Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

Trabajo Práctico 1

Sistemas Operativos

Segundo Cuatrimestre de 2013

Apellido y Nombre	LU	E-mail
Alejandro Nahuel Delgado	601/11	nahueldelgado@gmail.com
Leandro Lovisolo	645/11	leandro@leandro.me
Lautaro José Petaccio	443/11	lausuper@gmail.com

Índice

1. Introducción

Pendiente.

2. Ejercicio 1

Pendiente.

3. Ejercicio 2

El lote de tareas utilizado para el ejercicio es el siguiente:

```
@0:
TaskCPU 1000
TaskConsola 3 100 200
TaskConsola 5 150 270
```

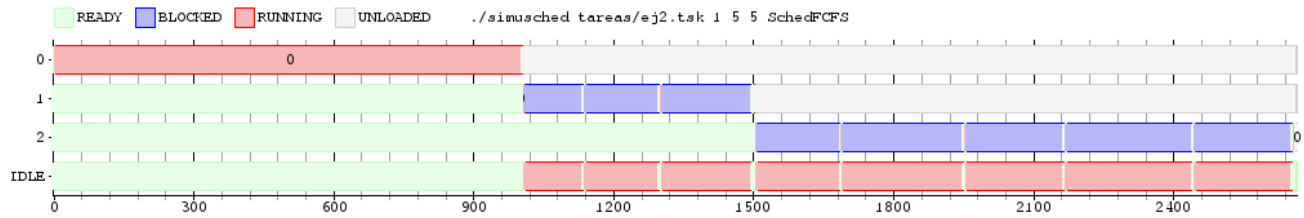

Figura 1: Scheduler FCFS corriendo el lote de tareas con 1 core y 5 de penalidad por task switch

Puede observarse como el único core del scheduler para esta ejecución del simulador corre la tarea 0 hasta que esta se termina (1000 ticks).

Luego corre la tarea 1, la cuál realiza varios bloqueos y desbloqueos. Se puede notar el tiempo dedicado a hacer el task switch entre la tarea IDLE que corre mientras la tarea principal está bloqueada y la tarea principal.

Termina su ejecución con la tarea 2 que también realiza bloqueos y desbloqueos, intercambiando entre esta y su tarea IDLE.

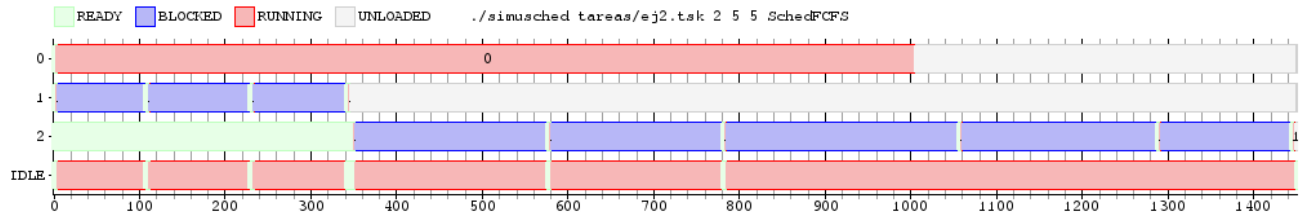

Figura 2: Scheduler FCFS corriendo el lote de tareas con 2 core y 5 de penalidad por task switch

Puede observarse como en la figura 2, relacionada a la ejecución del simulador con 2 núcleos, como cada uno se comporta adecuadamente, tomando el core 0 la tarea 0, el core 1 la tarea 1 y ejecutándola hasta su finalización. Luego, como la tarea 1 finaliza antes que la 0, el core 0 toma la tarea 2 y la ejecuta.

Observando la figura 3, puede notarse como cáda core toma cada una de las 3 tareas y las ejecuta simultáneamente hasta su finalización.

