

TEMA 2

task-struct: Define proceso o hilo. Contiene todo lo que necesita el SO para describir una tarea.

Estado de un proceso (que operaciones puedo hacer sobre un proceso), recursos del sistema que estan utilizando el proceso
files-struct: estructura de archivos que estan utilizando el proceso.
(files-struct: siempre hace falta porque todo proceso al menos tiene tres archivos abiertos (entrada, salida y error standard))

0,1,2

ffs-struct: almacena cual es el directorio de trabajo en cada momento

tty-struct: dispositivo de entrada/salida asociado

mm-struct: descripción del espacio de direcciones que estan utilizando el proceso
(de memoria que estan utilizando)

thread-info: da información de bajo nivel del procesador no tiene nada que ver con los hilos.

real-parent-parent: punteros, relaciones entre los procesos padre, hijos y hermanos
padre que adopta al proceso si el original muere

signal-struct: señales \Rightarrow eventos. Cuando el kernel tiene que notificar un evento manda una señal.

Ctrol + Z \rightarrow para el proceso (pasa al estado de parado)

Ctrl + C \rightarrow mata el proceso ambos generan señales.

de monios = hebras Kernel Linux) no necesita una descripción del espacio de usuario, se puede liberar el mm-struct del PCB.

Dividir el PCB en subestructuras tiene la ventaja de que se puede liberar la memoria asociada a la subestructura si no se va a necesitar (se pone el puntero a null) y crearlas cuando se necesiten.

Hilos: División de un proceso (subproceso) que permite tener varios hilos de control dentro de un mismo programa. Todos los hilos comparten la misma memoria (espacio de direcciones, mm-struct) \rightarrow de una misma tarea.

Cada nodo necesita una pila (en nodo Kernel necesita una pila Kernel y en nodo usuario una pila usuario). En Linux la pila Kernel se almacena junto con la estructura `thread_info`.

Funciones de `thread_info`: Permitir a una hebra saber en qué estado se está ejecutando. La tarea que se está ejecutando estar en current.

Salvaron la información de un proceso (registros hardware), seleccionar

el siguiente proceso, restaurarlos en la CPU los registros que tenía el proceso cuando se interrumpió.

Lo último que se almacena en la pila ^{Kernel} es el puntero de pila.

`Tif_Need_Resched`: bandera que se utiliza para llamar al planificador

`Tif_Sigpending`: señales que están pendientes de entrar en ese proceso.

Procesos intemperables por una señal: procesos que si están mucho tiempo bloqueados se les puede mandar una señal para matarlos (una entrada de teclado).

Inintemperables: no queremos despertarlos porque podrían matarlos por error (una lectura de disco) y que sabemos que van a darse de estar bloqueados.

Trasero: el proceso estar parado por alguna causa especial como en una depuración, en la que el proceso se bloquea en un punto de ruptura que indica el programador. Estar bloqueado de

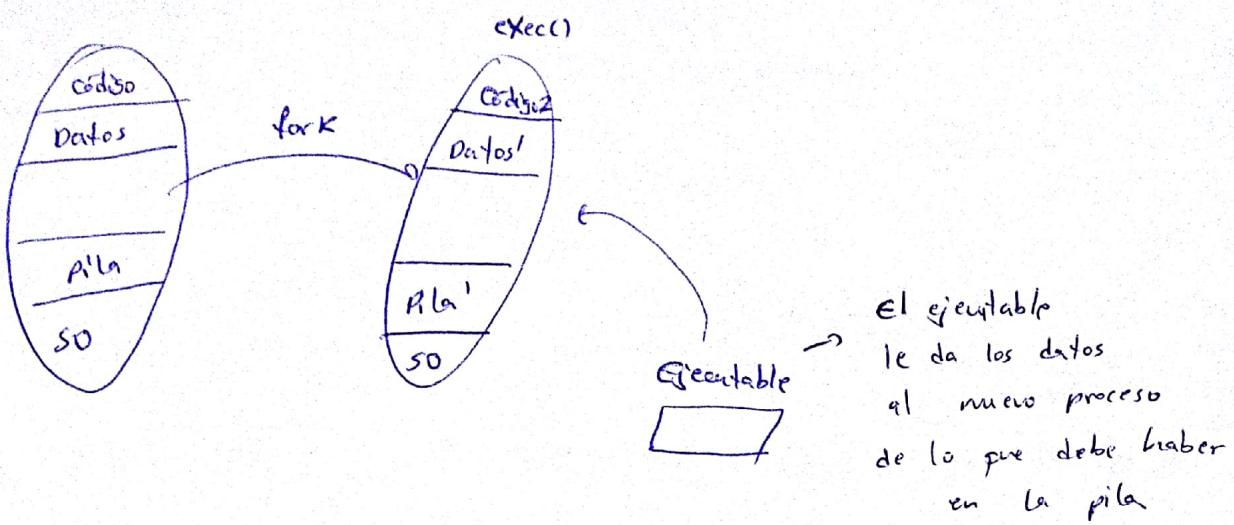
una forma especial. No todos los VIX soportan hebras.

