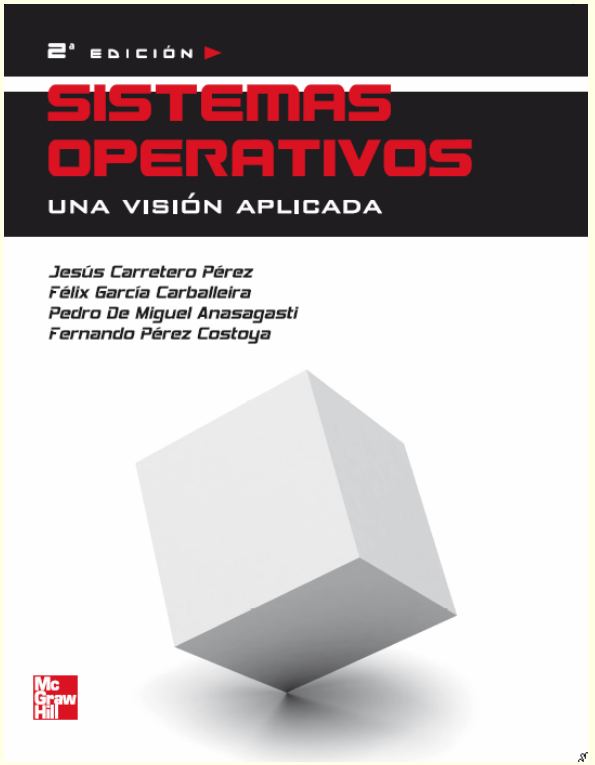
Sistemas operativos
2ª edición

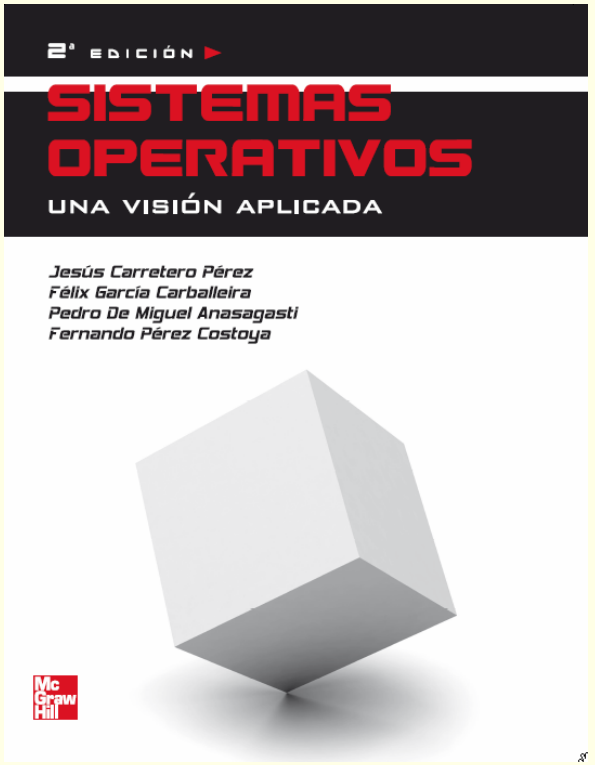


Capítulo 4
Planificación del procesador

Organización del tema

- Primera parte
  - Aspectos generales de la planificación
  - Planificación en sistemas monoprocesador
  - Planificación monoprocesador en Linux
- Segunda parte
  - Soporte para la implementación de contenedores
- Tercera parte
  - Planificación en sistemas multiprocesador
  - Planificación multiprocesador en Linux
- Cuarta parte
  - Planificación de máquinas virtuales
- Quinta parte
  - Planificación de aplicaciones paralelas y sistemas distribuidos

Sistemas operativos
2ª edición



Capítulo 4
Planificación del procesador
1ª parte: planificación en monoprocesador

Contenido

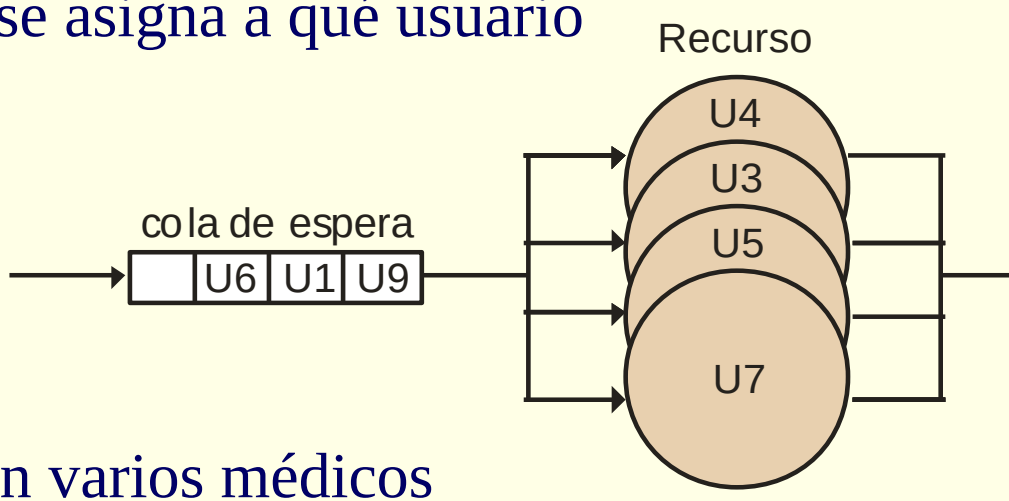
- Introducción
- Caracterización de los procesos
- Objetivos de la planificación
- Niveles de planificación
- Algoritmos de planificación
- Planificación monoprocesador en Linux
  - O(1) vs. CFS

Introducción

- SO planifica recursos: UCP recurso más importante
- 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
  - Tiempo real (no crítico): cumplir plazos
- Planificación 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



- Ejemplo: clínica de salud con varios médicos
  - Cualquier médico puede atender a cualquier paciente
  - Estados del paciente (estados del proceso):
    - Nuevo → entra en sala espera
    - En tratamiento → atendido por médico hasta:
      - fin consulta; "expulsado" para atender a otro; prescrita prueba médica
    - Espera resultado prueba médica
      - Al obtenerse vuelve a sala de espera

Aspectos generales de la planificación

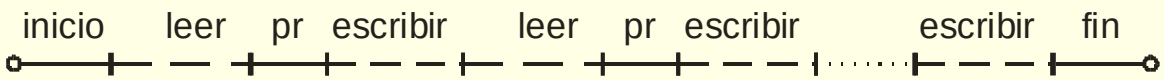
- Objetivos generales:
  - Optimizar uso
  - Minimizar tiempo de espera
  - Ofrecer reparto equitativo
  - Proporcionar grados de urgencia
- Tipos de planificación: expulsiva versus no expulsiva
- Afinidad a subconjunto de ejemplares de recurso
  - Estricta (*hard*): pedida por el usuario
  - Natural (*soft*): favorece rendimiento
- ¿Se puede quedar un recurso libre aunque alguien lo necesite?
  - Normalmente, no (planificadores *work-conserving*)
  - Algunos planificadores (*non-work-conserving*) pueden hacerlo:
    - Por tarificación, por rendimiento (SMT),…

