

marinamuca01

www.wuolah.com/student/marinamuca01

13891

Tema-2-Procesos-y-hebras-Linux.pdf

Resúmenes Teoría



2º Sistemas Operativos



Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada



Descarga la APP de Wuolah.
Ya disponible para el móvil y la tablet.





**KEEP
CALM
AND
ESTUDIA
UN POQUITO**

TEMA 2

Procesos y hebras en Linux

IMPLEMENTACIÓN DE PROCESO/HEBRA EN LINUX: TASK

Núcleo identifica a los procesos (task) por su PID (Process IDentifier).

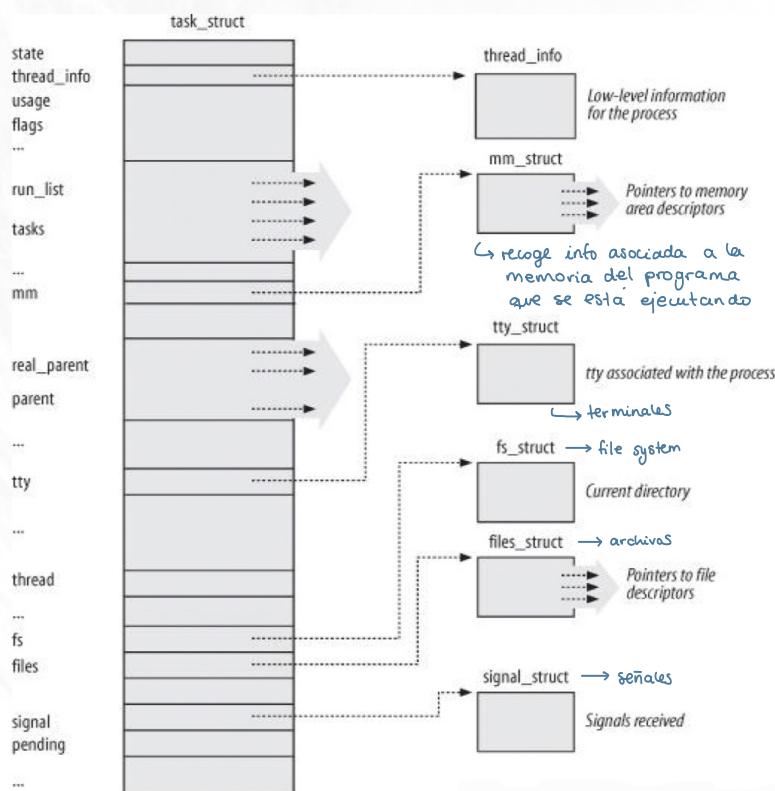
En Linux proceso = entidad que se crea con fork() (excepto el proceso 0) y clone().

Procesos especiales que existen durante la vida del sistema

{ Proceso 0 → creado "a mano" cuando arranca el sistema. Crea al proceso 1
Proceso 1 (init) → antecesor de cualquier proceso del sistema.

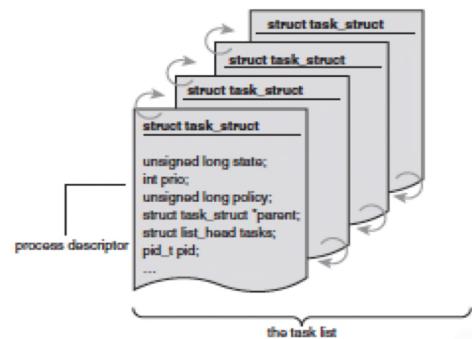
En Linux tarea (task) engloba el concepto de proceso y hebra.

