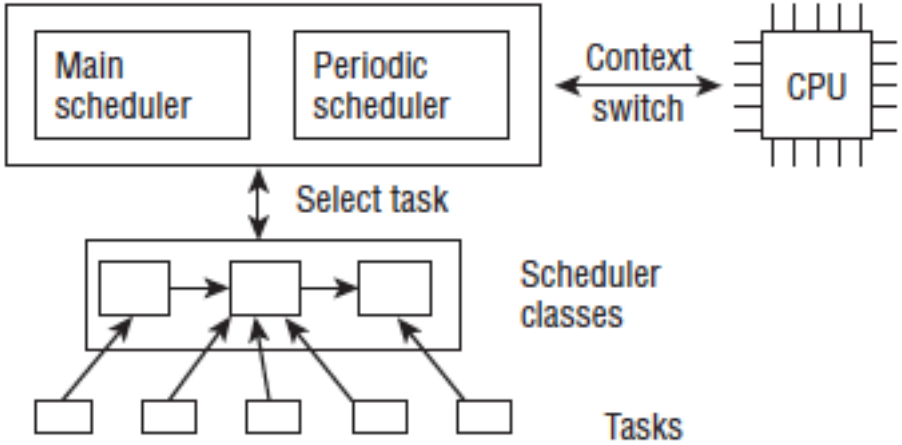
3. PLANIFICACION EN LINUX (Kernel 2.6.24)

3.1 Una visión global de la planificación [Mau08 2.5.2] [Lov10 pág.46].

El planificador de Linux es modular, permitiendo que diferentes algoritmos de planificación se apliquen a diferentes tipos de procesos. Esto se implementa con el concepto de clases de planificación.



* El kernel dispone de diferentes clases de planificación, cada una tiene asociada una política de planificación, y cada una de las cuales tendrá asociado un conjunto de procesos:

planificación de tiempo real

planificación neutra o limpia (CFS: Completely Fair Scheduling): para procesos normales (identificados como SCHED_NORMAL)

planificación de la tarea "idle" (cuando no hay trabajo que realizar)

* Cada clase de planificación tiene una prioridad; el planificador principal va recorriendo las distintas clases de orden de mayor a menor prioridad, encontrando la primera que no está vacía, y el método de planificación asociado determina el siguiente proceso a ejecutar.

* En la figura se muestra ...

una **parte del planificador que se activa periódicamente**,

el **planificador principal** (función schedule que se verá más tarde)

Elije el siguiente proceso a ejecutar y provocará un cambio de contexto;

Se activa cuando el proceso actual no desee seguir ejecutándose bien porque se bloquea o finaliza, o por la llegada de un nuevo proceso ejecutable o el desbloqueo de uno preexistente que resulta tener una mayor prioridad que el actual.

3.2 Estructuras de datos [Mau08 2.5.2]

a) Elementos en la task_struct para la planificación

```
int prio, static_prio, normal_prio;
```

static_prio es la prioridad estática o nominal de un proceso:

se asigna al proceso cuando es creado; puede ser modificado con las llamadas nice y sched_setscheduler.

normal_prio y prio son las prioridades dinámicas del proceso.

normal_prio se calcula en función de static_prio y de la política de planificación del proceso. Es heredada por el hijo tras un fork.

prio es la prioridad que influye en cómo este proceso es elegido para obtener CPU;

se calcula a partir de static_prio, de la trayectoria anterior del proceso y del tipo de proceso (normal o de tiempo real)

```
const struct sched_class *sched_class;
```

sched_class es la clase de planificación a la que pertenece el proceso.

```
struct sched_entity se;
```

se es la entidad de planificación asociada al proceso

La planificación no opera únicamente sobre el concepto de proceso, sino que maneja conceptos más amplios en el sentido de manejar grupos de procesos;

> por ejemplo, el tiempo de CPU podría repartirse entre el grupo de procesos creados por un mismo usuario y entre ellos se reparte el tiempo asignado al grupo. Surge así el concepto de entidad de planificación.

Una entidad de planificación se representa mediante una instancia de sched_entity

En el caso más simple, la planificación se realizaría a un nivel de procesos, caso al que nos referimos en general en el presente estudio.

Puesto que el planificador está diseñado para trabajar con entidades de planificación, cada proceso debe poder verse como una entidad. Así, el campo se alberga una instancia de sched_entity sobre la que el planificador opera.

```
unsigned int policy;
```

policy es la política de planificación que se aplica al proceso. Puede tener uno de estos cinco valores:

Políticas manejadas por el planificador CFS....

