


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. Ejercicio 2

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

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

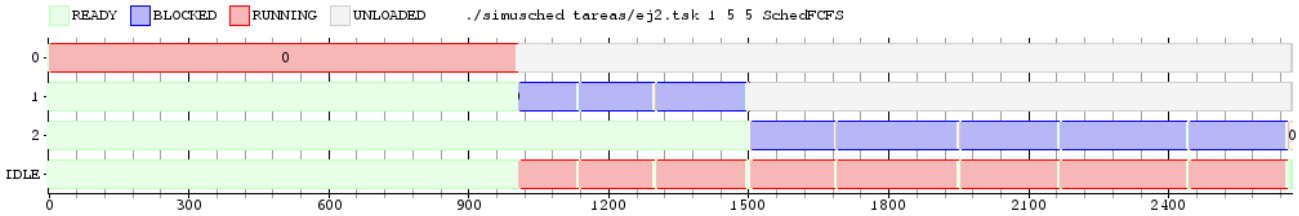


Figura 1: Scheduler FCFS corriendo el lote de tareas con 1 core y 5 de penalidad por task switch

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.

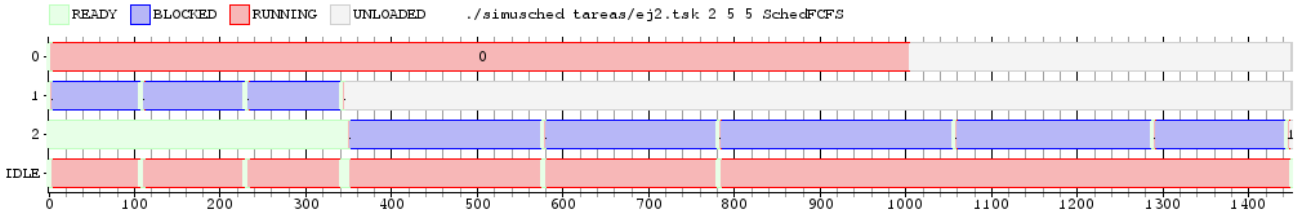


Figura 2: Scheduler FCFS corriendo el lote de tareas con 2 core y 5 de penalidad por task switch

Puede observarse como en la figura **??**, 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.

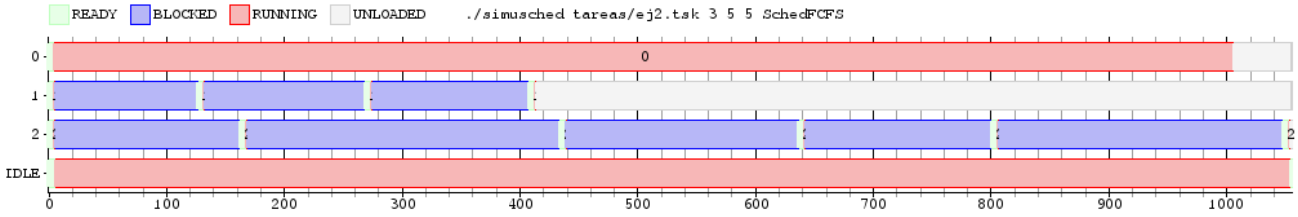


Figura 3: Scheduler FCFS corriendo el lote de tareas con 3 core y 5 de penalidad por task switch

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

## 2. 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 **??** 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.

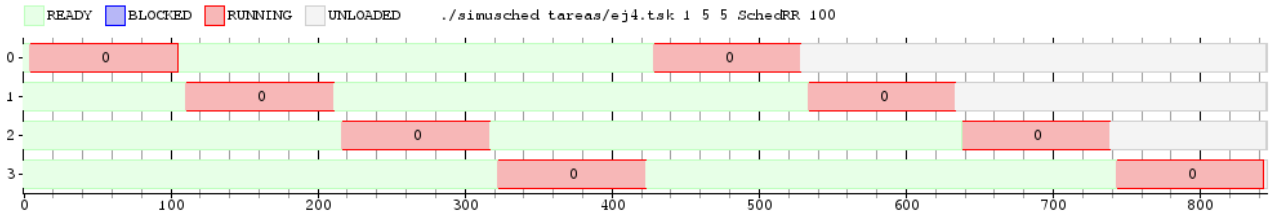


Figura 4: Scheduler RR corriendo el lote de tareas con 1 core con 100 de quantum y 5 de penalidad por task switch

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

@0:
TaskCPU 50
TaskCPU 200
TaskCPU 200
TaskCPU 200

Puede notarse nuevamente, en el gráfico de la figura **??**, 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.

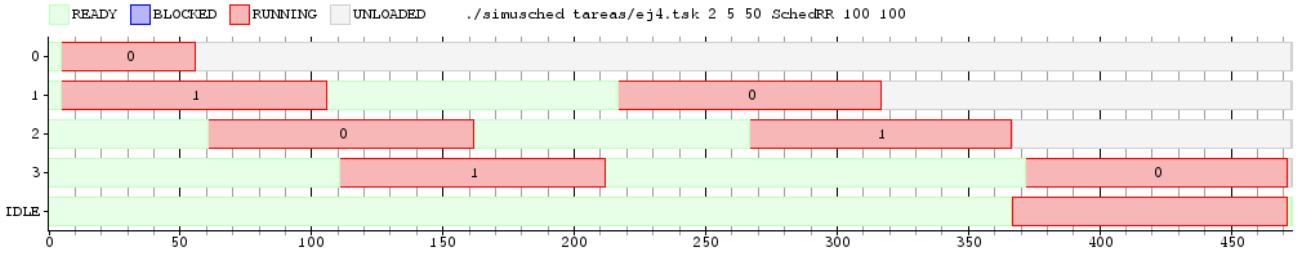


Figura 5: Scheduler RR corriendo el lote de tareas con 2 core de 100 de quantum cada uno y 50 de penalidad por task switch

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

@0:
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 **??**, 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.