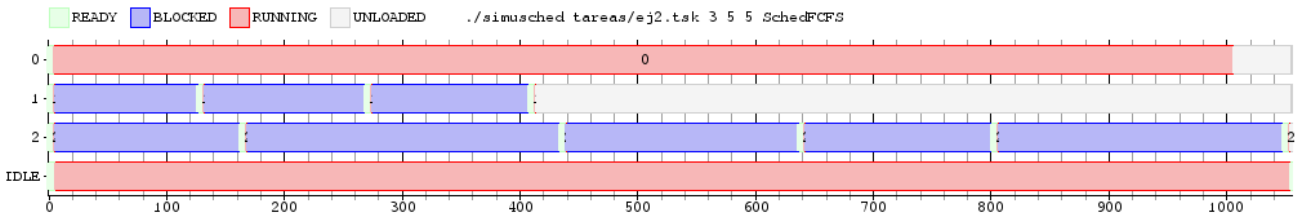

Figura 3: Scheduler FCFS corriendo el lote de tareas con 3 core y 5 de penalidad por task switch

4. Ejercicio 3

Pendiente.

5. Ejercicio 4

El lote de tareas utilizado para la figura 4 es el siguiente:

@0:
TaskCPU 200
TaskCPU 200
TaskCPU 200
TaskCPU 200

Puede observarse en la figura 4 la ejecución del simulador con el Scheduler RoundRobin con 1 core con quantum 100 y penalidad de 5 para las task switch.

El gráfico muestra como las 4 tareas se ejecutan de forma cíclica, se ejecuta la tarea 0 a la 3 (haciendo cambio de tareas debido a que se les acaba el quantum) y se repite nuevamente este ciclo. Este comportamiento cíclcio de ejecución es el correspondiente al de un algoritmo de scheduling RoundRobin.

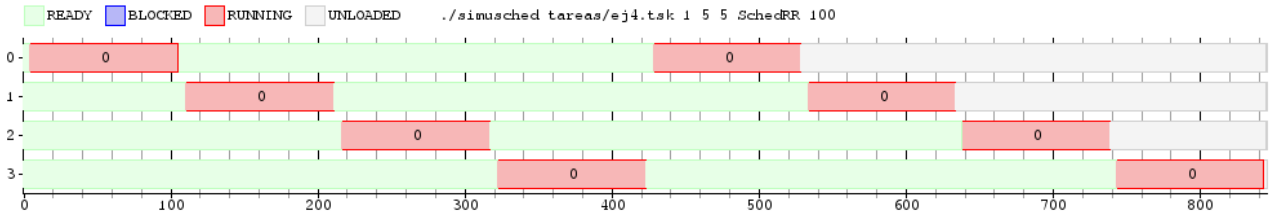

Figura 4: Scheduler RR corriendo el lote de tareas con 1 core con 100 de quantum y 5 de penalidad por task switch

El lote de tareas utilizado para la figura 5 es el siguiente:

@0:
TaskCPU 50
TaskCPU 200
TaskCPU 200
TaskCPU 200

Puede notarse nuevamente, en el gráfico de la figura 5, el cambio cíclico y ordenado de las tareas. En este caso, al tener una tarea de menor duración al inicio en el core 0, este queda asimétrico en relación al tiempo y a la ejecución de tareas, haciendo que en la próxima recorrida del ciclo del algoritmo, este tome las tareas del core 1, recibiendo una penalización de 50 ticks.

El lote de tareas utilizado para la figura 6 es el siguiente:

@0:

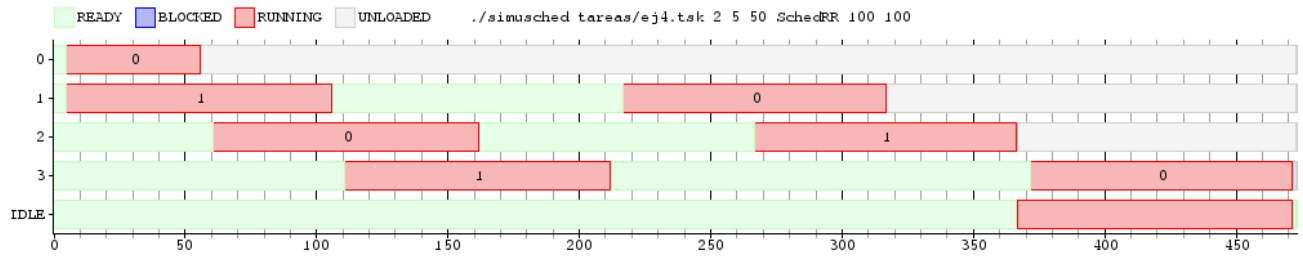

Figura 5: Scheduler RR corriendo el lote de tareas con 2 core de 100 de quantum cada uno y 50 de penalidad por task switch

TaskCPU 200
TaskIO 50 400
TaskCPU 200
TaskCPU 200

Por último, para demostrar la correctitud de la implementación del scheduler RoundRobin ante tareas bloqueantes, en la figura 6, puede verse como la tarea 1 se bloquea a los 50 ticks; el scheduler la remueve del ciclo continuando la ejecución las tareas sin tenerla en cuenta hasta que se realiza su desbloqueo, se agrega nuevamente al ciclo y termina.

