Proceso nulo

> ¿Qué ocurre si el planificador a corto plazo no encuentra procesos preparados para ejecutarse cuando es invocado?

> La solución optima es introducir el proceso nulo, es decir un proceso que:

- siempre este listo para ejecutarse.
- tenga la prioridad más baja.

Proceso nulo: implementación

> 1ª solución:

```
Planificador() {
while (true) {
  if
(cola_preparados==vacia)
       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”
  - Nenor prioridad del sistema (no compite con el resto por la CPU)

Linux y la apropiatividad

> Los kernel actuales son apropativos y permiten ajustar el grano de apropiativdad 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.

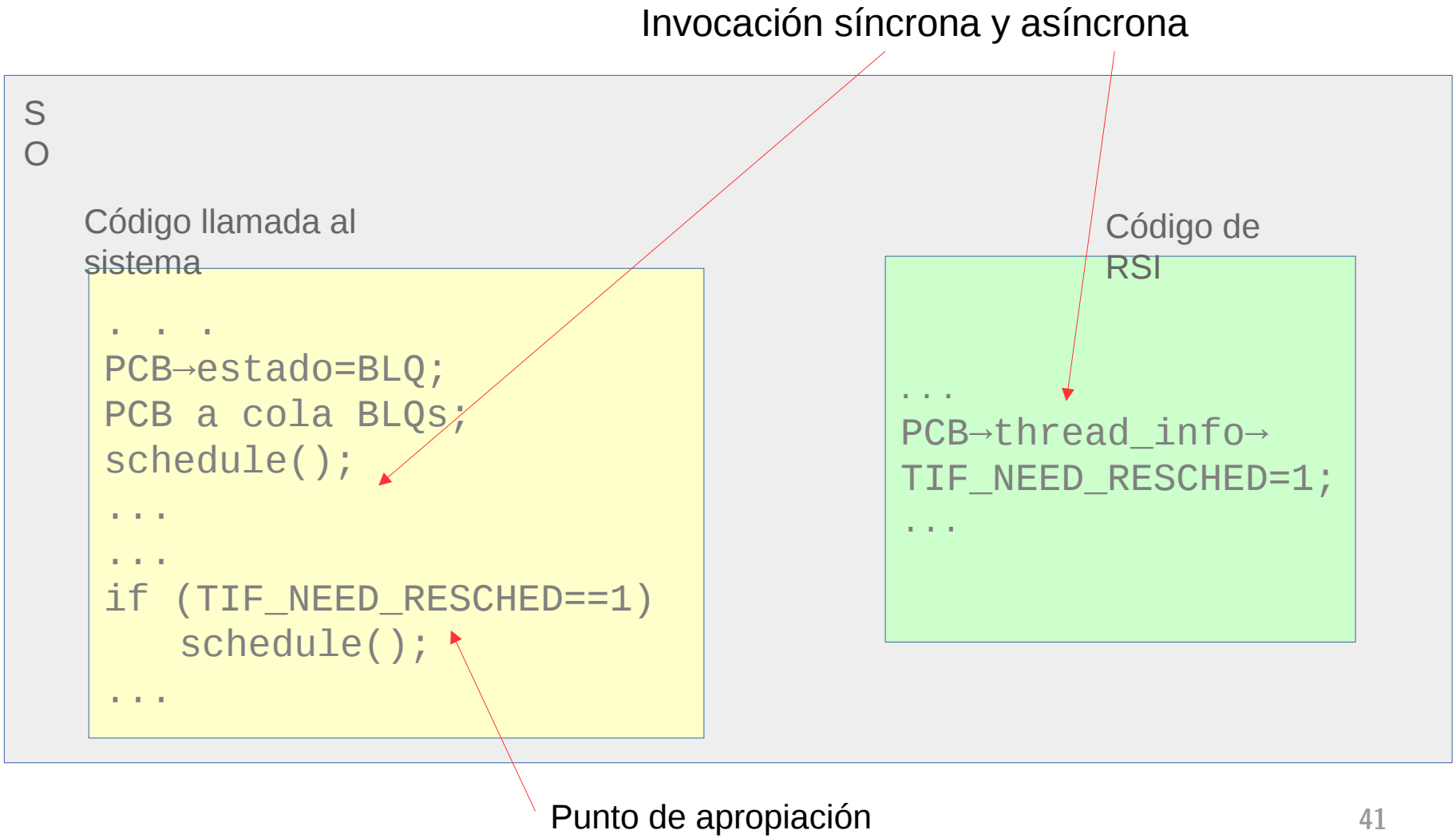
> Dado que la invocación asíncrona del planificador podría generar condiciones de carrera, esta se difiere hasta alcanzar un punto de apropiación.

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



Trabajo en grupo 2.2

> Supongamos que ejecutamos un kernel que utiliza planificación NO apropiativa (linux <2.6): ¿Pueden 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

> En RTOS el SO debe planificar las taréas 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 cede la CPU.
. Componente asíncrono: se activa cada cierto tiempo.

Clases de planificación

