



## DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

---

DC - UBA

Sistemas operativos

Trabajo Práctico N° 1

Integrante	LU	Correo electrónico
Rodrigo Kapobel	695/12	rok_35@live.com
Esteban Luciano Rey	657/10	estebanlucianorey@gmail.com
Nicolas Hernandez	122/13	nicoh22@gmail.com



**Facultad de Ciencias Exactas y Naturales**  
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

<http://www.exactas.uba.ar>

## Índice

<b>1. Ejercicio 1</b>	<b>3</b>
1.1. Descripción de implementación . . . . .	3
1.2. Gráfico de lote . . . . .	3
<b>2. Ejercicio 2</b>	<b>4</b>
2.1. 1 núcleo . . . . .	4
2.2. 2 núcleos . . . . .	4
2.3. 3 núcleos . . . . .	5
<b>3. Ejercicio 3</b>	<b>6</b>
3.1. Descripción del set . . . . .	6
3.2. 2 núcleos . . . . .	6
3.3. 3 núcleos . . . . .	6
<b>4. Ejercicio 4</b>	<b>8</b>
<b>5. Ejercicio 5</b>	<b>10</b>
<b>6. Ejercicio 6</b>	<b>12</b>
6.1. Set de tareas . . . . .	12
6.2. Resultados . . . . .	12
6.2.1. 1 - Se prioriza el menor uso de CPU . . . . .	12
6.2.2. 2 - Scheduler preemptive . . . . .	13
6.2.3. 3 - Se prioriza en base a un parámetro de prioridad . . . . .	14
6.2.4. 4 - Se prioriza el parámetro de prioridad, y luego el uso de CPU . . . . .	15
<b>7. Ejercicio 7</b>	<b>16</b>
7.1. Set de pruebas . . . . .	16
7.2. Pruebas con tareas sin bloqueo . . . . .	16
7.2.1. Lote con cpu creciente sobre 1 core . . . . .	16

---

7.2.2.	Lote con cpu creciente sobre 2 cores . . . . .	17
7.2.3.	Lote con cpu decreciente sobre 1 core . . . . .	18
7.2.4.	Lote con cpu decreciente sobre 2 cores . . . . .	19
7.2.5.	Algunas observaciones . . . . .	20
7.3.	Pruebas con tareas con llamadas bloqueantes . . . . .	20
7.3.1.	Lote con bloqueo creciente sobre 1 core . . . . .	21
7.3.2.	Lote con bloqueo creciente sobre 2 cores . . . . .	22
7.3.3.	Lote con bloqueo decreciente sobre 1 core . . . . .	23
7.3.4.	Lote con bloqueo decreciente sobre 2 cores . . . . .	24
7.4.	Resultados . . . . .	25
7.5.	Conclusiones . . . . .	25

## 1. Ejercicio 1

### 1.1. Descripción de implementación

Para implementar la funcionalidad pedida se utilizó un ciclo simple para llamar a **n** veces a la función bloqueante de duración pseudoaleatoria entre **bmin** y **bmax**

```
void TaskConsola(int pid, vector<int> params)
{
    int n = params[0];
    int bmin = params[1];
    int bmax = params[2];

    time_t t;
    srand((unsigned) time(&t));

    for(int i = 0; i < n; i++)
    {
        // llamada bloqueante con duracion random
        uso_IO(pid, (rand() % (bmax - bmin)) + bmin);
    }
    return;
}
```

### 1.2. Gráfico de lote

Para ejemplificar el funcionamiento de la tarea se utilizó el siguiente archivo de lote:

```
TaskConsola 3 1 6
@10:
TaskConsola 1 3 9
@3:
*2 TaskConsola 7 1 3
```

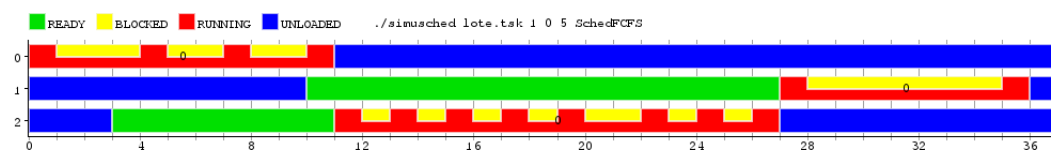


Figura 1.1: ./simusched lote.task 1 0 5 SchedFCFS

Se observa en las 3 tareas las repeticiones parametrizadas con duraciones aleatorias. La primera tarea se bloquea en rangos de 1 a 6, la segunda 1 sola vez con un rango de 3 a 9 y la última de 7 repeticiones con duracion de bloqueo entre 1 y 3.

## 2. Ejercicio 2

Dado que el algoritmo del scheduler no es preentivo, el tiempo de latencia de las tareas va a ser equivalente al del de espera total (waiting time), con lo cual se englobará ambas mediciones bajo el título de "tiempo de espera"

### 2.1. 1 núcleo

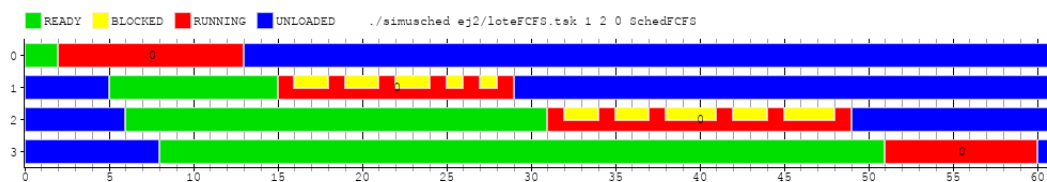


Figura 2.1: Scheduler FCFS con 1 core

**throughput:** 4 tareas terminadas en 60 ciclos = 1 tarea cada 15 ciclos  
**tiempo de espera:** promedio de tiempo en ready" de las tareas = 20 ciclos

### 2.2. 2 núcleos

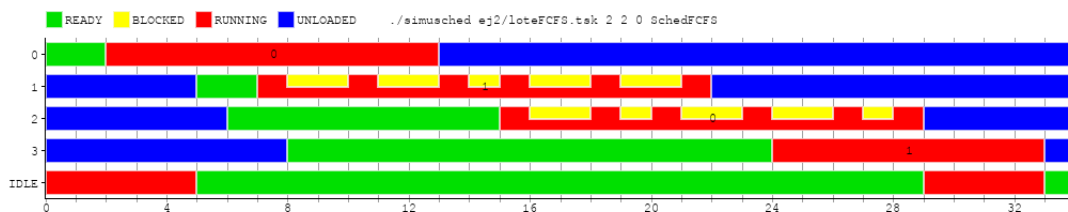


Figura 2.2: Scheduler FCFS con 2 core

**throughput:** 4 tareas terminadas en 33 ciclos = 1 tarea cada 8,25 ciclos  
**tiempo de espera:** promedio de tiempo en ready" de las tareas = 7,25 ciclos

### 2.3. 3 núcleos

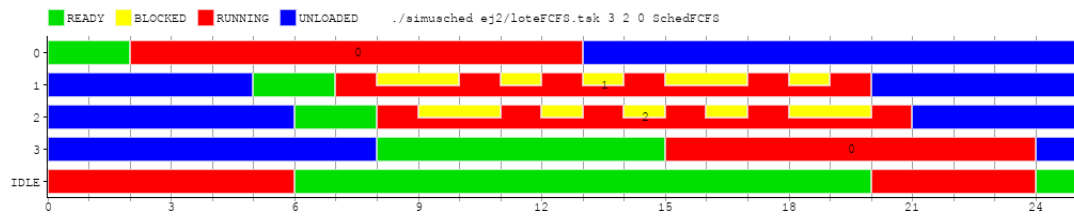


Figura 2.3: Scheduler FCFS con 3 core

**throughput:** 4 tareas terminadas en 24 ciclos = 1 tarea cada 6 ciclos  
**tiempo de espera:** promedio de tiempo en ready de las tareas = 3,25 ciclos

### 3. Ejercicio 3

#### 3.1. Descripción del set

```
TaskPajarillo 3 3 3
@10:
TaskPajarillo 5 6 1
@3:
TaskPajarillo 7 1 6
```

Se eligió un set en donde la primera tarea tuviese los mismos tiempos de bloqueo como de cpu, y las otras 2 con un bloqueo alto y cpu bajo y viceversa.

