

## 2. DISEÑO E IMPLEMENTACION DE PROCESOS EN LINUX

En los puntos 2 y 3 de este tema nos basamos el kernel 2.6 de linux. Podemos descargar los fuentes de [www.kernel.org](http://www.kernel.org).

Nomenclatura usada para las rutas de archivos : aludiremos los archivos fuente expresando su ruta relativa desde el directorio donde hemos descargado todos los fuentes.

Nota sobre términos:

En el entorno Linux otro nombre para proceso es tarea (task);

Usaremos como equivalentes los términos proceso y tarea.

Internamente, el kernel de Linux usa la palabra tarea para referirse a un proceso.

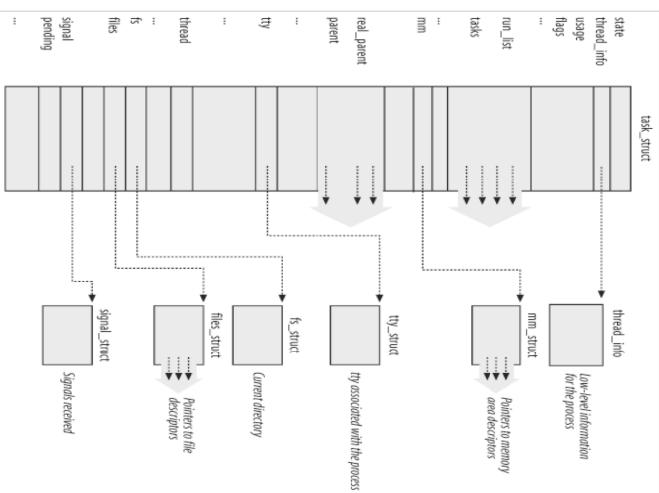
Cada proceso puede tener varias hebras (threads).

### 2.1 Representación de los procesos

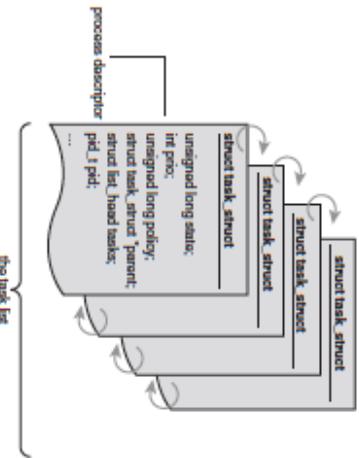
[Mau08.2.3]

En Linux, un proceso es representado por estas dos estructuras:

el PCB que es una estructura del tipo struct task\_struct  
y una estructura del tipo struct thread\_info



El kernel almacena la lista de procesos como una lista circular doblemente enlazada llamada **lista de tareas** (task list).



Cada elemento en la task list es un **descriptor de proceso** de tipo `struct task_struct` (definido en <include/linux/sched.h>):

```
struct task_struct { //> del kernel 2.6.24
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
    int lock_depth; /* BKL lock depth */
    #ifdef CONFIG_SMP
    #ifndef ARCH_WANT_UNLOCKED_CXSW
    int oncpu;
    #endif
    #endif
    int prio, static_prio, normal_prio;
    struct list_head run_list;
    const struct sched_class *sched_class;
    struct sched_entity se;
    #ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier */
    struct hlist_head preempt_notifiers;
    #endif
    unsigned short ioprio;
    /* fpu counter contains the number of consecutive context switches
     * that the FPU is used. If this is over a threshold, the lazy fpu
     * saving becomes unlazy to save the trap. This is an unsigned char
     * so that after 256 times the counter wraps and the behavior turns
     * lazy again. This to deal with bursty apps that only use FPU for
     * a short time
    unsigned char fpu_counter;
    s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
}
```

```

#endif CONFIG_BLK_DEV_IO_TRACE
unsigned int bttrace_seq;
#endif
/* pttrace_list/ptrace_children forms the list of my children
 * that were stolen by a ptracer.
 */
struct list_head ptrace_children;
struct list_head ptrace_list;
struct mm_struct *mm, *active_mm;
/* task state */
struct linux_binfmt *binfmt;
int exit_state;
int exit_code, exit_signal;
int pedath_signal; /* The signal sent when the parent dies */
/* ?? */
unsigned int personality;
unsigned did_exec;
pid_t pid;
pid_t tgid;

#ifndef CONFIG_CC_STACKPROTECTOR
/* canary value for the -fstack-protector gcc feature */
unsigned long stack_canary;
#endif
*/
/*
 * pointers to (original) parent process', youngest child, younger sibling,
 * older sibling respectively. (p->father can be replaced with
 * p->parent->pid)
 */
struct task_struct *real_parent; /* real parent process (when being debugged) */
struct task_struct *parent; /* parent process */
/*
 * children_sibling forms the list of my children plus the
 * tasks I'm ptracing.
 */
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children list */
struct task_struct *group_leader; /* threadgroup leader */
/*
 * PID/PID hash table linkage.
 */
struct pid_link pids[PIDTYPE_MAX];
struct list_head thread_group;
struct completion *fork_done; /* for vfork() */
int __user *set_child_tid; /* CLONE_CHILD_SETTID */
int __user *clear_child_tid; /* CLONE_CHILD_CLEARTID */
unsigned int rt_priority;
cpu_time_t utime, stime, utimescaled, stimescaled;
cpu_time_t prev_utime, prev_stime;
unsigned long nvcsw, nvcsv; /* context switch counts */
struct timespec start_time; /* monotonic time */
struct timespec real_start_time; /* boot based time */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-
specific */
unsigned long min_flt, maj_flt;
unsigned long prof_expires, virt_expires;
unsigned long long long_it_sched_expires;
struct list_head cpu_timers[3];