PCB en Linux: struct task_struct



task list (lista de procesos) = Lista doblemente enlazada

Cada elemento es un descriptor de proceso (PCB) definido en </include/linux/sched.h>



```
struct task_struct { // del kernel 2.6.24
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    ...
    /* Información para planificación */
    int prio, static_prio, normal_prio;
    struct list_head run_list;
    const struct sched_class *sched_class;
    struct sched_entity se;
    ...
    unsigned int policy; // man ps -> policy
    cpumask_t cpus_allowed;
    unsigned int time_slice;
    ...
}
```

```

/* Memoria asociada a la tarea */
struct mm_struct *mm, *active_mm; // exec libera mm_struct y reserva nuevo
...
pid_t pid;
/* Relaciones entre task_struct */
struct task_struct *parent; /* parent process */
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children list */
/* Información para planificación y señales */
unsigned int rt_priority;
sigset_t blocked, real_blocked;
sigset_t saved_sigmask; /* To be restored with TIF_RESTORE_SIGMASK */
struct sigpending pending;
/*...*/
}

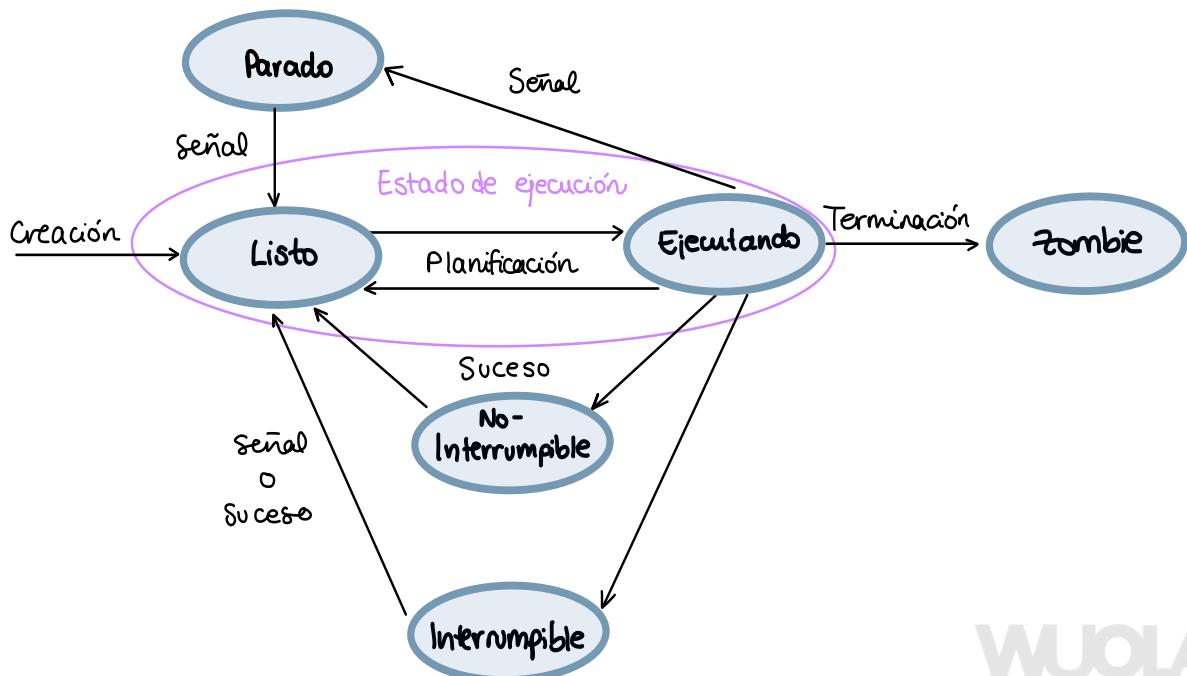
```

Estados de una tarea (task)

El campo **state** de **task_struct** especifica el **estado actual del proceso**:

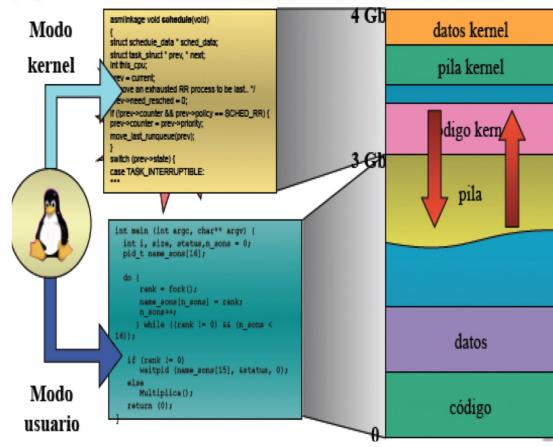
| | |
|-------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Ejecución TASK_RUNNING | Se corresponde con dos: ejecutándose o preparado para ejecutarse (en la cola de procesos preparados) |
| Interrumpible TASK_INTERRUPTIBLE | El proceso está bloqueado y sale de este estado cuando ocurre el suceso por el cual está bloqueado o porque le llegue una señal |
| No interruptible TASK_UNINTERRUPTIBLE | El proceso está bloqueado y sólo cambiará de estado cuando ocurra el suceso que está esperando (no acepta señales) |
| Parado TASK_STOPPED | El proceso ha sido detenido y sólo puede reanudarse por la acción de otro proceso (ejemplo, proceso parado mientras está siendo depurado) |
| TASK_TRACED | El proceso está siendo traceado por otro proceso |
| Zombie EXIT_ZOMBIE | El proceso ya no existe pero mantiene la estructura task hasta que el padre haga un wait() (EXIT_DEAD) |

