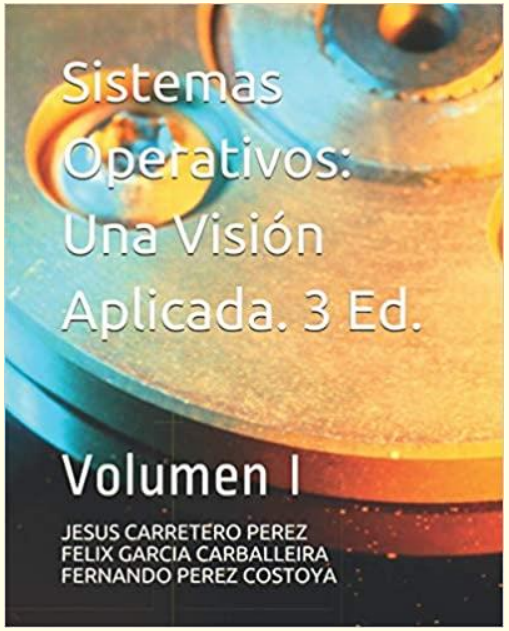
Sistemas operativos avanzados

Planificación del procesador



Capítulo 4 del libro

Organización del tema

- Primera parte:
  - Conceptos generales
  - Algoritmos de planificación en sistemas monoprocesador
- Segunda parte:
  - Planificación en multiprocesadores
- Tercera parte:
  - La planificación en Linux
- Cuarta parte:
  - Planificación en entornos de virtualización

Sistemas operativos avanzados

Planificación del procesador

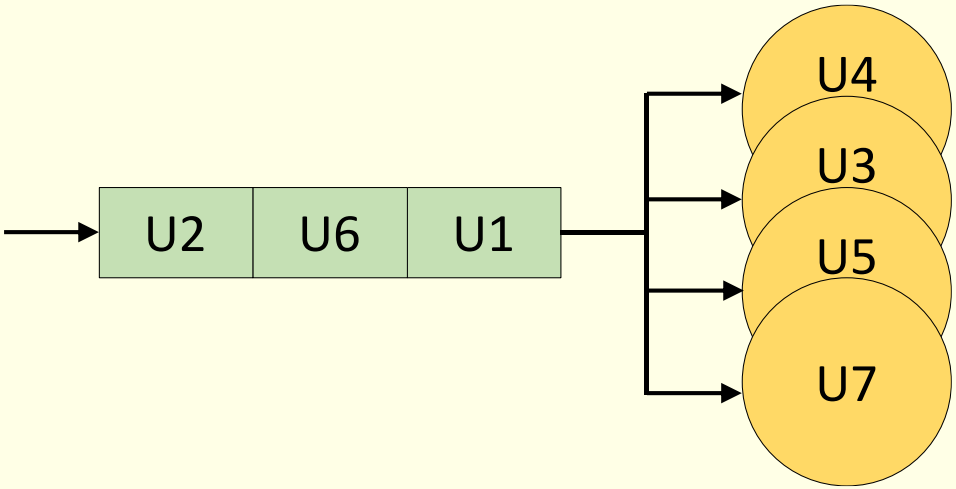
1ª parte: Conceptos generales y planificación monoprocesador

Introducción

- SO planifica recursos: UCP recurso más importante
  - Debe seleccionar qué procesos listos ejecutan en las UCPs
- Evolución del planificación del procesador
  - Aparece con multiprogramación
  - Sistemas por lotes: aprovechamiento de UCP
  - Tiempo compartido: reparto equitativo entre usuarios
  - PC: interactividad; buen tiempo de respuesta
  - Multiprocesadores: aprovechar paralelismo (consumo energía)
  - Tiempo real (no crítico): cumplir plazos
  - Virtualización: contextos de ejecución aislados
- Planificación UCPs en S.O. de propósito general: reto complejo
  - Variedad de plataformas: de empotrados a NUMA
  - Variedad de cargas: servidor, interactivo, científico, etc.
  - NOTA: En S.O. actual la entidad planificable es el *thread*
    - Pero "por tradición" seguimos hablando de procesos

El problema general de la planificación

- Recurso con múltiples ejemplares utilizado por varios usuarios
- Planificación: qué ejemplar se asigna a qué usuario
  - Reparto espacial y temporal del recurso



- Ejemplo: clínica de salud con varios médicos tal que
  - Cualquier médico puede atender a cualquier paciente

Objetivos generales de la planificación de recursos

- Optimizar el uso de los recursos planificados
- Minimizar el tiempo de espera de los usuarios
- Ofrecer un reparto equitativo (*fair scheduling*), evitando inanición
  - Permitiendo usuarios con ≠ pesos (*proportional fair scheduling*)
- Proporcionar grados de urgencia (pueden causar inanición)
- Satisfacer requisitos temporales de usuarios en el uso de recursos
  - Necesidad de control de admisión para garantizarlos
  - Relacionado con urgencia pero no es lo mismo
  - Ancho de banda de uso de recurso por usuario: t. usado/periodo
    - Requisitos temporales → requiere ancho banda mínimo (reserva)
    - A veces, control de ancho de banda máximo (p.e. en pago por uso)
- Otras propiedades deseables en el planificador:
  - Eficiente, escalable, configurable y seguro

Tipos de planificación de recursos

- Planificación expulsiva versus no expulsiva
- Planificación por prioridad
- Planificación equitativa proporcional
- Planificación de tiempo real frente a de “mejor esfuerzo”
- Planificación *work-conserving* o *non-work-conserving*
- Planificación de recursos con múltiples ejemplares

Planificación expulsiva versus no expulsiva

- O con y sin desalojo
- Planificación no expulsiva
  - Mantiene asignación recurso mientras usuario lo necesite
  - Única factible si el recurso no es expropiable
  - Pero puede aplicarse también a recursos expropiables
  - Dificulta reparto equitativo y establecimiento grados de urgencia
- Planificación expulsiva
  - Reasignar recurso mientras está siendo usado
    - Buscando reparto equitativo, por grados de urgencia…
  - Planificador se activa no solo cuando recurso se queda libre

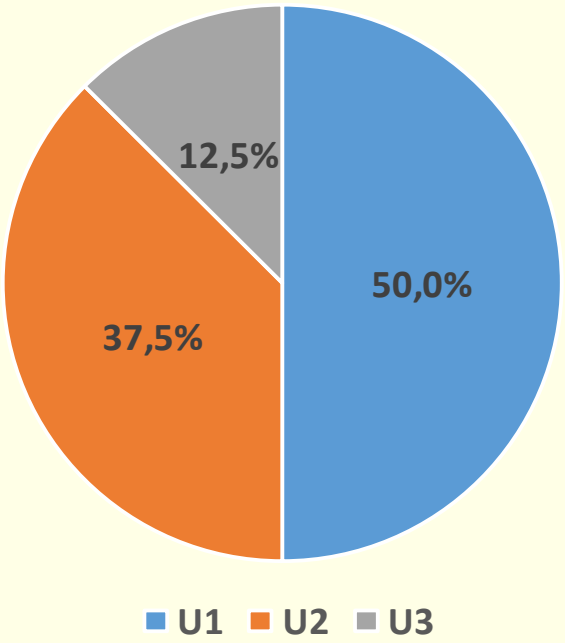
Planificación por prioridad

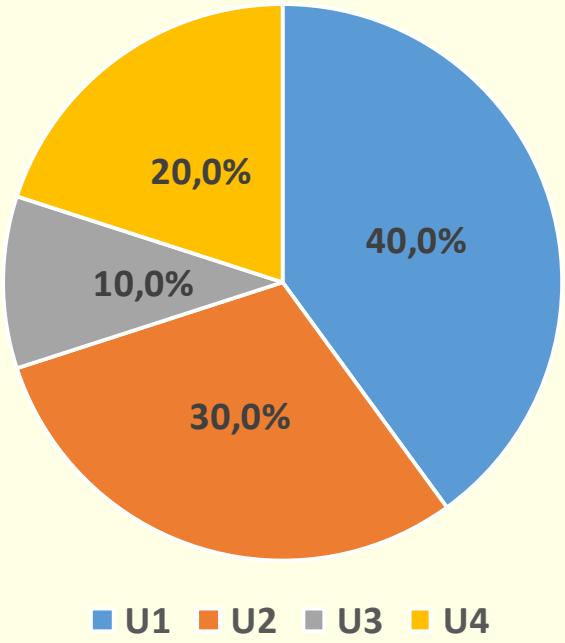
- Se establece orden de prioridad entre peticiones
- Peticiones se sirven siguiendo ese orden
- Permite establecer grados urgencia
- Y minimizar tiempos espera
- Idealmente, petición más prioritaria no debería esperar
  - Espera por petición de menos prio. → Inversión de prioridades
  - Inevitable con recursos no expropiables
  - Pero cuidado con espera no acotada
- Puede causar inanición
- Prioridad conlleva acaparamiento de recursos
  - Alternativa: fijar ancho de banda máximo por usuario

Planificación equitativa proporcional

- $Uso\ ideal\ del\ recurso\ por\ U_i\ en\ intervalo\ t = t \times \frac{Peso_i}{\sum_{j=1}^{N} Peso_j}$
- Irrealizable: requeriría una reasignación continua del recurso
- Tendencia a equidad manteniendo acotado sesgo del objetivo
  - $Sesgo\ de\ U_i = Uso\ ideal\ por\ U_i - Uso\ real\ por\ U_i$
- Importancia basada en pesos vs en prioridades
  - Mayor cuota vs. acaparamiento
- Pueden aplicarse pesos a jerarquía de grupos de usuarios
- No establece ancho de banda mínimo de uso de recurso/usuario
  - P.e. no garantiza que un usuario reciba al menos 10% del recurso
  - Depende de qué usuarios existan en cada momento
- Ni ancho de banda máximo aunque puede extenderse para hacerlo
  - P.e. en sistema de pago por uso

Planificación equitativa proporcional





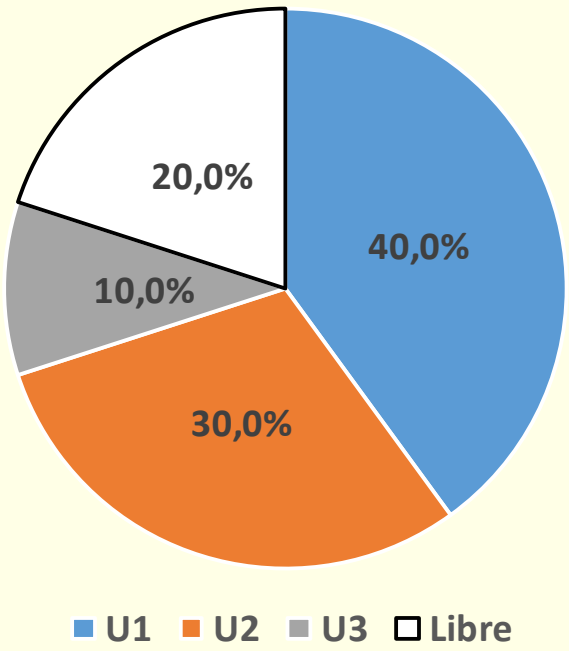
Usuario	Peso
U1	4
U2	3
U3	1

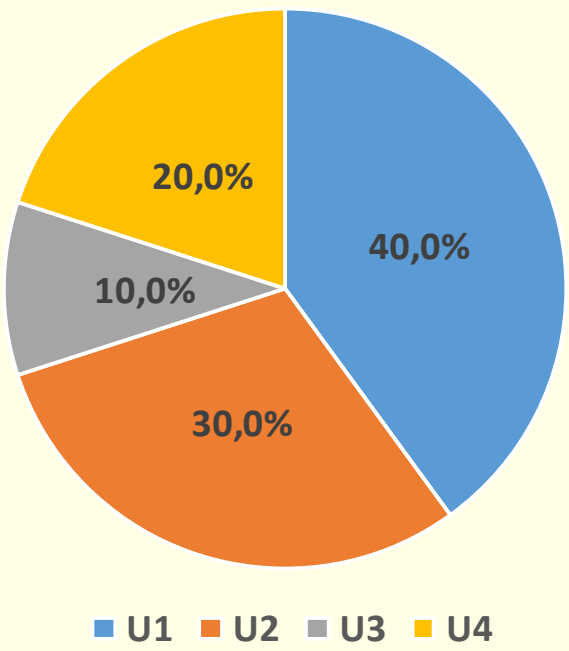
Usuario	Peso
U1	4
U2	3
U3	1
U4	2

Planificación de tiempo real

- Hasta ahora: Alg. planificación de "mejor esfuerzo" (*best-effort*)
- Usuarios con requisitos temporales en el uso de recursos
  - Requiere usar recurso *X* u.t. en las próximas *Y* u.t.
- Planificación de tiempo real
  - Asegura ancho de banda mínimo de uso de recurso a usuario
- Necesidad de control de admisión
  - $\sum_{i=1}^{N} \frac{tiempo\ T_i\ de\ uso\ del\ recurso\ requerido\ por\ U_i}{durante\ periodo\ P_i} \leq 1$
  - Creación nuevo proceso se rechaza si no factible sus necesidades
- Sistema con mezcla de usuarios con y sin restricciones temporales
  - Planificación jerárquica
- Asegura requisitos; ¿Qué hacer con tiempo sobrante?
  - Repartirlo entre usuarios o dejar recurso libre
  - ¿Se puede quedar un recurso libre aunque alguien lo necesite?

Planificación de tiempo real





Usuario	Uso	Periodo
U1	4	10
U2	3	10
U3	1	10

Usuario	Uso	Periodo
U1	4	10
U2	3	10
U3	1	10
U4	2	10

Planificación *work-conserving* vs *non-work-conserving*

- Planificación *work-conserving* (WC):
  - No puede dejarse un recurso sin usar si alguien lo necesita
  - La opción más habitual
- Planificación *non-work-conserving* (NWC):
  - Puede dejarse un recurso sin usar aunque alguien lo necesite
  - Por tarificación (pago por uso)
    - Control de ancho de banda máximo
  - Por rendimiento: no se asigna un recurso libre a un usuario
    - si puede perjudicar a otro usuario más importante que está usando otro ejemplar del mismo recurso relacionado con el primero

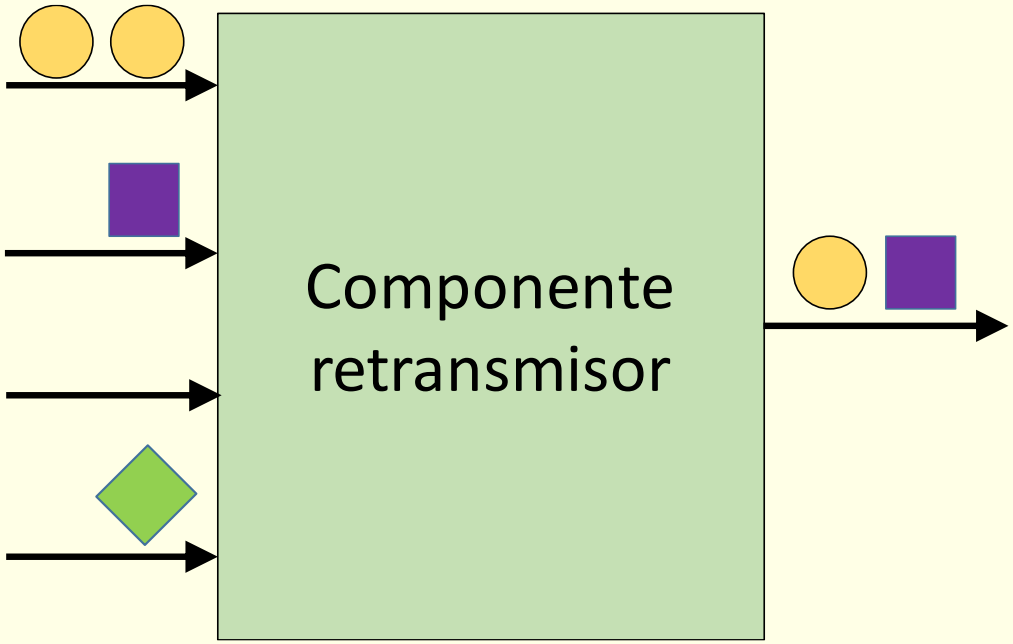
Planificación de recursos con múltiples ejemplares

- En principio, petición se satisface con cualquier ejemplar libre
  - Puede ser conveniente condicionar esa decisión → **afinidad**
- Afinidad a subconjunto de ejemplares de recurso
  - Estricta (*hard*): pedida por el usuario
  - Natural (*soft*): de carácter interno; favorece rendimiento
    - Volver a asignar mismo ejemplar a un usuario
    - Recurso "tiene memoria del usuario"
- A veces, los ejemplares no son independientes
  - Extensión de la afinidad natural a esta dependencia
  - Ej. clínica: salas pareadas comparten puerta y climatización
    - Planificación basada en el consumo
- En general, usuario requiere varios ejemplares de varios recursos
- Prestación de servicio a un conjunto de usuarios relacionados

Planificación de los recursos de un computador

- Procesador: objeto de este tema
  - Recurso expropiable con múltiples ejemplares
  - Reparto espacial y temporal de cada ejemplar
- Memoria
  - Recurso expropiable con múltiples ejemplares (marcos página)
  - Reparto espacial y temporal de cada ejemplar
- Disco
  - Normalmente, recurso no expropiable con un único ejemplar
  - Reparto espacial no expulsivo
- Mecanismos de sincronización
  - No expropiable por su propia esencia

Planificación de colas de mensajes en la red



- Campo muy estudiado
- Similitudes con planificación del procesador
  - Pero también diferencias:
    - No expropiable, uso del recurso conocido (tamaño del mensaje)…

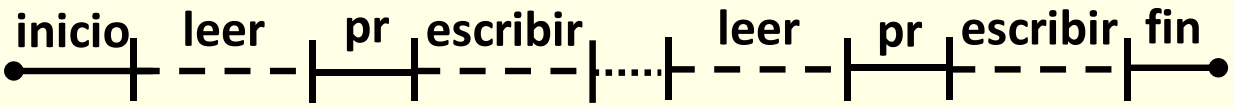
Caracterización de los procesos

- Perfil de uso del procesador (*CPU bound* vs. *I/O bound*)
  - Mejor primero proceso con **ráfaga** de UCP más corta
- Grado de interactividad del proceso
  - Asegurar buen tiempo de respuesta de interactivos
- Nivel de urgencia del proceso
- Equidad y grupos de procesos: *threads* no independientes
  - Incluidos en procesos
  - Que pueden ser parte de aplicaciones
  - Que pertenecen a usuarios
  - Que pueden estar organizados en grupos jerárquicos
- Requisitos temporales del proceso
  - Necesidad de control de admisión
  - Requisitos críticos vs. no críticos

Uso intensivo de la E/S versus la UCP

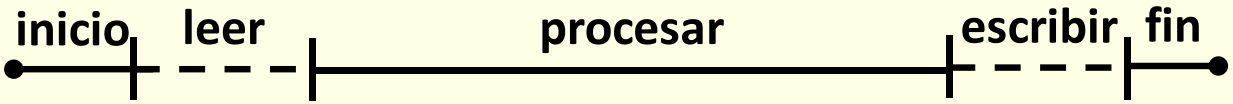
Programa

```
inicio();

Repetir
   leer(fichero_ent, dato);
   /* procesado sencillo */
   res = pr(dato);
   escribir(fichero_sal, res);
hasta EOF(fichero_ent);

fin();
```



Programa

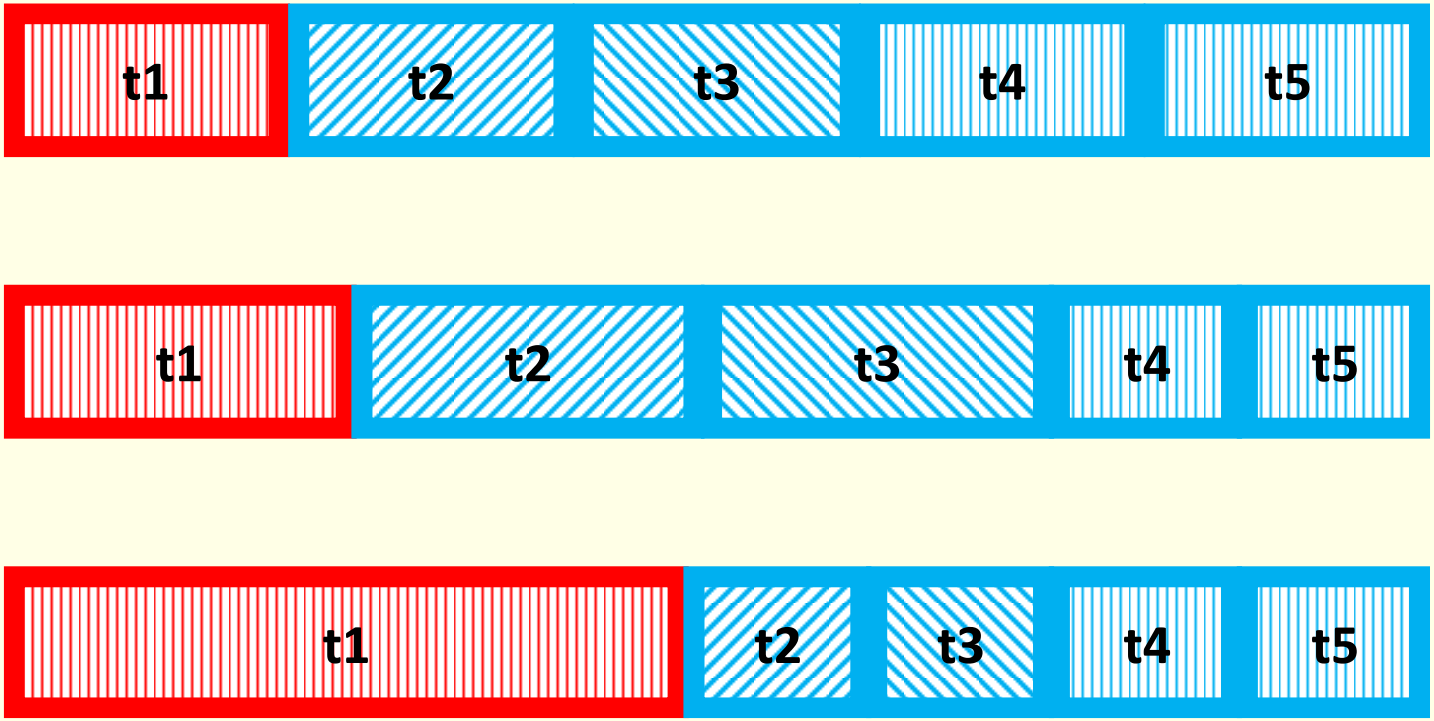
```
inicio();

leer(fichero, matriz);

/* procesado complejo */
procesar(matriz);

escribir(fichero, matriz);

fin();
```



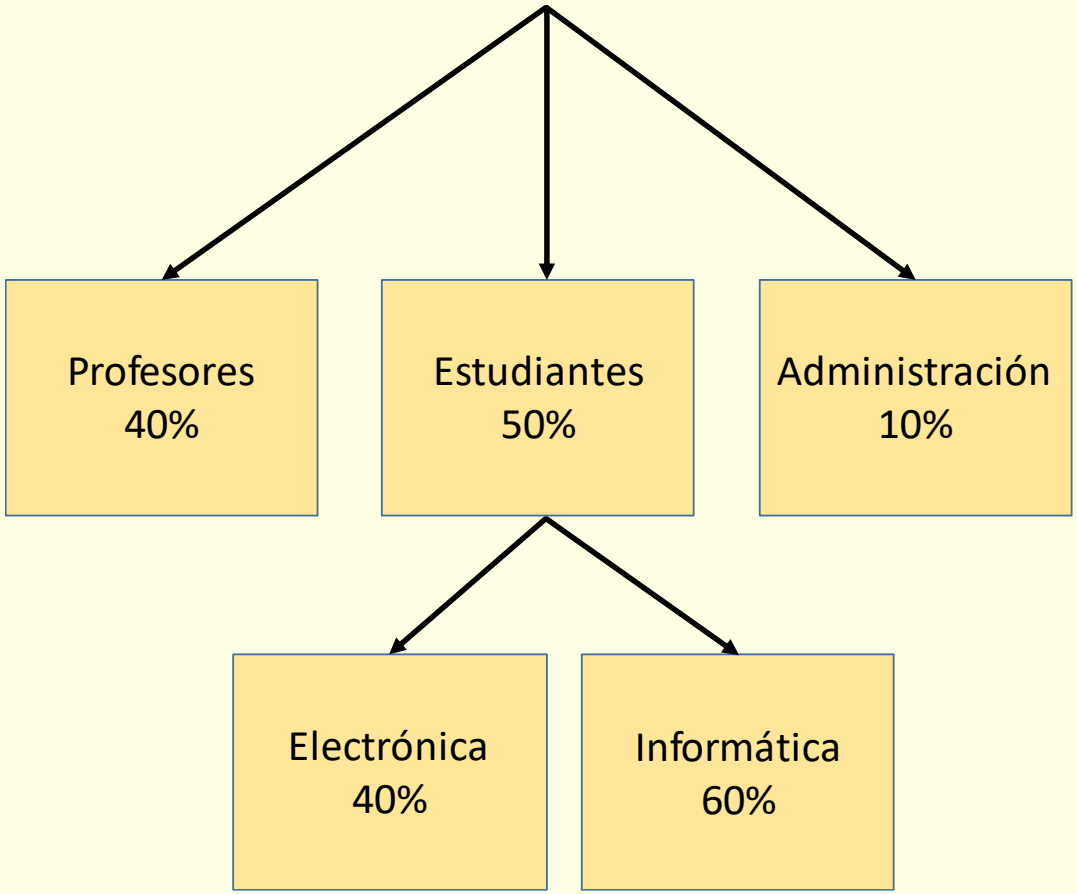
Reparto equitativo: ¿para quién?

- ¿Para grupos, usuarios, aplicaciones, procesos o *threads*?
- Ejemplo con solo 3 niveles: usuarios (U y V), procesos y *threads*
  - U ejecuta aplicación con 1 proceso que incluye 1 *thread* (t1)
  - V ejecuta 3 procesos
    - El primero solo tiene un *thread* (*t2*).
    - El segundo también incluye un único *thread* (*t3*).
    - El tercero tiene dos *threads* (*t4* y *t5*).
  - Equidad para:
    - *Threads*: cada *thread* 20% de UCP (PTHREAD_SCOPE_SYSTEM)
    - Procesos: t1, t2 y t3 25%; t4 y t5 12,5% (PTHREAD_SCOPE_PROCESS)
    - Usuarios: t1 50%; otros *threads* 12,5%
- Conveniencia de esquema general configurable
  - P.e. investigadores, profesores, estudiantes, personal admin.
- Planificadores con esa flexibilidad: ***Fair-share schedulers***

Fair-share schedulers



Fair-share schedulers: grupos de procesos



$$\%\text{ uso de Estudiantes de Informática } = \frac{60}{100} * \frac{50}{100} = 30\%$$

Soporte de múltiples clases de procesos

- 2 tipos de algoritmos: propósito general (*best-effort*) vs tiempo real
  - Dentro de los primeros: por prioridades o equitativos
- Algoritmos de propósito general basados en prioridades
  - favorece procesos con uso intensivo E/S, interactivos y urgentes
    - + técnicas para lograr cierto grado de equidad y evitar inanición
- Algoritmos de prop. general con esquema equitativo proporcional
  - servicio equitativo extendido a jerarquía de grupos de procesos
    - + técnicas favorece proc. uso intensivo E/S, interactivos y urgentes
- Algoritmos de tiempo real
  - garantizan requisitos temporales de los procesos
    - ignorando equidad, uso intensivo de entrada/salida o interactividad
- Convivencia de planificación de tiempo real y de propósito general
  - Extensión de prioridades vs. planificación jerárquica
- Uso de planificador de procesos para planificar máquinas virtuales

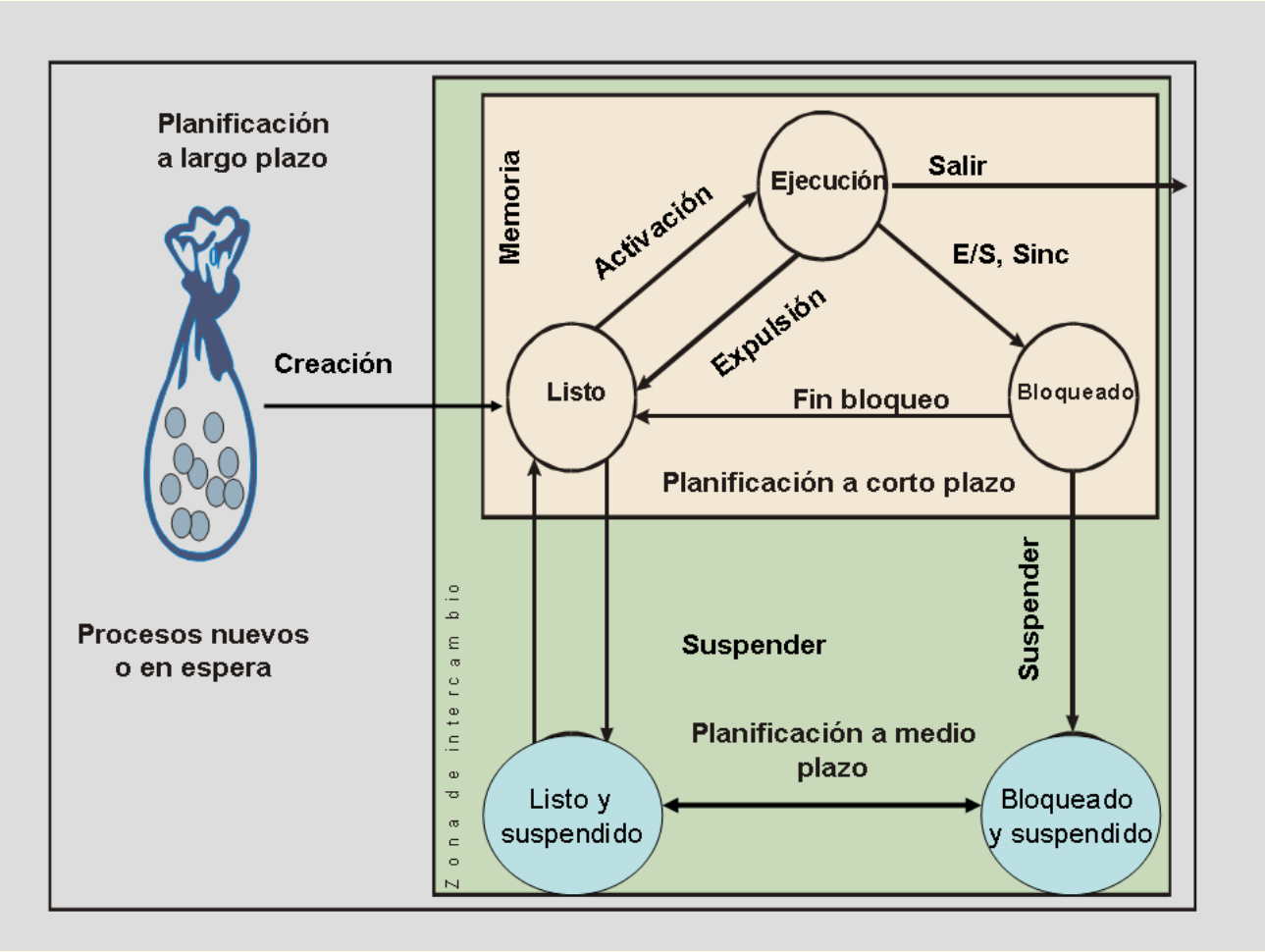
Objetivos de la planificación del procesador

- Optimizar el comportamiento del sistema
- Diferentes parámetros (a veces contrapuestos)
  - Minimizar t. espera puede empeorar uso del procesador
    - Más cambios de contexto involuntarios
- Parámetros por proceso: minimizar
  - **Tiempo de ejecución**
  - **Tiempo de espera**
  - **Tiempo de respuesta**
- Parámetros globales: maximizar
  - **Uso del procesador**
  - **Tasa de trabajos completados**
  - **Equidad**

Niveles de planificación

- A largo plazo (planificador de trabajos)
  - Control de entrada de trabajos al sistema
  - Característico de sistemas por lotes
  - Corresponde al control de admisión en planificación de t. real
  - No presente habitualmente en SO de propósito general
- A medio plazo
  - Control carga (suspensión procesos) para evitar hiperpaginación
  - Habitual en SS.OO. de propósito general
- A corto plazo: el planificador propiamente dicho
  - Qué proceso en estado de listo usa el procesador

Niveles de planificación



Anidamiento de planificadores

- Planificador: asigna entidades planificables a entidades computación
  - Entidad de computación puede ser planificable en otro nivel
- 3 niveles: en sistema con hilos de usuario y hipervisor tipo I
  - Planificador de hilos de usuario
    - Hilos usuario del proceso sobre hilos núcleo asignados al mismo
    - Distintos modelos de correspondencia: 1:1, M:1, M:N
    - Incluido en biblioteca o *runtime* de un lenguaje
  - Planificador de hilos de núcleo
    - Hilos de núcleo del SO sobre UCPs virtuales asignadas al mismo
    - Englobado en el SO
  - Planificador de UCPs virtuales
    - UCPs virtuales sobre UCPs físicas asignadas a esa máquina virtual
    - Forma parte del hipervisor

Puntos de activación

- Puntos del SO donde puede invocarse el planificador:

CCV

1. Proceso en ejecución finaliza voluntaria o involuntariamente
2. Proceso realiza llamada que lo bloquea
3. Proceso produce excepción que lo bloquea (p.e. fallo de página)
4. Proceso realiza llamada que cede el procesador (*sched_yield*)
5. Proceso cambia afinidad estricta y no puede seguir
6. Proceso realiza llamada declarándose menos importante

CCI

7. Proceso cambia afinidad estricta de otro proceso
8. Proceso realiza llamada cambiando importancia de otro proceso
9. Proceso realiza llamada que desbloquea proceso más importante
10. Interrupción desbloquea proceso más importante
11. Proceso crea otro proceso más importante
12. Interrupción de reloj marca fin de rodaja de ejecución

- Depende de cada algoritmo

Algoritmos de planificación

- Dos tipos de algoritmos:
  - no expulsivos: solo C.C. voluntarios
  - expulsivos: además C.C. involuntarios
    - basta con un único punto de activación expulsivo
  - No confundir con núcleo expulsivo o no expulsivo
    - Núcleo expulsivo sin sentido si planificación no expulsiva
- Vamos a estudiar:
  - Primero en llegar primero en ejecutar (FCFS)
  - Primero el trabajo más corto (SJF/SRTF)
  - Turno rotatorio
  - Planificación basada en prioridades
  - Cola multinivel con realimentación
  - Lotería
  - Planificación basada en tiempo virtual

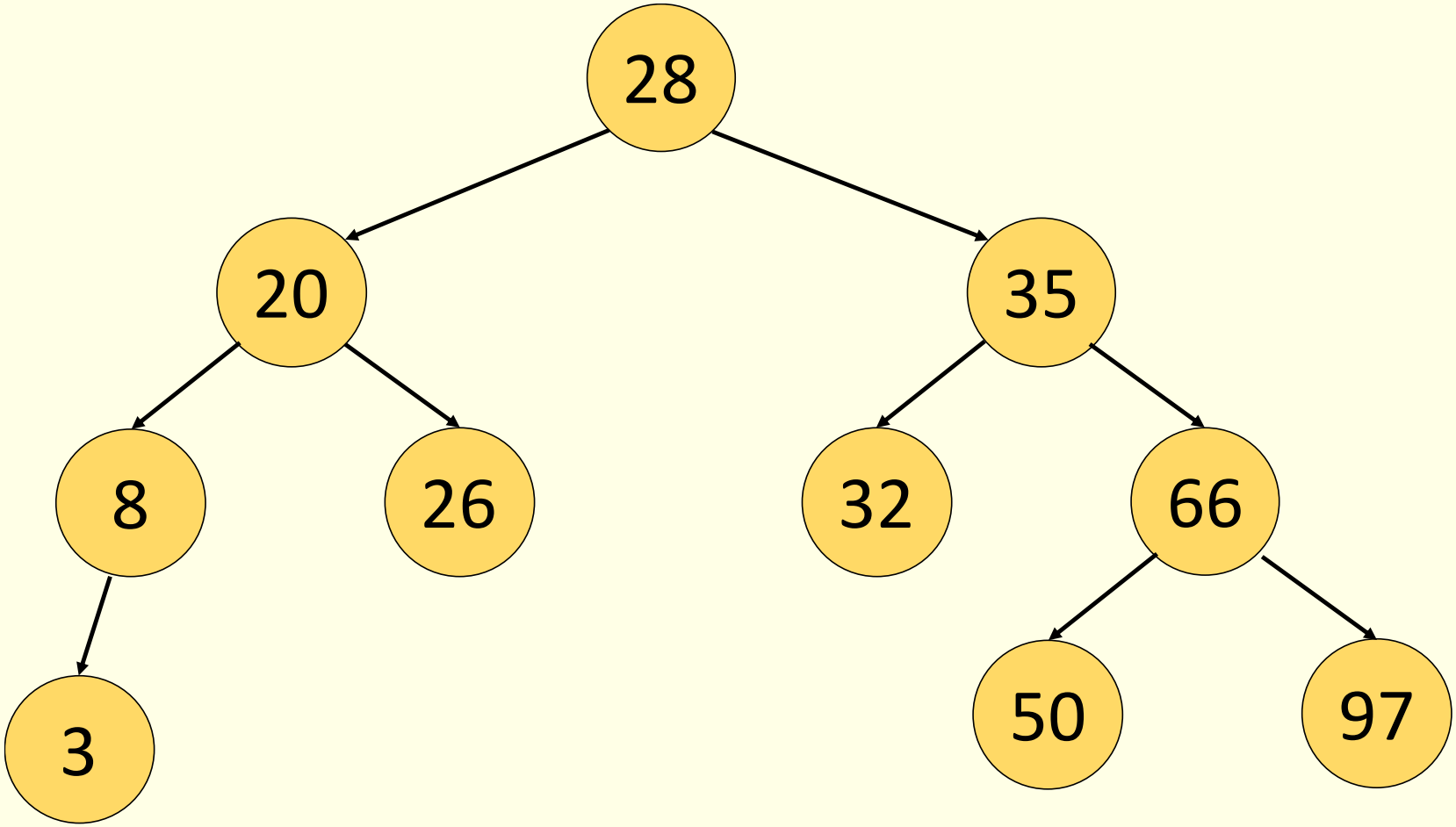
Primero en llegar primero en ejecutar (FCFS)

- Selección: Proceso que lleva más tiempo en cola de listos
- Algoritmo no expulsivo (puntos de activación de 1 a 5)
- Fácil de implementar (inserción, borrado y selección: O(1))
  - Cola de listos gestionada en modo FIFO
- Efecto convoy: procesos *I/O bound* detrás proceso *CPU bound*
  - Como turismos detrás de un camión
- Satisfacción de objetivos generales:
  - ✓ Optimizar uso
  - ↓ Minimizar tiempo de espera
  - ↓ Ofrecer reparto equitativo proporcional (extendido a grupos)
  - ↓ Proporcionar grados de urgencia

Primero el trabajo más corto (SJF)

- Selección: Proceso listo con próxima ráfaga UCP más corta
- Versión no expulsiva (puntos de activación de 1 a 5)
- Versión expulsiva: 1º el de menor tiempo restante (**SRTF**)
  - se activa si pasa a listo proceso con ráfaga < tiempo restante actual
    - puntos de 8 a 11
- ¿Cómo se conoce a priori la duración de la próxima ráfaga?
  - Estimación τ a partir de anteriores t (p.e. media exponencial)
    - $\tau_{n+1} = \alpha \times t_n + (1-\alpha) \times \tau_n = \sum_{i=0}^{n}((1-\alpha)^i \times \alpha \times t_{n-i});\ 0 < \alpha < 1$
- Implementación:
  - Lista no ordenada: O(1) inserción/borrado pero O(N) selección
    - si ordenada, selección O(1) pero inserción/borrado O(N)
  - Árbol binario: O(log N) inserción/borrado/selección
- Puede producir inanición
- Punto fuerte: Minimizar tiempo de espera

SJF/SRTF



Turno rotatorio (*Round Robin*, RR)

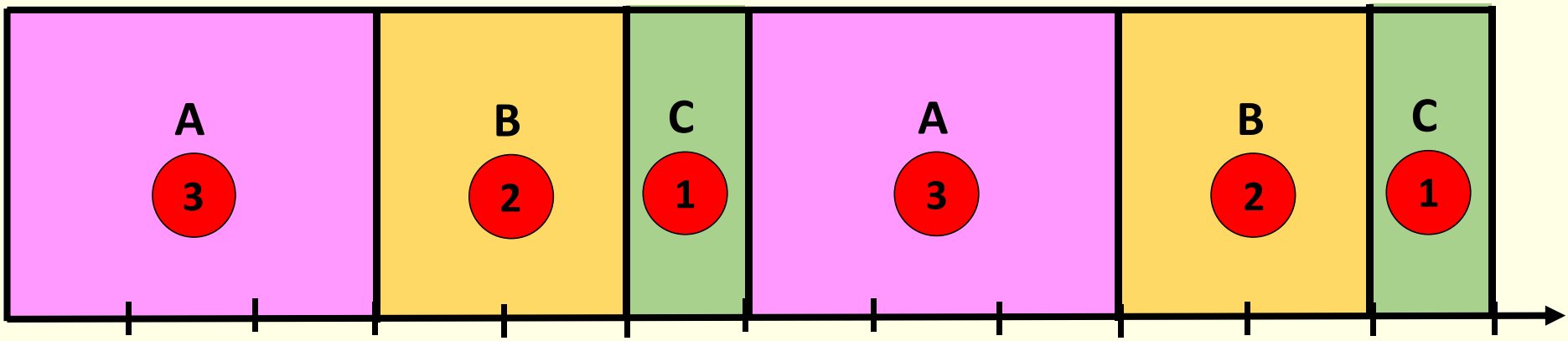
- FCFS + plazo (rodaja o cuanto)
- Algoritmo expulsivo pero solo con fin del cuanto (punto 12)
- Reparto equitativo (si N procesos, cada uno 1/N del procesador)
  - En una **ronda** se ejecuta una rodaja de cada proceso listo
- Tiempo de respuesta acotado (nº procesos listos * tam. rodaja)
  - Pero tiempo medio de ejecución peor que en FCFS
- Tamaño rodaja: grande (→FIFO) vs. pequeño (sobrecarga)
  - Más pequeño → menor t. respuesta y mejor equidad
  - Configurable (Windows cliente 20 ms. y servidor 120 ms.)
- Tamaño rodaja de proceso puede variar durante vida del proceso
  - Windows triplica rodaja de proceso al pasar a primer plano
- Si proceso no consume su rodaja, cuando vuelva a ejecutar:
  - Otorgar la parte de la rodaja que no consumió
  - Asignar rodaja completa: favorece procesos uso intensivo E/S
  - Rodaja + tiempo no consumido: **turno rotario con déficit**
- Punto fuerte: Ofrecer reparto equitativo

