

Sistemas Operativos
2º Curso
Doble Grado en Ingeniería Informática y Matemáticas

Tema 2:

Procesos e hilos



José Antonio Gómez Hernández, 2020.

Contenidos

- ▷ Implementación de las abstracciones proceso/hilo:
 - Descriptor de proceso/hilo
- ▷ Diagrama de estados y transiciones
 - Estados y sus propiedades
 - Operaciones sobre procesos
- ▷ Planificación de la CPU:
 - Tipos de planificadores y algoritmos de planificación básicos
 - El planificador de Linux
 - Ahorro de energía



Contenidos: bibliografía

▷ Teoría general de SO's:

- W. Stallings, *Sistemas Operativos. Aspectos internos y principio de diseño*, 5ª Ed., Prentice Hall, 2005.
- W. Stallings, *Operating Systems: Internals and Design Principles*, Pearson, 2018.

▷ Sistema Linux:

- W. Mauerer, [*Linux Professional Kernel Architecture*](#), Wiley, 2008. D.P. Bovet, M. Cesati, [*Understanding the Linux Kernel*](#), O'Reilly, 2006.
- R. Love, *Linux Kernel Development*, 3rd, Novell Press, 2010.
- R. Bharadwaj, [*Mastering Linux Kernel Development: a kernel developer's reference*](#), Pack, 2017.
- Z. Jiong, *A Heavily Commented Linux Kernel Source Code*, China Machine Press, 2019.



Código fuente de Linux: navegación

- ▷ Navegación interactiva por el código fuente:
 - ▷ Navegación por el código fuente:
<https://elixir.bootlin.com/linux/latest/source> para cualquier versión del kernel.
 - ▷ Mapa interactivo: http://makelinux.net/kernel_map/



0.

Repaso

Conceptos sobre procesos e hilos

Conceptos a repasar

- ▷ Qué es un proceso y un hilo
- ▷ Qué es y qué contiene la imagen de un proceso
- ▷ Qué es el Bloque de Control de Proceso (PCB)
- ▷ Estados básicos de procesos/hilos
- ▷ Transiciones posibles entre estados y eventos que las disparan
- ▷ Qué es y cómo se realiza en cambio de contexto (proceso).



Dónde podemos revisarlo

W. Stallings, *Sistemas Operativos. Aspectos internos y principio de diseño*, 5ª Ed., Prentice Hall, 2005.

Capítulo 3: 3.1 a 3.4

Capítulo 4: 4.1 y 4.2

Disponible en tanto en Biblioteca como en línea:

<https://elibro.net/es/ereader/ugr/45337>

Nota: Los Apartados hacen referencia a la Edición 5ª (la única traducida al español), pero sería recomendable acceder a ediciones posteriores.





1.

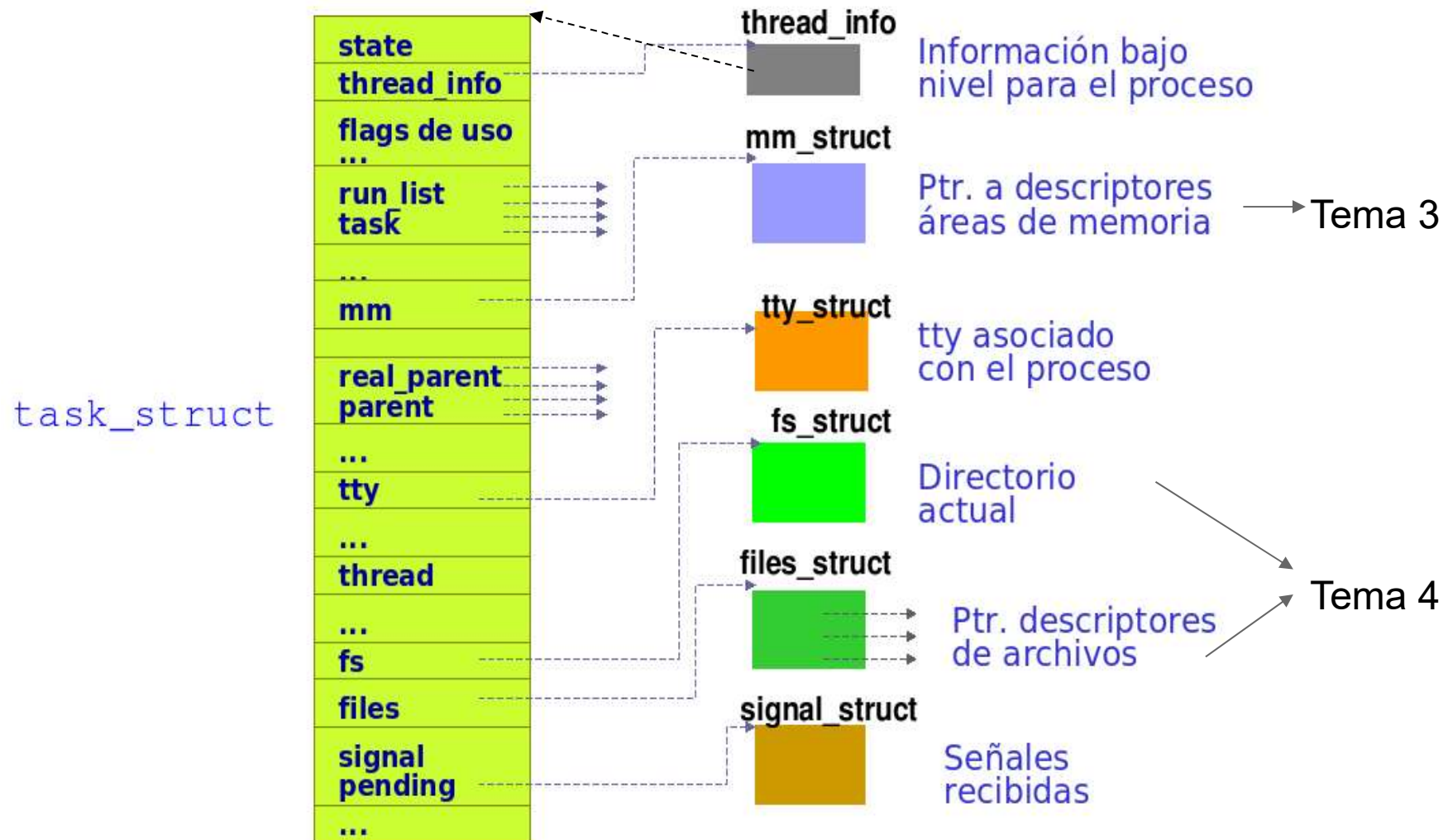
Implementación de las abstracciones proceso e hilo

Cómo se construye un proceso/hilo en Linux

El Bloque de Control de Proceso

- ▷ En Linux, el PCB (Bloque de control de proceso) se denomina **descriptor de proceso**.
- ▷ Viene definido por la estructura `task_struct`, que podemos encontrar en la cabecera de las fuentes *include/linux/sched.h*:
(<https://elixir.bootlin.com/linux/v4.7/source/include/linux/sched.h#L1458>)

Descriptor de proceso (Linux)



task_struct: definición

```
struct task_struct {
volatile long state; /* Estado del proceso: -1 unrunnable, 0 runnable, >0 stopped */
Void *stack;        /* pila kernel */
unsigned long flags; /* Indicadores estado: starting, exiting, signaling, ... */
int sigpending;      /* Señales pendientes */
. . .
volatile long need_resched; /* Bandera de re-planificación/
. . .
Int prio, ...          /* Prioridad, ...*/
long nice;             /* Valor nice */
unsigned long policy; /* Política de planificación */
unsigned int time_slice; /* tiempo de CPU asignado */
. . .
struct mm_struct *mm; /* ED gestión de memoria a nivel de usuario */
int has_cpu, processor; /* Tiene CPU asignada */
struct list_head run_list; /* Lista de ejecutables */
unsigned long sleep_time; /* Tiempo que lleva bloqueado */
. . .
struct task_struct *next_task, *prev_task; /* Ptrs. enlazar en lista de tareas */
struct mm_struct *active_mm; /* Espacio de direcciones activo */
. . .
int exit_code, exit_signal; /* Código y señal de finalización */
. . .
pid_t pid;              /* PID del proceso */
pid_t pgrp;             /* GID del proceso */
. . .
pid_t tgid;            /* TID del hilo */
. . .
```

Código (cont.)

```
/* Punteros para mantener la jerarquía de procesos */
struct task_struct *p_opptr, *p_pptr, *p_cptra, *p_ysptr, *p_osptra;
struct list_head thread_group;
. . .
uid_t uid, euid, suid, fsuid;      /* Credenciales proceso: ID de usuario: real, efectivo, ...*/
gid_t gid, egid, sgid, fsgid;     /* Credenciales proceso: ID de grupo: real, efectivo, ...*/
. . .
kernel_cap_t cap_effective, cap_inheritable, cap_permitted; /* Capacidades */
. . .
struct rlimit rlim[RLIM_NLIMITS]; /* Límite de recursos asignables*/
. . .
int link_count;
struct tty_struct *tty; /* tty asociada; NULL if no tty */
unsigned int locks; /* How many file locks are being held */
. . .
struct thread_struct thread; /* Estado específico de la CPU de esta tarea */
struct fs_struct *fs;      /* Información del sistema de archivos */
struct files_struct *files; /* Tabla de archivos abiertos */
spinlock_t sigmask_lock;    /* Protege y bloquea señales*/
struct signal_struct *sig; /* Descriptor de señales */
. . .
sigset_t blocked;           /* Señales bloqueadas */
struct sigpending pending;   /* Señales pendientes */
. . .
spinlock_t alloc_lock;      /* Protección de (des)asignación: mm, files, fs, tty */
};
```

Comentarios

▷ Las sub-estructuras¹ `mm_struct`, `files_struct`, `fs_struct`, `tty_struct`, `signal_struct` se desgajan de la principal por varios motivos:

- No se asignan cuando no es necesario. Por ejemplo, una hebra kernel o 'demonio' no tiene asignado terminal por lo que `tty_struct→NULL`, tampoco tiene definido espacio de usuario (solo código y datos kernel) por lo que `mm_struct→NULL`.
- Permiten su compartición cuando sea necesario (volveremos sobre ellos al hablar de hilos y `clone()`).

¹ La estructura `mm_struct` se verá en el tema de memoria, `files_struct` y `fs_struct` en archivos.

Estructura `thread_info`

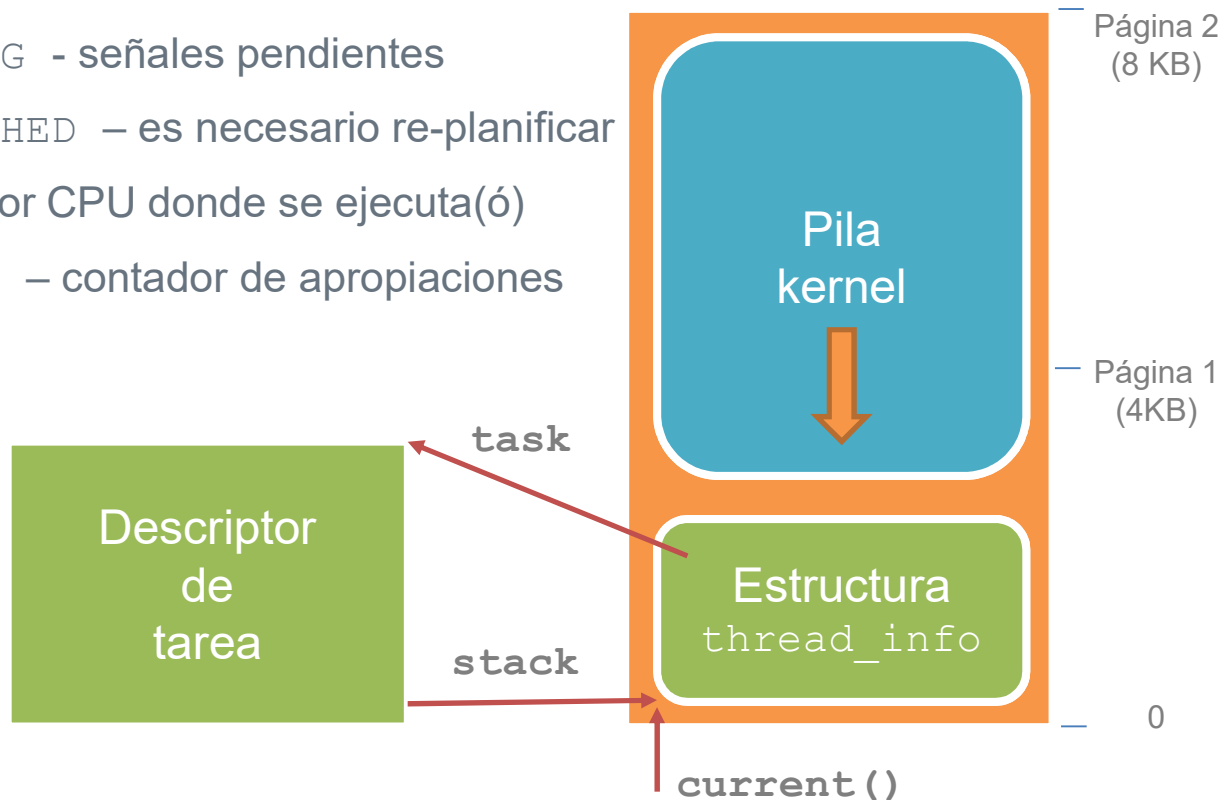
▷ Contiene información de ejecución de bajo nivel sobre el proceso/hilo y permite acceder a la *pila kernel* del mismo.

▷ Algunos campos:

- `TIF_SIGPENDING` - señales pendientes
- `TIF_NEED_RESCHED` - es necesario re-planificar
- `cpu` - identificador CPU donde se ejecuta(ó)
- `preempt_count` - contador de apropiaciones

▷ Macros:

- `current()`





2.

