Sistemas Operativos Avanzados

CC-571

César Lara Avila

Universidad Nacional de Ingeniería

(actualización: 2020-06-19)

Bienvenidos

Sesión 3

Virtualización

Temario de la sesión 3

- Introducción a la planificación
- La cola de retroalimentación de varios niveles (MLFQ)
- Cuota proporcional
- Planificación en un multiprocesador

Introducción a la planificación

Vocabulario

- Carga de trabajo: conjunto de descripciones de trabajo
- Planificador: lógica que decide cuándo se ejecutan los trabajos
- Métricas: medición de la calidad de la programación
- Álgebra del planificador, dadas 2 variables, encuentra una tercera:

$$f(W, S) = M$$

.

Ejemplo

```
W =  workload??

S = FIFO

M = tiempo de entrega alto.
```

Supuestos de carga de trabajo:

1. Cada trabajo se ejecuta en la misma cantidad de tiempo.
2. Todos los trabajos llegan al mismo tiempo.
3. Una vez iniciado, cada trabajo se ejecuta hasta su finalización.
4. Todos los trabajos solo usan la CPU (es decir, no realizan E/S).
5. Se conoce el tiempo de ejecución de cada trabajo.

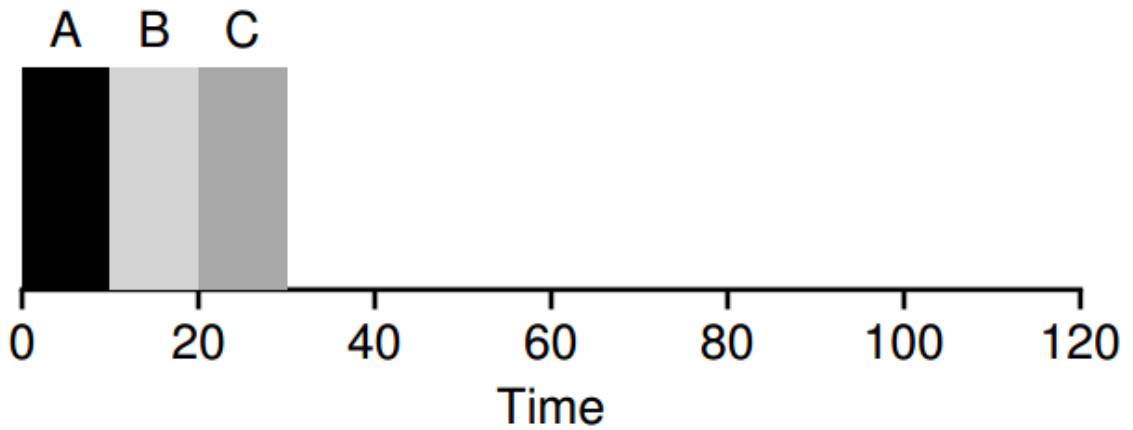
Métricas de planificación

- Métrica de rendimiento: tiempo de entrega- turnaround time (TAT)
  - El tiempo en la que se completa el trabajo menos el tiempo en que el trabajo llegó al sistema :

    $T_{\text{entrega}} = T_{\text{completación}} - T_{\text{arribo}}$
- Otra métrica es la equidad (Índice de equidad)
  - El rendimiento y la equidad a menudo están en desacuerdo en la planificación.

First In, First Out (FIFO)

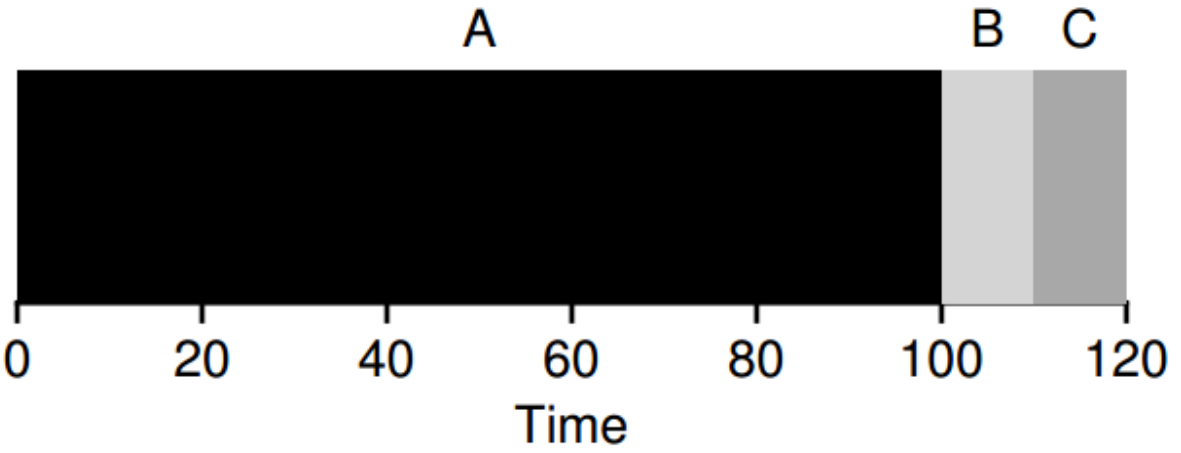
- Primero en llegado, primero en ser servido (FCFS)
  - Simple y fácil de implementar
- Ejemplo:
  - A llegó justo antes de B, que llegó justo antes de C.
  - Cada trabajo se ejecuta durante 10 segundos.



$T_{\text{entrega}} = \frac{10+20+30}{3} = 20\text{sec.}$

Problemas con FIFO - Efecto convoy

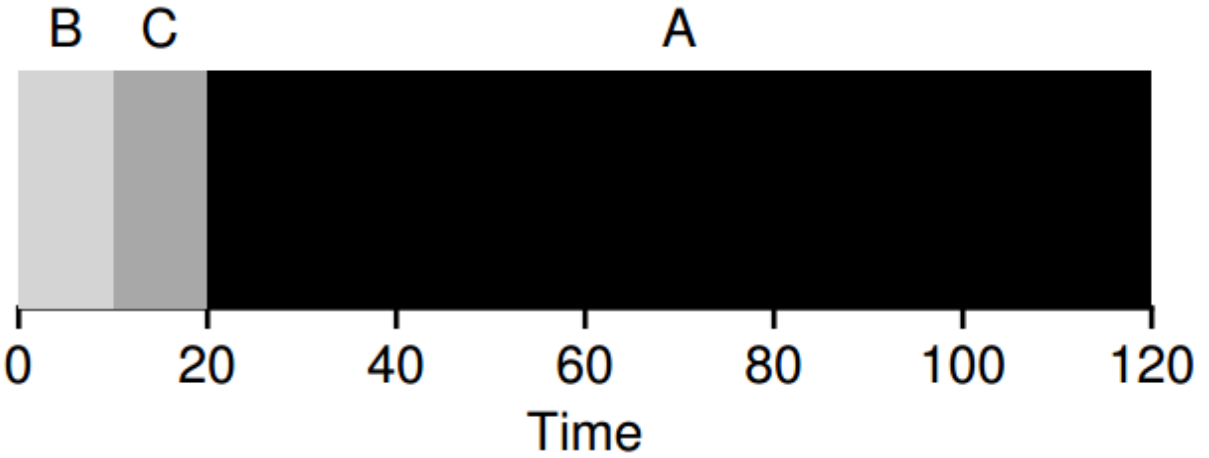
- Vamos a relajar el supuesto 1: cada trabajo ya no se ejecuta en la mismo cantidad de tiempo.
- Ejemplo:
  - A llegó justo antes de B, que llegó justo antes de C.
  - A corre por 100 segundos, B y C corren por 10 cada uno.



$T_{\text{entrega}} = \frac{110+110+120}{3} = 110$sec. --> los tiempos de entrega tienden a ser altos.

