SO: TP 1 - Scheduling

Andreas Sturmer
remruts@gmail.com

Dan Zajdband
dan.zajdband@gmail.com

Javier Casal
javierjcasal@gmail.com

Índice general

Ejercicio 1

En este primer ejercicio se nos pide programar una tarea interactiva que realice n llamadas bloqueantes de una duración al azar comprendida entre dos de los parámetros que la tarea recibe (el tercer parámetro es n, es decir, la mencionada cantidad de llamadas bloqueantes).

Implementación

Para ello empezamos por registrar la tarea en la función provista *tasks_init* según se indica. Luego implementamos la función recién registrada valiéndonos de la función *uso_IO* que resulta en una llamada bloqueante de t ciclos de reloj en completar (donde t es el único parámetro que *uso_IO* recibe. Falta únicamente mencionar que, para definir un tiempo al azar que esté comprendido entre los valores solicitados, realizamos una suma que adicione el mínimo necesario a un valor aleatroio comprendido entre la diferencia entre el mínimo (*bmin*) y el máximo (*bmax*) recibidos:

$$\text{bmin} + (\text{rand}() \,\%\, ((\text{bmax} + 1) - \text{bmin}))$$

Gráfico de un lote de tareas

El gráfico a continuación (*Figura 1*) corresponde a un lote de 1 tarea que realiza 4 llamadas bloqueantes de entre 2 y 7 tiempos de duración:

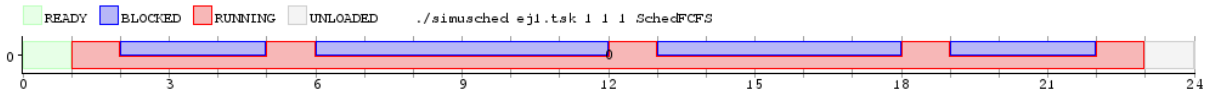

Figura 1

No son muchas las cosas a revisar, entre ellas:
- Las llamadas bloqueantes son, en efecto, 4
- Las llamadas bloqueantes en ningún caso duran menos de 2 tiempos
- Las llamadas bloqueantes en ningún caso se exceden de 7 tiempos de duración

Ejercicio 2

En este segundo ejercicio estamos a cargo de Rolando, su complejo algoritmo, y sus distracciones al momento de sentarse a la computadora.
Nuestro primer encargo es escribir un lote de tareas que simule la situación de Rolando descripta en el enunciado. Este lote incluye entonces una primer corrida de 100 ciclos de CPU en ningún caso bloqueantes buscando simular los 100 ciclos de corrida del algoritmo complejo de Rolando. Nos valemos para ello de la función provista *TaskCPU*. Lo siguiente serán 2 tareas con llamadas bloqueantes de una duración variable dentro de un rango; en otras palabras, 2 tareas que realizan exactamente aquello que se nos pidió en el primer ejecicio del trabajo práctico. Usamos entonces la tarea *TaskConsola* con los parámetros correspondientes para simular las tareas de ocio de Rolando.

Gráficos utilizando 1 y 2 núcleos

Los gráficos a continuación (*Figura 2*) corresponden a la ejecución de la simulación utilizando el algoritmo **FCFS** para 1 core (Figura 2) y para 2 cores (Figura 3). En ambos casos el costo de cambio de contexto es de 4 ciclos.

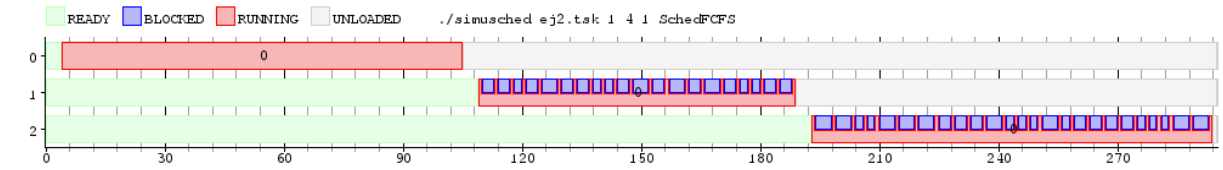

Figura 2

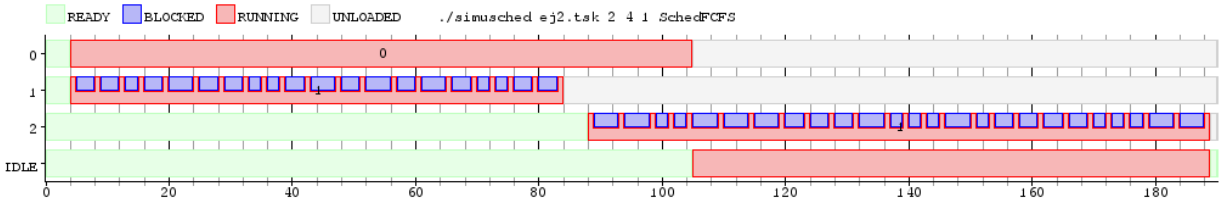

Figura 3

Latencia

La siguiente asignatura es el cálculo de la latencia de cada una de las tareas en cada uno de los dos casos anteriores:

	Tarea 0	Tarea 1	Tarea 2
1 core	4	109	193
2 cores	4	4	88

Desventajas

Hablar de desventajas suena a poco: si lo Rolando cuenta con una computadora de un núcleo que usa un algoritmo de scheduling **FCFS**, va a tener que matar su aburrimiento con alguna otra actividad que no

incluya la computadora en la que está corriendo su complejo algoritmo, ya que ningún otro proceso va a tener CPU disponible para comenzar a ejecutar. En otras palabras, Rolando no podrá reproducir ningún tema musical hasta que el algoritmo no concluya. Esta circunstancia se puede llegar a apreciar en el gráfico correspondiente a la ejecución de la simulación utilizando el algoritmo **FCFS** para 1 core (Figura 2), en donde el primer proceso (proceso 0) se ejecuta de corrido, sin parar ni dejar posibilidad de ejecución a otros procesos. La diferencia es más notable considerando e gráfico correspondiente a la ejecución de la simulación utilizando el algoritmo **FCFS** para 2 cores (Figura 3) en donde la existencia de un core extra implica la posibilidad de que (a lo sumo) un proceso distinto del proceso 0 pueda ser ejecutado sin necesidad de esperar la conclusión del primero.

Ejercicio 3

El tercer ejercicio nos invita a programar **TaskBatch**, esto es, una tarea que realiza *cant_ bloqueos* llamadas bloqueantes en momentos elegidos de forma pseudoaleatoria, y que utilice *total_ cpu* ciclos de reloj.

Implementación

Para la implentación de la tarea pedida utilizamos 2 vectores:
- el primero de ellos es un vector de booleanos (de nombre *ticks* en el código) y nos permitió discretizar el tiempo que la tarea debe permanecer corriendo. Tiene tamaño *total_ cpu*, y cada posición representa un ciclo de reloj. ticks[i] es **true** si y sólo si durante el ciclo i la tarea se bloquea
- mientras que el segundo, *posTiempo*, es un vector de enteros de mismo tamaño que *ticks* inicializado con el valor correspondiente a la posición en cada caso.

Es posible que se trate de detalles poco significativos de la implementación, pero si llegamos hasta acá, los mencionamos: lo que hacemos es desordenar *posTiempo* y asignar el valor **true** a cada una de las posiciones de *ticks* correspondientes a los primeros *cant_ bloqueos* valores de *posTiempo*.
Como hicimos anteriormente, las llamadas bloqueantes las implementamos usando la función *uso_ IO*, mientras que aquellas no bloqueantes usando *uso_ CPU*.

Gráfico

A continuanción (Figura 4) exponemos el gráfico resultado de de un lote de 3 tareas **TaskBatch** con diferentes parámetros (distinta duración y distinta cantidad de bloqueos en cada caso) y **FCFS** como scheduler.

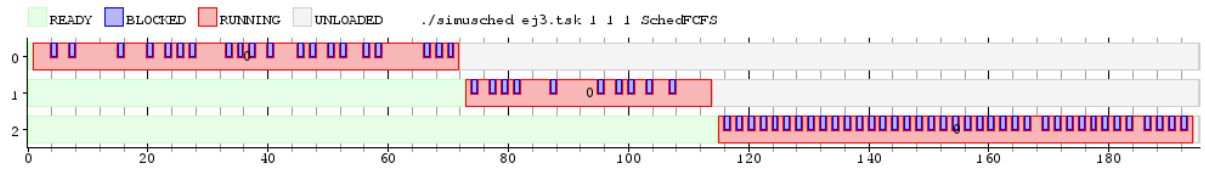

Figura 4

No se nos sugirió ninguna catidad de cores en particular, de modo que elegimos 1 core como opción para nuestra corrida. Esto se puede apreciar notando la ejecución secuencial de las tareas, que en ningún caso arrancan sin que la tarea que las precede concluya.

Ejercicio 4

El cuarto ejercicio no solicita más que la implementación del scheduler *Round-Robin* siguiendo algunas aclaraciones. EOM

Ejercicio 5

La consigna del quinto ejercicio de este trabajo empieza de la siguiente forma:

> Diseñe un lote con 3 tareas de tipo **TaskCPU** de 50 ciclos y 2 de tipo **TaskConsola** con 5 llamadas blooqueantes de 3 ciclos de duración cada una. Ejecutar y graficar la simulación utilizando el scheduler *Round-Robin* con quantum 2, 10 y 50.

Figura 5 corresponde al gráfico de la simulación hecha utilizando un quantum de 2. Figura 6 y Figura 7 corresponden a los gráficos de las simulaciones hechas con 10 y 50 quantums respectivamente.

