Sistemas operativos avanzados

Planificación del procesador

2ª parte: planificación en multiprocesadores

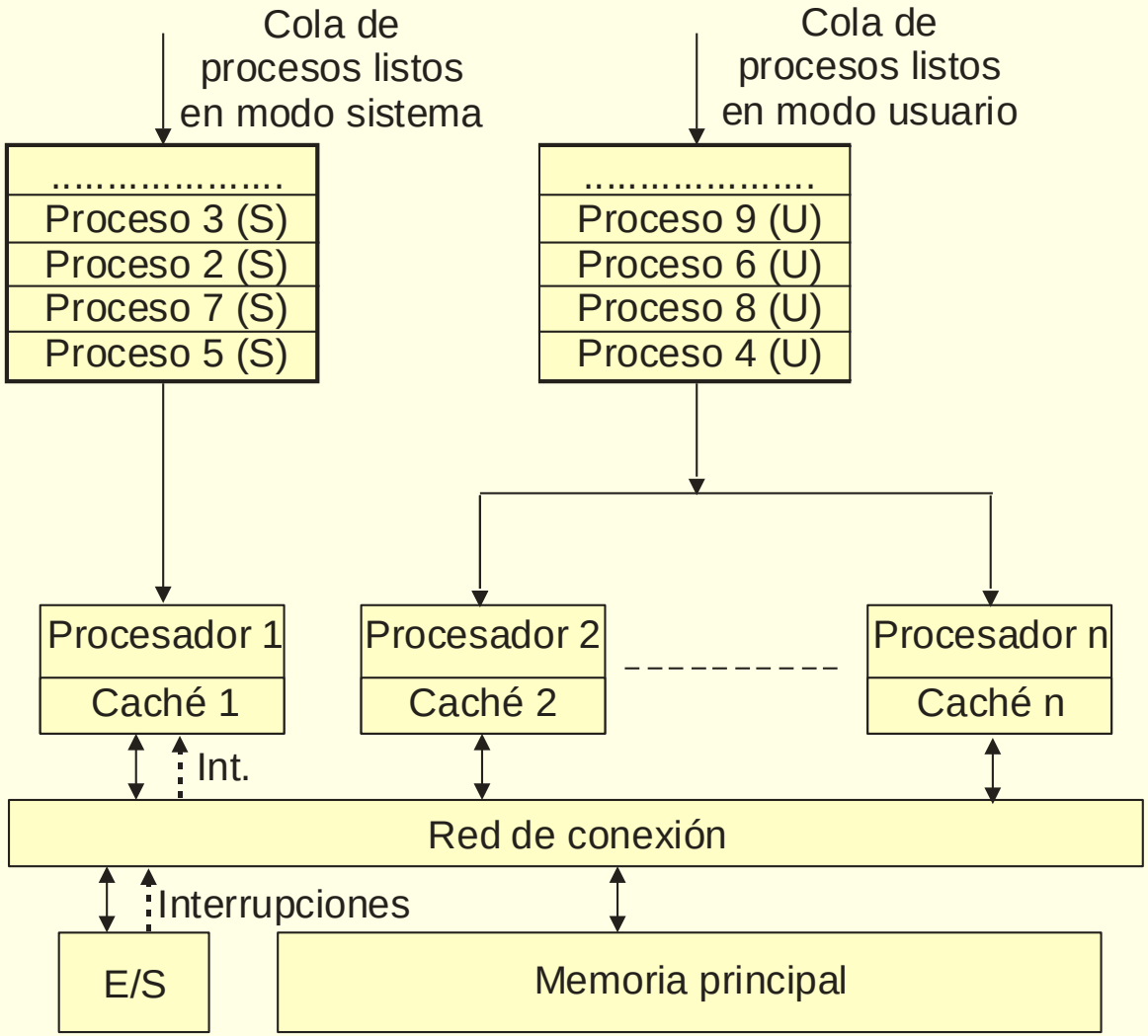
Contenido

- ASMP *versus* SMP
- Planificación en multiprocesadores
- Planificación con cola única
- Sistema multiprocesador jerárquico
- Planificación con una cola por procesador
- Planificación de multiprocesadores en Linux

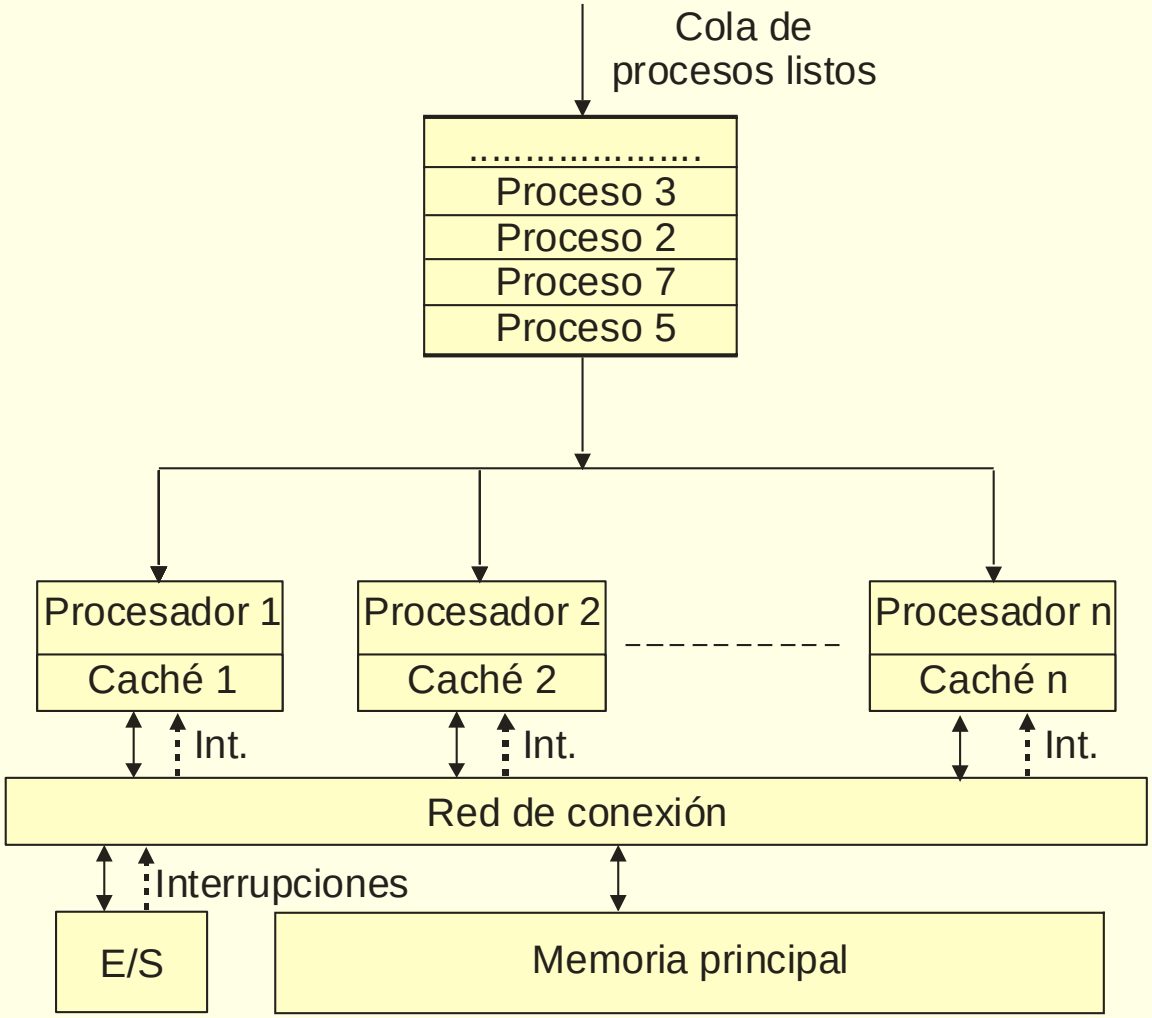
Multiprocesamiento asimétrico vs. simétrico