Shortest Job First (SJF)

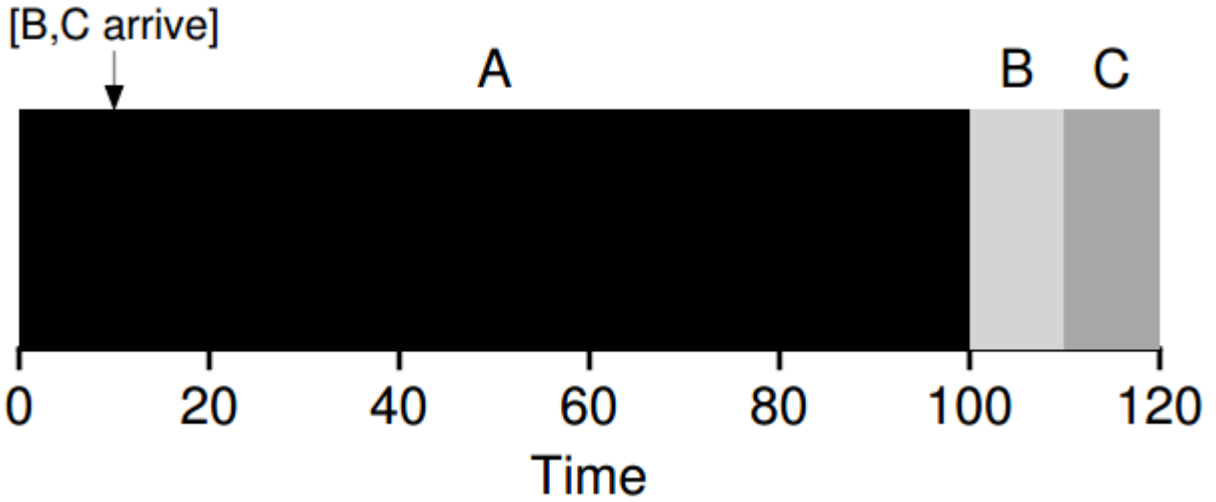
- Primero se ejecuta el trabajo más corto, luego el siguiente más , y así sucesivamente
  - Planificador no preventivo
- Ejemplo:
  - A llegó justo antes de B, que llegó justo antes de C.
  - A corre por 100 segundos, B y C corren por 10 cada uno.



$T_{\text{entrega}} = \frac{10+20+120}{3} = 50\text{sec.}$

SJF con llegadas tardías de B y C

- Relajemos la suposición 2: los trabajos pueden llegar en cualquier momento.
- Ejemplo:
  - A llega en $t = 0$ y necesita ejecutarse durante 100 segundos.
  - B y C llegan en $t = 10$ y cada uno necesita ejecutarse durante 10 segundos.



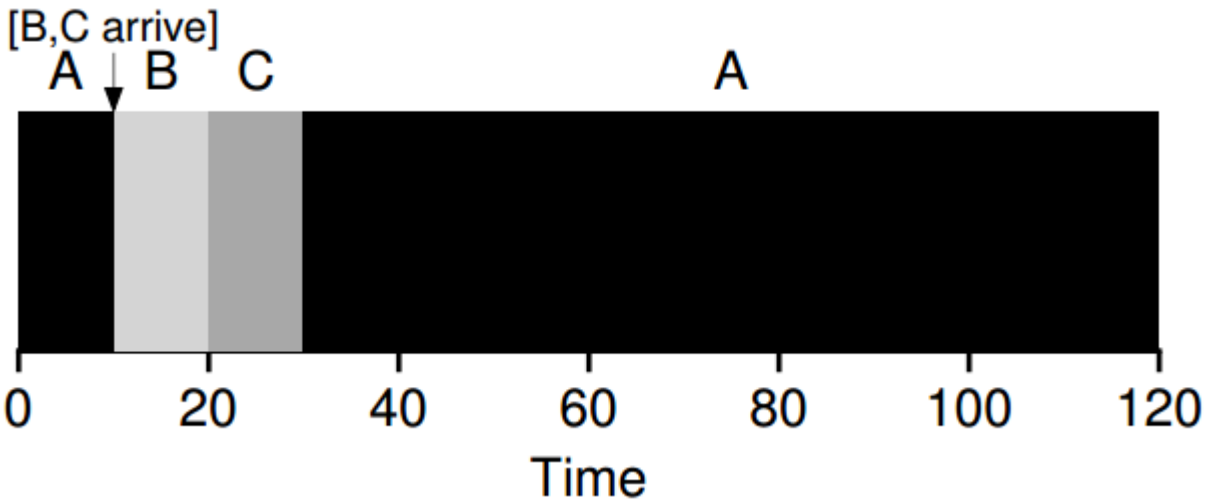
$T_{\text{entrega}} = \frac{100+(110-10)+(120-10)}{3} = 103.33\text{sec.}$

Shortest Time-to-Completion First (STCF)

- Añadir prevención a SJF
  - También conocido como Preemptive Shortest Job First (PSJF)
- Un nuevo trabajo ingresa al sistema:
  - Determina los trabajos restantes y el nuevo trabajo
  - Planifica el trabajo que le queda menos tiempo.

Shortest Time-to-Completion First (STCF)

- Relajamos el suposición 3: los trabajos deben ejecutarse hasta su finalización.
- Ejemplo:
  - A llega en $t = 0$ y necesita ejecutarse durante 100 segundos
  - B y C llegan en $t = 10$ y cada uno necesita ejecutarse durante 10 segundos



$T_{\text{entrega}} = \frac{(120-0)+(20-10)+(30-10)}{3} = 50\text{sec.}$

Nueva métrica de planificación: tiempo de respuesta

- El tiempo desde que llega el trabajo hasta la primera vez que está planificado.

  $T_{\text{respuesta}} = T_{\text{primer\_planificación}} - T_{\text{arribo}}$

- El STCF y las disciplinas relacionadas no son particularmente buenas para el tiempo de respuesta.

¿Cómo podemos construir un planificador que sea sensible al tiempo de respuesta?

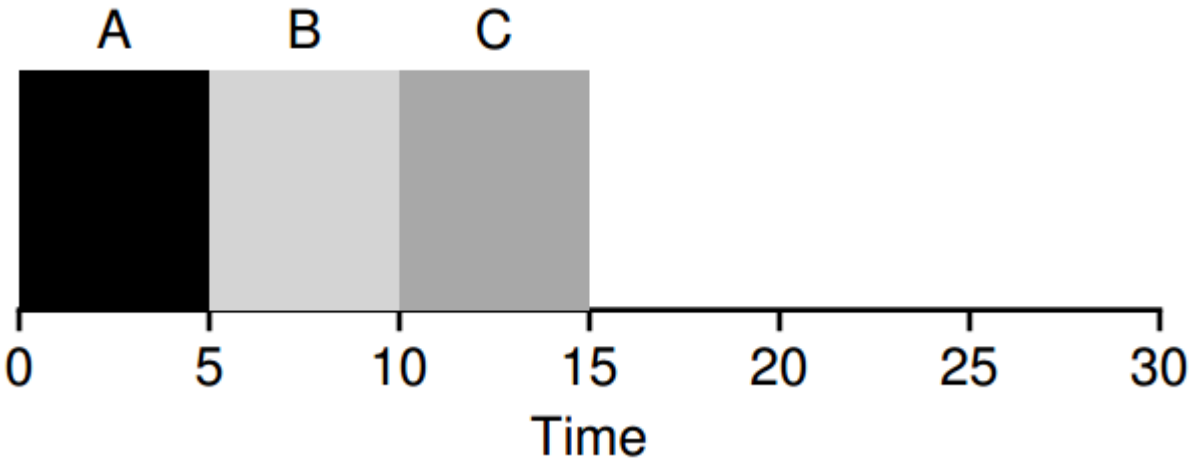
Round Robin (RR)

- Planificación de segmentos de tiempo
- Ejecuta un trabajo por un segmento de tiempo y luego cambia al siguiente trabajo en la cola de ejecución hasta que finalicen los trabajos.
- El segmento de tiempo a veces se denomina **scheduling quantum**.
- Lo hace repetidamente hasta que los trabajos hayan finalizado.
- La longitud de un segmento de tiempo debe ser un múltiplo del período de interrupción del temporizador.

RR es justo, pero funciona mal en métricas como el tiempo de entrega (TAT).

