Sistemas Operativos
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1: Scheduling

Adrian Vinocur	adrian.vinocur@gmail.com
Alejandro Albertini	ale.dc@hotmail.com
Raul Brum	brumraul@gmail.com

Resumen

En este trabajo estudiaremos el problema de planificar (scheduling) la ejecución de tareas. Analizaremos de manera experimental algunos de los algoritmos clasicos como FCFS, Round Robin. Además analizaremos otros que utilizan prioridades para resolver el problema de planificación en tiempo real como RM (Rate Monotonic) y EDF (Earlier Deadline First).

Palabras Claves: Scheluder, Scheduling, FCFS, Round Robin, RM, Rate Monotonic, EDF, Earlier Deadline First, Quantum

Índice

1. Introducción

En el presente trabajo nos proponemos a implementar la lógica de varios planificadores de tareas.

Nuestro trabajo consistirá en implementar, utilizando las estructuras provistas por la cátedra, ciertos algoritmos de *scheduling* en lenguaje c++. Realizaremos experimentos empíricos e intentaremos hallar el *quantum* óptimo para el algoritmo de *Round Robin*, con respecto a varias métricas. Estas métricas podrían implicar criterios contradictorios, por lo que el objetivo será encontrar un balance adecuado entre ellas.

Además analizaremos un paper sobre scheduling basados en prioridades para procesos en tiempo real con ciertos requerimientos de deadlines obligatorios. Implementaremos estos últimos y buscaremos ejemplos para ilustrar el comportamiento de estos.

El trabajo se encuentra dividido en 10 ejercicios cuya meta de estos es abarcar lo anteriormente comentado. Se decidio omitir una sección resultados agregando los gráficos directamente en el desarrollo. Lo que se busca con esta estructuración del informe es mantener cada ejercicio con explicaciones y gráficos juntos para comodidad del lector.

2. Desarrollo

2.1. Ejercicio 1

La tarea TaskConsola simula una tarea interactiva en donde realiza n llamadas bloqueantes de duración de tiempo aleatoria entre $bmin$ y $bmax$. Se pasa por parámetro la variable pid, y un vector con las variables n, $bmin$ y $bmax$, en ese orden.

Se coloca un ciclo *for*, donde se itera n veces y por cada iteración se produce una llamada bloqueante de tiempo aleatorio entre $bmin$ y $bmax$, para un respectivo pid. Ese número aleatorio se arma usando una función llamada "$rand()$" que genera un número aleatorio cualquiera y lo que hacemos es dividirlo por la diferencia entre $bmax$ y $bmin + 1$, quedándonos con el resto y a eso sumarle $bmin$ para acomodarlo entre el rango $[bmin, bmax]$.

2.2. Ejercicio 2

En este ejercicio escribimos un lote de 3 tareas con 1 intensiva de CPU y 2 *taskConsola*. El objetivo es ver un ejemplo interesante donde el algoritmo FCFS no tiene un buen rendimiento, especialmente para procesadores de un núcleo.

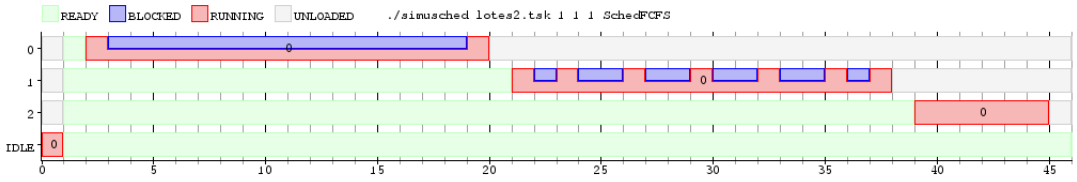
A continuación mostraremos el lote de tareas elegido:

```
TaskConsola 1 15 20
@1:
TaskConsola 6 1 2
@1:
TaskCPU 5
```

Las 3 tareas llegan en el mismo instante (en el instante 1), la primer taskConsola produce una sola llamada bloqueante pero entre 15 y 20 ticks de reloj, la segunda taskConsola ejecuta 6 llamadas bloqueantes pero cada una entre 1 y 2 ticks de reloj y la tarea CPU utiliza 5 ticks de reloj.

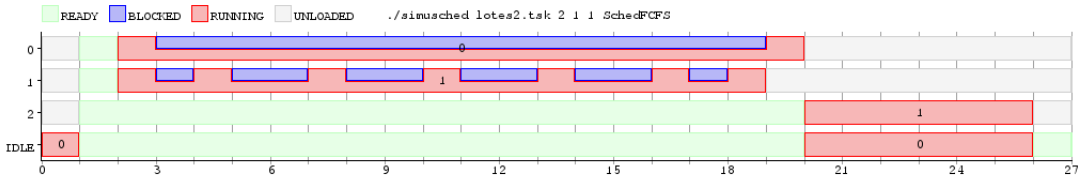
Básicamente tenemos:

- 1 intensiva en CPU.
- 1 *taskConsola* que solo ejecuta una llamada bloqueante pero de muchos ciclos de reloj.
- 1 *taskConsola* que ejecuta 6 llamadas bloqueantes pero de pocos ciclos de reloj.

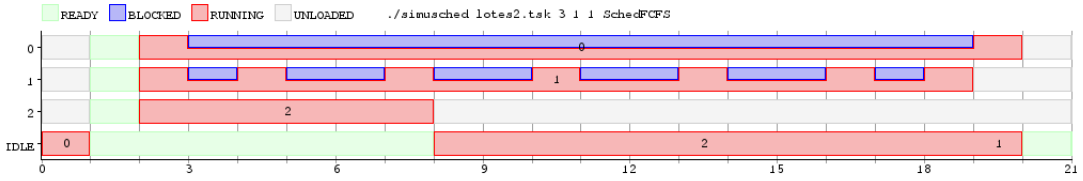


Para un solo núcleo el problema que surge para la tarea CPU es que como el scheduler decidió que primero se ejecuten las dos taskConsola (a pesar de haber llegado las 3 al mismo tiempo), la tarea CPU tiene que esperar a que terminen las dos taskConsola para luego poder ejecutar solo 5 ciclos de reloj.

