**EJERCICIOS DE FINAL - SCHEDULING**

**¿ Qué es un context switch? En un context switch cuales de las siguientes cosas no deben/deben ser guardadas y porque:**

**a. Registros de proposito general**

**b. TLB (Translation lookaside buffer)**

**c. Program Counter**

**d. Page directory entry**

**e. Pcb entry**

**f. Ninguna**

Un context switch es cuando el scheduler cambia un proceso por otro, consiste en guardarse todo lo necesario para poder en el futuro volver a poner en marcha el proceso en el mismo estado en el cual fue desalojado, este proceso tiene un contexto asociado los registros del procesador por ejemplo, cuando el proceso vuelva a correr necesitará volver a reanudar desde su estado anterior, si no se guardaran los registros y por ejemplo el scheduler justo decidió parar el proceso en mitad de una cuenta y después otro proceso pisa los registros con otros valores entonces se perderá la cuenta parcial e incluso será inconsistente, por eso es importante guardar el contexto viejo. Como mencionamos anteriormente, los registros de propósito general sería interesante guardarlos dentro del contexto dado que representa parcialmente el estado del proceso, la TLB sería interesante tenerla en el contexto dado que cuando el proceso vuelva a ejecutar tendríamos el beneficio de ya tener cargadas en caché el estado parcial y aprovechando los principios de localidad temporal y espacial probablemente esto resultaría en una mejor tasa de hits. Ahora considerando que habría que leer toda la TLB (y que además esta puede ser de varios niveles) y que luego habría que volver a cargar la TLB para cuando se restaure el contexto quizá esto haga el proceso de context switch muy lento, recordemos que queremos que el context switch sea rápido, sino afectamos las métricas de response time y turnaround, además cabe destacar que la TLB es algo interno de la MMU, **no hay un “store” en TLB**. Program counter es importante guardarlo en el contexto porque necesitamos saber en qué parte del código nos quedamos para luego poder reanudarlo desde ese mismo punto cuando el scheduler vuelva a seleccionar al proceso para ejecutarse. El page directory entry (la page table asociada al proceso) si es necesaria dado que cada proceso tiene su address space y su información de traducción está allí. Si es necesario guardar el PCB dado que es en donde se guarda todo lo anterior. Ninguna no es una opción porque perderíamos todo el progreso del proceso al cambiar entre procesos.

**Explique la política de scheduling MLFQ detalladamente. Sea 1 proceso, cuyo tiempo de ejecución total es de 40 ms, el time slice por cola es de 2ms/c pero el mismo se incrementa en 5ms por cola. Cuantas veces se interrumpe y en qué cola termina su ejecución.**

La politica de scheduling MLFQ (Multilevel feedback queue) consta de tener múltiples colas cada una con distinto nivel de prioridad, y se rigen las siguientes reglas/políticas:

* Si un proceso A está en una cola de mayor prioridad que B, se ejecuta A.
* Si la prioridad de A es igual de B, se aplica round robin con estos procesos.
* Cuando una tarea entra al sistema entra con la mayor prioridad, en caso de ser un proceso corto se ejecuta y sale del sistema rápidamente.
* Una vez que una tarea usó su asignación de tiempo su prioridad se reduce independientemente de cuántas veces haya renunciado al uso de CPU. Esto impide que un proceso ventajee (gaming) llamando a un I-O para relegar CPU y quedarse siempre en una prioridad alta, dado que aunque ceda CPU va a terminar bajando de prioridad.
* Si una tarea no usó toda su asignación de tiempo de CPU esta permanece en la misma cola de prioridad, en cambio si está usó todo el CPU en el tiempo que se le asignó bajará de prioridad dado que se considerará un proceso con alto consumo de CPU.
* Después de un período S, los procesos reciben un boost y se mueven a la cola con más prioridad. Esta regla impide que aparezca starvation en los procesos interactivos de I-O y que nunca se lleguen a ejecutar los procesos de baja prioridad. Dado que con el boost en algún momento recibirán atención todos los procesos.