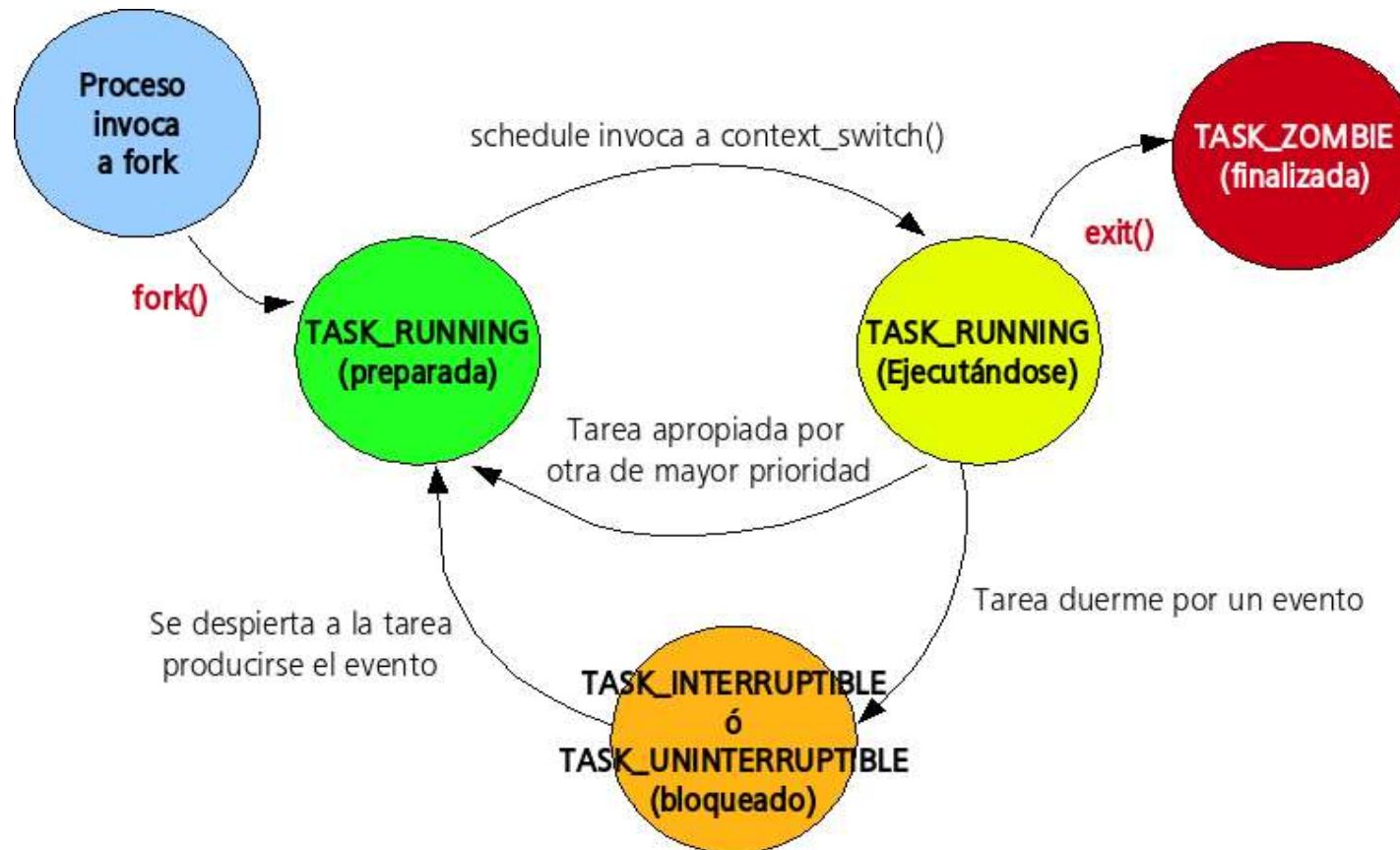
Diagrama de estados y transiciones

Cuales son los estados de los procesos en Linux y cómo transicionan entre los mismos

Estados de los procesos

- ▷ El campo `state` almacena el estado:
 - `TASK_RUNNING`: El proceso es ejecutable o esta en ejecución.
 - `TASK_INTERRUPTIBLE`: el proceso esta bloqueado (dormido) de forma que puede ser interrumpido por una señal.
 - `TASK_UNINTERRUPTIBLE`: proceso bloqueado no despertable por una señal.
 - `TASK_TRACED`: proceso que esta siendo “traceado” por otro.
 - `TASK_STOPPED`: la ejecución del proceso se ha detenido por algunas de las señales de *control de trabajos*.
- ▷ El campo `exit_state` almacena la condición en que ha finalizado:
 - `EXIT_DEAD`: va a ser eliminado, su padre ha invocado `wait()`:
 - `EXIT_ZOMBIE`: el padre aún no ha realizado `wait()`.

Diagrama de estados de procesos/hilos



Transiciones entre estados

▷ Llamadas al sistema:

- `clone()`: llamada al sistema para crear un proceso/hilo.
- `exit()`: llamada para finalizar un proceso.

▷ Funciones del kernel:

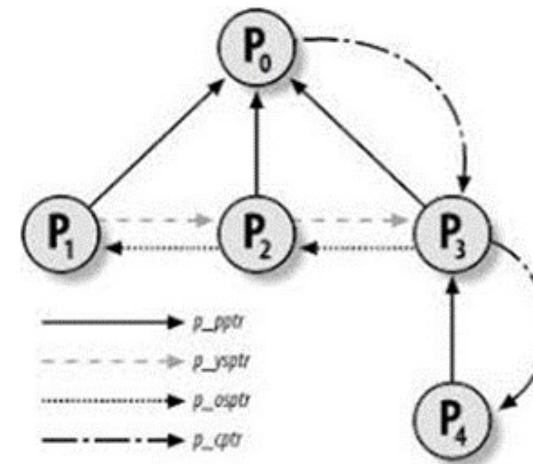
- `sleep()`: usada por el proceso llamador para bloquearse/dormir en espera de un determinado evento.
- `wakeup()`: un componente de kernel desbloquea/desperta a un proceso dormido cuando se ha producido el evento por el que espera.
- `schedule()`: usada por el planificador para decidir que proceso tendrá tras su invocación el control de la CPU.

Colas asociadas a estado

- ▷ Existe una lista de procesos doblemente enlazada con todos los procesos del sistema y a la cabeza esta el *swapper* (PID=0, `task_struct_init_task`).
- ▷ Los estados `TASK_STOPPED`, `EXIT_ZOMBIE`, ó `EXIT_DEAD` no están agrupados en colas.
- ▷ Los procesos `TASK_RUNNING` están en diferentes **colas de procesos ejecutables**: una cola por procesador (lo veremos en §Planificación).

Jerarquía de procesos

- ▷ Para gestionar procesos de forma conjunta, todos los procesos forman parte de una jerarquía, con el proceso `systemd / init` (PID=1) a la cabeza.
 - Todo proceso tiene exactamente un padre
 - Procesos *hermanos* (*siblings*) = procesos con el mismo padre.
- ▷ La relación entre procesos se almacena en el PCB:
 - `parent`: puntero al padre
 - `children`: lista de hijos.
 - ...



Fuente: D.P. Bovet and M. Cesati,
Understanding Linux Kernel, O'Reilly, 2006.

Manipulación de procesos

- ▷ Algunas llamadas para la manipulación de procesos:
 - **clone()** : crea un proceso o hilo desde otro con las características que se especifican en los argumentos.
 - **exec()** : ejecuta un programa dentro de un proceso existente a partir del ejecutable que se pasa como argumento.
 - Al invocar a **exec()** el SO destruye el espacio de direcciones a nivel de usuario del proceso invocador, solo se queda el descriptor de proceso, y construye un nuevo ED de usuario a partir de la información del formato ELF del programa invocado.

clone()

- ▷ El prototipo de la función en la biblioteca *glibc* es:

```
#define <sched.h>
int clone(int (*func()) (void *), void *child_stack,
          int flags, void *func_arg, ... /*pid_t *ptid,
          struct user_desc *tls, pid_t ctid */);
```

- ▷ El modelo de hilos de Linux es 1:1 (una hebra de usuario soportada por una hebra kernel).
- ▷ `clone()` es una llamada endémica de Linux, en sistemas UNIX, la llamada para crear procesos (no hilos) es `fork()`. En Linux, `fork()` se implementa como:

```
clone(SIGCHLD, 0)
```

Nota: El prototipo de la llamada al sistema `clone()` es

```
clone(flags, stack, ptid, ctid, regs)
```

clone() (cont.)

▷ Significado de algunos de los indicadores de creación:

- CLONE_FILES – Hilo padre e hijo comparten los mismos archivos abiertos
- CLONE_FS – Padre e hijo comparten la información del sistema de archivos
- CLONE_VM – Padre e hijo comparten el espacio de direcciones de usuario
- CLONE_SIGHAND – Comparten los manejadores de señales y señales bloqueadas
- CLONE_THREAD – Ambos procesos/hilos pertenecen al mismo grupo (mismo GID).

Ejemplo de clone

```
#include <stdio.h>  #include <unistd.h> #include
<sys/types.h>
#include <linux/unistd.h> #include <sys/syscall.h>
#include <errno.h>
#include <linux/sched.h> #include <malloc.h>

int variable=3;

int thread(void *p) {
    int tid;
    variable++;
    tid = syscall(SYS_gettid);
    printf("\nPID - TID del hijo: %d - %d, var hijo:
           %d\n",getpid(),tid, var);
    sleep(5);
}
```


Ejemplo (cont.)

```
int main() {
    void **stack;
    int i, tid;
    stack = (void **)malloc(15000);
    if (!stack)
        return -1;
    i = clone(thread, (char*) stack + 15000, CLONE_VM|

        CLONE_FILES|CLONE_FS|CLONE_THREAD|CLONE_SIGHAND,
        NULL); /* (1) */

    sleep(2);
    if (i == -1)
        perror("clone");
    tid = syscall(SYS_gettid);
    printf("\nPID - TID del padre: %d - %d, var padre
%d\n\n", getpid(), tid, var);
    return 0;
}
```

Ejecución

> Si compilamos y ejecutamos el ejemplo tal como esta:

```
> ./clon
```

```
PID - TID del hijo: 7917 - 7918, var hijo: 4
```

```
PID - TID del padre: 7917 - 7917, var padre 4
```

> Si en (1) Si solo dejamos:

```
i = clone(thread, (char*) stack + 15000, NULL, NULL)
```

```
> ./clon2
```

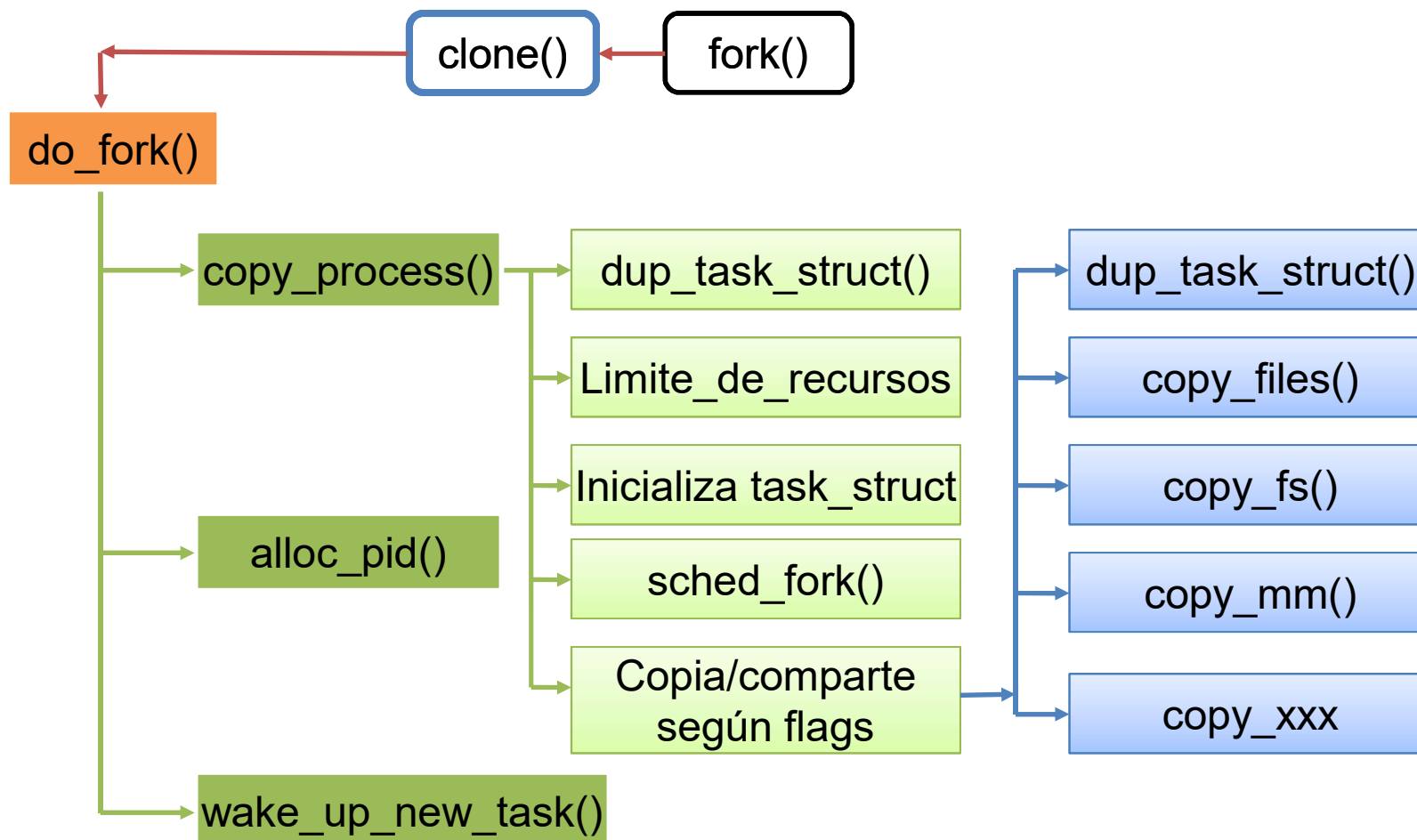
```
PID - TID del hijo: 7971 - 7971, var hijo: 4
```

```
PID - TID del padre: 7970 - 7970, var Padre 3
```

> ¿Qué ha pasando?

Clone: implementación

▷ La estructura del código de `clone()`:



clone() : explicación

▷ `do_fork()` esta implementada en *kernel/fork.c*.