#### 3.2. 2 núcleos

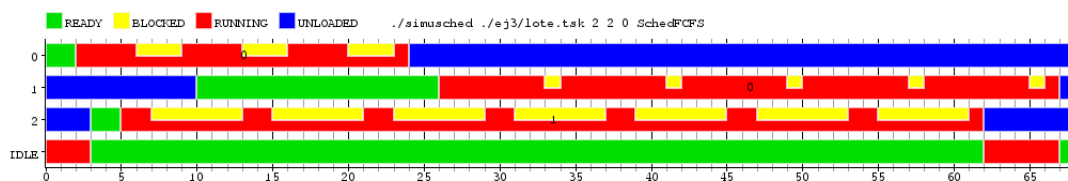


Figura 3.1: Scheduler FCFS con 2 core

**throughput:** 3 tareas terminadas en 67 ciclos = 1 tarea cada 23,333 ciclos  
**tiempo de espera:**

- Tarea 0 = 2 ciclos
- Tarea 1 = 16 ciclos
- Tarea 2 = 2 ciclos
- Promedio de tiempo de espera = 10 ciclos

#### 3.3. 3 núcleos

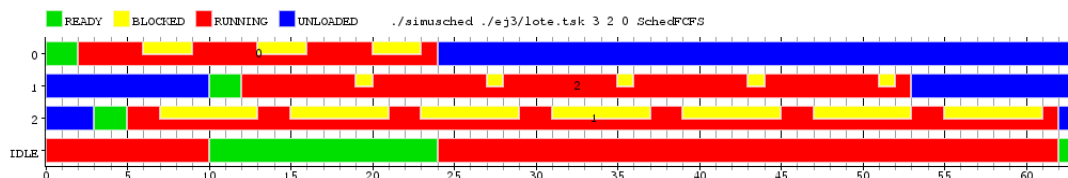


Figura 3.2: Scheduler FCFS con 2 core

**throughput:** 3 tareas terminadas en 62 ciclos = 1 tarea cada 20,666 ciclos  
**tiempo de espera:**

- Tarea 0 = 2 ciclos
- Tarea 1 = 2 ciclos
- Tarea 2 = 2 ciclos
- Promedio de tiempo de espera = 2 ciclos



## 4. Ejercicio 4

Para la implementación del scheduler Round Robin hemos decidido hacer una versión *solidaria*. Con esto queremos decir que cuando una tarea se bloquea, al finalizar este bloqueo, su quantum se reinicia. El marco de solidaridad se da porque esta tarea mencionada estaría permitiendo que otras tareas se ejecuten en su lugar, por lo cual, al recobrar todo su quantum, permitiríamos que la tarea termine lo antes posible y en el caso de que sea una tarea con muchos bloqueos, sería una decisión justa el darle el beneficio de tener más tiempo de ejecución. Por último, el quantum consumido por la salida del proceso de ejecución no se cuenta como parte del quantum del proceso.

Esto además permite una implementación más sencilla del scheduler donde solo se mantiene un quantum por core.

Para mostrar el comportamiento hemos elegido el siguiente lote de cuatro tasks pajarillo:

```
TaskPajarillo 2 1 4
TaskPajarillo 2 2 3
TaskPajarillo 2 3 2
TaskPajarillo 2 4 1
```

Como puede observarse, las tareas repiten dos veces uso de cpu y bloqueo donde el primero crece y el segundo decrece de a una unidad de tiempo.

Para estos tests se decidió no incluir tiempo por cambio de contexto ni cambio de core. Esto hará mas sencillo entender los resultados mostrados en los diagramas. Además se decide utilizar el mismo quantum para cada core con dos unidades de tiempo. Los tests son una seguidilla de ejecuciones de uno a cuatro cores con la idea de mostrar el correcto funcionamiento en la administración de los quantums por parte del scheduler.

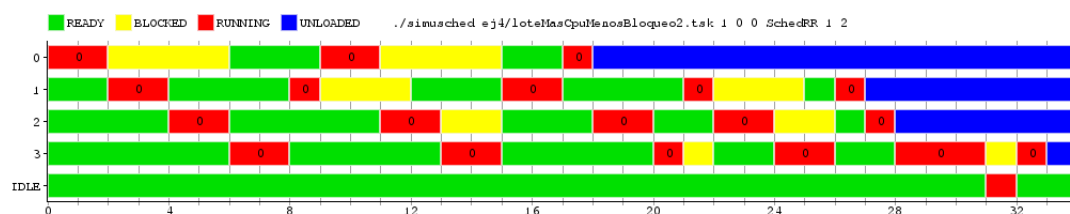


Figura 4.1: ./simusched loteMasCpuMenosBloqueo2.tsk 1 0 0 SchedRR 1 2

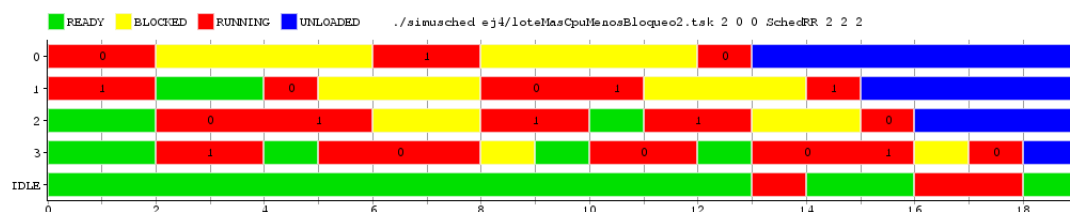


Figura 4.2: ./simusched loteMasCpuMenosBloqueo2.tsk 2 0 0 SchedRR 2 2 2

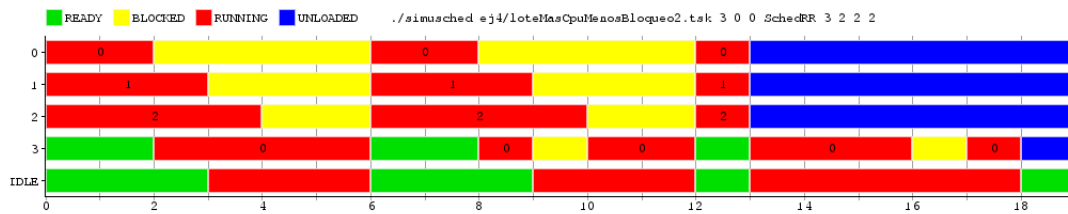


Figura 4.3: `./simusched ej4/loteMasCpuMenosBloqueo2.tsk 3 0 0 SchedRR 3 2 2 2`

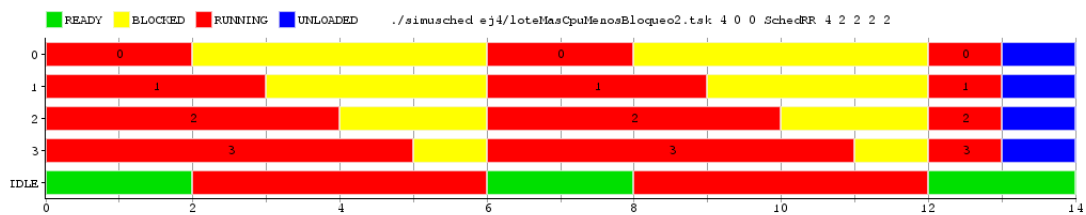


Figura 4.4: `./simusched loteMasCpuMenosBloqueo2.tsk 4 0 0 SchedRR 4 2 2 2 2`

Puede observarse que el scheduler realiza el cambio de tareas cuando el quantum es consumido. Y que cuando una tarea vuelve del bloqueo recupera todo su quantum mostrando así que se cumple la lógica *solidaria* mencionada.