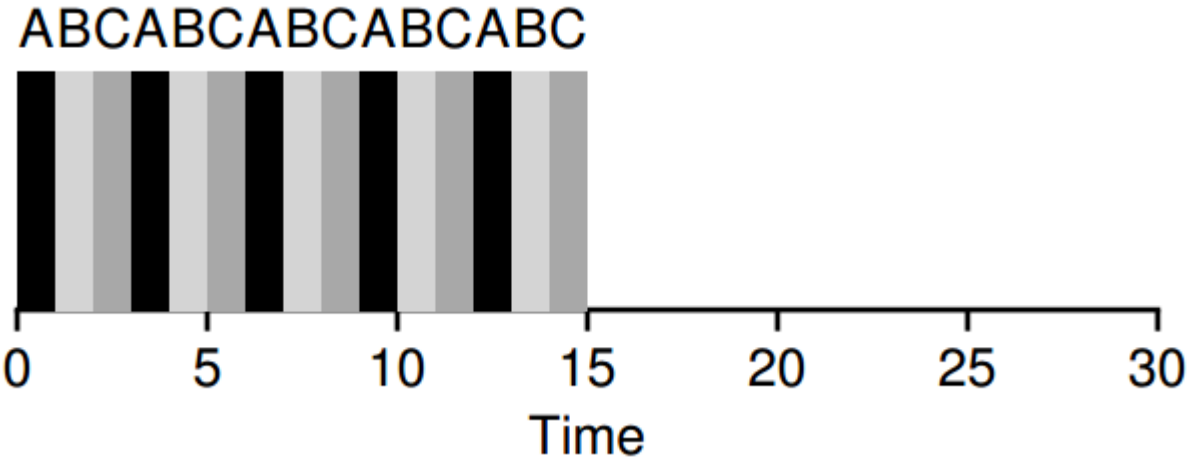
Ejemplo de Round Robin

- A, B y C llegan al mismo tiempo
- Cada uno de ellos desea correr durante 5 segundos



- SJF --> malo para el tiempo de respuesta.
- $T_{\text{respuesta promedio}} = \frac{0+5+10}{3} = 5\text{sec.}$

Continuación ...



- RR con un segmento de tiempo de 1 segundo -->bueno para el tiempo de respuesta.
- $T_{\text{respuesta promedio}} = \frac{0+1+2}{3} = 1\text{sec.}$

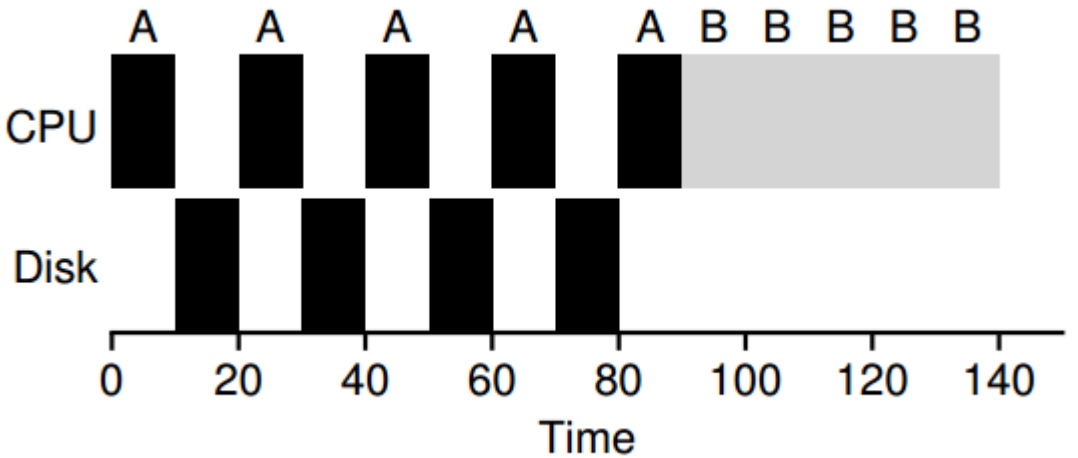
La duración del segmento de tiempo es crítica.

- Segmento de tiempo más corto
  - Mejor tiempo de respuesta
  - El costo del cambio de contexto dominará el rendimiento general.
- Segmento de tiempo más largo
  - Amortizar el costo de cambio de contexto
  - Peor tiempo de respuesta.

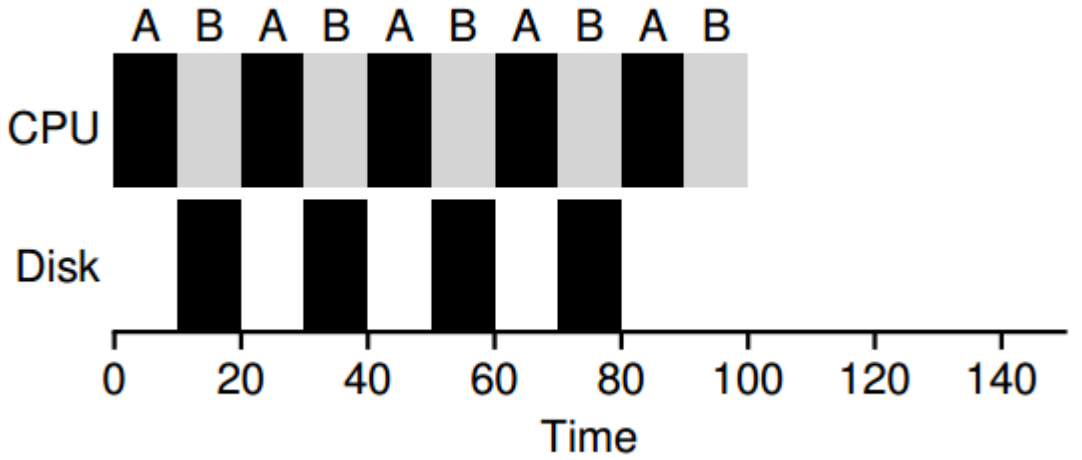
Decidir la duración del segmento de tiempo presenta una tarea al diseñador de sistemas.

Incorporando E/S

- Relajemos el supuesto 4: todos los programas realizan E/S
- Ejemplo:
  - A y B necesitan 50 ms de tiempo de CPU cada uno.
  - A se ejecuta durante 10 ms y luego emite una solicitud de E/S
  - E/S cada uno toma 10 ms
  - B simplemente usa la CPU durante 50 ms y no realiza E/S
  - El planificador ejecuta A primero, luego B después.



Continuación...



- La superposición permite un mejor uso de los recursos.

Incorporación de E/S

- Cuando un trabajo inicia una solicitud de E/S.
  - El trabajo está bloqueado esperando la finalización de E/S
  - El planificador debe programar otro trabajo en la CPU.
- Cuando se completa la E/S
  - Se levanta una interrupción
  - El OS mueve el proceso de bloqueado al estado listo.

La cola de retroalimentación de varios niveles (MLFQ)

- Un planificador que aprende del pasado para predecir el futuro.
- Objetivo:
  - Optimice el tiempo de entrega --> Ejecuta primero los trabajos más cortos.
  - Minimizar el tiempo de respuesta sin un conocimiento previo de la duración del trabajo.

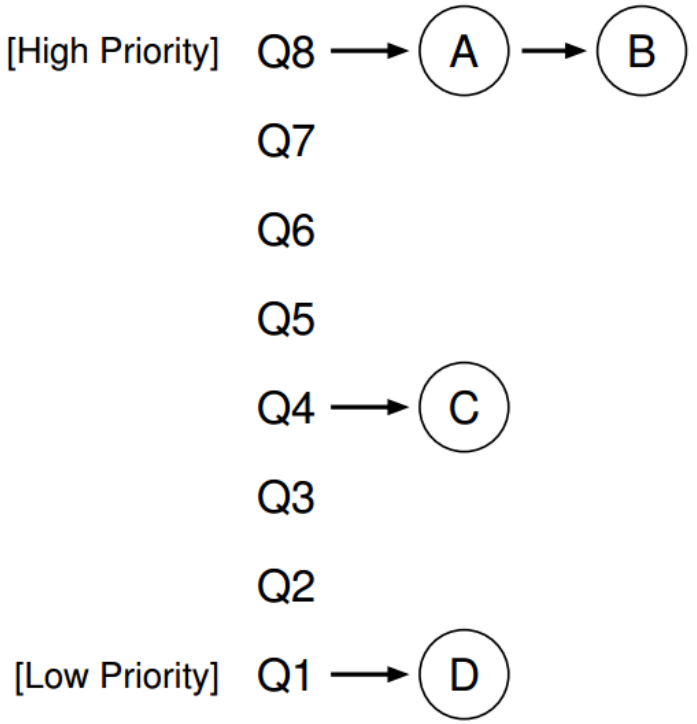
Reglas básicas del MLFQ

- MLFQ tiene varias colas distintas.
  - A cada cola se le asigna un nivel de prioridad diferente.
- Un trabajo que está listo para ejecutarse está en una sola cola.
  - Se elige ejecutar un trabajo en una cola superior.
  - Se utiliza la planificación round-robin entre trabajos en la misma cola.
- Regla 1: Si Prioridad (A)> Prioridad (B), A se ejecuta (B no).
- Regla 2: si Prioridad (A) = Prioridad (B), A y B se ejecutan en RR.

