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

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 + 1)) + bmin);
        }
        return;
}
```

1.2. Gráfico de lote

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

```
*2 TaskConsola 1 1 15
TaskConsola 3 3 20
TaskConsola 3 5 5
```

Se grafican a continuación tres corridas sobre el *scheduler* FCFS con los mismos parámetros:

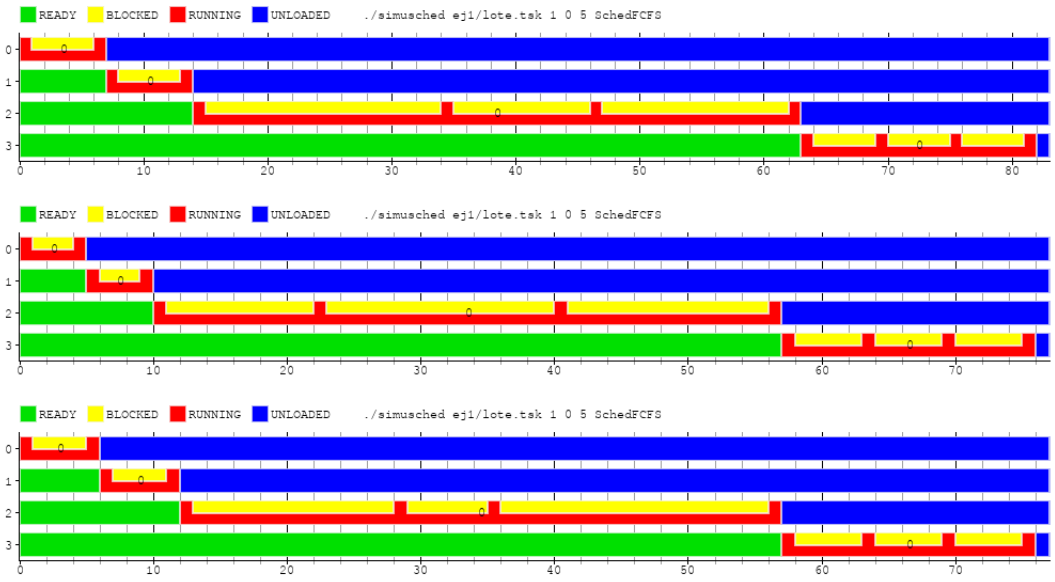

Figura 1.1: Tres corridas diferentes sobre el scheduler FCFS

Mirando las tareas 0 y 1 para las diferentes corridas tenemos:

- Corrida 1: un bloqueo de 5 unidades de tiempo.
- Corrida 2: un bloqueo de 3 unidades de tiempo.
- Corrida 3: un bloqueo de 4 unidades de tiempo.

Esto evidencia un comportamiento aleatorio de la tarea. Notamos también que las tareas 0 y 1 se comportan exactamente igual dentro de una misma corrida. Esto se debe a un comportamiento del simulador; el mismo en lugar de correr 2 veces la primera linea del lote, corre una sola *TaskConsola* y luego usa una copia de la misma.

En la tarea 2 observamos que pasa al incrementar el parámetro n de la tarea. Efectivamente, el parámetro coincide con la cantidad de bloqueos que realiza la tarea y se ve que cada uno de ellos tiene una duración aleatoria:

- Corrida 1: Los bloqueos duran 19, 11 y 15 unidades de tiempo.
- Corrida 2: Los bloqueos duran 11, 17 y 15 unidades de tiempo.
- Corrida 3: Los bloqueos duran 15, 6 y 20 unidades de tiempo.

El ultimo bloqueo de la corrida 3 evidencia que el algoritmo permite generar bloqueos de longitud *bmax* inclusive.

Con la ultima linea del lote (correspondiente a la tarea 3 en los gráficos de la figura 1.1) buscamos mostrar que el algoritmo de la tarea genera las duraciones de las llamadas bloqueantes respetando los limites *bmin* y *bmax*. Si $bmin = bmax$ el comportamiento esperado es que respete ambos limites y en el caso de nuestro lote la duración de los bloqueos siempre sea 5. Luego, al cumplirse esto en todas la corridas, *TaskConsola* respeta ambos limites.

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 titulo de *tiempo de espera*.

2.1. 1 núcleo

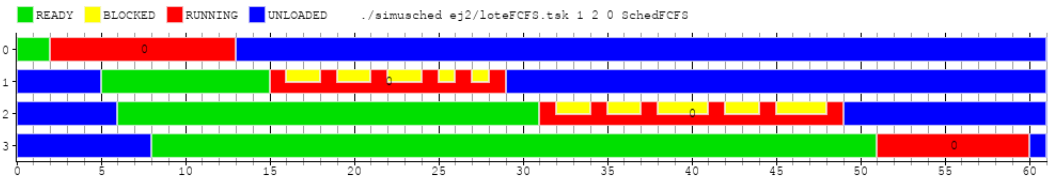

Figura 2.1: Scheduler FCFS con 1 core

Throughput: 4 tareas terminadas en 60 ciclos = 1 tarea cada 15 ciclos
Tiempo de espera promedio: (Tiempo de espera de t0 + Tiempo de espera de t1 + Tiempo de espera de t2 +Tiempo de espera de t3)$/4 = (2+10+25+43)/4 = 20$

2.2. 2 núcleos

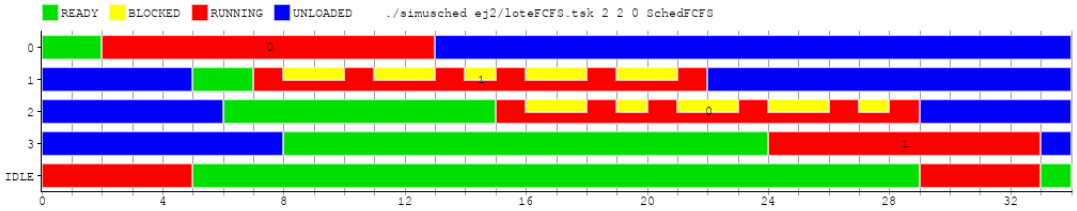

Figura 2.2: Scheduler FCFS con 2 core

throughput: 4 tareas terminadas en 33 ciclos = 1 tarea cada 8,25 ciclos
Tiempo de espera promedio: (Tiempo de espera de t0 + Tiempo de espera de t1 + Tiempo de espera de t2 +Tiempo de espera de t3)$/4 = (2+2+9+16)/4 = 7{,}25$

2.3. 3 núcleos

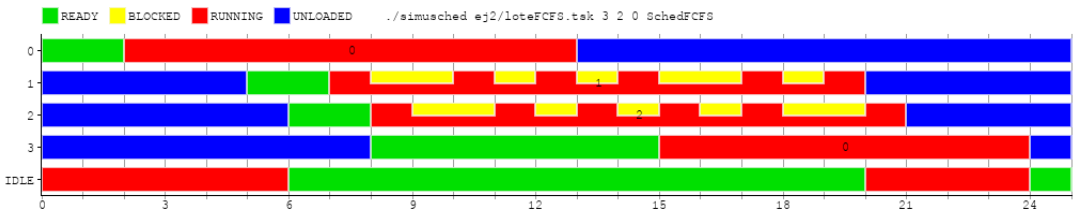

Figura 2.3: Scheduler FCFS con 3 core

throughput: 4 tareas terminadas en 24 ciclos = 1 tarea cada 6 ciclos
Tiempo de espera promedio: (Tiempo de espera de t0 + Tiempo de espera de t1 + Tiempo de espera de t2 +Tiempo de espera de t3)$/4 = (2+2+2+7)/4 = 3{,}25$

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

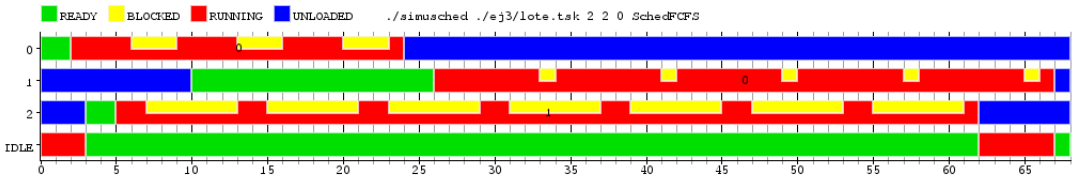

Figura 3.1: Schedduler FCFS con 2 core

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

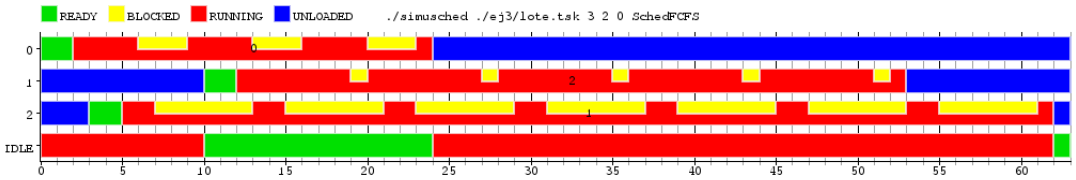

Figura 3.2: Schedduler FCFS con 2 core

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 que consideramos más *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 permitiríamos que la tarea termine con menos interrupciones, lo cual es útil en casos de tareas con muchos bloqueos.

Esto decisión permite una implementación más sencilla del scheduler donde se mantiene un quantum por cada core y no por cada tarea, lo que requeriría de estructuras de datos más complicadas.

Para mostrar el comportamiento hemos elegido separar las carácteristicas del scheduler más relevantes y mostrarlas en diferentes gráficos las cuales son:

1. Comportamiento de los Quantums
2. Comportamiento frente a los bloqueos
3. Quantums diferentes por core

El quantum utilizado será de 2 tiempos para todos los cores salvo para el test 3 donde se exhibirá un lote de tareas donde se mostrará que el scheduler también puede y funciona correctamente con quantums diferentes por core.

Para cada uno de los puntos hemos decidido utilizar lotes de tasks pajarillo.

Para la fácil comprensión del lector, se decidió utilizar un solo core en los tests 1 y 2.

No se incluirá tiempo por cambio de contexto ni cambio de core en ninguno de los tests, ya que complican demasiado la lectura y referencia de los gráficos y no aportan detalles en cuanto al funcionamiento real del scheduler.

1. Comportamiento de los Quantums

Con el siguiente lote podremos ver como se respeta el quantum establecido para los cores, desalojando una tarea y alojando otra cuando sea el caso y como al recobrar la ejecución una tarea, su quantum se reinicia:

```
TaskPajarillo 1 3 0
TaskPajarillo 1 1 0
TaskPajarillo 1 4 0
TaskPajarillo 1 2 0
```

El lote está conformado por una serie de tres tareas que repiten una vez uso de cpu por tarea, de uno a cuatro tiempos y sin bloqueo en todos los casos, lo que permitirá comprender mejor el manejo de los quantums.

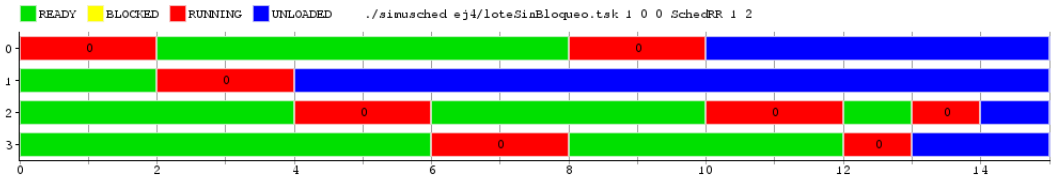

Figura 4.1: ./simusched loteSinBloqueo.tsk 1 0 0 SchedRR 1 2

Puede observarse que en el caso de la tarea *cero*, que hace uso de tres tiempos de cpu, la misma es desalojada cuando cumple los dos tiempos de quantum establecidos para dar paso a la ejecución de la tarea *uno*, que empieza y termina dentro del mismo quantum dado que solo utiliza un tiempo de cpu. No debemos olvidar que hay un tiempo de cpu para quitar de ejecución una tarea.

Para el caso de la tarea *dos*, podemos observar como el quantum es reiniciado dado que la tarea utiliza cuatro tiempos de cpu. Por lo tanto, en el tiempo 10, cuando la tarea vuelve a ejecución, su quantum se reinicia. El caso de la tarea *dos* es el menos interesante. La tarea hace exactamente dos usos de cpu, igual que la cantidad del quantum establecido para el core. Por lo tanto la misma termina en el tiempo 12 que es luego de que terminan las tareas *cero* y *dos* que son las que más duración tienen.

Esto es algo que también puede observarse muy fácilmente en el gráfico y es el correcto orden de finalización de las tareas, el cual es: 1, 3, 0, 2 (por orden de id) respetando el uso de cpu que cada una realiza.

2. Comportamiento frente a los bloqueos

Para mostrar el funcionamiento de los bloqueos se utilizaran los siguientes lote:

Lote 1:

```
TaskPajarillo 1 4 0
TaskPajarillo 1 0 1
```

Lote 2:

```
TaskPajarillo 1 4 0
TaskPajarillo 1 0 3
```

Como puede verse en ambos lotes, solo son dos tareas de las cuales la segunda es quien realiza un bloqueo de uno y tres tiempos para *Lote* 1 y 2 respectivamente sin hacer uso de cpu, mientras que la primera tarea funciona de manera opuesta. De esta manera podremos mostrar que sucede al realizarse un bloqueo durante la ejecución.

Lote 1:

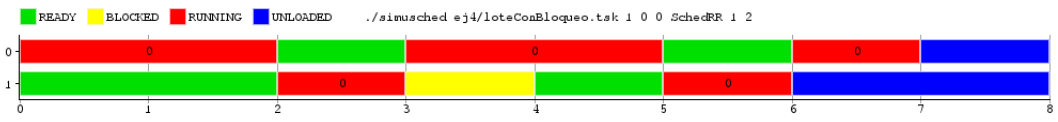

Figura 4.2: ./simusched loteConBloqueo1.tsk 1 0 0 SchedRR 1 2

Lote 2:

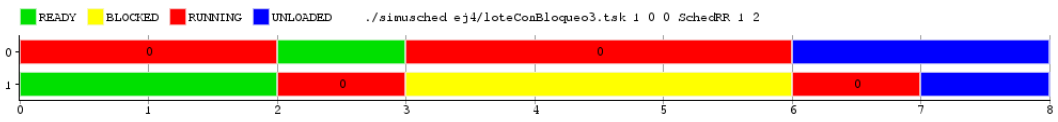

Figura 4.3: ./simusched loteConBloqueo3.tsk 1 0 0 SchedRR 1 2

Puede observarse que en ambos lotes, cuando una tarea entra en bloqueo, otra ocupa su lugar en el core. En este caso la tarea *cero* ocupa el lugar de la tearea *uno* en el core *cero*. Para el *Lote* 1 puede observarse que cuando la tarea *uno* sale del bloqueo queda en estado *ready* hasta que la tarea *cero* consume su quantum. En el caso del *Lote* 2, como el bloqueo dura lo suficiente, la tarea *cero* finaliza su ejecución y además es sacada de ejecución. Cuando el bloqueo de la tarea *uno* finaliza, la tarea *cero* ya ha finalizado.

Lo último que puede notarse en ambos lotes, es que aunque la tarea *uno* no realiza uso de cpu propio, consume un tiempo de cpu antes de entrar en bloqueo, como fué definido por enunciado.

3. Quantums diferentes por core

Para este test utilizaremos 2 cores con quantums de uno para el core *cero* y dos para el core *uno*.

El lote utilizado consta de tres tareas de las cuales las dos primeras son las tareas designadas para mostrar como trabajan los quantums de cada core y la tarcera se utilizará a modo de tarea desalojadora, simplemente para que quede en evidencia el uso de los quantums.

```
*2 TaskPajarillo 1 4 0
TaskPajarillo 1 3 0
```

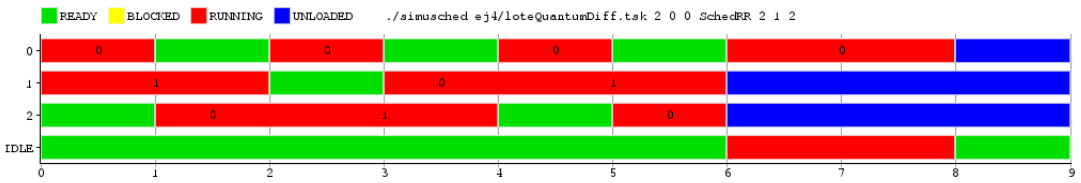

Figura 4.4: ./simusched loteQuantumDiff.tsk 2 0 0 SchedRR 2 1 2

Puede observarse que la tarea *cero*, que siempre corre en el core *cero*, lo hace con quantum *uno*. En cambio la tarea *uno* cada vez que corre en el core *uno* (tiempos 0 a 2 y 4 a 6) lo hace con quantum *dos*. La tarea *dos* también evidencia lo mismo para el core *uno* (tiempos 2 a 4). Por lo que así se muestra que el scheduler es capas de funcionar con cores de diferente quantum si así fuera deseado.

5. Ejercicio 5

Para experimentar con el **SchedMistery** 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.

Para elegir los lotes a probar, se toma en cuenta que el scheduler solo toma tareas del tipo taskCpu. Por lo tanto el factor mas influyente en el comportamiento del mismo es el tiempo de cpu que tomarán las tareas en ejecutar. Luego se decide generar el lote distintasMayor y distintasMenor. Lo que queremos observar al ejecutarlas es si el scheduler tiene alguna prioridad basada en el tiempo de cpu o si solamente usa algún ordenamiento basado en los tiempos de llegada de las tareas. Con el lote largasLuegoCortas se desea ver, si se realizan desalojos tanto sea por existir un *quantum* de ejecución como por priorización. Luego se usa una tarea desproporcionadamente larga en comparación a las demás.

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

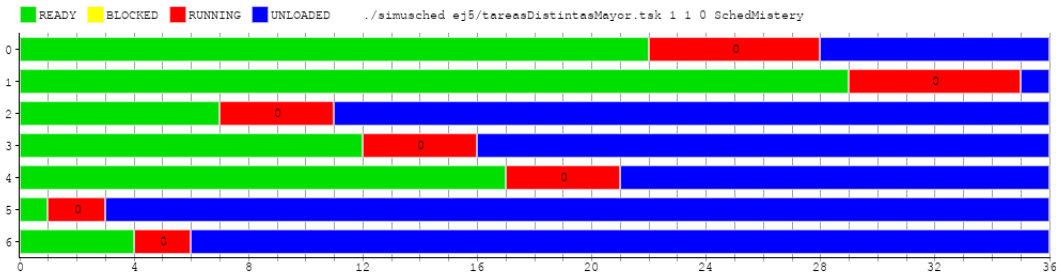

Figura 5.2: DistintasMayor

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 elije, de las tareas en estado *ready*, aquella con menor tiempo de cpu.

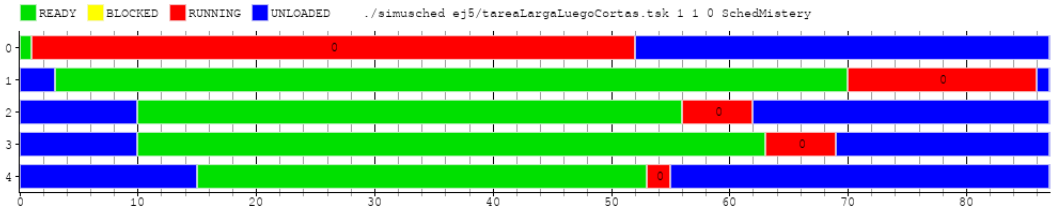

Figura 5.3: LargaLuegoCortas

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 **SchedMistery** 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 tareas en función a los siguientes objetivos:

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 scheduling es preemptive