Turno rotatorio con pesos

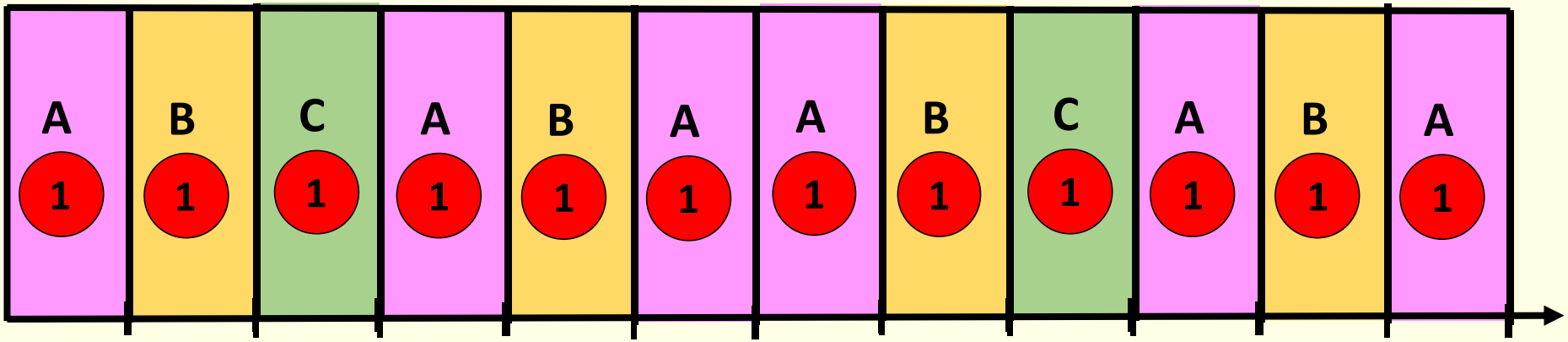
- Tamaño rodaja proporcional a peso del proceso
  - $Grado\ uso\ de\ UCP\ del\ Proceso_i = \frac{Rodaja_i}{\sum_{j=1}^{n} Rodaja_j} = \frac{Peso_i}{\sum_{j=1}^{n} Peso_j}$
  - Importancia de procesos con peso en vez de con prioridad
- Implementación alternativa:
  - Rodaja fija pero nº rodajas asignadas depende del peso
  - Menos tiempo de espera pero más cambios de contexto
- Punto fuerte: Ofrecer reparto equitativo proporcional
- Pero deficiencias a la hora de acotar el sesgo en la equidad
  - Ejemplo: 1 proceso (A) peso 10 y 10 procesos de peso 1
  - En ventana de tamaño *X*, A debería obtener ≈ 50% UCP
  - Para X=10 en ambas implementaciones, uso de UCP de A
    - Oscila entre 0% y 100%
- Planificación de grupos usando turno rotatorio
  - Generalizar asignación rodajas a jerarquía grupos de procesos
  - **Turno rotatorio jerárquico**: soporte *fair-share scheduling*

Turno rotatorio con pesos: alternativas

Turno rotario con rodajas variables

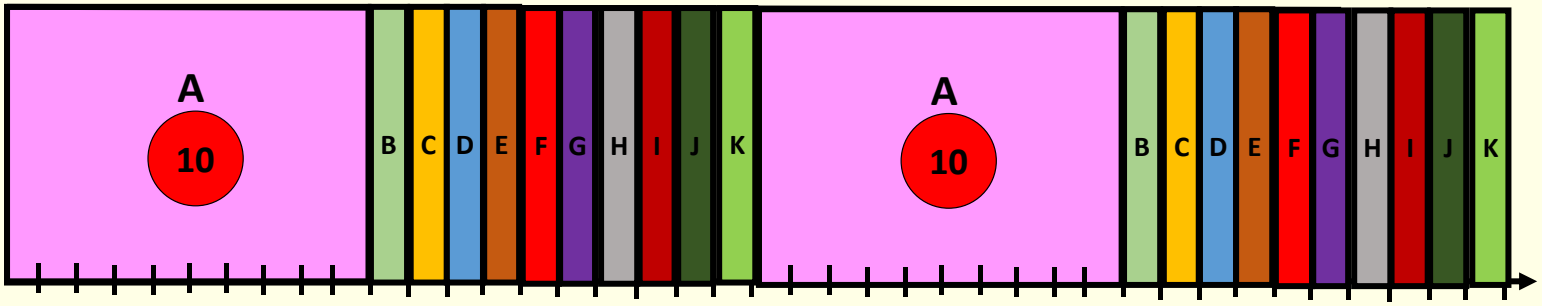


Turno rotario con misma rodaja

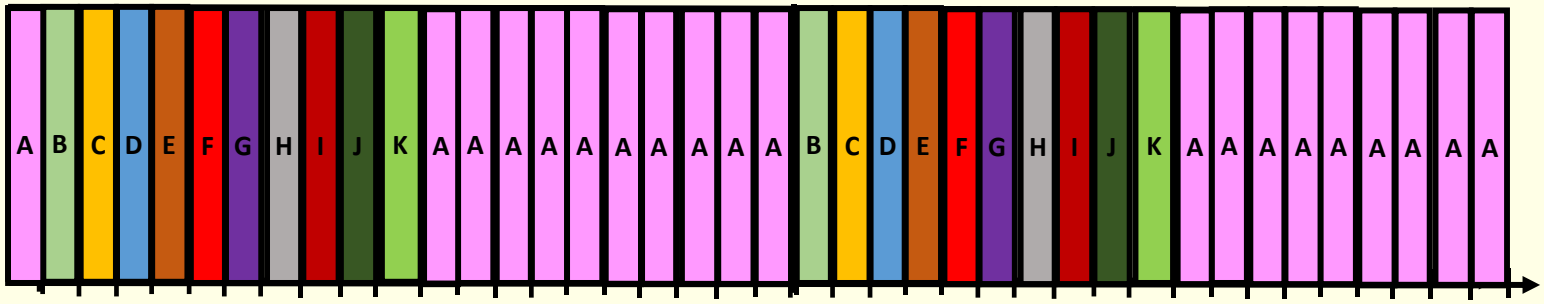


Turno rotatorio con pesos: desviación de equidad

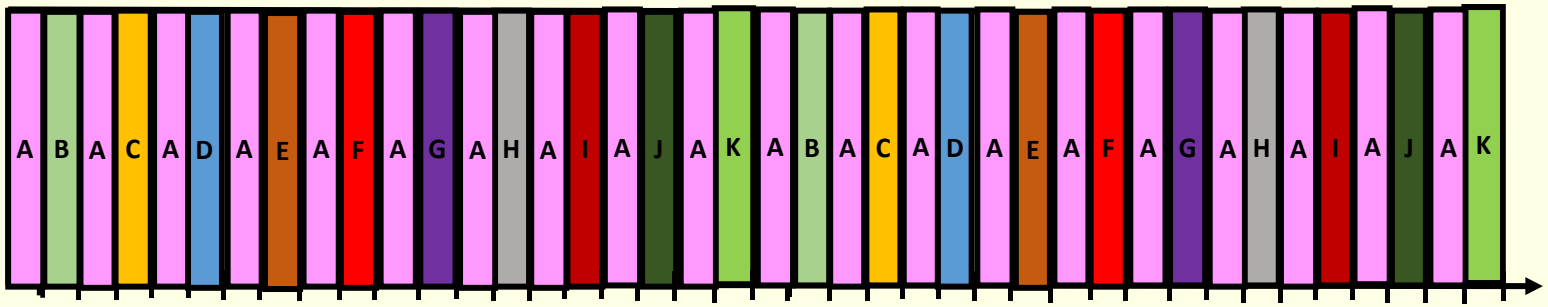
Turno rotario con rodajas variables



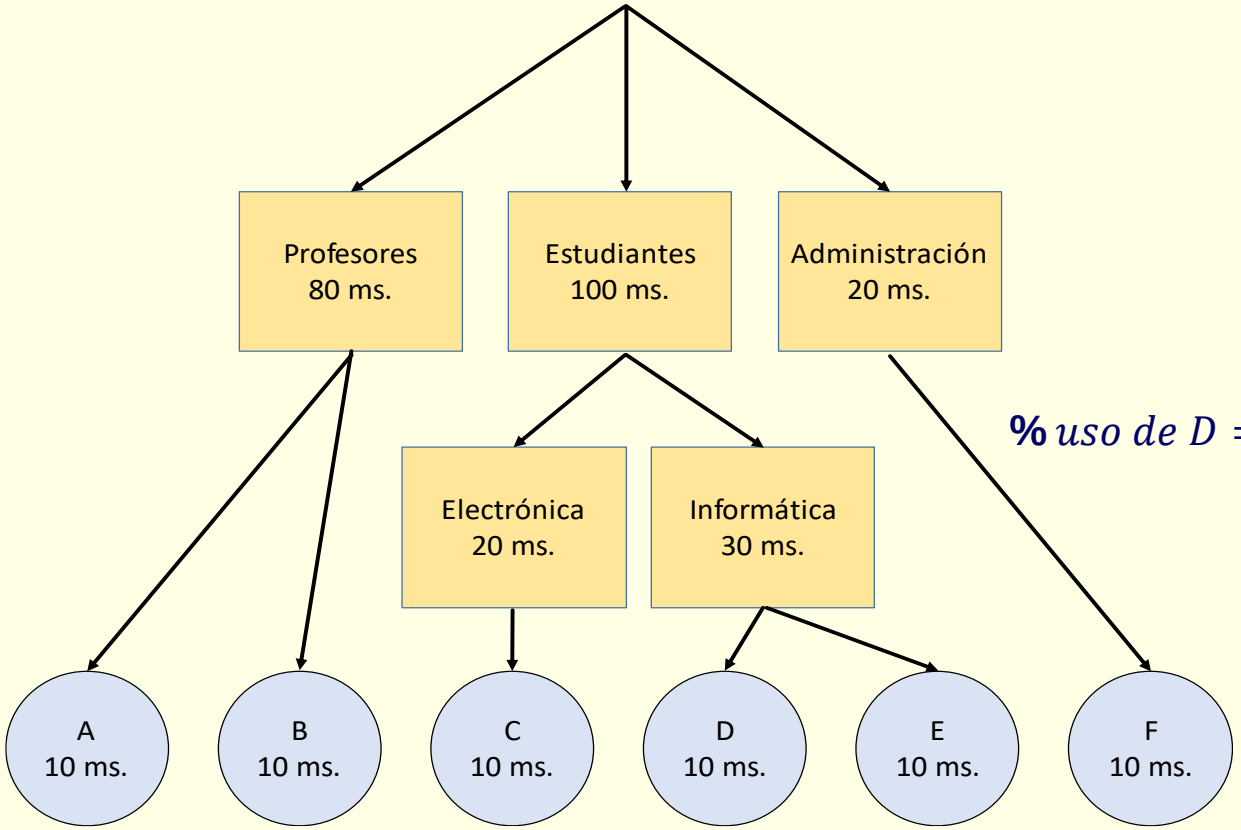
Turno rotario con misma rodaja



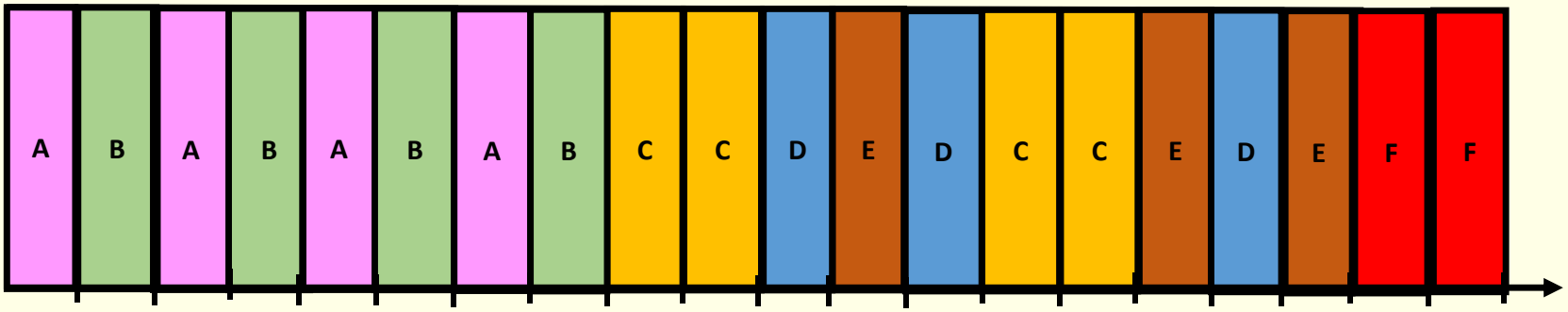
Solución ideal



Turno rotatorio jerárquico



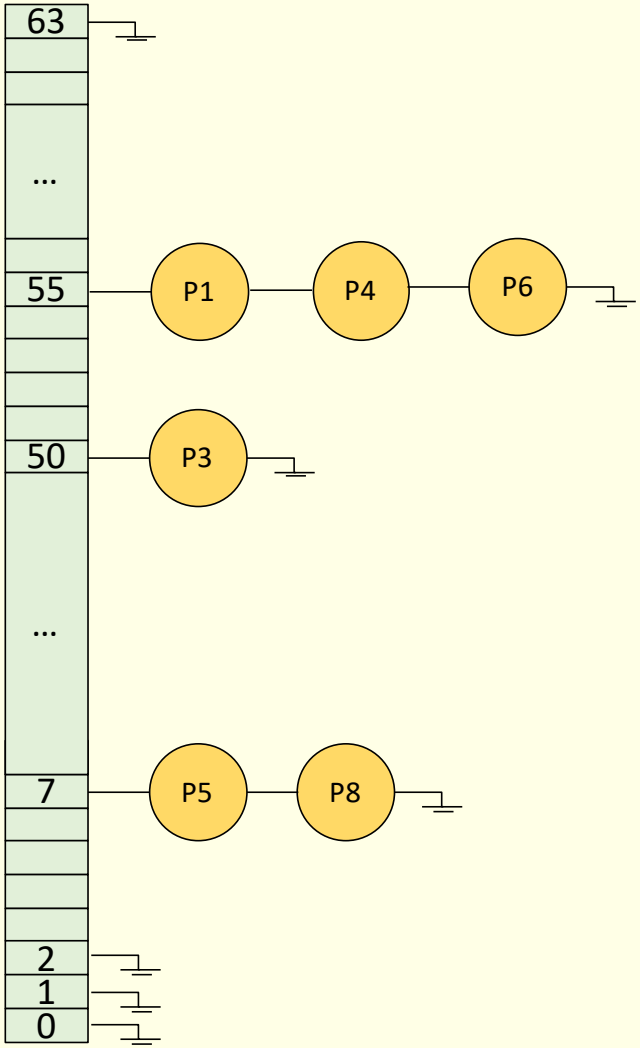
$$\%\, uso\ de\ D = \frac{10}{10+10} * \frac{30}{20+30} * \frac{100}{80+100+20} = 15\%$$



Planificación por prioridad

- Cada proceso tiene asignada una prioridad
- Selección: Proceso en cola de listos que tenga mayor prioridad
- Versión no expulsiva (puntos 1 a 6) y expulsiva (puntos 7 a 11)
- Prioridad estática: solo puede cambiarla el usuario
- Prioridad dinámica: cambia a partir de base según ejecuta proceso
  - Para favorecer, por ejemplo, a procesos con uso intensivo de E/S
- Puede producir inanición: posible solución "Envejecimiento"
  - Prioridad aumenta si proc. listo no ha ejecutado en un intervalo
- ¿Qué hacer con los procesos de la misma prioridad?
  - FCFS, turno rotatorio…
- Implementación:
  - Nº de prioridades fijo y reducido: **cola multinivel** (O(1))
  - Si no: igual que SJF: O(N) si usa lista; O(log N): árbol binario
- Punto fuerte: Proporcionar grados de urgencia
  - Mal esquema para la equidad;
  - ¿minimizar tiempo espera?→ cola multinivel con realimentación

Planificación por prioridad: cola multinivel



0000000010000100000000000000000000000000000000000000000000010000000

Planificación por prioridad: clases de procesos

Parámetro	Linux	Windows
Clases de planificación	3 (POSIX)	2
Prioridades normales (dinámicas)	40; de -20 a 19	15; de 1 a 15
Prioridades de t real FCFS (fijas)	99; de 1 a 99	16; de 16 a 31
Prioridades t real con rodaja (fijas)	99; de 1 a 99	–
Orden de prioridades normales	Alta → Baja	Baja → Alta
Orden de prioridades de t. real	Baja → Alta	Baja → Alta

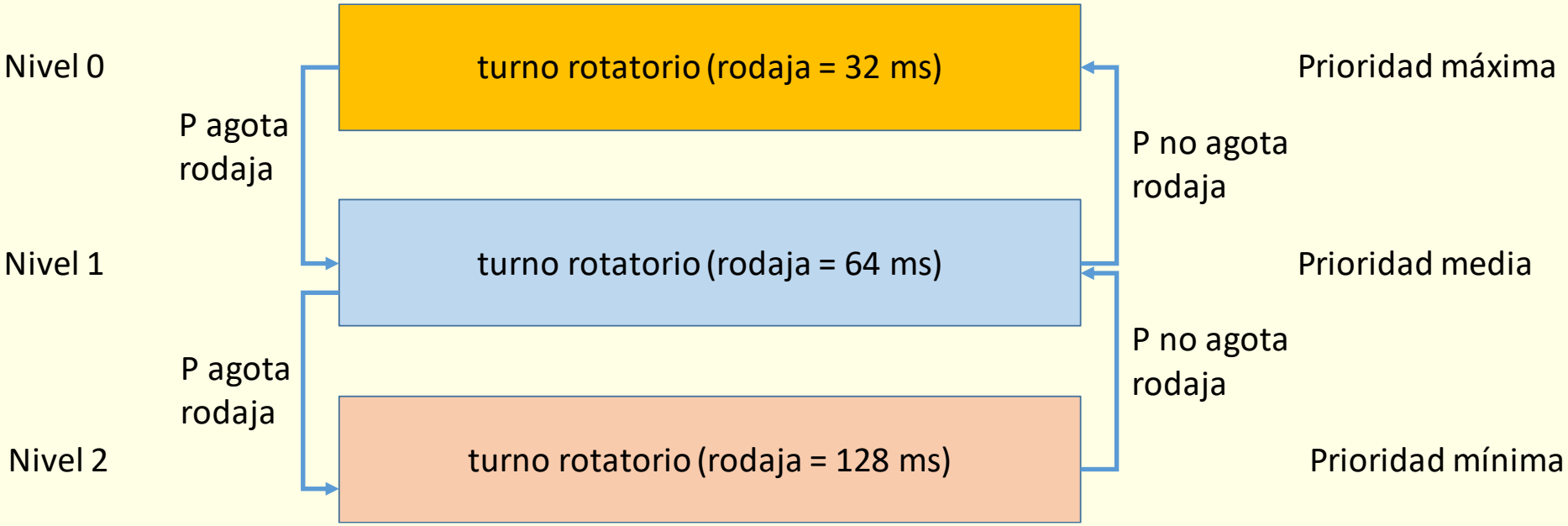
Planificación por prioridad: modo de operación

- UCP libre: selección 1er proceso lista mayor prioridad no vacía
  - Continúa ejecutando hasta fin, bloqueo, cesión o expulsión
    - o termine su rodaja si turno rotatorio entre iguales
- Si bloqueo: selección siguiente proc. lista de esa misma prioridad
  - Si vacía, lista no vacía con mayor prioridad y selecciona 1º
  - Cuando bloqueado vuelva a listo, al final de lista de su prioridad
- Si cede UCP (o fin rodaja), al final de lista de su prioridad
  - Selección 1er proceso de esa lista, que podría ser el mismo
- Si proceso en ejecución solicita disminuir su prioridad
  - Pasa a principio de la lista de listos de esa prioridad
    - Si ya no más prioritario, planificador selecciona el correspondiente
- Si proceso en ejecución pide aumentar prioridad de otro
  - este 2º pasa al final de la lista de listos de esa nueva prioridad.
- Si aparece proceso más prioritario que el actualmente en ejecución
  - Por creación de nuevo, por aumento prioridad o por desbloqueo
    - se expulsa proceso en ejecución, que queda 1º lista de su prioridad.

Cola multinivel con realimentación

- Idea propuesta en 1962 (Fernando J. Corbató, MIT)
- Cola multinivel + t. rotario entre iguales + favorecer *I/O bound*
  - proceso se le sube o baja de nivel por su comportamiento.
- Habitual rodaja diferente para cada nivel (< para + prioridad)
- Proceso consume rodaja de su nivel, pasa a cola nivel inferior
  - proceso con uso intensivo UCP desciende hasta nivel mínimo
- Para ascender, si no consume rodaja de su nivel, se le promociona
- Para evitar la inanición, proceso periódico de envejecimiento
  - Aumenta prioridad proceso que no recibe UCP en cierto periodo
- Puede fijarse nº máximo niveles que proceso asciende o desciende
  - Impide que se distorsione totalmente prioridad base original
- Usado en muchos sistemas reales (típico de familia UNIX)
  - ¿Cómo incorporar equidad a este esquema?

Colas multinivel con realimentación



Decay usage scheduling

- Variante de cola multinivel con realimentación
- Por cada proceso P_i se estima su promedio de uso UCP (U_i)
  - repercusión grado de uso puntual se reduce con paso del tiempo
- Cada int. reloj, si P_i en ejecución U_i++
- Periodo reducción (cada 1s), calcula U de **todos** los procesos O(N)
  - reduciéndolo (*decay*) por un cierto factor δ: $U_i = \frac{U_i\ previo}{\delta}$
  - recalcula prioridades efectivas usando siguiente fórmula
- Prioridad efectiva (nivel): mezcla prioridad base y uso UCP

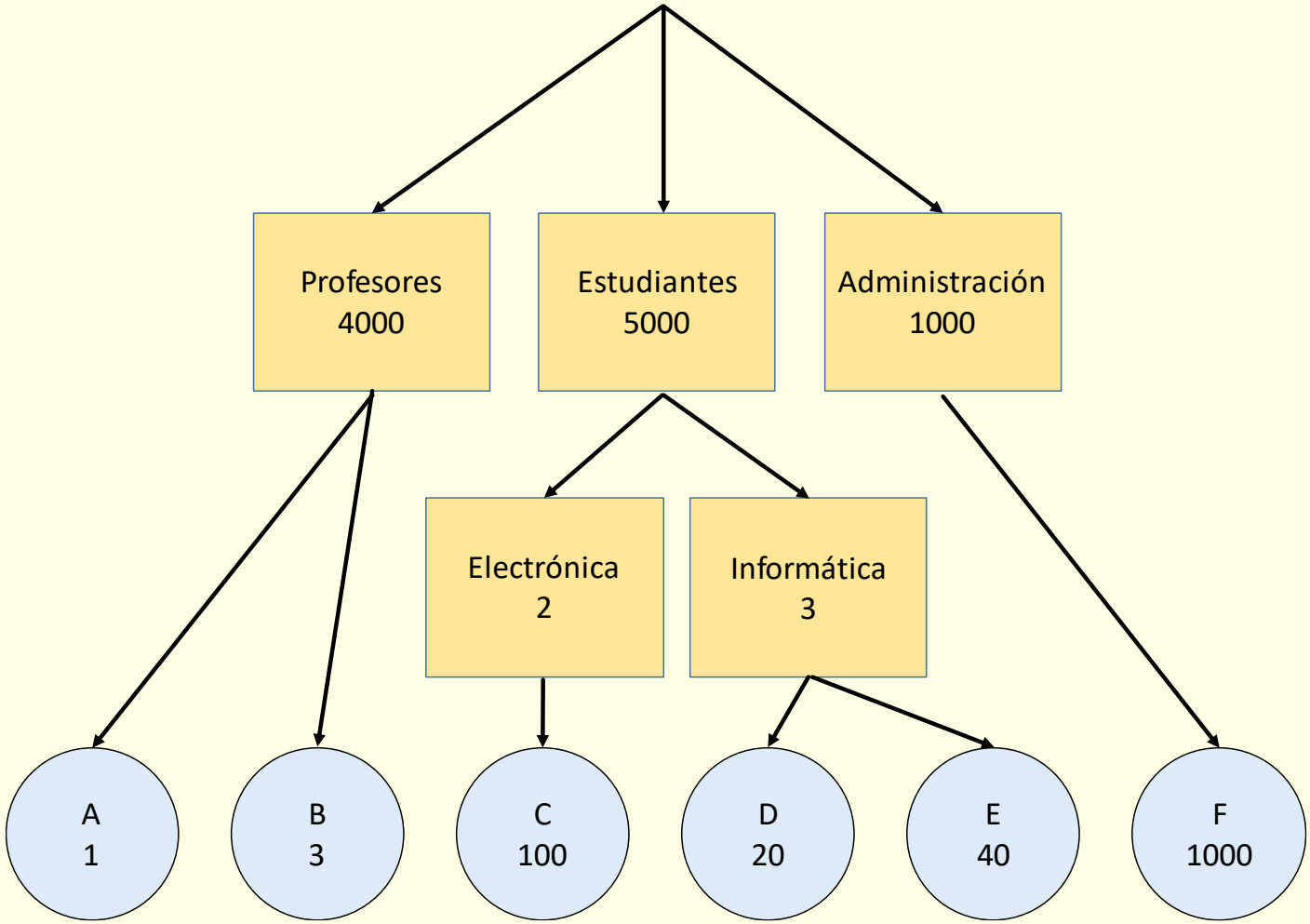
$$Prioridad\ efectiva\ de\ P_i = \alpha \times prioridad\ base\ de\ P_i + \beta \times uso\ de\ UCP\ de\ P_i$$

  - Prioriza procesos *I/O bound* y trata de evitar la inanición
- Media exponencial (similar SJF), secuencia reducciones con δ = 2:
  - $\frac{T_0}{2}; \left(\frac{T_1+\frac{T_0}{2}}{2}\right) = \frac{T_1}{2} + \frac{T_0}{4}; \left(\frac{T_2+\frac{T_1}{2}+\frac{T_0}{4}}{2}\right) = \frac{T_2}{2} + \frac{T_1}{4} + \frac{T_0}{8}$
  - Generalizando → $\sum_{j=0}^{n} \frac{T_j}{\delta^{n-j}}$

Planificación por lotería

- Cada proceso tiene uno o más billetes de lotería
- Selección: proceso que posee billete aleatoriamente elegido
- Versión expulsiva: rodaja de tiempo entre sorteos
- No inanición
- Importancia de los procesos: nº de billetes que poseen
  - Planificador equitativo proporcional
- Favorecer procesos con E/S e interactivos
  - Billetes de compensación si proceso no consume su rodaja
- Transferencia de billetes: p.e. un cliente a un servidor
- Posibilita *Fair-share scheduling*:
  - Concepto de conversión entre unidades monetarias
- Punto fuerte: Ofrecer reparto equitativo
  - Similar a turno rotatorio pero probabilístico

Planificación *fair-share* con algoritmo de la lotería



Cambios de unidades monetarias y cuota uso UCP

- Cálculo de equivalencia de C monedas nivel N a nivel del padre :
  - $Conversión\ de\ nivel\ N\ a\ nivel\ del\ padre\ (C) = \frac{C \times nº\ de\ monedas\ del\ padre}{nº\ total\ de\ monedas\ en\ nivel\ N}$

- Cálculo para proceso D con 20 unidades monetarias:
  - $Conversión\ de\ nivel\ informática\ a\ estudiantes\ (20) = \frac{20*3}{20+40} = 1$
  - $Conversión\ de\ nivel\ estudiantes\ a\ nivel\ superior\ (1) = \frac{1*5000}{2+3} = 1000$

- Que conlleva un grado de utilización del procesador:
  - $Grado\ de\ uso\ del\ procesador\ de\ D = \frac{1000}{4000+5000+1000} = 0{,}1\ (10\%)$

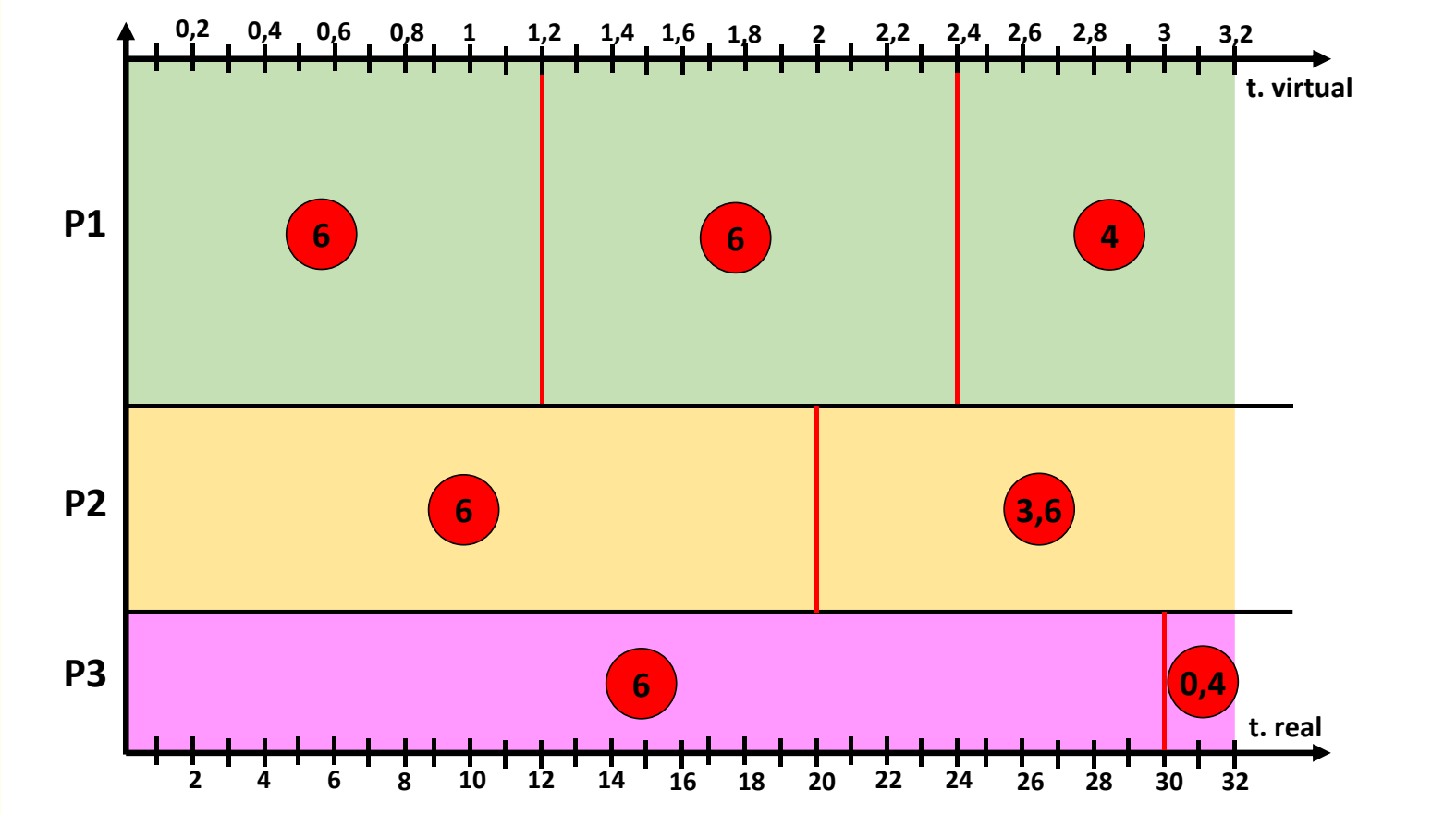
Planificación basada en tiempo virtual

- Algoritmo irrealizable *Generalized processor sharing* (GPS)
- Suponga UCP utópica que consigue equidad perfecta
  - t. infinitesimal ejecuta nº inst cada proceso proporcional al peso

- Tiempo virtual: 1 unidad de tiempo virtual en GPS es igual a:
  - Tiempo requerido para ejecutar $\sum_{j=1}^{N} Peso_j$ unidades de t. real
  - $Tiempo\ virtual\ en\ t\ unidades\ de\ tiempo\ real = \frac{t}{\sum_{j=1}^{N} Peso_j}$

- Tiempo virtual invariante independiente del nº de procesos
  - $X = t.real\ que\ tarda\ ejecutar\ P_i\ un\ tiempo\ t = t \times \frac{\sum_{j=1}^{N} Peso_j}{Peso_i}$
  - $T.virtual\ en\ X\ unidades\ de\ tiempo\ real = \frac{X}{\sum_{j=1}^{N} Peso_j} = \frac{\left(t \times \frac{\cancel{\sum_{j=1}^{N} Peso_j}}{Peso_i}\right)}{\cancel{\sum_{j=1}^{N} Peso_j}} = \frac{t}{Peso_i}$

GPS: ejemplo

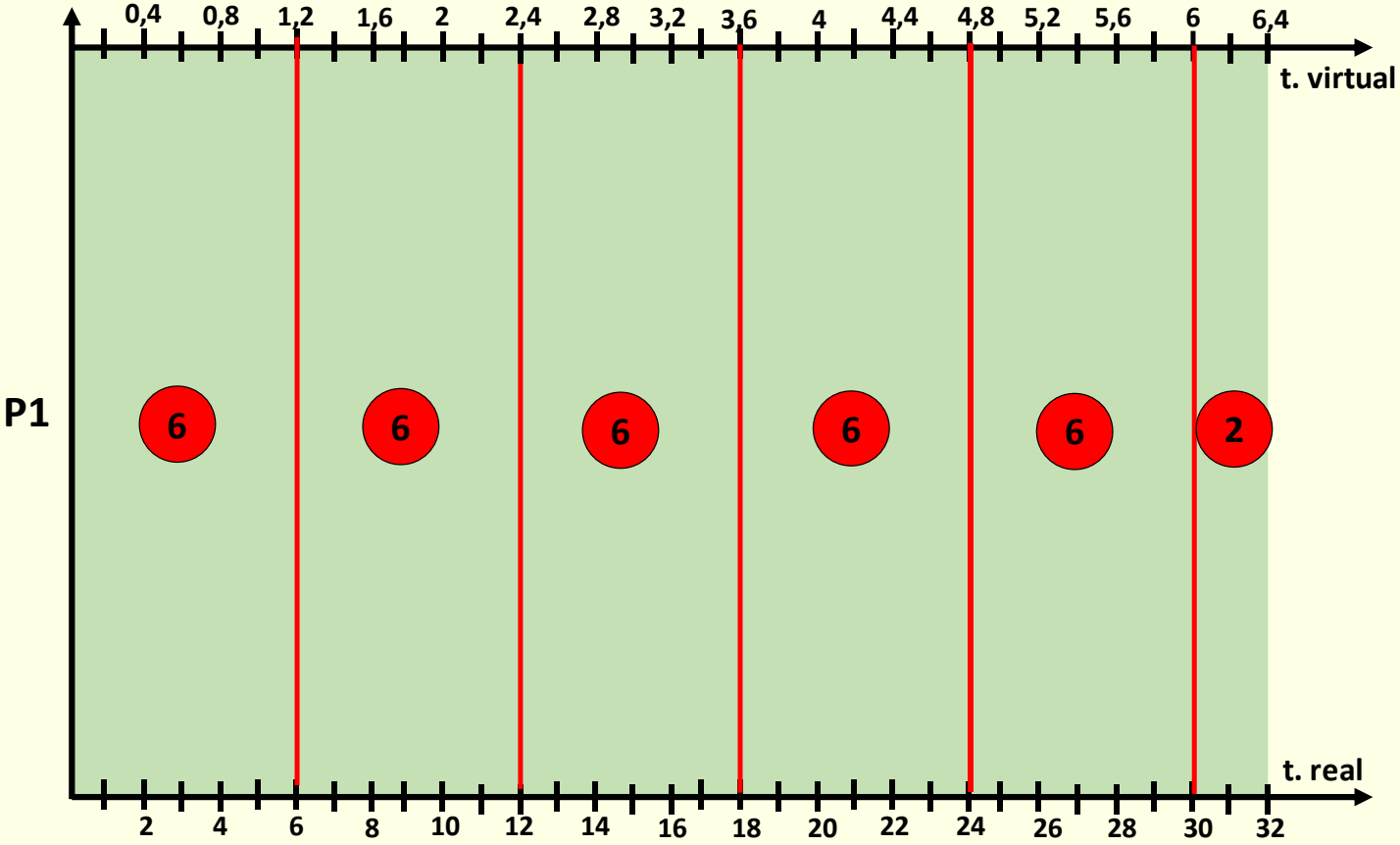


Rodaja de 6; P_1, P_2 y P_3 con pesos 5, 3 y 2

1 unidad tiempo virtual = 10 de tiempo real

Rodaja P_1 12 tiempo real = 1,2 tiempo virtual

GPS: Tiempo virtual invariante

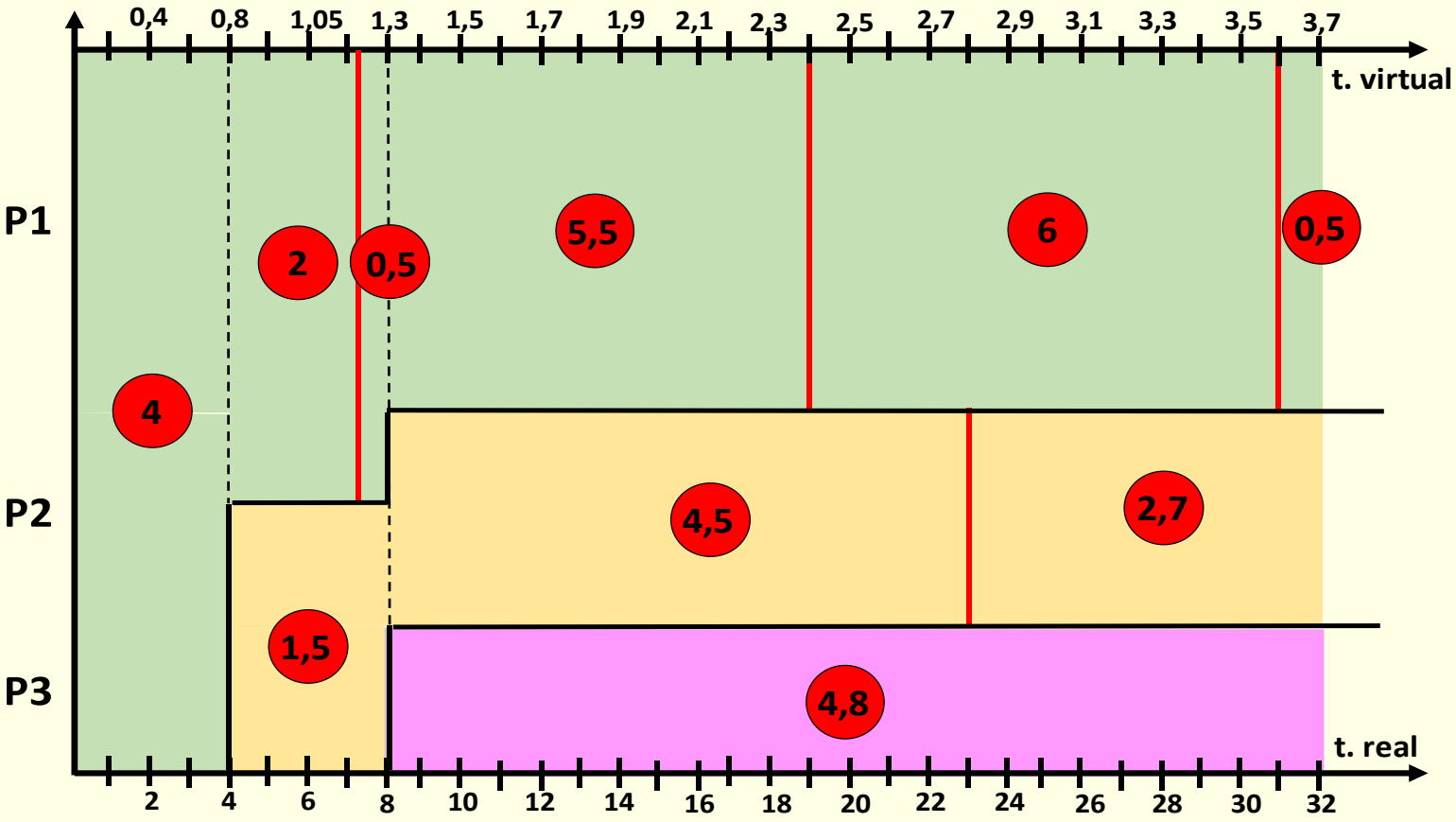


Rodaja de 6; P_1, P_2 y P_3 con pesos 5, 3 y 2

1 unidad tiempo virtual = 5 de tiempo real

Rodaja P_1 6 tiempo real = 1,2 tiempo virtual

GPS: con procesos escalonados



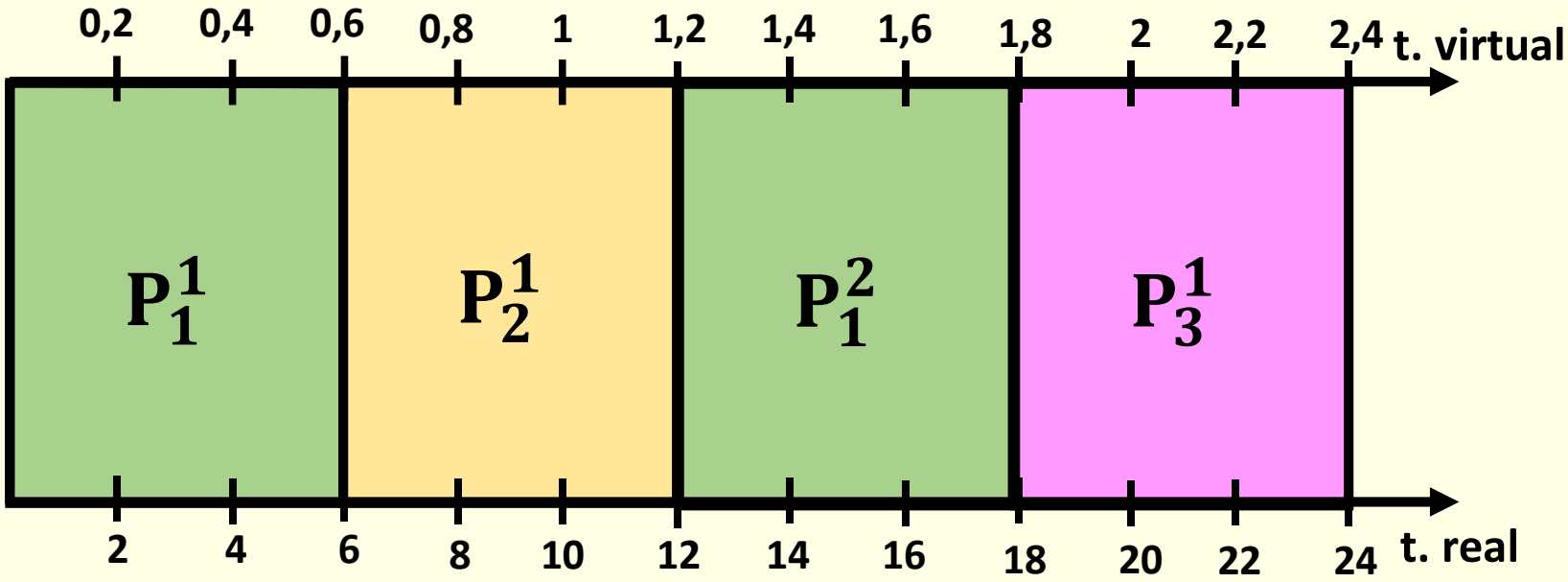
Rodaja de 6; P_1, P_2 y P_3 con pesos 5, 3 y 2
Inicio P_1, P_2 y P_3 en 0, 4 y 8