Si la tarea CPU hubiese llegado 1 solo ciclo de reloj antes se podría haber evitado la espera de las dos taskConsola y se hubiese ejecutado primero, reduciendo muchísimo su tiempo de espera. Otros de los puntos negativos es que la primer taskConsola tiene un tiempo de bloqueo muy grande. Esto hace que se pierda mucho tiempo esperando que se resuelva. La segunda no tiene ningún bloqueo grande pero tiene 6 bloqueos de 1 ó 2 ticks de reloj, lo que hace que se pierda mucho tiempo con la CPU bloqueada, esperando por ejemplo algún evento de entrada/salida.



En el caso en que tenemos 2 núcleos cambia mucho el panorama con respecto al de un núcleo. Porque si bien otra vez la tarea CPU no es la elegida por el scheduler para comenzar, arrancan las dos taskConsola a la vez y la taskConsola2 de muchos ticks cortos no tiene que esperar a que termine de ejecutarse la primera, evitando así la taskConsola de un bloqueo grande, reduciendo muchísimo el tiempo de espera de la taskConsola2.



Y el caso donde tenemos 3 núcleos es el ideal, ya que ninguna de las tareas tiene que esperar a que termine de ejecutarse la otra, todas empiezan a ejecutarse inmediatamente después de llegar al scheduler reduciendo drásticamente el tiempo de espera con respecto a los casos anteriores.

Tick final de la ejecución de la última de las 3 tareas:

- 1 núcleo: 45
- 2 núcleos: 26
- 3 núcleos: 20

Mediciones del tiempo de espera para el algoritmo FCFS (en ticks unidades de tiempo):

- Para 1 núcleo: 19,66
- Para 2 núcleos: 10,25 (52,14 % con respecto al de 1 núcleo)
- Para 3 núcleos: 2,5 (12,72 % con respecto al de 1 núcleo)

Mediciones del turnaround para el algoritmo FCFS (en ticks unidades de tiempo):

- Para 1 núcleo: 33,33
- Para 2 núcleos: 22 (66 % con respecto al de 1 núcleo)
- Para 3 núcleos: 16 (48 % con respecto al de 1 núcleo)

2.3. Ejercicio 3

El algoritmo de scheduling de Round Robin[1] (RR) está diseñado específicamente para cumplir con las necesidades de los sistemas de *time-sharing*. Es similar al scheduler FCFS, con la adición de desalojo (o preemption), lo cual le permite al sistema intercambiar los procesos en ejecución. Se define un quantum, que es una pequeña unidad de tiempo, por lo general de 10 a 100 milisegundos de longitud. La cola de "listos" (ready) se trata como una cola circular. El scheduler recorre la cola, asignando CPU a cada proceso por a los sumo 1 quantum de tiempo. Para implementar RR, mantenemos la cola de ready como una cola FIFO de procesos. Los nuevos procesos son añadidos al final de la cola. El CPU toma el primer proceso de la cola de ready, establece un timer para interrumpir luego de 1 quantum, y realiza el dispatch del proceso.

Luego pueden suceder una de dos cosas:

- El proceso utiliza CPU por menos que 1 quantum. En este caso el proceso mismo devolverá el CPU voluntariamente (ya sea porque terminó o porque se bloqueó). El scheduler procederá entonces con el siguiente proceso en la cola de ready.
- El proceso continúa corriendo luego de 1 quantum. En este caso el timer se disparará, elevando una interrupción al sistema operativo. Entonces se realizará un cambio de contexto, en el cual el scheduler desalojará al proceso, que sera ubicado al final de la cola de ready, y procederá con el siguiente proceso en la mencionada cola.

En el algoritmo de Round Robin, a ningún proceso se le asigna CPU por más tiempo que un quantum (a menos que sea el único proceso que pueda ser ejecutado). Si un proceso excede el tiempo de un quantum en CPU, el mismo es desalojado y puesto nuevamente en la cola. Round Robin es por lo tanto un algoritmo con desalojo. Si hay n procesos en la cola de ready y el quantum es q, entonces cada proceso recibe $1/n$ de tiempo de CPU en pedazos de a lo sumo q unidades de tiempo. Cada proceso esperará a lo sumo $(n-1) * q$ unidades de tiempo hasta su próximo quantum. La performance del algoritmo de Round Robin depende, como veremos más adelante, del tamaño del quantum. En un extremo, si el quantum es demasiado largo, se comporta como si fuera FCFS. En contraste, si el quantum es extremadamente corto, esto se denomina procesador compartido, y (en teoría) crea la apariencia de que cada uno de los n

procesos cuenta con su propio procesador corriendo a $1/n$ veces la velocidad real del procesador.

En esta sección nos propusimos simular un algoritmo de Round Robin que permita el intercambio entre procesadores. Esto significa, en este caso, que cada proceso recibirá el primer procesador libre que se encuentre, sin importar que sea o no el mismo procesador en el que se haya ejecutado previamente. Para lograrlo, utilizamos las siguientes estructuras auxiliares:

- Una cola de ready, que contiene los PID de los procesos que están listos para ser ejecutados.
- Un mapa que almacena el proceso y el tiempo restante que cuenta para correr en cada core.

El scheduler se inicializa con la tarea IDLE corriendo en cada core. El quantum asignado a la tarea IDLE es siempre cero, para que pueda ser desalojada a continuación por cualquier tarea "real" que requiera el uso de CPU.

Con cada tick del reloj, se evalúa qué hacer según el motivo del mismo:

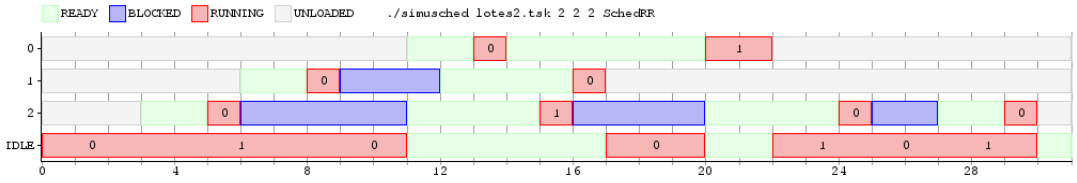
- Si es un bloqueo, se remueve al proceso del CPU y el mismo renuncia a su quantum restante. El mismo será colocado de vuelta al final de la cola de ready cuando se reciba la señal de que se ha desbloqueado. Si hay un proceso disponible en la cola de ready, se le asignará ese core y se establecerá su tiempo disponible como el total del quantum. Caso contrario, se asignará un ciclo de ejecución en ese core a la tarea IDLE.
- Si el proceso ha finalizado, se remueve el proceso del CPU, y se procede al siguiente en la cola como en el caso anterior.
- Si el proceso continúa en ejecución, se decrementa su tiempo disponible en CPU. Si ha excedido su quantum, será desalojado y puesto en el final de la cola de ready. Se retornará como proceso siguiente al primero de la cola, que podría ser el mismo proceso si no hubiera otro disponible.

