

Sistemas Operativos

Planificación

¿Planificar por qué?



Contenido



- Conceptos básicos
- Algoritmos de planificación
- Ejemplo: linux
- Planificación en multiprocesadores SMP

Planificación de procesos/hilos

- Objetivos:
 - Optimizar uso de las CPUs
 - Minimizar tiempo de espera
 - Ofrecer reparto equitativo (justicia)
 - Proporcionar grados de urgencia (prioridades)
- Tipos de planificación
 - No expropiativa: el proceso conserva la CPU hasta que se bloquea, la cede expresamente o termina su ejecución.
 - Expropiativa: el SO expulsa al proceso de la CPU
 - Exige un reloj que interrumpe periódicamente
- Colas de procesos/hilos (run queues)

¿Cómo saber si estamos planificando bien?

 En qué nos tendríamos que fijar para hacer una planificación

Métricas de un planificador

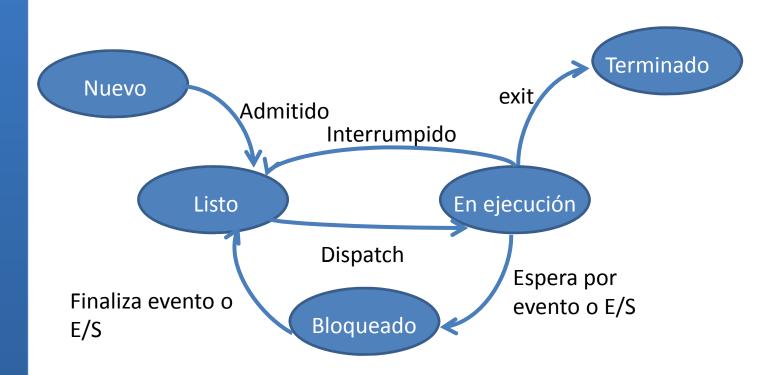
- Parámetros por entidad (proceso o thread)
 - Tiempo de ejecución: creación terminación
 - Tiempo de espera: tiempo total listo y sin CPU
 - Tiempo de respuesta: creación 1^{er} uso de CPU
- Parámetros globales
 - Porcentaje de utilización del procesador
 - Productividad: número de trabajos completados por unidad de tiempo

Tipos de planificadores

- Planificador: selecciona el proceso
 - A largo plazo: añadir procesos a ejecutar (batch)
 - A medio plazo: añadir procesos a RAM
 - A corto plazo: qué proceso tiene la CPU
 - Planificación de E/S
- ¿Tres tipos de planificadores?

Tipos de planificadores

 Activador o Dispatcher: da control al proceso (cambio de contexto)



Estados del proceso

Para implementar el modelo de procesos, el sistema operativo mantiene una tabla de procesos, con sólo una entrada por cada proceso

Estados del proceso

- Qué pasa si un proceso pasa de en Ejecución a Bloqueado
- Cómo somos capaces de volver al punto exacto dónde estaba ese proceso