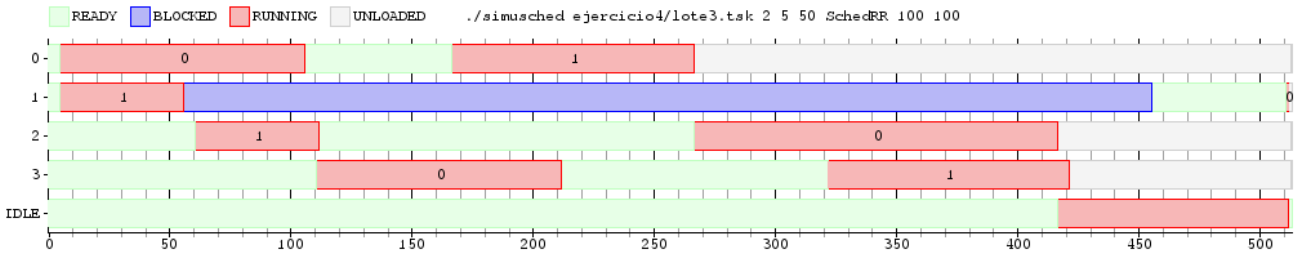


Figura 6: Scheduler RR corriendo el lote de tareas con 2 core de 100 de quantum cada uno, 50 de penalidad por task switch y una tarea bloqueante