A medida que se dispone de mayor cantidad de cores sucede lo esperado. Cada tarea corre en un core por separado, logrando así mayor paralelismo entre procesos y que las ejecuciones finalicen más rápido.

## 5. Ejercicio 5

Para experimentar con el **SchedMystery** se arman 3 lotes de tareas:

- **DistintasMayor**: Las tareas llegan (cronológicamente) al scheduler ordenadas de mayor a menor por tiempo de cpu que consumen.
- **DistintasMenor**: Ídem anterior pero ordenadas de menor a mayor.
- **LargaLuegoCortas**: La primera tarea en llegar al scheduler es una tarea muy larga en comparación al resto.

Los lotes son los siguientes:

- **DistintasMayor**:

```
*2 TaskCPU 5
*3 TaskCPU 3
*2 TaskCPU 1
```

- **DistintasMenor**:

```
*2 TaskCPU 1
*3 TaskCPU 3
*2 TaskCPU 5
```

- **LargaLuegoCortas**:

```
TaskCPU 50
@3:
TaskCPU 15
@10:
*2 TaskCPU 5
@15:
TaskCPU 1
```

Luego ejecutamos la simulación y realizamos los gráficos de Gantt:

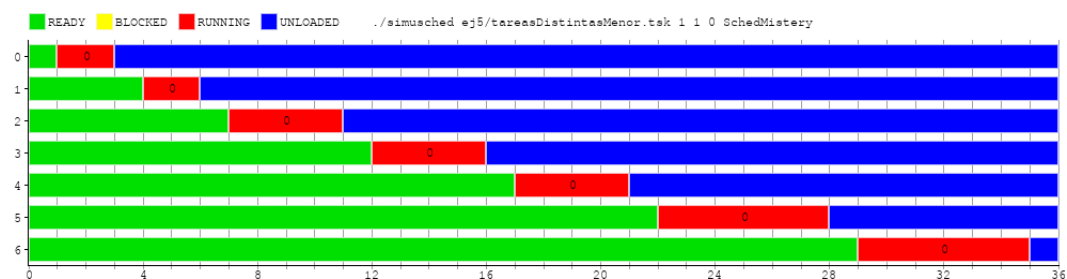


Figura 5.1: DistintasMenor

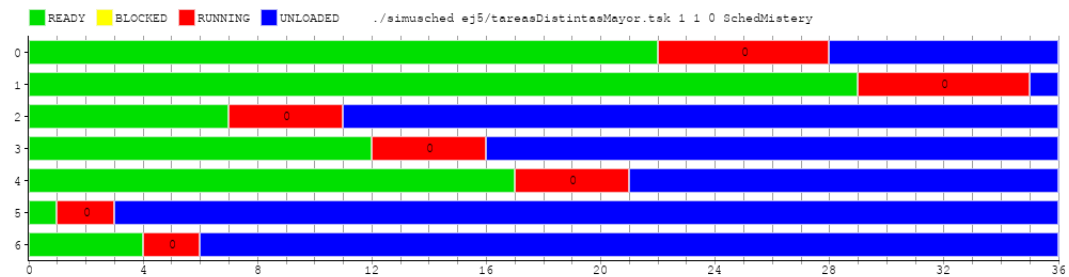


Figura 5.2: DistintasMayor

Observando los gráficos de los lotes DistintasMayor y DistintasMenor, notamos que las tareas iban siendo despachadas de acuerdo a una prioridad, siendo la misma el menor tiempo de cpu. Es decir, al momento en el que se debe decidir la próxima tarea a ejecutar siempre se elige, de las tareas en estado *ready*, aquella con menor tiempo de cpu.

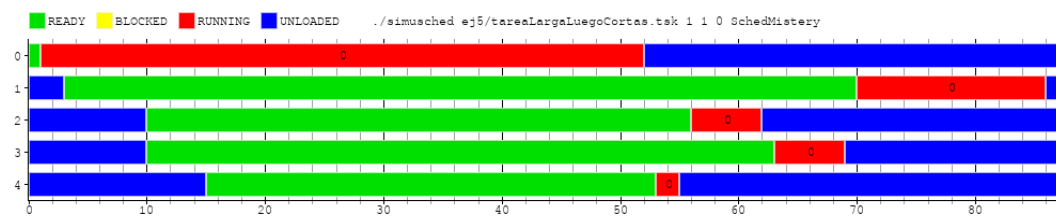


Figura 5.3: LargaLuegoCortas

Lo que concluimos a partir del gráfico del lote LargaLuegoCortas es que este scheduler no es *preemptive*: no desaloja basado en quantums como un *round robin* ni desaloja si una tarea con más prioridad llega al scheduler luego de una de menos prioridad.

Estas dos características combinadas nos permiten inferir que **SchedMystery** es un scheduler que respeta la funcionalidad de un **SJF** (*shortest job first*).

## 6. Ejercicio 6

### 6.1. Set de tareas

Para las pruebas se seleccionaron el siguiente set de tareas:

1. Determinar el tipo de ordenamiento realizado con respecto al uso de CPU por parte de las tareas.

TaskPriorizada 1 4	TaskPriorizada 1 1	TaskPriorizada 1 3
TaskPriorizada 1 3	TaskPriorizada 1 2	TaskPriorizada 1 4
TaskPriorizada 1 2	TaskPriorizada 1 3	TaskPriorizada 1 1
TaskPriorizada 1 1	TaskPriorizada 1 4	TaskPriorizada 1 2

2. Determinar si el schedduling es preemptive TaskPriorizada 1 3 @2: TaskPriorizada 1 4 @3: TaskPriorizada 1 1 @3: TaskPriorizada 1 2

TaskPriorizada 1 4	TaskPriorizada 1 3
@2:	@2:
TaskPriorizada 1 3	TaskPriorizada 1 4
@3:	@3:
TaskPriorizada 1 2	TaskPriorizada 1 1
@4:	@4:
TaskPriorizada 1 1	TaskPriorizada 1 2

3. Determinar el tipo de ordenamiento realizado con respecto a la prioridad de las tareas.

TaskPriorizada 4 1	TaskPriorizada 1 1	TaskPriorizada 3 1
TaskPriorizada 3 1	TaskPriorizada 2 1	TaskPriorizada 4 1
TaskPriorizada 2 1	TaskPriorizada 3 1	TaskPriorizada 1 1
TaskPriorizada 1 1	TaskPriorizada 4 1	TaskPriorizada 2 1

4. Determinar el tipo de ordenamiento principal y secundario en cuanto a uso de CPU y prioridad.

TaskPriorizada 3 1
TaskPriorizada 3 4
TaskPriorizada 3 7
TaskPriorizada 2 2
TaskPriorizada 2 5
TaskPriorizada 2 8
TaskPriorizada 1 3
TaskPriorizada 1 6
TaskPriorizada 1 9