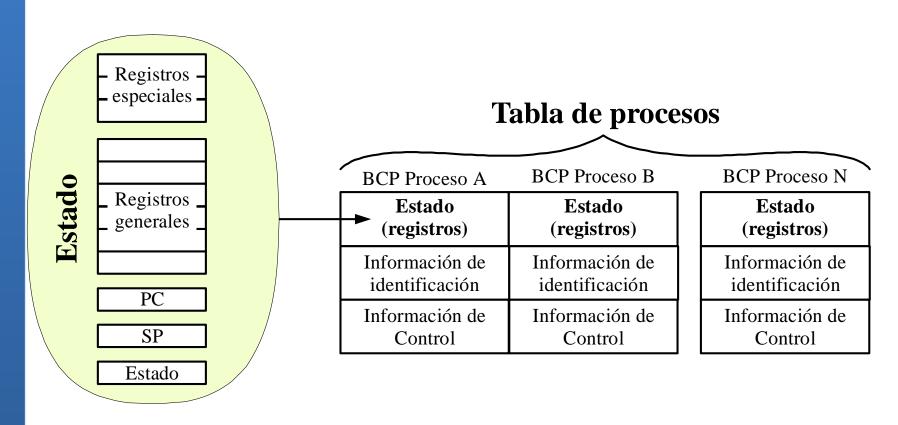
Cambio de contexto

- El cambio de contexto es el conjunto de acciones que realiza el SO para cambiar el proceso que está actualmente en ejecución en una CPU
- Acciones (simplificación):
 - 1. Salvar el contexto del proceso saliente (registros del modelo de programación) en su BCP
 - 2. Cambiar el estado del proceso saliente (En ejecución -> Otro Estado)
 - 3. Intercambio de los espacios de direcciones
 - Segmentos o regiones de memoria que puede usar un proceso
 - En x86. GDT (Global descriptor Table)
 - En algunas arquitecturas → Invalidación de entradas de la TLB
 - 4. Cambiar el estado del proceso entrante, (Listo -> En ejecución) Restaurar su contexto (BCP -> registros) y volver a modo usuario
- Puede llegar a ser una operación bastante costosa
- El cambio de modo de ejecución del procesador no siempre desencadena un cambio de contexto

Cambio de contexto

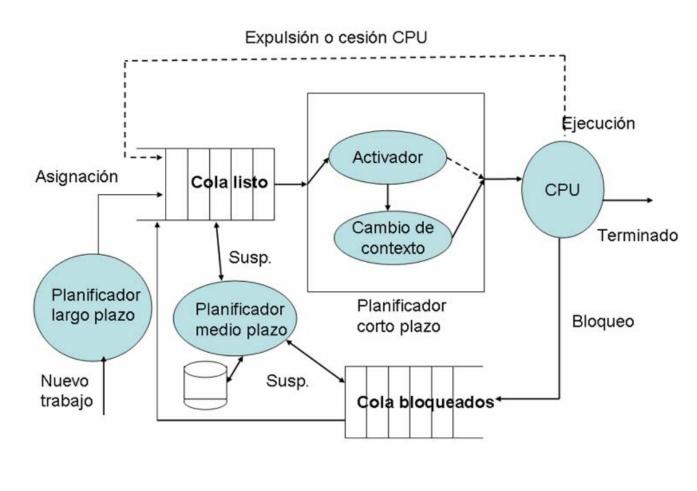
Se salva el estado:





Estructuras del planificador





13

Puntos de activación del planificador

- Periódicamente (interrupción del temporizador de la CPU)
- Como resultado del procesamiento de alguna interrupción generada por otros dispositivos de E/S
- El proceso en ejecución causa una excepción que lo bloquea (fallo de página) o fuerza su terminación (violación de segmento)
- Cuando el proceso en ejecución termina
- El proceso realiza una llamada bloqueante
- Cesión voluntaria del procesador
 - sched_yield()
- Se desbloquea un proceso más "importante" que el actual
 - Cambios de prioridad en el sistema de procesos

Contenido



- Conceptos básicos
- Algoritmos de planificación
- Ejemplo: linux
- Planificación en multiprocesadores SMP

Algoritmos de planificación



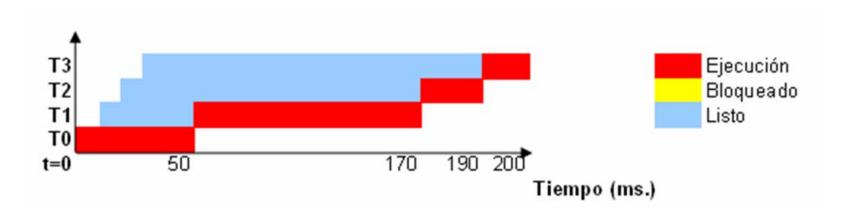
Algoritmos no expropiativos

- El planificador no quita la CPU al proceso una vez que está en ejecución, a no ser esta la ceda voluntariamente, termine o se bloquee por E/S
- Algoritmos
 - Primero en llegar primero en ejecutar FCFS (First-Come First-Served)
 - Primero el trabajo más corto SJF (Shortest Job First)
 - Planificación basada en prioridades

Primero en llegar primero en ejecutar FCFS (1)