Modelo de estados para tasks.

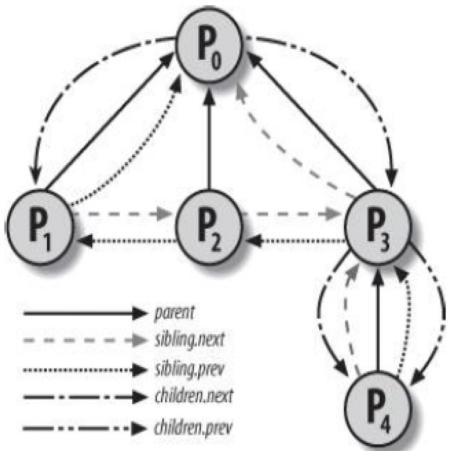


Espacio de direcciones en el contexto de un proceso

Acceso a memoria en modo user y modo Kernel dentro del contexto de un proceso



Árbol de procesos



- Si P0 hace un fork() y genera a P1
↳ P0 es el proceso padre y P1 el hijo
- Si P0 hace varias llamadas a fork() genera varios procesos hijo y la relación entre ellos es de hermanos (sibling)
- Todos los procesos son descendientes del proceso init (cuyo PID es 1)

{
 A **task_struct de su parent**: struct task_struct *parent;
 A **una lista de hijos** (children): struct list_head children;
 A **una lista de hermanos** (sibling): struct list_head sibling;

Cada task_struct tiene un puntero
Kernel dispone de procedimientos eficaces para las acciones usuales de manipulación de la lista

Implementación de hilos (threads)

Desde el punto de vista del Kernel no hay distinción entre hebra y proceso: **task = process / thread**

Linux implementa concepto de **hebra como proceso sin más, que comparte recursos con otros procesos**. Cada hebra tiene su propio **task_struct**.

clone() → llamada al sistema que crea un nuevo proceso o hebra dependiendo del parámetro **flags**.

```

#define _GNU_SOURCE
#include <sched.h>
int clone ( int (*fn) (void *), void *child_stack, int flags, void *arg);
  
```

clone() system call flags

CLONE_FILES
CLONE_FS
CLONE_SIGHAND
CLONE_THREAD
CLONE_VM
 ↳ comparten mm_struct

Parent and child share open files.
 Parent and child share filesystem information.
 Parent and child share signal handlers and blocked signals.
 Parent and child are in the same thread group.
 Parent and child share address space.

WUOLAH

ENCENDER TU LLAMA CUESTA MUY POCO



Hébras kernel

Se crean cuando es necesario o útil que el Kernel realice operaciones en segundo plano.
No tienen espacio de direcciones (puntero mm == NULL).

Se ejecutan únicamente en el espacio del Kernel.

Son planificadas y pueden ser expropiadas.

Solo se pueden crear por otra hébra Kernel mediante:

```
#include <include/linux/kthread.h>
struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
void *data, int node, const char namefmt[], ...);
```

Terminan cuando realizan una operación do_exit() o cuando otra parte del Kernel provoca su finalización.