Caracterización de los procesos

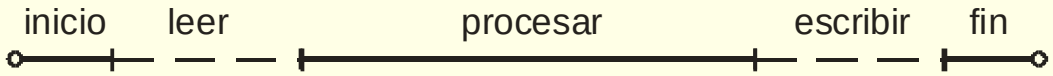
- Perfil de uso del procesador (*CPU bound* vs. *I/O bound*)
  - Mejor primero proceso con ráfaga de UCP más corta
- Grado de interactividad
  - Asegurar buen tiempo de respuesta de interactivos
- Nivel de urgencia (o importancia)
  - Especialmente trascendente en sistemas de tiempo real
- *Threads* (entidad planificable) no son independientes:
  - Incluidos en procesos
  - Que pueden ser parte de aplicaciones
  - Que pertenecen a usuarios
  - Que pueden estar organizados en grupos

Uso intensivo de la E/S versus la UCP

```
Programa

inicio();

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

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 dos usuarios U y V
  - U ejecuta aplicación con 1 proceso que incluye 1 *thread* (t1)
  - V ejecuta dos aplicaciones
    - Una con 2 procesos con un solo *thread* cada uno (t2 y t3)
    - Otra con 1 un único proceso con dos *threads* (t4 y t5)
  - Equitatividad para:
    - *Threads*: cada *thread* 20% de UCP (PTHREAD_SCOPE_SYSTEM)
    - Procesos: t1, t2 y t3 25%; t4 y t5 12,5% (PTHREAD_SCOPE_PROCESS)
    - Aplicaciones: t1 33,33%; otros *threads* 16,67%
    - 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***

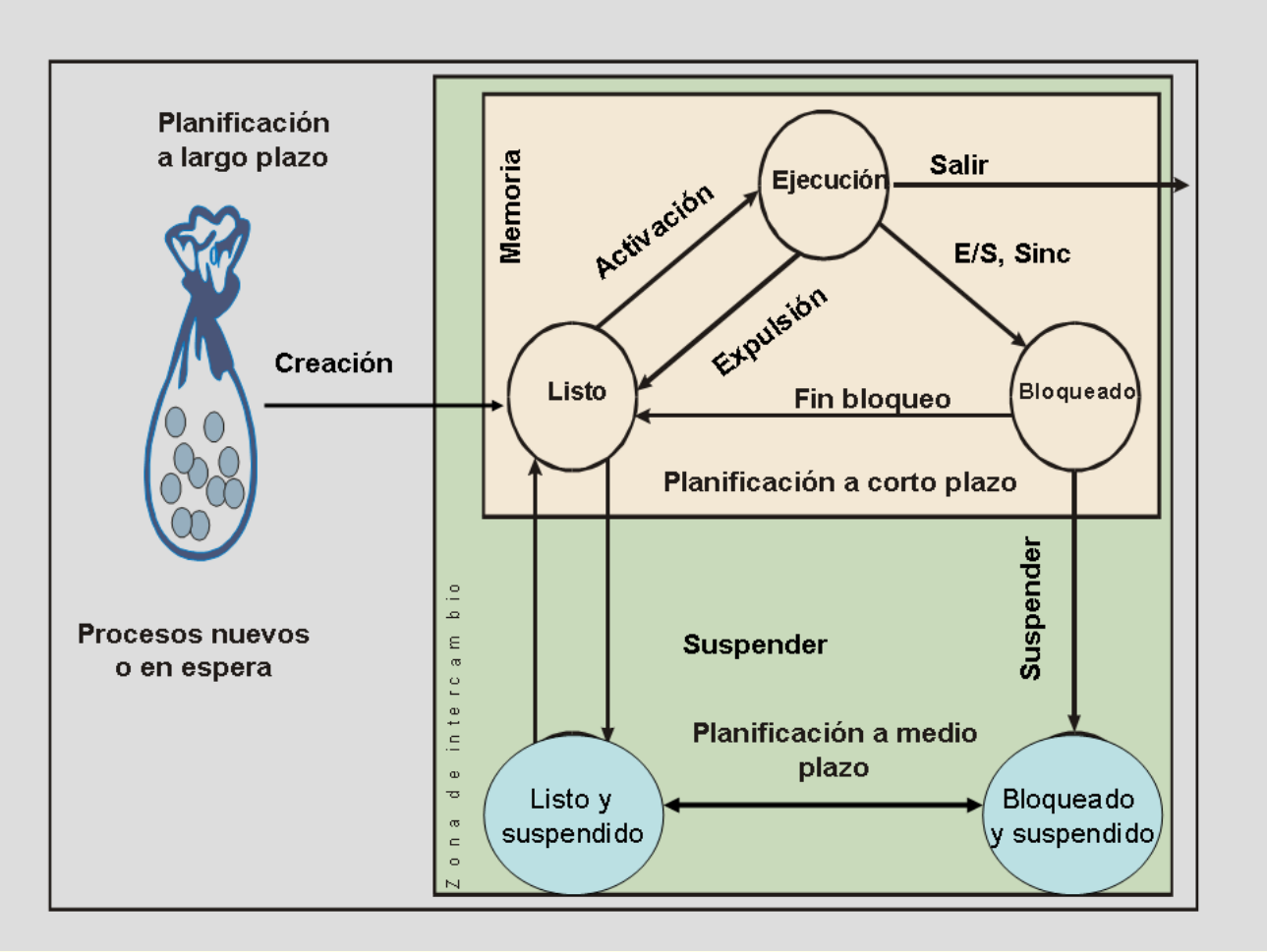
Objetivos de la planificación del procesador

- 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**
  - **Equitatividad**

Tipos de planificadores

- A largo plazo (planificador de trabajos)
  - Control de entrada de trabajos al sistema
  - Característico de sistemas por lotes
  - 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

Tipos de planificadores



Múltiples niveles de planificación

- Hasta 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
      - Corresponde a PTHREAD_SCOPE_PROCESS
    - 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 (*vCPU*)
    - 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 :
  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
- Dos tipos de algoritmos:
  - no expulsivos: sólo C.C. voluntarios (1 y 2)
  - expulsivos: además C.C. involuntarios (3, 4, 5 y/o 6)