Muy sencillo y óptimo en uso de CPU

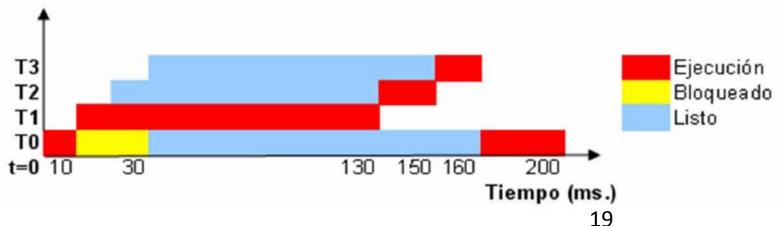
Proceso o thread	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10



Primero en llegar primero en ejecutar FCFS (II)

- Programas con E/S son encolados al final
- Programas largos afectan al sistema

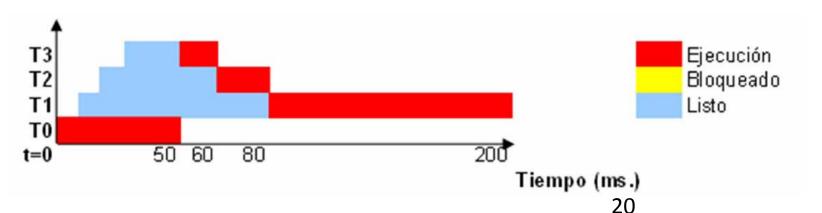
Proceso o thread	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
тз	30	10



Primero el trabajo más corto SJF

- Bueno para programas interactivos
- Necesita conocer el perfil de las tareas
- Problemas de inanición

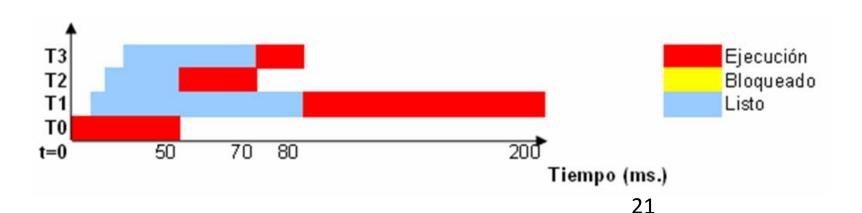
Proceso o thread	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10



Prioridades

- Bueno para sistemas con grados de urgencia
- Problema de inanición:
 - Aumento de la prioridad con la edad

Proceso o thread	Instante de llegada	Tiempo de procesador (ms)	Prioridad
T0	0	50	4
T1	10	120	3
T2	20	20	1
T3	30	10	2



Algoritmos expropiativos

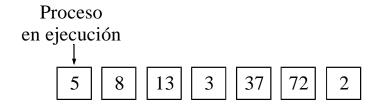
- No expropiativos no son adecuados para SSOO de propósito general
 - Mezcla de trabajos interactivos y trabajos intensivos en CPU
- Algoritmos expropiativos
 - Round Robin (turno rotatorio)
 - Primero el de menor tiempo restante (SRTF)
 - Shortest Remaining Time First
 - Prioridad expulsiva
 - Colas multinivel

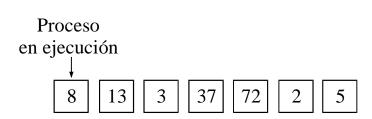
Algoritmos expropiativos

- La planificación se realiza dividiendo el tiempo de CPU en rodajas llamadas quanto o timeslice
- El planificador se activa periódicamente
 - El temporizador de la CPU se configura para generar interrupciones cada tick (~ms)
 - El timeslice se expresa en ticks (Dependiente de SO)
 - Cada tick, el planificador realiza CPU accounting
 - Ej: incrementar el contador de ticks que el hilo ha usado desde que entró a la CPU
 - Cuando el planificador lo considera pertinente, cambia el proceso/hilo que hay ejecutando por otro

Round Robin (I)

- FCFS + timeslice
- Asignación de procesador rotatoria
- Uso en sistemas de tiempo compartido
 - Equitativo (mejor hacerlo por uid y no por proceso)
- Se asigna un tiempo máximo de procesador que el proceso puede consumir sin ser expropiado (timeslice)