Planificación basada en GPS

- GPS no es factible. Algoritmos basados en GPS:

1. CFS Linux: mismo consumo t. virtual para todos los procesos
2. Ejecución tal que se completen rodajas en mismo orden que GPS
   - Seleccionar proceso tal que sea menor:
     - $t.virtual\ fin\ próxima\ rodaja\ P_i = t.virtual\ fin\ anterior + \frac{tam.próxima\ rodaja}{Peso_i}$
   - Si cede UCP: se usa tiempo consumido en vez de t. rodaja
   - Proceso nuevo: tiempo virtual inicial = el actual del sistema
   - Si se bloquea: varias alternativas
     - T. virtual cuando se bloqueó: acapararía la UCP
     - Tratarlo como un proceso nuevo
     - Favorecer procesos con uso intensivo de E/S (CFS)
   - Ej. 1º: P_1 termina en 1,2; 2,4; 3,6; 4,8; 6;…; P_2 en 2; 4; 6; 8; 10;…; P_3 en 3; 6; 9; 12; 15;…
     - Orden de ejecución: $P_1(1,2)$ $P_2(2)$ $P_1(2,4)$ $P_3(3)$ $P_1(3,6)$ $P_2(4)$ $P_1(4,8)$…
   - Aplicación directa no acota sesgo en el reparto equitativo: EEVDF

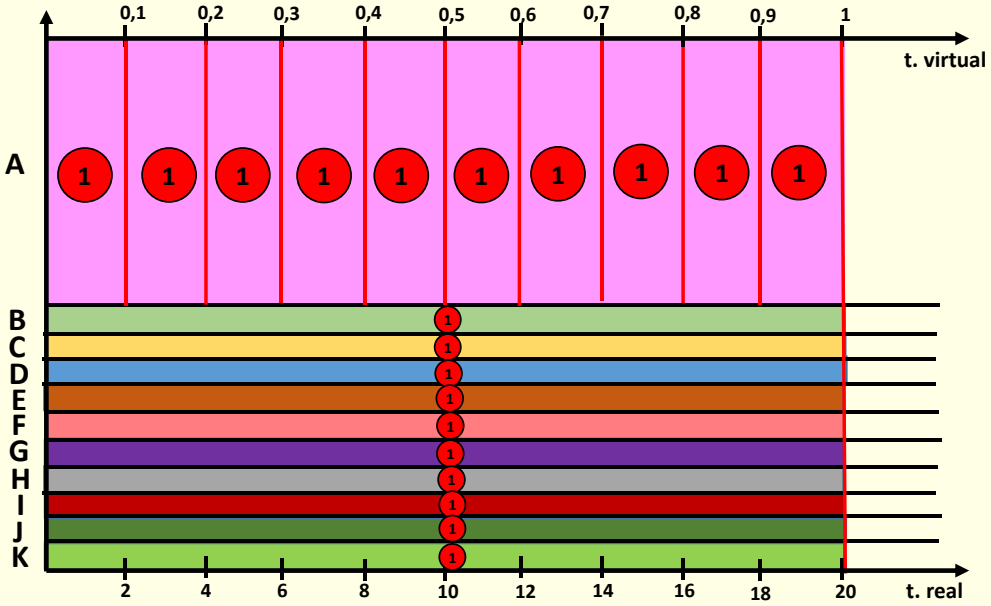
Planificación basada en tiempo virtual: ejemplo 1



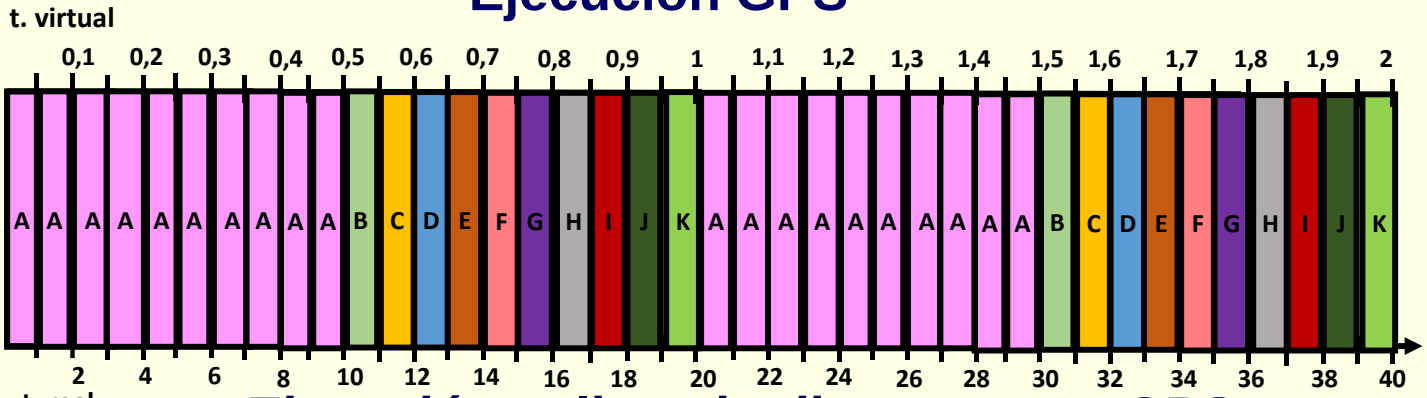
Rodaja de 6; P_1, P_2 y P_3 con pesos 5, 3 y 2

Sesgo en la equidad

1 proceso (A) con peso 10 y 10 procesos (de B a K) con peso 1



Ejecución GPS



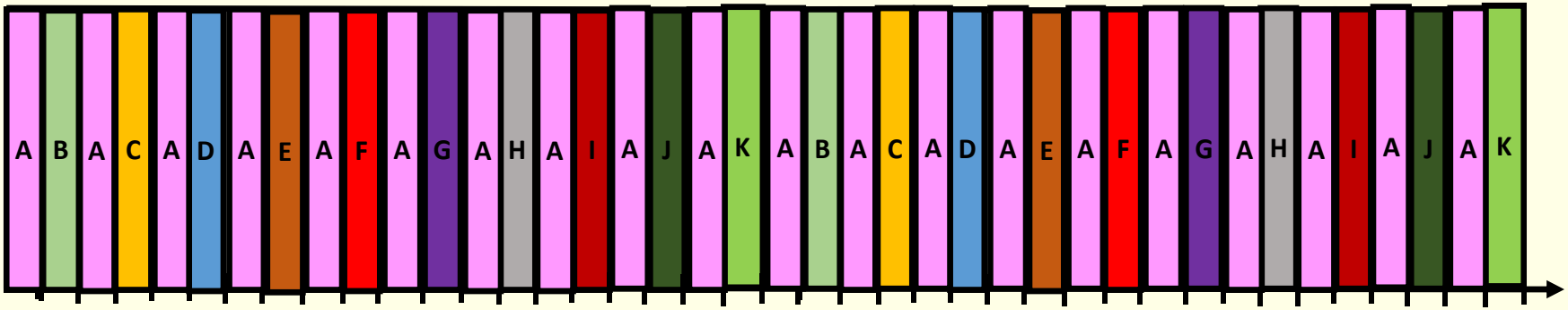
Ejecución aplicando directamente GPS

Earliest Eligible Virtual Deadline First (EEVDF)

- Incorpora concepto de cuándo un proceso es *elegible*
- Selección de proceso a ejecutar tiene en cuenta 2 parámetros:
  - T. virtual de fin de próxima rodaja (como hasta ahora)
  - T. virtual de fin de la anterior (tiempo de *elegibilidad*)
    - El que se usó para seleccionarle la última vez que ejecutó
- Elige proceso con t. virtual fin rodaja más cercano pero entre *elegibles*
  - procesos t. virtual de fin de anterior ≤ t. virtual actual del sistema
- Si t. virtual fin de anterior de un proceso > t. virtual actual del sistema
  - Indica que el proceso tiene un sesgo positivo y debe esperar
  - Ha completado su rodaja "antes de tiempo"

EEVDF: Reducción del sesgo

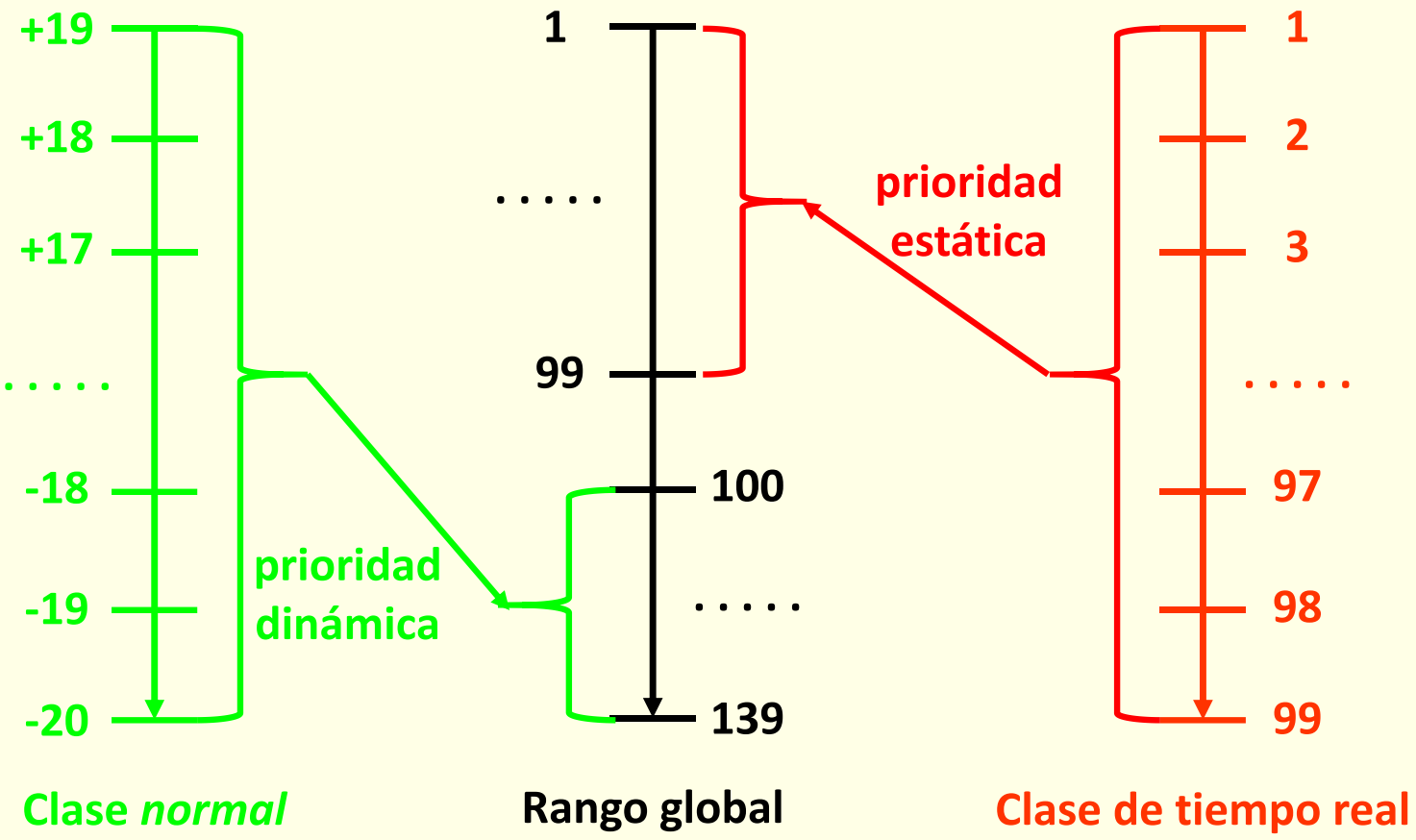
- Traza ejemplo 11 procesos: 1 con peso 10 y 10 con peso 1; rodaja=1
  - tv=0: todos elegibles
    - se selecciona A por fin de rodaja más cercano = 1/10 = 0,1
    - Resto fin de rodaja = 1/1 = 1
  - tv=1/20=0,05: A termina rodaja
    - 2ª rodaja de A terminaría en t. virtual 0,2; 1ª rodaja de restantes en 1
    - Pero A no es elegible:
      - t. elegible A → fin anterior rodaja 0,1 > t.v. actual sistema 0,05
    - Selecciona B
- Sesgo en la equidad óptimo



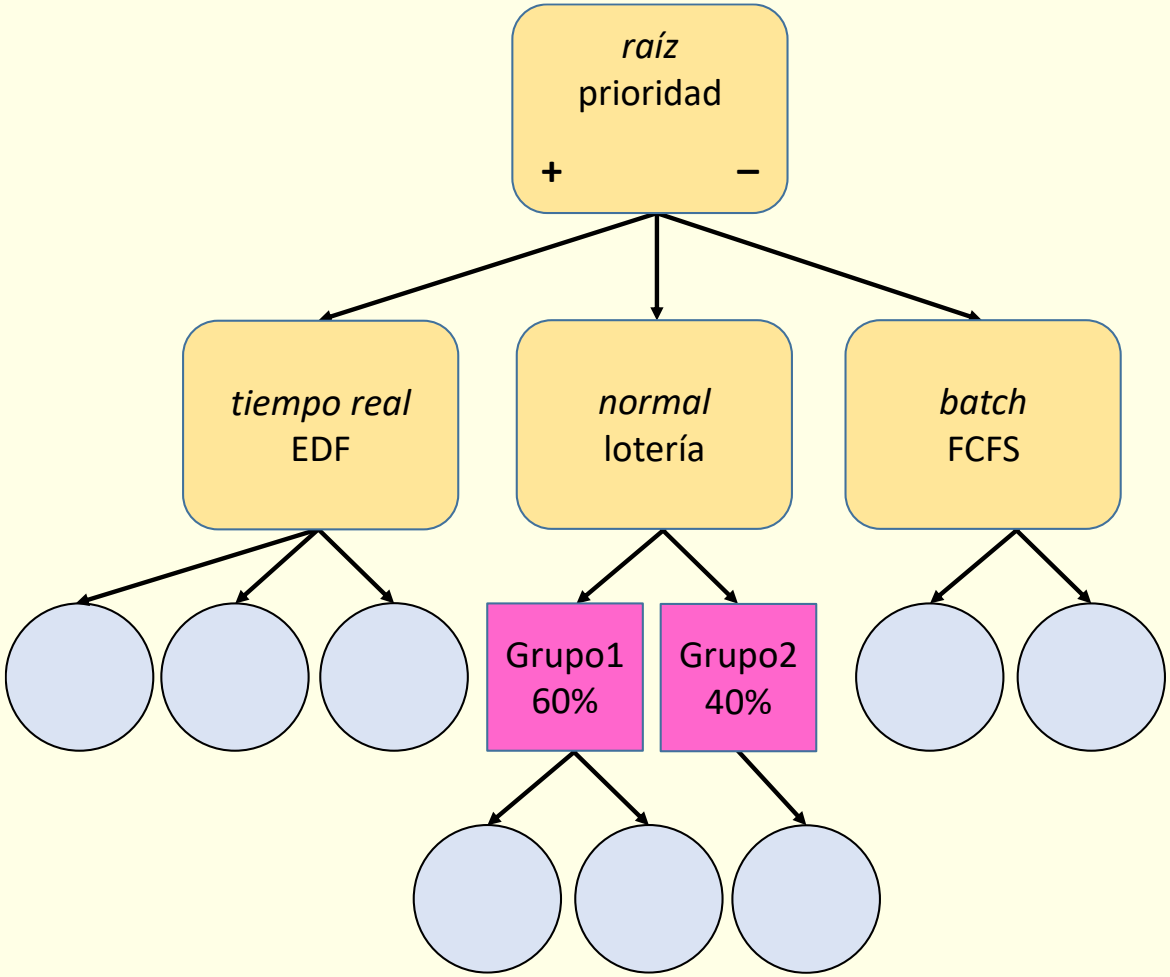
Planificación jerárquica

- SO propósito general mezcla de procesos con distintos requisitos
  - P.e. Procesos de tiempo real, interactivos y *batch*
- ¿Cómo gestionarlos?
  - Esquema integrado de prioridades vs. Planificación jerárquica
- Planificación jerárquica: Distinguir entre clases de procesos
  - Planificador nivel superior: reparte UCP entre clases
  - Planificador de cada clase
- Generalizable a múltiples niveles
- Proporciona aislamiento entre planificadores clases de procesos
  - Cada uno reparte el tiempo que le otorga el nodo padre
  - Dificultad con procesos de tiempo real
    - Solo aplicable si se asigna prioridad máxima a su clase
    - Si se asigna % de UCP pueden incumplir sus plazos

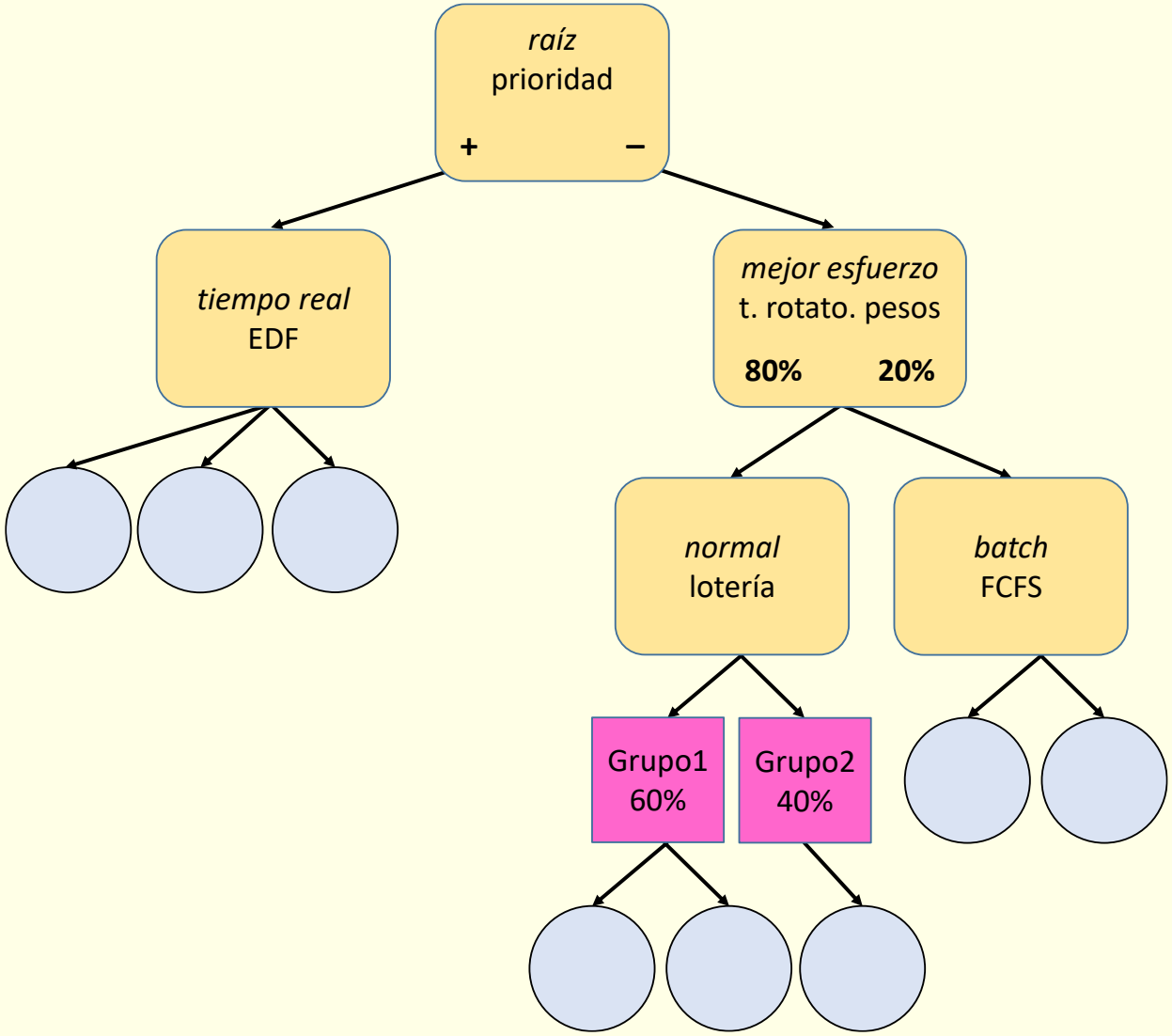
Gestión integrada de prioridades en Linux



Planificación jerárquica con dos niveles



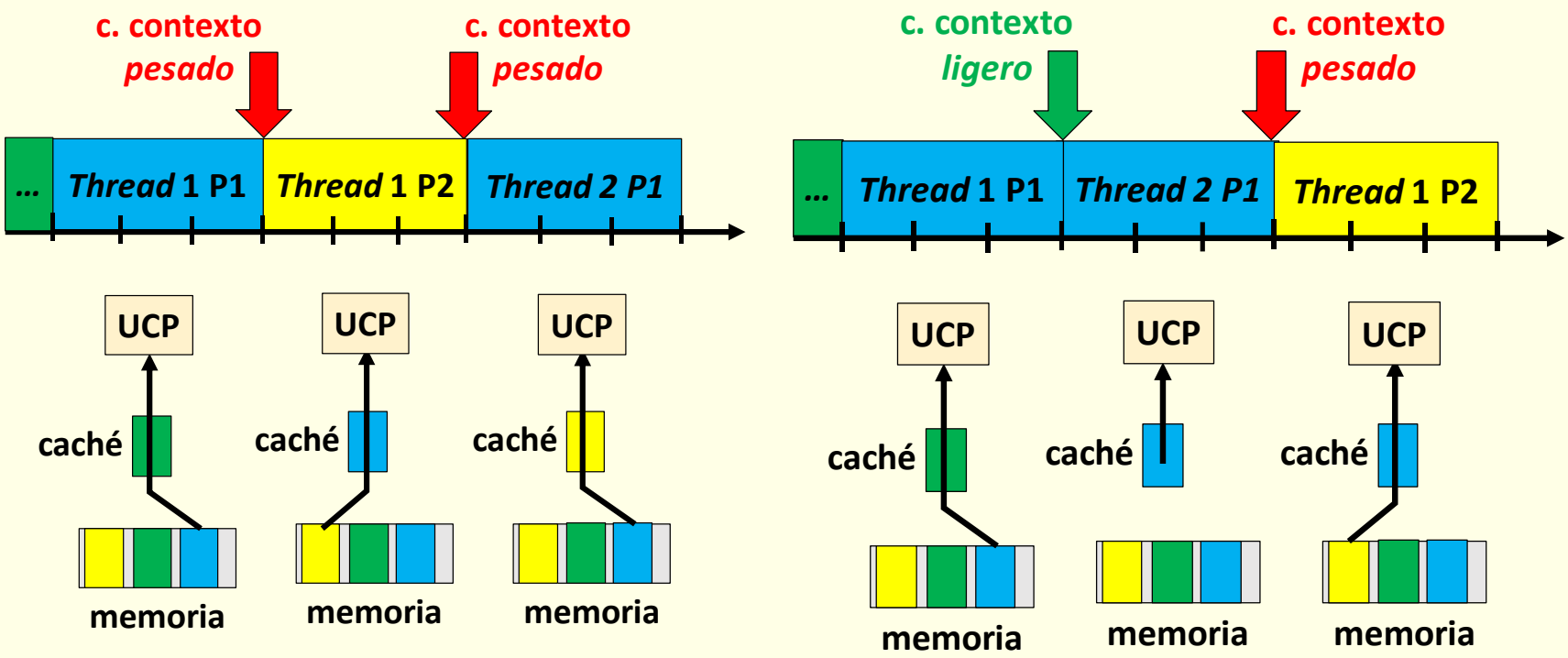
Planificación jerárquica con múltiples niveles



Otras consideraciones sobre la planificación

- Planificador componente independiente
  - Puede modificarse sin afectar a otras partes del SO
- Selección de mejor planificador puede ser polémica (Kolivas)
  - Difícil comparar rendimiento del planificador
- Debería evitarse parámetros de configuración mágico
  - Mejor autoajustable
- Debería ser escalable: no complejidad O(N)
- Técnicas de optimización específicas:
  - Cesión de prioridad, *handoff scheduling*…
  - Después de *fork* mejor que ejecute primero el hijo
    - Es probable que realice un *exec* y así se reduce gasto por COW
  - Optimizaciones en la planificación de *thread*

Optimización en la planificación de *threads*



Sistemas operativos avanzados

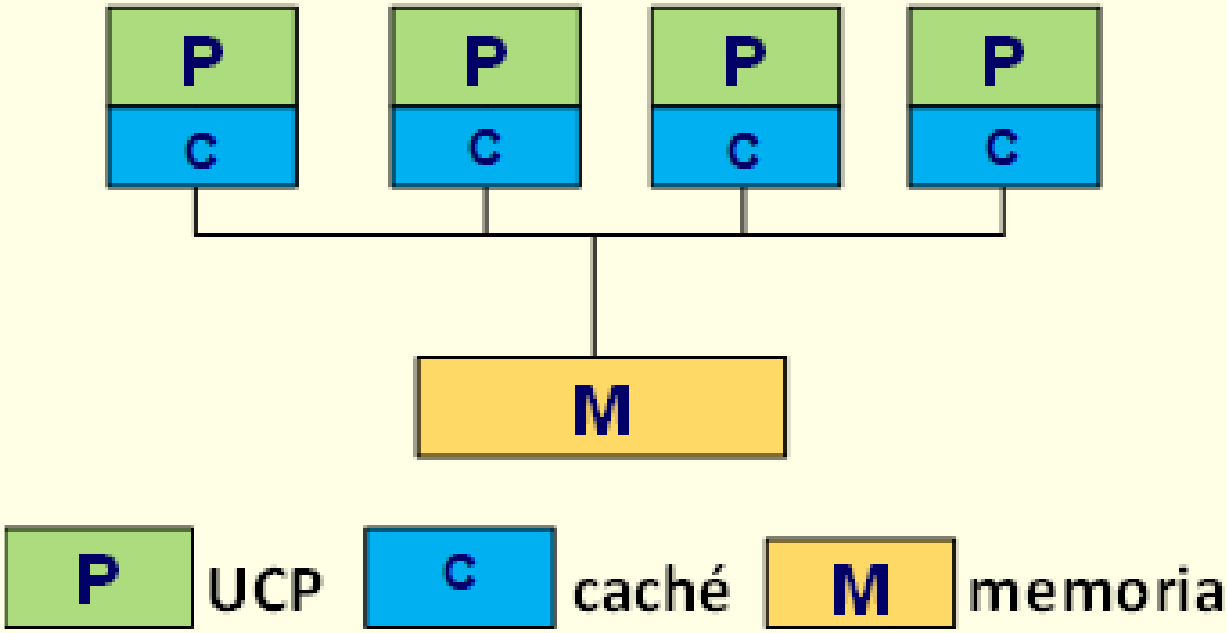
Planificación del procesador

2ª parte: planificación en multiprocesadores

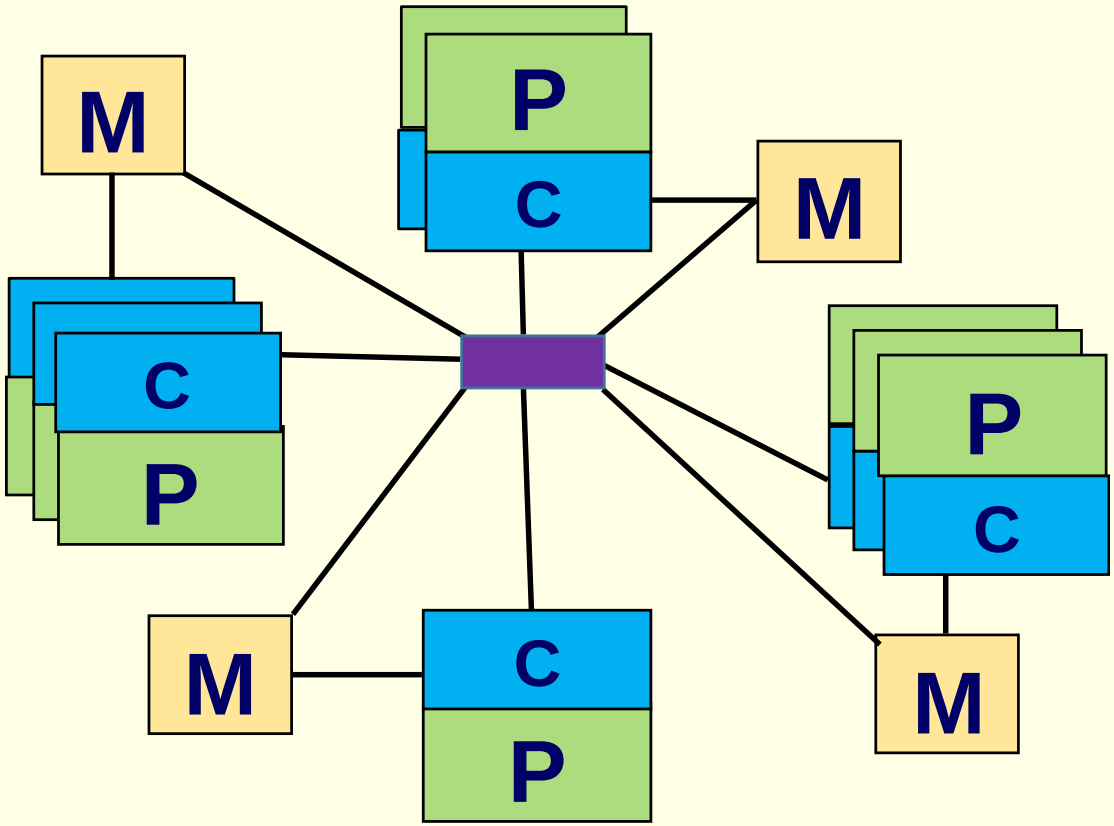
Planificación en multiprocesadores

- *Trivial*: *N* UCPs ejecutan *N* procesos elegidos por planificador
- No es tan fácil; hay que tener en cuenta:
  - Afinidad estricta
  - Afinidad natural
  - Multiprocesadores jerárquicos (SMT, CMP, NUMA…)
    - SMT: *Simultaneous Multithreading* (*hyperthreading*)
    - CMP: *Chip Multiprocessing* (*multicore*)
    - NUMA: *Non-Uniform Memory Access*
  - 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

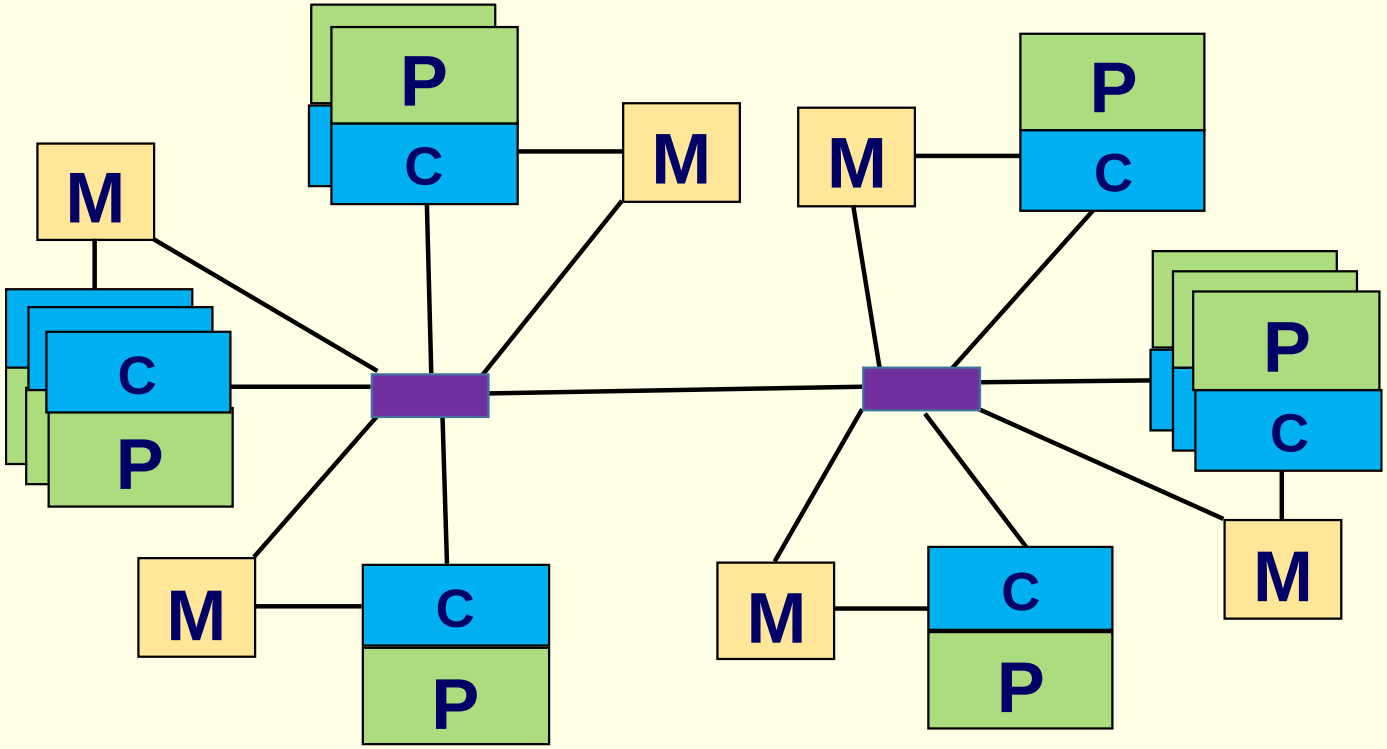
Evolución de los multiprocesadores: UMA



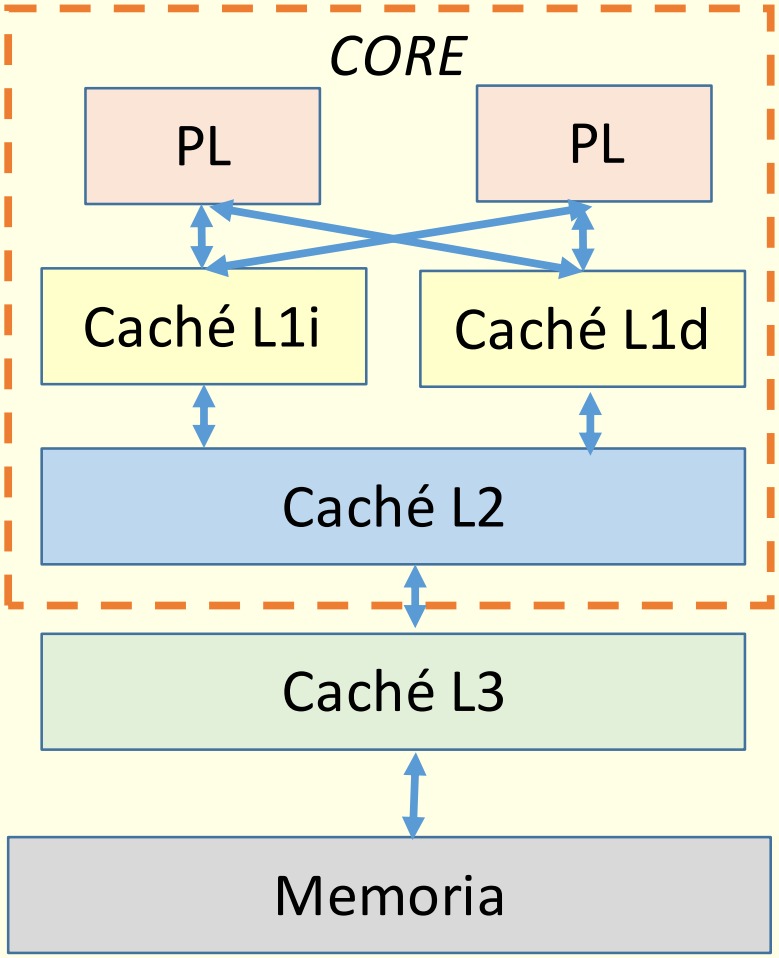
Evolución de los multiprocesadores: NUMA