Bien ahora yendo al ejemplo, el proceso entrara en la cola con mayor prioridad, suponemos que hay infinitas colas y que sólo existe este único proceso en el scheduler dado que no se especificó en la consigna:

Q0 -> corre 2 ms, se interrumpe por el scheduler y como no termino baja de prioridad

Q1 -> corre 7ms, se interrumpe por el scheduler y como no termino baja de prioridad

Q2 -> corre 12ms, se interrumpe por el scheduler y como no termino baja de prioridad

Q3 -> corre 17ms, se interrumpe por el scheduler y como no termino baja de prioridad

Q4 -> corre 2ms, el proceso termina.

Por lo tanto el proceso se interrumpió 4 veces hasta finalizar, y terminó en la cola Q4.

**Describa la política de scheduling que utiliza el kernel de linux.**

El scheduler de linux se llama CFS (completely fair scheduler), implementa lo que se denomina fair-share scheduling de una forma altamente eficiente y de forma escalable. Para lograr la meta de ser eficiente, CFS intenta gastar muy poco tiempo tomando decisiones de planificación de dos formas:

* Por su diseño
* Debido al uso inteligente de estructuras de datos para esa tarea.

Los schedulers tradicionales se basan en el concepto de time-slice fijo, CFS opera de forma diferente. El objetivo es dividir de forma justa la CPU entre todos los procesos que están compitiendo por ella. Esto lo hace mediante una simple técnica para contar, llama Vruntime (virtual runtime). El Vruntime (se mide en nanosegundos) es el tiempo que se está ejecutando el proceso normalizado por el número de procesos runnable.

A medida que un proceso se ejecuta este acumula Vruntime. En el caso más básico cada Vruntime se incrementa con la misma tasa, en proporción al tiempo (real) físico. Cuando una decisión de planificación ocurre, CFS seleccionará el proceso con menos Vruntime para que sea el próximo a ser ejecutado (por eso se dice que esta técnica intenta ser justo y compartir el CPU a todos los procesos que lo soliciten).

Hay que tener en cuenta:

* Si el CFS switchea de proceso en tiempos muy pequeños estará garantizando que todos los procesos se ejecutan a costa de pérdida de performance, demasiados context switches.
* Si CFS switchea pocas veces, la performance del scheduler es buena pero el costo está puesta del lado de la equitatividad (fairness).

Por lo tanto CFS maneja este trade off mediante varios parámetros de control.

* **sched\_latency**: este valor determina por cuánto tiempo un proceso tiene que ejecutarse antes de considerar un switcheo (es como un time-slice pero dinámico). Un valor típico de este parámetro es de 48ms, CFS divide este valor por el número de procesos ejecutándose en la CPU para determinar el time-slice de un proceso, y entonces se asegura que por ese periodo de tiempo, CFS va a ser completamente justo. Por ejemplo con 4 procesos ejecutándose, CFS divide el valor de sched\_latency, quedando 48ms / 4 = 12 ms para cada proceso. CFS planifica el primer job y lo ejecuta hasta que ha utilizado sus 12ms de (virtual) runtime, y luego chequea si hay algún otro proceso con menos Vruntime, si lo hay switchea.

Observación, si hay muchos procesos ejecutándose esto llevaría a muchos context switch y por ende pequeños time slices. Para subsanar esto se agrega el siguiente parámetro:

* **min\_granularity**: que normalmente se setea con el valor de 6 ms. Entonces CFS nunca ajustaría el time slice de un proceso por debajo de ese número, por ende esto asegura que no haya overhead por context switch.

Observación, CFS utiliza una interrupción periódica de tiempo, lo que significa que sólo puede tomar decisiones en periodos de tiempos fijos (1 ms).

