

Sistemas Operativos



Gestión de Procesos
Multitarea y Planificación



Agenda

- 1** Multitarea
- 2** Estructura del planificador
- 3** Intervención del planificador
- 4** Estrategias de planificación
 - Algoritmos no expropiativos
 - Algoritmos expropiativos
- 5** Colas Multinivel con Realimentación
- 6** Reparto proporcional/equitativo
- 7** Planificación en Multiprocesadores
- 8** FAQ

Agenda

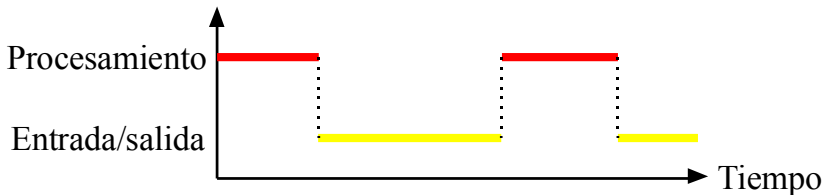


- 1** Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador
- 4 Estrategias de planificación
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores
- 8 FAQ

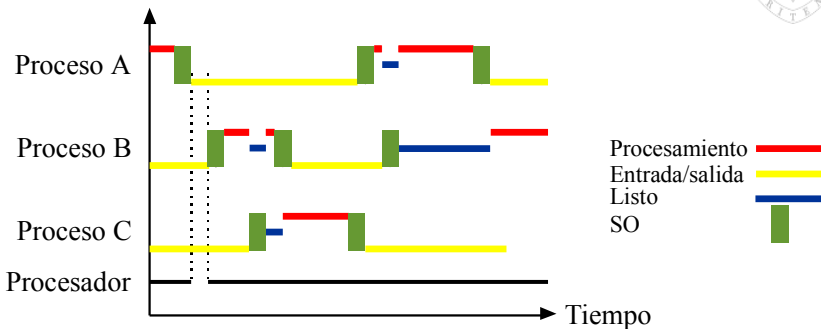


Base de la Multitarea

- Los procesos pasan por fases de E/S y CPU
- Mientras un proceso espera por E/S otro podría utilizar la CPU
- El SO mantiene en memoria varios procesos activos



Ejecución en un sistema multitarea



Proceso nulo o idle

Proceso que mantiene la cpu ejecutando instrucciones cuando no hay ningún otro proceso listo en el sistema. No hace nada útil. Es el menos prioritario.

Ventajas de la multitarea

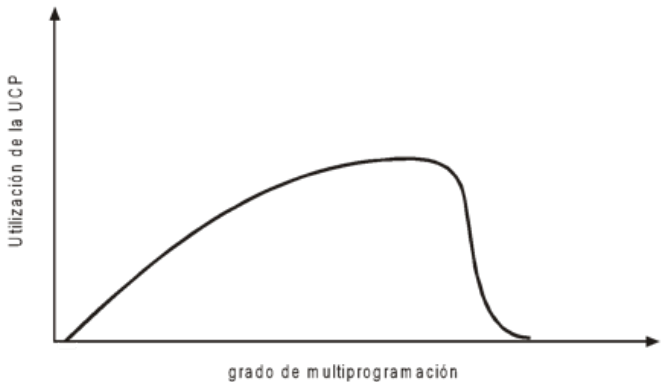


- Facilita la programación, dividiendo los programas en procesos (modularidad)
- Permite el servicio interactivo simultáneo de varios usuarios de forma eficiente
- Aprovecha los tiempos que los procesos pasan esperando a que se completen sus operaciones de E/S
- Aumenta el uso de la CPU



Grado de multiprogramación

- Grado de multiprogramación: nº de procesos activos
- Aumentarlo demasiado puede degradar el rendimiento global por paginación.

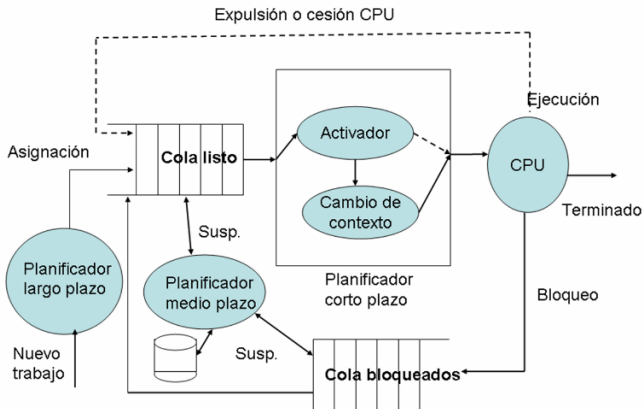


Agenda



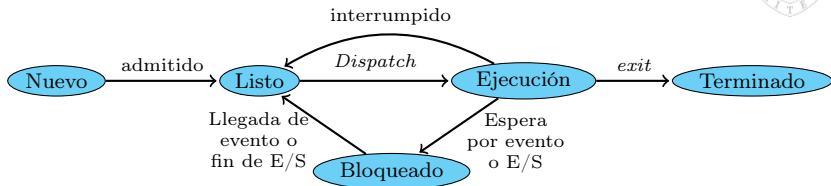
- 1 Multitarea
- 2 Estructura del planificador**
- 3 Intervención del planificador
- 4 Estrategias de planificación
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores
- 8 FAQ

Niveles de planificación



- **A largo plazo:** añadir procesos a ejecutar (batch)
- **A medio plazo:** añadir procesos a RAM
- **A corto plazo:** qué proceso tiene la CPU

Planificación a corto plazo



Activador o dispatcher: cede el control al siguiente proceso

- Escogido entre los hilos en estado **Listo (ready)**
- Depende de la **política de planificación**
- Cambiar de proceso implica un cambio de contexto