→ UNIX solo en Linux
`fork` o `clone`: se crea un proceso/nodo nuevo.

`exit`: se finaliza la ejecución del proceso.

En Windows no hay jerarquía de procesos.

`fork + exec`: programa nuevo en un nuevo proceso. Se ve al ejecutable que estar en disco de un programa y lo ejecuta.



```

int *func void*: nombre de la función que quiero que ejecute mi
hilo

funcion hilo () {
}

main () {
}

crear-hilo (hilo, ... )
}
  
```

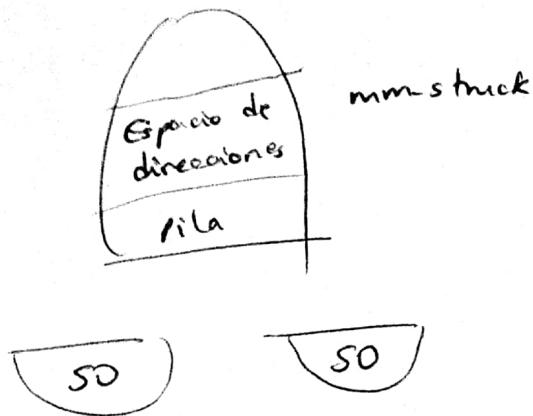
flags: dicen como es el nuevo proceso que quiero crear

1:1: un proceso o hilo a nivel de usuario estar soportado por un proceso/hilo a nivel Kernel, todos los hilos de usuario son visibles por el kernel y puede planificarlos independientemente (so)

Cuando se crea un proceso/hilo a nivel de usuario se crea otro a nivel de Kernel

) CLONE-FILES, CLONE-THREAD

Cuando padre/hijo comparten mm_struct tienen el mismo espacio, de nombres de usuario y se ejecutan en el mismo espacio



Ejemplo : NULL al final para decir que no necesitamos darle argumentos.

TID → Threat identification → En las hebras pid es el identificador del
getpid → muestra el pid grupo y TID el del proceso (la hebra)

syscall : le damos el nombre de la llanada que queremos hacer (la función)

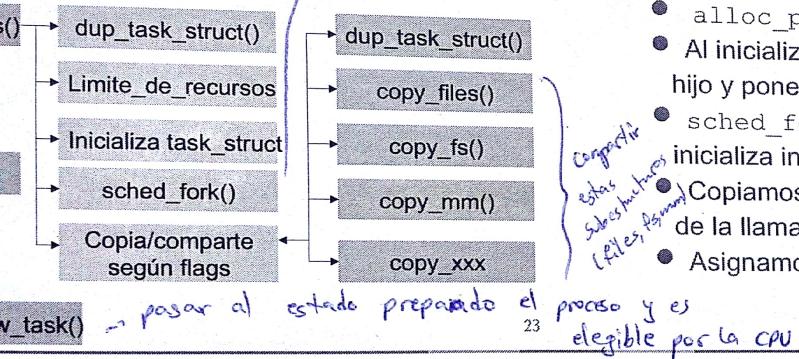
Si le quitamos VH la memoria del padre e hijo ya son diferentes,
la variable es diferente (3 y 4) y al quitale el compartir los
recursos el el PID es diferente

clone(): implementación

código de clone(): se crea una estructura igual y se va rellenando conforme a los valores del padre > do_fork() esta implementada en kernel/fork.c.

clone(): explicación
los recursos del padre y el hijo son compartidos pero la dirección de retorno del hijo es diferente, para que no vuelva a la misma que la del padre.