Luego para manejar la prioridad entre procesos CFS lo hace mediante Weighting/Niceness, de forma tal que usuarios y administradores pueden asignar más CPU a un determinado proceso. Esto se hace con un mecanismo clásico de UNIX llamado nivel de proceso nice, este valor va de -20 a 19, con un valor por defecto de 0, los valores positivos de nice implican una prioridad baja y los valores negativos implican una prioridad más alta. Esto se mapea mediante un array de 40 elementos, esos pesos permiten calcular efectivamente el time slice para cada proceso teniendo en cuenta las distintas prioridades para cada proceso

Observación, la tabla de pesos conserva las proporciones de ratio de la CPU cuando la diferencia en valores de nice es constante, es decir mientras los delta de nice se mantengan el cálculo los tiempos de Vruntime asignados a cada proceso se van a mantener constantes.

Finalmente, como el foco de CFS es la eficiencia, además de implementar las políticas mencionadas anteriormente se debe seleccionar un tipo de dato adecuado para encontrar el próximo job a ser ejecutado dado que las listas no escalaban bien. Por eso se optó por utilizar árboles rojo-negro (un tipo de arbol que es balanceado) que escalan en O(log n). El algoritmo sería el siguiente:

* Elegir el nodo más a la izquierda del árbol de planificación, dado que el de menor Vruntime, y ejecutarlo
* Si el proceso completa su ejecución este es eliminado del árbol. Caso contrario si alcanza su máximo tiempo de ejecución o se interrumpe su ejecución por alguna interrupción es es reinsertado en el árbol basado en su nuevo Vruntime.
* Se repite el primer punto eligiendo el nodo de más a la izquierda.

**¿Qué es la afinidad de cache y para qué se utiliza? Ponga un ejemplo.**

La afinidad de caché es un concepto que se usa bajo un contexto de un scheduler con multiprocesadores con caché, la idea es que cuando un proceso corre sobre un CPU arma parte de su estado sobre la caché de esa CPU, por lo tanto la próxima vez que le toque correr sería beneficioso que lo hiciera en la misma CPU para aprovechar ese estado parcial en caché y de esta forma tener una mayor tasa de hits en la caché y consecuentemente tener un mayor performance general dado que es sabido que la penalidad de ir a memoria es alta. Un ejemplo podría ser un programa que haga operaciones de lectura sobre un arreglo, por lo tanto imaginar que la primera vez serán miss compulsivos en caché y luego serán todos hit. Suponiendo que el proceso es preempted y que el scheduler lo vuelve a asignar al mismo CPU entonces tendrá la info parcial en caché y tendrá mejor performance dado que se evitaron por lo menos algunos misses compulsivos. Si en cambio hubiese sido cargado en otra CPU hubiera tenido que traer todo a cache resultando en un performance menor.

**Se tiene el siguiente esquema de procesos:**

| **Procesos** | **Duración** |
| --- | --- |
| 1 | 69 |
| 2 | 17 |
| 3 | 68 |
| 4 | 8 |
| 5 | 17 |
| 6 | 24 |

**Q1 = RR (quantum(time slice) = 15)**

**Q2 = RR (quantum(time slice) = 25)**

**Q3 = FIFO**

**El time arrival de los procesos es 0 para todo y no hay Boost time.**

**Se pide calcular utilizando MLFQ:**

**\* Completion time: tiempo t en el que se completó el proceso.**

**\* Turnaround time (CT-AT)**

**\* Waiting time( TAT - Burst)**

*Paso 0*

Q1: P1, P2, P3, P4, P5, P6

Q2:

Q3:

TP1: 69

TP2: 17

TP3: 68

TP4: 8

TP5: 17

TP6: 26

*Paso 1*

Q1: P2, P3, P4, P5, P6

Q2: P1

Q3:

TP1: 54

TP2: 17

TP3: 68

TP4: 8

TP5: 17

TP6: 26

*Paso 2*

Q1: P3, P4, P5, P6

Q2: P1, P2

TP1: 54

TP2: 2

TP3: 68

TP4: 8

TP5: 17

TP6: 26

*Paso 3*

Q1: P4, P5, P6

Q2: P1, P2, P3

Q3:

TP1: 54

TP2: 2

TP3: 53

TP4: 8

TP5: 17

TP6: 26

*Paso 4*

Q1: P5, P6

Q2: P1, P2, P3

Q3:

TP1: 54

TP2: 2

TP3: 53

TP4: -

TP5: 17

TP6: 26

*Paso 5*

Q1: P6

Q2: P1, P2, P3, P5

Q3:

TP1: 54

TP2: 12

TP3: 53

TP4: -

TP5: 2

TP6: 26

*Paso 6*

Q1:

Q2: P1, P2, P3, P5, P6

Q3:

TP1: 54

TP2: 12

TP3: 53

TP4:  -

TP5: 2

TP6: 11

*Paso 7*

Q1:

Q2: P2, P3, P5, P6

Q3: P1

TP1: 29

TP2: 12

TP3: 53

TP4:  -

TP5: 2

TP6: 11

*Paso 8*

Q1:

Q2: P3, P5, P6

Q3: P1

TP1: 29

TP2:  -

TP3: 53

TP4: -

TP5: 2

TP6: 11

*Paso 9*

Q1:

Q2: P5, P6

Q3: P1, P3

TP1: 29

TP2:  -

TP3: 28

TP4: -

TP5: 2

TP6: 11

*Paso 10*

Q1:

Q2: P6

Q3: P1, P3

TP1: 29

TP2:  -

TP3: 28

TP4: -

TP5: -

TP6: 11

*Paso 11*

Q1:

Q2:

Q3: P1,P3

TP1: 29

TP2:  -

TP3: 28

TP4: -

TP5: -

TP6: -

*Paso 11*

Q1:

Q2:

Q3: P3

TP1: -

TP2:  -

TP3: 28

TP4: -

TP5: -

TP6: -

*Paso 11*

Q1:

Q2:

Q3:

TP1: -

TP2:  -

TP3: -

TP4: -

TP5: -

TP6: -

P1

Turnaround time = Completion time = 158 + 29 = 187

Waiting time = 15\*5 + 8 + 25\*2 + 12 + 2 + 11 = 158

P2

Turnaround time = Completion time = 108 + 12 = 120

Waiting time = 15\*5 + 8 + 25 = 108

P3

Turnaround time = Completion time = 187 + 28 = 215

Waiting time = 187

P4

Turnaround time = Completion time = 45 + 8 = 53

Waiting time = 45

P5

Turnaround time = Completion time = 145 + 2  = 147

Waiting time = 145

P6

Turnaround time = Completion time = 132 + 11 = 143

Waiting time = 132

**Se tiene el siguiente esquema de procesos:**

| **Procesos** | **Duración (Burst)** |
| --- | --- |
| **P1** | **79** |
| **P2** | **37** |
| **P3** | **98** |
| **P4** | **45** |
| **P5** | **57** |
| **P6** | **54** |

**Q1 = RR (Quantum = 3)**

**Q2 = RR (Quantum = 3)**

**Q3 = FIFO**

**- El time arrival de los procesos es 0 para todo.**

**- Boost Time = 60.**

**Usando MLFQ, calcular:**

**- Completion Time**

**- Turnaround Time**

**- Waiting Time**

MLFQ = Es un metodo de planificacion de procesos que utiliza el Kernel. Es un conjunto de colas multiniveles que utilizan feedback para determinar la prioridad de una tarea dada.

Este metodo tiene 5 reglas:

1. Si prioridad de A > prioridad de B -> Se ejecuta A y NO se ejecuta B.
2. Si prioridad de A = prioridad de B -> Se ejecutan ambos en RR.
3. Cuando un proceso entra al sistema se coloca en la Q de mayor prioridad.
4. Una vez que un proceso usa su asignacion de tiempo -> Baja a una Q de menos prioridad.