SCHED_NORMAL: se aplica a los proceso normales en los que nos centramos (frente a los de tiempo real)

SCHED_BACH: tareas menos importantes, concretamente procesos bach con gran proporción de uso de CPU para cálculos. Los procesos de este tipo son considerados menos importantes por el planificador: nunca pueden desplazar a otros procesos y por tanto no podrán molestar a los procesos interactivos.

SCHED_IDLE: tareas de este tipo tienen un peso mínimo para ser elegidas para asignación de CPU

Políticas manejadas por el planificador de tiempo real....

SCHED_RR y SCHED_FIFO: métodos Round-Robin y FIFO respectivamente.

```
cpumask_t cpus_allowed;
```

cpus_allowed es un campo de bits usado en sistemas multiprocesador para restringir en qué CPUs se puede ejecutar un proceso.

```
struct list_head run_list;
.....
unsigned int time_slice;
```

Los campos run_list y time_sile se usan en la planificación Round-Robin de los procesos de tiempo real

run_list denota la cabeza de lista donde está el proceso;

time_slice es el resto de tiempo que queda por consumir.

b) Estructura thread_info

* Tan importante como los datos anteriores es el flag TIF_NEED_RESCHED (de la estructura thread_info del proceso): cuando está establecido el kernel sabe que debe redecidirse a qué proceso darle la CPU.

* Describamos en más detalle la estructura thread_info, estrechamente unida a la task_struck como se estudiaba anteriormente.

Contiene datos sobre los procesos dependientes de la arquitectura; aunque se define de forma diferente de un procesador a otro, su contenido es similar al siguiente en la mayoría de los sistemas:

```
struct thread_info {
        struct task_struct *task; /* main task structure */
        struct exec_domain *exec_domain; /* execution domain */
        unsigned long flags; /* low level flags */
        unsigned long status; /* thread-synchronous flags */
        __u32 cpu; /* current CPU */
        int preempt_count; /* 0 => preemptable, <0 => BUG */
        mm_segment_t addr_limit; /* thread address space */
        struct restart_block restart_block;
                /* necesitado para implementar el mecanismo de señales */
}
```

task: puntero a la task_struct del proceso

flags: contiene varios flags específicos del proceso, dos de los cuales son particularmente interesantes para la planificación:

TIF_SIGPENDING está establecido si el proceso tiene señales pendientes

TIF_NEED_RESCHED está establecido si se debe activar al planificador para que elija el proceso que debe ejecutarse

cpu: nº de CPU en que el proceso se está ejecutando

preempt_count: contador para implementar la apropiación en modo kernel, se verá más tarde.

addr_limit: hasta qué dirección del espacio de direcciones virtuales puede ser utilizada por el proceso. Este límite existe para procesos normales, mientras que las hebras kernel pueden acceder al la totalidad de las direcciones, incluyendo las partes pertenecientes únicamente al kernel.

c) Clases de planificación

Las clases de planificación relacionan el planificador general y los diversos métodos de planificación

Cada método de planificación está representado por diversos punteros a funciones recogidos en la estructura de datos sched_class. Cada operación que pueda ser requerida al planificador es representada por un puntero.

```
struct sched_class {
   const struct sched_class *next;
   void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
   void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
   void (*yield_task) (struct rq *rq);
   void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);
   struct task_struct * (*pick_next_task) (struct rq *rq);
   void (*put_prev_task) (struct rq *rq, struct task_struct *p);1
...
   void (*set_curr_task) (struct rq *rq);
   void (*task_tick) (struct rq *rq, struct task_struct *p);
   void (*task_new) (struct rq *rq, struct task_struct *p);
};
```

* Una instancia de struct sched_class existe para cada clase de planificación; cada proceso tiene en su descriptor a qué clase de planificación pertenece.

* Las clases de planificación están relacionadas en una jerarquía plana:

los procesos de tiempo real son los más importantes,

después están los procesos normales (con una política CFS),

y tras ellos los procesos calificados como idle task.

* El elemento next conecta las diferentes instancias de sched_class en el orden anterior.

* Esta jerarquía se establece en tiempo de compilación y no hay forma de añadir una nueva clases de planificación dinámicamente.

* Las operaciones que se pueden realizar sobre cada clase de planificación son:

enqueue_task añade un nuevo proceso a la runqueue; ocurre cuando un proceso cambia de estado dormido a ejecutable.

dequeue_task elimina un proceso de una runqueue; ocurre cuando un proceso deja de estar en estado ejecutable o cuando el kernel decide apropiarse del proceso.