# 3. Ejercicio 7

Diseñamos el siguiente lote de tareas:

```
*8 TaskBatch 8 1
*4 TaskBatch 8 2
*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** Relación entre los ciclos destinados a cómputos útiles y los ciclos totales utilizados hasta terminar de ejecutar el lote de tareas. Por cómputo útil se entiende un cómputo que forma parte de alguna de las tareas del lote, en lugar de uno que forma parte del scheduling (cambios de contexto, migraciones entre cores y tiempo idle).

## 3.1. Experimentos

Realizamos 4 experimentos para cada métrica: simulando 1, 2, 3 y 4 cores. Para cada experimento realizamos una simulación por cada una de las combinaciones de quantums posibles por core en el rango $1, \ldots, 8$.

Para los casos con 2 o más cores, recorremos el rango de quantums sin evaluar dos combinaciones equivalentes. [1] Esto lo logramos siguiendo las secuencias a continuación:

- Para 2 cores: $(1,1), \ldots, (1,8), (2,2), \ldots, (2,8), (3,3), \ldots, (8,8)$.
- Para 3 cores: $(1,1,1), \ldots, (1,1,8), (1,2,2), \ldots, (1,2,8), (1,3,3), \ldots, (8,8,8)$.
- Para 4 cores: $(1,1,1,1), \ldots, (1,1,1,8), (1,1,2,2), \ldots, (1,1,8,8), (1,2,2,2), \ldots, (1,8,8,8)$, $(2,2,2,2), \ldots, (2,8,8,8), (3,3,3,3), \ldots, (8,8,8,8)$.

En los resultados a continuación mostramos los valores mínimos y máximos observados para cada métrica.

[1] Por ejemplo, para el caso de 2 cores: las combinaciones de quantums $(5,10)$ y $(10,5)$ son equivalentes.

## 3.2. Tiempo de ejecución

### 3.2.1. 1 core

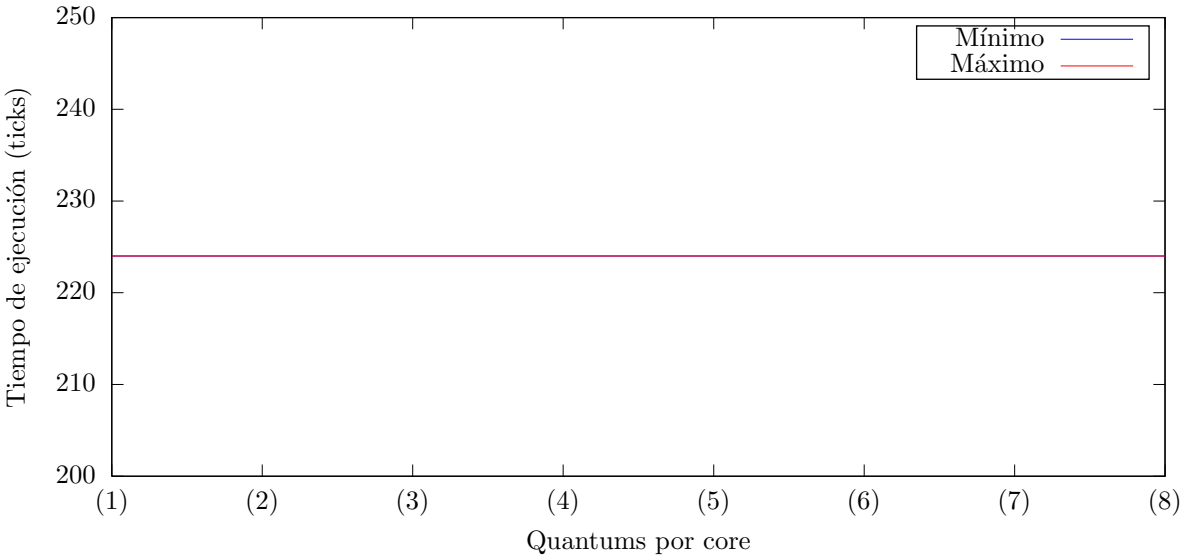


Figura 7: Tiempo de ejecución en 1 core

### 3.2.2. 2 cores

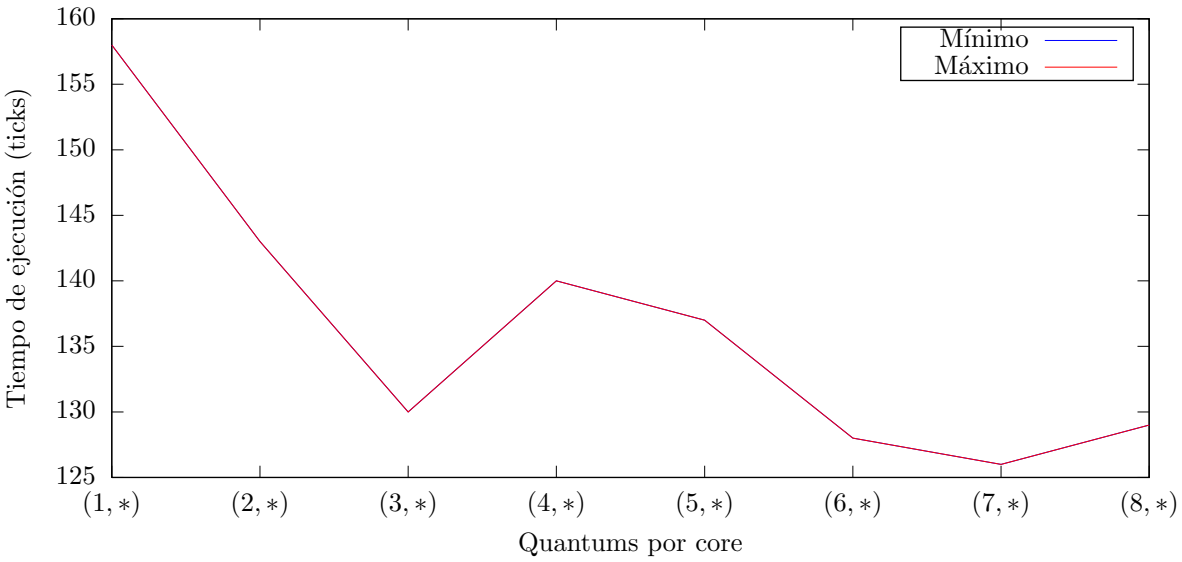


Figura 8: Tiempo de ejecución en 2 cores

### 3.2.3. 3 cores

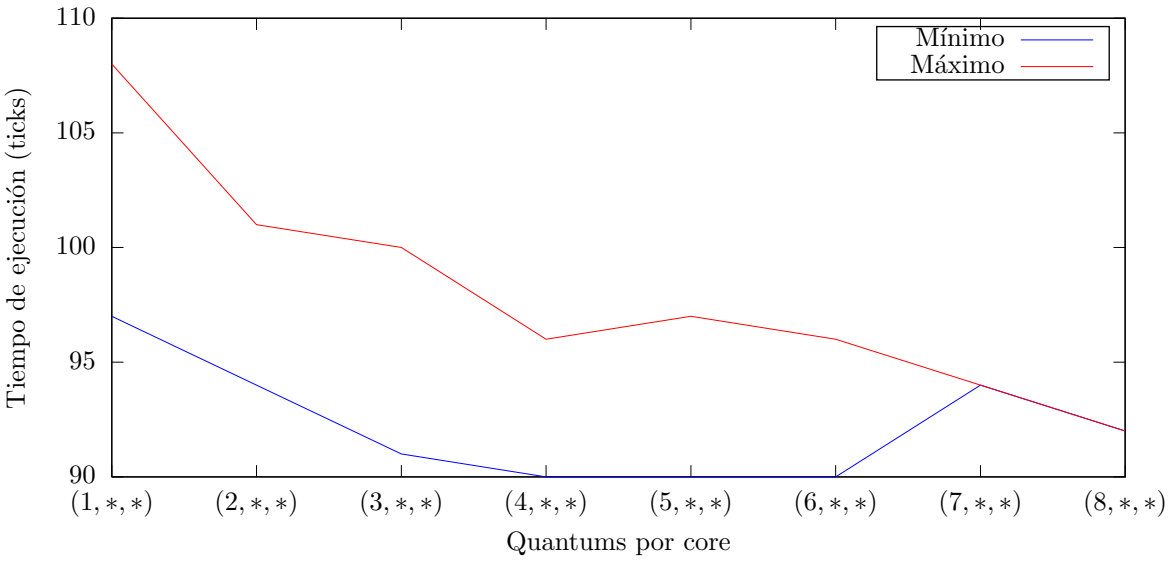