5. Pasado el Boost time -> todos los procesos suben a la Q con prioridad mas alta.

Siguiendo la REGLA 3, todos los procesos entran a Q1. Cabe destacar que el orden de los procesos es indiferente ya que el tiempo de arribo de todos los procesos es 0, es decir arriban todos a la vez.

Q1 = P1(79), P2(37), P3(98), P4(45), P5(57), P6(54)

Q2 = -

Q3 = -

La Q1 es RR, esto significa que todos los procesos se van a ejecutar durante un determinado tiempo.

En este caso ese tiempo es 3 Quantums. Luego Siguiendo la REGLA 4, una vez que que se ejecutaron durante su asignacion (3 Quantums) bajan a una Q de menos prioridad.

Q1 = -

Q2 = P1(76), P2(34), P3(95), P4(42), P5(54), P6(51)

Q3 = -

Como la Q2 es RR, sucede nuevamente lo mismo.

Q1 = -

Q2 = -

Q3 = P1(73), P2(31), P3(92), P4(39), P5(51), P6(48)

La Q3 es FIFO, esto significa que el primer proceso en entrar a la Q3 es el primero en ejecutarse completamente.

A todo esto no hay que olvidarse del Boost Time (60 Quantum). El Boost Time lo que logra es pasado su tiempo (60 Quantums) sube a TODOS los procesos a la Q de mayor prioridad, en este caso Q1.

Hasta este momento (justo antes que se ejecuten los P en Q3) el tiempo total de ejecución es = 2 x (3 x 6) = 36 Quantums.

Cuando el P1(73) cumpla 24 Quantums de ejecución se acciona el Boost Time y todos los P suben a Q1.

Q1 = P1(49), P2(31), P3(92), P4(39), P5(51), P6(48)

Q2 = -

Q3 = -

Ahora se repite el ciclo:

(Nota: En cada ciclo hago de cuenta que la cuenta empieza en cero devuelta para simplificar los cálculos pero claramente no es así, el tiempo sigue corriendo.)

Q1 = -

Q2 = P1(46), P2(28), P3(89), P4(36). P5(48), P6(45)

Q3 = -

Q1 = -

Q2 = -

Q3 = P1(43), P2(25), P3(86), P4(33), P5(45), P6(42)

Nuevamente cuando el P1(43) cumpla 24 Quantums de ejecución se acciona el Boost Time.

Q1 = P1(19), P2(25), P3(86), P4(33), P5(45), P6(42)

Q2 = -

Q3 = -

Q1 = -

Q2 = P1(16), P2(22), P3(83), P4(30), P5(42), P6(39)

Q3 = -

Q1 = -

Q2 = -

Q3 = P1(13), P2(19), P3(80), P4(27), P5(39), P6(36)

Ahora nuevamente deben pasar 24 Quantums para que se accione el Boost Time.

En este caso se termina de ejecutar el P1 y se comienza a ejecutar un poco de P2.

(Nota: Al terminar P1, el planificador lo saca de la cola.)

Q1 = P2(8), P3(80), P4(27), P5(39), P6(36)

Q2 = -

Q3 = -

*OJO! ACA CAMBIAN LOS VALORES PARA EL BOOST TIME !!!!!*

Al finalizar el P1 el tiempo de ejecución de en cada cola es menor.

Por ejemplo en Q1 antes hacíamos 6 x 3 = 18, ahora es 5 x 3 =15.

Q1 = -

Q2 = P2(5), P3(77), P4(24), P5(36), P6(33)

Q3 = -

Q1 = -

Q2 = -

Q3 = P2(2), P3(74), P4(21), P5(33), P6(30)

Ahora no faltan 24 Quantums para que se accione el Boost Time. Hasta el momento justo antes que se ejecute P1 en Q3 paso (2 x (5 x 3)) = 30 Quantums. Entonces deben pasar 30 Quantums mas para que se accione el Boost Time.