Aunque se utiliza el término “cola”, recalquemos que no necesariamente los algoritmos de planificación gestionen sus procesos mediante una cola sino con otras estructuras de datos (como por ejemplo el rbtree de CFS).

yield_task. Cuando un proceso quiera dejar el control de la CPU voluntariamente puede usar la llamada al sistema sched_yield que dispara yield_task en la instancia de sched_class de la clase a la que pertenece.

check_preempt_curr se usa para retirar la CPU al proceso actual.

pick_next_task y put_prev_task se usan para dar el control a una tarea y retirárselo, respectivamente (aunque para realizar el cambio de contexto se necesiten adicionalmente operaciones a bajo nivel)

set_curr_task permite cambiar la política de planificación de un proceso

task_tick es ejecutada por el planificador periódico cada vez que es activado.

new_task es ejecutada en la creación de un proceso (lo cual conecta fork y el planificador) y cuando un proceso se despierta.

* Cuando un proceso es incluido en una runqueue, el elemento on_rq de la instancia de sched_entity empotrada en la task_struct es establecido a 1, en los restantes casos vale 0.

* Las aplicaciones en el espacio de usuario no interaccionan directamente con las clases de planificación sino únicamente con las constantes SCHED* definidas anteriormente.

El kernel relaciona cada valor SCHED* con la clase de planificación oportuna:

SCHED_NORMAL, SCHED_BATCH y SCHED_IDLE

se relacionan con fair_sched_class

SCHED_RR y SCHED_FIFO se relacionan con rt_sched_class

Tanto fair_sched_class como rt_sched_class son instancias de struct sched_class que representan respectivamente el planificador CFS y el planificador de tiempo real.

d) Colas de ejecución[Mau08 2.5.2]

Son la estructura central del planificador.

Cada CPU tiene su propia cola de ejecución;

cada proceso activo aparece en solo una cola de ejecución .

No es posible ejecutar un proceso en varias CPUs al mismo tiempo.

> Sin embargo, hebras que se originan de un mismo proceso se pueden ejecutar en diferentes procesadores puesto que la gestión de procesos no hace distinción entre procesos y tareas.

Las colas de ejecución se implementan en la siguiente estructura de datos (de <include/kernel/sched.c>) que presentamos simplificada (se han eliminado elementos de naturaleza estadística y alusivos a sistemas multiprocesador):

```
struct rq {
        unsigned long nr_running;
        #define CPU_LOAD_IDX_MAX 5
        unsigned long cpu_load[CPU_LOAD_IDX_MAX];
        ...
        struct load_weight load;
        struct cfs_rq cfs;
        struct rt_rq rt;
        struct task_struct *curr, *idle;
        u64 clock;
    ...
};
```

nr_running: nº de procesos ejecutables en la cola independientemente de su prioridad o clases de planificación.

load: medida de la carga actual de la cola de ejecución (esencialmente, es proporcional al numero de procesos activos en la cola, considerándose el peso asociada la prioridad de cada uno de ellos).

cpu_load: permite rastrear cómo ha sido la carga en el pasado.

cfs y rt son sub-colas de ejecución empotradas, asociadas al planificador CFS y al planificador de tiempo real respectivamente.

curr: puntero a la task struct del proceso actual

idle: puntero a la task struct del proceso "idle" llamado cuando no hay procesos ejecutables.

clock y prev_raw_clock: se usan en la implementación de un reloj por cola

Todas las colas de ejecución del sistema se mantiene en un array de colas de ejecución, que contiene un elemento por cada CPU en el sistema.

e) Entidades de planificación

Puesto que el planificador puede operar con entidades más generales que tareas, se define la estructura sched_entity (en <include/linux/sched.h>)

```
struct sched_entity {
        struct load_weight load; /* for load-balancing */
        struct rb_node run_node;
        unsigned int on_rq;
        u64 exec_start;
        u64 sum_exec_runtime;
        u64 vruntime;
        u64 prev_sum_exec_runtime;
        ...}
```

- load: peso de esta entidad
- run_node: elemento de un árbol, así la entidad puede ser ordenada en un red-blak tree
- on_rq: expresa si la entidad está actualmente planificada en una cola de ejecución o no.
- y los últimos campos informan sobre los tiempos de CPU consumidos para así tener caracterizado el comportamiento del proceso.

