Planificación de CPU

Semestre II-2013

Planificación de CPU

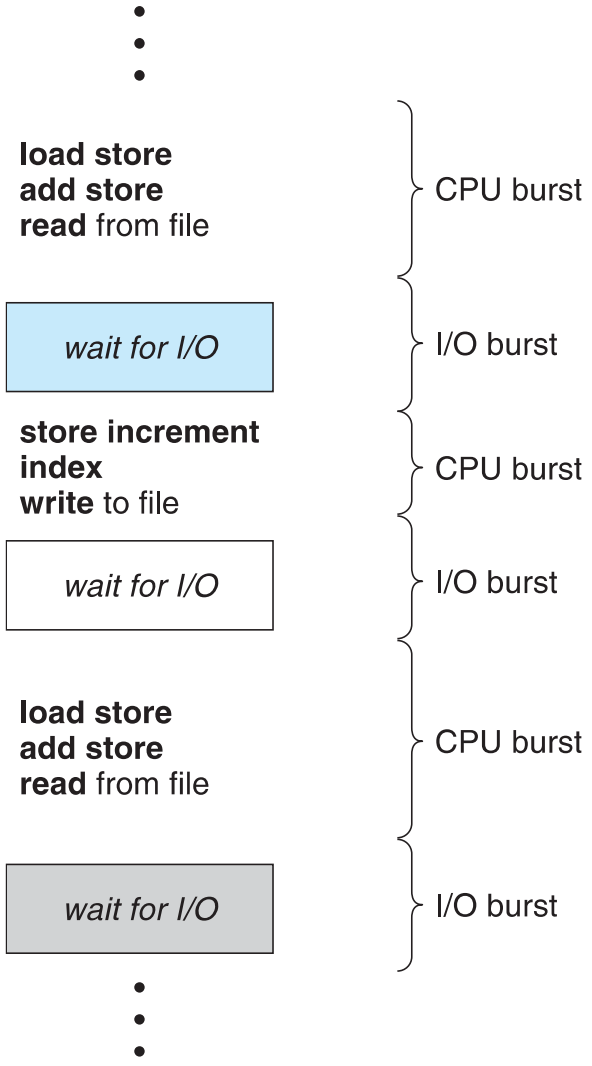
- Conceptos Básicos
- Criterios de Planificación
- Algoritmos de Planificación
- Planificación de Threads
- Planificación en Sistemas Multiprocesador
- Ejemplos de Planificación en SO

Objetivos

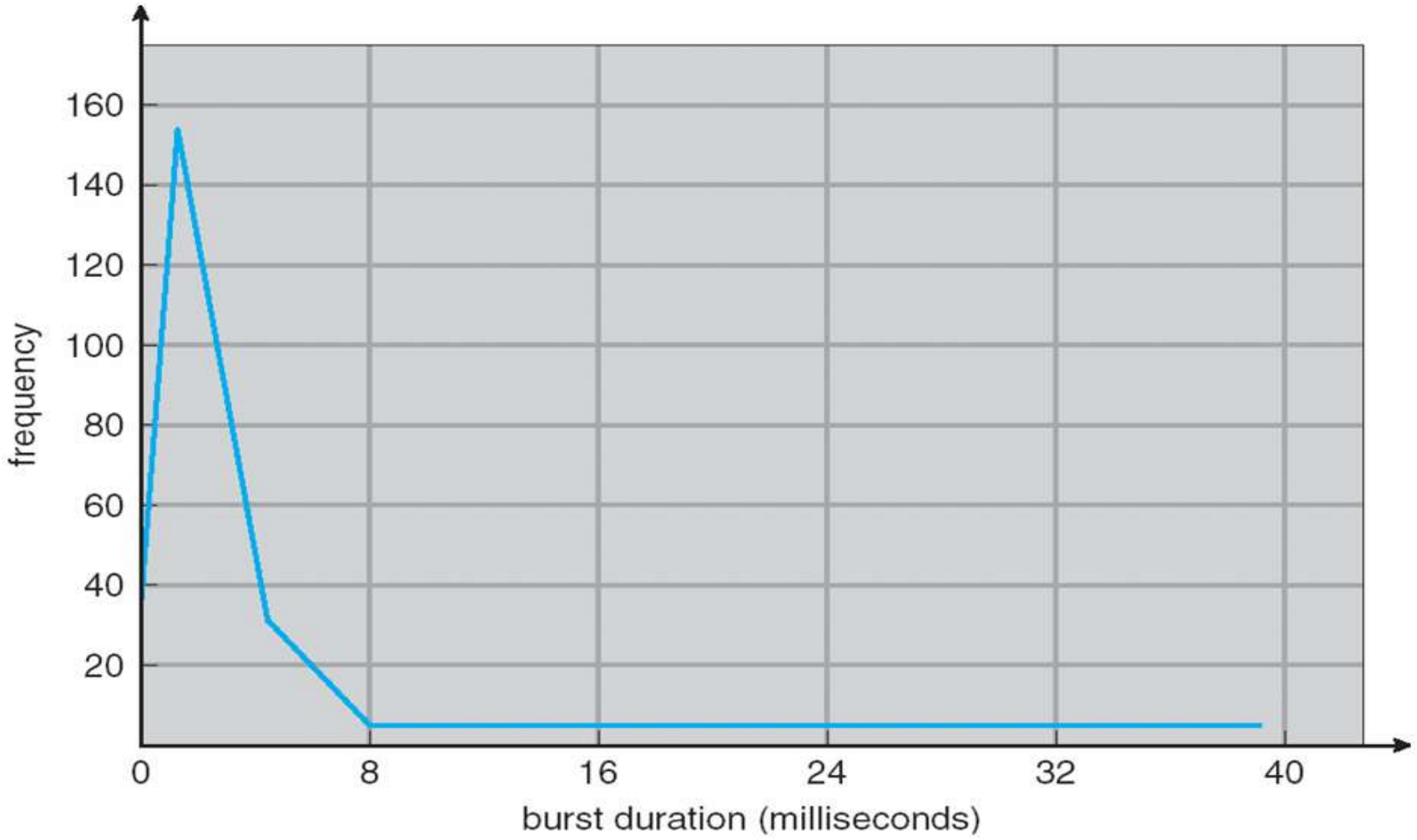
- Introducir el concepto de planificación de CPU, el cual es un concepto básico en SO multiprogramados
- Describir varios algoritmos de planificación de CPU
- Discutir los criterios de evaluación para la selección de algoritmos de planificación de CPU para sistemas
- Examinar los algoritmos de planificación presentes en diferentes SO

Conceptos Básicos

- Maximizar la utilización del CPU mediante multiprogramación
- Ciclos de CPU-I/O – La ejecución de procesos se basa en ciclos de ejecución de CPU y ciclos de espera en I/O
- Una ráfaga de CPU suele ser seguida de una ráfaga de I/O
- Las ráfagas de CPU deben ser distribuidas de manera coherente