- No confundir con núcleo expulsivo o no expulsivo

Algoritmos de planificación

- Primero en llegar primero en ejecutar (FCFS)
- Primero el trabajo más corto (SJF/SRTF)
- Planificación basada en prioridades
- *Round robin* (turno rotatorio)
- Lotería
- Colas multinivel

- Algoritmo real es mezcla de teóricos + ajustes empíricos

Primero en llegar primero en ejecutar (FCFS)

- Selección: Proceso que lleva más tiempo en cola de listos
- Algoritmo no expulsivo
- 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
  - ↓ Proporcionar grados de urgencia

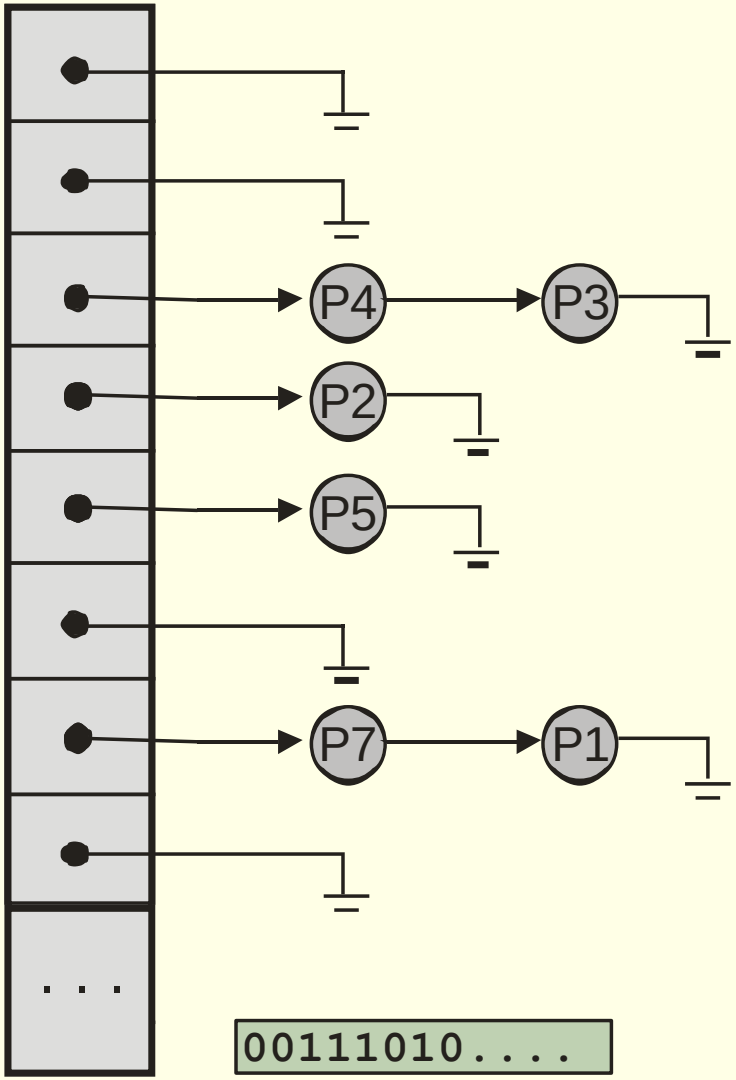
Primero el trabajo más corto (SJF)

- Selección: Proceso listo con próxima ráfaga UCP más corta
- Versión no expulsiva: Sólo si proceso se bloquea o termina
- Versión expulsiva: 1º el de menor tiempo restante (SRTF)
  - También se activa si proceso pasa a listo (puntos 3 y 4)
- ¿Cómo se conoce a priori la duración de la próxima ráfaga?
  - Estimaciones a partir de las anteriores (p.e. media exponencial)
    - $t.estimadoRodaja_{N+1} = \alpha\ t.realRodaja_N + (1 - \alpha)\ t.estimadoRodaja_N$
- Puede producir inanición
- 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
- Punto fuerte: Minimizar tiempo de espera

Planificación por prioridad

- Cada proceso tiene asignada una prioridad
- Selección: Proceso en cola de listos que tenga mayor prioridad
- Existe versión no expulsiva y expulsiva
  - Si proceso pasa a listo o actual baja su prioridad (3, 4 y 5)
- Las prioridades pueden ser estáticas o dinámicas
- Prioridad puede venir dada por factores externos o internos
- Puede producir inanición
  - "Envejecimiento": Prioridad aumenta con el tiempo
- En general, mal esquema para la equitatividad
- Implementación:
  - Nº de prioridades fijo y reducido: O(1)
  - Si no: igual que SJF
    - O(N) si usa lista de procesos; O(log N): árbol binario
- Punto fuerte: Proporcionar grados de urgencia

Planificación por prioridad: implementación O(1)



Turno rotatorio (*Round Robin*, RR)

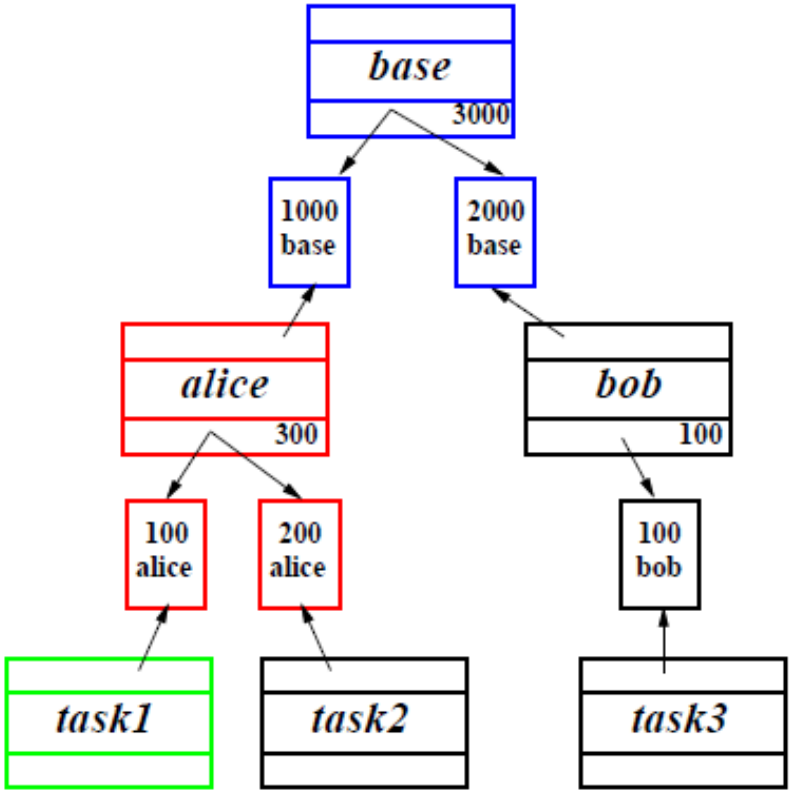
- FCFS + plazo (rodaja o cuanto)