2.4. Ejercicio 4

Se realizaron distintas corridas del algoritmo Round Robin, para ilustrar su comportamiento:

Corrida 1:

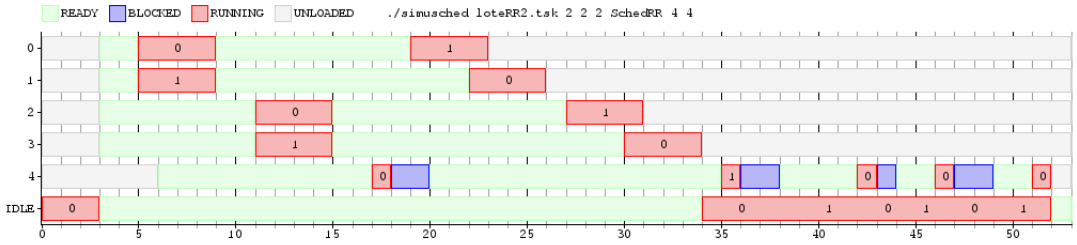
```
@11:
TaskCPU 2
@6:
TaskConsola 1 3 4
@3:
TaskConsola 3 2 6
```



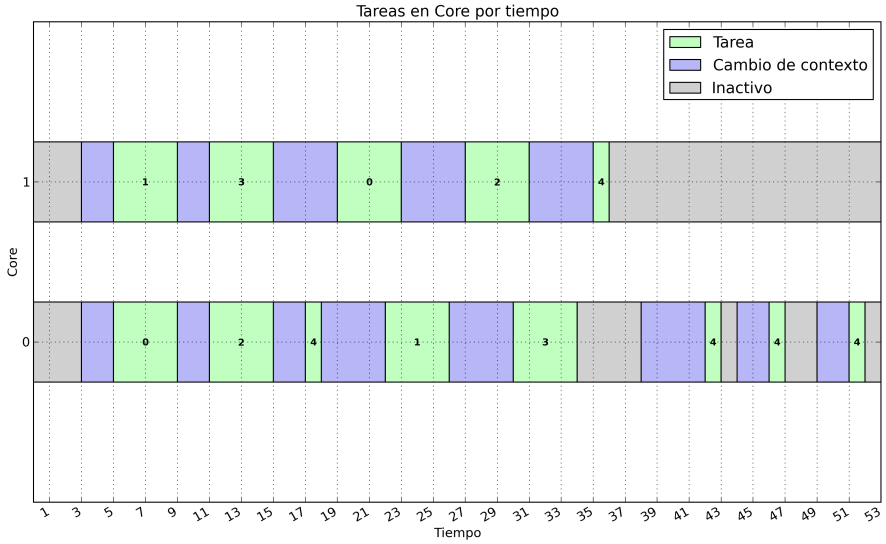
Como se puede observar, los procesos liberan el CPU al bloquearse. También se observa que los mismos cambian de procesador, como es el caso del 0 y el 2. Se nota en estos casos la penalidad de 2 ciclos por cambio de procesador (añadida a los 2 ciclos de cambio de contexto). En este caso el quantum de ambos procesadores es 4.

Corrida 2:

```
@3:
*4 TaskCPU 7
@6:
TaskConsola 4 1 3
```



En este caso corren 4 tareas de cpu, y una de consola. El quantum es de 4 ciclos y se cuenta con 2 procesadores. Se puede observar como las tareas comparten el uso de CPU y son desalojadas al finalizar su quantum. También se observa que el scheduler, por simplicidad de implementación, no identifica en qué procesador estaba corriendo la tarea, causando una penalidad extra de cambio de procesador. También se observa que los tiempos de espera son bastante elevados en este caso.

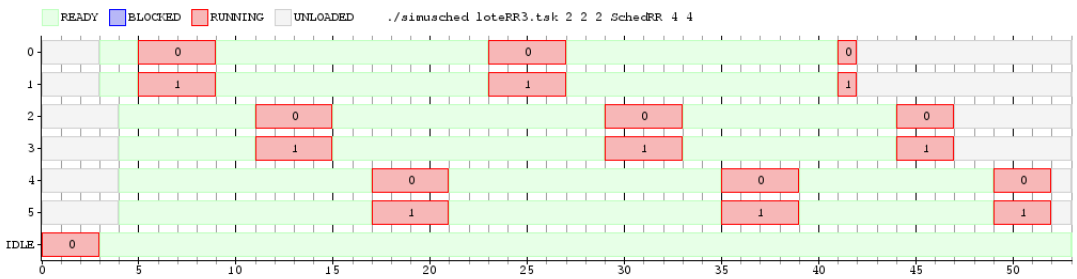


En el gráfico anterior se observa más claramente que el algoritmo no es muy eficiente en este caso, ya que se está inviertiendo mucho tiempo en cambios de contexto (es casi lo mismo que el tiempo de ejecución real de las tareas).

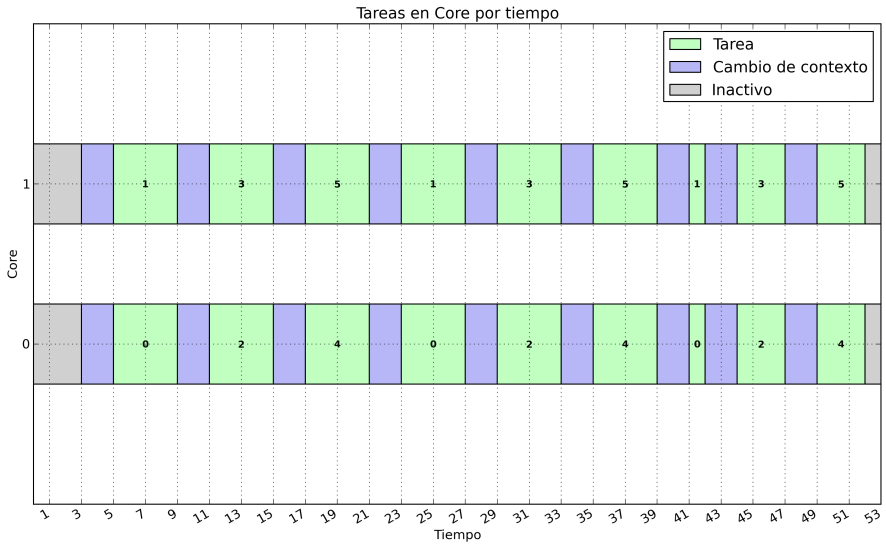
Corrida 3:

```
@3:
*2 TaskCPU 8
@4:
*4 TaskCPU 10
```

Se corrió en este caso con 2 cores y quantum 4. En este caso no se incluyeron tareas interactivas.