Evolución de los multiprocesadores: NUMA

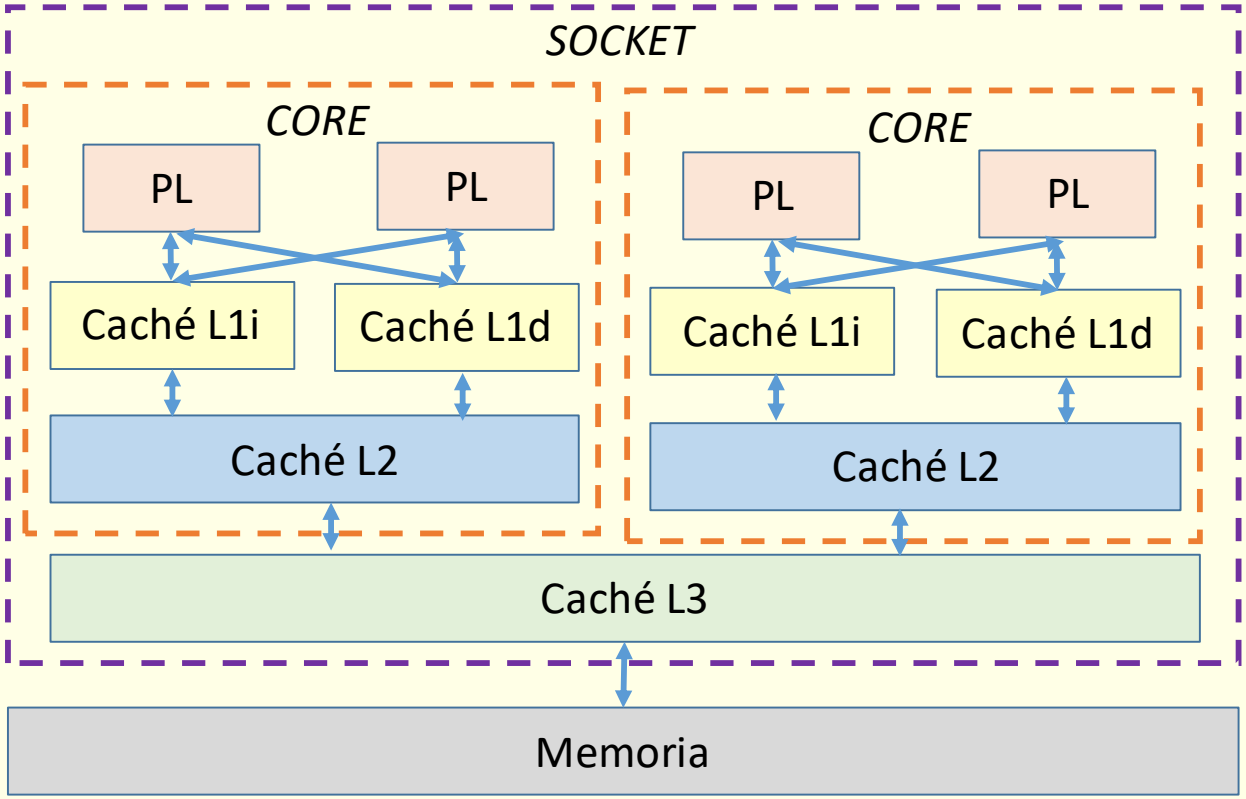


Evolución de los multiprocesadores: SMT



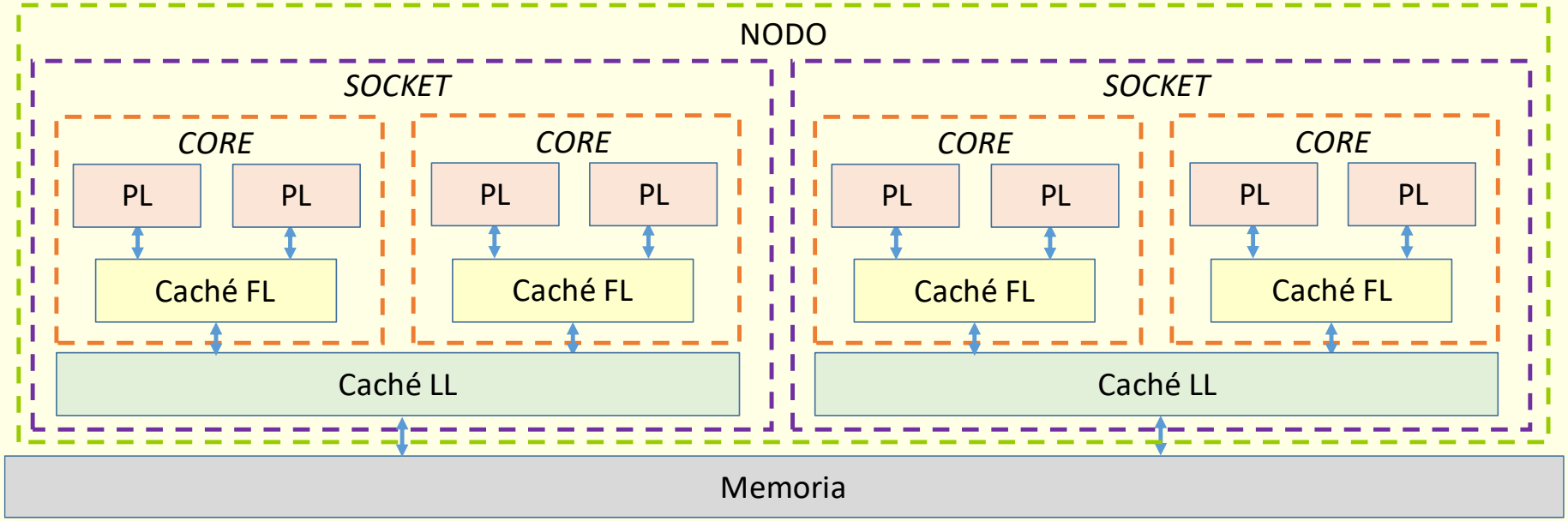
Paralelismo intra-chip

Evolución de los multiprocesadores: SMT y CMP



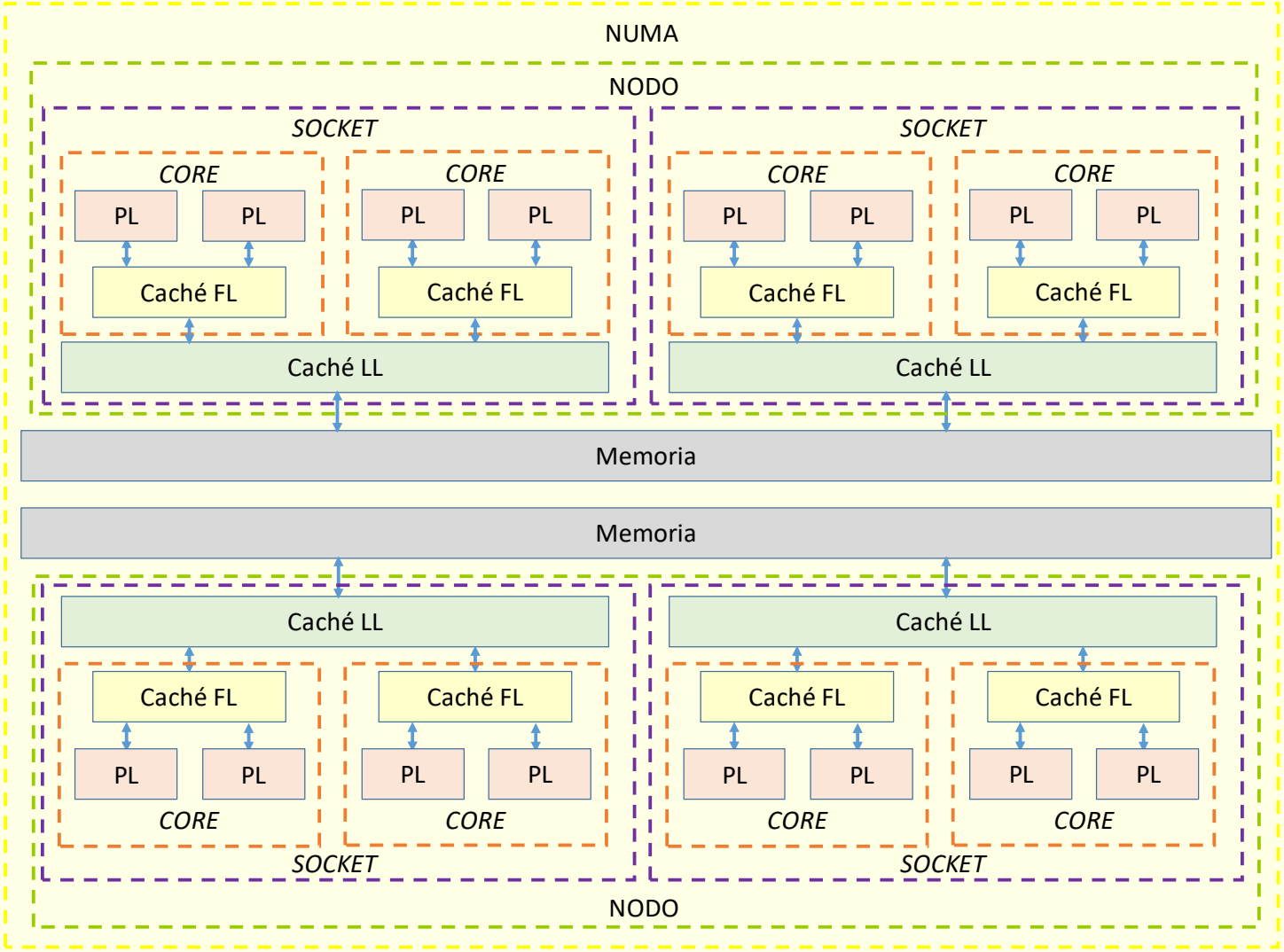
Paralelismo intra-chip

Evolución de los multiprocesadores: *multisocket*



Para simplificar y generalizar:
Solo FL (*First level*) y LL (*Last Level*)
y sin segregar entre instrucciones y datos

Evolución de los multiprocesadores: NUMA

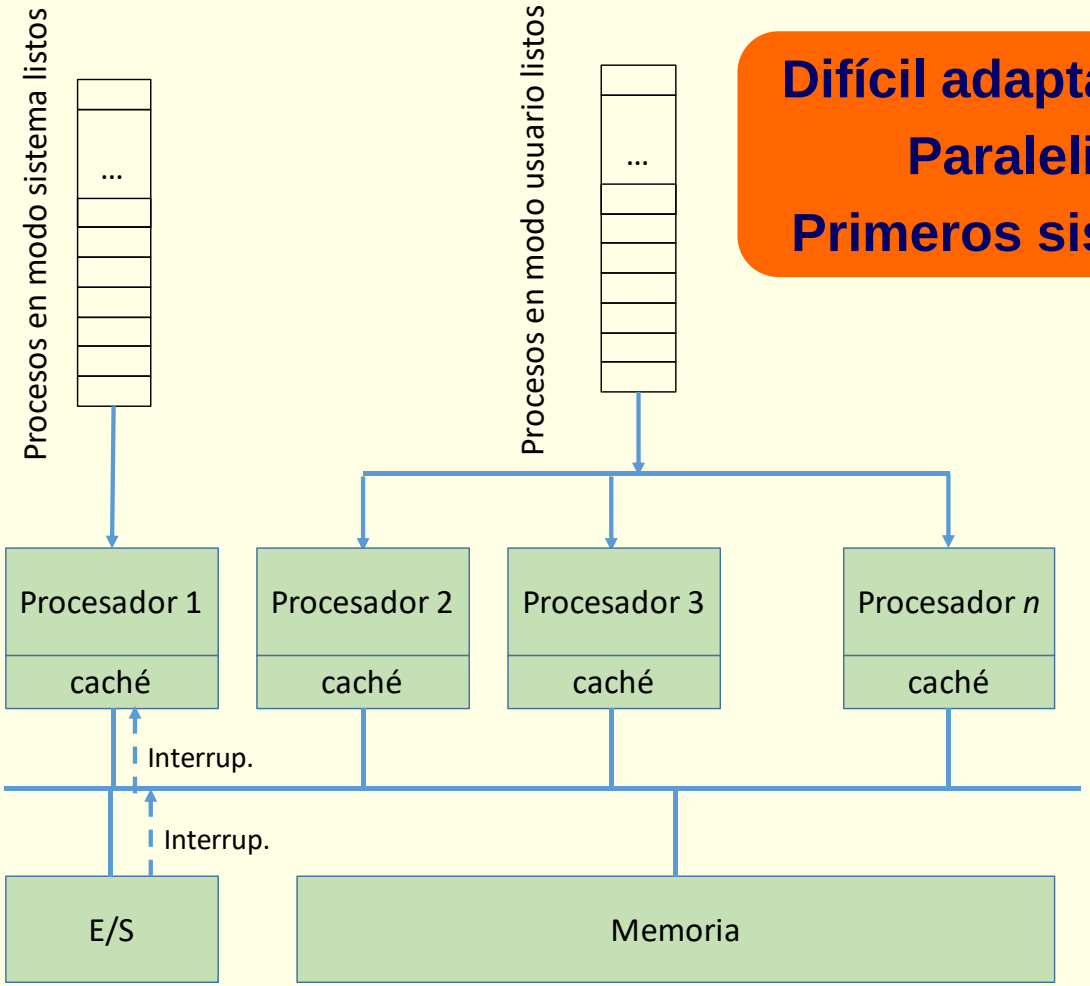


Compartición de recursos en multiprocesador

- Procesadores lógicos comparten caché de primer nivel.
- Núcleos de mismo socket comparten caché de último nivel.
- Núcleos de mismo socket comparten alimentación.
- Nodos NUMA acceso memoria remota mucho menos eficiente.
- Dentro del nivel de NUMA
  - Nodos separados por 1 elemento de comunicación (1 salto).
  - Nodos separados por 2 elementos de comunicación (2 saltos).
  - ……..

- Gran pregunta:
  - SO debe conocer esta jerarquía o puede tratar = a todas UCPs

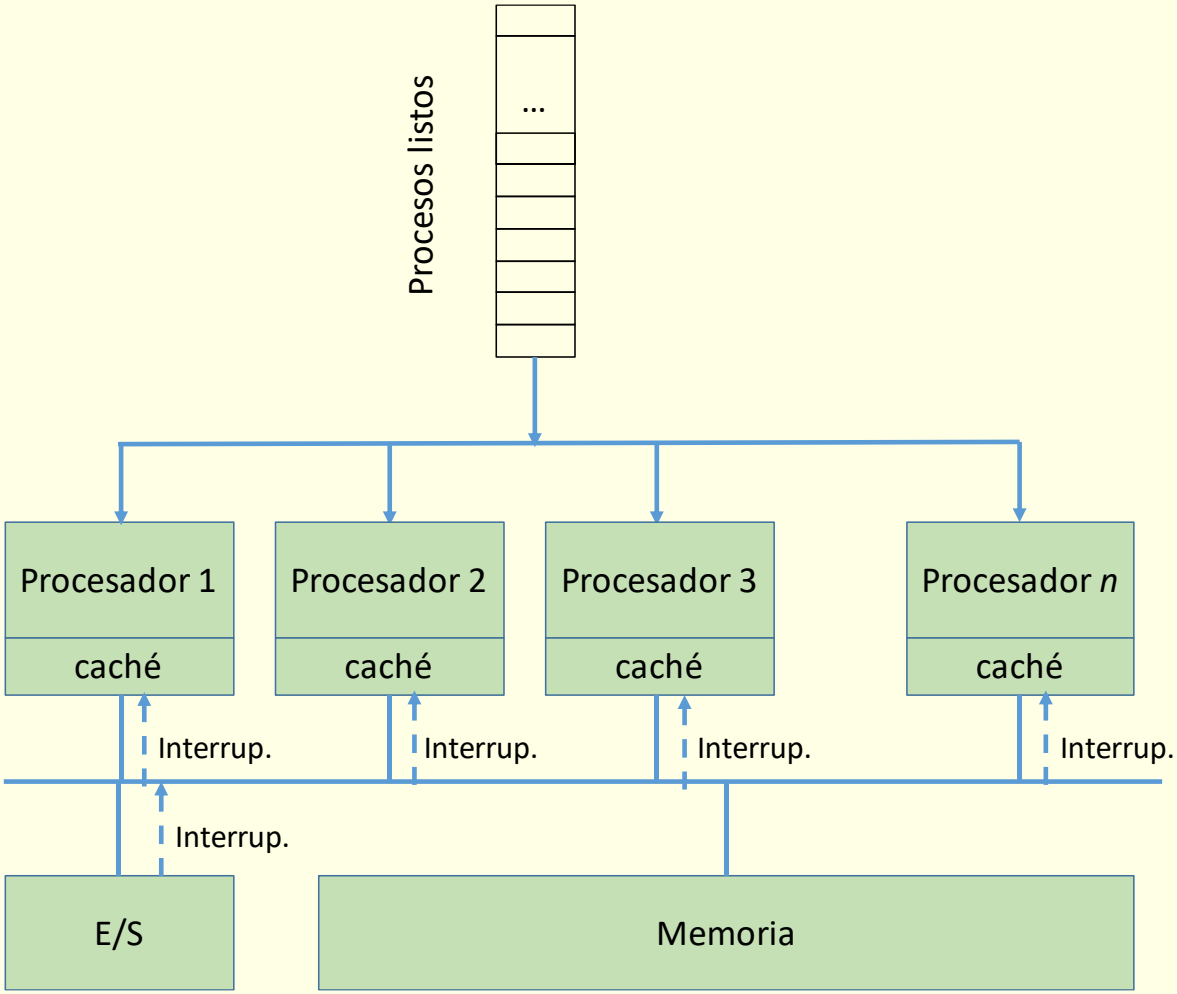
Multiprocesamiento asimétrico (ASMP)



Difícil adaptar SSOO a MP
Paralelismo real
Primeros sistemas ASMP

Tratamiento de eventos del SO en mismo procesador

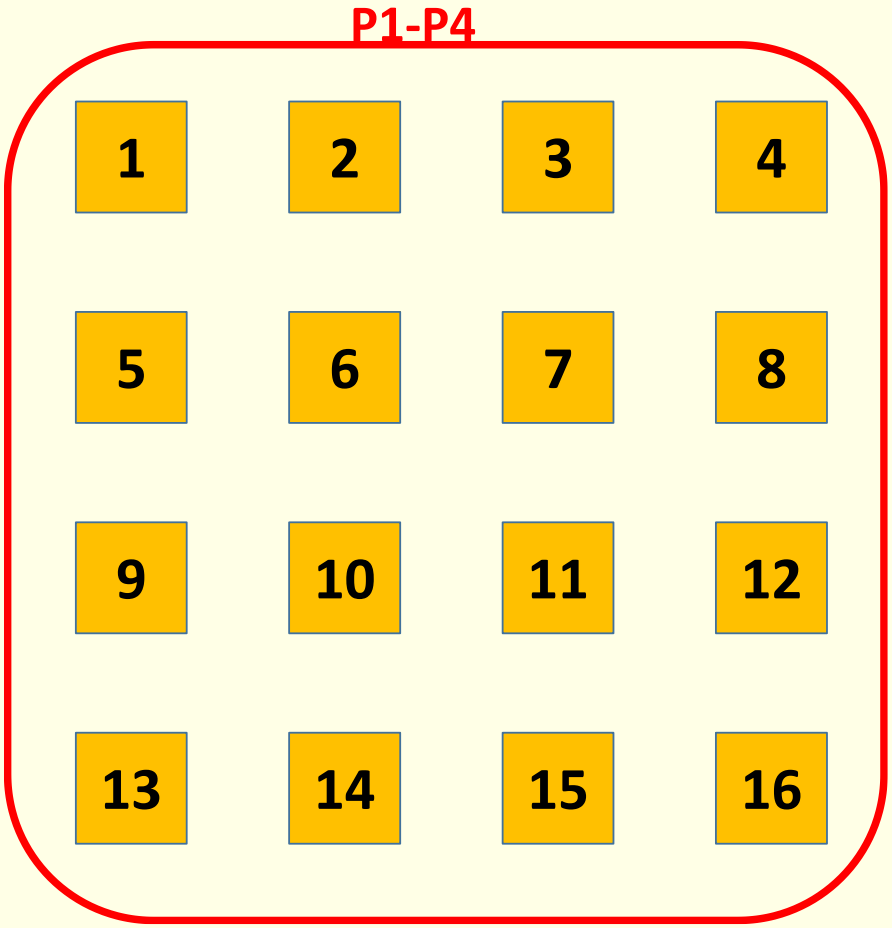
Multiprocesamiento simétrico (SMP)



Tratamiento de eventos del SO en cualquier procesador

Modelos de planificación de un MP: Global

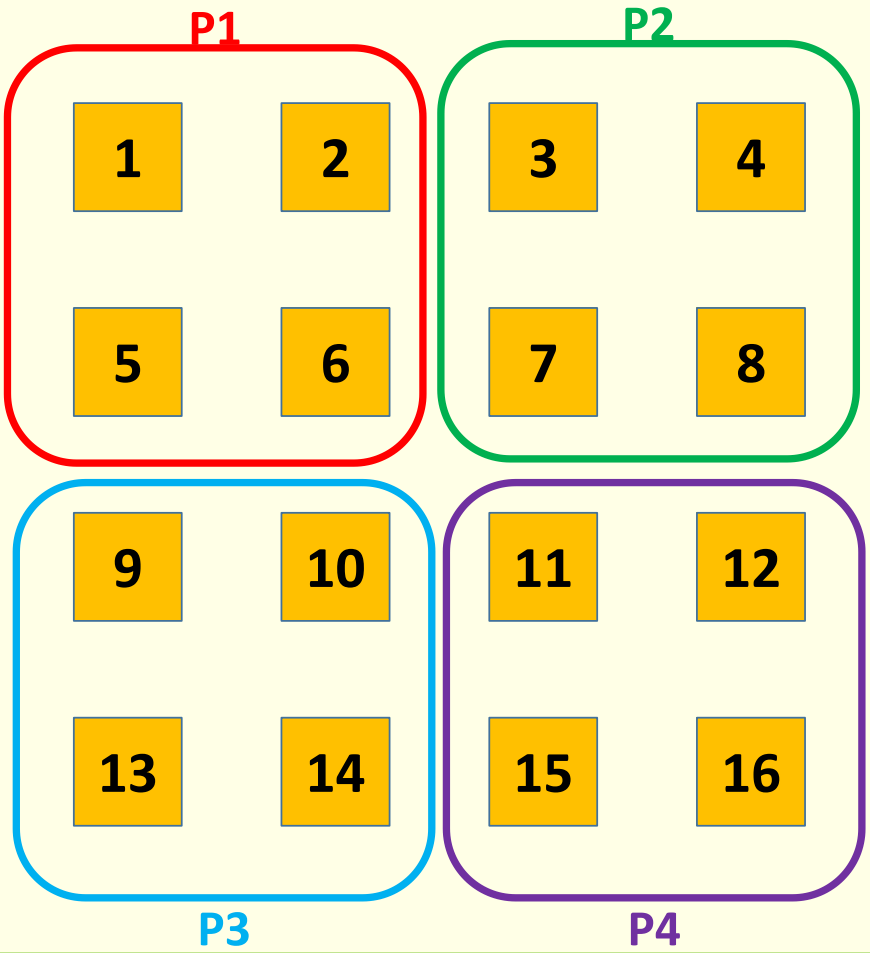
Asignación de procesos a procesadores



Parece solución más natural

Cualquier proceso en cualquier procesador

Modelos de planificación de un MP: Particionada



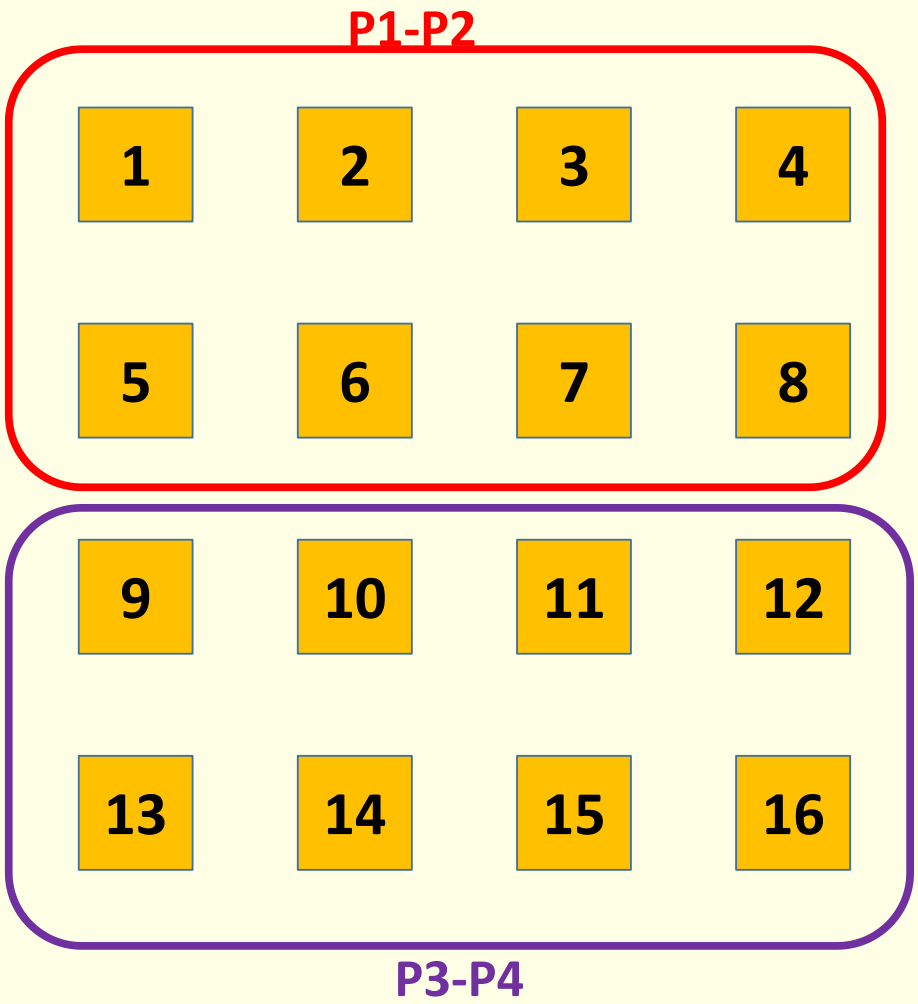
Proceso asignado a un procesador

Solución más habitual

Puede incluir migración para paliar desequilibrios:

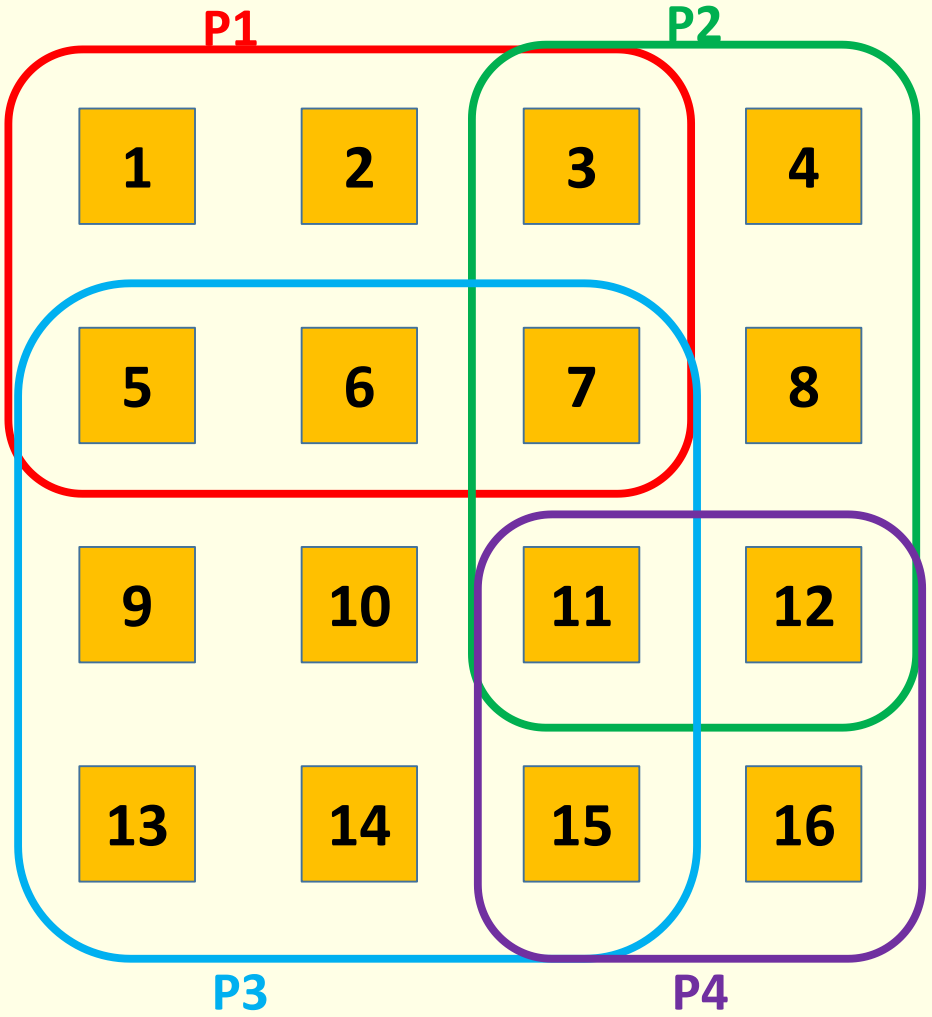
- ***pull*: nodo descargado toma procesos de cargado**
- ***push*: nodo cargado cede procesos a descargado**

Modelos de planificación de un MP: Conjuntos



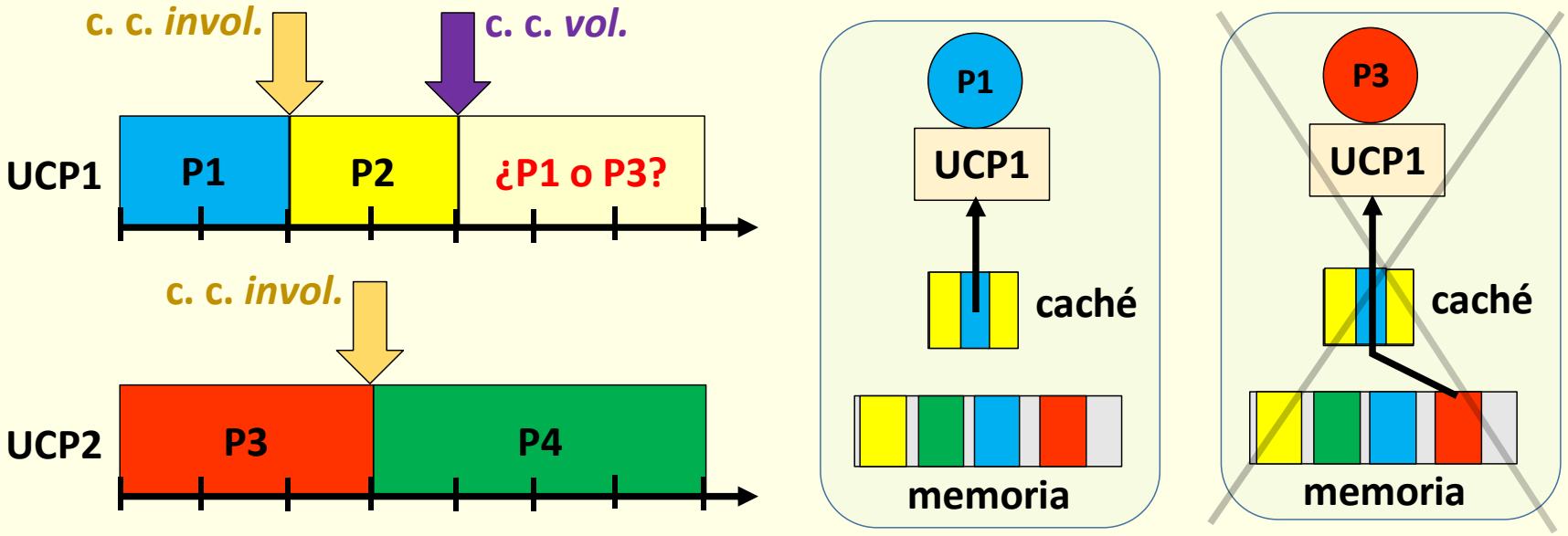
Proceso asignado a un conjunto de procesadores

Modelos de planificación de un MP: Arbitraria



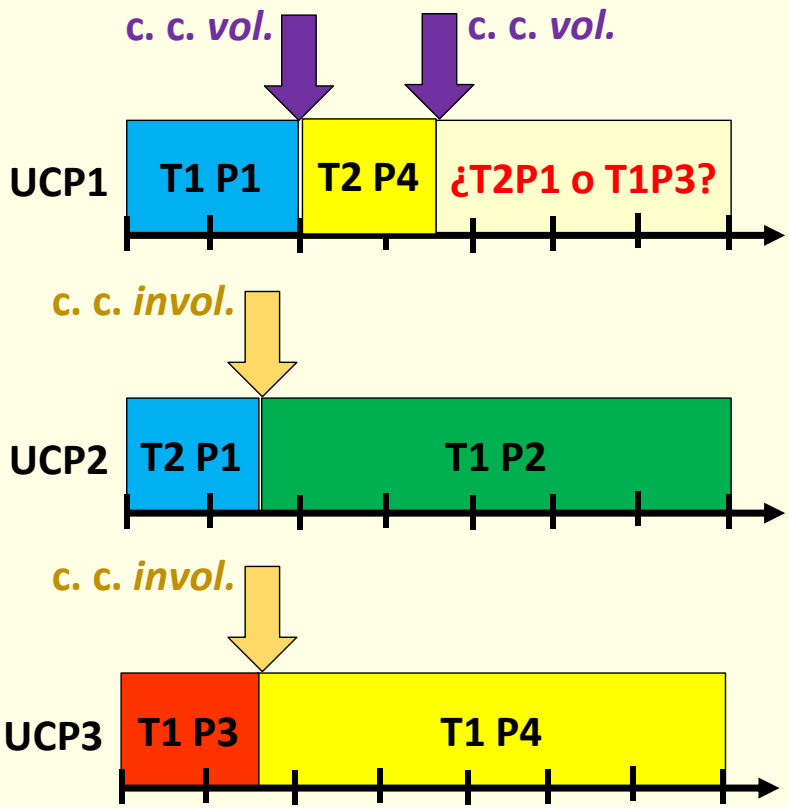
Afinidad natural

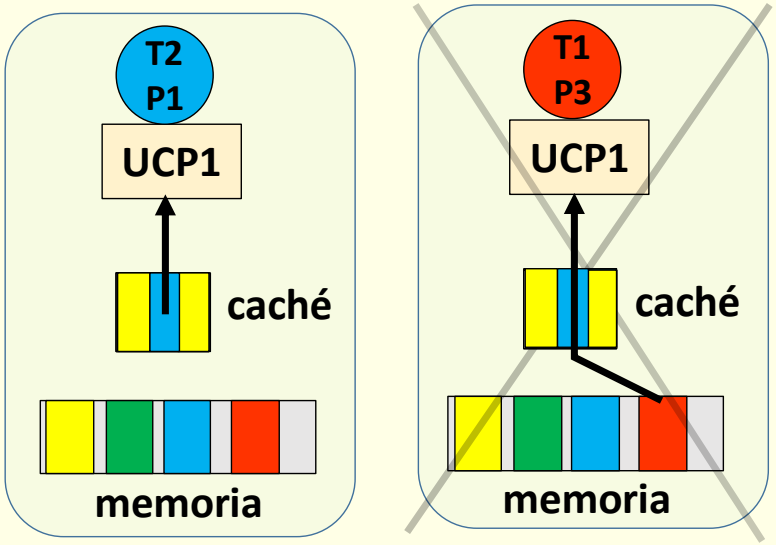
- Ejecutar ráfagas proceso misma UCP para aprovechar caché



Afinidad natural indirecta

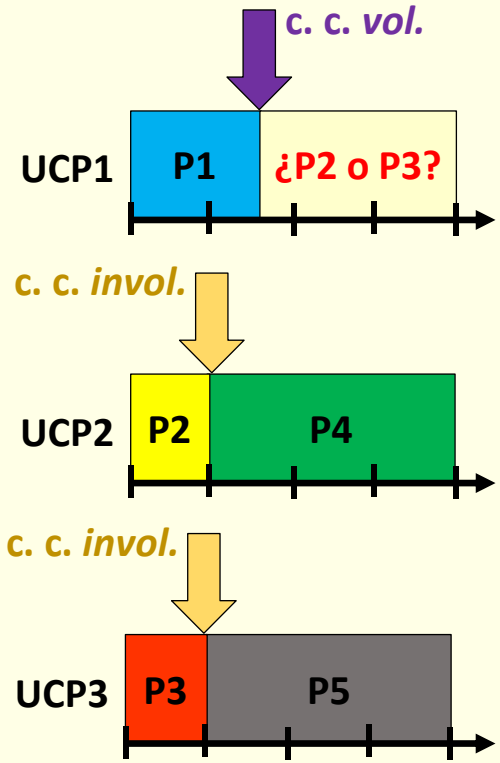
- Entre procesos que comparten mucha información:
  - p.e. *threads* del mismo proceso

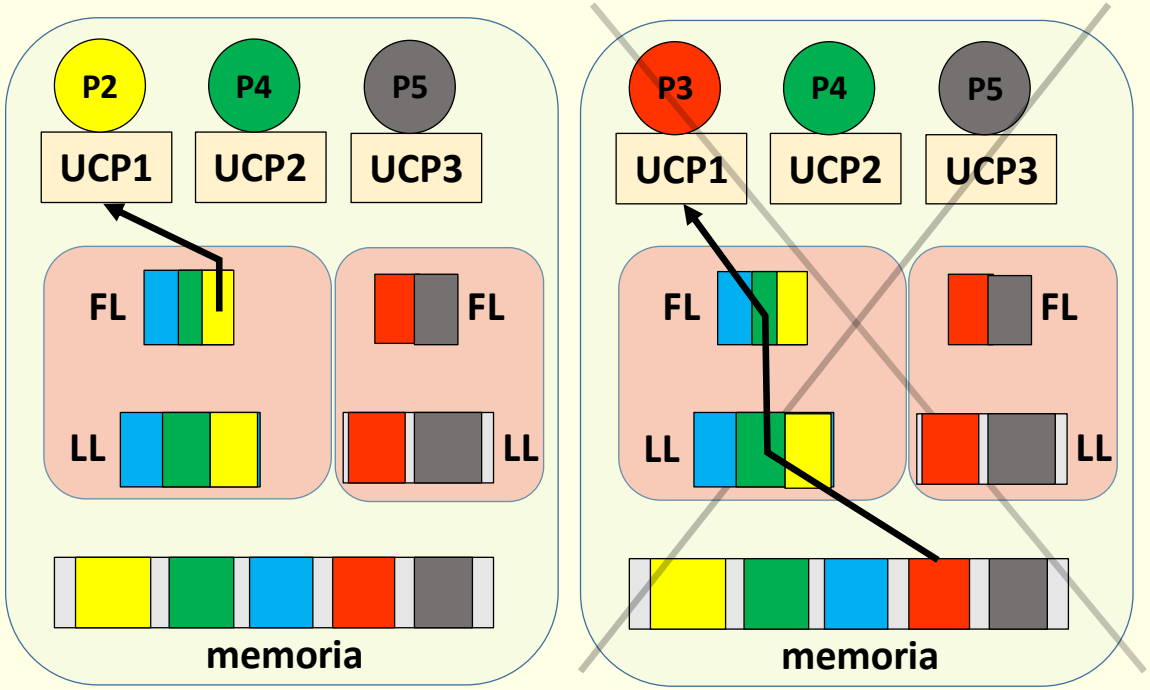




Afinidad natural jerárquica

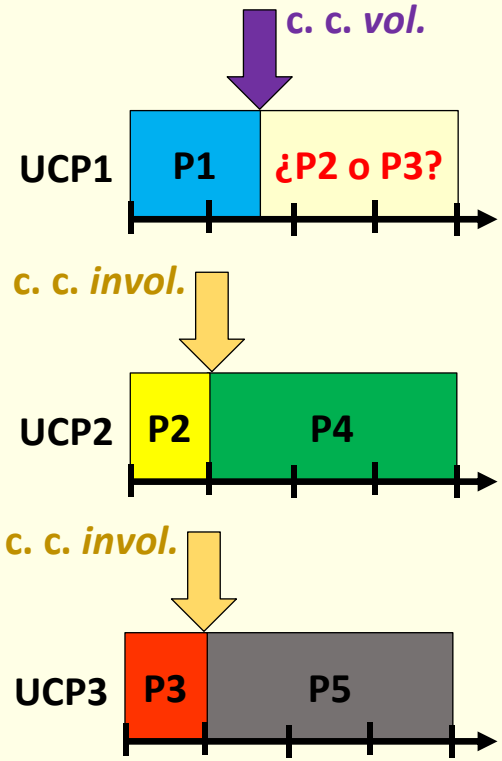
- UCPs mismo núcleo comparten toda la jerarquía de cachés

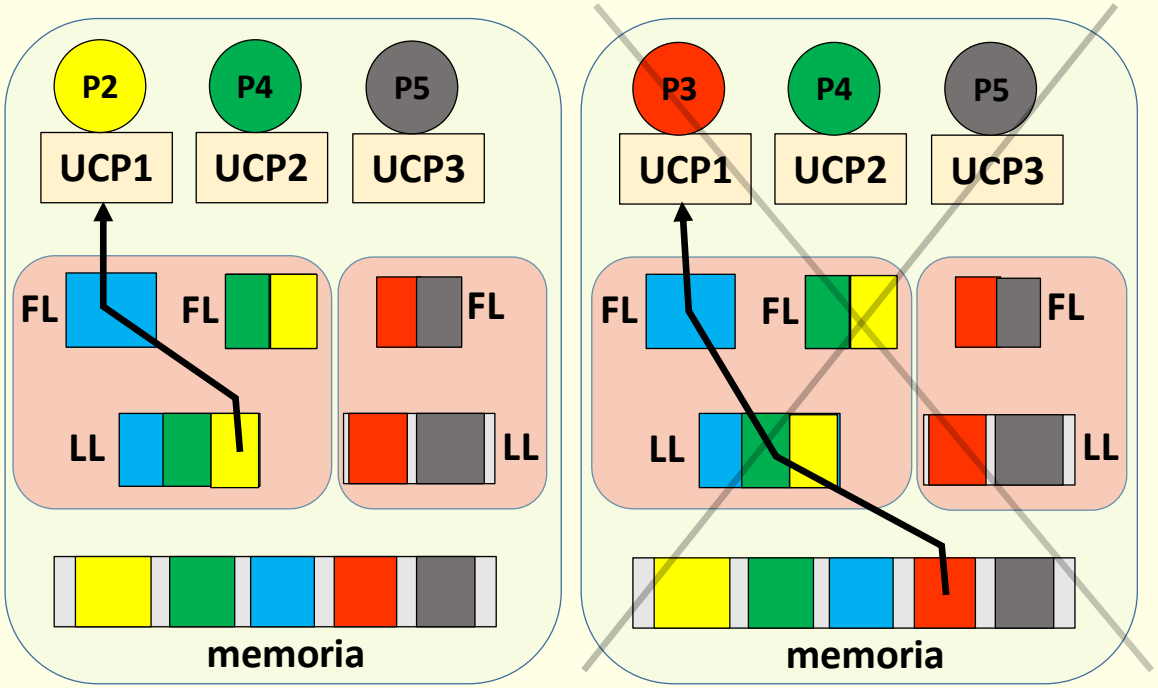




Afinidad natural jerárquica

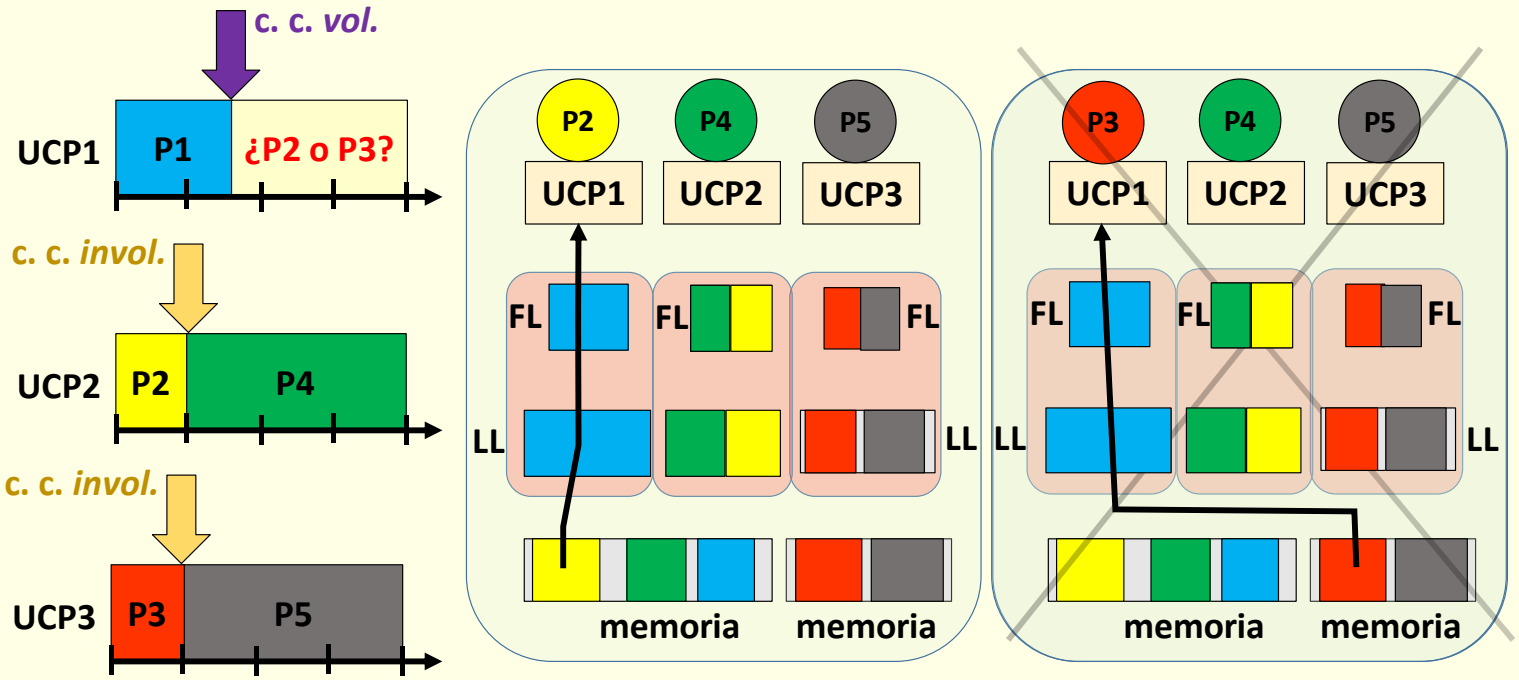
- UCPs mismo paquete comparten caché último nivel





