DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº1

Scheduling

3 de mayo de 2016 — Sistemas Operativos

Integrante	LU	Correo electrónico
Fourcade de Maussion, Gastón	531/13	gaston.fdm@gmail.com
Tilve Neyra, Dana Marlene	614/14	tilve.dana@gmail.com
Zamboni, Gianfranco	219/13	gianfranco376@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

Introducción

Con este trabajo se busca entender el funcionamiento de distintas técnicas de scheduling asi como identificar pros y cons de cada una de ellas en las distintas situaciones planteadas. Particularmente, se estudiará el comportamiento de los scheduler de tipo *First Came First Served (FCFS)*, *Round Robin (RR)* y además como el uso de colas de prioridad y los distintos tipos de tareas afecta el funcionamiento de los scheduler del segundo tipo.

Cabe aclarar que cuando se intenten reproducir los gráficos de lotes de tareas, en las que se vean involucradas aquellas que se valgan del azar para definir sus variables, no necesariamente se conseguirá el mismo output.

1. Ejercicio 1

Se pidió realizar una tarea `TaskConsola` que realice n llamadas bloqueantes consecutivas. Cada llamada debe gastar entre *bmin* y *bmax* ticks.

Para su implementación, se realizan n llamadas a la función `uso_IO` que realiza una llamada bloqueante tomando como parámetro el ID del proceso y el tiempo de duración del bloqueo, este segundo parametro es generado pseudoaleatoriamente con la función `dameRandEntre`.

La función `dameRandEntre` hace uso de la función `rand()` de la libreria estándar de `c++`. Para asegurar que el número pertenezca al intervalo especificado, se calcula la distancia ente *bmin* y *bmax*, se calcula el modulo entre el numero random y esta distancia para luego sumarlo a *bmin*. Es decir, si *nr* es el número devuelto por la función, su calculo sería el siguiente:

$$nr = (\texttt{rand()} \,\%(bmax - bmin)) + bmin$$

Esto consigue simular una tarea de consola, donde la duración azarosa es producto del input del usuario.

Se realizo un lote de 3 tareas del tipo `TaskConsola` mostrados a cotinuación:

- La primer tarea se carga en el tiempo 0 y realiza 5 bloqueos de entre 7 y 19 ticks.
- La segunda tarea se carga en el tiempo 5 y realiza 2 bloqueos de entre 2 y 13 ticks.
- La segunda tarea se carga en el tiempo 11 y realiza 3 bloqueos de entre 3 y 17 ticks.

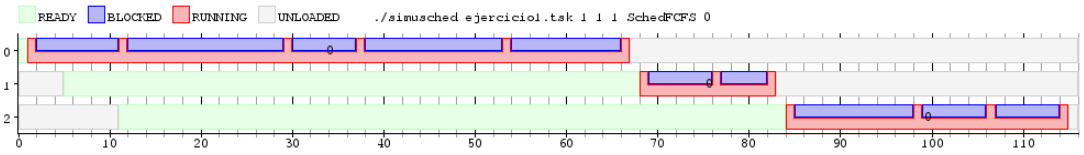


Figura 1: 3 `TaskConsolas` con FCFS para 1 core

2. Ejercicio 2

Se pide realizar un lote de tareas que represente las actividades del grupo de Data Mining de la facultad. Par esto se implementaron 4 tareas que funcionaran acorde a cada uno de los usuarios del sistema (tanto humanos como no humanos).

Estas tareas fueron:

- `procesoPesado` que simula una tarea que usa la cpu durante 500 ciclos de cpu.

- `usuario1`, `usuario2` y `usuario3` que usan el cpu durante 9, 19 y 29 ciclos, respectivamente, para luego bloquearse durante un intervalo de entre 1 y 4 ciclos.