Figura 9: Tiempos de ejecución mínimos y máximos en 3 cores

### 3.2.4. 4 cores

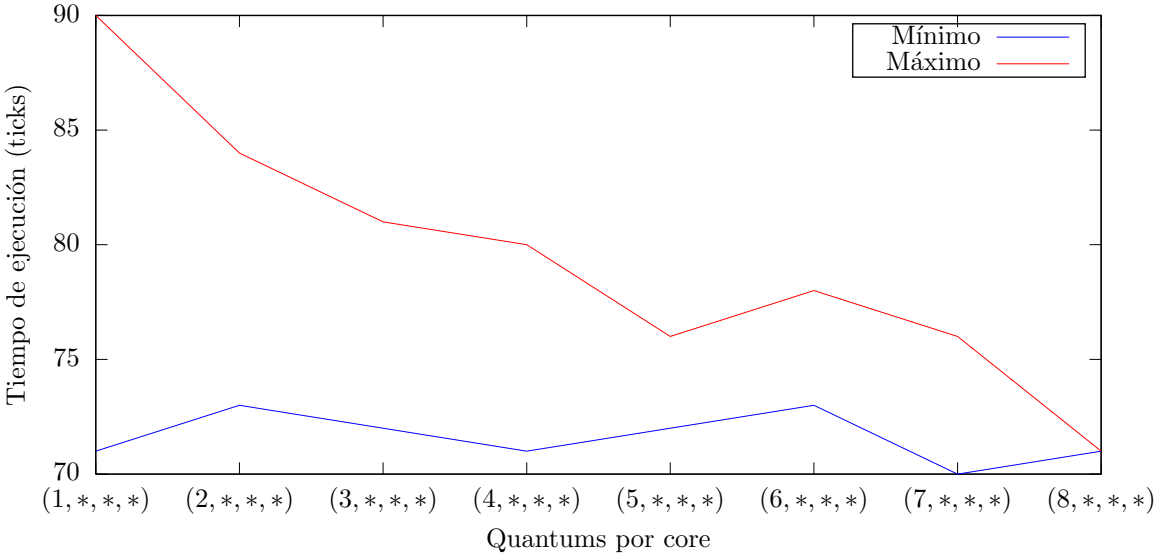


Figura 10: Tiempos de ejecución mínimos y máximos en 4 cores

## 3.3. Eficiencia

### 3.3.1. 1 core

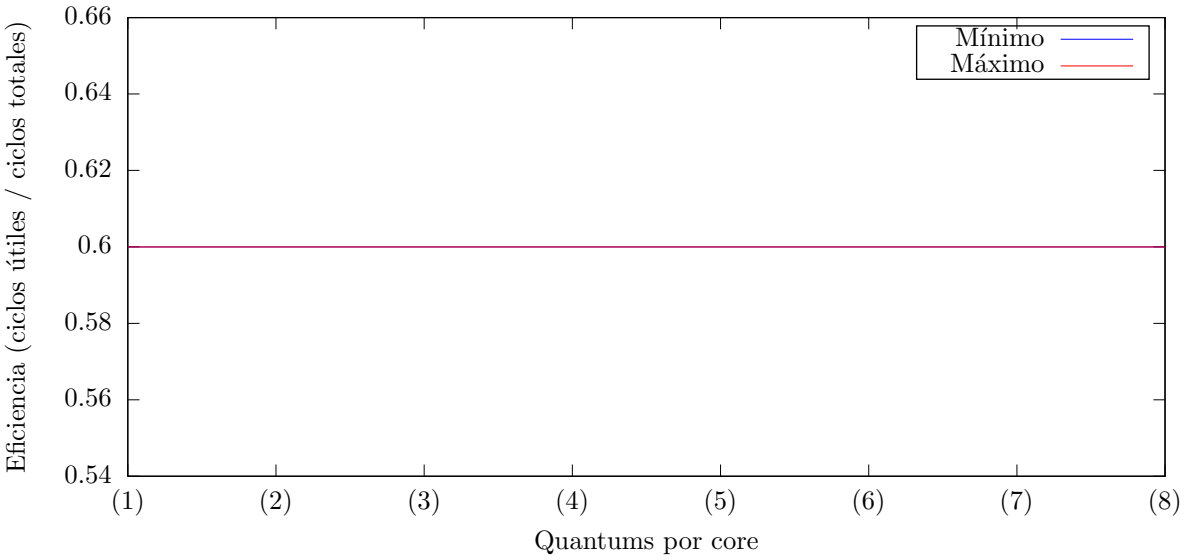


Figura 11: Tiempos de ejecución mínimos y máximos en 1 core

### 3.3.2. 2 cores

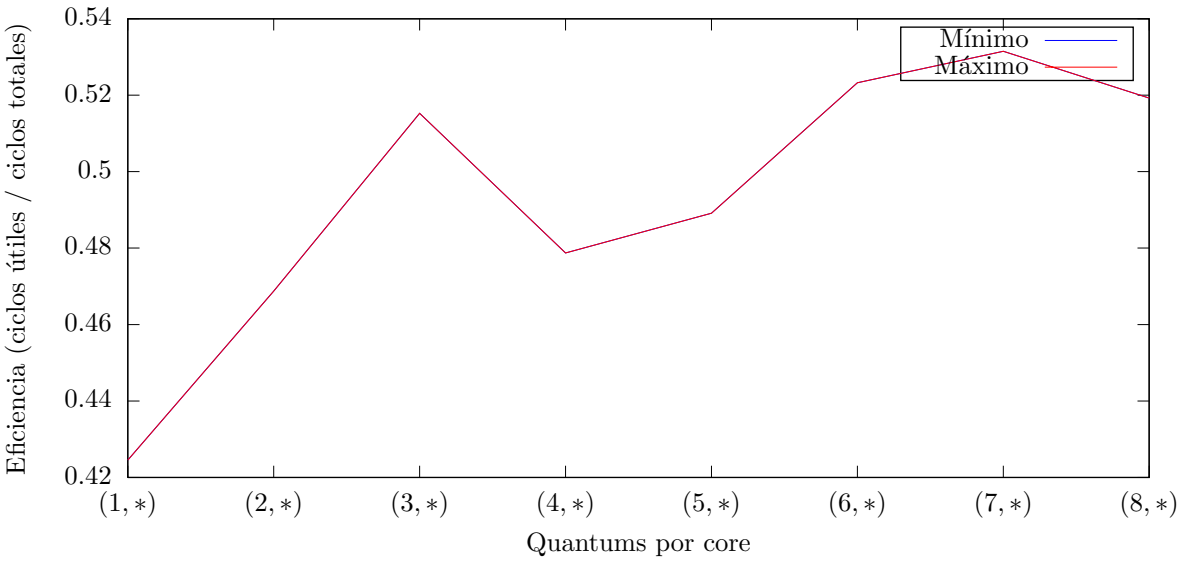


Figura 12: Tiempos de ejecución mínimos y máximos en 2 cores

#### 3.3.3. 3 cores

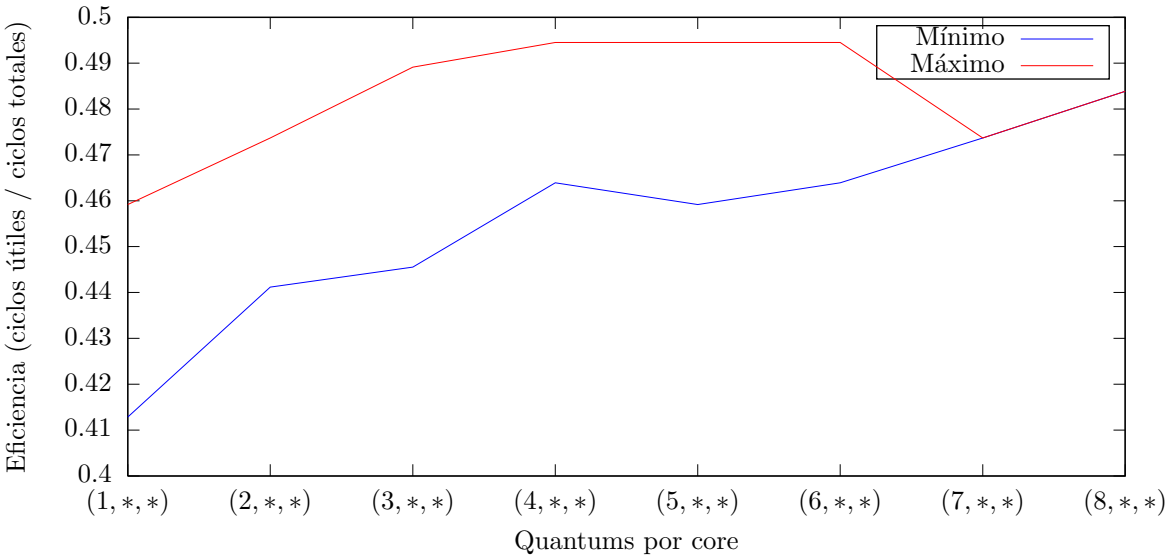


Figura 13: Tiempos de ejecución mínimos y máximos en 3 cores

#### 3.3.4. 4 cores

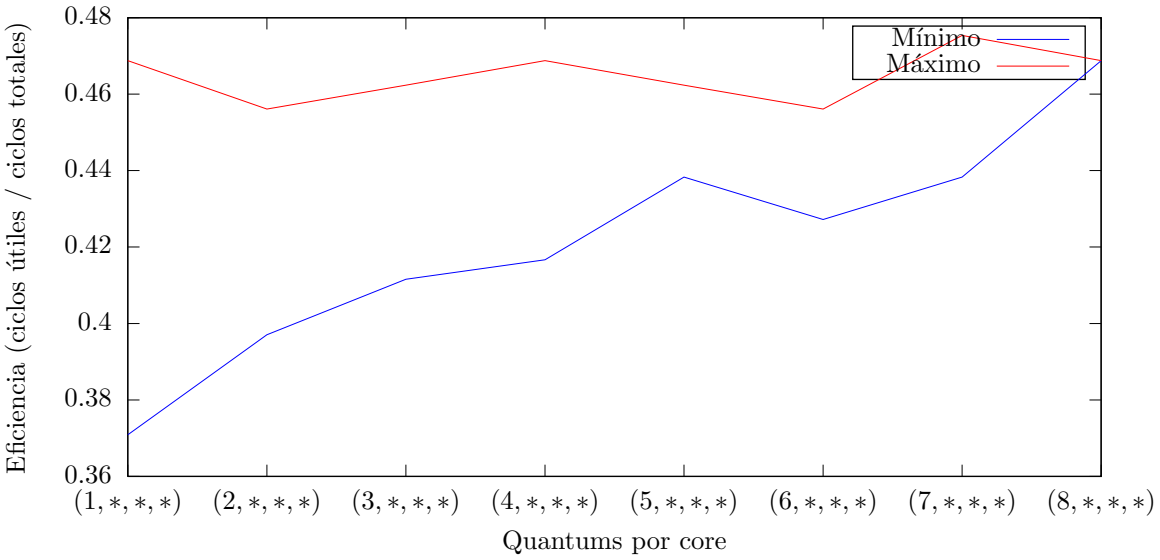


Figura 14: Tiempos de ejecución mínimos y máximos en 4 cores

### 3.4. Resultados

Los resultados a continuación son las configuraciones óptimas de quantums por cantidad de cores para cada métrica. Estas configuraciones óptimas no siempre son únicas. En caso de haber más de una, se muestra la que primero se alcanza realizando los experimentos en la secuencia descrita en la sección anterior.

Recordar que estas configuraciones son óptimas **sólo para el lote de tareas estudiado**. Para lotes distintos estos valores podrían ser subóptimos.

#### 3.4.1. Tiempo de ejecución

**1 core:** 1. Tiempo demorado: 224 ticks.

**2 cores:** $(7, 7)$. Tiempo demorado: 126 ticks.

**3 cores:** $(3, 7, 7)$. Tiempo demorado: 91 ticks.

**4 cores:** $(7, 8, 8, 8)$. Tiempo demorado: 70 ticks.

#### 3.4.2. Eficiencia

**1 core:** 1. Eficiencia alcanzada: 60 %.

**2 cores:** $(7, 7)$. Eficiencia alcanzada: 53.15 %.

**3 cores:** $(4, 8, 8)$. Eficiencia alcanzada: 49.45 %.

**4 cores:** $(7, 8, 8, 8)$. Eficiencia alcanzada: 47.53 %.