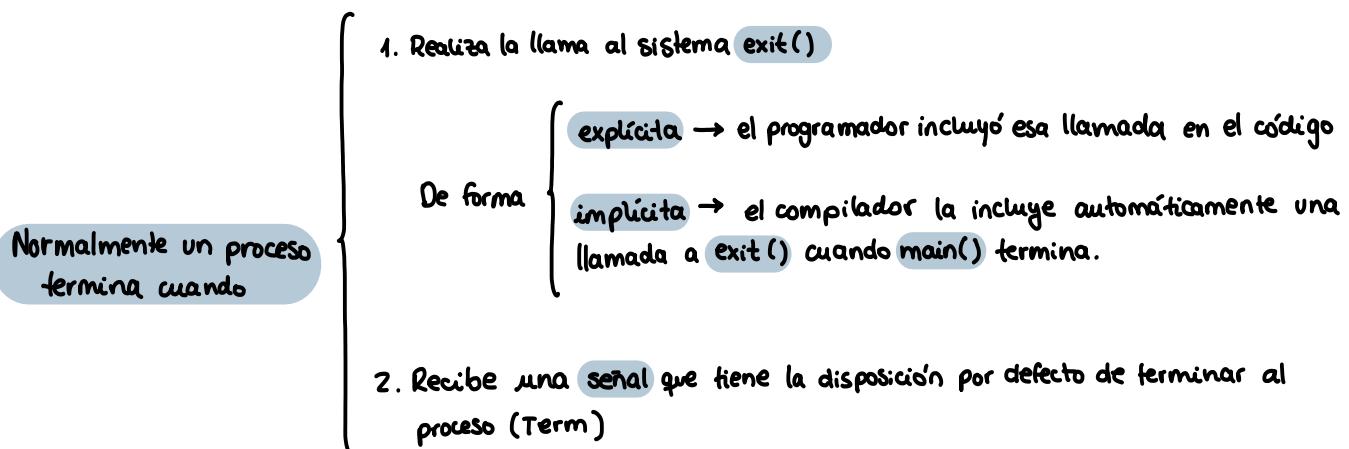
Creación de tareas (tasks): task_struct

fork() → clone() → do_fork() → copy_process()
↓
llama a

- Actuación de copy_process()
1. Crea la estructura thread_info (pila Kernel) y el task_struct para la nueva tarea con los valores de la tarea actual
 2. Asigna valores iniciales a los campos de la task_struct de la tarea hija que deben tener los valores distintos a los de la tarea padre
 3. Se establece el campo estado de la tarea hija TASK_UNINTERRUPTIBLE mientras se realizan las restantes acciones.
 4. Se establecen valores adecuados para los flags de la task_struct de la tarea hija:
 - flag PF_SUPERPRIV = 0 (tarea no usa privilegio de superusuario)
 - flag PF_FORKNOEXEC = 1 (proceso ha hecho fork pero no exec()).
 5. Se llama a alloc_pid() para asignar PID a la nueva tarea.
 6. Segun cuáles sean los flags pasados a clone(), duplica o comparte recursos como archivos abiertos, información de sistemas de archivos, manejadores de señales, espacio de direccionamiento del proceso,...
 7. Se establece el estado de la tarea hija a TASK_RUNNING.
 8. Finalmente copy_process() devuelve un puntero a la task_struct de la tarea hija.

Terminación de tareas (tasks): task_struct

Cuando un proceso termina, el Kernel libera todos sus recursos y notifica al padre su terminación.



El trabajo de liberación lo hace la función do_exit() definida en <linux/Kernel/exit.c>

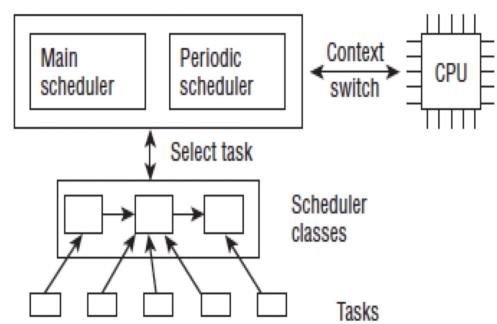
Activación de do_exit()

1. Activa el flag PF_EXITING de task_struct
 2. Para cada recurso que esté utilizando el proceso, se decrementa el contador correspondiente que indica el nº de procesos que lo están utilizando.
Si contador == 0 → operación de destrucción oportuna sobre el recurso
(e.g.: recurso = zona memoria ⇒ se liberaría el descriptor de memoria (mm_struct) correspondiente.)
 3. El valor que se pasa como argumento a exit() se almacena en el campo exit_code de task_struct (esta es info de terminación para que el padre pueda hacer un wait() o waitpid() y recogerla).
 4. Se manda una señal al padre (SIGCHLD) indicando la finalización de su hijo.
 5. Si el proceso aún tiene hijos, se establece como padre de dichos hijos al proceso init (PID=1).
- Nota: Dependiendo de las características del grupo de procesos al que pertenezca el proceso, podría ponerse como padre a otro miembro de ese grupo de procesos.
6. Se establece el campo state de task_struct a EXIT_ZOMBIE.
 7. Se llama a schedule() para que el planificador elija un nuevo proceso a ejecutar.