Histograma de la Ráfagas de CPU en el Tiempo



Planificación de CPU

- El planificador a corto plazo selecciona a que proceso de la cola de listos se le asignará el CPU
  - Esta cola puede estar ordenada de diferentes maneras
- La planificación de CPU es una decisión que se lleva acabo cuando un proceso:
  1. Cambia del estado ejecución al estado bloqueado
  2. Cambia del estado ejecución al estado listo
  3. Cambia del estado bloqueado al estado listo
  4. Termina
- La planificación del 1 al 4 es no apropiativa
- No apropiación vs Apropiación
- Considere las siguientes situaciones bajo un esquema apropiativo
  - Considere el acceso a un conjunto de datos compartidos
  - Considere la apropiación que podría llevarse acabo mientras se esta en modo kernel
  - Considere que ocurra una interrupción durante una actividad crucial del SO

Despachador

- El módulo de despacho se encarga de darle el control del CPU al proceso seleccionado por el planificador a corto plazo; esto implica:
  - Realizar los cambios de contexto pertinentes
  - Realizar el cambio de modo
  - Saltar a la posición indicada del programa de usuario para reiniciar la ejecución de dicho programa
- Latencia de despacho – Se define como el tiempo que tarda el despachador en detener un proceso e iniciar otro

Criterios de Planificación

- Utilización de CPU – Mantener el CPU ocupado tanto tiempo como sea posible
- Throughput - # de procesos que completan su ejecución por unidad de tiempo
- Tiempo de ejecución – Cantidad de tiempo para ejecutar un proceso en particular
- Tiempo de espera – Cantidad de tiempo en la que un proceso se encuentra en la cola de listo
- Tiempo de respuesta – Cantidad de tiempo que transcurre desde que una solicitud ha sido enviada hasta que se obtiene la primera respuesta

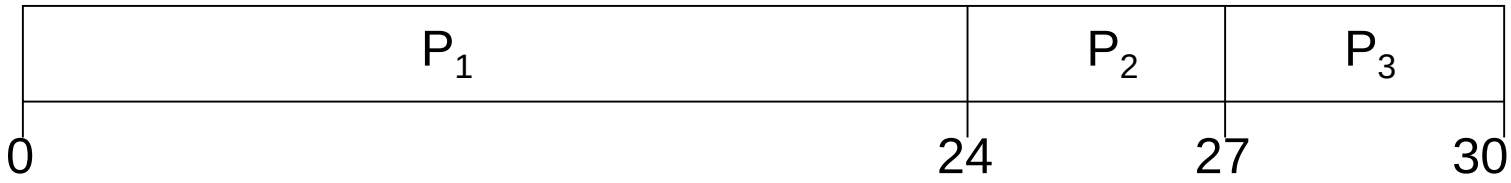
Criterios de Optimización para Algoritmos de Planificación

- Maximizar la utilización del CPU
- Maximizar el throughtput
- Minimizar el tiempo de ejecución
- Minimizar el tiempo de espera
- Minimizar el tiempo de respuesta

Planificación First-Come, First-Served (FCFS)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suponga que los procesos llegan en este orden: P_1 , P_2 , P_3
  El Diagrama de Gantt para dicha planificación es:

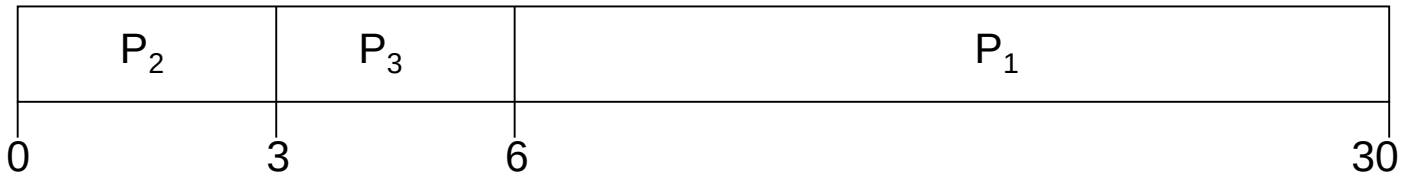


- Tiempo de espera para P_1 = 0; P_2 = 24; P_3 = 27
- Tiempo promedio de espera: (0 + 24 + 27)/3 = 17

Planificación First-Come, First-Served (FCFS)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suponga que los procesos llegan en este orden: P_2 , P_3 , P_1
  El Diagrama de Gantt para dicha planificación es:



- Tiempo de espera para P_1 = 6; P_2 = 0; P_3 = 3
- Tiempo promedio de espera: (6 + 0 + 3)/3 = 3
- Desempeño muy superior al caso anterior
- Efecto caravana – Procesos cortos detrás de procesos largos
  - Considere un proceso orientado a CPU y muchos orientados a I/O

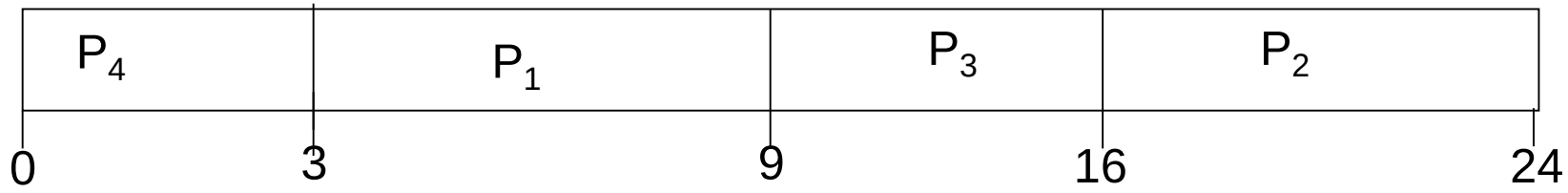
Planificación Shortest-Job-First (SJF)

- Asocia a cada proceso la longitud de su próxima ráfaga de CPU
  - Utiliza esta longitud para determinar el próximo proceso a planificar con base a la longitud más corta
- SJF es óptimo – Ofrece el menor promedio en tiempo de espera a un conjunto de procesos
  - Puede ser complicado conocer la longitud de la próxima ejecución en CPU
  - Podría consultarse con el usuario

Ejemplo de SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- Planificación usando SJF



