Universidad de Concepción

Ingeniería Civil Informática

Sistemas Operativos

Tarea II : Planificador Linux

Integrantes:

Javier Arriagada.
Felipe Cortes.
Diego Dominguez.
Leonardo Quiroz.
Angel Rodriguez.
Natalia Soto.
Jeremías Torres.
Pablo Venegas.
Carlos Von Plessing.

Docente:

Cecilia Hernández.

Concepción - Octubre, 2017

Índice

1. Introducción.

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

En el siguiente informe se detalla el funcionamiento del planificador, hablando de su estructura, programación, y una pequeña reseña historica, para dar paso a las conclusiones.

1.1. Planificadores Linux, pequeña historia.

1.1.1. Génesis

Antiguamente el primer planificador de procesos de Linux, lanzado con la versión 0.01 del kernel en 1991, usaba un diseño mínimo y no estaba pensado para manejar múltiples procesadores o procesos al mismo tiempo, en esos momentos lo que ocupaba este planificador era la función Schedule(), que permitía, en una única cola, ir ejecutando los procesos que se necesitaban realizar, todo dentro de un ciclo infinito, acotado por FIRST_TASK y LAST_TASK, dado que cada "tarea" era realizada de manera secuencial, este planificador tenía un orden de O(n) por cada tarea que debía desarrollar, lo cual era muy ineficiente.

1.1.2. Planificador O(n)

Posteriormente en la versión 2.4 del kernel de Linux se implementó un algoritmo de planificación que asignaba porciones de tiempo para que cada proceso se ejecutara, luego de que todos los procesos completaban su porción de tiempo, se elegía, en cada ciclo completo de ejecuciones, la tarea más "deseable" para el planificador, todo esto gracias a la función goodness(), la complicación de esto, era que mientras más tareas se tuvieran, el planificador demoraría más, dado que debía analizar una gran variedad de tareas, para escoger la mejor.

1.1.3. Planificador O(1)

En las primeras versiones del kernel 2.6 se introdujo un nuevo planificador de procesos, capaz de alcanzar el orden O(1), sin importar la cantidad de procesos que estuvieran corriendo al mismo tiempo en el sistema. Este planificador O(1), incorporó nuevas funcionalidades:

- Escala de prioridad global
- Cuando un proceso entra en el estado de TASK_RUNNING, el planificador chequea si su prioridad es mayor que las de las tareas actualmente en ejecución, de serlo, el planificador le otorga CPU a la tarea recién ejecutada.
- Prioridad estática para las tareas en tiempo real.

- Prioridad dinámica para las tareas en tiempo normal, dependiendo de su interactividad, recalculando luego de cada porción de tiempo asignada por el planificador.

Este método recalcula la prioridad de la tarea al finalizar la porción de tiempo o quantum que le asigna el planificador, y mantiene 2 arreglos de prioridad para cada CPU disponible, uno para los procesos activos (los procesos que se están ejecutando y les queda por terminar su porción de tiempo) y otro para los "expirados" (los que ya vieron acabado su quantum), de esta forma los procesos en el arreglo de expirados no son tomados en cuenta a la hora de recalcular los procesos que deben ser planificados para "activarse" en el próximo ciclo.

Cada cola de ejecución contiene un arreglo de dos elemento del tipo struct prio_array que representan el conjunto de procesos ejecutándose e incluyen:

- 140 doble listas ligadas (uno por cada valor de prioridad).
- Un bitmap de prioridad.
- El número de tareas en él.

Este planificado a diferencia de los anteriores, escalaba mejor, era más rápido, no se tomaba mucho tiempo para cambiar entre tareas, además que incorporaba, métricas de interactividad y todo esto en una estructura agradable de analizar. Sin embargo una de las desventajas de este algoritmo era la de que cuando se analizan heurísticas muy complejas muchas tareas eran marcadas como interactivas o con E/S. Los cálculos eran tan complejos y daban paso a errores y a que los procesos hicieran cosas para las cuales no estaban pensados.

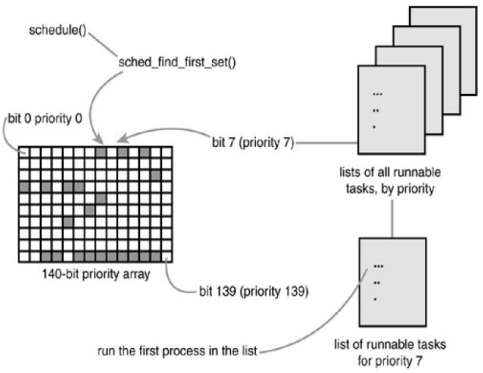

Figura 1: The Linux O(1) scheduler algorithm.