Agenda



- 1 Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador**
- 4 Estrategias de planificación
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores
- 8 FAQ

Excepciones/Interrupciones



El SO **siempre** entra a través de una excepción/interrupción

- El tratamiento de la interrupción supone:
 - 1 Detener la ejecución de instrucciones (programa actual)
 - 2 Cambiar a *modo privilegiado*
 - 3 Salvar el estado arquitectónico (en la pila de kernel del proceso)
 - 4 Ejecutar la rutina de tratamiento de excepción/interrupción
- La RTI
 - es código del SO
 - es ejecutada en el contexto del proceso activo
 - finaliza con un proceso de retorno de interrupción que restaura el contexto arquitectónico
- Algunas interrupciones y/o excepciones desencadenan la ejecución del planificador

Activación del Planificador



- 1** Periódicamente, en cada tick del reloj del sistema
 - El HW provee al SO de un contador programable que genera interrupciones periódicas (ticks)
 - Actualiza estadísticas para el algoritmo de planificación
 - Comprueba si le toca planificar otro proceso

Activación del Planificador



- 1** Periódicamente, en cada tick del reloj del sistema
 - El HW provee al SO de un contador programable que genera interrupciones periódicas (ticks)
 - Actualiza estadísticas para el algoritmo de planificación
 - Comprueba si le toca planificar otro proceso
- 2** Por el tratamiento de la interrupción de algún dispositivo de E/S

Activación del Planificador



- 1** Periódicamente, en cada tick del reloj del sistema
 - El HW provee al SO de un contador programable que genera interrupciones periódicas (ticks)
 - Actualiza estadísticas para el algoritmo de planificación
 - Comprueba si le toca planificar otro proceso
- 2** Por el tratamiento de la interrupción de algún dispositivo de E/S
- 3** Por una excepción provocada por el proceso en ejecución:
 - que lo bloquea (ej: fallo de página, llamada al sistema bloqueante)
 - que fuerza su terminación (ej: violación de segmento)
 - que termina voluntariamente el proceso en ejecución (`_exit()`)
 - que cede voluntariamente el procesador (`sched_yield()`)
 - que desbloquea o crea otro proceso (`fork, clone, ...`)

Activación del Planificador



- 1 Periódicamente, en cada tick del reloj del sistema
 - El HW provee al SO de un contador programable que genera interrupciones periódicas (ticks)
 - Actualiza estadísticas para el algoritmo de planificación
 - Comprueba si le toca planificar otro proceso
- 2 Por el tratamiento de la interrupción de algún dispositivo de E/S
- 3 Por una excepción provocada por el proceso en ejecución:
 - que lo bloquea (ej: fallo de página, llamada al sistema bloqueante)
 - que fuerza su terminación (ej: violación de segmento)
 - que termina voluntariamente el proceso en ejecución (`_exit()`)
 - que cede voluntariamente el procesador (`sched_yield()`)
 - que desbloquea o crea otro proceso (`fork, clone, ...`)
- 4 Porque se desbloquea un proceso más *importante* que el actual
 - Sucede realmente por alguno de los mecanismos anteriores

Cambio de Contexto



- Acciones realizadas por el SO para cambiar el proceso en ejecución en una CPU
 - Salvar el contexto del proceso saliente (registros →BCP)
 - Cambiar el estado del proceso saliente (En ejecución →Otro)
 - Configurar la MMU para el espacio de direcciones del proceso entrante:
 - Segmentos o regiones de memoria que puede usar
 - Puntero a la tabla de páginas
 - Flush de TLB
 - Cambiar el estado del proceso entrante, (Listo →En ejecución)
 - Restaurar su contexto (BCP →registros)
 - Realizar el retorno de Interrupción para continuar la ejecución del proceso (en modo usuario)
- Puede llegar a ser una operación bastante costosa



Agenda

- 1 Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador
- 4 Estrategias de planificación**
 - Algoritmos no expropiativos
 - Algoritmos expropiativos
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores
- 8 FAQ

Objetivos

- Optimizar uso de las CPUs
- Minimizar tiempo de espera
- Ofrecer reparto equitativo (justicia)
- Proporcionar grados de urgencia (prioridades)





- Parámetros por entidad (proceso o hilo)
 - Tiempo de ejecución (*turnaround*): creación - terminación
 - Tiempo de espera: tiempo total listo y sin CPU
 - Tiempo de respuesta: creación – 1er uso de CPU
- Parámetros globales
 - Porcentaje de utilización del procesador
 - Justicia: equitatividad en el reparto de CPU
 - Productividad: número de trabajos completados por unidad de tiempo



Agenda

- 1 Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador
- 4 Estrategias de planificación**
 - Algoritmos no expropiativos
 - Algoritmos expropiativos
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores
- 8 FAQ

Algoritmos No Expropiativos



El proceso en ejecución conserva la CPU hasta que:

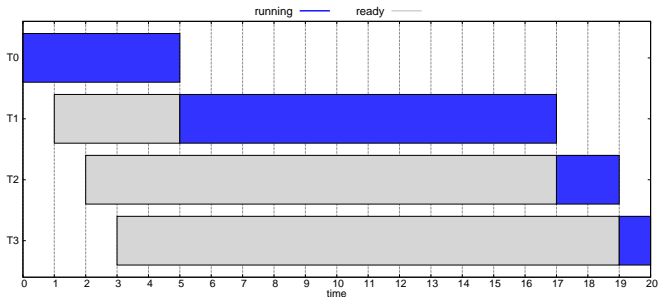
- se bloquea
- cede expresamente la CPU
- termina su ejecución.