Como se observa en los gráficos, este es un caso más "feliz", donde el costo de cambio de contexto es mucho menor, y la utilización del CPU es más eficiente (más uso en ejecución de las tareas que en cambios de contexto)



2.5. Ejercicio 5

El paper[2] se centra en la problemática de programar varias tareas con un solo procesador en un determinado contexto. Dicho contexto establece que las tareas tiene que cumplir un deadline y hay que garantizar el cumplimiento de éste.

Analizan únicamente schedulers que utilizan prioridades y que adminten desalajo (preemptive) de tareas (En particular RM, EDF y una versión mixta).

El algoritmo de la sección 7 lo introducen por las limitaciones de RM.

La cota que definen los autores para poder saber si un conjunto de N tareas es factible para N grande es aproximadamente de $ln(X)$ es decir un poco menos del 70 % de utilización de CPU. Es decir si el conjunto de N tareas utiliza aprox menos de $ln(X)$ de tiempo de CPU entonces se puede programar con RM cumpliendo los deadlines, si no cumple esa cota dependerá del conjunto y los periodos de las tareas para determinar si es factible o no.

El algoritmo de EDF proporciona una cota de factibilidad mucho mejor, si y solo si es factible programar un conjunto de tareas si su procesamiento no supera el 100 % de uso de CPU.

Esta cota de factibilidad de programación de tareas que tiene EDF implica que un conjunto de tareas puede ser programado respetando deadlines por un scheduler si y solo si EDF es capaz de hacerlo.

El teorema 7 deduce la cota de factibilidad de programación de un conjunto de tareas utilizando el algoritmo EDF.

Consiste en analizar el consumo de CPU que asigna a cada tarea el algoritmo EDF en un rango de tiempo grande. Sumando estos tiempos se obtiene el uso total de CPU asignado a tareas en ese rango.

La demostración de que si las tareas usan más del 100 % no se pueden programar con un solo CPU es simple dado que vale para cualquier scheduler, en ese sentido de la demostración ni siquiera usa que es EDF.

La demostración de que si las tareas tiene un consumo de CPU total no mayor al 100 % entonces se pueden programar con EDF es un poco más compleja y requiere de un teorema anterior de ese paper que dice que "Cuando se usa EDF no hay tiempo desperdiciado (procesador IDLE) antes de un overflow (incumplimiento de deadline)". Gracias a esta propiedad de EDF en el teorema 7 se demuestra (utilizando el absurdo) que se pueden programar las tareas.

Diseño de SchedFixed:

Se implemento con una cola de prioridad para los procesos listos, donde se asignó mayor prioridad a las tareas que tenían menor período (i.e. mayor frecuencia).

Además generalizamos este algoritmo para más de un core y con desalajo por bloqueo. Nota: Sabemos que no era necesario porque en el paper solo se menciona un core y un solo recurso el CPU.

Al programar mantuvimos el invariante de que en cada momento se trata de ejecutar la tarea de mayor prioridad comparando en cada tick la tarea que esta corriendo en el core con las que estan esperando en la cola de listos. En caso de encontrar una tarea más prioritaria en la cola de listo, se desaloja la que esta corriendo (Se hace un swap entre estas tareas).

Diseño de SchedDynamic:

Se implementó con una cola de prioridad para los procesos listos y adicionalmente con un diccionario para guardar los deadlines, si bien el deadline se guarda en la cola de listos junto a su pid en una tupla. Cuando se necesita retirar de la cola de listos (ejemplo: cuando se bloquea o esta ejecutando en algún core) se guarda temporalmente en la otra estructura.

También en este caso generalizamos a varios cores e implementamos el desalojo por bloqueo.

Al programar mantuvimos el invariante de que en cada momento se trata de ejecutar la tarea cuyo deadline esta más próximo. Para esto se compara en cada tick la tarea que esta corriendo contra las tareas de la cola de listos, si se encuentra una que tiene deadline más proximo, se hace un swap de las tareas.

2.6. Ejercicio 6

La tarea taskBatch lo que hace recibir dos parámetros total_cpu y cant_bloqueos, y realizar cant_bloqueos llamadas bloqueantes en momentos aleatorios. Cada llamada bloqueante durará 1 tick de reloj y el uso total de cpu va a ser total_cpu.

Lo primero que hacemos es preguntar si total_cpu es menor que cant_bloqueos ya que si eso pasa, la taskBatch no va a ser posible realizarla de forma correcta ya que necesitamos 1 tick de reloj por cada llamada bloqueante.

Luego construimos un vector de unos y ceros, donde las posiciones van a simular ticks de reloj. Generamos el número aleatorio y vamos rellenando de ceros el vector hasta que llegamos a la posición del número aleatorio y ahí seteamos con 1 para saber en qué momentos hacer las llamadas bloqueantes.

A medida que vamos completando el vector, reducimos en 1 la cantidad total de ticks que tenemos y la cantidad de bloqueos, para no realizar mas llamadas bloqueantes de las que tenemos que realizar.

Una vez que terminamos de construir el vector, creamos una variable que llamada *cpu_acumulado*, a fin de saber cuánto CPU será necesario utilizar hasta que se realice la proxima llamada bloqueante.

Llamamos a la función uso_cpu, luego hacemos una llamada bloqueante de 1 tick de reloj de duración. Reiniciamos a 0 la variable *cpu_acumulado*, y repetimos lo mismo hasta cumplir con todas las llamadas bloqueantes.

Por último, si realizamos todas las llamadas bloqueantes pero no agotamos todo el CPU asignado para ese proceso (*total_cpu*), entonces realizamos una llamada con *uso_cpu* para todo el CPU restante.

2.7. Ejercicio 7

Como mencionamos anteriormente, la duración del quantum resulta de vital importancia para la performance del scheduler de Round Robin. Si el quantum es demasiado largo, el algoritmo tiende a comportarse como FCFS, extendiendo mucho los tiempos de espera. Si por el contrario el quantum es demasiado corto, se pagará el costo de estar cambiando de contexto muy a menudo.