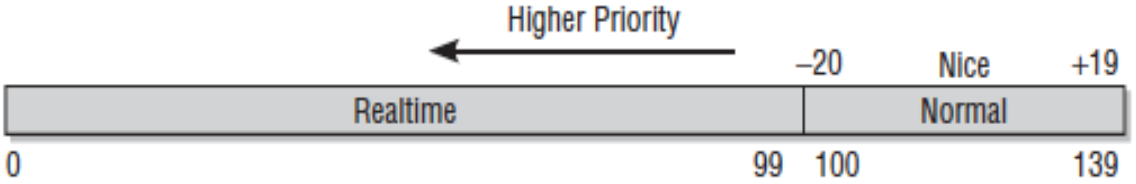
3.3 Sobre prioridades [Mau08 2.5.3]

* La prioridad estática de un proceso se establece en el espacio de usuario mediante la llamada al sistema nice; el valor nice de un proceso está en el intervalo [-20,+19]

* Internamente el kernel usa el intervalo [0,139]; los valores de nice en el intervalo [-20,+19] son trasladados al rango [100,139].

* Rango de valores de prioridad para static_prio:

- **[0, 99]** Prioridades para procesos de **tiempo real**.
- **[100, 139]** Prioridades para los procesos normales o regulares.



* El kernel calcula las prioridades dinámicas de la siguiente manera;

	normal_prio	**prio**
procesos normales	static_prio	static_prio
procesos de tiempo real	MAX_RT_PRIO - 1 – rt_priority MAX_RT_PRIO = 99 rt_priority es la prioridad para procesos de tiempo real almacenado en task_struct	

3.4 El planificador periódico [Mau08 2.5.4]

* El planificador periódico se implementa en scheduler_tick,

función llamada automáticamente por el kernel con frecuencia HZ

(constante cuyos valores están normalmente en el rango 1000 y 100Hz)

* Tareas principales:

actualizar estadísticas del kernel

activar el método de planificación periódico de la clase de planificación a que corresponde el proceso actual (task_tick)

* Cada clase de planificación tiene implementada su propia función task_tick

* Si se concluye que hay que replanificar, el planificador de la clase de planificador en cuestión activará el flag TIF_NEED_RESCHED asociado al proceso en su thread_info, y provocará que se llame al planificador principal.

3.5 El planificador principal [Mau08 2.5.4]

* Se implementa en la función schedule, invocada en diversos puntos del kernel para tomar decisiones sobre asignación de la CPU.

* La función schedule es invocada de forma explícita cuando un proceso se bloquea o termina.

* el kernel chequea el flag TIF_NEED_RESCHED del proceso actual al volver al espacio de usuario desde modo kernel (ya sea al volver de una llamada al sistema o en el retorno de una interrupción y si está establecido llama a schedule.

* Actuación de schedule

- Determina la actual runqueue y establece el puntero prev a la task_struct del proceso actual

- Actualiza estadísticas y limpia el flag TIF_NEED_RESCHED

- Si el proceso actual estaba en un estado TASK_INTERRUPTIBLE y ha recibido la señal que esperaba, se establece su estado a TASK_RUNNING

- Se llama a pick_next_task de la clase de planificación a la que pertenezca el proceso actual para que se seleccione el siguiente proceso a ejecutar, se establece next con el puntero a la task_struct de dicho proceso seleccionado.

- Si hay cambio en la asignación de CPU, se realiza el cambio de contexto llamando a context_switch

3.6 Cambio de contexto [Lov10 pág. 62]

Función context_switch (<include/kernel/sched.c>): interfaz con los mecanismos de bajo nivel dependientes de la arquitectura que se deben realizar cuando hay un cambio en la asignación de CPU.

Básicamente, context_switch realiza estas dos acciones:

- Lama a switch_mm (declarada en <asm/mmu_contexto.h>) donde se realiza el cambio del mapa de memoria del proceso previo al proceso nuevo.

- Llama a switch_to (declarada en <asm/system.h>) que realiza el cambio del estado del procesador correspondiente al proceso previo por el correspondiente al proceso nuevo. Esto involucra salvar y restaurar la información de pila, los registros del procesador y demás información de estado dependiente del procesador que sea particular de cada proceso.

3.7 La clase de planificación CFS (Completly Fair Scheduler) [Lov10 pag48-50] [Mau08 2.6]

a) Conceptos básicos de CFS

* Objetivo básico de CFS o “Completly Fair Scheduler”: cada proceso debe recibir 1/n del tiempo de CPU (siendo n el numero de procesos)

Una posibilidad sería implementar una política Round Robin en que el valor del quantum no es fijo sino que se calcula en función del número total de procesos ejecutables.