- Promedio en tiempo de espera = (3 + 16 + 9 + 0) / 4 = 7

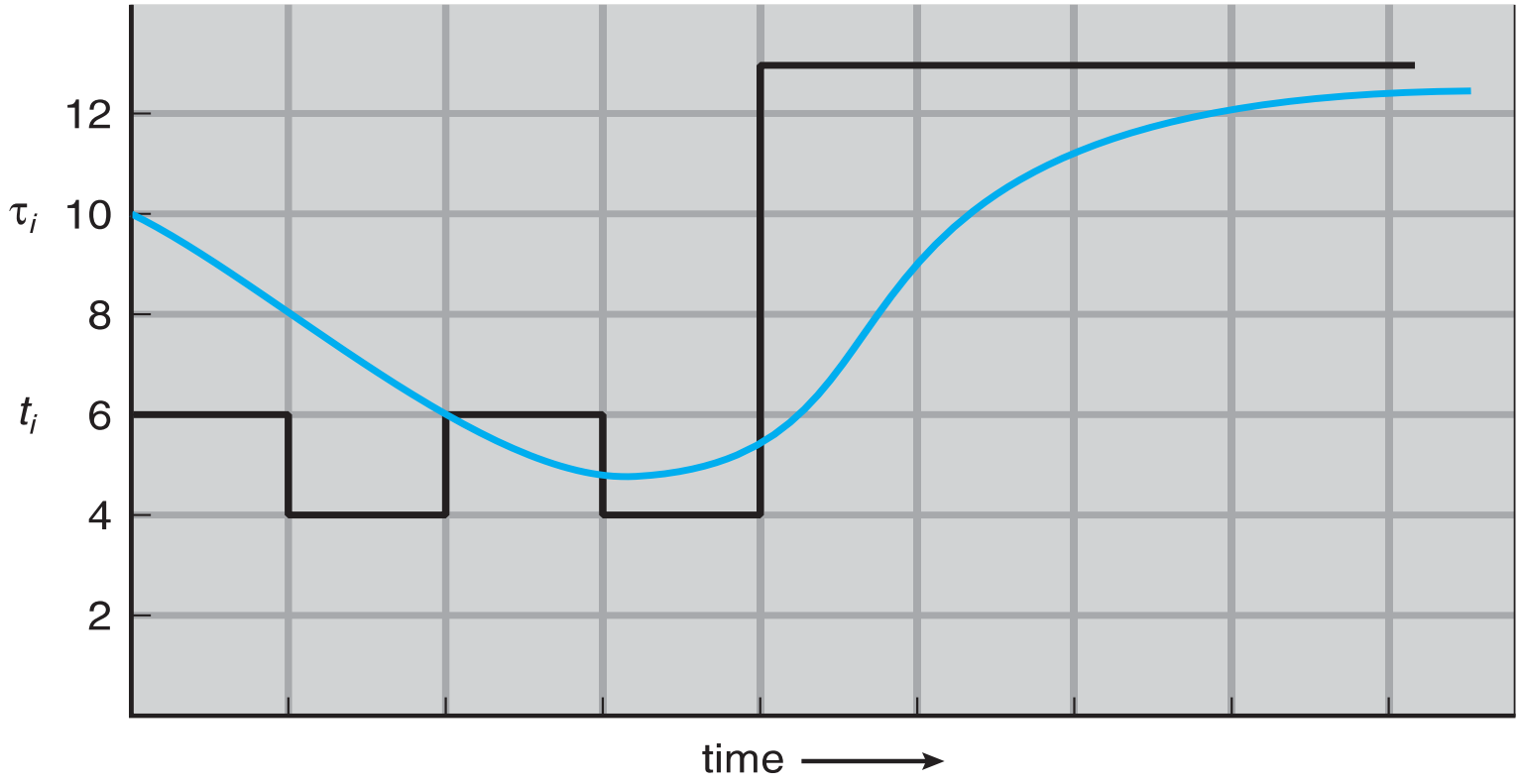
Determinando la Longitud de la Próxima Ráfaga de CPU

- Podría solo estimarse la longitud – El comportamiento debería ser similar al ejemplo anterior
  - Entonces se selecciona el proceso cuya predicción de próxima ráfaga de CPU sea menor
- Podrían usarse las longitudes de ráfagas previas de CPU, usando un promedio exponencial

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- Generalmente, α vale ½
- La versión apropiativa es llamada Shortest-Remaining-Time-Fisrt

Predicción de la Longitud de la Próxima Ráfaga de CPU



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Ejemplos del Promedio Exponencial

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - La historia reciente no cuenta
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Solo cuenta la última ráfaga de CPU
- Si se expande la formula se obtiene:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

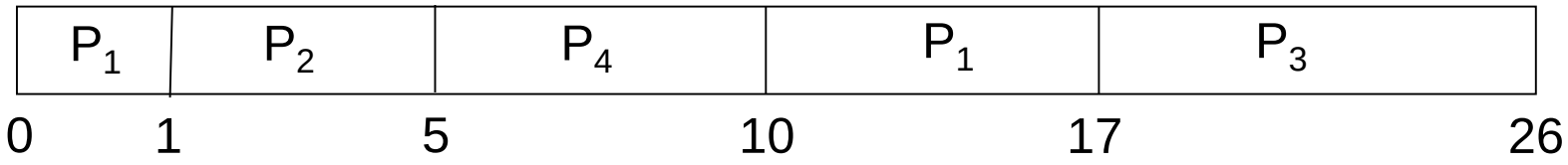
- Dado que α y $(1 - \alpha)$ son menores o iguales a 1, cada termino sucesivo tiene menos peso que su predecesor

Ejemplo de Shortest-Remaining-Time-First

- Ahora nosotros añadiremos los conceptos de tiempo interactivo variante y apropiación al análisis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Diagrama de Gantt* SJF apropiativo



- Tiempo promedio de espera = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

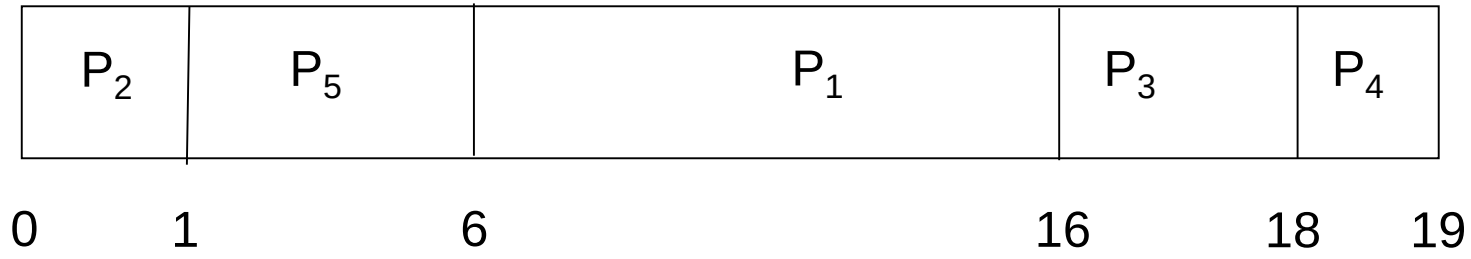
Planificación por Prioridad

- Una numero (entero) de prioridad se asocia con cada proceso
- El CPU se asigna al procesos con más alta prioridad (por ejemplo un entero pequeño → mayor prioridad)
  - Apropiativa
  - No apropiativa
- SJF es un planificación donde la prioridad es el inverso de la predicción de la longitud de la próxima ráfaga de CPU
- Problema – Inanición – Procesos con baja prioridad podrían nunca ser ejecutados
- Solución – Envejecimiento – Con el tiempo los procesos incrementa su prioridad en el sistema