- Difícil adaptar SO de UP para MP
  - Concurrencia se convierte en paralelismo real
- Solución de compromiso: Multiprocesamiento asimétrico (ASMP)
  - Simetría en hardware pero no en software
  - SO sólo se ejecuta en UCP maestra
    - Llamadas al SO, excepciones e interrupciones en esa UCP
  - Se convierte en "cuello de botella": SO no escalable
  - Beneficioso sólo para programas paralelos que usan poco el SO
- Solución definitiva: Multiprocesamiento simétrico (SMP)
  - SO se ejecuta en cualquier UCP
    - Llamadas al SO y excepciones en UCP donde se producen
    - Interrupciones en UCP que las recibe

Multiprocesamiento asimétrico (ASMP)



Multiprocesamiento simétrico (SMP)



Planificación en multiprocesadores

- *Trivial*: *N* UCPs ejecutan *N* procesos elegidos por planificador
- No es tan fácil; hay que tener en cuenta:
  - Afinidad natural (*soft*) y estricta (*hard*)
  - Multiprocesadores jerárquicos (SMT, CMP, SMP, NUMA,...)
    - Procesadores lógicos, *cores*, *sockets*, nodos,…
    - Compartimiento de recursos entre algunos procesadores
      - P.e. *cores* de mismo *socket* pueden compartir caché L2 o alimentación
  - Evitar congestión en operación del planificador
    - P.e. debida al uso de cerrojos al acceder a cola de listos
  - Además de rendimiento puede haber otros parámetros
    - P.ej. minimizar consumo
- 2 esquemas: Cola única vs. Una cola/procesador
  - Cola única: UCP elige qué proceso ejecuta (*self-scheduling*)
  - Cola por UCP: cada UCP se planifica de forma independiente
  - Linux a partir de versión 2.6: uso de una cola/UCP

Puntos de activación en multiprocesador

- Puntos del SO donde puede invocarse el planificador:
  1. Proceso en ejecución finaliza
  2. Proceso realiza llamada que lo bloquea
  3. Proceso realiza llamada que desbloquea proceso más urgente
  4. Interrupción desbloquea proceso más urgente
  5. Proceso realiza llamada declarándose menos urgente
  6. Interrupción de reloj marca fin de rodaja de ejecución
  7. **Proceso cambia su afinidad estricta**
     - P.e. si proceso impide su ejecución en UCP actual

Afinidad estricta

- Planificación debe respetar afinidad estricta (*hard affinity*)
- Proceso informa de qué UCPs desea usar
- Cambios en el esquema de planificación
  - Proceso pasa a listo: Sólo UCPs en su afinidad estricta
  - UCP queda libre: Sólo procesos que incluyan a esa UCP

- Servicio POSIX para el control de afinidad estricta

  int **sched_setaffinity**(pid_t pid, unsigned int longit, cpu_set_t *máscara)
  - *máscara* define en qué UCPs puede ejecutar *pid*
  - Usada por mandato `taskset` de Linux
- Servicio Windows para el control de afinidad estricta

  BOOL **SetProcessAffinityMask**(HANDLE hpr, DWORD_PTR máscara)
  - *máscara* define en qué UCPs puede ejecutar proceso

Planificación en MP con cola única

- 1ª versión: Teniendo en cuenta afinidad estricta pero no natural
- Proceso en ejecución en UCP *U* pasa a bloqueado, listo o termina
  - Planificador elige proceso listo *Q más importante* (*máx(Prio(Q))*)
    - Sólo entre los que tienen a *U* en su máscara de afinidad estricta
  - *Self-scheduling*: reparto equilibrado de carga automático
- Proceso *P* pasa a listo por desbloqueo o nuevo: se le asigna
  1. Cualquier UCP libre *U*
     - Sólo entre las que *P* tiene en su máscara de afinidad estricta
  2. Si no UCP libre: Busca *U* entre UCPs $\subset$ en afinidad estricta *P*
     - donde ejecute proceso *Q* con menos prioridad en el sistema
     - Si *Prio(P)>Prio(Q)* $\rightarrow$ *Q* debe ser expulsado (sino *P* sigue listo)
  - Si UCP elegida *U*$\neq$UCP donde se produce desbloqueo/creación
    - IPI para forzar CCI en *U*

Propagación de expulsiones

- En principio, las expulsiones no se deberían propagar
  - Pr. expulsado no debería ejecutar: es el – *importante* del sistema
  - Pero afinidad estricta puede provocar la propagación
- Ejemplo con prioridad y máscara de afinidad estricta:
  - 4 UCPs: P1 en UCP1 prio 4 más 1100; P2 en UCP2 prio 3 más 0110; P3 en UCP3 prio 2 más 0011; P4 en UCP4 prio 1 más 1001
  - Se desbloquea P5 prio 5 más 1000
  - P5 expulsa a P1; P1 a P2; P2 a P3 y P3 a P4
  - Resultado: P5 en UCP1; P1 en UCP2; P2 en UCP3; P3 en UCP4
- Después expulsar proceso comprobar si debe ejecutar en otra UCP
  - Mismo tratamiento con expulsado *E* que en desbloqueo o nuevo
    - Busca entre UCPs $\subset$ en afinidad estricta *E* donde ejecute *Q*
      - con menos prioridad en el sistema y *Prio(E)>Prio(Q)*

Otros escenarios de propagación de expulsiones

- Expulsiones en cadena por cambio máscara de afinidad estricta
- Ejemplo: proceso elimina UCP actual de máscara afinidad estricta
  - P1 en UCP1 prio 4 más 1100; P2 en UCP2 prio 3 más 0110; P3 en UCP3 prio 2 más 0011; P4 en UCP4 prio 1 más 1001
  - P1 más 0100→ P1 expulsa a P2; P2 a P3; P3 a P4 y P4 a proc. nulo
- Situaciones que requieren reubicación de procesos
  - ¡Proceso expulsa a proceso más prioritario!
  - No contempladas por algunos SS.OO.
- Ejemplo: proceso extiende su máscara de afinidad estricta
  - P5 listo prio 1 más 1000
  - P1 en UCP1 prio 2 más 1000; P2 en UCP2 prio 3 más 0110; P3 en UCP3 prio 4 más 0011; P4 en UCP4 prio 0 más 1001
  - P1 más 1100→ P1 UCP2; P2 UCP3; P3 UCP4; P5 UCP1

Introduciendo la afinidad natural en la planificación

- Solución planteada es adecuada para sistemas t. real (no crítico)
  - *N* UCPs ejecutan *N* procesos más importantes
  - Pero no para sistema de propósito general
- Afinidad natural: mejor ejecutar proceso en misma UCP
  - Fundamental aprovechar información en sistema de cachés
    - Factor crítico para obtener buen rendimiento en multiprocesador
  - En SO propósito general + importante que respetar prioridad
- Como consecuencia de dar soporte a la afinidad natural:
  - Se asigna a UCP un proceso que no es el más *importante*
  - Se expulsa un proceso que no es el menos *importante*
- Implementación: se suma *bonus* a prioridad de proceso *P* afín
  - Si (P->ultima_UCP == U) → P->prio + bonus;
  - Valor de *bonus* no debe desvirtuar totalmente prioridad
- Proceso nuevo no tiene afinidad: puede usar cualquier UCP

Planificación en MP cola única y afinidad natural

- Proceso en ejecución en UCP *U* pasa a bloqueado, listo o termina
  - **Planificador elige proceso listo Q con máx(Prio(Q)+bonus)**
    - Sólo entre los que tienen a *U* en su máscara de afinidad estricta
    - **Si *(Q->ultima_UCP == U)* → afín: se suma *bonus* a prioridad**
- Proceso *P* pasa a listo por desbloqueo o nuevo: se le asigna *U*
  1. ***UCP afín → si libre y está en máscara afinidad estricta de P***
  2. Cualquier UCP libre incluida en máscara afinidad estricta de *P*
     - **Por afinidad mejor UCP que lleve más tiempo "sin ejecutar"**
  3. Si no UCP libre: Busca *U* entre UCPs ⊂ en afinidad estricta *P*
     - donde ejecute proceso *Q* con menos prioridad en el sistema
     - Si *Prio(P)>Prio(Q)* → *Q* debe ser expulsado (sino *P* sigue listo)
     - **Si *(P->ultima_UCP == U)* → afín: se suma *bonus* a su prioridad**
  - Si UCP elegida *U*≠UCP donde se produce desbloqueo/creación
    - IPI para forzar CCI en *U*