### 6.2. Resultados

#### 6.2.1. 1 - Se prioriza el menor uso de CPU

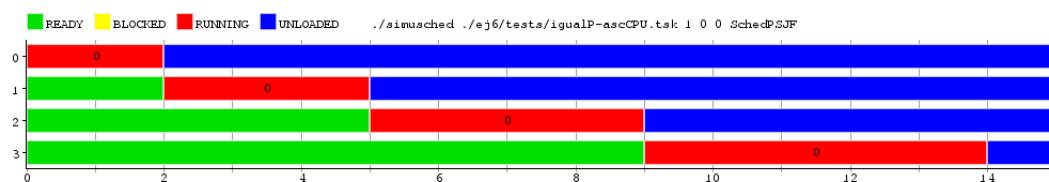


Figura 6.1: Archivo tareas ordenadas de menor a mayor uso de CPU

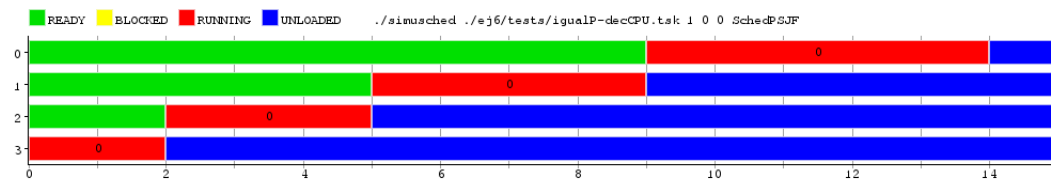


Figura 6.2: Archivo tareas ordenadas de mayor a menor uso de CPU

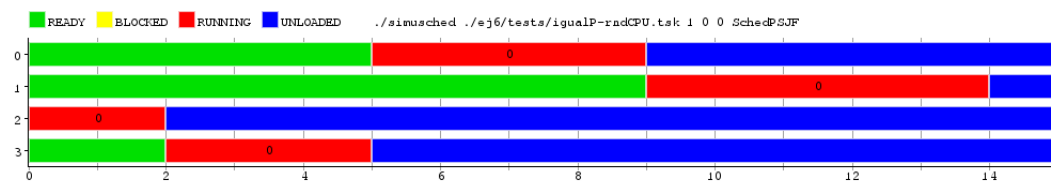


Figura 6.3: Archivo tareas desordenado respecto al uso de CPU

Podemos ver evidenciado en los gráficos que el orden en que son encoladas las tareas no influye en la decisión del scheduler sino que son ordenadas por el consumo que tenga cada una del CPU. A priori podemos decir que respeta el comportamiento esperado en una estrategia **SJF**.

### 6.2.2. 2 - Scheduler preemptive

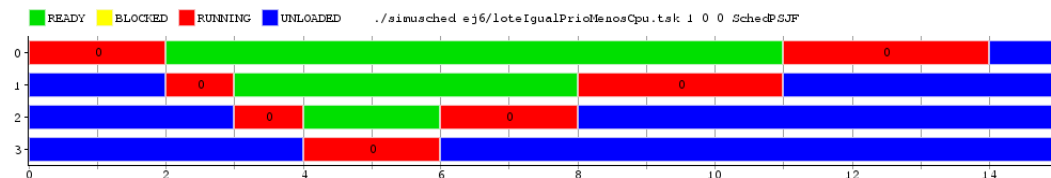


Figura 6.4: Archivo tareas ordenadas de mayor a menor uso de CPU pero con inicios desfasados

Se puede ver que a medida que entran tareas de menor uso de CPU, el scheduler decide dar lugar de ejecución a la más corta, desalojando la tarea más larga. Podríamos decir que el scheduler se comporta de forma *preemptive*. Con respecto a la acción del scheduler al tener que asignar el CPU a una tarea que anteriormente desalojó podemos observar lo siguiente:

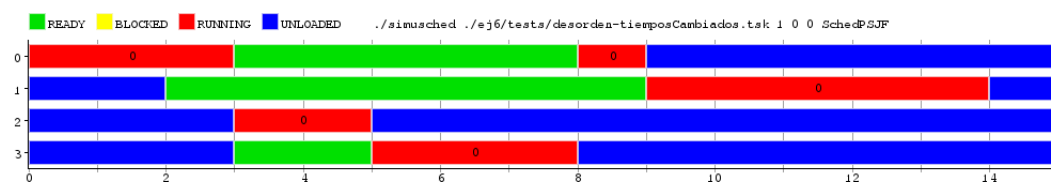


Figura 6.5: Archivo tareas con inicios desfasados sin orden específico de uso de CPU

En el gráfico podemos ver que la primer tarea es reemplazada por la tercera por menor tiempo de ejecución, y no por la segunda que entró primero. Al terminar la tercer tarea se le pasa el CPU a la tarea 4 por ser más corta que las otras 2 restantes. Al terminar la cuarta, se reanuda la primera en vez de empezar la segunda que esta en estado ready, porque es más corta en su tiempo total. En esta evaluación se puede ver que el orden de prioridad no solo se toma en el ingreso de la tarea al sistema, sino que cada vez que se elige la siguiente tarea a ejecutar, manteniendo el ordenamiento por menor uso de CPU total.

### 6.2.3. 3 - Se prioriza en base a un parámetro de prioridad

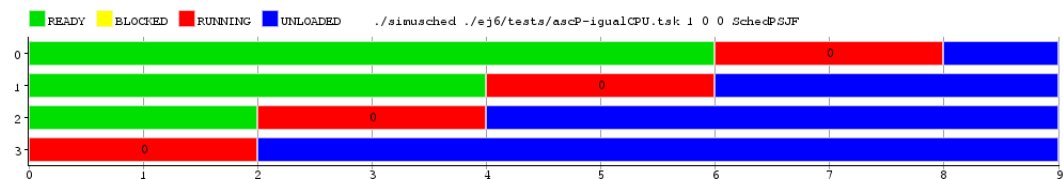


Figura 6.6: Archivo tareas ordenadas de menor a mayor prioridad

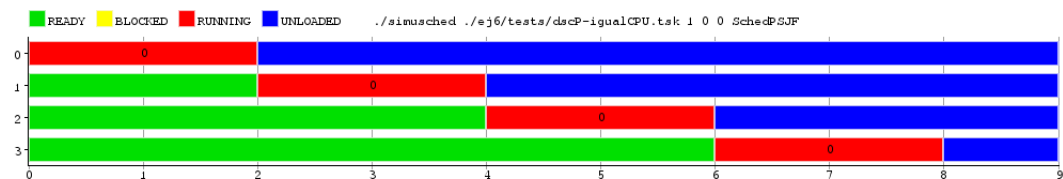


Figura 6.7: Archivo tareas ordenadas de mayor a menor prioridad

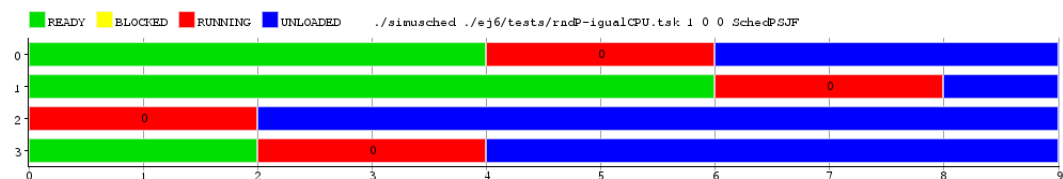


Figura 6.8: Archivo tareas desordenado respecto a la prioridad

Podemos ver evidenciado en los gráficos que el orden en que son encoladas las tareas no influye en la decisión del scheduler sino que son ordenadas por la prioridad que tenga cada una. A priori podemos decir que respeta el comportamiento esperado en una estrategia **PSJF**.