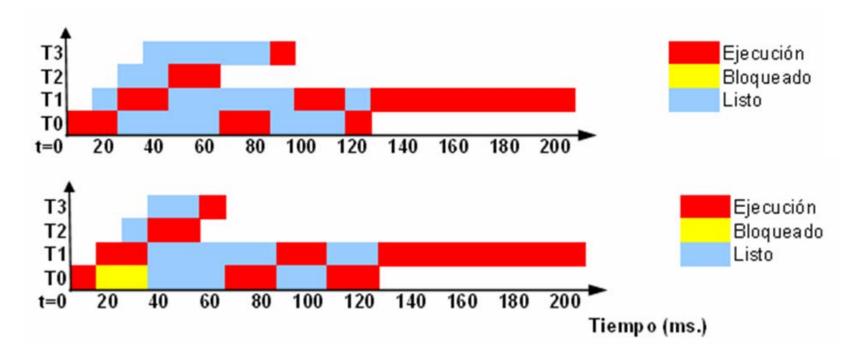






Round Robin (II)

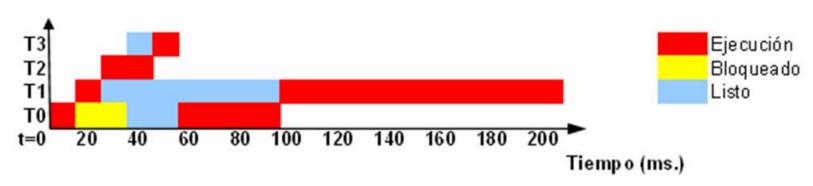
Proceso o thread	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10



Primero el de menor tiempo restante SRTE

- SJF + timeslice variable
 - Bueno para programas interactivos
 - Necesita conocer el perfil de las tareas
 - Problemas de inanición

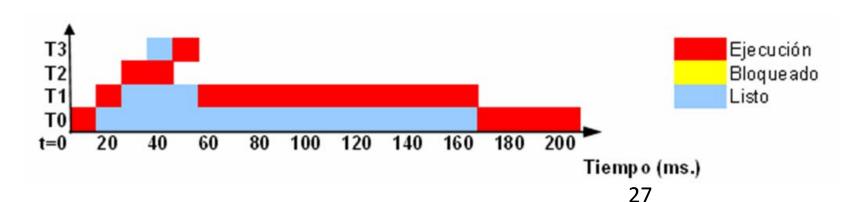
Proceso o thread	Instante de llegada	Tiempo de procesador (ms)
T0	0	50
T1	10	120
T2	20	20
T3	30	10



Expriopiativo basado en prioridades

- Bueno para sistemas con grados de urgencia
- Problema de inanición:
 - Aumento de la prioridad con la edad

Proceso o thread	Instante de llegada	Tiempo de procesador (ms)	Prioridad
T0	0	50	4
T1	10	120	3
T2	20	20	1
T3	30	10	2

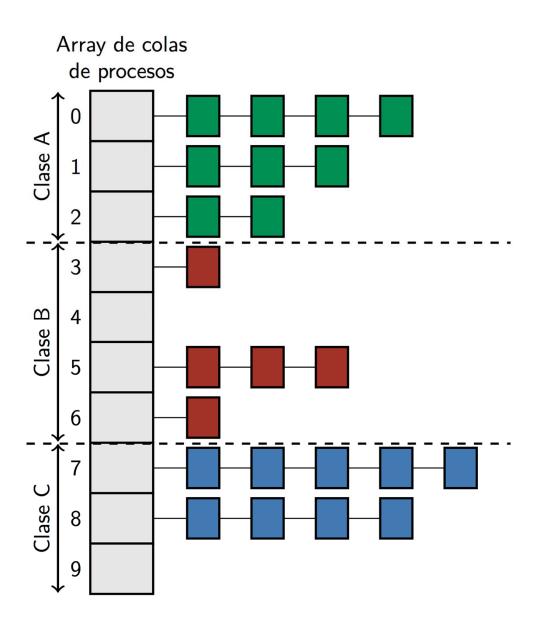


Planificación con colas multinivel (1)

- Objetivo: dar soporte a distintas clases de procesos
- En el sistema existen k niveles de prioridad, cada uno con una cola de procesos asociada
 - El SO gestiona realmente un array de colas de procesos
 - En cada nivel de prioridad puede haber un timeslice diferente
- Los niveles de prioridad se agrupan en rangos para dar servicio a distintos tipos de procesos/hilos
 - Tiempo real
 - Hilos de sistema
 - Interactivos
 - Batch

— ...