Utilizaremos dos métricas para evaluar la performance del scheduling, que intentaremos optimizar:

- **Tiempo de Espera** es la suma del tiempo que un proceso pasa en la cola de listos. Indica cuánto tiempo tuvo que esperar un proceso para ejecutarse.

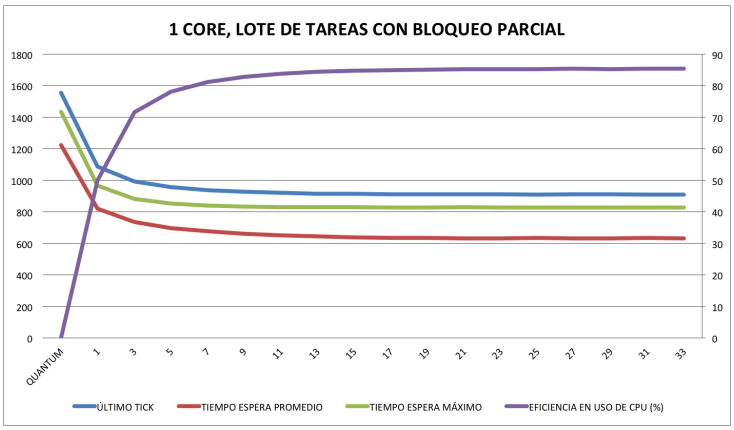
- **Eficiencia de CPU** lo tomaremos como el tiempo que se estuvo utilizando el CPU para realizar procesamiento, en relación con el tiempo total que le insumió al sistema completar las tareas. Nos dará una medida de la eficiencia con la que están siendo utilizados los recursos. Es decir, contaremos los cambios de contexto y los ciclos IDLE como ineficientes, mientras que contaremos como ciclos productivos aquellos en los que las tareas se están ejecutando.

También tendremos en cuenta el tiempo de ejecución (el último tick del reloj de la corrida). Para el tiempo de espera, analizaremos lo que sucede con el tiempo de espera promedio y con el valor máximo. Intentaremos establecer el quantum óptimo en distintos escenarios. Parae ello, ante un lote de tareas de tipo *TaskBatch*, variaremos los parámetros de las mismas para generar lotes de tareas poco, parcial y altamente bloqueantes. Variaremos la cantidad de cores en configuraciones de 1, 2, 4, 6 y 8 e intentaremos hallar el quantum para cada una de estas configuraciones.

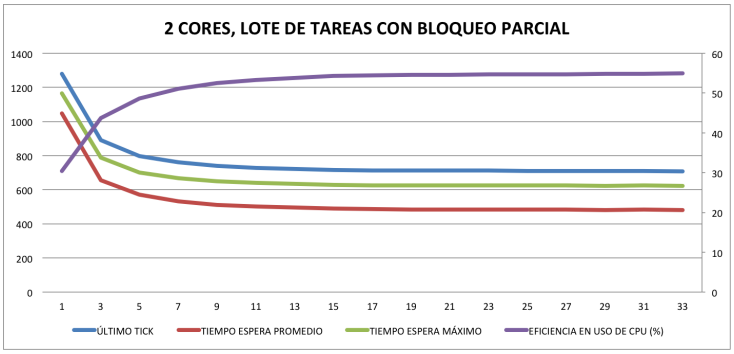
El primer lote que analizamos fue uno con tareas parcialmente bloqueantes:

```
*5 TaskBatch 30 5
@20
*7 TaskBatch 50 10
@40
*2 TaskBatch 70 7
@60
*3 TaskBatch 40 5
```

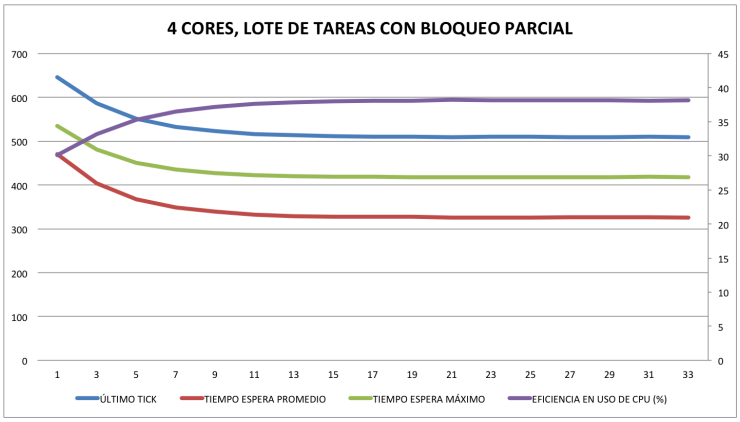
Para un core, observamos que la eficiencia en el uso de CPU incrementa al aumentar la duración del quantum, hasta llegar a un valor en el cual se estabiliza. Los tiempos de espera decrecen hasta estabilizarse asintóticamente al superar el quantum de 15 ciclos de duración.



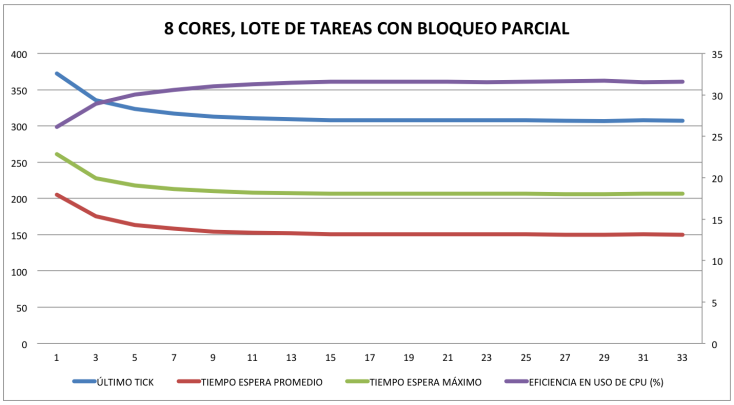
Al aumentar el número de cores, los valores observados fueron sensiblemente distintos: la eficiencia en la utilización del CPU disminuóy, dado que al no variar el lote de tareas existieron cores que no fueron utilizados al final del lote, y momentos en los que algunos cores no tuvieron carga de tareas. Los valores de los tiempos de espera también resultaron inferiores.