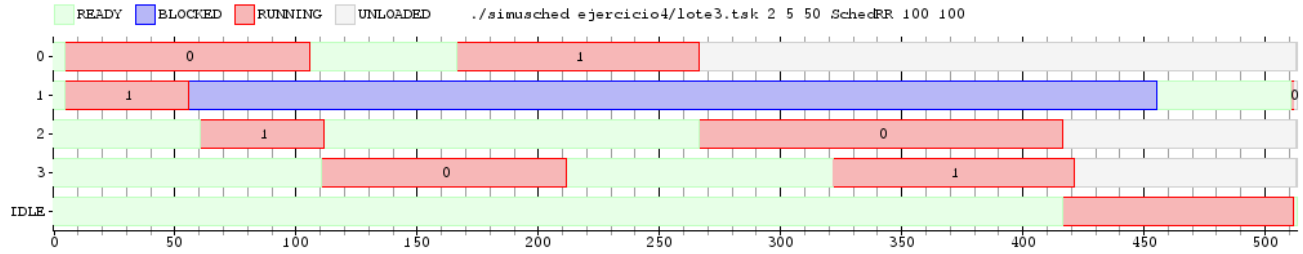

Figura 6: Scheduler RR corriendo el lote de tareas con 2 core de 100 de quantum cada uno, 50 de penalidad por task switch y una tarea bloqueante

6. Ejercicio 5

Pendiente.

7. Ejercicio 6

Pendiente.

8. Ejercicio 7

Diseñamos el siguiente lote de tareas:

```
TaskBatch 8 1
TaskBatch 8 2
TaskBatch 8 4
TaskBatch 8 8
```

Sobre el cual estudiamos las métricas a continuación:

Tiempo de ejecución Cantidad de ticks necesarios para completar todo el lote de tareas. Es el tiempo total que percibe el usuario desde que inicia el procesamiento de su lote de tareas hasta que éste termina.

Eficiencia Porcentaje del tiempo que el procesador está computando algo útil. Por algo útil se entiende un cómputo que forma parte de alguna de las tareas del lote, en lugar de uno que forma parte del scheduling; como por ejemplo cambios de contexto, migraciones entre núcleos y tiempo idle.

8.1. Experimentos

Realizamos 4 conjuntos de experimentos: con 1, 2, 3 y 4 cores. Para cada conjunto de experimentos evaluamos todas las combinaciones de quantums posibles por core en el rango $1, \ldots, 8$.

Presentamos los resultados a continuación.

8.2. Tiempo de ejecución

8.2.1. 1 core

El lote demora **65 ticks** en terminar para todos los valores de quantum en el rango estudiado.

8.3. 2 cores

El gráfico a continuación muestra el tiempo de ejecución del lote para los pares de quantums en los rangos $(1, 1 \ldots 8)$, $(2, 2 \ldots 8)$, $\ldots$, $(8, 8)$. Los presentamos agrupados de esta manera porque para cada par dentro de un mismo rango se mide el mismo tiempo de ejecución en nuestros experimentos.