A continuación se incluyen los gráficos obtenidos de la simulación utilizando las cuatro tareas con el algoritmo de scheduling FCFS para 1, 2 y 4 cores:

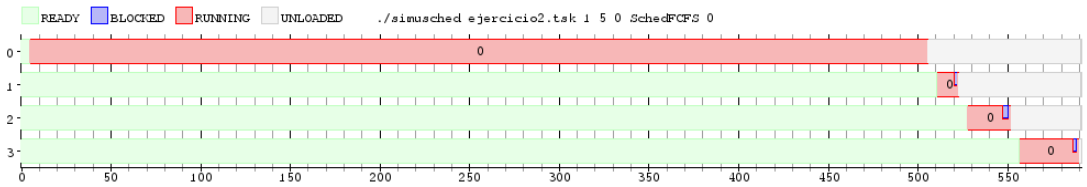


Figura 2: FCFS utilizando 1 core.

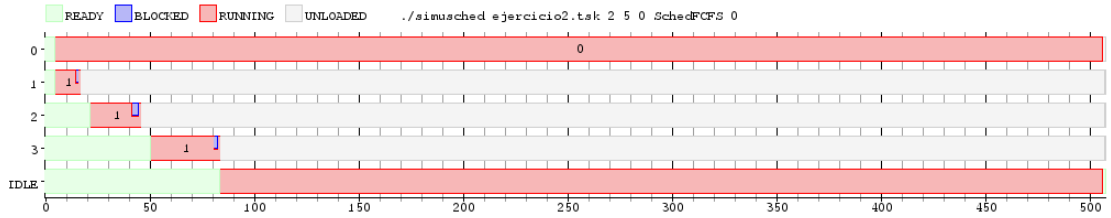


Figura 3: FCFS utilizando 2 cores.

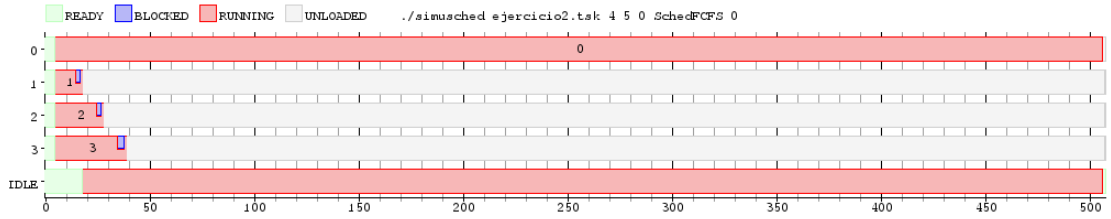


Figura 4: FCFS utilizando 4 cores.

Considerando la evidencia gráfica anterior en conjunto con el siguiente cuadro comparativo mostrando los resultados de los cálculos de latencia en ciclos para cada proceso muestran que, de seguir manteniendo como algoritmo de scheduling FCFS usando una computadora con un único nucleo, los procesos que precisan hacer entrada salida quedan en espera de que finalice el proceso que más ciclos consume (de aquí en más llamado "proceso pesado") cuando podrían tener poca a despreciable latencia utilizando una computadora con 2 cores para el primer caso o 4 cores para el segundo..

Observación: dentro del cálculo de latencia se consideraron los ciclos insumidos en cambio de contexto.

3. Ejercicio 3

Se pide programar la tarea `TaskBatch`, la cual debe recibir por parámetros *total_cpu* (tiempo total de ejecución de una tarea, no incluyendo el tiempo que permanece bloqueada aunque sí el tiempo que toma para hacer la llamada) y *cant_bloqueos* (número de llamadas bloqueantes).Los bloqueos deben tener una duración de 2 ciclos de reloj.

	1 core	2 cores	4 cores
proceso pesado	5	5	5
usuario 1	505	5	5
usuario 2	515	15	5
usuario 3	535	35	5

Cuadro 1: Latencia

Para cumplir con el objectivo de tener bloqueos aleatorios se implementó la función `fillWithOrderedRands`, que toma como parámetros el valor *total_cpu* y un vector de tamaño *cant_bloqueos*, con números aleatorios generados con la función antes mencionada `dameRandEntre` con parámetros 0 y *total_cpu*.

Para evitar el solapamiento de las llamadas estos números aleatorios se incluyen en el vector sólo si cumplan la propiedad `isAcceptable` que indica si, dados los otros números en el vector, es posible ingresarlo de manera que se cumpla que tiene al menos tres ciclos (uno de la llamada y dos de la duración del bloqueo) de diferencia con el siguiente y al menos un ciclo con el anterior para entrar al mismo. Para ello se ordenan en cada iteración.