#### 6.2.4. 4 - Se prioriza el parámetro de prioridad, y luego el uso de CPU

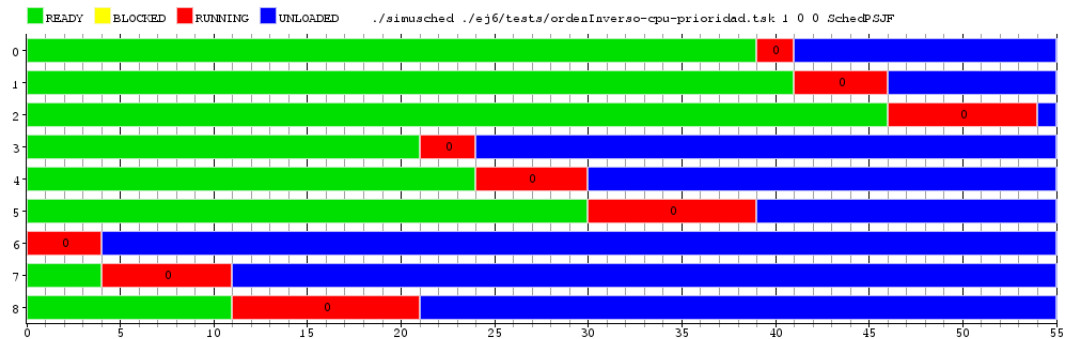


Figura 6.9: Doble ordenamiento

Corriendo el set de pruebas, podemos ver como el scheduler ordena la ejecución de las tareas primero por prioridad de parametrizada a la tarea (Se puede ver que la tarea 6 se ejecuta antes de la 3, y esta antes de la 0 a pesar de que el uso de CPU aumenta alrevéz) y luego por uso de CPU (Como ejemplificación: el set 6, 7, 8 se ejecuta en orden de uso de CPU)



## 7. Ejercicio 7

### 7.1. Set de pruebas

Para mostrar ventajas y desventajas de los schedulers, dividiremos los tests en aquellos con tareas que solamente usan cpu y aquellos con tareas que realizan llamadas bloqueantes. Todas las pruebas se realizan para 1 y para 2 núcleos.

### 7.2. Pruebas con tareas sin bloqueo

Los lotes que probaremos en esta sección son:

- Cpu creciente: Los tiempos de uso de cpu de las tareas crecen en el orden en el que ellas aparecen en el lote.
- Cpu decreciente: Los tiempos de cpu decrecen en el orden de llegada al scheduler de las tareas.

También notaremos que, debido a que PSJF es un scheduler que usa tareas con prioridades, necesitamos usar tareas con prioridades en los lotes de pruebas del mismo. Por eso, en los gráficos se observaran dos clases:

- PSJF creciente: La prioridad de las tareas crece en el orden de llegada al scheduler de las mismas.
- PSJF decreciente: La prioridad de las tareas decrece en el orden de llegada al scheduler de las mismas.

No vamos a testear el caso en el cual las prioridades son las mismas para todas las tareas, ya que al tener todas las tareas la misma prioridad, desempatan por tiempo de cpu y se observa un comportamiento muy similar a un scheduler *SJF*.

En los siguientes gráficos para un lote de tareas se observan la latencia, el tiempo de espera y el tiempo de completación de cada tarea y luego se toma el promedio de ellas. También se calcula el *throughput*.

#### 7.2.1. Lote con cpu creciente sobre 1 core

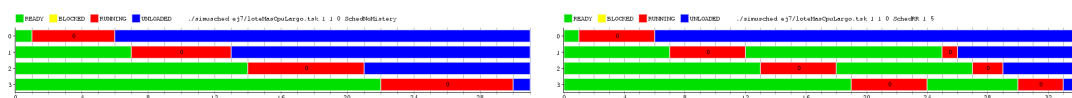


Figura 7.1: No Mistery y Round Robin

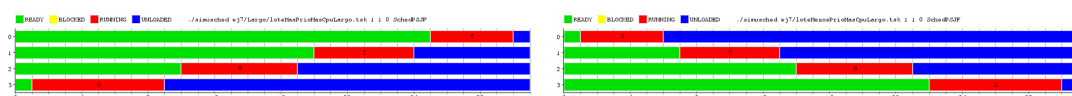


Figura 7.2: PSJF creciente y decreciente en prioridad

- Throughput NoMystery: 0.1333333
- Throughput Round Robin: 0.166666666666667
- Throughput PSJF creciente: 0.133333333333333
- Throughput PSJF decreciente: 0.133333333333333

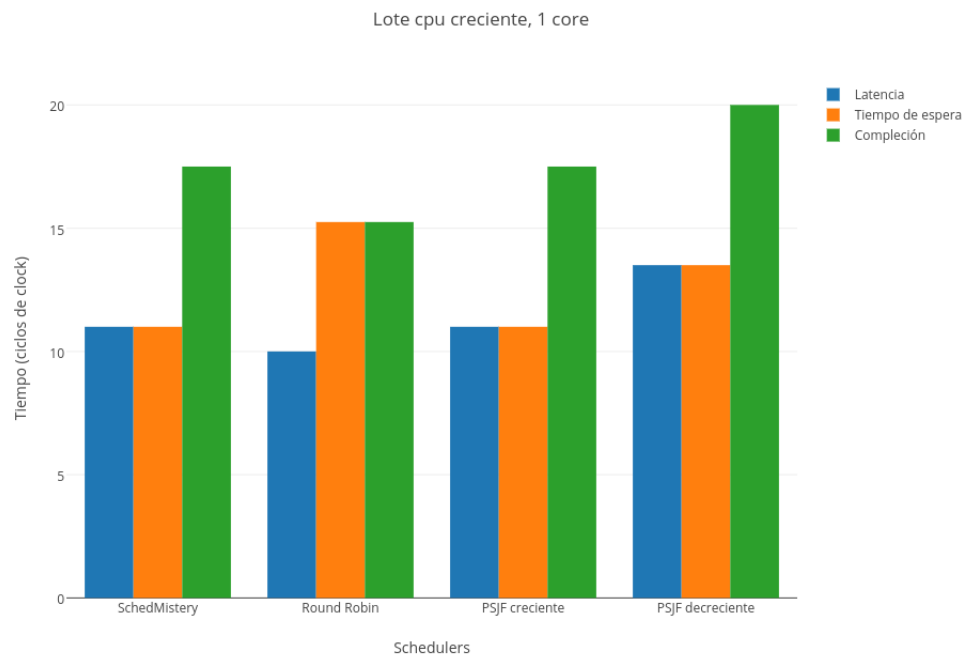


Figura 7.3: Comparación de métricas

## 7.2.2. Lote con cpu creciente sobre 2 cores

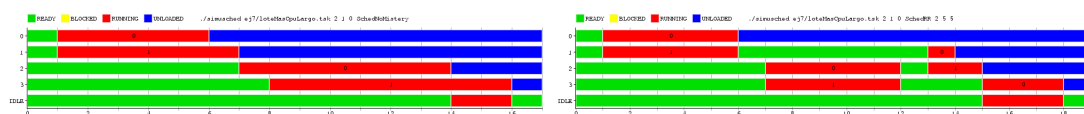


Figura 7.4: No Mystery y Round Robin

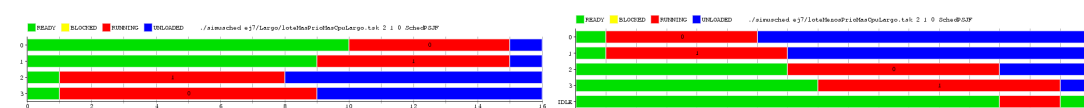


Figura 7.5: PSJF creciente y decreciente en prioridad

- Throughput NoMystery: 0.25

- Throughput Round Robin: 0.22222222222222
- Throughput PSJF creciente: 0.266666666666667
- Throughput PSJF decreciente: 0.25

