera implementación

- Todas sus operaciones corren en tiempo constante independientemente de la cantidad de procesos en el sistema (O(1))
- Maneja colas separadas por procesador
- Intenta que los procesos corran siempre en la misma CPU
- Está optimizado para el caso más común de pocos procesos listos, pero escala a varios procesadores con muchos procesos listos
- Está basado completamente en consideraciones empíricas

Prioridades

- El planificador de Linux está basado en prioridades
- Ejecuta siempre el proceso de mayor prioridad
- Además el timeslice es mayor cuanto mayor sea la prioridad.
- Linux usa prioridades dinámicas: el planificador puede aumentar o disminuir la prioridad de un proceso
- Para procesos normales la prioridad va de 100 a 139
  - Se corresponden con nice de -20 a 19
- Para procesos de tiempo real va de 0 a 99

Timeslice

- El cuanto por defecto de Linux es de 100 ms
- El cuanto de un proceso depende de su prioridad (estática)
  - (140 – prioridad) x 20 si prioridad < 120
  - (140 – prioridad) x 5 si prioridad >= 120
- Por ejemplo
  - prioridad = 100 → cuanto = 800 ms
  - prioridad = 139 → cuanto = 5 ms
- No es necesario usar todo el cuanto de una vez, al bloquearse se guarda el resto
- Al hacer un fork el timeslice se divide entre dos

Prioridad dinámica

- El planificador usa en realidad la prioridad dinámica
  - max (100, min (prio. estática – bonus + 5, 139))
- Bonus depende de la historia del proceso
  - Si es menos de 5 penaliza, si es más de 5 favorece
  - Cuando un proceso se despierta su bonus se setea igual al tiempo que estuvo durmiendo (con 10 como tope)
  - Por cada tick en que el proceso está corriendo se le resta uno
  - Busca favorecer a los procesos interactivos
- Se guarda en el campo sleep_avg del task_struct

Estructuras del planificador

- La estructura principal es la runqueue
- Existe una por procesador y contiene las listas de procesos listos para esa CPU
- Tiene 140 listas de procesos listos y 140 de procesos con timeslice expirado (una por prioridad)
- Hay además un bitmap que indica si cada lista está vacía o no
- La búsqueda del próximo proceso listo consiste en encontrar el primer bit 1 del bitmap y sacar al primer proceso de la lista (O(1))

Estructuras del planificador

```
struct runqueue {
        spinlock_t          lock;   /* spin lock that protects this runqueue */
        unsigned long       nr_running;          /* number of runnable tasks */
        unsigned long       nr_switches;         /* context switch count */
        unsigned long       expired_timestamp;     /* time of last array swap */
        unsigned long       nr_uninterruptible;   /* uninterruptible tasks */
        unsigned long long  timestamp_last_tick;  /* last scheduler tick */
        struct task_struct  *curr;                /* currently running task */
        struct task_struct  *idle;            /* this processor's idle task */
        struct mm_struct    *prev_mm;         /* mm_struct of last ran task */
        struct prio_array   *active;          /* active priority array */
        struct prio_array   *expired;         /* the expired priority array */
        struct prio_array   arrays[2];        /* the actual priority arrays */
        struct task_struct  *migration_thread; /* migration thread */
        struct list_head    migration_queue;   /* migration queue*/
        atomic_t            nr_iowait; /* number of tasks waiting on I/O */
};
```

Fuente: Linux Kernel Development, segunda edición

Planificador en funcionamiento

- Cuando un proceso finaliza su cuanto pasa la cola de procesos expirados y se le recalcula el cuanto
- Esto evita recalcular todos los cuantos juntos
- Puede entonces ser reemplazado por otro listo de menor prioridad
- Cuando no hay más procesos en la cola de listos se intercambian las dos y se vuelve a empezar
- A los procesos con alta prioridad se les divide su cuanto en partes para que no monopolicen la CPU

Planificador en funcionamiento

- Los procesos interactivos pueden seguir ejecutándose aunque terminen su cuanto
- Un proceso se considera interactivo si
  - bonus – 5 >= prio. estática / 4 – 28
- Ejemplos
  - prio. estática = 100 → intercativo si bonus > 2
  - prio. estática = 139 → nunca puede ser interactivo
- Un proceso que hace mucho que está en la cola de expirados también puede pasar a la de listos

Problemas con el planificador

- Al asignar timeslices dependiendo de la prioridad se pueden tener demasiados context switches
- Es complicado asignar valores razonables a los timeslices. Por ejemplo:
  - prio 139 = 5ms, prio 138 = 10ms
  - prio 120= 100ms, prio 119 = 105ms
- Los valores de los timeslices deben ser múltiplos del tick del timer
- ¿Qué hacer con los procesos interactivos con el cuanto expirado?