Nota: Puesto que este es el último código que ejecuta un proceso do_exit() nunca retorna.

PLANIFICACIÓN DE CPU EN LINUX

- Planificador modular: clases de planificación.
- Planificación de tiempo real.
- Planificación neutra o limpia (CFS: Completely Fair Scheduling).
- Planificación de la tarea IDLE (no hay trabajo que realizar).



Planificador modular: clases de planificación

Cada clase de planificación tiene un rango de prioridad que determina el número de colas de prioridad, una por número dentro del rango.

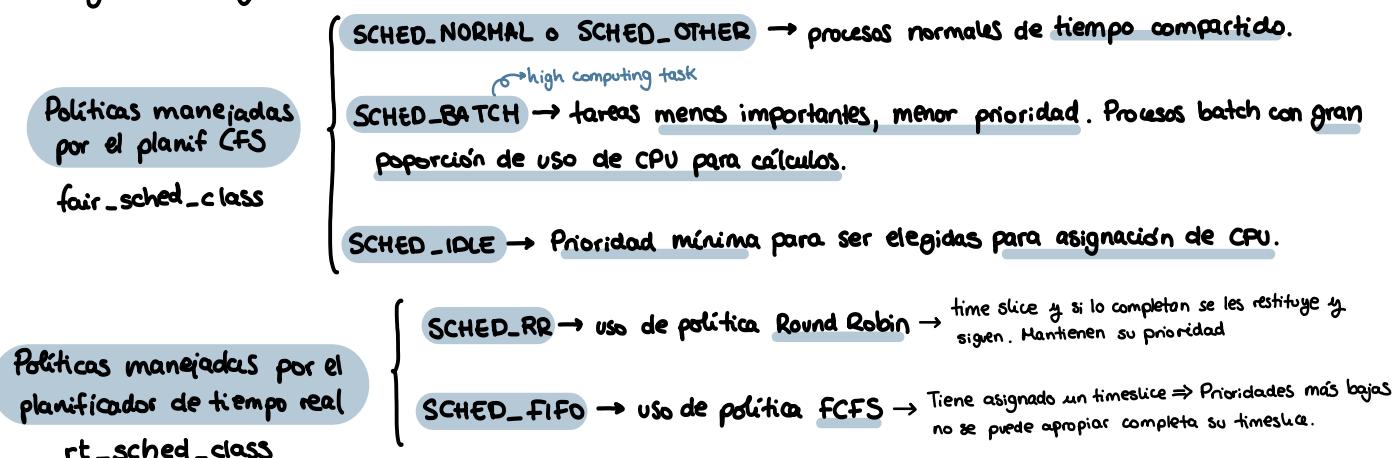
Se usa un algoritmo de planificación entre las clases de planificación por prioridades apropiativo.

Cada clase de planificación usa una o varias políticas para gestionar los procesos incluidos en cada una de sus colas.

La planificación no opera únicamente sobre el concepto de proceso, sino que maneja conceptos más amplios en el sentido de manejar grupos de procesos: Entidad de planificación (sched_entity).

Políticas de planificación

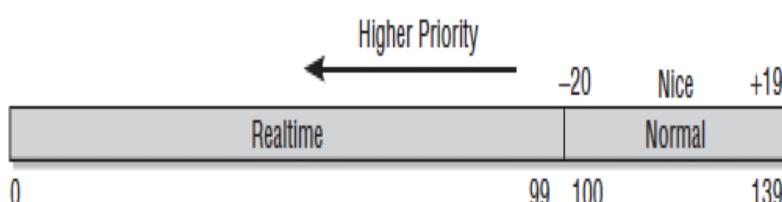
unsigned int policy; // política que se aplica al proceso



Prioridades

Siempre se cumple que el proceso que está en ejecución es el más prioritario

Intervalo de valores para prio → [0, 99] Prioridades para procesos de tiempo real
[100, 139] Prioridades para los procesos normales o regulares.