> Un SO conforme POSIX1.b debe soportar al menos tres clases de planificación:

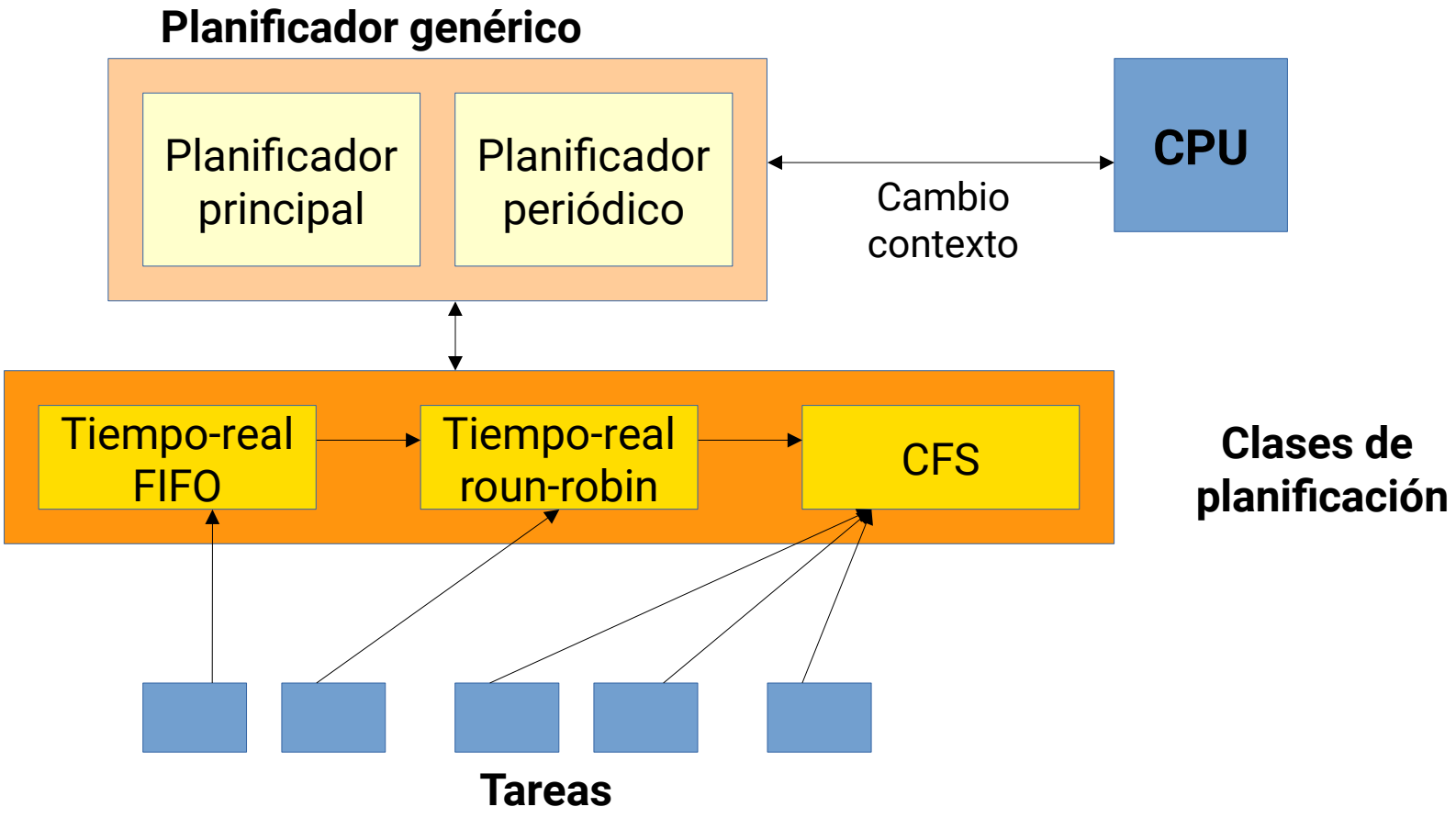
- Dos de tiempo-real: SCHED_FIFO y SCHED_RR
- Una de tiempo compartido: SCHED_OTHERS

> Linux soporta estas tres clases.

SCHED_OTHERS se denomina SCHED_NORMAL. Que a su vez contiene las sub-clases SCHED_BATCH y SCHED_IDLE.

> El planificador soporta maquinas NUMA, UMA, multicores y hiperhilos.

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 apropian a otro proceso gestionado por CFS y por interfieren con trabajos interactivos. Es aconsejable en situaciones en las que no se desea decrementar la prioridad estática con nice, pero la tarea no debería influenciar la interactividad del sistema.
  - `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:
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

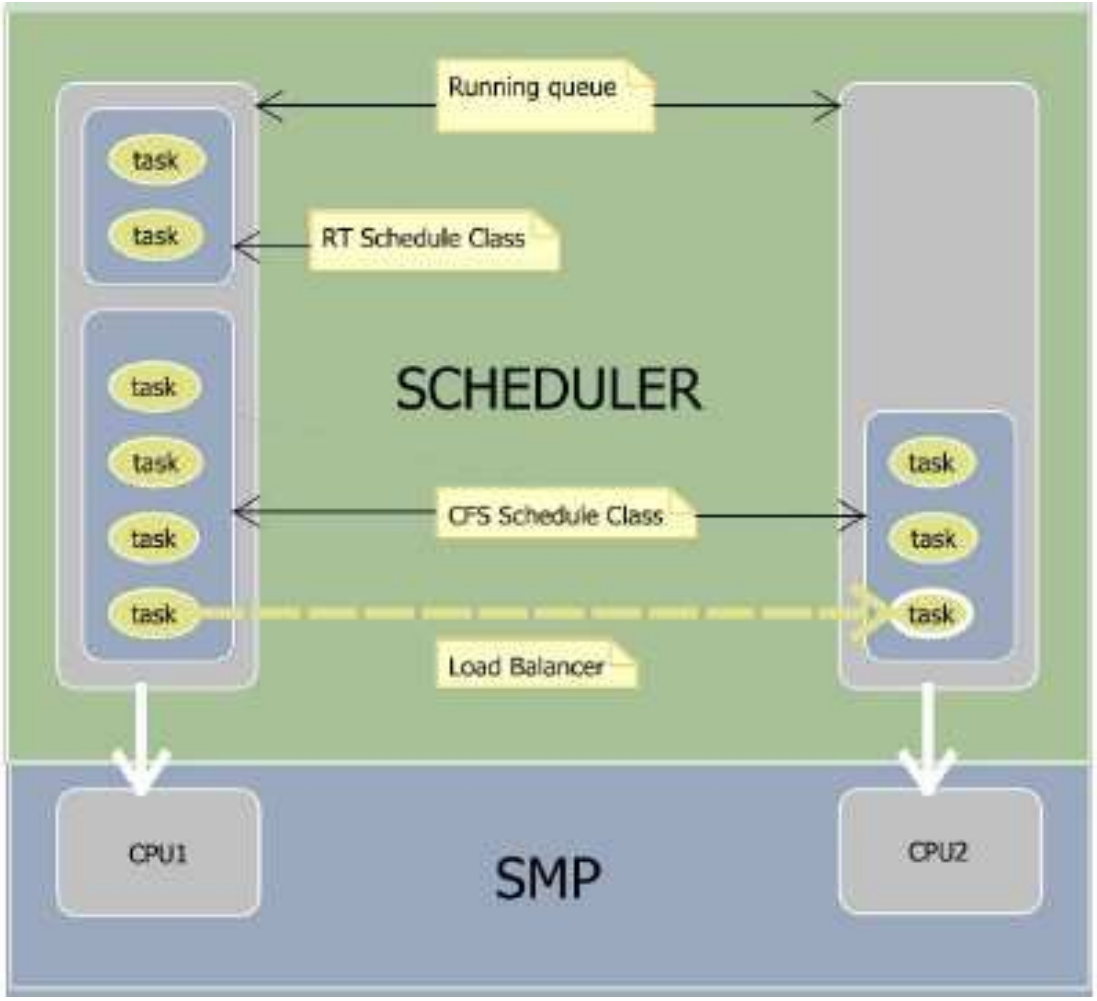
```
struct sched_class {
      const struct sched_class *next;
      void (*enqueue_task();      /* añade proceso a la cola de
                                     ejecución rq, y nr_running++*/
      void (*dequeue_task();      /*lo elimina de rq,y nr_running--*/
      void (*yield_task) ();      /* 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; /* nº 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