Reglas básicas del MLFQ

- MLFQ varía la prioridad de un trabajo en función de su comportamiento observado.
- Ejemplo:
  - Un trabajo renuncia repetidamente a la CPU mientras espera el E/S --> mantiene alta su prioridad
  - Un trabajo utiliza la CPU de forma intensiva durante largos períodos de tiempo --> reduce su prioridad.

Ejemplo de MLFQ



- MLFQ intentará aprender sobre los procesos a medida que se ejecutan y, por lo tanto, usará el historial del trabajo para predecir su comportamiento futuro.

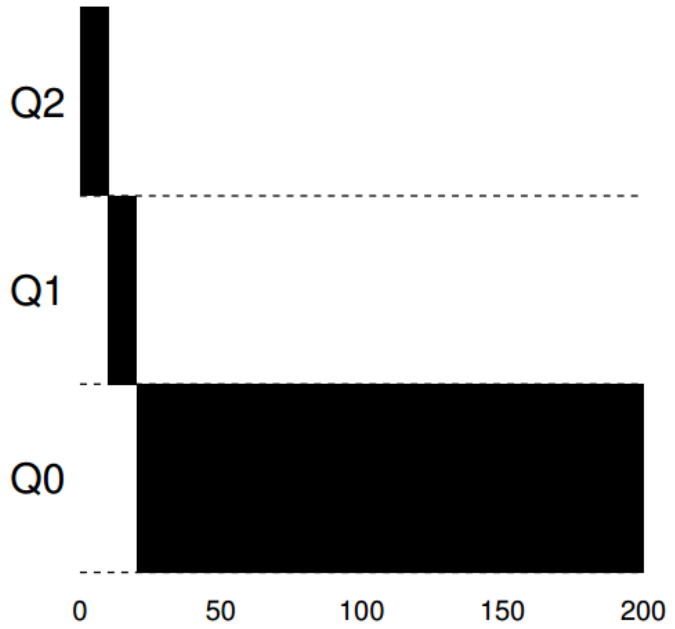
MLFQ: Cómo cambiar la prioridad

- Algoritmo de ajuste de prioridad MLFQ:
  - Regla 3: cuando un trabajo ingresa al sistema, se le asigna la máxima prioridad.
  - Regla 4a: Si un trabajo usa un segmento de tiempo completo mientras se ejecuta, su prioridad se reduce (es decir, se mueve hacia abajo en la cola).
  - Regla 4b: si un trabajo abandona la CPU antes de que se agote el segmento de tiempo, permanece en el mismo nivel de prioridad.

De esta manera, MLFQ se aproxima a SJF.

Ejemplo 1: un solo trabajo de larga duración

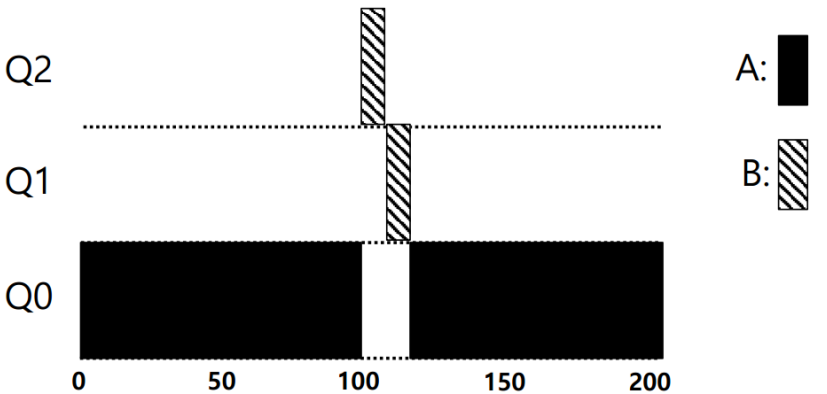
- Suposiciones:
  - Un planificador de tres colas con un intervalo de tiempo de 10 ms.



- Parece simple?

Ejemplo 2: Junto a un trabajo largo vino un trabajo corto

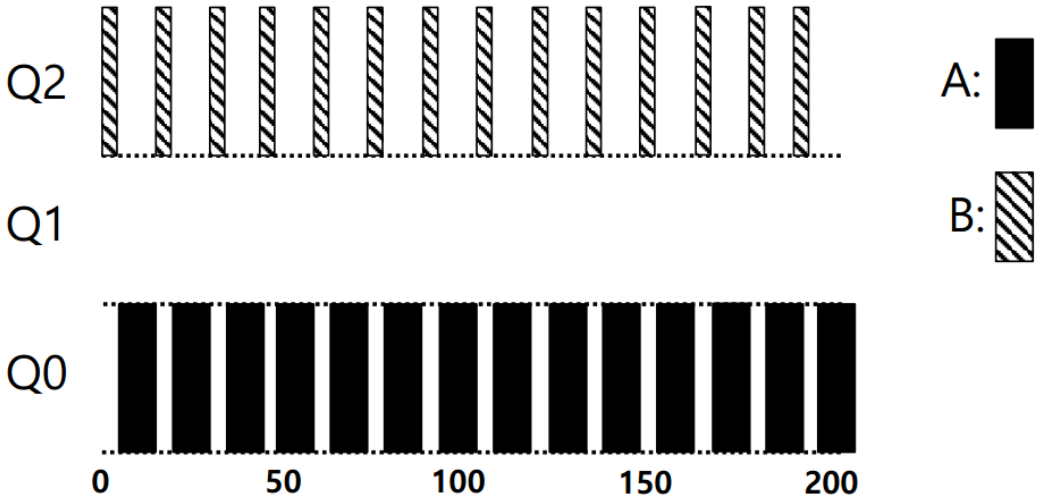
- Suposiciones:
  - Trabajo A: un trabajo de uso intensivo de CPU de larga duración
  - Trabajo B: un trabajo interactivo de corta duración (tiempo de ejecución de 20 ms)
  - A ha estado funcionando durante algún tiempo, y luego B llega al tiempo $T = 100$.



- En el algoritmo, dado que no sabe si un trabajo será un trabajo corto o un trabajo de largo, primero supone que podría ser un trabajo corto.

Ejemplo 3: ¿Qué pasa con las E/S?

- Suposiciones:
  - Trabajo A: un trabajo de uso intensivo de CPU de larga duración
  - Trabajo B: un trabajo interactivo que necesita la CPU solo durante 1ms antes de realizar un E/S.



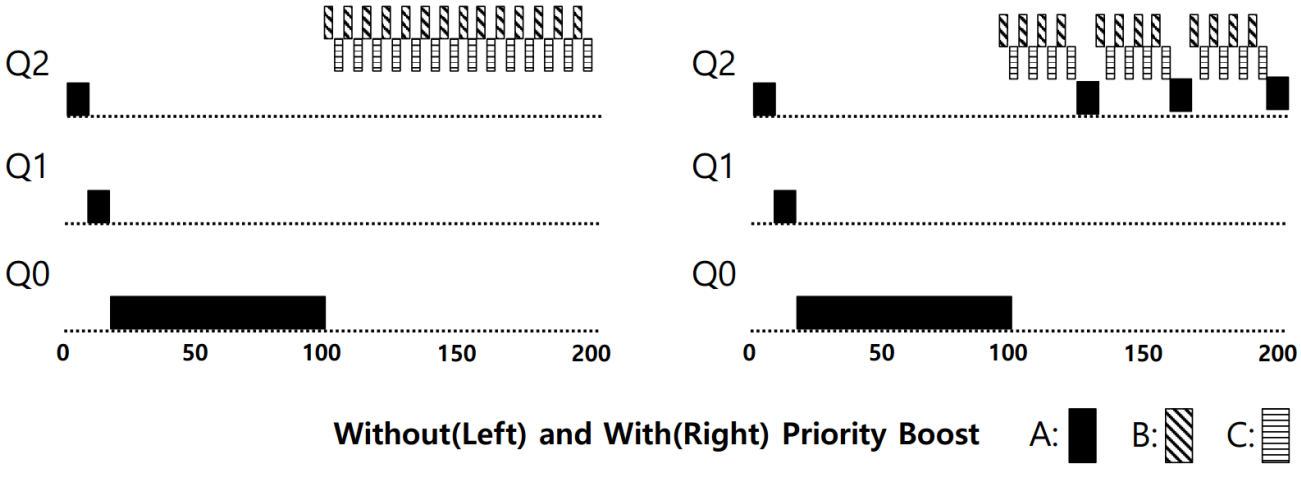
- El enfoque MLFQ mantiene un trabajo interactivo con la máxima prioridad.