- Algoritmo expulsivo pero sólo con fin del cuanto (punto 6)
- Tiempo de respuesta acotado (nº procesos listos * tam. rodaja)
- Tamaño rodaja: grande (→FIFO) vs. pequeño (sobrecarga)
  - Más pequeño → menor t. respuesta y mejor equitatividad
  - Configurable (Windows cliente 20 ms. servidor 120 ms.)
- Tamaño rodaja de proceso varía durante vida del proceso
  - Windows triplica rodaja de proceso al pasar a primer plano
- Más general: Round-robin con pesos (WRR)
  - Tamaño rodaja proporcional a peso del proceso
  - Implementar importancia de procesos en vez de con prioridad
- Round-robin por niveles para *Fair-share scheduling*
  - Reparto rodaja entre usuarios; rodaja usuario entre sus procesos.
- Punto fuerte: Ofrecer reparto equitativo

Planificación por lotería

- Cada proceso tiene uno o más billetes de lotería
- Selección: proceso que posee billete aleatoriamente elegido
- Versión no expulsiva o expulsiva (rodaja de tiempo entre sorteos)
- No inanición
- Importancia de los procesos: nº de billetes que poseen
- 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* por niveles: *ticket currencies*
  - P.e. Usuario 1 y Usuario 1 tienen 100 billetes cada uno
    - U1 procesos P y Q 500 billetes cada; U2 proceso R 10 billetes
    - Valor global: P 50; Q 50; R 100
- Punto fuerte: Ofrecer reparto equitativo

http://www.waldspurger.org/carl/papers/lottery-osdi94-slides.pdf

Currency Implementation

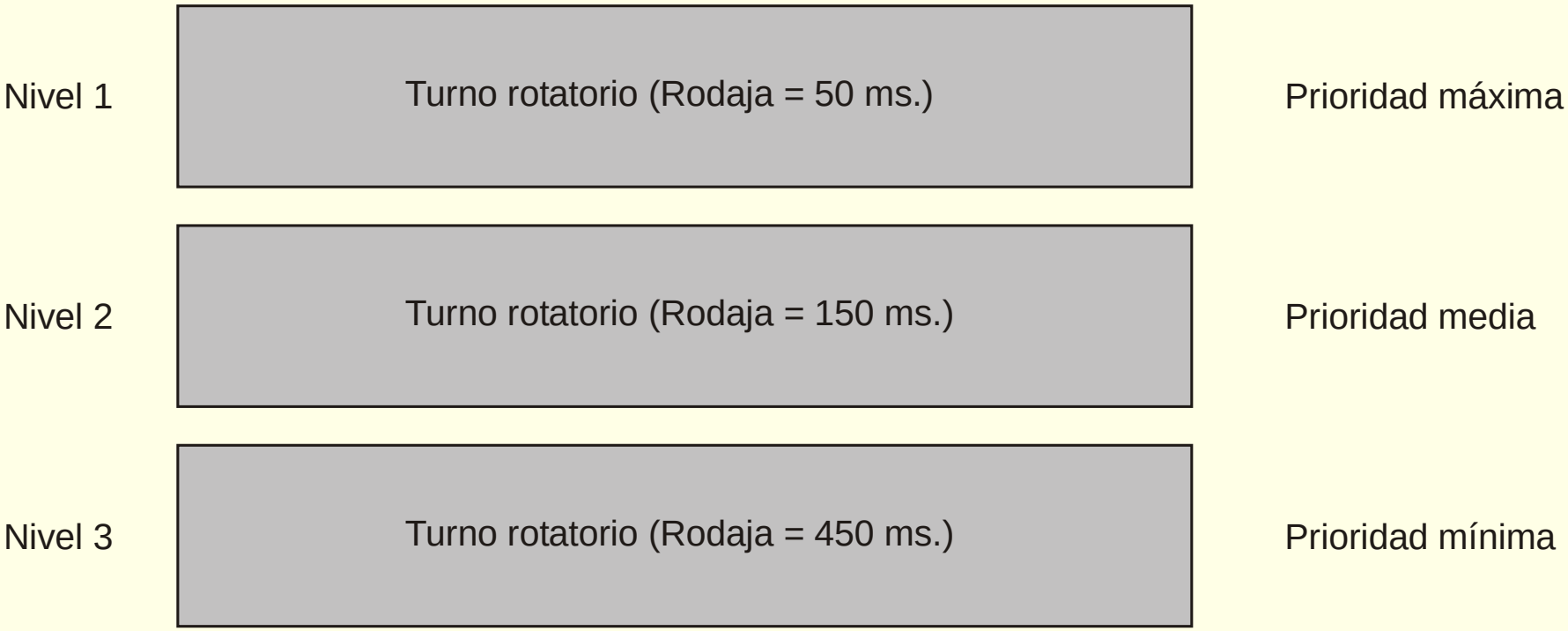


- **Computing Values**
  - currency: sum value of backing tickets
  - ticket: compute share of currency value
- **Example**
  - task1 funding in base units?
  - $\frac{100}{300} \times 1000$
  - 333 base units

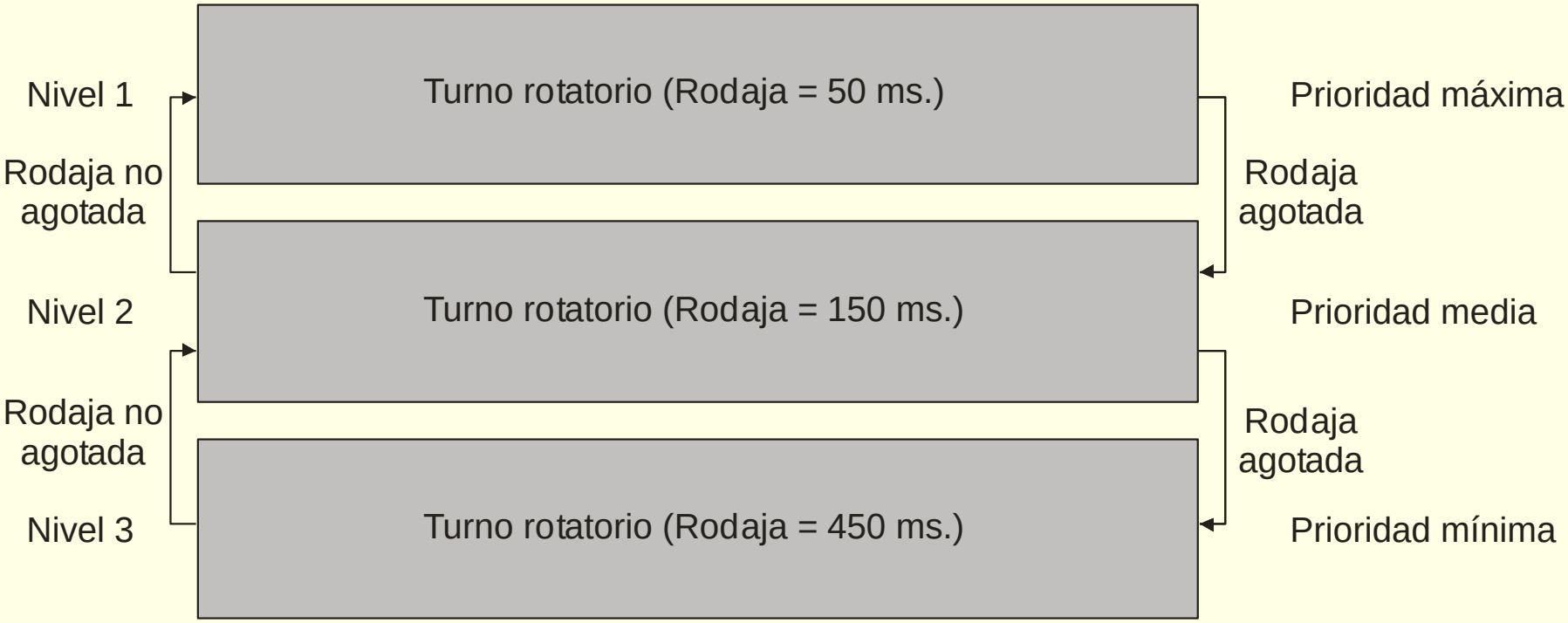
Colas multinivel

- Generalización: Distinguir entre clases de procesos
- Parámetros del modelo:
  - Número de niveles (clases)
  - Algoritmo de planificación de cada nivel
  - Algoritmo de reparto del procesador entre niveles
- Colas con o sin realimentación:
  - Sin: proceso en la misma cola durante toda su vida
  - Con: proceso puede cambiar de nivel

Colas multinivel sin realimentación



Colas multinivel con realimentación