Este vector es luego utilizado por la tarea para obtener los momentos de bloqueo pedidos.

Finalmente, para tener control sobre la cantidad de cpu que se pide utilizar en total para la tarea se mantienen las variables `cpu_usado` y `cpu_a_usar` que por cada iteración controlan que la cantidad de tiempo que se llama a `uso_CPU` sea el necesario antes de bloquearse.

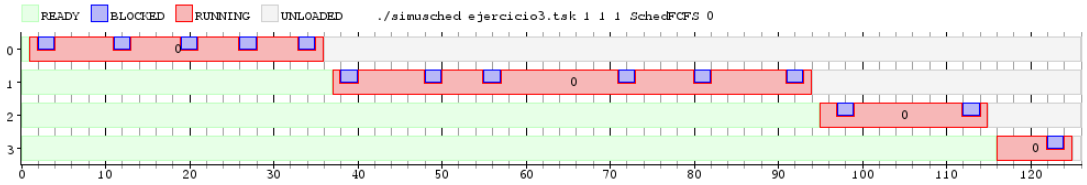


Figura 5: Tareas `TaskBatch` en FCFS

4. Ejercicio 4

4.1. Implementación

Se pide realizar la implementación de un scheduler con algoritmo *Round-Robin*. Para ello se completan las funciones de inicialización, load, unblock y tick. Las estructuras globales que se mantienen son dos: vectores (el primero mantiene los quantums originales y el segundo sirve de contador para las tareas que utilizan actualmente ese core), una única cola para las tareas en *ready*, y una cola de dos puntas para las tareas *blocked*.

La función de inicialización recibe por parámetro la cantidad de cores y sus quantum; en ella se inicializan los vectores mencionados anteriormente.

La función `load` realiza un `push_back` a la cola de tareas.

La función `unblock` itera sobre la cola de tareas bloqueadas buscando el `pid` pasado por parámetro, lo elimina y lo vuelve a poner en la cola de tareas *ready*.

La función `getNextTask` toma como parámetro un core y retorna la siguiente tarea a ejecutarse de la siguiente manera: si en la cola no hay más tareas, entonces la siguiente debería ser la `IDLE_TASK`, sino se toma la siguiente de la cola de tareas, se elimina de la cola y se retorna. Finalmente, la función `tick` toma como parámetro un motivo que puede ser `EXIT`, `BLOCK` o `TICK` y debe retornar la siguiente tarea a ejecutarse. Efectúa distintas acciones en función a las decisiones siguientes:

- la tarea es `IDLE` o el motivo fue un `EXIT`: se resetea el contador de quantums restantes del core y se retorna la siguiente tarea.
- la tarea se queda sin quantum y el motivo es un `TICK`: la tarea vuelve a ser colocada en la cola de tareas *ready*, se resetea el contador de quantums para el core y se retorna la siguiente tarea.
- el motivo es un `BLOCK`: la tarea ingresa a la cola de tareas bloqueadas y se retorna la siguiente tarea.
- para cualquier otro caso, la tarea aún tiene quantum disponible para utilizar, por lo que se decrementa el contador de quantums para el core correspondiente y se retorna la misma tarea que ingresa.

El resultado de la experimentación con distintas canitdades de núcleos de la implementación mencionada fue el siguiente:

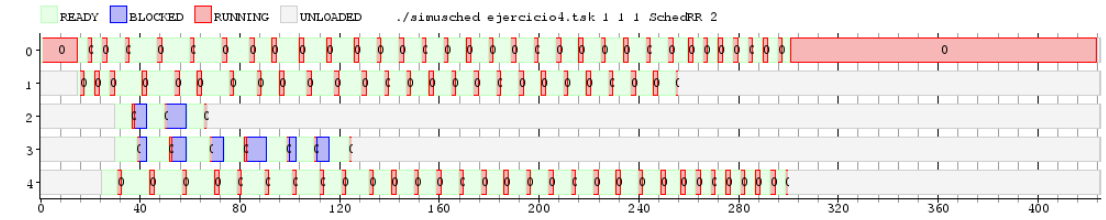


Figura 6: Round-Robin con 1 núcleo.