* **Latencia**

El ideal sería que en cualquier periodo de tiempo, *por pequeño que fuera*, hubieran visto la CPU los n procesos, pero eso dispararía el número de cambios de contexto.

Se establece un valor al menor periodo de tiempo en que se asegura que todos los procesos han sido elegidos para ejecutarse (targeted latency o latencia).

A menor valor de latencia se consigue un mejor reparto del tiempo de cpu entre los procesos pero un mayor coste en tiempo de cpu para cambios de contexto.

*** Granularidad mínima**

Si el número de procesos tiende a infinito la proporción de tiempo asignada a cada proceso tiende a cero con la consiguiente elevación excesiva del coste en cambios de contexto.

Se introduce el concepto de granularidad mínima (por omisión 1ms):

Es la cantidad de tiempo que se asegura que disfruta un proceso la CPU de forma consecutiva aunque el número de procesos tendiera a infinito

(siendo estrictos, habría algún momento de tiempo en que no se proporcionaría reparto equitativo).

* Si contemplamos ahora procesos con distintas prioridades, CFS repartiría el tiempo de cpu de modo que todos los procesos de la misma prioridad tengan igual proporción de uso de CPU.

b) Implementación de CFS en Linux 2.6.24

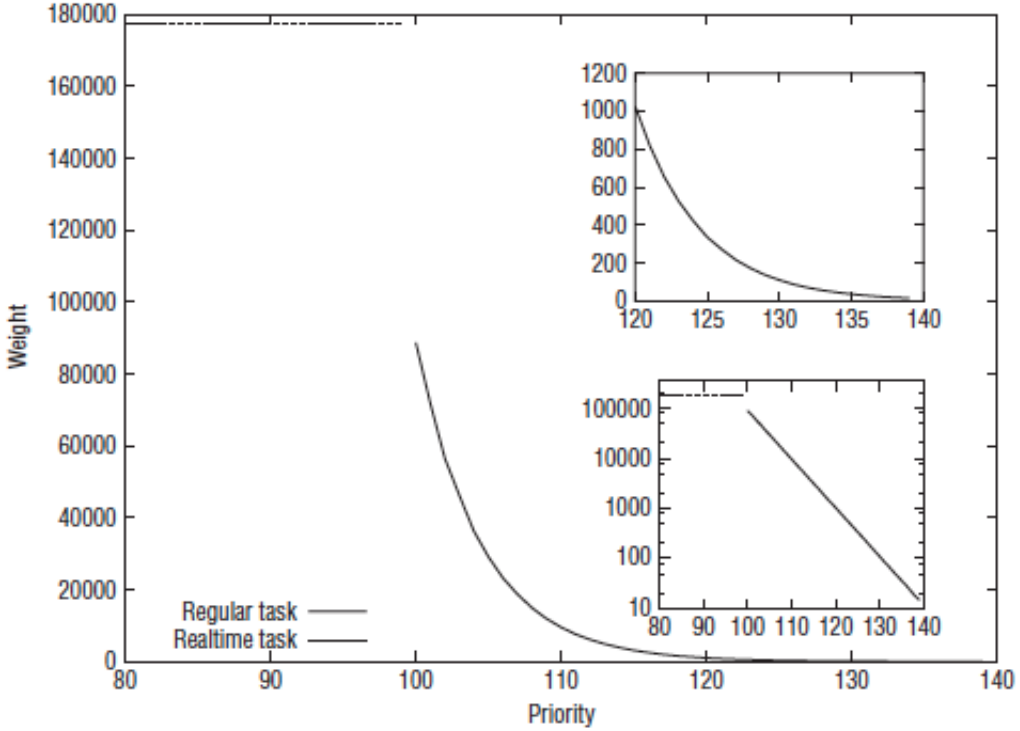
* Se define la clase de planificación fair_sched_class asociada a la política de planificación CFS.

Se han ido describiendo anteriormente cuándo se llaman las funciones aludidas en fair_sched_class por el planificador principal.

* Se mantienen datos sobre los tiempos consumidos por los procesos en la estructura sched_entity.

* El kernel calcula una magnitud llamada peso para cada proceso en la entidad de planificación se de task_struct (omitimos los cálculos exactos).