```
TaskPriorizada 1 4          TaskPriorizada 1 3
@2:                         @2:
TaskPriorizada 1 3          TaskPriorizada 1 4
@3:                         @3:
TaskPriorizada 1 2          TaskPriorizada 1 1
@4:                         @4:
fTaskPriorizada 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

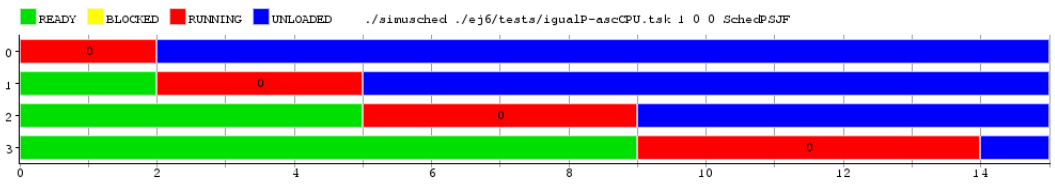

Figura 6.1: Archivo tareas ordenadas de menor a mayor uso de CPU

El ordenamiento en este caso resulto en la misma en el que esta ordenado el archivo de entrada

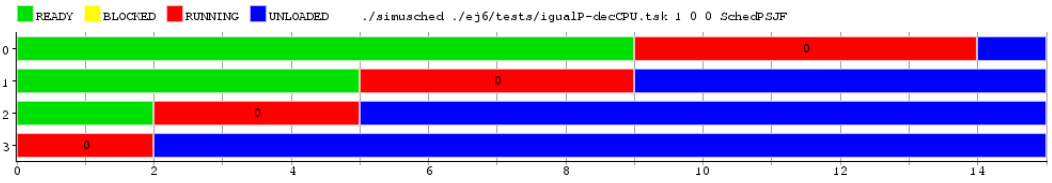

Figura 6.2: Archivo tareas ordenadas de mayor a menor uso de CPU

A diferencia del anterior, en este caso se invirtió el orden para que la ejecución sea equivalente a la anterior, dejando las tareas ordenadas por uso de cpu de forma creciente.

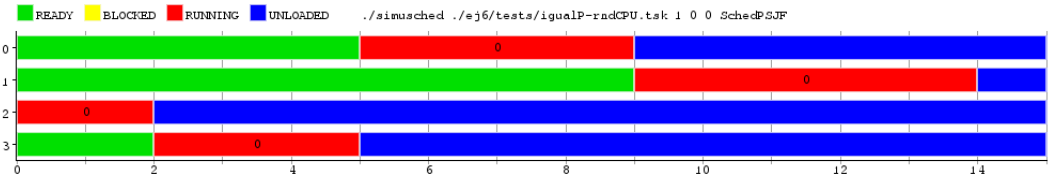

Figura 6.3: Archivo tareas desordenado respecto al uso de CPU

Finalmente el último lote desordenado nos produce una ejecución equivalente a las 2 anteriores: las tareas fueron ordenadas por 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: al probar organizar el set de tareas de distintas formas (menor a mayor, mayor a menor y aleatorio), se puede ver que el uso de cpu es una variable para el ordenamiento. A priori, podemos decir que respeta el comportamiento esperado en una estrategia **SJF**.

6.2.2. 2 - Scheduler preemptive

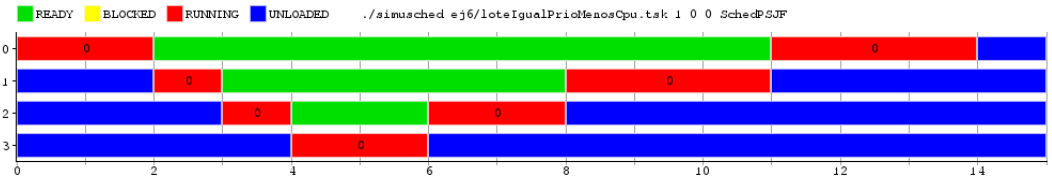

Figura 6.4: Archivo tareas ordenadas de mayor a menor uso de CPU pero con inicios desfasados

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:

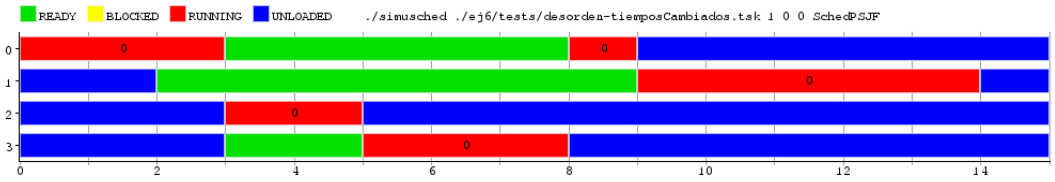

Figura 6.5: Archivo tareas con inicios desfasados sin orden especifico de uso de CPU