Planificación con colas multinivel (II)



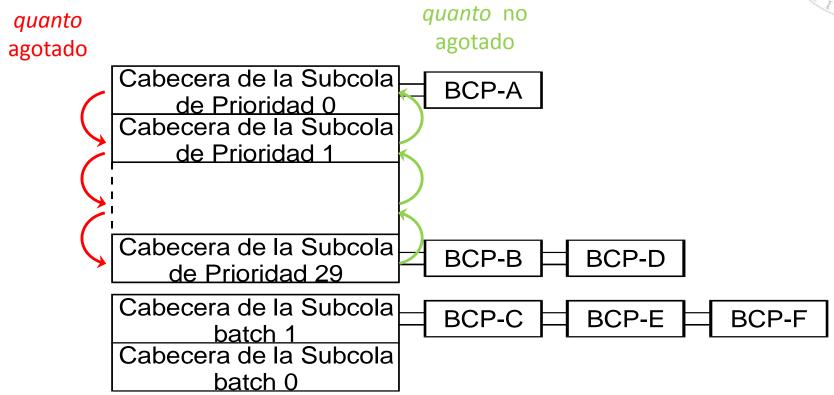


Planificación con colas multinivel (III)

- Las colas dentro de un rango de prioridades pueden gestionarse de dos formas:
 - Sin realimentación (procesos con prioridad fija)
 - Proceso en la misma cola durante toda su vida
 - Con realimentación (procesos con prioridad dinámica)
 - Los procesos pueden cambiar de nivel
 - El cambio de nivel sólo se produce dentro del rango de prioridades gestionado por el "planificador local"
 - Necesario definir *política de cambio de nivel*
 - Ejemplo: Política para favorecer a procesos interactivos
 - Si proceso agota su timeslice, baja de nivel
 - Si proceso no agota su timeslice (p. ej., bloqueo E/S), sube de nivel

Colas de procesos





Contenido



- Conceptos básicos
- Algoritmos de planificación
- Ejemplo: linux
- Planificación en multiprocesadores SMP

Ejemplo: Linux

- El kernel de gnu-Linux utiliza un planificador para decidir cuál va a ser el siguiente proceso a ejecutar
- Para esto utiliza un sistema de prioridades
 - Procesos con una prioridad más alta se van a ejecutar mas a menudo
 - Procesos con una baja prioridad se ejecutarán menos frecuentemente
 - En gnu-Linux los procesos no pueden superar el intervalo de ejecución asignado (time slice)
 - Lo procesos se reanudan y se suspenden si su coperación (preemptively multitasked)

Ejemplo: Linux

- En un sistema multitarea como Linux, es probable que haya varios programas que compitan por el mismo recurso:
 - Los programas que llevan a cabo trabajos en forma de ráfaga y se detienen brevemente para la entrada se consideran comportan mejor que los que acaparan el procesador calculando continuamente algún valor o continuamente consultando la sistema para ver si la nueva entrada está disponible.
 - "Nice": El sistema operativo determina la prioridad de un proceso basado en un valor "nice", que por defecto es 0, y en el comportamiento del programa.
 - Los programas que se ejecutan durante largos períodos sin detenerse en general, obtendrán un valor de nice bajo.
 - Los programas que se detienen, por ejemplo, a la espera para la entrada, se ven recompensados su valor de amabilidad "nice". Esto ayuda a mantener un programa que interactúa con el usuario responda; mientras se está a la espera de alguna entrada del usuario, el sistema aumenta su prioridad, de manera que cuando esté listo para continuar, tiene una alta prioridad.