```

```

/* process credentials */
uid_t uid,euid,suid,fsuid;
gid_t gid,egid,sgid,fsgid;
struct group_info *group_info;
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
unsigned int keep_capabilities; /* keep capabilities */

struct user_struct *user;

#endif /* CONFIG_KEYS */

/* file system info */
int link_count, total_link_count;
#endif /* CONFIG_SYSVIPC */
/* ipc stuff */

struct sysv_sem sysvsem;
#endif /* CPU-SPECIFIC STATE OF THIS TASK */
struct thread_struct thread;
/* filesystem information */
struct fs_struct *fs;
/* open file information */
struct files_struct *files;
/* namespaces */
struct nsproxy *nsproxy;
/* signal handlers */
struct signal_struct *signal;
struct sighand_struct *sighand;

sigset_t blocked, real_blocked;
#endif /* CONFIG_SECURITY */

#ifndef CONFIG_AUDIT
struct audit_context *audit_context;
seccomp_t seccomp;
#endif /* CONFIG_AUDIT */

/* Thread group tracking */
u32 parent_exec_id;
u32 self_exec_id;
/* protection of (de-)allocation: mm, files, fs, tty, keyrings */
spinlock_t alloc_lock;
/* Protection of the PI data structures: */
spinlock_t pi_lock;

#ifndef CONFIG_RT_MUTEXES
/* PT waiters blocked on a rt_mutex held by this task */
struct plist_head pi_waiters;
/* Deadlock detection and priority inheritance handling */
struct rt_mutex_waiter *pi_blocked_on;
unsigned int irq_events;
#endif /* CONFIG_RT_MUTEXES */
/* mutex deadlock detection */
struct mutex_waiter *blocked_on;
#endif /* CONFIG_TRACE_IRQFLAGS */
#endif /* CONFIG_RT_MUTEXES */

```

```

int hardirqs_enabled;
unsigned long hardirq_enable_ip;
unsigned int hardirq_enable_event;
unsigned long hardirq_disable_ip;
unsigned int hardirq_disable_event;
int softirqs_enabled;
unsigned long softirq_disable_ip;
unsigned long softirq_disable_event;
unsigned long softirq_enable_ip;
unsigned int softirq_enable_event;
int hardirq_context;
int softirq_context;

#endif /* CONFIG_LOCKDEP */

#ifndef CONFIG_LOCKDEP
#define MAX_LOCK_DEPTH 30UL
u64 curr_chain_key;
int lockdep_depth;
struct held_locks[MAX_LOCK_DEPTH];
unsigned int lockdep_recursion;
#endif

```

```

/* journaling filesystem info */
void *jjournal_info;

/* stacked block device info */
struct bio *bio_list, **bio_tail;

/* VM state */
struct reclaim_state *reclaim_state;
struct backing_dev_info *backing_dev_info;
struct io_context *io_context;
unsigned long ptrace_message;
siginfo_t *last_siginfo; /* For ptrace use. */
...}

```

## ALGUNOS CONTENIDOS DE task\_struct:

- \* estado e información de ejecución tales como señales pendientes, Pid, puntero al padre y a otros procesos relacionados (se verá más adelante), prioridades, información sobre el tiempo de CPU.
- \* Información sobre asignación de memoria
- \* Credenciales del proceso como identificativos de usuario y de grupo
- \* Archivos usados
- \* Información sobre comunicación entre procesos
- \* Manejadores de señales usados por el proceso para responder a las señales

Dentro del kernel las tareas son referenciadas mediante un **puntero a su task\_struct**.