1.1.4. Planificadores Staircase

En 2004 Con Kolivas lanzo el "Staircase Scheduler". Este planificador está implementado con una cola de ejecución de procesos en el arreglo ordenados (para cada CPU). Inicialmente cada proceso es puesto en el arreglo dependiendo de su rango de prioridad, el planificador entonces escoge el primer proceso de la mayor prioridad disponible, muy parecido al planificador anterior, pero no ocupando la técnica del arreglo de "expirados" si no que la prioridad del proceso ejecutado sería degradado de prioridad, o bajar un escalón de la "escala de prioridad", por eso el nombre, de esta forma con cada ciclo completado el proceso cada vez va viendo disminuida su prioridad para la CPU, de esta forma a los procesos más abajo en la escalera se les otorga más quantum o porciones de tiempo para que sean desarrollados, dejando a los procesos interactivos en la parte más alta de la escalera.

Luego en 2007 Kolivas vuelve a modificar el algoritmo y lanza un nuevo planificador, "The Rotating Staircase Deadline Scheduler", apuntando nuevamente a una mejor interactividad, haciendo que los procesos no estuvieran esperando mucho para que les fuera asignada CPU para ejecutarse, o en otros términos, no están hambrientos por CPU.

Lamentablemente la comunidad se quejaba de que este tipo de planificador estaba muy orientado a desarrollo en escritorio, lo que Linux en ese momento no estaba dirigido, por ende no fueron usados.

Luego de esto Ingo Molnar, el creador del planificador O(1) y el jefe de mantenimiento de las políticas de los planificadores desarrollaron el CFS(Completely Fair Scheduler), planificador completamente justo, por sus siglas en inglés, que estaba basado en las ideas desarrolladas por Kolivas en el diseño del planificador escalera.

2. Basic.

2.1. Listas Ligadas

Linux usa una lista ligada doble circular para guardar los procesos y, como cualquier proyecto de software Linux ofrece su propia implementación genérica de estas, fomentando la reutilización de código para así no estar reinventando la rueda cada vez que alguien quiere guardar alguna cosa. Antes de adentrarse más en el tema se verá cómo Linux implementa listas ligadas.

```
struct list_element{
    void *data;
```

```
    struct list_element *next; /* siguiente elemento */
    struct list_element *prev; /* elemento previo */
};
```

Aquí se ve una implementacion básica de una lista ligada doble, Linux se acerca a esta de una forma un poco distinta, en vez de tener una estructura de lista ligada, se integra un nodo de una lista ligada a una estructura

```
struct list_head{
    struct list_head *next, *prev;
};
```

Agregando esto a nuestra estructura:

```
struct task{
    int num; /* number of the task to do */
    bool is_awesome; /* is it something awesome? */
    struct list_head mylist; /* the list of all tasks */
};
```

Uno de los beneficios de esto es que uno puede moverse por los datos guardados en ella, donde linux provee de funciones para eso, todo esto en 0(n) (donde n es numero de nodos).

3. Planificador.

3.1. Estructura.

La estructura de datos central en el planificador de procesos activos se llama runqueue. Una runqueue, de acuerdo con su nombre, tiene tareas que están en un estado ejecutable en cualquier momento dado, esperando al procesador. Cada CPU (núcleo) en el sistema tiene su propio runqueue, y cualquier tarea se puede incluir en un unueue; no es posible ejecutar una tarea en varias CPUs diferentes al mismo tiempo. Sin embargo, las tareas pueden "saltar"de una cola a otra, si un equilibrador de carga multiprocesador lo decide. El trabajo del planificador de procesos consiste en seleccionar una tarea de una cola y asignarla para que se ejecute en una CPU.

```
struct task_struct {
   int prio, static_prio, normal_prio;
   unsigned int rt_priority;
   const struct sched_class *sched_class;
   struct sched_entity se;
   struct sched_rt_entity rt;

   unsigned int policy;
   cpumask_t cpus_allowed;
}
```