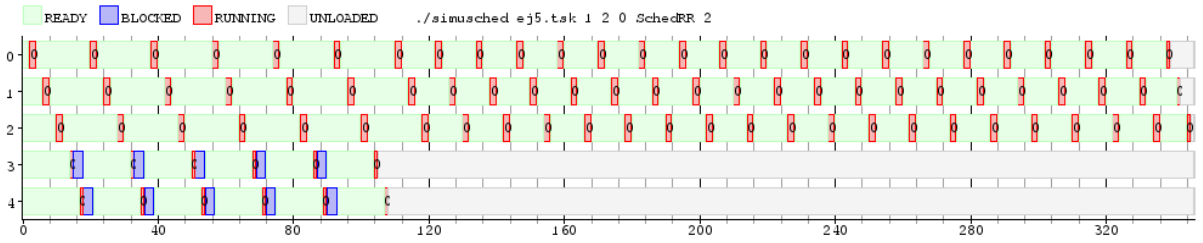

Figura 5

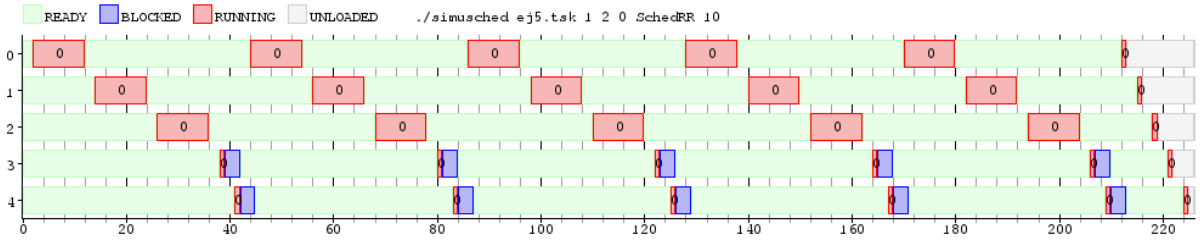

Figura 6

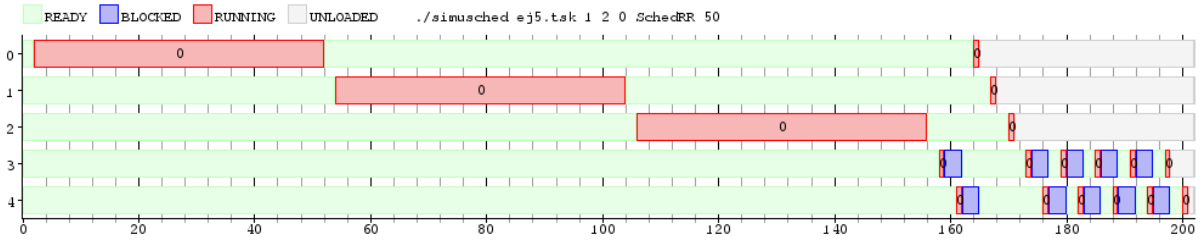

Figura 7

Latencia, waiting time y tiempo total de ejecución

Sigamos adelante con la segunda parte del ejercicio.

Con un cambio de contexto de 2 ciclos y un sólo núcleo calcular la *latencia*, el *waiting time* y el tiempo total de ejecución de las cinco tareas para cada quantum.

	Tarea 0	Tarea 1	Tarea 2	Tarea 3	Tarea 4	Total	Promedio
Quantum 2	2	6	10	14	17	49	9,8
Quantum 10	2	14	26	38	41	121	24,2
Quantum 50	2	54	106	158	161	481	96,2

Cuadro 1: Latencia

	Tarea 0	Tarea 1	Tarea 2	Tarea 3	Tarea 4	Total	Promedio
Quantum 2	289	292	295	84	87	1047	209,4
Quantum 10	162	165	168	201	204	900	180
Quantum 50	114	117	120	177	180	708	141

Cuadro 2: Waiting time

	Tarea 0	Tarea 1	Tarea 2	Tarea 3	Tarea 4	Total	Promedio
Quantum 2	339	342	345	105	108	1239	247,8
Quantum 10	213	216	219	222	225	1095	219
Quantum 50	165	168	170	198	201	902	180,4

Cuadro 3: Tiempo total de ejecución

Finalmente:

¿En cuál es mejor cada uno? ¿Por qué ocurre esto?

Afinemos el lápiz a partir de los números en las tres tablas anteriores y tratemos de entender qué ocurre en cada caso. Empezamos por la latencia, que parece el más evidente. Como podemos ver (Cuadro 1), la relación es lineal: a menor quantum, menor latencia. Parece razonable este comportamiento, basta pensar en qué entendemos cuando hablamos de *quantum*: un menor quantum es en definitiva un menor tiempo de CPU para cada proceso en cada una de las *iteraciones*. Pero esto no es otra cosa que, dado un proceso cualquiera, (con un costo de cambio de contexto fijo para hacer las comparaciones, como es el caso) menor tiempo de espera hasta ser *atendido* por primer vez, ya que todos los procesos que lo precedean en el orden de obtención de recursos de ejecución *tardan* menos. En otra forma de exponerlo, consideremos un proceso cualquiera

$$p_i$$

Sea i su *turno*, es decir, el orden asignado por el scheduler para ser atendido. Hay i - 1 procesos que tendrán CPU antes que aquel que consideramos. Luego, su latencia estará dada por la suma de i - 1 cambios de contexto (esto es un número fijo, según mencionamos) más a lo sumo i - 1 veces el quantum definido. Es por ello que disminuir el quantum disminuirá el segundo sumando, resultando en una menor latencia.
Los análisis respecto del waiting time y el tiempo total de ejecución son similares, ya que (aunque bien podríamos no saberlo de forma anticipada) el tiempo de ejecución que necesitan los procesos definidos es siempre el mismo. Al cambiar el quantum que se asigna, lo que hacemos es modificar la forma (temporal) de atender esas necesidades de procesamiento, no así el tiempo de procesamiento que le llevará a los procesos

