DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 1

Scheduling

Primer Cuatrimestre de 2017 — Sistemas Operativos

Integrante	LU	Correo electrónico
Langberg, Andrés	249/14	andreslangberg@gmail.com
Sticco, Patricio	337/14	patosticco@gmail.com
Cadaval, Matías	345/14	matias.cadaval@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) 4576-3359

http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

Se pide programar un tipo de tarea `TaskConsola`, que simulará una tarea interactiva. La tarea debe realizar n llamadas bloqueantes, cada una de una duración al azar entre los parámetros *bmin* y *bmax* (inclusive).

1.2. Desarrollo

Para calcular el tiempo de bloqueo de cada una de las llamadas bloqueantes, se generó un valor aleatorio utilizando la función *rand()*. Para que los valores se encuentren entre bmin y bmax, se tomó modulo (bmax-bmin+1) y luego se sumó bmin.

En la figura a continuación, se muestran los resultados obtenidos al ejecutar el siguiente lote de tareas:

```
TaskConsola 8 2 5
TaskConsola 3 4 6
@5:
TaskConsola 1 1 10
```

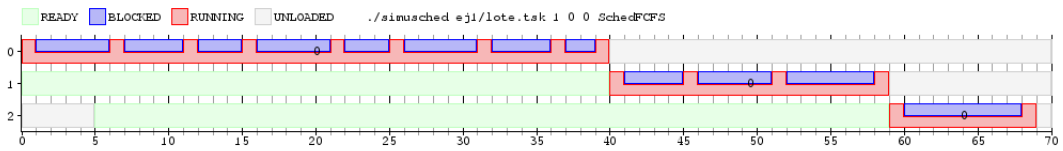

Figura 1: Ejecución del lote de tareas

2. Ejercicio 2

2.1. Introducción

Se pide calcular la *latencia*, el *waiting time* de cada tarea y el *throughput* en distintas simulaciones a partir de un mismo grupo de tareas, utilizando el algoritmo `FCFS` para escenarios con 1, 2 y 3 núcleos con un *context switch* de dos ciclos.

2.2. Desarrollo

A continuación, se muestran los resultados obtenidos al ejecutar el siguiente lote de tareas:

```
TaskCPU 10
@5:
TaskConsola 5 1 3
@6:
TaskConsola 5 1 4
@8:
TaskCPU 8
```

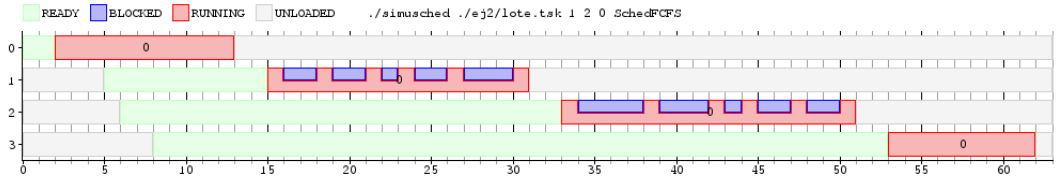

Figura 2: Simulación para 1 núcleo

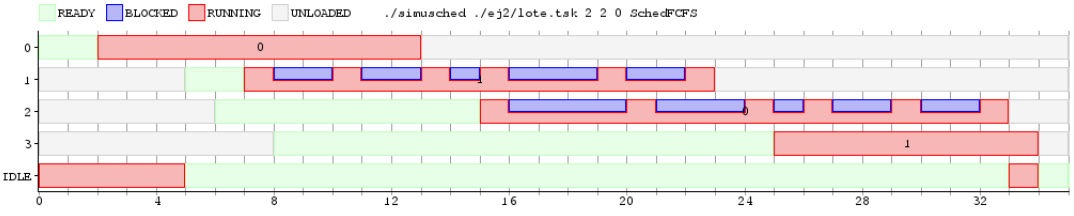

Figura 3: Simulación para 2 núcleos

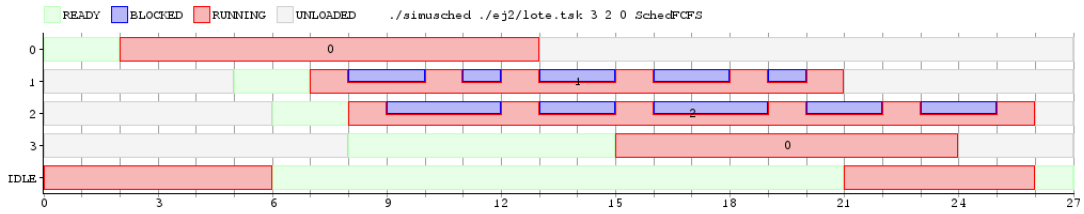

Figura 4: Simulación para 3 núcleos

Con respecto a la *latencia*, el *waiting time* de cada tarea y el *throughput*, se presentan a continuación los resultados:

Resultados: Latencia			
Tarea	1 núcleo	2 núcleos	3 núcleos
0	2	2	2
1	10	2	2
2	27	9	2
3	45	17	7
Promedio	21	7.5	3.25

Cuadro 1: Latencia expresada en unidades de tiempo.

Resultados: Waiting time			
Tarea	1 núcleo	2 núcleos	3 núcleos
0	2	2	2
1	10	2	2
2	27	9	2
3	45	17	7
Promedio	21	7.5	3.25

Cuadro 2: Waiting time expresado en unidades de tiempo.

Resultados: Throughput	
1 núcleo	0.065
2 núcleos	0.118
3 núcleos	0.154

Cuadro 3: Throughput

Dado que se realizó la simulación con un algoritmo de scheduling `FCFS`, la latencia es igual al *waiting time* (ya que las tareas se ejecutan hasta finalizar sin ser interrumpidas).

3. Ejercicio 3

3.1. Introducción

Se pide programar un tipo de tarea `TaskPajarillo`.

La tarea debe realizar *cant_repeticiones* ciclos, donde en cada iteración se deberá realizar una llamada a `uso_CPU`(*tiempo_cpu*) y luego una llamada bloqueante de duración *tiempo_bloqueo*.