- El campo prio denota la prioridad del proceso en el sistema. Es decir, lo importante que es una tarea determinada.
- sched_class indica la clase de programación en la que se encuentra el proceso.
- La estructura sched_entity tiene un significado especial,ya que linux es capaz de programar no solo tareas individuales, sino grupos de procesos o incluso usuarios como un todo
- sched_rt_entity denota a una entidad, programada con preferencias en tiempo real.
- La policy tiene las políticas de programación de una tarea, esto significa básicamente las decisiones especiales de programación para un grupo particular de procesos La versión actual del kernel de Linux admite cinco tipos de políticas:
  - SCHED_NORMAL
  - SCHED_BATCH
  - SCHED_IDLE
  - SCHED_FIFO y SCHED_RR
- cpus_allowed es un tipostruct, que contiene el campo de bits, que indica la afinidad de una tarea de vinculación a un determinado conjunto de núcleos de CPU.

3.1.1. Prioridades.

Todos los sistemas operativos basados en UNIX, implementan una programación basada en prioridades. La idea principal es asignar algo de valor a cada tarea del sistema y, a continuación, determinar cuán importante es una tarea mirando este valor y comparándolo con los valores de otras tareas. Si dos procesos tienen el mismo valor, ejecútelos uno tras otro repetidamente. Linux usa prioridades dinámicas, es decir el planificador puede aumentar o disminuir la prioridad de un proceso:

- Para procesos normales la prioridad va de 100 a 139, donde menor es el número mayor es la prioridad
  -Se corresponden con nice de -20 a 19
- Para procesos de tiempo real va de 0 a 99, donde mayor es el número mayor es la prioridad

El kernel utiliza la llamada a sistema nice para cambiar la prioridad de un proceso, sin embargo el usuario también puede cambiar la prioridades de estos, pero, sólo de los procesos que ha iniciado.Para cambiar la prioridad de un proceso que se comienza a ejecutar :
Para procesos Normales:
$ nice -n (incremento) (command_to_execute)

para cambiar prioridad de procesos que ya están en ejecución se utiliza

$ renice -n (priority) -p (PID)

Para Procesos en Tiempo Real.

$ chrt -p prio pid

3.1.2. Cálculo de prioridades.

Las prioridades del proceso hacen que algunas tareas sean más importantes que otras. Mientras mas alta es la prioridad de una tarea, más tiempo de CPU se le permite utilizar. La relación entre amabilidad y los tiempos de los procesos se mantienen mediante el cálculo de pesos de carga.

El kernel de Linux define una matriz, que contiene el peso correspondiente para cada nivel de amabilidad.

```
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,
};
```

La matriz contiene una entrada para cada valor agradable [-20, 19]. Para que esto tenga sentido, considere dos tareas, ejecutándose por defecto en el nivel 0 - peso 1024. Cada una de estas tareas recibe el 50 % del tiempo de CPU, porque 1024 / (1024 + 1024) = 0,5. De repente, una de las tareas obtiene un impulso de prioridad de 1 nivel nice (-1). Ahora, de acuerdo con la documentación, una tarea debe obtener alrededor de un 10 % de tiempo de CPU más, y el otro 10 % menos: 1277 / (1024 + 1277) = 0,55; 1024 / (1024 + 1277) = 0,45 - diferencia del 10 %, como es esperado.

3.1.3. Clases de Programación

El actual planificador de kernel de Linux ha sido diseñado de tal manera que introduce una jerarquía extensible de módulos de planificador. Estos módulos incluyen los detalles de la política de programación y son manejados por el núcleo del planificador sin el código básico del esqueleto.

- core.c contiene la parte central del planificador;
- fair.c implementa el actual planificador de Linux para "tareas normales"
- rt.c es una clase para programar tareas en tiempo real; es decir, una política de programación completamente diferente de CFS;
- idle_task.c maneja el proceso inactivo del sistema, también llamado la tarea swapper, que se ejecuta si no se ejecuta ninguna otra tarea.

Las clases de planificación se implementan a través de la estructura sched_class, que contiene punteros a funciones, implementadas dentro de diferentes clases que deben ser llamadas en caso de un evento correspondiente. Los campos son los siguientes:

- enqueue_task: se llama cuando una tarea entra en un estado ejecutable.
- dequeue_task: cuando una tarea ya no es ejecutable
- yield_task: llamada cuando una tarea quiere abandonar voluntariamente la CPU, pero no salir de estado ejecutable
- check_preempt_curr: esta función comprueba si una tarea que entró en estado runnable debe interrumpir la tarea actualmente en ejecución
- pick_next_task: esta función elige la tarea más apropiada para ejecutarse a continuación.
- set_curr_task: esta función se llama cuando una tarea cambia su clase de planificación o grupo de tareas;
- task_tick: en su mayoría llamadas desde las funciones de ticks de tiempo; podría conducir al interruptor del proceso
- task_fork y task_dead: notifican al programador que una nueva tarea se generó o murió, respectivamente.