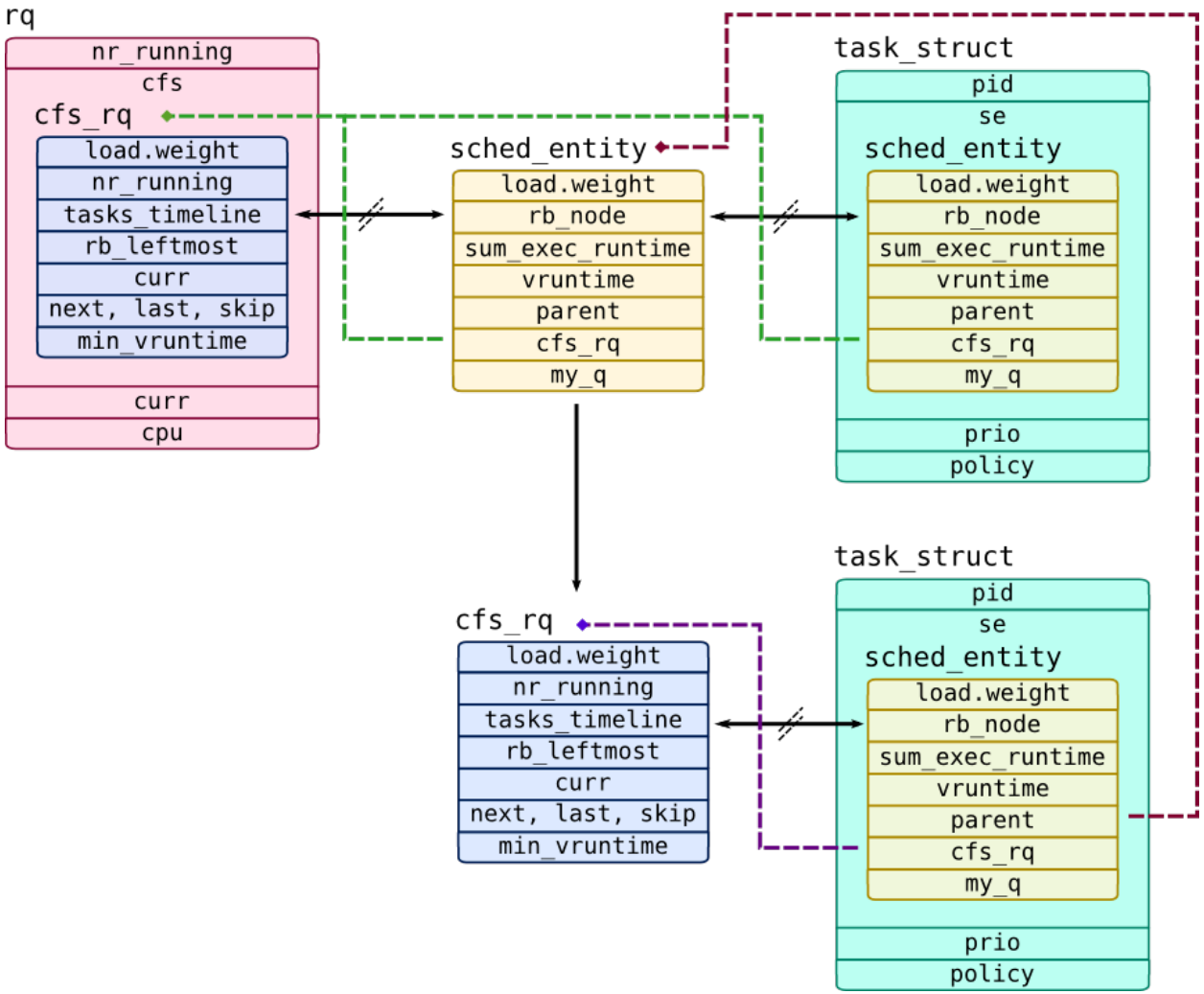


Entidades de planificación

> La estructura que describe una entidad:

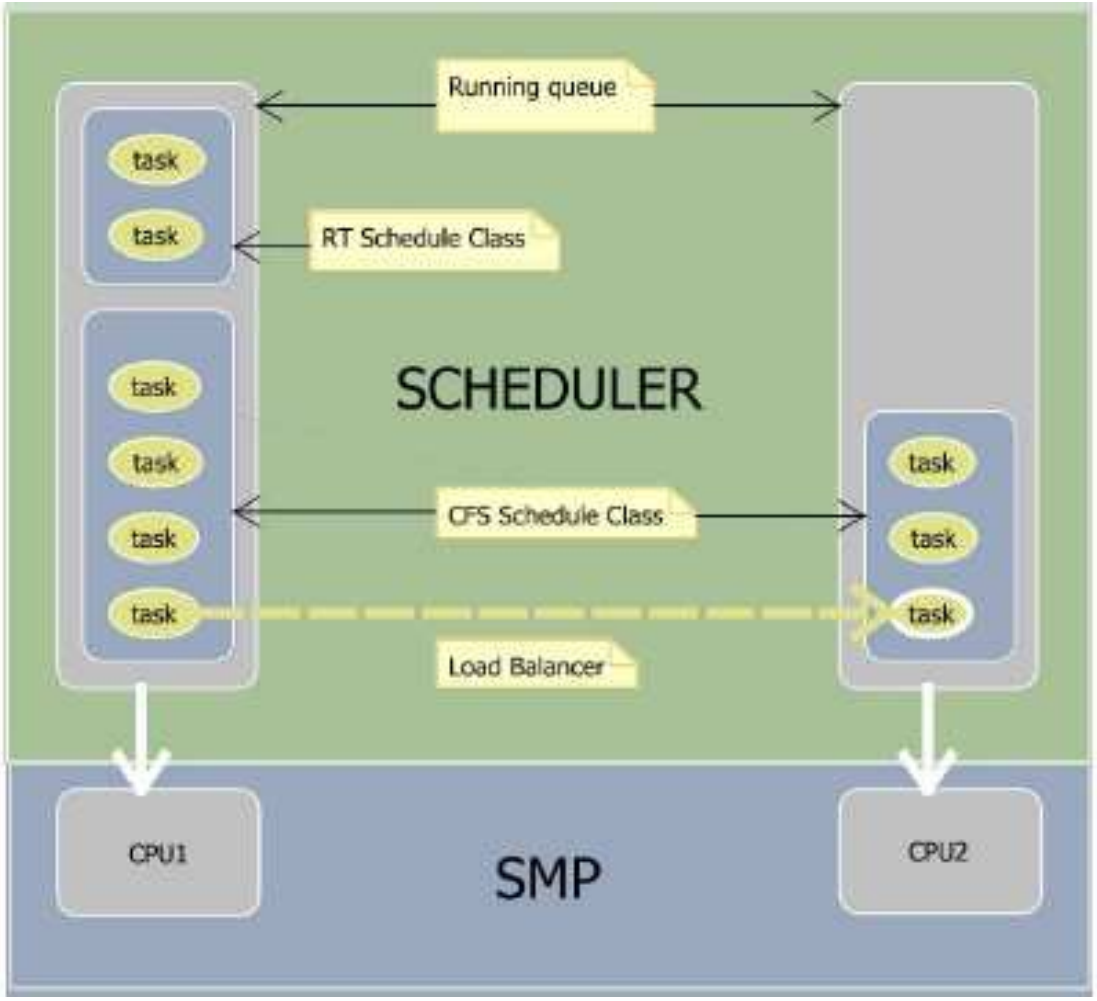
```
struct sched_entity {
  struct load_weight load;    /*para equilibrio de carga*/
  struct rb_node run_node;    /*nodo de arbol rojo-negro*/
  unsigned int on_rq;         /*indica si la entidad esta
                                planificada en una cola*/
  u64 exec_start;            /* tiempo inicio ejecución */
  u64 sum_exec_start;        /*t consumido de CPU*/
  u64 vrtime;                /*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

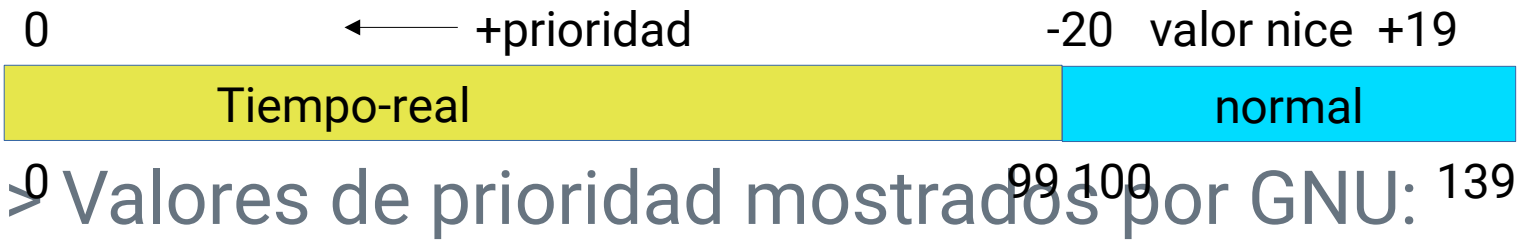
Colas de ejecución



http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main/

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:
`-l` muestra el valor `intprio`
`-l -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.