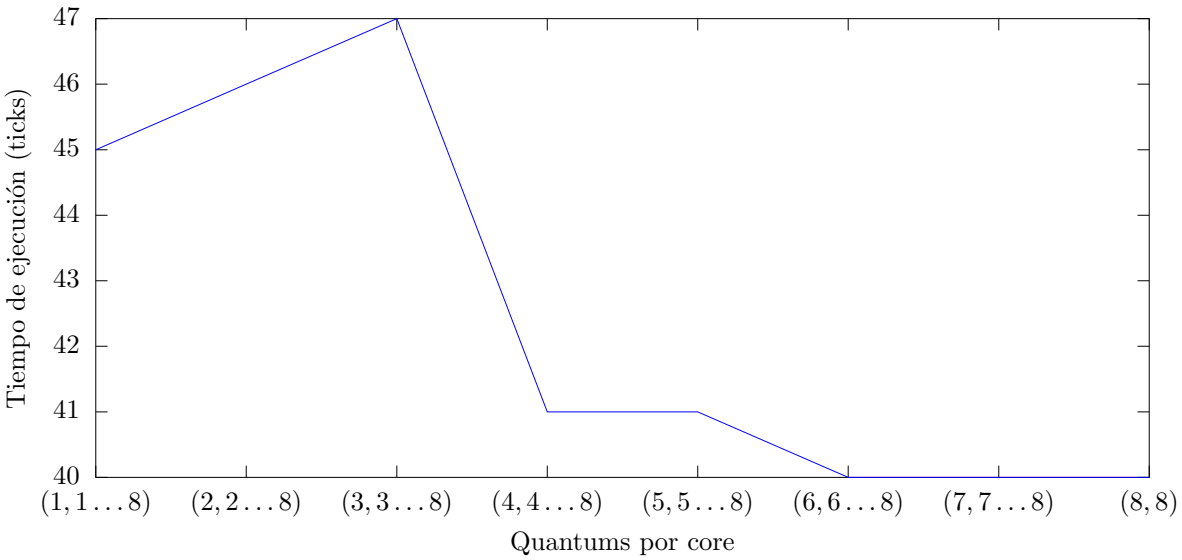

Figura 7: Tiempo de ejecución en 2 cores

8.4. 3 cores

En este caso los quantums utilizados se generaron siguiendo la secuencia $(1, 1, 1 \ldots 8)$, $(1, 2, 2 \ldots 8)$, $\ldots$, $(1, 8, 8)$, $(2, 2, 2 \ldots 8)$, $(2, 3, 3 \ldots 8)$, $\ldots$, $(8, 8, 8)$. De esta forma recorremos todas las configuraciones po-

sibles de quantums, sin evaluar combinaciones equivalentes más de una vez. [1]

El gráfico a continuación muestra los tiempos de ejecución mínimos y máximos medidos para cada rango de configuraciones $(1, *, *)$, $(2, *, *)$, ..., $(8, *, *)$, respetando las restricciones impuestas por la secuencia anterior.