Con 4 cores, la eficiencia del uso de CPU bajó a un 40 %, lo cual pareciera indicar que el lote de tareas elegido no fue suficientemente exigente para esta configuración.



Con 6 y 8 cores, las mejorías al aumentar el quantum fueron menos notorias, aunque se mantuvo la tendencia de estabilizarse y a ser asintóticas a partir de un valor de quantum determinado.

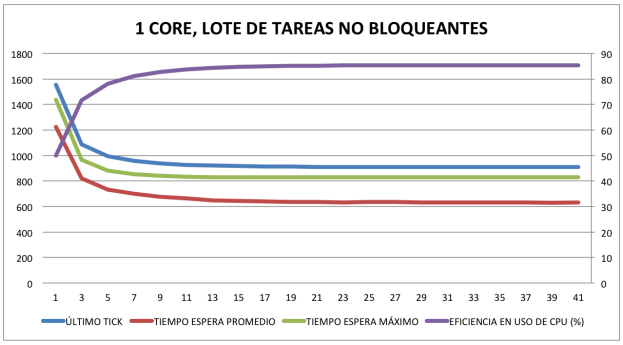


Curiosamente, el valor del quantum en el que se estabilizaron resultó similar para todas las configuraciones, ubicándose en torno a los 15 ciclos de duración.

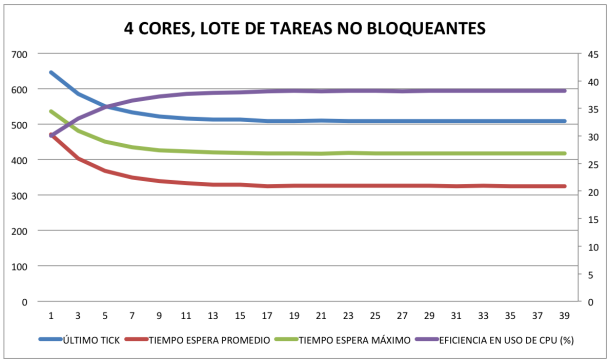
El segundo lote utilizado fue uno con tareas que se bloquean una única vez:

```
*5 TaskBatch 30 1
@20
*7 TaskBatch 50 1
@40
*2 TaskBatch 70 1
@60
*3 TaskBatch 40 1
```

Observamos que este lote se comportó de manera muy similar al anterior.



Esto se repitió para todas las configuraciones de cores.



El lote de tareas con mucho bloqueo arrojó resultados análogos.

Una explicación posible para los resultados similares entre los distintos lotes es que si bien las configuraciones de las tareas cambió, mantuvimos la configuración de los tiempos de arribo y duración de las tareas en el afán de obtener resultados comparables. Como las tareas de tipo *taskBatch* se mantienen bloqueadas únicamente por un ciclo (por definición del ejercicio 6), las mismas vuelven a la cola de *ready* casi inmediatamente, por lo que el hecho de bloquearse no afecta tanto como podría suponerse. A su vez, estas tareas se bloquean seguido, por lo que las aquellas que se bloquearon antes recuperan rápidamente el uso del CPU.

Lo que observamos en los casos anteriores muestra cómo un quantum mayor aumenta el rendimiento de CPU. Esto es así porque proporcionalmente estará más tiempo en ejecución que realizando cambios de contexto entre tareas. El tiempo de espera disminuye ya que el lote se encontraba diseñado de manera pareja, y las tareas eran de duraciones similares. De esta forma, el mayor quantum favorece a que algunas tareas finalicen antes, eliminando la necesidad de esperar otra vuelta hasta su próximo turno.

Podrían existir casos en los que esto no funcione de igual manera. Si tomamos un lote de tareas donde ingresan primero algunas de muy larga duración, para luego llegar muchas tareas cortas, encontraremos que al aumentar el quantum estaremos favoreciendo a ls tareas que llegaron primero. Sin embargo, el tiempo de espera de las tareas cortas podría empeorar. Consideremos el siguiente lote:

```
*3 TaskBatch 200 0
@3
*7 TaskBath 5 0
@50
*5 TaskBatch 30 0
```

2.8. Ejercicio 8

El algoritmo de Round Robin implementado en el ejercicio 3 no efectúa ningún tipo de restricciones con respecto a los cores utilizados por los distintos procesos. Es decir, cuando una determinada tarea se encuentra en ready, podrá ser ejecutada en el primer procesador disponible. Dado que migrar un proceso a otro core conlleva un costo, podría resultar útil contar con un algoritmo de Round Robin que limite estos cambios entre procesadores. En este ejercicio implementamos una simulación de Round Robin en la cual no se permite el cambio entre procesadores. Esto significa que a cada tarea se le asigna un procesador determinado al momento del load, y luego toda su ejecución será en ese procesador.

Para implementar el simulador, se requirieron algunas estructuras adicionales, para garantizar que cada tarea se mantenga en el mismo procesador:

- Un mapa de entero a entero, donde se almacena la afinidad de los procesos con los procesadores. La clave es el PID, y a cada uno le corresponde un core determinado. Al PID se le asigna el procesador al momento del load, y se elimina del mapa cuando la tarea finaliza su ejecución.
- Un mapa de entero a entero donde se almacena la cantidad de tareas activas por core. Al momento del load de una tarea, se utiliza para determinar cuál es el core con menor cantidad de tareas activas, y se le asigna este core a la nueva tarea. Este contador se incrementa al momento del load, y se decrementa cuando la tarea finaliza su ejecución.
- Un vector de colas de ready, donde se cuenta con una cola para cada core. Cada tarea es encolada en la cola respectiva a su core, y al momento de buscar una tarea para un core determinado, el mismo sólo lo hará en la cola que tiene asignada.

- Un mapa que almacena el proceso y el tiempo restante que cuenta para correr en cada core.