```
desactivate_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;
```

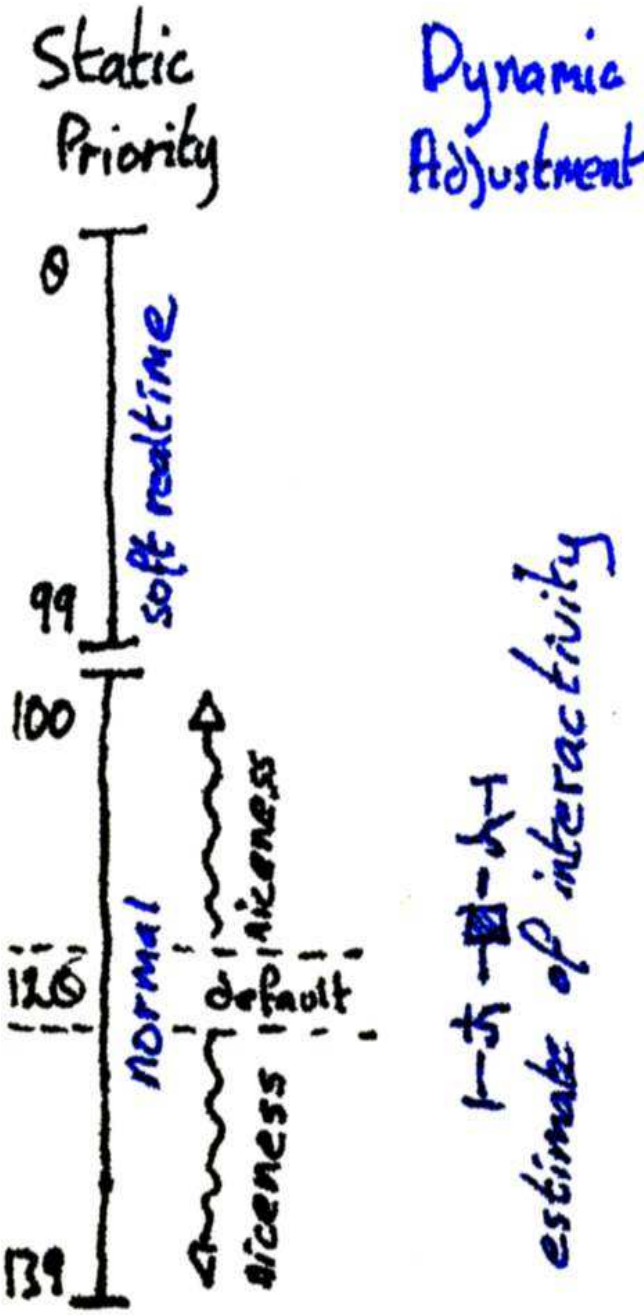
Cambio de contexto

> 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



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 ≈ 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 vi ene dado por la prioridad, o por su **peso de carga** (*load weight*).
> Para ello, el kernel convierte prioridades de peso 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 (*Completely Fair Scheduler*) 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.

$$TPi=(PesoPi/\sum PesosPj)* P$$

con $p=sched\_latency$ si $n>nr\_latency$
ó $min\_granularidad*n$, en otro caso

Ya que si $n \rightarrow \infty$, $TPi \rightarrow 0$, se define una granularidad mínima (suelo de tiempo asignado). En la implementación actual: $sched\_latency=8$, $nr\_latency=8$ y $min\_granularity= 1$ us.

Tiempo asignado a un proceso

> Como la carga es dinámica, para el cálculo tiempo asignado se usa un **periodo**:

- 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

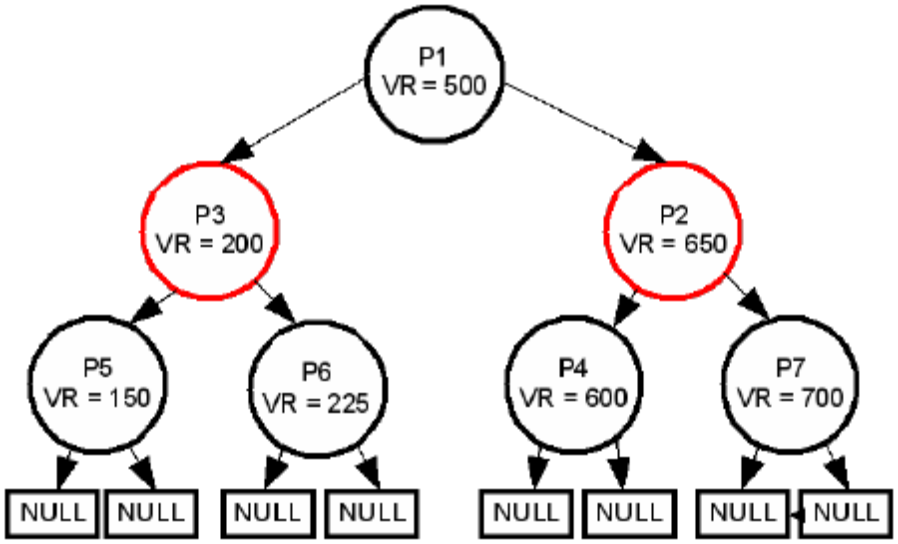
> ***Tiempo virtual de ejecución*** (*vruntime*)= tiempo de ejecución real normalizado por el número de procesos en ejecución. Este tiempo es gestionado por `update_curr()` definida en *kernel/sched_fair.c*:

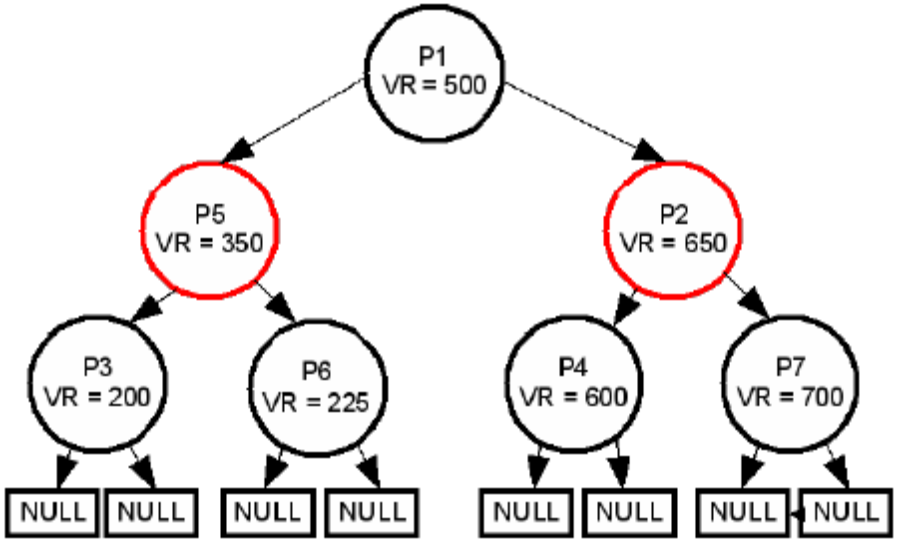
```
vruntime =(PesoNice0/Peso cola)x tiempo_real
         = tiempo_ejecucion_actual* 1024/peso rq
```

> 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.





Extraído de Ktown Coder, “Process Scheduling In Linux” en www.countrycoders.com/comp314.php

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
```