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

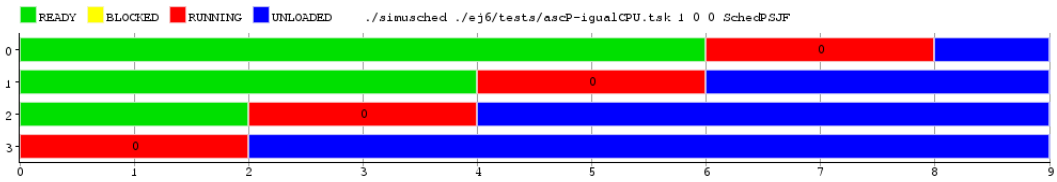

Figura 6.6: Archivo tareas ordenadas de menor a mayor prioridad

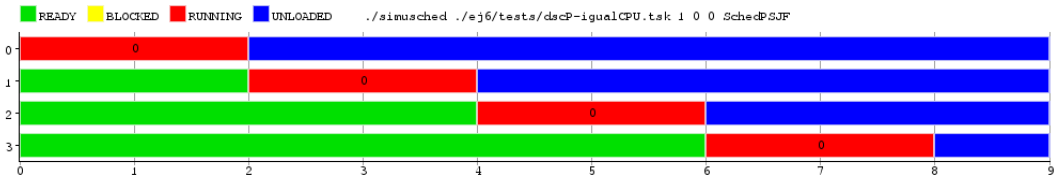

Figura 6.7: Archivo tareas ordenadas de mayor a menor prioridad

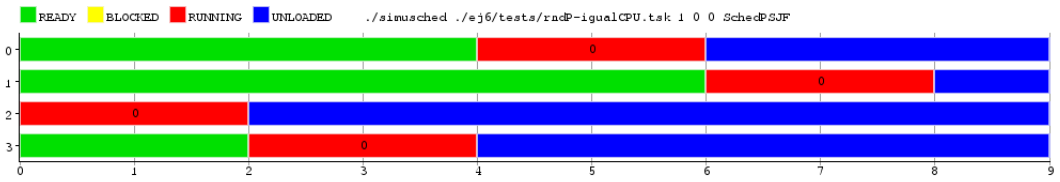

Figura 6.8: Archivo tareas desordenado respecto a la prioridad

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

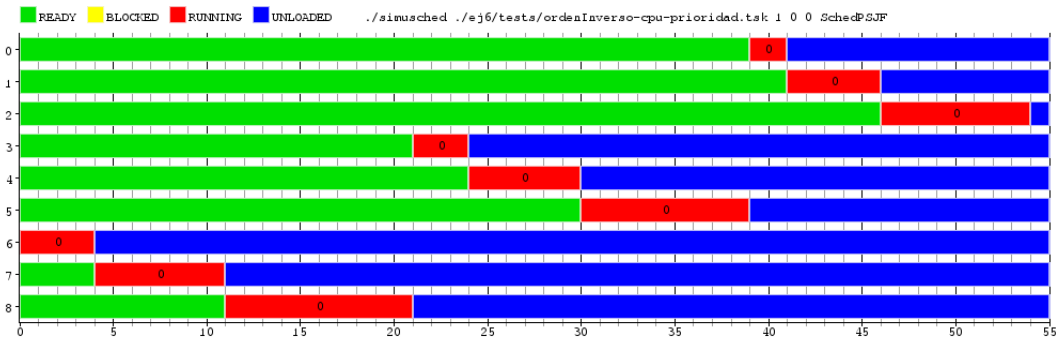

Figura 6.9: Doble ordenamiento

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 al revés) 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* (excepto que PSJF desalojaría al entrar una tarea con menor tiempo de cpu que la que estuviese corriendo en un dado momento).

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

7.2.1. Lote con cpu creciente sobre 1 core

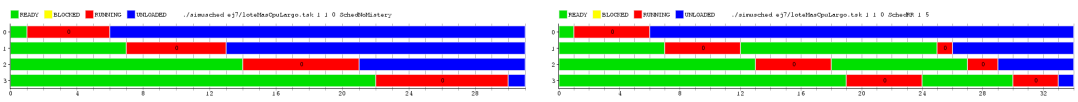

Figura 7.1: No Mistery y Round Robin

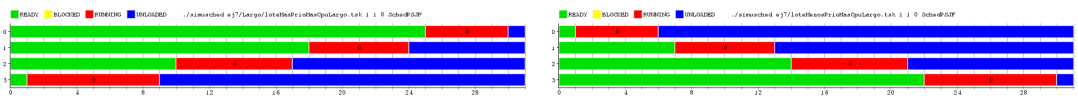

Figura 7.2: PSJF creciente y decreciente en prioridad

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

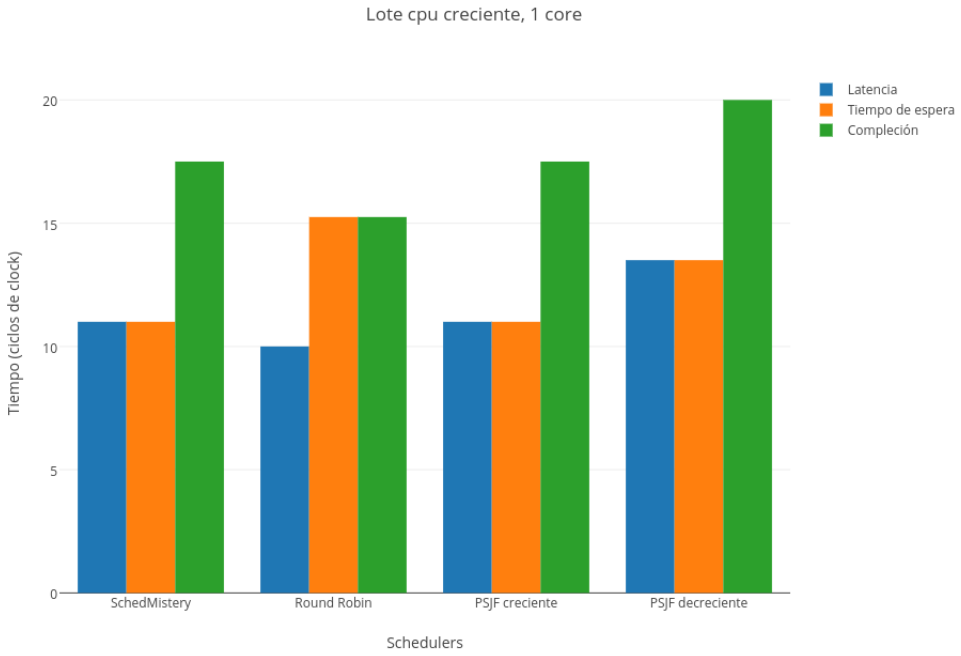

Figura 7.3: Comparación de métricas

7.2.2. Lote con cpu creciente sobre 2 cores

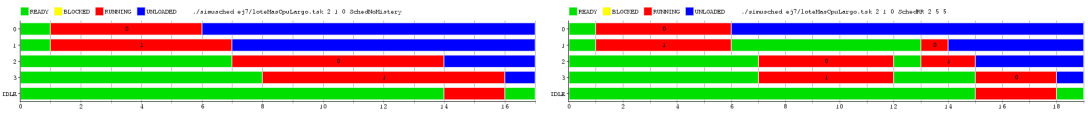

Figura 7.4: No Mistery y Round Robin

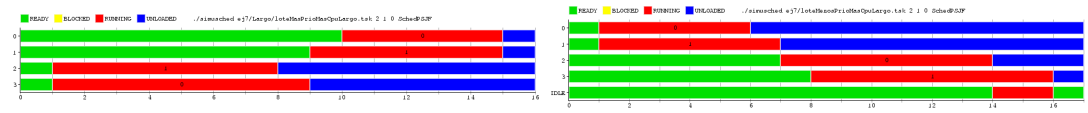

Figura 7.5: PSJF creciente y decreciente en prioridad

- Throughput NoMistery: 0.25

- Throughput Round Robin: 0.222222222222222
- Throughput PSJF creciente: 0.266666666666667
- Throughput PSJF decreciente: 0.25