Planificación uniprocesador en Linux

- Planificador quizás no sea la pieza más brillante de Linux
  - O(N) hasta versión 2.6
  - Demasiados parámetros configurables
  - No exento de polémica (Kolivas)
- Planificador: módulo relativamente cambiante
  - Es una función de selección
  - No ofrece API a las aplicaciones
- 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
- Linux da soporte extensiones t. real POSIX
  - Cola multinivel sin realimentación: t. real (no crítico) y normal
    - 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 (ya estudiado en SEU)
    - 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
    - Novedad de últimas versiones: SCHED_DEADLINE
      - Implementación de EDF en Linux
      - *https://en.wikipedia.org/wiki/SCHED_DEADLINE*
  - Clase normal (SCHED_NORMAL, *aka* SCHED_OTHER)
    - El planificador por defecto de Linux (O(1); actualmente CFS).
  - SCHED_BATCH: similar a SCHED_NORMAL pero:
    - Asume que proceso es *CPU-Bound* y no expulsa a otros procesos
  - SCHED_IDLE: procesos que no tienen ningún grado de urgencia

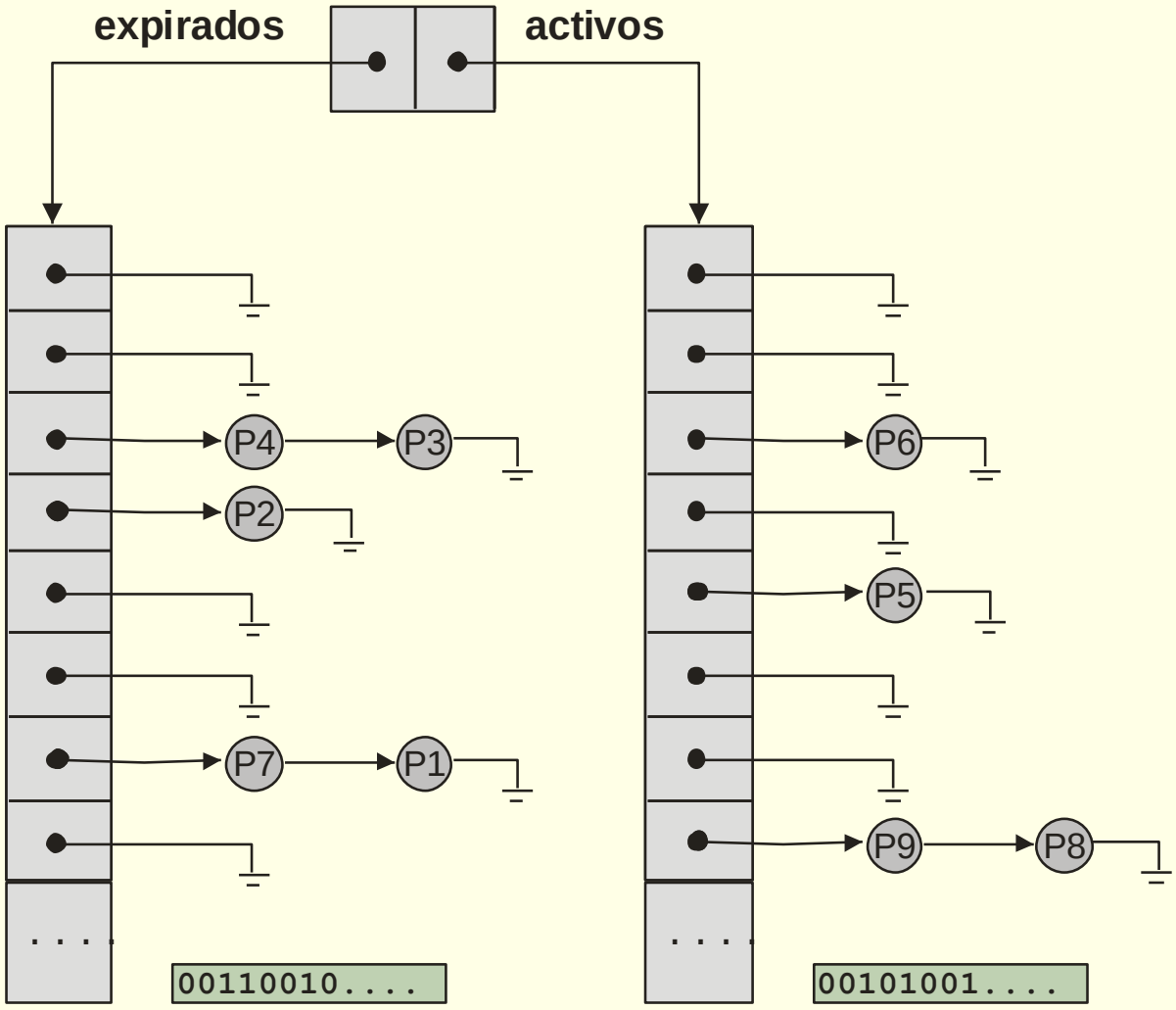
Planificador de Linux O(1)

- Mezcla de prioridades y round-robin
- 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
      - Sino procesos con rodaja 800ms tendrían mal tiempo de respuesta
    - "Fin de ronda": intercambio de listas
- *http://lxr.linux.no/linux-bk+v2.6.9/kernel/sched.c*

Planificador de Linux O(1)



Limitaciones de O(1)

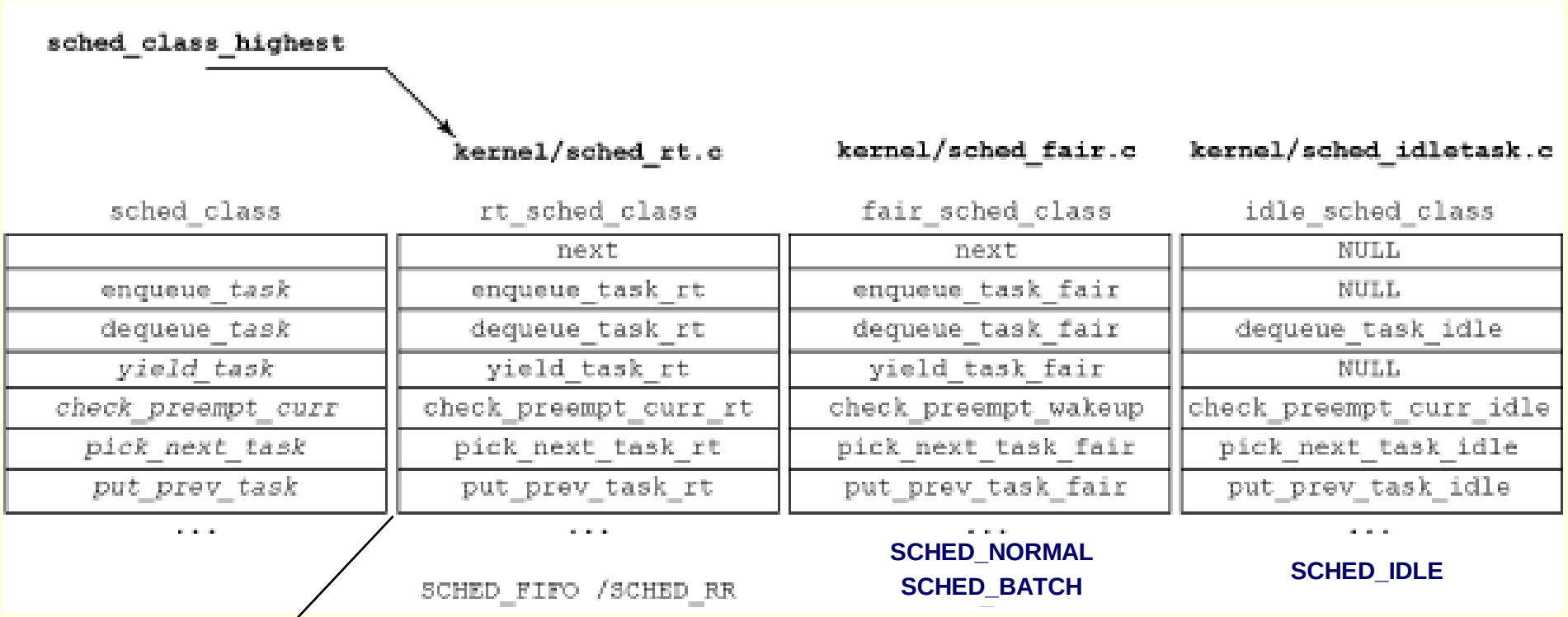
- Lleno de parámetros difícilmente ajustables
  - *Bonus*, cuándo considerar "interactivo" un proceso,…
- Gestiona simultáneamente todas las clases de procesos
  - Usa 140 listas de procesos: 100 de tiempo real y 40 normales
  - Sería mejor un módulo separado por cada clase
  - O incluso mejor entorno genérico donde añadir nuevas clases
- Presenta ciertas patologías:
  - P.e. Sólo 2 procesos prio. mínima → RR 5 ms
    - Sin embargo, sólo 2 de prio. máxima → RR 800 ms
  - Tamaño de rodaja debería depender también de la carga
    - En el ejemplo sería mejor que rodaja fuera igual en ambos casos