A partir de un set de 3 tareas de este tipo, se realiza una simulación usando el algoritmo `FCFS` para 2 y 3 núcleos con un *context switch* de dos ciclos, aclarando la *latencia*, el *waiting time* y el *throughput* de ejecución.

3.2. Desarrollo

Para realizar las pruebas detalladas anteriormente, utilizaremos el siguiente lote de tareas:

```
TaskPajarillo 2 2 3
@6:
TaskPajarillo 1 4 5
@7:
TaskPajarillo 2 3 5
```

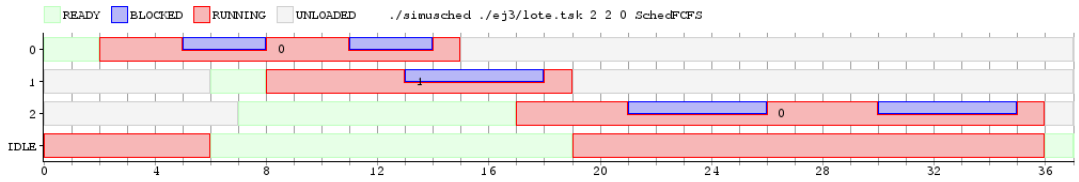

Figura 5: Simulación para 2 núcleos

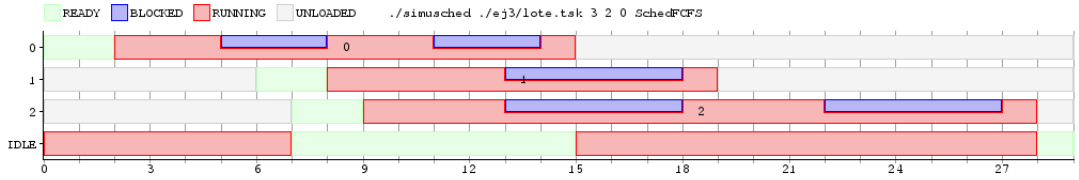

Figura 6: Simulación para 3 núcleos

Con respecto a la *latencia*, el *waiting time* de cada tarea y el *throughput*, se presentan a continuación los resultados:

Resultados: Latencia		
Tarea	2 núcleos	3 núcleos
0	2	2
1	2	2
2	10	2
Promedio	4.67	2

Cuadro 4: Latencia expresada en unidades de tiempo.

Resultados: Throughput	
2 núcleos	0.083
3 núcleos	0.107

Al igual que en el ejercicio anterior, dado a que se realizó la simulación con un algoritmo de scheduling `FCFS`, la latencia es igual al *waiting time* (dado a que las tareas se ejecutan hasta finalizar).

4. Ejercicio 4

4.1. Introducción

En este ejercicio, extenderemos el simulador con la inclusión de un nuevo algoritmo de scheduling, `Round-Robin`. La implementación recibe como primer parámetro la cantidad de núcleos y a continuación los valores de sus respectivos *quantums*.

4.2. Desarrollo

El nuevo algoritmo de scheduling, se implementó mediante una clase de C++ (hereda de `SchedBase`). Para ser un scheduler válido, el mismo implementa tres métodos principales:

- `LOAD`(*pid*): Notifica al scheduler la llegada de un nuevo *pid*.
- `UNBLOCK`(*pid*): Notifica al scheduler que el proceso *pid* deja de estar bloqueado.
- `TICK`(*cpu*, *motivo*): El parámetro *cpu* indica qué núcleo es el que realiza el tick, mientras que *motivo* indica qué ocurrió con la tarea que estuvo en posesión del CPU durante el último ciclo de reloj (puede ser un *tick*, *block* o *exit*).

Ya entrando en detalle sobre nuestra implementación del `Round-Robin`, para almacenar la información de cada CPU utilizamos 3 vectores (donde cada posición representa un núcleo en particular). Los mismos guardan el *quantum* inicial, los ticks restantes para completar el *quantum* y el *pid* de la tarea ejecutándose en dicho momento.

En cuanto a los procesos, utilizamos una cola para guardar las tareas en estado *ready*. Sobre esta idea, los métodos `LOAD` y `UNBLOCK` se reducen a encolar el proceso indicado por parámetro.

La implementación del método `TICK` se divide en dos casos dependiendo si el procesador se encuentra en `IDLE` o si se está ejecutando un proceso.

- Caso `IDLE`: Si no hay ninguna tarea en estado *ready*, se continúa en `IDLE`, mientras que si encuentra alguna, la desencola y la pasa a ejecutar.
- Caso proceso: En este caso, si el *motivo* es *tick*, se le resta una unidad a los ticks restantes para completar el *quantum* del procesador que llama a la función y se verifica si llegaron a cero. Si es así, se enconla nuevamente el proceso en estado *ready* y se carga uno nuevo (si hay disponible). Si el motivo es *block*/*exit*, simplemente se restablece el *quantum* del procesador y se busca una nueva tarea a ejecutar (la siguiente de la cola o la `IDLE` en caso de que la misma esté vacía).

Ejemplo de ejecución:

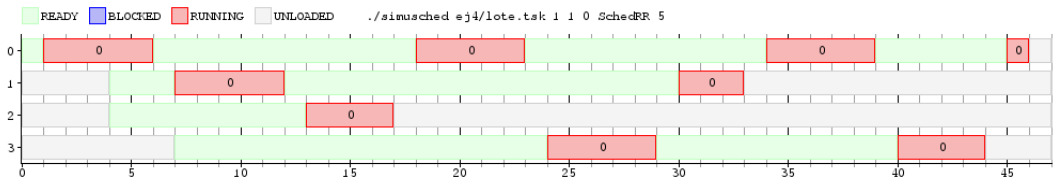

Figura 7: Simulación SchedRR

5. Ejercicio 5

5.1. Introducción

Se pide experimentar con el scheduler `SchedMistery` provisto por la cátedra para deducir su funcionamiento, y luego replicarlo en `SchedNoMistery`. Realizaremos tres experimentos, y en cada uno de ellos explicaremos qué características del scheduler nos permitió identificar. Luego, replicaremos su funcionamiento en `SchedNoMistery` implementado en el archivo `sched_no_mistery.cpp`. Hay que tener en cuenta que `SchedMistery` sólo recibe tareas de tipo `TaskCPU`.