Problemas con el MLFQ básico

- Starvation
  - Si hay "demasiados" trabajos interactivos en el sistema.
  - Los trabajos de larga duración nunca recibirán tiempo de CPU.
- Juego del planificador
  - Después de ejecutar el 99% de un segmento de tiempo, se emite una operación de E/S.
  - El trabajo gana un mayor porcentaje de tiempo de CPU.
- Un programa puede cambiar su comportamiento con el tiempo.
  - Proceso vinculado a la CPU --> proceso vinculado al E/S

El aumento de prioridad

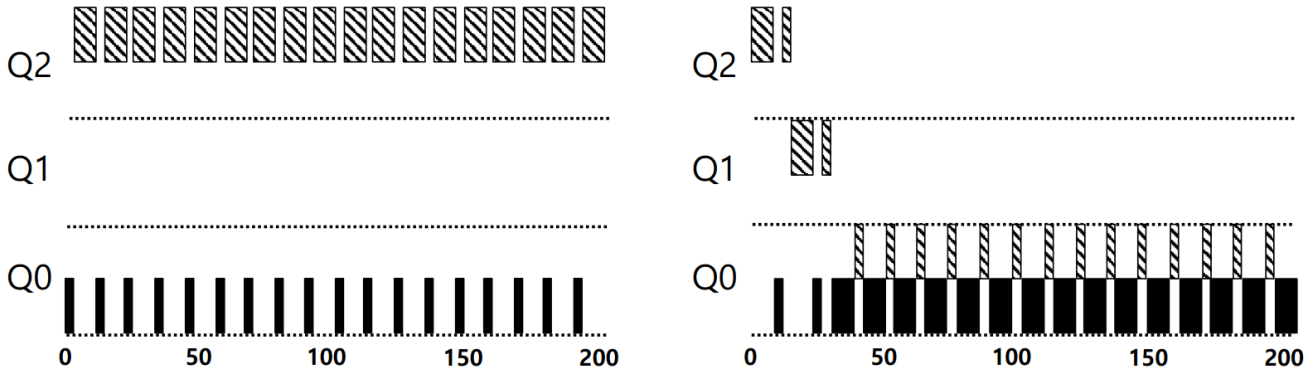
- Regla 5: Después de un período de tiempo S, mueva todos los trabajos del sistema a la cola superior.
- Ejemplo:
  - Un trabajo de larga duración (A) con dos trabajos interactivos de corta duración (B, C)



  - Al menos menos se garantiza que el trabajo de larga duración progresará un poco, siendo impulsado a la máxima prioridad cada 50 ms y, por lo tanto, ejecutarse periódicamente.

Mejor contabilidad

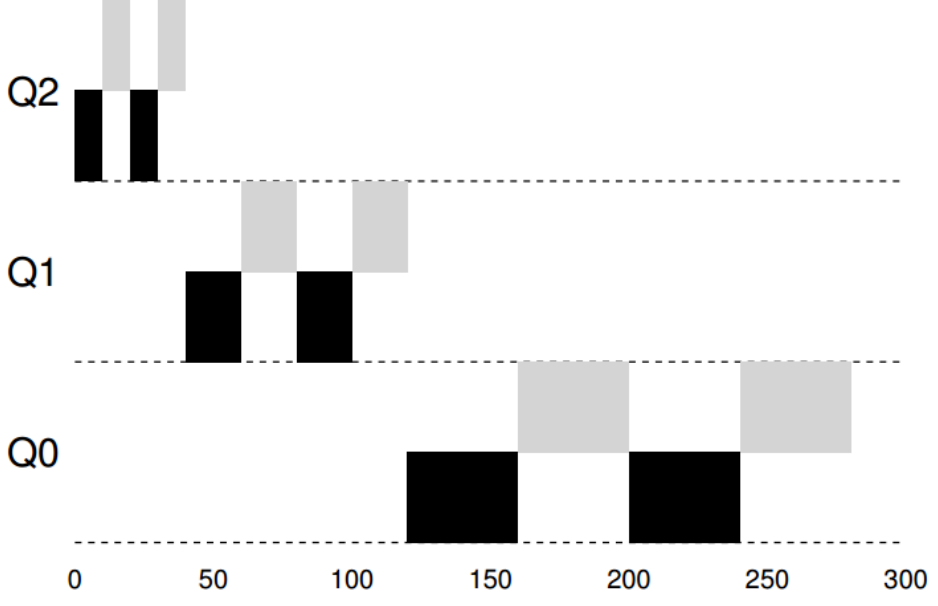
- ¿Cómo evitar los juegos del planificador?
- Solución:
  - Regla 4 (reescribe las reglas 4a y 4b): una vez que un trabajo agota su asignación de tiempo en un nivel determinado (independientemente de cuántas veces haya abandonado la CPU), su prioridad se reduce (es decir, baja en la cola).



- No se puede obtener una parte injusta de la CPU.

Ajustes del MLFQ y otros problemas

- Las colas de alta prioridad --> Segmentos de tiempo pequeños
  - Por ejemplo, 10 o menos milisegundos
- La cola de baja prioridad --> Segmentos de tiempo grandes
  - Por ejemplo, 100 milisegundos



- Prioridad más baja, quanta más grande.

Cuota proporcional

- Planificador de reparto justo
  - Garantiza que cada trabajo obtenga un cierto porcentaje de tiempo de CPU
  - No optimizado para tiempo de entrega (TAT) o respuesta
- Un ejemplo es la planificación de lotería mencionado por Waldspurger y Weihl en el paper: **Lottery Scheduling: Flexible Proportional-Share Resource Management**.

Conceptos basicos

- Tickets
  - Representan la parte de un recurso que un proceso (o usuario o lo que sea) debería recibir
  - El porcentaje de tickets representa su parte del recurso del sistema en cuestión.
- Ejemplo
  - Hay dos procesos, A y B:
    - El proceso A tiene 75 tickets --> recibe el 75% de la CPU
    - El proceso B tiene 25 tickets --> recibe el 25% de la CPU

Planificador de lotería

- El planificador elige un boleto ganador
  - Carga el estado de ese proceso ganador y lo ejecuta.
- Ejemplo
  - Hay 100 entradas
    - El proceso A tiene 75 tickets: 0 -74
    - El proceso B tiene 25 tickets: 75 -99

  Tickets ganadores del planificador: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63

  Planificador resultante: A B A A B A A A A A A B A B A
- Cuánto más tiempo compitan estos dos trabajos, más probabilidades hay de que alcancen los porcentajes deseados.

Mecanismos de tickets

- Ticket currency
  - Un usuario asigna tickets entre sus propios trabajos en la moneda que desee
  - El sistema convierte la moneda en el valor global correcto
  - Ejemplo
    - Hay 200 boletos (moneda global)
    - El proceso A tiene 100 boletos
    - El proceso B tiene 100 boletos
    - El proceso A ejecuta dos trabajos A1 y A2
    - El proceso B ejecuta un solo trabajo B1

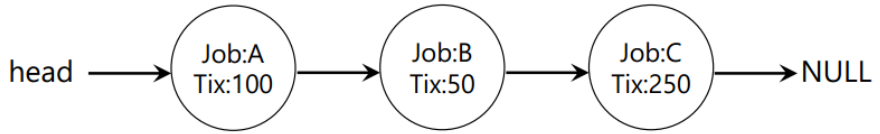
```
Usuario A --> 500 (monedas de A) a A1 --> 50 (moneda global)
          --> 500 (monedas de A) a A2 --> 50 (moneda global)

Usuario B --> 10 (monedas de B) a B1 --> 100 (moneda global)
```

Mecanismo de tickets

- Transferencia de boletos
  - Un proceso puede transferir temporalmente sus tickets a otro proceso. Este mecanismo es útil cuando se esta utilizando la arquitectura cliente/servidor.
- Inflación de tickets
  - Un proceso puede aumentar o disminuir temporalmente la cantidad de tickets que posee
  - Si algún proceso necesita más tiempo de CPU, puede aumentar sus tickets.
  - Este método puede ser utilizado en un ambiente en el cual los procesos confían entre ellos.