▷ El trabajo que realiza es:

- `dup_task_struct`: copia el descriptor del proceso actual (`task_struct`, pila y `thread_info`).
- `alloc_pid`: le asigna un nuevo PID
- Al inicializar la `task_struct`, diferenciamos los hilos padre e hijo y ponemos a este último en estado no-interrumpible.
- `sched_fork`: marcamos el hilo como `TASK_RUNNING` y se inicializa información sobre planificación
- Copiamos o compartimos componentes según los indicadores de la llamada.
- Asignamos ID, relaciones de parentesco, etc.

clone () : consideraciones

- ▷ Clone debe hacer algunas comprobaciones:
 - Algunos indicadores no tienen sentido juntos, por ejemplo, `CLONE_NEWNS` y `CLONE_FS`.
 - Otros debe aparecer a la vez: `CLONE_THREAD` con `CLONE_SIGHAND`, o `CLONE_SIGHAND` con `CLONE_VM`.
- ▷ Cuando aparece el indicador `CLONE_XXX`, la estructura `xxx_struct` se comparte (no se copia).

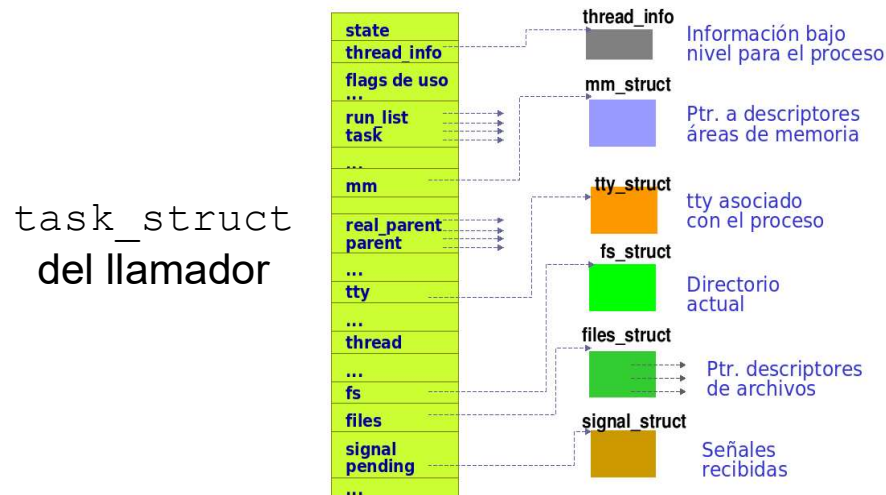
Nota: El kernel asigna a cada estructura compartida un **contador de referencias** que lleva la cuenta que cuantos hilos la comparten. Cada vez que borramos una de las referencias a la estructura, decrementamos dicho contador; si este llega a cero, la estructura se libera. Esto elimina la necesidad de un recolector de basura.

Actividad en grupo

- ▷ En el ejemplo anterior, hemos utilizado la función `clone()` con los argumentos siguientes:

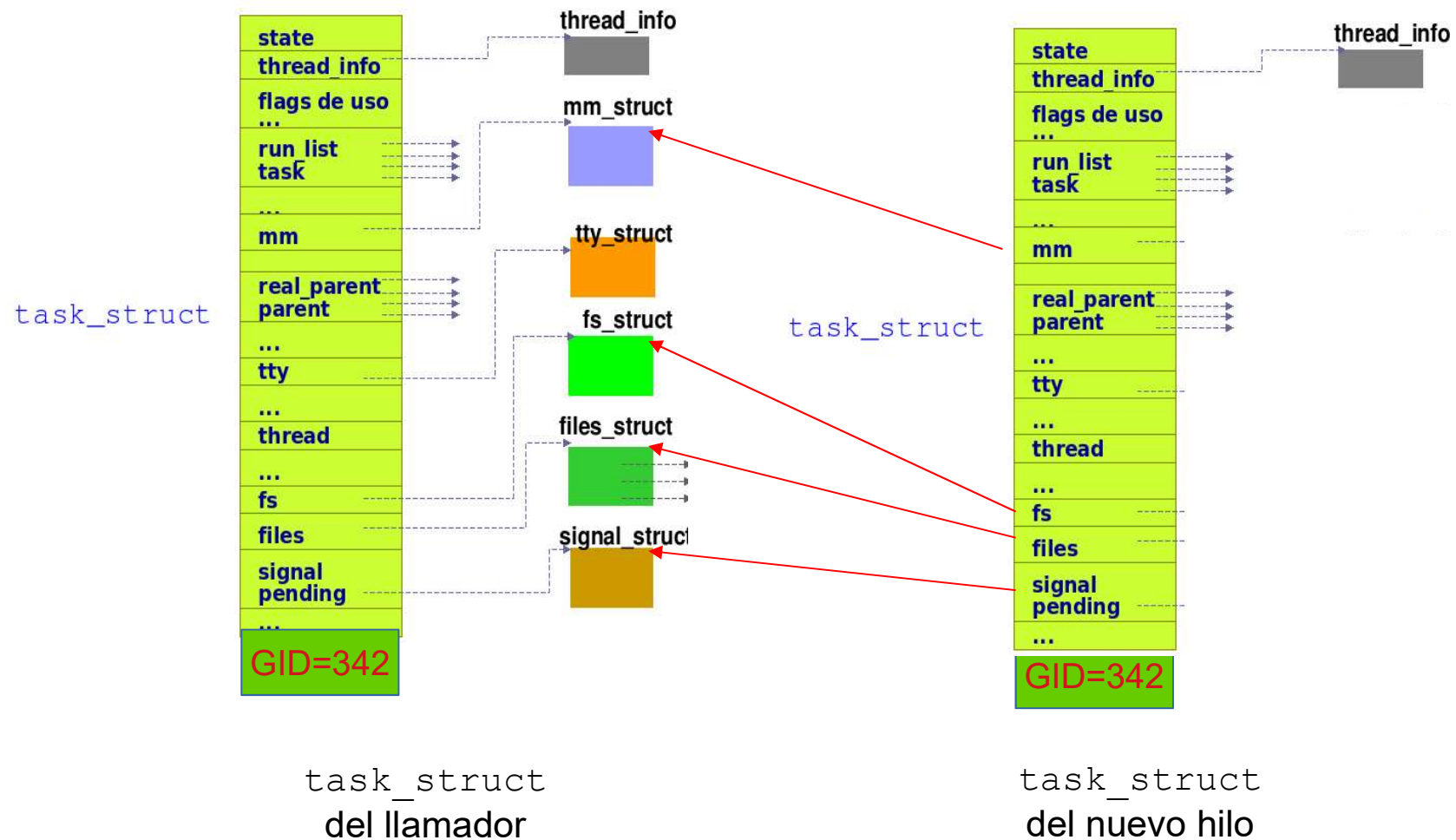
`CLONE_VM|CLONE_FILES|CLONE_FS|CLONE_THREAD|CLONE_SIGHAND`

Hacer un dibujo que represente los principales elementos de las `task_struct` de ambos procesos relacionados con los indicadores mencionados.



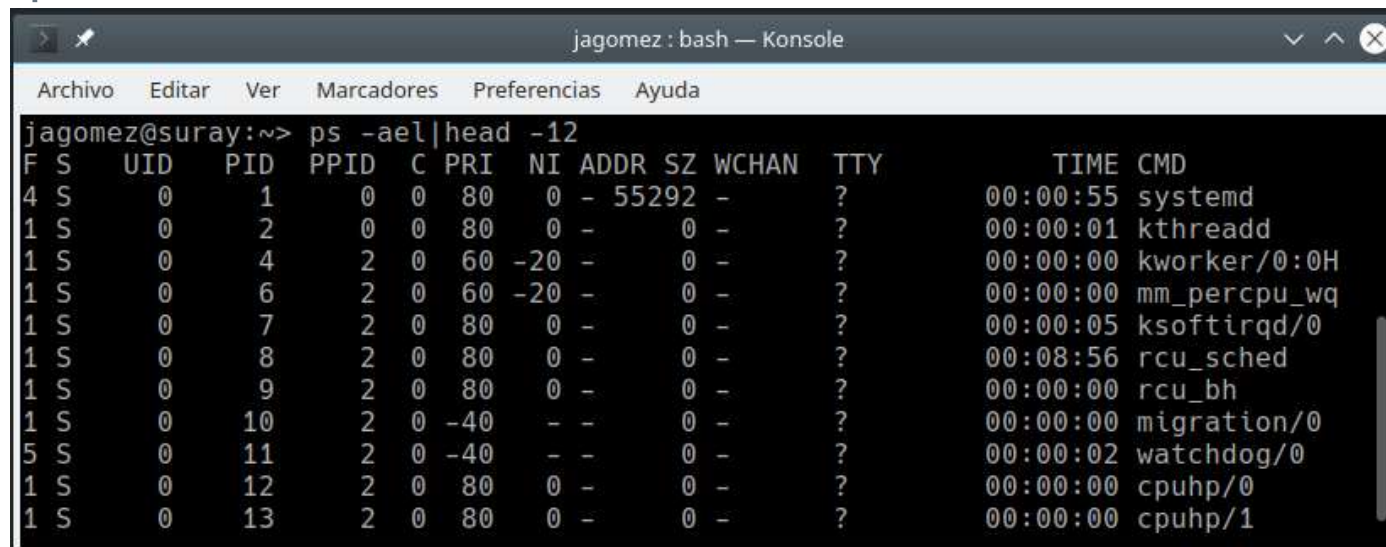
¿Dibujar la
task_struct
del nuevo hilo ?

Actividad en grupo: respuesta



Hilos kernel

- ▷ Son hilos que no tienen espacio de direcciones de usuario. Por tanto, su descriptor tiene `task_struct->mm=NULL`.
- ▷ Realizan labores de sistema, sustituyendo a los antiguos *demonios* de Unix (introducidos por eficiencia).
- ▷ Solo se pueden crear desde otro hilo kernel con la función `kthread_create()`.
- ▷ Ejemplo:



```
jagomez@suray:~> ps -ael | head -12
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	0	1	0	0	80	0	-	55292	-	?	00:00:55	systemd
1	S	0	2	0	0	80	0	-	0	-	?	00:00:01	kthreadd
1	S	0	4	2	0	60	-20	-	0	-	?	00:00:00	kworker/0:0H
1	S	0	6	2	0	60	-20	-	0	-	?	00:00:00	mm_percpu_wq
1	S	0	7	2	0	80	0	-	0	-	?	00:00:05	ksoftirqd/0
1	S	0	8	2	0	80	0	-	0	-	?	00:08:56	rcu_sched
1	S	0	9	2	0	80	0	-	0	-	?	00:00:00	rcu_bh
1	S	0	10	2	0	-40	-	-	0	-	?	00:00:00	migration/0
5	S	0	11	2	0	-40	-	-	0	-	?	00:00:02	watchdog/0
1	S	0	12	2	0	80	0	-	0	-	?	00:00:00	cpuhp/0
1	S	0	13	2	0	80	0	-	0	-	?	00:00:00	cpuhp/1

Terminar un proceso

- ▷ La terminación de un proceso se produce de forma:
 - *Voluntaria* – cuanto éste en el `main()` invoca a:
 - `exit()` o `return()` – función que finaliza el proceso primero a nivel de biblioteca, luego a nivel de sistema.
 - `_exit()` - es la llamada al SO para ponerle fin y que no da la oportunidad de finalizar el proceso a nivel de biblioteca.
 - *Involuntariamente*: el proceso recibe una señal y se aplica la acción por defecto, que es terminar su ejecución.
- ▷ Como un proceso que finaliza debe pasar a su padre algunos datos (código finalización, tiempos de ejecución, ..), la finalización total de un proceso en el sistema se materializa en dos etapas, cada una de ellas materializada por las llamadas `exit()` y `wait()`, respectivamente.

exit()

- ▷ Invoca a `do_exit()` para finalizarlo (*kernel/exit.c*):
 - Activa `PF_EXITING`
 - Decrementa los contadores de uso de `mm_struct`, `fs_struct`, `files_struct`. Si estos contadores alcanzan el valor 0, libera los recursos.
 - Ajusta el `exit_code` del descriptor, que será devuelto al padre, con el valor pasado al invocar a `exit()`.
 - Envía al padre la señal de finalización; si tiene algún hijo le busca un padre en el grupo o el *init*, y pone el estado a `TASK_ZOMBIE`.
 - Invoca a `schedule()` para ejecutar otro proceso.
- ▷ Ya solo queda: la pila `kernel`, `thread_info` y `task_struct`, de cara a que el padre pueda recuperar el código de finalización. ¿Cuando se libera el resto?

wait()

- ▷ `wait()`: llamada que bloquea a un proceso padre hasta que uno de sus hijo finaliza; cuando esto ocurre, devuelve al llamador el PID del hijo finalizado y el estado de finalización (código de finalización, *coredump* y señal).
- ▷ Esta función invoca a `release_task()` que:
 - Elimina el descriptor de la lista de tareas.
 - Si es la última tarea de su grupo, y el líder esta zombi, notifica al padre del líder zombi.
 - Libera la memoria de la pila `kernel`, `thread_info` y `task_struct`.



3.

Planificación de la CPU

A qué proceso y cómo asignar las CPUs

Contenido del apartado

- ▷ Tipos de planificadores.
- ▷ Tipos de planificación.
- ▷ Criterios, algoritmos y métricas de planificación: Prioridades, RR, colas múltiples.
- ▷ Planificación en Linux:
 - Planificador CFS (*Completely Fair Scheduler*)
 - Planificación de tiempo-real
 - Planificación en multiprocesadores
- ▷ Ahorro de energía

Trabajo individual

- ▷ Tipos de planificadores - §9.1 del W. Stallings “Tipos de planificación del procesador”.
- ▷ Criterios, algoritmos y métricas de planificación - §9.2 del W. Stallings “Algoritmos de planificación”. Ver:
 - FIFO
 - Prioridades
 - *Round-Robin*
 - Colas múltiples (con/sin realimentación)

Tipos de planificadores

▷ Planificador a:

- - **Largo plazo** – procesos por lotes
- - **Corto plazo** o *scheduler*
- - **Medio plazo** – gesto de memoria

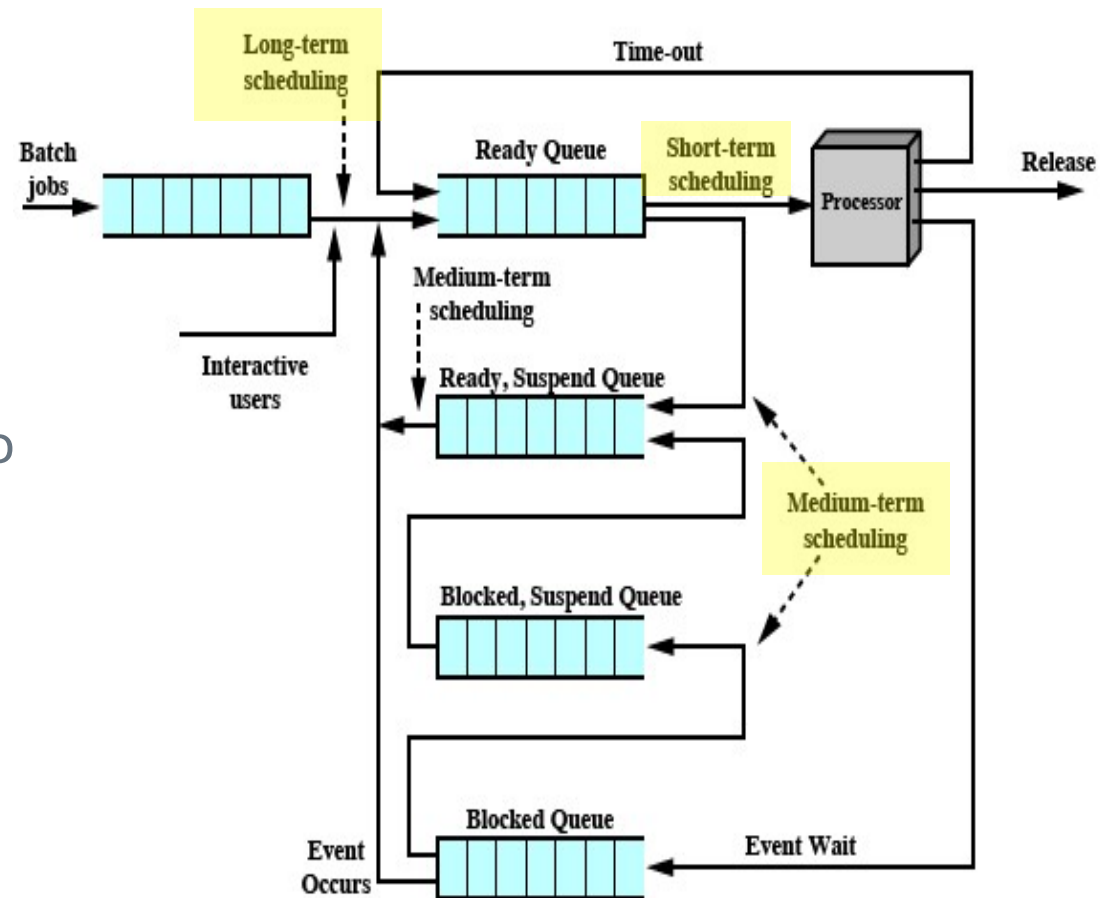


Figura del libro de W. Stallings

Tipos de planificación

▷ **Planificación apropiativa** (*preemptive*) – al proceso actual se le puede retirar la CPU.

▷ **Planificación no apropiativa** (*non preemptive*) – al proceso actual NO se le puede retirar la CPU.

- ▷ La NO apropiación ha sido utilizada por los constructores de SOs como mecanismo de grano grueso de sincronización en modo kernel.
- ▷ Los kernel de tiempo-real necesitan ser apropiativos.

Proceso nulo

- ▷ ¿Qué ocurre si el planificador a corto plazo no encuentra procesos preparados para ejecutarse cuando es invocado?
- ▷ La solución óptima es introducir el proceso nulo, es decir un proceso que:
 - siempre este listo para ejecutarse.
 - tenga la prioridad más baja.

Proceso nulo: implementación

▷ 1ª aproximación:

```
Planificador() {  
  while (true) {  
    if  
    (cola_preparados==vacía)  
      halt;  
    else {  
      Selecciona (Pj);  
  
      Cambio_contexto(Pi,Pj);  
    }  
  }  
}
```

▷ Solución óptima:

```
while (true){  
  Selecciona (Pj);  
  
  Cambio_contexto(Pi,Pj);  
}
```

- donde hemos creamos un proceso nulo/ocioso que:
 - Siempre “preparado”
 - Menor prioridad del sistema (no compite con el resto por la CPU)

Linux y la apropiatividad

- ▷ Los kernel actuales son apropiativos y permiten ajustar el grano de apropiatividad según el uso que le vayamos a dar.
- ▷ Se implementa a través de **puntos de apropiación**: puntos del flujo de ejecución kernel donde es posible apropiar al proceso actual sin incurrir en una condición de carrera.
- ▷ Dado que la invocación asíncrona del planificador podría generar condiciones de carrera, esta se difiere hasta alcanzar un punto de apropiación.

Planificación asíncrona

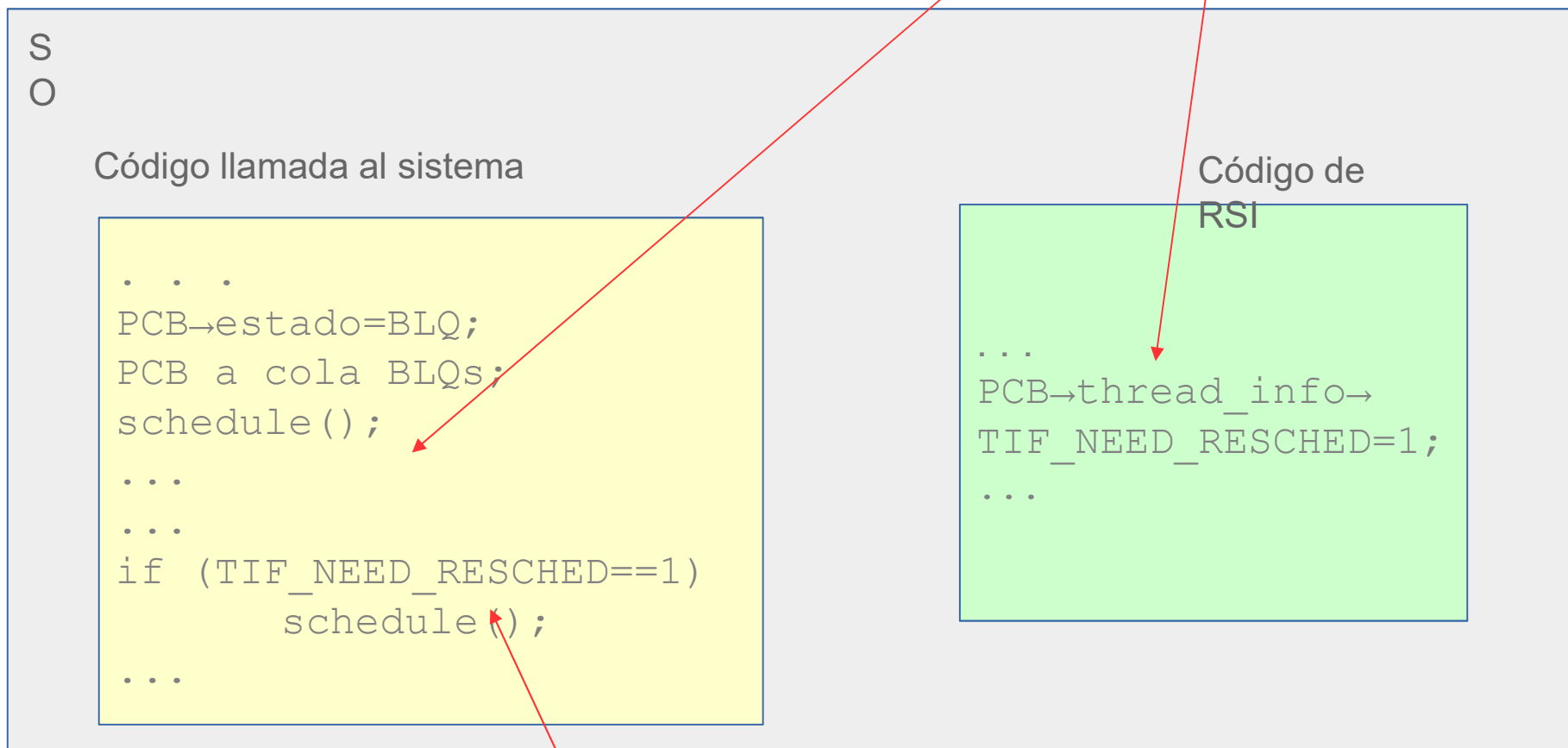
- ▷ Las RSIs en lugar de invocar a `schedule()` directamente, solo indican que es necesario planificar cuando sea posible:

```
task_struct->thread_info->TIF_NEED_RESCHED=1
```

- ▷ Fuera del tratamiento de la interrupción y cuando las EDs kernel estén en estado seguro (punto de apropiación), se invocará al planificador.

Planificación síncrona y asíncrona

Invocación síncrona y asíncrona



Punto de apropiación

Trabajo en grupo 2.2

- ▷ Supongamos que ejecutamos un kernel que utiliza planificación NO apropiativa (linux <2.6), ¿puede el procesador y el kernel manejar las interrupciones de los dispositivos?

Trabajo en grupo 2.2: respuesta

- ▷ Debemos de tratarlas pues el no hacerlos afectaría a la responsividad del sistema operativo frente a los dispositivos
- ▷ La cuestión es tener claro que pueden ejecutarse en tanto en cuanto tras la ejecución de la RSI el contexto del proceso que se estaba ejecutando cuando se produjo la misma quede inalterado → al finalizar la RSI dejo la máquina en el estado encontrado al inicio → no planifico.

Planificación en tiempo-real

- ▷ Un RTOS debe planificar las tareas para que todas puedan cumplir sus plazos (*deadlines*).
- ▷ Esto afecta:
 - Planificador de tiempo-real: debe tener en cuenta los plazos (los SOS de propósito general no lo hacen. Ej. Algoritmo EDF - *Early Deadline First*)
 - Debemos reducir la latencia de despacho:
 - El kernel debe ser apropiativo.
 - No debe permitir la “planificación oculta”.

Latencia de despacho



- ❑ Apropiar al proceso en ejecución, y liberar los recursos usados por el proceso de baja prioridad, y necesitados por el de alta prioridad.
- ❑ Si no se liberan se puede producir **inversión de prioridad**.

- ▷ Protocolos para evitar la inversión de prioridad:
 - Herencia de prioridad: PI-mutex en Linux
 - Protocolo tope (*ceiling*)

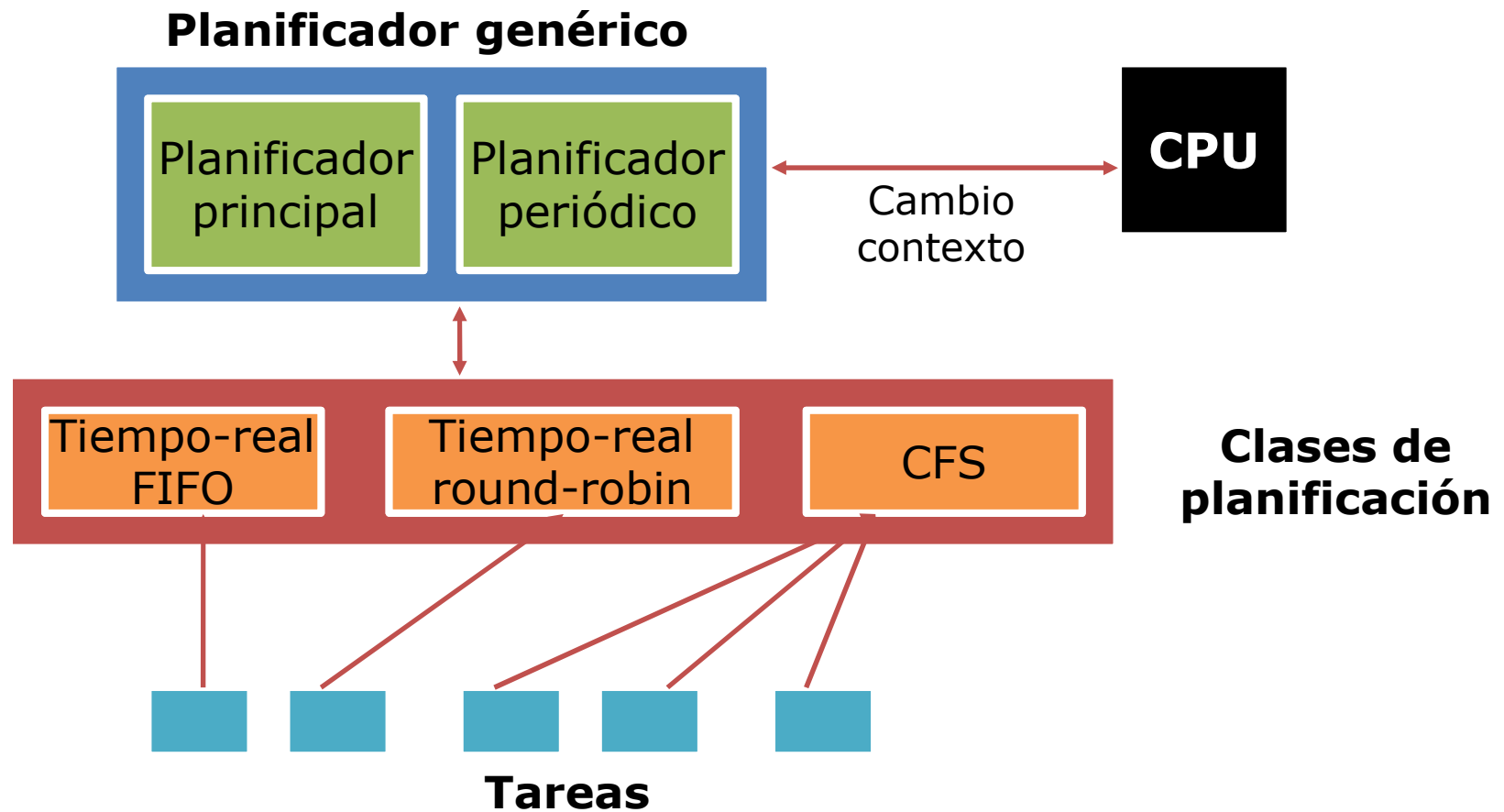
El planificación de Linux

- ▷ Función `schedule()` en *kernel/sched.c*.
- ▷ El planificador genérico tiempo dos componentes:
 - Componente *síncrono*: se activa cuando un proceso va a ceder la CPU.
 - Componente *asíncrono*: se activa periódicamente.

Clases de planificación

- ▷ Un SO conforme POSIX1.b como Linux debe soportar al menos tres clases de planificación:
 - 2 de tiempo-real: `SCHED_FIFO` y `SCHED_RR`
 - 1 de tiempo compartido: `SCHED_OTHERS` (en Linux, `SCHED_NORMAL`), que a su vez contiene las sub-clases:
 - `SCHED_BATCH` – identificar procesos por lotes
 - `SCHED_IDLE` – hebras ociosas
- ▷ El planificador soporta maquinas NUMA, UMA, multinúcleos y multihebrado.

Componentes del planificador



Planificación y task_struct

- ▷ Algunos campos relacionados con la planificación:
 - `prio`: prioridad usada por el planificador
 - `rt_priority`: prioridad de tiempo-real.
 - `sched_class`: clase de planificación a la que pertenece el proceso.
 - `sched_entity`: el mecanismo *cgroups* establece que se planifiquen entidades (proceso o grupo de procesos) permitiendo asegurar un porcentaje de CPU al grupo completo.

Planificación y task_struct (ii)

▷ `policy`: política de planificación aplicable:

- `SCHED_NORMAL`: manejados por CFS, como:
- `SCHED_BATCH`: procesos no interactivos acotados por computo, y desfavorecidos por las decisiones de planificación. Nunca apropien a otro proceso gestionado por CFS y no interfieren con trabajos interactivos. Aconsejable en situaciones en las que no se desea decrementar la prioridad estática con `nice`, pero la tarea no debería influenciar la interactividad del SO.
- `SCHED_IDLE`: tareas de poca importancia con peso relativo mínimo. No es responsable de planificar la tarea ociosa.
- `SCHED_RR` y `SCHED_FIFO`: implementan procesos de tiempo-real blandos (soft) gestionados por la clase de tiempo-real

Clases de planificación

- ▷ Suministran la conexión entre el planificador genérico y el planificador individual.
- ▷ Hay una instancia de la estructura por clase de planificación.
- ▷ Forman una jerarquía plana que determina el orden de ejecución de los procesos:

procesos tiempo-real > procesos CFS > procesos idle.

- ▷ La jerarquía se establece en tiempo de compilación (no hay mecanismo para añadir una nueva clase dinámicamente).

Clases de planificación: definición