Afinidad natural jerárquica

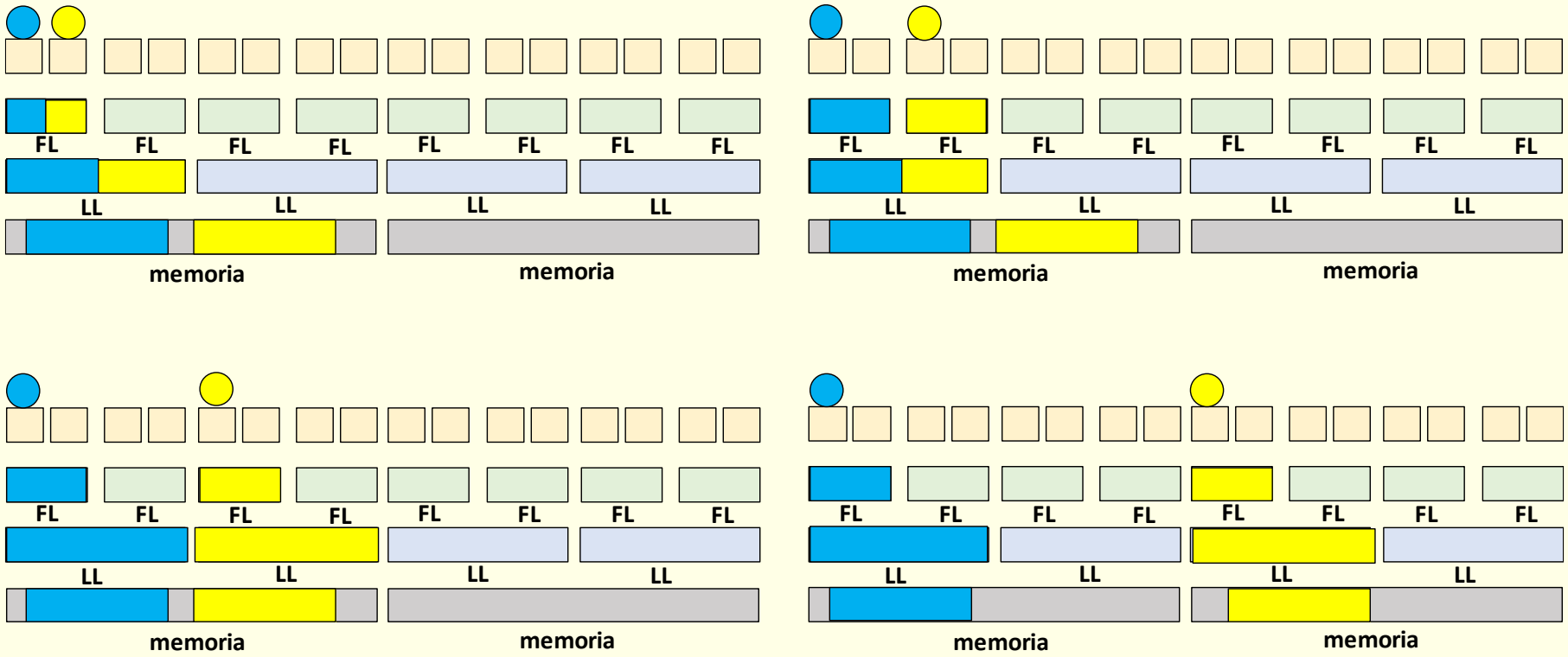
- UCPs mismo nodo acceso a memoria local
  - UCPs de distinto nodo tienen *anti-afinidad*



Competencia y afinidad natural

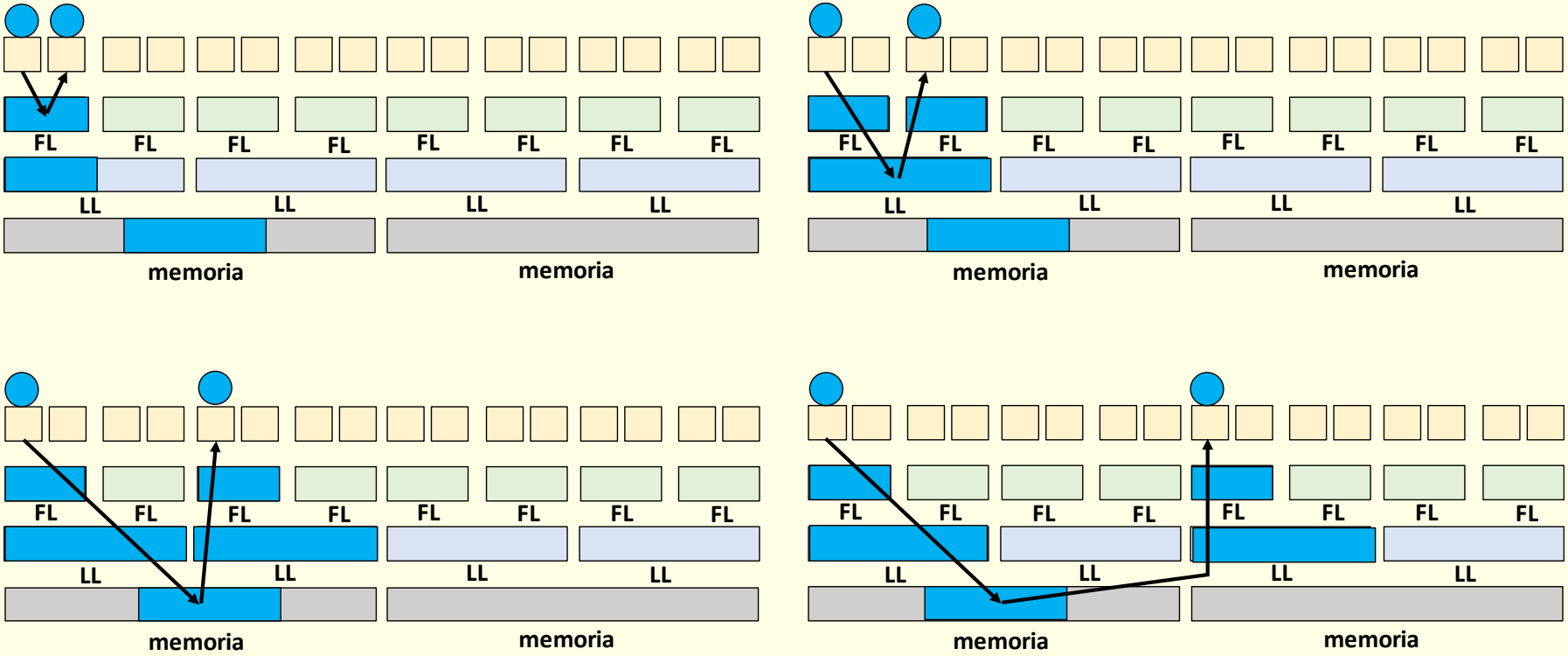
- Aplicando afinidad natural:
  - Durante ejecución de proceso se mantiene en UCPs afines
- ¿Ubicación inicial de un nuevo proceso?
- Consecuencias de compartir caché con otro proceso
  - Si independientes: compiten por espacio en caché
    - Tendencia a la diseminación
  - Si comparten memoria: comunicación rápida a través de caché
    - Tendencia a la concentración
- Posibles políticas NWC:
  - No usar procesador lógico libre para nuevo proceso
    - Si otro p. lógico de *core* ocupado por proceso independiente
  - No usar UCP libre de otro nodo NUMA para nuevo proceso
    - Si comparte memoria con un proceso ejecutando en este nodo
  - En ambos casos, si era la única UCP libre, no se usa (NWC)
- Planificación basada en el consumo

Competencia con procesos independientes



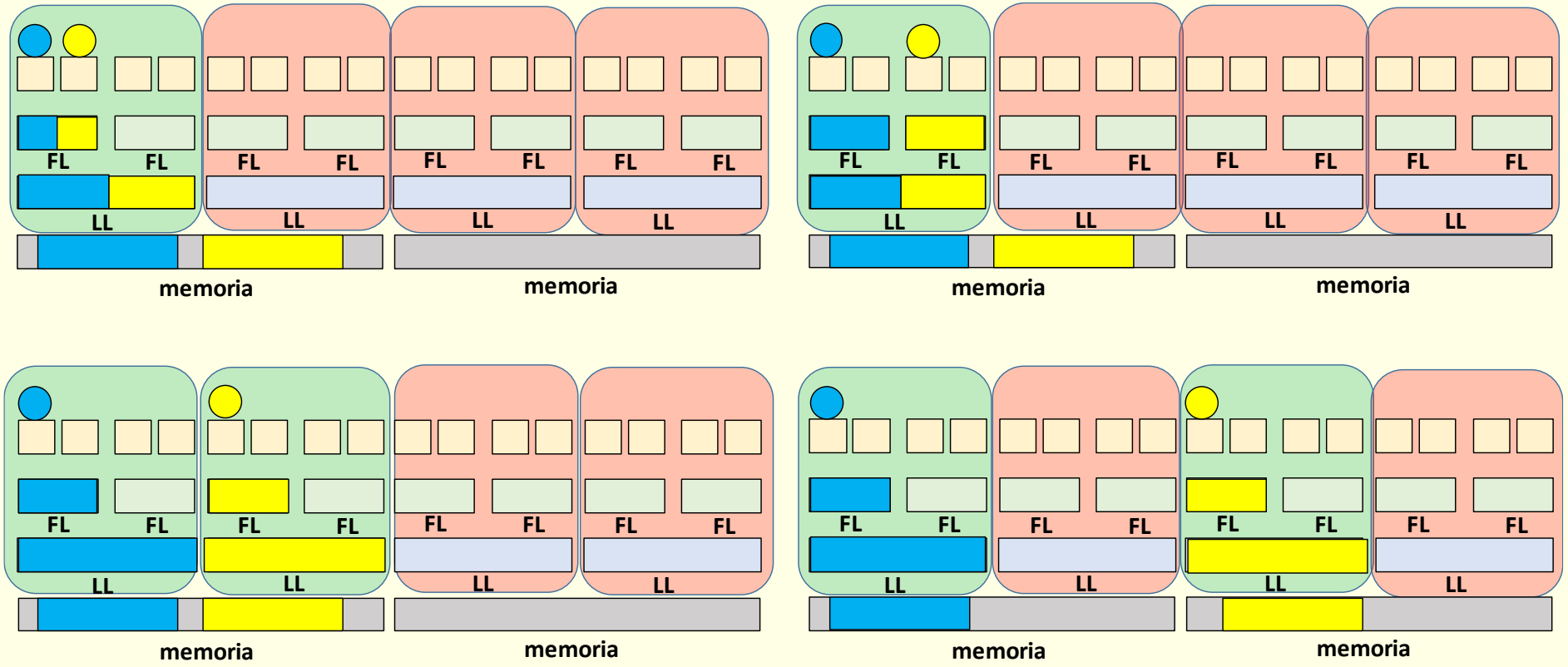
“Fuerza centrífuga”: diseminación

Competencia con procesos con compartición



“Fuerza centrípeta”: concentración

Competencia con procesos independientes: consumo

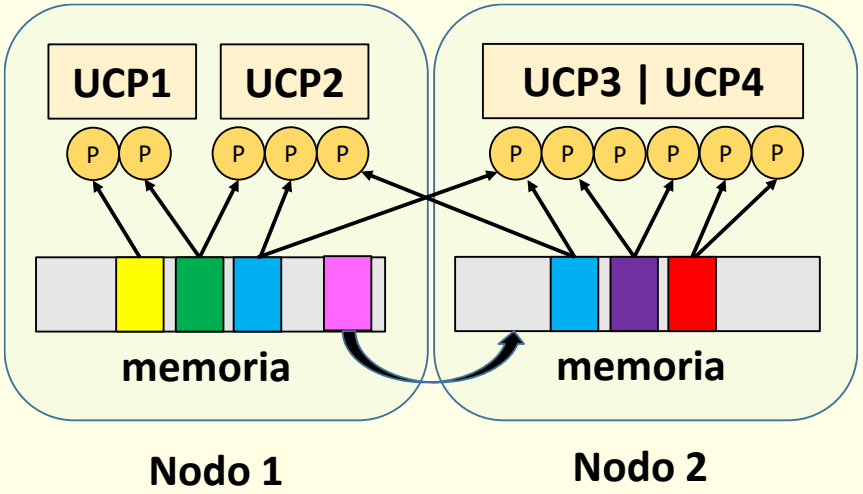


Diseminación dentro del mismo socket

No usar nuevo socket hasta llenar previo

Afinidad estricta

- SO permite a proceso definir en qué UCPs quiere ejecutar
  - POSIX: sched_setaffinity (usado por mandato taskset de Linux)
  - Se especifica como una máscara de bits: máscara de afinidad
- Permite establecer cualquiera de los modelos de planificación
- Mecanismo complementario en Linux: conjuntos de procesadores
- Ejemplos de uso:
  - Dedicar una UCP a un solo proceso para mejorar t. de respuesta
  - Diseminar *threads* de una aplicación en un sistema NUMA



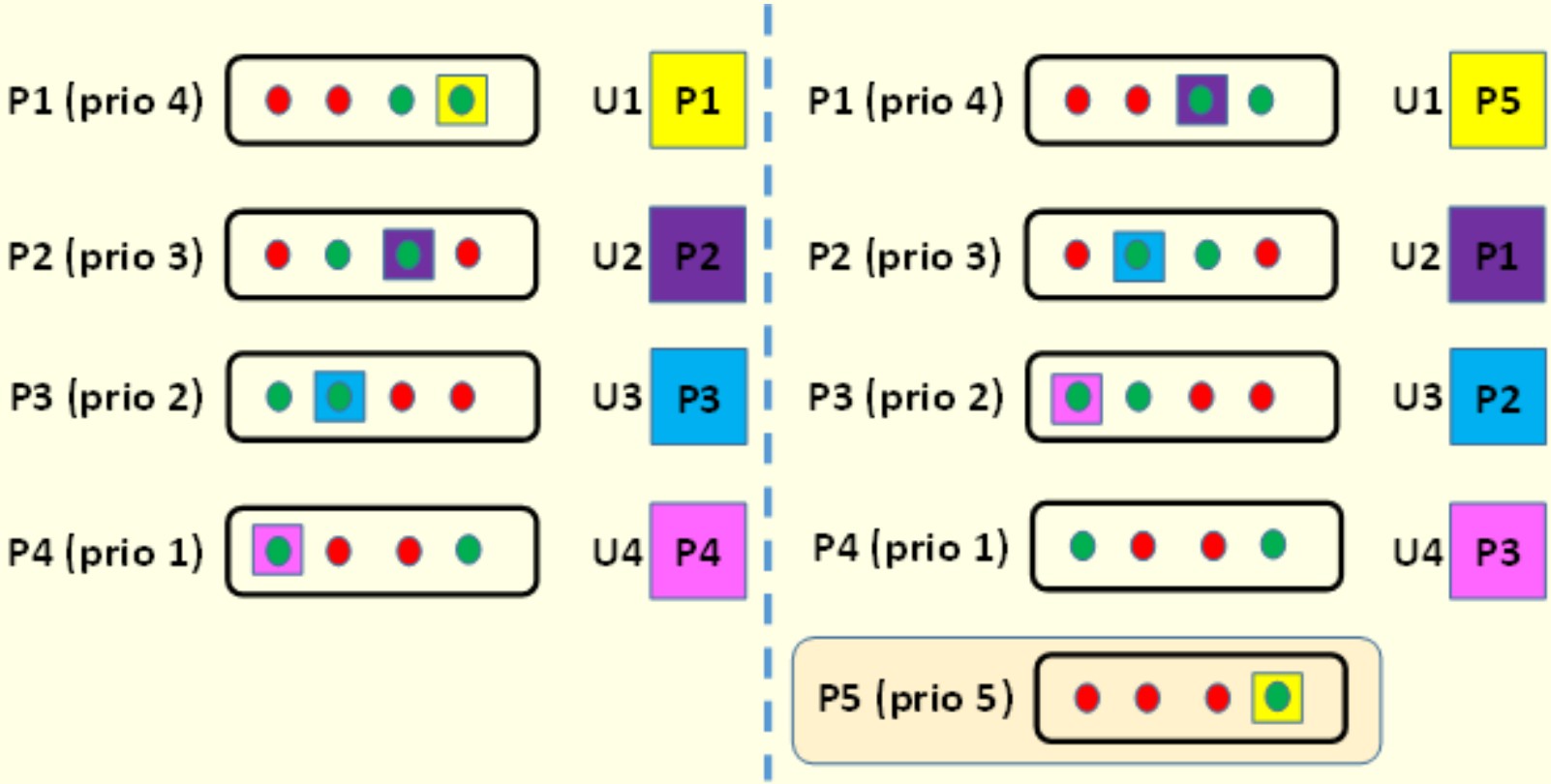
Servicios control uso memoria en NUMA

mbind, move_pages…

Planificación global

- Cola proc. listos común para todas UCPs: Solución más "natural"
- 2 escenarios planificación: UCP queda libre; proceso pasa a listo
- UCP *U* queda libre
  - Planificador elige proceso listo *Q más importante* (*máx(Prio(Q))*)
    - Solo 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*
     - Solo entre las que *P* tiene en su máscara de afinidad estricta
  2. Si no UCP libre: Busca *U* entre UCPs ⊂ en afinidad estricta *P*
     - donde ejecute proceso *Q* con menos prioridad en el sistema
     - Si $Prio(P)>Prio(Q) \rightarrow Q$ debe ser expulsado (si no *P* sigue listo)
  - Si UCP elegida $U \neq$UCP donde se produce desbloqueo/creación
    - IPI para forzar CCI en *U*
- Afinidad estricta puede causar reasignación UCPs

Afinidad estricta y reasignación de UCPs



Desbloqueo de P5 causa efecto dominó

Planificación global y afinidad natural

- Solución planteada es adecuada para sistemas t. real (no crítico)
  - *N* UCPs ejecutan *N* procesos más importantes
  - Pero no para SO propósito general: no respeta afinidad natural
- ¿Cómo integrar afinidad natural y prioridades sin desvirtuarlas?
  - Uso de *bonus* si proceso ejecutó última ráfaga en esa UCP
- UCP *U* libre: proceso *P* más importante teniendo en cuenta *bonus*
  - Si (P->ultima_UCP== U) → P->prio + bonus; U⊂ afinidad estricta *P*
- Proceso *P* pasa a listo por desbloqueo o nuevo: se le asigna

1. UCP afín libre; si no cualquier UCP libre
   - Solo entre las que *P* tiene en su máscara de afinidad estricta
2. Si no UCP libre: Busca *U* entre UCPs ⊂ en afinidad estricta *P*
   - donde ejecute proceso *Q* con menos prioridad en el sistema
   - teniendo en cuenta *bonus*
3. Si P nuevo no hay afinidad natural (no UCP afín y no *bonus*)
   - Excepto si comparte mem. con otros procesos (afinidad indirecta)

Ejemplos planificación global y afinidad natural

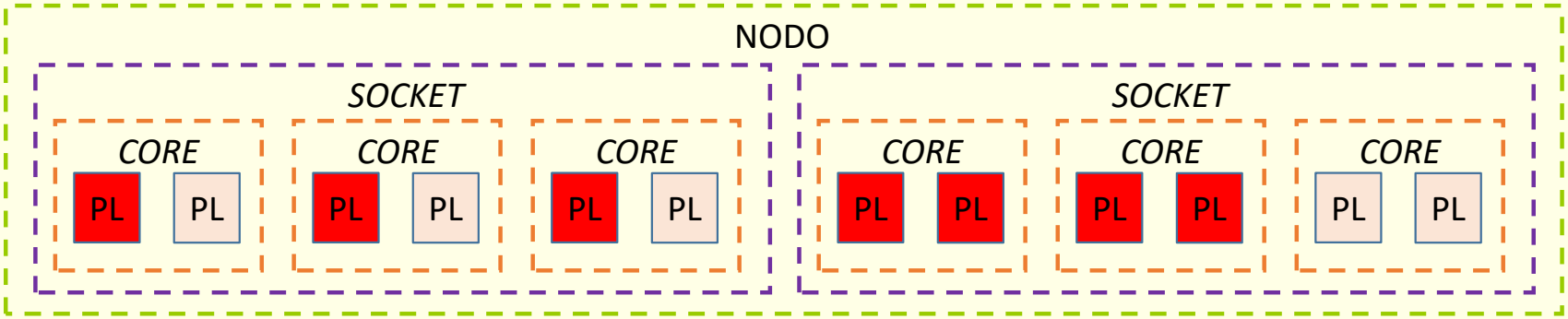
- bonificación de 2
- Ejemplo 1: UCP1 queda libre
  - procesos listos:
    - P1 prioridad 12, última ráfaga UCP1
    - P2 prioridad 13, última ráfaga UCP2
    - P3 prioridad 11, última ráfaga UCP3
  - Selección P1 ejecuta en UCP1
- Ejemplo 2: desbloqueo P4 prioridad 17, última ráfaga UCP2
  - procesos en ejecución:
    - P1 prioridad 13 en UCP1
    - P2 prioridad 14 en UCP2
    - P3 prioridad 20 en UCP3
  - Selección P2 expulsado por P4 de UCP2

Planificación global y afinidad natural jerárquica

- Extensión del *bonus* a la jerarquía
  - *Bonus* procesador lógico > *bonus core* > *bonus socket*
- UCP *U* libre: proceso *P* más importante teniendo en cuenta *bonus*
  - No asigna (NWC) si todos listos han ejecutado otros nodos NUMA
- Proceso *P* pasa a listo por desbloqueo o nuevo: se le asigna

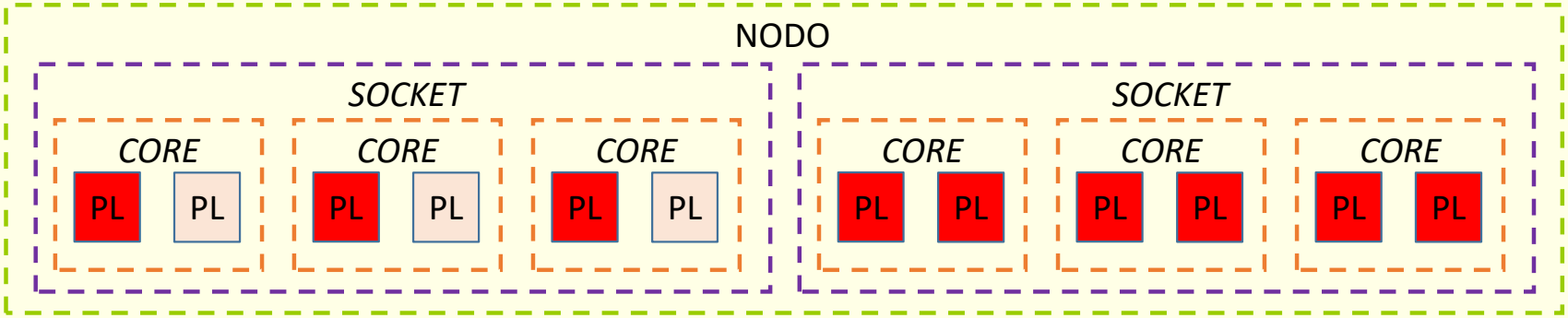
1. UCP afín libre; si no en *core* afín, en *socket* afín (respeto máscara)
   - No asigna (NWC) si todas UCPs libres distintos nodos NUMA que *P*
2. Si no UCP libre: Busca *U* entre UCPs ⊂ en afinidad estricta *P*
   - donde ejecute proceso con menos prioridad teniendo en cuenta *bonus*
3. Si nuevo pero comparte mem. → afinidad indirecta jerárquica
   - **Concentración** de procesos/*threads* relacionados
4. Si nuevo y no comparte memoria → **diseminación**
   - Buscar nodo *N* con más UCPs libres; en *N* socket *S* con más UCPs libres; en *S* core *C* con más UCP libres; en *C* UCP libre ⊂ máscara
   - Si prima reducción de consumo: diseminación dentro de mismo socket

Estimación de capacidad de cómputo disponible



¿qué *socket* tiene más capacidad de cómputo disponible? <u>El 2º</u>

Linux estima que cada PL adicional añade solo 10% de potencia



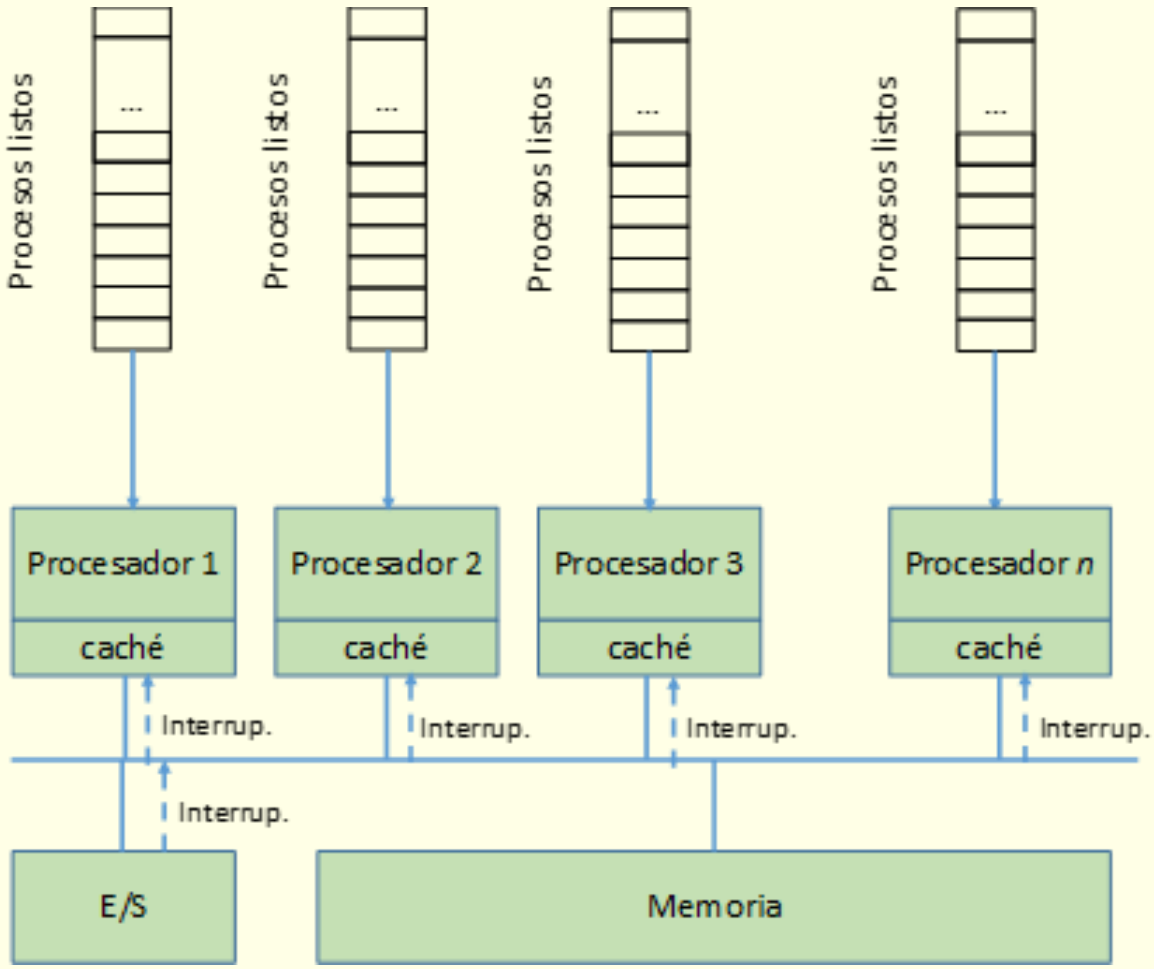
NWC: No asignar proceso listo a ninguna UCP lógica libre

Para no perturbar ejecución de proceso en otra UCP lógica del *core*

Planificación particionada con migración

- Desventajas planificación global con cola común:
  - Accesos a cola requieren cerrojo: mala escalabilidad
  - Soporte de la afinidad natural "tangencial"
  - Algoritmo mezcla UP y MP (p.e. *bonus*)
- Planificación particionada: 1 cola por UCP
  - No hay congestión por cerrojo; Se aprovecha mejor afinidad
  - Algoritmo de planificación UP para cada cola
  - Decisiones MP externas al algoritmo de planificación
- Uso de migración para equilibrar carga
- Esquema más habitual; sobre todo para MP grandes
- Diversos esquemas dependiendo de asignación UCP a proceso
- Vamos a describir un esquema con dos puntos de decisión:
  - Asignación de UCP a proceso cuando se crea
  - Reasignación de UCP de proceso en una migración

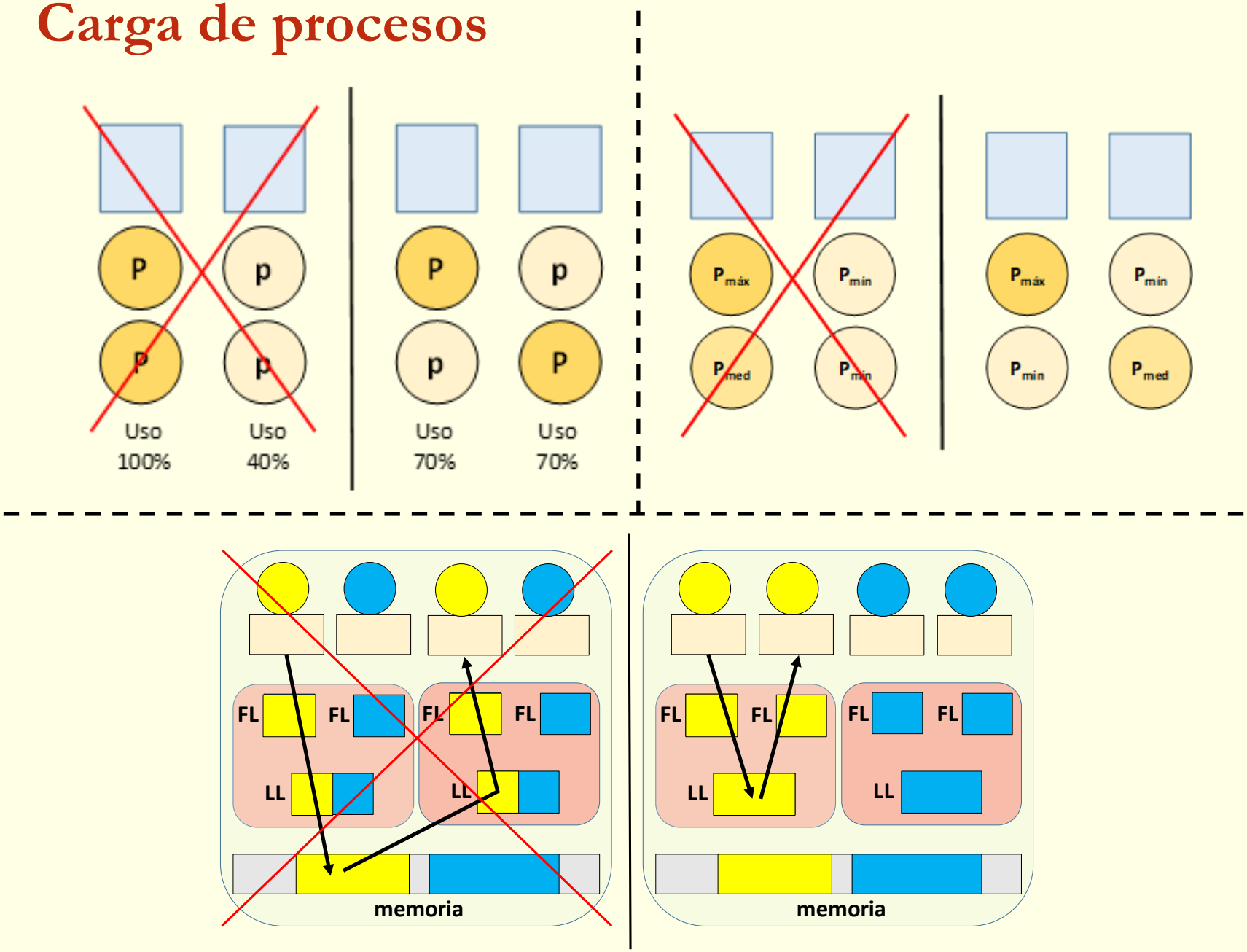
Planificación particionada con 1 cola/UCP



Asignación de UCP a nuevo proceso

- ¿Qué UCP se asigna a un nuevo proceso?
- Si proceso nuevo independiente: diseminación
  - UCP asignada corresponde a
    - Nodo menos cargado (*N*)
    - *Socket* (S) menos cargado de *N*
    - *Core* (C) menos cargado de *S*
    - Procesador lógico (L) menos cargado de F
  - Si minimizar consumo: diseminación dentro de *socket* ya usado
- Si proceso comparte memoria: concentración
  - Asignar misma UCP o UCP del mismo *core*
- Cálculo de la carga de un componente: nº procesos/nº UCPs
  - Denominador potencia de cómputo: interacción entre UCPs
    - Potencia dos UCPs mismo *core* < de distinto *core*
  - Numerador carga: depende de características de procesos

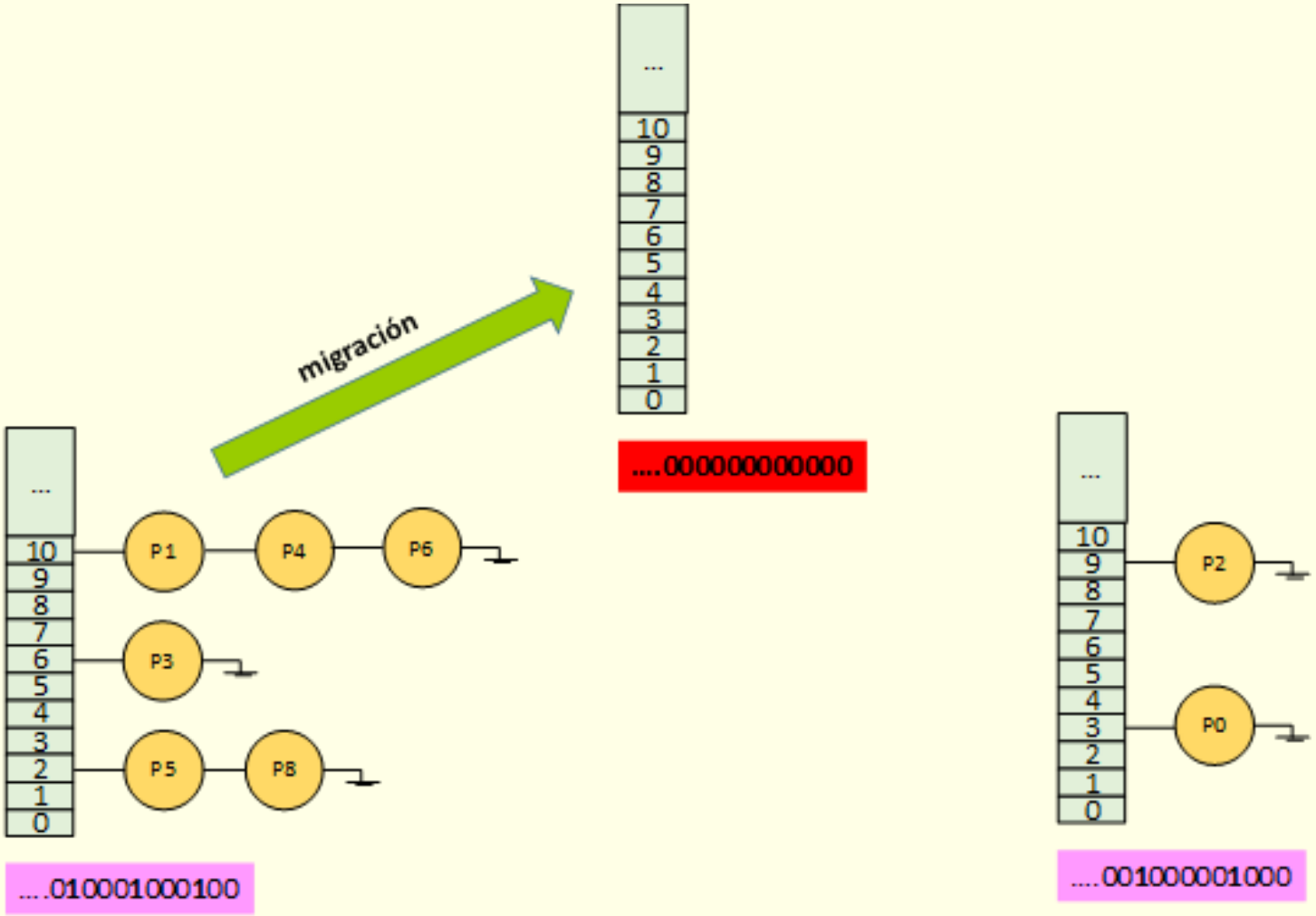
Carga de procesos



Migración de procesos

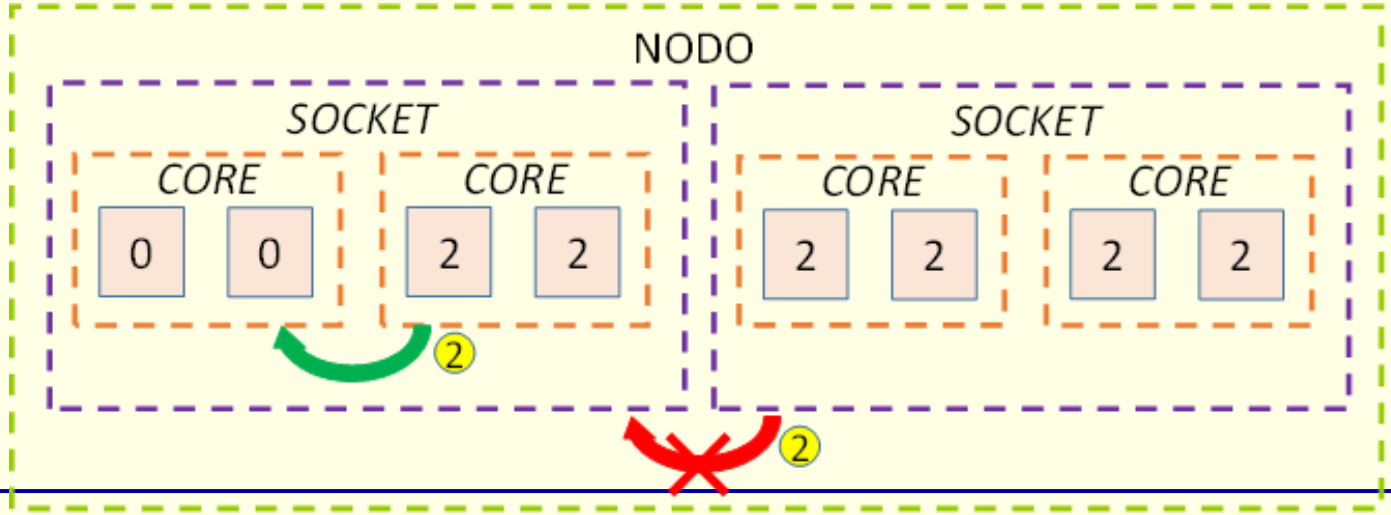
- Equilibrado de carga mediante migración de procesos:
  - Comprobar periódicamente si desequilibrios
  - Si cola de una UCP queda vacía
- Dos estrategias de equilibrado:
  - *Pull*: UCP mueve a su cola procesos de otra UCP
  - *Push*: UCP mueve procesos de su cola a la de otra UCP
  - Si cola vacía → *pull*; habitual usar *pull* en ambos casos
- Pero requiere 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
- Migración debe tener en cuenta afinidad jerárquica
  - Mayor penalización cuanto menos afines UCPs involucradas

Equilibrado mediante migración de procesos

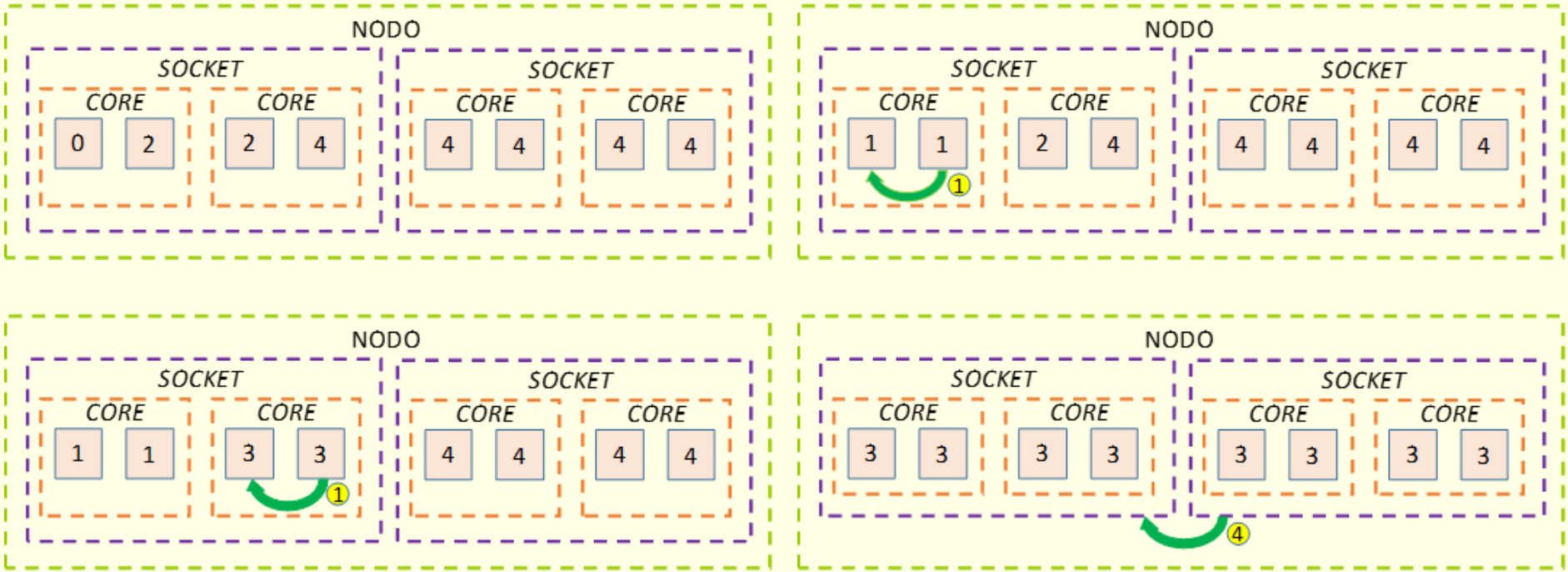


Migración y afinidad natural jerárquica

- Equilibrado siguiendo jerarquía
  - Equilibrar procesadores lógicos de cada *core*
  - Equilibrar *cores* de cada *socket*
  - Equilibrar *sockets* de cada nodo
  - Equilibrar nodos
- Migración teniendo en cuenta jerarquía de afinidades
  - mejor migrar entre UCPs que pertenezcan al mismo núcleo
  - mejor migrar entre UCPs que pertenezcan al mismo *socket*
  - mejor migrar entre UCPs que pertenezcan al mismo nodo