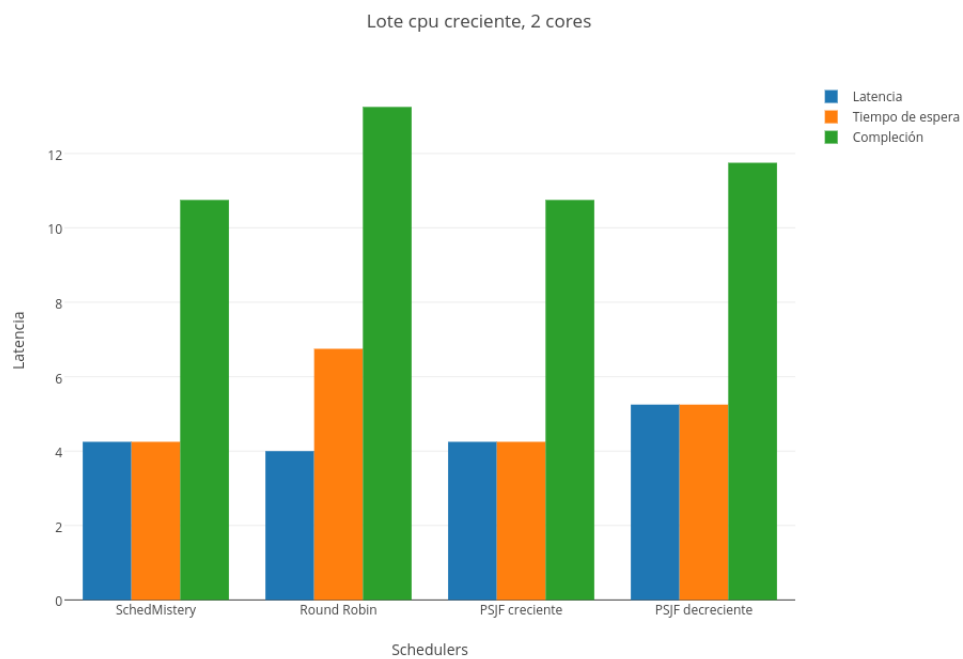


Figura 7.6: Comparación de métricas

### 7.2.3. Lote con cpu decreciente sobre 1 core

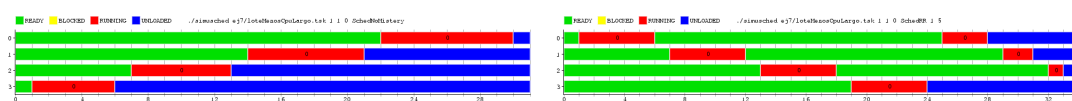


Figura 7.7: No Mystery y Round Robin

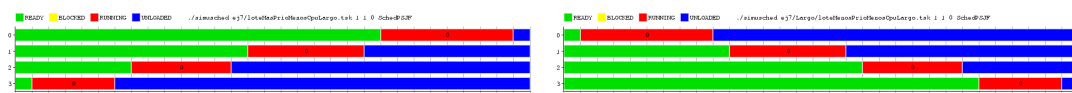


Figura 7.8: PSJF creciente y decreciente en prioridad

- Throughput NoMystery: 0.133333333333333
- Throughput Round Robin: 0.121212121212121

- Throughput PSJF creciente: 0.133333333333333
- Throughput PSJF decreciente: 0.133333333333333

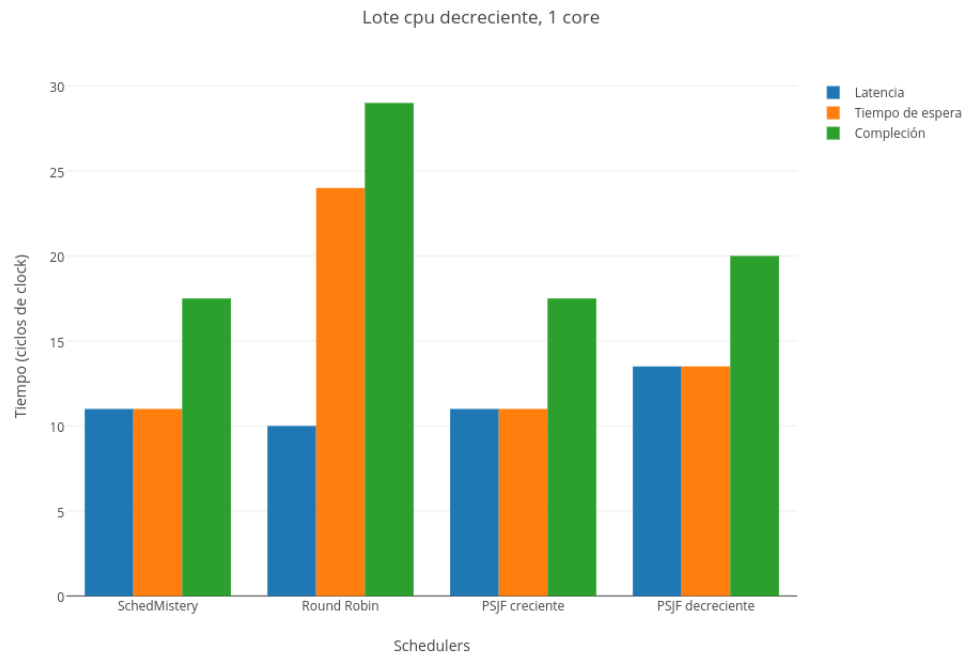


Figura 7.9: Comparación de métricas

#### 7.2.4. Lote con cpu decreciente sobre 2 cores

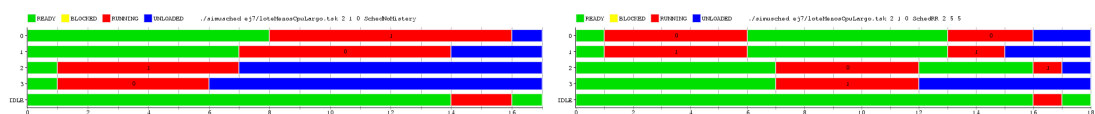


Figura 7.10: No Mystery y Round Robin

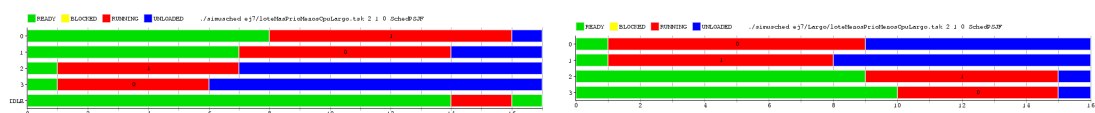


Figura 7.11: PSJF creciente y decreciente en prioridad

- Throughput NoMystery: 0.25
- Throughput Round Robin: 0.235294117647059

- Throughput PSJF creciente: 0.25
- Throughput PSJF decreciente: 0.2666666666666667

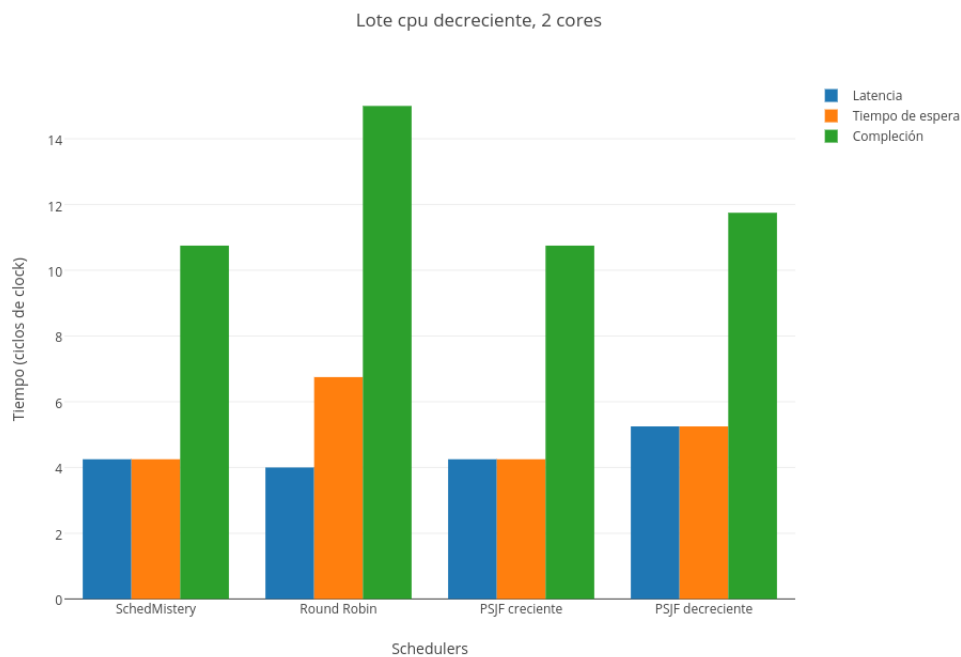


Figura 7.12: Comparación de métricas

### 7.2.5. Algunas observaciones

Debemos observar ciertas cuestiones antes de proseguir. Los lotes con tiempo de cpu decreciente causan que el scheduler *Round Robin* tenga un pobre desempeño, mientras que en los demás ejecutan las tareas de la misma manera (debido a que ellos ordenan las tareas basándose en un criterio antes de ponerlas en ejecución). También notamos que el PSJF con lotes de prioridad decreciente registra tiempos de latencia, espera y compleción más largos. Estos hechos justificarán algunas decisiones tomadas en la siguiente sección.