Planificador Linux (v3.14)

- 140 niveles de prioridad 100 para procesos real-time (Deadline, RR y FIFO)
 - 40 para procesos normales (CFS)
- Objetivos CFS (Completely Fair Scheduler)
 - Aproximar planificación completamente justa
 - No se asigna un timeslice por prioridad sino un porcentaje del procesador
 - Proporcionar buenos tiempos de respuesta
 - Entornos interactivos (GUIs)

Idea planificador CFS

- Repartir el tiempo de CPU entre hilos de forma similar a lo que ocurriría si pudieran ejecutarse simultáneamente
- Si hay 4 hilos en el sistema ejecutándose durante 10ms de tiempo, lo justo sería que cada uno se hubiese ejecutado 2.5ms en ese período
 - ¿Y si los hilos tienen diferentes pesos?

Ejemplo planificación CFS



Tarea	Peso
Α	4
В	4
С	2

 Tarea A debe ejecutarse un 40% del tiempo total (8ms)

$$T_{CPU}(P) = sched_period \cdot \frac{peso(P)}{\sum_{i=1}^{n} peso(i)}$$
 \rightarrow $T_A = 20ms * (4/10) = 8ms$



Parámetros CFS

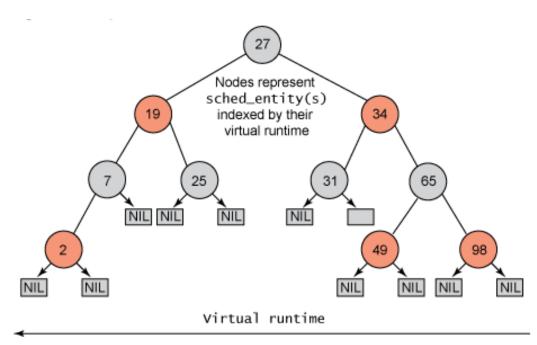
- sysctl_sched_latency (sched_period)
 - Período durante el cual cada tarea activa se debería planificar al menos una vez
 - $timeslice_{proc. n} \sim sched period *peso de proc. n / \Sigma$ (pesos proc. activos)
- Peso de una tarea proporcional a su prioridad (nice)
 - Existe una tabla de conversión prioridad <-> peso
 - Una diferencia de 1 en dos valores de nice supone un cambio de 10% en el reparto de CPU
- sysctl_sched_min_granularity
 - Tiempo mínimo que podrá ejecutar cualquier tarea antes de ser expropiado

CFS: siguiente tarea para ejecutar

- El hilo actual puede abandonar la CPU por...
 - Haber agotado su slice
 - Abandonar voluntariamente (E/S, yield...)
 - Otro hilo se merece más la CPU
- Siguiente tarea: la de menor virtual runtime
 - No hay colas de prioridades
 - El tiempo de CPU virtual transcurre más rápidamente para procesos de menor prioridad y más lentamente para los de mayor prioridad
 - Justicia: que todos los procesos reciban el mismo vruntime

CFS

- Cola de ejecución-> Red-black tree
 - Se ejecuta el proceso de más a la izquierda
 - Árbol balanceado: operaciones O(log N)



Interacción con el planificador

- nice permite ejecutar un comando con una prioridad entre -19 y 20
 - Recuerda: 40 niveles de prioridad
 - man nice (o, mejor, info nice)
- renice permite alterar la prioridad de un proceso en ejecución
- sysctl permite consultar/modficar parámetros del kernel
 - sysctl –A | grep sched

Comprobación CFS

 Creamos dos procesos CPU bound y forzamos a que se ejecuten en el core 0

```
> taskset -c 0 dd if=/dev/zero of=/dev/null &
> taskset -c 0 dd if=/dev/zero of=/dev/null &
```

- Hacemos top para comprobar el reparto de CPU
 - Deberían estar al 50%
 - > top
- Bajamos la prioridad a uno de ellos

```
> su
> renice 1 <pid>>
```