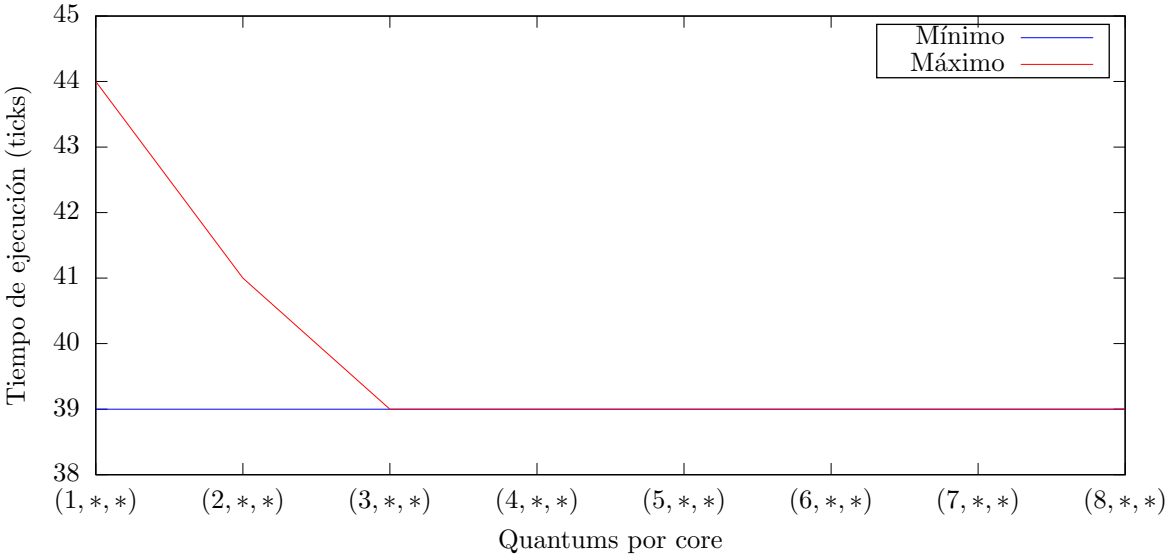

Figura 8: Tiempos de ejecución mínimos y máximos en 3 cores

9. Ejercicio 8

Consideramos como unidad de medición los ticks realizados por los procesadores para un lote de tareas. Esta elección se debe a que, para nuestro criterio, un scheduler tiene una mejor pérformance (se desempeña mejor), si logra terminar el lote de tareas en menos tiempo (ticks) que otro.

Analizando los dos algoritmos, deducimos dos hipótesis que muestran los beneficios y los contra que poseen ambos algoritmos.

La primera hipótesis es que, el algoritmo de SchedRR presentaría tiempos sin uso del procesador debido a la penalidad de cambio de nucleos, mientras que el algoritmo de SchedRR2 no los tendría debido a que este no lo permite, haciendo que necesite más ticks para completar el lote de tareas.

Como segunda hipótesis, teniendo una penalidad baja para migrar tareas de un core a otro, el algoritmo SchedRR se comportaría mejor que SchedRR2 debido a que al algoritmo SchedRR2 no le es posible migrar procesos, interrumpiendo en alguna medida el paralelismo otorgado por tener múltiples cores.

Realizamos una serie de tests para verificar esta diferencia.

Para realizar la comparación, se consiguieron 400 tiempos promedios de ejecución. Estos tiempos promedios se calcularon como la suma de la cantidad de ticks utilizados de 50 lotes de tareas dividido 50. Se utilizaron 2 cores de quantum 100 en ambos y 4 tareas *RandomTask* las cuales tienen bloqueos y consumos de CPU aleatorios (utilizando una seed para realizar el mismo experimento en ambos schedulers) en duración y en cantidad. En cada uno de los 400 promedios, se incrementó en ambos la penalidad de core switch.

Como puede observarse en la figura 7, a partir de una penalidad de 50, el SchedRR empieza a necesitar más ticks para completar las tareas que el SchedRR2, haciendo del SchedRR2 una mejor opción para el caso de penalidades altas. Sin embargo, de 50 ticks para abajo, el SchedRR realiza un mejor trabajo, terminando la ejecución del lote de tareas más rápido que el SchedRR2.

[1] Por ejemplo, la secuencia incluye la tripla $(1, 2, 3)$ pero no incluye $(2, 1, 3)$, pues estaríamos realizando dos experimentos con la misma elección de quantums pero asignados a cores distintos.