#### 3.4.3. Observaciones

Notamos que los valores de ambas métricas mejoran a medida que aumenta el quantum. El tiempo de ejecución se decrementa a medida que aumenta el quantum ya que al tener quantums más largos el procesador dedica menos tiempo a realizar cambios de contexto, y esa reducción en *overhead* se traduce en una finalización más temprana de las tareas. De manera similar, la eficiencia aumenta al incrementar la duración del quantum, ya que al producirse menos cambios de contexto, aumenta la relación entre el tiempo de procesador dedicado a ejecutar las tareas y el tiempo de ejecución total.

Observamos además que los valores mínimos y máximos registrados para ambas métricas se acercan a medida que el quantum crece (más a la derecha en los gráficos.) Esto es así debido a la forma en la que elegimos representar los rangos de quantums en los gráficos: por ejemplo, el punto $(1, *, *, *)$ agrupa valores de quantums entre 1 y 8 para todos los cores, mientras que el punto $(7, *, *, *)$ agrupa valores entre 7 y 8; entonces el primer punto agrupa mediciones mucho más variadas que el segundo y por eso tiene valores mínimo y máximo mucho más distantes entre sí que el segundo punto.

# 4. Ejercicio 8

## 4.1. Medición e hipótesis

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.

Las hipótesis son las siguientes:

1. 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.

2. 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.

## 4.2. Experimentos

### 4.2.1. Hipótesis 1

Para realizar el experimento del mal funcionamiento del SchedRR utilizando alta penalidad, 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.

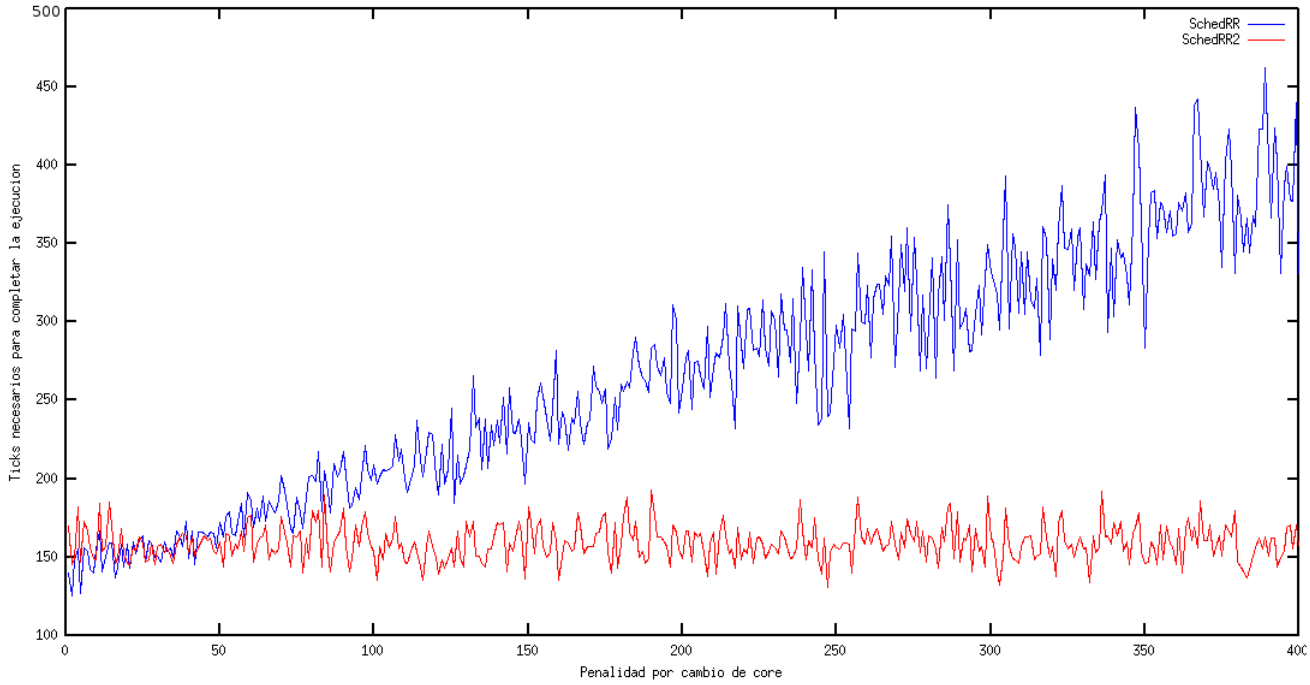


Figura 15: Comparación de ticks necesarios para ambos schedulers

Como puede observarse en la figura **??**, 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.