Bonus en acción

- *Bonus*=2; Procesos listos:

  P1 prio 12,últimaUCP 1;P2 prio 13,últimaUCP 2; P3 prio 11,últimaUCP 3;
- Queda libre UCP 1 : selecciona P1 (en vez de P2)
  - Máximo(**P1: 12 + 2**; P2: 13 + 0; P3: 11 + 0) → P1

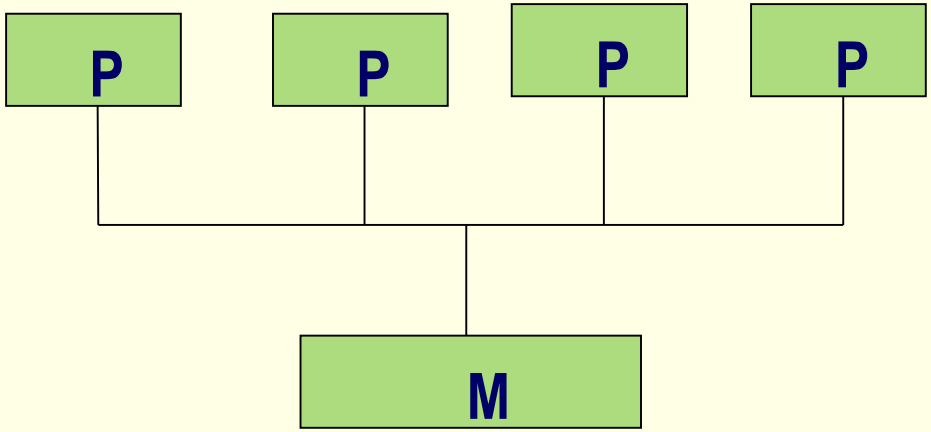
- *Bonus*=2; Procesos en ejecución:

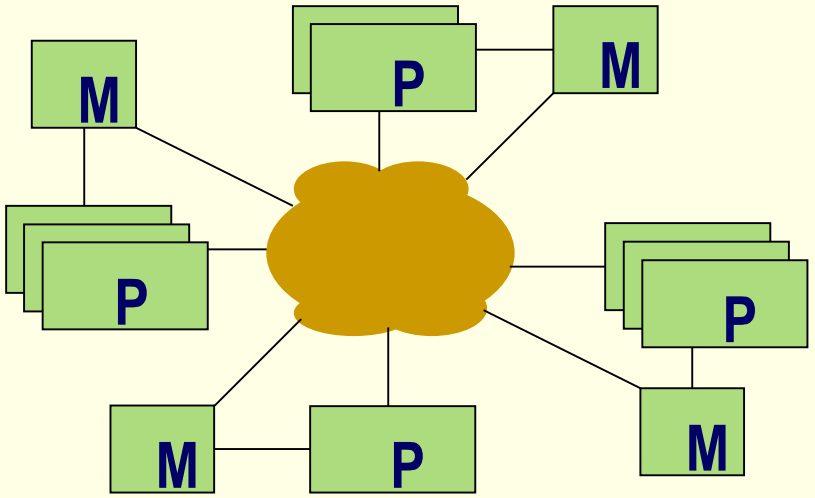
  P1 prio 13, UCP 1; P2 prio 14, UCP 2; P3 prio 18, UCP 3;
- Se desbloquea P4 prio 15 últimaUCP 2: selecciona UCP2 (en vez UCP1)
  - UCP1: P4(15+0)=P1(13+2)
  - **UCP2: P4(15+2)>P2(14+2)**
  - UCP3: P4(15+0)<P3(18+2)

Sistema multiprocesador jerárquico

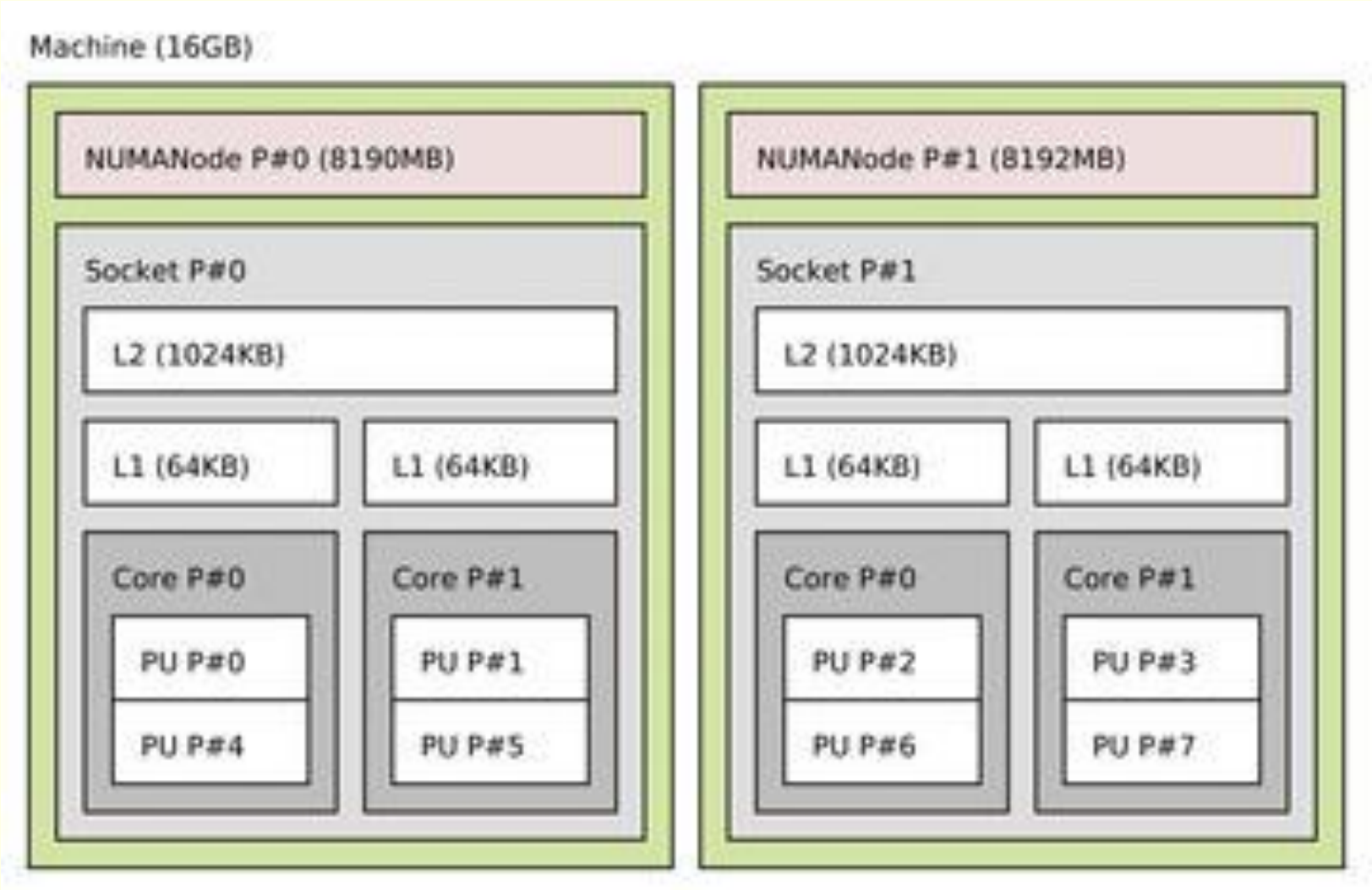
- Multiprocesador no es un conjunto de UCPs al mismo nivel
  - Multiprocesador tiene carácter jerárquico
- Sistema NUMA con múltiples nodos
  - UCP accede a toda la memoria pero local mucho más eficiente
- Cada nodo puede tener varios *sockets* (paquetes/chips)
- Cada *socket* varios núcleos/*cores* (CMP: *Chip MultiProcessing*)
- Cada núcleo varios procesadores lógicos (SMT: *Simultaneous Multithreading*)
- Algunas UCP no independientes: pueden compartir recursos
  - Procesadores lógicos $\in$ mismo núcleo $\rightarrow$ comparten caché L1
  - núcleos $\in$ mismo *socket* $\rightarrow$ pueden compartir caché L2 o L3
  - núcleos $\in$ mismo *socket* $\rightarrow$ alimentación común
- ¿Afecta esta jerarquía al SO? ¿Y a la planificación?