3.2. Invocando el planificador.

Schedule() es una función de gran importancia en el kernel de linux, tiene llamadas en diferentes partes del código del núcleo y en varias ocasiones se utiliza esta función. La idea es sincronizar los procesos, para que se lleven a cabo las tareas de administración del sistema, pero también la parte de proceso de usuario. Con esto se puede establecer un control de aquellos procesos que requieren de cierto evento para poder continuar su ejecución y con esto dar paso a aquellos que necesitan ejecutarse (que se encuentren ya en la cola de proceso listos). Yendo a un ambiente normal de ejecución, el proceso que se encuentra ejecutándose eventualmente finaliza su segmento de tiempo (en caso de utilizar algún algoritmo como el Round Robin el cual usa quantums para establecer un tiempo determinado de ejecución) El **planificador de Linux** recoge otro proceso que tenga prioridad de ejecución y asigna los ciclos de CPU a ese proceso. Sin embargo, un proceso también puede renunciar voluntariamente a la CPU. La función **schedule()** puede ser utilizada por un proceso para indicar voluntariamente al planificador que puede programar algún otro proceso en el procesador. Cuando a un proceso se le termina su tiempo de ejecución, las variables utilizadas para ese proceso (como el timer) son inicializadas por la función **scheduler_tick()**. Una

parte importante de **scheduler_tick()** se dedica a actualizar el contador del quantum del proceso y quitarlo de la cola si lo ha agotado.

3.2.1. Planificador periódico.

Como ya se mencionó antes, la función **scheduler_tick()** se llama en cada interrupción del temporizador. Cuando el manejador de la interrupción de reloj se ejecuta invoca la función **update_process_times()** que realiza, entre otras cosas, una llamada a **scheduler_tick()**, donde cómo vamos a ver se actualizan todos los parámetros y datos relacionados con la planificación.

```
void scheduler_tick(void){
  int cpu = smp_processor_id();
  struct rq *rq = cpu_rq(cpu);
  struct task_struct *curr = rq->curr;
    /*...*/
  update_rq_clock(rq); //acutaliza el reloj
  curr->sched_class->task_tick(rq, curr, 0);
  update_cpu_load_active(rq);
    /*...*/
}
```

La función actualiza el reloj de la cola ejecución en update_rq_clock(). En el planificador de procesos predeterminado de Linux por ejemplo, apunta a task_tick_fair(), implementado en <kernel/sched/fair.c>. Internamente, la clase de planificador puede decidir si la tarea actual tiene que ser adelantada por algún otro y establecería el indicador TIF_NEED_RESCHED, que le indica al kernel que llame a schedule() lo antes posible.

3.2.2. Proceso actualmente en ejecución entra en estado de suspensión.

Como lo dice el título, una tarea o proceso que entra en reposo o se bloquea, es necesario hacer que renuncie voluntariamente a la CPU para que alguien más pueda tomarla. Cuando eso sucede es porque generalmente, la tarea está esperando a que ocurra cierto evento. La tarea es agregada a una cola de bloqueados y estará en un ciclo hasta que la condición deseada se cumpla. Mientras se va a dormir, una tarea se establece en el estado **TASK_INTERRUPTIBLE** o **TASK_UNINTERRUPTIBLE**. El último significa que la tarea no manejará ninguna señal pendiente hasta que se despierte. Justo antes de que una tarea se vaya a dormir, el planificador se llamará para seleccionar la siguiente tarea que se va a ejecutar.

3.2.3. Despertar de tarea dormida.

Un proceso se encuentra en la cola de bloqueados y generalmente está pendiente de que se le vuelva a llamar. Cuando se le llama: se establece un estado de listo y se vuelve a poner en la cola de procesos listos. Si el proceso tiene una prioridad mayor a cualquier proceso en la cola, la bandera **TIF_NEED_RESCHED** se posiciona en él y se llama a **schedule()** para realiza su ejecución.