  - Prioridades deberían considerarse de forma relativa
- Y sobretodo: No proporciona una solución *fair-share*
  - Reparto de procesador sólo entre *threads*

Completely Fair Scheduler (CFS)

- Incorporación a Linux de un *fair-share scheduler*
- Además, se ha creado entorno genérico de clases de planificación
- Meta: todos los procesos usan "mismo tiempo" de UCP (*vruntime*)
  - Implementación precisa irrealizable en la práctica
    - Continuamente habría que reasignar el procesador
  - Se comprueba "injusticia" cada cierto tiempo
- Tiempo ponderado por factor que depende de la prioridad
  - Cada prio. (*nice*) tiene un peso: -20→88761, 0→1024, 19→15
    - http://lxr.free-electrons.com/source/kernel/sched/sched.h?v=4.4#L1128
    - Peso ≈ $1024/1{,}25^{nice}$
  - + prioridad → + "lento" corre el reloj; con prio 0 reloj real
- Selección: proceso i con menor *vruntime* (tratado más injustamente)
  - Se le asigna un t. ejecución que depende de su prio. y la carga
    - Evita patología de O(1) tratando prioridades de forma relativa
  - Tiempo de ejecución asignado ≈ rodaja

Clases de planificación *(Scheduling classes)*

http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/



Actualmente también
stop_sched_class
dl_sched_class

http://lxr.free-electrons.com/source/kernel/sched/sched.h#L1238

http://lxr.free-electrons.com/source/kernel/sched/core.c#L3029

CFS: asignación de tiempo ejecución

- Proporción entre su peso y el total de los listos
- "Rodaja" = ($peso_i$ / peso total listos) * periodo
- http://lxr.free-electrons.com/source/kernel/sched/fair.c?v=4.4#L620
- En periodo ejecutan todos proc listos→%UCP proporción a peso
  - Normalmente periodo = sched_latency
  - Influye en t. de respuesta: configurable; 6ms. por defecto
    - http://lxr.free-electrons.com/source/kernel/sched/fair.c?v=4.4#L50
- Si nº listos (NL) alto → rodaja demasiado pequeña (sobrecarga)
  - sched_min_granularity mínima rodaja permitida
    - configurable; 0,75ms. por defecto
    - sched_nr_latency = sched_latency / sched_min_granularity
      - Cuántos procesos "caben" en sched_latency (defecto 8)