Equilibrado mediante migración jerárquica



Sistemas operativos avanzados

Planificación del procesador

3ª parte: planificación Linux

Planificación en Linux

- Durante mucho tiempo no ha sido la pieza más brillante de Linux
  - O(N) hasta versión 2.6, con planificación global para MP
  - Demasiados parámetros configurables
  - No exento de polémica (Kolivas)
  - Errores graves en la gestión MP:
    - "*The Linux Scheduler: a Decade of Wasted Cores*"
  - Quizás Linus Torvalds ha minusvalorado este componente
- Estudiaremos dos esquemas de planificación de Linux:
  - Planificador O(1): introducido en la versión 2.6
  - *Completely Fair Scheduler* (CFS) introducido a partir de 2.6.23
- Ambos usan para MP planificación particionada con migración
  - Decisiones de planificación MP externas al algoritmo UP
- Linux da soporte extensiones t. real POSIX
  - Procesos t. real siempre más prioritarios que normales

Clases de procesos en Linux

- Se definen mediante llamada *sched_setscheduler*
- Clases actualmente definidas:
  - Clases de tiempo real
    - Prioridad estática expulsiva (1..99) con FCFS o RR para iguales
    - SCHED_FIFO: continúa en ejecución hasta que:
      - Termina (CCV), se bloquea (CCV) o se desbloquea uno + prioritario
    - SCHED_RR: Igual que FIFO pero además por fin de rodaja
    - Específicos de Linux: SCHED_DEADLINE
      - Implementación de EDF en Linux
  - Clase normal (SCHED_NORMAL, *aka* SCHED_OTHER)
    - El planificador por defecto de Linux (actualmente CFS)
    - Prioridades entre -20 (máxima) y 19 (mínima)
  - SCHED_BATCH: similar a SCHED_NORMAL pero:
    - Asume que proceso es *CPU-Bound* y no expulsa a otros procesos
  - SCHED_IDLE: procesos con urgencia extremadamente baja

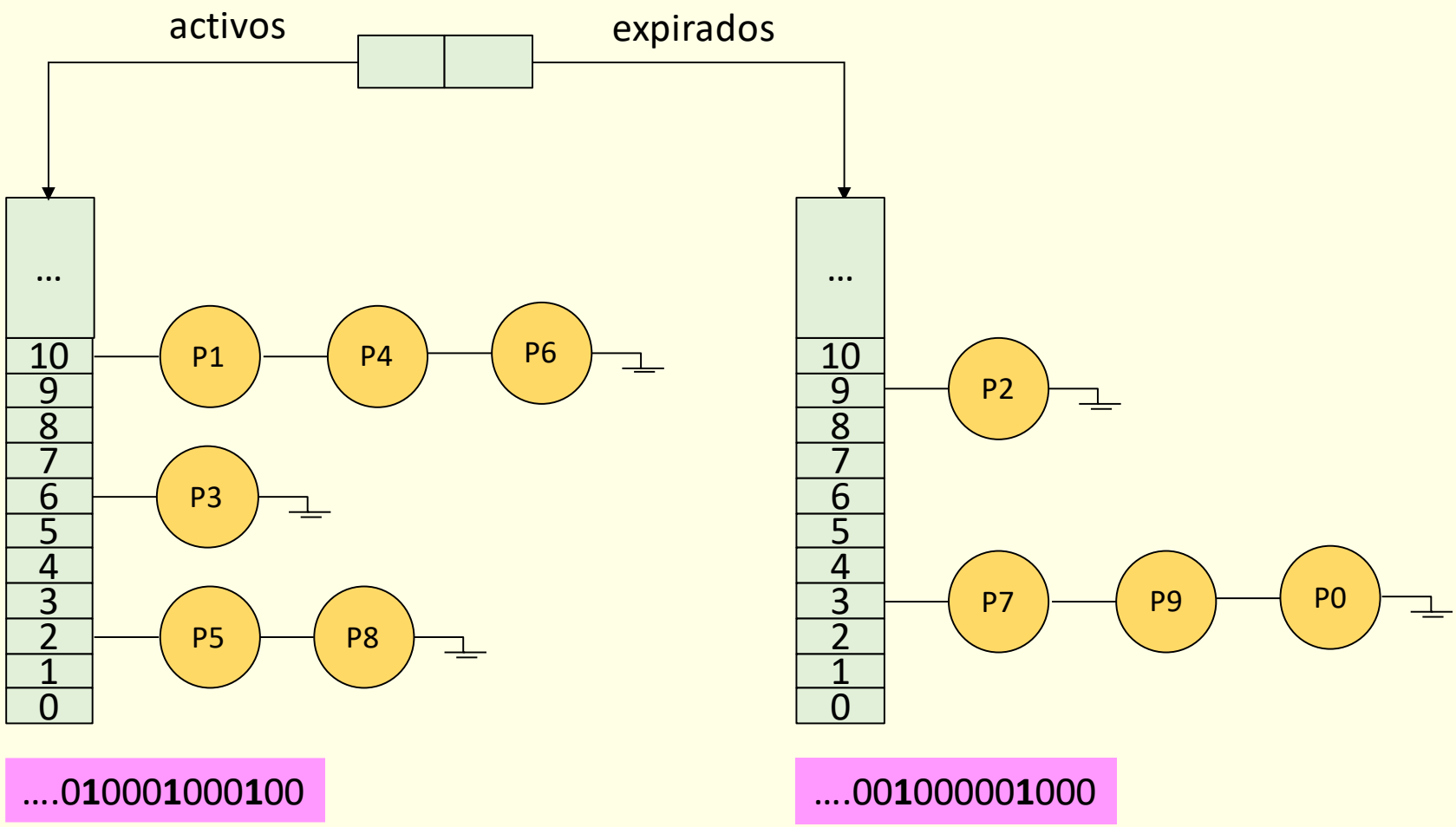
Planificador de Linux O(1)

- Esquema de prioridad basado en cola multinivel con realimentación
  - Uso de listas de procesos/prioridad + bitmap → O(1)
- Esquema expulsivo con prioridad dinámica
  - Prioridad base estática (*nice*): -20 (máx.) a 19 (mín.)
  - Ajuste dinámico (*bonus*) de -5 (mejora) a +5 (empeora)
    - Depende de uso de UCP y tiempo de bloqueo (+ UCP → - Prio)
    - Favorece procesos interactivos y con E/S y reduce inanición
- Rodaja con tamaño fijo que depende de la prioridad base estática
  - 5ms (prio 19); 100ms (prio 0); 800 ms (prio -20)
  - En `fork` padre e hijo se reparten la mitad de rodaja restante
- Proceso más prioritario con doble ventaja sobre menos prioritario
  - Lo expulsa y obtiene mayor porcentaje de procesador
  - ¿Proceso con prio. máx. no deja ejecutar nunca a uno con mín?

Planificador de Linux O(1)

- ¿Cómo concilia prioridad y equitatividad?:
  - 2 listas de procesos listos: activos y expirados
  - Planificador selecciona proceso + prioritario de lista de activos
  - Si proceso agota su rodaja pasa a lista de expirados
    - Excepto si "interactivo" (poca UCP) que vuelve a activos
      - A no ser que haya procesos en lista de expirados que lleven mucho tiempo en ella o más prioritarios que el que agotó la rodaja
  - Si se bloquea, cuando se desbloquee pasa a lista de activos
  - Se crean "rondas de ejecución" (épocas)
    - Procesos agotan rodaja y terminan todos en lista de expirados
    - En una ronda proceso va gastando su rodaja por tramos
      - Si no procesos con rodaja 800ms tendrían mal tiempo de respuesta
    - "Fin de ronda": intercambio de listas

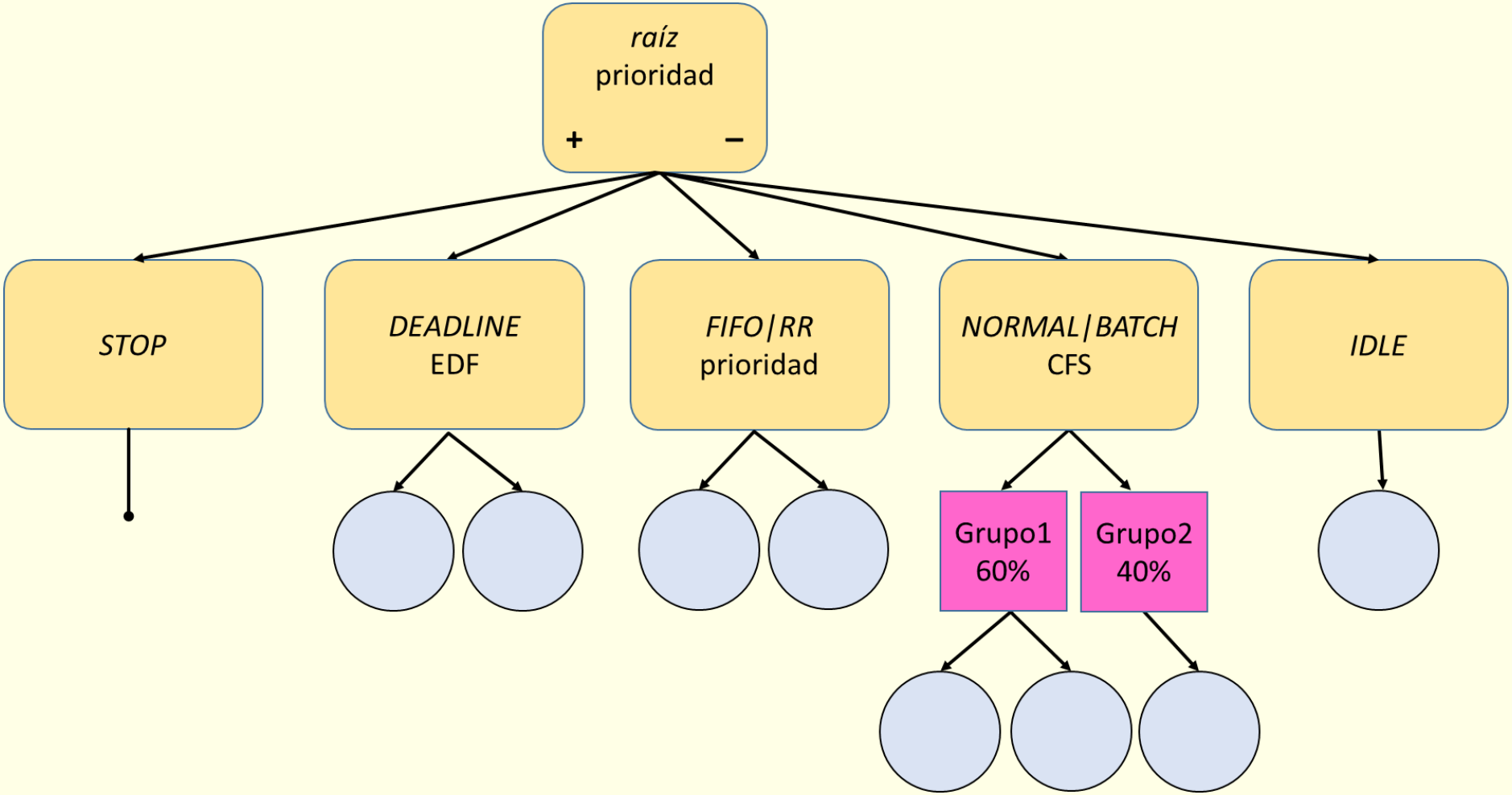
Planificador de Linux O(1)



Limitaciones de O(1)

- Lleno de parámetros difícilmente ajustables
  - *Bonus*, cuándo considerar "interactivo" un proceso…
- Gestión de clases de procesos con esquema de prioridad integrado
  - Usa 140 listas de procesos: 100 de tiempo real y 40 normales
- Mejor esquema jerárquico genérico
  - Cada clase su propio algoritmo de planificación
- No proporciona una solución *fair-share*
- Prioridades deberían considerarse de forma relativa
  - P.e. solo 2 procesos prio. mínima → RR 5 ms
    - Sin embargo, solo 2 de prio. máxima → RR 800 ms
    - Deberían tener un comportamiento similar
- Nuevo esquema de planificación CFS:
  - Marco genérico jerárquico de planificación con dos niveles
  - Algoritmo CFS propiamente dicho

Planificación jerárquica de clases de procesos



Completely Fair Scheduler (CFS)

- Algoritmo basado en tiempo virtual (GPS)
- Cálculo tiempo virtual consumido por proceso depende de su peso
- Objetivo: Todos los procesos usen "mismo tiempo virtual" de UCP
- Selección: proceso con < t. virtual consumido: tratado +injustamente
- Cada prio. (*nice*) tiene un peso: -20→88761, 0→1024, 19→15
  - Peso ≈ $1024/1{,}25^{nice}$
- Tiempo virtual (*vruntime*) normalizado respecto a proceso prio 0:

$$vruntime_{P_i} += \frac{T * Peso_{P_0}}{Peso_{P_i}}$$

  - + prioridad → + "lento" corre el reloj
    - Para proceso con prioridad 0: tiempo virtual = tiempo real
- Implementación precisa irrealizable en la práctica
  - Continuamente habría que reasignar el procesador
  - Pr. seleccionado ejecuta un tiempo: depende de prio. y la carga
  - Al completarse ese tiempo se vuelve a ejecutar algoritmo

CFS: asignación de tiempo ejecución

- Se establece tiempo (periodo) para ejecución de todos proc. listos
  - Normalmente periodo = sched_latency (6 ms. por defecto)
- Si muchos proc. listos (n), sobrecarga por frecuentes CCI
  - Se amplía periodo (sched_min_granularity 0,75ms. por defecto)

$$Si\ \left(\frac{sched_latency}{n}\right) < sched_min_granularity \rightarrow periodo = n * sched_min_granularity$$

- Tiempo asignado a Pi proporción entre su peso y total de los listos

$$Rodaja_{P_i} = \frac{periodo * Peso_{P_i}}{\sum_{j=1}^{n} Peso_{P_j}}$$

- Se asigna mismo tiempo virtual a todos los procesos:

$$vruntime_{P_i} += \frac{Rodaja_{P_i} * Peso_{P_0}}{Peso_{P_i}} = \frac{periodo * \cancel{Peso_{P_i}} * Peso_{P_0}}{\sum_{j=1}^{n} Peso_{P_j} * \cancel{Peso_{P_i}}} = \frac{periodo * Peso_{P_0}}{\sum_{j=1}^{n} Peso_{P_j}}$$

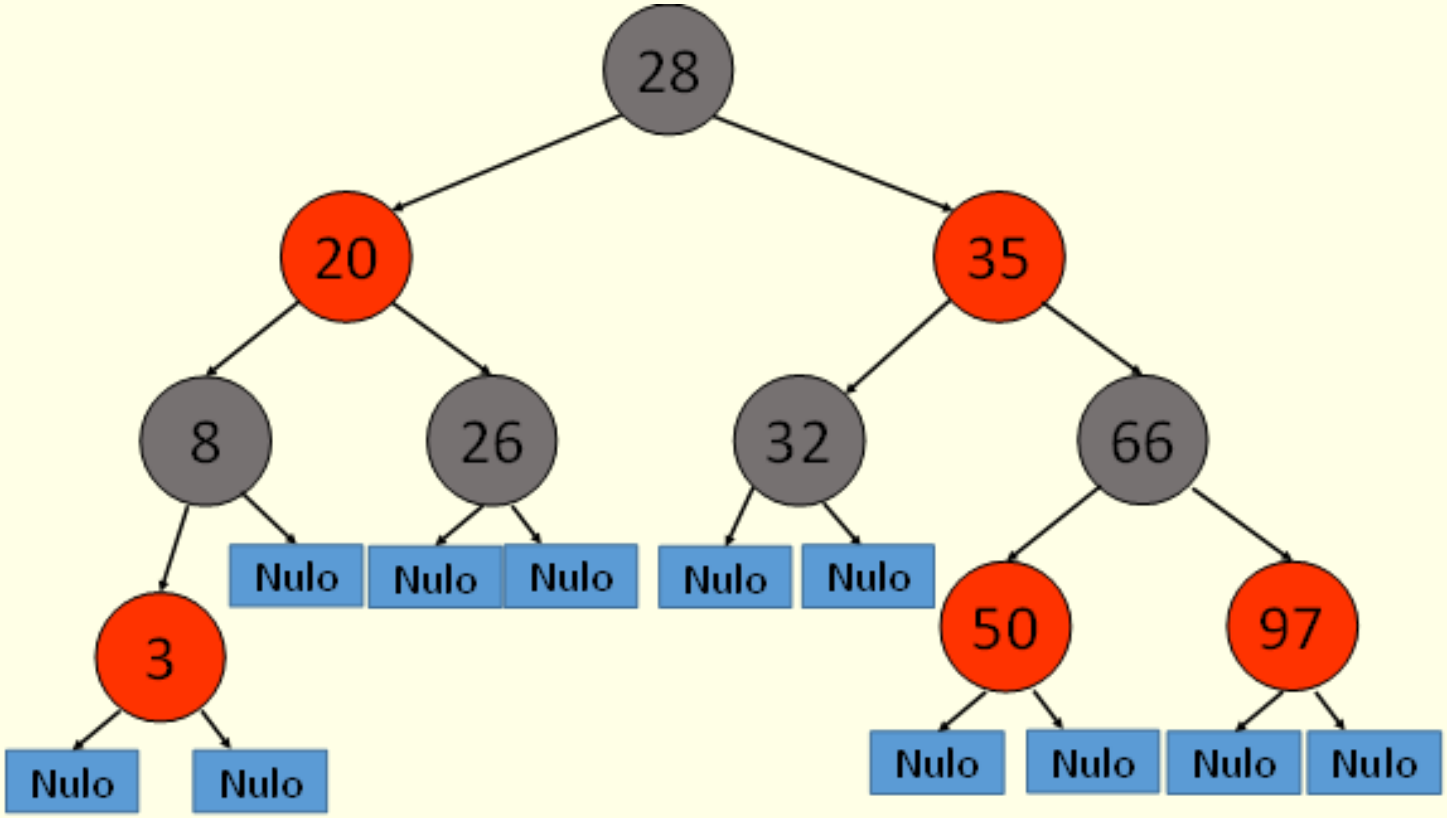
CFS: dinámica del algoritmo

- Cuando proceso i se bloquea o termina rodaja actualiza su vruntime
  - Añadiendo tiempo ejecutado (T) normalizado con peso prio 0

$$vruntime_{P_i} += \frac{T * Peso_{P_0}}{Peso_{P_i}}$$

- Proceso nuevo se le asigna min_vruntime (el menor vruntime actual)
  - Pronto podrá usar la UCP
- Procesos con E/S e interactivos automáticamente beneficiados
  - Su vruntime se queda parado
  - Para evitar proceso bloqueado acapare UCP, en desbloqueo
    - $vruntime_{P_i} = máximo(vruntime_{P_i}, min_vruntime - sched_latency)$
- Implementación O(log N): requiere árbol binario *red-black tree*

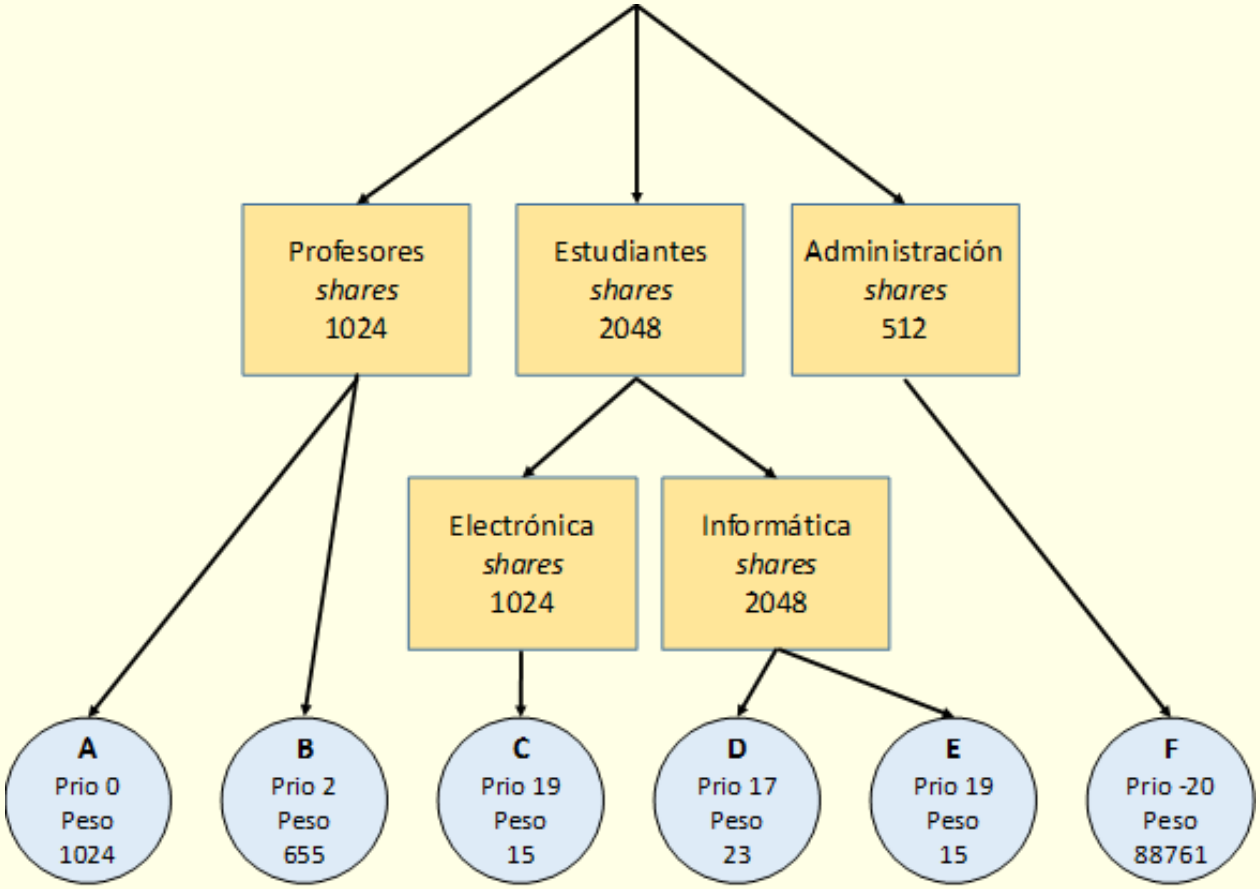
CFS: RBT


28
20
35
8
26
32
66
Nulo
Nulo
Nulo
Nulo
Nulo
3
50
97
Nulo
Nulo
Nulo
Nulo
Nulo
Nulo

Fair-share scheduling usando CFS

- Por defecto CFS reparto equitativo solo entre procesos
  - Pero también permite planificar grupos de procesos
  - ¿Cómo crear grupos? → 4ª parte
- Planificador gestiona entidades planificables: procesos o grupos
- Cada grupo tiene su propio RBT
  - Que contiene sus procesos o subgrupos
- Selección de proceso a ejecutar:
  - Empieza en RBT nivel superior
    1. Busca en RBT la entidad con menor vruntime
    2. Si es proceso → selecciona ese proceso y termina
    3. Si es un grupo → volver a paso 1 sobre el RBT de ese grupo
- Actualización de *vruntime* se propaga hacia arriba en jerarquía
- Permite cualquier esquema de niveles de *fair-share scheduling*

CFS: Ejemplo de planificación de grupos



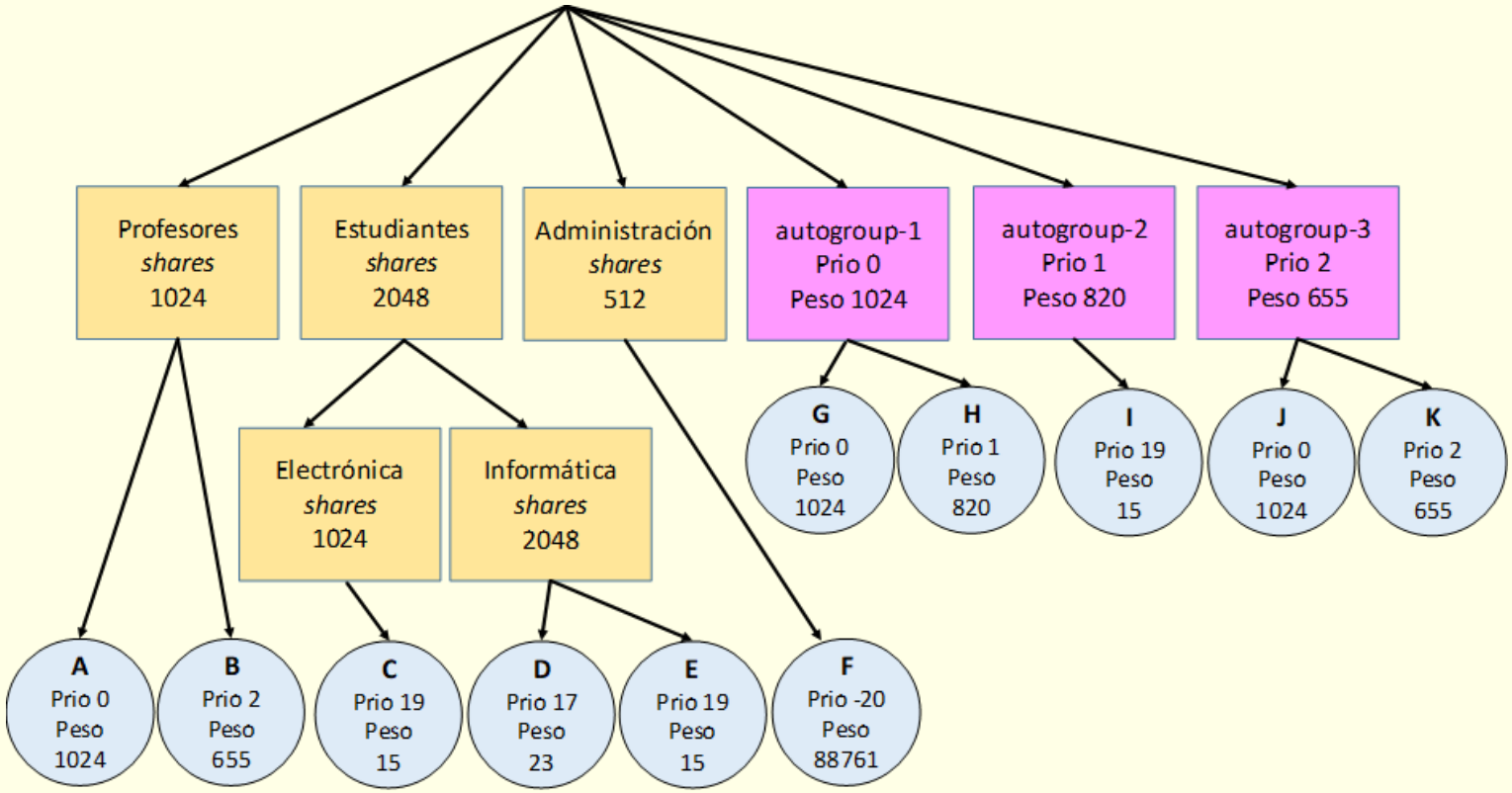
CFS: Ejemplo de planificación de grupos

$$"Rodaja"_D = \frac{periodo * Peso_D}{Peso_D + Peso_E} * \frac{Share_{inf}}{Share_{elec} + Share_{inf}} * \frac{Share_{est}}{Share_{prof} + Share_{est} + Share_{adm}} =$$

$$\frac{6 * 23}{23 + 15} * \frac{2048}{1024 + 2048} * \frac{2048}{1024 + 2048 + 512} = 1{,}38\, ms.$$

<table>
<tr><th>Pr</th><th>Rodaja</th><th>Cuota</th><th>vruntime
nivel proceso</th><th colspan="2">Vruntime
nivel intermedio</th><th colspan="2">Vruntime
nivel superior</th></tr>
<tr><td>A</td><td>1,05 ms</td><td>17,5%</td><td>1,05 ms</td><td colspan="2">——</td><td rowspan="2">Prof.</td><td rowspan="2">1,05+0,67
=1,72 ms</td></tr>
<tr><td>B</td><td>0,67 ms</td><td>11,17%</td><td>1,05 ms</td><td colspan="2">——</td></tr>
<tr><td>C</td><td>1,14 ms</td><td>19%</td><td>78,02 ms</td><td>Elec.</td><td>1,14 ms</td><td rowspan="3">Alum.</td><td rowspan="3">0,58+0,69+0,45
=1,72 ms</td></tr>
<tr><td>D</td><td>1,38 ms</td><td>23%</td><td>61,59 ms</td><td rowspan="2">Inf.</td><td rowspan="2">0,69+0,45=
1,14 ms</td></tr>
<tr><td>E</td><td>0,90 ms</td><td>15%</td><td>61,59 ms</td></tr>
<tr><td>F</td><td>0,86 ms</td><td>14,33%</td><td>0,01 ms</td><td colspan="2">——</td><td>Adm.</td><td>1,72 ms</td></tr>
</table>

CFS: *autogroups*

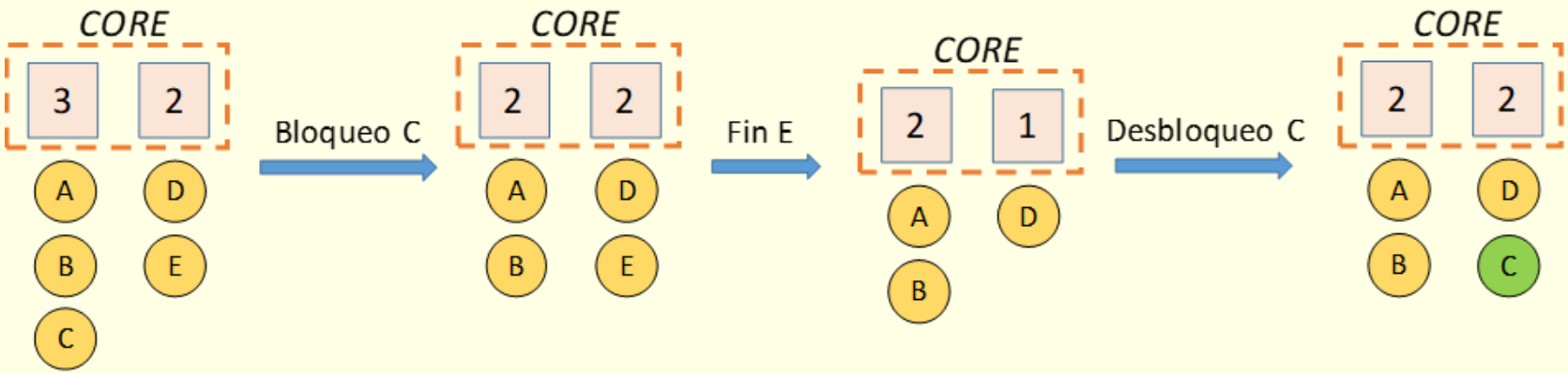


Por cada sesión se crea automáticamente un grupo que incluye todos sus procesos no asignados explícitamente a un grupo

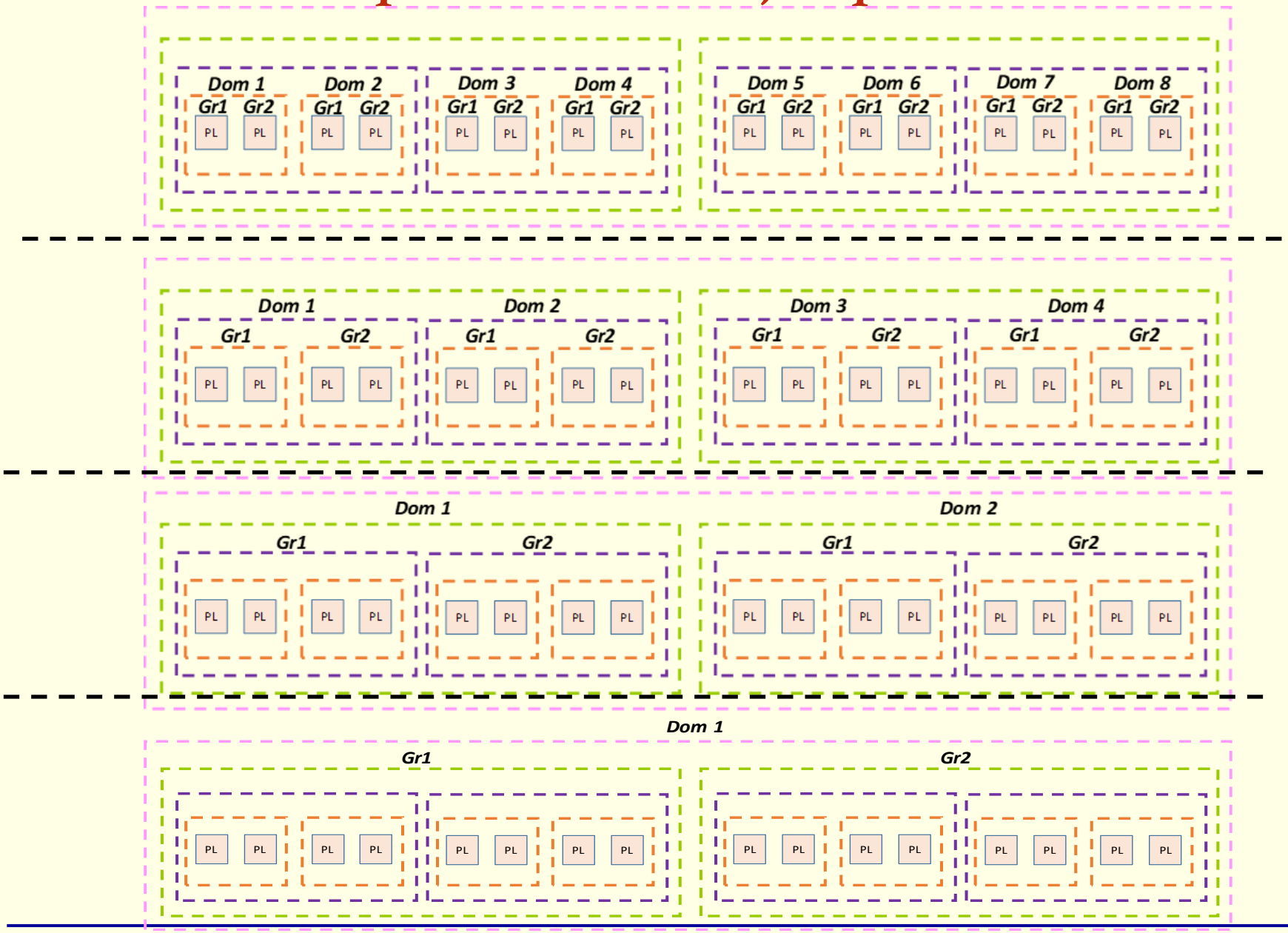
Planificación MP en Linux

- Particionada con migración: similar explicado de forma genérica
  - Añade posible migración al desbloquearse un proceso
  - Puede ser razonable entre UCPs con mucha afinidad
- Gestión de la jerarquía usando concepto **dominio de planificación**
  - Conjunto de procesadores organizados en grupos
  - Objetivo mantener equilibrada la carga de todos sus grupos
- Mecanismo genérico; por cada dominio se definen aspectos como:
  - Frecuencia de comprobación de desequilibrios
    - Mayor cuanta más afinidad exista entre los grupos a equilibrar
  - Umbral de desequilibrio para comenzar migración
    - Menor cuanta más afinidad exista entre los grupos a equilibrar
  - Permitir migración entre grupos de un dominio
    - Normalmente, no se permite entre nodos NUMA

Ejemplo de migración en desbloqueo



Dominios de planificación: ejemplo con 4 niveles



Sistemas operativos avanzados

Planificación del procesador

4ª parte: planificación y virtualización

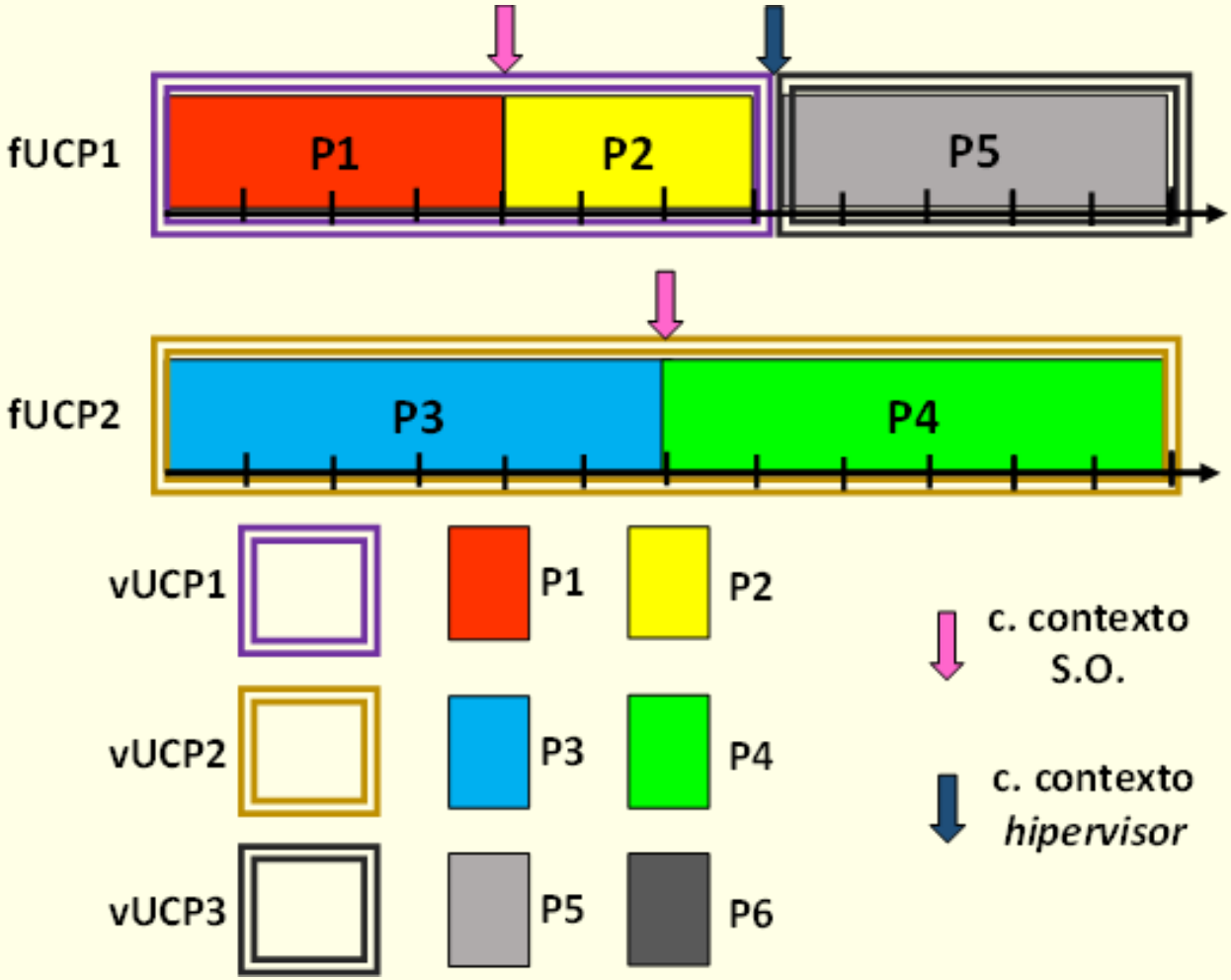
Planificación y virtualización

- Virtualización concierne al SO en dos vertientes:
  - Soluciones de virtualización tradicionales
    - Hipervisor como gestor de recursos comparte aspectos con SO
  - Virtualización soportada por el SO: contenedores
    - Vinculada a la gestión de grupos de procesos por el SO
- Con respecto a la planificación
  - ¿Nuevos algoritmos para planificar máquinas virtuales?
  - ¿Nuevos algoritmos para planificar contenedores?
- Índice de esta sección:
  - Planificación en la virtualización tradicional
  - Gestión de grupos de procesos
  - Linux *namespaces* y *cgroups*
  - Implementación de la virtualización soportada por el SO

Planificación en la virtualización tradicional

- Similitudes entre planificación de procesos y de máquinas virtuales
  - Aunque hay diferencias significativas:
    - MVs usan más recursos, viven más tiempo y hay menos instancias
- Estados MV similares a los del proceso
  - Lista
  - En ejecución
  - Parada: SO alojado ha ejecutado instrucción HALT
  - Bloqueada: p.e. esperando resolver fallo de página de hipervisor
    - Nota: Bloqueo de proceso en SO alojado no bloquea MV
      - SO alojado CCV a otro proceso
  - En pausa