First Come First Served (FCFS)



- Se planifican los procesos por orden de entrada en la cola de listos
- Muy sencillo y óptimo en uso de CPU

| Proceso o Hilo | Instante de Llegada | Tiempo de CPU (ms) |
|----------------|---------------------|--------------------|
| T0 | 0 | 5 |
| T1 | 1 | 12 |
| T2 | 2 | 2 |
| T3 | 3 | 1 |

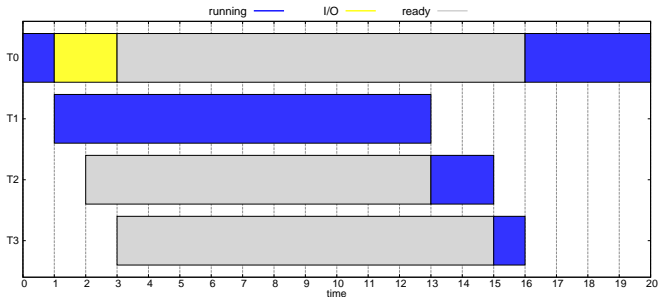




FCFS: tiempos medios altos

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

| Proceso o Hilo | Instante de llegada | Tiempo de CPU (ms) |
|----------------|---------------------|--------------------|
| T0 | 0 | 5 |
| T1 | 1 | 12 |
| T2 | 2 | 2 |
| T3 | 3 | 1 |



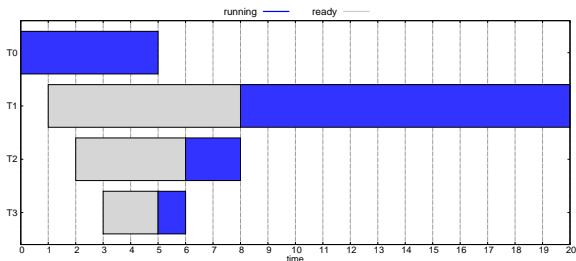
Shortest Job First (SJF)



Se planifica primero el proceso más corto

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

| Proceso o Hilo | Instante de llegada | Tiempo de CPU (ms) |
|----------------|---------------------|--------------------|
| T0 | 0 | 5 |
| T1 | 1 | 12 |
| T2 | 2 | 2 |
| T3 | 3 | 1 |

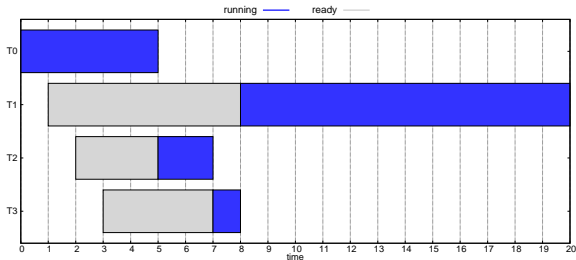


Basado en Prioridades



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

| Proceso o Hilo | Instante de llegada | Tiempo de CPU (ms) | Prioridad |
|----------------|---------------------|--------------------|-----------|
| T0 | 0 | 5 | 4 |
| T1 | 1 | 12 | 3 |
| T2 | 2 | 2 | 1 |
| T3 | 3 | 1 | 2 |



Agenda



- 1 Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador
- 4 Estrategias de planificación**
 - Algoritmos no expropiativos
 - Algoritmos expropiativos
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores
- 8 FAQ

Algoritmos Expropiativos



- Cuando el planificador lo considera pertinente, cambia el proceso/hilo que hay ejecutando por otro
 - La decisión depende del algoritmo de planificación
- Más adecuados para SO de propósito general
- Manejan bien mezclas de trabajos interactivos y trabajos intensivos en CPU

Round Robin



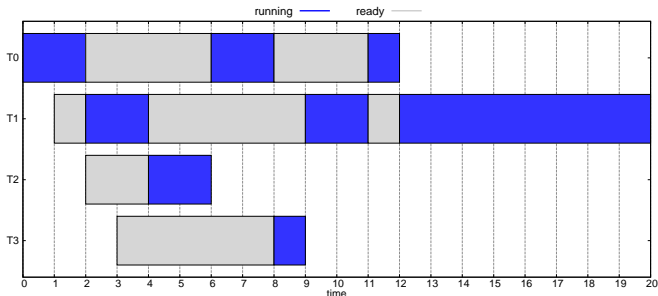
- FCFS + time slice
 - Asignación rotatoria de CPU
 - Se asigna un tiempo máximo de procesador que el proceso puede consumir sin ser expropiado (**time slice** o **cuanto**)
- Uso en sistemas de tiempo compartido
 - Equitativo (mejor por uid que por proceso)

Round Robin: ejemplo (I)



Cuanto: 2

| Proceso o Hilo | Instante de llegada | Tiempo de CPU (ms) |
|----------------|---------------------|--------------------|
| T0 | 0 | 5 |
| T1 | 1 | 12 |
| T2 | 2 | 2 |
| T3 | 3 | 1 |

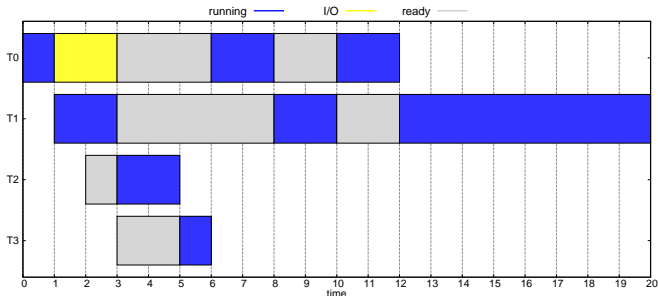


Round Robin: ejemplo (II)