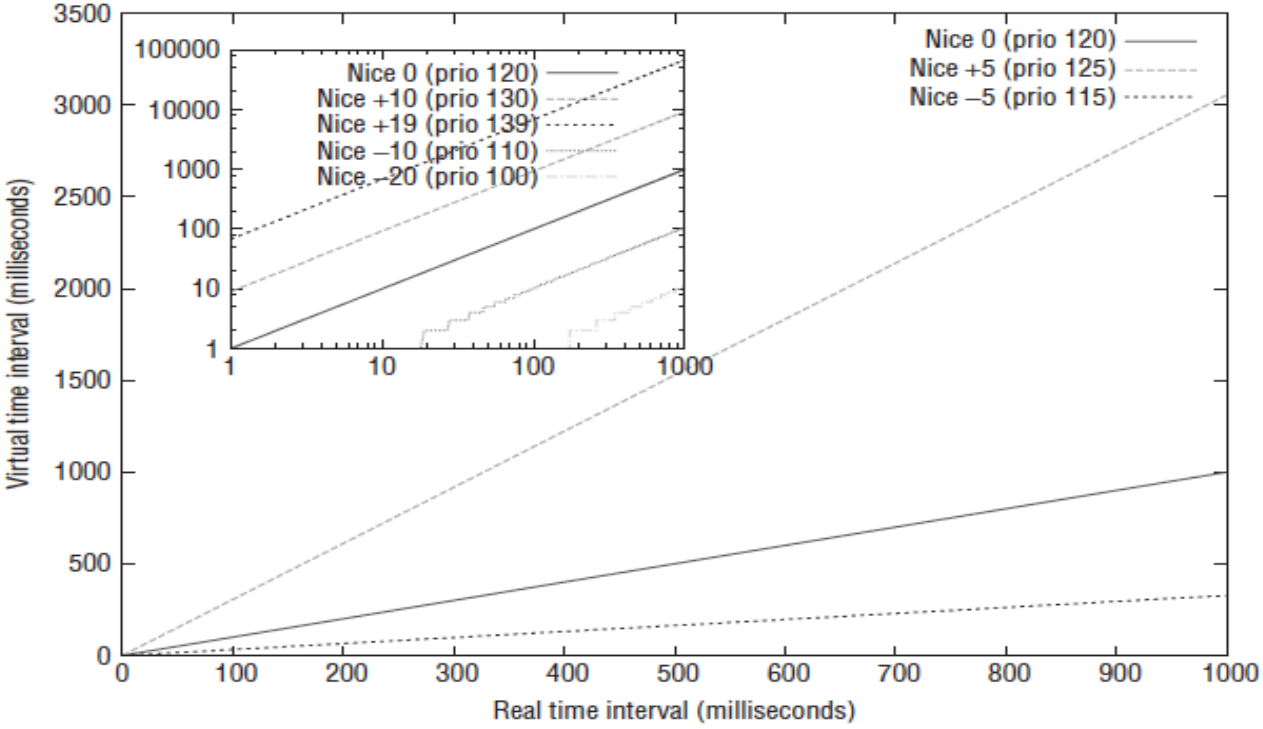
En la figura vemos la relación entre peso y prioridad estática para procesos regulares y de tiempo real (los procesos iddle tendrían un valor de peso mínimo):



*

* El elemento vruntime (virtual runtime) de sched_entity almacena el así llamado *tiempo virtual* que un proceso ha consumido:

se calcula a partir del tiempo real que el proceso ha hecho uso de la CPU, su prioridad estática y su peso



* El valor vruntime del proceso actual se actualiza ...

- periódicamente
- cuando llega un nuevo proceso ejecutable
- cuando el proceso actual se bloquea

* Cuando se decide qué proceso ejecutar a continuación, se elige el que tenga un valor menor de vruntime.

* Para realizar esto CFS utiliza un rbtree (red blak tree):

estructura de datos que almacena nodos identificados por una clave y que permite una eficiente búsqueda dada una determinado valor de la clave.

En el kernel se implementan funciones para gestionar esta estructura de datos: añadir nodos, eliminar nodos, seleccionar nodos....

* Cuando un proceso va a entrar en estado dormido (bloqueado)...

- será añadido a una cola asociada con la fuente de bloqueo
- se establece el estado del proceso a TASK_INTERRUPTIBLE
  o a TASK_NONINTERRUPTIBLE según sea conveniente
- es eliminado del rbtree de procesos ejecutables
- se llama a schedule para que se elija un nuevo proceso a ejecutar.

* Cuando un proceso vuelve del estado dormido....