Figura 7: Round-Robin con 2 núcleos.

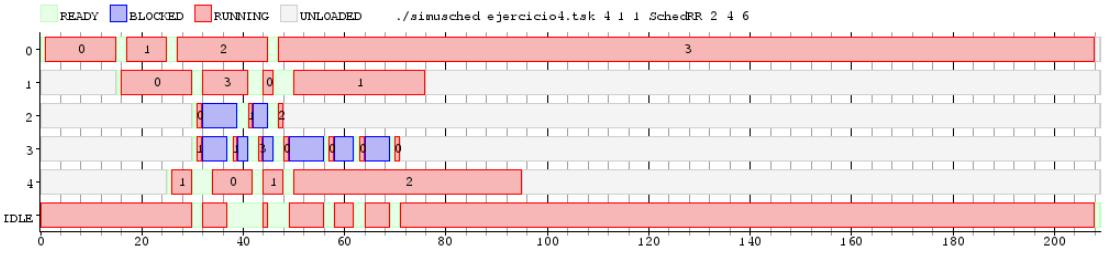


Figura 8: Round-Robin con 4 núcleos.

5. Ejercicio 5

Se pide crear un lote con 3 tareas de tipo TaskCPU de 70 ciclos y 2 de tipo TaskConsola con 3 llamadas bloqueantes de 3 ciclos de duración cada una. Y luego ejecutar una simulación con un scheduler del tipo *Round Robin*. A continuación se incluyen los gráficos obtenidos a partir de la simulación:

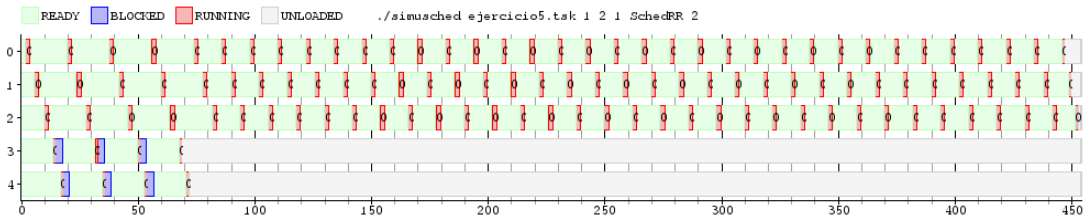


Figura 9: RoundRobin de quantum 2.

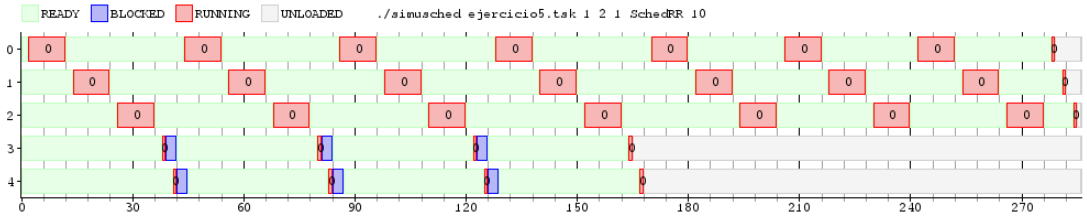


Figura 10: RoundRobin de quantum 10.

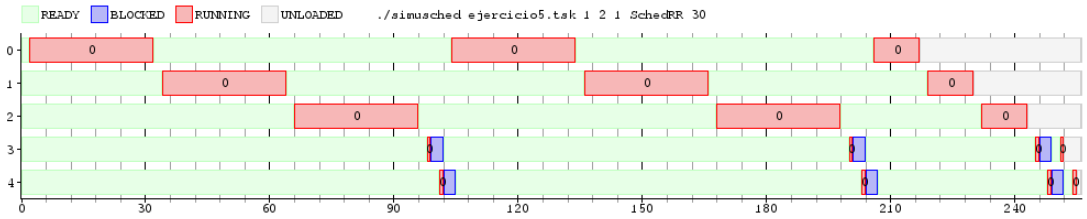


Figura 11: RoundRobin de quantum 30.