concluir su tarea. Lo que puede concluirse viendo los resultados (Cuadro 2 y Cuadro 3) es que a mayor quantum, menor waiting time y tiempo total de ejecución. Una vez más, este resultado es razonable. Considerando fuertemente la aclaración respecto de la nula variación del tiempo efectivo de procesamiento, lo que varía entonces es el tiempo que un proceso espera estando listo. Este tiempo de espera es una suma de 2 valores: por un lado, el *tiempo efectivo de espera*, es decir, el tiempo que efectivamente un proceso espera que otro proceso libere CPU, y costo de los cambios de contexto, por otro. Como el *tiempo efectivo de espera* es inevitable, dada la limitante en los recursos de procesamiento, lo único que puede reducirse es el costo de los cambios de contexto. ¿Cómo reducimos el costo de los cambios de contexto? Cambiando de contexto lo menos posible. ¿Cómo cambiamos de contexto lo menos posible? Asignando un quantum mayor, según se verifica en los resultados.

Ejercicio 6

Lo que tenemos a continuación (Figura 8) es, según se pide, el gráfico correspondiente al mismo lote de tareas del ejercicio anterior pero en este caso para el scheduler **FCFS**.

Figura 8

Tanto conceptualmente, como en la práctica, Round-Robin y **FCFS** son schedulers diametralmente distintos. Son distintos desde la definición hasta (o, mejor dicho, *como consecuencia de*) aquellos aspectos que buscan privilegiar al momento de asignar los recursos de procesamiento del sistema. **FCFS** es, por definición, un scheduler que asigna los recursos al primer proceso que los solicita. ***F**irst **C**ome **F**irst **S**erve* es el significado de sus siglas en inglés. En nuestro ejemplo, el proceso que llega en primer lugar es el proceso 0 (*p0* de ahora en adelante). En la Figura 8 se puede ver cómo p0 es el primer proceso en comenzar a ejecutar. El mencionado es un aspecto importante de la definición del **FCFS**, aunque no la abarca totalmente. Bien vistos, los gráficos correspondientes a las corridas del Ejecicio 5 (Figura 5, Figura 6, y Figura 7) nos advierten que con lo mencionado hasta acá no alcanza para diferenciar al **FCFS** del Round-Robin: sin importar el quantum utilizado, el primer proceso en empezar a ejecutar es, en todos los casos, p0. No es (desde ya) una casualidad. Un scheduler Round-Robin tiene por lo general una política **FIFO** (***F**irst **I**n **F**irst **O**ut*) para la elección del orden de los procesos a ejecutar. Esto es aparentemente muy similar a lo que ocurre con un **FCFS**... ¿entonces? Round-Robin agrega un extra: *preemption*. Como dato de color, la traducción de *preemption* según el portal translate.google.com es:
- derecho preferente de compra
- preempción
De más está seguramente aclarar que no será esa traducción nuestro punto de partida. Hablar de *preemption* es hablar de desalojo. Round-Robin, a diferencia de **FCFS**, desaloja procesos. El criterio para hacerlo es el tiempo que el proceso lleva efectivamente ejecutando. *Quantum* es el nombre que usamos para referirnos al máximo tiempo que el scheduler le permite a un proceso ejecutar antes de darle permiso a otro proceso (a costa del primero, que es desalojado). Volviendo a los gráficos podemos ver cómo aquellos que usan un scheduler Round-Robin (Figura 5, Figura 6, y Figura 7) nos muestran procesos que ejecutan sólo durante el quantum asignado (ahora sí hace diferencia el quantum definido), en contraste con aquel que usa un scheduer **FCFS** (Figura 8) en donde los procesos ejecutan hasta que terminar (en caso de terminar). Dijimos anteriormente que son distintos los aspectos que cada tipo de scheduler busca privilegiar. Ahora podemos agregar más detalle a aquella sentencia: mientras que un scheduler **FCFS** busca minimizar el Waiting time y el tiempo total de ejecución vía minimizar el costo por los sucesivos cambios de contexto (de la forma detallada en el Ejercicio 5), un scheduler Round-Robin paga ese costo a cambio de un criterio más *justo*. Justo en lo que a tiempo de espera se refiere.

Ejercicio 7

Este ejercicio parte de la observación del comportamiento de un scheduler llamado SchedMistery, del cual poseemos solamente una versión compilada y no su código fuente. Se nos plantea entender su comportamiento y generar un scheduler SchedNoMistery que lo imite.

Descripción del comportamiento observado

Las observaciones de múltiples lotes (que serán detallados más adelante en la sección) nos llevaron a entender las decisiones tomadas por el SchedMistery.