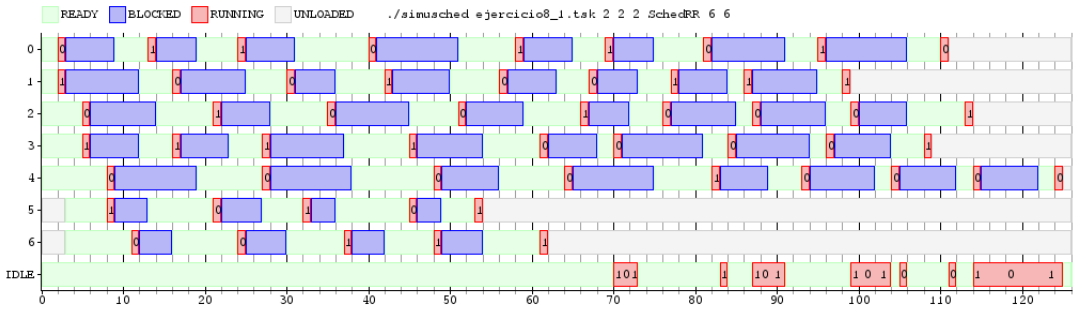
El algoritmo es análogo al utilizado en el ejercicio 3, teniendo en cuenta que cada core utilizará sólo la cola que le corresponde, y que se mantiene un contador de procesos activos por core como se mencionó previamente.

Existen casos en los que restringir el cambio de core produce mejores resultados que permitir que los procesos migren de core. Un ejemplo sería cuando se cuenta con muchas tareas interactivas. Las tareas interactivas se bloquean esperando una respuesta, y cuando se reactivan lo hacen por poco tiempo. En estos casos, el cambio de core produce una penalidad muy grande, en comparación con el tiempo que se está procesando.

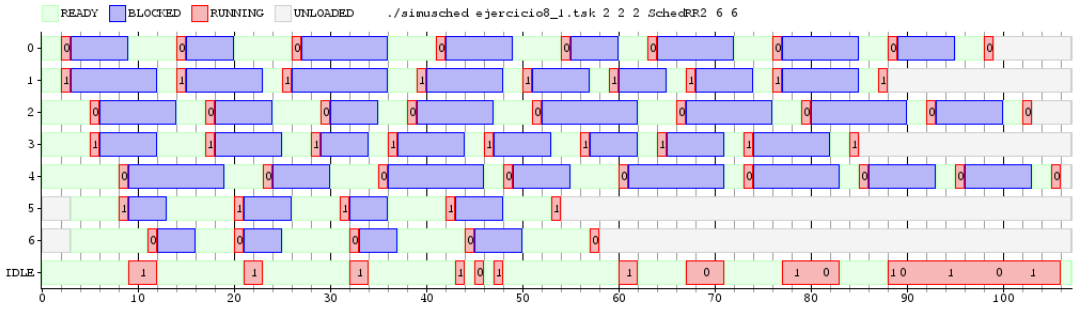
Tomemos el siguiente caso, con 2 cores y un quantum de 6:

```
*5 TaskConsola 8 5 10
@3
*2 TaskConsola 4 3 5
```

Con Round Robin con migración de cores, una corrida produce el siguiente resultado:



La misma corrida sin migración de cores consume unos 20 ciclos menos de CPU para finalizar:

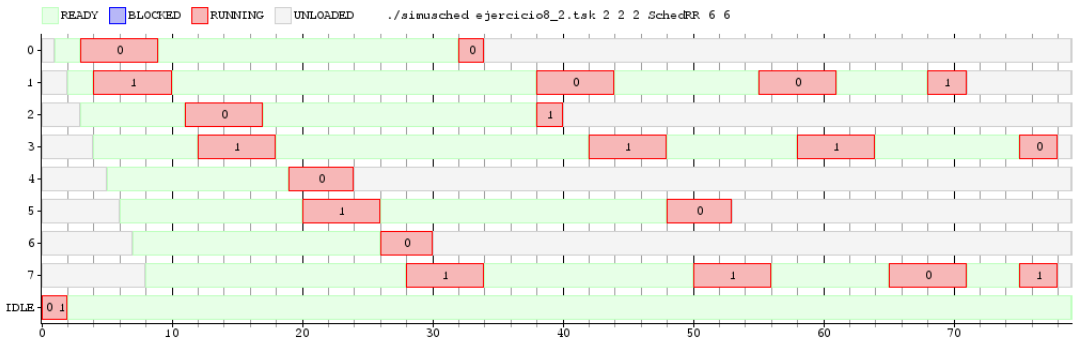


Existen casos en los que, en cambio, es conveniente migrar de cores. Consideremos un lote de tareas que se encuentran intercaladas entre tareas de larga duración y tareas cortas. El algoritmo de Round Robin sin migración de cores no sabe identificar este caso, por lo que quedarán acumuladas las tareas largas en un core, y las cortas en otro. Luego el segundo core quedará ocioso, mientras las otras tareas tienen que compartir el único core que tienen asignado hasta finalizar.

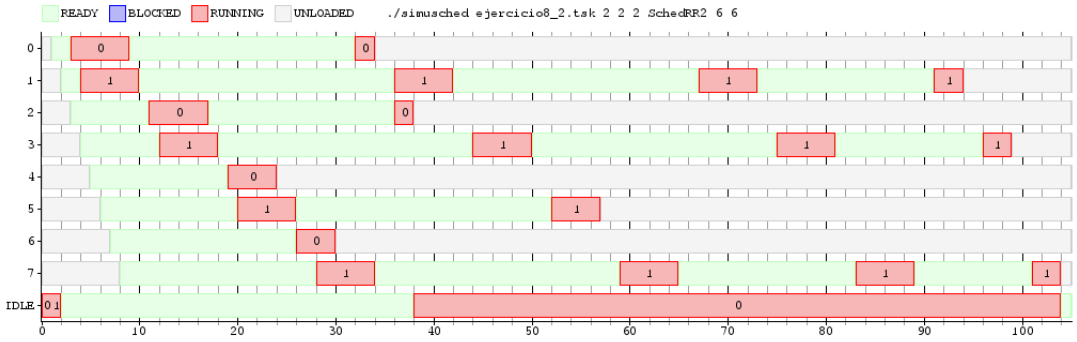
Consideremos el siguiente lote:

```
@1
TaskCPU 7
@2
TaskCPU 20
@3
TaskCPU 7
@4
TaskCPU 20
@5
TaskCPU 4
@6
TaskCPU 10
@7
TaskCPU 3
@8
TaskCPU 20
```

Corrida con Round Robin con migración de cores:



Corrida con Round Robin sin migración de cores:



Notar que a partir del ciclo 38 el core 0 se encuentra ocioso, mientras aín quedan 4 tareas por completar en el core 1.