A continuación se presentan las métricas comparativas extraídas de los gráficos y las conclusiones a las que se arribaron respecto a la selección de la mejor:

	RRQ2	RRQ10	RRQ30
TaskCPU_1	2	2	2
TaskCPU_2	6	14	34
TaskCPU_3	10	26	66
TaskConsola_1	14	44	98
TaskConsola_2	18	47	101

Cuadro 2: Latencia

Para la métrica de latencia, se concluye que es mejor el Round Robin con quantum 2. Esto es así porque para que una nueva tarea empiece a correr debe esperar a que termine el quantum de la anterior, es decir mayor quantum implica mayor espera.

	RRQ2	RRQ10	RRQ30
TaskCPU_1	446	279	217
TaskCPU_2	450	282	230
TaskCPU_3	454	285	243
TaskConsola_1	69	151	252
TaskConsola_2	72	153	255

Cuadro 3: Tiempo total de ejecución

Para la métrica de tiempo total de ejecución se concluye que para las tareas que no realizan llamadas bloqueantes es mejor en el Round Robin de quantum 30 mientras que para las tareas que sí realizan llamadas bloqueantes es mejor en el de quantum 2.

Esto último ocurre porque las tareas de tipo `TaskConsola` tienen uso mínimo de CPU, por lo que al cumplir las llamadas bloqueantes pedidas, que en este caso fueron 3, terminan. Entonces, si llamamos "vuelta" al momento en el que el Round-Robin elige nuevamente la primer tarea que se encoló, este tipo de tarea necesita 3 vueltas para terminar. Es fácil ver entonces que en cuanto más rápido se de una vuelta más rápido va a terminar, y esto ocurre cuando el quantum es más pequeño.

Por otro lado, las tareas que sólo hacen uso de la CPU se completan más rápido cuanto mayor es el quantum dado que hay menos espera entre vuelta y vuelta.

	RRQ2	RRQ10	RRQ30
TaskCPU_1	377	207	147
TaskCPU_2	379	209	149
TaskCPU_3	382	211	151
TaskConsola_1	56	151	239
TaskConsola_2	59	153	241

Cuadro 4: Waiting Time

Para la métrica de waiting time se concluye que es mejor en el Round-Robin con quantum 30 para las tareas que no realizan llamadas bloqueantes mientras que para las tareas que sí realizan llamadas bloqueantes es mejor en el de quantum 2.

Con el mismo análisis que antes: para una tarea con llamadas bloqueantes es fácil ver que en cuanto más rápido se de una vuelta más rápido va a terminar, lo que implica menos tiempo de espera.

Por otro lado, para las tareas sin llamadas bloqueantes, es más conveniente un quantum mayor para evitar tiempos excesivos de cambio de contexto.

6. Ejercicio 6

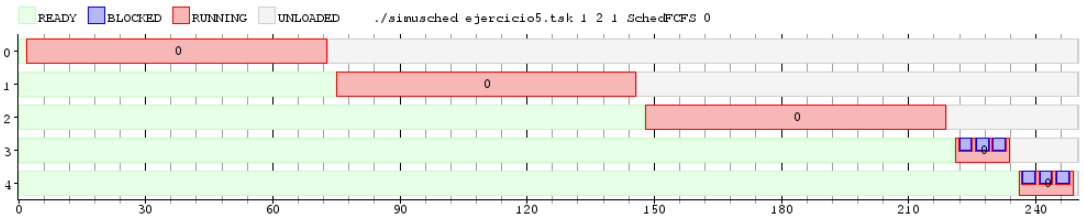
Se busca comparar la política de scheduling FCFS implementado por la cátedra contra nuestra implementación de Round Robin del ejercicio anterior.