Este scheduler toma 1 o más enteros como parámetros. La cantidad de parámetros indica la cantidad de colas de prioridad para los procesos y cada parámetro referencia el quantum de los procesos en cada cola de prioridad.

Nos encontramos frente a un Round-Robin de múltiples colas con prioridad. La forma en que cambian de prioridad los procesos se da en los siguientes casos:

- Cuando una tarea se desbloquea entonces sube un nivel de prioridad. - Cuando una tarea pierde su quantum baja un nivel de prioridad.

Implementación

El comportamiento del scheduler queda descripto en el apartado anterior con lo cual nos vamos a limitar a explicitar las estructuras de datos intervinientes.

Para representar las múltiples colas utilizamos un vector de colas, cuya cantidad de elementos se corresponde con la cantidad de argumentos recibidos por el scheduler. A la vez, un vector de enteros guarda un entero por cada cola de prioridad, determinando el quantum de los procesos que la contienen. Nos resultó cómodo utilizar un mapa de entero a entero que, para cada pid devuelve la prioridad del proceso correspondiente. Pensamos inicialmente en usar un struct para guardar información del proceso, pero el mapa fue más efectivo además de requerir menos cuidado a la hora de modificarse.

Lotes de tareas de experimentación

Lote 1

*3 TaskCPU 20

*2 TaskConsola 5 3 3

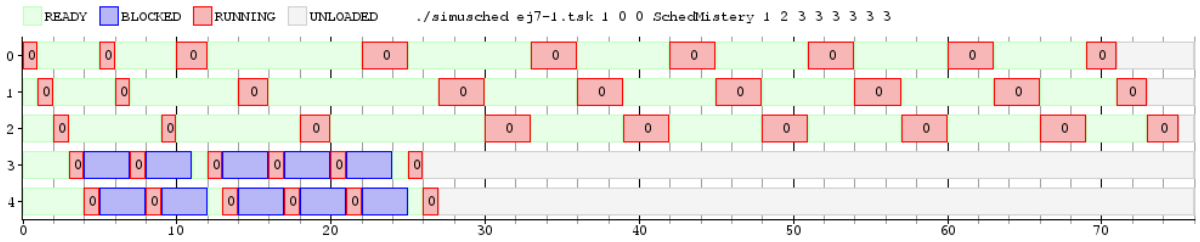

Figura 9: Ejecución Lote 1

Lo que nos dice la figura 9 es que este scheduler hace Round-Robin sobre los procesos que ejecuta y además prioriza las tareas desbloqueadas por sobre el resto. También vemos que los quantum van increciendo hasta 3, correspondiendose con los parámetros pasados al scheduler ‘1 2 3 3 3 3 3 3‘. Esto habla de la necesidad

de al menos 2 colas de prioridad para manejar los niveles de prioridad para los casos de desbloqueo y la nacesidad de utilizar un algoritmo de Round-Robin para el intercambio de tareas.

Lote 2

TaskAlterno 3 22 1 3 3
@1
TaskAlterno 7 10 3
@20
TaskCPU 5
@22
TaskCPU 5
@30
TaskCPU 5
@32
TaskCPU 5

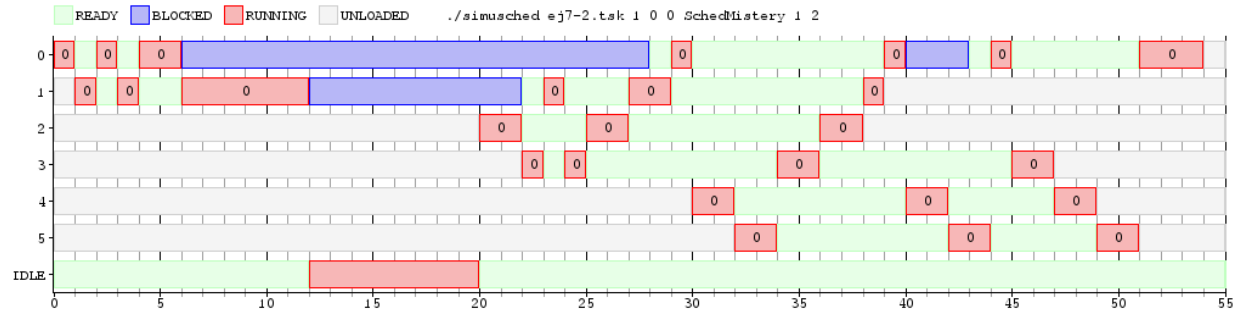

Figura 10: Ejecución Lote 2

Un punto importante en la determinación del comportamiento del SchedMistery está relacionada con la inicialización de tareas. Lo que determinamos a través de la ejecución de este lote, y el posterior análisis de la figura 10, es que los procesos inicializan con la mayor prioridad posible. También comprobamos que la suba de prioridad del desbloqueo se complementa con el de incialización de tareas.

Lote 3

*2 TaskCPU 10
@5
*2 TaskCPU 10
@10
*2 TaskCPU 10

Lo que determina este lote, que se podía intuir en los lotes anteriores, es que las colas de prioridad tienen un quantum que se corresponde con los parámetros pasados al scheduler y, agregando a lo conocido por el Lote 1, por cada parámetro existe una cola de prioridad donde los procesos se van encolando. Es así que logramos comprender como funciona el Scheduler.