UMA vs NUMA



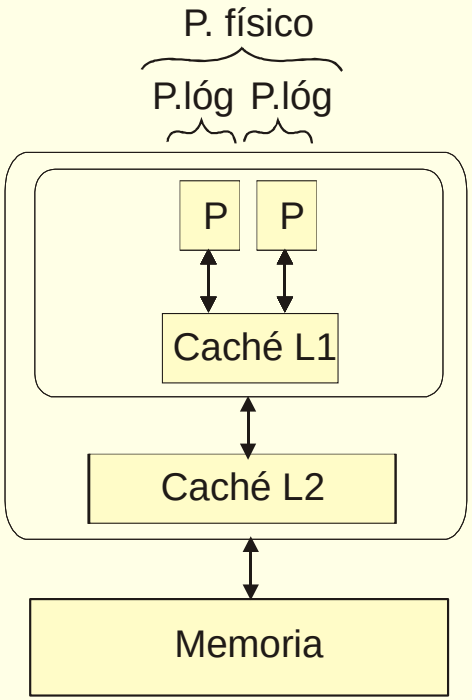


Ejemplo de sistema multiprocesador jerárquico



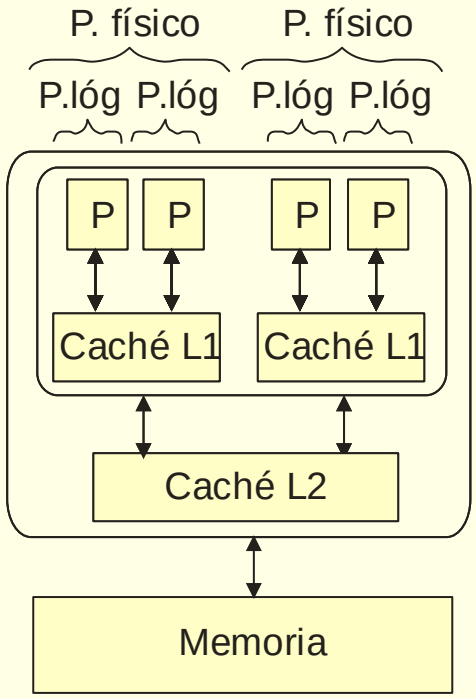
http://www.admin-magazine.com/HPC/Articles/hwloc-Which-Processor-Is-Running-Your-Service