Ejemplo de Planificación con Prioridades

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Diagrama de Gantt usando prioridades



- Promedio de tiempo de espera = 8.2 msec

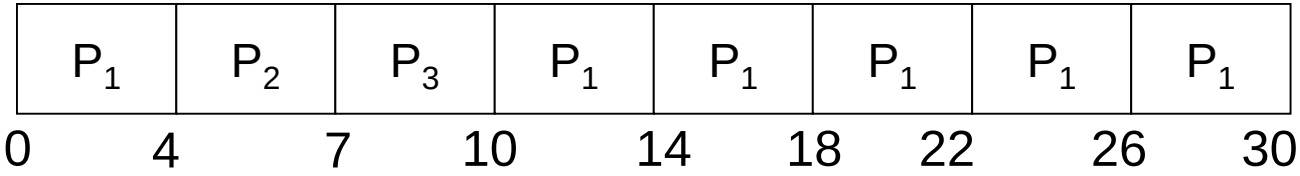
Round Robin (RR)

- Cada proceso toma pequeñas unidades de tiempo en CPU (quantum de tiempo q), usualmente entre 10 y 100 milisegundos. Luego que este tiempo finaliza, el proceso es desalojado del CPU y se coloca al final de la cola de listos
- Si hay n procesos en la cola de listos y el quantum de tiempo es q, entonces cada proceso tomara 1/n trozos de tiempo en el CPU de a lo sumo q unidades de tiempo. Ningún proceso deberá esperar más de (n-1)q unidades de tiempo para poder ejecutarse.
- Una interrupción ocurre cada quantum de tiempo para planificar al próximo proceso
- Desempeño
  - q muy grande → FIFO
  - q muy pequeño → q debe ser largo con respecto al tiempo invertido en un cambio de contexto, de otra manera la sobrecarga será muy grande

Ejemplo de RR con q=4

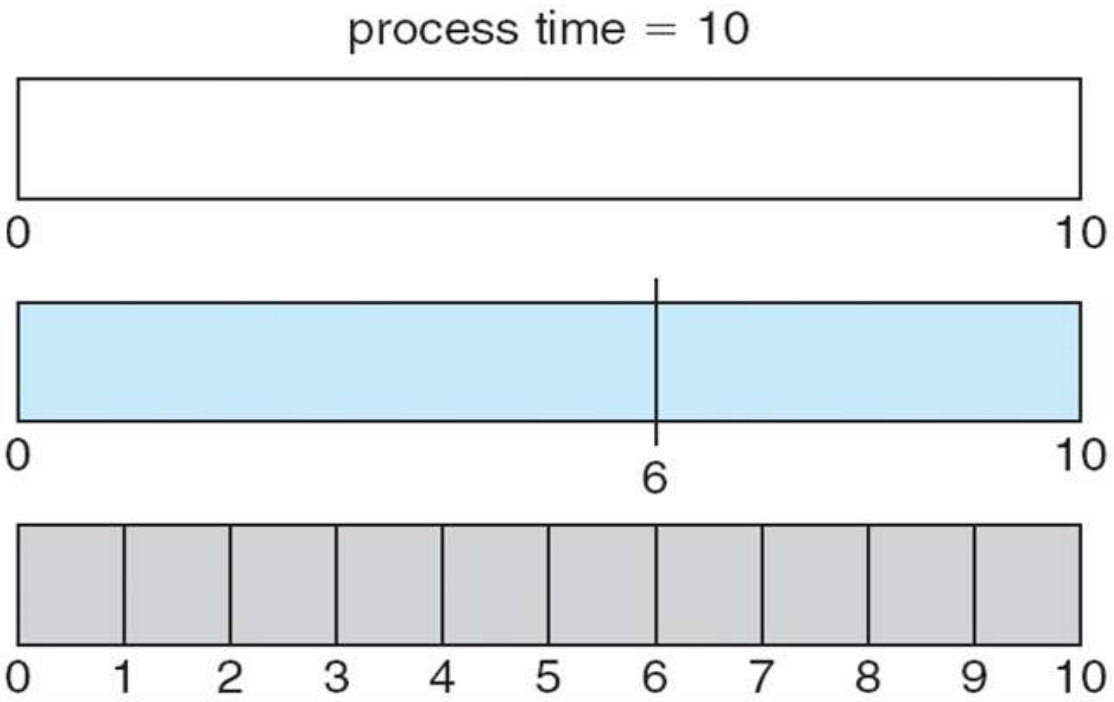
Process	Burst Time
P_1	24
P_2	3
P_3	3

- El diagrama de Gantt es:



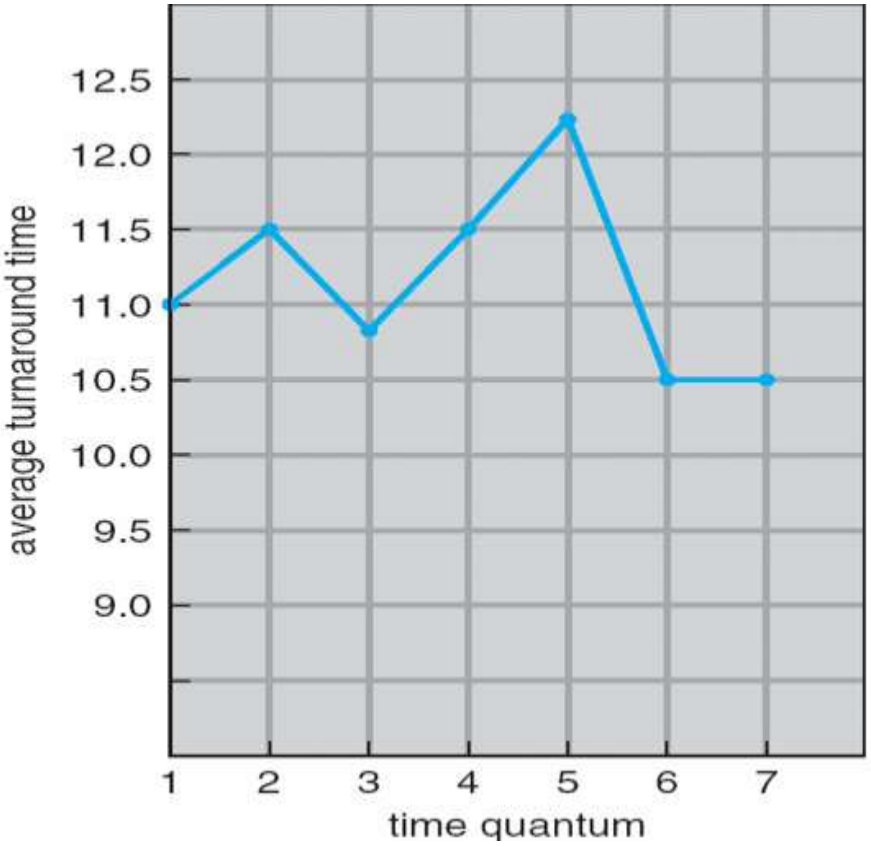
- Típicamente , el promedio de tiempo de ejecución es mayor que SJF, pero se obtiene una mejor respuesta
- q debe ser largo comparado con el tiempo necesario para los cambios de contexto
- q usualmente oscila entre 10 y 100 ms, el cambio de contexto toma aproximadamente < 10 useg