Implementación

- Ejemplo: hay procesos, A, B y C en una lista:




```
1         // counter: used to track if we've found the winner yet
2         int counter = 0;
3
4         // winner: use some call to a random number generator to
5         // get a value, between 0 and the total # of tickets
6         int winner = getrandom(0, totaltickets);
7
8         // current: use this to walk through the list of jobs
9         node_t *current = head;
10
11        // loop until the sum of ticket values is > the winner
12        while (current) {
13                  counter = counter + current->tickets;
14                  if (counter > winner)
15                            break; // found the winner
16                  current = current->next;
17        }
18        // 'current' is the winner: schedule it...
```

- Utilizamos una lista, con un simple contador para ayudarnos a encontrar al ganador.

Métricas para el mecanismo de tickets

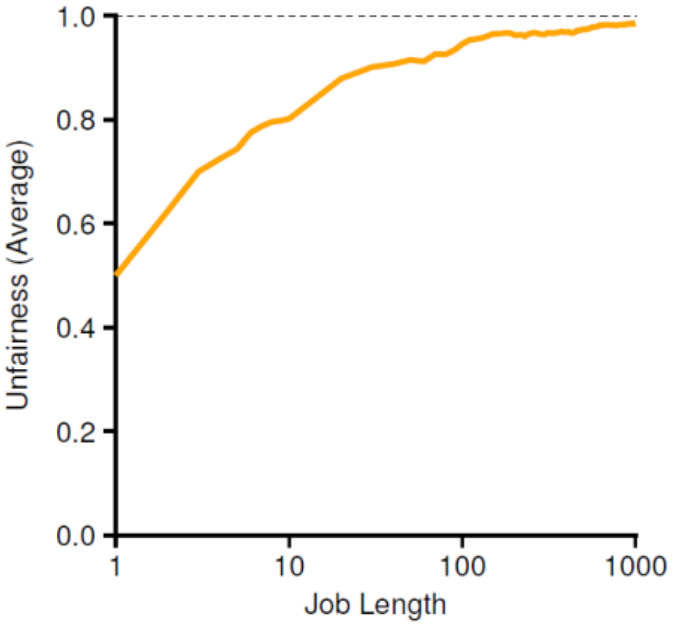
- U: métrica de inequidad
  - El tiempo en que se completa el primer trabajo dividido por el tiempo en que se completa el segundo trabajo
- Ejemplo:
  - Hay dos trabajos, cada trabajo tiene tiempo de ejecución 10:
    - El primer trabajo termina en el tiempo 10
    - El segundo trabajo termina en el tiempo 20

$$U = \frac{10}{20} = 0.5$$

- U estará cerca de 1 cuando ambos trabajos terminen casi al mismo tiempo.

Estudio de equidad en la planificación de loteria

- Hay dos trabajos
  - Cada trabajo tiene el mismo número de tickets (100)



- Cuando la duración del trabajo no es muy larga, la inequidad promedio puede ser bastante mala.

Strides

- Stride de cada proceso
  - (Un gran número)/(el número de tickets del proceso)
  - Ejemplo: un número grande = $10,000$
    - El proceso A tiene 100 tickets --> el stride de A es 100
    - El proceso B tiene 50 tickets --> el stride de B es 200
- Se ejecuta un proceso, se incrementa un contador (valor de pase) para él por su stride.

```
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
```

  - Se escoge el proceso a ejecutar que tenga el valor de pase más bajo.

Ejemplo de planificación con strides

- Sean trabajos A, B y C, con 100, 50 y 250 tickets, respectivamente, el stride para A, B y C es 100, 200 y 40.

Pass(A) (stride=100)	**Pass(B)** (stride=200)	**Pass(C)** (stride=40)	**Who Runs?**
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	...

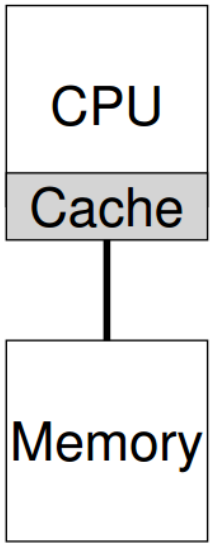
- Si ingresa un nuevo trabajo con el valor de pase 0 y ¡monopolizará la CPU!.

Planificación en un multiprocesador

- El aumento del procesador multicore es la fuente de la proliferación de la planificación multiprocesador.
    - Multicore: múltiples kernel de CPU se empaquetan en un solo chip.
- Agregar más CPU no hace que esa única aplicación se ejecute más rápido.
    - Tendrás que volver a escribir la aplicación para que se ejecute en paralelo, utilizando hilos.

¿Cómo planificar trabajos en múltiples CPU?

Única CPU con Caché



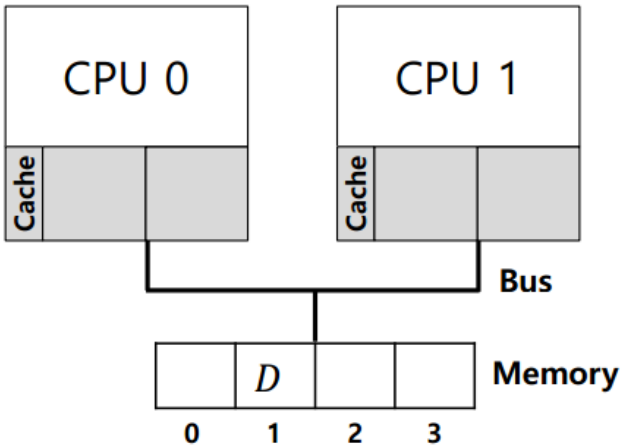
- Caché
  - Memoria rápida y pequeña.
  - Mantiene copias de datos que se encuentran en la memoria principal.
  - Utilizar la localidad temporal y espacial.
- Memoria principal
  - Contiene todos los datos.
  - El acceso a la memoria principal es más lento que en el caché.

Al mantener los datos en caché, el sistema puede hacer que la memoria lenta parezca rápida.

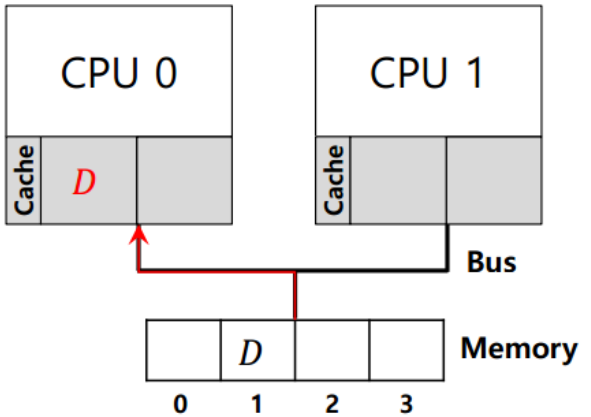
Coherencia de caché

- Consistencia de los datos de recursos compartidos almacenados en múltiples cachés.

1 . Dos CPU con cachés que comparten memoria

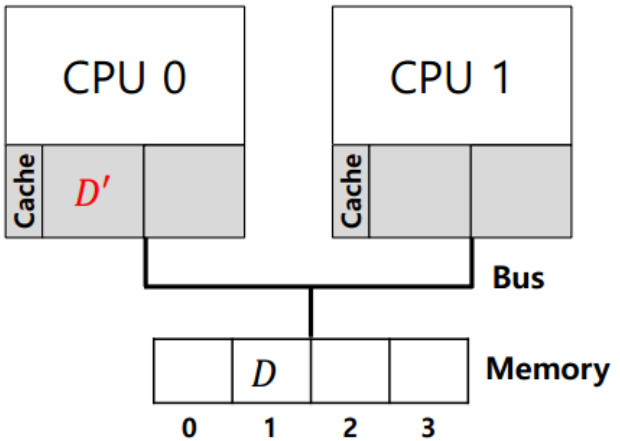


2 . CPU 0 lee datos en la dirección 1

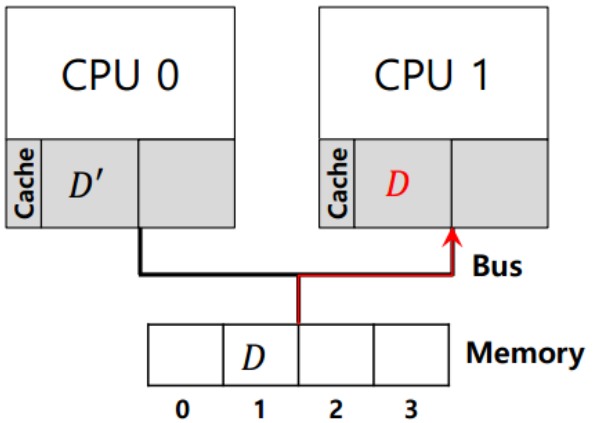


Coherencia de caché

3 . D se actualiza y CPU 1 es programada.