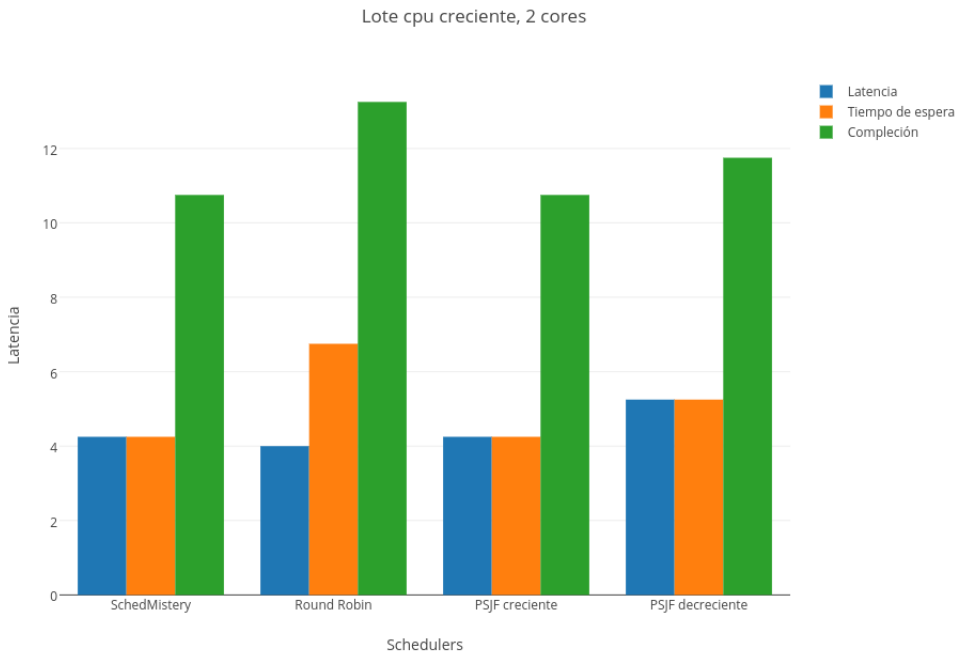

Figura 7.6: Comparación de métricas

7.2.3. Lote con cpu decreciente sobre 1 core

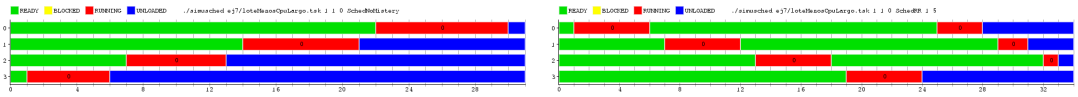

Figura 7.7: No Mistery y Round Robin

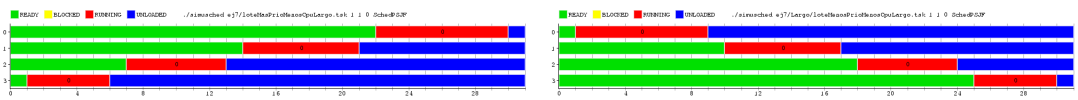

Figura 7.8: PSJF creciente y decreciente en prioridad

- Throughput NoMistery: 0.133333333333333
- Throughput Round Robin: 0.121212121212121

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

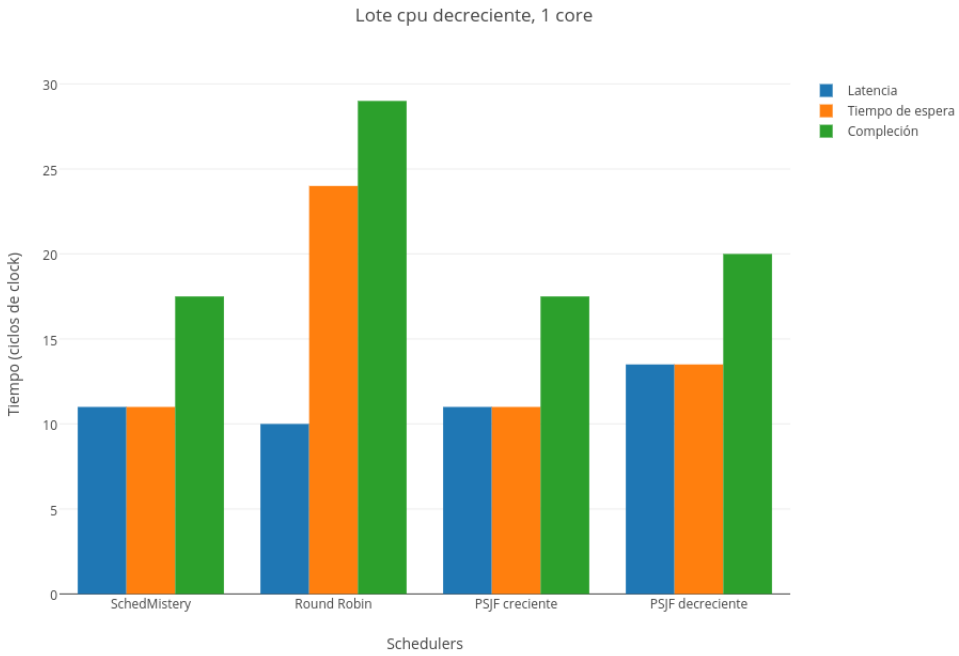

Figura 7.9: Comparación de métricas

7.2.4. Lote con cpu decreciente sobre 2 cores

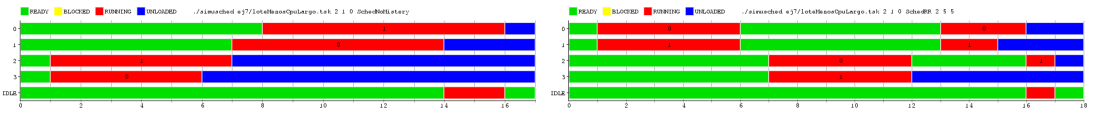

Figura 7.10: No Mistery y Round Robin

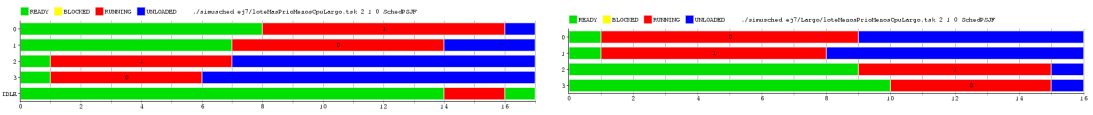

Figura 7.11: PSJF creciente y decreciente en prioridad

- Throughput NoMistery: 0.25
- Throughput Round Robin: 0.235294117647059

- Throughput PSJF creciente: 0.25
- Throughput PSJF decreciente: 0.266666666666667