La macro **current** proporciona un puntero al descriptor de proceso de la tarea que se está ejecutando actualmente.

## 2.2 Estados de un proceso [Mau08 2.3]

La variable `state` de `task_struct` especifica el estado actual del proceso.

Valores: (son constantes que resuelve el preprocesador definidas en <sched.h>)

**TASK\_RUNNING:** proceso ejecutable, tanto si está actualmente ejecutándose o está esperando a que se le asigne CPU.

Si el proceso se está ejecutando puede estar tanto en ejecución en el espacio de usuario como en el espacio del kernel.

**TASK\_INTERRUPTIBLE:** proceso bloqueado o durmiendo; la tarea no está lista para ejecutarse porque espera un evento;

cuando el kernel notifique al proceso (mediante una señal) que el evento ha ocurrido se calificará en el estado `TASK_RUNNING` y podrá reanudar su ejecución cuando el planificador le asigne la CPU

**TASK\_UNINTERRUPTIBLE:** estado idéntico al anterior excepto que el proceso no despierta si recibe una señal;

11

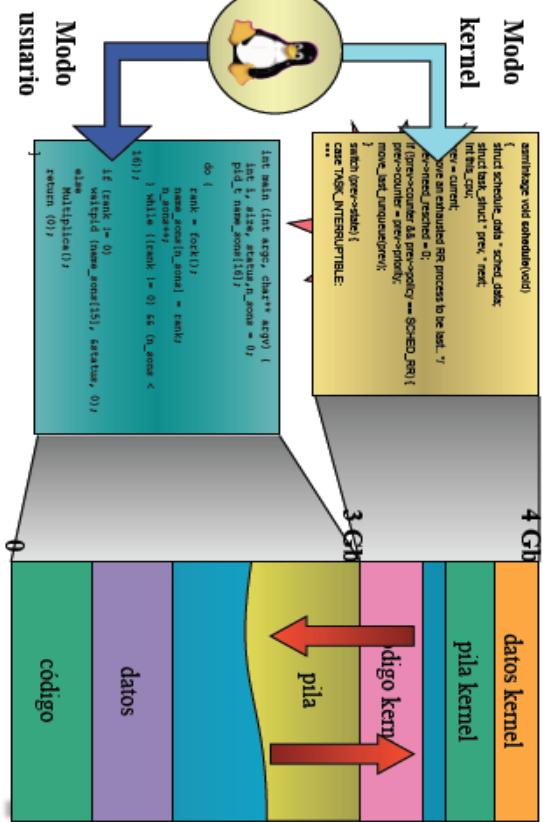
**TASK\_STOPPED:** proceso parado o detenido, no se está ejecutando ni es elegible para ejecutarse. Ocurre cuando la tarea recibe señales como SIGSTOP o ciertas señales de depuración.

**TASK\_TRACED:** el proceso está siendo traceado por otro proceso.

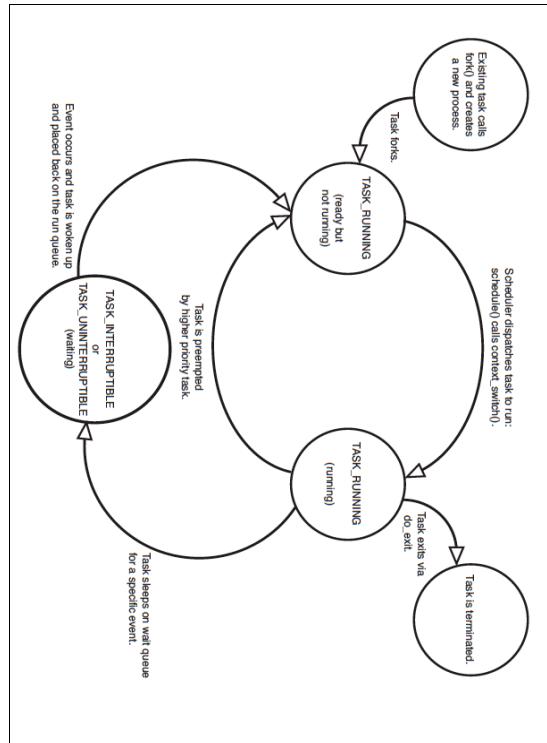
**EXIT\_ZOMBIE:** una tarea está en estado zombie cuando ha terminado pero todavía el padre no ha tomado su valor de retorno, éste debe permanecer almacenado hasta que el padre lo recoja con un wait, por tanto no se puede destruir todavía su task\_struct.