A continuación se incluye el gráfico obtenido de la simulación utilizando tres tareas `TaskCPU` y dos tareas `TaskConsola` con 3 llamadas bloqueantes de 3 ciclos cada una y 2 de cambio de contexto, utilizando el algoritmo de scheduling FCFS:

Diferencias a destacar:

Tiempo de ejecución: tareas sin llamadas bloqueantes, que es mucho menor al tiempo tomado cualquiera de las opciones Round-Robin. Para las tareas con llamadas bloqueantes su tiempo de ejecución es similar al peor caso en Round-Robin.

Latencia: Es mayor para todas las tareas (excepto la primera) a cualquiera de las opciones de Round-Robin.

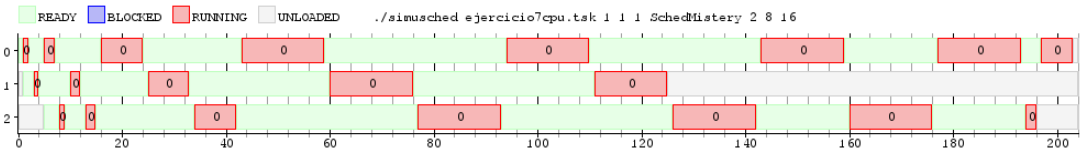


Waiting time: el waiting time promedio (que circula alrededor de los 133,8) es mejor a cualquiera de las opciones de Round-Robin, cuyo mejor waiting time promedio mínimo es 185,4 (Round-Robin con quantum 30). Cabe aclarar que el waiting time para las tareas de tipo `TaskConsola` supera al peor waiting time de cualquiera de las opciones de Round-Robin, mientras que para las tareas de tipo `TaskCPU` ocurre lo contrario.

7. Ejercicio 7

Se tiene el archivo objeto de un scheduler llamado *SchedMistery* del cual no se conoce exactamente su funcionamiento. Se pidió realizar pruebas con este scheduler, deducir su funcionamiento y replicarlo en el scheduler *SchedNoMistery*.

Para la primer prueba se creó un lote de 3 tareas de tipo `TaskCPU` con tiempos 80, 40 y 60 ciclos. Además, al scheduler, se pasaron los números 2, 3, 5, en ese orden.



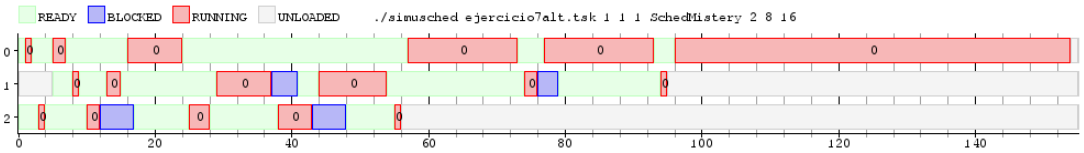
Se puede ver que se asigna un quantum de 1, luego uno de 2, uno de 8 y por último uno de 16 a cada tarea. Esto podría significar que el scheduler trabaja con 3 colas de prioridad a las que se les asigna un quantum relacionado con los parámetros que se pasaron.

Agregando un parámetro más al scheduler, se ve que hay un quantum más con distinto cantidad de ciclos a los demás (se asignan quantums de 1, de 2, 4, 8 y 16 a cada tarea).

Por ahora, asumimos que cuando un proceso inicia se lo carga en una cola (se le da la prioridad mas alta) para luego, cada vez que termina su quantum, cambiarlo a la siguiente cola (baja su prioridad en 1). Además, creemos que el quantum asignado a cada cola se relaciona con los números pasados como parámetros.

Despues de varias pruebas realizadas, las cuales reforzaron nuestras hipotesis, se decidió implementar el scheduler sin soporte para bloqueo. Cuando se logro replicar el comportamiento para tareas de uso de cpu, nos comenzamos a preocupar por las tareas bloqueantes.