Nuevo Planificador

- Se llama CFS (Completely Fair Scheduler)
- Intenta modelar un CPU multi-tarea ideal
  - Corre n tareas en paralelo a velocidad 1/n
- A cada tarea se le agrega un valor vruntime que mide el tiempo que cada proceso ejecutó normalizado por la cantidad de tareas
- Este valor crece siempre que la tarea está ejecutando a velocidad n
- Determina el tiempo que hubiera ejecutado la tarea en un CPU ideal
- Se ejecuta siempre la tarea con menor vruntime

CFS

- Se lleva también un contador min_vruntime que cuenta el menor vruntime de todas las tareas (siempre crece)
- Se usa para inicializar el vruntime de las nuevas tareas para que no queden con un vruntime desparejo
- No hay time slices fijos ni ninguna heurística para calificar a los procesos
  - Hay un parámetro llamado targeted latency que es el tiempo en el que deben correr todos los procesos
  - Si hay n procesos cada uno tiene timeslice de targeted latency / n
- Hay solamente una granularidad mínima de uso de CPU (1 ms) para no cambiar de proceso todo el tiempo y arruinar el cache

CFS

- Las prioridades se implementan cambiando el tiempo. El tiempo de CPU pasa “más rápido” para los procesos de menos prioridad
- Los procesos se agrupan en un red-black tree ordenados por vruntime
- Se ejecuta siempre la tarea de “más a la izquierda”
- Luego de ejecutar el proceso se lo vuelve a colocar en el árbol (va a quedar más a la derecha)
- Tiene orden log n
- Se agrega al task_struct un puntero a sched_entity que tiene un puntero al nodo del árbol (run_node)
- La estructura sched_entity tiene un entero de 64 bits vruntime

Red-Black Tree

- Es un árbol binario de búsqueda cuyos nodos pueden ser de uno de dos colores: rojos o negros
- Cumple las siguientes propiedades
  - La raíz es negra
  - Las hojas son negras
  - Ambos hijos de un nodo rojo son negros
  - Todos los caminos descendentes desde un nodo determinado hasta una hoja tienen la misma cantidad de nodos negros
- Propiedad fundamental:
  - El camino dese la raíz hasta la hoja más lejana es a lo sumo el doble que el camino hasta la más cercana

Otras Mejoras

- Group scheduling
  - Se pueden agrupar los procesos en grupo para ser más justo
  - Ej: un grupo de 20 procesos recibe la misma CPU que otro grupo de 2 procesos
- Scheduler classes
  - Permite implementar una interfaz para crear nuevos algoritmos de planificación
  - Las clases predefinidas son las mismas

Multiprocesadores

- Cada CPU tiene su propia estructura de planificación, por lo que se planifica localmente
- Puede pasar que una CPU tenga varios procesos listos asociados y otra no tenga nada
- El load balancer se encarga de mantener todas las colas balanceadas
- Se invoca:
  - por schedule() cuando no hay procesos listos
  - cada 1 ms si no hay trabajo y si no cada 200 ms

Load Balancer

- Trata de encontrar una CPU con 25% más de procesos listos que la actual
- Si encuentra decide si va a sacar procesos de la cola de activos o de la de expirados (en el caso del viejo planificador)
- Busca los procesos de mayor prioridad de la cola
- Entre ellos se busca uno que no esté corriendo, no este “cache hot” y no tenga processor affinity con la CPU
- Esto se repite mientras las colas permanezcan desbalanceadas

Dominios de planificación

- Para soportar diferentes diseños de multiprocesadores se usan los dominios de planificación
- Un dominio de planificación es un conjunto de CPUs cuya carga debe mantenerse balanceada
- Tienen una estructura jerárquica donde el dominio de más alto nivel abarca todas las CPUs del sistema
- El trabajo se balancea entre miembros del mismo dominio de planificación
- Un proceso no se mueve si está cache hot o si el dominio es una misma CPU con hyper threading

“Preemtion”

- El cambio de procesos se realiza mediante una llamada a schedule()
- Se realiza automáticamente al retornar al espacio de usuario luego:
  - de un system call
  - de ejecutar un manejador de interrupción
- Solo se hace si la bandera need_resched está activada
- El propio kernel de linux también es preemptivo
- Se puede cortar el código del kernel siempre y cuando no se tenga ningún lock (contador preempt_count en thread_info)

Procesos de tiempo real

- Su prioridad va de 0 (máxima) a 99 (mínima)
- Un proceso activo de mayor prioridad no permite la ejecución de ningún proceso de menor prioridad (RT o no)
- Si es SCHED_RR se hace un round robin entre los procesos de tiempo real de máxima prioridad
- Si es SCHED_FIFO se ejecuta el primero de la cola hasta que se bloquee o deje el procesador voluntariamente
- No hay prioridades dinámicas para procesos de tiempo real

Procesos de tiempo real

- Un proceso de tiempo real es reemplazado por otro cuando:
  - Está listo un proceso de tiempo real de mayor prioridad
  - Se bloquea (TASK_INTERRUPTIBLE o TASK_UNINTERRUPTIBLE)
  - Se detiene (TASK_STOPPED)
  - Se muere
  - Voluntariamente deja la CPU (yield())
  - Es RR y se le termina el cuanto