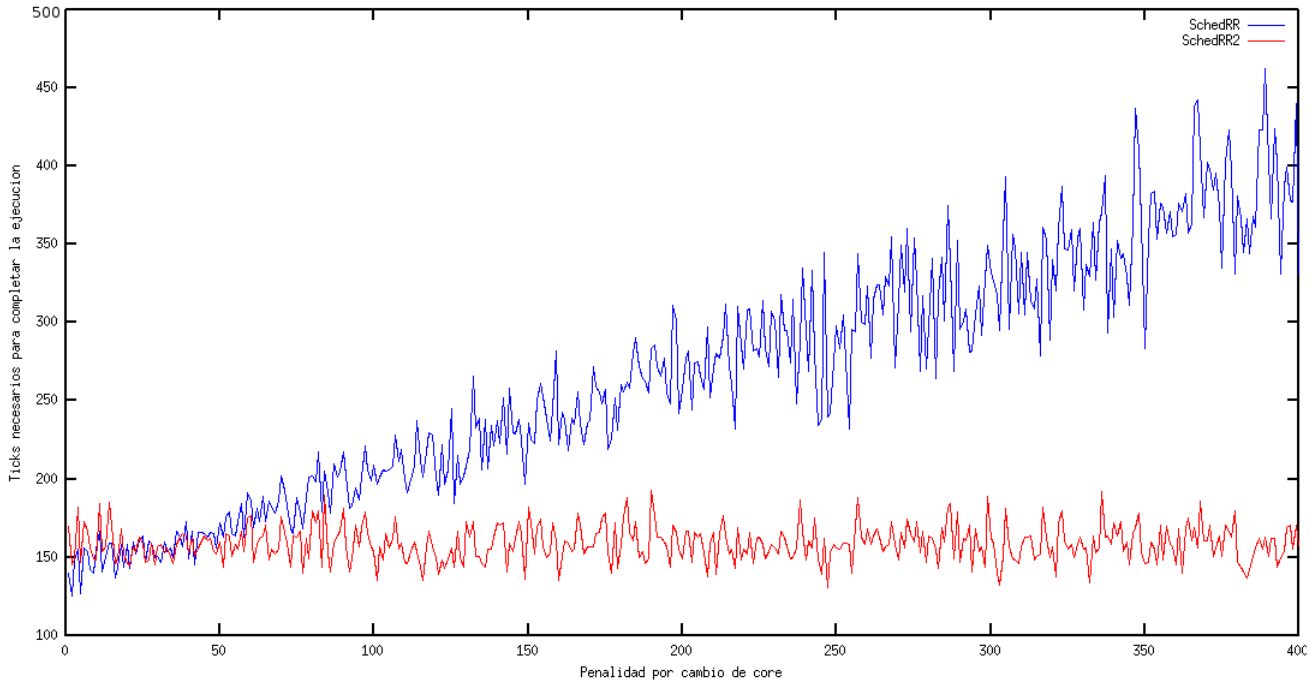

Figura 9: Comparación de ticks necesarios para ambos schedulers

En igualdad de condiciones, realizando otra vez un test aleatorio, esta vez sin incrementar el costo de migración y realizando un promedio de tiempo total, obtenemos la figura 8.

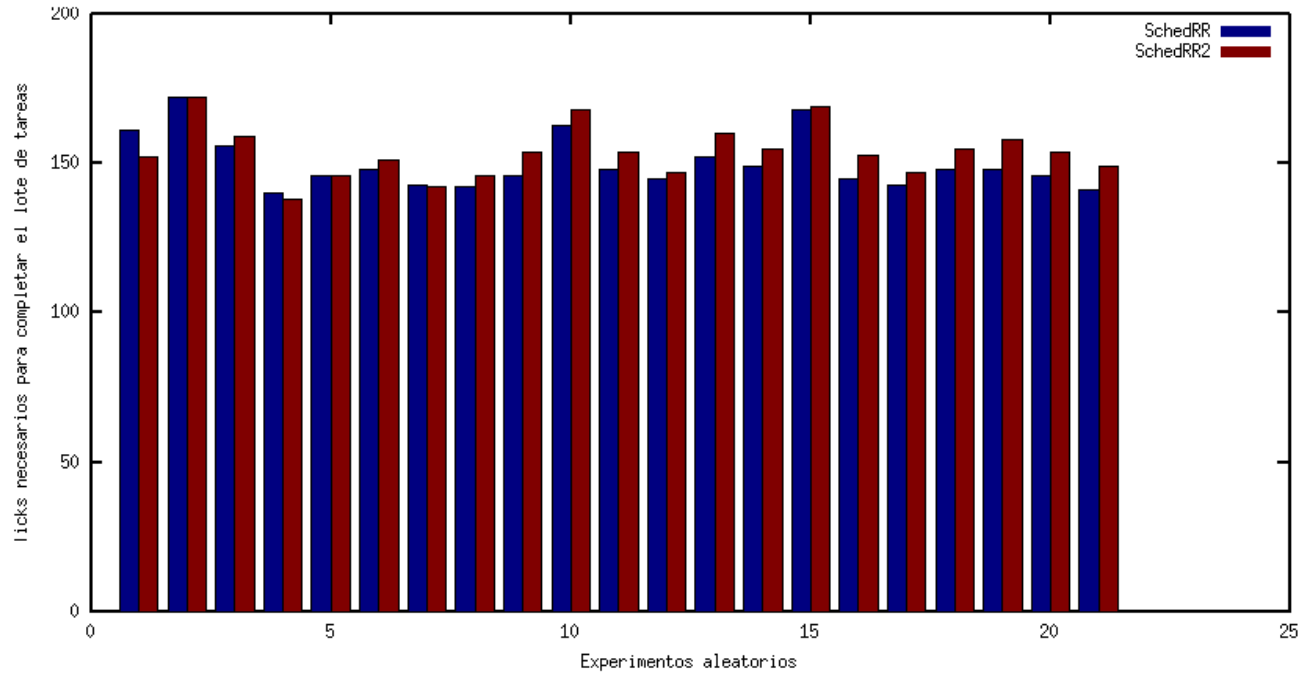

Figura 10: Comparación de ticks necesarios para ambos schedulers con igualdad de parámetros