```
struct sched_class {
    const struct sched_class *next;
    void (*enqueue_task()); /* añade proceso a la cola de
                             ejecución rq, y nr_running++ */
    void (*dequeue_task()); /* lo elimina de rq, y nr_running-- */
    void (*yield_task) (); /* cede el control de la CPU */
    struct task_struct * (*pick_next_task) (); /* selecciona
                                                siguiente tarea a ejecutar */
    void (*put_prev_task) (); /* retira la tarea del procesador */
    void (*set_curr_task) (); /* se invoca al cambiar la tarea de
                               clase */
    void (*task_tick) (); /* invocada por el planificador
                           periodico en cada invocación */
    void (*task_new) (); /* notifica al planificador de la
                          creación de una tarea */
    . . . };
```

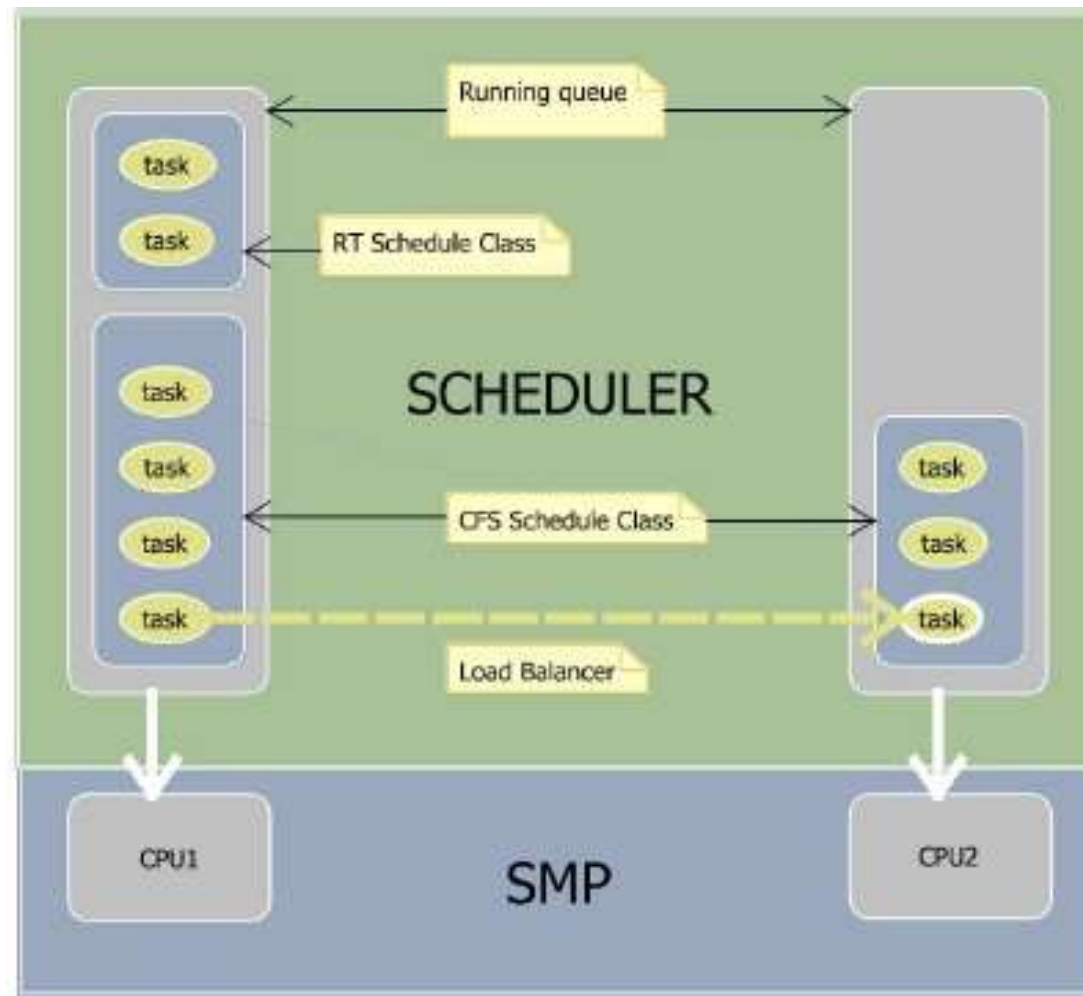

Cola de ejecución

- ▷ Cada procesador tiene su cola de ejecución (`rq` – *run queue*) y cada proceso esta en una única cola.

```
struct rq {
    unsigned long nr_running; /* n° procesos ejecutables */
    #define CPU_LOAD_IDX_MAX 5;
    unsigned long cpu_load[CPU_LOAD_IDX:MAX]; /*historico de
la carga*/
    struct load_weight load; /*carga de la cola */
    struct cfs_rq cfs; /* cola embebida para cfs*/
    struct rt_rq rt; /*cola embebida para rt*/
    struct task_struct *curr, *idle; /*procesos actual y
ocioso*/
    u64 clock; /* reloj por cola; actualizado al invocar al
planificador periodico*/
    . . .
}
```

Colas de ejecución

>

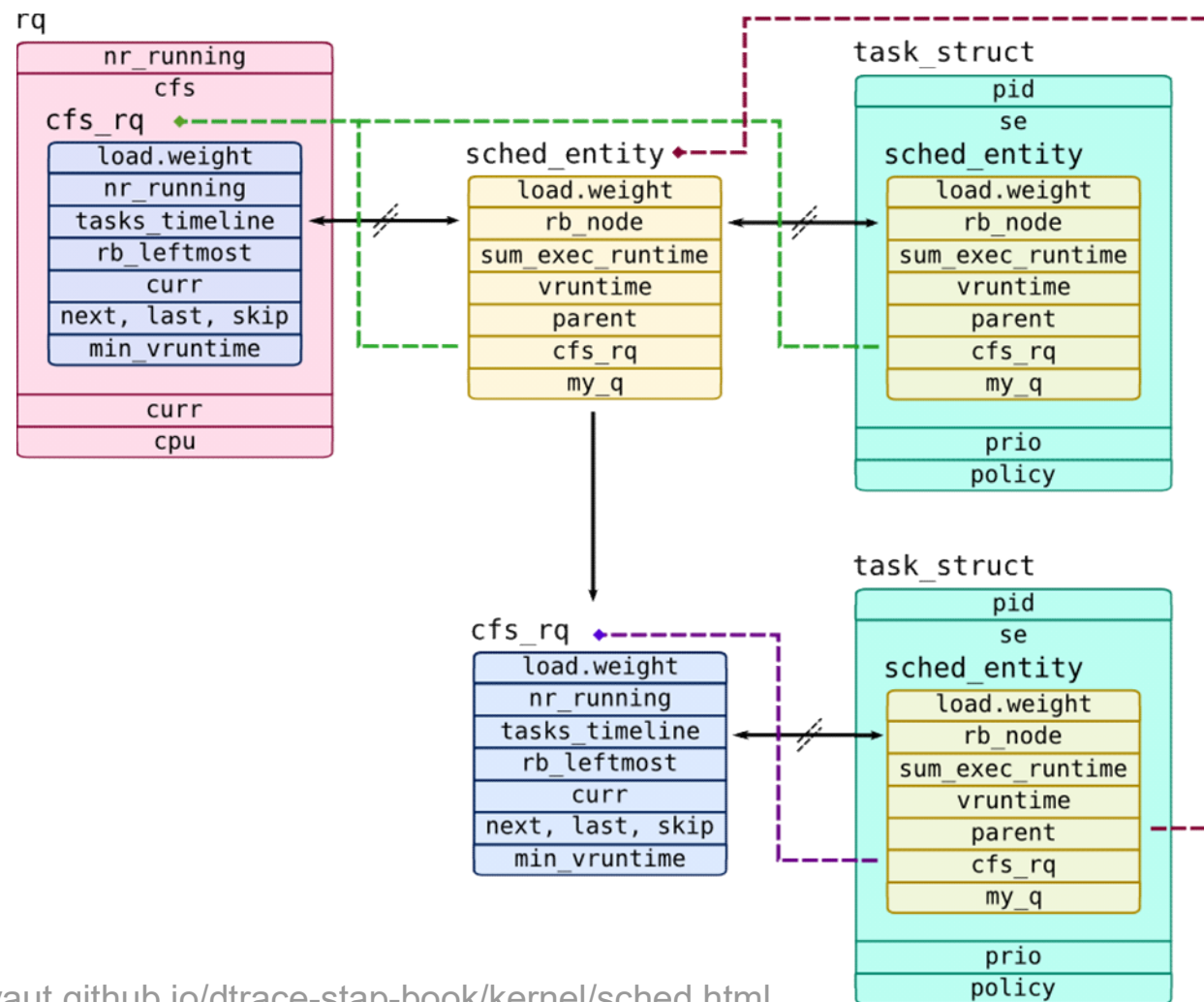


Entidades de planificación

▷ La estructura que describe una entidad:

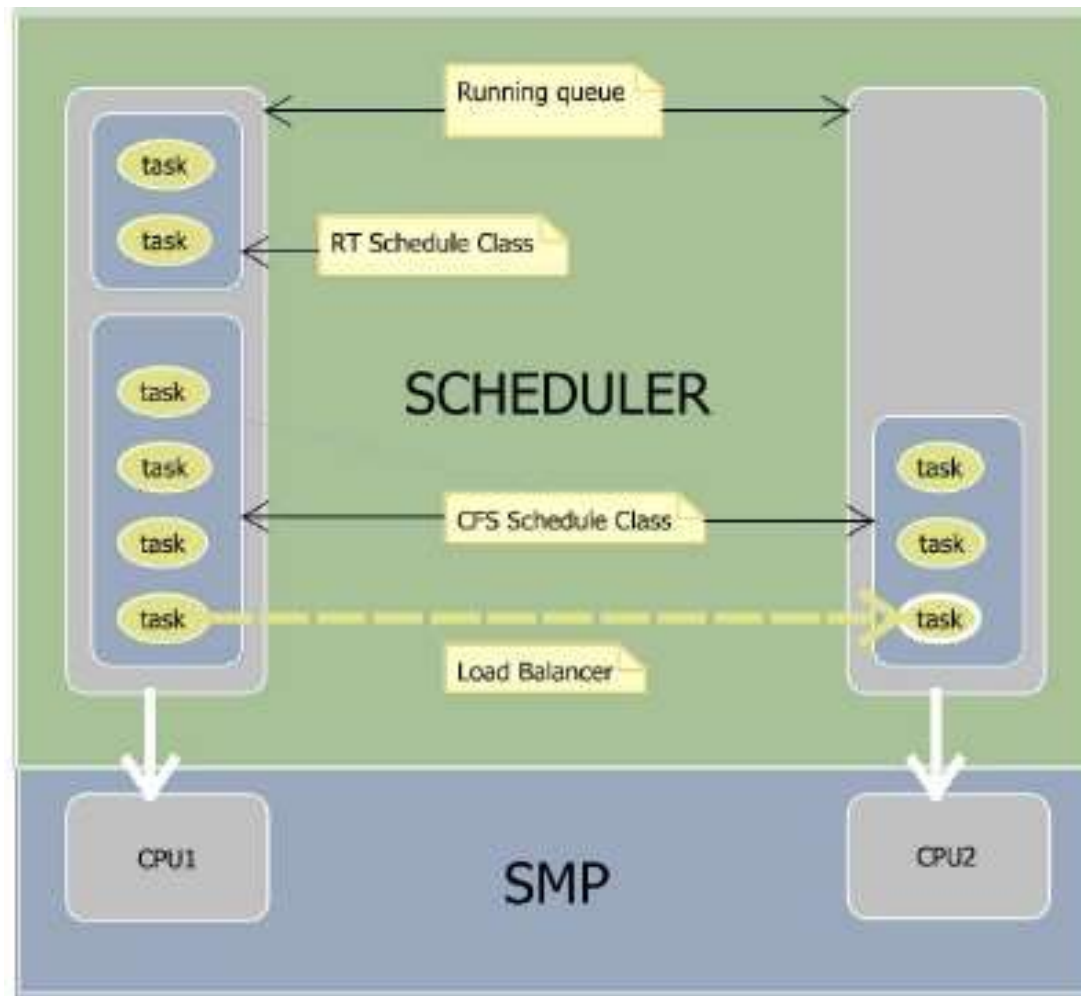
```
struct sched_entity {
    struct load_weight load;      /*para equilibrio de carga*/
    struct rb_node run_node;      /*nodo de arbol rojo-negro*/
    unsigned int on_rq;           /*indica si la entidad esta
                                   planificada en una cola*/
    u64 exec_start;               /* tiempo inicio ejecución */
    u64 sum_exec_start;           /*t consumido de CPU*/
    u64 vruntime;                 /*tiempo virtual */
    u64 prev_sum_exec_runtime;    /* valor salvado de
sum_exec_start al quitarle control CPU*/
    . . .
};
```

Estructuras: relaciones



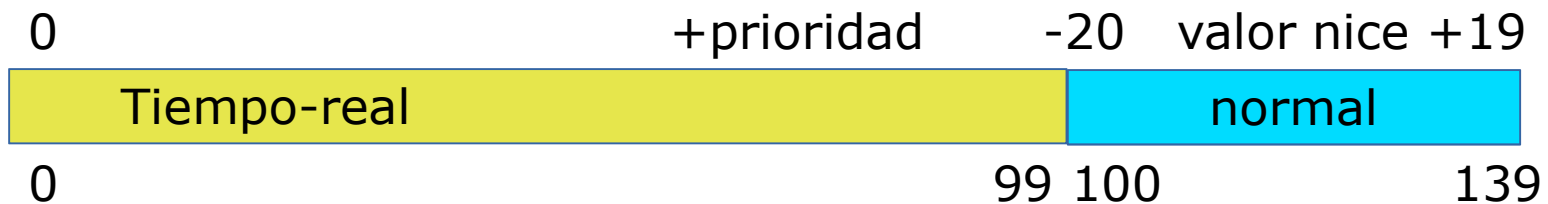
Fuente: <http://myaut.github.io/dtrace-stap-book/kernel/sched.html>

Colas de ejecución



Prioridades

- ▷ El kernel utiliza el rango de prioridades:



- ▷ Valores de prioridad mostrados por GNU:

```
- prioirity    p->prio
- intpri      60 + p->prio
- opri        60 + p->prio
- pri_foo     p->prio - 20
- pri_bar     p->prio + 1
- pri_baz     p->prio + 100
- pri        39 - p->priority
- pri_api    -1 - p->priority
```

```
top:
muestra el valor de ps -o priority
ps con:
-1 muestra el valor intprio
-1 -c muestra -o pri
```

Política de planificación

- ▷ El planificador siempre selecciona para ejecución al proceso con mayor prioridad.
- ▷ Si el proceso pertenece a la clase FIFO, lo ejecutará hasta el final o hasta que se bloquee.
- ▷ En el resto de clases, si hay dos procesos con la misma prioridad, ejecutará al que lleva más tiempo esperando.

Planificador periódico: `scheduler_tick()`

- ▷ Se invoca con una frecuencia de HZ, en `update_process_times()` para contabilizar el *tick* transcurrido al proceso actual (*kernel/timer.c*)
- ▷ Tiene dos funciones principales:
 - Maneja estadísticas kernel relativas a planificación.
 - Activar el planificador periódico de la clase de planificación responsable del proceso actual, delegando la labor en el planificador de clase:

```
curr->sched_class->task_tick(rq, curr);
```

Si debemos replanificar la tarea actual, el método de la clase básicamente activa el bit `TIF_NEED_RESCHED`.

Planificador principal

- ▷ La función `schedule()` se invoca directamente en diversos puntos del kernel para cambiar de proceso.
- ▷ Además, cuando retornamos de una llamada al sistema, comprobamos si hay que replanificar mediante `TIF_NEED_RESCHED`, y si es necesario se invoca a `schedule()`.
- ▷ La función la podemos ver en <http://lxr.linux.no/#linux+v3.1/kernel/sched.c#L4260>.

Planificador: algoritmo

1. Seleccionar la cola y el proceso actual:

```
rq=cpu_rq(cpu);  
prev=rq->cur;
```

2. Desactivar la tarea actual de la cola.

```
deactivate_task(rq, prev, 1);
```

3. Seleccionar el siguiente proceso a ejecutar.

```
next=pick_next_task(rq, next);
```

4. Invocar al cambio de contexto:

```
if (likely(prev!=next)  
    context_switch(rq, prev, next);
```

5. Comprobar si hay que replanificar:

```
if (need_resched())  
    goto need_resched;
```

Cambio de contexto

- ▷ El cambio de contexto descansa en las funciones:
 - `switch_mm()` que cambia el contexto de memoria de usuario descrito por `task_struct->mm`
 - `switch_to(prev, next, prev)` - cambia los contenidos de los registros del procesador y la pila kernel. Equivale a `prev=switch(prev, next)`.
- ▷ Si la tarea entrante/saliente es una hebra kernel utiliza un mecanismo denominado *TLB perezoso*, en el que se indica al procesador que realmente no hay que conmutar de memoria.

Planificador CFS

<https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/>



Pesos de carga

- ▷ La importancia de un proceso viene dado por la prioridad, o por su **peso de carga** (*load weight*).
- ▷ *Idea: un proceso que disminuye su prioridad en un nivel nice obtiene el 10% más de CPU, mientras que aumentar un nivel resta un 10% de CPU.*
- ▷ Ejemplos de uso con dos procesos:
 - Con `nice=0`: cada uno obtiene el 50% de CPU: $1024/(1024+1024)$.
 - Un proceso sube un nivel `nice` (prioridad +1) → decrementa su peso ($1024/1.25 \approx 820$). Ahora, $1024/(1024+820) = 0.55$ y $820/(1024+820) = 0.45$. Un proceso obtiene el 55% de CPU y el otro un 45% (diferencia del 10%).
- ▷ Estos cálculos los realiza `set_load_weight()`.

Pesos de carga (ii)

- ▷ La importancia de un proceso viene dada por la prioridad, o por su **peso de carga** (*load weight*).
- ▷ Para ello, el kernel convierte prioridades en pesos de carga:

```
static const int prio_to_weight[40] = {  
/* -20 */ 88761, 71755, 56483, 46273, 36291,  
/* -15 */ 29154, 23254, 18705, 14949, 11916,  
/* -10 */ 9548, 7620, 6100, 4904, 3906,  
/* -5 */ 3121, 2501, 1991, 1586, 1277,  
/* 0 */ 1024, 820, 655, 526, 423,  
/* 5 */ 335, 272, 215, 172, 137,  
/* 10 */ 110, 87, 70, 56, 45,  
/* 15 */ 36, 29, 23, 18, 15,  
};
```

La matriz contiene un valor para cada nivel *nice* en el rango [0,39]. El multiplicador entre entradas es ≈ 1.25 .

Clase CFS

- ▷ CFS intenta modelar un procesador multitarea perfecto.
- ▷ **No asigna rodajas de tiempo**, asigna una **proporción del procesador** dependiente de la carga del sistema. Cada proceso se ejecuta durante un tiempo proporcional a su peso dividido por la suma total de pesos.

$$TP_i = (\text{Peso}_i / \sum \text{Pesos}_j) * p$$

con $\begin{cases} p = \text{sched_latency}, & \text{si } n > \text{nr_latency} \\ \text{min_granularidad} * n, & \text{en otro caso} \end{cases}$

- ▷ Ya que si $n \rightarrow \infty$, $TP_i \rightarrow 0$, se define una `granularidad_mínima` (suelo de tiempo). En la implementación actual:
 - `sched_latency=8`
 - `nr_latency=8`
 - `min_granularity= 1 us.`

Tiempo asignado a un proceso

- ▷ Como la carga es dinámica, para el cálculo tiempo asignado se usa un **periodo** que se asigna en base al número de procesos en la cola:
 - 5 o menos procesos: 20ms
 - Sistema cargado: 5ms más por proceso.

Tiempo asignado= (longitud periodo*peso) / peso rq

- ▷ Ejemplo:

Proceso	P ₁	P ₂	P ₃
Nice	+5	0	-5
Peso	335	1024	3121
Tiempo asignado	1.5 ms	4.5 ms	14 ms

Clase CFS: definición

▷ Definida en *kernel/sched_fair.c*:

```
static const struct sched_class fair_sched_class = {  
    .next=&idle_sched_class,  
    .enqueue_task= enqueue_task_fair,  
    .dequeue_task= dequeue_task_fair,  
    .yield_task= yield_task_fair,  
    .check_preempt_curr= check_preempt_wakeup,  
    .pick_next_task= pick_next_task_fair,  
    .put_prev_task= put_prev_task_fair,  
    . . .  
    .task_tick= task_tick_fair,  
    . . .  
}
```

CFS: selección de proceso

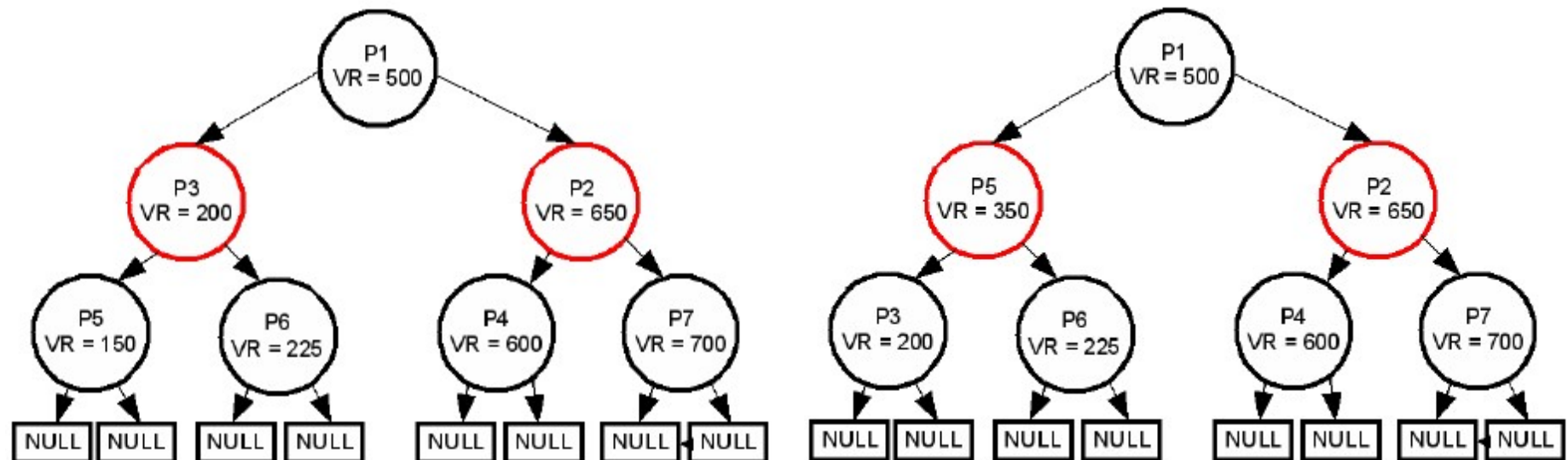
- ▷ **Tiempo virtual de ejecución** (*vruntime*)= tiempo de ejecución real normalizado por el peso de todos los procesos en ejecución. Este tiempo es gestionado por `update_curr()` definida en *kernel/sched_fair.c*:

```
vruntime=(PesoNice0/Peso cola)x tiempo_real  
        = tiempo_ejecucion_actual* 1024/peso_rq
```

- ▷ CFS intenta equilibrar los tiempos virtuales de ejecución de los procesos con la regla: “se elige para ejecución el proceso con *vruntime* más pequeño”.

CFS: selección de proceso

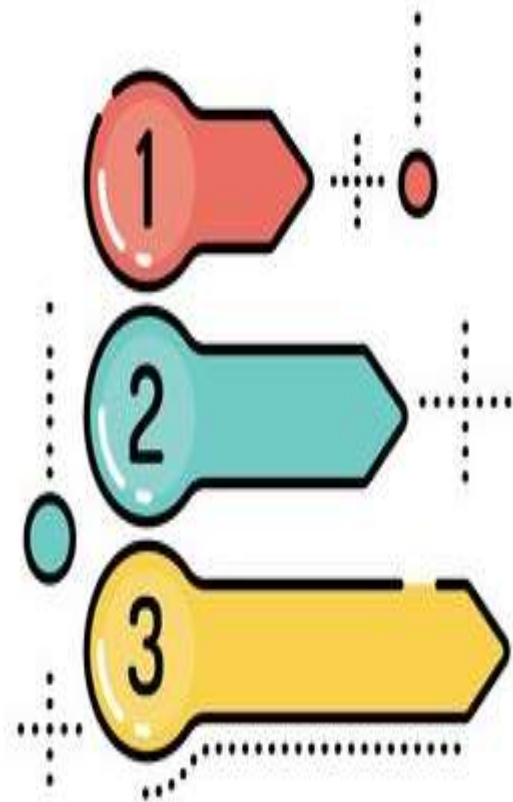
- Cola de ejecución de CFS es un *árbol rojo-negro* (`rbtree`): árbol de búsqueda binario auto-equilibrado donde la clave de búsqueda es `vruntime`. Inserción/borrado con $O(\log n)$.
El proceso con `vruntime` menor es la hoja más a la izquierda en el árbol.



Parámetros de planificación

- ▷ Lista de las variables relacionadas con planificación:
`% sysctl -A | grep "sched" | grep -v "domain"`
- ▷ Valor actual de las variables ajustables:
`/proc/sched_debug`
- ▷ Estadísticas cola actual:
`/proc/schedstat`
- ▷ Información planificación proceso PID:
`/proc/<PID>/sched`

Planificador de tiempo-real

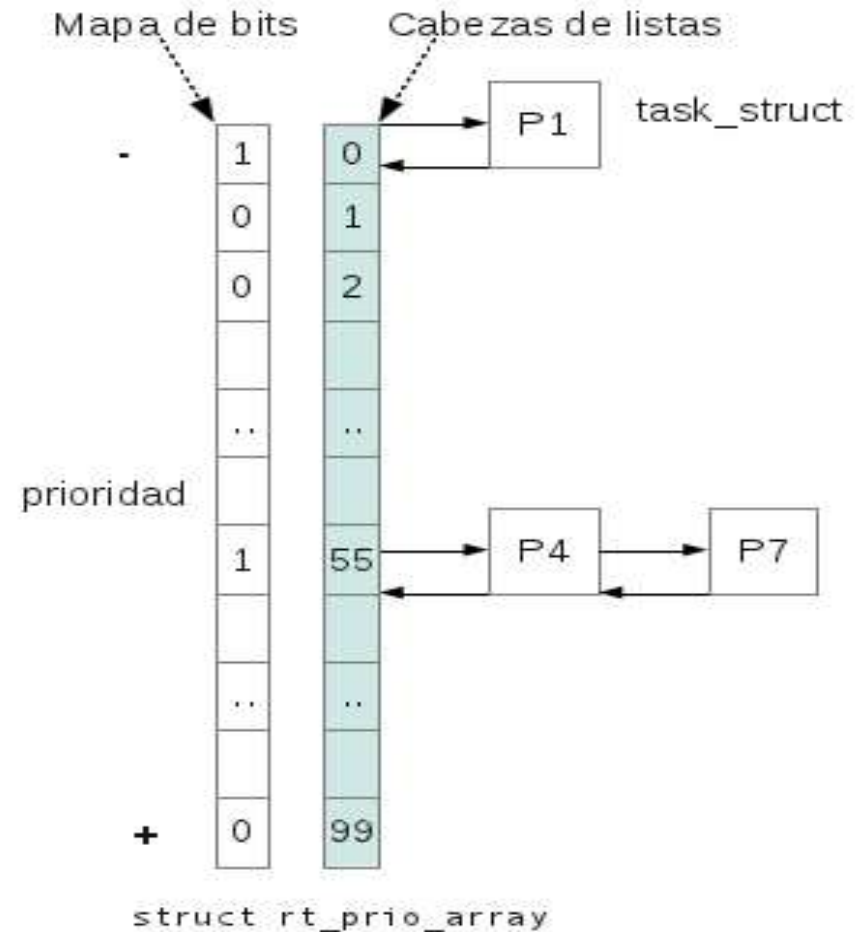


PRIORITIES

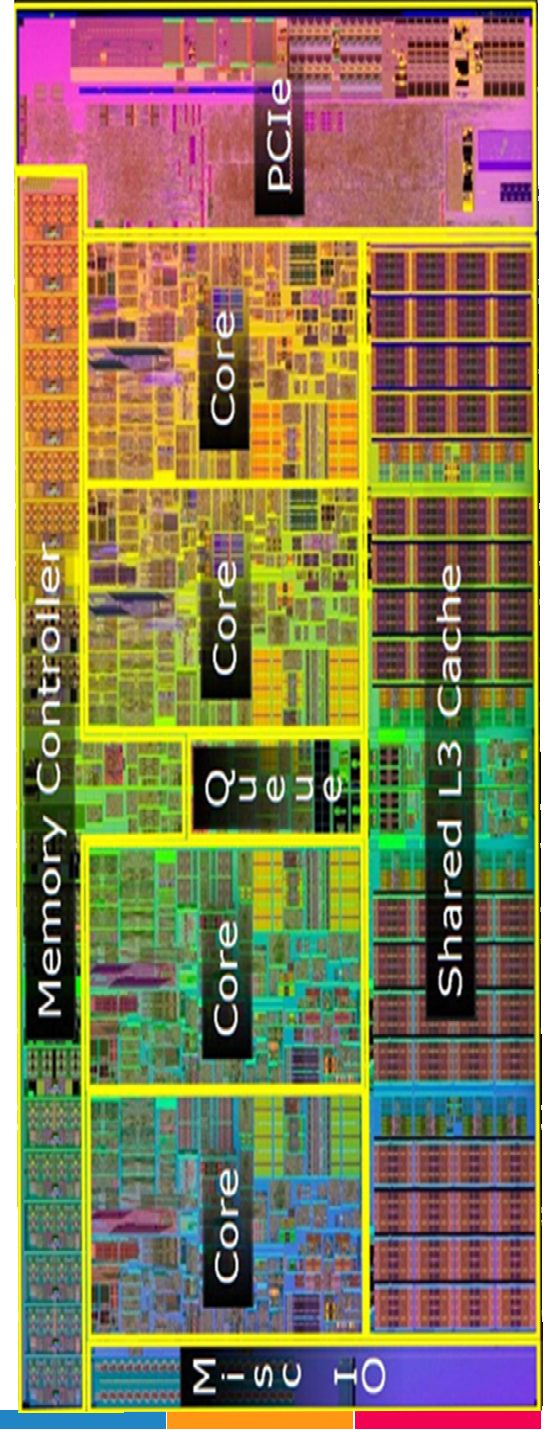


Clase de tiempo-real

- ▷ Si existe un proceso de tiempo-real ejecutable en el sistema este se ejecutará antes que el resto, salvo que haya otro de prioridad mayor.
- ▷ La cola de ejecución es simple, como puede verse en la Figura.
- ▷ El *mapa de bits* permite seleccionar la cola con procesos en 2 (64 bits) o 4 (32 bits) instrucciones en ensamblador.

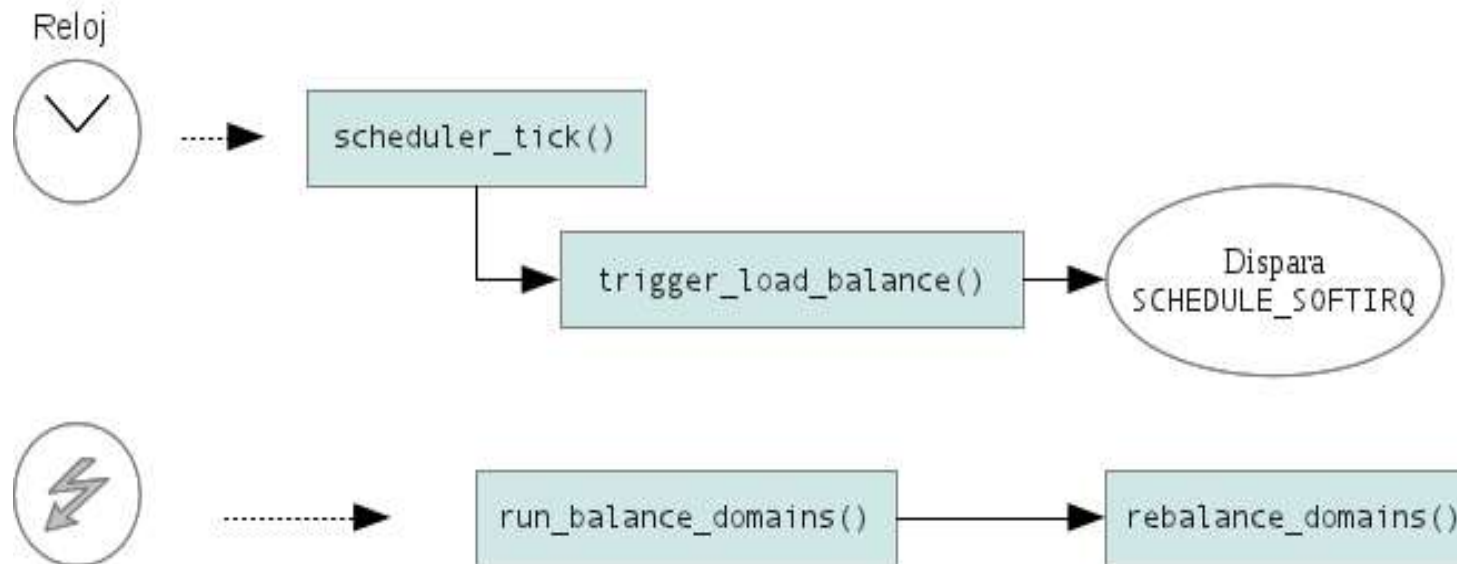


Planificación en multi-procesadores



Planificación SMP

- ▷ Aspectos considerados:
- Compartición de la carga de CPU imparcial
 - Afinidad de una tarea por un procesador
 - Migración de tareas sin sobrecarga



Gestión de la energía

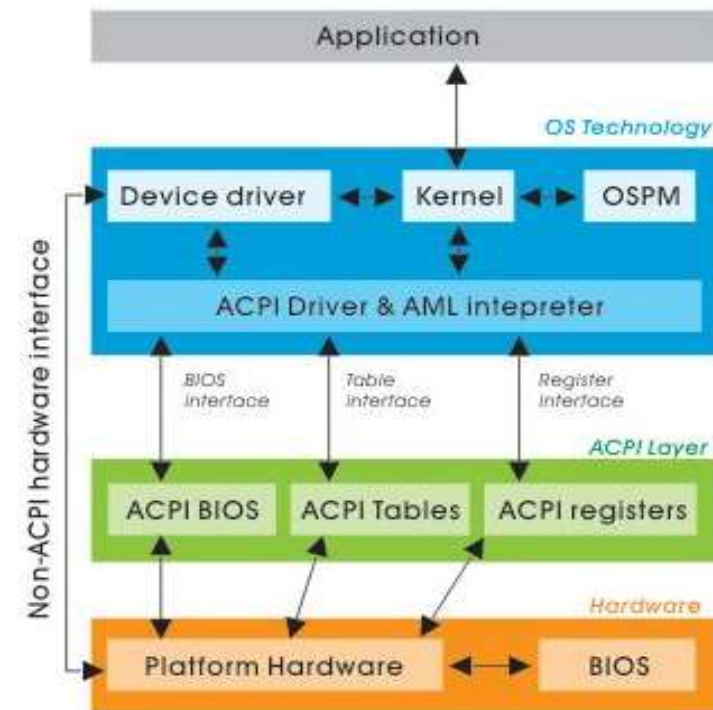


Gestión de la energía

- ▷ Un aspecto importante a considerar en los diseño actuales es la gestión de potencia, encaminada a mantener la potencia de cómputo reduciendo:
 - Los costes de consumo de energía
 - Los costes de refrigeración
- ▷ Esta gestión se realizar a varios niveles:
 - Nivel de CPU: P-states, C-states y T-states.
 - Nivel de SO: CPUfreq (paquetes cpufrequtils y cpupower) y planificación

Especificación ACPI

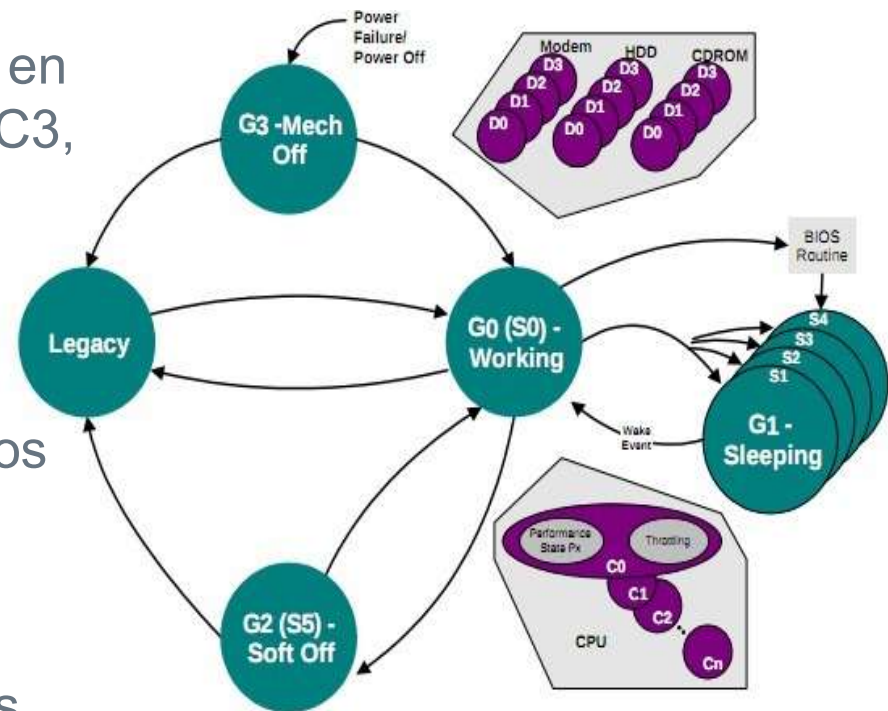
- ▷ **Advanced Configuration and Power Interface**: especificación abierta para la gestión de potencia y gestión térmica controladas por el SO.
- ▷ Desarrollada por Microsoft, Intel,
- ▷ HP, Phoenix, y Toshiba.
- ▷ Define cuatro estados Globales
- ▷ (G-estados):
 - G0: estado de funcionamiento: estados-C y estados-P
 - G1: estado dormido – S-estados
 - G2: Estado apagado soft
 - G3: Estado apagado mecánico



Techarp, *PC Power Management Guide Rev. 2.0*, disponible en <http://www.techarp.com/showarticle.aspx?artno=420>

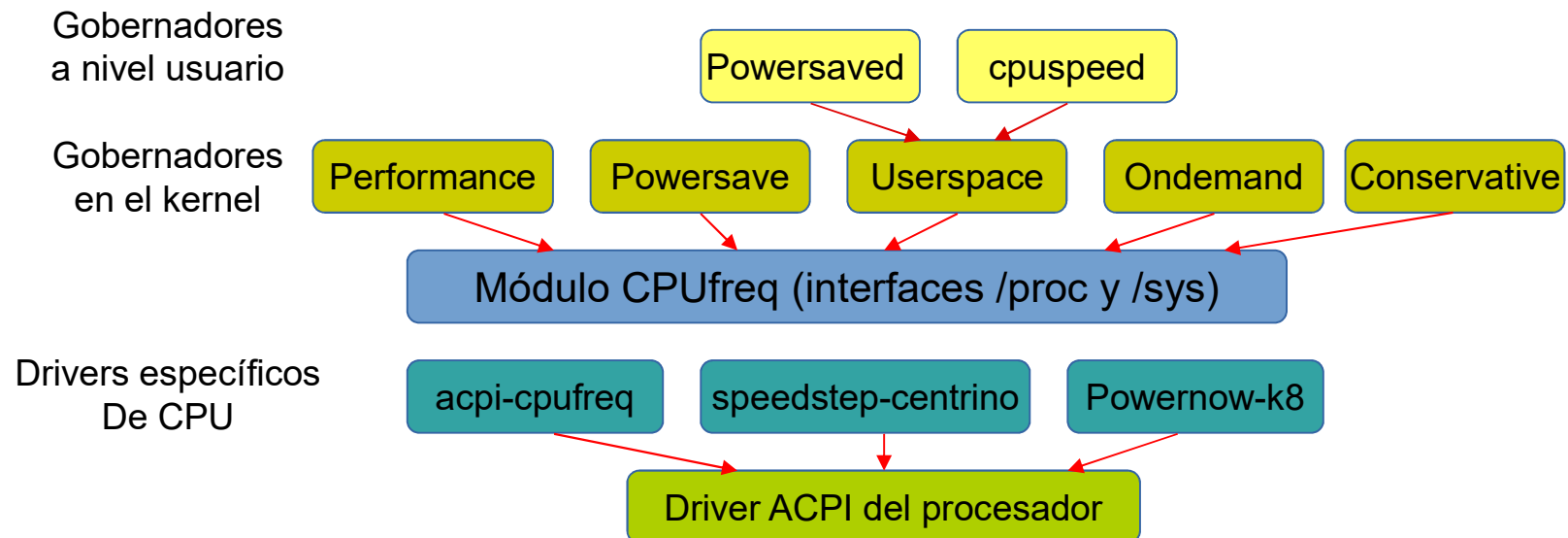
Estados de la CPU

- ▷ **S-estados:** estados dormidos en G1. Van de S1 a S5.
- ▷ **C-estados:** estados de potencia en G0. C0:activo, C1:halt, C2:stop, C3, deep sleep,...
- ▷ **P-estados:** relacionados con el control de la frecuencia y voltaje del procesador. Se usan con G0 y C0. P1-Pn, a mayor n menos freq y volt.
- ▷ **T-estados:** estados acelerados (*throttles*) relacionados con la gestión térmica. Introducen ciclos ociosos.



Estructura CPUfreq

- ▷ El **subsistema CPUfreq** es el responsable de ajustar explícitamente la frecuencia del procesador.
- ▷ Estructura modularizada que separa políticas (gobernadores) de mecanismos (*drivers* específicos de CPUs).



Gobernadores

- ▷ **Performace** – mantiene la CPU a la máxima frecuencia posible dentro un rango especificado por el usuario.
- ▷ **Powersave** – mantiene la CPU a la menor frecuencia posible dentro del rango.
- ▷ **Userspace** – exporta la información disponible de frecuencia a nivel de usuario (sysfs) permitiendo su control al mismo.
- ▷ **On-demand** – ajusta la frecuencia dependiendo del uso actual de la CPU.
- ▷ **Conservative** – Como 'ondemand' pero ajuste más gradual (menos agresivo).
- ▷ Podemos ver el gobernador por defecto en:
`/sys/devices/system/cpu/cpuX/cpufreq/scaling_governor`

Herramientas

- ▷ **Cpufrequtils** – podemos ver, modificar los ajustes del kernel relativos al subsistema CPUfreq. Las órdenes cpufreq* son útiles para modificar los estados-P, especialmente escalado de frecuencia y gobernadores.
- ▷ **Cpupower** – ver todos los parámetros relativos a potencia de todas las CPUs, incluidos los estados-turbo. Engloba a la anterior.
- ▷ **PowerTOP** (<https://01.org/powertop>)– ayuda a identificar las razones de un consumo alto innecesario, por ejemplo, procesos que despiertan al procesador del estado ocioso.
- ▷ Se pueden crear perfiles en */etc/pm-profiler*.
- ▷ **TLP** (linrunner.de/en/tlp/tlp.html) herramienta para gestionar energía de forma avanzada.

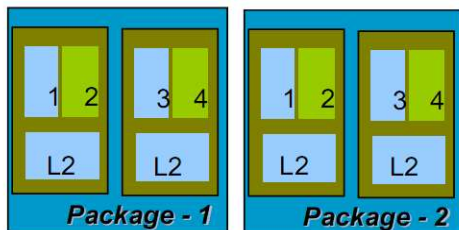
Planificación y energía

- ▷ En CMP con recursos compartidos entre núcleos de un paquete físico, el rendimiento máximo se obtiene cuando el planificador distribuye la carga equitativamente entre todos los paquetes.
- ▷ En CMP sin recursos compartidos entre núcleos de un mismo paquete físico, se ahorrará energía sin afectar al rendimiento si el planificador distribuye primero la carga entre núcleos de un paquete, antes de buscar paquetes vacíos.

Algoritmos de planificación y energía

- ▷ El administrador puede elegir el algoritmo de planificación modificando las entradas *sched_mc_power_saving* y *sched_smt_power_saving*:

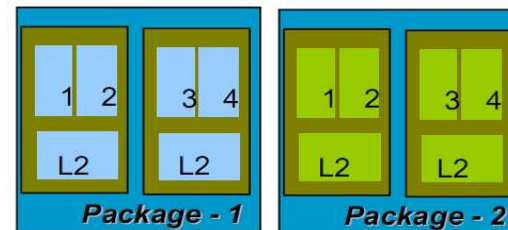
Rendimiento óptimo



Non Idle Idle