**EXIT\_DEAD:** es el estado al que pasa un proceso tras la llamada al sistema wait, hasta que sea completamente eliminado del sistema.

Los valores EXIT\_ZOMBIE y EXIT\_DEAD pueden ser usados tanto en el campo state de task\_struct como en el campo exit\_state, que es específico de aquellos procesos que están terminando



## 2.3 Estructura interna de un proceso en linux



**Pla:** zona del espacio de direcciones lógicas de un proceso que se utiliza para gestionar las llamadas a función que se efectúa en el código del programa.

Está formada por un conjunto de capas con estructura LIFO.

Cada vez que se realiza una llamada a una función **se crea un marco nuevo en la pila que contiene...**

las variables locales de la función,

sus parámetros actuales,

una dirección de retorno,

y la dirección del marco anterior para poder eliminar éste.

Cada vez que una función retorna **se elimina el marco actual**.

Como el proceso puede estar trabajando en dos modos (usuario o kernel), **habrá una pila para cada modo**.

## 2.4 Contexto de un proceso [Lov10 pag. 29]

\* La ejecución usual de un proceso ocurre en el espacio de usuario (**user-space**);

--- Si se produce una llamada al sistema, o se genera una excepción o interrupción,  
.....entra en el espacio del kernel (**kernel-space**)

....y se dice entonces que el kernel “**se está ejecutando en el contexto del proceso**”, o que está en el contexto del proceso.

--- Cuando se termina dicha llamada o tratamiento de excepción o interrupción el proceso reanuda su ejecución en su espacio de usuario (a no ser que un proceso de más alta prioridad se haya convertido en ejecutable en ese espacio de tiempo en cuyo caso el planificador lo elegirá para asignarle CPU).

\* En resumen, se pasa a ejecutar código del kernel cuando ocurre alguno de estos acontecimientos:

una interrupción o excepción,

se hace una llamada al sistema,

o se cambia la asignación de la CPU de un proceso a otro.

Observemos que los acontecimientos anteriores pueden presentarse de forma anidada, por tanto el kernel deberá salvar y restaurar la información necesaria para que esto se resuelva adecuadamente.

Se consigue así que haya simultáneamente más de una ejecución de código kernel, consiguiéndose la interesante característica de kernel reentrant.

Re-entrant kernel: Several processes may be in Kernel Mode at the same time

A re-entrant kernel is able to suspend the current running process even if it is in the Kernel Mode

## 2.5 El árbol de procesos [Lov10 pag 29-30]

\* En todos los entornos Unix hay una jerarquía de procesos definida como sigue:

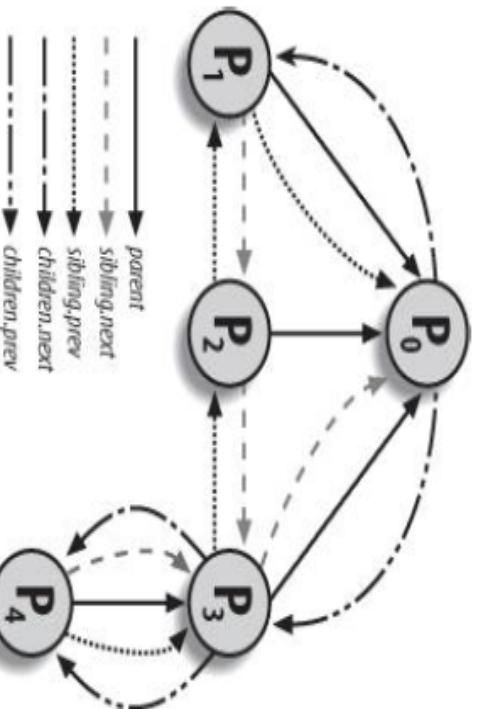
Si el proceso P0 hace una llamada la sistema fork y así genera el proceso P1, se dice que P0 es el proceso **padre** y P1 es el hijo.

Si el proceso P0 hace varios fork general de varios proceso hijos P1,P2,P3, la relación entre ellos es de **hermanos** (sibling)

\* Todos los procesos son descendientes del proceso **init** (cuyo pid es 1);

El kernel comienza init en el último paso del proceso de arranque del sistema.

El proceso init lanza a los demás procesos completando el proceso de arranque.