- **Los procesos deben esperar a los eventos.**
- **Los procesos que esperan eventos se establecen en una cola de bloqueados.**
- **El proceso puede despertar antes de que ocurra su evento, es por ello que debe comprobar su condición y si no se ha cumplido volver a bloqueados.**

```
DECLARE_WAIT_QUEUE(wait, current);
/* Proceso duerme en la cola "q" */
add_wait_queue(q, &wait);
while (!condition){
  set_current_state(TASK_INTERRUPTIBLE);
  if(signal_pending(current))
/* Maneja la senal (Realiza una senal) */
    schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(q, &wait);
```

Despertar un proceso

- **En la cola de bloqueados de toma el proceso que se encontraba bloqueado y se le despierta.**
- **Hay posibilidad de que varios procesos estén esperando un mismo evento, es decir, que tratan de leer un solo bloque de disco.**
- **El proceso bloqueado debe estar dentro de un ciclo para mantenerlo así hasta que la condición se cumpla.**

Schedule()
Esta es la función principal del núcleo Linux planificador. Como hemos visto anteriormente, se puede llamar desde muchos lugares diferentes dentro del código del núcleo, ya sea activa, cuando una tarea da voluntariamente la CPU, o de un modo vago mediante la activación de la bandera "need_resched" que el núcleo menudo comprueba. El objetivo de la función **Schedule()** es reemplazar actualmente el proceso de ejecución con otro. Aunque, en algunos casos, no es posible encontrar una nueva tarea. (no se realiza ningún cambio de proceso).

3.3. CFS.

El planificador completamente justo (Completely Fair Scheduler) (CFS) apareció en la versión 2.6.23 del kernel de Linux a principios de 2007 y ha sido el planificador por defecto desde entonces.

Su objetivo es que el usuario nunca vea una baja de rendimiento maximizando el uso de la CPU.

Éste planificador calcula cuánto tiempo debe ejecutarse una tarea en función de todos los procesos actualmente ejecutables. El tiempo que una tarea se ejecutará será calculado dividiendo su "peso" por el "peso" total de todos los procesos ejecutables, por lo que, CFS establece una cantidad de tiempo limitada. Por ejemplo, si tenemos 10 ms con 2 tareas con misma prioridad, cada tarea se ejecutará por 5 ms, si fueran 5 tareas, se ejecutarán en 2 ms.

En caso de que el tiempo no sea suficiente para planificar todas las tareas, simplemente se amplía.

Cada proceso tendrá un niceness, por ejemplo, con un tiempo de 20 ms y 2 procesos, uno con niceness 0 y otro con 5, la proporcionalidad será 1/3, por lo que el proceso con mayor prioridad tendrá un tiempo aprox de 15 ms mientras que el otro proceso tendrá 5 ms. Ahora si hay 2 procesos con niceness 5 y 10, el niceness absoluto es mayor que el ejemplo anterior, pero la proporcionalidad será la misma, por lo que la división de tiempo también será igual.

En general CFS se denomina justo, porque a cada proceso se le da una parte de la CPU en relación a los otros procesos que se van a ejecutar.

3.3.1. Runqueue.

El diseño de CFS es un rb-tree, cuyos valores de los nodos serán tiempos de ejecución virtual de los procesos. Cuanto más pequeño el valor, más a la

izquierda del árbol estará, por lo que el planificador siempre selecciona el nodo más a la izquierda como la tarea a ejecutar.

En <kernel / sched / sched.h>:

```
struct cfs_rq {
struct load_weight load; (Peso acumulado de todas las tareas
    ejecutables)
unsigned int nr_running ...; (Cantidad de tareas ejecutables)
u64 min_vruntime; (Tarea con menor tiempo de ejecucion)
struct rb,_root tasks_timeline; (Raiz del rb-tree)
struct rb_node *rb_leftmost; (Tarea siguiente a ejecutar, el
    mas a la izquierda)

struct sched_entity *curr, ...; (Entidad actualmente en
    ejecucion)
...
}
```

3.3.2. Entidades de Planificación.

El planificador predeterminado de Linux (CFS) puede trabajar con entidades más generales que tareas. Éstas entidades corresponden a un conjunto de tareas que son ejecutadas a la vez.