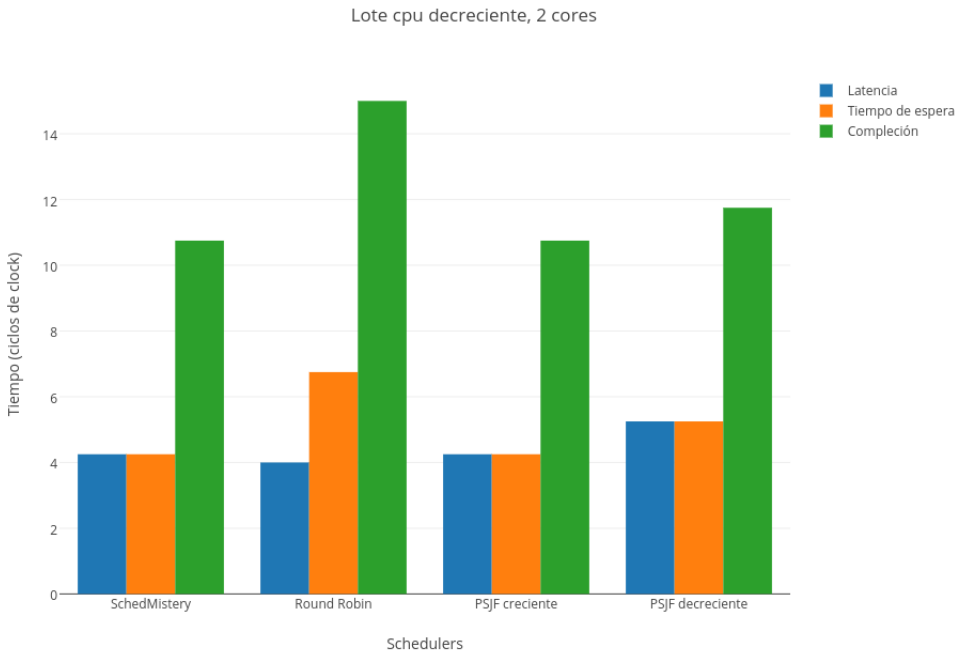

Figura 7.12: Comparación de métricas

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 más *waiting time* y menor *throughput*, 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