5.2. Desarrollo

En primer lugar haremos un experimento básico, en el que utilizamos un único núcleo, con tiempos de *context switch* y *core switch* cero, y usando un lote de tres tareas con *latencia* cero. El lote utilizado es el siguiente:

```
TaskCPU 10
TaskCPU 5
TaskCPU 7
```

Ejecutamos el scheduler y el diagrama arrojado fue el siguiente:

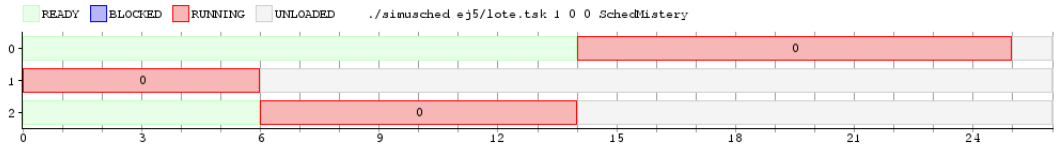

Figura 8: Experimento 1

Podemos observar en la Figura 8 que las tareas se ejecutaron en órden creciente con respecto a la duración de las mismas. Esto nos da la pauta de que cuando hay más de una tarea *ready*, el scheduler opta por aquella que menor tiempo de ejecución tenga. Por otra parte, también observamos que las tareas fueron ejecutadas de corrido, es decir, una vez que una tarea es asignada a un núcleo, este la ejecuta hasta que la misma finalice. Por lo tanto, parece no haber ningún *quantum* definido.

Ahora veamos cómo responde el scheduler cuando un núcleo está ejecutando una tarea y se carga una nueva tarea con duración menor a la actual. ¿Cambiará de tarea o seguirá ejecutando la misma?

Al igual que en el experimento anterior, utilizamos un único núcleo con tiempos de *context switch* y *core switch* cero. Utilizamos el siguiente lote de tareas:

```
TaskCPU 10
@1:
TaskCPU 3
TaskCPU 1
```

Ejecutamos el scheduler y el diagrama arrojado fue el siguiente:

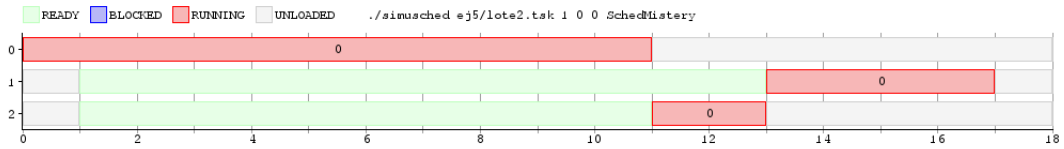

Figura 9: Experimento 2

Observando la Figura 9 concluimos que la respuesta a nuestra pregunta es que la tarea se continúa ejecutando. Esto nos permite dar cuenta de que cuando se carga un nueva tarea, el scheduler simplemente deja a la misma en espera, y recién cuando haya un núcleo disponible la tendrá en cuenta para asignársela al mismo, dependiendo de su duración con respecto al resto de las tareas en estado *ready*. Entonces, ya tenemos una buena idea de cómo funciona `SchedMistery`: cuando hay un núcleo disponible, le asigna la tarea *ready* que menor duración tenga, y una vez asignada, esta es ejecutada por dicho núcleo hasta finalizar. Esta descripción se corresponde con el funcionamiento de un scheduler `SJF` (*Shortest Job First*). Sin embargo, para estar seguros, hagamos un último experimento, para ver si responde de la misma manera cuando se dispone de más de un núcleo.

Haremos el experimento utilizando el mismo lote de tareas y tiempos de *context switch* y *core switch* que en el Experimento 2, pero usando dos núcleos.

Ejecutamos el scheduler y el diagrama arrojado fue el siguiente:

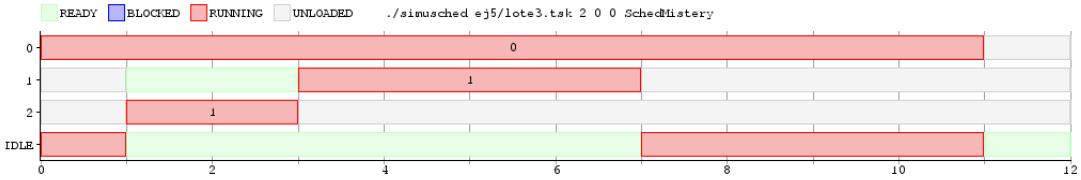

Figura 10: Experimento 3

Observemos que en el instante 0 la única tarea *ready* es la 0 y por ende la de menor duración. Entonces esta es asignada al núcleo 0. Luego, en el instante 1, tenemos dos tareas *ready*, y se le asigna al núcleo 1 la de menor duración, que es la 2. Por último el núcleo 1 finaliza con la tarea 2 y comienza a ejecutar la tarea 1, que es la única restante. Mientras tanto, el núcleo 0 continúa con la ejecución de la tarea 0. Entonces, confirmando nuestra hipótesis, con dos núcleos se comporta de la misma manera, y su funcionamiento corresponde al de un scheduler `SJF`.

Cabe aclarar, que en los experimentos no fuimos variando los parámetros *context switch* y *core switch* ya que dichos valores no aportan ninguna información sobre el funcionamiento particular del scheduler. Por esta razón los dejamos siempre con valor cero.

Para la implementación de `SchedNoMistery` simplemente utilizamos una cola de prioridad de pares <*pid, duración*>, y definimos el operador < (menor estricto) para pares, de manera que internamente la cola ordene las tareas como es deseado. En cuanto a las funciones del scheduler, estas son bastante simples:

- `load`: consiste simplemente en encolar un nuevo elemento a la cola de prioridad.
- `unblock`: no hace nada, ya que el enunciado dice que sólo funciona para tareas de tipo `TaskCPU`, y por ende esta función no debería ser utilizada en ningún momento.
- `tick`: si es llamada con motivo `EXIT`, devuelve la siguiente tarea de la cola u `IDLE` si esta está vacía. Si es llamada con motivo `TICK`, se continuará con la misma tarea,

exceptuando cuando la tarea acutal es `IDLE`, y hay al menos una tarea *ready* en la cola, caso en el que se cambia a la tarea de menor duración.