Sistema multiprocesador jerárquico 1 nivel



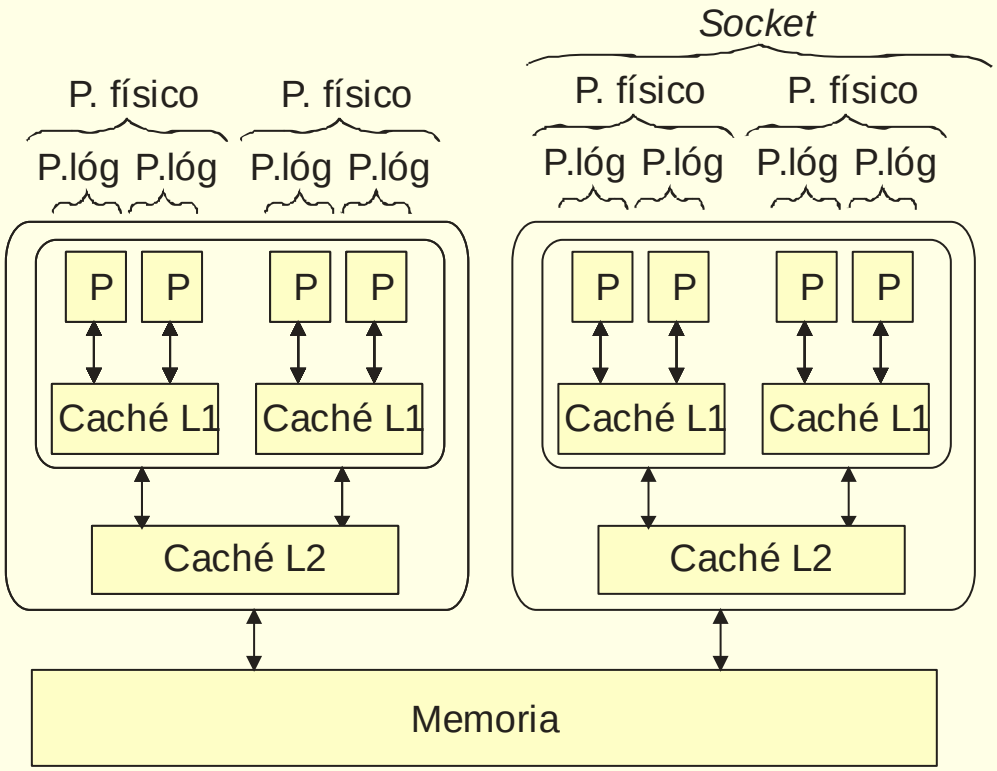
1 núcleo, 2 p. lógico/núcleo

Sistema multiprocesador jerárquico 2 niveles



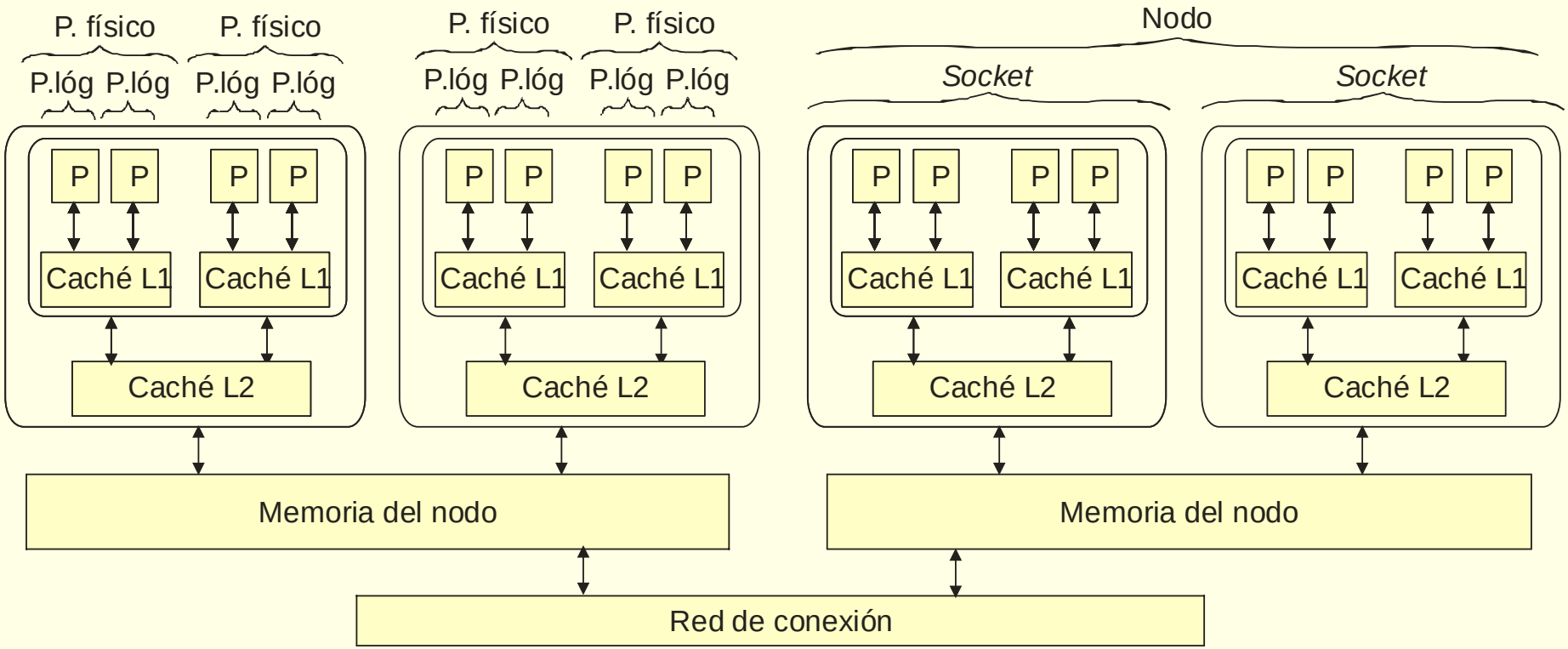
2 núcleos, 2 p. lógico/núcleo

Sistema multiprocesador jerárquico 3 niveles



2 sockets, 2 núcleos/socket, 2 p. lógico/núcleo

Sistema multiprocesador jerárquico 4 niveles



NUMA 2 nodos, 2 sockets/nodo, 2 núcleos/socket, 2 p. lógico/núcleo

Ejemplos de configuraciones MP para Linux

CPUs	Visible CPUs	Memory	Description
2 cores	2	2GB	Low end x86 desktop system 2008
2 cores x 2 threads x 2 sockets	8	4-8GB	Middle-end x86 desktop system 2009
4 cores x 2 threads x 2 sockets	16	8-32GB	Standard low end x86 server 2009
6 cores x 4 sockets	24	32-128GB	Standard 4 socket x86 server 2009
8 cores x 2 threads x 4 sockets	64	128-512GB	Standard 4 socket x86 server 2010
8 cores x 2 threads x 8 sockets	128	128GB-1TB	8 socket x86 server 2010
2 cores x 32 sockets	64	512GB-2TB	High end commercial server 2008
2 cores x 512 sockets	1024	>1TB	Super computer 2007

A. Kleen. "Linux multi-core scalability". *In Proceedings of Linux Kongress*, Octubre 2009.

Sistema operativo para multiprocesador jerárquico

- SO actual debe ser consciente de jerarquía de MP
- CONFIG_SMP

  http://lxr.free-electrons.com/source/kernel/sched/core.c?v=3.19#L411
- CONFIG_NUMA

  http://lxr.free-electrons.com/source/kernel/sched/core.c?v=3.19#L6187
- CONFIG_SCHED_MC

  http://lxr.free-electrons.com/source/kernel/sched/core.c?v=3.19#L6318
- CONFIG_SCHED_SMT

  http://lxr.free-electrons.com/source/kernel/sched/core.c?v=3.19#L6315

Planificación con cola única en MP jerárquico

- Compartimiento de recursos entre algunos procesadores
  - Afecta a afinidad natural: Extensión de afinidad a la jerarquía
  - Afecta a asignación de UCPs libres a proc nuevos (sin afinidad)
    - En MP no jerárquico: vale cualquier UCP libre
- Jerarquía de afinidades: se desbloquea proceso
  - Intenta ejecutar en UCP de ráfaga previa
  - SMT: Afinidad a núcleo
    - Si no disponible intenta ejecutar en UCP lógica $\in$ mismo núcleo
  - CMP: Afinidad a *socket*
    - Si no disponible intenta ejecutar en núcleo $\in$ mismo *socket*
  - NUMA: Afinidad a nodo
    - Si no disponible intenta ejecutar en mismo nodo
- Prioridad matizada por la afinidad natural
  - *Bonus* dependiendo de nivel de afinidad (SMT>CMP>NUMA)

Planificación con cola única en MP jerárquico

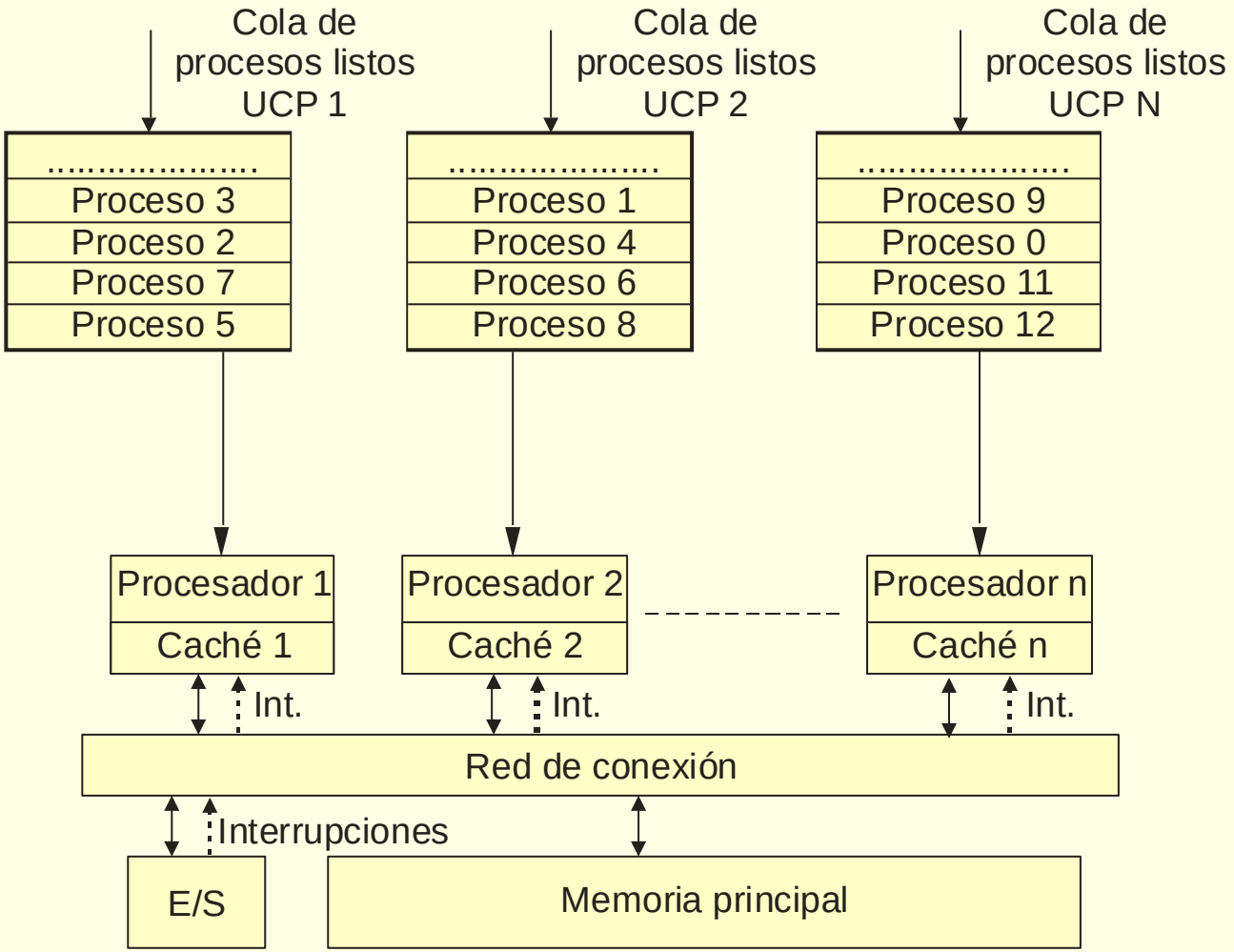
- Asignación UCPs libres a nuevos procesos: eficiencia vs. consumo
  - eficiencia→ "diseminar" procesos; consumo→ "empaquetarlos"
- Asignación buscando eficiencia y paralelismo
  - Si 2 UCP comparten recursos: potencia total < 2*potencia/UCP
    - 2 UCP lógicas ∈ mismo núcleo → comparten caché L1
    - 2 núcleos ∈ mismo *socket* → pueden compartir caché L2 o L3
  - Mejor ir ocupando UCPs con mayor grado de independencia
    - Ejecución de nuevo proceso busca UCP lógica libre
      - Dentro de nodo con más UCPs libres, el socket con más libres y en éste el núcleo con más libres
- Asignación para minimizar consumo
  - Si 2 UCPs comparten alimentación: núcleos ∈ mismo socket
    - Mejor usar núcleo libre de *socket* ocupado que de libre
      - Permite mantener *socket* libre en bajo consumo
- Linux: administrador decide qué prima (versión 3.4)

```
echo 1 > /sys/devices/system/cpu/sched_mc_power_savings
```