Ver Mauerer figura 1-11 (pag 21) y 2-6 (pag 63)

19

\* Cada task\_struct tiene un puntero ...

a la task\_struct de su padre:

```
struct task_struct *parent
```

a una lista de hijos (llamada children):

```
struct list_head children;
/* apunta a la cabeza a lista de mis hijos*/
```

y a una lista de sus hermanos (llamada sibling):

```
struct list_head sibling
... a la lista de hijos de mi padre */
```

El kernel dispone de procedimientos eficaces para las acciones usuales de manipulación de la lista. (Para una explicación detallada de list\_head ver [Mau08 1.3.13])

\* Dado el proceso actual, es posible obtener el descriptor de proceso de su padre mediante ...este código:  
`struct task_struct *my_parent = current->parent;`

- \* El descriptor de proceso del proceso init está almacenado estáticamente como **init\_task**.

```
* El código siguiente pone de manifiesto la relación existente entre cualquier proceso y  
el init:  
  
struct task_struct *task;  
  
for (task = current; task != &init_task; task = task->parent);  
  
/* task now points to init */
```

- \* Recorrer todos los procesos en el sistema es fácil porque la lista de tareas es una lista circular doblemente enlazada

```
* Se proporciona la macro for_each_process(task) que recorre la lista completa de tareas y  
en cada iteración task apunta a la siguiente tarea en la lista:  
  
struct task_struct *task;  
  
for_each_process(task) {  
  
    /* this pointlessly prints the name and PID of each task */  
  
    printk(KERN_INFO "%d\n", task->comm, task->pid);  
}
```

2.1

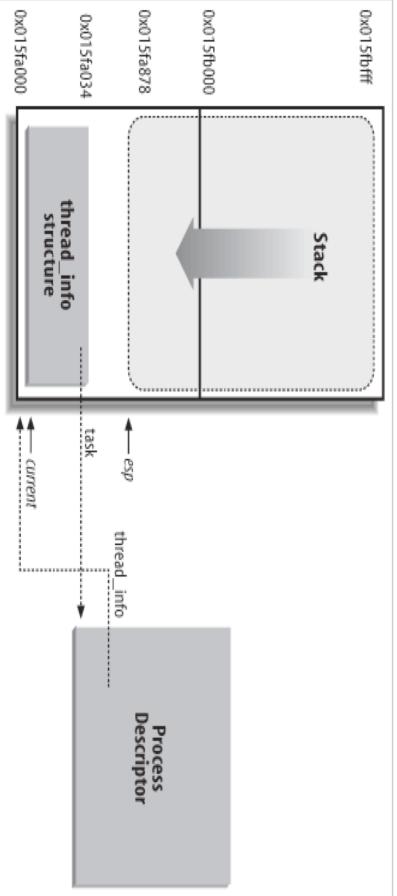
## 2.6 La estructura **thread\_info**

- \* Contiene la información de bajo nivel del proceso, acedida por código ensamblador dependiente de la arquitectura.

- \* Forma parte de una estructura del tipo union **thread\_union** (include/linux/sched.h) donde está también la pila kernel de dicho proceso:

```
union thread_union {  
    struct thread_info thread_info;  
    unsigned long stack[THREAD_SIZE/sizeof(long)];  
};
```

La figura siguiente muestra la relación entre **task\_struct**, **thread\_info** y la pila kernel:



(Ver "Understanding the Linux Kernel" Bovet, D.; 3.2.2.1.)

\* Cuando algún componente del kernel utilizará demasiado espacio de pila, la pila kernel podría pisar la información de `thread_info`; para evitar eso el kernel tiene una función para determinar si una determinada dirección está dentro de una porción válida de la pila o no.

23

## 2.7 Implementación de hilos en linux [Lov10 pag33]

### Creación de hebras

\* Desde el punto de vista del kernel no hay distinción entre hebra y proceso;

**Linux implementa el concepto de hebra como un proceso sin más, que simplemente comparte recursos con otros procesos.**

\* Cada hebra tiene su propia `task_struct`.

\* La llamada al sistema `clone` crea un nuevo proceso o hebra;

\* La figura siguiente muestra los flags de control que podemos pasar a `clone`;

podemos especificar cómo deseamos que se comporte el nuevo proceso y cómo se quiere que sea la compartición de recursos entre padre e hijo:

Flag	Meaning
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share filesystem information.
CLONE_IDLE TASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Child is to have same parent as its parent.
CLONE_PTRACE	Continue tracing child.
CLONE_SETTID	Write the TID back to user-space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals.
CLONE_SYSVSEM	Parent and child share System V SEM_UNDO semantics.
CLONE_THREAD	Parent and child are in the same thread group.
CLONE_VFORK	vfork() was used and the parent will sleep until the child wakes it.
CLONE_UNTRACED	Do not let the tracing process force CLONE_PTRACE on the child.
CLONE_STOP	Start process in the TASK_STOPPED state.
CLONE_SETTLS	Create a new TLS (thread-local storage) for the child.