ajustar los parámetros de planificación del hijo



- 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

- Número de enlaces
que hay a ese archivo, hoy el número
de acceso a ese archivo
- que le vamos a
desligar del espacio
de nombres
- no tiene sentido que
el mismo sistema
de archivos
- para decir si el espacio de
nombres del hijo es el
- Clone debe hacer algunas comprobaciones:
- Algunos indicadores no tienen sentido juntos, por ejemplo, CLONE_NEWNS y CLONE_FS.
 - Otros deben 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. - que gasta recursos
- borrar todos los enlaces para eliminar el archivo.
- espacio de direcciones necesarias para compartir el espacio de señales
- Hacer un dibujo de las task_struct del llamador
- task_struct del llamador
- state
thread_info
flags de uso
...
real_task
...
parent
...
tcb
...
parent
...
tcb
...
files
signal_pending
...
TID

25 & si el contador no está a 0 no libra el archivo

Consideraciones

comprobaciones:
en sentido juntos, por
CLONE_FS.

Z: CLONE_THREAD con
SIGHAND con CLONE_VM.
o r CLONE_XXX, la
mparte (no se copia).
ctura compartida un **contador**
enta que cuantos hilos la
s una de las referencias a la
ontador; si este llega a cero,
a necesidad de un recolector

25 & si el contador no estar
a 0 no borra

el archivo

el
direcciones de usuario.
truct->mm=NULL.
yendo a los antiguos
iciencia).

Kernel con la función

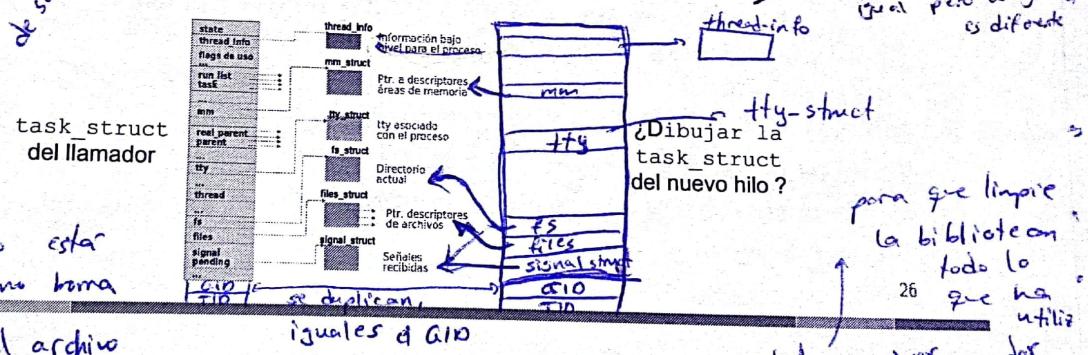
para decir si el espacio de
nombres del hijo es el mismo o no que el del padre

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.



Terminar un proceso

A nivel de biblioteca, da la oportunidad al programador si lo pone el elaborador lo incluye para que el programa finalice

- > Se produce cuando un proceso invoca:
 - Voluntariamente a `exit()` o `return()` -en el `main()`- que finalizan el proceso primero a nivel de biblioteca; o
 - directamente a `_exit()` (llamada al SO) que no da la oportunidad de finalizar el proceso a nivel de biblioteca

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()`. Permite a una hebra kernel crear otra hebra kernel en modo usuario no kernel. Ver los hebras kernel se puede leer ni escribir en terminal.

UID	FID	PPID	C	STIME	TTY	TIME	CMD
Root	1	0	0	08:54	?	00:00:01	/.../systemd..
root	2	0	0	08:54	?	00:00:00	[kthreadd] ~ hebras Kernel
root	3	2	0	08:54	?	00:00:00	[ksftirqd/0] entre parentesis
root	5	2	0	08:54	?	00:00:00	[kworker/0:0H]

Al ser una estructura monolítica un servicio en modo kernel se ejecuta como un programa que el usuario demanda

Terminar un proceso

> Se produce cuando un proceso invoca:

- Voluntariamente a `exit()` o `return()` -en el `main()`- que finalizan el proceso primero a nivel de biblioteca; o directamente a `_exit()` (llamada al SO) que no da la oportunidad de finalizar el proceso a nivel de biblioteca.
 - Involuntariamente: recibe una señal y se aplica la acción por defecto, que es terminar.

al recibir la señal, si el usuario no ha introducido
navegador de señal se aplica la opción por
defecto.