3.3.3. Reloj Virtual.

CFS utiliza un reloj virtual para calcular el tiempo que una tarea se ejecuta, cuando una tarea se ejecuta, su tiempo de ejecución virtual aumenta, por lo que se mueve a la derecha en el rb-tree. Los procesos que no han sido ejecutados por el procesador en un periodo largo de tiempo aumentan su prioridad, mientras que los que han sido ejecutados por mayor tiempo, reducen su prioridad.

3.4. Planificador en tiempo real.

En primeras versiones de Linux, el planificador era "real-time POSIX compatible" Cuando aparecen las clases de planificación, los procesos en tiempo real (rt process) pasan a ser manejados por un planificador aparte (definido en <kernel/sched/rt.c>). Es decir el CFS solo tiene que saber que los procesos en tiempo real existen.

Para el sistema, la diferencia entre un proceso rt y los demás es que al aparecer un proceso rt, este debe ser ejecutado a menos que haya otro proceso rt con mayor prioridad en ejecución. Así, para los procesos rt, existen dos tipos de políticas:

- SCHED_FIFO: es un planificador FIFO básico. Las tareas bajo esta política no tienen timeslices, se ejecutan hasta que se bloquean o ceden el CPU pero de forma voluntaria, o llega un proceso rt de más alta prioridad. Una tarea SCHED_FIFO precede a cualquier tarea SCHED_NORMAL. Si se tienen varias tareas SCHED_FIFO con la misma prioridad, se ejecutan bajo Round Robin.

- SCHED_RR: Similar al SCHED_FIFO pero con timeslices. Las tareas se ejecutan según Round Robin hasta que llega un proceso de más alta prioridad o hasta que su timeslice se acaba y pasan al final de la cola. El timeslice está definido (<include/linux/sched/rt.h>) y es de 100ms por defecto, se puede modificar con:

  ```
  #define RR_TIMESLICE (100 * HZ / 1000)
  ```

El kernel da prioridad estática a las tareas rt (no se recalcula) y solo se puede cambiar con el comando chrt. Con esto, se asegura que un proceso rt se ejecute antes de un proceso normal y se respete el orden entre procesos rt de distintas prioridades.

3.4.1. Reprensentación

Los procesos rt tienen una estructura de planificación predefinida en:

```
<include/linux/sched.h>

struct sched_rt_entity {
 struct list_head run_list;
 unsigned long timeout;
 unsigned long watchdog_stamp;
 unsigned int time_slice;
...
struct sched_rt_entity *back;
...
struct sched_rt_entity *parent;
 /* rq on which this entity is (to be) queued: */
struct rt_rq *rt_rq;
...
};
```

Para SCHED_FIFO, time_slice = 0

La runqueue principal tambien tiene sub-runqueue para procesos rt

```
<kernel/sched/sched.h>
 /* Real-Time classes' related field in a runqueue: */
struct rt_rq {
 struct rt_prio_array active;
 unsigned int rt_nr_running;
...
 struct rq *rq;      /* main runqueue */
 };

/* This is the priority-queue data structure of the RT
    scheduling class: */
struct rt_prio_array {
  /* include 1 bit for delimiter */
  DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1);
  struct list_head queue[MAX_RT_PRIO];
 };
```

Las tareas rt con igual prioridad se guardan en active.queue[prio] (lista ligada), y hay un bitmap (active.bitmap) que señala que una cola particular no está vacía.

Como en CFS, para planificadores RT hay una funcion update_curr_rt() que guarda el tiempo de CPU utilizado por el proceso, registra estadísticas, actualiza el timeslice y llama al planificador cuando sea conveniente. Los cálculos son realizados con tiempo actual, no con reloj virtual.

Al tratar con tiempo real en Linux no hay garantías, sólo existe lo que se llama "Soft Real Time" (puede aceptar ciertos errores, pero no demasiados).

3.5. BFS.

A 8 años del lanzamiento del planificador "Brain Fuck Schedule", mejor conocido por sus siglas "BFS", ha mantenido sus metas desde un inicio. Al ser una alternativa al planificador **CFS** *"Completely Fair Scheduler"* (ya que este último es el planificador predeterminado en el kernel principal) el BFS cuenta con un diseño con mayor simplicidad que el CFS; el motivo es que el "Brain Fuck Scheduler" está orientado a mejorar la experiencia de planificación de la CPU para computadores de versión de escritorio y sistemas que cuenten con pocos núcleos; lo que deja claro que no es un reemplazo para el planificador CFS al no hacerlo ideal para estaciones de trabajo que requieran mayor potencia (como lo son los servidores).