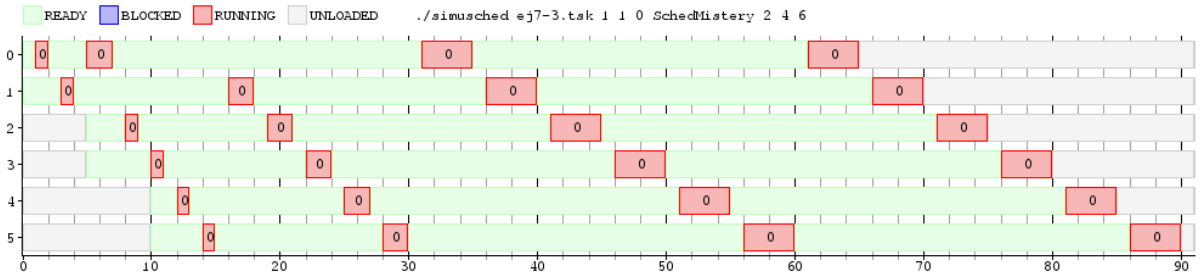

Figura 11: Ejecución Lote 3

Creamos una batería de lotes de tests para contemplar otros casos posibles y logramos el mismo comportamiento con nuestra implementación de SchedNoMystery que el de SchedMistery. Sin embargo los 3 casos más emblemáticos son los anteriormente detallados, cumpliendo con la consigna de hablar de los 3 lotes más significativos.

Ejercicio 8

En este punto, se nos asignó la tarea de implementación de un scheduler Round-Robin que no permita la migración de procesos entre núcleos. La asignación de CPU se realizaría en el momento de carga de un proceso y el núcleo correspondiente al mismo sería aquel con menor cantidad de procesos activos totales.

Implementación

La implementación de este scheduler no difiere mucho del de la del ejercicio 4, sólo que en este caso se tiene una cola de procesos por CPU. Además, se posee un entero por CPU (en un vector) que indica la cantidad de procesos activos para cada uno de estos, a ser utilizados cuando se cargue un proceso nuevo. Por último, poseemos un mapa de procesos para saber a qué CPU corresponde cada uno.

Al cargar un proceso, se obtiene el índice (CPU) del mínimo elemento del vector de ästivosÿ con eso se empuja el proceso a la cola correspondiente y se aumenta la cantidad de activos. Por último, se agrega el proceso con su CPU al mapa de procesos.

Al hacer un tick, la única diferencia con el código del ejercicio 4, además de elegir la cola correspondiente al cpu indicado, es al momento de finalización de una tarea. En este caso, se le resta uno a la cantidad de activos del CPU y se elimina al proceso del mapa de procesos. Luego, es análogo.

Al desbloquearse un proceso, simplemente se obtiene el CPU del mismo a través del mapa de procesos. Con esto, se empuja al proceso al core correspondiente.

Ejemplo 1

Un escenario en el que este scheduler es peor en contraposición con el Round-Robin tradicional, es cuando se van alternando procesos de corta duración con procesos que tarden bastante, como por ejemplo sería el caso de dos cálculos que difieran mucho en complejidad. Imaginemos el siguiente caso. Somos la central que controla los subtes de una mediana ciudad. Por una falta de previsión en el diseño del sistema de subterraneos, hay ocasiones en que los subtes quedan ubicados de forma tal que es inviable su correcto funcionamiento. Por suerte, el sistema logra anticipar este escenario, a partir de lo cuál debe estimar el tiempo que demandará la solución, por un lado, y definir la correcta sucesión de movimientos que tiene que hacer cada unidad para evitar el colapso. En este caso, terminarían primero los procesos de corta duración, quedando CPUs sin ningún proceso asignado; no se estarían utilizando, realentizando el tiempo de compleción de los procesos y probablemente su tiempo de espera. Sin embargo, la latencia no se vería afectada y sería la misma en ambos casos.

Para ejemplificar esto, construimos un lote de 7 tareas "TaskCPU", alternadas en tiempos de 1 y 8 ciclos. A continuación, se presenta un gráfico de tal lote que simula la situación planteada:

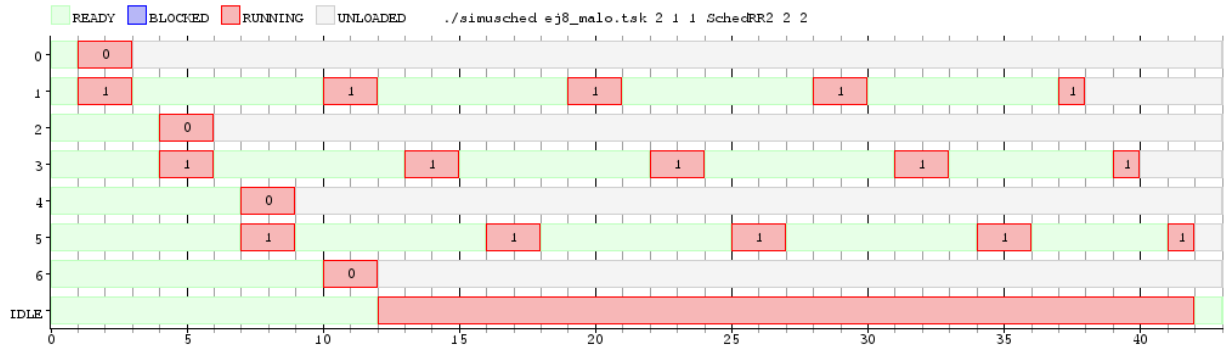

Figura 12: Ejecución en Round-Robin 2

Como se muestra en la figura 15, podemos notar lo antes mencionado. Con este nuevo scheduler, a partir del instante 16, el CPU 0, ya no posee ninguna tarea a ejecutar, siendo que terminaron todas las que fueron asignadas al mismo al momento de carga. Por lo tanto, queda en IDLE, mientras que el CPU 1 ejecuta alternadamente los 3 procesos restantes.

En la siguiente tabla mostramos distintas métricas, correspondientes a esta simulación:

Cuadro 4: Ejecución en Round-Robin 2

	Latencia	Espera	Compleción	Ratio (E/C)
0	1	1	3	0.333
1	1	29	38	0,763
2	4	4	6	0.666
3	4	31	40	0.775
4	7	7	9	0.777
5	7	33	42	0,785
6	10	10	12	0.833
AVG	4,857	16,428	21,428	0,705

En cambio, en la figura 13 (que se presenta a continuación), podemos observar, como con el Round-Robin tradicional, cuando terminan de ejecutarse las tareas cortas (0, 2, 4 y 6), el CPU 1 sigue siendo utilizado por las restantes. Notar que aunque perdemos 1 ciclo extra por cada cambio de CPU, los procesos siguen terminando antes que con el nuevo scheduler implementado.

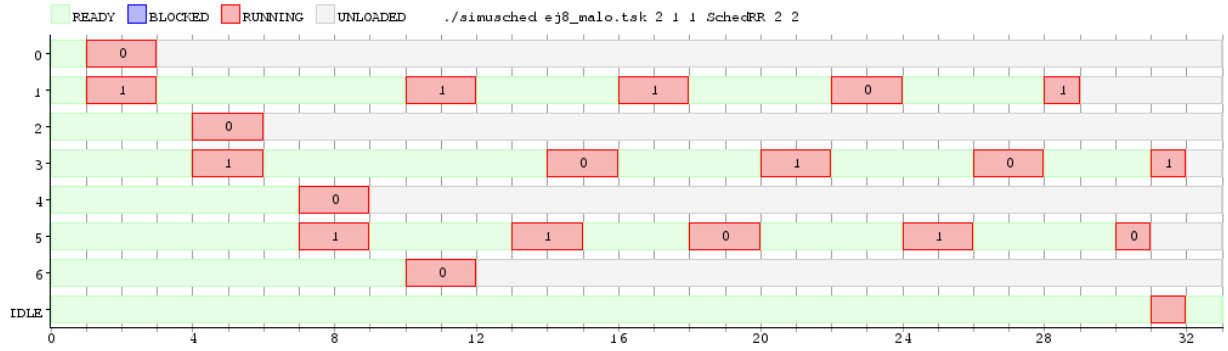

Figura 13: Ejecución en Round-Robin Tradicional

En la siguiente tabla volcamos datos con las mismas métricas que en la tabla correspondiente a *Round-Robin 2*.

Cuadro 5: Ejecución en Round-Robin Tradicional

	Latencia	Espera	Compleción	Ratio (E/C)
0	1	1	3	0.333
1	1	20	29	0,690
2	4	4	6	0.666
3	4	23	32	0,719
4	7	7	9	0.777
5	7	22	31	0,710
6	10	10	12	0.833
AVG	4,857	12,429	17,428	0,675

Como supusimos anteriormente, podemos ver como el tiempo de espera y compleción de los procesos largos es menor en *Round-Robin Tradicional*, por lo que en promedio también es mejor. La latencia se mantiene estable.

Ejemplo 2

Sin embargo, este nuevo tipo de scheduler resulta útil en algunas situaciones. Tal es el caso cuando se quisiera lanzar un proceso que ejecute en tiempo real. A modo ilustrativo, en el ejemplo anterior, podría el usuario querer ejecutar alguna especie de simulación. Imaginemos que ya se terminaron de ejecutar los procesos cortos y queda un core libre. Entonces, la simulación correría sin interrupciones, y aunque llegaran más procesos, en principio caerían probablemente en el mismo CPU (si los otros procesos no terminaron), pero estaría balanceado. El tiempo de espera sería menor, como probablemente también la latencia de la simulación.