Tiempo del Quantum y Tiempo de Cambio de Contexto



quantum	context switches
12	0
6	1
1	9

El Tiempo de Ejecución Respecto al Quantum de Tiempo



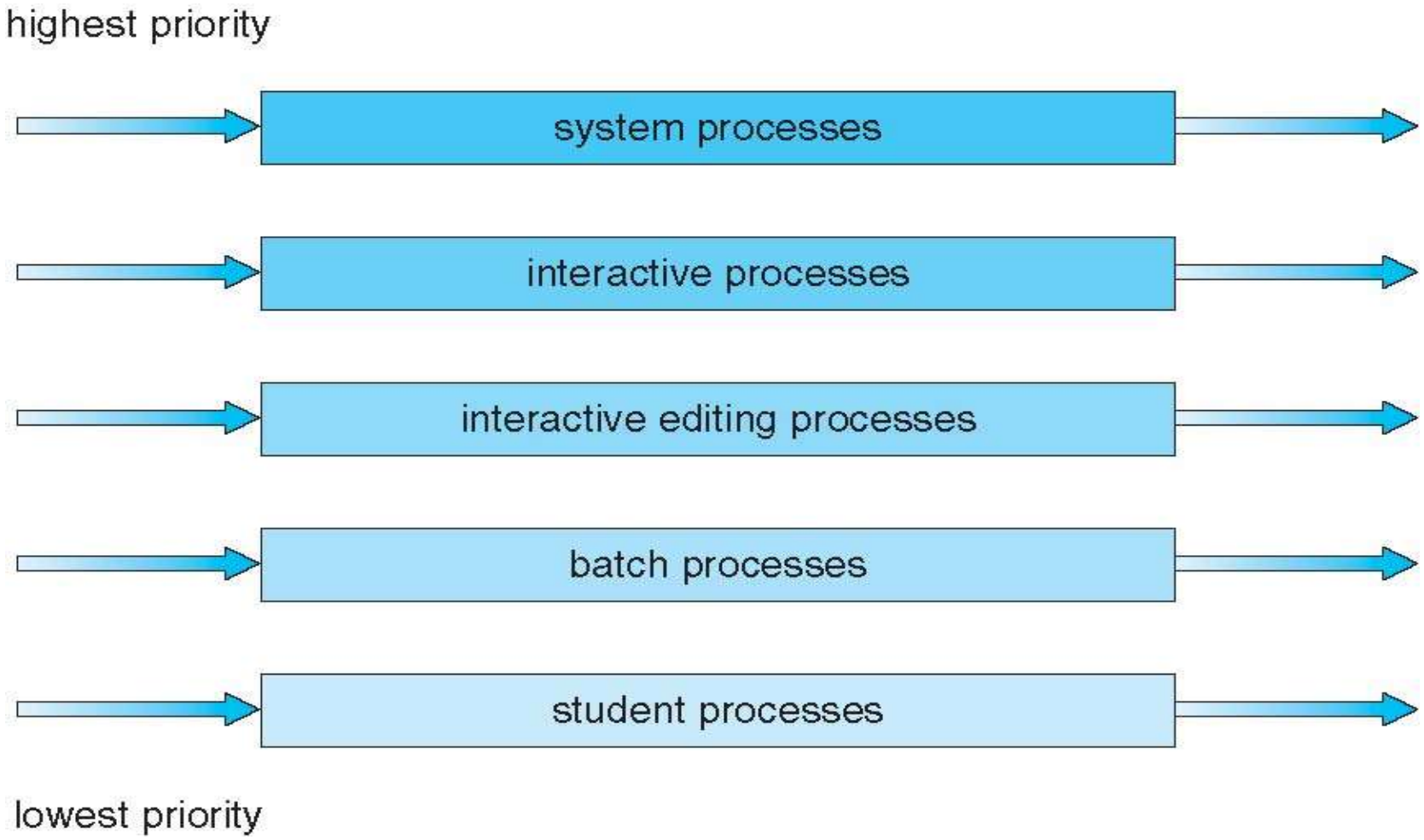
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Colas Multinivel

- La cola de listo se particiona en colas separadas, por ejemplo:
  - foreground (interactivos)
  - background (batch)
- Los procesos permanecen en un cola determinada
- Cada cola posee su propio algoritmo de planificación:
  - foreground – RR
  - background – FCFS
- La planificación debe realizarse entre ambas colas:
  - Planificación por prioridad fija, por ejemplo, se atiende primero a los procesos foreground y luego a los background. Es posible la inanición
  - Ranuras de tiempo – A cada cola se le asigna una cierta cantidad de tiempo en CPU de acuerdo a los procesos que contenga, por ejemplo, 80% del tiempo para los procesos foreground, y 20% para los demás

Planificación usando Colas Multinivel

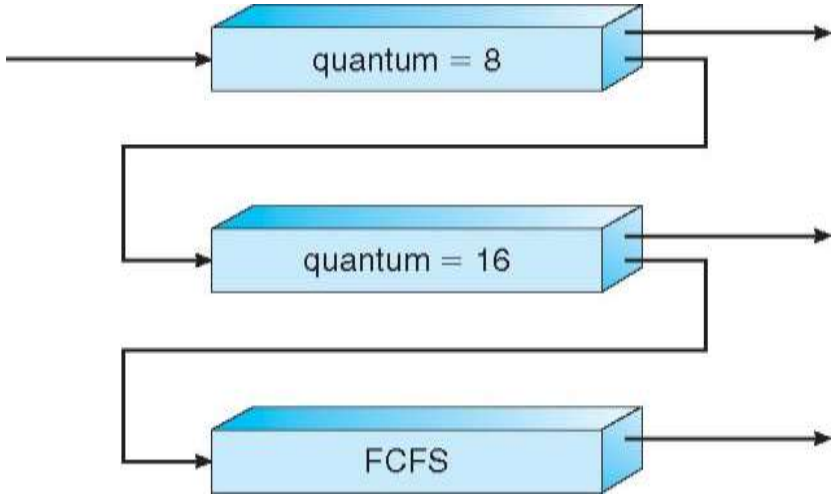


Colas Multinivel Retroalimentadas

- Un proceso puede moverse entre varias colas. El envejecimiento puede ser utilizado para implementar este concepto.
- En una planificación que utilice colas multinivel retroalimentadas es necesario definir los siguientes parámetros.
  - Número de colas
  - Algoritmo de planificación a ejecutar en cada cola
  - Método usado para determinar cuando un proceso asciende
  - Método usado para determinar cuando un proceso es degradado
  - Método para determinar a cual cola ingresará un proceso que necesita ser servido