Se puede notar, como en la mayoría de los casos, en igualdad de condiciones y para costos de migración bajos, el SchedRR tarda menos que el SchedRR2 en ejecutar un lote de tareas.

Esto se debe a que el algoritmo de SchedRR al tener una única cola de procesos, puede asignar siempre una tarea a un core libre, mientras que en SchedRR2, uno de los cores puede haber terminado de ejecutar su cola de tareas más rápido que el otro core, quedando este sin utilizarse.

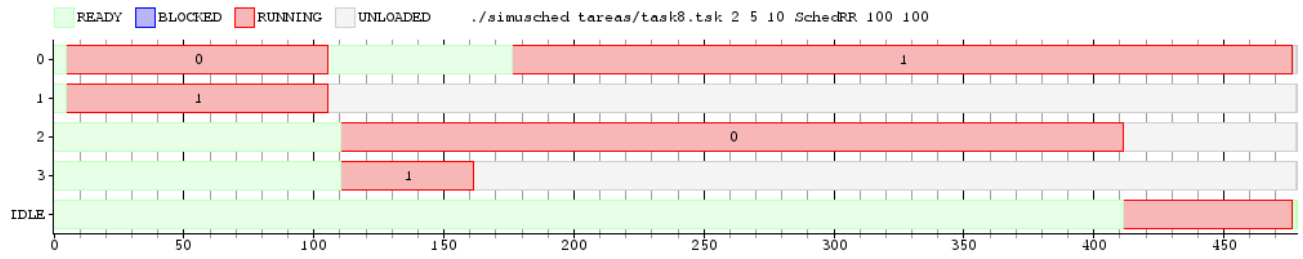

Figura 11: Ejecución de lote de tareas utilizando SchedRR

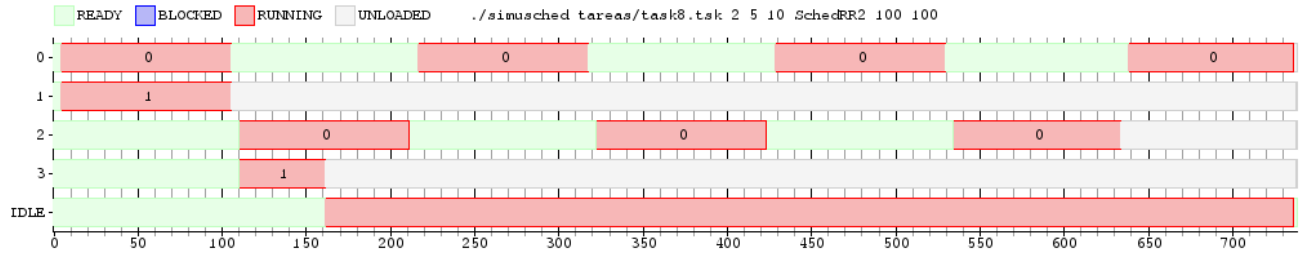

Figura 12: Ejecución de lote de tareas utilizando SchedRR2

Puede notarse la diferencia mencionada en la figura 9 y la figura 10. En la figura 9, relacionada al SchedRR, cuándo el core 1 termina de ejecutar la tarea 3, pasa a ejecutar la tarea 0, dejando que el core 0 se encargue de la ejecución de la tarea 2. Por el contrario, en la figura 10, al terminar el core 1 la ejecución de la tarea 3, este queda sin utilizarse, dejandole la tarea de la ejecución de las tareas 0 y 2 únicamente al core 0 el cuál, sin el core 1, necesita más ticks para completar estas tareas.

10. Ejercicio 9

Pendiente.

11. Ejercicio 10

Pendiente.

12. Conclusiones

Pendiente.

Apéndices

A. Apéndice

Pendiente.