Código de finalización

↳ bien otra cosa
algún error)
código que
se envía al
padre para
declarar como
no terminado
el proceso

exit()

↑ el proceso se está
eliminando y no se puede utilizar

- > Finaliza el proceso do_exit() en kernel/exit.c:
 - Activa PF_EXITING → marca el proceso como terminado.
 - Decrementa los contadores de uso de mm_struct, fs_struct, files_struct. Si estos contadores alcanzan el valor 0, libera los recursos. → si no son 0 se dejan por si alguien los utiliza
 - 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 parente 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 parente pueda recuperar el código de finalización.
¿Cuando se libera el resto?
- > wait(): llamada que uno de sus hijos finaliza, llamando el PID del hijo (código de finalización)
 - > Esta función invoca
 - Elimina el descriptor
 - Si es la última tarea notifica al parente
 - Libera la memoria task_struct.

wait()

> wait(): llamada que bloquea a un proceso padre hasta que uno de sus hijos 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 es zombi, notifica al padre del líder zombi.
 - Libera la memoria de la pila kernel, thread_info y task_struct.

wait(estado): el padre se bloquea hasta que el hijo finaliza el hijo (la señal ^{al nanda} siACTHLD indica al padre que el hijo ha finalizado).

30

fork + exec: no espera a que el hijo finalice

muchos sistemas
no tienen
trabajos batch,
no siempre
están en la
cola de la
planificación
mientras los
procesos
se preparan

Stallings

mos

Tipos de planificadores

> Planificador a:

- Largo plazo –
procesos por lotes

- Corto plazo o
scheduler *el que nos
referimos en general*
- Medio plazo –
gestor de memoria

Si no hay procesos en la cola de preparados
(es bueno, normalmente hay % de CPU que
no se está utilizando).

Se introduce un proceso disimulado para que
cuando no haya otros procesos ejecutándose

se ejecute él. Nunca se debe bloquear
para que esté siempre en la cola de
preparados.

Proceso nulo

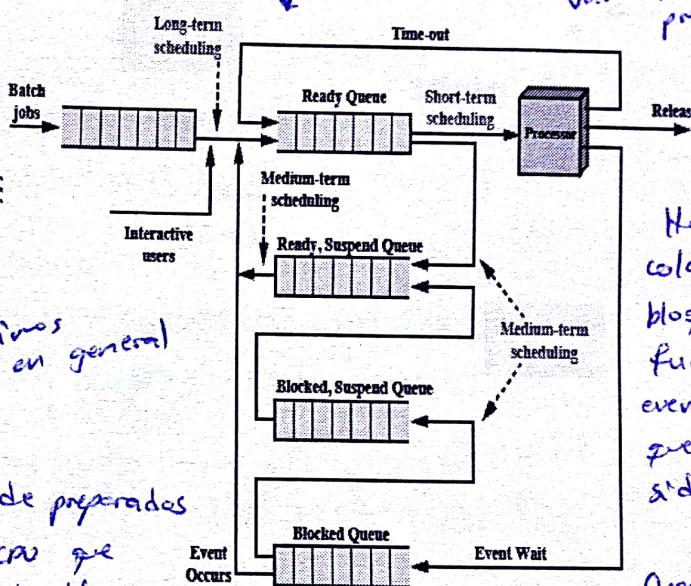


Figura del libro de W. Stallings

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)

multos sistemas
no tienen
tardos batch,
más largos
ellos se
mueve a la cola dentro
de preparar los procesos
en de la cola batch

- > Planificadores
 - Largo plazo
 - procesos por
 - Corto plazo
 - scheduler
 - Medio plazo
 - gestor de m

Si no hay procesos en
(es buzo, normalmente no se está utilizando).

Se introduce un proceso cuando no haya

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.

mechanismo de protección de secciones críticas (se hace todo el kernel una

sección crítica y así no se puede interrumpir algo importante)

En modo kernel
el planificación era una
apropiativa

se le utilizan
los SO actuales

1

desde otro componente fuera del SO

se le puede quitar la CPU porque
no haya terminado, para que
otra tarea más importante

la grano de
longitud de
que tarda el proceso en
abrir de nuevo la CPU

modo
busto

> ¿Qué ocurre
encuentras cuando es decir

- siem
- teng

↓ el sistema no responde
bien en no apri
y tiene

35

Se gente él. Nunca se debe bloquear para que esté siempre en la cola de preparadas.