- se cambia su estado a ejecutable
- se elimina de la cola de bloqueo en que estaba
- se añade al rbtree de procesos ejecutables

3.8 La clase de planificación de tiempo real[Lov10 pág 64-65][Mau08 2.7]

* Se define la clase de planificación rt_sched_class asociada a los procesos de tiempo real.

* Los procesos de tiempo real tienen son más prioritarios que los normales, y mientras existan procesos de tiempo real ejecutables éstos serán elegidos frente a los normales.

* Linux 2.6 tiene dos políticas de planificación de tiempo real:

Roun-Robin (SCHED_RR): algoritmo de planificación Round Robin. Las tareas que siguen este tipo de planificación hacen uso de la CPU durante un intervalo de tiempo predeterminado. Una tarea con mayor prioridad siempre pasa por delante de una tarea menos prioritaria.

FIFO (SCHED_FIFO): algoritmo FIFO; se ejecuta hasta que se bloquea o termina.

Una tarea podría ejecutarse de forma indefinida.

Puede ser expulsada de la CPU por otra tarea de mayor prioridad que siga una de las dos políticas de planificación de tiempo real.

* Un proceso de tiempo real queda determinado por la prioridad que tiene cuando se crea, el kernel no incrementa o disminuye su prioridad en función de su comportamiento.

*** Planificación en Tiempo real ligero (soft Real-Time) en Linux.**

En general, los sistemas de tiempo real se pueden clasificar según lo siguiente:

- Sistemas de tiempo real estricto (Hard real-time): Son aquéllos en los que es absolutamente imperativo que las respuestas se produzcan dentro del tiempo límite especificado.
- Sistemas de tiempo real no estricto (Soft real-time): Son aquéllos en los que los tiempos de respuesta son importantes pero el sistema seguirá funcionando correctamente aunque los tiempos límite no se cumplan ocasionalmente. También se conocen como sistemas de tiempo real ligero o flexible.

* Las políticas de planificación de tiempo real SCHED_RR Y SCHED_FIFO posibilitan que el kernel Linux pueda tener un comportamiento soft real-time.

En este tipo de comportamiento el kernel trata de planificar diferentes tareas dentro de un rango temporal determinado, pero no se garantiza la consecución de la planificación dentro del rango temporal, sino que simplemente se intenta.

3.9 Apropiación (expropiación) [Lov10 pag62-64].

* Los sistemas operativos multitarea pueden ser categorizados de dos modos diferentes: multitarea cooperativa y apropiativa (o expropiativa).

> **Planificación cooperativa**: un proceso se ejecuta hasta que voluntariamente decide dejar de hacerlo. El acto de que un proceso voluntariamente desee dejar de ejecutarse se denomina yielding. El planificador no puede tomar decisiones globales en base a cómo se están ejecutando los procesos.

> **Planificación apropiativa:** el planificador decide cuando una tarea debe ser interrumpida y expulsada de la CPU, y cuando una nueva tarea debe iniciar o reanudar su ejecución. Es el caso en que nos situamos.

* La apropiación se puede entender como el hecho de que el sistema operativo retire la asignación de la CPU al proceso actual aunque pudiera seguir ejecutándose cuando decide que hay otro procesos preferentes.

* Se puede distinguir entre dos tipos diferentes de expropiación: expropiación en modo usuario y expropiación en modo kernel.

Apropiación en modo usuario.

La apropiación en modo usuario se da cuando se retorna a modo usuario ya sea de una interrupción o de una llamada al sistema y se encentra activado el flag TIF_NEED_RESCHED; entonces se llama al planificador y se elige para ejecutarse a un proceso distinto al actual.

Puesto que se está volviendo a modo usuario, todas las actualizaciones del kernel se han terminado y es correcto tanto pasar la CPU a otro proceso como al proceso actual.

En resumen, puede ocurrir una apropiación en modo usuario

- Cuando se vuelve al espacio de usuario tras una llamada al sistema
- Cuando se vuelve al espacio de usuario tras un tratamiento de interrupción

Apropiación en modo kernel.

En un **kernel no apropiativo**, cuando el código del kernel está ejecutando un tratamiento de interrupción o llamada al sistema, se ejecuta entero este tratamiento hasta que termina.

Pero en un **kernel apropiativo**, podemos quitar el control de CPU a un proceso mientras éste está ejecutando código kernel que responde a un tratamiento de interrupción o llamada al sistema,

es decir se puede expulsar a un proceso en cualquier punto de su ejecución.