Hacemos top nuevamente

Contenido

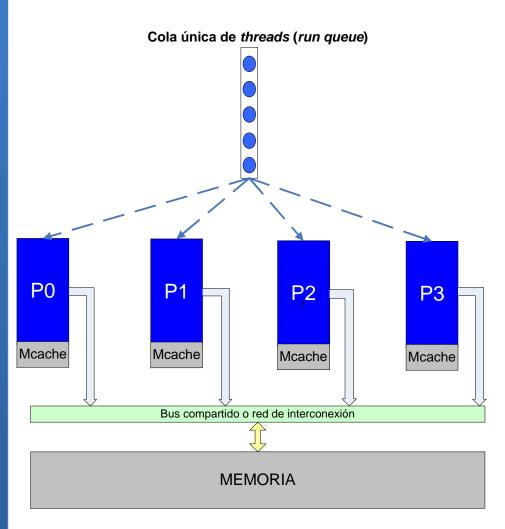


- Conceptos básicos
- Algoritmos de planificación
- Ejemplo: linux
- Planificación en multiprocesadores SMP

Planificación SMP

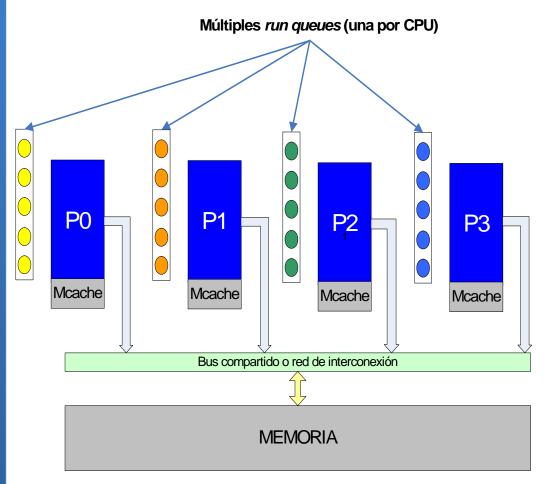
- Garantizar un equilibrio de carga
 - Que no haya un procesador ocioso y otros con mucha carga de trabajo
- Tener en cuenta la afinidad de procesos y procesadores
 - Importante al replanificar un proceso (evitar migrar)
- Tener en cuenta la compartición de datos entre procesos/hilos si hay varios nodos de memoria (NUMA)
 - Si dos hilos comparten memoria, probablemente sea bueno que compartan todo lo posible su nivel de jerarquía

Planificación SMP (Linux v2.4)



- Una única run queue para todos los procesadores
- Bueno para el load balancing
 - Todos los procesadores tienen, potencialmente el mismo trabajo
- Malo para la afinidad
 - El proceso A se ejecutó en la CPU1 y luego se le envía
 CPU2 → migración
 - ¿Qué supone una migración?
- Problemas de escalabilidad 50

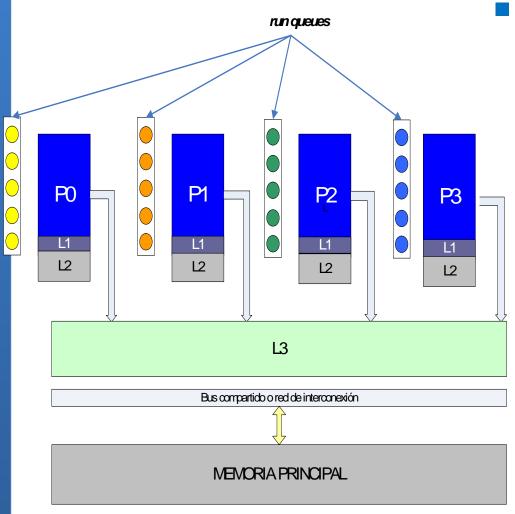
Planificación SMP (Linux v2.6.x+)



- Una cola de ejecución por procesador
- Mayor escalabilidad
- Periódicamente (o bajo demanda) se ejecuta el equilibrador de carga
 - Considera qué procesos pueden/deben migrarse
 - Tiene en cuenta la afinidad
- El usuario puede especificar la afinidad hilo-CPUs permitidas

→ Este modelo es el que utilizan la mayor parte de SSOO actuales de propósito general (Linux, Solaris, FreeBSD o MS Windows)

Planificación en multicore y SMT



- El SO ve cada core como un procesador independiente, pero no lo es
 - Algún nivel de cache compartido entre cores
 - Cores pueden tener
 SMT (ej:
 Hyperthreading) dos
 CPUs lógicas para el SO
 comparten todo el
 datapath (registros,
 unidades funcionales,
 colas...)