4 . CPU 1 vuelve a leer en la dirección A.



CPU 1 obtiene el valor anterior D en lugar del valor correcto D'.

Solución a la coherencia de caché

- **Bus snooping**
  - Cada caché presta atención a las actualizaciones de memoria al observar el bus.
  - Cuando una CPU ve una actualización para un elemento de datos que tiene en su caché, notará el cambio e *invalidará* su copia o la *actualizará*.

No olvides la sincronización

- Al acceder a los datos compartidos entre las CPU, es probable que se usen primitivas de exclusión mutua para garantizar la corrección. El siguiente código se usa para eliminar un elemento de una lista vinculada compartida.

```
1         typedef struct __Node_t {
2                 int value;
3                 struct __Node_t *next;
4         } Node_t;
5
6         int List_Pop() {
7                 Node_t *tmp = head;           // remember old head ...
8                 int value = head->value;      // ... and its value
9                 head = head->next;            // advance head to next pointer
10                free(tmp);                    // free old head
11                return value;                 // return value at head
12        }
```

Pregunta

- Explica el código y dar una solución al problema.

Solución

```
1         pthread_mtuex_t m;
2         typedef struct __Node_t {
3                   int value;
4                   struct __Node_t *next;
5         } Node_t;
6
7         int List_Pop() {
8                   lock(&m)
9                   Node_t *tmp = head;           // remember old head ...
10                  int value = head->value;      // ... and its value
11                  head = head->next;            // advance head to next pointer
12                  free(tmp);                    // free old head
13                  unlock(&m)
14                  return value;                 // return value at head
15        }
```

- Se corrige esas rutinas mediante el bloqueo (locking).

Afinidad de caché

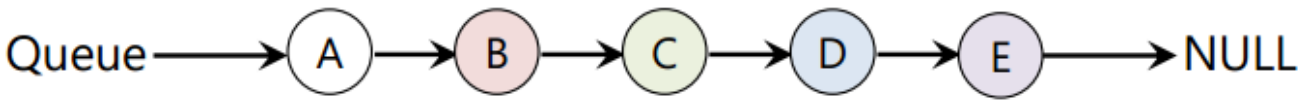
- Sigue un proceso en la misma CPU si es posible.
  - Un proceso acumula un poco de estado en el caché de una CPU.
  - La próxima vez que se ejecute el proceso, se ejecutará más rápido si parte de su estado ya está presente en la memoria caché de esa CPU.

A tomar en cuenta:

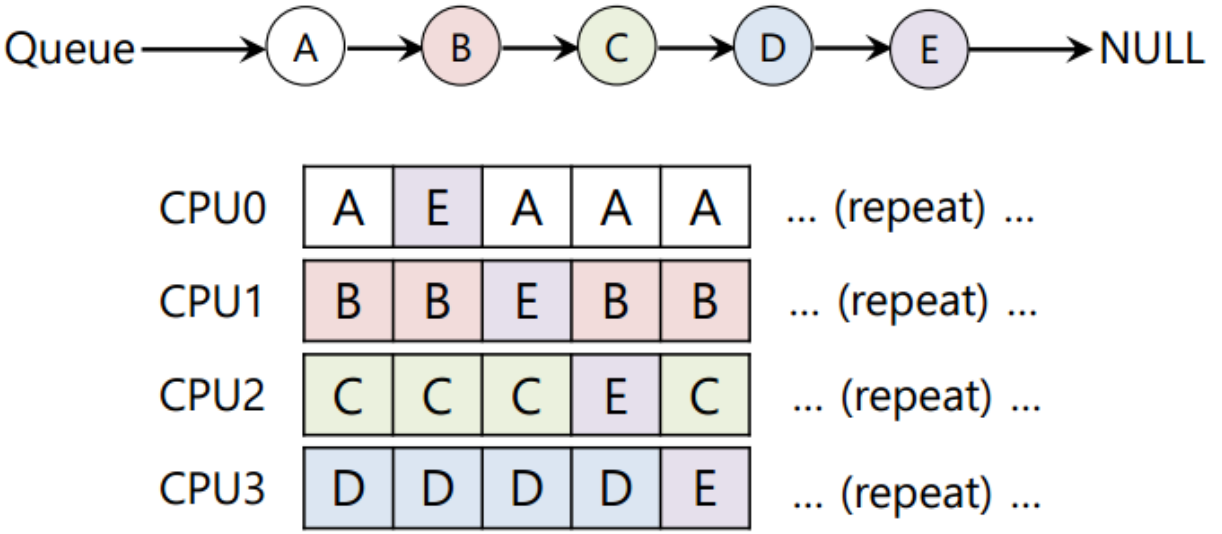
Un planificador multiprocesador debe considerar la afinidad de caché al tomar su decisión de planificación.

Planificador multiprocesador de cola única (SQMS)

- Coloca todos los trabajos que deben programarse en una sola cola.
  - Cada CPU simplemente elige el siguiente trabajo de la cola compartida globalmente.
  - Desventajas:
    - Se debe insertar alguna forma de bloqueo --> Falta de escalabilidad
    - Afinidad de caché
  - Ejemplo



Ejemplo de planificación con afinidad de caché



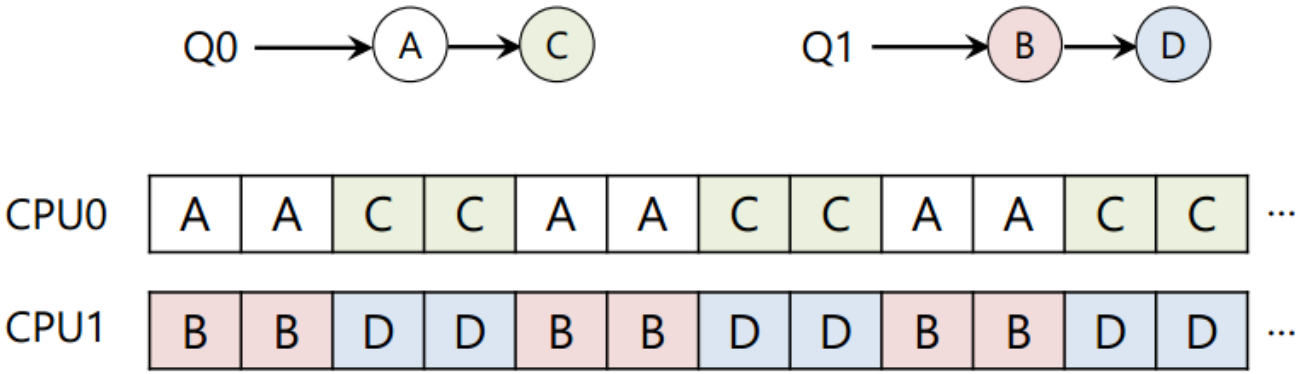
- Los trabajos A a D no se mueven a través de procesadores, solo el trabajo E migrando de CPU a CPU, preservando así la afinidad para la mayoría.
- Implementar tal esquema puede ser complejo.

Planificador multiprocesador de múltiples colas (MQMS)

- MQMS consta de múltiples colas de planificación.
- Cada cola seguirá una disciplina de planificación particular.
- Cuando un trabajo ingresa al sistema, se coloca exactamente en una cola de planificación.
- Evita los problemas de intercambio de información y sincronización.