Cuanto: 2

| Proceso o Hilo | Instante de llegada | Tiempo de CPU (ms) |
|----------------|---------------------|--------------------|
| T0 | 0 | 5 |
| T1 | 1 | 12 |
| T2 | 2 | 2 |
| T3 | 3 | 1 |

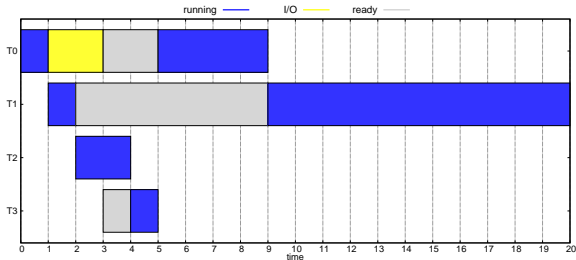


Shortest Remaining Time First (SRTF)



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

| Proceso o Hilo | Instante de Llegada | Tiempo de CPU (ms) |
|----------------|---------------------|--------------------|
| T0 | 0 | 5 |
| T1 | 1 | 12 |
| T2 | 2 | 2 |
| T3 | 3 | 1 |

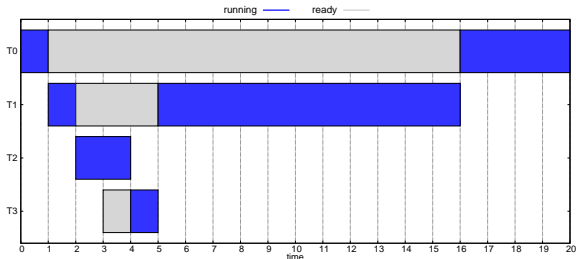


Expropiativo basado en prioridades



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

| Proceso o Hilo | Instante de llegada | Tiempo de CPU (ms) | Prioridad |
|----------------|---------------------|--------------------|-----------|
| T0 | 0 | 5 | 4 |
| T1 | 1 | 12 | 3 |
| T2 | 2 | 2 | 1 |
| T3 | 3 | 1 | 2 |



Agenda



- 1 Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador
- 4 Estrategias de planificación
- 5 Colas Multinivel con Realimentación**
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores
- 8 FAQ

Motivación



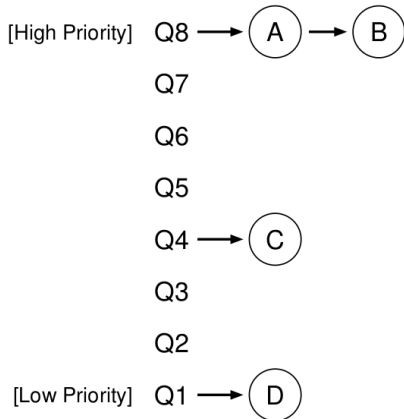
Colas Multinivel con Realimentación (MLFQ)

- Queremos reducir el tiempo de ejecución medio (*turnaround*)
 - Como lo hace SJF
 - Sin conocer a priori el tiempo de ejecución de cada tarea
 - El sistema debe aprender las características de cada tarea
- Queremos reducir el tiempo de respuesta
 - Como lo hace Round Robin
 - Importante para tareas interactivas
- Objetivos contrapuestos



Reglas básicas

- Varias colas con distinto nivel de prioridad
- La prioridad de un proceso es la de la cola a la que está asignado
- Si $\text{Prioridad}(A) > \text{Prioridad}(B)$, A se ejecuta y B no
- Si $\text{Prioridad}(A) == \text{Prioridad}(B)$, se usa RR para planificar la ejecución de A & B
- En el ejemplo:
 - A y B se alternan en uso de la CPU
 - C y D no se ejecutan



Tareas con prioridades dinámicas



- Variar la prioridad de las tareas en función de su comportamiento
 - Si un trabajo repetidamente libera la CPU bloqueándose por E/S debemos asignarle alta prioridad
 - Si una tarea repetidamente usa la CPU de forma intensiva por largos periodos de tiempo, debemos asignarle menor prioridad

El pasado para predecir el futuro

MLFQ intenta *aprender* de los procesos mientras ejecutan, usando así su *historia* para predecir su comportamiento futuro.

Asignación de prioridades: intento 1

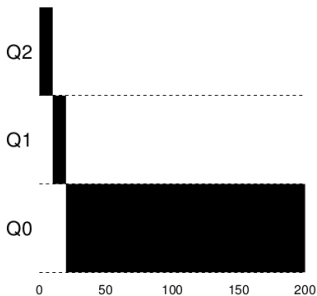


- Cuando una tarea entra en el sistema, se le asigna la máxima prioridad (cola superior)
- Si una tarea consume su cuanto de tiempo completo mientras ejecuta, se *reduce* su prioridad (se mueve a la siguiente cola)
- Si una tarea cede la CPU antes de consumir su cuanto de tiempo, se mantiene en el *mismo* nivel de prioridad.

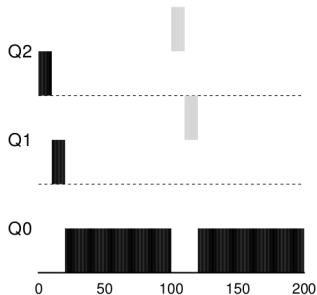


Ejemplo:

Un proceso de larga duración:



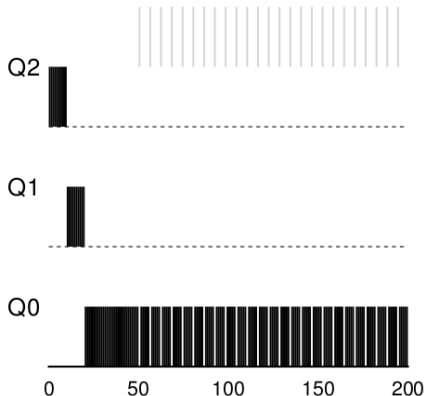
Aparece una tarea más corta:





Ejemplo: tarea interactiva

- Tarea A intensiva en CPU, de larga duración (negro)
- Tarea B interactiva (gris)
 - Ráfagas cortas de CPU, menores que un cuanto, seguidas de fases de E/S



Problemas



Problemas de la estrategia vista hasta ahora:

1 Posible inanición

- Tareas intensivas en CPU de larga duración
- Con muchas tareas interactivas, que pueden copar la CPU

2 Posibilidad de engañar al planificador

- Ceder cpu antes de agotar el cuanto
- Permanecemos en máximo nivel de prioridad
- Apurando al 99% el cuanto podemos monopolizar la CPU

3 No se adapta a los cambios de fase de las tareas

- Una tarea intensiva en CPU que pasa a una fase interactiva se queda en el nivel menos prioritario

Evitar el problema de inanición



Priority Boost:

- Periódicamente incrementar la prioridad de todos los procesos del sistema
 - Habitualmente, con algún periodo S , todos los procesos se mueven a la cola más prioritaria

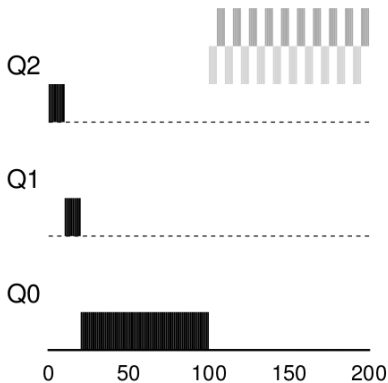
Soluciona dos problemas de golpe:

- Inanición: al pasar periódicamente a las colas más prioritarias todos los procesos podran progresar
- Adaptación a las fases de las tareas. Un cambio a una fase interactiva será detectada y atendida cuando se haga el *Priority Boost*

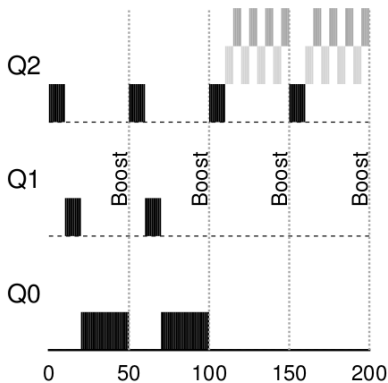


Priority Boost: ejemplo

- Tarea A: larga, intensiva en CPU
- Tareas B y C: interactivas, aparecen a los 100ms



Sin Priority Boost
Inanición de A



Con Priority Boost cada 50ms
A ejecuta cada 50ms

Impedir el engaño al planificador



Es necesario mejorar el *Accounting*:

- Llevar la cuenta del tiempo de CPU total de cada tarea
 - En una o varias ráfagas

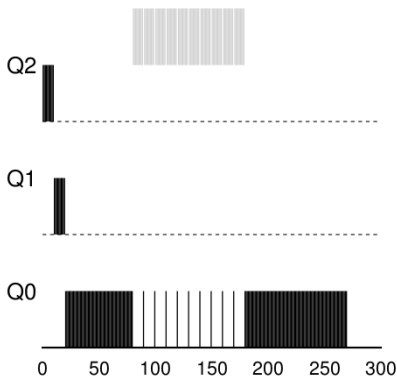
Y modificar las reglas de cambio de prioridades:

- Cuando el tiempo de CPU consumido supere la asignación para ese nivel, la tarea se mueve al siguiente nivel (reduce prioridad)

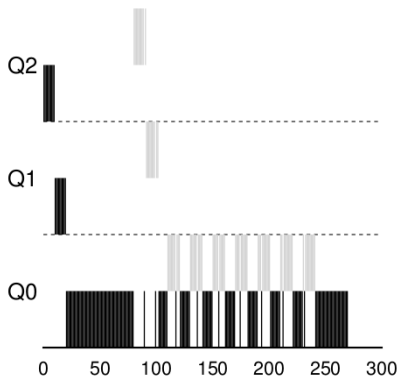


Impedir engaño: ejemplo

- Tarea A: larga, intensiva en CPU
- Tarea B: trata de engañar al planificador, cediendo cpu antes de consumir el cuanto



Sin protección de engaño
B monopoliza la CPU



Con protección de engaño
B no puede monopolizar la CPU

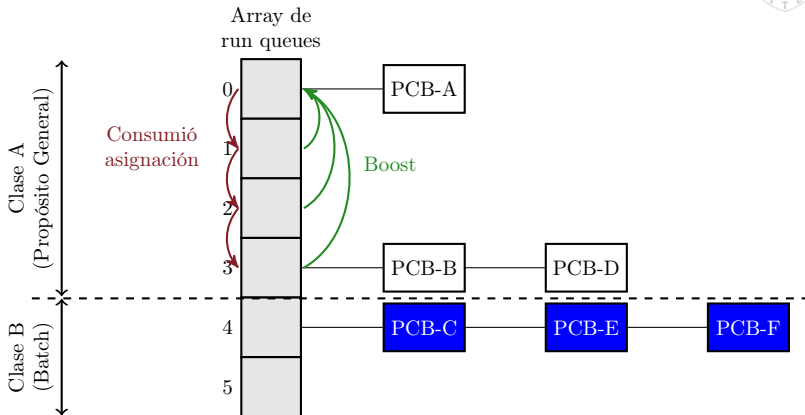
Variaciones habituales



Hay múltiples variantes sobre MLFQ:

- Organización de colas por clases, para distintos tipos de tareas
 - Tareas de kernel, Procesos de usuario, Tareas de tiempo real, Trabajos por lotes (batch)
- Número de colas, cuantos e intervalos de boosting:
 - Solaris: 60 colas para la clase Time Sharing, con cuantos variando de 20ms a 100ms, y Priority Boost en intervalos de 1s
- Mecanismos de cambio prioridad
 - FreeBSD: uso de formulas, asignación de prioridad en función del tiempo de CPU y con decaimiento del tiempo de CPU en lugar de Boosting.
 - Solaris: mecanismo de cambio configurable por el administrador del sistema (tablas en ficheros de configuración).
- Admitir consejo de los usuarios para modificar las prioridades
 - Llamada al sistema `nice` en UNIX

MLFQ: estructura ejemplo



Agenda



- 1 Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador
- 4 Estrategias de planificación
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo**
- 7 Planificación en Multiprocesadores
- 8 FAQ

Objetivos



Objetivos del reparto equitativo (fair):

- No optimizar el tiempo de ejecución o el de respuesta
- Tratar de garantizar un reparto equitativo
 - Que cada trabajo obtenga un cierto porcentaje de uso de CPU
 - Los porcentajes se fijan por tipo de trabajo, prioridad, etc

Claves:

- ¿Como podemos diseñar un planificador que garantice un reparto equitativo?
- ¿Cuáles son los mecanismos clave para ello?
- ¿Son efectivos y eficientes?

Concepto básico: Tickets



Tickets

La cantidad de tickets que tiene un proceso representa la porción de CPU que le corresponde.

Ejemplo:

Supongamos un sistema con dos procesos, A y B, en el que A tiene 75 tickets y B tiene sólo 25. En este caso el reparto de CPU deseado sería:

- A: $\frac{75}{75+25} \cdot 100 = 75\%$ de CPU
- B: $\frac{25}{75+25} \cdot 100 = 25\%$ de CPU



Planificador por Lotería

- Conoce el número total de tickets del sistema (N)
- Los procesos se ordenan por número de tickets, asignándoles los números de lotería. En nuestro ejemplo:
 - A con 75 tickets, números del 0 al 74
 - B con 25 tickets, números del 74 al 99
- Periódicamente (*time slice*) se saca un número aleatorio en $[0, N - 1]$.
- Se cede la CPU al proceso que tenga asignado ese número

```
// numero aleatorio del ganador
int winner = getrandom(0, totaltickets);
int counter = 0;
node_t *current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break;
    current = current->next;
}
// Current queda apuntando al ganador
```



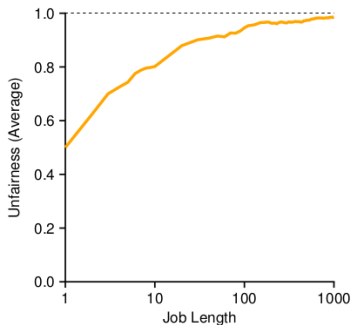
Ejemplo:

Planificación en 20 *time slices* consecutivos de dos procesos: A con 75 tickets y B con 25 tickets:

| | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|----|----|
| 63 | 85 | 70 | 39 | 76 | 17 | 29 | 41 | 36 | 39 | 10 | 99 | 68 | 83 | 63 | 62 | 43 | 0 | 49 | 49 |
| A | | A | A | | A | A | A | A | A | A | | A | | A | A | A | A | A | A |
| | B | | | B | | | | | | | B | | B | | | | | | |

Resultado:

- B ejecuta $4/20 = 20\%$, inferior al 25% deseado.
- Carácter aleatorio hace que no se alcance la equitatividad perfecta con trabajos cortos
- La equitatividad mejora conforme aumenta la longitud de las tareas.



Problema inherente: ¿Cómo repartir los tickets?

Planificación equitativa determinista



Planificador de zancada (stride), propuesto por Waldspurger

- A cada tarea se le asigna una *zancada*, inversamente proporcional al número de tickets que tiene asignados (más tickets, más prioridad, menor *zancada*)
- Para cada tarea se lleva la cuenta de sus *pasos*
- Cada vez que una tarea ejecuta se incrementan sus *pasos* en una cantidad igual a su *zancada*
- Se planifica siempre la tarea con menor número de *pasos*

Implementación muy sencilla también:

```
current = remove_min(queue);           //pick client with minimum pass
schedule(current);                     //use resource for quantum
current->pass += current->stride;       //compute next pass using stride
insert(queue, current);                //put back into the queue
```



Ejemplo:

Tareas: A, B y C; con 100, 50 y 250 tickets (total 400 tickets).
Zancada(X) = 10000 / Tickets(X). El resultado sería:

| Pasos(A) (zancada=100) | Pasos (B) (zancada=200) | Pasos(C) (zancada=40) | Ejecuta |
|---------------------------|----------------------------|--------------------------|---------|
| 0 | 0 | 0 | A |
| 100 | 0 | 0 | B |
| 100 | 200 | 0 | C |
| 100 | 200 | 40 | C |
| 100 | 200 | 80 | C |
| 100 | 200 | 120 | A |
| 200 | 200 | 120 | C |
| 200 | 200 | 160 | C |
| 200 | 200 | 200 | ... |

$$A: 2/8 = 100/400 = 25\%, \quad B: 1/8 = 50/400 = 12.5\%, \\ C: 5/8 = 250/400 = 62.5\%$$

- Planificador de lotería → equitatividad con el tiempo
- Planificador de zancada → equitatividad al final del ciclo

Completely Fair Scheduler (CFS)