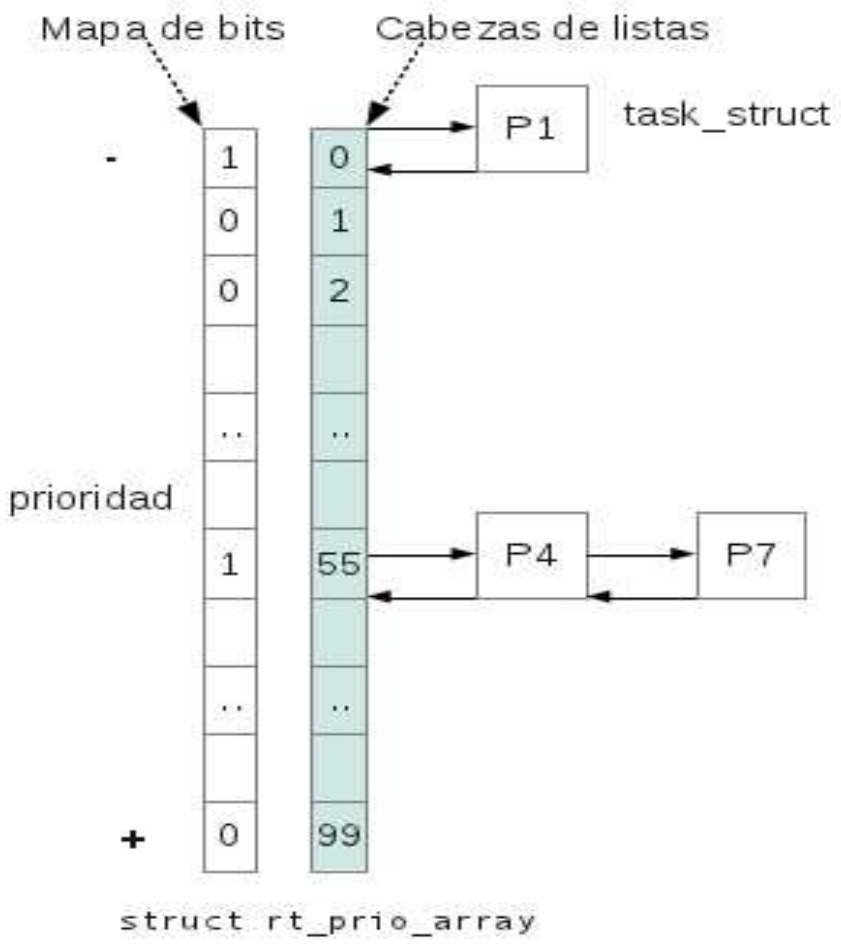
> Información planificación proceso PID:

```
/proc/<PID>/sched
```

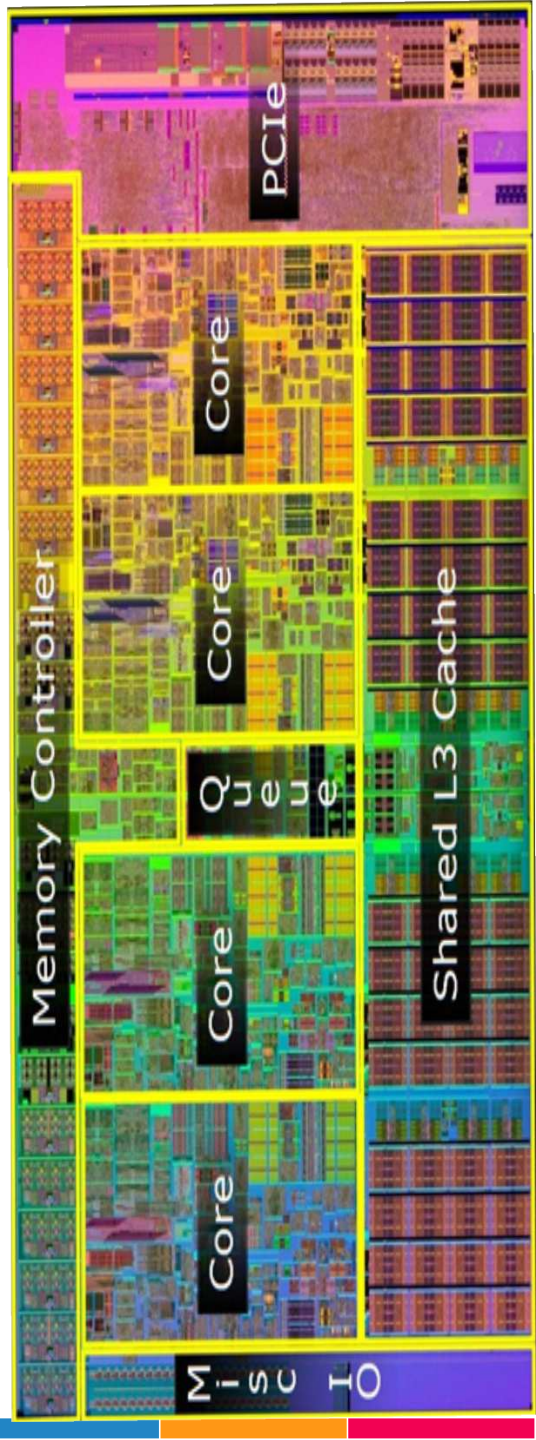
Planificador de tiempo-real

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.