Ejemplo MQMS

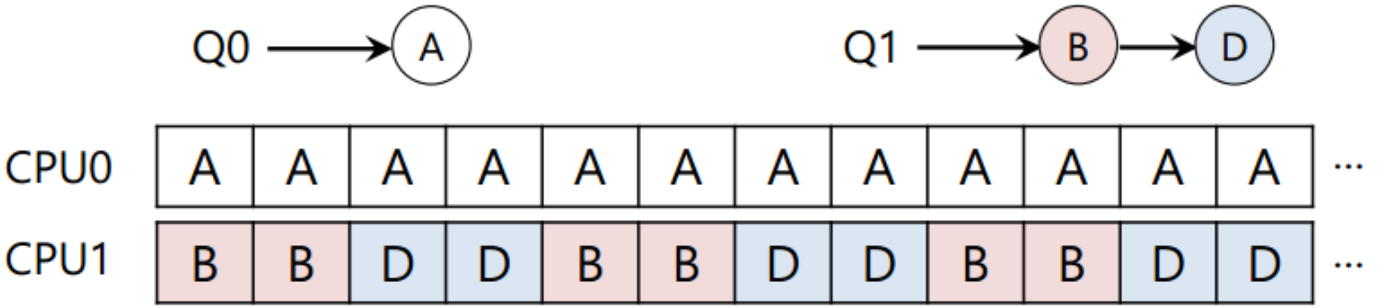
- Con round robin, el sistema puede generar una planificación similar al siguiente:



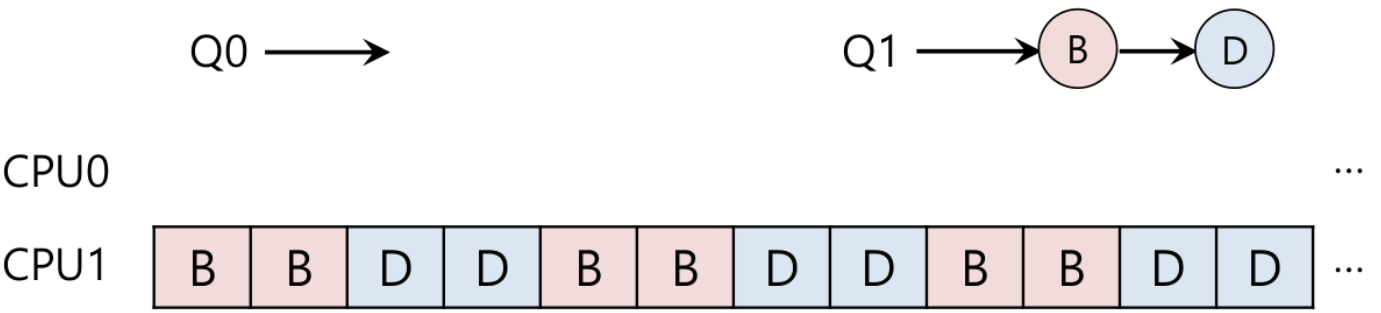
- MQMS proporciona más escalabilidad y afinidad de caché.

Problema de desbalanceo de carga de MQMS

- Después de que termine el trabajo C en Q0 --> A obtiene el doble de CPU que B y D:



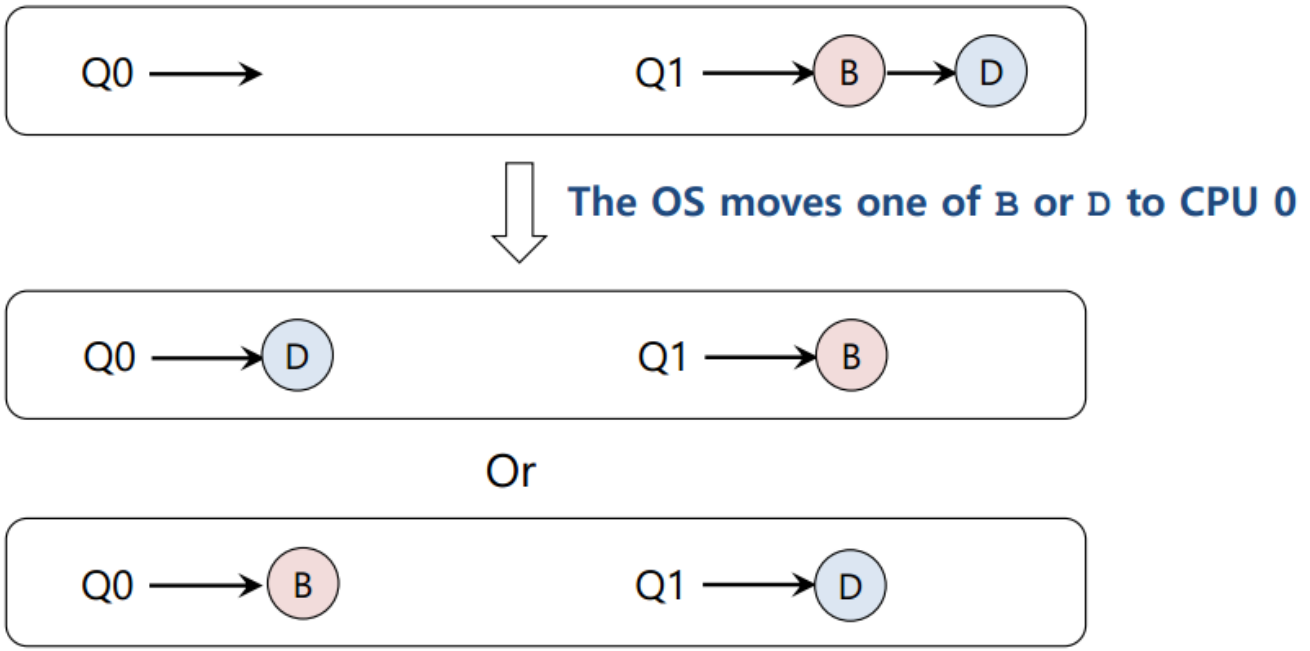
- Después de que termina el trabajo A en Q0 --> CPU 0 quedará inactivo!:



¿Cómo lidiar con el desbalanceo de carga?

- La respuesta es mover los trabajos (migración).

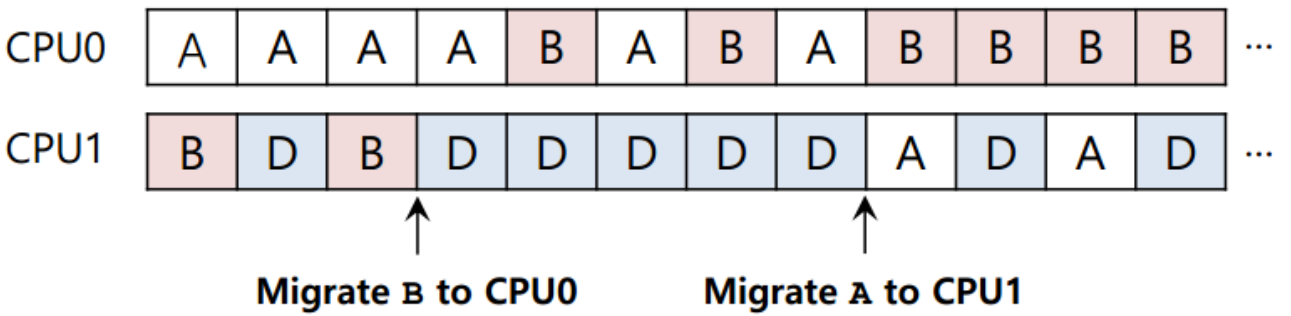
Ejemplo



¿Cómo lidiar con el desbalanceo de carga?

- Un caso más complicado:

- Un posible patrón de migración:
  - Seguir cambiando de trabajo



Work Stealing

- Mueve trabajos entre colas
  - Implementación:
  - Se selecciona una cola de origen que tiene pocos trabajos.
  - La cola de origen ocasionalmente se asoma a otra cola de destino
  - Si la cola de destino está más llena que la cola de origen, la fuente "robará" uno o más trabajos de la cola de destino.
- Contras:
  - Elevados gastos generales y problemas de escalamiento.

Planificadores multiprocesador de Linux

- **O(1)**
  - Un planificador basado en prioridades
  - Usa múltiples colas
  - Cambia la prioridad de un proceso a lo largo del tiempo
  - Programa aquellos con mayor prioridad
  - La interactividad es un enfoque particular.
- **Planificador completamente justo (CFS)**
  - Enfoque determinista de participación proporcional
  - Múltiples colas
- **Planificador BF (BFS)**
  - Un enfoque de cola única
  - Cuota proporcional
  - Basado en el Earliest Eligible Virtual Deadline First (EEVDF).

Fin!

[1] Algunos de los gráficos de esta presentación se basan en el texto **Operating Systems: Three Easy Pieces** de Remzi H. Arpaci-Dusseau y Andrea C. Arpaci-Dusseau.

[2] Algunos de los gráficos de esta presentación se basan en las notas de Youjip Won, Hanyang University, Embedded Software Systems Laboratory.