- Planificación a corto plazo en Linux 2.6.23+
- Objetivos
 - Aproximar planificación completamente equitativa
 - Proporcionar buenos tiempos de respuesta
 - Optimizar al máximo el tiempo de planificación, que escale bien.
 - Cola de planificación implementada como un *árbol* balanceado
 - Soportar niveles de prioridad
 - 100 para procesos real-time (RR y FIFO)
 - 40 para procesos normales (CFS)
- Consultar código en:
<https://code.woboq.org/linux/linux/kernel/sched/fair.c.html>



CFS: Conceptos básicos

- Un contador de tiempo virtual (*vruntime*) para cada tarea
- Al planificar se escoge siempre la tarea con menor *vruntime*
 - Es la que se merece más la CPU en ese momento
- Se asigna un tiempo variable a las tareas
 - *sched_latency_ns*: unidad de tiempo que se reparte equitativamente entre todas las tareas listas
 - Se asigna una fracción **proporcional** al *peso* de la tarea.
 - Es una medida de la prioridad de la tarea
- El contador *vruntime* de cada tarea se incrementa en una cantidad **inversamente proporcional** al *peso* de la tarea
 - El *vruntime* de las tareas irá incrementándose al mismo ritmo



CFS: Pesos y nice

La prioridad/peso se fija en función del valor *nice*

- Inversamente proporcional al nice (n): $W = K/\alpha^n$
 - Una tarea *amable* demanda poca CPU
- Objetivo: dos procesos con $\Delta \text{nice} = 1$ queremos que tengan un $\Delta T_{\text{CPU}} = 10\%$

$$T_a = \frac{K/\alpha^n}{K/\alpha^n + K/\alpha^{n+1}} = \frac{\alpha}{\alpha + 1}$$

$$T_b = \frac{K/\alpha^{n+1}}{K/\alpha^n + K/\alpha^{n+1}} = \frac{1}{\alpha + 1}$$

$$T_a - T_b = \frac{\alpha - 1}{\alpha + 1} = 0.1 \rightarrow \alpha = 1.222222\dots$$

El kernel de Linux aproxima a $\alpha = 1.25$ ($\Delta T_{\text{CPU}} \approx 11\%$), que es representable de forma exacta en punto fijo, y redondea los pesos para minimizar el error de la inversa ($M/1.25^n$) con aritmética de punto fijo.

CFS: Pesos y nice



El peso se fija en función del valor *nice*: $W = 1024/1.25^{nice}$

```
const int sched_prio_to_weight[40] = {  
    /* -20 */    88761,      71755,      56483,      46273,      36291,  
    /* -15 */    29154,      23254,      18705,      14949,      11916,  
    /* -10 */    9548,       7620,       6100,       4904,       3906,  
    /*  -5 */    3121,      2501,      1991,      1586,      1277,  
    /*   0 */    1024,      820,       655,       526,       423,  
    /*   5 */    335,      272,      215,      172,      137,  
    /*  10 */    110,      87,       70,       56,       45,  
    /*  15 */    36,      29,      23,      18,      15,  
};
```

■ Consultar tablas en:

<https://code.woboq.org/linux/linux/kernel/sched/core.c.html#7089>



CFS: ejemplo 1

- 2 tareas intensivas en CPU, una con nice 0 y otra con 3
- *sched_latency* de 24ms
- El tiempo de CPU se reparte proporcionalmente al peso:

$$T_{CPU}(x) = sched_latency \cdot \frac{peso(x)}{\sum_{i=0}^n peso(i)}$$

- El *vruntime* se incrementaría para cada proceso según:

$$\Delta vruntime_i = \frac{peso_{nice=0}}{peso(i)} \cdot \Delta runtime_i = \frac{1024}{peso(i)} \cdot \Delta runtime_i$$

Al ser intensivas en CPU, asumimos $\Delta runtime_i = T_{CPU}(i)$:

| Tarea | nice | Peso | T_{CPU} (ms) | % CPU | $\Delta vruntime(ms)$ |
|-------|------|------|----------------|-------|-----------------------|
| A | 0 | 1024 | 15.855484 | 66.06 | 15.855484 |
| B | 3 | 526 | 8.144516 | 33.93 | 15.855484 |



CFS: ejemplo 1

Ejemplo con el planificador CFS

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

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|----------|----|----|------|-----|-----|---|------|------|---------|---------|
| 22548 | christi+ | 20 | 0 | 7828 | 744 | 680 | R | 49.5 | 0.0 | 0:04.12 | dd |
| 22513 | christi+ | 20 | 0 | 7828 | 740 | 676 | R | 49.2 | 0.0 | 0:06.23 | dd |
| ... | | | | | | | | | | | |

```
> sudo renice 3 22513
> top
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|----------|----|----|------|-----|-----|---|------|------|---------|---------|
| 22548 | christi+ | 20 | 0 | 7828 | 744 | 680 | R | 65.8 | 0.0 | 0:41.23 | dd |
| 22513 | christi+ | 23 | 3 | 7828 | 740 | 676 | R | 33.2 | 0.0 | 0:34.67 | dd |
| ... | | | | | | | | | | | |



CFS: ejemplo 2

3 tareas, con valores de nice 0, 0 y 3, y *sched_latency* de 24ms

| Tarea | nice | Peso | T_{CPU} (ms) | % CPU | $\Delta vruntime(ms)$ |
|-------|------|------|----------------|-------|-----------------------|
| A | 0 | 1024 | 9.547785 | 39.78 | 9.547785 |
| B | 0 | 1024 | 9.547785 | 39.78 | 9.547785 |
| C | 3 | 526 | 4.904429 | 20.43 | 9.547785 |

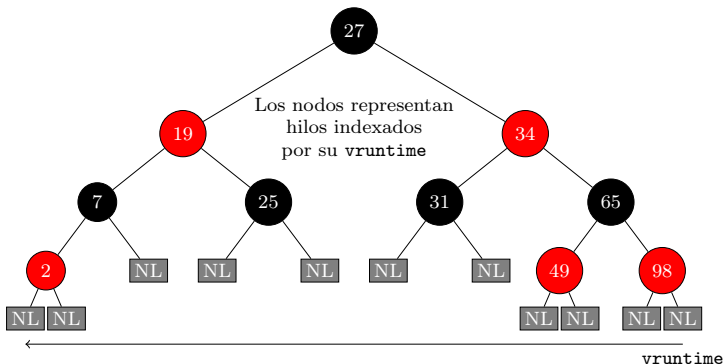
Ejemplo con el planificador CFS