Para ver como se comportan las tareas bloqueantes, se preparó un lote de tareas que consta de dos `TaskAlterno` y un `TaskCPU`



Al realizarse una operación bloqueante, se cambia el orden en que los task son ejecutados y, además, se ve que el quantum que le sigue al bloqueo es distinto al quantum que le corresponderia

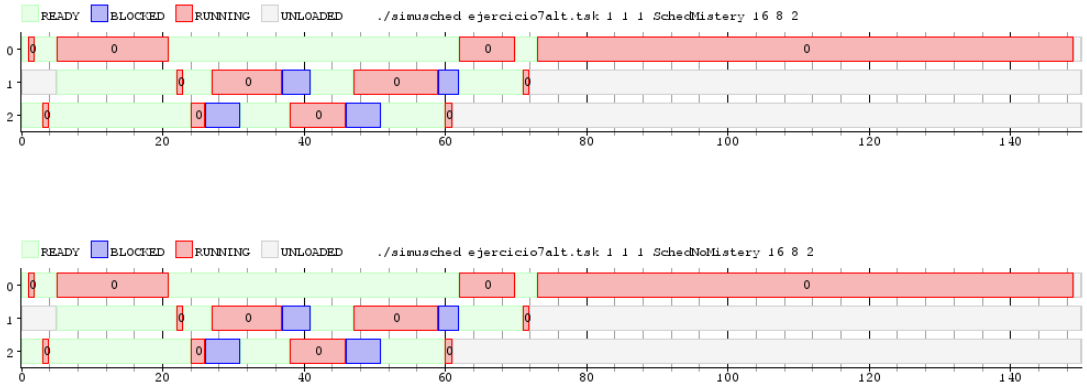
si bajase un nivel de prioridad, es más, la cantidad de ciclos totales no se corresponde a ninguno de los quantums pasados por parametros.

Sin embargo, podemos apreciar, por ejemplo en el tercer quantum del task 2, que la cantidad de ciclos corridos fue la suma del quantum de la cola de mayor prioridad con el quantum de la cola de prioridad 1 (La cola prioridad 0 tiene quantum 1, la de prioridad 1 tiene quantum 2 y el task corre durante 3 quantums).

Este razonamiento es factible debido a que, en su tercer quantum, los otros dos task ya estan colas de prioridades más bajas que 1. Lo que implicaria que tanto la cola de prioridad 0 como la de prioridad 1 están vacias. Por lo que primero se ejecutaria con prioridad 0, se pushea a la de prioridad 1 y despues, al ser la única en esa cola se vuelve a correr la misma tarea.

En resumen, tenemos un scheduler del tipo *Round Robin* con tantas colas de prioridad como parámetros le seen pasados y además, el i-ésimo parámetro se corresponde con el quantum de la cola de prioridad i. Además, el cambio de prioridades se da con la terminación de un quantum. Si el quantum de un proceso termino debido a que se realizo una llamada bloqueante se le aumenta la prioridad en 1, si es debido a la finalización correcta del quantum (se le acabaron los ciclos de reloj del quantum) se le baja la prioridad en 1.

A continuación, mostramos el resultado de correr el mismo lote de tareas con bloqueos usados en el experimento anterior, pero con distintos parámetros, corriendo en el scheduler *Mistery* y otro corriendo el scheduler *NoMistery*.



8. Ejercicio 8

8.1. Implementación

Creamos, en este ejercicio, un scheduler que no permita la migración de procesos entre cores del procesador. ¿Qué significa esto del lado de la implementación? Buscamos fijar un proceso a un core desde su inicio (load). Si un Round Robin convencional (ver ejercicio 4) usa una cola global para poder distribuir el tiempo de procesador al proceso correspondiente, nuestra implementación va a usar una cola individual para cada core de nuestro procesador. Esto fuerza a cada proceso a utilizar el core al que fue asignado.
Podríamos comparar este funcionamiento a tener un scheduler Round Robin para cada core, y el lote de tareas es totalmente independiente para cada uno.