  - Si NL > sched_nr_latency → periodo = sched_min_granularity * NL
    - Hay que ampliar el periodo → peor tiempo de respuesta

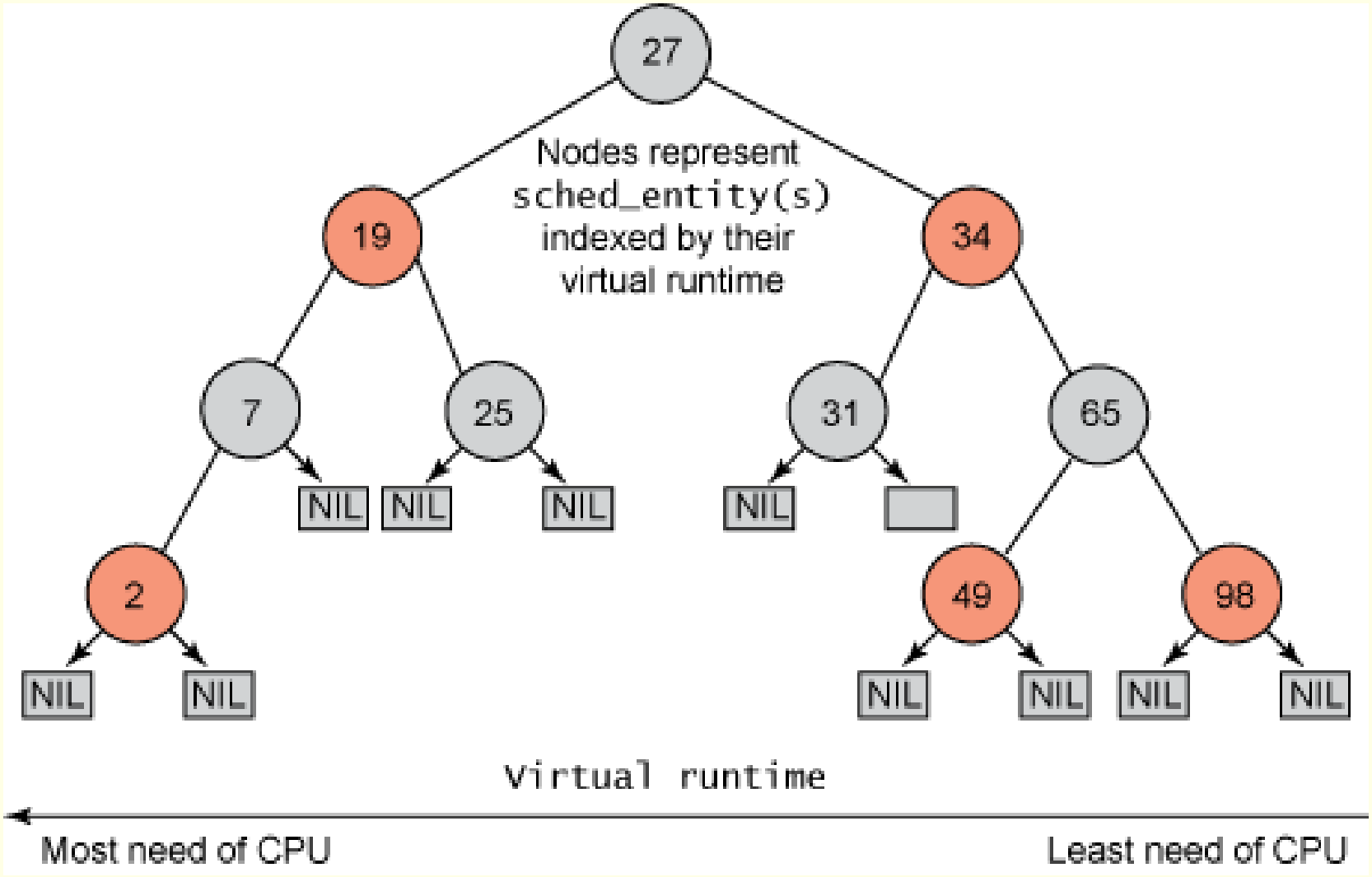
CFS vs O(1): ejemplo prioridades y % uso de UCP

- Pesos elegidos para que prio. se traten como valores relativos
- CFS
  - Ej1. 3 procesos prio/peso: P1 0/1024, P2 1/820 y P3 2/655
    - P1 41% (2,46 ms.); P2 33% (1,97 ms.); P3 26% (1,57 ms.)
  - Ej2. 3 procesos prio/peso: P4 17/23, P5 18/18 y P6 19/15
    - P4 41% (2,46 ms.); P5 32% (1,93 ms.); P6 27% (1,61 ms.)
- O(1)
  - Ej1. 3 procesos prio/rodaja: P1 0/100ms, P2 1/95 y P3 2/90
    - P1 35% (100 ms.); P2 33% (95 ms.); P3 32% (90 ms.)
  - Ej2. 3 procesos prio/rodaja: P4 17/15ms, P5 18/10 y P6 19/5
    - P4 50% (15 ms.); P5 33% (10 ms.); P6 17% (5 ms.)

CFS: dinámica del algoritmo

- Cuando proceso i se bloquea o termina rodaja actualiza su vruntime
  - Añadiendo tiempo ejecutado (T) normalizado con peso prio 0
    - $vruntime_i$ += (T / $peso_i$) * peso_prio_0
    - En Ej1 si agotan rodaja todos suman 2,46 ms. a su vruntime
    - En Ej2 si agotan rodaja todos suman 109,71 ms. a su vruntime
    - Si se bloquean después de ejecutar 1 ms. cada proceso suma
      - P1 1; P2 1,24; P3 1,56; P4 44,52; P5 56,89; P6 68,27
      - Proceso con prioridad máxima (-20; peso 88761) sumaría 0,01 ms.
- 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 que proceso con largo bloqueo acapare UCP
    - Desbloqueo: vruntime = max(vruntime, min_vruntime - sched_latency)
- Implementación O(log N): requiere árbol binario *red-black tree*

CFS: *Red-black Tree (RBT)*

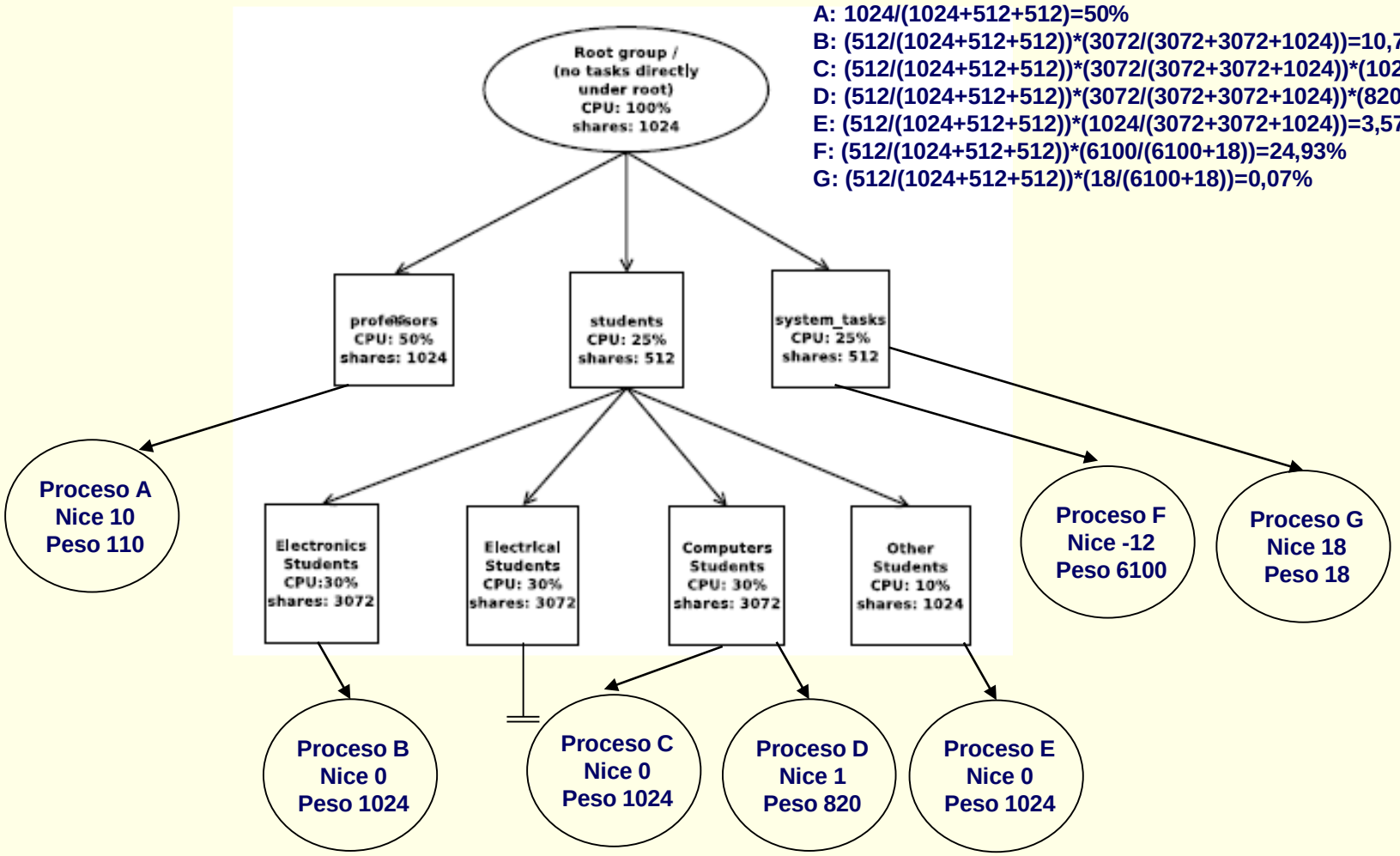


http://www.ibm.com/developerworks/library/l-completely-fair-scheduler

CFS: planificación de grupos

- Por defecto CFS provee *fair-share scheduling* sólo entre procesos
- Planificación de grupos: basado en mecanismo *cgroups*
  - Permite crear grupos de procesos formando una jerarquía
    - Hijos de un grupo pueden ser procesos u otros grupos
  - Y repartir recursos (UCP, memoria,…) entre esos grupos
  - Se usa, por ejemplo, para crear contenedor (segunda parte)