6. Ejercicio 6

6.1. Introducción

Se nos pide definir un nuevo tipo de tarea llamada `TaskPriorizada`, que utilice sólo CPU, y que reciba dos parámetros; el primero la prioridad (de 1 a 5 y cuanto más baja, más prioritaria), y el segundo la duración. Además debemos implementar un scheduler de tipo `PSJF` (*Preemptive Shortest Job First*) que reciba tareas de tipo `TaskPriorizada`.

A continuación, explicaremos la implementación de `SchedPSJF` y realizaremos algunas pruebas de ejecución que muestren las características esperadas del mismo.

6.2. Desarrollo

Al igual que el resto de los schedulers del trabajo práctico, `SchedPSJG` se implementó mediante una clase de C++ que hereda de la clase `SchedBase`. En este caso, definimos tres atributos internos que utiliza la clase, y son: una cola de prioridad de tuplas <*pid, prioridad, duración*>, un vector de enteros que contiene para cada procesador la prioridad de la tarea en ejecución y otro vector de enteros que contiene para cada procesador la duración de la tarea en ejecución. Al igual que con `SchedNoMistery`, tuvimos que definir un operador $<$ (menor estricto) para que la cola de prioridad pueda comparar las triplas de enteros.

Además definimos dos constantes que determinan la duración y prioridad de la tarea `IDLE`. Estas son `IDLE_PRIORITY` y `IDLE_DURATION`, con valores 6 y 0 respectivamente (la duración es irrelevante en el algoritmo, sin embargo la definimos por prolijidad).

Ahora veamos brevemente el funcionamiento de cada uno de los métodos ofrecidos por el scheduler:

- `constructor`: inicializa los atributos internos como si todas las cpu estuvieran ejecutando `IDLE`.
- `load`: encola la nueva tarea a la cola de prioridad.
- `unblock`: no hace nada, ya que el enunciado dice que sólo funciona para tareas de tipo `TaskCPU`, y por ende esta función no debería ser utilizada en ningún momento.
- `tick`: si es llamada con motivo `EXIT` se devuelve la siguiente de la cola, o `IDLE` si esta está vacía. Si es llamada con motivo `TICK` y se compara la tarea actual con la siguiente de la cola, y se cambia de tarea sólo si la actual es menos prioritaria. Si no es menos prioritaria, se continua con la misma tarea que se estaba ejecutando.

Observación: si bien el enunciado dice que al cargar una tarea, debemos comparar la actual contra la cargada, observar que la comparamos contra la siguiente de la cola, que podría no ser la tarea recién cargada. Sin embargo el funcionamiento del algoritmo es correcto, ya que si la tarea cargada no es la siguiente de la cola entonces no es la más prioritaria, y por ende no será más prioritaria que la que se está ejecutando.

Ahora veamos algunas ejecuciones de `SchedPSJF` que muestren las características esperadas del mismo. Utilizaremos un único procesador y costos de *context switch* y *core switch* igual a 0.

En el primer ejemplo utilizamos un set de cuatro tareas; dos de ellas tienen la misma prioridad, pero distinta duración, y las dos restantes tienen distintas prioridades que las otras.

```
TaskPriorizada 4 2
TaskPriorizada 1 5
TaskPriorizada 2 10
TaskPriorizada 2 5
```

Con este ejemplo buscamos ver la elección de tareas que realiza el scheduler al tener múltiples tareas en estado *ready* en el momento de asignarle una tarea a un procesador. Veamos si el diagrama de Gant refeleja el comportamiento esperado.

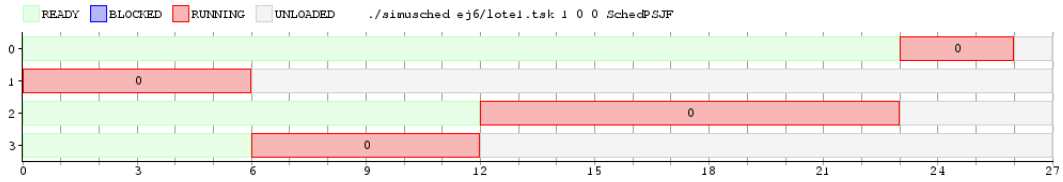

Figura 11: Ejemplo 1

Como todas las tareas son cargadas en el instante 0 se comienza a ejecutar en primer lugar la más prioritaria de las cuatro, que es la tarea 1, con prioridad 1. Luego, las tareas 2 y 3 tienen la misma prioridad, pero se ejecuta la número 2, dado que su duración es menor. Finalmente, se ejecutan las tareas 2 y 3, en ese orden. Hasta aquí, el comportamiento del scheduler, es el esperado.

Ahora observemos si ante la carga de una tarea más prioritaria, se desaloja la tarea en ejecución. Para ello, utilizaremos el siguiente lote de tareas:

```
TaskPriorizada 5 8
TaskPriorizada 3 6
@3:
TaskPriorizada 2 7
@8:
TaskPriorizada 2 5
```

Ejecutamos el scheduler y el diagrama arrojado fue el siguiente:

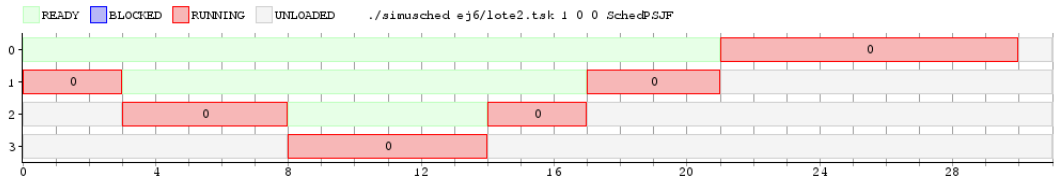

Figura 12: Ejemplo 2

En este caso, la tarea 1 es la primera en ejecutarse, ya que en el instante 0 es la más prioritaria. Luego, en el instante 3 se carga la tarea 2, que es más prioritaria que la 1, y por ende la 1 es desalojada y comienza a ejecutarse la 2. Poco después, en el instante 8 ocurre algo similar; se carga la tarea 3 y se desaloja la 2 para que esta comience a ejecutar. Sin embargo, en este caso, las prioridades coinciden, pero la duración de 3 es menor, por lo que se produce el cambio de tarea.