Convivencia de planificación de SO e hipervisor



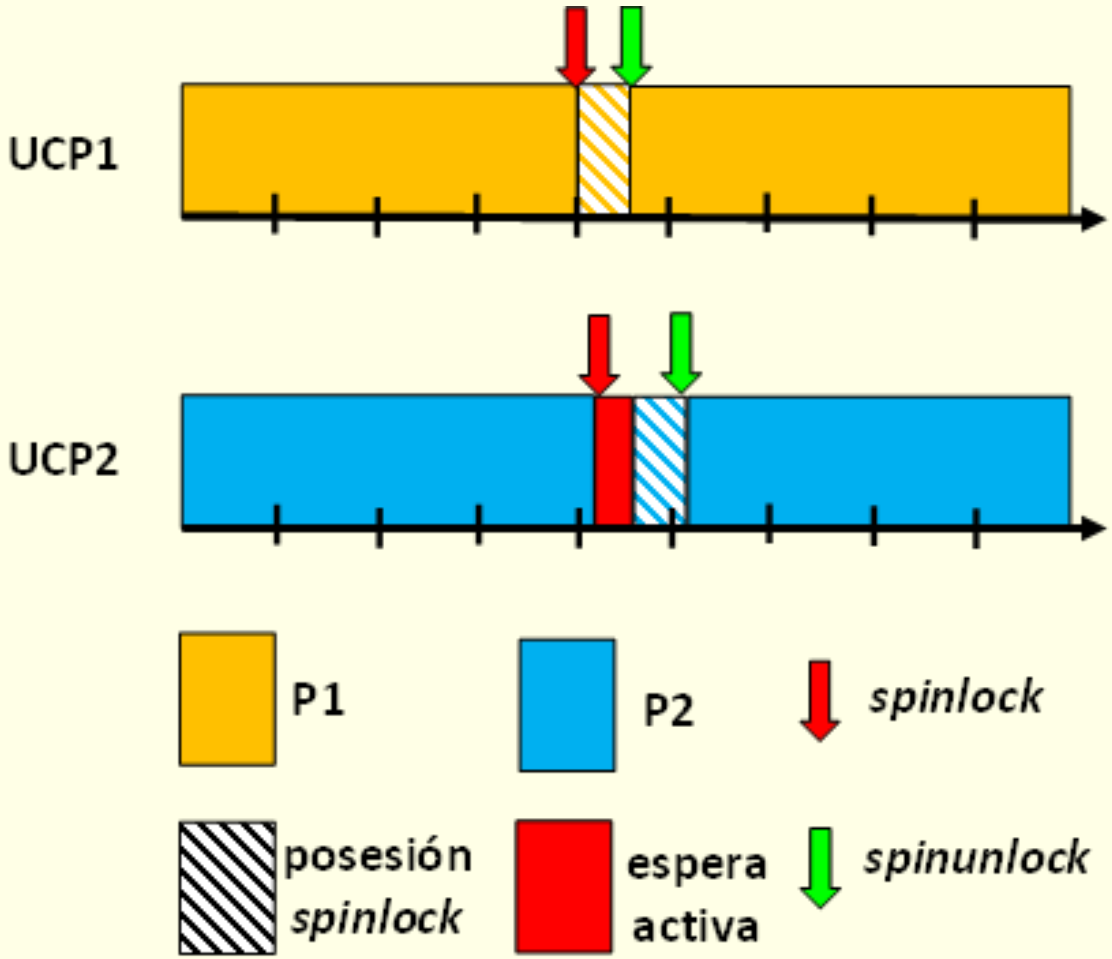
Requisitos del planificador de MV

- Reparto equitativo proporcional al peso de cada MV
- Reserva ancho banda mínimo para MV (*reservations* en Vmware)
- Fijar ancho de banda máximo para una MV (*limits* en Vmware)
- Favorecer MVs con más E/S
- Control de afinidad estricta y aprovechamiento de afinidad natural
- Gestión adecuada de multiprocesador jerárquico
  - Aprovechar afinidad natural jerárquica
  - Intentar evitar MVs en procesadores lógicos del mismo *core*
  - Intentar no mover MV a otro nodo NUMA
- Mecanismos asignar recursos a grupos (*resource pools* en Vmware)
- Dados los requisitos parece adecuado uso de planificador *fair-share*
  - KVM usa CFS para planificar sus MVs

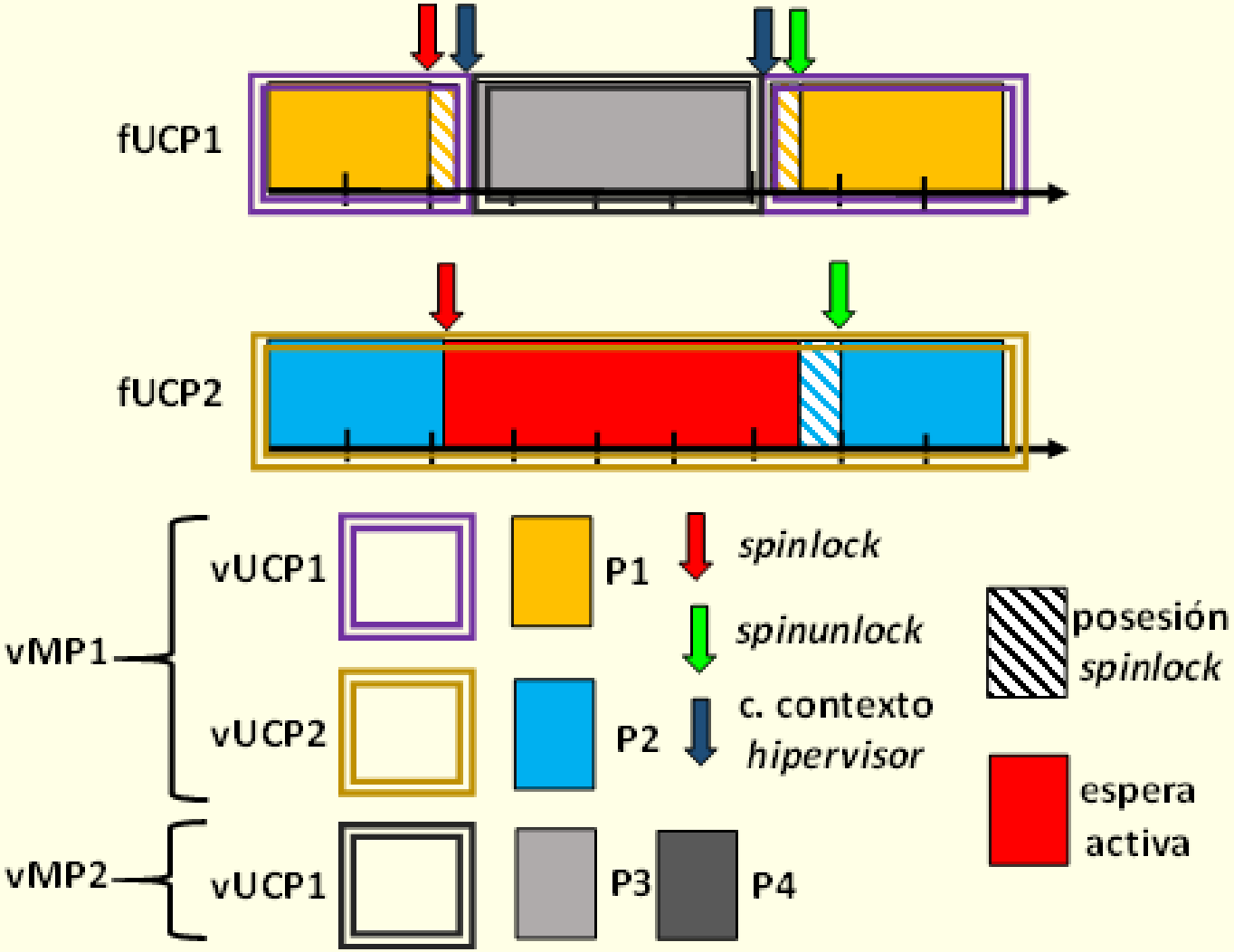
Multiprocesadores virtuales (MPV)

- MV con múltiples procesadores (NP)
  - Por eficiencia, NP ≤ nº *cores*
- Dos aspectos que difieren entre un MPV y un MP real
  - En MP real el tiempo corre de forma continúa
  - En MP real la topología no cambia
- Primer aspecto causa problemas con mecanismos con espera activa
  - Como los *spinlocks*
- Alternativas para afrontar ese primer problema:
  - Planificación por pandas (aka co-planificación):
    - MVs de MVP se planificación juntas; Problema de fragmentación
  - Co-planificación relajada: no planificación conjunta pero
    - Se fija un máximo desfase entre las MVs de un MVP
    - Si se alcanza, MV más adelantada espera por las otras

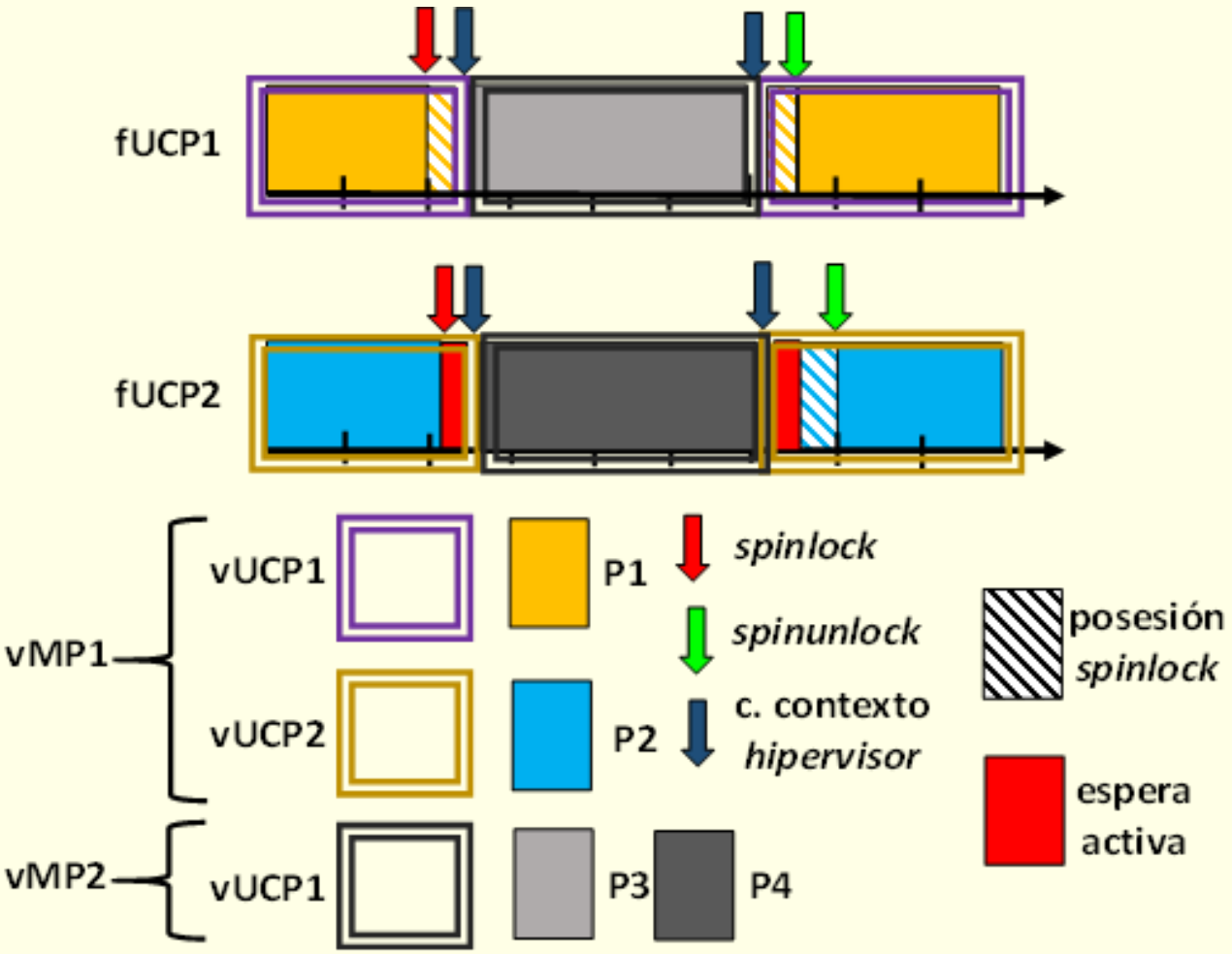
Uso de *spinlock* en MP real



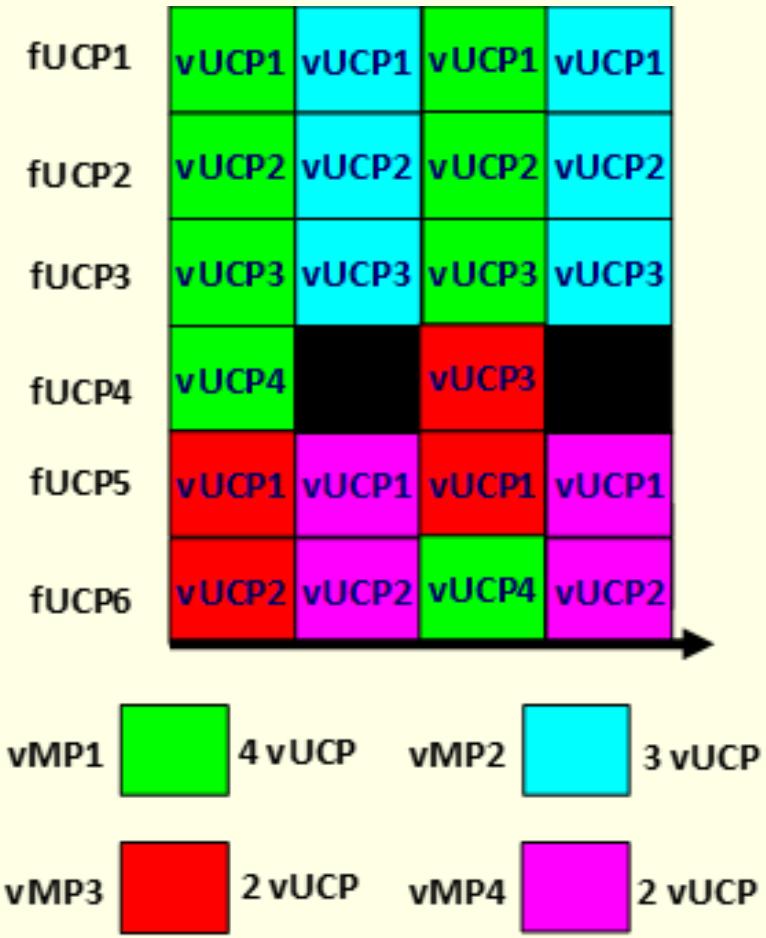
Uso de *spinlock* en MPV



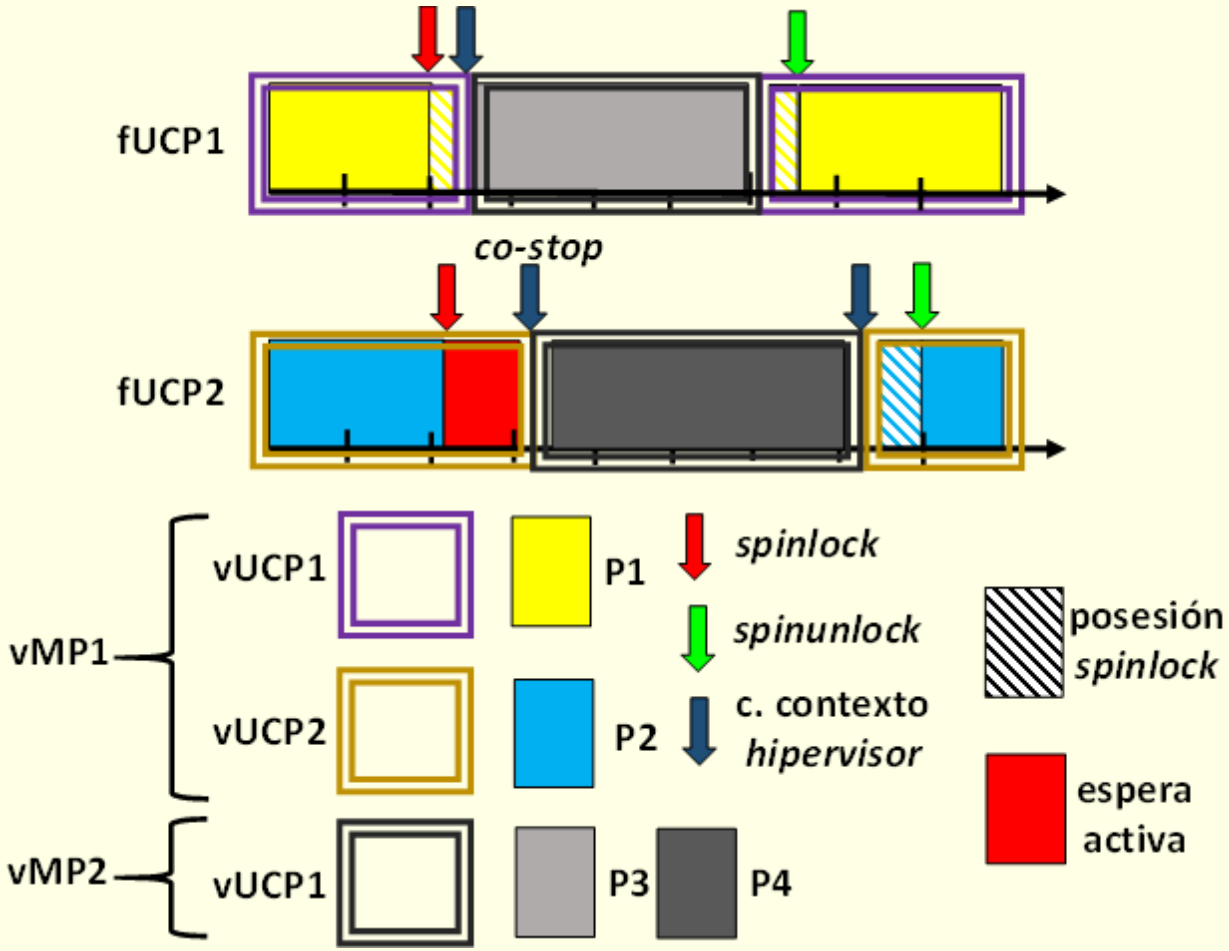
Uso de *spinlock* con planificación por pandas



Fragmentación con planificación por pandas



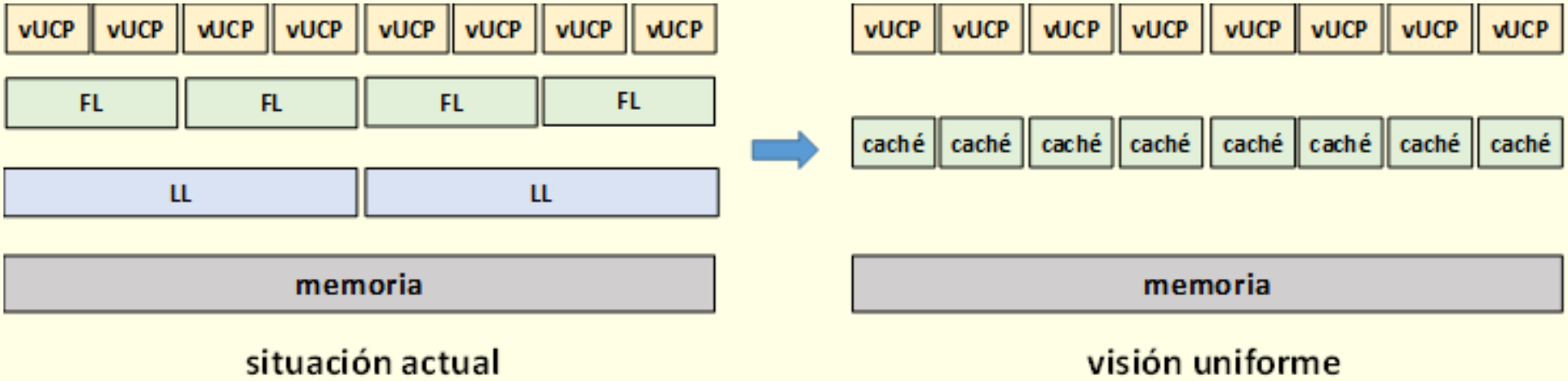
Uso de *spinlock* con co-planificación relajada



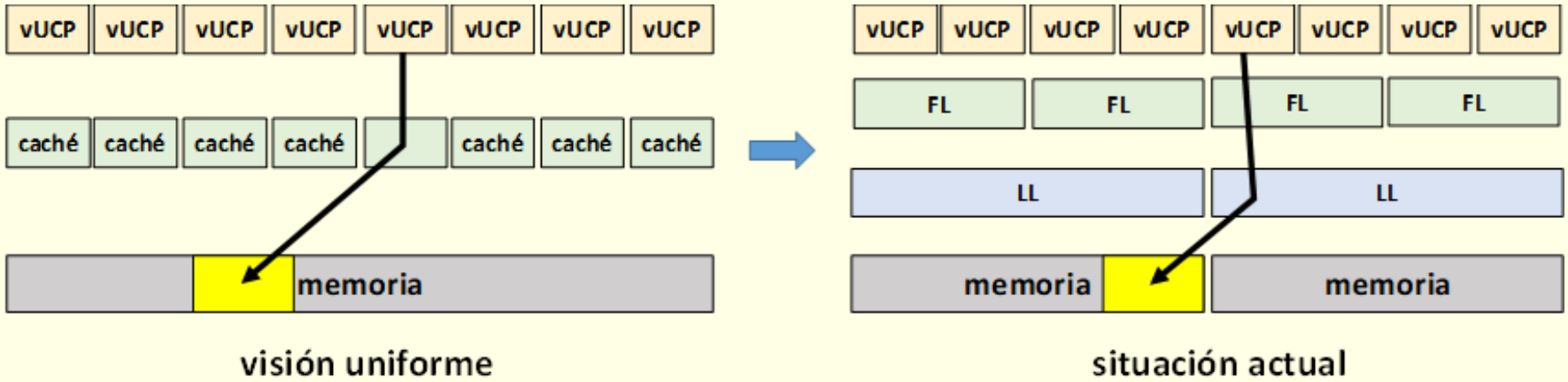
Topología del MPV

- SO alojado realiza planificación adaptada a topología del MP
- Pero en entorno virtual topología del MVP puede cambiar
  - Hipervisor puede migrar MV a otro procesador
    - P.e. a un procesador de otro *socket*
  - Optimizaciones de SO alojado basadas en topología fracasan
- Soluciones:
  - *Firmware* virtual que notifique a SO cambios en topología
    - SSOO no están preparados para ese dinamismo
  - Ofrecer una visión uniforme del MPV
    - Se pierde el aprovechamiento de la afinidad jerárquica
  - Visión uniforme en NUMA no es factible por ineficiente
    - vNUMA: visión uniforme dentro de cada nodo

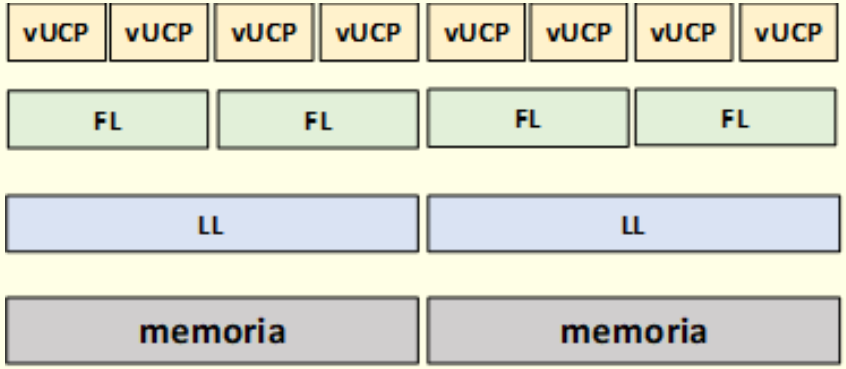
Visión uniforme de un MPV



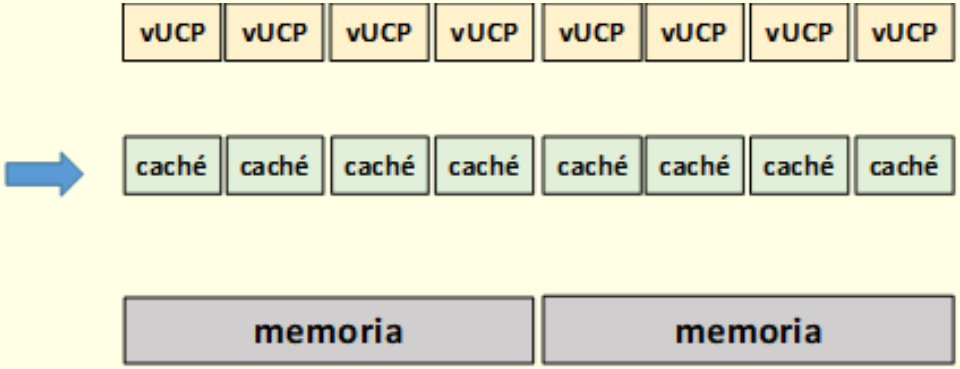
Visión uniforme no factible en MP NUMA



vNUMA



situación actual



visión uniforme vNUMA

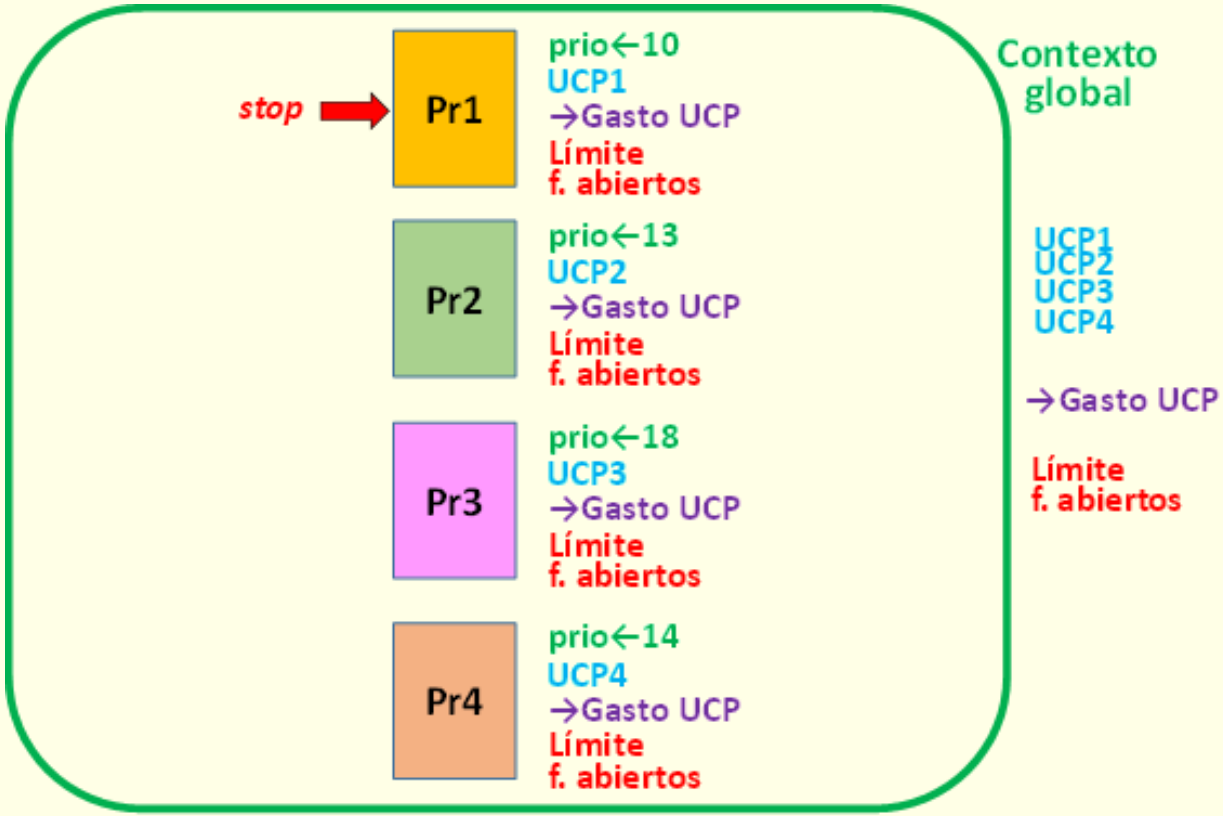
Gestión de grupos de procesos

- Requerido por planificación *fair-share*
  - Tanto uso general como por hipervisor y por contenedores
- Precedentes en la gestión de grupos de procesos en UNIX
  - Procesos de una sesión (SIGHUP a todos en *logout*)
  - Procesos en mandato con tuberías (Crtl-C → SIGINT a todos)
- Uso tradicional: Mecanismo de utilización interna
- Objetivo: mecanismo controlado por el usuario para crear grupos:
  - Para el control de uso de recursos
  - Para la visibilidad de los recursos
- Primero, haremos una descripción genérica de la funcionalidad
- A continuación, estudiaremos su implementación en Linux
  - *Control groups* y *namespaces*

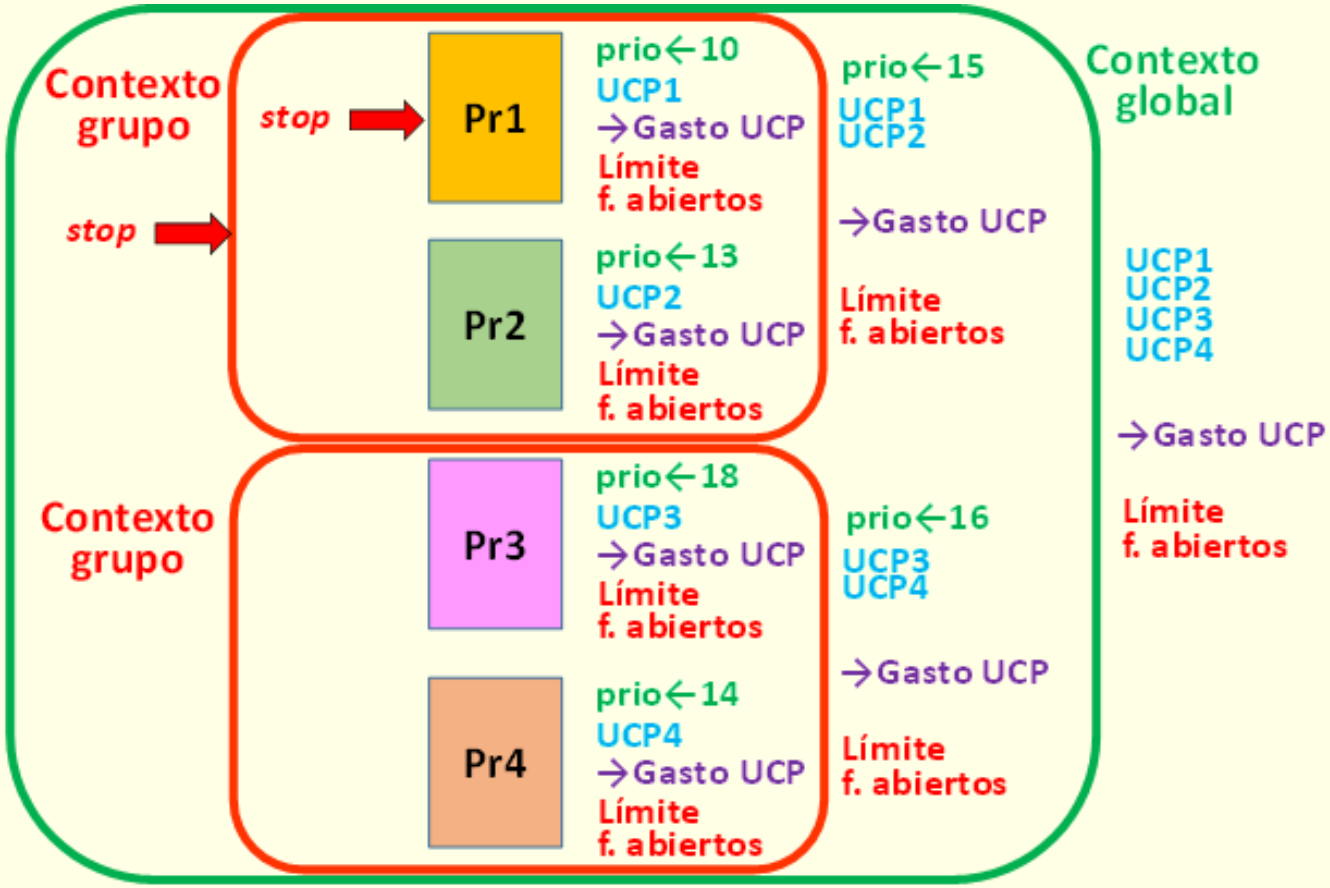
Grupos para control de recursos

- SO ofrece mecanismos para control recursos de proceso y sistema
  - Límite uso de un recurso por parte de un proceso o en el sistema
    - Por ejemplo, nº de ficheros abiertos
  - Contabilidad de uso de un recurso por un proceso o en el sistema
  - Información de planificación de un proceso (p.e. su prioridad)
  - Asignación de recursos a un proceso (p.e. máscara de afinidad)
  - Control de la ejecución de un proceso (p.e. detener su ejecución)
- Nuevo mecanismo de gestión de grupos por parte de un usuario
  - Permite crear jerarquía de grupos
  - Establece límites por grupo
  - Obtiene contabilidad de uso de recursos por grupo
  - Asignación de info. planificación y de recursos por grupo
  - Control de la ejecución por grupos

Control de uso de recursos convencional



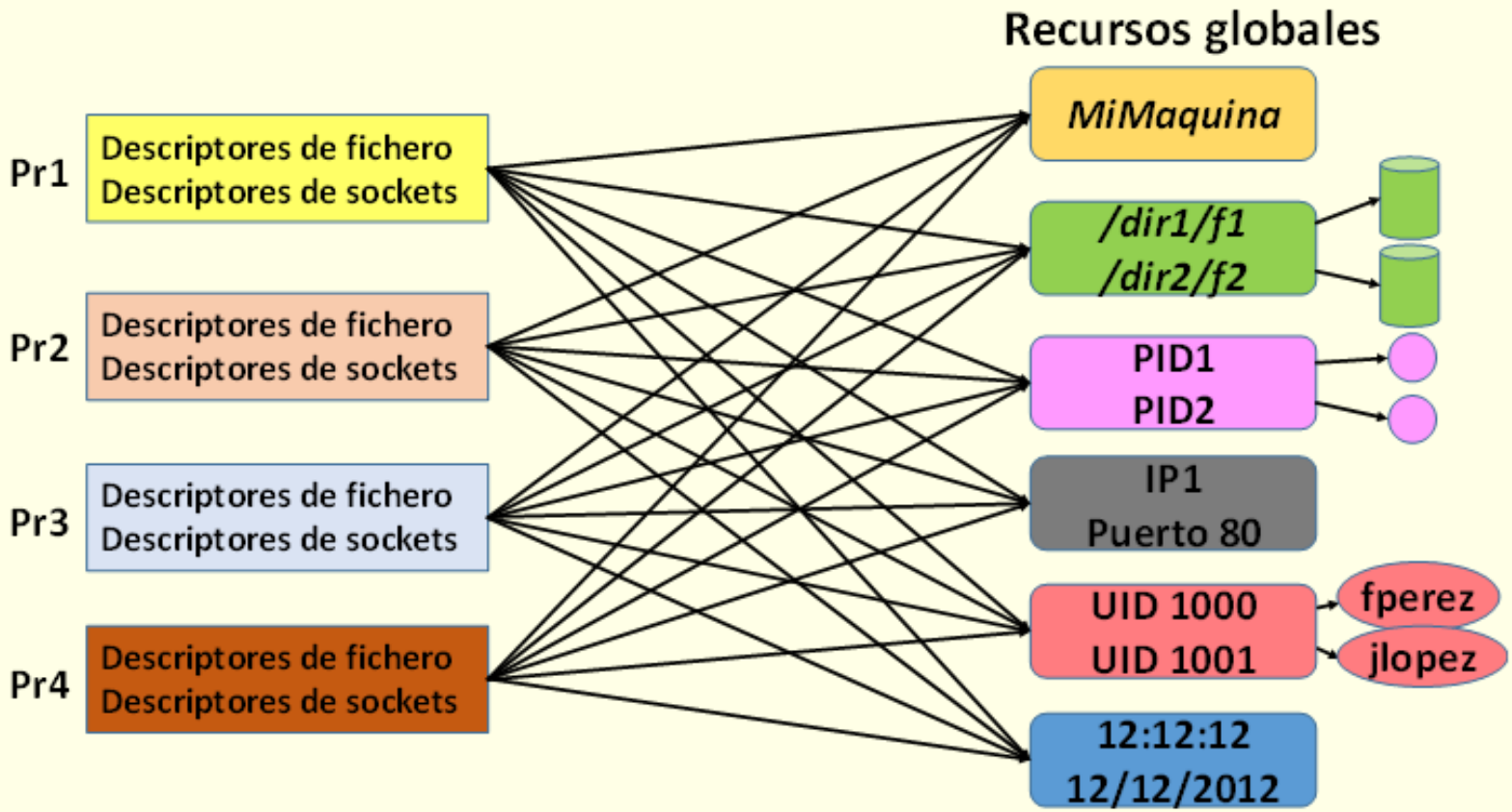
Control de uso de recursos mediante grupos



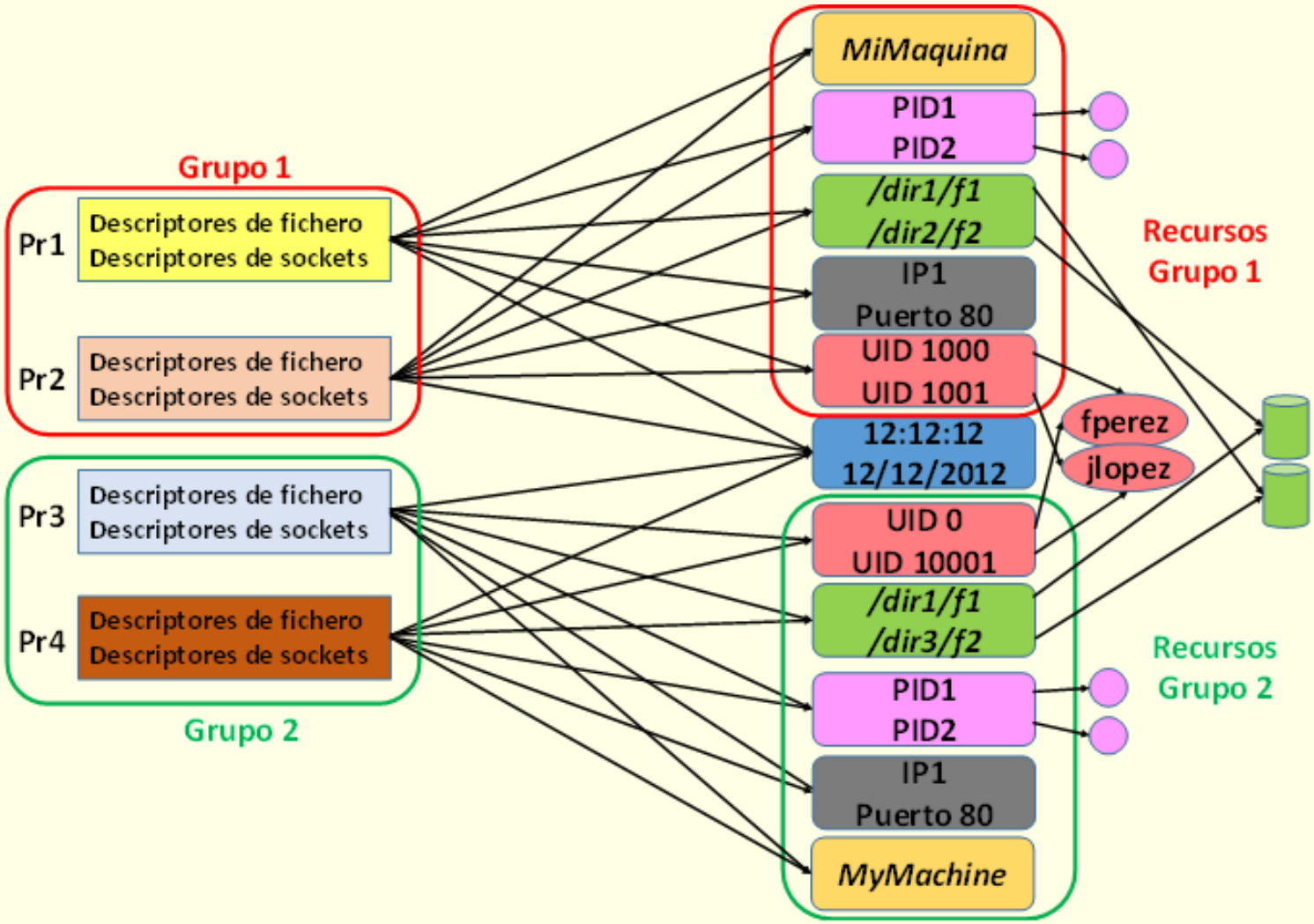
Espacios de nombres para grupos de procesos

- SO tradicional proporciona dos ámbitos de visibilidad
  - Espacio de nombres de proceso (p.e. descriptor fichero abierto)
  - Espacio de nombres de sistema (p.e. un PID)
- Crear un nuevo ámbito de visibilidad para un grupo de procesos
  - Procesos en grupo tienen su propia visión de un recurso global
  - Nombre de la máquina
  - PIDs
  - Árbol de ficheros
  - UIDs/GIDs
  - Red: interfaces, direcciones, puertos…
  - Mecanismos de comunicación y sincronización

Ámbitos de visibilidad en SO convencional



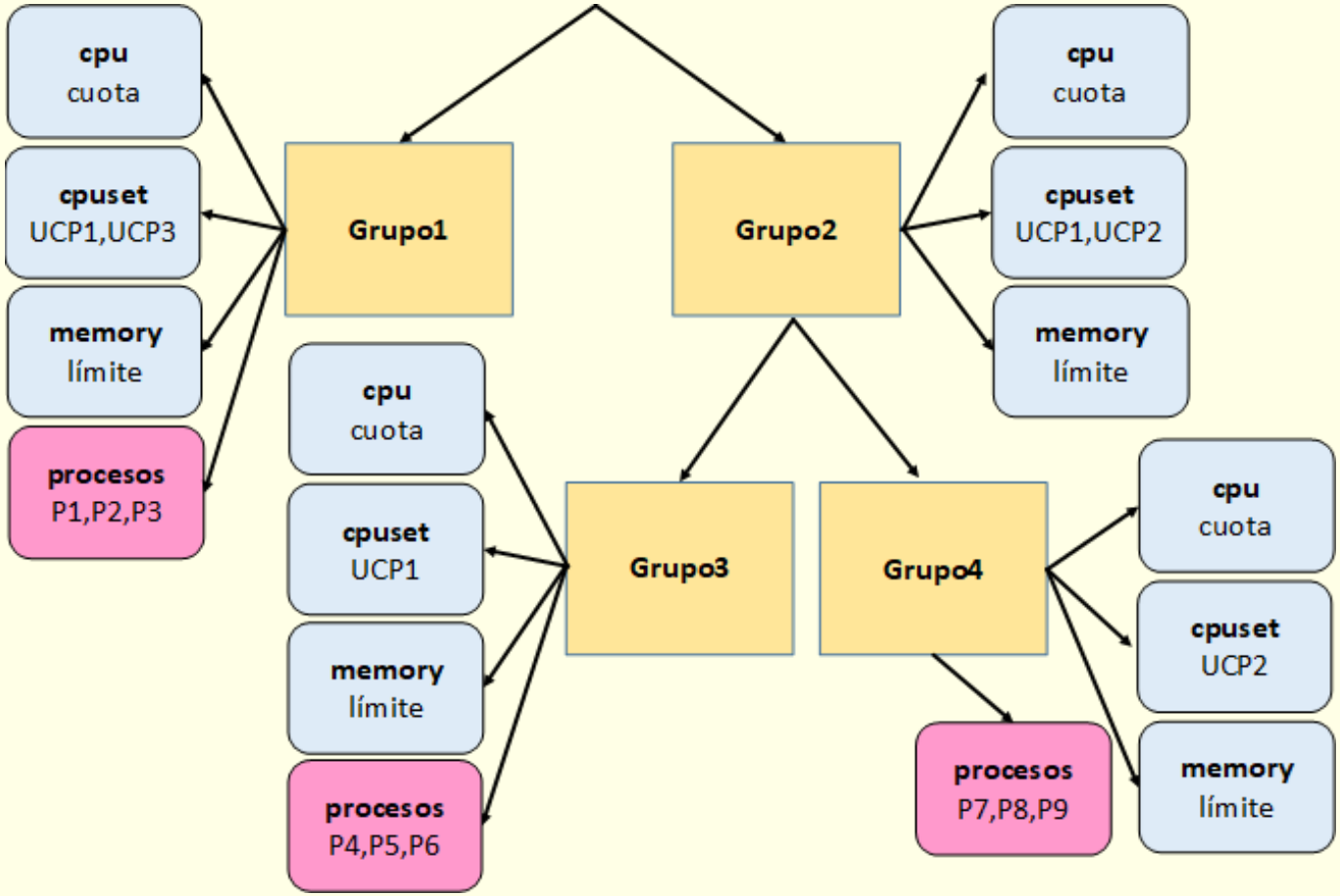
Ámbitos de visibilidad en SO convencional