Ejemplo Colas Multinivel Retroalimentadas

- Tres colas
  - Q0 – RR, q= 8
  - Q1 – RR, q= 16
  - Q2 – FCFS
- Planificación
  - Un trabajo nuevo ingresa en la cola Q0, donde se sirve usando FCFS
    - Cuando este obtiene el CPU, el trabajo se ejecuta por 8 unidades de tiempo
    - Si el trabajo no finaliza en 8 unidades de tiempo, este es movido a la cola Q1
  - Un trabajo en la cola Q1 nuevamente se sirve usando FCFS y recibe 16 unidades de tiempo adicionales
    - Si aún con estas 16 unidades de tiempo el trabajo no termina su ejecución, este se remueve del CPU y pasa a la cola Q2



Planificación de Threads

- Es necesario distinguir entre threads a nivel de usuario y threads a nivel de kernel
- Cuando existe el soporte de threads, se planifican threads, no procesos
- En los modelos muchos a uno y muchos a muchos, la planificación la realiza la biblioteca de threads en el espacio de usuario haciendo uso del LWP
  - Conoce el alcance de la contienda del proceso (process-contention scope → PCS), ya que la planificación se realiza sobre los threads que pertenecen a un proceso
  - Típicamente la planificación se basa en prioridades, las cuales establece el programador
- Los threads a nivel de kernel son planificados de acuerdo a la disponibilidad del CPU basándose en el alcance de contienda del sistema (system-contention scope → SCS), el cual define básicamente la competencia de todos los threads del sistema.