```
$ cpupower set -m 0
```

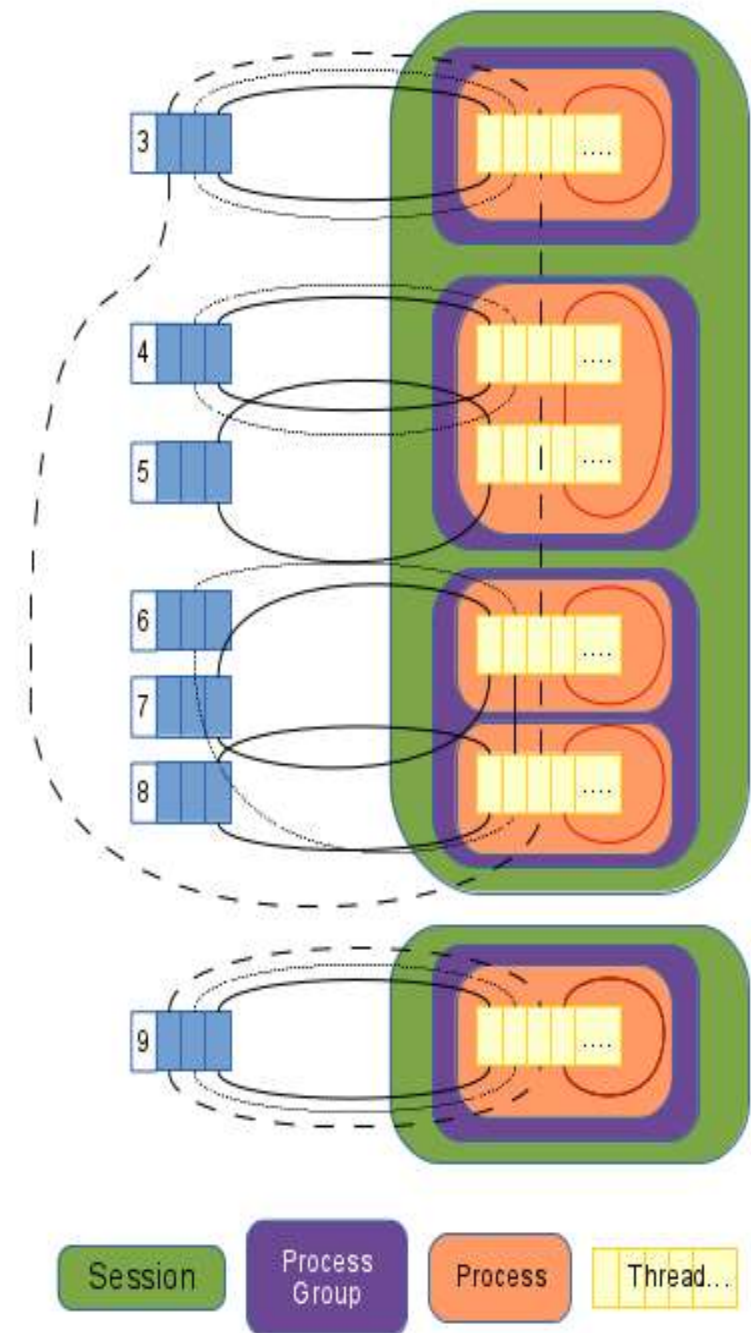
Ahorro de energía



Non Idle Idle

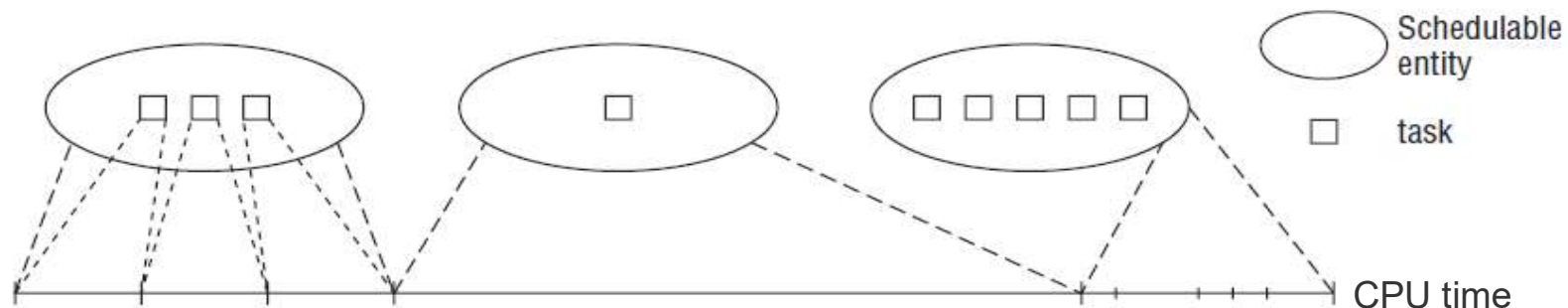
```
$ cpupower set -m 1
```

Grupos de control



Grupos de control

- ▷ El planificador trata con **entidades planificables**, que no tienen por que ser procesos.
- ▷ Esto permite definir grupos de planificación – Diferentes procesos se asignan a diferentes grupos. El planificador reparte la CPU imparcialmente entre grupos, y luego entre proceso de un grupo. Esto reparte imparcialmente la CPU entre usuarios.



Grupos de control: usos

- ▷ Suministran un mecanismo para:
 - Asignar/limitar/priorizar recursos: CPU, memoria, y dispositivos.
 - Contabilidad: medir el uso de recursos.
 - Aislamiento: espacios de nombres separados por grupo.
 - Control: congelar grupos o hacer checkpoint/restart.
- ▷ Los *cgroups* son jerárquicos: un grupo hereda los límites del grupo padre.

Subsistemas cgroups

- ▷ Existen diferentes subsistemas (controladores de recursos):
 - **cpu**: usado por el planificador para suministrar el acceso de las tareas de un *cgroup* a la CPU.
 - **cpuacct**: genera automáticamente informes de la CPU utilizada por las tareas de un cgroup.
 - **cpuset**: asigna CPUs individuales y memoria en sistemas multicore.
 - **devices**: permite/deniega el acceso de las tareas a un dispositivo.
 - **freezer**: detiene la ejecución de todos los procesos de un grupo.
 - **memory**: limita el uso de memoria a tareas de un cgroup, y genera informes automáticos del uso de memoria de esas tarea.
 - **blkio**: establece los límites de accesos de E/S desde/hacia dispositivos de bloques (discos, USB, ...)
 - **net_cls**: etiqueta paquetes de red con un identificador de clase (*classid*) que permite al controlador de tráfico (*tc*) identificar los paquetes originados en una tarea de un grupo.
 - **ns**: subsistemas de espacios de nombres (namespaces).

Uso de los cgroups

- ▷ Pueden utilizarse de diferentes modos:
 - *Seudo-sistema de archivos cgroups* (cgroupfs).
 - Herramientas de *libcgroup*: cgcreate, cgexec, cgclassify, ...
 - El demonio *engine rules* los gestiona según la información de los archivos de configuración.
 - Indirectamente a través de otros mecanismos como Linux Containers (LXC), libvirt, systemd.

Método 1: cgroupsFS

- ▷ Los subsistemas se habilitan como una opción de montaje de cgroupfs:
`mount -t cgroup -o$subsistema`
- ▷ Habilitamos los archivos del subsistema en cada cgroup (directorio):
`/dev/cgroup/migrupo/subsysA.optionB`
- ▷ Podemos verlos en */proc/cgroups*.
- ▷ En Ubuntu, podemos instalarlo con
`$ sudo aptitude install cgroups-bin libcgroup1`
- ▷ Esto nos monta por defecto los siguientes fs:
`$ ls -l /sys/fs/cgroup`
`cpu cpuacct devices memory`
- ▷ Podemos ver los grupos de control con `cat /proc/cgroups`

CgroupsFS: ejemplo

- ▷ Crear dos grupos para asignar tiempo de CPU:
 - Creamos el correspondiente subdirectorio en *sys/fs/cgroup/cpu*:

```
$ mkdir Navegadores; mkdir multimedia
```
 - Asignamos el porcentaje de cpu al grupo escribiendo en el archivo *cpu.shares*:

```
$ echo 2048 > /sys/fs/cgroup/cpu/multimedia/cpu.shares
```

```
$ echo 1024 > /sys/fs/cgroup/cpu/multimedia/cpu.shares
```
 - Movemos una tarea al cgrupo escribiendo su PID en el archivo *tasks*.

```
$ firefox&
```

```
$ echo $! > /sys/fs/cgroup/cpu/navegadores/tasks
```

```
$ mplayer micancion.mp3&
```

```
$ echo $! > /sys/fs/cgroup/cpu/multimedia/tasks
```