## 7.3. Pruebas con tareas con llamadas bloqueantes

Al ver los resultados del punto anterior, nos interesó analizar el desempeño de los *schedulers* al trabajar con tareas que realizan llamadas bloqueantes. Con ese objetivo, utilizamos *taskPajarillo* para armar nuevos lotes de prueba. También debimos adaptar *taskPajarillo* para poder medir sobre el PSJF.

Para esta sección decimos además usar lotes en los cuales el tiempo de cpu utilizado vaya creciendo conforme las tareas llegan al scheduler lo cual nos deja con dos tipos de test: el primero con tiempo de bloqueo creciente y el segundo con tiempo de bloqueo decreciente. El PSJF solo será evaluado con este tipo de tareas.

### 7.3.1. Lote con bloqueo creciente sobre 1 core

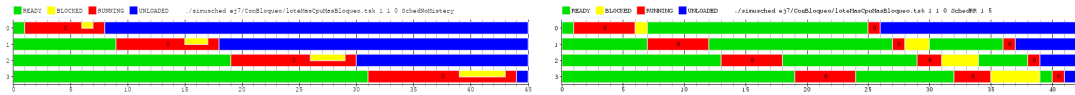


Figura 7.13: No Mistery y Round Robin

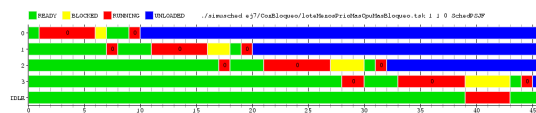


Figura 7.14: PSJF

- Throughput NoMistery: 0,090909090909091
- Throughput Round Robin: 0.097560975609756
- Throughput PSJF decreciente: 0.088888888888889

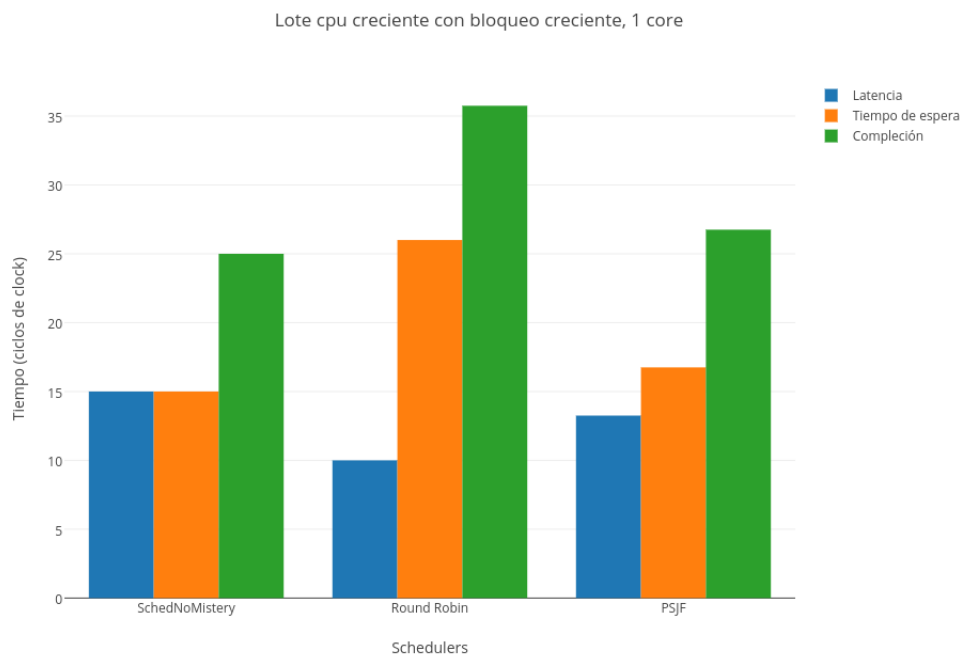


Figura 7.15: Comparación de métricas

### 7.3.2. Lote con bloqueo creciente sobre 2 cores

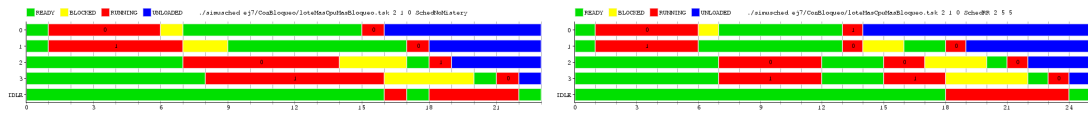


Figura 7.16: No Mistery y Round Robin

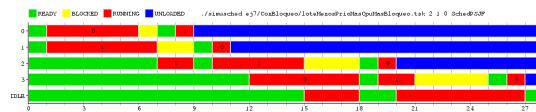


Figura 7.17: PSJF

- Throughput NoMistery: 0,166666666666667
- Throughput Round Robin: 0.166666666666667
- Throughput PSJF decreciente: 0.148148148148148