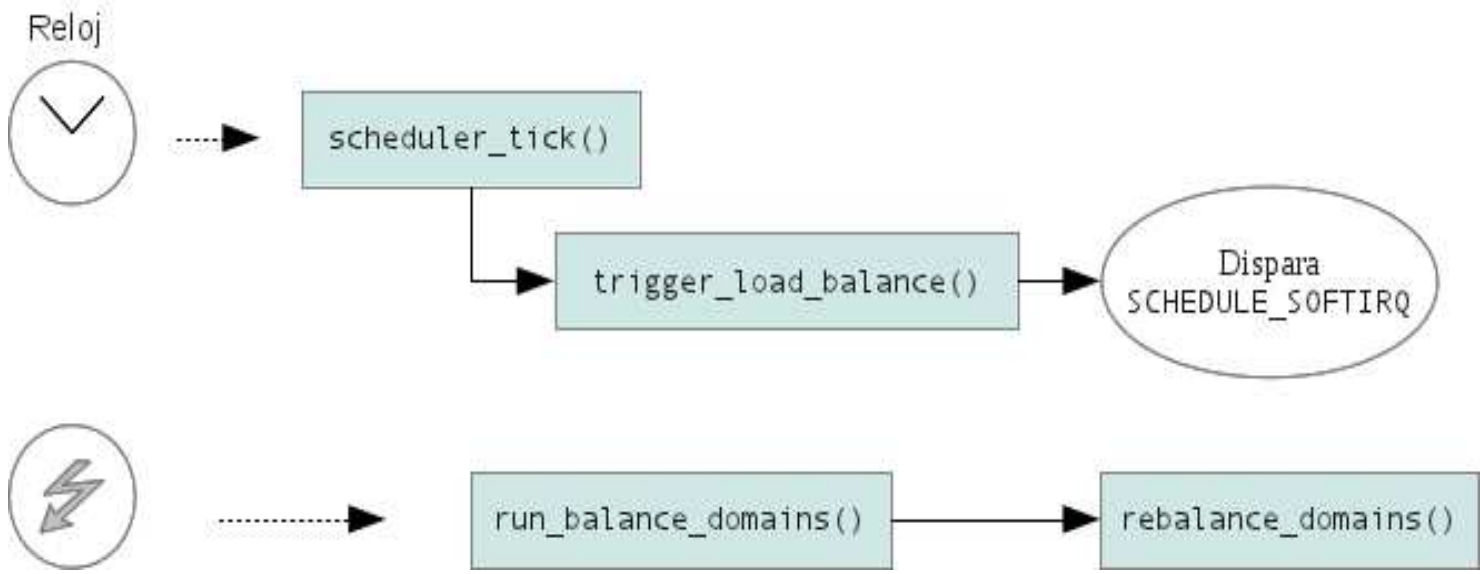


Planificador en sistemas multiprocesadores



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

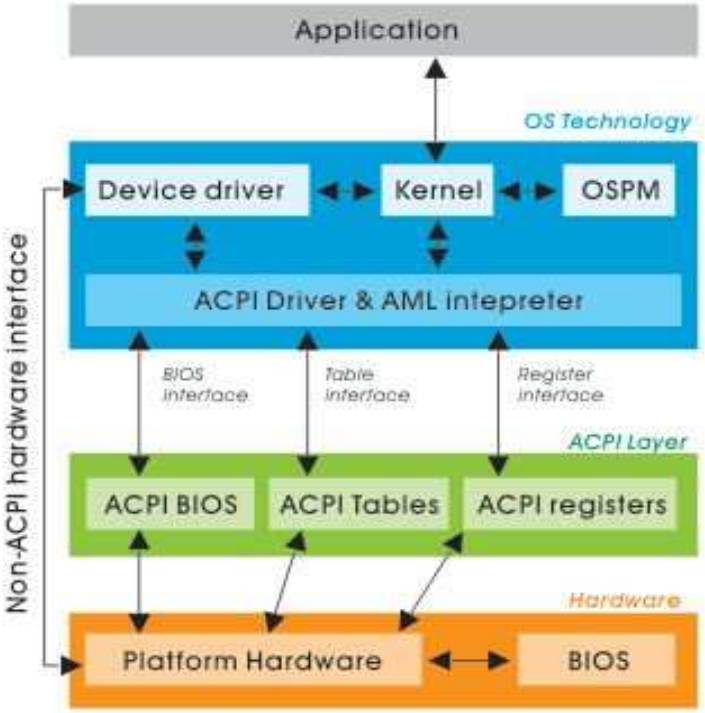
> Esta gestión se realizar a varios niveles:

- 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 p 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):

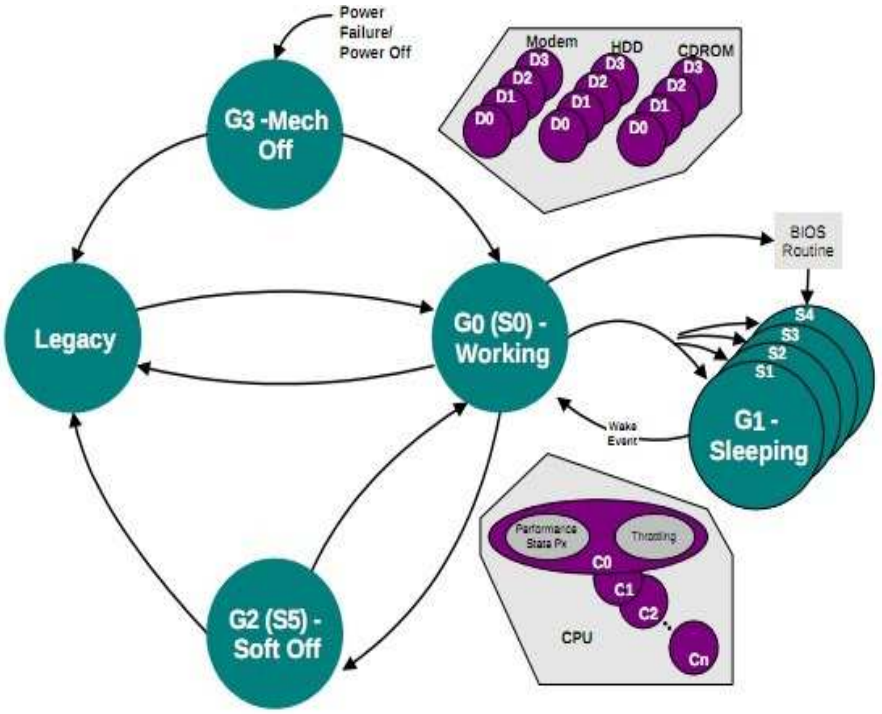
- G0: estado de funcionamiento: estados-C y estados-P
- G1: estado dormido – S-estado
- G2: Estado apagado soft
- G3: Estado apagado mecánico