Para reflejar este comportamiento, modificamos el lote de tareas anterior, agregando la tarea 7 (TaskCPU de 12 ciclos) en el momento 4. Además, agregamos dos tareas pequeñas más que interrumpen a la tarea 7. El gráfico correspondiente se encuentra a continuación:

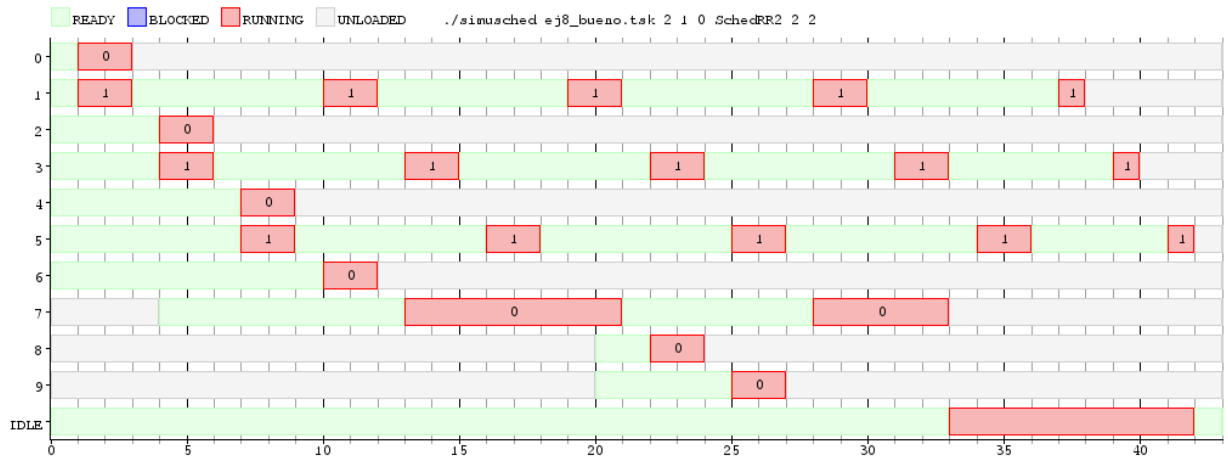

Figura 14: Ejecución en Round-Robin 2

Como mencionamos, si bien la tarea 7 es interrumpida, tiene un corto tiempo de espera (7 ciclos).

Compararlo con la figura 15 que se presenta a continuación:

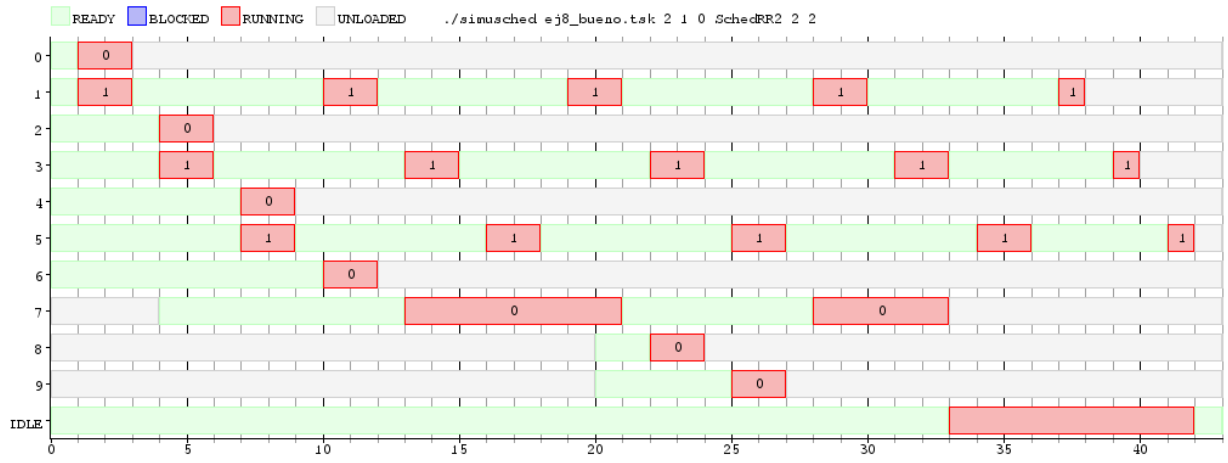

Figura 15: Ejecución en Round-Robin Tradicional

Si bien la tarea 7 tiene la misma latencia en ambos casos, el tiempo de espera es mucho mayor en el *Round-Robin Tradicional*. Como sólo nos importa esta tarea realmente, presentamos una tabla comparativa entre ambos Schedulers para la tarea 7.

Cuadro 6: Comparación entre schedulers para la tarea 7

	Latencia	Espera	Compleción	Ratio (E/C)
RR	1	15	28	0.536
RR2	1	7	20	0.35

Notar que si no existieran las tareas 8 y 9 que interrumpen ó si hubiera un core libre, la espera sería 0. Igualmente, es notable la mejora que provee el nuevo scheduler en esta situación particular.

Además de esto, en la realidad hay otros factores a considerar que no podrían ser medidos con la simulación utilizada. Por ejemplo, en arquitecturas SMP, es importante mantener cierta afinidad al procesador asignado al proceso, para aprovechar la caché. Si hubiera cambio de procesador, el proceso llegaría con la caché vacía, produciéndose un miss y teniendo que ir a buscar los datos a memoria principal. A esto se le suma el tiempo de los cambios de procesador y de contexto que no son despreciables. En ese sentido, el nuevo scheduler sería preferible al *Round-Robin Tradicional*.