### Hebras kernel [Llo10 pag 35]

\* A veces es útil que el kernel realiza algunas operaciones en segundo plano, para lo cual se crean hebras kernel (procesos que existen únicamente en el espacio del kernel).

\* La principal diferencia entre hebras kernel y procesos normales es que las hebras kernel no tienen un espacio de direcciones (su puntero mm es NULL), se ejecutan únicamente en el espacio del kernel.

\* Por lo demás son planificadas y podrían ser expropiadas, como procesos normales.

\* Se crean por el kernel al levantar el sistema, mediante una llamada a clone().

\* Y como todo proceso, termina cuando realizan una operación do\_exit o cuando otra parte del kernel provoca su finalización

## 2.8 Ejecutando llamadas al sistema para gestión de procesos

- \* Nos centramos en las llamadas al sistema para gestión de procesos como fork, vfork, y clone.

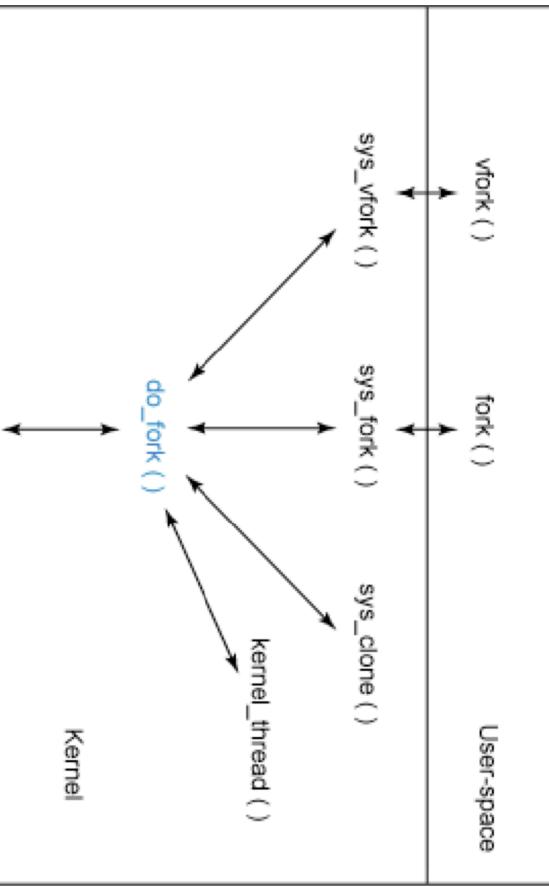
\* Normalmente estas llamadas se invocan a través de las librerías de C que realizan la comunicación con el kernel (los métodos para cambiar de modo usuario a modo kernel varían de una arquitectura a otra).

\* El punto de entrada para fork, vfork y clone son las funciones sys\_fork, sys\_vfork y sys\_clone;

\* su implementación es dependiente de la arquitectura puesto que la forma en que los parámetros se pasan entre el espacio de usuario y el espacio del kernel son diferentes en las distintas arquitecturas.

La labor de las anteriores funciones es extraer la información suministrada en el espacio de usuario (parámetros de la llamada) e invocar a la función do\_fork (independiente de la arquitectura) que es quien realiza la duplicación de procesos.

27



## 2.9 Creación de procesos con fork [Llo10 pag 32]

\* Situémonos en el espacio de usuario: se produce una llamada a alguna de las rutinas de librería para crear un nuevo proceso como fork(), vfork() o clone();

Dado que todas ellas, básicamente, realizan la misma función que es crear un nuevo proceso (aunque varían en las características de éste), a grandes rasgos ocurre la misma secuencia de llamadas:

```
.... se transfiere el control a la función do_fork() del kernel (definida en
<kernel/fork.c>)
    ... que a su vez llama a la función copy_process(), que realiza en sí la creación
    del nuevo proceso
```

tras el fin de copy\_process, do\_fork provocará que el nuevo hijo se ejecute.