Planificación en Pthread

- El API permite especificar tanto el PCS como el SCS durante la creación del thread
  - PTHREAD_SCOPE_PROCESS planifica a los threads usando la planificación definida en PCS
  - PTHREAD_SCOPE_SYSTEM planifica los threads usando la planificación definida en SCS
- Puede ser limitado por el SO – En Linux y Mac OS X solo se permite PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread t tid[NUM THREADS];
   pthread attr t attr;
   /* get the default attributes */
   pthread attr init(&attr);
   /* first inquire on the current scope */
   if (pthread attr getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD SCOPE PROCESS)
         printf("PTHREAD SCOPE PROCESS");
      else if (scope == PTHREAD SCOPE SYSTEM)
         printf("PTHREAD SCOPE SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM THREADS; i++)
      pthread create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM THREADS; i++)
      pthread join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread exit(0);
}
```

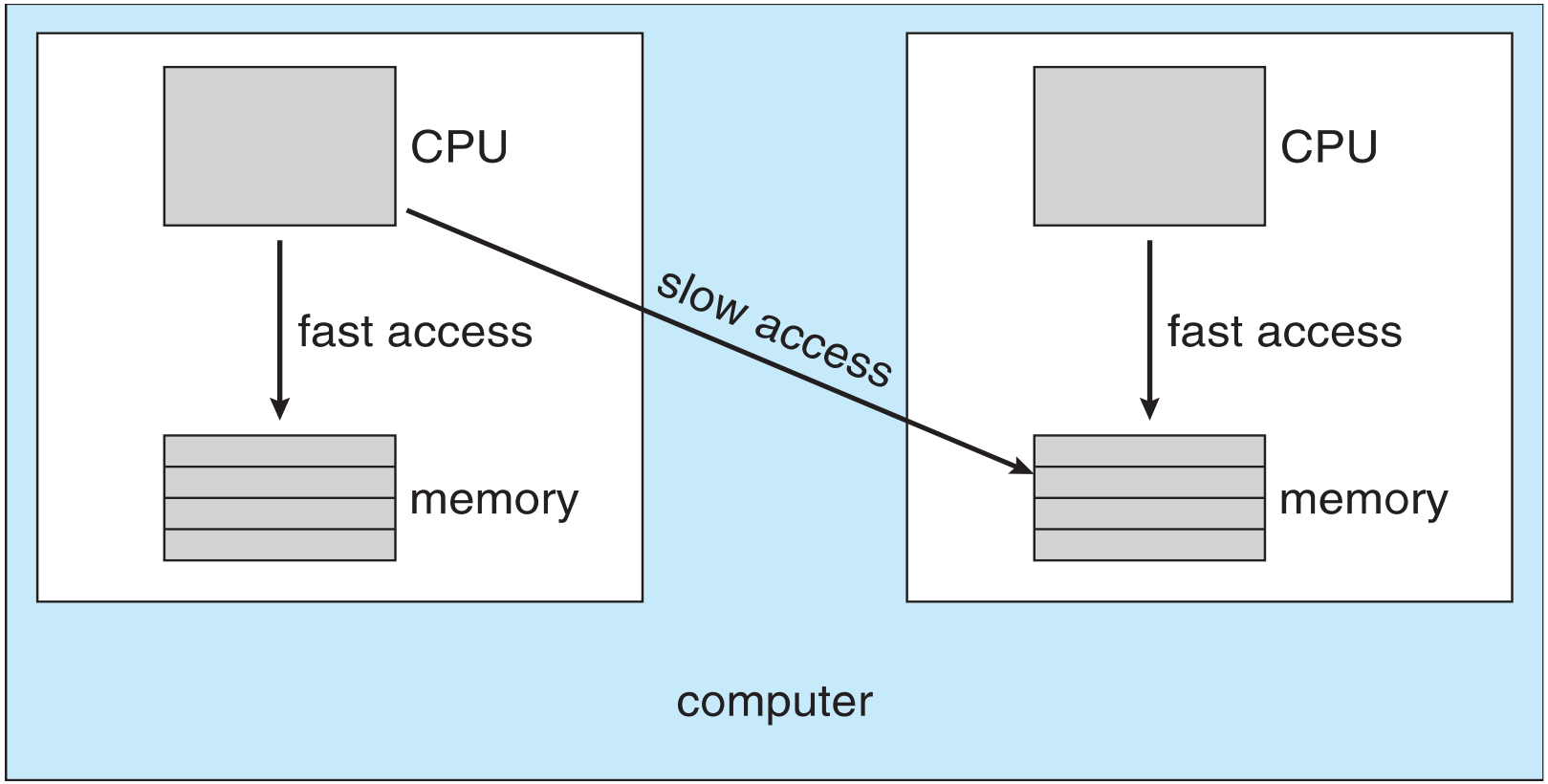
Planificación en Sistemas Multiprocesador

- La planificación del CPU es más compleja cuando múltiples CPUs se encuentran disponibles
- **Procesadores Homogéneos** dentro de un multiprocesador
- **Multiprocesamiento Asimétrico** – Solo un procesador accede a las estructuras de datos del sistema, aliviando la necesidad de compartición de datos
- **Multiprocesamiento Simétrico (SMP)** – Cada procesador se planifica por separado. Todos los procesos comparten una cola de listo común, o la cola de listo es individual por cada procesador
  - Actualmente, la última opción es lo más común

Planificación en Sistemas Multiprocesador

- Afinidad con un procesador – Los procesos pueden tener afinidad por el procesador en el cual se encuentra ejecutándose
  - Afinidad suave
  - Afinidad fuerte
  - ¿Qué utilidad tiene la afinidad en este contexto?

Planificación de CPU y NUMA



Note that memory-placement algorithms can also consider affinity

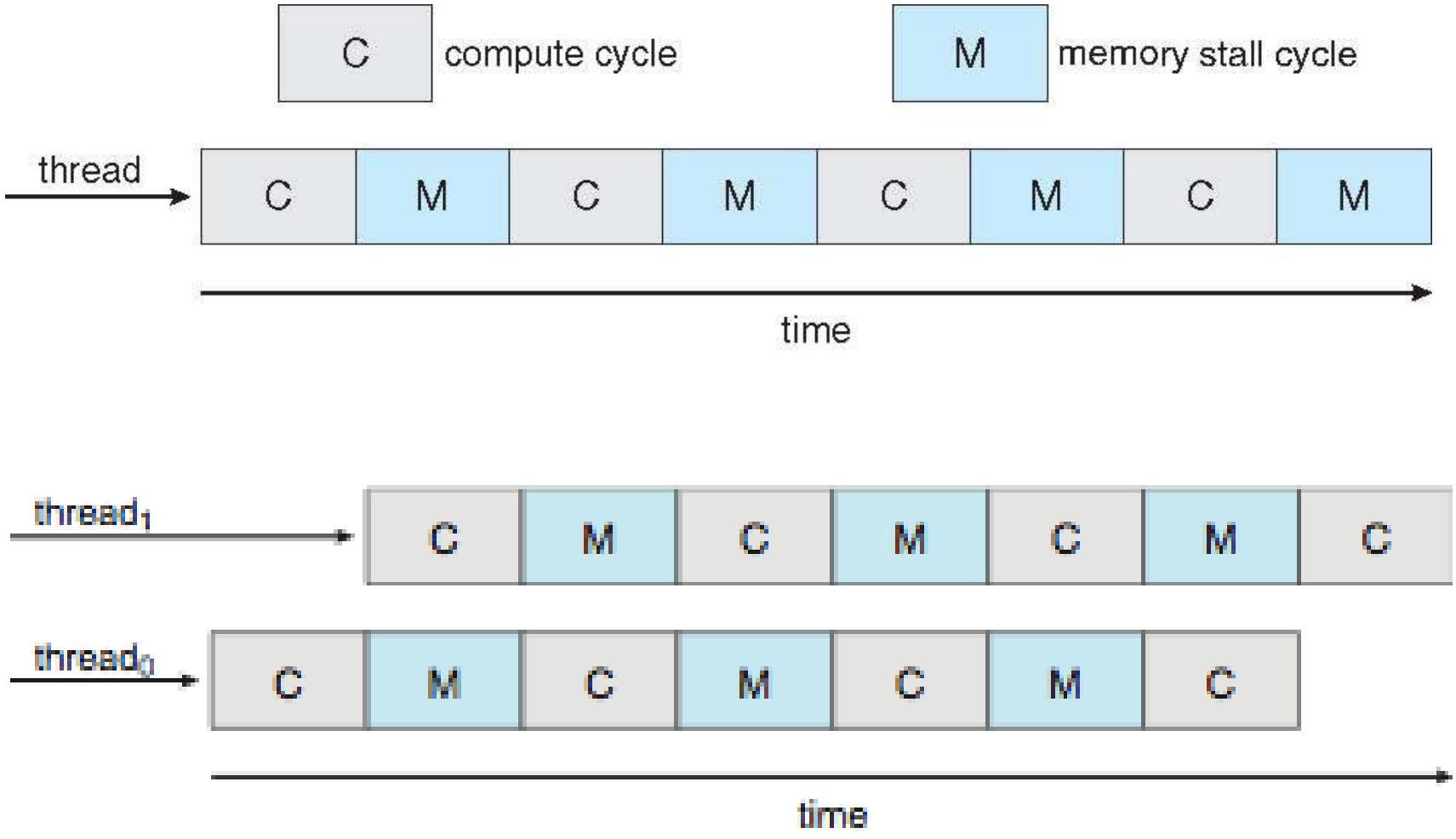
Planificación en Sistema Multiprocesador – Balanceo de Carga

- En SMP es conveniente mantener con carga a todos los CPUs del sistema → Eficiencia
  - ¿Cómo hago?
- El balanceo de caga intenta mantener las cargas de trabajo distribuidas de manera equitativa en el sistema
- **Push migration** – Periódicamente una tarea revisar la carga de cada procesador, y si consigue un CPU sobrecargado realiza un push de tareas a otro CPU
- **Pull migration** – Un procesador ocioso realiza un pull de una tarea en la cola de un procesador ocupado

Procesadores Multicore

- Recientemente la tendencia se centra en colocar múltiples cores en el mismo chip físico
- Se gana velocidad y se ahorra energía
- La cantidad de threads por core también se ha incrementado
  - Se toma ventaja de las perdidas de memoria (memory stall) para avanzar en otro thread mientras se recuperan los datos por ejemplo de la memoria principal

Multithreaded en Sistemas Multicore



Virtualización y Planificación

- El software de virtualización planifica múltiples sistemas guests en el CPU
- Cada guest realiza su propia planificación
  - Sin saber que no controla en realidad al CPU
  - Podría resultar en pobres tiempos de respuesta
- Es posible que los algoritmos de planificación en los guest sean infructuosos
- ¿Qué hacer?
  - Ideas

Ejemplos de Planificación en SO

- Planificación en Linux
- Planificación en Windows

Planificación en Linux hasta la Versión 2.5

- Kernels posteriores a la versión 2.5, ejecutaban una variación del algoritmo de planificación estándar UNIX
- El orden de ejecución en tiempo del planificador era O(1)
  - Apropiativo, basado en prioridades
  - Dos tipos de prioridades organizados en rangos: Tiempo compartido y Tiempo real
  - El rango de tiempo real oscilaba entre 0 y 99, mientras que el rango de tiempo compartido oscilaba en el complemento
  - Menores valores implican mayor prioridad
  - Tareas con alta prioridad se ejecutan quantums de tiempo más largos