3.5.1. Agregar una tarea.

Para agregar una tarea, el planificador BFS las inserta en sus respectivas colas como una inserción del planificador *O(1)*. Al principio de insertar la tarea, se deberá comprobar cada una de las colas en ejecución, con el motivo de percatarse si la nueva tarea que será agregada en cola puede ejecutarse en cualquiera de las colas ociosas (las cuales son colas en donde las tareas se ejecutarán solo si no hay otros procesos que realizar). De esta manera es como el BFS sobresale de los demás planificadores, a pesar de que este enfocado para un hardware simple, se encarga de obtener latencia relativamente baja.

3.5.2. Seguridad en los datos globales/locales

El planificador, al trabajar sobre una cola de ejecución global, necesita brindar la seguridad de obtener una exclusión en los datos sobre las operaciones a realizar para proteger los datos del proceso de la tarea en ejecución

de manera local. Si se desea agregar, hacer una eliminación o una modificación de los datos de una tarea que se encuentra en la cola de ejecución global, está claro que deberá adquirir el bloqueo global. La tarea al obtener la CPU, está cuenta con una copia propia de los datos locales de la tarea, que únicamente esa CPU asignada tiene el privilegio de acceder y modificarlos. La CPU al finalizar la ejecución de una de las tareas, está la deberá eliminar de la cola global, ya que fue realizada con éxito.

3.5.3. Fecha limite Virtual

Con Kolivas implementó un *"tiempo límite"* (virtual deadline) para las ejecuciones de las tareas, con el fin de obtener un nivel de latencia bajo y llevar la justicia en la planificación de estas. El planificador deberá elegir entre la sub-cola de prioridad (en ella existen tareas que cuentan con diferentes propiedades) y las que se ejecutan en tiempo real (las cuales son ejecutadas por medio del algoritmo de planificación *Round Robin*), para poder ejecutar una tarea. Es por eso que se implementa una fecha límite virtual para que se garantice que de las dos colas se estarán ejecutando tareas, independientemente si son de tiempo real o no. El planificador otorgará la CPU a una tarea, además de asignarle un **time_slice** (igual al intervalo asignado al algoritmo *Round Robin*) y un plazo que será virtual. La fecha límite será establecida por la siguiente ecuación:

$$jiffies + (prio\_ratio * rr\_interval)$$

Donde: *jiffies*: Su único uso es almacenar el número de señales producidas desde el inicio del sistema. Se incrementa en uno. *prio_ratio*: Significa prioridad, proporcional a *'nice'* dada por el usuario, aumenta un 10 % cada *nice level* a partir de *nice -20*. *rr_interval*: Intervalo en "Round Robin".

3.5.4. Busqueda de Tareas.

Una vez que una tarea actual termine, se tendrá que realizar la búsqueda de cual tarea será la siguiente para adquirir el tiempo de CPU. Existen 103 colas de prioridad, las cuales, las primeras 100 son de tiempo real. Después, la cola número 101 corresponde a la prioridad de planificación asíncrona. La 102 está dedicada a la prioridad normal, y por último la 103 es correspondiente a la prioridad de planificación ociosa. Al principio deberá observar las tareas que cuenten con prioridad estática que están en la cola. Si el planificador BFS encuentra una tarea que cuenta con alguna propiedad en tiempo real, se deberá comprobar la cola en la que está y se

deberá tomar la primera tarea enumerada en la cola. Si la prioridad llegara a ser a una tarea **SCHED_ISO**, se deberá hacer de manera *FIFO (First In First Out)*. En otro caso, si la prioridad llegara a ser **SCHED_NORMAL** o **SCHED_IDLEPRIO**, la nueva búsqueda de la tarea se convertirá en *O(n)*, se deberá verificar cual es la que tiene el menor tiempo. En el último de los casos, si el plazo de la tarea finalizara provocará que la búsqueda aborte y continúe con la siguiente (recordando que la manera en la se está implementada la cola es *FIFO*).

```
  #define MAX_USER_RT_PRIO 100
  #define MAX_RT_PRIO  (MAX_USER_RT_PRIO + 1)
  #define ISO_PRIO    (MAX_RT_PRIO)
  #define NORMAL_PRIO  (MAX_RT_PRIO + 1)
  #define IDLE_PRIO  (MAX_RT_PRIO + 2)
  #define PRIO_LIMIT  ((IDLE_PRIO) + 1)
struct global_rq {
  raw_spinlock_t lock;
  unsigned long nr_running;
  unsigned long nr_uninterruptible;
  /* ... */
  struct list_head queue[PRIO_LIMIT];
  DECLARE_BITMAP(prio_bitmap, PRIO_LIMIT + 1);
    u64 niffies; /* En nanosegundos*/
}
```