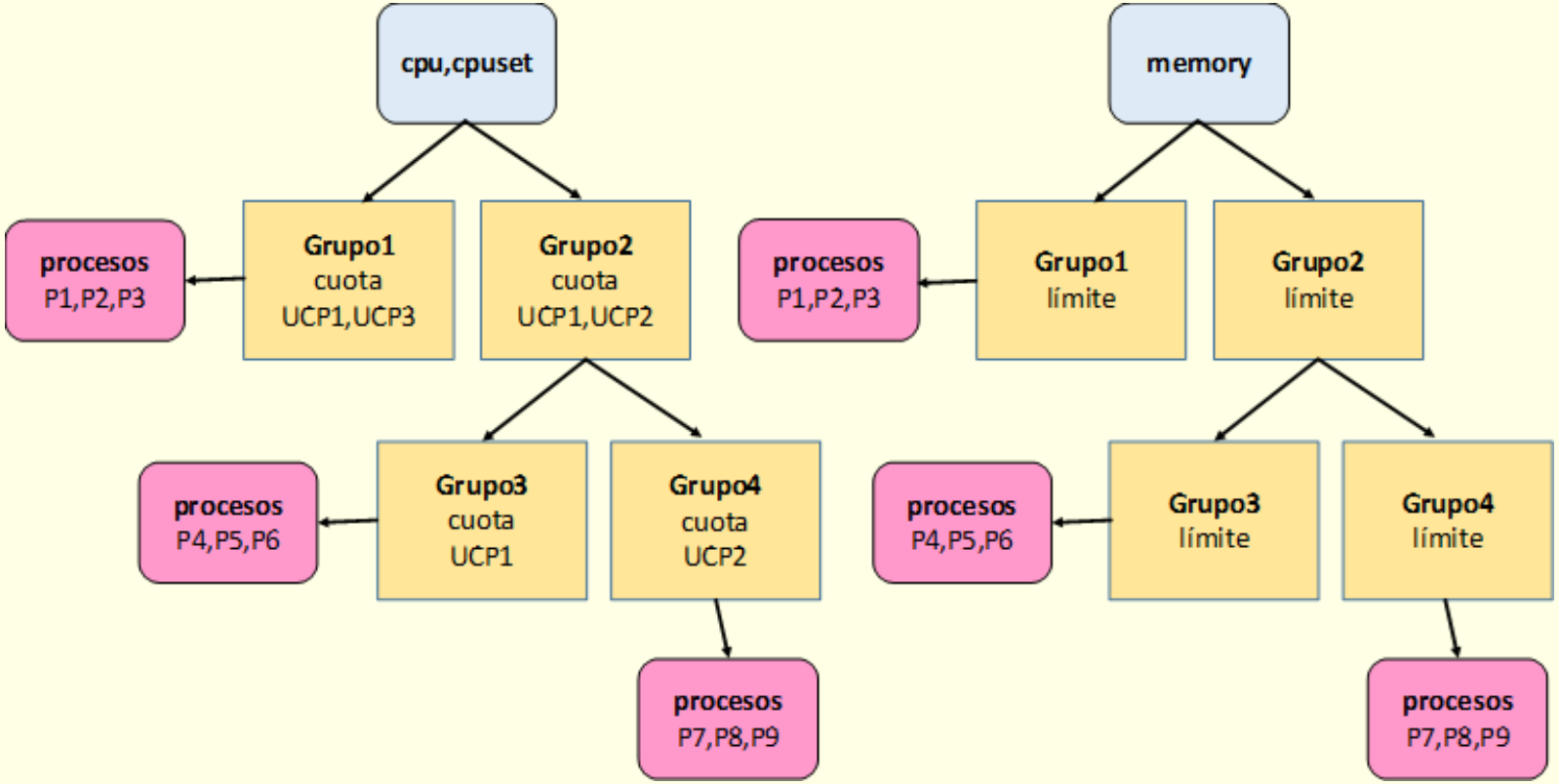
Control groups de Linux

- Grupos de procesos para el control de recursos en Linux
- Diferente a genérico: Jerarquía por tipo de recurso (subsistema)
- No ofrece llamadas al sistema específicas
- API basado en SF de tipo *cgroup* (nueva versión *cgroup2*)
  - Crear un grupo → crear un directorio
  - Añadir proceso a grupo → escribir PID en *cgroup.procs* del grupo
- Cada subsistema ofrece conjunto atributos para controlar recursos
  - Hay un fichero por cada uno; basta con leer/escribir en fichero
- Algunos subsistemas:
  - *cpu*: planificación (p.e. cuota CFS en atributo *cpu.shares*)
  - *cpuacct*: contabilidad uso CPU/grupo
  - *cpuset*: asigna cjto UCPs a grupo (complemento máscara afinidad)
  - *memory*: control uso memoria; *blkio*: control uso disp. bloques
  - *devices*: control acceso a disp.; freezer: parar/reanudar grupo
  - *net_cls*, *net_prio*: control paquetes enviados por grupo
  - *pids*: límite de procesos por grupo

Visión conceptual de la gestión de recursos



Implementación en *control groups*



Ejemplo de *cgroups*: subsistema *freezer*

```
mkdir /sys/fs/cgroup/freezer/relojes
mkdir /sys/fs/cgroup/freezer/relojes/digitales
xclock -update 1 & # reloj analógico
echo $! >/sys/fs/cgroup/freezer/relojes/cgroup.procs
xclock -d -update 1 & # reloj digital
echo $! >/sys/fs/cgroup/freezer/relojes/digitales/cgroup.procs
xclock -d -utime -update 1 & # reloj digital en segundos
echo $! >/sys/fs/cgroup/freezer/relojes/digitales/cgroup.procs

echo FROZEN > /sys/fs/cgroup/freezer/relojes/freezer.state

echo THAWED > /sys/fs/cgroup/freezer/relojes/freezer.state

rmdir /sys/fs/cgroup/freezer/relojes/digitales
rmdir /sys/fs/cgroup/freezer/relojes
```

Ejemplo de *cgroups*: *cpuset* y *memory*

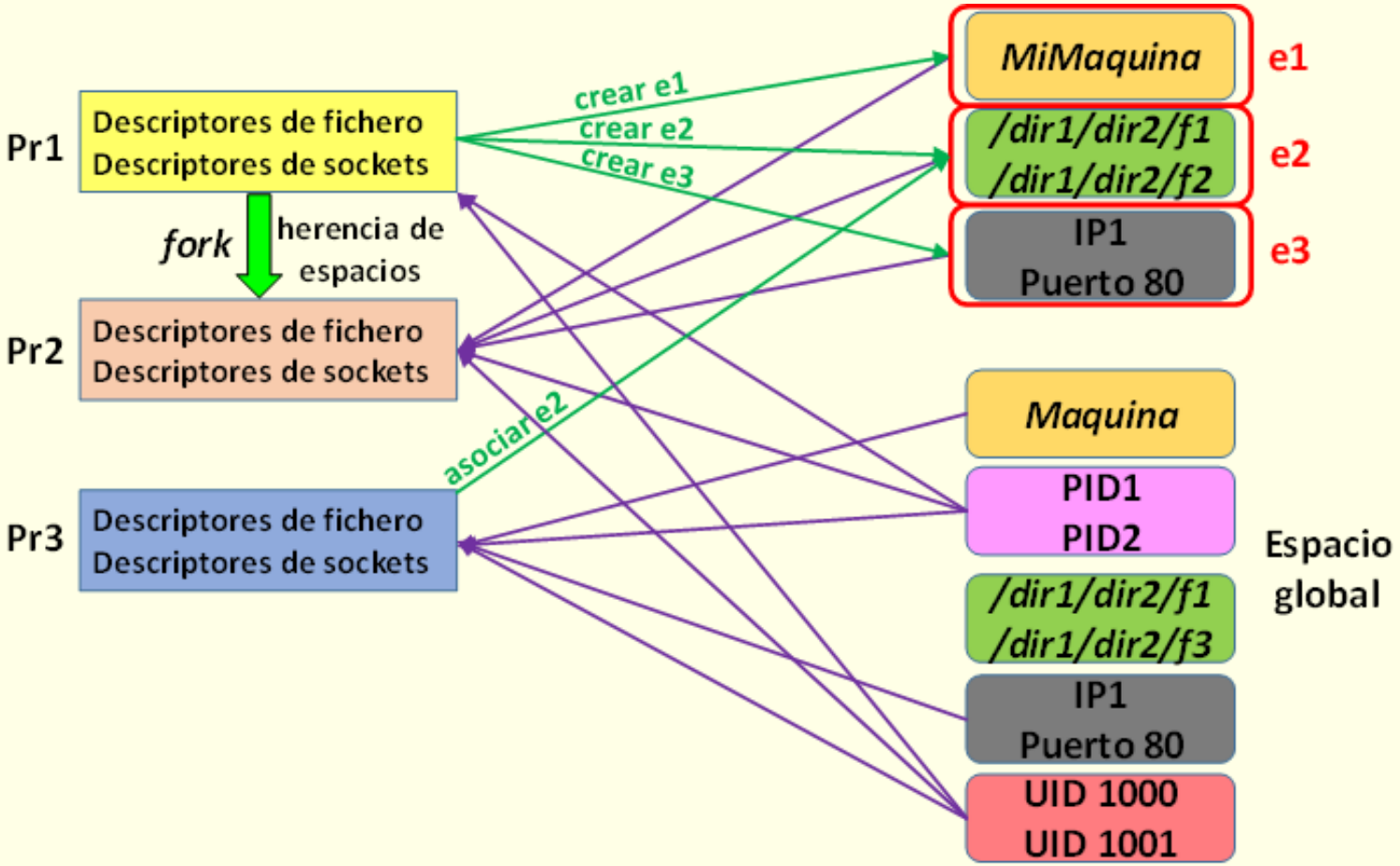
```
mkdir /sys/fs/cgroup/cpuset/grupo1
echo 0-2,4 > /sys/fs/cgroup/cpuset/grupo1/cpuset.cpus
# añade procesos al grupo

echo $PID >/sys/fs/cgroup/cpuset/grupo1/cgroup.procs
rmdir /sys/fs/cgroup/cpuset/grupo1
```

```
# se crea un grupo de procesos para limitar uso de memoria
mkdir /sys/fs/cgroup/memory/mem_limit
# se limita el uso de memoria por parte de los procesos del grupo
echo 256K > /sys/fs/cgroup/memory/mem_limit/memory.limit_in_bytes
# añade procesos al grupo

# se destruye el grupo de procesos
rmdir /sys/fs/cgroup/memory/mem_limit
```

Namespaces de Linux

- Independiente de *control groups*; carácter anidado
- Diferente a genérico: no existe concepto explícito de grupo
  - Proceso crea espacio nombres para un cierto tipo de recurso
    - Para el propio proceso (*unshare*) o al crear un hijo (*clone*)
  - Procesos hijos lo heredan
  - Otro proceso se asocia a un espacio de nombres creado (*setns*)
  - Mandatos *unshare* y *nsenter*
- Espacios de nombres:
  - *mnt*: espacio de nombres de montajes
  - *uts*: espacio de nombres de máquinas
  - *pid*: PID de proceso espacio nuevo ≠ PID en espacio padre
  - *ipc*: mecanismos de IPC de System V
  - *net*: espacio recursos red propios (interfaces,direcciones,puertos..)
  - *cgroup*: oculta pertenencia a subgrupos de *cgroup*
  - *user*: define correspondencia UID/GID entre espacio padre e hijo
- Solo para *root* excepto si se crea a la vez un espacio *user*

Visión conceptual de los *namespaces* de Linux



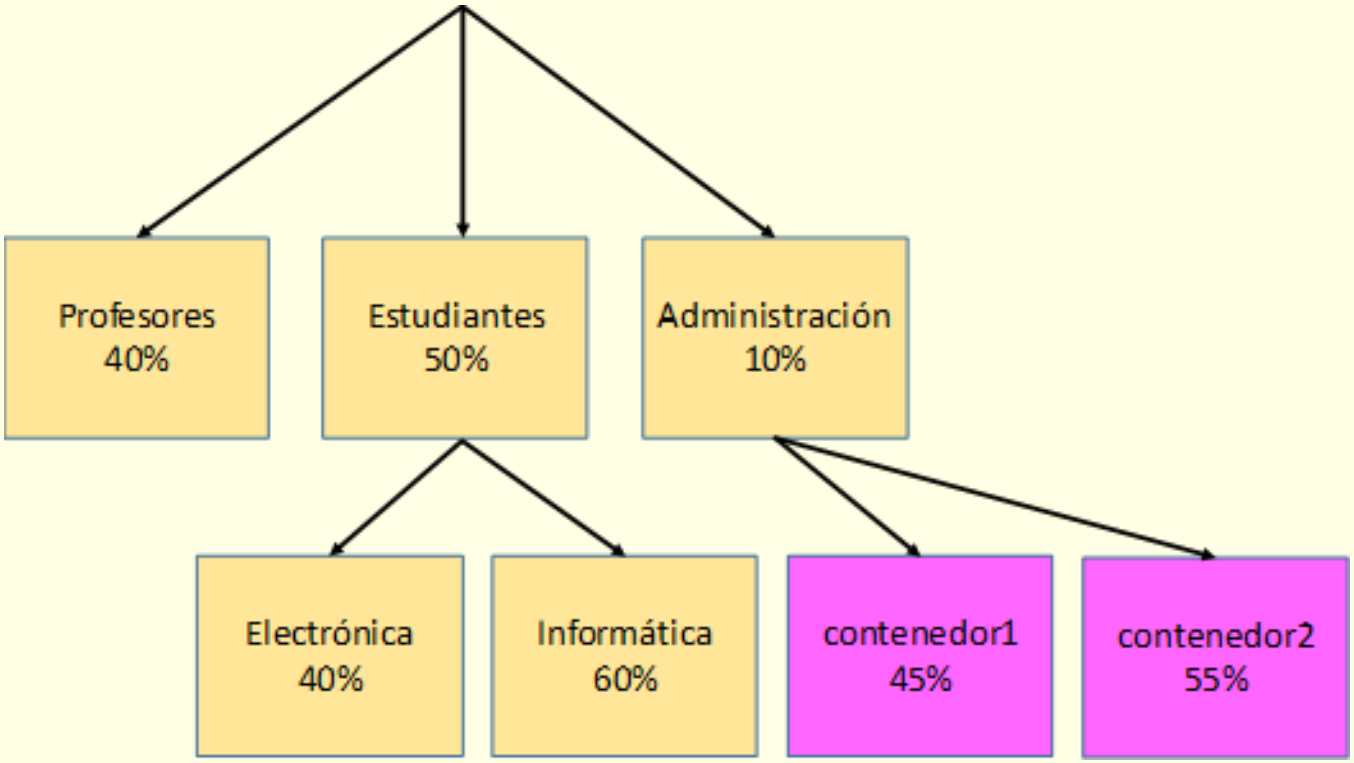
Ejemplo que crea bash con nuevos espacios para todos los recursos

unshare -u -p -i -n -m -r -f --mount-proc # NO requiere root

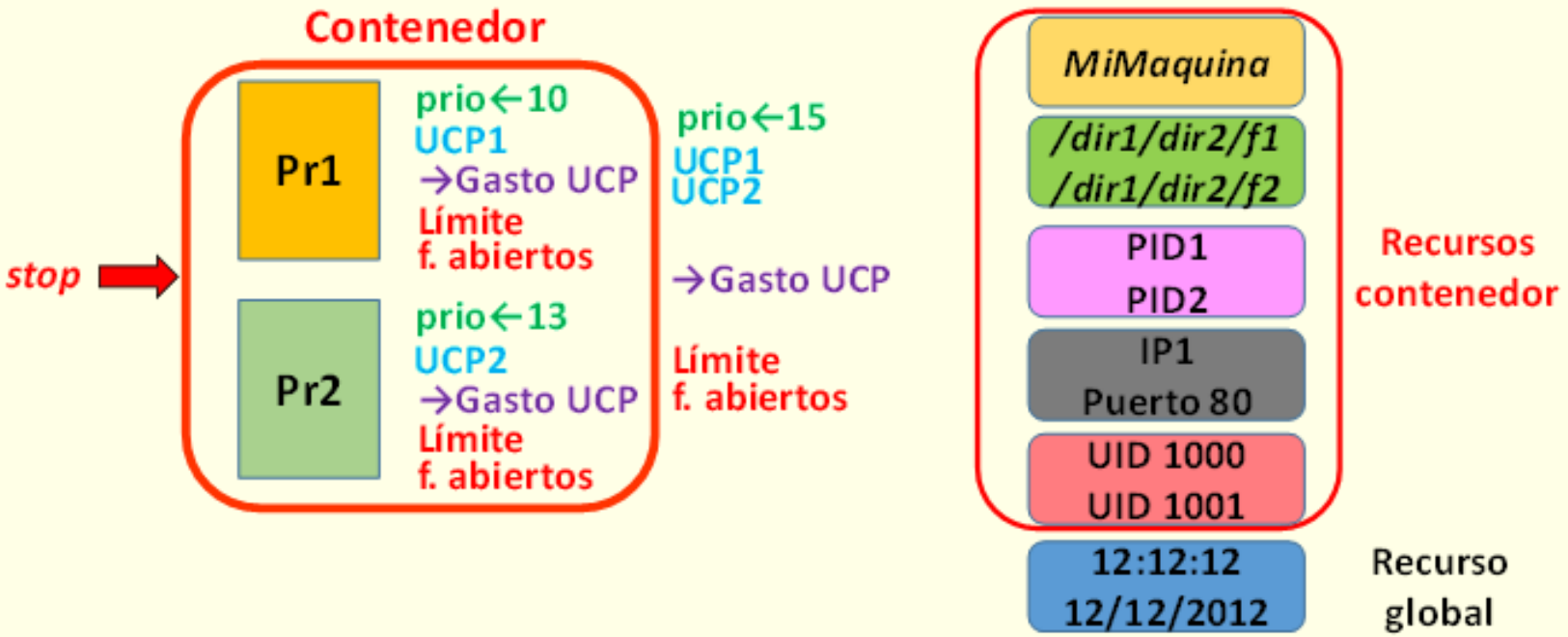
Virtualización basada en contenedores: requisitos SO

- Control uso de recursos por un contenedor
  - procesador, memoria, discos, red…
  - Cuotas de reparto y límites de uso máximo
- Gestionar convivencia procesos contenedor con el resto
- Llevar contabilidad de uso de recursos por contenedor
- Permitir realizar ops. conjuntas sobre procesos de contenedor
  - P.e. detener temporalmente ejecución de todos sus procesos
- Dotar a contenedor su propia visión de recursos (PIDs,*hostname*...)
- Ofrecer a contenedor su propio sistema de almacenamiento
  - Y compartir información entre contenedores y con anfitrión
- Ofrecer a contenedor la funcionalidad de un nodo de red
- Poder configurar correspondencia UIDs contenedor y anfitrión
  - Contenedores privilegiados y no privilegiados
  - Privilegiado: ejecuta en contexto de superusuario del anfitrión
- Mecanismo de grupos de procesos resuelven la mayoría
  - Uso combinado de control de recursos y de visibilidad
- Planificador *fair-share* integra los procesos de los contenedores

Integración de contenedores en la planificación



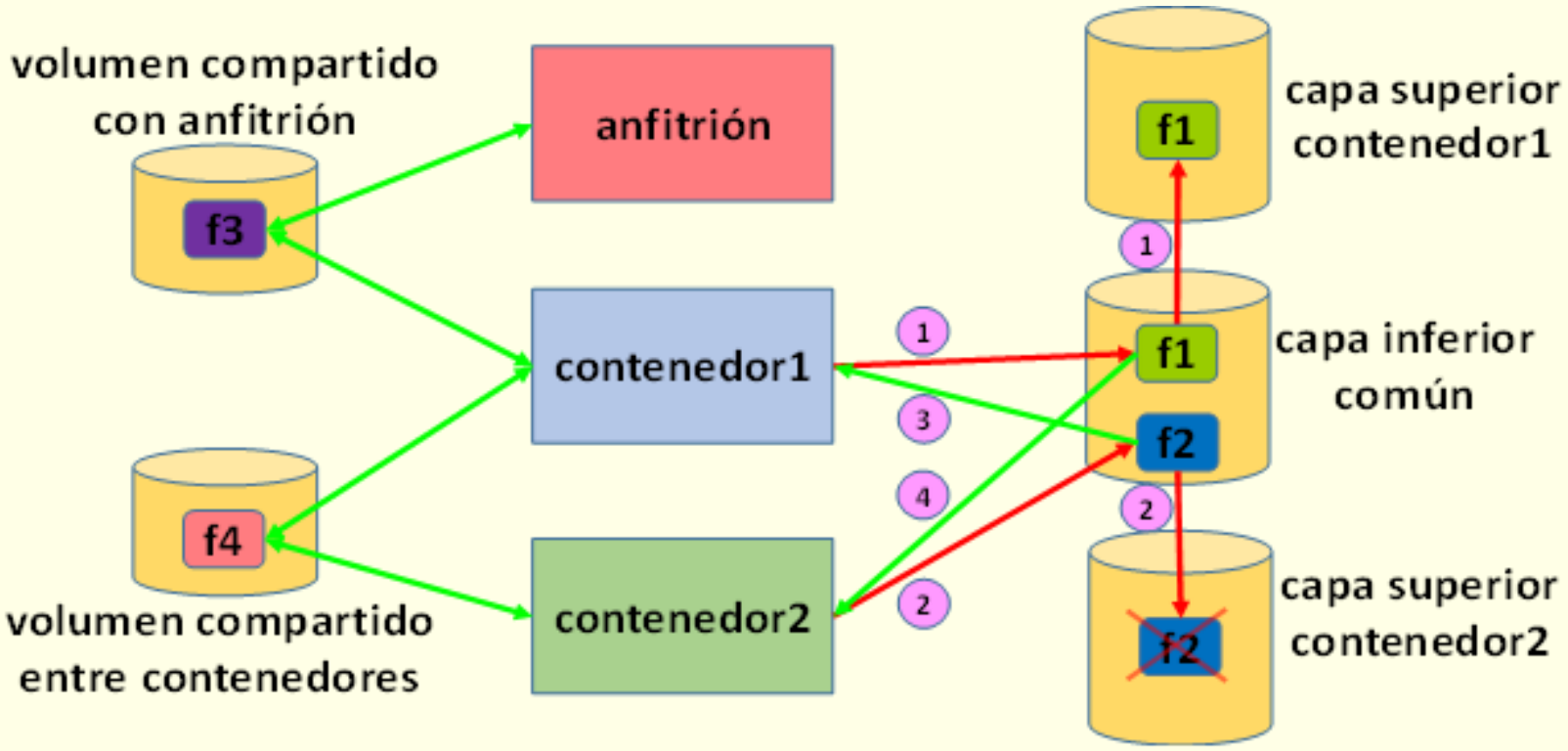
Contenedor usando grupos de procesos



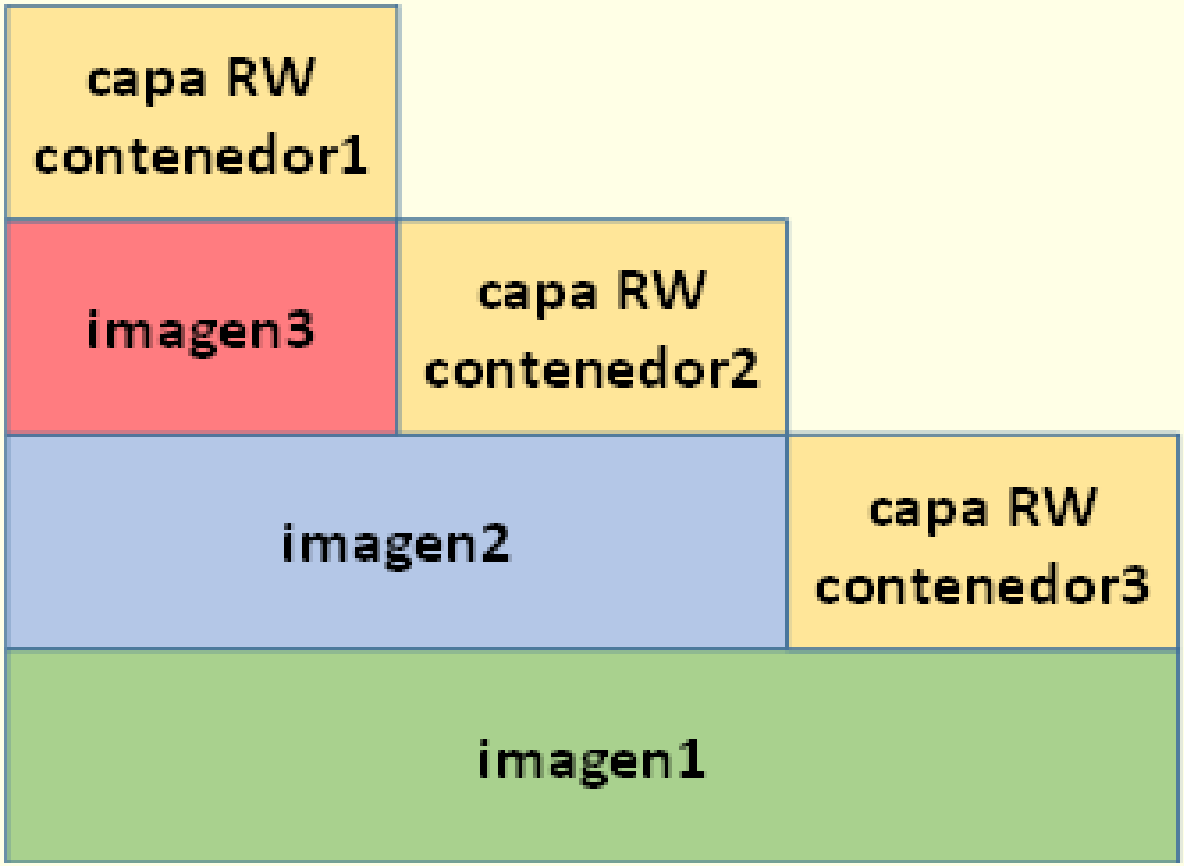
Sistemas de almacenamiento del contenedor

- Dos tipos de almacenamiento: espacio propio y compartido
- Espacio propio en anfitrión puede corresponder a:
  - Fichero (*loop device*, partición, volumen lógico…)
  - Inicialmente contiene la imagen del contenedor
    - Facilidad para compartirla en modo COW con otros contenedores
    - Sobre todo, en contenedores de aplicación que usan capas
  - Uso de *Union File Systems* vs *COW File Systems*
- Volúmenes compartidos
- Ambos tipos requieren espacios de nombres
  - Contenedor tiene su propia visión del SF

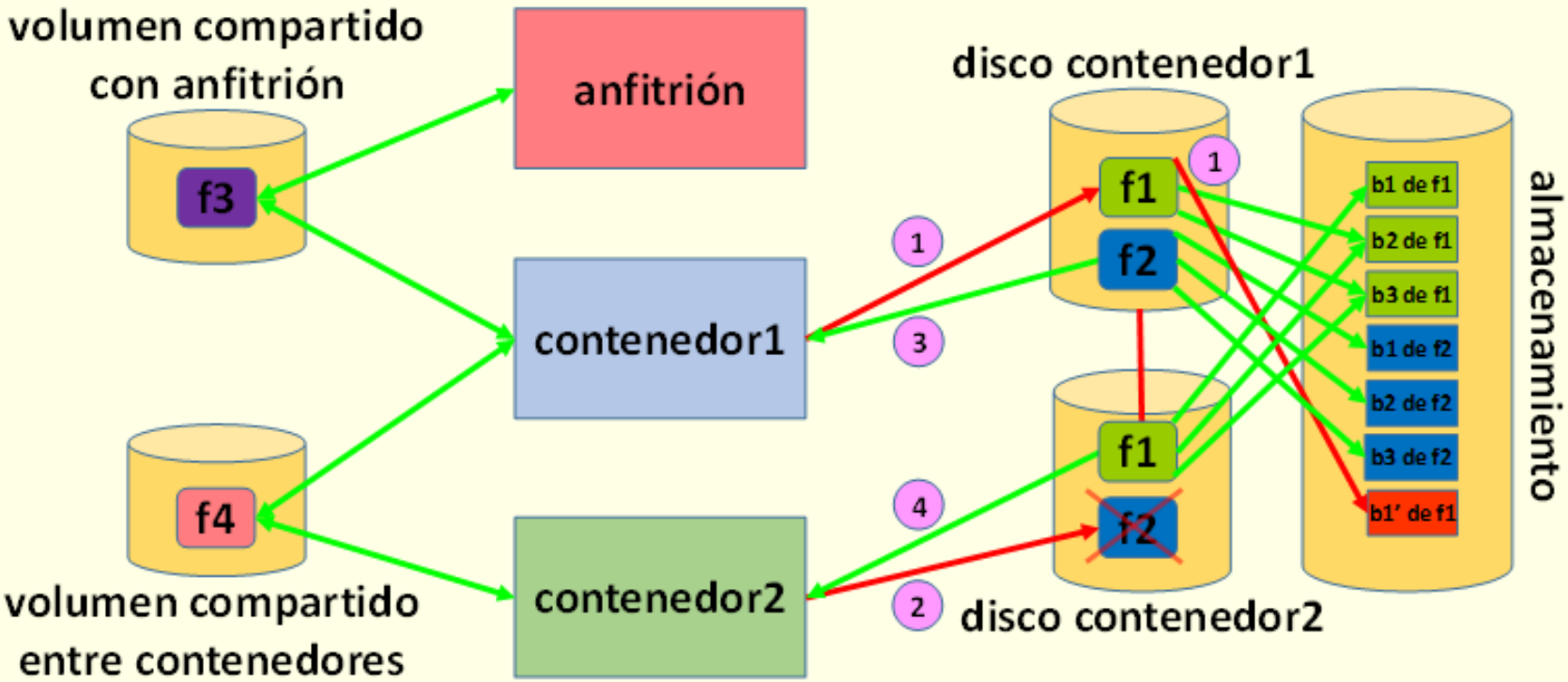
Union File Systems



Compartición de imágenes entre contenedores



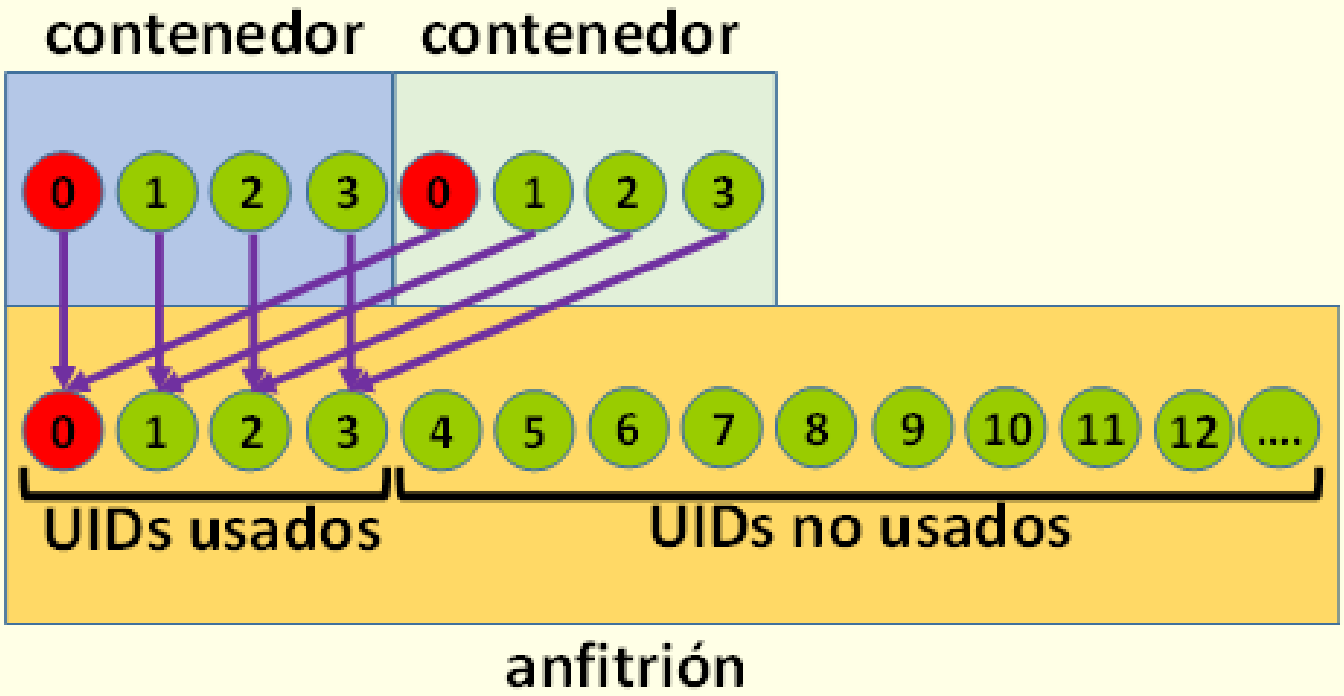
COW File Systems



Gestión de usuarios en un contenedor

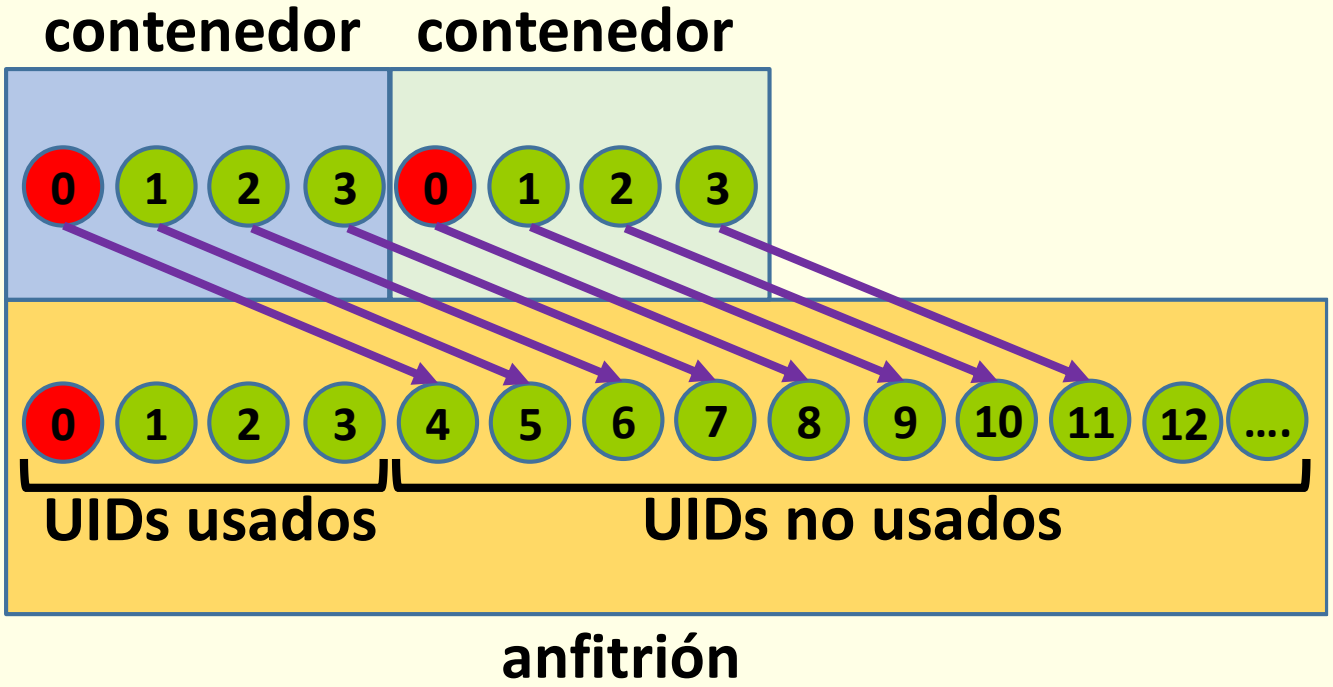
- Contenedores ejecutan en mismo SO
  - En principio, si dos usan mismo UID son el mismo usuario
- Espacios nombres usuario establecen correspondencia entre
  - UID/GID de nuevo espacio y el espacio padre
  - Ej: usuario X en espacio nuevo → Y en el padre
    - Fichero creado por usuario X en nuevo espacio
    - Visible como de usuario Y en espacio padre
- Dos objetivos contrapuestos:
  - Correspondencia disjunta en aras de la seguridad
  - Compartición de UID/GID
    - Facilita compartir ficheros en volúmenes compartidos

Correspondencia directa entre usuarios

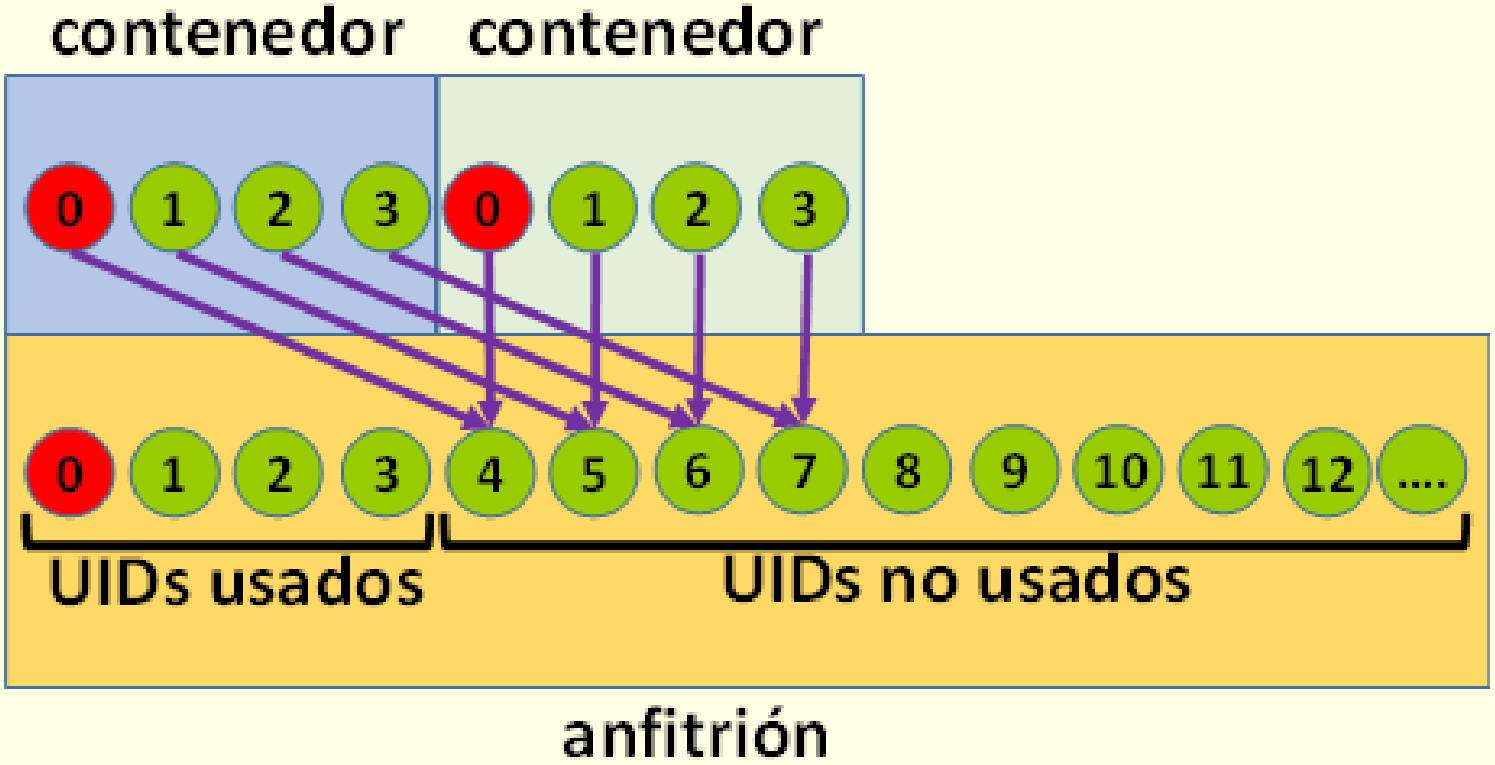


No requiere espacios de nombres de usuarios

Estanqueidad entre usuarios



Compartición de usuarios entre contenedores



¿De qué están hechos los contenedores?

- Linux no da soporte directo a contenedores (ni se pretende)
  - Uso integrado de varias tecnologías:
    - *cgroups*, *namespaces*, *Union File System* vs *COW File System*
    - Otros: *capabilities*, *seccomp*…
- Algunas acciones para crear un contenedor en Linux:
  - Crear directorio *cgroup*
  - Establecer cuotas y límites mediante *cgroup* creado
  - Crear primer proceso con espacios de nombres requeridos
    - Docker, por defecto, no crea espacio de nombres de usuario
    - Kubernetes mismo espacio de red para contenedores de mismo *pod*
  - Incorporar PID de ese proceso al *cgroup*
  - Selección y creación de mecanismo de almacenamiento
  - Elección mecanismo compartición volumen propio: UFS o COWFS
  - Montar volumen propio como raíz y montar volúmenes compartidos
  - Habilitar conectividad de red virtual configurada
  - Establecer correspondencias de UIDs/GIDs