Proceso nulo *preparadas.*

- No hace nada actualmente hace halt ?
(rompe el ciclo de ejecución de la CPU hasta que aparece una interrupción)

 - > ¿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.

l sólo se ejecuta cuando no
hay otros procesos antes

no responde bien en no apropiativo y tiempo-real pues tienen que esperar las tareas de tiempo real a que terminen otras que no son de tiempo real

Proceso nulo: implementación

> 1^a solución:

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

Cambio_contexto(Pi, Pj);

> Solución óptima

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

- donde hemos creado proceso nulo/ocioso
- Siempre “preparado”
- Nenor prioridad de sistema (no comunica)

los que componen
siempre la cola. Si
nos aseguramos de
nunca estar
vacía no hay
que comprobarlo
(más optimo)

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?

Cuando estás a punto de volver de la interrupción a modo usuario planificado, cuando se termina la ejecución en modo kernel y ~~se~~ se retorna a modo usuario.
 Se le ocultan las interrupciones a los procesos

los flujos de interrupciones no pueden afectar a los procesos (se
No hay algoritmos de tiempo-real específicos.

Planificación en tiempo-real

Planificador: algoritmo

→ genérico,
independiente
de la base

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;
```

se vuelve a planificar
y se repite la
secuencia

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 (^{un paquete} cpufrequtils y cpupower) y planificación

Se controla a
nivel de SO

Cuando el
procesador está
ocioso podemos
dismuir la frecue
el voltaje, hace menos
trabajo

→ se implementan
estos estados en
el hardware, el SO
no tiene que
intervenir

→ todos los
procesadores
actuales pueden
funcionar
a diferentes
voltajes y
frecuencias
y puede hacer
cosas
y viccias
77
y producir
menos
ciclos por unidad
de tiempo

esta función
permite
que el sistema