Planificación en MP con una cola por UCP

- Cola única:
  - Accesos a cola requieren cerrojo: mala escalabilidad
  - Limitado aprovechamiento de la afinidad natural
    - Procesos cambian de UCP → "*cache line bouncing*"
  - Algoritmo de planificación con extensiones para MP
- Cola por UCP: UCP se planifica de forma independiente
  - No hay congestión por cerrojo
  - Se aprovecha mejor afinidad
  - Algoritmo de planificación UP para cada cola
- Nuevo proceso se le asigna UCP: ¿Cuál?
  - Menos cargada
  - ¿Cómo determina carga: importancia del proceso, uso UCP,…?
- Continúa en la misma excepto si migración por equilibrio de carga

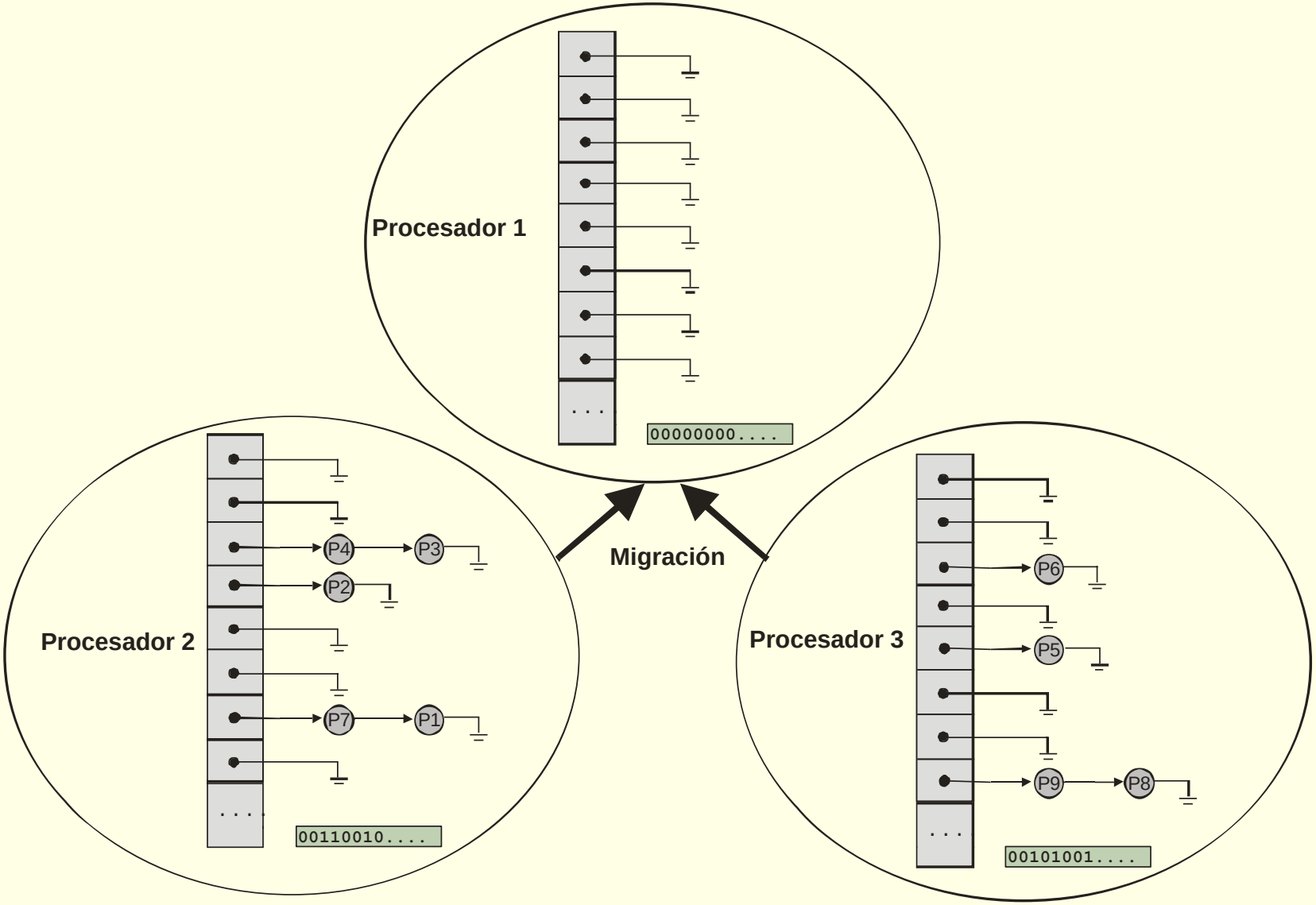
Multiprocesamiento simétrico (SMP) cola/UCP



Equilibrado de carga

- Mecanismo de equilibrado de carga debe ser explícito
  - Migración de procesos ante desequilibrios
- 2 escenarios de equilibrado:
  - Comprobar periódicamente si desequilibrios
  - Si cola de una UCP queda vacía
- Dos estrategias de equilibrado:
  - *Pull*: UCP mueve a su cola procesos de otra UCP
  - *Push*: UCP mueve procesos de su cola a la de otra UCP
- Migración no implica ningún tipo de copia
  - Sólo mover un BCP entre dos colas (pero pierde info. caché)
- Aunque puede requerir cerrojos sobre las colas para estimar carga
  - Justo lo que se pretendía evitar con una cola por UCP
  - Frecuencia de comprobación de desequilibrios ajustada para
    - Evitar sobrecarga de cerrojos pero sin alargar desequilibrios

Equilibrado mediante migración de procesos



Planificación con una cola/UCP: MP jerárquico

- ¿Qué UCP se asigna a un nuevo proceso?
  - Si meta eficiencia: Procesador seleccionado corresponde a
    - Nodo menos cargado (N)
    - *Socket* (S) menos cargado de N
    - Procesador físico (F) menos cargado de S
    - Procesador lógico (L) menos cargado de F
  - Si meta minimizar consumo: justo lo contrario
- Equilibrado siguiendo jerarquía
  - Equilibrar procesadores lógicos de cada núcleo
  - Equilibrar núcleos de cada *socket*
  - Equilibrar *sockets* de cada nodo
  - Equilibrar nodos
  - Migración teniendo en cuenta jerarquía de afinidades
    1. mejor migrar entre UCPs que pertenezcan al mismo núcleo
    2. mejor migrar entre UCPs que pertenezcan al mismo *socket*
    3. mejor migrar entre UCPs que pertenezcan al mismo nodo