En esta caso finaliza el P2 y se ejecuta parte de P3.

Nuevamente van a cambiar los valores para el Boost Time.

Q1 = P3(46), P4(21), P5(33), P6(30)

Q2 = -

Q3 = -

Q1 = -

Q2 = P3(43), P4(18), P5(30), P6(27)

Q3 = -

Q1 = -

Q2 = -

Q3 = P3(40), P4(15), P5(27), P6(24)

Justo antes que se empiecen a ejecutar los P en Q3, tenemos tiempo total de = 2 x (4 x 3) = 24. Entonces ahora nos faltan 36 Quantums para que se accione el Boost Time.

Se ejecuta P3 durante 36 Quantums y suben todos los P a Q1.

Q1 = P3(4), P4(15), P5(27), P6(21)

Q2 = -

Q3 = -

Q1 = -

Q2 = P3(1), P4(12), P5(24), P6(18)

Q3 = -

Q1 = -

Q2 = -

Q3 = P4(9), P5(21), P6(15)

Ahora justo antes que se ejecuten los P en Q3 en total tenemos (4 x 3) + 1 + (3 x 3) = 12 + 1 + 9 = 22

Ahora faltan 38 Quantums para que se accione el Boost Time.

Se ejecuta P4 durante 9 Quantums y finaliza luego se ejecuta P5 durante 21 Quantums y finaliza. Ahora el único proceso restante P6(15) se ejecuta por 8 Quantums.

Q1 = P6(7)

Q2 = -

Q3 = -

Q1 = -

Q2 = P6(4)

Q3 = -

Q1 = -

Q2 = -

Q3 = P6(1)

Se termina de ejecutar P6 antes de que se accione el Boost Time.

Ahora debemos calcular:

- *Completion Time* -> Tiempo T que tardo en completarse cada proceso

- *Turnaround Time* -> Completion Time - Time Arrival

En este caso el Turnaround Time = Completion Time porque el Time Arrival de cada proceso es 0.

- *Waiting Time* -> Turnaround Time - Burst (Duracion de cada proceso)

Es el tiempo que cada proceso espero para ser ejecutado

Completion Time:

CT-P1 = 18 + 18 + 24 +18 + 18 + 24 + 18 + 18 + 13 = 169 Quantums

CT-P2 = 169 + 11 + 15 + 15 + 2 = 212 Quantums

CT-P3 = 212 + 28 + 12 + 12 + 36 + 12 + 1 = 313 Quantums

CT-P4 = 313 + 9 + 9 = 331 Quantums

CT-P5 = 331 + 21 = 352 Quantums

CT-P6 = 352 + 8 + 3 + 3 + 1 = 367 Quantums

Turnaround Time:

TAT-P1 = CT-P1 - 0 = 169 Quantums

TAT-P2 = CT-P2 - 0 = 212 Quantums

TAT-P3 = CT-P3 - 0 = 313 Quantums

TAT-P4 = CT-P4 - 0 = 331 Quantums

TAT-P5 = CT-P5 - 0 = 352 Quantums

TAT-P6 = CT-P6 - 0 = 367 Quantums

Waiting Time:

WT-P1 = TAT-P1 - 79 = 90 Quantums

WT-P2 = TAT-P2 - 37 = 175 Quantums

WT-P3 = TAT-P3 - 98 = 215 Quantums

WT-P4 = TAT-P4 - 45 = 286 Quantums

WT-P5 = TAT-P5 - 57 = 295 Quantums

WT-P6 = TAT-P6 - 54 = 313 Quantums

**Explique tres políticas de Scheduling y compárelas entre ellas**

Comparación entre FIFO, SJF y RR:

Ejemplo:

| Procesos | Duración |
| --- | --- |
| P1 | 50 |
| P2 | 20 |
| P3 | 20 |

Llegan los 3 procesos en 0.  
  