conserven

> **Advanced Configuration and Power Interface**: especialmente abierta para la gestión térmica controlada.

> Desarrollada por Microsoft, HP, Phoenix, y Toshiba.

> Define cuatro estados (G-estados):

- G0: estado de funcionamiento, entre estados-C y estado dormido.
- G1: estado dormido.
- G2: Estado apagado.
- G3: Estado apagado.

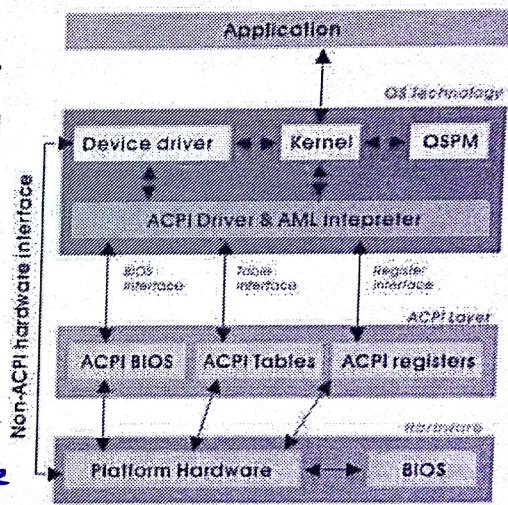
Especificación ACPI

hardware

✓ no importante,
sólo para que
nos suene

- > **Advanced Configuration and Power Interface:** especificación abierta para la gestión de potencia y gestión térmica controladas por el S/I
> 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 ^{sleep}
- G3: Estado apagado mecánico



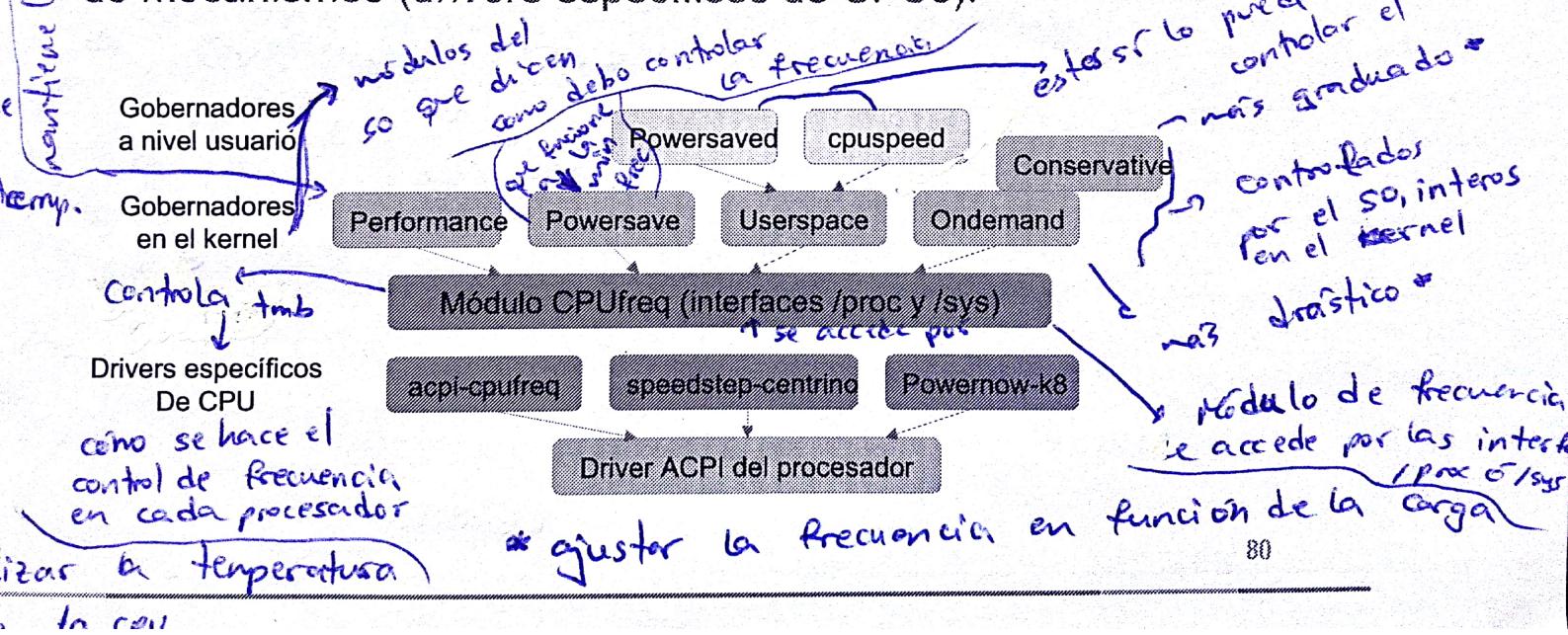
apagones el ordenador pero
se sigue
funcionando
calientado

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

78

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).

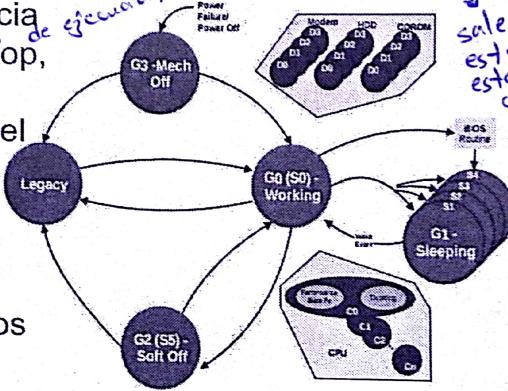


trabajo

Estados de la CPU

Si puede regular el voltaje solo que no de forma continua

- > **S-estados:** estados dormidos en G1. Van de S1 a S5. *→ Vamos apagando*
 - > **C-estados:** estados de potencia en G0. C0: activo, C1:halt, ^{modo de ciclo de} C2:stop, ^{pausa} 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.



ACPI spec v5.0, Dic. 2011

donde no hace trabajo la cpu

- Gobernadores a nivel usuario
- Gobernadores en el kernel

Drivers específicos
De CPU

Primero que el usuario tenga dos opciones: que fracione a la matina velocidad (a tipo) o el matino abajo

puede haber diferentes gobernadores en diferentes procesadores

Gobernadores

- > **Performance** – 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`

> Cp
relat
utilis
frecu
> Cp
de to
anter
> Pa
razon
proce
> Se
> TL
energ

Herramientas

para manejar el
control de frecuencia
en el SO

- > **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-profiles.