Por último, analizaremos un caso bien específico, en el que dos tareas coinciden tanto en su prioridad, como en su duración. El comportamiento esperado, es que se ejecuten por orden de llegada. Para este ejemplo proponemos el siguiente lote de tareas:

```
TaskPriorizada 2 5
TaskPriorizada 2 5
@10:
TaskPriorizada 3 7
@12:
TaskPriorizada 3 7
```

Ejecutamos el scheduler y el diagrama arrojado fue el siguiente:

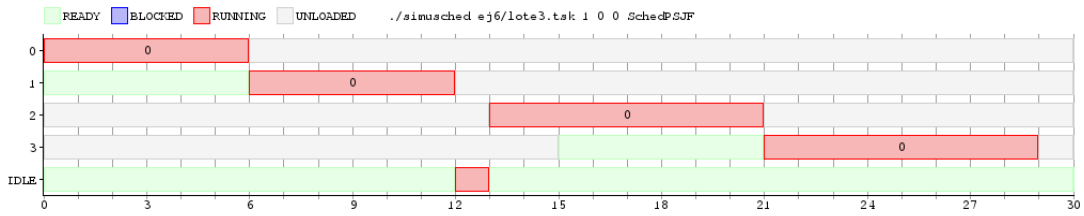

Figura 13: Ejemplo 3

Claramente observamos que el orden de llegada fue respetado, tanto entre las tareas 0 y 1 que se cargan en el instante 0, como así también entre 2 y 3. Observar que 3 fue cargada después que 2 y no produce el desalojo de 2, sino que se respeta el órden de llegada.

7. Ejercicio 7

7.1. Introducción

En este ejercicio, vamos a realizar una serie de experimentos comparando los algoritmos de scheduling `SJF`, `Round-Robin` y `PSJF`, con el objetivo de encontrar ventajas y desventajas en dos contextos particulares: cuando tenemos tareas críticas o con prioridad y cuando no es así, es decir, cuando todas las tareas tienen la misma importancia. Los experimentos consisten en simulaciones para 1 y 2 núcleos con un *context switch* de un ciclo. En particular, para el scheduler `Round-Robin` utilizaremos un quantum de cinco ciclos para cada procesador .

Dado que estos algoritmos toman distintos tipos de proceso (`SJF` y `Round-Robin` toman procesos de tipo `TaskCPU` mientras que `PSJF` admite de tipo `TaskPriorizada`), no es posible realizar una comparación con un lote de tareas común. Por lo tanto, vamos a realizar comparaciones con lotes particulares para cada tipo de scheduler, tratando así de obtener conclusiones acerca de los mismos. Haremos un análisis cuantitativo sobre los schedulers mediante el uso de las métricas como *latencia*, *waiting time* y *throughput*, como así también un análisis cualitativo haciendo referencia a las ventajas o desventajas que conlleva el uso de cada scheduler en determinado contexto de uso.

7.2. Experimentación

7.2.1. Experimento 1

Presentación

En el siguiente experimento, vamos a analizar el comportamiento de los tres algoritmos de scheduling mencionados en la introducción, en un contexto en el que no hay tareas más importantes o prioritarias que otras.

Realizaremos una comparación en términos de las métricas de latencia, waiting time y throughput con el siguiente lote de tareas:

```
TaskCPU 9
@4:
TaskCPU 6
@4:
TaskCPU 11
@7:
TaskCPU 8
```

Dado que `PSJF` sólo admite tareas de tipo `TaskPrioritaria`, para este scheduler utilizaremos el lote presentado a continuación, que define a todas las tareas con la misma prioridad.

```
TaskPrioritaria 5 9
@4:
TaskPrioritaria 5 6
@4:
TaskPrioritaria 5 11
@7:
TaskPrioritaria 5 8
```

Hipótesis

Teniendo en cuenta el lote utilizado, creemos que los algoritmos de scheduling `SJF` y `PSJF` tendrán comportamientos similares, exceptuando la situación en la que se carga un proceso de menor duración que el que se está ejecutando. En dicho caso, un scheduler `PSJF`, al ser de tipo *preemptive*, realizará un desalojo de la tarea en ejecución, y asignará al procesador la tarea recientemente cargada. Por su parte, un scheduler `SJF` no hará desalojos.

Por otro lado, debido a la cantidad de cambios de contexto y desalojos que se producen en el `Round-Robin`, esperamos que el waiting time sea mayor que el arrojado por los otros dos schedulers. Sin embargo, la latencia del `Round-Robin` será menor, ya que la longitud de la ronda de ejecución por tarea está acotada por el quantum definido. Con respecto al throughput, estimamos que al no invertir tantos ciclos en en cambios de contexto, `PSJF` y `SJF` tendrán mayores valores.

Resultados

Los diagramas obtenidos a partir de la ejecución de los schedulers con los lotes especificados y un único núcleo fueron los siguientes:

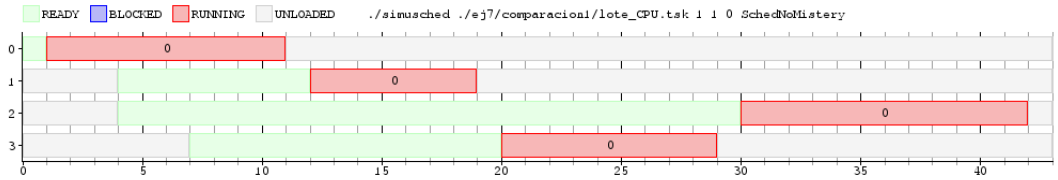

Figura 14: Simulación SJF (un núcleo)