### Actuación de copy\_process()

1. Se **crea una nueva pila kernel**, la estructura **thread\_info** y la **task\_struct** para el nuevo proceso con los valores de la tarea actual.
2. Para los elementos de task\_struct del hijo que deban tener valores distintos a los del padre, se les dan los valores iniciales correctos (como por ejemplo datos para estadísticas).
3. Se establece el estado del hijo a **TASK\_UNINTERRUPTIBLE** mientras se realizan las restantes acciones.
4. Se establecen valores adecuados para los flags de la task\_struct del hijo:
  - pone a 0 el flag **PF\_SUPERPRIV** (indica si la tarea usa privilegio de superusuario)
  - pone a 1 el flag **PF\_FORKNOEXEC** (que indica si el proceso ha hecho fork pero no exec)

5. Se llama a alloc\_pid() para asignar un **PID** a la nueva tarea.

6. Según cuáles sean los flags pasados a clone(), copy\_process() **duplica o comparte recursos** como archivos abiertos, información de sistemas de archivos, manejadores de señales, espacio de direccionamiento del proceso....  
Normalmente estos recursos son compartidos entre tareas de un mismo proceso (contrario al caso de que se creen nuevos recursos para el hijo con los valores iniciales que tenía del padre).

7. Finalmente copy\_process() termina devolviendo un puntero a la task\_struct del hijo.

## **Copy-on-Write**

[Loy10 pag 31]

\*Con esta técnica, en la creación de un nuevo proceso, al hijo no se le asigna nuevo espacio de memoria sino que las páginas del padre resultan ahora compartidas por ambos, padre e hijo.

\* A cada páginas se le asigna un bit llamado copy-on-write (**cow**) con valor 1;

\* **cow=1** significa página compartida por varios procesos:

- las operaciones de lectura están permitidas
- cuando un proceso intente escribir se producirá una excepción por “violación de protección”; el sistema operativo, al resolver esta excepción, hace una copia de la página para el proceso que generó la excepción en un nuevo marco de página.

- \* Si finalmente la página va a ser escrita, con esta técnica se poserga lo más posible la asignación de nuevos marcos de página para que cada proceso tenga una página privada.

\* Si finalmente nadie escribe en ella, hemos ahorrado un marco de página.

- \* Conclusión: "Copy-on-write" es una técnica que **evita el consumo excesivo de recursos que supondría que fork duplicara físicamente todos los recursos del padre al hijo**, particularmente en el caso, bastante frecuente, de que el proceso hijo haga pronto un exec.

## Terminación de procesos [Lov10 pag36]

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

\* Normalmente un proceso termina cuando .....  
realiza la **llamada al sistema exit()**;

esto puede ser **explícito** si el programador incluyó esa llamada en el código del programa,

o **implícito**, pues el compilador incluye automáticamente una llamada a exit() cuando main() termina.

2) **recibe una señal** ante la que tiene la acción establecida de terminar

- \* Independientemente de qué acontecimiento ha provocado el fin del proceso, el grueso del trabajo lo hace la función **do\_exit()** definida en <linux/kernel/exit.c>

## Actuación de do\_exit()

1. Establece el flag `PF_EXITING` de task\_struct

2. Para cada recurso que esté utilizando el proceso, por ejemplo espacio de direcciones o archivos, se decrementa el contador correspondiente que indica el nº de procesos que lo están utilizando;

si este contador vale 0 se realiza la operación de destrucción oportuna sobre el recurso, por ejemplo si fuera una zona de memoria, se liberaría.

Así por ejemplo, se libera el campo `mm_struct` de este proceso; si no hay otros procesos que estén usando este espacio de direcciones el kernel lo destruya.

3. El valor que se pasa como argumento a `exit()` se almacena en el campo `exit_code` de `task_struct`. Hay que almacenar lo por si el padre quisiera obtenerlo para tener información sobre cómo ha terminado el hijo.

4. Se manda una señal al padre indicando la finalización de su hijo.

35

5. Si aún tiene hijos, se pone como padre de éstos al proceso init

(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 `exit_state` de `task_struct` to `EXIT_ZOMBIE`

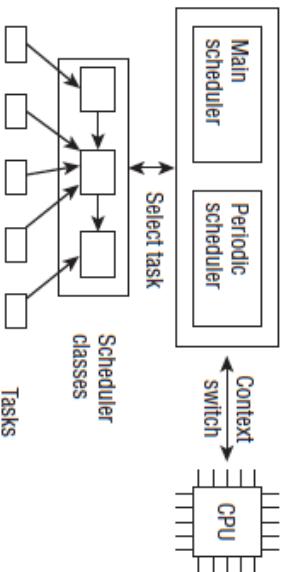
7. Se llama a `schedule()` para que el planificador elija un nuevo proceso a ejecutar

Puesto que este es el último código que ejecuta un proceso, `do_exit` nunca retorna.