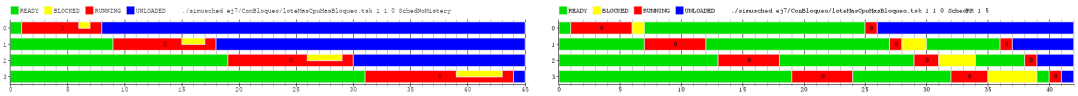

Figura 7.13: No Mistery y Round Robin

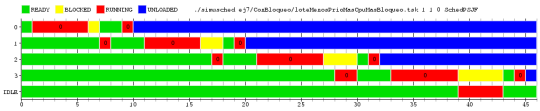

Figura 7.14: PSJF

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

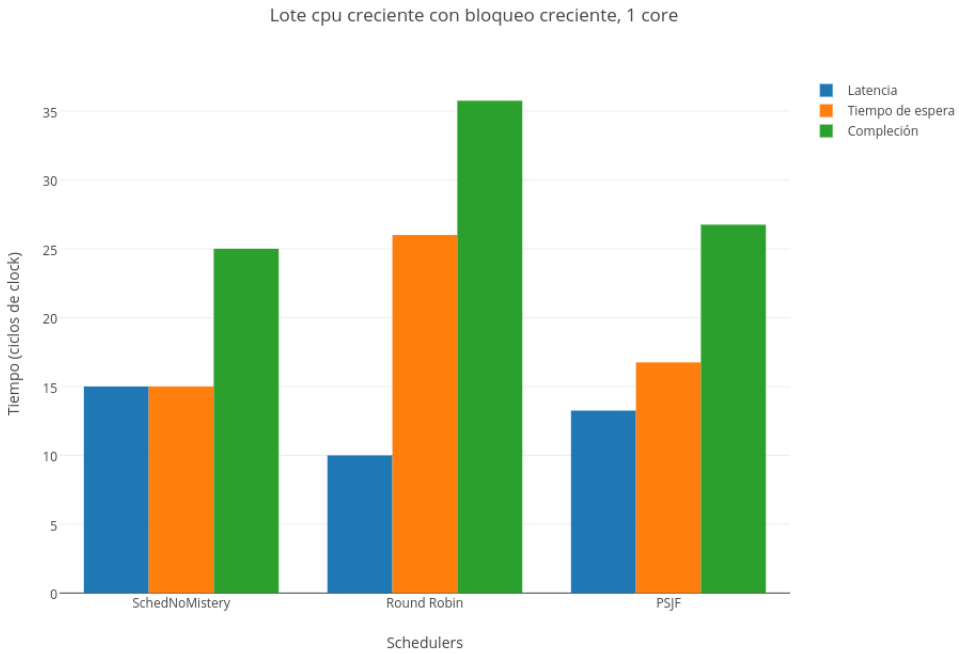

Figura 7.15: Comparación de métricas

7.3.2. Lote con bloqueo creciente sobre 2 cores

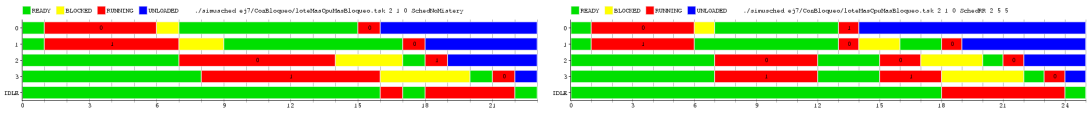

Figura 7.16: No Mistery y Round Robin

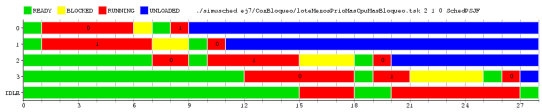

Figura 7.17: PSJF

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

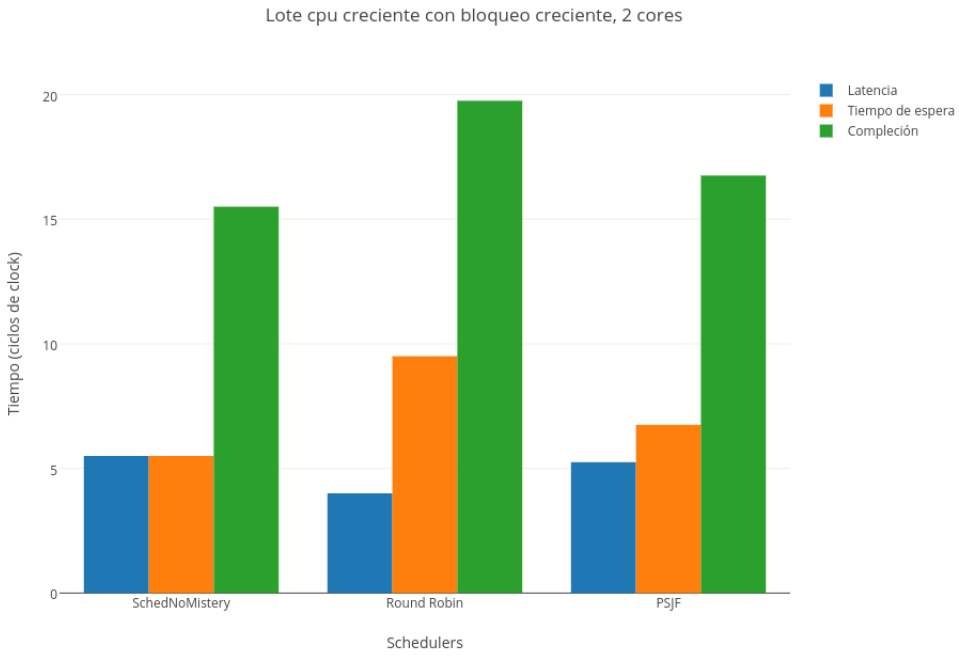

Figura 7.18: Comparación de métricas

7.3.3. Lote con bloqueo decreciente sobre 1 core

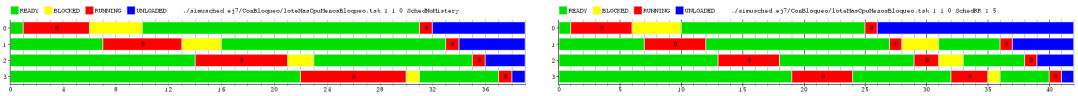

Figura 7.19: No Mistery y Round Robin

Figura 7.20: PSJF

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

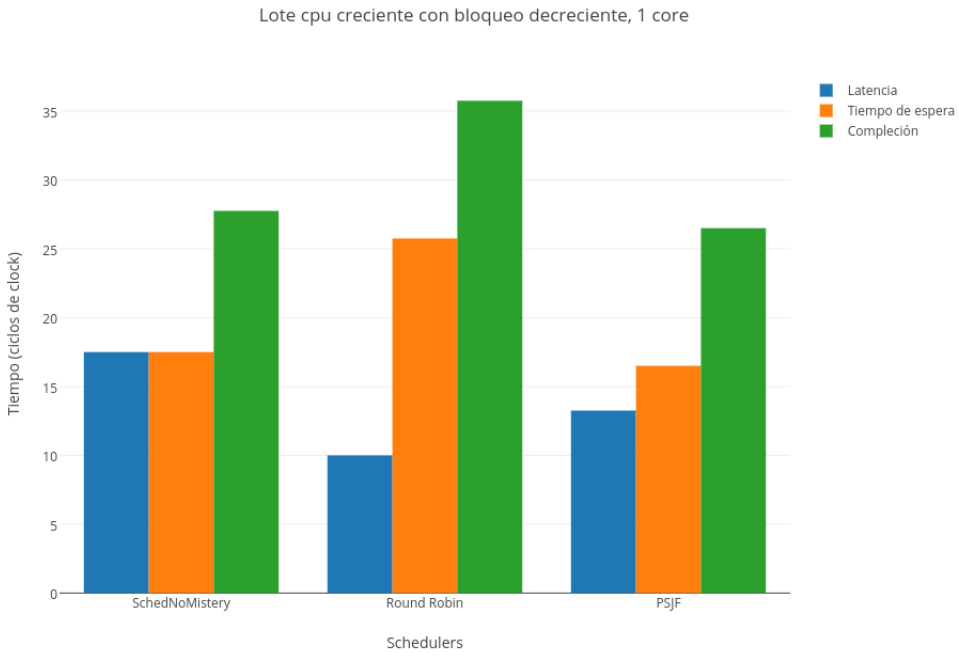

Figura 7.21: Comparación de métricas

7.3.4. Lote con bloqueo decreciente sobre 2 cores

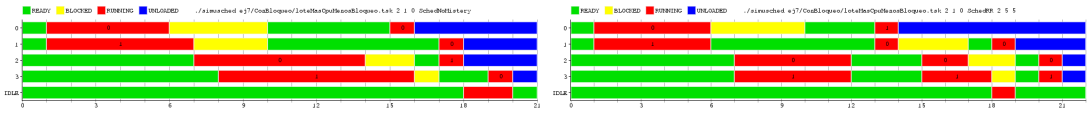

Figura 7.22: No Mistery y Round Robin

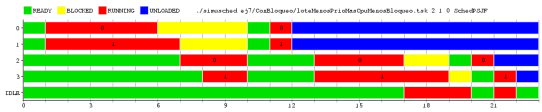

Figura 7.23: PSJF

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

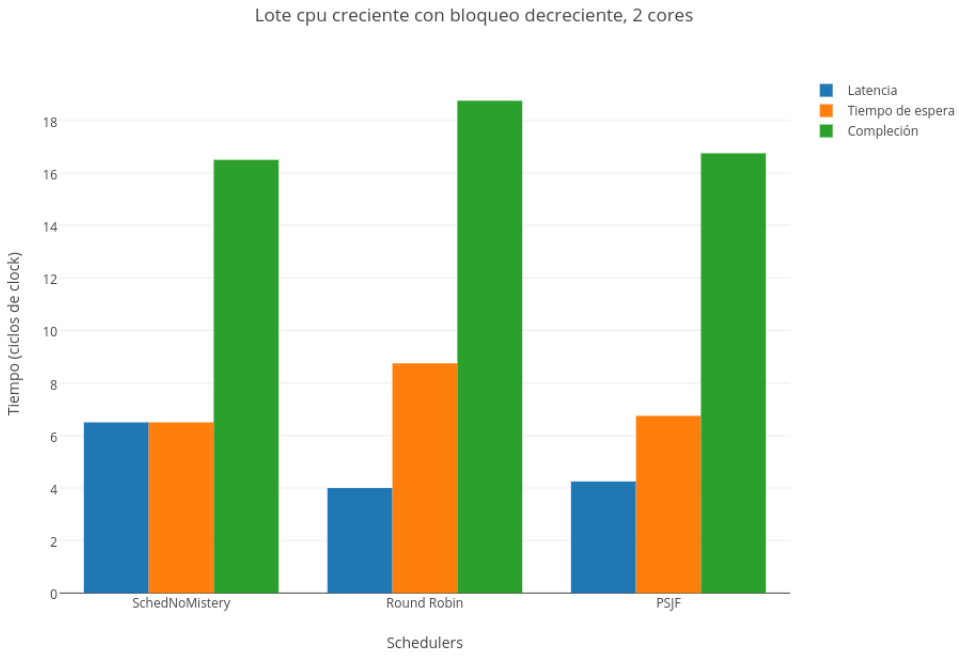

Figura 7.24: Comparación de métricas

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.