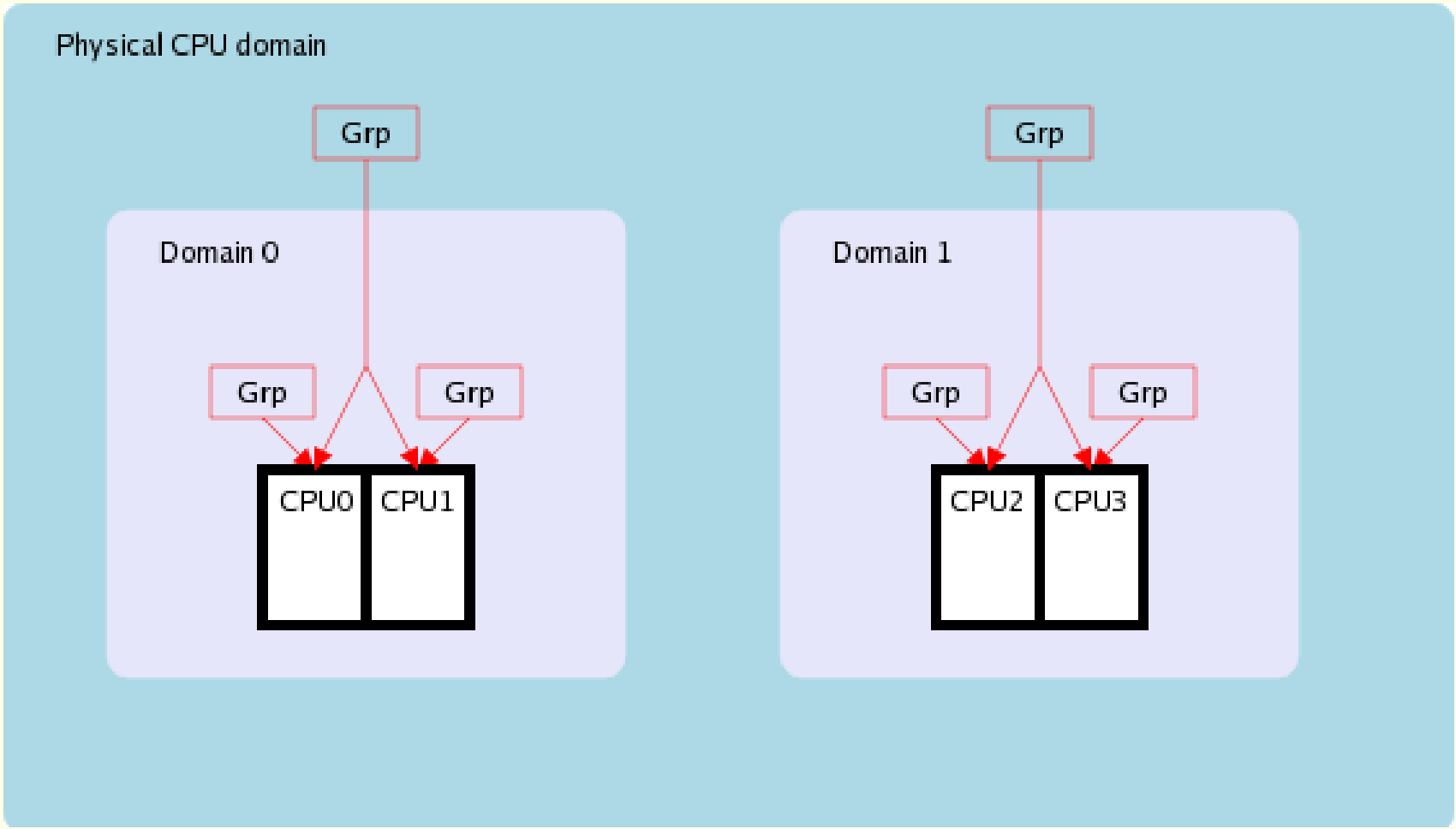
Asignación de procesador en UNIX

- Modelo de procesos de UNIX conlleva 3 puntos de "creación":
  - exec. Pérdida total de afinidad
    - Puede ser buen momento para migrar
    - Buscar procesador menos cargado aplicando jerarquía
  - pthread_create.
    - Mantiene afinidad
      - Razonable asignar mismo procesador
      - Aunque pierde paralelismo
      - Puede aplicarse jerarquía de afinidades
  - fork.
    - Situación intermedia: Afinidad pero hasta que se rompe COW
      - ¿Mismo procesador o no?

Planificación multiprocesador en Linux

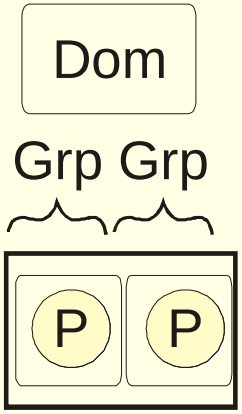
- Mejoras en versión 2.6: uso de una cola por UCP
  - En principio, esquema de planificación ya maduro
  - Pero siempre hay sorpresas (*bugs* que dejan núcleos sin usar):
    - *The Linux Scheduler: a Decade of Wasted Cores*
    - *http://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf*
- Gestión de carácter jerárquico: **dominios de planificación**
  - Dominio=conjunto de grupos
  - Dominio intenta mantener carga equilibrada en sus grupos
- 2 parámetros: poder de cómputo de UCP y carga de un proceso
  - Cada grupo tiene un "poder de cómputo" (*CPU Power*)
    - Tiene en cuenta grado de independencia de UCP
    - 2 UCP lógica $\in$ mismo núcleo $\rightarrow$ *CPU Power del grupo* = 1,1
  - Carga de proceso basada en peso y su uso medio del procesador

Dominios de planificación: 2 núcleos con 2 p.lógicos



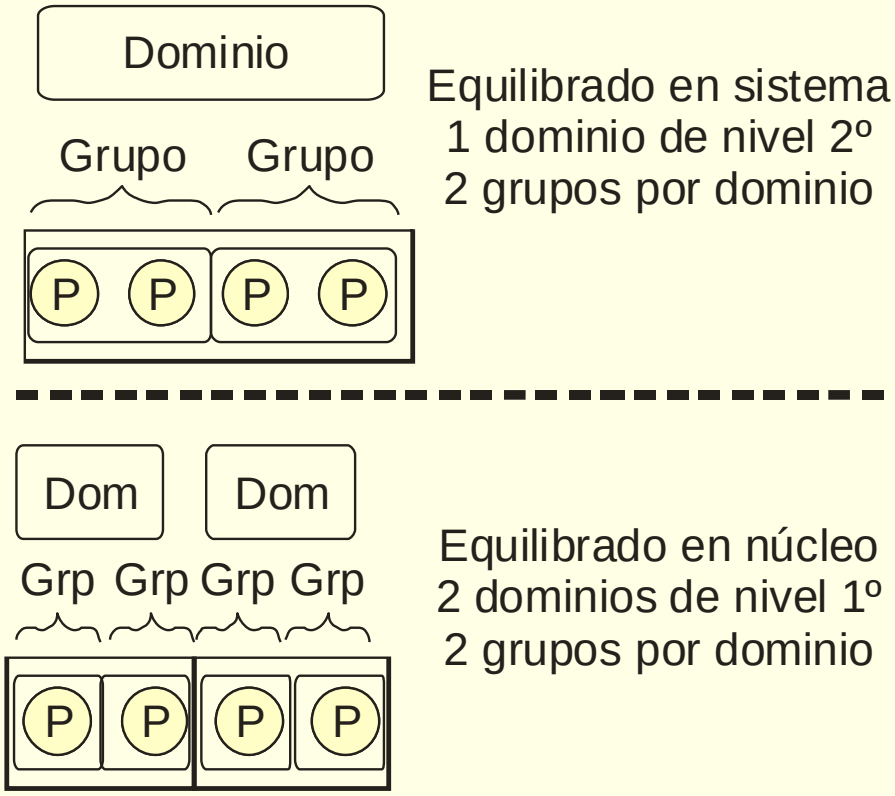
LWN.NET: *Scheduling domains*. https://lwn.net/Articles/80911/

Dominios de planificación: MP jerárquico 1 nivel

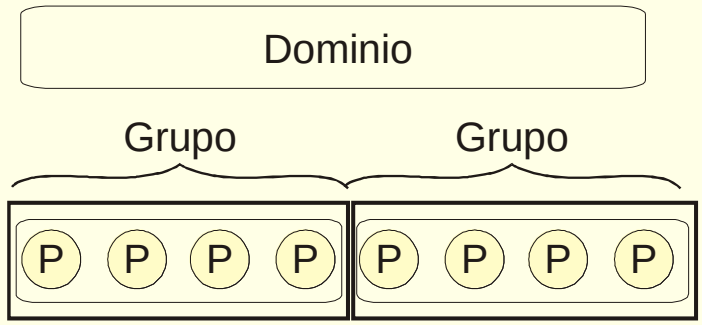


Equilibrado en sistema
1 dominio de nivel 1º
2 grupos por dominio

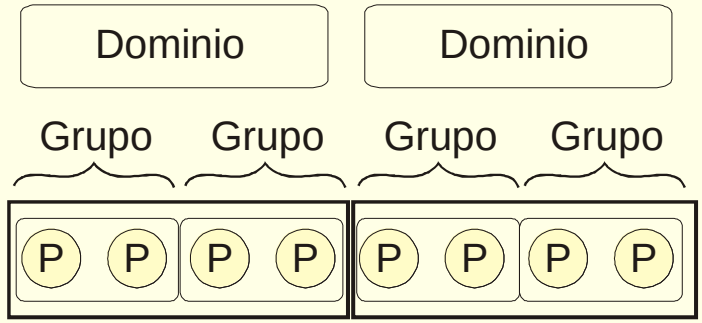
Dominios de planificación: MP jerárquico 2 niveles