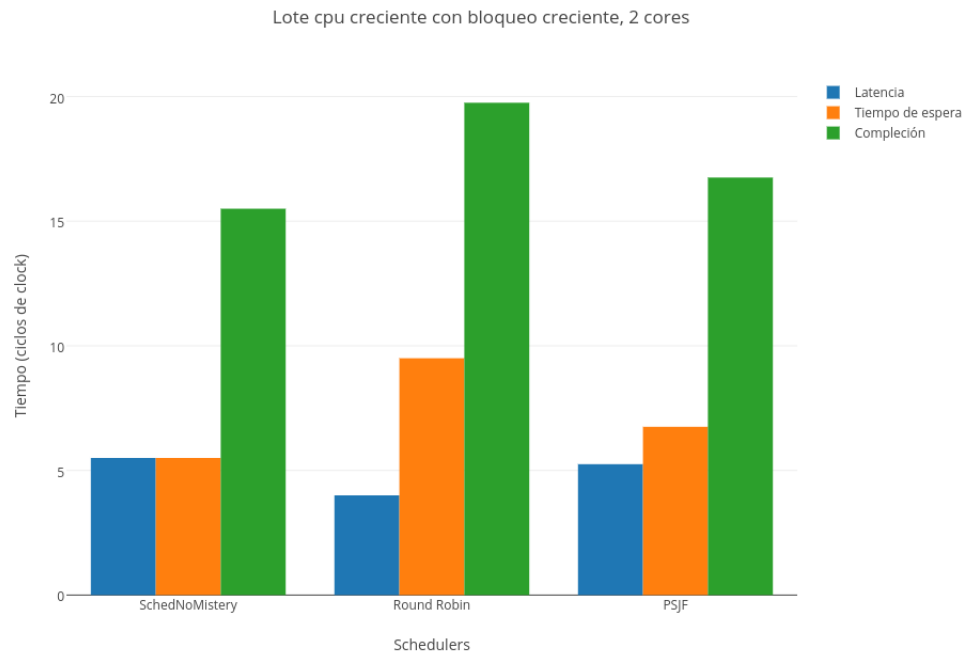


Figura 7.18: Comparación de métricas

### 7.3.3. Lote con bloqueo decreciente sobre 1 core

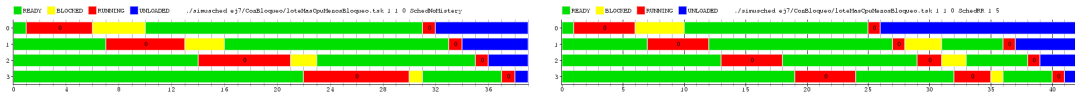


Figura 7.19: No Mistery y Round Robin

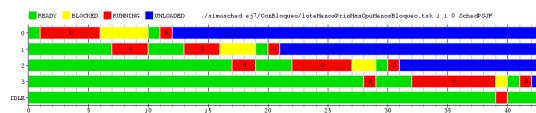


Figura 7.20: PSJF

- Throughput NoMistery: 0,090909090909091
- Throughput Round Robin: 0.097560975609756
- Throughput PSJF decreciente: 0.095238095238095

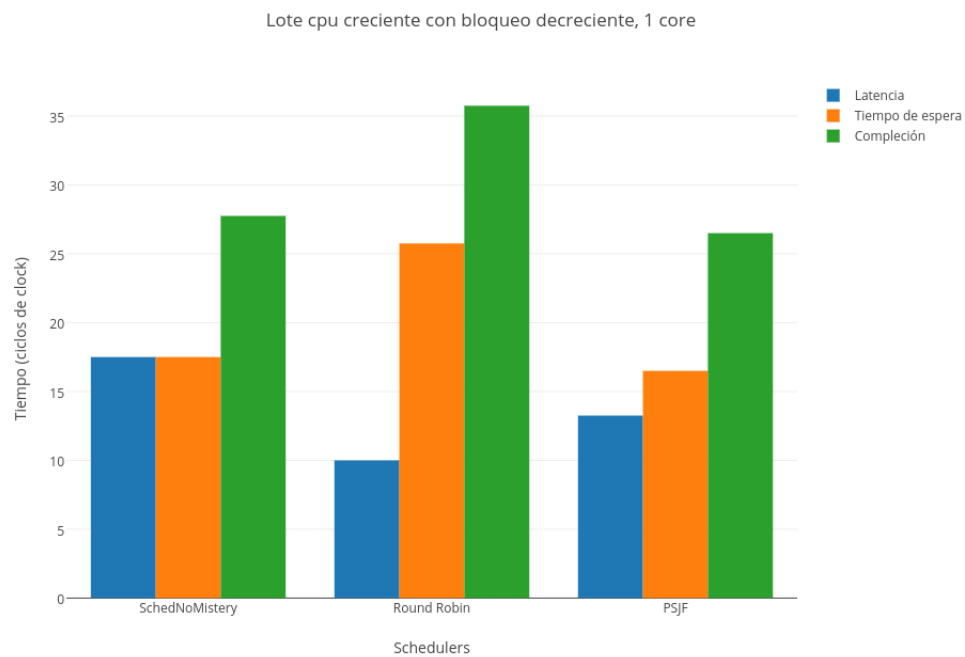


Figura 7.21: Comparación de métricas



### 7.3.4. Lote con bloqueo decreciente sobre 2 cores

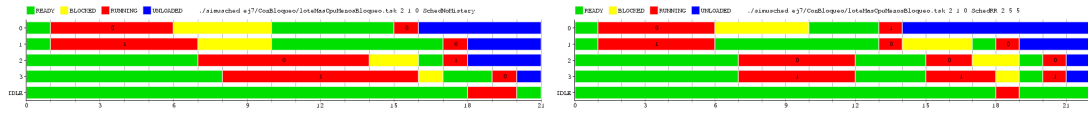


Figura 7.22: No Mistery y Round Robin

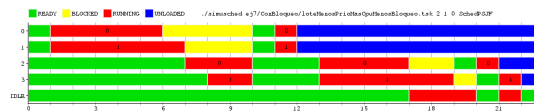


Figura 7.23: PSJF

- Throughput NoMistery: 0,181818181818182
- Throughput Round Robin: 0.19047619047619
- Throughput PSJF decreciente: 0.181818181818182

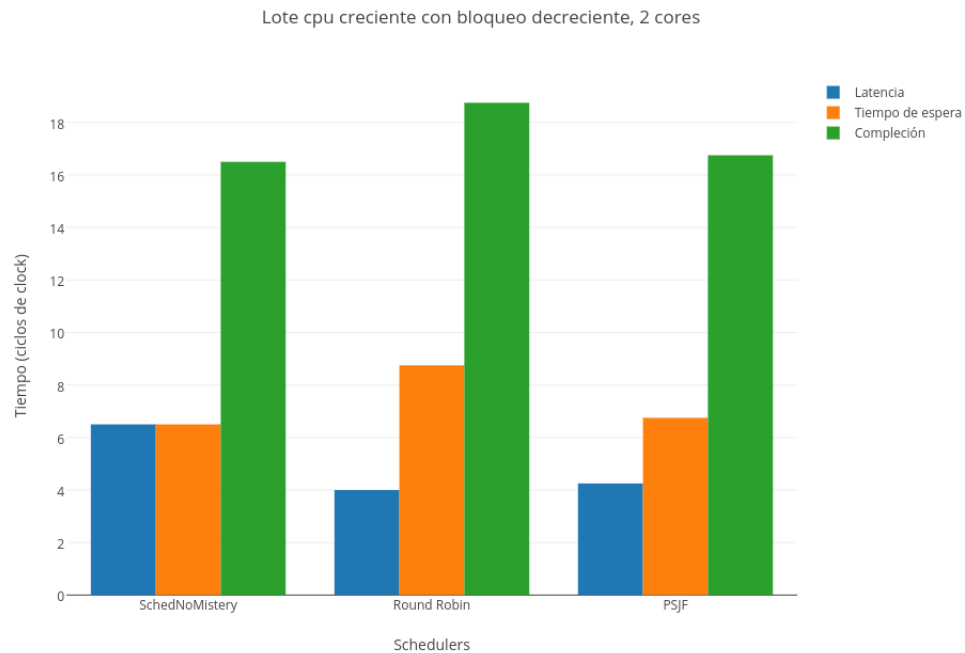


Figura 7.24: Comparación de métricas

## 7.4. Resultados

Observando solamente los resultados de los lotes con tareas sin llamada bloqueantes, se ve que el scheduler *SJF* tiene tiempos de espera y compleción más cortos, mientras que el *Round Robin* tiene siempre una mejor latencia que los demás, sobre todo en los lotes de tareas con llamadas bloqueantes.

En cuanto al waiting time del *Round Robin*, se puede ver que su desempeño es mucho más pobre que el resto, ya que paga el costo del cambio de contexto repetitivo de las tareas. Esto se refleja en el tiempo de compleción de las tareas, el cual es superior al resto.

## 7.5. Conclusiones

Respecto a las observaciones en la sección 7.2.5, concluimos que *Round Robin* se desempeña mejor si las tareas llegan ordenadas de menor a mayor respecto al tiempo de cpu, ya que fuerza una orden de ejecución de las tareas parecido al que se puede ver en los schedulers de tipo *SJF*.

Podemos concluir que un scheduler de tipo *Round Robin* se desempeña mejor en entornos interactivos, dónde la latencia resulta un factor crucial para minimizar el percibido tiempo de respuesta. En los casos en que se tengan tareas de uso intensivo de cpu y se requiera el menor tiempo de compleción, una estrategia *SJF* se adapta mejor, sobre todo si se sabe de antemano el tiempo de cpu que las tareas demandarán.