WUOLAH

El planificador periódico

Implementado en `scheduler_tick()`, función llamada automáticamente por el Kernel con frecuencia (Hz) constante cuyos valores están normalmente en el rango 1000 y 100 Hz.

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()` (contabiliza el % de CPU consumido)

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

El planificador principal: `schedule()`

Se implementa en la función `schedule()`, invocada en diversos puntos del Kernel para tomar decisiones sobre asignación de 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 justo antes de volver de una llamada al sistema, de una rutina de servicio de interrupción (RSI) o de una rutina de servicio de excepción (RSE); si está activo este flag se invoca a `schedule()`.

Actuación de `schedule()` → `context_switch()`

Determina la actual runqueue y establece el puntero `prev` a la `task_struct` del proceso actual
↳ puntero al PCB del proceso actualmente en CPU

Actualiza estadísticas y limpia el flag `TIF_NEED_RESCHED`.

Si el proceso actual va a un estado `TASK_IN_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; y se establece `next` con el puntero a la `task_struct` de dicho proceso seleccionado.
↳ `planif de CPU ← pick_next_task`

Si hay cambio en la asignación de CPU, se realiza el cambio de contexto llamando a `context_switch()` que salva y carga los registros correspondientes.

↳ `nuestro dispatcher()`

`schedule() == nuestro context_switch()`

↳ `planif. CPU == pick_next_task()`
`dispatcher == context_switch()`

ENCENDER TU LLAMA CUESTA MUY POCO



La clase de planificación CFS

Idea general → repartir el tiempo de CPU de forma imparcial, garantizando que todos los procesos se ejecutarán y, dependiendo del número de procesos en cola, asignarles más o menos tiempo de CPU.

Mantiene datos sobre los tiempos consumidos por los procesos.

El Kernel calcula un peso para cada proceso. Cuanto mayor sea el valor de la prioridad (prio) de un proceso, menor será el peso que tenga.

vruntime (virtual runtime) de una entidad es el tiempo virtual que un proceso ha consumido y se calcula a partir del tiempo real que el proceso ha hecho uso de la CPU, su prioridad $\frac{1}{\text{prio}}$ y su peso.

El valor vruntime del proceso actual se actualiza:

- Periódicamente (el planificador periódico ajusta los valores de tiempo de CPU consumido).
- Cuando llega un nuevo proceso ejecutable.
- Cuando el proceso actual se bloquea.

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

Para implementar esto CFS utiliza un red black tree (rbtree), que es una estructura de datos árbol binario que almacena nodos identificados por una clave, vruntime, y que permite una búsqueda eficiente por valor de clave.

Cuando un proceso va a entrar en estado bloqueado:

1. Se añade a una cola asociada con el motivo del bloqueo
2. Se establece el estado del proceso a TASK_INTERRUPTIBLE o a TASK_NONINTERRUPTIBLE según esté clasificado el motivo del bloqueo.
3. Se quita del rbtree de procesos ejecutables la referencia a la task_struct del proceso.
4. Se llama a schedule() para que se elija un nuevo proceso a ejecutar.

Cuando un proceso vuelve de estado bloqueado:

1. Se cambia su estado a ejecutable, TASK_RUNNING
2. Se elimina de la cola de bloqueo en que estaba
3. Se añade al rbtree de procesos ejecutables.

La clase de planificación de tiempo real

Se define la clase de planificación `rt_sched_class`.

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

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.

Las políticas de planificación de tiempo real `SCHED_RR` y `SCHED_FIFO` posibilitan que el Kernel de Linux pueda tener un comportamiento soft real-time (sistemas de tiempo real no estricto).

Al crear el proceso también se especifica la política bajo la cual se va a planificar y existe una llamada al sistema para cambiar la política asignada.

Planificación de CPU en SMP

Para realizar correctamente la planificación en un entorno SMP (multi procesador), el Kernel deberá tener en cuenta:

- 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).

Periodicamente una parte del Kernel deberá comprobar que se da un equilibrio entre las cargas de trabajo de las distintas CPUs y si detecta que una tiene más procesos que otra, reequilibra pasando procesos de una CPU a otra.