2.9. Ejercicio 9

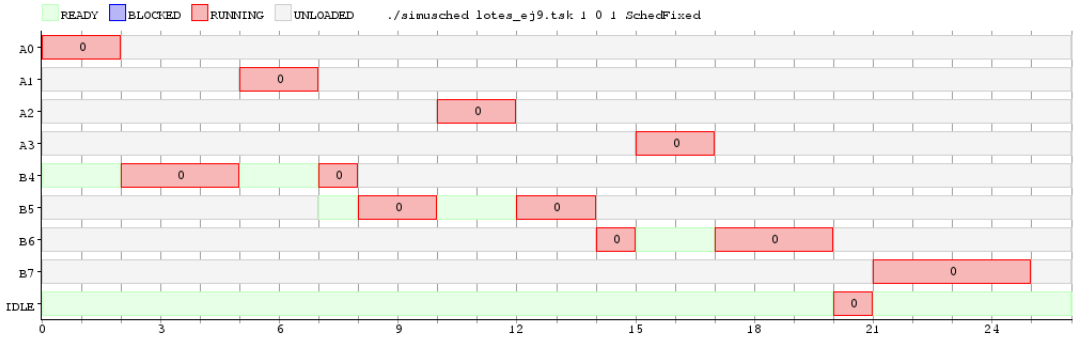
Para mostrar con un ejemplo un caso que no sea factible para el scheduler RM y sí lo sea para EDF nos bastó el siguiente lote de 2 tareas periódicas:

Nota: Para simplicar el ejemplo consideramos en este caso despreciable el tiempo de cambio de contexto.

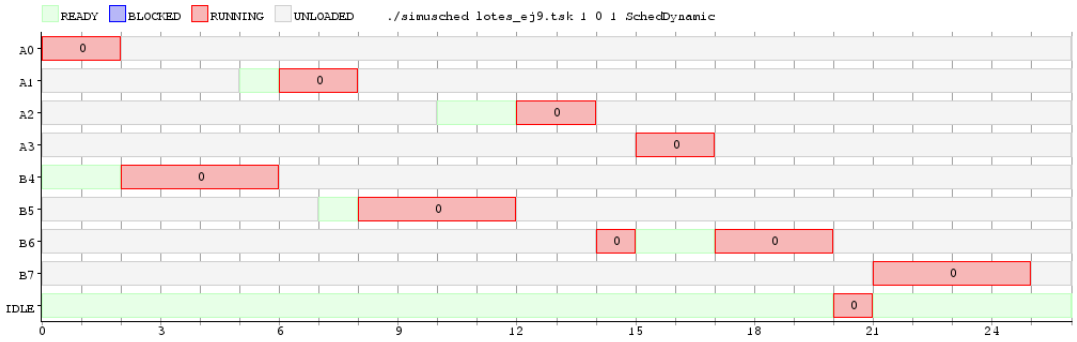
Lote:

```
@0:
&A4,5,1
@0:
&B4,7,3
```

Produciendo los siguientes gráfico:
Para RM:



Para EDF:



La tarea de la familia A tiene un periodo menor que el de la familia B por lo cual según el scheduler RM tiene mayor prioridad asignada. Esta elección de priorizar a la familia A provoca un perjuicio en la familia B por lo que la familia B no puede cumplir su deadline en $T = 7$ (Se produce un overflow).

En el caso del gráfico de EDF se puede observar que le da el CPU un poco más a la familia B por tener el deadline más próximo logrando evitar así el inclumplimiento del deadline (en $T = 5$ deja esperando a la tarea de la familia A).

Algo interesante para destacar es que en este caso ambos scheduler ocuparon el mismo tiempo el CPU con trabajos, lo que cambia es que el scheduler RM a diferencia de EDF tiende a gastar más tiempo de computo en tareas de mayor prioridad dejandolo poco procesamiento a las de menor prioridad.

2.10. Ejercicio 10

En este caso encontramos un ejemplo en el cual las tareas son factibles tanto para RM como para EDF pero con un mejor uso de CPU por parte de EDF.

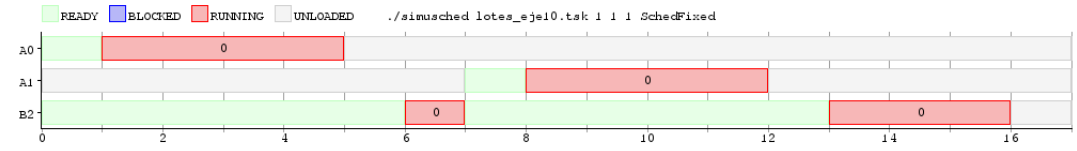
Nota: En este caso asignamos al cambio de contexto un costo 1.

Lote:

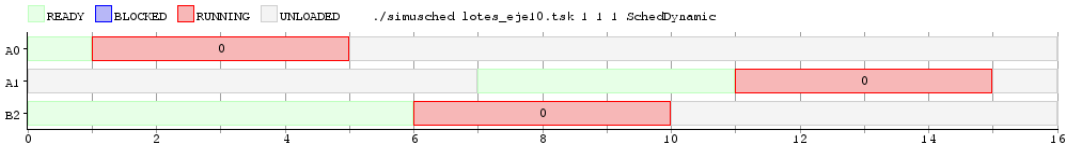
```
@0:
&A2,7,3
@0:
&B1,10,3
```

Produciendo los siguientes gráficos:

Para RM:



Para EDF:



En este caso vemos que el scheduler RM necesita 1 cambio de contexto más que EDF lo que hace que el tiempo neto de procesamiento en tareas sea mayor en EDF (cambios de contexto con RM en T={0, 5, 7, 12} mientras que EDF en T={0, 5, 10}).

Intuitivamente se nos ocurre una conjetura para explicar este resultado, que sería que RM en general necesita más cambios de contexto porque toda tarea de menor prioridad es desalojada por las de mayor prioridad y como estas últimas tienen menor periodo (i.e. mayor frecuencia) la cantidad de desalojos es mayor.

3. Conclusiones

De este trabajo, nos quedo claro que para implementar un scheduler adecuado hay que fijarse en varios aspectos:

- Como son las tareas a planificar (ej si son periodicas, si tienen deadlines, si tienden a bloquearse).
- Como es el hardware en el que van a correr (Si tiene uno o varios nucleos, costo de cambio de contexto, costo de migración de nucleos).
- Analizar el contexto o los requerimientos (Que peso tiene cada una de las diferentes métricas, tiempo de espera, eficicia de CPU, ..., etc).

Referencias

[1] Abraham Silberschatz. *Operating system concepts*. J. Wiley & Sons, Hoboken, NJ, 2009.

[2] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.