- Proporciona mecanismo flexible para el administrador:
  - Reparto por usuarios, por tipos de usuar., por apps/servicios,...
- Con respecto a UCP, cada grupo tiene definido un *share*:
  - Determina % uso UCP con respecto a sus grupos "hermanos"
    - Relativo al porcentaje de uso de UCP del padre
  - Procesos de 1 grupo se reparten tiempo del mismo según pesos

Ejemplo de jerarquía con *shares*



Procesos y %UCP

A: 1024/(1024+512+512)=50%
B: (512/(1024+512+512))*(3072/(3072+3072+1024))=10,71%
C: (512/(1024+512+512))*(3072/(3072+3072+1024))*(1024/(1024+820))=5,95%
D: (512/(1024+512+512))*(3072/(3072+3072+1024))*(820/(1024+820))=4,76%
E: (512/(1024+512+512))*(1024/(3072+3072+1024))=3,57%
F: (512/(1024+512+512))*(6100/(6100+18))=24,93%
G: (512/(1024+512+512))*(18/(6100+18))=0,07%

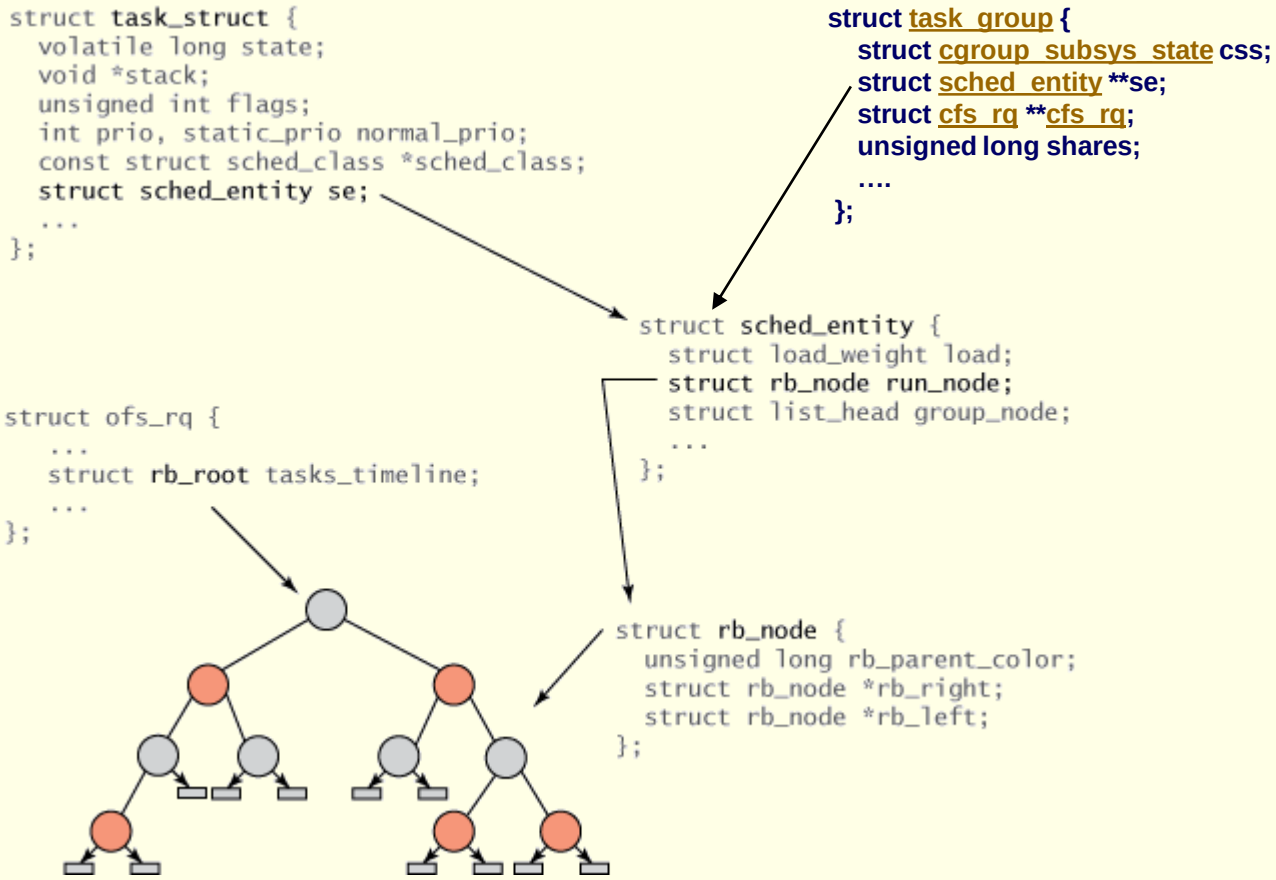
"CPU bandwidth control for CFS". Paul Turner, Bharata B Rao, Nikhil Rao

CFS: algoritmo de planificación de grupos

- Planificador gestiona entidades planificables: procesos o grupos
- Cada grupo tiene su propia cola de entidades planificables:
  - Su propio RBT
  - Hijos del grupo (procesos o subgrupos) incluidos en su RBT
- 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: planificación de grupos

http://lxr.free-electrons.com/source/kernel/sched/sched.h?v=4.4#L239



http://www.ibm.com/developerworks/library/l-completely-fair-scheduler

Configuración de grupos: *cgroups*

https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

```
$ mount -t tmpfs cgroup_root /sys/fs/cgroup
$ mkdir /sys/fs/cgroup/cpu
$ mount -t cgroup -ocpu none /sys/fs/cgroup/cpu
$ cd /sys/fs/cgroup/cpu
$ mkdir multimedia
# create "multimedia" group of tasks
$ mkdir browser
# create "browser" group of tasks
#Configure the multimedia group to receive twice the CPU bandwidth that of browser group
$ echo 2048 > multimedia/cpu.shares
$ echo 1024 > browser/cpu.shares
$ firefox & # Launch firefox and move it to "browser" group
$ echo $!  > browser/tasks  # <firefox_pid>
$ gmplayer & # Launch gmplayer and move it to “multimedia" group
$ echo $! > multimedia/tasks # <movie_player_pid>
```