8.2. Migración de núcleos contraproducente

A continuación se presentan gráficos generados a partir del mismo lote de tareas con algoritmos de scheduling Round-Robin:

Con un costo de migración de núcleo “realista”, es decir más grande que el costo de cambio de contexto, vemos cómo la cola global no nos permite distribuir adecuadamente el tiempo de

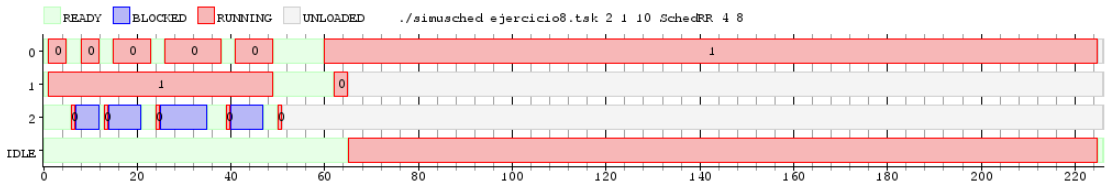


Figura 12: Round-Robin con migración de núcleos

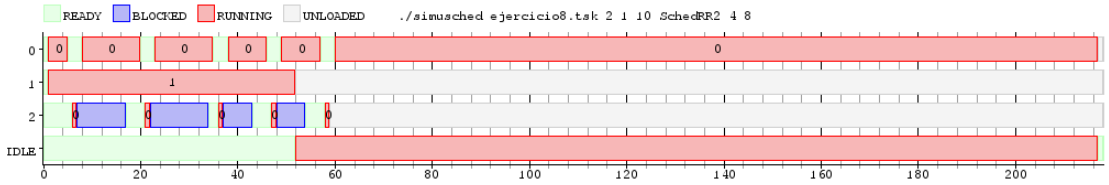


Figura 13: Round-Robin sin migración de núcleos

procesamiento a cada tarea ya que al migrar un proceso de un núcleo a otro nos cuesta más, en términos temporales. También observamos que la cola individual del nuevo Round Robin optimiza la ejecución de este lote de tareas al no permitir el tiempo muerto anterior.

Plasmamos estas conclusiones en el cálculo del tiempo de ejecución de cada tarea de nuestro lote y el throughput de cada política de scheduling:

	RR	RR2
$Task_0$	225	217
$Task_2$	65	52
$Task_3$	50	58
Throughput	8,82E-3	9,17E-3

Cuadro 5: Tiempos de ejecución

8.3. Migración de núcleos beneficiosa

Nos preguntamos entonces cuándo es ventajosa la migración de un proceso entre los núcleos del procesador. Podríamos imaginarnos situaciones reales en las cuales sea observable un cambio positivo, como por ejemplo, usando el mismo lote de tareas y siendo el único cambio el quantum asignado a cada núcleo:

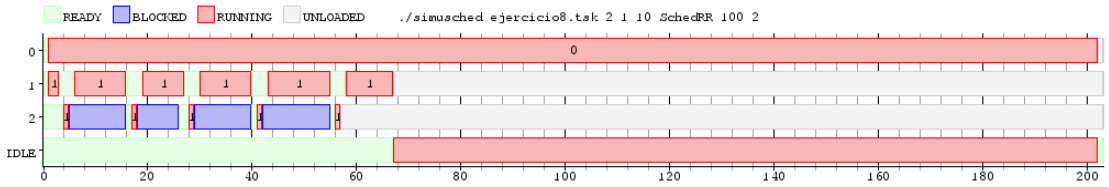


Figura 14: Round-Robin con migración de núcleos

Figura 15: Round-Robin sin migración de núcleos

Vemos claramente cómo la cola global de procesos nos posibilita ejecutar todo el lote de tareas mientras que las múltiples colas retrasan considerablemente la ejecución de una tarea. Tener quantums considerablemente distintos para cada núcleo es contraproducente en una política de scheduling sin una cola global. Desde un cuarto del tiempo de ejecución, un núcleo del nuevo Round Robin queda absolutamente abandonado. Esto repercute intensamente en nuestras mediciones, específicamente la latencia, el tiempo de espera y el tiempo de ejecución de una de las tareas.

	RR	RR2
Latencia	6	102
Tiempo de Espera	22	195
Tiempo de Ejecución	53	240

Cuadro 6: Mediciones para Task 2

Podemos deducir claramente que la migración de procesos fomenta activamente la ejecución de tareas interactivas y permite una ejecución razonable de un lote variado de tareas. Esta característica tiene que estar acompañada de diferencias importantes en el quantum para cada núcleo.