FIFO:   
Inicia el P1 en 0 y finalizaremos en 50  
Inicia P2 en 50 y finaliza en 70  
Inicia P3 en 70 y finaliza en 90  
  
El TAT promedio será de (50 + 70 + 90)/3 = 70  
El Tiempo de respuesta promedio es (0 + 50 + 70)/3 = 40

SJF:  
Inicia el P2 en 0 y finaliza en 20  
Inicia el P3 en 20 y finaliza en 40  
Inicia el P1 en 40 y finaliza en 90  
  
TAT Promedio (20 + 40 + 90)/3 = 50  
Tiempo de respuesta promedio = (0 + 20 + 40)/3 = 20  
  
RR (Time Slice = 10)  
Inicia el P1 en 0 y se detiene en 10  
Inicia el P2 en 10 y se detiene en 20  
Inicia el P3 en 20 y se detiene en 30  
Continua el P1 en 30 y se detiene en 40  
Continua el P2 en 40 y finaliza en 50  
Continua el P3 en 50 y finaliza en 60  
Continua el P1 en 60 y finaliza en 90  
  
TAT Promedio (50 + 60 + 90)/3 = 66.6666  
Tiempo de respuesta promedio = (0 + 10 + 20) = 10  
  
Podemos ver que SJF tiene mejores resultados tanto para TAT  
como para tiempos de respuesta que el FIFO.   
  
El RRen este caso tambien resulta mejor que FIFO para ambos cálculos, pero en un caso donde los procesos tengan tamaños mas similares, tendria mejor TAT el FIFO.  
  
El RR comparado con el SJF tiene mejor tiempo de respuesta ya que  
detiene procesos para que otros tengan su turno, y esto permite que puedan iniciar otros procesos antesde que finalize el primero. Pero por este mismo motivo de no esperar a que se termine, aumenta el TAT.

**Elija dos políticas de planificación y compárelas con un ejemplo concreto. explicando cual es mejor**

Voy a comparar Shortest Job First (SJF) contra Round Robin (RR).

SJF se basa en lanzar primero el proceso con menor tiempo de finalización entre todos los disponibles. Esto ya supone un problema, y es que en la vida real no sabemos cuanto va a durar un proceso.

RR por su parte, consiste en ir rotando entre los distintos procesos

cada un cierto periodo de tiempo predefinido. SJF tiene la desventaja de ser No-Preentive, esto quiere decir que si un proceso muy largo inicia porque es el único en la lista, cuando llega un nuevo proceso corto no se podrá

ejecutar hasta que el primero termine.

Supongamos un ejemplo donde inicialmente tenemos un proceso A que durara 100ut, a las 15ut llega otro proceso B con una duración de apenas 20ut.

- En SJF: Iniciara el proceso A (por ser el unico al comienzo). A las 15ut llegara el proceso B pero seguirá ejecutandose el proceso A hasta terminar.

A las 100ut, el proceso A termina y comienza el B. A las 115ut terminan todos los procesos.

- En RR: Suponemos un "time slice" de 10ut. También iniciara el proceso A. A las 10ut saltara una interrupción que hara que el scheduler desaloje al proceso. Como no hay otro proceso, continuara el mismo. A las 15ut llega

el proceso B. Depende de la implementación, podría suceder que el scheduler se levante automáticamente y reemplace el proceso actual o que espere hasta el próximo "time slice". Para el ejemplo tomaremos este segundo caso.

A las 20ut el scheduler hará el context switch hacia el proceso B, quedando el A con 80ut restantes. A las 30ut se volverá al proceso A, dejando a B con solo 5ut restantes. A las 40ut se volverá nuevamente a B.

A los 45ut B finalizara, con lo cual el scheduler volvera a correr el proceso A durante los 70ut que le quedan para finalizar, terminando en los 115ut.

Podemos ver que ambos métodos utilizan el mismo tiempo total para realizar ambos procesos, con la diferencia que en el caso de RR el proceso B finaliza mucho antes que en SJF, y el A solo un poco después, reduciendo los tiempos

promedio. Eso hace que RR sea mejor que SJF.