### 4.2.2. Hipótesis 2

Realizamos un test de entradas aleatorias en iguales condiciones que el experimento anterior, esta vez sin incrementar el costo de migración, obteniendo la figura **??**

Esta figura, muestra, en 400 pruebas realizadas, con penalidad baja de intercambio de core (10) cuántas veces un scheduler utiliza menos ticks para terminar su ejecución a comparación del otro.

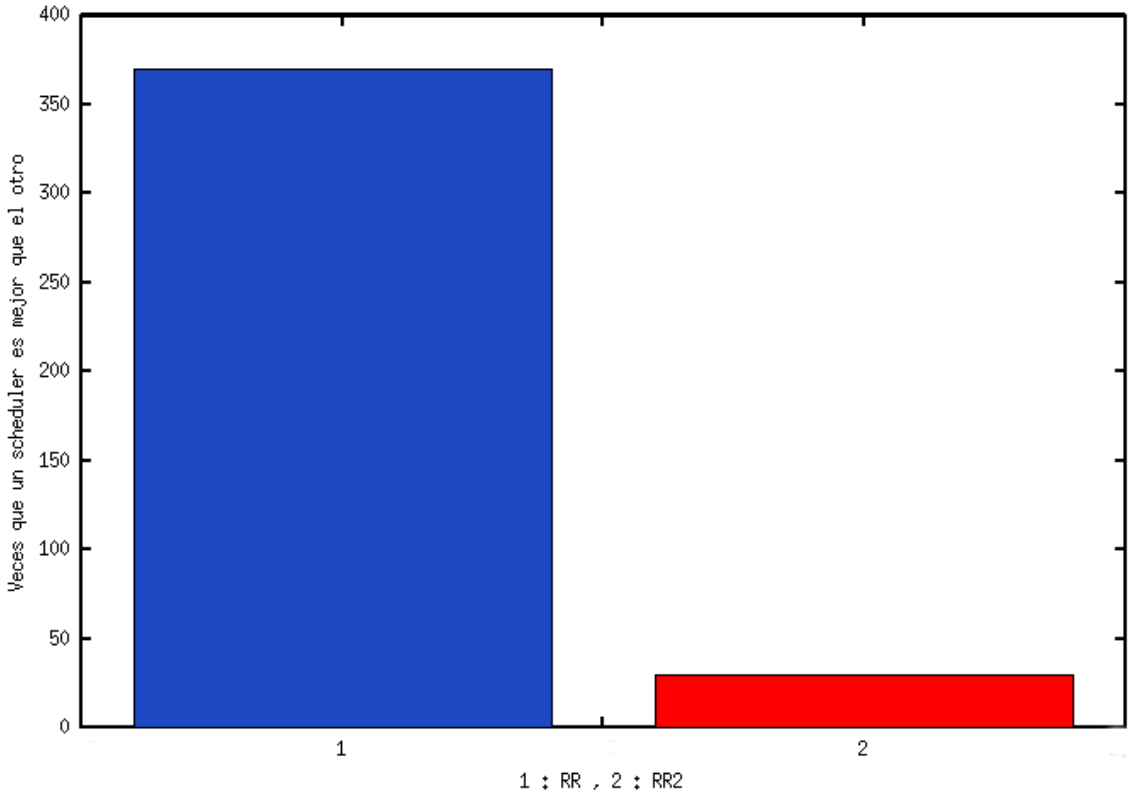


Figura 16: Comparación de efectividad de schedulers con penalidad de intercambio de core baja (10)

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.

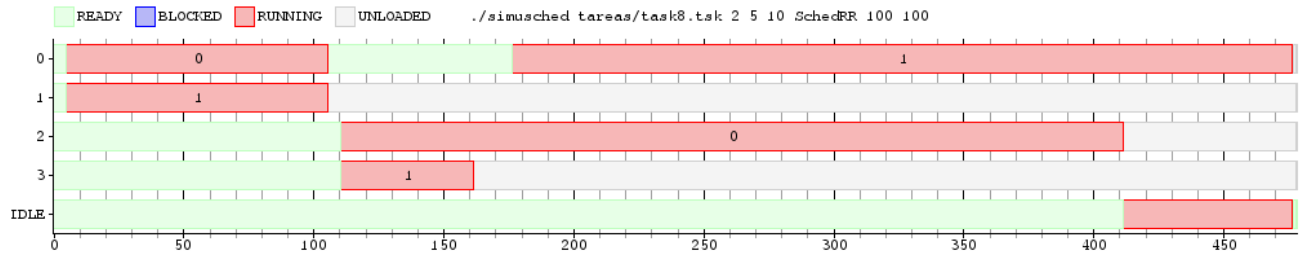


Figura 17: Ejecución de lote de tareas utilizando SchedRR

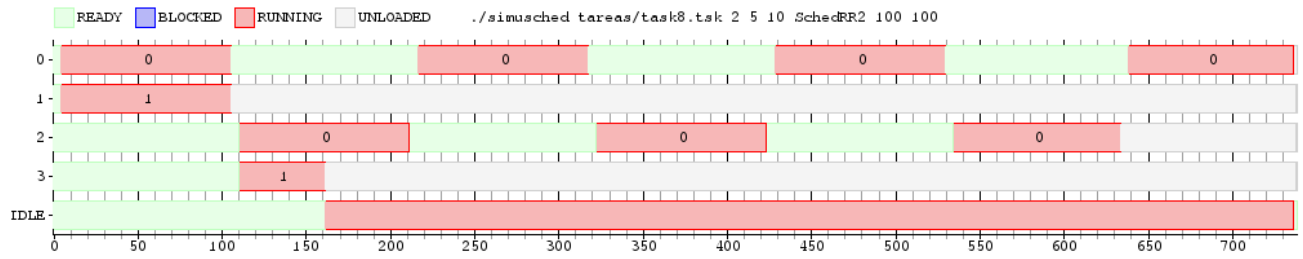


Figura 18: Ejecución de lote de tareas utilizando SchedRR2

Puede notarse la diferencia mencionada en la figura **??** y la figura **??**. En la figura **??**, 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 **??**, 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.

# 5. Ejercicio 9

Para evaluar la ecuanimidad del *lottery scheduler*, planteamos la siguiente hipótesis:

"Sea un lote de $k$ tareas idénticas que comienzan en el primer tick, sobre el que se realizan $n$ simulaciones idénticas,[2] de las cuales se observan los primeros $t$ ticks. Entonces los promedios $p_1, \ldots, p_k$ de ciclos de CPU ejecutados por cada tarea hasta el tick $t$ convergerán a alguna cantidad $p$, a medida que $n$ crece."

Es decir, $\lim_{n\to\infty} p_i \to p$, para todo $i$ entre 1 y $k$.

Si el scheduler es justo, entonces tiene sentido esperar que cada tarea reciba aproximadamente la misma cantidad de recursos luego de un número suficientemente grande de simulaciones.

Para poner a prueba nuestra hipótesis diseñamos el siguiente lote de tareas:

```
*4 TaskCPU 100
```

Luego llevamos a cabo una serie de 1000 simulaciones con los siguientes parámetros: 1 core, costo de cambio de contexto 0, quantum de 5 ticks de duración, y una semilla de secuencia pseudoaleatoria distinta para cada simulación.

En cada simulación observamos **los primeros 100 ticks** ejecutados por el simulador y extraímos la cantidad de ciclos de CPU utilizados por cada tarea.

El siguiente gráfico muestra el promedio de ciclos por tarea en función del número de simulación.

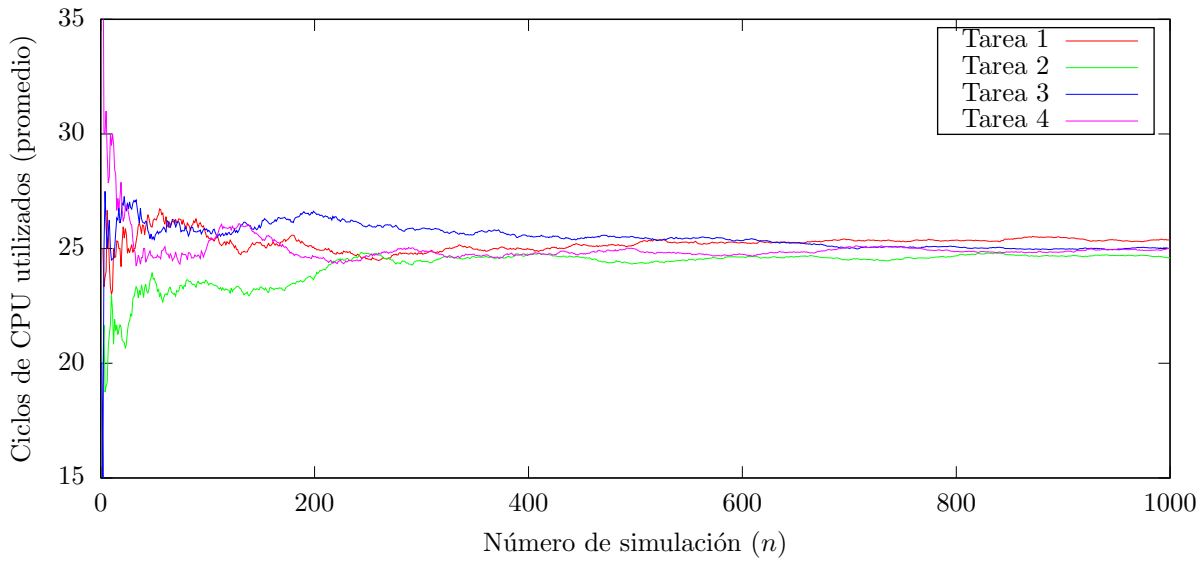


Figura 19: Ecuanimidad del *lottery scheduler*

En el gráfico se puede ver cómo los promedios por tarea tienden a un mismo valor a medida que $n$ crece. **Esto verifica nuestra hipótesis**.

[2] Utilizando este scheduler con distintas semillas y fijando el resto de los parámetros.

En particular, los promedios por tarea convergen a 25 ciclos por simulación. Esto tiene sentido, pues estamos ejecutando 4 tareas que nunca bloquean, sin costo de cambio de contexto, y como estamos observando los primeros 100 ticks simulados, el promedio esperado es de $\frac{100}{4} = 25$ ciclos por tarea.

La elección de los parámetros del experimento fue arbitraria. El número de ticks observado y la duración del quantum utilizado nos permitieron alcanzar la convergencia en una cantidad de tiempo razonable. Pueden obtenerse resultados similares con otro conjunto de parámetros luego de un número suficiente de simulaciones.

Notar que el número de ticks observado debe ser menor que la duración total de cada simulación, ya que de lo contrario el número de ciclos de CPU utilizados por cada tarea hasta ese tick será siempre el mismo (la duración de la tarea), y esto no es útil para medir la ecuanimidad del scheduler.

# 6. Ejercicio 10

Para evaluar la necesidad de la aplicación del los *compensation tickets* en el algoritmo de scheduling *Lottery* se utilizó el siguiente lote de tareas:

```
@0:
TaskBatch 100 10
*3 TaskCPU 100
```

El lote contiene tres tareas que utilizan la CPU (100 ticks) como su único recurso utilizado y una tarea que realiza 10 entradas/salidas y tiene una utilización de CPU de 100 ticks. La elección de este lote tiene como función demostrar cómo al no utilizar *compensation tickets* y al tener al menos una tarea que se bloquee, el scheduler deja de ser equitativo en la selección de tareas.

A continuación, ejemplos de la ejecución del lote con y sin compensación:

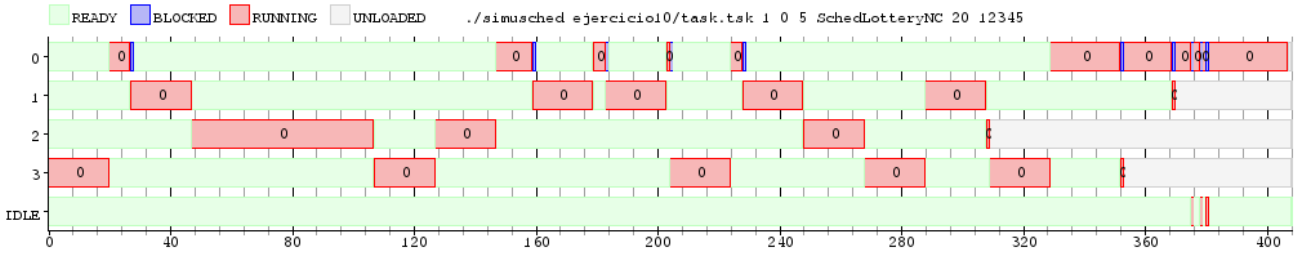


Figura 20: Ejecución de lote de tareas utilizando Lottery sin compensación

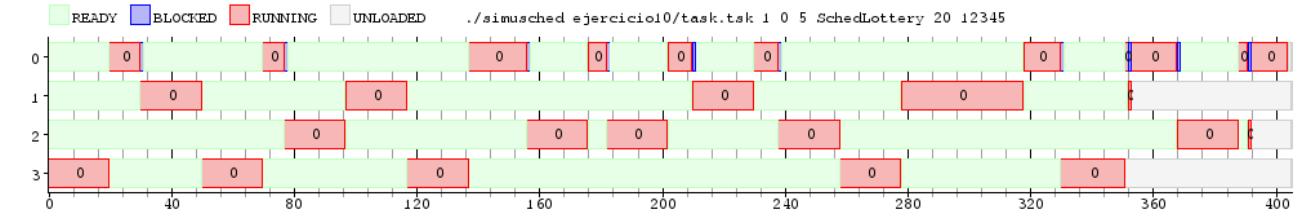


Figura 21: Ejecución de lote de tareas utilizando Lottery con compensación

Puede notarse como la ejecución del lote de tareas sin compensación produce que la tarea bloqueante (0) al bloquearse por un solo tick, pierda totalmente el quantum restante y no pueda realizar el uso del CPU necesario equitativamente en comparación a las otras tareas, lo que culmina en la tarea 0 ejecutándose casi por complejo una vez terminadas las tareas restantes.

Por otro lado, en la ejecución del lote de tareas con compensación, puede notarse como el scheduler asigna la tarea 0 para su ejecución más veces que las demás, haciendo que el uso del CPU para las tareas sea equitativo independientemente de los bloqueos que esta produzca.

Para una mejor comparación, realizamos el gráfico de la prueba de ecuanimidad realizado en el ejercicio 9.

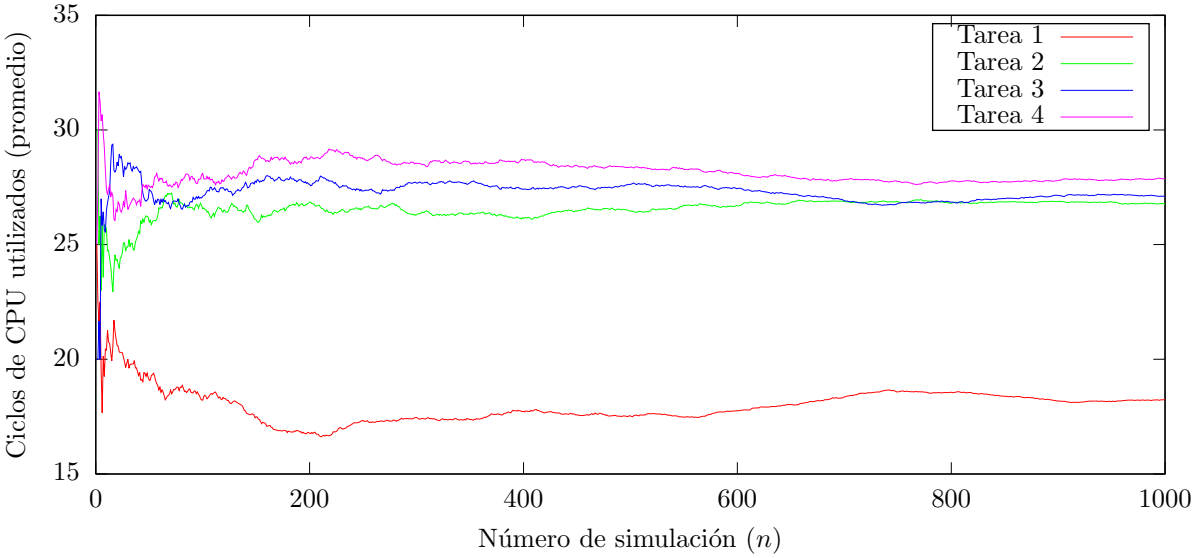


Figura 22: Test de ecuanimidad sin utilizar compensation tickets

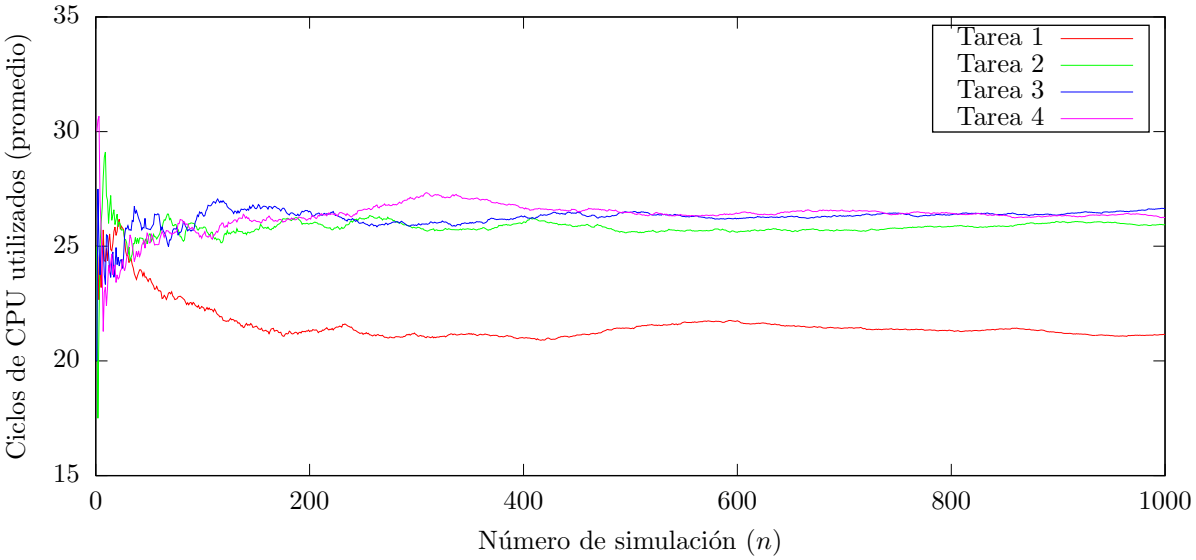


Figura 23: Pérdida de ecuanimidad utilizando compensation tickets

En ambos casos, la tarea bloqueante mantiene un promedio inferior de ciclos de ejecución en comparación a las demás tareas no bloqueantes en el lote, esto se debe a que durante la ejecución de los 100 ticks en la prueba, esta mantiene una parte de su tiempo bloqueada.

En el test de ecuanimidad utilizando *compensation tickets* puede notarse como el promedio de ciclos utilizados es mayor al promedio de ciclos utilizados sin *compensation tickets*, lo que equivale en una mayor ecuanimidad en ejecución de tareas.