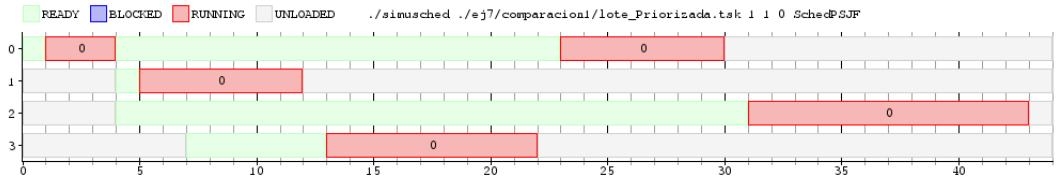

Figura 15: Simulación PSJF (un núcleo)

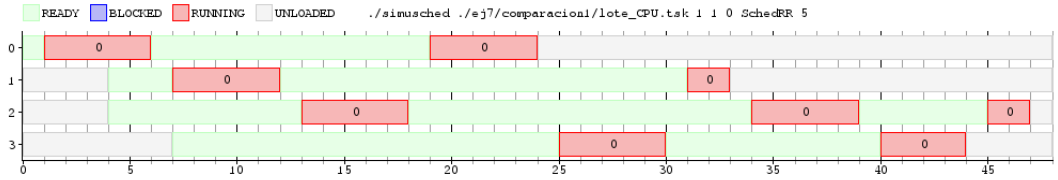

Figura 16: Simulación RR (un núcleo)

A primera vista, podemos ver como la ejecución de `SJF` y `PSJF` son similares, con la salvedad de que se realiza un desalojo intermedio en el segundo caso. En cuanto a latencia, waiting time y throughput, veamos los resultados de manera mas precisa:

Resultados: Latencia			
Tarea	SJF	PSJF	RR
0	1	1	1
1	8	1	3
2	26	27	9
3	13	6	18
Promedio	12	8.75	7.75

Resultados: Waiting Time			
Tarea	SJF	PSJF	RR
0	1	20	14
1	8	1	22
2	26	27	31
3	13	6	28
Promedio	12	13.5	23.75

Resultados: Throughput	
SJF	0.093
PSJF	0.095
RR	0.085

Los diagramas obtenidos a partir de la ejecución de los schedulers con los lotes especificados y dos núcleos fueron los siguientes:

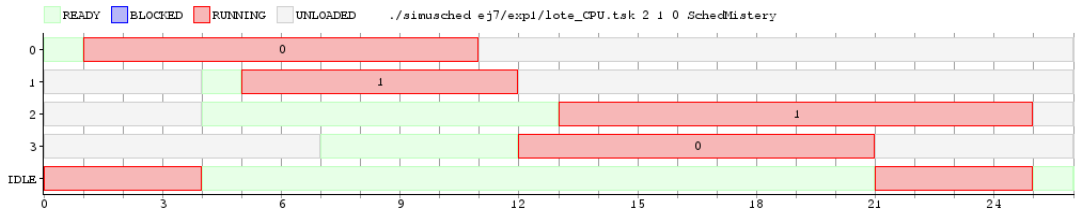

Figura 17: Simulación SJF (dos núcleos)

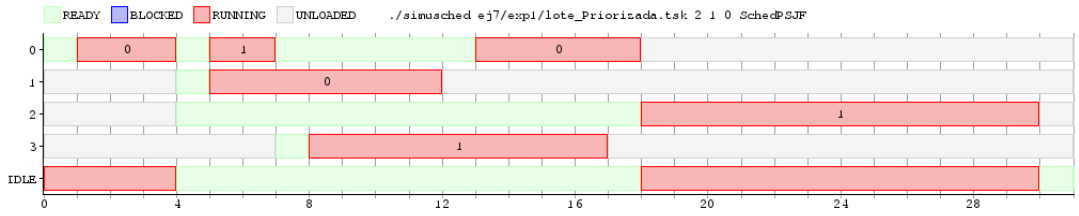

Figura 18: Simulación PSJF (dos núcleos)

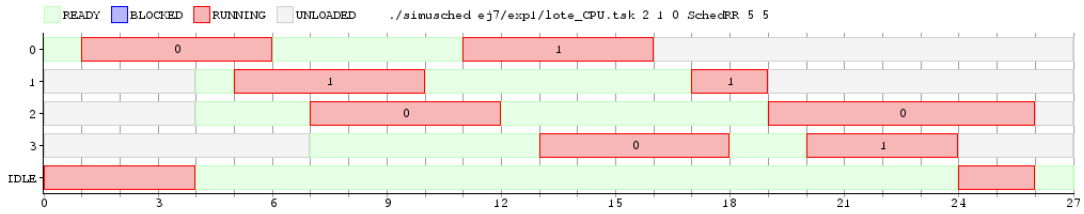

Figura 19: Simulación RR (dos núcleos)

Al igual que en los diagramas de las simulaciones con un núcleo, podemos notar que `SJF` y `PSJF` son similares. En cuanto a latencia, waiting time y throughput, veamos los resultados de manera mas precisa:

Resultados: Latencia			
Tarea	SJF	PSJF	RR
0	1	1	1
1	1	1	1
2	9	14	3
3	5	1	6
Promedio	4	4.25	2.75

Resultados: Waiting Time			
Tarea	SJF	PSJF	RR
0	1	8	6
1	1	1	8
2	9	14	10
3	5	1	8
Promedio	4	6	8

Resultados: Throughput	
SJF	0.160
PSJF	0.133
RR	0.154

Conclusiones

Analizando los resultados, podemos concluir que en casi todos los casos se cumplieron nuestras hipótesis. En primer lugar, el aspecto *preemptive* del `PSJF` tuvo un impacto importante en términos de latencia en la simulación que utiliza un único núcleo, ya que al cargarse un proceso con mayor prioridad se desalojó el que estaba en ejecución, reduciendo así la latencia del mismo. En el caso de dos núcleos también hubo una diferencia de latencia entre estos dos, pero no tan importante como la anterior. Por su parte, podemos ver cómo `Round-Robin`, al priorizar el aspecto de *fairness*, va alternando la ejecución entre las tareas y esto provoca una disminución marcada de la latencia, con un consecuente incremento del waiting time, producto del elevado número de cambios de contexto.