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

Estados de la CPU

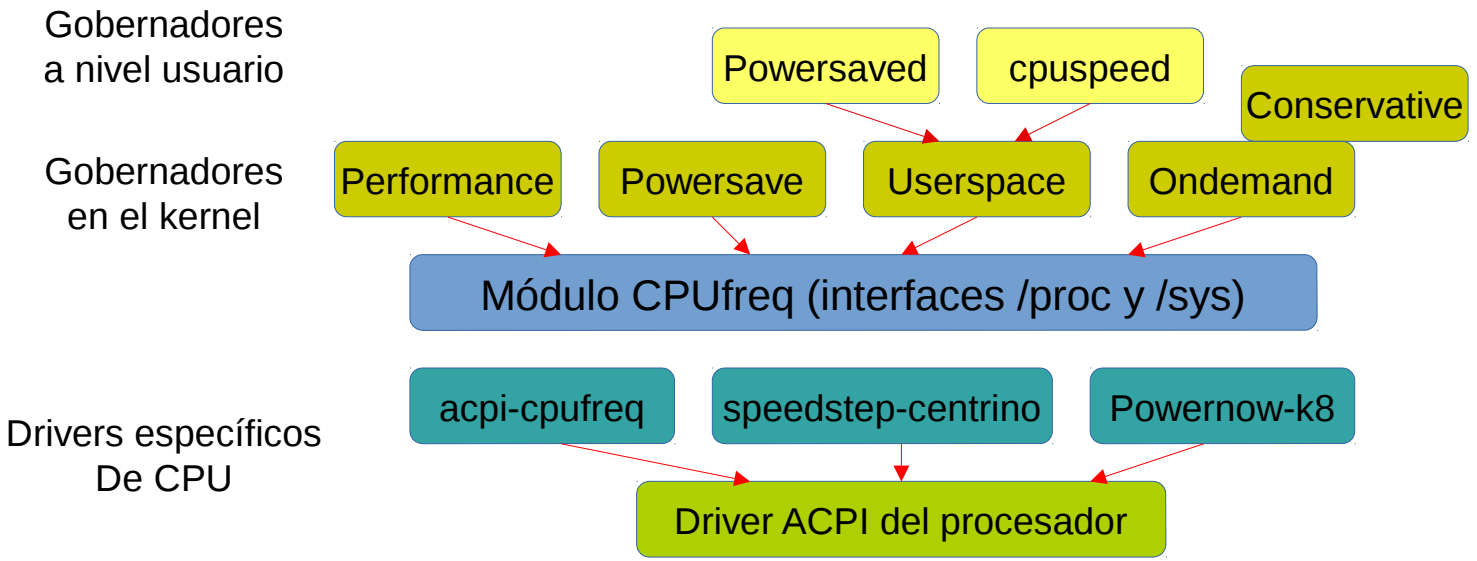
> **S-estados**: estados dormidos en G1. Van de S1 a S5.
> **C-estados**: estados de potenc en G0. C0:activo, C1:halt, C2:stc C3, deep sleep,...
> **P-estados**: relacionados con control de la frecuencia y voltaje del procesador.
Se usan con G0 y C0. P1-Pn, a mayor *n* menos freq y volt.
> **T-estados**: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



ACPI spec v5.0, Dic. 2011

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 utiles 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.

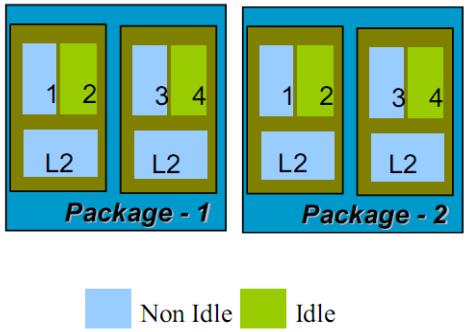
Planificación y energía

> 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 lo 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

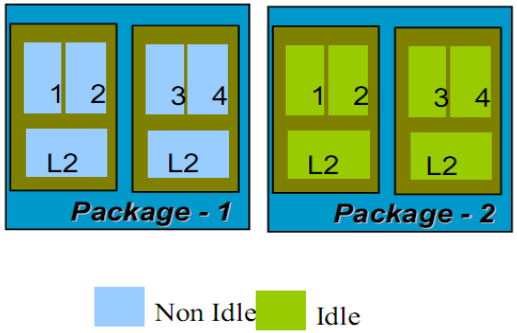
> El administrador puede elegir el algoritmo de planificación moficando las entradas *sched_mc_power_saving* y *sched_smt_power_saving:*