¿Qué precauciones de deben tomar?

Es posible expulsar una tarea de la CPU siempre y cuando el kernel se encuentre en un estado seguro, concepto que en general se define así:

el kernel está en un estado seguro si todas las estructuras del kernel están o bien actualizadas y consistentes o bien bloqueadas mediante algún mecanismo de sincronización.

* El kernel Linux utiliza locks para delimitar las zonas de código del kernel en que se está en un estado seguro:

> si una tarea tiene bajo su control algún lock entonces el kernel no está en un estado seguro

El kernel Linux puede llevar a cabo la apropiación de la CPU de una tarea mientras ésta no tenga bajo su control ningún lock, así se satisfarán los requisitos de reentrancia si se pasara el control ahora a otra tarea. Así es cómo se consigue que el kernel de linux es SMP-safe.

Se ha establecido un contador llamado **preempt_count** en la estructura thread_info. Este contador que, inicialmente vale 0, se va incrementando a medida que la tarea se hace con un lock y, se decrementa cuando lo libera.

Cuando se retorna del tratamiento de una interrupción a modo privilegiado,

se evalúan el flag TIF_NEED_RESCHED y el contador preempt_count (que no se evaluaba en la apropiación en modo usuario):

Si el flag TIF_NEED_RESCHED está activo:

si preempt_count =0 entonces el planificador será invocado.

si preempt_count != 0 entonces la tarea que está ejecutándose tiene en su poder algún lock y por tanto, no es seguro planificar otra tarea.

En este caso se vuelve de la interrupción y se deja la misma tarea que estaba ejecutándose. Cuando la tarea actual libera todos los locks y el contador retorna 0, el código del unlock verifica el flag TIF_NEED_RESCHED y, si está activo, el planificador es invocado.

La apropiación en modo privilegiado también se puede realizar de forma explícita, llamando explícitamente a la función schedule o bien bloqueando la tarea en ejecución.

No necesita codificación adicional para asegurar o comprobar en que estado se encuentra el kernel.

Se asume que en el código en que se invoca al scheduler se sabe si el estado es seguro, es decir, que no se tiene retenido ningún lock.

En resumen, la apropiación en modo privilegiado puede ocurrir...

- Cuando un manejo de interrupción termina y antes de volver al modo usuario.
- Cuando el código del kernel vuelve a ser expropiativo.
- Si una tarea en el kernel invoca explícitamente a la función schedule.
- Si una tarea en el kernel se bloquea con el consiguiente resultado de la invocación de schedule.

3.10 Particularidades en SMP [Mau08 2.8.1]

Ver Introducción a SMP: [Sta05 4.2]

* Aunque las estructuras de datos que se han visto están pensadas para adecuarse a un entorno SMP (Symmetric MultiProcessing o multiprocesamiento simétrico), será necesario incluir nuevos campos y nuevas estructuras de datos al pasar a SMP.

* Para realizar correctamente la planificación en un entorno SMP, el kernel deberá tener en cuenta estas consideraciones adicionales:

    Se debe repartir equilibradamente la carga entre las distintas CPUs

    Se debe tener en cuenta la afinidad de una tarea con una determinada CPU..

    El kernel debe ser capaz de migrar procesos de una CPU a otra (puede ser una operación costosa)

* Periódicamente una parte del kernel deberá comprobar que se da un equilibrio entre las cargas de trabajo de las distintas CPUs

Si se detecta que una tiene más procesos que otra, se reequilibran pasando procesos de una CPU a otra.

3.11 Llamadas al sistema alusivas a la planificación [Lov10 pag65 y ss].

Linux proporciona una familia de llamadas al sistema para la gestionar parámetros de planificación, y proporcionar mecanismos explícitos para que un proceso ceda (yields) la CPU a otro proceso. Son implementadas por funciones de librería en C.

System Call	Description
`nice()`	Sets a process's nice value
`sched_setscheduler()`	Sets a process's scheduling policy
`sched_getscheduler()`	Gets a process's scheduling policy
`sched_setparam()`	Sets a process's real-time priority
`sched_getparam()`	Gets a process's real-time priority
`sched_get_priority_max()`	Gets the maximum real-time priority
`sched_get_priority_min()`	Gets the minimum real-time priority
`sched_rr_get_interval()`	Gets a process's timeslice value
`sched_setaffinity()`	Sets a process's processor affinity
`sched_getaffinity()`	Gets a process's processor affinity
`sched_yield()`	Temporarily yields the processor