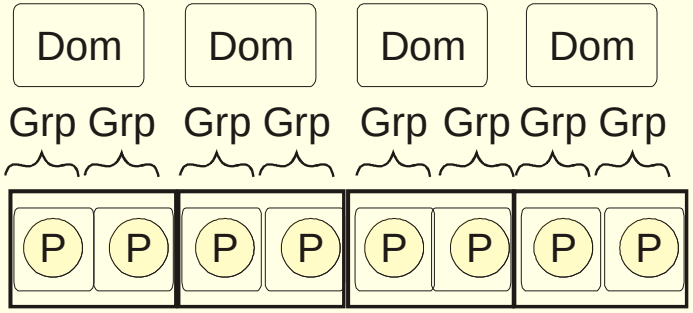
Dominios de planificación: MP jerárquico 3 niveles



Equilibrado en sistema
1 dominio de nivel 3º
2 grupos por dominio

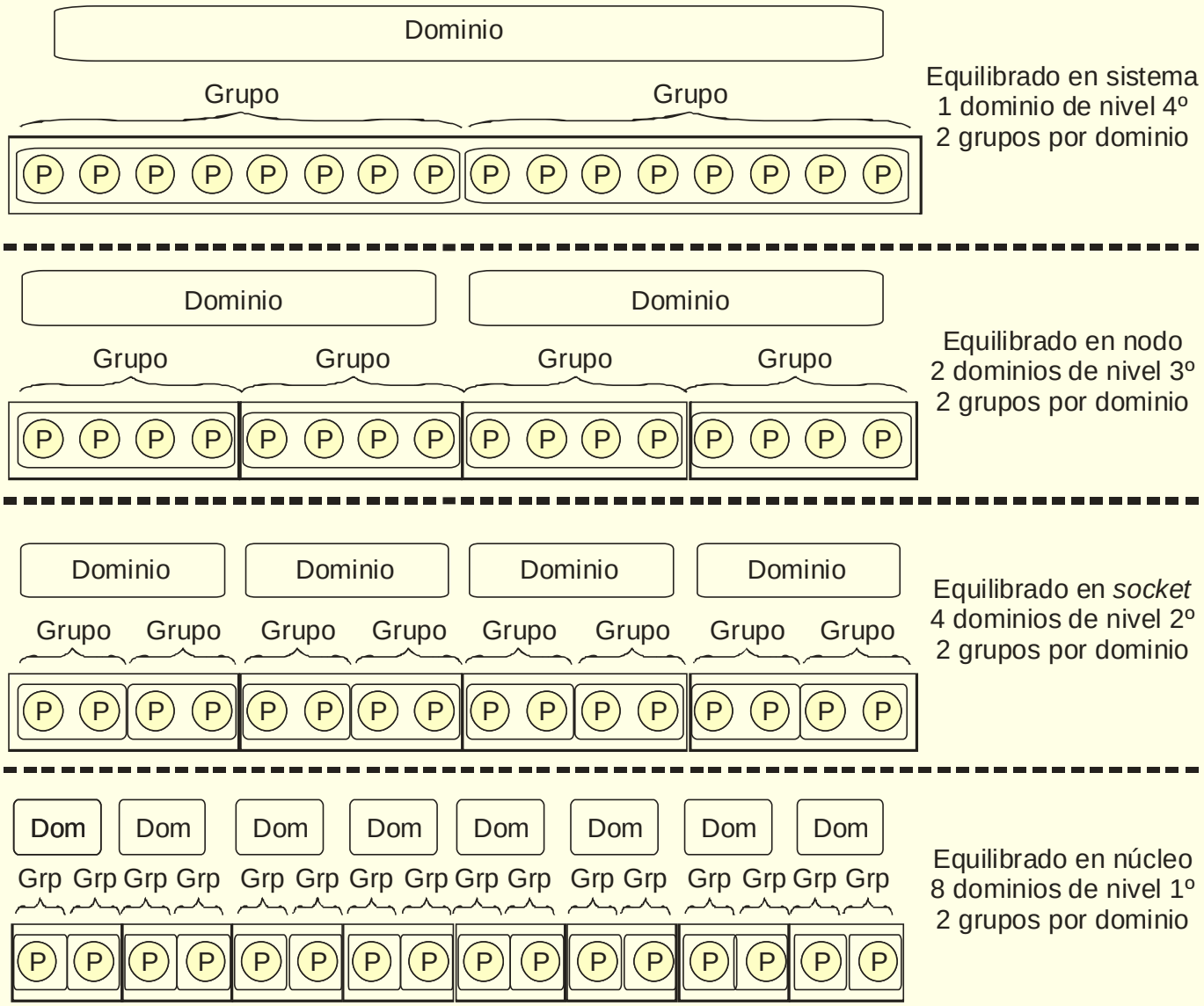


Equilibrado en *socket*
2 dominios de nivel 2º
2 grupos por dominio



Equilibrado en núcleo
4 dominios de nivel 1º
2 grupos por dominio

Dominios de planificación: MP jerárquico 4 niveles



CPU Power en acción

- Sistema con dos *cores*: uno con un *thread* y otro con dos
  - *Core 1: CPU Power 1*
  - *Core 2: CPU Power 1,1*
  - 42 procesos listos: Equilibrio de carga
    - *Core 1*: 20 procesos (42*1/(1+1,1))
    - *Core 2*: 22 procesos (42*1,1/(1+1,1)) → 11/*thread*
- Sistema con dos *cores*: uno con un *thread* y otro con cuatro
  - *Core 1: CPU Power 1*
  - *Core 2: CPU Power 1,3*
  - 92 procesos listos: Equilibrio de carga
    - *Core 1*: 40 procesos (92*1/(1+1,3))
    - *Core 2*: 52 procesos (92*1,3/(1+1,3)) → 13/*thread*

Equilibrado de carga

- Basado en información de configuración de cada nivel
  - Frecuencia de comprobación de desequilibrios
    - Mayor en niveles más bajos
  - Nivel de desequilibrio que provoca migración
    - Menor en niveles más bajos
  - Coste de la migración (mayor cuanto más afinidad se pierda)
    - Entre UCPs lógicas del mismo núcleo → Coste 0
- Equilibrado entre dominios (rebalance_domains)
  - Ejecuta periódicamente en cada UCP *U*
  - Asciende jerarquía de dominios de planificación de *U*
    - Por cada dominio, si se ha cumplido plazo de comprobación
      - Busca el grupo *G* + cargado y si desequilibrio > umbral
      - Mueve a *U* uno o más procesos de UCP más cargada de *G*
- Tratamiento de UCP que se queda libre (idle_balance)
    - Similar al equilibrado entre dominios

Selección de una cola para un proceso

- Selección de cola para un proceso (select_task_rq)
  - Recibe como parámetro el tipo de escenario
- select_task_rq invocada en:
  - exec (sched_exec) y con nuevo proceso (wake_up_new_task)
    - parámetro SD_BALANCE_EXEC y SD_BALANCE_FORK respectivamente
    - Si meta eficiencia: Procesador seleccionado corresponde a
      - Nodo menos cargado (N)
      - *Socket* (S) menos cargado de N
      - Procesador físico (F) menos cargado de S
      - Procesador lógico (L) menos cargado de F
  - Al desbloquearse un proceso (parámetro SD_BALANCE_WAKE)
    - Normalmente, al desbloquearse un proceso vuelve a su cola
    - Pero en algunas circunstancias podría migrarse a una UCP afín