3.5.5. Ejecución de Tareas.

El BFS cuenta con 3 maneras de elegir cual será la siguiente tarea a ejecutar: la primera forma es cuando el *time_slice*, asignado a la tarea, se termina. Esto quiere decir que una tarea se tendrá que ejecutar fuera de su *time_slice*, es por eso que se deberá rellenar y volver a actualizar la fecha límite. La segunda forma es cuando una tarea se va a dormir y no necesita de la CPU. En este caso se mantiene el mismo *time_slice* y la fecha límite y se volverá a retomar cuando esta despierte. Y por último, es cuando una tarea despierta y cuenta con una prioridad mayor que la prioridad que tiene la tarea que actualmente se está ejecutando. Aquí, el planificador asigna la CPU a la tarea con mayor prioridad.

3.5.6. Escalabilidad.

El creador de este planificado BFS, deja en claro que la única limitante es la escalabilidad. Como ya se ha mencionado antes, el planificador *Brain Fuck Scheduler* está orientado para un hardware de usuarios promedio, los cuales cuentan con un computador de escritorio (ya que al ser un planificador simple, la necesidad de adquirir una escalabilidad mayor es incongruente si solo se cuenta con un hardware de ese tipo).

3.5.7. Planificación Isócrona.

Se refiera a que se realiza *"en el mismo tiempo"* de la planificación. Está orientada a los usuarios, que no cuentan los permisos/privilegios de un súper-usuario, al brindarles un rendimiento muy parecido al tiempo real. Esto gracias al implementar la llamada **SCHED_ISO**.

La forma en que se implementa esta función es cuando alguna tarea necesita la obtención de prioridad en tiempo real, pero el usuario carece de los permisos necesarios.

3.5.8. Planificación Idlepro.

Su implementación es muy parecida a la forma *SCHED_ISO*. En este tipo de planificación, se implementa la forma **IDLE_PRIO**, en la cual son colas en donde las tareas se ejecutarán solo si no hay otros procesos que realizar. Estos dos tipos de planificaciones son más que etiquetas para las tareas al tratar de obtener prioridad pero se carece de los permisos y/o privilegios del súper-usuario.

4. Conclusión.

Un planificador de procesos es una parte fundamental de un sistema operativo, la importancia del planificador radica en que con él, la percepción visual que tiene el usuario se ve afectada ya que para él las cosas están ocurriendo al mismo tiempo.

Hemos visto la evolución del planificador de Linux desde su origen, el cómo operaban y sus grandes desventajas. Aunque existen dos factores importantes que determinan el buen funcionamiento del planificador: el avance tecnológico y los requerimientos del usuario. El avance tecnológico ha ayudado a facilitar distintos ámbitos de nuestra vida personal y/o profesional, sin embargo, no todos los planificadores trabajan bajo el mismo escenario. Es por eso que se realizan actualizaciones con algunas mejoras para aprovechar estas nuevas tecnologías, además de que el usuario que las opera, se le ofrezca una mejor experiencia. En el caso de usuarios promedio que cuentan con un computador de escritorio, el planificador *BFS* cuenta con un algoritmo simple haciendo que este aproveche sus recursos al máximo, disminuyendo la latencia. En cambio, utilizar este planificador no es el más óptimo para utilizar en arquitecturas más costosas de hardware.

Hasta la fecha los planificadores más recientes, como lo es el *CFS* (dado que es el planificador predeterminado de Linux) y el *BFS*, tienen como base algoritmos de planificadores antiguos como el *O(n)* y el *O(1)*. Ahora bien, en un futuro, quizá no tan lejano, estos planificadores tendrán que seguirse actualizando, aunque llegarán a un punto en que perderán su principal esencia, y es ahí donde darán paso a nuevos planificadores que puedan ofrecer una solución a los nuevos problemas que llegaran a surgir con los nuevos avances tecnológicos y el tipo de exigencias que las tareas generen.

Referencias

[1] Ishkov, N. A complete guide to linux process scheduling. *M.Sc. Thesis* (Feb 2015).