En cuanto a la cantidad de procesos finalizados por unidad de tiempo, es prácticamente igual entre `SJF` y `PSJF` para un núcleo, mientras que `Round-Robin` resultó ser un poco menor, como habíamos predicho. Sin embargo, en contra de nuestra suposición, para dos núcleos,

`Round-Robin` resultó tener un *throughput* casi igual que `SJF` e incluso mayor que `PSJF`. Observando los diagramas, concluimos que este resultado se debe a la constante rotación entre las tareas, realizada en `Round-Robin`, la cual produjo una mejor distribución de la ejecución de tareas entre ambos núcleos. Por el contrario, tanto en el diagrama de `SJF`, como en el de `PSJF` podemos observar que esto no fue así; y queda evidenciado por la cantidad de ciclos que hay un procesador en `IDLE`.

7.2.2. Experimento 2

Presentación

En el siguiente experimento, vamos a analizar el comportamiento de los tres algoritmos de scheduling mencionados en la introducción, en un contexto de uso en el que hay tareas más importantes o críticas que otras. En este experimento habrá una tarea crítica, considerada más importante que el resto, y por ende es necesario ejecutarla lo antes posible.

Para `SJF` y `Round-Robin` utilizaremos el siguiente lote de tareas:

```
TaskCPU 5
TaskCPU 3
TaskCPU 1
TaskCPU 5
TaskCPU 2
TaskCPU 10
@5:
TaskCPU 4
```

Para `PSJF` utilizaremos el siguiente lote de tareas, en el que se especifican las prioridades de cada una de las tareas, además de su duración:

```
TaskPriorizada 5 5
TaskPriorizada 5 3
TaskPriorizada 5 1
TaskPriorizada 5 5
TaskPriorizada 5 2
TaskPriorizada 5 10
@5:
TaskPriorizada 1 4
```

Hipótesis

Para este experimento esperamos que `PSJF` sea el scheduler que anteriormente ejecute la tarea crítica (la última del lote), mientras que `Round-Robin` y `SJF` demorarán más debido a que es la última en cargarse. Dado que el resto de las tareas se cargan todas juntas en el instante 0 no habrán cambios de contexto adicionales en `PSJF`, al menos en la ejecución con un único núcleo. Por lo tanto no esperamos que el waiting time difieran demasiado

Resultados

Los diagramas obtenidos a partir de la ejecución de los schedulers con los lotes especificados y un único núcleo fueron los siguientes:

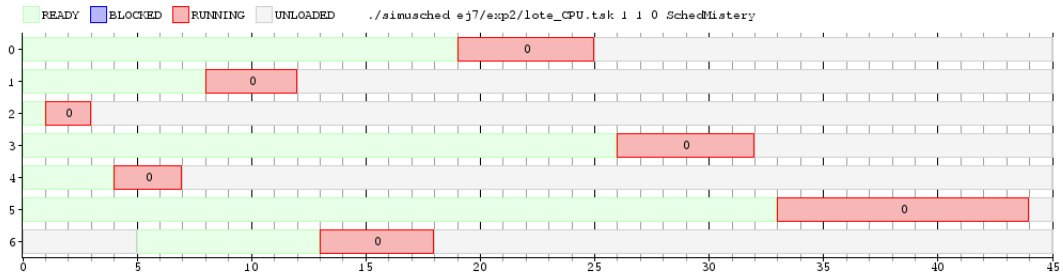

Figura 20: Simulación SJF (un núcleo)

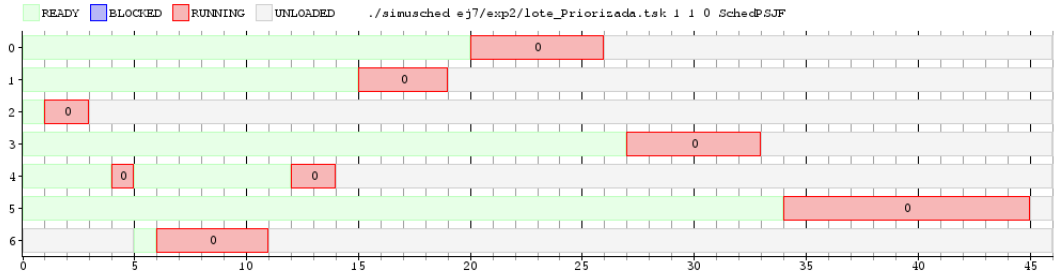

Figura 21: Simulación PSJF (un núcleo)

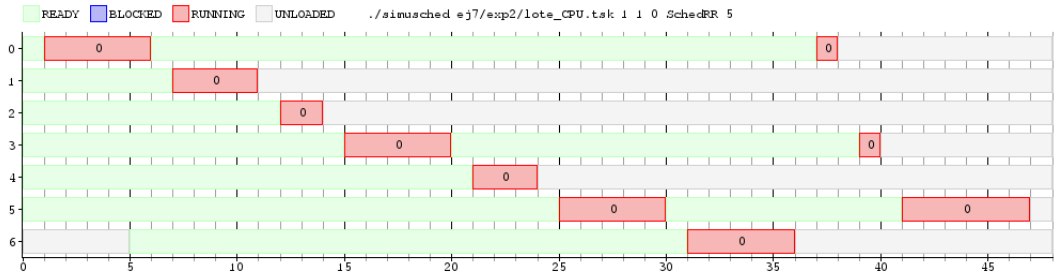

Figura 22: Simulación RR (un núcleo)

Resultados: Latencia			
Tarea	SJF	PSJF	RR
0	19	20	1
1	8	15	7
2	1	1	12
3	26	27	15
4	4	4	21
5	33	34	25
6	8	1	26
Promedio	14.14	14.57	15.28

Resultados: Waiting Time			
Tarea	SJF	PSJF	RR
0	19	20	32
1	8	15	27
2	1	1	12
3	26	27	34
4	4	11	21
5	33	34	36
6	8	1	26
Promedio	14.14	15.57	26.86