  - Las tareas se ejecutarán el restante de tiempo de su ranura (activas)
  - Si la tarea ha agotado su ranura de tiempo, debe esperar que las demás tareas en el sistemas usen su ranura (espiradas)

Planificación en Linux hasta la Versión 2.5

- Todas las tareas capaces de ejecutarse están preasignadas a una cola de CPU
  - Se mantienen dos arreglos (activas y expiradas)
  - Las tareas están indexadas por prioridad
  - Cuando no hay más tareas activas, los arreglos son intercambiados
- Trabaja bastante bien, pero los procesos interactivos obtienen pobres tiempos de respuesta

Planificación en Linux- Versiones mayores a 2.6.23

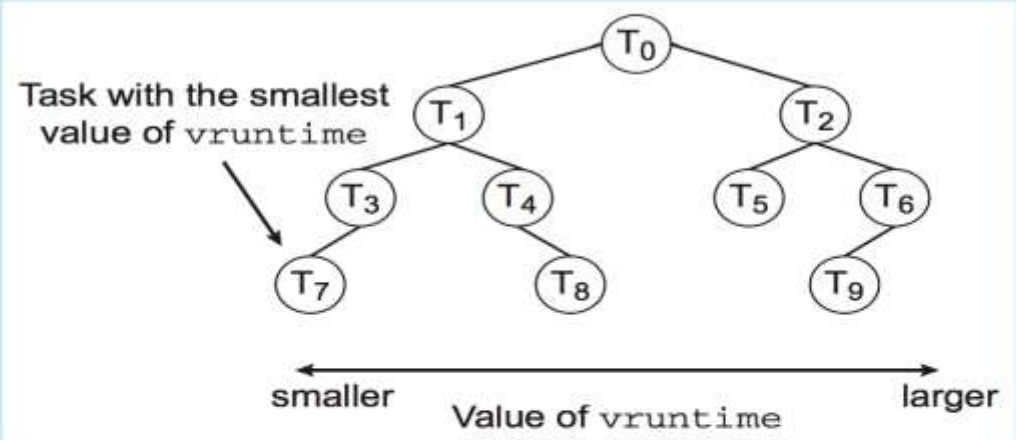
- Completely Fair Scheduler (CFS)
- Clases de planificación
  - Cada clase tiene una prioridad específica
  - El planificador elige la tarea de mayor prioridad en la clase de planificación más alta
  - En lugar de basarse en quantums de tiempo fijo, se basa en una proporción del tiempo de CPU
  - Se incluyen dos clases de planificación, y otras pueden ser agregadas
    - Por defecto
    - Tiempo real
  - En quantum se calcula en base a un nice value desde -20 a 19
    - El valor más bajo es la prioridad más alta
    - Se calcula una latencia objetivo – Intervalo de tiempo durante el cual la tarea debe ejecutarse al menos una vez
    - La latencia objetivo pueden incrementarse si se incrementa el número de tareas activas

Planificación en Linux- Versiones mayores a 2.6.23

- El planificador CFS mantiene por tarea una virtual run time en la variable vruntime
  - Asociado con un factor de decaimiento basado en la prioridad de la tarea – Una menor prioridad implica una mayor tasa de decaimiento
- Para decidir la próxima tarea a ejecutarse el planificador selecciona la tarea con el menor vruntime

Desempeño de CFS

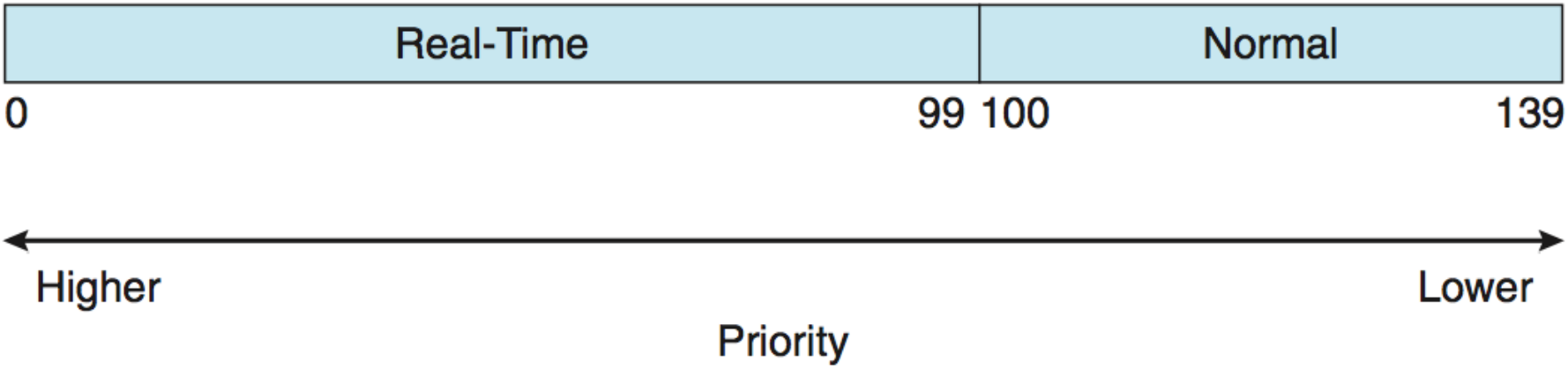
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Planificación en Linux

- La planificación de tiempo real se realiza de acuerdo a los establecido en POSIX. 1b
  - Las tareas de tiempo real tienen asignadas prioridades fijas
- Un nice value de -20 corresponde a una prioridad global de 100
- Un nice value de +19 corresponde a una prioridad de 139



Planificación en Windows

- Windows usa una planificador apropiativo basado en prioridades
- Entonces primero se ejecuta el thread con mayor prioridad
- El despachador es planificado
- Un thread se ejecuta durante:
  - Un bloque de instrucciones
  - Un porción de tiempo
  - Es desalojado por un thread con mayor prioridad
- Los threads de tiempo real pueden ser apropiados
  - ¿Entonces?
- El esquema de prioridades posee 32 niveles
  - Clase variable va del 1 al 15
  - Clase de tiempo real va del 16 al 31

Planificación en Windows

- Solo un thread en el sistema posee prioridad 0
  - El manejador de memoria
- Existe una cola por cada prioridad
- Si no existen threads en ejecución, entonces se ejecuta el thread idle
- El API de Win32 especifica diferentes clases de prioridades
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - Todas son variables excepto REALTIME
- Los hilos dentro de una clase poseen una prioridad relativa
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- La prioridad de la clase y la prioridad relativa se combinan para obtener una prioridad numérica

Planificación en Windows

- La prioridad base dentro de una clase es NORMAL
- Si el quantum expira, la prioridad decae, pero nunca estará por debajo de la prioridad base
- Si el thread se bloquea la prioridad aumenta dependiendo de por quién se esta esperando
- Las ventanas en primer plano tienen un factor de incremento de tres (3)
- Windows 7 agrega user-mode schedulling (UMS)
  - Los threads de aplicación se crean y manejan de forma independiente del kernel
  - Para una gran cantidad de threads, este enfoque es muy eficiente
    - ¿Por qué?
  - El planificado UMS se encuentra embebido en bibliotecas de programación, por ejemplo, en C++ Concurrent Runtime (ConcRT)

Prioridades en Windows

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1