Rendimiento óptimo



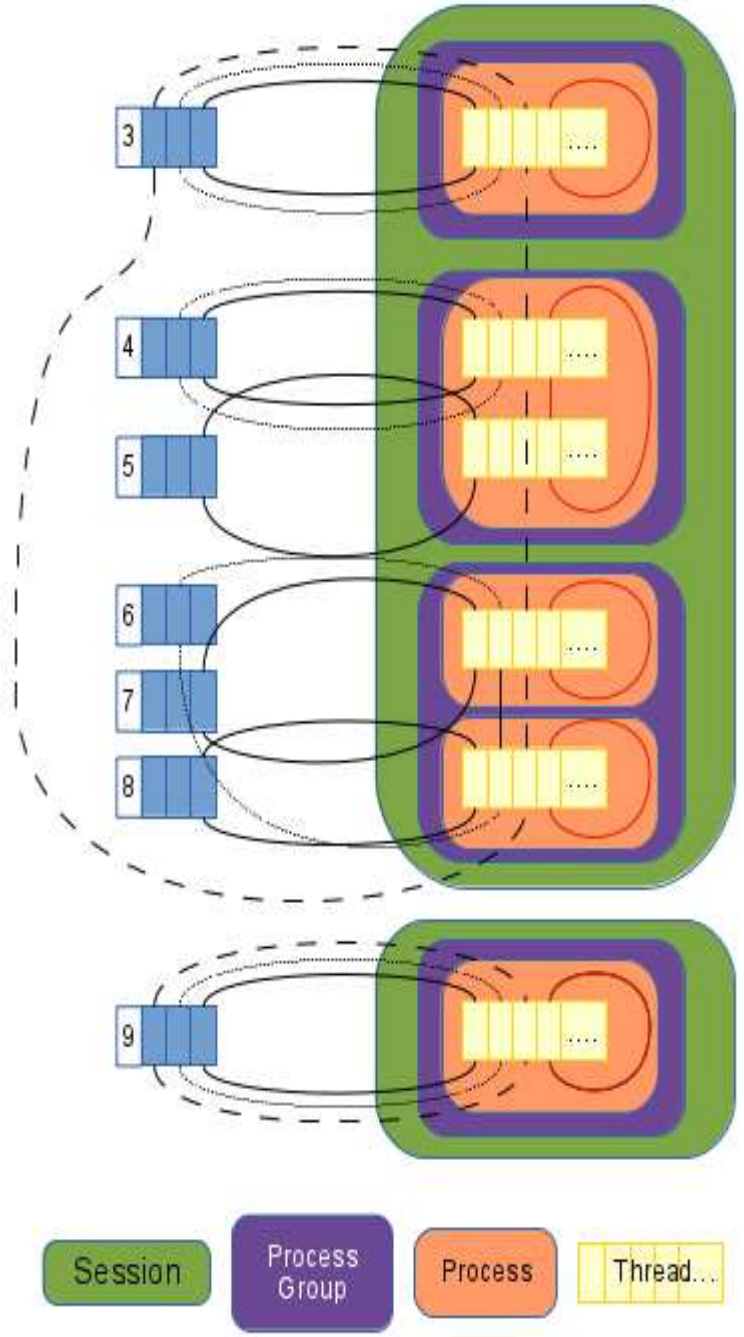

```
$ cpupower set -m 0
```

Ahorro de energía



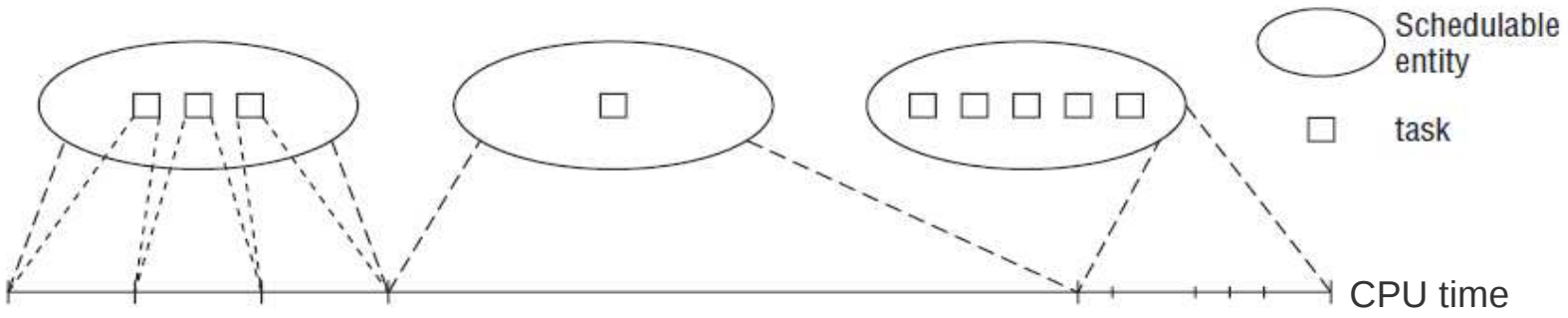

```
$ cpupower set -m 1
```

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

> uministran 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):

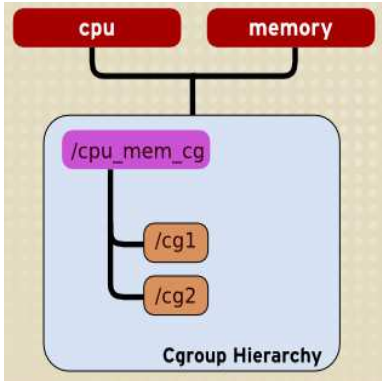
- **cpu**: utilizado por el planificador para suministrar el acceso de las tareas de un cgroup a la CPU.
- **cpuacct**: genera automáticamente informes de la CPU utilizada por las tareas de un cgroup.
- **cpuset**: asigna CPUs individuales y memoria en sistemas multicore.
- **devices**: permite/deniega 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**:_subsitemas de espacios de nombres (namespaces).

Reglas de uso

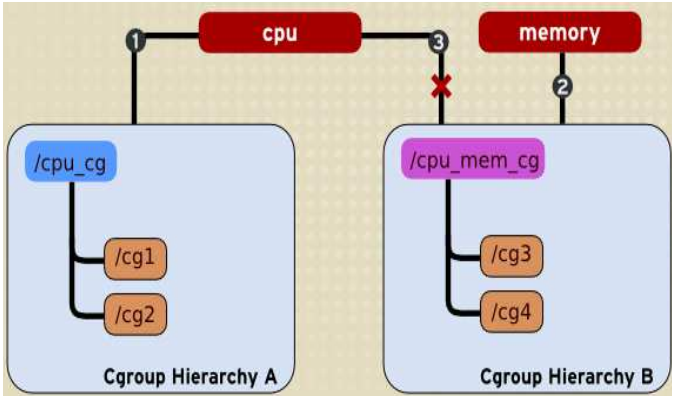
> *Regla 1*: Una única jerarquía puede tener uno o más subsistemas ligados.
> *Regla 2*: Un subsistema ligado a una jerarquía A no puede estar ligado a una jerarquía B si la B tiene un subsistema diferente ligado a ella.
> *Regla 3*: Una tarea no puede ser miembro de dos cgroups diferentes en la misma jerarquía.
> *Regla 4*: Una tarea hija hereda los mismos cgroups que su padre.

Reglas de uso: ilustración

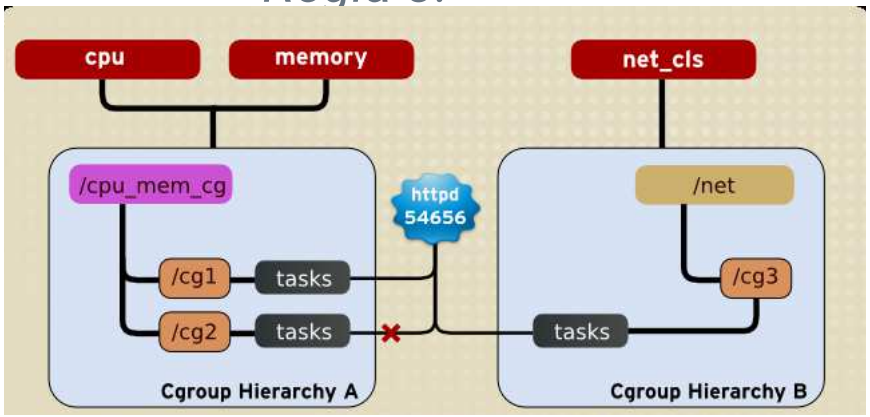
Regla 1:



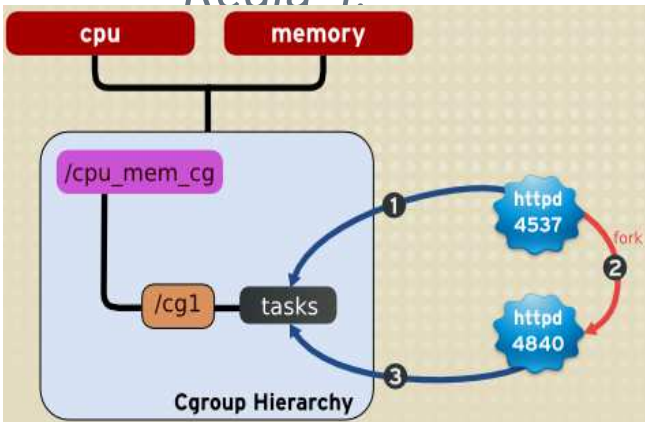
Regla 2:



Regla 3:



Regla 4:



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
```