Resultados: Throughput	
SJF	0.159
PSJF	0.156
RR	0.149

Los diagramas obtenidos a partir de la ejecución de los schedulers con los lotes especificados y dos núcleos fueron los siguientes:

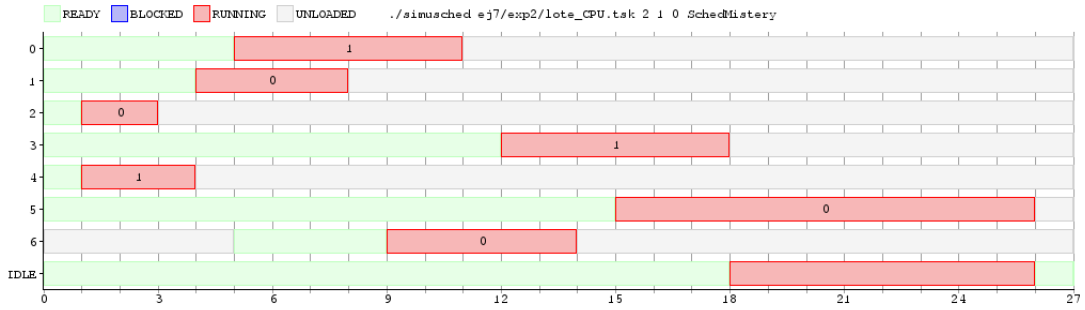

Figura 23: Simulación SJF (dos núcleos)

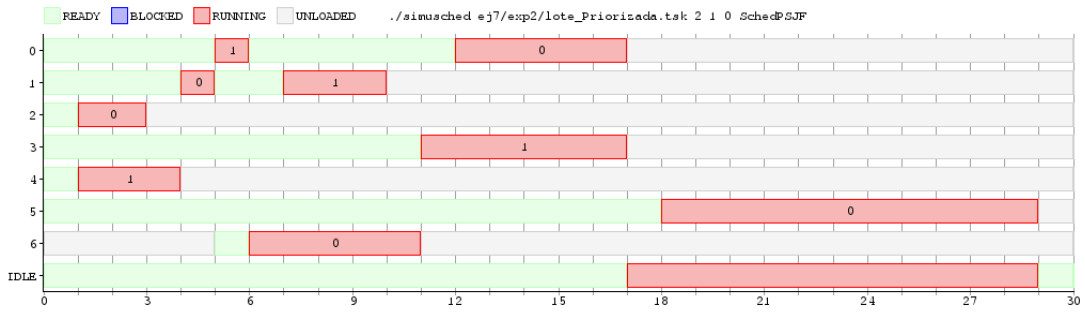

Figura 24: Simulación PSJF (dos núcleos)

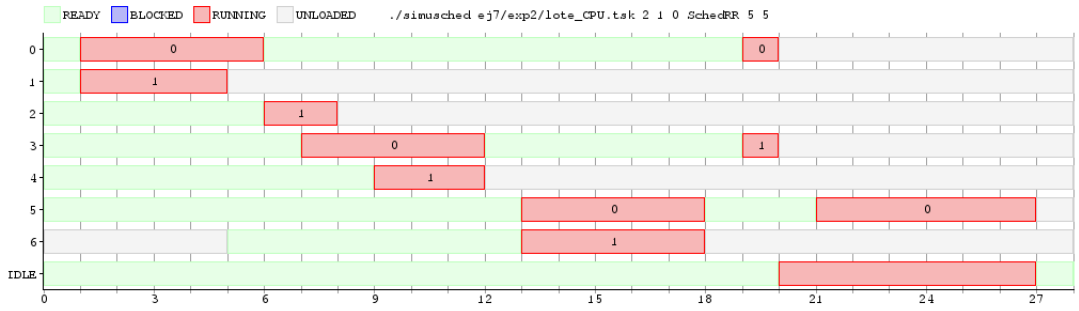

Figura 25: Simulación RR (dos núcleos)

Resultados: Latencia			
Tarea	SJF	PSJF	RR
0	5	5	1
1	4	4	1
2	1	1	6
3	12	11	7
4	1	1	9
5	15	18	13
6	4	1	8
Promedio	6	5.86	6.43

Resultados: Waiting Time			
Tarea	SJF	PSJF	RR
0	5	11	14
1	4	6	1
2	1	1	6
3	12	11	14
4	1	1	9
5	15	18	16
6	4	1	8
Promedio	6	7	9.71

Resultados: Throughput	
SJF	0.269
PSJF	0.241
RR	0.259

Conclusiones

Analizando los resultados presentados anteriormente se puede concluir que sólo partes de nuestra hipótesis fueron validadas.

La tarea 6 tuvo mucha menor latencia y waiting time en *PSJF*, pero el impacto en las otras tareas no fue el esperado. En las tablas se puede apreciar que si bien la latencia de las otras tareas en promedio aumentó, no ocurrió lo mismo con el waiting time como habíamos pensado desde un principio. Esto se debe a que algunas de las tareas planteadas tenían una

duración mayor a la del *quantum* fijado en *Round Robin*, haciendo que deban ser desalojadas para dejarle el lugar a otra tarea, debiendo esperar a que sea nuevamente su turno.

Cabe destacar que dependiendo de la circunstancia pueda ser benéfico pagar los incrementos en la latencia para ejecutar una tarea con mayor prioridad. Por ejemplo en el lote utilizado para experimentar la tarea 6 tenía prioridad 1, por lo que era una tarea crítica. Esta tarea en *Round Robin* hubiese tenido que esperar un tiempo mucho mayor para ser ejecutada, siendo este un comportamiento no deseado. A su vez, hay casos en los que tener un scheduler que maneje prioridades puede ser contraproducente, si la tarea 6 hubiese tenido prioridad 4, esta hubiese tenido una prioridad apenas mayor que la del resto de las tareas ya cargadas, haciendo que tal vez no sea tan necesario que ésta tarea se ejecute inmediatamente ya que como vimos con anterioridad esto incrementa la latencia del resto de las tareas. Por todo lo anteriormente dicho concluimos entonces que es necesario estudiar bien el contexto antes de elegir que políticas va a utilizar el scheduler planteado.