## 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**.



37

\* El kernel dispone de diferentes clases de planificación, cada una de las cuales tendrá asociado un conjunto de procesos: (de mayor a menor prioridad)

**planificación de tiempo real**

**planificación neutra o limpia (CFS: Completely Fair Scheduling)**

**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 en 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)

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

Se activa cuando el proceso actual no deseé 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.

se calculan a partir de static\_prio y de la política de planificación del proceso (no detallaremos más)

```
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**...

**SCED\_NORMAL**: se aplica a los procesos normales en los que nos centramos (frente a los de tiempo real)

**SCED\_BACH**: tareas menos importantes, concretamente procesos batch 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.

**SCED\_IDLE**: tareas de este tipo tienen un peso mínimo para ser elegidas para asignación de CPU

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

**SCED\_RR** y **SCED\_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_slice` 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 rededicarse a qué proceso darle la CPU.

\* Describímos en más detalle la estructura thread\_info, estrechamente unida a la task\_struct 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_prempt_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 clase 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.

47

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_prempt_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.h>) 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 número 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 empotadas, 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-black 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.
```

53

## 3.3 Sobre prioridades

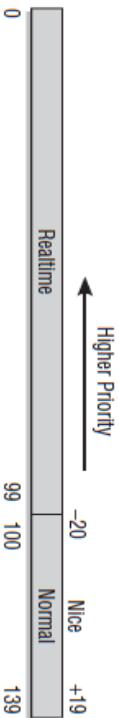
[Mau08 2.5.3]

\* 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.



54

## 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

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

57

## 3.6 Cambio de contexto

[Ley10 pág. 62]

Función `context_switch` (<include/linux/sched.h>): 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:

Llama a `switch_mm` (declarada en <asm/mmu\_context.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.

58

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

### a) Conceptos básicos de CFS

\* Objetivo básico de CFS o "Completely Fair Scheduler": cada proceso debe recibir  $1/n$  del tiempo de CPU (siendo  $n$  el número 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

Se llama latencia al menor periodo de tiempo en que se asegura que todos los procesos han sido elegidos para ejecutarse.

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 (en Linux 2.6 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 tendería a infinito  
(siendo estrictos, habría algún momento de tiempo en que no se proporcionaría reparto equitativo).

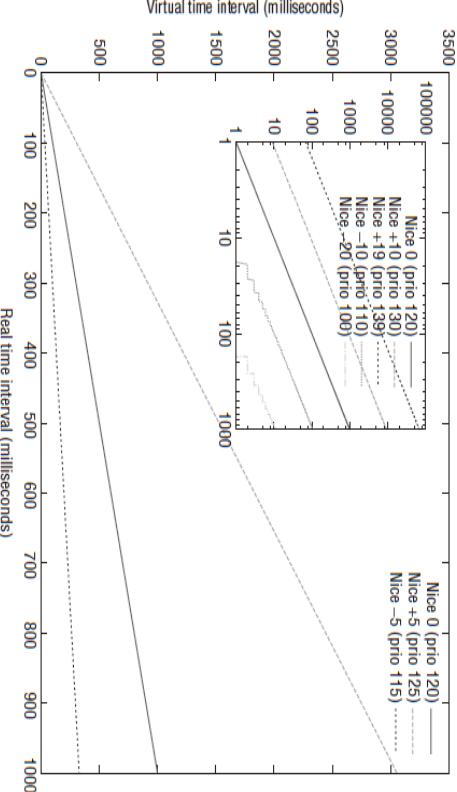
\* Si contemplamos ahora procesos con distintas prioridades, CFS repartiría el tiempo de cpu de modo que todos los procesos de la misma prioridad tengan igual proporción de uso de CPU.

## 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 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,  
y de su prioridad estática



\* 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 black 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

[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.

65

- \* 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 aquellos 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) [Lav10 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 desea 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 cuándo 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.

67

- \* 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 encuentra 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 pasa el control ahora a otra tarea. Así es como 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 decremente 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.

En el futuro, cuando la tarea actual libere todos los locks y el contador retorna 0, se verificará el flag `TIF_NEED_RESCHED` y, si está activo, el planificador será invocado.

### La apropiación en modo privilegiado también se puede realizar de forma explícita, llamando explícitamente a la función `schedule`.

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 expropriativo.
- Si una tarea en el kernel invoca explícitamente a la función `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 acomodarse 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)

\* Períódicamente 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.

73

## 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

74