```
> taskset -c 0 dd if=/dev/zero of=/dev/null &
> taskset -c 0 dd if=/dev/zero of=/dev/null &
> taskset -c 0 dd if=/dev/zero of=/dev/null &
> top
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
27540 christi+  20   0   7828    744    680 R   33.2   0.0   0:54.70 dd
27638 christi+  20   0   7828    684    620 R   33.2   0.0   0:51.17 dd
27643 christi+  20   0   7828    744    680 R   33.2   0.0   0:50.77 dd
...
> sudo renice 3 27643
> top
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
27638 christi+  20   0   7828    684    620 R   39.9   0.0   1:46.91 dd
27540 christi+  20   0   7828    744    680 R   39.5   0.0   1:50.42 dd
27643 christi+  23   3   7828    744    680 R   20.6   0.0   1:44.66 dd
...
```



CFS: planificación rápida

- Siguiente hilo: el de menor *vruntime*
 - Hilos en un árbol balanceado *Red-Black*
 - Serie de **videos en youtube** sobre árboles *red-black*



Comandos Linux de interés



Consultar la página de manual de los siguientes comandos:

- **nice**: ejecutar un comando con una prioridad entre -19 y 20
- **renice**: alterar la prioridad de un proceso en ejecución
- **taskset**: consultar o establecer la afinidad de CPU de un proceso o comando a ejecutar
- **sysctl**: consultar/modificar parámetros del kernel

sysctl

```
> sudo sysctl -A | grep sched
kernel.sched_child_runs_first = 0
kernel.sched_domain.cpu0.domain0.busy_factor = 64
...
kernel.sched_domain.cpu1.domain0.busy_factor = 64
kernel.sched_domain.cpu1.domain0.busy_idx = 2
...
kernel.sched_latency_ns = 12000000
kernel.sched_migration_cost_ns = 500000
kernel.sched_min_granularity_ns = 1500000
kernel.sched_nr_migrate = 32
kernel.sched_rr_timeslice_ms = 25
```

Agenda



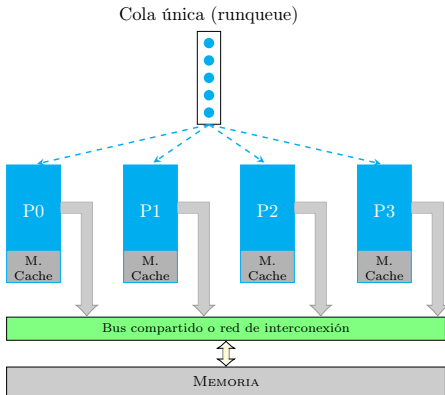
- 1 Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador
- 4 Estrategias de planificación
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores**
- 8 FAQ

Objetivos adicionales



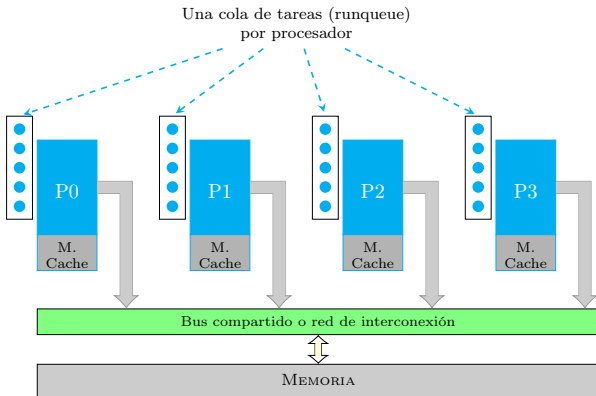
- 1 Garantizar un equilibrio de carga
 - Que no haya un procesador ocioso y otros con mucha carga
- 2 Tener en cuenta la afinidad de procesos y procesadores
 - Importante al replanificar un proceso (evitar migrar)
- 3 Tener en cuenta la compartición de datos entre procesos/hilos si hay varios nodos de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

Opción 1: cola de planificación única



- Bueno para el balanceo de carga
- Malo para afinidad: los procesos pueden ir cambiando de CPU
- La cola es un cuello de botella: problema de escalabilidad

Opción 2: una cola de planificación por CPU



- Mayor escalabilidad
- Equilibrado de carga: periódico o bajo demanda
 - Considera qué procesos pueden/deben migrarse
 - Tiene en cuenta la afinidad
- Utilizada en la mayoría de los sistemas actuales



Agenda

- 1 Multitarea
- 2 Estructura del planificador
- 3 Intervención del planificador
- 4 Estrategias de planificación
- 5 Colas Multinivel con Realimentación
- 6 Reparto proporcional/equitativo
- 7 Planificación en Multiprocesadores
- 8 **FAQ**



Preguntas por responder

- Si dos o más procesos quieren cooperar
 - ¿Cómo intercambian información?
 - Si el planificador decide cuándo se ejecuta cada uno, ¿no hay problema en el acceso a recursos compartidos?
 - ¿Pueden sincronizarse y controlar el acceso a un recurso compartido?
¿Qué recursos son necesarios?
- ¿Qué diferencias hay en el caso de que sean hilos de un mismo proceso los que deseen cooperar?