Hay que ser cuidadoso, de todas formas, porque el RR implica muchos context switch, los cuales implican perdida de tiempo. Es por esto que es muy importante calcular bien el "time slice".

**Explique detalladamente MLFQ con boost con un ejemplo que contenga 5 procesos algunos que sean altamente interactivos y por lo menos 2 que realice mucho computo. El ejemplo debe mostrar la evolución del sistema de colas y además debe tener por lo menos dos colas con RR y una con otra política. Los ejemplos deben cumplir con los requerimientos del enunciado. Por lo menos con 2 boots.**

Teniendo los siguientes procesos:   
  
P1(5) P2(6) P3(15) P4(65) P5(70)  
  
Siendo P1, P2 y P3 tareas cortas e interactivas  
Siendo P4, P5 tareas de mucho computo  
  
Utilizaremos el siguiente sistema de colas   
  
Q1 - RR (5 ut)  
Q2 - RR (10 ut)  
Q3 - SJF  
Boost --> (20 ut)  
  
Suponemos que todos los procesos llegan a la vez, se pondran en la cola de mayor prioridad, Q1y se procesaran de menor tiempo a mayor tiempo. Quedando de la siguiente forma en la primera iteración:  
  
Q1 - P1(5) P2(10) P3(15) P4(65) P5(70)  
Q2 -  
Q3 -  
  
El primer RR tiene un tiempo de 5 ut, y una vez que se cumple es tiempo, el proceso deja de ejecutarsey pasa a la siguiente cola de prioridad. En este caso,  el P1 se completa y termina (dura lo mismo que las ut del RR)  
Una vez transcurre las 5 ut, queda asi (a cada proceso se le resta dicho tiempo a su tiempo restante) Se ejecutaran el P1, P2 P3 y P4:  
  
Q1 - P5(70)   
Q2 - P2(5) P3(10) P4(60)  
Q3 -   
  
En ese momento se cumplen los segundos correspondientes al boost y todos  los procesos vuelven a la colade mas alta prioridad (se sigue ejecutando P5)

Se ejecuta entonces P5 y queda:  
  
Q1 - P2(5) P3(10) P4(60)  
Q2 - P5(65)  
Q3 -  
  
En el siguiente paso seguimos con el RR, termina su ejecucion P2  
  
Q1 -   
Q2 - P5(65) P3(5) P4(55)   
Q3 -  
  
Vuelven a pasar entonces los segundos correspondientes al boost (20 ut) entonces todos los procesos vuelven a pasar a Q1  
  
Q1 - P5(65) P3(5) P4(55)  
Q2 -  
Q3 -  
  
Se ejecutan entonces los procesos y vuelven a pasar a la siguiente prioridad  (P3 termina su ejecución)  
  
Q1 -   
Q2 - P5(60) P4(50)  
Q3 -   
  
Aquí el boost esta en 15 ut  
  
Cuando se ejecute el siguiente proceso, restara 5 ut de los 10 correspondientes al RR y se cumplirá el boost asi que regresaran los 2 procesos a Q1  
  
Q1 - P5(60) P4(45)  
Q2 -   
Q3 -  
  
Asi seguiran subiendo y bajando hasta que terminen de ejecutarse ambos (muchos pasos iguales, no llego con el tiempo a dibujarlos)  
Finalmente, las colas quedaran liberadas  
  
Q1 -  
Q2 -  
Q3 -  
  
Un problema que tiene este ejemplo es que por los tiempos elegidos para los RR y para los Boosts,los procesos nunca llegan a la cola Q3 porque el Boost los tira siempre para arriba antes.  
Lo que sucederia si llegaran (tomando otros numeros) es que se ejecutaria el proceso mas corto primero,hasta que el boost los tire nuevamente para Q1 (Shortest Job First)

Por temas de tiempo no llego a cambiar el ejemplo para que pase por Q3